Lab 8-9

Jeler Andrei-Iulian

934

Github: https://github.com/SummerRolls99/FLCD/tree/main/lab%20-%20lex%26yacc

Lab 8

Statement: Use lex

You may use any version (LEX or FLEX)

1) Write a LEX specification containing the regular expressions corresponding to your language specification - see lab 1

2) Use Lex in order to obtain a scanner. Test for the same input as

in lab 1 (p1, p2).

Deliverables: pdf file containing lang.lxi (lex specification file) + demo

Lab 9

Statement: Use yacc

You may use any version (yacc or bison)

1. Write a specification file containing the production rules corresponding to the language specification (use syntax rules from lab1).
2. Then, use the parser generator (no errors)

Deliverables: lang.y (yacc specification file)

BONUS: modify lex to return tokens and use yacc to return string of productions

LEX

lang.lxi

```
%{

#include <stdio.h>

#include <string.h>
#include "y.tab.h"


int currentLine = 1;
%}


%option noyywrap
%option caseless


DIGIT           [0-9]
NZ_DIGIT    [1-9]
ZERO      [0]
NUMBER     {NZ_DIGIT}{DIGIT}*
SIGN      [+]|[-]
INTEGER                    {ZERO}|{NUMBER}|{SIGN}{NUMBER}
SIGNER_INTEGER {SIGN}{NUMBER}
SPECIAL_CHAR    "_"|"."|","|";"|":"|"?"|"!"|"@"|"/"|"("|")"|"-
"|"+"|"="|"{"|"}"|"*"|"["|"]"|"$"|"%"|"^"|" "
CHAR      {DIGIT}|{SPECIAL_CHAR}|[a-zA-Z]
CHARACTER      "'"{CHAR}"'"
STRING          [\"]{CHAR}*[\"]
CONSTANT                   {STRING}|{INTEGER}|{CHARACTER}
IDENTIFIER                 [a-zA-Z_][a-zA-Z0-9_]*


%%


and {return AND;}
or {return OR;}
not {return NOT;}
if {return IF;}
```

```
else {return ELSE;}
elif {return ELIF;}
while {return WHILE;}
for {return FOR;}
read {return READ;}
write {return WRITE;}
integer {return INTEGER;}
string {return STRING;}
char {return CHAR;}
program {return PROGRAM;}
bool {return BOOL;}
return {return RETURN;}

{CONSTANT} {return CONSTANT;}
{IDENTIFIER} {return IDENTIFIER;}

; {return SEMI_COLON;}
"," {return COMMA;}
\t {return DOT;}
\{ {return OPEN_CURLY_BRACKET;}
\} {return CLOSED_CURLY_BRACKET;}
\[ {return OPEN_SQUARE_BRACKET;}
\] {return CLOSED_SQUARE_BRACKET;}
\( {return OPEN_ROUND_BRACKET;}
\) {return CLOSED_ROUND_BRACKET;}

\+ {return PLUS;}
\- {return MINUS;}
\* {return MUL;}
```

```
\/ {return DIV;}
\% { return PERCENT;}
\< { return LT;}
\> { return GT;}
\<= { return LE;}
\>= { return GE;}
"=" { return ATRIB;}
\== { return EQ;}
\!= { return NOT_EQ;}

[\n\r] {currentLine++;}
[ \t\n]+ {}

[a-zA-Z_0-9][a-zA-Z0-9_]* {printf("%s - illegal identifier found at line %d\n", yytext, currentLine);}
\'[a-zA-Z0-9]*\' {printf("%s - illegal char at line %d, did you mean string?\n", yytext, currentLine);}
[\"]{CHAR}* {printf("%s - illegal string constant at line, you forgot to close it %d\n", yytext, currentLine);}

. {printf("%s - illegal token found at line %d\n",yytext, currentLine);}

%%
```

p1.in (file for testing)

```
program
{
integer a, b, c;
string printMessage = "is the biggest number";
read(a);
read(b);
read(c);

a = -2;

if (a > b and a > c)

{

write("a", printMessage);

}

elif (b > a and b > c)

{

write("b", printMessage);

}

else

{

write("c", printMessage);

}

return 0;

}
```

How to run:

```
lex lang.lxi

gcc lex.yy.c -o lex.exe -ll

./lex.exe p1.in
```

Or

```
./lex.exe < p1.in
```

YACC

lang.y

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


#define YYDEBUG 1
%}


%token AND
%token OR
%token NOT
%token IF
%token ELSE
%token ELIF
%token WHILE
%token FOR
%token READ
%token WRITE
%token INTEGER
%token STRING
%token CHAR
%token BOOL
%token RETURN
%token PROGRAM
```

```
%token IDENTIFIER
%token CONSTANT
%token SEMI_COLON
%token COMMA
%token DOT
%token OPEN_CURLY_BRACKET
%token CLOSED_CURLY_BRACKET
%token OPEN_SQUARE_BRACKET
%token CLOSED_SQUARE_BRACKET
%token OPEN_ROUND_BRACKET
%token CLOSED_ROUND_BRACKET
%token PLUS
%token MINUS
%token MUL
%token DIV
%token PERCENT
%token LT
%token GT
%token LE
%token GE
%token ATRIB
%token EQ
%token NOT_EQ


