76_ Adnan Shaikh

Date
Page 1

Subject: SPCC Sem: VI

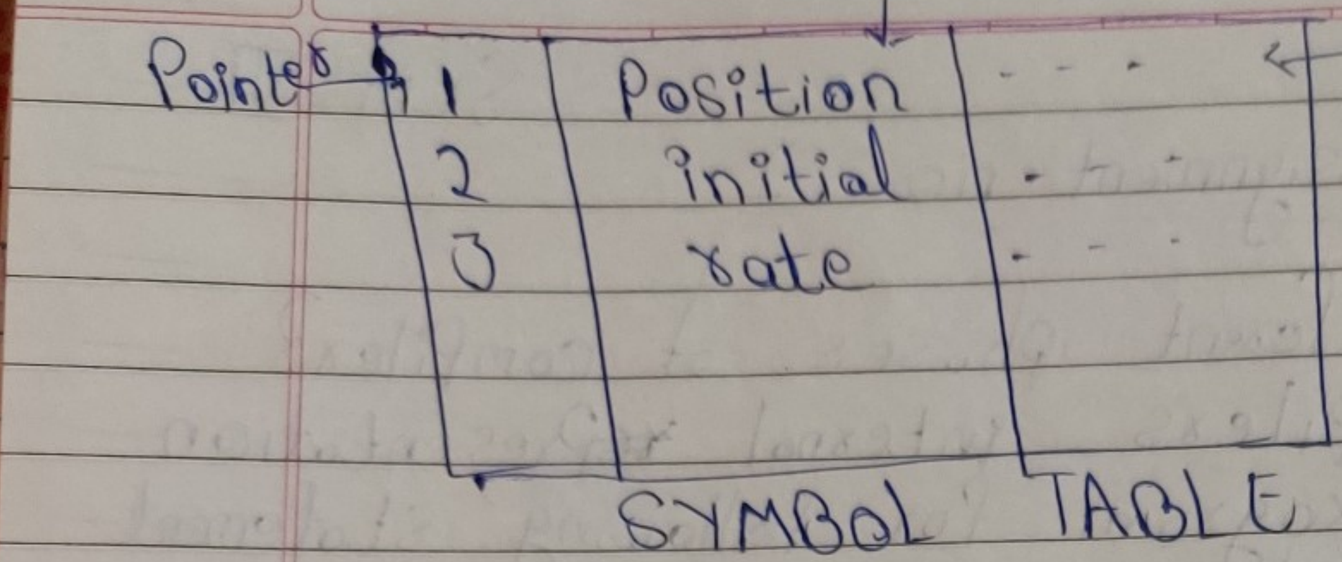
Assignment no. 2

1. What are different phases of compiler? Illustrate compilers internal representation of source program for following statement after each phase position = initial + rate * 60.

① Lexical Analysis: (a) The first phase of a compiler is called lexical analysis or scanning.
(b) The lexical analyzer reads the stream of characters making up source program and groups the character into meaningful sequences called lexemes.
(c) For each lexeme, the lexical analyzer produces as output a token of the form
<token-name, attribute-value>
that it passes on to the subsequent phase, Syntax analysis. In the token,
(d) In the token, the first component token-name is an abstract symbol that is used during Syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.
(e) Information from the symbol-table entry is needed for semantic analysis and code generation.

Lexical analysis of position = initial + rate * 60 is given by:

① - <id, 1> <=> <id, 2> <+> <id, 3> <*> <60>
For some tokens attributes values are implicit.



② Syntax Analysis: (a) The second phase of compiler is syntax analysis or parsing.
(b) The parser uses the first components of the tokens produced by the lexical analyzer to create a tree like intermediate representation that depicts the grammatical structure of the token stream.
(c) The intermediate node represents operation & the children represent the arguments of the operation.
(d) A syntax tree for the token stream (1) is:

