C.C.

* Why ~~independent~~ intermediate code generation is required?
(Hint: platform independance.)

```
# include <stdio.h>
# define Pi 3.14
main()
{
  int a,r;
  a = Pi * r * r;
}
```

expand

Replace by 3.14

↓ .c HLL

[preprocessor]

↓ pure HLL

[compiler]

↓ .asm Assembly language

[assembler]

↓ Relocatable machine code

[Linker/Loader] → Machine code

→ Other tool that uses same process as compiler:

[Source language]

→ Analysis phase: Lexical Analysis
Syntax
Semantic

Synthesis Phase: Intermediate code generation
Code optimization
final code generation

[Target]

Identifier

a = b + c * 6

operator

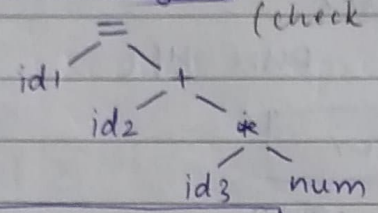
↓

Lexical Analysis (check using regular expression)

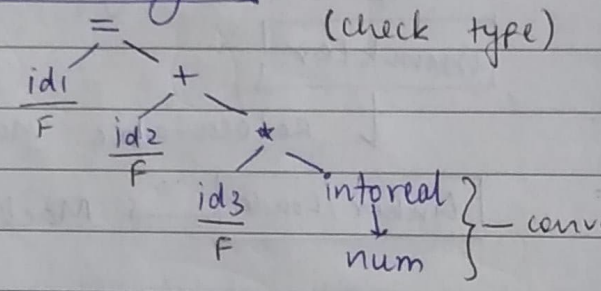
id1 = id2 + id3 * num (draws DFA)

↓

Syntax Analysis (check using grammar) (draws tree)



Semantic Analysis (check type)



Intermediate Code Generator (only one variable at a time)

t_1 = intoreal (num)

t_2 = ~~id~~ t_1 * id3

t_3 = id2 + t_2

id1 = t_3

Code optimizer

t_1 = id3 * intoreal (6)

id_1 = id2 + t_1

Code generation (code using registers)

```
MOVF  id3, R2
MULF  6, R2
MOVF  id2, R1
ADDF  R2, R1
MOVF  R1, id1
```

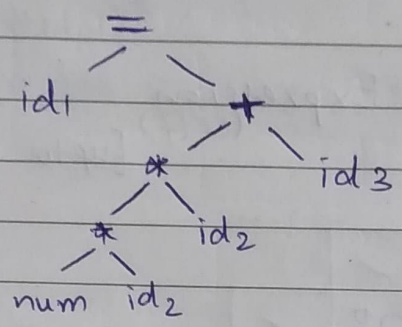
Ex:

* $a = \pi r^2 + v$

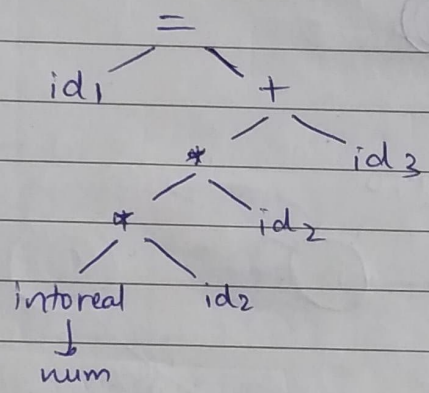
$a = 3.14\, r^2 + v$

Lex: $id_1 = num * id_2 * id_2 + id_3$

Syn:



Sem:



ICG: ~~t_1 = intoreal(num)~~ ~~$t_1 = id_2 * id_2 * 3$~~

$t_1 = num$

$t_2 = t_1 * id_2$

$t_3 = t_2 * id_2$

$t_4 = t_3 + id_3$

$id_1 = t_4$

CO: $t_1 = 3.14 * id_2 * id_2$

$id_1 = t_1 + id_3$

CG:

```
MOVF  id2 , R2
MULF  R2  , R2
MULF  3.14, R2
MOVF  id3 , R1
ADDF  R2  , R1
MOVF  R1  , id1
```

Single pass:

1) Forward reference problem : (encounter variable before declaration) (solution: back patching)
2) Carry whole code

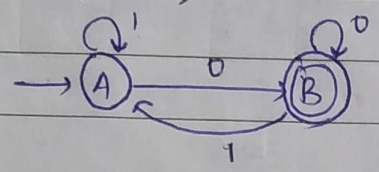
Multi pass:

1) Require more time (complex)

