CD - Unit - 3

(a) Introduction to Semantic Analysis :-

→ This phase checks the source program for semantic errors.
→ It uses the hierarchical structure determined by the syntax analysis phase to identify the operators and operands of expressions and statements.
→ It performs type checking, i.e, it checks that whether each operator has operands that are permitted by the source language specification.

Ex:-
- If a real number is used to index an array like a[1.5] then the compiler will report an error. This error is handled during Semantic analysis.

(b) Semantic errors :-

- undeclared names (or) multiple declaration of a variable.
- Type incompatibilities.

→ Semantic errors can be detected both at compile time and at runtime.
→ The errors will be of declaration (or) scope of variables
For example, undeclared (or) multiply-declared identifiers
→ leaving an identifier undeclared gives the "undeclared identifier" error message unless it is declared properly.
→ Type incompatibilities between operators and operands and between formal and actual parameters are another common source of semantic errors that can be detected at compile time.

(c) Intermediate forms of Source Programs :-

→ A compiler while translating a source program into a functionally equivalent object code representation, may first generate an intermediate representation.

Position of Intermediate code generator

→ The Syntax-directed methods are used to translate into an intermediate form programming language constructs such as,

declarations,
assignments and
flow-of-control statements.

→ The Intermediate code representations are of three types.

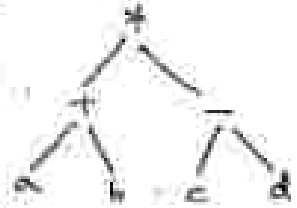
(i) Abstract Syntax Trees
(ii) Polish Notations
(iii) Three address code

→ These three representations are used to represent the intermediate languages.

(i) Abstract Syntax Trees:-

→ The natural hierarchical structure is represented by syntax trees.

Ex:- (a+b) * (c-d)

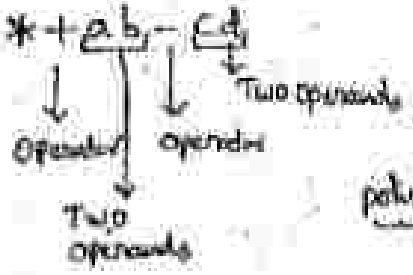


Abstract Syntax tree

(ii) Polish Notations:-

→ Basically, the linearization of Syntax tree is polish notation.
→ In this representation, the operator can be easily associated with the corresponding operands.
→ This is the most natural way of representation in expression evaluation.
→ The polish notation is also called as prefix notation in which the operator occurs first and then operands are arranged.

Ex:- (a+b) * (c-d)



polish notation

(iii) Three address code:-

→ In the three address code form at the most three addresses are used to represent any statement.

→ The general form of three address code representation is,

$$a := b \ op \ c$$

- where a, b, c are the operands that can be names, constants, compiler generated temporaries and
- "op" represents the operator.

Ex:- (a+b) * (c-d)

Three address code

$t_1 := a + b$
$t_2 := c - d$
$t_3 := t_1 * t_2$

Implementation of three address statements:-

→ Implementation of three address statements are of three types.

a) Quadruples
b) Triples
c) Indirect Triples

a) Quadruples:-

→ A quadruple is a record structure with four fields op, arg1, arg2 and result.

Ex:- (a+b) * (c-d)

	op	arg1	arg2	result
(0)	+	a	b	t_1
(1)	-	c	d	t_2
(2)	*	t_1	t_2	t_3

→ The contents of fields arg1, arg2 and result are normally pointers to the symbol table entries for the names represented by these fields.

→ If so, temporary names must be entered into the symbol table as they are created.

b) Triples:-

→ A triple is a record structure with three fields op, arg1 and arg2.

Ex:- (a+b) * (c-d)

	op	arg1	arg2
(0)	+	a	b
(1)	-	c	d
(2)	*	(0)	(1)

→ The fields arg1 and arg2 are either pointers to the symbol table (or) pointers to the tuple structure.

→ This method is used to avoid entering temporary names into the symbol table.

→ Temporary value is referred by the position of the statement that computes it.

c) Indirect Triples :-

→ Listing pointers to triples rather than listing the triples themselves are called Indirect Triples.

Ex:- (a+b) * (c-d)

	Statement
(0)	(10)
(1)	(11)
(2)	(12)

	op	arg1	arg2
(10)	+	a	b
(11)	-	c	d
(12)	*	(10)	(11)

) Direct Acyclic Graphs:- (DAG)

→ A DAG is a directed graph with no cycles which gives a picture of how the value computed by each statement in a basic block is used in subsequent statements in the block.

