CH.4 SYNTAX ANALYSIS / PARSER

→ Only Context free Grammar used to build parser.

→ Benefits of
- Representing any rule will be easier
- Parser built for this grammar will be optimised

→ Types of Parser: (parser can be built for)
i) Universal → any Grammar
ii) Top down → LL Grammar
iii) Bottom up → LR Grammar

LL → L → left to right scanning, L → left to right derivation

LR → right to left derivation

→ Role of Parser : It should detect the error as soon as possible and gives error message. (error recovery)
The error handler should not slow down the process of compilation.

→ ERROR RECOVERY TECHNIQUES IN PARSER

i) Error productions
ii) Global Correction.
i) Panic Mode : → On discovery of any error it discards (skips) the input symbols, one at a time until it encounters a synchronizing token.

eg: ; let ; be a synchronous symbol.
int a,, b ;
here after 'a' it encounters syntax error so it discards everything till it reads ; ∴ only int a; is read which is correct.

ii) Phrase level : → On discovering error it replaces remaining part of i/p with some other string.
(insert a char, delete a char, replace a char)

eg: int a, b,, c ;
here it skips the second comma.
→ Before doing this it reports the error.

phrase level better than panic

iii) Error Productions : you write the production for commonly occurring error.
eg: we commonly forget semicolon.

so, p1 : Decl → DT id-list

here, semicolon missing so if that is the case -

error correction
EC : Insert ;
Report ; missing

iv) Global Correction : → Algorithms available for choosing a minimal sequence of changes to obtain a globally least cost correction.

practically impossible to implement

→ Algo which calculates the cost associated with transforming the i/p into a related correct parse tree by insertion, deletion of tokens, etc.

see video, explained 24/1

eg:

CONTEXT FREE GRAMMARS

→ Consists of terminals, nonterminals, start symbol, and productions

G1:
eg: E → E + E (p1) | E * E (p2) | num (p3)

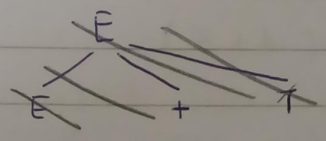
Non-terminals - E
terminals - num, +, *

G2: E → E + T | T
T → T * F | F
F → Num

input: num + num * num using left most derivation on G2

G2

E ⇒(lm) E + T (leftmost)
⇒ T + T
⇒ F + T
⇒ Num + T
⇒ Num + T * F
⇒ Num + F * F
⇒ Num + Num * F
⇒ Num + Num * Num



solved this way so * done first

in G1 it is not clear if * should be done first or + This is not correct in terms of math as we always want * to be done first.

But in G2, the Bottom most step is * which means it will be executed first.

G1

E ⇒(lm) E + E
⇒ E * E + E
⇒

here if we do leftmost derv. then + will be the bottom most ∴ it will be executed first This is wrong mathematically.

here you can say we can use p2 first instead of p1 to get * calculated first but there is no way to impose such a rule

→ G_2 is imposing in such a way that * will be calculated first always.

→ ∴ G_2 is an ambiguous grammar as there are multiple leftmost derivations for same input. (seen in next page)

Q $E \rightarrow E+E \mid E*E \mid -E \mid (E) \mid id$

input : $-(id+id)$

Sol" Leftmost Derivation

$$E \underset{lm}{\Rightarrow} -E$$
$$\underset{lm}{\Rightarrow} -(E)$$
$$\underset{lm}{\Rightarrow} -(E+E)$$
$$\underset{lm}{\Rightarrow} -(id+E)$$
$$\underset{lm}{\Rightarrow} -(id+id)$$

Rightmost Derivation

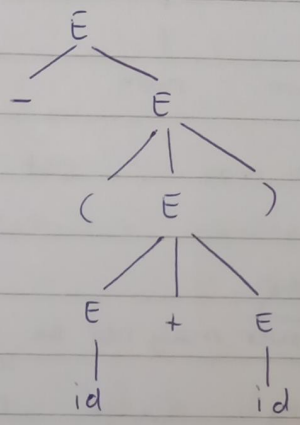
$$E \underset{rm}{\Rightarrow} -E$$
$$\underset{rm}{\Rightarrow} -(E)$$
$$\underset{rm}{\Rightarrow} -(E+E)$$
$$\underset{rm}{\Rightarrow} -(E+id)$$
$$\underset{rm}{\Rightarrow} -(id+id)$$

→ Parse Tree : Terminals are at leaf nodes & non-terminals are internal nodes.

In syntax tree, operands are leaf nodes & operators are internal nodes.

PARSE TREES & DERIVATION

→ Parse Tree for prev Q

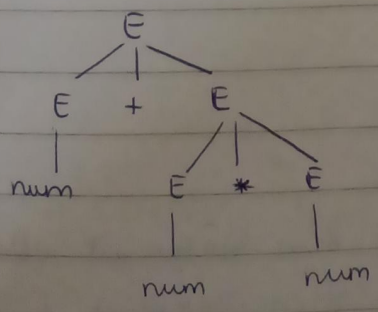


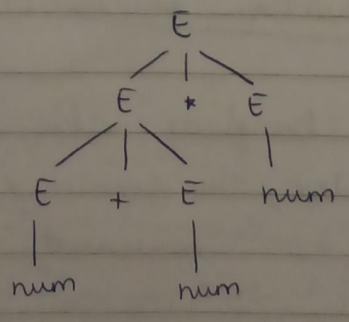
This is top down approach.

Q Build a parse tree, top to down approach using G1 & G2 grammar (done earlier) to obtain input: ~~id + id * id~~ num + num * num

Sol^n G1 has multiple parse trees ∴ ambiguous grammar

G1 : $E \rightarrow E+E \mid E*E \mid num$

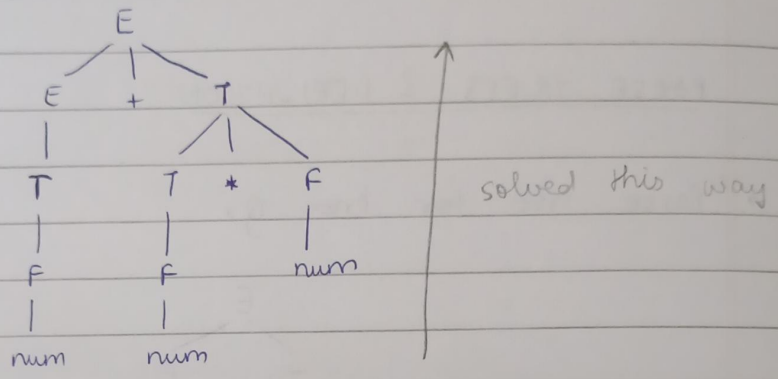




G_2 : $E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$f \rightarrow num$



only on tree ∴ not ambiguous.

WRITING A GRAMMAR

→ Sometimes a given grammar may not be suitable to build parser. Some techniques need to be applied to transform the grammer:

technique 1) Eliminating Ambiguity

technique 2) Eliminating left Reursion

technique 3) Performing left Factoring

always follow this order.

if ambiguous → technique 1

else if left recursive → technique 2

else if common prefix → technique 3

→ This techniques are used if you want to perform top down parser

→ No need for bottom up parser as they are strong enough to deal with this.

I ELIMINATING AMBIGUITY

→ The Dangling "else" Problem.

Consider, The grammar for if-else

Statement → if Expr then statement else Statement |
if Expr then Statement |
Otherstatement.

Terminals: if, then, else, Expr, otherstatement

→ a correct input string is

if Expr then if Expr then OtherStatement else OtherStatement

here 2 possible interpretations :

1) if Expr then
   if Expr then
   OtherStatement
   else
   OtherStatement

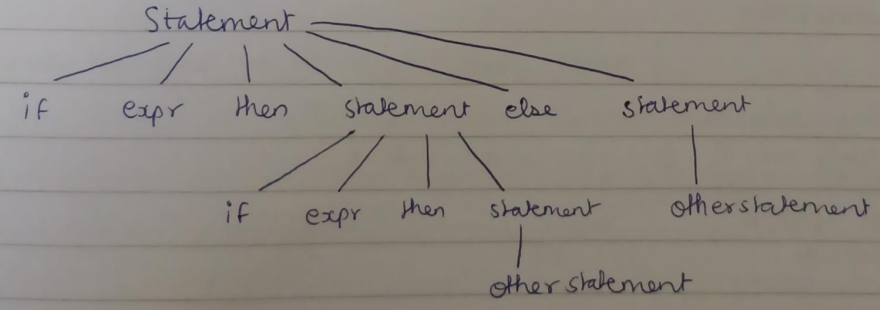
2) if Expr then
   if Expr then
   OtherStatement
   else
   OtherStatement

in, 1) the else statement corresponds to inner if
2) " " " " " outer if

∴ it is ambiguous which if does the else correspond to.

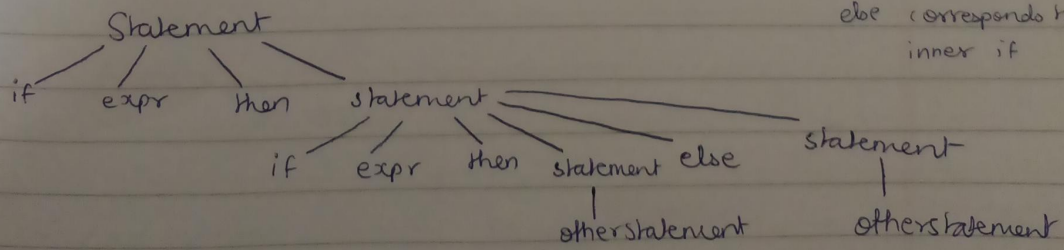
→ Parse Tree for the given input string :

2)



else corresponds to outer if

1)



else corresponds to inner if

→ Since there are 2 parse trees for same input, grammar is ambiguous.

→ There are 2 ways to eliminate ambiguity:

Method 1: Match else with the closest unmatched then

Method 2: Rewriting grammar so that the resultant grammar is not ambiguous. (be careful that new grammar accepts same language as old grammar)

In our dangling else problem if we use method 2:

(1) We rewrite unambiguous grammar by permitting only a "matched-stmt" (which is a new non-terminal introduced) to come before an else

i.e b/w then and else there should always be matched-stmt

(2) Matched-stmt has if then else construct

(3) Only matched-stmt can lead to a non terminal.

∴ New Grammar:

if then else — matched_stmt | unmatched_stmt — if then

stmt → matched_stmt | unmatched_stmt

matched-stmt → if expr then matched-stmt else matched-stmt | otherstatement

unmatched-stmt → if expr then stmt |
if expr then matched-stmt else unmatched-stmt

→ this takes care of if expr then otherstmt else if expr then otherstmt

→ This grammar ensures that if and else are always matched correctly.

