USN	1	P	I							

PES Institute of Technology, Bangalore

14CS351

Department of Computer Science / Information Science & Engineering

SEMESTER END EXAMINATION (SEE) B. E. VI SEMESTER

[Session: **May, 2017**]

14CS351 - COMPILER DESIGN

Time: 3 hrs. **Answer All Questions** Max Marks: **100**

1.	a)	Consider the following tokens and their associated regular expressions, given as a lex-like specification: %% (01|10) print ("course") 0(01) *1 print ("compiler") (1010*1|0101*0) print ("design") Give an input to this scanner such that the output string is **(compiler ^{11} design ^{2}) ^{4} course ^{3}** Where, A^i denotes A repeated i times. (And, of course, the parentheses are not part of the output.) You may use similar shorthand notation in your answer.	5
1.	b)	Recall from the lecture that, when using regular expressions to scan an input, we resolve conflicts in lexer by taking the largest possible match at any point. That is, if we have the following lex scanner specification: **%%** **do { return T_Do ; }** **[A-Za -z_][A-Za -z0 -9_]* { return T_Identifier ; }** and we see the input string "dot", we will match the second rule and emit T_Identifier for the whole string, not T_Do. However, it is possible to have a set of regular expressions for which we can tokenize a particular string, but for which taking the largest possible match will fail to break the input into tokens. **Give an example of a set of regular expressions and an input string such that:** a) The string can be broken into substrings, where each substring matches one of the regular expressions, b) and using our usual lexer algorithm, taking the largest match at every step, will fail to break the string in a way in which each piece matches one of the regular expressions. Explain how the string can be tokenized and why taking the largest match won't work in this case.	5
1.	c)	Explain the front end of a compiler using the following example: **if(x>10) x = x + 100/x;**	10
2.	a)	Consider the following simple context free grammars: G_1: $S \rightarrow Aa$, $A \rightarrow \varepsilon$, $A \rightarrow bAb$ G_2: $S \rightarrow Aa$, $A \rightarrow \varepsilon$, $A \rightarrow Abb$ Note that the grammars generate the same language: strings consisting of even numbers of b's (including 0 of them), followed by an a.	10 (3 + 7)

USN	1	P	I							

		I. Attempt to show a shift-reduce parse of the string bbbba for a parser for grammar G_1. Show the contents of the stack, the input, and the actions (i.e., shift, reduce, error, accept). You don't need to create a parse table; just use your knowledge of the grammar and how the parser works. Be sure to indicate any conflicts and explain why they are conflicts. **II.** Using the definition and properties of various parsers that you studied in Compiler design course, justify whether or not: 1. G1 is LR(1)? 2. G2 is LR(0)? 3. G1 is LL(1)? : 4. G2 is LL(1)? **[You get credit only for correct justification not for writing yes or No]**	
2.	b)	Consider the following grammar over the alphabet $\sum = \{ u, v, w, x, y, z\}$ S -> UVW U -> u | Wv | ϵ V -> w | xU | ϵ W -> y | z **I.** Give the first sets and follow sets of each non-terminal in the grammar. **II.** Is this grammar LL (1)? Justify.	10

3.	a)	Consider grammar and rules given below for array address translation and generating 3 address code for array references:
		• L.array.basename means name of the array
		• L.array.typeofelement means type of the element of the array
		• L.type.width means width of L.type
		Assume size of integer to be 4 bytes, and lower bound of the arrays to be 0
		Let A, B and C be 10 X 5, 5 X 7, and 10 X 7 arrays of integers respectively. Let i, j, and k be integers. **Construct an annotated parse tree for the expression**
		C[i][j] + A[i][k] * B[k][j]
		and show the 3-address code sequence generated for the expression.
		$E \rightarrow E_1 + E_2$ {E.addr = newtemp();
		gen(E.addr '=' E_1.addr '+' E_2.addr);}
		$E \rightarrow E_1 * E_2$ {E.addr = newtemp();
		gen(E.addr '=' E_1.addr '*' E_2.addr);}
		| id {E.addr = id.lexeme; }
		| L {E.addr = newtemp();
		gen(E.addr '=' L.array.basname '[' L.addr ']'); }
		L → id [E] {L.array = id.lexeme; L.type = L.array.typeofelement;
		L.addr=newtemp();
		gen(L.addr '=' E.addr '*' L.type.width);}
		| L_1 [E] {L.array = L_1.array; L.type = L_1.type.typeofelement;
		t = newtemp(); L.addr = newtemp();
		gen(t '=' E.addr '*' L.type.width);
		gen(L.addr '=' L_1.addr '+' t); }
			10

USN	1	P	I							

<table>
<tr><td>3.</td><td>b)</td><td>Provide an implementation of SDT scheme for simple type declaration (grammar is given below for reference) during LR parsing. For full credit, provide proper explanation and show parser stack when necessary. Consider for example the input string as int a, b

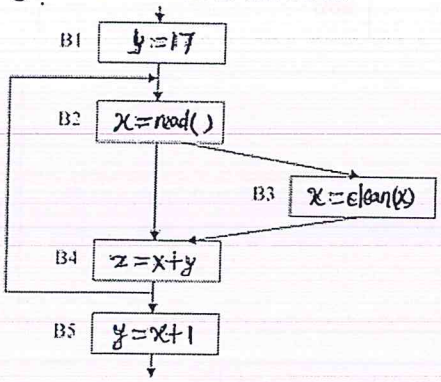
D -> TL
T -> int | float
L -> L, id | id

