20CB/CS/DS/IT601

Hall Ticket Number:

III/IV B.Tech (Regular) DEGREE EXAMINATION

July, 2023 **Common to CB,CS,DS &IT Branches**

Sixth Semester **Compiler Design**

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

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

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

1	a)	Define Language processor.	CO1	L1	1M
	b)	Give the key difference between a lexeme and token.	CO1	L2	1M
	c)	Mention two limitations of top down parsing.	CO1	L1	1M
	d)	What is left factoring?	CO2	L1	1M
	e)	List various types of conflicts possible in bottom up parsing.	CO2	L1	1M
	f)	Why LR parser is also called as a canonical parser?	CO2	L2	1M
	g)	Obtain syntax tree for x = (a+b)*(c+d)-e*f	CO3	L1	1M
	h)	List various forms of three address code.	CO3	L1	1M
	i)	Define symbol table.	CO4	L1	1M
	j)	Name two limitations of stack allocation strategy.	CO4	L1	1M
	k)	Define basic block.	CO3	L1	1M
	l)	Define activation tree.	CO4	L1	1M
	m)	List various compiler construction tools.	CO1	L1	1M
	n)	Consider the grammar S →aS | a. Identify handle to derive input string aa.	CO2	L1	1M
		Unit-I			
2	a)	Explain why analysis portion of the compiler is separated into lexical analysis and syntax analysis phase.	CO1	L2	4M
	b)	Explain the phases of the compiler with a neat sketch with an example	CO1	L2	10M
		(OR)			
3	a)	Consider the CFG with productions S → a | (L), L → L, S | S. (i) Is the grammar suitable for predictive parser? (ii) Do the necessary changes to make it suitable for LL(1) parser. (iii) Compute first and follow sets for each non-terminal.	CO1	L3	6M
	b)	Obtain the parsing table for 3(a) and hence show the sequence of moves made by the predictive parser to derive the input string **(a,(a,a))**.	CO1	L4	8M
		Unit-II			
4	a)	Construct SLR parsing table for the following grammar S → L = R | R L →*R | id R → L	CO2	L3	10M
	b)	Apply the SLR parsing table obtained in 4(a) and show the sequence of moves carried by the SLR parser to derive the input string ***id = **id** and conclude whether the parser accepts the string or not ?	CO2	L3	4M
		(OR)			
5	a)	What is the need for syntax directed translation. Differentiate between synthesized and inherited attributes.	CO2	L2	7M
	b)	Obtain SDT scheme to evaluate the given expression **2*3+4**.	CO2	L3	7M
		Unit-III			
6	a)	What is a basic block and flow graph? Explain how can we identify basic blocks and generate a flow graph with suitable example.	CO3	L2	7M
	b)	Give SDT scheme for the Boolean expressions and generate three address code for **a>b && c<d**	CO3	L3	7M
		(OR)			
7	a)	Explain different ways to implement three address codes and give that to the example a=b*(-c) + b*(-c).	CO3	L2	7M
	b)	Generate code for the arithmetic expression z = (u+v)-((w+x)-y) by applying code generation algorithm. Specify the memory required in memory words for the generated code.	CO3	L3	7M
		Unit-IV			
8	a)	Explain scope representation in symbol table.	CO4	L2	4M
	b)	Explain in detail different storage allocation strategies	CO4	L2	10M
		(OR)			
9	a)	Discuss various data structures used to implement a symbol table.	CO4	L3	10M
	b)	Describe the general structure of an activation record. Explain the purpose of each item in the activation record	CO4	L2	4M

Scheme of evaluation

1.

a. Define Language processor. 1M
b. Give the key difference between a lexeme and token. 1M
c. Mention two limitations of top down parsing. 1M
d. What is left factoring? 1M
e. List various types of conflicts possible in bottom up parsing. 1M
f. Why LR parser is also called as a canonical parser? 1M
g. Obtain syntax tree for x = (a+b)*(c+d)-e*f 1M
h. List various forms of three address code. 1M
i. Define symbol table. 1M
j. Name two limitations of stack allocation strategy. 1M
k. Define basic block. 1M
l. Define activation tree. 1M
m. List various compiler construction tools. 1M

2.

a. Explain why analysis portion of the compiler is separated into lexical analysis and syntax analysis phase.(4M)

Three reasons ----4M

b. Explain the phases of the compiler with a neat sketch with an example. 10M

Listing phase names correctly-----1M

Figure------1M

Each phase explanation with example----1M (1*8=8)

3.

a. Consider the CFG with productions S → a | (L), L → L, S | S. 6M

i. Is the grammar suitable for predictive parser? ---- 1M
ii. Do the necessary changes to make it suitable for LL(1) parser.----- 2M
iii. Compute first and follow sets for each non-terminal. ----- 3M

b. Obtain the parsing table for 3(a) and hence show the sequence of moves made by the predictive parser to derive the input string **(a,(a,a))**. 8M

Parsing table : 5M
Parser moves for string 3M

4.

a. Construct SLR parsing table for the following grammar 10M
S → L = R | R
L →*R | id
R → L

LR(0) Item sets -----6M
SLR prasing table----4M

b. Apply the SLR parsing table obtained in 4(a) and show the sequence of moves carried by the SLR parser to derive the input string ***id = **id** and conclude whether the parser accepts the string or not ? 4M

Parser moves with correct showing of error and jusrification-----4M

5.

a. What is the need for syntax directed translation. Differentiate between synthesized and inherited attributes. 7M

STD definition with an example----3M
Differentiation of attributes---------4M

b. Obtain SDT scheme to evaluate the given expression **2*3+4**. 7M

SDT scheme---5M
Annotated parse for evaluation ----2M

6.

a. What is a basic block and flow graph? Explain how can we identify basic blocks and generate a flow graph with suitable example. 7M