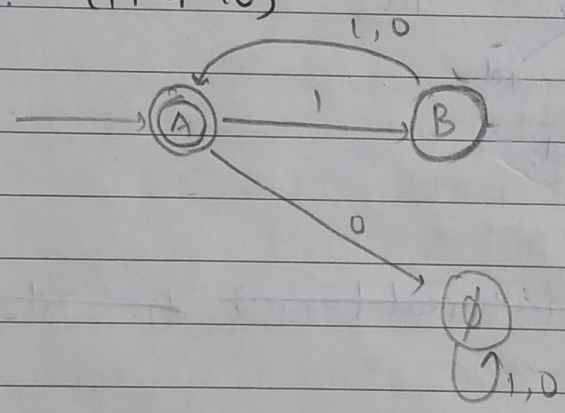
Ch-3 Regular Expression (self) [upto minimization of DFA]

Practice:

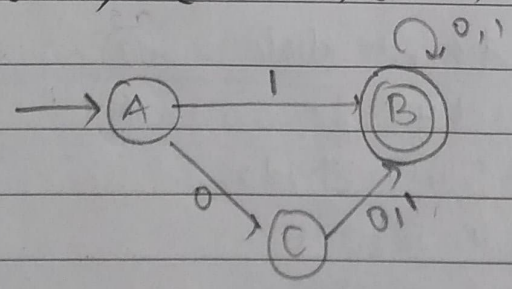
Q.1) $(0+1)^*0$



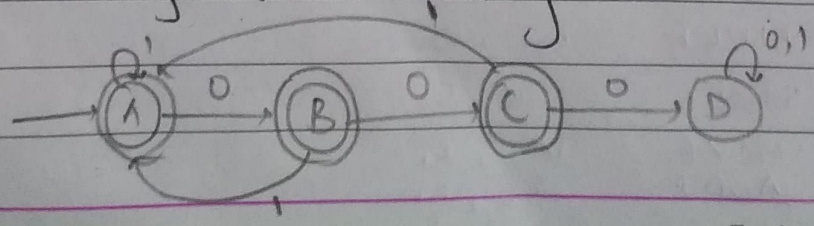
Q.2. $(11+10)^*$



Q.3. $(0+1)^*(1+00)(0+1)^*$



Q4 string not containing 000



Page No.
Date: / /

Q. $E \rightarrow E + E \mid E - E \mid E * E \mid id$

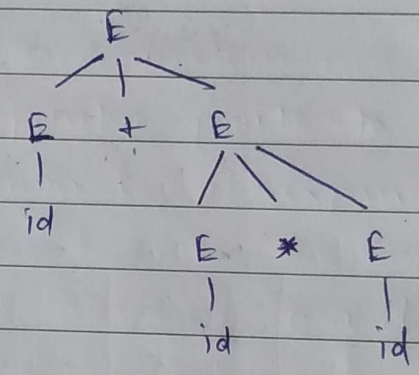
$G = (E, (+, -, *, id), P, E)$

* <u>Parsing</u>
  - ↳ Top down parsing (expansion)
  - ↳ Bottom up parsing (reduction)

① Top down: start with starting symbol and expand using production rules.

eg. id + id * id

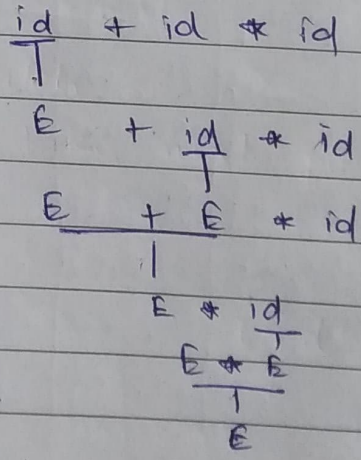
<u>problems</u>: - left recursion
- Ambiguity



② Bottom up: start with input string and get to start symbol.