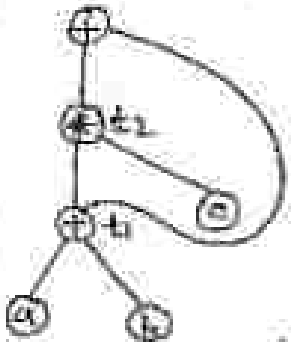
→ The DAG for an expression identifies the common subexpressions in the expression.

Ex:- (a+b)*c + (a+b)

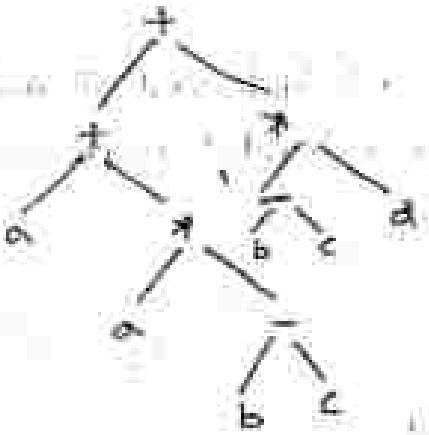
- Here (a+b) subexpression is repeated
- This can be easily identified using DAG

Three address code

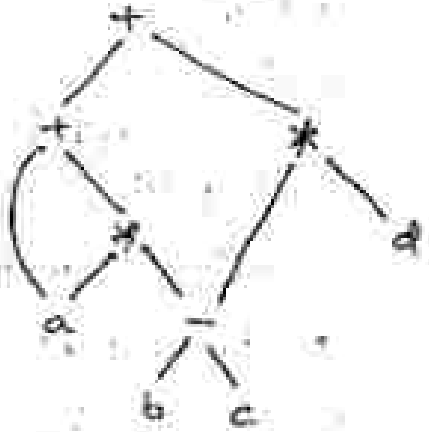
$t_1 := a+b$
$t_2 := t_1 * c$
$t_3 := t_2 + t_1$



a + a * (b-c) + (b-c) * d



Abstract Syntax Tree



DAG

(2) TYPE CHECKER

(i) Type checking :-

→ Type checking is a methodology to check whether the source program follows both the syntactic and semantic conventions of the source language.

→ It ensures that the errors will be detected and reported.

→ A type checker is placed between the parser and Intermediate code generator as

Position of a type checker

(ii) Types of Type checking :-

→ There are two types in Type checking

(i) Static type checking

(ii) Dynamic type checking

(i) Static type checking :-

→ Type checking done at compiletime is known as Static type checking.

→ A programming language uses static checking when it wants type checking to be done at compile time.

→ Languages like C, C++, C#, Java and Haskell uses static checking.
→ Static checking is also called as "Early Binding".
→ During Static Checking programmatic errors are caught early. This causes program execution to be efficient
→ Usage of Static checking makes programs more reliable for execution
→ Static checking not only increases efficiency and reliability of the compiled program, but also the execution is made quicken

(ii) Dynamic type checking :-

→ Type checking done at runtime is called as dynamic typechecking
→ A programming language uses dynamic checking when it wants type checking to be done at runtime.
→ Languages like Perl, python and LISP uses dynamic checking
→ Dynamic checking is also called as "Late Binding"
→ Dynamic checking allows constructs that some static checking might reject as illegal.
→ Meta Programming can be made powerful and easy to use by using dynamic checking.

→ Some examples of Static checks are,

- Type checks :- A compiler should report an error if an operator is applied to an incompatible operand.
- Flow-of-control checks :- There should be some place where the control has to go when a statement leaves the construct, example, break statement in 'C'.
- Uniqueness checks :- This checks whether the variable is defined exactly once.
- Name-related checks :- This checks for the name to appear two (or) more times when required.

(iii) Type Conversions :-

→ Type Conversion is a method of converting a variable from its datatype to another depending on the operations and other operands.

It is a conversion from one datatype to another datatype.

Ex :- int to float (or) float to int.
(widening) (narrowing)

→ Different types have different representations within a computer, for example, the representation of real is different from integer.
→ When an expression involves different types, type conversion is required.
→ For example, consider an arithmetic expression a+b, where the types of 'a' and 'b' is real and integer respectively. To compute its value, the compiler first needs to convert the type of one of the operands of '+'

Operands to another to ensure that the operator '+' have the operands of same type.

