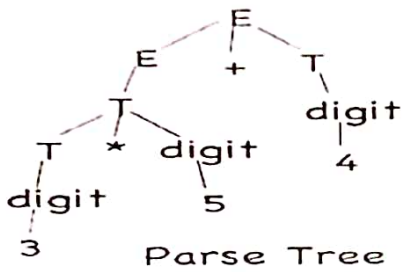
EXPERIMENT NO: 05

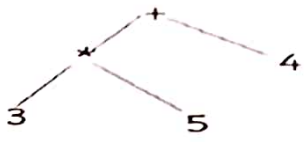
Title: Program to create a syntax tree for simple expression in C language using Recursive descent parsing techniques
Aim: Program to create a syntax tree for simple expression in C language using Recursive descent parsing techniques

Theory:

A syntax tree is a condensed form of parse tree.



Parse Tree



Syntax Tree

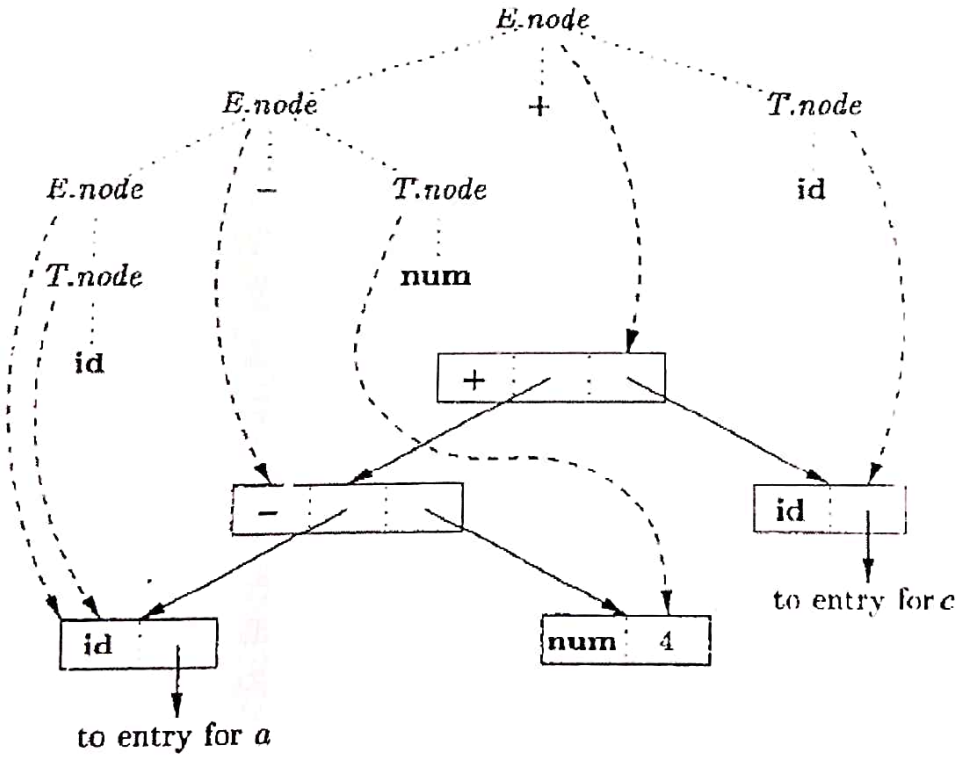
• Syntax trees are useful for representing programming language constructs like expressions and statements.
• They help compiler design by decoupling parsing from translation.
• Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.
• e.g. a syntax-tree node representing an expression E1 + E2 has label + and two children representing the sub expressions E1 and E2
• Each node is implemented by objects with suitable number of fields; each object will have an op field that is the label of the node with additional fields as follows:
——————————— If the node is a leaf, an additional field holds the lexical value for the
leaf . This is created by function Leaf(op, val)
——————————— If the node is an interior node, there are as many fields as the node has
children in the syntax tree. This is created by function Node(op, c1, c2,....ck) .

Example: The S-attributed definition in figure below constructs syntax trees for a simple expression grammar involving only the binary operators + and -. As usual, these operators are at

the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute node, which represents a node of the syntax
tree.

