2017-2018

ANKARA UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

**COM241**

**LEXICAL ANALYSIS AND PARSING**

16290102 Çağla Deniz KILIÇOĞLU

16290085 Yeter Tuğba ÇETİN

**Table of Contents**

1. **Objective 3**

**1.1 Compilation 3**

1. **Rules of the Language 3**
2. **BNF Notations 5**
3. **Lex File 6**
4. **Yacc File 8**
5. **Lex Code Descriptions 10**
6. **Yacc Code Descriptions 13**
7. **OBJECTIVE**

To learn the lexical and syntax analysis of context-free grammars using lex&yacc and to design a syntax analyzer for your own programming language.

**1.1 COMPILATION**

-lex mpl.l

-yacc -d mpl.y

-gcc -o mpl y.tab.c -lfl

-./mpl < filename.mpl

1. **RULES OF THE LANGUAGE**

We designed a lexical and syntax analyzer for our own programming language and called it as MPL(My Programming Language).

The rules for this language are shown below:

* All commands must be ended by “~”
* A calculation statement includes five types of operators : +, -, /, \*, %
* && (AND) and | | (OR) operators can be used for logical expressions.
* Four types of comparison operators exist: >, <, >=, <=
* To see if the condition depends on something you should use “jedi” and “sith” operators. You can only calculate one expression at a time. There should be no extra newline between them.

**jedi ( ... )**

**{ ... }**

**sith ( ... )**

**{ ... }**

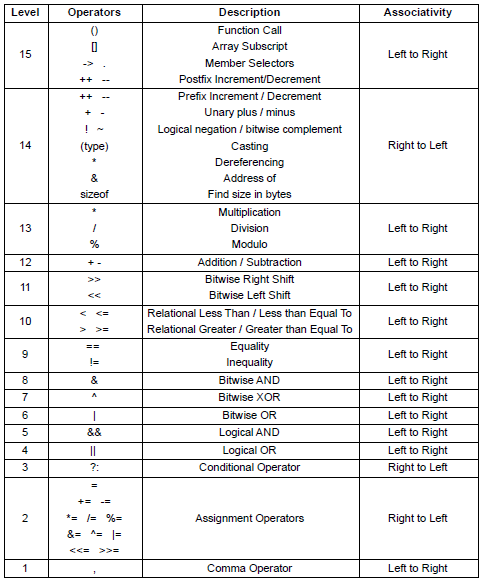
* To see the condition work for more than one time you should use “yell” operator. The loop will execute nine times. You can only calculate one expression at a time. There should be no extra newline between them.

**yell ( ... )**

**{ ... }**

* Program must be ended with “quit” or “quit ~” command to exit.
* If the program executes correctly an “OK” message will be given. Otherwise you will see an error like “Syntax Error!”
* The presedence of the operators are given as in the Presedence Table.

Table 1.0 Operator Precedence



1. **BNF NOTATIONS**

<program> -> <program> <statement> ~

| <program>

<statement> -> <expr>

| <variable> = <expr>

| JEDI ( <expr>) { <statement> } SITH { <statement> }

| YELL ( <expr> ) { <statement> }

| QUIT

<expr> -> <expr> + <expr>

| <expr> - <expr>

| <expr> \* <expr>

| <expr> / <expr>

| <expr> % <expr>

| <expr> < <expr>

| <expr> > <expr>

| <expr> <= <expr>

| <expr> >= <expr>

| <expr> && <expr>

| <expr> || <expr>

| <expr> == <expr>

| <expr> != <expr>

| ( <expr> )

| int

| variable

1. **LEX FILE (mpl.l)**

%{

#include <stdio.h>

#include "y.tab.h"

#define YY\_DECL int yylex()

void yyerror(char \*);

%}

%%

[a-z] {

yylval = \*yytext - 'a';

return VARIABLE;

}

[0-9]+ {

yylval = atoi(yytext);

return INTEGER;

}

"\n" {return NEWLINE;}

"~" {return TILDE;}

"sith" {return SITH;}

"jedi" {return JEDI;}

"yell" {return YELL;}

"quit" {return QUIT;}

"+" {return ADD;}

"-" {return SUB;}

"\*" {return MULT;}

"/" {return DIV;}

"%" {return REM;}

"=" {return EQ;}

"{" {return L\_BRA;}

"}" {return R\_BRA;}

"(" {return L\_PAR;}

")" {return R\_PAR;}

"<" {return SMALLER;}

">" {return GREATER;}

"<=" {return LESS\_EQ;}

">=" {return GREATER\_EQ;}

"&&" {return AND\_OP;}

"||" {return OR\_OP;}

"==" {return EQ\_OP;}

"!=" {return NOT\_OP;}

[ \t] ;

