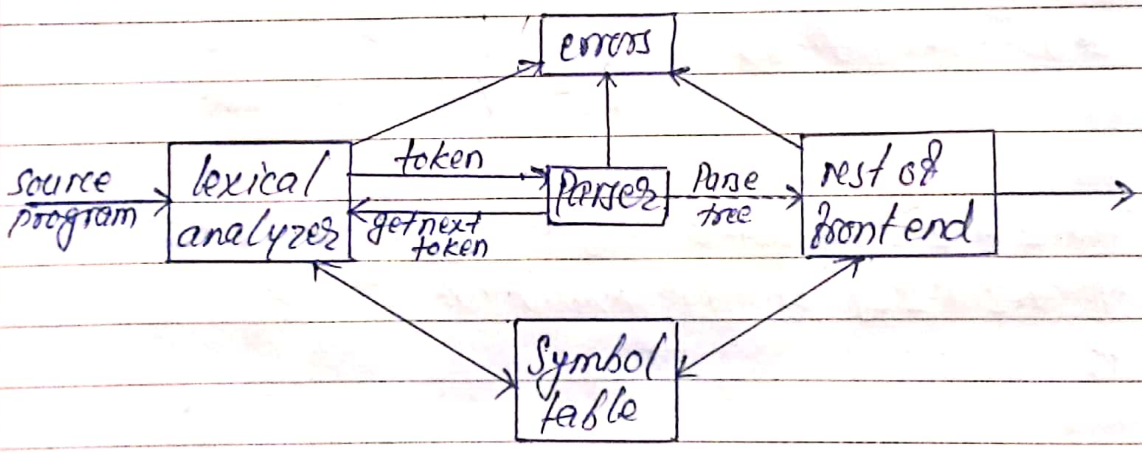
Date ______
Page ______

Syntax Analysis (Parsing)

Role of parser

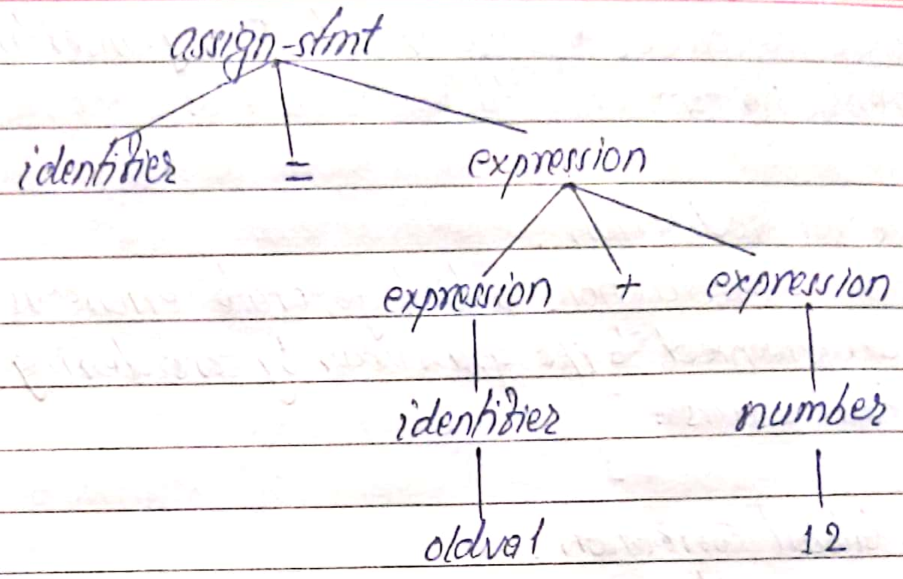
→ The second phase of the compilation process is syntax analysis commonly known as parsing.
A parser obtains the tokens from the lexical analyzer and analyzes syntactically according to the grammar of the source language whether the string can be generated or not from the grammar.
The parser works with the lexical analyzer as shown in fig:



A syntax analyzer (parser) takes tokens produced by lexical analyzer and generates a parse tree as output.

The task of parser can be expressed as:
- A parser implements a context free grammar.
- Generates the parse tree.
- Determine the errors and tries to handle them.

E.g.
newval = oldval + 12



Error Recovery strategies (Error Handling)

→ A parser should be able to detect and report any ~~progr~~ error in a program.
Programs may contain errors at different levels such as: (→ misspelling of identifier, keyword, operator)

- lexical errors : name of some identifier typed incorrectly.
- syntax errors : missing semicolon or unbalanced parenthesis.
- semantic errors : incompatible value assignment.
- logical errors : code not reachable, infinite loop.

Error Recovery Techniques:

1. Panic Mode Recovery:

Once an error is found, the parser discards the input symbol one at a time until one of the designated (like end, semicolon) set of synchronizing tokens is found.

2. Phrase-level Recovery:

When a parser encounters an error, it ~~perform~~ necessary correction on remaining input so that the rest of input statement

Date
Page

allow the parser to parse ahead. For e.g. inserting missing semicolon, replacing comma with semicolon etc.

3. Error Production :
Productions which generate erroneous constructs are augmented to the grammar by considering common errors that occurs.

4. Global correction :
The parser examines the whole program and tries to find out the closest match for it which is error free.

Context Free Grammar (CFG)

A CFG is defined by 4-tuples as G = (V, T, P, S) where
- V is a finite set of variable symbols (Non-terminals)
- T is a finite set of terminal symbols
- P is a set of production rules
- S is a start symbol, $S \in V$

E.g:

(i) $E \to EAE \mid (E) \mid -E \mid id$
$A \to + \mid - \mid * \mid / \mid \uparrow$
} → Grammar to define an infix expression

Here, E and A are non-terminals with E as start symbol and other symbols are terminals.

(ii) $S \to 0S0 \mid 1S1$
$S \to 0 \mid 1$
} Grammar to define a palindrome string over binary string.