eg. id + id * id

<u>problems</u>: - Ambiguity



<u>solution</u>: Remove Ambiguity

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

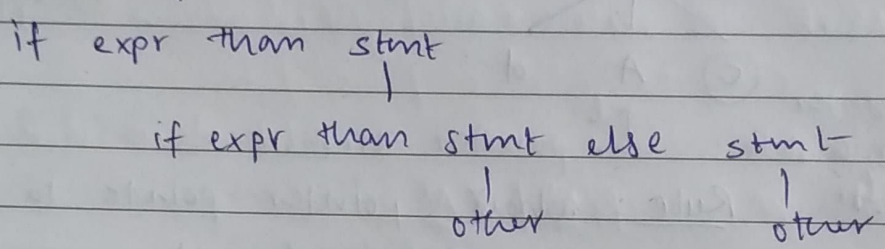
Teacher's Sign

$S \rightarrow abcS \mid abcSdS \mid x$

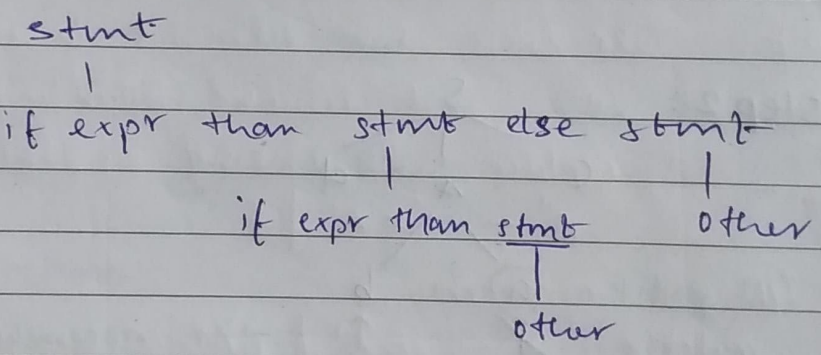
① . stmt → if expr than stmt |
if expr than stmt else stmt |
other.

eg. if expr than ~~[illegible]~~ if expr than other
else other

①



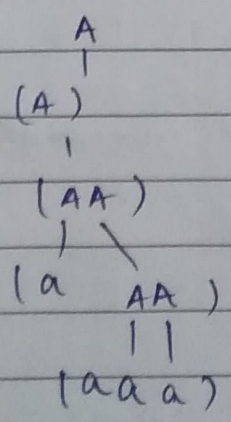
②



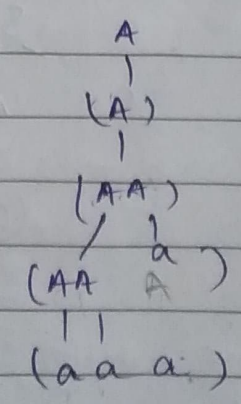
② $A \rightarrow AA \mid (A) \mid a$

eg ~~(AAA)~~ (aaa)

①



②



Top down parsing: RDP [Recurrsive Descent Parsing]

↳ It will take first production at start

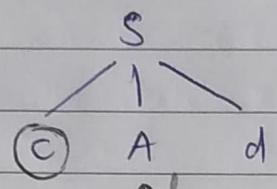
* $S \to cAd$
  $A \to ab/a$

grammar

I/P string → [RDP] → Parse tree

<u>I/P string</u>: cad

step 1:



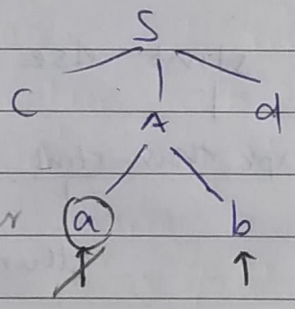
c a d
↑ → pointer i

I/p pointer matches pointer i.

↑ → pointer increments

<u>Rule</u>: when pointer points to non-terminal, it expands

step 2:



c a d
↑

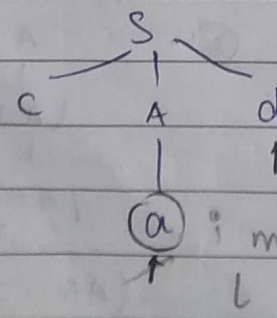
I/P pointer matches

pointer increment

c a d
↑

Pointer does not match.

Sol^n: Backtrack to take next production rule



c a d
↑

i goes to previous position before expansion

pointer increment

i matches i

cad
↑

last pointer matches, string is accepted.

Teacher's Sign

* $E \rightarrow iE'$ 4 - functions
$E' \rightarrow +iE' / \epsilon \rightarrow null$

① E(){
{ if (l=='i'){
match('i');
E'();
}
}

② E'(){
if (l=='+'){
match('+');
match('i');
E'();
}
else
return;
}

③ match(char t){
if (l==t){
l= getchar();
else
printf("Error");
}

④ main(){
E();
if(l==$)
printf("success");
}

left recursion problem

* $E \rightarrow Ea$

E() {
E() {
E() {
⋮

Problem occurs when start symbol matches non-terminal at first position.

Sol^n: Make it right recursion

* $A \rightarrow \boxed{A\alpha} / \beta / \gamma$ or $A \rightarrow \beta / A\alpha / \gamma$ [same]

combination of terminal & non-terminal production.

Step 1: Mark the rule causing recursion

Step 2: Write the rest & Add new non-terminal

$A \rightarrow \beta A' / \gamma A'$

Step 3: Take new non-terminal to be the problematic rule and remove starting symbol & replace with same symbol

$A' \rightarrow \alpha A' / \epsilon$

Final grammar.

