Department of Computer Engineering

Academic Term : Jan-May 23-24

Class: T.E. (Computer)
Subject Name : System Programming and Compiler Construction
Subject Code : (CPC601)

Practical No:	01
Title:	SYMBOL TABLE
Date of Performance:	24/01
Date of Submission:	31/01
Roll No:	9539
Name of the Student:	CRYSTAL FERNANDES

Evaluation:

Sr. No	Rubric	Grade
1	Timeline (2)	01
2	Output(3)	03
3	Code optimization (2)	02
4	Postlab (3)	03
		09

Signature of the Teacher: 14/02/2024.

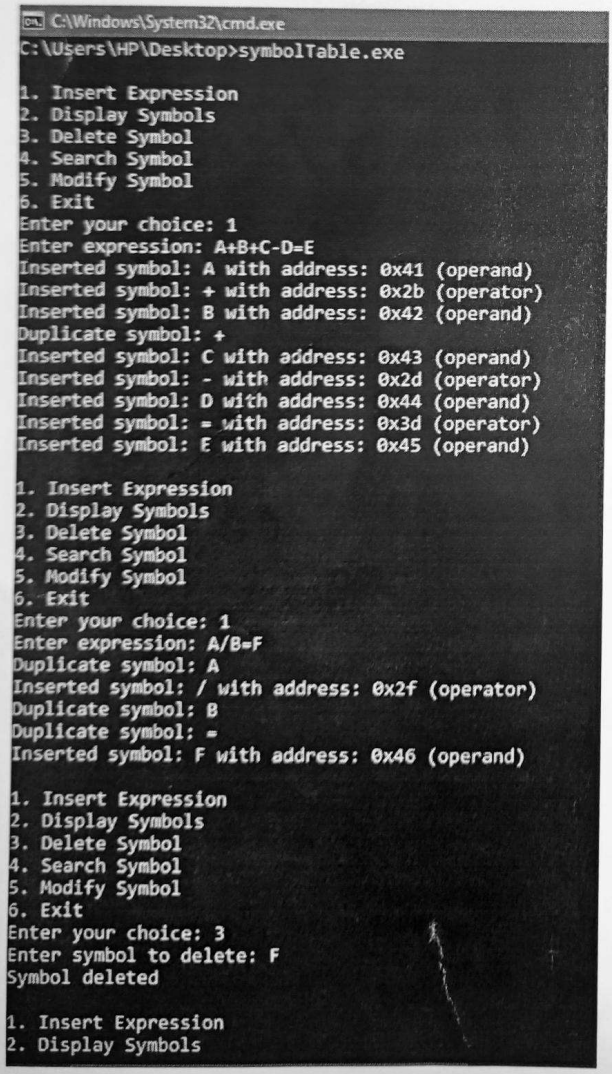
```
#include <iostream>
#include <map>
#include <string>
#include <sstream>
#include <algorithm>
#include <vector>
#include <cctype>

using namespace std;

struct SymbolTableEntry {
    string address;
    string type;
};

map<string, SymbolTableEntry> symbol_table;

string generateAddress(const string& symbol) {
    int sum = 0;

    for (char ch : symbol) {
        sum += ch;
    }

    stringstream ss;
    ss << hex << sum;

    return "0x" + ss.str();
}

bool isKeyword(const string& symbol) {
    static const vector<string> keywords = {"if", "else", "while", "for", "switch", "case", "default", "break",
"continue", "return", "int", "float", "char", "double", "void", "bool", "auto", "const", "extern", "static",
"register", "volatile", "nullptr", "do", "goto", "sizeof", "struct", "typedef", "union", "unsigned", "signed",
"long", "short", "enum", "true", "false", "and", "or", "not"};
    return find(keywords.begin(), keywords.end(), symbol) != keywords.end();
}

bool isOperator(char ch) {
    static const string operators = "+-*/%=><&|^!~";
    return operators.find(ch) != string::npos;
}

bool isOperand(const string& symbol) {
    return !isKeyword(symbol) && !all_of(symbol.begin(), symbol.end(), [](char ch) { return
isOperator(ch) || isspace(ch); });
}

void insertExpressionSymbols(const string& expression) {
    string symbol;
    for (char ch : expression) {
        if (isspace(ch) || isOperator(ch)) {
```