Basic block & flow graph definition----2M

Algorithm for basic blocks construction—2M
Example and basic blocks and flow graph construction--3M(consider any example)

b. Give SDT scheme for the Boolean expressions and generate three address code for **a>b && c<d** **7M**

STD scheme for Boolean expressions with backpaching---4M
Generating code to example---3M

7.

a. Explain different ways to implement three address codes and give that to the example a=b*(-c) + b*(-c). 7M

3 different ways explanation------4M
For example----------3M

b. Generate code for the arithmetic expression z = (u+v)-((w+x)-y) by applying code generation algorithm. Specify the memory required in memory words for the generated code. 7M

Three address statements----1M
Code generation alogorithm---2M
Target code generation---------4M

8.

a. Explain scope representation in symbol tables. 4M

Explanation-----2M
Example--------2M

b. Explain in detail different storage allocation strategies. 10M

3 strategies explanation with example----(3+4+3)

9.

a. Discuss various data structures used to implement symbol table. 10M

3 data structures with example explanation----(3+3+4)

b. Describe the general structure of an activation record. Explain the purpose of each item in the activation record. 4M

Definition------1M
Figure----------1M
Explanation of fields-----2M

1.

a. **Define Language processor.** **1M**

Language processor is a software program that translates the program codes into machine codes.

b. **Give the key difference between a lexeme and token.** **1M**

A token is a group of characters having collective meaning: typically a word or punctuation mark, separated by a lexical analyzer and passed to a parser. A lexeme is an actual character sequence forming a specific instance of a token, such as num.

c. **Mention two limitations of top down parsing.** **1M**

Cannot handle left recursive grammars
Back tracking

d. **What is left factoring?** **1M**

The process of eliminating common prefixes in the grammar production to eliminate backtracking is called left factoring.

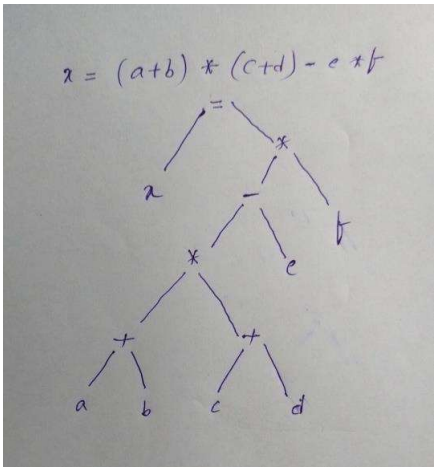
e. **List various types of conflicts possible in bottom up parsing.** **1M**

1. Shift reduce conflict
2. Reduce reduce conflict

f. **Why LR parser is also called as a canonical parser?** **1M**

If we fail to construct CLR parser for the grammar then the grammar is not LR. Therefore LR parser is also called as CLR .

g. **Obtain syntax tree for x = (a+b)*(c+d)-e*f** **1M**



h. **List various forms of three address code.** **1M**

x= y op z where op is any arithmetic operator
x=op y where op is unary operator
x=y[i]
if a>b goto label
goto label

i. **Define symbol table.** **1M**

It is a data structure used to store scope and binding information about the names in source program.

j. **Name two limitations of stack allocation strategy.** **1M**

Stack memory is very limited.
Random access is not possible.

k. **Define basic block.** **1M**

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

l. **Define activation tree.** **1M**

The flow of control between procedures can be depicted by using a tree called an activation tree.

m. **List various compiler construction tools.** **1M**

Scanner generators like lex and flex
Parser generators like YACC

n. **Consider the grammar S →aS | a. Identify handle to derive input string aa.** **1M**

aa
aS S→ a is handle
S S→aS is handle

2.

a. Explain why analysis portion of the compiler is separated into lexical analysis and syntax analysis phase. (4M)

Reasons for separating the analysis phase of the compiler into lexical analysis and syntax analysis:

1. Simple to design is most important consideration
2. Compiler efficiency is improved
3. Compiler portability can be enhanced

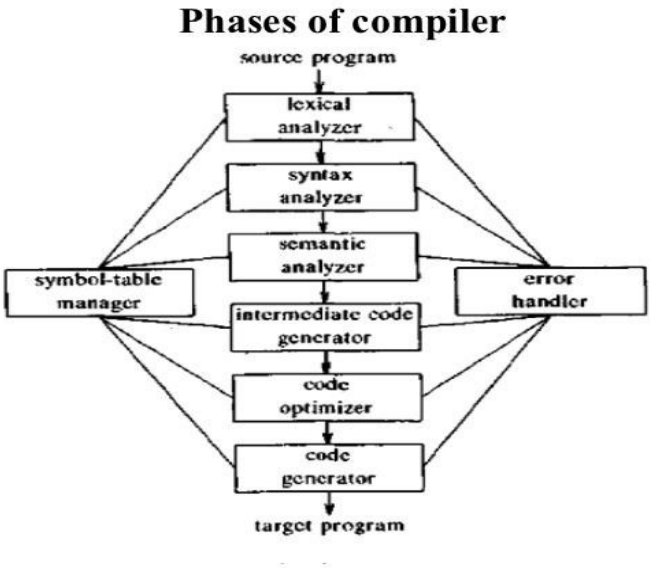
b. Explain the phases of the compiler with a neat sketch with an example. (10M)

- The process of converting the source code into machine code involves several phases , which are collectively known as the phases of a compiler. Phase is a logically cohesive operation which takes one representation of the language as input and produces another representation of the language as output.

The phases are

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate code generation
5. Code optimization
6. Code generation

In addition to these phases there are to additional modules which interact with all the phases. Those are
Symbol table
Error handler



Lexical analysis:

It is also called scanning or linear analysis .
It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language.
It reads the source program character by character from left to right and group them into lexemes.
For each lexeme lexical analyzer generates token.
Lexeme is actual stream of characters.
Token a sequence of characters with logical meaning.
To generate tokens from lexemes it uses regular expressions.
A token is represented by a pair
<token name, token value or token attribute>
It also removes lexical errors (e.g., erroneous characters), comments, and white space.

