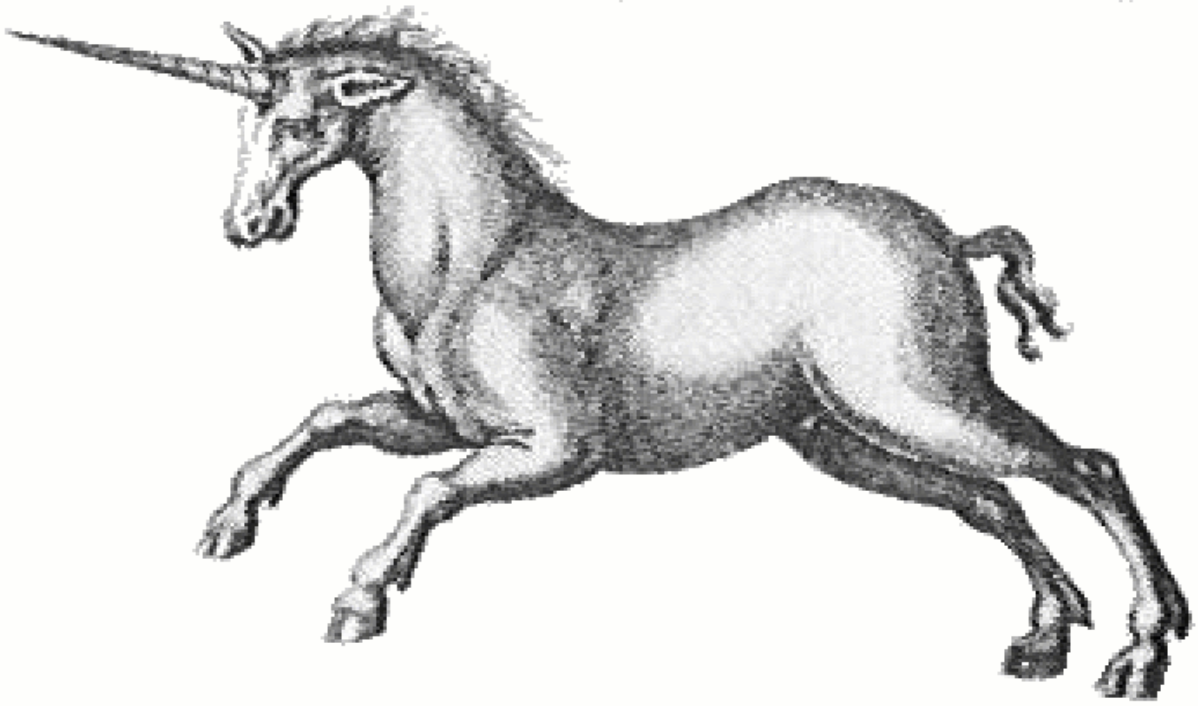
2nd Edition



Programming with Unicon

Very high level object−oriented application and system programming

Clinton Jeffery Shamim Mohamed
Jafar Al Gharaibeh Ray Pereda Robert Parlett

This page intentionally left blank.

Programming with Unicon

2nd edition

Clinton Jeffery
Shamim Mohamed
Jafar Al Gharaibeh
Ray Pereda
Robert Parlett

This is a draft manuscript dated May 6, 2024. Send comments and errata to support@unicon.org.

This document was prepared using LaTeX.

Contents

Preface to the Second Edition vii

I Core Unicon 5

1 Programs and Expressions 7
1.1 Your First Unicon Program . . . 7
1.2 Command Line Options . . . 12
1.3 Expressions and Types . . . 13
1.4 Numeric Computation . . . 14
1.5 Strings and Csets . . . 15
1.6 Goal-directed Evaluation . . . 16
1.7 Fallible Expressions . . . 18
1.8 Generators . . . 18
1.9 Iteration and Control Structures . . . 20
1.10 Procedures . . . 22

2 Structures 29
2.1 Tables . . . 30
2.2 Lists . . . 31
2.3 Records . . . 33
2.4 Sets . . . 34
2.5 Using Structures . . . 34
2.6 Summary . . . 40

3 String Processing 41
3.1 The String and Cset Types . . . 41
3.1.1 String Indexes . . . 41
3.1.2 Character Sets . . . 43
3.1.3 Character Escapes . . . 43
3.2 String Scanning . . . 44
3.3 Pattern Matching . . . 48

3.3.1 Regular Expressions . . . 49
3.3.2 Pattern Composition . . . 49
3.3.3 Pattern Match Operators . . . 51
3.3.4 Scopes of Unevaluated Variables . . . 51
3.4 String Scanning and Pattern Matching Miscellany . . . 51
3.4.1 Grep . . . 51
3.4.2 Grammars . . . 52

4 Advanced Language Features **57**
4.1 Limiting or Negating an Expression . . . 57
4.2 List Structures and Parameter Lists . . . 58
4.3 Co-expressions . . . 59
4.4 User-Defined Control Structures . . . 60
4.5 Parallel Evaluation . . . 61
4.6 Coroutines . . . 62
4.7 Permutations . . . 63
4.8 Simulation . . . 65
4.9 Arrays . . . 68

5 The System Interface **69**
5.1 The Role of the System Interface . . . 69
5.2 Files and Directories . . . 70
5.3 Programs and Process Control . . . 73
5.4 Networking . . . 77
5.5 Messaging Facilities . . . 82
5.6 Tasks . . . 84
5.7 Summary . . . 90

6 Databases **91**
6.1 Language Support for Databases . . . 91
6.2 Memory-based Databases . . . 92
6.3 DBM Databases . . . 93
6.4 SQL Databases . . . 94
6.5 Tips and Tricks for SQL Database Applications . . . 100
6.6 Summary . . . 102

7 Graphics **103**
7.1 2D Graphics Basics . . . 103
7.2 Graphics Contexts . . . 106
7.3 Events . . . 108
7.4 Colors and Fonts . . . 110

7.5 Images, Palettes, and Patterns . . . 111
7.6 3D Graphics . . . 116
7.7 Textures . . . 122
7.8 Summary . . . 134

8 Threads **135**
8.1 Threads and Co-Expressions . . . 136
8.2 First Look at Unicon Threads . . . 136
8.3 Thread Safety . . . 140
8.4 Thread Synchronization . . . 142
8.5 Thread Communication . . . 153
8.6 Practical examples using threads and messages . . . 162
8.6.1 Disk space usage . . . 166
8.6.2 More suggestions for parallel processing . . . 168
8.7 Summary . . . 169

9 Execution Monitoring **171**
9.1 Monitor Architecture . . . 171
9.2 Obtaining Events Using evinit . . . 179
9.3 Instrumentation in the Icon Interpreter . . . 181
9.4 Artificial Events . . . 183
9.5 Monitoring Techniques . . . 184
9.6 Some Useful Library Procedures . . . 186
9.7 Conclusions . . . 186

II Object-oriented Software Development **187**

10 Objects and Classes **189**
10.1 Objects in Programming Languages . . . 189
10.2 Objects in Program Design . . . 192
10.3 Classes and Class Diagrams . . . 193
10.4 Declaring Classes . . . 195
10.5 Object Instances and Initially Sections . . . 196
10.6 Object Invocation . . . 198
10.7 Comparing Records and Classes . . . 199
10.8 Summary . . . 201

11 Inheritance and Associations **203**
11.1 Inheritance . . . 203
11.2 Associations . . . 215

11.3 Aggregation . . . 215
11.4 User-defined associations . . . 216
11.5 Summary . . . 219

12 Writing Large Programs **221**
12.1 Abstract Classes . . . 221
12.2 Design Patterns . . . 223
12.3 Packages . . . 228
12.4 HTML documentation . . . 232
12.5 Summary . . . 232

13 Use Cases and Supplemental UML Diagrams **235**
13.1 Use Cases . . . 236
13.2 Statechart Diagrams . . . 239
13.3 Collaboration Diagrams . . . 241
13.4 Summary . . . 242

III Example Applications **243**

14 CGI Scripts **245**
14.1 Introduction to CGI . . . 245
14.2 The CGI Execution Environment . . . 248
14.3 An Example HTML Form . . . 249
14.4 An Example CGI Script: Echoing the User's Input . . . 251
14.5 Debugging CGI Programs . . . 252
14.6 Appform: An Online Scholarship Application . . . 252

15 System and Administration Tools **255**
15.1 Searching for Files . . . 255
15.2 Finding Duplicate Files . . . 257
15.3 User File Quotas . . . 263
15.4 Capturing a Shell Command Session . . . 268
15.5 Filesystem Backups . . . 270
15.6 Filtering Email . . . 274
15.7 Summary . . . 279

16 Internet Programs **281**
16.1 The Client-Server Model . . . 281
16.2 An Internet Scorecard Server . . . 282
16.3 A Simple "Talk" Program . . . 285
16.4 Summary . . . 291

17 Genetic Algorithms **293**
17.1 What are Genetic Algorithms? . . . 293
17.2 Operations: Fitness, Crossover, and Mutation . . . 294
17.3 The GA Process . . . 297
17.4 ga_eng: a Genetic Algorithm Engine . . . 298
17.5 Color Breeder: a GA Application . . . 303
17.6 Picking Colors for Text Displays . . . 305

18 Object-oriented User Interfaces **307**
18.1 A Simple Dialog Example . . . 307
18.2 A More Complex Dialog Example . . . 310
18.3 Containers . . . 317
18.4 Menu Structures . . . 319
18.5 Other Components . . . 322
18.5.1 Trees . . . 323
18.5.2 Borders . . . 327
18.5.3 Images and icons . . . 327
18.5.4 Scroll bars . . . 328
18.5.5 Custom Components . . . 328
18.5.6 Tickers . . . 338
18.6 Advanced List Handling . . . 343
18.6.1 Selection . . . 343
18.6.2 Popups . . . 344
18.6.3 Drag and drop . . . 344
18.7 Programming Techniques . . . 351
18.8 ivib . . . 353
18.9 Summary . . . 360

IV Appendices **361**

A Language Reference **363**
A.1 Immutable Types: Numbers, Strings, Csets, Patterns . . . 363
A.2 Mutable Types: Containers and Files . . . 366
A.3 Variables . . . 367
A.4 Keywords . . . 368
A.5 Control Structures and Reserved Words . . . 374
A.6 Operators and Built-in Functions . . . 378
A.7 Preprocessor . . . 411
A.8 Execution Errors . . . 413
A.9 Syntax . . . 420

B The Icon Program Library **429**
B.1 Procedure Library Modules 430
B.2 Application Programs, Examples, and Tools 471
B.3 Selected IPL Authors and Contributors 491

C The Unicon Component Library **493**
C.1 GUI Classes 493

D Differences between Icon and Unicon **507**
D.1 Extensions to Functions and Operators 507
D.2 Objects 507
D.3 System Interface 507
D.4 Database Facilities 508
D.5 Multiple Programs and Execution Monitoring Support 508

E Portability Considerations **509**
E.1 POSIX extensions 509
E.2 Microsoft Windows 514

F Installation **517**

G Experimental Features **519**
G.1 User defined operators 520
G.2 Extensions to &random 520
G.3 Plugins 520
G.3.1 Bitman 520
G.3.2 SecureHash 522
G.3.3 SQLite 525

Bibliography **533**

Preface to the Second Edition

This book will raise your level of skill at computer programming, regardless of whether you are presently a novice or expert. The field of programming languages is still in its infancy, and dramatic advances will be made every decade or two until mankind has had enough time to think about the problems and principles that go into this exciting area of computing. The Unicon language described in this book is such an advance, incorporating many elegant ideas not yet found in most contemporary languages.

Unicon is an object-oriented, goal-directed programming language based on the Icon programming language. Unicon can be pronounced however you wish; we pronounce it variably depending on mood, whim, or situation; the most frequent pronunciation rhymes with "lexicon".

For Icon programmers this work serves as a "companion book" that documents material such as the Icon Program Library, a valuable resource that is underutilized. Don't be surprised by language changes: the book presents many new facilities that were added to Icon to make Unicon and gives examples from new application areas to which Unicon is well suited. For people new to Icon and Unicon, this book is an exciting guide to a powerful language.

It is with sweet irony that we call this book the 2nd Edition, since the first edition was never formally published but instead existed solely as an online document, although laser-printed hard copies could be requested. A lot has happened to Unicon since the first edition of this book, which culminated in 2004. This "2nd Edition" catches readers up with things like concurrent threads and vastly improved 3D graphics facilities. Along the way, the games chapter and parts of the internet programming chapter got spun off into a separate work, the so-called *Manual of Puissant Skill at Game Programming*.

Organization of This Book

This book consists of four parts. The first part, Chapters 1-8, presents the core of the Unicon language, much of which comes from Icon. These early chapters start with simple expressions, progress through data structures and string processing, and include advanced programming topics and the input/output capabilities of Unicon's portable system interface. Part two, in Chapters 9-12, describes object-oriented development as a whole and

presents Unicon's object-oriented facilities in the context of object-oriented design. Object-oriented programming in Unicon corresponds closely to object-oriented design diagrams in the Unified Modeling Language, UML. Some of the most interesting parts of the book are in part three; Chapters 13-18 provide example programs that use Unicon in a wide range of application areas. Part four consists of essential reference material presented in several Appendixes.

Acknowledgments

Thanks to the Icon Project for creating a most excellent language. Thanks especially to those unsung heroes, the university students and Internet volunteers who implemented the language and its program library over a period of many years. Icon contributors can be divided into epochs. In the epoch leading up to the first edition of this book, we were inspired by contributions from Gregg Townsend, Darren Merrill, Mary Cameron, Jon Lipp, Anthony Jones, Richard Hatch, Federico Balbi, Todd Proebsting, Steve Lumos and Naomi Martinez. In the epoch since the first edition of this book, the Unicon Project owes a debt of gratitude to Ziad al Sharif, Hani bani Salameh, Jafar Al Gharaibeh, Mike Wilder, and Sudarshan Gaikaiwari.

The most impressive contributors are those whose influence on Icon has spanned across epochs, such as Ralph Griswold, Steve Wampler, Bob Alexander, Ken Walker, Phillip Thomas, and Kostas Oikonomou. We revere you folks! Steve Wampler deserves extra thanks for serving as the technical reviewer for the first edition of this book. Phillip Thomas and Kostas Oikonomou have provided extensive support and assistance that goes way beyond the call of duty; in many ways this is their book.

This manuscript received critical improvements and corrections from many additional technical reviewers, including, David A. Gamey, Craig S. Kaplan, David Feustel, David Slate, Frank Lhota, Art Eschenlauer, Wendell Turner, Dennis Darland, and Nolan Clayton.

The authors wish to acknowledge generous support from the National Library of Medicine and AT&T Bell Labs Research. This work was also supported in part by the National Science Foundation under grants CDA-9633299, EIA-0220590 and EIA-9810732, and the Alliance for Minority Participation.

Clinton Jeffery
Shamim Mohamed
Jafar al Gharaibeh
Ray Pereda
Robert Parlett

Introduction

Software development requires thinking about several dimensions simultaneously. For large programs, writing the actual computer instructions is not as difficult as figuring out the details of what the computer is supposed to do. After analyzing what is needed, program design brings together the data structures, algorithms, objects, and interactions that accomplish the required tasks. Despite the importance of analysis and design, programming is still the central act of software development for several reasons. The weak form of the Sapir-Whorf hypothesis suggests that the programming language we use steers and guides the way we think about software, so it affects our designs. Software designs are mathematical theorems, while programs are proofs that test those designs. As in other branches of mathematics, the proofs reign supreme. In addition, a correct design can be foiled by an inferior implementation.

This book is a guide and reference for an exciting programming language called Unicon that has something to offer both computer scientists as well as casual programmers. You will find explanations of fundamental principles, unique language idioms, and advanced concepts and examples. Unicon exists within the broader context of software development, so the book also covers software engineering fundamentals. Writing a correct, working program is the central task of software engineering. This does not happen automatically as a result of the software design process. Make no mistake: if you program very much, the programming language you use is of vital importance. If it weren't, we would still be programming in machine language.

Prototyping and the Spiral Model of Development

A software prototype is a working subset of a software system. Prototypes help check software designs and user interfaces, demonstrate key features to customers, or prove the feasibility of a proposed solution. A prototype may generate customer feedback on missing functionality, provide insight on how to improve the design, lead to a decision about whether to go ahead with a project or not, or form a starting point for the algorithms and data structures that will go into the final product. Prototyping is done early in the software development process. It fits naturally into the *spiral model* of development proposed by Barry Boehm (1988). Figure I-1 shows the spiral model; time is measured by the distance

from the center. Analysis, design, coding, and evaluation are repeated to produce a better product with each iteration. "Prototyping" is the act of coding during those iterations when the software is not yet fully specified or the program does not yet remotely implement the required functionality.

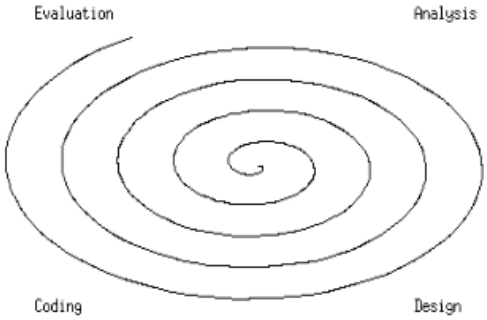


Figure I-1 The Spiral Model of Software Development

Tight spirals are better than loose spirals. The more powerful the prototyping tools, the less time and money spent in early iterations of development. This translates into either faster time to market, or a higher quality product. Some prototypes are thrown away once they have served the purpose of clarifying requirements or demonstrating some technique. This is OK, but in the spiral model some prototypes are gradually enhanced until they become the final production system.

Icon: a Very High Level Language for Applications

Icon is a programming language developed at the University of Arizona. Icon generalizes its developers' experience creating an earlier language, SNOBOL4. Icon embodies seminal research ideas, but it is also more fun and easier to program than other languages. Most very high-level languages revel in cryptic syntax, while Icon is not just more powerful, but often more *readable* than its competitors. This gain in expressive power without losing readability is an addicting result of Icon's elegant design.

The current Arizona Icon, version 9.5, is described in *The Icon Programming Language, 3rd edition* by Ralph and Madge Griswold (1996). Its reference implementation is a virtual machine interpreter. Icon evolved through many releases over two decades and is far more capable than it was originally. It is apparently a finished work.

Enter Unicon: More Icon than Icon

The name "Unicon" refers to the descendant of Icon described in this book and distributed from www.unicon.org. Unicon is Icon with portable, platform-independent access to hardware

and software features that have become ubiquitous in modern applications development, such as objects, networks, and databases. Unicon is created from the same public domain source code that Arizona Icon uses, so it has a high degree of compatibility. We were not free to call it version 10 of the Icon language, since it was not produced or endorsed by the Icon Project at the University of Arizona.

Just as the name Unicon frees the Icon Project of all responsibility for our efforts, it frees us from the requirement of backward compatibility. While Unicon is almost entirely backward compatible with Icon, dropping full compatibility allows us to clear out some dead wood and more importantly, to make some improvements in the operators that will benefit everyone at the expense of...no one but the compatibility police. This book covers the features of Icon and Unicon together. A compatibility check list and description of the differences between Icon and Unicon are given in Appendix D.

The Programming Languages Food Chain

It is interesting to compare Icon and Unicon with the competition. Mainstream programming languages such as C, C++, and Java, like the assembler languages that were mainstream before them, are ideal tools for writing all sorts of programs, so long as vast amounts of programmer time are available. Throwing more programmers at a big project does not work well, and programmers are getting more expensive while computing resources continue to become cheaper. These pressures inexorably lead to the use of higher-level languages and the development of better design and development methods. Such human changes are incredibly slow compared to technological changes, but they are visibly occurring nevertheless. Today, the most productive programmers are using extra CPU cycles and memory to reduce the time it takes to develop useful programs.

There is a subcategory of mainstream languages, marketed as *rapid application development* languages, whose stated goals seem to address this phenomenon. Languages such as Visual Basic or PowerBuilder provide graphical interface builders and integrated database connectivity, giving productivity increases in the domain of data entry and presentation. The value added in these products are in their programming environments, not their languages. The integrated development environments and tools provided with these languages are to be acclaimed and emulated, but they do not provide productivity gains that are equally relevant to all application domains. They are only a partial solution to the needs of complex applications.

Icon is designed to be easier and faster to program than mainstream languages. The value it adds is in the expressive power of the language itself, in the category of very high level languages that includes Lisp, APL, Smalltalk, REXX, Perl, Tcl, Python, and Ruby; there are many others. Very high-level languages can be subdivided into scripting languages and applications languages. Scripting languages often glue programs together from disparate sources. They are typically strong in areas such as multilingual interfacing and file

system interactions, while suffering from weaker expression semantics, typing, scope rules, and control structures than their applications-oriented cousins. Applications languages typically originate within a particular application domain and support that domain with special syntax, control structures, and data types. Since scripting *is* an application domain, scripting languages are just one prominent subcategory of very high-level languages.

Icon is an applications language with roots in text processing and linguistics. Icon programs tend to be more readable than similar programs written in other very high-level languages, making Icon well-suited to the aims of *literate programming*. For example, Icon was used to implement Norman Ramsey's literate programming tool noweb (Ramsey, 1994). Literate programming is the practice of writing programs and their supporting textual description together in a single document.

Unicon makes the core contributions of Icon useful for a broader range of applications. This book's many examples illustrate the range of tasks for which Unicon is well suited, and these examples are the evidence in support of Unicon's existence. Consider using Unicon when one or more of the following conditions are true. The more conditions that are true, the more you will benefit from Unicon.

- Programmer time must be minimized.
- Maintainable, concise source code is desired.
- The program includes complex data structures or experimental algorithms.
- The program involves a mixture of text processing and analysis, custom graphics, data manipulation, network or file system operations.
- The program must run on several operating systems and have a nearly identical graphical user interface with little or no source code differences.

Unicon is not the last word in programming. You probably should not use Unicon if your program has one or more of the following requirements:

- The fastest possible performance is needed.
- The program has hard real-time constraints.
- The program must perform low-level or platform-specific interactions with the hardware or operating system.

Programming languages play a key role in software development. The Unicon language is a very high level object-oriented language with a unique combination of expressive power and scalable rapid development. In this book, many examples from a wide range of application areas demonstrate how to apply and combine Unicon's language constructs to solve real-world problems. It is time to move past the introductions. Prepare to be spoiled by this language. You may have the same feelings that Europeans felt when they gave up using Roman numerals and switched to the Hindu-Arabic number system. "This multiplication stuff isn't that hard anymore!"

Part I

Core Unicon

Chapter 1

Programs and Expressions

This chapter presents many of the key features of Unicon, starting with those it has in common with other popular languages. Detailed instructions show how to compile and run programs. Soon the examples introduce important ways in which Unicon is different from other languages. These differences are more than skin deep. If you dig deeply, you can find dozens of details where Unicon provides just the right blend of simplicity, flexibility, and power. After this chapter, you will know how to

- edit, compile, and execute Unicon programs
- use the basic types to perform calculations
- identify expressions that can fail, or produce multiple results
- control the flow of execution using conditionals, looping, and procedures

1.1 Your First Unicon Program

This section presents the nuts and bolts of writing and running an Unicon program, after which you will be able to try the code examples or write your own programs. Before you can run the examples here or in any subsequent chapter, you must install Unicon on your system. (See Appendix F for details on downloading and installing Unicon from the Unicon web site, http://unicon.org.) We are going to be very explicit here, and assume nothing about your background. If you are an experienced programmer, you will want to skim this section, and move on to the next section. If you are completely new to programming, have no fear. Unicon is pretty easy to learn.

All programs consist of commands that use hardware to obtain or present information to users, and perform calculations that transform information into a more useful form. To program a computer you write a document containing instructions for the computer to carry out. In Unicon a list of instructions is called a *procedure*, and a *program* is a collection of one or more procedures. In larger programs, groups of related procedures are organized

into classes or packages; these features are presented in Part II of this book. Unicon programs are text files that may be composed using any text editor. For the purposes of demonstration this section describes how to use Ui, the program editor and integrated development tool that comes with Unicon.

It is time to begin. Fire up Ui by typing "ui" from the command line, or launching the menu item or icon labeled "Unicon," and type:

```
procedure main()
    write("Hello, amigo!")
end
```

Your screen should look something like Figure 1-1. The large upper area of the window is the editing region where you type your program code. The lower area of the window is a status region in which the Ui program displays a message when a command completes successfully, or when your program has an error. Until you explicitly name your file something else, a new file has the name noname.icn. The font Ui uses to display source code is selectable from the Options menu.

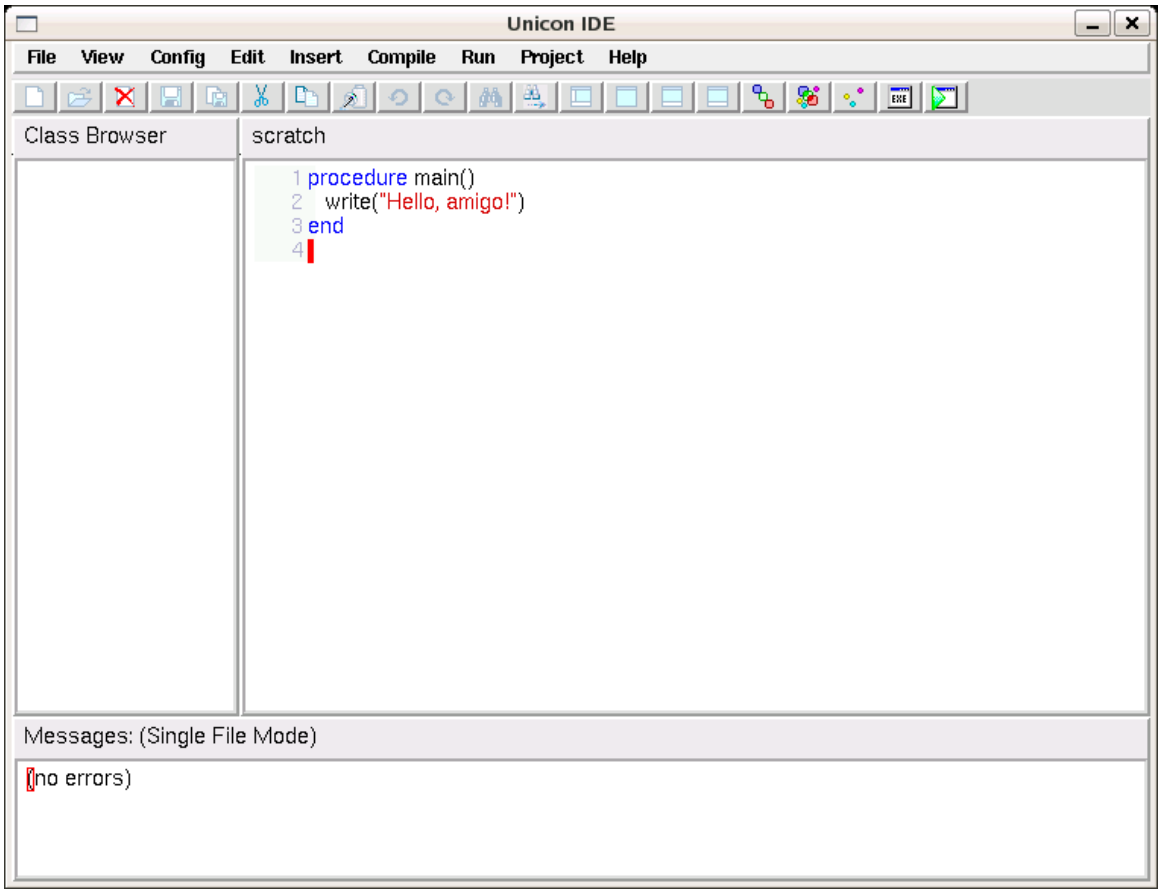


Figure 1-1: Writing an Unicon program using the Ui program.

The list of instructions that form a *procedure* begins with the word procedure and ends with the word end. Procedures have names. After writing a list of instructions in a procedure you may refer to it by name without writing out the list again. The write() instruction is just such a procedure, only it is already written for you; it is built in to the language. When you issue a write() instruction, you tell the computer what to write. The details a procedure uses in carrying out its instructions are given inside the parentheses following that procedure's name; in this case, "Hello, amigo!" is to be written. When you see parentheses after a name in the middle of a list of instructions, it is an instruction to go execute that procedure's instructions. Inside the parentheses there may be zero, one, or many values supplied to that procedure.

Besides writing your program, there are a lot of menu commands that you can use to control the details of compiling and executing your program within Ui. For instance, if you select **Run→Run**, Ui will do the following things for you.

1. Save the program in a file on disk. All Unicon programs end in .icn; you could name it anything you wished, using the **File→SaveAs** command.

2. Compile the Unicon program from human-readable text to (virtual) machine language. To do this step manually, you can select the **Compile→Make executable** command.

3. Execute the program. This is the main purpose of the **Run** command. Ui performed the other steps in order to make this operation possible.

If you type the hello.icn file correctly, the computer should chug and grind its teeth for awhile, and

```
Hello, amigo!
```

should appear in a window on your screen. This ought to be pretty intuitive, since the instructions included the line

```
write("Hello, amigo!")
```

in it. That's how to write to the screen. It's that simple.

The first procedure to be executed when a program runs is called main(). Every instruction listed in the procedure named main() is executed in order, from top to bottom, after which the program terminates. Use the editor to add the following lines right after the line write("Hello, amigo") in the previous program:

```
write("How are you?")
write(7 + 12)
```

The end result after making your changes should look like this:

```
procedure main()
    write("Hello, amigo!")
    write("How are you?")
    write(7 + 12)
end
```

Run the program again. This example shows you what a list of instructions looks like, as well as how easy it is to tell the computer to do some arithmetic.

Note

It would be fine (but not very useful) to tell the computer to add 7 and 12 without telling it to write the resulting value. On seeing the instruction

```
7 + 12
```

the computer would do the addition, throw the 19 away, and go on.

Add the following line, and run it:

```
write("7 + 12")
```

This illustrates what quotes are for. Quoted text is taken literally; without quotes, the computer tries to simplify (do some arithmetic, or compute the value of what is written), which might be difficult if the material in question is not an expression!

```
write(hey you)
```

makes no sense and is an error. Add this line, and run it:

```
write(7 + "12")
```

The 12 in quotes is taken literally as some text, but that text happens to be digits that comprise a number, so adding it to another number makes perfect sense. The computer will not have as much success if you ask it to add 7 to "amigo". The computer views all of this in terms of values. A *value* is a unit of information, such as a number. Anything enclosed in quotes is a single value. The procedure named write() prints values on your screen. *Operators* such as + take values and combine them to produce other values, if it is possible to do so. The values you give to + had better be numbers! If you try to add something that doesn't make sense, the program will stop running at that point, and print an error message.

By now you must have the impression that writing things on your screen is pretty easy. Reading the keyboard is just as easy, as illustrated by the following program:

```
procedure main()
    write("Type a line ending with <ENTER>:")
    write("The line you typed was" , read())
end
```

Run the program to see what it does. The procedure named read() is used to get what the user types. It is built in to the language. The read() instruction needs no directions to do its business, so nothing is inside the parentheses. When the program is run, read() grabs a line from the keyboard, turns it into a value, and produces that value for use in the program, in this case for the enclosing write() instruction.

The write() instruction is happy to print out more than one value on a line, separated by commas. When you run the program, the "the line you typed was" part does not get printed until after you type the line for read() and that instruction completes. The write() instruction must have all of its directions (the values inside the parentheses) before it can go about its business.

Now let's try some variations. Can you guess what the following line will print?

```
write("this line says " , "read()")
```

The read() procedure is never executed because it is quoted! Quotes say "take these letters literally, not as an equation or instruction to evaluate." How about:

```
write("this line says , read()")
```

Here the quotes enclose one big value, which is printed, comma and all. The directions one gives to a procedure are *parameters*; when you give a procedure more than one parameter, separated by commas, you are giving it a *parameter list*. For example,

```
write("this value ", "and this one")
```

Compile and run the following strange-looking program. What do you think it does?

```
procedure main()
    while write( "" ~== read() )
end
```

This program copies the lines you type until you type an empty line by pressing Enter without typing any characters first. The "" are used just as usual. They direct the program to take whatever is quoted literally, and this time it means literally nothing - an empty line. The operator ~== stands for "not equals". It compares the value on its left to the value on its right, and if they are not equal, it produces the value on the right side; if they are equal, it *fails* - that is, the "not equals" operator produces no value. If you have programmed in other languages, this may seem like a strange way to describe what is usually performed with nice simple Boolean values True and False. For now, try to take this description at face value; Unicon has no Boolean type or integer equivalent, it uses a more powerful concept that we will examine more fully in the chapters that follow.

Thus, the whole expression "" ~== read() takes a line from the keyboard, and if it is not empty, it produces that value for the enclosing write() instruction. When you type an empty

line, the value read() produces is equal to "", and ~== produces no value for the enclosing write() instruction, which similarly fails when given no value. The while instruction is a "loop" that repeats the instruction that follows it until that instruction fails (in this case, until there is no more input). There are other kinds of loops, as well as another way to use while; they are all described later in this chapter.

So far we've painted you a picture of the Unicon language in very broad strokes, and informally introduced several relevant programming concepts along the way. These concepts are presented more thoroughly and in their proper contexts in the next sections and subsequent chapters. Hopefully you are already on your way to becoming an Icon programmer *extraordinaire*. Now it is time to dive into many of the nuts and bolts that make programming in Unicon a unique experience.

1.2 Command Line Options

Unicon comes with an IDE, but you can edit programs with any editor, and compile and run them from your operating system's command line. This section describes the Unicon command line tools along with several useful options. The Unicon compiler executable is named unicon, and to compile the program foo.icn you would type

```
unicon foo
```

To execute the resulting program, just type

```
foo
```

To compile and link a program consisting of several modules, you can type them all on the command line, as in

```
unicon foo bar baz
```

but often you will want to compile them separately (using the -c command line option) and link the resulting object files, called ucode files; their extension is .u

```
unicon -c foo
unicon -c bar
unicon -c baz
unicon foo.u bar.u baz.u
```

Some of the other useful command line options include:

- -o arg name the resulting output file arg
- -x args execute the program immediately after linking; this option goes *after* the program filenames

- -t turn on tracing
- -u produce a warning for undeclared variables
- -E output preprocessed source code
- -C compile to C code and link

These options can be specified in the Ui program under the Compile menu's Compile Options command. Other options exist; consult your Unicon and Icon manual pages and platform-specific help files and release information for more details.

1.3 Expressions and Types

Each procedure in a Unicon program is a sequence of *expressions.* Expressions are instructions for obtaining values of some type, such as a number or a word; some expressions also cause side effects, such as sending data to a hardware device. Simple expressions just read or write a value stored in memory. More interesting expressions specify a computation that manipulates zero or more *argument* values to obtain *result* values by some combination of operators, procedures, or control structures.

The simplest expressions are *literals*, such as 2 or "hello, world!". These expressions directly specify a value stored in memory. When the program runs, they do not do any computation, but rather evaluate to themselves. Literals are combined with other values to produce interesting results. Each literal has a corresponding *type*. This chapter focuses on the atomic types. Atomic types represent individual, immutable values. The atomic types in Unicon are integer and real (floating-point) numbers, string (a sequence of characters), and cset (a character set). Atomic types are distinguished by the fact that they have literal values, specified directly in the program code and represented as data in the compiled code. Values of other types such as lists are constructed during execution. Later chapters describe structure types that organize collections of values, and system types for interacting with the operating system via files, databases, windows, and network connections.

After literals, references to *variables* are the next simplest form of expression. Variables are named memory locations that hold values for use in subsequent expressions. You refer to a variable by its name, which must start with a letter or underscore and may contain any number of letters, underscores, or numbers. Use names that make the meaning of the program clear. The values stored in variables are manipulated by using variable names in expressions like i+j. This expression results in a value that is the sum of the values in the variables i and j, just like you would expect.

Some words may not be used as variable names because they have a special meaning in the language. These *reserved words* include procedure, end, while, and so on. Other special variables called *keywords* start with the ampersand character (&) and denote special values.

For example, global variables are initialized to the null value represented by the keyword &null. Other keywords include &date, &time, and so on. Complete lists of reserved words and keywords are given in Appendix A.

Unlike many languages where you have to state up front (*declare*) all the variables you are going to use and specify their data type, in Unicon variables do not have to be declared at all, and any variable can hold any type of value. However, Unicon will not allow you to mix incompatible types in an expression. Unicon is *type safe*, meaning that every operator checks its argument values to make sure they are compatible, converts them if necessary, and halts execution if they cannot be converted.

1.4 Numeric Computation

Unicon supports the usual arithmetic operators on data types integer and real. Integers are signed whole numbers of arbitrary magnitude — Unicon is not limited to the range of whole numbers provided by the underlying hardware. The real type is a signed floating point decimal number whose size on any platform is the largest size supported by machine instructions, typically 64-bit double precision values. In addition to addition (+), subtraction (-), multiplication (*) and division (/), there are operators for modulo (%) and exponentiation (^). Arithmetic operators require numeric operands.

Note

Operations on integers produce integers; fractions are truncated, so 8/3 produces 2. If either operand is a real, the other is converted to real and the result is real, so 8.0/3 is 2.66666...

As a general rule in Unicon, arguments to numeric operators and functions are automatically converted to numbers if possible, and a run-time error occurs otherwise. The built-in functions integer(x) and real(x) provide an explicit conversion mechanism that fails if x cannot be converted to numeric value, allowing a program to check values without resulting in a run-time error.

In addition to the operators, built-in functions support several common numeric operations. The sqrt(x) function produces the square root of x, and exp(x) raises e to the x power. The value of pi (3.141...) is available in keyword &pi, the Golden Ratio (1.618...) is available in &phi, and e (2.718...) is available in &e. The log(x) function produces the natural log of x. The common trigonometric functions, such as sin() and cos() take their angle arguments in radian units. The min(x1, x2, ...) and max(x1, x2, ...) routines return minimum and maximum values from any number of arguments. Appendix A gives a complete list of built-in functions and operators.

Listing 1-1 shows a simple Unicon program that illustrates the use of variables in a numeric computation. The line at the beginning is a comment for the human reader. Comments begin with the # character and extend to the end of the line on which they appear. The compiler ignores them.

Listing 1-1 Mystery program

```
# What do I compute?
procedure main()
    local i, j, old_r, r
    i := read()
    j := read()
    old_r := r := min(i, j)
    while r > 0 do {
        old_r := r
        if i > j then
            i := r := i % j
        else
            j := r := j % i
        }
    write(old_r)
end
```

This example illustrates *assignment*; values are assigned to (or "stored in") variables with the := operator. As you saw in the previous section, the function read() reads a line from the input and returns its value. The modulo operator (%) is an important part of this program: i % j is the remainder when i is divided by j.

While loops can use a reserved word do followed by an expression (often a compound expression in curly braces). The expression following the do is executed once each time the expression that controls the while succeeds. Inside the while loop, a conditional if-then-else expression is used to select from two possible actions.

The names of the variables in this example are obscure, and there are no comments in it other than the one at the top. Can you guess what this program does, without running it? If you give up, try running it with a few pairs of positive numbers.

In addition to arithmetic operators, there are *augmented assignment* operators. To increment the value in a variable by 2, these two statements are equivalent:

```
i +:= 2
i := i + 2
```

Augmented assignment works for most binary operators, not just arithmetic. The expression i *op*:= expr means the same as i := i *op* expr.

1.5 Strings and Csets

The non-numeric atomic types in Unicon are character sequences (strings) and character sets (csets). Icon came from the domain of string processing, and from it Unicon inherits

many sophisticated features for manipulating strings and performing pattern matching. This section presents the simple and most common operations. More advanced operations and examples using strings and csets are given in Chapter 4.

String literals are enclosed in double quotes, as in "this is a string", while cset literals are enclosed in single quotes, as in 'aeiou'. Although strings and csets are composed of characters, there is no character type; a string (or cset) consisting of a single character is used instead.

Current implementations of Unicon use eight-bit characters, allowing strings and csets to be composed from 256 unique characters. ASCII representation is used for the lower 128 characters, except on EBCDIC systems. The appearance of non-ASCII values is platform dependent. Like integers, strings can be arbitrarily large, constrained only by the amount of memory you have on your system.

Several operators take string arguments. The *s operator gives the length of string s. The expression s1||s2 produces a string consisting of the characters in s1 followed by s2. The subscript operator s[i] produces a one-letter substring of s at the ith position. Indices are counted starting from position 1. If i is nonpositive, it is from the end of the string, for example s[-2] is the second to the last character in the string.

Csets support set operators. c1++c2 produces a cset that is the union of c1 and c2. The expression c1**c2 is the intersection, while c1--c2 is the difference. In addition, several keywords are commonly used csets. The keywords &letters, &lcase, and &ucase denote the alphabetic characters, lower case characters a-z, and upper case characters A-Z, respectively, while &digits is the set from 0-9, &ascii is the lower 128 characters, and &cset is the set of all (256, on most implementations) characters.

Many built-in functions operate on strings and csets. Some of the simple string functions are reverse(x), which produces the reverse of a string (or list) x, and trim(s,c), which produces a substring of s that does not end with any character in cset c.

Functions and operators that require string arguments convert numeric values to strings automatically, and halt execution with a run-time error if given a value that cannot be converted to a string.

1.6 Goal-directed Evaluation

So far, the examples of how expressions are evaluated have included nothing you wouldn't find in ordinary programming languages. It is time to push past the ordinary. In most conventional languages, each expression always computes *exactly one* result. If no valid result is possible, a sentinel value such as -1, NULL, EOF (end-of-file) or INF (infinity) is returned instead. This means that the program must check the return value for this condition. For example, while reading integers from the input and performing some operation on them you might do something like this:

```
while (i := read()) ~= -1 do
```

```
process(i)
```

This will work, of course, except when you really need to use -1 as a value! It is somewhat cumbersome, however, even when a sentinel value is not a problem. Unicon provides a much nicer way to write this type of code, developed originally in the Icon language. In Unicon, expressions are *goal-directed*. This means that every expression when evaluated has a goal of producing results for the surrounding expression. If an expression succeeds in producing a result, the surrounding expression executes as intended, but if an expression cannot produce a result, it is said to fail and the surrounding expression cannot be performed and in turn fails.

Now take a look at that loop again. If it weren't for the termination condition, you would not need the intermediate variable i. If you would like to say:

```
process(read())
```

your wishes are answered by Unicon; you can indeed write your program like this. The expression read() tries to produce a value by reading the input. When it is successful, process() is called with the value; but when read() cannot get any more values, that is, at the end of the file, it *fails*. This failure propagates to the surrounding expression and process() is not called either. Here is the clincher: control expressions like if and while don't check for Boolean (true/false) values, they check for success! So our loop becomes

```
while process(read())
```

The do clause of a while loop is optional; in this case, the condition does everything we need, and no do clause is necessary.

Consider the if statement that was used in the earlier arithmetic example:

```
if i > j then ...
```

Comparison operators such as > succeed or fail depending on the values of the operands. This leads to another question: if an expression like i < 3 succeeds, what value should it produce? No "true" value is needed, because any result other than failure is interpreted as "true." This allows the operator to return a useful value instead! The comparison operators produce the value of their right operand when they succeed. You can write conditions like

```
if 3 < i < 7 then ...
```

that appear routinely in math classes. Other programming languages only dream about being this elegant. First, Unicon computes 3 < i. If that is true, it returns the value i, which is now checked with 7. This expression in fact does exactly what you'd expect. It checks to see that the value of i is between 3 and 7. (Also, notice that if the first comparison fails, the second one will not be evaluated.)

1.7 Fallible Expressions

Because some expressions in Unicon can fail to produce a result, you should learn to recognize such expressions on sight. These *fallible expressions* control the flow of execution through any piece of Unicon code you see. When failure is expected it is elegant. When it is unexpected in your program code, it can be disastrous, causing incorrect output that you may not notice or, if you are lucky, the program may terminate with a run-time error.

Some fallible expressions fail when they cannot perform the required computation; others are *predicates* whose purpose is to fail if a condition is not satisfied. The subscript and sectioning operators are examples of the first category. The expression x[i] is a subscript operator that selects element i out of some string or structure x. It fails if the index i is out of range. Similarly, the sectioning operator x[i:j] fails if either i or j are out of range.

The read() function is illustrative of a large number of built-in functions that can fail. A call to read() fails at the end of a file. You can easily write procedures that behave similarly, failing when they cannot perform the computation that is asked. Unfortunately, for an arbitrary procedure call p(), you can't tell if it is fallible without studying its source code or reference documentation. The safest thing is to expect any procedure call is fallible and check whether it failed, unless you know it is not fallible or its failure doesn't matter. Following this advice may avoid many errors and save you lots of time. In this book we will be careful to point out fallible expressions when we introduce them.

The less than operator < is a typical predicate operator, one that either fails or produces exactly one result. The unary predicates /x and \x test a single operand, succeeding and producing the operand if it is null, or non-null, respectively. The following binary predicates compare two operands. The next section presents some additional, more complex fallible expressions.

< <= > >= = ~=	numeric comparison operators
<< <<= >> >>= == ~==	lexical (alphabetic) comparison
=== ~===	reference comparison

1.8 Generators

So far we have seen that an expression can produce no result (failure) or one result (success). In general, an expression can produce any number of results: 0, 1, or many. Expressions that can produce more than one result are called *generators*. Consider the task of searching for a substring within a string:

```
find("lu", "Honolulu")
```

In most languages, this would return one of the substring matches, usually the first position at which the substring is found. In Unicon, this expression is a generator, and

can produce *all* the positions where the substring occurs. If the surrounding expression only needs one value, as in the case of an if test or an assignment, only the first value of a generator is produced. If a generator is part of a more complex expression, then the return values are produced in sequence until the whole expression produces a value.

Let us look at this example:

```
3 < find("or", "horror")
```

The first value produced by find() is 2, which causes the < operation to fail. Execution then resumes the call to find(), which produces a 5 as its next value, and the expression succeeds. The value of the expression is the first position of the substring greater than 3.

The most obvious generator is the alternation operator |. The expression

```
expr1 | expr2
```

produces its left-hand side followed by its right-hand side, if needed by the surrounding expression. This can perform many computations quite compactly. For example,

```
x = (3 | 5)
```

checks to see if the value of x is 3 or 5. More complex expressions follow logically:

```
(x | y) = (3 | 5)
```

checks to see if either x or y has the value 3 or 5. It is the Unicon equivalent of C's

```
(x == 3) || (x == 5) || (y == 3) || (y == 5)
```

In understanding Unicon code, it helps if you identify the generators, if there are any. In addition to the alternation operator | and the function find(), there are a few other generators in Icon's built in repertoire of operators and functions. We mention them briefly here, so you can be on the lookout for them when reading code examples.

The expression i to j is a generator that produces all the values between i and j. The expression i to j by k works similarly, incrementing each result by k; i, j, and k must all be integer or real numbers, and k must be non-zero. The expression i to j is equivalent to i to j by 1. The unary ! operator is a generator that produces the elements of its argument. This works on every type where it makes sense. Applied to a string, it produces all its characters (in order). Sets, tables, lists, or records produce the members of the structure.

Generators get resumed for more results as needed in order for the surrounding expression to succeed, and this may propagate through many levels of nested enclosing expressions. However, special expressions called *bounded expressions* will never resume their generator subexpressions. For example, the conditional expressions used in if and while are never resumed if they succeed; if they produce a result the then-branch or the loop body

is executed, and if that code fails, it does not cause generators in the conditional to be resumed. Those bounded conditional expressions are re-evaluated starting from scratch if execution comes their way again. Another popular bounded expression is the semi-colon operator. The expression expr1 ; expr2 evaluates the two expressions in order, and bounds the first expression, so you don't have to worry about backtracking into it if the second expression fails.

1.9 Iteration and Control Structures

You have already seen two control structures in Unicon: the if and the while loop, which test for success. Unicon has several other control structures that vary in complexity and usefulness. Unicon is expression-based, and all control structures are expressions that can be used in surrounding expressions if desired. The big difference between control structures and ordinary operators or procedures is that ordinary operators and procedures don't execute until their arguments have been evaluated and produced a result; they don't execute at all if an argument fails. In contrast, a control structure uses one of its arguments to decide whether (or how many times) to evaluate its other arguments.

Since control structures are expressions, they may produce a result for a surrounding expression. For example, the result of an if expression is the result of either its then part or its else part, whichever one was selected. On the other hand, a loop executes until its test fails, after which there is no meaningful result for it to produce; loops usually fail as far as surrounding expressions are concerned.

The control structure every processes the entire sequence of values produced by a generator. The expression

every *expr1* do *expr2*

evaluates *expr2* for each result generated by *expr1*. This loop looks similar enough to a while loop to confuse people at first. The difference is that a while loop re-evaluates *expr1* from scratch after each iteration of *expr2*, but every *resumes* *expr1* for an additional result where it left off the last time through the loop. Using a generator to control a while loop makes no sense; the generator will restart each iteration, which may give you an infinite loop. Similarly, *not* using a generator to control an every loop also makes no sense; if *expr1* is not a generator the loop body executes at most one time.

The classic example of every is a loop that generates the number sequence from a to expression, assigning the number to a variable that can be used in *expr2*. In many languages these are called "for" loops. A for loop in Unicon is written like this:

```
every i := 1 to 10 do write(i)
```

Of course, every and to are not limited to this BASIC-style for loop. Generators are more flexible; the for loop above looks clumsy when compared with the equivalent

```
every write(1 to 10)
```

A generator operand to a non-generator forms a generator. The enclosing non-generator (such as write()) is re-run each time the generator suspends and resumes.

Unicon's every keyword generalizes the concept of *iterators* found in other languages. Iterators are special control structures that walk through a collection of data. Instead of being a special feature, in Unicon iterators are just one of many ways to utilize generators. The sequence of results from an expression does not have to be stored in a structure to iterate over them. The sequence of results does not even have to be finite; many generator expressions produce an infinite sequence of results, as long as the surrounding expression keeps asking them for more. Here is another example of how every expressions are more flexible than for loops. The expression

```
every f(1 | 4 | 9 | 16 | 25 | 36)
```

executes the function f several times, passing the first few square numbers as parameters. A shorter equivalent that uses the power operator (^) is every f((1 to 6)^2). An example in the next section shows how to generalize this to work with all the squares.

The if, while, and every expressions are Unicon's primary control structures. Several other control structures are available that may be more useful in certain situations. The loop

```
until expr1 do expr2
```

is while's evil twin, executing *expr2* as long as *expr1* fails; on the other hand

```
repeat expr
```

is an infinite loop, executing *expr* over and over.

There are also variations on the if expression introduced earlier for conditional execution. First, the else branch is optional, so you can have an if expression that does nothing if the condition is not satisfied. Second, there is a special control structure introduced for the common situation in which several alternatives might be selected using a long sequence of if ... else if ... else if ... expressions. The case expression replaces these chains of if expressions with the syntax:

```
case expr of {
branch1: expr1
branch2: expr2
...
default: expr_i
}
```

When a case expression executes, the *expr* is evaluated, and compared to each branch value until one that matches exactly is found, as in the binary equality operator ===. Branch expressions can be generators, in which case every result from the branch is compared with *expr*. The default branch may be omitted in case expressions for which no action is to be taken if no branch is satisfied.

When we introduced the repeat loop, you probably were wondering how to exit from the loop, since most applications do not run forever. One answer would be to call one of the built-in functions that terminate the entire program when it is time to get out of the loop. The exit(), stop(), and runerr() functions all serve this valuable purpose; their differences are described in Appendix A.

A less drastic way to get out of any loop is to use the break expression:

```
break expr
```

This expression exits the nearest enclosing loop; *expr* is evaluated outside the loop and treated as the value produced by executing the loop. This value is rarely used by the surrounding expression, but the mechanism is very useful, since it allows you to chain any number of breaks together, to exit several levels of loops simultaneously. The break expression has a cousin called next that does not get out of a loop, but instead skips the rest of the current iteration of a loop and begins the next iteration of the loop.

1.10 Procedures

Procedures are a basic building block in most languages. Here is an example of an ordinary procedure. This one computes a simple polynomial, $ax^2 + bx + c$.

```
procedure poly(x,a,b,c)
   return a * x^2 + b * x + c
end
```

Parameters

Procedure parameters are passed by value except for structured data types, which are passed by reference. This means that when you pass in a string, number, or cset value, the procedure gets a *copy* of that value; any changes the procedure makes to its copy will not be reflected in the calling procedure. On the other hand, structures that contain other values, such as lists, tables, records, and sets are not copied. The procedure being called gets a handle to the original value, and any changes it makes to the value will be visible to the calling procedure. Structure types are described in the next chapter.

When you call a procedure with too many parameters, the extras are discarded. This feature is valuable for prototyping but can be dangerous if you aren't careful! Similarly,

if you call a procedure with too few parameters, the remaining variables are assigned the null value, &null. The null value is also passed as a parameter if you omit a parameter.

Now it is time to describe the unary operators \ and /. These operators test the expression against the null value. The expression /x succeeds and returns x if the value is the null value. This can be used to assign default values to procedure arguments: Any arguments that have a null value can be tested for and assigned a value. Here's an example:

```
procedure succ(x, i)
   /i := 1
   return x + i
end
```

If this procedure is called as succ(10), the missing second parameter, i, is assigned the null value. The forward slash operator then tests i against the null value, which succeeds; therefore the value 1 is assigned to i.

The backward slash checks if its argument is non-null. For example, this will write the value of x if it has a non-null value:

```
   write("The value of x is ", \x)
```

If x does in fact have the null value, then \x will fail, which will mean that the write() procedure will not be called. If it helps you to not get these two mixed up, a slash pushes its operand "up" for non-null, and "down" for a null value.

The procedure succ() shows one way to specify a default value for a parameter. The built-in functions use such default values systematically to reduce the number of parameters needed; consistent defaults for entire families of functions make them easy to remember. Another key aspect of the built-in operations is implicit type conversion. Arguments to built-in functions and operators are converted as needed.

Defaulting and type conversion are so common and so valuable in Unicon that they have their own syntax. Parameter names may optionally be followed by a coercion function (usually the name of a built in type) and/or a default value, separated by colons. These constructs are especially useful in enforcing the public interfaces of library routines that will be used by many people.

```
procedure succ(x:integer, i:integer:1)
end
```

This parameter declaration is a more concise equivalent of

```
procedure succ(x, i)
   x := integer(x) | runerr(101, x)
   /i := 1 | i := integer(i) | runerr(101, i)
end
```

Variables and scopes

Variable names used as parameters are fundamentally different from procedure names. The *scope* of parameters is limited to the body of a single procedure, while procedure names are visible to the entire program. Parameters and procedures are special forms of two basic kinds of variables in Unicon: local variables and global variables.

The scope of global variables, including procedure names, consists of the entire program. Their value is stored in memory for the entire execution of the program. There are two kinds of local variables; both introduce names that are defined only within a particular procedure. Regular local variables are created when a procedure is called, and destroyed when a procedure fails or returns a result; a separate copy of regular local variables is created for each call. Static local variables are global variables whose names are visible only within a particular procedure; a single location in memory is shared by all calls to the procedure, and the last value assigned in one call is remembered in the next call.

Variables do not have to be declared, and by default they are *local*. To get a global variable, you have to declare it outside any procedure with a declaration like this:

```
global MyGlobal
```

Such a declaration can be before, after, or in between procedures within a program source file, but cannot be inside a procedure body. Of course, another way to declare a global variable is to define a procedure; this creates a global variable initialized with the appropriate procedure value containing the procedure's instructions.

Regular and static local variables may be declared at the top of a procedure body, after the parameters and before the code starts, as in the following example:

```
procedure foo()
   local x, y
   static z
   ...
end
```

Each declared local variable name may be followed by a := and an *initializer* expression that specifies the variable's initial value. Without an initializer, variables start with the value &null. Although you do not have to declare local variables, large programs, library code or multi-person projects should declare all local variables. If you don't, and some other part of the code introduces a global variable by the same name as your undeclared local, your variable will be interpreted as a reference to the global. To help avoid this problem, the -u command line option to the compiler causes undeclared local variables to produce a compilation error message.

A procedure body can begin with an initial clause, which executes only the first time the procedure is called. The initial clause is mainly used to initialize static variables in ways

that aren't handled by initializers. For example, the following procedure returns the next number in the Fibonacci sequence each time it is called, using static variables to remember previous results in the sequence between calls.

```
procedure fib()
static x,y
local z
initial {
    x := 0
    y := 1
    return 1
    }
    z := x + y
    x := y
    y := z
    return z
end
```

Writing your own generators

When a procedure returns a value, the procedure and its regular local variables cease to exist. But there are expressions that don't disappear when they produce a value: generators! You can create your own generators by writing procedures that use suspend instead of return. suspend is different from return in that it saves the point of execution within the procedure; if another value is required by the calling expression, the generator continues execution from where it previously left off.

Here is a procedure to generate all the squares. Instead of using multiplication, it uses addition to demonstrate generators! The code uses the fact that if we keep adding successive odd numbers, we get the squares.

```
procedure squares()
    odds := 1
    sum := 0
    repeat {
      suspend sum
      sum +:= odds
      odds +:= 2
    }
end
```

To perform a computation on the squares, we can use it in an every statement:

```
    every munge(squares())
```

Warning

This is an infinite loop! (Do you know why? Whether munge() succeeds or fails, every will always resume squares() for another result to try; squares() generates an infinite result sequence.)

The fail expression makes the procedure fail. Control goes to the calling procedure, returning no value, and the procedure call ceases to exist; it cannot be resumed. A procedure also fails implicitly when control flows off the end of the procedure's body.

Here is a procedure that produces all the non-blank characters of a string, but bails out if the character # is reached:

```
procedure nonblank(s)
   every c := !s do {
      if c == "#" then fail
      if c ~== " " then suspend c
   }
end
```

Recursion

A recursive procedure is one that calls itself, directly or indirectly. There are many cases where it is the most natural way of solving the problem. Consider the famous "Towers of Hanoi" problem. Legend has it that when the universe was created, a group of monks in a temple in some remote place were presented with a problem. There are three diamond needles, and on one of them is a stack of 64 golden disks all of different sizes, placed in order with the largest one at the bottom and the smallest on top. All the disks are to be moved to a different needle under the conditions that only one disk may be moved at a time, and a larger disk can never be placed on a smaller disk. When the monks finish this task, the universe will come to an end.

How can you move the n smallest disks? If n is 1, just move it. Since it's the smallest, this will not violate the condition. If n is greater than 1, here's what we can do: first, move the *n-1* upper disks to the intermediate needle, then transfer the nth disk, then move the *n-1* upper disks to the destination needle. This whole procedure does not violate the requirements either (satisfy yourself that such is the case).

Now write the procedure hanoi(n) that computes this algorithm. The first part is simple: if you have one disk, just move it.

```
procedure hanoi(n, needle1:1, needle2:2)
   if n = 1 then write("Move disk from ", needle1, " to ", needle2)
```

Otherwise, perform a recursive call with *n-1*. First, to find the spare needle we have:

```
   other := 6 - needle1 - needle2
```

Now move the *n-1* disks from needle1 to other, move the biggest disk, and then move the *n-1* again. The needles are passed as additional parameters into the recursive calls. They are always two distinct values out of the set {1, 2, 3}.

```
hanoi(n-1, needle1, other)
write("Move disk from ", needle1, " to ", needle2)
hanoi(n-1, other, needle2)
```

That's it! You're done. Listing 1-2 contains the complete program for you to try:

Listing 1-2 Towers of Hanoi

```
procedure main()
   write("How many disks are on the towers of Hanoi?")
   hanoi(read())
end
procedure hanoi(n:integer, needle1:1, needle2:2)
local other
   if n = 1 then write("Move disk from ", needle1, " to ", needle2)
   else {
     other := 6 - needle1 - needle2
     hanoi(n-1, needle1, other)
     write("Move disk from ", needle1, " to ", needle2)
     hanoi(n-1, other, needle2)
   }
end
```

Turn on tracing see how this program works. To enable tracing, compile your program with a -t option, or assign the keyword &trace a non-zero number giving the depth of calls to trace. Setting &trace to -1 will turn on tracing to an infinite depth.

To move n disks, 2^n - *1* individual disk movements will be required. If the monks move one disk a second, it will take 2^{64} - *1* seconds, or about 60 trillion years. Wikipedia has listed the age of the universe at around 13.75 billion years. It seems unlikely that we need worry about the monks finishing their task!

Summary

In this chapter you have learned:

- Unicon is an expression-based language organized as a set of procedures starting from a procedure called main().
- Unicon has four atomic types: arbitrary precision integers, real numbers, arbitrary length strings of characters, and character sets.

- There is no Boolean concept in Unicon; instead, control is driven by whether an expression succeeds in producing a result or fails to do so. This eliminates the need for most sentinel values, shortening many expressions.
- Generator expressions can produce multiple results as needed by the surrounding expression in order for it to produce results.
- Procedure parameters are passed by value for atomic types, and by reference for all other data types. Unicon features extensive argument defaults and automatic type coercion throughout its built-in function and operator repertoire.
- Unicon has two scope levels: global and local. Undeclared variables are implicitly defined to be local.

Chapter 2

Structures

The examples in the previous chapter employed data types whose values are *immutable*. For example, all operations that manipulate numbers and strings compute new values, rather than modify existing values. This chapter presents structured types that organize and store collections of arbitrary (and possibly mixed) types of values. When you complete this chapter, you will understand how to use these types.

- Tables associate their elements with key values for rapid lookup.
- Lists offer efficient access by position as well as by stack or queue operations.
- Records store values using a fixed number of named fields.
- Sets support operations such as union and intersection on groups of elements.
- Using structures to represent trees, graphs, and matrices.

There are several structure types that describe different basic relationships between values. The philosophy of structures in Unicon is to provide built-in operators and functions for common organization and access patterns - the flexible "super glue" that is needed by nearly all applications. Their functionality is similar to the C++ Standard Template Library or generic classes in other languages, but Unicon's structure types are much simpler to learn and use, and are well supported by the expression evaluation mechanism described in the previous chapter.

All structure types in Icon share many aspects in common, such as the fact that structures are *mutable*. The values inside them may change. In that respect, structures are similar to a collection of variables that are bundled together. In many cases, Unicon's structure types are almost interchangeable! Operators like subscripts and built-in functions such as insert() are defined consistently for many types. Code that relies on such operators is *polymorphic*: it may be used with multiple structure types in an interchangeable way.

For both the structures described in this chapter and the strings described in the next chapter, be aware that Unicon performs automatic storage management, also known as

garbage collection. If you have used a language like C or C++, you know that one of the biggest headaches in writing programs in these languages is tracking down bugs caused by memory allocation, especially dynamic heap memory allocation. Unicon transparently takes care of those issues for you.

Another big source of bugs in languages like C and C++ are pointers, values that contain raw memory addresses. Used properly, pointers are powerful and efficient. The problem is that they are easy to use incorrectly by accident; this is true for students and practicing software engineers alike. It is easy in C to point at something that is off-limits, or to trash some data through a pointer of the wrong type.

Unicon has no pointer types, but all structure values implicitly use pointer semantics. A *reference* is a pointer for which type information is maintained and safety is strictly enforced. All structure values are references to data that is allocated elsewhere, in a memory region known as the heap. You can think of a reference as a safe pointer: the only operations it supports are copying the pointer, or dereferencing it using an operation that is defined for its type.

Assigning a structure to a variable, or passing it as a parameter, gives that variable or parameter a copy of the reference to the structure but does not make a copy of the structure. If you want a copy of a structure, you call the function copy(x), which makes a "shallow" copy of a single table, list, record, or set. If that structure contains references to other structures as its elements, those substructures are not copied by copy(). To copy a "deep" structure (lists of lists, tables of records, etc.) you can use the procedure deepcopy() that is given as an example later in this chapter.

2.1 Tables

Tables are unordered collections of values that are accessed using associated *keys*. They are Unicon's most versatile type. All of the other structure types can be viewed as special cases of tables, optimized for performance on common operations. Most operations that are defined for tables are defined for other structure types as well.

Subscripts are used for the primary operations of associating keys with values that are inserted into the table, and then using keys to look up objects in the table. The table() function creates a new empty table. For example, the lines

```
T := table()
T["hello"] := "goodbye"
```

create a new table, and associate the key "hello" with the value "goodbye". The table() function takes one optional argument: the default value to return when lookup fails. The default value of the default value is &null, so after the above example, write(T["goodbye"]) would write an empty line, since write() treats a null argument the same as an empty string, and write() always writes a newline. Assigning a value to a key that is not in the table inserts a value

into the table. This occurs in the second line of the example above, so write(T["hello"]) writes out "goodbye".

Subscripts are the primary operation on tables, but there are several other useful operations. The insert(T, k_1, x_1, k_2, x_2, ...) function adds new key-value pairs to T. The delete(T, k_1, k_2, ...) function deletes values from T that correspond to the supplied keys. Icon's unary * operator produces the size of its argument; for a table, *T is the number of key-value pairs in the table. Unary ! generates elements from a collection; for a table, !T generates the values stored in the table. Unary ? is the random operator; for a table, ?T produces a random value stored in the table. Both unary ! and ? produce values stored in a table, not the keys used to lookup values.

Function member(T, k) succeeds if k is a key in T and fails otherwise. Function key(T) generates the keys that have associated values. The following example prints word counts for the input (assuming getword() generates words of interest):

```
wordcount := table(0)
every word := getword() do wordcount[word] +:= 1
every word := key(wordcount) do write(word, " ", wordcount[word])
```

The default value for the table is 0. When a new word is inserted, the default value gets incremented and the new value (that is, 1) is stored with the new word. Tables grow automatically as new elements are inserted.

Tables are closely related to the set data type (discussed later in this chapter). The keys of a table are a set; the associated values accessed via the subscript operator are sort of a bonus data payload. In any case, tables behave in certain set-like ways; when their elements are generated by the ! operator, they come out in a pseudo random order. Like sets, and csets in the previous chapter, the operators T1++T2, T1**T2, and T1--T2 are the union, intersection, and difference of the tables T1 and T2 based on their keys. These operators construct new tables and do not modify their operands. In union and intersection, when duplicate table keys occur in the two operands, the associated values from the left operand are what goes in the new table that holds the result.

2.2 Lists

Lists are dynamically sized ordered collections of values. They are accessed by subscripts, with indexes starting at 1. You can also insert or remove elements from the beginning, middle, or end of the list. Lists take the place of arrays, stacks, queues, and deques found in other languages and data structures textbooks.

There are three ways to explicitly construct a list. In the most generic form, a list is created by calling the function list(), which takes optional parameters for the list's initial size and the initial value given to all elements of the list. The default size is 0 and the default initial value is &null.

The second form of list constructor is when you create a list by enclosing a comma-separated sequence of 0 or more values in square brackets. For example

```
L := ["linux", 2.0, "unix"]
```

creates a list with three elements, a string, a real number, and another string.

A third form of list constructor, called *comprehension*, looks like the previous form, except the square brackets contain adjacent colon characters and have an expression inside.

```
L := [: expr :]
```

In a comprehension the constructed list's initial values are obtained by fully evaluating an expression and placing all of its results into the list, in order. The expression fails if *expr* fails; if you wanted that to be an empty list you may need to append |[].

Lists are dynamic. Lists grow or shrink as a result of stack and queue operations. The push() and pop() functions add and remove elements from the front of a list, while put() and pull() add and remove elements at the end of the list. In addition, *insert(L, i, x) inserts x at position i, and delete(L, i) deletes the element at position i.* The expression [] is another way to create an empty list; it is equivalent to calling list() with no arguments. The previous list could have been constructed one element at a time with the following code. *put() accepts a variable number of arguments.*

```
L := [ ]
put(L, "linux")
put(L, 2.0)
put(L, "unix")
```

Elements of the list can be obtained either through list manipulation functions or by subscripting. Given the list L above, in the following code the first line writes "unix" while the second line moves the first element to the end of the list.

```
write(L[3])
put(L, pop(L))
```

There is no restriction on the kinds of values that may be stored in a list. For example, the elements of a list can themselves be lists. You can create lists like

```
L := [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

and index them with multiple subscripts. L[2][3] is equivalent to L[2,3] and yields the value 6 in this example.

Lists also support several common operators. The operator *L produces the size of list L. The operators !L and ?L generate the elements of L in sequence, and produce a single random element of L, respectively. The following procedure uses the unary ! operator to sum the values in list L, which must be numbers.

```
procedure sum(L)
    total := 0
    every total +:= !L
    return total
end
```

Comparing the two, lists are like tables with boring keys: the positive integers starting from 1. *Function member(L, k) succeeds if 0 < integer(k) <= *L*, while key(L) is equivalent to the expression 1 to *L. List indexes are contiguous, unlike table keys, and so lists can support a *slice* operator to produce a sublist, given a pair of indexes to mark the bounds. The L[i:j] expression produces a new list that contains a copy of the elements in L between positions i and j. The L[i+:j] expression is equivalent to L[i:i+j]. List concatenation is another valuable operator. The L1 ||| L2 expression produces a new list whose elements are a copy of the elements in L1 followed by a copy of the elements in L2.

2.3 Records

A record is a fixed-sized, ordered collection of values whose elements are accessed using user-defined named *fields*. A record is declared as a global name that introduces a new type with a corresponding constructor procedure, as in the following example. The field names are a comma-separated list of identifiers enclosed in parentheses.

```
record complex(re, im)
```

Record *instances* are created using a constructor procedure with the name of the record type. The fields of an instance are accessed by name using dot notation or string subscript, or by integer index subscript. You can use records as records, or as special tables or lists with a constant size and fixed set of keys.

member(R,s) tests whether s is a field in R; key(R) generates R's field names. Functions like insert(), or push() are not supported on records, since they change the size of the structure that they modify. Here is a demonstration of record operations.

```
a := complex(0, 0)
b := complex(1, -1)
if a.re = b.re then write("not likely")
if a["re"] = a[2] then write("a.re and a.im are equal")
```

Unicon provides a mechanism for constructing new record types on the fly, described in Chapter 6, as well as the ability to declare classes, which are new data types that form the building blocks for object-oriented programs, described starting in Chapter 9. Records are closely related to classes and objects: they can be considered to be an optimized special case of objects that have no methods.

2.4 Sets

A set is an unordered collection of values with the uniqueness property: an element can only be present in a set once. The function set(x...) creates a set containing its arguments. For the sake of backward compatibility with Icon, list arguments to set() are not inserted; instead, the list elements are inserted. As with other structures, the elements may be of any type, and may be mixed. For example, the assignment

```
S := set("rock lobster", 'B', 52)
```

creates a set with three members: a string, a cset, and an integer. The equivalent set is produced by set(["rock lobster","B", 52]). To place a list in a set set constructor, wrap it in another list, as in set([L]), or insert the list into the set after it is created. Because the set constructor function initializes directly from a list argument, "set comprehension" follows trivially from list comprehension. For example, set([: 2 to 20 by 2 :]) creates a set containing the even integers from two to twenty.

The functions member(), insert(), and delete() do what their names suggest. As for csets in the previous chapter, S1++S2, S1**S2, and S1--S2 are the union, intersection, and difference of sets S1 and S2. Set operators construct new sets and do not modify their operands. Because a set can contain any value, it can contain a reference to itself. This is one of several differences between Unicon sets, which are mutable structures, and mathematical sets. Another difference is that Unicon sets have a finite number of elements, while mathematical sets can be infinite in size.

As a short example, consider the following program, called uniq, that filters duplicate lines in its standard input as it writes to its standard output. Unlike the UNIX utility of this name, our version does not require the duplicate lines to be adjacent.

```
procedure main()
    S := set()
    while line := read() do
      if not member(S, line) then {
         insert(S, line)
         write(line)
         }
end
```

Sets are closely related to the table data type. They are very similar to an optimized special case of tables that map all keys to the value &null. Unlike tables, sets have no default value and do not support the subscript operator.

2.5 Using Structures

Structures can hold other structures, allowing you to organize information in whatever way best fits your application. Building complex structures such as a table of lists, or a list

of records that contain sets, requires no special trickery or new syntax. Examples of how such structures are accessed and traversed will get you started. Recursion is often involved in operations on complex structures, so it plays a prominent role in the examples. The concept of recursion was discussed in Chapter 1.

A Deep Copy

The built-in function copy(x) makes a one-level copy of structure values. For a multi-level structure, you need to call copy() for each substructure if the new structure must not point into the old structure. This is a natural task for a recursive function.

```
procedure deepcopy(x)
    local y
    case type(x) of {
    "table"|"list"|"record": {
      y := copy(x)
      every k := key(x) do y[k] := deepcopy(x[k])
      }
    "set": {
      y := set()
      every insert(y, deepcopy(!x))
      }
    default: return x
    }
    return y
end
```

This version of deepcopy() works for arbitrarily deep tree structures, but the program execution will crash if deepcopy() is called on a structure containing cycles. It also does not copy directed acyclic graphs correctly. In both cases the problem is one of not noticing when you have already copied a structure, and copying it again. The Icon Program Library has a deep copy procedure that handles this problem, and we present the general technique that is used to solve it in the next section.

Representing Trees and Graphs

Since there is no restriction on the types of values in a list, they can be other lists too. Here is an example of how a tree may be implemented with records and lists:

```
record node(name, links)
    ...
barney := node("Barney", list())
betty := node("Betty", list())
bambam := node("Bam-Bam", [barney, betty])
```

The structure created by these expressions is depicted in Figure 2-1. The list of links at each node allows trees with an arbitrary number of children at the cost of extra memory and indirection in the tree traversals. The same representation works for arbitrary graphs.

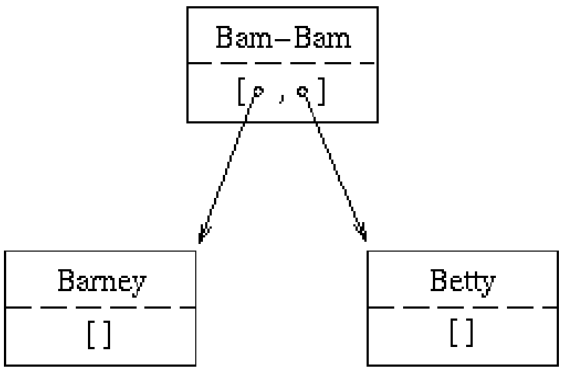


Figure 2-1: A Record Containing a List of Two Records

To find every node related to variable bambam, follow all the links reachable starting from bambam. Here is a procedure that performs this task.

```
procedure print_relatives(n)
local i
static relatives
initial relatives := set()
every i := n | !n.links do {
      if not member(relatives, i.name) then {
        write(i.name)
        insert(relatives, i.name)
        print_relatives(i)
        }
      }
end
```

Calling print_relatives(bambam) will print

```
Bam-Bam
Barney
Betty
```

Static variables and the initial clause are explained in Chapter 1. Can you guess what purpose static variable relatives serves? For a proper tree structure, it is not needed at all, but for more general data structures such as directed graphs this static variable is very important! One defect of this procedure is that there is no way to reset the static variable and call print_relatives() starting from scratch. How would you remove this defect?

The n-Queens Example

The 8-Queens problem is a classic backtracking problem. The goal is to place eight queens on a chessboard so that none of the queens attack any other. Here is a solution to a more general form of the problem, that of placing n queens on an n x n board. The solution we present is by Steve Wampler, and it is in the Icon Program Library.

An array of size n stores the solutions, with each element representing a column. The values in the array are integers specifying the row in each column that has the queen. (Since the queens cannot attack each other, each column must contain exactly one queen.) The problem size n and the array are declared global so that all procedures can see them; this allows the program to avoid passing these variables in to every procedure call. Use globals sparingly, and only where they are appropriate, as is the case here.

```
link options
global solution, n
procedure main(args)
local i, opts
```

The program starts by handling command-line arguments. In Unicon programs, main() is called with a single parameter that is a list of strings whose elements are the command-line arguments of the program.

The n-queens program recognizes only one thing on the command line: the option -n followed by an integer specifies the size of board to use. Thus the command line queens -n 9 will generate solutions on a 9x9 board. The default value of n is 6. The options() procedure is an Icon Program Library procedure described in Appendix B; it removes options from the command line and places them in a table whose keys are option letters such as "n". Library procedures such as options() are incorporated into a program using the link declaration, as in the link options that begins the code fragment above. A link declaration adds the procedures, global variables, and record types in the named module (in this case, procedure options() came from a file options.icn) to the program.

```
    opts := options(args,"n+")
    n := \opts["n"] | 6
    if n <= 0 then stop("-n needs a positive numeric parameter")
```

The value n gives the size for the solution array and also appears in a banner:

```
    solution := list(n)   # a list of column solutions
    write(n,"-Queens:")
    every q(1)          # start by placing queen in first column
end
```

Now comes the meat of the program, the procedure q(c). It tries to place a queen in column c and then calls itself recursively to place queens in the column to the right. The q(c) procedure uses three arrays: rows, up, and down. They are declared to be *static*, meaning that their values will be preserved between executions of the procedure, and all instances of the procedure will share the same lists. Since each row must have exactly one queen, the rows array helps to make sure any queen that is placed is not on a row that already has a queen. The other two arrays handle the diagonals: up is an array (of size *2n-1*) of the upward slanting diagonals, and down is an array for the downward slanting diagonals. Two queens in positions *(r_1, c_1)* and *(r_2, c_2)* are on the same "up" diagonal if *n+r_1-c_1 = n+r_2-c_2* and they are on the same "down" diagonal if *r_1+c_1-1 = r_2+c_2-1*. Figure 2-2 shows some of the "up" and "down" diagonals.

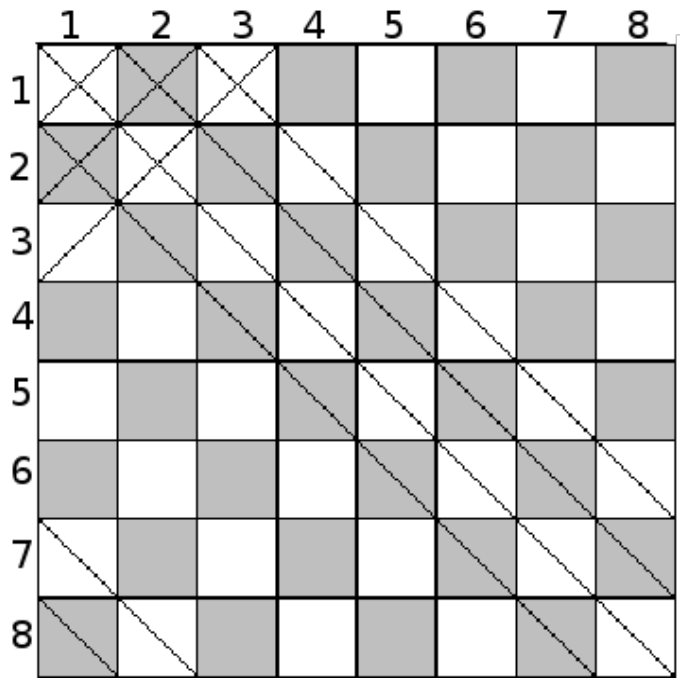


Figure 2-2: Up and Down Diagonals in the n-Queens Problem

```
#
# q(c) - place a queen in column c.
#
procedure q(c)
local r
static up, down, rows
initial {
    up := list(2*n-1,0)
    down := list(2*n-1,0)
    rows := list(n,0)
    }
```

The next expression in q() is an every loop that tries all possible values for the queen in row c. The variable r steps through rows 1 to 8. For any row at which the program places a queen, it must ensure that

1. rows[r] is zero, that is, no other column has a queen in row r,
2. up[n+r-c] is 0, that is, there is not already a queen in the "up" diagonal, and
3. down[r+c-1] is 0, that is, there is not already a queen in the down diagonal.

If these conditions are met, then it is OK to place a queen by assigning a 1 to all those arrays in the appropriate position:

```
    every 0 = rows[r := 1 to n] = up[n+r-c] = down[r+c-1] &
      rows[r] <- up[n+r-c] <- down[r+c-1] <- 1 do {
```

For assignment, instead of := this expression uses the *reversible assignment* operator <-. This assigns a value just like in conventional assignment, but it remembers the old value; if it is ever resumed, it restores the old value and fails. This causes the appropriate entries in the row, up, and down arrays will be reinitialized between iterations.

When the every loop found a good placement for this column, either the program is done (if this was the last column) or else it is time to try to place a queen in the next row:

```
      solution[c] := r       # record placement.
      if c = n then show()
      else q(c + 1)          # try to place next queen.
      }
end
```

That's it! The rest of the program just prints out any solutions that were found.

Printing the chess board is similar to other reports you might write that need to create horizontal lines for tables. The repl() function is handy for such situations. The repl(s, i) function returns i "replicas" of string s concatenated together. The show() function uses it to create the chessboard.

```
#
# show the solution on a chess board.
#
procedure show()
static count, line, border
initial {
    count := 0
    line := repl("|   ",n) || "|"
    border := repl("----",n) || "-"
    }
    write("solution: ", count+:=1, "\n  ", border)
    every line[4*(!solution - 1) + 3] <- "Q" do {
```

```
        write("  ", line, "\n  ", border)
        }
    write()
end
```

2.6 Summary

Unicon's structures are better than sliced bread. To be fair, this is because Icon's inventors really got things right. These structures are the foundations of complex algorithms and the glue that builds sophisticated data models. They are every computer scientists' buzzword-compliant best friends: polymorphic, heterogeneous, implicitly referenced, cycle-capable, dynamically represented, and automatically reclaimed. They provide a direct implementation of the common information associations used in object-oriented design. But most important of all, they are extremely simple to learn and use.

Chapter 3

String Processing

In addition to its groundbreaking expression evaluation, by combining compelling string processing features from its ancestors Icon and SNOBOL4, Unicon provides some of the most flexible and readable built-in string processing facilities found in any language. If you are used to string processing in a mainstream language, hold on to your hat: things are about to get interesting.

In this chapter you will learn

- How to manipulate strings and sets of characters
- the string scanning control structure, used to match patterns directly in code
- the pattern type, used to match patterns constructed as data
- How to write custom pattern matching primitives, with backtracking
- techniques for matching regular expressions and context free grammars

3.1 The String and Cset Types

All mainstream programming languages have a string type, but the details of Unicon's string type set it apart from other languages. And almost no other mainstream languages feature a data type dedicated to character sets, which are quite useful.

3.1.1 String Indexes

You have already seen string literals delimited by double quotes, and the most common operators that work on strings: the size of a string is given by the unary * operator, substrings can be picked out with square-bracketed indexes, and two strings can be concatenated with the || operator. It is time for a deeper explanation of the meaning of indexes as they are used with strings and lists.

Indexes in a string refer to the positions *between* characters. The positions are numbered starting from 1. The index 0 refers to the position after the last character in the string, and negative indices count from the right side of the string:

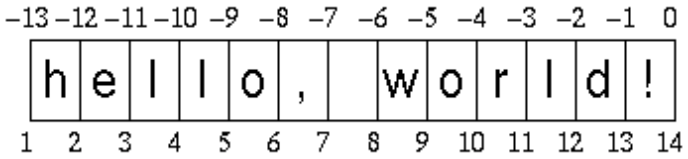


Figure 3-1: Positive and Negative String Indices

The expression s[i:j] refers to the substring of s that lies between positions i and j. If either i or j is not a valid index into s, the expression fails. The expression s[k] is short for s[k:k+1] and refers to a single character at position k. The expression s[k+:n] is the substring of length n starting at position k. If s is the string "hello, world!" then the expressions

```
s[7] := " puny "
s[13:18] := "earthlings"
```

change s into "hello, puny earthlings!", illustrating the ease with which insertions and substitutions are made. The first assignment changes the string to "hello, puny world!", replacing a single character with six characters and thereby increasing its length. The second assignment operates on the modified string, replacing "world" with "earthlings".

Strings are values, just like numbers; if you copy a string and then work on the copy, the original will be left unchanged:

```
s := "string1"
new_s := s
new_s[7] := "2"
```

Now the value of new_s is "string2" but s is left unchanged.

As mentioned in Chapter 1, strings can be compared with string comparison operators such as ==.

```
if line[1] == "#" then ...
```

If you find you are writing many such tests, the string processing you are doing may be more cleanly handled using the string scanning facilities, described below. But first, here is some more detail on the character set data type, which is used in many of the string scanning functions.

3.1.2 Character Sets

A cset is a set of characters. It has the usual properties of sets: order is not significant, and a character can only occur once in a cset. A cset literal is represented with single quotes:

```
c := 'aeiou'
```

Since characters can only occur once in a cset, duplicates in a cset literal are ignored; for example, 'aaiiee' is equivalent to 'aie'. Strings can be converted to csets and vice versa. Since csets do not contain duplicates, when a string is converted to a cset, all the duplicates are removed.

Therefore to see if a string is composed of all the vowels and no consonants:

```
if cset(s) == 'aeiou' then ...
```

Or, to find the number of distinct characters in a string:

```
n := *cset(s)
```

The ! operator generates the members of a cset in sorted order; this is also useful in some situations.

3.1.3 Character Escapes

Both strings and csets rely on the backslash as an escape character within string literals. A backslash followed by an *escape code* of one or more characters specifies a non-printable or control character. Escape codes may be specified by a numeric value given in hex or octal format - for example, "\x41". Alternatively, any control character may be specified with an escape code consisting of the caret (^) followed by the alphabetic letter of the control character. A cset containing control-C, control-D, and control-Z could be specified as '\^c\^d\^z'. For the most common character escapes, a single-letter code is defined, such as "\t" for the tab character, or "\n" for the newline. For all other characters, the character following the backslash is the character; this is how quotes or backslashes are included in literals. The escape codes are summarized in Table 3-1.

Table 3-1
Escape Codes and Characters

Code	Character	Code	Character	Code	Character	Code	Character
\b	backspace	\d	delete	\e	escape	\f	form feed

\l	line feed	\n	newline	\r	carriage return	\t	tab
\v	vertical tab	\'	quote	\"	double quote	\\	backslash
ooo	octal	\x*hh*	hexadecimal	\^*x*	Control-*x*		

3.2 String Scanning

Strings are ordered sequences of symbols. A string's vital information is conveyed both in its individual elements and in the number and order in which the symbols appear. There is a fundamental duality between writing the code to analyze a string, and writing down some data that describes or abstracts that string. The same duality is seen in Unicon's string scanning control structure described in this section, and the pattern data type used in matching operators, which is described in this next section.

Unicon's main building block for string analysis is a control structure called *string scanning*. A scanning environment consists of a string *subject* and an integer *position* within the subject at which scanning is to be performed. These values are held by the keyword variables &subject and &pos. Scanning environments are created by an expression of the form

s ? *expr*

The binary ? operator sets the subject to its left argument and initializes the position to 1; then it executes the expression on the right side.

The expression usually has an interesting combination of various *matching functions* in it. Matching functions change the position, and return the substring between the old and new positions. For example: move(j) moves the position j places to the right and returns the substring between the old and new position. This string will have exactly j characters in it. When the position cannot move as directed, for example because there are less than j characters to the right, move() fails. Here is a simple example:

```
text ? {
   while move(1) do
     write(move(1))
   }
```

This code writes out every other character of the string in variable text.

Another function is tab(i), which sets the position &pos to its argument and returns the substring that it passed over. So the expression tab(0) will return the substring from the current position to the end of the string, and set the position to the end of the string.

Several string scanning functions examine a string and generate the interesting positions in it. We have already seen find(), which looks for substrings. In addition to the other parameters that define what the function looks for, these string functions end with three optional parameters: a string to examine and two integers. These functions default their

string parameter to &subject, the string being scanned. The two integer positions specify where in the string the processing will be performed; they default to 1 and 0 (the entire string), or &pos and 0 if the string defaulted to &subject. Here is a generator that produces the words from the input:

```
procedure getword()
local wchar, line
    wchar := &letters ++ &digits ++ '\'-'
    while line := read() do
      line ? while tab(upto(wchar)) do {
        word := tab(many(wchar))
        suspend word
        }
end
```

Variable wchar is a cset of characters that are allowed in words, including apostrophe (which is escaped) and hyphen characters. upto(c) returns the next position at which a character from the cset c occurs. The many(c) function returns the position after a sequence of characters from c, if one or more of them occur at the current position. The expression tab(upto(wchar)) advances the position to a character from wchar; then tab(many(wchar)) moves the position to the end of the word and returns the word that is found. This is a generator, so when it is resumed, it takes up execution from where it left off and continues to look for words (reading the input as necessary).

Notice the first line: the cset wchar is the set union of the upper- and lowercase letters (the value of the keyword &letters) and the digits (the keyword &digits). This cset union is performed each time getword() is called, which is inefficient if getword() is called many times. The procedure could instead calculate the value once and store it for all future calls to getword().

Declaring the variable to be static will cause its value to persist across calls to the procedure. Normal local variables are initialized to the null value each time a procedure is entered. To do this, add these two lines to the beginning of the procedure:

```
static wchar
initial wchar := &letters ++ &digits ++ '\'-'
```

The match(s) function takes a string argument and succeeds if s is found at the current position in the subject. If it succeeds, it produces the position at the end of the matched substring. This expression

```
if tab(match("-")) then sign := -1 else sign := 1
```

looks to see if there is a minus sign at the current position; if one is found, &pos is moved past it and the variable sign is assigned a -1; otherwise, it gets a 1. The expression tab(match(s))

occurs quite often in string scanning, so it is given a shortcut: =s. The section on pattern matching later in this chapter will explain that this "unary equals" operator has an additional, more powerful use.

The last two string scanning functions to round out Icon's built-in repertoire are any(c) and bal(c1,c2,c3). any(c) is similar to many(), but only tests a single character being scanned to see if it is in cset c. The bal() function produces positions at which a character in c1 occurs, similar to upto(), with the added stipulation that the string up to those positions is *balanced* with respect to characters in c2 and c3. A string is balanced if it has the same number of characters from c2 as from c3 and there are at no point more c3 characters present than c2 characters. The c1 argument defaults to &cset. Since c2 and c3 default to '(' and ')', bal() defaults to find balanced parentheses.

The restriction that bal() only returns positions at which a character in c1 occurs is a bit strange. Consider what you would need to do in order to write an expression that tells whether a string s is balanced or not.

You might want to write it as s ? (bal() = *s+1) but bal() will never return that position. Concatenating an extra character solves this problem:

```
procedure isbalanced(s)
    return (s || " ") ? (bal() = *s+1)
end
```

If string s is very large, this solution is not cheap, since it creates a new copy of string s. You might write a version of isbalanced() that doesn't use the bal() function, and see if you can make it run faster than this version. An example later in this chapter shows how to use bal() in a more elegant manner.

File Completion

Consider the following gem, attributed to Jerry Nowlin and Bob Alexander. Suppose you want to obtain the full name of a file, given only the first few letters of a filename and a list of complete filenames. The following one line procedure does the trick:

```
procedure complete(prefix, filenames)
    suspend match(prefix, p := !filenames) & p
end
```

This procedure works fine for lists with just a few members and also for cases where prefix is fairly large.

Backtracking

The matching functions we have seen so far, (tab() and move()), are actually generators. That is, even though they only produce one value, they suspend instead of returning. If

expression evaluation ever resumes one of these functions, they restore the old value of &pos. This makes it easy to try alternative matches starting from the same position in the string:

```
s ? (="0x" & tab(many(&digits ++ 'abcdefABCDEF'))) |
    tab(many(&digits))
```

This expression will match either a hexadecimal string in the format used by C or a decimal integer. Suppose s contains the string "0xy". The first part of the expression succeeds and matches the "0x"; but then the expression tab(many(&digits ++ 'abcdef')) fails; this causes Unicon to resume the first tab(), which resets the position to the beginning of the string and fails. Unicon then evaluates the expression tab(many(&digits)) which succeeds (matching the string "0"); therefore the entire expression succeeds and leaves &pos at 2.

Warning

Be careful when using tab() or move() in a surrounding expression that can fail! The fact that tab() and move() reset &pos upon expression failure causes confusion and bugs when it happens accidentally.

Concordance Example

Listing 3-1 illustrates the above concepts and introduces a few more. Here is a program to read a file, and generate a concordance that prints each word followed by a list of the lines on which it occurs. Short words like "the" aren't interesting, so the program only counts words longer than three characters.

Listing 3-1 A simple concordance program

```
procedure main(args)
    (*args = 1) | stop("Need a file!")
    f := open(args[1]) | stop("Couldn't open ", args[1])
    wordlist := table()
    lineno := 0

    while line := map(read(f)) do {
      lineno +:= 1
      every word := getword(line) do
        if *word > 3 then {
          # if word isn't in the table, set entry to empty list
          /wordlist[word] := list()
          put(wordlist[word], lineno)
          }
      }
    L := sort(wordlist)
```

```
    every l := !L do {
      writes(l[1], "\t")
      linelist := ""
      # Collect line numbers into a string
      every linelist ||:= (!l[2] || ", ")
      # trim the final ", "
      write(linelist[1:-2])
      }
end

procedure getword(s)
    s ? while tab(upto(&letters)) do {
      word := tab(many(&letters))
      suspend word
      }
end
```

If we run this program on this input:

```
Half a league, half a league,
Half a league onward,
All in the valley of Death
Rode the six hundred.
```

the program writes this output:

```
death   3
half    1, 2
hundred 4
league  1, 1, 2
onward  2
rode    4
valley  3
```

First, note that the main() procedure requires a command-line argument, the name of a file to open. Also, we pass all the lines read through the function map(). This is a function that takes three arguments, the first being the string to map; and the second and third specifying how the string should be mapped on a character by character basis. The defaults for the second and third arguments are the uppercase letters and the lowercase letters, respectively; therefore, the call to map() converts the line just read in to all lowercase.

3.3 Pattern Matching

Pattern matching in Unicon is like string scanning on steroids. Patterns encode as data what sort of strings to match, instead of writing string scanning code to perform the match

directly. A pattern data type allows complex patterns to be composed from pieces. The patterns can then be used in the middle of string scans to give that notation a boost, or used on their own. Arguably, you don't need patterns because anything that can be done to strings, can be done using string scanning. But when the pattern solution is usually shorter, more readable, and runs faster, why wouldn't everyone use them?

Patterns are understood in terms of two different points in time, the point when the pattern is constructed, and the times at which it is used to match a string. Most of the programming work for patterns involves formulating the pattern construction, but most of the computation occurs later on during the pattern matches. The next two subsections describe ways to create many and various complex patterns, while the two notations for using patterns are relatively simple and require little space. All of this only becomes clear with numerous examples that will follow.

3.3.1 Regular Expressions

The literal values of the pattern type are regular expressions, enclosed in less than (<) and greater than (>) symbols. The notation of regular expressions is very old and very famous in computer science, and readers already familiar with them may wish to skim this section.

Within < and > symbols, the normal Unicon interpretation of operators does not apply; instead a set of regular expression operators is used to express simple string patterns concisely. The following are examples of regular expressions.

regular expression	is a pattern that...
<abc>	matches abc
<a|b|c>	matches a or b or c
<[a-c]>	matches a or b or c
<ab?c>	matches a followed optionally by b followed by c
<ab*c>	matches a followed by 0 or more b's followed by c
<a*b*c*>	matches a's followed by b's followed by c's

3.3.2 Pattern Composition

Regular expressions are awesome, but there are many patterns that they cannot express. Unicon has many pattern functions and operators that construct new patterns, often from existing pattern arguments. Sometimes, they simply make it convenient to store parts of patterns in variables that can then be used at various places in larger patterns. Other times, they make it possible to write patterns are not easily written as regular expressions.

composer	constructs a pattern that...
p1 || p2	matches if pattern p1 is followed by pattern p2
p1 .| p2	matches if pattern p1 or pattern p2 matches
p -> v	assigns s to v if an entire pattern match succeeds
p => v	assigns s to v if a pattern match makes it here
.> v	assigns the current position in the match to v
‘v‘	evaluates v at pattern match time
Abort()	causes an entire pattern match to fail immediately
Any(c)	matches a character in cset c
Arb()	matches anything
Arbno(p)	matches pattern p as few (zero or more) times as possible
Bal()	matches the shortest non-null substring with balanced parentheses
Break(c)	matches the substring until a character in cset c occurs
Breakx(c)	matches the substring until a character in cset c occurs
Fail()	fails to match, triggering any alternative(s)
Fence()	fails to match, preventing alternative(s)
Len(i)	matches any i characters
NotAny(c)	matches any one character not in cset c
Nspan(c)	matches 0 or more characters in cset c, as many as possible
Pos(i)	sets the cursor to position i
Rem()	matches the remainder of the string
Span(c)	matches 1 or more characters in cset c, as many as possible
Succeed()	causes the entire pattern match to succeed immediately
Tab(i)	matches from the current position to position i, moving to that location
Rpos(i)	sets the position i characters from the right end of the string
Rtab(i)	matches from the current position to i characters from the right end.

This table summarizes a facility for which an entire chapter could be written. Besides what extra information you find on these functions in this chapter and in Appendix A, Unicon Technical Report 18 covers these constructors in more detail. The concepts generally are translated directly from SNOBOL4, so consulting SNOBOL4 books may also be of use.

Most operands and arguments are required to be of type pattern, with the exception of those marked as type integer (i) or cset (c), and those which are variables (v). If a pattern is required, a cset may be supplied, with semantics equivalent to the pattern which will match any member of the cset. Otherwise if the argument is not a pattern it will be converted to a string; strings are converted to patterns that match the string.

Variable operands may be simple variables or references with a subscript or field operator. The translator may not currently handle arbitrarily complex variable references within patterns. The unevaluated expression (backquotes) operator does handle function calls and simple method invocations in addition to variables.

3.3.3 Pattern Match Operators

A pattern match is performed within a string scanning environment using the unary equals operator, =p. If p is matched at the current position, =p produces the substring matched and moves the position by that amount.

There is also a pattern match control structure, s ?? p, which creates a new string scanning environment for s and looks for pattern p within s, working from left to right.

3.3.4 Scopes of Unevaluated Variables

Since a pattern can be passed as a parameter, variables used in patterns might get used outside of the scope where the pattern was constructed, potentially anywhere in the program. In SNOBOL4 this was not an issue mostly because all variables were global. In Unicon variables are not global by default, and the variables used during pattern matching are evaluated in the scope of the pattern match, not references to locals that existed back during pattern construction time.

To make things more fun, it is impractical to apply the usual rules for implicit variable declaration to variables that do not appear in a procedure body because they are referenced in a pattern that was constructed elsewhere. If you use a variable in a pattern and pass that pattern into a different scope, you must declare that variable explicitly, either as a global or in the scope where it is used in a pattern match.

3.4 String Scanning and Pattern Matching Miscellany

Many topics related to string scanning and pattern matching do not easily fit into one of the preceding sections, but are nevertheless important.

3.4.1 Grep

Grep, an acronym defined variously, is one of the oldest UNIX utilities, which searches files for occurrences of a pattern defined by a regular expression.

Listing 3-2 A simple grep-like program

```
link regexp
procedure main(av)
    local f, re, repl
    every (f|re|repl) := pop(av)
    f := open(f) | stop("can't open file named: ", f)
    while line := read(f) do
      write(re_sub(line, re, repl))
end
```

```
procedure re_sub(str, re, repl)
   result := ""
   str ? {
     while j := ReFind(re) do {
       result ||:= tab(j) || repl
       tab(ReMatch(re))
       }
     result ||:= tab(0)
     }
   return result
end
```

To replace all occurrences of "read|write" with "IO operation" you could type

```
igrep mypaper.txt "read|write" "IO Operation"
```

Since the program has access to the pattern matching operation at a finer grain, more complex operations are possible, this search-and-replace is just an example.

3.4.2 Grammars

Grammars are collections of rules that describe *syntax*, the combinations of words allowed in a language. Grammars are used heavily both in linguistics and in computer science. Pattern matching using a grammar is often called *parsing*, and is one way to match patterns more complex than regular expressions can handle. This section presents some simple programming techniques for parsing context free grammars. Context free grammars utilize a stack to recognize a fundamentally more complex category of patterns than regular expressions can; they are defined below.

For linguists, this treatment is elementary, but introduces useful programming techniques. If you are not interested in grammars, you can skip the rest of this chapter.

A context-free grammar or CFG is a set of rules or *productions*. Here is an example:

```
S -> S S
   | ( S )
   | ( )
```

This grammar has three productions. There are two kinds of symbols, *non-terminals* like S that can be replaced by the string on the right side of a rule, and *terminals* like (and). An application of a production rule is called a derivation. One special non-terminal is called the *start symbol*; a string is accepted by the grammar if there is a sequence of derivations from the start symbol that leads to the string. By convention the start symbol is the first non-terminal in the definition of the grammar. (This grammar only has one non-terminal, and it is also the start symbol.)

This grammar matches all strings of balanced parentheses. The string (()(()())) can be matched by this derivation:

```
S -> (S) -> (SS) -> (()S) -> (()(S)) ->
    (()(SS)) -> (()(()S)) -> (()(()()))
```

Parsing

This section is a discussion of parsers written by hand in Unicon. It would not be right to talk about parsing context free grammars without mentioning the standard tool, iyacc, that the Unicon language translator itself is written in. Iyacc is an industrial strength parser generator, derived from the open source "Berkeley yacc", that generates parsers as .icn source files compatible with Icon and Unicon. Iyacc comes with Unicon distributions and is documented in Unicon Technical Report 3 at http://unicon.org/utr/utr3.pdf.

Unicon can parse grammars in a natural way using matching functions. A production

```
A -> B a D
   | C E b
```

can be mapped to this matching function:

```
procedure A()
   suspend (B() & ="a" & D()) | (C() & E() & ="b")
end
```

This procedure first tries to match a string matched by B, followed the character a, followed by a string matched by D. If D fails, execution backtracks across the ="a" (resetting &pos) and resume B(), which will attempt the next match.

If the sub-expression to the left of the alternation fails, then execution will try the sub-expression on the right, C() & E() & ="b" until something matches - in which case A succeeds, or nothing matches - which will cause it to fail.

Parsers for any CFG can be written in this way. However, this is an expensive way to do it! Unicon's expression evaluation will try all possible derivations trying to match a string. This is not a good way to parse, especially if the grammar is amenable to lookahead methods.

Doing It Better

Many grammars can be parsed more efficiently using well-known techniques - consult a book on compilers for details. Here is one way of parsing a grammar using some of the built-in functions. Consider this grammar for an arithmetic expression:

```
E -> T | T + E
T -> F | F * T
F -> a | b | c | ( E )
```

Listing 3-3 is an Unicon program that recognizes strings produced by this grammar:

Listing 3-3 Expression parser

```
procedure main()
    while line := read() do
      if expr(line) == line then write("Success!")
      else write("Failure.")
end
procedure expr(s)
    s ? {
      while t := tab(bal('+')) do {
        term(t) | fail ; ="+"
        }
      term(tab(0)) | fail
      }
    return s
end
procedure term(s)
    s ? {
      while f := tab(bal('*')) do {
        factor(f) | fail ; ="*"
        }
      factor(tab(0)) | fail
      }
    return s
end
procedure factor(s)
    s ? suspend ="a" | ="b" | ="c" | ( ="(" || expr(tab(bal(')'))) || =")" )
end
```

The interesting procedure here is bal(). With ')' as its first argument, bal() scans to the closing parenthesis, skipping over any parentheses in nested subexpressions, which is exactly what is needed here.

The procedure factor() is written according to the rule in the previous section. The procedures expr() and term() have the same structure. The expr() procedure skips any subexpressions (with balanced parentheses) and looks for a +. We know that this substring is a well-formed expression that is not a sum of terms, therefore, it must be a term. Similarly term() looks for * and it knows that the expression does not contain any * operators at the same nesting level; therefore it must be a factor.

Notice that the procedures return the strings that they matched. This allows us to check if the whole line matched the grammar rather than just an initial substring. Also, notice that factor() uses string concatenation instead of conjunction, so that it can return the matched substring.

Summary

Unicon's string processing facilities are extensive. Simple operations are very easy, while more complex string analysis has the support of a special control structure, string scanning. String scanning is not as concise as regular expression pattern matching, but it is fundamentally more general because the code and patterns are freely intermixed.

Chapter 4

Advanced Language Features

The previous chapters described a wide range of built-in computational facilities that help to make Unicon a great language. This chapter delves into interesting features that make Unicon more than just the sum of its parts. This chapter demonstrates:

- Controlling expressions more precisely
- Using list structures and procedure parameter lists interchangeably
- Holding a generator expression in a value so that its results can be used in different locations throughout the program
- Defining your own control structures
- Evaluating several generator expressions in parallel
- Permuting strings using sophisticated mappings
- Using a more efficient list representation

4.1 Limiting or Negating an Expression

Chapter 1 described generators and the expression mechanism without mentioning many methods for using them, other than every loops. Suppose you wish to generate five elements from a table. If the table has thousands of elements, then you may want to generate just five elements precisely in a situation where generating all the table elements with !T is infeasible. You could write an every loop that breaks out after five iterations, but this solution isn't easy to use within some more complex expressions. The binary backslash operator *expr* \ *i* limits *expr* to at most *i* results. If expr has fewer results, the limitation operator has no effect; once *i* results have been obtained, limitation causes the expression to fail even if it could produce more results.

Unicon does not have a boolean type, so Chapter 1 downplayed the standard logical operators. The alternation operator (|) resembles a short-circuit OR operator, since it

generates its left operand and only evaluates its right operand if the left operand or the surrounding expression fails. The conjunction operator (&) resembles a short-circuit AND operator; it evaluates its left operand, and if that operand succeeds, the result of the conjunction is the result of its right operand. The reserved word not rounds out the boolean-like operators. If *expr* produces no results, then not *expr* will succeed (and produce a null value); if *expr* produces any results, then the not expression fails. The not operator can remedy certain forms of generator confusion. Compare the following two expressions:

```
if not (s == ("good"|"will"|"hunting")) then write("nope")
if (s ~== ("good"|"will"|"hunting")) then write("uh huh")
```

The first expression uses not to ensure that string s is none of the three words. The second expression always writes "uh huh", because any string s that you pick will be not equal (~==) to at least one of the three strings in the alternation. The then part will always execute, which is probably not what was intended.

Note

Negating an == operator is not the same as using a ~== operator!

The conjunction operator $expr_1$ & $expr_2$ has an alternate syntax, a comma-separated list of expressions in parentheses: ($expr_1$, $expr_2$). Any number of expressions may be present; the whole expression succeeds if they all succeed. This looks similar to procedure call syntax because it is similar: a procedure call mutually evaluates all the actual parameters before the procedure is invoked. Invocation allows a string or integer in the "procedure" slot in front of a parenthesized argument list. For a string, as in s(x), a procedure by the name given in s is called; if s had the value "foo", then s(x) is the same as foo(x). For an integer value i, after all arguments are evaluated, the value of the entire expression is the value of the i'th argument.

4.2 List Structures and Parameter Lists

The functions write() and put() take any number of arguments; this flexibility is powerful and convenient. You can write variable argument procedures of your own by ending the last parameter in your procedure declaration with empty square brackets:

```
procedure myfunc(x, y, z[])
```

In this case, instead of throwing away all arguments after the third, the third parameter and all parameters that follow are placed into a newly-constructed list. A call to the above procedure with myfunc(1, 2, 3, 4, 5) causes z to have the value [3, 4, 5].

It is also useful to do the opposite and construct a list of dynamic (or user-supplied) length, and then call a procedure with that list as its parameter list. The apply operator, binary ! performs this feat. If you call write ! L, then all the elements of L are written contiguously on a single line (unless they contain newline characters).

4.3 Co-expressions

A co-expression is an independent, encapsulated thread-like context, where the results of an expression (hopefully a generator!) can be picked off one at a time. Suppose you are writing a program that generates code, and you need something that will generate unique variable names. This expression will generate names:

```
"name" || seq()
```

Function seq() produces a progressive sequence of integers, by default starting at 1, so the whole expression generates "name1", "name2", "name3", ... You can use this expression anywhere in your code; but you may need it in several different places.

There are times when you need to separate the *evaluation* of an expression from its location in the program. The normal way to do this would be a procedure. You can make separate calls to a procedure from different locations in your program, but there is no easy way to use the results from a single instance of a generator in multiple locations. You can put all the results in a list (not a good idea for generators with infinite result sequences) or rewrite the procedure to produce the sequence using separate calls, but this requires static or global variables, and is awkward at best:

```
procedure nameseq()
static i
initial i := 0
    return "name" || (i+:= 1)
end
```

Now, consider the code generating program again. It may need not one name sequence, but two kinds of names: statement labels and temporary variables. It would be poor engineering to write a different procedure for each such sequence. The nameseq() procedure was already cumbersome for so simple a task, but generalizing it for multiple kinds of names makes it *really* messy. By creating a pair of co-expressions, you can capture exactly what is needed with a lot less code:

```
labelname := create ("_L" || seq())
varname := create("_V" || seq())
```

In both cases, create *expr* allocates and initializes an evaluation context plus the memory needed to evaluate expression *expr*, but does not start to evaluate it. Since the co-expression value may be used outside the procedure call where it is created, the evaluation context includes a copy of the local variables and parameters used in the expression. When a co-expression is *activated*, it produces the next value. A co-expression is activated by the @ operator. Each activation of labelname produces the next string in the sequence "_L0", "_L1", "_L2", and so on. Similarly, each activation @varname produces the next in the sequence "_V0", "_V1", "_V2", and so on.

```
loop_name := @labelname
tempvar_name := @varname
```

After a co-expression has produced all its results, further evaluation with @ will fail. The ^ operator produces a new co-expression with the same expression as its argument, but "rewound" to the beginning.

```
c := ^c
```

4.4 User-Defined Control Structures

Control structures are language elements that determine in what order, and how many times, expressions are executed. Co-expressions are used to implement new control structures when procedures that take co-expression parameters control the order and number of times they are activated. Consider a control structure that selects values from the first expression at positions specified by the second. This could be called as:

```
seqsel([create fibonacci(), create primes()])
```

Assuming that you have a pair of generator procedures that produce the Fibonacci numbers (1, 1, 2, 3, 5, 8, 13, ?) and the primes (2, 3, 5, 7, 11, ?), this expression produces the numbers 1, 2, 5, 13, 89, ... Here is the implementation of seqsel():

```
procedure seqsel(a)
    (*a = 2) | stop("seqsel requires a list of two arguments")
    e1 := a[1]; e2 := a[2]
  # position in the first stream we are looking at
    index := 1
    repeat {
      # Get the next index
      (i := @e2 ) | fail
    # Keep getting values from the second expression until
    # we get to the i'th one. If e1 cannot produce that
    # many values, we fail.
      every index to i do
          (value := @e1) | fail
      suspend value
      index := i+1
      }
end
```

Unicon provides a syntactic short-cut for this kind of usage:

```
proc([create e1, create e2, ..., create en])
```

can also be written with curly brackets, as

```
proc{e1, e2, ..., en}
```

4.5 Parallel Evaluation

Co-expressions can be used to evaluate expressions "in parallel". This program writes a table of ASCII characters with the hex, decimal, and octal equivalents:

```
procedure main()
    dec := create(0 to 255)
    hex_dig := "0123456789abcdef"
    hex := create(!hex_dig || !hex_dig)
    oct := create((0 to 3) || (0 to 7) || (0 to 7))
    char := create image(!&cset)
    while write(@dec, "\t", @oct, "\t", @hex, "\t", @char)
end
```

Co-expression dec produces the sequence 0, 1, 2, ... 255; hex the sequence "00", "01", "03", ... "ff"; oct the sequence "001", "002", ... "377"; and char the sequence ..., " ", "!", ..., "A", ... "Z", ..., "a", ... "z", and so forth.

Every invocation of write() results in all the co-expressions being activated once, so they are all run in lock-step, producing this table:

```
0       000     00      "\x00"
1       001     01      "\x01"
2       002     02      "\x02"
...
45      055     2d      "-"
46      056     2e      "."
47      057     2f      "/"
48      060     30      "0"
49      061     31      "1"
50      062     32      "2"
...
90      132     5a      "Z"
91      133     5b      "["
92      134     5c      "\\"
93      135     5d      "]"
94      136     5e      "^"
95      137     5f      "_"
96      140     60      "`"
97      141     61      "a"
...
255     377     ff      "\xff"
```

Parallel evaluation can also be used to assign to a set of variables:

```
ce := create !stat(f)
every (dev | ino | mode | lnk | uid | gid) := @ ce
```

Note

stat() returns file information. It is presented in the next chapter.

Co-expressions can be expensive. This is probably not a good way to assign a series of values to a group of variables but it demonstrates an interesting technique.

4.6 Coroutines

In conventional invocation, procedures have an asymmetric relationship; when control is transferred from the caller to the callee, the callee procedure starts execution at the top. Coroutines have an equal relationship: when control is transferred from one coroutine to another, execution starts from the point that execution was suspended. This process is called resumption. The producer/consumer problem is a good example of procedures that have an equal relationship. Figure 4-1 shows how the control flow between coroutines is different from that of conventional procedures.

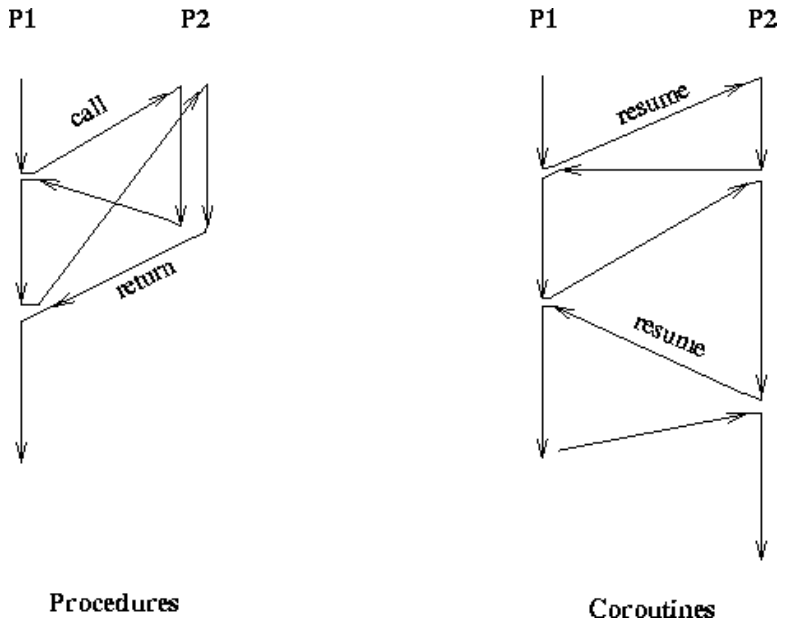


Figure 4-1: The Difference Between Procedures and Coroutines

Can you tell what the next example computes from its integer command-line argument?

Listing 4-1
Producer and Consumer Coroutines

```
procedure main(args)
    C1 := create consumer(args[1])
    C2 := create producer(C1)
    @C2
end

procedure producer(ce)
    x := 1
    repeat {
      val := x ^ 2
      ret := val @ ce | break
      x +:= 1
      }
    @ &main
end

procedure consumer(limit)
    value := @ &source
    repeat {
      # process value
      if value > limit then break
      if value % 2 = 0 then write(value)
      value := retval @ &source
      }
end
```

When producer resumes consumer, it passes value; the consumer passes a return code (retval) back. &source is the coexpression that activated the current co-expression.

Note

This example doesn't mean the producer/consumer problem should always be done with coroutines!

4.7 Permutations

We have seen one usage of map(), where it transformed mixed-case strings to all lowercase. In that type of usage, the first string is the one that we are manipulating, and the other two arguments tell it how the string is to be modified. Interesting results can be achieved by treating the *third* argument as the string to manipulate. Consider this code:

```
s := "abcde"
write(map("01234", "43201", s))
```

What does this code example do? The transformation is: "4" should be mapped to "a", "3" to "b", "2" to "c", "0" to "d", and "1" to "e". When this mapping is applied to "01234", we

get "decba" - a permutation of the string s! It is exactly the permutation that is suggested by the first two arguments of map(). To arrange this sort of permutation, all three strings must be the same size, and there must be no repeated letters in the second string.

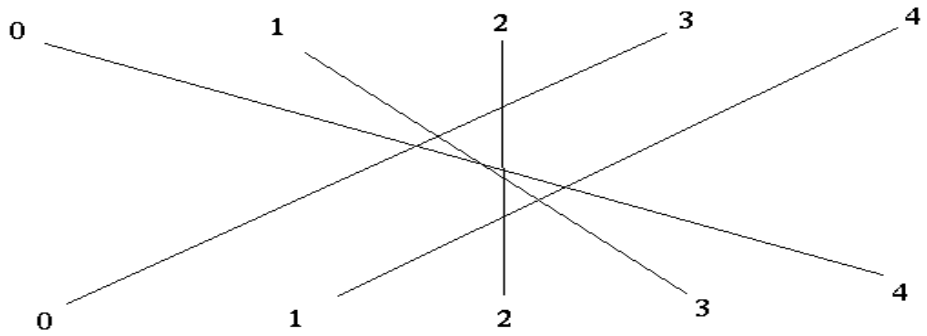


Figure 4-2: Permuting a string with the map() function

Here is an example: In the USA, dates are represented with the month coming first, as in 12/25/1998, but in many other places the day comes first: 25/12/1998. This conversion is, of course, just a permutation; we can do this with a simple map:

```
map("Mm/Dd/XxYy", "Dd/Mm/XxYy", date)
```

Here is another example. Unicon has a built-in random facility, the ? operator. Applied to a string or cset, it returns a random character from the argument; applied to a structure, a random member of that structure; and applied to an integer, a random integer between 1 and that number. This is a very useful feature and allows us to write programs that shuffle cards or run simulations of things like rolling dice.

By default in Unicon, the random sequence generated by ? is different for each run of the program. This is one of the few areas where Unicon is deliberately different from Icon, which uses the same seed each run by default. Icon's semantics is good when debugging, because we want the program to behave predictably while it is broken! However, in most applications that use random numbers, such as games, different runs of the program should create different numbers. The random number seed is keyword &random. It can be assigned a value at the start of main() in order to get Icon-style repeatability. Here's how to assign it a number based on the current date and time. Unicon's default semantics do something similar.