→ The type checker is used to insert these conversion operations into the intermediate representation of the source program.

→ It inserts postfix notation.

Ex:- Let us take the expression a+b.

The postfix notation is

a b inttoreal real+

where inttoreal operator converts b from int to real and then real+ performs real addition on its operands.

→ Type conversion is of two types.

(i) Implicit Type Conversion

(ii) Explicit Type Conversion

→ The conversion from one datatype to another datatype represented manually is known as Explicit type conversion.

Syntax:- In 'c' language,

Variable = (datatype) (expression)

(Explicit)

Ex:- int a;
float b;
float k;
k = (float) a + b

→ The Implicit type conversion is also called as type coercion.

→ The type conversion is said to be implicit, if the conversion is done by the compiler automatically.

→ The type coercion occurs only when there is no loss of information.

→ For example, the conversion of integer to real is automatic, but the reverse, i.e, from real to integer is not possible.

→ The conversion from real to integer is possible, when a real number must fit into the same number of bits used by an integer.

→ In 'c', the type conversion is implicit. Here the ASCII characters are converted to integers between 0 to 127 in arithmetic expressions.

→ The type checking rules for coercion from integer to real are,

Production	Semantic Rule
E → number	E.type := integer
E → number.number	E.type := real
E → id	E.type := LOOKUP(id.entry)
E → E_1 op E_2	if E_1.type = integer and E_2.type = integer then E.type := integer else if E_1.type = real and E_2.type = real then E.type := real else if E_1.type = integer and E_2.type = real then E.type := real else if E_1.type = real and E_2.type = integer then E.type := real else E.type := type_error.

3) Type Systems:- (Types and declarations)

→ A Type System is a set of rules that describe how type expressions are assigned to the various parts of a program.
→ It is implemented by a type checker.
→ Different compilers or processors of the same language may use different type systems.

(I) Type Expressions:-

→ A Type expression denotes the type of a language construct.
→ It is a basic type or is created by applying a type constructor to other type expressions.
→ Some of the type expressions are
  (i) A basic type such as boolean, char, integer and real is a type expression
  (ii) A special basic type type_error is a type expression which is used to indicate an error during type checking
  (iii) A basic type void is also a type expression, used to check the statements which have no return value.
  (iv) The name of a type expression is a type expression.
  (v) A type expression is created by applying a type constructor to other type expressions is a type expression.
    → Type Constructors are
      - arrays Ex:- array[0...9]
      - Products Ex:- type1 x type2
      - records Ex:- list
      - pointers Ex:- var ptr: ↑integer
      - functions:-
        → type of a function is $D \rightarrow R$
        Here D - Domain.
        R - Range.

(II) Equivalence of Type Expressions:-

→ The type checking rules for expressions tells that, "if two expressions are equal then return a type else return type_error"
→ So it should be known when the two type expressions are equal and when they are not equal.
→ This is known as Equivalence of Type expressions.
→ The type equivalence is of two types
  (i) Structural Equivalence
  (ii) Name Equivalence

(i) Structural Equivalence:-

→ The structural equivalence of type expressions tells that, "two type

expressions are structurally equivalent if and only if they are identical.

→ That is, the two expressions should be of same basic type or they should be formed by applying the same constructor.

EXAMPLE:

a) The type expression integer is equivalent only to integer, because they are of same basic type.

b) Pointer(integer) is equivalent to pointer(integer), because the two are formed by applying the same constructor pointer to equivalent types.

→ An algorithm for testing structural equivalence of type expressions formed by the type constructors array, product, pointers and functions,

```
function sturcteq(S, t) : boolean;
begin
    if S and t are the same basic type then
        return true
    else if S = array(S1, S2) and t = array(t1, t2) then
        return sturcteq(S1, t1) and sturcteq(S2, t2)
    else if S = S1 × S2 and t = t1 × t2 then
        return sturcteq(S1, t1) and sturcteq(S2, t2)
    else if S = pointer(S1) and t = pointer(t1) then
        return sturcteq(S1, t1)
    else if S = S1 → S2 and t = t1 → t2 then
        return sturcteq(S1, t1) and sturcteq(S2, t2)
    else
        return false
end;
```

(ii) Name Equivalence :-

→ Two type expressions are said to be name equivalent, if both have the same type name.

