Regularity	Content	Viva-voce	Timely Submission	Total	Dated Sign of Subject Teacher
2	4	2	2	10	

Date of Performance:................................... Expected Date of Completion:.......................

Actual Date of Completion:..

Assignment No: 1

Title of the Assignment: ASSIGNMENT BASED ON ASSEMBLER.

Problem Statement:

1. Write a program to implement pass I of a II pass assembler.
a. Consider following cases only (Literal processing not expected)
b. Forward references
c. DS and DC statement
d. START, EQU
e. Error handling: symbol used but not defined, invalid instruction/register etc.

Objective of the Assignment:

Implementation of TWO Pass assembler with hypothetical Instruction set.Instruction set should include all types of assembly language statements such as Imperative, Declarative and Assembler Directive.

Prerequisite: Basic C language Concepts.

Theory:

1.Introduction: Assembler: Definition

- Translating source code written in assembly language to object code.
- Assigning machine address to symbolic labels

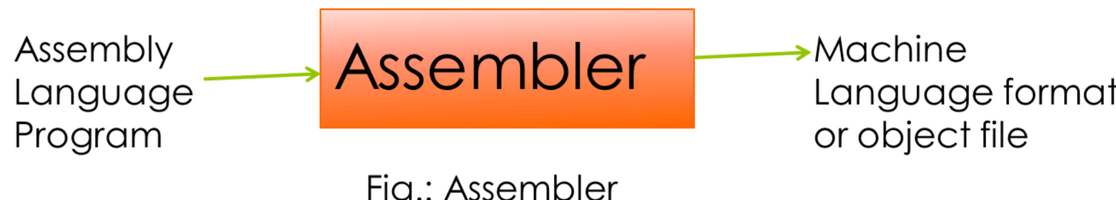


Fig.: Assembler

- The Assembler language is a symbolic representation of machine language.

- It uses a mnemonic to represent each low-level machine instruction or operation.
- Assemblers with different syntax for a particular CPU or instruction set architecture.

2.Elements of assembly language programs:

Assembly lang. Provides 3 basic features:

1. Basic features
2. Statement format
3. Operation code

1. Basic features:

1.1 Mnemonic Operation Codes (Opcodes)

Use mnemonic operation code

Ex: MOVER or MOVEM

1.2. Symbolic Operand:

Symbolic names can be associated with data or instructions, these symbolic names can be used as operands in assembly stmt., the assembler performs memory binding to these names.

Ex. X DS 5 [DS – Declare as storage]
Y DC 4 [DC – Declare as Constant]

1.3. Data Declaration: Data can be declared in a variety of notation including the decimal notation

Ex: X DC '-10.5'

2.Statement Format:

[Label] <opcode> <operand1> [<operand2>..]

Label-Optional

Opcode-it contain symbolic operation code

Operand- Operand can also be a CPU register: AREG, BREG,CREG.

Example-

LOOP : MOVER AREG, '=5'

3.Assembly language statements

3 types of Statement in Assembly Language

1. Imperative
2. Declaration
3. Assembler directives

3.1. Imperative Statement

- These are executable statement.
- Each imperative statement indicates an action to be taken during execution of the program.
- Each imperative statement translates into a machine instruction
- **Ex.: class IS**

MOVER BREG, X
STOP
READ X
PRINT Y
BC NE,L1

3.2 Declaration Statement

- Reserves memory for variables
- Initial value of a variable can also be specified.

[label] DS <const specifying size of memory to be reserved>
[label] DC <Initial value of variable>
Class-DL

3.3 Assembler Directive

- A **directive** is a direction for the assembler
- A **directive** is also known as pseudo instruction
- **Machine code** is **not generated** for AD.

3.3.1 START

Syntax: START<Constant>

- It indicates that the **first word** of the m/c code should be placed in the memory word with the address **<CONSTANT>**

3.3.2 EQU

EQU set the address of Symbol at the location of address specification
Syntax:
<symbol> **EQU** <address specification>
Where,
<address specification> **:**can be **operand specification** or a **constant**
<symbol>**: EQU** Associates **symbol** with the <address specification>
Ex. BACK EQU LOOP
The symbol **BACK** is set to the address of **LOOP**

4.Forward Reference:

Symbols that are defined in the later part of the program are called forward referencing. There will not be any address value for such symbols in the symbol table in pass 1.

- It is reference to the entity which precedes its declaration.
  Examples: x=y+5 ;
  int x,y;
- The compiler will not be able to generate the m/c code for the statement "x=y+5" **until** it has been declaration of two variables x & y

5.Types of Assembler

They are three types of Assemblers

1. Load and Go-Assembler
2. One-pass Assembler
3. Two-pass Assembler

5.1 Load and Go-Assembler:

- It is Simplest form of assembler
- It produces machine language as output which are loaded directly in main memory and executed
- The ability to design code and test the different program components in parallel

5.2.One Pass Assembler

- Normally , it does not allow forward referencing.
- An assembler cannot generate m/c code for an assembly instruction with FR.

- Machine code is generated ,after the address of variable used in the instruction is known.
- Symbol table is used to record the address of the variables.

5.3 Two Pass Assembler

- Two passes(stages),

Pass I

1. Separate the symbols, mnemonic op-code and operand fields.
2. Determine the storage requirement for every assembly language statement and update the location counter.
3. Build the symbol table.

Pass II

Generate machine code.

5.4. Design of Two Pass Assembler

1.First Pass:

Pass-I of the assembler involve scanning of the source file.

- First pass has to fix address of variables
- Data structures required:
- – MOT-use to search the opcode
- – Symbol table-use to search the symbol
- – Literal table
- – Pool table : starting literal number of each pool.

2.Second Pass:

- **t**he second pass can generate the machine code.

5.4.1 MOT-Machine operation code-MOT are fixed for an assembler

A MOT Contains

1. Opcode in mnemonic form
2. Machine code associated with the opcode

Mnemonic opcode	Machine code for opcode	Class
STOP	00	IS
ADD	01	IS
SUB	02	IS
MULT	03	IS
MOVER	04	IS
MOVEM	05	IS
COMP	06	IS
BC	07	IS
DIV	08	IS

READ	09	IS
PRINT	10	IS
END	02	AD
ORIGIN	03	AD
EQU	04	AD
LTROG	05	AD
DS	01	DL
DC	02	DL
AREG	01	RG
BREG	02	RG
CREG	03	RG

5.4.2 Class Field Indicate:

Sr.No.	Type of Mnemonic Symbol	Value of class Field
1	Imperative Statement(IS)	1
2	Declaration Statement(DL)	2
3	Assembler directive(AD)	3
4	CPU Register(RG)	4
5	Condition code(CC)	5

5.4.3 Symbol Table:

- It contains:

1. Name of variable or a label
2. Its address
3. Its size in number of words

5.4.4 Literal Table

- It contains:

1. Value of the literal
2. Address of the memory location associated with the literal.

5.4.5 Pool Table

- This table contains the literal number of the **starting literal** of each literal pool.