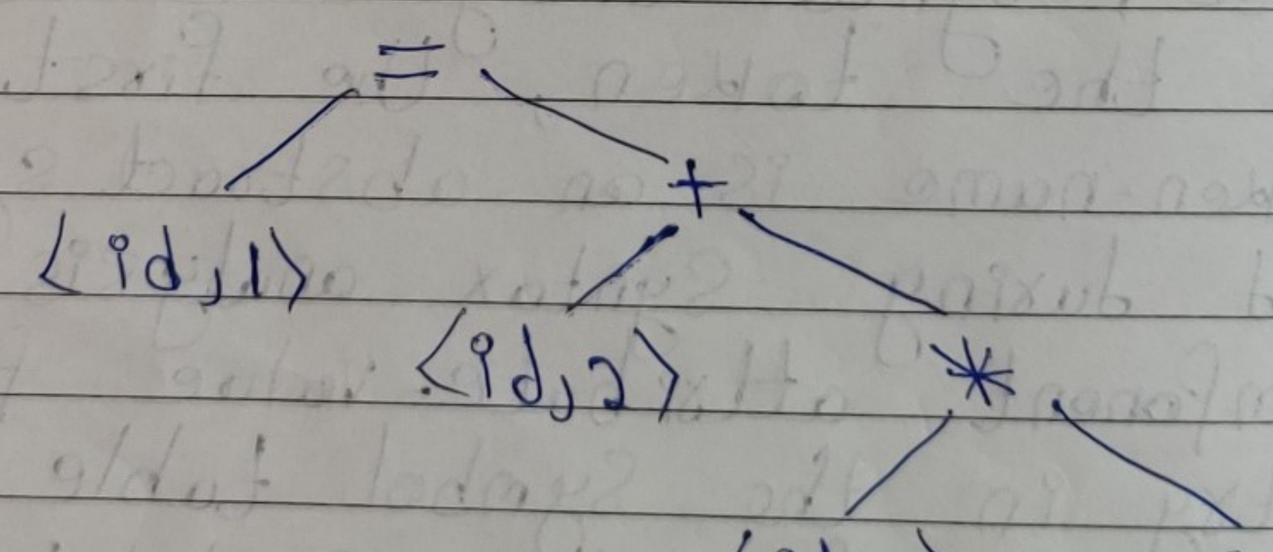


Fig: 1
SYNTAX TREE

(3) Semantic Analysis: (a) The Semantic analyzer uses the syntax tree and the information in the Symbol table to check the source program for Semantic consistency with the language definition
(b) It also gathers & check type information and saves it in either Syntax tree or the Symbol table, for subsequent use during intermediate-code generation.
(c) The language specification may permit some type conversions called coercions.
(d) Semantic Analysis of fig 1.:

