SRN										

PES UNIVERSITY, Bangalore

UE18/19CS351

(Established under Karnataka Act No. 16 of 2013)

END SEMESTER ASSESSMENT (ESA) - B.TECH VI SEMESTER – May, 2022

UE18/19CS351 - COMPILER DESIGN

Time: 3 Hrs **Answer All Questions** **Max Marks: 100**

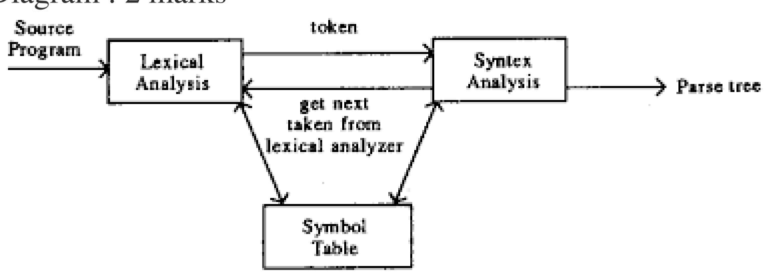
1 a With a neat diagram, explain the interaction between first two phases of the compiler using the following input: **10**

x = a + b * c;

(Note : clearly mention ALL input(s) and output of each phase and their role)

Solution :

Diagram : 2 marks



Role of a Lexer : 1 marks (any 2 points)

- To generate tokens
- To remove comments and extra whitespaces
- Correlate error messages with line numbers

Role of a Parser: 1 marks (any 2 points)

- To validate the syntax of the programming language
- To generate Parse tree as output
- To report syntax errors if any.

Input to lexer : 2 marks

- Source code
- Set of patterns with corresponding rules

Input to Parser : 2 marks

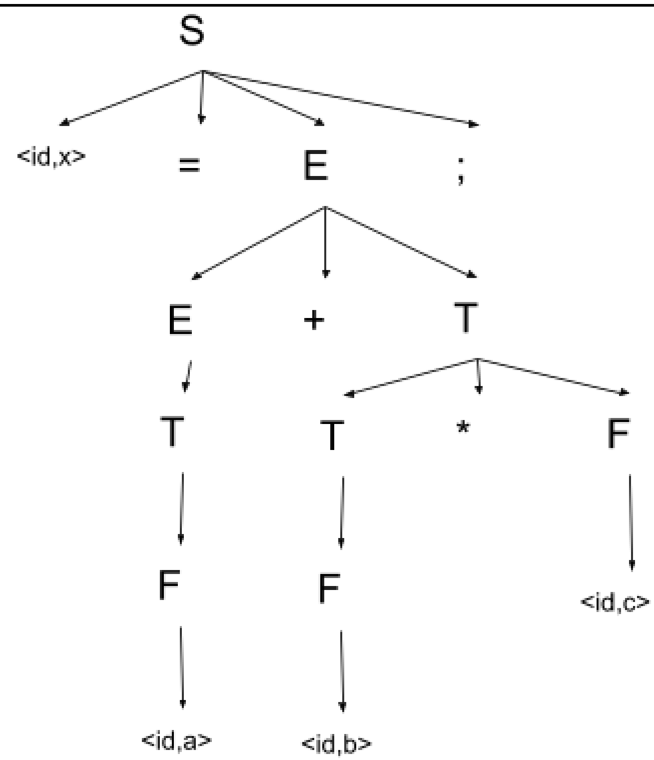
- Tokens generated by lexer
- Grammar file : $S \rightarrow id = E$, $E \rightarrow E + T \mid T$, $T \rightarrow T * F \mid F$, $F \rightarrow id$

Output of lexer : 1 mark

<id, x> < = > <id, a> <arith_op, +> <id, b> <arith_op, *> <id, c> <;>

Output of Parser : 1 mark

SRN



b Given the lex script, answer the following : **5 (2+1+2)**

```
%%
a?aab printf("2");
a?b    printf("1");
aab    printf("3");
```

I. What will be output of a lexer for the input string : **aab**
II. Specify an input string for which the lexer will print **321** as the output.
III. Which of the aforementioned rules are useless and why?

