Resul Murat DENİZ 05120000307

Tolgahan AKGÜN 05130000248 ANALİZ AŞAMASI

TESLİM TARİHİ:02.05.2014

**ANALİZ AŞAMASI**

Lexical analyzer bir programda verilen girdileri alıp karşılık gelen lexeme’lere ayrılma işlemidir. Yani lexical analyzer her çağrısında tek bir lexeme ve ona karşılık gelen token ı döndürür. Lexical analyzer’ın çalışma sürecinde yorumları atlama ve boşluk karakterlerini geçme işlemi vardır. Ayrıca lexical analyzer sözdizimsel hataları tespit edip kullanıcıya bu hataları rapor eder.

Programlama aşamasında programın karmaşıklığının azalması ve okunabilirliği arttırmak amacıyla alt programlara ihtiyaç duyduk. Bunlardan biri bir çok özelliğe sahip getChar fonksiyonu idi. getChar fonksiyonu çağırıldığında kullanıcının input olarak verdiği programdaki sonraki girdi karakterini alır ve global değişken olan nextChar değişkenine atar. getChar fonksiyonu ayrıca okunan karakterin sınıfını belirlemek ve başka bir global değişken olan charClass a atmak için kullanılır. Lexeme bir karakter dizisidir ve programda sözel ifadelerin çözümleyicisi olarak kullanılmıştır.

nextChar karakteri koyma süreci için addChar fonksiyonunu kullandık.

Lexical analyzer çağrıldığında girdinin ve lexeme’in aynı karakterde olması gerekmektedir. Bu ihtiyacı karşılayabilmek için getNonBlank fonksiyonu ile programdaki boşluk gibi karakterlerin geçilmesi sağlanmıştır.

Son fonksiyonumuz lookup ise programdan aldığı tokenlerı tek tek işleme sokar. Bizim yaptığımız programda bu işlemlere örnek olarak sol parantez mi sağ parantez mi veya çarpma operatörü olup olmadığına bakar.

**TOKENLAR**

**INT\_LIT**

(0,1,2…9) Rakamları alan token. Tamsayı olarak ele alır.

**IDENT**

Programdaki tanıyıcılardır. Pascal dilinde büyük küçük harf ayrımı yoktur değişkenleri aynı kabul eder.(Programda bu özellik göz önüne alınmamıştır)

**KEY\_WORD**

Eğer identifier a karşılık gelen bir anahtar kelime varsa pascal dilinde bunu anahtar kelime olarak alıp dile ait key word kabul eder.

**LOGIC\_OP**

("not","and","or","xor") Dildeki mantıksal işlemleri ifade eden tokendır.

**COMPARISON\_OP**

Karşılaşma operatörlerini ifade eden tokendır.(<,> gibi operatörler)

**STRING\_LIT**

String ifadelerini ifade eden tokendır.

**COMMENT\_LIT**

Kodun belli bir bölümünde (\* veya { ile varsa bu yorum ifadesi olduğunu belirtir. Yorum ifadelerini alan tokendır.

**ASSIGN\_OP**

Eğer programda := ifadesi varsa atama operatörüdür ve o tokenı belirtir.

**ADD\_OP**

Program kodu okuduğunda + varsa artırım operatörü olduğunu ifade eder.

**SUB\_OP**

Program kodu okuduğunda - varsa azaltım operatörü olduğunu ifade eder.

**ASTERISK\_OP**

Program kodu okuduğunda \* varsa çarpım operatörü olduğunu ifade eder.

**SLASH\_OP**

Program kodu okuduğunda / varsa çarpım operatörü olduğunu ifade eder.

**LEFT\_PAREN**

Program kodu okuduğunda ( varsa çarpım operatörü olduğunu ifade eder.

**RIGHT\_PAREN**

Program kodu okuduğunda ) varsa çarpım operatörü olduğunu ifade eder.

**COMMA\_OP**

Program kodu okuduğunda , varsa çarpım operatörü olduğunu ifade eder.

