Fr. Conceicao Rodrigues College of Engineering
Fr. Agnel Ashram, Bandstand, Bandra(W)
Mumbai – 400050
Department of Computer Engineering

2023 - 24

Subject: SPCC (Sem VI) CSC601

Student Name:Riya Jaison Kunnumkada **Roll No.:9553**

Sr. No.	Experiment Title	Submission Date	Remarks
1.	To write a program for implementing Symbol Table.		
2.	Write a program to implement Lexical analyzer.		
3.	Design recursive descent parser.		
4.	To generate an Intermediate code		
5.	Study of Lexical analyzer tool -Flex/Lex and Yacc		
6.	Generate a target code for the optimized code.		
7.	Write a program to implement Two Pass Assembler		
8.	Write a program to implement two pass Macro Processor		
9.	Assignment 1		
10.	Assignment 2		
11.	Assignment 3		

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:	Implementing a Symbol Table
Date of Performance:	02-02-2024
Date of Submission:	09-02-2024
Roll No:	9553
Name of the Student:	Riya Jaison Kunnumkada

Evaluation:

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

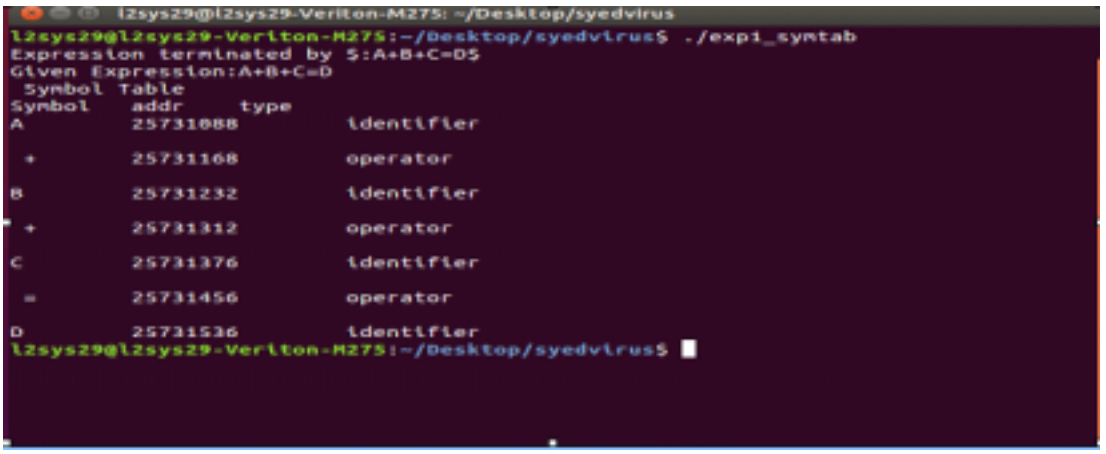
Signature of the Teacher:

AIM:
To write a program for implementing a Symbol Table.

ALGORITHM

Step1: Start the program for performing insert, display, delete, search and modify option in symbol table
Step2: Define the structure of the Symbol Table
Step3: Enter the choice for performing the operations in the symbol Table **Step4:** If the entered choice is 1, search the symbol table for the symbol to be inserted. If the symbol is
already present, it displays “Duplicate Symbol”. Else, insert the symbol
and the corresponding address in
the symbol table.
Step5: If the entered choice is 2, the symbols present in the symbol table are displayed. **Step6:** If the entered choice is 3, the symbol to be deleted is searched in the symbol table. **Step7:** If it is not found in the symbol table it displays “Label Not found”. Else, the symbol is deleted.
Step8: If the entered choice is 5, the symbol to be modified is searched in the symbol table.

Sample Input and Output:



Code for Symbol Table:
Code:

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SYMBOLS 100

struct Symbol
{
```

```
    char name;
    void *address;
    char type[20];
};

struct Symbol symbolTable[MAX_SYMBOLS];
int symbolCount = 0;

void insertSymbol(char name, void *address, const char *type)
{
    symbolTable[symbolCount].name = name;
    symbolTable[symbolCount].address = address;
    strcpy(symbolTable[symbolCount].type, type);
    symbolCount++;
}

void deleteSymbol(char name)
{
    for (int i = 0; i < symbolCount; i++)
    {
        if (symbolTable[i].name == name)
        {
            for (int j = i; j < symbolCount - 1; j++)
            {
                symbolTable[j] = symbolTable[j + 1];
            }
            symbolCount--;
            printf("Symbol '%c' deleted successfully.\n", name);
            return;
        }
    }
    printf("Symbol '%c' not found.\n", name);
}

void modifySymbol(char name, void *newAddress, const char *newType)
{
    for (int i = 0; i < symbolCount; i++)
    {
        if (symbolTable[i].name == name)
        {
            symbolTable[i].address = newAddress;
            strcpy(symbolTable[i].type, newType);
            printf("Symbol '%c' modified successfully.\n", name);
            return;
```

```
        }
    }
    printf("Symbol '%c' not found.\n", name);
}

void displaySymbolTable()
{
    printf("Symbol Table:\n");
    printf("Symbol\tAddress\tType\n");
    for (int i = 0; i < symbolCount; i++)
    {
        printf("%c\t%p\t%s\n", symbolTable[i].name, symbolTable[i].address,
symbolTable[i].type);
    }
}

int main()
{
    int i = 0, j = 0, x = 0, n;
    void *add[5];
    char ch, srch, b[15], c;
    printf("Expression terminated by $:");
    while ((c = getchar()) != '$')
    {
        b[i] = c;
        i++;
    }
    n = i - 1;
    printf("Given Expression:");
    i = 0;
    while (i <= n)
    {
        printf("%c", b[i]);
        i++;
    }
    printf("\n");
    printf("Symbol Table\n");
    printf("Symbol\tAddress\tType\n");
    while (j <= n)
    {
        c = b[j];
        if (isalpha(c))
        {
            add[x] = malloc(sizeof(char));
```

```
        *(char *)add[x] = c;
        insertSymbol(c, add[x], "Identifier");
        printf("%c\t%p\tIdentifier\n", c, add[x]);
        x++;
        j++;
    }
    else
    {
        ch = c;
        if (ch == '+' || ch == '-' || ch == '*' || ch == '=')
        {
            add[x] = malloc(sizeof(char));
            *(char *)add[x] = ch;
            insertSymbol(ch, add[x], "Operator");
            printf("%c\t%p\tOperator\n", ch, add[x]);
            x++;
            j++;
        }
    }
}

int choice;
char symbolToDelete, symbolToModify;
char newType[20];
void *newAddress;

do
{
    printf("\nOptions:\n");
    printf("1. Insert Symbol\n");
    printf("2. Delete Symbol\n");
    printf("3. Modify Symbol\n");
    printf("4. Display Symbol Table\n");
    printf("5. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice)
    {
    case 1:
        printf("Enter symbol to insert: ");
        scanf(" %c", &ch);
        add[x] = malloc(sizeof(char));
        *(char *)add[x] = ch;
```

```
            printf("Enter type of symbol (Identifier/Operator): ");
            scanf("%s", newType);
            insertSymbol(ch, add[x], newType);
            x++;
            break;
        case 2:
            printf("Enter symbol to delete: ");
            scanf(" %c", &symbolToDelete);
            deleteSymbol(symbolToDelete);
            break;
        case 3:
            printf("Enter symbol to modify: ");
            scanf(" %c", &symbolToModify);
            printf("Enter new address: ");
            scanf("%p", &newAddress);
            printf("Enter new type of symbol: ");
            scanf("%s", newType);
            modifySymbol(symbolToModify, newAddress, newType);
            break;
        case 4:
            displaySymbolTable();
            break;
        case 5:
            printf("Exiting...\n");
            break;
        default:
            printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 5);

    // Free allocated memory
    for (int i = 0; i < symbolCount; i++)
    {
        free(symbolTable[i].address);
    }

    return 0;
}
```

Output for Symbol Table:

```
Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 1
Enter symbol name: A
Enter address: 1234
Enter type (0 for variable, 1 for function): 0

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 2
Symbol Table:
Name    Address Type
A       1234    Variable

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 1
Enter symbol name: +
Enter address: 5678
Enter type (0 for variable, 1 for function): 1

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 1
Enter symbol name: B
Enter address: 9012
Enter type (0 for variable, 1 for function): 0

Symbol Table Operations:
```

```
Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 1
Enter symbol name: -
Enter address: 3456
Enter type (0 for variable, 1 for function): 1

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 1
Enter symbol name: C
Enter address: 7890
Enter type (0 for variable, 1 for function): 0

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 2
Symbol Table:
Name    Address Type
A       1234    Variable
+       5678    Function
B       9012    Variable
-       3456    Function
C       7890    Variable

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice:
```

```
Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 2
Symbol Table:
Name    Address Type
(       1234    Function
A       5678    Variable
+       4367    Function
B       2389    Variable
)       6589    Function
-       4332    Function
C       3562    Variable

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 3
Enter symbol name to delete: (
Symbol deleted successfully.

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 2
Symbol Table:
Name    Address Type
A       5678    Variable
+       4367    Function
B       2389    Variable
)       6589    Function
-       4332    Function
C       3562    Variable

Symbol Table Operations:
1. Insert
```

```
Symbol Table:
Name    Address Type
A       5678    Variable
+       4367    Function
B       2389    Variable
)       6589    Function
-       4332    Function
C       3562    Variable

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 4
Enter symbol name to modify: )
Enter new address: 5454
Symbol modified successfully.

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 2
Symbol Table:
Name    Address Type
A       5678    Variable
+       4367    Function
B       2389    Variable
)       5454    Function
-       4332    Function
C       3562    Variable

Symbol Table Operations:
1. Insert
2. Display
3. Delete
4. Modify
5. Exit
Enter your choice: 5
Exiting...
universe@lenovo14:~/Desktop/9553$
```

Postlab:

RIYA JASON KUNNUMKADA TE COMPS A 9553

SPCC - EXPERIMENT 01

(1) a = b + c - d * 5

a : Identifier (id1)

= : Assignment OPTR

b : Identifier (id2)

+ : Arithmetic OPTR

c : Identifier (id3)

- : Arithmetic OPTR

d : Identifier (id4)

* : Arithmetic OPTR

5 : Constant / Number

Phase 1: Lexical Analysis

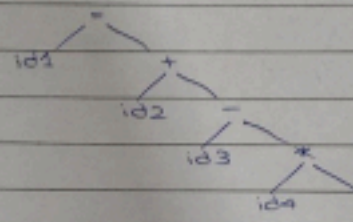
This phase takes as input the original source program and ~~tokens generated by phase 1 and generates a parse tree, if syntax is correct or generates errors~~ meaningful units called tokens.

[a] [=] [b] [+] [c] [-] [d] [*] [5]

Phase 2: Syntax Analysis:

This phase takes as an input the tokens generated by phase 1 and generates its parse tree if syntax is correct or generates error.

[a] [=] [b] [+] [c] [-] [d] [*] [5]
id1 id2 id3 id4 ~~id5~~



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:	Implementing Lexical Analysis
Date of Performance:	02-02-2024
Date of Submission:	09-02-2024
Roll No:	9553
Name of the Student:	Riya Jaison Kunnumkada

Evaluation:

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

Signature of the Teacher :

FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

System Programming and Compiler Construction

VI Semester (Computer) **Academic Year: 23-24**

Experiment No 2

Aim: Write a program to implement Lexical analyzer

Learning Objective: Converting a sequence of characters into a sequence of tokens.

Theory:

THE ROLE OF LEXICAL ANALYZER

The lexical analyzer is the first phase of a compiler. Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis. Upon receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

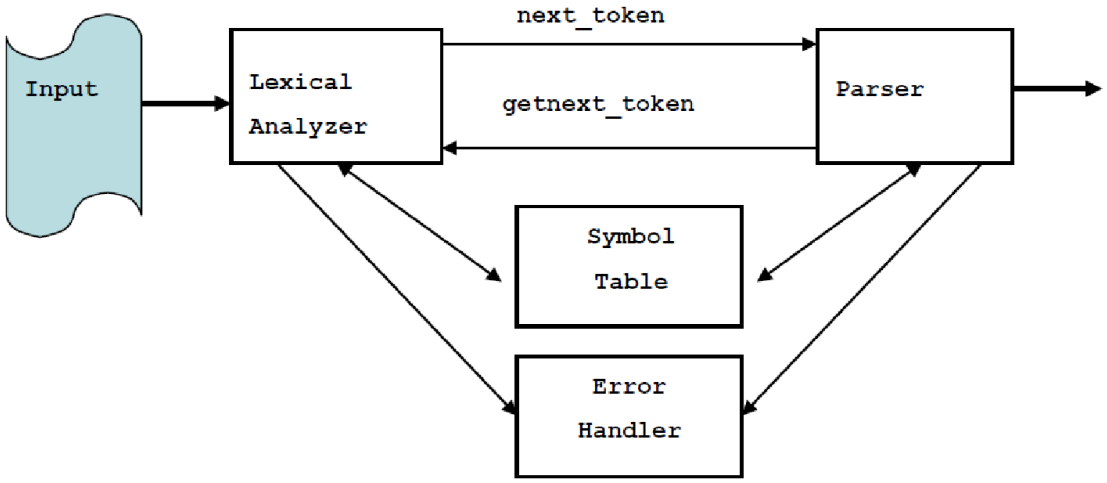


Figure 4.1 Interaction of Lexical Analyzer with Parser

Since the lexical analyzer is the part of the compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is stripping out from the source program comments and white spaces in the form of blank, tab, and new line characters. Another is correlating error messages from the compiler with the source program. Sometimes lexical analyzers are divided into a cascade of two phases first called "scanning" and the second "lexical analysis". The scanner is responsible for doing simple tasks, while the lexical analyzer properly does the more complex operations.

Implementation Details

1. Read the high level language as source program
2. Convert source program in to categories of tokens such as Identifiers, Keywords, Constants, Literals and Operators.

Test cases:

1. Input undefined token

Code:

```
#include <ctype.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LENGTH 100
bool isDelimiter(char chr)
{
	return (chr == ' ' || 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;
}
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 = (char*)malloc((subLength + 1) * sizeof(char));
```

```
        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]))
        printf("Token: Operator, Value: %c\n", input[right]);
        else if (input[right] != ' ') // Add this condition to skip space
        printf("Token: Delimiter, Value: %c\n", input[right]);

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

        if (isKeyword(subStr))
        printf("Token: Keyword, Value: %s\n", subStr);

        else if (isInteger(subStr))
        printf("Token: Integer, Value: %s\n", subStr);
```