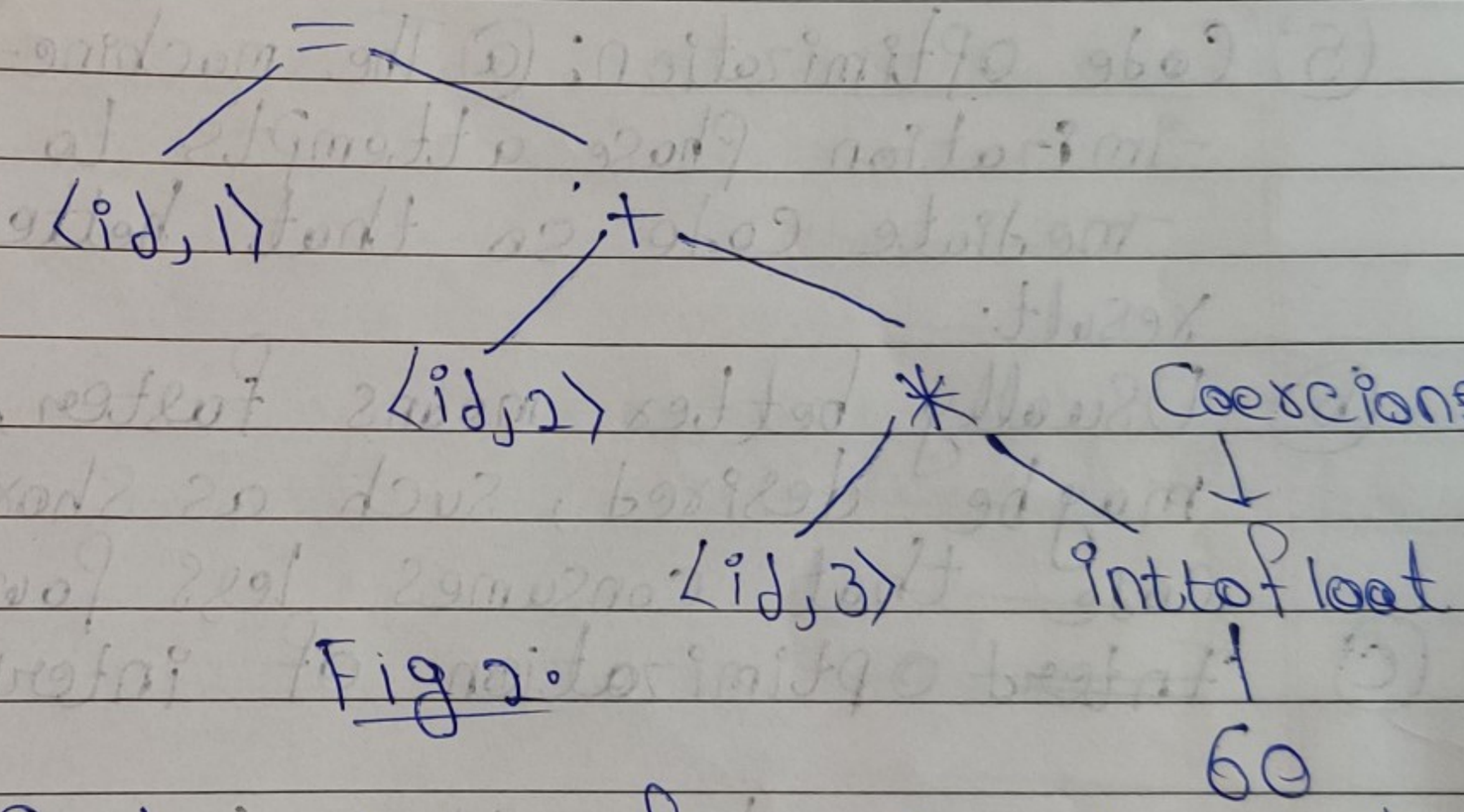


Fig 2.

Syntax tree after Semantic Analysis

(4) Intermediate Code generation: (a) Many compilers generate an explicit low level or machine-like Intermediate representation, which we can think of as a Program for an abstract machine.
(b) This intermediate representation should have two important properties: It should be easy to Produce and it should be easy to translate into the target machine.

(c) We consider a intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

(d) Intermediate code of sytax tree shown in fig. 2:

$$t_1 = \text{inttofloat}(60)$$
$$t_2 = id3 * t_1 \quad \text{--- (2)}$$
$$t_3 = id2 + t_2$$
$$id1 = t_3$$

(5) Code optimization: (a) The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.

(b) Usually better means faster, but other objective may be desired, such as shorter code, or target code that consumes less power.

(c) ~~Intert~~ optimization of intermediate code in (1):

$$t_1 = id3 * 60.0 \quad \text{--- (3)}$$
$$id_1 = id_2 + t_1$$

(6) Code Generation: (a) The code generator takes as input an intermediate representation of the source program and maps ~~it~~ it into the target language.

(b) If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.

(c) Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.

