NIST INSTITUTE OF SCIENCE & TECHNOLOGY
(AUTONOMOUS)

SUBJECT :

ASSIGNMENT # DUE DATE

NAME :

NIST ROLL # SECTION

Writing space begins here. Do not leave this Page blank.

1. State the various phases of a compiler indicating the inputs and outputs of each phase in translating the statement "position = initial + rate * 60"

Ans. Various phases of a Compiler are :-

(i) Lexical Analysis :-

It is the first phase when compiler scans the source code. Here the character stream from the source program is grouped in meaningful sequences by identifying the tokens.

(ii) Syntax Analysis :-

It is all about discovering structure in code. It determine whether or not a text follows the expected format.

(iii) Semantic Analysis :-

It checks the semantic consistency of the code. It uses the parse tree of the previous phase along with the symbol table to verify the code is semantically consistent or not.

(iv) Intermediate code generation :-

Once the semantic analysis phase is over the compiler generate intermediate code for the target machine. It represents a Program for some abstract machine.

(v) Code optimization :-

This phase removes unnecessary code line arrange them in a sequence to generate a code that runs faster & occupies less space.

(vi) Code generation :-

It gets inputs from code optimization phase & produces the page code or object code as a result.

*) Given,

position = cinitial + rate * 60

position := cinitial + rate * 60

↓

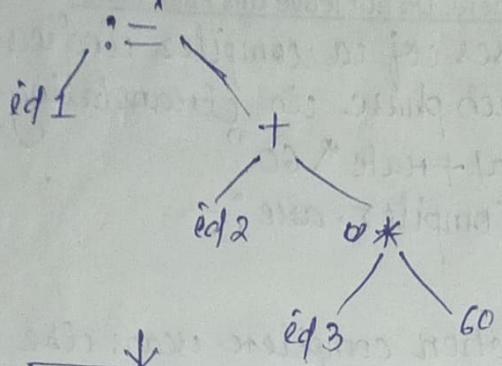
Lexical Analyzer

↓

id1 := id2 + id3 * 60

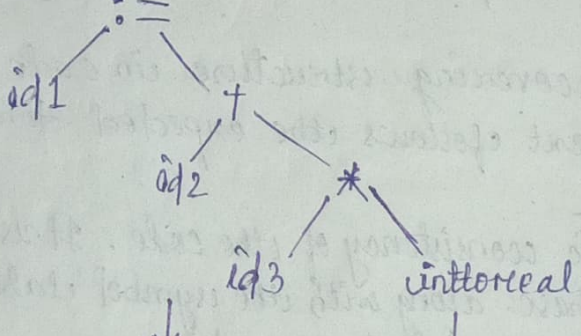
↓

Syntax Analyzer



↓

Semantic Analyzer



↓

intermediate code generator

t_1 = int to real (60)

