Deficulties in Top-down parser

1) Backtracking
2) order of Alternate is matt
3) during failier location of error can not be discovere
4) Left Recursion –

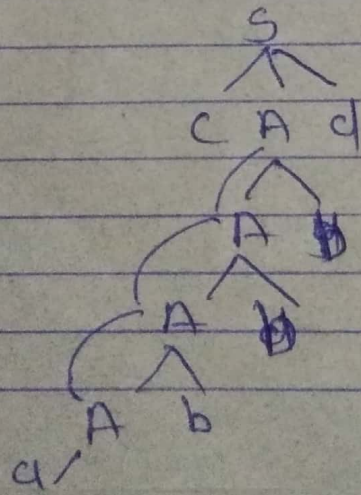
A grammer is left Recursive if it has a no terminal 'A' such that there is a derivation

$$A \overset{+}{\Longrightarrow} A\alpha$$

+ → After some ste

(A tense After some $A\alpha$)

– Left most tree is increase till to wh we stop it. for exam



of this Action func

1) shif
2) shif
3) Re... =
4)

E → E on first position on Right hand expression like

$E \rightarrow E + T \mid T$

Elimenation – if 'A' gives

$A \rightarrow A\alpha \mid \beta$ where β does not begain with 'A' then we can elemenate Left Recursion by Replacing with it

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

Example –

consider the following grammer

$E \rightarrow E + T \mid T$
$T \rightarrow T * f \mid f$
$f \rightarrow (E) \mid id$

– elemenate left Recursion

$E \rightarrow E + T \mid T$
↓ ↓
A A

$E \rightarrow E + T \mid T$ can be Replace by following Rules.

$E \rightarrow E + T \mid T$
↓ ↓ ↓ ↓
A A α β

Recursive decent Parser [Top-Down Parser]

- It is a Parser that uses a set of Recursive Procedures to Recognize it's input with no back-tracking

- the grammer provided for this Parser should not be left Recursive

Consider the grammer

Ex. $E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

The given gramer is left Recursive. so we need to Remove left Recursion first.

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

$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$

terminal { id, ε, (,), +, * }

Non terminal { E, E', T, T', F }

* Left factoring Ex.

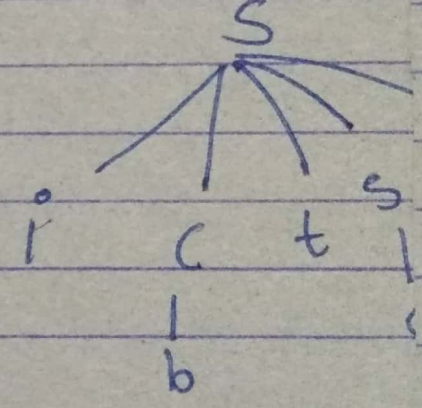
Consider the grammer.

* $S \rightarrow iCtS / iCtSeS / a$

$C \rightarrow b$

$w = ibtaeca$

OR (danalling - else problem)



Left - Factoring -

Few Grammers even though they do not have left Recursion are same times not Suitable for Recursive descent Parser

Example is dangling else problem. to over come this we obtain left factors by the following Rule