$A \rightarrow \beta A' / \gamma A'$

$A' \rightarrow \alpha A' / \epsilon$

eg. 1. $E \rightarrow \boxed{E + T} / T$

$T \rightarrow T * F / F$

$F \rightarrow id$

Direct Left Recursion

① $E \rightarrow TE'$

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

$T \rightarrow \boxed{T * F} / F$

(2) $T \rightarrow FT'$

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

(3) $F \rightarrow id$

Final grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow id$

eg. 2. $A \rightarrow BC / a$ Indirect Left Recursion

$B \rightarrow CA / Ab$

$C \rightarrow AB / CC / a$

Step 1: Check for direct recursion in first production. If not then ~~check in second~~ move to second.

$A \rightarrow BC / a$ [If direct recursion present ~~then remove it~~]

Step 2: - Check if previous production exist in the rule as the first non-terminal.

$B \rightarrow \underline{C}A / Ab$ matches previous production

- substitute matched production

$B \rightarrow CA / \underline{BC}b / ab$ → Direct left Recursion

- Remove left recursion.

$B \rightarrow CAB' / abB'$

$B' \rightarrow CbB' / \epsilon$

Page No.
Date: / /

step 3: - Repeat step two for all the next productions.

① $C \rightarrow AB \mid CC \mid a$ [Direct recursion exists but remove after substitution]

② $C \rightarrow BCB \mid aB \mid CC \mid a$

③ $C \rightarrow CAB'CB \mid abB'CB \mid aB \mid CC \mid a$

- Remove Recursion

$C \rightarrow abB'CBC' \mid aBC' \mid aC'$

$C' \rightarrow AB'CBC' \mid CC' \mid \epsilon$

Final grammar:

$A \rightarrow BC \mid a$

$B \rightarrow CAB' \mid abB'$

$B' \rightarrow CbB' \mid \epsilon$

$C \rightarrow abB'CBC' \mid aBC' \mid aC'$

$C' \rightarrow AB'CBC' \mid CC' \mid \epsilon$

* Left Factoring

- It can be done on production having same prefix in its rules.

- Saves time of parser. (parser scans prefix only once)

eg. ① $A \rightarrow \alpha B_1 \mid \alpha B_2 \mid \alpha B_3 \mid \ldots \mid \alpha B_n$

Sol^n $A \rightarrow \alpha A'$

$A' \rightarrow B_1 \mid B_2 \mid B_3 \mid \ldots \mid B_n$

② $A \rightarrow aB \mid abc$

Sol^n $A \rightarrow aA'$

$A' \rightarrow B \mid bc$

③ $S \rightarrow iEtS \mid iEtSeS \mid a$

Sol^n $S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

Teacher's Sign

Predictive Parsing:

1. Eliminate Left recursion
2. Apply Left Factoring
3. Compute First & Follow.

- Two varients:
  - Recursive
  - Non-Recursive

* First & Follow

Grammar: $E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

First:

- First(E) = First(T)
  = {(, id}

  First(T) = First(F)
  = {(, id}

  First(F) = {(, id}

- First(E') = {+, ε}
- First(T') = {*, ε}

Rules:

1. ~~S → AB~~ $A \rightarrow ab / c / d / \epsilon$

   First(A) = {a, c, d, ε}

2. $S \rightarrow AB$
   $A \rightarrow a / \epsilon$
   $B \rightarrow b$

   it has null, so write

   First(S) = First(A) First(next)
   = {a, b}

   First(B)

   - no null as First(B) does not have null

3. $S \rightarrow AB$
   $A \rightarrow a / \epsilon$
   $B \rightarrow b / \epsilon$

   First(S) = {a, b, ε}

Follow:

- Follow(E) = { $,) }
  F → (E) / id
- Follow(T) = { + } ∪ { $,) }
  = { + , $,) }
- Follow(E') = { $,) } E → TE'
- Follow(F) = First(T') ∪ Follow(T)
  = { * , + , $,) }
- Follow(T') = Follow(T)
  = { + , $,) }

Rules: A → γ
1. B → αAβ A is start
   Follow(A) = First(β)
2. Follow(A) always has $ (start)
3. Null never comes in Follow
4. if β → i/ε, then replace β by ε.
   B → αA
   ∴ Follow(A) = First(β) ∪ Follow(B)

Table:

- Table of First & Table of Follow

Parsing Table

Non-terminals

Terminals	id	*	+	(	)	$
E	E → TE'			E → TE'		
E'			E' → +TE'		E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → *FT'	T' → ε		T' → ε	T' → ε
F	F → id			F → (E)		

First(E') & First(T') has ε, thus mark Follow(E') & Follow(T') too

eg.