Solution:

I. 2
II. No such input exists
III. aab printf("3"); is useless as aab will always be matched by a?aab pattern

c With a neat diagram explain the design of a lexical analyzer. **5**

Solution:

The program that serves as the lexical analyzer includes a fixed program that simulates an automaton; at this point we leave open whether that automaton is deterministic or nondeterministic. The rest of the lexical analyzer consists of components that are created from the Lex program by Lex itself.

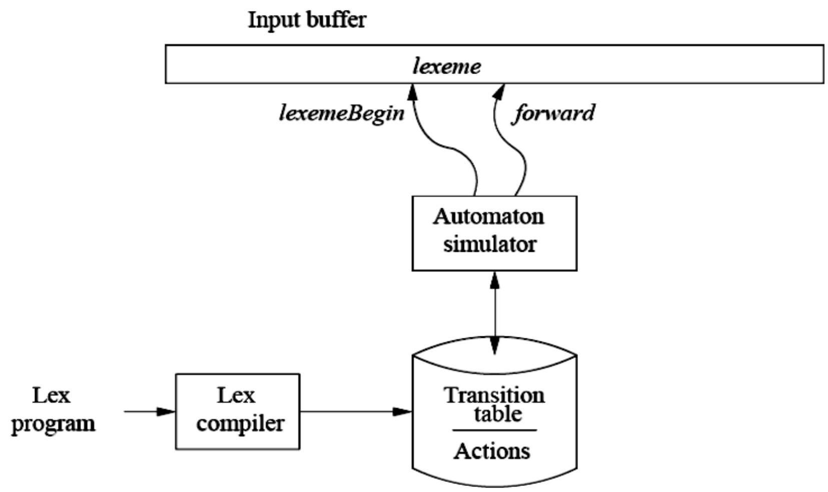
These components are:

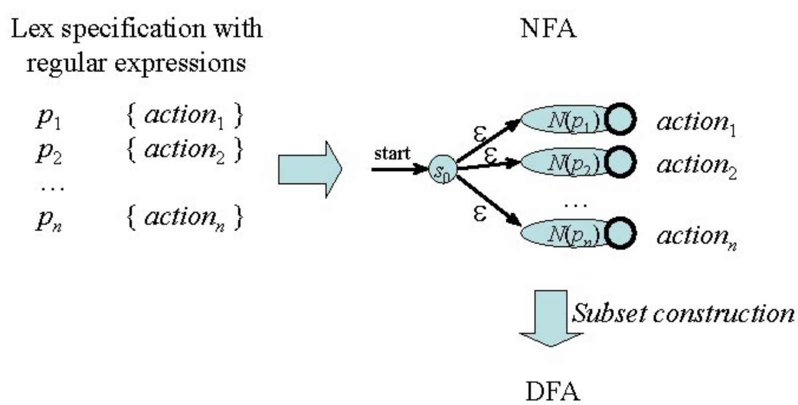
a) A transition table for the automaton.
b) Those functions that are passed directly through Lex to the output
c)The actions from the input program, which appear as fragments of code to be invoked at the appropriate time by the automaton simulator.

To construct the automaton, we begin by taking each regular-expression pattern in the Lex program and converting it to an NFA. We need a single automaton that will recognize lexemes matching any of the

SRN										

patterns in the program, so we combine all the NFA's into one by introducing a new start state with e-transitions to each of the start states of the NFA's *N{* for pattern *pi.*





Any one of the above diagram - 3 marks
explanation - 2 marks

2 a Given the grammar: **10 (2+3+5)**

A → B C | a
B → b C | λ
C → ab

Answer the following :

I. What is first(A)?
II. What is follow(A) and follow(C)?
III. Construct the LL(1) table and specify whether the grammar is in LL(1) or not?