If $A \to \alpha\beta / \alpha\gamma$, then replaced by

$$A \to \alpha A' \quad \text{---} (1)$$

$$A' \to \beta / \gamma \quad \text{---} (2)$$

Compare it with our above grammer

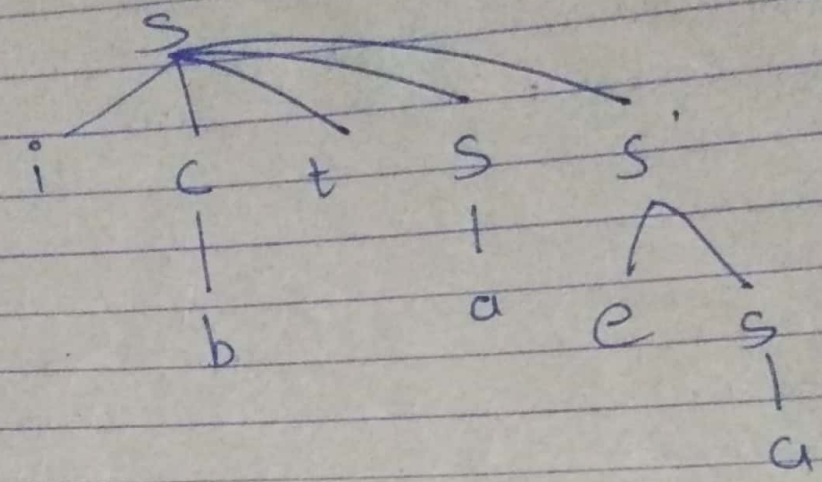
$A = S$, $\alpha = ict\,s$, $\beta = \epsilon$, $\gamma = es$

and substitute in eqn (1) & (2)

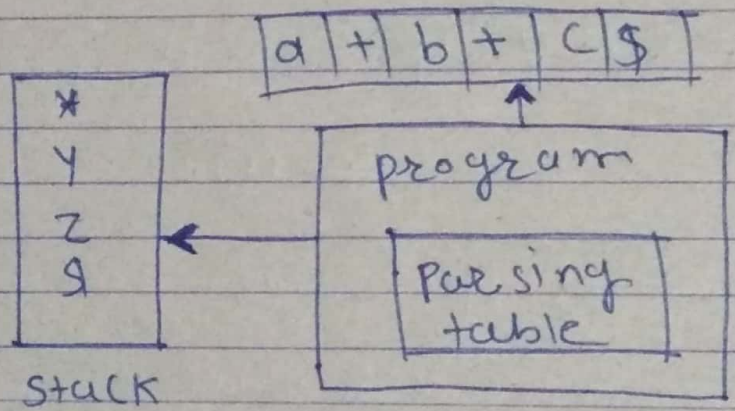
$S \to ict\,s\,s'$
$s' \to \epsilon / es$
$S \to a$
$C \to b$

} new grammer.

w = ibtae d



* Predictive Parser → RDP & TOP-do



Model of Predictive parser.

- A more efficient way of implementing Recursive Descent Parser is to use stack called Activation Records.

- This is done in Predictive Parsers.

* First

In SHORT

1) X $\quad FIRST(X) = \{X\}$

2) $X \rightarrow a\alpha \quad FIRST(X) = \{a\}$

$X \rightarrow \epsilon \quad FIRST(X) = \{\epsilon\}$

3) $X \rightarrow Y_1, Y_2 \ldots Y_K \quad FIRST(Y_j),$

$j = 1 \ldots i-1$

$FIRST(Y_1)$

$j = 1 \ldots K.$

• Rules to find FOLLOW set:

① Put $ in FOLLOW(S) if S is the start symbol.

② If there is a production $A \to \alpha B\beta$ where $\beta \neq \epsilon$ then then $FOLLOW(B) = FIRST(\beta) - \{\epsilon\}$

③ If $A \to \alpha B$ or $A \to B\beta$ where $\beta = \epsilon$ then $FOLLOW(B) = FOLLOW(A)$.

• FOLLOW(E) :-

$FOLLOW(E) = \{\$\}$
by rule ①
$F \to (E)$
Compare with $A \to \alpha B\beta$
$\therefore A = F,\ B = E,\ \alpha = (\ ,\ \beta =)$
As $\beta \neq \epsilon$, apply ② rule
$FOLLOW(B) = FIRST(\beta) - \{\epsilon\}$
$FOLLOW(E) = FIRST()) - \{\epsilon\}$
$= \{)\} - \{\epsilon\}$

- If epsilon FIRST(Y_j) for all $j=1$ ---- k then & add epsilon to FIRST(X).

- Consider the previous grammar & find FIRST & Follow sets.