$S \rightarrow ACB / CbB / Ba$

$A \rightarrow da / BC$

$B \rightarrow g / \epsilon$

$C \rightarrow h / \epsilon$

	First	Follow
A:	$\{d, g, h, \epsilon\}$	$\{h, g, \$\}$
B:	$\{g, \epsilon\}$	$\{\$, a, h, g\}$
C:	$\{h, \epsilon\}$	$\{g, \$, b, h\}$
S:	$\{d, g, h, \epsilon, a, b\}$	$\{\$\}$

Stack	Input	Production
$ E	id + id * id $	$E \to TE'$
$ E'T	id + id * id $	$T \to FT'$
$ E'T'F	id + id * id $	$F \to id$
$ E'T'id	id + id * id $	
$ E'T'	+ id * id $	$T' \to \epsilon$
$ E'	+ id * id $	$E' \to +TE'$
$ E'T +	+ id * id $	
$ E'T	id * id $	$T \to FT'$
$ E'T'F	id * id $	$F \to id$
$ E'T'id	id * id $	
$ E'T'	* id $	$T' \to *FT'$
$ E'T'F*	* id $	
$ E'T'F	id $	$F \to id$
$ E'T'id	id $	
$ E'T'	$	$T' \to \epsilon$
$ E'	$	$E' \to \epsilon$
$	$	

Teacher's Sign

$A \to \alpha$	First(α)	Follow(A)
$S \to iEtSS'$	i	e $
$S \to a$	a	e $
$S' \to eS$	e	e $
$S' \to \epsilon$	ϵ	e $
$E \to b$	b	t

Parsing table

	i	a	e	b	t	$
S	$S \to iEtSS'$	$S \to a$				
S'			$S' \to eS$			
			$S' \to \epsilon$			$S' \to \epsilon$
E				$E \to b$		

$A \to \alpha$	First(α)	Follow(A)
$S \to aBDh$	a	$
$B \to cC$	c	g, h, f
$C \to bC$	b	g, h, f
$C \to \epsilon$	ϵ	
$D \to EF$	g, f, ϵ	h
$E \to g$	g	f, h
$E \to \epsilon$	ϵ	
$F \to f$	f	h
$F \to \epsilon$	ϵ	

Parsing table

	a	b	c	f	g	h	$
S	S→aBDh						
B			B→cC				
C		C→bC		C→ϵ	C→ϵ	C→ϵ	
D				D→EF	D→EF	D→EF	
E				E→ϵ	E→g	E→ϵ	
F				F→f		F→ϵ	

* <u>Error Recovery</u> (self)

Bottom up Parsing: Shift Reduce Parsing
↳ It takes input string & Reduce it using production rules.

$S \to aABb$
$A \to aA \mid a$
$B \to bB \mid b$

input string: aaabb

$aaAbb$

$aAbb$

$aABb$

S

Find: **Handle** - substring that matches right side of production rule.
(not all substring that match is handle)

eg $E \to E + T \mid T$
$T \to T * F \mid F$
$F \to (E) \mid id$ No need to remove recursion

input: id + id * id

Right most Sentinal Form	Reducing Production
id + id * id	$F \to id$
F + id * id	$T \to F$
T + id * id	$E \to T$
E + id * id	$F \to id$
E + F * id	$T \to F$
E + T * id	$F \to id$
E + T * F	$T \to T * F$
E + T	$E \to E + T$
E	

4- Actions:

- ↳ shift → shift next symbol to top of stack
- ↳ Reduce → Replace handle with non-terminal
- ↳ Accept
- ↳ Error

Stack	Input	Action
$	id + id * id $	shift
$ id	+ id * id $	Reduce (F → id)
$ F	+ id * id $	Reduce (T → F)
$ T	+ id * id $	Reduce (E → T)
$ E	+ id * id $	Shift
$ E +	id * id $	shift
$ E + id	* id $	Reduce (F → id)
$ E + F	* id $	Reduce (T → F)
$ E + T → shift - Reduce conflict	* id $	Both options possible
	compiler doesn't know	what to choose
$ E + T	* id $	shift
$ E + T *	id $	shift
$ E + T * id	$	Reduce (F → id)
$ E + T * F → Reduce - Reduce conflict	$	T → F or T → T * F
		Two Reduce option
$ E + T * F	$	Reduce (T → T * F)
$ E + T	$	Reduce (E → E + T)
$ E	$	Accept

Grammar: L R (k)

- L: Left to right scanning
- R: Right most derivation
- k: look head

① Operator precedance parsing
↳ only operator Grammar

Rule : - Two non-terminal symbol should not be together

~~eg.~~ - No null

eg. $E \to AB$ | $E \to EOE$ | $E \to E+E$
$A \to a$ | $E \to id$ | $E \to E/E$
$B \to b$ | $O \to + | * | -$ | $E \to id$
✗ | ✗ | ✓

<u>Step 1</u> : Check grammar to be operator grammar

<u>Step 2</u> : Create Precedance Relation matrix.

eg. $E \to E+E / E*E / id$

①

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	

> higher precedance
< lower precedance
= equal precedance

+ & * is left associative hence

+ > +

& * > *

input: id + id * id

$ < id > + < id > * < id > $

↳ precedance

Finding Handle :

1. Scan string from left to right until first > is encountered
2. Then scan backwards (right to left) over any = until < is encountered
3. Everything in betn < & > is handle.