(Note: Provide the general structure of parser stack used during LR parsing.)</td><td>10</td></tr>
<tr><td>4.</td><td>a)</td><td>What are the different ways in which a procedure can access non-local data on a run time stack? Explain each technique using a simple example.</td><td>5</td></tr>
<tr><td>4.</td><td>b)</td><td>Generate Target Code for the following procedure call assuming static allocation.
<table><tr><td></td><td>Code is kept at address</td><td>Activation record is kept at address</td></tr><tr><td>main</td><td>100</td><td>600</td></tr><tr><td>p</td><td>400</td><td>800</td></tr></table><table><tr><td>//main()
n = 6
i = 0
L1 : if i >= 6 goto L2
i = i + 1
goto L1
L2 : call p
halt</td><td>//p()
i = 60
return</td></tr></table></td><td>5</td></tr>
<tr><td>4.</td><td>c)</td><td>Generate 3-address code for the following program and convert it into a CFG.
<pre>int prime(int n)
{
  int n, i, flag = 0;
  for(i=2; i<=n/2; ++i) {
    if(n%i==0){
      flag=1;
      break;
    }
  }
  if (flag==0)
      return 0;
  else
      return 1;
}</pre></td><td>10</td></tr>
<tr><td>5.</td><td>a)</td><td>What are the issues in the design of a code generator?</td><td>5</td></tr>
<tr><td>5.</td><td>b)</td><td>Optimize the code below by applying the following code transformations: constant propagation, constant folding, copy-propagation, dead-code elimination and strength reduction.</td><td>5</td></tr>
</table>

USN	1	P	I							

```
L0: t1 = t1 + 1
    t2 = 0
    t3 = t1 * 8
    t4 = t3 + t2
    t5 = t4 * 4
    t6 = *t5
    t7 = FP + t3
    *t7 = t2
    t8 = t1
    if (t8 > 0) goto L1
L1: goto L0
L2: t1 = 1
    t10 = 16
    t11 = t1 * 2
    goto L1
```

5. c) Perform Live variable analysis on the following CFG. Provide your answer in the table format as given below next to the CFG. — 5



	use	def	IN	OUT
B1				
B2				
B3				
B4				
B5				

Note : clean(x) is some function performed on the value of x.

read() reads the value of x from the user.

5. d) Construct DAG for the following Block and optimize. — 5

1. t1 : = 4 * I
2. t2 := A − 4
3. t3 := t2 [t1]
4. t4 := 4 * I
5. t5 : = B − 4
6. t6 : = t5 [t4]
7. t7 : = t3 * t6
8. t8 : =PROD + t7
9. PROD : = t8
10. t9 : = I + 1
11. I = t9
12. if I ≤ 20 goto (1).

Scheme & Solution

USN	1	P	I							

PES Institute of Technology, Bangalore 14CS351

Department of Computer Science / Information Science & Engineering

SEMESTER END EXAMINATION (SEE) B. E. VI SEMESTER

[Session: **May, 2017**]

14CS351 - COMPILER DESIGN

Time: 3 hrs. **Answer All Questions** Max Marks: **100**

1.	a)	Consider the following tokens and their associated regular expressions, given as a lex-like specification: %% (01|10) print ("course") 0(01) *1 print ("compiler") (1010*1|0101*0) print ("design") Give an input to this scanner such that the output string is **(compiler ^{11} design ^{2}) ^{4} course ^{3}** Where, A^i denotes A repeated i times. (And, of course, the parentheses are not part of the output.) You may use similar shorthand notation in your answer. **Solution:** **((0011) ^{11} 0100 ^{2}) ^{4} 011001**	5
1.	b)	Recall from the lecture that, when using regular expressions to scan an input, we resolve conflicts in lexer by taking the largest possible match at any point. That is, if we have the following lex scanner specification: **%%** **do { return T_Do ; }** **[A-Za -z_][A-Za -z0 -9_]* { return T_Identifier ; }** and we see the input string "dot", we will match the second rule and emit T_Identifier for the whole string, not T_Do. However, it is possible to have a set of regular expressions for which we can tokenize a particular string, but for which taking the largest possible match will fail to break the input into tokens. **Give an example of a set of regular expressions and an input string such that:** a) The string can be broken into substrings, where each substring matches one of the regular expressions, b) and using our usual lexer algorithm, taking the largest match at every step, will fail to break the string in a way in which each piece matches one of the regular expressions. Explain how the string can be tokenized and why taking the largest match won't work in this case. **Solution:** a) Consider the following scanner: %% a { return A; }	5