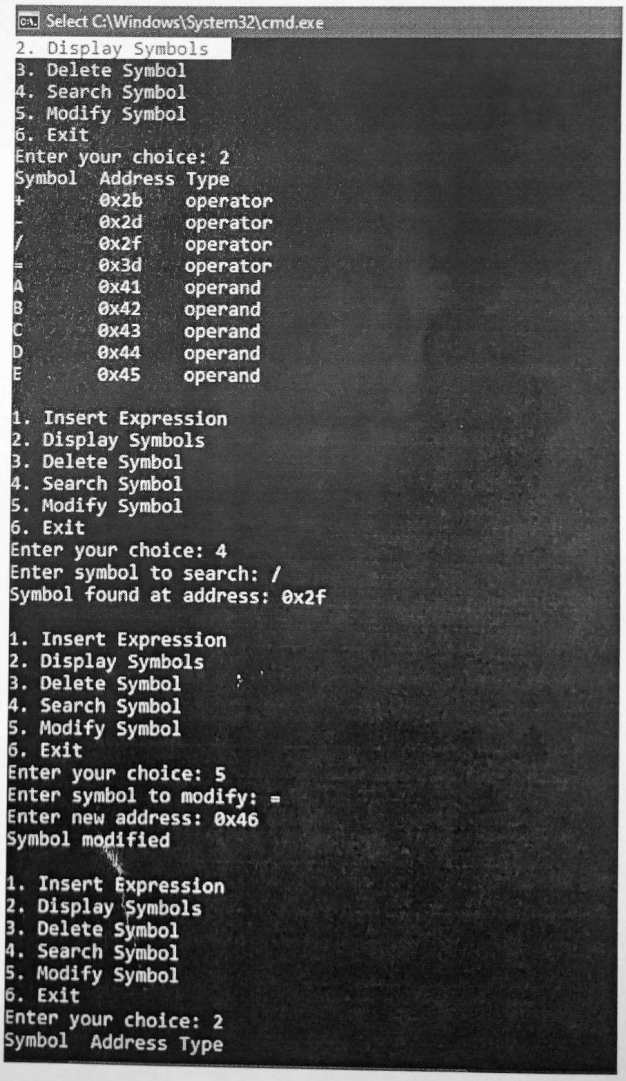
```
            if (!symbol.empty()) {
                if (isOperand(symbol)) {
                    if (symbol_table.find(symbol) == symbol_table.end()) {
                        string address = generateAddress(symbol);
                        symbol_table[symbol] = {address, "operand"};
                        cout << "Inserted symbol: " << symbol << " with address: " << address << " (operand)"
    << endl;
                    } else {
                        cout << "Duplicate symbol: " << symbol << endl;
                    }
                }
                symbol.clear();
            }
            if (isOperator(ch)) {
                string op(1, ch);
                if (symbol_table.find(op) == symbol_table.end()) {
                    string address = generateAddress(op);
                    symbol_table[op] = {address, "operator"};
                    cout << "Inserted symbol: " << op << " with address: " << address << " (operator)" <<
    endl;
                } else {
                    cout << "Duplicate symbol: " << op << endl;
                }
            }
        } else {
            symbol += ch;
        }
    }
    if (!symbol.empty()) {
        if (isOperand(symbol)) {
            if (symbol_table.find(symbol) == symbol_table.end()) {
                string address = generateAddress(symbol);
                symbol_table[symbol] = {address, "operand"};
                cout << "Inserted symbol: " << symbol << " with address: " << address << " (operand)" <<
endl;
            } else {
                cout << "Duplicate symbol: " << symbol << endl;
            }
        }
    }
}

int main() {
    int choice;
    string expression;

    while (true) {
        cout << "\n1. Insert Expression\n"
             "2. Display Symbols\n"
             "3. Delete Symbol\n"
             "4. Search Symbol\n"
             "5. Modify Symbol\n"
```