$ < id > + < id > * < id > $	E → id	$ E + id * id $
$ < + < id > * < id > $	E → id	$ E + E * id $
$ < + < * < id > $	E → id	$ E + E * E $
$ < + < * > $	E → E * E	$ E + E $
$ < + > $	E → E + E	$ E $

Step 3 : Parsing.

→ Actions:

1. < ⇒ shift to stack
2. > ⇒ Reduce

eg. $ < id ⇒ shift id to stack

id > + ⇒ Reduce id in stack with E → id

$ < id > + < id > * < id > $

Stack	Input	Action
$	id + id * id $	(<) Shift
$ id	+ id * id $	(>) Reduce E → id
$	+ id * id $	(<) Shift
$ +	id * id $	(<) shift
$ + id	* id $	(>) Reduce E → id
$ +	* id $	(<) Shift
$ + *	id $	(<) shift
$ + * id	$	(>) Reduce E → id
$ + *	$	(>) Reduce E → E * E
$ +	$	(>) Reduce E → E + E
$	$	Accept

For non-~~term~~ operator terminals:

$U \rightarrow \alpha \underbrace{s_1 s_2}_{\text{terminals}} \beta$ ✓ operator grammar

Precedance:

- $s_1 = s_2$ if $U \Rightarrow \alpha s_1 s_2 \beta$
  or $U \Rightarrow \alpha s_1 X s_2 \beta$

- $s_1 < s_2$ if $U \Rightarrow \alpha s_1 X \beta$ &
  $X \Rightarrow s_2 \ldots \mid Y s_2 \ldots$

- $s_1 > s_2$ if $U \rightarrow \alpha X s_1 \beta$ &
  $X \rightarrow s_2 \ldots \mid Y s_2 \ldots$

eg.

	id	(	)	$
id	e3	e3	>	>
(	<	<	=	e4
)	e3	e3	>	>
$	<	<	e2	e1

error (solution)

e3: missing operator (insert +/*/-/)

e4: missing right parenthesis (pop '(')

e2: unbalanced right parenthesis (delete ')')

e1: missing operand (insert id)

Precedence Function:

Method:

1. Create f_a & g_b for $\forall a \in$ terminal
2. Partition ~~gro~~ symbol into groups such that if $a=b$, f_a & g_b are in same group
3. Create directed graph, where if $a<b$, place edge from g_b to f_a & $a>b$, place edge from f_a to g_b
4. If graph has cycle, no precedence fn exist.

SLR Parsing:

Grammar:

$E \rightarrow E + T \;|\; T$ (1, 2)

$T \rightarrow T * F \;|\; F$ (3, 4)

$F \rightarrow \cancel{(id)} \;|\; (E) \;|\; id$ (5, 6)

pointer

* $E' \rightarrow .E$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

I_0

goto (I_0, E)

$E' \rightarrow E.$

$E \rightarrow E. + T$

I_1

goto (I_0, T)

$E \rightarrow T.$

$T \rightarrow T. * F$

I_2

goto (I_0, F)

$T \rightarrow F.$

I_3

goto $(I_0, ($

$F \rightarrow (.E)$

$E \rightarrow .E + T$

$E \rightarrow .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

I_4

goto (I_0, id)

$F \to id.$	I_5

goto (I_1, +)

$E \to E + .T$	
$T \to .T * F$	
$T \to .F$	I_6
$F \to .(E)$	
$F \to .id$	

goto (I_2, *)

$T \to T * .F$	
$F \to .(E)$	I_7
$F \to .id$	

goto (I_4, E)

$F \to (E.)$	I_8
$E \to E. + T$	

goto (I_4, T) $\to I_2$
goto (I_4, F) $\to I_3$
goto (I_4, ( ) $\to I_4$
goto (I_4, id) $\to I_5$

goto (I_6, T)

$E \to E + T.$	I_9
$T \to T. * F$	

goto (I_6, F) $\to I_3$
goto (I_6, ( ) $\to I_4$
goto (I_6, id) $\to I_5$

Teacher's Sign

goto (I_7, f)

$T \to T * f.$ | I_{10}

goto $(I_7, ()) \to I_4$

goto $(I_7, id) \to I_5$

goto $(I_8,))$

$F \to (E).$ | I_{11}

goto $(I_8, +) \to I_6$

~~goto $(I_8, +$~~ goto $(I_9, *) \to I_7$

	id	+	*	(	)	$	E	T	F
I_0	s5			s4			1	2	3
I_1		s6				acc			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7			r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Note: to calculate reduce, calculate follow.