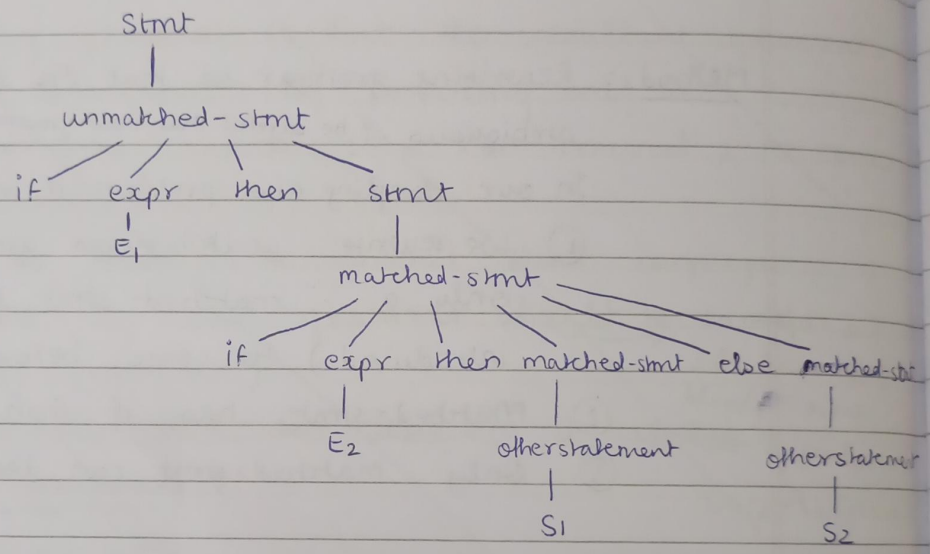
→

g) Write Parse Tree for:

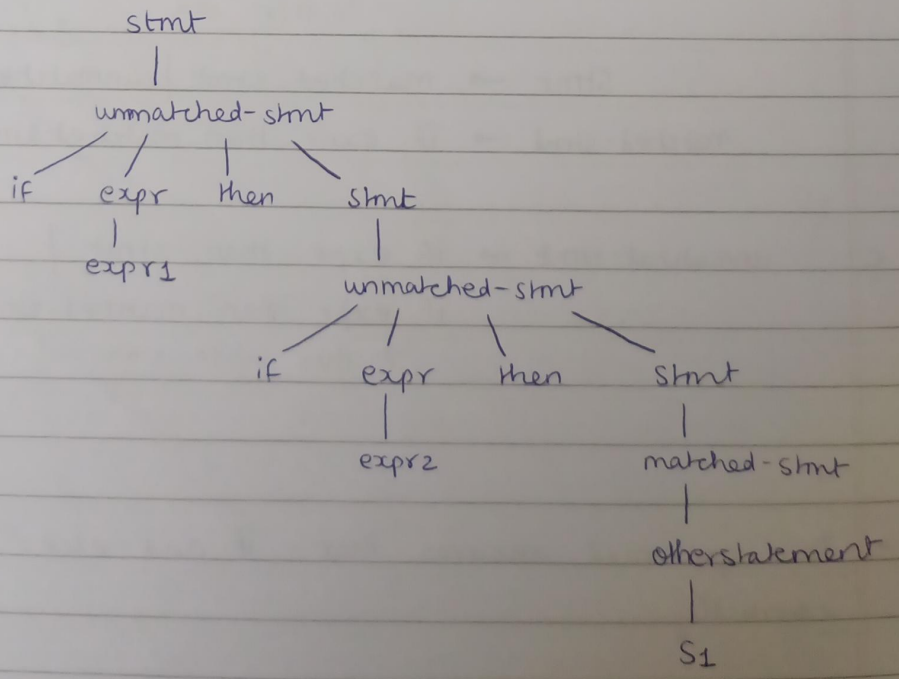
① if E_1 then if E_2 then S_1 else S_2

② if expr1 then if expr2 then S_1

Sol^n ①



②



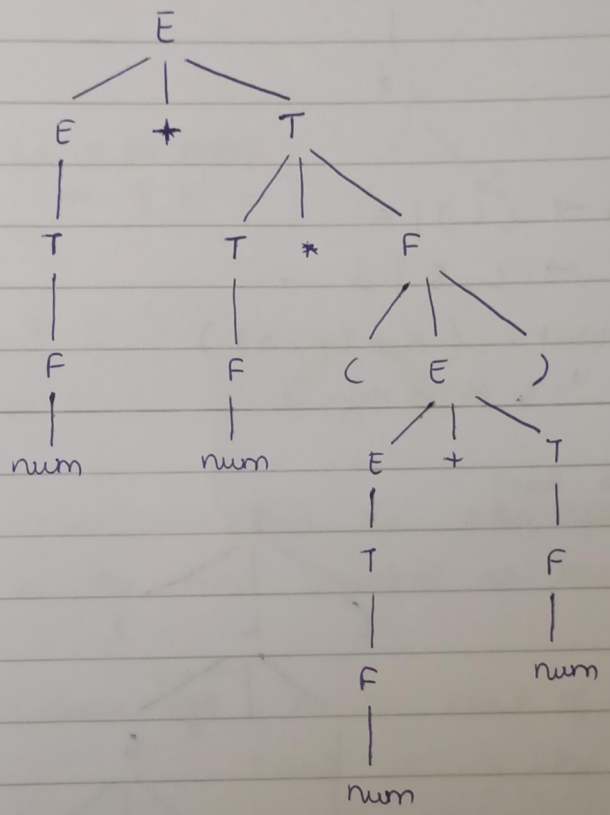
8) Derive a string using Parse Tree which contains 3 operators.

Grammar : $E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid num$

Sol^n one possibility,



String obtained : num + num * (num + num)

↳ read from left to right leaf nodes

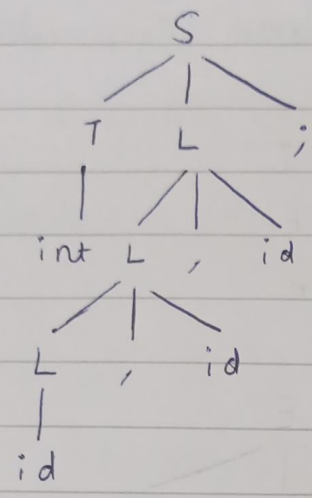
many such strings can be obtained

9 i) Grammar : $S \rightarrow TL;$

$T \rightarrow int \mid float;$

$L \rightarrow L, id \mid id$

Obtain string : int id, id, id;

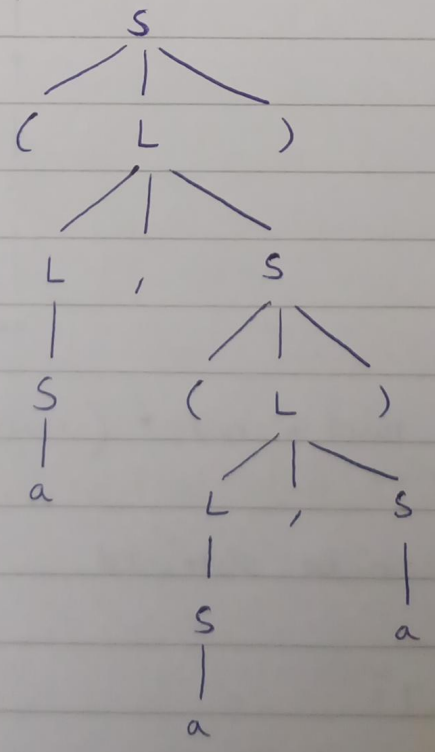


ii) $S \rightarrow (L) \mid a$

$L \rightarrow L, S \mid S$

input : (a , (a , a))

↳ i.e. obtain string



Writing grammar 2nd technique

II ELIMINATION OF LEFT RECURSION

$A \rightarrow A\alpha \mid \beta$ here, α & β can be terminals or non terminals

↳ general form of left recursive grammar

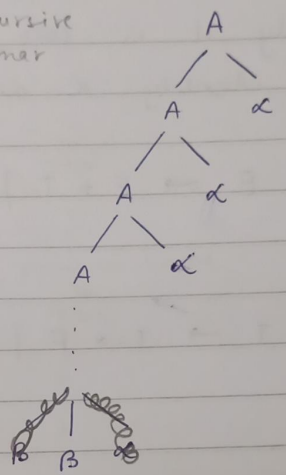
$A \underset{lm}{\Rightarrow} A\alpha$

$\underset{lm}{\Rightarrow} A\alpha\alpha$

$\underset{lm}{\Rightarrow} A\alpha\alpha\alpha$

$\underset{lm}{\Rightarrow} \beta\alpha^*$ → i.e 0 or more no. of α

β or $\beta\alpha$ or $\beta\alpha\alpha$ or ...



→ Left Recursive Grammar : • When the head of production appears as the first symbol on the body of the production.

in above grammar, A is head of production and it is the first symbol in the body

$A \rightarrow A\alpha$

head, first symbol

• This could cause infinite loop

→ Rewriting the grammar:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$ ↳ epsilon

} remember this formula

→

Q Rewrite

head body

$E \to E + T \mid T$

$T \to T * F \mid F$

head body

$F \to (E) \mid id$

Sol" Clearly it is left recursive.

now identify α and β

$E \to \overset{\alpha}{E + T} \mid \overset{\beta}{T} \quad \Rightarrow \quad E \to TE'$

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

$T \to \overset{\alpha}{T * F} \mid \overset{\beta}{F} \quad \Rightarrow \quad T \to FT'$

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

∴ New grammar:

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$ → remains same as not left recursive.

NOTE: In general,

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \beta_n$$

⇓ rewrite

$$A \to \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \cdots \mid \beta_n A'$$

$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

Q Rewrite ① $S \to Aa \mid b$

$A \to Ac \mid Aad \mid bd \mid e$

② $S \to a \mid \uparrow \mid (T) \mid T$

$T \to T, S \mid a \mid \uparrow \mid (T)$

① $A \to \underbrace{Ac}_{\alpha_1} \mid \underbrace{Aad}_{\alpha_2} \mid \underbrace{bd}_{\beta_1} \mid \underbrace{e}_{\beta_2}$

$\Downarrow$

$A \to bdA' \mid eA'$

$A' \to cA' \mid adA' \mid \epsilon$

remains same

$\therefore S \to Aa \mid b$

$A \to bdA' \mid eA'$

$A' \to cA' \mid adA' \mid \epsilon$

② $T \to \underbrace{T, S}_{\alpha_1} \mid \underbrace{a}_{\beta_1} \mid \underbrace{\uparrow}_{\beta_2} \mid \underbrace{(T)}_{\beta_3}$

$\Downarrow$

$T \to aT' \mid \uparrow T' \mid (T)T'$

$T' \to , ST' \mid \epsilon$

remains same

$\therefore S \to a \mid \uparrow \mid (T) \mid T$

$T \to aT' \mid \uparrow T' \mid (T)T'$

$T' \to , ST' \mid \epsilon$

III LEFT FACTORING

→ Some productions may have a common prefix. This makes it difficult to decide which production to choose.

eg: $A \Rightarrow \underbrace{Bcd}_{} \mid \underbrace{BdD}_{}$

both start with B so its confusing

→ This is solved using left factoring by introducing new non terminal.