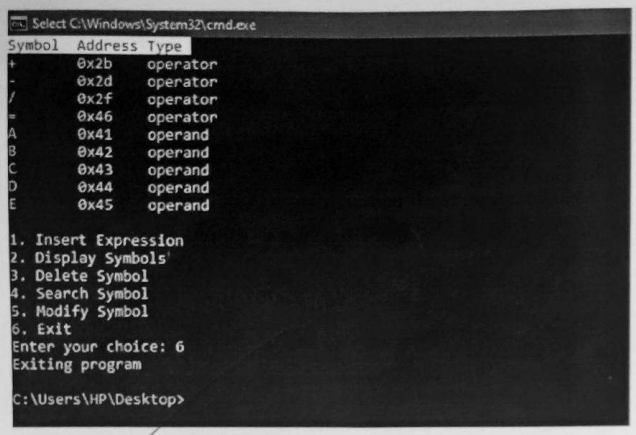
```
        "6. Exit\n";
    cout << "Enter your choice: ";
    cin >> choice;

    switch (choice) {
      case 1:
        cout << "Enter expression: ";
        cin.ignore();
        getline(cin, expression);
        insertExpressionSymbols(expression);
        break;

      case 2:
        cout << "Symbol\tAddress\tType\n";
        for (const auto& entry : symbol_table) {
          cout << entry.first << "\t" << entry.second.address << "\t" << entry.second.type << endl;
        }
        break;

      case 3:
        {
          string symbol;
          cout << "Enter symbol to delete: ";
          cin >> symbol;
          if (symbol_table.find(symbol) == symbol_table.end()) {
            cout << "Symbol not found\n";
          } else {
            symbol_table.erase(symbol);
            cout << "Symbol deleted\n";
          }
        }
        break;

      case 4:
        {
          string symbol;
          cout << "Enter symbol to search: ";
          cin >> symbol;
          if (symbol_table.find(symbol) != symbol_table.end()) {
            cout << "Symbol found at address: " << symbol_table[symbol].address << endl;
          } else {
            cout << "Symbol not found\n";
          }
        }
        break;

      case 5:
        {
          string symbol;
          cout << "Enter symbol to modify: ";
          cin >> symbol;
          if (symbol_table.find(symbol) == symbol_table.end()) {
```

```
bool isInteger(char* str)
{
	if (str == NULL || *str == '\0') {
		return false;
	}
	int i = 0;
	while (isdigit(str[i])) {
		i++;
	}
	return str[i] == '\0';
}

char* getSubstring(char* str, int start, int end)
{
	int length = strlen(str);
	int subLength = end - start + 1;
	char* subStr = new char[subLength + 1];
	strncpy(subStr, str + start, subLength);
	subStr[subLength] = '\0';
	return subStr;
}

int lexicalAnalyzer(char* input)
{
	int left = 0, right = 0;
	int len = strlen(input);

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

		if (isDelimiter(input[right]) && left == right) {
			if (isOperator(input[right]))
				cout << "Token: Operator, Value: " << input[right] << endl;

			right++;
			left = right;
		}
		else if (isDelimiter(input[right]) && left != right
				|| (right == len && left != right)) {
			char* subStr = getSubstring(input, left, right - 1);

			if (isKeyword(subStr))
				cout << "Token: Keyword, Value: " << subStr << endl;

			else if (isInteger(subStr))
				cout << "Token: Integer, Value: " << subStr << endl;

			else if (isValidIdentifier(subStr)
					&& !isDelimiter(input[right - 1]))
				cout << "Token: Identifier, Value: " << subStr << endl;
```

```
            cout << "Symbol not found\n";
        } else {
            cout << "Enter new address: ";
            string address;
            cin >> address;
            symbol_table[symbol].address = address;
            cout << "Symbol modified\n";
        }
    }
    break;

case 6: // Exit the program
    cout << "Exiting program\n";
    return 0;

default:
    cout << "Invalid choice. Please enter a valid choice.\n";
  }
 }
}
```