```
        else if (isValidIdentifier(subStr) && !isDelimiter(input[right - 1]))
        printf("Token: Identifier, Value: %s\n", subStr);

        else if (!isValidIdentifier(subStr) && !isDelimiter(input[right - 1]))
        printf("Token: Unidentified, Value: %s\n", subStr);

        left = right;
        }
        }
        return 0;
}

int main()
{
        FILE *file = fopen("sample.txt", "r");
        if (file == NULL) {
        printf("Error opening file.\n");
        return 1;
        }

        char line[MAX_LENGTH];
        while (fgets(line, sizeof(line), file)) {
        // Remove the newline character if present
        size_t len = strlen(line);
        if (len > 0 && line[len - 1] == '\n') {
        line[len - 1] = '\0';
        }

        printf("For Expression \"%s\":\n", line);
        lexicalAnalyzer(line);
        printf("\n");
```

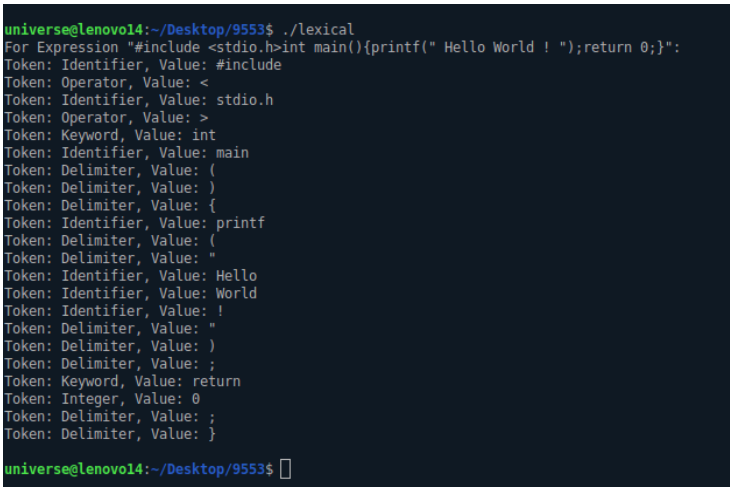
```
    }

    fclose(file);

    return 0;
}
```

sample.txt: #include <stdio.h>int main(){printf(" Hello World ! ");return 0;}

Output:



Conclusion: By properly identifying and treating delimiters, including special characters like double quotes, in lexical analysis, tokenization of programming expressions can be accurately achieved.

Post Lab Questions:

1. Explain the role of automata theory in compiler design.
2. What are the errors that are handled by the Lexical analysis phase?

SPCC EXP 02 POSTLAB RIYA JAISON TE COMPS A 9553

(1) Explain the role of automata theory in compiler design

Ans -Automata theory plays a crucial role in compiler design by providing formal models to describe the structure and behaviour of programming languages

(1) Lexical Analysis: Automata describe token patterns in source code using regular expressions, aiding in token recognition.

(2) Parsing: CFG transform syntax rules into parsing algorithms for recognizing program structure.

(3) Semantic Analysis: Automata ensure syntax adherence, indirectly influencing semantic analysis for consistency

(4) Code Generation: Automata-based phases ensure correct translation of source code into target code

(2) What are the errors handled by Lexical Analysis phase?

Ans. Lexical Errors: Incorrect token sequences or unrecognized characters

Identifier Errors: Invalid identifiers or keywords

Numeric Errors: Malformed numbers or constants

String Errors: Incorrectly terminated or malformed strings

Comment Errors: Improperly formatted or unterminated comments

Whitespace Errors: Improper handling of whitespace characters

Symbol Errors: Misuse or unrecognized symbols

Escape Sequence Errors: Incorrect usage of escape sequences in strings or characters

14/02/2024 10:47

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:	09-02-2024
Date of Submission:	16-02-2024
Roll No:	9553
Name of the Student:	Riya Jaison Kunnumkada

Evaluation:

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

Signature of the Teacher :

Experiment No 3

Aim : Design recursive descent parser.

Theory :

A **recursive descent parser** is a kind of top-down parser built from a set of mutually-recursive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

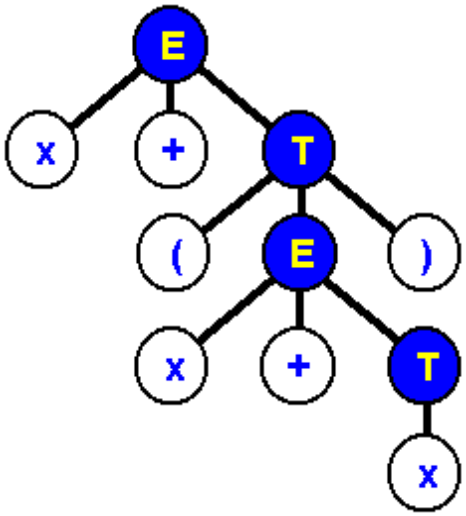
This parser attempts to verify that the syntax of the input stream is correct as it is read from left to right. A basic operation necessary for this involves reading characters from the input stream and matching then with terminals from the grammar that describes the syntax of the input. Our recursive descent parsers will look ahead one character and advance the input stream reading pointer when proper matches occur.

What a recursive descent parser actually does is to perform a depth-first search of the derivation tree for the string being parsed. This provides the 'descent' portion of the name. The 'recursive' portion comes from the parser's form, a collection of recursive procedures.

As our first example, consider the simple grammar

E ® x+T
T ® (E)
T ® x

and the derivation tree in figure 2 for the expression x+(x+x)



Derivation Tree for x+(x+x)

A recursive descent parser traverses the tree by first calling a procedure to recognize an E. This procedure reads an 'x' and a '+' and then calls a procedure to recognize a T. This would look like the following routine.

```
Procedure E()
Begin
If (input_symbol='x') then
next();
If (input_symbol='+') then
Next();
T();
Else
Errorhandler();
END
```

Procedure for E

Note that the 'next' looks ahead and always provides the next character that will be read from the input stream. This feature is essential if we wish our parsers to be able to predict what is due to arrive as input.

Note that 'errorhandler' is a procedure that notifies the user that a syntax error has been made and then possibly terminates execution.

In order to recognize a T, the parser must figure out which of the productions to execute. This is not difficult and is done in the procedure that appears below.

```
Procedure T()
Begin
Begin
If (input_symbol='(') then
next();
E();
If (input_symbol=')') then
next();
end
else If (input_symbol='x') then
next();
else
Errorhandler();
END
```

In the above routine, the parser determines whether T had the form (E) or x. If not then the error routine was called, otherwise the appropriate terminals and nonterminals were recognized.

Algorithm:

1. Make grammar suitable for parsing i.e. remove left recursion(if required).

2. Write a function for each production with an error handler.

3. Given input is said to be valid if input is scanned completely and no error function is called.

Code:

```
import java.util.Scanner;
public class spcc3{
	private static final int MAX_LENGTH = 100;
	private String input;
	private int currentIndex;

	public spcc3() {
	input = "";
	currentIndex = 0;
	}

	public void acceptInputString() {
	Scanner scanner = new Scanner(System.in);
	System.out.print("Enter input string: ");
	input = scanner.nextLine();
	scanner.close();
	}

	public void errorHandler() {
	System.out.println("Input String not valid!");
	System.exit(0);
	}

	public void parse() {
	procedureE();
	System.out.println("Input string parsed successfully!");
	}

	private void procedureE() {
	String nextSymbol = getNextSymbol();
	if (nextSymbol.equals("x")) {
	currentIndex += nextSymbol.length() + 1;
	nextSymbol = getNextSymbol();
	if (nextSymbol.equals("+")) {
	currentIndex += nextSymbol.length() + 1;
	procedureT();
	} else {
	errorHandler();
	}
	} else {
	errorHandler();
	}
	}
```

```
        private void procedureT() {
        String nextSymbol = getNextSymbol();
        if (nextSymbol.equals("(")) {
        currentIndex += nextSymbol.length() + 1;
        procedureE();
        nextSymbol = getNextSymbol();
        if (nextSymbol.equals(")")) {
        currentIndex += nextSymbol.length() + 1;
        } else {
        errorHandler();
        }
        } else if (nextSymbol.equals("x")) {
        currentIndex += nextSymbol.length() + 1;
        } else {
        errorHandler();
        }
        }

        private String getNextSymbol() {
        return input.substring(currentIndex).split("\\s+")[0];
        }

        public static void main(String[] args) {
        spcc3 parser = new spcc3();
        parser.acceptInputString();
        parser.parse();
        }
}
```

Output:

```
universe@lenovo15:~/Desktop/9553$ java spcc3
Enter input string: x + x
Input string parsed successfully!
```

Conclusion: We have successfully learned how to design and implement the Recursive Descent Parser.

PostLab:

1. What is left Recursion ? Write the rules for removing left recursion.
2. What is left factoring ? Write rules for eliminating left factoring.
3. Difference between top down and Bottom up parsing

SPCC EXP 03 # POSTLAB RIYA JAISON TE COMPS A 9553

(1) What is left recursion? Write rules for removing left recursion

Ans. - Occurs when a non-terminal A directly derives itself as the leftmost symbol in one or more of its productions
- Can cause issues in ~~passing~~ parsing algorithms as recursive descent parsers
- RULES
(a) Identify - When a non-terminal directly or indirectly produces itself as the leftmost symbol.
(b) For each left recursive non-terminal, create a new non-terminal and split productions in two sets:
(i) Non-recursive production
(ii) Recursive productions.
(c) Replace left-recursive productions of A with productions using the new terminal A, ensuring that the recursion is moved to the right side of the production
(d) Adjust the new non-terminal's productions to maintain original language defined by the grammar

(2) What is left factoring? Write rules for eliminating left factoring.

Ans. Left factoring is a grammar transformation technique used to eliminate common prefix in alternative productions of a non-terminal. It is used to make parsing more efficient and reduce ambiguity in the grammar
Rules for eliminating left factoring:
Given a production $A \rightarrow \alpha\beta_1 / \alpha\beta_2$, where α is a common prefix and β_1 and β_2 are distinct suffixes. This introduces a new non-terminal A' to represent the common prefix α, and the original alternatives are factored into seperate productions.

(3) Difference between top-down and bottom-up parsing.

TOP-DOWN	BOTTOM-UP
- Starts from the root of the parse tree and works downwards towards the leaves	- Starts from the input taken and works upwards towards the root of the parse tree
- Works with the start symbol & selecting production rule based on the current non-terminal and next input.	- Works beginning with input tokens and applying productions rules in reverse until reaching the start symbol.
- Better suited for LL(k) grammar which are typically simpler and more restricted	- Handles a wide range of grammars, including LL(k) grammar.
Eg: Recursive Descent Parsing, LL(k) parsing.	Eg: Shift-Reduce Parsing LR(k) parser.

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:	Generate an Intermediate Code
Date of Performance:	16-02-2024
Date of Submission:	23-02-2024
Roll No:	9553
Name of the Student:	Riya Jaison Kunnumkada

Evaluation:

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

Signature of the Teacher :

Experiment No 4

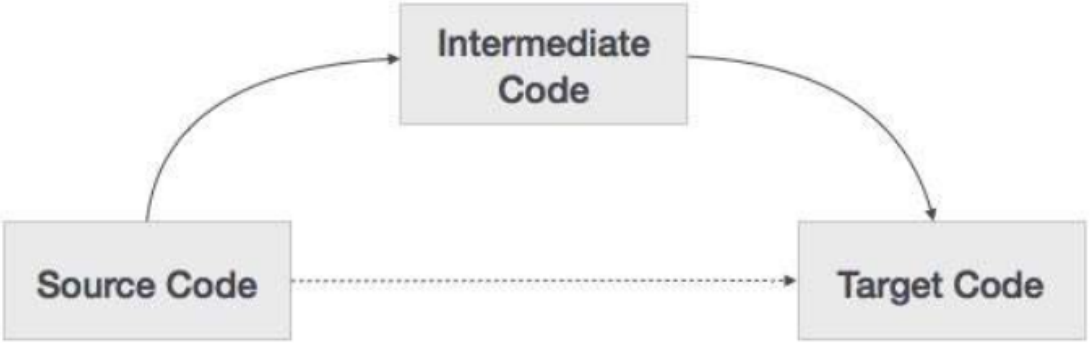
FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

System Programming and Compiler Construction

VI Semester (Computer) Academic Year: 23-24

Aim : To generate an Intermediate code.

Description:



- If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
- Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
- The second part of compiler, synthesis, is changed according to the target machine.
- It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

- # Three-Address Code

  Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.
- For example:

```
a = b + c * d;
```

- The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

- r being used as registers in the target program.
- A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg_1	arg_2
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Code:

```
import java.util.Scanner;
public class SPCC_Exp_4
{
  private static int tempCount = 1;
  public static void main (String[]args)
  {
            Scanner scanner = new Scanner (System.in);
            while (true)
              {
                  System.out.println("Generate three-address code for: 1)Assignment
2)Conditional 3)Indexed Assignment 4)Copy 5)Exit");
                  System.out.print ("Enter your choice: ");
                  int choice = scanner.nextInt ();
                  scanner.nextLine ();
                  System.out.println ();
                  switch (choice)
                    {
                    case 1:
                          generateAssignmentCode ();
                          break;
                          case 2:generateConditionalCode ();
                          break;
                          case 3:generateIndexedAssignmentCode ();
                          break;
                          case 4:generateCopyCode ();
                          break;
                          case 5:System.out.println ("Exiting...");
                          scanner.close ();
                          System.exit (0);
                          default:System.out.
                            println ("Invalid choice. Please enter a number from 1 to 5.");
                    }
                  System.out.println ();
  }
  private static void generateAssignmentCode ()
  {
            String expression = "a = b * c + d";
            System.out.println ("Expression: " + expression);
            String[]parts = expression.split (" = ");
            String lhs = parts[0].trim ();
            String rhs = parts[1].trim ();
            String[]terms = rhs.split (" ");
            String result = lhs;
            StringBuilder code = new StringBuilder ();
            for (int i = 0; i < terms.length; i++)
              {
                  if (terms[i].equals ("+") || terms[i].equals ("*"))
                    {
                          String op = terms[i];
```

```
                        String operand1 = terms[i - 1];
                        String operand2 = terms[i + 1];
                        String temp = "t" + tempCount++;
                        code.append (temp).append (" = ").append (operand1).append (" ").append (op).append (" ").append (operand2).append (";\n");
                        terms[i + 1] = temp;
                    }
                }
            code.append (result).append (" = ").append (terms[terms.length - 1]).append (";");
            System.out.println ("Three-address code:");
            System.out.println (code.toString ());
    }
    private static void generateConditionalCode ()
    {
            String expression = "if (a < b) then c = d + e else c = d - e";
            System.out.println ("Expression: " + expression);
            String[]parts = expression.split (" then ");
            String condition = parts[0].substring (3).trim ();
            String thenPart = parts[1].split (" else ")[0].trim ();
            String elsePart = parts[1].split (" else ")[1].trim ();
            String trueLabel = "L" + tempCount++;
// String falseLabel = "L" + tempCount++;
            String endLabel = "L" + tempCount++;
            StringBuilder code = new StringBuilder ();
            code.append ("if ").append (condition).append (" goto ").append (trueLabel).append (";\n");
            code.append (elsePart).append (" goto ").append (endLabel).append (";\n");
            code.append (trueLabel).append (": ").append (thenPart).append (" goto ").append (endLabel).append (";\n");
            code.append (endLabel).append (": ");
            System.out.println ("Three-address code:");
            System.out.println (code.toString ());
    }
    private static void generateCopyCode ()
    {
            String expression = "x = y";
            System.out.println ("Expression: " + expression);
            String[]parts = expression.split (" = ");
            String target = parts[0].trim ();
            String source = parts[1].trim ();
            StringBuilder code = new StringBuilder ();
            code.append (target).append (" = ").append (source).append (";");
            System.out.println ("Three-address code:");
            System.out.println (code.toString ());
    }
    private static void generateIndexedAssignmentCode ()
    {
            String expression = "x[2] = y + 5";
            System.out.println ("Expression: " + expression);
            String[]parts = expression.split (" = ");
```