(c) Intermediate code
(d) Machine code for intermediate code in (3)

```
LDF   R2, id3        (Immediate constant)
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

2.) Eliminate left recursion in the following grammar (Remove Direct and Indirect recursion)

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \epsilon$

Ans) Left recursion in A : $A \rightarrow Ac$ & $A \xrightarrow{*} Aa$

$\therefore S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$

$\therefore A \rightarrow Ad_1 \mid Ad_2 \mid \cdots \mid B_1 \mid B_2 \mid \cdots$
is equivalent to
$A \rightarrow B_1A' \mid B_2A' \mid \cdots$
$A' \rightarrow d_1A' \mid d_2A' \mid d_3A^3 \mid \cdots \mid \epsilon$

In our case, $d_1 = c$, $d_2 = ad$
$B_1 = bd$, $B_2 = \epsilon$

$\therefore A \rightarrow Ad_1 \mid Ad_2 \mid B_1 \mid B_2$
Can be rewritten as:

Date
Page 6

$A \rightarrow B_1 A' \mid B_2 A'$
$A A' \rightarrow \alpha_1 A' \mid \alpha_2 A'$

Re substituting α_i & B_i
$S \rightarrow Aa \mid b$
$\therefore A \rightarrow bdA' \mid A'$
$A' \rightarrow cA' \mid adA' \mid \epsilon$

3. Consider the following grammar
$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$ to construct the Predictive Parser table.

Ans) ① Eliminating Left Recursion

$E \rightarrow E+T \mid T$
$\therefore E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$

$T \rightarrow T*F \mid F$
$\therefore T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$

Non-recursive Left recursive Grammar:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id.$

(2) First and Follow:

First(E) = { (, id }

First(E') = { +, ε }

First(T) = { (, id }

First(T') = { *, ε }

First(F) = { (, id }

~~Follow(E) = { (, id~~

Follow(E) = {), $ }

Follow(E') = {), $ }

Follow(T) = { +,), $ }

Follow(T') = { +,), $ }

Follow(F) = { *, +,), $ }

Predictive Parser table:

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

$E \rightarrow TE' \rightarrow First(TE') = First(T) = \{c, id\}$

$E' \rightarrow +TE' \rightarrow First(+TE') = \{+\}$

$E' \rightarrow \epsilon \rightarrow First(\epsilon) = \{\} \Rightarrow Follow(E')$
$Follow(E') = \{), \$\}$

$T' \rightarrow FT' \rightarrow First(F) = \{c, id\}$

$T' \rightarrow *FT' \rightarrow First(*) = \{*\}$

$T' \rightarrow \epsilon \rightarrow First(\epsilon) = \{\} \Rightarrow Follow(T')$

& $Follow(T') = \{+,), \$\}$

$F \rightarrow (E) \rightarrow First(c) = \{c\}$
$F \rightarrow id \rightarrow First(id) = \{id\}$

Date
Page 9

4. Differentiate Top down and Bottom-up parsing techniques. Explain Shift reduce Parser with example.

Ans)

Top-down	Bottom-up
a) Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder.	A Bottom-up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root.
2) left most Derivation is used	Right most derivation is used.
3) It attempts to find the left most derivat-ions for an input string.	It attempts to reduce the input string to Start symbol of a grammar.
4) It's main decision is to select what production rule to use in order to construct the string.	Its main decision is to select when to use a production rule to reduce the string to get the Starting Symbol.

Shift reduce parser:

Shift-reduce parsing is a form of bottom parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.

We use $ to mark the bottom of the stack and also the right end of the input. Conventionally, In 'bottom-up parsing, we show the top of the stack on the right. Initially, the stack is empty, and the string w is on the input, as follows:

STACK	INPUT
$	w$

During a left-to-right scan of the input string the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of the appropriate production.
The parser repeats this cycle until it has detected an error or until the stack contains the start symbol & the input is empty:

STACK	Input
$S	$

E.g. Consider, Grammar:

$E \rightarrow E+T \mid T$, $T \rightarrow T*F \mid F$, $F \rightarrow (E) \mid id$

and input: a*b id

Initial Configuration

STACK	Input
$	a*b$

Scan the postfix string from the left,

76_Afnan Shaikh

Date
Page 11

STACK	Input	Action
$	a * b $	Shift
$ a	* b $	Reduce
$ F	* b $	Reduce
$ T	* b $	Shift
$ T *	b $	Shift
$ T * b	$	Reduce
$ T * F	$	Reduce
$ E	$	Halt, Successful Parsing completion

Q6) Construct SLR Parsing table for following grammar.

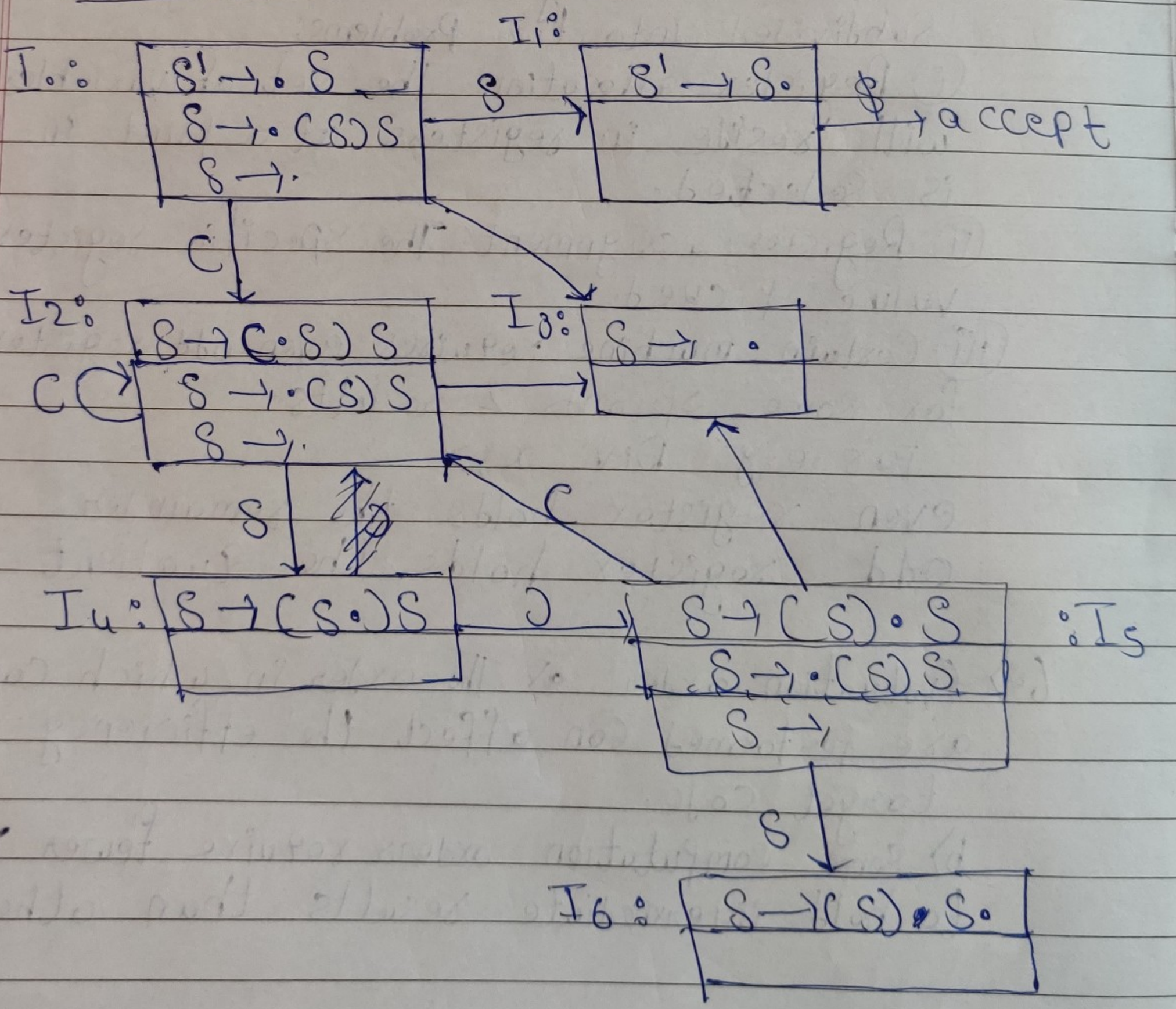
$S \rightarrow (S)S$
$S \rightarrow \varepsilon$

Soln: Augmented:

$S' \rightarrow S$ (1)
$S \rightarrow (S)S$ (2)
$S \rightarrow \varepsilon$ (3)

Follow(S') = {$} ; Follow(S) = {$,)}

Items Sets:



State	Actions				GoTo
	(	)		$	S
0	S2		S3		1
1				acc	
2	S2		S3		4
3		r3		r3	
4		S5			
5	S2	~~S2~~	S3		6
6		r2		r2	

Q5) For the given grammar below construct operator precedence relations matrix, assuming *, + are binary operators and id as terminal symbol & E as non-terminal Symbol.

E → E+E ; E → E*E ; E → id.

Soln: ① Construct operator precedence relation matrix

$\Sigma = \{ id, +, *, \$ \}$

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	

Apply operator precedence parsing algorithm to obtain skeletal syntax tree for the statement: id + id * id

Input String $ id + id * id $

$\Rightarrow$ $ $\lessdot$ id $\gtrdot$ + $\lessdot$ id $\gtrdot$ * $\lessdot$ id $\gtrdot$ $

$\Rightarrow$ $ E + E * E $

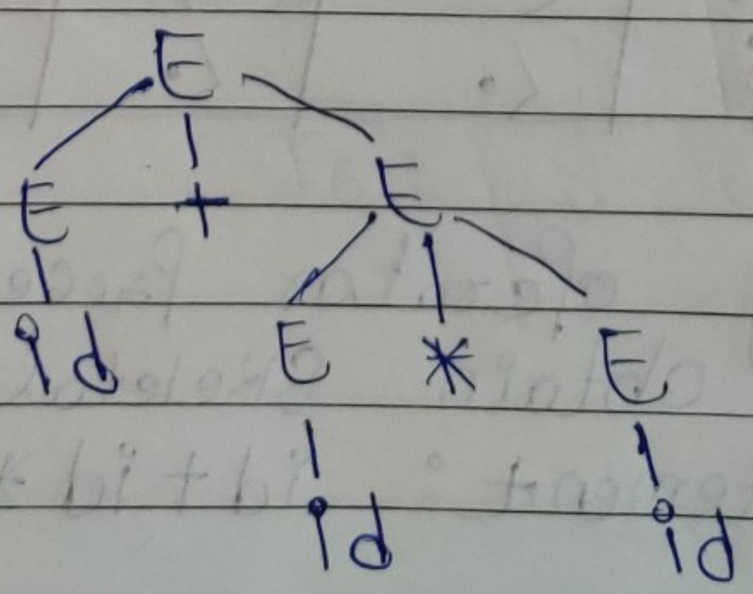
$\Rightarrow$ $ + * $ $\Rightarrow$ $ $\lessdot$ + $\lessdot$ * $\gtrdot$ $

$\Rightarrow$ ~~$ +~~ $ $\lessdot$ + $ $\Rightarrow$ $ $\lessdot$ + $\gtrdot$ $ $\Rightarrow$ $ $ $\Rightarrow$ Accepted

Using Stack: {a $\lessdot$ b, ~~a~~ a $\doteq$ b} $\Rightarrow$ Shift; a $\gtrdot$ b reduce

Stack	Input String	a op b	Action
$	id+id*id$	$ $\lessdot$ id	Shift
$ id	+id*id$	id $\gtrdot$ +	reduce
$ E	+id*id$	$ $\lessdot$ +	Shift
$ E+	id*id$	+ $\lessdot$ id	Shift
$ E+id	*id$	id $\gtrdot$ *	Reduce
$ E+E	*id$	+ $\lessdot$ *	Shift
$ E+E*	id$	* $\lessdot$ id	Shift
$ E+E*id	$	id $\gtrdot$ $	Reduce
$ E+E*E	$	* $\gtrdot$ $	Reduce
$ E+E	$	+ $\gtrdot$ $	Reduce
$ E	$	$ $\doteq$ $	Accept

Tree:



Q.6) Construct

Q.7) Write a Short note on Syntax Directed definitions.

Ans) (1) A Syntax-directed definition (SDD) is a Context-free grammar together with attributes and rules.

(2) Attributes are associated with grammar Symbols & rules are associated with the productions. If X is a Symbol and a is one of its attributes, then we write X.a to denote the value of 'a' at a particular parse-tree node labeled X.

(3) Attributes may be of any Kind: numbers, types, table, references, or Strings, for instance.

(1) There are two Kinds of ~~non-term~~ attributes for nonterminals:

(a) A Synthesized attribute for a nonterminal A at a parse-tree Node N is defined by Semantic rule associated with the production at N.

(6) An inherited attribute for a non-terminal B at a parse-tree Node N is defined by a semantic rule associated with the production at the parent of N.

(5) An SDD that involves only synthesized attributes is called S-attributed

(6) In an S-attributed SDD, value of head is computed with the production body. i.e parent value is computed using children value.

(7) The "L-attributed translations" (Left-to-Right) which encompass virtually all translations that can be performed during parsing is mostly applied

(8) e.g of infix-to-postfix translator:

Production	Semantic Rule
$E \rightarrow E_1 + T$	E.code = E_1.code ‖ T.code ‖ '+'

code is an attribute & ‖ used for concatenation. E & E_1 are same non-terminal but subscript is used for distinguish between instances.

(9) Rules can be written as follow:

$E \rightarrow E_1 + T$ { print '+' }

The code fragment appears between '{' '}' which is called as Semantic action