→ In name equivalence, each type name is viewed as a distinct type.

→ This requires that each type is given a name.

Ex:- Consider the following Pascal program segment

```
Type link = ↑cell;
var next : link;
    last : link;
    p : ↑cell
    q, r : ↑cell
```

Here, the identifier link is declared as a name for the type ↑cell

Now, to check whether next, last, p, q, r all have the identical types, we allow the type expressions to be named and allow these names to appear in type expressions.

Ex:- if Cell is the name of a type expression, then pointer(cell) is a type expression

→ The type expressions with its associated variable for the pascal program is

Variable	Type expression
next	link
last	link
p	pointer(cell)
q	pointer(cell)
r	pointer(cell)

→ When name equivalence is considered, the variables next and last have the same type because they have the same type expressions associated with them.

→ Also, p, q, r have the same type

→ When structural equivalence is considered, all of five variables are considered to have the same type.

(a) Advantages and disadvantages of structural equivalence:-

Advantages:-

→ Structural equivalence is less restrictive

→ It can work with unnamed types

→ It makes a language more suitable / flexible

Disadvantages:-

→ Testing of type expressions for structural equivalence is difficult.

(b) Advantages and disadvantages of name equivalence:-

Advantages:-

→ Name equivalence is easier to implement

→ It is faster and more restrictive

→ It is efficient and can be defined easily.

Disadvantages:-

→ It requires that the types involved in type expression must be defined before use with names.

(*) Overloading of functions and operators:-

(i) Overloading of operators:- (operator overloading)

→ A symbol is said to be overloaded if it has different meaning depending on its context (or) use.

→ An operator is overloaded if the same operator performs different

operations

Ex:-

→ An arithmetic expression a+b, the addition operator '+' is overloaded because it performs different operations, when a and b are of different types like integer, real and complex, so on.

→ Another example for operator overloading, is the overloaded parentheses in ADA, i.e, the expression Sum(K) has different meanings

- It can be the K^{th} element of an array Sum
- a calling function Sum with argument K and so on.

→ The process of resolving overloading is called operator identification, because it specifies what operation an operator performs.

→ The overloading of arithmetic operators can be easily resolved by seeing only the operands of an operator.

(iii) Function Overloading:-

→ like Operators the functions can also be overloaded.

→ In function overloading, the functions have the same name but different arguments of different numbers and different types.

→ In Ada, the operator '*' has the standard meaning, that it takes pairs of integers and returns an integer. The function '*' can be overloaded by adding the following declarations.

```
function '*' (a,b: integer) return integer
function '*' (l, m: Complex) return Complex
```

→ The declarations of '*' include the possible types.

```
integer x integer ⟶ integer
integer x integer ⟶ Complex
Complex x Complex ⟶ Complex
```

→ Another example of function overloading is

```
sum(int a, int b)
sum( )
sum(float a, float b)
sum(int a, float b, double c)
```

→ Type checking rule for the function overloading is,

$E \rightarrow E_1(E_2)$	{ E.type := if E_2.type = S and E_1.type = S→t then t else type_error

Generic function :-

→ A generic function is built-in function that accepts arguments of several type.
→ Most generic functions for math are polymorphic
Ex:- the function Sin(x) is polymorphic because x can be a type of int, float, double etc.
→ A generic function uses parametric polymorphism, that is, it executes the same algorithm regardless of the type of its arguments.
→ In ADA, generic functions are parametrically polymorphic.
→ When a generic function is called, overload dispatch occurs which selects the proper function based on its arguments.

polymorphic functions :-

→ A polymorphic function is a function that can evaluate to and can be applied to values of different types.
→ A polymorphic function allows the statements in its body to be executed for any type of arguments.
→ The type of polymorphic function is expressed as

$$\forall x \cdot \text{procedure\_name}(x) \rightarrow x$$

Here, the symbol $\forall$ is the universal quantifier meaning "for any type (x) for all".
x is a type variable.
→ A type expression that uses $\forall$ symbol is referred to as a polymorphic type.
→ A type variable can be letters a, b, c ..... etc and can only be used in type descriptions.
→ The type variable to which the $\forall$ symbol is applied is said to be bound by it.