```
&random := map("sSmMhH", "Hh:Mm:Ss", &clock) +
           map("YyXxMmDd", "YyXx/Mm/Dd", &date)
```

The calls to map() remove punctuation characters from the fixed-format strings produced by &clock and &date. The resulting strings of digits are converted to integers, added, and stored as a seed in &random. Now every time the program is run, the random number facility will be initialized with a different number.

4.8 Simulation

A Galton Box demonstrates how balls falling through a lattice of pegs will end up distributed binomially. A simulation of a Galton box combines several of the techniques described previously. Figure 4-3 is an illustration of the program's screen.

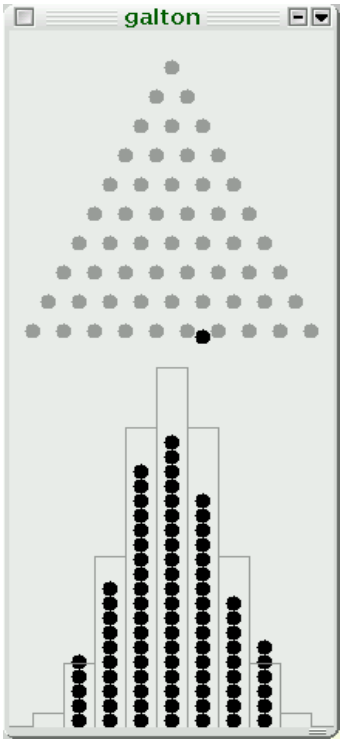


Figure 4-3: A Galton Box Simulation

The simulation's output window is a good example of Unicon's high-level graphics facilities. Graphics is a broad area, discussed in Chapter 7 of this book; the on-line references or the Icon graphics book (Griswold, Jeffery, Townsend 1998) contain substantial additional details. Graphics are part of the system interface. Some of the graphics functions used in this example include:

- FillArc(x,y,width,height) fills an ellipse defined by a bounding rectangle. The shape is filled by the current foreground color and/or fill pattern. The height defaults to be the same as the width, producing a circle. Given additional arguments, FillArc() fills parts of an ellipse similar to pieces of a pie in shape.
- WAttrib("attr") or WAttrib("attr=value"), the generic getter/setter for window attributes. In this case the attributes fg (foreground color) and drawop (raster drawing operation) are set to various colors and reversible output.
- Window("attr=value", ...) opens a window with characteristics specified by string attribute values. The WDelay(t) function waits until t milliseconds have passed. The

WDone() function waits for the user to dismiss the output window by pressing "q" and then terminates the program and closes the window.

Listing 4-2 contains the code for a simplified version of the simulation. A couple elements of the image above are omitted in order to make the example easy to follow. (Both this program and the one that created the screen shot above are included on the book's web site, http://unicon.org/book/)

Listing 4-2 A Simple Galton Box Simulation

```
link graphics
global pegsize, height, width, pegsize2

procedure main(args)
local n, steps := 10
    pegsize := 10
    pegsize2 := pegsize * 2
    n := integer(args[1]) | 100
    setup_window(steps)
    every 1 to n do galton(steps)
    WDone()
end

procedure setup_window(n)

local max, xpos, ypos, i, j
    # Draw the n levels of pegs
    # Pegboard size is 2n-1 square
    # Expected max value of histogram is (n, n/2)/2^n
    # ... approximate with something simpler?
    max := n*n/pegsize
    width := (2*n+1)*pegsize
    height := width + n*n/2*pegsize
    Window("size=" || width || "," || height, "fg=grayish-white")
    WAttrib("fg=dark-grey")
    every i := 1 to n do {
      ypos := i * pegsize2
      xpos := width/2 - (i - 1) * pegsize - pegsize/2
      every j := 1 to i do {
        FillArc(xpos, ypos, pegsize, pegsize)
        xpos +:= pegsize2
        }
      }
    # Set up drawing mode to draw the falling balls
    WAttrib("fg=black")
```

```
    WAttrib("drawop=reverse")
end
# Do it!
procedure galton(n)
local xpos, ypos, oldx, oldy
    xpos := oldx := width/2 - pegsize/2
    ypos := oldy := pegsize
    # For every ball...
    every 1 to n do {
      if ?2 = 1 then xpos -:= pegsize
      else xpos +:= pegsize
      ypos +:= pegsize2
      animate(oldx, oldy, xpos, ypos)
      oldx := xpos; oldy := ypos
      }
    # Now the ball falls to the floor
    animate(xpos, ypos, xpos, ypos + 40)
    animate(xpos, ypos+40, xpos, ypos + 200)
    # Record this ball
    draw_ball(xpos)
end
procedure animate(xfrom, yfrom, xto, yto)
    animate_actual(xfrom, yfrom, xto, yfrom, 4)
    animate_actual(xto, yfrom, xto, yto, 10)
end
# Drawing op is already set to "reverse", and fg colour is black.
procedure animate_actual(xfrom, yfrom, xto, yto, steps)
local x := xfrom, y := yfrom, xstep, ystep, i, lastx, lasty
    xstep := (xto - xfrom)/steps
    ystep := (yto - yfrom)/steps
    every i := 1 to steps do {
      lastx := x; lasty := y
      FillArc(x, y, pegsize, pegsize)
      WDelay(1)
      FillArc(x, y, pegsize, pegsize)
      x +:= xstep; y +:= ystep
      }
end
procedure draw_ball(x)
static ballcounts
initial ballcounts := table(0)
    ballcounts[x] +:= 1
    FillArc(x, height-ballcounts[x]*pegsize, pegsize, pegsize)
end
```

4.9 Arrays

Unicon uses two different representations for a list: one of them, an *array*, is optimized for fast access to the list-elements or for interfacing with another language that expects a contiguous vector of values. The size of an array is fixed when it is first created and consists entirely of integer elements (or entirely of reals). If any of the array elements is assigned a value of a different type, or the size of the array changes (e.g. by using delete or push etc.), the array is automatically converted into the other form — the more general representation of a list.

The general list representation occupies approximately twice as much storage as an array of the same size and type on most implementations of Unicon.

To create an array, call list(n, x), where x is an integer or real value and n > 0. If you want to create a more general list, rather than an array (perhaps you know the list will change its size in the future and you wish to avoid the cost of converting the array when it does) then use list comprehension: i.e. instead of

```
l := list(20000, 0)
```

use

```
l := [: (| 0) \ 20000 :]
```

list comprehension never produces an array.

Although the conversion from the array form to the more general representation is automatic, when it is required, the conversion in the other direction is not. To convert a general list into an array it may be supplied as the only parameter to the list function. If the list meets the requirements (either all the elements are integers or all of them are reals) the function will return a new array that consists of the values in the list. If the list does not meet the requirements, the function returns the list parameter unchanged.

Summary

Unicon is particularly powerful when language features are combined. The ability to combine features in interesting ways is the result of its novel expression semantics. Co-expressions add substantial value to the concept of generators, although most programs use them only sparingly. They fit well into a philosophy that says that simple things should be easy to do . . . and complex things should be easy to do as well.

Chapter 5

The System Interface

The system interface is Unicon's connection to the outside world, defining input/output interactions with the operating system. This chapter shows how to

- Manipulate files, directories, and access permissions
- Launch and interact with other programs
- Handle abnormal events that would otherwise terminate your program
- Write Internet client and server applications.

5.1 The Role of the System Interface

Unicon's predecessor Icon is highly portable; it runs on everything from mainframes to Unix machines to Amigas and Macs. This platform independence is both a virtue and a limitation. Icon takes a greatest common denominator approach to the system interface. Icon programs run with no source modifications across platforms, but with little access to the underlying system. Icon historically could not be used easily for many tasks such as system administration or client/server programming. Both the Icon graphics facilities, and now the Unicon system interface, "raise the bar" of what portable facilities programmers can expect to be provided by their programming language, at the cost of making it more difficult to port the language to new platforms.

The interface described in this chapter relies on underlying standards including ANSI C's standard library, and the IEEE Portable Application Standards Committee's POSIX operating system standard (http://www.pasc.org). Unicon relies on standards, but is simpler and higher level. It is also less platform-specific than the POSIX standard. The goal was to define facilities that can be implemented to a great extent on all modern operating systems. Non-POSIX Unicon implementations may provide a subset of the functionality described in this chapter, but important facilities such as TCP/IP Internet communications are ubiquitous and warrant inclusion in the language definition. So far the complete

Unicon system interface is implemented for Linux, Solaris, and Windows; the challenge to port these facilities to all platforms on which they are relevant and useful now rests with Unicon's user community.

5.2 Files and Directories

The file type is used for any connection between a program and an external piece of hardware. In reality, a file is a reference to resources allocated by the operating system for the purpose of input or output. Different kinds of files support different operations, but most files support the basic functions given in this section.

Files are commonly used to manipulate named repositories of data on a storage device. The contents of files exist independent of the program that creates them, and persist after that program finishes. To read data from a file or save data to a file, the functions read() and write() are often used. These functions by default use special files denoted by the keywords &input and &output, respectively. There is a third file keyword, &errout, that refers to the location to which the program should write any error messages. Unless the files were redirected, they refer to the keyboard and the display. If you pass read() or write() a value of type *file* as an argument, the operation is performed on that file. The function open() creates a value of type file:

```
f := open("myfile.txt", "w")
write(f, "This is my text file.")
```

The open() function takes two parameters: a file name and a mode. The default mode is "r" for reading; the example above uses mode "w" for writing. Other modes denote other kinds of system interfaces. They are described in later sections.

The read() function reads and returns a line of text, removing the line terminator(s). Function write() similarly adds a line terminator after writing its arguments. Another way to read lines is via the generate operator, unary !. The expression !f generates the lines of file f, so every put(L, !f) puts the lines of f into list L.

On systems with multi-character line terminators, appending an extra letter to the mode parameter of open() indicates whether newlines are to be translated (mode "t") or untranslated (mode "u"). Text files should be translated, while binary files should not. The default is to translate newlines to and from operating system format.

Besides read() and write(), which always process a single line of text, the functions reads(f, i) and writes(f, s, ...) read (up to i characters) and write strings to a file. These functions are not line-oriented and do no newline processing of their own, although they still observe the translation mode on systems that use one.

When operations on a file are complete, close the file by calling close(f). The only exceptions are the standard files, &input, &output, and &errout; since you didn't open them, don't close them. For the rest, most operating systems have a limit on the number of files

that they can have open at any one time, so not closing your files can cause your program to fail in strange ways if you use a lot of files.

Directories

A directory is a special file that contains a collection of named files. Directories can contain other directories to form a hierarchical structure. The chdir() function returns the current working directory as an absolute path name. Called with a string argument, the chdir(*dirname*) function sets the current working directory to dirname. The call open(*dirname*) opens a directory to read its contents. Directories can only be opened for reading, not for writing. Every read() from a directory returns the name of one file. Directory entries are not guaranteed to be in any order. The expression every write(!open(".")) writes the names of the files in the current directory, one per line. It is not good practice to call an open() that you don't close().

The mkdir(s) function creates a directory. An optional second parameter specifies access permissions for the directory; controlling file ownership and access is discussed below. Files or directories can be renamed with rename(s1,s2). Renaming does not physically move the file, so if s1 and s2 denote locations on different hardware devices or file systems then rename() will fail, and you will need to "copy and then delete" the file. Individual files or directories are removed with remove(s). Only empty directories may be removed. To remove an entire directory including its contents:

```
procedure deldir(s)
    f := open(s)
    every remove( s || "/" || ("." ~== (".." ~== !f)))
    close(f)
    remove(s)
end
```

How would you change this function to delete subdirectories? You might be able to devise a brute force approach using what you know, but what you really need is more information about a file, such as whether it is a directory or not.

Obtaining file information

Metadata is information about the file itself, as opposed to information stored in the file. Metadata includes the owner of the file, its size, user access rights, and so forth. This information is produced by the stat() system call. Its argument is the name of a file or (on UNIX systems only) an open file. The stat() function returns a record with the information about the file. Here is a subset of ls, a UNIX program that reads a directory and lists information about its files. Keyword &errortext contains information about the most recent error that resulted in an expression failure; it is written if opening the directory fails. This

version of ls only works correctly if its arguments are the names of directories. How would you modify it, using stat(), to take either ordinary file names or directory names as command line arguments?

```
link printf
procedure main(args)
    every name := !args do {
      f := open(name) | stop(&errortext, name)
      L := list()
      while line := read(f) do
        push(L, line)
      every write(format(stat(n := !sort(L)), n))
      }
end
procedure format(p, name)
  s := sprintf("%-7d %-5d %s %-4d %-9d %-9d %-8d %s %s",
       p.ino, p.blocks, p.mode, p.nlink, p.uid, p.gid, p.size,
       ctime(p.mtime)[5:17], name)
    if p.mode[1] == "l" then
      s ||:= " -> " || \(p.symlink)
    return s
end
```

The record returned by stat() contains many fields. Not all file systems support all of these fields. Two of the most important portable fields are size, the file size in bytes, and mtime, the file's last modified time, an integer that is converted into a human readable string format by ctime(i). Another important field is mode, a string that indicates the file's type and access permissions. Its first letter (mode[1]) is "-" for normal files, "d" for directories, and some file systems support additional types. The other characters of the mode string are platform dependent. On UNIX there are nine letters to encode read, write, and execute permissions for user, group, and world, in the format: "rwxrwxrwx". On a classic Windows FAT file system, there is only "rwa" to indicate the status of hidden, read-only, and archive bits (if it is set, the system bit is indicated in mode[1]).

Some file systems support duplicate directory entries called *links* that refer to the same file. In the record returned by stat(), a link is indicated by a mode[1] value of "l". In addition, field nlinks ("number of links") will be > 1 and/or field symlink may be the string filename of which this file is an alias. Appendix E includes information on each platform's support for the mode field, as well as stat()'s other fields.

Controlling file ownership and access

The previous section shows how different platforms' file systems vary in their support for the concepts of file ownership and access. If the system supports ownership, the user and

group that own a file are changed by calling chown(fname, user, group). The chown() function only succeeds for certain users, such as the super user. User and group may be string names, or integer user identity codes on some platforms.

File access rights are changed with chmod(fname, mode). The chmod() function only succeeds for the owner of a given file. The mode is a nine-letter string similar to stat()'s mode field, or an octal encoding of that information (see Appendix E).

Another piece of information about files is called the *umask*. This is a variable that tells the system what access rights any newly created files or directories should have. The function call umask("rwxr-xr-x") tells the system that newly created directories should have a permission of "rwxr-xr-x" and files should have permissions of "rw-r--r--". The mkdir(s, mode) function takes an optional mode parameter, which can override the umask for for newly created directories. Ordinary files are never given execute permission by the system, it must be set explicitly with chmod().

File locks

Files can be locked while a program is updating some information. If the contents of the file are in an inconsistent state, other programs may be prevented from reading (or especially writing) the file. Programs can cooperate by using file locks:

```
flock(filename, "x")
```

The first call to flock() creates a lock, and subsequent calls by other programs will block, waiting till the writing program releases its lock. The flag "x" represents an *exclusive* lock, which should be used while writing; this means no other process can be granted a lock. For reading, "s" should be used to create a shared lock so that other programs that are also just reading can do so. In this way you can enforce the behavior that only one process may open the file for writing, and all others will be locked out; but many processes can concurrently open the file for reading.

5.3 Programs and Process Control

Unicon's system interface is similar but higher level than the POSIX C interface. An include file posix.icn defines constants used by some functions. Include files are special code, consisting mainly of defined symbols, intended to be textually copied into other code files. They are handled by the preprocessor, described in Appendix A. To include posix.icn in a program, add this line at the top of your program:

```
$include "posix.icn"
```

When a system call fails, the integer keyword &errno indicates the error that occurred. As seen earlier, a human-readable string is also available in &errortext. Error codes (such

as EPERM, or EPIPE) are defined in posix.icn; &errno can be compared against constants like ENOENT. In general, however, human readers will prefer to decipher &errortext.

In the discussion to follow, a *program* is the code, while a *process* is such a program in execution. This distinction is not usually important, but for network applications it matters, since the same program can run in multiple processes, and a process can change the program that it is running.

Signals

A signal is an asynchronous message sent to a process either by the system (usually as a result of an illegal operation like a floating point error) or by another process. A program has two options to deal with a signal: it can allow the system to handle it in the default manner (which may include termination of the process) or it can register a function, called a signal handler, to be run when that signal is delivered.

Signals are trapped or ignored with the trap(s, p) function. Argument s is the string name of the signal. The signal names vary by platform; see Appendix E. You can trap any signal on any machine; if it is not defined it will be ignored. For example, Linux systems don't have a SIGLOST. Trapping that signal has no effect when a program runs on Linux. The trap() function's second argument is the procedure to call when the signal is received. The previous signal handler is returned from trap() so it can be restored by a subsequent call to trap(). The signal handler defaults to the default provided by the system. For instance, SIGHUP is ignored by default but SIGFPE will cause the program to terminate.

Here is an example that handles a SIGFPE (floating point exception) by printing out a message and then runs the system default handler:

```
global oldhandler
    ...
    trap("SIGFPE", sig_ignore)
    oldhandler := signal("SIGSEGV", handler)
    ...
  # restore the old handler
    trap("SIGSEGV", oldhandler)
end

procedure sig_ignore(s); end
procedure handler(s)
    write(&errout, "Got signal ", s)
    (\oldhandler)(s)
            # propagate the signal
end
```

Launching programs

Many applications execute other programs and read their results. In many cases, the best way to do this is to call open() with mode "p" (pipe) to launch a command. In mode "p" the string argument to open() is not a filename, it is an entire command string. Piped commands opened for reading (mode "p" or "pr") let your program read the command's standard output, while piped commands open for writing (mode "pw") allow your program to write to the command's standard input.

The more general function system(x,f1,f2,f3,mode) runs an external command (argument x) with several options. If x is a list, x[1] is the command to execute and the remaining list elements are its command line arguments. If x is a string, it is parsed into arguments separated by spaces. Arguments with spaces in them may be escaped using double quotes. A program that calls system() normally waits for the launched program to complete before continuing, and system() returns the integer status of the completed command. If s ends in an ampersand (&) or the optional mode argument is 1 or "nowait", system() does not wait for the command to complete, but instead launches the command in the background and returns an integer process id. The system() function takes three optional file arguments that specify redirected standard input, output, and error files for the launched program.

Using file redirection and pipes

One common scenario is for a program to run another program but with the input and output redirected to files. On command-line systems like the Unix shells or the MS-DOS command prompt, you may have used redirection:

```
prog < file1
```

File redirection characters and other platform-dependent operations are supported in the command string passed to system(), as in the following system() call:

```
system("prog < file1")
```

Pipes to and from the current program are nicely handled by the open() function, but sometimes the input of one program needs to be connected to the output of another program. You may have seen uses like this:

```
prog1 | prog2
```

The pipe() function returns a pair of open files in a list, with the property that anything written to the second file will appear on the first. Here's how to hook up a pipe between two programs:

```
L := pipe() | stop("Couldn't get pipe: ", &errortext)
system("prog1 &", , L[2])
system("prog2 &", L[1])
close(L[1])
close(L[2])
```

Process information

The integer process identity can be obtained with getpid(). The user id of the process can be obtained with getuid() if the platform supports it. Calls to obtain additional information such as group identity on some platforms are described in Appendix E.

A parent process may want to be notified when any of its children quit (or change status). This status can be obtained with the function wait(). When a child process changes state from "running" to either "exited" or "terminated" (and optionally "stopped"), wait() returns a string of the form

```
pid:status:arg:core
```

The ":core" will only be present if the system created a core file for the process. The status can be any of "exited", "terminated" or "stopped". The arg field is either: a) the exit status of the program if it exited; or b) the signal name if it was terminated. Typically wait() will be used in the handler for the SIGCHLD signal which is sent to a process when any of its children changes state.

The arguments to wait() are the pid of the process to wait for and any options. The default for pid is to wait for all children. The options may be either "n", meaning wait() should not wait for children to block but should return immediately with any status that's available, or "u", meaning that any processes that stopped should also be reported. These options may be combined by using "nu".

The select() system call

Some programs need to be able to read data from more than one source. For example, a program may have to handle network traffic and also respond to the keyboard. The problem with using read() is that if no input is available, the program will block and will not be able to handle the other stream that may in fact have input waiting on it. To handle this situation, you can use the function select(x1,x2,...i). The select() function tells the system which files you are interested in reading from, and when input becomes available on any of those sources, the program will be notified. The select() function takes files or lists of files as its arguments, and returns a list of all files on which input is waiting. If an integer argument is supplied, it is a timeout that gives the maximum milliseconds to wait before input is available. If the timeout expires, an empty list is returned. If no timeout is given, the program waits indefinitely for input on one of the files.

```
while *(L := select(f1, f2, f3, timeout)) = 0 do
    handle_timeout()
(&errno = 0) | stop("Select failed: ", &errortext)
every f := !L do {
    # Dispatch reads pending on f
    ...
    }
```

When using select() to process input from multiple files, you may need to pay some attention to avoid blocking on any one of your files. For example the function read() waits until an entire line has been typed and then returns the whole line. Consider this code, which waits for input from either a file (or network connection) or a window designated by keyword &window:

```
while L := select(f, &window) do
  if !L === f then c := read(f)
```

Just because select() has returned doesn't mean an entire line is available; select() only guarantees that at least one character is available. The command shell log application in Chapter 14 shows the usage of select(). Another primary application area for select() is network programming, described later in this chapter. For network connections, the function reads(f, i) will return as soon as it has some input characters available, rather than waiting for its maximum string size of i. But if no input is available, reads() blocks.

Non-blocking input and the ready() function

The function ready(f, i) is like reads(f, i) except that it is non-blocking, that is, it returns immediately with up to i bytes if they are available, but it does not wait around. It is ideal for use with select() and in situations where a server or client needs to interact with multiple remote connections.

5.4 Networking

Unicon provides a very high-level interface to Internet communications. Applications with custom communications use one of the major Internet applications protocols, TCP and UDP. An higher level interface to several popular Internet protocols such as HTTP and POP is provided by means of Unicon's messaging facilities.

TCP

A TCP connection is a lot like a phone call: to make a connection you need to know the address of the other end, just like a phone number. For TCP, you need to know the name

of the machine to connect to, and an address on that machine, called a *port*. A server listens for connections to a port; a client sends requests to a port. Also, there are two kinds of ports, called "Internet Domain" and "Unix Domain." The distinction is beyond the scope of this book; we will just mention that Internet Domain ports are numbers, and Unix Domain ports look like files. Also, a connection to a Unix domain port can only be made from the same machine, so we will not consider the Unix domain further here.

A call to open() with mode "n" (network) requests a network connection. The first argument to open() is the network address, a host:port pair for Internet domain connections, and a filename for Unix domain sockets. If the address contains no host name and therefore starts with ":", the socket is opened on the same machine. The value returned by open() is a file that can be used in select() and related system functions, as well as normal reading and writing.

A client uses mode "n" with open() to open a connection to a TCP server. Here is a simple version of the Internet "finger" program:

```
procedure main(argv)
   local fserv := getserv("finger") |
     stop("Couldn't get service: ", &errortext)
   name := argv[1]
   host := ""
   argv[1] ? {
     name := tab(find("@")) & ="@" & host := tab(0)
     }
   if *host > 0 then write("[", host, "]")
   f := open(host || ":" || fserv.port, "n") |
     stop("Couldn't open connection: ", &errortext)

   write(f, name) | stop("Couldn't write: ", &errortext)
   while write(read(f))
end
```

Notice the use of getserv(). The posix_servent record it returns includes fields for the name, aliases, port, and protocol used by the Internet service indicated in getserv()'s argument. The Internet protocols specify the ports to be used for various services; for instance, email uses port 25. Instead of having to remember port numbers or hard-coding them in our program, we can just use the name of the service and have getserv() translate that into the port number and protocol we need to use.

To write a server, all we need to do is add "a" (accept) to the mode after the "n" in open(). Here is a simple TCP server that listens on port 1888:

```
procedure main()
   while f := open(":1888", "na") do {
     system("myserverd -servicerequest &", f, f)
```

```
        close(f)
        }
    (&errno = 0) | stop("Open failed: ", &errortext)
end
```

The call open(":1888", "na") blocks until a client connects. The returned value is a file that represents the network connection. This example server responds to requests by launching a separate process to handle each request. The network connection is passed to myserverd as its standard input and output, so that process had better be expecting a socket on its standard I/O files, and handle it appropriately. This works on UNIX; on other platforms a different approach is needed.

Launching a separate process to handle requests is standard operating procedure for many Internet servers, but besides the portability concerns, it uses a lot of memory and CPU time. Many servers can be implemented in a single process. Chapter 15 includes an example of such a server. Mode "na" is less than ideal for one-process servers: it only supports one connection at a time. When waiting for a new connection, the process is not doing any computation, and when servicing a connection, the program is not listening for any other connection requests. Unless each connection is of short duration, the server will appear to be down, or appear to be unacceptably slow, to anyone trying to connect while an existing request is being processed.

Determining IP numbers

Many programs need the IP number of the machine they are talking to. Given a network connection f, image(f) will show the IP address and port of the client machine that is connected (this is sometimes called the *peername*).

Some programs need to know their own IP number, but each machine can have several IP numbers, one for each kind of physical network hardware in operation. To obtain a list of local IP numbers, a program can read the output of /sbin/ifconfig (UNIX) or ipconfig (Windows). To find the IP number used for a particular network connection n, on some platforms you can call gethost(n), which returns a string with the IP number and port used by the local machine for a given connection.

If you do determine your IP number in one of these ways, it is usually not the number seen by the world, because most devices are connecting through some form of network address translation. To see the number that the world sees, you have to connect to someone else and ask them to tell you what IP number they see you at.

```
procedure main(argv)
    n := open(argv[1],"n") |
        stop("can't connect to ", argv[1]|"missing host")
    write("connected to: ", image(n)[6:-1])
    write("using: ", gethost(n))
end
```

Non-blocking network opens

Servers need to never block. The call open(":port","nl") creates a *listener* on the specified port, without waiting around for someone to actually connect to it. The network file returned from open() is not open for reading or writing, so it is not good for much...yet. About the only thing you can do with such a file is include it (along with any other network connections you have going) as an argument in a call to select(). If a listener matches a current connection request, select() converts it into a regular network connection as per mode "na".

In addition to non-blocking servers' listener connections, in the real-world clients need a way to do an almost non-blocking connection as well. TCP connections over long distances take a highly variable amount of time, but most clients do not want to "freeze" for a couple of minutes while the connection attempt times out. The network client versions of open() allows an optional third parameter to supply it with a timeout value, in milliseconds.

UDP

UDP is another protocol used on the Internet. TCP is like a phone call: all messages you send on the connection are guaranteed to arrive in the same order they were sent. UDP on the other hand is more like the postal service, where messages are not guaranteed to reach their destination and may not arrive in the same order they were sent in. Messages sent via UDP are called *datagrams*. It's a lot cheaper (faster) to send UDP messages than TCP, though, especially if you are sending them across the Internet rather than to a local machine. Sending a postcard is usually cheaper than a long distance phone call!

UDP datagrams can be sent either with an open()/writes() pair, or with send(). Typically a server sends/receives on the same socket so it will use open() with read() and write(). A client that only sends one or two datagrams uses send()/receive().

The following example provides a service called "rdate" that allows a program to ask a remote host what time it has. The server waits for request datagrams and replies with the date and time. The "u" flag added to the mode in the call to open() signifies that a UDP connection should be used. The function receive() waits for a datagram to arrive, and then it constructs a record having the address the message came from and the message in it. The server uses the address to send the reply.

```
f := open(":1025", "nua")
while r := receive(f) do {
   # Process the request in r.msg
   ...
   send(r.addr, reply)
   }
```

The record returned by receive() has two fields: the addr field contains the address of the sender in "*host*:*port*" form, and the msg field contains the message.

To write a UDP client, use mode "nu" Since UDP is not reliable, the receive() is guarded with a select(); otherwise, the program might hang forever if the reply is lost. The timeout of five seconds in the call to select() is arbitrary and might not be long enough on a congested network or to access a very remote host. Notice the second argument to getserv(); it restricts the search for Internet service information to a particular network protocol, in this case UDP.

```
procedure main(args)
    (*args = 1) | stop("Usage: rdate host")
    host := args[1]
    s := getserv("daytime", "udp")
    f := open(host||":"||s.port, "nu") |
      stop("Open failed: ", &errortext)
    writes(f, " ")
    if *select(f, 5000) = 0 then
      stop("Connection timed out.")
    r := receive(f)
    write("Time on ", host, " is ", r.msg)
end
```

From these examples you can see that it is relatively easy to write programs that use Internet communication. But TCP and UDP are very general, somewhat low-level protocols; most programs employ a higher-level communication protocol, either by defining their own, or using a standard protocol. If you need to define your own Internet protocol, you can do it on top of TCP or UDP; if your program needs to use a standard Internet protocol, you should check first to see if the protocol is built-in to the language as part of the messaging facilities, described in the next section.

Secure Sockets

Unicon offers secure sockets to encrypt data when using the TCP/UDP protocols.

Example server code:

```
procedure main()
    sock := open("localhost:6600", "nae", "key=server.key", "cert=server.crt", "ca=ca.crt") |
                stop(&errortext)
    select(sock) # wait for input
    msg := ready(sock)
    write("Message from client:", msg)
    writes(sock, msg) # echo back the same message to the client
    close(sock)
end
```

Example client code:

```
procedure main()
    sock := open("localhost:6600", "ne") | stop(&errortext)
    writes(sock, "Hello SSL Socket")
    select(sock) # wait for input
    msg := ready(sock)
    write("Message from server:", msg)
    close(sock)
end
```

5.5 Messaging Facilities

Unicon's messaging facilities provide higher level access to many popular Internet protocols. A call to open() using mode "m" establishes a messaging connection. The filename argument to a messaging connection is a URI (Uniform Resource Indicator) that specifies the protocol, host machine, and resource to read or write. The protocols implemented thusfar are HTTP, HTTPS, Finger, SMTP, and POP. Extra arguments to open() are used to send headers defined by the protocol. For example, the call

```
open("mailto:unicon-group", "m", "Reply To: jeffery@cs.uidaho.edu")
```

supplies a Reply To field as a third parameter to open() on an SMTP connection.

Header fields from the server's response to a connection are read by subscripting the message connection value with a string header name; an example is in the next section.

HTTP and HTTPS

HTTP is used to read or write to Web servers; the content read or written typically consists of HTML text. The following program, httpget.icn, fetches a remote file specified by a URI on the command line, and saves it as a local file. The Icon Program Library module basename is used to extract the filename from the URI.

```
link basename

procedure main(argv)
    f1 := open(argv[1],"m")
    f2 := open(basename(argv[1]),"w")
    while write(f2, read(f1))
end
```

This example retrieves the actual data for a successful HTTP request; for a failed request the connection returns no data, appearing to be an empty file. Programs can check the HTTP status code in order to determine the nature of the problem. Status codes and other metadata from HTTP header lines are inspected by subscripting the connection with

the desired header. For example, in the previous program, checking f1["Status-Code"] would allow us to detect HTTP errors, and f1["Location"] would allow us to find the new location of the file, if the HTTP server had informed us that the file had moved. You can retrieve this status information on a remote file without retrieving the file itself. If you open a URI with mode "ms" instead of "m", an HTTP request for header information is made, but no data is retrieved.

HTTPS is HTTP communicated over a secure-socket encryption layer. The encryption requires the use of encryption keys and certificates to validate the authenticity of the remote site. Certificates are typically stored in a directory or a database of some kind. Mode "m-" may be used in open() to skip the validation of the certificate provided by the remote site.

SMTP and POP

SMTP is used to send a mail message. Mail is delivered via an SMTP server machine on which the user must have a valid account. These default to the current user on the current host machine. Two environment variables UNICON_SMTPSERVER and UNICON_USERADDRESS can be set to override the default behavior.

POP is used to read mail messages. POP is the least file-like of these protocols. A POP connection feels like a list of strings, each string containing an entire mail message, rather than a simple sequence of bytes. Function read() and operator ! produce entire messages which in general contain many newline characters. POP messages may be removed by either delete() or pop(); messages are buffered in such a way that message removal on the server occurs when the connection is explicitly and successfully closed.

Here's a simple program that illustrates the use of messaging to get email from a POP server. It gets messages from the server without deleting them and, for every message, prints out who the message is from as well as the subject line.

```
procedure main(argv)
   user := argv[1] | getenv("USER") | stop("no user")
   passwd := argv[2] | getenv("PASSWD") | stop("no password")
   server := argv[3] | getenv("MAILHOST") | "mailhost"
   conn := open("pop://"||user||":"||passwd||"@"||server, "m") |
     stop("couldn't connect to the POP server ", server)
   every msg := !conn do msg ?
     while line := tab(find("\n")) do
       if =("From: " | "Subject: ") then write(line)
   close(conn)
end
```

You should improve the password handling in this program if you use it! Chapter 14 includes another example use of Unicon's POP messaging facilities: a spam filter.

5.6 Tasks

A *task* is an executing program within the Unicon virtual machine. A single task called the *root* is created when the interpreter starts execution. Additional tasks are created dynamically as needed.

The tasking facilities allow Unicon programs to load, execute, communicate with, and control one another, all within a single instantiation of the Unicon interpreter.

Although Unicon programs can use the tasking facilities to load and execute any number of other programs within the same interpreter space, the tasking facilities do not introduce a concurrent programming construct nor do they include special support for multiprocessor hardware. Their domain is that of high-level language support for programs that benefit from or require a tighter coupling than that provided by inter-process communication; that is, programs that access each other's state extensively.

Co-expressions provide the tasking facilities' program execution model, and co-expression activation serves as the communication mechanism. The extensions are general enough to be useful in a wide variety of contexts. For example, programs that use the multi-tasking interface can communicate directly without resorting to external files or pipes.

At the language level, the tasking facilities include several built-in functions and keywords, but no new types, declarations, or control structures. Several existing functions have been extended to offer additional support for the multi-tasking environment. Separate memory allocation regions are established for each task.

Preliminary terminology

Before describing the task model, a few definitions are needed. These definitions pertain to regions of memory referenced by programs during execution.

A *name space* is a mapping from a set of program source-code identifiers to a set of associated memory locations [Abelson85]. Icon programs have a global name space shared across the entire program and various name spaces associated with procedures. Procedures each have a static name space consisting of memory locations shared by all invocations of the procedure and local name spaces private to each individual invocation of the procedure. When a co-expression is created, a new local name space is allocated for the currently executing procedure, and the current values of the local variables are copied into the new name space for subsequent use by the co-expression.

An Icon program has an associated *program state* consisting of the memory associated with global and static name spaces, keywords, and dynamic memory regions. Similarly, a co-expression has an associated *co-expression state* consisting of an evaluation stack that contains the memory used to implement one or more local name spaces. Co-expressions in an Icon program share access to the program state and can use it to communicate.

Tasks as extended co-expressions

A task consists of a main co-expression and zero or more child co-expressions that share a *program state* consisting of the global and static name spaces, keywords, and dynamic memory regions. At the source-language level, tasks are loaded, referenced, and activated solely in terms of one of their member co-expressions; the task itself is implicit. Co-expressions share access to the program state and can use it to communicate. Unicon provides the task model as a mechanism for multi-tasking, but does not predefine the policy; matters such as the scheduling algorithm used and whether multi-tasking is co-operative or pre-emptive are programmable at the user level in terms of co-expression activations.

Task creation

A task can create other tasks. The function

```
load(s, L, f1, f2, f3, i1, i2, i3)
```

loads an icode file [Gris86] specified by the file name s, creates a task for it and returns a co-expression corresponding to the invocation of the procedure main(L) in the loaded icode file. L defaults to the empty list. Unlike conventional Icon command-line argument lists, the argument list passed to load() can contain values of any type, such as procedures, lists, and tables in the calling task.

The task being loaded is termed the child task, while the task calling load() is termed the parent. The collection of all tasks forms a tree of parent-child relationships.

f1, f2, and f3 are file arguments to use as &input, &output, and &error in the loaded task; &input, &output, and &error default to those of the loading task. i1, i2, and i3 are three integer arguments that supply initial region sizes for the task's block, string, and stack memory areas, respectively. i1 and i2 default to 65000, while i3 defaults to 20000 (the defaults may be changed by the environment variables BLKSIZE, STRSIZE, and MSTKSIZE).

Running other programs

A co-expression created by load() is activated like any other co-expression. When activated with the @ operator, the child task begins executing its main procedure. Unless it suspends or activates &source, the child task runs to completion, after which control is returned to the parent. Chapter 5 presents an alternative means of executing a child with which the parent retains control over the child as it executes.

This default behavior is illustrated by the program seqload, which loads and executes each of its arguments (string names of executable Unicon programs) in turn. Each of the strings passed on the command line and extracted from the list using the element-generation operator, !) is passed in turn to load(), which reads the code for each argument and creates a task in which to execute the loaded program. The tasks are then executed one-by-one

by the co-expression activation operator, @. There is nothing special about this example except the semantics of the load() function and the independent execution environment (separate global variables, heaps, and so forth), that load() provides to each task.

```
# seqload.icn
procedure main(arguments)
    every @load(!arguments)
end
```

For example, if three Icon programs whose executable files are named translate, assemble, and link are to be run in succession, the command

```
seqload translate assemble link
```

executes the three programs without reloading the interpreter for each program.

Data access

Although tasks have separate program states, they reside in the same address space and can share data; values can be transmitted from task to task through main()'s argument list, through co-expression activation, or by use of event monitoring facilities described in Chapter 10. In the following pair of programs, the parent receives a list value from the child and writes its elements out in reverse order.

```
# parent.icn
procedure main()
    L := @ load("child")
    while write(pull(L))
end

# child.icn
procedure main()
    L := [ ]
    while put(L, read())
    return L
end
```

Access Through Task Argument Lists

The following program takes its first argument to be an Icon program to load and execute as a child, sorts its remaining arguments, and supplies them to the child program as its command-line arguments (pop() and sort() are Icon built-in functions that extract the first list element and sort elements, respectively):

```
procedure main(arguments)
    @load(pop(arguments), sort(arguments))
end
```

Argument lists allow more sophisticated data transfers; the seqload example presented earlier can be extended to transmit arbitrary structures between programs using argument lists in the following manner. As in seqload, each string naming an executable Icon program is passed into load(), and the resulting task is activated to execute the program. In this case, however, any result that is returned by one of the programs is assigned to local variable L and passed to the next program in the list via the second argument to load().

```
# seqload2.icn
procedure main(arguments)
    every program := !arguments do
        L := @load(program, L)
end
```

The net effect of seqload2.icn is similar to a UNIX pipe, with an important difference: Arbitrary Icon values can be passed from program to program through the argument lists. This capability is more interesting in substantial multipass tools such as compilers, where full data structures can be passed along from tool to tool instead of writing out text encodings of the structures to a file.

Inter-task Access Functions

Several built-in functions provide inter-task access to program data. These functions are usable in any multi-task Unicon context, but are especially useful in program execution monitoring, discussed in Chapter 9.

For example, the variable() function takes a co-expression value as an optional second argument denoting the task from which to fetch the named variable. variable(s, C) is useful for assigning to or reading from another task's variables. In the following seqload example, the parent initializes each child's Parent global variable (if there is one) to refer to the parent's &main co-expression. A child task can then use Parent to determine whether it is being run standalone or under a parent task. The variable() function is useful in inspecting values, especially at intermediate points during the monitored execution of a TP.

```
# seqload3.icn
procedure main(arguments)
    every arg := !arguments do {
        Task := load(arg)
        variable("Parent", Task) := &main
        @Task
        }
end
```

In addition to extending existing functions for monitoring, several new functions have been added. The use of these monitoring functions are illustrated in many example monitors in Unicon's ipl/mprogs directory. Some of the intertask access functions are listed in Figure 5-1. In these functions, parameter C refers to a co-expression that may be from a task other than the one being executed. Functions that *generate* values can produce more than one result from a given call.

cofail(C)	transmit failure to C.
fieldnames(r)	generate fieldnames of record r.
globalnames(C)	generate the names of C's global variables.
keyword(s, C)	produces keyword s in C.
	Keywords are special global variables that have
	special semantics in certain language facilities.
localnames(C,i)	generates the names of C's local variables,
	i calls up from the current procedure call.
paramnames(C,i)	generates the names of C's parameters.
staticnames(C,i)	generates the names of C's static variables.
structure(C)	generates the values in C's block region, or heap.
	The heap holds structure types such as lists and tables.
variable(s,C,i)	produces variable named s, interpreted i levels
	up within C's procedure stack.

Figure 5-1: Unicon interprogram access functions.

Shared icode libraries

Programs that are written to take advantage of the multi-tasking environment gain in space efficiency and modularity. Code sharing is one natural way to achieve space efficiency in a collection of programs. Since procedures are first-class data values in Icon, code sharing can be implemented via data sharing. Programs executing in a single invocation of the interpreter can share code easily if the code is not required to produce side effects on global variables in the calling task's program state. If side-effects to the calling task's program state are required, the shared code must generally be written with care to explicitly reference the calling task's state. Side effects in the client task can also be achieved through the parameters passed in and results obtained by calling the shared procedure.

Loading shared code

Consider a collection of applications that make extensive use of procedures found in the Icon program library (IPL) [Griswold90c]. If those applications are run using MT Icon, the IPL routines need be loaded only once, after which they may be shared.

In order to reference shared code from a loaded task, two additional considerations must be satisfied: the shared code must be loaded, and the client tasks must be able to dynamically link shared routines into their generated code.

Both of these problems can be solved entirely at the source level: In order to introduce a shared Icon procedure into the name space, a global variable of the same name must be declared. Managing the loading of shared libraries is itself a natural task to assign to an Icon procedure that uses a table to map strings to the pointers to the procedures in question.

Code sharing example

The following collection of three programs illustrate one schema that allows code sharing. Other conventions can certainly be devised, and much of the sharing infrastructure presented here can be automatically generated. Program calc.icn consists of a shared library procedure named calc() and a main procedure that exports a reference to calc() for sharing:

```
# calc.icn
procedure calc(args...)
    # code for calc
    # (may call other routines in calc.icn if there are any)
end

procedure main()
    # initialization code, if any
    return calc
end
```

Note that a module exporting shared procedures can also have global variables (possibly initialized from other command-line arguments). Shared modules can export other values besides procedures using the same mechanism.

The parent task that loads the various shared library clients implements a procedural encapsulation (loadlib() in this example) of an Icon table to store references to shared routines. The parent passes this procedure to clients. Each client calls the procedure for each shared routine. Routines that are already loaded are returned to requesting tasks after a simple Icon table lookup. Whenever a routine is requested that has not been loaded, the load() function is called and the shared library activated.

```
procedure main(arguments)
    @load("client",put(arguments,loadlib))
end

procedure loadlib(s, C)
    static sharedlib
```

```
    initial sharedlib := table()
    sharedlib[s] := @load(s)
    variable(s, C) := sharedlib[s]
end
```

A client of calc declares a global variable named calc, and assigns its value after inspecting its argument list to find the shared library loader:

```
global loadlib
global calc
procedure main(arguments)
    if /loadlib then stop("no shared libraries present")
    loadlib("calc", &current)
    # ... remainder of program may call shared calc
end
```

Sharing procedure collections

The primary deficiency of the previous example is that it requires one shared library procedure per Icon module, that is, separate compilation. In practice it is more convenient to have a collection of related procedures in a given Icon compilation unit. Shared libraries can employ such a mechanism by resorting to a simple database that maps procedure names to load modules.

5.7 Summary

Unicon's system facilities provide a high-level interface to the common features of modern operating systems, such as directories and network connections. This interface is vital to most programs, and it also presents the main portability challenges, since a Unicon design goal is for applications to require no source code changes and no conditional code needed to run on most operating systems. Of course some application domains such as system administration are inevitably platform dependent.

There are two major areas of the system interface that are whole application domains extensive enough to warrant an entire chapter of their own: databases and graphics. Databases can be viewed as a hybrid of the file and directory system interface with some of the data structures described in Chapter 2. Since many databases are implemented using a client/server architecture, the database interface also includes aspects of networking. Databases are presented in the Chapter 6.

Graphics is another crucial component of the system interface, rich enough to warrant special features built-in to the language, and deep enough to warrant an entire book. Like databases, graphics can be viewed as an extension of the file data type described in this chapter. Unicon's powerful 2D and 3D graphics facilities are discussed in Chapter 7.

Chapter 6

Databases

Databases are technically part of the system interface described in the last chapter, but they are an important application area in their own right. Different kinds of databases are appropriate in different situations depending on how much information is to be stored and what kinds of accesses to the information are supported. This chapter describes three kinds of databases for which Unicon provides direct support, enabling you to:

- Read and write memory-based structures to data files.
- Use DBM databases as a persistent table data type.
- Manipulate SQL databases through the ODBC connection mechanism or the SQLite plugin (discussed on page 525).

6.1 Language Support for Databases

Unicon provides transparent access to databases stored in local files and on remote servers. The term *transparent* means that the built-in functions and operators used to access information in a database are the same as those used to access information in the memory-based structures presented in Chapter 2. To do this, connections to databases are represented by new built-in types that are extensions of the file and table data types.

Some people might prefer a different syntax for databases from what is used for data structures. A different syntax, such as one based purely on function calls, would be consonant with the difference in performance the programmer can expect to see when accessing data in files as opposed to memory-based data structures. However, the performance of operators already depends on the type of the operands. Consider the expression `!x`. If `x` is a structure, its elements are generated from memory, but if `x` is a file, `!x` reads and generates lines from the file. The goal for Unicon is to make databases just as easy to learn and use as the rest of the language, and to minimize the introduction of new concepts.

The word "database" means different things to different people; for some, it is the short form of "relational database." This chapter uses the term database to refer to any method of providing *persistent structures* that store information from one program run to the next. The operators used to access a database determine whether one element at a time is read or written, or whether many operations are buffered and sent to the database together.

6.2 Memory-based Databases

If the entire database fits in memory at once, you can achieve vast speed-ups by avoiding the disk as much as possible. For example, all queries to read the database can be performed from memory. The database may be modified in memory immediately, and updated on the disk later on. Memory-based databases become increasingly feasible as main memories grow larger. They are an excellent choice for many applications.

One way to implement a memory-based database is to build up your arbitrary structure in memory, and then use the Icon Program Library module xcodes to write them out and read them in. The xcodes procedures convert structures to a string format that can be written to a file, and convert such strings back into the corresponding structure. The following sequence saves the contents of structure db to a file named db.dat.

```
db := table()
db["Ralph"] := "800-USE-ICON"
db["Ray"] := "800-4UN-ICON"
dbf := open("db.dat","w")
xencode(db, dbf)
close(dbf)
```

The converse operation, reading in a structure from a file is also simple:

```
dbf := open("db.dat")
db := xdecode(dbf)
close(dbf)
write(db["Ralph"])
```

This approach works great for databases that do not need to be written to disk on an on-going basis and for which the queries can readily be expressed as operations on structure types. For example, a telephone rolodex application would be well-served by this type of database. The data fits comfortably in memory, updates often occur in batches, and simple queries (such as a student's name) provide sufficient access to the data. The other two kinds of databases in this chapter use traditional database technologies to efficiently address situations where this type of database is inadequate.

6.3 DBM Databases

A classic database solution on the UNIX platform is provided by the DBM family of library functions. DBM stands for Data Base Manager, and the functions maintain an association between keys and values on disk, which is very similar to the table data type. DBM was eventually followed by compatible superset libraries called NDBM (New Data Base Manager) and GDBM (GNU Data Base Manager). Unicon uses GDBM on all platforms.

DBM databases are accessed via the open() function using mode "d" to open a database for reading and writing, or mode "dr" for read-only access. Once opened, DBM databases resemble the table data type and are manipulated using table operations. For example, if d is a DBM file, d[s] performs a database insert/update or lookup, depending on whether the expression is assigned a new value, or dereferenced for its current value. Values can also be inserted into the database with insert(d, k, v) and read from it with fetch(d, k). delete(d, k) similarly deletes key k from the database. DBM databases are closed using the close() function. The following example program takes a database and a key on the command line, and writes out the value corresponding to that key.

```
procedure main(args)
    d := open(args[1], "d") | stop("can't open ", args[1])
    write(d[args[2]])
end
```

If you are wondering why the call to open() isn't followed by a call to close(), you are right, it is proper to close files explicitly, although the system closes all files when the program terminates. How would you generalize this program to accept a third command-line argument, and insert the third argument (if it is present) into the database with the key given by the second argument? You might easily wind up with something like this:

```
procedure main(args)
    d := open(args[1], "d") | stop("can't open ", args[1])
    d[args[2]] := args[3]
    write(d[args[2]])
    close(d)
end
```

DBM databases are good for managing data sets with a simple organization, when the size of the database requires that you update the database a record at a time, instead of writing the entire data set. For example, if you wrote a Web browser, you might use a DBM database to store the user's set of bookmarks to Web pages of interest.

There is one basic limitation of DBM databases when compared with the table data type that you should know about. DBM databases are string-based. The keys and values you put in a DBM database get converted and written out as strings. This makes the

semantics of DBM databases slightly different from tables. For example, a table can have two separate keys for the integer 1 and the string "1", but a DBM database will treat both keys as the string "1". This limitation on DBM databases also means that you cannot use structure types such as lists as keys or values in the database. If the type is not convertible to string, it won't work. You can use the functions xencode() and xdecode(), described in the previous section, to manually convert between strings and structures for storage in a DBM database if you really need this capability.

6.4 SQL Databases

DBM is great for data that is organized around a single key, but it is inadequate for complex databases. The industry standard choice for enterprise-level data organization is the Structured Query Language (SQL). SQL is supported by every major database vendor.

Unlike DBM, a SQL database can contain multiple tables, and those tables are accessed by walking through a set of results to a query, rather than by accessing individual elements directly. SQL is designed for industrial-strength relational databases.

The SQL language

The SQL language was invented by IBM and based on relational database theory developed by E.F. Codd. A database is a collection of *tables*, and each table is a collection of *rows*. The rows in a table contain information of various types in a set of named *columns*. Rows and columns are similar to records and fields, except that they are logical structures and do not describe the physical form or layout of the data. There is an ANSI standard definition of SQL, but many vendors provide extensions, and most vendors are also missing features from the ANSI standard. Unicon allows you to send any string you want to the database server, so you can write portable "vanilla SQL" or you can write vendor-specific SQL as needed.

SQL was originally intended for text-based interactive sessions between humans and their databases. Nowadays, SQL is primarily used "under the covers" by database applications that accommodate novice users with a graphical interface that does not require any knowledge of SQL, while supporting a SQL "escape hatch" for advanced users who may wish to do custom queries. Such an escape hatch is also a major potential security and stability hole, so be cautious about allowing a user to type SQL commands themselves.

The duality of pre-cooked GUI-supported SQL versus arbitrary SQL strings for power users is paralleled in the Unicon language by the fact that Unicon's built-in database operators and functions duplicate a subset of the capabilities of SQL. There are often two ways to do things: using Unicon operations or using SQL statements.

SQL statements can be divided into several categories, the most prominent of which are data definition and data manipulation. When using SQL within a Unicon program, you

build up string values containing SQL statements. In the following examples, the SQL is given unadorned by double quotes or other Unicon artifacts.

New tables are created with a CREATE TABLE statement, such as

```
create table addresses (name varchar(40), address varchar(40),
        phone varchar(15))
```

Tables have a primary key that must be unique among rows in the table. By default the primary key is the first one listed, so name is the primary key in table addresses above.

SQL's data manipulation operations include SELECT, INSERT, UPDATE, and DELETE. SELECT determines the data set being operated on, picking rows and columns that form some projection of the original table. SELECT also allows you to combine information from multiple tables using relational algebra operations. Most databases are long-lived and evolve to include more columns of information over time. SQL's ability to select and operate on projections is an important feature, since code that works with a certain set of columns continues to work after the database is modified to include additional columns.

INSERT, UPDATE, and DELETE all modify the table's contents. INSERT adds new rows to a table. For example:

```
insert into addresses (name, address, phone)
    values ('Nick K', '1 Evil Empire', '(123)456-7890')
insert into addresses (name, address, phone)
    values ('Vic T', '23 Frozen Glade', '(900)888-8888')
```

UPDATE and DELETE can modify or remove sets of rows that match a particular criterion, as in

```
update addresses set address = '666 RTH, Issaquah'
        where name = 'Nick K'
delete from addresses where name = 'Vic T'
```

This section presented only a few aspects of the SQL language. For simple database tasks you can in fact ignore SQL and use the Unicon facilities described in the rest of this chapter. However, for more complex operations the best solution is to formulate some SQL commands to solve the problem. A full discussion of SQL is beyond the scope of this book. For more information on this topic you might want to read one of the following books: Ramez Elmasri and Shamkant Navanthe's *Fundamentals of Database Systems,* C.J. Date and Hugh Darwen's *A Guide to the SQL Standard.*

Database architectures and ODBC

SQL databases are accessed through an underlying Open DataBase Connectivity (ODBC) transport mechanism. This mechanism allows the programmer to ignore the underlying architecture. Hiding this complexity from application programmers is important.

The database architecture may vary from a single process accessing a local database, to client/server processes, to three or more tiers spanning multiple networks. Figure 6-1 illustrates the role played by ODBC in providing Unicon with database access in one- and two-tier configurations. While this chapter cannot present a complete guide to the construction of database systems, it provides concrete examples of writing Unicon client programs that access typical database servers.

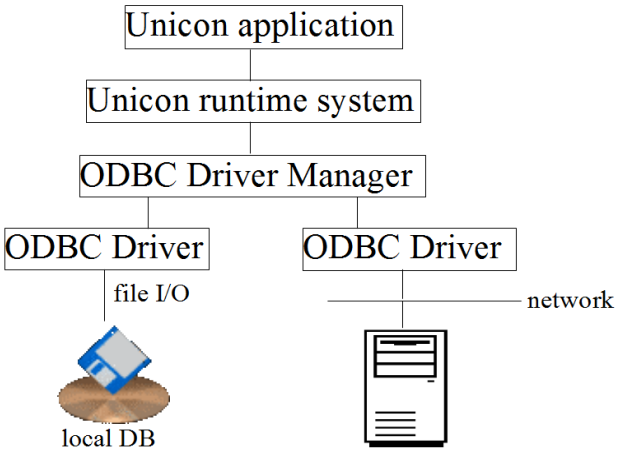


Figure 6-1: Unicon and ODBC hide underlying architecture from applications

To use Unicon's SQL facilities, you must have several software components in place. First, you need a SQL database server that supports ODBC. You can buy a commercial SQL server, or use a free server such as MySQL (www.mysql.com) or PostgreSQL (www.postgresql.org).

Second, you need an account, password, and permissions on the SQL server in order to connect to it. The details are server-dependent and outside the scope of this book.

Third, your client machine needs an ODBC driver manager, and an ODBC driver for your SQL server; these must be configured properly. The driver manager is a component that connects applications to various databases; ODBC drivers are dynamic link libraries that database vendors supply to talk to their database. Figure 6-2 shows database configuration for MySQL via the MyODBC GUI dialog box on Windows (left), and in a ~/odbc.ini file on Linux (right). In both cases, configuration involves knowing the internet server name or IP address, the port, and the database to connect to. For that triplet, you get to define a name called a DSN or data source name, which is the name that Unicon will pass in to open(). In the Windows dialog, this name is a text field explicitly named as a DSN, while in the Linux odbc.ini file, it is at the top, inside the square brackets.

In both cases, there are a lot of additional options which are beyond the scope of this book. On Windows, each ODBC driver may have its own custom dialogs for configuration, while on Linux the odbc.ini file is more the property of the driver manager and is used to configure all the various drivers. As a fair warning, the details required in the dialogs or

the exact syntax of the odbc.ini and its required entries for a given driver change slightly from time to time and are beyond the scope of this book. Consult current ODBC documentation for the driver manager on your platform and the specific database to which you are connecting.

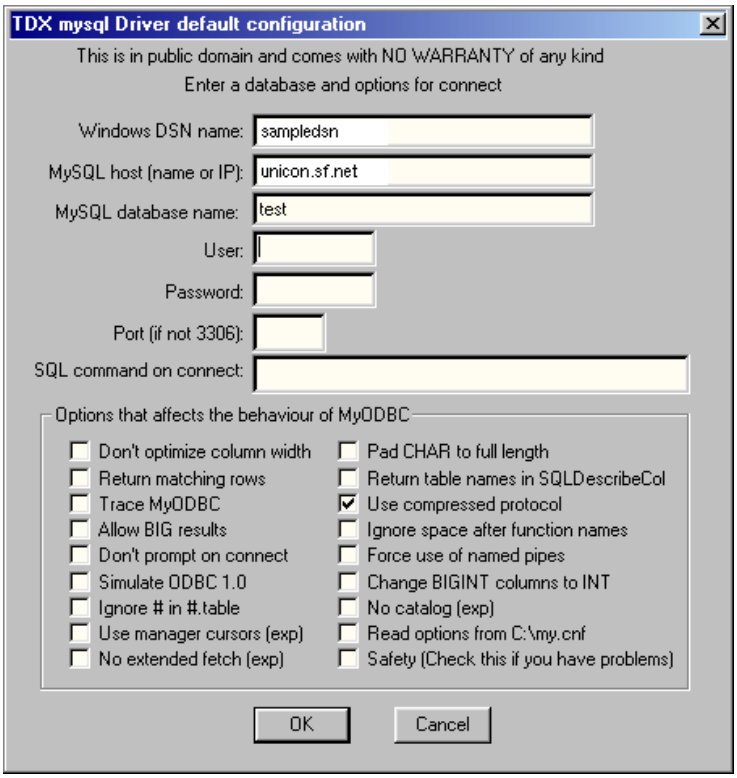


Figure 6-2: Configuring ODBC on Windows (left) and Linux (right)

Once you have the ODBC software set up, writing the Unicon client program to connect to your database is straightforward.

Opening a SQL database

To connect to a SQL database, call open() with mode "o". This establishes a session with a data source. The filename argument to open() is the data source name to which you are connecting; it is associated with a particular ODBC driver, remote database server machine or IP number, and port within the ODBC driver manager.

Mode "o" is followed by additional string arguments to open(). The first is an optional default table name used in the various functions that take a table name. Applications that send their own custom SQL strings via the sql() function may find that their SQL strings always specify what table they are operating on, rendering this optional parameter unnecessary. The next two arguments are the user name and password to use in connecting to the specified data source. Here is an example that establishes connections to a database (unicondb) as user scott with password tiger, with and without specifying an initial table

mydbtable. Most applications will not need two open connections to their database, but see below for an example that does.

```
db := open("unicondb", "o", "mydbtable", "scott", "tiger")
db2 := open("unicondb", "o", "scott", "tiger")
```

The open() function returns a value that supports many of the operations of both a file and a table, or fails if the connection cannot be established. The underlying session information is shared by multiple calls to open() the same database. In addition to the network connection and SQL session information that is retained, each database file value maintains a current *selection* consisting of a set of rows corresponding to the current query, and a *cursor position* within that selection. When a database is first opened, the selection consists of a null set containing no rows and no columns.

Querying and modifying a SQL database

Subsequent queries to the database can be made by calling sql(db, sqlcmd). The sql() function sets the current selection within the database and places the cursor at the beginning of the set of selected rows. For example, to obtain Vic T's phone number you might say

```
sql(db, "select phone from addresses where name='Vic T'")
```

Vic's phone number is included if you use the original select * query, but the more specific your query, the less time and network bandwidth is wasted sending data that your client application must filter out. You should do as much work on the server (in SQL) as possible to make the client more efficient.

Since the function sql() transmits arbitrary SQL statements to the server, it can be used for many operations besides changing the current selection. The sql() function returns a null value when there is no return value from the operation, such as a create table statement. Its return value can be of varying type for other kinds of queries, and it can fail, for example if the SQL string is malformed or requests an impossible operation.

Traversing the selected rows

To walk through the rows of your current database selection, you call fetch(db). The value returned is a *row* that has many of the operations of a record or a table, namely field access and subscripting operators. For example, if fetch(db) returns a row containing columns Name, Address, and Phone, you can write

```
row := fetch(db)
write(row.Name)
write(row["Address"])
```

Called with one argument, fetch(db) advances the cursor one position. With two arguments, fetch(db, col) produces a column by name from the current row, without advancing the cursor. The preceding example could have been written

```
write(fetch(db,"Name"))
write(fetch(db,"Address"))
```

A SQL example application

A human resources database might include two tables. One table might maintain employee information, such as names, identification numbers, and phone numbers, while another table maintains entries about specific jobs held, including employee's ID, the pay rate, a code indicating whether pay is hourly or salaried, and the job title. Note that the SQL is embedded within a string literal.

```
sql(db, "create table employees (id varchar(11), name varchar(40),
        phone varchar(15))")
sql(db, "create table jobs (id varchar(11), payrate integer, is_salaried char,
        title varchar(40))")
```

Inserting rows into the database looks like

```
sql("insert into employees (id, name, phone) values(32, 'Ray', '274-2977')")
```

Now, how can you print out the job title for any particular employee? If you have the employee's identification number, the task is easy, but let's say you just have their name. These are the kinds of jobs for which SQL was created. Information from the employees table is effortlessly cross-referenced with the jobs table by the following SQL. The string is long so it is split into two lines. A Unicon string literal spans multiple lines when the closing double quotes has not been found and the line ends with an underscore character.

```
sql(db, "select name,title from employees,jobs _
      where name='Ray' and employees.id = jobs.id")
while write(fetch(db).Title)
```

SQL types and Unicon types

SQL has many data types, most of which correspond to Unicon types. CHAR and VARCHAR correspond to Icon strings; INTEGER and SMALLINT correspond to integers; FLOAT and REAL correspond to reals, and so on. The philosophy is to convert between Icon and SQL seamlessly and with minimal changes to the data format, but you should be aware that these are not exact matches. For example, it is possible to define a FLOAT with more precision than an Icon real, and it is easy to produce an Icon string that is longer than the

maximum allowed VARCHAR size on most SQL servers. Unicon programmers writing SQL clients must be aware of the limitations of the SQL implementations they use.

Unicon has structure types for which there is no SQL equivalent. Values of these types cannot be inserted into a SQL database unless you explicitly convert them to a SQL-compatible type (usually a string) using a function such as xencode(). SQL also has types not found in Unicon such as bit strings, timestamps, and BLOBS; they are represented by strings, and strings are used to insert such values into SQL databases. Strings are also used to represent out-of-range values when reading SQL columns into Unicon.

More SQL database functions

SQL databases are feature-rich enough to warrant a suite of functions in addition to those they share with other kinds of files and databases. These functions are described in detail in Appendix A, but some of them deserve special mention. The function dbtables(db) is useful to obtain a listing of the data objects available within a particular database. Function dbcolumns(db) provides detailed information about the current table that is useful in writing general tools for viewing or modifying arbitrary SQL databases.

The functions dbproduct(db) and dbdriver(db) produce information about the DBMS on which db resides, and the ODBC driver software used in the connection. The function dblimits(db) produces the upper bounds for many DBMS system parameters, such as the maximum number of columns allowed in a table. These functions return their results as a record or list of records whose field names and descriptions are given in Appendix A.

6.5 Tips and Tricks for SQL Database Applications

In addition to the complexity of learning SQL itself, SQL database applications have a characteristic flavor which may or may not seem natural to the Unicon programmer.

Operating on large files

Asking for 200MB of data in a remote SQL database is a good way to bring a computer to its knees. Some SQL operations are slow due to an inefficient query on the remote server, while others are slow because large amounts of data are transmitted over a limited network connection. For a fixed amount of data, operation time will vary radically depending on how it is organized; fewer, larger tuples are transmitted faster than many smaller tuples.

Use multiple connections to nest queries

It is common to use more than one table at once. Some times this is using SQL's JOIN operation, but sometimes it is not. If you try to nest a second query inside a first one, you

will quickly find that on a given connection, only one SELECT and one row set is maintained at a time. The second SELECT replaces the first, so for example:

```
db := open("mydsn", "o", ...)
sql(db, “SELECT ...”)
while r := fetch(db) do {
    sql(db, “SELECT ...”)
    while r2 := fetch(db) do write(r2.foo)
    }
```

does not work. Within your operating system and database server's limits, the easy solution is to open multiple connections to your database:

```
db1 := open("mydsn", "o", ...)
db2 := open("mydsn", "o", ...)
sql(db, “SELECT ...”)
while r := fetch(db) do {
    sql(db2, “SELECT ...”)
    while r2 := fetch(db2) do write(r2.foo)
    }
```

Dynamic records

Rows are represented as a special kind of Unicon record whose fields are determined at run-time from the names of selected columns. Record types introduced at runtime are called *dynamic records*, and they are useful in other contexts besides databases.

The function constructor(rname, field, field, ...) produces a procedure that constructs records named rname with the given fields. The field names can be arbitrary strings, but only legal identifiers will be subsequently accessible via the field operator (.)

The db library

The declaration link db provides simplified SQL access routines for non-SQL programmers. This library will not allow you to avoid learning SQL for long, but may ease the conversion from Unicon structure values into SQL strings for transmission over the network. The most useful of these procedures is dbupdate(), which sends a record (tuple) to the database. The following example updates two columns within a row returned by fetch().

```
row := fetch(db)
row.Name := "Bill Snyder"
row["Address"] := "6900 Tropicana Blvd"
dbupdate(db, row)
```

Of course, before a fetch can be performed, a row set must have been selected. The procedure dbselect(db, columns, filter, order) selects tuples containing columns from the database, with optional filter(s) and ordering rule(s). Inserting and deleting rows is performed by procedures dbinsert() and dbdelete(). The dbinsert() function takes two parameters for each column being inserted, the column name and then the value.

Unwritable tuples

Many SQL selections are read-only. The relational combination of columns from different tables is powerful, but the resulting selections are non-updatable. Another example of a read-only query is a GROUP BY query, which is usually applied before an aggregate count. Executing a SELECT * on a single table is updatable, but if you do something fancier, you will have to know the semantics of SQL to tell whether the result may be modified.

6.6 Summary

Databases are a standard form of persistent storage for modern applications. The notation for manipulating a database looks like a sequence of table and record operations, comprising a combination of Unicon and SQL statements. Database facilities give programmers direct access and control over the information flow to and from permanent storage.

Chapter 7

Graphics

Unicon provides a rich high level interface to 2D and 3D raster graphics, text fonts, colors, and mouse input facilities provided by an underlying system, such as the X Window System. Unicon's graphics are portable across multiple platforms. The most important characteristics of the graphics facilities are:

- Simplicity, ease of learning
- Windows are integrated with Unicon's existing I/O functions
- Straightforward input event model

This chapter presents Unicon's 2D and 3D graphics facilities. Some material on 2D graphics comes from University of Arizona CS TR93-9. The 3D graphics sections come from Unicon TR 9, whose original author is Naomi Martinez. The definitive reference for the 2D graphics facilities is "Graphics Programming in Icon" by Griswold, Jeffery, and Townsend, and this book is of value for writing 3D programs. Online references for the graphics facilities also come with the software distributions.

Because different platforms have radically different capabilities, there is a trade-off between simplicity, portability, and full access to the underlying machine. Unicon aims for simplicity and ease of programming rather than full low-level access.

7.1 2D Graphics Basics

Unicon's 2D facilities provide access to graphics displays without enforcing a particular user interface look-and-feel. Events other than keystrokes and mouse events are handled automatically by the runtime system. Chapter 17 describes the standard class library and user interface builder for Unicon applications.

Graphic interfaces are *event driven*; an event reading loop is the control mechanism driving the application. For example, if an application must be ready to redraw the contents of its window at all times, it may not compute for long periods without checking

for window events. This event driven paradigm used in the underlying implementation is optional at the Unicon application level. Since Unicon windows handle many events automatically and "take care of themselves", applications follow the event driven paradigm only when it is needed. Unicon's extensive use of default values make simple graphics applications extremely easy to program, while providing flexibility where it is needed in more sophisticated applications.

A window is a special file opened with mode "g", appearing on-screen as a rectangular space for text and/or graphics. Windows support text I/O, much as one uses a text terminal. A simple Unicon graphics program might look like this:

```
procedure main()
  w := open("hello", "g")
  write(w, "hello, world")
  # do processing ... use w as if it were a terminal
  close(w)
end
```

Windows are open for both reading and writing, and support the usual file operations with the exceptions of seek() and where(). Unlike regular files, the type() of a window is "window". Like other files, windows close automatically when the program terminates, so the call to close() in the above example is optional.

Bit-mapped, or raster, graphics constitute a second programming model for windows. There are no programming "modes" and code that uses graphics may be freely intermixed with code that performs text operations. There are many graphics functions and library procedures, detailed in Appendices A and B.

&window: the Default Window Argument The keyword &window is a default window for graphics. &window starts with a null value; only window values (and &null) may be assigned to &window. &window is a default argument to most graphics functions and is used implicitly by various operations. If a program uses &window, the argument can be omitted from calls to functions such as EraseArea() and WAttrib(). The window argument is required for calls to file functions such as write() and writes() since these functions default to &output, not &window. The default window shortens the code for graphics-oriented programs and makes it faster.

2D Graphics Coordinates The 2D graphics functions use an integer coordinate system based on pixels (picture elements). Like the text coordinate system, 2D graphics coordinates start in the upper-left corner of the screen. From that corner the positive x direction lies to the right and the positive y direction moves down. Unlike text coordinates, the graphics coordinate axes are zero-based, which is to say that the very top leftmost pixel is (0,0) by default.

Angles are real numbers given in radians, clockwise starting at the 3 o'clock position. Many functions operate on rectangular regions specified by x, y, width, and height components. Width and height may be negative to extend the rectangle left or up from x and y. Screen output may be limited to a rectangle within the window called the clipping region. The clipping region is set or unset using the function Clip().

Window Attributes A window's state has many *attributes* with associated values. Some values are defined by the system, while others are under program control, with reasonable defaults. When opening a window, open() allows string arguments after the filename and mode that specify initial values of attributes when the window is created. For example, to say hello in italics on a blue background one may write:

```
procedure main()
  w := open("hello", "g", "font=sans,italics", "bg=blue")
  write(w, "hello, world")
  # processing ...
end
```

After a window is created, its attributes are read and set using the function WAttrib(w,s1,s2,...). Arguments to WAttrib() that have an equals sign are assignments that set the given value if possible; WAttrib() fails otherwise. open() only allows such attribute assignments. Some attributes can only be read by WAttrib() and not set.

String arguments to WAttrib() that have an attribute name but no value are queries which return the attribute value. WAttrib() generates a string result for each argument; a query on a single argument produces just the value of that attribute; for multiple arguments and in the case of assignment, the result is the *attr=val* form attribute assignments take. Attributes are also frequently set implicitly by the user's manipulation of the window; for instance, cursor or mouse location or window size.

Table 7-1 lists attributes that are maintained on a per-window basis. Attribute values are string encodings. Usage refers to whether the attribute may be read, written or both. RWO and WO attributes can be assigned only when the window is opened. Although all attribute values are encoded as strings, they represent a range of window system features. The attribute pointer refers to mouse pointer shapes that may be changed by the application during different operations. The attribute pos refers to the position of the upper-left corner of the window on the screen. Screen position is specified by a string containing x,y coordinates, e.g. "pos=200,200".

Table 7-1
Canvas Attributes

Name	***Type / Example***	***Description***	***Usage***
size	pixel pair	Size of window	RW
pos	pixel pair	Position of window on screen	RW
canvas	normal, hidden	Canvas state	RW
windowlabel	string	Window label (title)	RW
inputmask	string	select categories of input events	RW
pointer	arrow, clock	Pointer (mouse) shape	RW
pointerx, pointery	pixel	Pointer (mouse) location	RW
display	device / "my.cs.esu.edu:0"	(X11) device window resides on	RWO
depth	# of bits	Display depth	R
displaywidth, displayheight	pixel	Display size	R
image	string	Initial window contents	WO

7.2 Graphics Contexts

Some attributes are associated with the window itself, while others are associated with the *graphics context*, a set of resources used by operations that write to windows. This distinction is unimportant in simple applications but is useful in more sophisticated applications that use multiple windows or draw many kinds of things in windows. A graphics context has colors, patterns, line styles, and text fonts and sizes.

Although they are called graphics contexts, text operations use these attributes. Text is written using the foreground and background colors and font defined in the graphics context. Table 7-2 lists the attributes associated with a graphics context.

Table 7-2
Context Attributes

Name	***Type / Example***	***Description : Default***	***Usage***
fg	color / "red"	Foreground color : black	RW
bg	color / "0,65535,0"	Background color : white	RW
font	font name	Text font : fixed	RW
fheight, fwidth	integer	Text font max char height and width	R
leading	integer	Vertical # pixels between text lines	RW

Name	*Type / Example*	*Description : Default*	*Usage*
ascent, descent	integer	Font height above/below baseline	R
drawop	logical op / reverse	Drawing operation: copy	RW
fillstyle	stippled, opaquestippled	Graphic fill style : solid	RW
pattern	"4,#5A5A"	Fill pattern	RW
linestyle	onoff, doubledash	Drawing line style : solid	RW
linewidth	integer	Drawing line width	RW
clipx, clipy, clipw, cliph	integer	Clip rectangle position and extent: 0	RW
dx, dy	integer	Output coordinate translation : 0	R
image	string / "flag.xpm"	Initial window contents	WO

Binding Windows and Graphics Contexts Together Graphics contexts can be shared among windows, and multiple graphics contexts can be used on the same window. An Unicon window value is a *binding* of a canvas (an area that may be drawn upon) and a graphics context. A call open(s,"g") creates both a canvas, and a context, and binds them together, producing the binding as its return value.

Clone(w) creates a binding of the canvas and attributes of w with a new graphics context that is manipulated independently. Clone() also accepts any number of string attributes to apply to the new window value, as in open() and WAttrib(). After calling Clone(), two or more Unicon window values write to the same canvas. The cursor location is stored in the canvas, not the graphics context. Writing to the windows produces concatenated (rather than overlapping) output. Closing one of the window values removes the canvas from the screen but does not destroy its contents; the remaining binding references an invisible pixmap. The canvas is destroyed after the last binding associated with it closes. Use of Clone() can significantly enhance performance for applications that require frequent graphics context manipulations.

Subwindows The function Clone() can also be used to create subwindows, which are canvases that reside within other windows. Clone(w, "g", ...) opens a 2D subwindow within w, and Clone(w, "gl", ...) opens a 3D subwindow within w. Applications must supply position and size attributes when they create a subwindow. Input events to a subwindow are not seen on the enclosing parent window and vice versa; both windows must be polled or supplied to WActive() or select() in order to handle input.

Coordinate Translation In 2D, context attributes dx and dy perform output coordinate translation. dx and dy take integer values and default to zero. These integers are added into the coordinates of all output operations that use the context; input coordinates in &x and &y are not translated.

7.3 Events

User input such as keystrokes and mouse clicks are called *events*. Many events are handled by Unicon automatically, including window redrawing and resizing, etc. Other events are put on a queue in the order they occur, for processing by the Unicon program. When reading from a window using file input functions such as reads(w, 1), only keyboard events are produced; mouse and other events are dropped.

The primary input function for windows is Event(w), which produces the next event for window w and removes it from the queue. If the event queue is empty, Event() waits for an event. Keyboard events are returned as strings, while mouse events are returned as integers. Special keys, such as function and arrow keys, are also returned as integers, described below. Event() also removes the next two elements and assigns the keywords &x and &y the pixel coordinates of the mouse at the time of the event. The values of &x, &y remain available until a subsequent call to Event() again assigns to them. Event() sets the keyword &interval to the number of milliseconds that have elapsed since the last event (or to zero for the first event). Keywords &control, &shift, and &meta are set by Event() to return the null value if those modifier keys were pressed at the time of the event; otherwise they fail. For resize events, &interval is set to zero and modifier keywords fail. Keywords associated with event processing on windows are summarized in Table 7-3:

Table 7-3
Window Input Event Keywords

Keyword	***Description***
&x	Mouse location, horizontal
&y	Mouse location, vertical
&row	Mouse location, text row
&col	Mouse location, text column
&interval	Time since last event, milliseconds
&control	Succeeds of Control key pressed
&shift	Succeeds if Shift key pressed
&meta	Succeeds if Alt (meta) key pressed

Keyboard Events The regular keys that Unicon returns as one-letter strings correspond approximately to the lower 128 characters of the ASCII character set. These characters include the control keys, the escape key, and the delete key. Modern keyboards have many additional keys, such as function keys, arrow keys, "page down", etc. Unicon produces integer events for these special keys. A collection of symbol definitions for special keys is available in the library include file keysyms.icn. The most common of these are Key_Down, Key_Up, Key_Left, Key_Right, Key_Home, Key_End, Key_PgUp, Key_PgDn, Key_F1...Key_F12, and Key_Insert.

Mouse Events Mouse events are returned from Event() as integers indicating the type of event, the button involved, etc. Keywords allow the programmer to treat mouse events symbolically. The event keywords are:

Table 7-4
Window Input Event Codes

Keyword	***Event***
&lpress	Mouse press left
&mpress	Mouse press middle
&rpress	Mouse press right
&lrelease	Mouse release left
&mrelease	Mouse release middle
&rrelease	Mouse release right
&ldrag	Mouse drag left
&mdrag	Mouse drag middle
&rdrag	Mouse drag right
&resize	Window was resized

The following program uses mouse events to draw a box that follows the mouse pointer around the screen when a mouse button is pressed. The attribute drawop=reverse allows drawing operations to serve as their own inverse; see [Griswold98] for more about the drawop attribute. Function FillRectangle() draws a filled rectangle on the window and is described in the reference section. Each time through the loop the program erases the box at its old location and redraws it at its new location; the first time through the loop there is no box to erase so the first call to FillRectangle() is forced to fail by means of Unicon's \ operator.

```
procedure main()
  &window := open("hello", "g", "drawop=reverse")
  repeat if Event() === (&ldrag | &mdrag | &rdrag) then {
     # erase box at old position, then draw at new position
     FillRectangle(\x, \y, 10, 10)
     FillRectangle(x := &x, y := &y, 10, 10)
     }
end
```

The program can inspect the window's state using WAttrib(). Between the time the mouse event occurs and the time it is produced by Event(), the mouse may have moved. In order to get the current mouse location, use QueryPointer() (see below).

When more than one button is depressed as the drag occurs, drag events are reported on the most recently pressed button. This behavior is invariant over all combinations of presses and releases of all three buttons.

Resize events are reported in the same format as mouse events. In addition to the event code, &x, &y, &col and &row are assigned integers that indicate the window's new width and height in pixels and in text columns and rows, respectively. Resize events are produced when the window manager (usually at the behest of the user) resizes the window; no event is generated when an Unicon program resizes its window.

Key Release, Mouse Motion, and Window Closure Events The canvas attribute inputmask allows programs to request three kinds of additional input events on windows. These events pose enough performance or portability obstacles that they are not produced by default. An "m" in the inputmask requests mouse motion events when no mouse button is depressed; by default only *drag* events are reported. If the inputmask contains a "k", events will be generated when keyboard keys are released. An inputmask attribute containing a "c" requests an event when a window closure is externally triggered, as in the case when a titlebar x button is pressed.

Event Queue Manipulation The event queue is an Unicon list that stores events until the program processes them. When a user presses a key, clicks or drags a mouse, or resizes a window, three values are placed on the event queue: the event itself, followed by two integers containing associated event information.

Pending(w) produces the event queue for window w. If no events are pending, the list is empty. The list returned by Pending() is attached to the window. Additional events may be added to it at any time during program execution. It is an ordinary list and can be manipulated using Unicon's list functions and operators.

When several windows are open, the program may need to wait for activity on any of the windows. Each pending list could be checked until a nonempty list is found, but such a busy-waiting solution is wasteful of CPU time. The function Active() waits for window activity, relinquishing the CPU until an event is pending on one of the open windows, and then returns a window with a pending event. A window is said to starve if its pending events are never serviced. Active() cycles through open windows on repeated calls in a way that avoids window starvation.

7.4 Colors and Fonts

Unicon recognizes a set of string color names based loosely on a color naming system found in [Berk82]. The color names are simple English phrases that specify hue, lightness, and saturation values of the desired color. The syntax of colors is

[*lightness*] [*saturation*] [*hue*[ish]] *hue*

where lightness is one of: pale, light, medium, dark, or deep; saturation is one of weak, moderate, strong, or vivid; and where hue is any of black, gray, grey, white, pink, violet, brown,

red, orange, yellow, green, cyan, blue, purple, or magenta. A single space or hyphen separates each word from its neighbor. When two hues are supplied (and the first hue has no ish suffix), the resulting hue is halfway in between the two named hues. When a hue with an ish suffix precedes a hue, the resulting hue is three-fourths of the way from the ish hue to the main hue. When adding ish to a word ending in e, the e is dropped (for example, purple becomes purplish); the ish form of red is reddish. Mixing radically different colors such as yellow and purple does not usually produce the expected results. The default lightness is medium and the default saturation is vivid. Note that human perception of color varies significantly, as do the actual colors produced from these names on different monitors.

Color Coordinate Systems and Gamma Correction In addition to the standard color names, platform-specific color names may be supported. Colors may also be specified by strings encoding the red, green, and blue components of the desired color. Unicon accepts the hex formats "#rgb" in which r, g, and b are 1 to 4 hex digits each. Red, green, and blue may also be given in decimal format, separated by commas, using a linear scale from 0 to 65535 ("0,0,0" is black; "65535,65535,65535" is white), although displays typically offer far less precision and nonlinear colors. For example, "bg=32767,0,0" requests a medium red background; if the display is incapable of such, it approximates it as closely as possible from the available colors. "fg=0,65000,0" requests a vivid green foreground.

If colors appear darker than they should, the window system is not providing linear colors. Unicon can be told to perform the correction by means of the gamma attribute; 1.0 is a default (no gamma correction), and experimenting with values between 2 and 3 usually provides satisfactory results.

Fonts

Fonts are specified by a comma-separated string of up to four fields supplying the font's family name, followed by optional size or italic or bold designations in any order. The fonts available vary widely from system to system. Four font family names available on all Unicon systems include serif, sans, typewriter, and mono. These families map onto the system's closest approximation of Times, Helvetica, Courier, and a monospaced console font. Font sizes are given in pixel height.

7.5 Images, Palettes, and Patterns

DrawImage(x, y, s) draws a rectangular area using an image string. String s has the form "*width,palette,data*". *width* is the width of the rectangle drawn, palette is a code that defines the colors corresponding to data values, and the rest of the data supplies pixel values. Predefined palettes and palette functions help to provide this capability. Image and palette functions are described fully in [Griswold98].

The context attribute fillstyle determines the pixels used by draw and fill functions like FillPolygon(). If the fillstyle is not solid, a pattern in the filled area is drawn in the foreground color; other pixels are drawn in the background color ("fillstyle=textured"). The function Pattern(w,s) associates a pattern denoted by s with w's context. String s is a built-in pattern name, or a representation of bits that define the pattern. Pattern representations are of the form "width,#bits" where 1 <= width <= 32. The window system may limit the pattern's width and height to as little as 8.

The height of the pattern is defined by the number of rows in the bits component of the pattern string. Bits consists of a series of numbers, each supplying one row of the pattern, in hexadecimal format. Each digit defines four bits and each row is defined by the number of digits required to supply width bits. For example, the call

```
Pattern("4,#5A5A")
```

defines a 4x4 pattern where each row is defined by one hex digit.

pme: a pixmap editor

A simple image editor called pme demonstrates event processing including mouse events. pme displays both a small and a "magnified" display of the image being edited, allows the user to set individual pixels, and allows the user to save the image; it is well-suited for constructing and hand-editing small images such as icons and textures for use in larger 2D or 3D scenes. pme consists of four procedures and employs several graphics functions. A sample screen image of pme is presented in Figure 7-1. The "real" image is in the upper left corner; underneath it is a mouse icon which shows what color is drawn by each of the mouse buttons.

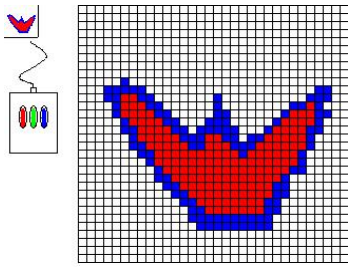


Figure 7-1 pme editing a 32x32 image

pme starts by declaring and initializing several variables.

```
link dialog, file_dlg
global lmargin, colors, colorbinds
procedure main(argv)
    local i := 1, j, s, e, x, y, width := 32, height := 32
```

The image width and height can be specified on the command line with a -size option, for example, pme -size 16,64.

```
if argv[1]=="-size" then {
  i +:= 1
  argv[2] ? {
    width := integer(tab(many(&digits))) | stop("bad -size")
    =","  | stop("bad -size")
    height := integer(tab(0)) | stop("bad -size")
    i +:= 1
    }
  }
```

Following the size arguments, pme checks for a filename specifying the bitmap to edit. If one is found, it is read into the regular scale image, and then the magnified scale image is constructed by reading each pixel using the function Pixel(), and filling an 8x8 rectangle with the corresponding color.

```
i := j := 0
every p := Pixel(0, 0, width, height) do {
  Fg(p)
  FillRectangle(j * 8 + lmargin + 5, i * 8, 8, 8)
  j +:= 1
  if j = width then { i +:= 1; j := 0 }
  }
```

After the images are loaded with their initial contents, if any, a grid is drawn on the magnified image to delineate each individual pixel's boundary. The user's mouse actions within these boxes change the colors of corresponding pixels in the image. An list of three bindings to the window, each with an independently-set foreground color, is used to represent the color settings of the mouse buttons.

```
colors := [Clone("fg=red"),Clone("fg=green"),Clone("fg=blue")]
```

The main event processing loop of pme is simple: Each event is fetched with a call to Event() and immediately passed into a case expression. The keystroke "q" exits the program; the keystroke "s" saves the bitmap in a file by calling WriteImage(), asking for a file name if one has not yet been supplied.

```
case e := Event() of {
"q"|"\e": return
"s"|"S": {
  if /s | (e=="S") then s := getfilename()
  write("saving image ", s, " size ", width,",", height)
  WriteImage( s, 0, 0, width, height)
  }
```

Mouse events result in drawing a single pixel in both the magnified and regular scale bitmaps using one of the colors depicted on the mouse icon.

```
&lpress | &ldrag | &mpress | &mdrag | &rpress | &rdrag : {
  x := (&x - lmargin - 5) / 8
  y := &y / 8
  if (y < 0) | (y > height-1) | (x > width) then next
  if x >= 0 then dot(x, y, (-e - 1) % 3)
```

To change the color drawn by a mouse button, you click on it.

```
  else { # x < logical 0. User clicked in mouse area
    if &x < 21 then getacolor(1, "left", height)
    else if &x < 31 then getacolor(2, "middle", height)
    else getacolor(3, "right", height)
    until Event() === (&mrelease | &lrelease| &rrelease)
    }
  }
```

Pixel drawing is handled by procedure dot(), whose third argument specifies which button, and therefore which color to draw. The dot is drawn using FillRectangle() in the magnified window; in the regular scale window DrawPoint() suffices.

```
procedure dot(x, y, c)
   if (x|y) < 0 then fail
   FillRectangle(colors[c+1], x*8 + lmargin + 5, y*8, 8, 8)
   DrawPoint(colors[c+1], x, y)
   DrawRectangle(x*8 + lmargin + 5, y*8, 8, 8)
end
```

pme illustrates several aspects of the Unicon graphics facilities. Note the event-handling: a case expression handles various keystrokes and mouse events with simpler control structure than in most languages' GUI event processing.

Listing 7-1 **pme**: a Unicon bitmap editor

```
link dialog
link file_dlg
global lmargin, colors
procedure main(argv)
   local i := 1, j, s, e, x, y, width := 32, height := 32
   if argv[1]=="-size" then {
     i +:= 1
     argv[2] ? {
       width := integer(tab(many(&digits))) | stop("bad -size")
```

```
      =","  | stop("bad -size")
      height := integer(tab(0)) | stop("bad -size")
      i +:= 1
      }
   }
lmargin := max(width, 65)
if s := argv[i] then {
   &window := open("pme", "g", "image="||s) | stop("cannot open window")
   width  <:= WAttrib("width")
   height <:= WAttrib("height")
   lmargin := max(width, 65)
   WAttrib("size="||(width*8+lmargin+5)||","||(height*8))
   i := j := 0
   every p := Pixel(0, 0, width, height) do {
      Fg(p)
      FillRectangle(j * 8 + lmargin + 5, i * 8, 8, 8)
      j +:= 1
      if j = width then { i +:= 1; j := 0 }
      }
   }
else {
   &window := open("pme", "g", "size=" || (lmargin+width*8+5)||","||(height*8+5)) |
   stop("cannot open window")
   }
colors := [Clone("fg=red"),Clone("fg=green"),Clone("fg=blue")]
every i := 1 to 3 do {
   DrawArc(4+i*10, height+68, 7, 22)
   FillArc(colors[i], 5+i*10, height+70, 5, 20)
   }
DrawRectangle( 5, height+55, 45, 60, 25, height+50, 5, 5)
DrawCurve(27, height+50, 27, height+47, 15, height+39,
         40, height+20, 25, height+5)
Fg("black")
every i := 0 to height-1 do
   every j := 0 to width-1 do
      DrawRectangle(j*8+lmargin+5, i*8, 8, 8)
DrawLine(0, height, width, height, width, 0)
repeat {
   case e := Event() of {
   "q"|"\e": return
   "s"|"S": {
      if /s | (e=="S") then s := getfilename()
      write("saving image ", s, " size ", width,",", height)
      WriteImage( s, 0, 0, width, height)
```

```
            }
          &lpress | &ldrag | &mpress | &mdrag | &rpress | &rdrag : {
            x := (&x - lmargin - 5) / 8
            y := &y / 8
            if (y < 0) | (y > height-1) | (x > width) then next
            if x < 0 then {
              if &x < 21 then getacolor(1, "left", height)
              else if &x < 31 then getacolor(2, "middle", height)
              else getacolor(3, "right", height)
              until Event() === (&mrelease | &lrelease| &rrelease)
              }
            else dot(x, y, (-e-1)%3)
            }
          }
        }
    end
    procedure dot(x, y, c)
        if (x|y) < 0 then fail
        FillRectangle(colors[c+1], x*8 + lmargin + 5, y*8, 8, 8)
        DrawPoint(colors[c+1], x, y)
        DrawRectangle( x*8 + lmargin + 5, y*8, 8, 8)
    end
    procedure getacolor(n, s, height)
        if ColorDialog(["Set "||s||" button color"],Fg(colors[n]))=="Okay" then {
          Fg(colors[n], dialog_value)
          FillArc(colors[n], 5 + n * 10, height + 70, 5, 20)
        }
    end
    procedure getfilename()
        f := FileDialog()
        f.show_modal()
        return f.file.contents
    end
```

7.6 3D Graphics

Three-dimensional graphics are provided in Unicon on platforms which support the industry standard OpenGL libraries. Unicon provides the basic primitives, transformations, lighting, and texturing elements of 3D computer graphics in a simplified fashion, providing a good basis to rapidly construct 3D scenes. The Unicon 3D interface consists of sixteen new functions and six functions that were extended from the 2D graphics facilities, compared with OpenGL's 250+ functions. While Unicon's 3D interface vastly simplifies some aspects of 3D programming compared with the OpenGL C interface, it does not currently provide

access to several features of OpenGL including blending, fog, anti aliasing, display lists, selection, and feedback.

This section explains in detail how to use Unicon's 3D facilities, for programmers who already have some idea of how 3D graphics work. A 3D window is opened using mode "gl" and is very similar to a 2D window, so many ideas earlier in this chapter are needed for 3D programming. 3D graphics use the 2D windowing functions and attributes and introduce several new ones.

A primary difference between 2D and 3D is that graphics operations in 2D windows use (x,y) integer pixel coordinates relative to the upper left corner of the window, while 3D windows use (x,y,z) real number coordinates in an abstract geometric world. A mobile viewer's position, and the direction they are looking, determine what is visible. Coordinates of 3D objects go through a series of translations, scalings, and rotations to determine their final location; these matrix transformations are used to compose aggregate objects from their parts. In addition to the coordinate system difference, 3D scenes usually employ a rich lighting model, and use materials and textures to draw objects more frequently than a solid foreground color. For this reason, the fg attribute is extended in the 3D realm to denote a foreground *material*, including color as well as how the object appears in different types of lighting.

Opening windows for 3D graphics

To open a 3D graphics window, call the built in function open(), passing in the title of the window to be opened and mode "gl".

```
W := open("win", "gl")
```

As in the 2D facilities, if a window is assigned to the keyword variable &window, it is a default window for subsequent 3D function calls.

3D attributes

Features such as lighting, perspective, textures, and shading give a scene the illusion of being three-dimensional. A Unicon programmer makes use of context attributes to control these features. By assigning new values to various attributes, the programmer controls many aspects of the scene. Attributes to control the coordinate system, field of view, lighting and textures are included in the Unicon 3D graphics facilities.

Some of the most basic context attributes concern the coordinate system. In 3D graphics, x-, y-, and z-coordinates determine where to place an object. The objects that are visible on the screen depend on several things, the eye position, the eye direction, and the orientation of the scene. If these items are not taken into account, the scene desired by the user and the scene drawn may be two very different things.

To think about these attributes, imagine a person walking around a 3D coordinate system. What the person sees becomes the scene viewed on the screen. The eye position specifies where the person is standing. Things close to the person appear larger and seem closer than objects further away. The eye direction specifies where the person is looking. If the person is looking toward the negative z-axis, only the objects situated on the negative z-axis are viewed in the scene. Anything on the positive z-axis is behind the viewer. Finally, the up direction can be described by what direction is up for the person.

The eye position is given by the attribute eyepos. By default this is set to be at the origin or (0, 0, 0). The eye direction is given by the attribute eyedir. By default this is set to be looking at the negative z-axis. The up direction can be specified by the attribute eyeup and by default is (0, 1, 0). The attribute eye allows the user to specify eyepos, eyedir, and eyeup with a single value. Changing any of these attributes causes the scene to redraw itself with the new eye specifications.

Table 7.5 lists the added context attributes used on 3D windows.

Table 7-5
3D Attributes

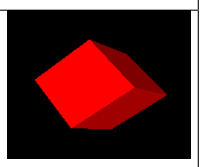
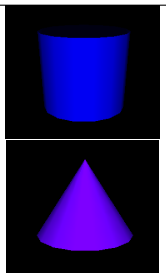
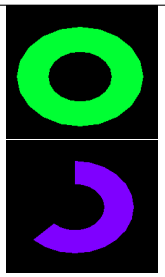
Name	***Type / Example***	***Description : Default***	***Usage***
buffer	boolean / "on"	Buffer mode : off	RW
dim	integer / 3	Dimension : 2	RW
eye	xyz nonuple / "0,0,0,0,0,0,0,0,0"	Eye position,direction,up : "0,0,0,0,0,-1,0,1,0"	RW
eyedir	xyz triple / "0,0,0"	Eye direction/target : "0,0,-1"	RW
eyepos	xyz triple / "0,0,0"	Eye position : "0,0,0"	RW
eyeup	xyz triple / "0,0,0"	Eye up vector : "0,1,0"	RW
meshmode	string / "triangles"	Polygon mesh mode : "polygon"	RW
normals	real array	Normal vectors: n/a	RW
rings	integer	Number of rings in spheres/cylinders : 10	RW
selection	boolean / "off"	Selection	RW
slices	integer	Number of slices in spheres, cylinders : 15	RW
texcoord	vector of reals	Texture (u,v) coordinates	RW
texture	image	Texture	RW

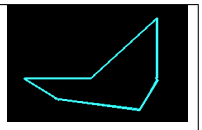
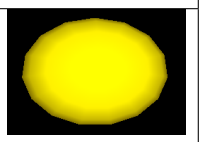
3D drawing primitives

In 2D, programs draw points, lines, polygons, and circles. Functions that have been extended for 3D include DrawPoint(), DrawLine(), DrawSegment(), DrawPolygon(), and FillPolygon(). The 3D facilities introduce many new primitives, including cubes, spheres, tori, cylinders, and disks. These are described in Table 7-6 below.

Many scenes are drawn using a mixture of 2D, 3D and 4D objects. The context attribute dim allows the program to switch between the different dimensions when specifying the vertices an objects. A user can draw 2D, 3D, or 4D objects by assigning dim the values of 2, 3, or 4. For primitives that take x, y, and z coordinates, specifying only x and y coordinate is not sufficient. For this reason, "dim = 2" disallows the use of these primitives. These functions are DrawSphere(), DrawTorus(), DrawCube(), and DrawCylinder(). By default the value of dim is three.

Table 7-6
Types of 3D Primitives

Primitive	Function	Parameters	Picture
Cube	DrawCube()	x, y, and z coordinates of the lower left front corner, and the length of the sides.	
Cylinder	DrawCylinder()	x, y, and z coordinates of the center, the height, the radius of the top, the radius of the bottom. If one radius is smaller than the other, a cone is formed.	
Disk	DrawDisk()	x, y, and z coordinates of center, the radius of the inner circle, and the radius of the outer circle. An additional two angle values specify a partial disk.	
Solid Polygon	FillPolygon()	x, y, and z coordinates of each vertex of the polygon.	
Line	DrawLine()	x, y, and z coordinates of each vertex.	

Polygon	DrawPolygon()	x, y, and z coordinates of each vertex.	
Point	DrawPoint()	x, y, and z coordinates of each point.	
Segment	DrawSegment()	x, y, and z coordinates of each vertex.	
Sphere	DrawSphere()	x, y, and z coordinates of center and the radius of the sphere.	
Torus	DrawTorus()	x, y, and z coordinates of the center, an inner radius and an outer radius.	

Coordinate transformations

Matrix multiplications are used to calculate transformations such as rotations on objects and the field of view. Functions to perform several matrix operations in support of coordinate transformation are available. The main transformation functions are Translate(dx,dy,dz), Scale(mx,my,mz), and Rotate(a,x,y,z).

In many 3D graphics applications, transformations are composed as the pieces of an object are drawn relative to one another. Transformations are saved and restored as objects are traversed. Unicon uses the system's matrix stacks to keep track of the current matrix with a stack of matrices, where the top of the stack is the current matrix. Several functions manipulate the matrix stack. The function PushMatrix() pushes a copy of the current matrix onto the stack. By doing this the user can compose several different transformations. The function IdentityMatrix() changes the current matrix to the identity matrix. To discard the top matrix and to return to the previous matrix, the function PopMatrix() pops the top matrix off the matrix stack.

There are different matrix stacks for the projection and model view. The projection stack contains matrices that perform calculations on the field of view, based on the current eye attributes. If these eye attributes are changed, previous manipulations of the projection matrix stack are no longer valid. The maximum depth of the projection matrix stack is two. Trying to push more than two matrices onto the projection matrix stack will generate a runtime error. The model view stack contains matrices to perform calculations on objects within the scene. Transformations on the model view stack affect the subsequently drawn objects. The maximum depth of this stack is 32; pushing more than 32 matrices on the model view stack results in an error. Furthermore, only one matrix stack can be

manipulated at any given time. The function MatrixMode() switches between the two matrix stacks.

Lighting and materials

Lighting is important in making a graphics scene appear to be 3D. Adding lighting to a scene can be complicated and the hardware support for lighting is at present a very crude approximation. Light sources emit different types of light. *Ambient* light has been scattered so much that is difficult to determine the source; backlighting in a room is an example. *Diffuse* light comes from one direction and is central in defining what color the object appears to be. Finally, *specular* light not only comes from one direction, but also tends to bounce off the objects in the scene.

Applications control lighting using context attributes set using WAttrib(). For a 3D scene in Unicon, eight lights are available. Attributes light0 - light7 control the eight lights. Each light can be turned on or off and has a position and lighting value.A lighting value is a string which contains one or more semi-colon separated lighting properties. A lighting property is of the form

```
[diffuse|ambient|specular] color name
```

A new lighting value can be specified without turning the light on or off. The following call turns light1 on and gives it diffuse yellow and ambient gold lighting properties.

```
WAttrib(w, "light1=on, diffuse yellow; ambient gold")
```

The following expression sets light0 to the default values for the lighting properties.

```
WAttrib(w, "light0=diffuse white; ambient black; _
        specular white; position 0.0, 1.0, 0.0")
```

Interacting with the lights, the objects in a scene may have several material properties. The material properties are ambient, diffuse, and specular, which are similar to the light properties, plus emission, and shininess. If an object has an emission property, it emits light of a specific color. Using combinations of these material properties one can give an object the illusion of being made of plastic or metal.

In 2D, the foreground color is controlled using the context attribute fg and set with Fg() or WAttrib(). In 3D, the attribute fg is extended to allow a semi-colon separated list of material properties with the color that property should have. A programmer can specify a material property as a simple color value or by providing comma-separated red, green, and blue intensities as real numbers between 0.0 and 1.0. More general material properties are of the form

```
[ diffuse | ambient | specular | emission ] color name
```

or "shininess n", where n is some integer in the range 0 <= n <= 128.

The default material property type is diffuse, so the call Fg("red") is equivalent to Fg("diffuse red"). For shininess, a value of 0 spreads specular light broadly across an object and a value of 128 focuses specular light at a single point. The following line of code changes the current material property to diffuse green and ambient orange.

```
WAttrib(w, "fg=diffuse green; ambient orange")
```

The default values of the material properties are given in the following example.

```
Fg(w, "diffuse light grey; ambient grey; specular black; emission black; _
      shininess 50")
```

Using lights and materials in Unicon was simplified by extending the design of the 2D graphics facilities. The fg attribute greatly reduces the number of lines of code needed for a scene. Thanks to this design along with the extensive use of defaults, a programmer can use lighting in a 3D graphics application without much effort.

7.7 Textures

Another important area of 3D graphics is textures. Adding textures to a scene can give a scene a realistic feel. There are several aspects to using textures. A texture is a rectangular image that is "glued" onto objects in a scene. The appearance of the textured objects in the scene depends on several pieces of information supplied by the programmer. These include the texture image and what parts of the texture image is mapped to what parts of the object.

The attribute texmode enables or disables textures, which are disabled by default. WAttrib("texmode=on") enables textures. When textures are enabled and a texture image is given, the texture is applied to the objects drawn in the scene.

Unicon provides several formats to specify a texture image. A texture can be a Unicon window, an image file, or a string. String textures are encoded in one of the language standard formats "width,pallet,data" or "width,#data" described in the 2D graphics facilities. In the first case the pallet will determine what colors appear in the texture image. In the second case, the foreground color and background color are used. The ability to use another Unicon window as a texture provides great flexibility for texture images, allowing programs to create texture images dynamically.

Textures must have a height of 2^n pixels and width of 2^m pixels where n and m are integers. If not, the texture dimensions are automatically scaled down to the closest power of 2. Rescaling affects application performance and may cause visual artifacts, so it may be wise to create textures with appropriate sizes in the first place. Examples of how to use textures specified in the different forms are given below.

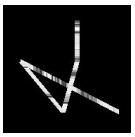
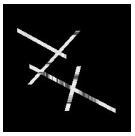
A programmer specifies a texture either by calling WAttrib("texture=...") or using Texture(t). These methods differ only in that a window cannot be used as a texture with WAttrib(), so Texture() must be called when a window is used as a texture.

A program can specify how a texture is applied to a particular object by specifying texture coordinates and vertices. Texture coordinates are x and y coordinates within the texture; texture coordinate (0.0, 0.0) is the lower left corner of the texture image. Texture coordinates are mapped to the vertices of an object in the scene. Together, the texture coordinates and the vertices determine what the object looks like after textures have been applied. Since texture coordinates are complex, defaults are provided. Assigning attribute texcoord the value auto causes system default texture coordinates to be used. The defaults are dependent on the type of primitive.

Non-default texture coordinates are given in several ways, such as WAttrib("texcoord=*s*") where *s* is a comma separated string of real number values between 0.0 and 1.0. Each pair of values is taken as one texture coordinate; there must be an even number of real values or the assignment of texture coordinates fails. One can assign texture coordinates by calling Texcoord(x1,y1,...) where x and y are real number values between 0.0 and 1.0. Finally one can use Texcoord(L) where L is a list of real number texture coordinates. The texture coordinates given by the programmer are used differently depending on the type of primitive to be drawn. If the primitive is a point, line, line segment, polygon, or filled polygon, then a texture coordinate given is assigned to each vertex. If there are more texture coordinates than vertices, unused texture coordinates are ignored. If there are more vertices than texture coordinates the application of a texture will fail. In order to use non-default texture coordinates with cubes, tori, spheres, disks, and cylinders a programmer should approximate the desired mapping with filled polygons. These specifications are given in Table 7-7.

Table 7-7
Texture coordinates and primitives

Primitive	Default Texture Coordinates (from [OpenGL00] chapter 6)	Effect of Texture Coordinates	Picture
Cube	The texture image is applied to each face of the cube.	None	

Sphere Cylinder	The y texture coordinate ranges linearly from 0.0 to 1.0. On spheres this is from z= -radius to z=radius; on cylinders, from z = 0 to z = height. The x texture coordinate ranges from 0.0 at the positive y-axis to 0.25 at the positive x-axis, to 0.5 at the negative y-axis to 0.75 at the negative x-axis back to 1.0 at the positive y-axis.	None	 
Filled Polygon Line Polygon Segment	The x and y texture coordinates are given by $p_1x_0+p_2y_0+p_3z_0+p_4w_0$	A texture coordinate is assigned to a vertex.	   

Torus	The x and y texture coordinates are given by $p_1x_0+p_2y_0+p_3z_0+p_4w_0$	None	

3D Examples

Changing Context Attributes The user can change attributes throughout a program. Multiple attributes can be changed with one call to WAttrib(). The following line of code changes the eye position to (0.0, 0.0, 5.0) and the eye direction to look at the positive z-axis. An assignment to eyepos, eyedir, eyeup or eye redraws the screen; a given call to WAttrib() will only redraw the scene once.

```
WAttrib("eyepos=0.0,0.0,5.0","eyedir=0.0,0.0,1.0")
```

The values of the attributes can also be read by using the function WAttrib(). The current eye position could be stored in variable ep by the call:

```
ep := WAttrib("eyepos")
```

Drawing Primitives Here is an example that uses some of the drawing primitives.

```
Fg(w, "ambient yellow")
DrawDisk(w, 0.4, -0.5, -4.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.5, -5.0, 0.5, 1.0)
Fg(w, "diffuse white")
DrawDisk(w, 0.4, -0.5, -4.0, 0.0, 1.0, 0.0, 225.0,1.0, 0.5, -5.0, 0.5,1.0,0.0,125.0)
Fg(w, "ambient pink")
DrawCylinder(w, 0.0, 1.0, -5.0, 1.0, 0.5, 0.3)
Fg(w, "specular navy")
DrawDisk(w, -0.5, -0.5, -2.0, 0.5, 0.3)
Fg(w, "emission green")
DrawSphere(w, 0.5, 1.0, -3.0, 0.5)
WAttrib(w, "light0=on, diffuse white")
```

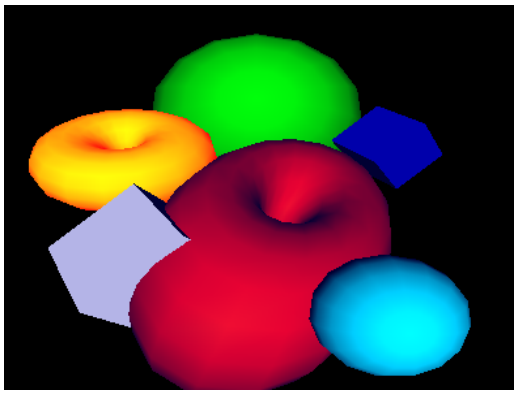


Figure 7-2: 3D Drawing Primitives Made From Various Materials

The function Fg() specifies the material properties of subsequently drawn objects that affect their color and appearance. In this example, a cube with a diffuse green material is drawn with sides of length 0.7. Then a sphere with a diffuse purple and ambient blue material is drawn with radius 0.5 and center (0.4, -0.5, -4.0). Next a diffuse yellow and ambient grey torus with center (-1.0, 0.4, -4.0), an inner radius of 0.4, and an outer radius of 0.5 is drawn. Finally a filled polygon with a diffuse red material property and three vertices, (0.25, -0.25, -1.0), (1.0, 0.25, -4.0) and (1.3, -0.4, -3.0) is drawn.

Different Types of Lighting The next example shows the difference between the different types of lighting that can be used in a scene. Each window is the same scene rendered using different lighting. The upper right scene has an ambient blue-green light. The upper left scene was drawn using a diffuse blue-green light. The lower right scene uses only a specular blue-green light. The scene in the lower left uses all three types of lighting.

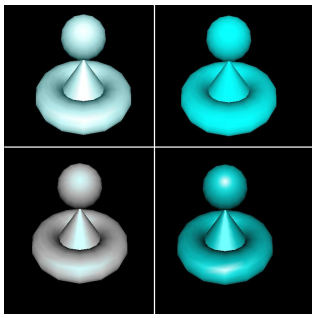


Figure 7-3: Different Types of Lighting

```
w := open("ambient.icn","gl", "bg=black", "size=400,400")
    WAttrib(w,"light0=on, ambient blue-green","fg=specular white")
    DrawCylinder(w, 0.0, -0.2, -3.5, 0.75, 0.5, 0.0)
    DrawTorus(w,0.0, -0.2, -3.5, 0.3, 0.7)
    DrawSphere(w,0.0, 0.59, -2.2, 0.3)
x := open("diffuse.icn","gl", "bg=black", "size=400,400")
    WAttrib(x,"light0=on, diffuse blue-green","fg=specular white")
    DrawCylinder(x, 0.0, -0.2, -3.5, 0.75, 0.5, 0.0)
    DrawTorus(x,0.0, -0.2, -3.5, 0.3, 0.7)
    DrawSphere(x, 0.0, 0.59, -2.2, 0.3)
y := open("specular.icn","gl", "bg=black", "size=400,400")
    WAttrib(y,"light0=on,specular blue-green","fg=specular white")
    DrawCylinder(y, 0.0, -0.2, -3.5, 0.75, 0.5, 0.0)
    DrawTorus(y, 0.0, -0.2, -3.5, 0.3, 0.7)
    DrawSphere(y, 0.0, 0.59, -2.2, 0.3)
z := open("all.icn","gl", "bg=black", "size=400,400")
    WAttrib(z, "light0=on, diffuse blue-green; _
```

```
    specular blue-green; ambient blue-green","fg=specular white")
DrawCylinder(z, 0.0, -0.2, -3.5, 0.75, 0.5, 0.0)
DrawTorus(z, 0.0, -0.2, -3.5, 0.3, 0.7)
DrawSphere(z, 0.0, 0.59, -2.2, 0.3)
```

Figure 7-4 shows the effects of emission color on an object.

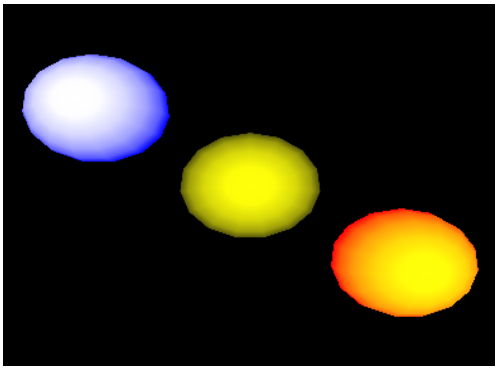


Figure 7-4: Mixing Emission and Diffuse Material Properties

```
Fg(w, "emission blue; diffuse yellow")
DrawSphere(w, -1.5, 1.0, -5.0, 0.7)
Fg(w, "emission black")
DrawSphere(w, 0.0, 0.0, -5.0, 0.7)
Fg(w, "emission red")
DrawSphere(w, 1.5, -1.0, -5.0, 0.7)
```

In the above example, three yellow spheres are drawn. An emission color of blue makes the sphere appear white with a blue ring. With a red emission color, the sphere remains yellow, but now has an orange-red ring. The middle sphere shows the effect of having no emission color. In order to obtain the diffuse yellow sphere in the center, the emission color was changed to black, without changing the diffuse property.

Textures This section contains examples of the use of textures in a scene. The following example uses a file as a texture. A .gif image of a map of the world is used to texture a torus using the default texture coordinates.

Figure 7-5: A Texture from a GIF Image is Mapped onto a Torus

```
WAttrib(w, "texmode=on", "texture=map.gif")
DrawTorus(w, 0.0, 0.0, -3.0, 0.3, 0.4)
```

Instead of using WAttrib(w, "texture=map.gif") to specify the .gif file, a call to Texture(w, "map.gif") could be used to obtain the same result.

The next example uses an image string to specify a texture image. The string used for this example is taken from Graphics Programming in Icon [Griswold98] page 156. This string is used as a texture on a cube using the default texture coordinates.

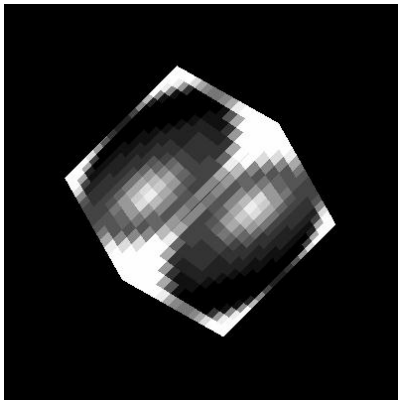


Figure 7-6: A Texture Supplied via an Image String

```
WAttrib(w, "texmode=on")
  sphere:= "16,g16, FFFFB98788AEFFFF" ||
    "FFD865554446AFFF FD856886544339FF E8579BA9643323AF"||
    "A569DECA7433215E 7569CDB86433211A 5579AA9643222108"||
    "4456776533221007 4444443332210007 4333333222100008"||
    "533322221100000A 822222111000003D D41111100000019F"||
    "FA200000000018EF FFA4000000028EFF FFFD9532248BFFFF"
  Texture(w, sphere)
  DrawCube(w, 0.0, 0.0, -3.0, 1.2)
```

The next example shows the use of a Unicon window as a texture. An image of a lamp is drawn on the first window in gl mode. This window is then used as a texture on a cylinder. The same method can be used to embed 2D window contents in 3D scenes. Note that in the following code the first window is opened with size 256 x 256. Texture images must have height and width that are powers of 2, or the system must rescale them. The default coordinates for cylinders are used.

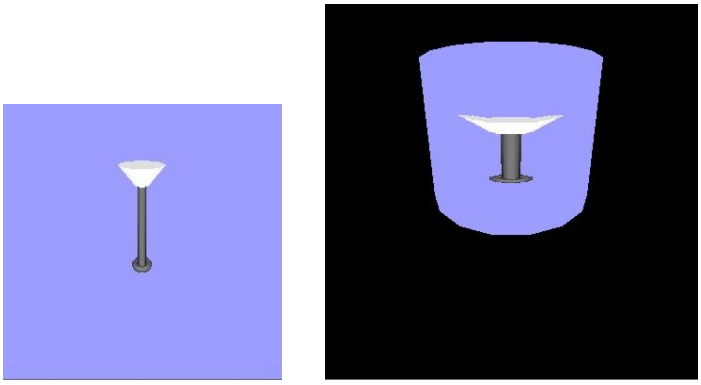


Figure 7-7: A Texture Obtained from Another Window's Contents

```
w := open("win1","gl","bg=light blue","size=256,256")
Fg(w, "emission pale grey")
PushMatrix(w)
Rotate(w, -5.0, 1.0, 0.0, 0.0)
DrawCylinder(w, 0.0, 0.575, -2.0, 0.15, 0.05, 0.17)
PopMatrix(w)
Fg(w, "diffuse grey; emission black")
PushMatrix(w)
Rotate(w, -5.0, 1.0, 0.0, 0.0)
DrawCylinder(w, 0.0, 0.0, -2.5, 0.7, 0.035, 0.035)
PopMatrix(w)
DrawTorus(w, 0.0, -0.22, -2.5, 0.03, 0.06)
DrawTorus(w, 0.0, 0.6, -2.5, 0.05, 0.03)

w2 := open("win2.icn","gl","bg=black","size=400,400")
WAttrib(w2, "texmode=on")
Texture(w2, w)
Fg(w2, "diffuse purple; ambient blue")
DrawCylinder(w2, 0.0, 0.0, -3.5, 1.2, 0.7,  0.7)
```

The next two examples illustrate the use of the default texture coordinates versus texture coordinates specified by the programmer. In both examples, a bi-level image is used as the texture image. The format for such a string is described in section 2.7. This image is taken from Graphics Programming in Icon page 159. The first example uses the default texture coordinates for a filled polygon, which in this case is just a square with sides

of length one. In this case the default texture coordinates are as follows. The coordinate (0.0, 0.0) of the texture image is mapped to the vertex (0.0, 0.0, -2.0) of the square, (0.0, 1.0) is mapped to (0.0, 1.0, -2.0), (1.0, 1.0) is mapped to (1.0, 1.0, -2.0), and (1.0, 0.0) is mapped to (1.0, 0.0, -2.0).

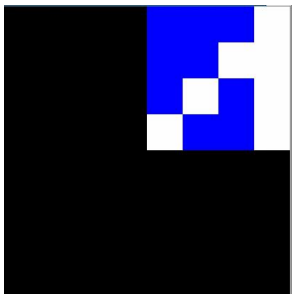


Figure 7-8: Default Texture Coordinates

```
WAttrib(w,"fg=white","bg=blue","texmode=on","texture=4,#8CA9")
Fg(w, "diffuse purple; ambient blue")
FillPolygon(w, 0.0, 0.0, -2.0, 0.0, 1.0, -2.0, 1.0, 1.0, -2.0, 1.0, 0.0, -2.0)
```

This example uses the same texture image and the same object to be textured, but instead uses the texture coordinates (0.0, 1.0), (1.0, 1.0), (1.0, 1.0), and (1.0, 0.0). So the coordinate (0.0, 1.0) of the texture image is mapped to the vertex (0.0, 0.0, -2.0) of the square, (1.0, 1.0) is mapped to (0.0, 1.0, -2.0),(1.0, 1.0) is mapped to (1.0, 1.0, -2.0), and (1.0, 0.0) is mapped to (1.0, 0.0, -2.0).

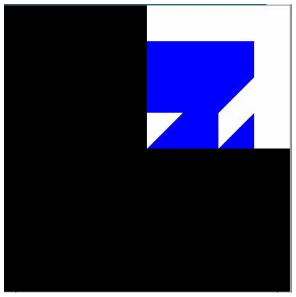


Figure 7-9: Custom Texture Coordinates

```
WAttrib(w,"fg=white","bg=blue","texmode=on","texture=4,#8CA9",
      "texcoord=0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0")
FillPolygon(w, 0.0, 0.0, -2.0, 0.0, 1.0, -2.0, 1.0, 1.0, -2.0, 1.0, 0.0, -2.0)
```

Instead of using WAttrib() with the attribute texcoord, the function Texcoord() could be used. So the line

```
WAttrib(w,"texcoord=0.0, 1.0, 1.0, 1.0, 1.0, 1.0,1.0, 0.0")
```

could be replaced by

```
Texcoord(w, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0,1.0, 0.0)
```

A Larger Textures Example The following more complicated example uses many features of the Unicon 3D graphics facilities described in the previous sections. This example also illustrates the effect of adding texture to a scene. The scene on the left is a scene drawn without any texturing. The scene on the right contains texturing. The scene on the right is a much more realistic scene than the one on the left.

All textures used in the textured scene, except for the unicorn, where captured using a digital camera. These images were then converted into .gif files and scaled to width and height of 2^n. Directly using an image file is one feature of the Unicon 3D graphics facilities that makes adding textures simpler than using OpenGL.

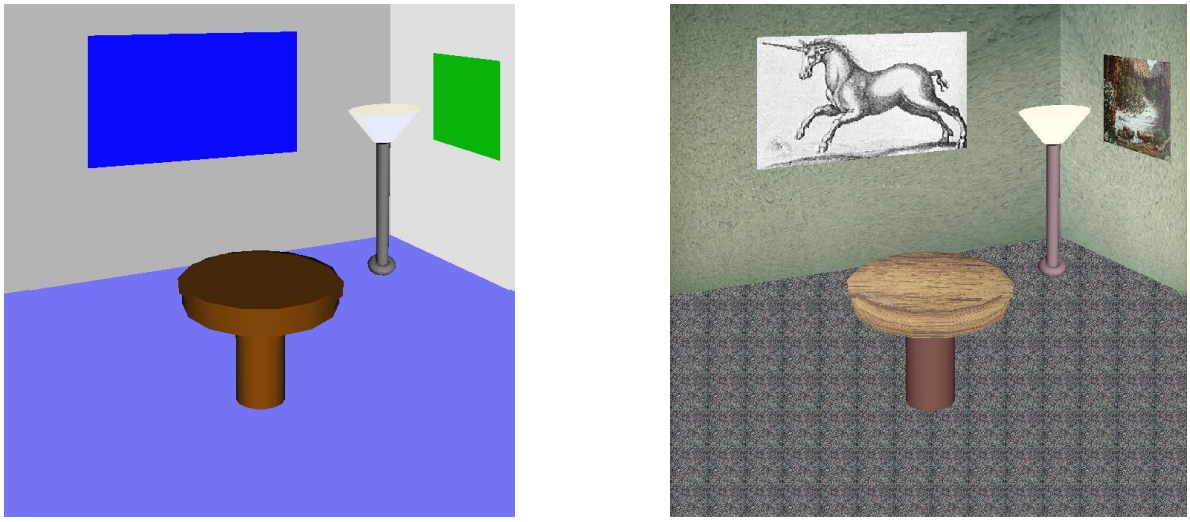


Figure 7-10: Untextured and Textured Versions of the Same Scene

```
procedure main()
    &window := open("textured.icn","gl","bg=black","size=700,700")

    # Draw the floor of the room
    WAttrib("texmode=on", "texture=carpet.gif")
    FillPolygon(-7.0, -0.9, -14.0, -7.0, -7.0, -14.0,
            7.0, -7.0, -14.0, 7.0, -0.9, -14.0, 3.5, 0.8, -14.0)

    # Draw the right and left walls
    WAttrib("texture=wall1.gif", "texcoord=0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0")
    FillPolygon(2.0, 4.0, -8.0, 8.3, 8.0, -16.0, 8.3, -1.2, -16.0, 2.0, 0.4, -8.0)
    WAttrib("texture=wall2.gif")
    FillPolygon(2.0, 4.0 ,-8.0, -9.0, 8.0, -16.0, -9.0,-1.2,-16.0, 2.0, 0.4, -8.0)

    # Draw a picture
    WAttrib("texture=poster.gif", "texcoord=0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0")
```

```
    FillPolygon(1.0, 1.2, -3.0, 1.0, 0.7, -3.0, 1.2, 0.5, -2.6, 1.2, 1.0, -2.6)
    # Draw another picture
    WAttrib("texture=unicorn.gif", "texcoord=1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0")
    FillPolygon(0.8, 2.0, -9.0, -3.0, 1.6, -9.0, 3.0, 3.9,-9.0, 0.8, 4.0, -9.0)
    # Draw the lamp
    WAttrib("texmode=off")
    PushMatrix()
    Translate(0.7, 0.20, -0.5)
    Fg("emission pale weak yellow")
    PushMatrix()
    Rotate(-5.0, 1.0, 0.0, 0.0)
    Rotate( 5.0, 0.0, 0.0, 1.0)
    DrawCylinder(-0.05, 0.570, -2.0, 0.15, 0.05, 0.17)
    PopMatrix()
    Fg("diffuse grey; emission black")
    PushMatrix()
    Rotate(-5.0, 1.0, 0.0, 0.0)
    Rotate( 6.0, 0.0, 0.0, 1.0)
    DrawCylinder(0.0, 0.0, -2.5, 0.7, 0.035, 0.035)
    PopMatrix()
    PushMatrix()
    Rotate(6.0, 0.0, 0.0, 1.0)
    DrawTorus(-0.02, -0.22, -2.5, 0.03, 0.05)
    PopMatrix()
    PopMatrix()

    # Draw the table
    WAttrib("texcoord=auto", "texmode=on", "texture=table.gif")
    PushMatrix()
    Rotate(-10.0, 1.0, 0.0,0.0)
    DrawCylinder(0.0, 0.2, -2.0, 0.1, 0.3, 0.3)
    PopMatrix()
    PushMatrix()
    Translate(0.0, -0.09, -1.8)
    Rotate(65.0, 1.0, 0.0, 0.0)
    DrawDisk(0.0, 0.0, 0.0, 0.0, 0.29)
    PopMatrix()
    WAttrib("texmode=off", "fg=diffuse weak brown")
    PushMatrix()
    Rotate(-20.0, 1.0, 0.0,0.0)
    DrawCylinder(0.0, 0.2, -2.2, 0.3, 0.1, 0.1)
    PopMatrix()
    while (e := Event()) ~== "q" do write(image(e), ": ", &x, ",", &y)
end
```

In order to apply textures to the scene, texturing must be turned on. Next, the texture to be applied is specified. The floor of the scene is drawn using a filled polygon. The default texture coordinates are used to apply the carpet texture to the floor of the room. The tiled appearance on the floor is caused by the use of the default texture coordinates. This can be avoided using user-supplied texture coordinates, as is done for the textures that are applied to the walls and the pictures in the room.

The lamp does not have a texture, so it is necessary to turn off texturing before drawing the lamp. Also for the lamp to be centered properly in the room, transformations are used. Matrices are used to isolate the transformations of the lamp. Finally to draw the table with a textured top and an untextured base, two cylinders and a disk are used. Texturing is applied to a cylinder and the disk. Notice the call

```
WAttrib(w, "texcoord=auto")
```

This resets the texture coordinates to the defaults. Finally, texturing is turned off to draw the base of the table.

Animation

Graphics animation is performance sensitive, and Unicon is slower than systems programming languages such as C and C++. Nevertheless, it is possible to write 3D animations in Unicon with acceptable frame rates.

3D animations redraw the entire scene each time an object moves or the user changes point of view. An application can call EraseArea() followed by the appropriate graphics primitives to redraw a scene, but the results often appear to flicker. It is better to let Unicon's runtime system do the redrawing. Unicon maintains a *display list* of graphics operations to execute whenever the screen must be redrawn; these operations are effectively everything since the last EraseArea(). The display list for a window can be obtained by calling WindowContents(). The elements of the list are Unicon records and lists containing the string names and parameters of graphics primitives. For example, a call to DrawSphere(w,x,y,z,r) returns (and adds to the display list) a record gl_sphere("DrawSphere",x,y,z,r). Instead of redrawing the entire scene to move an object, you can modify its display list record and call Refresh(). The following code fragment illustrates animation by causing a ball to slide up and down. In order to *bounce*, the program would need to incorporate physics.