Date
Page

CFG Terms:

- Terminal : An atomic building block.
- Non-terminal: A building block of the CFG that are made of other terminals and/or non-terminals.
- Production rule : Rules that define how a non-terminal is constructed from other terminals and non-terminals.
- start symbol : The start symbol is a non-terminal symbol that defines the starting point for the definition of the entire grammar.

CFG Notational Conventions:

- Terminals are denoted by lower-case letters, symbols (like operators), bold strings.
  E.g. a, b, c, 0, 1 ∈ T
- Non-terminals are denoted by italicized letters (string) or upper-case letters.
  E.g. A, B, C, .. ∈ V   expr, term, stmt ∈ V
- Production rules are of the form
  $A \rightarrow a|b$ i.e. read as "A produces a or b".

Derivation

- Process of obtaining the terminal strings from the start symbol of the grammar.

$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$

one-step derivation:

$\alpha \Rightarrow \beta$ (β can be derived from α by applying the production of α)

zero or more step derivation:

$\alpha \overset{*}{\Rightarrow} \beta$

↳ derived by 0 or more production rule from α.

one or more step derivation:

$\alpha \overset{+}{\Rightarrow} \beta$

Date ______
Page ______

leftmost derivation:

If we ~~apply production~~ replace the left most non-terminal symbol in each derivation step, then this derivation is called left-most derivation.

E.g.

$E \Rightarrow EAE$

$\underset{lm}{\Rightarrow} id\ AE$

$\underset{lm}{\Rightarrow} id * E$

$\underset{lm}{\Rightarrow} id * id$

Right-most derivation:

If we replace the right-most non-terminal symbol in each derivation step, then this derivation is called right-most derivation.

E.g.

$E \Rightarrow EAE$

$\underset{rm}{\Rightarrow} EA\ id$

$\underset{rm}{\Rightarrow} E * id$

$\underset{rm}{\Rightarrow} id * id$

Parse Tree

- A graphical representation of the derivation of any string from the grammar.

- The root node is labeled by start symbol.
- Inner nodes of parse tree are non-terminal symbol.
- The leaves of parse tree are terminal symbol.

e.g.

① Grammar

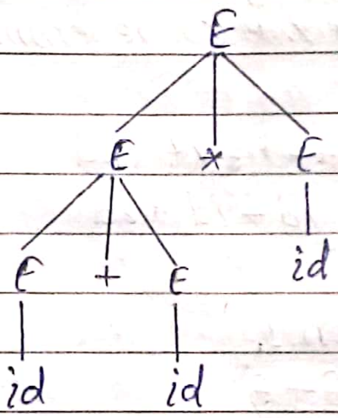
$E \rightarrow E+E \mid E*E \mid id$

string: id+id*id

Derivation:

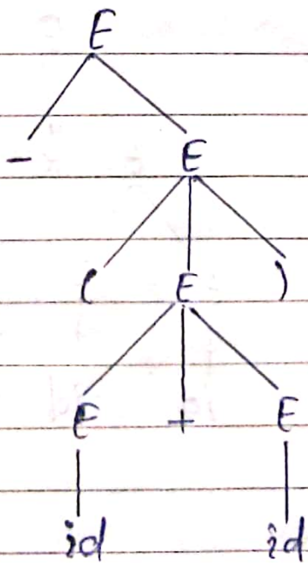
$E \Rightarrow E*E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

Parse tree:



② $E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$

string: -(id+id)



Date ____
Page ____

Ambiguity of a grammar

If a same terminal string can be derived from the grammar using two or more distinct left-most derivation (or right most), then the grammar is said to be ambiguous i.e. from an ambiguous grammar, we can get two or more distinct parse tree for the same terminal string.

e.g.

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

string : $id + id * id$

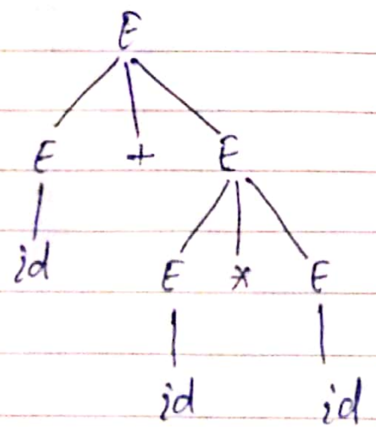
Derivation 1:

$E \Rightarrow E+E$
$\Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

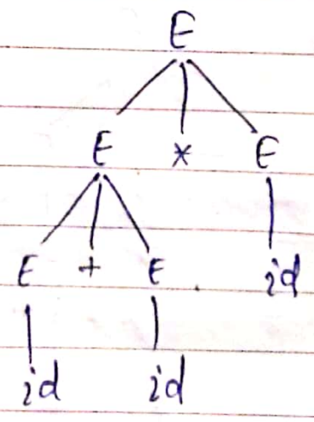
Derivation 2:

$E \Rightarrow E*E$
$\Rightarrow E+E*E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$.

Parse tree 1:



Parse tree 2:



∴ The given grammar is ambiguous.

Left Recursion

A left recursive grammar is one that has rules like $A \to A\alpha$, for some string α.

- Top down parsing technique cannot handle left-recursive grammars. so, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.

The left recursion from the grammar can be removed as;
In general, left recursive rules like,

$A \to A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ where β_i do not start with A

Removing left recursion as

$A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$

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

E.g.

(1) ~~A~~

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

eliminating immediate left recursion

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

Sentential => कुनै ~~वटा~~ string drived from variable.

Parsing

Given a stream of input tokens, parsing involves the process of reducing them into a non-terminal.
Parsing can be either top-down or bottom up.

Top down parsing: - build the parse tree from root to leave

Top-down parsing involves generating the string from the first non-terminal (start-symbol) and repeatedly applying the production of grammar.