Ans:-

FIRST():-

Non-terminals - {E, E', T, T', F}

Terminals - {id, +, *, (,)}

(1) Find FIRST set for all terminal symbols by one.

FIRST(id) = {id} FIRST(() = {(}

FIRST(+) = {+} FIRST()) = {)}

FIRST(*) = {*}

(2) By two, Find the FIRST set of non-terminals whose RHS starts with terminals.

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

  FIRST(E') = {+, ε}

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

  FIRST(T') = {*, ε}

- $F \rightarrow id / (E)$

  FIRST(F) = {id, (}

(3) By three, find FIRST set of symbols E & T.

• $E \rightarrow TE'$

FIRST (E) = FIRST (T)

$T \rightarrow FT'$

FIRST (T) = FIRST (F)

FIRST (F) = {id, (}

FIRST (T) = {id, (}

FIRST (E) = {id, (}

• Rules to find FOLLOW set:

① Put $ in FOLLOW (S) if S is the start symbol.

② If there is a production $A \rightarrow \alpha B\beta$ where $\beta \neq \epsilon$ then then FOLLOW (B) = FIRST (β) - {ϵ}

③ If $A \rightarrow \alpha B$ or $A \rightarrow B\beta$ where $\beta = \epsilon$ then FOLLOW (B) = FOLLOW (A).

• FOLLOW (E) :-

FOLLOW (E) = {$}

by rule ①

$F \rightarrow (E)$

Compare with $A \rightarrow \alpha B\beta$

∴ A = F, B = E, α = (, β =)

As $\beta \neq \epsilon$, apply ② rule

FOLLOW (B) = FIRST (β) - {ϵ}

FOLLOW (E) = FIRST ()) - {ϵ}

= {)} - {ϵ}

FOLLOW (E) = {$,)}

• FOLLOW (E') :-

Compare it with $A \rightarrow \alpha B\beta$

~~$A \rightarrow \alpha$~~ $E \rightarrow TE'$

A = E, B = E', α = T, $\beta \neq \epsilon$

As $\beta = \epsilon$, apply rule ③

FOLLOW (B) = FOLLOW (A)

FOLLOW (E') = FOLLOW (E)

FOLLOW (E') = {), $}

• FOLLOW (T) :-

Case 1 :- Compare it with $A \rightarrow \alpha B\beta$

$E \rightarrow TE'$

A = E, B = T, α = ϵ, β = E'

As $\beta \neq \epsilon$, apply ② rule.

FOLLOW (B) = FIRST (β) - {ϵ}

FOLLOW (T) = FIRST (E') - {ϵ}

= {), $} - {$\epsilon$}

FOLLOW (T) = {), $, ϵ}

Case 2 :-

• Follow (T'):-

$T \rightarrow FT'$

Compare it with $A \rightarrow \alpha B \beta$

$A = T$, $B = T'$, $\alpha = F$, $\beta = \epsilon$

As $\beta = \epsilon$, apply rule (3)

Follow (B) = Follow (A)

Follow (T') = Follow (T)

Follow (T') = {), $, + }

• Follow (F) :-

~~$T \rightarrow FT'$~~

~~$A = T$, $B = F$, $\alpha = \epsilon$, $\beta = T'$~~

~~$\beta \neq \epsilon$, apply rule (2)~~

case 1 :- $T' \rightarrow *FT'$

compare it with $A \rightarrow \alpha B \beta$

$A = T'$, $B = F$, $\beta = T'$, $\alpha = *$

As, $\beta \neq \epsilon$, apply rule (2)

Follow (B) = FIRST (β) − (ϵ)

= FIRST (T') − (ϵ)

Follow (F) = { *, ϵ }

case 2 :-

$T' \rightarrow \epsilon$

$A = T'$, $B = F$, $\alpha = \epsilon$, $\beta = \epsilon$

As, $\beta = \epsilon$, apply rule (3)

Follow (B) = Follow (A)

Follow (F) = Follow (T')

Follow (F) = {), $, +, *, ϵ }

* Construction of parsing Table.