```
            String target = parts[0].trim ();
            String operation = parts[1].trim ();
            int startIndex = target.indexOf ('[');
            int endIndex = target.indexOf (']');
            String arrayName = target.substring (0, startIndex).trim ();
            String index = target.substring (startIndex + 1, endIndex).trim ();
            String[]operationParts = operation.split ("\\s+");
            String operand1 = operationParts[0];
            String operator = operationParts[1];
            String operand2 = operationParts[2];
            String intermediateVar = "T" + tempCount++;
            StringBuilder code = new StringBuilder ();
            code.append (intermediateVar).append (" = ").append (operand1).append (" ").append (operator).append (" ").append (operand2).append (";\n");
            code.append (arrayName).append ("[").append (index).append ("] = ").append (intermediateVar).append (";");
            System.out.println ("Three-address code:");
            System.out.println (code.toString ());
    }
}
```

```
Generate three-address code for: 1)Assignment 2)Conditional 3)Indexed Assignment 4)Copy 5)Exit
Enter your choice: 1

Expression: a = b * c + d
Three-address code:
t1 = b * c;
t2 = t1 + d;
a = t2;

Generate three-address code for: 1)Assignment 2)Conditional 3)Indexed Assignment 4)Copy 5)Exit
Enter your choice: 2

Expression: if (a < b) then c = d + e else c = d - e
Three-address code:
if (a < b) goto L3;
c = d - e goto L4;
L3: c = d + e goto L4;
L4:

Generate three-address code for: 1)Assignment 2)Conditional 3)Indexed Assignment 4)Copy 5)Exit
Enter your choice: 3

Expression: x[2] = y + 5
Three-address code:
T5 = y + 5;
x[2] = T5;

Generate three-address code for: 1)Assignment 2)Conditional 3)Indexed Assignment 4)Copy 5)Exit
Enter your choice: 4

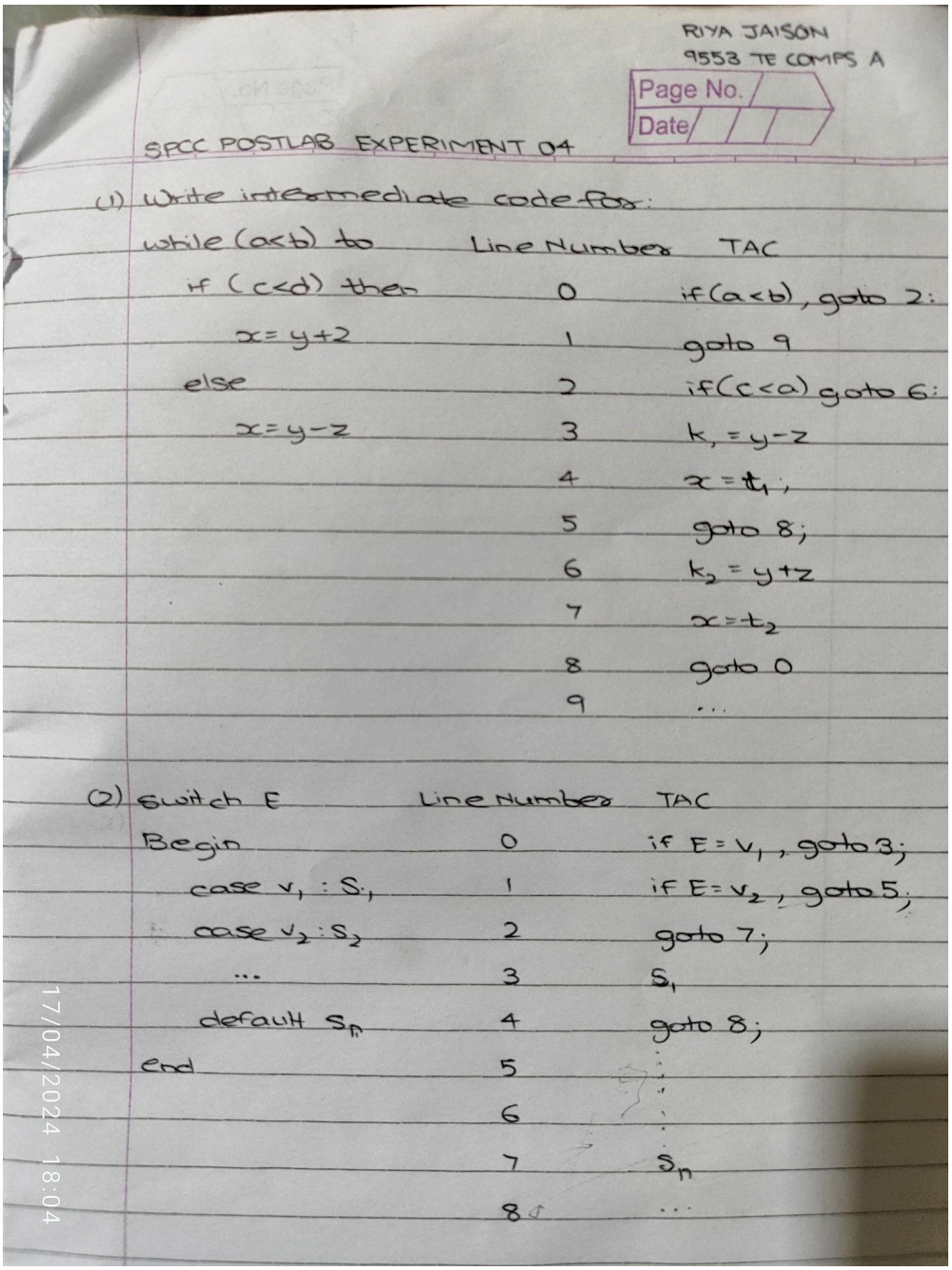
Expression: x = y
Three-address code:
x = y;

Generate three-address code for: 1)Assignment 2)Conditional 3)Indexed Assignment 4)Copy 5)Exit
Enter your choice: 5

Exiting...
PS C:\Users\DELL\Desktop\Lecture PDF\College Softwares\SPCC (Sem-VI)>
```

Postlab Question

1. Write the intermediate code generated for ---- while (a<b) do
   If (c< d) then
   X= y+z
   Else
   X= y-z
2. Write the intermediate code generated for ---- switch E
   Begin
   case V_1 : S_1
   case V_2 : S_2
   ….
   default: S_n
   end



RIYA JAISON
9553 TE COMPS A

Page No.
Date

SPCC POSTLAB EXPERIMENT 04

(1) Write intermediate code for:

```
while (a<b) do
  if (c<d) then
    x=y+z
  else
    x=y-z
```

Line Number	TAC
0	if(a<b), goto 2:
1	goto 9
2	if(c<a) goto 6:
3	$k_1 = y-z$
4	$x = t_1$;
5	goto 8;
6	$k_2 = y+z$
7	$x = t_2$
8	goto 0
9	...

(2) switch E

```
Begin
  case v1 : S1
  case v2 : S2
  ...
  default Sn
end
```

Line Number	TAC
0	if $E = v_1$, goto 3;
1	if $E = v_2$, goto 5;
2	goto 7;
3	S_1
4	goto 8;
5	:
6	:
7	S_n
8	...

17/04/2024 18:04

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:	Lex and Yacc
Date of Performance:	23-02-2024
Date of Submission:	01-03-2024
Roll No:	9553
Name of the Student:	Riya Jaison Kunnumkada

Evaluation:

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

Signature of the Teacher :

Experiment No 5

Aim : Study of Lexical analyzer tool -Flex/Lex

Learning Objective: Recognise lexical pattern from given input file.

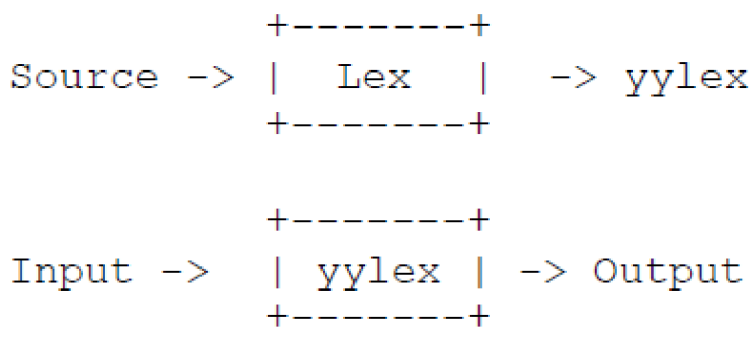
Theory:

Lex is a program generator designed for lexical processing of character input streams. Itaccepts a high-level, problem oriented specification for character string matching, andproduces a program in a general purpose language which recognizes regular expressions.The regular expressions are specified by the user in the source specifications given to Lex.The Lex written code recognizes these expressions in an input stream and partitions theinput stream into strings matching the expressions. At the boundaries between stringsprogram sections provided by the user are executed. The Lex source file associates theregular expressions and the program fragments. As each expression appears in the input tothe program written by Lex, the corresponding fragment is executed.The user supplies the additional code beyond expression matching needed to complete histasks, possibly including code written by other generators. The program that recognizesthe expressions is generated in the general purpose programming language employed for theuser's program fragments. Thus, a high level expression language is provided to write thestring expressions to be matched while the user's freedom to write actions isunimpaired. This avoids forcing the user who wishes to use a string manipulationlanguage for input analysis to write processing programs in the same and ofteninappropriate string handling language.

Lex is not a complete language, but rather a generator representing a new language featurewhich can be added to different programming languages, called ``host languages." Just asgeneral purpose languages can produce code to run on different computer hardware, Lexcan write code in different host languages. The host language is used for the output codegenerated by Lex and also for the program fragments added by the user. Compatible runtimelibraries for the different host languages are also provided. This makes Lex adaptableto different environments and different users. Lex itself exists on UNIX, GCOS, and OS/370; but the code generated by Lex may be taken anywhere where appropriate compilers exist.

Lex turns the user's expressions and actions (called source in this pic) into the hostgeneral-purpose language; the generated program is named yylex. The yylex program

willrecognize expressions in a stream (called input in this pic) and perform the specifiedactions for each expression as it is detected.



An overview of Lex

For a trivial example, consider a program to delete from the input all blanks or tabs at the

ends of lines.

%%

[\t]+$;

is all that is required. The program contains a %% delimiter to mark the beginning of therules, and one rule. This rule contains a regular expression which matches one or moreinstances of the characters blank or tab (written \t for visibility, in accordance with the Clanguage convention) just prior to the end of a line. The brackets indicate the characterclass made of blank and tab; the + indicates ``one or more ..."; and the $ indicates ``end ofline," as in QED. No action is specified, so the program generated by Lex (yylex) willignore these characters. Everything else will be copied. To change any remaining string ofblanks or tabs to a single blank, add another rule:

%%

[\t]+$;

[\t]+ printf(" ");

The finite automaton generated for this source will scan for both rules at once, observingat the termination of the string of blanks or tabs whether or not there is a newlinecharacter, and executing the desired rule action. The first rule matches all strings of blanksor tabs at the end of lines, and the second rule all remaining strings of blanks or tabs.

Lex can be used alone for simple transformations, or for analysis and statistics gatheringon a lexical level. Lex can also be used with a parser generator to perform the lexicalanalysis phase.

The general format of Lex source is:

{definitions}

%%

{rules}

%%

{user subroutines}

where the definitions and the user subroutines are often omitted. The second %% isoptional, but the first is required to mark the beginning of the rules. The absoluteminimum Lex program is thus

%%

(no definitions, no rules) which translates into a program which copies the input to theoutput unchanged.In the outline of Lex programs shown above, the rules represent the user's controldecisions; they are a table, in which the left column contains regular expressions and the rightcolumn contains actions, program fragments to be executed when the expressions arerecognized. Thus an individual rule might appearinteger printf("found keyword INT");to look for the string integer in the input stream and print the message ``found keyword INT"whenever it appears. In this example the host procedural language is C and the C libraryfunction printf is used to print the string. The end of the expression is indicated by the firstblank or tab character. If the action is merely a single C expression, it can just be given on theright side of the line; if it is compound, or takes more than a line, it should be enclosed inbraces.

Implementation Details:

1. Open file in text editor

2. Enter keywords, rules for identifier and constant, operators and relational operators. In the following format

    a) %{

    Definition of constant /header files

    %}

    b) Regular Expressions

    %%

    Transition rules

    %%

    c) Auxiliary Procedure (main( ) function)

3. Save file with **.l** extension e.g. **Mylex.l**

4. Call lex tool on the terminal e.g. [root@localhost]# lex Mylex.l This lex tool will convert ".l" file into ".c" language code file i.e. **lex.yy.c**

5. Compile the file lex.yy.c e.g. **gcc lex.yy.c** .After compiling the file lex.yy.c, this will create the output file **a.out**

6. Run the file a.out e.g. **./a.out**

7. Give input on the terminal to the **a.out** file upon processing output will be displayed

Sample Code

```
%{
#include<stdio.h>
int key_word=0;
%}
%%
"include"|"for"|"define" {key_word++;}
%%
int main()
{
printf("enter the sentence");
yylex();
printf("keyword are: %d\n ",key_word);
}
int yywrap() { return 1; }
```

Example: Program for counting number of vowels and consonant

```
%{

#include <stdio.h>

int vowels = 0;

int consonants = 0;

%}

%%


[aeiouAEIOU]  vowels++;

[a-zA-Z]  consonants++;

[\n]  ;

.  ;

%%
```

```
int main()

{

printf ("This Lex program counts the number of vowels and ");

printf ("consonants in given text.");

printf ("\nEnter the text and terminate it with CTRL-d.\n");

yylex();

printf ("Vowels = %d, consonants = %d.\n", vowels, consonants);

return 0;

}
```

Output:

```
#lex alphalex.l

#gcc lex.yy.c

#./a.out

This Lex program counts the number of vowels and consonants in given text.

Enter the text and terminate it with CTRL-d.