Bottom-up parsing:

Bottom-up parsing involves repeatedly rewriting the input string until it ends up in the first non-terminal of the grammar.

E.g.
Grammar:

$S \rightarrow E$

$E \rightarrow E+T \mid E*T \mid T$

$T \rightarrow id$ or $a \mid b \mid c$

Input string : $a+b*c$

Top down Parsing	Bottom-up parsing
S	$a+b*c$
$= E$	$= T+b*c$
$= E*T$	$= E+b*c$
$= E+T*T$	$= E+T*c$
$= T+T*T$	$= E*c$
$= a+T*T$	$= E*T$
$= a+b*T$	$= E$
$= a+b*c$	$= S$

Following are the top-down parsing Algorithms:
1. Recursive Descent Parsing
2. Non-recursive predictive parsing

- If grammar is left recursive then recursive descent parsing cannot solve.

Recursive Descent Parsing

Recursive descent parsing is a top-down passing technique that uses a set of recursive procedure to scan its input from left to right. It may involve backtracking. It starts from the root node (start symbol) and use the 1st production, match the input string against the production rules and if there is no match backtrack and apply another rule.

Rules:
1. use two pointers 'iptr' for pointing input string and 'optr' for output string ~~and initially iptr = optr = 0~~. Initially output is start symbol S.
2. If the symbol pointed by optr is non-terminal use the first production rule for expansion.
3. while the symbol pointed by iptr and optr is same increment the both pointers.
4. The loop at the above step terminates when
   a.) A non-terminal is encountered in output
   b.) end of string is reached
   c.) Symbol pointed by iptr and optr is unmatched.
5. If 'a' is true, then goto step 2 & expand non-terminal with 1st prod
6. If 'b' is true, terminate with success.

7. If 'c' is true, decrement both pointers to place the last non-terminal expansion and use the next production rule for non-terminal.

e.g.

① Grammar

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Input: cad

Initially, iptr = optr = 0

Input	output	
iptr (cad)	optr (S)	
iptr (cad)	optr (cAd)	
c iptr (ad)	c optr (Ad)	
c iptr (ad)	c optr (abd)	– Replace with 1st prod.
ca iptr (d)	ca optr (bd)	– iptr(ω) ≠ optr(α) so backtrack
c iptr (ad)	c optr (Ad)	– Replace with 2nd prod.
c iptr (ad)	c optr (ad)	
ca iptr (d)	ca optr (d)	
cad iptr (eof)	cad optr (eof)	
	parsing Complete.	

② Given the following grammar

$R \rightarrow id\,S \mid (R) \mid S$

$S \rightarrow +RS \mid .RS \mid *S \mid e$

Then token id can be one of {a, b}. Determine whether the following strings are in the grammar using recursive descent parsing.

1. a.(a+b)*.b 2. (b.a.b)* 3. a.b..b.a

1)

Input: a.(a+b)*.b

Input	output	
iptr [a.(a+b)*.b]	optr [R]	
iptr [a.(a+b)*.b]	optr [id S]	
a iptr [.(a+b)*.b]	id optr [S]	
a iptr [.(a+b)*.b]	id optr [+RS]	→ No matching, backtrack
a iptr [.(a+b)*.b]	id optr [S]	
a iptr [.(a+b)*.b]	id optr [.RS]	
a. iptr [(a+b)*.b]	id. optr [RS]	
a. iptr [(a+b)*.b]	id. optr [id S]	→ No matching, backtrack
a. iptr [(a+b)*.b]	id. optr [RS]	
a. iptr [(a+b)*.b]	id. optr [(R)S]	
a.(iptr [a+b)*.b]	id.(optr [R)S]	
a.(iptr [a+b)*.b]	id.(optr [id S)S]	→ Backtr
~~a.(a iptr [+b)*.b]~~	~~id.(id optr [R)S]~~	
~~a.(iptr [+b)*.b]~~	~~id.(optr [(R))S]~~	— Backtrack
~~a.(iptr [+b)*.b]~~	~~id.(optr [R)S]~~	
~~a.(iptr [+b)*.b]~~	~~id.(optr [S)S]~~	
~~a.(iptr [+b)*.b]~~	id.(optr [+RS)S]	
a.(a iptr [+b)*.b]	id.(id optr [S)S]	Replace id by a & b
a.(a iptr [+b)*.b]	id.(id optr [+RS)S]	
a.(a+ iptr [b)*.b]	id.(id+ optr [RS)S]	
a.(a+ iptr [b)*.b]	id.(id+ optr [id SS)S]	
a.(a+b iptr [)*.b]	id.(id+id optr [SS)S]	
a.(a+b iptr [)*.b]	id.(id+id optr [+RSS)S]	— backtrack
a.(a+b iptr [)*.b]	id.(id+id optr [SS)S]	
a.(a+b iptr [)*.b]	id.(id+id optr [.RSS)S]	— Bt
a.(a+b iptr [)*.b]	id.(id+id optr [SS)S]	
a.(a+b iptr [)*.b]	id.(id+id optr [*SS)S]	
a.(a+b iptr [)*.b]	id.(id+id optr [SS)S]	
a.(a+b iptr [)*.b]	id.(id+id optr [εS)S]	— Complete (Not completed)

Predictive parsing

A predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

The predictive parser consists of following components: Input buffer, Parse table, stack. ↓

Non-Recursive predictive parsing

Non-recursive predictive parsing is a table driven parser. The table driven parser has stack, input buffer, parsing table and output stream.

