iyacc: A Parser Generator for Icon and Unicon

Ray Pereda and Clinton Jeffery

Unicon Technical Report: 3a

2018-02-21

Abstract

iyacc is software tool for building language processors. It is based on byacc, a well-known tool for the C programming language. This report describes how to use the tool.

Unicon Project
http://unicon.org

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

1 Introduction

The iyacc program takes a context free grammar that you specify and generates a parser that recognizes whether an input conforms to that grammar. iyacc is a modified version of byacc (Berkeley YACC), modified to generate Icon code as an alternative to C. The name iyacc stands for Icons yacc, where yacc stands for Yet Another Compiler Compiler.

iyacc has been used in large production tools for quite some time, such as the Unicon translator itself. If this report is your first introduction to the yacc family of parser generator tools, you may wish to also read "Lex & Yacc", by John Levine [Levine92].

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 also allows you to preserve 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 ASGN expression ;
expression : NUMBER
   | expression + NUMBER
   | expression - NUMBER
   ;
```

This grammar has rules for constructing two symbols to the left of a colon; these symbols are called non-terminal symbols. Such non-terminal symbols denote an abstraction for a logically related sequence of symbols, analogous to a phrase (in natural language) or some part of a syntax construct (in a programming language), such as a statement, or an expression.

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 programs input files are structured into three sections, separated by double percent signs (%%): 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 Parser for a Desktop Calculator

The following calculator program is a minimalist example of using iyacc. Iyacc's generated parser reads tokens by calling a function yylex(). yylex() can be written by hand, or generated by a lexical analyzer generator program such as ulex [Ray03]. In the example, a very crude hand-written lexical analyzer is provided for completeness.

```
%{
## 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
procedure yylex()
   local tok
   static token_char, line
   initial {
```

```
      token_char := ~ ' \t'
      line := ""
   }
   if line == "" then line := (read()||"\n") | exit()

   line ? while tab(upto(token_char)) do {
      if yylval := tab(many(&letters)) then tok := NAME
      else if yylval := tab(many(&digits++'.')) then tok := NUM
      else if yylval := =":=" then tok := ASSIGNMENT
      else { yylval := move(1); tok := ord(yylval) }
      line := tab(0)
      return tok
      }

   tok := ord('\n')
   line := ""
   return tok
end
```

iyacc allows single quoted character constants to be used as tokens without declaring them, so '+', '', '*', '/', '^', '(', and ')' 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 such character constants are yacc's notation for small integers with the corresponding ASCII values; '+' in a grammar is a shorthand for the token that matches ord("+") here, not a cset. Tokens NUM, NAME, ASSIGNMENT, and NEG are declared.

4 Semantic Actions and Values

When the parser matches a rule, it executes the 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 global 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 iyacc, it is very easy because variables can hold values of any type.

5 Command-Line Usage

The preceding calculator example is standalone and can be compiled by just saying

```
iyacc -i calc.y
```

followed by invoking icont (or unicon) on the resulting calc.icn.

Here is the sequence of commands you would type for creating a calculator if you were using ulex to generate the lexical analyzer. This would mainly add a -d option to specify that iyacc should produce an include file, y_tab.icn, containing the $define's for the terminal symbols that yylex() must produce as return values when called by yyparse(). You would normally use a "make" program to manage the dependencies of these different files and programs, to avoid typing this all by hand.

```
$ iyacc -i -d calc.y              creates calc.icn & y_tab.icn
$ ulex calc_lex.l                 creates calc_lex.icn
$ unicon 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 (EOF) stops the program
```

References

[Jeffery00] Jeffery, C., Al-Gharaibeh, J., Mohamed, S., Pereda, R. and Parlett, R., Programming with Unicon: very high-level object-oriented application and system programming. unicon.org/book/ub.pdf
[Levine92] Levine, J. R., Mason, T., and Brown, D. Lex & Yacc, OReilly and Associates, Cambridge, Massachusetts, 1992.
[Ray03] Ray, K., Pereda, R., and Jeffery, C., Ulex A Lexical Analyzer Generator for Unicon, Unicon Technical Report 2a, unicon.org/utr/utr2/utr2a.pdf, May 2003.