Solution:

I. first(A) = {a, b}
II. follow(A) = \$, follow(C) = {\$, a}
III. LL(1) table

SRN										

	a	b	$
A	A → B C A → a	A → B C	
B	B → λ	B → b C	
C	C → ab		

The grammar is not in LL(1) as M[A,a] has 2 productions resulting into a conflict.

b Given the LALR parsing table, parse the string: ***id=id** 5

Grammar:
1. $S' \to S$
2. $S \to L = R$
3. $S \to R$
4. $L \to * R$
5. $L \to$ **id**
6. $R \to L$

	id	*	=	$	*S*	*L*	*R*
0	s5	s4			1	2	3
1				acc			
2			s6	r6			
3				r3			
4	s5	s4				9	7
5			r5	r5			
6	s5	s4				9	8
7			r4	r4			
8				r2			
9			r6	r6			

Solution:

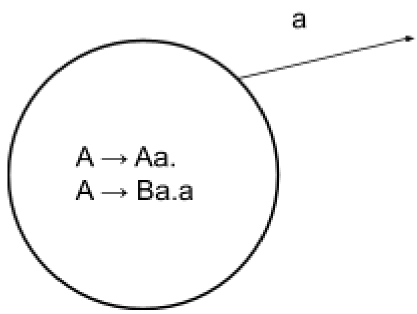
Stack	Input Buffer	Action
$0	*id = id $	A[0,*] = s4
$0*4	id = id $	A[4, id] = s5
$0*4 id 5	= id $	A [5, =] = r5, (L → id)
$0*4L9	= id $	A[9, =] = r6, (R → L)
$0*4R7	= id $	A[7, =] = r4, (L → *R)
$0L2	= id $	A[2, =] = s6
$0L2=6	id $	A[6, id] = s5
$0L2=6id5	$	A[5,$] = r5, (L → id)
$0L2=6L9	$	A[9,$] = r6, R → L
$0L2=6R8	$	A[8,$] = r2, S → L = R
$0S1	$	A[1,S] = accept

SRN										

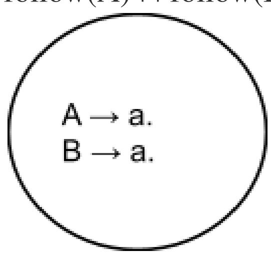
	c	When does s/r and r/r conflict occur in SLR parser? Explain with an example or a dummy state diagram to depict the scenarios.	5

Solution:

A s/r conflict occurs in SLR parser when there is a final item and a shift on a terminal such that follow(A) of final item contains symbol ‘a’ on which the parser is shifting:



A r/r conflict occurs in SLR parser when there are two final items and
follow(A) ∩ follow(B) ≠ Φ



3	a	Write short notes on: I. Types of attributes II. Types of Syntax Directed Definitions III. Procedure to convert SDD to SDT scheme	10 (4+4+2)

Solution:

I. Types of attributes – There are two types of attributes:

1. **Synthesized Attributes** – These are those attributes which derive their values from their children nodes i.e. value of synthesized attribute at node is computed from the values of attributes at children nodes in parse tree.
2. **Inherited Attributes –** These are the attributes which derive their values from their parent or sibling nodes i.e. value of inherited attributes are computed by value of parent or sibling nodes.

II. Types of Syntax Directed Definitions

1. **S-attributed Definitions:** Syntax directed definition that involves only synthesized attributes is called S-attributed.

SRN										

2. **L-attributed Definitions:** The syntax directed definition in which the edges of dependency graph for the attributes in production body, can go from left to right and not from right to left is called L-attributed definitions. Attributes of L-attributed definitions may either be synthesized or inherited.

   If the attributes are inherited, it must be computed from:

   • Inherited attribute associated with the production head.

   • Either by inherited or synthesized attribute associated with the production located to the left of the attribute which is being computed.

   • Either by inherited or synthesized attribute associated with the attribute under consideration in such a way that no cycles can be formed by it in the dependency graph.

III. **Procedure to convert SDD to SDT scheme :**

Synthesized attribute calculation must be kept at the end of the rule [1 mark] inherited attribute calculation must be kept before the non-terminal appears in the production.

b | Write an SDD for simple type declaration consisting of basic type T and list of identifiers L. | 5

D → T L
T → int | float
L → L, id | id
Solution:

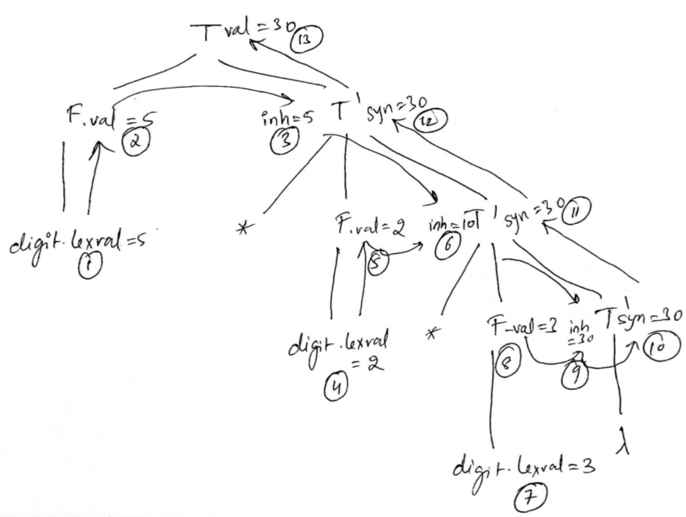
	PRODUCTION	SEMANTIC RULES
1)	$D \to T\ L$	$L.inh = T.type$
2)	$T \to \textbf{int}$	$T.type = \text{integer}$
3)	$T \to \textbf{float}$	$T.type = \text{float}$
4)	$L \to L_1\ ,\ \textbf{id}$	$L_1.inh = L.inh$ $addType(\textbf{id}.entry, L.inh)$
5)	$L \to \textbf{id}$	$addType(\textbf{id}.entry, L.inh)$

SRN										

c Carry out the given SDD over the input string **5 * 2 * 3** and provide an annotated parse tree as output. — 5

	Production	Semantic Rules
1)	$T \to F\,T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \textbf{digit}$	$F.val = \textbf{digit}.lexval$

Solution:



-1 if correct attribute names are not used
-1 if dependency graph not shown.

4 a Answer the following : — 10 (5+5)

I. What is a basic block? [1 mark]
   A. What is a leader? [1 mark]
   B. How are leaders identified? [3 marks]

II. What is Code Optimization? [1 mark]
   Briefly explain the following optimizations with an example:
   A. Constant folding and propagation [2 marks]
   B. Loop optimization (Any one) [2 marks]

Solution:

I. **Basic block :** A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.
   A. The first statement in some block is called a **leader**.
   B. **Identifying Leaders**:
      1. The first statement is a leader.

SRN										

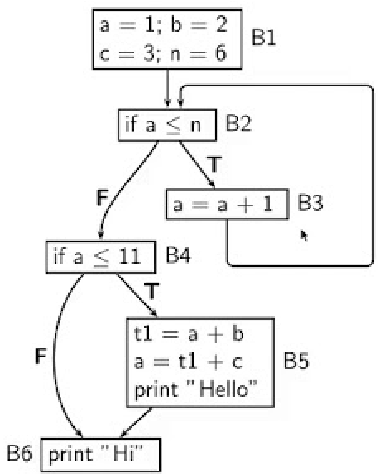
		2. Statement L is a leader if there is an conditional or unconditional goto statement like: if....goto L or goto L.
		3. Instruction L is a leader if it immediately follows a goto or conditional goto statement like: if goto B or goto B.
		II. The **code optimization** in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result.
		A. **Constant Folding :** This is an optimization technique which eliminates expressions that calculate a value that can be determined before code execution. If operands are known at compile time, then the compiler performs the operations statically. An Example,
		int x = (2 + 3) * y → int x = 5 * y
		int z = 300 * 78 * 6 → int z = 140400
		Constant Propagation : This is the substitution of values of known constants and expressions. That is, if the value of a variable is known to be a constant, then the compiler will replace its use by that constant. The value of the variable is propagated forward from the point of assignment. An example,
		int x = 5;
		int y = x * 2;
		optimize...
		int y = 5 * 2;
		optimize...
		int y = 10;
		B. **Loop optimization :** Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops.
		Loop Optimization Techniques: (Any one)
		Frequency Reduction (Code Motion): In frequency reduction, the amount of code in loop is decreased. A statement or expression, which can be moved outside the loop body without affecting the semantics of the program, is moved outside the loop.
		Loop Unrolling:Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.
		or strength reduction using induction variables.	

	b	Convert the given program to SSA form:
		x = a[i];
		if x > n
		  x = x - n;
		else
		  x = x + n;
		y = x * 5;
		Solution:
			5