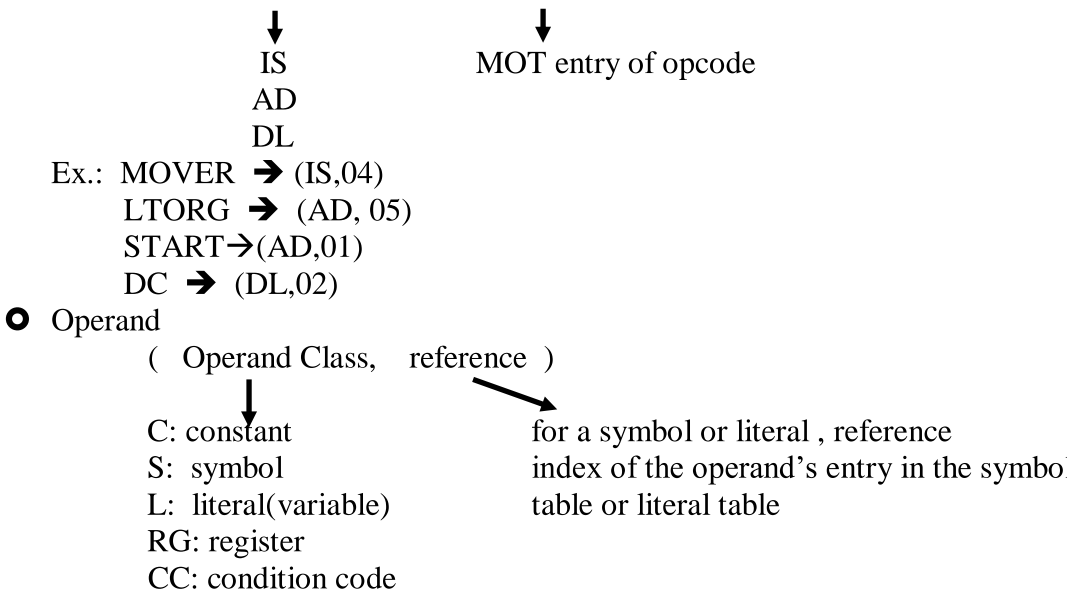
5.4.6 Intermediate Code:

- Intermediate code is equivalent representation
- Every opcode search in MOT table
- Every symbol search in symbol table(ST)
- Every opcode is searched in MOT
- Every operand is searched in symbol table.
- Intermediate code helps in avoiding:
  1. Scanning of source file in PASS-II
  2. Searching MOT and ST in PASS-II

5.4.7 Format of Intermediate Code

- Each Mnemonic opcode field is represented as:

(Statement class , Machine code)



6. Error Handling:

An Assembly language program may contain errors. It may be necessary to report these errors effectively. Some errors can be reported during processing in Pass-I. Some error can only be reported at the end of the resource program. Some of the typical error includes:

- Syntax errors like missing commas etc.
- Invalid opcode
- Duplicate definition of a symbol
- Undefined Symbol
- Missing START Statement
- Missing END Statement
- Symbol defined but not used

Input: Assembly language source program

START 101
MOVER AREG B
MOVER BREG A
STOP
B DS 1
A DC 1
END

Conclusion:

Thus we Implement Pass1 & pass2 Assembler

FAQs:

1. What do you mean by forward referencing?
2. What is one-pass assembler and two pass assembler?
3. What is output of pass-1 in two-pass assembler?

4. Which directives are known as advanced assembler directives?
5. Which types of mnemonic symbols are available for assembler?

Regularity	Content	Viva-voce	Timely Submission	Total	Dated Sign of Subject Teacher
2	4	2	2	10	

Date of Performance:.................................... Expected Date of Completion:.......................
Actual Date of Completion:..

Assignment No: 2

Title of the Assignment: ASSIGNMENT BASED ON ASSEMBLER.

Problem Statement:

2. Write a program to implement Pass-II of Two-pass assembler for output of Assignment 1 (The subject teacher should provide input file for this assignment)
Consider Literal processing, forward references not expected
a. Use of literals and not symbols
b. LTORG, END
c. Error handling

Objective of the Assignment:

Implementation of TWO Pass assembler with hypothetical Instruction set.Instruction set should include all types of assembly language statements such as Imperative, Declarative and Assembler Directive.

Prerequisite: Basic C language Concepts.

Theory:

1.Introduction: Assembler: Definition

- Translating source code written in assembly language to object code.
- Assigning machine address to symbolic labels

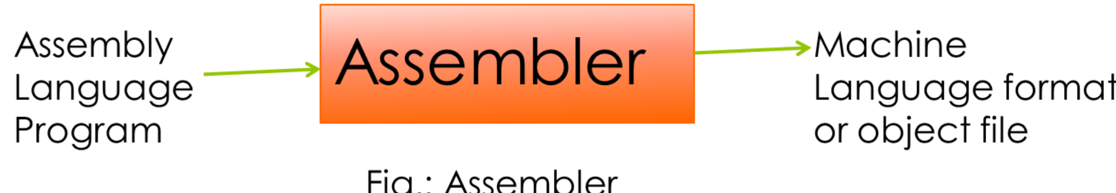


Fig.: Assembler

- The Assembler language is a symbolic representation of machine language.
- It uses a mnemonic to represent each low-level machine instruction or operation.

- Assemblers with different syntax for a particular CPU or instruction set architecture.

2.Literal

- A literal is an immediate operand
- A literal is an operand with constant value.
- In the c-statement
  int z=5;
  x=y+5;
- The constant value is ‘5’ known as literal.
- Literal can not be change during program execution
- They are specified using immediate addressing.

3.Assembler Directive:END

END [<OPERAND SPECIFICATION>]

- Optional, indicates address of the instruction where the **address of program should begin**.
- By default, execution begins from the first instruction.
- It indicates the end of the source program.
- **Class**:AD

4.LTORG

- The LTORG statement permits a programmer to specify where literal should be placed.
- If the LTORG statement not present, literal are present at the END statement
- At every LTORG Statement, memory is allocated to the literal of the current pool of literals.
- The pool contains all literal used in the program since the start of the program or since the last LTORG statement.

Input: Assembly language source program of PASS-1 Assembler

Conclusion:

Thus we Implement Pass1 & pass2 Assembler

FAQs:

1. What is Literal?
2. How to handle literal in single pass assembler?
3. What is important of Intermediate code?

Regularity	Content	Viva-voce	Timely Submission	Total	Dated Sign of Subject Teacher
2	4	2	2	10	

Date of Performance:.................................... Expected Date of Completion:.......................