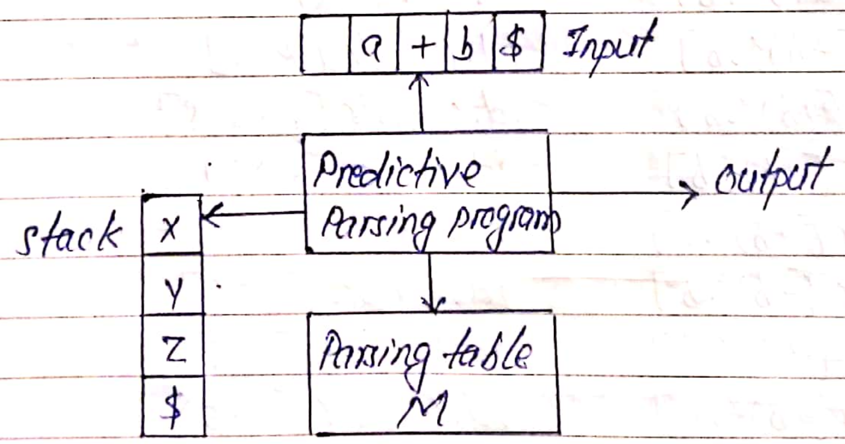


Fig: Non-recursive predictive parsing

- Input buffer contains the string to be parsed followed by a special symbol $.
- The stack contains symbol of grammar. Initially stack contains the symbol $. When the stack is empty i.e. only $ left in the stack, the parsing is completed.
- Parsing table is a two dimensional array M[A,a] where 'A' is non-terminal & 'a' is terminal symbol. Each entry in the parsing table holds the production rule.

Algorithm for Non-Recursive Predictive Parsing:

Input: A string w and a parsing table for the grammar G.

1. Set ip to the first symbol of the input string.
2. set the stack to $S where S is the start symbol of the grammar
3. let X be the top stack symbol and 'a' be the symbol pointed by ip then

   Repeat
   - a) If X is a terminal or $ then
     - i) If X = a then pop X from the stack and advance ip
     - ii) else error()
   - b) Else
     - i) If $M[X, a] = Y_1 Y_2 Y_3 \ldots Y_K$ then
       - Pop X from stack
       - Push $Y_K, Y_{K-1}, \ldots, Y_2, Y_1$ on the stack with Y_1 on top.
       - output the production $X \rightarrow Y_1 Y_2 Y_3 \ldots Y_K$
     - ii) else error()

   until X = $

Example

1) Given a grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

Given the parsing table for the grammar G as:

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

→ LL(1) Parsing table

Input : id+id*id

Stack	Input	output
$E	id+id*id$	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id$	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

Parsing complete with success.

The leftmost derivation:

$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow id\,T'E' \Rightarrow id\,E' \Rightarrow id+TE' \Rightarrow id+FT'E'$
$\Rightarrow id+id\,T'E' \Rightarrow id+id*FT'E' \Rightarrow id+id*id\,T'E'$
$\Rightarrow id+id*id\,E' \Rightarrow id+id*id$

---

2) $S \rightarrow aBa$
$B \rightarrow bB \mid \epsilon$

Parsing table:

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \epsilon$	$B \rightarrow bB$	

→ Given table

Input: abba

stack	Input	output
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \epsilon$
$a	a$	
$	$	Accept.

Parsing Complete with success.

The leftmost derivation:

$S \Rightarrow aBa \Rightarrow abBa \Rightarrow abbBa \Rightarrow abba$

Date ____
Page ____

Constructing LL(1) Parsing table / Predictive parsing table.

The parse table construction requires two functions: FIRST and FOLLOW.

FIRST(α) : set of symbols (terminal) that can appear at the begining of string derived from α.

FOLLOW(α) : set of symbols (terminal) that immediately follows the symbol α in any sentential.

ABaS ⇒ Follow(B) = {a}
FOLLOW(A) = FIRST(B)

Computation of FIRST :

1. For all terminals 'a'
   FIRST(a) = {a}
2. For any non-terminal X, if $X \rightarrow \epsilon$ then
   FIRST(X) = FIRST(X) ∪ {ε}
3. If X is non-terminal and $X \rightarrow Y_1 Y_2 \dots Y_K$ is a production then
   FIRST(X) = FIRST(X) ∪ FIRST(Y_1)
   - If FIRST(Y_1) contains ε then
     FIRST(X) = FIRST(X) ∪ FIRST(Y_2)
   - If ε is in FIRST(Y_1) ∩ FIRST(Y_2) ∩ .... ∩ FIRST(Y_K) then
     FIRST(X) = FIRST(X) ∪ {ε}

FIRST/FOLLOW calculate गर्दा left recursion छ भने हटाएर मात्र निकाल्ने |
& left factoring

Date ______
Page ______

Example:

(1)
$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid a$

$FIRST(E) = FIRST(E+T) \cup FIRST(T)$
$= FIRST(E) \cup FIRST(T*F) \cup FIRST(F)$
$= FIRST(T) \cup FIRST(F) \cup \{(, a\}$
$= FIRST(F) \cup FIRST(F) \cup \{(, a\}$
$= FIRST(F)$
$FIRST(E) = FIRST(T) = FIRST(F) = \{(, a\}$

[illegible]

Removing left recursion
$E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to (E) \mid a$

$FIRST(E) = FIRST(T) = FIRST(F) = \{(, a\}$
$FIRST(E') = \{+, \epsilon\}$
$FIRST(T') = \{*, \epsilon\}$

(2)
$A \to Bab \mid bb$
$B \to cBa \mid \epsilon$
$C \to cC$

$FIRST(A) = FIRST(Bab) \cup FIRST(bb)$
$= \{b, c, \epsilon\}$
$FIRST(B) = \{c, \epsilon\}$ $\quad$ $FIRST(C) = \{c\}$

③ $A \rightarrow BXb \mid bb$
$B \rightarrow CBa \mid \epsilon$
$X \rightarrow a \mid y \mid z$

FIRST(B) null भएकोले X सँग union गर्ने

FIRST(X) = {a, y, z}
FIRST(A) = FIRST(B) ∪ FIRST(X)

