Hall Ticket Number: | | | | | | | | | | |

20CB601/20CS601/20DS601/20IT601

III/IV B. Tech (Regular/Supplementary) DEGREE EXAMINATION

May, 2025 — **Common to CB, CS, DS, & IT**

Sixth Semester — **Compiler Design**

Time: Three Hours — **Maximum:** 70 Marks

Answer question 1 compulsorily. **(14X1 = 14 Marks)**

Answer one question from each unit. **(4X14 = 56 Marks)**

			CO	BL	M
1	a)	Define language processor.	CO1	L1	1M
	b)	What is the role of the Lexical Analyzer in a compiler?	CO1	L1	1M
	c)	What is a token in the context of lexical analysis?	CO1	L1	1M
	d)	Define Bottom-Up Parsing.	CO2	L1	1M
	e)	Define Syntax-Directed Definitions (SDDs).	CO2	L1	1M
	f)	Define YACC.	CO2	L1	1M
	g)	What is three-address code?	CO3	L1	1M
	h)	Define leader	CO3	L1	1M
	i)	What are the main issues in the design of a code generator?	CO3	L1	1M
	j)	What is a flow graph?	CO3	L1	1M
	k)	What is the layout of an activation record?	CO4	L1	1M
	l)	What is an activation tree?	CO4	L1	1M
	m)	Define Scope in the context of a symbol table.	CO4	L1	1M
	n)	What is a Symbol Table?	CO4	L1	1M
		Unit-I			
2	a)	Analyze the working of Lex tool in generating lexical analyzers.	CO1	L4	7M
	b)	Consider the grammar: $E \rightarrow E + T \mid T$ $T \rightarrow T * F \mid F$ $F \rightarrow (E) \mid id$ Eliminate left recursion in each production. Rewrite the grammar.	CO1	L4	7M
		(OR)			
3		Consider the grammar: $S \rightarrow Aa \mid b$ $A \rightarrow Ac \mid Sd \mid \varepsilon$ Compute FIRST and FOLLOW for each non-terminal. Try constructing the parsing table. Identify if any conflicts exist in the table. Is it LL(1)? Why or why not?	CO1	L4	14M
		Unit-II			
4	a.	Construct Canonical LR(1) Parsing Table Grammar: $S \rightarrow A B$ $A \rightarrow a \mid \varepsilon$ $B \rightarrow b$	CO2	L3	7M
	b.	What is the difference between SDD and SDT. Explain about SDT for evaluation of an arthematic expression.	CO2	L2	7M
		(OR)			
5	a.	Construct LALR Parsing Table Grammar: $S \rightarrow A B$ $A \rightarrow a \mid \varepsilon$ $B \rightarrow b$	CO2	L3	7M
	b.	Explain about different ways to evaluate SDD.	CO2	L2	7M
		Unit-III			
6	a.	Given an example of a simple program, break it down into basic blocks and represent it using a flow graph. Illustrate the steps you would take to generate optimized code from this representation.	CO3	L3	7M
	b.	Explain about different ways to implement three address code for the given expression z = (a+b)*(c-d)+(a+b)	CO3	L2	7M
		(OR)			
7	a.	Explain and write SDT for conversion from assignment statement to Boolean expression with an example.	CO3	L3	9M
	b.	Write down different types of three address code. Explain variants of syntax trees.	CO3	L2	5M
		Unit-IV			
8	a.	What is an activation record? Illustrate the key components of an activation record.	CO4	L2	7M
	b.	Explain about implementation of different data structures in symbol table.	CO4	L2	7M
		(OR)			
9	a.	Differentiate static and dynamic memory allocation? Provide examples of when static allocation is preferable and when dynamic allocation is necessary.	CO4	L2	7M
	b.	What is activation tree? Explain with an example.	CO4	L2	7M

1a) Define language processor.

A language processor is a system software that translates programs written in high-level programming languages into machine language.

1b) What is the role of the Lexical Analyzer in a compiler?

The role of lexical analyzer is to read the source code character by character to convert into token, eliminate comment lines and whitespaces.

1c) What is a token in the context of lexical analysis?

Token is a pair containing two components:<token name, attribute value>

1d) Define Bottom-Up Parsing.

Bottom-Up Parsing: The process of deriving a string from leaf to root nodes. The decision in Bottom – Up is, which production we have to reduce next.

1e) Define Syntax-Directed Definitions (SDDs).

A Syntax Directed Definition (SDD) is a context free grammar with attributes and semantic rules. The attributes are associated with grammar symbols whereas the semantic rules are associated with productions.

1f) Define YACC.

YACC (Yet Another Compiler Compiler) is a parser generator tool which generates parse tree with tokens as input.

1g) What is three-address code?

Three-address code (TAC) is an intermediate representation in a compiler where each instruction contains at most three addresses (operands).