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

```
%start program_stmt

%%

program_stmt : PROGRAM compound_stmt {printf("program end\n");}
        ;

compound_stmt : OPEN_CURLY_BRACKET stmt_list CLOSED_CURLY_BRACKET {printf("compound stmt\n");}
         ;

stmt_list : stmt stmt_temp
      ;

stmt_temp : /* empty */
      | stmt_list
      ;

stmt : simple_stmt
   | complex_stmt
   ;
```

```
simple_stmt : decl_stmt {printf("declaration stmt\n");}
        | assign_stmt SEMI_COLON {printf("assign stmt\n");}
        | return_stmt SEMI_COLON {printf("return stmt\n");}
        | IO_stmt SEMI_COLON {printf("IO stmt\n");}
        ;


complex_stmt : if_stmt {printf("if stmt\n");}
          | loop_stmt
          ;


IO_stmt : READ OPEN_ROUND_BRACKET IDENTIFIER CLOSED_ROUND_BRACKET {printf("read IO\n");}
       | WRITE OPEN_ROUND_BRACKET expression write_expressions {printf("write IO\n");}
          ;


write_expressions : COMMA expression write_expressions
               | CLOSED_ROUND_BRACKET
               ;


decl_stmt : type IDENTIFIER NZidentifier
        | type IDENTIFIER ATRIB expression NZEidentifier
        | type IDENTIFIER ATRIB OPEN_CURLY_BRACKET CONSTANT array_values
        ;
```

```
array_values : COMMA CONSTANT array_values
             | CLOSED_CURLY_BRACKET SEMI_COLON
             ;


NZidentifier : COMMA IDENTIFIER NZidentifier
             | SEMI_COLON
             ;


NZEidentifier : COMMA IDENTIFIER ATRIB expression NZEidentifier
              | SEMI_COLON
              ;


type : primary_types
     | array_types
     ;


primary_types : INTEGER
              | CHAR
              | STRING
              | BOOL
              ;


array_types : primary_types OPEN_SQUARE_BRACKET CONSTANT CLOSED_SQUARE_BRACKET
            ;
```

```
assign_stmt : IDENTIFIER ATRIB expression
        ;


expression : term operator expression
        | term
        ;


operator : PLUS
      | MINUS
      ;


term : factor MUL term
    | factor DIV term
    | factor
    ;


factor : OPEN_ROUND_BRACKET expression CLOSED_ROUND_BRACKET
     | IDENTIFIER
     | IDENTIFIER OPEN_SQUARE_BRACKET expression CLOSED_SQUARE_BRACKET
     | CONSTANT
     ;
```

```
return_stmt : RETURN expression

          ;



if_stmt : IF OPEN_ROUND_BRACKET condition CLOSED_ROUND_BRACKET compound_stmt {printf("simple if\n");}

       | IF OPEN_ROUND_BRACKET condition CLOSED_ROUND_BRACKET compound_stmt elif_stmt {printf("if with elif/else\n");}

       ;



elif_stmt : ELIF OPEN_ROUND_BRACKET condition CLOSED_ROUND_BRACKET compound_stmt elif_stmt

         | ELIF OPEN_ROUND_BRACKET condition CLOSED_ROUND_BRACKET compound_stmt

         | ELSE compound_stmt

         ;



loop_stmt : for_stmt {printf("for stmt\n");}

         | while_stmt {printf("while stmt\n");}

         ;



for_stmt : FOR OPEN_ROUND_BRACKET for_first condition SEMI_COLON assign_stmt CLOSED_ROUND_BRACKET compound_stmt {printf("larger stmt\n");}

        | FOR OPEN_ROUND_BRACKET for_first condition CLOSED_ROUND_BRACKET compound_stmt {printf("shorter for\n");}

        ;



for_first : decl_stmt
```

```
        | assign_stmt SEMI_COLON
        ;


while_stmt : WHILE OPEN_ROUND_BRACKET condition CLOSED_ROUND_BRACKET compound_stmt
         ;


condition : expression relational_operator expression conditional_operator condition
        | NOT expression relational_operator expression conditional_operator condition
        | expression relational_operator expression
        | NOT expression relational_operator expression


relational_operator : GT
                | LT
                | GE
                | LE
                | EQ
                | NOT_EQ
                ;


conditional_operator : AND
                 | OR
                 ;


%%
```

```
yyerror(char *s)
{
  printf("%s\n", s);
}


extern FILE *yyin;


main(int argc, char **argv)
{
  if (argc > 1)
    yyin = fopen(argv[1], "r");
  if ( (argc > 2) && ( !strcmp(argv[2], "-d") ) )
    yydebug = 1;
  if ( !yyparse() )
    fprintf(stderr,"No errors detected\n");
}



%%
```

<u>LEX & YACC – how to run</u>

```
lex lang.lxi
```

```
yacc lang.y

gcc lex.yy.c y.tab.c -o l.exe -lfl

./l.exe < p1.in
```