Ex:- A count function

$$\forall x \cdot \text{count}(x) \rightarrow \text{integer}$$

- if the type of elements in the list are integers, then declaration is

$$\text{Count(integer)} \rightarrow \text{integer}$$

- if the type of elements in the list are char, then

$$\text{Count(char)} \rightarrow \text{integer}$$

→ Determining the type of function from its body is known as type inference.
→ The inference rules decides which version of a polymorphic function to be called at compile time based on the arguments of the function.
→ A language that allows the language constructs to have more than one type is said to be a polymorphic language.

TRANSLATION OF EXPRESSIONS

Q20. Give the translate scheme to convert an expression grammar into three-address code.

Answer :

A data type for an expression can be integer, real, array and record. The translation scheme for assignment statements into three-address code shows how to look up names in the symbol table.

Names Lookup

The translation scheme to produce three-address code for assignment statements is given below.

$S \rightarrow id := E$ {

```
    x := LOOKUP (id.name);
    if x ≠ nil
    then EMIT (x '=' E.place)
    else
      error
}.
```

```
F → F1 + F2   {
                F.place := Temp( );
                EMIT    (F.place ':=' F1.place '+' F2.place)
              }
F → E1 * E2   {
                F.place := TEMP ( );
                EMIT (F.place ':=' E1.place '*' E2.place)
              }
F → (F1)      {
                F.place := TEMP ( );
                EMIT (F.place ':=' 'Uminus' F1.place)
              }
F → (F1)      {
                F.place := F1.place
              }
F → id        {
                x := LOOKUP (id.name);
                if x ≠ nil
                then F.place := x
                else
                error
              }
```

The attribute id.name gives the lexeme for the name represented by id. The function LOOKUP ( ) locates the name in the symbol table. If it is present, it returns a pointer to the entry otherwise returns nil. The function EMIT ( ) produces and sends the three address statements for nonterminals to an output file rather than creating the code attributes for them.

The function TEMP ( ) generates and returns a new temporary name each time a temporary is needed. The bulk of temporaries generated fill the symbol table and require the space to hold their values. To prevent it from happening they can be reused by modifying the function TEMP.

Q26. Write a short note on short circuit code.

Answer :

Short circuit code is a code, wherein the boolean operators like && and || (or) and ! not are translated directly into jumps. It is also called jumping code. The code that is translated does not contain boolean operators instead the impact of a boolean expression can be reflected by a position of program, once the boolean expression gets evaluates for instance, consider the program.

if (p < 10 || P > 20 && P != q) P = 5;

The above source program is translated into the code as follows,

```
        if P < 10 go to L2
        if false P > 20 goto L1
        if false P != q goto L1
L2 :    P = 5
L1 :
```

Figure Short circuit Code

If the boolean expression is true then control goes to label L_2. On the other hand, if the boolean expression is false then control reaches to L_1 rather than label L_2 and assignment statement P = 5.

Q27. Give a syntax-directed translator scheme for converting the statements of the following grammar into three-address code

S → While expr do begin S end

| S; S

| break

| other.

Answer :

Model Paper-II, Q5(b)

Let the nonterminal E denote the nonterminal expr. The syntax directed translation scheme for the given statements is given below,

Production	Semantic Rule
S → while E do begin S_1 end	S.start := genLabel ( ); S.follow := genLabel ( ); S.code := generate (S.start ':' || E.code || generate ('if' E.place '=' '0' 'goto' S.follow) || S_1.code || generate ('goto' S.start) || generate (S.follow ':')
$S \rightarrow S_1 ; S_2$	S.code := generate (S.start '=' S.follow)
S → break	S.code := generate (S.start ':' || E.code || generate ('if' E.place '=' '0' 'goto' S.follow) || generate('break'S.start), || S_1.code || generate('goto'S.start)
S → other	S.code := generate(S.start ':') || E.code || generate ('if' E.place '=' '0' 'goto' S.follow) || S_1.code || generate('goto'S.start) || generate(E.follow) /*code for other statements*/ end generate ('other') /*code for other statements*/ end

This definition has two labels S.start and S.follow to mark the starting statement for S and the next statement after the code for S.

Q28. Write syntax directed translation scheme for generating three address code for the string generated by the following grammar.

S → if E then S | if E then S else S | while E do S.

Answer :

The grammar for control flow statements is,

$S \rightarrow$ if E then S

| if E then S else S

| while E do S.

The semantic rules that produces three address code for control flow statements is given below.

