Introduction to Yacc

Review of Parser

- Parser invokes scanner for tokens.
- Parser analyze the syntactic structure according to grammars.
- Finally, parser executes the semantic routines.
- YACC generates the definition for yyparse() in **y.tab.c** and LEX generates the definition for yylex() in **lex.yy.c**.

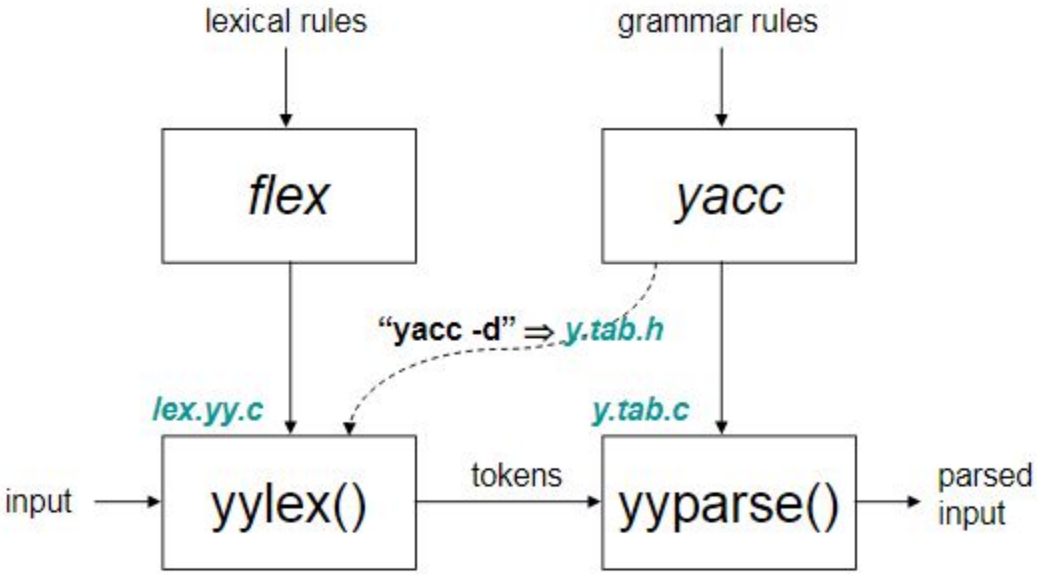
Introduction to yacc

- Yacc – yet another compiler compiler
- An LALR(1) parser generator.
- Yacc generates
  - Tables – according to the grammar rules.
  - Driver routines – in C programming language.
  - y.output – a report file.

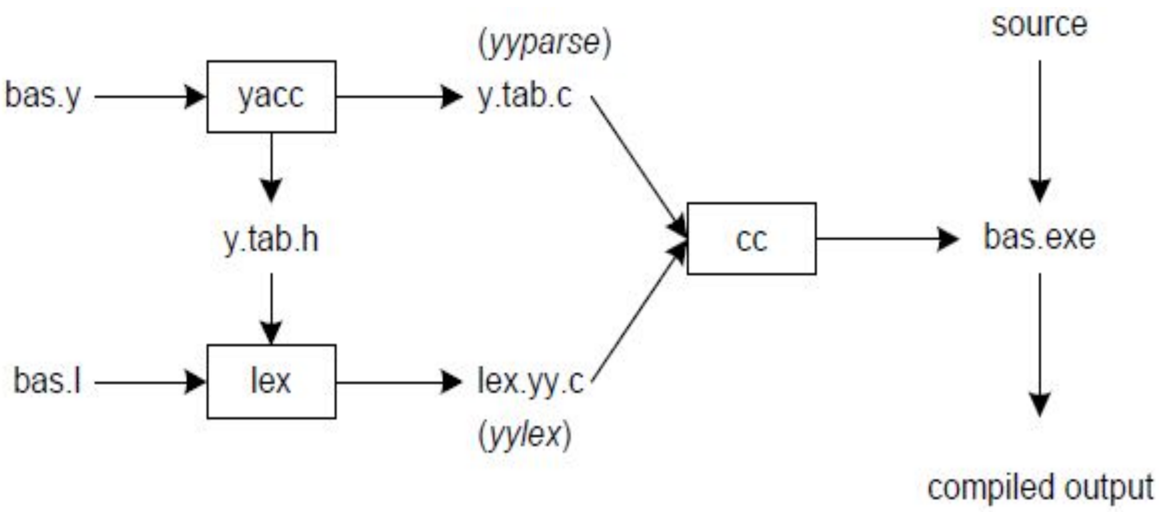
Cooperate with lex

- Invokes yylex() automatically.
- Generate *y.tab.h* file
- The lex input file must contains y.tab.h
- For each token that lex recognized, a number is returned (from *yylex()* function)

Using Yacc together with Lex



Linking lex&yacc



- Yacc environment
  - Yacc processes a yacc specification file and produces a y.tab.c file.
  - An integer function yyparse() is produced by Yacc.
    - Calls yylex() to get tokens.
    - Return non-zero when an error is found.
    - Return 0 if the program is accepted.
  - Need main() and and yyerror() functions.
  - Example:

```
yyerror(const char *str)
{    printf("yyerror: %s at line %d\n", str, yyline);
}
main()
{
  if (!yyparse()) {printf("accept\n");}
  else printf("reject\n");
}
```