Actual Date of Completion:...

Assignment No: 3

Title of the Assignment: ASSIGNMENT BASED ON MACROPROCESSOR.

Problem Statement: Study Assignment for Macro Processor. (Consider all aspects of Macro Processor)

Objective:

1. To understand macro facility, features and its use in assembly language programming.
2. To study how the macro definition is processed and how macro call results in the expansion of code.

Prerequisite: Basic C Concepts.

Theory:

1.Introduction:

- An assembly language macro facility is to extend the set of operations provided in an assembly language.
- In order that programmers can repeat identical parts of their program macro facility can be used.
- This permits the programmer to define an abbreviation for a part of program & use this abbreviation in the program.
- This abbreviation is treated as macro definition & saved by the macro processor.

- For all occurrences the abbreviation i.e. macro call, macro processor substitutes the definition.

2. Macro definition part:

It consists of

1. Macro Prototype Statement - this declares the name of macro & types of parameters.
2. Model statement - It is statement from which assembly language statement is generated during macro expansion.
3. Preprocessor Statement - It is used to perform auxiliary function during macro expansion.

3. Macro Call & Expansion:

- The operation defined by a macro can be used by writing a macro name in the mnemonic field and its operand in the operand field.
- Appearance of the macro name in the mnemonic field leads to a macro call.
- Macro call replaces such statements by sequence of statement comprising the macro. This is known as macro expansion.

4. Macro Facilities:

1. Use of AIF & AGO allows us alter the flow of control during expansion.
2. Loops can be implemented using expansion time variables.

5. Design Procedure:

1. Definition processing - Scan all macro definitions and for each macro definition enter the macro name in macro name table (MNT) and macro definition table position (MDTP).
2. Store entire macro definition in macro definition table (MDT)
3. Store parameter information in ALA- no. of positional parameters (#PP) and no. of key word parameters (#KP),
4. Macro expansion - Examine all statement in assembly source program to detect the macro calls. For each macro call locate the macro in MNT, retrieve MDTP, establish the correspondence between formal & actual parameters and Expand the macro.

6. Data structures required for macro definition processing -

Macro Name Table [MNT] - Name of Macro, MDTP (Macro Definition Table Pointer), Keyword Parameters

Parameter Name Table [PNTAB] -

Fields - Parameter Name

Keyword parameter Default Table [KPDTAB] -

Fields - Parameter Name, Default value

Macro Definition Table [MDT] -

Model Statement are stored in the intermediate code from as:

Opcode, Operands.

7.Main Program Logic :

1. Initialize KPDTAB_ptr, MDT_ptr to 0 and MNT_ptr to -1. These table pointers are common to all macro definitions (There could be more than one macro definition in program)
2. Read the statement from source file, one line at time
3. Separate the words from that line and count the no of words. Save the separated words in the array say word which is a array of strings
4. If count for words in a line is one then check if that only word matches with “MACRO” keyword, if MACRO keyword found then perform definition processing.
5. If it does not match then check whether first word of the line matches with any of the entries in the macro name table. (Search the complete macro name table for presence of macro call), if so then perform macro expansion routine.
6. If no Macro call or no definition then enter the line as it is in the expanded code file.
7. If not end of file go to step 3.

8. Macro Without Parameter:

Input: Source program in assembly language

Source program for Pass-I

START 101

ABC

COMP CREG,N

MACRO

ABC

MOVEM AREG,A

MEND

END

Output:

MNT..			**MDT..**	
			1	ABC
1	ABC	1	2	MOVEM AREG,A
			3	MEND

Intermediate code	**Expanded code (Output of Pass2)**
START 101	START 101
ABC	MOVEM AREG,A
COMP CREG,N	COMP CREG,N
END	END

9. Macro With Parameter:

Input: Source program in assembly language

Source program for Pass-I

START 101

ABC N,ADD

COMP CREG,N

MACRO

ABC &ARG1,&ARG2

MOVEM AREG,&ARG1

&ARG2 CREG,='9'

MEND

MACRO

END

Output:

MNT..			**MDT..**
1	ABC	1	1 MOVEM AREG,#0
			2 #1 CREG,’=9’
			3 MEND

Argument List Array

1 &ARG1

2 &ARG2

Intermediate code	**Expanded code (Output of Pass2)**
START 101	STARTS 101
ABC N,ADD	MOVEM AREG,N
COMP CREG,N	ADD CREG,='9'
END	COMP CREG,N
	END

Conclusion:

Thus we have implemented Pass1 & pass2 Macroprocessor.

FAQ's:

1. Define the term macro.
2. Distinguish between macro and a subroutine
3. Define and Distinguish between parameters that can be used in macros.
4. State various tables used in processing the macro.
5. Explain the role of stack in nested macros.
6. What are the contents for MNT and MDT?

Regularity	Content	Viva-voce	Timely Submission	Total	Dated Sign of Subject Teacher
2	4	2	2	10	

Date of Performance:...................................Expected Date of Completion:.......................

Actual Date of Completion:..

Assignment No: 4

Title of the Assignment: ASSIGNMENT BASED ON COMPILER

Problem Statement: Write a program to convert RE to DFA.

Objective: To understand the role of regular expressions and finite automata in applications such as Compilers.

Prerequisite: Basic Concept of Lexical Analyzer,TOC.

Theory:

1.Introduction:

Compiler is a program which converts the given source program in high-level language into an equivalent machine language. While doing so, it detects errors and reports errors. This process is quite complex and it is divided into number of phases such as Lexical Analysis, Syntax and Semantic Analysis, Intermediate Code generation, Code Generation and Code Optimization.

The lexical analysis phase of compiler reads the source program, character by character and then groups these characters into tokens which are further passed to next phase, which is nothing but parsing or syntax or semantic analysis. After syntax and semantic analysis, Intermediate Code is generated which is followed by actual code generation.

2.Lexical Analyzer:

- It recognizes the tokens from series of characters.

- A "C" program consists of tokens such as Identifiers, Integers, Floating Point Numbers, Punctuation symbols, relational and logical and arithmetic operators, keywords and comments (to be removed).
- To identify these tokens, lexical analyzer needs the specification of each of these symbols.
- The set of words belonging to a particular token type is a regular language.
- Hence each of these token types can be specified using regular expressions.

3.Regular expressions

Regular Expression are used to specify regular languages and finite automata are used to recognize the regular languages. Many computer applications such as compilers, operating system utilities, text editors make use of regular languages. In these applications, the regular expressions and finite automata are used to recognize this language.