state number — reverse default

Stack	input	Action
$ 0	id * id + id $	s5 , shift 5
$ 0 id 5	* id + id $	r6 , goto 3

pop 1 * 2 = 2 — 6 — F → id — $(I_0, F) = I_3$

symbol — original — symbol — symbol before F

$(I_0, F) = I_3$

$ 0 F 3	* id + id $	r4 - goto 2
$ 0 T 2	* id + id $	s7
$ 0 T 2 * 7	id + id $	s5
$ 0 T 2 * 7 id 5	+ id $	r6 goto 10 (7 F)
(pop 3×2=6) $ 0 T 2 * 7 F 10	+ id $	r3 goto 2
$ 0 T 2	+ id $	r2 goto 1
$ 0 E 1	+ id $	s6
$ 0 E 1 + 6	id $	s5
$ 0 E 1 + 6 id 5	$	r6 goto 3
$ 0 E 1 + 6 F 3	$	r4 goto 9
$ 0 E 1 + 6 T 9	$	r1 goto 1
$ 0 E 1	$	accept

* eg

$$S \to CC^1$$
$$C \to cC^2 / d^3$$

$S' \to .S$	I_0
$S \to .CC$	
$C \to .cC$	
$C \to .d$	

acc — goto (I_0, S)

$S' \to S.$	I_1

goto (I_0 , C)

$S \rightarrow C.C$

$C \rightarrow \cancel{c}.cC$

$C \rightarrow .d$

| I_2

goto (I_0 , c)

$C \rightarrow c.C$

$C \rightarrow .cC$

$C \rightarrow .d$

| I_3

production number

③ – goto (I_0 , d)

$C \rightarrow d.$

| I_4

① – goto (I_2 , C)

$S \rightarrow CC.$

| I_5

goto (I_2 , c) $\rightarrow$ I_3

goto (I_2 , d) $\rightarrow$ I_4

~~goto (I_5 , C)~~ ~~HO~~

② – goto (I_3 , C)

$C \rightarrow cC.$

| I_6

goto (I_3 , c) $\rightarrow$ I_3

goto (I_3 , d) $\rightarrow$ I_4

③ – follow(C) = { c , d , \$ }

① – follow(S) = { \$ }

② – follow(C) = { c , d , \$ }

	c	d	$	S	C
I_0	s3	s4		1	2
I_1			acc		
I_2	s3	s4			5
I_3	s3	s4			6
I_4	r3	r3	r3		
I_5			r_1		
I_6	r_2	r_2	r_2		

* eg

$S \to C$ 1

$C \to AB$ 2

$A \to a$ 3

$B \to a$ 4

$S' \to .S$

$S \to .C$

$C \to .AB$

$A \to .a$

I_0

acc - goto(I_0, S)

$S' \to S.$ | I_1

① - goto(I_0, C)

$S \to C.$ | I_2

goto(I_0, A)

$C \to A.B$

$B \to .a$ | I_3

③ - goto(I_0, a)

$A \to a.$ | I_4

② - goto(I_3, B)

$C \to AB.$ | I_5

④ - goto(I_3, a)

$B \to a.$ | I_6

Follow(S) = {$}
Follow(C) = {$}
Follow(A) = {a}
Follow(B) = {$}

	a	$	S	C	A	B
I_0	S4		1	2	3	
I_1		acc				
I_2		r1				
I_3	S6					5
I_4	r3					
I_5		r2				
I_6		r4				

* eg

$S \to L=R$ 1
$S \to R$ 2
$L \to *R$ 3
$L \to id$ 4
$R \to L$ 5

$S' \to .S$
$S \to .L=R$
$S \to .R$
$L \to .*R$
$L \to .id$
$R \to .L$

| I_0

(ace) - goto (I_0, S)

$S' \rightarrow S.$ | I_1

goto (I_0, L)

$S \rightarrow L. = R$
$R \rightarrow L.$ | I_2

(2) - goto (I_0, R)

$S \rightarrow R.$ | I_3

goto (I_0, *)

$L \rightarrow *.R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$ | I_4

(4) - goto (I_0, id)

$L \rightarrow id.$ | I_5

goto (I_2, =)

$S \rightarrow L = .R$
$R \rightarrow .L$
$L \rightarrow .*R$
$L \rightarrow .id$ | I_6

(3) - goto (I_4, R)

$L \rightarrow *R.$ | I_7

(5) - goto (I_4, L)

$R \rightarrow L.$ | I_8

Teacher's Sign

goto(I_4, *) $\longrightarrow$ I_4

goto(I_4, id) $\longrightarrow$ I_5

① - goto(I_6, R)

