END TERM EXAMINATION [MAY-JUNE 2017]
SIXTH SEMESTER [B.TECH]
COMPILER DESIGN [ETCS-302]

Time : 3 Hrs. **M.M. : 75**

Note: *Attempt any five questions include Q. No 1 which is compulsory. Select one question from each unit.*

Q.1. Attempt any five parts:

Q.1. (a) Explain the process of Bootstrapping in compiler design with example. **(5)**

Ans. Refer Q. No. 4 (a) First Term Examination 2017.

Q.1. (b) Differentiate between SDD and SDT with Example. **(5)**

Ans. SYNTAX-DIRECTED TRANSLATION(SDT) A formalism for specifying translations for programming language constructs. (attributes of a construct: type, string, location, etc)

- Syntax directed definition(SDD) for the translation of constructs
- Syntax directed translation scheme(SDTS) for specifying translation Postfix notation for an expression E
- If E is a variable or constant, then the postfix nation for E is E itself (E.t ≡ E).
- if E is an expression of the form E1 op E2 where op is a binary operator o E1' is the postfix of E1, o E2' is the postfix of E2 o then E1' E2' op is the postfix for E1 op E2
- if E is (E1), and E1' is a postfix then E1' is the postfix for E

eg) 9 - 5 + 2 ⇒ 9 5 - 2 + 9 - (5 + 2) ⇒ 9 5 2 + -

Syntax-Directed Definition(SDD) for translation

- SDD is a set of semantic rules predefined for each productions respectively for translation.
- A translation is an input-output mapping procedure for translation of an input X, o construct a parse tree for X. o synthesize attributes over the parse tree.
- Suppose a node n in parse tree is labeled by X and X.a denotes the value of attribute a of X at that node.
- compute X's attributes X.a using the semantic rules associated with X.

Q.1. (c) What is Left Recursion and Left Factoring ? Explain each with Example. **(5)**

Ans. Left Recursion

- A grammar is left recursive iff it contains a nonterminal A, such that A→+ Aα, where is any string.

Grammar {S → Sα | c} is left recursive because of S ⇒Sα

Grammar {S →Aα, A → Sb | c} is also left recursive because of S⇒Aα⇒Sbα

- If a grammar is left recursive, you cannot build a predictive top down parser for

(1) If a parser is trying to match S & S→Sα, it has no idea how many times S must be applied (2) Given a left recursive grammar, it is always possible to find another grammar that generates the same language and is not left recursive. 3) The resulting grammar might or might not be suitable for RDP.

• After this, if we need left factoring, it is not suitable for RDP.
• Right recursion: Special care/Harder than left recursion/SDT can handle.

S $\rightarrow$ Aa $\rightarrow$ b
A $\rightarrow$ **Ac/Sd/e** As we don't have immediate left recursion,

Step(1) Rename S,A as A_1,A_2 respectively

$A_1 \rightarrow A_2 a \mid b$
$A_2 \rightarrow A_2 c \mid A1\ d \mid e$

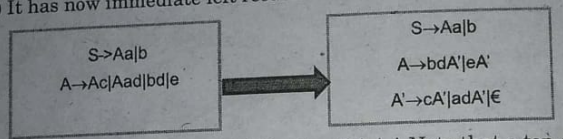
Step(2) Substitute value A_1 of statement (1) in R.H.S of statement (2)

$A_1 \rightarrow A_2 a \mid b$
$A_2 \rightarrow A_2 c \mid A_2\ a\ d \mid bd \mid e$

Step(3)
Again substitute

$A_1 = S, A_2 = A$
$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Aad \mid bd \mid e$

Step (4) It has now immediate left recursion



Eliminating Left Recursion Let G be S → S A | A Note that a top-down parser cannot parse the grammar G, regardless of the order the productions are tried. ⇒The productions generate strings of form AA..A

⇒ They can be replaced by S →A S' and S →A S' | ε

Left Factoring

• If a grammar contains two productions of form S → aβ and S →aα it is not suitable for top down parsing without backtracking. Troubles of this form can sometimes be removed from the grammar by a technique called the left factoring.