Iceream

Vowels =4, consonants =3.
```

Test Cases:

1. Input integer constant
2. Input special symbols

Conclusion:

Aim :Study of Parser generator tool –Yacc

Theory:

Parser for a grammar is a program which takes in the language string as its input and produces either a corresponding parse tree or a error.Syntax of a LanguageThe rules which tells whether a string is a valid program or not are called the syntaxSemantic s of Language The rules which give meaning to programs are called the semantic of a languageTokensWhen a string representing a program is broken into sequence of substrings, such that each substring represents a constant, identifier, operator, keyword etc of the language, these substrings are called the tokens of the language.

Lexical Analysis

The function of lexical Analyzer is to read the input stream representing the source program , one character at a time and translate into valid tokens.

Implementation Details
1: Create a lex file

The general format for lex file consists of three sections:

1. Definitions

2. Rules

3. User subroutine Section

Definitions consists of any external ‘C’ definitions used in the lex actions or subroutines. The other types of definitions are definitions are lex definitions which are essentially the lex substitution strings, lex start states and lex table size declarations. The rules is the basic part which specifies the regular expressions and their corresponding actions. The user Subroutines are the functions that are used in the Lex actions.
2 : Yacc is the Utility which generates the function ‘yyparse’ which is indeed the Parser. Yacc describes a context free, LALR(1) grammer and supports both bottom up and top-down parsing. The general format for the yacc file is very similar to that of the lex file.

1. Declarations

2. Grammar Rules

3. Subroutines

In declarations apart from the legal ‘C’ declarations here are few Yacc specific declarations which begins with a % sign.

1. %union It defines the Stack type for the Parser.

   It is union of various datas/structures/objects.

2. % token These are the terminals returned by the yylex function to the yacc. A token cal also have type associated with it for good type checking and syntax directed translation. A type of a token can be specified as % token <stack member> tokenName.

3. %type The type of non-terminal symbol in the grammar rule can be specified with this. The format is %type <stack member> non termainal.

4. %noassoc Specifies that there is no associativity of a terminal symbol.

5. %left Specifies the left associativity of aterminal symbol.

6. %rightSpecifies the right associativity of a terminal symbol.

7. %start specifies the L.H.S. non-terminal symbol of a production rule which specifies starting point of grammar rules.

8. %prac changes the precedence level associated with a particular rule to that of the following token name or literal.

The Grammar rules are specified as follows:

Context free grammar production-

p-> AbC

Yacc Rule-

P: A b C { /* 'C' actions*/}

The general style of coding the rules is to have all Terminals in lower –case and all non-terminals in upper –case.

To facilitate a proper syntax directed translation the Yacc has something calls pseudo-variables which forms a bridge between the values of terminals/non-terminals and the actions. These pseudo variables are $$, $1, $2, $3,……The $$ is the L.H.S value of the rule whereas $1 is the first R. H. S value of the rule, so is the $2 etc. The default type for pseudo variables is integer unless they are specified by % type.

%token <type> etc.

Perform the following steps, in order, to create the desk calculator example program:

1. Process the **yacc** grammar file using the **-d** optional flag (which tells the **yacc** command to create a file that defines the tokens used in addition to the C language source code):

   ```
   yacc -d calc.yacc
   ```

2. Use the **li** command to verify that the following files were created:
   y.tab.c The C language source file that the **yacc** command created for the parser.
   y.tab.h A header file containing define statements for the tokens used by the parser.
3. Process the **lex** specification file:
   ```
   lex calc.lex
   ```

4. Use the **li** command to verify that the following file was created:

   lex.yy.c The C language source file that the **lex** command created for the lexical analyzer.

5. Compile and link the two C language source files:

   ```
   cc y.tab.c lex.yy.c
   ```

6. Use the **li** command to verify that the following files were created:
   y.tab.o The object file for the **y.tab.c** source file
   lex.yy.o The object file for the **lex.yy.c** source file
   a.out The executable program file

7. To then run the program directly from the **a.out** file, enter:
8. ```
   $ a.out
   ```
   ```

1. **Find the number of vowels and consonants in a sentence**

```
%{
int vow_count=0;
int const_count =0;
%}
%%
[aeiouAEIOU] {vow_count++;}
[a-zA-Z] {const_count++;}
%%
int yywrap(){}
int main()
{
printf("Enter the string of vowels and consonants:");
yylex();
printf("Number of vowels are: %d\n", vow_count);
printf("Number of consonants are: %d\n", const_count);
return 0;
}
```

```
universe@lenovo4:~/9536$ lex vowel.l
universe@lenovo4:~/9536$ cc lex.yy.c -lfl
universe@lenovo4:~/9536$ ./a.out
Enter the string of vowels and consonants:hello from other side

Number of vowels are:  7
Number of consonants are:  11
```

2. **Operands Operation:**

```
int opr=0, opd=0, n=0;
```

```
%}
%%
[\+\-\*\/=] { printf("OPERATOR: %s\n", yytext); opr++; }
[a-zA-Z]+ { printf("OPERAND: %s\n", yytext); opd++; }
[0-9]+ { printf("NUMBER: %s\n", yytext); opd++; }
[a-zA-Z]+[\+\-\*\/][a-zA-Z]+ { n=0; }
[0-9]+[\+\-\*\/][0-9]+ { n=0; }
%%
int yywrap() {
return 1;
}
int main()
{
printf("Enter the expression: ");
yylex();
printf("\nNumber of operators: %d", opr);
printf("\nNumber of operands: %d", opd);
if (n == 0 && opd == opr + 1)
printf("\nValid expression\n");
else
printf("\nInvalid expression\n");
return 0;
}
```

```
Enter the expression:  a=b+4
 OPERAND: a
OPERATOR: =
OPERAND: b
OPERATOR: +
NUMBER: 4


Number of operators: 2
Number of operands: 3
Valid expression
```

3. **Find positive and negative integers and positive and negative fractions in the given text file**

```
%{
int postiveno=0;
int negtiveno=0;
int positivefractions=0;
int negativefractions=0;
%}
DIGIT [0-9]
%%
\+?{DIGIT}+ postiveno++;
-{DIGIT}+ negtiveno++;
\+?{DIGIT}*\.{DIGIT}+ positivefractions++;
-{DIGIT}*\.{DIGIT}+ negativefractions++;
. ;
%%
void main()
{
yylex();
printf("\nNo. of positive numbers: %d",postiveno);
```

printf("\nNo. of Negative numbers: %d",negtiveno);

printf("\nNo. of Positive fractions: %d",positivefractions);

printf("\nNo. of Negative fractions: %d\n",negativefractions);

}

input file - a.txt

a.txt - (+12,-123,1.1,-1.1,12,-2,-3,2.1,3.2,5.1,-5.5,-6.1,-7.7,-8.8)

```
No. of positive numbers: 2
No. of Negative numbers: 3
No. of Positive fractions: 4
No. of Negative fractions: 5
```

4. **Find the number of lines, words, small letters, capital letters, special characters, digits and total characters**

%{

#include<stdio.h>

int lines=0, words=0,s_letters=0,c_letters=0, num=0, spl_char=0,total=0;

%}

%%

\n { lines++; words++;}

[\t ' '] words++;

[A-Z] c_letters++;

[a-z] s_letters++;

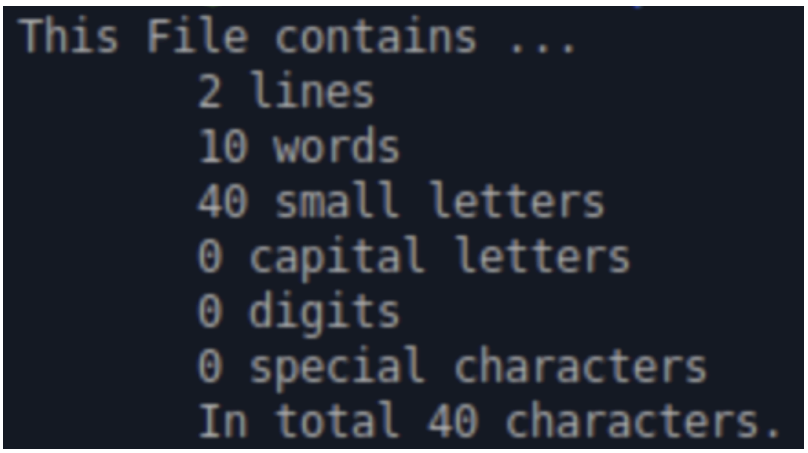
[0-9] num++;

. spl_char++;

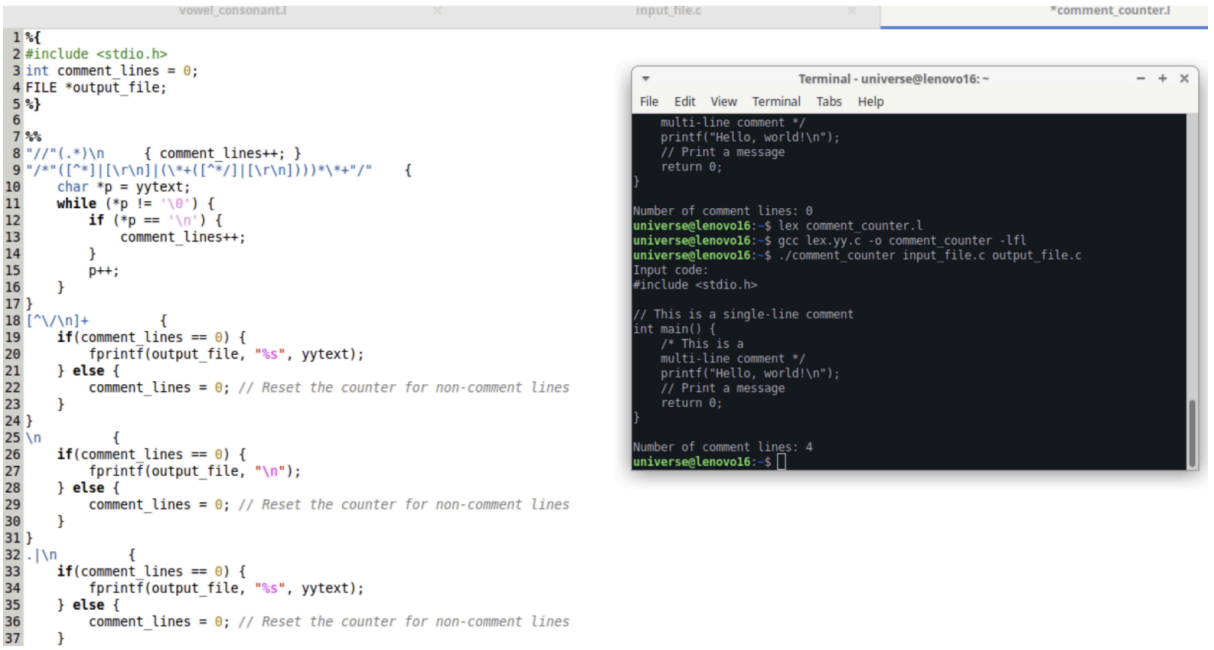
%%

main(void)

```
{
yyin= fopen("test.txt","r");
yylex();
total=s_letters+c_letters+num+spl_char;
printf(" This File contains ...");
printf("\n\t%d lines", lines);
printf("\n\t%d words",words);
printf("\n\t%d small letters", s_letters);
printf("\n\t%d capital letters",c_letters);
printf("\n\t%d digits", num);
printf("\n\t%d special characters",spl_char);
printf("\n\tIn total %d characters.\n",total);
}
int yywrap()
{
return(1);
}
```



5. **Find number of comment lines and eliminate the comment lines**



```
%{
#include <stdio.h>
int comment_lines = 0;
FILE *output_file;
%}

%%
"//"(.*)\n     { comment_lines++; }
"/*"([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+"/"    {
    char *p = yytext;
    while (*p != '\0') {
        if (*p == '\n') {
            comment_lines++;
        }
        p++;
    }
}
[^\/\n]+         {
    if(comment_lines == 0) {
        fprintf(output_file, "%s", yytext);
    } else {
        comment_lines = 0; // Reset the counter for non-comment lines
    }
}
\n          {
    if(comment_lines == 0) {
        fprintf(output_file, "\n");
    } else {
        comment_lines = 0; // Reset the counter for non-comment lines
    }
}
.|\n         {
    if(comment_lines == 0) {
        fprintf(output_file, "%s", yytext);
    } else {
        comment_lines = 0; // Reset the counter for non-comment lines
    }
```

```
Terminal - universe@lenovo16: ~
File  Edit  View  Terminal  Tabs  Help
    multi-line comment */
    printf("Hello, world!\n");
    // Print a message
    return 0;
}

Number of comment lines: 0
universe@lenovo16:~$ lex comment_counter.l
universe@lenovo16:~$ gcc lex.yy.c -o comment_counter -lfl
universe@lenovo16:~$ ./comment_counter input_file.c output_file.c
Input code:
#include <stdio.h>

// This is a single-line comment
int main() {
    /* This is a
    multi-line comment */
    printf("Hello, world!\n");
    // Print a message
    return 0;
}

Number of comment lines: 4
universe@lenovo16:~$
```

Conclusion: Thus we have successfully studied the Lexical Analyzer tool - Lex and the Parser generator tool - Yacc

Postlab:

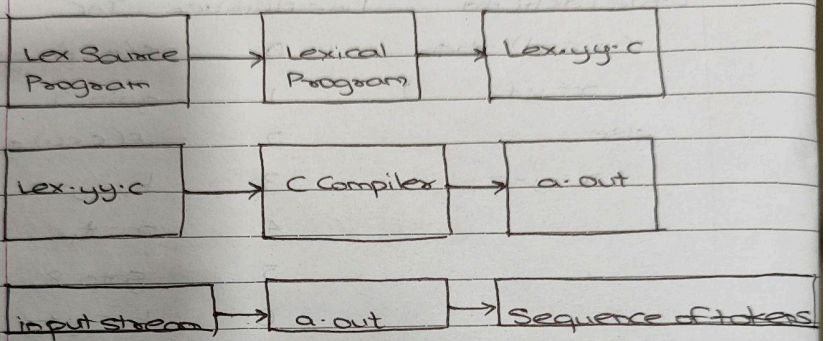
1. **Write the structure of Lex**
2. **Write the structure of Yacc**

RIYA JAISON
TE COMPS A 9553

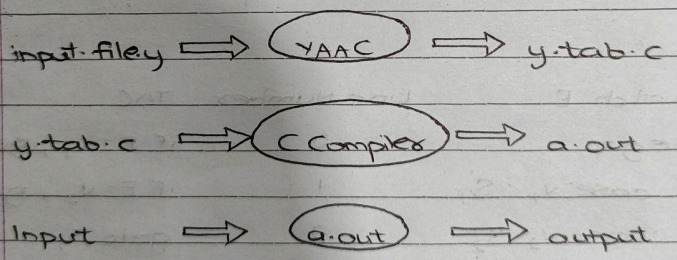
Page No.
Date

SPCC EXPERIMENT 05 POSTLAB

(1) Write structure of Lex



(2) Write the structure of Yacc.



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:	06
Title:	Target Code Generation
Date of Performance:	01-03-2024
Date of Submission:	15-03-2024
Roll No:	9553
Name of the Student:	Riya Jaison Kunnumkada

Evaluation:

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

Signature of the Teacher :

Experiment No 6

Aim : Generate a target code for the optimized code.

Algorithm:

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.

The code generation algorithm takes as input a sequence of three address statements constituting a basic block. For each three address statement of the form x=y op z we perform following function:

1. Invoke a function getreg to determine the location L where the result of computation y op z should be stored. (L cab be a register or memory location .

2. Consult the address descriptor for y to determine y, the current locations of y. Prefer the register for y if the value of y is currently both in memory and register. If value of y is not already in L , generate the instruction MOV y, L to place a copy of y in L.

3. Generate instruction po z, L where z is a current location of z. Again address descriptor of x to indicate that x is in location L. if L is a register, update its descriptor to indicate that it contains the value of x, and remove x from all other register descriptor.

4. If the current values of y and z have no next uses , are not live on exit from the block , and are in registers alter the register descriptor to indicate that , after execution of x=y op z, those register no longer will contain y and z, resply.

The function getreg:

The function getreg returns the location L to hold the values of x for the assignment x= y op z.

1.If the name y is in a reg that holds the value of no other names, and y is not live and has no next use after execution of x= y op z ,then return the register of y for L. Update the address descriptor of y to indicate that y is no longer in L.

2. Failing (1), return an empty register for L if there one.

3. Failing (2) , if X has a next use in the block, or op is an operator , such as indexing, that requires a register find an occupied register R. Store the values of R into a memory location (MOV R ,M) if it is not already in the proper memory location M, update the address descriptor for M , and return R. if R holds the value of several variables, a MOV instruction must be generated for each variable that needs to be stored. A suitable register might be one whose data is referenced furthest in the future, or one whose value is also in memory. We leave the exact choice unspecified, since there is no one proven best way to make the selection.

4. If x is not used in the block , or no suitable occupied register can found, select the memory location of x as L.

Code:

```
# Input must be in 3 address code form
import re   # Regex for splitting

registers = {}  # Dictionary to keep track of register assignment
output_code = []  # Output lines


def allocateRegister(operand):
    if operand in registers.values():  # Operand already present in one of the registers, use it
        for key, value in registers.items():
            if operand == value:
                return key
    else:
        register_name = "R" + str(len(registers))  # Allocate a new register for the operand
        registers[register_name] = operand  # Assign operand to register
          output_code.append("MOV "+operand+","+register_name)  # Add MOV statement to
output
        return register_name

def renameReg(location,operand):
    if operand in registers.values():
        for key, value in registers.items():
            if operand==value:
                registers[key]=location
                #print(registers)


input_code = list(line.strip() for line in open("code_gen_input.txt"))
for index,line in enumerate(input_code):
    line = re.split("([\=\+\-\*\/])", line)  # Split the TAG into operands and operator
    LHS, eq, op1, operator, op2 = line
    oper=op1
    if op2 in registers.values():
        op2=allocateRegister(op2)
    op1 = allocateRegister(op1)
    if operator.strip() == "+":
```

```
        output_line = "ADD "+str(op2)+","+str(op1)
        #print(oper,LHS)
        renameReg(LHS,oper)   # Register now hold output i.e. LHS
        #print(registers)
    elif operator.strip() == "-":
        output_line = "SUB " + str(op2) + "," + str(op1)
        renameReg(LHS,oper)
        #registers[op2] = LHS
    if(index==len(input_code)-1):
        output_line=output_line+("\nMOV "+op1+","+LHS)

    output_code.append(output_line)
for line in output_code:
    print(line)
```

Output:

```
c:\Users\riyaj\Downloads\SEMESTER 6\System Programming and Compiler Construction\exp6.py:29: SyntaxWarning: invalid escape sequence '\='
  line = re.split("([\=\+\-\*\/])", line)  # Split the TAG into operands and operator
MOV a,R0
SUB b,R0
MOV a,R1
SUB c,R1
MOV t ,R2
ADD R1,R2
ADD R1,R2
MOV R2,d
```

Conclusion: Thus, we have generated target code for the optimized code

Postlab:

Explain design issues of code generator phase?
What are basic blocks? State their properties

RIYA JAISON
TE COMPS A 9553

Page No.
Date

SPCC EXPERIMENT 06 POSTLAB

(1) Explain design issues of code generation phase
- Target architecture abstraction designing the code generator to abstract away target architecture intricacies with efficiently utilizing its features is critical
- Optimization - Balancing code size & execution speed while applying various techniques is challenging
- Target language representation - Efficiently mapping high-level language constructs to machine code instructions and data structures is essential.
- Handling control flow - generating code that accurately reflects control flow while optimizing for performance and minimizing overhead is vital.
- Error handling - Dealing with errors during code generation such as unsupported language features of invalid optimization requires careful consideration & meaning error messages

(2) What are the basic blocks? State their properties
- They are fundamental units of code used in various compiler optimizations and analysis.
(i) Single Entry, Single Exit - Simplifies control flow analysis.
(ii) No internal branching - Enables treating basic blocks as linear sequences of instructions
(iii) Atomic execution - Ensures sequential execution without interruption

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:	07
Title:	One Pass and Two Pass Assembler
Date of Performance:	15-03-2024
Date of Submission:	22-03-2024
Roll No:	9553
Name of the Student:	Riya Jaison Kunnumkada

Evaluation:

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

Signature of the Teacher :

FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

System Programming and Compiler Construction

VI Semester (Computer) **Academic Year: 23-24**

Experiment No 7

Aim: Write a program to implement Two Pass Assembler.

Learning Objective: Translating mnemonic operation codes to their machine language equivalents. Assigning machine addresses to symbolic labels used by the programmer. Lastly to convert assembly language to binary.

Algorithm:

Pass 1:

1. Start

2.Intialize location counter to zero

3.Read opcode field of next instruction.

4.search opcode in pseudo opcode Table(POT)

5.If opcode is found in POT

5.1 If it is ‘DS’ or ‘DC’

Adjust location counter to proper alignment.

Assign length of data field to ‘L’

Go to step 9

5.2 If it is ‘EQU’

Evaluate operand field

Assign values to symbol in label field

Go to step 3

5.3 If it is ‘USING’ or ‘DROP’ Go to step 3

5.4 If it is ‘END’

Assign value to symbol in label field

Go to step 3

6. Search opcode in Machine Opcode Table.

7. Assign its length to ‘L’.

8. Process any literals and enter them into literal Table.

9.If symbol is there in the label field

Assign current value of Location Counter to symbol

10. Location Counter= Location Counter +L.

11.Go to step 3.

12. Stop.

Pass2:

1.Start

2.Intialize location counter to zero.

3. Read opcode field of next instruction.

4.Search opcode in pseudo opcode table.

5.If opcode is found in pseudo opcode Table

5.1 If it is ‘DS’ or ‘DC’

Adjust location counter to proper alignment.

If it is ‘DC’ opcode form constant and insert in assembled program

Assign length of data field to ‘L’

Go to step 6.4

5.2 If it is ‘EQU’ or ‘START’ ignore it. Go to step 3

5.3 If it is ‘USING’

Evaluate operand and enter base reg no. and value into base table

Go to step 3

5.4 If it is ‘DROP’

Indicate base reg no . available in base table . Go to step 3

5.5 If it is ‘END’

Generate literals for entries in Literal Table

Go to step 12

6 Search opcode in MOT

7. Get opcode byte and format code

8. Assign its length to ‘L’.

9. Check type of instruction.

10.If it is type ‘RR’ type

10.1 Evaluate both register expressions and insert into second byte.

10.2 Assemble instruction

10.3 Location Counter= Location Counter +L.

10.4.Go to step 3.

11. If it is ‘RX’ type

11.1 Evaluate register and index expressions and insert into second byte.

11.2 Calculate effective address of operand.

11.3 Determine appropriate displacement and base register

11.4 Put base and displacement into bytes 3 and 4

11.5 Location Counter= Location Counter +L.

11.6 Go to step 11.2

13 Stop.

Implementation Details

1. Read Assembly language input file.
2. Display output of Pass1 as the output file with Op-code Table, Symbol Table.
3. Display output of pass2 as the Op-code Table, Symbol Table , Copy file.

Test Cases:

1 Input symbol which is not defined

2 Input Opcode which is not entered in MOT

Code:

PASS I:

```
3   #include <stdio.h>
4   #include <stdlib.h>
5   #include <string.h>
6
7   int main(){
8       char label[10],opcode[10],operand[10],mnemonic[10];
9       int locctr = 0;
10
11      FILE *fp1,*fp2,*fp3,*fp4;
12
13      fp1 = fopen("input.txt","r");   // input
14      fp2 = fopen("mot.txt","r");     // input
15      fp3 = fopen("output.txt","w");  // output
16      fp4 = fopen("st.txt","w");      // output
17
18      fscanf(fp1,"%s %s %s",label,opcode,operand); //opcode start
19
20      fprintf(fp3,"%d %s %s %s\n",locctr,label,opcode,operand);
21
22      fscanf(fp1,"%s %s %s",label,opcode,operand);//opcode using
23
24      fprintf(fp3,"%d %s %s %s\n",locctr,label,opcode,operand);
25
```

```
26      while(strcmp(opcode,"END")!=0){
27
28          if(strcmp(opcode,"DC")==0 || strcmp(opcode,"DS")==0){
29              fprintf(fp4,"%s %d\n",label,locctr);
30              fprintf(fp3,"%d %s %s %s\n",locctr,label,opcode,operand);
31              locctr += 4;
32          }
33          else{
34              fscanf(fp2,"%s",mnemonic);
35              fprintf(fp3,"%d %s %s %s\n",locctr,label,opcode,operand);
36              while(strcmp(mnemonic,"end")!=0){
37                  if(strcmp(opcode,mnemonic)==0){
38                      locctr += 4;
39                      break;
40                  }
41                  fscanf(fp2,"%s",mnemonic);
42              }
43              rewind(fp2);
44          }
45
46          fscanf(fp1,"%s %s %s",label,opcode,operand);
47      }
48
49      fprintf(fp3,"%d %s %s %s\n",locctr,label,opcode,operand);
50      fclose(fp1);
51      fclose(fp2);
52      fclose(fp3);
53      fclose(fp4);
54
55      return 0;
56  }
```

Input:

Spcc > exp7 > mot.txt

```
1    A 5A
2    L 6A
3    ST 7A
4    end
```

Spcc > exp7 > input.txt

```
1    PG1 START 0
2    ** USING *,15
3    ** L 1,FIVE
4    ** A 1,FOUR
5    ** ST 1,TEMP
6    FIVE DC F'5'
7    FOUR DC F'4'
8    TEMP DS 1F
9    ** END
```

Output:

Spcc > exp7 > output.txt

```
0 PG1 START 0
0 ** USING *,15
0 ** USING *,15
0 ** L 1,FIVE
4 ** A 1,FOUR
8 ** ST 1,TEMP
12 FIVE DC F'5'
16 FOUR DC F'4'
20 TEMP DS 1F
24 ** END 1F
```

Spcc > exp7 > st.txt

```
1   FIVE 12
2   FOUR 16
3   TEMP 20
4
```

Code:

PASS II:

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

int main(){
    char label[10],opcode[10],operand[10],mnemonic[10],locctr[10],mlabel[10];

    FILE *fp1,*fp2,*fp3,*fp4,*fp5;

    fp1 = fopen("input.txt","r");       //input
    fp2 = fopen("mot.txt","r");         //input
    fp3 = fopen("output.txt","r");      //input
    fp4 = fopen("outTable.txt","w");    //output
    fp5 = fopen("BT.txt","w");          //output

    fscanf(fp3,"%s %s %s %s",locctr,label,opcode,operand); //START line 1 ignore
    fscanf(fp3,"%s %s %s %s",locctr,label,opcode,operand); //USING Line 2

    fprintf(fp5,"Y  %c%c    00 00 00",operand[2],operand[3]); //Base table

    fscanf(fp3,"%s %s %s %s",locctr,label,opcode,operand);

```

```c
    fscanf(fp3,"%s %s %s %s",locctr,label,opcode,operand);

    while(strcmp(opcode,"END")!=0){

        if(strcmp(opcode,"DC")==0){
            fprintf(fp4,"%s\t%c\n",locctr,operand[2]);
        }
        else if(strcmp(opcode,"DS")==0){
            fprintf(fp4,"%s\t_\n",locctr);
        }
        else{
            fscanf(fp2,"%s %s",mnemonic,mlabel);
            while(strcmp(mnemonic,"end")!=0){
                if(strcmp(opcode,mnemonic)==0){
                    fprintf(fp4,"%s\t%s\n",mlabel,operand);
                    break;
                }
                fscanf(fp2,"%s %s",mnemonic,mlabel);
            }
            rewind(fp2);
        }

        fscanf(fp3,"%s %s %s %s",locctr,label,opcode,operand);

    }
    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
    fclose(fp4);
    fclose(fp5);

    return 0;
}
```

Input:

Spcc > pass2 > input.txt

```
PG1 START 0
** USING *,15
** L 1,FIVE
** A 1,FOUR
** ST 1,TEMP
FIVE DC F'5'
FOUR DC F'4'
TEMP DS 1F
** END
```

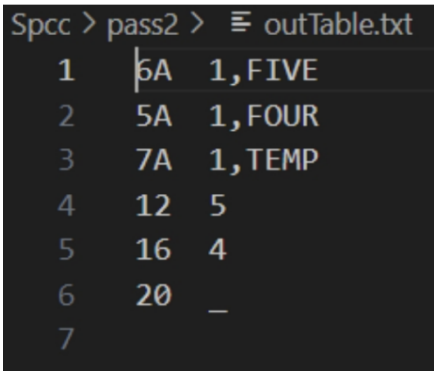
Spcc > pass2 > mot.txt

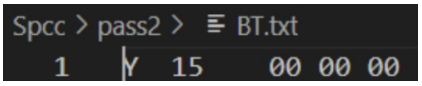
```
A 5A
L 6A
ST 7A
end
```

Spcc > pass2 > output.txt

```
0 PG1 START 0
0 ** USING *,15
0 ** USING *,15
0 ** L 1,FIVE
4 ** A 1,FOUR
8 ** ST 1,TEMP
12 FIVE DC F'5'
16 FOUR DC F'4'
20 TEMP DS 1F
24 ** END 1F
```

Output:





Conclusion: Thus we have implemented a Two Pass Assembler.

Post Lab Questions:

1. **Define the basic functions of assembler.**
2. **What is the need of SYMTAB (symbol table) in assembler?**
3. **What is the need of MOT in assembler**
4. **What is meant by one pass assembler?**

RIYA JAISON
TE COMPS A 9553

Page No.
Date

SPCC EXPERIMENT 07 POSTLAB

(1) Basic functions of assembler:
-Assemblers convert ALP into machine code:
(i) Symbolic representation of machine inst & data
(ii) Addressing modes translation
(iii) Label & symbol resolution

(2) SYMTAB (Symbol Table) is needed for:
(i) Tracking symbols & associated memory addresses as values.
(ii) Resolving references to symbols during assembly
(iii) Ensuring correct generation of machine code with proper symbol substitution.

