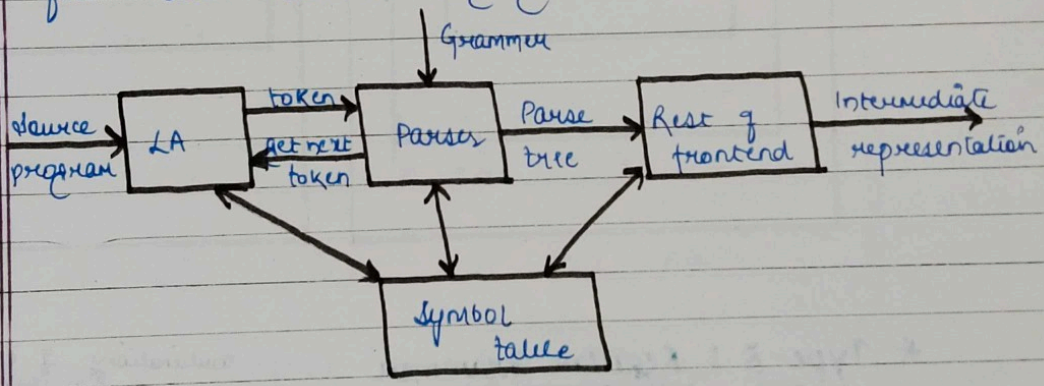
2nd Feb'17
Lecture-15

UNIT-2

Syntax Analysis

Role of parser :

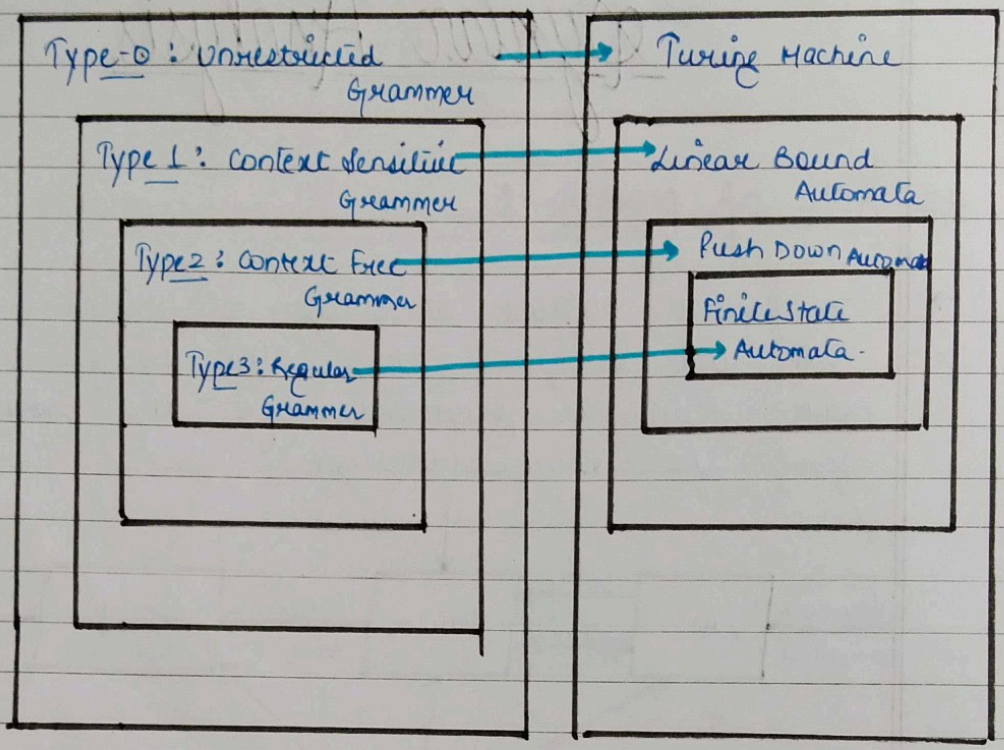
→ The parser obtains a string of tokens from the LA and verifies that the string of token names can be generated by the ~~grammar~~ grammar for the source language.



→ Infact, the parse tree can not be constructed explicitly since checking & translation actions can be done with parsing.

3rd Feb'17
Lecture-16

Classification of Grammers :



* Type-3 : Regular Grammer

combination of elements present in Σ given alphabet

$x \in \Sigma^*$

$B \in V_N$ → non-terminal

Syntax: right linear

$A \rightarrow xB$

$A \rightarrow x$

$A \rightarrow Bx$ ← left linear

$A \rightarrow x$

- A regular grammer is one that is either right or left linear.
- A linear grammer is that grammer in which almost 1 variable can occur at on the right side of any production without

restriction on the position of this grammer.

- Example : $A \to Ba \mid a$
  $A \to aB \mid a$

It is not RG because it is ~~neither~~ both left and right linear which is not allowed in RG. In RG, it can either be left or right linear but not both.

It is a linear grammer as it is both left & right linear.

$A \to asb$
$A \to ab$

It is linear but not RG.

* Type 2 : Context Free Grammer

Syntax :

$$A \to \alpha$$

Can be both terminal or non-terminal

$\alpha \in V^*$

$A \in V_N$ → non-terminal only

* Type 1 : Context Sensitive Grammer

Syntax :

$$u \to v \quad \text{where} : |u| \leq |v|$$

$$\alpha A \beta \to \alpha \gamma \beta$$

$\alpha, \beta \in V^*$

$\gamma \in V^+$

$A \in V_N$

Example :

$$\underset{\alpha}{aA}\ \underset{A}{A}\ \underset{\beta}{Ab} \rightarrow \underset{\alpha}{aA}\ \underset{\gamma}{Ab}\ \underset{\beta}{Ab}$$

* Type 0 : Unrestricted Grammar

Syntax : $\boxed{U \rightarrow v}$

$U \in V^* V_N V^*$

$v \in V^*$

Q. Find the highest type number which can be applied to the following grammar :

1.) $S \rightarrow Aa$
$A \rightarrow C | Ba$
$B \rightarrow abc$

2.) $S \rightarrow ASB | d$
$A \rightarrow a | A$

3) $S \rightarrow as | ab$

1.) Type 3 2.) Type 2 3.) Type 3

Q. $A \rightarrow A\$B | B$
$B \rightarrow B\#C | C$
$C \rightarrow C@D | D$
$D \rightarrow d$

Precedence = ?

$, # & @ are left recursive

@ > # > $ → ~~predesen~~ precedence

@ > @ → left associative

or

> # → $ > $

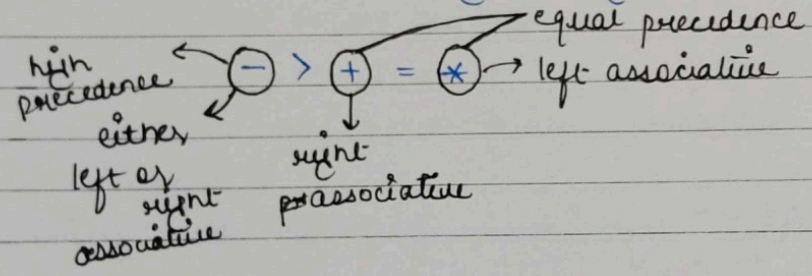
Q1

$E \rightarrow E * F$
$\rightarrow F + E$
$\rightarrow F$
$F \rightarrow F - F \mid id$

$- > + > *$
→ precedence :
$* > *$ or $+ > +$ or $- > -$
precedence.
* & - are left recursive
+ is right recursive

4th Feb'17
Lecture-17

'F' is both left & right recursive.
so, it is ambiguous grammar.