→ This helps in removing conflict and also delaying the decision.

eg: $A \rightarrow Bcd \mid BdD$

$\Downarrow$

$A \rightarrow BA'$ → new non terminal A'

$A' \rightarrow cd \mid dD$

Q Dangling else grammar

stmt → if "expr" then stmt else stmt | if expr then stmt | other

(common prefix)

(For this grammar we should only use ambiguity technique as it is ambiguous but for eg: sake we will do left factoring)

Soln

stmt → if expr then stmt stmt' | other

stmt' → else stmt | ∈

-X- In general, if

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma_1$

$\Downarrow$

$A \rightarrow \alpha A' \mid \gamma_1$

$A' \rightarrow \beta_1 \mid \beta_2$

$A \to \alpha$

Q) Perform left factoring

$E \to E \text{ sub } E \text{ sup } E \mid E \text{ sub } E \mid E \text{ sup } E \mid \{E\} \mid e$

↓

(satisfies left recursion but for eg: sake let's do left factoring)

Sol" Method 1: $E \to \underbrace{E}_{\alpha} \underbrace{\text{sub } E \text{ sup } E}_{\beta_1} \mid \underbrace{E}_{\alpha} \underbrace{\text{sub } E}_{\beta_2} \mid \underbrace{E}_{\alpha} \underbrace{\text{sup } E}_{\beta_3} \mid \underbrace{\{E\}}_{\gamma_1} \mid \underbrace{e}_{\gamma_2}$

$\Downarrow$

$E \to EE' \mid \{E\} \mid e$

$E' \to \underbrace{\text{sub } E}_{\alpha} \underbrace{\text{sup } E}_{\beta_1} \mid \underbrace{\text{sub } E}_{\alpha} \mid \underbrace{\text{sup } E}_{\gamma_1}$ ($\beta_2 = \epsilon$)

$\Downarrow$

$E' \to \text{sub } E E'' \mid \text{sup } E$

$E'' \to \text{sup } E \mid \epsilon$

$\therefore$ $E \to EE' \mid \{E\} \mid e$

$E' \to \text{sub } E E'' \mid \text{sup } E$

$E'' \to \text{sup } E \mid \epsilon$

Method 2: $E \to \underbrace{E \text{ sub } E}_{\alpha} \underbrace{\text{sup } E}_{\beta_1} \mid \underbrace{E \text{ sub } E}_{\alpha} \mid \underbrace{E \text{ sup } E}_{\beta_1} \mid \underbrace{\{E\}}_{\gamma_1} \mid \underbrace{e}_{\gamma_2}$

$\Downarrow$

$E \to \underbrace{E \text{ sub } E}_{\alpha} E' \mid \underbrace{E}_{\alpha} \underbrace{\text{sup } E}_{\beta_2} \mid \{E\} \mid e$

$E' \to \text{sup } E \mid \epsilon$

$\Downarrow$

$E \to EE'' \mid \{E\} \mid e$

$E' \to \text{sup } E \mid \epsilon$

$E'' \to \text{sub } E E' \mid \text{sup } E$

TOP DOWN PARSING (only for LL grammar)

→ 2 types in which it can be done:
i) Backtracking (powerful but not practical)
ii) Predictive Parsing

→ Parse Tree is built from start symbol of grammar & each non terminal is expanded

i) **Backtracking**

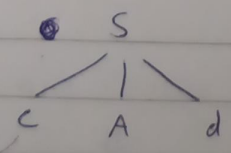
Note: in parsing end of input string is taken as $ symbol

Consider grammar,
S → cAd
A → ab | a

input string : c a d $

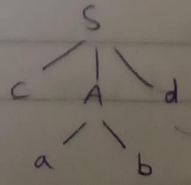
In backtring the first production in the list is used to expand a non terminal, if that is not a match we backtrack and choose next production & so on.
(i.e A → ab | a , ab will be chosen first.)

① ↓
c a d $



c matches first symbol move forward.

② c ↓
a d $



→ first prod^{n} expanded A → ab

input parse tree

a matches a

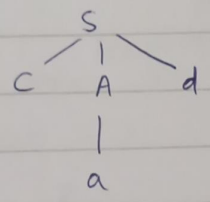
c a d $

input parse tree

d doesn't match b

Backtrack A and choose next production in list $A \rightarrow a$

③ c a d $



a matches a

c a d $

d matches d

∴ string is accepted.

Q

$S \rightarrow aAD \mid aAC \mid aCD$

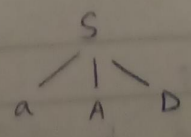
$A \rightarrow bD \mid bC \mid \epsilon$

$C \rightarrow b \mid bDa$

$D \rightarrow d \mid \epsilon$

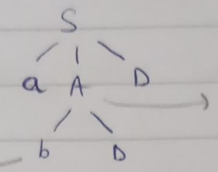
input : a b d a $

Soln ① a b d a $



a matches first symbol, move forward

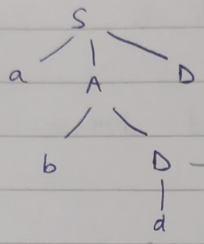
② a b d a $



expand with first prod" of A

b matches move forward

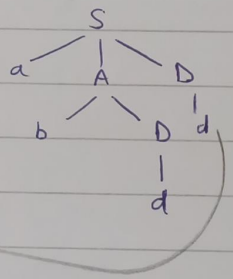
③ a b d a $



→ expand first prod" of D

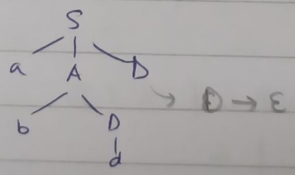
d matches move forward

④ a b d a $



doesn't match choose diff prod"

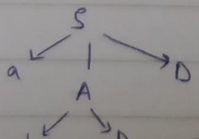
④ a b d a $



D → ε

no match more back track

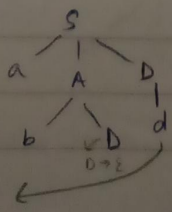
⑤ a b d a $



→ pick second prod" for D

D → ε

⑥ a b d a $



match

keep going until all prod" exhausted or string accepted.

ii) Predictive Parsing

→ 2 types:
- Recursive Descent
- LL(1) or table driven

→ Both methods use First and Follow for lookahead.

Recursive Descent

→ It looks ahead and then picks a suitable production unlike backtracking

i.e based on input string which production of a non terminal is best, that is chosen.

eg:
$E \rightarrow num\ T$
$T \rightarrow *\ num\ T \mid \epsilon$
Write pseudo code for recursive descent.

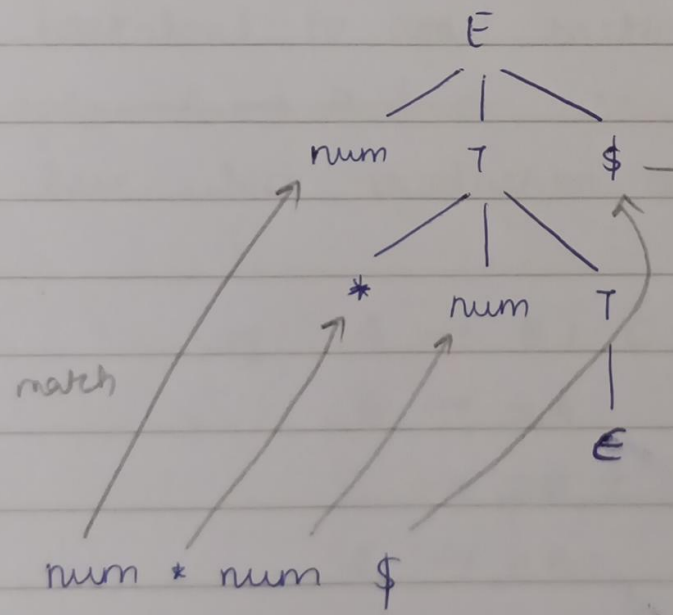
→ for terminal we check whether it matches with input symbol.
→ for non terminal we call this function
→ ϵ handled using First & Follow
→ lookahead pointer pointing to i/p symbol

```
Procedure T  →  i.e call this f" when T encountered
{
    if lookahead = '*'
    {
        lookahead = next token
        if lookahead = 'num'
        {
            lookahead = nexttoken ();
            T();
        }
```

```
        else
            error;        → i.e * not followed by num is not an
        }                     accepted string
        else NULL;
    }          → means ∈

Procedure E ()
{
                check slides.
    if ( lookahead = num ) then
    {
        match(num); lookahead = next token ;
        T();
    }
    else
        error;
    if ( lookahead = $ )          // start symbol so check if once
    {                                everything is parsed is final symbol
                                     $
        declare success;
    }
    else
        error;
}
```

→ Draw parse tree for input: num * num $, using ~~predictive parsing~~ recursive descent fn's defined in prev. page.



→ when resolving input string from left to right and matching parse tree symbols with them, if finally $ matches $ in the end then input string is accepted.

FLEX (lab & slides) it is part of theory portion

FIRST AND FOLLOW (used in predictive parsing)

→ both first and follow are set of terminals.

→ we use lookahead to avoid backtracking using first & follow sets

~~eg: A → B | C~~

→ Helps to chose which production to use to expand a non terminal.

eg: A → B | C

B → xA

C → yZ

Z → yB

when expanding A, just by looking at B & C we cannot tell which production is best.

Hence we look at first terminal given by each of the non terminals

like, B gives x first

C → Z → y ∴ C gives y first.

→ Consider a → alpha | beta

→ To choose among two alternatives based on input symbol.

→ FIRST (alpha) is set of terminals that can be beginning of the strings, derived from alpha

→ FOLLOW (A) is set of terminals that occurs to right of A

Rules for Calculation of FIRST

1) if 'x' is a terminal 'a' then FIRST(x) = {a}
2) if 'x' is a non terminal 'X' and X → aβ then FIRST(X) = {a}
3) if 'x' is a non terminal 'X' and X → ε (epsilon) then FIRST(X) = {ε}
4) if X → y1 y2 y3 ... yk then place FIRST(y1) in FIRST(X)

i) if FIRST(y_1) has ε then add FIRST(y_2) and so on

ii) if ε is in FIRST(y_j) where $j = 1, 2, \ldots k$ then add ε to FIRST(X). ↳ i.e every thing from FIRST(y_1) to FIRST(y_k) is ε

RULES FOR FOLLOW SET

1) Whenever you want to find follow of a symbol, find the symbol in R.H.S
2) if symbol is start symbol, add $ to follow set.
3) if symbol is followed by a terminal, straightaway add the terminal to the follow set.
4) if the symbol is followed by non-terminal, find the first of the non-terminal.
   i) if the first set contains ε, exclude ε from follow sets & replace it in the production.
   ii) if even after replacement by ε, it is followed by terminal or non-terminal, repeat the process.
5) if the symbol is last one in the production, then add follow of head into follow set.