• In the left factoring, we replace { S→ aα, S→ aβ } by { S → aS′, S′ → α, S′ → β } cf. S → a(α | β) (Hopefully α and β start with different symbols)

• left factoring for G { S → aSb | c | ab } S → aS' | c cf. S(=aSb | ab | c = a (Sb | b) | c) → a S' | c S′→Sb | b

Q.1. (d) What is Back Patching? Explain with Example. **(5)**

Ans. BACKPATCHING

The easiest way to implement the syntax-directed definitions for boolean expressions is to use two passes. First, construct a syntax tree for the input, and then walk the tree in depth-first order, computing the translations. The main problem with generating code for boolean expressions and flow-of-control statements in a single pass is that during one single pass we may not know the labels that control must go to at the time the jump statements are generated. Hence, a series of branching statements with the targets of the jumps left unspecified is generated. Each statement will be put on a list of goto statements whose labels will be filled in when the proper label can be determined. We call this subsequent filling in of labels backpatching.

To manipulate lists of labels, we use three functions :

1. makelist(i) creates a new list containing only i, an index into the array of quadruples; makelist returns a pointer to the list it has made.

2. merge(p1,p2) concatenates the lists pointed to by p1 and p2, and returns a pointer to the concatenated list.

3. backpatch(p,i) inserts i as the target label for each of the statements on the list pointed to by p.

Boolean Expressions:

We now construct a translation scheme suitable for producing quadruples for boolean expressions during bottom-up parsing. The grammar we use is the following:

(1) E E1 or M E2
(2) | E1 and M E2
(3) | not E1
(4) | (E1)
(5) | id1 relop id2
(6) | true
(7) | false
(8) M → ε

Synthesized attributes truelist and falselist of nonterminal E are used to generate jumping code for boolean expressions. Incomplete jumps with unfilled labels are placed on lists pointed to by E.truelist and E.falselist.

Q.1. (e) What is the Process of identifying basic blocks in code optimization phase?

Ans. OPTIMIZATION OF BASIC BLOCKS **(5)**

There are two types of basic block optimizations. They are :

- Structure-Preserving Transformations
- Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

a: =b+c
b: =a-d
c: =b+c
d: =a-d

The 2nd and 4th statements compute the same expression: b+c and a-d

Basic block can be transformed to

a: = b+c
b: = a-d
c: = a
d: = b

Dead code elimination:

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of

remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

Renaming of temporary variables:

• A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.

• In this we can transform a basic block to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

Two statements

t1:=b+c

t2:=x+y

can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

Algebraic Transformations:

Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.

Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.

he relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.

Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

a :=b+c e :=c+d+b

the following intermediate code may be generated:

a :=b+c t :=c+d e :=t+b

Example:

x:=x+0 can be removed

x:=y**2 can be replaced by a cheaper statement x:=y*y

The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y-x*z as x*(y-z) but it may not evaluate a+(b-c) as (a+b)-c.

Q.1. (f) Differentiate between top-down and bottom-up parsers with example. **(5)**

Ans. Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

• **Recursive descent parsing** : It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.

• **Backtracking** : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Bottom-up P

As the name s
to construct the pa

Example:

Input string : a

Production rule

$S \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow E * T$

$E \rightarrow T$

$T \rightarrow id$

Let us start b

a + b * c

Read the inpu

a + b * c

T + b * c

E + b * c

E + T * c

E * c

E * T

E

S

**Q.1. (g) Wri
taking suitabl**

Ans. If you
equivalent post

• Variabl

• E op F

• (E) be

1+2/3-4*

1. Start

2. Paren

3. Apply
the infix exp

A. P{(1+

B. P{(1-

C. P{1+

D. P{1}

E. 1 P{

F. 1 2

Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Example:

Input string : a + b * c

Production rules:

$S \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow E * T$

$E \rightarrow T$

$T \rightarrow id$

Let us start bottom-up parsing

a + b * c

Read the input and check if any production matches with the input:

a + b * c

T + b * c

E + b * c

E + T * c

E * c

E * T

E

S

Q.1. (g) Write a SDT for converting infix expression to post fix expression by taking suitable example. **(5)**

Ans. If you start with an infix expression, the following algorithm will give you the equivalent postfix expression.

- Variables and constants are left alone.
- E op F becomes E' F' op, where E' and F' are the postfix of E and F respectively.
- (E) becomes E', where E' is the postfix of E.

1+2/3-4*5

1. Start with 1+2/3-4*5
2. Parenthesize (using standard precedence) to get (1+(2/3))-(4*5)
3. Apply the above rules to calculate P[(1+(2/3))-(4*5)], where P[X] means "convert the infix expression X to postfix".