(10) The position of action in a production body determines the order in which the action is executed.

Date
Page 13

8.) Explain various form of intermediate code used by compiler

Ans) The various intermediate code forms are
a) Polish notation
b) Syntax tree
c) Three address codes: Triples and Quadruples
d) ~~Quadruples~~

a) Polish notation: The postfix notation for an Expression E can be defined inductively as follows:
1. If E is a variable or constant, then the postfix notation for E is E itself.
2. If E is an expression of the form E_1 op E_2, where op is any binary operator, then the postfix notation for E is $E_1' E_2'$ op, where E_1' and E_2' are the postfix notation for E_1 and E_2.
3. If E is a parenthesized expression of the form (E_1) then the postfix notation for E is the same as the postfix notation for E_1.

e.g.: $(9-5)+2$ $\xrightarrow{\text{Postfix}}$ $E=(E_1)+E_2$

$(E_1)=E_1=9-5=95-$
$E_2=2$
$\therefore E=(E_1)+E_2=(E_1)E_2+=95-2+$

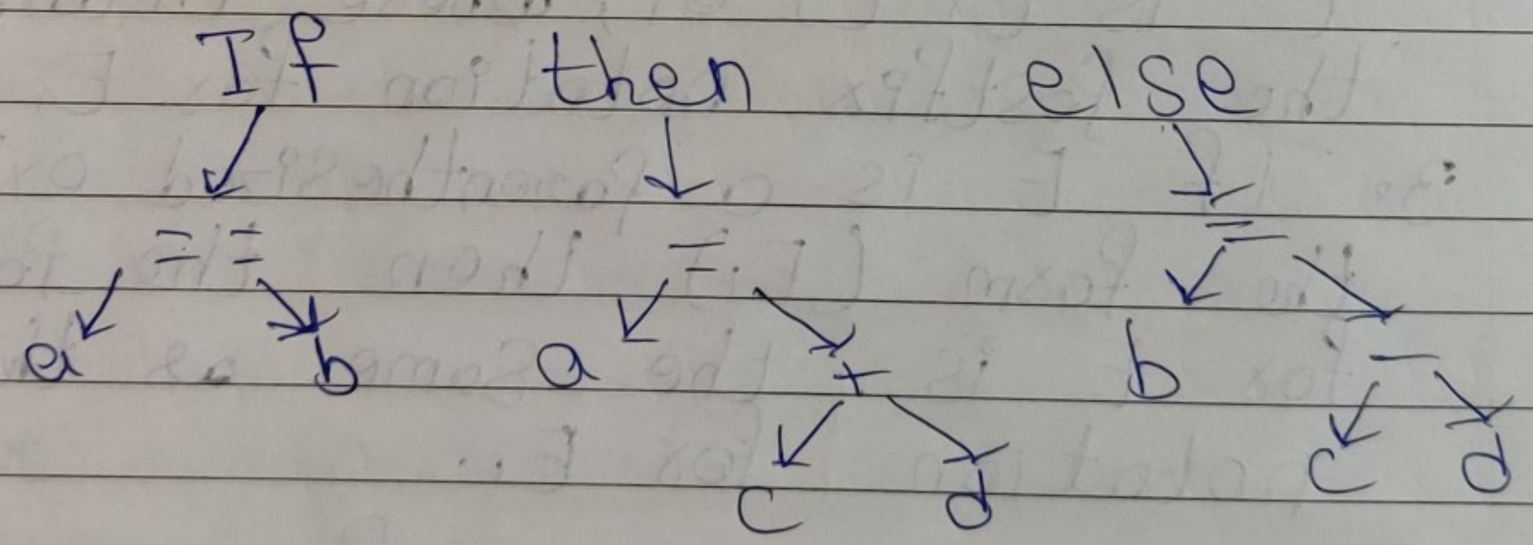
To evaluate postfix expression we repeatedly scan the postfix string from the left,

until we find an operator. Then, look to its left we find operands. Then, look to the left for the proper number of operands, & replace them by the result. Then repeat the process, continuing to the right & searching for another operator.

e.g: 95-2+ => 9-5 => 42+ => 4+2 => 6

b) Syntax tree: The parse tree itself is a useful intermediate language representation for a source program, especially in optimizing compilers where the intermediate code needs to extensively restructure.
We use an abstract Syntax tree, a tree in which each leaf represents an operand & each interior node an operator.

e.g: If a==b then a=c+d else b=c-d



c) Three-Address Code: ① In three address code, there is atmost one operator on the right side of an instruction. e.g: a = b+c.
② No more than 2 variables appear on right side & 1 on left of assignment operator.
③ It eliminates the ambiguity by evaluating only one operation at a time.

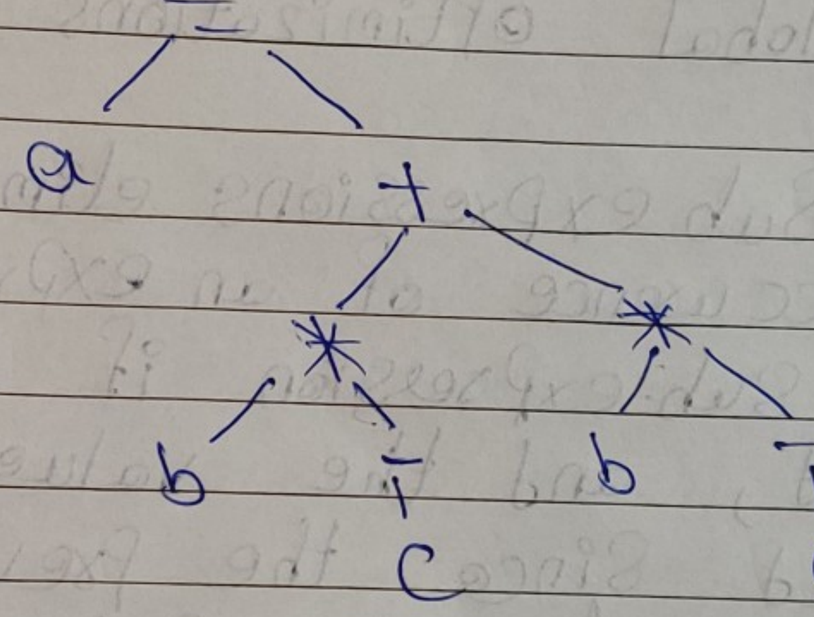
76_Adnan Shaikh

Date
Page 15

(i) Quadruples: A quadruple has four fields: op, arg1, arg2 & result
The evaluation is result of arg1 & arg2 of op is stored in result.

(ii) Triples: A triple has three fields: op, arg1 & arg2.
Triples store result of arg1 op arg2 in its position rather than explicit temporary name.

e.g:



=> a = b * -c + b * -c

Triple:

address	Instruction		op	arg1	arg2
35	(0)	0	minus	c	
36	(1)	1	*	b	(0)
37	(2)	2	minus	c	
38	(3)	3	*	b	(2)
39	(4)	4	+	(1)	(3)
40	(5)	5	=	a	(4)

Quadruple:

	op	arg1	arg2	result		op	arg1	arg2	result
0	minus	c		t1	4	+	t2	t4	t5
1	*	b	t1	t2	5	=	t5		a
2	min	c		t3					
3	4 *	b	t3	t4					

9.) Explain the principal sources of optimization

Ans) ① There are number of ways in which a compiler can improve a program without changing the function it computes.
Function preserving transformations example:
ⓐ Common Sub expressions elimination.
ⓑ Copy propagation;
ⓒ Dead-code elimination.
ⓓ Constant folding
The other transformations come up primarily when global optimizations are performed.

② Common Sub expressions elimination:
ⓐ An occurence of an expression E is called a common sub-expression if E was previously computed, and the values in E haven't changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.
ⓑ For example:

$t_1 := 4*i$
$t_2 := a[t_1]$
$t_3 := 4*i$ ← common
$t_4 := n$
$t_5 := b[t_3] + t_4$

optimized

$t_1 := 4*i$
$t_2 := a[t_1]$
$t_3 := n$
$t_4 := b[t_1] + t_3$

③ Copy propagation: ⓐ Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but it gives an opportunity to eliminate x.
for e.g: $x = pi;$ $A = x * r * r;$

optimized : A:= Pi*r*r
↑ x eliminated.

(4) Dead-code eliminations: (a) A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
(b) example:

