Date: / /20
Page No.:

Q- $E \to aA \mid aB$
$A \to \epsilon$
$B \to b$

∴ {don't apply left factoring}.

string = 'ab'

| | First | follow |
|---|---|---|
| E | {a} | {$} |
| A | {ε} | {$} |
| B | {b} | {$} |

| | a | ε | b | $ |
|---|---|---|---|---|
| E | $E \to aA$, $E \to aB$ | | | |
| A | | $A \to \epsilon$ | | |
| B | | | $B \to b$ | |

problem

| stack | input | action |
|---|---|---|
| $E | ab | - |

↳ here, as we have multiple entries, we call this ~~LL(1)~~ is not LL(1) grammar.

Date : / /20
Page No.:

⇒ **LL(1) grammar** - Grammar for which a predictive parsing table has no multiple entries.

For top down parsing ⇒ LL(1) → look ahead symbol.
↳ $E  ab$

- L → left to right scanning
- L → leftmost deriv^n. (for top down) parsing
- (1) → parser only looks 1 symbol ahead.

To check is grammar is LL(1):-

way-1) ☆ : only when parser sees multiple entries while parsing ~~to ab$~~ it comes to know grammar is not LL(1).
putting 2 product^n in table

- This way is not efficient., so parser doesn't use it.

So, we saw (1st way) make table & check.

way-2) 2nd way - condition checking

If any of the 3 conditions fails, then grammar is not LL(1)

① If grammar with product^n of type

$A \to \alpha | \beta$

$\boxed{first(\alpha) \cap first(\beta) = \phi}$

~~E → Eb' / E → Ab'~~

Date : / /20
Page No.:

eg - $E \to \underbrace{E+b}_{\alpha} \mid \underbrace{b}_{\beta}$

first(α) = first (E+b) = b ,
first(β) = first (b) = b

intersect°. union is not ∅

Hence grammar is not LL(1)

&

| | first | follow |
|---|---|---|
| eg⇒ E | {b} | {$, +} |

| | $ | b | + |
|---|---|---|---|
| E | | E → E+b<br>E → b | |

↳ problem

(2) If grammar is with product° of type.

$A \to \alpha \mid \beta$

$\beta \overset{*}{\Rightarrow} \epsilon$ (nullable β)

then first(α) cannot ~~start with~~ be this symbol that is present in follow(A).

eg - $E \to E+b \mid \epsilon$

first (E+b) = {+}

follow(E) ⇒ {$, +}

Date : / /20
Page No.:

| | + | b | ϵ | $ |
|---|---|---|---|---|
| E | E→E+E, E→ϵ | | | E→ϵ |

↳ multiple entries

(3) If the product is of the type $E \to \alpha | \beta$ then atmost 1 of $\alpha$ or $\beta$ can be nullable.

(no more than 1 can be nullable)

$E \to A | B$
$A \to a * b | \epsilon$
$B \to c + d | \epsilon$

| | First | Follow |
|---|---|---|
| E | {a, ϵ, c} | {$} |
| A | {a, ϵ} | {$} |
| B | {c, ϵ} | {$} |

don't write directly ϵ as E is not nullable directly

| | * | a | b | + | $ | c | d |
|---|---|---|---|---|---|---|---|
| E | | E→A | | | E→ϵ | E→B | |
| A | | A→a*b | | | A→ϵ | | |
| B | | | | | B→ϵ | B→c+d | |

** · so if grammar doesn't have left recurs, nondeter. & ambiguity, only then its LL(1) grammar.

Date : / /20
Page No.:

Q- $S \to AB \mid CDa$
$A \to ab \mid C$
$B \to dC$
$C \to eC \mid \epsilon$
$D \to fD \mid \epsilon$

$first(S) = first(A) = \{a\} \cup first(C) = \{a\} \cup \{e, \epsilon\} = \{a, e, \epsilon\}$
$= first(C) = \{e\}$

$\{e\} \cap \{a, e, \epsilon\} \neq \emptyset$
∴ Hence, not LL(1)

Q- $S \to Sa \mid Sbc \mid Sd \mid e \mid f \mid d$

$first(S) = first(Sa) = \{e, f, d\}$ → intersect $\neq \emptyset$
$= first(Sbc) = \{e, f, d\}$
Hence, not a LL(1) using rule (1).

