1 YACC - Yet Another Compiler Compiler

Yacc is a program which given a context-free grammar, constructs a C program which will parse input according to the grammar rules. Yacc was developed by S. C. Johnson an others at AT&T Bell Laboratories. Yacc provides for semantic stack manipulation and the specification of semantic routines. A input file for Yacc is of the form:

```
C and parser declarations/definitions
%%
Grammar rules and actions
%%
C subroutines
```

1.1 Definition/Declarations Section

The **definitions** section consists of token declarations and C code bracketed by "%" and "%". The first section of the Yacc file consists of a list of tokens (other than single characters) that are expected by the parser and the specification of the start symbol of the grammar. This section of the Yacc file may contain specification of the precedence and associativity of operators. This permits greater flexibility in the choice of a context-free grammar. Addition and subtraction are declared to be left associative and of lowest precedence while exponentiation is declared to be right associative and to have the highest precedence.

Definition Section

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char *);
%}
%token NUMBER
%token IDENTIFIER
%left '-' '+'
%left '*' '/'
%right '^'
% %
```

A declaration/definition section may thus contain

- Declarations of tokens. Yacc requires token names to be declared as such using the keyword %token.
- Declaration of the start symbol using the keyword %start
- C declarations: included files, global variables, types.
- C code between % and %.

1.2 Rules Section

The BNF grammar is placed in the **rules section**. The second section of the Yacc file consists of the context-free grammar for the language. Productions are separated by semicolons.The left-hand

side of a production, or nonterminal, is entered left-justified and followed by a colon. This is followed by the right-hand side of the production. Actions associated with a rule are entered in braces. The empty production is left empty, non-terminals are written in all lower case, and the multicharacter terminal symbols in all uppercase.

Internally yacc maintains two stacks in memory; a parse stack and a value stack. The parse stack contains terminals and nonterminals that represent the current parsing state. The value stack is an array of YYSTYPE elements and associates a value with each element in the parse stack. For example when lex returns an INTEGER token yacc shifts this token to the parse stack. At the same time the corresponding yylval is shifted to the value stack. The parse and value stacks are always synchronized so finding a value related to a token on the stack is easily accomplished.

A rule has the form A rule has the form:

```
nonterminal : sentential form
            | sentential form
            ..................
            | sentential form
            ;
```

Consider the grammar

```
E->E+E
```

We can write a rule for this as

```
expr: expr '+' expr { $$ = $1 + $3; }
```

Here we replace the right-hand side of the production in the parse stack with the left-hand side of the same production. In this case we pop *"expr '+' expr"* and push *"expr"*. We have reduced the stack by popping three terms off the stack and pushing back one term. We may reference positions in the value stack in our C code by specifying "$1" for the first term on the right-hand side of the production, "$2" for the second, and so on. "$$" designates the top of the stack after reduction has taken place. The above action adds the value associated with two expressions, pops three terms off the value stack, and pushes back a single sum. As a consequence the parse and value stacks remain synchronized.

Consider the rule

```
E->int
```

Numeric values are initially entered on the stack when we reduce from INTEGER to expr. After INTEGER is shifted to the stack we apply the rule

```
expr: INTEGER { $$ = $1; }
```

The INTEGER token is popped off the parse stack followed by a push of expr. For the value stack we pop the integer value off the stack and then push it back on again.

1.3 Subroutines Section

User subroutines are added in the subroutines section. The third section of the Yacc file consists of C code. There must be a main() routine which calls the function yyparse(). The function yyparse() is the driver routine for the parser. There must also be the function yyerror() which is used to report on errors during the parse.

Sample Subroutine Section

