Compiler Design Lab

CSS651

Group – 2

Members

Saharsh Ananta Jaiswal – 18CS8061

Pudi Pavan Kumar – 18CS8062

Ayesha Uzma – 18CS8063

Rithik Sureka – 18CS8064

Ravupalli Harsha Vardhan – 18CS8065

Assignment: 1

Lexical Analyzer in C language.

Software Used: Code::Blocks(IDE)

Theory:

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre-processors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

The lexical analyzer collects also information about tokens into their associated attributes:

- The tokens influence parsing decisions,
- The attributes influence the translation of tokens.

A lexical token or simply token is a string with an assigned and thus identified meaning. It is structured as a pair consisting of a token name and an optional token value. The token name is a category of lexical unit. Common token names are-

- identifier: names the programmer chooses;
- keyword: names already in the programming language;
- separator (also known as punctuators): punctuation characters and paired-delimiters;
- operator: symbols that operate on arguments and produce results;
- literal: numeric, logical, textual, reference literals;
- comment: line, block (Depends on the compiler if compiler implements comments as tokens otherwise it will be stripped).

Need of Lexical Analyzer

- Simplicity of design of compiler - The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.
- Compiler efficiency is improved - Specialized buffering techniques for reading characters speed up the compiler process.
- Compiler portability is enhanced

Code:

```c
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Returns 'true' if the character is a DELIMITER
bool isDelimiter(char ch)
{
    if (ch == ' ' || ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == ',' || ch == ';' || ch == '>' ||
        ch == '<' || ch == '=' || ch == '(' || ch == ')' ||
        ch == '[' || ch == ']' || ch == '{' || ch == '}' ||
        ch == '%' || ch == '!' || ch == '&' || ch == '|' ||
        ch == '\n' )
        return (true);
    return (false);
}

//Returns 'true' if the character can be continuation of a single operator
bool isOperator_secondHalf(char ch)
{
    if (ch == '=' || ch == '&' ||
        ch == '|' || ch == '+' || ch == '-')
        return (true);
    return (false);
}

//Returns 'true' if the character is a punctuator
bool isPunctuator(char ch)
{
    if(ch == '{' || ch == '}' ||
       ch == ',' || ch == ';')
        return (true);
    return (false);
}

// Returns 'true' if the character is an OPERATOR.
bool isOperator(char ch)
{
    if (ch == '+' || ch == '-' || ch == '*' ||
        ch == '/' || ch == '%')
        return (true);
    return (false);
}

//Returns 'true' if the character is the assignment operator
bool isAssignment(char ch)
{
    if(ch=='=')
        return (true);
    return (false);
}

//Returns 'true' if character array is a valid logical operator
bool isLogical_operator(char *arr)
{
    if (!strcmp(arr, "&&") || !strcmp(arr,"||") || !strcmp(arr, "!"))
        return (true);
    return (false);
}
```

```
//Returns 'true' if character array is a valid relational operator
bool isRelational_operator(char *arr)
{
    if (!strcmp(arr, ">") || !strcmp(arr, ">=") ||
        !strcmp(arr, "<") || !strcmp(arr, "<=") ||
        !strcmp(arr, "==") || !strcmp(arr, "!=") )
        return (true);
    return (false);
}

//Return 'true' if the character array is an increment or decrement operator
bool isOperator2(char *arr)
{
    if (!strcmp(arr, "++") || !strcmp(arr, "--"))
        return (true);
    return (false);
}

// Returns 'true' if the string is a VALID IDENTIFIER.
bool validIdentifier(char* str)
{
    int val = (int)str[0];
    if(val >= 48 && val <= 57 || isDelimiter(str[0]) == true)
        return (false);
    if ((val >= 65 && val <= 90) || (val >= 97 && val <= 122) )
    {
        return (true);
    }
    return (false);
}

// Returns 'true' if the string is a KEYWORD.
bool isKeyword(char* str)
{
    if (!strcmp(str, "if") || !strcmp(str, "else") ||
        !strcmp(str, "while") || !strcmp(str, "do") ||
        !strcmp(str, "break") || !strcmp(str,"for") ||
        !strcmp(str, "continue") || !strcmp(str, "case") ||
        !strcmp(str, "switch") || !strcmp(str, "goto") ||
        !strcmp(str,"else if"))
        return (true);
    return (false);
}

// Returns 'true' if the string is an INTEGER.
bool isInteger(char* str)
{
    int i, len = strlen(str);

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' || (str[i] == '-' && i > 0))
            return (false);
    }
    return (true);
}

// Returns 'true' if the string is a REAL NUMBER.
bool isRealNumber(char* str)
{
    int i, len = strlen(str);
```

```c
    bool hasDecimal = false;

    if (len == 0)
        return (false);
    for (i = 0; i < len; i++) {
        if (str[i] != '0' && str[i] != '1' && str[i] != '2'
            && str[i] != '3' && str[i] != '4' && str[i] != '5'
            && str[i] != '6' && str[i] != '7' && str[i] != '8'
            && str[i] != '9' && str[i] != '.' ||
            (str[i] == '-' && i > 0))
            return (false);
        if (str[i] == '.')
            hasDecimal = true;
    }
    return (hasDecimal);
}

// Extracts the SUBSTRING.
char* subString(char* str, int left, int right)
{
    int i;
    char* subStr = (char*)malloc(
                sizeof(char) * (right - left + 2));

    for (i = left; i <= right; i++)
        subStr[i - left] = str[i];
    subStr[right - left + 1] = '\0';
    return (subStr);
}

// Parsing the input STRING.
void parse(char* str)
{
    int left = 0, right = 0;
    int len = strlen(str);

    while (right <= len && left <= right)
    {
        if (isDelimiter(str[right]) == false)
            right++;

        if (isDelimiter(str[right]) == true && left == right)
        {
            char ch = str[right];
            int temp = right+1;
            char arr[] = "";
            arr[0] = ch;
            arr[1] = '\0';
            if(isPunctuator(ch) == true)
                printf("'%c' -> PUNCTUATOR\n", str[right]);
            else if(isOperator_secondHalf(str[temp])== true)
            {
                arr[1] = str[temp];
                arr[2] = '\0';
                if(isLogical_operator(arr) == true)
                {
                    printf("'%s' -> LOGICAL OPERATOR\n", arr);
                    right = temp;
                }
                else if (isRelational_operator(arr) == true)
                {
                    printf("'%s' -> RELATIONAL OPERATOR\n", arr);
                    right = temp;
                }
                else if (isOperator2(arr) == true)
                {
```

```c
                    printf("'%s' -> INCREMENT/DECREMENT OPERATOR\n", arr);
                    right = temp;
                }
            }
            if(right!=temp)
            {
                arr[1] = '\0';
                if (isOperator(ch) == true)
                    printf("'%c' -> ARITHMETIC OPERATOR\n", str[right]);
                else if (isAssignment(ch) == true)
                    printf("'%c' -> ASSIGNMENT OPERATOR\n", str[right]);
                else if (isLogical_operator(arr) == true)
                    printf("'%c' -> LOGICAL OPERATOR\n", str[right]);
                else if (isRelational_operator(arr) == true)
                    printf("'%c' -> RELATIONAL OPERATOR\n", str[right]);
                else if (ch == '&' || ch == '|')
                    printf("'%c' -> UNRECOGNIZED TOKEN\n", str[right]);
            }
            right++;
            left = right;
        }
        else if (isDelimiter(str[right]) == true && left != right
                || (right == len && left != right))
        {
            char* subStr = subString(str, left, right - 1);

            if (isKeyword(subStr) == true)
                printf("'%s' -> KEYWORD\n", subStr);

            else if (isInteger(subStr) == true)
                printf("'%s' -> INTEGER\n", subStr);

            else if (isRealNumber(subStr) == true)
                printf("'%s' -> REAL NUMBER\n", subStr);

            else if (validIdentifier(subStr) == true && isDelimiter(str[right - 1]) == false)
                printf("'%s' -> IDENTIFIER\n", subStr);

            else if (validIdentifier(subStr) == false && isDelimiter(str[right - 1]) == false)
                printf("'%s' -> INVALID IDENTIFIER\n",subStr);
            left = right;
        }
    }
    return;
}

// DRIVER FUNCTION
int main()
{
    // maximum length of string is 100 here
    printf("\tSYMBOL TABLE\n");
    printf("\nTOKEN \t\tCOMMENT\n");
    //sample
    char str[100] = "void main()\n{  int a,b,c,d,a2,3b;\n     int a = b ++ (c*d);\n    ad >= d;\n}";

    parse(str); // calling the parse function
    return (0);
}
```