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	*E.node* = **new** *Node* ('+', E_1.*node*, *T.node*)
2) $E \rightarrow E_1 - T$	*E.node* = **new** *Node* ('–', E_1.*node*, *T.node*)
3) $E \rightarrow T$	*E.node* = *T.node*
4) $T \rightarrow (E)$	*E.node* = *T.node*
5) $T \rightarrow$ **id**	*T.node* = **new** *Leaf* (**id**, **id**.*entry*)
6) $T \rightarrow$ **num**	*T.node* = **new** *Leaf* (**num**, **num**.*val*)

Syntax tree for a-4+c using the above SDD is shown below.



Steps in the construction of the syntax tree for a-4+c

If the rules are evaluated during a post order traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown below ends with p5 pointing to the root of the constructed syntax tree.

1) p_1 = **new** *Leaf* (**id**, *entry-a*);
2) p_2 = **new** *Leaf* (**num**, 4);
3) p_3 = **new** *Node* ('−', p_1, p_2);
4) p_4 = **new** *Leaf* (**id**, *entry-c*);
5) p_5 = **new** *Node* ('+', p_3, p_4);

Sample Questions:

1. **Explain Syntax Directed definition**
2. **What is Synthesized Attribute?**
3. **What is L-Attributed Definition?**

EXPERIMENT NO: 06

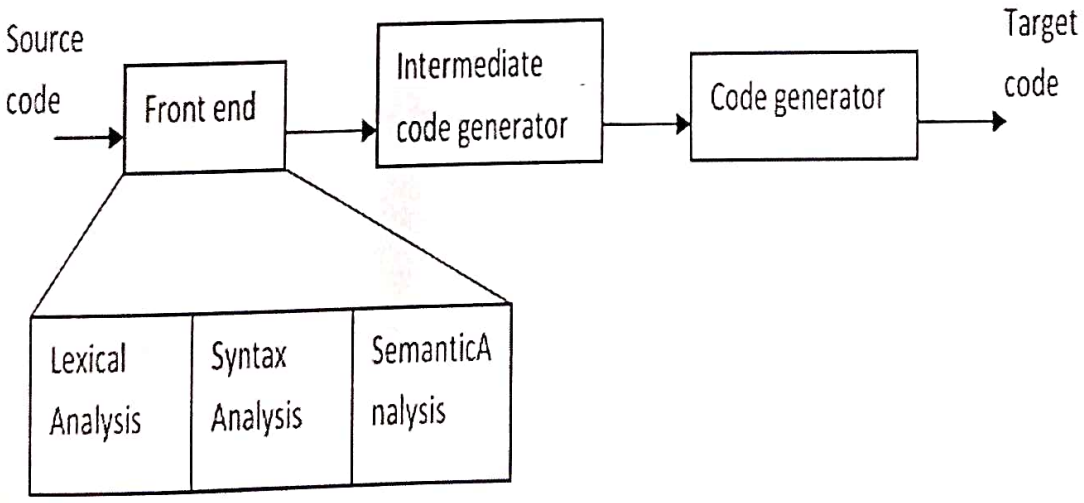
Title: Implement intermediate code generator for the Boolean expression in three Address code format.
Aim: Implement intermediate code generator for the Boolean expression in three Address code format.

Theory:

In the first pass of the compiler, source program is converted into intermediate code. The second pass converts the intermediate code to target code. The intermediate code generation is done by intermediate code generation phase. It takes input from front end which consists of lexical analysis, syntax analysis and semantic analysis and generates intermediate code and gives it to code generator. Fig. shows the position of intermediate code generator in compiler. Although some source code can be directly converted to target code, there are some advantages of intermediate code. Some of these advantages are:

a. Target code can be generated to any machine just by attaching new machine as the back end. This is called retargeting.

b. It is possible to apply machine independent code optimization. This helps in faster generation of code.



Three address code

Most instruction of three address code is of the form
a = b op c
where b and c are operands and op is an operator. The result after applying operator op on b and c is stored in a. Operator op can be like +, -, * or ÷. Here operator op is assumed as binary operator. The operands b and c represents the address in memory or can be constants or literal value with no runtime address. Result a can be address in memory or temporary variable.

Example: a = b * c + 10
The three address code will be
t1 = b * C
t2 = t1 + 10
a = t2
Here t1 and t2 are temporary variables used to store the intermediate result.

Types of three address code

There are different types of statements in source program to which three address code has to be generated. Along with operands and operators, three address code also use labels to provide flow of control for statements like if-then-else, for and while. The different types of three address code statements are:

i. Assignment statement a = b op c

In the above case b and c are operands, while op is binary or logical operator. The result of applying op on b and c is stored in a.

ii. Unary operation a = op b
This is used for unary minus or logical negation.