%%

int yywrap(void)

{

return 1;

}

1. **YACC FILE (mpl.y)**

%{

void yyerror(char \*);

int yylex(void);

int sym[26];

#include <stdio.h>

#include "lex.yy.c"

%}

%%

program:

program statement TILDE NEWLINE

| program NEWLINE

|

;

statement:

expr { printf("%d\n", $1); }

| VARIABLE EQ expr { sym[$1] = $3; }

| QUIT { printf("\nOK\n"); exit(1); }

| JEDI L\_PAR expr R\_PAR NEWLINE

L\_BRA expr R\_BRA NEWLINE

SITH NEWLINE

L\_BRA expr R\_BRA

{

if($3==0) { printf("\nWelcome to the dark side. We have cookies.\n"); printf("%d\n", $11); }

else if ($3 ==1) { printf("\nMay the force be with you.\n"); printf("%d\n", $6); }

}

| YELL L\_PAR expr R\_PAR NEWLINE

L\_BRA expr R\_BRA

{

if($3 == 1) { int i; printf("\nYell nine times for Mortal Man doomed to die!\n"); for( i= 0; i<9; i++) {printf("%d ", $7); } printf("\n"); }

else if ($3 ==0) printf("\nYou Shall Not Pass!\n");

}

;

expr:

INTEGER

| VARIABLE { $$ = sym[$1]; }

| expr ADD expr { $$ = $1 + $3; }

| expr SUB expr { $$ = $1 - $3; }

| expr MULT expr { $$ = $1 \* $3; }

| expr DIV expr { if ( $3 == 0 ) { printf("%d Cannot divide by zero!\n", $1); exit(1); } else $$ = $1 / $3; }

| expr REM expr { $$ = $1 % $3; }

| expr SMALLER expr { $$ = $1 < $3; }

| expr GREATER expr { $$ = $1 > $3; }

| expr LESS\_EQ expr { $$ = $1 <= $3; }

| expr GREATER\_EQ expr { $$ = $1 >= $3; }

| expr AND\_OP expr { $$ = $1 && $3; }

| expr OR\_OP expr { $$ = $1 || $3; }

| expr EQ\_OP expr { $$ = $1 == $3; }

| expr NOT\_OP expr { $$ = $1 != $3; }

| L\_PAR expr R\_PAR { $$ = $2; }

;

%%

void yyerror(char \*s)

{

fprintf(stderr, "Syntax Error!\n");

}

int main(int argc, char \*argv[])

{

yyin = fopen(argv[1], "r");

yyparse();

fclose(yyin);

return 0;

}

1. **LEX CODE DESCRIPTIONS**

%{

#include <stdio.h>

#include "y.tab.h"

#define YY\_DECL int yylex()

void yyerror(char \*);

%}

1. #include <stdio.h> is to use C language.
2. #include “y.tab.h” has definitions for tokens.
3. #define YY\_DECL int yylex() is implies the main input point for Lex, reads the input stream and generates the tokens, returning zero at the end of the input stream.void
4. yyerror(char \*); is to create a message when there is an error in the program.

%%

[a-z] {

yylval = \*yytext - 'a';

return VARIABLE;

}

[0-9]+ {

yylval = atoi(yytext);

return INTEGER;

}

"\n" {return NEWLINE;}

"~" {return TILDE;}

"sith" {return SITH;}

"jedi" {return JEDI;}

"yell" {return YELL;}

"quit" {return QUIT;}

1. [a-z] {yylval = \*yytext – 'a'; return VARIABLE; } is for declaring VARIABLE as the letters of the alphabet in non-capital letters.
2. [0-9]+ { yylval = atoi(yytext); return INTEGER; } is for declaring INTEGER matches a string of one or more digits.
3. "\n" {return NEWLINE;} is newline.
4. "~" {return TILDE;} is for declaring “~” symbol as TILDE in yacc.
5. "jedi" {return JEDI;} is for declaring “jedi” as JEDI token in yacc.
6. "sith" {return SITH;} is for declaring “sith” as SITH token in yacc.
7. "yell" {return YELL;} is for declaring “yell” as YELL token in yacc.
8. "quit" {return QUIT;} is for declaring “quit” as QUIT token in yacc.