⇒ $S \to Sa \mid e \mid f \mid d$ ⇒ $S \to SA \mid e \mid f \mid d$
$A \to a \mid bc \mid d$ $A \to a \mid bc \mid d$

⇒ $S \to eS' \mid fS' \mid dS'$ ⇒ $S \to eS' \mid fS' \mid dS'$
$S' \to AS' \mid \epsilon$ $S' \to AS' \mid \epsilon$
$A \to a \mid bc \mid d$ $A \to a \mid bc \mid d$

Date : / /20
Page No.:

| | First | follow |
|---|---|---|
| S | {e, f, d} | {$} |
| A | {a, b, d} | {$} |
| S' | {a, b, d, ∈} | {$} |

| | e | f | d | a | b | $ |
|---|---|---|---|---|---|---|
| S | S→eS' | S→fS' | S→dS' | | | |
| S' | | | S'→AS' | S'→AS' | S'→AS' | S'→∈ |
| A | | | A→d | A→a | A→bC | |

self:) eg:) fdbca

| stack | i/p | action |
|---|---|---|
| $S | fdbca$ | - |
| $S'f | fdbca$ | S→fS' |
| $S'A | dbca$ | S'→AS' |
| $S'd | dbca$ | A→d |
| $S'A | bca$ | S'→AS' |
| $S'Cb | bca$ | A→bC |
| $S'A | a$ | S'→AS' |
| $S'a | a$ | A→a |
| $S'∈ | $ | S'→∈ |
| $ | $ | |

- only lexical & syntax analyzer updates symbol table.
- Rest phases only read.

Date: / /20
Page No.:

class → **Error recovery strategies of Parser**

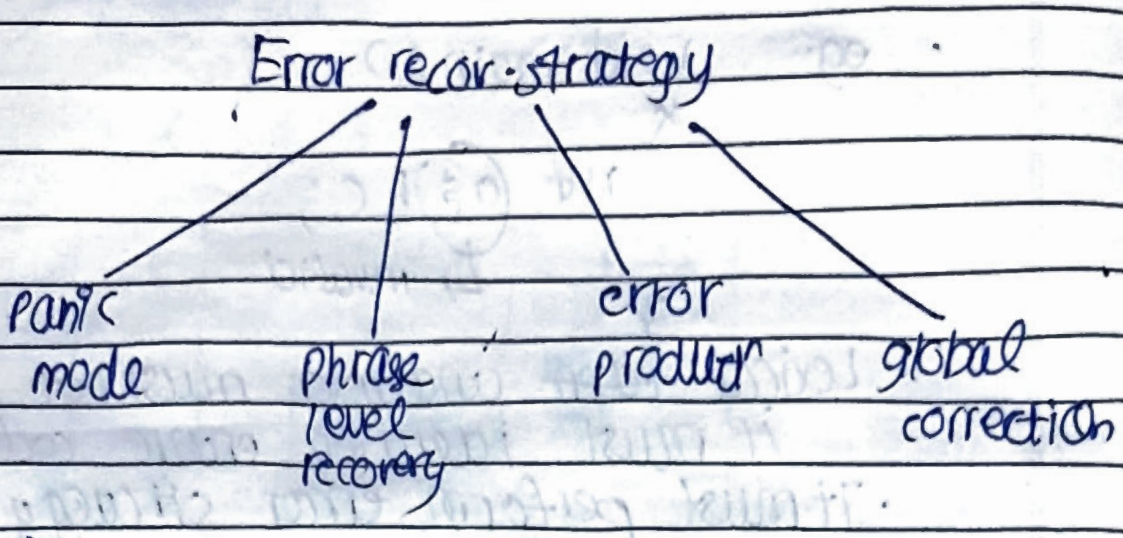
- eg- void main()
{
int a$ b, c, ;
↳ invalid
}

- Lexical ~~iden~~ analyzer must not stop here, it must identify entire code.
- It must perform error strategy to recover & jump to next stage.
- Error recovery is step taken by error handler to recover & proceed in the program- machine must not stop, & verify other parts.
- So, error handler is needed to help machine complete in 1 pass.