```
sphere := DrawSphere(w, x, y, z, r)
increment := 0.2
every i := 1 to 100 do
    every j := 1 to 100 do {
      sphere.y +:= increment
      Refresh(w)
      }
```

This technique gives animation rates of hundreds of frames per second on midrange PC hardware. Unicon supports smooth animation for a number of objects which varies widely depending on the underlying graphics hardware and software.

Selective rendering and object selection

Many 3D applications model scenes with far more objects than are needed at any particular instant. For example, a virtual building might have many rooms on multiple floors, but only a small fraction is visible from any particular location. The 3D facilities remove objects that are not visible, but doing so becomes too slow for large numbers of objects. An application with a large scene will generally have to perform at least approximate visibility calculations to achieve smooth animation. Such visibility calculations can be performed for each frame, and if the visible objects change, the scene can be re-rendered by rebuilding the display list from scratch. At this point Unicon's speed can be an issue, as discussed in the previous section.

The function WSection() comes to the rescue. It plays two vital roles. First, it allows portions of the display list to be skipped during rendering, without having to rebuild the display list. Second, it forms the basis for specifying portions of the display list that the user may select (click on) when interacting with the scene. In both cases, calls to WSection() come in pairs, first a call WSection(s) identifies a portion of the display list of interest, then the sequence of 3D calls to render some object or portion of the scene, then a call to WSection() defines the end of that section. Parameter s must be a unique string name or identifier for the section.

The call to create a new section returns a record that contains a field named skip. Setting skip to a non-null value causes the section to be omitted whenever the scene is redrawn. Using WSection() for 3D user input is similar. A program calls WAttrib("pick=on") to turn on 3D selection, after which keyword &pick generates the identifying names for all objects intersected by the ray from the camera through the (x,y) screen location where the mouse was located on the last call to Event().

7.8 Summary

Graphics are ubiquitous in modern applications. Unicon provides 2D and 3D graphics capabilities that are easy to use, portable building blocks for many programs. The 2D facilities are mature; the 3D interface is new and will evolve. Many elements of the 2D graphics system are used in the 3D graphics interface. Further integration of the 2D and 3D graphics systems is likely in the future.

Chapter 8

Threads

Threads are building blocks for concurrent (also known as *parallel*) execution. In a concurrent program, the instructions specify multiple things to compute at the same time, during some or most of the program run. On a classic single-processor system these computations will only happen one at a time, but on most modern multiprocessor and multicore systems, the hardware is designed to do several computations at once, and if your program does not ask for as much, it is underutilizing (sometimes severely) the platform.

This chapter describes concurrency in Unicon. It is based on Unicon Technical Report 14; UTR14 on the unicon.org site may amend or supercede this chapter with added features in the future. Threads are an extension of the co-expression type described in Chapter 4 and the system interface described in Chapter 5. Consulting those chapters may be helpful in studying this one.

Concurrent programming introduces techniques, concepts, and difficulties that do not exist in sequential programs. In some situations concurrent programming is a natural way to write programs, such as a server where threads serve different clients. In other situations, concurrent programming improves performance. Long-running programs can run faster by having several threads running cooperatively on several CPU cores, or programs that do a lot of slow I/O operations can allow other non-blocked threads to proceed and utilize the CPU. However, for programs that have a lot of dependencies and are sequential in nature, the complexities of parallelizing them can outweigh the benefits.

This chapter is not a comprehensive concurrent programming guide. It assumes that the reader has some basic knowledge about threads, their programming techniques, and problems, such as synchronization and race conditions. Readers who are unfamiliar with concurrency can refer to a myriad of resources such as [Andr83] or [Bute97] for an overview. Since Unicon's concurrency facilities are implemented on top of POSIX threads (pthreads), many of the concepts from pthreads programming apply, often with more concise, or higher-level ways of writing things.

8.1 Threads and Co-Expressions

Co-expressions are independent, explicitly sequential execution contexts. Only one co-expression is active at any given moment. When a co-expression is activated, the calling co-expression blocks until the child co-expression returns the execution to it or fails. Threads, on the other hand, can run simultaneously and independently. Threads in Unicon are like special co-expressions that are marked to run asynchronously.

In a concurrent program with two or more threads, each thread has its own program counter, stack pointer, and other CPU registers. However, all of the threads in a program share the address space, open files, and many other pieces of process-wide information. This enables very fast communication and cooperation between threads, which leads to less blocking, faster execution and more efficient use of resources.

Unicon programs start execution in the main() procedure. In a multi-thread programming environment, the procedure main() is the entry point for a special thread referred to as the *main thread*. This main thread is created by the operating system when the program begins execution. The main thread can create new threads, which can create even more threads. Each thread has an entry point, where it begins executing. Usually this is a procedure but it can be any Unicon expression, as is the case for co-expressions. When a thread first starts running in the entry point, it goes on its own execution path, separate from the thread that created it, which continues to run. A thread never returns. When it ends, it simply terminates; other threads continue to run. An important exception is the main thread; if the main thread ends, the whole program ends. If there are any other threads running, all of them will be terminated.

Since the emergence of the first computer, processors have been increasing in computational power. CPU speeds grew faster than almost all of the other units in the computer, especially the I/O units. This causes programs, especially those which are I/O bound, to spend most of their execution time blocked, waiting for I/O to complete. On systems with multitasking support, several programs run at the same time. When one program blocks for I/O for example, another program is scheduled to run, allowing a better utilization of the system resources. Multitasking offered a way to increase the overall system throughput and boosted the utilization of the increasingly powerful processors. However, multitasking could not help make a process run faster, even on multiprocessor systems.

8.2 First Look at Unicon Threads

Unicon threads facilities give the programmer flexibility in choosing the programming styles that suit the problem at hand. In many situations the same problem can be solved in different ways, using implicit features or explicit ones. The following sections cover the functions and features provided by the thread facilities in Unicon.

Thread creation

Threads can be created in two ways in Unicon, using the thread reserved word or using the function spawn(). The difference between the two is the separation between creating a thread and running it. The thread reserved word creates a thread and starts its execution. The function spawn() however, takes a previously created co-expression and turns it into a thread. In many cases the thread reserved word allows more concise code. spawn() on the other hand is useful in situations where several threads need to be created and initialized before running them. spawn() also takes optional parameters to control some aspects of the newly-created thread. The following code creates and runs a hello world thread:

```
thread write("Hello World!")
```

This is equivalent to

```
spawn( create write("Hello World!"))
```

or to

```
co := create write("Hello World!")
spawn(co)
```

Both thread and spawn() return a reference to the new thread. The following program creates 10 threads:

```
procedure main()
    every i := !10 do thread write("Hello world! I am thread: ", i )
    write("main: done")
end
```

In this example, the main thread continues to execute normally after firing 10 threads. Because of the non-deterministic nature of threads, there is no guarantee which thread gets to print out its "hello world" message first, or in what order the messages are printed out, including the message from the main thread "main: done". All of the possible permutations are valid. No assumptions can be made about which thread will continue running or finish first. It depends on the host OS CPU process/thread scheduler. The order is unpredictable.

Furthermore, the main thread might finish and terminate the program before some or all of the threads get executed or print out messages. To avoid such situations, the main thread needs to wait for other threads to finish before exiting the program. This is can be achieved by using the function wait(), which blocks the calling thread until the target thread is done. The above program can be rewritten as follows:

```
procedure main()
    L := [ ]
```

```
    every i := !10 do put(L, thread write("Hello world! I am thread: " , i))
    every wait(!L)
    write("main: done")
end
```

wait(!L) tells the main the thread to wait for every thread to finish, causing the message "main: done" to be the last thing printed out before the program ends. wait() is useful in cases where threads need to synchronize so that one thread blocks until another finishes. wait() provides a very basic synchronization technique, but most concurrent programming tasks need more synchronization than waiting for a thread to finish. Advanced synchronization mechanisms are discussed below.

Thread evaluation context

Similar to co-expressions, threads have their own stack, starting from a snapshot of parameters and local variables at creation time. This allows co-expressions and threads to be used outside the scope where they are created. It also allows a thread to start by using the variable values at the time of its creation, rather than when running it in the case of spawn(). An important side effect of this process is avoiding race conditions, because each thread gets a copy of the variables instead of having all the threads competing over the same shared variables. Race conditions and thread-safe data will be covered in depth in the following sections. The following example and its output demonstrate the idea of an evaluation context:

```
procedure main()
    local x:= 10, y:=20, z:=0
    write( "Main thread: x=", x, ", y=", y, ", z=", z)
    thread (x:=100) & write("Thread 1: x=", x)
    thread (y:=200) & write("Thread 2: y=", y)
    thread (z:=x+y) & write("Thread 3: z=", z)
    delay(1000)
    write( "Main thread: x=", x, ", y=", y, ", z=",z)
end
```

The output is:

```
Main thread: x=10, y=20, z=0
Thread 3: z=30
Thread 1: x=100
Thread 2: y=200
Main thread: x=10, y=20, z=0
```

The delay(1000) should give the threads enough time to finish before the main program finishes. This should not be left to chance: wait() will block until the threads finish, instead of a 1 sec delay.

The output shows that the changes to the variables are per-thread, and not visible in the main thread or in the other threads. The copies of local variables in different threads can be thought of as passing parameters by value to a procedure. This is true for *local* variables of *immutable* data types; on the other hand, *global* variables and *mutable* types, such as lists, are shared. Any change in the structure of such types is visible across all threads. Contrast the following example with the one above:

```
procedure main()
    local L
    L := [20, 10, 0]
    write( "Main thread: L[1]=", L[1], ", L[2]=", L[2], ", L[3]=", L[3])
    thread (L[1]:=100) & write("Thread 1: L[1]=", L[1])
    thread (L[2]:=200) & write("Thread 2: L[2]=", L[2])
    thread (L[3]:=L[1]+L[2]) & write("Thread 3: L[3]=", L[3])
    delay(1000)
    write( "Main thread: L[1] =", L[1], ", L[2]=", L[2], ", L[3]=",L[3])
end
```

with output

```
Main thread: L[1]=20, L[2]=10, L[3]=0
Thread 2:  L[2]=200
Thread 3:  L[3]=300
Thread 1:  L[1]=100
Main thread: L[1] =100, L[2]=200, L[3]=300
```

Instead of using 3 variables x, y, and z, a list of size 3 is used. x from the previous example maps to L[1], y to L[2], and z to L[3]. The program does the same thing as before, but any change to the content of L is visible in other threads. Unlike the output in the first case, where the values of x, y, and z remained the same in the main thread, this output shows that the changes to the list elements in the other threads were visible in the main thread.

Passing arguments to threads

When creating a new thread for a procedure, the parameters that are passed to the procedure at creation time can be thought of as a one-time one-way communication between the creator thread and the new thread. This is very useful in initializing the new thread or passing any data that the thread is supposed to work on. The following program has 3 threads in addition to the main thread. The main thread passes a list to each "worker" thread, and each worker sums the list and prints the sum to the screen:

```
procedure main()
    L1 := [1, 2, 3]
    L2 := [4, 5, 6]
    L3 := [7, 7, 9]
    t1 := thread sumlist(1, L1)
    t2 := thread sumlist(2, L2)
    t3 := thread sumlist(3, L3)
    every wait(t1|t2|t3)
end

procedure sumlist(id, L)
    s := 0
    every s +:= !L
    write(" Thread id=", id, ", result=", s)
end
```

The output is

```
Thread id=2, result=15
Thread id=1, result=6
Thread id=3, result=23
```

Since the lists are independent, there is no possibility of a race condition. The example shows that the second thread was the first to finish and print its result. If the problem solution requires sharing data or guaranteeing that one thread should finish before another, then a synchronization mechanism should be used. These topics are discussed in the next two sections.

8.3 Thread Safety

Threads cooperate with each other to get the job done; they can send information back and forth, as described later in the section on thread communication. Other than such intentional communication, when programming multiple threads it is important that each thread perform its required computation without interfering with the other threads and vice versa. *Thread safety* is the property of multi-threaded code that ensures that threads do not alter each other's computations unintentionally. The opposite is any operation, including data structure traversal, where threads may affect each other's correct operation, leading to data corruption or incorrect results. Such operations are called *thread-unsafe*.

An example of thread-unsafe code can be found in the Icon Program Library wrap.icn module:

```
procedure wrap(s,i)
    local t
    static line
    initial line := ""
    /s := "" ; /i := 0
    if *(t := line || s) > i then
        return "" ~== (s :=: line)
    line := t
end
```

If two threads were trying to use wrap() at the same time, they would overwrite each others' values for the static variable line. To avoid this, one might require the caller to provide a means of storing its own line information in a third parameter. Since Unicon does not provide reference parameters, this might be passed, for example, as a record field.

```
record lineinfo(line)
procedure wrap(s,i,li)
    local t
    if /li then stop("wrap(): missing third parameter")
    /(li.line) := ""
    /s := "" ; /i := 0
    if *(t := li.line || s) > i then
        return "" ~== (s :=: li.line)
    li.line := t
end
```

This example works and is thread-safe. Its chief flaw is that the programmer who wants to call wrap() now has a more complicated API to learn, and existing code that calls the old wrap() would have to be modified to use the thread-safe version. Millions of C programmers have suffered through this for years. A more extreme solution that makes the existing API thread-safe is given below. To understand it, read the section on thread-synchronization below, particularly the subsection on thread-safe data structures. The static variable line is replaced with a table protected by a mutex; this table is indexed by the current thread &current every time it is used inside of wrap(). As an exercise for the reader, compare the performance of this approach with the previous approach that used an extra parameter.

```
procedure wrap(s,i)
    static line
    local t
    initial line := mutex(table())
    /(line[&current]) := ""
    /s := "" ; /i := 0
    if *(t := line[&current] || s) > i then
        return "" ~== (s :=: line[&current])
```

```
    line[&current] := t
end
```

From such an example, one can infer some general principles for writing thread-safe code. Each thread's data must be completely independent of all other threads. Each thread gets its own stack, so local variables and parameters are thread-safe for free. Global variables, including statics, are totally the bane of multi-threaded programming, so to achieve thread-safety you may have to avoid or rewrite any procedures or class libraries that use globals or statics. For example, if part of a procedure's results were assigned to a global for the caller to use, you might need to rewrite it to return (or generate) such results instead of working through a global.

A more subtle issue arises when referencing mutable structures such as lists, tables, or objects. Although each thread gets its own heap for allocation purposes, if another thread has a reference to such a structure, operations that alter the structure made by either thread, are unsafe. For some computations, you might avoid such a problem by making a separate copy of the structure for each thread to use independently, but when threads need to share a structure, the list or table or whatever constitutes part of their communication mechanism. In that case, thread-unsafe code or data structures can be made thread-safe via a synchronization mechanism to achieve correct behavior and results. Such mechanisms are presented in the next section.

8.4 Thread Synchronization

Thread synchronization can be done in many different ways. Some problems require more synchronization than others. Some may require advanced synchronization mechanisms and rely on the language support to achieve full control over the execution of threads and protect shared data. This section covers many synchronization techniques in Unicon, used primarily to avoid the problem of *race conditions* in multi-threaded code.

The non-deterministic behavior of threads

Programming with threads introduces a whole new set of concepts and challenges that non-threaded programs do not have to deal with. In most multi-threaded programs, threads need to communicate through shared data. Because threads run in a non-deterministic order, they access and update shared data in a non-deterministic fashion. Consider the following popular example where two threads, T_1 and T_2, try to increment a shared variable x whose initial value is 0.

T_1	T_2
x := x+1	x := x+1

While x:=x+1 may not look like it could cause a problem, in reality it does because it is not atomic. In many computer systems, it can be broken down into three operations: fetch the value of x, add 1 to it, and store the new value back in x. These three operations might occur at different times in different threads. Thread T_2 for example might fetch x, followed by T_1 also fetching it, but before T_2 stores back the new value of x, leaving T_1 working on the old value of x. T_1 should not be allowed to read the value of x while another thread, such as T_2, is updating it. Consider the following scenarios, starting with x:=0:

Scenario 1		Scenario 2	
T_1	T_2	T_1	T_2
fetch x (0)	fetch x (0)	fetch x (0)	
increment x (1)	increment x (1)	increment x (1)	
store x (1)	store x (1)	store x (1)	
			fetch x (1)
			increment x (2)
			store x (2)
The final value of x is 1.		The final value of x is 2.	

In scenario 1 the final value of x is 1, even though there are two increments done by the two threads. In scenario 2 however the final value is 2. This outcome is not necessarily a problem, or a bug that must be fixed. Non-deterministic execution is a part of multi-threaded programming that many programs can live with. For example, if one or more threads depend on a counter to update the screen every 100 increments or so, but this number does not need to be exactly 100, then the threads can increment the counter without worrying about races and about synchronizing access to the shared counter. If deterministic execution must be guaranteed, programmers have to take extra steps to ensure a specific order and predictable results. That is where thread synchronization comes into play.

User-defined synchronization

For some simple situations, synchronization can be achieved without relying on special primitives provided by the language. For example, if one thread is waiting for another to finish a task, a shared flag variable can be used. In the following example, the main thread might finish before the child thread:

```
procedure main()
    thread write("I am a thread: Hello world!")
end
```

As seen in the previous section, this can be handled using wait() or delay(). The wait() function is the best solution for this situation. delay() also works but there are two problems associated with it: it forces the program to wait a lot longer than necessary, and second,

if the delay time is not long enough, depending on the system, the main thread might still finish before the child thread. Actually even with a long delay, there is no guarantee the child thread will finish first. In a real application, delay() would be a poor choice. Finally, here is an alternative solution that does not use wait():

```
global done
procedure main()
  thread (write("I am a thread: Hello world!") & done := "true")
  until \done
end
```

In this case, the loop until \done ensures that the main thread keeps spinning until the child thread set the variable done to a non-null value. It avoids the problems with using delay(), at the expense of fully occupying one CPU in a spin-lock. Note that declaring done to be global is key. If done were local, the main thread would spin indefinitely because any change to done in the child thread would be invisible in the main thread.

If none of these approaches seems acceptable, that is a good sign. Use techniques from the following sections to avoid such inefficient synchronization.

Language support for synchronization

Using function wait() or global variables to synchronize threads might be sufficient in some situations, but most problems require the more efficient synchronization made possible by mutexes and condition variables.

Critical regions and mutexes A *mutex* (from mutual exclusion) is a synchronization construct used to protect shared data and serialize threads in *critical regions*, sequences of instructions in which only one thread may execute at a time or an error will occur. In the example discussed at the beginning of this chapter, two threads compete to increment the variable x. The end result might not be what the programmer intended. In such cases a mutex may be used to protect access to the variable x.

A mutex object is created using the mutex() function. The returned object can be locked/unlocked via the functions lock() and unlock() to serialize execution in a critical region. The following example demonstrates the use of a mutex to protect increments to the global variable x:

```
global x
procedure main()
    mtx_x := mutex()
    x := 0
    t1 := thread inc_x(mtx_x)
    t2 := thread inc_x(mtx_x)
```

```
    every wait(t1 | t2)
    write("x=", x)
end

procedure inc_x(region)
    lock(region)
    x := x + 1
    unlock(region)
end
```

It is important to note that the mutex object has to be initialized only once and can then be shared between all threads (here t1 and t2) accessing the critical region (x := x + 1). lock(region) marks the beginning of the critical region protected by the mutex, and unlock(region) marks its end. When a thread calls lock(region), it tries to acquire the mutex region. If region is not "owned" by any other thread, lock(region) succeeds, the thread becomes the owner of the mutex region, and then enters the critical region. Otherwise the thread blocks until the current owner of the mutex leaves the critical region by calling unlock(region). Since there are two threads and x := x+1 is protected by a mutex, the output of the program is guaranteed to be x=2, unlike the case where a mutex is not used, and where x=1 or x=2 are possible outputs.

The more critical-regions/mutexes a concurrent program has, the slower it runs. The length of the critical region also affects the performance. The longer the critical region, the more time it takes a thread to traverse it and release the mutex, which increases the probability that other threads become blocked waiting to acquire the mutex and enter the critical region. Locking a mutex and forgetting to unlock it is very likely to lead to a deadlock, a common problem in concurrent programming, where all threads block waiting for each other, and for resources to become available. Because all threads are blocked, resources will not be freed, and the block persists indefinitely.

Unicon provides a special syntax for critical regions equivalent to a lock()/unlock() pair, that aims mainly to guarantee that a mutex is released at the end of a critical region, besides enhancing the readability of the program. Here is the syntax:

```
critical mtx: expr
```

This is equivalent to:

```
lock(mtx)
expr
unlock(mtx)
```

Given a global variable named region that has been initialized as a mutex, the code to increment x in the previous example can be written as:

```
critical region: x := x + 1
```

The critical region syntax only unlocks the mutex if it executes to the end. If there is a return or break in the region's body, it is the programmer's responsibility to explicitly unlock the mutex. For example:

```
critical region: {
    if x > 100 then { unlock(region); return }
    x := x + 1
    }
```

In some situations, a thread might have several tasks to finish and may not want to block waiting for a mutex that is locked by another thread. For example, if a thread is creating items that can be inserted in one of several shared queues, the thread can insert every new item in the first queue that it acquires. trylock() is an alternative *non-blocking* function for locking.n If the thread cannot acquire the mutex immediately, the function fails. The most suitable way to use trylock() is to combine it with an if statement, where the then body unlocks the mutex after finishing the work on the protected object, as follows:

```
if trylock(mtx) then {
    expr
    unlock(mtx)
    }
```

Both lock() and trylock() return a reference to the mutex or the object they acquired (upon succeeding in case of trylock()). This makes it very convenient to write code like the following, assuming L1 and L2 are lists that are both marked as shared:

```
item := newitem()
if L := trylock(L1 | L2) then {
    put(L, item)
    unlock(L)
    }
```

Note that trylock() may fail to lock any of the lists, leaving item unprocessed. Depending on what the code needs to do, if it is required to guarantee that it does not proceed before one of the locks to L1 or L2 succeeds, then it can be written as follows:

```
item := newitem()
until L := trylock(L1 | L2)
put(L, item)
unlock(L)
```

Initial clause A procedure in a Unicon program can have an initialization clause at its top. This gets executed only once the first time the procedure is entered. The initial clause provides a very convenient way to place local static variables and their initialization in the same procedure, instead of relying on global variables and having to initialize them somewhere else. A procedure that produces a sequence of numbers one at each call can be written as:

```
procedure seq()
  static i
  initial i := 0
  i := i + 1
  return i
end
```

Initial clauses are thread-safe. They can be thought of as a built-in critical region that is run only once. No thread is allowed to enter the procedure if there is a thread still executing in the initial block. This can be useful in a concurrent environment to do critical initialization, such as creating a new mutex object instead of declaring a mutex variable to be global and initializing it somewhere else, or passing it from one function to another where it will be actually used. A concurrent version of seq() would look like this:

```
procedure seq()
  local n
  static i, region
  initial { i := 0; region := mutex() }
  critical region: n := i := i+1
  return n
end
```

With the use of the initial clause, seq() is self-contained and thread-safe. Note the use of the local variable n to temporarily hold the value of the counter i while still in the critical region. That is because once the thread leaves the critical region, there is no guarantee that the value of i would remain the same before it is returned. Using the variable n guarantees that the value returned is correct, even if the value of i is changed by another thread.

Thread-safe data structures In Unicon, mutexes are not just independent objects as described above, they are also attributes of other objects, namely attributes of the mutable data types. Any data structure in Unicon that can be used in a thread-unsafe manner can be protected by turning on its mutex attribute. Instead of declaring a separate mutex and locking and unlocking it, the structure can just be marked as "needs a mutex/protection" and the language does an implicit locking/unlocking, protecting the operations that might affect the integrity of the structure. For example, if several threads are pushing and popping

elements into and out of a list, these are thread-unsafe operations that require protection. The value of implicit mutexes is made clear after considering the alternative. The following producer-consumer example uses a list to send and receive data, and protects it using an explicit mutex:

```
procedure main()
  L := [ ]
  mtx := mutex()
  p := thread produce(L, mtx)
  c := thread consume(L, mtx)
  every wait(p | c)
end

procedure produce(L, region)
  every i := !10 do
    critical region: put(L, i)
end
procedure consume(L, region)
    i := 0
    while i < 10  do
        critical region: if x := get(L) then i +:= 1 & write(x)
end
```

Using a thread-safe list results in fewer lines of code, and in a more efficient program doing less locking and unlocking at the language level, or even not doing explicit locking at all. For example, the above program may be rewritten as:

```
procedure main()
  L := mutex([ ])
  p := thread produce(L)
  c := thread consume(L)
  every wait(p | c)
end

procedure produce(L)
  every put(L, !10)
end
procedure consume(L)
  i := 0
  while i < 10  do
    if x := get(L) then i +:= 1 & write(x)
end
```

The produce() and consume() procedures do not do any locking, making concurrent programming in such a case just as easy as writing a sequential program. It is only necessary

to notify the language at the beginning that the data structure is shared, by passing it to the mutex() function. This function takes a second optional parameter denoting an existing mutex object or an object that is already marked as shared (has a mutex attribute). Instead of creating a new mutex object for the data structure, the existing mutex is then used as an attribute for the data structure. If the second object is a structure that is not marked as shared, a new mutex is created. This is useful when two objects need to be protected by the same mutex. For example, the list L and the table T in the following example share the same mutex:

```
mtx := mutex()
L := mutex([ ], mtx)
T := mutex(table(), mtx)
```

which is equivalent to the following if the mutex does not need to be explicit:

```
L := mutex([ ])
T := mutex(table(0), L)
```

or

```
L := [ ]
T := mutex(table(0), L)
```

In all cases, lock(L) and lock(T) lock the same mutex, serializing execution on both data structures. Not all operations on data structures produce correct results, only "atomic" operations do. In other words, implicit locking/unlocking takes place per operation, which means that even if each of the two operations is safe, the combination might not be. A critical region is still needed to combine the two. For example, if L[1] has the value 3 and two threads are trying to increment L[1]:

```
L[1] := L[1] + 1
```

the resulting L[1] could be 4 or 5. That is because reading L[1] (the right side of the assignment) and storing the result back in L[1] are two different non-atomic operations, separated in time. The good news is that solving such an issue does not require an extra explicit mutex. If L is marked as shared (passed to the mutex() function) it can be passed to lock()/unlock() functions. It can be used with the critical syntax like this:

```
critical L: L[1] := L[1] + 1
```

Thread safe assignment without a mutex Although protecting a global variable that is written to concurrently by several threads is always the correct thing to do, there are some situations where a protecting mutex may be safely discarded. If the type of the global variable never changes (because every thread writes a value of the same type) then concurrent assignments *are* thread safe with one exception: the integers. The reason for the exception is that the underlying implementation actually uses two different types to represent integers, one for large integers that are greater than some implementation defined constant, and one for "normal" integers. If you can *guarantee* that all of the integers written to the global variable are all either small or all large (but not a mixture) then you may also discard the protecting mutex in this case too.

Doing without a mutex should be considered carefully on a case by case basis. In most cases the overhead introduced by the mutex will have an insignificant effect on the program's performance and it is better to be safe than sorry. In the rare cases where the mutex has a considerable impact on performance, following the guidelines above should give a worthwhile improvement.

If values of different types are written concurrently to a global variable then a mutex *must* be used to avoid the risk of the descriptor that the implementation uses to manage the variable having one type whilst referring to a value of a different type. Either corrupt data — if you are lucky — or program termination is the likely outcome of such an error.

Condition variables Mutexes are used to protect shared data in critical regions, and block threads if there is more than one thread trying to enter the region. *Condition* variables take thread blocking/resumption to a new level that is not tied to accessing shared data like a mutex. A condition variable allows a thread to block until an event happens or a condition is satisfied. For example, the previous section showed a producer/consumer problem where the consumer keeps spinning to get values out of the shared list. In real-life applications, any spinning could be a waste of resources; other threads, including producer threads could be using the resources to do something useful instead. The consumer needs to block until there is data to process in the list. This is where a condition variable comes into play. A condition variable is created using the function condvar(). The returned object is a condition variable that can be used with wait() and signal() functions. wait(cv) blocks the current thread on the condition variable cv. The thread remains blocked until another thread does a signal(cv), which wakes up one thread blocked on cv. A very important aspect of using a condition variable is that the variable must always be associated with a mutex. More specifically, the wait() function has to be always protected by a mutex. Unicon provides a built-in mutex for condition variables which can be thought of as an attribute similar to thread-safe data structures. This means that a condition variable can also be used with lock()/unlock() functions or the critical clause. It is important to realize that not only wait() has to be protected by a critical region, but also the condition or the test that leads a thread to wait on a condition variable. See the following example:

```
if x=0 then wait(cv)
```

A thread wants to wait on cv if x=0, but what happens if the value of x has changed between the test and the call to wait(cv)? If a second thread changes the value of x and signals cv to wake up the first thread, while the first thread transitions from the test to wait(), it may miss the wake up signal and might block indefinitely because it is waiting on a condition variable that it should not wait on. The correct way to use wait() with a condition variable is

```
lock(cv)
  if x=0 then wait(cv)
unlock(cv)
```

or :

```
critical cv: if x=0 then wait(cv)
```

Because other threads might need to access the condition variable while some threads are waiting on it, the wait() function atomically blocks the thread and releases its corresponding mutex. After receiving a wake-up signal, the blocked thread wakes up, acquires the mutex (blocking if necessary) and continues executing, and that is when wait() returns. It is good practice to do the condition variable test before assuming that it is in one state or another. This leads to a more correct way to use condition variables that ensures that a thread does not leave wait() before guaranteeing the test is in a specific state, as follows:

```
critical cv: while x=0 do wait(cv)
```

Using a while in place of if will ensure that the thread goes back to sleep if it happens to wake up and the condition has not changed.

The producer/consumer example mentioned above can be rewritten using a condition variable. Since the consumer needs to sleep/wake up depending on the availability of elements in the list, the state of the list must be guaranteed to remain the same while interacting with the condition variable for the reason explained above (missing wake-up signals). The list and the condition variable have to be protected by the same mutex. condvar() allows an optional argument, an existing mutex that is to be associated with the condition variable. In the original example, using an independent mutex to protect the condition variable looks like this:

```
procedure main()
  L := [ ]
  mtx := mutex()
  cv := condvar(mtx)
  p := thread produce(L, cv)
```

```
   c := thread consume(L, cv)
   every wait(p | c)
end

procedure produce(L, cv)
   every i := !10 do {
      critical cv: put(L, i)
      if *L=1 then signal(cv)
      }
end

procedure consume(L, cv)
   i := 0
   while i < 10  do {
      if *L=0 then critical cv: until *L>0 do wait(cv)
      if x := get(L) then i +:= 1 & write(x)
      }
end
```

Another way to write this program is on top of the thread-safe list example. Since there is no explicit mutex to pass to the condvar() function in the original example, a mutex can be first created and then passed along with the list to the mutex() function. The same mutex then can be passed to condvar(). The function binds the mutex already associated with the list to the condition variable. The final result is the same as in the explicit mutex example, a list and a condition variable sharing the same mutex. Here is the example again with the shared list and a condition variable:

```
procedure main()
   mtx := mutex()
   L := mutex([], mtx)
   cv := condvar(mtx)
   p := thread produce(L, cv)
   c := thread consume(L, cv)
   every wait(p | c)
end

procedure produce(L, cv)
   every put(L, !10) & *L=1 & signal(cv)
end

procedure consume(L, cv)
    i := 0
    while i < 10  do
        if x := get(L) then
```

```
            i +:= 1 & write(x)
        else
            critical cv: until *L>0 do wait(cv)
end
```

In previous examples, calls to signal() are not protected by any mutex. signal() does not need protection, because it does not block the thread, and there are no worries about a deadlock. The worst thing that can happen is signaling a condition variable that does not have any thread waiting on it, which is not a problem. However protecting calls to signal() is not an issue either. Depending on the problem, doing it one way or the other might be more or less efficient. There is no need to have a thread that spends a lot of time locking/unlocking a mutex if it is not necessary, creating contention in the critical region. But a thread that keeps wasting time signaling condition variables that have no threads waiting on them is also undesirable.

The signal() function takes a second optional parameter: the number of threads to be woken up. The default is one, but it can be any positive value. For example:

```
every !4 do signal(cv)
```

can be written as:

```
signal(cv, 4)
```

Furthermore, if all of the threads waiting on cv need to be woken up, a special 0 (or CV_BROADCAST) value can be passed to signal(), causing it to broadcast a wakeup call for all threads waiting on cv:

```
signal(cv, 0)
```

or

```
signal(cv, CV_ BROADCAST)
```

8.5 Thread Communication

Traditionally, co-expressions communicate implicitly, or explicitly using the @ operator. All co-expression communication is synchronous; the calling co-expression is blocked and the called co-expression runs. This simple communication model is called *activation* in Unicon. A co-expression C1 can activate another co-expression C2 using the syntax x@C2, where x is an optional value to be transmitted from C1 to C2. C1 waits until it gets activated by C2 or any other co-expression directly or indirectly activated by C2. As mentioned earlier, implicit activation takes place whenever a co-expression produces a value or falls off its

end. With implicit activation, the co-expression activates its parent (the last co-expression to activate it).

Threads take co-expression communication to a new level with their dynamic nature. Threads run concurrently; in many cases, a running thread just wants to send a value to another thread without waiting for a reply, or receive a value from another thread, if there is one, without waiting. The @ operator is not suitable for this kind of (asynchronous) communication. Unicon adds four operators dedicated to asynchronous communication. These are @>, @>>, <@ and <<@. The operators correspond to send, blocking send, receive, and blocking receive.

Thread messaging queues

Before exploring how these communication operators are used, look at messaging queues and how they are utilized to support communication between threads. Each thread maintains two queues called the *inbox* and *outbox* that are created with the thread. When a thread sends a message with an explicit destination, the message is queued in the destination's inbox. Otherwise, it is queued into the sender's outbox. A thread can receive messages from another thread by dequeuing messages from the source's outbox if there is an explicit source, otherwise it dequeues messages from its own inbox. Figure 8-1 presents two threads with inboxes and outboxes.

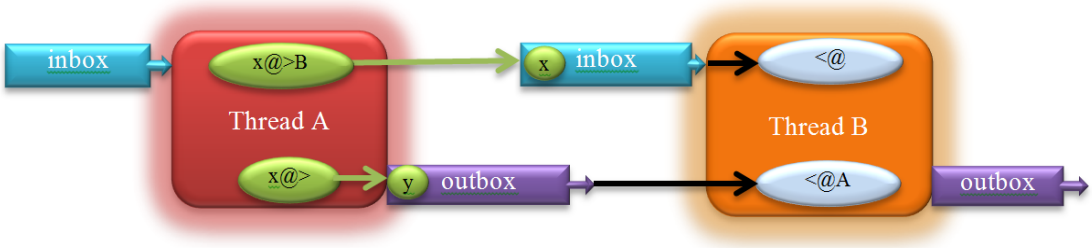


Figure 8-1: Inboxes and Outboxes for Thread Communication.

send and receive operators

The @> (send) and <@ (receive) operators communicate messages containing arbitrary data between threads. The operators support co-expressions as well, with the same semantics. The send operator has the syntax

```
x @>T
```

where x can be any data type, including null, which is equivalent to omitting it. T refers to a thread to which x is transmitted: x is placed in T's inbox. x can be picked by T using the receive operator which is presented later. x @> &main may be used to send a message to the main thread. The send operator can also have no destination, as in

```
x @>
```

In this case x is sent to no one, instead it is placed in the sender's outbox. The operator can be read as "produce x". x can then be picked up later by any thread wanting a value from this sender. For example the sender in this case might be a creating prime numbers and placing them in its outbox to be ready for other threads.

The receive operator is symmetric to send, and takes two forms, with explicit source or with no source, as follows:

```
<@T
<@
```

The first case reads "receive a value from T"; it obtains a value from T's outbox. In the prime number example mentioned above, <@T would be the way to get a prime number produced by T. <@ on the other hand reads values directly from the receiver's inbox. It reads messages sent explicitly to the thread doing the receive operation.

Both @> and <@ can succeed or fail. In the case of <@ the operator succeeds and returns a value from the corresponding queue (inbox/outbox) depending on the operand if the queue is not empty. If the queue is empty the operation fails directly. In the case of @>, if the value is placed in the corresponding queue the operation succeeds and returns the size of the queue. If the queue is full, the send operation fails. The inbox/outbox for each thread is initialized to have a limited size (it can hold up to 1024 values by default). This limit can be increased or decreased depending on the application needs. The limits are useful so that queue sizes do not explode quickly by default. They also provide an implicit communication/synchronization as explained later in following sections. Let us look at the producer/consumer example again written using the new operators:

```
procedure main()
  p := thread produce()
  c := thread consume(p)
  every wait(p | c)
end

procedure produce()
  every !10 @>   # place values in my outbox
end

procedure consume(p)
  i := 0
  while i < 10 do
    if x := <@ p then  # get values from p
      i +:= 1 & write(x)
end
```

Each thread has exactly one inbox and one outbox, and each operator call is mapped to only one of these inboxes or outboxes as seen in Figure 1. All messages from all threads coming to thread B in the figure end up in its inbox. All threads trying to receive messages from A compete on A's outbox. Both the inbox and the outbox queues are public communications channels, and it is impossible to distinguish the source of a message if there are several threads sending messages to the same thread at the same time. Furthermore, if <@ has an explicit source like A in Figure 1, it only looks in A's outbox, and does not see messages from A coming directly to the inbox. Applications that require the sender's address can attach that information to messages by building them as records with two fields, one field for data and the other containing the sender's address. A better approach for private communications for some applications is the use of lists shared between the two communicating threads or the use of private communication channels discussed later in this document.

Inbox/Outbox and the Attrib() function

As seen in previous sections, communication between threads is done though inbox/outbox queues which have size limits. The size limit, which defaults to 1024, and the actual size dictate how synchronization is merged with the communication. The size operator * can be used with a thread to query its actual outbox size (how many values it contains, not the maximum limit), as follows:

```
outbox_size := *T
```

But this is only a single attribute for one queue. To access or change other queue attributes, a new function Attrib() is introduced. This function uses the form Attrib(handle, attribcode, value, attribcode, value, ...). The integer codes used by this function are defined in an include file threadh.icn. This header file is part of the threads package, which can be used by a program via

```
import threads
```

When values are omitted, Attrib() generally returns attribute values. To get the size of the outbox (the same as the * operator), the code is

```
outbox_size := Attrib(T, OUTBOX_SIZE)
```

similarly,

```
inbox_size := Attrib(T, INBOX_SIZE)
```

gets the current size of the inbox. On the other hand

```
Attrib(T, INBOX_LIMIT, 64, OUTBOX_LIMIT, 32)
```

sets the inbox and outbox size limits to 64 and 32 respectively. The following table summarizes the available attributes and their meanings.

Attribute	Meaning	Read/Write?
INBOX_SIZE	Number of items in the inbox	Read Only
OUTBOX_SIZE	Number of items in the outbox	Read Only
INBOX_LIMIT	The maximum number of items allowed in the inbox	Read/Write
OUTBOX_LIMIT	The maximum number of items allowed in the outbox	Read/Write

Blocking send and receive

In many situations senders and receivers generate and consume messages at different speeds or based on needs. Instead of overloading slow receivers or busy waiting for slow senders, the two ends of the communication need a synchronizing mechanism to tell them when to send new messages or when a new message is available. Two more send and receive operators provide such functionality, the blocking send operator @>> and the blocking receive operator <<@. These can be used in the same way as @> and <@, except that instead of failing when the operation cannot be completed, the new operators block and wait until the operation succeeds. In the simple producer/consumer example, the producer is only producing 10 values, since the default size of the queue is 1024, using a blocking send would not make any difference. The consumer however, can make use of the blocking receive instead of spinning in some cases while the queue is empty and the original blocking receive just keeps failing. Take a closer look at the consumer code again:

```
procedure consume(p)
  i := 0
  while i < 10  do
    if x := <@ p then  # get values from p
      i +:= 1 & write(x)
end
```

Using an if statement with <@ checks whether the operation succeeds in receiving a value. A blocking receive is more suitable in this case and it simplifies the loop slightly, since the if can be dropped, and also the counter is not necessary anymore. The counter was previously necessary because the loop needs to keep track of how many <@ were needed to count to 10. The consumer can be rewritten as

```
procedure consume(p)
 # get exactly 10 values from p, block if necessary
  every !10 do write(<<@ p)
end
```

In some cases, a thread might want to use a blocking receive to get values from a second thread, but it is not willing to block indefinitely; it may do some other useful work instead of waiting. The <<@ operator accepts a timeout parameter to impose a limit on how long to wait for a result before giving up. Here is how <<@ would look in this case:

```
result := timeout <<@     #  get from my inbox
```

or

```
result := timeout <<@ T    # get from T's outbox
```

The timeout operand is a non-negative integer denoting the maximum time to wait in milliseconds. Negative integers are treated as a null value, defaulting to an indefinite blocking receive. A 0 operand indicates no waiting time, effectively resulting in a non-blocking receive. The following table summarizes the different forms of the send and receive operators and their operands:

Operator	Operands	Behavior
@> (send)	msg@>	Place msg in my outbox, fail if the outbox is full
	msg@>T	Place msg in T's inbox, fail if T's inbox is full
<@ (receive)	<@	get a message from my inbox, fail if the inbox is empty
	<@T	get a message from T's outbox, fail if T's outbox is empty
@>> (blocking send)	msg@>>	Place msg in my outbox, block if the outbox is full
	msg@>>T	Place msg in my T's inbox, block if the T's inbox is full
<<@ (blocking receive)	<<@	Get a message from my inbox, block if the inbox is empty
	<<@T	Get a message from T's outbox, block if it is empty
	n<<@	Get a message from my inbox, block up to n milliseconds waiting for an inbox message to become available
	n<<@T	Get a message from T's outbox, block up to n milliseconds waiting for a message to become available there

Most applications use only a few of these modes. In a fast sender/slow receiver application, the sender would block when the queue is full and unblock when the queue is empty (using @>>). The receiver would consume messages from the queue until it is empty, and then block until there is a new message added to the queue (<<@). For some applications

however this communication scheme might not be optimal, hence many options are provided. The different options in the table above give the programmer a wide range of control over when to block or resume a thread based on the availability of data in the communication queues. This control covers the needs of many applications and provides simple ways to abstract concurrent programming activities such as load balancing and efficient use of resources.

Private communication channels

As mentioned in the previous sections, inbox and outbox communication queues are visible by all threads all the time. In some scenarios two or more threads need to communicate with each other without worrying about other threads sending and receiving messages at the same shared queues. While it is possible to build a protocol at the application level on top of the inbox and outbox queues to achieve such behavior, it is simpler and more efficient to have the threads communicate privately. This kind of communication can be done by sharing a list between two threads and protecting it by an explicit mutex, or using a thread-safe list. A more formal way for such communication is to use the channel() function.

Starting a private communication is similar to a network connection, except that this connection is taking place between two threads in the same process instead of two different processes that may be on different machines. A private communication channel between two threads can be created using the library procedure channel().

channel() is part of the threads package, so import threads is necessary to use it. It takes one parameter, which is the thread with which the connection will be initiated. If channel() succeeds, it returns a list representing a communication channel between the two threads. Representing a bidirectional channel that can be used by the two threads, given that each thread calls the function channel() with the other thread as an argument. Here is an example.

In thread A:

```
chB := channel(B)  | "failed to open a channel with B"
```

In thread B:

```
chA := channel(A)  | "failed to open a channel with A"
```

A channel is a *directional* communication medium. One thread should use it as an outbox, and the other should use it as an inbox; only one thread will send messages over the channel while the other receives them from the other end. The provided channel can be used with the communication operators (all four of them) with the same semantics as before. The only difference in this case is that the right operand is a communication channel instead of a thread. In the channel example below, the main thread transmits the consumer's identity to the producer (c @> p), who receives it via c := <<@:

```
import threads
procedure main()
  p := thread produce()
  c := thread consume(p)
  c @> p
  every wait(p | c)
end

procedure produce()
  c := <<@
  chC := channel(c)
  every !10 @> chC   # place values in channel c
end

procedure consume(p)
  chP := channel(p)
  every !10 do write(<<@chP)
end
```

A simple thread pool

In some cases the explicit creation of a thread for each concurrent activity is the simplest and most transparent way of writing the program, especially if the threads need access to the local variables of the procedure that created them. In other cases the work can be more expeditiously carried out by a pool of "worker" threads, which execute tasks that are handed to them. The threads package contains a simple thread pool that may be used for this purpose: it has four procedures.

MakePool(n) Create a pool of n worker threads. The default value for n is 2 + the number of processors reported in &features.There is usually not much to be gained by having many more active threads than the number of available processors (unless a significant number are idle, waiting for an event to happen).

Dispatch(proc, params, ...) Queue a task to be executed by a thread from the pool. If a thread is available the procedure will be called immediately with the supplied parameters, otherwise it will be called when a thread becomes available.

---

isIdle() Succeeds if no worker threads are active and there are no tasks in the queue.

ClosePool() Shuts down the pool after remaining tasks have finished (including those that are in the queue). ClosePool does not return until the pool has been shut down and all the threads have finished, which provides a simple way of synchronizing the concurrent activities with the controller thread (often &main).

Although waiting for everything to finish is the most usual (and safest) technique, if waiting is not required a simple way to achieve it is to write

```
thread{ClosePool()}
```

Note that calling ClosePool directly in a worker thread will lead to deadlock (because the thread will be waiting for itself to terminate). The same thing happens if you write Dispatch(ClosePool). There is some risk attached to not waiting for the pool to complete its work because if the main thread terminates the whole program finishes — regardless of the state of the thread pool.

The thread pool is minimalist by design. There are a number of extra facilities that could, perhaps, be added — cancellation of a task, place a task at the front of the queue, rather than the rear — but these are left as an exercise for the reader who needs them.

Thread-local storage

It is notable that thread–local storage can be implemented in Unicon without any special support from the runtime system. The threads package contains a TLS procedure that can be invoked on either side of an assignment operator to store and recover thread-local values. A plausible implementation of TLS would be

```
procedure TLS(var)
  local thr, tls
  static mtx, TT     # A table of tables indexed by thread id
  initial { mtx := mutex(); TT := table() }

  thr := serial()
  critical mtx: tls := if not member(TT,thr) then TT[thr] := table() else TT[thr]
  return tls[var]
end
```

In fact, the implementation of TLS is

```
procedure TLS(var)
  local thr; static TT    # A table of tables indexed by thread id
  initial TT := mutex(table())
  /TT[thr:=serial()] := table()
  return TT[thr][var]
end
```

which takes full advantage of the automatic locking and unlocking features discussed earlier (and is approximately 50% faster). Note that TLS is a misnomer: it actually implements *co-expression* local storage but, unless you are writing a multi-threading program that also uses co-expressions within each thread, the distinction is unlikely to matter.

8.6 Practical examples using threads and messages

This section starts with a discussion of an early version of a program that forms part of an indexing system for LaTeX files, which are the source for a book. The system operates in three phases:

1. An analysis phase, where the possible words to be indexed are gathered from the source files.

2. A manual review phase to select the index terms. Good indexing is an art and some judgement must be exercised when choosing what to index and how to refer to it.

3. An insertion phase where the chosen terms are located and the indexing terms inserted into the source files.

We focus on the first (analysis) phase. Here the files are read in and every "word" is put into a table that counts how many times that word occurs. Words that occur too many times (either in an individual source file, or the document as a whole) are rejected as indexing candidates because they are likely to be the common words that are of no value in an index. A simple program to analyze the files is something like the following. It uses three nested loops to read each file, split every line into words and put the results into a file table. At the end of each file, it copies eligible words from the file table into the document table. Two parameters, perFile and perDoc, govern the limits that cause a particular word to be rejected as an index candidate. perDoc is used in the reportWords procedure, which is not shown.

```
global perFile    # If a wordcount exceeds this in a single tex file it is rejected
global perDoc    # If a wordcount exceeds this in the whole document it is rejected

procedure main(args)
    local nFiles
    local f, wt, dt, word, texWord, fileName, line, count, x

    # argument and option processing omitted for clarity

    dt := table(0)
    texWord := &letters ++ "_-\'"
    count := 0

    every fileName := !args do {
        if f := open(fileName, "r") then {
            wt := table(0)
            every line := !f do { # put each word in the word table
                line ? {while tab(upto(texWord))
```

```
                word := tab(many(texWord)); wt[word] +:= 1; count +:=1 }
                }
            }

            # Add the candidate words used in this file to the master table.
            every word := key(wt) do if (x := wt[word]) <= perFile then dt[word] +:= x
            close(f)
        } else {write(&errout, "Cannot open ", fileName) }
    }
    reportWords(dt)

    return  # Success
end
```

Whilst this program works, albeit with an idiosyncratic definition of what constitutes a word, it suffers from a serious defect: it only analyzes one file at a time so a large proportion of the available processing power is unused (on the author's machine, which reports 8 cores, the figure works out at 87.5% idle). We can do *much* better than that.

In the following version, which uses the threads library package, each file is processed in parallel by a separate thread drawn from a pool of worker threads. After the analysis of each file is complete, the results are sent to a separate "accumulator" thread that aggregates the results.

```
import threads
global perFile    # If a wordcount exceeds this in a single tex file it is rejected
global perDoc    # If a wordcount exceeds this in the whole document it is rejected
global countingThread

procedure main(args)
    local nFiles

    # argument and option processing omitted for clarity

    nFiles := *args
    MakePool() # no parameter means default of 2 + no. of processors

    # Start a "counting thread" to accumulate answers from the analysis threads.
    Dispatch(accumulator)

    # Analyze each file in a separate thread. Send the results to the accumulator.
    every Dispatch(analyze, !args)

    waitFor(nFiles) # Wait for all files to be analyzed.
```

```
    # Since all the analyzers have now finished, it is safe to end the counting thread.
    # It will get all the answers that have previously been sent before receiving "end".
    "end" @>> countingThread

    ClosePool() # ClosePool returns when all threads have finished.

    return # Success
end
```

The procedures called by main (except for waitFor) are all pretty much the same as the corresponding lines of code in the preceding example. Each analysis thread sends a message to the main thread when it has finished. Since the main thread knows how many files there are to be processed, it can wait until every file has been analyzed.

```
# Wait for the specified number of messages before returning
procedure waitFor(messages :integer)
    repeat { <<@ ; if 0 >= (messages -:= 1) then return }
end
```

The analyze procedure is the same as the previous example, except that it sends a result to the accumulator thread and a "finished" message to the main thread.

```
procedure analyze(fileName :string)
    local f, line, count := 0, word, wt := table(0)
    local texWord := &letters ++ "_-\'"
    if f := open(fileName, "r") then {

        every line := !f do { # put each word in the word table
            line ? {while tab(upto(texWord)) do {
                    word := tab(many(texWord)); wt[word] +:= 1; count +:=1 }
            }
        }
        wt @>> countingThread # Send the words to the accumulator thread
        close(f)
    } else { write(&errout, "Cannot open ", fileName) }

    "end" @> &main # Tell the main thread that a file has been analyzed
    return # success
end
```

The accumulator thread gets messages from the analysis threads and from the main thread. It uses the type of the message to distinguish between them. Before starting, it writes its thread id to a global variable so other threads know where to send messages.

```
procedure accumulator()
```

```
    local msg, word, x, dt := table(0)
    # publish the thread Id so other threads can send messages.
    countingThread := &current
    repeat {
        msg := <<@
        case type(msg) of {
            "table" : # Add the candidate words in this file to the master table.
                { every word := key(msg)
                    do if (x := msg[word]) <= perFile then dt[word] +:= x }
            "string" : # Final message from the main thread.
                { reportWords(dt); return }
            default: stop("Invalid message sent to counting thread")
        }
    }
end
```

Note: this design has a race condition — it is possible that an analysis thread that has started *after* the accumulator could finish its analysis *before* the accumulator has even started. In that case the program would terminate because of an attempt to send a message to &null: it has never happened, but the behaviour is theoretically possible (and a correct concurrent program may make *no* assumptions about timing). One cure would be for the main program to wait for a message from the accumulator thread before starting the others. A less elegant solution would be to delay until the countingThread variable is not null.

Although the tables themselves may be quite large, because a table is a mutable type it is passed by reference, so the messages passed between threads are quite small. Extra work has to be done to pass messages and to coordinate the threads but the savings outweigh the extra work by a considerable margin. The graph below plots the run times of the original sequential program and the concurrent program using a different number of threads to perform the analysis.

Because of the accumulator thread, the number of threads performing the analysis is one less than the graph shows. This explains the slight "bump" at two threads: there is only one analysis thread, so we get the performance of the sequential version **plus** the overheads of message passing. With two analysis threads (three in total) the run time is halved and with four analysis threads the performance is roughly quadrupled. Adding more threads doesn't really increase the performance in this particular example (the test machine reports eight processors but it's really four dual hyper-threaded cores: the lack of speed-up after four analysis threads suggests that the dual hyper-threads don't have quite as much "grunt" as two separate cores)

There is no explicit synchronization because the analysis threads are not contending with each other — if the table that counted words in the whole document were global and each analysis thread added it's own results to the global table then contention for the table might be a performance bottleneck — instead, the analysis threads are just passing their

results in a message and getting on with their day. An accumulator, for the price of an extra thread, can often result in a worthwhile increase in performance.

This design pattern (process in parallel and send results to a single accumulator) can be used in many different circumstances and, in most cases, the reduction in contention more than makes up for the cost of the extra thread.

8.6.1 Disk space usage

The unix du utility can be used to traverse a filesystem and report on the space used. If, instead of a recursive traversal of each directory, the separate directories are analyzed in parallel and the results sent to an accumulator the result is usually faster. The process may be initiated by a procedure like the following

```
# Analyze a directory and wait until the analysis is finished
procedure analyze(path)
    local thisThread := &current
    MakePool()
    Adder := thread { Dispatch(du, (\path | "." )); GatherResults() @>> thisThread }
    write("total size = ", <<@ )
    ClosePool()
    return # success
end
```

Note that instead of using a thread from the pool as an accumulator, one is created on the fly (this is another way of avoiding the start-up race discussed in the previous example). The du procedure analyzes a directory, adding up the size for regular files, ignoring special

files and handing off (sub) directories to another thread. At the end it sends off the total for that directory (but not it's children) to the accumulator thread.

Before analyzing a directory, it also sends off a message to the accumulator announcing its intent. The reasons for this are discussed later. In the interests of clarity, some code dealing with loops in the filesystem has been omitted.

```
# Get the disk usage for a directory. Do sub-dirs in parallel with this one.
procedure du(d, parent)
    local st, fd, f, path, kb := 0

    fd := open(d) | { Report("Cannot open", d); return }
    # send a "starting analysis of d" message to the Adder
    [d, &null] @>>Adder

    while f := !fd do {
        if f == ("." | "..") then next

        if st := stat(path := d || "/" || f) then {
            case st.mode[1] of {
                "-": # Normal file - add its rounded size to the total for this directory
                    kb +:= ((st.size < st.blksize) |
                                st.blksize * ceil(st.size/(0.0 + st.blksize)))/1024

                "d": # Directory - hand it off to a worker thread to analyze in parallel
                    Dispatch(du, path, d, f)

                "l" | "s" | "b" | "c" | "p" | "|": # Ignore special files, symbolic links, pipes etc.
                next

                default: Report("Cannot handle mode "", st.mode, "", file ", path)
            }
        } else {
            Report("Cannot stat", path)
        }
    }

    close(fd)
    [d, kb,] @>> Adder # Send the result from analysis of d to the Adder
end
```

The Adder thread, which calls procedure GatherResults, gets results for each directory until it's all over. Each directory results in two messages: [d, null] followed, a little later by [d, size]. Results for child directories of d might come *before* the second message, but will never precede the first.

```
procedure GatherResults()
    local msg, kb := 0

    repeat {
        msg := <<@; kb +:= \msg[2]

        # Have we finished? The question is trickier than it looks!
        if IsIdle() & (Attrib(Adder,INBOX_SIZE) = 0) then return kb
    }
end
```

Now to the discussion of "Have we finished?" The reason the question is tricky is because GatherResults operates in parallel with the analysis; perhaps before it has even started. We must ensure that we don't bail out before at least one directory has been analyzed, which is achieved by placing the test after the reception of the first message — this is one of the reasons for sending two messages per directory. We must check the analysis is finished i.e. there is no work in progress. The WIP logic depends on du queuing new work *before* reporting the result of analyzing a directory. Finally, we must have processed all of the messages.

Note that IsIdle() must be true before checking the message queue is empty; otherwise, there is a race between the analysis thread and GatherResults (We might see an empty message queue, then the analyzer posts [d, size] and finishes before we call IsIdle(): The result would be that we'd ignore the final message, or messages).

It is sometimes true that deciding when a concurrent algorithm has finished — without terminating prematurely or discarding some of the final results or never terminating — is harder than writing the processing algorithm itself!

The reader may be wondering why the directory name is passed to the GatherResults procedure, which doesn't use it. The reason is that these examples are edited extracts from a larger program that builds a structure that represents the directory and its child sub-directories. It then displays a series of pie charts (one for each directory) showing where all the space has gone. It needs the directory names to label the segments of each pie chart. The other reason for passing two messages per directory is that the full program needs to set up the structure for a directory before receiving any results for its children.

When analyzing a fairly large (500GB) directory, the pie chart program — which runs on the Unicon interpreter and is based on the code examples above — outperforms the built in du program by almost an order of magnitude on an eight core processor; the built in program is presumably written in C and optimised but, crucially, it is single threaded.

8.6.2 More suggestions for parallel processing

If several files are involved, it is often quite easy to see how the processing may be done in parallel but there are other cases — some more obvious than others — where it might

prove useful:

Monte Carlo methods Any problem that calls for a large number of trials, where the result of one trial does not affect subsequent trials is amenable to being written as a parallel application.

Matrix multiplication Large matrices may be multiplied in parallel, either by a naive rewrite of the sequential ($O(n^3)$) algorithm or by dividing the matrix up into blocks (divide and conquer) and handling each block in a separate thread.

Unicon compiler Analysis and code generation is largely independent for each Unicon procedure. It might be possible to farm out larger procedures to a thread pool and thereby increase the overall performance of the compiler.

grep If the regular expression is computationally expensive, spreading out the analysis work for each line of the file to a thread pool might be faster.

The last two suggestions are speculative but demonstrate that the world can look quite different when viewed through concurrent spectacles.

8.7 Summary

True concurrency opens up major new application domains for the Unicon language. More importantly, it enables the language to utilize more than the small fraction of modern processors utilized by traditional sequential execution. For example, on a typical quad-core desktop, many applications will be able to get between 2× and 4× the performance of a sequential Unicon program with relatively minor changes. This is comparable to the speedup typically delivered by the optimizing compiler. Some applications will be able to do even better on processors with more cores.

This document presented Unicon's concurrency facilities from a programmer's perspective. The implementation and its performance are described in more detail in [Al-G12]. There are major areas for future work, including GPU- and APU support, and various forms of implicit concurrency that can be added to the language.

Chapter 9

Execution Monitoring

Unicon's execution monitoring facilities allow the user to execute a Unicon program under the observation of one or more monitoring programs, also written in Unicon. This chapter presents the monitoring architecture and a standard execution monitoring scenario, followed by details of the language features involved. This chapter is based on "Program Monitoring and Visualization" [Jeff99], which has many additional examples.

9.1 Monitor Architecture

The monitoring facilities components are summarized in Figure 9-1. Many of these components are general-purpose language features that are useful independent of execution monitoring.

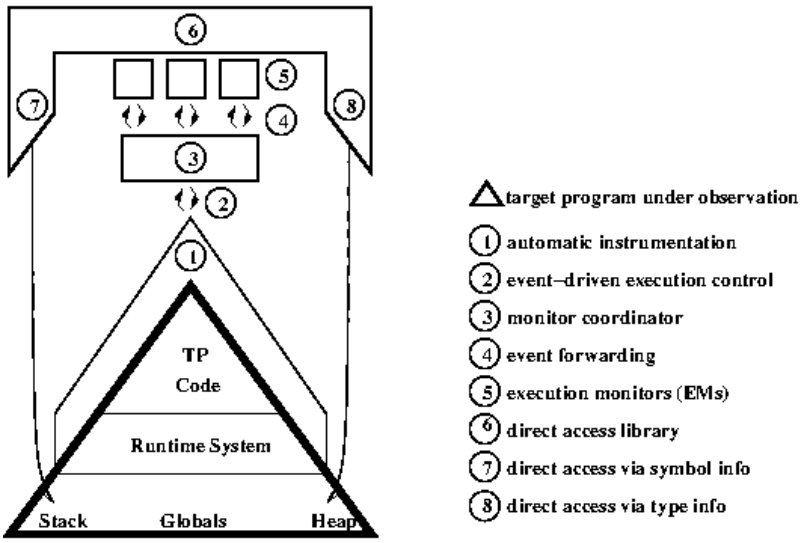


Figure 9-1: The Alamo architecture

Monitor Terminology

The terminology used in discussing Unicon's execution monitoring architecture relates to events and the linguistic features associated with them.

Dynamic loading — The ability to load multiple programs into a shared execution environment supports monitor access to target program data. *Dynamic linking* is *not* desirable in the context of execution monitoring; the names in the monitor are distinct from those in the target program.

Synchronous execution — The monitor and target program execute independently, but not concurrently. This allows the monitor to control target program execution using a simple programming model. Unicon's *co-expression* data type is used to support the relationship between monitor and target program.

High-level instrumentation — Information about program execution is available to the monitor from locations in the language runtime system that are coded to report significant events. This obviates the need for, and offers higher performance than target program instrumentation. The runtime system instrumentation is a generalization of an earlier special-purpose monitoring facility oriented around dynamic memory allocation and reclamation [Town89]. It also supercedes Icon and Unicon's procedure tracing mechanism.

Events — The primary language concept added in order to support execution monitoring is an *event*. An event is the smallest unit of execution behavior that is observable by a monitor. In practice, an event is the execution of an instrumentation point in the code (a *sensor*) that is capable of transferring control to the monitor. This definition limits events to those aspects of program behavior that are instrumented in the language runtime system or the program itself. If instrumentation does not exist for an aspect of program behavior of interest, it is often possible to monitor the desired behavior by means of other events. In the present implementation, for example, no instrumentation exists for file input and output. If an EM wishes to monitor I/O behavior, it can monitor function and operator events and act on those functions and operators that relate to input and output. A similar example involving the monitoring of Icon's built-in string scanning functions is presented in [Jeffery99]

In Unicon, events occur whether they are monitored or not, and each event may or may not be observed by any particular monitor. This definition is useful in a multi-monitor environment, in which EMs are not coupled with the instrumentation and multiple EMs can observe a TP's execution.

Event codes and values — From the monitor's perspective, an event has two components: an *event code* and an *event value*. The code is generally a one-character

string describing what type of event has taken place. For example, the event code C denotes a procedure call event. Event codes all have associated symbolic constants used in program source code. For example the mnemonic for a procedure call event is E_Pcall. These constants are available to programmers as part of a standard event monitoring library described below.

The event value is an Icon value associated with the event. The nature of an event value depends on the corresponding event code. For example, the event value for a procedure call event is an Icon value designating the procedure being called, the event value for a list creation event is the list that was created, the event value for a source location change event is the new source location, and so forth. Event values can be arbitrary Icon structures with pointer semantics; the EM accesses them just like any other source language value.

Event reports — The number of events that occurs during a program execution is extremely large—large enough to create serious performance problems in an interactive system. Most EMs function effectively on a small fraction of the available events; the events that an EM uses are said to be *reported* to the EM. An *event report* results in a transfer of control from the TP to the EM. Efficient support for the selection of appropriate events to report and the minimization of the number of event reports are primary concerns.

Event masks — A monitor controls the target program by means of this dynamic form of filtering events. An event mask is a set that describes the execution behavior of interest to the monitor. Because event codes are one-letter strings, the *cset* data type is used for event masks. Csets are represented internally by bit vectors, so a cset membership test is very efficient compared to the set data type, whose membership test is a hash table lookup.

Event masking allows the monitor to specify which events are to be reported and to change the specification at runtime. Events that are of no interest to the execution monitor are not reported and do not impose unreasonable execution cost. When a monitor starts or resumes execution of the program being monitored, the monitor selects a subset of possible event codes from which to receive its first report. The program executes until an event occurs with a selected code, at which time the event is reported. After the monitor has finished processing the report, it transfers control back to the program, again specifying an event mask. Dynamic event masking enables the monitor to change the event mask in between event reports.

When an event report transfers control from TP to EM, the two components of the event are supplied in the Icon keywords &eventcode and &eventvalue, respectively. As discussed earlier, these keywords are special global variables that are given their values by the runtime system during an event report, rather than by explicit user

assignment. The monitor then can act upon the event based on its code, display or manipulate its value, etc.

Standard Execution Monitoring Scenario

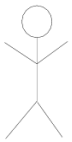
The following scenario presents the relationship between execution monitors and target program in its simplest form. More sophisticated relationships between the monitor and target program, such as running many monitors on a single target program via a monitor coordinator, are described in "Program Monitoring and Visualization" [Jeffery99]. In addition, the expected user and range of program behavior observable using these monitoring facilities are characterized.

Scenario Definitions

target program (TP) — The target program is the Unicon program under study, a translated Unicon executable file. Monitoring does not require that the TP be recompiled, nor that the TP's source code be available, although some monitors make use of program text to present information.

execution monitor (EM) — An execution monitor is a Unicon program that collects and presents information from an execution of a TP.

program behavior — Program behavior denotes the results of executing the TP. Behavior is meant in a general sense that includes program output, execution time, and the precise sequence of actions that take place during execution.



user — In our standard scenario, the user is a human capable of understanding the TP's execution behavior. The user must know the target language in order to make good use of many EMs or to write a new EM. In general, the user need not necessarily be familiar with the TP's source code.

Execution monitoring begins with a user who has questions about the behavior of a TP (Figure 9-2). Typical questions relate to correctness or performance, such as "How is the result calculated?" or "What is taking so long?". Questions may be more general in nature if the user is just trying to understand how a program works.

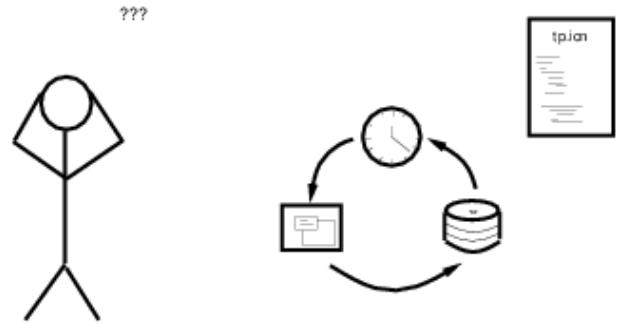


Figure 9-2: Monitoring starts with a user, a program, and questions.

Answers to important questions can be found by following the execution at the source language level, but key behavior often depends upon language semantics, implemented by the language runtime system. In Figure 9-3, iconx.c denotes the set of files in the Unicon language runtime system. Many monitors can provide useful information about runtime behavior even if the TP's source code is not available. Figure 9-3 could be elaborated to include dependencies on the platform on which the program is running.

Figure 9-3: Behavior depends on the language, not just the program.

Selecting or Developing Appropriate Monitors

Rather than focusing on one monolithic EM that attempts to accommodate all monitoring tasks, the framework advocates development of a suite of specialized EMs that observe and present particular aspects of a TP's behavior. The user is responsible for selecting an appropriate EM or set of EMs that address the user's concerns.

If no available EM can provide the needed information, the user can modify an existing EM or write a new one. This end user development of execution monitors is also useful when an existing EM provides the needed information, but it is obscured by other information; existing EMs can be customized to a particular problem.

Running the Target Program

The user runs the TP, monitored by a selection of EMs (Figure 9-4). General-purpose EMs provide an overall impression of program behavior. Visualization techniques enable the presentation of a large amount of information and abstract away detail.

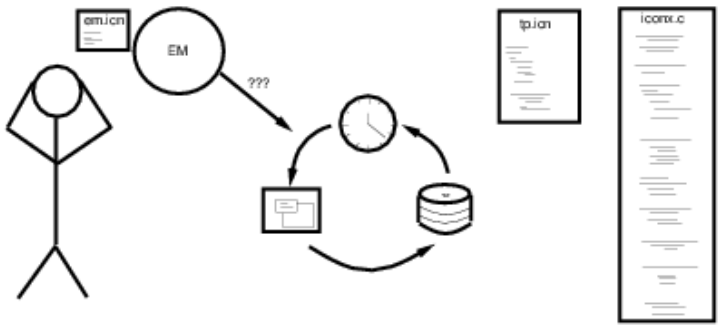


Figure 9-4: EMs can answer questions about TP behavior.

Obtaining specific information often requires that the user interact with the EMs to control the TP's execution, either to increase the amount of information presented during specific portions of execution or to pause execution to examine details. In order to provide this interactive control, EMs present execution information as it happens during the TP's execution, rather than during a postmortem analysis phase.

Framework Characteristics

The preceding scenario requires language support in several areas: controlling a program's execution, obtaining execution information, presenting large quantities of information, and interacting with the user. To support these tasks, the framework provides synchronous shared address multitasking and an event-driven execution control model. The first two of these features are the focus of this chapter.

Multitasking

In the monitoring execution model, in which an EM is a separate program from the TP , the relationship is almost that of two co-expressions, except that activations of the monitor are implicit occurrences of events within the runtime system, rather than expression results or explicit activations using the @ operator. Event reports are transfers of control to the monitor as well as the primary source of execution information from a TP (Figure 9.5).

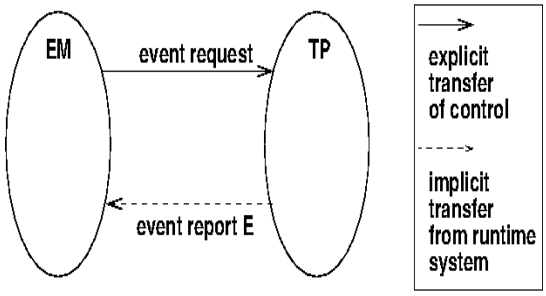


Figure 9-5: EM and TP are separately loaded coroutines

Multitasking has the following benefits for monitoring: the EM and TP are independent programs, the EM has full access to the TP, and the mechanism accommodates multiple EMs. These benefits are described in more detail below.

Independence

Because the EM and TP are separate programs, the TP need not be modified or even recompiled in order to be monitored by an EM, and EMs can be used on different target programs. By definition, execution of tasks such as EMs and TPs is synchronous. The TP is not running when an EM is running, and vice versa. This synchronous execution allows EMs and TPs to be independent without introducing the complexities of concurrent programming. In a concurrent context, each thread might have a monitor, but the target thread and its associated monitor are not concurrent — and monitoring concurrent programs is not yet supported by the implementation.

Another degree of EM and TP independence is afforded by separate memory regions; EMs and TPs allocate memory from separate heaps. Memory allocation in the EM does not affect the allocation and garbage collection patterns in the TP. Because Unicon is a type-safe language with runtime type checking and no pointer data types, EMs and TPs cannot corrupt each others' memory by accident; only code that contains explicit references to another program's variables and data can modify that program's behavior. EMs can (and some do) modify TP values in arbitrary ways; the purpose of separate memory regions is to minimize *unintentional* data intrusion.

Access

An address space is a mapping from machine addresses to computer memory. Within an address space, access to program variables and data is direct, efficient operations such as single machine instructions. Accessing program variables and data from outside the address space is slow and requires operating system assistance.

The EM and TP reside within the same address space. This allows EMs to treat TP data values in the same way as their own: EMs can access TP structures using regular Unicon operations, compare TP strings with their own, and so forth. Because of the shared address

space, the co-expression switch used to transfer execution between EMs and TPs is a fast, lightweight operation. This is important because monitoring requires an extremely large number of task switches compared to typical multitasking applications.

Multiple Monitors and Monitor Coordinators

Unicon's dynamic loading capabilities allow simultaneous execution of not just a single EM and a single TP, but potentially many EMs, TPs, and other Icon programs in arbitrary configurations. Although uses for many such configurations can be found, one configuration merits special attention when many specialized EMs are available: the execution of multiple monitors on a single TP (Figure 9-6, left).

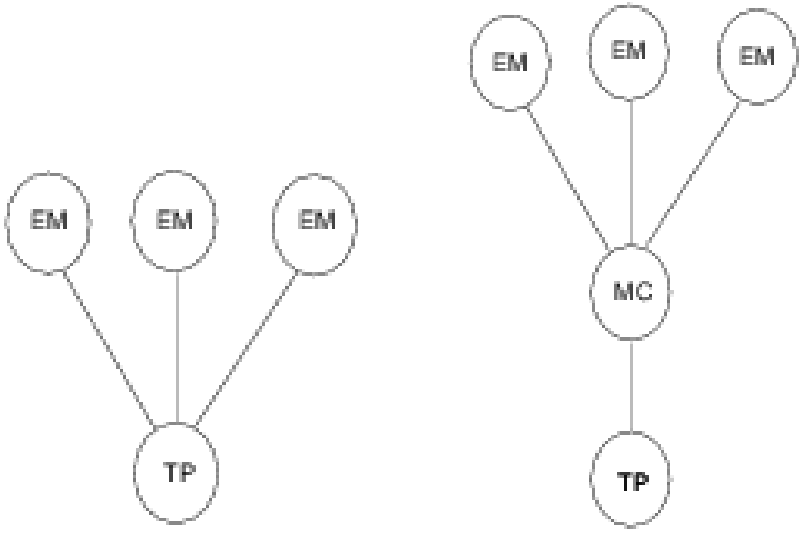


Figure 9-6: Multiple EMs (left); EMs under a monitor coordinator (right)

The difficulty posed by multiple monitors is not in loading the programs, but in coordinating and transferring control among several EMs and providing each EM with the TP execution information it requires. Since EMs are easier to write if they need not be aware of each other, things are much simpler if EMs run under a *monitor coordinator* (MC), a special EM that monitors a TP and provides monitoring services to one or more additional EMs (Figure 9-6, right). EMs receiving an MC's services need not be aware of the presence of an MC any more than a TP need be aware of the presence of an EM.

The virtual monitor interface provided by MCs makes adding a new monitor to the system extremely easy. A new monitor could conceivably be written, compiled, linked, and loaded during a pause in the TP's execution. In addition, constructing efficient MCs that provide high-level services is another area of research that is supported within the Alamo Icon framework.

Support for Dual Input Streams

An EM typically has two primary input streams: the event stream from the TP, and the input stream from the user (Figure 9-7). Although these two input streams are conceptually independent and may be treated as such, for many EMs this unnecessarily complicates the central loop that obtains event reports from TP—the EM must also check its own window for user activity.

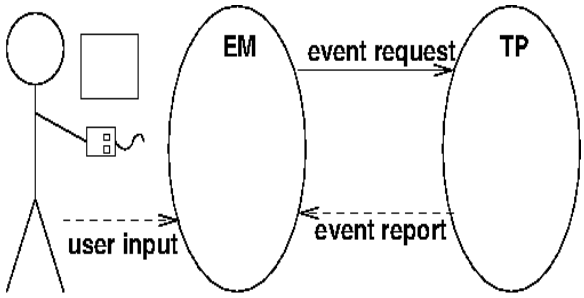


Figure 9-7: Monitors have two input streams

The runtime system instrumentation includes code that optionally checks for EM input and reports it as an event by the execution monitoring facility, instead of requiring that the EM explicitly check the user input stream. This simplifies EM control flow and improves EM performance.

9.2 Obtaining Events Using evinit

A standard library called evinit provides EMs with a means of obtaining events. Programs wishing to use the standard library include a link declaration such as link evinit. In addition, monitors include a header file named evdefs.icn to obtain the symbolic names of the event codes.

Setting Up an Event Stream

An EM first sets up a source of events; the act of monitoring then consists of a loop that requests and processes events from the TP. Execution monitoring is initialized by the procedure EvInit(x*[,input,output,error]*). If x is a string, it is used as an icode file name in a call to the Unicon function load(). If x is a list, its first argument is taken as the icode file name and the rest of the list is passed in to the loaded function as the arguments to its main procedure. EvInit() assigns the loaded TP's co-expression value to EM's &eventsource keyword. The input, output, and error arguments are files used as the loaded program's standard files.

EMs generally call the library procedure EvTerm() when they complete, passing it their main window (if they use one) as a parameter. EvTerm() informs the user that execution

has completed and allows the final screen image to be viewed at the user's leisure, waiting for the user to press a key or mouse button in the window and then closing it.

The typical EM, and all of the EMs presented as examples in this book, follow the general outline:

```
$include "evdefs.icn"
link evinit
procedure main(arguments)
    EvInit(arguments) | stop("can't initialize monitor")
    # ... initialization code, open the EM window
    # ... event processing loop (described below)
    EvTerm()
end
```

This template is generally omitted from program examples for the sake of brevity.

EvGet()

Events are requested by an EM using the function EvGet(mask). EvGet(mask) activates the co-expression value of the keyword &eventsource to obtain an event whose code is a member of the cset mask. mask defaults to &cset, the universal set indicating all events are to be reported. The TP executes until an event report takes place; the resulting code and value are assigned to the keywords &eventcode and &eventvalue. EvGet() fails when execution terminates in TP.

Event Masks and Value Masks

EvGet() allows a monitor the flexibility to change event masks each time the event source is activated. Another function that sets event masks is eventmask(). eventmask(C,c) sets the event mask of the task owning co-expression C to the cset value given in c.

Event masks are the most basic filtering mechanism in Alamo, but there are situations where they are not specific enough. For example, instead of handling events for all list operations, you may want events only for specific lists. This situation is supported by the concept of *value masks*. A value mask is an Icon set or cset whose members are used to filter events based on their &eventvalue, just as an event mask filters based on the &eventcode. You may specify a different value mask for each event code. Value masks for all event codes are supplied in a single Icon table value whose keys map event codes to corresponding value masks. This table is passed as an optional second parameter to EvGet() or third parameter to eventmask(). Note that no value mask filtering is performed for event codes that are not key in the value mask. Note also that value masks persist across calls to EvGet(). They are replaced when a new value mask is supplied, or disabled if a non-table is passed as the value mask parameter.

There is one special case of value masks that receives extra support in Icon: virtual machine instructions. Requesting an event report for the execution of the next virtual machine instruction is performed by calling EvGet() with an event mask containing E_Opcode. VM instructions occur extremely frequently; dozens of them can occur as a result of the execution of a single line of source code. Consequently, performance is severely affected by the selection of all VM instruction events.

However, a particular instruction or small set of instructions may be of interest to a monitor. In that case, the EM need not receive reports for all instructions. The function opmask(C, c) allows EM to select a subset of virtual machine instructions given by c in C's task. Subsequent calls to EvGet() in which E_Opcode is selected reports events only for the VM instructions designated by c.

The event values for E_Opcode are small non-negative integers. They fall in a limited range (< 256), which is what allows a cset representation for them. Symbolic names for individual virtual machine instructions are defined in the include file opdefs.icn. opmask(C, c) is equivalent to:

```
t := table()
t[E_Opcode] := c
eventmask(C, , t)
```

9.3 Instrumentation in the Icon Interpreter

This section describes the instrumentation used by Unicon to produce events at various points in the runtime system. Significant points in interpreter execution where transfer of control might be warranted are explicitly coded into the runtime system with tests that result in transfer of control to an EM when they succeed. When execution reaches one of these points, an event occurs. Events affect the execution time of the TP; execution is either slowed by a test and branch instruction (if the event is not of interest to the EM), or stopped while the event is reported to the EM and it processes information. Minimizing the slowdown incurred due to the presence of monitoring instrumentation has been a focus of the implementation.

There are several major classes of events that have been instrumented in the Unicon interpreter. Most of these events correspond to explicit elements within the source code; others designate actions performed implicitly by the runtime system that the programmer may be unaware of. A third class of event that has been instrumented supports user interaction with the EM rather than TP behavior.

Explicit Source-Related Events

The events that relate behavior observable from the source code are:

Program location changes — Source code locations are reported in terms of line numbers and columns.

Procedure activity — There are events for procedure calls, returns, failures, suspensions, and resumptions. In addition to these explicit forms of procedure activity, events occur for implicit removals of procedure frames.

Built-in functions and operations — Events that correspond to Icon built-ins describe many areas of behavior from numeric and string operations to structure accesses and assignments. Like procedures, events are produced for function and operator calls, returns, suspensions, resumptions, and removals.

String scanning activity — Icon's pattern matching operations include scanning environment creation, entry, change in position, and exit. To obtain a complete picture of string scanning, monitors must observe these events along with the built-in functions related to string scanning.

Implicit Runtime System Events

Events that depict important program behavior observed within the runtime system include:

Memory allocations — Memory is allocated from the string and block regions in the heap. Allocation events include size and type information. This instrumentation is based on earlier instrumentation added to Icon for a memory monitoring and visualization system [Town89].

Garbage collections — The storage region being collected (Icon has separate regions for strings and data structures), the memory layout after compaction, and the completion of garbage collection are reported by several events.

Type conversions — In Icon, automatic conversions are performed on parameters to functions and operators. Information is available for conversions attempted, failed, succeeded, and found to be unnecessary.

Virtual machine instructions — Icon's semantics may be defined by a sequence of instructions executed by the Icon virtual machine [Gris86]. The program can receive events for all virtual machine instructions, or an arbitrary subset.

Clock ticks — The passage of CPU time is indicated by a clock tick.

Most EMs, except completely passive visualizations and profiling tools, provide the user with some degree of control over the monitoring activity and must take user interaction into

account. For example, the amount of detail or the rate at which the monitor information is updated may be variables under user control. Since an EM's user input occurs only as often as the user presses keys or moves the mouse, user interaction is typically far less frequent than events in TP. Even if no user input occurs, polling for user input may impose a significant overhead on the EM because it adds code to the central event processing loop.

In order to avoid this overhead, the event monitoring instrumentation includes support for reporting user activity in the EM window as part of the TP's event stream. Monitor interaction events are requested by the event code E_MXevent. An example of the use of monitor interaction events is presented further in this chapter in the section entitled "Handling User Input". A complete list of event codes is presented in Appendix ?? in order to indicate the extent of the instrumentation.

9.4 Artificial Events

As described above, the Unicon co-expression model allows interprogram communication via explicit co-expression activation or implicit event reporting within the runtime system. *Artificial events* are events produced by explicit Icon code; they can be viewed at the language level as co-expression activations that follow the same protocol as implicit events, assigning to the keyword variables &eventcode and &eventvalue in the co-expression being activated.

There are two general categories of artificial events, *virtual events* meant to be indistinguishable from implicit events and *pseudo events* that convey control messages to an EM. Virtual events are generally used either to produce event reports from manually instrumented locations in the source program, to simulate event reports, or to pass on a real event from the primary EM that received it to one or more secondary EMs. Pseudo events, on the other hand, are used for more general inter-tool communications during the course of monitoring, independent of the TP's execution behavior.

Virtual Events Using event()

The function event(code, value, recipient) sends a virtual event report to the co-expression recipient, which defaults to the &main co-expression in the parent of the current task, the same destination to which implicit events are reported.

There are times when a primary EM wants to pass on its events to a secondary EM. An example would be an event transducer that sits in between the EM and TP, and uses its own logic to determine which events are reported to EM with more precision than is provided by the masking mechanism. A transducer might just as easily report extra events with additional information it computes, in addition to those received from TP. A more substantial application of virtual events is a monitor coordinator, an EM that coordinates and produces events for other monitors. Such a tool is presented in [Jeffery99]

Pseudo Events for Tool Communication

EMs generally have an event processing loop as their central control flow mechanism. The logical way to communicate with such a tool is to send it an event. In order to distinguish a message from a regular event report, the event code must be distinguishable. In the monitoring framework, this is achieved simply by using an event code other than a one-letter string, such as an integer. Since not all EMs handle such events, they are not delivered to an EM unless it passes a non-null second argument (the "value mask argument") to EvGet(), such as EvGet(mask, 1).

The framework defines a minimal set of standard pseudo events, which well-behaved EMs should handle correctly [Jeffery99]. Beyond this minimal set, pseudo events allow the execution monitor writer to explore communication between EMs as another facility to ease programming tasks within the monitoring framework.

9.5 Monitoring Techniques

Monitors generally follow a common outline and use a common set of facilities, which are described below.

Anatomy of an Execution Monitor

The execution monitoring interface uses a form of *event driven* programming: the central control flow of EM is a loop that executes the TP for some amount of time and then returns control to EM with information in the form of an event report. The central loop of an EM typically looks like:

```
while EvGet(eventmask) do
    case &eventcode of {
        # a case clause for each code in the event mask
        }
```

Event-driven programming is more commonly found in programs that employ a graphical user interface, where user activity dominates control flow. Because monitoring employs a programming paradigm that has been heavily studied, many coding techniques developed for graphical user interface programming, such as the use of callbacks [Clark85], are applicable to monitors. Several of the example EMs in the IPL use a callback model to take advantage of a higher-level monitoring abstraction available by means of a library procedure.

Handling User Input

An EM that handles user input could do so by polling the window system after each event in the main loop:

```
while EvGet(eventmask) do {
    case &eventcode of {
        # a case clause for each code in the event mask
        }
    # poll the window system for user input
    }
```

If the events being requested from the TP are relatively infrequent, this causes no great problem. However, the more frequent the event reports are, the more overhead is incurred by this approach relative to the execution in TP. In typical EMs, polling for user events may slow execution from an imperceptible amount to as much as 15 percent. Relative frequency for different types of events varies wildly; it is discussed in [Jeffery99].

Since the slowdown is a function of the frequency of the event reports and not just the cost of the polling operation itself, techniques such as maintaining a counter and polling once every n event reports still impose a significant overhead. In addition, such techniques reduce the responsiveness of the tool to user input and therefore reduce the user's control over execution.

Monitor interaction events, presented earlier in this chapter, address this performance issue by allowing user input to be supplied via the standard event stream produced by EvGet(). Since the E_MXevent event occurs far less frequently than other events, it makes sense to place it last in the case expression that is used to select actions based on the event code. Using this feature, the main loop becomes:

```
while EvGet() do
    case &eventcode of {
        # other cases update image to reflect the event
        E_MXevent: {
            # process user event
            }
        }
```

EvGet() reports pending user activity immediately when it is available; the control over execution it provides is comparable to polling for user input on each event.

After each event report, EMs can use Unicon's intertask data access functions to query TP for additional information, such as the values of program variables and keywords. The access functions can be used in several ways, such as

- applying a predicate to each event report to make monitoring more specific,
- *sampling* execution behavior not reported by events by polling the TP for information unrelated to the event reports [Ogle90], or
- presenting detailed information to the user, such as the contents of variables.

9.6 Some Useful Library Procedures

As mentioned, several library procedures are useful in EMs. The following library procedures that are included in the evinit library.

Location decoding and encoding procedures are useful in processing location change event values, but they are also useful in other monitors in which two-dimensional screen coordinates must be manipulated. Besides program text line and columns, the technique can variously be applied to individual pixels, to screen line and columns, or to screen grid locations in other application-specific units.

In addition, various EMs use utility procedures. Figure 9-8 lists some library procedures that are recommended for use in monitors.

procedure	returns or computes
evnames(s)	converts event codes to text descriptions and vice versa
evsyms()	two-way table mapping event codes to their names
typebind(w,c)	table mapping codes to color coded Clones for w
opnames()	table mapping VM instructions to their names
location()	encodes a two dimensional location in an integer
vertical()	y/line/row component of a location
horizontal()	x/column component of a location
prog_len()	number of lines in the source code for TP
procedure_name()	name of a procedure
WColumns()	window width in text columns
WHeight()	window height in pixels
WRows()	window height in text rows
WWidth()	window width in pixels

Figure 9-8: Additional library procedures for monitors.

9.7 Conclusions

Unicon includes facilities to exploit instrumentation available within its runtime system. Writing a monitor consists of writing an ordinary application. The key concepts introduced for Unicon's event monitoring facilities are events, event reports, event codes and values, and event masks. Monitors also make use of a standard monitoring library and the graphics facilities.

Part II

Object-oriented Software Development

Chapter 10

Objects and Classes

Object-oriented programming means different things to different people. In Unicon, object-oriented programming starts with encapsulation, inheritance, and polymorphism. These ideas are found in most object-oriented languages as well as many that are not object-oriented. This and following chapters present these ideas and illustrate their use in design diagrams and actual code. Diagrams and code are alternative notations by which programmers share their knowledge. This chapter explores the essence of object-orientation and gives you the concepts needed before you delve into diagrams and code examples. In this chapter you will learn:

- How different programming languages support objects in different ways
- To simplify programs by encapsulating data and code
- The relationship between objects and program design
- Draw diagrams that show class names, attributes, and methods
- Write corresponding code for classes and their methods
- To create instances of classes and invoke methods on those objects

10.1 Objects in Programming Languages

Object-oriented programming can be done in any language, but some languages make it easier than others. Support for objects should not entail strange syntax or programs that look funny in a heterogeneous desktop-computing environment. Smalltalk has these problems. C++ avoids these programs, but its low-level machine-orientation is less than ideal as an algorithmic notation usable by non-experts. Java offers a simple object model and familiar syntax. The advantages Unicon has over Java are fundamentally higher-level built-in types, operations, and control structures.

Many object-oriented languages require that *everything* be done in terms of objects, even when objects are not appropriate. Unicon provides objects as just another tool to aid

in the writing of programs, especially large ones. Icon already provides a powerful notation for expressing a general class of algorithms. The purpose of object-orientation is to enhance that notation, not to get in the way of it.

Icon does not support user-defined objects, although its built-in types have nice object-like encapsulation and polymorphism properties. Unicon's object-oriented facilities descend from a package for Icon called Idol. In Idol, a preprocessor implemented objects with no support from the underlying Icon runtime system. In contrast, Unicon has support for objects built-in to the language. This simplifies the notation and improves the performance of object-related computations.

Object-orientation adds several general concepts into procedure-based programming. The single overriding reason for object-oriented programming is to reduce complexity in large programs. Simple programs can be written easily in any language. Somewhere between the 1,000-line mark and the 10,000-line mark most programmers can no longer keep track of their entire program at once. By using a very high-level programming language, fewer lines of code are required; a programmer can write perhaps ten times as large a program and still be able to keep track of things.

As programmers write larger and larger programs, the benefit provided by very high-level languages does not keep up with program complexity. This obstacle has been labeled the "software crisis", and object-oriented programming is one way to address this crisis. In short, the goals of object-oriented programming are to reduce the complexity of coding required to write very large programs and to allow code to be understood independently of the context of the surrounding program. The techniques employed to achieve these goals are discussed below.

A second reason to consider object-oriented programming is that the paradigm fits certain problem domains especially well, such as simulation, and graphical user interfaces. The first well-known object-oriented language, Simula67, certainly had the domain of simulation in mind. The second pioneering object-oriented language, Smalltalk, popularized fundamental aspects of bitmapped graphical user interfaces that are nearly universal today. Three decades of experience with object-oriented techniques has led many practitioners to conclude that the concepts presented below are very general and widely applicable, but not all problems fit the object-oriented mold. Unicon advocates the use of objects, but this is a suggestion, not a rule.

Encapsulation

The primary concept advocated by object-oriented programming is the principle of encapsulation. Encapsulation is the isolation, in the source code that a programmer writes, of a data representation and the code that manipulates the data representation. In some sense, encapsulation is an assertion that no other routines in the program have *side-effects* with respect to the data structure in question. It is easier to reason about encapsulated data

because all of the source code that could affect that data is immediately present with its definition.

Encapsulation does for data structures what the procedure does for algorithms: it draws a line of demarcation in the program source code. Code outside this boundary is irrelevant to the code that is inside, and vice versa. Communication across the boundary occurs through a public interface. An encapsulated data structure is called an object. Just as a set of named variables called parameters is the interface between a procedure and the code that uses it, a set of named procedures called methods comprises the interface between an object and the code that uses it.

This textual definition of encapsulation as a property of program source code accounts for the fact that good programmers can write encapsulated data structures in any language. The problem is not capability, but verification. To verify encapsulation some languages require programmers to specify the visibility of every piece of information in each data structure as *public* or *private*. There are even multiple forms of privacy (semi-private?). Unicon instead stresses simplicity.

Inheritance

In large programs, the same or nearly the same data structures are used over and over again for myriad different purposes. Similarly, variations on the same algorithms are employed by structure after structure. To minimize redundancy, techniques are needed to support code sharing for both data structures and algorithms. Code is shared by related data structures through a programming concept called inheritance.

The basic premise of inheritance is simple: when writing code for a data structure similar to a structure that is already written, specify the new structure by giving the differences between it and the old structure, instead of copying and modifying the old structure's code. There are times when the inheritance mechanism is not useful, such as if the two data structures are more different than they are similar, or if they are simple enough that inheritance would only confuse things, for example.

Inheritance addresses multiple programming problems found at different conceptual levels. The most obvious software engineering problem it solves might be termed enhancement. During the development of a program, its data structures may require extension via new state variables or new operations or both; inheritance is especially useful when both the original structure and the extension are used by the application. Inheritance also supports simplification, or the reduction of a data structure's state variables or operations. Simplification is analogous to *argument culling,* an idea from lambda calculus (don't worry if this sounds like Greek to you), in that it describes a logical relation between structures. In general, inheritance may be used in source code to describe any sort of relational hyponymy, or special casing. In Unicon the collection of all inheritance relations defines a directed (not necessarily acyclic) graph.

Polymorphism

From the perspective of the writer of related data structures, inheritance provides a convenient method for code sharing, but what about the code that uses objects? Since objects are encapsulated, that code is not dependent upon the internals of the object at all, and it makes no difference to the client code whether the object in question belongs to the original class or the inheriting class.

We can make a stronger statement. Due to encapsulation, different executions of code that uses objects to implement an algorithm may operate on objects that are not related by inheritance at all. Such code can utilize any objects that implement the operations that the code invokes. This facility is called polymorphism, and such algorithms are called generic. This feature is found in many non-object-oriented languages; in object-oriented languages it is a natural extension of encapsulation.

10.2 Objects in Program Design

Another fundamental way to think about objects is from the point of view of software design. During program design, objects are used to model the problem domain. The different kinds of objects and relationships between objects capture fundamental information that organizes the rest of the program's design and implementation. Program design includes several other fundamental tasks such as the design of the user interface, or interactions with external systems across a network. Additional kinds of modeling are used for these tasks, but they all revolve around the object model.

For small, simple, or well-understood software projects, a prose description may be all the documentation that is needed. The Unified Modeling Language (UML) is a notation for building software models of larger software systems for which a prose description alone would be inadequate. It was invented by Grady Booch, Ivar Jacobson, and James Rumbaugh. In UML, software models document the purpose and function of a software system. The advantage of a model is that it conveys information that is both more precise and more readily understood than a prose description. UML is used during multiple phases of the software lifecycle. UML defines several kinds of diagrams, of which we will only consider four in this book.

- **Use case diagrams** show the organization of the application around the specific tasks accomplished by different users.
- **Class diagrams** show much of the static structure of the application data.
- **Statechart diagrams** model dynamic behavior of systems that respond to external events, including user input.
- **Collaboration diagrams** model interactions between multiple objects

These diagrams describe key aspects of many categories of software applications. The reader should consult the UML Notation Guide and Semantics documents for a complete description of UML. A good introduction is given in *UML Toolkit*, by Hans-Erik Eriksson and Magnus Penker (1998).

A typical application of UML to a software development project uses these diagrams in sequence. You start by constructing use case diagrams and detailed descriptions of the different kinds of users and tasks performed using the system. Then develop class diagrams that capture the relationships between different kinds of objects in the system. Finally, construct statechart and collaboration diagrams as needed to describe the sequences of events that can occur and the corresponding operations performed by various objects in response to those events.

Use case and statechart diagrams are important, but their purpose is to elaborate on an object model described in class diagrams. For this reason, class diagrams are presented first, along with the corresponding programming concepts. Use case diagrams, statecharts, and collaboration diagrams are discussed in Chapter 12.

10.3 Classes and Class Diagrams

Classes are user-defined data types that model the information and behavior of elements in the application domain. In Unicon they are records with associated procedures, called methods. Instances of these special record types are called objects. But the language constructs originated from a need to model application domain concepts, so it is appropriate to introduce them from that perspective.

Modeling a software system begins with identifying things that are in the system and specifying how they are related. A class diagram shows a static view of relationships between the kinds of elements that occur in the problem domain. A class diagram is a data-centric, object-centric model. In contrast, a user-centric view is provided by use cases. Class diagrams have several basic components.

Classes are represented by rectangles. A *class* denotes a concept of the application domain that has state information (depicted by named *attributes*) and/or behavior (depicted by named operations, or *methods*) significant enough to be reflected in the model. Inside the rectangle, lines separate the class name and areas for attributes and operations. Figure 10-1 shows an example class.

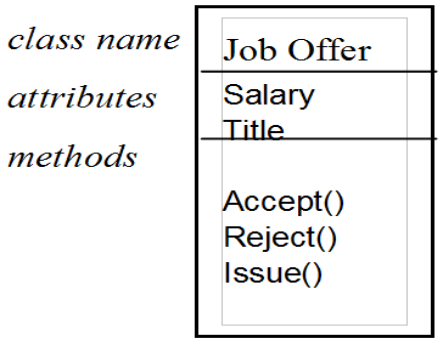


Figure 10-1: Class Job Offer has two Attributes and Three Methods

Classes in an object model are implemented using programming language classes, which are described in the next section. The degree of separation between the notion of a class in the model and in the implementation depends on the programming language. In the case of Unicon, the separation is minimal, because built-in types such as lists and tables take care of almost all data structures other than those introduced specifically to model application domain elements. In the case of C++ or Java, many additional implementation artifacts typically have to be represented by classes.

The same class can appear in many class diagrams to capture all of its relationships with other classes. Different diagrams may show different levels of detail, or different aspects (projections) of the class relevant to the portion of the model that they depict. In the course of modeling it is normal to start with few details and add them gradually through several iterations of the development process. Several kinds of details may be added within a class. Such details include:

- The *visibility* of attributes and operations. A plus sign (+) before the attribute name indicates that the attribute is public and may be referenced in code external to the class. A minus sign (-) before the attribute name indicates that the attribute is private and may not be referenced in code outside the class.

- Types, initial values, and properties of attributes

- Static properties of the class that will not be relevant at run-time

Attribute names may be suffixed with a colon and a type, an equal sign and a value, and a set of properties in curly braces. Figure 10-2 shows two very different levels of detail for the same class. Each level of detail is appropriate in different contexts.

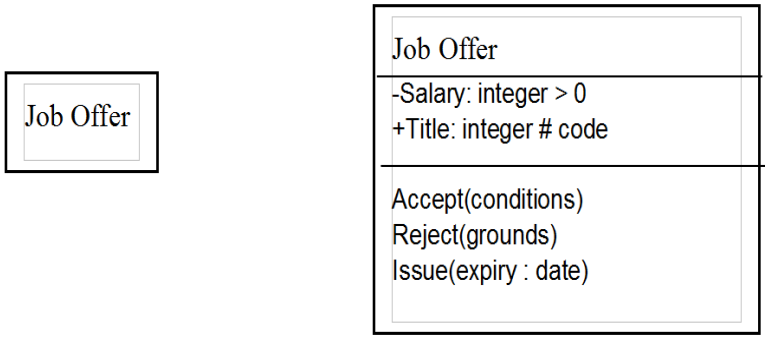


Figure 10-2: A Class is Drawn with Different Levels of Detail in Different Diagrams

You can draw rectangles with names of classes inside them all day, but unless they say something about program organization, such diagrams serve little purpose. The main point of drawing a class diagram is not the classes; it is the relationships between classes that are required by the model. These relationships are called *associations*. In a class diagram, associations are lines that connect classes. Accompanying annotations in the diagram convey relevant model information about the relationships. The details of associations and their implementation are described in Chapter 10.

10.4 Declaring Classes

In Unicon program code, the syntax of a class is:

```
class foo(attribute1, attribute2, attribute3, ...)
    # procedures (methods) to access class foo objects

# code to initialize class foo objects
end
```

The procedures that manipulate class objects are called *methods*. Methods define a class' interface from the rest of the program. The syntax of a method like a procedure:

```
method bar(param1, param2, param3, ...)
    # Unicon code that may access fields of a class foo object
end
```

Execution of a class method is always associated with a given object of that class. The method has access to an implicit variable called self that is a record containing fields whose names are the attributes given in the class declaration. Fields from the self variable are directly accessible by name. In addition to methods, classes may also contain global and record declarations; such declarations have the standard semantics and exist in the global name space.

10.5 Object Instances and Initially Sections

Like records, instances of a class type are created with a constructor function whose name is that of the class. Instances of a class are called objects, and behave similar to records. The fields of an instance generally correspond directly to the class attributes. Fields may be initialized explicitly in the constructor in exactly the same way as for records. For example, after defining a class foo(x, y) one may write:

```
procedure main()
   f := foo(1, 2)
end
```

In this case x would have the value 1, and y would have the value 2, the same as for a record type. The fields of an object do not have to be initialized by a parameter passed to the class constructor. Many constructors initialize objects' fields to some standard value. In this case, the class declaration includes an initially section after its methods are defined and before its end. An initially section is just a special method that is invoked automatically by the system when it creates each instance of the class.

An initially section begins with the word initially and an optional parameter list, followed by lines of code that are executed when an object of that class is constructed. These lines typically assign values to one or more of the attributes of the object being created.

For example, suppose you want an enhanced table type that permits sequential access to elements in the order they are inserted into the table. You can implement this using a combination of a list and a table, both of which would be initialized to the appropriate empty structure:

```
class taque(L, T) # pronounced "taco"
   # methods to manipulate taques,
   # e.g. insert, index, foreach...
initially
   L := [ ]
   T := table()
end
```

In such a case you can create objects without including arguments to the constructor:

```
procedure main()
  mytaque := taque()
end
```

Although the default behavior of classes is the same as records, and constructor arguments normally assign each of the fields in the class in order, there are some important rules that override classes' record-like constructor behavior.

- With no initially section, a class constructor behaves exactly like a record constructor. Class fields are assigned in order from parameters and missing arguments default to the null value.

- With an initially(...) that has a parenthesized list of zero or more formal parameter names, the constructor parameters are used to initialize the formal parameters of the initially() method. In this case, the constructor does not assign class fields from parameters implicitly. Instead the initially section may initialize fields as it sees fit, including initializing them using values from the named parameters of method initially() if it so chooses.

- an initially section without a parenthesized formal parameter list behaves somewhere in between the above two cases. Normal record-like assignment of parameters to fields is performed. Missing arguments start with a null value. However, the initially section may assign those fields without the caller having passed values into the constructor.

The initially section with no parameter list makes it possible to write classes with some fields that are initialized explicitly by the constructor parameters, while other fields are initialized implicitly by code in the initially section. In this case you should declare the automatically initialized fields after those initialized by parameters in the constructor. The parameters are assigned to fields in the constructor exactly in the order they appear in the class declaration.

This default semantics for constructor parameters is awkward in some cases, so there is an alternative. When an initially section includes a parameter list, no implicit initialization of objects' fields is performed. This frees the constructor from having the same number and order of parameters as the declared class fields. In the following example, class C is constructed with only a single parameter even though it has three fields. The actual parameter "Greenwich Village" is bound to initially formal parameter x. The third field in the class (c) is initialized from the constructor parameter x, overriding the default behavior of initializing fields in the declared order. This capability becomes important in large classes with many fields.

```
class C(a, b, c)
initially(x)
    a := ["vital", "urgent", "gentrified"]
    b := promptedread("tell me about " || x)
    c := x
end
...
procedure main()
v := C("Greenwich Village")
end
```

Hopefully all this has convinced you that initially sections are important and useful. They are in fact just a special method that gets called when an object is constructed, but that means they play the role of a *constructor function* that is found in many other object-oriented languages. The next chapter, which goes into inheritance in detail, points out that the inheritance rules apply to initially sections just like any other method. A subclass initially will usually need to invoke its superclass(es)' initially sections, along with initializing any new fields that it introduces.

10.6 Object Invocation

Once you have created an object with a class constructor, you manipulate the object by invoking its class methods. Since objects are both procedures and data, object invocation is a combination of a procedure call and a record access. The syntax is

```
object . methodname ( arguments )
```

If an object's class is known, object methods can also be called using a normal procedure call. This allows object oriented Unicon code to be called from Icon. Called as a procedure, the name of a method is prefixed with the class name and an underscore character. The object itself is always the first parameter passed to a method. In the absence of inheritance (discussed in the next chapter) if x is an object of class C, x.method(arguments) is equivalent to C_method(x, arguments).

Although object methods can be called using procedure calls, the field operator has the advantage that it handles inheritance and polymorphism correctly, allowing algorithms to be coded generically using polymorphic operations. Generic algorithms use any objects whose class provides the set of methods used in the algorithm. Generic code is less likely to require change when you later enhance the program, for example adding new subclasses that inherit from existing ones. In addition, if class names are long, the field syntax is considerably shorter than writing out the class name for the invocation. Using the taque example:

```
procedure main()
    mytaque := taque()
    mytaque.insert("greetings", "hello")
    mytaque.insert(123)
    every write(mytaque.foreach())
    if mytaque.index("hello") then write(", world")
end
```

For object-oriented purists, using the field operator to invoke an object's methods in this manner is the only way to access an object. In Unicon, visibility issues such as "public" and "private" are addressed in an application's design and documentation. A good starting

point is to consider all fields "private" and all methods "public". Nevertheless, an object is just a kind of record, complete with record-style field access.

Direct external access to an object's data fields using the usual field operator is not good practice, since it violates the principle of encapsulation. Within a class method, on the other hand, access to an object's attributes is expected. The implicit object named self is used under the covers, but attributes and methods are referenced by name, like other variables. The taque insert method is thus:

```
method insert(x, key)
   /key := x
   put(L, x)
   T[key] := x
end
```

The self object allows field access just like a record, as well as method invocation like any other object. Using the self variable explicitly is rare.

10.7 Comparing Records and Classes

The concepts of classes and objects are found in many programming languages. The following example illustrates Unicon's object model and provides an initial impression of these concepts' value. To motivate Unicon's OOP constructs, our example contrasts conventional Icon code with object-oriented code that implements the same behavior.

Before objects

Suppose you are writing some text-processing application such as a text editor. Such applications need to be able to process structures holding the contents of various text files. You might begin with a simple structure like the following:

```
record buffer(filename, text, index)
```

where filename is a string, text is a list of strings corresponding to lines in the file, and index marks the current line at which the buffer is being processed. Icon record declarations are global; in principle, if the above declaration needs to be changed, the entire program must be rechecked. A devotee of structured programming would write procedures to read the buffer in from a file; write it out to a file; examine, insert and delete individual lines; and so on. These procedures, along with the record declaration given above, can be placed in their own source file (buffer.icn) and understood independently of the program(s) in which they are used. Here is one such procedure:

```
# read a buffer in from a file
procedure read_buffer(b)
    f := open(b.filename) | fail
    b.text := [ ]
    b.index := 1
    every put(b.text, !f)
    close(f)
    return
end
```

There is nothing wrong with this example; in fact its similarity to the object-oriented example that follows demonstrates that a good, modular design is the primary effect encouraged by object-oriented programming. Using a separate source file to contain a record type and those procedures that operate on the type allows an Icon programmer to maintain a voluntary encapsulation of that type.

After objects

Here is part of the same buffer abstraction coded in Unicon. The purpose here is to facilitate a direct comparison with the preceding record-based example. The example lays the groundwork for a further object-oriented illustration in the following chapter. Classes are record types with associated code. Methods are procedures that are always called in reference to a particular object (a class instance).