A. P[(1+(2/3))-(4*5)]

B. P[(1+(2/3))] P[(4*5)] -

C. P[1+(2/3)] P[4*5] -

D. P[1] P[2/3] + P[4] P[5] * -

E. 1 P[2] P[3] / + 4 5 * -

F. 1 2 3 / + 4 5 * -

UNIT–I

Q.2. (a) For the grammar given below: **(7.5)**

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon,$$
$$F \rightarrow (E) \mid ID$$

construct the LL(1) parsing table

Ans. First( ) :

FIRST(E) = { (, id}
FIRST(E') ={+ , ε }
FIRST(T) = { (, id}
FIRST(T') = {*, ε }
FIRST(F) = { (, id }
Follow( ):
FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +, $,) }
FOLLOW(T') = { +, $,) }
FOLLOW(F) = {+, * , $,) }

Predictive parsing table:

NON-TERMINAL	id	+	*	(	)	$
E	E → TE'			E → TE'		
E'		E' → + TE'			E'→ ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

Stack implementation:

Stack	Input	Output
$E	id + id*idS	
$E'T	id+id*id$	E → TE'
$ET'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id$	
$E'	+id*id$	T' → ε
$E'T+	+id*id$	E' → + TE'
$E'T	id*id$	
$ET'F	id*id$	T → FT'
$E'T'id	id*id$	F → id
$ET'	*id$	
$ET'F*	*id $	T' → *FT'
$ET'F	id $	
$ET'id	id $	F → id
$ET'	$	
$E'	$	T' → ε
$	$	E' → ε

Q.2. (b) Check Whether the following grammar is LL(1) or not **(5)**

i. S → A/a, A → a

ii. S → aSA/ε ,A → c/ε

Ans. acc to the rule:

(1) First(A) → First(a) = {a}

(2) First(S) → First (ε) = ε

this grammar doesnot follow the rules that's why it is not LL(1)

Q.3. (a) What do you mean by handle? Check whether the grammar **(5)**

E → E + T/T

T → a or (id)is LR(0) or not

Ans. *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 → β 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 αAw αβw, then A → β in the position following α is a handle of αβw.

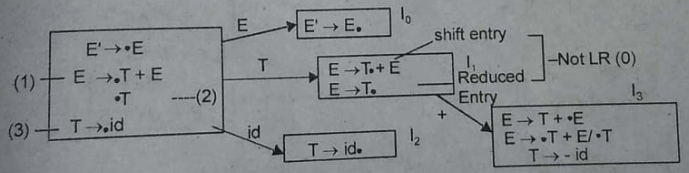
o The string w to the right of the handle contains only terminal symbols.

o 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. the augmented grammar is:

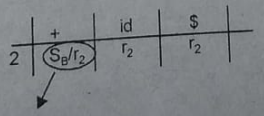
E' → E

E → T+E/T

T → a



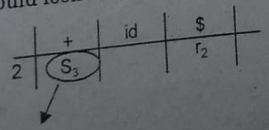
The LR(0) parsing table would look like



SR conflict → Not LR(0)

But for SLR E → T. reduced entry will go only in follow (E) = $

So SLR parsing table would look like



No SR conflect

So, the grammer is SLR but not LR (0)

Q.3. (b) Contruct a LR(1) parsing table for (7.5)

$S \rightarrow aa/bAc/dc/bda$

$A \rightarrow d$

Ans. Augmented grammar is :

$S' \rightarrow S$
$S \rightarrow Aa$
$S \rightarrow bAc$
$S \rightarrow Bc$
$S \rightarrow bBa$
$A \rightarrow d$
$B \rightarrow d$

Step 2: Find Closure & goto i.e. construct set of LR (1) items. Hence the boxes represents new states.

I_0: $S' \rightarrow \bullet S,\$$
$S' \rightarrow \bullet Aa,\$$
$S' \rightarrow \bullet bAc,\$$
$S \rightarrow \bullet Bc,\$$
$S \rightarrow \bullet bBa,\$$
$A \rightarrow \bullet d,a$
$B \rightarrow \bullet d,c$

$I_1 = \text{goto}(I_0, S)$
$S' \rightarrow S\bullet,\$$

$I_2 = \text{goto}(I_0, A)$
$S' \rightarrow A \bullet a,\$$

$I_3 = \text{goto}(I_0, b)$
$S \rightarrow b\bullet\ A\ c,\$$
$S \rightarrow b\bullet\ B\ a,\$$
$A \rightarrow \bullet d,\ c$
$B \rightarrow \bullet\ d,a$

$I_4 = \text{goto}(I_0, B)$
$S' \rightarrow B\bullet c,\$$

$I_5 = \text{goto}(I_0, d)$
$A \rightarrow d\bullet,a$
$B \rightarrow d\bullet,c$