t_2 = id3 * t_1

t_3 = id2 + t_2

id_1 = t_3

↓

Code optimizer

t_1 = id3 * 60.0

id_1 = id2 + t_1

↓

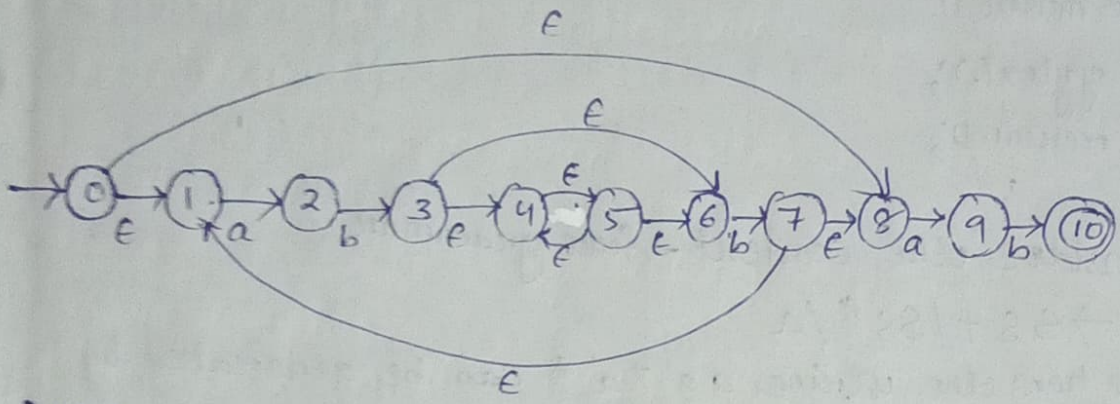
code generator

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id2
```

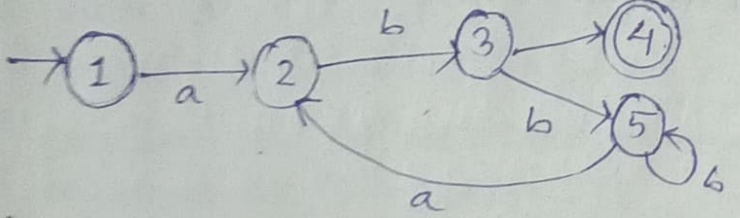
2) Construct the NFA for the RE (ab+b)*ab then to DFA.

Ans. (ab+b)*ab

NFA :-



DFA :-



3) Discuss the concept and functions of Lexical Analyzer Generator. Write a lex program to count the number of words.

Ans. Lexical Analyzer Generator :-

→ A lex is a tool used to generate a lexical analyzer. It translates a set of regular expressions given as input from an input file into a implementation of a corresponding finite state machine.

→ The lexical Analyzer takes in a stream of input characters & returns a stream of tokens.

lex source program (lex.l) → [lex (or flex) Compiler] → lex.yy.c

*) Lex program to count no. of words :-

```
%{
#include <stdio.h>
#include <string.h>
int i=0;
%}
%%
([a-zA-Z0-9])* {i++;}
"\n" {printf("%d\n", i); i=0;}
%%
int yywrap(void) { }
int main()
 { yylex();
   return 0;
 }
```

4) Consider the context-free grammar

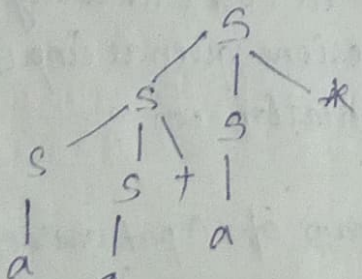
$S \rightarrow SS+ / SS* / a$

(i) Show how the string aa+a* can be generated by this grammar.

Ans:(i) Apply the left most derivative for the string aa+a* we get :

$S \rightarrow SS*$
$S \rightarrow SS+S*$
$S \rightarrow aS+S*$
$S \rightarrow aa+S*$
$S \rightarrow aa+a*$

(ii) Construct a parse tree for this string.



{parse Tree}

(iii) What language does this grammar generate ? Justify your answer.

L = {postfix expression consisting of digits, plus & multiple signs}

5) Explain the concepts and rules of FIRST() and FOLLOW(), Calculate the first and follow functions for the given grammar -

$S \rightarrow aBDh$
$B \rightarrow cC$
$C \rightarrow bC / \epsilon$
$D \rightarrow EF$
$E \rightarrow g / \epsilon$
$F \rightarrow f / \epsilon$

Ans. The First & follow functions are as follows :-

*) First functions :-

- First (S) = {a}
- First (B) = {c}
- First (C) = {b, ε}
- First (D) = {First(E) - ε} ∪ First(F) = {g, f, ε}
- First (E) = {g, ε}
- First (F) = {f, ε}

*) Follow functions :-

- Follow (S) = {$}
- Follow (B) = {First (D) - ε} ∪ First (h) = {g, f, h}
- Follow (C) = follow (B) = {g, f, h}
- Follow (D) = First (h) = {h}
- Follow (E) = {First (F) - ε} ∪ Follow (D) = {f, h}
- Follow (F) = follow (D) = {h}

6) Consider the context - free grammar where *, + and a are the terminals $S \rightarrow SS*/SS+/a$

(i) Describe the language described by this grammar.

Ans. $S \rightarrow SS*/SS+/a$

(i) The language described by the grammar is

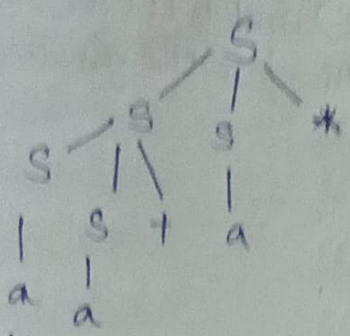
L = {postfix expression consisting of digits, plus and multiple signs}

because this grammar contains a plus & multiply sign at the end.

(ii) Is this grammar ambiguous ? Explain your answer.

Ans.

(ii) The grammar is unambigous grammar because it doesn't contain more than one left most derivation or more than one right most derivation or more than one pass tree for the given input String.



As we can see there is only one parse tree, hence the grammar is unambiguous.

7) Construct recursive-descent parsers starting with the below grammar.

$S \rightarrow +SS/-SS/a$

Ans. $S \rightarrow +SS/-SS/a$

$S \rightarrow +SS$

$S \rightarrow -SS$

$S \rightarrow a$

