Formal Languages and Compiler Design - Lab9

Requirement

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

Solution

lang.lxi

```
%{
#include "y.tab.h"
#include <math.h>
%}

NONZERO_DIGIT     [1-9]
DIGIT         [0-9]
INTEGER_CT    0|(-?{NONZERO_DIGIT}{DIGIT}*)
CHAR_CT        \'[A-Z0-9]\'
STRING_CT    \"[A-Z0-9]*\"
BOOLEAN_CT    true|false
ID         [A-Z_][A-Z0-9_]*
ERROR        [+-]0|0{DIGIT}+|{DIGIT}+[A-Z0-9_]+
%%

"START"        { return START;}
"ENDPRG"    { return ENDPRG; }
"INT"        { return INT; }
"BOOLEAN"    { return BOOLEAN; }
"CHAR"        { return CHAR; }
"STRING"    { return STRING;}
"ARRAY"        { return ARRAY; }
"BEGIN"        { return BEGIN_STMT; }
"END"        { return END; }
"READ"        { return READ; }
"WRITE"        { return WRITE; }
"IF"        { return IF; }
"THEN"        { return THEN; }
"ELSE"        { return ELSE; }
"WHILE"        { return WHILE; }
"DO"        { return DO; }
```

```
"+"        { return ADD; }
" - "        { return SUBTRACT; }
"*"        { return MULTIPLY; }
"/"        { return DIV; }
"%"        { return MOD; }
"<"        { return SMALLER; }
"<="        { return SMALLER_OR_EQUAL; }
">"        { return GREATER; }
">="        { return GREATER_OR_EQUAL; }
"="        { return EQUAL; }
"!="        { return DIFFERENT; }
":="        { return ASSIGNED; }
"AND"        { return AND; }
"OR"        { return OR; }

"("        { return PARA_OPEN; }
")"        { return PARA_CLOSED; }
"["        { return SQUARE_BRACKET_OPEN; }
"]"        { return SQUARE_BRACKET_CLOSED; }
"{"        { return CURLY_BRACKET_OPEN; }
"}"        { return CURLY_BRACKET_CLOSED; }
";"        { return SEMI_COLON; }
":"        { return COLON; }

{ERROR}        printf("Error: %s\n", yytext);

{INTEGER_CT}    { printf("Integer constant: %s\n", yytext); return ct;}

{CHAR_CT}    { printf("Char constant: %s\n", yytext); return ct; }

{STRING_CT}    { printf("String: %s\n", yytext); return ct; }

{BOOLEAN_CT}    { printf("Boolean constant: %s\n", yytext); return ct; }

{ID}            { return id; }

"{"[^}\n]*"}"          /* eat up one-line comments */

[ \t\n]+         /* eat up whitespace */


. printf("Eroare\n");
%%
```

lang.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define YYDEBUG 1

int yylex();
void yyerror(char *s);
```

```
%}

%token START
%token ENDPRG
%token BEGIN_STMT
%token END
%token READ
%token WRITE
%token IF
%token THEN
%token ELSE
%token WHILE
%token DO

%token id
%token ct

%token INT
%token BOOLEAN
%token CHAR
%token STRING
%token ARRAY

%token ADD
%token SUBTRACT
%token MULTIPLY
%token DIV
%token MOD
%token SMALLER
%token SMALLER_OR_EQUAL
%token GREATER
%token GREATER_OR_EQUAL
%token EQUAL
%token DIFFERENT
%token ASSIGNED
%token AND
%token OR

%token PARA_OPEN
%token PARA_CLOSED
%token SQUARE_BRACKET_OPEN
%token SQUARE_BRACKET_CLOSED
%token CURLY_BRACKET_OPEN
%token CURLY_BRACKET_CLOSED
%token SEMI_COLON
%token COLON

%%

program:    START decllist compstmt ENDPRG  { printf("program -> START decllist compst
        ;
decllist:      { printf("decllist -> E\n");}
        | declaration SEMI_COLON decllist { printf("decllist -> declaration ; decllist
        ;
declaration:     id COLON type  { printf("declaration -> id : type\n");}
```