"+" {return ADD;}

"-" {return SUB;}

"\*" {return MULT;}

"/" {return DIV;}

"%" {return REM;}

1. "+" {return ADD;} is for declaring “+” symbol as ADD token.
2. "-" {return SUB;} is for declaring “-” symbol as SUB token.
3. "\*" {return MULT;} is for declaring “\*” symbol as MULT token.
4. "/" {return DIV;} is for declaring “/” symbol as DIV token.
5. "%" {return REM;} is for declaring “%” symbol as REM token.

"=" {return EQ;}

"{" {return L\_BRA;}

"}" {return R\_BRA;}

"(" {return L\_PAR;}

")" {return R\_PAR;}

"<" {return SMALLER;}

">" {return GREATER;}

"<=" {return LESS\_EQ;}

">=" {return GREATER\_EQ;}

1. "=" {return EQ;} is for declaring “=” symbol as EQ token.
2. "{" {return L\_BRA;} is for declaring “{” symbol as L\_BRA token.
3. "}" {return R\_BRA;} is for declaring “}” symbol as R\_BRA token.
4. "(" {return L\_PAR;} is for declaring “(” symbol as L\_PAR token.
5. ")" {return R\_PAR;} is for declaring “)” symbol as R\_PAR token.
6. "<" {return SMALLER;} is for declaring “<” symbol as SMALLER token.
7. ">" {return GREATER;} is for declaring “>” symbol as GREATER token.
8. "<=" {return LESS\_EQ;} is for declaring “<=” symbol as LESS\_EQ token.
9. ">=" {return GREATER\_EQ;} is for declaring “>=” symbol as GREATER\_EQ token.

"&&" {return AND\_OP;}

"||" {return OR\_OP;}

"==" {return EQ\_OP;}

"!=" {return NOT\_OP;}

[ \t] ;

%%

1. "&&" {return AND\_OP;} is for declaring “&&”symbol as AND\_OP token.
2. "||" {return OR\_OP;} is for declaring “||”symbol as OR\_OP token.
3. "==" {return EQ\_OP;} is for declaring “==”symbol as EQ\_OP token.
4. "!=" {return NOT\_OP;} is for declaring “!=”symbol as NOT\_OP token.
5. [ \t] ; is for declaring “;” null operator.

int yywrap(void)

{

return 1;

}

1. int yywrap(void) { return 1; } it is called by lex when input is exhausted (or at EOF). default yywrap always return 1.
2. **YACC CODE DESCRIPTIONS**

%token INTEGER VARIABLE NEWLINE

%token JEDI SITH YELL QUIT

%token ADD SUB MULT DIV REM

%token EQ

%token L\_BRA R\_BRA L\_PAR R\_PAR

%token SMALLER GREATER LESS\_EQ GREATER\_EQ

%token AND\_OP OR\_OP EQ\_OP NOT\_OP

%token TILDE

* %token is for passing the names that we described in lex.

%left L\_PAR R\_PAR

%left L\_BRA R\_BRA

%left MULT DIV REM

%left ADD SUB

%left SMALLER GREATER LESS\_EQ GREATER\_EQ

%left EQ\_OP NOT\_OP

%left AND\_OP OR\_OP

%right EQ

1. %left is for left-to-right associations.
2. %right is for right-to-left associations.

* We have to write associativity rules after we have described the tokens.

%{

void yyerror(char \*);

int yylex(void);

int sym[26];

#include <stdio.h>

#include "lex.yy.c"

%}

* Declerations for error message, input stream, variable length, C library and lex.yy.c indicates that the lex output is intended for a yacc parser.

%%

program:

program statement TILDE NEWLINE

| program NEWLINE

|

;

1. program: Our program starts here.
2. program statement TILDE NEWLINE

<program> -> <program> <statement> TILDE NEWLINE

1. | program NEWLINE

<program> -> <program> NEWLINE

1. | Is for empty command.
2. ;

* When program has been called it returns program statement TILDE NEWLINE

or program NEWLINE or empty command.

statement:

expr { printf("%d\n", $1); }

| VARIABLE EQ expr { sym[$1] = $3; }

| QUIT { printf("\nOK\n"); exit(1); }

| JEDI L\_PAR expr R\_PAR NEWLINE

L\_BRA expr R\_BRA NEWLINE

SITH NEWLINE

L\_BRA expr R\_BRA

{

if($3==0) { printf("\nWelcome to the dark side. We have cookies.\n"); printf("%d\n", $11); }

else if ($3 ==1) { printf("\nMay the force be with you.\n"); printf("%d\n", $6); }