Note: Follow set never contains ε.

Q $X \rightarrow y_1\ y_2\ y_3$

$y_1 \rightarrow a \mid \varepsilon$

$y_2 \rightarrow b \mid \varepsilon$

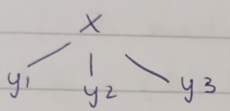
$y_3 \rightarrow c \mid \varepsilon$

Write First and Follow sets for all Non-terminals.

→

Non Terminal	First	Follow
X	{a, b, c, ε}	{$}
y_1	{a, ε}	{b, c, $}
y_2	{b, ε}	
y_3	{c, ε}	

① FIRST (X) :



FIRST (X) will have FIRST (y_1)

y_1 → a → include a in FIRST(X)

FIRST (y_1) = {a, ε}

~~ε ∈ FIRST(y_1)~~ but if y_1 → ε

Then we move to FIRST (y_2)

y_2 → b → include b

FIRST (y_2) = {b, ε}

but if y_2 → ε

Then we move to FIRST (y_3)

y_3 → c → include c

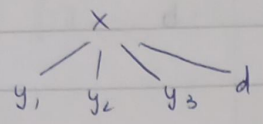
FIRST (y_3) = {c, ε}

but if y_3 → ε

This would mean FIRST(y_1), FIRST (y_2), FIRST (y_3) all have ε so, FIRST(X) also has ε

let's say if $X \rightarrow y_1\ y_2\ y_3\ d$ (d: non terminal)

now,



then, even though FIRST(y_1), FIRST(y_2) & FIRST(y_3) have ε

FIRST(d) = {d}

it does not have ε ∴ FIRST(X) will not have ε

∴ FIRST(X) = {a, b, c, d}

i.e FIRST(X) will only have ε if FIRST of everything on rhs of X has ε.

---

(2) FOLLOW(X):

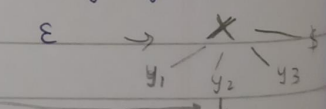
- Start symbol is X so its follow set includes $
- X does not appear in any other production so it won't be followed by anything else.

(3) FOLLOW(y_1):

y_1 is followed by y_2 as seen in production of X

∴ include FIRST(y_2) in FOLLOW(y_1) except ε

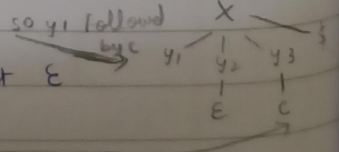
↳ {b, ε}



but if y_2 was replaced with ε then y_1 is followed by y_3

∴ include FIRST(y_3) in FOLLOW(y_1) except ε

↳ {c, ε}



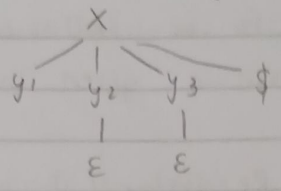
but if y_3 was also replaced with ε then y_1 is followed by nothing, y_1 becomes last symbol of production. So FOLLOW of its parent i.e FOLLOW(X) is added to FOLLOW(y_1)

↳ X → y_1 X is parent of y_1

Note: if a terminal is not followed by anything add ~~FIRST~~ Follow of its parent (head of prod^n) to the terminal's follow set. (As seen in case of y_1)

y_1 followed by $ when both

i.e {$}



~~similarly~~ ∴ FOLLOW(y_1) = { b, c, $ }

similarly, FOLLOW(y_2) = { c, $ } ← y_2 followed by y_3 → when y_3 is ε nothing is followed by y_2 so take parent X follow set

FOLLOW(y_3) = {$}

$X \rightarrow y_1\ y_2\ y_3$ — nothing follows y_3 → nothing follows y_3 so only parent follow set added

if, $X \rightarrow y_1\ y_2\ y_3\ d$

FOLLOW(X) = { ~~to~~ $ } ← if y_2 & y_3 both are ε still y_1 followed by d

FOLLOW(y_1) = { b, c, d }

FOLLOW(y_2) = { c, d } ← if y_3 is ε, y_2 followed by d

FOLLOW(y_3) = { d } → y_3 always followed by d.

→ FIRST is used to choose among the productions.

→ FOLLOW is used to check if we should expand the non-terminal with an ε production or not, if we do choose ε but the follows doesnot contain any symbol matching the current symbol then we can correctly report error.

Eg: input: a x for prev grammar

↑ y_1 ↑ cur symbol

→ FIRST used to calc FOLLOW

→ Steps needed to be followed for constructing a parse table & to parse a given input string

- Eliminate left Recursion (if any) in the grammar G.
- Perform left factoring on G.
- Find FIRST and FOLLOW on the symbols in G.
- Construct predictive parse table
- Check if the input string is accepted by parser using the parse table entries.

Q $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

Find First & Follow set

Sol" Step 1: Remove left Recursion

i) $E \to \overbrace{E+T}^{\alpha} \mid \overbrace{T}^{\beta}$

$\Downarrow$

$E \to TE'$

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

ii) $T \to \overbrace{T*F}^{\alpha} \mid \overbrace{F}^{\beta}$

$\Downarrow$

$T \to FT'$

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

New grammar ∴

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

Step 2 : There is no left factoring (i.e common prefix not there)

Step 3 :

Non Terminals	First	Follow
E	{ (, id }	{ $,) }
E'	{ +, ε }	{ $,) }
T	{ (, id }	{ +, $,) }
T'	{ *, ε }	{ +, $,) }
F	{ (, id }	{ *, +, $,) }

FOLLOW(E) : $ and in F → (E)
↳ here E followed by)

∴ { $,) }

FOLLOW(E') : in none of the productions E' is followed by anything so, FOLLOW (E') = FOLLOW (parent)
here parent is E.

∴ { $,) }

FOLLOW (T) : T is followed by E'
So FIRST (E') added { + } → no ε
if E' was replaced with ε then T would be followed by nothing
So, FOLLOW (E') & FOLLOW (E) added
→ parents of T

∴ { +, $,) }

FOLLOW (T') : T' is followed by nothing in all of the productions
∴ FOLLOW (T') = FOLLOW (T)
↳ parent

∴ { +, $,) }

FOLLOW(F) : F is followed by T' so
add FIRST(T') i.e { * } → no ε
if T' replaced by ε then nothing follows
F so; FOLLOW(T) & FOLLOW(T') added
→ parent ←

∴ { *, +, $,) }

TABLE DRIVEN PARSERS / LL(1) PARSERS

(comes under predictive parsing which is under top down parsing)

→ Points to remember while constructing Parse Table

- Make an empty table in which rows are labelled with the non terminals & cols with the terminals of the grammar
- $ is explicitly added as one of the col. headers in every parse table
- Never add ε as col. header in parse table

→ Construction of Predictive Parse Table

Algorithm: Construction of Predictive Parse Table
Input: Grammar G
Output: Parsing Table M

Method:

① For each production of the form A → α of the grammar, do ② & ③

② For each terminal symbol 'a' in FIRST(α), add A → α to M[A, a] → i.e row header A & col header a

③ IF ε is in FIRST(α) add A → α to M[A, b] for each terminal b in FOLLOW(A), ~~add A → α to M[A, $]~~ IF ε

~~④ Make each undefined entry of M to be error~~ is in FIRST(α) & $ is in FOLLOW(A), add A → α to M[A, $].

④ Make each undefined entry of M to be error.

→ LL(1) Parser / Table Driven Parser

1) Check if grammar is ambiguous. if so, eliminate
2) Remove left recursion, if required
3) Perform left factoring, if required
4) Compute First & Follow
5) Construct predictive parse table
6) Check if i/p string is accepted by parser using the parse table entries.

stupid adal I already wrote this

Q
$A \rightarrow BC$
$B \rightarrow S$
$C \rightarrow ; A \mid \varepsilon$

Construct LL(1) Parser for the given grammar & show parsing action for the i/p string S;S$

1) No ambiguity
2) No left recursion
3) Left factoring not needed

4)

Symbol	FIRST	FOLLOW
A	{ S }	{ $ }
B	{ S }	{ ;, $ }
C	{ ;, ε }	{ $ }

here, FOLLOW(A) = FOLLOW(C)
and FOLLOW(C) = FOLLOW(A)
∴ deadlock.

$C \rightarrow ; A$
$A \rightarrow BC$

as both are last symbols in each other's prod^n

Since, A is start symbol so, FOLLOW(A) includes $ ∴ FOLLOW(A) = FOLLOW(C) = $

→

5) Parse Table

Non Terminal ↓ / Terminal →	s	;	$
A	A → BC		
B	B → s		
C		C → ; A	C → ε

For prod^n $A \rightarrow BC$, $\alpha = B$

we see FIRST(B) = {s}

∴ from rule ② of parse table construction method

add $A \rightarrow BC$ to M[A, s] ↳ here $\alpha = B$

For prod^n $B \rightarrow s$, $\alpha = s$

use FIRST(s) = {s} as s is terminal

∴ add $B \rightarrow s$ to M[B, s]

For prod^n $C \rightarrow ;A$, $\alpha = ;$

FIRST(;) = {;}

add $C \rightarrow ;A$ to M[C, ;]

For prod^n $C \rightarrow \varepsilon$, $\alpha = \varepsilon$

FIRST(ε) = {ε}

so, check FOLLOW(C) = {$}

∴ add $C \rightarrow \varepsilon$ to M[C, $]

∴ only for ε prod^n we need to check FOLLOW set of head. If the terminal is in FOLLOW set we can add ε prod^n in that column.

→

6) Parsing i/p string

Stack	input buffer	Action
A $	s ; s $	A → BC
BC $	s ; s $	B → s
sC $	s ; s $	Match
C $	; s $	C → ; A
; A $	; s $	Match
A $	s $	A → BC
BC $	s $	B → s
sC $	s $	Match
C $	$	C → ε
$	$	Accept

→ Stack starts with start symbol A ($ will already be there)
so, A $

→ In each step we are comparing top of stack with first symbol in input buffer. We put the corresponding parse table entry in action.

eg: M[A, s] = A → BC

Since an entry exists we ~~push~~ pop the symbol from stack and push its corresponding prod^n.

∴ pop A and push BC

→ When a terminal is on top of stack and it matches the first symbol in i/p buffer then Action becomes Match and we pop the terminal from stack and increment input buffer pointer to next symbol.

→ When $ matches $ we accept the input string.