④ $S \rightarrow ABC$
$A \rightarrow a \mid b$
$B \rightarrow c \mid d \mid \epsilon$
$C \rightarrow e \mid \epsilon$

FIRST(C) = FIRST(e) ∪ FIRST(ε) = {e} ∪ {ε} = {e, ε}
FIRST(B) = {c, d, ε}
FIRST(A) = {a, b}
FIRST(S) = FIRST(ABC) = FIRST(A) = {a, b}

⑤ $S \rightarrow ab \mid CB \mid D$
$B \rightarrow bB \mid D$
$C \rightarrow cC \mid \epsilon$
$D \rightarrow aD \mid cc$

FIRST(D) = {a, c}
FIRST(C) = {c, ε}
FIRST(B) = {b} ∪ FIRST(D) = {b} ∪ {a, c} = {a, b, c}
FIRST(S) = {a} ∪ FIRST(C) ∪ FIRST(D)
~~= {a} ∪ FIRST(B) ∪ FIRST(D)~~
~~= {a} ∪ {a, b, c} ∪ {~~
= {a, c, ε} ∪ FIRST(B) /Because C is nullable.
= {a, b, c, ε}

Date ______
Page ______

* Computation of FOLLOW:

1. Place \$ in Follow(S), where S is the start symbol.
2. If there is a production of the form $A \to \alpha B \beta$

   $FOLLOW(B) = FIRST(\beta)$ except ϵ
3. If there is a rule $A \to \alpha B$ or a rule $A \to \alpha B \beta$ where $\epsilon \in First(\beta)$ then

   $FOLLOW(B) = FOLLOW(A)$

Example:

① $E \to TE'$
$E' \to +TE' \mid \epsilon$
$T \to FT'$
$T' \to *FT' \mid \epsilon$
$F \to id \mid (E)$

[Follow मा ∈ राख्दैनौं]

$FOLLOW(E) = \{ \$,) \}$
$FOLLOW(E') = \{ \$,) \}$
$FOLLOW(T') = FOLLOW(T) = FIRST(E') = \{+\} \cup \{ \$,) \}$
$= \{ +, \$,) \}$

→ first of E' ∈ आएकोले follow of E' पनि union गर्ने

→ FIRST(T') ∈ आएकोले Follow of T' पनि union गर्ने

$FOLLOW(F) = \{ *, +, \$,) \}$

② $S \to iEtSS' \mid a$
$S' \to eS \mid \epsilon$
$E \to b$

$FOLLOW(S) = \{ \$, e \}$
$FOLLOW(S') = \{ \$, e \}$
$FOLLOW(E) = \{ t \}$

Date ____
Page ____

(3) $S \to ABCDE$
$A \to a \mid \epsilon$
$B \to b \mid \epsilon$
$C \to c$
$D \to d \mid \epsilon$
$E \to e \mid \epsilon$

FOLLOW(S) = {\$}
FOLLOW(A) = {b, c}
FOLLOW(B) = {c}
FOLLOW(C) = {d, e, \$}
FOLLOW(D) = {e, \$}
FOLLOW(E) = {\$}

(4) $S \to Bb \mid Cd$
$B \to aB \mid \epsilon$
$C \to cC \mid \epsilon$

FOLLOW(S) = {\$}
FOLLOW(B) = {b}
FOLLOW(C) = {d}

(5) $S1 \to S\#$
$S \to qABC$
$A \to a \mid bbD$
$B \to a \mid \epsilon$
$C \to b \mid \epsilon$
$D \to c \mid \epsilon$

FIRST(S1) = {q}
FIRST(S) = {q}
FIRST(A) = FIRST(a) U FIRST(bbD)
= {a} U {b}
= {a, b}
FIRST(B) = FIRST(a) U FIRST(ε)
= {a, ε}
FIRST(C) = FIRST(b) U FIRST(ε)
= {b, ε}
FIRST(D) = FIRST(c) U FIRST(ε)
= {c, ε}

FOLLOW(S1) = {\$}
FOLLOW(S) = {#}
FOLLOW(A) = ~~{FIRST(BC) − {ε}~~ {a, b, #} → FIRST(BC) − {ε} U FOLLOW(S)
FOLLOW(B) = {b, #} → FIRST(C) − {ε} U FOLLOW(S)
FOLLOW(C) = FOLLOW(S) = {#}
FOLLOW(D) = FOLLOW(A) = {a, b, #}

Algorithm for constructing LL(1) parsing table / predictive parsing table:

Input : LL(1) grammar G
Output : Parsing table M

For each production $A \rightarrow \alpha$ of the grammar do,

1. For each terminal $a \in FIRST(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $FIRST(\alpha)$, for each b in $FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, b]$
3. If ϵ is in $FIRST(\alpha)$ and \$ is in $FOLLOW(A)$ then add $A \rightarrow \alpha$ to $M[A, \$]$.

Example

① $R \rightarrow id\,S \mid (R)S$
$S \rightarrow +RS \mid .RS \mid *S \mid \epsilon$

The FIRST and FOLLOW is computed as

$FIRST(R) = \{id, (\}$ $\qquad$ $FOLLOW(R) = \{), +, ., *, \$\}$

$FIRST(S) = \{+, ., *, \epsilon\}$ $\qquad$ $FOLLOW(S) = \{), +, ., *, \$\}$

The parsing table is as below:

	id	(	)	+	.	*	\$
R	$R \rightarrow idS$	$R \rightarrow (R)S$					
S			$S \rightarrow \epsilon$	$S \rightarrow +RS$ $S \rightarrow \epsilon$	$S \rightarrow .RS$ $S \rightarrow \epsilon$	$S \rightarrow *S$ $S \rightarrow \epsilon$	$S \rightarrow \epsilon$

If given grammar is ambiguous then parsing table of this grammar contains multiple different entity for some $M[x, a]$.