Example: a = b * (- c) + d
Three address code for the above example will be
t1 = -c
t2 = t1 * b
t3 = t2 + d
a = t3

iii. Copy Statement a = b
The value of b is stored in variable a.

iv. Unconditional jump goto L
Creates label L and generates three-address code 'goto L'

v. Conditional jump if exp go to L
Creates label L, generate code for expression exp, If the exp returns value true then go to the statement labelled L. exp returns a value false go to the statement immediately following the if statement.

vi. Function call For a function fun with n arguments a1,a2,a3....an ie.,
fun(a1, a2, a3,...an),
the three address code will be
Param a1
Param a2
...
Param an
Call fun, n

Where param defines the arguments to function.

vii. Array indexing- In order to access the elements of array either single dimension or multidimension, three address code requires base address and offset value. Base address consists of the address of first element in an array. Other elements of the array can be accessed using the base address and offset value.

Example: x = y[i]
Memory location m = Base address of y + Displacement i
x = contents of memory location m
similarly x[i] = y
Memory location m = Base address of x + Displacement i
The value of y is stored in memory location m

viii. Pointer assignment x = &y x stores the address of memory location y
x = *y y is a pointer whose r-value is location
*x = y sets r-value of the object pointed by x to the r-value of y
Intermediate representation should have an operator set which is rich to implement most of the operations of source language. It should also help in mapping to restricted instruction set of target machine.

EXPERIMENT NO: 07

Title: Implement intermediate code generator for the conditional statements in three Address code format
Aim: Implement intermediate code generator for the conditional statements in three Address code format

Theory:

- The basic idea of converting any flow of control statement to a three address code is to simulate the "branching" of the flow of control.

- This is done by skipping to different parts of the code (label) to mimic the different flow of control branches.

- Flow of control statements may be converted to three address code by use of the following functions:-

  - ✓ newlabel – returns a new symbolic label each time it is called.
  - ✓ gen () – "generates" the code (string) passed as a parameter to it.

- The following attributes are associated with the non-terminals for the code generation:-

  - ✓ code – contains the generated three address code.
  - ✓ true – contains the label to which a jump takes place if the Boolean expression associated (if any) evaluates to "true".
  - ✓ false – contains the label to which a jump takes place if the Boolean expression (if any) associated evaluates to "false".
  - ✓ begin – contains the label / address pointing to the beginning of the code chunk for the statement "generated" (if any) by the non-terminal.

FOR LOOP	in 3 TA code

a=3;
b=4;
for(i=0;i<n;i++){
a=b+1;
a=a*a;
}
c=a;
	a=3;
	b=4;
	i=0;
	L1:
	VAR1=i<n;
	if(VAR1) goto L2;
	goto L3;
	L4: i++;
	goto L1;
	L2: VAR2=b+1;
	a=VAR2;
	VAR3=a*a;
	a=VAR3;
	goto L4
	L3: c=a;

- EXAMPLES:-
  Lets try converting the following c code

WHILE Loop	in 3 TA code

a=3;
b=4;
i=0;
while(i<n){
a=b+1;
a=a*a;
i++;
}
c=a;
	a=3;
	b=4;
	i=0;
	L1:
	VAR1=i<n;
	if(VAR1) goto L2;
	goto L3;
	L2: VAR2=b+1;
	a=VAR2;
	VAR3=a*a;
	a=VAR3;
	i++;
	goto L1
	L3: c=a;

DO WHILE Loop	**in 3 TA code**

a=3;
b=4;
i=0;
do{
a=b+1;
	a=3;
	b=4;
	i=0;
	L1:
	VAR2=b+1;

a=a*a;
i++;
}while(i<n);
c=a;
	a=VAR2;
	VAR3=a*a;
	a=VAR3;
	i++;
	
	VAR1=i<n;
	if(VAR1) goto L1;
	goto L2;
	
	L2: c=a;

Suppose we have the following grammar:-

S --> if E then S_1

S --> if E then S_1 else S_2

S --> while E do S_1

- SEMANTIC RULES:-

  - S --> if E then S_1

{

E.true := newlabel ;

E.false := S.next ;

S_1.next := S.next ;