USN	1	P	I							

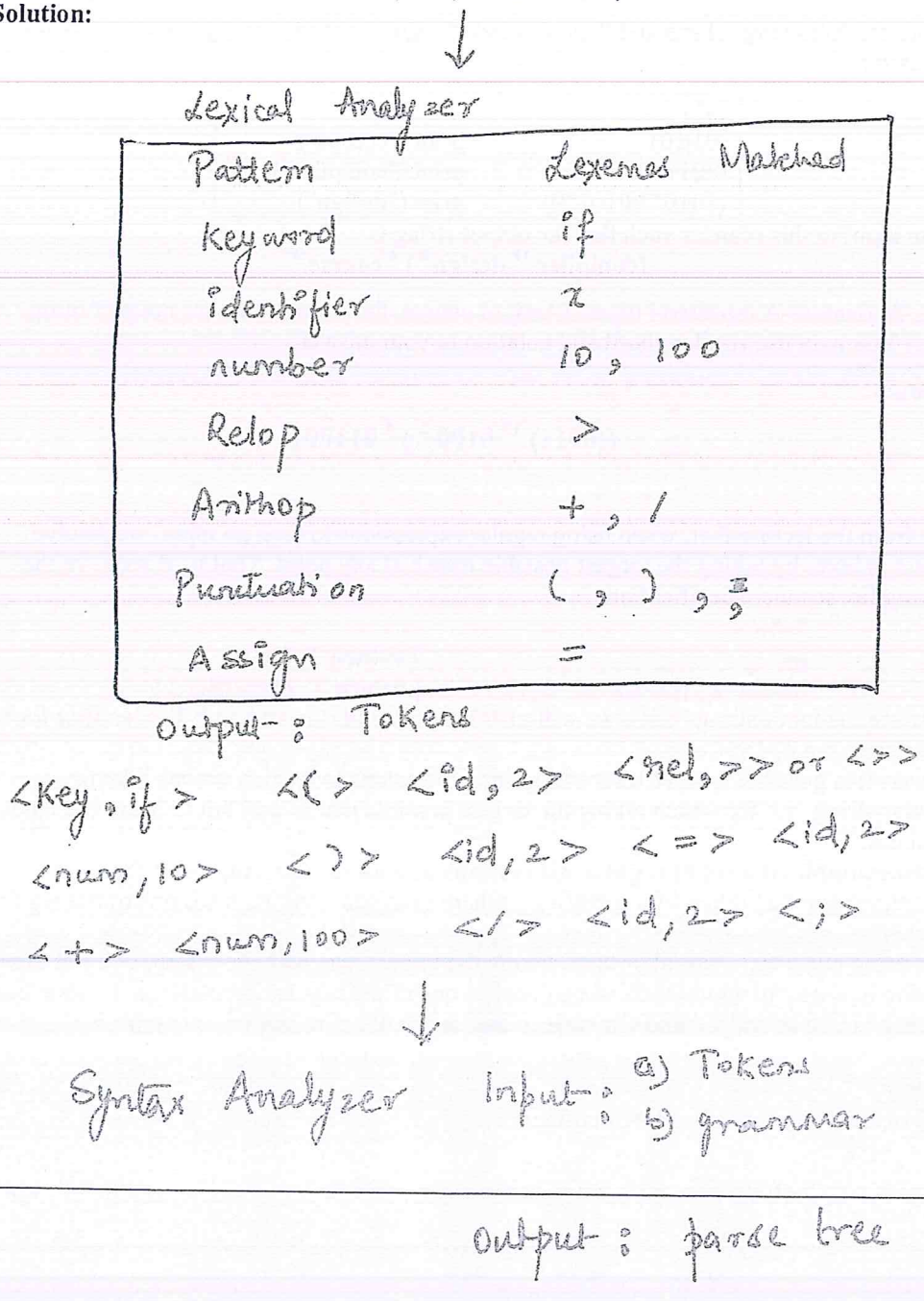
aba { return B; }
bab { return C; }

and the string "abab". This can be broken into `a', followed by `bab'.

b) However, the largest possible match strategy will first consume the beginning of the string as `aba', then stop when it finds that the remainder of the input is just `b', which can't be matched to any token.

1. c) Explain the front end of a compiler using the following example: **if(x>10) x = x + 100/x;** — 10

Solution:



USN	1	P	I							

Grammar : S → if (cond) S | id = E ;