$I_6 = \text{goto}(I_2, a)$
$S \rightarrow A\ a\bullet,\ \$$

$I_7 = \text{goto}(I_3, A)$
$S \rightarrow b\ A \bullet c,\ \$$

$I_8 = \text{goto}(I_3, B)$
$S \rightarrow bB \bullet a,\$$

$I_9 = \text{goto}(I_3, d)$
$A \rightarrow d\bullet,c$
$B \rightarrow d\bullet,a$

$I_{10} = \text{goto}(I_4, c)$
$S \rightarrow Bc\ \bullet,\$$

$I_{11} = \text{goto}(I_7, c)$
$S \rightarrow bAc\bullet\ ,\$$

$I_{12} = \text{goto}(I_8, a)$
$S \rightarrow b\ B\ a\bullet,\$$

State	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	

So, the

Q.4. (a)

Ans:

Example:

Q.4. (
the steps
number f
Ans.
by having
which mu
tree durir
parsing b
created a
grammar
parsing.
If an
attribute
written

LR (1) Parsing Table

State	Action				goto			
	a	b	c	d	$	S	A	B
0		s3		s5		1	2	4
1					accept			
2	s6			s9			7	8
3			s10					
4			r6					
5	r5				r1			
6			s11					
7								
8	s12		r5					
9	r6				r3			
10					r2			
11					r4			
12								

So, the LR (1) Parsing table has no multiple entries ∴ Grammar is LR (1).

UNIT–2

Q.4. (a) Write an SDT to count the number of binary digits in abinary number (5)

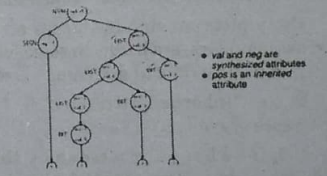
Ans:

Example: Evaluate signed binary numbers

Production	Semantic Rules
NUM → SIGN LIST	LIST.pos = 0 if SIGN.neg NUM.val = −LIST.val else NUM.val = LIST.val
SIGN → +	SIGN.neg = false
SIGN → −	SIGN.neg = true
LIST → BIT	BIT.pos = LIST.pos LIST.val = BIT.val
$LIST_1$ → $LIST_2$ BIT	$LIST_2$.pos = $LIST_1$.pos + 1 BIT.pos = $LIST_1$.pos $LIST_1$.val = $LIST_2$.val + BIT.val
BIT → 0	BIT.val = 0
BIT → 1	BIT.val = 2^BIT.pos

Example (continued)

The attributed parse tree for –101:

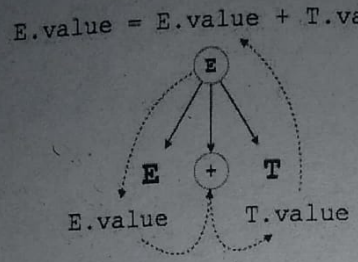


- *val* and *neg* are *synthesized* attributes
- *pos* is an *inherited* attribute

Q.4. (b) Differentiate between S-attributed and L-attributedSDT's.Write the steps to create the SDT for any problem and write SDT for converting any number from binary to decimal. (7.5)

Ans. S-Attributed Grammars are a class of attribute grammars characterized by having no inherited attributes, but only synthesized attributes. Inherited attributes, which must be passed down from parent nodes to children nodes of the abstract syntax tree during the semantic analysis of the parsing process, are a problem for bottom-up parsing because in bottom-up parsing, the parent nodes of the abstract syntax tree are created *after* creation of all of their children. Attribute evaluation in S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing.

If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).



As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

L-attributed grammars are a special type of attribute grammars. They allow the attributes to be evaluated in one depth-first left-to-right traversal of the abstract syntax tree. As a result, attribute evaluation in L-attributed grammars can be incorporated conveniently in top-down parsing.

A syntax-directed definition is L-attributed if each inherited attribute of Xj on the right side of

$$A \rightarrow X1\ X2\ ..Xn$$

depends only on

1.The attributes of the symbols X1, X2, .., Xj-1

2.The inherited attributes of A

Every S-attributed syntax-directed definition is also L-attributed.

Implementing L-attributed definitions in Bottom-Up parsers requires rewriting L-attributed definitions into translation schemes.

Many programming languages are L-attributed. Special types of compilers, the narrow compilers, are based on some form of L-attributed grammar. These are a strict superset of S-attributed grammars. Used for code synthesis.