S.code := E.code || gen(E.true ':') || S_1.code

}

- S --> if E then S_1 else S_2

{

E.true := newlabel ;

E.false := newlabel ;

S_1.next := S.next ;

S_2.next := S.next ;

S.code := E.code || gen(E.true ':') || S_1.code || gen('goto' S.next) || gen(E.false ':') || S_2.code

}

S --> while E do S_1

{

S.begin := newlabel ;

E.true := newlabel ;

E.false := S.next ;

S_1.next := S.begin ;

S.code := gen(S.begin ':') || E.code || gen(E.true ':') || S_1.code || gen('goto' S.begin)

}

Sample Questions:

1. Explain Three Address code

2. What is Semantic Rules?

EXPERIMENT NO: 08

Title: Design a LALR Bottom Up Parser for the given grammar
Aim: To Design and implement an LALR bottom up Parser for checking the syntax of the statements in the given language.

Theory:

- **Bottom-up parsers** build parse trees from the leaves and work up to the root.
- Bottom-up syntax analysis known as shift-reduce parsing.
- An easy-to-implement shift-reduce parser is called operator precedence parsing.
- General method of shift-reduce parsing is called LR parsing.
- Shift-reduce parsing attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
- At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production, and if the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse.

o Consider the grammar

S → aABe

A → Abc | b

B → d

The sentence abbcde can be reduced to S by the following steps.

abbcde

aAbcde

aAde

aABe

S

These reductions trace out the following rightmost derivation in reverse.

S aABe aAde aAbcde abbcde

- **Handles:**

o A handle of a string is a substring that matches the right side of a production, and whose reduction to the nonterminal on the left side of the production represents one step along the reverse of a rightmost derivation.

- **Precise definition of a handle:**

o A handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

o i.e., if $S \quad \alpha Aw \quad \alpha\beta w$, then $A \rightarrow \beta$ in the position following α is a handle of

αβw.

- The string w to the right of the handle contains only terminal symbols.
- In the example above, abbcde is a right sentential form whose handle is A → b at position 2. Likewise, aAbcde is a right sentential form whose handle is A → Abc at position 2.

Handle Pruning:

A rightmost derivation in reverse can be obtained by handle pruning.

i.e., start with a string of terminals w that is to parse. If w is a sentence of the grammar at hand, thenw = γ_n, where γ_n is the nth right sentential form of some as yet unknown rightmost derivation.

$S = \gamma_0 \quad \gamma_1 \quad \gamma_2 \quad \dots \quad \gamma_{n-1} \quad \gamma_n = w.$

Example for right sentential form and handle for grammar

E → E + E

E → E * E

E → (E)

E → id

RIGHT SENTENTIAL FORM	HANDLE	REDUCTION PRODUCTION
$id_1 + id_2 * id_3$	id_1	E → id
$E + id_2 * id_3$	id_2	E → id
$E + E * id_3$	id_3	E → id
E + E * E	E * E	E → E * E
E + E	E + E	E → E + E
E		

- **Shift Reduce Parsing**

Stack implementation of Shift-reduce parsing:

- There are two problems that must be solved to parse by handle pruning.
  - The first is to locate the substring to be reduced in a right sentential form.
  - The second is to determine what production to chose in case there is more than one production with that substring on the right side.
- The type of data structure to use in a shift reduces parser.
- **Implementation of Shift-Reduce Parser:**
  - To implement shift-reduce parser, use a stack to hold grammar symbols and an input buffer to hold the string w to be parsed.
  - Use $ to mark the bottom of the stack and also the right end of the input.
  - Initially the stack is empty, and the string w is on the input, as follows:

Stack	Input
$	w $

  - The parser operates by shifting zero or more input symbols onto the stack until a handle β is on top of the stack.

§ The parser then reduces β to the left side of the appropriate production.
§ The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

Stack	Input
$S	$

§ After entering this configuration, the parser halts and announces successful completion of parsing.
§ There are four possible actions that a shift-reduce parser can make: 1) shift 2) reduce 3) accept 4) error.

1. In a **shift** action, the next symbol is shifted onto the top of the stack.
2. In a **reduce** action, the parser knows the right end of the handle is at the top of the stack. It must then locate the left end of the handle within the stack and decide with what nonterminal to replace the handle.
3. In an **accept** action, the parser announces successful completion of parsing.
4. In an **error** action, the parser discovers that a syntax error has occurred and calls an error recovery routine.