Example:

Position= initial + rate * 60
Lexemes are
Position

=
Initial
+
Rate
*
60

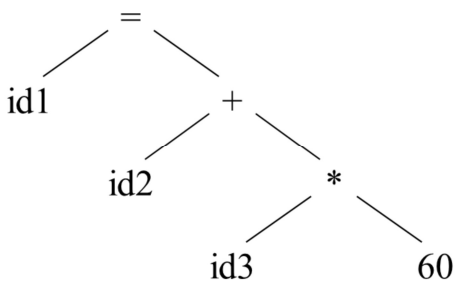
Tokens are

Position : <ID,1>
= : <=>
Initial : <ID,2>
+ :<+>
Rate :<ID,3>
* :<*>
60 :<NUM,60>

Syntax analysis:

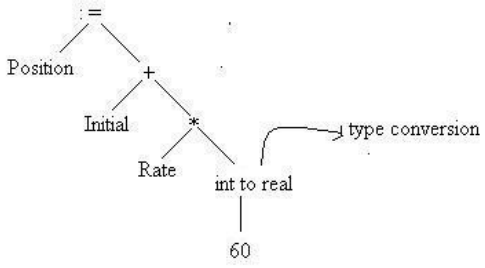
It is also called parsing.

The parser groups the tokens returned by lexical analyzer into parse tree or syntax tree.
The grouping is hierarchical in nature.
The syntax tree may not be constructed in the physical sense.
The parser checks if the expression made by the tokens is syntactically correct.
Parsing is the process of determining if a string of tokens can be generated by the grammar.
Syntax tree for the tokens given by the lexical analyzer is
id1 = id2 + id3*60



Sematic analysis:

This phase checks the source program for semantic errors.
This phase gathers type information.
An important aspect of semantic analysis is type checking.
An error is generated whenever a real number is used to index an array.
The output of the semantic analysis for the above example is:



Intermediate code generation:

This phase transforms the syntax tree into explicit intermediate representation.

The intermediate representation can have different forms.
One of them is three-address code.
Intermediate code is machine independent.
Three-address code for the above syntax tree is

temp1 := inttofloat(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3

Code optimization:

This phase attempts to improve the intermediate code so that faster running machine code will result.

For the above example the optimized code is
temp1 = id3* 60.0
id1 = id2 + temp1
It transforms the code so that it consumes fewer resources and produces more speed.
The meaning of the code being transformed is not altered.
Optimization can be categorized into two types: machine-dependent and machine-independent.

Code generation:

This code generates either assembly code or re-locatable machine code.

The intermediate instructions are translated into a sequence of assembly language or machine instructions.
For the above example the assembly code is

```
MOV  id3, R2

MUL  #60.0, R2
MOV  id2, R1
ADD  R2, R1
MOV R1, id1
```

Symbol table:

It is a data structure used to store scope binding information about the names in the source program.

These names are used to identify various program elements like variables, constants, procedures and labels of the statements of the source program.
During lexical analysis phase symbol table is created.
During the first two phases the information about the names is entered into the symbol table
In remaining phases, the information stored in the symbol table is used by the compiler.
When a new name is identified by the lexical analyzer a new entry is created in the symbol table.
A symbol table must have an efficient mechanism for accessing information stored in it.

Error handler:

The main function of the error handler is detection and reporting of errors in the source program.

It invokes error recovery strategy to continue with the compilation process.
Compiler can handle three types of errors.

- Syntax errors
- Semantic errors

Syntax errors:

- It is an error due to grammatical mistakes in the sentence.
- Example: if a>b)

Semantic errors:

- It is an error that the given sentence has type casting errors.
- int a = "20"

During the runtime we may get errors. Those errors are called fatal errors.
Examples: Memory insufficient and segmentation fault.

3.

a. **Consider the CFG with productions S → a | (L), L → L, S | S.** **(6M)**

i. Is the grammar suitable for predictive parser?

The given grammar is left recursive grammar. So it is not suitable for predictive parser. Because top down parsers like predictive parsers can not handle left recursive grammars.

ii. Do the necessary changes to make it suitable for LL(1) parser.

From the above grammar we eliminate the left recursion then the grammar will be LL(1).
The grammar after eliminating the left recursion is

S→ a | (L)
L → SL'
L'→ ,SL' | ε

iii. Compute first and follow sets for each non-terminal.

FIRST(S)={a, (}
FISRT(L)={a, (}
FISRT(L')={ ε, ,}
FOLLOW(S)={$, , ,)}
FOLLOW(L)={) }
FOLLOW(L')={) }

b. **Obtain the parsing table for 3(a) and hence show the sequence of moves made by the predictive parser to derive the input string (a,(a,a)).** **(8M)**

Predictive parsing table:

Nonterminals	terminals

	(	)	a	,	$
S	S→(L)		S→ a		
L	L → SL'		L → SL'		
L'		L' → ε		L'→ ,SL'	

Parser moves to recognize its input (a,(a,a)):

Stack	Input	Output
S$	(a,(a,a))$	
(L)$	(a,(a,a))$	S→(L)
L)$	a,(a,a))$	POP
SL')$	a,(a,a))$	L→SL'
aL')$	a,(a,a))$	S→a
L')$	,(a,a))$	POP
,SL')$	,(a,a))$	L'→,SL'
SL')$	(a,a))$	POP
(L)L')$	(a,a))$	S→(L)
L)L')$	a,a))$	POP
SL')L')$	a,a))$	L→SL'
aL')L')$	a,a))$	S→ a
L')L')$	,a))$	POP
,SL')L')$	,a))$	L'→,SL'
SL')L')$	a))$	POP
aL')L')$	a))$	S→a
L')L')$	))$	POP
)L')$	))$	L' → ε
L')$	)$	POP
)$	)$	L' → ε
$	$	accept