```
int compute_dead(int c){
int a:= c * 2;
int b:= a*c; → dead code
return a;
}
```

in above code b is assigned but never used it can eliminated from there.

(5) Constant folding: (a) Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
(b) e.g: a:= 3.14157/2 optimized→ a:= 1.570

(6) Loop optimizations: (a) In loops, especially in the inner loops, programs tend to spend the bulk of their time. The running time of a program may be improved if the number of expressions in an inner loop is decreased, even if we increase the amount of code outside the loop.
(b) Three techniques are important for loop optimization.
(i) Code motion
(ii) Induction-variable elimination.
(iii) Reduction in strength.

(7) Code Motion: (a) An important modification that decreases the amount of code in a loop is code motion.
(b) This transformation takes an expression that yields the same result independent of the number of times a loop is executed (loop invariant) and places the expression in the outter loop if exist or "before the loop" if entry point exist.

e.g:

```
while(i<=limit-2){          optimized →   t=limit-2
#not Changing →  Stmt;                    while(i<=t){
#limit          }                           Stmt;
                                          }
```

limit-2 will remain same through out the loop instead of calculating it each time we move it out & calculated once.

(8) Induction variables: (a) Loops are usually processed inside out. ~~for example:~~ j = k;

(b) for e.g: j = k ; v = ~~so do~~ some value