Q $S \to a \mid \uparrow \mid (T)$ input string: $(a,a)\$$

$T \to T, S \mid S$

1) Left Recursion present

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

$T \to \underbrace{T,S}_{\alpha} \mid \underbrace{S}_{\beta}$

$\Downarrow$

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

$T \to ST'$

$T' \to ,ST' \mid \varepsilon$

2) No left factoring needed

FIRST & FOLLOW 3)

Symbols	FIRST	FOLLOW
S	{ a, ↑, (}	{ , , $ }
T	{ a, ↑, (}	{) }
T'	{ , , ε }	{) }

Parse Table 4)

NT \ T →	a	↑	(	)	,	$
S	$S \to a$	$S \to \uparrow$	$S \to (T)$			
T	$T \to ST'$	$T \to ST'$	$T \to ST'$			
T'				$T' \to \varepsilon$	$T' \to ,ST'$	

for prod^n $T \to ST'$, $\alpha = S$

$FIRST(S) = \{ a, \uparrow, (\}$

$\therefore$ $M[T, a]$, $M[T, \uparrow]$ and $M[T, (]$ are all $T \to ST'$

5)

Stack	Input buffer	Action
S $	(a,a) $	S → (T) ← M[S, (]
(T) $	(a,a)$	Match
T) $	a,a)$	T → ST'
ST') $	a,a) $	S → a
aT') $	a,a) $	Match
T') $	,a) $	T' → ,ST'
,ST') $	,a) $	Match
ST') $	a) $	S → a
aT') $	a) $	Match
T') $	) $	T' → ε
) $	) $	Match
$	$	Accept

NOTE: LL(1) is called so, because we are looking ahead by 1 character only.
If it is not clear or ambiguous whether which production to choose by looking ahead by just one character then it is not LL(1) grammar (i.e M[A,a] has multiple entries)

Q Construct table driven parser for the given grammar & show parsing actions for string ba $

S → AaAb | BbBa
A → ε
B → ε

Soln
1) No left recursion
2) No left factoring needed

→

3)

Symbols	FIRST	FOLLOW
S	{a, b}	{$}
A	{ε}	{a, b}
B	{ε}	{a, b}

4) Parse Table

NT \ T	a	b	$
S	S → AaAb	S → BbBa	
A	A → ε	A → ε	
B	B → ε	B → ε	

5)

Stack	Input Buffer	Action
S$	baa$	S → BbBa
BbBa$	baa$	B → ε
bBa$	baa$	Match
Ba$	aa$	B → ε
a$	aa$	Match
$	$	Accept

Now, parse input aba$

Stack	Input Buffer	Action
S$	aba$	S → AaAb
AaAb$	aba$	A → ε
aAb$	aba$	Match
Ab$	ba$	A → ε
b$	ba$	Match
$	a$	Unsuccessful

here, $ and a both terminals, do not match.
∴ string rejected

check out $E \to E+T \mid T$ problem in textbook

Q $S \to UVW$

$U \to (S) \mid asb \mid d$

$V \to aV \mid \varepsilon$

$W \to cW \mid \varepsilon$

i/p string : daacc $

Sol^n 1) No left recursion

2) No left factoring needed

3)

Symbol	FIRST	FOLLOW
S	{ (, a, d }	{ $,), b }
U	{ (, a, d }	{ a, c, $,), b }
V	{ a, ε }	{ c, $,), b }
W	{ c, ε }	{ $,), b }

4)

NT \ T	(	a	c	d	b	)	$
S	S→UVW	S→UVW		S→UVW			
U	U→(S)	U→asb		U→d			
V		V→aV	V→ε		V→ε	V→ε	V→ε
W			W→cW		W→ε	W→ε	W→ε

6)

Stack	Input Buffer	Action
S$	daacc$	S→UVW
UVW$	daacc$	U→d
dVW$	daacc$	Match
VW$	aacc$	V→aV
aVW$	aacc$	Match
VW$	acc$	V→aV
aVW$	acc$	Match
VW$	cc$	V→ε
W$	cc$	W→cW
cW$	cc$	Match

→

Stack	Input	Action
W$	c$	W → cW
cW$	c$	Match
W$	$	W → ε
$	$	Accept

slight change from prev q

Q $S \to UVW$

$U \to (S) \mid asb \mid d \mid \varepsilon$

$V \to aV \mid \varepsilon$

$W \to cW \mid \varepsilon$

ip string : daacc$

Sol^n

Symbol	FIRST	FOLLOW
S	{ (, a, d, c, ε }	{ $,), b }
U	{ (, a, d, ε }	{ a, c, $,), b }
V	{ a, ε }	{ c, $,), b }
W	{ c, ε }	{ $,), b }

NT \ T	a	b	c	d	(	)	$
S	S → UVW	S → UVW	S → UVW	S → UVW	S → UVW	S → UVW	S → UVW
U	U → asb, U → ε	U → ε	U → ε	U → d	U → (S)	U → ε	U → ε
V	V → aV		V → ε			V → ε	V → ε
W		W → ε	W → cW			W → ε	W → ε

$S \to UVW$, FIRST(U) has ε

and FOLLOW(S) = { $,), b }

so all of them will have $S \to UVW$.

∴ M[U, a] has 2 entries ∴ This is not LL(1) grammar

TOP DOWN VS BOTTOM UP PARSING

→ In top down parsing you expand non terminals with their productions.
In the end we match all the leaf nodes from left to right with input string. (head of prod" expanded to body of prod")

→ In bottom up parsing we start with the input string (consisting of only terminals)
We reduce terminals to non terminals using their productions (body of production reduced to head of prod")

eg: Bottom up parsing

Grammar:

$e \longrightarrow (e+e)$

$e \longrightarrow (e*e)$

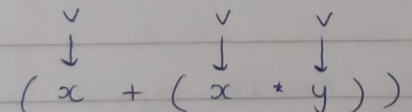
$e \longrightarrow v$

$v \longrightarrow x$

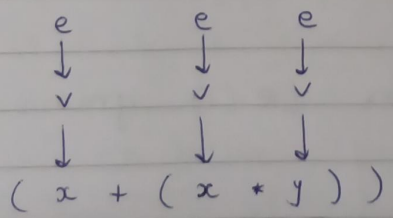
$v \longrightarrow y$

Input : $(x+(x*y))$

①



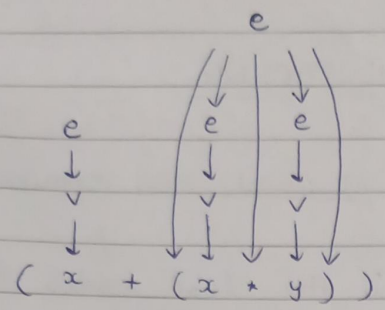
②



Now check for v in RHS, only $e \rightarrow v$ so reduce

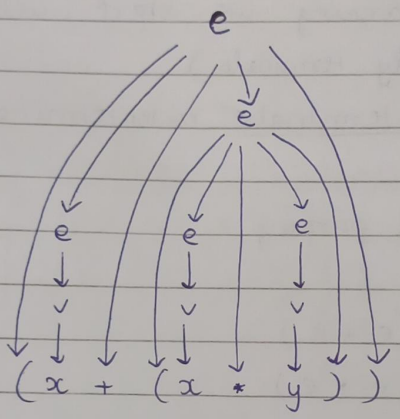
Now, we can reduce $(e*e)$ to e

③



So on until,

Final Parse Tree:



∴ we were able to obtain start symbol e from input string.

HANDLE PRUNING

→ Handle is the substring which matches right side of the production and we can reduce such string by a non terminal on the LHS of the production.

→ Reduction of a string or handle by a suitable Non terminal is called pruning.

Note: Bottom up parsing uses LR(1) grammar

TOP DOWN (cont.)

use for NCQ → Predictive parsers can be constructed for a class of grammars LL(1) where 1 symbol is used for look ahead.

for 3M draw parse table

→ A grammar G is LL(1) if and only if the following conditions hold good for $A \to \alpha \mid \beta$:

i) For no terminal 'a' do both α & β derive strings beginning with 'a'.
i.e FIRST(α) and FIRST(β) should be disjoint.

ii) Atmost 1 of α & β can derive empty string.

iii) If $\beta \overset{*}{\Rightarrow} \varepsilon$ (if β derives ε in one or more steps) then α doesn't derive any strings beginning with the terminal in FOLLOW(A). i.e FIRST(α) & FOLLOW(A) are disjoint.
(∵ itll be ambiguous to chose which prod")

eg:

$S \to A \mid B$
$A \to) \mid X$
$X \to (S)$
$B \to \epsilon$

here, S can be expanded to B or A
→ if B is chosen we get ϵ so we check ↳ FIRST(B)
FOLLOW(S).
FOLLOW(S) is) as seen in prod" X
→ if A is chosen we get) again. ↳ FIRST(A)

∴ it becomes ambiguous whether to chose B or A as both give)

here, ~~B is X~~ A is α and S is A
~~A is B~~ B is β from $A \to \alpha \mid \beta$

here, $B \overset{*}{\Rightarrow} \varepsilon$ and FOLLOW(S) & FIRST(A) are not disjoint ∴ this is not LL(1) grammar

LR(1) Grammar

→ Shift Reduce Parsers

G: $E \rightarrow E + E$

$E \rightarrow id$

input string: id + id + id

rightmost derivation

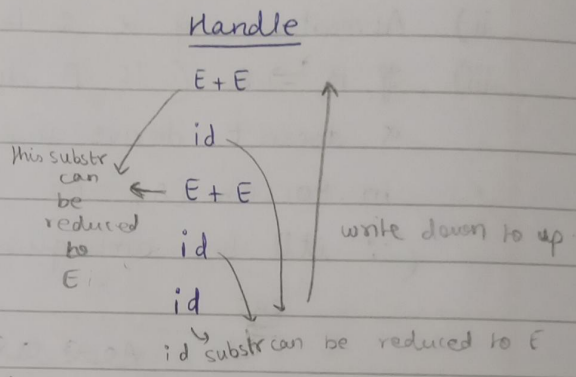
$E \overset{rm}{\Rightarrow} E + \underline{E}$

$\overset{rm}{\Rightarrow} \underline{E} + id$

$\overset{rm}{\Rightarrow} E + \underline{E} + id$

$\overset{rm}{\Rightarrow} \underline{E} + id + id$

$\overset{rm}{\Rightarrow} id + id + id$



Basically from bottom to top we checked if any substr of the input is a handle of any prod^n and accordingly reduced it.

Note: In bottom up parser, in stack the $ is written on the left as compared to top down where its written on the right.

→ 4 operations in top down parsing:

1) **Shift** - Shifting symbol from input and pushing it to stack.

2) **Reduce** - Reducing the handle by the non-terminal
↳ which is on top of stack
i.e replace handle with head of prod^n.