Note: an important fact that justifies the use of a stack in shift-reduce parsing: the handle will always appear on top of the stack, and never inside.

§ **Example**

Consider the grammar

$E \rightarrow E + E$
$E \rightarrow E * E$
$E \rightarrow (E)$
$E \rightarrow id$

and the input string $id_1 + id_2 * id_3$. Use the shift-reduce parser to check whether the input string is accepted by the **above grammar**

Stack	Input	Action
$	$id_1 + id_2 * id_3$	shift
$id_1	$+ id_2 * id_3$	reduce by E→id
$E	$+ id_2 * id_3$	shift
$E +	$id_2 * id_3$	shift
$E + id_2	$* id_3$	reduce by E→id
$E + E	$* id_3$	shift
$E + E *	id_3	shift
$E + E * id_3	$	reduce by E→id
$E + E * E	$	reduce by E→E * E
$E + E	$	reduce by E→E + E
$E	$	accept

o **Viable Prefixes:**

§ The set of prefixes of right sentential forms that can appear on the stack of a shift-reduce parser are called **viable prefixes.**

o **Conflicts during shift-reduce parsing:**

§ There are CFGs for which shift-reduce parsing cannot be used.
§ For every shift-reduce parser for such grammar can reach a configuration in which the parser cannot decide whether to shift or to reduce (**a shift-reduce conflict**), or cannot decide which of

several reductions to make (**a reduce/reduce conflict**), by knowing the entire stack contents and the next input symbol.

§ **Example of such grammars:**

- These grammars are not LR(k) class grammars, refer them as no-LR grammars.
- The k in the LR(k) grammars refer to the number of symbols of lookahead on the input.
- Grammars used in compiling usually fall in the LR(1) class, with one symbol lookahead.
- An ambiguous grammar can never be LR.

Stmt → **if** expr **then** stmt
| **if** expr **then** stmt **else** stmt
| **other**

- In this grammar there is a shift/reduce conflict occur for some input string.
- So this is not LR(1) grammar.

ALGORITHM/PROCEDURE/CODE:

LALR Bottom Up Parser

```
<parser.l>
%{
#include<stdio.h>
#include "y.tab.h"
%}
%%
[0-9]+ {yylval.dval=atof(yytext);
return DIGIT;
}
\n|. return yytext[0];
%%
<parser.y>
%{
/*This YACC specification file generates the LALR parser for the program
considered in experiment 4.*/
#include<stdio.h>
%}
%union
{
double dval;
}
%token <dval> DIGIT
%type <dval> expr
%type <dval> term
%type <dval> factor
%%
```

```
line : expr '\n' {
;
printf("%g\n", $1);
}
expr : expr '+' term { $$=$1 + $3 ; }
| term
;
term: term '*' factor {$$=$1 * $3 ;}
| factor
;
factor: '(' expr ')' {$$=$2 ;}
| DIGIT
;
%%
39 | P a g e
int main()
{
Print(" Enter AE:");
yyparse();
}
yyerror(char *s)
{
printf("%s",s);
}
```

Output:

```
$ lex parser.l
$ yacc –d parser.y
$cc lex.yy.c y.tab.c –ll –lm
$./a.out
2+3
5.0000
```

Sample Questions:

1. What is yacc? Are there any other tools available for parser generation?
2. How do you use it?
3. Structure of parser specification program
4. How many ways we can generate the Parser

Lab Assignment:
1. Construct LALR parsing table to accept the arithmetic expressions of the C language.

EXPERIMENT NO: 09

Title: Implement code generator from a labeled tree.
Aim: Implement code generator from a labeled tree.

Theory:

Code Generation from a Labeled Tree:

The algorithm uses the recursive procedure gencode(n) to produce machine code evaluating the subtree of labeled tree T with root n into a register. The procedure gencode uses a stack rstack to allocate registers. Initially rstack contains all available registers. When gencode returns, it leaves the registers on rstack in the same order it found them. The resulting code computes the value of the tree T in the top register on rstack.

The function swap(rstack) interchanges the top two registers on rstack. The use of swap is to make sure that a left child and its parent are evaluated into the same register. The procedure gencode uses a stack tstack to allocate temporary memory locations.

The code-generation algorithm is to call gencode on the root of T. The gencode procedure can be explained by these cases:

Case 0: That is, n is a leaf and the leftmost child of its parent. Therefore we generate just a load instruction.