**EXP\_OP**

Program kodu okuduğunda \*\* varsa çarpım operatörü olduğunu ifade eder.

**END\_OF\_LINE**

Program kodu okuduğunda ; varsa çarpım operatörü olduğunu ifade eder.

**SOURCE CODE**

#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

/\* Character classes \*/

#define LETTER 0

#define DIGIT 1

#define LOGIC\_OR\_RELATIONAL 2

#define COMMENT 3

#define COMMENT\_OR\_LP 4

#define UNKNOWN 99

/\* Token codes \*/

#define INT\_LIT 10

#define IDENT 11

#define KEY\_WORD 12

#define LOGIC\_OP 13

#define COMPARISON\_OP 14

#define STRING\_LIT 15

#define COMMENT\_LIT 16

#define ASSIGN\_OP 17

#define ADD\_OP 18

#define SUB\_OP 19

#define ASTERISK\_OP 20

#define SLASH\_OP 21

#define LEFT\_PAREN 22

#define RIGHT\_PAREN 23

#define COMMA\_OP 24

#define EXP\_OP 25

#define REAL\_LIT 26

#define END\_OF\_LINE 27

/\*Error codes\*/

#define EX\_LONG\_ID 100

#define EX\_STRING\_UNCLOSED 101

#define EX\_UNKNOWN 102

#define EX\_UNDEFINED\_COMPORLOGIC\_OPERATOR 103

#define EX\_INCORRECT\_ID 104

char \*keyword[] = {"asm","array","begin","case","const","constructor","destructor","div","do","downto","else",

"end","exports","file","for","function","goto","if","implementation","in","inherited","inline","interface","label",

"library","mod","nil","object","of","packed","procedure","program","record","repeat","set","shl","shr",

"string","then","to","type","unit","until","uses","var","while","with"};

char \*logical[]= {"not","and","or","xor"};

char \*relational[] = {"<",">",">=","<="};

int charClass;

char lexeme [255];

char nextChar='\n';

char previousChar='\n';

char \*tkn\_name;

int lexLen;

int token;

int nextToken;

int line\_no=1;

int column\_no=0;

int error=0;

int temp\_column\_no;

FILE \*in\_fp,\*tkn\_fp;

char commentChar='\n';

#define KEY\_WORD\_SIZE 39

/\* Function declarations \*/

void addChar();

void getChar();

void getNonBlank();

int lex();

int isKeyWord();

int isLogical ();

void ERROR\_MESSAGE(int exception);

char \*token\_name(int token\_code);

int isRealOrInteger();

int main()

{

char filename[50];

char \*new\_filename;

printf("Write the name of the file:");

gets(filename);

/\* Open the input data file and process its contents \*/

if ((in\_fp = fopen(strcat(filename,".pas"), "r")) == NULL)

printf("ERROR - cannot open front.in \n");

else

{

new\_filename = strtok (filename,".");

new\_filename=strcat(new\_filename,".lex");

tkn\_fp=fopen(new\_filename,"w+");

getChar();

do

{

lex();

}

while (nextToken != EOF);

}

return 0;

}

int isLogical()

{

int i;

for(i=0; i<3; i++)

if(strcasecmp(logical[i],lexeme)==0)

return LOGIC\_OP;

ERROR\_MESSAGE(EX\_UNDEFINED\_COMPORLOGIC\_OPERATOR);

return EX\_UNDEFINED\_COMPORLOGIC\_OPERATOR;

}

char \*token\_name(int token\_code)

{

switch(token\_code)