Productions	Semantic Rules
$S \rightarrow$ if E then S_1	E.t := gen Label; E.f := S.next; S_1.next := S.next; S.code := E.code || generate (E.t ':') || S_1.code
$S \rightarrow$ if E then S_1 else S_2	E.t := gen Label; E.f := gen Label; S_1.next := S.next; S_2.next := S.next; S.code := E.code || generate (E.t ':') || S_1.code || generate ('goto' S.next) || generate (E.f ':') || S_2.code
$S \rightarrow$ while E do S_1	S.start := gen Label; E.t := gen Label; E.f := S.next; S_1.next := S.start; S.code := generate (S.begin ':') || E.code || generate (E.t ':') || S_1.code || generate ('goto' S.start)

Table: Syntax Directed Definition for Control Flow Statements

In the syntax directed definition there are two labels *E.t* and *E.f*.

- *E.t* is a label to which control will flow when *E* is true.
- *E.f* is a label to which control will flow when *E* is false.

6 BACKPATCHING

Q29. What is backpatching?

Answer : **Model Paper-II, Q5(b)**

Backpatching

Backpatching is defined as the process of filling in the temporarily left unspecified targets of the jumps with appropriate semantic actions during code generation.

Backpatching is mainly used to overcome the problem of generating three-address code (or quadruples) for boolean expressions in only one pass using a syntax-directed translation scheme. If we generate the code using syntax directed translation scheme with only one pass then it becomes difficult to predict the target locations of jump statements. The most appropriate method is to use syntax direct translation scheme using two passes. In one pass, we can leave the targets of jumps unspecified and in another pass we can fill in the targets.

Code Generation for Boolean Expressions Using Backpatching

Consider the following grammar for boolean expressions for which semantic actions are to be produced.

$E \rightarrow E_1$ OR N E_2

$E \rightarrow E_1$ AND N E_2

$E \rightarrow$ NOT E_1

$E \rightarrow (E_1)$

$E \rightarrow$ id, relopid,

$E \rightarrow$ TRUE

$E \rightarrow$ FALSE

$N \rightarrow \varepsilon$

In semantic actions following function are used

1. **MAKELIST (i)**

   It creates the new list. i refers to the index of the array of quadruple.

2. **MERGE (P_1, P_2)**

   This function takes two pointers as parameters, which point to two lists. It concatenates these lists and returns a pointer to the new list.

3. **BACKPATCH (P, i)**

   This function gives the name 'i' as the target label to the instruction pointed by pointer P. The syntax directed translation scheme for boolean expressions using backpatching is as follows.

Production Rule	Semantic Action

$E \rightarrow E_1$ OR N E_2	{
	BACKPATCH (E_1. Flist, N.addr);
	E.Elist: = MERGE (E_1. Elist, E_2 Elist);
	E.Flist: = E_2. Flist
	}

$E \rightarrow E_1$ AND N E_2	{
	BACKPATCH (E_1.Elist, N. addr);
	E Elist: = E_2. Elist;
	E. Flist: = MERGE(E_1. Flist, E_2. Flist)
	}

Prodiction Rule	Semantic Action

$E \rightarrow$ NOT E_1	{
	E. Elist: = E_1.Flist;
	E. Flist: = E_1. Elist
	}

$E \rightarrow (E_1)$	{
	E.Elist: = E_1.Elist;
	E.Flist: = E_1.Flist
	}

$E \rightarrow$ id, relop id,	{
	E. Elist: = MAKELIST (next state);
	E. Flist: = MAKELIST (next state + 1);
	GENCODE ('IF' id,.place relop id, place 'goto_')
	GENCODE ('goto_')
	}

E → TRUE	{
	E. Elist = MAKELIST (nextstate);
	GENCODE('goto_')
	}

E → FALSE	{
	T. Flist = MAKELIST (nextstate);
	GENCODE('goto_')
	}

N → ε	{
	N.addr = nextstate
	}

- Elist and Flist are the synthesized attributes which generate the jumping codes for the boolean expressions. E. Elist will be generated for the true statement and E.Flist for the false statement.
- N is the marker non terminal. It is used to mark the exact point where the semantic action should select the next statement.
- The attribute 'addr' holds the address (or number) of the statement and is related to N. (N state). The 'nextstate' attribute points to the next statement.

Q30. Explain back patching with reference to the three address code generated for the expression p > q and q < r.

Answer :

First we will scan this statement from left to right.

The first expression E_1 = p > q matches with the production rule $E \rightarrow id_1$ relop id_2.