1h) Define leader.

A leader is the first instruction in a basic block of code.

1i) What are the main issues in the design of a code generator?

The main issues in the design of a code generator are:

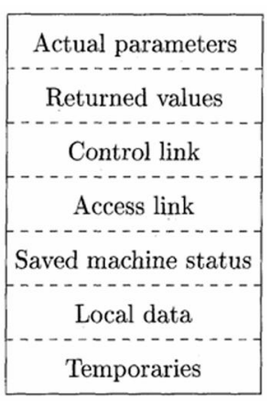
a) Input to the code generator
b) Target program
c) Instruction selection
d) Register allocation issues
e) Evaluation order

1j) What is a flow graph?

Flow graph is a directed graph in which the flow control information is added to the basic blocks.

1k) What is the layout of an activation record?

The layout of an activation record:



1l) What is an activation tree?

Activation Tree: A tree where each node represents an activation, and edges represent procedure calls.

1m) Define Scope in the context of a symbol table.

Scope = A lifetime of a variable in a particular block.

1n) What is a Symbol Table?

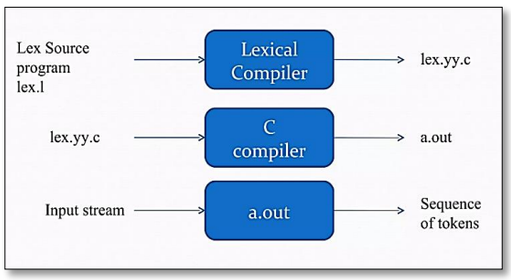
Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrences of various entities such as variable names, function names, objects, classes and interfaces etc

UNIT-I

2a) Analyze the working of Lex tool in generating lexical analyzers.

Lex (The Lexical-Analyzer Generator):

Lex is a tool or language that converts the stream of characters into tokens. The Lex tool itself is a compiler. It was written by Mike Lesk and Eric Schmidt.



A Lex program has the following form or structure:

Declarations %{… %}

translation rules %% ... %%

auxiliary functions

1. The declarations section includes declarations of variables, manifest constants and regular definitions.

2. The translation rules each have the form: Pattern {Action}

Each pattern is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C, although other languages can also be used

3. The third section holds whatever additional functions are used in the actions. Alternatively, these functions can be compiled separately and loaded with the lexical analyzer.