Select C:\Windows\System32\cmd.exe
Symbol Address Type
+ 0x2b operator
- 0x2d operator
/ 0x2f operator
= 0x46 operator
A 0x41 operand
B 0x42 operand
C 0x43 operand
D 0x44 operand
E 0x45 operand
1. Insert Expression
2. Display Symbols
3. Delete Symbol
4. Search Symbol
5. Modify Symbol
6. Exit
Enter your choice: 6
Exiting program
C:\Users\HP\Desktop>

9539 - EXPERIMENT 1 - SPCC

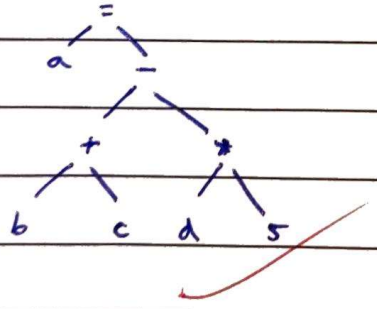
Q1] COMPILER PHASES

- Lexical Analysis : Source code broken into tokens (identifiers, keywords, literals) by a lexer.
- Syntax Analysis : Tokens are analyzed to determine their syntactic structure based on the grammar rules of the language
- Semantic Analysis : Checks for semantic correctness (variable declaration, type compatibility).
- Intermediate Code Generation : Translates source code into intermediate representation
- Code Optimization : memory & space optimization
- Code generation : translates optimized intermediate code into machine code for the target architecture

Eg : a = b + c - d * 5

Tokens : [a, =, b, +, c, -, d, *, 5]

Syntax Tree :



Intermediate Code : ... optimized is same

t1 = b + c

t2 = d * 5

a = t1 - t2

Machine code

```
LOAD b          LOAD t1
ADD c           SUB t2
STORE t1        STORE a
LOAD d
MUL 5
STORE t2
```

14/08/2024

Department of Computer Engineering

Academic Term : Jan-May 23-24

Class: T.E. (Computer)
Subject Name : System Programming and Compiler Construction
Subject Code : (CPC601)

Practical No:	02
Title:	LEXICAL ANALYZER
Date of Performance:	31/01
Date of Submission:	14/02
Roll No:	9539
Name of the Student:	CRYSTAL FERNANDES

Evaluation:

Sr. No	Rubric	Grade
1	Timeline (2)	02
2	Output(3)	03
3	Code optimization (2)	02
4	Postlab (3)	03
		10