Left Recursion :

→ A grammar is left recursive if :

"It has a non-terminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α."

→ The left recursive pair of productions $A \rightarrow A\alpha \mid \beta$ (α, β are the combination of terminals & non-terminals) can be replaced by the non-left recursive production :

Must be same

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

Date: __/__/__
Page:

Example :

$E \xrightarrow{A} E \xrightarrow{\alpha} +T \mid \xrightarrow{\beta} T$ > left recursive

$T \xrightarrow{A} T \xrightarrow{\alpha} *F \mid \xrightarrow{\beta} F$

$F \rightarrow (E) \mid id$

Replace left recursive with non left recursive production :

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

→ Immediate left recursion can be eliminated by the following technique which works for any number of A productions.

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \cdots\cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots\cdots \mid \beta_n$

it will be replaced by :

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

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \cdots\cdots \mid \alpha_m A' \mid \epsilon$

Example : * $\xrightarrow{A} S \rightarrow \underline{S} \xrightarrow{A} \underline{0} \xrightarrow{\alpha} \underline{S} \xrightarrow{\alpha} \underline{1} S \mid \xrightarrow{\beta} \underline{0} \underline{1}$

$A \rightarrow S \rightarrow 01S'$

$S' \rightarrow 0S1SS' \mid \epsilon$

* $S \rightarrow (L) | x$

$L \rightarrow L, S | S$

$L \rightarrow SL'$

$L' \rightarrow ,SL' | \epsilon$

$S \rightarrow (L) | x$

* $S \rightarrow Aa | b$

$A \rightarrow Ac | Sd | \epsilon$

We will use the given production as:

$S \rightarrow Aa | b$

$A \rightarrow Ac | Aad | bd | \epsilon$

$A \rightarrow bdA' | A'$

$A' \rightarrow cA' | adA' | \epsilon$

$S \rightarrow Aa | b$

Left_factoring :

If :

$$A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \cdots | \alpha\beta_n | \gamma$$

then it is replaced by :

$$A \rightarrow \alpha A' | \gamma$$
$$A' \rightarrow \beta_1 | \beta_2 | \cdots | \beta_n$$

→ left factored

Example : $S \rightarrow iEtS | iEtSeS | a$ ($\beta_1 = \epsilon$, $\beta_2 = eS$, $\alpha = iEtS$)

$E \rightarrow b$

9th Feb'17
Lecture-18

$S \rightarrow iEtSS' \mid a$

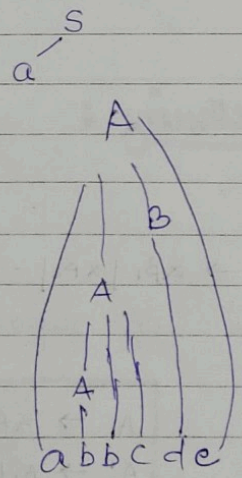
$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

~~9th Feb'17~~
~~Lecture-18~~

~~MID-2~~

$S \rightarrow abbcde$



MID-2

Date: __/__/__
Page:

9th Feb'17
Lecture-18

Unit - II.

PARSER.

Ambiguous X

Parsers

- Top Down Parser (TDP)
  - Top Down Parser with Full Backtracking (try all possibilities)
  - Top Down Parser without Backtracking (exactly 1 is correct)
    - Recursive Descent Parser
    - Non-recursive Descent Parser (LL(1) Parser)
      - left to right
      - left most derivation
      - look ahead symbol
    - Left recursion X, NDX ⇒ Predictive parser
- Bottom up Parser (SR parsers) — SR Parser; SR = Shift Reduce
  - Operator Precedence Parser
  - LR Parser — left to right, RMD in reverse
    - LR(0) — Simple
    - SLR(1) — Simple
    - LALR(1) — Look-ahead
    - CLR (1) — Canonical

* Working of top down & bottom up parsers :

0.) Top down parser :

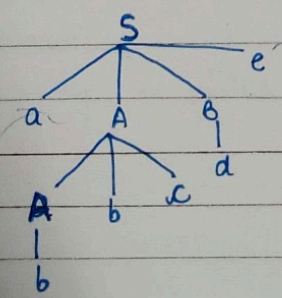
$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

w : abbcde

- what to use
- LMD



$S \Rightarrow aABe$

$\Rightarrow aAbcBe$

$\Rightarrow abbcBe$

$\Rightarrow abbcde$

b.) Bottom up approach :

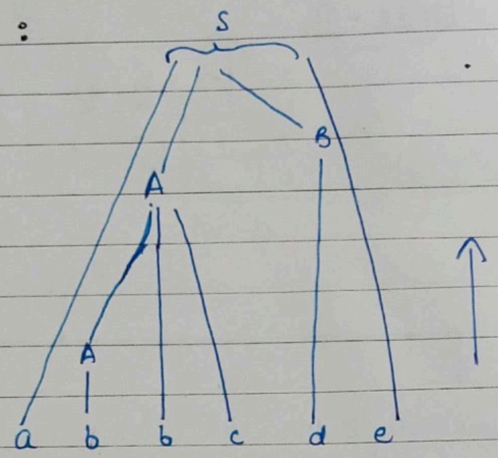
main task:
when to reduce
→ handle
reverse

$S \Rightarrow aABe$

$\Rightarrow aAde$

$\Rightarrow aAbcde$ ↑

$S \Rightarrow abbcde$



10th Feb'17
Lecture-19

I. Top Down Parser :

1. Top Down Parser with backtracking

2. Top Down Parser without backtracking :

i.) Recursive Descent Parser :

A recursive descent parsing program consists of a set of procedures, one for each non-terminal.

Execution begins with the procedure for the start symbol which halts & ~~announces~~ announces success if its procedure body scans the entire input string.

Example : $E \rightarrow iE'$

$E' \rightarrow +iE' | \epsilon$

```
E()
{
 if(l = 'i')
 {
   match('i');
   E'();
 }
}
E'()
{
 if(l = '+')
 {
   match('+');
   match('i');
   E'();
 }
 else
  return;
}
match(char t)
{
 if (l==t)
   l = getchar();
else
  printf("error");
}
```

```
main()
{
  E();
  if (l == '$')
    printf("Parsing done");
}
```

Date: __/__/__
Page:

11th Feb'17
Lecture-20

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

```
E()
{
  if(l=='+')
  E();
```

It is left recursive.
∴ removing left recursion.