- Writing a parser with YACC (Yet Another Compiler Compiler).
    - YACC file format:

      ```
      declarations      /* specify tokens, and non-terminals */
      %%
      translation rules  /* specify grammar here */
      %%
      supporting C-routines
      ```

    - Command “yacc yaccfile” produces y.tab.c, which contains a routine yyparse().
        - yyparse() calls yylex() to get tokens.
        - yyparse() calls a routine called **yylex()** everytime it wants to obtain a token from the input.
        - **yylex()** returns a value indicating the *type* of token that has been obtained. If the token has an actual *value*, this value (or some representation of the value, for example, a pointer to a string containing the value) is returned in an external variable named **yylval**.
    - yyparse() returns 0 if the program is grammatically correct, non-zero otherwise

The structure of YACC programs

DECLARATIONS

%%

RULES

%%

USER SUBROUTINES

Declarations

The declarations section consists of two parts:

(i) C declarations and

(ii) YACC declarations .

The C Declarations are delimited by **%{** and **%}**. This part consists of all the declarations required for the C code written in the *Actions* section and the *Auxiliary functions* section.

YACC copies the contents of this section into the generated y.tab.c file without any modification. %token : declares ALL terminals which are not literals.

%type : declares return value type for non-terminals.

Rules

A rule in a YACC program comprises of two parts

(i) the production part and

(ii) the action part.

production_head : production_body {action in C } ;

Rules Section

- Each rule contains LHS and RHS, separated by a colon and end by a semicolon.
- White spaces or tabs are allowed.
- Ex:

```
statement: name EUQALSIGN expression
     | expression ;

expression: number PLUSSIGN number
     | number MINUSSIGN number ;
```

Semantic Routines

- The action in semantic routines are executed for the production rule.
- The action is actually C source code.
- LHS: `$$`   RHS: `$1  $2  ……`
- Default action: { `$$ = $1;` }
- Action between a rule is allowed. For ex:

```
expression : simple_expression
  | simple_expression {somefunc($1);} relop simple_expression;
```

- Semantic actions
  - Semantic actions associate with productions can be specified.

    - $$ is the attribute associated with the left handside of the production
    - $$ represents the result of the non-terminal on the left-hand side of the rule.
    - $1, $2, $3, etc., represent the values of the right-hand side components in the production.
    - $1 is the attribute associated with the first symbol in the right handside, $2 for the second symbol, …
  - An action can be in anywhere in the production, it is also counted as a symbol.

user subroutines

The user subroutines/auxiliary functions section contains the definitions of two mandatory functions main(), yyerror().

Lex program

example.l

```
%{
#include <stdlib.h>
#include "y.tab.h"
%}
%%
[a-z]*  { return VARIABLE;   }
[0-9]+ {   yylval = atoi(yytext);      return INTEGER;   }
[-+()=/*\n] { return *yytext; }
[ \t]       ;
%%
int yywrap(void)
{   return 1;
}
```

Yacc program

example.y

```
%{
  #include<stdio.h>
      %}
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%%
program:
   program statement '\n’    |    ;
statement:    expr             { printf("%d\n", $1); }
        | VARIABLE '=' expr       { $1 = $3; }
             ;
 expr:    INTEGER    | VARIABLE  { $$ = $1; }
       | expr '+' expr          { $$ = $1 + $3; }
       | expr '-' expr          { $$ = $1 - $3; }        | expr '*' expr          { $$ = $1 * $3; }       | expr '/'
     expr          { $$ = $1 / $3; }        | '(' expr ')'          { $$ = $2; }    ;
%%
          void yyerror()
          {  printf("\nEntered arithmetic       expression is Invalid\n\n");
          }
```

- Define the operator precedence and associativity: + and - have the same precedence level and are left-associative.
- * and / have higher precedence than + and -, and they are also left-associative.
- **The program rule consists of one or more statements, separated by newlines ('\n').**
- The first part handles multiple statements (program statement '\n'), while the second part (;) allows for an empty program (i.e., no statements at all).

- A statement can either be an expression (expr) or an assignment of an expression to a variable.In the first case, the value of the expression ($1) is printed using printf("%d\n", $1).
- In the second case, a variable (VARIABLE) is assigned the value of an expression ($3). The variable's value is stored in $1 (since the variable itself is left-hand side).

- An expr can be:An INTEGER (where $$ = $1 means the value of the expression is the integer itself).
- A VARIABLE (where $$ = $1 means the value of the expression is the value of the variable).
- An expression with binary operators (+, -, *, /). For example, expr '+' expr means the sum of the first ($1) and third ($3) expressions.
- A parenthesized expression ('(' expr ')'), where $$ = $2 means the value of the expression is the value of the expression inside the parentheses.

```
(base) psg@psg-OptiPlex-3060:~$ lex example.l
(base) psg@psg-OptiPlex-3060:~$ yacc example.y
(base) psg@psg-OptiPlex-3060:~$ gcc lex.yy.c y.tab.c -ll
(base) psg@psg-OptiPlex-3060:~$ ./a.out
a=15
a+30
45
35/5
7
20*4
80
```