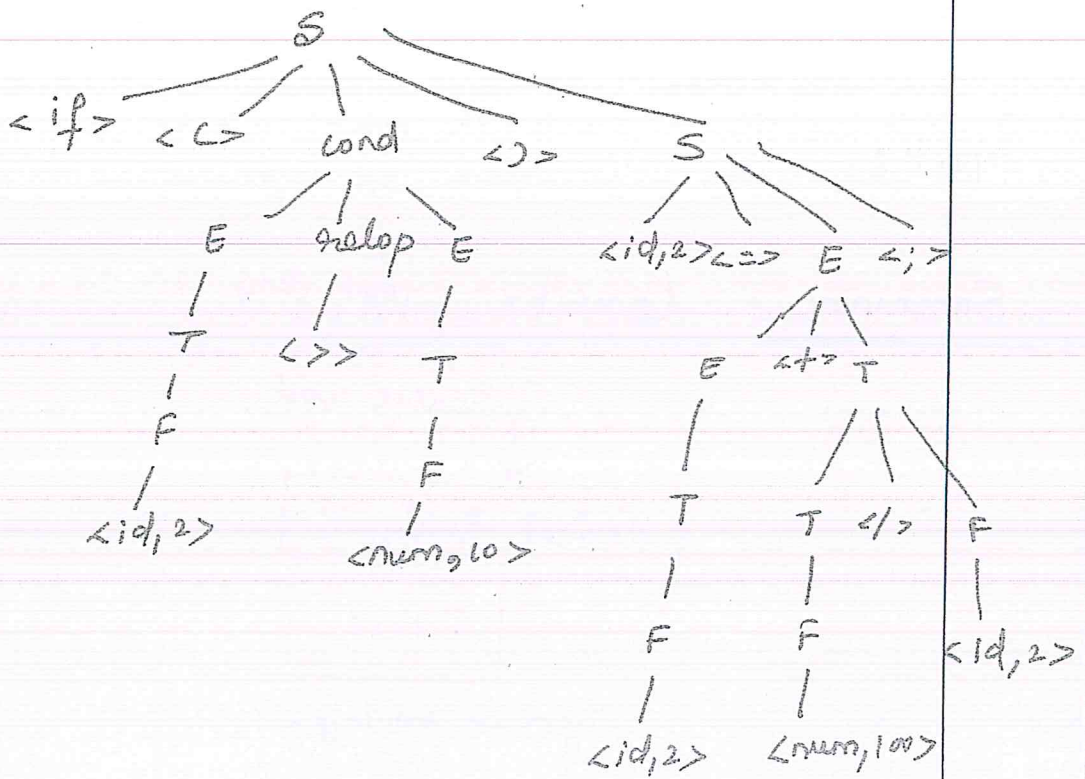
cond → E relop E

relop → < | <= | >= | == | !=

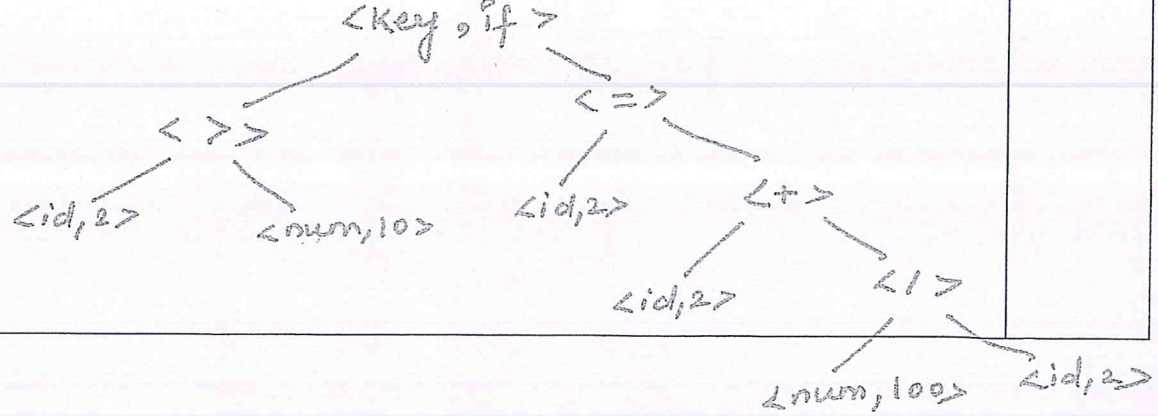
E → E+T | T

T → T*F | F | T/F

F → id | num



Semantic Analyze : outputs semantically checked Syntax tree



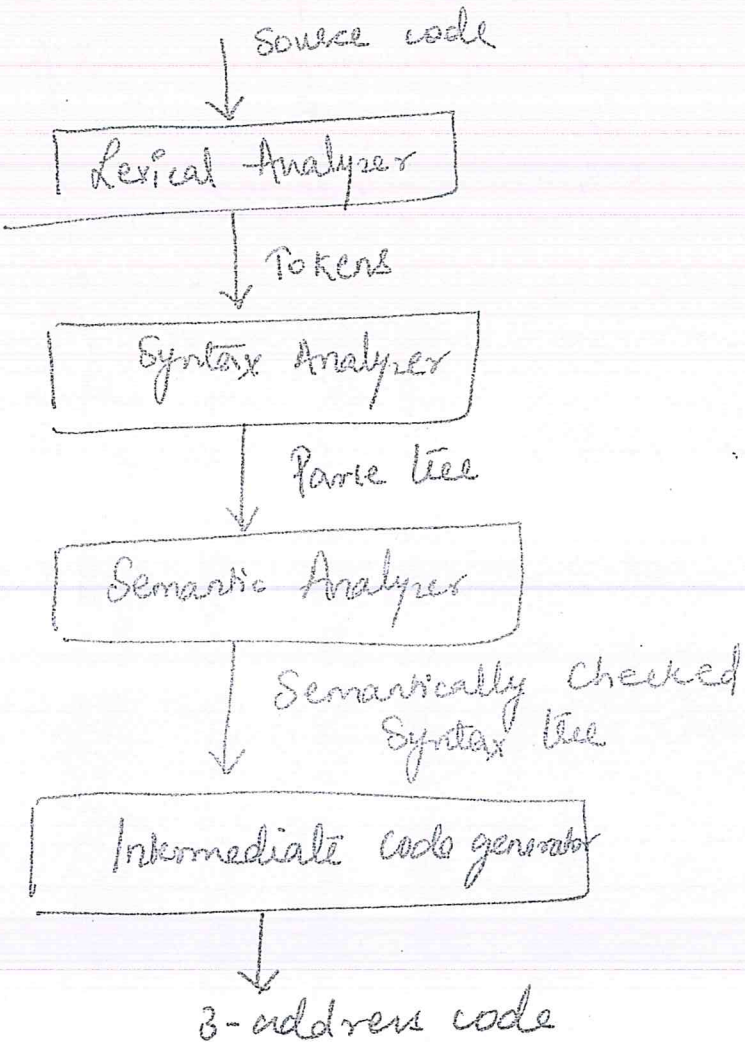
Intermediate Code generator

→ produces as output the 3-address code

```
iffalse x > 10 goto next
t1 = 100/x
t2 = x + t1
x = t2
```

next:

Summary : Compiler front-end



USN	1	P	I							

2.	a)	Consider the following simple context free grammars: G_1: $S \rightarrow Aa$ $A \rightarrow \varepsilon$ $A \rightarrow bAb$ G_2: $S \rightarrow Aa$ $A \rightarrow \varepsilon$ $A \rightarrow Abb$	10 (3 + 7)

Note that the grammars generate the same language: strings consisting of even numbers of b's (including 0 of them), followed by an a.

I. Attempt to show a shift-reduce parse of the string bbbba for a parser for grammar G_1. Show the contents of the stack, the input, and the actions (i.e., shift, reduce, error, accept). You don't need to create a parse table; just use your knowledge of the grammar and how the parser works. Be sure to indicate any conflicts and explain why they are conflicts.

II. Using the definition and properties of various parsers that you studied in Compiler design course, justify whether or not:

1. G1 is LR(1)?
2. G2 is LR(0)?
3. G1 is LL(1)? :
4. G2 is LL(1)?

[You get credit only for correct justification not for writing yes or No]

Solution:

I.

Stack (with top at right)	Input	Action
$	bbbba$	shift b
$b	bbba$	shift b
$bb	bba$	CONFLICT: reduce A → ε or shift b?

II.

1. G1 is LR(1)?

G1 is not LR(1) as in one of the closures:

A->b°Ab, a

A->e°, b

A->°bAb, b

we'll get a shift-reduce conflict.

2. G2 is LR(0)?

The grammar is in LR(0) as there are no conflicts.

USN	1	P	I							

3. G1 is LL(1)?

G1 is not LL(1) as follow(A) = {a, b}

Hence A → epsilon and A → bAb would create a conflict for A on b in the parsing table.

4. G2 is LL(1)?

The grammar is not LL(1) as follow(A) = {a, b}

Hence A → epsilon and A → Abb would create a conflict for A on b in the parsing table.

2. b) Consider the following grammar over the alphabet $\sum = \{ u, v, w, x, y, z\}$ — 10 (8 + 2)

```
S -> UVW
U -> u | Wv | ϵ
V -> w | xU | ϵ
W -> y | z
```

I. Give the first sets and follow sets of each non-terminal in the grammar.
II. Is this grammar LL (1)? Justify.

Solution:

I.

	First	Follow
S	{u, w, x, y, z}	{$}
U	{u, y, z, ϵ}	{w, x, y, z}
V	{w, x, ϵ}	{y, z}
W	{y, z}	{v, $}

II.

	u	v	w	x	y	z	$
S	UVW		UVW	UVW	UVW	UVW	
U	u		ϵ	ϵ	Wv/ϵ	Wv/ϵ	
V			w	xU	ϵ	ϵ	
W					y	z	

Since there are multiple entries for some table locations, so the grammar is not LL(1).

3. a) Consider grammar and rules given below for array address translation and generating 3 address code for array references: — 10

- L.array.basename means name of the array
- L.array.typeofelement means type of the element of the array
- L.type.width means width of L.type

USN	1	P	I							

Assume size of integer to be 4 bytes, and lower bound of the arrays to be 0
Let A, B and C be 10 X 5, 5 X 7, and 10 X 7 arrays of integers respectively. Let i, j, and k be integers. **Construct an annotated parse tree for the expression**

C[i][j] + A[i][k] * B[k][j]

and show the 3-address code sequence generated for the expression.