```
class buffer(filename, text, index)
    # read a buffer in from a file
    method read()
      f := open(filename) | fail
      text := [ ]
      index := 1
      every put(text, !f)
      close(f)
      return
    end
    # ...additional buffer operations, including method erase()
initially
    if \filename then read()
end
```

This example does not illustrate the full object-oriented style, but it is a start. The object-oriented version offers encapsulation and polymorphism. A separate name space for each class's methods allows shorter names. The same method name, such as read(), can be used in each class that implements a given operation. This notation is more concise than is

possible with procedures, and it enables an algorithm to work on objects of any class that implements the operations required by that algorithm.

Consider the initialization of a new buffer. Constructors allow the initialization of fields to values other than &null. In the example, the read() method is invoked if a filename is supplied when the object is created. This can be simulated using records by calling a procedure after the record is created; the value of the constructor is that it is automatic. The programmer is freed from the responsibility of remembering to call this code everywhere objects are created in the client program(s). This tighter coupling of memory allocation and its corresponding initialization removes one more source of program errors, especially on multi-programmer projects.

The preceding two paragraphs share a common theme: the net effect is that each piece of data is made responsible for its own behavior in the system. Although this example dealt with simple line-oriented text files, the same methodology applies to more abstract entities such as the components of a compiler's grammar.

The example illustrates an important scoping issue. Within class buffer, method read() makes the regular built-in function read() inaccessible! Beware of such conflicts. It would be easy to capitalize the method name to eliminate the problem. If renaming the method is not an option, as a last resort you could get a reference to the built-in function read(), even within method read(), by calling proc("read", 0). The function proc() converts a string to a procedure; supplying a second parameter of 0 tells it to skip scoping rules and look for a built-in function by that name.

10.8 Summary

Classes are global declarations that define a record data type and a set of procedures (methods) that operate on that type. Class instances, called objects, are normally manipulated solely by calling the class' methods; such object privacy is a matter of design, documentation, and convention. All methods execute within an object of interest, whose fields and methods are visible without the record dot notation, in a class scope that is introduced in between the local and global scopes.

Chapter 11

Inheritance and Associations

Relationships between classes are depicted in UML class diagrams by lines drawn between two or more class rectangles. One of the most important relationships between classes describes the situation when one class is an extension or minor variation of another class: this is called generalization, or *inheritance*. Most other relationships between classes are really relationships between those classes' instances at run-time; these relationships are called *associations*. This chapter starts with inheritance, and then describes a variety of associations. In this chapter you will learn how to:

- Define a new class in terms of its differences from an existing class
- Compose aggregate classes from component parts.
- Specify new kinds of associations
- Supply details about the roles and number of objects in an association
- Use structure types to implement associations

11.1 Inheritance

In many cases, several classes of objects are very similar. In particular, many classes arise as enhancements of classes that have already been defined. Enhancements might consist of added fields, added methods, or both. In other cases a class is just a special case of another class. For example, if you have a class fraction(numerator, denominator), you could define class inverses(denominator) whose behavior is identical to that of a fraction, but whose numerator is always 1.

Both of these ideas are realized with the concept of inheritance. When the definition of a class is best expressed in terms of the definition of another class or classes, we call that class a *subclass*. The class or classes from which a subclass obtains its definition are called *superclasses*. The logical relation between the subclass and superclass is called *hyponymy*. It means an object of the subclass can be manipulated just as if it were an object of one

of its defining classes. In practical terms it means that similar objects can share the code that manipulates their fields.

Inheritance appears in a class diagram as a line between classes with an arrow at one end. The arrow points to the superclass, the source of behavior inherited by the other class. Consider Figure 11-1, in which an offer of a salaried appointment is defined as one kind of job offer. The attributes (salary, title) and methods (Accept() and Reject()) of class JobOffer are inherited by class SalariedAppointment, and do not need to be repeated there. A new attribute (term) is added in SalariedAppointment that is not in JobOffer.

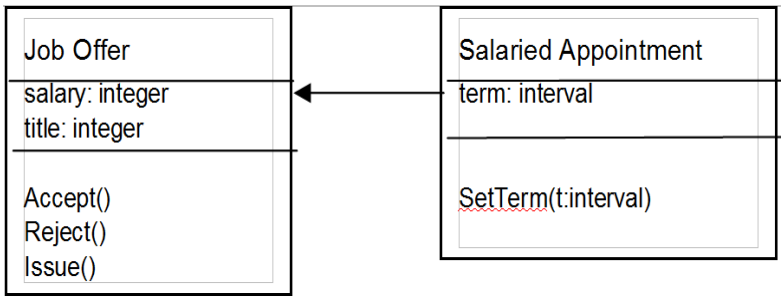


Figure 11-1: A Salaried Appointment is a subclass of a Job Offer

The syntax of a subclass is

```
class classname superclasses (attributes)
    methods
initially_section
end
```

Where *superclasses* is an optional list of class names separated by colons, *attributes* is an optional list of variable names separated by commas, *methods* is an optional list of declarations for the class methods, and the *initially_section* is optional initialization code for the class constructor. For example

```
class SalariedAppointment : JobOffer (term)
    method SetTerm(t : interval)
      term := t
    end
initially
    /term := "unknown term"
end
```

As you can see, a subclass declaration is identical to a regular class, with the addition of one or more superclass names, separated by colons. The meaning of this declaration is the subject of the next section.

Inheritance semantics

There are times when a new class might best be described as a combination of two or more classes. Unicon classes may have more than one superclass, separated by colons in the class declaration. This is called multiple inheritance.

Subclasses define a record type consisting of the field names of the subclass itself and all its superclasses. The subclass has associated methods consisting of those in its own body, those in the first superclass that were not defined in the subclass, those in the second superclass not defined in the subclass or the first superclass, and so on. In ordinary single-inheritance, this adding of fields and methods follows a linear bottom-up visit of each superclass, followed in turn by its parent superclass.

When a class has two or more superclasses, the search generalizes from a linear sequence to an arbitrary tree, directed acyclic graph, or full graph traversal. Multiple inheritance adds fields and methods in an order defined by a depth-first traversal of the parent edges of the superclass graph. Think of the second and following superclasses in the multiple inheritance case as adding methods and fields only if the single-inheritance case (following the first superclass and all its parents) has not already added a field or method of the same name.

Warning

Care should be taken employing multiple inheritance if the two parent classes have any fields or methods of the same name!

Fields are initialized by parameters to the constructor or by the class initially section, which is a method and is inherited in the normal way. It is common for a subclass initially section to call their superclasses' initially sections, for example:

```
class sub : A : B(x)
initially
    x := 0
    A.initially()
    B.initially()
end
```

It is also common to have some attributes initialized by parameters, and assign others in the initially section. For example, to define a class inverse (for numbers of the form 1 / n) in terms of a class fraction(numerator, denominator) one can write:

```
class inverse : fraction (denominator)
initially
    numerator := 1
end
```

Objects of class inverse can be manipulated using all the methods defined in class fraction; the code is actually shared by both classes at runtime.

Viewing inheritance as the addition of superclass elements not found in the subclass is the opposite of the more traditional object-oriented view that a subclass is an instance of the superclass as augmented in the subclass definition. Unicon's viewpoint adds quite a bit of leverage, such as the ability to define classes that are subclasses of each other. This feature is described further below.

Invoking superclass operations

When a subclass defines a method of the same name as a method defined in the superclass, invocations on subclass objects execute the subclass' version of the method. This can be overridden by explicitly including the superclass name in the invocation:

```
object$superclass.method(parameters)
```

This facility allows the subclass method to do any additional work required for added fields before or after calling an appropriate superclass method to achieve inherited behavior. The result is frequently a chain of inherited method invocations.

Since initially sections are methods, they can invoke superclass operations including superclass initially sections. This allows a chain of initially sections to be specified to execute in either subclass-first or superclass-first order, or some mixture of the two.

Inheritance examples

Several inheritance examples follow from the buffer example from Chapter 10. Suppose the program does more than edit text, it has a word-associative dictionary, bibliography, spell-checker, and thesaurus. These features can be implemented using the table type. The contents vary, but they all use string keyword lookup. As external data, the databases can be stored in text files, one entry per line, with the keyword at the beginning. The format of the rest of the line varies from database to database.

Figure 11-2 shows a class diagram with subclasses derived from buffer. A class buftable refines the buffer class, adding support for random access. Other classes are defined as subclasses of buftable. The implementation of these classes is given below.

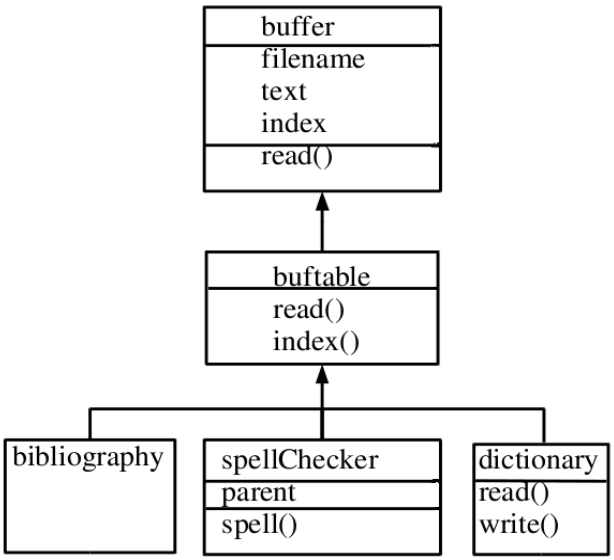


Figure 11-2: Some Subclasses of the Buffer Class

Although all these types of data are different, the code used to read the data files can be shared, as well as the initial construction of the tables. In fact, since we are storing our data one entry per line in text files, we can use the code already written for buffers to do the file I/O itself.

```
class buftable : buffer()
  method read()
     self.buffer.read()
     tmp := table()
     every line := !text do
       line ? { tmp[tab(many(&letters))] := line | fail }
     text := tmp
     return
  end
  method index(s)
     return text[s]
  end
end
```

This concise example shows how little must be written to achieve data structures with vastly different behavioral characteristics, by building on code that is already written. The superclass read() operation is one important step of the subclass read() operation. This technique is common enough to have a name: it is called *method combination* in the literature. It allows you to view the subclass as a transformation of the superclass. The buftable class is given in its entirety, but our code sharing example is not complete: what about the data structures required to support the databases themselves? They are all

variants of the buftable class, and a set of possible implementations follow. Note that the formats presented are designed to illustrate code sharing; clearly, an actual application might make different choices.

Bibliographies Bibliographies might consist of a keyword followed by an uninterpreted string of information. This imposes no additional structure on the data beyond that imposed by the buftable class. An example keyword would be Jeffery99.

```
class bibliography : buftable()
end
```

Spell-checkers The database for a spell-checker might be a list of words, one per line, the minimal structure required by the buftable class. Some classes introduce new terminology rather than to define a new data structure. This example introduces a lookup operation that can fail, for use in tests. In addition, since many spell-checking systems allow user definable dictionaries in addition to their central database, spellChecker objects may chain together for the purpose of looking up words.

```
class spellChecker : buftable(parent)
  method spell(s)
     return \text[s] | (\parent).spell(s)
  end
end
```

Dictionaries Dictionaries are slightly more involved. Each entry might consist of a part of speech, an etymology, and an arbitrary string of uninterpreted text comprising a definition for that entry, separated by semicolons. Since each such entry is itself a structure, a sensible decomposition of the dictionary structure consists of two classes: one that manages the table and external file I/O, and one that handles the manipulation of dictionary entries, including their decoding and encoding as strings.

```
class dictionaryentry(word, partofspeech, etymology, definition)
  # decode a dictionary entry into its components
  method decode(s)
     s ? {
       word        := tab(upto(';'))
       move(1)
       partofspeech := tab(upto(';'))
       move(1)
       etymology    := tab(upto(';'))
       move(1)
       definition   := tab(0)
     }
```

```
  end
  method encode()  # encode a dictionary entry into a string
     return word || ";" || partofspeech || ";" || etymology || ";" || definition
  end
initially
  if /partofspeech then {
     # constructor was called with a single string argument
     decode(word)
     }
end

class dictionary : buftable()
  method read()
     self.buffer.read()
     tmp := table()
     every line := !text do
       line ? { tmp[tab(many(&letters))] := dictionaryentry(line) | fail }
     text := tmp
  end
  method write()
     f := open(filename, "w") | fail
     every write(f, (!text).encode)
     close(f)
  end
end
```

Thesauri Although an oversimplification, one might conceive of a thesaurus as a list of entries, each of which consists of a comma-separated list of synonyms followed by a comma-separated list of antonyms, with a semicolon separating the two lists. Since the code for such a structure is nearly identical to that given for dictionaries above, we omit it here (you might start by generalizing class dictionaryentry to handle arbitrary strings organized as fields separated by semicolons).

A (toy) Dictionary Program

The above examples were intended to illustrate a pedagogical point about how inheritance facilitates specialization of existing code. Although they are legal Unicon code, they were always toy examples that illustrate ideas, not an extract from a real software application. Consult the Unicon translator implementation, or the Unicon GUI classes, for many real-world examples of inheritance.

Having said that, here is a complete program that makes use of some of the preceding class examples. The program, called deen (from Deutsch-English) reads in a dictionary in

plain text format, originally obtained from http://ftp.tu-chemnitz.de/pub/Local/urz/ding/de-en/ and writes out English entries for one or more German language words given on the command-line. German was chosen fairly arbitrarily here, but the code does depend on the file format of the dictionary file, de-en.txt.

```
#
# deen.icn - give English equivalents of German words
#
$define DEEN "http://ftp.tu-chemnitz.de/pub/Local/urz/ding/de-en/de-en.txt.gz"

procedure main(av)
    if av[1]=="-all" then all := pop(av)
    dd := DeenDictionary()
    every s := !av do
        if lu := dd.lookup(s) then {
            if \all then
                every write(s, ": ", dd.lookup(s).definition)
            else write(s, ": ", lu.definition)
            }
        else write(s, " is not in the dictionary.")
end

class buffer(filename, text)
    # read a buffer in from a file
    # todo: decompress if .gz extension
    method read()
        if match("http://", filename) then mode := "m" else mode := "r"
        f := open(filename, mode) | stop("can't open ", image(filename))

        if filename[-3:0] == ".gz" then {
            if mode=="m" then { # download_to_local_file
                if not (f2 := open("de-en.txt.gz","w")) then
                    stop("can't write")
                while s := reads(f, 1000000) do writes(f2, s)
                close(f)
                close(f2)
                }
            system("gunzip de-en.txt.gz")
            f := open("de-en.txt") | stop("can't read de-en.txt")
            }

        writes("Opened ",image(f),".\nReading")
        text := [ ]
        every put(text, !f) do if *text%1000=0 then writes(".")
```

```
        close(f)
        write("\ndone. Read ", *text, " lines")
        return
    end
    method erase()
        #... ?
    end
    # ...additional buffer operations
initially
    if \filename then read()
end

class buftable : buffer()
    method read()
        self.buffer.read()
        tmp := table()
        every line := !text do {
            line ? {
                word := tab(many(&letters)) | stop("failed on ", image(line))
                tmp[word] := line
                }
            }
        text := tmp
        return
    end
    method lookup(s)
        suspend ! \ (text[s])
    end
end

class dictionaryentry(word, part, etymology, definition)
    # decode a dictionary entry into its components
    # assumed format is word;pos;eym;def
    method decode(s)
        s ? {
            word := tab(find(";"))
            move(1)
            part := tab(find(";"))
            move(1)
            etymology := tab(find(";"))
            move(1)
            definition := tab(0)
            }
    end
```

```
    method encode() # encode a dictionary entry into a string
        return word || ";" || part || ";" || etymology || ";" || definition
    end
initially
    if /part then # constructor was called with a single string argument
        decode(word)
end

class dictionary : buftable()
    method read()
        self.buffer.read()
        tmp := table()
        every line := !text do
            line ? {
                word := tab(many(&letters)) | stop("failed on ", image(line))
                tmp[word] := dictionaryentry(line) | fail
                }
        text := tmp
    end
    method Write()
        f := open(filename, "w") | fail
        every write(f, (!text).encode)
        close(f)
    end
end

class DeenEntry : dictionaryentry(gender)
initially(de, en)

    de ? {
        if word := trim(tab(find("{"))„0) then {
            ="{"
            gender := tab(find("}"))
            }
        else { # here is one without gender info
            word := trim(tab(find("[")|0)„0)
            gender := "?"
            }
    }
    definition := en
end

#
# Return a list of dictionary entries for a given line of text
```

```
#
procedure get_entries(s)
    subentries := []
    s ? {
        deutsch := tab(find("::")) | stop("no :: in ", image(s))
        ="::"; tab(many(' '))
        english := tab(0)

        deutsch ? {
            while *(deutschwort := tab(find("|") | 0))>0 do {
                deutschwort := trim(deutschwort,,0)
                ="|"
                tab(many(' '))
                englishword := trim(english[1:find("|",english)|0],,0)
                english ?:= {
                    tab(many(' \t'))
                    =englishword
                    tab(many(' \t'))
                    ="|"
                    tab(many(' \t'))
                    tab(0)
                    }
                if i := find(";", deutschwort) then {
                    deutschwort ? {
                        while *(dword := tab(find(";") | 0))>0 do {
                            =";"
                            tab(many(' '))
                            if gronk:=englishword[1:upto(';|', englishword)|0] then {
                                if *gronk>0 then {
                                    eword := gronk
                                    }
                                }
                            put(subentries, DeenEntry(dword, eword))
                            englishword ?:= { =eword; =";"; tab(many(' ')); tab(0)}
                            }
                        }
                    }
                else {
                    put(subentries, DeenEntry(deutschwort, englishword))
                    }
                }
            }
        }
    return subentries
```

```
end

class DeenDictionary : dictionary()
    method read()
        self.buffer.read()
        tmp := table()
        every line := !text do
            line ? {
                if ="#" | line=="" then next
                if not (L := get_entries(line)) then
                    stop("get_entries failing on ", image(line))
                every x := !L do {
                    if not member(tmp, x.word) then
                        tmp[x.word] := [x]
                    else put(tmp[x.word], x)
                    }
                }
        text := tmp
    end
initially
    if stat("de-en.txt") then filename := "de-en.txt"
    else if stat("de-en.txt.gz") then filename := "de-en.txt.gz"
    else filename := DEEN
    self.read()
end
```

Superclass cycles and type equivalence

In many situations, there are several ways to represent the same abstract type. Two-dimensional points might be represented by Cartesian coordinates x and y, or equivalently by radial coordinates expressed as a distance d and angle r given in radians. If one implements classes corresponding to these types there is no reason one of them should be considered a subclass of the other; they are interchangeable and equivalent.

In Unicon, expressing this equivalence is simple and direct. In defining classes Cartesian and Radial we may declare them to be superclasses of each other:

```
class Cartesian : Radial (x, y)
# code that manipulates objects using Cartesian coordinates
end

class Radial : Cartesian (d, r)
# code that manipulates objects using radial coordinates
end
```

These superclass declarations make the two types equivalent names for the same type of object; after inheritance, instances of both classes will have fields x, y, d, and r, and support the same set of operations.

Equivalent types each have a unique constructor given by their class name. Often the differing order of the parameters used in equivalent types' constructors reflects different points of view. Although they export the same set of operations, the actual procedures invoked by equivalent types' instances may be different. For example, if both classes define an implementation of a method print(), the method invoked by a given instance depends on which constructor was used when the object was created.

If a class inherits methods from one of its equivalent classes, it is responsible for initializing the state of the fields used by those methods in its constructor. It is also responsible for maintaining the state of the inherited fields when its methods make state changes to its own fields. In the geometric example given above, for class Radial to use any methods inherited from class Cartesian, it must at least initialize x and y explicitly in its constructor from calculations on its d and r parameters. This added responsibility is minimized in those classes that treat an object's state as immutable.

11.2 Associations

An association denotes a relationship between classes that occurs between specific instances of the classes at runtime. Just as objects are instances of classes, associations have instances called *links*. Like objects, links have a lifetime, from the instant at which the relationship is established to the time at which the relationship is dissolved. Besides serving to connect two objects, links may have additional state information or behavior; in the most general case links can be considered to be objects themselves: special objects whose primary role is to interconnect other objects.

11.3 Aggregation

Inheritance may be the most famous kind of relationship between classes, but it is not the most indispensable. Many languages that provide objects do not even bother with inheritance. The composition of assembly objects from their component parts, on the other hand, is a truly ubiquitous and essential idea called *aggregation*. The class dictionary defined in the previous section is a good example of an aggregate object; its component parts are dictionaryentry objects.

Aggregation is depicted in class diagrams by a line between classes with a diamond marking the aggregate, or assembly, class. Figure 11-3 shows an aggregate found in the domain of sports, where a team is comprised of a group of players.

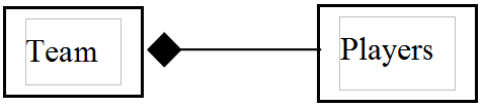


Figure 11-3: A Team is an Aggregation of Players

Unlike inheritance, aggregation describes a relationship between instances at run-time. Different aggregate instances are assembled from different component part instances. While a given player might play for different teams over a period of time, at any given instant a player is normally part of at most one team.

11.4 User-defined associations

All the other relationships between classes in UML are left to the application designer to specify as custom associations. User-defined associations are depicted by lines, annotated with the association name next to the line, midway between the two classes. Figure 11-4 shows a silly user-defined association that describes a family relationship called Marriage. For a diagram containing such an association to be well defined, the semantics of such a relationship must be specified in external documentation that describes Marriage for the purposes of the application at hand.

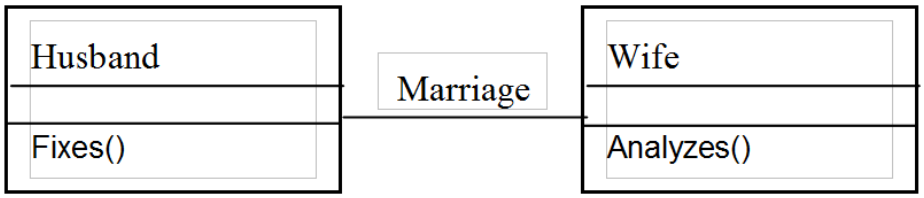


Figure 11-4: A User-Defined Association

Multiplicities, roles, and qualifiers

Whether they are aggregations or user-defined application domain relationships, associations are not completely specified until additional details are determined during program design, such as how many instances of each type of object may participate in a given relationship. These additional details appear in canonical positions relative to the line that depicts the association within a class diagram.

A *multiplicity* is a number or range that indicates how many instances of a class are involved in the links for an association. In the absence of multiplicity information an association is interpreted as involving just one instance. It normally appears just below the association line next to the class to which it applies. Figure 11-5 shows a BasketballTeam that is an aggregate with a multiplicity of five Players.

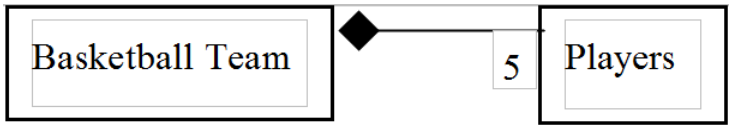


Figure 11-5: Multiplicity of a Basketball Team

A multiplicity range is expressed as a pair of numbers separated by two periods, as in 1..3. The value * may be used by itself to indicate that a link may associate any number (zero or more) of objects. The * value may also be used in a range expression to indicate no upper bound is present, as in the range 2..*.

A *role* is a name used to distinguish participants and responsibilities within an association. Roles are drawn just above or below the association line, adjacent to the class to which they refer. They are especially useful if a given association may link multiple instances of the same class in asymmetric relationships. Figure 11-6 shows a better model of the classes and roles involved in a Marriage association depicted earlier in Figure 11-4.

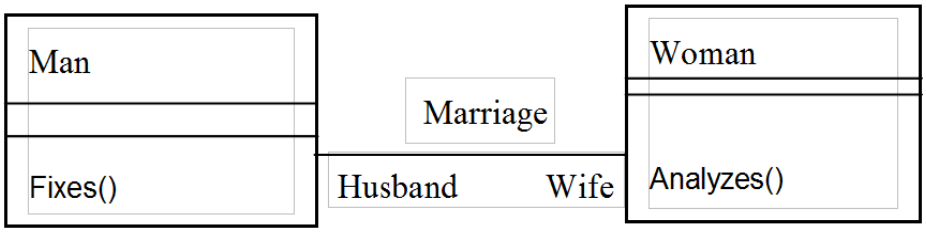


Figure 11-6: Roles in an Association

A *qualifier* is a key value used to distinguish instances in a link, in lieu of a large multiplicity that would otherwise be inefficient. For example, a directory may contain many files, but each one may be directly accessed by name. A qualifier is drawn as a rectangular association end with the qualifier key given inside the rectangle. Figure 11-7 shows a reinterpretation of the basketball aggregation in which the players on the team are distinguished using a qualifier key named position.

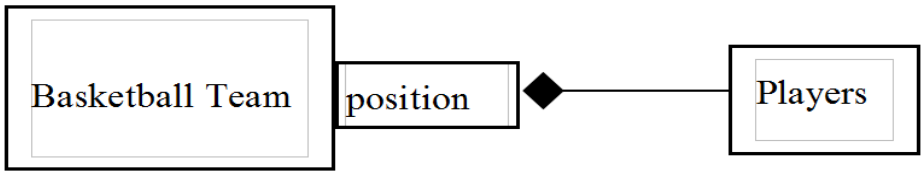


Figure 11-7: Using a Qualifier in lieu of multiplicity

Implementing associations

Unlike inheritance, which is implemented by the language and resolved at compile time, associations involve dynamic relationships established at runtime, and are implemented by the programmer...or are they? In the most general case, an association may be implemented by writing a class whose instances are links between the related classes' objects. In the

narrowest special case, an association can be implemented by adding an attribute in one class to contain a reference to an object in an associated class. Much of the value introduced by multiplicity and qualifier information is to narrow the association semantics down to what is readily implementable using the built-in structure types instead of writing classes for them. If an association can be implemented using a list or table instead of defining a new class, the resulting code will be smaller and faster.

In all cases, associations will introduce additional fields into the classes being associated. The following code example implements the Marriage association from Figure 11-6. For illustration purposes it is modeled as a one-one relationship at any given point in time. Notice the intertwining methods in the two classes that establish a bi-directional relationship. Error checking is left as an exercise for the reader.

```
class Man(wife)
    method Marry(w)
      wife := w
      if not (self === w.husband) then w.Marry(self)
    end
end
class Woman(husband)
    method Marry(m)
      husband := m
      if not (self === m.wife) then m.Marry(self)
    end
end
```

As a general rule, an association that has a qualifier is implemented with a table. The following example corresponds to the basketball team diagram in Figure 11-7. The players attribute might be a list or set in class Team, but the qualifier allows class BasketballTeam to override this and implement players using a table. Such a refinement can be awkward in a statically typed object-oriented language. Depending on whether its player parameter is supplied or is null, method Player() serves to either lookup a player, given a position, or to insert a player into the association. In either case the player at the designated position is returned.

```
class BasketballTeam : Team ()
    method Player(position, player)
      players[position] := \player
      return players[position]
    end
initially
    players := table()
end
```

Associations with multiplicity might be implemented using sets or lists, with lists being favored when multiplicity is bounded to some range, or when there is a natural ordering among the instances. The following version of the BasketballTeam class uses a list of five elements to implement its aggregation, which occupies less space than either a set or the table in the last example.

```
class BasketballTeam : Team (players)
    method Player(player)
      if player === !players then fail # already on the team
      if /!players := player then return # added at null slot
      ?players := player # kick someone else off team to add
    end
initially
    players := list(5)
end
```

Defining a new class to implement an association handles rare cases such as many-many relationships and associations that have their own state or behavior. Other examples of associations and their implementation are given in Part 3 of this book.

11.5 Summary

The relationships between classes are essential aspects of the application domain that are modeled in object-oriented programs. You can think of them as the "glue" that connects ideas in a piece of software. Class diagrams allow many details of such relationships to be specified graphically during design. Unicon's structure types allow most associations to map very naturally onto code. In order to understand the subtleties of how these features are implemented, you may wish to study the output of unicon -E for various examples; the -E option writes out the Icon translation of the object-oriented code.

Chapter 12

Writing Large Programs

This chapter describes language features, techniques, and software tools that play a supporting role in developing large programs and libraries of reusable code. You can write large programs or libraries without these tools; the tools just make certain tasks easier. These facilities are not specific to object-oriented programs. However, they serve the same target audience since one of the primary benefits of object-orientation is to reduce the difficulty of developing large programs.

In the case of Unicon, "large" might mean any program so complex that it is not self-explanatory without any special effort. This includes all programs over a few thousand lines in size, as well as shorter programs with complex application domains, and those written by multiple authors or maintained by persons other than the original author.

Writing and maintaining large programs poses additional challenges not encountered when writing small programs. The need for design and documentation is greater, and the challenge of maintaining the correspondence between design documents and code is more difficult. Design patterns can help you with the design process, since they introduce easily recognizable idioms within a design. The more familiar the design is, the less cognitive load is imposed by the task of understanding it.

This chapter shows you how to:

- Understand the difference between abstract and concrete classes
- Use design patterns to simplify and improve software designs
- Organize programs and libraries into packages
- Generate HTML indices and reference documentation for your code

12.1 Abstract Classes

In programming languages, it turns out there are at least two very different kinds of things referred to by the word "class". Most classes denote a data type, of which there are one or

more instances. The word class is also used to denote a general category of objects, of which there are no actual instances. Such classes are called abstract classes. Abstract classes are used by defining subclasses which do have instances. In a small program you might not run into a need for abstract classes, but if you write a large object-oriented program, or your small program uses someone else's large object-oriented class library, you are likely to need to understand abstract classes.

The whole idea of class and inheritance originated by analogy from biology, but if you think about it, biology uses a host of different terms to denote categories (phylum, family, genus, etc.), that are distinct from the term that denotes a class of instances (species). There are plenty of instances of species *Homo sapiens*, but there are no instances of a genus *Homo*, there are only instances of *Homo*'s subclasses. Similarly, if you are designing software to model the behavior of cars and trucks, you may identify a lot of shared behavior and place code for it in a superclass Vehicle, but there are no instances of Vehicle, only its subclasses.

In a larger inheritance graph (or tree) most classes may well be abstract. It may be those classes (primarily leaves) of which there are actual instances that are special and deserve a different term besides "class". We could easily think of all classes as abstract by default, and refer to classes that have instances as "concrete classes" or "instantiation classes". The analogy to biology would be better served by such terminology. But for better or for worse, most software engineers will have to live with "abstract class" and "class" to denote general categories and instantiable categories, respectively.

Numerous larger and more concrete examples of abstract classes appear in the Unicon GUI class library, described in Chapter 17. Some classes, such as Button, denote general categories of widgets that have code and behavior in common (such as the fact that you can click them). There are no instances of Button, there are only subclasses such as TextButton, IconButton, and CheckBox, that may be instantiated. In larger applications, the code sharing (or duplication avoidance) of abstract classes may be compelling, but like all uses of inheritance they do require you to read and understand multiple bodies of code (the class and all its superclasses) in order to use the class effectively.

Although some abstract classes like Button look exactly like regular classes and have to be identified as abstract via their supporting comments or documentation, Unicon does have one language feature that is only used within abstract classes. The reserved word abstract may be given preceding a method header in lieu of providing the actual body of the method. Such an *abstract method declaration* implies that subclasses must provide an implementation of the method in question, as it will be called by other methods in the abstract class or in client code. For example, the Unicon GUI Component class is an abstract class that is a superclass of all GUI components. It provides many methods that components can all inherit and use or override. It also declares one abstract method that all subclasses must implement to indicate what they do (if anything) in response to the passage of time:

```
abstract method tick()
```

Class Button is a subclass of Component and provides an implementation of method tick():

```
method tick()
    if dispatcher.curr_time_of_day() > self.repeat_delay then
        fire(BUTTON_HELD_EVENT)
end
```

12.2 Design Patterns

Class and function libraries provide good mechanisms for code reuse, and inheritance helps with code reuse in some situations. But in large programs it is desirable to reuse not just code, but also successful designs that capture the relationships between classes. Such design reuse is obtained by identifying *design patterns* with known successful uses in a broad range of application domains. For example, the practice of using pipes to compose complex filters from more primitive operations has been successfully used in compilers, operating system shells, image processing, and many other application areas. Every programmer should be familiar with this pattern.

The field of software design patterns still quite young. Practitioners are producing simple catalogs of patterns, such as the book *Design Patterns*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. When the field is more mature it will include syntactic and/or semantic rules for how patterns are combined to form higher-order patterns, as is the case for building architecture. This section presents a few classic patterns and their implementation in Unicon.

At best, this discussion of patterns may whet your appetite to go read more about the subject. In addition to their design reuse value, patterns also provide software developers with a common vocabulary for discussing recurring design concepts. The judicious use of one or more abstract classes seems to be a recurring theme throughout most of the design patterns identified by Gamma et al.

Singleton

Perhaps the simplest design pattern is the singleton, describing a class of which exactly one instance is required. Singletons are interesting because they are related to *packages*, a language feature described later in this chapter. A package is a mechanism for segregating a group of global objects so that their names do not conflict with the rest of the program. This segregation is similar to that provided by object encapsulation; a package is similar to a class with only one instance.

Consider as an example a global table that holds all the records about different employees at a small company. There are many instances of class Employee, but only one instance

of class EmployeeTable. What is a good name for this instance of EmployeeTable, and how can you prevent a second or subsequent instance from being created? The purpose of the singleton pattern is to answer these questions.

In Unicon, one interesting implementation of a singleton is to replace the constructor procedure (a global variable) by the instance. Assigning an object instance to the variable that used to hold the constructor procedure allows you to refer to the instance by the name of the singleton class. It also renders the constructor procedure inaccessible from that point on in the program's execution, ensuring only one instance will be created.

```
class EmployeeTable(...)
initially
    EmployeeTable := self
end
```

There are undoubtedly other ways to implement singleton classes.

Proxy

A proxy is a "stand-in" for an object. The proxy keeps a reference to the object it is replacing, and implements the same interface as that object by calling the object's version of the corresponding method each time one of its methods is invoked. Figure 12-1 shows a proxy serving a client object in lieu of the real object.

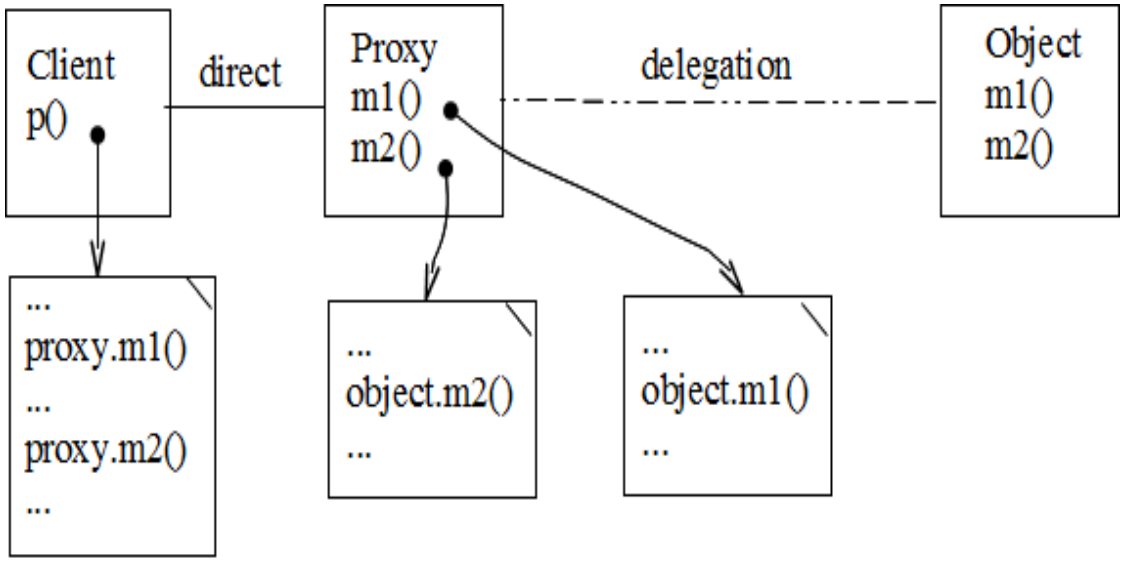


Figure 12-1: The Proxy Pattern

Proxies are used when the original object cannot or should not be invoked directly. If the object is on a remote machine, the proxy can take care of network communication and hide the location of the object from the clients. As another example, the original object

may be instantiated lazily - it may be a large object that is not loaded into memory unless one of its operations is invoked.

Similar to both of the preceding examples, mobile objects might be implemented using proxies. If several machines are running a computation jointly and communicating, some gigantic object might be kept on one machine at a time. In applications with strong locality of reference, whenever a machine needs to do a call on the gigantic object it might do hundreds of calls on that object. In that case the object should move from wherever it is, to the machine where it is needed. The rest of the program does not need to be aware of whether the object is local, remote, or mobile; it just interacts with the proxy instance.

```
class gigantic(x1,x2,...,x1000)
    method invoke()
      ...
    end
initially
    # Gigantic object's state is loaded from network
end
class proxy(g)
    method invoke()
      /g := gigantic()
      return g.invoke()
    end
    method depart()
      g := &null
    end
end
```

Chain of responsibility

This pattern is similar to a proxy, in that an object is *delegating* one or more of its methods to a second object. It is not presented in detail, but its similarity to proxies is mentioned because many design patterns in the Gamma book seem incredibly similar to each other; reading the book is like having déjà vu all over again. Perhaps there ought to be some kind of orthogonality law when it comes to patterns.

The difference between a chain of responsibility and a proxy is that the proxy forwards *all* method invocations to the "real" object, while in a chain of responsibility, the object may handle some methods locally, and only delegate certain methods to the next object in the chain. Also, proxies are normally thought of as a single level of indirection, while the chain of responsibility typically involves multiple linked objects that jointly provide a set of methods. The following example illustrates a chain of responsibility between a data structure object (a cache) and an Image class that knows how to perform a computationally intensive resolution enhancement algorithm.

```
class Image(...)
    method enhance_resolution(area)
      # enormous computation...

    end
initially
    # ... lots of computation to initialize lots of fields
end

class hirez_cache(s, t)
    method enhance_resolution(area)
      if member(t,area) then { # proxy handles
        return t[area]
        }
      # else create the gigantic instance
      /im := image()
      return t[area] := im.enhance_resolution(area)
    end
initially
    t := table()
    # Insert some known values for otherwise enormous computation.
    # Don't need im if user only needs these values.
    t[1] := 1
    t[2] := 1
end
```

The instance of class Image is not created until one is needed, and image's method enhance_resolution() is not invoked for previously discovered results. Of course, enhance_resolution() must be a pure mathematical function that does not have any side effects for this caching of results to be valid.

Visitor

The visitor pattern is a classic exercise in generic algorithms. It is fairly common to have a structure to traverse, and an operation to be performed on each element of the structure. Writing the code for such a traversal is the subject of many data structure texts. In fact, if you have one operation that involves traversing a structure, there is a good chance that you have (or will someday need) more than one operation to perform for which the same traversal is used. Figure 12-2 illustrates the visitor pattern.

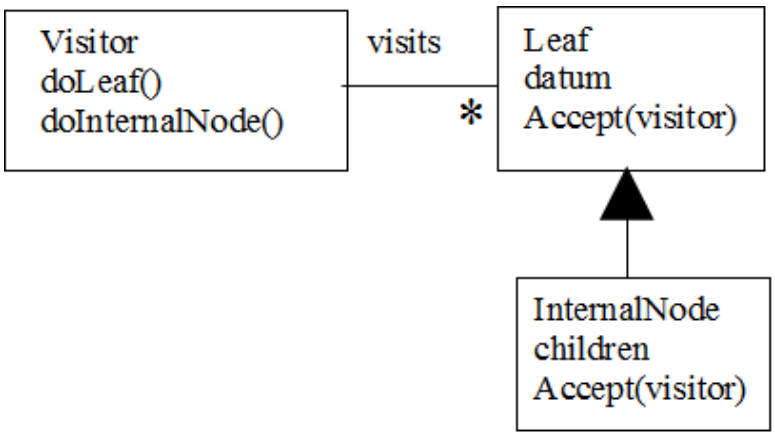


Figure 12-2: The Visitor Pattern

The visitor pattern says you can separate out the traversal algorithm from the operation performed at each element of the structure, and reuse the traversal on other operations. The following code illustrates this separation of traversal (implemented by method Accept()) from visitation (implemented by methods DoLeaf() and DoInternalNode() in the visitor). Where there is one kind of visitor there may be many, and in that case, class Visitor may be an abstract class, instantiated by many concrete Visitor subclasses that have the same method names but do not share code. Note also that this code example allows for heterogeneous structures: the Visitor just defines a "Do..." method for each type of node in the structure.

```
class Visitor()
    method DoLeaf(theLeaf)
      # ... visit/use theLeaf.datum
    end
    method DoInternalNode(theNode)
      # ... visit/use theNode.datum
    end
end
class Leaf(datum)
    method Accept(v)
      v.DoLeaf(self)
    end
end
class InternalNode : Leaf(children)
    method Accept(v)
      every (!children).Accept(v)
      v.DoInternalNode(self)
    end
end
```

Executing a traversal from a root object looks like root.Accept(myvisitor) where myvis-

itor is an instance of some Visitor class. The point of the Visitor pattern is that you can define different Visitor classes. For example, here are Visitors to print a tree, and to calculate and store the heights of all nodes in the tree:

```
class Printer()
    method DoLeaf(theLeaf)
      writes(theLeaf.datum, " ")
    end
    method DoInternalNode(theNode)
      writes(theNode.datum, " ")
    end
end
class Heights()
    method DoLeaf(theLeaf)
      theLeaf.datum := 1
    end
    method DoInternalNode(theNode)
      theNode.datum := 0
      every theNode.datum <:= (!children).datum
      theNode.datum +:= 1
    end
end
```

12.3 Packages

In large programs, the global name space becomes crowded. You can create a disaster if one of your undeclared local variables uses the same name as a built-in function, but at least you can memorize the names of all the built-in functions and avoid them. Memorization is no longer an option after you add in hundreds of global names from unfamiliar code libraries. You may accidentally overwrite some other programmer's global variable, without any clue that it happened.

Packages allow you to partition and protect the global name space. A package is similar to a "singleton" class with only one instance. Every global declaration (variables, procedures, records, and classes) is "invisible" outside the package, unless imported explicitly.

The **package** declaration

A package declaration specifies that all global symbols within a source file belongs to a package. The package declaration looks similar to the link declaration. You provide the package name, an identifier, or a string filename:

```
package foo
```

or

```
package "/usr/local/lib/icon/foo"
```

There can be only one package declaration in a source file. It need not be at the beginning of the source file, but this is conventional. Within a package, global names defined inside the package are referenced normally. Global names outside the package are not visible by default. Here is an example source file that declares some globals and adds them to a package.

```
# pack1.icn
package first
procedure my_proc()
    write("In my_proc")
end
class SomeClass()
    method f()
      write("In SomeClass.f")
    end
end
```

When this code is compiled, the information that package first contains the symbols my_proc and SomeClass is recorded into a database and that using package first implies linking in pack1.u along with any other files that are part of package first. In order to prevent name conflicts the compiler also applies a name mangling process to the global symbols, described below.

The **import** declaration

To access symbols within another package, use the import declaration, which has the following syntax:

```
import foo
```

This causes the compiler to look up the package in its database and identify its symbols. Import declarations use the IPATH environment variable in the same way as do link declarations. In particular, an import declaration *is* a link declaration, augmented with scope information about the names defined in the package.

Explicit package references

Sometimes, two imported packages may define the same symbol, or an imported symbol conflicts with a global declaration in one of your files. To resolve these problems, you can explicitly specify the package to use for particular symbol references. For example, if packages first and second both define a procedure named write, then

```
import first, second
procedure main()
    first::write()    # calls write() in package first
    second::write()    # calls write() in package second
    ::write()      # calls the global write()
end
```

The use of the :: operator on its own is a useful way to refer to a global procedure from within a class that has a method of the same name, as in

```
class Abc(x)
    method write()
      ::write("Abc x=", x)
    end
end
```

In this example, omitting the :: would cause the write() method to call itself until the program runs out of memory and produces a runtime error.

Name conflicts and name mangling

The purpose of packages is to reduce name conflicts, especially accidental ones. You will get a link error if you declare the same name twice in the same package. You will get a compile error if you try to import a package that contains a variable that is already declared. In Unicon, unlike Arizona Icon, you will also get a warning message if you declare a global variable of the same name as a built-in function, or assign a new value to such a name. Often this is done on purpose, and it shows off the flexibility of the language. But other times when it happens by accident, it is a disaster. Such warnings can be turned off with the -n option to the unicon compiler.

Under the hood, packages are implemented by simple name mangling that prefixes the package name and a pair of underscores onto the front of the declared name. You can easily defeat the package mechanism if you try, but the reason to mention the name mangling is so you can avoid variable names that look like names from other packages.

A similar name mangling constraint applies to classes. Also, the compiler reserves field names __s and __m for internal use; they are not legal class field names. Identifiers consisting of _n, where n is an integer are reserved for Unicon temporary variable

names. Finally, for each class foo declared in the user's code, the names foo, foo__state, foo__methods, and foo__oprec are reserved, as are the names foo_bar corresponding to each method bar in class foo.

Compilation order and the **unidep** tool

When possible, you should compile all files in a package before you import that package. Even if you do, if multiple source files belong to the same package, the order in which they are compiled is significant. Consider the following code in three source files:

```
# order1.icn
package demo
procedure first()
    write("first")
end

# order2.icn
package demo
procedure second()
    write("second")
    first()
end

# order3.icn
import demo
procedure main()
    second()
end
```

Files order1.icn and order2.icn belong to a package demo, which is used by order3.icn. You can rightly guess that order3.icn should be compiled after order1.icn and order2.icn, but does it matter which of them is compiled first? If order2.icn is compiled first, Unicon's database does not know symbol first is part of the package, and does not mangle the name; if you compile these files out of order you will get a runtime error.

The brute force solutions you have available to you are: to always place all of a package in the same source file, or to compile the files twice. Neither of these options is especially appealing. The symbol references in each package's files form a graph of dependencies on the other files in the same package. As long as this graph is acyclic, a correct order can be calculated. Unidep is a program that automates this task and generates a makefile specifying the dependencies in build rules. For example, given the program above, and the following makefile:

```
order: order1.u order2.u order3.u
```

```
  unicon -o order order1.u order2.u order3.u
%.u: %.icn
  unicon -c $*
```

Running the command "unidep order1.icn order2.icn order3.icn" will append the required additional dependencies. In this case these are:

```
order1.u: order1.icn
order2.u: order2.icn order1.u
order3.u: order3.icn order2.u
```

With these dependencies added, the makefile will compile the files in the correct order. You will want to add a rule to invoke Unidep from the makefile, and rerun it when your program changes significantly.

12.4 HTML documentation

Iplweb is an Icon documentation generator, inspired loosely by Java's JavaDoc program, and based on an HTML-generating program called iplref, by Justin Kolb. Iplweb depends on your program being in "IPL normal form", which is to say that comments in your source files should be in the format used in the Icon Program Library. From these comments and the signatures of procedures, methods, records, and classes, Iplweb generates reference documentation in HTML format.

This approach produces reference documentation automatically, without altering the original source files. Run Iplweb early, and run it often. It is common for reference documentation to diverge over time from the source code without such a tool. It is especially suitable for documenting the interfaces of procedure and class libraries. What it doesn't help with is the documentation of how something is implemented. It is designed primarily for the users, and not the maintainers, of library code.

12.5 Summary

Writing and maintaining large programs poses additional challenges not encountered when writing small programs. The need for design and documentation is greater, and the challenge of maintaining the correspondence between design documents and code is more difficult. Design patterns can help you with the design process, since they introduce easily recognizable idioms or sentences within a design. The more familiar the design is, the less cognitive load is imposed by the task of understanding it.

Packages have little or nothing to do with design patterns, but they are just as valuable in reducing the cognitive load required to work with a large program. Packages are not

just a source code construct. They actually do play a prominent role in software design notations such as UML. From a programmer's point of view, packages protect a set of names so that their associated code is more reusable, without fear of conflicts from other reusable code libraries or the application code itself.

Chapter 13

Use Cases and Supplemental UML Diagrams

When starting a new software project, it is tempting to begin coding immediately. An advocate of stepwise refinement starts with the procedure main() that every program has, and grows the program gradually by elaboration from that point. For complex systems, a software designer should do more planning than this. Chapters 9 and 10 covered the basics of class diagramming, an activity that allows you to plan out your data structures and their interrelationships.

The hard part about class diagramming is figuring out what information will need to be stored in attributes, and what object behavior will need to be implemented by methods. For many large projects there are basic questions about what the program is supposed to do that must be answered before these details about the application's classes can be determined. In addition, class diagrams depict static information but model nothing about the system that involves changes over time.

This chapter discusses some UML diagramming techniques that are useful before you start coding. They can help you figure out the details that belong in your class diagrams, by modeling dynamic aspects of your application's behavior. When you are finished with this chapter you will know how to:

- Draw use case diagrams that show the relationships between different kinds of users and the tasks for which they will use the software.
- Describe the details of use cases that define an application's tasks.
- Draw statechart diagrams that depict an object's behavior as states and transitions between states that model the dynamic aspects of the application.
- Specify conditions and activities that occur when an event causes an object to change its state.

- Draw collaboration diagrams that illustrate dynamic interactions between groups of objects.

13.1 Use Cases

A *use case* is an individual task. It defines a unit of functionality that the software enables one or more users to carry out. Sometimes it is a challenge to figure out what makes a reasonable "unit of functionality" in an application where long sequences of complex tasks are performed. Should the use cases correspond to small units such as individual user actions such as mouse clicks, or longer jobs such as updating a spreadsheet? One way to identify the appropriate units of functionality is to ask, for any given user action, whether it completes a change to the state of the application data. If the user would likely want to be able to save their work afterwards, the task is large enough to constitute a use case.

A diagram showing all the use cases helps early on in development to identify the overall scope and functionality of the software system as seen from the outside. The components of a use case diagram are depicted in Figure 13-1.

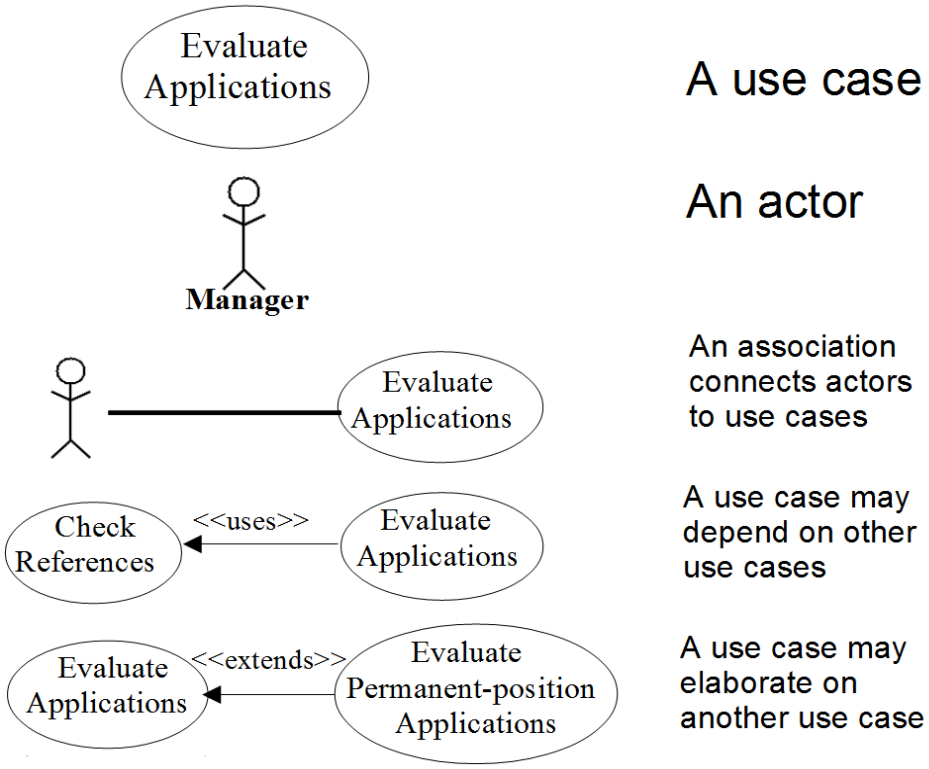


Figure 13-1: The main components of use case diagrams

The *use cases* themselves are shown as ovals. The name of the use case is inside the oval. The use cases have an accompanying description; an example description is given in

the next section. A use case is not represented in software by a class, but rather in the logic of the program's control flow. A use case relates several otherwise unassociated objects for a limited time to accomplish a particular task.

The term *actor* denotes both human users and external hardware or software systems that interact with the software system under design. Actors are shown in use case diagrams as stick figures. Each stick figure in the diagram represents a different kind of actor that interacts with the system during one or more use cases. The name of the role is written under the stick figure. An actor is really just a special kind of class that represents an external, asynchronous entity.

The *associations* between use cases and the actors that perform those tasks are drawn as plain lines. A use case may be performed by one or several actors. Use case associations identify the actors that participate in each use case. They are only slightly related to the associations between classes found in class diagrams.

Dependencies and *elaborations* between use cases are drawn as lines with arrows, annotated with a label between « and ». Some use cases use other use cases as part of a more complex task. Other use cases are defined as extensions of another use case.

Use case diagrams

A use case diagram consists of a set of use case ovals, bordered by a rectangle that signifies the extent of the software system. Actors are drawn outside the rectangle, with connecting lines to those use cases in which they participate. When some actors are non-human external systems, by convention the human actors are depicted on the left, and the non-humans go on the right.

An example use case diagram is shown in Figure 13-2, which depicts a recruiting management system. The manager hiring a new employee may interact with the company's legal department to produce an acceptable position advertisement. Many applicants might apply for a given position. The manager evaluates applications, possibly interviewing several candidates. When a candidate is selected, the manager interacts with the legal department to make a job offer.

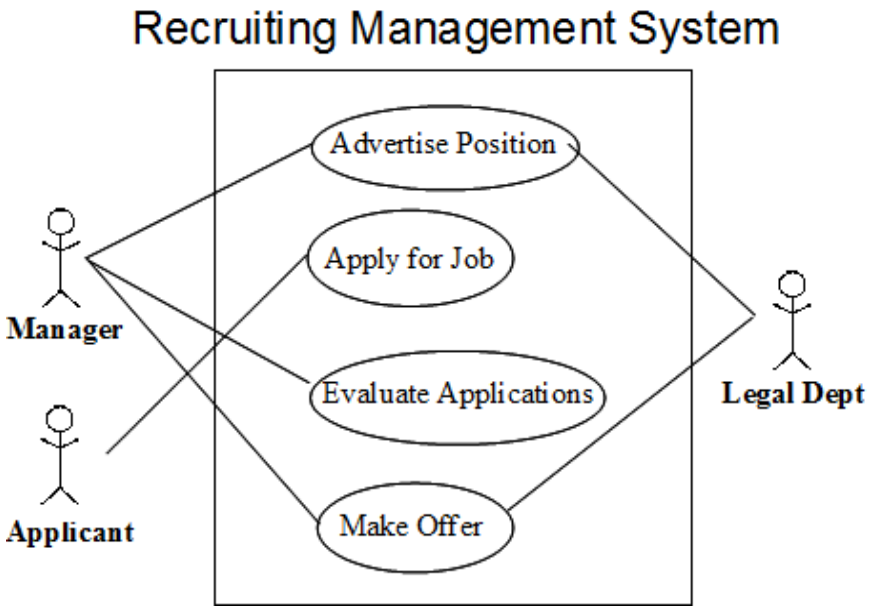


Figure 13-2 A Use Case Diagram

Use case descriptions

The details of each use case are specified in a related *use case description*. This description may include prose text, such as the following description of the "Make Offer" use case:

Make Offer is started by the manager when an applicant has been selected from among the candidates for a position. The manager obtains approval from the legal department, commits necessary budget resources, and generates an offer letter with details on salary, benefits, and the time frame in which a decision is required.

The use case description may be organized into fields, or more detailed than this. For example, one field might consist of the most common sequence of events, emphasized by an explicit enumeration. The common variations on the primary event sequence are also of value. A more organized description of the Make Offer use case might be

Make Offer **Initiated**: by manager, after candidate for a position has been selected.

Terminates: when the candidate receives the offer in writing.

Sequence:

1. Manager obtains approval from legal department.
2. Manager commits resources from budget
3. Manager telephones candidate with offer
4. Manager generates offer letter
5. Offer letter is express mailed to candidate.

Alternatives:

In step 2, Manager may request extra non-budgeted resources.

In step 3, Manager may fax or e-mail offer in lieu of telephone.

13.2 Statechart Diagrams

Statecharts are diagrams that depict finite state machines. A finite state machine is a set of states, drawn as circles or ovals, plus a set of transitions, drawn as lines that connect states. Statecharts generally have an *initial state*, which may be specially designated by a small, solid circle, and one or more *final states*, which are marked by double rings.

In object modeling, states represent the values of one or more attributes within an object. Transitions define the circumstances or events that cause one state to change to another. Statecharts are a tool for describing allowable sequences of user interactions more precisely than is captured by use cases. Discovering the events that cause transitions between states, as well as the conditions and actions associated with them, helps the software designer to define the required set of operations for classes.

Figure 13-3 shows an example statechart diagram for a real estate application. A house enters the FORSALE state when a listing agreement is signed. The house could leave the FORSALE state with a successful offer at the listed price (entering a REVIEW period) or by utter failure (if the listing agreement expires), but the most common occurrence is for a buyer to make an offer that is less than the asking price. In that case, a NEGOTIATION state is entered, which may iterate indefinitely, terminating when either the buyer or seller agrees to the other party's offer or walks away from the discussion. When an offer is accepted, a PENDING period is entered in which financing is arranged and inspections and walkthroughs are performed; this period is terminated when escrow is closed, title is transferred, and the house is SOLD.

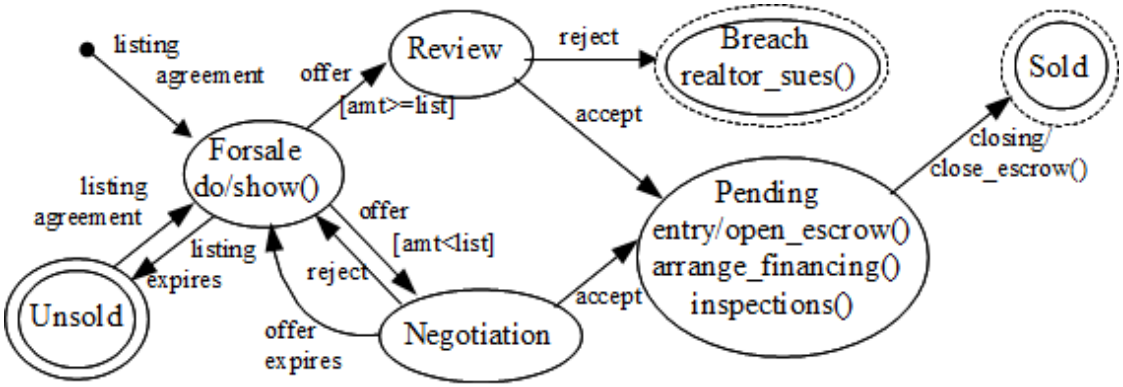


Figure 13-3: A Statechart Diagram

Since a state represents the values of one or more attributes within an object, a transition coincides with assignments that alter those attributes' values. The purpose of the diagram is to identify when and why those values change, in terms of the application domain.

Events and conditions

Most transitions in a statechart are triggered by an *event*. In Figure 13-3 the events were things like "offer" and "closing". Typically, an event describes an asynchronous communication received from another object. An event is instantaneous, while a state corresponds to some possibly lengthy interval of time until an object transitions into some other state. From the point of view of the object being modeled in the statechart, the event is an interrupt that affects the object's behavior. Such an event would normally be implemented by defining a method for the object with a name derived from the event name.

It is common during modeling to have a transition that can only occur if a Boolean condition is satisfied. In Figure 13-3, the event offer was used for two transitions out of the same state, with different conditions (amount >= list price versus amount < list price) to determine which transition would be taken. In statechart diagrams, conditions are given after the event name, in square brackets, as in [amt < list].

For a condition on a transition, it might make sense for that transition to require no trigger event at all. The transition would occur immediately if the condition were ever satisfied. Such a constraint-based transition would potentially introduce condition tests at every point in the object's code where the condition could become true, such as after each assignment to a variable referenced in the condition. This may work in special cases, but poses efficiency problems in general. Transitions without trigger events make sense in one other situation. If a state exits when a particular computation completes, you can use a triggerless transition to the new state that the object will be in when it is finished with the job it is performing in the current state.

Actions and activities

Events are not the only class methods that are commonly introduced in statecharts. In addition to a condition, each event can have an associated *action*. An action is a method that is called when the event occurs. Since events are instantaneous, action methods should be of bounded duration. Similarly, states can have a whole regalia of related methods called *activities*. There are activities that are called when a state is entered or exited, respectively. The most common type of activity is a method that executes continuously as long as the object is in that state. If more than one such activity is present, the object has internal concurrency within that particular state.

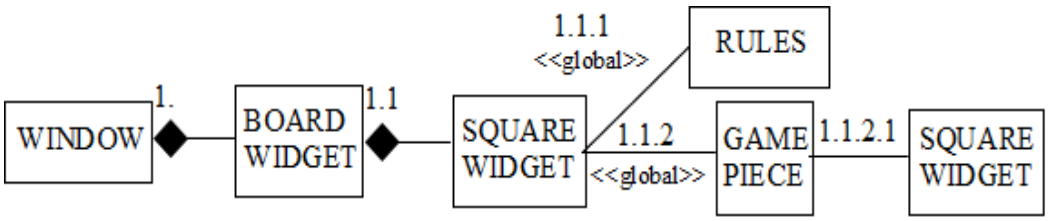
In statechart diagrams, actions are indicated by appending a slash (/) and an action after the event name and any condition. Activities are listed within the state oval. If a keyword and a slash prefix the activity, special semantics are indicated. For example, the do keyword indicates repeated activity. In Figure 13-3, the activity do / show() says that the house will be shown repeatedly while it is in the FORSALE state. The activity entry / open_escrow() indicates that the method open_escrow() is called on entry to the PENDING state, after which inspections() and arrange_financing() activities are performed.

13.3 Collaboration Diagrams

Statecharts normally model the state of one object. They show how the object reacts to events that come from the other objects in the system, but do not depict where those events came from. In a complex system, it is useful to understand the interactions among many objects. An event that changes one object's state may trigger events in many other objects, or a group of objects may trigger events in one another in a cyclic fashion.

Collaboration diagrams show such interactions between objects. They are drawn similarly to class diagrams. A group of rectangles are drawn to represent instances of classes, and lines depict the relationships between those classes. But while a class diagram emphasizes the static structures, representing details such as class attributes, and association multiplicity, a collaboration diagram depicts a specific sequence of messages sent from object to object during the completion of some task. The messages are annotated alongside the links between objects to indicate sender and recipient, and numbered to show both the sequence and the tree structure of the nested messages. In the general case more than one message number can be annotated for a given link, since multiple messages may be transmitted between the same objects in the course of completing the use case.

Figure 13-4 shows an example collaboration diagram. This particular collaboration illustrates the input processing of a user event in a game application in which pieces are moved about a board, such as chess or checkers. The incoming event is sent as a message from the window object to the board widget (message 1). The board widget uses its layout to map mouse (x, y) coordinates onto a particular square to which the user is moving the currently selected piece, and forwards a message to that square (1.1). The square sends a message to a rules object, which checks the validity of the user's move (1.1.1), and if the move is legal, the square sends a message to the game piece, effectively telling it to move itself (1.1.2). The game piece sends an "erase" message to the square where it was formerly located (1.1.2.1) before changing its link to refer to the square to which it is moving.



1. user mouse click
1.1 square select
1.1.1 legal move check
1.1.2 update position
1.1.2.1 erase (at old position)

Figure 13-4: A Collaboration Diagram

There are a couple more annotations worth noting in Figure 13-4. The links between window, board widget, and square widget are identified as aggregations since they denote

geometric containment; this information is redundant with the class diagram, but is given to explain how the objects are linked to allow message transmission. The connections between the square widget and the rules and game piece objects are marked as <<global>> to indicate that the square widget obtains references to these objects from global variables. The link between the game piece and the square widget in which it is located is a regular association and does not require further annotation. Besides <<global>> you can annotate a link as a <<parameter>> or <<local>> to indicate other non-association references through which messages are transmitted.

13.4 Summary

This chapter introduced four UML diagram types that are useful in modeling dynamic aspects of a program's behavior. To learn more about these techniques and others, consult a primary UML resource, such as *The Unified Modeling Language User Guide*, by Grady Booch, James Rumbaugh, and Ivar Jacobson.

No one technique is a complete solution, but some combination of use cases, statecharts, and collaboration diagrams will allow you to sufficiently model most applications. Use cases are particularly valuable for describing tasks from the point of view of the application domain and human user. Statecharts are good for modeling event-based systems such as user interfaces or distributed network applications. Collaboration diagrams describe interactions between objects that allow you to model the big picture in a complex system.

In terms of primacy and chronological order, for most applications you should start with use cases and try to develop them completely. For those use cases that seem complex, or for which the conventional use case description seems inadequate, you can then bring in statecharts or collaboration diagrams to assist in completing an understandable design.

Class diagrams are the backbone of a detailed object oriented design. They can be developed by extracting details from the other kinds of diagrams, and should reflect programmers' understanding of the application domain for which the software is being written.

Part III

Example Applications

Chapter 14

CGI Scripts

CGI scripts are programs that read input forms and generate dynamic HTML content for the World Wide Web. CGI programs are often written in scripting languages, but they can be written in any language, such as C. Unicon is ideal for writing CGI scripts, since it has extraordinary support for string processing. In this chapter you will learn how to

- Construct programs whose input comes from a web server.
- Process user input obtained from fields in HTML forms
- Generate HTML output from your Icon programs

14.1 Introduction to CGI

The Common Gateway Interface, or CGI, defines the means by which Web servers interact with external programs that assist in processing Web input and output. CGI scripts are programs that are invoked by a Web server to process input data from a user, or provide users with pages of dynamically generated content, as opposed to static content found in HTML files. The primary reference documentation on CGI is available on the Web from the National Center for Supercomputer Applications (NCSA) at http://hoohoo.ncsa.uiuc.edu/cgi/. If you need a gentler treatment than the official reference, *The CGI Book*, by Bill Weinman, is a good book on CGI. Although other methods for writing web applications on the server have been developed, CGI is the most general, portable method and is likely to remain in wide use for some time.

This chapter describes cgi.icn, a library of procedures for writing CGI scripts. The cgi.icn library consists of a number of procedures to simplify CGI input processing and especially the generation of HTML-tagged output from various data structures. The cgi.icn reference documentation can be found in Appendix B, which describes many important modules in the Icon Program Library.

Note

To use cgi.icn, place the statement link cgi at the top of your program.

CGI programs use the hypertext markup language HTML as their output format for communicating with the user through a Web browser. Consequently, this chapter assumes you can cope with HTML, which is beyond the scope of this book. HTML is an ASCII format that mixes plain text with *tags* consisting of names enclosed in angle brackets such as <HTML>. HTML defines many tags. A few common tags will be defined where they occur in the examples. Most tags occur in pairs that mark the beginning and end of some structure in the document. End tags have a slash character preceding the name, as in </FONT>. More details on HTML are available from the World Wide Web Consortium at http://www.w3.org/MarkUp/.

Organization of a CGI script

CGI programs are very simple. They process input data supplied by the Web browser that invoked the script (if any), and then write a new Web page, in HTML, to their standard output. When you use cgi.icn the input-processing phase is automatically completed before control is passed to your program, which is organized around the HTML code that you generate in response to the user. In fact, cgi.icn includes a main() procedure that processes the input and writes HTML header and tail information around your program's output. For this reason, when you use cgi.icn, you must call your main procedure cgimain().

Processing input

The HTTP protocol includes two ways to invoke a CGI program, with different methods of supplying user input, either from the standard input or from a QUERY_STRING environment variable. In either case, the input is organized as a set of fields that were given names in the HTML code from which the CGI program was invoked. For example, an HTML form might include a tag such as:

```
<INPUT TYPE = "text" NAME = "PHONE" SIZE=15>
```

which allows input of a string of length up to 15 characters into a field named PHONE.

After the CGI library processes the input, it provides applications with the various fields from the input form in a single table, which is a global variable named cgi. The keys of this table are exactly the names given in the HTML INPUT tags. The values accessed from the keys are the string values supplied by the user. For example, to access the PHONE field from the above example, the application could write

```
cgi["PHONE"]
```

Processing output

The main task of the CGI program is to write an HTML page to its standard output, and for this task cgi.icn provides a host of procedures. Typically these procedures convert a structure value into a string, wrapped with an appropriate HTML tag to format it properly. A typical example is the library procedure cgiSelect(name,values), which writes an HTML SELECT tag for a field named name. The SELECT tag creates a list of radio buttons on an HTML form whose labels are given by a list of strings in the second parameter to cgiSelect(). A programmer might write

```
cgiSelect("GENDER", ["female", "male"])
```

to generate the HTML

```
<SELECT NAME="GENDER">
<OPTION SELECTED>female
<OPTION>male
</SELECT>
```

Common CGI environment variables

The official CGI definition includes a set of standard environment variables that are set by the Web server as a method of passing information to the CGI script. Programmers access these environment variables using getenv(), as in

```
getenv("REMOTE_HOST")
```

Table 14-1 presents a summary of the CGI environment variables as a convenience so that this book can serve as a stand-alone reference for writing most CGI scripts. For a complete listing of all the environment variables supported by CGI go to http://hoohoo.ncsa.uiuc.edu/cgi/env.html on the Internet.

Table 14-1

CGI Environment Variables

Variable	Explanation
CONTENT_LENGTH	The length of the ASCII string provided by method="POST".
HTTP_USER_AGENT	The user's browser software and proxy gateway, if any. The format is *name/version*, but varies wildly.
QUERY_STRING	The information submitted through the form, which follows the ? in the URL when using method="GET". QUERY_STRING data is parsed and inserted into a table stored in the global variable cgi, so cgi.icn scripts do not generally consult this environment variable.
REMOTE_ADDR	The IP address of the client machine.
REMOTE_HOST	The hostname of the client machine. Defaults to IP held by REMOTE_ADDR.
REQUEST_METHOD	The method (GET or POST) used to invoke the CGI script.
SERVER_NAME	The server's hostname. It defaults to the IP address.
SERVER_SOFTWARE	The Web server that invoked the CGI script. The format is *name/version*.

14.2 The CGI Execution Environment

CGI scripts do not execute as stand-alone programs and aren't launched from a command line; a Web server executes them. The details of this are necessarily dependent on the operating system and Web server combination in use. The following examples are based on a typical UNIX Apache server installation in which users' HTML files are located under $HOME/public_html. Check with your system administrator or Web server documentation for the specific filenames, directories, and permissions required to execute scripts from your Web server. Some web servers do not allow scripts at all, and most others run scripts with a special userid in a limited/protected file system where absolute pathnames are different from how you see them.

Under Apache, you need a directory under $HOME/public_html named cgi-bin. Both $HOME/public_html and its cgi-bin subdirectory should have "group" and "other" permissions set to allow reading and executing for the Web server to run the programs you place there. Do not give anyone but yourself write permissions! The following commands set things up on a typical Apache system. The percent sign (%) is not part of the command; it is the UNIX shell prompt. The period in the final command is part of the command and refers to the current working directory.

```
% mkdir $HOME/public_html
% cd $HOME/public_html
% mkdir cgi-bin
% chmod go+rx . cgi-bin
```

The next two example files will allow you to verify that your directories and permissions

are correct for your Web server. Despite all the attempts to make the world's web servers secure, the only security you can count on is your own. From security expert David A. Gamey we have the following tips:

- Use no data without checking for validity. Even HTTP header data can be wrong. If you expect a number, make sure the supplied data is a number.
- It is a very bad idea to make any system() calls (or open piped commands, etc.) from scripts. If you absolutely have to run something external, construct command strings yourself or fully parse user data being used to check for command separators and hidden commands, etc.
- Don't rely on looking for known bad characters; restrict input to known good characters. Use known good values such as those selected from list boxes
- Sensitive data should be sent using POST, not GET.
- Check for and prevent file system attacks, such as paths including .. in them.
- Log everything received by your script, so you can tell when attacks occur.

14.3 An Example HTML Form

CGI scripts are typically invoked from HTML pages. When you view the following example page in your browser, it should look something like the one shown in Figure 14-1. For this test, create an HTML form $HOME/public_html/simple.html containing Listing 14-1. When you have a CGI script compiled and ready to run, you can edit the URL in this file to point at your CGI program, the simple.cgi executable.

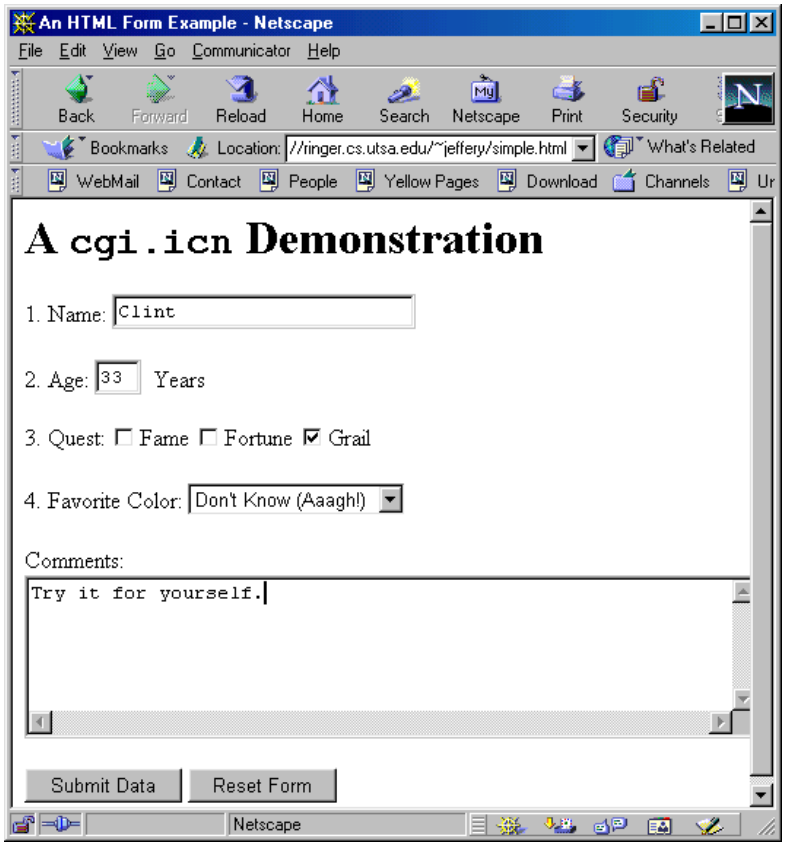


Figure 14-1: An HTML Form Example

Listing 14-1 An HTML form

```
<HTML><HEAD><title> An HTML Form Example </title></HEAD>
<BODY>
<h1> A <tt>cgi.icn</tt> Demonstration</h1>
<form method="GET"
      action="http://www.cs.uidaho.edu/~jeffery/cgi-bin/simple.cgi">
1. Name: <input type="text" name="name" size=25> <p>
2. Age:  <input type="text" name="age" size=3>  Years <p>
3. Quest:
<input type="checkbox" name="fame">Fame</input>
<input type="checkbox" name="fortune">Fortune</input>
<input type="checkbox" name="grail">Grail</input><p>
4. Favorite Color:
<select name="color">
<option>Red
<option>Green
<option>Blue
<option selected>Don't Know (Aaagh!)
</select><p>
Comments:<br>
```

```
<textarea rows=5 cols=60 name="comments"></textarea><p>
<input type="submit" value="Submit Data">
<input type="reset" value="Reset Form">
</form>
</BODY>
</HTML>
```

14.4 An Example CGI Script: Echoing the User's Input

The following script, named simple.cgi might be invoked from the FORM tag above. The simple.cgi script is produced from an Unicon source file, simple.icn, that you can copy from the book web site (http://unicon.org/book/). This program needs to be compiled with the command

```
unicon −o simple.cgi simple.icn
```

Many Web servers are configured so that CGI scripts must end with the extension .cgi. Check with your system administrator about CGI naming conventions if the .cgi extension does not work for you. In addition to the web server being configured to allow user invocation, unless you use compiler option -B to bundle the virtual machine into your executable, your program must be able to find and execute the virtual machine from whatever user id CGI's are executed.

The program reads the form input specified in simple.html, and writes it back out. All cgiEcho() is doing in this case is adding an HTML newline tag after each call. If you look it up in Appendix B, you will find that it will copy its arguments to both the HTML output and a log file if given a file as its first argument.

```
link cgi
procedure cgimain()
    cgiEcho("Hello, ", cgi["name"], "!")
    cgiEcho("Are you really ", cgi["age"], " years old?")
    cgiEcho("You seek: ", cgi["fame"]==="on" & "fame")
    cgiEcho("You seek: ", cgi["fortune"]==="on" & "fortune")
    cgiEcho("You seek: ", cgi["grail"]==="on" & "grail")
    cgiEcho("Your favorite color is: ", cgi["color"])
    cgiEcho("Your comments: ", cgi["comments"])
end
```

Generating an output page that rehashes the user's input is a good test of your HTML form before you deploy it with a CGI script that actually does something. In some cases, it is also helpful in allowing the user to recheck their submitted input and confirm or cancel before acting on it.

14.5 Debugging CGI Programs

CGI programs can be a pain to debug. You may have to debug your CGI execution environment, before you can even start debugging your CGI program itself. If your CGI script returns an "internal server error", or no output at all, you may have file permissions wrong, or the CGI script may not be able to find the Unicon virtual machine in order to run the program. Some web servers execute CGI scripts under a special userid such as "web", others will run them under your user id. Some web servers run CGI scripts under a protected file system where the root directory "/" is not the same as the root directory visible to your user account, so the path to iconx that you normally use may be invalid in your CGI program. CGI scripts may have a very limited PATH for security reasons, not the PATH you set for your user account. Your best bet is probably to use the -B Unicon compiler option to bundle the Unicon interpreter into your executable file; alternatively you can probably copy the virtual machine "iconx" into your cgi-bin directory

Debugging your CGI program itself may require special tricks. Because your CGI program is executed by a web server, its standard error output may not be visible to you. You can try to redirect error output to standard out, but your error output may not be readable unless it is converted into HTML (say, by adding
 at each newline). One way to accomplish this is to write *two* programs: one that performs the primary task, and a second program that calls the first one, catches any error messages, and converts any plain text output to HTML.

14.6 Appform: An Online Scholarship Application

The next example, appform.icn, is a CGI script for an on-line scholarship application that was used at a university. Its structure is similar to the previous example, with a twist: the user input is printed for the convenience of the scholarship administrators. As a backup, the CGI script also e-mails the application to the scholarship administrator. This is useful if the print job fails for some reason. The program is a single cgimain() procedure, which starts by processing each of the user input fields. The program then opens a temporary file with a .txt extension, and writes a nicely formatted document containing the user's scholarship application information.

The code for Appform is shown in Listing 14-2. To run it you must adapt it to your environment. As written, it prints to lpr, and sends mail to jeffery@cs.uidaho.edu. When running a CGI script it is important to realize you will run in a different directory, and with different user id and PATH environment, than your regular account. The program runs with whatever user id and permissions the system administrator assigns the Web server process. For example, its root (/) directory may not be at the root of your regular filesystem, so absolute paths may not work.

Listing 14-2

An online application form

```
############################################################
#       File:    appform.icn
#       Subject: CGI program to process scholarship applications
#       Author:  Clinton Jeffery
#       Date:    July 11, 2002
############################################################
# This program processes a bunch of input fields defined in an on-line schol-
# arship application at http://unicon.org/book/appform.html and from them,
# generates a text file, prints and e-mails it to the scholarship coordinator.
############################################################

link cgi, io
$define ADMIN "jeffery@cs.uidaho.edu"
procedure cgimain()
  fname := tempname("appform", ".txt", "/tmp")
  f := open(fname, "w") | stop("can't open ", fname)
  write("Generating typeset copy of application form...")
  write(f,"Scholarship Program Application\n")
  write(f, "Name: ", cgi["NAME"], "\t\t Phone: ", cgi["PHONE"])
  write(f, "Address: ", cgi["ADDRESS1"], ' \t\t Social Sec. Number: ", cgi["SOC"])
  write(f, cgi["ADDRESS2"], " \t\t Gender (M/F): ",cgi["GENDER"], "\n")
  write(f,"Semester hours completed: ", cgi["CREDITS"])
  write(f,"College GPA:  Overall ", cgi["GPA"])
  write(f,"Present Employer: ", cgi["EMPLOYER"])
  write(f,"Position: ", cgi["POSITION"], " Hours/week: ", cgi["HOURS"])
  write(f,"Educational Background")
  write(f,"High School: name, year, GPA, graduated?")
  write(f, cgi["HIGH1"], "\n", cgi["HIGH2"])
  write(f,"For each college, list name, dates attended, credit hours,")
  write(f,"degrees awarded", cgi["COLL1"], "\n", cgi["COLL2"], "\n\n")
  write(f,"Academic honors, scholarships, and fellowships")
  write(f,cgi["HONOR1"], "\n", cgi["HONOR2"], "\n")
  write(f,"Extracurricular interests:", cgi["EXTRA1"], "\n", cgi["EXTRA2"])
  write(f,"Professional organizations:", cgi["ORGS1"], "\n", cgi["ORGS2"])
  write(f,"Research interests:")
  write(f,cgi["RESEARCH1"], "\n", cgi["RESEARCH2"])
  write(f,"Name(s) of at least one person you have asked to")
  write(f,"write an academic reference letter.")
  write(f,"Name          Address        Relationship")
  write(f,cgi["REF1"], "\t", cgi["REFADD1"], "\t",cgi["REFREL1"])
  write(f,cgi["REF2"], "\t", cgi["REFADD2"], "\t",cgi["REFREL2"])
  write(f,"\nI certify that information provided on this")
  write(f,"application is correct and complete to my knowledge.\n")
```

```
  write(f,"Signature: ", repl("_", 60), "\n Date: ", repl("_", 60), "\n\^l\n")
  write(f,"Please write a short statement of purpose, including")
  write(f,"information about your background, major, and career")
  write(f,"interests, and professional goals.\n")
  write(f, cgi["INFO"])
  close(f)
  write("Mailing form to program director...")
  f := open(fname)
  m := open("mailto:" || ADMIN, "m", "Subject: appform")
  while write(m, read(f))
  close(m)
  close(f)
  write("Printing hard copy...")
  system("lpr " || fname || "; rm " || fname)
  cgiEcho("Thank you for applying, ", cgi["NAME"])
  cgiEcho("Your application has been submitted to " || ADMIN)
end
```

Summary

Writing CGI scripts in Unicon is easy. The input fields are handed to you elegantly in a global variable, and library functions allow you to write terse code that generates correct HTML output. The only thing certain about the fast-changing Internet standards is that they will get continually more complex at a rapid pace. CGI scripting is no substitute for JavaScript, XML, or any newer buzzword that may be hot this week. But it is a lasting, multi-platform standard for how to run a program on a Web server from a browser, and it may be the simplest and best solution for many Internet applications for some time.

Chapter 15

System and Administration Tools

In an open computing environment, users build their own tools to extend the capabilities provided by the system. Unicon is an excellent language for programmers who wish to control and extend their own system. This chapter presents techniques used to write several system utilities of interest to general users as well as system administrators. Best of all, many of these utilities work across multiple platforms, thanks to Unicon's high degree of system portability. You will see examples of

- Traversing and examining directories and their contents.
- Finding duplicate files.
- Implementing a quota system for disk usage.
- Doing your own custom backups.
- Capturing the results of a command-line session in a file.

15.1 Searching for Files

To begin, consider a simple problem: that of finding a file whose name matches a specified pattern. Regular expressions are commonly used to describe the patterns to match, so you may want to link in the regular expression library. Here is the start of a file-search application.

```
#
# search.icn
#
# Search for files whose (entire) names match a pattern given
# as a regular expression
#
# Usage: ./search <pattern> [dirs]
```

```
link regexp
```

The application starts by processing the command-line arguments. There must be at least one argument: the pattern to search for. Arguments following that one are directories to search. If no directories are specified, you can use the current working directory. The procedure findfile() performs the actual task of searching for the files:

```
procedure main(args)
    (*args > 0) | stop("Usage: search <pattern> [directories]")
    pattern := pop(args)
    if *args = 0 then findfile(".", pattern)
    else
        every dir := !args do findfile(dir, pattern)
    exit(0)
end
```

The search algorithm is a depth-first search. In each directory, check the type of each file. If you find a directory, make a recursive call to findfile(). But before you do that, check to see if the name of the file matches the pattern.

For efficiency, since the program uses the same regular expression for all searches, you can compile the pattern into a static variable called pat. The regular expression library allows you to perform this compilation once, the first time that findfile() is called, and then reuse it in subsequent calls.

```
procedure findfile(dir, pattern)
    local d, f, s
    static pat
    initial {
        pat := RePat(pattern) | stop("Invalid pattern ", image(pattern))
        }
    d := open(dir) | {
      write(&errout, "Couldn't access ", dir, " ", &errortext)
      return
      }
```

While you read the names of the files in the directory, be sure to not go into the special entries "." and ".." that represent the current directory and the parent directory, respectively. Except for these names, the directory hierarchy is a tree so you don't need to check for cycles. Some file systems support the concept of links, described in Chapter 5; links can introduce cycles, so the code below recursively calls itself only on regular directories, not on links.

```
    while name := read(d) do {
      if name == ("." | "..") then next
      f := dir || "/" || name
      s := stat(f) | {
        write(&errout, "Couldn't stat ", f, " ", &errortext)
        next
        }
```

Here is the check of the file name against the pattern:

```
      name ? if tab(ReMatch(pat)) & pos(0) then write(f)
```

Note

Regular expressions do not use the same notation as file-matching wildcards used on the command line. The regular expression notation used by RePat() and ReMatch() is given in the documentation for the regexp module in Appendix B.

Finally, if f is the name of a directory, make the recursive call. Note that since the pattern has already been compiled and stored in a static variable, you don't need to pass it in as a parameter for the recursive call.

```
      if s.mode[1] == "d" then findfile(f)
      }
    close(d)
    close(d)
end
```

This is a very simple demonstration of some systems programming techniques for Unicon. You will see this sort of depth-first traversal of the file system again, in the section on file system backups later in this chapter.

15.2 Finding Duplicate Files

An interesting variation on the previous program is to find files whose contents are identical. This is valuable for those of us who make many copies of files in various subdirectories over time, and then forget which ones are changed. Since this task deals with lots of files, there are some things to think about. Reading a file is an expensive operation, so you should try to minimize the files you read. Since you can find the length of a file without reading it, you can use that to perform the first cut: you won't need to compare files of different sizes. The first step, then, is to solve the simpler problem of finding files that have the same size.

The previous program example shows how to traverse the directory structure. For the lengths you can use a table - with each possible length, store the names of the files of that length. Since there are lots of files, try to be smart about what you store in the table. The natural structure is a list. This leads to the following code:

```
procedure scan(dir)
    f := open(dir) | {
        write(&errout, "Couldn't access ", dir, " ", &errortext)
        return
        }
    while name := read(f) do {
        filename := dir || "/" || name
        r := stat(filename)
        case r.mode[1] of {
            "-" : {
                /lengths[r.size] := list()
                push(lengths[r.size], filename)
                }
            "d" : name == ("." | "..") | scan(filename)
            }
        }
    close(f)
end
```

The main program scans all the directories, and for each list, compares all the files in it to each other. The global table named lengths maps lengths to filenames.

```
global lengths
procedure main()
    lengths := table()
    scan("/.")
    every l := !lengths do {
        if *l = 1 then next
        find_dups(l)
        }
end
```

For example, if in a directory there are files A, B, C, and D with lengths of 1, 2, 5, and 2 bytes, respectively, the lengths table will contain the following:

```
lengths[1] === [ "A" ]
lengths[2] === [ "B", "D" ]
lengths[5] === [ "C" ]
```

If a list only has one element, there is no reason to call the function to compare all the elements together.

All of this makes sense, but in many cases there will be only one file that has a certain size. Creating a list for each file size is a small waste of space. What if, for the first entry, you only store the name of the file in the table? Then if you get a second file of the same size, you can convert the table entry to store a list. That is, in the above example you could have

```
lengths[1] === "A"
lengths[2] === [ "B", "D" ]
lengths[5] === "C"
```

Now for most of the files, the program is only storing a string, and it creates a list only where it needs one. You can say that the table is *heterogeneous* if you want to get technical about how its elements are a mixture of strings and lists. With this change, the main procedure becomes:

```
global lengths
procedure main()
    lengths := table()
    scan("/")
    every l := !lengths do {
      if type(l) == "string" then next
      find_dups(l)
      }
end
```

Instead of checking to see if the list has only one element, the code checks to see if the value from the table is a string, and ignores those entries.

The scan procedure has to do a little more work. Instead of initializing the value to a list, you can use the name of the current file; if the value already in the table is a string, create a list and add both the name from the table and the name of the current file to the list. If the value in the table is a list already, then you can just add the current filename to it.

```
while name := read(f) do {
      filename := dir || "/" || name
      r := stat(filename)
      case r.mode[1] of {
        "-" :
          case type(lengths[r.size]) of {
            "null" : lengths[r.size] := filename
            "string" : {
              lengths[r.size] := [lengths[r.size]]
              push(lengths[r.size], filename)
            }
            "list" : push(lengths[r.size], filename)
          }
        "d" : name == ("." | "..") | scan(filename)
      }
    }
```

To compare two files together, you will of course have to read both files. One way to do it would be to read in the files to memory and compare them, but that would take a lot of space. Most files even when they have the same size will probably be different; you only need to read the files until you find a difference. At the same time, you shouldn't read the files one byte at a time, since the I/O system is optimized to read larger chunks at a time. The chunk size to use will depend on the exact configuration of the computer the program is running on, that is, the speed of the file reads compared to the CPU speed and the available RAM storage.

The actual comparison is simple: keep reading chunks of both files, failing if you find a difference and succeeding if you reach the ends of the files without finding any.

```
procedure compare(file1, file2)
    static maxline
    initial maxline := 1000
    f1 := open(file1) | fail
    f2 := open(file2) | { close(f1); fail }
    while l1 := reads(f1, maxline) do {
      l2 := reads(f2, maxline)
      if l1 ~== l2 then {
        every close(f1 | f2)
        fail
        }
      }
    every close(f1 | f2)
    return
end
```

One technique that is sometimes used for comparing long strings and so forth is *hashing*. To use hashing, you define a function that computes an integer from the string. If the hash values of two strings are different, you know that they cannot be the same. However hash values being equal doesn't necessarily imply that the strings are equal, so in the worst case scenario you still have to compare the two strings. If you aren't familiar with hashing, we encourage you to consult a book on algorithms to learn more about this technique and think about how the program may be further improved with it. One example hash function used internally by Unicon is equivalent to:

```
procedure hash(s)
    local i := 0
    every i +:= ord(s[1 to min(*s, 10)]) do
    i *:= 37
    i +:= *s
    return i
end
```

To complete the application, one piece remains: given a list of filenames, you need to go through them all and make all the possible comparisons. This can be done with a simple loop, calling compare() to perform the actual comparisons.

```
procedure find_dups(l)
    every i := 1 to *l do {
      f1 := l[i]
      every j := i+1 to *l do {
        if compare(f1, l[j]) then write(f1, " == ", l[j])
        }
      }
end
```

This is relatively inefficient: for a list of n files, it performs n(n-1)/2 comparisons and reads each file n-1 times. In certain situations it is possible to get by with not doing so many comparisons; we leave this as an exercise to the reader.

Tip

Clever use of hashing might let you get away with reading each file just once.

Listing 14-1 shows the complete program, with added comments, and also some error checking. If any directory or file open fails, it prints a message and proceeds with the processing.

Listing 14-1 A program for finding duplicate files.

```
#
# duplicate.icn
#
# Find files in the filesystem that are identical

global lengths
procedure main()
    lengths := table()
    # On some systems, a leading "//" in a filename may have
    # a different meaning, so we use "/." instead of just "/"
    scan("/.")
    every l := !lengths do {
      if type(l) == "string" then next
      find_dups(l)
      }
  exit(0)
end

# Scan all the directories and add files to the length map -
# the global table "lengths"
```

```
procedure scan(dir)
    f := open(dir) | {
      write(&errout, "Couldn't open ", dir, "; ", &errortext)
      fail
      }
    while name := read(f) do {
      filename := dir || "/" || name
      r := stat(filename) | {
        write(&errout, "Couldn't stat ", filename, "; ", &errortext)
        next
        }

      # A small optimisation: there are probably quite a few
      # zero-length files on the system; we ignore them all.
      r.size > 0 | next

      case r.mode[1] of {
        "-" :
          # ordinary file
          case type(lengths[r.size]) of {
            # if null, it's the first time; just store filename
            "null" : lengths[r.size] := filename

            # if string, one element already exists; create a list and make
            # sure we add both the old filename and the new one.
            "string" : {
              lengths[r.size] := [lengths[r.size]]
              push(lengths[r.size], filename)
              }
            # There's already a list; just add the filename
            "list" : push(lengths[r.size], filename)
          }
        "d" :
          # A directory. Make sure to not scan . or ..
          name == ("." | "..") | scan(filename)
      }
    }
    close(f)
    return ""
end

# Given a list of filenames, compare the contents of each with every other.
procedure find_dups(l)
    # This is O(n^2)
```

```
    every i := 1 to *l do {
      f1 := l[i]
      every j := i+1 to *l do {
        if compare(f1, l[j]) then write(f1, " == ", l[j])
        }
      }
end

# Compare two files; by reading in 1000 byte chunks. This value may need
# to be adjusted depending on I/O speed compared to CPU speed and memory.
procedure compare(file1, file2)
    static maxline
    initial maxline := 1000

    # are f1 and f2 identical?
    f1 := open(file1) | {
      write(&errout, "Couldn't open ", file1, "; ", &errortext)
      fail
      }
    f2 := open(file2) | {
      close(f1)
      write(&errout, "Couldn't open ", file2, "; ", &errortext)
      fail
      }
    while l1 := reads(f1, maxline) do {
      l2 := reads(f2, maxline) |
        # The files are supposed to be the same size! How could
        # we read from one but not the other?
        stop("Error reading ", file2)
      if l1 ~== l2 then {
        every close(f1 | f2)
        fail
        }
      }
    every close(f1 | f2)
    return
end
```

15.3 User File Quotas

Many computing platforms offer a filesystem quota facility, where each user has only so much of the disk to store files. The system does not allow files to grow once the limit has been reached. However, many systems don't have this facility, and on other systems

the available quota mechanism is not enabled because the user might have an urgent and immediate need to exceed his or her quota for a short time.

For these uses this section presents an alternate filesystem quota method. The quota for each user is stored in a file, and at regular intervals (perhaps overnight) the system examines the disk usage of each user. If it is above the quota, a message is sent. A summary message is also sent to the administrator so that any user that is over quota for more than a few days will be noticed.

First, declare a few global variables that will hold the strings that are to be used in the messages that are sent:

```
global complaint_part1, complaint_part2, summary_header

procedure main(args)
    # Read database of users; get disk usage for each user and
    # check against his/her quota
    db := "/usr/lib/quotas/userdb"
    administrator := "root"
    init_strings()
```

Calculating disk usage

The next step is to read the database of users, and for every directory, calculate the disk usage. On UNIX systems you can just run du (the UNIX tool that measures disk usage) and read its output from a pipe to obtain this figure. The -s option tells du to print a summary of disk usage; it prints the usage and the name of the directory, separated by a tab character. If your platform doesn't have a du command, how hard is it to write one in Unicon? The du command is straightforward to write, using the techniques presented in the previous two programming examples.

```
procedure du(dir)
    local s := stat(dir), d, sum := 0

    # If it's not a directory, just return its size
    if s.mode[1] ~== "d" then
      return s.size

    # Otherwise, find the usage of each entry and add
    d := open(dir) | fail
    while filename := read(d) do sum +:= du(filename)
    close(d)
    return sum
end
```

Using this procedure, the program fills in the table of usages.

```
    L := read_db(db)
    owners := L[1]
    quotas := L[2]
    daysover := L[3]
    usages := table(0)
    over := table()
    every dir := key(owners) do {
      usages[dir] := du(dir)
      user := owners[dir]
```

If the usage reported is greater than the quota, increment the "days over" field. Save the results and send all the email later; this will allow us to only send one message to a user that owns more than one directory.

```
      if usages[dir] > quotas[dir] then {
        /over[user] := []
        daysover[dir] +:= 1
        l := [dir, usages[dir], quotas[dir], daysover[dir]]
        push(over[user], l)
        }
      else daysover[dir] := 0
      }
    every user := key(over) do complain(user, over[user])
```

Finally, the program saves the database and sends a summary of the over-quota directories to the administrator.

```
    write_db(db, owners, quotas, daysover)
    send_summary(administrator, owners, quotas, daysover, usages)
end
```

Sending mail messages

Two procedures in the quota program send mail messages as their primary task. This is done in many older system administration scripts by executing an external mail client using the system() function. Calling system() is a potential portability problem and security hole in many applications. The quota program uses Unicon's messaging facilities to send mail, avoiding both of these problems.

Procedure complain() sends a message to the user notifying him/her that certain directories are over quota. The entry for each user is a list, each member of which is a record of an over-quota directory, stored as a list. This list has the directory name, the usage, the quota and the number of days it has been over quota.

```
procedure complain(user, L)
    msg := "Dear " || user || complaint_part1
    every l := !L do {
      msg ||:= "\t" || l[1] || "\t" || l[2] || "\t" || l[3] || "\t" || l[4] || "\n"
    }
    msg ||:= complaint_part2
    m := open("mailto:"|| user, "m", "Subject: Over Quota")
    write(m, msg)
    close(m)
end
```

Procedure send_summary() sends mail to the administrator summarizing all over-quota directories. It uses the function key() to generate the indexes for tables usages, quotas, owners, and daysover, which are maintained in parallel. This use of key() is quite common. It might be possible to combine all these parallel tables into one big table and eliminate the need to call key().

```
procedure send_summary(admin, owners, quotas, daysover, usages)
    m := open("mailto:"|| admin, "m", "Subject: Quota Summary")
    write(m, summary_header)
    every dir := key(owners) do
      if usages[dir] > quotas[dir] then {
       writes(m, dir, "\t", owners[dir], "\t", usages[dir]||"/"||quotas[dir], "\t",
                daysover[dir])
         # Flag anything over quota more than 5 days
         if daysover[dir] > 5 then writes(m, " ***")
         write(m)
      }
    close(m)
end
```

The quota database

The database is stored as a plain text file so that the system administrator can easily make changes to it. It has four fields, separated by white space (spaces or tabs): a directory, the owner of the directory, the quota, and the number of days the directory has been over quota. Procedure read_db() reads in the database. Blank lines or lines starting with '#' are ignored.

```
procedure read_db(db)
    owners := table()
    quotas := table(0)
    daysover := table(0)
    dbf := open(db) | stop("Couldn't open ",db)
```

```
    while line := read(dbf) || "\t" do
      line ? {
        tab(many('\t'))
        if pos(0) | ="#" then next
        dir := tab(upto('\t'));   tab(many(' \t'))
        user := tab(upto('\t'));  tab(many(' \t'))
        quota := tab(upto('\t')); tab(many(' \t'))
        days := tab(0) | ""
        # The "days" field can be absent in which case 0 is
        # assumed.
        if days == "" then days := 0
```

If multiple quota lines occur for a directory, the tables must be updated appropriately. The semantics of the tables require varying approaches. The owners table writes a warning message if quota lines with different owners for the same directory are found, but otherwise the owners table is unaffected by multiple entries. The actual quotas table allows multiple quota lines for a directory; in which case the quotas are added together. The daysover table retains the maximum value any quota line is overdue for a directory.

```
        if \owners[dir] ~== user then
          write(&errout, "Warning: directory ", dir, " has more than one owner.")
        owners[dir] := user
        quotas[dir] +:= quota
        daysover[dir] := days
      }
    close(dbf)
    return [owners, quotas, daysover]
end
```

Procedure write_db() rewrites a quota database with current quota information. Notice how the code preserves the comments and the blank lines that were present in the database file. This is very important when dealing with human-editable files. It also writes to a temporary file and then renames it to the correct name. This ensures that a consistent copy of the database is always present.

```
procedure write_db(db, owners, quotas, daysover)
    new_db := db || ".new"
    db_old := open(db)
    db_new := open(new_db, "w") | stop("Couldn't open", new_db)
    while line := read(db_old) do {
      line ? {
        tab(many('\t'))
        if pos(0) | ="#" then {
          write(db_new, line)
```

```
            next
            }
         dir := tab(upto('\t'))
         write(db_new, dir, "\t", owners[dir], "\t", quotas[dir], "\t", daysover[dir])
       }
     }
     close(db_old)
     close(db_new)
     rename(db, db || ".bak")
     rename(new_db, db)
end
```

Lastly, procedure init_strings() initializes global strings used for email messages. Concatenation is used to improve the readability of long strings that run across multiple lines.

```
procedure init_strings()
     complaint_part1 := ":\n" ||
       "The following directories belonging to you are"|| "over their quota:\n\n"||
       "Directory    \tUsage \tQuota \tDays Over\n"
     complaint_part2 := "\nPlease take care of it."
     summary_header := "\n"||
       "Over-quota users\n\n"||
       "Directory   \tOwner \tUsage/Quota \tDays Over\n"
end
```

15.4 Capturing a Shell Command Session

Many applications including debugging and training can benefit from the ability record a transcript of a session at the computer. This capability is demonstrated by the following program, called script. The script program uses a feature of POSIX systems called the *pty*. This is short for pseudo-tty. It is like a bi-directional pipe with the additional property that one end of it looks exactly like a conventional tty. The program at that end can set it into "no-echo" mode and so forth, just like it can a regular terminal. This application's portability is limited to the UNIX platforms.

The script program has only one option: if -a is used, output is appended to the transcript file instead of overwriting it. The option is used to set the second argument to open():

```
# script: capture a script of a shell session (as in BSD)
# Usage: script [-a] [filename]
# filename defaults to "typescript"

procedure main(L)
     if L[1] == "-a" then {
```

```
      flags := "a"; pop(L)
      }
   else flags := "w"
```

Now the program must find a pty to use. One method is to go down the list of pty device names in sequence until an open() succeeds; then call procedure capturesession(), to perform the actual logging. On POSIX systems the pty names are of the form */dev/ptyp-s0-a*. The tty connected to the other end of the pipe then has the name */dev/ttyXY*, where X and Y are the two characters from the pty's name.

```
   # Find a pty to use
   every c1 := !"pqrs" do
      every c2 := !(&digits || "abcdef") do
         if pty := open("/dev/pty" || c1 || c2, "rw") then { # Aha!
            capturesession(fname := L[1] | "typescript", pty, c1 || c2, flags)
            stop("Script is done, file ", image(fname))
         }
   stop("Couldn't find a pty!")
end
```

Note

If you do not have read-write permissions on the pseudotty device the program uses, the program will fail. If this program does not work, check the permissions on the /dev/tty* device it is trying to use.

The script program uses the system() function, executing the user's shell with the standard input, standard output, and standard error streams all redirected to be the tty end; then it waits for input (using select()) either from the user or from the spawned program. The program turns off echoing at its end, since the spawned program will be doing the echoing. The program sends any input available from the user to the spawned shell; anything that the shell sends is echoed to the user, and also saved to the script file.

```
procedure capturesession(scriptfile, pty, name, flags)
   f := open(scriptfile, flags) | stop("Couldn't open ", image(scriptfile))
   tty := open("/dev/tty" || name, "rw") | stop("Couldn't open tty!")
   shell := getenv("SHELL") | "/bin/sh"
   system([shell, "-i"], tty, tty, tty, "nowait")

   # Parent
   close(tty)
   system("stty raw -echo")

   # Handle input
   while L := select(pty, &input) do {
```

```
      if L[1] === &input then writes(pty, reads()) | break
      else if L[1] === pty then {
        writes(f, inp := reads(pty)) | break
        writes(inp)
      }
    }

    (&errno = 0) | write(&errout, "Unexpected error: ", &errortext)
    system("stty cooked echo")
    close(f)
end
```

When script gets an EOF on either stream, it quits processing and closes the file, after resetting the parameters of the input to turn echoing back on.

15.5 Filesystem Backups

By now you have probably been told a few thousand times that regular backups of your files are a good idea. The problem arises when you are dealing with a system with a large amount of file storage, like modern multi-user systems. For these systems, *incremental backups* are used. Incremental backups exploit the fact that a very large number of files on the system change rarely, if ever. There is no need to save them all to the backup medium every time. Instead, each backup notes the last time that a backup was performed, and only saves files that have been modified since then.

Each backup is given a number; the files in backup n were modified later than the last backup of a lower number. A Level 0 backup saves all the files on the system.

This section presents an incremental backup utility called backup. The backup utility saves all the files in a directory specified with the -o flag. Typically this will be the external backup storage, like a floppy disk (for small backups) or a Zip disk. The program re-creates the directory structure on the external device so that files may easily be recovered. The disadvantage of this strategy is that it does not compress the whole archive together, and therefore requires more storage than is strictly necessary. On the positive side, this approach avoids the fragility of compressed archives, in which the loss of even a small amount of data can render the whole archive unreadable.

Note

This program only saves to media that has a directory structure on which regular files may be written, such as jump drives. It does not work on backup devices such as tape drives that require media to be written in a proprietary format.

Another feature of this backup program is that certain directories can be automatically excluded from the backup, such as temporary directories like /tmp on UNIX systems. One directory that *must* be excluded is the output device itself, or you will find that a very large

amount of storage is needed! One of the best parts about backup is that you can modify this program to suit your needs: file compression, error recovery, support for multiple discs, or anything else that you require.

```
#
# backup.icn - incremental filesystem backups
#
# Usage:
#   ./backup [-nlevel] [-ooutput] [dir]
#
# Save all files that have changed since the last backup of higher level
# (a level 0 backup is the highest level and saves all files; it is the
# default). The files are all saved to the directory "output", which is
# probably a mounted backup device like a flash drive.
#
# Example:
#    backup -n3 -o/mnt/zip /home/bob

link options

global dbase, exclude, levels
global output, last

procedure main(args)
    dbase := "/var/run/backups.db"
    exclude := ["/mnt", "/tmp", "/dev", "/proc"]

    # Process arguments
    opt := options(args, "-n+ -o:")
    level := integer(\opt["n"]) | 0
    output := opt["o"]
    dir := args[1] | "/"

    \output | stop("An output directory (option -o) must be specified!")
    if level < 0 | level > 9 then stop("Only levels 0..9 can be used.")

    # Get the time of the previous lower-numbered backup
    last := get_time(level)
    # Now look for files newer than "last"
    traverse(dir)

    # Write the database
    save_time(level)
end
```

Procedure traverse() is the interesting part of the program. It recursively descends the filesystem hierarchy, saving all the files it finds that are newer than the last backup. When a recent plain file is found, the procedure copy_file() is called.

```
procedure traverse(dir)
   # Skip excluded directories
   if dir == !exclude then return

   # Read all the files; for any non-special files, copy them
   # over to the output dir, creating directories as necessary
   d := open(dir) | {
     write(&errout, "Couldn't stat ", dir, " ", &errortext)
     return
     }
   if dir[-1] ~== "/" then dir ||:= "/"
   while name := read(d) do {
     if name == ("." | "..") then next
     s := stat(dir || name) | {
       write(&errout, "Couldn't stat ", dir||name, " ", &errortext)
       next
     }
     if s.mode[1] == "d" then traverse(dir || name)
     else {
       # Only save plain files
       if s.mode[1] == "-" & s.ctime > last then copy_file(dir, name)
       }
     }
end
```

To copy a file, you must first ensure that its parent directory exists. If it doesn't, the parent directory is created; then the file itself is copied. For efficiency, backup uses the system program (cp on UNIX) to copy the file. When directories are created, backup copies the owner and mode from the directory being backed up.

Note

This program must be run with administrator privileges for it to be able to read all the files on the system and also to be able to set the owner and mode.

```
procedure copy_file(dir, name)
   # First, make sure the directory exists
   mkdir_p(output, dir)
   system("cp " || dir || "/" || name || " " || output || "/" || dir)
end

procedure mkdir_p(prefix, dir)
```

```
    # The name is supposed to be reminiscent of "mkdir -p"
    # Start at the first component and keep going down it,
    # copying mode and owner.
    dir ||:= "/"
    d := ""
    dir ? while comp := tab(upto('/')) do {
      tab(many('/'))
      d ||:= "/" || comp
      if \stat(prefix || d) then {
        # The directory doesn't exist; create it. d is the
        # directory being copied over; get its uid and mode.
        s := stat(d)
        mkdir(prefix || d, s.mode[2:11])
        chown(prefix || d, s.uid, s.gid)
      }
    }
end
```

The database file is very simple: for every level, the date of the last backup at that level is stored. Dates are stored in the system native format so that comparisons with file modification dates can be easily performed. If no earlier backup is found, procedure get_time() returns the earliest possible time (the epoch); all files will have newer modified times, and therefore be backed up.

All the dates found are stored in a global table so that they will be accessible when backup writes out the database later.

```
procedure get_time(n)
    # Get the date of earlier backup
    levels := table()
    f := open(dbase)

    while line := read(\f) do
      line ? {
        lev := integer(tab(upto(' ')))
        move(1)
        date := tab(0)
        levels[lev] := date
        }
    close(\f)
    every i := integer(!&digits) do
      if i < n then prev := \levels[i]
    /prev := 0              # default: the epoch
    return prev
end
```

Finally, the program saves the database of dates. It fetches the current time to save with the current level, and deletes all higher-numbered backups from the database.

```
procedure save_time(n)
   levels[n] := &now

   f := open(dbase, "w") | stop("Couldn't open table ", dbase)
   every i := integer(!&digits) do
     if i <= n then write(f, i, " ", \levels[i])
     else break
   close(f)
end
```

15.6 Filtering Email

Unicon's messaging facilities can be used for filtering email messages obtained from a POP server. A filtering rule has two components: a pattern to look for, and an action to perform if the string is found. The action to be perform can take a few forms: the message can be saved in a folder; it can be deleted; or it can be forwarded to some other address. We can use a record to represent a rule:

```
record rule(pattern, action, args)
```

Here are some example rules represented as records:

```
   rule("From: .*spammer\.com", "delete")
   rule("From: .*unicon-list@lists\.sourceforge\.net", "save", "folders/unicon")
   rule("Subject: .*xyzzy", "forward" "shamim@home-domain.org")
```

We read all the rules from a file and use them to filter all messages. To connect to the POP server, we need some parameters: the server to connect to, the username and the password. These values are stored in a parameter file that we read. The parameters from the file can be stored in a global table called params. (We will deal with reading the file later.) First, we connect to the POP server and for every message, invoke the filter:

```
   params := get_parameters()
server := \params["server"] | stop("No server specified in .popfilter file")
user := \params["user"] | stop("No user specified in .popfilter file")
password := \params["password"] |
            stop("No password specified in .popfilter file")
\params["inbox"] | stop("No inbox specified in .popfilter file")

url := "pop://" || user || ":" || password || "@" || server
```

```
  s := open(url, "m") | stop("Couldn't connect to ", image(url))
  while filter(pop(s), rules) do
    # Make sure there are no errors
    if s["Status-Code"] >= 300 then {
      close(s)
      stop(&progname, ": POP error: ", s["Status-Code"],
          " ", s["Reason-Phrase"] | "")
    }
```

After reading each message, we make sure that there wasn't an error in talking to the POP server – if there was one, we print the error and exit.

The filtering procedure tries to match the message against every rule; if one matches, it hands the message off to the action procedure. If no rule matches, the message is saved to the default folder (the inbox).

```
procedure filter(message, rules)
  local filter_rule
  every filter_rule := !rules do
    if message? ReFind(filter_rule.pattern) then
      perform(filter_rule.action, filter_rule.args, message) |
        write(&errout, "Error in ",image(filter_rule.action),
            " ", image(filter_rule.args))
  # No action matched so we save it in the inbox
  perform("save", params["inbox"], message)
end
```

Since the patterns are regular expressions, we link in the IPL regexp package for the procedure ReFind() that will do the pattern matching.

```
procedure perform(action, args, message)
  local procname
  procname := \proc("filterproc_" || action, 2) |
    stop(&progname, ": action ", image(action), " unknown.")
  procname(message, args) | fail
  return ""
end
```

We use Unicon's string invocation (and we'd better remember to put invocable all at the top of the program) to call the procedure associated with each defined action. The function proc() tries to find the procedure with the right name; if it fails, it is an unknown action. The names of these procedures all have the form "filterproc_" appeneded with the name of the action. We make sure that the procedures succeed if there was no error in performing the action.

```
procedure filterproc_delete(message, args)
  return ""              # We don't have to do anything
end

procedure filterproc_save(message, filename)
  local f
  f := open(filename, "a") | fail
  write(f, message)
  close(f)
  return ""
end
procedure filterproc_forward(message, address)
  local subject, s

  # Find the subject of the message and use that
  message? (tab(find("Subject: ")) & subject := tab(upto('\n')))
  subject := "Subject: " || \subject

  s := open("mailto://" || address, "m", subject) | fail
  write(s, message)
  close(s)
  return ""
end
```

It would be easy to implement an action that sends messages to external programs – in procedure perform(), if the first character of the action is the pipe symbol "|" we can open a pipe and write the message to it instead of calling the right action procedure. (This would be a good "exercise left to the reader!")

We still need to read the files and set up the params table and the list of rules. It is a good idea to allow blank lines and comments in files we read, since humans will be editing these files and we like comments to remind ourselves what things do. We need a procedure that reads a file and strips out comments and blank lines:

```
# Read a line from a file, skipping comments and blank lines
procedure getline(f)
  local line, s
  while line := read(f) do line? {
    s := tab(upto('#') | 0)
    if *trim(s) = 0 then next
    return s
  }
end
```

We read the POP parameters from a file named .popfilter in the user's home directory

and return a table of name-value pairs. The global variable WS is a cset that contains the space and tab characters; it is initialized in main(), when the program first starts executing.

```
# Read the ~/.popfilter file and return a table of name -> value pairs
procedure get_parameters()
  local f, P := table(), line
  f := open(getenv("HOME") || "/.popfilter") | return P

  while line := getline(f) do line ? {
    pname := tab(upto(WS))
    tab(many(WS))
    P[pname] := tab(0)
  }
  close(f)
  return P
end
```

The filtering rules are read from the file named on the command line. The pattern is separated from the action by an exclamation point, and (as usual) comments, blank lines and whitespace (around the "!") are allowed. Whitespace inside the regular expression is significant and should not be discarded so we use trim() to make sure we only remove trailing whitespace. Searches are anchored to the begin of line so we can easily look for individual headers; instead of separating a message into lines and using the "beginning of line" regular expression operator, we simply we add a newline character to the front of the pattern.

```
# Read the file and return a list of rules. All the work is
# actually done in getline() and parse()
procedure read_rulefile(filename)
  local f, rules := [ ]
  f := open(filename) | fail
  while push(rules, parse(getline(f)))
  close(f)
  return rules
end

# Parse a line into a pattern, an action, and optionally
# arguments to the action.
procedure parse(s)
  local regexp, action

  s ? {
    tab(many(WS))
    regexp := tab(upto('!')) & move(1)
    tab(many(WS))
```

```
    action := tab(upto(WS) | 0)
    pos(0) | (tab(many(WS)) & args := tab(0))
  }
  return rule("\n" || trim(\regexp), \action, args)
end
```

Here is an example action file that includes the example rules from above:

```
# Flush evil spammers!
From: .*spammer\.com        !delete
Subject: .*MAKE MONEY FAST  !delete    # Got enough, thanks!