Lex program to recognize keywords, identifiers, relation operators and numbers

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
```

%%

{id} {printf(“%s is an identifier”, yytext);}

if {printf(“%s is a keyword”, yytext);}

“<“ {printf(“%s is less than operator”, yytext);}

“<=“ {printf(“%s is less than or equal to operator”, yytext);}

“>“ {printf(“%s is greater than operator”, yytext);}

“>=“ {printf(“%s is greater than or equal to operator”, yytext);}

“==“ {printf(“%s is less than operator”, yytext);}

“!=“ {printf(“%s is not equal to operator”, yytext);}

{number} {printf(“%s is a number”, yytext);}

%%

2b) Consider the grammar:

E → E + T | T
T → T * F | F
F → (E) | id

Eliminate left recursion in each production. Rewrite the grammar

Given Grammar:

E → E + T | T
T → T * F | F
F → (E) | id
we need to convert the left recursive grammar into right recursive grammar as follows:

A→ Aα / β

A →βA′
A′ → ε / α A′

E → E + T | T

E → TE′

E′ → ε / +T E′

T → T * F | F

T → FT′

T′ → ε / *F T′

Grammar after elimination of left recursion:

E → TE′

E′ → ε / +T E′

T → FT′

T′ → ε / *F T′

F → (E) | id

3) Consider the grammar:

S → Aa | b
A → Ac | Sd | ε

Compute FIRST and FOLLOW for each non-terminal. Try constructing the parsing table. Identify if any conflicts exist in the table. Is it LL(1)? Why or why not?

Consider the grammar:

S → Aa | b

A → Ac | Sd | ε

The grammar is in left recursive, converting it into right recursive grammar

S → Aa | b

A → SdA’ | A’

A’ → cA’ | ε

First and Follow:

Non-Terminal	FIRST ()	FOLLOW ()
S	{a, b, c}	{d, $}
A	{a, b, c, ε}	{a}
A’	{c, ε}	{a}

Parsing Table:

Non-Terminal	a	b	c	d	$
S	S → Aa	S → Aa S → b	S → Aa	S → Aa	S → Aa
A	A →SdA' A → A'	A →SdA'	A →SdA' A → A'		
A'	A' → ε		A → cA'		

Here the given grammar will not be LL(1) because in parsing table there are two productions in a single cell which violating the constraint (i.e. S → Aa | b and A → SdA' | A')

UNIT – II

4a) Construct Canonical LR(1) Parsing Table

Grammar:

S → A B
A → a | ε
B → b

Canonical LR (1):

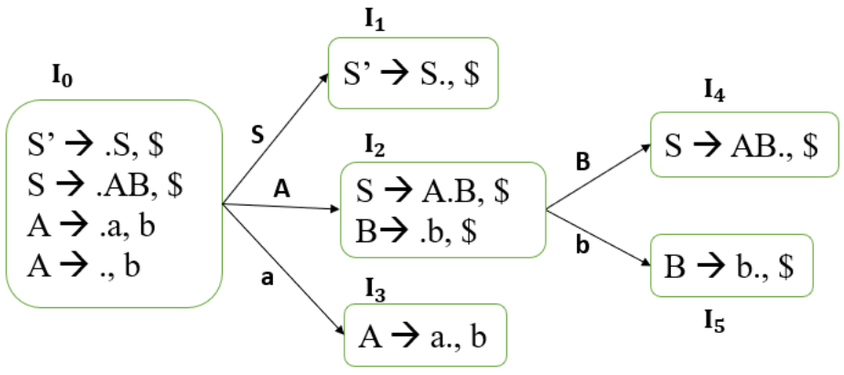
Grammar:

S → A B
A → a | ε
B → b

Augmented Grammar:

S' → S
1. S → A B
2. A → a
3. A →ε
4. B → b

Canonical collection of LR (1) items:



Parsing Table:

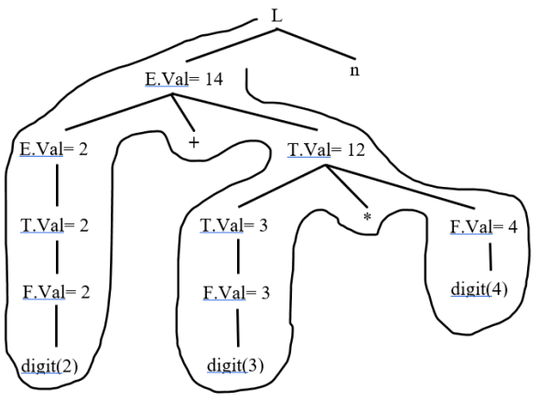
	a	b	$	S	A	B
0	Shift 3			1	2	
1			Accept			
2		Shift 5				4
3		Reduce 2				
4			Reduce 1			
5			Reduce 4			

4b) What is the difference between SDD and SDT. Explain about SDT for evaluation of an arithmetic expression.

SDD	SDT
Attribute Grammar	Translation Schemes
SDD: Specifies the values of attributes by associating semantic rules with the productions.	SDT: Embeds program fragments (also called semantic actions) within production bodies.
E → E + T E.val:= E1.val + T.val	E → E + T { print('+'); }
More Readable	More Efficient
Used to specify the non-terminals.	Used to implement S-Attributed SDD and L-Attributed SDD.
Used to know the value of non-terminals.	Used to generate Intermediate Code.
