YACC
("Yet Another Compiler Compiler")-
Lab Activity

mariaiuliana.dascalu@gmail.com

FILS, 2012

What is it?

- a tool used to write parsers/syntax analyzers designed Stephen C. Johnson at AT & T Bell Laboratories in 1980
- receives, as input, code in GDL (Grammar Description Language), which represents the description of a language syntax
- returns, as output, code in C of a LALR(1) parser, which can be taken by a C compiler to obtain the exe code

What does a parser do?

- calls the user-supplied low level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream; these tokens are organized according to some input structure rules, called grammar rules ;

- when one of the rules has been recognized, then user code supplied for that rule, an action , is invoked

- actions have the ability to return values and make use of the values of other actions

How it works? (1)

- The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the *lookahead token). The current state is* always the one on the top of the stack. The states of the finite state machine are given small integer labels; initially, the machine is in state 0, the stack contains only state 0, and no lookahead token has been read.

How it works? (2)

- The machine has only four actions available to it, called *shift , reduce , accept , and error .*
- Based on its current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs one, and does not have one, it calls *yylex to obtain the next token.*
- Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

Ambiguity and Conflicts (1)

- A set of grammar rules is *ambiguous if there is some input string that can be structured in two or* more different ways.
- *shift / reduce conflict*
- *reduce / reduce conflict*

Read more from “Yacc: Yet Another Compiler-Compiler””, by Stephen C. Johnson

Ambiguity and Conflicts (2)

- there are never any ‘‘shift/shift’’ conflicts
- when there are shift/reduce or reduce/reduce conflicts, Yacc still produces a parser: it does this by selecting one of the valid steps wherever it has a choice
- by default:
  - In a shift/reduce conflict, the default is to do the shift.
  - In a reduce/reduce conflict, the default is to reduce by the *earlier grammar rule*

Download from…

- Bison is the free variant for yacc, available also for Win OS :

http://gnuwin32.sourceforge.net/packages/bison.htm

How to work with a yacc file?

bison -dy prog.y
gcc y.tab.c -o prog.exe
./prog
OR (if you have an external lexical analyzer…)
flex prog.l
bison -dy prog.y
gcc lex.yy.c y.tab.c -o prog.exe
./prog
You need a C compiler! (Try **Cygwin)**

How to write the input in GDL code?

- The general syntax of a program written in GDP :

  declarations
  %%
  grammar rules
  %%
  program

- The order of the 3 sections has to be maintain
- The first or the 3rd section may be missing

Declaration Section

- code in C, which has to be put between %{ and %}: this code is copied in the output program
- tokens, with the reserved word %token
- the start symbol of the grammar is declared with the reserved word %start
- the associativity of the symbols is declared with %left, %right and %nonassoc
- The priority of the symbols is deducted from their order: the last defined symbol has the higher priority

Rules Section

- Contains the rules of a type 2 grammar
- The non-terminals are written without codification
- The terminals(tokens) are written between ' ' or " " or using tokens previously specified and offered by yylex() function
- The left side and the right side of a rule are separated through ‘:’; the option in the right side is made with ‘|’; the rules are finished with ‘;’
- We can attach semantic actions to rules, using { and }; a semantic action means writing to the screen or an arithmetic calculus
- $$ makes a reference to the value of the left side of the rule
- $k makes a reference to the value of the k-th symbol from the right side of the rule

Program Section

- It is copied in the output program, without any change
- There are at least 3 functions which have to be described by the programmer: main(), yylex() and yyerror()
- An interesting situation is when yylex() is given by an external file lex.l, written in SDL: this is an example of collaboration between LEX and YACC

An example of yacc file: compile and test it!

```
%{
#include <stdio.h>
%}

%start S

%%

S   : 'a' S S 'b' { fprintf(stdout,"S->aSSb\n"); }
    | 'c'         { fprintf(stdout,"S->c\n"); }
    ;

%%

int main()
{
    yyparse();
}

void yyerror( char *s )
{
    fprintf(stderr,"\nPARSE ERROR : %s\n",s);
}

int yylex( void )
{
    int c;
    while ((c=getchar()) == ' ' || c == '\t');
    if(c==EOF || c=='\n') return 0;
    return c ;
}
```

If you obtain an inexplicable error like this:

error: conflicting types for 'yyerror‘

And the function with problems is this one:

void yyerror (**const char** *s) /* Called by yyparse on error */ { printf ("%s\n", s); }

Here is the solution:

•you have to have a prototype for it in the first section of your yacc file

```
%{ #include <stdio.h>
void yyerror(char const *);
 %}
```

Take a lot at the following example and then try to create a simple expression evaluator. It would be advisable to work with an external lexical analyzer!

```
%{
#include <stdio.h>
%}

%start START

%left '+' '-'
%left '*' '/'

%%

START   : EXPR;

EXPR    : 'a'               { fprintf(stdout,"E->a\n");   }
        | EXPR '+' EXPR     { fprintf(stdout,"E->E+E\n"); }
        | EXPR '-' EXPR     { fprintf(stdout,"E->E-E\n"); }
        | EXPR '*' EXPR     { fprintf(stdout,"E->E*E\n"); }
        | EXPR '/' EXPR     { fprintf(stdout,"E->E/E\n"); }
        | '(' EXPR ')'      { fprintf(stdout,"E->(E)\n"); }
        ;

%%

int main()
{
     yyparse();
}

void yyerror( char *s )
{
     fprintf(stderr,"\nPARSE ERROR : %s\n",s);
}
```

```
int yylex( void )
{
     int c;
     while ((c=getchar()) == ' ' || c == '\t' ) ;
     if( c==EOF || c=='\n') return 0;
     return c;
}
```

Annexes

- installer for flex
- installer for bison
- installer for Cygwin
- yacc_article.pdf
- an example of a small interpreter for an user-defined language (made with lex and yacc): in helpt.txt, there is the description of the language