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

```
E()
{
  T();
  E'();
```

```
T'()
{
  if(l=='*')
  {
    match('*');
    F();
```

```
T''()
{
  F();
  T();
```

```
F()
{
  if(l=='(')
  {
    match('(');
    E();
    match(')');
  }
else
```

```
E'()
{
  if(l=='+')
  {
    match('+');
    T();
```

11th Feb'17

Lecture-20

Method-2

```
Procedure E()
begin
    T();
    E'();
END

Procedure E'()
begin
    if Input Symbol= '+';
        ADVANCE();
        T();
        E'();
  End

Procedure T()
begin
    F();
    T'();
End

Procedure T'()
begin
    if Input Symbol= '*';
        ADVANCE ();
        F();
        T'();
End

Procedure F()
begin
    if Input Symbol= 'id'
        ADVANCE ();
```

Else if i/p Symbol = 'c' ii)

```
ADVANCE();

i/p Symbol = ')';
ADVANCE();
End
```

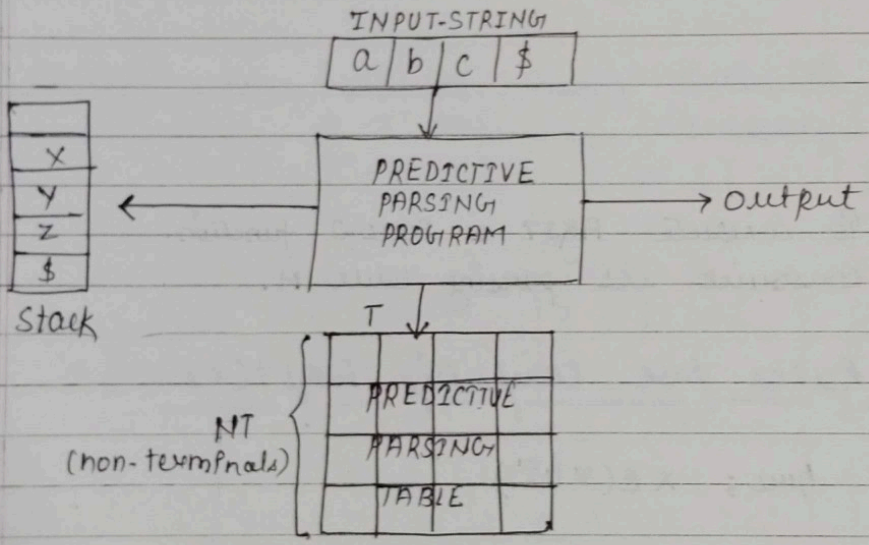
id + id * id

Procedure	O/p String
E()	id + id * id
T()	id + id * id
F()	id + id * id
ADVANCE()	id + id * id
T'()	id + id * id
E'()	id + id * id
ADVANCE()	id + id * id
T()	id + id * id
F()	id + id * id
ADVANCE()	id + id * id
T'()	id + id * id
ADVANCE()	id + id * id
F()	id + id * id
ADVANCE()	

ii) **Predictive - Parsing :-** [Non-Recursive Descent / LL(1) :

It is a special case of Recursive Descent parsing where no backtracking is required.

It should be Non-left-Recursive, unambiguous.



The predictive parser is an I/p Buffer & O/p string.

Input Buffer:

It consists of string to be pass followed by '$' to indicate the end of the I/p string.

Stack:

It consists a sequence of Grammer Symbol

Preceded by '$' to indicate the Bottom of Stack. Initially the stack contains the start symbol on top of dollar.

Parsing Table:

It is a 2D-array M[A,a] where A is a Non-terminal and 'a' is a Terminal.

14th Feb'17
Lecture-21.

Predictive parser or LL1 parser requires 2 phases:

1. To compute FIRST & FOLLOW function.
2. Construct LL1 parsing table M.

* Rules for computing FIRST(X):

here; $X \in (V \cup \Sigma)$

Rule-1:
If X is a terminal then FIRST(X) = {X}

Rule 2:
If $X \to \epsilon$ then add ϵ to FIRST(X).

Ex: $A \to a \mid \epsilon$

FIRST(a) = {a}

FIRST(A) = {a, ε}

Rule 3:

If X is a non-terminal & $X \to Y_1 Y_2 \cdots Y_k$ for some $k \geq 1$ then place 'a' in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), FIRST(Y_2) --- FIRST(Y_{i-1}): $Y_i \cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$.

If ϵ is FIRST(Y_j) for all $j = 1$ to k then, add ϵ to FIRST(X).

Example:

	FIRST	FOLLOW
$S \to Bb \mid Cd$	{a, b, c, d}	{$}
$B \to aB \mid \epsilon$	{a, ε}	{b}
$C \to cC \mid \epsilon$	{c, ε}	{d}

(Rule 3) follow of E' will be follow of E

	FIRST	FOLLOW
$E \to TE'$	{ (, id }	{$,) }
$E' \to +TE' \mid \epsilon$	{+, ε}	{$,)}
$T \to FT'$	{ (, id}	{+, $,)}
$T' \to *FT' \mid \epsilon$	{*, ε}	{+, $,)}
$F \to (E) \mid id$	{(, id}	{*, +, $,)}

	FIRST	FOLLOW
$S \to ACB \mid CbB \mid Ba$	{d, g, h, b, a, ε}	{$}
$A \to da \mid BC$	{d, g, h, ε}	{h, g, $}
$B \to g \mid \epsilon$	{g, ε}	{$, a, h, g}
$C \to h \mid \epsilon$	{h, ε}	{g, $, b, h}

15th Feb'17
Lecture-22.

* Rules for computing FOLLOW(A):

To compute FOLLOW(A) for all non-terminals A apply the following rules until nothing can be added to any FOLLOW set.

Rule 1:
Place $ in FOLLOW(S) where S is the start symbol and $ is the i/p right end marker.

Rule 2:
If there is a production $A \rightarrow \alpha B \beta$ then everything in FIRST(β) except ϵ is in FOLLOW(B).

Rule 3:
If there is a production $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ then everything in FOLLOW(A) is in FOLLOW(B).

16th Feb'17
Lecture-23

Algorithm for construction of LL(1) parsing table: (Predictive)

Input : Grammer G

Output : Parsing table M

Method : For each production $A \rightarrow \alpha$ do the following :

Step-1 :

for each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a].

Step-2 :

If ϵ is in FIRST(α) then, for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, b].

If ϵ is in FIRST(α) and \$ is in FOLLOW(A) then, add $A \rightarrow \alpha$ to M[A, \$] as well.