```
        ;
simple_type:    INT  { printf("simple_type -> INT\n");}
        | BOOLEAN  { printf("simple_type -> BOOLEAN\n");}
        | CHAR  { printf("simple_type -> CHAR\n");}
        | STRING  { printf("simple_type -> STRING\n");}
        ;
array_type:    ARRAY SQUARE_BRACKET_OPEN INT SQUARE_BRACKET_CLOSED simple_type { print
        ;
type:        simple_type  { printf("type -> simple_type\n");}
        | array_type  { printf("type -> array_type\n");}
        ;
compstmt:    BEGIN_STMT stmtlist END  { printf("compstmt -> BEGIN stmtlist END\n");}
        ;
stmtlist:      { printf("stmtlist -> E\n");}
        | stmt SEMI_COLON stmtlist  { printf("stmtlist -> stmt ; stmtlist\n");}
        | stmt stmtlist  { printf("stmtlist -> stmt stmtlist\n");}
        ;
stmt:        simple_stmt  { printf("stmt -> simple_stmt\n");}
        | struct_stmt  { printf("stmt -> struct_stmt\n");}
        ;
simple_stmt:    assign_stmt  { printf("simple_stmt -> assign_stmt\n");}
        | io_stmt  { printf("simple_stmt -> io_stmt\n");}
        ;
assign_stmt:    id ASSIGNED expression  { printf("assign_stmt -> id := expression\n");
        ;
expression:    term signed_expression  { printf("expression -> term signed_expression\
        ;
signed_expression:      { printf("signed_expression -> E\n");}
            | operator expression  { printf("signed_expression -> operator expression\
            ;
term:        id  { printf("term -> id\n");}
        | ct  { printf("term -> ct\n");}
        ;
operator:    ADD  { printf("operator -> +\n");}
        | SUBTRACT  { printf("operator -> -\n");}
        | MULTIPLY  { printf("operator -> *\n");}
        | DIV  { printf("operator -> /\n");}
        | MOD  { printf("operator -> %\n");}
        ;
io_stmt:    READ PARA_OPEN id PARA_CLOSED  { printf("io_stmt -> READ ( id  )\n");}
        | WRITE PARA_OPEN id PARA_CLOSED  { printf("io_stmt -> WRITE ( id  )\n");}
        ;
struct_stmt:    compstmt  { printf("struct_stmt -> compstmt\n");}
        | ifstmt  { printf("struct_stmt -> ifstmt\n");}
        | whilestmt  { printf("struct_stmt -> whilestmt\n");}
        ;
ifstmt:        IF condition THEN compstmt elsestmt  { printf("ifstmt -> IF condition T
        ;
elsestmt:      { printf("elsestmt -> E\n");}
        | ELSE compstmt  { printf("elsestmt -> ELSE compstmt\n");}
        ;
whilestmt:    WHILE condition DO compstmt  { printf("whilestmt -> WHILE condition DO c
        ;
condition:    expression RELATION expression  { printf("consition -> expression RELATI
        ;
```

```
RELATION:    SMALLER  { printf("RELATION -> <\n");}
        | SMALLER_OR_EQUAL  { printf("RELATION -> <=\n");}
        | GREATER  { printf("RELATION -> >\n");}
        | GREATER_OR_EQUAL  { printf("RELATION -> >=\n");}
        | EQUAL  { printf("RELATION -> =\n");}
        | DIFFERENT  { printf("RELATION -> !=\n");}
        | ASSIGNED  { printf("RELATION -> :=\n");}
        | AND  { printf("RELATION -> AND\n");}
        | OR  { printf("RELATION -> OR\n");}
        ;
%%

void 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,"syntactically correct\n");
}
```

Tests

p1.txt

- **Input**

```
START A : INT ;
BEGIN
READ ( A ) ;
END
ENDPRG
```

- **Output**

```
syntactically correct
```

p1err.txt

- **Input**

```
START A : INT ;
BEGIN
READ ( A ) ;
ENDPRG
```

- **Output**

```
syntax error
```

p2.txt

- **Input**

```
START
 A : INT ; B : INT ; AUX : INT ; R : INT ;

 BEGIN
  READ ( A ) ;
  READ ( B ) ;

  IF A > B THEN
   BEGIN
    AUX := A ;
    A := B ;
    B := AUX ;
   END

  WHILE R != 0 DO
   BEGIN
    R := B % A ;
    A := B ;
    B := R ;
   END

  WRITE ( A ) ;
 END
ENDPRG
```

- **Output**

```
syntactically correct
```

p3.txt

- **Input**

```
START
 A := -0
 A:=-15;
 A := -7 - 10;
 A: INT; B: INT; C: INT; MX1: INT; MX: INT;
 BEGIN
  READ (A);
  READ (B);
  READ (C);
  IF A > B THEN
   BEGIN
    MX1 := A ;
   END
  ELSE
   BEGIN
    MX1 := B ;
   END

  IF C > MX1 THEN
   BEGIN
    MX := C ;
   END
  ELSE
   BEGIN
    MX := MX1 ;
   END

  WRITE (MX) ;
 END
ENDPRG
```

- **Output**

```
syntax error
```