Either "Inherited attributes" or "synthesized attributes" associated with the occurrence of symbol X1,X2---Xn

Q.5. (a) what do you mean by three address code? Explain how the three address code is represented visquadruples,triples and indirect triples with examples. **(7.5)**

Ans: Three-Address Code:

Three-address code is a sequence of statements of the general form

x : = y op z

where x, y and z are names, constants, or compiler-generated temporaries; op stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on booleanvalued data. Thus a source language expression like x+ y*z might be translated into a sequence

t1 : = y * z t2 : = x + t1

where t1 and t2are compiler-generated temporary names.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the

operands. Three such representations are:

Quadruples

Triples

Indirect triples

Quadruples:

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

→ The op field contains an internal code for the operator. The three-address statement x : = y op z is represented by placing y in arg1, z in arg2 and x in result.

→ 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.

Triples:

→ To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.

→ If we do so, three-address statements can be represented by records with only three fields: op, arg1 and arg2.

→ The fields arg1 and arg2, for the arguments of op, are either pointers to the symbol table or pointers into the triple structure (for temporary values).

→ Since three fields are used, this intermediate code format is known as triples.

Indirect Triples:

Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples

Q.5.(b) Write the three address code for: **(5)**

(I) While (a < 5) do a : b + 2

(II) – a (a + b)* (c + d) + (a + b + c)

Ans. (i) While (a<5) do a:b +2

1. if a<5
2. T1 = b+2
3. a = T1

(ii) –a (a+b)*(c+d) +(a+b+c)

Ans: t1 = a+b

t2 = -uminus(a)

t3 = c+d

t4 = t1*t3

t5 = t1+c

t6 = t3 +t5

UNIT-III

Q.6. (a) What do you mean by symbol table? Write an example that shows how different phases of compiler interact with symbol table. **(6)**

Ans: Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

<symbolname, type, attribute>

Q.6. (b) How the data is stored in symbol table for block and non block structured languages? **(6.5)**

Ans. Symbol table organization for non-blocked structured language:-

By a non-blocked structure language, we mean a language in which each separately compiled unit is a single module that has no submodules. All variables declared in a module are known throughout the module.

There are four modules

- Unordered
- Ordered
- Tree
- Hash

1. Unordered

The simplest method of organizing a symbol table is to add the attribute entries to the table in the order in which the variables are declared. So there is no particular ordering.

- Here the insertion is very easy as no comparision are required.
- The searching is more difficult hence it is very time consuming.
- For delete operation, on the average, a search length of (n+1)/2 is required, assuming there a n records.
- Time to insert a key → 1
- Unsuccessful search the time wanted is (n+1)

An unwanted table organization should be used only if the expected size of the symbol table is small, since the average time for insertion and deletion is directly proportional to the table size.

The algorithm is given for selecting a data whose key is known to us.

```
procedure select(table, key, data value, found)
  var i unteger;
  begin
        with table do
            begin
                entry[n+1].key = key;
                i=1;
                while(entry[i].key<>key
                    i=i+1;
                if i<=n then
```

```
                begin
                data value=entry[i].value;
                found=TRUE;
                end
            else
                found=FALSE;
        end
    end
```

So the complexity is O((n+1)/2). .....

Now let's see various aspects: (Actual use)

(i) Fixed as variable length entries.

In fixed , the number of entries possible in a table are fix where as in variable length it is changed.

(ii) Size of key(in byte).

(iii) Method of access

(iv)How frequent the insertion and deletion.

Language specific consideration:

(i) BASIC

Variables are stored as,

<letter> as <letter><digit>

So with this format at the most 286 entries are possible. To know the address of a given key, we have formula as follows:

<Address> = <key> - 65 OR

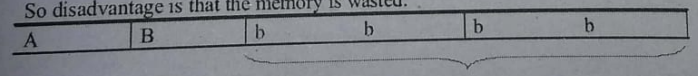
<Address> = (< key – letter> – 65) * 10 + <digit> - 48

(ii) FORTRAN

In this language the fixed sized length of variable is used.

So disadvantage is that the memory is wasted.

So disadvantage is that the memory is wasted.

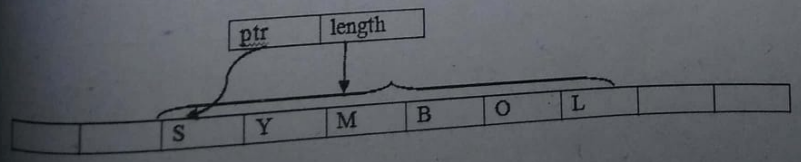


Padding (This much bytes are wasted)

(iii) C or PASCAL

In such high-level languages, the variable length key – strange are used.

With this type of key strange, we get the better memory utilization but the method is not fast.