Example :

	FIRST	FOLLOW
$E \rightarrow TE'$	{(, id}	{\$,)}
$E' \rightarrow +TE' / \epsilon$	{+, ϵ}	{\$,)}
$T \rightarrow FT'$	{(, id}	{+, \$,)}
$T' \rightarrow *FT' / \epsilon$	{*, ϵ}	{+, \$,)}
$F \rightarrow (E) / id$	{(, id}	{*, +, \$,)}

Parsing table :

LL(1) Grammer

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

⇒ E → TE'

FIRST(TE') = FIRST(T)
= {(, id}

Add E → TE' to
M[A, (] & M[A, id]

⇒ E' → +TE' | ϵ

FIRST(+TE') = {+}

add E' → +TE' to
M[E', +]

E' → ϵ

Follow(E') =
{$,)}

add E' → ϵ to
M[E', $] & M[E',)]

⇒ T → FT'

FIRST(FT') = FIRST(F)
= {(, id}

add T → FT' to
M[T, (] & M[T, id]

⇒ T' → *FT' | ϵ

FIRST(*FT') = {*}

add T' → FT' to
M[T', *]

T' → ϵ

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

Add T' → ϵ to
M[T', +], M[T', $],
M[T',)]

⇒ F → (E) | id

FIRST((E)) = {(}

add F → (E) to
M[F, (]

F → id

FIRST(F) = {id}

add F → id to
M[F, id]

* Note :

→ For a grammer to be LL(1) there must be a single entry in each cell. If there are more than one entry in a cell then it is not LL(1) grammer.

→ Left recursive or ambiguous grammers are not LL(1) grammer. as parsing table will have a multiple entry in any cell.

Q. Consider a grammer :

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

Compute : a.) FIRST & FOLLOW
b.) Construct predictive parsing table
c.) is this grammer LL(1) or not.

a.

	FIRST	FOLLOW
$S \rightarrow iEtSS' \mid a$	$\{i, a\}$	$\{\$, e\}$
$S' \rightarrow eS \mid \epsilon$	$\{e, \epsilon\}$	$\{\$, e\}$
$E \rightarrow b$	$\{b\}$	$\{t\}$

b.

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

$S \rightarrow iEtSS' \mid a$

$S \rightarrow iEtSS'$ $\quad$ $S \rightarrow a$

FIRST(iEtSS') = {i} $\quad$ FIRST(a) = {a}

add $S \rightarrow iEtSS'$ to M[S, i] $\quad$ add $S \rightarrow a$ to M[S, a]

$S' \rightarrow eS \mid \epsilon$

$S' \rightarrow eS$

FIRST(eS) = {e}

add $S' \rightarrow eS$ to M[S', e]

$S' \rightarrow \epsilon$

FOLLOW(S') = {$, e}

add $S' \rightarrow \epsilon$ to M[S', $] & M[S', e]

$E \rightarrow b$

FIRST(b) = {b}

add $E \rightarrow b$ to M[E, b]

6c) It is not LL(1) grammer as $S' \rightarrow eS$ & $S' \rightarrow \epsilon$ are present in the same cell.

Algorithm for predictive parser :

Input : A string w and a parsing table M for grammer G.

Output : If w is in L(G), a leftmost derivation of w otherwise an error indication.

Method : Initially the parser is in a configuration with w $ in the input buffer and the start symbol S of G on top of stack above $.

Let, a be the first symbol of w;
Let, x be the top stack symbol;
while (x ≠ $)
{
    if (x = a), pop the stack and let a be the
        next symbol of w;
    else if (x is a terminal)
        error();
    else if (M[x, a] is an error entry)
        error();
    else if (M[x, a] = x → y1y2 ---yk)
    {
        output the production x → y1y2--yk;
        pop the stack;
        push yk, yk-1, --- y1 onto the stack, with y1
        on top;
    }
    let x be the top stack symbol;
}

Example :

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

parsing table made.

MATCH	STACK	INPUT	ACTION
	E$	id+id*id$	
	TE'$	id+id*id$	E→TE'
	FT'E'$	id+id*id$	T→FT'
	idT'E'$	id+id*id$	F→id
id	T'E'$	+id*id$	match id
id	E'$	+id*id$	T'→ε
id	+TE'$	+id*id$	E'→+TE'
id+	TE'$	id*id$	match +
id+	FT'E'$	id*id$	T→FT'
id+	idT'E'$	id*id$	F→id
id+id	T'E'$	*id$	match id
id+id	*FT'E'$	*id$	T'→*FT'
id+id*	FT'E'$	id$	match *
id+id*	idT'E'$	id$	F→id
id+id*id	T'E'$	$	match id

MATCH	STACK	INPUT	ACTION
id+id*id	E'	$	$T' \to \epsilon$
id+id*id	$	$	$E' \to \epsilon$
id+id*id$			match $

Q.

		FIRST	FOLLOW
	$S' \to S\#$	{a, b}	{$}
	$S \to ABC$	{a, b}	{#}
	$A \to a \mid bbD$	{a, b}	{a, b}
w = abc#ab	$B \to a \mid \epsilon$	{a, ε}	{b}
	$C \to b \mid \epsilon$	{b, ε}	
	$D \to c \mid \epsilon$	{c, ε}	

LL(1) top down parsing table

21st Feb '17
Lecture 24

LL(1) Grammer:

→ Predictive parsers can be constructed for the class of grammer called LL(1). The 1st "L" stands for scanning the i/p from left to right. The 2nd "L" stands for producing a left most derivation (LMD) and "1" stands for using 1 i/p symbol of look ahead of at each step to make parsing action decision.

→ No left recursive, left factor & ambiguous grammer can be LL(1).

→ A grammer G can be LL(1) iff whenever $A \to \alpha | \beta$ are α distinct production of G the following conditions hold:

- for no terminal a both α & β derive strings with a.
- atmost one of α & β can derive the empty string.
- if $\beta \overset{*}{\Rightarrow} \epsilon$ then α does not derive any string beginning with a terminal in FOLLOW(A), likewise if $\alpha \overset{*}{\Rightarrow} \epsilon$ then β does not derive any string beginning with a

terminal in FOLLOW(A).

Example :

$S \to aSbS$	$\{a\}$	Not LL(1)
$\mid bSaS$	$\{b\}$	
$\mid \epsilon$	$\{\$, a, b\}$	

	FIRST	FOLLOW	
$S \to aABb$	$\{a\}$	$\{\$\}$	
$A \to c \mid \epsilon$	$\{c, \epsilon\}$	$\{d, b\}$	LL(1)
$B \to d \mid \epsilon$	$\{d, \epsilon\}$	$\{b\}$	

	FIRST	FOLLOW	
$S \to A \mid a$	$\{a\}$	$\{\$\}$	Not LL(1)
$A \to a$	$\{a\}$	$\$$	

	a	b	c	$
S	S→aB		S→ε	
B		S→bC	B→ε	
C			S→cS	C→ε

	FIRST	FOLLOW	
$S \to aB \mid \epsilon$	$\{a, \epsilon\}$	$\{\$\}$	
$B \to bC \mid \epsilon$	$\{b, \epsilon\}$	$\{\$\}$	LL(1)
$C \to cS \mid \epsilon$	$\{c, \epsilon\}$	$\{\$\}$	