3) error (stack empty but i/p buffer still has some symbols left)

4) accept. ($ matches $)

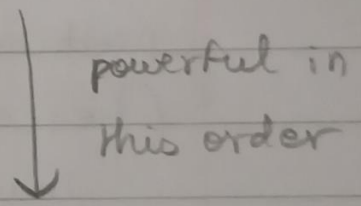
Shift - Reduce Conflict - Sometimes a handle is ready to be reduced but if we shift instead that new handle could also be reduced to some other prod^n. (check anything else problem in slides)

Reduce - Reduce Conflict - When i/p substring appears as handle in more than one production. So it'll be ambiguous which non-terminal to reduce to.

→ Its difficult for parsers to eliminate reduce-reduce conflict.

LR PARSER

- Shift reduce parser is general class of bottom up parser.
- 1 level down in hierarchy, LR parser
- Types of LR parsers
  - → SLR parser - simple LR (basic)
  - → Cannical LR parser
  - → LALR parser - lookahead



- Bottom up parser is more powerful than top down.
- More complex, so difficult to construct in hand.
- LR parser generator is usually used.

→ LR can have shift-reduce conflict but no reduce-reduce conflict.

WHY LR PARSERS ?

→ LR parser can be constructed to recognize most of the programming languages for which CFG can be written
→ LR parser works using non shift reduce technique.
→ LR parser can detect a syntactic error as soon as possible
→ Class of grammar that can be parsed by LR parser is a superset of class of grammars that can be parsed using predictive parsing.
(i.e all grammars with top down parsers can have bottom up parsers but not vice versa)
↳ eg: ambiguous grammars like dangling else can be done using bottom up but not top down
(i.e grammars that can be parsed using LL can be parsed using LR but not vise versa)

SLR(1) PARSING

ITEMS AND LR(0) AUTOMATON

→ Items & LR(0) automaton used to avoid shift-reduce conflict.
↳ i.e DFA

→ LR parser maintains states to keep track of where we are in a parse to avoid shift-reduce conflict.

→ Each state represents a set of "items".
→ An LR(0) item of a grammar G is a prod" of G with a dot at some position of the body.
↳ this indicates how much of that prod" has been parsed.
→ ∴ an item indicates how much of a prod" we have seen at a given point in the parsing process.

eg: Production - $A \rightarrow XYZ$

Then, Items are

1) $A \rightarrow \cdot XYZ$ means nothing parsed yet
2) $A \rightarrow X \cdot YZ$ means X has been parsed
3) $A \rightarrow XY \cdot Z$ means XY have been parsed
4) $A \rightarrow XYZ \cdot$ means XYZ have been parsed. Now that whole production is parsed, XYZ can be reduced to A.

∴ when • reaches the end, we are ready to reduce.

∴ if a state contains items of type 4 only then we reduce
if a state contains items of type 1, 2, 3 we shift
thus avoiding shift-reduce conflict.

eg: $A \rightarrow \varepsilon$, what is the item?

since ε means we are not consuming input, we are ready to reduce immediately

∴ $A \rightarrow \cdot$ is the item.

Q $S' \rightarrow S$

$S \rightarrow (S)S \mid \varepsilon$

Note: This is an augmented grammar i.e a new start symbol introduced as the original start symbol was appearing in the RHS of prod^n creating confusion.

Why augmentation? given in slides

so, $S \rightarrow (S)S \mid \varepsilon$

⇓

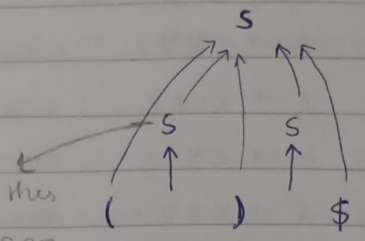
$S' \rightarrow S$

$S \rightarrow (S)S \mid \varepsilon$

now S' is the start symbol

we did this becaus when S appears in the parse tree, we

can't tell if that is the start symbol S or the S which is part of prod".



does this S mean we have reached end of parsing i.e start symbol found or is there more? This eg: is simple so you can't tell but for complex grammars it becomes very confusing to understand.

Steps in Bottom up Parsing:

1) Augment the grammar

∴ all items for given G are:

i) $S' \to \cdot S$
ii) $S \to (S)S \cdot$
iii) $S \to \cdot (S)S$
iv) $S \to (\cdot S)S$
v) $S \to (S \cdot)S$
vi) $S \to (S) \cdot S$
vii) $S \to (S)S \cdot$
viii) $S \to \cdot$

CLOSURE OF ITEM SETS

→ I - set of items for G (grammar)

→ Closure (I) - 2 rules

i) Initially add every item in I to closure(I).

ii) IF $A \to \alpha \cdot B\beta$ is in closure(I) and $B \to \gamma$ is a production then add item $B \to \cdot \gamma$. (basically if B is non terminal add all its prod")

eg 1

$S \to e$ (augmented)
$e \to e+v \mid v$
$v \to x \mid y$

What is closure({$S \to \cdot e$})?

$S \to \cdot e$ ← include item itself
$e \to \cdot e + v$
$e \to \cdot v$
$v \to \cdot x$
$v \to \cdot y$

$S \to \cdot e$ when taken of the form $A \to \alpha \cdot B\beta$ B = e since e is a non terminal we add its prods also with dot

• followed by e but e prod" already in closure so ignore

• followed by v and v is non terminal so add all its prod" $v \to \cdot x$ & $v \to \cdot y$. Since x & y are terminals no prod" added for them

LR(0) automaton (DFA) is made of LR(0) items.

→ Closure of item of start production gives start state of LR(0) automaton. ∴ CLOSURE ({ S → ·e }) is S_0 (state 0)

Note: Basically to obtain closure of an item or set of items first include all these items in the closure. Then for each item check symbol after (·) dot
i) if symbol is terminal do nothing
ii) " " " non terminal add the productions for that non terminal with dot (·) in start.

GOTO FUNCTIONS

→ used to obtain the transitions of a DFA

→ Goto (I, x)
I - set of items
x - grammar symbol (non terminal or terminal)

→ It is the closure of the set of all items [A → α·Xβ] such that [A → α·Xβ] is in I.

→ Goto (I, x) specifies the transition from state for I (i.e state consisting of items I) when encountering input x.

Q GOTO (S_0, e) ? I = S_0, e = X

Soln S_0 = { S → ·e, e → ·e + v, e → ·v, v → ·x, v → ·y } — only these 2 are of the form A → α·Xβ

⇒ GOTO (S_0, e) = CLOSURE ({ S → e·, e → e· + v })
= { S → e·, S → e· + v }
→ add all items in closure initially
→ No symbol after · in S → e·
so do nothing
→ in S → e· + v, + is after · and it is non terminal so do nothing

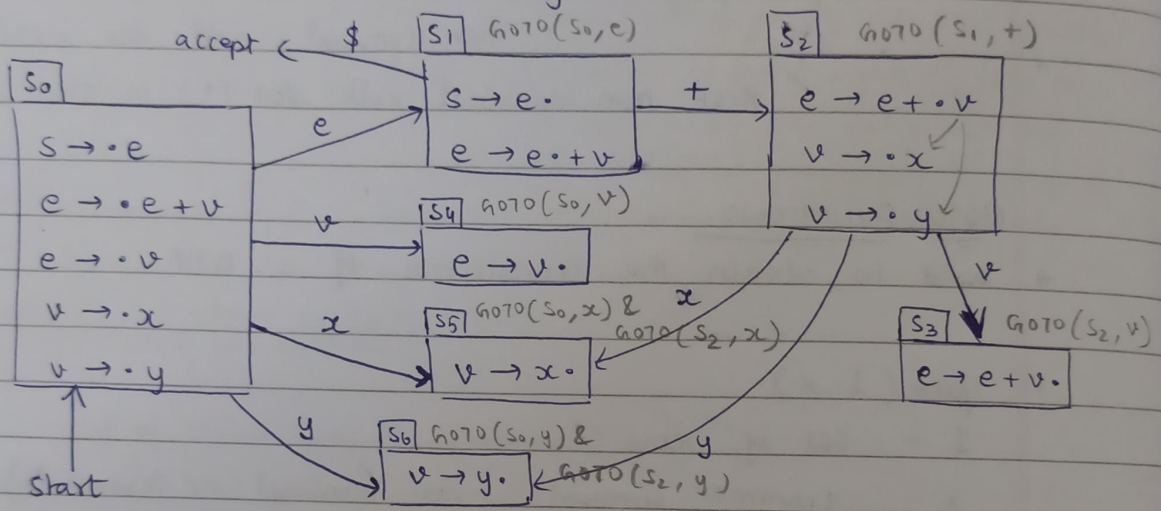
∴ { S → e·, S → e· + v } is the state you go to from S_0 on encountering e. This state can be S_1.

→ similarly we calculate GOTO for every X followed by • recursively until we complete our LR(0) Automaton and obtain all its states and transitions

So, get GOTO(S_0, v) , GOTO(S_0, x) , GOTO(S_0, y)

GOTO(S_1, +) and so on for all states.

∴ LR(0) Automaton for prev Q is



When we are in S_1 we have item S → e • which means we have reached end of start prod^n S → e.

∴ After that if we encounter $ the input is accepted.

To do this only we augmented our grammar

Q E → E+n | n

Sol^n ① Augment the grammar

E' → E

E → E+n

E → n

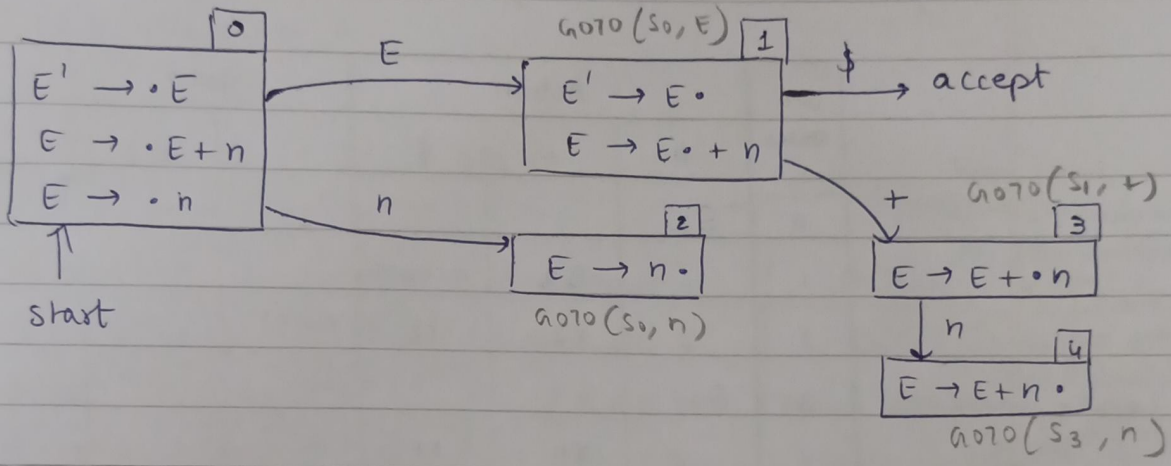
② Start state of LR(0) automaton (DFA)

CLOSURE ({ E' → • E })

↳ start prod^n

$$= \{ \quad E' \to \cdot E$$
$$E \to \cdot E + n$$
$$E \to \cdot n$$
$$\}$$

(3) LR (0) automata



Note: Error recovery in predictive parsing - self study
also read algo & pseudo codes to perform parsing.