Implementation details are hidden	Implementation details are visible
Order of the evolution of parse tree not specified	Order in which semantic rules should be evaluated is specified

SDT for Evolution Expression

Production	Semantic Action
L→En	{print(E.value)}
E → E + T	{E.value = E.value + T. value}
E → T	{E.value = T. value}
T → T * F	{T.value = T.value * F. value}
T → F	{T.value = F.value}
F → digit	{F.value = digit.lexvalue}
W= 2 + 3 * 4n	



Output: 14

5a) Construct LALR Parsing Table

Grammar:

S → A B

A → a | ε

B → b

LALR (1):

Grammar:

S → A B

A → a | ε

B → b

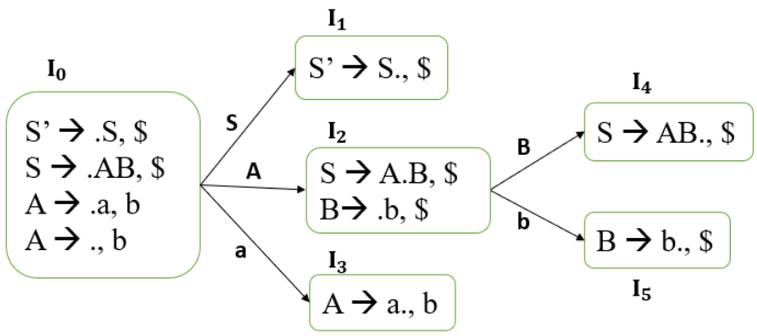
Augmented Grammar:

S' → S

1. S → A B
2. A → a
3. A →ε

4. B → b

Canonical collection of LR (1) items:



Parsing Table:

	a	b	$	S	A	B
0	Shift 3			1	2	
1			Accept			
2		Shift 5				4
3		Reduce 2				
4			Reduce 1			
5			Reduce 4			

5b) Explain about different ways to evaluate SDD.

Evolution orders for SDD's:

1. **Dependency Graphs**

Definition: A graph that shows the flow of information which helps in computation of various attribute values in a particular parse tree is called Dependency graph.

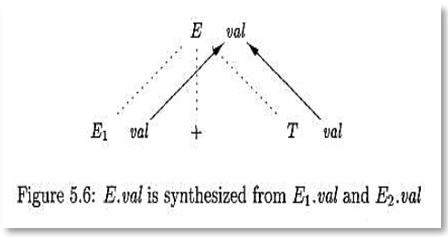


Figure 5.6: *E.val* is synthesized from E_1.*val* and E_2.*val*

- An edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules.
- **Consider the following production and rule**:

  Production Semantic Rule

  E ->E1+ T E.val= E1.val + T.val

2. Ordering the Evaluation of Attributes

- If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N. Thus, the only allowable orders of evaluation are those sequences of nodes N1, N2..., Nk such that if there is an edge of the dependency graph from Ni to Nj, then i < j. Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.
- If there is any cycle in the graph, then there are no topological sorts; that is there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort.
- The dependency graph of Fig. 5.7 has no cycles. One topological sort is the order in which the nodes have already been numbered: 1,2 ..., 9. Notice that every edge of the graph goes from a node to a higher-

numbered node, so this order is surely a topological sort. There are other topological sorts as well, such as 1,3,5,2,4,6,7,8,9.

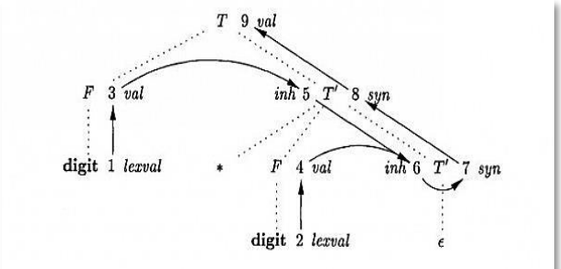


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

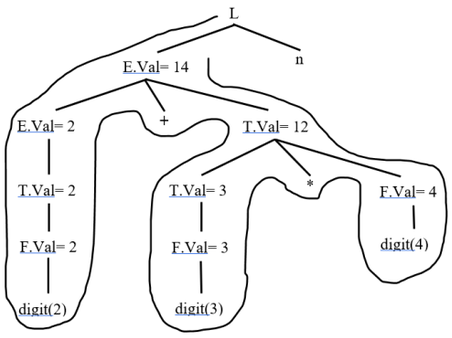
3. S-Attributed Definitions

- An SDD that involves only synthesized attributes is called S-attributed;

  In S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

Production	Semantic Rule
L→En	L.Value = E.value
E → E + T	E.value = E.value + T. value
E → T	E.value = T. value
T → T * F	T.value = T.value * F. value
T → F	T.value = F.value
F → digit	F.value = digit.lvalue
W= 2 + 3 * 4n	

- Each attribute, L.val, E.val, T.val, and F.val is synthesized. S-attributed definitions can be implemented during **bottom-up parsing**, since a bottom-up parse corresponds to **a post-order traversal**. Specifically, post-order corresponds exactly to the order in which an LR parser reduces a production body to its head.