- Symbol table & error handler is connected to each phase.
- ★ error recovery is done by error handler. error generated by a phase (lexical, syntax

Date : / /20
Page No.:

→ Parser uses 4 error recovery strategies




1) panic mode

: when parser gets error
(eg - int a, b; (; missing)

↳ missing

. error handler skips all the symbols till synchronizing token identified.

↳ till ; or } identified

semicolon bracket

eg - switch ( )
{
case → wrongly spelled
(i) → not colon but semicolon so wrong

skip all symbols till

}

Date: / /20
Page No.:

## Advantage

- Simple
- never goes in ∞ loop
- suitable when more errors are there in single line

## Disadv.

- skips considerable amt. of i/p when checking for additional errors. → tomuchskip.
- may generate spurious error.

⇒ lexical analyzer only used panic mode

int (a) b, c;
skip

→ spurious error - error generated by program or machine (any phase) to recover from the existing error is known as spurious error.

+ predefined oper's
(insert, ~~update~~ ~~delete~~ ~~transpose~~, delete)

eg int a1, b, c;
insert a. as missing.
↓
now if
int a1, b, c a1;
already present
↓
so lexical analyzer gen. error itself (multiple errors)

Date : / /20
Page No.:

eg- int a, b, c;
↳ delete can be performed

eg- int (a), b, c;
↳ transpose → [a]

→ Phrase level Recovery

- Local correctn by parser on remaining i/p, by some string which allows parser to continue.

eg- int a, b; c; → undeclared variable c.
↑
local correctn, replace ; by ,
∴ int a, b, c;
↳ now it can parse ahead.

- Replacing , by ; or inserting extra semicolon etc.
- Local correctn can be any kind of syntax.

→

Date : / /20
Page No.:

→ Adv

- can correct any i/p string
- this method is used in many errors repairing compilers ↳ eg- cross compilers.

Disadv

- Replacement program should be prevented from ∞ loop.

→ Error production

eg- S → A
A → aA | b
B → cd

eg ⇒ abcd can't be generated by parser.

- So, parser will introduce augmented grammar → introducing new ~~gram~~ product

we can add new gen. error constructs

eg- E → SB
S → A
A → aA | b
B → cd

} now "abcd" can be parsed

- such parser will detect anticipated errors when an error production is used.

Date : / /20
Page No.:

* Grammar must not change -
eg- we can't say :) $S \to AB$ in grammar changes.

Adv
- error diagnostics are readily avail. for such anticip. errors.
- syntactic phase error are generally recov. by error producers.

Disadv
- Difficult to maintain method as grammar can change.
- Difficult to maintain by the developers.

→ This correction is local.

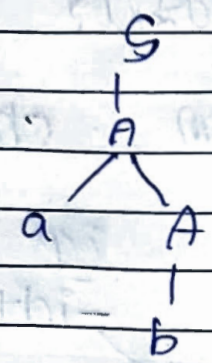
→ Global correction

- In above case to gen. 'abcd', we changed locally.
- Here, for any i/p string x algo finds a parse tree for a related y, such that no. of inserts, delets & token change req. for converting x to y is minimum.

Date : / /20
Page No.:

eg- $S \rightarrow A$
$A \rightarrow aA \mid b$
$B \rightarrow cd$.

here we can gen 'ab'.
'abcd' is matched with a string with a valid parse tree




- now we can delete 'cd' to get correct string.
- This is global correctn.

- ~~Ideally~~ This correctn is also locally applied.
- Ideally min. changes should be made to i/p string.

→ Adv.

- Makes few changes in processing an incorrect i/p string.

→ Disadv

- Too costly for time & space.