Case 1: we have a subtree of the form for which we generate code to evaluate n_1 into register R=top(rstack) followed by the instruction op name R.

Case 2: A subtree of the form where n_1 can be evaluated without stores but n_2 is harder to evaluate than n_1 as it requires more registers. For this case, swap the top two registers on rsatck, then evaluate n_2 into R=top(rstack).We remove R from rstack and evaluate n_1 into S = top(rstack). Then generate the instruction op R, S, which produce the value of n in register S. Another call to swap leaves rstack as it was when this call of gencode begins.

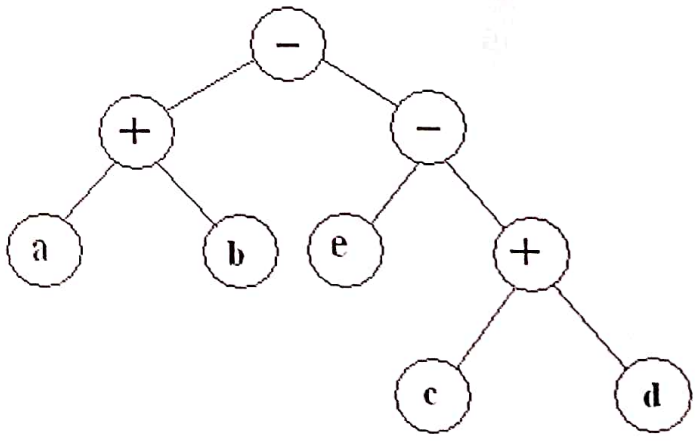
Case 3: It is similar to case 2 except that here the left subtree is harder and is evaluated first. There is no need to swap registers here.

Case 4: It occurs when both subtrees requires r or more registers to evaluate without stores. Since we must use a temporary memory location, we first evaluate the right subtree into the temporary T, then the left subtree, and finally the root.

```
Procedure gencode(n);
Begin
/* case 0 */
if n is a left leaf representing operand name and n is the leftmost child of its parent then
        print 'MOV' || name || '.' || top(rstack)
else if n is an interior node with operator op, left child n1, and right child n2 then
/* case 1 */
        if label(n2) = 0 then begin
              let name be the operand represented by n2;
              gencode(n1);
              print op || name || '.' || top(rsatck)
        end
 /* case 2 */
        else if 1 ≤ label (n1) < label(n2) and label(n1) < r then begin
              swap(rsatck);
              gencode(n2 );
              R := pop(rstack); /* n2  was evaluated into register R */
              gencode(n1);
              print op || R || '.' || top(rstack);
              push(rstack,R);
              swap(rstack)
         end
  /* case 3 */
        else if 1 ≤ label (n2) < label(n1) and label(n2) < r then begin
              gencode(n1);
```

```
        R := pop(rstack); /* n1 was evaluated into register R */
        gencode(n2);
        print op || R || '.' || top(rstack);
        push(rstack,R);
      end
/* case 4, both labels ≥ r, the total number of registers */
    else begin
        gencode(n2 );
        T := pop(tstack);
        Print 'MOV' || top(rstack) || '.' || T;
        gencode(n1);
        push(rstack,R);
        print op || T || '.' || top(rstack)
    end
end
```

Example:



The code for the labled tree shown in Fig with rstack = R0 , R1 initially. The sequence of calls to gencode and code printing steps are given as:

```
gencode(t4)
    gencode(t3)
        gencode(e)
            print MOV e, R1
        gencode(t2)
            gencode(c)
                print MOV c, R0
            print SUB r0, R1
    gencode(t1)
        gencode(a)
            print MOV a, R0
        print ADD b, R0
    print SUB R1, R0
```

Sample Questions:

1. What is Labeled Tree?
2. What Code Generation?
3. Explain Procedure of Code Generation from a Labeled Tree.

EXPERIMENT NO: 10

Title: Demonstration of compiler and interpreter using Lex and Yacc.
Aim: Demonstration of compiler and interpreter using Lex and Yacc.

Theory:

A compiler or an interpreter performs its task in 3 stages:

1) Lexical Analysis: Lexical analyzer: scans the input stream and converts sequences of characters into tokens. Token: a classification of groups of characters.