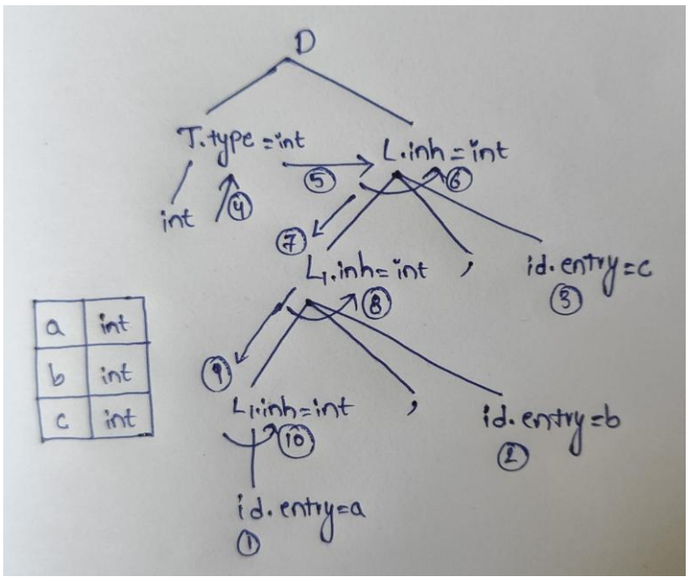


4. L-Attributed Definitions

- Uses both **inherited** and **Synthesized** attributes.
- Each inherited attribute is restricted to inherit either from parent & left Siblings only
  A→XYZ Y.S=A.S, Y·S=X.S
- Attributes are evaluated by traversing parse tree **depth first, left to right**

Production	Semantic Rule
D→TL	L.in =T.type
T → int	T.type = integer
T → real	T. type=real
L→ L_1, id	L.in=L.in
	addtype (id.entry, L.in)
L→ id	addtype (id.entry, L.in)

int a, b, c;



UNIT – III

6a) Given an example of a simple program, break it down into basic blocks and represent it using a flow graph. Illustrate the steps you would take to generate optimized code from this representation.

Basic Blocks and Flow Graphs

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

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:

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. 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.

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

B_1
```
i = 1
```

B_2
```
j = 1
```

B_3
```
t1 = 10 * i
t2 = t1 + j
t3 = 8 * t2
t4 = t3 - 88
j = j + 1
if j <= 10 goto B3
```

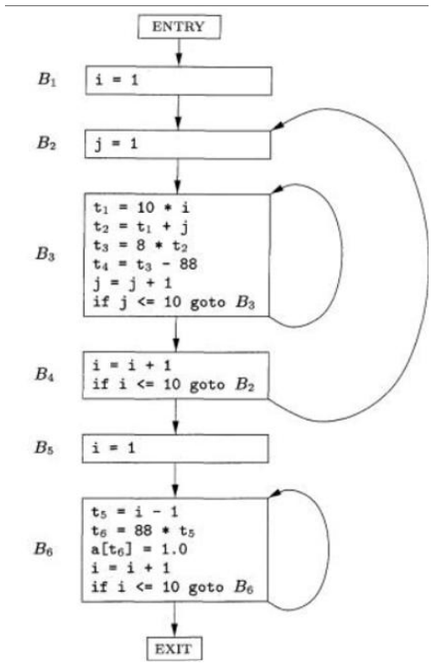
B_4
```
i = i + 1
if i <= 10 goto B2
```

B_5
```
i = 1
```

B_6
```
t5 = i - 1
t6 = 88 * t5
a[t6] = 1.0
i = i + 1
if i <= 10 goto B6
```

Flow Graph:

Flow graph is a directed graph in which the flow control information is added to the basic blocks



- We can often obtain a substantial improvement in the running time of code merely by performing local optimization within each basic block by itself.
- More thorough global optimization, which looks at how information flows among the basic blocks of a program.

Machine independent optimization

Structure preserving transformation	Algebraic Transformation
1. Common subexpression elimination (DAG) 2. Dead code elimination (DAG) 3. Renaming of temporary variables (DAG) 4. Interchange of two independent adjacent Statements (DAG) 5. Representation of array references (DAG) 6. Loop optimization (DAG) a. code motion or frequency reduction b. loop unrolling c. loop jamming	1. Constant folding 2. Copy propagation 3. Strength reduction 4. Algebraic simplification

6b) Explain about different ways to implement three address code for the given expression

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

Implementation/Representation of three address codes

1. Quadruples: A quadruple is a record structure with four fields,

which are, op, arg1, arg2 and result.

2. Triples: A triple has only three fields: op, arg1 and arg2.

3. Indirect triples: listing pointers to 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 field's 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.

Example: z = (a+b) * (c-d) + (a+b)

Three address code:

```
t1 = a+b
t2 = c-d
t3 = a+b
t4 = t1 * t2
t5 = t4 + t3
z = t5
```

Quadruples:

	Op	Arg1	Arg2	Result
0	+	a	b	t1
1	-	c	d	t2
2	+	a	b	t3
3	*	t1	t2	t4
4	+	t4	t3	t5
5	=	t5		z

Triples

- A triple has 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).

Example: z = (a+b) * (c-d) + (a+b)

Three address code:

```
t1 = a+b
t2 = c-d
t3 = a+b
t4 = t1 * t2
t5 = t4 + t3
z = t5
```

Triples:

	Op	Arg1	Arg2
0	+	a	b
1	-	c	d
2	+	a	b
3	*	(0)	(1)
4	+	(3)	(2)
5	=	z	(4)

Indirect Triples

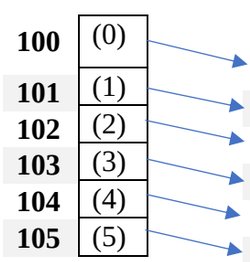
- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.

Example: z = (a+b) * (c-d) + (a+b)

Three address code:

```
t1 = a+b
t2 = c-d
t3 = a+b
t4 = t1 * t2
t5 = t4 + t3
z = t5
```

Indirect Triples:



	Op	Arg1	Arg2
0	+	a	b
1	-	c	d
2	+	a	b
3	*	(0)	(1)
4	+	(3)	(2)
5	=	z	(4)

Quadruples	Triples	Indirect triples
Advantages: Statement can be moved around	Advantages: Space is not wasted	Advantages: Statement can be moved around
Disadvantages: Too much space is wasted	Disadvantages: Statement can’t be moved around	Disadvantages: Two memory access

7a) Explain and write SDT for conversion from assignment statement to Boolean expression with an example.

SDT for Assignment Statements

- In the syntax directed translation, *assignment statement* is mainly *deals with expressions*. The *expression* can be of *type real, integer, array and records*.
- Consider the grammar

  S → id := E
  E → E1 + E2
  E → E1 * E2
  E → -E1
  E → (E1)
  E → id

The translation scheme of above grammar is given:

- *Lookup* checks for *identifier* in *symbol table.*
- The *p* returns the *entry for id.name* in the *symbol table.*
- The *Emit* function is used for *appending the three address code* to the output file.
- The *newtemp()* is a function used to generate *new temporary variables.*
- *E.place* holds the value of *E.*

Production Rule	Semantic actions

S → id :=E	{p = lookup(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

E → E1 + E2	{E.place = newtemp(); Emit (E.place = E1.place '+' E2.place) }
E → E1 * E2	{E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) }
E → -E1	{E.place = newtemp(); Emit (E.place = uminus E1.place}
E → (E1)	{E.place = E1.place}

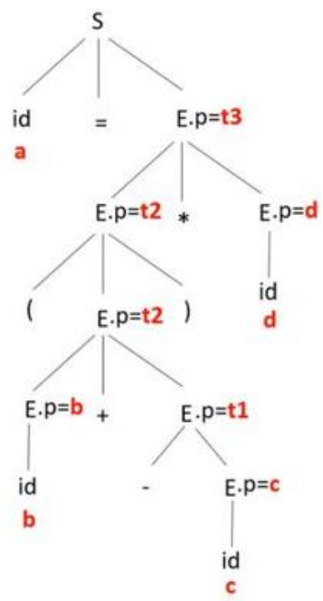
E → id	{p = lookup(id.name);
	If p ≠ nil then
	E.place = p
	Else
	Error;
	}

Production Rule	Semantic actions

S → id :=E	{p = lookup(id.name);
	If p ≠ nil then
	Emit (p = E.place)
	Else
	Error;
	}

E → E1 + E2	{E.place = newtemp(); Emit (E.place = E1.place '+' E2.place) }
E → E1 * E2	{E.place = newtemp(); Emit (E.place = E1.place '*' E2.place) }
E → -E1	{E.place = newtemp(); Emit (E.place = uminus E1.place}
E → (E1)	{E.place = E1.place}

E → id	{p = lookup(id.name);
	If p ≠ nil then
	E.place = p
	Else
	Error;
	}



Example: a = (b + -c)*d

Three Address Code:

t1 = uminus c
t2 = b + t1
t3 = t2 * d
a = t3

7b) Write down different types of three address code. Explain variants of syntax trees.

Types of Three address codes:

1. Assignment statements of the form **x = y op z**
2. Assignment instructions of the form **x = op y**
3. Copy statements of the form **x = y**
4. The unconditional jump **goto L**.
5. Conditional jumps such as **if x relop y goto L**.
6. Indexed assignments of the form **x= y[i] and x[i] = y**.
7. Address and pointer assignments of the form

   x = &y, x= *y, and *x= y.

Variants of Syntax Trees

Directed Acyclic Graph

- An important derivative of abstract syntax tree is known as Directed Acyclic Graph.

- It is used to reduce the amount of memory used for storing the Abstract Syntax Tree data structure.

Consider an expression: a + a * (b - c) + (b - c) * d;

The AST and DAG is shown in the fig below