{

case INT\_LIT:

tkn\_name="INT\_CONSTANT";

break;

case IDENT:

tkn\_name="IDENTIFIER";

break;

case KEY\_WORD:

tkn\_name="KEYWORD";

break;

case LOGIC\_OP:

tkn\_name="LOGIC\_OPERATOR";

break;

case COMPARISON\_OP:

tkn\_name="COMPARISON\_OPERATOR";

break;

case STRING\_LIT:

tkn\_name="STRING\_CONSTANT";

break;

case COMMENT\_LIT:

tkn\_name="COMMENT\_CONSTANT";

break;

case ASSIGN\_OP:

tkn\_name="ASSIGNMENT\_OPERATOR";

break;

case ADD\_OP:

tkn\_name="INCREMENT\_OPERATOR";

break;

case SUB\_OP:

tkn\_name="SUBTRACT\_OPERATOR";

break;

case ASTERISK\_OP:

tkn\_name="ASTERISK\_OPERATOR";

break;

case SLASH\_OP:

tkn\_name="SLASH\_OPERATOR";

break;

case LEFT\_PAREN:

tkn\_name="LEFT\_PARENTHESES";

break;

case RIGHT\_PAREN:

tkn\_name="RIGHT\_PARENTHESES";

break;

case COMMA\_OP:

tkn\_name="COMMA\_OPERATOR";

break;

case EXP\_OP:

tkn\_name="EXPONENT\_OPERATOR";

break;

case REAL\_LIT:

tkn\_name="REAL\_CONSTANT";

break;

case END\_OF\_LINE:

tkn\_name="END\_OF\_LINE";

break;

default:

tkn\_name="END\_OF\_FILE";

break;

}

return tkn\_name;

}

int lookup(char ch)

{

switch (ch)

{

case '(':

addChar();

nextToken = LEFT\_PAREN;

break;

case ';':

addChar();

nextToken = END\_OF\_LINE;

break;

case ')':

addChar();

nextToken = RIGHT\_PAREN;

break;

case '+':

addChar();

nextToken = ADD\_OP;

break;

case '-':

addChar();

nextToken = SUB\_OP;

break;

case '\*':

addChar();

nextToken = ASTERISK\_OP;

break;

case '/':

addChar();

nextToken = SLASH\_OP;

break;

case '<':

addChar();

nextToken = COMPARISON\_OP;

break;

case '>':

addChar();

nextToken = COMPARISON\_OP;

break;

case ',':

addChar();

nextToken = COMMA\_OP;

break;

case '=':

addChar();

nextToken=COMPARISON\_OP;

break;

case '\'':

addChar();

temp\_column\_no=column\_no;

do

{

getChar();

addChar();

}

while(nextChar!= '\n' && nextChar != '\'');

if (nextChar == '\'')

nextToken = STRING\_LIT;

else

{

//ERROR\_MESSAGE(EX\_STRING\_UNCLOSED);

nextToken=EX\_STRING\_UNCLOSED;

}

break;

default:

addChar();

nextToken = EX\_UNKNOWN;

//ERROR\_MESSAGE(EX\_UNKNOWN);

break;

}

return nextToken;

}

void addChar()

{

if (lexLen <= 254)//max variable length

{

lexeme[lexLen++] = nextChar;

lexeme[lexLen] = 0; //?????

}

else

printf("Error - lexeme is too long \n");

}

void getChar()

{

previousChar=nextChar;

if ((nextChar = getc(in\_fp)) != EOF)

{

if(nextChar=='{')//????

charClass=COMMENT;

else if (isalpha(nextChar))

charClass = LETTER;

else if (isdigit(nextChar))

charClass = DIGIT;

else if(nextChar=='<' || nextChar=='>' || nextChar=='=')

charClass=LOGIC\_OR\_RELATIONAL;

/\* {

switch(previousChar)

{

case '<': if(nextChar=='=') break;

case '>'

}

}\*/

else if(nextChar==':')

charClass=LOGIC\_OR\_RELATIONAL;

else if(nextChar=='(')

charClass=COMMENT\_OR\_LP;

else

charClass = UNKNOWN;

//charClass = UNKNOWN;

}

else

charClass = EOF;

column\_no++;

if(nextChar=='\n')

{

line\_no++;

column\_no=0;

}

}

void getNonBlank()

{

while (isspace(nextChar))

getChar();

}

int lex()