# Cool people get special attention
From: .*unicon-list@lists\.sourceforge\.net  !save folders/unicon
From: .*(parlett|jeffery|pereda)    !save folders/urgent

# The secret password!! Forward it to the pager.
Subject: .*xyzzy            !forward pager@shamims-domain.org
```

And here's an example .popfilter file:

```
# Settings for popfilter.icn, a POP client and email filter
# July 2002

# Where incoming mail is stored locally
inbox /var/mail/shamim

# The POP server
server jantar

# Remember, this is the username and password on jantar
user spm
password Xyz!1234
```

We have introduced the main procedure in pieces; here it is all in one place:

```
link regexp
record rule(pattern, action, args)
global WS


# Email and POP parameters
global params

# We're using string invocation
```

```
invocable all
procedure main()
  local server, user, password
  WS := ' \t'               # whitespace
  params := get_parameters()
  server := \params["server"] | stop("No server specified in .popfilter file")
  user := \params["user"] | stop("No user specified in .popfilter file")
  password := \params["password"] | stop("No password specified in .popfilter file")
  \params["inbox"] | stop("No inbox specified in .popfilter file")
  if *args = 0 then stop("Usage: ", &progname, " filter-rule-file")

  url := "pop://" || user || ":" || password || "@" || server

  filter_rules := read_rulefile(args[1]) |
    stop("Couldn't read filter rules from ", image(args[1]))
  filter(testmsg(), filter_rules)

  s := open(url, "m") | stop("Couldn't connect to ", image(url))

  while filter(pop(s), filter_rules) do
    if s["Status-Code"] >= 300 then {
      close(s)
      stop(&progname, ": POP error: ", s["Status-Code"], " ", s["Reason-Phrase"] | "")
    }
  close(s)
end
```

The complete program popfilter.icn can be obtained from the book's website. There are still a few things this program would need to be truly useful. For example, it should allow non-anchored searches, and the user should be able to specify whether case is significant while looking for a pattern – or perhaps a part of a pattern may be case sensitive by itself. Also, what if we want a "!" in the regular expression? We hope that you will be inspired to make these additions yourself.

15.7 Summary

Writing utilities and system administration tools is easy in Unicon. These programs rely on the system facilities described in Chapter 5. While we do not advocate using Unicon in every case, it is an advantage that your applications language is also an effective scripting language. Ordinary programs can easily take on scripting tasks, and programs that might otherwise be written in a scripting language have Unicon's cleaner design and more robust set of control and data structures available.

Chapter 16

Internet Programs

The Internet is central to modern computing. Because it is ubiquitous, programmers should be able to take it for granted. Writing applications that use the Internet should be just as easy as writing programs for a standalone desktop computer. In many respects this ideal can be achieved in a modern programming language. The core facilities for Internet programming were introduced with simple examples as part of the system interface in Chapter 5. This chapter expands on this important area of software development. This chapter presents examples that show you how to

- Write Internet servers and clients
- Build programs that maintain a common view of multiple users' actions

16.1 The Client-Server Model

The Internet allows applications to run on multiple connected computers using any topology, but the standard practice is to implement a client/server topology in which a user's machine plays the role of a client, requesting information or services from remote machines, each of which plays the role of a server. The relationship between clients and servers is many-to-many, since one client can connect to many servers and one server typically handles requests from many clients.

Writing a client can be easy. For simple read-only access to a remote file, it is just as easy as opening a file on the hard disk. Most clients are more involved, sending out requests and receiving replies in some agreed-upon format called a *protocol*. A protocol may be human readable text or it may be binary, and can consist of any number of messages back and forth between the client and server to transmit the required information. The most common Internet protocols are built-in parts of Unicon's messaging facilities, but some applications define their own protocol.

Writing a server is more difficult. A server sits around in an infinite loop, waiting for clients and servicing their requests. When only one client is invoking a server, its job is

simple enough, but when many simultaneous clients wish to connect, the server program must either be very efficient or else the clients will be kept waiting for unacceptably long periods.

Although the following example programs emphasize how easy it is to write Internet clients and servers in Unicon, writing "industrial strength" applications requires additional security considerations which are mostly beyond the scope of this book. For example, user authentication and encryption are essential in most systems, and many modern servers are carefully tuned to maximize the number of simultaneous users they support, and minimize their vulnerability to denial-of-service attacks.

16.2 An Internet Scorecard Server

Many games with numeric scoring systems feature a list of high scores. This feature is interesting on an individual machine, but it is ten times as interesting on a machine connected to the Internet! The following simple server program allows games to report their high scores from around the world. This allows players to compete globally. The scorecard server is called scored. By convention, servers are often given names ending in "d" to indicate that they are daemon programs that run in the background.

The scorecard client procedure

Before examining the server code, take a look at the client procedure that a game calls to communicate with the scored server. To use this client procedure in your programs, add the following declaration to your program.

```
link highscor
```

The procedure highscore() opens a network connection, writes four lines consisting of the protocol name "HSP", the name of the game, the user's identification (which could be a nickname, a number, an e-mail address, or anything else), and that game's numeric score. Procedure highscore() then reads the complete list of high scores from the server, and returns the list. Most games write the list of high scores to a window for the user to ponder.

```
procedure highscore(game, userid, score, server)
   if not find(":", server) then server ||:= ":4578"
   f := open(server, "n") | fail

   # Send in this game's score
   write(f, "HSP\n", game, "\n", userid, "\n", score) |
     stop("Couldn't write: ", &errortext)
```

```
    # Get the high score list
    L := ["High Scores"]
    while line := read(f) do put(L, line)
    close(f)
    return L
end
```

The Scorecard server program

The scorecard server program, scored.icn illustrates issues inherent in all Internet servers. It must sit at a port, accepting connection requests endlessly. For each connection, a call to score_result() handles the request. The main() procedure given below allows the user to specify a port, or uses a default port if none is supplied. If another server is using a given port, it won't be available to this server, and the client and server have to agree on which port the server is using.

```
procedure main(av)
    port := 4578 # a random user-level port
    if av[i := 1 to *av] == "-port" then port := integer(av[i+1])

    write("Internet Scorecard version 1.0")
    while net := open(":" || port, "na")  do {
      score_result(net)
      close(net)
      }
    (&errno = 0) | stop("scored net accept failed: ", &errortext)
end
```

The procedure score_result() does all the real work of the server, and its implementation is of architectural significance. Any delay in handling a request implies the server will be unable to handle other simultaneous client requests. For this reason, many servers immediately spawn a separate process to handle each request. You could do that with system(), as illustrated in Chapter 5, or launch a thread for it, but for scored this is overkill. The server handles each request almost instantaneously.

Some small concessions to security are in order, even in a trivial example such as this. If a bogus Internet client connects by accident, it will fail to identify our protocol and be rejected. More subtly, if a rogue client opens a connection and writes nothing, we do not want to block waiting for input or the client will deny service to others. A call to select() is used to guarantee the server receives data within the first 1000 milliseconds (1 second). A last security concern is to ensure that the "game" filename supplied is valid; it must be an existing file in the current directory, not something like /etc/passwd for example.

The score_result() procedure maintains a static table of all scores of all games that it knows about. The keys of the table are the names of different games, and the values in

the table are lists of alternating user names and scores. The procedure starts by reading the game, user, and score from the network connection, and loading the game's score list from a local file, if it isn't in the table already. Both the score lists maintained in memory, and the high scores files on the server, are sequences of pairs of text lines containing a userid followed by a numeric score. The high score files have to be created and initialized manually with some N available (userid,score) pairs of lines, prior to their use by the server.

```
procedure score_result(net)
    local s := ""
    static t, gamenamechars
    initial {
        t := table()
        gamenamechars := &letters++&digits++'-_'
        }

    select(net, 1000) | { write(net, "timeout"); fail }
    (s ||:= ready(net)) ? {
        = "HSP\n" | { write(net, "wrong protocol"); fail }
        game := tab(many(gamenamechars)) | { write(net,"no game?"); fail }
        = "\n"
        owner := tab(many(gamenamechars)) | { write(net,"no owner?"); fail }
        = "\n"
        score := tab(many(&digits)) | { write("no score?"); fail }
        }

    if t[game] === &null then {
        if not (f := open(game)) then {
            write(net, "No high scores here for ", game)
            fail
            }
        t[game] := L := []
        while put(L, read(f))
        close(f)
        }
    else
      L := t[game]
```

The central question is whether the new score makes an entry into the high scores list or not. The new score is checked against the last entry in the high score list, and if it is larger, it replaces that entry. It is then "bubbled" up to the correct place in the high score list by repeatedly comparing it with the next higher score, and swapping entries if it is higher. If the new score made the high score list, the list is written to its file on disk.

```
    if score > L[-1] then {
```

```
      L[-2] := owner
      L[-1] := score
      i := -1
      while L[i] > L[i-2] do {
        L[i] :=: L[i-2]
        L[i-1] :=: L[i-3]
        i -:= 2
        }
      f := open(game,"w")
      every write(f, !L)
      close(f)
      }
```

Note

List L and t[game] refer to the same list, so the change to L here is seen by the next client that looks at t[game].

Lastly, whether the new score made the high score list or not, the high score list is written out on the network connection so that the game can display it.

```
   every write(net, !L)
end
```

Is this high score application useful and fun? Yes! Is it secure and reliable? No! It records any scores it is given for any game that has a high score file on the server. It is utterly easy to supply false scores. This is an honor system.

16.3 A Simple "Talk" Program

E-mail is the king of all Internet applications. After that, some of the most popular Internet applications are real-time dialogues between friends and strangers. Many on-line services rose to popularity because of their "chat rooms," and Internet Relay Chat (IRC) is a ubiquitous form of free real-time communication. These applications are evolving in multiple directions, such as streaming multimedia, and textual and graphical forms of interactive virtual reality. While it is possible to create arbitrarily complex forms of real-time communication over the Internet, for many purposes, a simple connection between two users' displays, with each able to see what the other types, is all that is needed.

The next example program, called italk, is styled after the classic BSD UNIX talk program. The stuff you type appears on the lower half of the window, and the remote party's input is in the upper half. Unlike a chat program, the characters appear as they are typed, instead of a line at a time. In many cases this allows the communication to occur more smoothly with fewer keystrokes.

The program starts out innocently enough, by linking in library functions for graphics, defining symbolic constants for font and screen size. Among global variables, vs stands for vertical space, cwidth is column width, wheight and wwidth are the window's dimensions, and net is the Internet connection to the remote machine.

```
link graphics
$define ps 10              # The size of the font to use
$define lines 48           # No. of text lines in the window
$define margin 3           # Space to leave around the margins
$define START_PORT 1234 $define STOP_PORT  1299
global vs, cwidth, wheight, wwidth, net
```

The main() procedure starts by calling win_init() and net_init() to open up a local window and then establish a connection over the network, respectively. The first command line argument is the user and/or machine to connect to.

```
procedure main(args)
   win_init()
   net_init(args[1] | "127.0.0.1")
```

Before describing the window interaction or subsequent handling of network and window system events, consider how italk establishes communication in procedure net_init(). Unlike many Internet applications, italk does not use a conventional client/server architecture in which a server daemon is always running in the background. To connect to someone on another machine, you name him or her in the format user@host on the command line. The code attempts to connect as a client to someone waiting on the other machine, and if it fails, it acts as a server on the local machine and waits for the remote party to connect to it.

```
procedure net_init(host)
   host ?:= {
     if user := tab(find("@")) then move(1)
     tab(0)
     }
   net := (open_client|open_server|give_up)(host, user)
end
```

The attempt to establish a network connection begins by attempting to open a connection to an italk that is already running on the remote machine. The italk program works with an arbitrary user-level set of ports (defined above as the range 1234-1299). An italk client wades through these ports on the remote machine, trying to establish a connection with the desired party. For each port at which open() succeeds, the client writes its user name, reads the user name for the process on the remote machine, and returns the connection if the desired party is found.

```
procedure open_client(host, user)
    port := START_PORT
    if \user then {
      while net := open(host || ":" || port, "n") do {
        write(net, getenv("USER") | "anonymous")
        if user == read(net) then return net
        close(net)
       port +:= 1
        }
      }
    else {
      net := open(host || ":" || port, "n")
      write(net, getenv("USER") | "anonymous")
      read(net) # discard
      return net
      }
end
```

The procedure open_server() similarly cycles through the ports, looking for an available one on which to wait. When it receives a connection, it checks the client user and returns the connection if the desired party is calling.

```
procedure open_server(host, user)
    repeat {
     port := START_PORT
      until net := open(":" || port, "na") do {
        port +:= 1
        if port > STOP_PORT then fail
        }
      if not (them := read(net)) then {
        close(net)
        next
        }
      if /user | (them == user) then {
        write(net, getenv("USER") | "anonymous")
        WAttrib("label=talk: accepted call from ", them)
        return net
        }
      WAttrib("label=talk: rejected call from ", them)
      write(net, getenv("USER") | "anonymous")
      close(net)
      }
end
```

This connection protocol works in the common case, but is error prone. For example,

if both users typed commands at identical instants, both would attempt to be clients, fail, and then become servers awaiting the other's call. Perhaps worse from some users' point of view would be the fact that there is no real authentication of the identity of the users. The italk program uses whatever is in the USER environment variable. The UNIX talk program solves both of these problems by writing a separate talk server daemon that performs the *marshalling*. The daemons talk across the network and negotiate a connection, check to see if the user is logged in and if so, splash a message on the remote user's screen inviting her to start up the talk program with the first user's address.

The next part of italk's code to consider is the event handling. In reality, each italk program manages and multiplexes asynchronous input from two connections: the window and the network. The built-in function that is used for this purpose is select(). The select() function will wait until some (perhaps partial) input is available on one of its file, window, or network arguments.

The main thing to remember when handling input from multiple sources is that you must not block for I/O. This means: use listener mode for new connections or a timeout parameter with open(), and when handling network connections, never use read(), only use reads() or better yet ready(). For windows you must also avoid read()'s library procedure counterpart, WRead(). The code below checks which connection has input available and calls Event() as events come in on the window, and calls reads() on the network as input becomes available on it. In either case the received input is echoed to the correct location on the screen. Ctrl-D exits the program. To accept a command such as "quit" would have meant collecting characters till you have a complete line, which seems like overkill for such a simple application.

```
    repeat {
      *(L := select(net, &window))>0 | stop("empty select?")
      if L[1] === &window then {
        if &lpress >= integer(e := Event()) >= &rdrag then next
        if string(e) then {
          writes(net, e) | break
          handle_char(2, e) | break
          WSync()
          }
        }
      else {
        s := reads(net) | break
        handle_char(1, s) | break
        WSync()
        }
      }
    close(net)
end
```

After such a dramatic example of input processing, the rest of the italk program is a bit anticlimactic, but it is presented anyhow for completeness sake. The remaining procedures are all concerned with managing the contents of the user's window. Procedure handle_char(w, c), called from the input processing code above, writes a character to the appropriate part of the window. If w = 1 the character is written to the upper half of the window. Otherwise, it is written to the lower half. The two halves of the window are scrolled separately, as needed.

```
procedure handle_char(w, c)
    # Current horiz. position for each half of the window
    static xpos
    initial xpos := [margin, margin]
    if c == "\^d" then fail     # EOF

    # Find the half of the window to use
    y_offset := (w - 1) * wheight/2

    if c == ("\r"|'\n') | xpos[w] > wwidth then {
      ScrollUp(y_offset+1, wheight/2-1)
      xpos[w] := margin
      }
    if c == ("\r"|'\n') then return
    #handles backspacing on the current line
    if c =="\b" then {
      if xpos[w] >= margin + cwidth then {
        EraseArea(xpos[w]-cwidth, y_offset+1+wheight/2-1-vs,cwidth,vs)
        xpos[w] -:= cwidth
        return
        }
      }
    DrawString(xpos[w], wheight/2 + y_offset - margin, c)
    xpos[w] +:= cwidth
    return
end
```

Scrolling either half of the window is done a line at a time. The graphics procedure CopyArea() is used to move the existing contents up one line, after which EraseArea() clears the line at the bottom.

```
procedure ScrollUp(vpos, h)
    CopyArea(0, vpos + vs, wwidth, h-vs, 0, vpos)
    EraseArea(0, vpos + h - vs, wwidth, vs)
end
```

The window is initialized with a call to the library procedure, WOpen(), which takes attribute parameters for the window's size and font. These values, supplied as defined symbols at the top of the program, are also used to initialize several global variables such as vs, which gives the vertical space in pixels between lines.

```
procedure win_init()
    WOpen("font=typewriter," || ps, "lines=" || lines, "columns=80")
    wwidth := WAttrib("width")
    wheight := WAttrib("height")
    vs := WAttrib("fheight")
    cwidth := WAttrib("fwidth")
    DrawLine(0, wheight/2, wwidth, wheight/2)
    Event()
end
```

Lastly, the procedure give_up() writes a message and exits the program, if no network connection is established. If user is null and the non-null test (the backslash operator) fails, the concatenation is not performed and alternation causes the empty string to be passed as the second argument to stop().

```
procedure give_up(host, user)
    stop("no connection to ", (\user || "@") | "", host)
end
```

What enhancements would make italk more interesting? An obvious extension would be to use a standard network protocol, such as that of UNIX talk, so that italk could communicate with other users that don't have italk. UNIX talk also offers a more robust connection and authentication model (although you are dependent on the administrator of a remote machine to guarantee that its talkd server is well behaved). Another feature of UNIX talk is support for multiple simultaneously connected users.

One neat extension you might implement is support for graphics, turning italk into a distributed whiteboard application for computer-supported cooperative work. To support graphics you would need to extend the window input processing to include a simple drawing program, and then you would need to extend the network protocol to include graphics commands, not just keystrokes. One way to do this would be to represent each user action (a keystroke or a graphics command) by a single line of text that is transmitted over the network. Such lines might look like:

```
key H
key i
key !
circle 100,100,25
```

and so forth. At the other end, the program deciphering these commands translates them into appropriate output to the window, which would be pretty easy, at least for simple graphics. The nice part about this solution is that this particular collaborative whiteboard application would work fine across differing platforms (Linux, Microsoft Windows, and so on) and require only a couple hundred lines of code!

16.4 Summary

Writing Internet programs can be easy and fun, although it is easy to underestimate the security needed. There are several different ways to write Internet programs in Unicon. The database interface presented in Chapter 6 allows you to develop client/server applications without *any* explicit network programming when the server is a database. A SQL server is overkill for many applications such as the high score server, and it is not appropriate for other non-database network applications such as the italk program.

For these kinds of programs, it is better to "roll your own" network application protocol. Once a connection is established (perhaps using a client/server paradigm), the actual communication between programs is just as easy as file input and output. If you do roll your own network application, keep the protocol simple; it is easy enough to write yourself into deadlocks, race conditions, and all the other classic situations that make parallel and distributed programming perilous.

Chapter 17

Genetic Algorithms

The previous three chapters showed you how to write Unicon programs with many kinds of Internet and system capabilities that are expected of most modern applications. Unicon is great for generic computing tasks, but it really excels when its advanced features are applied in application areas where the development of the algorithms is complex. This chapter describes how to use Unicon to build an entire, somewhat complex application with reusable parts. The field of *genetic algorithms* (GAs) is an exciting application domain with lots of opportunities for exploratory programming. When you're finished with this chapter, you will

- Understand the basics of genetic algorithms.
- See how to build genetic algorithm engine in Unicon.
- Use that GA engine to build programs for your own projects.

17.1 What are Genetic Algorithms?

The broad field of evolutionary computation has been an area of active research since the 1950s. Initially, it was led by computer scientists that believed evolution could be used as an optimization tool for engineering problems. Genetic Programming (GP) focuses on evolving computer programs to perform various tasks. On the other hand, GAs focus on the simpler task of evolving data that is used to solve a problem. The GAs that we'll study have a binary representation. Increasingly there has been a shift towards non-binary representations such as floating-point numbers in GA-related projects. That field has typically been given the more general name of evolutionary algorithms. GAs have one of the most well-defined mathematical foundations in all of evolutionary computation, and are a good place to start exploring.

John Holland invented the first GAs in the 1960s. His goal was to study adaptation as it occurs in nature and then to create computer systems to model the adaptive process.

Holland combined four elements that are common to all GAs:

- A population of individuals
- Selection based on fitness
- Mating of individuals
- Random mutation

Consider the very simple problem of finding the largest number encoding in binary with six digits. Assume the GA knows nothing about binary encoding.

While making use of a population might seem to be a necessary element for any evolutionary computation, it is not. Instead, you could focus all your efforts on improving one individual. In this case, fitness is exactly the numerical value of an individual. For example, you could examine the fitness of this one individual with an exhaustive search:

```
best := 0
every i := 0 to 2^6 do
  best <:= i
```

Suppose you only make use of the elements of a population and selection based on fitness. You could have a population of six individuals, randomly initialized, and you could attempt to improve the overall fitness of the six by replacing the lowest fitness individual with a random one. The code below shows how to implement this idea:

```
maxi := 2^6
population := [?maxi, ?maxi, ?maxi, ?maxi, ?maxi, ?maxi]
every i := 1 to 100 do {
  worst := 1
  every i := 1 to 5 do
    if population[worst] <= population[i] then
      worst := i
  population[i] := ?maxi
  }
```

Before modeling mating and mutation, you'll have to create a more detailed representation of the internals of an individual.

17.2 Operations: Fitness, Crossover, and Mutation

An individual is represented by a string from a binary alphabet. Incidentally, natural evolution of DNA is based on a quaternary alphabet, but the size of the alphabet is unimportant for a computer model. In Unicon, you could represent these individuals with lists of integers. However, strings of "1" and "0" characters provide a representation that is easier to use. So, here is a more explicit representation of a population:

```
population := ["010111", "000101", "111101", "111011", "111110", "010110"]
```

Fitness

The fitness can be computed by converting the string representation into an integer as follows: integer("2r" || population[1]). The 2r means that this is a literal representation of an integer in base two form. There are many different possible selection schemes used in GAs. This chapter uses one that has proven to be very robust in a large number of different GA applications, called *tournament selection*. The general idea is to group the individuals and have them compete head-to-head. The winners of the tournaments are selected to live in the next generation; their bits are copied into an element in the new population. Tournaments of size two work well. All you must do is randomly pair up the individuals, and move the one with the higher fitness to the next generation. Because you generally will want the population size to remain constant, you'll have to do this pairing twice. Here is a tournament selection on the above population:

```
population[1] := "010111" # 23 winner
population[6] := "010110" # 22
population[3] := "111101" # 61 winner
population[4] := "111011" # 59
population[5] := "111110" # 62 winner
population[2] := "000101" # 5
```

The second round of selections is listed here:

```
population[5] := "111110" # 62 winner
population[6] := "010110" # 22
population[2] := "000101" # 5
population[3] := "111101" # 61 winner
population[4] := "111011" # 59 winner
population[1] := "010111" # 23
```

The end result of tournament selection is listed here:

```
next_gen[1] := population[1] := "010111" # 23
next_gen[2] := population[3] := "111101" # 61
next_gen[3] := population[5] := "111110" # 62
next_gen[4] := population[5] := "111110" # 62
next_gen[5] := population[3] := "111101" # 61
next_gen[6] := population[4] := "111011" # 59
```

Notice how there are two copies of population[3] and population[5] in next_gen. On the other hand, there are no copies of population[2].

Crossover

Mating, more technically known as crossover, involves the sharing of information between members of the population. Again there are many different types of mating schemes, but this chapter describes one called two-point crossover (Figure 17-1) that has proven to be very robust for a wide range of GA applications. Once again, randomly pair up the individuals, but this time instead of competing, individuals will mate. First you must transform the linear strings into circular rings. For each pair, randomly select two points in the ring and cut the rings at the two selected points. Then swap the ring segments to form two new rings.

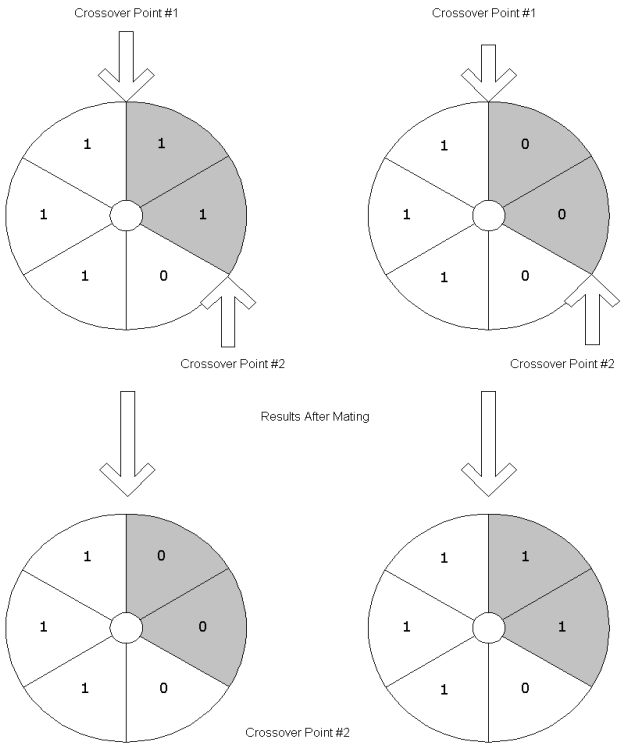


Figure 17-1: Two-point crossover.

The code for two-point crossover is presented below. The ?(lchrom+1) expression picks a random number between one and the length of the chromosome. Variables a and b are initialized to two different values in this range; a is made the smaller of the two indices. Two children in the new generation are formed by splicing portions of parent1 and parent2 within the range from a to b.

```
a := ?(lchrom+1)
while ((b := ?(lchrom+1)) = a)
if a > b then a :=: b
ncross +:= 1
child1 := parent1[1:a] || parent2[a:b] || parent1[b:0]
child2 := parent2[1:a] || parent1[a:b] || parent2[b:0]
```

Mutation

The last GA operation is mutation. Mutation works at the independent level of single binary digits. To implement mutation, take a look at each bit of each individual of the population. With a fixed probability, flip the value of the bit. This is the basic mechanism for injecting completely new information into the population. Almost all of that information will be useless, but as the GA evolves it will weed out the useless information and keep the useful information.

17.3 The GA Process

Now that you have a handle on the basic operations, it is time to describe the basic GA algorithm for applying these operations.

1. Generate a random population of n individuals each with l-bits.
2. Calculate the fitness of each individual.
3. Perform tournament selection on the population to produce a new generation.
4. With probability p_c, mate pairs of individuals using two-point crossover.
5. With probability p_m, mutate the bits in the population.
6. Replace the old population with the new generation.
7. Go to step 2, until the population meets some desired condition.

Even with this mechanical algorithm, applying a GA to any specific problem remains an art. For example, step 7 leaves the desired stopping condition up to the implementer. Typically the stopping condition might be something like: until the average fitness has not risen significantly in the last five generations. This is one of many implementation choices you have to make, from variations on crossover to adjusting the mutation and mating rates. Here are some time-tested rules of thumb:

1. Encode the solutions to a problem with as few bits as possible but not at the expense of making the encoding very complex.

2. Let the size of the population be at least twenty but not so large that your computer program is intolerably slow.

3. Mating rates between 30 percent and 90 percent work for a large range of problems.

4. Mutation rates should be near $1/$*(the number of bits)*, so that each individual undergoes about one mutation on average per generation.

5. Once the average fitness of the population does not change significantly after ten generations, the population has converged on the solution. At this point, stop the GA and study the population of solutions.

17.4 ga_eng: a Genetic Algorithm Engine

The ga_eng engine is a general purpose reusable GA engine that can quickly and easily be adapted to solve a variety of problems. This section presents its key elements. The full source code for ga_eng is on the book's web site.

From the preceding sections, you can tell that a GA maintains a large amount of information as it transitions from one generation to the next. What is the state information that the engine needs to track? Below is a list of the most obvious things to record:

- n − the size of the population
- l − the length of the individual's bit representation
- p_c − the probability of crossover mating
- p_m − the probability of mutation
- population − a list of a current population's individuals

Given the above state information, what can a GA engine do?

- init() − initialize the state information and population
- evolve() − move a population from one generation to the next
- tselect() − perform tournament of selection on the population
- stats() − collect statistics about the current population to monitor progress

The fitness function

A key application-specific interface issue is the fitness function. The user of the GA engine sets the fitness values of each member of the population. If that value is not set, the default value will be the average of the fitness of each of the parents. Each parent's contribution to the fitness is weighted by how many bits it contributed to the offspring. This has the nice property of not requiring that each individual's fitness be computed at every generation.

To use the GA engine, the programmer supplies an application-specific fitness function f(x) that is applied to each individual of the population every generation. Given a binary string s, f(s) would return a numeric fitness value.

Methods and attributes of class ga_eng

Class ga_eng implements the engine. It provides two public methods, evolve() and set_params(), as well as eight private methods:

```
method evolve()
method set_params(fitness_func, popsize, lchrom, pcross, pmutation, log_file)
method tselect()
method crossover2(parent1, parent2)
method generation()
method stats(Pop)
method report(Pop)
method random_chrom()
method initpop()
method init()
```

The set_params() method sets the fitness function, the population size, the length of the chromosomes, the probability of crossover, the probability of mutation, and the log file for generating a trace of a run. The constructor ga_eng() takes the same input parameters as set_params() but it also re-initializes the engine by creating a new population from scratch. The evolve() method moves the population from one generation to the next.

The tselect() method operates on the whole population by performing tournament selection. The crossover2() method takes two individuals and returns two new individuals in a list after doing two-point crossover. The stats() method collects statistics on the given population, and report() writes the results out to the log file if it is not set to &null. The init() and initpop() methods initialize the GA.

You might be asking what is a population of *individuals*? The key pieces of information about an individual are its chromosomes, which are represented as a string of zeros and ones. For implementation reasons, we bundle the following information for each individual using the following Unicon record:

```
record individual(chrom, fitness, parent1, parent2, xsite)
```

This stores the fitness values, the index of two parents, and the crossover sites from when the parents were mated.

The class ga_eng makes use of the following instance variables:

- oldpop, newpop − two populations; selection goes from oldpop into newpop
- popsize, lchrom − the population size and the length of the chromosomes
- gen, maxgen − current and max generation number
- pcross, pmutation − the probability of crossover and mutation

- sumfitness − the sum of the fitness of the entire population
- nmutation − number of mutation in the current generation
- ncross − number of crossovers (or matings) in the current generation
- avg, max, min − average, maximum, and minimum fitness in the population
- best − the location of the individual with the highest fitness
- log_file − a text file where statistics are written during a GA run
- fitness_func − the user-supplied fitness function. It reads a string of zeros and ones and returns a number representing the fitness.

A Closer Look at the evolve() method

Space limitations preclude discussing every method of ga_eng, but evolve() is a method that is called by user code, and defines the basic architecture of the engine. The evolve() method initializes the population the first time the method is invoked by each instance of ga_eng. After initialization and in subsequent calls, evolve() does three things: it collects statistics, writes the results to a log file, and then evolves the population for one generation. The method generation() then becomes the focus of activity. The code for evolve() is:

```
method evolve()
    if /initialized := 1 then {
        gen := 0
        init()
        statistics(oldpop)
        if \log_file then report(oldpop)
        }
    gen +:= 1
    generation()
    statistics(newpop)
    if \log_file then report(newpop)
    oldpop := newpop
end
```

Generation consists of three high-level operations. Tournament selection is performed via a call to tselect(), which makes copies of selected individuals from oldpop to newpop. After this, all operations take place on individuals in newpop. The next two operations are performed in a loop, on pairs of individuals. The crossover2() method does the mating; it encapsulates the relatively low-level operation of mutation. The last high-level operation that a GA does is call the user supplied fitness_func to assign a fitness value to each individual in the new population. The GA keeps track of only two generations for the evolve() method

to continue as long as needed. Once each individual has been evaluated, that generation is complete. The oldpop variable is assigned the value of the newpop, and the process is ready to start again. Listing 17-1 shows the code for the method generation():

Listing 17-1
A method for producing a new generation.

```
method generation()
   local j := 1, mate1, mate2, jcross, kids, x, fitness1, fitness2, selected
   newpop    := list(popsize)
   nmutation := ncross := 0
   selected  := tselect()
   repeat  {
     mate1 := selected[j]
     mate2 := selected[j+1]
     kids := crossover2(oldpop[mate1].chrom, oldpop[mate2].chrom )
     fitness1 := fitness_func(oldpop[mate1].chrom)
     fitness2 := fitness_func(oldpop[mate2].chrom)
     newpop[j]:= individual(kids[1], fitness1, mate1, mate2, kids[3])
     newpop[j+1] := individual(kids[2], fitness2, mate1, mate2, kids[3])
     if j > popsize then break
     j +:= 2
     }
end
```

Using ga_eng

GAs are extremely robust. It is easy to create a buggy GA that works so well that the bugs go undetected. Masking of bugs by robust algorithms is not unique to GA; it occurs in many numerical algorithms. To prevent this, the following code tests the GA on a simple problem where having one bit increases the fitness values by one. This is a ready to compile and run GA application, albeit a very simple one. As you can see, ga_eng() and evolve() are all the interface methods you need to build a complete genetic algorithm. Listing 17-2 shows all the code needed to use the GA engine for a simple test problem.

Listing 17-2
Using **ga_eng()** for a simple test problem

```
procedure decode(chrom)
   local i := 0
   every !chrom == "1" do i +:= 1
   return i
end
```

```
# a simple test of the engine, fitness is based on number of 1 bits in chrom
procedure main()
    log_file  := open("test_ga.log", "w") | stop("cannot open log_file.log")
    ga := ga_eng(decode, 100, 20, 0.99, 1.0/real(20), log_file)
    every 1 to 100 do ga.evolve()
    write(log_file, "best location => ", ga.best)
    write(log_file, "best fitness  => ", ga.newpop[ga.best].fitness)
end
```

Log files

The log file test_ga.log contains a lot of statistics. Listing 17-3 is a fragment of the log file that is generated. The complete log file has over twelve thousand lines.

Listing 17-3
A trace of the GA for a simple test problem

```
Log_File for Genetic Algorithm (GA)

 Population size          = 100
 Chromosome length        = 20
 Maximum # of generations =
 Crossover probability    = 0.99
 Mutation probability     = 0.05

 Initial population maximum fitness =  5.00e-1
 Initial population average fitness =  5.00e-1
 Initial population minimum fitness =  5.00e-1
 Initial population sum of  fitness =  5.00e1
-------------------------------------------------------------
Population Report
                        Generation 0

      #    parents       xsite   chromo               fitness
-------------------------------------------------------------
      1) (  0,   0)       0    00010001110001000010  5.00e-1
      2) (  0,   0)       0    11011101011010001100  5.00e-1
      3) (  0,   0)       0    01101100000110100111  5.00e-1
.........
      100) (  0,   0)       0    11011100110011001010  5.00e-1
-------------------------------------------------------------
Statistics:
min =  5.00000000e-1  avg =  5.00000000e-1  max =  5.00000000e-1
 no. of mutations        = 0
```

```
 no. of crossovers       = 0
 location of best chromo = 1
-------------------------------------------------------------
dateline = Wednesday, January 27, 1999  10:37 pm
.................
Population Report
                        Generation 100

      #    parents      xsite   chromo              fitness
-------------------------------------------------------------
      1) (  1,   2)   15:21    00011011011000010100  9.19e0
      2) (  1,   2)   15:21    11011001011111100010  1.08e1
      3) ( 90,  72)   14:18    10011111111111111111  1.78e1
.....
      100) ( 57,  27)   13:15    11111010110110111110  1.78e1
-------------------------------------------------------------
Statistics:
min =  9.19999999e0   avg =  1.77200000e1   max =  1.99500000e1
 no. of mutations        = 111
 no. of crossovers       = 51
 location of best chromo = 46
-------------------------------------------------------------
dateline = Wednesday, January 27, 1999  10:37 pm
best location => 46
best fitness  => 19.95
```

17.5 Color Breeder: a GA Application

Normally, the human eye is capable of distinguishing millions of different colors. If you have a device capable of producing a large number of different colors such as a PC with a color monitor or color printer, then finding exactly the color that you have in mind can be a tricky task. The color space is quite large. For example, imagine that you want to create a Web page with a blue background. But this is no ordinary blue; you want a blue that is like the blue you saw on the Mediterranean skyline on your last trip to Greece.

One option is to look at a color wheel and click on the part that represents the blue that you have in mind. Unfortunately, the color you choose is typically only a very small part of the color wheel. Also the color is surrounded by many different colors that may be distracting or misleading in your evaluation. When you use the same color for the entire background, you get a different sense of the color.

A second option is to tinker with numeric color codes by hand. Through trial and error, by examining a large number of colors you can select the right one. This can be very time consuming and you may become impatient and only experiment with a small number of

colors, in which case you settle for a quick approximation of the color you intended.

The following *color breeder* program has properties from both of the color wheel and the tinkering methods. It uses a GA to explore the problem space. The program cb first displays sixteen randomly generated colors. The fitness of an individual color is based entirely on user preference. Using scrollbars, the user ranks the individuals, and then hits a breed button to generate a new population of colors.

Scrollbars sbar[*i*] are indexed with *i*, where *i* runs from 1 to 16. The value of the scrollbar is inverted because scrollbar values correspond to the y-axis by default, which grows from top to bottom; when the tab is lowered the value of scrollbar goes up. The user of cb sets higher tabs to indicate higher fitness.

```
every i := 1 to 16 do {
   f := VGetState(vidgets["sbar"||i])
   ga.newpop[i].fitness := 1 - f
}
```

Gradually the population evolves from a set of random colors to a set of colors that look alike, with ever so slight variations on that Mediterranean sky blue you were thinking of. You can then save a snapshot of the screen in a GIF format if you wish, and can see the numeric codes that represents the color by clicking on a color.

Figure 17-2 is a screenshot of cb. The higher the user slides the scrollbar tab, the more fit the color. There are three color resolutions to select from: 12-, 24-, and 48-bit. The bits are equally divided into three segments, each representing a color intensity. There are two mutually exclusive advanced modes: patterns and text. Patterns are square bit patterns that specify the mix of foreground and background colors in three resolutions: 6x6, 8x8, and 10x10. In text mode, the foreground color is displayed as text atop the background.

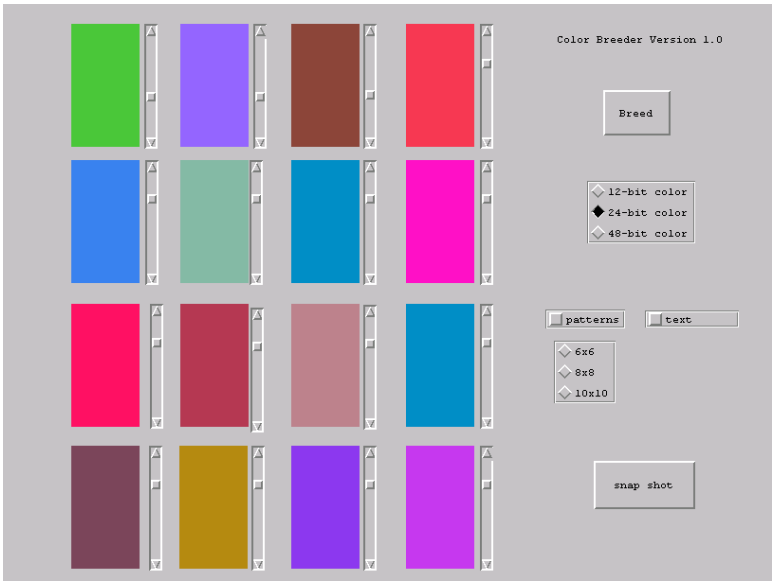


Figure 17-2: cb - a genetic algorithm for picking colors.

Breeding textures

The user can turn on bi-level patterns of the form: *width, #data* as described in Chapter 7. A bi-level pattern is tiled using the foreground and background color to fill an area; a 1 in the pattern denotes the foreground, and a 0 denotes the background. The patterns come in three resolutions: 8x8, 10x10, and 12x12. In low resolutions it is hard to find interesting patterns; in high resolution, most patterns look like a random mixture and are hard to evolve into something more structured. Figure 17-3 (left) shows cb in pattern mode.

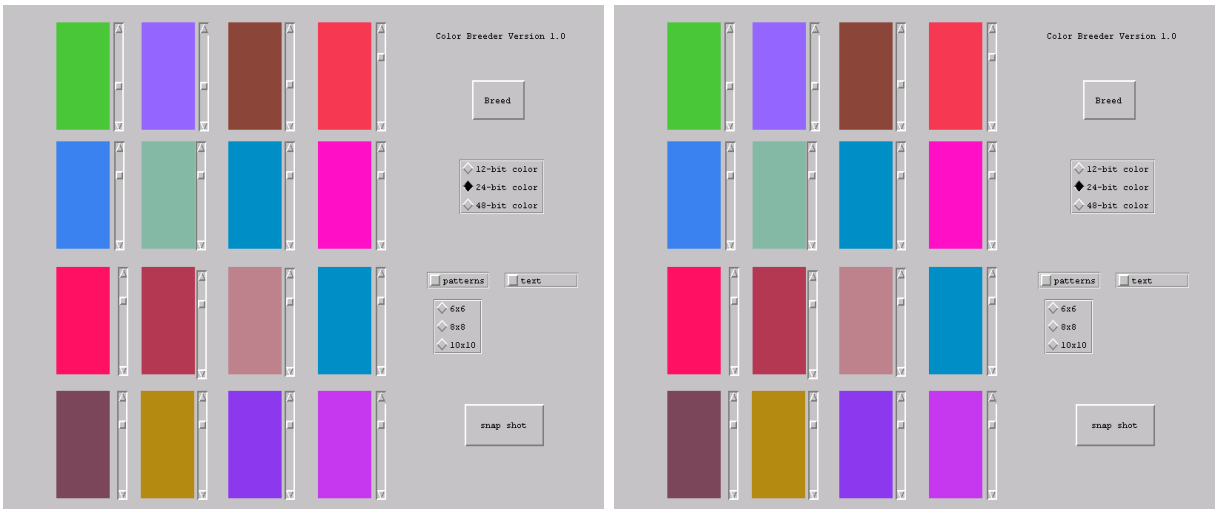


Figure 17-3: cb in pattern mode (left) and text mode (right)

The user can click on the snap shot button to create a GIF image file that is a snapshot of the whole screen. If you left click on a color region, the color and patterns specification will be displayed in a pop window. For the sake of compactness, the colors and patterns are represented with hexadecimal strings as opposed to binary strings.

17.6 Picking Colors for Text Displays

Perhaps the most fun use of cb is to pick colors for your Web pages. Figure 17-3 (right) is a snapshot of cb in text mode. In text mode with 24-bit color, the user can right click on a color region to generate a sample HTML document such as the one shown in Figure 17-4 that demonstrates how to incorporate the selected color combination inside of a Web page. It is also possible to get the hexadecimal representation of a pattern or a color by right clicking on it.

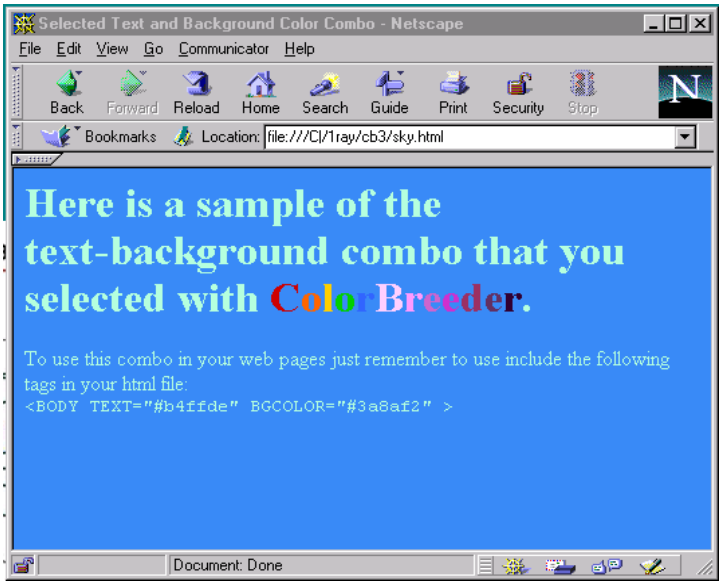


Figure 17-4: A sample HTML page generated by cb

Summary

GAs are a branch of evolutionary computation that allow a general-purpose optimization technique. The three main GA operations are selection, mating, and mutation. Object-oriented techniques enable a generic GA engine that can be adapted to a large variety of problems. By making the engine very general purpose it is possible to create applications with novel properties like using user preferences to set the fitness of a color!

Chapter 18

Object-oriented User Interfaces

Many applications interact with users through a graphical user interface. While Unicon's graphics facilities are excellent for drawing to the screen, the standard elements of graphical user interfaces are not built-in to the language.

This chapter presents a user interface class library developed by Robert Parlett. Object-oriented design is used to reduce complexity, resulting in an elegant, extensible library accessed by importing the package gui. Although this chapter describes the components provided by the GUI toolkit, you may need to consult Graphics Programming in Icon [Griswold98] to create advanced custom user interfaces with application-specific graphics.

The GUI classes are supported by a tool named ivib that allows interfaces to be constructed by drawing a dialog on the screen interactively. ivib generates a Unicon program that can be filled in to create an application. This chapter shows how to:

- Construct programs that employ a graphical user interface.
- Manipulate the attributes of objects such as buttons and scrollbars.
- Draw a program's interface using Unicon's improved visual interface builder.

18.1 A Simple Dialog Example

Object-orientation seems to be a big help in designing graphical user interfaces. The best way to see how the GUI classes work is to try out a simple example program. Listing 18-1 shows the source code in full; the code is explained in detail below.

Listing 18-1
The TestDialog Program

```
import gui
$include "guih.icn"
```

```
class TestDialog : Dialog()
    method component_setup()
      local b, l := Label("label=Click to close","pos=50%,33%", "align=c,c")
      add(l)
      b := TextButton("label=Button", "pos=50%,66%", "align=c,c")
      b.connect(self, "dispose", ACTION_EVENT)
      add(b)
      attrib("size=215,150", "bg=light gray", "font=serif", "resize=on")
    end
end

procedure main()
    TestDialog().show_modal()
end
```

If the program is stored in a file testdialog.icn, the following command will compile it:

```
unicon testdialog
```

The result should be an executable file called testdialog. This example program, along with several others, is in the guidemos directory of the Unicon distribution. Run this program, and the window shown in Figure 18-1 appears; it closes when the button is clicked.

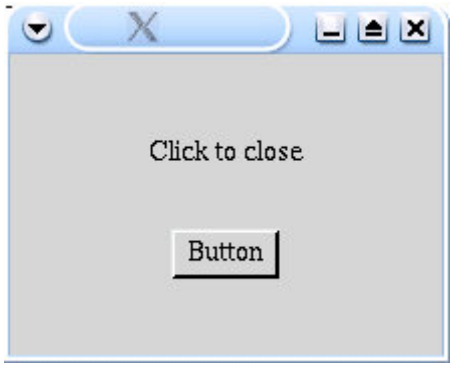


Figure 18-1: TestDialog window

This example program begins by declaring a class, TestDialog. The line

```
class TestDialog : Dialog()
```

indicates that TestDialog is a subclass of the class Dialog that is defined in the toolkit. This subclass relationship is true of all dialog windows.

Adding Components and Attaching Listeners

The remainder of the class's code is contained in a single method, component_setup(). This method is invoked by the toolkit; it is a convenient place to setup the dialog's content. Inside the component_setup() method is the code that adds the label to the dialog:

```
l := Label("label=Click to close","pos=50%,33%", "align=c,c")
add(l)
```

This assigns l to a new Label object and sets the label string. The horizontal position is set to 50 percent of the window width and the vertical to 33 percent of the window height. The alignment of the object is centered both vertically and horizontally about the position. Finally, the label is added to the dialog with the line add(l).

The code to add a button is very similar, but a TextButton object is created rather than a Label object, and the vertical position is 66 percent of the window height.

The next line is more interesting:

```
b.connect(self, "dispose", ACTION_EVENT)
```

This adds a *listener* to the button, and tells the toolkit than whenever the button fires an ACTION_EVENT, which it will when it is pressed, the dispose() method in the class self, should be invoked. self of course refers to the TestDialog class, and dispose() is a method inherited from the base class Dialog, which simply closes the window. So all this means is that when the button is pressed, the dialog will close.

The next line sets the attributes of the dialog window, including its initial size. Try changing these values to experiment with other dialog styles.

```
attrib("size=215,150", "bg=light gray", "font=serif", "resize=on")
```

After the class comes a standard Icon main() procedure. This simply creates an instance of the dialog and invokes the method show_modal(). This call displays the dialog window and goes into the toolkit's event handling loop.

Positioning Objects

The button and label were positioned above using percentages of the window size. An object can also be positioned by giving an absolute position, or by giving a percentage plus or minus an offset. So the following are all valid position specifiers:

```
"100"
"10%"
"25%+10"
"33%-10"
```

Positions are often specified in constructors with x and y values separated by commas. By default, positions are relative to the top left corner of the object. The "align" attribute, which takes two alignment specifiers, changes this default. The first alignment specifier is an "l", "c", or "r", for left, center, or right horizontal alignment, respectively; the second is a "t", "c", or "b", for top, center, or bottom vertical alignment, respectively. Another attribute, "size", has specifiers that take the same format as the position attribute. Most of the toolkit objects default to sensible sizes, and the size attribute can often be omitted. For example, a button's size defaults fit its string label based on the font in use.

The method attrib(attribs...) in utility class SetFields implements attribute processing, mostly farming out work to special-purpose methods that can be called directly: set_pos(x,y), set_label(s), set_align(horizontal, vertical), and set_size(w, h). Other than these setter methods, Icon graphics attributes can be interspersed. For example, the attribute "bg=green" will set the object's background color.

Here are some examples of position, alignment, and size parameters, and a description of their meaning. In the call

```
attrib("pos=50%,100", "align=c,t", "size=80%,200")
```

the object is centered horizontally in the window, using 80 percent of the width; vertically its top starts at 100 and its height is 200 pixels. In contrast, the code

```
attrib("pos=100%,100%", "align=r,b", "size=50%,50%")
```

specifies that the object fills up the bottom right quarter of the window. The call

```
attrib("pos=33%+20,0%", "size=100,100%")
```

directs that the object's left hand side is at one-third of the window size plus 20 pixels; it is 100 pixels wide. It fills the whole window vertically.

18.2 A More Complex Dialog Example

Now it's time to introduce some more component types. Listing 18-2 shows our next example program in full.

Listing 18-2
SecondTest Program

```
import gui
$include "guih.icn"
class SecondTest : Dialog(
    text_list, table, list, text_field, oses, languages, shares
    )
```

```
#
# Add a line to the end of the text list
#
method put_line(s)
  local l := text_list.get_contents()
  put(l, s)
  text_list.set_contents(l)
  text_list.goto_pos(*l)
end

#
# Event handlers - produce a line of interest.
#
method handle_check_box_1(ev)
  put_line("Favorite OS is " || oses[1])
end
method handle_check_box_2(ev)
  put_line("Favorite OS is " || oses[2])
end
method handle_check_box_3(ev)
  put_line("Favorite OS is " || oses[3])
end
method handle_text_field(ev)
  put_line("Contents = " || text_field.get_contents())
end
method handle_list(ev)
  put_line("Favorite language is " || languages[list.get_selection()])
end
method handle_text_menu_item_2(ev)
  put_line("You selected the menu item")
end

#
# The quit menu item
#
method handle_quit(ev)
  dispose()
end

method handle_table(ev)
  local i := table.get_selections()[1]
  put_line(shares[i][1] || " is trading at " || shares[i][2])
end
```

```
method handle_table_column_1(ev)
  put_line("Clicked on column 1")
end
method handle_table_column_2(ev)
  put_line("Clicked on column 2")
end
#
# Invoked for components that may potentially want to handle an event
# (by firing an event to its listeners for example). A dialog is just
# another component. It can override this method to do any custom processing.
#
method handle_event(ev)
  put_line("Icon event " || ev)
  self.Dialog.handle_event(ev)
end

method component_setup()
  local menu_bar, menu, panel_1, panel_2, panel_3, panel_4, panel_5,
       label_1, label_2, label_3, label_4, label_5,
       quit_menu_item, text_menu_item_2,
       check_box_1, check_box_2, check_box_3,
       table_column_1, table_column_2, check_box_group

  #
  # Initialize some data for the objects.
  #
  oses := ["Windows", "Linux", "Solaris"]
  languages := ["C", "C++", "Java", "Icon"]
  shares := [["Microsoft", "101.84"], ["Oracle", "32.52"], ["IBM", "13.22"],
        ["Intel", "142.00"]]

  #
  # Set the attributes, then setup a simple menu system
  #
  attrib("size=490,400", "min_size=490,400", "font=sans",
       "bg=light gray","label=Second example", "resize=on")
  menu_bar := MenuBar()
  menu := Menu("label=File")
  quit_menu_item := TextMenuItem("label=Quit")
  quit_menu_item.connect(self, "handle_quit", ACTION_EVENT)
  menu.add(quit_menu_item)
  text_menu_item_2 := TextMenuItem("label=Message")
  text_menu_item_2.connect(self,"handle_text_menu_item_2",ACTION_EVENT)
```

```
menu.add(text_menu_item_2)
menu_bar.add(menu)
add(menu_bar)

#
# Set-up the checkbox panel
#
check_box_group := CheckBoxGroup()
panel_1 := Panel("pos=20,50", "size=130,130")
label_2 := Label("pos=0,0", "internal_alignment=l", "label=Favorite OS")
panel_1.add(label_2)
check_box_1 := CheckBox("pos=0,30")
check_box_1.set_label(oses[1])
check_box_1.connect(self, "handle_check_box_1", ACTION_EVENT)
check_box_group.add(check_box_1)
panel_1.add(check_box_1)
check_box_2 := CheckBox("pos=0,60")
check_box_2.set_label(oses[2])
check_box_group.add(check_box_2)
check_box_2.connect(self, "handle_check_box_2", ACTION_EVENT)
panel_1.add(check_box_2)
check_box_3 := CheckBox("pos=0,90")
check_box_3.set_label(oses[3])
check_box_group.add(check_box_3)
check_box_3.connect(self, "handle_check_box_3", ACTION_EVENT)
panel_1.add(check_box_3)
add(panel_1)
#
# The text-list of messages.
#
panel_2 := Panel("pos=220,50", "size=100%-240,50%-60")
label_1 := Label("pos=0,0", "internal_alignment=l", "label=Messages")
panel_2.add(label_1)
text_list := TextDisplay("pos=0,30", "size=100%,100%-30")
text_list.set_contents([])
panel_2.add(text_list)
add(panel_2)
#
# The table of shares.
#
panel_3 := Panel("pos=220,50%","size=100%-240,50%-40")
table := Table("pos=0,30","size=100%,100%-30", "select_one")
table.connect(self, "handle_table", SELECTION_CHANGED_EVENT)
table.set_contents(shares)
```

```
        table_column_1 := TableColumn("label=Company", "internal_alignment=l",
                                      "column_width=100")
        table_column_1.connect(self, "handle_table_column_1", ACTION_EVENT)
        table.add_column(table_column_1)
        table_column_2 := TableColumn("label=Share price", "internal_alignment=r",
                                      "column_width=100")
        table_column_2.connect(self, "handle_table_column_2", ACTION_EVENT)
        table.add_column(table_column_2)
        panel_3.add(table)
        label_5 := Label("pos=0,0", "internal_alignment=l", "label=Shares")
        panel_3.add(label_5)
        add(panel_3)
        #
        # The drop-down list of languages.
        #
        panel_4 := Panel("pos=20,190", "size=180,50")
        list := List("pos=0,30", "size=100,")
        list.connect(self, "handle_list", SELECTION_CHANGED_EVENT)
        list.set_selection_list(languages)
        panel_4.add(list)
        label_3 := Label("pos=0,0", "internal_alignment=l", "label=Favorite language")
        panel_4.add(label_3)
        add(panel_4)
        #
        # The text field.
        #
        panel_5 := Panel("pos=20,280", "size=180,50")
        label_4 := Label("pos=0,0", "internal_alignment=l", "label=Enter a string")
        panel_5.add(label_4)
        text_field := TextField("pos=0,30", "size=130,", "draw_border=t")
        text_field.connect(self, "handle_text_field", TEXTFIELD_CHANGED_EVENT)
        panel_5.add(text_field)
        add(panel_5)
    end
end
#
# Simple main procedure just creates the dialog.
#
procedure main()
    SecondTest().show_modal()
end
```

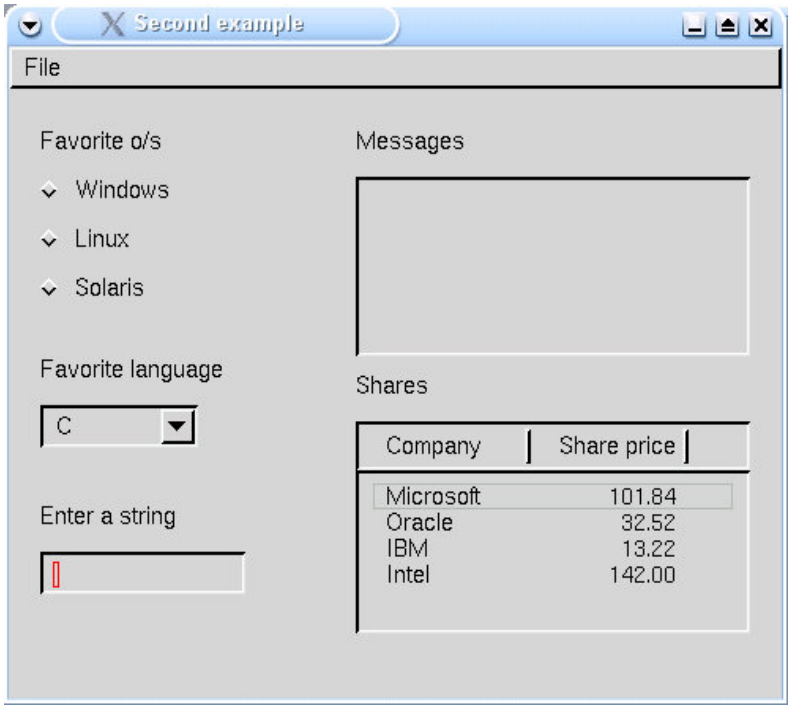


Figure 18-2: SecondTest window

Constructing Component Hierarchies

Examine this program's the component_setup() method at the end; it initializes some data and sets attributes. This includes built-in graphics attributes, as well as the minimum size, which is an attribute of the Dialog class. The next part creates a menu bar structure. Menus are presented in detail later in this chapter, but notice that this code creates two text menu items within a Menu object, which is itself within a MenuBar object, which is added to the dialog. Both menu items are connected to event handler methods.

The next section sets up three check boxes, placed with the label "Favorite OS" in a Panel, a container that allows objects to be treated as a whole. Objects within a Panel have their size and position computed relative to the Panel rather than the window. For example, the first Label object is positioned with "pos=0,0". This places it at the top left-hand corner of the Panel. Percentage specifications relate to the enclosing Panel.

Each CheckBox is a separate object. To make the three check boxes behave as a group so that when one is checked another is unchecked, they are placed in a CheckBoxGroup object; this just "brackets" them together. When grouped together in this way the checkboxes are called *radio buttons* because they work like the tuner buttons found on old car radios. Note that each CheckBox is added to the CheckBoxGroup *and* the Panel.

The next section is a Panel that holds a label ("Messages") and a TextList object. In this case the object holds a list of message strings that scroll by like a terminal window.

A TextList object can also be used for selecting one or more items from a list of strings and there is an EditableTextList, which can be used for editing text.

The third panel contains a label ("Shares"), and a Table object, which is used for displaying tabular data. Note that class Table has nothing to do with Icon's table data type. Adding a TableColumn object to the table sets up each column. A table column's initial column width is specified with attribute column_width and the alignment of the column's contents is set with attribute internal_alignment. Attribute select_one configures the table to allow one row to be highlighted at a time. The default is not to allow highlighting of rows; the other option is to allow several to be highlighted at once with select_many.

The next panel contains another label ("Favorite language") and a drop-down list of selections, created using the List class. The selections are set using the set_selection_list() method. The final panel contains a label ("Enter a string") and a TextField object, which is used to obtain entry of a string from the keyboard.

Several of the components are connected to event handlers in the class. Each handler method adds a line to the list of strings in the TextList by calling the put_line() method. The text list thus gives an idea of the events being produced by the toolkit. The exception is the menu item Quit, which exits the program.

This dialog overrides the handle_event() method that is invoked by the toolkit for any component (including dialogs) that may elect to handle an event. In this case, the dialog just prints out the Icon event code for the particular event. This method also checks for the Alt-q keyboard combination, which closes the dialog.

More About Event Handling

As shown above, components generate events when something of interest happens. For example, a button generates an ACTION_EVENT when it is pressed. Different components generate different events, but some basic events are generated by all components:

MOUSE_PRESS_EVENT	a mouse press within the component's region.
MOUSE_DRAG_EVENT	a mouse drag within the component's region.
MOUSE_RELEASE_EVENT	a mouse release within the component's region.

For any non-mouse events, the Dialog class fires an ICON_EVENT.

Events are passed to listeners in a Notification object. This object contains three fields, with corresponding getter methods, as follows:

get_source()	Returns the component which fired the event.
get_type()	Returns the type code, eg ICON_EVENT
get_param()	Returns an arbitrary parameter depending on the type. In nearly all cases this is the original underlying Icon graphics event.

The get_param() method is necessary, for example, to distinguish between a left mouse

click and a right mouse click on a MOUSE_RELEASE_EVENT; for instance

```
method on_release(ev)
  if ev.get_param() === &rrelease then
    ... process right mouse up
end
```

18.3 Containers

Containers are components that contain other components. The Dialog class is a container, as is the Panel class seen in the last example. Two other useful container objects in the standard toolkit are TabSet and OverlaySet.

TabSet

This class contains several tabbed panes, any one of which is displayed at any given time. The user switches between panes by clicking on labeled tabs at the top of the object. The TabSet contains several TabItems, each of which contains the components for that particular pane. To illustrate this, Listing 18-3 presents a simple example of a TabSet that contains three TabItems, each of which contains a single label.

Listing 18-3
TabSet Program

```
import gui
$include "guih.icn"
#
# Simple example of a TabSet
#
class Tabs : Dialog(quit_button)
    method change(e)
      write("The tabset selection changed")
    end

    method component_setup()
      local tab_set, tab_item_1, tab_item_2, tab_item_3
      attrib("size=355,295", "font=sans", "bg=light gray",
            "label=TabSet example", "resize=on")
      #
      # Create the TabSet
      tab_set := TabSet("pos=20,20", "size=100%-40,100%-80")
      #
      # First, second, and third panes
```

```
        #
        tab_item_1 := TabItem("label=Pane 1")
        tab_item_1.add(Label("pos=50%,50%", "align=c,c", "label=Label 1"))
        tab_set.add(tab_item_1)
        tab_item_2 := TabItem("label=Pane 2")
        tab_item_2.add(Label("pos=50%,50%", "align=c,c", "label=Label 2"))
        tab_set.add(tab_item_2)
        tab_item_3 := TabItem("label=Pane 3")
        tab_item_3.add(Label("pos=50%,50%", "align=c,c", "label=Label 3"))
        tab_set.add(tab_item_3)
        tab_set.set_which_one(tab_item_1)
        tab_set.connect(self, "change", SELECTION_CHANGED_EVENT)
        add(tab_set)
        #
        # Add a quit button; close the dialog when it is pressed
        quit_button := TextButton("pos=50%,100%-30", "align=c,c", "label=Quit")
        quit_button.connect(self, "dispose", ACTION_EVENT)
        add(quit_button)
        connect(self, "dispose", CLOSE_BUTTON_EVENT)
    end
end
procedure main()
    Tabs().show_modal()
end
```

The resulting window is shown in Figure 18-3:

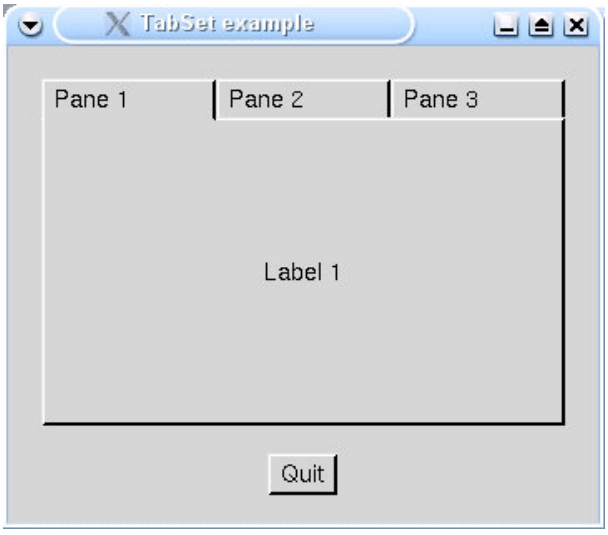


Figure 18-3: TabSet example window

One interesting point in this dialog is the line:

```
connect(self, "dispose", CLOSE_BUTTON_EVENT)
```

A dialog generates an event whenever the close button is pressed. Connecting this event to the dispose method configures the dialog to close when this button is pressed.

OverlaySet

An OverlaySet is like a TabSet, but the pane on display is under program control; there are no tabs to click. Instead of adding items to TabItem structures, OverlayItem objects are used. An empty OverlayItem can be used when the area should be blank. The current OverlayItem on display is set by set_which_one(x), where x is the desired OverlayItem.

18.4 Menu Structures

The toolkit provides the standard building blocks required to create a menu system (see Table 18-1). Customized components can be added; this is discussed later.

Table 18-1
Standard Menu System Components

Component	Description
MenuBar	The menu area along the top of the window, containing one or more Menus.
MenuButton	A floating menu bar containing one Menu.
Menu	A drop down menu pane containing other Menus or menu components.
TextMenuItem	A textual menu item.
CheckBoxMenuItem	A checkbox in a menu which can be part of a CheckBoxGroup if desired.
MenuSeparator	A vertical separation line between items.

Items in a Menu can have left and right labels as well as customized left and right images. To see how this all fits together, Listing 18-4 shows our next example program.

Listing 18-4
A Menu Example Program

```
import gui
$include "guih.icn"

class MenuDemo : Dialog()
    method component_setup()
        local file_menu, menu_bar, check_box_group, text_menu_item,
```

```
        labels_menu, images_menu, checkboxes_menu, group_menu,
        alone_menu, menu_button, check_box_menu_item, button_menu

    attrib("size=426,270", "font=sans", "bg=light gray", "label=Menu example")
check_box_group := CheckBoxGroup()

#
# Create the menu bar. The position and size default to
# give a bar covering the top of the window.
# The first menu ("File") - just contains one text item.
menu_bar := MenuBar()
file_menu := Menu("label=File")
text_menu_item := TextMenuItem("label=Quit")
text_menu_item.connect(self, "dispose", ACTION_EVENT)
file_menu.add(text_menu_item)
menu_bar.add(file_menu)

#
# The second menu ("Labels") - add some labels, followed by
# a separator and another text item
labels_menu := Menu("label=Labels")
labels_menu.add(TextMenuItem("label=One"))
labels_menu.add(TextMenuItem("label=Two", "label_left=ABC"))
labels_menu.add(MenuSeparator())
labels_menu.add(TextMenuItem("label=Three", "label_right=123"))
#
# A sub-menu in this menu, labeled "Images", contains
# three text items with custom images. The rather unwieldy
# strings create a triangle, a circle and a rectangle.
images_menu := Menu("label=Images")
text_menu_item := TextMenuItem("label=One")
text_menu_item.set_img_left("15,c1,~~~~~~0~~~~~~_
  ~~~~~~0~~~~~~~~~~000~~~~~~~~~~000~~~~~_
  ~~~~00~00~~~~~~~~00~00~~~~~~~~00~~00~~~~_
  ~~~~00~~~00~~~~~~00~~~~00~~~~~00~~~~00~~~_
  ~~00~~~~~~~00~~~~00~~~~~~00~~~00~~~~~~~~00~_
  ~0000000000000~000000000000000")
images_menu.add(text_menu_item)
text_menu_item := TextMenuItem("label=Two")
text_menu_item.set_img_left("15,c1,~~~~~000~~~~~_
  ~~~~0000000~~~~~000~~~~000~~~~00~~~~~~00~~_
  ~00~~~~~~~~00~~0~~~~~~~~~0~00~~~~~~~~~00_
  00~~~~~~~~~0000~~~~~~~~~00~0~~~~~~~~~0~_
  ~00~~~~~~~~00~~~00~~~~~~00~~~~000~~~~000~~_
```

```
  ~~~~0000000~~~~~~~~~000~~~~~~")
images_menu.add(text_menu_item)
text_menu_item := TextMenuItem("label=Three")
text_menu_item.set_img_left("15,c1,_
  00000000000000000000000000000000_
  00~~~~~~~~~~0000~~~~~~~~~~0000~~~~~~~~~~00_
  00~~~~~~~~~~0000~~~~~~~~~~0000~~~~~~~~~~00_
  00~~~~~~~~~~0000~~~~~~~~~~0000~~~~~~~~~~00_
  00~~~~~~~~~~0000~~~~~~~~~~00_
  00000000000000000000000000000000")
images_menu.add(text_menu_item)
labels_menu.add(images_menu)
menu_bar.add(labels_menu)

#
# The third menu ("Checkboxes")
# Sub-menu - "Group" - two checkboxes in a checkbox group.
#
checkboxes_menu := Menu("label=Checkboxes")
group_menu := Menu("label=Group")
check_box_menu_item := CheckBoxMenuItem("label=One")
check_box_group.add(check_box_menu_item)
group_menu.add(check_box_menu_item)
check_box_menu_item := CheckBoxMenuItem("label=Two")
check_box_group.add(check_box_menu_item)
group_menu.add(check_box_menu_item)
checkboxes_menu.add(group_menu)
#
# Sub-menu - "Alone" - two checkboxes on their own
#
alone_menu := Menu()
alone_menu.set_label("Alone")
check_box_menu_item_3 := CheckBoxMenuItem("label=Three")
alone_menu.add(check_box_menu_item_3)
check_box_menu_item_4 := CheckBoxMenuItem("label=Four")
alone_menu.add(check_box_menu_item_4)
checkboxes_menu.add(alone_menu)
menu_bar.add(checkboxes_menu)
add(menu_bar)
#
# Finally, create a menu button - a mini floating menu with
# one menu inside it.
#
menu_button := MenuButton("pos=350,50%", "align=c,c")
```

```
        #
        # This is the menu, its label appears on the button.  It just
        # contains a couple of text items for illustration purposes.
        #
        button_menu := Menu("label=Click")
        button_menu.add(TextMenuItem("label=One"))
        button_menu.add(TextMenuItem("label=Two"))
        menu_button.set_menu(button_menu)
        add(menu_button)
    end
end

procedure main()
    MenuDemo().show_modal()
end
```

The output of this program with the middle menu and its submenu open appears in Figure 18-4.

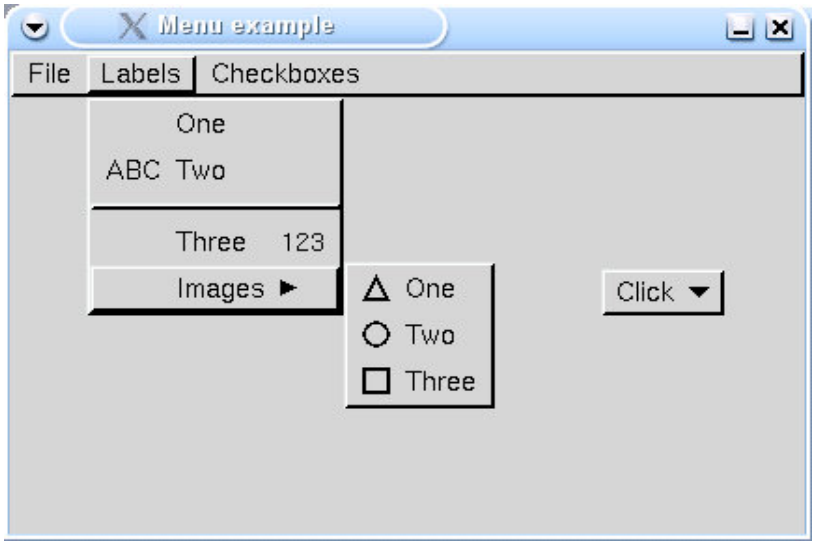


Figure 18-4: Menus

18.5 Other Components

This section gives examples of some components that have not yet been encountered. For full details of how to use these classes, and the available methods and options, please see the GUI class library reference in Appendix C.

18.5.1 Trees

The toolkit contains a tree component, which can be used to represent hierarchical data. To use it, it is necessary to create a tree-like data structure of Node objects. Children are added to a Node using its add() method. For example:

```
root := Node("label=Root")
child1 := Node("label=Child1")
child2 := Node("label=Child2")
root.add(child1)
root.add(child2) # ...etc
```

After setting up the tree of Nodes, the root is passed to the Tree for display:

```
tree := Tree("pos=0,0", "size=100,100") tree.set_root_node(root)
```

The tree data structure can change dynamically over time. When this occurs, the Tree must be notified of the change by invoking the tree_structure_changed() method.

The Tree class generates events when the selected Node (or Nodes) changes, and also when part of the tree is expanded or collapsed by the user.

The next example uses a Tree with a Table and a Sizer to provide a file system explorer program. The Sizer is a narrow area between the tree and the table which can be dragged to resize both dynamically. Because of the toolkit's relatively simple layout mechanism, the resizing code in handle_sizer() is quite awkward.

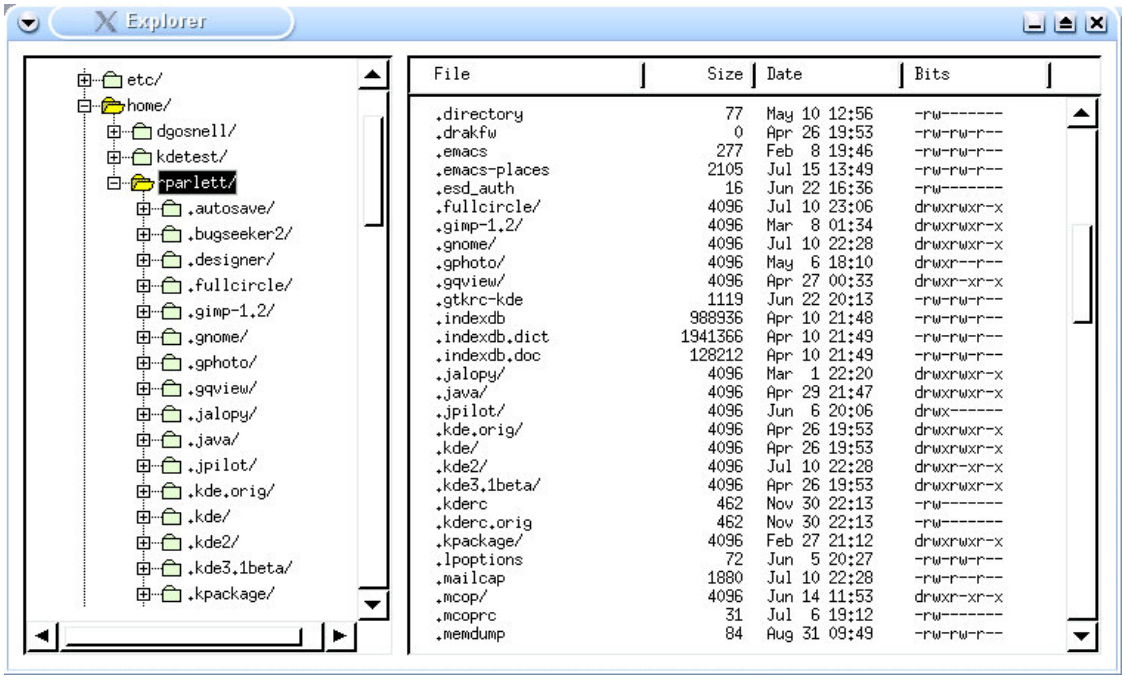


Figure 18-5: Explorer

Listing 18-5
The Explorer Program

```
import gui

$include "keysyms.icn"
$include "guih.icn"

#
# A very simple filesystem explorer with a tree and a table.
#
class Explorer : Dialog(tree, sizer, tbl)

    #
    # Given a Node n, get the full file path it represents by
    # traversing up the tree structure to the root.
    #
    method get_full_path(n)
      local s := ""
      repeat {
        s := n.get_label() || s
        n := n.get_parent_node() | break
        }
      return s
    end

    #
    # Invoked when a sub-tree is expanded (ie: the little + is
    # clicked).  An expansion event also includes contractions too
    #
    method handle_tree_expansion()
      local n := tree.get_last_expanded()
      #
      # Check whether it was an expansion or a contraction.  If
      # an expansion, load the subtree and refresh the tree.
      #
      if n.is_expanded() then {
        load_subtree(n)
        tree.tree_structure_changed()
        }
    end

    #
```

```
# Invoked when a row in the tree is selected (or de-selected).
# If something is selected, load the table. We may not have something
# selected if the user contracted the parent node of the selected node.
#
method handle_tree_selection()
  local n
  if n := tree.object_get_selections()[1] then
    load_table(n)
end

#
# Given a Node n, load its children with the sub-directories.
#
method load_subtree(n)
  local name, r1, dir_list := get_directory_list(get_full_path(n))
  n.clear_children()
  every name := !dir_list[1] do {
    if (name ~== "./") & (name ~== "../") then {
      r1 := Node("always_expandable=t")
      r1.set_label(name)
      n.add(r1)
    }
  }
end

#
# Given a Node n, load the table with the sub-files and sub-directories.
#
method load_table(n)
  local s := get_full_path(n), l := [ ], t := get_directory_list(s)
  every el := !sort(t[1] ||| t[2]) do {
    p := stat(s || el) | stop("No stat")
    put(l, [el, p.size, ctime(p.mtime)[5:17], p.mode])
    }

  tbl.set_contents(l)
  tbl.goto_pos(1, 0)
end

#
# The sizer has moved, so reset the sizes and positions of the
# table, tree and sizer.  Then call resize() to reposition
# everything.
#
```

```
method handle_sizer(ev)
  result_x := sizer.get_curr_pos()
  tree.set_size(result_x - 10, tree.h_spec)
  sizer.set_pos(result_x, sizer.y_spec)
  tbl.set_pos(result_x + 10, tbl.y_spec)
  tbl.set_size("100%-" || string(result_x + 20), tbl.h_spec)
  resize()
end

#
# Catch Alt-q to close the dialog.
#
method quit_check(ev)
  if ev.get_param() === "q" & &meta then dispose()
end

#
# Override resize to set the sizer's min/max locations.
#
method resize()
  self.Dialog.resize()
  sizer.set_min_max(135, get_w_reference() - 160)
end

method component_setup()
  local root_node

  attrib("size=750,440", "resize=on", "label=Explorer")
  connect(self, "dispose", CLOSE_BUTTON_EVENT)
  connect(self, "quit_check", ICON_EVENT)
  tree := Tree("pos=10,10", "size=250,100%-20", "select_one")
  tree.connect(self, "handle_tree_expansion", TREE_NODE_EXPANSION_EVENT)
  tree.connect(self, "handle_tree_selection", SELECTION_CHANGED_EVENT)
  add(tree)

  tbl := Table("pos=270,10", "size=100%-280,100%-20", "select_none")
  tbl.add_column(TableColumn("label=File", "column_width=150"))
  tbl.add_column(TableColumn("label=Size", "column_width=75", "internal_alignment=r"))
  tbl.add_column(TableColumn("label=Date", "column_width=100"))
  tbl.add_column(TableColumn("label=Bits", "column_width=100"))
  add(tbl)

  sizer := Sizer("pos=260,10", "size=10,100%-20")
  sizer.connect(self, "handle_sizer", SIZER_RELEASED_EVENT)
```

```
        add(sizer)

        #
        # Initialize the tree data structure.
        #
        root_node := Node("label=/")
        load_subtree(root_node)
        tree.set_root_node(root_node)
        tree.object_set_selections([root_node])
        load_table(root_node)
    end
end

procedure main()
    Explorer().show_modal()
end
```

18.5.2 Borders

This class provides decorative borders. Optionally, a single other component can be the title of the Border. This would normally be a Label object, but it could also be a CheckBox or an Icon, or whatever is desired. The set_title(c) method is used to set the title. Here is a program fragment to create a border with a label as its title:

```
b := Border()
#
# Add a Label as the title
#
l := Label("label=Title String")
b.set_title(l)
add(b)
```

The Border class acts as a container (like a Panel), and so objects may be placed within it in the same way.

18.5.3 Images and icons

The toolkit supports both images loaded from files and bitmap icons defined by strings. Images stored in files are manipulated using the Image class. The method set_filename(x) specifies the location of the file to be displayed. The image is scaled down to the size of the object, and optionally may be scaled up if it is smaller. A border may be used if desired.

Icons are created using the Icon class. The icon string is set using the set_img() method; again a border can be used if desired. Finally, the IconButton class lets icons serve as buttons, producing events when they are clicked.

18.5.4 Scroll bars

Horizontal and vertical scroll bars are available with the ScrollBar class. Scroll bars can be used either in the conventional way, in which the button in the bar represents a page size, and the whole bar represents a total, or as a slider in which case the button simply moves over a specified range of numbers.

18.5.5 Custom Components

This section looks at how you can create customized components that can be added to dialogs. You might want to do this to have circular rather than rectangular buttons in your dialog, for example. Or perhaps you might want to add a substantial new component type in an application, such as the main grid area in a spreadsheet program.

Creating new components

Every component is a subclass of the Component class. This class contains several methods and variables that can be used by a new component. Custom components will extend this class by implementing its abstract class display(), and very possibly overriding and replacing one or more of Component's methods, such as resize().

A full list of the methods and variables defined in the Component class is given in the GUI class library reference in Appendix C. Please refer to that section when reading this next example, which comes from the gui package itself: a simple "progress bar" component.

Listing 18-6
A ProgressBar Component

```
package gui
$include "guih.icn"

#
# A progress bar
#
class ProgressBar : Component(
    p,                  # The percentage on display.
    bar_x, bar_y,
    bar_h, bar_w)   # Maximum bar height and width.

    method resize()
      #
      # Set a default height based on the font size.
      /h_spec := WAttrib(cwin, "fheight") + 2 * DEFAULT_TEXT_Y_SURROUND
      #
      # Call the parent class's resize method (this is mandatory).
```

```
   self.Component.resize()
   #
   # Set bar height and width figures - this just gives a
   # sensible border between the "bar" and the border of the
   # object. By using these constants, a consistent
   # appearance with other objects is obtained.
   bar_x := x + DEFAULT_TEXT_X_SURROUND
   bar_y := y + BORDER_WIDTH + 3
   bar_w := w - 2 * DEFAULT_TEXT_X_SURROUND
   bar_h := h - 2 * (BORDER_WIDTH + 3)
end

method display(buffer_flag)
   #
   # Erase and re-draw the border and bar
   EraseRectangle(cbwin, x, y, w, h)
   DrawRaisedRectangle(cbwin, x, y, w, h)
   FillRectangle(cbwin, bar_x, bar_y, bar_w * p / 100.0, bar_h)
   #
   # Draw the string in reverse mode
   cw := Clone(cbwin, "drawop=reverse")
   center_string(cw, x + w / 2, y + h / 2, p || "%")
   Uncouple(cw)
   #
   # Copy from buffer to window if flag not set.
   #
   if /buffer_flag then CopyArea(cbwin, cwin, x, y, w, h, x, y)
end

#
# Get the current percentage.
#
method get_percentage()
   return p
end

#
# Set the percentage.
#
method set_percentage(p)
   p <:= 0
   p >:= 100
   self.p := p
   invalidate()
```

```
      end

      #
      # Provide an extra attribute, "percentage"
      #
      method set_one(attr, val)
        case attr of {
          "percentage" : set_percentage(int_val(attr, val))
          default: self.Component.set_one(attr, val)
          }
      end

      initially(a[])
        self.Component.initially()
        set_percentage(0)
        set_fields(a)
end
```

An example of a ProgressBar in use is shown in Figure 18-6.

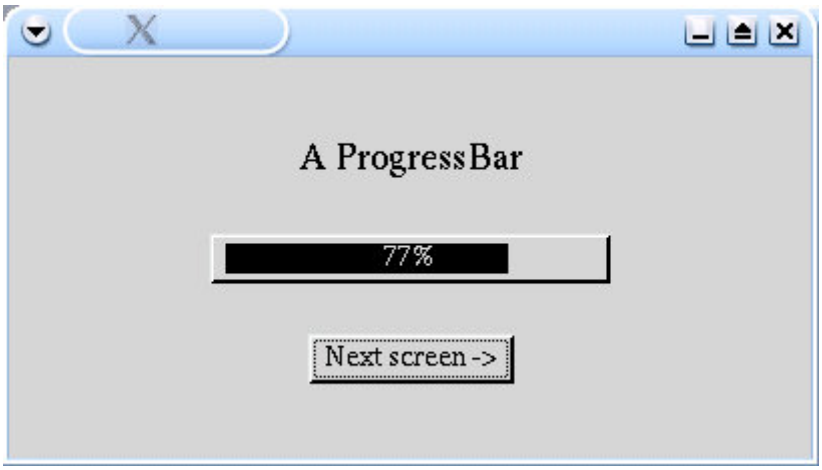


Figure 18-6: Progress bar in use

More complex components use other components within themselves. For example, the TextList class contains two ScrollBars, each of which in turn contains two IconButtons. The Component class has support for contained objects built in, so creating components of this sort is quite easy.

Figure 18-7 shows a dialog window containing another example component which contains an IconButton and a Label as subcomponents. A list of strings is given as an input parameter. When the button is pressed the label changes to the next item in the list, or goes back to the first one. The user can thus select any of the items.

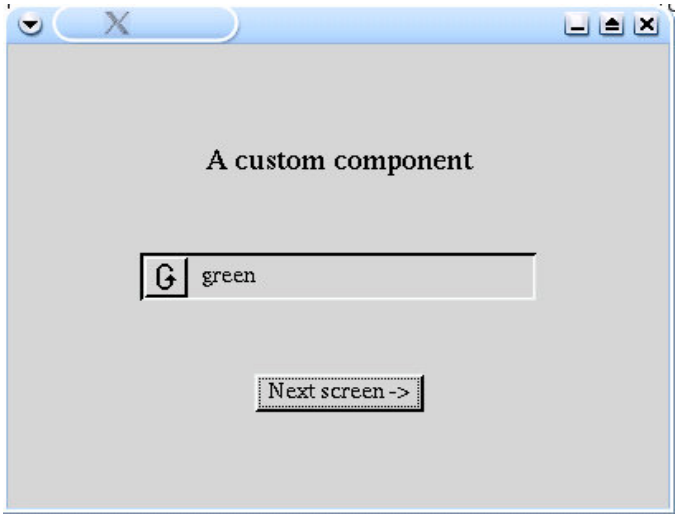


Figure 18-7: Circulate component in use

In this example, sub-components b and l are initialized in the component's constructor, and added to the component using the add method. This ensures they are properly initialized. A listener is connected to the button so that the selection moves on one when it is pressed. The set_selection() method is available to a client program to change the selection programmatically. Note how this method invokes invalidate(). This tells the toolkit that the component needs to be re-displayed. To keep the GUI responsive, the toolkit will only schedule this re-display when there are no user input events to be processed. So, invalidate() doesn't actually invoke the component's display() method directly.

The resize() method sets the sub-components' position and size, and then calls the resize() method for each of them. The display() method erases the component's rectangular area, draws a border, and draws the two sub-components into the area. The set_one() method is also overridden to provide some custom attributes for the component. For all other attributes, the parent class's set_one() method is invoked, meaning all the standard attributes work too. So, a client can construct a Circulate with something like :

```
c := Circulate("size=,250", "pos=20,20",
               "selection_list=hot,warm,cold", "selection=2", "bg=green")
```

Note how the height is omitted; the resize() method will set a default value.

Listing 18-7
Circulate component

```
package gui
link graphics

$include "guih.icn"
```

```
#
# Selection from a list
#
class Circulate : Component(selection, selection_list, b, l)
    #
    # Set the list from which selections are made.
    #
    # @param x the list of selection strings
    method set_selection_list(x)
      selection_list := x
      set_selection(1)
      return x
    end

    #
    # Set the selection to the given index into the selection list.
    #
    # @param x an index into the selection list
    method set_selection(x)
      selection := x
      l.set_label(selection_list[selection])
      invalidate()
      return x
    end

    #
    # Return the current selection, as an index in the selection list.
    #
    # @return an integer, being the current selection
    method get_selection()
      return selection
    end

    #
    # Called once at startup, and whenever the window is resized.
    #
    method resize()
      /h_spec := WAttrib(cwin, "fheight") + 16
      compute_absolutes()

      #
      # Set button position and size
      b.set_pos(BORDER_WIDTH, BORDER_WIDTH)
```

```
   b.set_size(h - 2 * BORDER_WIDTH, h - 2 * BORDER_WIDTH)
   b.resize()

   l.set_pos(h - BORDER_WIDTH + DEFAULT_TEXT_X_SURROUND, h / 2)
   l.set_align("l", "c")
   l.set_size(w - h - 2 * DEFAULT_TEXT_X_SURROUND,
              h - 2 * BORDER_WIDTH)
   l.resize()
   return
end

#
# Display the object.  In this case, double buffering is not necessary.
#
method display(buffer_flag)
   W := if /buffer_flag then cwin else cbwin
   EraseRectangle(W, x, y, w, h)
   DrawSunkenRectangle(W, x, y, w, h)
   l.display(buffer_flag)
   b.display(buffer_flag)
   do_shading(W)
end

#
# The handler for the button - move the selection forward.
#
method on_button_pressed(ev)
   set_selection(1 + selection % *selection_list)
   create_event_and_fire(SELECTION_CHANGED_EVENT, e)
end

method set_one(attr, val)
   case attr of {
      "selection" : set_selection(int_val(attr, val))
      "selection_list" : set_selection_list(val)
      default: self.Component.set_one(attr, val)
   }
end

initially(a[])
   self.Component.initially()
   l := Label()
   l.clear_draw_border()
   add(l)
```

```
        b := IconButton()
        b.connect(self, "on_button_pressed", ACTION_EVENT)
        add(b)
        b.set_draw_border()
        b.set_img("13,c1,~~~~0000~~~~~_
          ~~~000000~~~~~00~~~~00~~~~00~~~~~00~~_
          ~00~~~~~00~~~00~~~~~~~~~00~~~~~~~~_
          ~00~~~~~~0~~~~00~~~~~000~~~00~~~~00000~_
          ~00~~~0000000~00~~~~~~00~~~00~~~~~~00~~_
          ~00~~~~~~00~~~~00~~~~00~~~~~000000~~~~_
          ~~~~0000~~~~~")
        set_fields(a)
end
```

Customized menu components

Listing 18-8 contains a custom menu component. The class hierarchy for menu structures is different to other components, and so this component is a subclass of SubMenu, rather than Component. The component allows the user to select one of a number of colors from a Palette by clicking on the desired box. Again, please read this example in conjunction with the reference section on menus.

Listing 18-8
Color Palette Program

```
import gui

#
# Include the standard constants. Define width of one colour cell in pixels.
#
$include "guih.icn"
$define CELL_WIDTH 30

class Palette : SubMenu(
    w, h,               # width and height
    colour,             # Color number selected
    palette,            # List of colors
    box_size,           # Width/height in cells
    temp_win            # Temporary window
    )

    #
    # Get the result
    #
```

```
method get_colour()
  return palette[colour]
end

#
# Set the palette list
#
method set_palette(l)
  box_size := integer(sqrt(*l))
  return palette := l
end

#
# This is called by the toolkit; it is a convenient place to initialize sizes.
#
method resize()
  w := h := box_size * CELL_WIDTH + 2 * BORDER_WIDTH
end

#
# Called to display the item.  The x, y co-ordinates have been set up
# for us and give the top left hand corner of the display.
#
method display()
  if /temp_win then {
    #
    # Open a temporary area for the menu and copy.
    temp_win := WOpen("canvas=hidden", "size=" || w || "," || h)
    CopyArea(parent_component.get_parent_win(),
          temp_win, x, y, w, h, 0, 0)
   }

  cw := Clone(parent_component.cwin)

  #
  # Clear area and draw rectangle around whole, then draw the color grid
  EraseRectangle(cw, x, y, w, h)
  DrawRaisedRectangle(cw, x, y, w, h)
  y1 := y + BORDER_WIDTH
  e := create "fg=" || !palette
  every 1 to box_size do {
    x1 := x + BORDER_WIDTH
    every 1 to box_size do {
      WAttrib(cw, @e)
```

```
      FillRectangle(cw, x1, y1, CELL_WIDTH, CELL_WIDTH)
      x1 +:= CELL_WIDTH
    }
    y1 +:= CELL_WIDTH
  }
  Uncouple(cw)
end

#
# Test whether pointer in palette_region, and if so which cell it's in
#
method in_palette_region()
  if (x <= &x < x + w) & (y <= &y < y + h) then {
    x1 := (&x - x - BORDER_WIDTH) / CELL_WIDTH
    y1 := (&y - y - BORDER_WIDTH) / CELL_WIDTH
    return 1 + x1 + y1 * box_size
   }
end

#
# Will be called if our menu is open.
#
method handle_event(e)
  if i := in_palette_region() then {
    if integer(e) = (&lrelease | &rrelease | &mrelease) then {
      colour := i
      # This is a helper method in the superclass which
      # closes the menu system and fires an ACTION_EVENT
      succeed(e)
    }
  } else {
    if integer(e) = (&lrelease | &rrelease | &mrelease |
                  &lpress | &rpress | &mpress) then
      # This is a helper method in the superclass which
      # closes the menu system, without firing an event.
      close_all()
  }
end

#
# Close this menu. Restore window area.
#
method hide()
  EraseRectangle(parent_component.cwin, x, y, w, h)
```

```
      CopyArea(temp_win, parent_component.get_parent_win(), 0, 0, w, h, x, y)
      WClose(temp_win)
      temp_win := &null
   end

   # Support a "palette" attrib
   method set_one(attr, val)
      case attr of {
         "palette" : set_palette(val)
         default : self.MenuComponent.set_one(attr, val)
         }
   end

   initially(a[])
      self.SubMenu.initially()
      #
      # Set the image to appear on the Menu above ours. We could design a tiny
      # icon and use that instead of the standard arrow if we wished.
      # Call set_fields to support the attrib style constructor.
      #
      set_img_right(img_style("arrow_right"))
      set_fields(a)
end

#
# Test class dialog.
#
class TestPalette : Dialog(palette)
   method on_palette(ev)
      write("Colour selected : " || palette.get_colour())
   end

   method on_anything(ev)
      write("Anything item selected")
   end

   method component_setup()
      local menu_bar, menu, text_menu_item, close
      attrib("size=400,200")

      #
      # Create a MenuBar which includes our palette as a sub-menu
      menu_bar := MenuBar("pos=0,0")
      menu := Menu("label=Test")
```

```
        text_menu_item := TextMenuItem("label=Anything")
        text_menu_item.connect(self, "on_anything", ACTION_EVENT)
        menu.add(text_menu_item)

        palette := Palette("label=Test menu",
                   "palette=red,green,yellow,black,white,purple,gray,blue,pink")
        palette.connect(self, "on_palette", ACTION_EVENT)
        menu.add(palette)
        menu_bar.add(menu)
        add(menu_bar)

        #
        # Add a close button.
        close := TextButton("pos=50%,66%", "align=c,c", "label=Close")
        close.connect(self, "dispose", ACTION_EVENT)
        add(close)
    end
end

procedure main()
    TestPalette().show_modal()
end
```

The resulting window, with the Palette menu active is shown in Figure 18-8.

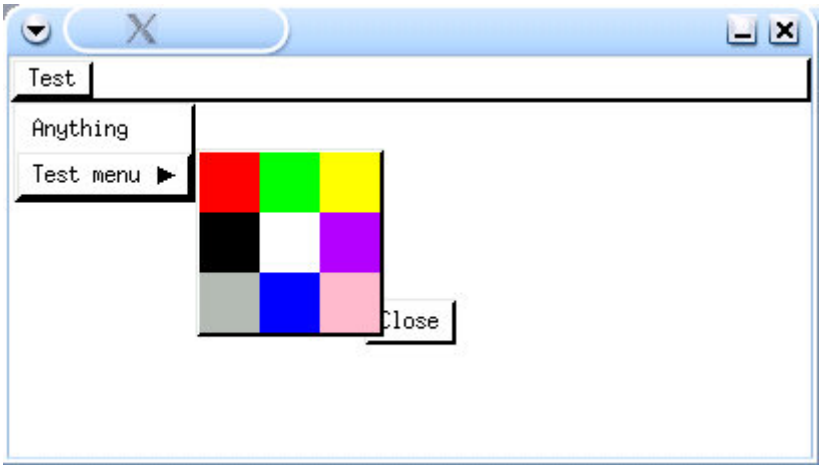


Figure 18-8: TestPalette window

18.5.6 Tickers

GUI programs spend most of their time waiting for user input events. The toolkit allows this spare time to be exploited by user programs by the use of "tickers". These are classes

which subclass Ticker, and implement a single method, tick(), which is invoked at a specified interval. If the toolkit is busy processing events or updating the display, the actual interval may be much greater than that requested, but it will never be less.

Many components make use of tickers. For example when the mouse button is dragged below the bottom of an EditableTextList and held, the cursor scrolls downwards without any user events occurring. This is handled with a simple ticker:

```
class FirstTicker : Ticker()
    method tick()
      write("Doing something in FirstTicker")
    end
end
```

If we had an instance t of this ticker, we would start it with:

```
t.set_ticker(1000)
```

and stop it with

```
t.stop_ticker()
```

Both calls would have to be made whilst a dialog was open, because it is from within the toolkit's event processing loop that tickers are scheduled.

For convenience, the base Component class is a subclass of Ticker. This means that a dialog class (which is itself a subclass of Component, and hence of Ticker too), can simply implement a tick() method to use the ticker facility. It simply needs to invoke set_ticker() and stop_ticker() to start and stop the ticker, respectively. Another method, retime_ticker(), allows the rate of an active ticker to be changed.

One important rule regarding ticker programming has to be borne in mind, and that is that the tick() method implementation must return quite quickly; at most within a few tenths of a second. If it does not, then the GUI may become unresponsive to user events, which cannot be processed whilst control is in the tick() method.

The tick() method must return quickly. If the task you want to implement in the background is by nature deeply structured and takes a long time, it may be difficult to get out of the tick() method promptly and continue in the same state on the next tick. Co-expressions can be helpful here. A co-expression maintains its own stack and can suspend itself, and then continue again later with the state (including stack) intact.

Here is an example program which illustrates these points. It generates prime numbers using the Erastothenes' sieve method, in a ticker, using a co-expression to conveniently suspend generation after each prime. This program also introduces a new component, a slider which is used to increase or decrease the ticker rate dynamically.

Listing 18-9

Erastothenes' Sieve Program

```
import gui
$include "guih.icn"
$define PRIME_LIMIT 20000

#
# A program to calculate prime numbers in a background ticker,
# and display them in a dialog.
#
class Sieve : Dialog(prime_ce, interval, count_label, prime_label,
                     rate_label, start, stop)

    #
    # Bring the label and ticker into line with the interval slider.
    # If the ticker is running, retime it.
    #
    method synch_interval()
      rate_label.set_label(interval.get_value() || " ms")
      if is_ticking() then retime_ticker(interval.get_value())
    end

    #
    # Toggle the grey state of the start/stop buttons.
    #
    method toggle_buttons()
      start.toggle_is_shaded()
      stop.toggle_is_shaded()
    end

    #
    # When the start button is pressed, toggle the grey state and start the ticker.
    #
    method on_start()
      toggle_buttons()
      set_ticker(interval.get_value())
    end

    #
    # When the stop button is pressed, toggle the grey state and stop the ticker.
    #
    method on_stop()
      toggle_buttons()
      stop_ticker()
    end
```

```
#
# The tick method, which is invoked regularly by the toolkit.
# It just invokes the co-expression to display the next prime.
#
method tick()
  @prime_ce
end

#
# This method consitutes the co-expression body.
#
method primes()
  local prime_candidate, non_prime_set := set(), prime_count := 0

  every prime_candidate := 2 to  PRIME_LIMIT do {
    if not member(non_prime_set, prime_candidate) then {
      #
      # Update the UI.
      count_label.set_label(prime_count +:= 1)
      prime_label.set_label(prime_candidate)
      #
      # Update the non-prime set.
      every insert(non_prime_set,
        2 * prime_candidate to PRIME_LIMIT by prime_candidate)
      #
      # Suspend the co-expression until the next tick.
      @&source
    }
  }
end

method component_setup()
  local prime_border, rate, buttons, b

  attrib("size=325,200", "label=Sieve")
  connect(self, "dispose", CLOSE_BUTTON_EVENT)

  prime_border := Border("pos=20,20", "size=100%-40,78")
  prime_border.set_title(Label("pos=10,0", "label=Primes"))
  prime_ce := create primes()

  prime_border.add(Label("pos=20,18", "label=Prime:"))
  count_label := Label("pos=77,18", "size=40")
  count_label.set_label("")
```

```
        prime_border.add(count_label)
        prime_border.add(Label("pos=20,40", "label=Value:"))
        prime_label := Label("pos=77,40", "size=40")
        prime_label.set_label("")
        prime_border.add(prime_label)

        add(prime_border)

        rate := Panel("pos=20,112", "size=100%-40,30")
        rate.add(Label("pos=0,50%", "size=45", "align=l,c", "label=Rate:"))
        interval := Slider("pos=45,50%", "size=100%-90", "align=l,c",
                    "range=20,2020", "is_horizontal=t")
        interval.set_value(1000)
        interval.connect(self, "synch_interval", SLIDER_DRAGGED_EVENT)
        rate.add(interval)

        rate_label := Label("pos=100%,50%", "size=45", "align=r,c",
                "internal_alignment=r")
        rate.add(rate_label)
        synch_interval()
        add(rate)

        buttons := Panel("pos=50%,158", "size=161,25", "align=c,t")
        start := TextButton("pos=0,0", "label=Start")
        start.connect(self, "on_start", ACTION_EVENT)
        buttons.add(start)
        stop := TextButton("pos=58,0", "label=Stop", "is_shaded=t")
        stop.connect(self, "on_stop", ACTION_EVENT)
        buttons.add(stop)
        b := TextButton("pos=108,0", "label=Quit")
        b.connect(self, "dispose", ACTION_EVENT)
        buttons.add(b)
        add(buttons)
    end
end

procedure main()
    Sieve().show_modal()
end
```

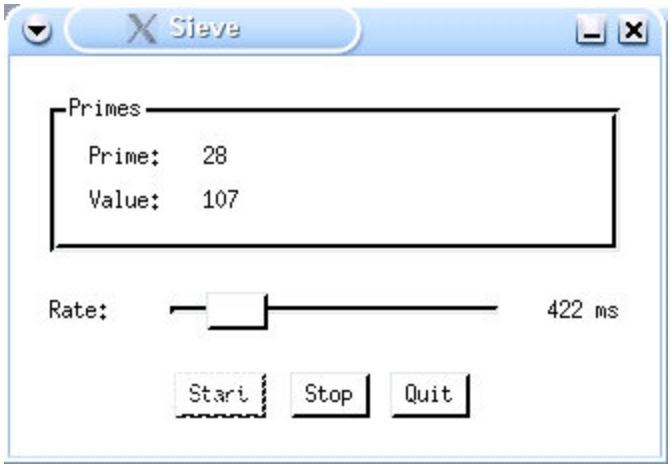


Figure 18-9: Sieve

18.6 Advanced List Handling

Several of the more sophisticated components extend a common base class, SelectableScrollArea, namely TextList, Table and Tree. (In fact, Table doesn't directly extend SelectableScrollArea, it contains a header component and a content component that does). It is quite easy to add some advanced features to these components, such as right-click popup menus, multi-selection and drag and drop, and this is explained in this section.

18.6.1 Selection

Selection handling is straightforward. First, configure the component so that it allows selection of no, one, or many rows using the methods set_select_none(), set_select_one() or set_select_many(), or the attributes "select_none", "select_one" or "select_many". Then, listen for changes by listening for a SELECTION_CHANGED_EVENT to be fired :

```
comp.connect(self, "handle_selection", SELECTION_CHANGED_EVENT)
```

When such an event does occur, the current selections can be retrieved in one of two ways. Either by getting the indexes of the selections using get_selections(), or by getting the objects selected, using object_get_selections(). The former returns a list of integers, the latter a list of objects whose type depends on the component. For a TextList, a list of strings is returned, for a Table, a list of lists (each being a row's data), and for a Tree, a list of Node objects is returned. There are corresponding setter methods for setting the selection dynamically.

18.6.2 Popups

Adding popup menus is also easy. First create a Popup component, ready to be shown. Then, listen for a MOUSE_RELEASE_EVENT. Finally, when an event occurs check that is a right mouse release, and that the object is in the state you want. If it is, just activate the popup via its popup() method.

18.6.3 Drag and drop

The toolkit supports a limited form of drag and drop, which works only between components within the same window. To implement drag and drop, a class, DndHandler, must be subclassed and an instance "plugged-in" to the component which is a source or target of a potential drag and drop operation, using its set_dnd_handler() method. The DndHandler class provides five callback methods which the toolkit uses to control a drag and drop operation. When using the SelectableScrollArea family of components, it is best to subclass SelectableScrollAreaDndHandler, which is a custom subclass of DndHandler, with several methods already defined appropriately.

All of the above features are brought together in the following example program which provides a Tree and a TextList. Drag and drop is enabled between the two components, and both provide popup menus for adding/deleting elements.

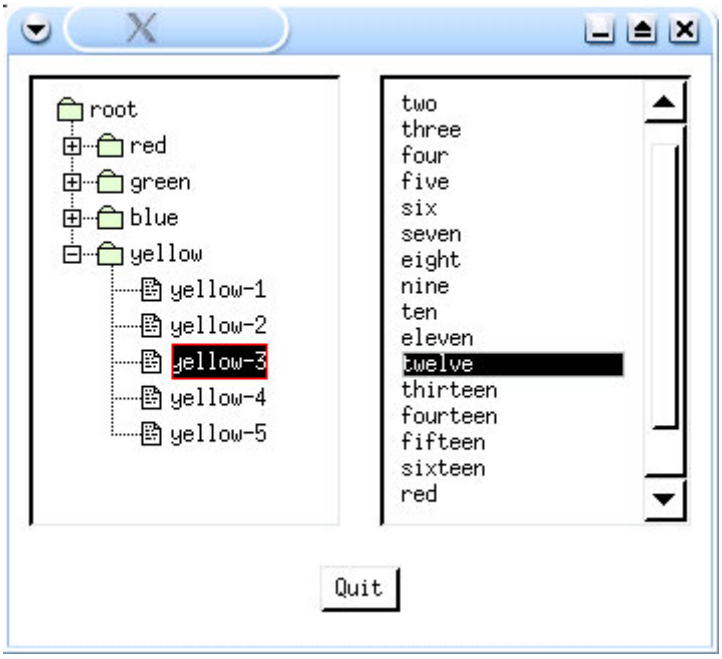


Figure 18-10: Drag and drop

Listing 18-10 The Drag and drop Program

```
import gui
$include "guih.icn"

#
# A DndHandler for the list
#
class ListDndHandler : SelectableScrollAreaDndHandler()
    #
    # A drop has occurred; we succeed iff we accept it
    #
    method can_drop(d)
      local l, ll
      if l := parent.get_highlight() then {
        if d.get_source() === parent then # Move within the list itself
          parent.move_rows(parent.get_gesture_selections(), l)
        else {
          #
          # Copy from tree to list. d.get_content() gives
          # a list of the nodes being dragged.
          ll := [ ]
          every el := !d.get_content() do # Don't drag folders.
            if /el.is_folder_flag then put(ll, el.get_label())
          parent.insert_rows(ll, l)
        }
        return
      }
    end

    #
    # This is invoked after a successful operation when the
    # list was the source.  If the destination (c) wasn't the
    # list, then we must delete the rows from the list.
    #
    method end_drag(d, c)
      if c ~=== parent then
        parent.delete_rows(parent.get_gesture_selections())
    end
end

#
# A DndHandler for the tree
#
class TreeDndHandler : SelectableScrollAreaDndHandler()
    #
```

```
# Called during a drag event. We succeed iff the user is dragging over
# a row (this is handled by the parent) AND the thing we're over is a folder.
#
method drag_event(d)
  if SelectableScrollAreaDndHandler.drag_event(d) then
    return \(parent.object_get_highlight().is_folder_flag)
end

#
# A drop has occurred; we succeed iff we accept it.
# Only consider a drop on a folder.
#
method can_drop(d)
  local s, other, n, el
  if other := parent.object_get_highlight() & \other.is_folder_flag then {
    if d.get_source() === parent then {
      #
      # If parent is the drop source, then we have a dnd from within
      # the tree. So, we just move the nodes.
      # d.get_content() will be a list of the nodes that were dragged.
      every el := !d.get_content() do {
        if el.get_parent_node().delete_node(el) then
          other.add(el)
      }
    } else {
      #
      # Drop from list.  In this case d.get_content() will be a list of strings.
      every el := !d.get_content() do {
        n := TreeNode()
        n.set_label(el)
        other.add(n)
      }
    }
    #
    # Notify the tree that the node data structure has altered.
    parent.tree_structure_changed()
    return
  }
end

#
# This is invoked after a successful operation when the tree was the source.
# If the destination (c) wasn't the tree, we must delete the nodes from the tree.
#
```

```
    method end_drag(d, c)
      if c ~=== parent then {
        #
        # Delete all the nodes which will have been dragged.
        every n := !parent.object_get_gesture_selections() do
          if /n.is_folder_flag then
            n.get_parent_node().delete_node(n)
        #
        # Notify the tree that the node data structure has altered.
        parent.tree_structure_changed()
      }
    end
end

#
# We use a custom Node subclass to also store an "is_folder_flag" flag.
#
class TreeNode : Node(is_folder_flag)
initially
    self.Node.initially()
    if \is_folder_flag then
      set_bmps([img_style("closed_folder"),
            img_style("closed_folder"), img_style("closed_folder")])
end

#
# The main dialog.
#
class DNDTest : Dialog(tree, lst, tree_popup, list_popup, new_folder_menu_item,
                       delete_node_menu_item, delete_rows_menu_item)

    #
    # Delete nodes handler
    #
    method on_delete_node()
      local n, i, l

      every n := !(tree.object_get_gesture_selections()) do
        n.get_parent_node().delete_node(n)

      tree.tree_structure_changed() # Notify tree of the change.
    end

    #
```

```
# Create a new folder
#
method on_new_folder()
  local n, o

  #
  # Simply add a new node under the cursor, and notify the
  # tree that the data structure changed.
  #
  if o := tree.object_get_cursor() then {
    n := TreeNode(1)
    n.set_label("New folder")
    o.add(n)
    tree.tree_structure_changed()
    }
end

#
# Delete rows from the list
#
method on_delete_rows()
  lst.delete_rows(lst.get_gesture_selections())
end

#
# Add some rows to the list, at the cursor position, or at
# the top if there is no cursor.
#
method on_new_rows()
  lst.insert_rows(["new1", "new2", "new3"], lst.get_cursor() | 1)
end

#
# Helper method to create a tree structure.
#
method create_tree()
  local r := TreeNode(1), n
  r.set_label("root")

  every s := ("red" | "green" | "blue" | "yellow") do {
    n := TreeNode(1)
    n.set_label(s)
    r.add(n)
    every t := 1 to 5 do {
```

```
      o := TreeNode()
      o.set_label(s || "-" ||t)
      n.add(o)
      }
    }
  return r
end

#
# A selection-up event on the tree
#
method on_tree_release(ev)
  local n
  #
  # If the Icon event was a right mouse release, display the popup at the cursor.
  if ev.get_param() === &rrelease then {
    n := tree.object_get_cursor() | fail
    #
    # Adjust the shading depending on the node type.
    if /n.is_folder_flag then
      new_folder_menu_item.set_is_shaded()
    else
      new_folder_menu_item.clear_is_shaded()
    if n === tree.get_root_node() then
      delete_node_menu_item.set_is_shaded()
    else
      delete_node_menu_item.clear_is_shaded()
    tree_popup.popup()
  }
end

#
# A mouse release event on the list
#
method on_list_release(ev)
  if ev.get_param() === &rrelease then {
    #
    # If some rows to delete...
    if lst.get_gesture_selections() then
      delete_rows_menu_item.clear_is_shaded()
    else
      delete_rows_menu_item.set_is_shaded()

    list_popup.popup()
```

```
  }
end

method component_setup()
  local m, quit, mi
  attrib("size=350,295", "resize=on")
  connect(self, "dispose", CLOSE_BUTTON_EVENT)
  tree := Tree("pos=50%-10,10", "size=50%-20,100%-70",
          "align=r,t", "select_many", "show_root_handles=f")
  tree.set_root_node(create_tree())
  tree.set_dnd_handler(TreeDndHandler(tree))
  tree.connect(self, "on_tree_release", MOUSE_RELEASE_EVENT)
  add(tree)
  quit := TextButton("pos=50%,100%-40", "align=c,t", "label=Quit")
  quit.connect(self, "dispose", ACTION_EVENT)
  add(quit)
  # Create a TextList, with some arbitrary content.
  lst := TextList("pos=50%+10,10", "size=50%-20,100%-70", "select_many",
              "contents=one,two,three,four,five,six,seven,eight,nine,ten,eleven,_
               twelve,thirteen,fourteen,fifteen,sixteen,red,blue,green")
  lst.connect(self, "on_list_release", MOUSE_RELEASE_EVENT)
  lst.set_dnd_handler(ListDndHandler(lst))
  add(lst)
  tree_popup := PopupMenu()
  m := Menu()
  tree_popup.set_menu(m)
  delete_node_menu_item := TextMenuItem("label=Delete")
  delete_node_menu_item.connect(self, "on_delete_node", ACTION_EVENT)
  m.add(delete_node_menu_item)
  new_folder_menu_item := TextMenuItem("label=New folder")
  new_folder_menu_item.connect(self, "on_new_folder", ACTION_EVENT)
  m.add(new_folder_menu_item)
  add(tree_popup)
  list_popup := PopupMenu()
  m := Menu()
  list_popup.set_menu(m)
  delete_rows_menu_item := TextMenuItem("label=Delete")
  delete_rows_menu_item.connect(self, "on_delete_rows", ACTION_EVENT)
  m.add(delete_rows_menu_item)
  mi := TextMenuItem("label=Insert rows")
  mi.connect(self, "on_new_rows", ACTION_EVENT)
  m.add(mi)
  add(list_popup)
end
```

```
end

procedure main()
    DNDTest().show_modal()
end
```

18.7 Programming Techniques

Some of the earlier example dialogs were effectively "application windows." In other words, the top-level window of a program. This section looks at some techniques for integrating dialog windows that are secondary or helper windows into a program.

Parameters

A dialog window will normally have parameters that the calling program will want to pass to it before it is displayed using the show() method. Possibly the attribute syntax "key=val" should be supported, and perhaps a default value should be set. All of these things are easily supported by following the following structure:

```
class AnyDialog : Dialog(a_variable)
    method set_a_variable(x)
      a_variable := x
    end
    ...
    method set_one(attr, val)
      case attr of {
        "a_variable" :
          set_a_variable(string_val(attr, val))
        default: self.Dialog.set_one(attr, val)
      }
    end

    method component_setup()
      # Initialize Components, possibly depending upon the value of a_variable
      ...
      # Configure the window itself...
      attrib("size=300,200")
    end

    initially(a[])
      self.Dialog.initially()
      a_variable := "default value"
      set_fields(a)
```

```
end
```

You then use the following code in the calling program:

```
d := AnyDialog()
d.set_a_variable("something")
```

or

```
d := AnyDialog("a_variable=something")
```

or just

```
d := AnyDialog()
```

to use the default for a_variable. Furthermore, the standard dialog attributes can still be used as you would expect :

```
d := AnyDialog("a_variable=something", "font=times", "bg=green", "fg=red")
```

Subclassing can follow the same pattern. For example:

```
class AnotherDialog : AnyDialog(another_variable)
    method set_another_variable(x)
      another_variable := x
    end
    ...
    method set_one(attr, val)
      case attr of {
        "another_variable" :
          set_another_variable(string_val(attr, val))
        default: self.AnyDialog.set_one(attr, val)
      }
    end

    method component_setup()
      self.AnyDialog.component_setup()
      ...
    end

    initially(a[])
      self.AnyDialog.initially()
      another_variable := "default value"
      set_fields(a)
end
```

At first sight, it might seem that set_fields() will be invoked twice, which may cause problems. In fact, because we are calling the AnyDialog constructor with no parameters, the set_fields() call in that constructor has no effect. We just have to remember to call set_fields(a) in the AnotherDialog constructor itself. This will delegate its work up to the parent classes' set_one() methods to handle all of the possible attributes we may give it.

Getting results out to the calling program is easy. The dialog can just set a result variable that can be retrieved by the caller using one of the dialog's methods.

18.8 ivib

It can take many compiles and runs to get the components correctly sized and positioned in a dialog window with many components. Much of the code in a dialog is tedious to write. An interface builder called Ivib reduces the effort required for many common dialogs. Ivib allows a user to draw (interactively place and configure) components in a window area. Ivib's saved files are program source code that implement the interface. Ivib was inspired by VIB, a program written by Mary Cameron and greatly extended by Gregg Townsend. The main window of Ivib, with a dialog under construction, is shown in Figure 18-11.

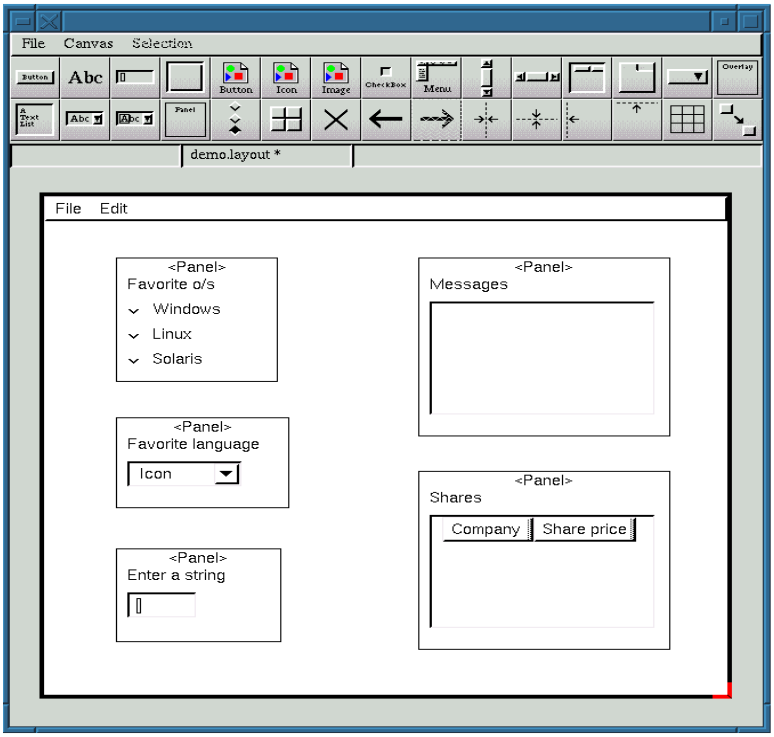


Figure 18-11: Ivib main window

To create a dialog window using Ivib, start the program with the name of a new source file. For example:

```
ivib myprog.icn
```

The Ivib window will appear with a blank "canvas" area, which represents the dialog window to be created. At startup, the attributes of this window are the default Icon window attributes. Before you learn how to change these attributes, here is how you add a button to the dialog. Clicking the button in the top left-hand corner of the toolbar does this. Try moving and resizing the resulting button by left-clicking on it with the mouse. To change the label in the button, click on it so that the red borders appear in the edges. Then press Alt-D. The dialog shown in Figure 18-12 (left) appears. Change the label by simply changing the text in the "Label" field and clicking "Okay".