	FIRST	FOLLOW	
$S \to AB$	$\{a, b, \epsilon\}$	$\{\$\}$	
$A \to a \mid \epsilon$	$\{a\}$	$\{b, \$\}$	LL(1)
$B \to b \mid \epsilon$	$\{b\}$	$\{\$\}$	

Q. Consider a grammar & show that grammar is LL(1) or not.

exp → term exp1
exp1 → opl term exp1 | ε
term opl → factor term1

	FIRST	FOLLOW
exp → term expl	{ (, id }	{ $,) }
expl → opl term expl | ϵ	{ +, -, ϵ }	{ $,) }
term → factor termL	{ (, id }	{ +, -, $,) }
termL → op2 factor termL | ϵ	{ *, ϵ }	{ +, -, $,) }
factor → (exp) | id	{ (, id }	{ *, +, -, $,) }
opl → + | -	{ +, - }	~~{ *, +, -, $,) }~~ ((, id)
op2 → *	{ * }	~~{ +, -, $,) }~~ (, id)

	(	)	id	+	-	*	$
exp							
expl							
term							
termL							
factor							
opl							
op2							

2

S → I | other
I → if (exp) { SE
E → else S | ϵ
exp → 0 | 1

3

S → A | B
A → ab
B → ac

22nd Feb'17
Lecture-25

II. Bottom up parsing :

It corresponds to the construction of a parse tree for an input string beginning at the leaves (bottom) & working up towards the root (top).

* Reductions :

Bottom up parsing as a process of reducing a string to the start symbol of grammar. At each reduction step a specific substring matching the body of a production is replaced by the non-terminal at the head of that production.

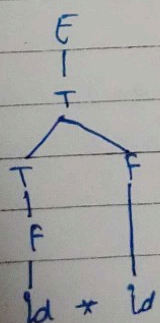
⇒ Handle Pruning :

Handle is a substring that matches the body of a production & whose reduction represents one step along the reverse of RMD (right most derivation).

Example :

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

w: id * id



handle : id
F
id
T*F
T

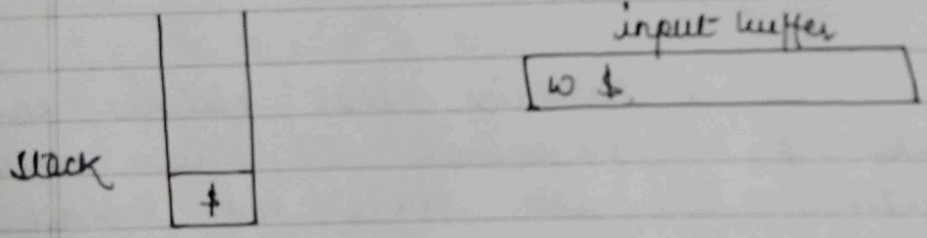
$E \rightarrow T$
$\rightarrow T*F$
$\rightarrow T*id$
$\rightarrow F*id$
$\rightarrow id*id$

Right sequential form	Handle	Action
id * id	id	F → id
F * id	F	T → F
T * id	T	E → T
E * id	id	F → id
T * F	T * F	T → T * F
T	T	E → T
E		

1 #. Operator Precedence

⇒ Shift reduce parsing :

It is a bottom up parsing in which a stack holds grammar symbol & an input buffer that holds the string to be parsed.

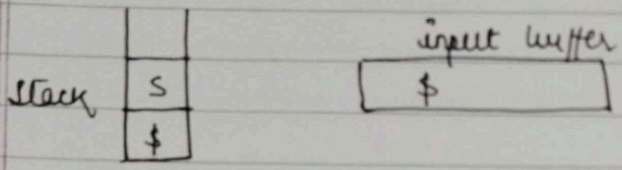


The initial configuration of shift reduce parsing (drawn above).

~~after success~~

The parser repeats the cycle until it has detected the error or until contains the

Start symbol & the input is empty.



=: Possible actions :

There are 4 possible actions that a shift reduce parser can make :

1. Shift :
   In this action, the next symbol is shifted on top of stack.

2. Reduce :
   Handle that appears on the top of stack is replaced with appropriate non-terminal.

3. Accept :
   Parser announces successful completion of parsing.

4. Error :
   A situation in which parser can not either shift or reduce the symbols & also can not even perform the accept action.

* Note :

- If the incoming operator has more priority than stack operator then perform shift.
- If the ~~stack~~ incoming operator has same or less priority then the priority of ~~incoming~~ stack operator then perform reduce.

Example : Parse the following string :

$S \to (L) \mid a$ $\quad (a,(a,a))$ $\quad S \to (L) \mid a$

$L \to L,S \mid S$ $\quad (\to L,a) \mid a$

using shift reduce parser.

priority : a ( ) ,

Stack	I/p buffer	Parsing action
$	(a,(a,a))$	shift
$(	a,(a,a))$	shift
$(a	,(a,a))$	reduce S→a
$(S	,(a,a))$	reduce L→S
$(L	,(a,a))$	shift
$(L,	(a,a))$	shift
$(L,(	a,a))$	shift
$(L,(a	,a))$	reduce S→a
$(L,(S	,a))$	reduce $L→S
$(L,(L	,a))$	shift
$(L,(L,	a))$	shift

Stack	I/P buffer	Parsing Action
$(L,(L,a	))$	Reduce S → a
$(L,(L,S	))$	Reduce L → L,S
~~$(L,(L,L~~	~~))$~~	~~Shift~~
~~$(L,(L,L)~~	~~)$~~	~~Reduce~~

$(L,(L	))$	Shift
$(L,(L)	)$	Reduce S → (L)
$(L,S	)$	Reduce L → L,S
$(L	)$	Shift
$(L)	$	Reduce S → (L)
$S	$	

Q.1 S → TL;
T → int | float
L → L, id | id

w : int id, id;

Q.2 E → E - E
E → E * E
E → id

w : id - id * id

Q.3

Q.1. $S \rightarrow TL;$ w: int id, id;

$T \rightarrow int \mid float$

$L \rightarrow L, id \mid id$

int float id , ;

Stack	I/p buffer	parsing action
$	int id, id; $	shift
$ int	id, id; $	reduce T→int
$ T	id, id; $	shift
$ T id	, id; $	reduce L→id
$ TL	, id; $	shift
$ TL,	id; $	shift
$ TL, id	; $	reduce L→L, id
$ TL	; $	shift
$ TL;	$	reduce S→TL;
$ S	$	

Q.2 $E \rightarrow E-E$

$E \rightarrow E*E$

$E \rightarrow id$

id - id * id

id * -

Stack	I/p buffer	parsing action
$	id - id * id $	shift
$ id	- id * id $	reduce E→id
$ E	- id * id $	shift
$ E -	id * id $	shift
$ E - id	* id $	reduce E→id
$ E - E	* id $	reduce E→E-E
$ E	* id $	shift
$ E *	id $	shift
$ E * id	$	reduce E→id

$ E - E *
$ E - E * id
$ E - E * E
$ E - E
$ E

$ E*E $ reduce E → E*E

$ E $

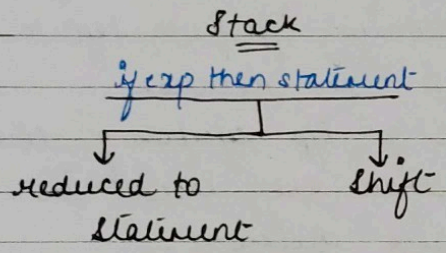
28th Feb'17 Lecture-26

Conflicts in shift reduce parsing:

1.) Shift reduce conflict:

Parser cannot decide whether to shift or reduce.

