KULVIR SINGH

19BCE20724

(23) 1) Handle :

A handle of a string is a substring that matches the right side of a production and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Handle Pruning :

In compiler design handle pruning is used to obtain a rightmost derivation in reverse. For eg., start with a string of terminals w that is to parse. In case w is a sentence of grammar then $w = \gamma_n$ where γ_n is the nth right sentential form of some as yet unknown rightmost derivation.

2) LR Parsing Table

$E \longrightarrow E + T$

$E \longrightarrow T$

$T \longrightarrow T * F$

$T \longrightarrow F$

$F \longrightarrow (E)$

$F \longrightarrow id$

id * id + id

Assuming E is start Symbol.

$E' \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

I_0 (Augmented grammar)

$E' \rightarrow .E, \$$

$E \rightarrow .E + T, \$/+$

$E \rightarrow .T, \$/+$

$T \rightarrow .T * F, \$/+/*$

$T \rightarrow .F, \$/+/*$

$F \rightarrow .(E), \$/+/*$

$F \rightarrow .id, \$/+/*$

I_1 goto (0, E)

$E' \rightarrow E., \$$

$E \rightarrow E. + T, \$/+$

I_2 goto (0, T)

$E \rightarrow T., \$/+$

$T \rightarrow T. * F, \$/+/*$

I_3 goto (0, F)

$T \rightarrow F., \$/+/*$

I_4 goto (0, ()

$F \rightarrow (.E), \$/+/*$

$E \rightarrow .E + T,)/+$

$E \rightarrow .T,)/+$

$T \rightarrow .T * F,)/+/*$

$T \rightarrow .F,)/+/*$

$F \rightarrow .(E),)/+/*$

$F \rightarrow .id,)/+/*$

I_5 goto (0, id)

$F \rightarrow id., \$/+/*$

I_6 goto (1, +)

$E \rightarrow E + .T, \$/+$

$T \rightarrow .T * F, \$/+/*$

$T \rightarrow .F, \$/+/*$

$F \rightarrow .(E), \$/+/*$

$F \rightarrow .id, \$/+/*$

I_7 goto(2, *)

$T \rightarrow T*.F$, $\$/+/*$

$F \rightarrow .(E)$, $\$/+/*$

$F \rightarrow .id$, $\$/+/*$

I_8 goto(4, E)

$F \rightarrow (E.)$, $\$/+/*$

$E \rightarrow E.+T$, $)/+$

I_9 goto(4, T)

$E \rightarrow T.$, $)/+$

$T \rightarrow T.*F$, $)/+/*$

I_{10} goto(4, F)

$T \rightarrow F.$, $)/+/*$

I_{11} goto(4, ()

$F \rightarrow (.E)$, $)/+/*$

$E \rightarrow .E+T$, $)/+$

$E \rightarrow .T$, $)/+$

$T \rightarrow .T*F$, $)/+/*$

$T \rightarrow .F$, $)/+/*$

$F \rightarrow .(E)$, $)/+/*$

$F \rightarrow .id$, $)/+/*$

I_{12} goto(4, id)

$F \rightarrow id.$, $)/+/*$

I_{13} goto(6, T)

$E \rightarrow E+T.$, $\$/+$

$T \rightarrow T.*F$, $\$/+/*$

I_{14} goto(7, F)

$T \rightarrow T*F.$, $\$/+/*$

I_{15} goto(8,))

$F \rightarrow (E).$, $\$/+/*$

I_{16} goto(8, +)

$E \rightarrow E+.T$, $)/+$

$T \rightarrow .T*F$, $)/+/*$

$T \rightarrow .F$, $)/+/*$

$F \rightarrow .(E)$, $)/+/*$

$F \rightarrow .id$, $)/+/*$

I_{17} goto(9, *)

$T \rightarrow T*.F$, $)/+/*$

$F \rightarrow .(E)$, $)/+/*$

$F \rightarrow .id$, $)/+/*$

I_{18} goto(11, E)

$F \rightarrow (E.)$, $)/+/*$

$E \rightarrow E.+T$, $)/+$

I_{19} goto (16, T)

$E \rightarrow E + T.$, $)/+$

$T \rightarrow T. * F$, $)/+/*$

I_{20} goto (17, F)

$T \rightarrow T * F.$, $)/+/*$

I_{21} goto (18,))

$F \rightarrow (E).$, $)/+/*$

(36) LALR(1) Parser

$S \rightarrow iCtS \mid CtSeS \mid a$

$C \rightarrow L.$

Start Symbol S.