4.

a. Construct SLR parsing table for the following grammar. (10M)

S → L = R | R
L →*R | id
R → L

Augmented grammar:

0. S' →S
1. S → L=R
2. S → R
3. L → *R
4. L → id
5. R → L

I0:

S′ → · S
S → · L = R
S → · R
L → · * R
L → · id
R → · L

I1:

S′ → S ·

I2:

S → L · = R
R → L ·

I3:

S → R ·

I4:

L → * · R
R → · L
L → · * R
L → · id

I5:

L → id ·

I6:
S → L = · R
R → · L
L → · * R
L → · id

I7:
L → * R ·

I8:
R → L ·

I9:
S → L = R ·

The FIRST and FOLLOW sets are as follows.

FIRST(S) = {*, id}
FIRST(L) = {*, id}
FIRST(R) = {*, id}

FOLLOW(S) = {$}
FOLLOW(L) = {$, =}
FOLLOW(R) = {$, =}

SLR parsing table:

state	action				goto		
	=	*	Id	$	S	L	R
0		S4	S5		1	2	3
1				accept			
2	S6/r5			R5			
3				R2			
4		S4	S5			8	7
5	R4			R4			
6		S4	S5			8	7
7	R3			R3			
8							
9	R1			R1			

b. Apply the SLR parsing table obtained in 4(a) and show the sequence of moves carried by the SLR parser to derive the input string *id = **id and conclude whether the parser accepts the string or not ? (4M)

Paser actions:

Stack	Input	output
0	*id=**id	shift
0 * 4	Id=**id	shift
0 * 4 id 5	=**id	Reduce by L → id
0 * 4 L 8	=**id	**error**

While parsing the string *id=**id we may get an error. And the parsing table also contains shift reduce conflicts. There for the given grammar is not SLR grammar. The string is not recognized by the SLR parser.

5.

a. **What is the need for syntax directed translation. Differentiate between synthesized and inherited attributes? (7M)**

The syntax-directed translation scheme is beneficial because it allows the compiler designer to define the generation of intermediate code directly in terms of the syntactic structure of the source language.
SDT is the generalized CFG in which each grammar symbol is associated with set of attributes and each production is associated with its semantic action.

S.NO	Synthesized Attributes	Inherited Attributes
1.	**An attribute is said to be Synthesized attribute if its parse tree node value is determined by the attribute value at child nodes.**	**An attribute is said to be Inherited attribute if its parse tree node value is determined by the attribute value at parent and/or siblings node.**

2.	The production must have non-terminal as its head.	The production must have non-terminal as a symbol in its body.
3.	A synthesized attribute at node n is defined only in terms of attribute values at the children of n itself.	A Inherited attribute at node n is defined only in terms of attribute values of n's parent, n itself, and n's siblings.
4.	It can be evaluated during a single bottom-up traversal of parse tree.	It can be evaluated during a single top-down and sideways traversal of parse tree.
5.	Synthesized attributes can be contained by both the terminals or non-terminals.	Inherited attributes can't be contained by both, It is only contained by non-terminals.
6.	Synthesized attribute is used by both S-attributed SDT and L-attributed SDT.	Inherited attribute is used by only L-attributed SDT.
7.	EX:- E.val -> F.val E val ↑ F val	EX:- E.val = F.val E val ↓ F val

b. **Obtain SDT scheme to evaluate the given expression 2*3+4.** **(7M)**

Grammar for arithemetic expressions is
E→ E+T | T
T→ T*F | F
F→ id | Num

E -> E+T { E.val = E.val + T.val then print (E.val)}

E → T { E.val = T.val}

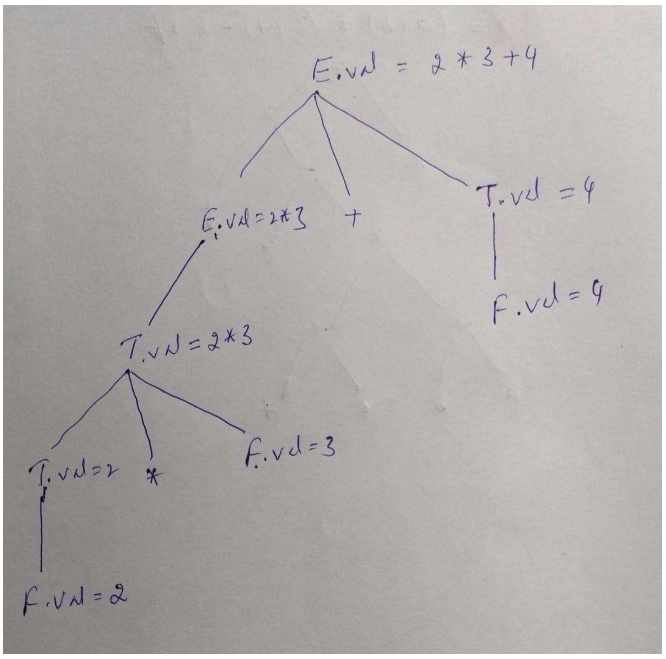
T -> T*F { T.val = T.val * F.val}

T → F { T.val = F.val}

F -> id {F.val = id.name}

F → Num {F.Val= Num.Val}

Annotated parse tree to evaluate the expression is:



6.

a. **What is a basic block and flow graph? Explain how can we identify basic blocks and generate a flow graph with suitable example.** **(7M)**

A basic block is a sequence of consecutive three address statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.
Flow graph is a set of basic blocks with basic blocks as nodes and control flow edges between two blocks Bi and Bj iff there is a possible control flow from last statement of the basic block Bi to th first statement of the basic block Bj.
Algorithm: Instructions from intermediate code which are leaders are determined. Following are the rules used for finding a leader:

1. The first three-address instruction of the intermediate code is a leader.

2. Instructions that are targets of unconditional or conditional jump/goto statements are leaders.

3. Instructions that immediately follow unconditional or conditional jump/goto statements are considered leaders.

For each leader thus determined its basic block contains itself and all instructions up to excluding the next leader.