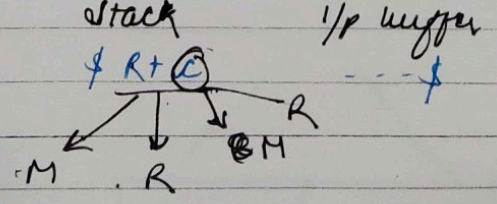
Example: statement → if exp then statement
| if exp then statement else statement
| other



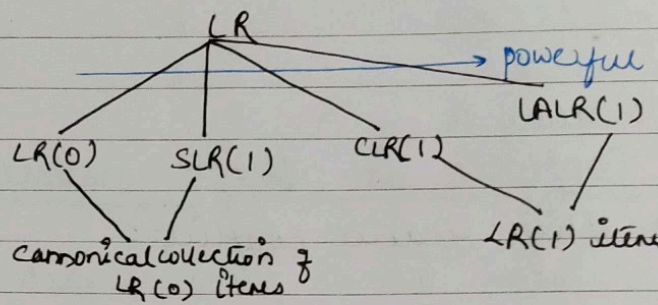
2.) Reduce Reduce conflict:

In this conflict, parser can not decide which production rule should be used to reduce the current handle.

Example: M → R+R | R+C | C
R → R+C | C



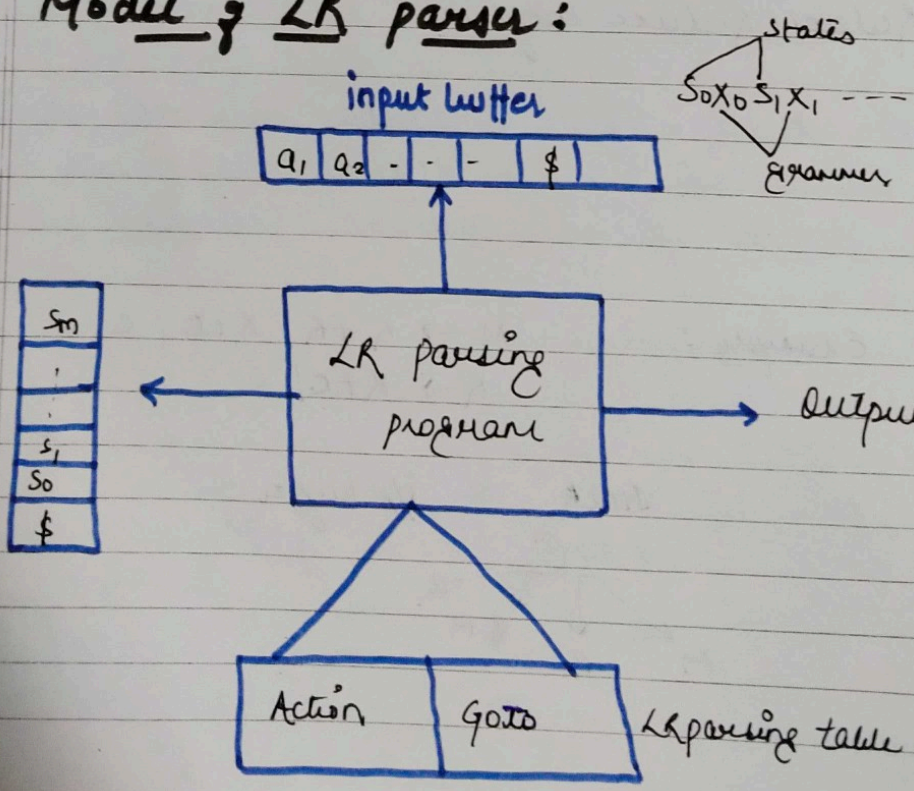
2. LR Parser:



It is an efficient bottom-up syntax analysis technique that is used to parse a large class of CFGs called LR(K) grammers where:

L — scanning i/p from left to right
R — RMD in reverse
K — no. of i/p symbols of look ahead that are used in making parsing decisions.

* Model of LR parser:



1. LR(0):

* Steps for constructing LR table:

Step-1:

Create an augmented grammer G' for a given CFG G.

CFG: $S \to AA$, $A \to ab$ --

augmented grammer → $S' \to S$, $S \to AA$, $A \to ab$ --

Step-2:

Create cannonical collection of items using GOTO & ~~closure~~ closure operations.

Step-3:

Design a parsing table using cannonical collection set.

1. Augmented Grammer:

If G is a grammer with start symbol S then G' will be the augmented grammer for G with a new start symbol S' and production $S' \to S$

Example: G: $S \to AA$, $A \to a/b$

G': $S' \to S$, $S \to AA$, $A \to a/b$

2 Canonical collection of items:

* $A \to XYZ$

four items:

$A \to .XYZ$

$A \to X.YZ$

$A \to XY.Z$

$A \to XYZ.$

* $A \to \epsilon$

only 1 item generated:

$A \to .$

An item indicates how much of a production we have seen at the given point in the parsing process.

Closure operation:

$A \to \alpha . B\beta$

$B \to \gamma$ (given)

$B \to .\gamma$ (included in closure)

Example:

$S' \to S$

$S \to AB$

$A \to a$

$B \to b$

closure ($S' \to S$)

$S' \to .S$

$S \to .AB$

$A \to .a$

* Closure of item sets:

If I is a set of items for grammer G then closure(I) is the set of items constructed from I by the 2 rules:

24th Feb'17
Lecture-27

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

2.) If $A \to \alpha . B\beta$ is in closure(I) and $B \to \gamma$ is a production then add the item $B \to .\gamma$ to closure(I). if it is not already there. Apply this rule until no more new item can be added to closure(I).

Example: G: $S \to AA$
$A \to aA \mid b$

G: $S' \to S$
$S \to AA$
$A \to aA \mid b$

closure($S' \to .S$) : $S' \to .S$
$S \to .AA$
$A \to .aA \mid .b$

* GOTO function:

If I is a set of items and X is a grammar symbol then GOTO(I, X) is defined as closure of the set of all items $[A \to \alpha X . \beta]$ such that $[A \to \alpha . X\beta]$ is in I.