For example, consider the token Identifier. In most of the programming languages, an identifier is a word which begins with an alphabet (capital or small) followed by zero or more letters or digits (0..9). This can be defined by the regular expression

(letter) . (letter | digit)* where

letter = A|B|C|……|Z| a| b |c |……|z

and digit = 0|1|2|….|9

One can specify all token types using regular expressions. These regular expressions are then converted to DFA's in the form of DFA transition table. Lexical analyzer reads a character from a source program and based on the current state and current symbol read, makes a transition to some other state. When it reaches a final state of DFA, it groups the series of characters so far read and outputs the token found.

4.Formal definition of Regular expression

The class of regular expressions over ∑ is defined recursively as follows:

1. The letters φ and Є are regular expressions over ∑ .

2. Every letter 'a' c ∑is a regular expression over ∑ .

3 If 'R1' and 'R are regular expressions over ∑, then so are '(R1|R2)', '(R1.R2)' and (R1)*

Where

'|' indicates alternative or parallel paths.

'.' Indicates concatenation

'*' indicates closure

4. The regular expressions are only those that are obtained using rules (1) and (2).

5. Formal definition of DFA:

The formal definition of finite automata is denoted by a tuple

(Q, ∑,d, qo, f)

Where

Q → Finite set of table

∑→finite input alphabet

qo→ Initial state of FA,qo, qo Q

F→ set of final states, F c Q

d→ Transition function called as state function mapping

Q * ∑ → Q

i.e. d= Q*∑→ Q

A FA is called deterministic (DFA) if from every vertex of its transition graph, there is an unique input symbol which takes the vertex state to the required next state.

DFA is constructed directly from an augmented regular expression (r)#. We begin by constructing a syntax tree T for (r) # and then we compute four functions Nullable, Firstpos, Lastpos and Followpos. The functions Nullable, Firstpos, Lastpos are defined on the nodes of a syntax tree and are used to compute Followpos which is defined on set of positions. We can short circuit the construction of NFA by building the DFA whose states correspond to the sets of positions in the tree. Positions, in particular , encode the information regarding when one position can follow another. Each symbol in an input string to a DFA can be matched by certain positions. An input symbol 'c' can only be matched by positions at which there is a 'c' but not every position with a 'c' can necessarily match a particular occurrences of 'c' in input stream.

6.Algorithm

The steps in algorithm are

1. Accept the given regular expression with end of character as #
2. Covert the regular expressions to its equivalent postfix form manually. (students need not write the code for converting infix to postfix but, they can directly accept postfix form of the infix expression)

3. Construct a syntax tree from the postfix expression obtained in step 2.
4. Assign positions to leaf nodes
5. Compute following functions.
   Nullable, Firstpos, Lastpos, Followpos

Firstpos, Lastpos, Nullable

- To evaluate followpos, we need three more functions to be defined for the nodes (not just for leaves) of the syntax tree.
- **Firstpos(n)** : The set of the positions of the first symbols of strings generated by the sub-expression rooted by n.
- **Lastpos(n)** : The set of the positions of the last symbols of strings generated by the sub-expression rooted by n.
- **Nullable(n)** : true If the empty string is a member of strings generated by the sub-expression rooted by n// false otherwise

<u>Computation of Nullables :</u>All nodes except the * nodes are not nullable. Also if some leaf node is for έ, then it is also nullable.

<u>Computation of Followpos :</u>Followpos(i) tells us what positions can follow positioniin the syntax tree. This can be computed as follows.

1. if n is a '.' (cat) Node, with a left child C1 and right child C2 and i is a position in the Lastpos(C1), then all positions in Firstpos(C2) are in Followpos(i)
2. if n is a * (closure) Node and i is a position in the Lastpos(n), then all positions in Firstpos(n) are Followpos(i)

6. Construct DFA from Follow Pos**.**

Note :*Step 5 can be done during construction of tree, since you are building the tree from bottom to top, and when computations at some root of sub tree are to be done, information of sub tree is available. So no need to do any traversal.*

Data Structures:

***7.*Node Structure for Parse Tree**

{

Leftchild and Rightchild : pointers to the node structure

Nullable : Boolean Type

Data : Character Type

Fistpos and Lastpos : set of integers

Pos : integer (this may or may not be part of tree node)

}

Stack : Stack is required to build the tree. This can be implemented either using link list (preferable) or as an array. Item or data that will be pushed into or popped out from stack is pointer to the node structure of a Parse Tree and not just a single character..

8.Computations of Firstpos and Lastpos.

Node n	Nullable(n)	Firstpos(n)	Lastpos(n)
N is a leaf labeled ἐ	**true**	ø	ø
N (|) with children C1, C2	Nullable(c1) **or** Nullable (c2)	Firstpos (C1) U Firstpos(C2)	Lastpos (C1) U Lasttpos(C2)
N is a leaf labeled with position i	**false**	{ i }	{ i }
n (•) with children C1, C2	Nullable(c1) **and** Nullable (c2)	If nullable (C1) then Firstpos (C1) U Firstpos(C2) else Firstpos(C1)	If nullable (C2) then Lastpos (C1) U Lastpos(C2) else Lastpos(C2)
n (*) with child C1	**true**	Firstpos (C1)	Lastpos (C1)

9.Algorithm for construction of DFA transition table

1. Initially , the only unmarked state in Dstates is firstpos(root), where root is the root of a syntax tree.

2. While there is an unmarked state T in Dstates do

   Begin

   Mark T

   For each input symbol a do

   Begin

   Let U be the set of positions that are in Followpos(P) for some P in T,

   such that the symbol at position P is a.

   If U is not empty and is not in Dstates then add U as an unmarked state to Dstates

Dtran [T,a] = U

End

End

Sample Input :

Enter postfix form of the regular expression (r.#) and use dot operator for concatenation and '|', for OR and * for closure. For Example (r. #) is : (a| b)*. a .b .b .# then equivalent Postfix Expression is ab|*a.b.b.#.

Sample output :

Post Order Traversal Traversal of the tree is ab|*a.b.b.#.

Show the Values Firstpos, Lastpos and the values of Positions for leaf nodes.