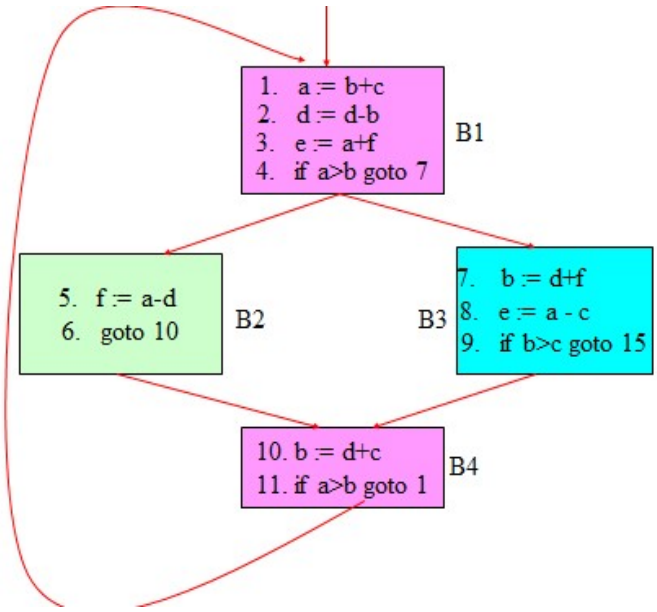
Example:

1. a := b+c
2. d := d-b
3. e := a+f
4. if a>b goto 7
5. f := a-d
6. goto 10
7. b := d+f
8. e := a-c
9. if b>c goto 15
10. b := d+c
11. if a>b goto 1

Identification of leaders:

1 , 5, 7 and 10 are leaders.

Basic blocks and its flow graph is shown below for the example.



b. **Give SDT scheme for the Boolean expressions and generate three address code for a>b && c<d** **(7M)**

Grammar for boolean expressions is:

E→ id1 relop id2
E→E OR M E
E→ E AND M E
E→NOT E
E→(E)
M→ ε

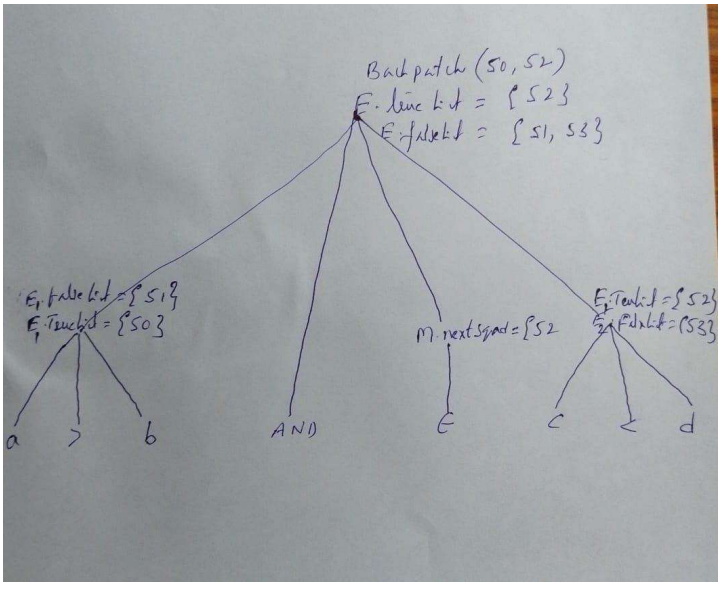
SDT Scheme is:

$E \rightarrow E_1$ or M E_2
{ BACKPATCH (E_1.FALSE, M. QUAD);
E.TRUE := MERGE(E_1.TRUE, E_2.TRUE);
E.FALSE :=E_2.FALSE }
$E \rightarrow E_1$ and M E_2
{ BACKPATCH(E_1.TRUE, M.QUAD);
E.TRUE :=E_2.TRUE;
E.FALSE :=MERGE(E_1.FALSE, E_2.FALSE) }
$E \rightarrow$ not E_1
{ E.TRUE := E_1.FALSE;
E.FALSE :=E_1.TRUE }
$E \rightarrow (E_1)$
{ E.TRUE :=E_1.TRUIE;
E.FALSE := E_1 .FALSE }

$E \rightarrow id_1$ relop id_2
{ E.TRUE :=MAKELIST(NEXTQUAD);
E.FALSE :=MAKELIST(NEXTQUAD + 1);
GEN(if id_1.PLACE relop id_2.PLACE goto-)
GEN(goto-)) }

M → ε
{ M.QUAD := NEXTQUAD }

Example: a>b && c<d
Three address code:
50. if a>b goto ----(52)
51. goto ---
52. if c<d goto----
53. goto ---



7.

a. **Explain different ways to implement three address codes and give that to the example a=b*(-c) + b*(-c).**

(7M)

Implementation of Three Address Code –

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple – It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example – Consider expression a = b * – c + b * – c. The three address code is:

```
t1 = uminus c
t2 = b * t1
t3 = uminus c
t4 = b * t3
t5 = t2 + t4
a = t5
```

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

2. Triples – This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example – Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

3. Indirect Triples – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example – Consider expression a = b * – c + b * – c