{

lexLen = 0;

int long\_error\_flag=0;

error=0;

getNonBlank();

switch (charClass)

{

/\* Parse identifiers \*/

case LETTER:

addChar();

getChar();

while (charClass == LETTER || charClass == DIGIT || nextChar=='\_')

{

if(lexLen!=31)

{

addChar();

getChar();

}

else

{

getChar();

long\_error\_flag=1;

}

}

if(long\_error\_flag==1)

{

ERROR\_MESSAGE(EX\_LONG\_ID);

nextToken=EX\_LONG\_ID;

}

else

nextToken = isKeyWord();

break;

/\* Parse integer literals \*/

case DIGIT:

addChar();

getChar();

while (charClass == DIGIT)

{

addChar();

getChar();

}

if(charClass==LETTER)

{

ERROR\_MESSAGE(EX\_INCORRECT\_ID);

nextToken=EX\_INCORRECT\_ID;

while (charClass == LETTER || charClass == DIGIT)

{

getChar();

}

}

else

nextToken=isRealOrInteger();

break;

case LOGIC\_OR\_RELATIONAL:

addChar();

getChar();

/\*while (charClass == LETTER )

{

addChar();

getChar();

}\*/

//nextToken=isLogical();

/\*if(nextChar=='>' || nextChar=='<' || nextChar=='=')//buraya diðer operatorleride eklemek lazým

{

//addChar();

//getChar();

if(nextChar=='=' || nextChar=='>')

nextToken=COMPARISON\_OP;

}\*/

if(previousChar=='<' && nextChar=='=')

{

nextToken=COMPARISON\_OP;

addChar();

getChar();

}

else if(previousChar=='>' && nextChar=='=')

{

nextToken=COMPARISON\_OP;

addChar();

getChar();

}else if(previousChar=='<')

{

nextToken=COMPARISON\_OP;

//addChar();

//getChar();

}else if(previousChar=='>')

{

nextToken=COMPARISON\_OP;

//addChar();

//getChar();

}

else if(previousChar=='=' && nextChar==' ')

nextToken=ASSIGN\_OP;

if(previousChar==':'&& nextChar=='=')

{

nextToken=ASSIGN\_OP;

addChar();

getChar();

}

break;

case COMMENT:

//addChar();

while(nextChar!='}' && nextChar!=EOF)

{

//commentChar=nextChar;

nextChar=getc(in\_fp);

//getChar();

if(nextChar!='}')

addChar();

/\*if(lexeme[lexLen-1]=='}' || lexeme[lexLen-1]=='\n'){

removeChar();

removeChar();

}

/\*if(lexeme[lexLen-1]=='\*')

removeChar();

if(nextChar!='(')

addCommentLit(commentChar);\*/

/\*if((commentChar=='\*' && nextChar==')') || nextChar=='}'){

getChar();

addChar();

break;

}\*/

}

while(nextChar!='\n' && nextChar!=' ' && nextChar!=EOF)

nextChar=getc(in\_fp);

line\_no++;

nextToken=COMMENT\_LIT;

//if(nextChar=='\n')nextToken=EOF;

if(nextChar==EOF)

charClass=EOF;

break;

// addChar();

// getchar();

//nextChar=getc(in\_fp);

/\* Parentheses and operators \*/

case COMMENT\_OR\_LP:

commentChar=getc(in\_fp);

if(nextChar=='(' && commentChar=='\*'){

//fseek(in\_fp,sizeof(char),SEEK\_CUR);

//charClass=UNKNOWN;

commentChar='0';

while(commentChar!='\*' || nextChar!=')')

{

commentChar=nextChar;

nextChar=getc(in\_fp);

addChar();

}

removeChar();

removeChar();

nextToken=COMMENT\_LIT;

nextChar=getc(in\_fp);

break;

}

else fseek(in\_fp,-sizeof(char),SEEK\_CUR);

case UNKNOWN:

lookup(nextChar);

getChar();

if(nextToken==ASTERISK\_OP && nextChar=='\*')