Node	Position	Nullable	Firstpositon	Lastposition
a	1	0	{1}	{1}
b	2	0	{2}	{2}
| (or)	0	0	{1,2}	{1,2}
* (closure)	0	1	{1,2}	{1,2}
a	3	0	{3}	{3}
. (cat)	0	0	{1,2,,3}	{3}
b	4	0	{4}	{4}
. (cat)	0	0	{1,2,,3}	{4}
b	5	0	{5}	{5}

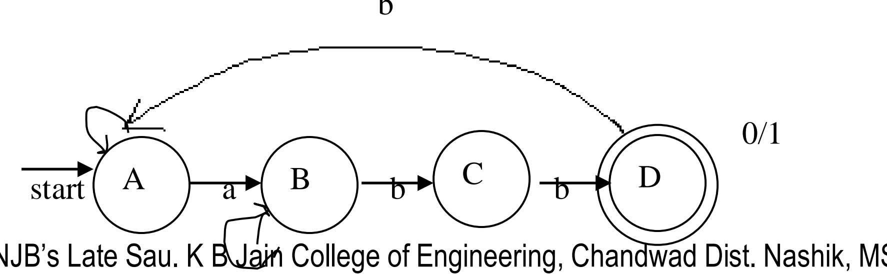
. (cat)	0	0	{1,2,,3}	{5}
#	6	0	{6}	{6}
.	0	0	{1,2,,3}	{6}

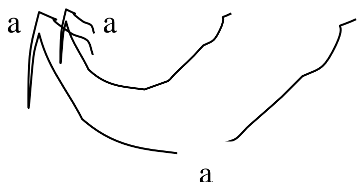
Follow Positions

Leaf Node position	Symbol at that position	Follow Positions
1	a	{1,2,3}
2	b	{1,2,3}
3	a	{4}
4	b	{5}
5	b	{6}

DFA:

	Input Symbol	
State	a	b
A {1,2,3}	B	A
B (1,2,3,4}	B	C
C {1,2,3,5}	B	D
D {1,2,3,6}	B	A





Instructions to the students for testing:

Test your program for following regular expressions

i) (a|b)*

ii) (a*|b*)*

iii) (a|b)*.a.b.b.(a|b)*

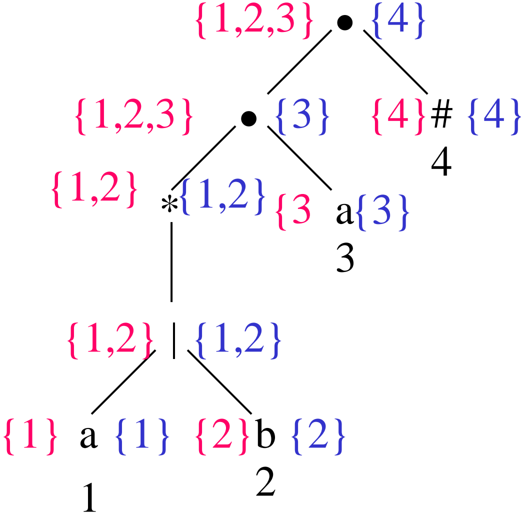
How to evaluate followpos

Two-rules define the function followpos:

1. If n is concatenation-node with left child c1 and right child c2,and i is a position in lastpos(c1), then all positions in firstpos(c2) are in followpos(i).
2. If n is a star-node, and i is a position in lastpos(n), then all positions in firstpos(n) are in followpos(i).

- If firstpos and lastpos have been computed for each node, followpos of each position can be computed by making one depth-first traversal of the syntax tree.

Example -- (a | b) * a #



Then we can calculate followpos

followpos(1) = {1,2,3}

followpos(2) = {1,2,3}

followpos(3) = {4}

followpos(4) = {}

After we calculate follow positions, we are ready to create DFA for the regular expression.

Input: Regular Expression

Output: Deterministic Finite Automata.

Conclusion:

Thus we have studied the Implementation of RE TO DFA.

FAQ'S:

1. What are the ways by which we can convert regular expression to DFA?
2. What is the need of finding first & last position of a node?
3. What are the rules for finding nullable nodes?
4. How to convert a regular expression to postfix expression?
5. How to calculate follow position of a node?
6. Which is the initial state (q0) in DFA?
7. How to construct DFA?

Regularity	Content	Viva-voce	Timely Submission	Total	Dated Sign of Subject Teacher
2	4	2	2	10	

Date of Performance:....................................Expected Date of Completion:........................

Actual Date of Completion:..

Assignment No:5

Title of the Assignment: ASSIGNMENT BASED ON LEX TOOL.

Problem Statement: Write a program to implement calculator using LEX and YACC.

Objective: To understand the process of lexical analysis and parsing through tools such as Lex and YACC.

Prerequisite: Basic Concept of lex and Yacc

Theory:

1.Introduction:Lexical analysis:

- The action of scanning the source program into proper syntactic classes is known as lexical analysis.
- Scanner / Lexical Analyzer scan the program and converts it into basic elements or tokens of the language.
- The goals of parsing are to check the validity of a source string & to determine its syntactic structure.
- For an invalid string the parser issues diagnostic messages reporting the cause & nature of errors in the string.
- For a valid string it builds a parse tree to reflect sequence of derivations or reduction performed. The parse tree is passed to the subsequent phase of compiler.

Bottom up Parsing:

- A bottom up parser constructs a parse tree for a source string through a sequence of reductions.
- The source string is valid if it can be reduced to ’S’, the distinguished symbol of grammar G.
- If not an error is indicated and reported during the process of reduction.

Top down Parsing:

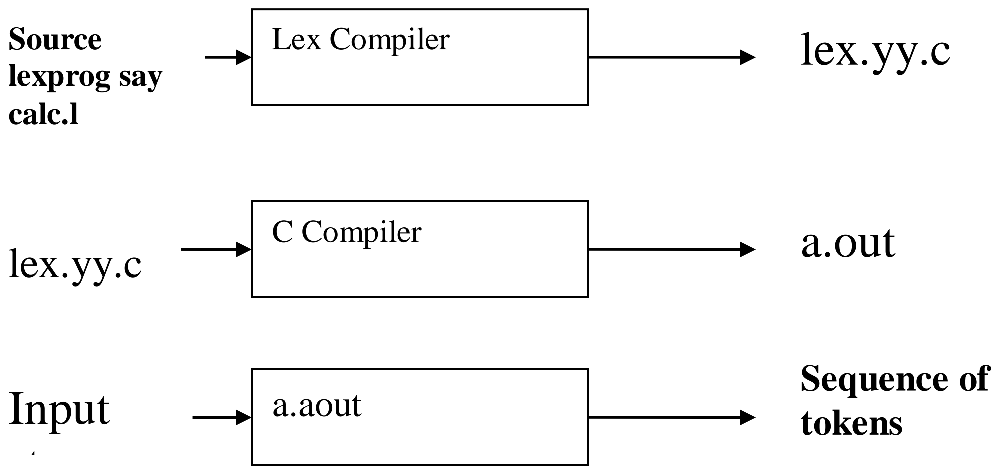
The top down parsing according to the grammar G tries to derive the matching source string through the sequence of derivations, starting with distinguished symbol of G.

Shift Reduce parser

- This method of parsing is bottom up parsing, which attempts to build the parse for an input string beginning at the leaves (bottom) and working up towards the root (up).
- In this process a string w is reduced to the start symbol of the Grammar.
- At each step a string matching the right side of the production is replaced by the symbol on the left.
- Each replacement of the right hand side of a production by left side in the process is called as reduction.