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

List of pointers to table

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

b. **Generate code for the arithmetic expression z = (u+v)-((w+x)-y) by applying code generation algorithm. Specify the memory required in memory words for the generated code. (7M)**

Three address code for z=(u+v)-((w+x)-y)

a=u+v
b=w+x
c=b-y
z=a-c

Code for the above three address code is

```
MOV u,R0        u in R0
ADD  v,R0         a in R0
MOV  w,R1       w in R1
ADD x, R1       b in R1
SUB y,R1        c in R1
SUB R1, R0      z in R0
MOV R0,z        z in Memory and in R0
MOV R1,c        c in memory and in R1
```

8.

a. **Explain scope representation in symbol table. (4M)**

Scope: Every name possesses a region of validity within the source program, called the "scope" of that name . The rules governing the scope of names in a block-structured language are as follows :

a. A name declared within a block B is valid only within B .
b. If block B2 is nested within B1, then any name that is valid for B1 is also valid for B2, unless the identifier for that name is re-declared in B2.

The scope information can be stored using a stack of symbol tables. An individual symbol table for each scope.

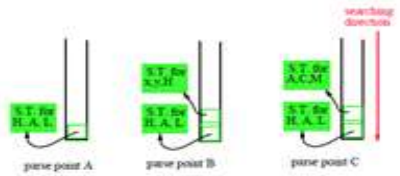
a. Use a stack to maintain the current scope.
b. Search top of stack first.
c. If not found, search the next one in the stack.
d. Use the first match.
e. Note: a popped scope can be destroyed in a one pass compiler, but it must be saved in multi-pass compiler.

Representing Scope Information

An individual symbol table for each scope.

- Use a stack to maintain the current scope.
- Search top of stack first.
- If not found, search the next one in the stack.
- Use the first match.
- Note: a popped scope can be destroyed in a one pass compiler, but it must be saved in a multi-pass compiler.

```
main()
{ /* open a new scope */
   int H,A,L; /* parse point A */
   ...
   { /* open another new scope */
      float x,y,H; /* parse point B */
      ...
      /* x and y can only be used here */
      /* H used here is float */
      ...
   } /* close an old scope */
   ...
   /* H used here is integer */
   ...
   { char A,C,M; /* parse point C */
      ...
   }
}
```



b. **Explain in detail different storage allocation strategies.** **(10M)**

The different ways to allocate memory are:

1. Static storage allocation
2. Stack storage allocation
3. Heap storage allocation

Static storage allocation

- In static allocation, names are bound to storage locations at compile time..
- Bindings do not change at runtime
- Every time a procedure is activated its names are bound to same storage locations.
- From the type of the name, the compiler determines the amount of memory to set aside for that name.
- The address of this storage consists of an offset from the end of the activation record for the procedure.
- The compiler must decide where the activation records go relative to the target code and to one another.
- Once this decision is made, the position of the activation record, and the storage of each name in the record is fixed.

Limitations with static allocation strategy:

- The size and position of data objects should be known at compile time.
- Recursive procedures are restricted, because all activations of a procedure use the same bindings for locals.
- Data structures cannot be created dynamically, since there is no mechanism form storage allocation at runtime.

Example:

```
Procedure consume()
Begin
    Character *50 Buf
    Integer Next
    Character c
    c= produce()
end;
```

```
procedure produce()
begin
        character * 80 Buf
        integer Next
        .........
        .....
End;
```

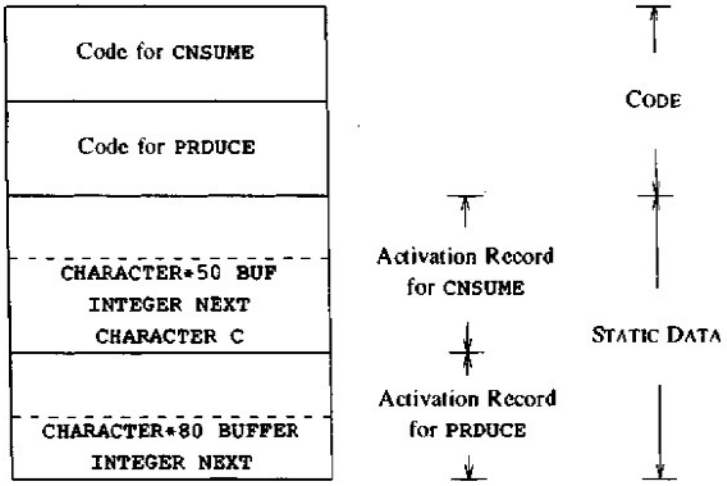


Fig. 7.11. Static storage for local identifiers of a Fortran 77 program.

```
Example:
PROGRAM sort(input,output);
VAR a : array[0..10] of Integer;
PROCEDURE readarray;
VAR i : Integer;
 BEGIN
for i:= 1 to 9 do read(a[i]);
 END;
FUNCTION partition(y,z : Integer):  Integer;
VAR i,j,x,v : Integer;
 BEGIN
…
END;
PROCEDURE quicksort(m,n : Integer);  VAR i : Integer;
BEGIN
if (n > m) then BEGIN
i := partition(m,n);  quicksort(m, i-1);  quicksort(i+1,n)
END
END;

BEGIN        /* of main */
a[0] := -9999;
a[10] := 9999;
readarray;
quicksort(1,9)
END.
```

Figure 7.13 shows the activation records that are pushed onto and popped from the run-time stack as control flows through the activation tree. Dashed lines in the tree go to activations that have ended. Execution begins with an activation of procedure s. When control reaches the first call in the b d y of s, procedure r is activated and its activation record is pushed onto the stack. When control returns from this activation, the record is **popped** leaving just the record for s in the stack. In the activation of s, control then reaches a **call** of q with actuals **1** and 9,and an activation record is allocated on top of the stack fix an activation of q, Whenever control is in an activation, its activation record is at the top of the stack. Several activations occur between the last two snapshots in Fig. In the last snapshot in Fig. activations p(1, 3) and q(1, 0) have begun and ended during the lifetime of q(1 , 3) , so their activation records have come and gone from the stack, leaving the activation record for q(1 , 3) on top.