{

addChar();

nextToken=EXP\_OP;

getChar();

}

break;

/\* EOF \*/

case EOF:

nextToken = EOF;

lexeme[0] = 'E';

lexeme[1] = 'O';

lexeme[2] = 'F';

lexeme[3] = 0;

break;

} /\* End of switch \*/

if(error!=1)

{

if(nextToken==STRING\_LIT)

{

printf("%s(%s)\n",token\_name(nextToken),lexeme);

fprintf(tkn\_fp,"%s(%s)\n",token\_name(nextToken),lexeme);

}

else

{

printf("%s(''%s'')\n",token\_name(nextToken),lexeme);

fprintf(tkn\_fp,"%s(''%s'')\n",token\_name(nextToken),lexeme);

}

}

else if(nextToken==EX\_UNKNOWN)

fprintf(tkn\_fp,"ERROR Unknown character! %s\n",lexeme);

else if(nextToken==EX\_STRING\_UNCLOSED)

fprintf(tkn\_fp,"ERROR Line %d Column %d: Unclosed String Literal!\n",line\_no-1,temp\_column\_no);

else if(nextToken==EX\_LONG\_ID)

fprintf(tkn\_fp,"ERROR Identifier is too long! It has no more than 31 characters.\n");

else if(nextToken==EX\_UNDEFINED\_COMPORLOGIC\_OPERATOR)

fprintf(tkn\_fp,"ERROR Undefined comparison operator!\n");

else if(nextToken==EX\_INCORRECT\_ID)

fprintf(tkn\_fp,"ERROR Line %d: Identifiers must start with a letter!\n",line\_no-1);

return nextToken;

} /\* End of function lex \*/

void ERROR\_MESSAGE(int exception)

{

error=1;

switch(exception)

{

case EX\_LONG\_ID:

printf("ERROR:Line %d : Identifier is too long! It has no more than 31 characters.\n",line\_no-1);

break;

case EX\_STRING\_UNCLOSED:

printf("ERROR Line %d Column %d: Unclosed String Literal!\n",line\_no-1,temp\_column\_no);

break;

case EX\_UNKNOWN:

printf("ERROR Line %d Column %d: Unknown character! %s\n",line\_no,column\_no,lexeme);

break;

case EX\_UNDEFINED\_COMPORLOGIC\_OPERATOR:

printf("ERROR Line %d: Undefined comparison operator!\n",line\_no-1);

break;

case EX\_INCORRECT\_ID:

printf("ERROR Line %d: Identifiers must start with a letter!\n",line\_no);

}

}

int isKeyWord(){

int i;

if(strcasecmp(lexeme,"END")==0)

{

getNonBlank();

if(charClass==LETTER)

{

lexeme[lexLen++]=' ';

lexeme[lexLen]=0;

while(charClass==LETTER)

{

addChar();

getChar();

}

}

return KEY\_WORD;

}

for(i=0; i<KEY\_WORD\_SIZE; i++)

if(strcasecmp(keyword[i],lexeme)==0)

return KEY\_WORD;

for(i=0; i<4; i++)

if(strcasecmp(logical[i],lexeme)==0)

return LOGIC\_OP;

return IDENT;

}

int isRealOrInteger()

{

if(nextChar=='.')

{

addChar();

getChar();

while(charClass==DIGIT || nextChar=='D' || nextChar=='E' || nextChar=='F' || nextChar=='G')

{

addChar();

getChar();

}

return REAL\_LIT;

}

else return INT\_LIT;

}

void removeChar()

{

lexeme[--lexLen]=NULL;

}

**ÖRNEKKOD**

{Pascal Lexical Analyzer Deneme}

(\*Author:Murat DENIZ and Tolgahan AKGUN\*)

program crt\_colour\_list;

uses crt;

var counter:integer;

begin;

clrscr;

for counter:=0 to 15 do

begin;

textcolor(counter);

writeln(counter);

end;

readln;

end.

**ÇIKTISI**