Assume that the initial value of nextstate = 0. It is incremented each time the ( )function is called the three address code generated in response to this production rule is as follows

(0) if p > q goto_

(1) goto_

The above code is based on the semantic actions for the production rule. The semantic actions are,

GENCODE ('if id_1.place relop id_2.place goto_')

GENCODE ('goto_')

Now, E_1. Tlist = {0};

E_1. Flist = {1} and

nextstate = 2 Since, GENCODE( ) is called two times.

Similarly, the three address code for the next expression,

E_2 = q < s will be

(2) if q < s goto_

(3) goto_

Now,

E_2. Tlist = {2}

E_2. Flist = {3}

nextstate = 4

The code generated so far is,

(0) if p > q goto

(1) goto_

(2) if q < t goto_

(3) goto_

Now, consider the entire expression if p > q and q < t, which corresponds to the production rule $E \rightarrow E_1$ AND N E_2. The semantic actin for this rule is BACKPATCH (E_1. Tlist, N.addr)

Since N.addr = nextstate = 2,

E. Tlist = {2}

E. Flist = {1, 3}

This semantic routine calls BACKPATCH ({0}, 2), which fills in 2 in the statement 0.

Therefore the code generated look like this.

(0) if p > q goto (2)

(1) goto_

(2) if q < t goto_

(3) goto_

The entire expression is true if and only if the goto of statement 2 is reached i.e., both the conditions are satisfied. It is false if and only if the goto's of statements 1 and 3 are reached i.e, either of the condition is not satisfied. The targets for the statements 1, 2 and 3 will be filled in at the time of compilation when it becomes apparent what action to perform depending on the evaluation of the expression.

The parse tree for the expression is shown below

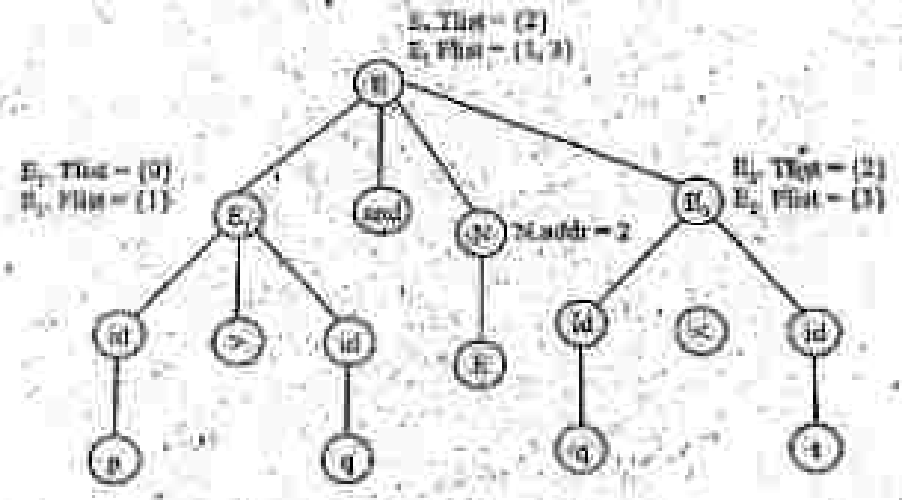


Figure: Parse Tree for p > q and q < t

SHORT QUESTIONS WITH SOLUTIONS

Q1. Write short notes on intermediate code generator.

Answer : **Model Paper-I, Q1(d)**

The intermediate code generator performs a conversion of syntax tree into a code that is not in target code form, it is into an intermediate code. The intermediate code should be easily translatable to target code and intermediate code generation should not take much time.

The intermediate code can be of three forms,

1. Postfix
2. Triples
3. Quadruples.

The intermediate code should have the following basic properties.

1. Each instruction can have only one operator in addition to the assignment operator.
2. Compiler should generate temporary locations to store the value evaluated in each instruction.
3. The code (3 address code) may have less than 3 operands in an instruction.

Q2. Compare various forms of three address code.

Answer :

Various forms of three-address code are compared based on how much indirection is used in the representation.

1. When a target code is produced, a run-time memory location is assigned for each temporary name or programmer defined name and this location is stored in the symbol table. In the quadruple structure, three address statement can access the location of temporary name, if it is used immediately from the symbol table.
2. The quadruple structure has high degree of indirection between the computation of a value and its use. This is possible through symbol table.
3. It is difficult to use a triple structure in an optimizing compiler. Since, changing a statement defining a temporary value requires to change all the occurrences of it in the fields operand 1 and operand 2.
4. Indirect triples structure is similar to quadruple which requires changes in the ordering of the statement list.
5. Quadruples and indirect triples requires the same amount of space.
6. In triples, the temporaries that need storage are deferred until the code generation phase.
7. The space required by indirect triples can be reduced by using the same temporary value more than once.