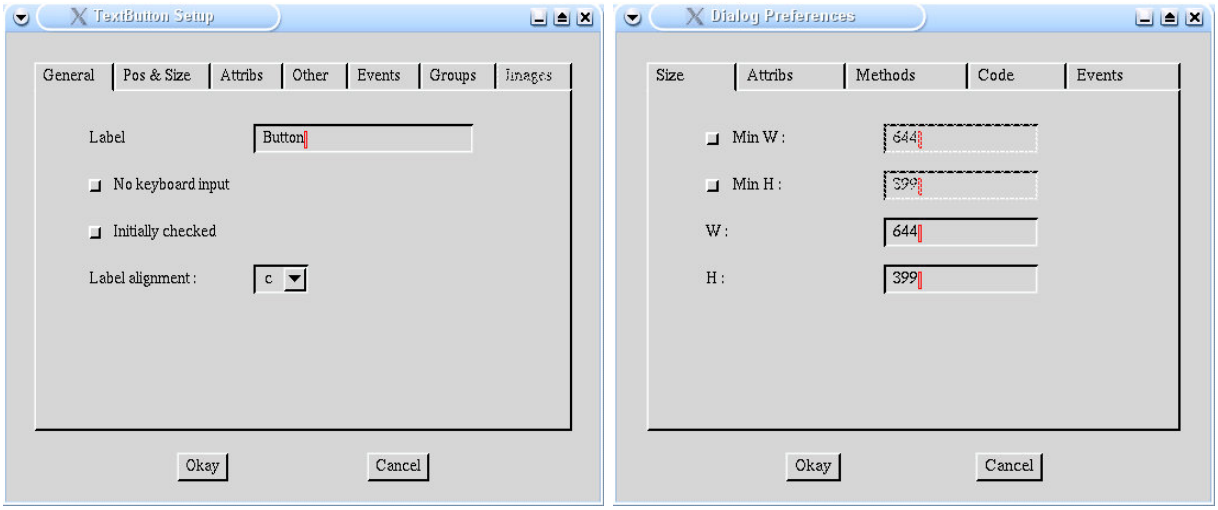


Figure 18-12 Button (left) and Dialog (right) configuration windows

As just mentioned, the dialog's attributes are initially the default window attributes. To change these, select the menu option Canvas -> Dialog prefs. The window shown in Figure 18-12 (right) appears. To change the dialog attributes:

1. Click on the Attribs tab
2. Click on the Add button
3. Edit the two text fields so that they hold the attribute name and the attribute value respectively; for example try adding "bg" and "pale blue".
4. Click on Apply.
5. Click on Okay.

Note that the button changes its background to pale blue too. Each object has its own attributes that it can set to override the dialog attributes. Click on the button and press Alt-D to bring up the button's configuration dialog again. Now click on the Attribs tab of

this dialog and set the background color to white, for example. Then click okay and you will see that the button's background changes to white.

You will recall from the previous example programs that some objects can be contained in other objects, such as the Panel class. This is handled conveniently in Ivib. Add a Panel object to the dialog by clicking on the Panel button (on the second row, fourth from the left). A panel appears. Now drag the button into the panel. A message should appear in the information label below the toolbar, "Placed inside container." Now try dragging the panel about, and you will observe that the button moves too - it is now "inside" the panel. Dragging it outside the panel's area moves it back out. This method applies to all the container objects.

There are several buttons that operate on objects. The large "X" button deletes the currently selected objects. Try selecting the button and deleting it. The arrow buttons are "redo" and "undo" operations. Clicking on the undo button will undo the delete operation and the button should reappear.

Now try saving your canvas. Press Alt-S, and select a filename, or accept the default. At the end of an ivib-enhanced Unicon source file is a gigantic comment containing ivib's layout information. This comment is ASCII text, but it is not really human-readable. If the program is called, for example, myprog.icn, then this can be compiled with

```
unicon myprog
```

to give an executable file myprog that, when run, will produce the same dialog shown in the canvas area. Of course, the resulting dialog will not do anything, and it is then up to the programmer to fill in the blanks by editing myprog.icn.

Hopefully, following the above steps will give you an idea of how the Ivib program works. Below are more details regarding the individual components, dialogs, and operations.

Moving, selecting and resizing

Select an object by clicking on it with the mouse. Its selection is indicated by red edges around the corners. Multi-select objects by holding the shift key down and clicking on the objects. The first selected object will have red corners; the others will have black corners. There are several functions which operate on multiple objects and map some attribute of the first selected object to the others; hence the distinction.

To move an object, select it by clicking on it with the left mouse button, and then drag it. Note that dragging an object even by one pixel will set the X or Y position to an absolute figure, disturbing any carefully set up percentage specification! Because this can be irritating when done accidentally, the X and/or Y position may be fixed in the dialog so that it cannot be moved in that plane. Alternatively, when selecting an object that you do not intend to move, use the right mouse button instead.

To resize an object, select it and then click on one of the corners. The mouse cursor will change to a resize cursor and the corner may be dragged. Note that, like the position

specification, resizing an object will set the size to an absolute number and will also reset the "use default" option. Again, this can be avoided by fixing the width and/or height in the object's dialog box.

Dialog configuration

This dialog, accessed via the menu selection Canvas -> Dialog prefs allows the user to configure some general attributes of the dialog being created. The tabs are described in this section.

Size The minimum width and height entries simply set the minimum dimensions of the window. The width and height may be configured here, or more simply by resizing the canvas area by clicking on the red bottom right-hand corner.

Attribs This has been described briefly above; the Add button produces a new entry that is then edited. The edited entry is placed in the table with Apply. The Delete button deletes the currently highlighted selection.

Code generation The part of the code to setup the dialog is written into a method called setup. If the "interpose in existing file" option is checked, then the program will read the present contents of the selected output file up to the current setup method, interpose the new setup, and copy the remainder out. This is useful if some changes have been made to the file output by a previous run. It is important to take a copy of the existing file before using this option, in case unexpected results occur.

The other options simply select which other methods should be produced. If a main procedure is produced, then the result will be an executable program.

Other This tab allows the name of the dialog to be set, together with a flag indicating whether it is "modal" or not. If so, then a method called pending is produced. This method is repeatedly called by the toolkit while it is waiting for events to occur.

Component configuration

Each component dialog has a standard tabbed pane area for configuration of attributes common to all components. The tabs are as follows:

Position & size The X, Y, W and H options set the position and size of the object. The drop-down list can be used for convenience, or a value may be entered by hand. The "fix" buttons prevent the object from being moved or sized outside the given parameter in the Canvas area. This is useful once an object's position has been finalized, and you don't wish

to accidentally move it. The "use default" buttons mean that the width/height will be set as the default for the object based on the parameters and the attributes. For example, a button's size will be based on the label and the font. For some objects there is no default size, so these buttons are shaded. The alignment of the object is also set from this tab.

Attribs This works in exactly the same way as the Attribs tab for the dialog, except that these attributes apply only to the object.

Other This tab allows the name used in the output program code to be set. The "Draw Border" button applies to some objects (see the reference section in Appendix C for further information). If the "Is Shaded" button is clicked, then the initial state of the object will be shaded. If the "Has initial focus" button is clicked, then this object will have the initial keyboard focus when the dialog is opened.

Component details

Components are added to the dialog by clicking on the toolbar buttons, and dialogs are produced by selecting the object and pressing Alt-D, as explained above. Most of the dialogs are hopefully straightforward, but some warrant further explanation.

TextButton The dialog for this component includes an option for the button to be added to a ButtonGroup structure. This is explained in detail shortly.

Border To select this component, rather than clicking inside the border, click in the area at the bottom right hand corner. It can then be resized or moved. Now try dragging another object, such as a CheckBox or Label and release it so that its top left-hand corner is within the area in the bottom right-hand corner of the Border object. The CheckBox/Label or whatever is now the title of the Border. Thus, any object can be in the title. To remove the object from the Border, just drag it out. The alignment of the title object is set in the dialog, but is by default left aligned.

Image Initially the Image object has an outline. When a filename is entered into the dialog however, the image itself is displayed.

CheckBox Customized up/down images may be set from the dialog, and a CheckBoxGroup may be selected if one is available; this is explained in more detail shortly.

MenuBar This dialog enables a complete multi-level menu to be created. Clicking on a line allows an item to be inserted at that point. Only menus can be inserted into the lowest level. Clicking on an item will allow insertion, deletion, or editing of the particular

item. A CheckBoxGroup can be created by selecting multiple check boxes (by holding down the Shift key while clicking on the lines) and clicking the button.

ScrollBar Both vertical and horizontal scroll bars are available; for details of how the various options work, please see the reference manual for the toolkit.

Table A table column is added by clicking the Add button; the details should then be edited and the Apply button pressed to transfer the details to the table. A selected column can be deleted with the Delete button. The drop-down list selects whether or not lines in the table can be selected by clicking them.

TabSet Add a TabItem (a tabbed pane) by clicking the Add button. A single pane is automatically present when the object is created. To switch between panes, select a TabItem button. An asterisk appears by the entry. When the dialog exits, this TabItem is on the front of the TabSet. To add items to the current pane, drag and drop them into it. The whole of the item must be in the pane, and a confirmation message appears to indicate that the item has been added to the container. To take it out of the container, drag it out of the pane. Note that the selected pane is the one that is configured to be initially at the front when the dialog is opened.

MenuButton The MenuButton component is a menu system with one root menu. The dialog is the same as MenuBar's except that the small icon can be configured.

OverlaySet The configuration for an OverlaySet is very similar to that for a TabSet, except that there are no tab labels to configure of course.

CheckBoxGroup This does not create an object, it places several selected CheckBox objects into a CheckBoxGroup object. They act as coordinated radio buttons. To use this button, select several CheckBox objects, and press the button.

The CheckBoxGroup is configured by selecting the menu item Canvas -> CheckBoxes. The only attribute to be configured is the name of the CheckBoxGroup. Once a CheckBoxGroup has been created, it cannot be deleted. A CheckBox can be taken out or put into a CheckBoxGroup from its configuration dialog.

ButtonGroup The ButtonGroup operates in a very similar fashion to CheckBoxGroup, except that it places buttons into a ButtonGroup.

Other editing functions

Other editing functions can be applied to the dialog being created. They are accessed either via the toolbar, the Selection menu, or the Edit menu.

Delete The Delete function simply deletes all the selected objects; note that deleting a container also deletes all the objects inside it.

Undo and Redo The Undo and Redo functions undo and redo changes. The size of the buffer used for storing undo information can be configured in the File -> Preferences dialog; by default it allows 7 steps backward at any one time.

Center Horizontally The Center Horizontally operation sets the selected objects' X position specification to "50%", their alignment to center, and fixes them in that horizontal position. To "unfix" an object, uncheck the "fix" box in its dialog box. Center vertically naturally works in just the same way for the y position.

Align Horizontally This operation sets the X position and alignment of all the selected objects to the X position and alignment of the first selected object. Whether the objects end up appearing to be left-, center-, or right-aligned will depend on the alignment of the first selected object. "Align vertically" works just the same way.

Grid To use the Grid function, place several items roughly in a grid, select them all, perform the operation, and hopefully they will be nicely aligned.

Copy The Copy function simply creates a duplicate of each selected object.

Equalize widths This function simply copies the width of the first selected object to all of the other selections. "Equalize heights" naturally does the same for heights.

Even Space Horizontally This operation moves the objects so that they are equally spaced between the leftmost and rightmost objects of the current selections, which are not moved. "Even Space Vertically" does the same vertically.

Reorder The Reorder function is used to reorder the selected objects in the program's internal lists. This is useful so that they are produced in the program output in the desired order, for example to ensure that the tab key moves from object to object in the right sequence. By selecting several objects and using reorder, those objects appear first in sequence in the order in which they were selected.

18.9 Summary

The Unicon GUI toolkit offers a full-featured, attractive way of constructing interfaces. The toolkit has many features–tables, tabbed property sheets, and multi-line text editing capability–that are not present in Icon's vidgets GUI library. Many components present in both libraries are more flexible in the GUI toolkit, supporting fonts, colors, and graphics that the vidgets library does not handle. The ivib interface builder tool provides programmers with easy access to the GUI toolkit.

Object-orientation mainly affects this class library's extensibility, although it also contributes to the simplicity of the design. Inheritance, including multiple inheritance, is used extensively in the 37 classes of the GUI toolkit. Inheritance is the main object-oriented feature that could not be easily mimicked in a procedural toolkit such as the vidgets library, and inheritance is the primary extension mechanism.

Part IV

Appendices

Appendix A

Language Reference

Unicon is expression-based. Nearly everything is an expression, including the common control structures such as while loops. The only things that are not expressions are declarations for procedures, methods, variables, records, classes, and linked libraries.

In the reference, types are listed for parameters and results. If an identifier is used, any type is allowed. For results, generator expressions are further annotated with an asterisk (*) and non-generators that can fail are annotated with a question mark (?). A question mark by itself (short for null?) denotes a predicate whose success or failure is what matters; the predicate return value (&null) is not significant.

A "Road Narrows" sign in either margin — like the sign reproduced here — indicates that the function or operation is not thread-safe and should be protected from different threads executing it at the same time (the sign is intended to suggest that only one thing should be allowed through at any one time). In some cases, notably the augmented operations (+:= etc.) and the 3D operations, the entire group is not thread-safe. In these cases the signs that would be beside the individual functions or operations are replaced by a single cautionary sign at the head of the group. In a few instances, a small sign showing

parallel arrows is used to highlight a general comment about concurrency (rather than a specific thread-safety issue).

A.1 Immutable Types: Numbers, Strings, Csets, Patterns

Unicon's immutable types are integers, real numbers, strings, and csets. Values of these types cannot change. Operators and functions on immutable types produce new values rather than modify existing ones. The simplest expressions are literal values, which occur only for immutable types. A literal value evaluates to itself.

Integer

Integers may have an arbitrary magnitude. Decimal integer literals are contiguous sequences of the digits 0 through 9, optionally preceded by a + or - sign. Suffixes K, M, G, T, or P multiply a literal by 1024, 1024^2, 1024^3, 1024^4, and 1024^5, respectively.

Radix integer literals use the format *radix*R*digits*, where *radix* is a base in the range 2 through 36, and *digits* consists of one or more numerals in the supplied radix. After values 0-9, the letters A-Z are used for values 10-35. Radix literals are case insensitive, unlike the rest of the language, so the R may be upper or lower case, as may the following alphabetic digits.

Large Integers

Unicon automatically converts an integer value into a "large integer" value when its magnitude exceeds the limits of the underlying hardware's native integer representation. Large integers are interchangeable with native integers in most circumstances, but there are a few places where only a native integer (here denoted by i, j and k) is acceptable.

- i to j by k and, by implication, !i
- seq(i,j)
- Assignment to integer valued keywords
- *expr* \ i
- exit(i)

A number of other standard functions will cause a runtime error — or fail, depending on the value of &error — if given a large integer as a parameter (for example, delay or the second parameter of get), but it is difficult to envisage a situation where a large integer value makes sense in most of these contexts. Perhaps, as the capabilities of computers increase, this view will come to be seen as a failure of imagination.

The penalty paid for supporting large integers if they are not used is very small so, although the support for large integers can be removed via a build option, almost all implementations provide it (and it is enabled by default).

Note that large integer literals in a program are converted to actual large integers when evaluated during program execution. Consequently, such literals should not be placed in loops or other places in which they are evaluated frequently. Arithmetic operations on native integers are considerably faster than using large integers so, if they can avoided without major effort, it is probably worth doing so.

Real

Reals are double-precision floating-point values. Real decimal literals are contiguous sequences of the digits 0 through 9, with a decimal point (a period) somewhere within or at either end of the digits. Real exponent literals use the format *number*E*integer*; E may be upper or lower case. Note that the integer must be a decimal integer (including the optional + or - sign); Radix integer literals are not supported in an exponent.

String

Strings are sequences of 0 or more characters, where a character is a value with a platform-dependent size and symbolic representation. On platforms with multi-byte character sets, multiple Icon characters represent a single symbol using a platform-dependent encoding. String literals consist of 0 or more characters enclosed in double quotes. A string literal may include escape sequences that use multiple characters to encode special characters. The escape sequences are given in Table A-1. Incomplete string literals may be continued on the next line if the last character on a line is an underscore (_). In that case, the underscore, the newline, and any whitespace at the beginning of the next line are not part of the string literal.

Table A-1
Escape Codes and Characters

Code	Character	Code	Character	Code	Character	Code	Character
\b	backspace	\d	delete	\e	escape	\f	form feed
\l	line feed	\n	newline	\r	carriage return	\t	tab
\v	vertical tab	\'	quote	\"	double quote	\\	backslash
ooo	octal	\x*hh*	hexadecimal	\^*x*	Control-*x*		

Cset

Csets are sets of 0 or more characters. Cset literals consist of 0 or more characters enclosed in single quotes. As with strings, a cset literal may include escape sequences that use multiple characters to encode special characters.

Pattern

Patterns are an immutable structure type used in matching, parsing or categorizing strings. Pattern literals consist of regular expressions enclosed in less than (<) and greater than (>) symbols. Within such marks, operators and reserved words do not have their normal meaning; instead concatenation becomes the implicit operator and a few characters have

special interpretations, including asterisk, plus, question mark, curly braces, square brackets, and the period character. In addition to pattern literals, patterns may be composed using a number of pattern constructor operators and functions.

A.2 Mutable Types: Containers and Files

Mutable types' values may be altered. Changes to a mutable value affect its allocated memory or its associated OS resource. Mutable types include lists, tables, sets, records, objects, and files, including windows, network connections and databases. These types are described in the entries for constructors that create them. Structure types hold collections of elements that may be of arbitrary, mixed type. Mutable types are not thread-safe.

List

Lists are dynamically sized, ordered sequences of zero or more values. They are constructed by function, by an explicit operator, or implicitly by a call to a variable argument procedure. They change size by stack and queue functions.

Table

Tables are dynamically sized, unordered mappings from keys to elements. They are constructed by function. The keys may be of arbitrary, mixed type.

Set

Sets are unordered collections. They are constructed by function.

Record

Records are ordered, fixed length sequences of elements accessed via named fields.

Object

Objects are ordered, fixed length sequences of elements that may be accessed via named fields and methods. Accessing an object's fields from outside its methods (using it as a record) is legal but deprecated.

File

Files are system resources corresponding to data on secondary storage, areas on users' displays, network connections, or databases. Operations on files cause input or output side

effects on the system outside of the program execution.

A.3 Variables

Variables are names for locations in memory where values can be stored. Values are stored in variables by assignment operators. A variable name begins with a letter or underscore, followed by zero or more letters, underscores, or digits. Variable names are case-sensitive. A variable name cannot be the same as one of Icon's reserved words, nor can it be the same as one of Icon's keywords if it follows an adjacent ampersand character. Variables can hold values of any type, and may hold different types of values at different times during program execution.

There are four kinds of variables: global, local, static, and class. Global, local, and static variables are declared by introducing one of the reserved words (global, local, or static) followed by a comma-separated list of variable names. Global variables are declared outside of any procedure or method body, while local and static variables are declared at the beginning of procedure and method bodies. Local and static variable names may be followed by an assignment operator and an initial value; otherwise variables other than procedure and class names begin with the value &null.

Aliasing occurs when two or more variables refer to the same value, such that operations on one variable might affect the other. Aliasing is a common source of program bugs. Variables holding integer, real, string, or cset values are never aliased, because those types are immutable.

Global

Global variables are visible everywhere in the program, and exist at the same location for the entire program execution. Declaring a procedure declares a global variable initialized to the procedure value that corresponds to the code for that procedure. Global variables are not thread-safe.

Local

Local variables exist and are visible within a single procedure or method only for the duration of a single procedure invocation, including suspensions and resumptions, until the procedure returns, fails, or is *vanquished* by the return or failure of an ancestor invocation while it is suspended. Undeclared variables in any scope are implicitly local, but this dangerous practice should be avoided in large programs.

Variables that are declared as *parameters* are local variables that are preinitialized to the values of actual parameters at the time of a procedure or method invocation. The semantics of parameter passing are the same as those of assignment.

Static

Static variables are visible only within a single procedure or method, but exist at the same location for the entire program execution. The value stored in a static variable is preserved between multiple calls to the procedure in which it is declared. Static variables are not thread-safe.

Class

Class variables are visible within the methods of a declared class. Class variables are created for each instance (object) of the class. The lifespan of class variables is the life span of the instance to which they belong. The value stored in a class variable is preserved between multiple calls to the methods of the class in which it is declared. Class variables are not thread-safe.

A.4 Keywords

Keywords are names with global scope and special semantics within the language. They begin with an ampersand character. Some keywords are names of common constant values, while others are names of variables that play a special role in Icon's control structures. The name of the keyword is followed by a : if it is read-only, or a := if it is a variable, followed by the type of value the keyword holds.

&allocated : integer* **report memory use**

&allocated generates the cumulative number of bytes allocated in heap, static, string, and block regions during the entire program execution.

&ascii : cset **ASCII character set**

&ascii produces a cset corresponding to the ASCII characters.

&clock : string **time of day**

&clock produces a string consisting of the current time of day in hh:mm:ss format. See also keyword &now.

&collections : integer* **garbage collection activity**

&collections generates the number of times memory has been reclaimed in heap, static, string, and block regions.

&column : integer **source code column**

&column returns the source code column number of the current execution point. This is especially useful for execution monitoring.

&cset : cset **universal character set**

&cset produces a cset constant corresponding to the universal set of all characters.

&current :co-expression **current co-expression**

&current produces the co-expression that is currently executing.

&date : string **today's date**

&date produces the current date in yyyy/mm/dd format.

&dateline : string **time stamp**

&dateline produces a human-readable time stamp that includes the day of the week, the date, and the current time, down to the minute.

&digits : cset **digit characters**

&digits produces a cset constant corresponding to the set of digit characters 0-9.

&dump := integer **termination dump**

&dump controls whether the program dumps information on program termination or not. If &dump is nonzero when the program halts, a dump of local and global variables and their values is produced.

&e : real **natural log e**

&e is the base of the natural logarithms, 2.7182818...

&errno : integer? **system error code**

&errno is the platform-specific error code for the previous failed system call, if there was one.

&error := integer **fail on error**

&error controls whether runtime errors are converted into expression failure. By assigning to this keyword, error conversion can be enabled or disabled for specific sections of code. The integer &error is decremented by one on each error, and if it reaches zero, a runtime error is generated. Assigning a value of -1 effectively disables runtime errors indefinitely. There is not one &error integer for each thread — the value applies to the whole program, not just the thread that sets it.

&errornumber : integer? **runtime error code**

&errornumber is the error number of the last runtime error that was converted to failure, if there was one.

&errortext : string? **runtime error message**

&errortext is the error message of the last error that was converted to failure.

&errorvalue : any? **offending value**

&errorvalue is the erroneous value of the last error that was converted to failure.

&errout : file **standard error file**

&errout is the standard error file. It is the default destination to which runtime errors and program termination messages are written.

&eventcode := integer **program execution event**

&eventcode indicates the kind of behavior that occurred in a monitored program at the time of the most recent call to EvGet(). This keyword is only supported under interpreters built with execution monitoring support.

&eventsource := co-expression **source of program execution events**

&eventsource is the co-expression that transmitted the most recent event to the current program. This keyword is null unless the program is an execution monitor. See also &source. Under a monitor coordinator, &eventsource is the coordinator and global variable Monitored is the target program.

&eventvalue := any **program execution value**

&eventvalue is a value from the monitored program that was being processed at the time of the last program event returned by EvGet(). This keyword is only supported under interpreters built with execution monitoring support.

&fail : none **expression failure**

&fail never produces a result. Evaluating it always fails.

&features : string* **platform features**

&features generates strings that indicate the non-portable features supported on the current platform.

&file : string? **current source file**

&file is the name of the source file for the current execution point, if there is one. This is especially useful for execution monitoring.

&host : string **host machine name**

&host is a string that identifies the host computer Icon is running on.

&input : file **standard input file**

&input is a standard input file. It is the default source for file input functions.

&lcase : cset **lowercase letters**

&lcase is a cset consisting of the lowercase letters from a to z.

&letters : cset **letters**

&letters is a cset consisting of the upper and lowercase letters A-Z and a-z.

&level : integer **call depth**

&level gives the nesting level of the currently active procedure call. This keyword is not supported under the optimizing compiler, iconc.

&line : integer **current source line number**

&line is the line number in the source code that is currently executing.

&main : co-expression **main task**

&main is the co-expression in which program execution began.

&now : integer **current time**

&now produces the current time as the number of seconds since the epoch beginning 00:00:00 GMT, January 1, 1970. See also &clock

&null : null **null value**

&null produces the null value.

&output : file **standard output file**

&output is the standard output file. It is the default destination for file output.

&phi : real **golden ratio**

&phi is the golden ratio, 1.618033988...

&pi : real **pi**

&pi is the value of pi, 3.141592653...

&pos := integer **string scanning position**

&pos is the position within the current subject of string scanning. It is assigned implicitly by entering a string scanning environment, moving or tabbing within the environment, or assigning a new value to &subject. &pos may not be assigned a value that is outside the range of legal indices for the current &subject string. Each thread has its own instance of &pos; assigning a value to it in one thread does not affect the string scanning environment of any another thread.

&progname := string **program name**

&progname is the name of the current executing program.

&random := integer **random number seed**

&random is the seed for random numbers produced by the random operator, unary ?. It is assigned a different sequence for each execution but may be explicitly set for reproducible results. Each thread has its own instance of &random; setting it in one thread does not affect the random sequence produced by another thread.

&regions : integer* **region sizes**

&regions produces the sizes of the static region, the string region, and the block region. The first result is zero; it is included for backward compatibility reasons.

&source : co-expression **invoking co-expression**

&source is the co-expression that activated the current co-expression.

&storage : integer* **memory in use**

&storage gives the amount of memory currently used within the static region, the string region, and the block region. The first result is always zero and is included for backward compatibility reasons.

&subject := string **string scanning subject**

&subject holds the default value used in string scanning and analysis functions. Assigning to &subject implicitly assigns the value 1 to &pos. Each thread has its own instance of &subject; assigning a value to it in one thread does not affect the string scanning environment of any another thread.

&time : integer **elapsed time**

&time gives the number of milliseconds of CPU time that have elapsed since the program execution began. For wall clock time see &now or &clock.

&trace := integer **trace program**

&trace specifies the number of procedure actions (call, return, suspend, resume, or fail) for which program execution will be traced. 0 means no tracing. A negative value enables tracing with no limit. &trace can be set outside the program using the TRACE environment variable, or set to -1 via the -t compiler option.

&ucase : cset **upper case letters**

&ucase is a cset consisting of all the upper case letters from A to Z.

&version : string **version**

&version is a string that indicates which version of Unicon or Icon is executing.

Graphics keywords

Several of the graphics keywords are variables with assignment restricted to value of a particular type or types. Graphics keywords are more fully described in [Griswold98].

&col := integer **mouse location, text column**

&col is the mouse location in text columns during the most recent Event(). If &col is assigned, &x gets a corresponding pixel location in the current font on &window.

&control : integer **control modifier flag**

&control produces the null value if the control key was pressed at the time of the most recently processed event, otherwise &control fails.

&interval : integer **time since last event**

&interval produces the time between the most recently processed event and the event that preceded it, in milliseconds.

&ldrag : integer **left mouse button drag**

&ldrag produces the integer that indicates a left button drag event.

&lpress : integer **left mouse button press**

&lpress produces the integer that indicates a left button press event.

&lrelease : integer **left mouse button release**

&lrelease produces the integer that indicates a left button release event.

&mdrag : integer **middle mouse button drag**

&mdrag produces the integer that indicates a middle button drag event.

&meta : integer **meta modifier flag**

&meta produces the null value if the meta (Alt) key was pressed at the time of the most recently processed event, otherwise &meta fails.

&mpress : integer **middle mouse button press**

&mpress produces the integer that indicates a middle button press event.

&mrelease : integer **middle mouse button release**

&mrelease produces the integer that indicates a middle button release event.

&pick : string* **pick 3D objects**

&pick generates the object IDs selected at point (&x,&y) at the most recent Event(), if the event was read from a 3D window with the attribute pick=on.

&rdrag : integer **right mouse button drag**

&rdrag produces the integer that indicates a right button drag event.

&resize : integer **window resize event**

&resize produces the integer that indicates a window resize event.

&row := integer **mouse location, text row**

&row is the mouse location in text rows during the most recent Event(). If &row is assigned, &y gets a corresponding pixel location in the current font on &window.

&rpress : integer **right mouse button press**

&rpress produces the integer that indicates a right button press event.

&rrelease : integer **right mouse button release**

&rrelease produces the integer that indicates a right button release event.

&shift : integer **shift modifier flag**

&shift produces the null value if the shift key was pressed at the time of the most recently processed event, otherwise &shift fails.

&window := window **default window**

&window is the default window argument for all window functions. &window may be assigned any value of type window.

&x := integer **mouse location, horizontal**

&x is the horizontal mouse location in pixels during the most recent Event(). If &x is assigned, &col gets a corresponding text coordinate in the current font on &window.

&y := integer **mouse location, vertical**

&y is the vertical mouse location in pixels during the most recent Event(). If &y is assigned, &row gets a corresponding text coordinate in the current font on &window.

A.5 Control Structures and Reserved Words

Unicon has many reserved words. Some are used in declarations, but most are used in control structures. This section summarizes the syntax and semantics introduced by all the reserved words of the language. The reserved word under discussion is written in a bold font. The surrounding syntax uses square brackets for optional items and an asterisk for items that may repeat.

---

abstract **declare unimplemented method**

The abstract reserved word declares that a named method must by provided by subclasses that implement a given class. The presence of one or more abstract methods implies that a class itself is abstract and should only be instantiated indirectly via a subclass that implements the abstract methods.

---

break expr **exit loop**

The break expression exits the nearest enclosing loop. *expr* is evaluated and treated as the result of the entire loop expression. If *expr* is another break expression, multiple loops will be exited.

---

expr1 to expr2 by expr3 **step increment**

The by reserved word supplies a step increment to a to-expression (the default is 1).

---

case expr of { ? } **select expression**

The case expression selects one of several branches of code to be executed.

---

class name [: superclass]* (fields) methods [initially] end **class declaration**

The class declaration introduces a new object type into the program. The class declaration may include lists of superclasses, fields, methods, and an initially section.

---

create expr **create co-expression**

The create expression produces a new co-expression to evaluate *expr*.

---

critical x : expr **serialize on x**

The critical expression serializes the execution of *expr* on value *x*. Value *x* must be a mutex or protected object that has a mutex. The critical section causes *x* to be locked before evaluating *e*xpr and unlocked afterward. Breaking, returning or failing out of *e*xpr does not automatically unlock *x*.

---

default : expr **default case branch**

The default branch of a case expression is taken if no other case branch is taken.

---

do expr **iteration expression**

The do reserved word specifies an expression to be executed for each iteration of a preceding while, every, or suspend loop (yes, suspend is a looping construct).

---

if expr1 then expr2 else expr3 **else branch**

The else expression is executed if *expr1* fails to produce a result.

---

end **end of declared body**

The reserved word end signifies the end of a procedure, method, or class body.

---

every *expr1* [do *expr2*] **generate all results**

The every expression always fails, causing *expr1* to be resumed for all its results.

---

fail **produce no results**

The fail reserved word causes the enclosing procedure or method invocation to terminate immediately and produce no results. The invocation may not be resumed. See also the keyword &fail, which produces a less drastic expression failure. fail is equivalent to return &fail

---

global *var* [, *var*]* **declare global variables**

Reserved word global introduces one or more global variables.

---

if *expr* then *expr2* [else *expr3*] **conditional expression**

The if expression evaluates *expr2* only if *expr1* produces a result.

---

import *name* [, *name*]* **import package**

The import declaration introduces the names from package *name* so that they may be used without prefixing them with the package name.

---

initial expr **execute on first invocation**

The initial expression is executed the first time a procedure or method is invoked. Any subsequent invocations (of the procedure or method) will not proceed until the initial expression has finished execution. A recursive invocation of the procedure inside the initial expression causes a runtime error.

---

initially [(parameters)] **initialize object**

The initially section defines a special method that is invoked automatically when an object is created. If the initially section has declared parameters, they are used as the parameters of the constructor for objects of that class.

---

invocable *procedure* [, *procedure*]* **allow string invocation**

invocable all **allow string invocation**

The invocable declaration indicates that procedures may be used in string invocation.

---

link *filename* [, *filename*]* **link code module**

The link declaration directs that the code in *filename* will be added to the executable when this program is linked. *filename* may be an identifier or a string literal file path.

---

local *var* [:=*initializer*] [, *var* [:= *initializer*]]* **declare local variables**

The local declaration introduces local variables into the current procedure or method. Variable declarations must be at the beginning of a procedure or method.

---

method name (params) body end **declare method**

The method declaration introduces a procedure that is invoked with respect to instances of a given class. The *params* and *body* are as in procedures, described below.

---

next **iterate loop**

The next expression causes a loop to immediate skip to its next iteration.

---

not expr **negate expression failure**

The not expression fails if *expr* succeeds, and succeeds (producing null) if *expr* fails.

---

case expr of { ? } **introduce case branches**

The of reserved word precedes a special compound expression consisting of a sequence of case branches of the form *expr* : *expr*. Case branches are evaluated in sequence until one matches the expression given between the word case and the of.

---

package name **declare package**

The package declaration segregates the global names in the current source file. In order to refer to them, client code must either import the package, or prepend *name.* (the package name followed by a period) onto the front of a name in the package.

---

procedure name (params) body end **declare procedure**

The procedure declaration specifies a procedure with parameters and code body. The parameters are a comma-separated list of zero or more variable names. The last parameter may be suffixed by [ ] to indicate that following parameters will be supplied to the procedure in a list. The body is an optional sequence of local and static variable declarations, followed by a sequence of zero or more expressions.

---

record name (fields) **declare record**

The record declaration introduces a new record type into the program.

---

repeat expr **infinite loop**

The repeat expression introduces an infinite loop that will reevaluate *expr* forever. Of course, *expr* may exit the loop or terminate the program in any number of ways.

---

return expr **return from invocation**

The return expression exits a procedure or method invocation, producing *expr* as its result. The invocation may not be resumed.

---

static *var* [, *var*]* **declare static variables**

The static declaration introduces local variables that persist for the entire program execution into the current procedure or method body. Variable declarations must be at the beginning of a procedure or method.

---

suspend *expr* [do *expr*] **produce result from invocation**

The suspend expression produces one or more results from an invocation for use by the calling expression. The procedure or method may be resumed for additional results if the calling expression needs them. Execution in the suspended invocation resumes where it left off, in the suspend expression. A single evaluation of a suspend expression may produce multiple results for the caller if *expr* is a generator. An optional do expression is evaluated each time the suspend is resumed.

---

if expr1 then expr2 **conditional expression**

The *expr2* following a **then** is evaluated only if *expr1* following an if succeeds. In that case, the result of the whole expression is the result of *expr2*.

---

thread expr **create thread**

The **thread** expression creates and launches a concurrent thread to evaluate *expr*.

---

expr1 to expr2 **generate arithmetic sequence**

The to expression produces the integer sequence from *expr1* to *expr2*. Neither *expr1* nor *expr2* may be a large integer.

---

until *expr1* [do *expr2*] **loop until success**

The until expression loops as long as *expr1* fails.

---

while *expr1* [do *expr2*] **loop until failure**

The while expression loops as long as *expr1* succeeds.

A.6 Operators and Built-in Functions

Icon's built-ins operators and functions utilize automatic type conversion to provide flexibility and ease of programming. Automatic type conversions are limited to integer, real, string, and cset data types. Conversions to a "number" will convert to either an integer or a real, depending whether the value to be converted has a decimal. Conversions between numeric types and csets go through an intermediate conversion to a string value and are not generally useful.

Indexes start at 1. Index 0 is the position after the last element of a string or list. Negative indexes are positions relative to the end. Subscripting operators and string analysis functions can take two indices to specify a section of the string or list. When two indices are supplied, they select the same string section whether they are in ascending or descending order.

Operators

The result types of operators are the same as the operand types except as noted.

Unary operators

! x : any* **generate elements**

The generate operator produces the elements of x. If x is a string variable or refers to a structure value, the generated elements are variables that may be assigned. !i is equivalent to (1 to i) for integer i. List, record, string, and file elements are generated in order, with string elements consisting of one-letter substrings. Set and table elements are generated in an undefined order. If x is a messaging connection to a POP server, !x produces complete messages as strings. Other types of files, including network connections, produce elements consisting of text lines. Care should be taken when generating the elements of a variable that might change during the generation.

/ x **null test**

\ x **nonnull test**

The null and nonnull tests succeed and produce their operand if it satisfies the test.

- number **negate**

+ number **numeric identity**

Negation reverses the sign of its operand. Numeric identity does not change its operand's value other than to convert to a required numeric type.

= pattern **anchored pattern match**

= string **tab/match**

The unary equals operator performs a pattern match on its operand in the current string scanning environment and advances the position beyond the matched string if successful. When the operand is a string, this is equivalent to calling tab(match(s)) on its operand.

*** x : integer?** **size**

The size operator returns the number of elements in string, cset, thread message queue or structure x. Other types are converted to a string and the size of the string is returned. Runtime error 112 occurs if the conversion to a string fails.

. x : x **dereference**

The dereference operator returns the value x.

? x : any **random element**

The random operator produces a random element from x. If x is a string, ?x produces a random one-letter substring. The result is a variable that may be assigned. If x is a positive integer, ?x produces a random integer between 1 and x. ?0 returns a real in the range from 0.0-1.0.

| x : x* **repeated alternation**

The repeated alternation operator generates results from evaluating its operand over and over again in an infinite loop.

~ cset **cset complement**

The complement operator returns a cset consisting of all characters not in its operand.

---

^ co-expression **refresh co-expression**

The refresh operator restarts a co-expression so the next time it is activated it will begin with its first result.

Binary operators

Most binary operators may be augmented with an assignment — see page 427 for the full list. If such an operator is followed by a := the left operand must be a variable, and the expression x *op*:= y is equivalent to x := x *op* y. For example, x +:= 5 is equivalent but faster than the expression x := x+5. In general, augmented operators are not thread-safe. They are only safe if applied to a local (non static) variable that has an atomic type. For example, sets are mutable (not safe anywhere) whereas csets are atomic (unsafe if global or static; safe if local).

---

number1* ^ *number2 **power**

number1* * *number2 **multiply**

number1* / *number2 **divide**

number1* % *number2 **modulo**

number1 **+** ***number2*** **add**
number1 **-** ***number2*** **subtract**
The arithmetic operators may be augmented.

---

set1 ** set2 **intersection**
set1 ++ set2 **union**
set1 -- set2 **difference**
The set operators work on sets, csets, or tables (via their keys). They may be augmented. In table union and intersection, the result table values are those of the left operand if available.

---

x . name **field**
object **. name (params)** **method invocation**
object **$ superclass .name (params)** **superclass method invocation**
The field operator selects field name out of a record, object, or package. For objects, *name* may be a method, in which case the field operator is being used as part of a method invocation. Superclass method invocation consists of a dollar sign and superclass name prior to the field operator.

---

number1 = number2 **equal**
number1 ~= number2 **not equal**
number1 < number2 **less than**
number1 <= number2 **less or equal**
number1 > number2 **greater than**
number1 >= number2 **greater or equal**
string1 == string2 **string equal**
string1 ~== string2 **string not equal**
string1 << string2 **string less than**
string1 <<= string2 **string less or equal**
string1 >> string2 **string greater than**
string1 >>= string2 **string greater or equal**
x1 === x2 **equivalence**
x1 ~=== x2 **non equivalence**
Relational operators produce their right operand if they succeed. They may be augmented.

---

var := expr **assign**
var1 :=: var2 **swap**
var <- expr **reversible assignment**
var1 <-> var2 **reversible swap**
The several assignment operators all require variables for their left operands, and swap operators also require variables for their right operands.

Assignment operators are usually thread safe but there are some situations where they

are not. See the discussion of thread safe assignment without a mutex (on page 150) for details. If in doubt, protect the global variable with a mutex.

---

string ? expr **scan string**

String scanning evaluates *expr* with &subject equal to string and &pos starting at 1. It may be augmented.

---

string ?? pattern **pattern match**

Pattern matching produces the substring(s) where *pattern* occurs within a string. It is conducted within a new string scanning environment as per string scanning above. It may be augmented.

---

x ! y **apply**

The binary bang (exclamation) operator calls x, using y as its parameters. x may be a procedure, or the string name of a procedure. y is a list or record.

---

[x] @ co-expression **activate co-expression**

The activate operator transfers execution control from the current co-expression to its right operand co-expression. The transmitted value is x, or &null if no left operand is supplied. Activation may be augmented.

---

[x] @> [y] **send message**
[x] @>> [y] **blocking send message**

The send operator places a message in another thread's public inbox, or in the current thread's public outbox. The normal version fails if the box is full; the blocking version waits for space to become available.

---

[x] <@ [y] **receive message**
[x] <<@ [y] **blocking receive message**

The receive operator obtains a message from another thread's public outbox, or the current thread's public inbox. The normal version fails if the box is empty; the blocking version waits for a message to become available.

---

string1 || string2 **concatenation**
pattern1 || pattern2 **pattern concatenation**
list1 ||| list2 **list concatenation**

The concatenation operators produce new values (or patterns that will match values) consisting of the left operand followed by the right operand. They may be augmented.

---

x1 & x2 **conjunction**
expr1 | expr2 **alternation**
pattern1 .| pattern2 **pattern alternation**

The conjunction operator produces x2 if x1 succeeds. Conjunction may be augmented. The alternation operator produces the results of expr1 followed by the results of expr2; it is a

generator. The pattern alternation operator produces a pattern that will match the results of pattern1 followed by the results of pattern2.

---

p -> v **conditional assignment**
p => v **immediate assignment**
.> v **cursor position assignment**

The conditional assignment operator assigns the substring matched by its left operand (a pattern) to a variable (its right operand) at the end of matching, if the whole pattern match succeeds. The immediate assignment operator assigns the substring matched by its left operand (a pattern) to a variable (its right operand) at the point during the match that the pattern match of the left operand occurs, whether or not the whole match succeeds. The cursor position assignment operator assigns the cursor position at a point during a pattern match to a variable (its operand).

---

x1 \ integer **limitation**

The limitation operator fails if it is resumed after its left operand has produced a number of results equal to its right operand.

(expr [, expr]*) **mutual evaluation**
p (expr [, expr]*) **invocation**

By themselves, parentheses are used to override operator precedence in surrounding expressions. A comma-separated list of expressions is evaluated left to right, and fails if any operand fails. Its value is the right of the rightmost operand.

When preceded by an operand, parentheses form an invocation. The operand may be a procedure, a method, a string that is converted to a procedure name, or an integer that selects the parameter to use as the result of the entire expression.

[ ] **empty list creation**
[expr [, expr]*] **list creation**
[: expr :] **list comprehension**
[expr : expr [; expr : expr]*] **initialized table creation**
expr1 [expr2 [, expr]*] **subscript**
expr1 [expr2 : expr3] **subsection**
expr1 [expr2 +: expr3] **forward relative subsection**
expr1 [expr2 -: expr3] **backward relative subsection**

With no preceding operand, square brackets create and initialize lists. Initializer values are comma-separated, except in list comprehension where the expression's values (obtained as if by **every**) are used to provide the initial list elements. When preceded by an operand, square brackets form a subscript or subsection. Multiple comma-separated subscript operands are equivalent to separate subscript operations with repeating square brackets, so x[y,z] is equivalent to x[y][z].

Subscripting selects an element from a structure and allows that element to be assigned or for its value to be used. Lists and strings are subscripted using 1-based integer indices, tables are subscripted using arbitrary keys, and records may be subscripted by either string fieldname or 1-based integer index. Message connections may be subscripted by string header to obtain server responses; POP connections may also be subscripted by 1-based integer message numbers.

Subsectioning works on strings and lists. For strings, the subsection is a variable if the string was a variable, and assignment to the subsection makes the variable hold the new, modified string constructed by replacing the subsection. For lists, a subsection is a new list that contains a copy of the elements from the original list.

---

expr1 ; expr2 **bound expression**

A semicolon bounds expr1. Once expr2 is entered, expr1 cannot be resumed for more results. The result of expr2 is the result of the entire expression. Semicolons are automatically inserted at ends of lines wherever it is syntactically allowable to do so. This results in many *implicitly bounded* expressions.

{ expr [; expr]* } **compound expression**

p { expr [, expr]* } **programmer defined control structure**

Curly brackets typically cause a sequence of bounded expressions to be treated as a single expression. Preceded by a procedure value, curly brackets introduce a programmer defined control structure in which a co-expression is created for each argument; the procedure is called with these co-expressions as its parameters, and can determine for itself whether, and in what order, to activate its parameters to obtain values.

Built-in functions

Unicon's built-in functions are a key element of its ease of learning and use. They provide substantial functionality in a consistent and easily memorized manner.

In addition to automatic type conversion, built-in functions make extensive use of optional parameters with default values. Default values are indicated in the function descriptions, with the exception of string scanning functions. String scanning functions end with three parameters that default to the string &subject, the integer &pos, and the end of string (0) respectively. The position argument defaults to 1 when the string argument is supplied rather than defaulted.

---

abs(N) : number **absolute value**

abs(N) produces the maximum of N or -N.

---

acos(r) : real **arc cosine**

acos(r) produces the arc cosine of r. The argument is given in radians.

---

any(c, s, i, i) : integer? **cset membership**

String scanning function any(c,s,i1,i2) produces min(i1,i2)+1 if s[min(i1,i2)] is in cset c, but fails otherwise.

---

args(x,i) : any **number of arguments**

args(p) produces the number of arguments expected by procedure p. If p takes a variable number of arguments, args(p) returns a negative number to indicate that the final argument is a list conversion of an arbitrary number of arguments. For example, args(p) for a procedure p with formal parameters (x, y, z[ ]) returns a -3. args(C) produces the number of arguments in the current operation in co-expression C, and args(C,i) produces argument number i within co-expression C.

---

asin(real) : real **arc sine**

asin(r1) produces the arc sine of r1. The argument is given in radians.

---

atan(r, r:1.0) : real **arc tangent**

atan(r1) produces the arc tangent of r1. atan(r1,r2) produces the arc tangent of r1 and r2. Arguments are given in radians.

---

atanh(r) : real **inverse hyperbolic tangent**

atanh(r) produces the inverse hyperbolic tangent of r. Arguments are given in radians.

---

bal(cs:&cset, cs:'(', cs:')', s, i, i) : integer* **balance string**

String scanning function bal(c1,c2,c3,s,i1,i2) generates the integer positions in s at which a member of c1 in s[i1:i2] is balanced with respect to characters in c2 and c3.

---

center(s, i:1, s:" ") : string **center string**

center(s1,i,s2) produces a string of i characters. If i > *s1 then s1 is padded equally on the left and right with s2 to length i. If i < *s1 then the center i characters of s1 are produced.

---

channel(TH) : list **communications channel**

channel(TH) creates a communications channel between the current thread and thread TH.

---

char(i) : string **encode character**

char(i) produces a string consisting of the character encoded by integer i.

---

chdir(s) : string **change directory**

chdir(s) changes the current working directory to s. chdir() returns the current working directory, which is shared between threads.

---

chmod(f, m) : ? **file permissions**

chmod(f, m) sets the access permissions ("mode") of a string filename (or on UNIX systems, an open file) f to a string or integer mode m. The mode indicates the change to be performed. The string is of the form

```
[ugoa]*[+-=][rwxRWXstugo]*
```

The first group describes the set of mode bits to be changed: u is the owner set, g is the group and o is the set of all others. The character a designates all the fields. The operator (+-=) describes the operation to be performed: + adds a permission, - removes a permission, and = sets a permission. The permissions themselves are:

r	read
w	write
x	execute
R	read if any other set already has r
W	write if any other set already has w
X	execute if any other set already has x
s	setuid (if the first part contains u and/or setgid if the first part contains g
t	sticky if the first part has o
u	the u bits on the same file
g	the g bits on the same file
o	the o bits on the same file

If the first group is missing, then it is treated as "all" except that any bits in the user's umask will not be modified in the mode. Not all platforms make use of all mode bits described here; the mode bits that are used is a property of the filesystem on which the file resides.

---

classname(r) : string **class name**

classname(r) produces the name of r's class.

---

close(f) : file | integer **close file**

close(f) closes file, pipe, window, network or message connection, or database f and returns any resources associated with it to the operating system. If f was a window, close(f) causes it to disappear, but the window can still be written to and copied from until all open bindings are closed. If f was a pipe or network connection, close() returns the integer exit status of the connection, otherwise it returns the closed file.

---

cofail(CE) : any **transmit co-expression failure**

cofail(ce) activates co-expression ce, transmitting failure instead of a result.

---

collect(i:0, i:0) : null **collect garbage**

collect(i1,i2) calls the garbage collector to ensure that i2 bytes are free in region i1. i1 can be 0 (no region in particular) 1 (static region) 2 (string region) or 3 (block region).

---

condvar() : condition variable **create condition variable**

condvar() creates a new condition variable.

---

constructor(s, ...) : procedure **record constructor**

constructor(label, field, field, ...) creates a new record type named label with fields named by its subsequent arguments, and returns a constructor procedure for this record type.

---

copy(any) : any **copy value**

copy(x) produces a copy of x. For immutable types (numbers, strings, csets, procedures) this is a no-op. For mutable types (lists, tables, sets, records, objects) a one-level deep copy of the object is made.

---

cos(r) : real **cosine**

cos(r) produces the cosine of r. The argument is given in radians.

---

cset(any) : cset? **convert to cset**

cset(x) converts x to a cset, or fails if the conversion cannot be performed.

---

ctime(i) : string **format a time value into local time**

ctime(i) converts an integer time given in seconds since the epoch, Jan 1, 1970 00:00:00 into a string in the local timezone. See also keywords &clock and &dateline.

---

dbcolumns(D,s) : list **ODBC column information**

dbcolumns(db, tablename) produces a list of record entries with fields: catalog, schema, tablename, colname, datatype, typename, colsize, buflen, decdigits, numprecradix, nullable, and remarks.The fields datatype and typename are SQL-dependent and data source dependent, respectively. Field colsize gives the maximum length in characters for SQL_CHAR or SQL_VARCHAR columns.. Field decdigits gives the number of significant digits right of the decimal. Field numprecradix specifies whether colsize and decdigits are specified in bits or decimal digits. Field nullable is 0 if the column does not accept null values, 1 if it does accept null values, and 2 if it is not known whether the column accepts null values.

---

dbdriver(D) : record **ODBC driver information**

dbdriver(db) produces a record with fields name, ver, odbcver, connections, statements, and dsn that describes the details of the ODBC driver used to connect to database db. Fields connections and statements are the maximums the driver can support. Fields ver and odbcver are the driver and ODBC version numbers. Fields name and dsn are the driver filename and Windows Data Source Name associated with the connection.

---

dbkeys(D,string) : list **ODBC key information**

dbkeys(db,tablename) produces a list of records with fields columnname, and sequencenumber containing information about the primary keys in tablename.

---

dblimits(D) : record **ODBC operation limits**

dblimits(db) produces a record with fields maxbinlitlen, maxcharlitlen, maxcolnamelen, maxgroupbycols, maxorderbycols, maxindexcols, maxselectcols, maxtblcols, maxcursnamelen, maxindexsize, maxrownamelen, maxprocnamelen, maxqualnamelen, maxrowsize, maxrowsizelong, maxstmtlen,

maxtblnamelen, maxselecttbls, and maxusernamelen that contains the upper bounds of the database for many parameters.

---

dbproduct(D) : record **database name**

dbproduct(db) produces a record with fields name and ver that gives the name and the version of the DBMS product containing db.

---

dbtables(D) : list **ODBC table information**

dbtables(db) returns a list of records with fields qualifier, owner, name, type, and remarks that describe all of the tables in the database associated with db.

---

delay(i) : null **delay for i milliseconds**

delay(i) pauses the current thread for at least i milliseconds. Runtime error 101 occurs if i is a large integer.

---

delete(x1, x2, ...) : x1 **delete element**

delete(x1, x2) deletes elements denoted by the 2nd and following parameters from set, table, list, DBM database, or POP connection x1 if it is there. In any case, it returns x1. If x1 is a table or set, elements x_i denote keys of arbitrary type. If x1 is a DBM database, indices must be strings. If x1 is a list or a POP messaging connection, the elements x_i are integer indices of elements to be deleted. POP messages are actually deleted when the close() operation closes that connection.

---

detab(string, integer:9,...) : string **replace tabs**

detab(s,i,...) replaces tabs with spaces, with stops at columns indicated by the second and following parameters, which must all be integers. Tab stops are extended infinitely using the interval between the last two specified tab stops.

---

display(i:&level, f:&errout, CE:&current) : null **write variables**

display(i,f) writes the local variables of i most recent procedure activations, plus global variables, to file f.

---

dtor(r) : real **convert degrees to radians**

dtor(r) produces the equivalent of r degrees, expressed in radians.

---

entab(s, i:9,...) : string **replace spaces**

entab(s,i,...) replaces spaces with tabs, with stops at columns indicated. Tab stops are extended infinitely using the interval between the last two specified tab stops.

---

errorclear() : null **clear error condition**

errorclear() resets keywords &errornumber, &errortext, and &errorvalue to indicate that no error is present.

---

eventmask(CE, cset) : cset | null **get/set event mask**

eventmask(ce) returns the event mask associated with the program that created ce, or &null if there is no event mask. eventmask(ce,cs) sets that program's event mask to cs.

EvGet(c, flag) : string **get event from monitored program**

EvGet(c,flag) activates a program being monitored until an event in cset mask c occurs. Under normal circumstances this is a one-character string event code.

EvSend(i, x, CE) : any **transmit event**

EvSend(x, y, C) transmits an event with event code x and event value y to a monitoring co-expression C.

---

exit(i:normalexit) **exit process**

exit(i) terminates the current program execution, returning status code i. The default is the platform-dependent exit code that indicates normal termination (0 on most systems). Runtime error 101 occurs if i is a large integer.

---

exp(r) : real **exponential**

exp(r) produces the result of &e ^ r.

---

fetch(D, s?) : string | row? **fetch database value**

fetch(d, k) fetches the value corresponding to key k from a DBM or SQL database d. The result is a string (for DBM databases) or a row (for SQL databases). For SQL databases, when the string k is omitted, fetch(d) produces the next row in the current selection, and advances the cursor to the next row. A row is a record whose field names and types are determined by the columns specified in the current query. fetch(d) fails if there are no more rows to return from the current query. Typically a call to dbselect() will be followed by a while-loop that calls fetch() repeatedly until it fails.

---

fieldnames(R) : string* **get field names**

fieldnames(r) produces the names of the fields in record r.

---

find(s, s, i, i) : integer* **find string**

String scanning function find(s1,s2,i1,i2) generates the positions in s2 at which s1 occurs as a substring in s2[i1:i2].

---

flock(f, s) : ? **apply or remove file lock**

flock(f,s) applies an advisory lock to the file. Advisory locks enable processes to cooperate when accessing a shared file, but do not enforce exclusive access. The following characters can be used to make up the operation string:

s	shared lock
x	exclusive lock
b	don't block when locking
u	unlock

Locks cannot be applied to windows, directories or database files. A file may not simultaneously have shared and exclusive locks.

---

flush(f) : file **flush file**

flush(f) flushes all pending or buffered output to file f.

---

function() : string* **name the functions**

function() generates the names of the built-in functions.

get(L,i:1) : any? **get element from queue**

get(L) returns an element which is removed from the head of the queue L. get(L, i) removes i elements, returning the last one removed.

---

getch() : string? **get character from console**

getch() waits for (if necessary) and returns a character typed at the keyboard, even if standard input was redirected. The character is not displayed.

---

getche() : string? **get and echo character from console**

getche() waits for (if necessary) and returns a character typed at the console keyboard, even if standard input was redirected. The character is echoed to the screen.

---

getenv(s) : string? **get environment variable**

getenv(s) returns the value of environment variable s from the operating system.

---

gettimeofday() : record **time of day**

Returns the current time in seconds and microseconds since the epoch, Jan 1, 1970 00:00:00. The sec value may be converted to a date string with ctime or gtime. See also keywords &now, &clock, and &dateline. Return value: record posix_timeval(sec, usec)

---

globalnames(CE) : string* **name the global variables**

globalnames(ce) generates the names of the global variables in the program that created co-expression ce.

---

gtime(i) : string **format a time value into UTC**

Converts an integer time in seconds since the epoch, Jan 1, 1970 00:00:00 into a string in Coordinated Universal Time (UTC).

---

iand(i, i) : integer **bitwise and**

iand(i1, i2) produces the bitwise AND of i1 and i2.

---

icom(i) : integer **bitwise complement**

icom(i) produces the bitwise complement (one's complement) of i.

---

image(any) : string **string image**

image(x) returns the string image of the value x.

insert(x1, x2, x3:&null) : x1 **insert element**

insert(x1, x2, x3) inserts element x2 into set, table, or list or DBM database x1 if not already there. Unless x1 is a set, the assigned value for element x2 is x3. For lists, x2 is an integer index; for other types, it is a key. insert() always succeeds and returns x1.

---

integer(any) : integer? **convert to integer**

integer(x) converts value x to an integer, or fails if the conversion cannot be performed.

---

ior(i, i) : integer **bitwise or**

ior(i1, i2) produces the bitwise OR of i1 and i2.

---

ishift(i, i) : integer **bitwise shift**

ishift(i, j) produces the value obtained by shifting i by j bit positions. If j is positive, the shift is to the left, and vacated bit positions are filled with zeros. If j is negative, the shift is to the right with sign extension.

---

istate(CE, s) : integer **interpreter state**

istate(ce, attrib) reports selected virtual machine interpreter state information. attrib must be one of: "count", "ilevel", "ipc", "ipc_offset", "sp", "efp", "gfp". Used by monitors.

---

ixor(i, i) : integer **bitwise xor**

ixor(i1, i2) produces the bitwise exclusive or of i1 and i2.

---

kbhit() : ? **check for console input**

kbhit() checks to see if there is a keyboard character waiting to be read.

---

key(x) : any* **table keys**

key(T) generates the key (entry) values from table T. key(L) generates the indices from 1 to *L in list L. key(R) generates the string field names of record R. key(D) generates the string key values of a DBM database D.

---

keyword(s,CE:&current,i:0) : any* **produce keyword value**

keyword(s,ce,i) produces the value of keyword s in the context of ce's execution, i levels up in the stack from the current point of execution. Used in execution monitors.

---

left(s, i:1, s:" ") : string **left format string**

left(s1,i,s2) formats s1 to be a string of length i. If s1 is more than i characters, it is truncated. If s1 is fewer than i characters it is padded on the right with as many copies of s2 as needed to increase it to length i.

---

list(integer:0, any:&null) : list **create list**

list(i, x) creates a list of size i, in which all elements have the initial value x. If x is a mutable value such as a list, all elements refer to the *same* value, not a separate copy of the value for each element.

load(s,L,f:&input,f:&output,f:&errout,i,i,i) : co-expression load Unicon program

load(s,arglist,input,output,error,blocksize,stringsize,stacksize) loads the icode file named s and returns that program's execution as a co-expression ready to start its main() procedure with parameter arglist as its command line arguments. The three file parameters are used as that program's &input, &output, and &errout. The three integers are used as its initial memory region sizes.

---

loadfunc(s, s) : procedure load C function

loadfunc(filename,funcname) dynamically loads a compiled C function from the object library file given by filename. funcname must be a specially written interface function that handles Icon data representations and calling conventions.

---

localnames(CE, i:0) : string* local variable names

localnames(ce,i) generates the names of local variables in co-expression ce, i levels up from the current procedure invocation. The default i of 0 generates names in the currently active procedure in ce.

---

lock(x) : x lock mutex

lock(x) locks the mutex x or the mutex associated with thread-safe object x. Mutexes are recursive (i.e. they may be locked again by the same co-expression or thread without blocking) but must be unlocked as many times as they are locked. It is an error to unlock a mutex more times than it has been locked.

---

log(r, r:&e) : real logarithm

log(r1,r2) produces the logarithm of r1 to base r2.

---

many(c, s, i, i) : integer? many characters

String scanning function many(c,s,i1,i2) produces the position in s after the longest initial sequence of members of c within s[i1:i2].

---

map(s, s:&ucase, s:&lcase) : string map string

map(s1,s2,s3) maps s1, using s2 and s3. The resulting string will be a copy of s1, with the exception that any of s1's characters that appear in s2 are replaced by characters at the same position in s3.

---

match(s, s:&subject, i:&pos, i:0) : integer? match string

String scanning function match(s1,s2,i1,i2) produces i1+*s1 if s1==s2[i1+:*s1], but fails otherwise.

---

max(n, ...) : number largest value

max(x, ...) returns the largest value among its arguments, which must be numeric.

---

member(x, ...) : x? **test membership**

member(x, ...) returns x if its second and subsequent arguments are all members of set, cset, list, table or record x but fails otherwise. If x is a cset, all of the characters in subsequent string arguments must be present in x in order to succeed.

---

membernames(x) : list **class member names**

membernames(x) produces a list containing the string names of the fields of x, where x is either an object or a string name of a class.

---

methodnames(x) : list **class method names**

methodnames(x) produces a list containing the string names of the methods defined in class x, where x is either an object or a string name of a class.

---

methods(x) : list **class method list**

methods(x) produces a list containing the procedure values of the methods of x, where x is either an object or a string name of a class.

---

min(n, ...) : number **smallest value**

min(x, ...) returns the smallest value among its arguments, which must be numeric.

---

mkdir(s, s?) : ? **create directory**

mkdir(path,mode) creates a new directory named path with mode mode. The optional mode parameter can be numeric or a string of the form accepted by chmod(). The function succeeds if a new directory is created.

---

move(i:1) : string **move scanning position**

move(i) moves &pos i characters from the current position and returns the substring of &subject between the old and new positions. This function reverses its effects by resetting the position to its old value if it is resumed.

---

mutex(x,y) : x **create a mutex**

mutex() creates a new mutex. For mutex(x) associates the new mutex with structure x. The call mutex(x,y) associates an existing mutex y (or mutex associated with protected object y) with structure x.

---

name(v, CE:&current) : string **variable name**

name(v) returns the name of variable v within the program that created co-expression c. Keyword variables are recognized and named correctly. name() returns the base type and subscript or field information for variables that are elements within other values, but does not produce the source code variable name for such variables.

---

numeric(any) : number **convert to number**

numeric(x) produces an integer or real number resulting from the type conversion of x, but fails if the conversion is not possible.

---

open(s, s:"rt", ...) : file? **open file**

open(s1, s2, ...) opens a file named s1 with mode s2 and attributes given in trailing arguments. The modes recognized by open() are:

"a" append; write after current contents
"b" open for both reading and writing (b does not mean binary mode!)
"c" create a new file and open it
"d" open a [NG]DBM database
"g" create a 2D graphics window
"gl" create a 3D graphics window
"n" connect to a remote TCP network socket
"na" accept a connection from a TCP network socket
"nau" accept a connection from a UDP network socket
"nl" listen on a TCP network socket
"nu" connect to a UDP network socket
"e" use SSL/TLS protocol to encrypt the network socket
"m" connect to a messaging server (HTTP, HTTPS, SMTP, POP, ...)
"o" open an ODBC connection to a (typically SQL) database
"p" execute a program given by command line s1 and open a pipe to it
"r" read
"t" use text mode, with newlines translated
"u" use a binary untranslated mode
"w" write

Directories may only be opened for reading, and produce the names of all files, one per line. Pipes may be opened for reading or writing, but not both. open() fails if the pipe is open for reading and the command line given by s1 produces no output: &errornumber may be used to distinguish between a successful command that produces no output and a command that returns a non zero (unsuccessful) exit code.

When opening a network socket: the first argument s1 is the name of the socket to connect. If s1 is of the form "s:i", it is an Internet domain socket on host s and port i; otherwise, it is the name of a Unix domain socket. If the host name is null, it represents the current host. Mode "n" allows an optional third parameter, an integer timeout (in milliseconds) after which open() fails if no connection has been established by that time.

For a UDP socket, there is not really a connection, but any writes to that file will send a datagram to that address, so that the address doesn't have to be specified each time. Also, read() or reads() cannot be performed on a UDP socket; use receive. UDP sockets must be in the INET domain; the address must have a colon.

For a DBM database, only one modifier character may be used: if s1 is "dr" it indicates that the database should be opened in read-only mode. For an ODBC database, following the mode letter "o" comes an optional string default table name used by functions such as

dbcolumns(), followed by two generally required strings giving the username and password authentication for the connection.

The filename argument is a Uniform Resource Indicator (URI) when opening a messaging connection. Mode "m-" may be given to skip the validation of an encryption certificate for HTTPS connections. Arguments after the mode "m" are sent as headers. The HTTP User-Agent header defaults to "Unicon Messaging/10.0" and Host defaults to the host and port indicated in the URI. The SMTP From: header obtains its default from a UNICON_USERADDRESS environment variable if it is present.

For 2D and 3D windows, attribute values may be specified in the following arguments to open(). open() fails if a window cannot be opened or an attribute cannot be set to a requested value.

---

opmask(CE, c) : cset **opcode mask**

opmask(ce) gets ce's program's opcode mask. The function returns &null if there is no opcode mask. opmask(ce,cs) sets ce's program's opcode mask to cs. This function is part of the execution monitoring facilities.

---

oprec(x) : record **get methods vector**

oprec(r) produces a variable reference for r's class' methods vector.

---

ord(s) : integer **ordinal value**

ord(s) produces the integer ordinal (value) of s, which must be of size 1.

---

paramnames(CE, i:0) : string* **parameter names**

paramnames(ce,i) produces the names of the parameters in the procedure activation i levels above the current activation in ce.

---

parent(CE) : co-expression **parent program**

parent(ce) returns &main for ce's parent program. This is interesting only when programs are dynamically loaded using the load() function.

---

pipe() : list **create pipe**

pipe() creates a pipe and returns a list of two file objects. The first is for reading, the second is for writing. See also function filepair().

---

pop(L | Message) : any? **pop from stack**

pop(L) removes an element from the top of the stack (L[1]) and returns it. pop(M) removes and returns the first message in POP mailbox connection M; the actual deletion occurs when the messaging connection is closed.

---

pos(i) : integer? **test scanning position**

pos(i) tests whether &pos is at position i in &subject.

---

proc(any, i:1, C) : procedure? **convert to procedure**

proc(s,i) converts s to a procedure if that is possible. Parameter i is used to resolve ambiguous string names; it must be either 0, 1, 2, or 3. If i is 0, a built-in function is returned if it is available, even if the global identifier by that name has been assigned differently. If i is 1, 2, or 3, the procedure for an operator with that number of operands is produced. For example, proc("-",2) produces the procedure for subtraction, while proc("-") produces the procedure for unary negation. proc(C,i) returns the procedure activated i levels up with C. proc(p, i, C) returns procedure p if it belongs to the program which created co-expression C.

pull(L,i:1) : any? **remove from list end**

pull(L) removes and produces an element from the end of a nonempty list L. pull(L, i) removes i elements, producing the last one removed.

push(L, any, ...) : list **push on to stack**

push(L, x1, ..., xN) pushes elements onto the beginning of list L. The order of the elements added to the list is the reverse of the order they are supplied as parameters to the call to push(). push() returns the list that is passed as its first parameter, with the new elements added.

put(L, x1, ..., xN) : list **add to list end**

put(L, x1, ..., xN) puts elements onto the end of list L.

read(f:&input) : string? **read line**

read(f) reads a line from file f. The end of line marker is discarded.

reads(f:&input, i:1) : string? **read characters**

reads(f,i) reads up to i characters from file f. It fails on end of file. If f is a network connection, reads() returns as soon as it has input available, even if fewer than i characters were delivered. If i is -1, reads() reads and produces the entire file as a string. Care should be exercised when using this feature to read very large files.

ready(f:&input, i:0) : string? **non-blocking read**

ready(f,i) reads up to i characters from file f. It returns immediately with available data and fails if no data is available. If i is 0, ready() returns all available input. It is not currently implemented for window values.

real(any) : real? **convert to real**

real(x) converts x to a real, or fails if the conversion cannot be performed.

receive(f) : record **receive datagram**

receive(f) reads a datagram addressed to the port associated with f, waiting if necessary. The returned value is a record of type posix_message(addr, msg), containing the address of the sender and the contents of the message respectively.

remove(s) : ? **remove file**

remove(s) removes the file named s.

---

rename(s, s) : ? **rename file**

rename(s1,s2) renames the file named s1 to have the name s2.

---

repl(x, i) : x **replicate**

repl(x, i) concatenates and returns i copies of string or list x.

---

reverse(x) : x **reverse sequence**

reverse(x) returns a value that is the reverse of string or list x.

---

right(s, i:1, s:" ") : string **right format string**

right(s1,i,s2) produces a string of length i. If i<*s1, s1 is truncated. Otherwise, the function pads s1 on left with s2 to length i.

---

rmdir(s) : ? **remove directory**

rmdir(d) removes the directory named d. rmdir() fails if d is not empty or does not exist.

---

rtod(r) : real **convert radians to degrees**

rtod(r) produces the equivalent of r radians, expressed in degrees.

---

runerr(i, any) **runtime error**

runerr(i,x) produces runtime error i with value x. Program execution is terminated.

---

seek(f, any) : file? **seek to file offset**

seek(f,i) seeks to offset i in file f, if it is possible. If f is a regular file, i must be an integer. If f is a database, i seeks a position within the current set of selected rows. The position is selected numerically if i is convertible to an integer; otherwise i must be convertible to a string and the position is selected associatively by the primary key.

---

select(x1, x2, ?) : list **files with available input**

select(files?, timeout) waits for a input to become available on any of several files, typically network connections or windows. Its arguments may be files or lists of files, ending with an optional integer timeout value in milliseconds. It returns a list of those files among its arguments that have input waiting.

If the final argument to select() is an integer, it is an upper bound on the time elapsed before select returns. A timeout of 0 causes select() to return immediately with a list of files on which input is currently pending. If no files are given, select() waits for its timeout to expire. If no timeout is given, select() waits forever for available input on one of its file arguments. Directories and databases cannot be arguments to select().

---

send(s, s) : ? **send datagram**

send(s1, s2) sends a UDP datagram to the address s1 (in host:port format) with the contents s2.

---

seq(i:1, i:1) : integer* **generate sequence**

seq(i, j) generates the progression i, i+j, i+2*j, ... j may not be 0. Runtime error 101 occurs if either i or j is a large integer. Runtime error 203 occurs if the value to be generated is a large integer.

---

serial(x) : integer? **structure serial number**

serial(x) returns the serial number for structure x, if it has one. Serial numbers uniquely identify structure values. serial() returns the serial number of the current co-expression (or thread).

---

set(x, ...) : set **create set**

set() creates a set. Arguments are inserted into the new set, with the exception of lists. set(L) creates a set whose members are the elements of list L.

---

setenv(s) : ? **set environment variable**

setenv() sets an environment variable s in the operating system.

---

signal(cv, i:1) : ?? **signal a conditional variable**

signal(x, y) signals the condition variable x. If y is supplied, the condition variable is signaled y times. If y is 0, a "broadcast" signal is sent waking up all threads waiting on x. Condition variables have no memory: signalling a condition variable that has no threads waiting on it has no effect.

---

sin(r) : real **sine**

sin(r) produces the sine of r. The argument is given in radians.

---

sort(x, i:1) : list **sort structure**

sort(x, i) sorts structure x in ascending order. If x is a table, parameter i is the sort method. If i is 1 or 2, the table is sorted into a list of lists of the form [key, value]. If i is 3 or 4, the table is sorted into a list of alternating keys and values. Sorting is by keys for odd-values of i, and by table element values for even-values of i.

---

sortf(x, i:1) : list **sort by field**

sortf(x,i) sorts a list, record, or set x using field i of each element that has one. Elements that don't have an i'th field are sorted in standard order and come before those that do have an i'th field.

---

spawn(CE, i, i) : thread **launch asynchronous thread**

spawn(ce) launches co-expression ce as an asynchronous thread that will execute concurrently with the current co-expression. The two optional integers specify the memory in bytes allocated for the thread's block and string regions. The defaults are 10% of the main thread heap size.

---

sql(D, s) : integer **execute SQL statement**

sql(db, query) executes arbitrary SQL code on db. This function allows the program to do vendor-specific SQL and many SQL statements that cannot be expressed otherwise using the Unicon database facilities. sql() can leave the database in an arbitrary state and should be used with care.

---

sqrt(r) : real **square root**

sqrt(r) produces the square root of r.

---

stat(f) : record? **get file information**

stat(f) returns a record with information about the file f which may be a path or a file object. The return value is of type: record posix_stat(dev, ino, mode, nlink, uid, gid, rdev, size, atime, mtime, ctime, blksize, blocks, symlink). Many of these fields are POSIX specific, but a number are supported across platforms, such as the file size in bytes (the size field), access permissions (the mode field), and the last modified time (the mtime field).

The atime, mtime, and ctime fields are integers that may be formatted with the ctime() and gtime() functions. The mode is a string similar to the long listing option of the UNIX ls(1) command. For example, "-rwxrwsr-x" represents a plain file with a mode of 2775 (octal). stat(f) fails if filename or path f does not exist.

---

staticnames(CE:&current, i:0) : string* **static variable names**

staticnames(ce,i) generates the names of static variables in the procedure i levels above the current activation in ce.

---

stop(s|f, ...) : **stop execution**

stop(args) halts execution after writing out its string arguments, followed by a newline, to &errout. If any argument is a file, subsequent string arguments are written to that file instead of &errout. The program exit status indicates that an error has occurred.

---

string(x) : string? **convert to string**

string(x) converts x to a string and returns the result, or fails if the value cannot be converted.

---

system(x, f:&input, f:&output, f:&errout, s) : integer **execute system command**

system(x, f1, f2, f3, waitflag) launches execution of a program in a separate process. x can be either a string or a list of strings. In the former case, whitespace is used to separate the arguments and the command is processed by the platform's command interpreter. In the second case, each member of the list is an argument and the second and subsequent list elements are passed unmodified to the program named in the first element of the list.

The three file arguments are files that will be used for the new process' standard input, standard output and standard error. The return value is the exit status from the process. If the waitflag argument is "nowait", system() returns immediately after spawning the new process, and the return value is then the process id of the new process.

---

tab(i:0) : string? **set scanning position**

tab(i) sets &pos to i and returns the substring of &subject spanned by the former and new positions. tab(0) moves the position to the end of the string. This function reverses its effects by resetting the position to its old value if it is resumed.

---

table(k,v, ..., x) : table **create table**

table(x) creates a table with default value x. If x is a mutable value such as a list, all references to the default value refer to the *same* value, not a separate copy for each key. Given more than one argument, table(k,v,...x) takes alternating keys and values and populates the table with these initial contents.

---

tan(r) : real **tangent**

tan(r) produces the tangent of r in radians.

---

trap(s, p) : procedure **trap or untrap signal**

trap(s, proc) sets up a signal handler for the signal s (the name of the signal). The old handler (if any) is returned. If proc is null, the signal is reset to its default value. Procedure proc will be called with a single parameter, which is the string name of the signal received. Unicon knows about 40 names; most folks will care mainly about "SIGINT" and "SIGPIPE".

Caveat: This is not supported by the optimizing compiler (the -C command line option, which invokes iconc).

---

trim(s, c:' ', i:-1) : string **trim string**

trim(s,c,i) removes characters in c from s at the back (i=-1, the default), at the front (i=1), or at both ends (i=0).

---

truncate(f, i) : ? **truncate file**

truncate(f, len) changes the file f (which may be a string filename, or an open file) to be no longer than length len. truncate() does not work on windows, network connections, pipes, or databases.

---

trylock(x) : x? **try locking mutex**

trylock(x) attempts to lock the mutex x or the mutex associated with thread-safe object x. trylock fails if x is locked by a different thread or co-expression. If x is already locked by the calling thread or co-expression, trylock will lock it again.

---

type(x) : string **type of value**

type(x) returns a string that indicates the type of x.

---

unlock(x) : x **unlock mutex**

unlock(x) unlocks the mutex x or the mutex associated with thread-safe object x.

---

upto(c, s, i, i) : integer* **find characters in set**

String scanning function upto(c,s,i1,i2) generates the sequence of integer positions in s up to a character in c in s[i1:i2], but fails if there is no such position.

---

utime(s, i, i) : null **file access/modification times**

utime(f, atime, mtime) sets the access time for a file named f to atime and the modification time to mtime. The ctime is set to the current time. The effects of this function are platform specific. Some file systems support only a subset of these times.

---

variable(s, CE:&current, i:0) : any? **get variable**

variable(s, c, i) finds the variable with name s and returns a variable descriptor that refers to its value. The name s is searched for within co-expression c, starting with local variables i levels above the current procedure frame, and then among the global variables in the program that created c.

---

wait(x) : ? **wait for thread or condition variable**

wait(x) waits for x. If x is a thread, wait() waits for it to finish. If x is is a condition variable wait() waits until that variable is subsequently signaled by another thread.

---

where(f) : integer? **file position**

where(f) returns the current offset position in file f. It fails on windows and networks. The beginning of the file is offset 1.

---

write(s|f, ...) : string|file **write text line**

write(args) outputs strings, followed by a newline, to a file or files. Strings are written in order to their nearest preceding file, defaulting to &output. A newline is output to the preceding file after the last argument, as well as whenever a non-initial file argument directs output to a different file. write() returns its last argument.

---

writes(s|f, ...) : string|file **write strings**

writes(args) outputs strings to one or more files. Each string argument is written to the nearest preceding file argument, defaulting to &output. writes() returns its last argument.

Graphics functions

The names of built-in graphics functions begin with upper case. The built-in graphics functions are listed here. These functions are more thoroughly described in [Griswold98]. Extensive procedure and class libraries for graphics are described in [Griswold98] and in Appendix B. In 2D, arguments named x and y are pixel locations in zero-based integer coordinates. In 3D windows coordinates are given using real numbers, and functions by default take three coordinates (x,y,z) per vertex. Attribute dim can be set to 2 or 4, changing most 3D functions to take vertices in a (x,y) or (x,y,z,w) format. Arguments named row and col are cursor locations in one-based integer text coordinates. Most functions' first

parameter named w defaults to &window and the window argument can be omitted in the default case. Most 3D functions are not thread-safe.

Active() : window **produce active window**

Active() returns a window that has one or more events pending. If no window has an event pending, Active() blocks and waits for an event to occur. Active() starts with a different window on each call in order to avoid window "starvation". Active() fails if no windows are open.

Alert() : window **alert the user**

Alert() produces a visual flash or audible beep that signifies to the user the occurrence of some notable event in the application.

Bg(w,s) : string **background color**

Bg(w) retrieves the background color. Bg(w,s) sets the background color by name, rgb, or mutable color value. Bg() fails if the background cannot be set to the requested color.

Clip(w,x:0,y:0,width:0,height:0) : window **clip to rectangle**

Clip(w,x,y,width,height) clips output to a rectangular area within the window. If width is 0, the clip region extends from x to the right side of the window. If height is 0, the clip region extends from y to the bottom of the window.

Clone(w,s,...) : window **clone context**

Clone(w) produces a new window binding in which a new graphics context is copied from w and bound to w's canvas. Additional string arguments specify attributes of the new binding, as in WAttrib(). If the first string argument is "g" or "gl", Clone() binds the new context to a subwindow with separate canvas and input queue inside of and relative to w. Clone() fails if an attribute cannot be set to a requested value.

Color(w, i, s,...) : window **set mutable color**

Color(w,i) produces the current setting of mutable color i. Color(w,i,s,...) sets the color map entries identified by i[j] to the corresponding colors s[j]. See [Griswold98].

ColorValue(w, s) : string **convert color name to rgb**

ColorValue(w,s) converts the string color s into a string with three comma-separated 16-bit integer values denoting the color's RGB components. ColorValue() fails if string s is not a valid name or recognized decimal or hex encoding of a color.

CopyArea(w1, w2,x:0,y:0,width:0,height:0,x2:0,y2:0) : window **copy area**

CopyArea(w1,w2,x,y,width,height,x2,y2) copies a rectangular region within w1 defined by x,y,width,height to window w2 at offset x2,y2. CopyArea() returns w1. &window is not a default for this function. The default copies all of w1.

Couple(w1, w2) : window **couple window to context**

Couple(w1,w2) produces a new value that binds the window associated with w1 to the graphics context associated with w2.

DrawArc(w, x, y, width, height:width, a1:0.0, a2:2*&pi, ...) : window draw arc

DrawArc(w,x,y,width,height,a1,a2,...) draws arcs or ellipses. Each is defined by six integer coordinates. x, y, width and height define a bounding rectangle around the arc; the center of the arc is the point (x+(width)/2,y+(height)/2). Angles are specified in radians. Angle a1 is the starting position of the arc, where 0.0 is the 3 o'clock position and the positive direction is counter-clockwise. Angle a2 is not the end position, but rather specifies the direction and extent of the arc.

DrawCircle(w, x, y, radius, a1:0.0, a2:2*&pi, ...) : window draw circle

DrawCircle() draws a circle or arc, centered at (x,y) and otherwise similar to DrawArc() with width=height.

DrawCube(w, x, y, z, len ...) : record draw cube

DrawCube(w, x, y, z, len...) draws a cube with sides of length len at the position (x, y, z) on the 3D window w. The display list element is returned. This procedure fails if the context attribute dim is set to 2.

DrawCurve(w, x1, y1, ...) : window draw curve

DrawCurve(w,x1,y1,...,xn,yn) draws a smooth curve connecting each x,y pair in the argument list. If the first and last point are the same, the curve is smooth and closed through that point. If there is no window argument, and &window is not set, DrawCurve() returns the points as alternating x, y values in a single list.

DrawCylinder(w, x, y, z, h, r1, r2, ...) : record draw cylinder

DrawCylinder(w, x, y, z, h, r1, r2, ...) draws a cylinder with a top of radius r1, a bottom with radius r2, and a height h on 3D window w. The disk is centered at the point (x, y, z). The display list element is returned. This procedure fails if the context attribute dim is set to 2.

DrawDisk(w, x, y, z, r1, r2, a1, a2, ...) : record draw disk

DrawDisk(W, x, y, z, r1, r2, a1, a2, ...) draws a disk or partial disk centered at (x, y, z) on 3D window w. The inner circle has radius r1 and the outer circle has radius r2. The parameters a1 and a2 are optional. If they are specified, a partial disk is drawn with a starting angle a1 and sweeping angle a2. The display list element is returned.

DrawImage(w, x, y, s) : window draw bitmapped figure

DrawImage(w,x,y, s) draws an image specified in string s at location x,y.

DrawLine(w, x1, y1, z1 ...) : window [list] draw line

DrawLine(w,x1,y1,...,xn,yn) draws lines between each adjacent x,y pair of arguments. In 3D, DrawLine() takes from 2-4 coordinates per vertex and returns the list that represents the lines on the display list for refresh purposes.

DrawPoint(w, x1, y1, ...) : window [list] draw point

DrawPoint(w,x1,y1,...,xn,yn) draws points. In 3D, DrawPoint() takes from 2-4 coordinates per vertex and returns the list that represents the points on the display list for refresh purposes.

DrawPolygon(w, x1, y1, [z1,] ...) : window [list] draw polygon

DrawPolygon(w,x1,y1,...,xn,yn) draws a polygon. In 3D, DrawPolygon() takes from 2-4 coordinates per vertex and returns the list that represents the polygon on the display list for refresh purposes.

DrawRectangle(w, x1, y1, width1, height1 ...) : window draw rectangle

DrawRectangle(w,x1,y1,width1,height1,...) draws rectangles. Arguments width and height define the perceived size of the rectangle; the actual rectangle drawn is width+1 pixels wide and height+1 pixels high.

DrawSegment(w, x1, y1, [z1,] ...) : window [list] draw line segment

DrawSegment(w,x1,y1,...,xn,yn) draws lines between alternating x,y pairs in the argument list. In 3D, DrawSegment() takes from 2-4 coordinates per vertex and returns the list that represents the segments on the display list for refresh purposes.

DrawSphere(w, x, y, z, r, ...) : record draw sphere

DrawSphere(w, x, y, z, r,...) draws a sphere with radius r centered at (x, y, z) on 3D window w. The display list element is returned. This procedure fails if the context attribute dim is set to 2.

DrawString(w, x1, y1, s1, ...) : window draw text

DrawString(w,x,y,s) draws text s at coordinates (x, y). This function does not draw any background; only the characters' actual pixels are drawn. It is possible to use "drawop=reverse" with this function to draw erasable text. DrawString() does not affect the text cursor position. Newlines present in s cause subsequent characters to be drawn starting at (x, current_y + leading), where x is the x supplied to the function, current_y is the y coordinate the newline would have been drawn on, and leading is the current leading associated with the binding.

DrawTorus(w, x, y, z, r1, r2, ...) : record draw torus

DrawTorus(w, x, y, z, r1, r2,...) draws a torus with inner radius r1, outside radius r2, and centered at (x,y,z) on 3D window w. The display list element is returned. This procedure fails if the context attribute dim is set to 2.

EraseArea(w, x:0, y:0, width:0, height:0. ...) : window erase rectangular area

EraseArea(w,x,y,width,height,...) erases rectangular areas within the window to the background color. If width is 0, the region cleared extends from x to the right side of the window. If height

is 0, the region erased extends from y to the bottom of the window. In 3D, EraseArea(W) clears the contents of the entire window.

Event(w, i:infinity) : string|integer **read event on window**

Event(w, i) retrieves the next event available for window w. If no events are available, Event() waits for i milliseconds. Keystrokes are encoded as strings, while mouse events are encoded as integers. The retrieval of an event is accompanied by assignments to the keywords &x, &y, &row, &col, &interval, &control, &shift, &meta, and if 3D attribute "pick=on", &pick. Event() fails if the timeout expires before an event occurs.

Fg(w, s) : string **foreground color**

Fg(w) retrieves the current foreground color. Fg(w,s) sets the foreground by name or value. Fg() fails if the foreground cannot be set to the requested color. In 3D, Fg(w, s) changes the material properties of subsequently drawn objects to the material properties specified by s. The string s must be one or more semi-colon separated material properties. A material property is of the form

[diffuse | ambient | specular | emission] *color name* or "shininess n", 0 <= n <= 128.

If string s is omitted, the current values of the material properties will be returned.

FillArc(w, x, y, width, height, a1, a2, ...) : window **draw filled arc**

FillArc(w,x,y,width,height,a1,a2,...) draws filled arcs, ellipses, and/or circles. Coordinates are as in DrawArc().

FillCircle(w, x, y, radius, a1, a2, ...) : window **draw filled circle**

FillCircle(w,x,y, radius,a1,a2,...) draws filled circles. Coordinates are as in DrawCircle().

FillPolygon(w, x1, y1, [z1,] ...) : window **draw filled polygon**

FillPolygon(w,x1,y1,...,xn,yn) draws a filled polygon. The beginning and ending points are connected if they are not the same. In 3D, FillPolygon() takes from 2-4 coordinates per vertex and returns the list that represents the polygon on the display list for refresh purposes.

FillRectangle(w, x:0, y:0, width:0, height:0, ...) : window **draw filled rectangle**

FillRectangle(w,x,y,width,height,...) draws filled rectangles.

Font(w, s) : string **font**

Font(w) produces the name of the current font. Font(w,s) sets the window context's font to s and produces its name or fails if the font name is invalid. The valid font names are largely system-dependent but follow the format family[,styles],size, where styles optionally add bold or italic or both. Four font names are portable: serif (Times or similar), sans (Helvetica or similar), mono (a mono spaced sans serif font) and typewriter (Courier or similar). Font() fails if the requested font name does not exist.

FreeColor(w, s, ...) : window **release colors**

FreeColor(w,s1,...,sn) allows the window system to re-use the corresponding color map entries. Whether this call has an effect is dependent upon the particular implementation. If a freed color is still in use at the time it is freed, unpredictable results will occur.

GotoRC(w, row:1, col:1) : window **go to row,column**

GotoRC(w,row,col) moves the text cursor to a particular row and column, given in numbers of characters; the upper-left corner is coordinate (1,1). The column calculation used by GotoRC() assigns to each column the pixel width of the widest character in the current font. If the current font is of fixed width, this yields the usual interpretation.

GotoXY(w, x:0, y:0) : window **go to pixel**

GotoXY(w,x,y) moves the text cursor to a specific cursor location in pixels.

IdentityMatrix(w) : record **load the identity matrix**

IdentityMatrix(w) changes the current matrix to the identity matrix on 3D window w. The display list element is returned.

Lower(w) : window **lower window**

Lower(w) moves window w to the bottom of the window stack.

MatrixMode(w, s) : record **set matrix mode**

MatrixMode(w, s) changes the matrix mode to s on 3D window w. The string s must be either "projection" or "modelview"; otherwise this procedure fails. The display list element is returned.

MultMatrix(w, L) : record **multiply transformation matrix**

MultMatrix(w, L) multiplies the current transformation matrix used in 3D window w by the 4x4 matrix represented as a list of 16 values L.

NewColor(w, s) : integer **allocate mutable color**

NewColor(w,s) allocates an entry in the color map and returns a small negative integer for this entry, usable in calls to routines that take a color specification, such as Fg(). If s is specified, the entry is initialized to the given color. NewColor() fails if it cannot allocate an entry.

PaletteChars(w, s) : string **pallete characters**

PaletteChars(w,s) produces a string containing each of the letters in palette s. The palletes "c1" through "c6" define different color encodings of images represented as string data; see [Griswold98].

PaletteColor(w, p, s) : string **pallete color**

PaletteColor(w,s) returns the color of key s in palette p in "*r*,*g*,*b*" format.

PaletteKey(w, p, s) : integer **pallete key**

PaletteKey(w,s) returns the key of closest color to s in palette p.

Pattern(w, s) : w **define stipple pattern**

Pattern(w,s) selects stipple pattern s for use during draw and fill operations. s may be either the name of a system-dependent pattern or a literal of the form *width,bits*. Patterns are only used when the fillstyle attribute is stippled or opaquestippled. Pattern() fails if a named pattern is not defined. An error occurs if Pattern() is given a malformed literal.

Pending(w, x, ...) : L **produce event queue**

Pending(w) produces the list of events waiting to be read from window w. If no events are available, the list is empty (its size is 0). Pending(w,x1,...,xn) adds x1 through xn to the end of w's pending list in guaranteed consecutive order.

Pixel(w, x:0, y:0, width:0, height:0) : i1...in **generate window pixels**

Pixel(w,x,y,width,height) generates pixel contents from a rectangular area within window w. width * height results are generated starting from the upper-left corner and advancing down to the bottom of each column before the next one is visited. Pixels are returned in integer values; ordinary colors are encoded nonnegative integers, while mutable colors are negative integers that were previously returned by NewColor(). Ordinary colors are encoded with the most significant eight bits all zero, the next eight bits contain the red component, the next eight bits the green component, and the least significant eight bits contain the blue component. Pixel() fails if part of the requested rectangle extends beyond the canvas.

PopMatrix(w) : record **pop the matrix stack**

PopMatrix(w) pops the top matrix from either the projection or modelview matrix stack on 3D window w, depending on the current matrix mode. This procedure fails if there is only one matrix on the matrix stack. The display list element is returned.

PushMatrix(w) : record **push the matrix stack**

PushMatrix(w) pushes a copy of the current matrix onto the matrix stack on 3D window w. The current matrix mode determines on what stack is pushed. This procedure fails if the stack is full. The "projection" stack is of size two; the "modelview" stack is of size thirty two. The display list element is returned.

PushRotate(w, a, x, y, z) : record **push and rotate**

PushRotate() is equivalent to PushMatrix() followed by Rotate().

PushScale(w, x, y, z) : record **push and scale**

PushScale() is equivalent to PushMatrix() followed by Scale().

PushTranslate(w, x, y, z) : record **push and translate**

PushTranslate() is equivalent to PushMatrix() followed by Translate().

QueryPointer(w) : x, y **produce mouse position**

QueryPointer(w) generates the x and y coordinates of the mouse relative to window w. If w is omitted, QueryPointer() generates the coordinates relative to the upper-left corner of the entire screen.

Raise(w) : window **raise window**

Raise(w) moves window w to the top of the window stack, making it entirely visible and possibly obscuring other windows.

ReadImage(w, s, x:0, y:0) : integer **load image file**

ReadImage(w,s,x,y) loads an image from the file named by s into window (or texture) w at offset x,y. x and y are optional and default to 0,0. GIF, JPG, PNG, and BMP formats are supported, along with platform-specific formats. If ReadImage() reads the image into w, it returns either an integer 0 indicating no errors occurred or a nonzero integer indicating that one or more colors required by the image could not be obtained from the window system. ReadImage() fails if file s cannot be opened for reading or is an invalid file format.

Refresh(w) : window **redraw the window**

Refresh(w) redraws the contents of window w. It is used mainly when objects have been moved in a 3D scene. The window w is returned.

Rotate(w, a, x, y, z) : record **rotate objects**

Rotate(w, a, x, y, z,...) rotates subsequent objects drawn on 3D window w by angle a degrees, in the direction (x,y,z). The display list element is returned.

Scale(w, x, y, z) : record **scale objects**

Scale(w, x, y, z,...) scales subsequent objects drawn on 3D window w according to the given coordinates. The display list element is returned.

Texcoord(w, x, y, ...) : list **define texture coordinates**

Texcoord(W, x_1,y_1, ..., x_n, y_n) sets the texture coordinates to x_1, y_1, ..., x_n, y_n in 3D window w. Each x, y, pair forms one texture coordinate. Texcoord(W, L) sets the texture coordinates to those specified in the list L. Texcoord(W, s) sets the texture coordinates to those specified by s. The string s must be "auto" otherwise the procedure will fail. In all cases the display list element is returned.

TextWidth(w, s) : integer **pixel width of text**

TextWidth(w,s) computes the pixel width of string s in the font currently defined for window w.

Texture(w, s) : record **apply texture**

Texture(w, s) specifies a texture image that is applied to subsequent objects drawn on 3D window w. The string s specifies the texture image as a filename, a string of the form

width,pallet,data or width,#,data, where pallet is a pallet from the Unicon 2D graphics facilities and data is the hexadecimal representation of an image. Texture(w1, w2) specifies that the contents of 2D or 3D window w2 be used as a texture image that is applied to subsequent objects on the window w1. The display list element is returned.

Translate(w, x, y, z, ...) : record **translate object positions**

Translate(w, x, y, z,...) moves objects drawn subsequently on 3D window w in the direction (x,y,z). The display list element is returned.

Uncouple(w) : window **release binding**

Uncouple(w) releases the binding associated with file w. Uncouple() closes the window only if all other bindings associated with that window are also closed.

WAttrib(w, s1, ...) : x, ... **generate or set attributes**

WAttrib(w, s1, ...) retrieves and/or sets window and context attributes. If called with exactly one attribute, integers are produced for integer-value attributes; all other values are represented by strings. If called with more than one attribute argument, WAttrib() produces one string result per argument, prefixing each value by the attribute name and an equals sign (=). If xi is a window, subsequent attributes apply to xi. WAttrib() fails if an attempt is made to set the attribute font, fg, bg, or pattern to a value that is not supported. A run-time error occurs for an invalid attribute name or invalid value.

WDefault(w, program, option) : string **query user preference**

WDefault(w,program,option) returns the value of option for program as registered with the X resource manager. In typical use this supplies the program with a default value for window attribute option from a program.option entry in an .XDefaults file. WDefault() fails if no user preference for the specified option is available.

WFlush(w) : window **flush window output**

WFlush(w) flushes window output on window systems that buffer text and graphics output. Window output is automatically flushed whenever the program blocks on window input. When this behavior is not adequate, a call to WFlush() sends all window output immediately, but does not wait for all commands to be received and acted upon. WFlush() is a no-op on window systems that do not buffer output.

WindowContents(w) : list **window display list**

WindowContents(w) returns a list of display elements, which are records or lists. Each element has a function name followed by the parameters of the function, or an attribute followed by its value.

WriteImage(w, s, x:0, y:0, width:0, height:0) : window **save image file**

WriteImage(w,s,x,y,width,height) saves an image of dimensions width, height from window w at offset x,y to a file named s. The default is to write the entire window. The file is written

according to the filename extension, in either GIF, JPG, BMP, PNG, or a platform specific format such as XBM or XPM. WriteImage() fails if s cannot be opened for writing.

WSection(w, s) : record **define window section**

WSection(w,s) starts a new window section named s on 3D window w and returns a display list section marker record. During window refreshes if the section marker's skip field is 1, the section is skipped. The section name s is produced by &pick if a primitive in the block is clicked on while attribute "pick=on". WSection(w) marks the end of a preceding section. WSection() blocks may be nested.

WSync(w, s) : w **synchronize with window system server**

WSync(w,s) synchronizes the program with the server attached to window w on those window systems that employ a client-server model. Output to the window is flushed, and WSync() waits for a reply from the server indicating all output has been processed. If s is "yes", all events pending on w are discarded. WSync() is a no-op on window systems that do not use a client-server model.

Pattern functions

Abort() **pattern cancel**

Abort() causes an immediate failure of the entire pattern match.

Any(c) **match any**

Any(c) matches any single character contained in c appearing in the subject string.

Arb() **arbitrary pattern**

Arb() matches zero or more characters in the subject string.

Arbno(p) **repetitive arbitrary pattern**

Arbno(p) matches repetitive sequences of p in the subject string.

Bal() **balanced parentheses**

Bal() matches the shortest non-null string which parentheses are balanced in the subject string.

Break(c) **pattern break**

Break(c) matches any characters in the subject string up to but not including any of the characters in cset c.

Breakx(c) **extended pattern break**

Breakx(c) matches any characters up to any of the subject characters in c, and will search beyond the break position for a possible larger match.

Fail() **pattern back**

Fail() signals a failure in the current portion of the pattern match and sends an instruction to go back and try a different alternative.

Fence() **pattern fence**

Fence() signals a failure in the current portion of the pattern match if it is trying to backing up to try other alternatives.

Len(i) **match fixed-length string**

Len(i) matches a string of a length of i characters in the subject string. It fails if i is greater than the number of characters remaining in the subject string.

NotAny(c) **match anything but**

NotAny(c) matches any single character not contained in character set c appearing in the subject string.

Nspan(c) **optional pattern span**

Nspan() matches the longest available sequence of zero or more characters from the subject string that are contained in c.

Pos(i) **cursor position**

Pos(i) sets the cursor or index position of the subject string to the position i according the Unicon index system shown below:

```
-6  -5  -4  -3  -2  -1  0
| U | n | i | c | o | n |
1   2   3   4   5   6   7
```

Rem() **remainder pattern**

Rem() matches the remainder of the subject string.

Span(c) **pattern span**

Span(c) matches one or more characters from the subject string that are contained in c. It must match at least one character.

Succeed() **pattern succeeds**

Succeed() produces a pattern that, when matched, will cause the surrounding pattern match to succeed without further scrutiny.

Tab(n) **pattern tab**

Tab(n) matches any characters from the current cursor or index position up to the specified position of the subject string. Tab() uses the Unicon index system shown in Pos() and position n must be to the right of the current position.

Rpos(n) **reverse cursor position**

Rpos(n) sets the cursor or index position of the subject string to the position n back from the end of the string, equivalent to using Unicon's negative indices in Pos().

```
6 5 4 3 2 1 0
|S|N|O|B|O|L|
```

Rtab(i) **pattern reverse tab**

Rtab(i) matches any characters from the current cursor or index position up to the specified position (back from the end) of the subject string, equivalent to using a negative index in Tab(). Position n must be to the right of the current position.

A.7 Preprocessor

Unicon features a simple preprocessor that supports file inclusion and symbolic constants. It is a subset of the capabilities found in the C preprocessor, and is used primarily to support platform-specific code sections and large collections of symbols.

Preprocessor commands

Preprocessor directives are lines beginning with a dollar sign. The available preprocessor commands are:

$define symbol text **symbolic substitution**

All subsequent occurrences of *symbol* are replaced by the *text* within the current file. Note that $define does not support arguments, unlike C.

$include filename **insert source file**

The named file is inserted into the compilation in place of the $include line.

$ifdef symbol **conditional compilation**
$ifndef *symbol* **conditional compilation**
$else **conditional alternative**
$endif **end of conditional code**

The subsequent lines of code, up to an $else or $endif, are discarded unless *symbol* is defined by some $define directive. $ifndef reverses this logic.

$error text **compile error**

The compiler will emit an error with the supplied text as a message.

$line *number* [*filename*] **source code line** #line *number* [*filename*] **source code line**

The subsequent lines of code are treated by the compiler as commencing from line *number* in the file *filename* or the current file if no filename is given.

$undef symbol **remove symbol definition**

Subsequent occurrences of *symbol* are no longer replaced by any substitute text.

EBCDIC transliterations **alternative bracket characters**

These character combinations were introduced for legacy keyboards that were missing certain bracket characters.

```
$( for {
$) for }
$< for [
$> for ]
```

These character combinations are substitutes for curly and square brackets on keyboards that do not have these characters.

Predefined symbols

Predefined symbols are provided for each platform and each feature that is optionally compiled in on some platforms. These symbols include:

Preprocessor Symbol	Feature
_V9	Version 9
_CMS	CMS
_MSDOS_386	MS-DOS/386
_MS_WINDOWS_NT	MS Windows NT
_MSDOS	MS-DOS
_MVS	MVS
_OS2	OS/2
_PORT	PORT
_UNIX	UNIX
_POSIX	POSIX
_DBM	DBM
_VMS	VMS
_ASCII	ASCII
_EBCDIC	EBCDIC
_CO_EXPRESSIONS	co-expressions
_CONSOLE_WINDOW	console window
_DYNAMIC_LOADING	dynamic loading
_EVENT_MONITOR	event monitoring
_EXTERNAL_FUNCTIONS	external functions
_KEYBOARD_FUNCTIONS	keyboard functions
_LARGE_INTEGERS	large integers

_MULTITASKING multiple programs
_PIPES pipes
_RECORD_IO record I/O
_SYSTEM_FUNCTION system function
_MESSAGING messaging
_GRAPHICS graphics
_X_WINDOW_SYSTEM X Windows
_MS_WINDOWS MS Windows
_WIN32 Win32
_PRESENTATION_MGR Presentation Manager
_DOS_FUNCTIONS MS-DOS extensions
_DEVMODE developer mode

A.8 Execution Errors

There are two kinds of errors that can occur during the execution of an Icon program: runtime errors and system errors. Runtime errors occur when a semantic or logic error in a program results in a computation that cannot perform as instructed. System errors occur when an operating system call fails to perform a required service.

Runtime errors

By default, a runtime error causes program execution to abort. Runtime errors are reported by name as well as by number. They are accompanied by an error traceback that shows the procedure call stack and value that caused the error, if there is one. The errors are listed below to illustrate the kinds of situations that can cause execution to terminate.

The keyword &error turns runtime errors into expression failure. When an expression fails due to a converted runtime error, the keywords &errornumber, &errortext, and &errorvalue provide information about the nature of the error. When a system function fails, keyword &errno and &errortext are set to indicate the nature of the system call failure; the numbering systems of &errornumber and &errno are unrelated; &errno numbers are platform dependent.

101 integer expected or out of range
102 numeric expected
103 string expected
104 cset expected
105 file expected
106 procedure or integer expected
107 record expected
108 list expected
109 string or file expected

110 string or list expected
111 variable expected
112 invalid type to size operation
113 invalid type to random operation
114 invalid type to subscript operation
115 structure expected
116 invalid type to element generator
117 missing main procedure
118 co-expression expected
119 set expected
120 two csets, two sets, or two tables expected
121 function not supported
122 set or table expected
123 invalid type
124 table expected
125 list, record, or set expected
126 list or record expected
127 pattern expected
128 unevaluated variable or function call expected
129 unable to convert unevaluated variable to pattern
130 incorrect number of arguments
131 string is not a class name
140 window expected
141 program terminated by window manager
142 attempt to read/write on closed window
143 malformed event queue
144 window system error
145 bad window attribute
146 incorrect number of arguments to drawing function
147 window attribute cannot be read or written as requested
148 graphics is not enabled in this virtual machine
150 drawing a 3D object while in 2D mode
151 pushed/popped too many matrices
152 modelview or projection expected
153 texture not in correct format
154 must have an even number of texture coordinates
155 3D graphics is not enabled in this virtual machine
160 nonexistent variable name
161 cannot convert unevaluated variable to pattern
162 uninitialized pattern

163 object, method, or method parameter problem in unevaluated expression
164 unsupported unevaluated expression
165 null pattern argument where name was expected
166 unable to produce pattern image, possible malformed pattern
170 string or integer expected
171 UDP socket expected
172 signal handler procedure must take one argument
173 cannot open directory for writing
174 invalid file operation
175 network connection expected
180 invalid mutex
181 invalid condition variable
182 illegal recursion in initial clause
183 concurrent threads are not enabled in this virtual machine
184 structure cannot have more than one mutex at the same time
185 converting an active co-expression to a thread is not yet supported
190 dbm database expected
191 cannot open dbm database
201 division by zero
202 remaindering by zero
203 integer overflow
204 real overflow, underflow, or division by zero
205 invalid value
206 negative first argument to real exponentiation
207 invalid field name
208 second and third arguments to map of unequal length
209 invalid second argument to open
210 non-ascending arguments to detab/entab
211 by value equal to zero
212 attempt to read file not open for reading
213 attempt to write file not open for writing
214 input/output error
215 attempt to refresh &main
216 external function not found
217 unsafe inter-program variable assignment
218 invalid file name
301 evaluation stack overflow
302 memory violation
303 inadequate space for evaluation stack
304 inadequate space in qualifier list

305 inadequate space for static allocation
306 inadequate space in string region
307 inadequate space in block region
308 system stack overflow in co-expression
309 pattern stack overflow
316 interpreter stack too large
318 co-expression stack too large
401 co-expressions not implemented
402 program not compiled with debugging option
500 program malfunction
600 vidget usage error
1040 socket error
1041 cannot initialize network library
1042 fdup of closed file
1043 invalid signal
1044 invalid operation to flock/fcntl
1045 invalid mode string
1046 invalid permission string for umask
1047 invalid protocol name
1048 low-level read or select mixed with buffered read
1049 nonexistent service or services database error
1050 command not found
1051 cannot create temporary file
1052 cannot create pipe
1053 empty pipe
1100 ODBC connection expected
1200 system error (see &errno)
1201 malformed URL
1202 missing username in URL
1203 unknown scheme in URL
1204 cannot parse URL
1205 cannot connect
1206 unknown host
1207 invalid field in header
1208 messaging file expected
1209 cannot determine smtpserver
1210 cannot determine user return address
1211 invalid email address
1212 server error
1213 POP messaging file expected

1214 cannot find certificate store
1215 cannot verify peer's certificate
1300 SSL error
1301 SSL context error
1302 bad ssl attribute
1303 private key error
1304 certificate error
1305 certificate authority error
1306 cipher error
1307 private key and certificate mismatch
1308 unknown protocol

System errors

If an error occurs during the execution of a system function, the program terminates. Unlike runtime errors, there is no way to convert the error to a failure (and continue execution).

The complete set of system errors is by definition platform specific. Error numbers above the value 1000 are used for system errors. Many of the POSIX standard system errors are supported across platforms, and error numbers between 1001 and 1040 are reserved for the system errors listed below. Platforms may report other system error codes so long as they do not conflict with existing runtime or system error codes.

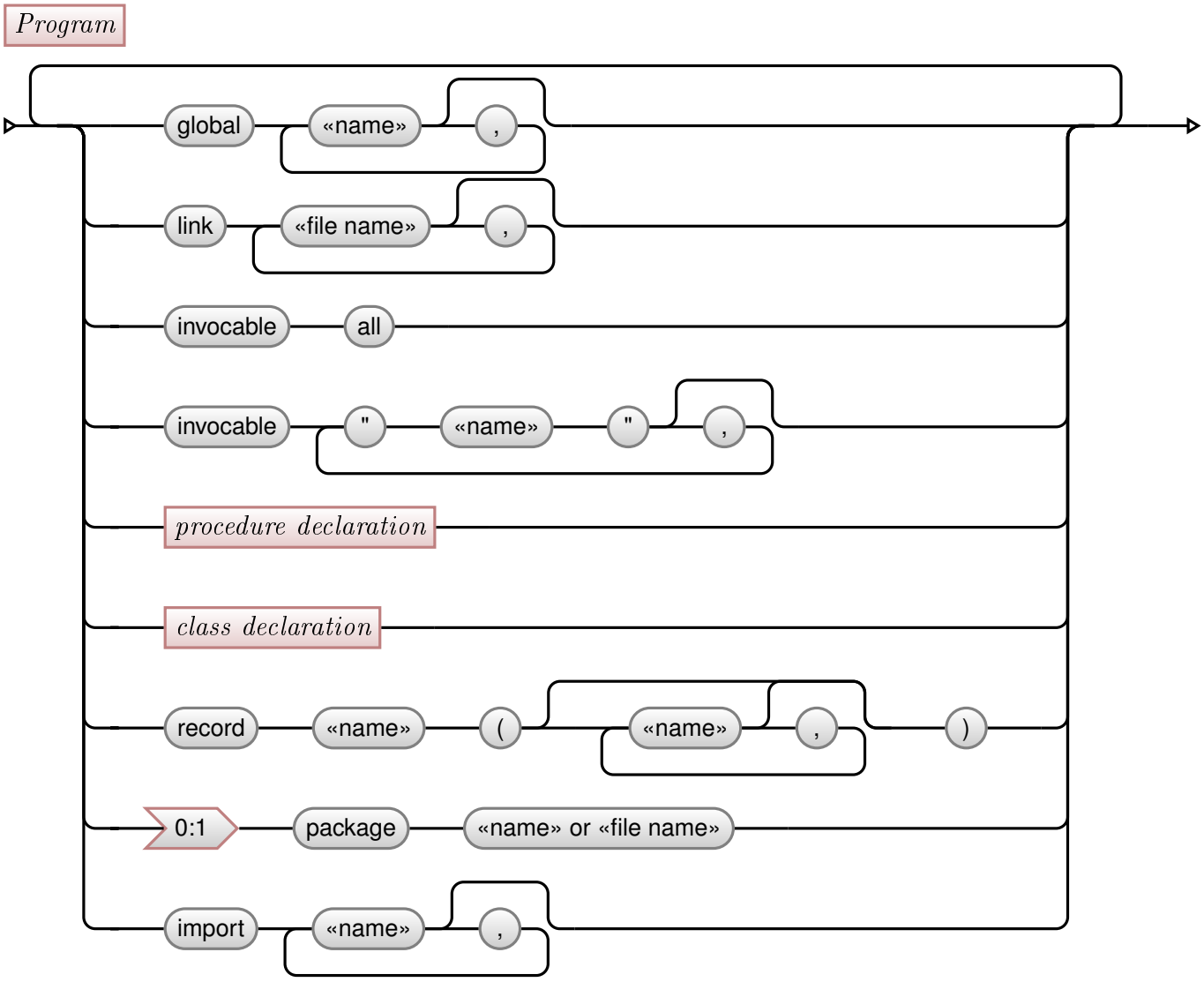
1001 Operation not permitted
1002 No such file or directory
1003 No such process
1004 Interrupted system call
1005 I/O error
1006 No such device or address
1007 Arg list too long
1008 Exec format error
1009 Bad file number
1010 No child processes
1011 Try again
1012 Out of memory
1013 Permission denied
1014 Bad address
1016 Device or resource busy
1017 File exists
1018 Cross-device link
1019 No such device
1020 Not a directory

1021	Is a directory
1022	Invalid argument
1023	File table overflow
1024	Too many open files
1025	Not a typewriter
1027	File too large
1028	No space left on device
1029	Illegal seek
1030	Read-only file system
1031	Too many links
1032	Broken pipe
1033	Math argument out of domain of func
1034	Math result not representable
1035	Resource deadlock would occur
1036	File name too long
1037	No record locks available
1038	Function not implemented
1039	Directory not empty
1040	socket error
1041	cannot initialize network library
1042	fdup of closed file
1043	invalid signal
1044	invalid operation to flock/fcntl
1045	invalid mode string
1046	invalid permission string for umask
1047	invalid protocol name
1048	low-level read or select mixed with buffered read
1100	ODBC connection expected
1200	system error (see &errno)
1201	malformed URL
1202	missing username in URL
1203	unknown scheme in URL
1204	cannot parse URL
1205	cannot connect
1206	unknown host
1207	invalid field in header
1208	messaging file expected
1209	cannot determine smtpserver (set UNICON_SMTPSERVER)
1210	cannot determine user return address (set UNICON_USERADDRESS)
1211	invalid email address

1212	server error
1213	POP messaging file expected

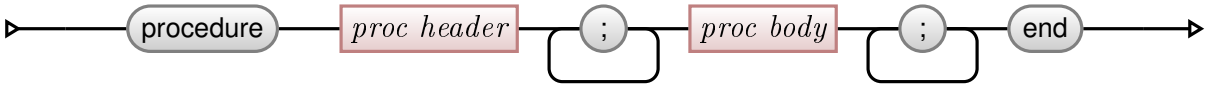
A.9 Syntax

The Unicon syntax is described here using "Railroad diagrams". Note that a newline may be substituted for the semicolon symbol (i.e. ;) wherever it appears in the diagrams. Some non-terminal symbols, which could be further expanded using another diagram, are represented by a single terminal symbol with the name enclosed in *guillemets*, for example «infix-operator».

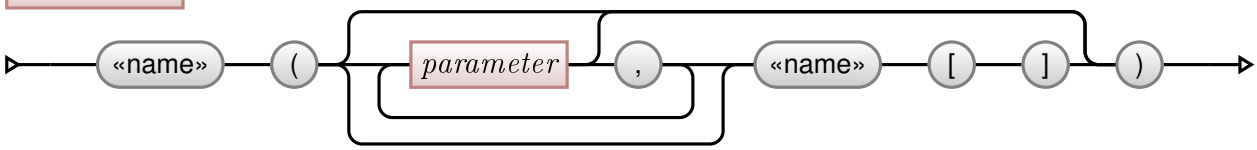


The n:m symbol is an extension to the usual railway syntax and denotes that a path may be taken between n and m times. 1 is an abbreviation for 1:1 meaning the path is compulsory and must appear exactly once.