$E \rightarrow E_1 + E_2$ {E.addr = newtemp();
gen(E.addr '=' E_1.addr '+' E_2.addr);}

$E \rightarrow E_1$ * E_2 {E.addr = newtemp();
gen(E.addr '=' E_1.addr '*' E_2.addr);}

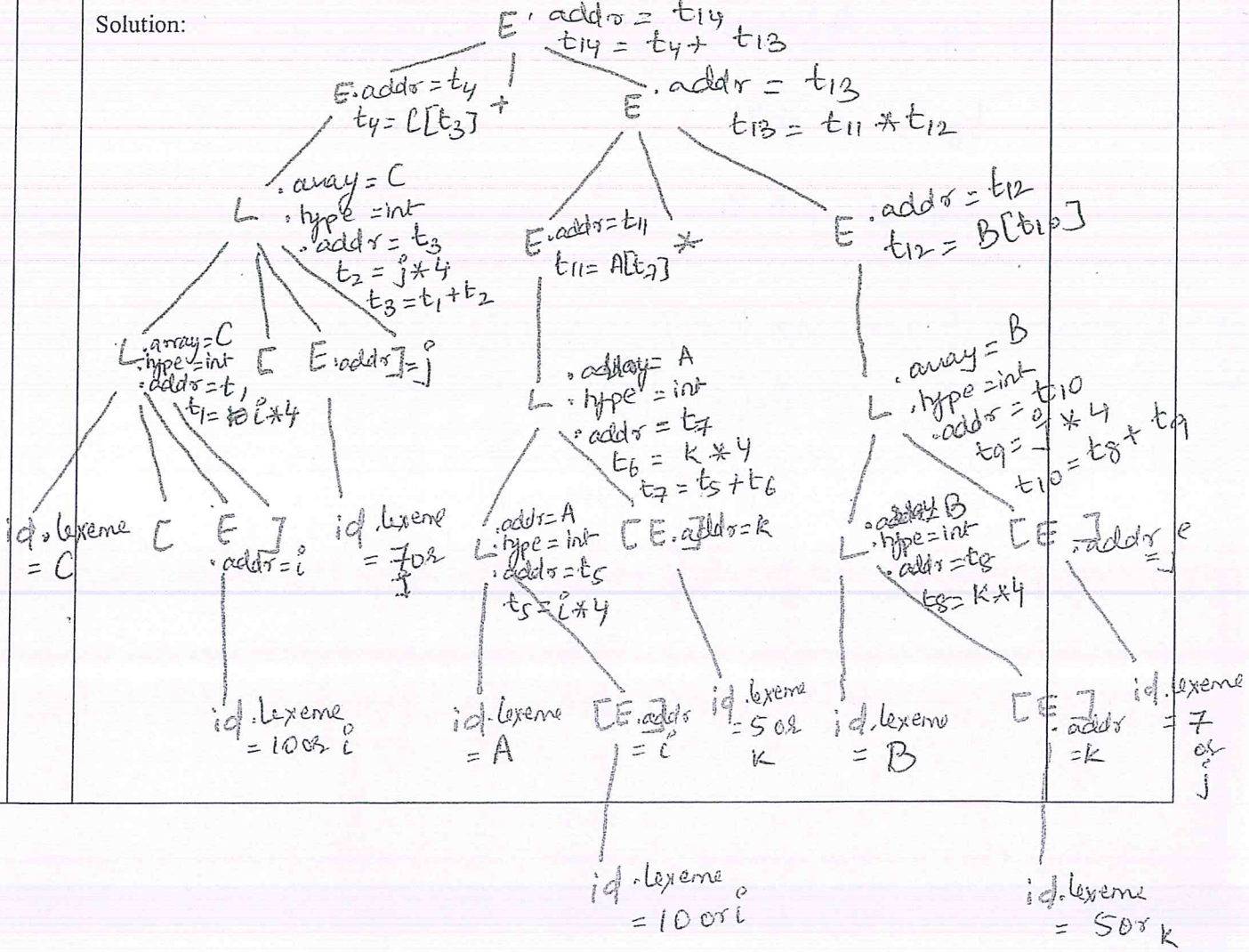
| id {E.addr = id.lexeme; }

| L {E.addr = newtemp();
gen(E.addr '=' L.array.basname '[' L.addr ']'); }

L → id [E] {L.array = id.lexeme; L.type = L.array.typeofelement;
L.addr=newtemp();
gen(L.addr '=' E.addr '*' L.type.width);}

| L_1 [E] {L.array = L_1.array; L.type = L_1.type.typeofelement;
t = newtemp(); L.addr = newtemp();
gen(t '=' E.addr '*' L.type.width);
gen(L.addr '=' L_1.addr '+' t); }

Solution:



Intermediate code generated is :

$t_1 = i * 4$

$t_2 = j * 4$

$t_3 = t_1 + t_2$

$t_4 = C[t_3]$

$t_5 = i * 4$

$t_6 = K * 4$

$t_7 = t_5 + t_6$

$t_{11} = A[t_7]$

$t_8 = K * 4$

$t_9 = j * 4$

$t_{10} = t_8 + t_9$

$t_{12} = B[t_{10}]$

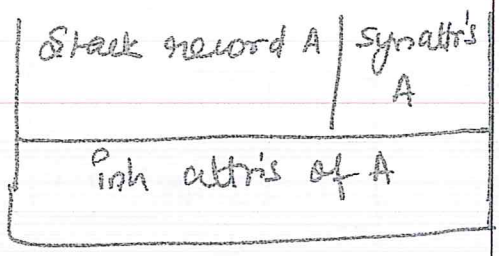
$t_{13} = t_{11} * t_{12}$

$t_{14} = t_4 + t_{13}$

USN	1	P	I							

3.	b)	Provide an implementation of SDT scheme for simple type declaration (grammar is given below for reference) during LR parsing. **For full credit, provide proper explanation and show parser stack when necessary.** Consider for example the input string as **int a, b** **[D -> TL** **T -> int|float** **L -> L, id | id]** (Note: Provide the general structure of parser stack used during LR parsing.) **Solution:**	10