(1) If the production has RHS value then find its Follow set

(2) If RHS of the production is ϵ then find its ~~follow~~ Follow set.

* $E \rightarrow TE'$

Row ← FIRST (RHS)

FIRST [TE'] = FIRST (T)

= { id, (} → column.

M [E, { id, (}] = { $E \rightarrow TE'$ }

* $E' \rightarrow +TE'$

FIRST (RHS)

FIRST ($+TE'$) = { + }

M [E', { + }] = { $E' \rightarrow +TE'$ }

* $E' \rightarrow \epsilon$

Follow (E') = {), $ }

M [E', {), $ }] = { $E' \rightarrow \epsilon$ }

2) Goto –

Goto fun^n takes a Stats and Grammer symbols as input and produces some State as output

E.g Consider this Grammer & Parsing table.

Extended → to generate new grammar

Augmented Grammer G —

it is new grammer obtained by adding a new Symbol S' & new production

S' → S

Ex - Find G' for G -

E → E + T ,
E → T

T → T * f
T → f
f → (E)
f → id

G'

E' → E
E → E + T
T → T * f
T → f
f → (E)
f → id

13/9/2019

UNIT 04

Syntax Directed Translation

- Syntax Directed Definitions :-
  It is CFG with attributes & rules

Type of Attributes -

1) Synthesized - Value of non-terminal on LHS is function of RHS values [Bottom-up is used] (LHS value depend on RHS values)

2) Inherited - Value on RHS is defined in terms of LHS [Top-down fashion used]

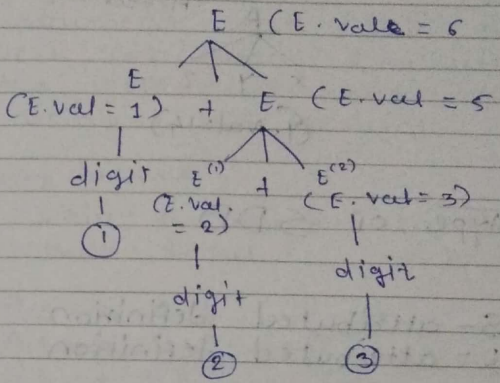
Ex - Production — Semantic Action

$E \rightarrow E + E$ — $\{E.VAL := E^{(1)}.VAL + E^{(2)}.VAL\}$

$E \rightarrow digit$ — $\{E.val := digit\}$

this is example of Synthesized Attribute as the value of expression is define from the Inputs in Bottom-up fashion

Ex 1 - Input — 1+2+3



Ex (2) -

Production — Semantic Action

$A \rightarrow XYZ$ — $\{Y.val := 2 * A.val\}$

It is an Example of inherited Attribute as the value of the RHS Symbol i.e 'Y' depends on value of 'A'

Q.1. Consider the grammer

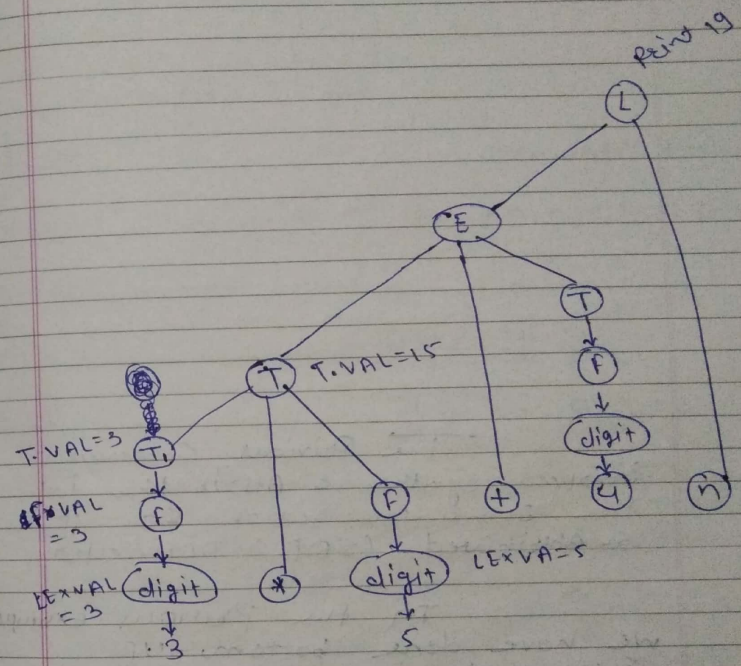
L → En
E → E1 + T
E → T
T → T1 * F
T → F
F → (E)
F → digit.