The GOTO function is used to define the transitions in the LR(0) automaton for a grammar.

$$GOTO(I, X) = closure(A \to \alpha X \cdot \beta)$$

Example: Consider a grammar & find LR(0) items:

$$S \to AA$$
$$A \to aA \mid b$$

Step 1: Convert given grammar into augmented grammar.

$$G' = S' \to S$$
$$S \to AA$$
$$A \to aA \mid b$$

Step-2: Closure.

$$C = closure(S' \to .S)$$

I_0 :
$S' \to .S$
$S \to .AA$
$A \to .aA$
$A \to .b$

now apply GOTO function on each item of I_0.

$$GOTO(I_0, S) = closure(S' \to S.)$$

$I_1 = S' \to S.$

GOTO(I_0, A) = closure($S \to A.A$)

I_2 = $S \to A.A$
$A \to .aA$
$A \to .b$

GOTO(I_0, a) = closure($A \to a.A$)

I_3 = $A \to a.A$
$A \to .aA$
$A \to .b$

GOTO(I_0, b) = closure($A \to b.$)

I_4 = $A \to b.$

now for I_2 :

GOTO(I_2, A) = closure($S \to AA.$)

I_5 = $S \to AA.$

GOTO(I_2, a) = closure($A \to a.A$)

I_3 = $A \to a.A$
$A \to .aA$
$A \to .b$

GOTO(I_2, b) = closure($A \to b.$)

I_4 = $A \to b.$

now for I_3 :

GOTO(I_3, A) = closure($A \to aA.$)

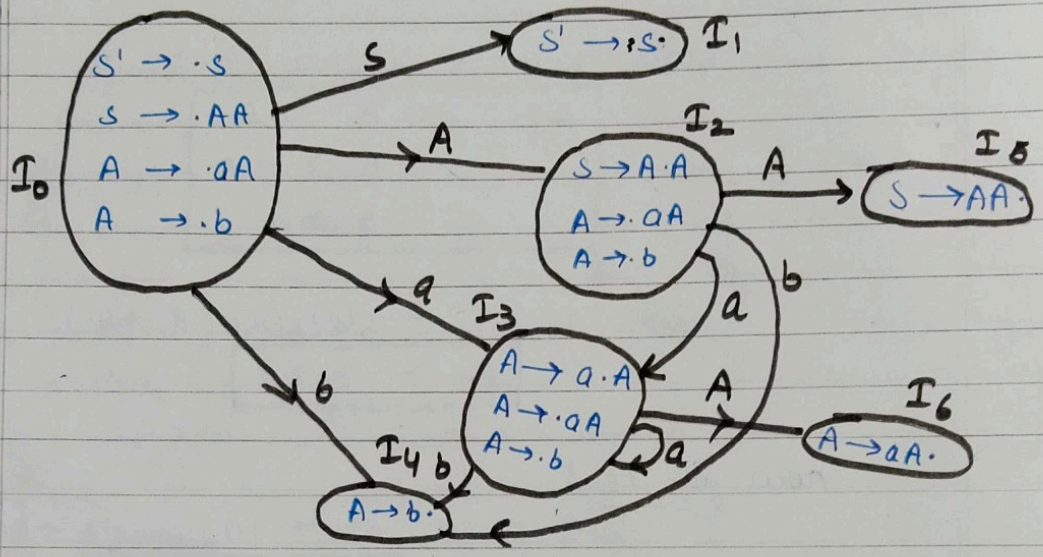
I_6 = $A \to aA.$

GOTO(I_3, a) = closure($A \to a.A$)

I_3 = $A \to a.A$
$A \to .aA$
$A \to .b$

GOTO (I_3, b) = closure (A → b·)

I_4 = A → b·

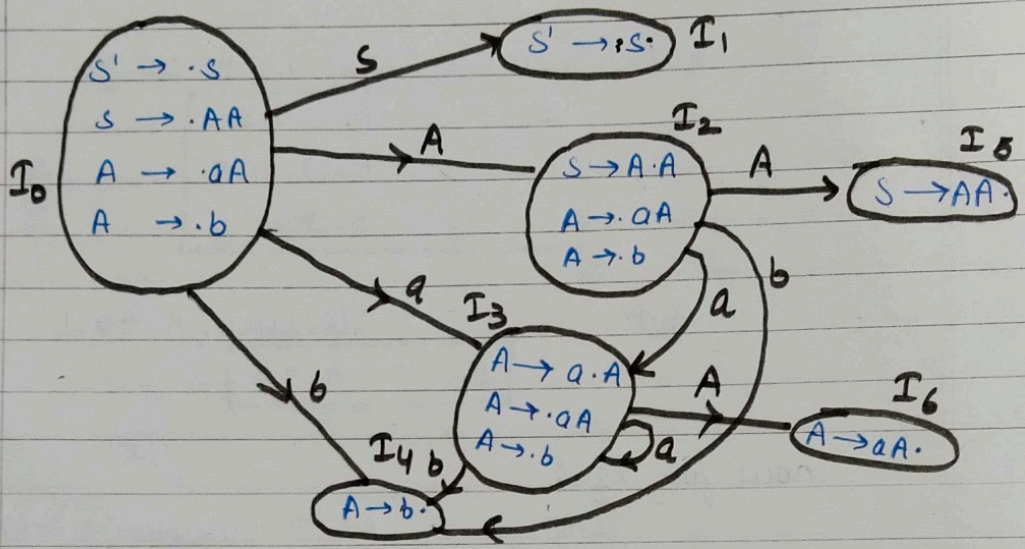


Procedure to compute canonical collection of sets of LR(0) items :

```
items(G')
{
  C = closure([S' → .S]);
  for(each set of items I in C)
     for(each grammar symbol X)
        if(GOTO(I,X) is not empty and not in C)
              add GOTO(I,X) to C;
              until no new sets of items are
              added to C on a round;
}
```

GOTO (I_3, b) = closure (A → b·)

I_4 = A → b·



Procedure to compute canonical collection of sets of LR(0) items :

```
items(G')
{
  C = closure( [S' → .S] );
  for(each set of items I in C)
     for(each grammar symbol X)
        if( GOTO(I,X) is not empty and not in C)
              add GOTO(I,X) to C;
              until no new sets of items are
              added to C on a round;
}
```