$S' \rightarrow S$

$S \rightarrow iCtS$

$S \rightarrow CtSeS$

$S \rightarrow a$

$C \rightarrow L$

I_0 (Augmented Grammar)

$S' \rightarrow .S$, $

$S \rightarrow .iCtS$, $

$S \rightarrow .CtSeS$, $

$S \rightarrow .a$, $

$C \rightarrow .B$, t

I_1 goto (0, S)

$S' \rightarrow S.$, $

I_2 goto (0, i)

$S \rightarrow i.CtS$, $/e

$C \rightarrow .B$, t

I_3 goto (0, C)

$S \rightarrow C.tSeS$, $/e

I_4 goto (0, a)

$S \rightarrow a.$, $/e

I_5 goto (0, B)

$C \rightarrow B.$, t

I_6 goto (2, C)

$S \rightarrow iC.tS$, $/e

I_7 goto (3, t)

$S \rightarrow Ct.SeS$, $/e

$S \rightarrow .iCtS$, e

$S \rightarrow .CtSeS$, e

$S \rightarrow .a$, e

$C \rightarrow .B$, t

I_8 goto (6, t)

$S \rightarrow iCt.S$, $/e

I_9 goto(7, S)

$S \rightarrow$ Ct S . eS , $/e

I_{10} goto(8, s)

$S \rightarrow$ i Cts , $/e

I_{11} goto(9, e)

$S \rightarrow$ Cts e.S , $/e

$S \rightarrow$ .iCts , $/e

$S \rightarrow$ .CtSeS , $/e

$S \rightarrow$ .a , $/e

$C \rightarrow$ .B , t

I_{12} goto(11, s)

$S \rightarrow$ CtSeS. , $/e

(33) 1) Predictive Parsers are Top Down Parsers

There are 2 type of Predictive Parsers

(i) Recursive discent Parser

(ii) Non-Recursive discent or LL(1) Parser

Recursive discent parser works with backtracking parser. It basically generates the parse tree by using Brute Force and backtracking

LL(1) Parser does not use backtracking. It uses passing table to generate the parse tree instead of backtracking

2) Predictive Parser for :-

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid id$

id + id * id

Removing left recursion we get.

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'$

$E' \rightarrow \epsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'$

$T' \rightarrow \epsilon$

$F \rightarrow (E)$

$F \rightarrow id$

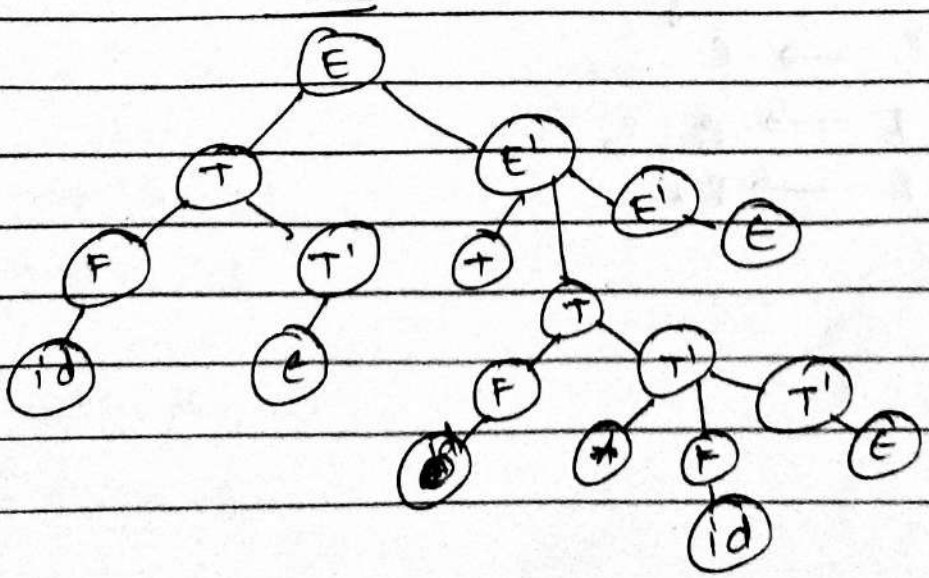
Predictive Parser Table

Non-Terminal	First	Follow	+	*	(	)	id	$
E	(, id	$,)			$E \to TE'$		$E \to TE'$	
E'	+, ϵ	$,)	$E' \to +TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T	(, id	+, $,)			$T \to FT'$		$T \to FT'$	
T'	*, ϵ	+, $,)	$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F	(, id	*, +, $,)			$F \to (E)$		$F \to id$	