Signature of the Teacher: 14/02/2024

```
#include <iostream>
#include <cstring>

#define MAX_LENGTH 100

using namespace std;

bool isDelimiter(char chr)
{
	return (chr == ' ' || chr == '+' || chr == '-' ||
			chr == '*' || chr == '/' || chr == ',' ||
			chr == ';' || chr == '%' || chr == '>' ||
			chr == '<' || chr == '=' || chr == '(' ||
			chr == ')' || chr == '[' || chr == ']' ||
			chr == '{' || chr == '}');
}

bool isOperator(char chr)
{
	return (chr == '+' || chr == '-' || chr == '*' ||
			chr == '/' || chr == '>' || chr == '<' ||
			chr == '=');
}

bool isValidIdentifier(char* str)
{
	return (str[0] != '0' && str[0] != '1' && str[0] != '2'
			&& str[0] != '3' && str[0] != '4'
			&& str[0] != '5' && str[0] != '6'
			&& str[0] != '7' && str[0] != '8'
			&& str[0] != '9' && !isDelimiter(str[0]));
}

bool isKeyword(char* str)
{
	const char* keywords[] =
		{ "auto", "break", "case", "char",
			"const", "continue", "default", "do",
			"double", "else", "enum", "extern",
			"float", "for", "goto", "if",
			"int", "long", "register", "return",
			"short", "signed", "sizeof", "static",
			"struct", "switch", "typedef", "union",
			"unsigned", "void", "volatile", "while" };
	for (int i = 0;
		i < sizeof(keywords) / sizeof(keywords[0]); i++) {
		if (strcmp(str, keywords[i]) == 0) {
			return true;
		}
	}
	return false;
}
```

```
                else if (!isValidIdentifier(subStr)
                        && !isDelimiter(input[right - 1]))
                    cout << "Token: Unidentified, Value: " << subStr << endl;

                left = right;
            }
        }
        return 0;
}

int main()
{
        // Input 01
        char lex_input[MAX_LENGTH] = "int a = b + c";
        cout << "For Expression \"" << lex_input << "\":" << endl;
        lexicalAnalyzer(lex_input);
        cout << endl;

        // Input 02
        char lex_input01[MAX_LENGTH] = "int x=ab+bc+30+switch+ 0y ";
        cout << "For Expression \"" << lex_input01 << "\":" << endl;
        lexicalAnalyzer(lex_input01);

        return 0;
}
```

```
For Expression "int a = b + c":
Token: Keyword, Value: int
Token: Identifier, Value: a
Token: Operator, Value: =
Token: Identifier, Value: b
Token: Operator, Value: +
Token: Identifier, Value: c

For Expression "int x=ab+bc+30+switch+ 0y ":
Token: Keyword, Value: int
Token: Identifier, Value: x
Token: Operator, Value: =
Token: Identifier, Value: ab
Token: Operator, Value: +
Token: Identifier, Value: bc
Token: Operator, Value: +
Token: Integer, Value: 30
Token: Operator, Value: +
Token: Keyword, Value: switch
Token: Operator, Value: +
Token: Unidentified, Value: 0y
```

9539 - EXPERIMENT 2 - SPCC

Q1] ROLE OF AUTOMATA THEORY IN COMPILER DESIGN

- It provides the theoretical foundation for lexical analysis
- Used to define the lexical structure of programming languages by breaking down the source code into tokens
- Finite automata can be constructed to recognize regular languages which represent the valid syntax patterns of tokens in programming languages grammar.
- Both NFA & DFA are used in lexical analysis to recognize patterns in the input source code.
- Techniques from automata theory (like minimizing DFA) are applied to optimize analyzers for better performance.

Q2] ERRORS HANDLED BY LEXICAL PHASE

- Invalid characters (dont conform to languages syntax rules)
- Illegal tokens
- Unexpected End Of File (premature termination)
- Ambiguous Lexemes (series of characters could be interpreted as multiple tokens)
- Overflow and underflow (numerical values)
- Mismatched Delimiters
- Reserved Words and Identifiers
- Comments (unterminated or improper comment delimiters)
- Whitespace Handling (unexpected whitespaces)

Department of Computer Engineering

Academic Term : Jan-May 23-24

Class: T.E. (Computer)
Subject Name : System Programming and Compiler Construction
Subject Code : (CPC601)

Practical No:	03
Title:	RECURSIVE DESCENT PARSER
Date of Performance:	14/02
Date of Submission:	28/02
Roll No:	9539
Name of the Student:	CRYSTAL FERNANDES

Evaluation:

Sr. No	Rubric	Grade
1	Timeline (2)	
2	Output(3)	
3	Code optimization (2)	
4	Postlab (3)	

Signature of the Teacher:

9539_SPCC_Experiment 3

NonTerminal Class Definition

In [27]:
```
class NonTerminal :
    def __init__(self, name) :
        self.name = name
        self.rules = []
    def addRule(self, rule) :
        self.rules.append(rule)
    def setRules(self, rules) :
        self.rules = rules
    def getName(self) :
        return self.name
    def getRules(self) :
        return self.rules
    def printRule(self) :
        print(self.name + " -> ", end = "")
        for i in range(len(self.rules)) :
            print(self.rules[i], end = "")
            if i != len(self.rules) - 1 :
                print(" | ", end = "")
        print()
```