Date : / /20
Page No.:

Q -

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

| | first | follow |
|---|---|---|
| E | {(, id} | {), $} |
| E' | {+, ε} | {), $} |
| T | {(, id} | {+, ), $} |
| T' | {*, ε} | {+, ), $} |
| F | {(, id} | {*, +, ), $} |

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

string : ~~)id*id*id$~~ ⇒ )id id*id$

Date: / /20
Page No.:

| stack | i/p | act^n |
|---|---|---|
| $ E | ) id * id $ | error |
| $ E | * id * id $ | E → synch |
| $ E' T | id * id $ | E → TE' |
| $ E' T' F | id * id $ | T → FT' |
| $ E' T' id | id * id $ | F → id |
| $ E' T' | * id $ | T' |

→ LL(1) parsing and predictive parsing are same

⇒ (1) When TOS is terminal & string not match, pop it. - using insert oper^ of panic mode.

| | stack | i/p | act^n |
|---|---|---|---|
| eg ⇒ | $ S b | a b $ | insert b in i/p due to panic mode, & pop |
| ⇒ | ($ S b) | (a b $) insert | |
| | $ S | a b $ | act^n |
| | ↓ | ↓ | |
| | continue. | | |

→

Date : / /20
Page No.:

(2) ⇒ $$SbdA  cadb$  ⇒ error

pop this nonterm (as not starting).
↳ synch prod.
- so, pop A, & skip i/p till we get any follow symbol of A

∴ $Sbd  db$
c & a skipped as not follow.

→ Indirect left recurs^.

eg - S → Ab | a  ①
A → Ad | Sa | b  ②

cycle ⇒ S → A... & A → S... (by substituting)

- write pseudo code for it :-

1) Arrange in order (starting symbol)
2) Don't touch 1st product (S), & check if 2nd product (A)
∴ A → Ad substituting ① in ②,
S → Ab | a
A → Ad | Aba | aa | b

$A \to A\alpha_1 | A\alpha_2 | \dots | \beta_1 | \beta_2$

$\therefore A \to \beta_1 A' | \beta_2 A' | \dots$

$A' \to \alpha_1 A' | \alpha_2 A' | \epsilon$

Date : / /20

Page No.:

- algo. says that $A_j \to \gamma$ should be substituted if $j < i$
  (S is above A, hence, S is substituted in A, not A in S).

~~$A \to AA'$~~ $\therefore S \to Ab | a$

$A \to aaA' | bA'$

$A' \to dA' | baA' | \epsilon$

- so, we arranged product (in any order) & the above product substituted in below products.
- we substitute & solve left recurs^n.
  - ↳ we can do it any way in any order. but higher gets substituted below.

## Input buffering

int a, b;

↳ lexical analyzer.

<kw, int>

<id, ptr. to symbol table, a>

⋮

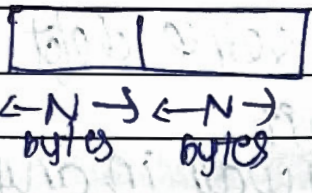
→

Date : / /20
Page No.:

- Lexical & syntax analyzer use memory for scanning i/p ⇒ that memory is **i/p buffering**

∴ Lexical analyzer stores i/p in i/p buffer.

- Lexical analyzer manages mem. inform of buffer
  - ↳ divided in 2 'N' size halves




- In (read-write oper^n, analyzer uses N bytes.

eg- 


start ptr. (first loc^n of N byte half)

forward ptr. (incremented by 1)

- If forw. ptr. reaches end of 1st N, then fill 2nd buff.

Date: / /20
Page No.:

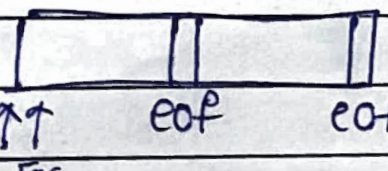
- So ⇒ If forward ptr. at end of first half, then begin reload 2nd half;
  forward = forward+1;
  end
  else if forward at end of 2nd half then
  begin reload first half.
  move forward to beginning of first half
  end
  else forward = forward+1;

⇒ For every move, we used 2 comparisons, so not efficient

- To solve this we use sentinels.

⇒ Sentinels

- Identify end of buffer. → represent end of buffer
- They are not part of keyword of language. ↳ reserved words
- let 'eof' be sentinel.