(3) MOT (opcod table) is necessary for:
(i) Mapping mnemonic instructions to their corresponding machine language opcodes.
(ii) Providing information about the format and characteristics of instructions.
(iii) Assisting in the generation of machine code by the assembler.

(4) One-pass assembler:
- Processes assembly language program in a single pass as iteration through source code.
- Generates machine code directly without needing to revisit previously scanned instructions.
- Typically used for simpler assembly languages when memory constraints are tight

17/04/2024 19:42

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:	08
Title:	One Pass and Two Pass Macroprocessor
Date of Performance:	22-03-2024
Date of Submission:	12-04-2024
Roll No:	9553
Name of the Student:	Riya Jaison Kunnumkada

Evaluation:

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

Signature of the Teacher :

Experiment No 8

Aim: Write a program to implement two pass Macro Processor.

Learning Objective: To understand how the pre-processor replaces all the macros in the program by its real definition prior to the compilation process of the program.

Algorithm:

Pass1:

1. Set the MDTC (Macro Definition Table Counter) to 1.

2. Set MNTC (Macro Name Table counter) to 1.

3. Read next statement from source program.

4. If this source statement is pseudo-opcode MACRO (start of macro definition)

5. Read next statement from source program (macro name line)

6. Enter Macro name found in step 5 in name field of MNT (macro name table)

7. Increment MNTC by 1.

8. Prepare ALA

9. Enter macro name into MDT at index MDTC

10.Increment MDTC by 1.

11. Read source statement from source program

12. Create and substitute index notation for arguments in the source statement if any.

13. Enter this line into MDT

14. Increment MDTC by 1.

15. Check if currently read source statement is pseudo-opcode MEND. If yes then goto step 3 else goto step 11.

16. Write source program statement as it is in the file

17.Check if pseudo-opcode END is encountered. If yes goto step 18 else goto step 19

18. Goto Pass2

19. Go to step 3

20. End of PASS1.

Pass2:

1. Read next statement from source program

2. Search in MNT for match with operation code

3. If macro name found then goto step 4 else goto step 11.

4. Retrieve MDT index from MNT and store it in MDTP.

5. Set up argument list array

6. Increment MDTP by one.

7. Retrieve line pointer by MDTP from MDT

8. Substitute index notation by actual parameter from ALA if any.

9. Check if currently retrieved line is pseuodo-opcode MEND, if yes goto step 1 else goto step 10

10. Write statement formed in step 8 to expanded source file and goto step 6

11. Write source statement directly into expanded source file

12. Check if pseudo-opcode END encountered, if yes goto step 13 else goto step 1

13. End of PASS II

Implementation Details

1.Read input file with Macros

2.Display output of Pass1 as the output file, MDT, MNT, and ALA tables.

3.Display output of pass2 as the expanded source file, MDT, MNT and ALA tables.

Test Cases :

1. Call macro whose definition is not present

2. Define macro without MEND

PASS I:

Code:

```
Spcc > exp8.1 > C main.c > ...
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   
5   
6   int main(){
7       char label[10],opcode[10],operand[10];
8       int mntc=1,mdtc=1;
9   
10      FILE *fp1,*fp2,*fp3,*fp4;
11      fp1 = fopen("input.txt","r");    //input
12      fp2 = fopen("mnt.txt","w");      //output
13      fp3 = fopen("mdt.txt","w");      //output
14      fp4 = fopen("copyfile.txt","w");//output
15  
16      fscanf(fp1,"%s %s %s",label,opcode,operand);
17
```

```c
    while(strcmp(opcode,"END")!=0){
        if(strcmp(opcode,"MACRO")==0){
            fscanf(fp1,"%s %s %s",label,opcode,operand);
            fprintf(fp2,"%d %s %d\n",mntc,opcode,mdtc);
            mntc++;
            while(strcmp(opcode,"MEND")!=0){
                fprintf(fp3,"%d %s %s %s\n",mdtc,label,opcode,operand);
                mdtc++;
                fscanf(fp1,"%s %s %s",label,opcode,operand);
            }
            fprintf(fp3,"%d %s %s %s\n",mdtc,label,opcode,operand);
            mdtc++;
        }
        else{
            fprintf(fp4,"%s %s %s\n",label,opcode,operand);
        }
        fscanf(fp1,"%s %s %s",label,opcode,operand);
    }

    fprintf(fp4,"%s %s %s\n",label,opcode,operand);

    fclose(fp1);
    fclose(fp2);
    fclose(fp3);
    fclose(fp4);


    return 0;
}
```

Input:

Spcc > exp8.1 > input.txt

```
** MACRO **
** MATH **
** ar 5,3
** sr 5,4
** MEND **
** MACRO **
** MUL **
** mr 5,3
** MEND **
pg1 START 0
** USING *,15
** 1 1,FIVE
** MATH **
FIVE DC H'5'
** MATH **
** MUL **
** END **
```

Output:

Spcc > exp8.1 > mdt.txt

```
1  1 ** MATH **
2  2 ** ar 5,3
3  3 ** sr 5,4
4  4 ** MEND **
5  5 ** MUL **
6  6 ** mr 5,3
7  7 ** MEND **
```

Spcc > exp8.1 > mnt.txt

```
1  1 MATH 1
2  2 MUL 5
```

Spcc > exp8.1 > copyfile.txt

```
1  pg1 START 0
2  ** USING *,15
3  ** 1 1,FIVE
4  ** MATH **
5  FIVE DC H'5'
6  ** MATH **
7  ** MUL **
8  ** END **
```

PASS II:

Code:

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


int main(){
    char label[10],opcode[10],operand[10],mntc[10],macroname[10],mdtc[10],mdtmdtc[10],ignore[10],mdtlabel[10],mdtvalue[10];
    int flag = 0;

    FILE *fp1,*fp2,*fp3,*fp4;
    fp1 = fopen("copyfile.txt","r");        //input
    fp2 = fopen("mnt.txt","r");             //input
    fp3 = fopen("mdt.txt","r");             //input
    fp4 = fopen("expandedSource.txt","w");  //output

    fscanf(fp1,"%s %s %s",label,opcode,operand);

    while(strcmp(opcode,"END")!=0){
        if(strcmp(opcode,"START")==0 || strcmp(opcode,"USING")==0 || strcmp(opcode,"DC")==0 || strcmp(opcode,"DS")==0 ){
            fprintf(fp4,"%s %s %s\n",label,opcode,operand);
        }
        else{
            fscanf(fp2,"%s %s %s",mntc,macroname,mdtc);
            while(!feof(fp2)){
                if(strcmp(opcode,macroname)==0){
                    flag = 1;
                    break;
                }
                fscanf(fp2,"%s %s %s",mntc,macroname,mdtc);
```

```c
                    fscanf(fp2,"%s %s %s",mntc,macroname,mdtc);
                }
                if(strcmp(opcode,macroname)==0){
                    flag = 1;
                }
                if(flag==1){
                    flag = 0;
                    fscanf(fp3,"%s %s %s %s",mdtmdtc,ignore,mdtlabel,mdtvalue);
                    while(strcmp(mdtc,mdtmdtc)!=0){
                        fscanf(fp3,"%s %s %s %s",mdtmdtc,ignore,mdtlabel,mdtvalue);
                    }
                    fscanf(fp3,"%s %s %s %s",mdtmdtc,ignore,mdtlabel,mdtvalue);
                    while(strcmp(mdtlabel,"MEND")!=0){
                        fprintf(fp4,"%s %s %s\n",ignore,mdtlabel,mdtvalue);
                        fscanf(fp3,"%s %s %s %s",mdtmdtc,ignore,mdtlabel,mdtvalue);
                    }
                    rewind(fp3);
                }
                else{
                    fprintf(fp4,"%s %s %s\n",label,opcode,operand);
                }
                rewind(fp2);
            }
            fscanf(fp1,"%s %s %s",label,opcode,operand);
        }
```

```
56
57      fprintf(fp4,"%s %s %s\n",label,opcode,operand);
58
59      fclose(fp1);
60      fclose(fp2);
61      fclose(fp3);
62      fclose(fp4);
63
64
65      return 0;
66  }
```

Input:

Spcc > exp8.1 > input.txt

```
1   ** MACRO **
2   ** MATH **
3   ** ar 5,3
4   ** sr 5,4
5   ** MEND **
6   ** MACRO **
7   ** MUL **
8   ** mr 5,3
9   ** MEND **
10  pg1 START 0
11  ** USING *,15
12  ** 1 1,FIVE
13  ** MATH **
14  FIVE DC H'5'
15  ** MATH **
16  ** MUL **
17  ** END **
```

Spcc > exp8.2 > copyfile.txt

```
1   pg1 START 0
2   ** USING *,15
3   ** 1 1,FIVE
4   ** MATH **
5   FIVE DC H'5'
6   ** MATH **
7   ** MUL **
8   ** END **
```

Spcc > exp8.2 > mdt.txt

```
1   1 ** MATH **
2   2 ** ar 5,3
3   3 ** sr 5,4
4   4 ** MEND **
5   5 ** MUL **
6   6 ** mr 5,3
7   7 ** MEND **
```

Spcc > exp8.2 > mnt.txt