SRN										

```
x_1 = a[i_0]; //-1 if student uses a_1 or a_0 as array names are immutable)
if x_1 > n_0
   x_2 = x_1 - n_0; //1 mark
else
  x_3 = x_1 + n_0; //1 mark
x_4 = PHI (x_2, x_3); //2 marks
y_1 = x_4 * 5; //1 mark
```

c Given the flow graph, perform live variable analysis. 5



Solution:

Block	use	def	in	out
B1	{ }	{a, b, c, n}	{ }	{a, b, c, n}
B2	{a, n}	{ }	{a, b, c, n}	{a, b, c, n}
B3	{a}	{a}	{a, b, c, n}	{a, b, c, n}
B4	{a}	{ }	{a, b, c}	{a, b, c}
B5	{a, b, c}	{a, t1}	{a, b, c}	{ }
B6	{ }	{ }	{ }	{ }

-1 for each of the highlighted parts, if wrong

5 a Consider the C code to compute Fibonacci numbers recursively. The questions below assume that the initial call is f(5). 10 (5+5)

```
int f(int n) {
    int t, s;
    if (n < 2) return 1;
    s = f(n-1);
    t = f(n-2);
```

SRN										

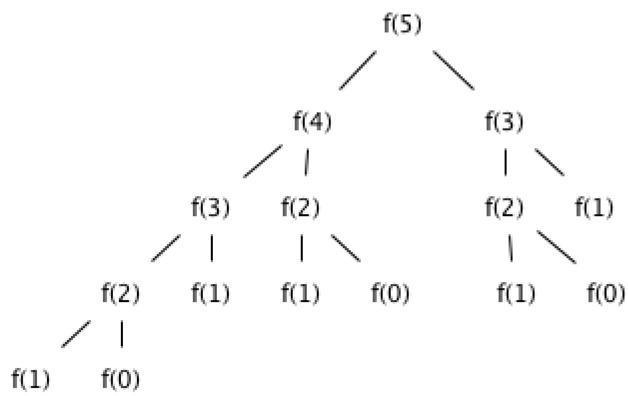
```
  return s+t;
}
```

I. Show the complete activation tree.

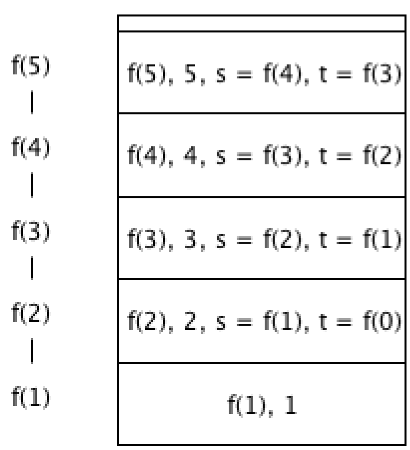
II. How does the stack and its activation records look like the first time f(1) is about to return?

Solution:

I.



II.



b Using Simple Code Generator algorithm, generate target code sequence for the given basic block: **5**

1. x = y+z
2. z = x*x
3. y = z
4. x = y+z

SRN										

Assume number of registers available are 2 i.e. R1 and R2. All the variables (x,y,z) are live on exit from the block.
Solution:

	Target Code Instruction	Register Descriptor		Address Descriptor		
		R1	R2	x	y	z
x= y+z				x	y	z
	LD R1, y	y		x	y,R1	z
	LD R2, z	y	z	x	y,R1	z,R2
	ADD R1, R1, R2	x	z	R1	y	z,R2
z= x * x	MUL R2, R1, R1	x	z	R1	y	R2
y = z		x	z,y	R1	R2	R2
x = y + z	ADD R1, R2, R2	x	z,y	R1	R2	R2
exit	ST x, R1	x	z,y	x,R1	R2	R2
	ST y, R2	x	z,y	x,R1	y,R2	R2
	ST z, R2	x	z,y	x,R1	y,R2	z,R2

c Provide the Activation Record structure. 5

Solution:

fp (frame pointer) →

Returned value
Actual parameters
Optional control link
Optional access link
Save machine status
Local data
Temporaries