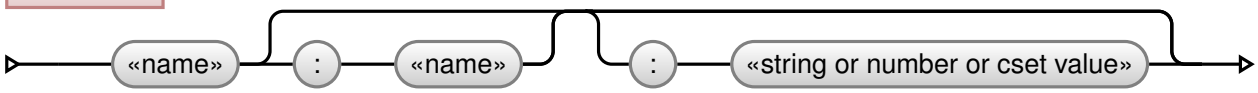
procedure declararation



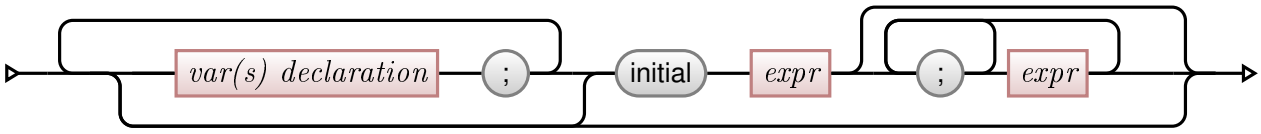
proc header



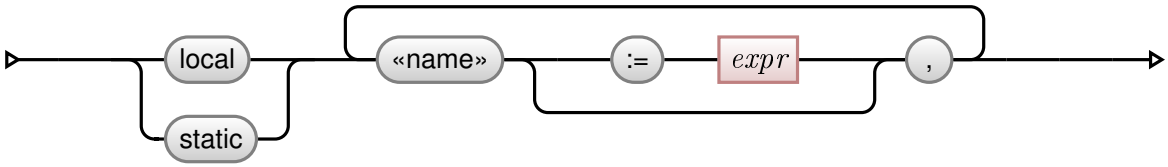
parameter



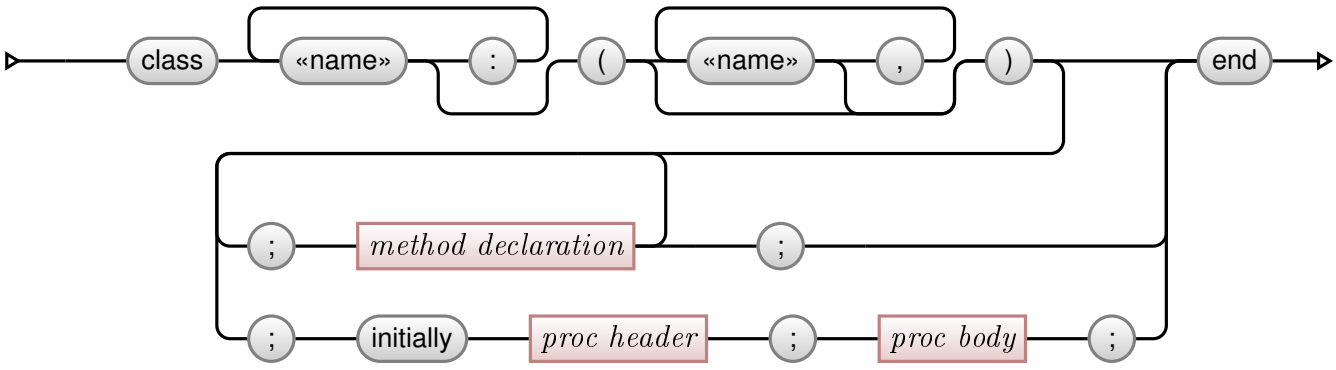
proc body



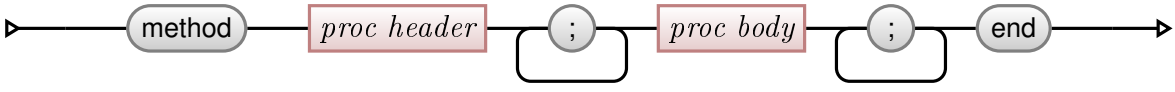
var(s) declaration

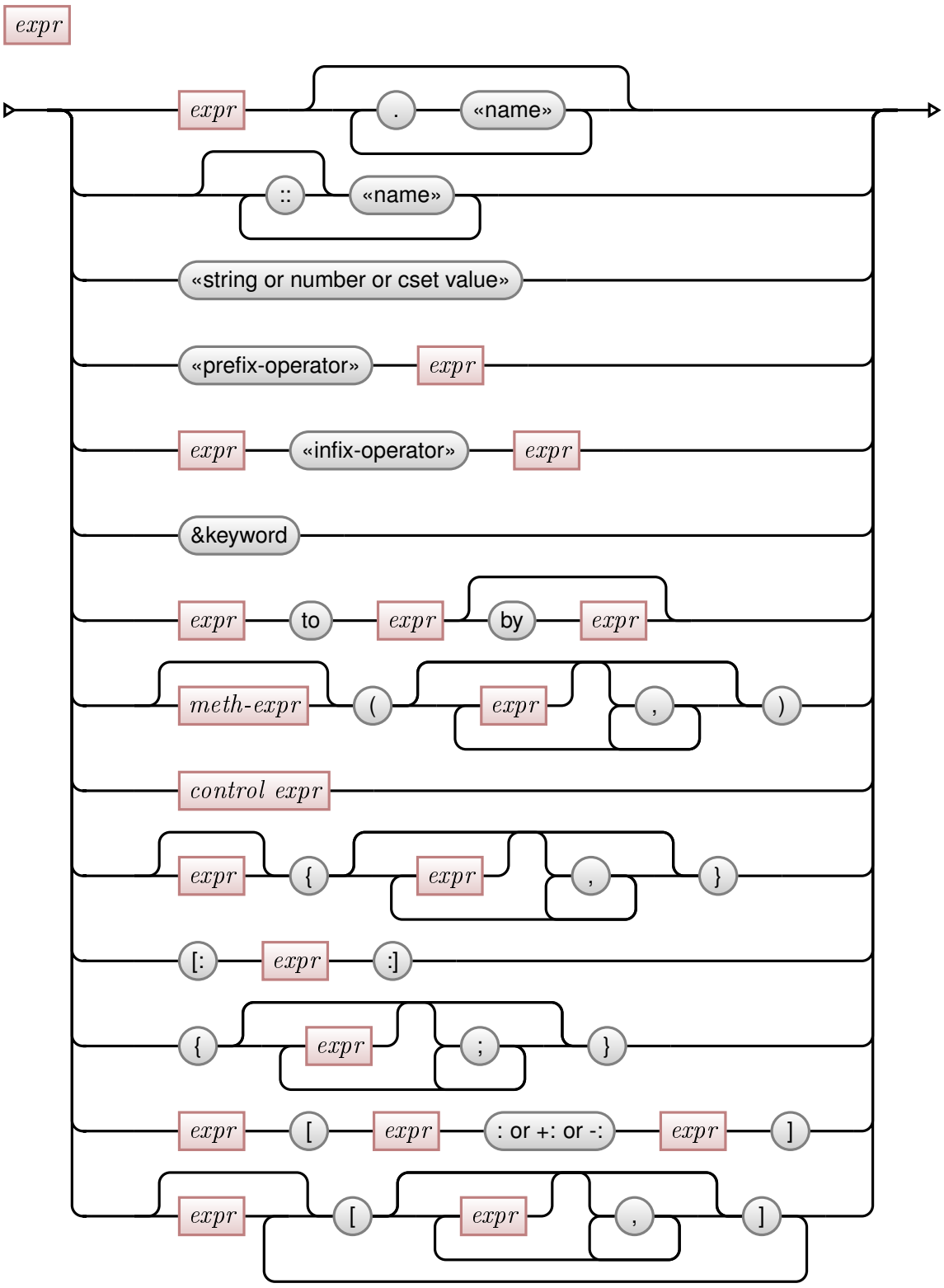


class declararation

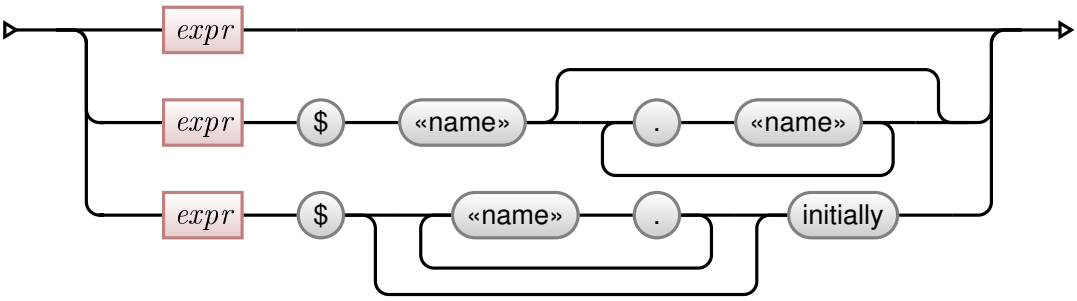


method declararation

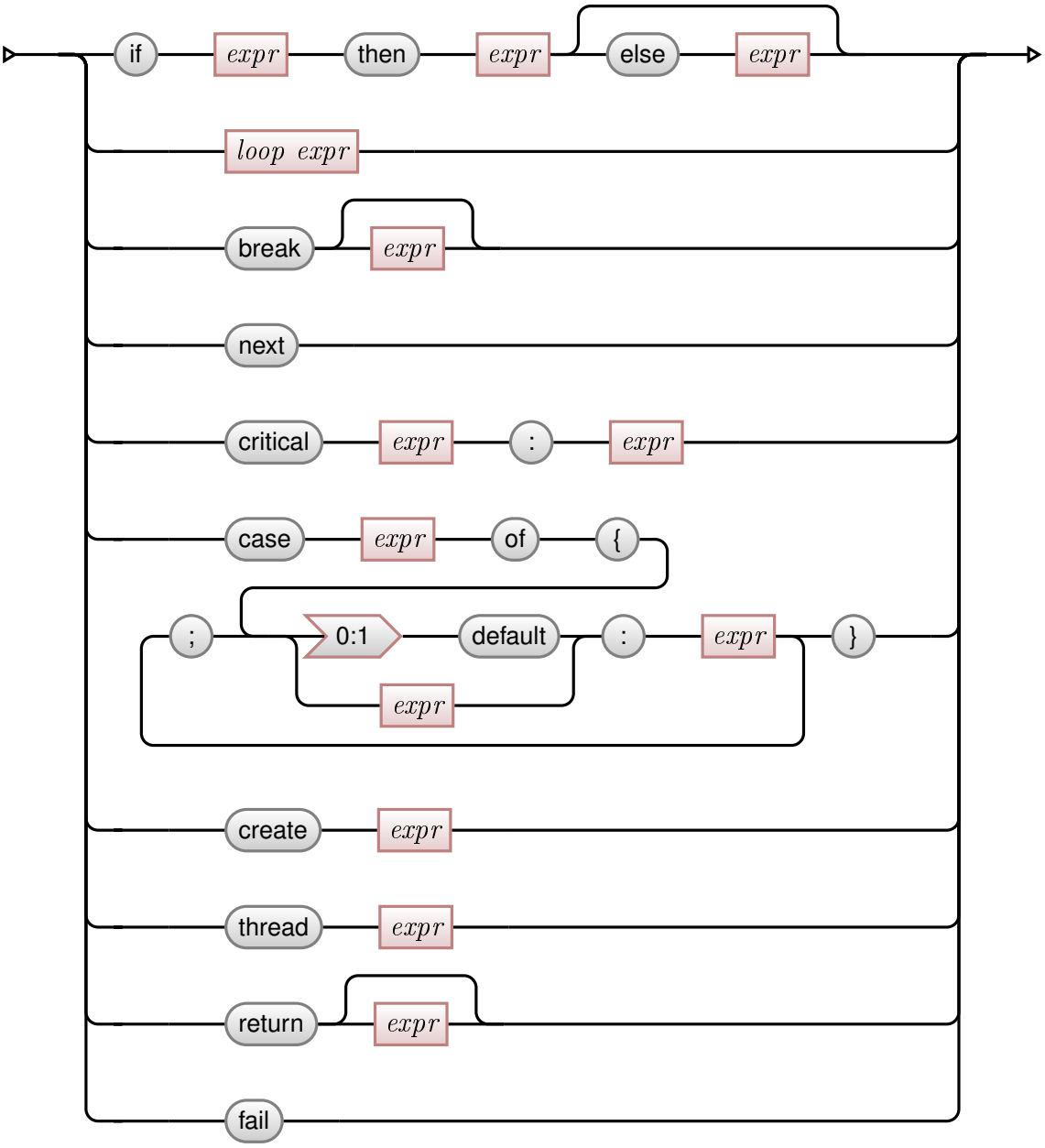


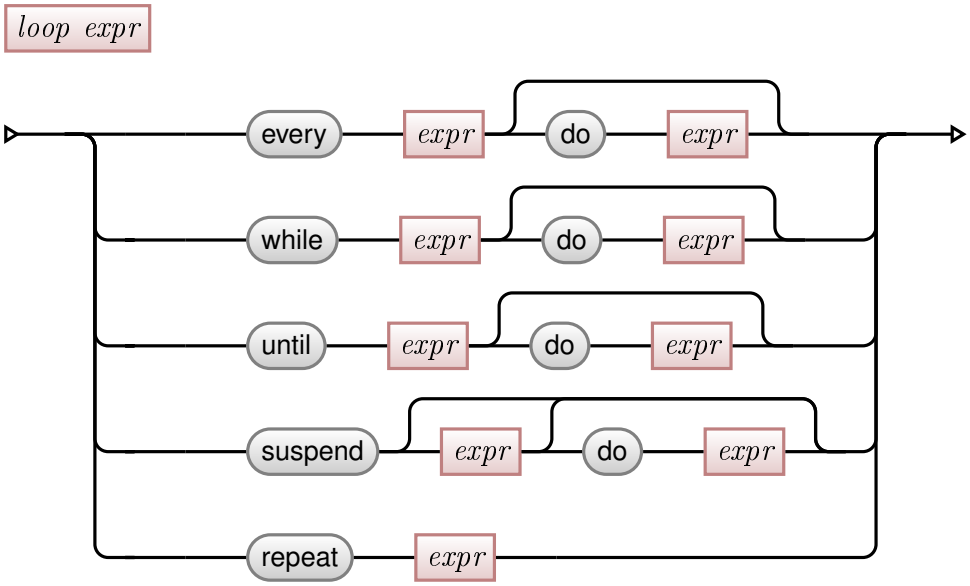

expr
expr
.
«name»
::
«name»
«string or number or cset value»
«prefix-operator»
expr
expr
«infix-operator»
expr
&keyword
expr
to
expr
by
expr
meth-expr
(
expr
,
)
control expr
expr
{
expr
,
}
[:
expr
:]
{
expr
;
}
expr
[
expr
: or +: or -:
expr
]
expr
[
expr
,
]

meth-expr



control expr





Operator precedence and associativity

Except where mentioned, infix operators are left–associative. The operators are listed in two columns and grouped from highest precedence to lowest: the groups are separated by horizontal lines and all operators in a group have the same precedence (i.e. the order within a group, or the column an operator is placed in, is not significant).

Operator Precedence

(*expr*)	[*expr1*, *expr2* ...]
expr (*expr1*, *expr2* ...)	*expr1* [*expr2*, *expr3* ...]
{ *expr1*; *expr2* ...}	*expr1* [*expr2* : *expr3*]
expr { *expr1*; *expr2* ...}	*expr1* [*expr2* +: *expr3*]
expr.*f*	*expr1* [*expr2* -: *expr3*]
*expr1*$initially(*expr2* ...)	*expr1*$*id*(*expr2* ...)
expr1$*id*.initially(*expr2* ...)	*expr1*$*id1*.*id2*(*expr2* ...)
[: *expr* :]	
< *regexp* > (regular expressions have different precedence rules)	
@<	@>
@<<	@>>
⇑ higher lower ⇓	
not *expr*	| *expr*

continued . . .

Operator Precedence (continued)

! *expr*	* *expr*
+ *expr*	- *expr*
/ *expr*	\ *expr*
. *expr*	++ *expr*
? *expr*	~ *expr*
^ *expr*	@ *expr*
@> *expr*()	@>> *expr*
@< *expr*	@<< *expr*
|| *expr*	||| *expr*
-- *expr*	** *expr*
= *expr*	~= *expr*
== *expr*	~== *expr*
=== *expr*	~=== *expr*
.| *expr*	

⇑ higher lower ⇓

expr1 \ *expr2*	
expr1 @ *expr2*	*expr1* ! *expr2*
expr1 @> *expr2*	*expr1* @>> *expr2*
expr1 @< *expr2*	*expr1* @<< *expr2*

⇑ higher lower ⇓

expr @>	*expr* @>>
expr @<	*expr* @<<

⇑ higher lower ⇓

expr1 ^ *expr2* (right–associative)

⇑ higher lower ⇓

expr1 * *expr2*	*expr1* / *expr2*
expr1 ** *expr2*	*expr1* % *expr2*

⇑ higher lower ⇓

expr1 + *expr2*	*expr1* - *expr2*
expr1 ++ *expr2*	*expr1* -- *expr2*
expr1 $$ *expr2*	*expr1* -> *expr2*

⇑ higher lower ⇓

expr1 || *expr2*	*expr1* ||| *expr2*

⇑ higher lower ⇓

expr1 < *expr2*	*expr1* <= *expr2*
expr1 > *expr2*	*expr1* >= *expr2*
expr1 = *expr2*	*expr1* ~= *expr2*
expr1 == *expr2*	*expr1* ~== *expr2*

continued . . .

Operator Precedence (continued)

expr1 === *expr2* — *expr1* ~=== *expr2*
expr1 << *expr2* — *expr1* <<= *expr2*
expr1 >> *expr2* — *expr1* >>= *expr2*

⇑ higher lower ⇓

expr1 | *expr2* — *expr1* && *expr2*

⇑ higher lower ⇓

expr1 .| *expr2*
expr1 to *expr2* — *expr1* to *expr2* by *expr3*

⇑ higher lower ⇓

expr1 ?? *expr2*

⇑ higher lower ⇓

expr1 := *expr2* (right–associative)
expr1 <- *expr2* (right–associative)
expr1 :=: *expr2* (right–associative)
expr1 <=> *expr2* (right–associative)
expr1 op:= *expr2* (operators that may be augmented are listed below)

⇑ higher lower ⇓

expr1 ? *expr2*

⇑ higher lower ⇓

expr1 & *expr2*

⇑ higher lower ⇓

break *expr1*
case *expr* of {*expr1*: *expr2*; *expr3*: *expr4*; ...}
create *expr*
every *expr1* do *expr2*
fail
if *expr1* then *expr2* else *expr3*
next
repeat *expr*
return *expr*
suspend *expr1* do *expr2*
until *expr1* do *expr2*
while *expr1* do *expr2*
critical *expr1* : *expr2*
thread *expr*

⇑ higher lower ⇓

expr1 ; *expr2*

Augmented operators

The following operators may be augmented with :=

||	|||	+	-	*	++	--	**	/	%
=	==	===	~=	~==	~===	^	?	&	@
>>=	>>	<<=	<<	>=	>	<=	<		

Appendix B

The Icon Program Library

The Icon Program Library (IPL) contains hundreds of complete programs ranging from simple demonstrations to complex applications. More importantly, it provides hundreds of library modules with thousands of procedures that can save you a lot of effort. To use these procedures, add link declarations to the modules in which they are declared. On standard installations, Unicon knows the location of the directory IPL hierarchy relative to the Unicon binaries. To move these binaries or add your own directories of reusable code modules, you must set the IPATH (and often LPATH) environment variables so they know where to find the IPL directories, in addition to any of your own you wish to add. Both IPATH and LPATH are semicolon-separated lists of directories. The names IPATH and LPATH can be confusing; IPATH directories are searched for .u code modules to be linked, while LPATH directories are searched for .icn source code files that you wish to $include.

The Icon Program Library's 20 directories of code, data, and supporting documentation is included in Unicon distributions. This reference appendix presents only the most useful components of the IPL, emphasizing those of potential value to other programmers. Many modules were written by Ralph Griswold; if he was not the author of a given module, the initials of authors are included in the module description. A list of authors' names appears at the end of the appendix. This appendix no doubt overlooks some excellent library modules and programs. The entries in this appendix loosely follow this template:

---

filename

Each module has a general description, ending with the authors' initials in parentheses. For some modules, the external interface of the module is then summarized. The number and type of procedures' parameters and return values are given. Library procedures return a null value unless otherwise noted. The filename at the beginning is linked in in order to use these procedures in a program. After the external interface, related modules are listed.

record types(fields) with summary descriptions

functions(parameters) : return types with summary descriptions

Links: other library modules (author's initials)

B.1 Procedure Library Modules

These files appear in the ipl/procs/ directory. Each library module in this directory is a collection of one or more procedures. Some modules also introduce record types or use global variables that are of interest to programmers that use the module.

---

adlutils

This module processes address lists; addresses are represented using a record type label. Procedures for extracting the city, state, and ZIP code work for U.S. addresses only.

Links: lastname, io, namepfx, title

---

apply

apply(L:list, argument):any applies a list of functions to an argument. An example is apply([integer,log],10) which is equivalent to integer(log(10)).

---

argparse

argparse(s:string):list parses s as a command line and returns a list of arguments. At present, it does not accept any escape conventions.

---

array

This module provides a multidimensional array abstraction with programmer-supplied base indices. create_array(lbs:list, ubs:list, value):array creates an n-dimensional array with specified lower bounds, upper bounds, and with each array element having the given initial value. ref_array(A, i1, i2, ...) references the i1-th i2-th ... element of A.

---

asciinam (RJA)

asciiname(s:string):string returns the name of unprintable ASCII character s.

---

base64 (DAG)

This module provides base64 encodings for MIME (RFC 2045). Among other things, this facilitates the writing of programs that read e-mail messages.

base64encode(string) : string returns the base64 encoding of its argument.

base64decode(string) : string? returns a base64 decoding of its argument. It fails if the string is not base64 encoded.

---

basename (REG, CAS)

basename(name, suffix) : string removes any path information as well as the specified suffix, if present. If no suffix is given, the whole filename (sans path) is returned.

---

binary (RJA)

These procedures support conversion of Icon data elements to and from binary data formats. The control procedures pack() and unpack() take a format string that controls conversions of several values, similar to the printf() C library function.

pack(template, *value*$_1$, ...) : packed_binary_string packs the *value* arguments into a binary structure, returning a string containing the structure. The elements of any lists in the *value* parameters are processed individually as if they were spliced into the *value* parameter list. The template characters give the order and type of values, as follows:

a	ASCII, null padded	A	ASCII, space padded
b	bitstring, low-to-high order	B	bitstring, high-to-low order
h	hexadecimal string, low-nibble-first	H	hexadecimal string, high-nibble-first
c	signed char	C	unsigned char
s	signed short	S	unsigned short
i	signed integer	I	unsigned integer
l	signed long	L	unsigned long
n	short in "network" order (big-endian)	N	long in "network" order (big-endian)
v	short in VAX order (little endian)	V	long in VAX order (little endian)
f	single-precision float in IEEE Motorola format	d	double precision floats in IEEE Motorola format
e	extended-precision float in IEEE Motorola format 80-bit	E	extended-precision float in IEEE Motorola format 96-bit
x	skip forward a byte (null-fill for pack())	X	back up a byte
@	go to absolute position (null-fill if necessary for pack())	u	uuencoded/uudecoded string

Each letter may be followed by a numeric count. Together the letter and the count make a field specifier. Letters and numbers can be separated by white space that is ignored. Types in [AaBbHh] consume one value from the "value" list and produce a string of the length given as the field-specifier-count. The other types consume "field-specifier-count" values from the "value" list and append the appropriate data to the packed string.

unpack(template, string) : value_list does the reverse of pack(): it takes a string representing a structure and expands it out into a list of values. The template has mostly the same format as for pack().

---

bincvt (RJA)

These procedures are for processing of binary data read from a file.

unsigned(s:string) : integer converts a binary byte string into an unsigned integer.

raw(s:string) : integer puts raw bits of characters of string s into an integer. If the size of s is less than the size of an integer, the bytes are put into the low order part of the integer, with the remaining high order bytes filled with zero. If the string is too large, the most significant bytes will be lost -- no overflow detection.

rawstring(i:integer, size) : string creates a string consisting of the raw bits in the low order size bytes of integer i.

---

bitint

int2bit(i) : string produces a string with the bit representation of i.

bit2int(s) : integer produces an integer corresponding to the bit representation i.

---

bitstr,bitstrm

These two modules operate on numeric values represented by arbitrarily long strings of bits, stored without regard to character boundaries. In conjunction with arbitrary precision integers, this facility can deal with bit strings of arbitrary size. record BitString(s, buffer, bufferBits) represents bit strings internally. See the header comments atop the source code for the public API and examples for these modules.

---

bufread (CAS)

These procedures provide lookahead within an open file. The procedures bufopen(), bufread(), and bufclose() mirror the built-in open(), read(), and close().

bufopen(s:&input): file? opens a file name s for buffered read and lookahead.

bufread(f:&input): string reads the next line from file f. You cannot bufread() &input unless you have previously called bufopen() on it.

bufnext(f:&input, n:1): string returns the next nth line from file f without changing the next record to be read by bufread().

bufclose(f:&input): file close file f.

In addition to processing the current line, one may process subsequent lines **before** they are logically read.

---

calls

Procedures to deal with procedure invocations encapsulated in records.

record call(proc, args) encapsulates a procedure and its argument list.

invoke(call) : any* invokes a procedure with an argument from a call record.

call_image(call) : string produces a string image of a call.

make_call(string) : call makes a call record from a string image of an invocation.

make_args(string) : list makes an argument list from a comma-separated string.

call_code(string) : string produces a string of Icon code to construct a call record.

write_calltable(T:table, p:procedure, f:file):null writes a table of calls (all to procedure p) out to a file. The format is name=proc:arg1,arg2,?,argn,

read_calltable(f:file) : table reads a call table file into a table.

Links: ivalue, procname.

---

capture (DAG)

capture(f:file) replaces write(), writes(), and stop() with procedures that echo those elements that are sent to &output to the file f.

uncaptured_write(s:string...), uncaptured_writes(s:string...) and uncaptured_stop(s:string...) allow output to be directed to &output without echoing to the capture file. These are handy for placing progress messages and other comforting information on the screen.

---

caseless (NJL)

These procedures are analogous to the standard string-analysis functions except that uppercase letters are considered equivalent to lowercase letters. They observe the string scanning function conventions for defaulting of the last three parameters.

anycl(c, s, i1, i2) succeeds and produces i1 + 1, provided map(s[i1]) is in cset(map(c)) and i2 is greater than i1. It fails otherwise.

balcl(c1, c2:'(', c3:')', s, i1, i2) generates the sequence of integer positions in s preceding a character of cset(map(c1)) in map(s[i1:i2]) that is balanced with respect to characters in cset(map(c2)) and cset(map(c3)), but fails if there is no such position.

findcl(s1, s2, i1, i2) generates the sequence of integer positions in s2 at which map(s1) occurs as a substring in map(s2[i1:i2]), but fails if there is no such position.

manycl(c,s,i1,i2) produces the position in s after the longest initial sequence of characters in cset(map(c)) within map(s[i1:i2]). It fails if map(s[i1]) is not in cset(map(c)).

matchcl(s1, s2, i1, i2) : integer? produces i1 + *s1 if map(s1) == map(s2[i1+:=*s1]).

uptocl(c, s, i1, i2) generates the sequence of integer positions in s preceding a character of cset(map(c)) in map(s[i1:i2]). It fails if there is no such position.

---

cgi (JvM, CLJ)

The cgi library provides support for development of Common Gateway Interface server side web based applications, commonly called CGI scripts.

global cgi : table contains keys that are the names of input fields in the invoking HTML page's form, and values are whatever the user typed in those input fields.

cgiInput(type, name, values) writes HTML INPUT tags with the given type and name for each of a list of values. The first value's input tag is CHECKED.

cgiSelect(name, values) writes an HTML SELECT with a given name and OPTION tags for a list of values. The first value's OPTION tag is SELECTED.

cgiXYCoord(hlst) : string is used with an ISMAP. If the x and y coordinates are between certain boundaries, it returns the value of the list element that was entered.

cgiMyURL() : string returns the URL for the current script, as obtained from the SERVER_NAME and SCRIPT_NAME environment variables.

cgiPrintVariables(T) prints a Unicon table using simple HTML formatting.

cgiError(L) generates an error message consisting of the strings in list L, with L[1] as the title and subsequent list elements as paragraphs.

cgiHexVal(c) produces a value from 0 to 15 corresponding to hex chars 0 to F.

cgiHexChar(c1,c2) produces a char corresponding to two char-encoded hex digits.

cgiColorToHex(s) : string produces a 24-bit hex color value corresponding to a string color name. At present, only the colors black, gray, white, pink, violet, brown, red, orange, yellow, green, cyan, blue, purple, and magenta are supported.

cgiPrePro(filename, def) copies out parts of a named HTML file, writing out anything between pairs of <! − − ALL or <! − − *def* comments.

cgiRndImg(L, s) writes an HTML IMG tag for a random element of L, which should be a list of image filenames. The tag has ALT text given in string s.

cgiOptwindow(opts, args...) : window? attempts to open an Icon window, either on the X server or else on display :0 of the client's machine (as defined by the IP address in REMOTE_ADDR). The Icon window is typically used to generate a .GIF image to which a link is embedded in the CGI program's output.

main(args) is included in the CGI library; you do not write your own. The CGI main() procedure generates an HTML header, parses the CGI input fields into a global table cgi, generates a background by calling the user's cgiBBuilder() function, if any, and calls the user's cgimain() function.

cgiBBuilder(args...) : table is an optional procedure that a CGI program can use to define the general appearance of its generated web page output. If the user application defines this function, it should return a table which contains keys "background", "bgcolor", "text", "link", "vlink", and "bgproperties" with appropriate values to go into the BODY tag and define background color and texture for the CGI page.

cgimain(args) is the entry point for CGI programs. When you use the CGI library, its main() initializes things and then calls your cgimain() to generate the HTML content body for the client's web page.

---

codeobj

This module provides a way of storing Icon values as strings and retrieving them.

encode(x:any) : string converts x to a string s.

decode(s:string): any converts a string in encode() format back to x.

These procedures handle all Unicon values, including structures of arbitrary complexity. For scalar types (null, integer, real, cset, and string), decode(encode(x)) === x. For structures (list, set, table, and record types) decode(encode(x)) is not identical to x, but it has the same "shape" and its elements bear the same relation to the original as if they were encoded and decode individually. Not much can be done with files, functions and procedures, and co-expressions except to preserve type and identification. The encoding of strings and csets

handles characters in a way that is safe to write to a file and read it back.

Links: escape, gener, procname, typecode. See also: object.icn.

---

colmize (RJA)

colmize(L:list,mx:80,sp:2,mi:0,tag,tagsp:2,tagmi:0,roww,dist) : string* arranges data items (from list of strings L) into multiple columns. mx is the maximum width of output lines. sp is the minimum number of spaces between columns. mi is the minimum column width. tag is a label to be placed on the first line of output. Items are arranged column-wise unless roww is nonnull; by default the sequence runs down the first column, then down the second, etc. colmize() prints the items in as few vertical lines as possible.

---

complete (RLG)

complete(s:string,st) : string* takes a partial string, s, and generates those strings in st that begin with s. st must be a list or set of strings.

---

complex

The following procedures perform operations on complex numbers.

record complex(r,i) creates a complex number with real part r and imaginary part i

cpxadd(x1,x2) : complex add complex numbers x1 and x2

cpxdiv(x1,x2) : complex divide complex number x1 by complex number x2

cpxmul(x1,x2) : complex multiply complex number x1 by complex number x2

cpxsub(x1,x2) : complex subtract complex numbers x2 from x1

cpxstr(x) : complex convert complex number x to string representation

strcpx(s) : complex convert string to complex number

---

conffile (DAG)

This module parses "configuration files" into Icon structures for easy access. The service is similar to command-line option handling, except that configuration files can contain structured data such as lists or tables. A configuration file supplies values for a set of named *directives*. The directives and their values are read in to a table, usually during program initialization. The types of all allowed directives are specified before the configuration file is accessed.

---

convert

This module contains numeric conversions between bases. There are several other procedures related to conversion that are not yet part of this module.

exbase10(i, j) : string converts base-10 integer i to base j.

inbase10(s, i) : integer converts base-i integer s to base 10.

radcon(s, i, j) : integer converts base-i integer s to base j.

---

created

created(string) : integer returns (approximately) the number of structures of a given type that have been created. Links: serial.

---

currency (RJA)

currency(amt, wid:0, neg:"-", frac:2, whole:1, sign:"$", decimal:".", comma:",") : string formats amt in a currency format that defaults to U.S. currency. amt can be a real, integer, or numeric string. wid is the output field width; its amount is right-adjusted. The returned string will be longer than width if necessary to preserve significance. neg is the character string to be used for negative amounts, and is placed to the right of the amount. frac and whole are the exact number of digits to use right of the decimal, and the minimum number of digits to appear left of the decimal, respectively. The currency sign prefixes the returned string. The characters used for decimal point and comma may also be supplied.

---

datecomp (CSM)

These procedures do simple date comparisons. The parameters are strings of the form mm/dd/yyyy or &date-compatible yyyy/mm/dd.

dgt(date1:string, date2:string) : ? succeeds if date1 is later than date2.

dlt(date1:string, date2:string) : ? succeeds if date1 is earlier than date2.

deq(date1:string, date2:string) : ? succeeds if date1 is equal to date2.

futuredate(date:string) : ? succeeds if date is in the future.

pastdate(date:string) : ? succeeds if date is in the past.

getmonth(date:string) : string returns the month portion of a date.

getday(date:string) : string returns the day portion of a date.

getyear(date:string) : string returns the year portion of a date.

---

datefns (CH)

Date and calendar adaptations of C functions from "The C Programming Language" (Kernighan and Ritchie, Prentice-Hall) and "Numerical Recipes in C" (Press et al, Cambridge). They represent dates using the record type below.

record date_rec(year, month, day, yearday, monthname, dayname)

initdate() initializes the global data before using the other functions.

today() : date_rec produces a computationally useful value for today's date

julian(date) : integer converts a date_rec to a Julian day number

unjulian(julianday) : date_rec produces a date from the Julian day number

doy(year, month, day) : integer returns the day-of-year from (year, month, day)

wrdate(leadin, date) writes a basic date string preceded by a leadin to &output

---

datetime (RJA, REG)

Miscellaneous date and time operations. See also: datefns.icn.

global DateBaseYear : 1970 is a time origin for several functions. An environment variable by the same name overrides the default value.

ClockToSec(string) : integer converts &clock format to seconds past midnight.

DateLineToSec(dateline,hoursFromGmt) : integer converts &dateline format to seconds past DateBaseYear. DateToSec(string) : integer converts &date format (yyyy/mm/dd) to seconds past DateBaseYear.

SecToClock(integer) : string converts a number of seconds past midnight to a string in the

format of &clock. SecToDate(integer) : string converts a number of seconds past DateBaseYear to a string in Icon &date format (yyyy/mm/dd).

SecToDateLine(sec,hoursFromGmt) : string yields a date in &dateline format.

SecToUnixDate(sec,hoursFromGmt) : string returns a date and time in typical UNIX format: Jan 14 10:24 1991.

calendat(j) : date1 returns a record with the month, day, and year corresponding to the Julian Date Number j.

date() : string produces the natural date in English.

dayoweek(day, month, year) : string yields the day of the week for a given date.

full13th(year1, year2) generates records giving the days on which a full moon occurs on Friday the 13th in the range from year1 through year2.

julian(m,d,y) yields the Julian Day Number for the given month, day, and year.

pom(n, phase) returns record with the Julian Day number of fractional part of the day for which the nth such phase since January, 1900. Phases are encoded as:

0 = new moon 1 = first quarter 2 = full moon 3 = last quarter

GMT is assumed.

saytime() computes the time in natural English. If an argument is supplied it is used as a test value to check the operation the program.

walltime() : integer produces the number of seconds since midnight. Beware wrap-around when used in programs that span midnight.

---

db

These procedures provide an interface to the ODBC database facilities that does not require knowledge of SQL. It also provides compatibility procedures for an earlier version of the ODBC interface.

dbdelete(db, filter...):integer deletes rows from db that satisfy filters. Warning: if the filter criteria are omitted, the database will be emptied by this operation!

dbinsert(db, row:record) inserts a record as a tuple (row) into db.

dbselect(db,columns,condition,ordering):integer selects columns from db. columns defaults to "all", condition defaults to unconditionally, and ordering defaults to unordered.

dbupdate(db:database,row:record) updates the db tuple corresponding to row.

dbopen(dsn, tabl, user, password):f is an alias for open(dsn,"o",...). tabl is optional.

dbclose(db) is an alias for close()

dbfetch(db) is an alias for fetch()

dbsql(db, query) is an alias for sql()

---

dif (RJA)

dif(strm:list,compare:"===",eof,group:groupfactor):list* generates differences between input streams. Returns a list of records, one for each input stream, with each record containing a list of items that differ and their positions in the input stream. The record type is as: record diff_rec(pos,diffs). dif() fails if there are no differences.

strm is a list of input streams from which dif() will extract its input "records". The elements can be any of the following types, with corresponding actions:

Type	Action
file	file is "read" to get records
co-expression	co-expression is activated to get records
list	records are "gotten" (get()) from the list
diff_proc	a record type that has two fields, a procedure to call and the argument to pass to it. Its definition is: record diff_proc(proc,arg). It allows procedures supplied by dif's caller to be called to get records.

compare is a procedure that succeeds if two records are "equal", and fails otherwise. The comparison must allow for the fact that the EOF object might be an argument, and a pair of EOFs must compare equal.

eof is an object that is distinguishable from other objects in the stream.

group is a procedure that is called with the current number of unmatched items as its argument. It must return the number of matching items required for file synchronization to occur. The default (procedure groupfactor()) is the formula Trunc((2.0 * Log(M)) + 2.0) where M is the number of unmatched items.

---

digitcnt

digitcnt(file:&input) : list counts the number of each digit in a file and returns a ten-element list with the counts.

---

equiv

equiv(x,y) : any? tests the equivalence of two values. For non-structures, it returns x1 === x2. For structures, the test is for *shape*. For example, equiv([],[]) succeeds. It handles loops, but does not recognize them as such. The concept of equivalence for tables and sets is weak if their elements are themselves structures. There is no concept of order for tables and sets, yet it is impractical to test for equivalence of their elements without imposing an order. Since structures sort by "age", there may be a mismatch between equivalent structures in two tables or sets.

---

escapesq (RJA)

These procedures manipulate escape sequences in Icon string format.

escapeseq() : string is a matching procedure for Icon string escape sequences

escchar(string) : string produces the character value of an Icon escape sequence

escape() converts a string with escape sequences to the string it represents. For example, escape("\\143\\141\\164") produces the string "cat".

quotedstring() matches a complete quoted string.

---

eval

eval(string) : any* analyzes a string representing an Icon function or procedure call and evaluates the result. Operators can be used in functional form, as in "*(2,3)". This procedure cannot handle nested expressions or control structures. It assumes the string is well formed. The arguments can only be Icon literals. Escapes, commas, and parentheses in string literals

are not handled. In the case of operators that are both unary and binary, the binary form is used. Links: ivalue.

---

evallist

evallist(expr, n, ucode, ...) : list returns a list of the results written by a program consisting of expression expr (normally a generator); n is the maximum size of the list, and the trailing arguments are ucode files to link with the expression. Requires: system(), /tmp, pipes. Links: exprfile.

---

everycat

everycat(x1, x2, ...) : string* generates the concatenation of every string from !x1, !x2, ... For example, if first := ["Mary", 'Joe', "Sandra"] and last := ["Smith", "Roberts"] then every write(everycat(first, " ", last)) writes Mary Smith, Mary Roberts, Joe Smith, Joe Roberts, Sandra Smith, Sandra Roberts. x1, x2, ... can be any values for which !x1, !x2, ... are convertible to strings. In the example, the second argument is a one-character string " ", so that !" ' generates a single blank.

---

exprfile

exprfile(exp, link, ...) : file produces a pipe to a program that writes all the results generated by exp. The trailing arguments name link files needed for the expression. exprfile() closes any previous pipe it opened and deletes its temporary file. Therefore, exprfile() cannot be used for multiple expression pipes. If the expression fails to compile, the global expr_error is set to 1; otherwise 0.

exec_expr(expr_list, links[]) : string* generates the results of executing the expression contained in the lists expr_list with the specified links.

Requires: system(), pipes, /tmp. Links: io.

---

factors (REG, GMT)

This file contains procedures related to factorization and prime numbers.

factorial(n) returns n!. It fails if n is less than 0.

factors(i, j) returns a list containing the factors of i, up to maximum j; by default there is no limit.

gfactorial(n, i) generalized factorial; n x (n - i) x (n - 2i) x ...

ispower(i, j) succeeds and returns root if i is k^j

isprime(n) succeeds if n is a prime.

nxtprime(n) returns the next prime number beyond n.

pfactors(i) returns a list containing the primes that divide i.

prdecomp(i) returns a list of exponents for the prime decomposition of i.

prime() generates the primes.

primel() generates the primes from a precompiled list.

primorial(i,j) product of primes j <= i; j defaults to 1.

sfactors(i, j) is the same as factors(i, j), except output is in string form with exponents for repeated factors

Requires: Large-integer arithmetic and prime.lst for primel(). Links: io, numbers.

---

fastfncs

These procedures implement integer-values using a best known method.

acker(i, j)	Ackermann's function
fib(i)	Fibonacci sequence
g(k, i)	Generalized Hofstader nested recurrence
q(i)	"Chaotic" sequence
robbins(i)	Robbins numbers

See also: iterfncs.icn, recrfncs.icn. Links: factors, memrfncs.

---

filedim

filedim(s, p) : textdim computes the number of rows and maximum column width of the file named s. The procedure p, which defaults to detab, is applied to each line. For example, to have lines left as is, use filedim(s, 1). The return value is a record that uses the declaration record textdim(cols, rows).

---

filenseq (DAG)

nextseqfilename(dir, pre, ext) : string? creates the next filename in a series of files (such as successive log files). Usage: fn := nextseqfilename(".", ".", "log") returns the (non-existent) filename next in the sequence .\.*.log (* represents 1, 2, 3, ...) or fails.

---

findre (RLG)

findre(s1,s2,i,j) : integer* is like the built-in function find(), except its first argument is a regular expression similar to the ones the Unix egrep command uses. A no argument invocation wipes out all static structures utilized by findre(), and then forces a garbage collection. findre() takes a shortest-possible-match approach to regular expressions. If you look for "a*", findre() will not even bother looking for an "a". It will just match the empty string.

---

gauss (SBW)

gauss_random(x, f) produces a Gaussian distribution about the value x. Parameter f can be used to alter the shape of the Gaussian distribution (larger values flatten the curve...) Produce a random value within a Gaussian distribution about 0.0. (Sum 12 random numbers between 0 and 1, (expected mean is 6.0) and subtract 6 to center on 0.0.

---

gdl, gdl2 (RLG)

gdl(dir:string) : list returns a list containing everything in a directory. You can use this file as a template, modifying the procedures according to the needs of the program in which they are used.

gdlrec(dir, findflag) : list does same thing as gdl except it recursively descends through subdirectories. If findflag is nonnull, the UNIX "find" program is used; otherwise the "ls" program is used.

---

gener

These procedures generate sequences of results.

days() : string* produces the days of the week, starting with "Sunday".

hex() : string* is the sequence of hexadecimal codes for numbers from 0 to 255

label(s,i) : string* produces labels with prefix s starting at i

multii(i, j) : integer* produces i * j i's

months() : string* produces the months of the year

octal() : string* produces the octal codes for numbers from 0 to 255

star(s) : string* produces the closure of s starting with the empty string and continuing in lexical order as given in s

---

genrfncs

These procedures generate various mathematical sequences of results. Too many are included to list them all here; consult the source code for a complete listing.

arithseq(i, j) : string* arithmetic sequence starting at i with increment j.

chaosseq() chaotic sequence

factseq() factorial sequence

fibseq(i, j, k) generalized Fibonacci (Lucas) sequence with additive constant k

figurseq(i) series of i'th figurate number

fileseq(s, i) generate lines (if i is null) or characters (except line terminators) from file s.

geomseq(i, j) geometric sequence starting at i with multiplier j

irepl(i, j) j instances of i

multiseq(i, j, k) sequence of (i * j + k) i's

ngonalseq(i) sequence of the i polygonal number

primeseq() the sequence of prime numbers

powerseq(i, j) sequence i ^ j, starting at j = 0

spectseq(r) spectral sequence integer(i * r), i - 1, 2, 3, ...

starseq(s) sequence consisting of the closure of s starting with the empty string and continuing in lexical order as given in s

Requires: co-expressions. Links: io, fastfncs, partit, numbers.

---

getmail (CS)

getmail(x):message_rec* reads an Internet mail folder and generates a sequence of records, one per message, failing when end-of-file is reached. Each record contains the message header and message text components parsed into fields. The argument x is either the name or the file handle. If getmail() is resumed after the last message is generated, it closes the mail folder and returns failure. If getmail() generates an incomplete sequence (does not close the folder and return failure) and is then restarted (not resumed) on the same or a different mail folder, the previous folder file handle remains open and inaccessible. If message_records are stored in a list, the records may be sorted by individual components (like sender, _date, _subject) using the sortf() function.

---

getpaths (RLG)

getpaths(args[]) : string* generates the paths supplied as arguments followed by those paths in the PATH environment variable, if one is available. A typical invocation might look like:

```
open(getpaths("/usr/local/lib/icon/procs") || filename)
```

getpaths() will be resumed in the above context until open succeeds in finding an existing, readable file.

---

graphpak

The procedures here use sets to represent directed graphs. See "The Icon Programming Language", third edition, pp. 233-236. A graph has two components: a list of nodes and a two-way lookup table. The nodes in turn are sets of pointers to other nodes. The two-way table maps a node to its name and vice-versa. Graph specifications are given in files in which the first line is a white-space separated list of node names and subsequent lines give the arcs, with -> between source and destination.

record graph(nodes:list, lookup:table) represents a graph

read_graph(f:file) : graph reads a graph from a file

write_graph(g:graph, f:file) : null writes a graph to a file

closure(node) : set computes the transitive closure of a node.

---

hexcvt (RJA)

hex(s) : integer converts a string of hex digits into an integer.

hexstring(i,n,lc) : string produces the hexadecimal representation of the argument. If n is supplied, a minimum of n digits appears in the result; otherwise there is no minimum, and negative values are indicated by a minus sign. If lc is non-null, lowercase characters are used instead of uppercase.

---

html (GMT)

These procedures assist in processing HTML files:

htchunks(file) : string* generates HTML chunks in a file. Results beginning with <!-- are unclosed comments (legal comments are deleted); < begins tags; others are untagged text.

htrefs(f) : string* generates the tagname/keyword/value combinations that reference other files. Tags and keywords are returned in upper case.

httag(string) : string produces the name of a tag contained within a tag string.

htvals(s) : string* generates the keyword/value pairs from a tag.

urlmerge(base,new) : string interprets a new URL in the context of a base URL.

---

ichartp (RLG)

ichartp implements a simple chart parser - a slow but easy-to-implement strategy for parsing context free grammars (it has a cubic worst-case time factor). Chart parsers are flexible enough to handle a variety of natural language constructs, and lack many of the troubles associated with empty and left-recursive derivations.

Links: trees, rewrap, scan, strip, stripcom, strings. Requires: co-expressions.

---

iftrace (SBW, REG)

These procedures trace Icon functions by using procedure wrappers to call the functions. iftrace(fncs[]) sets tracing for a list of function names. Links: ifncs.

---

image (MG, REG, DY)

Image(x, style:1) : string generalizes image(x), providing detailed information about structures. The style determines the formatting and order of processing. Style 1 is indented, with] and) at end of last item. Style 2 is also indented, with] and) on new line. Style 3 puts the whole image on one line. Style 4 is like style 3, with structures expanded breadthfirst instead of depthfirst as for other styles.

Structures are identified by a letter identifying the type followed by an integer. The tag letters are "L" for lists, "R" for records, "S" for sets, and "T" for tables. The first time a structure is seen, it is imaged as the tag followed by a colon, followed by a representation of the structure. If the structure is encountered again, only the tag is given.

---

inbits (RLG)

inbits(file,len:integer) : integer re-imports data converted into writable form by outbits(). See also: outbits.icn.

---

indices

indices(spec:list, last) : list produces a list of the integers given by the specification spec, which is a comma separated list of positive integers or integer spans, as in "1,3-10,..." If last is specified, it is used for a span of the form "10-". In a span, the low and high values need not be in order. For example, "1-10" and "10-1" are equivalent. Similarly, indices need not be in order, as in "3-10, 1,...". Empty values, as in "10,,12" are ignored. indices() fails if the specification is syntactically erroneous or if it contains a value less than 1.

---

inserts

inserts(table,key,value) : table inserts values into a table in which the same key can have more than one value (i.e., duplicate keys). The value of each element is a list of inserted values. The table must be created with default value &null. (RJA)

---

intstr (RJA)

intstr(i:integer,size:integer) : string produces a string consisting of the raw bits in the low order size bytes of integer i. This procedure is used for processing of binary data to be written to a file. Note that if large integers are supported, this procedure still will not work for integers larger than the implementation defined word size due to the shifting in of zero-bits from the left in the right shift operation.

---

io

These procedures provide facilities for handling input, output, and files. Some require loadfunc(). Links: random, strings.

fcopy(fn1:string,fn2:string) copies a file named fn1 to file named fn2.

exists(name:string) : file? succeeds if name exists as a file but fails otherwise.

filelist(s,x) : list returns a list of the file names that match the specification s. If x is nonnull, any directory is stripped off. At present it only works for UNIX.

filetext(f) : list reads the lines of f into a list and returns that list

readline(file) : string? assembles backslash-continued lines from the specified file into a single line. If the last line in a file ends in a backslash, that character is included in the last line read.

splitline(file, line, limit) splits line into pieces at first blank after the limit, appending a backslash to identify split lines (if a line ends in a backslash already, that's too bad). The pieces are written to the specified file.

Buffered input and output: ClearOut() remove contents of output buffer without writing

Flush() flush output buffer

GetBack() get back line written

LookAhead() look ahead at next line

PutBack(s) put back a line

Read() read a line

ReadAhead(n) read ahead n lines

Write(s) write a line

See also module bufread for a multi-file implementation of buffered input.

Path searching: dopen(s) : file? opens and returns the file s on DPATH.

dpath(s) : string? returns the path to s on DPATH.

pathfind(fname, path:getenv("DPATH")) : string? returns the full path of fname if found along the space-separated list of directories "path". As is customary in Icon path searching, "." is prepended to the path.

pathload(fname,entry) calls loadfunc() to load entry from the file fname found on the function path. If the file or entry point cannot be found, the program is aborted. The function path consists of the current directory, then getenv("FPATH"), and finally any additional directories configured in the code.

Parsing file names: suffix() : list parses a hierarchical file name, returning a 2-element list: [prefix,suffix]. For example, suffix("/a/b/c.d") produces ["/a/b/c","d"]

tail() : list parses a hierarchical file name, returning a 2-element list: [head,tail]. For example, tail("/a/b/c.d") produces ["/a/b","c.d"].

components() : list parses a hierarchical file name, returning a list of all directory names in the file path, with the file name (tail) as the last element. For example, components("/a/b/c.d") produces ["/","a","b","c.d"].

Temporary files: tempfile(prefix:"",suffix:"",path:".",len:8) produces a temporary file that can be written. The name is chosen so as not to overwrite an existing file. The prefix and

suffix are prepended and appended, respectively, to a randomly chosen number. The path is prepended to the file name. The randomly chosen number is fit into a field of len characters by truncation or right filling with zeros as necessary. It is the user's responsibility to remove the file when it is no longer needed.

tempname(prefix:"", suffix:"", path:".", len:8) returns a temporary file name.

---

iolib (RLG, NA)

This library provides control functions for text terminals, based on legacy ANSI and VT-100 devices. The TERM and TERMCAP environment variables must be set in order to use this library. TERM tells iolib what driver you are using, e.g. TERM=ansi-color. The TERMCAP variable gives the location of the termcap database file, as provided on your UNIX system.

Requires: UNIX, co-expressions. See also: iscreen.icn.

---

iscreen (RLG)

This file contains some rudimentary screen functions for use with iolib.icn.

clear() clears the screen (tries several methods)

clear_emphasize() clears the screen to all-emphasize mode.

emphasize() initiates emphasized (usually reverse video dark on light) mode

boldface() initiates bold mode

blink() initiates blinking mode

normal() resets to normal mode

message(s) displays message s on 2nd-to-last line

underline() initiates underline mode

status_line(s,s2,p) draws status line s on the 3rd-to-last screen line; if s is too short for the terminal, s2 is used; if p is nonnull then it either centers, left-, or right-justifies, depending on the value, "c", "l", or "r".

Requires: UNIX. Links: iolib.

---

isort (RJA)

isort(x,keyproc,y) : list is a customizable sort procedure. x can be any Icon data type that supports the unary element generation (!) operator. The result is a sorted list of objects. The sort keys are obtained by calling keyproc() on each element of strucure x to obtain the sort key for that element. If keyproc is a procedure, the first argument to each call to keyproc() is the element for which the key is to be computed, and the second argument is isort's argument y, passed unchanged. The keyproc must produce the extracted key. Alternatively, keyproc can be an integer, in which case it is a subscript applied uniformly to each element to select a sort key. If keyproc is omitted, sorting uses the standard Icon sort order.

---

itokens (RLG)

itokens(file, nostrip) : TOK* breaks Icon source files up into tokens for use in things like pretty printers, preprocessors, code obfuscators, and so forth. itokens() suspends values of

type record TOK(sym:string, str:string). sym contains the name of the next token, such as "CSET", or "STRING". str gives that token's literal value. For example, the TOK for a literal semicolon is TOK("SEMICOL", ";"). For a mandatory newline, itokens() would suspend TOK("SEMICOL", "\n"). itokens() fails on end-of-file. It returns syntactically meaningless newlines if the second argument is nonnull. These meaningless newlines are returned as TOK records with a null sym field (i.e. TOK(&null, "\n")). If new reserved words or operators are added to a given implementation, the tables in this module have to be altered. Note: keywords are implemented at the syntactic level; they are not tokens. A keyword like &features is suspended as an & token followed by an identifier token. Links: scan. Requires: co-expressions.

---

ivalue

ivalue(s):any turns a string from image() into the corresponding Icon value. It handles integers, real numbers, strings, csets, keywords, structures, and procedures. For the image of a structure, it produces a result of the correct type and size, but values in the structure are not correct, since they are not encoded in the image. For procedures, the procedure must be present in the environment in which ivalue() is evaluated. This generally is true for built-in procedures (functions). All keywords are supported. The values produced for non-constant keywords are the values they have in the environment in which ivalue() is evaluated. ivalue() handles non-local variables (image() does not produce these), but they must be present in the environment in which ivalue() is evaluated.

---

jumpque

jumpque(queue:list,y):list moves y to the head if it is in the queue; otherwise it adds y to the head of the queue. A copy of the queue is returned.

---

kmap

kmap(string) : string maps uppercase and control characters into the corresponding lowercase letters. It is for graphic applications in which the modifier keys for shift and control are encoded in keyboard events.

---

lastc

These string scanning functions follow standard conventions for defaulting the last three parameters to the current scanning environment. (DAG)

lastc(c:cset, s:string, i1:integer, i2:integer) : integer succeeds and produces i1, provided either that i1 is 1, or that s[i1 - 1] is in c and i2 is greater than i1.

findp(c:cset, s1:string, s2:string, i1:integer, i2:integer) : integer* generates the sequence of positions in s2 at which s1 occurs provided that s2 is preceded by a character in c, or is found at the beginning of the string.

findw(c1:cset, s1:string, c2:cset, s2:string, i1:integer, i2:integer) : integer* generates the sequence of positions in s2 at which s1 occurs provided that s2 is preceded and followed by the empty string or a member of c1 and c2, respectively.

---

lastname

lastname(s:string) : string produces the last name in string s, which must be a name in conventional form. Obviously, it doesn't work for every possibility.

---

list2tab

list2tab(list) : null writes a list as a tab-separated string to &output. Carriage returns in files are converted to vertical tabs. See also: tab2list, tab2rec, rec2tab.

---

lists (REG, RLG)

These procedures implement list functions similar to string functions, including an implementation of list scanning, similar to the string scanning functions.

lcomb(L,i):L* produces all sublist combinations of L that have i elements.

ldelete(L,spec:string):L deletes values of L at indices given in spec; see indices.

lequiv(L1,L2):L? tests if acyclic lists L1 and L2 are structurally equivalent.

lextend(L,i,x):L extends L to at least size i, using initial value x.

limage(L):string list image function that shows elements' images, one level deep.

linterl(L1,L2):L interleaves elements of L1 and L2. If L1 and L2 are unequal size, the shorter list is extended with null values to the size of the larger list.

llpad(L,i,x):L produces a new list that extends L on its front (left) side.

lltrim(L,S):L produces a copy of L with elements of S trimmed on the left.

lmap(L1,L2,L3):L maps elements of L1 according to L2 and L3, similar to function map(s1, s2,s3). The operation x === y is used to determine if elements x and y are equivalent. No defaults are provided for omitted arguments.

lpalin(L):L produces a list palindrome of L concatenated with its reverse.

lpermute(L):L* produces all the permutations of list L.

lreflect(L,i:0):L returns L concatenated with its reversal to produce a palindrome. Parameter i specifies end conditions: 0 = omit first and last elements, 1 = omit first element, 2 = omit last element, 3 = don't omit element.

lremvals(L, x1, x2, ...) : list produces a copy of L with x1, x2, ... removed

lrepl(L,i):L replicates L i times.

lrotate(L,i):L produces a list with the elements of L, rotated i positions.

lrpad(L,i,x):L is like lextend(), but produces a new list instead of changing L.

lrtrim(L,S:set):L produces a copy of L with elements of S trimmed on the left.

lswap(L):L produces a copy of L with odd elements swapped with even elements.

lunique(L):L produces a list containing only unique list elements.

List Scanning

List scanning depends on the following underlying state mechanisms.

global l_POS, l_SUBJ are the current list scanning environment

record l_ScanEnvir(subject,pos) represents (nested) list scanning environments

l_Bscan(e1) : l_ScanEnvir enter (possibly nested) list scanning environment

l_Escan(l_OuterEnvir, e2) : any exit list scanning environment

Within a list scanning environment, the following procedures are equivalent to their string scanning counterparts.

```
l_any(L1, L2,i,j) : integer?
l_bal(L1,L2,L3,l,i,j) : integer*
l_find(L1,L2,i,j) : integer*
l_many(L1,L2,i,j) : integer?
l_match(L1,L2,i,j) : integer?
l_move(i) : list
l_pos(i) : integer
l_tab(i) : list
l_upto(L1,L2,i,j) : integer
```

l_any(), l_many(), and l_upto() take either sets of lists or lists of lists. l_bal() has no defaults for the first four arguments, since there is no list analogue to &cset, etc. List scanning environments are not maintained implicitly as for string scanning. You must use a set of nested procedure calls l_Bscan() and l_Escan(), as explained in the *Icon Analyst* 1:6 (June, 1991), p. 1-2. You cannot suspend, return, or otherwise break out of the nested procedure calls; they can only be exited via failure. Here is an example of how list scanning might be invoked:

```
suspend l_Escan(l_Bscan(some_list_or_other), {
    l_tab(10 to *l_SUBJ) & {
    if l_any(l1) | l_match(l2) then old_l_POS + (l_POS-1)
    }
    })
```

The functions compare lists, not strings, so l_find("h", l), for instance, will yield an error message: use l_find(["h"], l) instead. This becomes confusing when looking for lists within lists. Suppose l1 := ["junk", [["hello"]," ",["there"]],"!","m","o","r","e","junk"] and you wish to find the position in l1 at which the list [["hello"],' ',["there']] occurs. If you assign L2 := [["hello"]," ",["there"]], then the l_find() call needs to look like l_find([l2],l1). Links: indices. See also: structs.icn.

---

loadfile

loadfile(exp, link,...) creates and loads a program that generates the results of exp. The trailing arguments name link files needed for the expression. loadfile() returns a procedure that generates the results. Requires: system(), pipes, /tmp. Links: io.

---

longstr (JN,SBW,KW,RJA,RLG)

String scanning function longstr(l,s,i,j) : integer? works like any(), except that instead of taking a cset as its first argument, it takes instead a list or set of strings (l). Returns i + *x, where x is the longest string in l for which match(x,s,i,j) succeeds, if there is such an x.

---

lrgapprx

lrgapprx(i) : string produces an approximate of an integer value in the form i.jx10^k. It is primarily useful for large integers.

---

lu

lu_decomp(M, I) : real? performs LU decomposition on the square matrix M using the vector I. Both M and I are modified. The value returned is +1.0 or -1.0 depending on whether the number of row interchanges is even or odd. lu_decomp() is used in combination with lu_back_sub() to solve linear equations or invert matrices. lu_decomp() fails if the matrix is singular.

lu_back_sub(M, I, B) solves the set of linear equations $M\ x\ X = B$. M is the matrix as modified by lu_decomp(). I is the index vector produced by lu_decomp(). B is the right-hand side vector and return with the solution vector. M and I are not modified and can be used in successive calls of lu_back_sub() with different Bs. These procedures are based on algorithms given in "Numerical Recipes; The Art of Scientific Computing" by Press, Flannery, Teukolsky, and Vetterling; Cambridge University Press, 1986.

---

mapbit

mapbit(s:string) : string produces a string of zeros and ones corresponding to the bit patterns for the characters of s. For example, mapbit("Axe") produces the string "010000010111100001100101". Links: strings.

---

mapstr (RLG)

mapstrs(string, l1:list, l2:list) : string works like map(), except that instead of taking ordered character sequences (strings) as arguments 2 and 3, it takes ordered string sequences (lists). Suppose, for example, you wanted to bowdlerize a string by replacing the words "hell" and "shit" with "heck" and "shoot." You would call mapstrs as follows:

mapstrs(s, ["hell", "shit"], ["heck", "shoot"]). If you want to replace one string with another, just use the IPL replace() routine (in strings.icn). If l2 is longer than l1, extra members in l2 are ignored. If l1 is longer, however, strings in l1 that have no correspondent in l2 are simply deleted. mapstr() uses a longest-possible-match approach, so that replacing ["hellish", "hell"] with ["heckish", "heck"] will work as one would expect. Links: longstr.

---

math

binocoef(n:integer, k:integer) : integer? produces the binomial coefficient n over k. It fails unless 0 <= k <= n.

cosh(r:real) : real produces the hyperbolic cosine of r.

sinh(r:real) : real produces the hyperbolic sine of r.

tanh(r:real) : real produces the hyperbolic tangent of r.

Links: factors.

---

matrix (SBW, REG)

This file contains procedures for matrix manipulation. Matrices (arguments beginning with M) are represented as lists of lists. Links: lu.

matrix_width(M) : integer produces the number of columns in a matrix.

matrix_height(M) : integer produces the number of rows in a matrix.

write_matrix(file, M, x) outputs a matrix to a file, one row per line. If x is nonnull, elements are comma-separated and rows are enclosed in square brackets.

copy_matrix(M) : list produces a copy of a matrix.

create_matrix(n,m,x) : list creates an n by m matrix with initial value x.

identity_matrix(n,m) : list produces an identity matrix of size n by m.

add_matrix(M1,M2) : list produces the matrix addition of M1 and M2.

mult_matrix(M1,M2) : list produces the matrix multiplication of M1 and M2.

invert_matrix(M) : list produces the matrix inversion of M.

determinant(M) : real produces the determinant of M.

---

memlog (GMT)

memlog(f:&output) : integer writes a message to file f recording the current memory amount in use, amount reserved, and number of collections in the string and block regions. memlog() returns the total current usage.

---

morse (REG, RM)

morse(s:string) : string converts s to its International Morse Code (a.k.a. Continental Code) equivalent, as used by radio amateurs (hams).

---

mset (JPR)

same_value(d1,d2) : ? compares d1 and d2 for structural equivalence.

insert2(S:set, el) : set inserts el into S

member2(S:set, el) : ? tests whether el (or its structural equivalent) is in S.

delete2(S:set, el) : set deletes el (or its structural equivalent) from S.

This module implements set operations in which no two identical (structurally equivalent) values can be present in a set.

---

namepfx

namepfx(s:string) : string produces the "name prefix" from a name in standard form -- omitting any title, but picking up the first name and any initials. namepfx() only knows how to omit common titles found in module titleset. Obviously, it can't always produce the "correct" result. Links: lastname, titleset.

---

ngrams

ngrams(file,n,c:&letters,t) : string* generates a tabulation of the n-grams in the specified file. If c is non-null, it is the set of characters from which n-grams are taken (other characters

break n-grams). If t is non-null, the tabulation is given in order of frequency; otherwise in alphabetical order of n-grams.

---

numbers (REG, RJA, RLG, TK)

These procedures format numbers in various ways:

amean(L) : real returns arithmetic mean of numbers in L.

ceil(r) : integer returns nearest integer to r away from 0.

commas(s) : string inserts commas in s to separate digits into groups of three.

decipos(r,i:3,j:5):string? position decimal point at i in r in field width j.

digred(i) : integer reduces a number by adding digits until one digit is reached.

div(i:number,j:number) : real produces the result of real division of i by j.

fix(i, j:1, w:8, d:3) : string? formats i / j as a real (floating-point) number in a field of width w with d digits to the right of the decimal point, if possible. If w is less than 3 it is set to 3. If d is less than 1, it is set to 1. The function fails if j is 0 or if the number cannot be formatted.

floor(r) : integer nearest integer to r toward 0.

gcd(i,j):integer? returns greatest common divisor of i and j. It fails if both are 0.

gcdl(L:list): integer returns the greatest common division of a list of integers.

gmean(args?) : real? returns the geometric mean of numbers.

hmean(args?) : real? returns the harmonic mean of numbers.

large(i) : integer? succeeds if i is a large integer but fails otherwise.

lcm(i, j) : integer? returns the least common multiple of i and j.

lcml(L): integer? returns the least common multiple of the integers in the list L.

mceil(r) : integer returns the least integer greater than or equal to r.

mfloor(r) : integer returns the greatest integer less than or equal to r.

npalins(n) : string* generates palindromic n-digit numbers

roman(i) : string? converts i to Roman numerals.

round(r:real) : integer returns nearest integer to r.

sign(r) : integer returns sign of r.

spell(i : integer) : string? spells out i in English.

trunc(r:real) : integer returns nearest integer less than r.

unroman(string) : integer converts Roman numerals to integers.

Note that ceil() and floor() are the traditional Icon definitions of the ceiling and floor functions; mceil() and mfloor() implement the more commonly used mathematical definitions.

Links: strings.

---

openchk (DAG)

OpenCheck() causes subsequent opens and closes to write diagnostic information to &errout. Useful for diagnosing situations where many files are opened and closed and there is a possibility that some files are not always being closed.

---

options (RJA, GMT)

options(args:list, opt:string, err:stop) : table separates, interprets, and removes UNIX-style command options from a list of strings, returning a table of option values. Options are introduced by a "-" character or -- characters. An option name is either a single printable character, as in "-n" or "-?", or a string of letters, as in "-geometry". Options that begin with -- may have - characters in their names, as in --non-Euclidean-geometry. Valueless single-character options may be combined, for example as "-qtv". Some options require values. Generally, the option name is one argument and the value appears as the next argument, for example: "-F file.txt"
With a single-character argument name, the value may be concatenated, as in "-Ffile.txt". With multi-character arguments, the value may also be concatenated by using =, as in "-File=file.txt" (= also works with single character arguments). Options may be interspersed with non-option arguments. An argument of "-" is treated as a non-option. The special argument "--" terminates option processing. Non-option arguments are left in args for use by the caller. options() replaces arguments of the form @filename with arguments retrieved from the file "filename". Each line of the file is taken as a separate argument, exactly as it appears in the file. An argument and its value may also be on the same line separated by an equals sign or spaces.

The options string is a concatenation, with optional spaces between, of one or more option specs of the form *-name%* where - (or --) introduces the option. *name* is either a string of letters or any single printable character % is one of the following flag characters:

!	No value is required or allowed	:	A string value is required
+	An integer value is required	.	A real value is required

The leading "-" may be omitted for a single-character option. The "!" flag may be omitted except when needed to terminate a multi-character name. If the options string is omitted, any single letter is assumed to be valid and require no data.

Options appear as keys in the table with the leading - character removed, so options like --name have "-name" as their key. To remove the - character from the key, make the *first* character of the option string a = character. In this sitation, the procedure will reject -opt and --opt as duplicates; without the =, they are both allowed (and result in "opt" and "-opt" keys, respectively, in the table).

The err procedure will be called if an error is detected in the command line options. The procedure is called with one argument: a string describing the error that occurred. After err() is called, options() immediately returns the outcome of errproc(), without processing further arguments. Processed arguments will have been removed from args.

---

outbits (RLG)

outbits(i:integer, len:integer) string* fits variable, non-byte-sized blocks into 8-bit bytes, suspending byte-sized chunks of i converted to characters (most significant bits first) until there is not enough left of i to fill up an 8-bit character. The remainder is stored in a buffer until outbits() is called again, at which point the buffer is combined with the new i and output in the same manner as before. The buffer is flushed by calling outbits() with a

null i argument. len gives the number of bits in i to preserve; bits that are discarded are the most significant ones. A trivial example of how outbits() might be used:

```
outtext := open("some.file.name","w")
L := [1,2,3,4]
every writes(outtext, outbits(!L,3))
writes(outtext, outbits(&null,3)) # flush buffer
```

List L may be reconstructed with inbits():

```
intext := open("some.file.name")
L := []
while put(L, inbits(intext, 3))
```

Note that outbits() is a generator, while inbits() is not. See also: inbits.icn.

---

packunpk (CT, RLG)

pack(num:i,width:i):string produces a binary-coded decimal representation of num in which each character contains two decimal digits stored in four bits each.

unpack(val:s,width:i):string converts a binary-coded decimal back to a string representation width characters long of the original source integer.

unpack2(val:string) : integer converts a binary-coded decimal back into its original source integer.

Links: convert.

---

partit

partit(i,min:i,max:i):L* generates the partitions of i; that is the ways that i can be represented as a sum of positive integers with minimum and maximum values.

partcount(i,min:i,max:i):integer returns the number of partitions.

fibpart(i):L returns a list of Fibonacci numbers that is a partition of i.

Links: fastfncs, numbers.

---

patterns

This module provides string scanning procedure equivalents for most SNOBOL4 patterns and some extensions. It is largely subsumed by Unicon's pattern type.

---

patword (KW)

patword(s:string) : string returns a letter pattern in which each different character in s is assigned a letter. For example, patword("structural") returns "abcdebdcfg".

---

phoname (TRH)

phoname(telno:string) : string* generates the letter combinations corresponding to the digits in a telephone number. The number of possibilities is very large. This procedure should be used in a context that limits or filters its output.

---

plural

plural(word:string) : string produces the plural form of a singular English noun. The procedure here is rudimentary and is not correct in all cases.

---

polystuf

These procedures are for creating and performing operations on single-variable polynomials (like ax^2 + bx + c). A polynomial is represented as a table in which the keys are exponents and the values are coefficients. (EE)

poly(c1, e1, c2, e2, ...) : poly creates a polynomial from the parameters given as coefficient-exponent pairs: c1x^e1 + c2x^e2 + ...

is_zero(n) : ? determines if n = 0

is_zero_poly(p) : ? determines if a given polynomial is 0x^0

poly_add(p1, p2) : poly returns the sum of two polynomials

poly_sub(p1, p2) : poly returns the difference of p1 - p2

poly_mul(p1, p2) : poly returns the product of two polynomials

poly_eval(p, x) : poly finds the value of polynomial p evaluated at the given x.

term2string (c, e) : string converts one coefficient-exponent pair into a string.

poly_string(p) : string returns the string representation of an entire polynomial.

---

printcol (RJA)

printcol(items, fields, title:"", pagelength:30000, linelength:80, auxdata) deals with the problem of printing tabular data where the total width of items to be printed is wider than the page. Simply allowing the data to wrap to additional lines often produces marginally readable output. This procedure facilitates printing such groups of data as vertical columns down the page length, instead of as horizontal rows across the page. That way many, many fields can be printed neatly.transformation can be a nuisance.

The arguments are:

items: a co-expression that produces a sequence of items (usually structured data, but not necessarily) for which data is to be printed.

fields: a list of procedures to produce the field's data. Each procedure takes two arguments. The procedure's action depends upon what is passed in the first argument:

header produces the row heading string to be used for that field (the field name).

width produces the maximum field width (including the column header).

other produces the field value string for the item passed as the argument.

The second argument is arbitrary data from the procedures with each invocation. The data returned by the first function on the list is used as a column heading (the item name).

auxdata: arbitrary auxiliary data to be passed to the fields procedures (see above).

---

printf (WHM, CW, PLT)

This module provides formatted output functions modeled on those in the C language.

printf(fmt:string, args[]) formats and writes arguments to &output.

fprintf(f:file,fmt:string,args[]) formats and writes arguments to f.

sprintf(fmt:string, args[]) : string formats arguments and produces a string result.

The format string fmt is modified by substituting arguments in place of the "format specifiers" within it, consisting of a percent sign followed by a specifier code:

code	argument printed in the form	code	argument printed in the form
%d	decimal integer	%r	real number
%e	scientific (exponential) notation	%s	string
%i	image	%x	hexadecimal
%o	octal		

Specifier code %r uses scientific notation if the integer portion is inexact. An hyphen after the percent sign indicates left justification, otherwise right justification is used. A number of digits after the percent sign may specify the minimum width of the field to use, or after a period they specify a number of digits of precision. For example, printf("%-5.2r", x) specifies that real number x be formatted as a string of at least 5 characters, left justified, with two digits after the decimal point.

---

prockind

prockind(p:procedure) : string? produces a code for the kind of the procedure p as follows: "p" (declared) procedure "f" (built-in) function, "o" operator, "c" record constructor. It fails if p is not of type procedure.

---

procname

procname(p:procedure, x) : string? produces the name of a procedure (including functions, operators, and record constructors) value. If x is null, the result is derived from image() in a relatively straightforward way. In the case of operators, the number of arguments is appended to the operator symbol. If x is nonnull, the result is put in a form that resembles an Icon expression. procname() fails if p is not of type procedure.

---

pscript (GMT)

epsheader(f, x, y, w, h, flags) writes an Encapsulated PostScript file header and initializes the PostScript coordinate system. This procedure is for writing PostScript output explicitly. It is the caller's responsibility to ensure that the rest of the file conforms to the requirements for EPS files as documented in the PostScript Reference Manual, second edition. (x,y,w,h) specify the range of coordinates that are to be used in the generated PostScript code. epsheader() generates PostScript commands that center this region on the page and clip anything outside it. If the flags string contains the letter "r" and abs(w) > abs(h), the coordinate system is rotated to place the region in "landscape" mode. The generated header also defines an "inch" operator that can be used for absolute measurements as shown in the example below.

```
f := open(filename, "w") | stop("can't open ",filename)
epsheader(f, x, y, w, h)
write(f, ".07 inch setlinewidth")
write(f,x1, " ", y1, " moveto ", x2, " ", y2, " lineto stroke") ... write(f, "showpage")
```

See also: psrecord.icn contains procedures that write PostScript as a side effect of normal graphics calls.

---

random (REG, GMT)

This file contains procedures related to pseudo-random numbers.

rand_num() : integer is a linear congruential pseudo-random number generator. Each call produces another number in the sequence and assigns it to the global variable random. With no arguments, rand_num() produces the same sequence(s) as the built-in random-number generator. Arguments can be used to get different sequences. The global variable random plays the same role that &random does for the built-in random number generator.

rand_int(i) : integer produces a random integer in the range 1 to i.

randomize() sets &random, based on the date and time of day.

randrange(min, max) : integer produces a random number from min <= i <= max.

randrangeseq(i, j) : integer* generates the integers from i to j in random order.

randseq(seed) : integer* generates the values of &random, starting at seed, that occur as the result of using ?x.

shuffle(x):x shuffles the elements of string, list, or record x. If x is a list or record it is altered in place instead of creating a new structure for the shuffled result.

Links: factors.

---

rational

These procedures perform arithmetic on rational numbers (fractions):

record rational(numer, denom, sign) is used to represent rational values.

str2rat(string):rational? converts a string such as "3/2" to a rational number.

rat2str(r:rational):string converts rational number r to its string representation.

addrat(r1:rational,r2:rational) : rational adds rational numbers r1 and r2.

subrat(r1:rational,r2:rational) : rational subtracts rational numbers r1 and r2.

mpyrat(r1:rational,r2:rational) : rational multiplies rational numbers r1 and r2.

divrat(r1:rational,r2:rational) : rational divides rational number r1 by r2.

negrat(r) : rational produces the negative of rational number r.

reciprat(r:rational) : rational produces the reciprocal of rational number r.

Links: numbers.

---

readtbl (RLG)

readtbl(f:file) : table reads SGML mapping information from a file. This module is part of the strpsgml package. The file specifies how each SGML tag in a given input text should be translated. Each line has the form:

SGML-designator start_code end_code

where the SGML designator is something like "quote" (without the quotation marks), and the start and end codes are the way in which you want the beginning and end of a <quote>...</quote> sequence to be translated. Presumably, in this instance, your codes would indicate some set level of indentation, and perhaps a font change. If you don't have an end code for a particular SGML designator, just leave it blank. Links: stripunb.

---

rec2tab

rec2tab(x) : null writes fields of a record as tab-separated string. Carriage returns in files are converted to vertical tabs. (Works for lists too.)

---

records

field(R, i) : string? returns the name of the ith field of R. Other record processing procedures may be added to this module in future editions.

---

recurmap

recurmap(recur:list) : string maps a recurrence declaration of the form

```
f(i):
if expr11 then expr12
if expr21 then expr22
    ...
```

else expr

The declaration is a list of strings. The result string is a declaration for an Icon procedure that computes corresponding values. At present there is no error checking and the most naive form of code is generated.

---

reduce

reduce(op, init, args[]) applies the binary operation op to the values in args, using init as the initial value. For example, reduce("+", 1, ...) produces the sum of the values in ...

---

regexp (RJA)

This module implements UNIX-like regular expression patterns. String scanning function default conventions are followed.

ReMatch(pattern,s,i1,i2) : integer* produces the sequence of positions in s past a substring starting at i1 that matches pattern, but fails if there is no such position.

ReFind(pattern,s,i1,i2) : integer* produces the sequence of positions in s where substrings begin that match pattern, but fails if there is no such position. Each position is produced only once. pattern can be either a string or a pattern list -- see RePat().

RePat(s) : list? creates a "pattern element list" from pattern string s, but fails if the pattern string is not syntactically correct. ReMatch() and ReFind() will automatically convert a pattern string to a pattern list, but it is faster to do the conversion explicitly if multiple operations are done using the same pattern.

ReCaseIndependent() : null, ReCaseDependent() : null set the mode for case-independent or case-dependent matching. The initial mode is case-dependent.

Accessible Global Variables After a match, the strings matched by parenthesized regular expressions are left in list Re_ParenGroups, and can be accessed by subscript.

Regular Expression Characters and Features Supported The regular expression format supported by procedures in this file model very closely those supported by the UNIX "egrep" program, with modifications as described in the Perl programming language definition. Following is a brief description of the special characters used in regular expressions. The abbreviation RE means regular expression.

c An ordinary character (not one of the special characters discussed below) is a one-character RE that matches that character.

\c A backslash followed by any special character is a one-character RE that matches the special character itself.

. A period is a one-character RE that matches any character.

[string] A non-empty string enclosed in square brackets matches any one character in that string. If the first character is "^" (circumflex), any character *not* in the remaining characters of the string is matched. The "-" (minus), when between two other characters,

indicates a range of consecutive ASCII characters (e.g. [0-9] is equivalent to [0123456789]). Other special characters stand for themselves in a bracketed string.

* Matches zero or more occurrences of the RE to its left.

+ Matches one or more occurrences of the RE to its left.

? Matches zero or one occurrences of the RE to its left.

{N} Matches exactly N occurrences of the RE to its left.

{N,} Matches at least N occurrences of the RE to its left.

{N,M} Matches from N to M occurrences of the RE to its left.

ˆ A caret at the beginning of an entire RE constrains that RE to match an initial substring of the subject string.

$ A currency symbol at the end of an entire RE constrains that RE to match a final substring of the subject string.

| Alternation: two regular expressions separated by "|" match either a match for the first or a match for the second.

() A RE enclosed in parentheses matches a match for the regular expression (parenthesized groups are used for grouping, and for accessing the matched string subsequently in the match using the \N expression).

\N where N is a digit in the range 1-9, matches a string of characters that was matched by a parenthesized RE to the left in the same RE. The sub-expression specified is that beginning with the Nth occurrence of "(" counting from the left. e.g., ˆ(.*)\1. matches a string consisting of two consecutive occurrences of the same string.

Extensions beyond UNIX egrep The following extensions to UNIX regular expressions, as specified in the Perl programming language, are supported.

\w matches any alphanumeric (including "_").

\W Matches any non-alphanumeric.

\b matches at a word-boundary (a word is a string of alphanumerics, as in \w).

\B matches only non-word-boundaries.

\s matches any white-space character.

\S matches any non-white-space character.

\d matches any digit [0-9].

\D matches any non-digit.

\w, \W, \s, \S, \d, \D can be used within [string] regular expressions.

---

repetit

repetit(L:list) : integer returns the length of the smallest range of values that repeat in a list. For example, if L := [1, 2, 3, 1, 2, 3, 1, 2, 3] , repetit(L) returns 3. If there is no repetition, repetit() returns the length of the list.

---

rewrap (RLG)

rewrap(s:string,i:70) : string? reformats text fed to it into strings <i in length. rewrap() utilizes a static buffer, so it can be called repeatedly with different string arguments, and still produce homogeneous output. This buffer is flushed by calling rewrap() with a null first argument. Here's a simple example of how rewrap() could be used. The following program reads the standard input, producing fully rewrapped output.

```
procedure main()
    every write(rewrap(!&input))
    write(rewrap())
end
```

Naturally, in practice you would want to do things like check for indentation or blank lines in order to wrap only on a paragraph by paragraph basis.

Note: If you want leading and trailing tabs removed, map them to spaces first. rewrap() only fools with spaces, leaving tabs intact. This can be changed easily enough, by running its input through the detab() function. See also: wrap.icn.

---

scan (RLG, DAG, REG, RLS, CW)

This module contains procedures related to string scanning: Where indicated by parameter names, they follow the string scanning conventions for parameter defaults.

balq(c1,c2,c3,c4:'\'"',c5:'\',s,i1,i2):integer* generates integer positions in s preceding a character of c1 in s[i1:i2] that are (a) balanced with respect to characters in c2 and c3 and (b) not "quoted" by characters in c4 with "escape" sequences in c5.

balqc(c1,c2,c3,c4:'\'"',c5,s1:"/*",s2:"*/",s3,i1,i2):integer* is like balq() with the addition that balanced characters within "comments", as delimited by the strings s1 and s2, are also excluded from balancing. In addition, if s1 is given and s2 is null then the comment terminates at the end of string.

limatch(L:list, c:cset) : integer? matches items in L delimited by characters in c. Returns the last cursor position scanned to, or fails

slashbal(c1,c2,c3,s,i,j) : integer* behaves like bal(), except that it ignores, for purposes of balancing, any c2 or c3 char which is preceded by a backslash.

slashupto(c, s, i, j) : integer? works just like upto(), except that it ignores backslash escaped characters.

snapshot(title:string, len:integer) writes a snapshot of the state of string scanning, showing the value of &subject and &pos. Two optional arguments specify a title written at the top of the snapshot, and a width (in characters) at which to wrap output.

A bar showing &pos is positioned under the &pos'th character (actual positions are between characters). If &pos is at the end of &subject, the bar is positioned under the quotation mark delimiting the subject. Escape sequences are handled properly.

---

scanset

scan_setup(s, i1, i2) : scan_setup_result? sets things up for user-written string-scanning procedures that are in the spirit of the built-ins. The values passed are the last three arguments to all scanning functions (such as upto(c,s,i1,i2)). scan_setup() supplies any appropriate defaults and returns needed values. The value returned is a record scan_setup_result(ss, offset) where ss is the substring of s to be scanned, and offset is the size of the substring of s that precedes the substring to be scanned. scan_setup() fails if i1 or i2 is out of range with respect to s. The user-written procedure can then match in the string ss to compute the position within ss appropriate to the scan (p). The value returned (or suspended) to the caller is p + offset (the position within the original string, s). For example, the following function finds two words separated by spaces:

```
procedure two_words(s,i1,i2)
   local p, x := scan_setup(s,i1,i2) | fail # fail if out of range
   x.ss ? suspend {
     tab(upto(&letters)) &
     pos(1) | (move(-1) & tab(any(~&letters))) &
     p := &pos & # remember starting position
     tab(many(&letters)) & tab(many(' ')) &
     tab(many(&letters)) &
     p + x.offset # return position in original s
     }
end
```

---

segment (WHM)

These procedures segment a string s into consecutive substrings consisting of characters that respectively do/do not occur in c.

segment(s,c) : string* generates the substrings, while seglist(s,c) : list produces a list of the segments. For example, segment("Not a sentence.",&letters) generates six string results: "Not" " " "a" " " "sentence" "." while seglist("Not a sentence.",&letters) produces a list of size six: ["Not"," ","a","sentence","."]

---

sentence, senten1 (RLG, PAB)

Two alternative modules provide a function sentence(f) :string* that generates sentences from file f. Many grammatical and stylistic analysis programs are predicated on the notion of a sentence. Some programs count the number of words in each sentence; others count the number and length of clauses. Still others pedantically check for sentence-final particles and prepositions. Neither module's definition of a sentence will handle all possible inputs properly; you may wish to try both of them to see which one works better on your inputs.

Module sentence requires co-expressions, while module senten1 does not. Module senten1 uses a definition of a "sentence" detailed in the source code.

---

seqimage

Seqimage{e,i,j}: string produces a string image of the result sequence for the expression e. The first i results are printed. If i is omitted, there is no limit. If there are more than i results for e, ellipses are provided in the image after the first i. If j is specified, at most j results from the end of the sequence are printed after the ellipses. If j is omitted, only the first i results are produced. For example, the expressions

```
Seqimage{1 to 12}
Seqimage{1 to 12,10}
Seqimage{1 to 12,6,3}
```

produce, respectively,

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...}
{1, 2, 3, 4, 5, 6, ..., 10, 11, 12}
```

If j is given and e has an infinite result sequence, Seqimage{} does not terminate.

---

sername

sername(p:"file", s:"", n:3, i:0) : string produces a series of names of the form p<nnn>s. If n is given it determines the number of digits in <nnn>. If i is given it resets the sequence to start with i. <nnn> is a right-adjusted integer padded with zeros. Ordinarily, the arguments only are given on the first call. Subsequent calls without arguments give the next name. For example, sername("image", ".gif", 3, 0) produces "image000.gif", and subsequently, sername() produces "image001.gif", "image002.gif", and so on. If any argument changes on subsequent calls, all non-null arguments are reset.

---

sets (AB, REG)

cset2set(c:cset): set returns a set that contains the individual characters in c.

domain(T:table) returns the domain (set of keys) of the function defined by T.

inverse(T:table, x) : table returns the inverse of the function defined by T. x values specify: &null (functional inverse), empty list (relational inverse), empty set: (relational inverse, but with each table member as a set instead of a list).

pairset(T:table) : set converts T to an equivalent set of ordered pairs.

range(T:table):set returns the range (set of values) of the function defined by T.

seteq(S1, S2) : set? tests equivalence of sets S1 and S2.

setlt(S1, S2) : set? tests inclusion (strict subset) of set S1 in S2.

---

showtbl

showtbl(title:"", T, mode, limit, order, posit, w1:10, w2:10, gutter:3, f1:left, f2:right) displays table T according to the arguments given. The remaining arguments are:

mode indicates the type of sorting, one of: "ref"|"val" (by key or decreasing value)

limit specifies the maximum lines of table output, if any
order gives the sort order, one of: "incr"|"decr" (not implemented yet)
posit is the first column position, one of: "ref"|"val" (not implemented yet)
w1 is the width of 1st column
w2 is the width of 2nd column
gutter is the width between columns
f1 supplies the formatting function used on the 1st column
f2 supplies the formatting function used on the 2nd column

showtbl() returns a record with the first element being a count of the size of the table and the second element the number of lines written.

---

shquote (RJA)

This module is useful for writing programs that generate shell commands. Certain characters cannot appear in the open in strings that are to be interpreted as "words" by command shells. This family of procedures assists in quoting such strings so that they will be interpreted as single words. Quoting characters are applied only if necessary -- if strings need no quoting they are returned unchanged.

shquote(s1, s2,..., sN) : string produces a string of words s1, s2, ..., sN that are properly separated and quoted for the Bourne Shell (sh).

cshquote(s1, s2,..., sN) : string produces a string of words s1, s2,..., sN that are properly separated and quoted for the C-Shell (csh).

mpwquote(s1, s2,..., sN) : string produces a string of words s1, s2,..., sN that are properly separated and quoted for the Macintosh Programmer's Workshop shell.

dequote(s1,s2:"\\") : string produces the UNIX-style command line word s1 with any quoting characters removed. s2 is the escape character required by the shell.

---

signed (RJA)

signed(s) : integer puts raw bits of characters of string s into an integer. The value is taken as signed. This procedure is used for processing of binary data read from a file.

---

sort (RJA,RLG,REG)

sortff(L, fields[]) is like sortf(), except it takes an unlimited number of field arguments. sortgen(T, m) : any* generates sorted output in a manner given by m:

"k+"	sort by key in ascending order	"k-"	sort by key in descending order
"v+"	sort by value in ascending order	"v-"	sort by value in descending order

sortt(T, i) is like sort(T, i) but produces a list of two-element records instead of a list of two-element lists.

---

soundex, soundex1 (CW, JDS)

soundex(name:string) : string produces a code for a name that tends to bring together variant spellings from Knuth, The Art of Computer Programming, Vol.3. Module soundex1 employs an approach proposed by M. K. Odell and R. C. Russell.

---

statemap

statemap() : table produces a "two-way" table to map state names (in the postal sense) to their postal abbreviations and vice-versa.

---

str2toks (RLG)

Scanning procedure str2toks(c:~(&letters++&digits), s, i1, i2) : string* suspends portions of s[i1:i2] delimited by characters in c. str2toks() is not a primitive scanning function in the sense that it suspends strings, and not integer positions. The code:

```
"hello, how are ya?" ? every write(str2toks())
```

writes to &output, on successive lines, the words "hello", "how", "are", and finally "ya" (skipping the punctuation). The beginning and end of the line count as delimiters. Note that if i > 1 or j < *s+1 some tokens may end up appearing truncated.

---

strings

These procedures perform operations on strings.

cat(s1, s2,...) : string concatenates an arbitrary number of strings.

charcnt(s, c) : integer returns the number of instances of characters in c in s.

collate(s1, s2) : string collates the characters of s1 and s2. For example,
collate("abc", "def") produces "adbecf".

comb(s, i) : string* generates combinations of characters of s taken i at a time.

compress(s, c:&cset) : string collapses consecutive occurrences of members of c in s.

csort(s) : string produces the characters of s in lexical order.

decollate(s, i:1) : string produces a string consisting of every other
character of s. If i is odd, the odd-numbered characters are selected, while if
i is even, the even-numbered characters are selected.

deletec(s, c) : string deletes occurrences of characters in c from s.

deletep(s, L) : string deletes all characters at positions specified in L.

deletes(s1, s2) : string deletes occurrences of s2 in s1.

diffcnt(s) : integer returns count of the number of different characters in s.

extend(s, n) : string replicates s to length n.

interleave(s1, s2) : string interleaves characters s2 extended to the length of s1
with s1. ispal(s) : string? succeeds and returns s if s is a palindrome

maxlen(L, p:proc("*",1)) : integer returns the length of the longest
string in L. p is applied to each string as a "length" procedure.

meander(s, n) : string produces a "meandering" string that contains all
n-tuples of characters of s.

minlen(L, p: proc("*", 1)) : integer returns the length of the shortest
string in L. p is applied to each string as a "length" procedure.

ochars(s) : string produces unique characters in the order that they appear in s.

palins(s, n) : string* generates the n-character palindromes from characters in s.

permute(s) : string* generates all the permutations of the string s.

pretrim(s, c:' ') : string trims characters from beginning of s.

reflect(s1, i, s2:"") : string returns s1 concatenated with s2 and the reversal of s1 to produce a partial palindrome. The values of i determine end conditions for the reversal: 0 = omit first and last characters, 1 = omit first character, 2 = omit last character, 3 = don't omit character.

replace(s1, s2, s3) : string replaces all occurrences of s2 in s1 by s3.

replacem(s,...) : string performs multiple replacements in the style of replace(), where multiple argument pairs may be given, as in replacem(s, "a", "bc", "d", "cd") which replaces all a's by "bc" and all d's by "cd". Replacements are performed sequentially, not in parallel.

replc(s, L) : string replicates characters of s by amounts given by the values in L.

rotate(s, i:1) : string rotates s i characters left (negative i rotates to the right).

schars(s) : string produces the unique characters of s in lexical order.

scramble(s) : string scrambles (shuffles) the characters of s randomly.

selectp(s, L) : string selects characters of s that are at positions given in L.

transpose(s1, s2, s3) : string transposes s1 using label s2 and transposition s3.

---

stripcom (RLG)

stripcom(s) : string? strips comments from a line of code. Fails on lines which, either stripped or otherwise, come out as an empty string. stripcom() can't handle lines ending in an underscore as part of a broken string literal, since stripcom() is not intended to be used on sequentially-read files. It just removes comments from individual lines.

---

stripunb (RLG)

stripunb(c1,c2,s,i,j,t:table) : string strips material from a line which is unbalanced with respect to the characters defined in arguments 1 and 2 (unbalanced being defined as bal() defines it, except that characters preceded by a backslash are counted as regular characters, and are not taken into account by the balancing algorithm). If you call stripunb() with a table argument as follows, stripunb('<','>',s,&null,&null,t) and if t is a table having the form

```
key: "bold" value: outstr("\e[2m", "\e1m")
key: "underline" value: outstr("\e[4m", "\e1m") etc.
```

then every instance of "<bold>" in string s will be mapped to "\e2m," and every instance of "</bold>" will be mapped to "\e[1m." Values in table t must be records of type outstr(on, off). When "</>" is encountered, stripunb() will output the .off value for the preceding .on string encountered. Links: scan.

---

tab2list, tab2rec

tab2list(string):list takes tab-separated strings and inserts them into a list. Vertical tabs in strings are converted to carriage returns.

tab2rec(s, x) : null takes tab-separated strings and assigns them into fields of

a record or list x. Vertical tabs in strings are converted to carriage returns.

See also: list2tab.icn, rec2tab.icn.

---

tables (REG, AB)

keylist(T) : list produces a list of keys in table T.

kvallist(T) : list produces values in T ordered by sorted order of keys.

tbleq(T1, T2) : table? tests the equivalence of tables T1 and T2.

tblunion(T1, T2) : table approximates T1 ++ T2.

tblinter(T1, T2) : table approximates T1 ** T2.

tbldiff(T1, T2) : table approximates T1 -- T2.

tblinvrt(T) : table produces a table with the keys and values of T swapped.

tbldflt(T) : any produces the default value for T.

twt(T) : table produces a two-way table based on T.

vallist(T) : list produces list of values in table T.

For the operations on tables that mimic set operations, the correspondences are only approximate and do not have the mathematical properties of the corresponding operations on sets. For example, table "union" is not symmetric or transitive. Where there is potential asymmetry, the procedures "favor" their first argument. The procedures that return tables return new tables and do not modify their arguments.

---

tclass

tclass(x) : string returns "atomic" or "composite" depending on the type of x.

---

title, titleset

title(name:string) : string produces the title of a name, such as "Mr." from "Mr. John Doe". The process is imperfect. Links titleset.

titleset() : set produces a set of strings that commonly appear as titles in names. This set is (necessarily) incomplete.

---

trees

depth(t) : integer compute maximum depth of tree t

ldag(s) : list construct a DAG from the string s

ltree(s) : list construct a tree from the string s

stree(t) : string construct a string from the tree t

tcopy(t) : list deep copy tree t.

teq(t1,t2) : list? compare trees t1 and t2

visit(t) : any* visit, in preorder, the nodes of the tree t by suspending them all

This module provides tree operations using a list representation of trees and directed acyclic graphs (DAGs). The procedures do not protect themselves from cycles.

---

tuple (WHM)

tuple(tl:list) : list implements a "tuple" feature that produces the effect of multiple keys. A tuple is created by an expression of the form tuple([exrp1, expr2,..., exprn]). The result can be used in a case expression or as a table subscript. Lookup is successful provided the

values of expr1, expr2,..., exprn are the same (even if the lists containing them are not). For example, consider selecting an operation based on the types of two operands. The following expression uses tuple() to drive a case expression using value pairs.

```
case tuple([type(op1), type(op2)]) of {
    tuple(["integer", "integer"]): op1 + op2
    tuple(["string", "integer"]) : op1 || "+" || op2
    tuple(["integer", "string"]) : op1 || "+" || op2
    tuple(["string", "string"]) : op1 || "+" || op2
    }
```

---

typecode

typecode(x) : string produces a one-letter string identifying the type of its argument. In most cases, the code is the first (lowercase) letter of the type, as in "i" for the integer type. Structure types are in uppercase, as in "L" for the list type. All records have the code "R". The code "C" is used for the co-expression type to avoid conflict for the "c" for the cset type. The code "w" is produced for windows.

---

unsigned (RJA)

unsigned(s) : integer puts raw bits of characters of string s into an integer. The value is taken as unsigned. This procedure is used for processing of binary data read from a file.

---

usage

These procedures provide various common services:

Usage(s) stops execution with a message indicating how to use a program.

Error(args?) writes arguments to &errout on a single line, preceded by "*** ".

ErrorCheck(l,f) : null reports to &output an error that was converted to failure.

Feature(s) : ? succeeds if feature s is available in this implementation.

Requires(s) terminates execution is feature s is not available.

Signature() writes the version, host, and features support in the running implementation of Icon.

---

varsub

varsub(s, varProc:getenv) obtains a variable value from the procedure, varProc. As with the UNIX Bourne shell and C shell, variable names are preceded by $. Optionally, the variable name can additionally be surrounded by curly braces {}, which is usually done when necessary to isolate the variable name from surrounding text. As with the C-shell, the special symbol ~<username> is handled. Username can be omitted; in which case the value of the variable HOME is substituted. If username is supplied, the /etc/passwd file is searched to supply the home directory of username (this action is obviously not portable to non-UNIX environments). (RJA)

---

version

version() : string? produces the version number of Icon on which a program is running. It only works if the &version is in the standard form.

---

vrml, vrml1lib, vrml2lib

These modules contain procedures for producing VRML files.

point_field(L) create VRML point field from point list L

u_crd_idx(i) create VRML coordinate index for 0 through i - 1

render(x) render node x

vrml1(x) produces VRML 1.0 file for node x

vrml2(x) produces VRML 2.0 file for node x

vrml_color(s) converts Icon color specification to vrml form

Not all node types have been tested. Where field values are complex, as in vectors, these must be built separately as strings to go in the appropriate fields. There is no error checking. Fields must be given in the order they appear in the node record declarations and field values must be of the correct type and form.

vrml1lib.icn contains declarations for VRML 1.0 nodes; vrml2lib.icn is for VRML 2.0 nodes. Although VRML 1.0 and 2.0 allow node fields to be given in any order, these modules require that they be specified in the order given in the record declarations. In VRML 1.0, omitted (null-valued) fields are ignored on output; group nodes require list arguments for lists of nodes. In VRML 2.0, group nodes require list arguments for lists of nodes. Links: records. Requires: graphics facilities, for color conversion.

---

wdiag

wdiag(s1, s2,...) : null writes the values of global variables s1, s2, ... with s1, s2, ... as identifying labels. It writes a diagnostic message to standard error if an argument is not the name of a global variable.

---

weighted (EE)

WeightedShuffle(sample, percentage) : list returns the list sample with a portion of the elements switched. Examples:

WeightedShuffle(X, 100) - returns a fully shuffled list

WeightedShuffle(X, 50) - every other element is eligible to be switched

WeightedShuffle(X, 25) - every fourth element is shuffled

WeightedShuffle(X, 0) - nothing is changed

The procedure will fail if the given percentage is not between 0 and 100, inclusive, or if it is not a numeric value.

---

wildcard (RJA)

These procedures deal with UNIX-like filename wild-card patterns containing *, ?, and [...], as found in the UNIX shells csh and sh; they are described briefly in the wild_pat() procedure. Recursive suspension is used to simulate conjunction of an arbitrary number of computed expressions. Default values of s, i1, and i2 are the same as for Icon's built-in string scanning procedures such as match(). The public procedures are:

wild_match(pattern,s,i1,i2) : integer* produces the sequence of positions in s past a substring starting at i1 that matches pattern, but fails if there is no such position. Similar to match(), but is capable of generating multiple positions.

wild_find(pattern,s,i1,i2) : integer* produces the sequence of positions in s where substrings begin that match pattern, but fails if there is no such position. Similar to find(). pattern can be either a string or a pattern list, see wild_pat(), below.

wild_pat(s) : L creates a pattern element list from pattern string s. A pattern element is needed by wild_match() and wild_find(). wild_match() and wild_find() will automatically convert a pattern string to a pattern list, but it is faster to do the conversion explicitly if multiple operations are done using the same pattern.

---

word (RJA)

String scanning function word(s, i1, i2) : integer? produces the position past a UNIX-style command line word, including quoted and escaped characters.

word_dequote(s) : string produces the UNIX-style command line word s with any quoting characters removed. Links: scanset.

---

wrap (RJA)

wrap(s:"",i:0) : string? accumulates small strings into longer ones, writing when the accumulated string would exceed a specified length. s is the string to accumulate, i is the width of desired output string. wrap() fails if the string s did not necessitate output of the buffered output string; otherwise the output string is returned (which never includes s). Calling wrap() with no arguments produces the buffer (if it is not empty) and clears it. wrap() does no output to files. Here's how wrap() is normally used:

```
wrap() # Initialize; not needed unless there was a previous use.
every i := 1 to 100 do # Loop to process strings to output --
    write(wrap(x[i],80)) # only writes when 80-char line filled.
write(wrap()) # Output what's in buffer, if something to write.
```

wraps(s,i) : string? is similar to wrap(), but intended for use with writes(). If the string s did not necessitate a line-wrap, s is returned. If a line-wrap is needed, s, preceded by a new-line character ("\n"), is returned.

---

xcodes (RJA, REG)

The xcodes module provides procedures to save and restore structures to disk. Records are encoded using canonical names: record0, record1, ... This allows programs to decode files by providing declarations for these names when the original declarations are not available. xcodes also provides for procedures and files present in the encoded file that are not in the decoding program.

xencode(x,f) stores x in file f such that it can be converted back to x by xdecode(f). The procedures handle most kinds of values, including structures of arbitrary complexity including cycles. The following sequence will output x and recreate it as y:

```
f := open("xstore","w")
xencode(x,f)
close(f)
```

```
f := open("xstore")
y := xdecode(f)
close(f)
```

For scalar types (null, integer, real, cset, and string), this sequence results in the relationship x === y. For structured types (list, set, table, record and object), y has the same shape as x and its elements bear the same relation to the original as if they were encoded and decoded individually. Files, co-expressions, and windows are not handled; they decode as empty lists, with the exception of the special files &input, &output, and &errout. Functions and procedures only preserve type and identification. xdecode() fails if given a file that is not in xcode format or if the encoded file contains a record for which there is no declaration in the program in which the decoding is done. If a record is declared differently in the encoding and decoding programs, the decoding may be bogus.

xencoden() and xdecoden() perform the same operations, except they take the name of a file to open, not an already-open file.

xencodet() and xdecodet() are like xencode() and xdecode() except that the trailing argument is a type name. If the encoded decoded value is not of that type, they fail. xencodet() does not take an opt argument.

xencode(x, f:&output, p:write) : f encodes x writing to file f using procedure p. p uses the same interface as write() (the first parameter is always the value passed as f).

xdecode(f:&input, p:read) : x returns the restored object where f is the file to read, and p is a procedure that reads a line from f using the same interface as read().

If p is provided to xencode(), then f can be any arbitrary data object -- it need not be a file. For example, to "write" x to a string:

```
record StringFile(s)
procedure main() ...
    encodeString := xencode(x,StringFile(""),WriteString).s
    ...
end
procedure WriteString(f,s[])
    (every f.s ||:= !s); f.s ||:= "\n"
    return
end
```

Links: escape. See also: object.icn, codeobj.icn.

---

xforms

These procedures produce matrices for affine transformation in two dimensions and transform point lists. A point list is a list of Point() records.

transform(p:list, M) : list transforms a point list by matrix

transform_points(pl:list,M) transforms point list

set_scale(x, y) : matrix produces a matrix for scaling

set_trans(x, y) : matrix produces a matrix for translation
set_xshear(x) : matrix produces a matrix for x shear
set_yshear(y) : matrix produces a matrix for y shear
set_rotate(x) : matrix produces a matrix for rotation

Links: gobject. See also: matrix. (SBW, REG)

---

ximage (RJA)

ximage(x) : s produces a string image of x. ximage() differs from image() in that it outputs all elements of structured data types. The output resembles Icon code and is thus familiar to Icon programmers. Additionally, it indents successive structural levels in such a way that it is easy to visualize the data's structure. Note that the additional arguments in the ximage() procedure declaration are used for passing data among recursive levels.

xdump(x1,x2,...,xn) : xn uses ximage() to write x1, x2, ..., xn to &errout.

---

xrotate

xrotate(X, i) rotates the values in X right by one position. It works for lists and records. This procedure is mainly interesting as a recursive version of

```
x1 :=: x2 :=: x3 :=: ... xn
```

since a better method for lists is push(L, pull(L)).

B.2 Application Programs, Examples, and Tools

The Icon Program Library progs directory contains 200+ programs that are useful for demonstration, entertainment, and/or practical utility. Highlights of usage information is presented with [optional] and repeated* command-line arguments.

---

adlcheck, adlcount, adlfilter, adlfirst, adllist, adlsort

This suite of address-list tools works on "address list files" and includes a correctness checker, a counter, a filterer, a lister, and a sorter. Read the source code for a description of these programs' many command line options. See also: address.doc, labels.icn.

---

animal (RJA)

animal is the familiar "animal game". The program asks the user a series of questions in an attempt to guess what animal he or she is thinking of. This expert system gets smarter as it plays. Typing "list" at any yes/no prompt will show an inventory of animals known, and there are some other commands too.

---

banner (CT)

banner is a utility that outputs enlarged letters (5x6 matrix) in portrait mode.

---

bj (CT, RLG)

bj is a simple but fun blackjack game. The original version was for an ANSI screen. This version has been modified to work with the UNIX termcap database file.

---

blnk2tab

blnk2tab is a UNIX-style filter that converts strings of two or more blanks to tabs.

---

c2icn (RJA)

The c2icn filter does easy work involved in porting a C program to Icon. It reformats comments, moves embedded comments to end of line, removes ";" from ends of lines, reformats line-continued strings, changes = to :=, and changes -> to .

---

calc

calc is a simple Polish "desk calculator". It accepts as values integers, reals, csets, and strings, as well as an empty line for the null value. Other lines of input are operations, either binary operators, functions, or one of: clear (remove all values from the calculator's stack), dump (write out the contents of the stack), or quit (exits from the calculator). Failure and most errors are detected; arguments are consumed and not left on the stack.

---

chkhtml (RJA)

This program checks an HTML file and detects the following errors: Reference to undefined anchor name; duplicated anchor name; warning for unreferenced anchor name; unknown tag; badly formed tag; improper tag nesting; unescaped <, >, ", or &; bad escape string; improper embedding of attributes; and bad (non-ASCII) characters. The program also advises on the use of <HTML>, <HEAD>, and <BODY> tags.

---

colm [-w linewidth] [-s space] [-m min_width] [-t tab_width] [-x] [-d] file* (RJA)

colm arranges a number of data items, one per line, into multiple columns. Items are arranged in column-wise order, that is, the sequence runs down the first column, then down the second, etc. If a null line appears in the input stream, it signifies a break in the list, and the following line is taken as a title for the following data items. No title precedes the initial sequence of items. See the program source code for command line options interpretation.

---

comfiles

comfiles lists common file names in two directories given as command-line arguments. Requires: UNIX

---

concord

concord produces a simple concordance from standard input to standard output. Words less than three characters long are ignored. Option -l n sets maximum line length to n (default 72), after which concord starts a new line. Option -w n sets the maximum width for word to n (default 15), after which words are truncated.

---

conman (WED)

conman responds to queries like "? Volume of the earth in tbsp". The commands (which are not reserved) are: load, save, print, ? (same as print), list, is, are (same as is).

---

countlst

countlst counts how many times each line of input occurs and writes a summary. By default, the output is sorted first by decreasing count and within each count, alphabetically. Option -a causes output to be sorted alphabetically; -t prints a total at the end.

---

cross (WPM)

cross takes a list of words and tries to arrange them in crossword format so that they intersect. Uppercase letters are mapped into lowercase letters on input. The program objects if the input contains a nonalphabetic character. It produces only one intersection.

---

crypt [key] <infile >outfile (PB, PLT)

crypt is an example encryption program. Do *not* use this in the face of competent cryptanalysis. Helen Gaines' book (Gaines, 1939) is a classical introduction to the field.

---

csgen

csgen takes a context-sensitive production grammar and generates random sentences from the corresponding language. Uppercase letters are nonterminal symbols and -> indicates the left-hand side can be rewritten by the right-hand side. Other characters are considered to be terminal symbols. Lines beginning with # are ignored. A line with a nonterminal symbol S followed by a colon and an integer i specifies generation of i sentences for the language with start symbol S. See the source code for details and explanation.

---

cstrings (RJA)

cstrings prints all strings (enclosed in double quotes) in C source files.

---

daystil (NL)

daystil calculates the number of days between the current date and the date given on the command line, which may be specified in many ways. For example, August 12 can be specified as "August 12", "Aug 12", "12 August", or "12 aUGuS", among others.

---

deal

deal [-h n:1] shuffles, deals, and displays n hands in the game of bridge.

---

declchck

declchck examines ucode files and reports declared identifiers that may conflict with function names. Requires: UNIX.

---

delamc (TRH)

delamc delaminates standard input into several output files according to the separator characters specified by the string following the -t option (default: tab). All output files contain the same number of lines as the input file.

---

detex (CLJ)

detex reads in documents written in the LaTeX typesetting language, and removes some of the common LaTeX commands to produce plain ASCII. Output must typically be further edited by hand to produce an acceptable result.

---

diffn file* (RJA)

diffn shows the differences between n files.

---

diffsum [file] (GMT)

diffsum reads output from the Unix diff(1) utility, either normal diffs or context diffs. For each pair of files compared, diffsum reports two numbers: The number of lines added or changed, and the net change in file size. The first of these indicates the magnitude of the changes and the second the net effect on file size.

---

diffu f1 f2 (RM)

diffu uses dif() to generate file differences like the UNIX diff(1) command.

---

diffword

diffword lists the different words in the input. The definition of "word" is naive.

---

duplfile

duplfile lists the file names that occur in more than one subdirectory and the subdirectories in which the names occur. This program runs slow on large directory structures.

---

envelope [options] < address(es) (RF)

envelope addresses envelopes on a PostScript or HP-LJ printer, including barcodes for the zip code. A line beginning with # or an optional alternate separator can be used to separate multiple addresses. The parser will strip the formatting commands from an address in a troff or LaTeX letter. Typically, envelope is used from inside an editor. In Emacs, mark the region of the address and do M-| envelope. In vi, put the cursor on the first line of the address and do :,+N w !envelope where N = number-of-lines-in-address.

---

farb, farb2

Dave Farber, co-author of the original SNOBOL programming language, is noted for his creative use of the English language. Hence the terms "farberisms" and "to farberate". This program produces a randomly selected farberism.

---

filecnvt [-i s1] [-o s2] infile outfile (BW)

filecnvt copies a text file, converting line terminators. Option -i s1 indicates the input file line termination system (default "u"). Option -o s2 directs output to use line terminators for system s2 (default "u"). The designations are d (DOS/Windows "\n\r"), m (Macintosh "\r"), and u (UNIX "\n").

---

fileprnt

fileprnt reads the file specified as a command-line argument and writes out a representation of each character in several forms: hex, octal, decimal, symbolic, and ASCII.

---

filesect

filesect *start nlines* writes the section of the input file starting at a line number and extending some number of lines. If the specifications are out of range, the file is truncated.

---

filtskel (RJA)

filtskel is a skeleton/template for creation of filter programs. Command line options, file names, and tabbing are handled; you need only provide the filtering code.

---

findstr (RJA)

findstr is a utility filter to list character strings embedded in data files such as object files. The option -l length gives the minimum string size to be printed (default 3); option -c chars specifies a string of characters (besides the standard ASCII printable characters) to be considered part of a string. String escape sequences can be used.

---

findtext (PLT)

findtext retrieves multiline text from a database indexed by idxtext. Each section of text lines follows a line declaring the index terms. Each index line begins with "::".

---

fixpath filename oldpath newpath (GMT)

fixpath changes file paths or other strings in a binary file by modifying the file in place. Each null-terminated occurrence of oldpath is replaced by newpath. If the new path is longer than the old one, a warning is given and the old path is extended by null characters, which must be matched in the file for replacement to take place.

---

format (RJA)

format is a filter program that word-wraps a range of text. Options include full justification. All lines that have the same indentation as the first line (or same comment leading character format if -c option) are wrapped. Other lines are left as is. This program is useful in conjunction with editors that can invoke filters on a range of selected text.

---

former

former takes a single line of input and outputs it in lines no greater than the number given on the command line (default 80).

---

fract (REG, GMT)

fract produces successive rational approximations to a real number. The option -n r specifies the real number to be approximated, default .6180339887498948482. Option -l i gives the limit on number of approximations, default 100.

---

fuzz (AC)

fuzz does "fuzzy" string pattern matching. The result of matching s and t is a number between 0 and 1, based on counting matching pairs of characters in increasing substrings

of s and t. Characters may be weighted differently.

---

gcomp (WHM, REG)

gcomp produces a list of the files in the current directory that do not appear among the arguments. For example, gcomp *.c produces a list of files in the current directory that do not end in .c. Requires: UNIX.

---

genqueen (PAB)

genqueen *n* solves the non-attacking n-queens problem for (square) boards of arbitrary size. The problem consists of placing chess queens on an n-by-n grid such that no queen is in the same row, column, or diagonal as any other queen. The output is each of the solution boards; rotations not considered equal.

---

gftrace (GMT)

gftrace writes a set of procedures to standard output. Those procedures can be linked with an Icon program to enable the tracing of calls to built-in functions. See the comments in the generated code for details. The set of generated functions reflects the built-in functions of the version of Icon under which this generator is run.

---

graphdem (MH)

graphdem is a simple bar graphics package with two demonstration applications. The first displays the 4 most frequently used characters in a string; the second displays the Fibonacci numbers. Requires: ANSI terminal support.

---

grpsort (TRH)

grpsort sorts input containing "records" defined to be groups of consecutive lines. One or more repetitions of a demarcation line beginning with the separator string separate each input record. The first line of each record is used as the key. The separator string defaults to the empty string; empty lines are default demarcation lines.

---

headicon

headicon prepends a standard header to an Icon program. It does not check to see if the program already has a header. The first command-line argument is taken as the base name of the file; default "foo". The second command-line argument is taken as the author; the default is "Ralph E. Griswold" -- but you can personalize it for your own use. The new file is brought up in the vi editor. The file skeleton.icn must be accessible via dopen().

---

hebcalen,hcal4unx (ADC, RLG)

The Jewish year harmonizes the solar and lunar cycle, using the 19-year cycle of Meton (c. 432 BCE). Startup syntax is hebcalen [date], where date is a year specification of the form 5750 for a Jewish year, +1990 or 1990AD or 1990CE or -1990 or 1990BC or 1990BCE for a civil year. Requires: keyboard functions, hebcalen.dat, hebcalen.hlp.

---

hr (CT)

hr implements a horse-race game.

---

ibar

ibar replaces comment bars by bars 76 characters long, the IPL standard.

---

ibrow (RJA)

ibrow browses Icon files for declarations, defaulting to browse all *.icn files in the current directory. The user interface is self-explanatory -- use "?" for help if you're confused.

---

icalc (SBW)

icalc is an infix calculator with control structures and compound statements. Features include: integer and real value arithmetic; variables; function calls to built-in functions; strings allowed as function arguments; unary operators + (absolute value) and - (negation); assignment := ; binary operators: +, -, *, /, %, and ^; relational operators: =, !=, <, <=, >, >= (use 1 for true and 0 for false); compound statements in curly braces with semicolon separators; if-then and if-then-else; while-do; and limited multiline input. The input is processed one line at a time, in calculator fashion, but compound statements can be continued across line boundaries.

---

icalls

icalls processes trace output and tabulates calls of procedures.

---

icn2c (RJA)

icn2c partially converts Icon to C. It reformats comments, line-continued strings and procedure declarations, changes := to =, and changes end to "}".

---

icontent <options> <Icon source file>... (RJA)

icontent builds a list, in Icon comment format, of procedures and records in an Icon source file. Multiple files can be specified as arguments, and are processed in sequence. Option -s specifies to sort names alphabetically (default is in order of occurrence); option -l lists in single column (default is to list in multiple columns).

---

icvt (CW, REG)

icvt converts Icon programs from ASCII syntax to EBCDIC syntax or vice versa. The option -a converts to ASCII, while the option -e converts to EBCDIC. The program given in standard input is written in converted form to standard output.

---

idepth

idepth processes trace output and reports the maximum depth of recursion.

---

idxtext [-a] file1 [file2 [...]] (RLG, PLT)

idxtext turns a file associated with the gettext() routine into an indexed text-base. Though gettext() will work fine with files that haven't been indexed via idxtext(), access is faster if the indexing is done if the file is, say, over 5k or 10k. file1, file2, etc. are the names of gettext-format files that are to be (re-)indexed. The -a flag tells idxtext to abort if an index

file already exists. Indexed files have the format:

keyname delimiter offset [delimiter offset [etc.]]\n.

---

ifilter op

ifilter applies the operation given on the command-line to each line of standard input, writing the results. For example, ifilter reverse <foo writes the lines of foo reversed end-for-end, while ifilter right 10 0 <foo calls right(line, "10", "0") for each line of foo.

---

igrep (RJA)

igrep emulates UNIX egrep using the enhanced regular expressions supported by regexp.icn. Options are nearly identical to those supported by egrep (no -b: print disk block number). The option -E allows Icon-type (hence C-type) string escape sequences in the pattern string. Beware: when -E is used, backslashes that are meant to be processed in the regular expression context must be doubled.

---

iheader

iheader lists the headers of Icon files whose names are given on the command line.

---

ihelp [-f helpfile] [item] [keyword ...] (RJA)

ihelp displays help information. The optional item name specifies the section of the help file that is to be displayed. If no item name is specified a default section will be displayed, which usually lists the help items that are available.

---

iidecode [infile] [-x], iiencode [infile] [-x] remotefile [-o outfile] (RLG, FJL)

These are Icon ports of the UNIX/C uudecode/uuencode utilities, based on freely distributable BSD code. File modes are always encoded with 0644 permissions. An option specifies xxencoded files (like uuencoding, but passes unscathed through EBCDIC sites).

---

ilnkxref [-options] <icon source file>... (RJA)

ilnkxref is a utility to create cross-reference of library files used in Icon programs (i.e., those files named in link declarations). Requires: UNIX. Option -p specifies to sort by "popularity"; option -v tells the program to report progress information

---

ilump

ilump copies Icon source files, incorporating recursively the source code for files named by link directives. This produces a whole source program in one file, useful with certain profiling and visualization tools. If a linked file is not found in the current directory, directories specified by the LPATH environment variable are tried.

---

imagetyp

imagetyp accepts file names from standard input and writes their image type to standard output. It relies on a procedure imagetyp(s) that attempts to determine the type of image file named s. Corrupted or fake files easily fool it. Examples of some image file types were not available for testing.

---

ineeds (RJA)

ineeds determines Icon modules required by an Icon module. It expects environment variable LPATH to be set properly as for the Icon Compiler.

---

inter

inter lists lines common to two files.

---

interpe, interpp (REG, JN)

interpe is a crude but effective interpreter for Icon expressions. If the expression is a generator, all its results are produced. If option -e is given, the expression is echoed.

interpp takes expressions prefixed with line numbers. You can resequence, list, and execute lines in order. Retype a line to change it. Use "?" to list the other commands.

---

ipatch file path (GMT)

ipatch changes the path to iconx, the Icon interpreter, which is embedded in an Icon executable file under Unix. Because the headers of such files are not designed to expand, a different form of header is written to accommodate a possibly longer path.

---

ipldoc

ipldoc collects selected information from documentation headers for Icon procedure files named on the command line. Option -s skips file headers; option -f sorts the procedure list by file, instead of the default sort by procedure name.

---

iplindex [-k i:16] [-p i:12] (REG, SBW)

iplindex produces an indexed listing of the Icon Program Library. Option -k i gives the width keyword field; option -p i specifies the width of field for program name. If a file except.wrd is readable in the current directory, the words in it are used instead.

---

iplkwic (SBW, REG)

iplkwic is a specialized version of kwic.icn used for producing kwic listings for the Icon program library. This is a simple keyword-in-context (KWIC) program. The "key" words are aligned at a specified column, with the text shifted as necessary. Text shifted left is truncated at the left. Tabs and other special characters may not be handled properly.

---

iplweb [-ipl source] [dest] (JK)

iplweb generates web pages from IPL header comments. Environment variable IPL locates the Icon Program Library if -ipl is not specified; dest defaults to the current directory. iplweb generates an HTML subdirectory in dest and makes an index to gprogs, gprocs, procs, and progs directories under HTML. These directories contain an .html file for each .icn file in the referenced directory. An index to all files is also generated. The .html files contain the IPL standard comment header info inside.

---

iprint (RJA)

iprint formats Icon programs, trying to keep whole procedures on the same page. It identifies the end of a print group (e.g. procedure) by looking for the string (defaulting to

"end") at the beginning of a line. Comments and declarations prior to the procedure are grouped with the procedure. Page creases are skipped over, and form-feeds (^L) embedded in the file are handled properly. Page headings (file name, date, time, page number) are printed unless suppressed by the -h option. See the program source code for details.

---

ipsort

ipsort reads an Icon program and writes an equivalent program with the procedures reordered. Global, link, and record declarations come first and aren't reordered. The main() procedure comes next followed by the remaining procedures in alphabetical order.

---

ipsplit

ipsplit reads an Icon program and writes each procedure to a separate file. The output file names consist of the procedure name with .icn appended. If -g is specified, any global, link, and record declarations are written to that file. Otherwise they are written in the file for the procedure that immediately follows them.

---

ipxref [file] (AJA)

ipxref cross-references Icon programs. It lists the occurrences of each variable by line number. Variables are listed by procedure or separately as globals. Variables that are followed by a left parenthesis are listed with an asterisk following the name. If a file is not specified, then standard input is used. The following options change the defaults:

-c n The column width (default:4) per line number.

-l n The left margin or starting column (default: 40) of the line numbers.

-w n The column width (default: 80) of the whole output line.

Normally only alphanumerics are cross-referenced. Two options broaden it:

-q Include quoted strings.

-x Include all non-alphanumerics.

This program assumes the subject file is a valid Icon program.

---

isrcline

isrcline counts the number of lines in an Icon program that actually contain code, as opposed to being comments or blank lines.

---

istrip

istrip removes comments, empty lines, and leading spaces from an Icon program.

---

itab [options] [source-program...] (RJA)

itab entabs an Icon program, leaving quoted strings alone. The options are:

-i Input tab spacing (default 8)

-o Output tab spacing (default 8).

itab observes Icon conventions for escapes and continuations in string constants. If no source-program names are given, standard input is "itabbed" to standard output.

---

itags [-aBFtvwx] [-f tagsfile] file... (RJA)

itags creates a tags file for an Icon program. The options are:

-a append output to an existing tags file.
-B use backward searching patterns (?...?).
-F use forward searching patterns (/.../) (default).
-x produce object names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output.
-t create tags for records.
-v produce on the standard output an index of the form expected by vgrind(1). This listing contains the function name, file name, and page number (assuming 64 line pages). Since the output will be sorted into lexicographic order, it may be desired to run the output through sort -f.
-w suppress warning diagnostics.

---

itrbksum

itrbksum summarizes traceback information produced on error termination by filtering out the bulk of the procedure traceback information.

---

itrcfltr [procs]

itrcfilter filters trace output. Command-line arguments are procedure names, and only lines with those names are written. The names of procedures to pass through can be given in a "response" file as accepted by options(), as in