impliment SDD (SDT) for the following expression

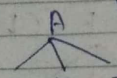
1) 3 * 5 + 4n

2) (3 + 4) * (5 + 6)n

Prodn	Semantic Action
L → En	Print E.val
E → E1 + T	E.val := E1.val + T.val
E → T	E.val := T.val
T → T1 * F	T.val := T1.val * F.val
T → F	T.val := F.val
F → (E)	F.val := E.val
F → digit.	F.val := LEXVAL



Input - 7



(Y.val=14)

* Type of SDD

1) S- attributed definition
2) L- attributed definition

1) S- attributed definition -

An SDD S-attributed if every Attribute is Synthe- -sized

Refer Ex- 1

2) L- attributed definition

-In this each Attribut must be either Synthesized or Inherited

Refer E-2

Example - Implementation of SDD

Implement SDT (SDD) for Desk calculator. Evaluate the the expression 23 * 5 + 4 $ using SDD scheme

1) writing Grammer
- Write a Grammer to describe the Input.

1) S → E $
2) E → E + E
3) E → E * E
4) E → (E)
5) E → I
6) I → I digit { I.val = I(1).val * 10 + Leaval }
7) I → digit { I.val := LEXVAL }

2) Semantic Action
- ADD Semantic Action to the productions of grammer.

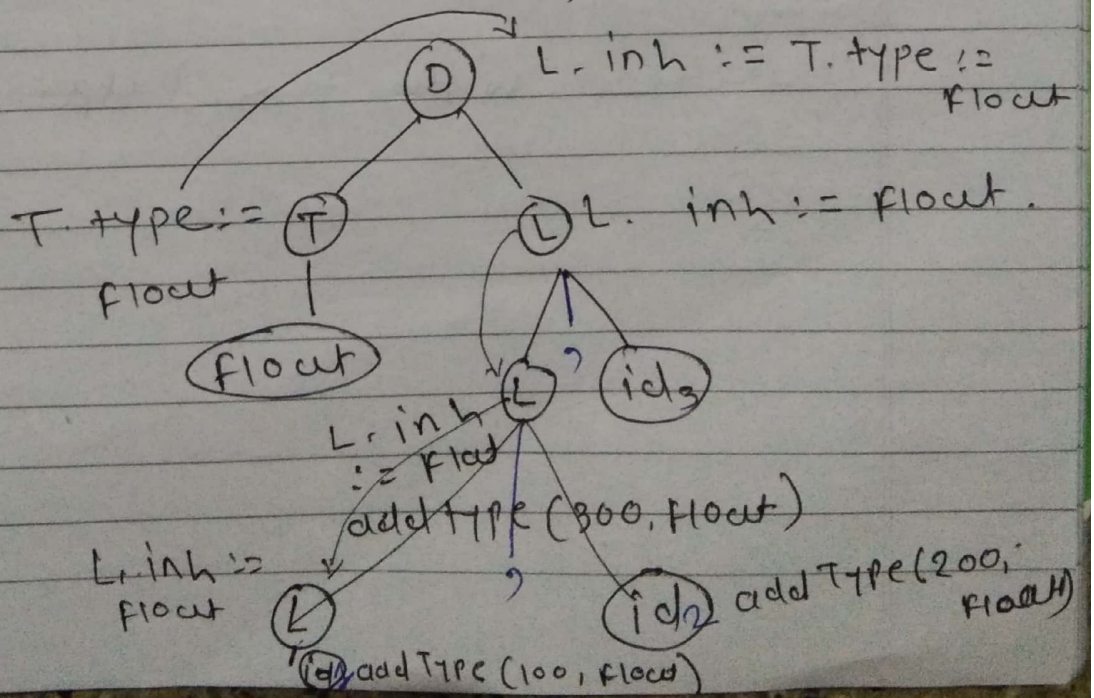
1) { Print E.val }
2) { E.val := E(1).val + E(2).val }
3) { E.val := E(1).val * E(2).val }
4) { E.val := E(1).val }
5) { E.val := I.val }
6) { I.val := I(1) val * 10 + LEXVAL }
7) { I.val := LEXVAL },