CONSTRUCTION OF PARSE TABLE FOR SLR(1)

→ Write states of DFA as rows.
→ Has two parts - action & goto
→ Under action, make columns for all terminals
→ Under goto, make columns for all non-terminals
→ For each state, refer DFA & fill table with:

- shift
- reduce
- accept
- error

} for action columns

- goto entries } for goto columns

→

SLR(1) Parse Table

eg: 0) $E' \rightarrow E$
1) $E \rightarrow E + n$
2) $E \rightarrow n$

} DFA drawn earlier

DFA States	Action			Goto	
	n	+	$	E	E'
0	S2			1	
1		S3	accept		
2		r2	r2		
3	S4				
4		r1	r1		

← consume symbol n Shift (s) the input pointer to next symbol & push n to stack and change state to state 2.

→ no input symbol consumed just change state to state 1

→ 2, 4 are reduce states as next step for them would be reduction i.e dot (•) has reached end of production.

→ 1 is not a reduce state as even though dot reaches end $E \rightarrow n\bullet$, it is a final accepting state

Follow (E) = { $, + }

Note: Shift entries are associated with state numbers.
i.e S2 means shift input pointer & move to state 2 & push symbol to stack

Reduce entries are associated with production numbers
→ i.e r2 means reduce the input to production 2.
↳ here 2) $E \rightarrow n$

∴ in State 2 we know that n is reduced to E as $E \rightarrow n$ ∴ after $E \rightarrow n\bullet$ which is state 2 we check Follow (E) = { $, + }
if any symbol from Follow (E) is encountered then n can be reduced to E i.e r2 (reduce to prod" 2). (∵ input pointer is not shifted it remains in same place)

∴ Done

∴ In general if we are in a reduce state which includes item for prod^n k $A \to BC\bullet$ then if a symbol from FOLLOW(A) is encountered then do rk where r → reduce, k → production no.

i.e BC is reduced to A

→ let i/p string be: n + n + n $

$ to left in bottom up

start state

from parse table Action[0,n] = S2 so, shift n from input to symbols. push 2 to stack & inc↑ input pointer to +

Stack	Symbols	Input	Action
$ 0	shifts n from input to symbols	n + n + n $	Shift
$ 0 2 (because we moved to state 2)	n	+ n + n $	Reduce by E → n → Action[2,+]
$ 0 1	E → n reduced to E	+ n + n $	Shift → Action[1,+] = S3 so shift + to symbols, push 3 to stack & inc↑ input ptr to n

pop 2 from stack after reducing. so we get $0

immediately after reduce always check goto for top of stack and symbol so, GOTO(0,E) = 1, so add 1 to stack. Input pointer is not incremented

$ 0 1 3	E +	n + n $	Shift → Action[3,n] = S4
$ 0 1 3 4	E + n	+ n $	Reduce by E → E+n
~~$ 0 0 0 0~~	~~0~~	~~to pop $~~	

here, length of handle = 3 ∴ on reduce pop 3 items from stack 1, 3, 4 (↳ i.e no. of symbols belonging to RHS of prod^n)

$ 0 1	E (E+n reduced to E)	+ n $	Shift → Action[1,+] = S3
$ 0 1 3	E +	n $	Shift → Action[3,n] = S4
$ 0 1 3 4	E + n	$	Reduce by E → E+n
~~$ 0 0 0~~			
$ 0 1	E	$	accept

↳ pop 3 stack items as symbols = 3

↳ goto(0,E) = 1 so add 1 to stack

GOTO(0,E) = 1

→ In Table Driven parser we had terminals & non terminals in stack but in SLR we have states

→ Initial stack is always $0

→ When shifting, shift input symbol to symbols and add state given by Action entry of parse table to the stack. Also increment input pointer

→ When reducing, pop stack by length of handle (i.e no of symbols getting reduced)
→ i.e not length of total symbols, only those symbols which are getting reduced

→ After reducing, always perform goto on top of stack and top of symbols (i.e rightmost state & rightmost symbol)

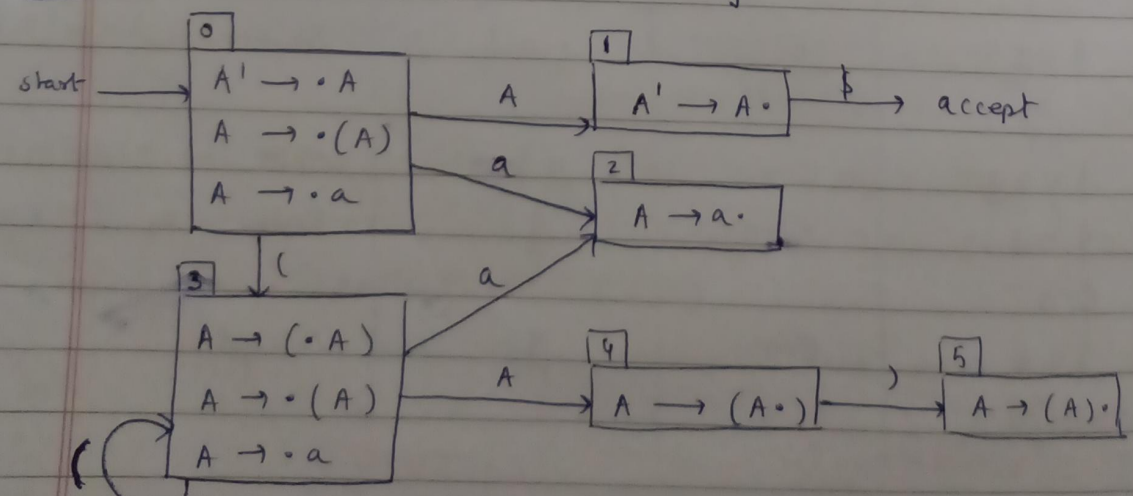
→ Goto & reduce don't consume input (i.e i/p ptr does not increment)

Q $A' \to A$
$A \to (A) \mid a$

Input: ((a))

Sol^n Closure $(\{A' \to \cdot A\}) = \{ A' \to \cdot A$
$A \to \cdot (A)$ → start state
$A \to \cdot a$
$\}$

LR(0) automaton



Goto(3, () = 3 i.e self loop

FOLLOW(A') = { $ }

→ 2, 5 are reduce states → FOLLOW(A) = {), $ }

→ 1 is accept state

0) A' → A
1) A → (A)
2) A → a

Parse Table

States	Action				goto	
	(	a	)	$	A	A'
0	S3	S2			1	
1				accept		
2			r2	r2		
3	S3	S2			4	
4			S5			
5			r1	r1		

A → a•
→ FOLLOW(A) =), $

A → (A)•
FOLLOW(A) =), $

Stack	Symbols	Input	Action
$0		((a)) $	Shift
$03	(	(a)) $	Shift
$033	((	a)) $	Shift
$0332	((a	)) $	Reduce by A → a
			only pop '2' From stack as handle length = 1 i.e only 'a' is getting reduced
$0334	((A	)) $	shift
goto(3,A)			
$03345	((A)	) $	Reduce by A → (A)
			pop 3 states, 345 and replace (A) by A
$034	(A	) $	Shift
goto(3,A)			
$0345	(A)	$	Reduce by A → (A)
$01	A	$	accept
goto(0,A)			

Note: When I say handle length it means length of RHS of prod^n by which we are reducing. eg: Reduce by $A \to (A)$
handle length = length of $(A) = 3$

Now my input: $(a)\$$

Stack	Symbol	Input	Action
$0		(a)$	Shift
$0 3	(	a)$	Shift
$0 3 2	(a	)$	Reduce by $A \to a$
$0 3 4	(A	)$	Shift
$0 3 4 5	(A)	$	Reduce by $A \to (A)$
$0 1	A	$	Accept

pop 2 as handle len = 1
goto (3, A)
pop 3, 4, 5 as handle len = 3
goto (0, A)

Note: for LR Parse you don't need to check for left recursion or left factoring. Bottom up parsers are strong. Only do this for top down.

Q $S \to (S)S \mid \varepsilon$

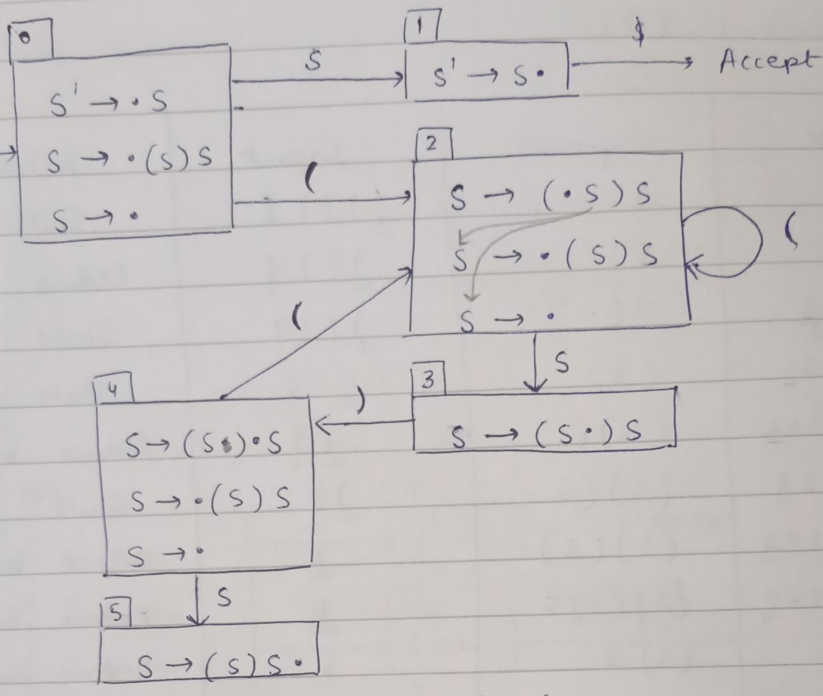
Sol^n ① $S' \to S$ (augmentation)
$S \to (S)S$
$S \to \varepsilon$

start state ② Closure ({ $S' \to \bullet S$ }) = { $S' \to \bullet S$
$S \to \bullet (S)S$
$S \to \bullet$
}

→

LR(0) automaton

③



Parse Table

④ Reduce states : 0, 2, 4, 5 → S → (S) S •

S → •

S → •

S → •

all these have a prod^n with • in the end hence reduce state

Accept state : 1

FOLLOW (S') = { $ }

FOLLOW (S) = {), $ } → means) & $ column will have reduce entries for reduce states containing productions with S as head

① S' → S

② S → (S) S

③ S → ε

States	Action			Goto	
	(	)	$	S	S'
0	S2	r2	r2	1	
1			Accept		
2	S2	r2	r2	3	
3		S4			
4	S2	r2	r2	5	
5		r1	r1		

Basically reduce entries are made in columns with terminals in FOLLOW of head of that prod^n which has dot (•) in the end in that state

Parsing

⑤ Input : ( ) ( )

Stack	Symbol	Input	Action
$0		( ) ( ) $	Shift
$0 2	(	) ( ) $	Reduce by S → ε (handle length = 0 so no popping)
$0 2 3	(S	) ( ) $	Shift
$0 2 3 4	(S)	( ) $	Shift
$0 2 3 4 2	(S) (	) $	Reduce by S → ε
$0 2 3 4 2 3	(S) (S	) $	Shift
$0 2 3 4 2 3 4	(S) (S)	$	Reduce by S → ε (pop 4 states from stack)
$0 2 3 4 2 3 4 5	(S) (S) S	$	Reduce by S → (S) S (pop 4 states)
$0 2 3 4 5	(S) S	$	Reduce by S → (S) S
$0 1	S	$	Accept

2 is not popped as handle length = 0

goto(2,S) → 3

goto(4,S) → 1

(empty is reduced to S)

Q $S \rightarrow Aa \mid bAc \mid dc \mid bda$

$A \rightarrow d$

Sol" ① Augument (here even though S is not appearing on RHS of prod" but still we augument as S has multiple prod", how do we know which prod" was chosen as start) So if we dont augment we'll have multiple accept states.