2. Ordered

In this method, the table is maintained is sorted form based on the variable's name. In such circumstances an insertion must be accomplished by a lookup procedure which determines where in the symbol table the variable attribute should be placed. The actual insertion of new entry may generate. Some additional overhead primarily because other entries may have to be moved to get the position of insertion.

For searching a particular key, we apply Binary-Search Technique. Suppose (K_1,V_1) .... (K_{mid},V_{mid}) .... (K_n,V_n) are the entries in the table.

Here mid= n div 2

Algorithm is described below

```
Find(low, high)
While low < high do
begin
    mid = (low+high) div 2;
    if k < entry[mid].key then
                                high = mid;
                            Find(low,high);
    else
                                low = mid + 1;
                            Find(low,high);
end
```

So the key for which we are sending is placed in high or low variable.

The time complexity of this algorithm is O(log n).

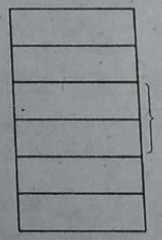
Methods of Sorting:

i. Array

We short the entries in the table in some particular .....with arrays.

With arrays, the searching of particular entry is very fast. But insertion is time consuming.

For inserting particular entry, first we have to find its position to locate it. And all the entries below it are shifted down.



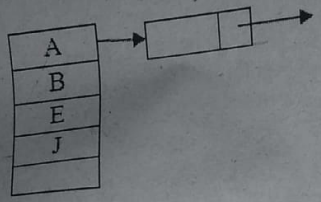
ii. Index

With this method, the insertion is easy.

Here only ptr field is manipulated. We have to do nothing with table. So, it is easier or insertion.

iii. Linked List

In this approach, we combined the array and linked list.



Here, array is for searching and linked list is used for insertion and deletion. Here there is no actual limit of number of entries in the table.

To search a key, starting symbol can be found by comparison and then entries can be counted to find the exact match.

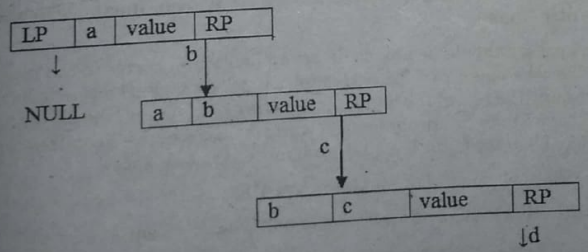
3. Tree

In a binary – tree structured symbol table, each node have the following format:

Left ptr	Key	Value	Right ptr

Here two new fields are present in the record structure. Thus two fields are left pointer and right pointer. Access to the tree is gained through the ... node. A search proceeds down the structural links of the tree until the desired node is found as a NULL link field is encounted.

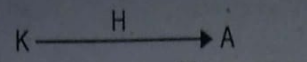
Let's take an example of storing a string abcd in this format.



Here in case of balanced binary tree the time complexity of a searching a node among the n node is given by $O(\log_2 n)$

4 Hash

A hashing function or key to address transformation is defined as a mapping H: K→A. That is, a hashing function H takes as its argument a variable named and produces a table address at which the set of attributes for that variables are stored. With this method the search time is essentially independent of the number of records in the table.



Let n be the number of entries in the table we define loading factor,

load factor = no of entries(n) / total address space (|A|)

If load factor is high, it is difficult to manage the table.

Now in practical we have

$$K >> |A|$$

So if we assign more than one key to one address, there is a problem of collision.

Pre conditioning:-

(1) Division Method:-

(2) Mid-square method:-

(3) Folding Method:-

(i) Length-dependent method:-

(a) Open Addressing:

(b) Chaining:

Symbol-Table Organization for Blocked Structured Language:-

(i) Stack symbol tables

(ii) Stack implemented tree structural tables

(iii) Stack – Implemented Hash-structured Symbol Table:

Q.7. (a) What are different types of errors that occur during lexical,syntactic and semantic phase. **(6)**

Ans: A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical** : name of some identifier typed incorrectly
- **Syntactical** : missing semicolon or unbalanced parenthesis
- **Semantical** : incompatible value assignment
- **Logical** : code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error productions

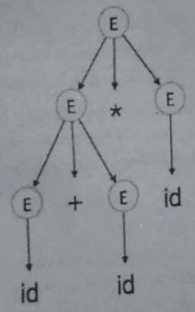
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

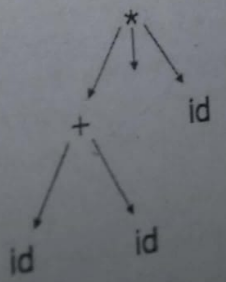
The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Abstract Syntax Trees

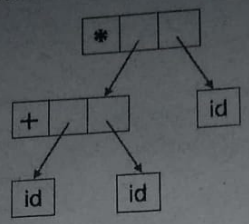
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



If watched closely, we find most of the leaf nodes are single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



Abstract tree can be represented as:



ASTs are important data structures in a compiler with least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.