Examples:

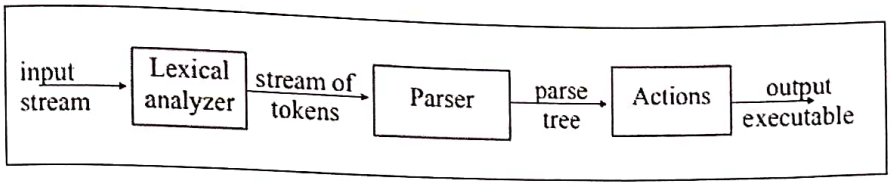
Lexeme	**Token**
Sum	ID
For	FOR
=	ASSIGN_OP
==	EQUAL_OP
57	INTEGER_CONST
*	MULT_OP
,	COMMA
(	LEFT_PAREN

Lex is a tool for writing lexical analyzers.

2) Syntactic Analysis (Parsing): Parser: reads tokens and assembles them into language constructs using the grammar rules of the language.

Yacc is a tool for constructing parsers.

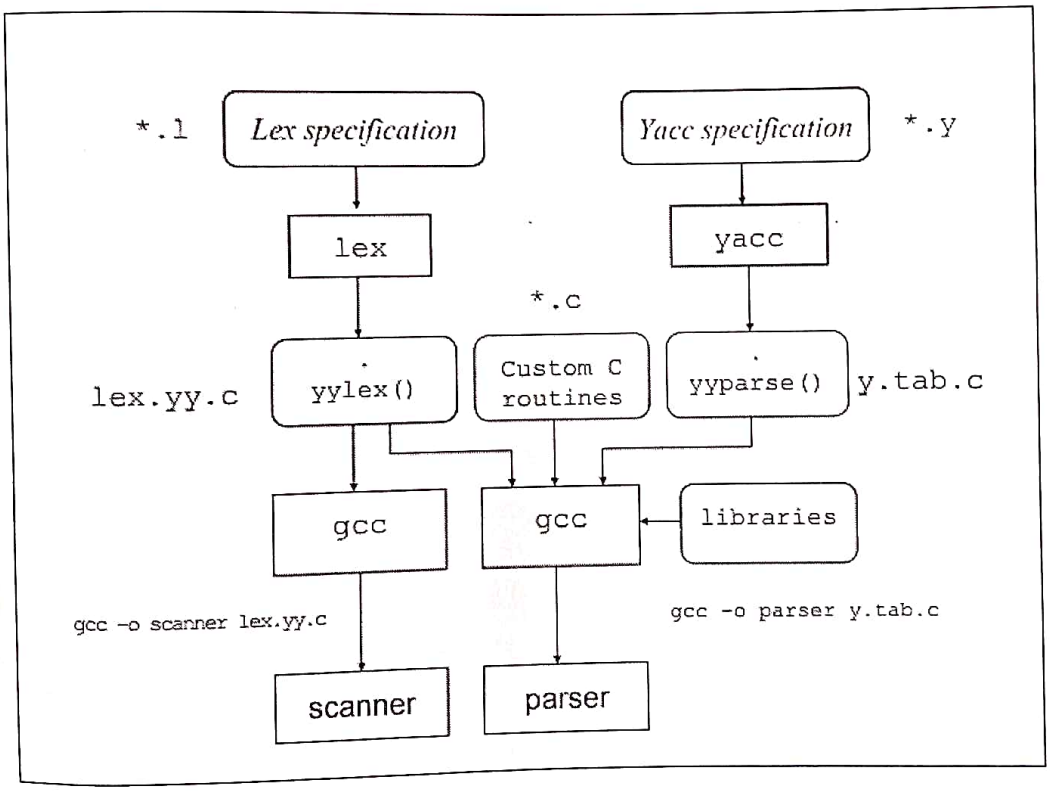
3) Actions: Acting upon input is done by code supplied by the compiler writer. Basic model of parsing for interpreters and compilers:



Lex: reads a specification file containing regular expressions and generates a C routine that performs lexical analysis. Matches sequences that identify tokens.

Yacc: reads a specification file that codifies the grammar of a language and generates a parsing routine.

Using lex and yacc tools:



Sample Questions:

1. **What is are the functions of a Scanner?**
2. **What is Token?**
3. **What is lexeme, Pattern?**

4. What is purpose of Lex?
5. What are the other tools used in Lexical Analysis?

Lab Assignment:

1) Write a LEX specification program for the tokens of C language.
2) Execute the above LEX file using any LEX tools
3) Generate tokens of a simple C program