`iyacc`: A Parser Generator for Icon

Ray Pereda

Unicon Technical Report UTR-03

February 25, 2000

Abstract

`iyacc` is software tool for building language processors. It is based on `byacc`, a well-known tool for the C programming language. This paper gives a brief description of how to use the tool.

The University of Nevada, Las Vegas
Department of Computer Science
Las Vegas, Nevada 89154, USA

1. Introduction

Parsing is the extraction of the grammatical structure of a sentence in some language. LALR(1) is a very general class of grammars for which iyacc can produce a parser. If this report is your first lex, you may wish to also read "Lex & Yacc," by [Levine92]. Also, the book by [Jeffery00] has a chapter on iyacc.

The iyacc program stands for Icon's Yet Another Compiler Compiler. It is a variant of a program called byacc (Berkeley YACC), modified to generate Icon code as an alternative to C. The iyacc program takes a grammar that you specify and generates a parser that recognizes correct sentences.

2. From Grammar to Parse Tree

A parser allows you to do more than just tell whether a sentence is valid according to the grammar, it allows you to record the structure of the sentence for later use. This internal structure explains *why* the sentence is grammatical, and it is also the starting point for most translation tasks. The structure is often a tree called a *parse tree*. The language used by iyacc to express the grammar is based on a form of BNF, which stands for Backus-Naur Form. A grammar in BNF consists of a series of production rules, each one specifying how a component of the parse is constructed from simpler parts. For example, here is a grammar similar to the one used to build the calculator:

```
assignment : NAME := expression
expression : NUMBER
| expression + NUMBER
| expression ??NUMBER
```

This grammar has two rules. The symbol to the left of the colon is called a *non-terminal* symbol. This means that the symbol is an abstraction for a larger set of symbols. Each symbol to the right of the colon can be either another non-terminal or a *terminal* symbol. If it is a terminal, then the scanner will recognize the symbol. If it is not, then a rule will be used to match that non-terminal. It is not legal to write a rule with a terminal to the left of the colon. The vertical bar means there are different possible matches for the same non-terminal. For example, an expression can be a number, an expression plus a number, or an expression minus a number.

The iyacc program's input files are structured much like iflex's input files. They have a definitions section, a rules section, and a procedures section. The definitions section includes code to be copied into the output file before the parser code, surrounded by %{and %}. This section also includes the declarations of each of the *tokens* that are going to be returned by the scanner. The term "token" is just another name for a terminal symbol, or a word. Tokens can be given a left or right associativity, and they are declared from lowest precedence to highest precedence. You may want to consult a reference on yacc for details on how these features are used. The next section is the rules section. The left and right sides of a rule are separated by a colon. The right side of the rule can be embedded with semantic actions consisting of Icon code that is surrounded by curly braces. Semantic actions execute when the corresponding grammar rule is matched by the input. The last section is the procedures section. In the calculator, there is one procedure, main(), that calls the parser once for each line.

3. A Complete Parser For The Desktop Calculator

The desktop calculator in calc.y

```
%{
## add any special linking stuff here
global vars
%}
/* YACC Declarations */
%token NUM, NAME, ASSIGNMENT
%left '-' '+'
%left '*' '/'
%left NEG /* negation--unary minus */
%right '^' /* exponentiation */
/* Grammar follows */
%%
input: /* empty string */
| input line
;
line: '\n'
| exp '\n' { write($1) }
| NAME ASSIGNMENT exp `\n' {
vars[$1] := $3
write($3)
}
;
exp: NUM { $$ := $1 }
| NAME { $$ := vars[$1] }
| exp '+' exp { $$ := $1 + $3 }
| exp '-' exp { $$ := $1 - $3 }
| exp '*' exp { $$ := $1 * $3 }
| exp '/' exp { $$ := $1 / $3 }
| '-' exp %prec NEG { $$ := -$2 }
| exp '^' exp { $$ := $1 ^ $3 }
| '(' exp ')' { $$ := $2 }
;
%%
procedure main()
  vars := table(0) # initialize all variables to zero
  write("iyacc Calculator Demo")
  repeat {
    write("expression:")
    yyparse()
  }
end
```

4. A Brief Overview of Semantics Actions

You can use single quoted characters as tokens without declaring them, so '+', '–', '*', '/', '^', '(', ')' are not declared in the above grammar. The unary minus makes use of the %prec keyword to give it priority over binary minus. Note that these are yacc notation for small integers; '+' is a shorthand for ord("+") here, not an Icon cset. Tokens NUM, NAME, ASSIGNMENT, and NEG are declared. When the parser matches a rule, it executes the Icon code associated with the rule, if there is any. Actions can appear anywhere on the right-hand side of a rule, but their semantics are intuitive only at the end a rule. Vertical bars actually mark alternative rules, so it makes sense to have a (possibly different) action for each alternative. The action code may refer to the value of the symbols on the right side of the rule via the special variables $1, $2... and they can set the value of the symbol on the left of the colon by assigning to the variable $$. The value of a terminal symbol is the contents of variable yylval, which is assigned by the scanner. The value of a non-terminal is the value assigned to the $$ variable in the grammar rule that produced that non-terminal. As was mentioned for the yylval variable, in other languages, assigning different kinds of values to $$ for different non-terminals is awkward. In Icon, it is very easy because variables can hold values of any type.

5. Basic Command-Line and iflex Interactions

Here is the sequence of commands you would type for creating a calculator given the iflex, (see [Pereda00]) and iyacc input files presented earlier. You might normally use a "make" program that manages the dependencies of different files and programs, to avoid typing this all by hand.

```
$ iyacc ?d calc.y creates calc.icn & y_tab.icn
$ iflex calc_lex.l creates calc_lex.icn
$ icont calc.icn calc_lex.icn creates the program calc*
```

The resulting calculator program might be executed in a session such as the following:

```
$ calc runs calc
iyacc Calculator Demo
expression: (3 + 5)*2 enter an expression
16 prints out the answer
expression: x := 16 enter an expression
16 prints out the answer
expression: 2 * x enter an expression
32 prints out the answer
expression: ^D control-D stop the program
```

References

[Jeffery00] Jeffery, C., Mohamed, S., Pereda, R. and Parlett, R., *Programming with Unicon: very high-level object-oriented application and system programming.* To be published.

[John75] Johnson, S. C., Yacc: Yet Another Compiler Compiler, *Computer Science Technical Report No. 32*, Bell Laboratories, Murray Hill, New Jersey (October 1975).

[Levine92] Levine, J. R., Mason, T., and Brown, D. *Lex & Yacc*, O'Reilly and Associates, Cambridge, Massachusetts, 1992.

[Paxson95] Paxson, Vern, flex – a fast lexical analyzer generator, many replication on the Internet. Also available on Redhat Linux via the man command.

[Pereda00] Pereda, Ray, *iflex – a lexical analyzer generator for Icon, Unicon Technical Report 00-01*, University of Nevada, Las Vegas, Las Vegas, Nevada, (February 2000).