(the above parsing table is ambiguous)

② E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id

FIRST(F) = { (, id }
FIRST(T') = { *, ε }
FIRST(T) = { (, id }
FIRST(E') = { +, ε }
FIRST(E) = { (, id }
FIRST(TE') = { (, id }
FIRST(+TE') = { + }
FIRST(ε) = {ε}
FIRST(FT') = { (, id }
FIRST(*FT') = { * }
FIRST(ε) = { ε }
FIRST((E)) = { (}
FIRST(id) = { id }

FOLLOW(E) = { $,) }
FOLLOW(E') = { $,) }
FOLLOW(T) = { +,), $ }
FOLLOW(T') = { +,), $ }
FOLLOW(F) = { *, +,), $ }

Parsing table:

terminal → / Non terminal ↓	+	*	(	)	id	$
E			E→TE'		E→TE'	
E'	E'→+TE'			E'→ε		E'→ε
T			T→FT'		T→FT'	
T'	T→ε	T'→*FT'		T→ε		T→ε
F			F→(E)		F→id	

Date ______
Page ______

(3) S → aSbS | bSaS | ε [show that this grammar is ambiguous by constructing parsing table]

FIRST(S) = {a, b, ε} FOLLOW(S) = {a, b, $}
FIRST(aSbS) = {a}
FIRST(bSaS) = {b}
FIRST(ε) = {ε}

Parsing table :

	a	b	$
S	S → aSbS S → ε	S → bSaS S → ε	S → ε

In above parsing table, ~~same~~ entries for M[S, a] and M[S, b] is multiply-defined. Hence the given grammar is ambiguous.

(4) S → A
A → aB | ad
B → bBC | f
C → g

FIRST(S) = {a} FIRST(d) = {d}
FIRST(A) = {a} FIRST(bBC) = {b}
FIRST(B) = {b, f} FIRST(f) = {f}
FIRST(C) = {g} FIRST(g) = {g}
FIRST(aB) = {a}

since the grammar is ε-free, FOLLOW sets are not required to be computed in order to enter the productions in to the parsing table.

parsing table :

	a	b	f	g	d	$
S	S → A					
A	A → aB				A → d	
B		B → bBC	B → f			
C				C → g		

Non-LL(1) grammar लाई LL(1) grammar मा convert गर्ने - no any general rules.

Date ______
Page ______

LL(1) Grammar

A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar.

- The first L in LL(1) corresponds to reading the input left to right and second 'L' corresponds to the left most derivation. The 1 in the parenthesis corresponds lookahead of 1 symbol.
- A left recursive or ambiguous grammar cannot be a LL(1) grammar.

In any LL(1) grammar, if there exists a rule of the form
$A \to \alpha | \beta$ [$A \to \alpha$ & $A \to \beta$] then,

1. $FIRST(\alpha) \cap FIRST(\beta) = \phi$
2. Either $\alpha \overset{*}{\Rightarrow} \epsilon$ or $\beta \overset{*}{\Rightarrow} \epsilon$, but not both.
3. If $\beta \overset{*}{\Rightarrow} \epsilon$, then α doesn't derive any string begining with the terminal in FOLLOW(A).

Example

$S \to AaAb \mid BbBa$
$A \to \epsilon$
$B \to \epsilon$

For a pair of production $S \to AaAb \mid BbBa$
$FIRST(AaAb) \cap FIRST(BbBa) = \{$

कुनै पनि ~~LL(1)~~ grammar LL(1) grammar हो होइन भनेर देखाउन आयो भने त्यसको parsing table बनाउने if कुनै entry multiply-defined भयो भने - not LL(1) grammar else LL(1).

Date ______
Page ______

Left Factoring

A grammar contains left factoring if it contains production rules in the form of

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$$

→ α is common.

Eliminate left factoring as:

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$$

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

E.g.

(i) $A \rightarrow aAB \mid aA$

$B \rightarrow bB \mid b$

Eliminating left factoring

$A \rightarrow aA'$

$A' \rightarrow AB \mid A$

$B \rightarrow bB'$

$B' \rightarrow B \mid \epsilon$

(ii) $A \rightarrow aAB \mid aA \mid a$

⇓

$A \rightarrow aA'$

$A' \rightarrow AB \mid A \mid \epsilon$

(सबैका a common निकाल्दा यसमा e नभएको कारणले ϵ लिएको)

(iii) $S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

⇓

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow \epsilon \mid eS$

$E \rightarrow b$

Bottom-up parsing

- Bottom-up parsing attempts to construct a parse tree for an input string starting from leaves (the bottom) and working up towards the root (the top).
- The process of replacing a substring by a non-terminal in bottom-up parsing is called reduction.

Date ______
Page ______

E.g.

S → aABe
A → Abc | b
B → d

String: abbcde can be reduced to s as

abbcde
aAbcde (using A → b)
aAde (A → Abc)
aABe (B → d)
S (S → aABe)

Handle:

A substring that can be replaced by a non-terminal when it matches its right sentential form is called handle.

Shift-Reducing Parsing / stack implementation of shift-reducing parsing

The process of reducing the given input string into the starting symbol is called shift-reduce parsing.

string —Reduced to→ start symbol

A shift reduce parser uses a stack to hold the grammar symbol and an input buffer to hold the input string w.

Algorithm:

1. Initially stack contains only the sentinal $, and the input

Scanned with CamScanner

buffer contains the input string w$.

2. While stack not equal to $S do
   a) While there is no handle at the top of the stack, shift the input buffer and push the symbol onto the stack.
   b) If there is a handle on the top of the stack, then pop the handle and reduce the handle with its non-terminal and push it onto stack.

→ There are four actions of shift-reduce parser:

i) Shift : In a shift action, the next symbol is shifted onto the top of the stack.

ii) Reduce : In a reduce action, the handle that appears on the top of stack is replaced with non-terminal.

iii) Accept : In an accept action, parser announces successful completion of parsing.

iv) Error : Parser finds a syntax error, and calls an error recovery routine.