* L- Attributed Definition.

Example - Consider the following Grammar used to perform Data type declarations

Production	Semantic Rules
$D \rightarrow TL$	$L.inh := T.Type$
$T \rightarrow int$	synthesize { $T.type := int$
$T \rightarrow float$	$T.type := float$
$L \rightarrow L_1, id$	$L1.inh := L.inh$ addtype(id.entry, L.inh)
$L \rightarrow id$	Add type(id.entry, L.inh)

Eg - float a, b, c;



if - then - else

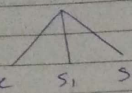


Fig: Syntax tree

* How to Draw Syntax tree

Construction of Syntax tree is similar to Translation of expression in postfix notation

- Use the following functions to create leave & non-leave nodes & Return a pointer

1) Function 1 - Mknode (op, left, Right)

it creates operator node (op) with lebel "op" and two fields to point to left & Right nodes

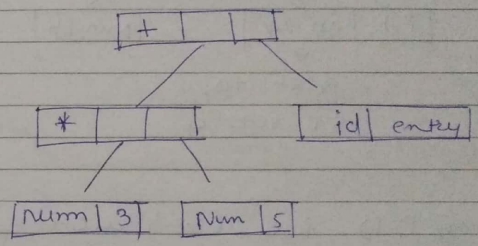
2) Function 2 - mkleaf (id, entry)

it creats node id with label "id" & a pointer to Symbol table as entry

3) function 3 - mkleaf (num, val)

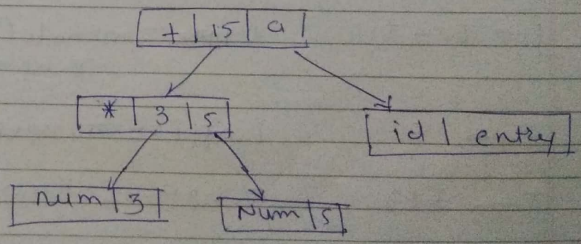
it create a number node with label as "num" & value in "val"

Ex

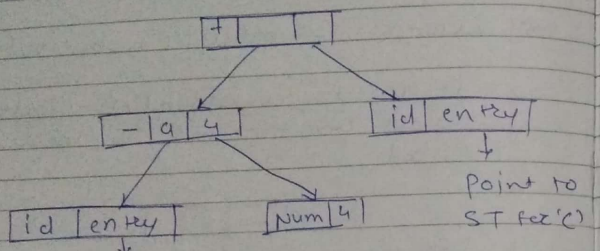


Ex- Create Syntax tree for the expression

3 * 5 + a



2) a - 4 + c



* SDD for constructing syntax trees

Prod^n	
$E \rightarrow E_1 + T$	E.nptr := mknode('+', E_1.nptr, T.nptr)
$E \rightarrow E_1 - T$	E.nptr := mknode('-', E_1.nptr, T.nptr)
$E \rightarrow T$	E.nptr := T.nptr
$T \rightarrow (E)$	T.nptr := E.nptr
$T \rightarrow id$	T.nptr := mkleaf(id, entry)
$T \rightarrow num$	T.nptr := mkleaf(num, val)

Q.1 Create syntax tree for the expression a-4+c using semantic rules.