```
Void S() {
    Switch (look ahead) {
        case "+":
        match ("+"); S(); S();
          break;
        case "-":
        match ("-"); S(); S();
         break;
         case "a";
          match ("a");
           break;
         default:
           throw new SyntaxException();
        }
    }
    void match(Teremenal t) {
       if(look ahead = t){
          look ahead = next Teremenal();
        } else {
              throw new Syntax Exception()
        }
    }
```

8) Consider the following grammar

$E \to E+T/T$

$T \to T*F/F$

$F \to (E)/id$

(a) Find the non-recursive predictive parser for the above grammar.

Ans:(a) Eliminate immediate left recursion

$E \to TE'$

$E' \to +TE'/\epsilon$

$T \to FT'$

$T' \to *FT'/\epsilon$

$F \to (E)/id$

First(F) = { (, id }

first (T') = { *, ε }

first (T) = { (, id }

First (E') = {+, ε }

first (E) = { (, id }

First (TE') = {(, id }

First (+TE') = {+}

First (ε)' = {ε}

First (FT') = {(, id}

First (*FT') = {*}

First ((E)) = {(}

First (id) = {id}

Follow (E) = { $,) }

Follow (E') = { $,) }

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

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

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

LL(1) parsing Table :-

	id	+	*	(	)	$
E	$E+ \to TE'$			$E+ \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$F' \to \epsilon$
T	$T \to FT'$			$T' \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T \to \epsilon$
F	$F \to id$			$F \to (E)$		

(b) Show the parsing of the string "((id+id)*id)+id" using the parsing table constructed above.

(b) ((id + id) * id) + id

Stack	Input	action
$E	((id+id)*id)+id$	—
$E'T	((id+id)*id)+id$	E → TE'
$E'T'F	((id+id)*id)+id$	T' → FT'
$E'T')E(	((id+id)*id)+id$	F → (E)
$E'T')E	(id+id)*id)+id$	POP (
$E'T')E'T	(id+id)*id)+id$	E → TE'
$E'T')E'T'F	(id+id)*id)+id$	T' → FT'
$E'T')E'T')E(	(id+id)*id)+id$	F → (E)
$E'T')E'T')E	id+id)*id)+id$	POP (
$E'T')E'T')E'T	id+id)*id)+id$	E → TE'
$E'T')E'T')E'T'F	id+id)*id)+id$	T → FT'
$E'T')E'T')E'T'id	id+id)*id)+id$	F → id
$E'T')E'T')E'T'	+id)*id)+id$	pop id
$E'T')E'T')E'	+id)*id)+id$	T' → ε
$E'T')E'T')E'T+	+id)*id)+id$	E' → +TE'
$E'T')E'T')E'T	id)*id)+id$	pop +
$E'T')E'T')E'T'F	id)*id)+id$	T → FT'
$E'T')E'T')E'T'id	id)*id)+id$	F → id
$E'T')E'T')E'T'	)*id)+id$	pop id
$(E'T')E'T')E'	)*id)+id$	T' → ε
$E'T')E'T')	)*id)+id$	E' → ε
$E'T')E'T'	*id)+id$	pop)
$E'T')E'T'F*	*id)+id$	T' → *FT'
$E'T')E'T'F	id)+id$	pop *
$E'T')E'T'id	id)+id$	F → id
$E'T')E'T'	)+id$	pop id
$E'T')E'	)+id$	T' → ε
$E'T')	)+id$	E' → ε
$E'T'	+id$	pop)
$E'	+id$	T' → ε
$E'T+	+id$	E' → +TE'
$E'T	id$	pop +
$E'T'F	id$	T → FT'

$E'T'id	id $	F → id
$E'T'	$	pop id
$E'	$	T' → ε
$	$	E' → ε

parsing is successful

9) check whether following grammar is LL(1) or not

S → iEtSA/a
A → eS/t
E → b

Ans. S → iEtSA/a
A → eS/t
E → b

Follow(S) = {$, e}
Follow(A) = {$, e}
follow(E) = {t}

First(iEtSA) = {i}
First(a) = {a}
First(eS) = {e}
First(t) = {t}
First(b) = {b}

	a	b	e	i	t	$
S	S→a			S→iEtSA		
A			A→eS		A→t	
E		E→b				

As any of the grammar in the parsing table contain more than one production rule. so it is a LL(1) grammar.

10) For the grammar E → E+E/E*E/(E)/id. show various shift reduce parsing action with respect to input string id1+id2*+id3

Ans.

E → E+E/E*E/(E)/id
E → E+E
E → E*E
E → (E)
E → id

id1 + id2 + id3

Stack	Input buffer	parsing action
$	id1+id2*id3$	Shift
$id1	+id2*id3$	Reduce E→id
$E	+id2*id3$	Shift
$E+	id2*id3$	Shift
$E+id2	*id3$	Reduce E→id2
$E+E	*id3$	Shift
$E+E*	id3$	Shift
$E-E*id3	$	Reduce E→id3
$E-E*E	$	Reduce E→E*E
$E-E	$	Reduce E→E*E
$E	$	Accept