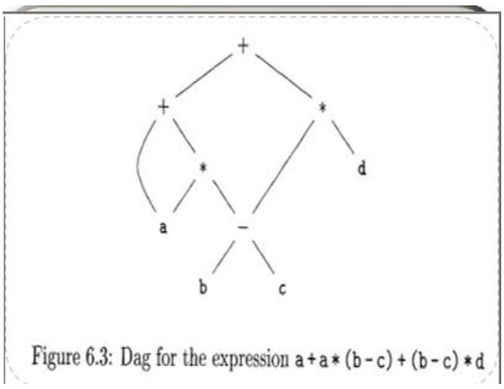
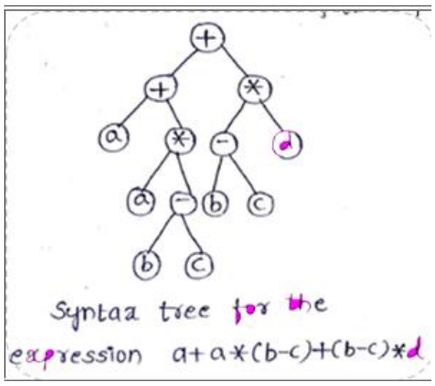


Figure 6.3: Dag for the expression a+a*(b-c)+(b-c)*d



UNIT – IV

8a) What is an activation record? Illustrate the key components of an activation record.

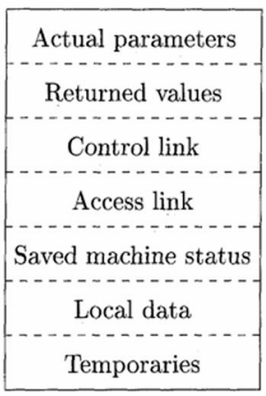
Activation Records:

The set of information needed to manage a single procedure activation (like return address, parameters, local variables).

Procedure calls and returns are usually managed by a run-time stack called the control stack.

Each live activation has an activation record (sometimes called a frame) on the control stack.

The latter activation has its record at the top of the stack.



1. Temporary values, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.

2. Local data belonging to the procedure whose activation record this is.

3. A saved machine status, with information about the state of the machine just before the call to the procedure. This information typically includes the return address and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.

4. An "access link" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.

5. A control link, pointing to the activation record of the caller.

6. Space for the return value of the called function, if any.

7. The actual parameters used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency.

8b) Explain about implementation of different data structures in symbol table.

Various Implementations (Data Structures) of Symbol Table:

There are four data structures used to implement symbol table:

1) Linear list

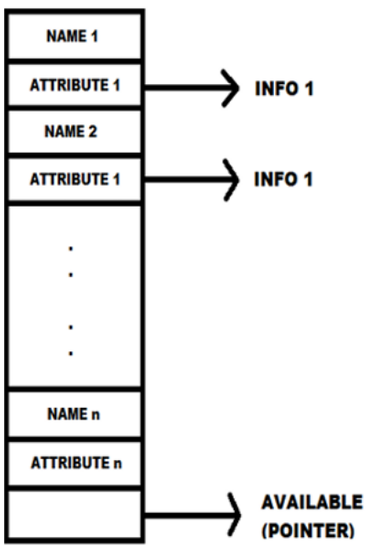
2) Self-organizing Lists

3) Search Trees

4) Hash tables

Implementation	Insert Time	Lookup Time	Disadvantages
1. Linear List			• Lookup time directly proportional to table size
i. Ordered list			
a. Arrays	O(n)	O(logn)	• Every insertion operation proceeded with lookup operation
b. Linked list	O(n)	O(n)	
ii. Unordered list			
a. Arrays	O(1)	O(n)	
b. Linked list	O(1)	O(n)	
2. Self-Organising List	O(1)	O(n)	• Poor performance when less frequent items searched
3. Search Trees	$O(\log_k n)$	$O(\log_k n)$	• We have to always keep it balanced
4. Hash Tables	O(1)	O(1)	• When there are too many collisions the time complexity increases to O(n)

1) Linear List:



• It is a simplest and easiest to implement data structure.

• Single array is used to store names and their associated information.

• New names are added to the list in the order in which they are encountered.

• To insert a new name, the list is scanned down to make sure that it is not already there. If not then add it otherwise an error message i.e., Multiple declared name.

• When the name is located, the associated information can be found in words following next.

• To retrieve information about a name, we search from the beginning of the array up to the position marked by AVAILABLE pointer, which indicates the beginning of the empty portion of array.

• If AVAILABLE pointer is reached without finding NAME, we have a fault - the use of an undefined name.

• One advantage of list organization is that the minimum possible space is taken in simple compiler

2) Self-Organizing Lists:

• Add a LINK field to each record, and we search the list in the order indicated by the LINK's.

• Substantial fraction of searching time is saved at the cost of little extra cost.

• In Fig. 9.7(a) the order is NAME3, NAME1, NAME4, NAME2.

• Names that are referenced frequently will tend to be at the front of the list to found it quickly.

• Preferred: when small set of names is heavily used

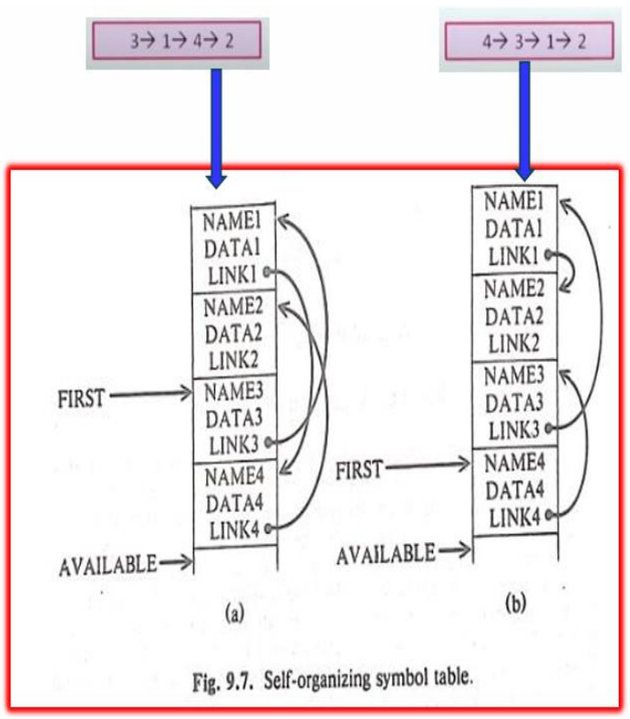


Fig. 9.7. Self-organizing symbol table.