Output:

```
"C:\Users\IDEAPAD 330S\Documents\Lexical_Analyzer.exe"
        SYMBOL TABLE

TOKEN           COMMENT
'void' -> IDENTIFIER
'main' -> IDENTIFIER
'{' -> PUNCTUATOR
'int' -> IDENTIFIER
'a' -> IDENTIFIER
',' -> PUNCTUATOR
'b' -> IDENTIFIER
',' -> PUNCTUATOR
'c' -> IDENTIFIER
',' -> PUNCTUATOR
'd' -> IDENTIFIER
',' -> PUNCTUATOR
'a2' -> IDENTIFIER
',' -> PUNCTUATOR
'3b' -> INVALID IDENTIFIER
';' -> PUNCTUATOR
'int' -> IDENTIFIER
'a' -> IDENTIFIER
'=' -> ASSIGNMENT OPERATOR
'b' -> IDENTIFIER
'++' -> INCREMENT/DECREMENT OPERATOR
'c' -> IDENTIFIER
'*' -> ARITHMETIC OPERATOR
'd' -> IDENTIFIER
';' -> PUNCTUATOR
'ad' -> IDENTIFIER
'>=' -> RELATIONAL OPERATOR
'd' -> IDENTIFIER
';' -> PUNCTUATOR
'}' -> PUNCTUATOR

Process returned 0 (0x0)   execution time : 0.043 s
Press any key to continue.
```

Assignment: 2

Implement a shift reduce parser whose input is a context free grammar and the input sentences and the output is the sequence of productions to be used to reduce to the start symbol.