```
itrcfltr @names <trace_file
```

where names is a file containing the names to be passed through. The option -a lists all trace messages; it overrides any procedure names given.

---

itrcsum

itrcsum provides a summary of Icon trace output.

---

iundecl [files] (RJA, REG)

iundecl invokes icont to find undeclared variables in an Icon program. The output is in the form of a local declaration, preceded by a comment line that identifies the procedure and file name from whence it arose. Undeclared variables aren't necessarily local; any intended to be global must be removed from the generated list. The program works only if the reserved words procedure and end are the first words on their respective lines.

---

iwriter

iwriter reads &input and writes code that when compiled and executed, writes out the original input. This is handy for incorporating message text in programs.

---

knapsack (AVH)

knapsack is a filter that solves a knapsack problem - how to fill a container to capacity by inserting items of various volumes. Its input consists of a string of newline-separated volumes. Its output is a single solution. knapsack demonstrates an underlying algorithm that might be useful in a variety of real-world applications. Knapsack may be tested conveniently by supplying its standard input with any sequence of random numbers.

---

krieg (DJS)

Kriegspiel ("war game") implements a variation of chess which has the same rules and goal as ordinary chess except that neither player sees the other's moves or pieces.

---

kross

kross accepts pairs of strings on successive lines. It diagrams all the intersections of the two strings in a common character.

---

kwic, kwicprep (SBW, REG)

kwic is a simple keyword-in-context program. The "key" words are aligned in column 40, with the text shifted as necessary. Text shifted left is truncated. Tabs and other characters whose "print width" is less than one may not be handled properly. If an integer is given on the command line, it overrides the default 40. Some noise words are omitted (see "exceptions" in the program text). If a file named except.wrd is open and readable in the current directory, the words in it are used instead. kwicprep prepares information to create keyword-in-context listings of the Icon program library.

---

labels

labels produces mailing labels using coded information taken from the input file. In the input file, a line beginning with # is a label header. Subsequent lines up to the next header or end-of-file are accumulated and written out centered horizontally and vertically on label forms. Lines beginning with * are treated as comments and are ignored. See the source code for options. See also: address.doc, adl*.icn, zipsort.icn.

---

lam files (TRH)

lam laminates files named on the command line, producing a concatenation of corresponding lines from each file named. If files are different lengths, empty lines are substituted for missing lines in the shorter files. Argument -s causes string s to be inserted between the concatenated file lines. Each command line argument is placed in the output line at the point that it appears in the argument list. For example, lines from file1 and file2 can be laminated with a colon between each line from file1 and the corresponding line from file2 by lam file1 -: file2. If a file is named more than once, its lines are duplicated on the output, but if standard input is named more than once, its lines will be read alternately.

---

latexidx (DSC)

latexidx processes LaTeX idx files. Input: A latex .idx file containing the \indexentry lines. Output: \item lines sorted in order by entry value, pagerefs in sorted order.

---

lc

lc simply counts the number of lines in standard input and writes the result to standard output. Assumes UNIX-style line terminators, and uses lots of memory.

---

lindcode

lindcode reads a file of L-system specifications and build Icon code that creates a table of records containing the specifications. If the option -e is given, symbols for which there is

no definition are included in the table with themselves as replacement. See also: lindrec.icn.

---

lineseq

lineseq reads values on separate lines and strings them together on a single line. The separating strings is specified by the -s option (default: " ").

---

lisp (SBW, PLT)

lisp is a simple interpreter for pure Lisp. It takes the name of the Lisp program as a command-line argument. The syntax and semantics are based on EV-LISP, as described in L. Siklossy's "Let's Talk LISP" (Prentice-Hall, 1976). Functions that have been predefined match those described in Chapters 1-4 of the book. The language is case-insensitive.

---

literat (MH)

literat manages information concerning literature. The program uses files literat.fil, literat2.fil and adress.fil to store its data. It has a predefined structure of the items and field labels. Requires: ANSI terminal support.

---

loadmap (SBW)

loadmap produces a listing of selected symbol classes from a compiled object file. The listing is by class, and gives the name, starting address, and length of the region for each symbol. The size of the last region in a symbol class is suspect

---

longest

longest writes the (last) longest line in the input file. If the command-line option -# is given, the number of the longest line is written first.

---

lower

lower maps the names of all files in the current directory to lowercase.

---

makepuzz [-f infil] [-o outfil] [-h ht] [-w wd] [-t sec] [-r reject] [-s] [-d] (RLG)

makepuzz takes a list of words, and constructs out of them one of those square find-the-word puzzles that some people like to bend their minds over. infile is a file containing words, one to a line, and outfile is the file to write the puzzle. ht (height) and wd (width) are the basic dimensions you want to try to fit your word game into (default 20x20). If the -s argument is present, makepuzz will scramble its output.

---

missile (CT)

missile is a text-based Missile Command game. It runs on systems that support the delay() function, and uses ANSI escape sequences for screen output. To play use 7, 8, and 9 to launch a missile. 7 is leftward, 8 is straight, and 9 is right. q quits the game.

---

miu (CAC, REG)

miu generates strings from the MIU string system. The number of generations is determined by the command-line argument; default is 7. Reference: Godel, Escher, and Bach: an Eternal Golden Braid, D. R. Hofstadter, Basic Books, 1979. pp. 33-36.

---

mkpasswd (JK)

mkpasswd creates a list of randomly generated passwords consisting of eight random characters in the range [A-Z0-9]. Number of passwords to generate is the first argument.

---

monkeys (SBW, REG, AB)

The old monkeys at the typewriters anecdote... monkeys uses ngram analysis to randomly generate text in the same 'style' as the input text.

---

morse (REG, RJA)

If morse is invoked without arguments, a Morse code table is printed. If arguments are given, the Morse code conversion is printed in dots and dashes. If the first argument's first character is a dot or dash, the arguments are Morse code and converted to a string.

---

mr [recipient] [-u user] [-f spool] (RF)

With no arguments, mr reads the default UNIX mail spool. Another user, a spool file, or the recipient for outgoing mail can be given as a command line argument. Help is available with the H command.

---

newicon

newicon creates a new file with a standard Icon program header and a skeleton mail procedure. The first command-line argument is taken as the base name of the file; default "foo". The second command-line argument is taken as the author. The default is "Ralph E. Griswold"; personalize it for your own use. The same comment applies to the skeleton file mentioned below. The new file is brought up in the vi editor. The options are:

- -f overwrite an existing file
- -p produce a procedure file instead of a program
- -o provide program skeleton with options()

The files skeleton.icn, skelproc.icn, and skelopt.icn must be accessible via dopen().

---

newsrc (ADC)

newsrc takes the .newsrc file, moves active groups to the beginning, and then appends inactive groups with the numbers omitted, and writes out a new file called newnewsrc.

---

nim show|save (JN)

The game of nim is played with a pile of 15 sticks. Each player can select 1, 2, or 3 sticks from the pile on their turn. The player to pick up the last stick(s) wins. There are two versions of nim here. The default version uses an algorithm that will never lose if it gets the first turn. The second version learns from each game. To invoke the learning version, pass an argument to the program. show displays the program's game memory after each game. save writes a file called ".nimdump" in the current directory with a dump of the program's game memory when you quit. When the game is played in learn mode it will initialize its game memory from the dump. You can run this program with both show and save at the same time.

---

oldicon

oldicon updates the date line in a standard Icon program header. The old file is saved with the suffix .bak. The file then is brought up in the vi editor unless the -f option is specified. Requires: system(), vi(1), UNIX.

---

pack

pack packages a list of files named on the command line into a single file, which is written to standard output. Files are separated by a header, #########, followed by the file name. This simple scheme does not work if a file contains such a header itself, and it's problematical for files of binary data. See also: unpack.icn.

---

paginate (PA)

paginate processes a document text file, inserting form feeds at appropriate places.

---

papply

papply applies the procedure given as a command-line argument to each line of standard input, writing out the results. For example, papply reverse <foo writes out the lines of foo reversed end-for-end. There is no way to provide other arguments. Except for use with (built-in) functions, this program must be linked with procedures that are used with it.

---

parens

parens produces parenthesis-balanced strings in which the parentheses are randomly distributed. See the source code for options. This program was motivated by the need for test data for error repair schemes for block-structured programming languages. A useful extension to this program would be a way to generate other text among the parentheses.

---

pargen

pargen reads a context-free grammar and produces an Icon program that parses the corresponding language. Nonterminal symbols are enclosed in angle brackets. ::= separates the LHS and RHS of the production. Vertical bars separate alternatives. Other characters are considered to be terminal symbols. The nonterminal on the first line is the goal.

Parentheses group symbols, as in <term>::=<element>(|*<term>) Empty alternatives are allowable. The terminals <, >, (,), and | are accessible through the built-in symbols <lb>, <rb>, <lp>, <rp>, and <vb>, respectively. Two other built-in symbols, <empty> and <nl> match the empty string and a newline, respectively. Lines beginning with = are passed through unchanged, allowing Icon declarations to be placed in the parser. Lines beginning with a # are ignored. If the name of a ucode file is given on the command line, a link declaration for it is provided in the output. Limitations: Left recursion in the grammar may cause the parser to loop. See also: recog.icn, matchlib.icn, and parscond.icn.

---

parse, parsex (KW, CW)

parse parses simple statements. parsex is another expression parser, adapted from C code written by Allen I. Holub (Dr. Dobb's Journal, Feb 1987). This program can evaluate any expression consisting of numbers and the following operators (listed according to precedence

level):

() - ! 'str'str' * / & + - < <= > >= == != && ||

All operators associate left to right unless () are present. The top - is a unary minus.

---

patchu (RM)

patchu reads a source file and a diff file, producing an updated file. The diff file may be generated by the UNIX diff(1) utility, or by diffu.icn, which uses dif.icn. The original patch(1) utility, written by Larry Wall, is widely used in the UNIX community. See diff(1) in a UNIX manual for more details. Requires: co-expressions.

---

pdecomp

pdecomp lists the prime factors of integers given in standard input.

---

polydemo (EE)

polydemo demonstrates the polystuf module. The user can create, output, delete, or operate up to 26 polynomials, indexed by letter.

---

post [-n groups] [-s subj] [-d dist] [-f followup] [-p prefix] [file] (RF)

post posts a news article to Usenet. Given the name of a file containing a news article, post creates a follow-up article, with an attribution and quoted text. On systems posting via inews, post validates newsgroups and distributions in the active and distributions files in the news library directory. Newsgroup validation assumes the active file is sorted.

---

press (RJA)

press is a file archiving utility that contains extensive tracing facilities that illustrate the LZW compression process in detail. The LZW compression procedures in this program are general purpose and suitable for reuse in other programs.

---

procprep

procprep is used to produce the data needed to index the "#:" comments on procedure declarations that is needed to produces a permuted index to procedures.

---

procwrap

procwrap takes procedure names from standard input and writes minimal procedure declarations for them. For example, the input line wrapper produces:

```
procedure wrapper()
end
```

This program is useful when you have a lot of procedures to write.

---

psrsplit file (GMT)

psrsplit separates psrecord.icn output pages. If a file produced by the procedures in psrecord.icn contains multiple pages, it cannot be easily incorporated into another document. psrsplit reads such a file and breaks it into individual pages. For an input file named xxxx or xxxx.yyy, the output files are named xxxx.p01, xxxx.p01, etc. for as many pages as are available. It is assumed that the input file was written by psrecord.icn; the likelihood of correctly processing anything else is small.

puzz (CT)

puzz creates word search puzzles.

qei (WHM, REG)

qei evaluates expressions interactively. A semicolon is required to complete an expression; without one the subsequent line is added to what already has been entered. qei accumulates expressions and evaluates all previously entered expressions before it evaluates a new one. A line beginning with a colon is a command. The commands are: :clear clears the accumulated expressions; :every generates all the results from the expression, otherwise, at most one is produced; :exit or quit terminates the session; :list lists the accumulated expressions; and :type toggles the switch that displays the type of the result.

qt [-a] (RJA)

qt writes out the time in English. If -a is present, only the time is printed: just after a quarter to three. Otherwise, time is printed as a sentence: It's just after a quarter to three.

queens (SBW)

queens displays the solutions to the non-attacking n-queens problem: the ways in which n queens can be placed on an n-by-n chessboard so that no queen can attack another. An integer command line argument specifies the number of queens (default: 6). For example, queens -n8 displays the solutions for 8 queens on an 8-by-8 chessboard.

ranstars (SBW)

ranstars displays a random field of "stars" on an ANSI terminal at randomly chosen positions on the screen until a specified maximum number is reached. It then extinguishes existing stars and creates new ones for a specified steady-state time.

reply [prefix] < *news-article* or *mail-item* (RF)

reply creates the appropriate headers and attribution, quotes a news or mail message, and uses system() to put the user in an editor and then mails the reply. The default quote prefix is "> ". The editor can be specified with the EDITOR environment variable.

repro (KW)

repro is the shortest known self-reproducing Icon program.

revsort

revsort sorts strings with characters in reverse order.

roffcmds

roffcmds processes standard input and writes a tabulation of nroff/troff commands and defined strings to standard output. Limitations: the program only recognizes commands that appear at the beginning of lines and does not attempt to unravel conditional constructions. Similarly, defined strings buried in disguised form in definitions are not recognized.

rsg

rsg generates random strings from a grammar. Grammars are context-free extended

BNF similar to that of pargen. This interactive program allows the user to build, test, modify, and save grammars. Input consists of various specifications, which can be intermixed: Productions define nonterminal symbols in syntax similar to BNF Generation specifications cause the generation of a specified number of sentences from the language defined by a given nonterminal symbol. Grammar output specifications cause the definition of a specified nonterminal or the entire current grammar to be written to a given file. Source specifications cause subsequent input to be read from a specified file. Any line beginning with = causes the rest of that line to be used as a prompt to the user whenever rsg is ready for input (there normally is no prompt). A line with a single = stops prompting.

Specifying a new production for a nonterminal symbol changes its definition. There are a number of special devices to facilitate the definition of grammars, including: <lb> for <, <rb> for >, <vb> for |, <nl> for newline, <> for the empty string, <&lcase> denoting any single lowercase letter, <&ucase> for any single uppercase letter, and <&digit> representing any single digit (Note: &digit, not &digits). In addition, if the string between a < and a > begins and ends with a single quotation mark, it stands for any single character between the quotation marks. For example, <'xyz'> is equivalent to x|y|z

---

ruler [length:80] [#lines] (RJA)

ruler writes a character ruler to standard output. Arguments give the length of the ruler in characters and the number of lines to write, with a line number on each line.

---

scramble (CT)

scramble takes a document and re-outputs it in a cleverly scrambled fashion. It uses the next two most likely words to follow.

---

setmerge file [[op] file]... (GMT)

setmerge combines sets of items according to the specified operators. Sets are read from files, one entry per line. Operation is from left to right without any precedence rules. After all operations are complete the resulting set is sorted and written to standard output.

---

shar text_file... (RJA)

shar creates a Bourne shell archive of text files.

---

shortest

shortest writes the (last) shortest line in the input file. If the command-line option -# is given, the number of the shortest line is written first.

---

shuffile

shuffile writes a version of the input file with the lines shuffled.

---

sing (FJL)

sing is an adaptation of a SNOBOL program by Mike Shapiro in (Griswold71). It writes the lyrics to the song, "The Twelve Days of Christmas" to a parameter that can be any file open for output. The algorithm used can be adapted to other popular songs.

---

snake [character:o] (RLG)

While away the idle moments watching the snake eat blank squares on your screen. To run snake, your terminal must have cursor movement and able to do reverse video.

---

solit (JN, PLT, REG)

solit was inspired by a solitaire game that was written by Allyn Wade in 1985. This program supports several common terminals and PC's. Note: The command-line argument, which defaults to support for the VT100, determines the screen driver.

---

sortname

sortname sorts a list of person's names by the last names.

---

splitlit

splitlit creates a string literal with continuations in case it's too long. Option -w i specifies the width of piece on line, default 50. Option -i i specifies the indent, default 3.

---

streamer

streamer outputs one long line obtained by concatenating the lines of the input file. Option -l i stops when line reaches or exceeds i; default no limit. Option -s s inserts separator s after each line; default no separator. Separators are counted in the length limit.

---

strpsgml [-f translation-file] [left-delimiter [right-delimiter]] (RLG)

strpsgml strips or translates SGML <>-style tags. The default left-delimiter is <, the default right delimiter is >. If no translation file is specified, the program strips material between the delimiters. The format of the translation file is:

```
code  initialization  completion
```

A tab or colon separates the fields. If you want to use a tab or colon as part of the text (and not as a separator), place a backslash before it. The completion field is optional.

---

tablc

tablc tabulates characters and lists each character and the number of times it occurs.

---

tablw

tablw tabulates words and lists number of times each word occurs. A word is a string of consecutive letters with at most one interior occurrence of a dash or apostrophe.

---

textcnt

textcnt tabulates the number of characters, "words", and lines in standard input and gives the maximum and minimum line length.

---

textcvt (RJA)

textcvt converts text file(s) among various platforms' formats. The supported text file types are UNIX, MS-DOS, and Macintosh. The files are either converted in-place by converting to a temporary file and copying the result back to the original, or are copied to a separate new file, depending on the command line options.

---

toktab

toktab reads the token files given on the command line and summarizes them in a single

file. Option -n sorts tokens by category in decreasing *numerical* order; default is alphabetical. Option -l i limits output in any category to i items; default no limit

---

trim [n]

trim copies lines from standard input to standard output, truncating the lines at n characters and removing trailing blanks. The default for n is 80.

---

ttt (CT)

ttt plays the game of tic-tac-toe.

---

turing (GMT)

turing simulates the operation of an n-state Turing machine. The machine starts in state 1 with an empty tape. A description of the Turing machine is read from the file given as a command-line argument, or from standard input if none is specified.

---

unique (AVH, RJA)

unique filters out (deletes) identical adjacent lines in a file.

---

unpack

unpack unpackages files produced by pack.icn. See that program for limitations.

---

upper

upper maps the names of all files in the current directory to uppercase.

---

verse [vocabularyfile:verse.dat (CT)

This verse maker was initially published in a 1980s Byte magazine in TRS80 Basic.

---

versum

versum writes the versum sequence for an integer to a file of a specified name. If such a file exists, it picks up where it left off, appending new values to the file.

---

webimage

webimage takes image filename arguments and writes a Web page that embeds each image. Option -a s specifies alignment, default "bottom". Option -t s supplies a title for the page; default "untitled". Option -n directs to include file names.

---

what (PLT)

what writes all strings beginning with "@" followed by "(#)" and ending with null, newline, quotes, greater-than or backslash. Follows UNIX what(1) conventions.

---

when (CT)

when is a date based ls command. UNIX find is a bit arcane, so when provides a simpler alternative. This program only works in the current directory. Requires: UNIX.

xtable (RJA, AB)

xtable prints tables for ASCII, EBCDIC and their decimal and octal values.

yahtz (CT, RLG, PLT)

A classic "dice poker" game.

zipsort

zipsort sorts labels produced by labels in ascending order of their postal zip codes. Option -d n sets the number of lines per label to n (default 9). See also: labels.icn.

B.3 Selected IPL Authors and Contributors

This Appendix presents the work of the following authors and contributors of Icon Program Library modules and programs. Ralph Griswold initiated and maintained the collection. The various authors' code is described in their own words, from the public domain documentation they have written about it. We acknowledge their contribution to the Icon community and to this book. Errors in this Appendix are solely our responsibility.

Paul Abrahams	Robert J. Alexander	Allan J. Anderson
Norman Azadian	Alan Beale	Phil Bewig
Peter A. Bigot	David S. Cargo	Alex Cecil
Alan D. Corre	Cary A. Coutant	William E. Drissel
Erik Eid	Ronald Florence	David A. Gamey
Michael Glass	Richard L. Goerwitz	Ralph E. Griswold
Matthias Heesch	Charles Hethcoat	Anthony V. Hewitt
Thomas R. Hicks	Clinton L. Jeffery	Jere K?pyaho
Justin Kolb	Tim Korb	Frank J. Lhota
Nevin J. Liber	William P. Malloy	C. Scott McArthur
Will Menagarini	Joe van Meter	William H. Mitchell
Rich Morin	Jerry Nowlin	Mark Otto
Robert Parlett	Jan P. de Ruiter	Randal L. Schwartz
Charles Shartsis	David J. Slate	John D. Stone
Chris Tenaglia	Phillip L. Thomas	Gregg M. Townsend
Kenneth Walker	Stephen B. Wampler	Beth Weiss
Robert C. Wieland	Cheyenne Wills	David Yost

Appendix C

The Unicon Component Library

C.1 GUI Classes

This section presents the various methods and class fields making up the classes implemented in the Unicon GUI library. The library is a package (gui) defined by a set of files in their own directory (uni/gui). In this section if a class has superclasses, their names are given, separated by colons. Superclasses should be consulted for additional methods and class fields used by subclasses. The default is to have no superclass. If a required field is omitted, an error message may be produced, such as:

```
gui.icn : error processing object TextList : x position unspecified
```

This means that the x position of a TextList object was not specified, probably because the set_pos() method had not been invoked.

You will generally need to consult "Graphics Programming in Icon", by Griswold, Jeffery, and Townsend in order to make the best use of these classes.

Notification

An instance of this class is generated by components and passed to the connected event handler method. It holds three elements, which are accessed as follows:

get_source() returns the component associated with the event. This may be a subclass of either Component or a MenuComponent. If this is &null, then an Icon event has occurred which has not produced an Event from any component.

get_type() returns a field to distinguish between different types of event generated by the same component (ACTION_EVENT, MOUSE_PRESS_EVENT, etc.).

get_param() returns the parameter associated with the event, if any.

Dialog : Container

This is the parent class of a dialog window.

resize_win(w, h) resizes the window to the given dimensions.

get_win() returns the Icon window associated with the dialog.

set_min_size(w, h) sets the minimum dimensions for a window. The user will not be able to resize the window below this size.
set_focus(c) sets the keyboard focus to component c. clear_focus() clears the focus.
dialog_event(e:Event) must be defined in the subclass. It is invoked on each event.
dispose(x) is normally invoked in response to an event. It sets a flag to indicate that the dialog should be closed. If x is non-null, the window is closed already.
set_unique(c) is called by a component c to indicate the beginning of unique event processing, whereby one component alone receives all events.

clear_unique(x) unsets unique event processing mode. If x is &null then the final event in unique processing mode will be passed to all objects; otherwise it will not.

show_modal(d) opens and displays the dialog window, and accepts events into it. As events occur they are handled, until a call to dispose() is made, then the window is closed and the method returns. Parameter d is the parent dialog, if any.

Open() opens the dialog window. Close() closes the dialog window.

process_event(e) processes Icon event e, calling event handlers.

win : window is the dialog's window.

is_open : flag indicates whether the window is open.

focus : Component specifies the component with the current focus.

unique_flag : flag controls unique processing, where one component receives all events.

re_process_flag : flag tells whether to distribute last Icon event during unique mode.

buffer_win is a buffer window for double buffering.

min_width : integer is the minimum width of window, or &null if no minimum.

min_height : integer is the minimum height of window, or &null if no minimum.

Component

This is the parent class of all the GUI components. All of its methods and variables therefore apply to its sub-classes.

get_x_reference(), get_y_reference(), get_w_reference(), and get_h_reference() produce this object's x position, y position, width, and height values from which to compute absolute positions. May be over-ridden; by default returns self.x, self.y, self.w, self.h.

get_cwin_reference(), get_cbwin_reference() produce the cloned window (the reference object inherits the attributes by cloning this window) and cloned buffer window.

fatal(s) prints an error message s together with the class name, and stops the program.

generate_components() : Component* generates the components for the object. By default this just returns the component itself.

is_shaded() : flag? succeeds if the component is shaded. A shaded component, such as a button, may be displayed differently, and will not generate events.

set_is_shaded() sets the shaded status of the component to shaded.

clear_is_shaded() sets the shaded status of the component to not shaded.

toggle_is_shaded() swaps the shaded status of the component.

set_draw_border() sets the component such that a border is drawn.

clear_draw_border() sets the component such that a border is not drawn.

toggle_draw_border() toggles whether or not to draw a border around the component. Different objects respond differently to this flag being set; some ignore it altogether.

display(buffer_flag) draws, or re-draws, the component in the dialog window. If buffer_flag is not null, then the component is displayed into the buffer window, not the dialog window (this is used for double-buffering purposes).

set_accepts_tab_focus() sets the accepts_tab_focus_flag, meaning that the object will gain the keyboard focus by way of the user pressing the tab key repeatedly.

clear_accepts_tab_focus() clears the accepts_tab_focus_flag.

attrib(x[]) sets the graphic attributes of the component to the given parameter list. For example: w.attrib("font=helvetica", "bg=pale blue"). Arguments may be either strings or lists of strings. A description of the attributes is in Chapter 7 and detailed in [GJT98].

get_parent_win() : window returns the window in which the component resides.

set_pos(x, y) sets the x and y position of the component. Each coordinate can be either an absolute pixel position, or a percentage plus or minus an offset.

```
c.set_pos(100, "25%")
c.set_pos("50%-20", "25%+100")
```

set_align(x_align:"l", y_align:"t") sets the alignment of the component. Options for x_align are "l", "c" and "r", for left, center, and right alignment. Options for y_align are "t", "c" and "b", for top center and bottom alignment. Examples:

```
# Place c centered in the center of the window,
# its top left corner at (10,10).
c.set_pos("50%", "50%")
c.set_align("c", "c")
c.set_pos(10, 10)
c.set_align("l", "t")
```

set_size(w, h) sets the size of the component; parameters are as for set_pos() above. Some components have sensible default sizes, but on others the size must be set explicitly.

handle_event(e) : Event? is over-ridden by all this class's subclasses. It is the method that handles an Icon event e. It would not normally be called by a user program. Its result is passed to the dialog_event() method of the dialog.

The first two fields of the Event structure are the Icon event e and the object itself. The third field is the code, which can be any integer. For example:

```
method handle_event(e)
  ...
 return Event(e, self, 0)
end
```

do_shading(w) is called from a component's display() method. This method filters the component to give a shaded appearance if the is_shaded_flag is set. w is the window to use (normally cwin).

in_region(x:&x,y:&y) succeeds for unshaded components if (x,y) lies within the component.

got_focus() is called when the component gets the keyboard focus. It may be extended in subclasses. For example:

```
method got_focus()
  self$Component.got_focus()
  #
  # Display the box cursor
  #
  display()
end
```

lost_focus() is called when the component loses the keyboard focus; it may be extended.

unique_start() initiates unique event processing for this object by calling the parent dialog's set_unique() method.

unique_end(x) ends unique event processing for this object by calling the parent dialog's clear_unique(x) method.

init() sets up the cloned windows for the component. This method should be called for any components created and used inside any custom components.

set_parent_dialog(x) sets the owning Dialog of the component to x.

get_parent_dialog() : Dialog returns the parent dialog of the component.

firstly() is invoked after the position of the object has been computed, but before the object has been displayed in the window. This method may be extended in subclasses.

finally() is invoked before the window is closed; it may be extended in subclasses:

```
method finally()
  self$Component.finally()
  # Do something here
  ...
  return
end
```

resize() computes the absolute positions and sizes from the specifications given by set_pos() and set_size(). This method needs to be extended for a component that contains other components. See the section on custom components for an example.

x_spec, y_spec are the x and y positions as specified by set_pos(), e.g. "50%"

w_spec, h_spec are the width and height specifiers as used in set_size(), e.g. "100%"

x_align, y_align are the x and y alignment as specified in set_align(), e.g. "l", "b".

The following four attributes are absolute dimensions in pixels, compiled from x_spec, y_spec, w_spec, and h_spec, and the dimensions of the enclosing object or window.

x and y are the x and y positions computed from x_spec and y_spec.

w and h are the width and height computed from w_spec and h_spec.

cwin is a cloned window created by combining the Dialog's canvas with the Component's attributes, so drawing into this window will draw straight to the Dialog window with the correct attributes. cbwin is a cloned window created by combining a buffer window with the Component's attributes. This is used solely for double-buffering purposes.

parent_dialog is the Dialog class instance of which this Component is a part.

attribs is a list of strings specifying graphics attributes, e.g. ["bg=blue", "resize=on"].

has_focus : flag indicates whether the Component currently has the keyboard focus.

accepts_tab_focus_flag : flag indicates whether the Component accepts keyboard focus by way of the tab key being pressed.

draw_border_flag : flag indicates whether the Component should have a border drawn around it. Many components (such as TextButtons) ignore this flag.

is_shaded_flag : flag indicates whether the Component currently is shaded.

reference links to the object used to calculate absolute sizes from percentage sizes. For objects placed directly in the Dialog, rather than in some other object, this will point to the Dialog itself, which over-rides the several methods get_x_reference() etc., appropriately.

Container : Component

This class contains other components. The container itself is invisible. Many of Component's methods are over-ridden by Container. A Dialog is a sub-class of this class.

add(c:Component) adds the component c to the Container.

Instance variable components are the components inside the Container.

VisibleContainer : Component

This is similar to a Container, except that the object itself is a capable of display.

add(c:Component) adds the component c to the VisibleContainer.

components are the components inside the VisibleContainer.

Button : Component

This is the parent class of button classes including TextButton and IconButton. A button produces an Event of code 0 when the button is depressed, and code 1 when it is released. By default, when a button holds the keyboard focus a dashed line appears just within the button. When return is pressed an event of code 2 is generated. The method Dialog.set_initial_focus() can be used to give the button the keyboard focus when the dialog is first displayed.

set_no_keyboard() disables the keyboard control over the button described above. No dashed line will ever appear in the button display and return will have no effect on the button even if it has the focus.

TextButton : Button

A button with a text label. The size of the button can either be set using set_size() or be left to default to a size based on the given label.

set_internal_alignment(x) sets the alignment of the label within the button. The parameter

should be either "l", "c" or "r" to set the alignment to left, center or right respectively. If this method is not invoked, then the alignment is centered.

set_label(x) sets the label in the button to the given string. Examples:

```
b := TextButton()
b.set_label("Cancel")
b.set_pos("50%", "80%")
b.set_align("c", "c")
add(b)
```

IconButton : Button

This is a button with an Icon image within it. There is a useful program in the Icon program library called xpmtoims, which will take an xpm file and output the equivalent Icon image string, which can then be inserted into a program. See also the X Window programs sxpm and pixmap for viewing and editing xpm files respectively.

A border may be requested with set_draw_border(). Unless explicitly specified, the size will default to the image's size, plus a standard surrounding area if a border is requested.

set_img(s:string) sets the image to s, which should be in Icon image format. Examples:

```
# Create a button with a diamond image and a border
b := IconButton()
b.set_draw_border()
b.set_img("11,c1,_
~~~~~0~~~~~_
~~~~000~~~~_
~~0000000~~_
~000000000~_
~~0000000~~_
~~~~000~~~~_
~~~~~0~~~~~_
")
```

ButtonGroup

This class groups several Buttons together. Then, when the mouse is clicked down on one of the Buttons and then dragged onto another before being released, the other Button will go "down". This is the common behavior for buttons in a bar along the top of an application.

Note: a Button must be added to both the ButtonGroup and the Dialog too. Examples:

```
bg := ButtonGroup()
b := TextButton()
b.set_label("Okay")
add(b)
bg.add(b)
```

add(c:Button) adds the given Button to the ButtonGroup.

Label : Component

This simply creates a text label in the dialog window. Calling set_draw_border() adds a border around the label. The size will default if not set.

set_label(s:string) sets the label to the given string.

set_internal_alignment(x) sets the horizontal alignment of the label within the area of the component; should be "l", "c" or "r". Default is "l". If the horizontal size is left to default, then setting this field should make no difference, because the size of the component will be set so that the string just fits into it.

Icon : Component

This displays an icon, supplied in Icon image format. A border may be requested with set_draw_border(). The size defaults to the image's size, plus a standard surrounding area if a border is requested. set_img(s:string) sets the image to be displayed.

Image : Component

This class loads an image from a file and displays it. The image should be in GIF format. A border may be included with set_draw_border(). The size of the area into which the image is drawn must be set with set_size().

set_filename(s:string) sets the name of the file from which to load the image; redisplay the image from the new file if appropriate.

set_scale_up() scales the image up to fit in the space specified by set_size(). The image is not distorted, but will be expanded to fill one of the dimensions depending on its shape. If the image is bigger than the specified size then it will be scaled down.

set_internal_alignment(x:"c", y:"c") sets the horizontal and vertical alignment of the image within the component; x should be "l", "c" or "r", y should be "t", "c" or "b".

Border : VisibleContainer

This class provides decorative borders. The add(c) method may optionally be used to set one other component to be the title of the Border. This would normally be a Label object, but it could also be a CheckBox, Icon, or whatever is desired.

set_internal_alignment(x) sets the alignment of the title to "l", "c" or "r".

```
b := Border()
#
# Add a Label as the title
#
l := Label()
l.set_label("Title String")
b.add(l)
add(b)
```

ScrollBar : Component

This class provides horizontal and vertical scroll bars. The first way to use a scroll bar is to set a total_size (represented by the whole bar), a page_size (represented by the

draggable button) and an increment_size (being the amount added/subtracted when the top/bottom button is pressed). The value then ranges from zero to (total_size - page_size) inclusive. An initial value must be set with the set_value() method. For example:

```
vb := ScrollBar()
vb.set_pos("85%", "25%")
vb.set_size(20, "40%")
vb.set_total_size(130)
vb.set_page_size(30)
vb.set_increment_size(1)
vb.set_value(0)
add(vb)
```

Alternatively, a scroll bar can be used as a slider over a given range of values. In this case, the range is set with set_range(). It is still necessary to set the increment_size and the initial_value, but page_size and total_size should not be set. Real numbers as opposed to integers can be used for the range settings if desired. For example:

```
vb := ScrollBar()
vb.set_pos("85%", "25%")
vb.set_size(20, "40%")
vb.set_range(2, 25)
vb.set_value(10)
vb.set_increment_size(1)
add(vb)
```

An Event is returned whenever the buttons are pressed or the bar dragged; the value can be retrieved by get_value(). The event code (obtainable by get_code()) is 1 if the bar has been dragged, and 0 if either button has been pressed or the bar released after being dragged. This fact can be used to reduce the number of events which are processed by the user's program - just ignore events with code 1.

set_is_horizontal() makes the scroll bar horizontal (default is vertical).

set_range(x, y) sets the scroll bar range (integer or real) from x to y inclusive.

set_total_size(x) sets the total size which the scroll bar area represents.

get_total_size() returns the total size.

set_page_size(x) sets the size that the bar in the scroll bar area represents.

get_page_size() gets the page size.

set_value(x) sets the value representing the top of the bar in the scroll bar. The value is forced into range if it is not in range already.

get_value() gets the value.

set_increment_size(x) sets the amount by which to increase when a button is pressed.

TextField : Component

TextField is a class for a single input line of text. The text can scroll within the area specified. By default, a border surrounds the text area; this can be turned of by using clear_draw_border(). The horizontal size must be set by the set_size() method: there is no default (the vertical size will default, however). An event is generated when return is pressed (with code 0), and whenever the contents are changed (with code 1).

get_contents() returns the present contents of the text field. set_contents(x) sets the contents of the field. If not invoked then the initial content is the empty string. Examples:

```
t := TextField()
t.set_pos(50, 250)
# Vertical size will default
t.set_size(100)
t.set_contents("Initial string")
add(t)
```

set_displaychar() controls whether the characters that are typed into the text field are displayed or not. set_displaychar("*") will display a "*" character for each character typed (useful for hiding passwords). The default is set_displaychar(&null) which displays the typed characters normally.

CheckBox : Component

This class creates a small button with a label which is either in an on or off state. The button is an Icon image, which may be specified by the user if desired. The images and size default to appropriate values if not specified.

set_imgs(x, y) sets the up/down images for the button. The images should be in Icon image format. The two images must have the same dimensions.

is_checked() succeeds if the button is down (checked); fail otherwise.

toggle_is_checked() toggles the initial status of the button.

set_is_checked() sets the status of the button to checked.

clear_is_checked() sets the status of the button to not checked.

set_label(x) sets the label of the component to the given string.

get_status() returns 1 if the CheckBox is checked, &null otherwise. Examples:

```
c := CheckBox()
c.set_pos(200, 100)
c.set_label("Checkbox")
add(c)
```

CheckBoxGroup

This class contains CheckBox objects that act together as "radio buttons". The image style of CheckBoxes in a CheckBoxGroup draws diamonds rather than boxes. The status of a CheckBoxGroup should be set with the set_which_one() method, not by turning the individual CheckBoxes on/off with their own methods - that would confuse the program. Note: a CheckBox must be added to both the CheckBoxGroup and the dialog box.

set_by_flag(i) sets the CheckBox which is down according to the integer i. If i = 1 then the first CheckBox is down, if i = 2 the second is down, etc for i = 4, 8, 16.

get_by_flag() returns an integer in the range 1, 2, 4, 8 ... depending upon whether the first, second, third etc CheckBox is down.

add(c:CheckBox) adds c to the CheckBoxGroup.

get_which_one() returns the CheckBox which is currently down.

set_which_one(x:CheckBox) sets which CheckBox is down to x. Examples:

```
#
# Create a CheckBoxGroup of 3 CheckBoxes
#
c := CheckBoxGroup()
c1 := CheckBox()
c1.set_pos(200, 50)
c1.set_label("Checkbox 1")
add(c1)
c.add(c1)

c2 := CheckBox()
c2.set_pos(200, 90)
c2.toggle_is_shaded()
c2.set_label("Checkbox 2")
add(c2)
c.add(c2)

c3 := CheckBox()
c3.set_pos(200, 130)
c3.set_label("Checkbox 3")
add(c3)
c.add(c3)
#
# Initially, set the first one "on"
#
c.set_which_one(c1)
```

TextList : Component

This class displays a list of strings. See the class EditableTextList if you require a multiline text input region. Horizontal and vertical scroll bars are displayed if necessary.

Optionally the user can be allowed to select either one line only, or several lines. In either case, an event is generated when a line is selected.

get_contents() : list returns the current contents as a list of strings.

set_contents(x:list, line, left_pos, preserve_selections) sets the contents to x and sets the position to line and left_pos. If these parameters are omitted then the default is to start

at line 1, with left offset zero if the window is not already open, or to retain the existing position if it is. If the last parameter is non-null then the current selections are retained; otherwise they are reset. This method has no effect if the component is in editable mode and the window is already open.

```
tl := TextList()
tl.set_contents(data)
 ...
# Amend data and go to end of data
put(data, "New line")
tl.set_contents(data, *data, 0)
```

The method set_select_one() specifies that only one line of the list may be highlighted, while method set_select_many() specifies that several lines of the list may be highlighted. Of no effect if in editable mode.

get_selections() : list returns a list of the numbers of the lines that are highlighted.

set_selections(L) sets the highlighted selections to a given list of line numbers.

```
tl := TextList()
tl.set_pos("50%", "50%")
tl.set_size("70%", "50%")
tl.set_align("c", "c")
tl.set_contents(data) # data is a list of strings
add(tl)
```

DropDown

This class is a superclass of List and EditList below.

set_selection_list(x) sets the list of selections to the list x.

get_selection():integer returns the index of the item in the list presently selected.

List : Component : DropDown

This component is for selecting one string from a list. When a button is pressed a list appears (possibly with a scroll bar) from which one item can be selected. An Event is generated whenever an item is selected. A width must be specified for this component.

set_selection(x) sets the selected item to element x.

set_constant_label(x) supplies a string that will always appear in the text part of the component, rather than the currently selected item.

The methods for handling the list of selections are mostly inherited from DropDown.

```
L := List()
L.set_selection_list(["Red","Green","Yellow","Blue","Orange"])
L.set_size(120)
L.set_pos(100, 100)
L.set_selection(2) # Green will be the first selection
add(L)
```

EditList : Component : DropDown

An EditList works like a List, but the user may edit the item that is selected. An extra method is therefore supplied to get the content, as it may not correspond to an element of the list. An Event is generated with code 0 if an element of the list is selected, with code 1 if return is pressed, and with code 2 if the user edits the selected item.

get_contents() returns the contents of the selected item (which may have been edited).

set_contents(x) sets the initial contents of the text to the given string.

MenuBar : Component

This class is the base from which menu systems are created. Menu items are added to this class; they are not separate components added to the dialog itself. The default position is (0, 0); the default size is 100% of the width of the screen and a reasonable height based on the font specified. add(c:Menu) adds c to the MenuBar. This will be one drop down menu. Items are then added to the Menu.

MenuButton : Component

This is similar to MenuBar, but holds just a single drop-down menu, rather than several. set_menu(x:Menu) sets the menu to be displayed when the component is clicked.

MenuComponent

This is the superclass of all the objects that make up the menu system (other than MenuBar of course). For components that appear in a menu with a label, an optional left/right string/image can be set.

set_label_left(x) sets the optional left label to the given string.

set_label_right(x) sets the optional right label to the given string.

set_img_left(x) sets the optional left image to the given Icon image.

set_img_right(x) sets the optional right image to the given Icon image.

toggle_is_shaded() toggles whether or not the item is shaded. If it is, it is displayed in a filtered way and will not accept input.

set_is_shaded() sets the shaded status of the component to shaded.

clear_is_shaded() sets the shaded status of the component to not shaded.

set_label(x) sets the center label to the given string.

SubMenu : MenuComponent

This class encapsulates a Menu object that when selected will display something outside the menu itself (for example a sub-menu of other menu items). It is intended to be extended by custom menu components, and should not be instantiated. Methods are empty.

hide_non_menu() and set_which_open(x) are called by Menu for any SubMenu object, but would not normally need to be over-ridden by a custom class.

resize() may be overridden to initialize the width and height of the object.

display() must be over-ridden; it displays the object.

handle_event(e) handles the Icon event e and must be over-ridden.

hide() hides (closes) the object's display and must be over-ridden.

Menu : SubMenu

This class encapsulates a drop down menu, or a sub-menu. The left, center and right labels/images of the elements within it are formatted within the menu automatically.

add(c:Component) adds the given component to the Menu.

TextMenuItem : MenuComponent

This class encapsulates a single text item in a Menu. It has no additional methods that the user need call other than are contained in its parent class, MenuComponent.

CheckBoxMenuItem : MenuComponent

This class encapsulates a check box in a menu. Several CheckBoxMenuItems may be added to a CheckBoxGroup structure to give "radio buttons" within menus.

set_imgs(x, y) sets the up and down images to x and y respectively. The default is boxes, unless the component is in a CheckBoxGroup in which case the default is diamonds.

is_checked() succeeds if the component is checked; fail otherwise.

set_is_checked() sets the status of the button to checked.

clear_is_checked() sets the status of the button to not checked.

MenuSeparator : MenuComponent

This is a horizontal bar in a Menu, for decorative purposes. It has no user methods.

TableColumn : TextButton

This class provides one column within a Table class, which displays a table of data. A column has a label with a button that produces an event when clicked. The column may be expanded or contracted by dragging the right edge of the button. Calling the set_label(x) method of the superclass, TextButton, sets the label.

set_column_width(x) sets the initial width of the column in pixels; this is required.

```
c1 := TableColumn()
c1.set_internal_alignment("r") # Label is right aligned
c1.set_column_width(80)
c1.set_label("Number")
```

Table : Component

This class displays a table, the columns of which are set up using TableColumns.

set_button_bar_height(x) sets the height of the buttons at the top in pixels.

set_contents(x) sets the contents of the table. The parameter should be a two dimensional list. Each element of the list should correspond to one row of the table.

set_contents(x, line:1, left_pos, preserve_selections) sets the contents to x and sets the position to line and left_pos. The default left_pos offset is the existing position if the window is already open, or zero otherwise. The last parameter is a flag that directs whether to retain the current selections.

add(c:TableColumn) adds the given TableColumn to the Table.

set_select_one() specifies that only one row of the table may be highlighted.

set_select_many() allows several rows of the table to be highlighted.

get_selections() returns a list of the numbers of the rows which are highlighted.

set_selections(L) sets the line numbers that are selected to the list of line numbers L.

TabItem : Container

This class represents a single pane in a TabSet. Components are added to the TabItem using Container's add() method. They are then displayed and accept input when that TabItem is selected. Components added to the TabItem are positioned relative to the position and size of the parent TabSet. Therefore for example set_pos("50%", "50%") refers to the center of the TabSet rather than the center of the screen. The components also inherit any window attributes of the TabSet, such as font, color and so on.

set_label(x) sets the TabItem's label.

TabSet : VisibleContainer

This class holds the several TabItems.

set_which_one(x) sets the displayed TabItem; default is the first one.

add(c:TabItem) adds the given TabItem to the TabSet.

Panel : VisibleContainer

This class simply contains other components. The components inside have their sizes and positions computed relative to the Panel and also inherit the Panel's windowing attributes. Components are added using the add() method of VisibleContainer.

OverlayItem : Container

This class is one "pane" in an OverlaySet, which is rather like a TabSet except that there are no tabs, and control over which pane is displayed is entirely the affair of the program. The components inside have their sizes and positions computed relative to the parent OverlaySet and also inherit the OverlaySet's windowing attributes. Components are added using the add() method of Container.

OverlaySet : VisibleContainer

An OverlaySet is a set of OverlayItems.

set_which_one(x) sets the currently displayed OverlayItem; default is the first.

add(c:OverlayItem) adds the given OverlayItem to the OverlaySet.

Appendix D

Differences between Icon and Unicon

This appendix summarizes the known differences between Arizona Icon and Unicon. Since the language has myriad additions covered by the whole book, the emphasis of this page is on *incompatibilities* that might require changes to existing Icon programs.

D.1 Extensions to Functions and Operators

Unicon broadens the meaning of certain pre-existing functions where it is consistent and unambiguous to do so. These extensions revolve primarily around the list type. For example, insert() allows insertion into the middle of a list, reverse() reverses a list, and so forth.

D.2 Objects

Unicon supports the concepts of classes and packages with declaration syntax. This affects scope and visibility of variable names at compile time. At runtime, objects behave similar to records in most respects. These additions include reserved words that are no longer valid variable names, such as class, package, and import.

D.3 System Interface

Unicon's system interface presumes the availability of hierarchical directory structure, communication between programs using standard Internet protocols, and other widely available facilities not present in Arizona Icon.

Unicon's graphics include extensive 3D facilities. The 2D facilities are extended with additional image file formats.

D.4 Database Facilities

Unicon supports GDBM and SQL databases with built-in functions and operators or the experimental SQLite plugin. The programmer manipulates data in terms of persistent table and record abstractions. SQL database support may not be present on platforms that do not provide ODBC open database connectivity drivers or the SQLite plugin.

D.5 Multiple Programs and Execution Monitoring Support

Unicon virtual machine interpreters by default support the loading of multiple programs so that various debugging and profiling tools can be applied to them without recompilation. The execution monitoring facilities are described in "Program Monitoring and Visualization: An Exploratory Approach", by Clinton Jeffery. Unicon optimizing compilers may omit or substitute for these facilities.

Appendix E

Portability Considerations

Unicon's POSIX-based system interface facilities presented in Chapter 5 are portable. You can expect the portable system interface to be available on any implementation of Unicon. This appendix presents additional, non-portable elements of the Unicon POSIX interface, as well as some notes on functionality specific to Microsoft Windows.

E.1 POSIX extensions

The extensions presented in this section may or may not be part of the POSIX standard, but they are a part of the Unicon language as implemented on major POSIX-compliant UNIX platforms such as Solaris and Linux. Ports of Unicon to non-POSIX or quasi-POSIX platforms may or may not implement any of these facilities.

Information from system files

There are four functions that read information from the system: getpw() to read the password file, getgr() for the group file, gethost() for hostnames, and getserv() for network services. Called with an argument (usually a string), they perform a lookup in the system file; called with no arguments, these functions step through the files one entry at a time.

The functions setpwent(), setgrent(), sethostent(), and setservent() do the same things as their POSIX C language counterparts; they reset the file position used by the get* routines to the beginning of the file. These functions return records whose members are similar to the C structures returned by the system functions getpwuid(2), gethostbyname(2), etc.

fork and exec

POSIX-compliant systems support a process-launch interface using the functions fork() and exec(). fork() makes a copy of the current process. After the fork there are two identical processes that share resources such as open files, and differ only in one respect: the return

value they received from the call to fork(). One process gets a zero and is called the child; the other gets the process id of the child and is called the parent.

Usually fork() is used to run another program. In that case, the child process uses the system call exec() which replaces the code of the process with the code of a new program. This fork()/exec() pair is comparable to calling system() and using the option to not wait for the command to complete.

The first argument to exec() is the filename of the program to execute, and the remaining arguments are the values of argv that the program will get, starting with argv[0].

```
exec("/bin/echo", "echo", "Hello,", "world!")
```

POSIX functions

These functions are present in all Unicon binaries, but you can expect them to fail on most non-UNIX platforms. Check the readme.txt file that comes with your installation to ascertain whether it supports any of these functions.

---

chown(chown(f, u:-1, g:-1) : ? **change owner**

chown(f, u, g) sets the owner of a file (or string filename) f to owner u and group g. The user and group arguments can be numeric ID's or names.

---

chroot(string) : ? **change filesystem root**

chroot(f) changes the root directory of the filesystem to f.

---

crypt(string, string) : string **encrypt password**

crypt(s1, s2) encrypts the password s1 with the salt s2. The first two characters of the returned string will be the salt.

---

exec(string, string, ...) : null **execute program**

exec(s, arg0, arg1, arg2, ...) *replaces* the currently executing Icon program with a new program named in s. The other arguments are passed to the program. Consult the POSIX exec(2) manual pages for more details. s must be a path to a binary executable program, not to a shell script (or, on UNIX) an Icon program. If you want to run such a script, the first argument to exec() should be the binary that can execute them, such as /bin/sh.

---

fcntl(file, string, options) **file control**

fcntl(file, cmd, arg) performs miscellaneous operations on the open file. See the fcntl(2) manual page for more details. Directories and DBM files cannot be arguments to fcntl(). The following characters are the possible values for cmd:

f	Get flags (F_SETFL)
F	Set flags (F_GETFL)
x	Get close-on-exec flags (F_GETFD)
X	Set close-on-exec flag (F_SETFD)

l	Get file lock (F_GETLK)
L	Set file lock (F_SETLK)
W	Set file lock and wait (F_SETLKW)
o	Get file owner or process group (F_GETOWN)
O	Set file owner or process group (F_SETOWN)

In the case of L, the arg value should be a string that describes the lock, otherwise arg is an integer. A record posix_lock(value, pid) will be returned by F_GETLK.

The lock string consists of three parts separated by commas: the type of lock (r, w or u), the starting position, and the length. The starting position can be an offset from the beginning of the file (e.g. 23), end of the file (e.g. -50), or from the current position in the file (e.g. +200). A length of 0 means lock till EOF. These characters represent the file flags set by F_SETFL and accessed by F_GETFL:

d	FNDELAY
s	FASYNC
a	FAPPEND

---

fdup(file, file) : ? **duplicate file descriptor**

fdup(src, dest) is based on the POSIX dup2(2) system call. It is used to modify a specific UNIX file descriptor, such as just before calling exec(). The dest file is closed; src is made to have its Unix file descriptor; and the second file is replaced by the first.

---

filepair() : list **create connected files**

filepair() creates a bi-directional pair of files analogous to the POSIX socketpair(2) function. It returns a list of two indistinguishable files; writes on one will be available on the other. The connection is bi-directional, unlike that of function pipe(). Caution: typically, the pair is created just before a fork(); after it, one process should close L[1] and the other should close L[2] or you will not get proper end-of-file notification.

---

fork() : integer **fork process**

fork() creates a new process that is identical to the current process except in the return value. The parent gets a return value that is the PID of the child, and the child gets 0.

---

getegid() : string **get effective group identity**

getegid() produces the effective group identity (gid) of the current process. The name is returned if it is available, otherwise the numeric code is returned.

---

geteuid() : string **get effective user identity**

geteuid() produces the effective user identity (uid) of the current process. The name is returned if it is available, otherwise the numeric code is returned.

---

getgid() : string **get group identity**

getgid() produces the real group identity (gid) of the current process. The name is returned if it is available, otherwise the numeric code is returned.

---

getgr(g) : record **get group information**

getgr(g) returns a record that contains group file information for group g, a string group name or an integer group code. If g is null, each successive call to getgr() returns the next entry. setgrent() resets the sequence to the beginning. Return type:

record posix_group(name, passwd, gid, members)

---

gethost(x) : record|string **get host information**

gethost(n) for network connection n returns a string containing the IP number and port this machine is using for a network connection. gethost(s) returns a record that contains host information for the machine named s. If s is null, each successive call to gethost() returns the next entry. sethostent() resets the sequence to the beginning. The aliases and addresses are comma separated lists of aliases and addresses (in a.b.c.d format) respectively. Its return type is record posix_hostent(name, aliases, addresses)

---

getpgrp() : integer **get process group**

getpgrp() returns the process group of the current process.

---

getpid() : integer **get process identification**

getpid() produces the process identification (pid) of the current process.

---

getppid() : integer? **get parent process identification**

getppid() produces the pid of the parent process.

---

getpw(u) : posix_password **get password information**

getpw(u) returns a record that contains password file information. u can be a numeric uid or a user name. If u is null, each successive call to getpw() returns the next entry and setpwent() resets the sequence to the beginning. Return type:

record posix_password(name, passwd, uid, gid, age, comment, gecos, dir, shell)

---

getserv(string, string) : posix_servent **get service information**

getserv(s, proto) returns a record that contains service information for the service s using protocol proto. If s is null, each successive call to getserv() returns the next entry. setservent() resets the sequence to the beginning. If proto is defaulted, it will return the first matching entry. Its return type is record posix_servent(name, aliases, port, proto)

---

getuid() : string **get user identity**

getuid() produces the real user identity (uid) of the current process.

---

kill(integer, x) : ? **kill process**

kill(pid, signal) sends a signal to the process specified by pid. The second parameter can be the string name or the integer code of the signal to be sent.

---

hardlink(string, string) : ? **link files**

hardlink(src,dest) creates a link named dest that points to src.

---

readlink(string) : string? **read symbolic link**

readlink(s) produces the filename referred to in a symbolic link at path s.

---

setgid(integer) : ? **set group identification**

setgid(g) sets the group id of the current process to g. See the UNIX setgid(2) man page.

---

setgrent() : null **reset group information cursor**

setgrent() resets and rewinds the pointer to the group file used by getgr() when getgr() is called with no arguments.

---

sethostent(integer:1) : null **reset host information cursor**

sethostent(stayopen) resets and rewinds the pointer to the host file used by gethost(). The argument defines whether the file should be kept open between calls to gethost(); a nonzero value (the default) keeps it open.

---

setpgrp() : ? **set process group**

setpgrp() sets the process group. This is equivalent to setpgrp(0, 0) on BSD systems.

---

setpwent() : null **reset password information cursor**

setpwent() resets and rewinds the pointer to the password file used by getpw() when getpw() is called with no arguments.

---

setservent(integer:1) : null **reset service information cursor**

setservent(stayopen) resets and rewinds the pointer to the services file used by getserv(). The argument defines whether the file should be kept open between calls to getserv(); a nonzero value (the default) keeps it open.

---

setuid(integer) : ? **set user identity**

setuid(u) sets the user id of the current process to u. See the setuid(2) man page.

---

symlink(string, string) : ? **symbolic file link**

symlink(src, dest) makes a symbolic link dest that points to src.

---

sys_errstr(i) : string **system error string**

sys_errstr(i) produces the error string corresponding to i, a value obtained from &errno.

---

umask(integer) : integer **file permission mask**

umask(u) sets the umask of the process to u, an nine-bit encoding of the read, write, and execute permissions of user, group, and world access. See also chmod(). Each bit in the umask turns *off* that access, by default, for newly created files. The old value of the umask is returned.

---

wait(integer:-1, integer:0) : string **wait for process**

wait(pid, options) waits for a process given by pid to terminate or stop. The default pid value causes the program to wait for all the current process' children. The options parameter is an OR of the values 1 (return if no child has exited) and 2 (return for children that are stopped, not just for those that exit). The returned string represents the pid and the exit status as defined in this table:

UNIX equivalent	example of returned string
WIFSTOPPED(status)	"1234 stopped:SIGTSTP"
WIFSIGNALLED(status)	"1234 terminated:SIGHUP"
WIFEXITED(status)	"1234 exit:1"
WIFCORE(status)	"1234 terminated:SIGSEGV:core"

Currently the rusage facility is unimplemented.

E.2 Microsoft Windows

Windows versions of Unicon support certain non-portable extensions to the system interfaces. Consult Unicon Technical Report 7 for details.

Partial support for POSIX

Windows supports getpid(), but omits other process-related functions such as getppid(). On Windows exec() and system() may only launch Windows 32-bit .EXE binaries.

Windows Unicon supports the following signals in functions such as kill(): SIGABRT, SIGBREAK, SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM.

Windows Unicon supports the umask() function, but ignores execute permission and treats user/group/world identically, using the most permissive access specified.

Native user interface components

Windows Unicon supports limited access to platform-native user interface components and multimedia controls.

WinButton(w,s,x,y,wd,ht) installs a pushbutton with label s on window w.

WinColorDialog(w, s) allows the user to choose a color for a window's context.

WinEditRegion(w, s, s2, x, y, wd, ht) installs an edit box with label s.

WinFontDialog(w, s) allows the user to choose a font for a window's context.

WinMenuBar(w, L1, L2,...) installs a set of top-level menus.

WinOpenDialog(w, s1, s2, i, s3, j, s4) allows the user to choose a file to open.

WinPlayMedia(w, x[]) plays a multimedia resource.

WinSaveDialog(w, s1, s2, i, s3, j, s4) allows the user to choose a file to save.

WinScrollBar(w, s, i1, i2, i3, x, y, wd, ht) installs a scrollbar.

WinSelectDialog(w, s1, buttons) allows the user to select from a set of choices.

Appendix F

Installation

The Downloads page of the Unicon web site (http://unicon.org) has links to the binary distributions of Unicon and to the source code. Unicon may be installed from a binary distribution for Intel based Windows and MacOS platforms. Users on other platforms will usually have to download the source code and build it themselves. This will generally require a supported C99 compiler and environment, such as a make program compatible with GNU make.

Unicon's source code can be downloaded as a compressed archive file with the extension .zip, or from a revision control system. The revision control system sources are much more up to date. There is a copy of the source code at (https://sourceforge.net/projects/unicon) but Unicon development takes place on GitHub (https://github.com/uniconproject/unicon): commits to the main branch are reflected to SourceForge after a small delay.

Unicon is customized using the configure script in the top level directory – the options for customization are displayed by the configure –help command – followed by a make command to build the software. The top level README file has more detailed instructions for the most popular platforms.

Appendix G

Experimental Features

The designers of Unicon have taken a very conservative approach when adding to the language and when changing existing features. With the small number of exceptions that have been previously noted on page 507, an Icon program that runs on the final version of Icon (version 9.5, first released in 1996) will run on the current Unicon system *and give the same results* a quarter of a century later. The conservative approach is continued when dealing with additions to Unicon; breaking existing Unicon programs by making an incompatible change to the language is, in most circumstances, considered to be a very bad thing to do.

Most of the development of Unicon starting from its progenitor has already been discussed but there are some more experimental features that are waiting in the wings. Some of them may never see the light of day in their present form – or, perhaps, in any form – so the most cautious approach is not to rely on any of them until they make their way from this appendix into the definition of the language in Appendix A.

The experimental features are not usually enabled by default in a release build of Unicon – they can only be accessed by making the appropriate pre-processor definitions (or, in some cases, by specifying additional arguments to configure) and rebuilding the system from the source code. Some features that are now part of the language – for example, the array extension to lists that makes them faster in many cases – are still guarded by pre-processor definitions, showing their pedigree as experimental additions, but are now enabled by default. The plugin mechanism and the installed plugins are an exception to the general rule: they *are* enabled by default but should still be considered experimental and subject to change.

A Unicon Technical Report (UTR) is the preferred vehicle for introducing a change to Unicon. The UTR, and associated code, may undergo several rounds of revision before being considered ready for adoption. When it is (ready), the UTR material will usually be incorporated somewhere into the book. UTRs may be found at http://unicon.org/reports.html. Note that *all* of the reports are there, including those that have served their purpose and are no longer under active development.

G.1 User defined operators

This feature extends the syntax of classes to allow the built-in operator symbols to be redefined when their operands are objects. It may be enabled by using the --enable-ovld option to configure before rebuilding the Unicon system.

G.2 Extensions to &random

This feature allows the programmer to choose from a portfolio of different random number generators (in addition to the one provided by Icon). It is also possible to implement other generators and use them without rebuilding Unicon. More than one generator may be in use at the same time. It may be enabled by defining the C preprocessor symbol RngLibrary before rebuilding the Unicon system.

G.3 Plugins

A Unicon plugin is a dynamically loaded library of routines that are encapsulated by a class, which provides access to the external routines (often written in another language) plus a simple facility to enumerate the routines that are available. The external routines may be part of the source code of the plugin or in a separately compiled library that is acquired and installed from elsewhere. At the minimum, a plugin provides a translation layer that converts the arguments to the routines from their Unicon representation into something compatible with the calling conventions of the external implementation language; but usually the plugin also "adds value" by providing a more Unicon-like way of accessing the underlying routines.

G.3.1 Bitman

The Bitman plugin provides low level bit manipulation routines that are the equivalent of the built-in functions (iand, ior, ishift etc.). The salient differences – and the reason for their existence – between the Bitman methods and the built-in functions is that the Bitman methods never produce a large integer, are confined to the natural word length of the machine (so the results are not portable between 32-bit and 64-bit implementations) and, except for bit shifting, do not make a special case of the sign bit. Bitman also provides some bit level enquiry and extraction methods that use a similar addressing convention to string indexing (indexing bits instead of characters).

The methods provided by the plugin are

band	bitwise AND	(iand)
bcom	bitwise one's complement	(icom)
bit	(single) bit extraction	
bits	enquiry and multi-bit extraction	
bor	bitwise inclusive OR	(ior)
brot	bit rotation	
bshift	bitwise shift	(ishift)
bxor	bitwise exclusive OR	(ixor)
ushift	unsigned bitwise shift	
test	confidence testing	

The actual bit manipulation code (written in C) is part of the source code for the plugin: no external libraries are required. The Bitman class has no attributes, so the constructor function has no parameters.

---

band(i, i) : integer **bitwise and**

band(i1, i2) produces the bitwise AND of i1 and i2.

---

bcom(i) : integer **bitwise complement**

bcom(i) produces the bitwise complement (one's complement) of i.

---

bit(i, i) : integer ? **single bit extraction**

bit(i,n) returns the value of the n^{th} bit of i. The indexing works the same way as strings:

bit(i,1) is the least significant bit of i.

bit(i,0) is the most significant bit of i.

If n is negative, indexing is from the most significant end, otherwise it is from the least significant end.

---

bits(i, i, i) : integer ? **multi-bit extraction**

bits(i,n,m) returns the value of the n^{th} to the m^{th} bit of i. The value is shifted down to the least significant end of the machine word. The bits are *not* reversed if m < n.

bits() returns the number of bits in a word – usually 32 or 64.

---

brot(i, i) : integer **bitwise rotation**

brot(i, j) produces the value obtained by rotating i by j bit positions. If j is positive, the rotation is to the left; if j is negative, the rotation is to the right.

---

bor(i, i) : integer **bitwise or**

bor(i1, i2) produces the bitwise OR of i1 and i2.

---

bshift(i, i) : integer **signed bitwise shift**

bshift(i, j) produces the value obtained by shifting i by j bit positions. If j is positive, the shift is to the left, and vacated bit positions are filled with zeros. If j is negative, the shift is to the right with sign extension.

ushift(i, i) : integer **unsigned bitwise shift**

ushift(i, j) produces the value obtained by shifting i by j bit positions. The shift is to the left, if j is positive, or to the right if j is negative. Vacated bit positions are filled with zeros.

G.3.2 SecureHash

The SecureHash plugin provides access to an implementation of the RFC6324 secure hash routines. This is an extract from the description:

> This file implements the Secure Hash Algorithms as defined in the U.S. National Institute of Standards and Technology Federal Information Processing Standards Publication (FIPS PUB) 180-3 published in October 2008 and formerly defined in its predecessors, FIPS PUB 180-1 and FIP PUB 180-2.
>
> A combined document showing all algorithms is available at
> http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf
>
> The five hashes are defined in these sizes:

SHA-1	20 byte / 160 bit
SHA-224	28 byte / 224 bit
SHA-256	32 byte / 256 bit
SHA-384	48 byte / 384 bit
SHA-512	64 byte / 512 bit

Modified versions of the RFC6324 routines – to make them thread-safe – are included in the source code of the plugin: no external libraries are required.

Access to the hashing routines is provided at two levels

- The higher (and most convenient) level is provided by the SecureHash class using the Sha method.
- At the lower level, the individual RFC6324 routines may be accessed directly, without using the SecureHash class, via the interface procedures provided by the plugin.

Parameters to Sha may be strings, csets, open files, numbers, records or lists.

String parameters are "fed to the underlying hash routines" (*fttuhr*);
if the parameter is a cset, each character in the set is *fttuhr*;
if the parameter is an open file, each line of the file is read and *fttuhr*;
if the parameter is a number, it is converted to a string and *fttuhr*.

Sha applies itself recursively to list or record parameters (depth first traversal). So records and lists may contain strings, csets, open files, numbers, records or lists. Sha is tolerant of null parameters and empty strings, csets, files or lists; they have no effect on the final hash value.

Sets and Tables are not allowed as parameters to Sha because their order of enumeration is not defined. Any other type (thread, co-expression, window ...) is also disallowed by

fiat, with one exception: A procedure value[1], using special "command procedures", may be used to signal that Sha should take some special action as follows:

- The procedure More is allowed as the final parameter. Normally the Sha method returns the secure hash of all its parameters, but if the final parameter is the procedure More, subsequent calls to Sha will continue the hashing operation. Note the style is

  ```
  Sha( ... , More )
  ```

  rather than

  ```
  Sha( ... , More() )
  ```

  although the latter has been made to work as a "concession to ease of use".

- The procedures Final1 ... Final7 may be used to signal that the final parameter to Sha is not a whole octet, only the specified number of bits are to be included in the hash. These procedures are only allowed just before the final parameter. Where the final number of bits is calculated, the Final(expr) procedure may be used. 0 <= expr <= 8. (There is a Final0 procedure, but using it explicitly would be slightly odd, since it will cause the final parameter to be ignored. There is also a Final8 procedure which causes all of the final parameter to be included. These procedures are intended for use by the Final procedure).

- The Raw procedure switches the default output from a string of hexadecimal characters to a string of half the length containing the actual bits returned by the RFC6234 result procedure. It may be placed anywhere in the list of parameters.

Note that Sha(number) will return a secure hash of the string representation of the number, not a hash of the underlying bits. If a hash of the bits is required, the only way to do it is to convert the number to a string of the correct length (and endianness) without altering the value of the bits – Unicon strings may contain characters with all values from 0 to 255 – this also applies to the lower level sha_Input procedure

Once Sha has produced a hash value (or failed) it will automatically reset the underlying hash routines the next time it is called.

The SecureHash class is not thread-safe. Using a shared SecureHash object in different threads without mutual exclusion is unlikely to produce predictable results. Using a SecureHash object that is private to each thread *is* safe because the underlying hash routines are thread-safe.

[1] A procedure value is a convenient way to steer the operation of the Sha routine because it cannot be confused with any data to be added to the hash value.

The class is initialized with an optional string parameter that determines the hash algorithm. Valid strings are "SHA1", "SHA224", "SHA256", "SHA384" or "SHA512". The default is "SHA512".

The methods of the SecureHash class are:

Reset(s : "SHA512") : ? **Reset Secure Hash**

Reset(s) re-initializes the secure hash instance. The optional parameter specifies the algorithm to use and must be one of "SHA1", "SHA224", "SHA256", "SHA384" or "SHA512".

Sha(any?, ...) : string ? **Secure Hash**

Sha(...) returns the secure hash value of its arguments, which are of the types discussed above.

The interface procedures of the SecureHash plugin are:

---

shaFunction(s?) : string ? **Set/Get default Hash function**

shaFunction(h) sets the default hash algorithm. Valid strings are "SHA1", "SHA224", "SHA256", "SHA384" or "SHA512". shaFunction() returns the name of the current default algorithm. The procedure is not thread-safe.

---

sha_Reset(s) : ctx ? **Initialize Secure Hash**

sha_Reset(h) initializes and returns an opaque context value that should be fed into the other interface procedures. h determines the hash algorithm. Valid strings are "SHA1", "SHA224", "SHA256", "SHA384" or "SHA512".

---

sha_Input(ctx, s) : ? **Hash String**

sha_Input(x, s) Adds the secure hash of s to the hash value stored in x.

---

sha_FinalBits(ctx, c, i) : ? **Hash the final partial octet**

sha_FinalBits(x, c, n) Adds the secure hash of the most significant n bits of c to the hash value stored in x. 1 <= n <= 7

---

sha_RawResult(ctx) : string ? **Return the raw hash value**

sha_RawResult(x) returns the final hash value in the exact form returned by the RFC6324 hash routines. The returned string will be a binary string and may contain any character value from char(0) to char(255).

---

sha_Result(ctx) : string ? **Return the hash value**

sha_Result(x) returns the final hash value converted to a string of hexadecimal characters.

G.3.3 SQLite

The SQLite plugin provides access to version 3 of the SQLite database engine. The SQLite software must be downloaded from https://sqlite.org/download.html (or from a mirror or a distribution specific site) and installed: the plugin does not provide it.

SQLite is a *transactional* database: all reads and writes take place within a transaction – either explicitly started by the application, or implicitly by SQLite itself – and a transaction is *atomic*; either all of the modifications in the transaction succeed or none of them do. SQLite supports many readers but only one writer at a time. To resolve conflicts between incompatible access requirements, requests are queued. Sometimes the error SQLITE_BUSY is returned to signal that the database is in use and to try again later. See https://sqlite.org/transactional.html and https://sqlite.org/lang_transaction.html for more details.

The plugin gives access at three levels:

1. The highest level is a simple class, built on the routines in level 2, that encapsulates an SQLite database connection (and, to a large extent, handles SQLITE_BUSY). SQLite is transactional but SQLite transactions may not be nested. The class provides limited support for what *appears* to be nested transactions allowing calls to start and finish a transaction to be nested[2]– which allows routines that use a transaction to be called from other routines that themselves use a transaction. In reality, there is only ever one (the outermost) transaction.

   There is also a derived class (RO_SQLite)that provides a "read-only" interface, which allows reading from the database but prohibits its alteration.

2. The next level consists of utility procedures that use the lowest level routines to return the rows of an SQL query as a Unicon data structure (list, set, table). This level makes no attempt to deal with SQLITE_BUSY, passing that error back to the caller to be dealt with there.

3. The lowest level provides access to the SQLite API. There are almost 300 routines in the API and, at present, this level only provides access to the routines that are used by the higher levels of the plugin. However, given the examples provided here, it is comparatively easy to extend this level to provide a routine that is missing.

The thread safety of the SQLite plugin is a complicated question. In summary, the external SQLite library itself *is* thread-safe but the plugin is not; however, it can be used in a thread-safe manner with a little extra effort. See below (page 527) for more detail.

The SQLite class

The class is initialized with a string parameter that is the file name of the database together with an optional timeout parameter – discussed below on page 528 – the default is 5 seconds.

[2] Methods like SQL_As_List use transactions internally to guarantee the consistency of the answer.

A simple example of the use of the SQLite class may be found in the testSQLite.icn file that accompanies the source code of the plugin. Using the class methods, the program creates a simple database of squares, cubes and fourth powers of the numbers from one to ten. It then checks the values returned by other methods of the class.
The methods of the SQLite class are:

BEGIN() : ? **Start a database transaction**

BEGIN() declares the start of an SQLite transaction: alterations to the database between BEGIN() and END() will either *all* succeed or, none of them will succeed. Calls to BEGIN may be nested for convenience but note that SQLite does not support nested transactions: from the point of view of the database software there is only ever one transaction – the nested transactions are a fiction provided by the SQLite class so that functions which, in isolation, call for a transaction may be conveniently amalgamated into a single overall transaction. After a successful call of BEGIN the SQLITE_BUSY status will not be returned by any SQLite routine up to (but not including) the corresponding call of END.

END() : ? **Complete a database transaction**

END() declares the end of an SQLite transaction. If the call succeeds, all modifications to the database after the call of the corresponding call to BEGIN will have succeeded.

ROLLBACK() : ? **Abandon a database transaction**

ROLLBACK() declares the end of an SQLite transaction. If the call succeeds, all modifications to the database after the call of the outermost call to BEGIN will have been undone and the state of the database will be as it was just before the call to BEGIN.

Close() : **Close a database**

Close() closes a database connection and recovers resources. If Close is called during a transaction, the transaction is automatically rolled back. It is important to call Close before exiting the program: do not rely on the Unicon system to close the connection for you.

Exec(s, ...) : row|val? **Execute SQL**

Exec(sql) prepares the SQL query in sql and then uses the SQLite library to execute it. The query should return, at most, one row of data. If the query asks for more than one value the values will be returned in a record whose field names are the names used in the query. A query that asks for a single value results in that value being returned directly (rather than a record with a single field). A query that results in no data being returned will either succeed or fail. Exec(sql, p1, p2 ...) will bind the parameters to the query before execution.

ErrMsg() : string **Return the most recent error message**

ErrMsg() returns a string (in English) that describes the most recent error detected by the SQLite database routines.

---

isTable(s) : ? **Test if a SQL table exists**

isTable(t) succeeds if the SQL table t exists in the database.

Rows(s) : integer ? **Count rows in a SQL table**

Rows(t) returns the number of rows in the SQL table t.

SQL_Row(s, ...) : row * **Get data from a query**

SQL_Row(sql, ...) prepares the query in sql and then returns the data one row at a time. The data is in the same form as Exec – a record with named fields or a single value. Note that calling SQL_Row fewer times than there are available rows will leak memory until the database connection is closed.

SQL_As_List(s, ...) : list ? **Return query data as a list**

SQL_As_List(sql, ...) prepares the query in sql and returns the results in a list. The list elements will either be records with named fields or single values (same format as Exec).

SQL_As_Set(s, ...) : set ? **Return query data as a set**

SQL_As_Set(sql, ...) prepares the query in sql and returns the results in a set. The set elements will either be records with named fields or single values (same format as Exec). Note that queries that return a single value will have duplicates removed but queries that return a record won't because all records are unique, even if the fields have identical values.

SQL_As_Table(s, i : 1, ...) : table ? **Return query data as a table**

SQL_As_Table(sql, n, ...) prepares the query in sql and returns the results in a Unicon table whose keys are taken from column n of the query data. The table element values will always be records, even if the data rows have only one value (and will include the indexing column value). If the indexing column has duplicate values then later rows will overwrite earlier rows with the same key.

Thread safety of the SQLite class

The main reason that the SQLite class is not thread-safe is down to how it handles transactions. SQLite itself supports multiple simultaneous read transactions coming from separate database connections, possibly in separate threads or processes, but only one simultaneous write transaction. Each instance of the SQLite class counts how many BEGIN methods are active on its connection and only isues a BEGIN IMMEDIATE TRANSACTION statement for the outermost (first) call and only issues a COMMIT statement when the outermost END method is called. If the instance is shared between threads then it is easy for this count to become confused, leading to a COMMIT statement being executed at an inappropriate point.

To avoid this, use a separate instance of the class in each thread. Doing so avoids most (but not all) problems. If concurrent access is made to the database from separate threads then it is possible that the SQLITE_BUSY error is returned. The BEGIN and END methods handle this by delaying for a short time and retrying but they will fail if the total time taken

is more than the wait time specified in the initialization of the class. Hence a program that uses concurrent access and has transactions that might take a long time must be prepared to deal with the failure.

An alternative design which avoids these problems is to use a single database access thread to serialize all SQL queries and to use Unicon's message passing facilities to send queries and receive answers. If there is a mixture of read and write requests the reduction in parallelism (compared with one class instance per thread) is not as drastic as it appears because the reduction in parallel queries has *already* occurred due to the way the SQLite class uses transactions – each method uses BEGIN and END internally to make sure that methods like SQL_As_List return a consistent set of data and don't fail in the middle of constructing the list. However, if most of the queries are read only, this design will reduce the number of concurrent queries dramatically (to one).

There is, in essence, an unfortunate trade-off between performance and convenience: for maximum throughput the only way is to eschew the SQLite class, use the utility procedures or the low level interface routines explicitly, do as much as possible in parallel and deal with SQLITE_BUSY wherever it occcurs. In practice, the SQLite class performs tolerably well and it is only those applications that demand the best performance and the highest possible level of parallel access to the database that may feel the need to replace it. In those circumstances, it might also be fruitful to reconsider the use of an interpreted language.

The SQLite utility procedures

There is a deliberate similarity in names between the methods of the SQLite class and the utility procedures used by them. The functionality is largely the same with the significant exception that the utility procedures make no attempt to handle SQLITE_BUSY – that error is passed back to the caller to be dealt with there. In many cases, the class method just wraps the utility procedure it uses in BEGIN ... END calls.

The utility procedures of the SQLite plugin are:

SQLi_isTable(s) : ? **Test if a SQL table exists**

SQLi_isTable(t) succeeds if the SQL table t exists in the database.

SQLi_Rows(s) : integer ? **Count rows in a SQL table**

SQLi_Rows(t) returns the number of rows in the SQL table t.

SQLi_Exec(s, ...) : row|val? **Execute SQL**

SQLi_Exec(sql) prepares the SQL query in sql and then uses the SQLite library to execute it. The query should return, at most, one row of data. If the query asks for more than one value the values will be returned in a record whose field names are the names used in the query. A query that asks for a single value results in that value being returned directly (rather than a record with a single field). A query that results in no data being returned

will either succeed or fail. SQLi_Exec(sql, p1, p2 ...) will bind the parameters to the query before execution.

SQLi_Row(s, ...) : row * Get data from a query

SQLi_Row(sql, ...) prepares the query in sql and then returns the data one row at a time. The data is in the same form as SQLi_Exec – a record with named fields or a single value. Note that calling SQLi_Row fewer times than there are available rows will leak memory until the database connection is closed.

SQLi_As_List(s, ...) : list ? Return query data as a list

SQLi_As_List(sql, ...) prepares the query in sql and returns the results in a list. The list elements will either be records with named fields or single values (same format as SQLi_Exec).

SQLi_As_Set(s, ...) : set ? Return query data as a set

SQLi_As_Set(sql, ...) prepares the query in sql and returns the results in a set. The set elements will either be records with named fields or single values (same format as SQLi_Exec). Note that queries that return a single value will have duplicates removed but queries that return a record won't because all records are unique, even if the fields have identical values.

SQLi_As_Table(s, i : 1, ...) : table ? Return query data as a table

SQLi_As_Table(sql, n, ...) prepares the query in sql and returns the results in a Unicon table whose keys are taken from column n of the query data. The table element values will always be records, even if the data rows have only one value (and will include the indexing column value). If the indexing column has duplicate values then later rows will overwrite earlier rows with the same key.

The SQLite interface routines

The plugin uses a small subset of the three hundred or so routines in the SQLite API. There is comprehensive documentation on each of the individual routines provided by the SQLite library at https://www.sqlite.org/docs.html and no attempt is made to reproduce that material here. Each of the interface procedures provided by the plugin checks that the supplied parameters are of the correct type, converts them from their Unicon representation into something that conforms to the C calling convention and passes them down to the underlying SQLite library routine.

It is anticpated that most users will not call these routines directly; instead, preferring to use the SQLite class interface or the higher level utility procedures of the plugin.

Interface procedure	SQlite routine	Notes
SQLi_Init(x?) :	sqlite3_config	This procedure should be called before any other interface routine. If the parameter is null, column numbers will start at one. If non-null they will start at zero. The higher level routines and the SQlite class assume that the column numbers start at one.
SQLi_libversion() : s	sqlite3_libversion	Returns the version of the SQLite library as a string.
SQli_libversion_number() : i	sqlite3_libversion_number	Returns the version of the SQLite library as an integer.
SQLi_open(s, s?) : ctx?	sqlite3_open_V2	Open (or create) the database named by the first parameter. The second parameter is "b" for read/write access and anything else for read-only access.
SQLi_close(ctx) :	sqlite3_close_V2	Close the database connection. It is important to call SQLi_close before exiting the program: do not rely on the Unicon system to close the connection for you.
SQLi_prepare(ctx, s) :sqst?	sqlite3_prepare_V2	Compile (prepare for execution) the SQL statement s.
SQLi_bindArg(sqst, i, x) : ?	sqlite3_bind_null sqlite3_bind_int64 sqlite3_bind_double sqlite3_bind_text	Bind a single parameter x to the column specified by i. x may be null, integer, real or a string.
SQLi_bind(sqst, x, ...) : ?	sqlite3_bind_null sqlite3_bind_int64 sqlite3_bind_double sqlite3_bind_text	Bind parameters, starting at the first column. Each parameter may be null, integer, real or a string.
SQLi_step(sqst, i: 0) : ?	sqlite3_step	Evaluate (execute) the prepared statement. A status return of SQLITE_BUSY will cause the program to stop unless i is non zero.

continued . . .

Interface procedure	SQlite routine	Notes
SQLi_errmsg(ctx) : s SQLi_Error(i) : s	sqlite3_errmsg sqlite3_errstr	Return the most recent error message. SQLi_Error(n) returns the error message associated with the return status n.
SQLi_column_count(sqst) : i	sqlite3_column_count	Return the number of columns in the result set returned by the prepared statement.
SQLi_column_type(sqst, i) : i	sqlite3_column_type	Return the data type code of the specified column in the result set returned by the prepared statement.
SQLi_column_name(sqst, i) : s	sqlite3_column_name	Return the assigned name of the specified column in the result set returned by the prepared statement.
SQLi_column_string(sqst, i) : s	sqlite3_column_text	Return the value of the specified column as a string.
SQLi_column_integer(sqst, i) : i	sqlite3_column_int64	Return the value of the specified column as an integer.
SQLi_column_real(sqst, i) : r	sqlite3_column_double	Return the value of the specified column as a real number.
SQLi_column(sqst, i) : x	sqlite3_column_type sqlite3_column_int64 sqlite3_column_double sqlite3_column_text	Return the value of the specified column as specified by its data type code.
SQLi_finalize(sqst)	sqlite3_finalize	Recover resources from a prepared SQL statement after execution.

Although the SQLite documentation is generally not repeated here, it is worth emphasizing this extract:

> The application must finalize every prepared statement in order to avoid resource leaks. It is a grievous error for the application to try to use a prepared statement after it has been finalized. Any use of a prepared statement after it has been finalized can result in undefined and undesirable behavior such as segfaults and heap corruption.

Bibliography

[Alg12] Jafar Al Gharaibeh. *Programming Language Support for Virtual Environments.* Ph.D. Dissertation, University of Idaho, 2012.

[And83] Gregory Andrews and Fred Schneider. *Concepts and Notations for Concurrent Programming.* ACM Computing Surveys. 15:1, March 1983, pp 3-43.

[Berk82] Berk, T., Brownstein, L., and Kaufman, A. *A New Color-Naming System for Graphics Languages.* IEEE Computer Graphics & Applications, pp 37-44, May 1982.

[Boehm88] Barry W. Boehm. *A Spiral Model of Software Development and Enhancement.* IEEE Computer, vol. 21 no. 5, pp 61-72. 1988.

[But97] David R. Butenhof. *Programming with Posix Threads.* Addison Wesley, 1997.

[Clark85] D. D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the Tenth ACM Symposium on Operating System Principles*, pages 171–180, Dec. 1985.

[Elm89] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Redwood City, CA. 1989.

[Eri98] Hans-Erik Eriksson and Magnus Penker. *UML Toolkit.* Wiley, New York, NY. 1998.

[Fol95] James D. Foley, Andries Van Dam, Steven K. Feiner, and John F. Hughes. *Fundamentals of Interactive Computer Graphics: Principles and Practice in C.* Reading, MA: Addison-Wesley Publishing Company, 1995.

[Gai39] Helen Fouche Gaines. *Cryptanalysis: a Study of Ciphers and Their Solution.* Dover 1939.

[Gris86] R. E. Griswold and M. T. Griswold. *The Implementation of the Icon Programming Language.* Princeton University Press, Princeton, New Jersey, 1986.

[Gris96] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language, 3rd ed.* Peer-to-Peer Communications, San Jose, CA. 1996.

[GJT98] Ralph E. Griswold, Clinton L. Jeffery, and Gregg M. Townsend. *Graphics Programming in Icon.* Peer-to-Peer Communications, San Jose, CA. 1998.

[GPP71] Griswold, Poage, and Polonsky. *The SNOBOL 4 Programming Language, 2nd ed.* Englewood Cliffs, N.J. Prentice-Hall, Inc. 1971.

[Jeff99] Clinton L. Jeffery. *Program Monitoring and Visualization: An Exploratory Approach.* Springer-Verlag, New York, NY. 1999.

[Lev90] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly and Associates, Sebastopol, CA, 1990.

[Ogle90] D. M. Ogle, K. Schwan, and R. Snodgrass. *The Dynamic Monitoring of Distributed and Parallel Systems.* Technical Report GIT-ICS-90/23, School of Information and Computer Science, Georgia Institute of Technology, Dec. 1990.

[Woo99] Woo, Mason; Neider, Jackie; Davis, Tom; Shreiner, Dave. *OpenGL Programming Guide: the Official Guide to Learning OpenGL, Third Edition.* Addison-Wesley, 1999.

[Shr00] Shreiner, Dave. *OpenGL Programming Guide: the Official Reference Document to OpenGL, Third Edition.* Addison-Wesley, 2000.

[Pax95] Vern Paxson. Flex: A fast scanner generator, ed. 2.5, www.gnu.org/manual/flex/, 1995.

[Ram94] Norman Ramsey. *Literate programming simplified.* IEEE Software, 11(5):97-105, September 1994.

[Town89] R. E. Griswold and G. M. Townsend. *The Visualization of Dynamic Memory Management in the Icon Programming Language.* Technical Report 89-30, Department of Computer Science, University of Arizona, Dec. 1989.

[Wall91] Larry Wall and Randall Schwartz. *Programming Perl.* O'Reilly and Associates, Sebastopol, CA, 1991.

Index

8-Queens problem, 37

Abort, 409
abs, 383
absolute value, 383
abstract, 374
abstract class, 222
access, 177
    functions, 88, 185
    structure, 173
    variable, 87
access time, set, 400
access, file, 72
acos, 383
action, statechart, 240
activate co-expression, 381
Active, 401
activity, statechart, 240
actor, 237
addition, 14
address space, 177
aggregation, 215
Alert, 401
Alexander, Bob, 46
aliasing, 367
alternation operator (|), 19, 53, 57, 290, 381, 459
alternation, repeated, 379
analysis
    post mortem, 176
    runtime, 176
anchored pattern match (=p), 379
AND operator, 58
Any, 409
any, 383
any(), 384
any(c), 46
APL, 3
apply operator, 58
apply(), 430
Arb, 409
Arbno, 409
arc cosine, 383
arc sine, 384
arc tangent, 384
args, 384
args(p), 384
argument, 13
argument culling, 191
arithmetic, 14
arithmetic operator, 380
array, 31, 68
ASCII, 16, 61, 355, 412, 459, 472, 474, 475, 477
ASCII, &ascii, 16, 368
asin, 384
assignment, 15, 39, 367, 380
    augmented, 379
    reversible, 380
    substring, 383
association, 195, 215, 237
associative memory, 30
atan, 384
atanh, 384
atomic types, 13
attribute
    class, 193
    GUI widget, 310
augmented assignment, 15
automatic instrumentation, 172
automatic storage management, 29

backtracking, 37, 41, 46

backups, 270
Bal, 409
bal, 384
bal(), 46, 54
balance string, 384
band, 521
bang
    binary, 381
basename, 431
bcom, 521
BEGIN, 526
Bg, 401
binary data, 432, 443
binary operator, 15, 379
binomial coefficient, 449
bit, 521
bit extraction, 521
bit rotation, 521
Bitman plugin, 520
bits, 521
BitString, 432
bitwise and, 389, 521
bitwise or, 390, 521
Boehm, Barry, 1
bor, 521
Border, 499
bounded expressions, 19, 383
bowdlerize, 449
Break, 409
break expression, 22, 374
Breakx, 409
brot, 521
bshift, 521
built-in functions, 383
Button, 497
button, 308
ButtonGroup, 498
by, to-by step, 374

call depth, 370
call, procedure, 382
callback, 177
Cameron, Mary, 353
Cartesian coordinates, 214
case expression, 21, 374, 466
ceil(r), 451
center, 384
center(), 384
CGI, 245, 433
channel, 384
char, 384
character, 16, 42, 384
character set, 13, 15
    event mask, 173
chart parser, 442
chdir, 384
chdir(), 71, 384
check boxes, 315
CheckBox, 501
CheckBoxGroup, 501
CheckBoxMenuItem, 505
chmod, 384
chmod(), 73, 384
chown(), 73
class
    abstract, 222
    declaration, 195, 374
    diagrams, 193
classname, 385
client, 69, 78, 281
client/server, 281
Clip, 401
clock tick, 182
Clone, 401
Close, 526
close, 385
close file, 385
close(), 70
closure, star(s), 441
co-expression, 59, 385
code reuse, 223
cofail, 385
collaboration diagram, 241
collate, 464
collect, 385
collect garbage, 385

Color, 401
color, 304
ColorValue, 401
column
    ODBC, 386
column number, 368
combinations, string, 464
command-line, 256
comment, 14, 232, 267, 460, 465, 472, 477, 479, 484
communication
    tool, 184
comparison operator, 17
    string, 42
compile, 7, 27, 230, 411
compile time, 217
compiler, 24, 201, 399, 411
complement, cset, 379
complete(), 435
complex numbers, 435
Component, 494
compound expression, 383
concatenation, 381
concordance, 47
condition, 17
condition variable, 385
conditional assignment, 382
conditional expression, 15
condvar, 385
conjunction &, 58, 381
constructor, 204, 299, 310, 366, 385, 456
    class, 196, 215, 224, 376
    dynamic record type, 101
    record, 33, 201, 456
Container, 497
container, GUI class, 317
context switch
    lightweight, 178
context-free grammar, 52
control flow, 177
control structure, 13, 20, 41, 57, 60, 363, 368, 374
    programmer defined, 383
conversion
    type, *see* type, conversion
conversion, type, 378
convert
    base, 435
    degrees to radians, 387
    radians to degrees, 396
    to cset, 386
    to integer, 390
    to number, 392
    to real, 395
    to string, 398
copy, 386
copy(x), 35, 386
CopyArea, 401
cos, 386
cos(), 14
cosine, 386
Couple, 401
create, 59, 375
    directory, 392
    set, 397
critical section, 375
cross-reference, 478, 480
cset, 13, 15, 365, 386
    event mask, 173
cset literal, 16, 43, 365
cset membership, 384
cset, universal &cset, 16, 369
ctime, 386
ctime(i), 386
current co-expression, 369
cursor position assignment, 382

DAG, 466
database, 90, 91, 291, 366, 385, 437, 475, 508
date, 369
date comparison, 436
days of the week, 441
dbcolumns, 386
dbdriver, 386
dbkeys, 386
dblimits, 386

DBM, 93, 388, 393, 412
dbproduct, 387
dbtables, 387
deepcopy(), 35
default
    case branch, 375
    scanning parameters, 44
default parameters
    string scanning functions, 383
default value
    table, 30
default, parameter, 23
define symbols, 411
delay, 387
delay(i), 387
delete, 387
delete element, 387
delete()
    DBM database, 93
    from POP mail, 83
    list, 32
    set, 34
    table, 31
deque, 31
dereference, 93, 379
design patterns, 223
detab, 387
Dialog, 493
dialog, 307
diff, 437, 474
difference (c1 -- c2), 16
difference (S1--S2), 34
difference (T1--T2), 31
digits, &digits, 16
digits, cset &digits, 369
directories, 69
directory, 71, 256, 384
    create, 392
disk usage, 264
display, 70, 387
display(i,f), 387
division, 14, 451
do clause, 17
do, iteration, 375
DrawArc, 402
DrawCircle, 402
DrawCube, 402
DrawCurve, 402
DrawCylinder, 402
DrawDisk, 402
DrawImage, 402
DrawLine, 402
DrawPoint, 403
DrawPolygon, 403
DrawRectangle, 403
DrawSegment, 403
DrawSphere, 403
DrawString, 403
DrawTorus, 403
driver manager
    ODBC, 96
DropDown, 503
dtor, 387
dtor(r), 387
dynamic loading, 172, 178

e, 2.71... &e, 14
E_MXevent, 183, 185
E_Opcode, 181
E_Pcall, 173
EBCDIC, 16, 412, 477, 478
EditList, 504
editor, 8, 199, 206, 474, 476, 484, 485, 487
elaboration, 237
elapsed time, 372
else, 375
empty list, 32, 382
encapsulation, 190
encryption, 473
END, 526
end, 375
end-of-file, 16
entab, 387
environment variable, 389
    CGI standard, 247
    DateBaseYear, 436

EDITOR, 487
IPATH, 229, 429
IPL, 479
LPATH, 429, 478
PATH, 442
TERMCAP, 445
TRACE, 372
USER, 288
equal
numeric =, 380
reference ===, 380
string ==, 380
equivalence
set, 462
EraseArea, 403
errno, keyword, 74
error
convert to failure, 369
floating point, 74
message, &errortext, 369
standard file &errout, 370
standard, &errout, 70
system, 417
errorclear, 387
errorclear(), 387
errortext, keyword, 71, 73
escape codes, 43
escape sequence, 438
eval(), 438
evdefs.icn, 179
Event, 404
event, 172
artificial, 183
categories, 181
code, 172, 173
mask, 173
pseudo, 183
report, 173
source, 179
value, 172, 173
virtual, 183
event code
program execution, 370
event driven, 103
event driven programming, 184
event mask, 173
event stream
dual, 179
event value
program execution, 370
event(), 183
event, statechart, 240
eventmask, 387
eventmask(ce), 388
every, 20, 375
EvGet, 388
EvGet(), 180, 185
EvInit(), 179
evinit, 186
evinit library, 179
EvSend, 388
EvSend(x, y, C), 388
EvTerm(), 179
exclusive or, 390
Exec, 526
execution
control, 176
execution control, 183
execution model
multitasking, 176
execution monitor
Alamo, 174
exit, 388
exit(i), 388
exp, 388
exp(x), 14
exponential, exp(r), 388
exponentiation ^, 14
expression, 13
expression failure, 7, 11, 17, 21, 26, 30, 39, 53, 57, 77, 363, 375
&fail, 370

factorial(n), 439
Fail, 409
fail, 28, 54, 375

co-expression, 60, 385
expression, 18, 71
system call, 73, 413
fallible expression, 18
Farber, Dave, 474
Feature(s), 467
features, 370
Fence, 410
fetch, 388
fetch()
DBM database, 93
SQL, 98
fetch(d, k), 388
Fg, 404
field, record or class, 380
fieldnames, 388
fieldnames(r), 388
file, 69, 366, 443
close, 385
information, 71, 398
lock, 73, 388
modified time, 72
permissions, 384
position, 400
redirection, 75
remove, 396
rename, 396
file ownership, 72
file size, 72, 398
filename completion, 46
FillArc, 404
FillCircle, 404
FillPolygon, 404
FillRectangle, 404
filtering, 173
find, 388
find string, 388
find(), 19, 44
findre(), 440
finite state machine, 239
floating point, 14
flock, 388
flock(f,s), 388
floor(r), 451
flush, 389
flush(f), 389
Font, 404
for loop, 20
fork(), 509
form feed, 485
fractions, 457
FreeColor, 404
function, 389
execution monitoring, 88

garbage collection, 30, 182, 368, 440
garbage collector, 385
Gaussian distribution, 440
generate elements, 378
generate operator !x, 19, 378
generate sentences, 461
generator, 18, 21, 25, 28, 45, 46, 57, 60, 68, 363, 377, 382, 453, 456, 476, 479
function, 88
generator, random number, 456
genetic algorithms, 293
get, 389
getch, 389
getche, 389
getenv, 389
getpid(), 76
getserv(), 78
gettimeofday, 389
getuid(), 76
global, 228, 367, 375, 389
global variables, 24
globalnames, 389
goal-directed evaluation, 16
golden ratio 1.618..., &phi, 14
golden ratio, &phi, 371
GotoRC, 405
GotoXY, 405
grammar, 52
graph, 36, 205, 442
graphical user interface, 307
grep, 440, 478

gtime, 389

halt, 398
Hanoi, 26
hashing, 260
heap, 30
    EM separate from TP, 177
hexadecimal, 441, 442
host machine name, &host, 370
HTML, 82, 232, 245, 433, 442, 472, 479
HTTP, 82, 246
HTTPS, 83
hyperbolic functions, 449

iand, 389
icom, 389
Icon, 499
Icon Program Library, 429
IconButton, 498
IdentityMatrix, 405
Idol, 190
if, 375
if statement, 17
if-then-else, 15
ifdef symbol, 411
Image, 499
image, 389
image(), 79
image(x), 389
immediate assignment, 382
immutable values, 13
import, 375
include, 411
independence
    of EM and TP, 177
index, subscript, 378
infinite loop, 377
infinity, 16
inheritance, 191, 203, 223
    multiple, 205
    semantics, 205
initial, 375
initial clause, 24
initially, 196, 376
input, 69, 443
    standard file &input, 370
input stream
    dual, 179
input, standard &input, 70
insert, 390
insert(), 29, 390, 507
    DBM database, 93
    list, 32
    set, 34
    table, 31
install, 7
instance, 38, 59, 203, 215, 299, 497
    class, 215, 223, 241, 368, 376
    record, 33
    superclass, 206
instances
    class object, 196
instrumentation, 172
integer, 13, 14, 364, 390
integer(x), 14
integrated development environment, 3
interface builder, 353
Internet, 69, 77, 281
intersection (c1**c2), 16
intersection (S1**S2), 34
intersection (T1**T2), 31
inverse hyperbolic tangent, 384
invocable, 376
invocation, 382
    object method, 198
ior, 390
iplweb, 232, 479
IRC, 285
ishift, 390
isTable, 527
istate, 390
iterator, 21
Ivib, 353
ixor, 390

Java, 3, 189, 194, 232
JavaDoc, 232

JavaScript, 254
julian, 436

kbhit, 390
kbhit(), 390
key, 390
    duplicate, 443
    ODBC, 386
key(), 266
    table, 31
key(x), 390
keyboard, 10, 70, 76, 316, 357, 389, 390, 412, 446, 476, 497
keys, table, 30
keyword, 13, 390
keyword(), 390
KWIC, 479, 482

Label, 499
LaTeX, 474, 482
left, 390
left(), 390
Len, 410
length operator (*x), 16
letters, &letters, 16
letters, cset &letters, 370
lexical comparison, 18
library procedures
    monitor, 179
limitation \, 382
limiting an expression, 57
line number, 371
link, 37, 228, 376, 429, 478, 485
    association instance, 241
    association instance, 215
    file system, 256, 417, 513
    HTML, 434
    recursion, 256
    symbolic file, 513
link, file system, 72
Lisp, 3, 483
List, 503
list, 366, 390
    invocation, 58
    size (*L), 32
list concatenation
    L1 ||| L2, 33
list creation, 382
list data type, 31
list functions, 447
list(), 31
list(i, x), 390
literal, 13
literate programming, 4
literature, 483
load, 391
    C function, 391
    Unicon program, 391
load(), 391
loadfunc, 391
local, 367, 376, 391
local time, 386
local variables, 24
localnames, 391
lock, 391, 399
lock, file, 388
log, 391
log(x), 14
logarithm, 391
longest match, 449
loop, 12, 15
Lower, 405
lower case, 16, 48
lowercase, cset &lcase, 370
LU decomposition, 449

mail folder, read, 441
mail spool, 484
main task, 371
many, 391
many(), 391
many(c), 45
map, 391
map string, 391
map(), 63
match, 391
match string, 391

match(s), 45
matching functions, 44
matrix manipulation, 450
MatrixMode, 405
max, 391
max(), 14, 391
mceil(r), 451
mean values, 451
member, 391
member(), 392
    set, 34
    table, 31
membernames, 392
memory addresses, 30
memory allocation, 30, 201
memory monitor, 182
memory regions
    separate, 177
memory use, 368, 372
Menu, 504
menu, 319
menu bar, 315
MenuBar, 504
MenuButton, 504
MenuComponent, 504
MenuSeparator, 505
messaging, 82
method, 195, 376
    combination, 207
    invocation, 380
    overriding, 206
methodnames, 392
methods, 392
mfloor(r), 451
MIME, 431
min, 392
min(), 14, 392
mkdir, 392
mkdir(), 71, 392
modulo %, 14, 379
monitor
    anatomy, 184
    template, 184
monitor coordinators, 178
months of the year, 441
Morse code, 450, 484
move, 392
move(i), 44, 392
multi-bit extraction, 521
multidimensional array, 430
multiplication, 14
multiplicity, 216
MultMatrix, 405
mutable value, 366
mutable values, 29
mutex, 392
mutex(), 392
mutual evaluation, 382
MySQL, 96

n queens, 37
n-grams, 450
n-queens, 476, 487
name, 392
name mangling, 230
name space, 228
name(v), 392
natural log, &e, 369
natural log, log(x), 14
networking, 77
NewColor, 405
news, 484, 486
next, iteration, 376
nonnull test \x, 18, 23, 378
not, 58, 376
not equals, 11
NotAny, 410
Notification, 493
now, 371
Nowlin, Jerry, 46
Nspan, 410
null test /x, 18, 23, 378
null value, 371
null value, &null, 14
numeric, 392
numeric comparison, 18

numeric operations, 14

object, 366
object-oriented programming, 4, 189, 190, 200
octal, 441
ODBC, 95, 100, 386, 393, 508
ODBC driver, 386
of, 376
open, 393
open file, 393
open(), 70
operations, 193
operators, 378
opmask, 394
oprec, 394
options(), 37
options, command-line, 452
OR operator, 57
ord, 394
ordinal value, 394
outline
    of an EM, 180
output, 69, 443
    standard file &output, 371
output, standard &output, 70
overlay, GUI class, 319
OverlayItem, 506
OverlaySet, 506

package, 223, 228, 376
PaletteChars, 405
PaletteColor, 405
PaletteKey, 405
palindrome, 464
Panel, 506
parallel evaluation, 61
parameter, 22, 30
parameter list, 11
parameter names, 394
parameters
    wrong number, 22
paramnames, 394
parent, 394
parentheses, 382
parse, 52, 442, 485
password, 484
patch, 486
Pattern, 406
pattern, 365
pattern alternative operator (.|), 382
pattern match, 381
pattern matching, 16, 41, 52, 475
patterns, design, 223
patterns, SNOBOL4, 453
peername, 79
Pending, 406
Perl, 3, 534
permissions, file, 384
permissions, file access, 69
permutations, 63
phi 1.618..., &phi, 14
phi, golden ratio &phi, 371
pi, 3.14... &pi, 14, 371
pipe, 75, 394
pipe(), 394
Pixel, 406
Plugins, 520
plural form, 454
pointer, 30, 35, 442
polling, 183
polymorphism, 29, 192
polynomials, 454
POP, 83
pop, 394
pop(), 32
    from POP mail, 83
pop(L), 394
PopMatrix, 406
Pos, 410
pos, 394
pos(i), 394
position, string, 44, 371
POSIX, 69
POSIX extensions, 509
PostgreSQL, 96
PostScript, 456, 474, 486
power, exponent ^, 379

PowerBuilder, 3
predefined symbols, 412
preprocessor, 411
prime number, 439
private, 194
proc, 394
proc(s,i), 395
procedure, 7, 13, 22, 376
procedure invocation, 432
process, 74
producer/consumer, 62
program, 7
program behavior, 174
program design, 192
program name, 371
program state, 85
programmer defined control structure, 383
protocol, 281
prototype, 1
proxy pattern, 224
pseudo events, 184
pseudo-tty, 268
public, 194
pull, 395
pull(L), 32, 395
push, 395
push(), 32, 395
PushMatrix, 406
PushRotate, 406
PushScale, 406
PushTranslate, 406
put, 395
put(), 32, 395
Python, 3

qualifier, 217
QueryPointer, 407
queue, 29, 31, 366, 389, 446
quota, 263

radial coordinates, 214
radian, 14, 214
radio buttons, 315
Raise, 407
random
    number generator, 456
    number seed, 64, 371
    operator, ?x, 379
randomize(), 456
rapid application development, 3
rational numbers, 457
read, 395
read(), 11, 70
read(f), 395
ReadImage, 407
reads, 395
reads(f,i), 395
ready, 395
ready(f,i), 395
real, 395
real number, 13, 365
real(x), 14
receive, 395
receive datagram, 395
receive(), 80
record, 33, 366, 377
record constructor, 386
recursion, 26, 35, 477, 485
reduce(), 458
reference, 13, 22, 24, 30, 34, 194, 218, 224, 358
    documentation, 232, 245
    file, 70
    HTML, 442
    page, 482
reference comparison, 18
Refresh, 407
refresh, co-expression, 379
region sizes, 371
regular expression, 440, 458
Rem, 410
remainder, 15
remove, 395
remove directory, 396
remove file, 396
remove(s), 71
rename, 396
rename(), 71

repeat loop, 21, 377
repl, 396
replacement, string, 465
replicate list, 447
replicate string, 396
reserved word, 13, 374
Reset, 524
Reset(), 524
result, 13
return, 377
reverse, 396
reverse(x), 16, 396
reversible assignment, 39, 380
reversible swap, 380
REXX, 3
right, 396
right(), 396
rmdir, 396
role, 217, 237
ROLLBACK, 526
Roman numerals, 451
Rotate, 407
Rows, 527
Rpos, 410
Rtab, 411
rtod, 396
Ruby, 3
run, 7
    loaded program, 85
    program under monitor, 176
run-time error, 14
runerr, 396
runtime error, 396, 413
runtime system
    Unicon, 175

sampling, 185
Sapir-Whorf, 1
Scale, 407
scan string, 381
scanning
    environment, 44
    list, 447
scope, 229, 236, 367, 368
scope„ 24
screen, 8, 307, 389, 487
scripting languages, 3, 245
ScrollBar, 499
sectioning operator, 18
SecureHash plugin, 522
seek, 396
seek(f,i), 396
segment(s,c), 461
select, 396
select(), 76, 269, 396
self, 195
semicolon insertion, 383
send, 396
send datagram, 396
send(), 80
sensor, 172
sentinel value, 16, 28
seq, 397
seq(), 59
sequence of results, 21
sequence, generate numeric, 397
sequences, 441
serial, 397
serial(x), 397
server, 69, 78, 282
set, 366, 397
set data type, 34
set(), 397
setenv, 397
setenv(), 397
SGML, 457, 489
Sha, 524
Sha(), 524
sha_FinalBits, 524
sha_FinalBits(), 524
sha_Input, 524
sha_Input(), 524
sha_RawResult, 524
sha_RawResult(), 524
sha_Reset, 524
sha_Reset(), 524

sha_Result, 524
sha_Result(), 524
shaFunction, 524
shaFunction(), 524
shape equivalence, 438
shared address, 176
shell commands, 463
shift, 390
signal, 74, 397
signed shift, 521
Simula67, 190
sin, 397
sin(), 14
sine, 397
singleton, 223
size operator, 379
skeleton
    of an EM, 180
slice, 382
    list L[i:j], 33
    string s[i:j], 42
Smalltalk, 3, 189
SMTP, 83
SNOBOL4, 2, 41, 453
sort, 397, 445, 463
sort by field, 397, 463
sort source file procedures, 480
sort(x, i), 397
sortf, 397
sortt, 463
soundex, 463
source code, 8, 18, 90, 232, 298, 368, 429, 478
    line, 371
source code $line, 411
source file, 370
Span, 410
spawn, 397
spawn(), 397
special purpose monitors, 175
spiral model, 1
SQL, 94, 99, 291, 393, 508
    fetch, 388
sql, 397
sql(), 98, 398
SQL_As_List, 527
SQL_As_Set, 527
SQL_As_Table, 527
SQL_Row, 527
SQLi_As_List, 529
SQLi_As_Set, 529
SQLi_As_Table, 529
SQLi_Exec, 528
SQLi_isTable, 528
SQLi_Row, 529
SQLi_Rows, 528
SQLite Plugin, 525
sqrt, 398
sqrt(x), 14
square root, 398
stack, 26, 29, 31, 52, 366, 394, 472
stat, 398
stat(f), 71, 398
state names, U.S., 464
statechart, 239
static, 36, 377, 398
static variables, 24
static, and class, 367
staticnames, 398
stop, 398
string, 13, 15, 365, 398, 464
    balance, 384
    center, 384
    closure, 441
    comparison, 380
    concatenation s1 || s2, 16
    indexes 1 based, 41
    indexes 1-based, 16
    length (*s), 16
    literal, 16
    multi-line, 365
    position, 44
    scanning, 44, 460
    subject, 44
    subscript (s[i]), 16
string(x), 398
structure types, 22, 366, 471

subclass, 198, 203, 206, 227, 308, 328, 334, 493
subject string, 44
subject, string scanning, 372
SubMenu, 504
subscript, 382
subscript operator, 16, 18, 31, 34, 93
subsection, 382
substring, 42
subtraction, 14
Succeed, 410
success, 17, 20, 377
superclass, 203, 380
    operations, 206
suspend, 377
swap, 380
synchronous execution, *see* coroutine
syntax, 52
sys_errstr(i), 513
system, 398
system command, 398
system error, 417
system interface, 69
system(), 75, 265, 269

Tab, 410
tab, 399
tab(i), 44, 399
tab,GUI class, 317
tab/match (=s), 379
TabItem, 506
Table, 505
table, 366, 399
    initializer, 382
    length *T, 31
table data type, 30
table lookup, 30
table(x), 399
TableColumn, 505
TabSet, 506
talk, 285
tan, 399
tangent, 399
target program
    Alamo, 174
task, 84
task switch, 178
Tcl, 3
TCP, 77
template
    of an EM, 180
Texcoord, 407
TextButton, 497
TextField, 500
TextList, 502
Texture, 407
TextWidth, 407
then, 377
thing, 526
thread, 59, 377
thread safety, 363
time
    of day, 389
    since start, 372
time of day &clock, 368
time stamp, 369
to, generator, 19, 377
to-by, generator, 19
today &date, 369
tool communication, 184
Townsend, Gregg, 353
trace, 443
tracing, 27, 372, 476, 486
Translate, 408
trap, 399
trap signal, 399
trap(), 74
traverse, 226, 272
tree, 29, 35, 205, 228, 241, 256, 466
trigonometric function, 14
trim, 399
trim(s,c), 16
trim(s,c,i), 399
truncate, 399
truncate file, 399
trylock, 399
Turing machine, 490

two-way table, 466
type, 13, 399
    conversion, 182
type conversion, 23, 378
    type conversion, 14
type safe, 14
type(x), 399

UDP, 80
Ui, 8
umask, 73
UML, 192, 216
Uncouple, 408
undeclared variables, 24
undef, 412
union (c1 ++ c2), 16
union (S1++S2), 34
union (T1++T2), 31
unlock, 399
unsigned shift, 522
until, 21, 377
upper case, 16, 372
upto, 399
upto(c), 45, 400
Usage(s), 467
use case, 236
user
    Alamo, 174
    input, 183, 184
    interaction, 176
user interface, 307
ushift, 522
utime, 400
uuencode, uudecode, 478

value, 13
value masks, 180
Variable, 24
variable, 13, 30, 195, 204, 353, 367, 376, 383, 391, 400
    implicit, 195
    no. of arguments, 58
variable(), 87
variable(s,c,i), 400
verse, 490
version, 372, 467
very high-level language, 2, 190
virtual machine
    instruction, 181, 182
virtual monitor, 178
visibility
    within class, 194
VisibleContainer, 497
visitor pattern, 226
Visual Basic, 3
VRML, 468

wait, 400
wait(), 76, 400
Wampler, Steve, 37
WAttrib, 408
WDefault, 408
WFlush, 408
where(f), 400
while, 12, 377
wild-card patterns, 468
WindowContents, 408
wrap, 460, 469, 475
write, 400
write(), 10, 70, 400
WriteImage, 408
writes, 400
writes(), 400
WSection, 409
WSync, 409

xcodes, 92, 469
XML, 254