}

| YELL L\_PAR expr R\_PAR NEWLINE

L\_BRA expr R\_BRA

{

if($3 == 1) { int i; printf("\nYell nine times for Mortal Man doomed to die!\n"); for( i= 0; i<9; i++) {printf("%d ", $7); } printf("\n"); }

else if ($3 ==0) printf("\nYou Shall Not Pass!\n");

}

;

1. statement:
2. expr { printf("%d\n", $1); }

<statement> -> <expr>

1. | VARIABLE EQ expr { sym[$1] = $3; }

<statement> -> VARIABLE = <expr>

1. | QUIT { printf("\nOK\n"); exit(1); }

| JEDI L\_PAR expr R\_PAR NEWLINE

L\_BRA expr R\_BRA NEWLINE

SITH NEWLINE

L\_BRA expr R\_BRA

{ if($3==0) { printf("\nWelcome to the dark side. We have cookies.\n"); printf("%d\n", $11); } else if ($3 ==1) { printf("\nMay the force be with you.\n"); printf("%d\n", $6); } }

<statement> -> JEDI ( <expr> ) NEWLINE { <expr> } SITH NEWLINE { <expr> }

1. | YELL L\_PAR expr R\_PAR NEWLINE

L\_BRA expr R\_BRA { if($3 == 1) { int i; printf("\nYell nine times for Mortal Man doomed to die!\n"); for( i= 0; i<9; i++) {printf("%d ", $7); } printf("\n"); } else if ($3 ==0) printf("\nYou Shall Not Pass!\n"); }

<statement> -> YELL ( <expr> ) NEWLINE { <expr> }

;

* When statement has been called it returns expr

or VARIABLE = expr

or “OK” message

or JEDI ( expr ) NEWLINE { expr } SITH NEWLINE { expr } *Since we have the 3rd token as condition we check whether it is true. If it is true it enters JEDI’s body and print “My the force be with you.” and the result , if it is false it enters SITH’s body and print “Welcome to the dark side. We have cookies.” and the result.*

or YELL ( expr ) NEWLINE { expr } *Since we have 3rd token as condition we check whether it is true. If it is true it enters the loop and prints “Yell nine times for Mortal Man doomed to die!” and also prints the result. If it is false it doesn’t enter the loop and prints “You Shall Not Pass!”.*

expr:

INTEGER

| VARIABLE { $$ = sym[$1]; }

| expr ADD expr { $$ = $1 + $3; }

| expr SUB expr { $$ = $1 - $3; }

| expr MULT expr { $$ = $1 \* $3; }

| expr DIV expr { if ( $3 == 0 ) { printf("%d Cannot divide by zero!\n", $1); exit(1); } else $$ = $1 / $3; }

| expr REM expr { $$ = $1 % $3; }

| expr SMALLER expr { $$ = $1 < $3; }

| expr GREATER expr { $$ = $1 > $3; }

| expr LESS\_EQ expr { $$ = $1 <= $3; }

| expr GREATER\_EQ expr { $$ = $1 >= $3; }

| expr AND\_OP expr { $$ = $1 && $3; }

| expr OR\_OP expr { $$ = $1 || $3; }

| expr EQ\_OP expr { $$ = $1 == $3; }

| expr NOT\_OP expr { $$ = $1 != $3; }

| L\_PAR expr R\_PAR { $$ = $2; }

;

%%

* When expr has been called it returns INTEGER *[0-9]+*

or VARIABLE *[a-z]*

or expr + expr *expr + expr is printed.*

or expr – expr *expr - expr is printed.*

or expr \* expr *expr \* expr is printed.*

or expr / expr *expr / expr is printed. If the divident is zero it prints “Cannot divide by zero!”.*

or expr % expr *expr % expr is printed.*

or expr < expr *1 or 0 is printed.*

or expr > expr *1 or 0 is printed.*

or expr <= expr *1 or 0 is printed.*

or expr >= expr *1 or 0 is printed.*

or expr && expr *1 or 0 is printed.*

or expr || expr *1 or 0 is printed.*

or expr == expr *1 or 0 is printed.*

or expr != expr *1 or 0 is printed.*

or ( expr )

void yyerror(char \*s)

{

fprintf(stderr, "Syntax Error!\n");

}

int main(int argc, char \*argv[])

{

yyin = fopen(argv[1], "r");

yyparse();

fclose(yyin);

return 0;

}

* yyerror() function tells us the message if there is an error.
* In main function, yyin is the input stream pointer so we assigned a readable file to it so MPL can take inputs from file and command line.