```
1   1 MATH 1
2   2 MUL 5
```

Output:

Spcc > exp8.2 > expandedSource.txt

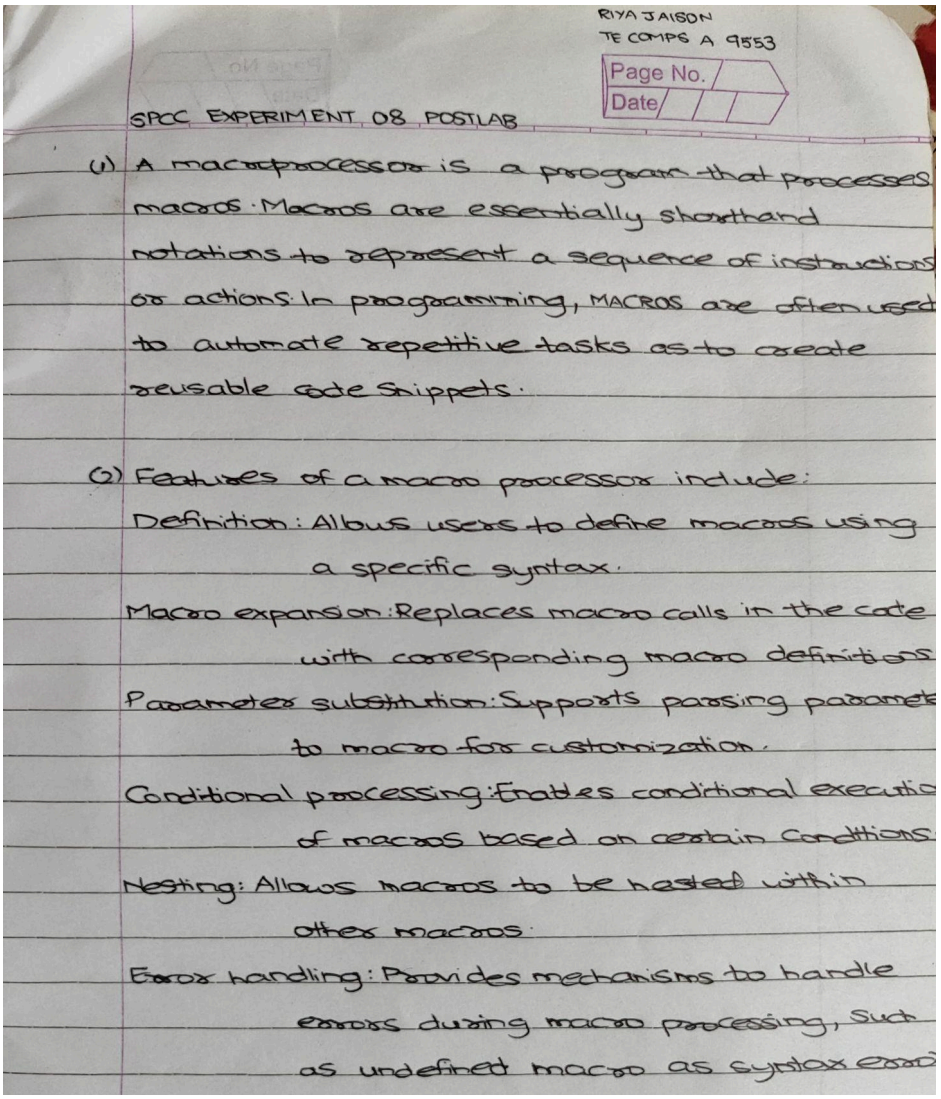
```
1   pg1 START 0
2   ** USING *,15
3   ** 1 1,FIVE
4   ** ar 5,3
5   ** sr 5,4
6   FIVE DC H'5'
7   ** ar 5,3
8   ** sr 5,4
9   ** mr 5,3
10  ** END **
```

Conclusion: Thus, we have implemented Two Pass Macroprocessor

Post Lab Questions:

1. **What is meant by macro processor**?

2. **What are the features of macro processor?**



RIYA JAISON
TE COMPS A 9553

Page No.
Date

SPCC EXPERIMENT 08 POSTLAB

(1) A macroprocessor is a program that processes macros. Macros are essentially shorthand notations to represent a sequence of instructions or actions. In programming, MACROS are often used to automate repetitive tasks as to create reusable code snippets.

(2) Features of a macro processor include:
Definition: Allows users to define macros using a specific syntax.
Macro expansion: Replaces macro calls in the code with corresponding macro definitions
Parameter substitution: Supports passing parameters to macro for customization.
Conditional processing: Enables conditional execution of macros based on certain conditions.
Nesting: Allows macros to be nested within other macros.
Error handling: Provides mechanisms to handle errors during macro processing, such as undefined macro as syntax error

SPCC ASSIGNMENT 01 RIYA JAISON TECOMPS A 9553

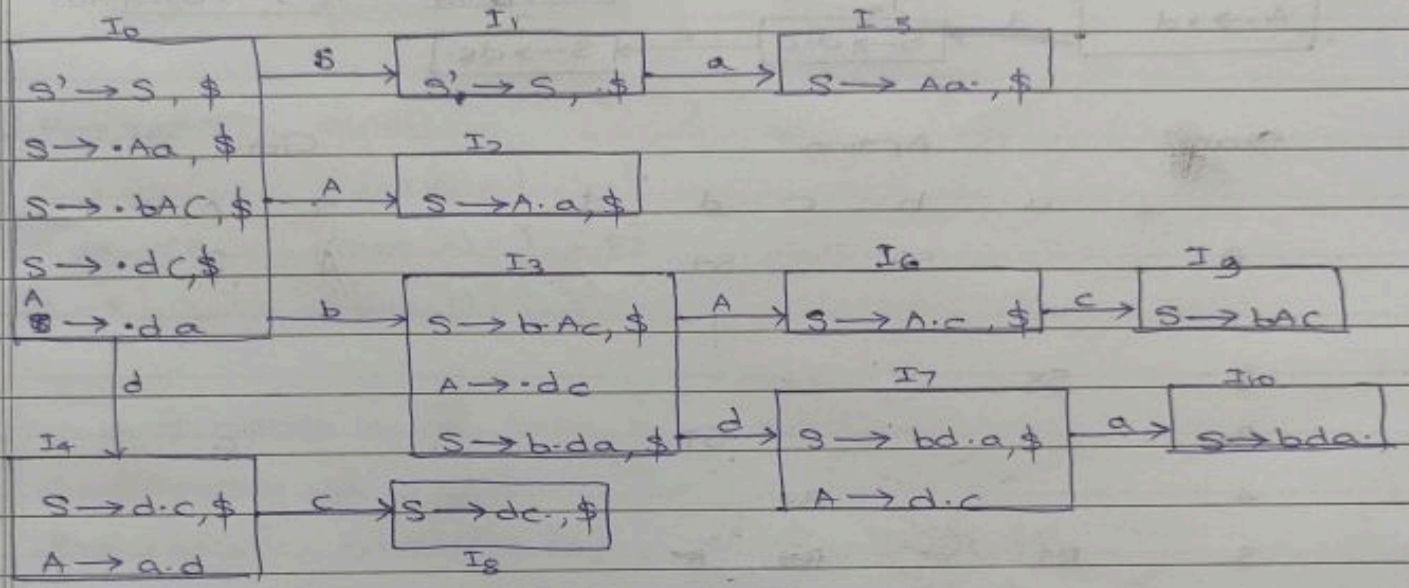
(1) $S \xrightarrow{①} Aa \;/\; \xrightarrow{②} bAC \;/\; \xrightarrow{⑤} dc \;/\; \xrightarrow{④} bda$

$A \xrightarrow{③} d$

Step 1: Make an ~~aug~~ augment state

$S' \rightarrow S$

Step 2: Go to Graph



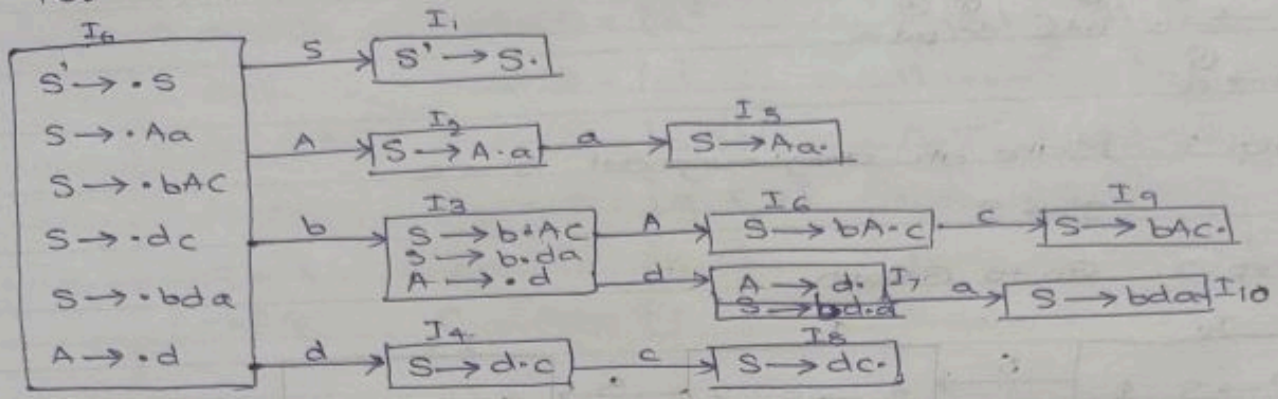
Step 3:

State	Action					Goto	
	a	b	c	d	$	S	A
0		S3		S4			2
1					Accept		
2	S3						
3				S7			
4			S8			6	
5							
6	R5		S9		R3		
7			R5		R2		
8	S10				R4		
9							
10							

26/02/2024 23:04

The LALR(1) is obtained by merging first common core.
The problem has no merging ∴ LALR(1) is same as LR(1)

For SLR(1)



State	Action					Goto	
	a	b	c	d	$	S	A
0		S3		S4		1	2
1							
2	S5						
3				S7			6
4			S8				
5	R1		~~R3~~	~~R~~			
6			S9				
7				R5			
8			R3				
9			R2				
10	R4						

(2)

S → aA		First ~~Follow~~(S) = {a}	Follow(S) = {$}
A → BD		First (A) = {b, d, ε}	Follow(A) = {$}
B → b/ε		First(B) = {b, ε}	Follow(B) = {d, ε, $}
D → d/ε		First(D) = {d, ε}	Follow(D) = {$}

(3)

(a)	S → Aa	First (S) = {b, d, ε}	Follow(S) = {$}
	A → BD	First (A) = {b, d, ε}	Follow(A) = {a}
	B → b/ε	First (B) = {b, ε}	Follow(B) = {d, ε, a}
	D → d/ε	First (D) = {d, ε}	Follow(D) = {a}

26/02/2024 23:02

(b) S → aBDh First(S) = {a} Follow(S) = {$}

B → cC First(B) = {c} Follow(B) = {g, f, h}

C → bc/ε First(C) = {b, ε} Follow(C) = {g, f, h}

D → EF First(D) = {g, f, ε} Follow(D) = {h}

E → g/ε First(E) = {g, ε} Follow(E) = {f, h}

F → f/ε First(F) = {f, ε} Follow(F) = {h}

(c) P → XQRS First(P) =

Q → yz/z First(Q) = {y, z}

R → w/ε First(R) = {w, ε}

S → y First(S) = {y}

? IS IT CAPITAL X OR SMALL X

Considering 'x'

P → xQRS First(P) = {x} Follow(P) = {$}

Q → yz/z First(Q) = {y, z} Follow(Q) = {w, y}

R → w/ε First(R) = {w, ε} Follow(R) = {y}

S → y First(S) = {y} Follow(S) = {$}

(4) S → aAbB / bA aB/ε First(S) = {a, b, ε} Follow(S) = {b, a

A → S First(A) = {a, b, ε} Follow(A) = {b, a, $}

B → S First(B) = {a, b, ε} Follow(B) = {a, b, $}

	a	b	$
S	S → aAbB	S → bAaB	S → ε
A	A → S	A → S	error
B	B →	B → S	B → S

E1 = S → aAbB

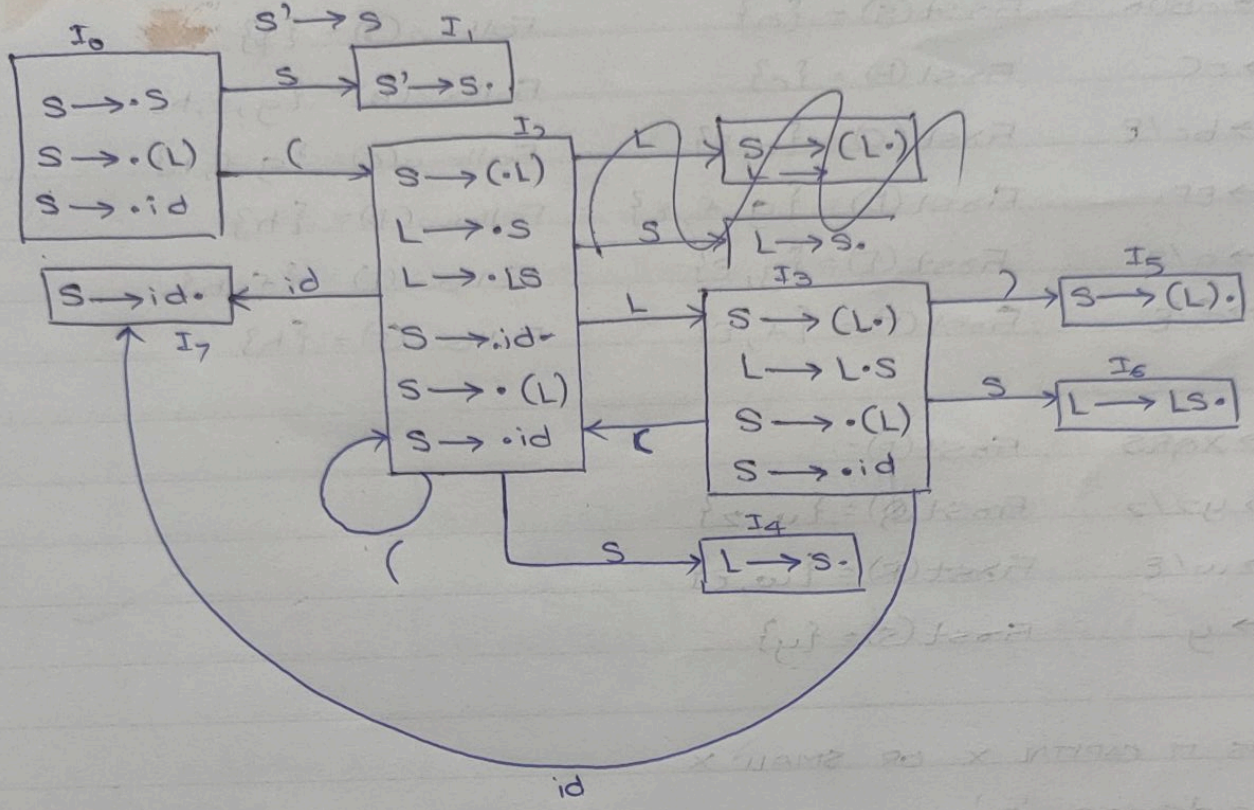
E2 = S → bAaB

E3 = B → S

(6) (a) $S \to (L)^{1} / id^{2}$

$L \to S^{3} / LS^{4}$

Step 1 : Augment the start stack



State	(	)	id	$	S	L
0					1	
1				Accept		
2	S2		S7		4	3
3	S2	S5	S7		6	
4	R3	R3	R3	R3		
5	R1	R1	R1	R1		
6	R4	R4	R4	R4		
7	R2	R2	R2	R2		

(b) $S \to aCDe$

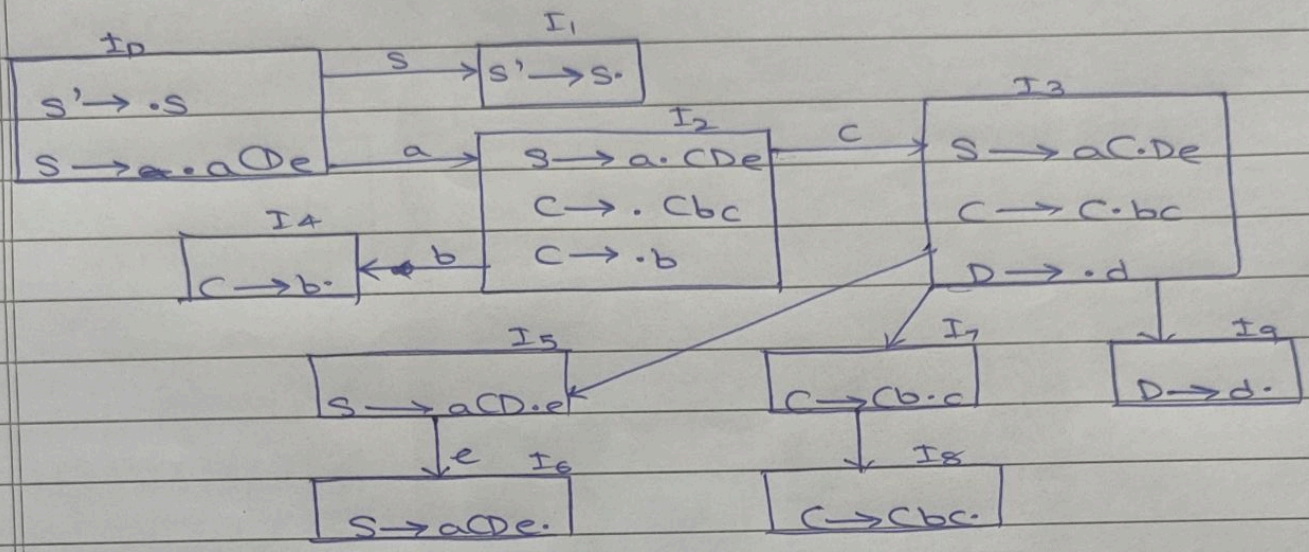
$C \to ~~EDe~~ \; Cb/c$

$C \to b$

$D \to d$

$S' \rightarrow S$

Go To Graph :



(6) 1. Lexical Analyzer
2. Semantic Analyzer
3. Syntax Analyzer
4. Lexical Analyzer
5. Lexical Analyzer
6. Lexical Analyzer
7. Semantic Analyzer
8. Syntax Analyzer
9. Semantic Analyzer
10. Semantic Analyzer

26/02/2024 23:02

RIYA JAISON
TE COMPS A
9553

Page No.
Date

SPCC ASSIGNMENT 02

(1) In a two pass assembler, both Pass 1 & Pass 2 utilize different data structures to process assembly code

PASS 1:

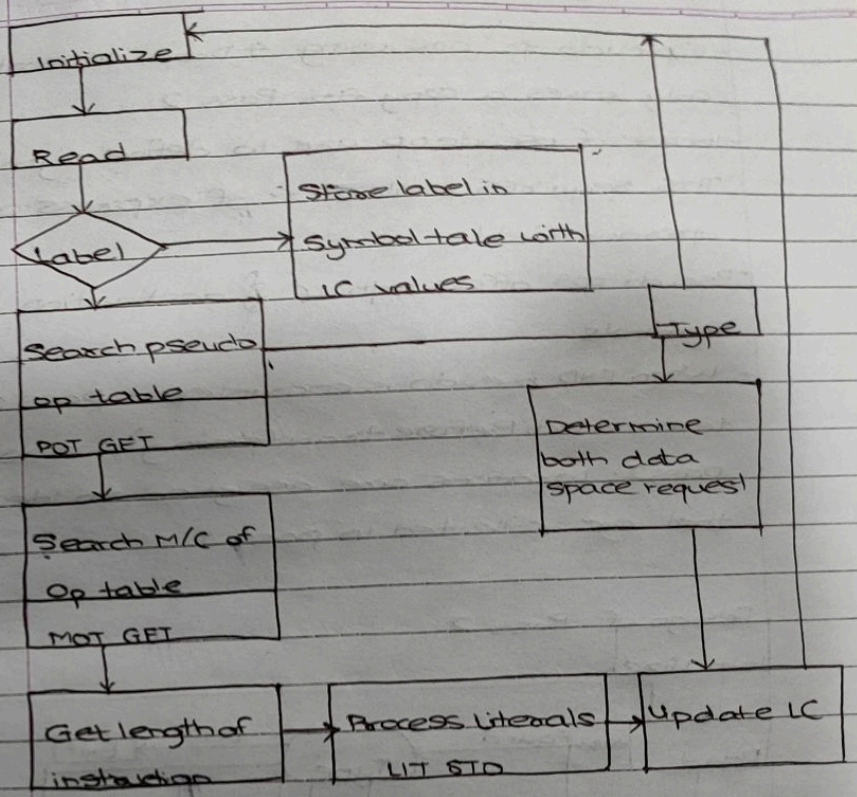
- Symbol Table: This table records all symbols encountered in source code along with their corresponding addresses. It helps resolve symbols & calculate addresses.
- Literal Table: Stores literal constants in source code along with addresses. This table is necessary for assigning address to literals.
- Location counter: Keeps track of current location in program being assembled. It is incremented as instructions are encountered allowing for calculation of addresses for calculation of symbols.

PASS 2:

- Intermediate Code: Assembler translates the assembly instructions into machine code or generates an intermediate representation of code. Table stores translated machine instruction or intermediate code along with address
- Symbol Table: Gen. in pass 1 - also used in pas 2 to resolve symbols & address
- Literal Table: Similar to pass 1, used to assign addresses to literals

Page No.
Date

(2)

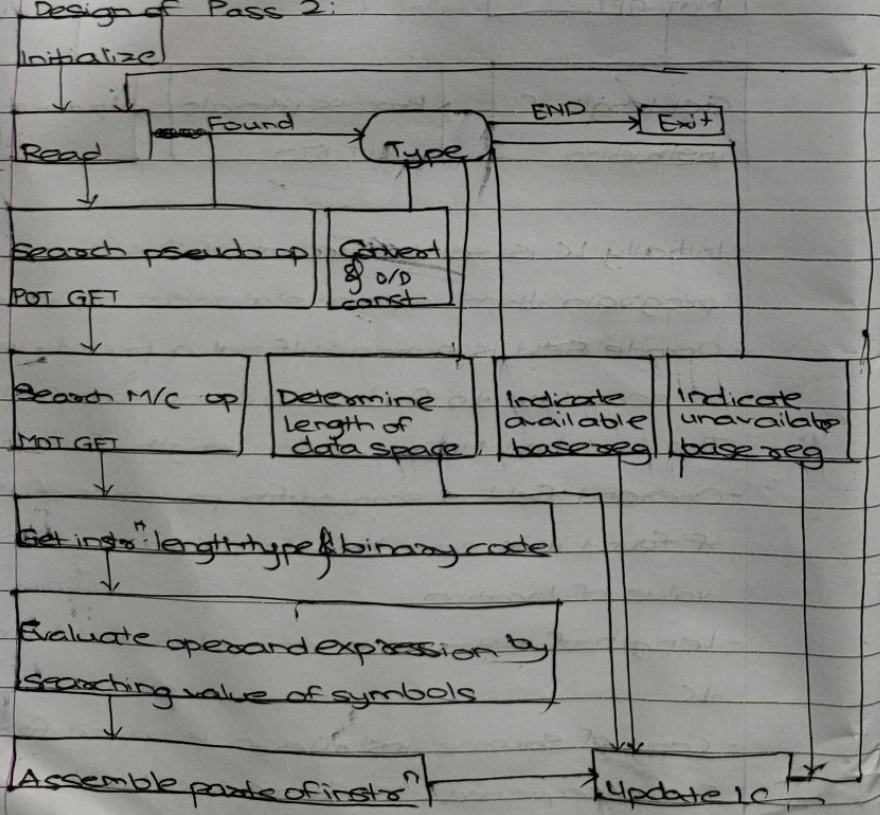


- Initially LC is set to location of first instruction in program then source statement is read.
- Opcode field is examined. If not a pseudo op, MOT is searched to find source op-code field.
- The matched MOT specifies length of instruction
- Operand field is scanned for presence of literal; if found it is saved in SYMTAB along with current value of location.
- Length of instruction stored is used to increment LC.
- Copy of source instruction is saved for pass 2
- Same sequence used for next instruction.

17/04/2024 22:18

- If pseudo ops are using 4 Dbop assembler only saves a copy for Pass 2.
- Incase of pseudo op are to define symbol. This requires evaluation of expression in operand field.
- Pseudo ops affect both LC & destination of symbols.
- When END pseudo is encountered pass 1 is terminated. Before transferring control to pass2. locations are assigned to literal that are collected in pass 1.

Design of Pass 2:



- ~~The operand fields of data instructions req~~ etc

(3)

INPUT	LC	MOT	
START 0	0	Index	Instruction
Using 15	0	01	L
L1, five	0	02	A
A1, four	4	03	S1
ST1, temp	8	04	SR
SR2, 2	12	05	AD
AD 1, 2	14		
five DC 7'5'	16	POT	
four DC 7'4'	20	Index	Assembler directive
temp DS 2f	24	01	START
temp DS 3f	32	02	Using
End put	44	03	END

Symbol Table

Symbol	Address	Length	R/L
five	16	1	R
four	20	4	R
temp	24	8	R
temp1	32	12	R

Source Prog.	LC	First Pass	Second Pass
start 0	0		
Using 2,15	0		
L1, five	0	L1,-(0,17)	01,1,16(0,15)
A1, four	4	A1,-(0,15)	02,1,20(0,14)
ST1, temp	8	ST1,-(0,14)	03,1,24(0,14)
SR 2,2	12	SR 2,2	04,2,2

AR1, 2	14	AR 1,2	05 1,2	
five DC 7'5'	16	5	10 5	
four DC 7'4'	20	4	20 4	
temp DS 2f	24		24 -	
temp1 DS 3f	32		32 -	
End Pul	44			

(4) Pg2 Start 0

Using *, B

L 1, two

A 1, three

A 1, three

M 1, 7'2'

ST 1, temp

SR 2,2

B Eav 4

two DC H'2'

three DC F'2'

temp DS tf

END

MOT

Index	Instruction
01	L
02	AR
03	A
04	M
05	ST
06	SR

POT

Index	Assembly directive
01	Start
02	Using
03	EQU
04	END

Symbol	Address	Length	R/A
PG2	0	1	R
B	4	1	A
two	22	2	R
three	24	4	R
temp	28	4	R

Literal	LC	Length	R/A
=7'2'	32	4	R

Source	LC	Pass 1	Pass 2
PG Start 0	0		
Using *B	0		
L1, two	0	L 1,-(0,4)	01,22(0,4)
AR 1, three	4	AR 1,-(0,4)	02,24(0,4)
A 1, three	8	A 1,-(0,4)	03,1,24(0,4)
m 1,=7'2'	12	m 1,-(0,4)	04 1,32(0,4)
ST 1, temp	16	ST 1,-(0,4)	05 1,23(0,4)
SR 2,2	20	SR 2,2	002, 2
Eav 4	22	-	
two DC A'2'	22	2	22 2
three DC f'2'	24	2	24 2
temp DS 1f	28	-	
End	32	-	

(5)

		LC
→ RAM	START 0	0
	Using *15	0
	L1, five	0
	A1, four	4
	ST 1, temp	8
four	DC f'4'	12
five	DC f'9'	16
temp	DC 1f	20
	END	24

17/04/2024 22:18

Date

SINGLE PASS ASSEMBLER	TWO PASS ASSEMBLER
-Performs translation in ~~Pass 1~~ one pass	-Performs translation in two passes
-Intermediate code not generated	-Generation of intermediate code
-Default address are zero for symbols & literals	-After pass 1, all symbols & literals are getting address update to address
-More memory req.	-Less memory req.
-Faster	-Slower

Symbol Table

Symbol Labels	Length	LC	R/A
RAM	—	0	R
FOUR	4	12	R
FIVE	4	16	R
TEMP	4	20	R

Forward Reference Table

Symbol	Stat NO	Instruction Address
five	3	0

17/04/2024 22:19

RIYA JAISON
TE COMPS A
9553

Page No.
Date

SPCC ASSIGNMENT 03

(1)

MACRO	FUNCTION
-Preprocessed	- Compiled
-No type checking done	-Type checking done
-Increases code length	- Code length unaffected
-May lead to side effects	-No side effects
-Faster speed of execution	- Slower speed of execution
-Useful when small code is repeated multiple times	-Useful when large code is written
-Does not check compile time errors	- Checks compile time errors

Functions preferred over macro as:
Type safety, debugging, scoping, readability

(2) (a) Textual substitution - Macroprocessors perform this replacing macros within corresponding text

Eg: ; Define a macro to calculate square