Parser Stack Structure :-



SDT Scheme :

$D \rightarrow T \quad \{ L.inh = T.type \}\ L$

$T \rightarrow int \quad \{ T.type = int \}$

$T \rightarrow float \quad \{ T.type = float \}$

$L \rightarrow \{ L_1.inh = L.inh \}\ L_1 , id \quad \{ addType(id.entry, L.inh) \}$

$L \rightarrow id \quad \{ addType(id.entry, L.inh) \}$

for LR parsing we change SDT Scheme as follows :

$D \rightarrow T\ M\ L$

$M \rightarrow \epsilon \quad \{ L.inh = T.type \}$

$T \rightarrow int \quad \{ T.type = int \}$

$T \rightarrow float \quad \{ T.type = float \}$

$L \rightarrow N\ L_1 , id \quad \{ addType(id.entry, L.inh) \}$

$N \rightarrow \epsilon \quad \{ L_1.inh = L.inh \}$

$L \rightarrow id \quad \{ addType(id.entry, L.inh) \}$

USN	1	P	I							

Implementation

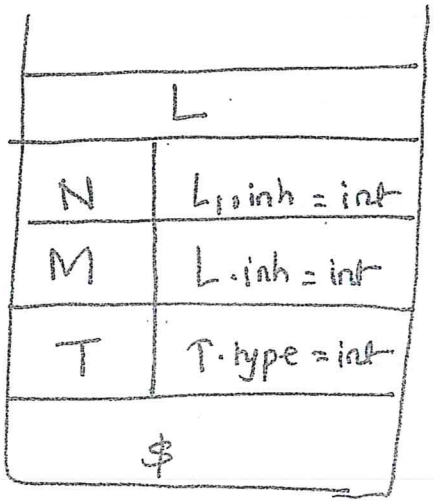
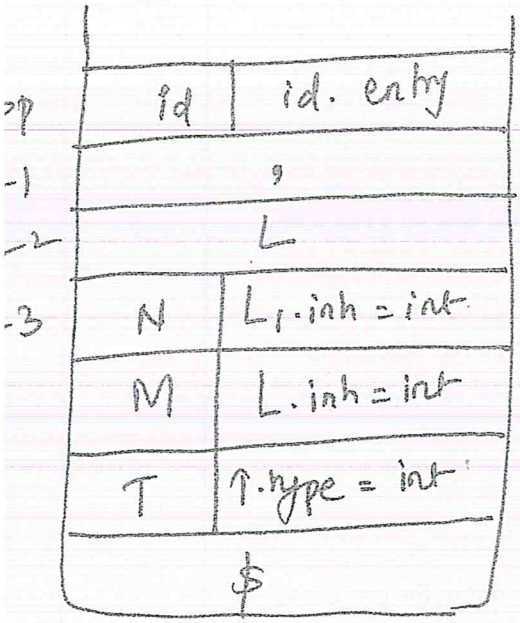
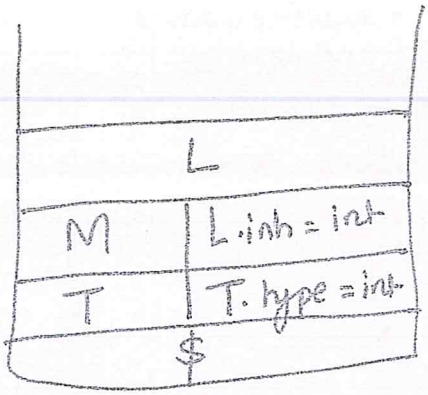
Stack	Input Buffer	Action
$	int id , id $	a) Shift int
$ int	id , id $	b) reduce using T → int
	id , id $	c) reduce using M → ∈ and perform the action
	id , id $	S[top-1].inh = S[top-1].type
	id , id $	d) reduce using N → ∈ and perform S[top].inh = S[top-1].inh
	id , id $	e) Shift id
	, id $	f) reduce using L → id and perform action addType (S[top].entry, S[top-1].inh)

T	T.type=int
$	

M	L.inh = int
T	T.type=int
$	

N	L_1.inh = int
M	L.inh = int
T	T.type = int
$	

id	id.entry
N	L_1.inh = int
M	L.inh = int
T	T.type=int
$	

Stack	Input Buffer	Action
	, id $	g) shift ,
	$	i) reduce using $L \rightarrow NL, id$ and perform action addType(S[top].entry, S[top-3].inh)
	$	j) reduce using $D \rightarrow TML$ and there's no action to be performed.

USN	1	P	I							

Stack Input Buffer Action

$D $ Accept.

Hence the string is accepted. //

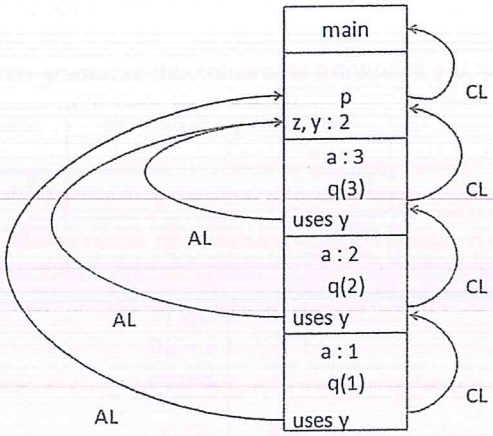
4.	a)	What are the different ways in which a procedure can access non-local data on a run time stack? Explain each technique using a simple example. **Solution:** The different ways in which a procedure can access non-local data on a run time stack are: a) Access Links b) Displays **Access Links:** • Access Link or a Static Link is a pointer to the activation record of the procedure under which it is defined. • Access Links are determined at compile time with the help of nesting depth of procedures. • All C functions are considered to have Nesting depth 1. If a procedure p is defined immediately within a procedure at nesting depth i, then give p the nesting depth i + 1. • Access links form a chain from the activation record at the top of the stack to a sequence of activations at progressively lower nesting depths.	5

USN	1	P	I							

Example:

```
main () {      // ND(main) = 1
  p () {   // ND(p) = 2
   int z, y = 2;
   int q(int a) {   // ND(q) = 3
   if (a=1)
    return 1;
   else
    return(a + y + q(a-1));
     }
   z = q(3);
     }
  p();
  }
```