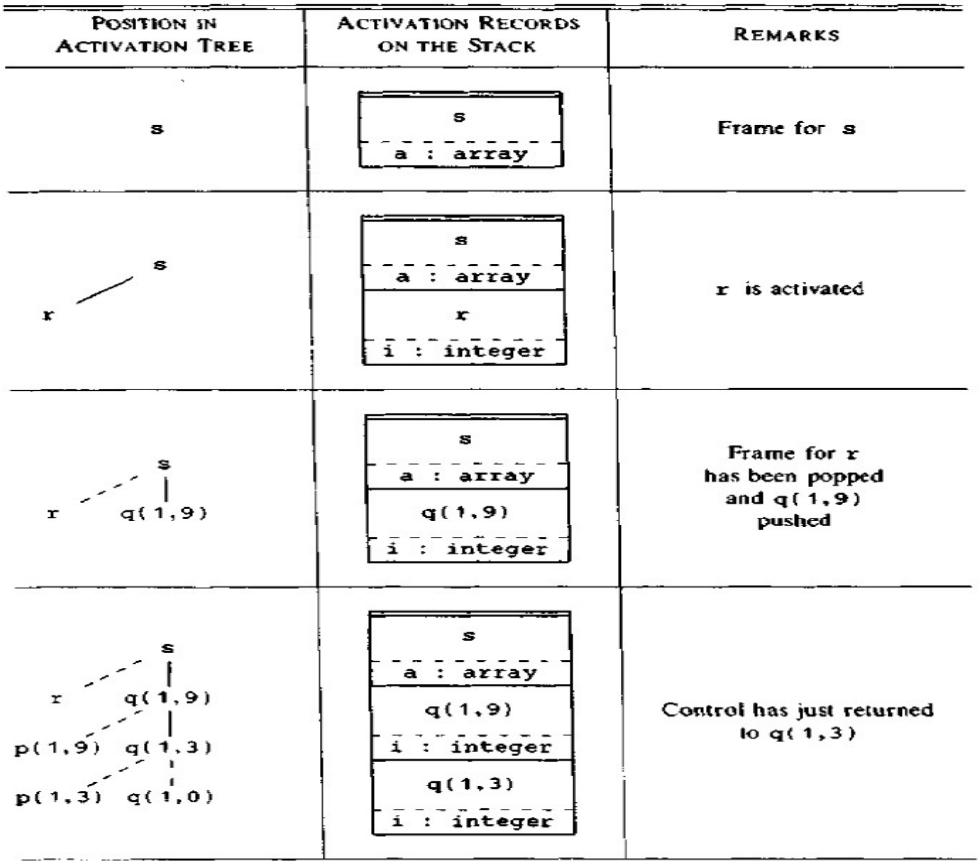


Fig. 7.13. Downward-growing stack allocation of activation records.

Heap Storage Allocation

- Heap allocation is the most flexible allocation scheme.
- In heap Allocation, memory can be allocated and deallocated in any order .
- So overtime the heap will consists of alternate areas that are free and in use
- Activation records for live activations need not be adjacent in heap.
- For efficient management of heap
  - For each size of interest, keep a linked list of free blocks of that size.
  - Fill the request for size s with a block of size s' where s' is the smallest size greater than or equal to s.
  - For large blocks of storage use heap manager.

9.

a. Discuss various data structures used to implement a symbol table. (10M)

Possible data structures for symbol table are arrays, linked lists, trees and hash tables. Scope access in Symbol tables should ensure the access to the correct record of an identifier that belongs to the current scope/context while compiling at a particular point of the program.

Unsorted Array:

The symbol table is organised as an array. As and when the names are encountered, the attributes are stored in the symbol table. The pointer Available indicates the next available location where the new entry can be stored in the array. Whenever a name is referenced, the table is searched from the end of the array to the beginning of the array. The scope information helps in validating the access. When the parse point is at scope level 1, the data in scopes 0 and 1 will be available. The most recent scope is always at the end of the table.

Example (Unsorted Array)

```
1.  PROGRAM Main
2.     GLOBAL   a, b
3.     PROCEDURE  P (PARAMETER  x)
4.        {Begin P}
5.            LOCAL  a
6.            ...a...
7.            ...b...
8.            ...x...
9.        {End P}
10.    {Begin Main}
11.       ...
12.       Call  P(a)
13.       ...
14.    {End Main}
```

Available →

Name	Type	Scope
Main	Program	0
a	Variable	0
b	Variable	0
P	Procedure	0
x	Parameter	1
a	Variable	1

Sorted Array:

The symbol table is organised as a sorted array. As and when the names are encountered, the attributes are stored in the symbol table. This entails in finding the correct location where the new record is to be inserted and shifting the records to accommodate the new record. The pointer Available indicates the last location. Whenever a name is referenced, the table is searched using binary search. No scope support is available directly. Could improve scope support by , using a sorted table of linked lists, or using a series of tables for each scope

Example (Sorted Array)

```
1.  PROGRAM Main
2.     GLOBAL   a, b
3.     PROCEDURE  P (PARAMETER  x)
4.        {Begin P}
5.            LOCAL  a
6.            ...a...
7.            ...b...
8.            ...x...
9.        {End P}
10.    {Begin Main}
11.       ...
12.       Call  P(a)
13.       ...
14.    {End Main}
```

