Unicon Language Reference

Clinton Jeffery, Shamim Mohamed, Jafar Al Gharaibeh

Unicon Technical Report: #8c

February 23, 2017

Abstract

Unicon is a very high level application programming language with particular strengths in the areas of complex data structure and algorithm development, text processing, graphics, network I/O, and concurrency. This language reference is adapted for online documentation purposes from Appendix A of "Programming with Unicon", by Jeffery, Mohamed, Al Gharaibeh, Pereda, and Parlett.

Unicon Project
http://unicon.org

University of Idaho
Department of Computer Science
Moscow, ID, 83844, USA

1 Introduction

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, the "Road Narrows" sign is also used to highlight a general comment about concurrency (rather than a specific thread-safety issue).

2 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 are of arbitrary precision. 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.

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.

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.

3 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. Mutual exclusion may be imposed for mutable values accessed by multiple threads, at a cost in performance.

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.

4 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.

5 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...

&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 gives the number of nesting levels to which program execution will be traced. 0 means no tracing. A negative value traces to an infinite depth. &trace is set outside the program using the TRACE environment variable or 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.

6 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.

---

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*.

---

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.

7 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.

= string **tab/match**

= pattern **anchored pattern 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.

. 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. 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 or csets. They may be augmented.

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 (in Chapter 8 of the Unicon book) 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**
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 (catalog, schema, tablename, colname, datatype, typename, colsize, buflen, decdigits, numprecradix, nullable, remarks) entries. 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 driver(name, ver, odbcver, connections, statements, dsn) that describes the details of the ODBC driver used to connect to database db. 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 record (columnname, sequencenumber) pairs 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 (name, 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 record (qualifier, owner, name, type, remarks) entries that describe all of the tables in the database associated with db.

---

delay(i) : null **delay for i milliseconds**
delay(i) pauses the program for at least i milliseconds.

---

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, elements xi are integer indices of the element 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).

---

exp(r) : real **exponential**

exp(r) produces the result of &e ˆ 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. Shifting is to the left if j>0, or to the right if j<0. j zero bits are introduced at the end opposite the shift direction.

---

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.

---

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 or table 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
"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.

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 infinite sequence i, i+j, i+2*j, ... j may not be 0.

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.

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 spasyswning the new process, and the return value is then the process id of the new process.

---

sys_errstr(i) : string? **system error string**

sys_errstr(i) produces the error string corresponding to i, a value obtained from &errno.

---

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[i2: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.

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.

8 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

_AMIGA	Amiga
_ACORN	Acorn Archimedes
_CMS	CMS
_MACINTOSH	Macintosh
_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
_ARM_FUNCTIONS	Archimedes extensions
_DOS_FUNCTIONS	MS-DOS extensions

9 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.

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 or two sets 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	invalid type to pattern operation
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
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
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 on directory or database
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
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
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

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
1214	cannot find certificate store
1215	cannot verify peer's certificate