AL : Access Link, CL: Control Link



Problem with Access Links:

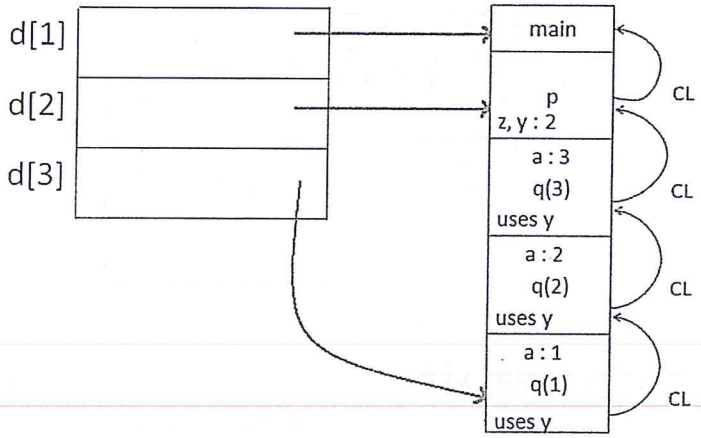
- One problem with the access-link approach to nonlocal data is that if the nesting depth gets large, we may have to follow long chains of links to reach the data we need.

Displays:

- An array of pointers to activation records can be used to access activation records.
- This array is called as displays.
- For each level, there will be an array entry.
- consists of one pointer for each nesting depth
- d[i] is a pointer to the highest activation record on the stack for any procedure at nesting depth *i*.
- Previous value of d[i] must be stored in the current A.R. with depth i.

USN	1	P	I							

Example:



4. b) Generate Target Code for the following procedure call assuming static allocation. 5

	Code is kept at address	Activation record is kept at address
main	100	600
p	400	800

//main()	//p()
n = 6 i = 0 L1 : if i >= 6 goto L2 i = i + 1 goto L1 L2 : call p halt	i = 60 return

Solution:

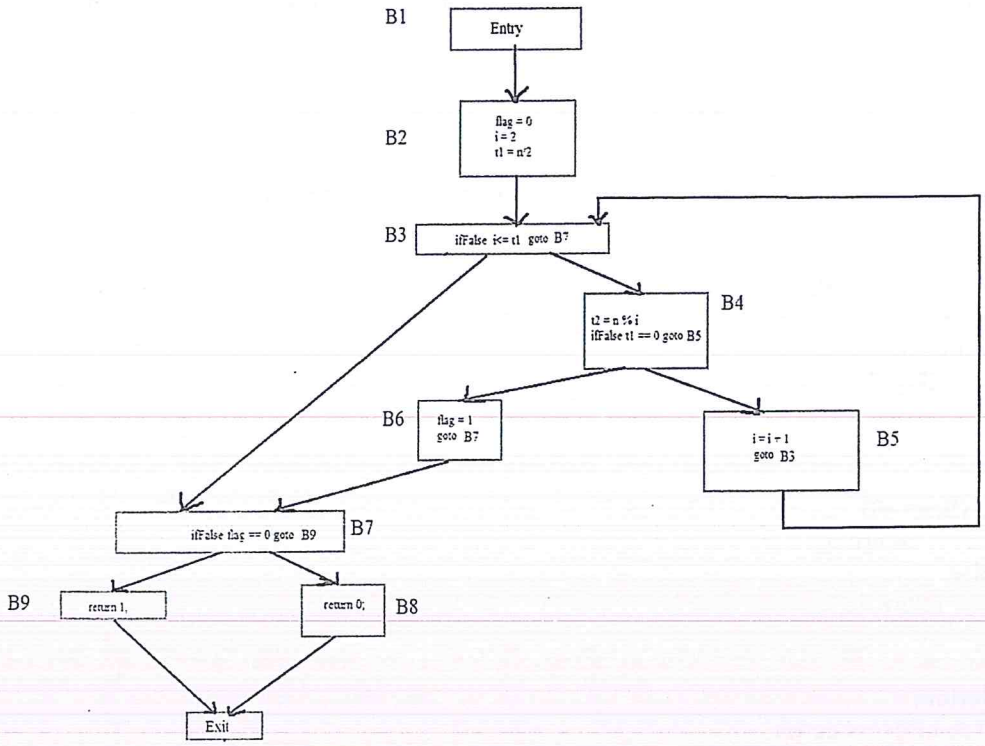
Code Area : contains procedure code	**Static Area : contains Activation Record**
//main() 100: MOV R1, #6 108: ST n, R1 116: MOV R2, #0 124 : ST i, R2 132 : SUB R3, R2, R1 136 : BGEZ #160 144 : ADD R2, R2, #1 152 : BR #124 160 : ST 800, #180 172 : BR 400 180 : HALT	**//main()** 600: .....

USN	1	P	I							

		//p0 400 : MOV R2, #60 408 : ST i, R2 416 : BR *800	//p0 800 : 180 //return address stored in first location of activation record

4.	c)	Generate 3-address code for the following program and convert it into a CFG.
		int prime(int n)
		{
		int n, i, flag = 0;
		for(i=2; i<=n/2; ++i) {
		if(n%i==0){
		flag=1;
		break;
		}
		}
		if (flag==0)
		return 0;
		else
		return 1;
		}
		
		Solution:
		Three-address code:
		flag = 0
		i = 2
		t1 = n/2
		L0 : ifFalse i<= t1 goto L2
		t2 = n % i
		ifFalse t1 == 0 goto L1
		flag = 1
		goto L2
		L1 : i = i + 1
		goto L0
		L2 : ifFalse flag == 0 goto L3
		return 0;
		goto next
		L3 : return 1;
		next :
			10

USN	1	P	I							



5.	a)	What are the issues in the design of a code generator? Code generator phase generates the target code taking input as intermediate code. The output of intermediate code generator may be given directly to code generation or may pass through code optimization before generating code. **Issues in Design of Code generation:** Target code mainly depends on available instruction set and efficient usage of registers. The main issues in design of code generation are • **Intermediate representation:** Linear representation like postfix and three address code or quadruples and graphical representation like Syntax tree or DAG. Assume type checking is done and input in free of errors. This chapter deals only with intermediate representation as three address code. • **Target Code:** The target code may be absolute code, re-locatable machine code or assembly language code. Absolute code can be executed immediately as the addresses are fixed. But in case of re-locatable it requires linker and loader to place the code in appropriate location and map (link) the required library functions. If it generates assembly level code then assemblers are needed to convert it into machine level code before execution. Re-locatable code provides great deal of flexibilities as the functions can be compiled separately before generation of object code.	5