Software Used: Code::Blocks(IDE)

Theory:

A shift-reduce parser scans and parses the input text in one forward pass over the text, without backing up. The parser builds up the parse tree incrementally, bottom up, and left to right, without guessing or backtracking. At every point in this pass, the parser has accumulated a list of sub trees or phrases of the input text that have been already parsed. Those sub trees are not yet joined together because the parser has not yet reached the right end of the syntax pattern that will combine them.

A shift-reduce parser works by doing some combination of Shift steps and Reduce steps, hence the name.

- A **Shift** step advances in the input stream by one symbol. That shifted symbol becomes a new single-node parse tree.
- A **Reduce** step applies a completed grammar rule to some of the recent parse trees, joining them together as one tree with a new root symbol.

The parser continues with these steps until all of the input has been consumed and all of the parse trees have been reduced to a single tree representing an entire legal input. A more general form of shift reduce parser is LR parser. This parser requires some data structures i.e.

- A input buffer for storing the input string
- A stack for storing and accessing the production rules.

The goto table is a table with rows indexed by states and columns indexed by non-terminal symbols. When the parser is in state s immediately after reducing by value N, then the next state to enter is given by goto[s][N].

The current state of a shift-reduce parser is the state on top of the state stack. The detailed operation of such a parser is as follows:

1. Initialize the parse stack to contain a single state s_0, where s_0 is the distinguished initial state of the parser.
2. Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:
   - If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.
   - If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r. Then consult the goto table and push the state given by goto[s'][N] onto the stack. The look ahead token is not changed by this step.
   - If the action table entry is accept, then terminate the parse with success.
   - If the action table entry is error, then signal an error.
3. Repeat step (2) until the parser terminates.

Code:

```
//Shift Reduce Parser

#include<stdio.h>
#include<string.h>

int k=0,z=0,i=0,j=0,c=0;
char a[16],ac[20],stk[15],act[10];
void check();
int main()
   {
      //get the grammar input from user
      puts("GRAMMAR is E->E+E \n E->E*E \n E->(E) \n E->id");
      puts("enter input string ");
      gets(a);
      c=strlen(a);
      strcpy(act,"SHIFT->");
      puts("stack \t input \t action");
      for(k=0,i=0; j<c; k++,i++,j++)
       {
         if(a[j]=='i' && a[j+1]=='d')
           {
              stk[i]=a[j];
              stk[i+1]=a[j+1];
              stk[i+2]='\0';
              a[j]=' ';
              a[j+1]=' ';
              printf("\n$%s\t%s$\t%sid",stk,a,act);
              check();
           }
         else
           {
              stk[i]=a[j];
              stk[i+1]='\0';
              a[j]=' ';
              printf("\n$%s\t%s$\t%ssymbols",stk,a,act);
              check();
           }
       }

   }
void check()
   {
     strcpy(ac,"REDUCE TO E");
     for(z=0; z<c; z++)
       if(stk[z]=='i' && stk[z+1]=='d')
         {
           stk[z]='E';
           stk[z+1]='\0';
           printf("\n$%s\t%s$\t%s",stk,a,ac);
           j++;
         }
     for(z=0; z<c; z++)
       if(stk[z]=='E' && stk[z+1]=='+' && stk[z+2]=='E')
         {
           stk[z]='E';
           stk[z+1]='\0';
           stk[z+2]='\0';
           printf("\n$%s\t%s$\t%s",stk,a,ac);
           i=i-2;
         }
     for(z=0; z<c; z++)
       if(stk[z]=='(' && stk[z+1]=='E' && stk[z+2]==')')
         {
           stk[z]='E';
           stk[z+1]='\0';
           stk[z+1]='\0';
           printf("\n$%s\t%s$\t%s",stk,a,ac);
           i=i-2;
         }
   }
```

Output:

"C:\Users\IDEAPAD 330S\Documents\Shift_Reduce_Parser.exe"

```
GRAMMAR is E->E+E
 E->E*E
 E->(E)
 E->id
enter input string
id+id+id*id+id
stack    input   action

$id       +id+id*id+id$ SHIFT->id
$E        +id+id*id+id$ REDUCE TO E
$E+        id+id*id+id$ SHIFT->symbols
$E+id        +id*id+id$ SHIFT->id
$E+E         +id*id+id$ REDUCE TO E
$E           +id*id+id$ REDUCE TO E
$E+           id*id+id$ SHIFT->symbols
$E+id           *id+id$ SHIFT->id
$E+E            *id+id$ REDUCE TO E
$E              *id+id$ REDUCE TO E
$E*              id+id$ SHIFT->symbols
$E*id              +id$ SHIFT->id
$E*E               +id$ REDUCE TO E
$E                 +id$ REDUCE TO E
$E+                 id$ SHIFT->symbols
$E+id                 $ SHIFT->id
$E+E                  $ REDUCE TO E
$E                    $ REDUCE TO E
Process returned 0 (0x0)   execution time : 22.130 s
Press any key to continue.
```