Examples

(Bottom-up)

S → aABe
A → Abc | b
B → d
string : abbcde

Stack	Input	Action
$	abbcde$	
$a	bbcde$	shift a
$ab	bcde$	shift b
$aA	bcde$	reduce A→b
$aAb	cde$	shift b

Stack	Input	Action
$aAbc	de$	shift c
$aA	de$	reduce A→Abc
$aAd	e$	shift d
$aAB	e$	reduce B→d
$aABe	$	shift e
$S	S	reduce by S→aABe

(2) $E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$ string : id+id*id

Stack	Input	Action
$	id+id*id $	
$ id	+id*id $	shift id
$ F	+id*id $	Reduce by F→id
$ T	+id*id $	Reduce by T→F
$ E	+id*id$	Reduce by E→T
$ E+	id*id$	shift +
$ E+id	*id$	shift id
$ E+F	*id$	Reduce by F→id
$ E+T	*id$	Reduce by T→F
$ E+T*	id $	shift *
$ E+T*id	$	shift id
$ E+T*F	$	Reduce F→id
$ E+T	$	Reduce T→T*F
$ E	$	Reduce E→E+T

shift * OR reduced by E→T [CONFLICT] → $E+T

$E \to E-E$

$E \to E*E$

$E \to id$ id-id*id

Date
Page

(3) $S' \to S$
$S \to (S)S \mid \epsilon$ Input : ( )

Stack	Input	output
\$	( )\$	shift (
\$(	)\$	reduce $S \to \epsilon$
\$(S	)\$	shift)
\$(S)	\$	reduce $S \to \epsilon$
\$(S)S	\$	reduce $S \to (S)S$
\$S	\$	reduce $S' \to S$
\$S'	\$	accept

Conflicts in shift-Reducing Parsing:

All grammars cannot use the shift-reduce parsing since there may be conflicts during the parsing actions. There are two kinds of shift-reduce conflicts:

1. shift/reduce conflict:

Here, the parser is not able to decide whether to shift or to reduce. E.g.

$A \to ab \mid abcd$, the stack contains \$ab and input buffer contains cd\$, the parser cannot decide whether to reduce \$ab to \$A or to shift two more symbols before reducing.

2. Reduce/Reduce conflict:

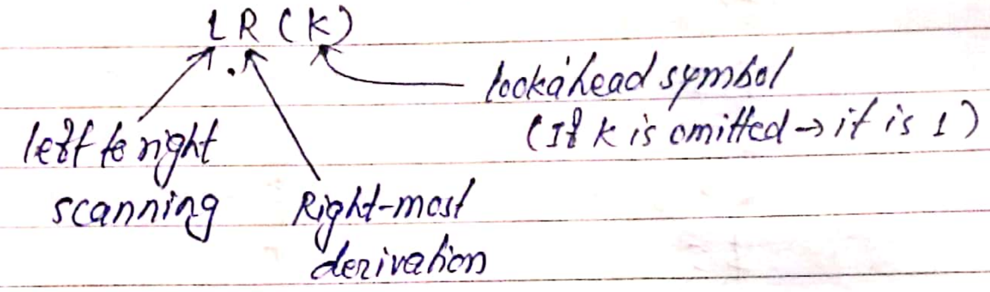
Here, the parser cannot decide which sentential form to use for reduction. E.g.

$A \to bc$
$B \to abc$ and stack contains \$abc, the parser cannot decide to reduce it to \$aA or to \$B.

2 are production of grammar

LR Parser

- LR parser is a non-recursive shift reduce bottom up parser.
- An LR-parser can ~~deect~~ detect a syntatic error so fast.
- It is also known as LR(K) parsing.

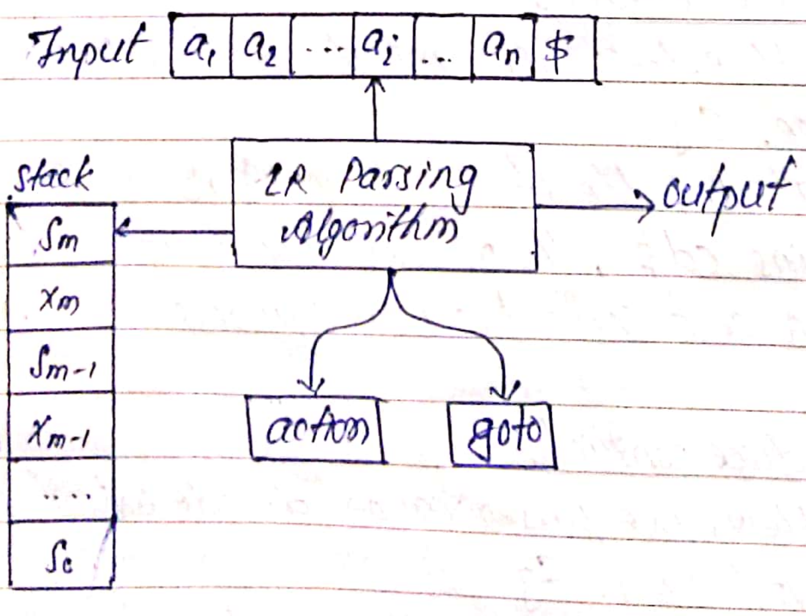


LR-parser cover wide range of grammars.

- SLR – simple LR parser
- LR – most general LR parser
- LALR – intermediate LR parser (look-ahead parser)

SLR, LR & LALR work same (they used the same algorithm), only their parsing tables are different.

Structure of LR parser:



A→ε generates only one item A→.

Date ______
Page ______

A LR parser contains a stack, an input buffer and a parsing table that has two parts: action and goto. The stack contains entries of the form $S_0 X_1 S_1 X_2 \ldots X_m S_m$ where every S_i is called state and every X_i is a grammar symbol.