USN	1	P	I							

		◦ **Address mapping:** Address mapping defines the mapping between intermediate representations to address in the target code. These addresses are based on the runtime environment used like static, stack or heap. The identifiers are stored in symbol table during declaration of variables or functions, along with type. Each identifier can be accessed in symbol table based on width of each identifier and offset. The address of the specific instruction (in three address code) can be generated using back patching ◦ **Instruction Set:** The instruction set should be complete in the sense that all operations can be implemented. Sometimes a single operation may be implemented using many instruction (many set of instructions). The code generator should choose the most appropriate instruction. The instruction should be chosen in such a way that speed is of execution is minimum or other machine related resource utilization should be minimum. **Register allocation:** If the operands are in register the execution is faster hence the set of variables whose values are required at a point in the program are to be retained in the registers. Familiarities with the target machine and its instruction set are a pre-requisite for designing a good code generator. Target Machine: Consider a hypothetical byte addressable machine as target machine. It has n general purpose register R1, R2 ------- Rn.	

5.	b)	Optimize the code below by applying the following code transformations: constant propagation, constant folding, copy-propagation, dead-code elimination and strength reduction.
		
		`L0: t1 = t1 + 1`
		`t2 = 0`
		`t3 = t1 * 8`
		`t4 = t3 + t2`
		`t5 = t4 * 4`
		`t6 = *t5`
		`t7 = FP + t3`
		`*t7 = t2`
		`t8 = t1`
		`if (t8 > 0) goto L1`
		`L1: goto L0`
		`L2: t1 = 1`
		`t10 = 16`
		`t11 = t1 * 2`
		`goto L1`
		
		Solution:
		
		Eliminating unreachable code
			5

USN	1	P	I							

```
L0:     t1 = t1 + 1
        t2 = 0
        t3 = t1 * 8
        t4 = t3 + t2
        t5 = t4 * 4
        t6 = *t5
        t7 = FP + t3
        *t7 = t2
        t8 = t1
        if (t8 > 0) goto L1
L1:     goto L0
```

2) Propagating value of t2

```
L0: t1 = t1 + 1
    t2 = 0
    t3 = t1 * 8
    t4 = t3 + 0
    t5 = t4 * 4
    t6 = *t5
    t7 = FP + t3
    *t7 = 0
    t8 = t1
    if (t8 > 0) goto L1
L1: goto L0
```

3) Applying Algebraic simplification:

t4 = t3 + 0 = t3

```
L0: t1 = t1 + 1
    t2 = 0
    t3 = t1 * 8
    t4 = t3
    t5 = t4 * 4
    t6 = *t5
    t7 = FP + t3
    *t7 = 0
    t8 = t1
    if (t8 > 0) goto L1
L1: goto L0
```

4) Copy propagation

t4 = t3; using t3 in place of t4

```
L0: t1 = t1 + 1
    t2 = 0
    t3 = t1 * 8
    t4 = t3
    t5 = t3 * 4
    t6 = *t5
    t7 = FP + t3
    *t7 = 0
    t8 = t1
    if (t8 > 0) goto L1
L1: goto L0
```

DCE : t4 = t3 and t2 = 0 is not used anywhere

USN	1	P	I							

```
L0: t1 = t1 + 1
    t3 = t1 * 8
    t5 = t3 * 4
    t6 = *t5
    t7 = FP + t3
    *t7 = 0
    t8 = t1
    if (t8 > 0) goto L1
L1: goto L0
```

5) Copy propagation : using t1 in place of t8

```
L0: t1 = t1 + 1
    t3 = t1 * 8
    t5 = t3 * 4
    t6 = *t5
    t7 = FP + t3
    *t7 = 0
    t8 = t1
    if (t1 > 0) goto L1
L1: goto L0
```

DCE : t8 = t1 and t6 = * t5

```
L0:  t1 = t1 + 1
     t3 = t1 * 8
     t5 = t3 * 4
     t7 = FP + t3
     *t7 = 0
     if (t1 > 0) goto L1
L1:  goto L0
```

t1, t3 and t5 are induction variables. Performing reduction in strength; Eliminating redundant goto.

t3 = t1 * 8

L0 : t1 = t1 + 1

t3 = t3 + 8

t5 = t3 * 4

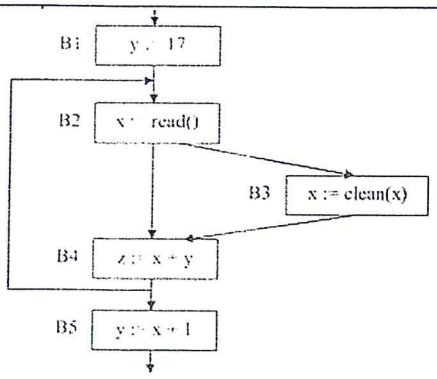
t7 = FP + t3

*t7 = 0

if (t1 > 0) goto L0

5.	c)	Perform Live variable analysis on the following CFG:	5

USN	1	P	I							



	use	def	IN	OUT
B1				
B2				
B3				
B4				
B5				

Solution:

	use	def	IN	OUT
B1	{ }	{ y }	{ }	{ y }
B2	{ }	{ x }	{ y }	{ x, y }
B3	{ x }	{ x }	{ x, y }	{ x, y }
B4	{ x, y }	{ z }	{ x, y }	{ x , y }
B5	{ x }	{ y }	{ x }	{ }

5. d) Construct DAG for the following Block and optimize. 5

1. t1 : = 4 * I
2. t2 : = A − 4
3. t3 : =t2 [t1]
4. t4 : =4 * I
5. t5 : = B − 4
6. t6 : = t5 [t4]
7. t7 : = t3 * t6
8. t8 : = PROD + t7
9. PROD : = t8
10. t9 : = I + 1
11. I = t9
12. if I ≤ 20 goto (1).