```
SQUARE_MACRO MACRO num
    MOV AX, num
    IMUL AX, num
; Usage of SQUARE_MACRO
SQUARE_MACRO 5  ; expands to MOV AX, 5,
                          IMUL AX
```

(b) Parameterized macros: Allowing dynamic behaviour based on input values.

Eg: #define MAX(a,b) ((a)>(b)?(a):(b))

```
int main () {
    int x=10, y=20;
    int max_value = MAX(x,y);
    return 0;
}
```

17/04/2024 23:23

(c) Conditional compilation:
Macroprocessors can enable or disable parts of code based on conditions.

```
Eg: #define DEBUG_ENABLED
    #ifdef DEBUG_ENABLED
    #define DEBUG_LOG(msg) printf("DEBUG:%s\n",msg)
    #else
    #define DEBUG_LOG(msg)
    #endif
    int main(){
      DEBUG_LOG("Debug msg");
      return 0;
    }
```

(d) Code Generation:
Macros can generate code snippets or entire block of code based on predefined rules

```
Eg: #define DECLARE_VARIABLE(type,name)type name;
    int main(){
      DECLARE_VARIABLE(int, num);
      return 0;
    }
```

(e) Iteration & looping:
Macroprocessors can be used for iterative tasks & looping constructs.

```
Eg: #define REPEAT(count, action)
    do {\
        for(int i=0; i<count; i++){\
          action; \
        }\
    } while(0)
    int main(){
      REPEAT (3, printf("Hellowworld");
      return 0;
    }
```

(f) Modularization & Code Reuse: Macros facilitate this by encapsulating common functions into reusable units.

```
Eg: #define ARRAY_LENGTH(arr) (sizeof(arr)/sizeof(arr[0]))
    int main(){
      int numbers[] = {1,2,3,4,5};
      int length = ARRAY_LENGTH(numbers);
      return 0;
    }
```

(3) Used for identifying macro performing expansion forward reference problem.
The assembler specifies that macro definition should occur anywhere in program so there can be chances of macro call before which give rise to forward ref problem of macro
Due to which macro is divided into two passes

1. Pass 1:
   Recognize macro definition, save
2. Pass 2:
   (i) Copy file - Contains output from Pass1
   (ii) MNT - used for recognizing macro name
   (iii) MDT - used to perform macro expansion

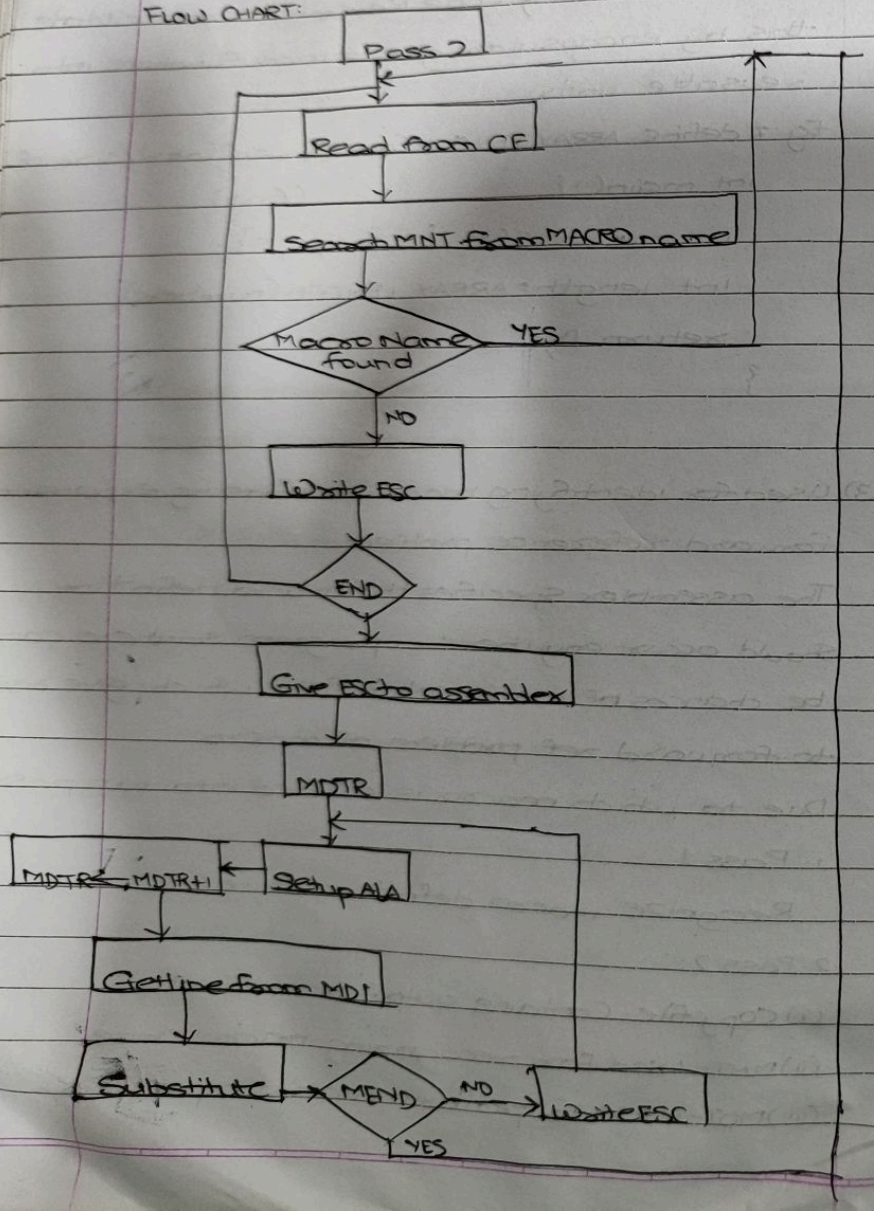
17/04/2024 23:23

(iii) MDTP - Used to point to index of MDT

(v) AIA - Used to replace index notation by actual value

(vi) ESC - Used to contain expanded macro call which is given to assembler for further processing

FLOW CHART:



(4) (i) MACROS Table

MACRO	INCR	&m	, = f'2'	&S12, temp	&S6 2,2	&m

(ii) Symbol Table

Symbol	Value	Length	R/C	Type
Pg2	0	-	R	Program name
*	b	-	R	Register
two	-	2	R	Constant
three	-	4	R	Constant
temp	-	1	R	Variable

(iii) Literals Table:

Literal	Value	Length	Type
f'2'	2	2	Numeric
f'5'	5	2	Numeric

(iv) Pseudo Instruction Table

Instruction	Operand 1	Operand 2
USING	*	b
START	0	

(v) Machine Instructions Table

Address	Mnemonic	Operand 1	Operand 2
-	L	1	two
-	AR	1	2
-	INCR		
-	A	1	three
-	S	5	=f'5'
-	D	5	three

17/04/2024 23:24

Page No.
Date

(vi) EQV Table

Symbol	value
b	4

(vii) Data section table

Symbol	Value	Length	Type
two	2		Constant
three	2.0		Constant
temp			Variable (1 word)