NIST INSTITUTE OF SCIENCE & TECHNOLOGY
(AUTONOMOUS)

SUBJECT : Compiler Design

ASSIGNMENT # 01 DUE DATE

NAME : B.Sai Likhith

NIST ROLL # 201912411 SECTION CSE-C

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

① 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 determines 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.

② Given,

position = initial + rate * 60

position := initial + rate * 60

↓

Lexical Analyzer

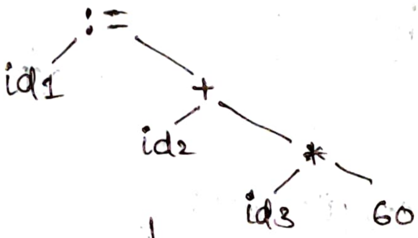
↓

id1 := id2 + id3 * 60

↓

syntax Analyzer

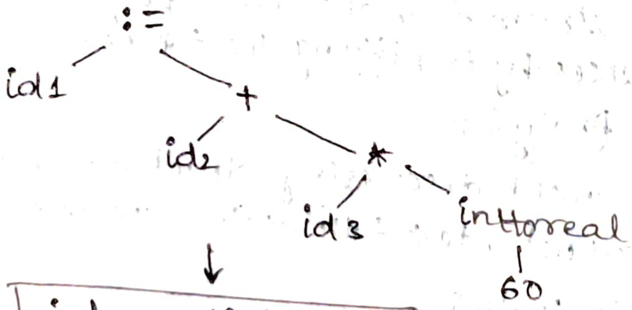
↓



↓

Semantic Analyzer

↓



↓

intermediate code generator

↓

```
t1 = inttoreal(60)
t2 = id3 * t1
t3 = id2 + t2
td1 = t3
```

↓

code optimizer

↓

```
t1 = id3 * 60.0
id1 = id2 + t1
```

↓

code generator

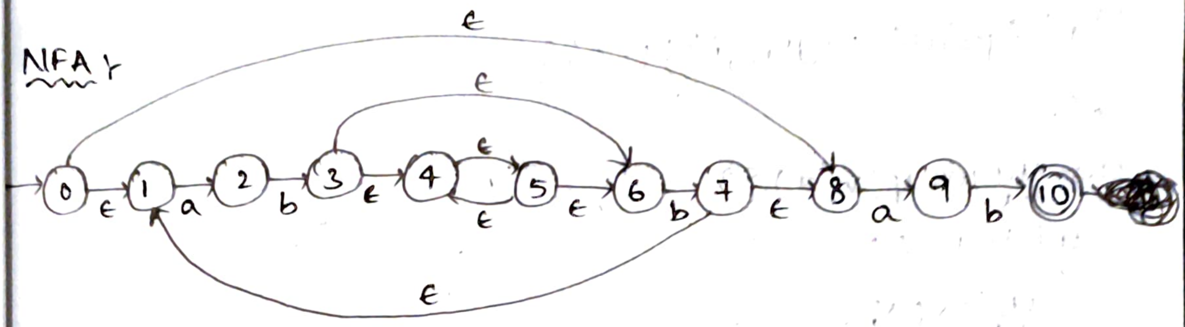
↓

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

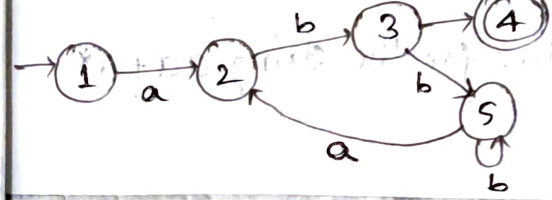
②. R.E

(ab+b)*ab.

NFA :-



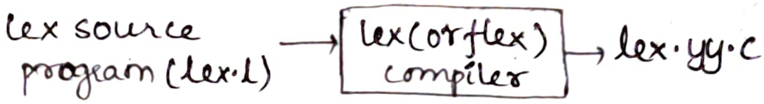
DFA :-

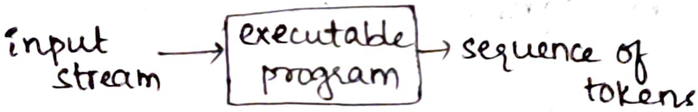


③. Lexical Analyzer Generator :-

→ 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 C implementation of a corresponding finite state machine.

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





⊛ Lex program to count no. of words :-

```
%{
 #include <stdio.h>
 #include <string.h>
  int i=0;
 %}
```

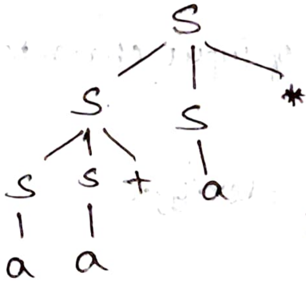
```
%%
([a-zA-Z0-9])* {i++;}
"\n" {printf("%d\n",i);i=0;}
%%
int yywrap(void){}
 int main()
 { yylex();
   return 0;
 }
```

---

④ (i) Apply the left most derivative for the string aa+a*
we get:

S → SS*
S → SS+S*
S → aS+S*
S → aa+S*
S → aa+a*

(ii)



{parse tree}

(iii) $\mathcal{L}$ = {postfix expression consisting of digits, plus & multiple signs}

---

⑤ The first & follow functions are as follows:-

(*) <u>First functions</u>:

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

---

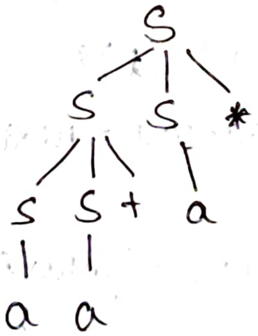
⑥ S → SS* | SS+ | a

ⓘ 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.

ⓘⓘ The grammar is unambiguous grammar because it doesn't contain more than one left most derivation (or) more than one right most derivation (or) more than one parse tree for the given input string.



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

⑦ $S \to +SS \mid -SS \mid a$

$S \to +SS$

$S \to -SS$

$S \to a$

```
void S(){
    switch (lookahead){
            case "+";
                match ("+"); S(); S();
                    break;
            case "-";
                match ("-"); S(); S();
                break;
            case "a";
                match ("a");
                    break;
        default:
            throw new syntaxException( );
        }
    }

void match (Terminal t){
        if (lookahead = t){
                lookahead = next Terminal ();
        }else{
            throw new SyntaxException ()
        }
    }
```

⑧ ⓐ Eliminate immediate left recursion.

$E \to TE'$

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

$T \to FT'$

$T' \to *FT' \mid \varepsilon$

$F \to (E) \mid id$

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

1. ⓑ ((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'
$ET')E(	((id+id)*id)+id$	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 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.

(1a) 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.

×

Submitted by,
B. Sai Likhith
201912411
CSE-C.