Grammar Class Definition

In [28]:

```
class Grammar:
    def __init__(self):
        self.nonTerminals = []

    def addRule(self, rule):
        nt = False
        parse = ""

        for i in range(len(rule)):
            c = rule[i]
            if c == ' ':
                if not nt:
                    newNonTerminal = NonTerminal(parse)
                    self.nonTerminals.append(newNonTerminal)
                    nt = True
                    parse = ""
                elif parse != "":
                    self.nonTerminals[len(self.nonTerminals) - 1].addRule(
                    parse = ""
            elif c != '|' and c != '-' and c != '>':
                parse += c
        if parse != "":
            self.nonTerminals[len(self.nonTerminals) - 1].addRule(parse)

    def inputData(self):
        self.addRule("S -> Sa | Sb | c | d")

    def solveNonImmediateLR(self, A, B):
        nameA = A.getName()
        nameB = B.getName()

        rulesA = []
        rulesB = []
        newRulesA = []
        rulesA = A.getRules()
        rulesB = B.getRules()

        for rule in rulesA:
            if rule[0 : len(nameB)] == nameB:
                for rule1 in rulesB:
                    newRulesA.append(rule1 + rule[len(nameB) : ])
            else:
                newRulesA.append(rule)
        A.setRules(newRulesA)
```

In [29]:

```
    def solveImmediateLR(self, A):
        name = A.getName()
        newName = name + "'"

        alphas = []
        betas = []
        rules = A.getRules()
        newRulesA = []
        newRulesA1 = []

        rules = A.getRules()

        # Checks if there is left recursion or not
        for rule in rules:
            if rule[0 : len(name)] == name:
                alphas.append(rule[len(name) : ])
            else:
                betas.append(rule)

        # If no left recursion, exit
        if len(alphas) == 0:
            return

        if len(betas) == 0:
            newRulesA.append(newName)

        for beta in betas:
            newRulesA.append(beta + newName)

        for alpha in alphas:
            newRulesA1.append(alpha + newName)

        # Amends the original rule

        A.setRules(newRulesA)
        newRulesA1.append("\u03B5")

        # Adds new production rule
        newNonTerminal = NonTerminal(newName)
        newNonTerminal.setRules(newRulesA1)
        self.nonTerminals.append(newNonTerminal)

    def applyAlgorithm(self):
        size = len(self.nonTerminals)
        for i in range(size):
            for j in range(i):
                self.solveNonImmediateLR(self.nonTerminals[i], self.nonTer
            self.solveImmediateLR(self.nonTerminals[i])

    def printRules(self):
        for nonTerminal in self.nonTerminals:
            nonTerminal.printRule()
```

Result

In [30]:
```
1 grammar = Grammar()
2 grammar.inputData()
3 grammar.applyAlgorithm()
4 grammar.printRules()
```

```
S -> cS' | dS'
S' -> aS' | bS' | ε
```

9539 - EXPERIMENT 3 - SPCC

Q1] LEFT RECURSION : Occurs when a non-terminal can derive itself directly or indirectly through one or more production rules

$A \rightarrow A\alpha \mid \beta$ .... example of left recursion

$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$
} removal of left recursion

(a) Replace it with equivalent right-recursion productions

(b) Introduce new non-terminals to represent recursive part

Q2] LEFT FACTORING : Process to eliminate common prefixes among the alternatives of a production in a grammar

$A \rightarrow \alpha\beta1 \mid \alpha\beta2 \mid \cdots \mid \alpha\beta n$ where α is a common prefix

(a) Factor out the common prefix α into a new non terminal

(b) Create new productions for each suffix

$\therefore A \rightarrow \alpha A'$ and $A' \rightarrow \beta1 \mid \beta2 \mid \beta3$

Q3] DIFF B/W TOP-DOWN & BOTTOM UP

TOP-DOWN	BOTTOM-UP
- Starts from the start symbol and tries to construct the tree from roots to leaves	- Starts from input symbol and tries to construct from leaves to roots
- Recursive descent is a common form	- SLR, LR(0), LR(1), LALR
- Less powerful generally handling ambiguity	- More powerful & capable
- Errors reported as soon as encountered	- Errors may be reported after the entire input is read.