Q3. Translate the executable statements of the following 'C' program into a three-address code by assuming each element of an array 'a' takes 4 bytes.

```
void main( )
{
  int i = 1, a[10];
  while(i + + <= 10) a[i] = 0;
}
```

Answer :

The size of an element in array 'a' is 4 bytes.

$\therefore$ $a[i] = addr(a) + 4*i$

let addr (a) be some address location 25

The three-address code is,

0. main : $i = 1$
1. $a[10]$
2. $i = i+1$
3. if $i <= 10$ Goto(5)
4. Goto(8)
5. $temp_1 : = 25 + 4*i$
6. $a[temp_1] = 0$
7. Goto (2).

Q4. Translate the following into three address code

while (A < B) do

if (C < 0) then X = Y + Z.

Answer :

The three address code for the following code is given as follows,

Label 1: While A < B then GOTO Label 2.

GOTO Label Next

Label 2: If C < 0 then GOTO Label 3

GOTO LABEL 3

Label 3: temp1 = Y + Z

x = temp1

GOTO Label 1

Label NEXT

Q5. Construct triples of the expressions: a[i]: = b and a: = b[i].

Answer : Model Paper-IV, Q1(d)

The triple representation for $a[i] := b$ is,

Line No.	Operator	Operand 1	Operand 2
(0)	[ ] =	a	i
(1)	equals	(0)	b

Assignment operator

For $a := b[i]$ is,

Line No.	Operator	Operand 1	Operand 2
(0)	= [ ]	b	i
(1)	equals	a	(0)

Q6. Write quadruple representation for a + a * (b – c) + (b – c) * d.

Answer : Model Paper-III, Q1(d)

Given statement is,

$a + a * (b - c) + (b - c) * d$

The three address code for above statement is,

temp1 = (b – c)

temp2 = a * temp1

temp3 = a + temp2

temp4 = temp1 * d

temp5 = temp3 + temp4

The quadruple for a + a * (b – c) + (b – c) * d is as follows,

argument1	argument2	operand	Result
b	c	–	temp1
a	temp1	*	temp2
a	temp2	+	temp3
temp1	d	*	temp4
temp3	temp4	+	temp5

Q7. Construct the triples of expression : a* – (b + c).

Answer :

The 3-address code for the expression a * – (b + c) is,

(0) $temp_1 := (b + c)$

(1) $temp_2 := - temp_1$

(2) $temp_3 := a * temp_2$

and the triple representation is,

Line No.	Operator	$Operand_1$	$Operand_2$
(0)	addop	b	c
(1)	uminus	(0)	
(2)	mulop	a	(1)
	↓		
	Internal representation		
	of * operator		

Q8. What are postfix SDT's?

Answer :

Postfix SDT's are the SDT's which perform every action at right ends of the grammar productions.

The following example illustrates postfix SDT implementing a desk calculator.

Grammar Production	Semantic Rules
$L \rightarrow E_n$	Print E.Value
$E \rightarrow E_1 + T$	E.Value := E_1.Value + T.Value
$E \rightarrow T$	E.Value := T.Value
$T \rightarrow T_1 * F$	T.Value := T_1.Value * F.Value
$T \rightarrow F$	T.Value := F.Value
$F \rightarrow (E)$	F.Value := E.Value
$F \rightarrow$ digit	F.Value := Digit. Value

Table: Postfix SDT's implementing Desk Calculator

Q9. Write short notes on semantic analysis phase and static checking.

Answer : Model Paper-A, Q1(d)

Semantic Analysis Phase

Semantic analyzer checks that the input gives any meaning or not. After the parser checks for syntactic errors, semantic analyzer checks for semantic errors. It does type checking for expressions and other statements.

Static Checking

The following are some important points about static checking.

1. A programming language uses static checking when it wants type checking to be done at compile time.
2. Static checking is also called as "Early Binding".
3. During static checking programmatic errors are caught early. This causes program execution to be efficient.
4. Usage of static checking makes programs more reliable for execution. This statement means, at execution time the program is hardly left with the basic errors, so the executor can freely execute the program.