Tracing Table

Stack	Input	Rule
$ E	id + id * id $	E → TE'
$ E' T	id + id * id $	T → FT'
$ E' T' F	id + id * id $	F → id
$ E' T' id	id + id * id $	
$ E' T'	+ id * id $	T' → ε
$ E'	+ id * id $	E' → + TE'
$ E' T +	+ id * id $	
$ E' T	id * id $	T → FT'
$ E' T' F	id * id $	F → id
$ E' T' id	id * id $	
$ E' T'	* id $	T' → * FT'
$ E' T' F *	* id $	
$ E' T' F	id $	F → id
$ E' T' id	id $	
$ E' T'	$	T' → ε
$ E'	$	E' → ε
$	$	

Tree



(Q4)

17 Symbol Table

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable name, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler

B =

LR(1) Parser.

$S \longrightarrow x \mid Ay$

$B \longrightarrow \epsilon \mid \$$

$A \longrightarrow Bx$

$S' \longrightarrow S$

$S \longrightarrow x$

$S \longrightarrow Ay$

$B \longrightarrow \epsilon$

$B \longrightarrow \$$

$A \longrightarrow Bx$

I_0 (Augmented Grammar)

$S' \rightarrow . S \;,\; \$$

$S \rightarrow . x \;,\; \$$

$S \rightarrow . Ay \;,\; \$$

$A \rightarrow . Bx \;,\; y$

$B \rightarrow . \;,\; x$

$B \rightarrow . z \;,\; x$

I_1 goto $(0, S)$

$S' \rightarrow S. \;,\; \$$

I_2 goto $(0, x)$

$S \rightarrow x. \;,\; \$$

I_3 goto $(0, A)$

$S \rightarrow A.y \;,\; \$$

I_4 goto $(0, B)$

$A \rightarrow B.x \;,\; y$

I_5 goto $(0, z)$

$B \rightarrow z. \;,\; x$

I_6 goto $(3, y)$

$S \rightarrow Ay. \;,\; \$$

I_7 goto $(4, x)$

$A \rightarrow Bx. \;,\; y$

First (S') = { x, ∈, z, y }

First (S) = { x, ∈, z, y }

First (B) = { ∈, z }

First (A) = { ∈, z, x }

LR(1) Table

State	ACTION				GOTO			
	x	y	z	$	S'	S	B	A
0	s2/r3		s5			1	4	3
1				acc				
2				r_1				
3		s6						
4	s7							
5	r_4							
6				r_2				
7		r_5						

25 1) LL(1)

$S \longrightarrow iEtSS' \mid a$

$S' \longrightarrow eS \mid \epsilon$

$E \longrightarrow b$.

First (S) = {i, a}

First (S') = {e, ε}

First (E) = {b}

Follow (S) = {$, e}

Follow (S') = {$, e}

Follow (E) = {t}

Predictive Parsing Table

Non-Terminal	i	t	a	e	b	$
S	$S \rightarrow iEtSS'$		$S \rightarrow a$			
S'				$S' \rightarrow eS \mid \epsilon$		$S' \rightarrow \epsilon$
E					$E \rightarrow b$	

2) Three address code:

S.begin = new-label() = L1

E.true = new-label() = L2

E.code = "if i < 10 goto"

E.false = S.next = L next

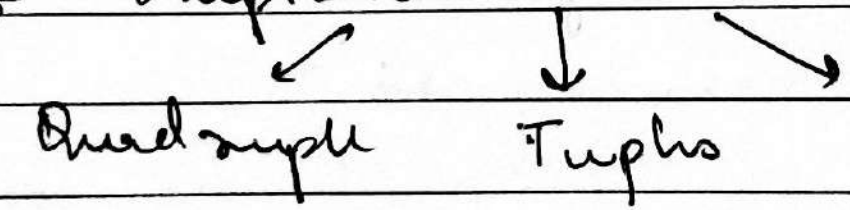
S1.code = x = 0

i = i + 1.

