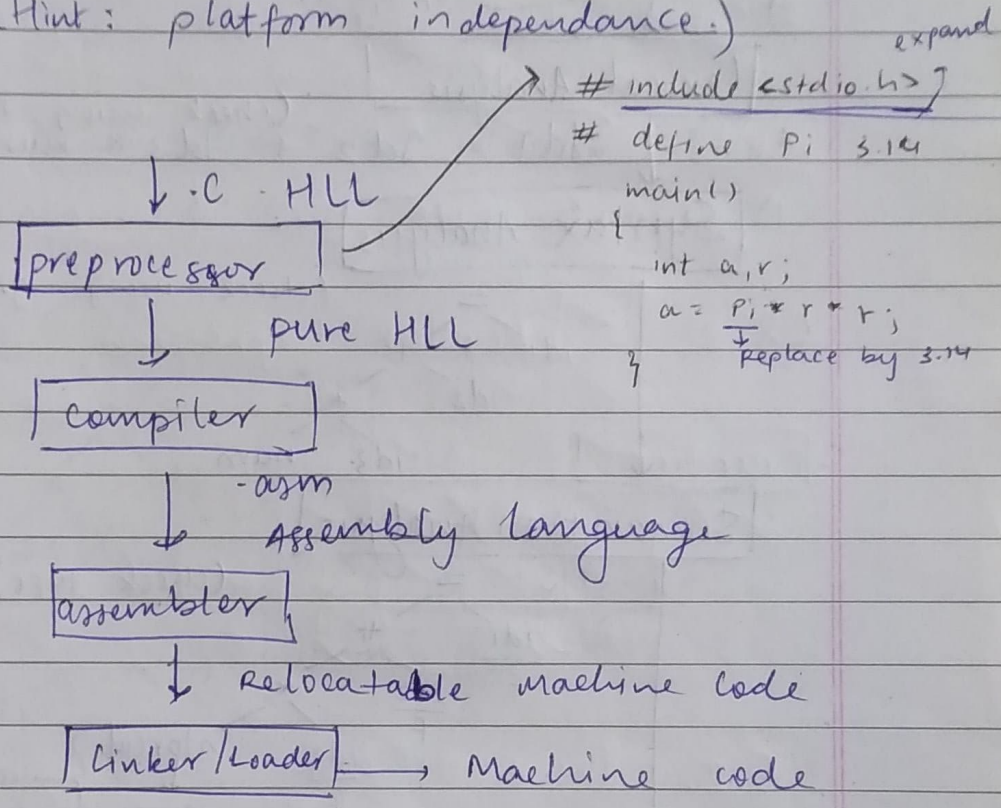
Page No.
Date: / /

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



→ 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

Teacher's Sign _____

Identifier

a = b + c * 6

operator

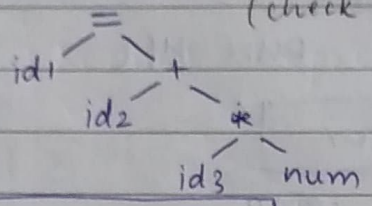
↓

Lexical Analysis (check using regular expression)

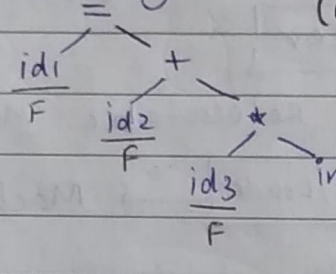
id1 = id2 + id3 * num (draws DFA)

↓

Syntax Analysis (check using grammar) (draws tree)



Semantic Analysis (check type)



— convert int to float

Intermediate Code Generator (only one variable at a time)

t_1 = intoreal (num)

t_2 = ~~id~~ t_1 * id3

t_3 = id_2 + t_2

id_1 = t_3

code optimizer

t_1 = id3 * intoreal (6)

id_1 = id_2 + t_1

Code generation (code using registers)

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

Teacher's Sign _____

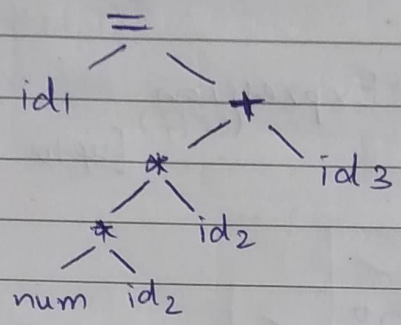
* Ex:

$a = \pi r^2 + v$

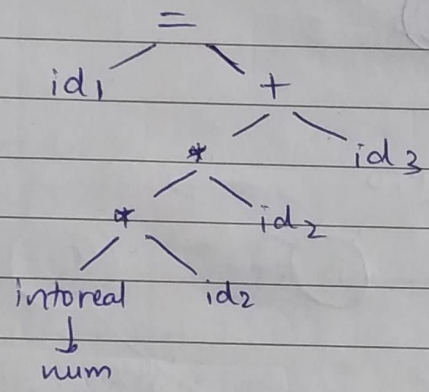
$a = 3.14\ r^2 + v$

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
```

Teacher's Sign _____

Page No.
Date: / /

Single pass:

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

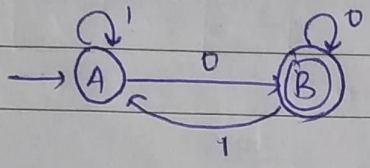
Multi pass:

1) Require more time (complex)

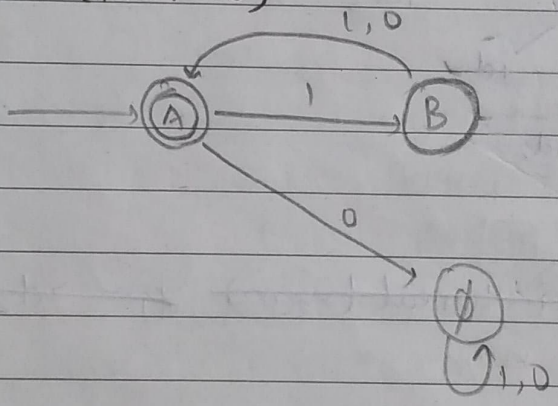
Ch-3 Regular Expression (self) [upto minimization of DFA]

Practice:

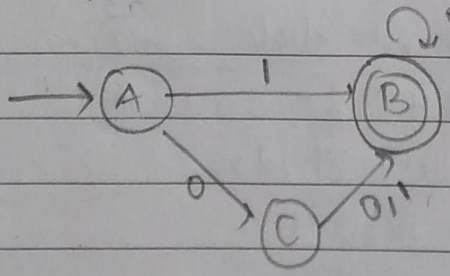
Q.1) $(0+1)^*0$



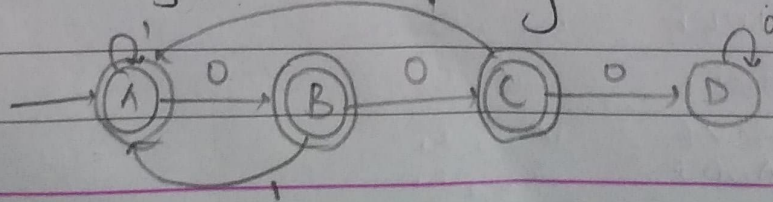
Q.2. $(11+10)^*$



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



Q4 string not containing 000



Page No.
Date: / /

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

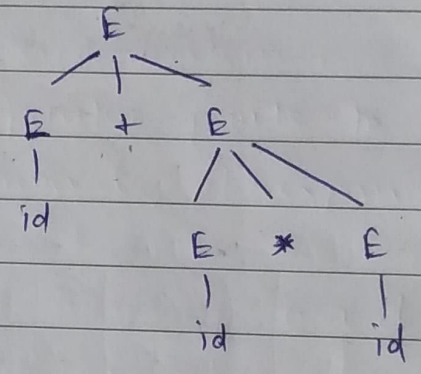
$G = (E, (+, -, *, id), P, E)$

* **Parsing**
  - ↳ Top down parsing (expansion)
  - ↳ Bottom up parsing (reduction)

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

eg. id + id * id

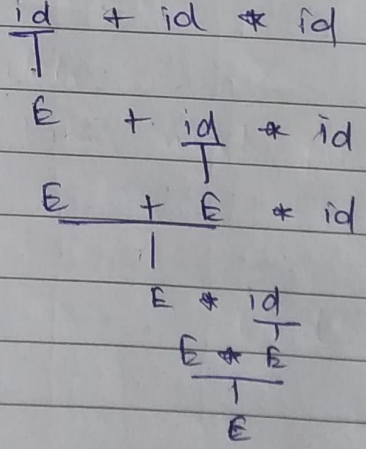
problems: -left recursion
-Ambiguity



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

eg. id + id * id

problems: -Ambiguity



solution: Remove Ambiguity

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

Teacher's Sign

S → abcS | abcSdS | x

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

eg. if expr than if expr than other
else other

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

② stmt
|
if expr than stmt else stmt
| |
if expr than stmt other
|
other

② A → AA | (A) | a

eg (aaa)

① A
|
(A)
|
(AA)
/ \
(a AA)
| |
(aaa)

② A
|
(A)
|
(AA)
/ \
(AA a)
| |
(aaa)

Teacher's Sign

Page No.
Date: / /

Top down parsing: RDP [Recursive Descent Parsing]

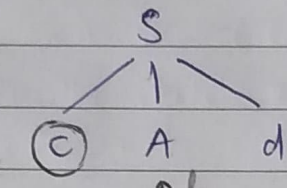
(It will take first production at start)

* S → cAd
  A → ab/a

grammar → I/P string → [RDP] → Parse tree

I/P string: cad

Step 1:



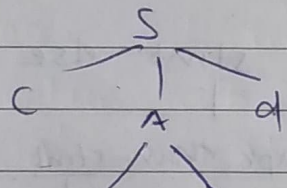
c a d
↑ → pointer i

I/p pointer matches pointer i.

↑ → pointer increments

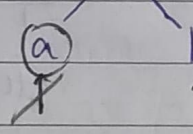
Rule: when pointer points to non-terminal, it expands

Step 2:



c a d
↑

I/P pointer matches

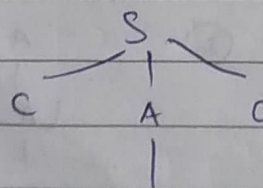


c a d
↑

↑ pointer increment

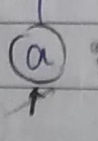
Pointer does not match.

Soln: Backtrack to take next production rule



c a d
↑ i goes to previous position before expansion

↑ pointer increment



i matches l

c a d
↑

last pointer matches, string is accepted.

Teacher's Sign

Page No.
Date: / /

* $E \rightarrow iE'$ 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(t==t){
l=getchar();
else
printf("Error");
}

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

Teacher's Sign

left recursion problem

* $E \rightarrow Ea$