```
while
do{
    j = j-1
    t1 = 4*j
    t2 = a[t1]
}while t2 > v
```

In above example values of j & t1 remain in lock-step; every time j = j-1 ⟹ t1 decreases by 4. i.e $t_1 = 4j_1 = 4(j-1) = 4j-4$
j_1 only implies previous value replaced with new one. Such identifiers are called induction.

(c) When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction variable elimination.

Foxe.g $j = k$; $t_1 = 4 * j$; $v =$ Some value

```
do {
    j = j - 1
    t1 = t1 - 4
    t2 = a[t1]
} while t2 > v
```

Here, we get ride of multiplication by substituting subtraction which is much cheaper & initialize t_1 at entry point of loop because it had no origin before assigning it value.

(9) Reduction in Strength (a) Reduction in Strength replaces expensive operations by equivalent cheaper ones on the target machine.

(b) for e.g: $x * x$ is cheaper to implement than x^2 which required a call to exponentiation routine.

$\therefore$ x^2 Optimized → $x * x$

4.) (a) Explain the various issues in the design of code generation.

Ans) (1) Input to code generator: (a) The input to the code generation consists of the intermediate representation of the source produced by front end, together with the information in the symbol table to determine run-time address of the data objects denoted by the names in the intermediate representation.

(b) Intermediate representation can be:
(i) Postfix Notation (ii) Three address code
(iii) Virtual Machine (Stack machine code)
(iv) Syntax tree.

(2) Target Program: (a) The output of the code generator is the target program. The output may be:
(i) Absolute machine language: It can be placed in a fixed memory location and can be executed immediately.
(ii) Relocatable machine language: It allows subprograms to be compiled separately.
(iii) Assembly language: Code generation is made easier.

(3) Memory management: (a) Names in the source program are mapped to addresses of data objects in run-time memory by front-end & code generator.
(b) It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
(c) Labels in three-address statements have to be converted to addresses for instructions.

Date
Page 21

4.) Instruction Selection: a) The instructions of target machine should be complete & uniform.
b) Instruction speeds & machine idioms are important factors when efficiency of target program is considered.
c) The quality of the generated code is determined by its speed and size.

5.) Register allocation: a) Instructions involving register operands are shorter & faster than those involving operands in memory. The use of registers ~~~~ is subdivided into two problems:
(i) Register allocation: The set of variables that will reside in registers at a point in the program is selected.
(ii) Register assignment: The specific register that a value picked.
(iii) Certain machine requires even-odd register pairs for some operands & results.
For e.g: Div x, y
even register holds the remainder.
odd register holds the quotient.

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