Q.7. (b)What are the storage allocation strategies in the runtime environment of compiler? **(6.5)**

Ans: A program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions. A program contains names for procedures, identifiers etc., that require mapping with the actual memory location at runtime.

By runtime, we mean a program in execution. Runtime environment is a state of the target machine, which may include software libraries, environment variables, etc., to provide services to the processes running in the system.

Runtime support system is a package, mostly generated with the executable program itself and facilitates the process communication between the process and the runtime environment. It takes care of memory allocation and de-allocation while the program is being executed.

Activation Trees

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.

The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries: Stores temporary and intermediate values of an expression.

Local Data: Stores local data of the called procedure.

Machine Status: Stores machine status such as Registers, Program Counter etc., before the procedure is called.

Control Link: Stores the address of activation record of the caller procedure.

Access Link: Stores the information of data which is outside the local scope.

Actual Parameters: Stores actual parameters, i.e., parameters which are used to send input to the called procedure.

Return Value: Stores return values.

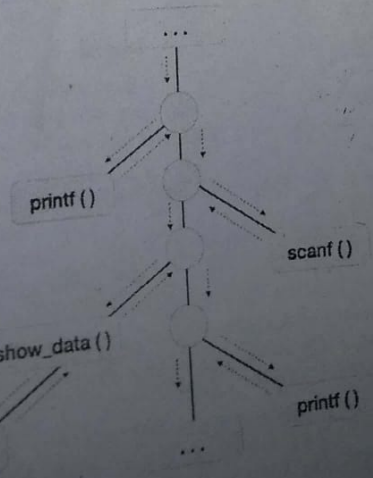
Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.

We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree**.

To understand this concept, we take a piece of code as an example:

```
...
printf("EnterYourName:");
scanf("%s", username);
show_data(username);
printf("Press any key to continue..");
...
intshow_data(char*user)
{
printf("Your name is%s", username);
return0;
}
...
```

Below is the activation tree of the code given.



Now we understand that procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

- **Code** : It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.
- **Procedures** : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- **Variables** : Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Static Allocation

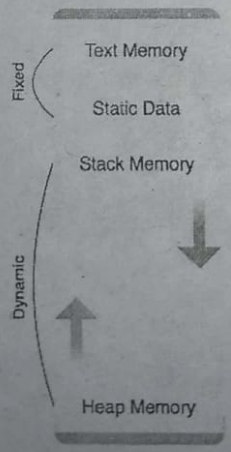
In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

Stack Allocation

Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

Heap Allocation

Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.



Except statically allocated memory area, both stack and heap memory can grow

UNIT–4

Q.8. (a) What do you mean by the term code optimization? What do you understand by the term leader? Write algorithm to identify out the basic blocks. (6)

Ans. Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Basic Blocks: A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.

The following sequence of three-address statements forms a basic block:

```
t1 : = a * a
t2 : = a * b
t3 : = 2 * t2
t4 : = t1 + t3
t5 : = b * b
t6 : = t4 + t5
```

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:We first determine the set of leaders, the first statements of basic blocks. The rules we use are of the following:

The first statement is a leader.

Any statement that is the target of a conditional or unconditional goto is a leader.Any statement that immediately follows a goto or conditional goto statement is a leader.

For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Consider the following source code for dot product of two vectors a and b of length 20 begin

```
prod :=prod+ a[i] * b[i];
i :=i+1;
end
while i <= 20
end
```

The three-address code for the above source program is given as :

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4* i
(4)   t2 := a[t1]      /*compute a[i] */
(5)   t3 := 4* i
(6)   t4 := b[t3]      /*compute b[i] */
(7)   t5 := t2*t4
(8)   t6 := prod+t5
(9)   prod := t6
(10)  t7 := i+1
(11)  i := t7
(12)  if i<=20 goto (3)
```

Q.8. (b) Identify the basic blocks in the following code and draw the DAG graph for the same: **(6.5)**