```
E() {
   E() {
      E() {
         :
```

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

Soln: 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. $E \rightarrow \boxed{E + T} / T$ Direct Left-Recursion

$T \rightarrow T * F / F$

$F \rightarrow id$

① $E \rightarrow TE'$

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

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

Teacher's Sign

(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.

$B \rightarrow \underline{C}A / (A)b$

matches previous production

- substitute matched production

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

- Remove left recursion.

$B \rightarrow CAB' / abB'$

$B' \rightarrow CbB' / \epsilon$

Teacher's Sign _____

Page No.
Date: / /

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

① C → AB / CC / a [Direct recursion exists but remove after substitution]
② C → BCB / aB / CC / a
③ C → CAB'CB / abB'CB / aB / CC / a

- Remove Recursion
C → abB'CBE' / aBC' / aC'
C' → AB'CBC' / CC' / ε

Final grammar:
A → BC / a
B → CAB' / abB'
B' → CbB' / ε
C → abB'CBC' / aBC' / aC'
C' → AB'CBC' / CC' / ε

* Left Factoring
- It can be done on a production having same prefix in its rules
- Saves time of parser. (parser scans prefix only once)

eg. ① A → αB1 / αB2 / αB3 / ..... / αBn
Sol^n A → αA'
A' → B1 / B2 / B3 / ..... / Bn

② A → aB / abc
Sol^n A → aA'
A' → B / bc

③ S → iEts / iEtseS / a
Sol^n S → iEtSS' / a
S' → eS / ε

Teacher's Sign

Predictive Parsing:

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

- Two variants:
  - 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 → ab / c / d / ε

   First(A) = {a, c, d, ε}

2. S → AB
   A → a / ε
   B → b

   First(S) = First(A) First(next)
   (it has null, so write)
   = {a, b}
   (b → First(B))

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

3. S → AB
   A → a / ε
   B → b / ε

   First(S) = {a, b, ε}

Teacher's Sign _____

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 (A is start)

1. B → αAβ
   Follow(A) = First(β)
2. Follow(A) (start) always has $
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

Teacher's Sign _____

eg: $S \rightarrow ACB / \epsilon bB / Ba$

$A \rightarrow da / BC$

$B \rightarrow g / \epsilon$

$C \rightarrow h / \epsilon$

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