2.Lex:

- Several tools have been built for constructing lexical analyzers from special purpose notations based on regular expressions.
- Several algorithms exist for compiling regular expressions into pattern matching algorithms Lex is a tool which uses such algorithm
- Lex is a tool widely used to specify lexical analyzers for variety of languages.
- The tool shows how the speciation of patterns using regular expression can be combined with actions that LA may be required to perform.



- Specification of Lexical Analyzer is prepared by creating a program lex.l in Lex language.
- Lex.1 is run through Lex compilers to produce lex.yy.c which consist of a tabular representation of transition diagram constructed from regular expression of Lex.1, together with a standard routine that uses the table to recognize lexemes actions associated with regular expressions in Lex.l are pieces of C code and are carried over directly to lex.yy.c finally lex.yy.c is run through C compiler to produce a.out which is the lexical analyzer that transforms an input stream into a sequence of tokens

3. Lex Specifications:

- Lex program consists of 3 parts

Declarations

% %

Transition Rules

% %

Auxiliary Procedures

- **Declaration Section Includes**

Declarations of variables manifest constants and regular expressions.

/* ***Example of Declaration*** */

```
% {

        #include "yy.tab.h"
        Extern int yylval;
% }

/*      Example of Regular Definitions      */

delim           [ \t \n ]
ws              [ delim ] +
letter          [a-z to A-Z]
digit           [0-9]
id              {letter} ( {letter} / {digit} )*
number          {digit} + (\.{digit}+) ? (E [+\-] ? {digit} +) ?

%%    /*      Transiotion Rules      */



{ws}            { /* No action, no return */ }
{number}        { yylval = install – num (); return (NUMBER); }
{id}            {yylval = install – id (); return (ID); }

%%    /*      Auxiliary Procedures  */
install_id ()
{

        /* procedure to install identifier

        */
}
install_num ()
{
        /* procedure to install literals */

}
```

Explanation

Declarations are surrounded by special brackets % { and % }. Anything appearing between these brackets is copied directly into lex.yy.c & is not treated as a part of regular definitions or translation rules. Same is applied to the auxiliary procedures in the third section & these procedures are copied into lex.yy.c

Regular Definitions Each such definitions consist of a name and a regular expression denoted by that name.

? →meta symbol – Zero or More occurrences of
\ . – decimal point , since '.' , itself represents character class of characters
[+\-] – slash is required before -, since – also represents range.

Translation Rules Structure of LA is such that it keeps trying to recognize tokens, until action associated with one found causes a return.

4.Pattern Matching Primitives

Regular expressions in lex are composed of metacharacters (Table 1). Pattern-matching examples are shown in Table 2. Within a character class normal operators lose their meaning. Two operators allowed in a character class are the hyphen ("-") and circumflex ("^"). When used between two characters the hyphen represents a range of characters. The circumflex, when used as the first character, negates the expression. If two patterns match the same string the longest match wins. In case both matches are the same length, then the first pattern listed is used.

Metacharacter	Matches
.	any character except newline
\n	Newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression

?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	**a** or **b**
(ab)+	one or more copies of **ab** (grouping)
"a+b"	literal "**a+b**" (C escapes still work)
[]	character class

Expression	Matches
abc	**Abc**
abc*	**ababcabccabccc ...**
abc+	**abc, abcc, abccc, abcccc, ...**
a(bc)+	**abc, abcbc, abcbcbc, ...**
a(bc)?	**a, abc**
[abc]	one of: **a, b, c**
[a-z]	any letter, a through z
[a\-z]	one of: **a, -, z**
[-az]	one of: **- a z**
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	Whitespace

[^ab]	anything except: a, b
[a^b]	a, ^, b
[a|b]	a, |, b
a|b	a, b

Table 2: Pattern Matching Examples

Name	Function
intyylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
intyywrap(void)	wrapup, return 1 if done, 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN condition	switch start condition
ECHO	write matched string

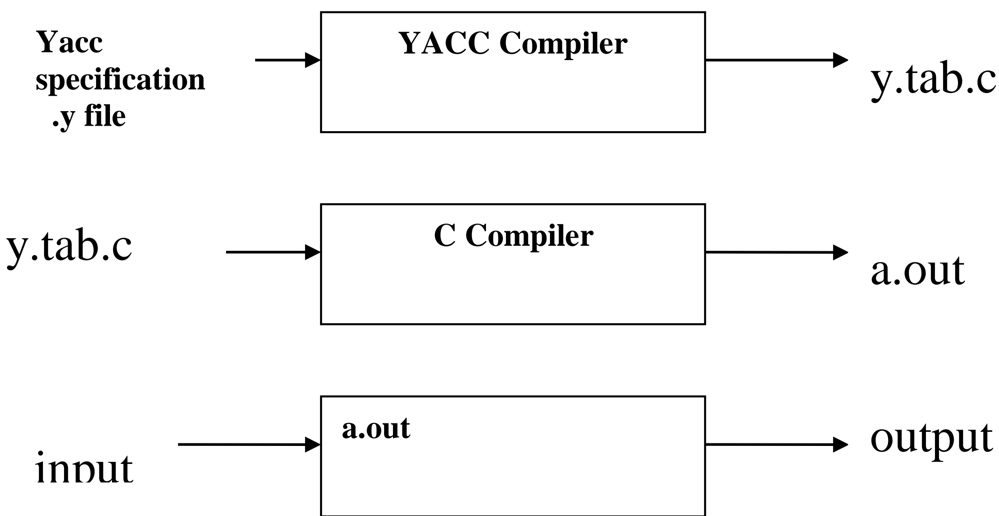
Table 3: Lex Predefined Variables

yylvalis a variable whose definition appears in Lex output lex.yy.c and which is also available to parser. Purpose of yylval is to hold the lexical value returned. 'return' statement can only return a code for token class.

Variable yytext corresponds to the pointer to the first character of lexeme &yyleng is integer telling how long the lexeme is.

5.Parser Generators :

- YACC : It is yet another compiler-compiler. Parser generator can be used to facilitate the construction of front end of the compiler.
- It makes use of shift reduce parser.
- Ambiguity in grammar can be resolved by specifying the operators and their associatively either left or right.
- The operators which appear on first line are of lowest priority while on next lines are higher precedence.



6.YACC source program has 3 parts

Declarations

% %

Transition rules

% %

supporting C routines

Declaration Part

1. Ordinary 'C' declarations within %{ %}
2. Token Declarations

Translation rules part → % % & then rules each translation rule consist of a grammar production & associated semantic action.