```
main()
{
int i = 0,n = 10;
int a[n];
while ( i< = (n-1))
{
a[i] = i*I;
i = i+1;
}
return;
}
```

Ans. Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.

For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

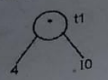
For case(iii), node n will be node(y).

Step 3: Delete x from the list cf identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

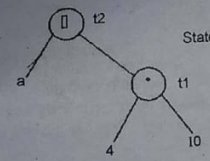
Example: Consider the block of three- address statements:

$t_1 := 4^* i$
$t_2 := a[t_1]$
$t_3 := 4^* i$
$t_4 := b[t_3]$
$t_5 := t_2 * t_4$
$t_6 := prod + t_5$
$prod := t_6$
$t_7 := i + 1$
$i := t_7$
if i<=20 goto (1)

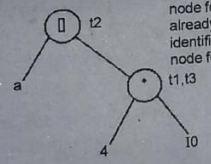
Stages in DAG Construction

(a)


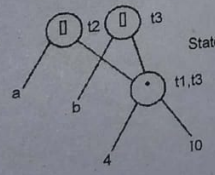
Statement (1)

(b)


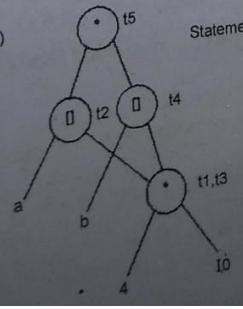
Statement (2)

(c)


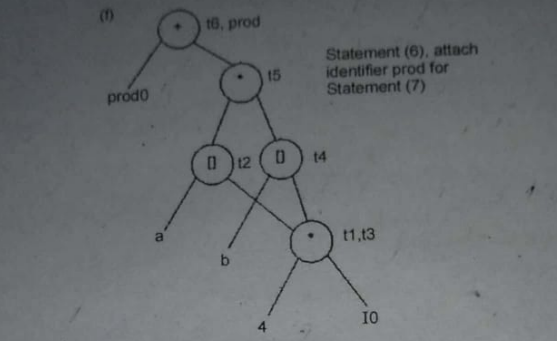
node for 4*10 exist already, hence attach identifier t3 to the existing node for Statement (3)

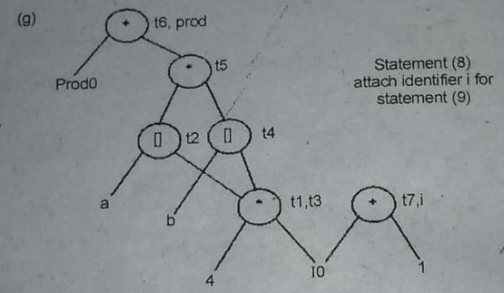
(d)


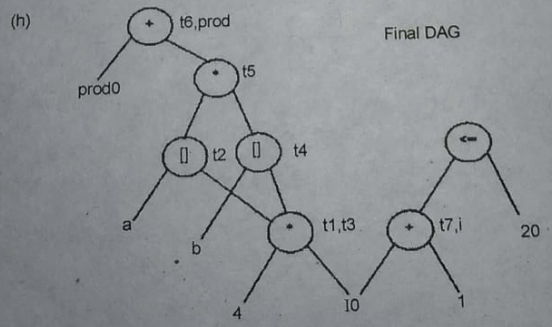
Statement (4)

(e)


Statement (%)







Q.9.(a) What do you mean by peephole optimization? Explain with example. (6)

Ans. Refer Q.1.(i) of End Term Examination 2016.

Q.9. (b) What are the issues that occurs during the code generation process? (6.5)

Ans. The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

1. Position of code generator
2. Source intermediate
3. Code program
4. Intermediate target program code

ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be : a. Linear representation such as postfix notation b. Three address representation such as quadruples c. Virtual machine representation such as stack machine code d. Graphical representations such as syntax trees and dags.

Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

The output of the code generator is the target program. The output may be : a. Absolute machine language - It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language - It allows subprograms to be compiled separately.

c. Assembly language - Code generation is made easier.

3. Memory management:

Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.

Labels in three-address statements have to be converted to addresses of instructions. For example, j :goto i generates jump instruction as follows : if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated. if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

4. Instruction selection:

The instructions of target machine should be complete and uniform.

Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

The quality of the generated code is determined by its speed and size.

The former statement can be translated into the latter statement as shown below:

5. Register allocation

→ Instructions involving register operands are shorter and faster than those involving operands in memory.

→ The use of registers is subdivided into two subproblems :

→ Register allocation – the set of variables that will reside in registers at a point in the program is selected.

→ Register assignment – the specific register that a variable will reside in is picked.

→ Certain machine requires even-odd register pairs for some operands and results. For example , consider the division instruction of the form : D x, y

where, x – dividend even register in even/odd register pair y – divisor even register holds the remainder odd register holds the quotient

6. Evaluation order : The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.