$S' \rightarrow S$

$S \rightarrow Aa$

$S \rightarrow bAc$

$S \rightarrow dc$

$S \rightarrow bda$

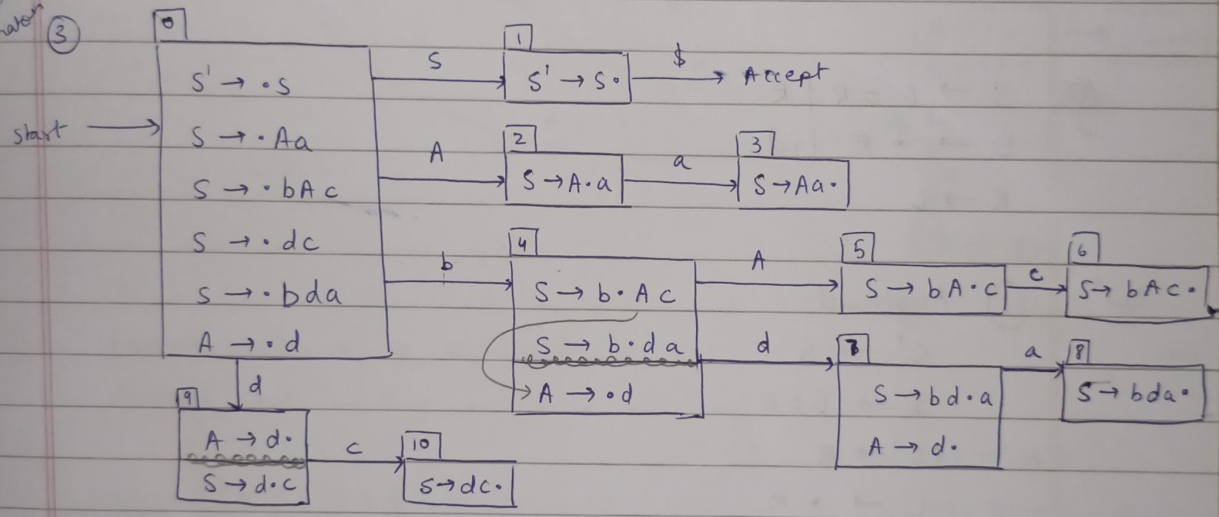
$A \rightarrow d$

Note: if while drawing LL(1) parse table or LR parse table you now have conflict in an entry, don't stop. Still continue with parsing and only if during parsing you have to consult the conflicting entry then stop else keep going

conflict means more than 1 entries in a cell

Start State (2) CLOSURE({ S' → ·S }) = {
S' → ·S
S → ·Aa
S → ·bAc
S → ·dc
S → ·bda
A → ·d
}

LR(0) automaton (3)



(4) SLR(1) Parse Table
because lookahead i.e. follows is being used.

0) S' → S
1) S → Aa
2) S → bAC
3) S → dc
4) S → bda
5) A → d

States	Action					Goto		
	a	b	c	d	$	S	A	S'
0		s4		s9		1	2	
1					accept			
2	s3							
3					r1			
4				s7			5	
5			s6					
6					r2			
7	r5 s8		r5		r5			
8					r4			
9	r5		r5 s10					
10					r3			

Reduce states: 3, 6, 7, 8, 9, 10

Accept state: 1

FOLLOW(S') = {$}
FOLLOW(S) = {$}
FOLLOW(A) = {a, c}

∴ 2 of the cells have shift-reduce conflict

∴ Clearly SLR(1) is not able to completely eliminate shift reduce conflict.

But still SLR(1) parser is stronger than shift reduce parser.

So the given grammar cannot be parsed using SLR(1) parser ∴ we say its not SLR(1) grammar.

Q $S \to L = R \mid R$

$L \to * R \mid id$

$R \to L$

Sol^n ① Augment

$S' \to .S$

$S \to L = R$

$S \to R$

$L \to * R$

$L \to id$

$R \to L$

Start state ② Closure($\{ S' \to \cdot S \}$) $= \{ S' \to \cdot S$

$S \to \cdot L = R$

$S \to \cdot R$

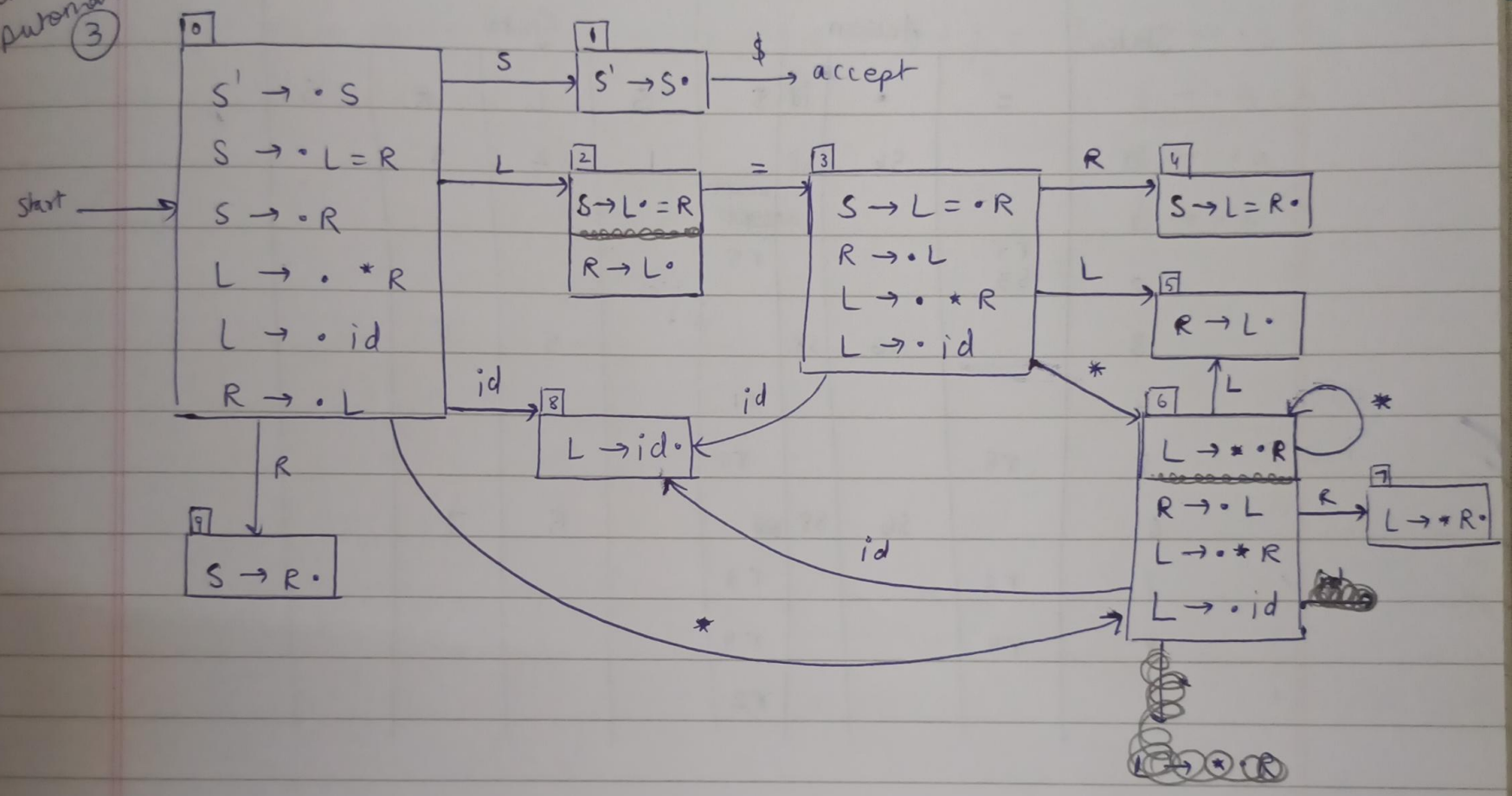
$L \to \cdot * R$

$L \to \cdot id$

$R \to \cdot L$

$\}$

LR(0) automaton (3)



Parse table (4)

Reduce states : 2, 4, 5, 7, 8, 9

Accept state : 1

FOLLOW (S') = { $ }

FOLLOW (S) = { $ }

FOLLOW (L) = { =, $ }

FOLLOW (R) = { =, $ }

0) S' → S

1) S → L = R

2) S → R

3) L → * R

4) L → id

5) R → L

→

States	Action				Goto			
	=	*	id	$	S	L	R	S'
0		S6	S8		1	2	9	
1				accept				
2	r5 S3			r5				
3		S6	S8			5	4	
4				r1				
5	r5			r5				
6		S6	S8			5	7	
7	r3			r3				
8	r4			r4				
9				r2				

∴ This also has shift - reduce conflict on M(2, =)
So, it is not SLR(1) grammar.

Q $S \to BCD$
$B \to a \mid b$
$C \to x \mid y$

get start state of LR(0) automaton

Sol" ~~No need to augment as S only has one prod"~~

CLOSURE ({ $S \to \cdot BCD$ }) =

always just augment

① Augment $S' \to S$
$S \to BCD$
$B \to a$
$B \to b$
$C \to x$
$C \to y$

(2) Start state , CLOSURE ({ S' → · S }) = { S' → · S

S → · B C D

B → · a

B → · b

}

Note: This ⓠ tells us that start state need not have all productions . Like here C → · x & C → · y are not in start state.