E.g. expr :expr '+' expr { $$ = $1+ $3; }
expr '-' expr {$$ = $1 - $3; }

;

1) ‘+’,’-‘ are quoted single characters of type ‘c’ and are treated as terminal symbols
2) Unquoted strings of letters and digits not declared to be tokens are taken as non-terminals –expr.
3) Alternative right sides can be separated by a vertical bar and a semicolon follows each left side with its alternatives & their semantic actions first left side is taken as start symbol.

YACC semantic action is a sequence of ‘C’ statements. In semantic action $$ refers to the value associated with non terminal on left and while $i refers to the value associated with ith grammar symbol [T \ NT] on right.

Supporting C – routines - other procedures such as error recovery routines may be provided.

7.Communication between Lex and YACC

yylex() produces pairs containing token and its associated value. If token NUMBER is return token NUMBER must be declared in first section of YACC. The attribute value associated with a token is communicated to the parser through YACC defined variable yylval.

calculator.l file

```
%{
#include "y.tab.h"
externintyylval;
%}
%%
[0-9]+ {yylval = atoi(yytext); return NUMBER; }
[ \t] ; /* ignore whitespace */
\n    return 0; /* logical EOF */
. returnyytext[0]; /* common catch all , return any character otherwise not handled as a
single charcter token to parser */
%%
```

calculator.y file

```
%{
/* this is yacc specification */
#include <stdio.h>
%}

%token NAME
%left '-' '+'
%left '*' '/'
%nonassoc UMINUS

%%
statement: NAME '=' exp
        | exp { printf("= %d\n",$1); }
        ;
exp: exp '+' exp { $$ = $1 + $3; }
  | exp '-' exp { $$ = $1 - $3; }
  | exp '*' exp { $$ = $1 * $3; }
  | exp '/' exp { $$ = $1 / $3; }
  | '-' exp %prec UMINUS { $$ = -$2; }
| '(' exp ')' { $$ = $2; }
  | NUMBER      { $$ = $1; }
  ;
%%
int main()
{
yyparse();
}
intyyerror()
{


}
intyywrap()
{
return 1;
```

}

8.Compiling and running simple calculator

[vvdmit@rehatvvdmit]$ yacc -d calculator.y // makes y.tab.c and y.tab.h

[vvdmit@rehatvvdmit]$ lexcalculator.l // makes lex.yy.c

[vvdmit@rehatvvdmit]$ cc -o calculator y.tab.clex.yy.c

[vvdmit@rehatvvdmit]$ ./calculator // compile and link files

(5+7)/(9-6)

= 4

[vvdmit@rehatvvdmit]$ ./calculator

-6+8*3

= 18

[vvdmit@rehatvvdmit]$

Explanation: Main function calls yyparse(), which is parser generated by yacc. Parser invokes yylex(), lexical analyzer for tokens. Lexical analyzer (Lex) returns token and value associated with it (NUMBER and yylval) to the parser. Parser is shift reduce parser which will reduce the expression and at the same time it will perform the semantic actions associated with it.

When yylex() reaches end of input, it calls yywrap() which returns a value 0 or 1. The default yywrap() always returns 1, indicating program is done and no more input. If value is 0, lex assumes that yywrap() has opened another file for it to read and continues to read from it.

9.Debugging Lex

- Lex has facilities that enable debugging.
- This feature may vary with different versions of lex so you should consult documentation for details.
- The code generated by lex in file **`lex.yy.c`** includes debugging statements that are enabled by specifying command-line option "**`-d`**".

10.DebuggingYacc

- Yacc has facilities that enable debugging.

- This feature may vary with different versions of yacc so be sure to consult documentation for details.
- The code generated by yacc in file **y.tab.c** includes debugging statements that are enabled by defining **YYDEBUG** and setting it to a non-zero value.
- This may also be done by specifying command-line option "**-t**". With **YYDEBUG** properly set, debug output may be toggled on and off by setting **yydebug**.
- Output includes tokens scanned and shift/ reduce actions.

11.FAQs

1. What are various phases of compiler?
2. Explain the task of lexical and syntax analysis.
3. Explain three sections of Lex tool.
4. Explain three sections of Yacc tool.
5. Explain terms yylval, yytext.
6. Explain the communication between lex and yacc.
7. What is meant by ambiguous grammar?
8. Explain use of RE in lexical analysis with the help of example.
9. What are types of grammar?
10. Explain context free grammar and the terminology used in it with example.
11. What is shift reduce conflict?
12. Compare top down and bottom up parsers.
13. When do you say that grammar is left recursive? How to remove left recursion ?
14. What is meant by left factoring and when it is used ?
15. State disadvantages of top down parsing ?
16. Explain how backtracking is eliminated in top down parsing ?

Regularity	Content	Viva-voce	Timely Submission	Total	Dated Sign of Subject Teacher
2	4	2	2	10	

Date of Performance:....................................Expected Date of Completion:.......................

Actual Date of Completion:...

Assignment No:6

Title of the Assignment:ASSIGNMENT BASED ON LEX TOOL.

Problem Statement: Intermediate Code generation using LEX and YACC for control Flow & Switch Case Statements.

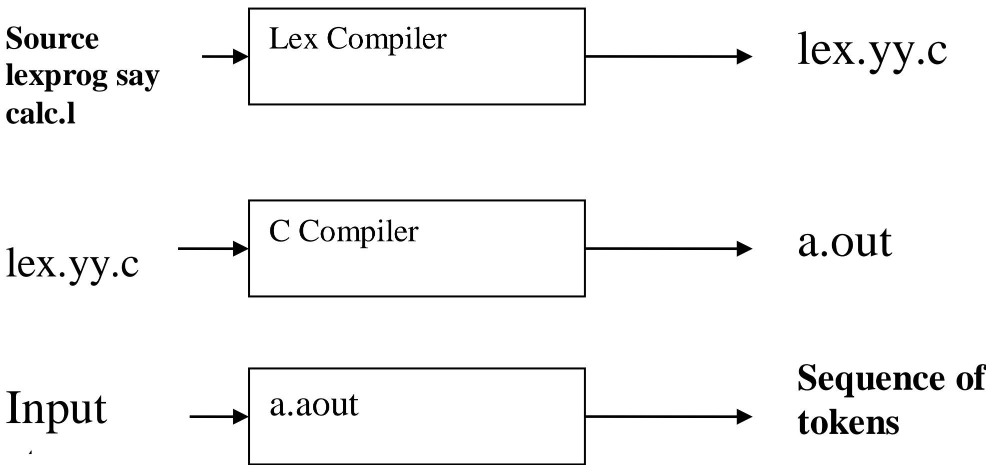
Objective: To understand the process of lexical analysis and parsing through tools such as Lex and YACC.