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

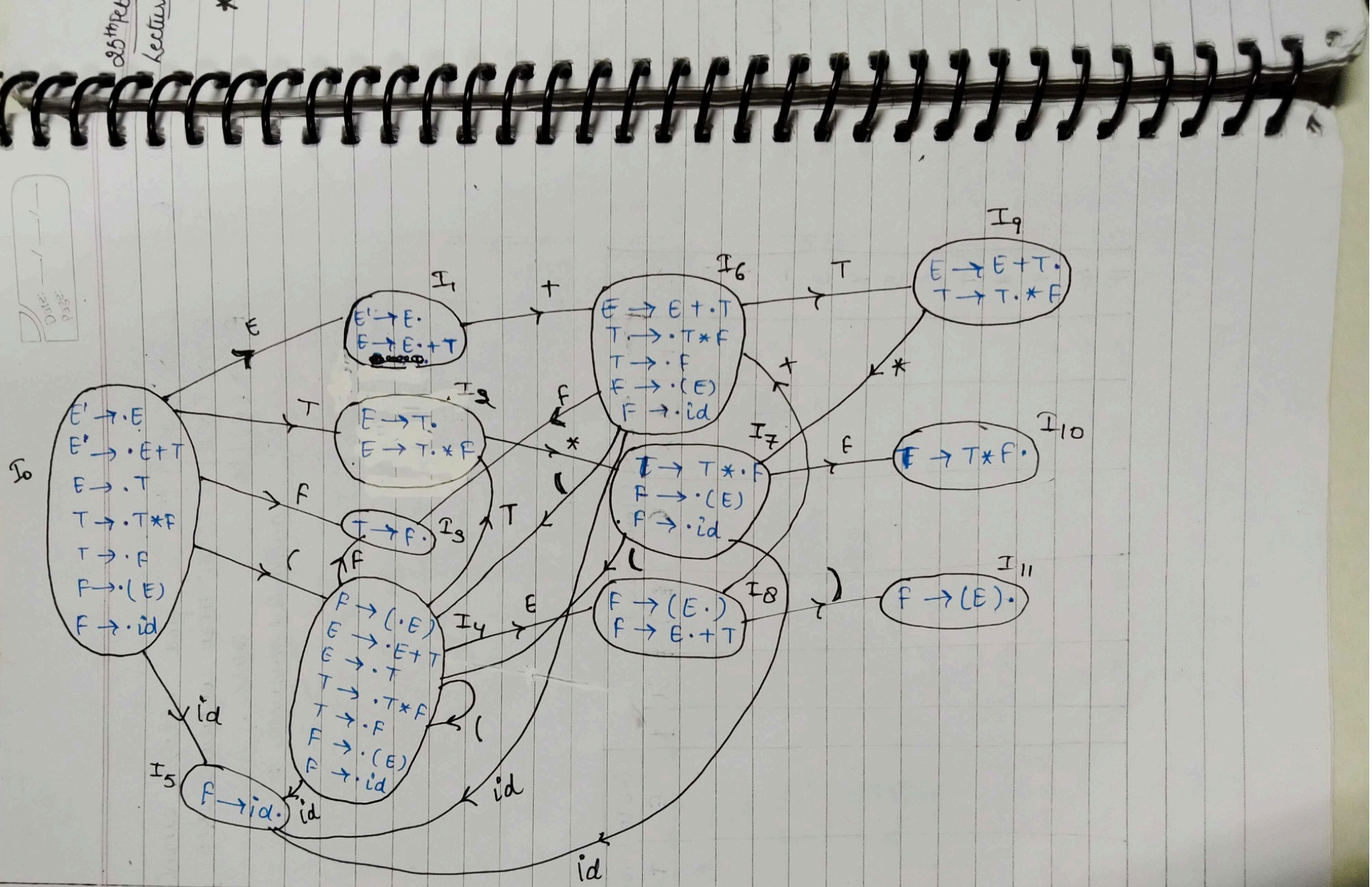
find LR(0), canonical collection & also construct LR(0) parsing table.

augmented grammar:
$E' \to E$ - *
$E \to E+T$ - 1
$E \to T$ - 2
$T \to T*F$ - 3
$T \to F$ - 4
$F \to (E)$ - 5
$F \to id$ - 6

Closure ($E' \to .E$)
$E' \to .E$
$E \to .E+T$
$I_0 :=$ $E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

	ACTION (terminals)						GOTO (non-terminals)		
I	+	*	(	)	id	$	E	T	F
0			S4		S5		1	2	3
1	S6					accept			
2	r2	S7/r2	r2	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4			S4		S5		8	2	3
5	r6	r6	r6	r6	r6				
6			S4		S5			9	3
7			S4		S5				10
8	S6			S11					
9	r1	S7/r1	r1	r1	r1				
10	r3	r3	r3	r3	r3				
11	r5	r5	r5	r5	r5				

$I_3, I_5, I_{10}, I_{11}, I_2, I_9$
↓ ↓ ↓ ↓ ↓ ↓
r4 r6 r3 r5 r2 r1


I0
E' → .E
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id
I1
E' → E.
E → E.+T
I2
E → T.
T → T.*F
I3
T → F.
I4
F → (.E)
E → .E+T
E → .T
T → .T*F
T → .F
F → .(E)
F → .id
I5
F → id.
I6
E → E+.T
T → .T*F
T → .F
F → .(E)
F → .id
I7
T → T*.F
F → .(E)
F → .id
I8
F → (E.)
E → E.+T
I9
E → E+T.
T → T.*F
I10
T → T*F.
I11
F → (E).
E
T
F
(
id
+
*
)

25th Feb'17
Lecture-28.

* Construction of LR(0) parsing table :

	ACTION (Terminals)			GOTO (Non-terminals)	
I	a	b	$	S	A
0	S3	S4	~~accept~~	1	2
1	~~S3~~		accept		~~5~~
2	S3	S4			5
3	S3	S4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

I_1, I_4, I_5 & I_6 have • at the last
So, they will perform reduce operation.

for I_4 :

$I_4 \Rightarrow$ [A → b•]

↓

it is from 3rd production

So, write r3 in 4

S' → S x — augmented so, no' numbering
S → AA ①
A → aA ②
A → b ③

for I_5 :

[S → AA•]

→ from 1st production

So, write r1

S' → S x
S → AA ①
A → aA ②
A → b ③

→ Algorithm :

Input : Augmented grammar G'

Output : The LR(0) parsing table functions ACTION & GOTO for G'.

Method : Step-1 : Construct $C = \{I_0, I_1, \cdots, I_n\}$, the collection of sets of LR(0) items for G'.

Step-2 :

State i is constructed from I_i. The parsing actions for state i are determined as follows :

a.) if $[A \to \alpha . a\beta]$ is in I_i and GOTO $(I_i, a) = I_j$ then set ACTION $[I, a]$ to "shift j". (here 'a' must be terminal).

b.) if $[A \to \alpha .]$ is in I_i then set reduce $A \to \alpha$ for all terminals.

c) if $[S' \to S.]$ is in I_i then set ACTION $[I, \$]$ to "accept".

Step-3 :

The GOTO transitions for state i are constructed for all non-terminals A using the rule :

if GOTO $(I_i, A) = I_j$ then
GOTO $(I_i, A) = j$

Step-4 :
All entries not defined by rules 2 & 3 are made "error".

Step-5 :
The initial state of the parser is constructed from the set of items containing $S' \rightarrow .S$.

ω = : aabb$

stack	i/p	Action
0	aabb$	shift
0a3	abb$	shift
0a3a3	bb$	shift
0a3a3b4	b$	reduce A → b
0a3A5	b$	reduce A → aA
0A2	b$	shift
0A2b4	$	reduce A → b
0A2A5	$	reduce S → AA
0S1	$	accept

reduce A → b : length of RHS ie; b = 1 so, pop 2 elements (4 & b pop)
reduce A → aA : pop 4 elements
pop 2
reduce S → AA (pop 4)