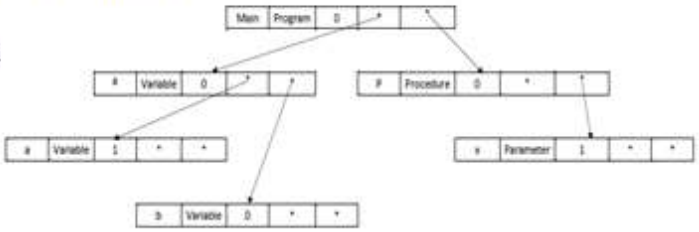
Name	Type	Scope
a	Variable	0
a	Variable	1
b	Variable	0
Main	Program	0
P	Procedure	0
x	Parameter	1

Binary search trees:

The symbol table is organised as a Binary search tree. As and when the names are encountered, the attributes are stored tree. This entails in finding the correct location where the new record is to be inserted and inserting the record as a leaf node. Whenever a name is referenced, the tree is searched using binary search tree find operation. No scope support is available directly. Could improve scope support by using a sorted table of linked lists, or using a series of tables for each scope. Find : O(log n), insert: O(log n), scope deletion: Implementation dependent.

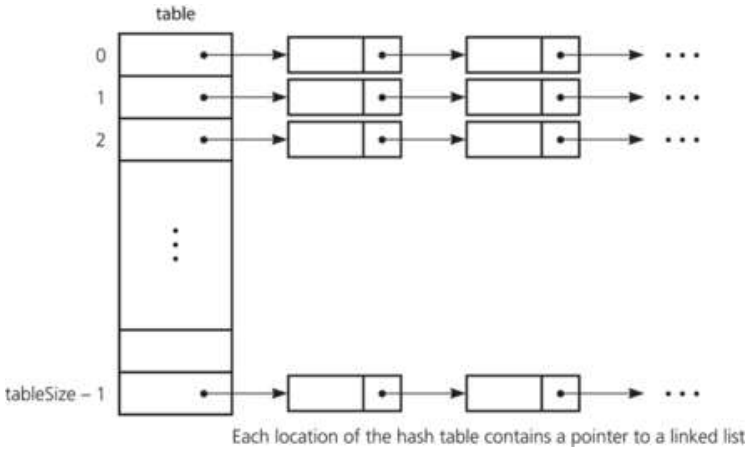
Example (Binary Tree)

```
1.  PROGRAM Main
2.     GLOBAL   a, b
3.     PROCEDURE  P (PARAMETER  x)
4.        {Begin P}
5.           LOCAL  a
6.           ...a...
7.           ...b...
8.           ...x...
9.        {End P}
10.    {Begin Main}
11.       ...
12.       Call P(a)
13.       ...
14.    {End Main}
```



Hash tables:

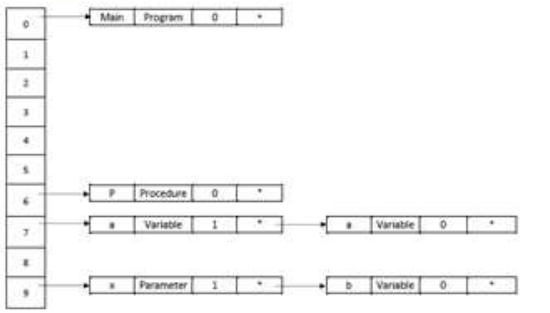
The symbol table is organised as a Hash Table which is an array of pointers to linear linked lists of symbol table records. All symbols that have the same name hash to the same list. As and when the names are encountered, the attributes are stored in the Hash table using the hash function. The new name is inserted at the begin of the linked list. This helps in quick find operation. Whenever a name is referenced, the table is searched using hashing and selecting the correct list. Good scope support is available as symbols in the most recently entered scope are at the front of the list. Scope deletion is slow because all the lists must be checked for records in the deleted scope. Find : O(1), insert: O(1), scope deletion: Implementation dependent.



Each location of the hash table contains a pointer to a linked list

Example (Hash Table)

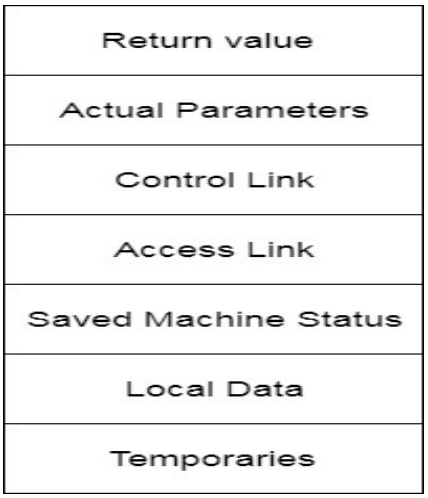
```
1.  PROGRAM Main
2.     GLOBAL   a, b
3.      PROCEDURE  P (PARAMETER  x)
4.         {Begin P}
5.              LOCAL  a
6.              ...a...
7.              ...b...
8.              ...x...
9.         {End P}
10.     {Begin Main}
11.        ...
12.        Call  P(a)
13.        ...
14.     {End Main}
```



b. Describe the general structure of an activation record. Explain the purpose of each item in the activation record.
(4M)

- Definition: information needed by a single execution of a procedure is managed using contiguous block of storage called an activation record or frame.
- It consists of collection fields shown in the following figure.
- When the procedure is called(activation begins) the activation record of a procedure is pushed on to the stack, and pop the activation record off the stack when it returns to the caller function(activation ends)

The diagram below shows the contents of activation records:



Return Value: It is used by calling procedure to return a value to calling procedure.

Actual Parameter: It is used by calling procedures to supply parameters to the called procedures.

Control Link: It points to activation record of the caller.

Access Link: It is used to refer to non-local data held in other activation records.

Saved Machine Status: It holds the information about status of machine before the procedure is called.

Local Data: It holds the data that is local to the execution of the procedure.

Temporaries: It stores the value that arises in the evaluation of an expression.