3) Search Trees:

It is a more efficient approach to symbol table organization. Here we add two link fields LEFT and RIGHT to each record

Left Child	Name of Symbol	Information	Right Child

Following algorithm is used to look for NAME in a binary search tree where p is initially a pointer to the root.

Binary tree search routine

1. while P ≠ null do
2. if NAME = NAME(P) then /* NAME found take action on success*/
3. else if NAME < NAME(P) then
   P:= LEFT(P) /* visit left child*/
5. else /* NAME (P) <NAME*/
   P:= RIGHT(p) /*visit right child*/

4) Hash Tables:

- Basic hashing schema is shown in figure (9.9).
- Two tables: hash table and a storage table are used.
- The hash table consists of k words numbered 0,1,2…, k-1. These words are pointers into the storage enable to the heads of k separate linked lists I (some lists may be empty).
- Each record in the symbol table appears on one of these lists.
- To determine whether NAME is in the symbol table, we apply NAME as a hash function h such that h(NAME) is an integer between 0 to k-1.

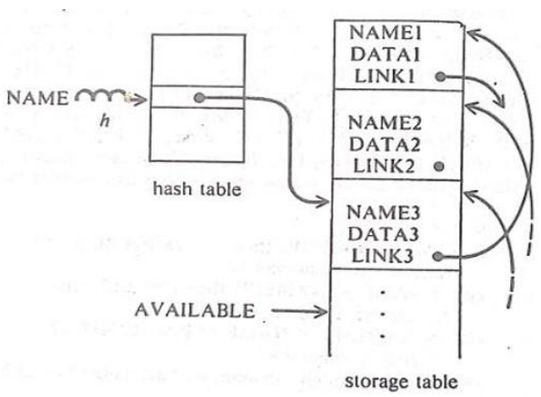


Fig. 9.9 Hash table.

9a) Differentiate static and dynamic memory allocation? Provide examples of when static allocation is preferable and when dynamic allocation is necessary.

Static Memory Allocation	Dynamic Memory Allocation
Allocation is done at compile time	Allocation is done at run time
Recursion is not possible	Recursion is possible
Dynamic Data Structures are not supported	Dynamic Data Structures are supported
Activation can have permanent life time	Activation can have arbitrary life time
Language like C (using arrays like int arr[100];)	Language like C (using malloc(), calloc(), etc.)
Automatically released when program/function ends	Must be manually freed using free()

Static Allocation – When Preferable:

When memory size is known and constant throughout the program.

Eg: int numbers[50]; // memory for 50 integers is allocated at compile time

Dynamic Allocation – When Necessary:

- When memory size is unknown at compile time or depends on user input.
- Required in data structures like linked lists, trees …

Eg:
```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int *arr = (int *)malloc(n * sizeof(int)); // dynamic allocation
    free(arr); // release memory
    return 0; }
```

9b) What is activation tree? Explain with an example

Activation Trees:

A tree where each node represents an activation, and edges represent procedure calls.

• During run-time organization, an activation tree is used to represent the hierarchical relationships between active procedures (functions or methods) during the execution of a program.

• Each time a procedure is called, a new activation (instance) of that procedure is created.

• The activation tree shows how these procedure calls relate to each other.

• Root: Represents the first (main) program or method call.

• Children: Represent procedures that are called by their parent procedure.

• Leaf Nodes: Procedures that do not call any other procedures.

Important Properties:

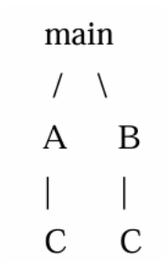
• Activations follow a stack discipline (last called, first finished).

• No two activations of the same procedure overlap unless recursion is used.

Simple Example:

Suppose we have the following pseudo-code

```
main ()
{
        A();
        B();
}
A ()
{
        C();
}
B ()
{
        C();
}
```



• main has two children: A and B.

• A has a child C.

• B has a child C.

•Execution starts at the root (main).

•Execution proceeds depth-first: A node calls its child and waits until the child finishes.

•After a child finishes, control returns to the parent.