28/02/2024

Department of Computer Engineering

Academic Term : Jan-May 23-24

Class: T.E. (Computer)
Subject Name : System Programming and Compiler Construction
Subject Code : (CPC601)

Practical No:	04
Title:	INTERMEDIATE CODE
Date of Performance:	28/02
Date of Submission:	13/03
Roll No:	9539
Name of the Student:	CRYSTAL FERNANDES

Evaluation:

Sr. No	Rubric	Grade
1	Timeline (2)	02
2	Output(3)	03
3	Code optimization (2)	02
4	Postlab (3)	03
		IV

Signature of the Teacher: 6/03/2024

9539_Experiment 4_SPCC

In [118]:
```
exp=input("Enter an expression: ").split(" ")
```

```
Enter an expression: a = b + c + d + e + 50
```

In [119]:
```
stack_operands = []
stack_operators = []

for i in exp:
    if i.isalnum():
        stack_operands.append(i)
    else:
        stack_operators.append(i)

print(stack_operands)
print(stack_operators)
```

```
['a', 'b', 'c', 'd', 'e', '50']
['=', '+', '+', '+', '+']
```

In [120]:
```
if (len(stack_operands)==3):
    print("3 add code")
else:
    num_temp = len(stack_operands)-2
```

In [121]:
```
new_var=[]
j = 1
for i in range(num_temp+1):
    if stack_operators[-1] !='=':
        new_var.append(stack_operands.pop())
        new_var.append(stack_operators.pop())
        new_var.append(stack_operands.pop())
        temp1=''.join(new_var)
        stack_operands.append("temp"+str(j))
        print(("temp"+str(j)+"="+temp1))
        j+=1
        new_var=[]
    else:
        print(stack_operands[0]+'='+stack_operands.pop())
        j+=1
        break

temp1=50+e
temp2=temp1+d
temp3=temp2+c
temp4=temp3+b
a=temp4
```

In [ ]:

9539 - EXPERIMENT 4 - SPCC

Q1]

L1: if a < b goto L2

goto L3

L2: if c < d goto L4

x = y - z

goto L1

L4: x = y + z

goto L1

L3: end

Q2]

switch E

L1: goto L2

case V1: goto L3

L2: goto L4

case V2: goto L5

...

default goto Ln.

Department of Computer Engineering

Academic Term : Jan-May 23-24

Class: T.E. (Computer)
Subject Name : System Programming and Compiler Construction
Subject Code : (CPC601)

Practical No:	05
Title:	LEXICAL ANALYZER TOOL
Date of Performance:	28/02
Date of Submission:	13/03
Roll No:	9539
Name of the Student:	CRYSTAL FERNANDES

Evaluation:

Sr. No	Rubric	Grade
1	Timeline (2)	
2	Output(3)	
3	Code optimization (2)	
4	Postlab (3)	

Signature of the Teacher:

//lexer.l

```
%{
#include <stdio.h>
#include <stdlib.h>
int COMMENT = 0;
%}

identifier [a-zA-Z][a-zA-Z0-9]*

%%

#.* {printf("\n%s is a preprocessor directive",yytext);}
int | float | char | double | while | for | struct | typedef | do | if | break |
continue | void | switch | return | else | goto {printf("\n\t%s is a keyword",yytext);}


<INITIAL>{identifier}\( {if(!COMMENT)printf("\nFUNCTION \n\t%s",yytext);}
<INITIAL>\{ {if(!COMMENT)printf("\n BLOCK BEGINS");}
<INITIAL>\} {if(!COMMENT)printf("BLOCK ENDS ");}
<INITIAL>{identifier}(\[[0-9]*\])? {if(!COMMENT) printf("\n %s IDENTIFIER",yytext);}
<INITIAL>\".*\" {if(!COMMENT)printf("\n\t %s is a STRING",yytext);}
<INITIAL>[0-9]+ {if(!COMMENT) printf("\n %s is a NUMBER ",yytext);}

<INITIAL>\)(\:)? {if(!COMMENT) {printf("\n\t");ECHO;printf("\n");}}
<INITIAL>\( ECHO;
<INITIAL>= {if(!COMMENT)printf("\n\t %s is an ASSIGNMENT OPERATOR",yytext);}
<INITIAL>\<= | <INITIAL>\>= | <INITIAL>\< | <INITIAL>== | <INITIAL>\> {if(!COMMENT)
printf("\n\t%s is a RELATIONAL OPERATOR",yytext);}

%%

int main(int argc, char **argv) {
   FILE *file;
   file=fopen("var.c","r");
   if(!file) {
      printf("could not open the file");
      exit(0);
   }
   yyin=file;
   yylex();
   printf("\n");
   return(0);
}

int yywrap() {
   return(1);
}
```

var.c

```
#include<stdio.h>
void main()
{
        int a,b,c;
        a=1;
        b=2;
        c=a+b;
        printf("Sum:%d",c);
}
```

OUTPUT:

```
C:\Windows\System32\cmd.exe
C:\Users\HP\Desktop\9539_SPCC>flex lexer.l

C:\Users\HP\Desktop\9539_SPCC>gcc lex.yy.c

C:\Users\HP\Desktop\9539_SPCC>a.exe

#include<stdio.h>  is a preprocessor directive

 void IDENTIFIER
FUNCTION
        main(
        )


 BLOCK BEGINS

FUNCTION
        int
 a IDENTIFIER,
 b IDENTIFIER,
 c IDENTIFIER;

 a IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 1 is a NUMBER ;

 b IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 2 is a NUMBER ;

 c IDENTIFIER
         = is an ASSIGNMENT OPERATOR
 a IDENTIFIER+
 b IDENTIFIER;

FUNCTION
       printf(
        "Sum:%d" is a STRING,
 c IDENTIFIER
        )
;
BLOCK ENDS
```

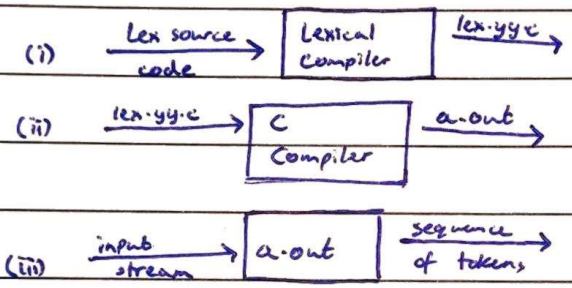
2937

9539 - Experiment 5 - SPCC

Q1] Lex is a tool that generates lexical analysis. The lex tool is in itself a compiler.

(i) It takes the source code which is in the lex language as input. After that an output file is generated of lex.yy.c

(ii) This file is used as input to the C compiler which gives output as a.out file.

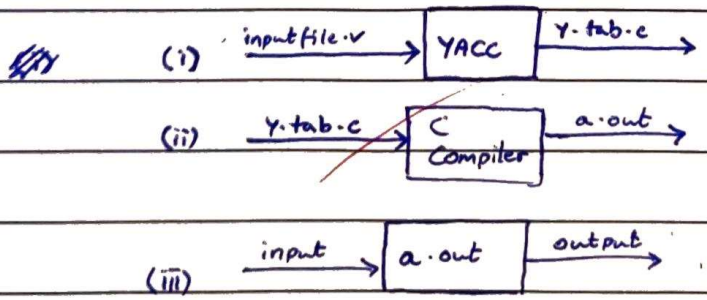


Q2] • Has the same three part structure as lex

• each part is seperated by a %% symbol

• The three parts are identical :

- definition section
- rules section
- code section



13/03/2024