(39) ⇒ Three Address code

It is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most 3 address and one operator to represent an expression and the value outputted at each instruction is started in temporary variable generated by compiler

Implementation



Eg 1

a * -(b + c) → 3 address code

t1 = b + c

t2 = uminus t1

t3 = a * t2

a) $S \rightarrow a \mid \uparrow \mid (T)$

$T \rightarrow T, S \mid S$

Preliminary Steps:

1) Augmentation of grammar

$S \rightarrow a \mid \uparrow \mid (T)$

$T \rightarrow T, S \mid S$

$\boxed{S' \rightarrow S}$

$S \rightarrow a$

$S \rightarrow \uparrow$

$S \rightarrow (T)$

$T \rightarrow T, S$

$T \rightarrow S$

2) Remove left factoring

$S' \rightarrow S$

$S \rightarrow a \mid \uparrow \mid (T)$

$T \rightarrow T, S \mid S$

3) Remove left recursion (No left recursion)

4) First & Follow :-

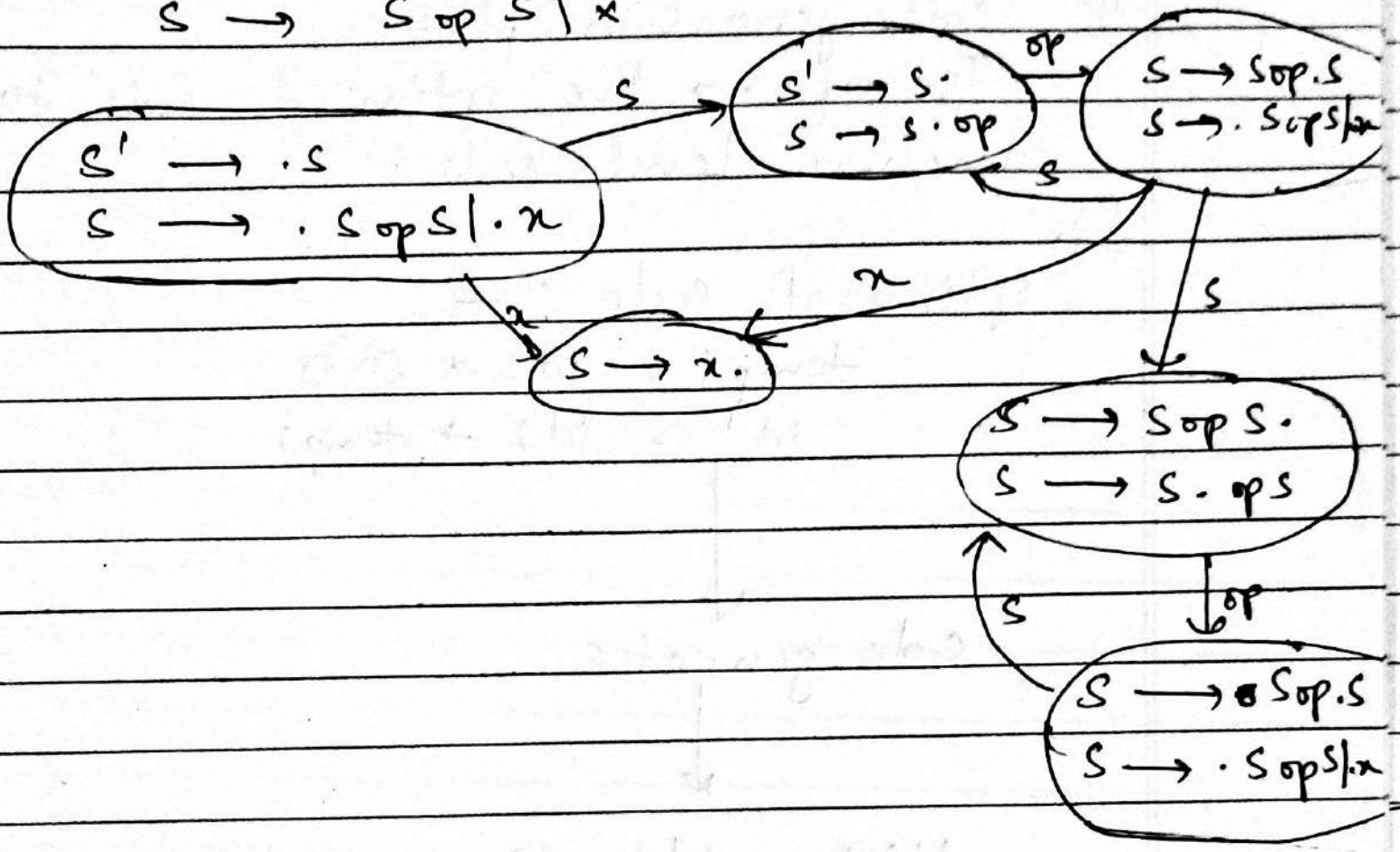
$S' \rightarrow S$

$S \rightarrow a \mid \uparrow \mid (T)$

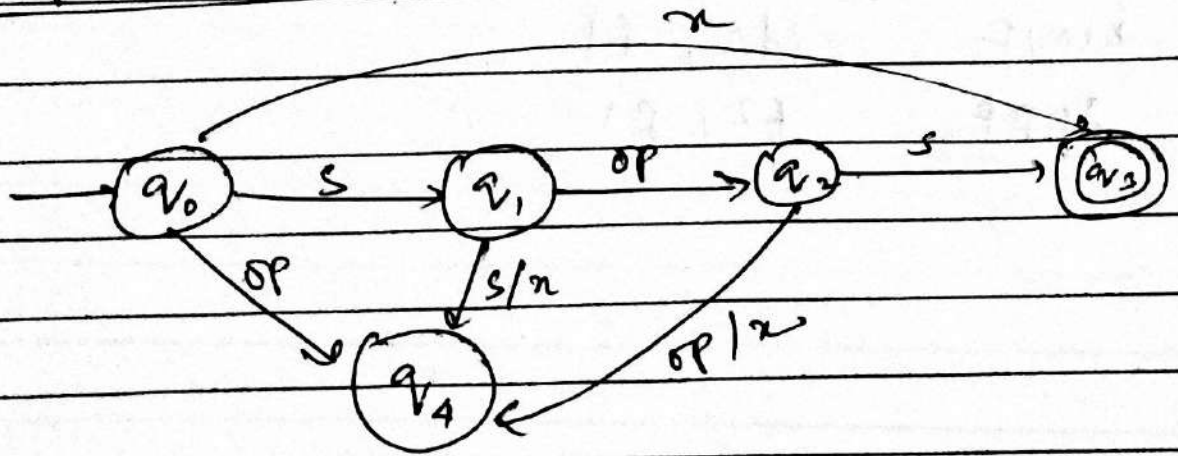
$T \rightarrow T, S \mid S$

	First	Follow
S'	a, ↑, (	$
S	a, ↑, (	$,), ,
T	∈	), ,

(8) 1) $S \rightarrow S\ op\ S \mid x$



Required DFA



1) Issues of lexical analyser

There are several reasons for the analysis phase of compiling into lexical analysis and parsing:

i) Compiler efficiency is improved

ii) compiler portability is enhanced

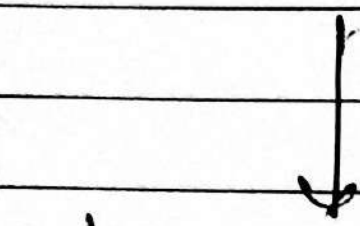
iii) simple design is perhaps the most important consideration. The separation of lexical analysis often allows us to simplify one of these phases.

(9) ⇒ Code generation phase
Transforms the optimised code to machine level code.

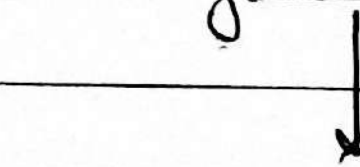
Eg ⇒

Optimised code :—

```
temp1 = id3 * 50.0
id1 = id2 + temp1
```



Code generator



```
MOV     id3, R2
MULF    #50.0, R2