$S \longrightarrow L = R.$ | I_9

goto(I_6, L) $\longrightarrow$ I_8

goto(I_6, *) $\longrightarrow$ I_4

goto(I_6, id) $\longrightarrow$ I_5

	id	*	=	$	S	L	R
I_0	S5	S4			1	2	3
I_1							
I_2			S6				
I_3							
I_4	S5	S4				8	7
I_5							
I_6	S5	S4				8	9
I_7							
I_8							
I_9							

* CLR

Look ahead:

$S' \longrightarrow .S \quad , \quad \$$ — parent

$S \longrightarrow .L = R \quad , \$$ } child

$S \longrightarrow .R \quad , \$$

Rule: parent (L.A) = child (L.A)

parent = $S \longrightarrow .L = R$ — L.A = '='

$L \longrightarrow .*R \quad , \quad =$

$L \longrightarrow .id \quad , \quad =$

parent = $S \longrightarrow .R$ — L.A = \$

$R \longrightarrow .L \quad , \$$

$L \longrightarrow .*R \quad , \quad \$$

$L \longrightarrow .id \quad , \quad \$$

finally,

I_0 ,

$S' \longrightarrow .S \quad , \$$	$L \longrightarrow .*R \quad , \; = / \$$
$S \longrightarrow .L = R \quad , \$$	$L \longrightarrow .id \quad , \; = / \$$
$S \longrightarrow .R \quad , \$$	$R \longrightarrow .L \quad , \$$

Note:

Rule – $A \longrightarrow \alpha . B \beta \quad , \quad a \rightarrow L.A$

$B(L.A) = First(\beta a)$

Date: / /

C.C.

eg. $S \rightarrow CC^1$
$C \rightarrow cC^2 / d^3$ CLR

		L.A.
I_0	$S' \rightarrow .S$	$
	$S \rightarrow .CC$	$
	$C \rightarrow .cC$	c/d
	$C \rightarrow .d$	c/d

goto (I_0, S)
$S' \rightarrow S.$, $ | I_1

goto (I_0, C)

$S \rightarrow C.C$	$	I_2
$C \rightarrow .cC$	$	
$C \rightarrow .d$	$	

goto (I_0, c)

$C \rightarrow c.C$	c/d	I_3
$C \rightarrow .cC$	c/d	
$C \rightarrow .d$	c/d	

- goto (I_0, d)
$C \rightarrow d.$, c/d | I_4

- goto (I_2, C)
$S \rightarrow CC.$, $ | I_5

goto (I_2, c)

$C \rightarrow c.C$	$	I_6
$C \rightarrow .cC$	$	
$C \rightarrow .d$	$	

Date: / /

- goto(I_2, d)
  $C \rightarrow d.$, $ | I_7

- goto(I_3, C)
  $C \rightarrow cC.$, c/d | I_8

  goto(I_3, c) $\longrightarrow$ I_3
  goto(I_3, d) $\longrightarrow$ I_4

- goto(I_6, C)
  $C \rightarrow cC.$, $ | I_9

  goto(I_6, c) $\longrightarrow$ I_6
  goto(I_6, d) $\longrightarrow$ I_7

	c	d	$	S	C
I_0	s3	s4		1	2
I_1			acc		
I_2	s6	s7			5
I_3	s3	s4			8
I_4	r3	r3			
I_5			r1		
I_6	s6	s7			9
I_7			r3		
I_8	r2	r2			
I_9			r2		

Reduce:
r1
production

Lookahead at I_4

	c	d	$
I_4	r3	r3	

LALR

I_0	S3	S4		1	2
I_1			acc		
I_2	S6	S7			5
I_{36}	S36	S47			89
I_{47}	r3	r3	r3		
I_5			r1		
I_{89}	r2	r2	r2		

LALR:
LookAhead LR.

Minimize CLR.

Date: / /

Ch-5, Semantic Analysis

Input : Parse tree

Syntax Directed Translation

Semantic rules (Translation rules) used with production rules

Grammar rule + Semantic rule = SDT

eg.

$S \longrightarrow S \# A / A$	$S.val = S.val * A.val$
$A \longrightarrow A \& B / B$	$A.val = A.val + B.val$
$B \longrightarrow id$	$B.val = \underline{l.val}$

↳ attribute

production rules (Syntax Grammar)

Semantic rules (Attribute Grammar)
syntax directed definition

eg.

Production	~~Sym~~ Semantic Rule
$L \longrightarrow E\,n$	print (E.val)
$E \longrightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \longrightarrow T$	$E.val = T.val$
$T \longrightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \longrightarrow F$	$T.val = F.val$
$F \longrightarrow (E)$	$F.val = E.val$
$F \longrightarrow digit$	$F.val = digit.lexval$

Deapth First Search is used on parse tree to allocate values.