Prerequisite: Basic Concept of lex and Yacc

Theory:

1.Introduction:Lex:

- Several tools have been built for constructing lexical analyzers from special purpose notations based on regular expressions.
- Several algorithms exist for compiling regular expressions into pattern matching algorithms Lex is a tool which uses such algorithm
- Lex is a tool widely used to specify lexical analyzers for variety of languages.
- The tool shows how the speciation of patterns using regular expression can be combined with actions that LA may be required to perform.



- Specification of Lexical Analyzer is prepared by creating a program lex.l in Lex language.
- Lex.1 is run through Lex compilers to produce lex.yy.c which consist of a tabular representation of transition diagram constructed from regular expression of Lex.1, together with a standard routine that uses the table to recognize lexemes actions associated with regular expressions in Lex.l are pieces of C code and are carried over directly to lex.yy.c finally lex.yy.c is run through C compiler to produce a.out which is the lexical analyzer that transforms an input stream into a sequence of tokens

Lex Specifications:

- Lex program consists of 3 parts

Declarations

% %

Transition Rules

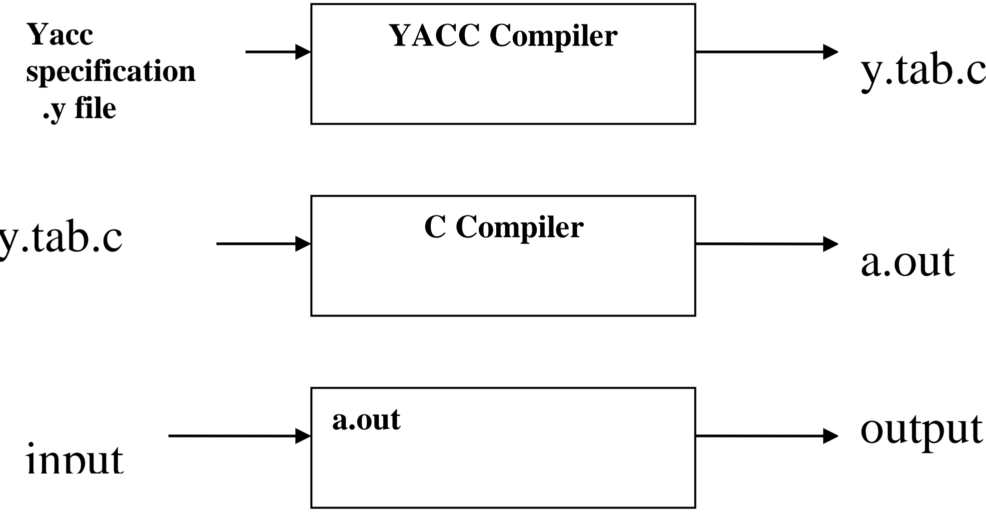
% %

Auxiliary Procedures

2.Parser Generators :

- YACC : It is yet another compiler-compiler. Parser generator can be used to facilitate the construction of front end of the compiler.
- It makes use of shift reduce parser.

- Ambiguity in grammar can be resolved by specifying the operators and their associatively either left or right.
- The operators which appear on first line are of lowest priority while on next lines are higher precedence.



YACC source program has 3 parts

Declarations

% %

Transition rules

% %

supporting C routines

2. Intermediate Code:

The task of compiler is to convert source program into machine program. This activity can be directly, but it is not always possible to generate such a machine code directly in one pass. Then, typically compilers generate am easy to represent form of source language which is called **intermediate language.** The generation of an intermediate language leads to efficient code generation.

2.1 Intermediate Languages:

There are mainly three types of intermediate code representation

- Syntax tree
- POSIX
- Three address code

2.1.1.Syntax Tree:

The natural hierarchical structure is represented by syntax trees. Directed Acyclic Graph or DAG is very much similar to syntax trees but they are in more compact form.
The code being generated as intermediate should be such that the remaining processing of the subsequent phases should be easy.

2.1.2.Posix

The posix notation is used using Postfix representation.
Consider that the input expression is x=-a*b+-a*b then required postfix form is
xa – b*a – b*+=
Basically, the linearization of syntax tree is posixnotation. In this representation, the operator can be easily associated with the corresponding operands. This is the most natural way of representation in expression evaluation.

2.1.3.Three address code

In three address code forms**at the most three addresses are used** to represent aystatement. The general form of three address code representation is

a =b op c

wherea,b or c are the operands that can be names,constants,compiler generated temporaries and op represents the operator.The operators can be fixed or floating point arithmetic operator or logical operators on Boolean valued data. Only single operation at right side of the expression is allowed at a time.

2.2. Implementation of three address code

Three address code is an abstract form of an intermediate code that can be implemented as a record with the ,addressfields.There are three representations used for three address code such as quadruples,triples and indirect triples.

2.2.1.Quadruple representation

The quadruple representation is a structure with at most four fields such as op,arg1,arg2,result.The op field is used to represent the two operands used and result field is used to store the result of an expression.
For example:Consider the input statement x=-a*b+-a*b

Quadrauple

	Op	**Arg1**	**Arg2**	**Result**

(0)	**uminus**	**a**		**t1**
(1)	*	**t1**	**b**	**t2**
(2)	**uminus**	**a**		**t3**
(3)	*	**t3**	**b**	**t4**
(4)	+	**t2**	**t4**	**t5**
(5)	=	**t5**		**x**

The three address code is

t1=uminus a

t2=t1*b

t3=-a

t4=t3*b

t5=t2+t4

x=t5

2.2.1.Triples

In the triple representation the use of temporary variable is avoided by referring the pointers in the symbol table.

For the expression x=-a*b+-a*b the triple representation is given below.

Number	Op	Arg1	Arg2
(0)	uminus	A	
(1)	*	(0)	b
(2)	uminus	A	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	X	(4)

2.2.3. Indirect Triples

In the indirect triple representation the listing of triples is been done. And listing pointers are used instead of using statements.

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(11)	b

(2)	uminus	a	
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	=	x	(15)

	Statement
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

3. Generation of three address code

The translation sceme for various language constructs can be generated by using the appropriate semantic actions.

3.1.Assignment statements

The assignment statement mainly deals with the expressions. The expression can be of type integer,real,array and record.

3.2.Boolean expressions

Normally there are two types of Boolean expressions used

1.for computing the logical values

2.In conditional expressions using if-then-else or while-do

3.3 Flow of Control Statements

In this section we will discuss the translation of Boolean expression into three address code.The control statements are if-then-else and while-do.Thegrammer for such statements is as shown below.

S->if E then S1

| if E then S1 else S2

| while E do S1

While generating three address code-

- To generate new symbolic label the function new_label() is used.
- With the expression E.true and E.false are the labels associated.
- S.code and E.code is for generating three address code.

3.4.Case Statements

The syntax for switch case statement can be as shown below.

Switch expression

{

Case value:statement

Case value:statement

....

Case value:statement

Default:statement

}