MOVF    id2, R1
ADDF    R2, R1
```

1) Issues of lexical analyzer

There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing:

(i) Compiler efficiency is improved

(ii) compiler portability is enhanced

(iii) simple design is perhaps the most important consideration. The separation of lexical analysis often allows us to simplify one of these phases.

(40) 1) Issues of lexical analyzer

There are several reasons for the analysis phase of compiling into lexical analysis and parsing:

(i) Compiler efficiency is improved

(ii) compiler portability is enhanced

(iii) simple design is perhaps the most important consideration. The separation of lexical analysis often allows us to simplify one of these phases.

a) Common sub expression elimination :

CSE is a compiler optimisation that searches for instances of identical expressions, ie, they evaluate to the same value and analyse if its worth while replacing with a single variable

Eg code:

```
a = b * c + g;
d = b * c * e;
```

Transformed code :

```
temp = b * c;
a = temp + g;
d = temp * e;
```

b) Copy propagation :

Process of replacing the occurrences of targets of direct assignments with their values.

Code :

```
y = x;
z = 3 + y;
```

Copy Propagation :

```
z = 3 + x;
```

c) Dead code elimination :

Removes code which does not affect the program results.

Eg

```
Code :
int global;
void f() {
int i;
 i = 1;
global = 1;
global = 2;
return;
 global = 3; }
```

Removing dead code :-

```
int global;
void f() {
 global = 2;
  return; }
```

d) Code Motion :

Refers that the code is moved out of the loop as it won't have any difference if it is performed inside the loop repeatedly or outside the loop once.

Eg

```
for (int x = 0; x < string.length(); x++)
{ // Other code here
}
```