⇒

Date : / /20
Page No.:

⇒ forw = forw + 1;
if forw.↑ = eof. then begin
if forw. at end of 1st half then begin
reload 2nd half
for := for + 1
end
elseif forw. at end of 2nd half then begin

Date : / /20
Page No.:

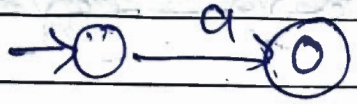
→ 2 methods to build Regex to DFA

1) Thompson ~~construct~~ subset construct.
2) Syntax tree

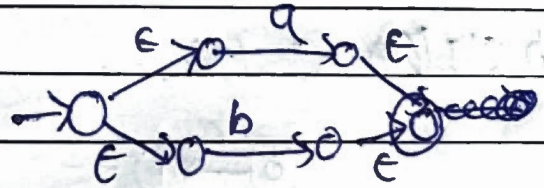
· we have seen 2

① Thompson subset construct

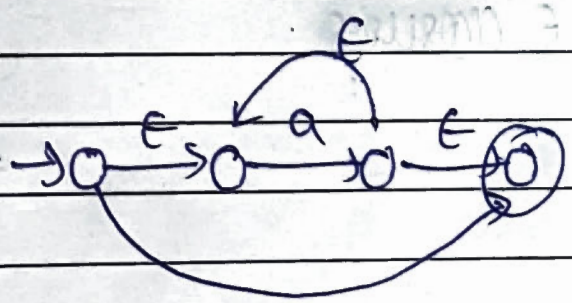
⇒ For any i/p symbol a, the corresp. FA is :-




⇒ For (a+b) or (a|b)

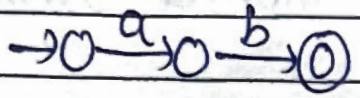



⇒ For a*,

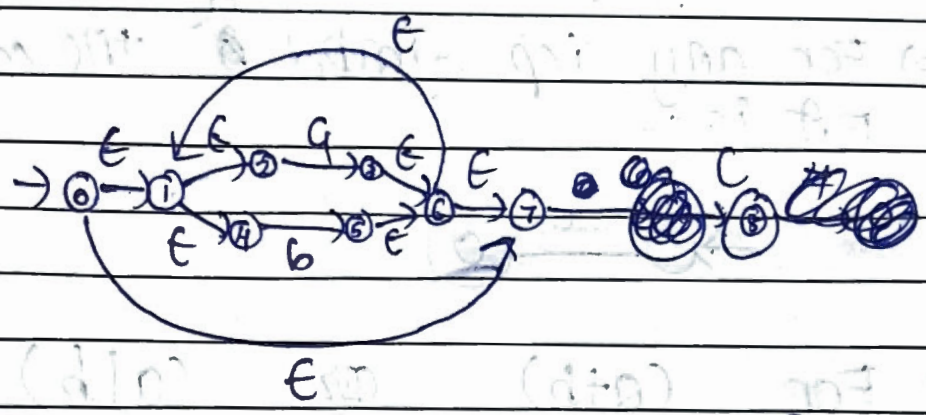


Date : / /20
Page No.:

⇒ For 'ab' [a.b]
↳ concat.




eg- (a+b)* c #
↳ endmarker (ignore)




· If '#' not given, then  ignore #.

· now label each state → 0, 1, 2, ...


· we need to remove ε transitn.
· so, we find ε closures

→

Date : / /20
Page No.:

| State | ε closure |
|---|---|
| 0 | {0,1,2,4,7} = A |
| 1 | |
| 2 | |
| 3 | |
| 4 | |
| 5 | |
| 6 | |
| 7 | |

- In thor, we used δ(A,a) to show transitn.
- Here, we do mov(A,a) to show transitn.

now, mov(0,a) = mov({0,1,2,4,7}, a) → ε closure
= ε closure(3) = {3,6,1,2,4,7} ⇒ B

... complete.

⇒ NOTE ⇒ If ~~str~~ regex ⇒ $a^+bc^*\#$

$aa^*bc^*\#$ (ε)

we can't already write $a^+$ in thompson construct°.

Date : / /20
Page No.:

→ 0 or more occurrence in Regex

eg ⇒ $a?b^*((c+d)^*d\#$

$\Downarrow$

$(a+\epsilon)b^*((c+d)^*d\#$

∴