```
C and parser declarations
%%
Grammar rules and actions
%%
main( int argc, char *argv[] )
{
yyin = fopen( argv[0], "r" );
yyparse ();
}
yyerror (char *s) /* Called by yyparse on error */
{
printf ("%s\n", s);
}
```

The parser, as written, has no output however, the parse tree is implicitly constructed during the parse. As the parser executes, it builds an internal representation of the the structure of the program. The internal representation is based on the right hand side of the production rules. When a right hand side is recognized, it is reduced to the corresponding left hand side. Parsing is complete when the entire program has been reduced to the start symbol of the grammar.

Compiling the Yacc file with the command *bison -d file.y* causes the generation of two files *file.tab.h* and *file.tab.c*. The *file.tab.h* contains the list of tokens which is included in the file which defines the scanner. The file *file.tab.c* defines the C function *yyparse()* which is the parser.

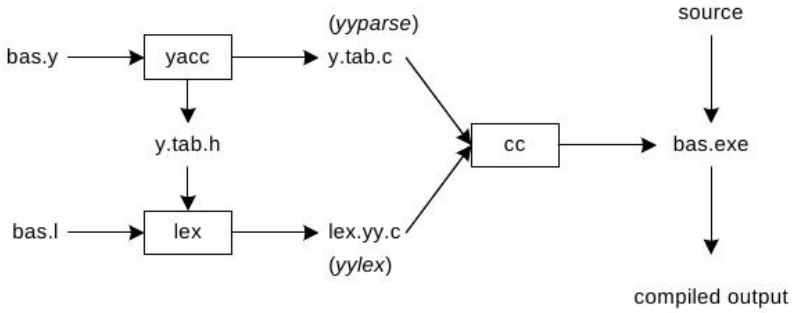

Figure 2: Building a Compiler with Lex/Yacc

1.4 LEX-YACC INTERACTION

yyparse() calls yylex() when it needs a new token.

LEX	YACC
return(TOKEN)	%token TOKEN TOKEN is used in production

The tokens declared using *%token* in YACC file should be defined using lex program. The lex program should return these tokens. In the YACC example given above a token NUMBER is declared. So in the corresponding LEX program we should return NUMBER token as shown

Pattern Specification part in Lex

```
[0-9]+ {yylval=atoi(yytext);
        return NUMBER;}
```

The external variable yylval yyval should be declared as extern type. It is used in a LEX source program to return values of lexemes, yylval is assumed to be integer if you take no other action. Changes related to yylval must be made in the definitions section of YACC specification by adding new types. Lex program should also contain

```
#include "yacc.tab.h"
```

2 Simple Program

Lex program: ch301.l

```
%{
#include "ch301.tab.h"
extern int yylval;
%}

%%
[0-9]+   { yylval = atoi (yytext);
           printf ("scanned the number %d\n", yylval);
           return NUMBER; }
[ \t]    { printf ("skipped whitespace\n"); }
\n       { printf ("reached end of line\n");
           return 0;
           }
.        { printf ("found other data \"%s\"\n", yytext);
           return yytext[0];
           /* so yacc can see things like '+', '-', and '=' */
           }
%%
```

Yacc program: ch301.y

```
%{
#include <stdio.h>
#include"lex.yy.c"
int yyerror(char *);
int yywrap();
%}

%token NUMBER
%%
```

```
expression:expression '+' NUMBER { $$ = $1 + $3;
                                   printf ("Recognized '+' expression.\n");
                                   }
          |expression '-' NUMBER { $$ = $1 - $3;
                                   printf ("Recognized '-' expression.\n");
                                   }
          |NUMBER        { $$ = $1;
                           printf ("Recognized a number.\n");
                           }
        ;
%%
int main (void) {
        return yyparse();
        }

int yyerror (char *msg) {
        return fprintf (stderr, "YACC: %s\n", msg);
        }
int yywrap()

{
return -1;
}
```

3 Executing YACC

```
$ flex ch301.l
$ bison ch301.y -d
$ gcc -o out ch301.tab.c
$ ./out
2+3
scanned the number 2
Recognized a number.
found other data "+"
scanned the number 3
Recognized '+' expression.
reached end of line
```

Unless otherwise instructed Yacc will resolve the parsing action conflicts using the following rules.
1. A reduce-reduce conflict by 1st production
2. A shift-reduce in favor of shift