Constructing SLR Parsing Table

For constructing a SLR parsing table of given grammar we need, to construct the canonical LR(0) collection of the grammar, which uses the 'closure' operation & 'goto' operation.

* Item: [LR(0) items]

An 'item' is a production rule that contains dot (.) somewhere in the right side of the production.

E.g.

The production $A \rightarrow \alpha A \beta$ yields the following items

$A \rightarrow .\alpha A \beta$, $A \rightarrow \alpha . A \beta$, $A \rightarrow \alpha A . \beta$, $A \rightarrow \alpha A \beta .$

* closure operation:

If I is a set of items for a grammar G, then the closure (I) is the set of items constructed from I using the following rules:

1. Initially, every item in I is added to closure (I)
2. If $A \rightarrow \alpha . B \beta$ is in closure(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow .\gamma$ to closure(I) if it is not already there.

Repeat until no more new items can be added to closure(I).

Date ________
Page ________

E.g.

$E' \to E$

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (E) \mid id$

If $I = \{E' \to .E\}$ then

$$closure(I) = \{ E' \to .E,\ E \to .E+T,\ E \to .T,\ T \to .T*F,\ T \to .F,\ F \to .(E),\ F \to .id \}$$

> **closure**
> → Itself
> → • को पछाडि variable आयो भने त्यसको production मा • राखेर लेख्दै जाने।

* Goto operation:

~~In any item I, for all production A → α.Xβ that are in I,~~
~~Goto [I, X]~~

If I is a set of items and X is a grammar symbol (terminal or non-terminal) then goto[I, X] is defined as;

If $A \to \alpha.X\beta$ in I then every item in closure($\{A \to \alpha X.\beta\}$) will be in goto[I, X].

Example;

In above E.g.

If I_0 = closure($\{E' \to .E\}$) then

goto[I_0, E] = closure($\{E' \to E.,\ E \to E.+T\}$) since the closure($\{E' \to .E\}$) is $\{E' \to .E,\ E \to .E+T,\ E \to .T,\ T \to .T*F,\ T \to .F,\ F \to .(E),\ F \to .id\}$

Construction of canonical LR(0) collection

A collection of sets of LR(0) items is called canonical LR(0) collection.

Algorithm:

Augment the grammar by adding production $S' \to S$

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

repeat the following until no more set of LR(0) item can be added to C.

for each I in C and each grammar symbol x

if goto(I,x) is not empty and not in C

add goto(I,x) to C.

Q. Compute the canonical LR(0) items collection for the following grammar.

$E \to E+T / T$

$T \to T*F / F$

$F \to (E) / id$

Sol^n

The augmented grammar is

$E' \to E$

$E \to E+T / T$

$T \to T*F / F$

$F \to (E) / id$

$I_0 = closure(\{E' \to .E\})$

$= \{ E' \to .E,$

Date ______
Page ______

$E \to .E+T,$
$E \to .T,$
$T \to .T*F,$
$T \to .F,$
$F \to .(E),$
$F \to .id$
$\}$

[. को पछाड़ि जे जे आएछ त्यो सबैको goto निकाल्ने]

$goto(I_0, E) = closure(\{E' \to E., E \to E.+T\})$
$= \{ E' \to E.,$
$E \to E.+T$
$\}$
$= I_1$

[goto मा closure निकाल्दा . लाई 1 shift गर्ने]

$goto(I_0, T) =$ ~~closure({E→T.,~~
$closure(\{E \to T., T \to T.*F\})$
$= \{ E \to T.,$
$T \to T.*F$
$\} = I_2$

$goto(I_0, F) = closure(\{T \to F.\})$
$= \{T \to F.\}$
$= I_3$

$goto(I_0, () = closure(\{F \to (.E)\})$
$= \{ F \to (.E),$
$E \to .E+T,$
$E \to .T,$
$T \to .T*F,$
$T \to .F,$
$F \to .(E),$
$F \to .id$
$\} = I_4$

Date
Page

goto (I_0, id) = closure ({F → id.})
= { F → id. }
= I_5

goto (I_1, +) = closure ({ E → E+.T })
= { E → E+.T,
T → .T*F,
T → .F,
F → .(E),
F → .id
} = I_6

goto (I_2, *) = closure ({ T → T*.F })
= { T → T*.F,
F → .(E),
F → .id
} = I_7

goto (I_4, E) = closure ({ F → (E.), E → E.+T })
= { F → (E.),
E → E.+T
} = I_8

goto (I_4, T) = closure ({E → T., T → T.*F})
= { E → T.,
T → T.*F
} = I_2

goto (I_4, F) = closure ({ T → F. })
= { T → F. }
= I_3

$goto(I_4, () = closure(\{F \to (.E)\})$

$= \{ F \to (.E),$

$E \to .E+T,$

$E \to .T,$

$T \to .T*F,$

$T \to .F,$

$F \to .(E),$

$F \to .id$

$\} = I_4$

$goto(I_4, id) = closure(\{F \to id.\})$

$= I_5$

$goto(I_6, T) = closure(\{E \to E+T., T \to T.*F\})$

$= \{ E \to E+T.,$

$T \to T.*F$

$\}$

$= I_9$

$goto(I_6, F) = closure(\{T \to F.\})$

$= I_3$

$goto(I_6, () = closure(\{F \to (.E)\})$

$= I_4$

$goto(I_6, id) = closure(\{F \to id.\})$

$= I_5$

$goto(I_7, F) = closure(\{T \to T*F.\})$

$= \{T \to T*F.\}$

$= I_{10}$

goto (I_7, ( ) = closure ({ F→(.E)}

= I_4

goto (I_7, id) = closure ({ F→id.})

= I_5

goto (I_8,)) = closure({ F→(E).})

= { F→(E).}

= I_{11}

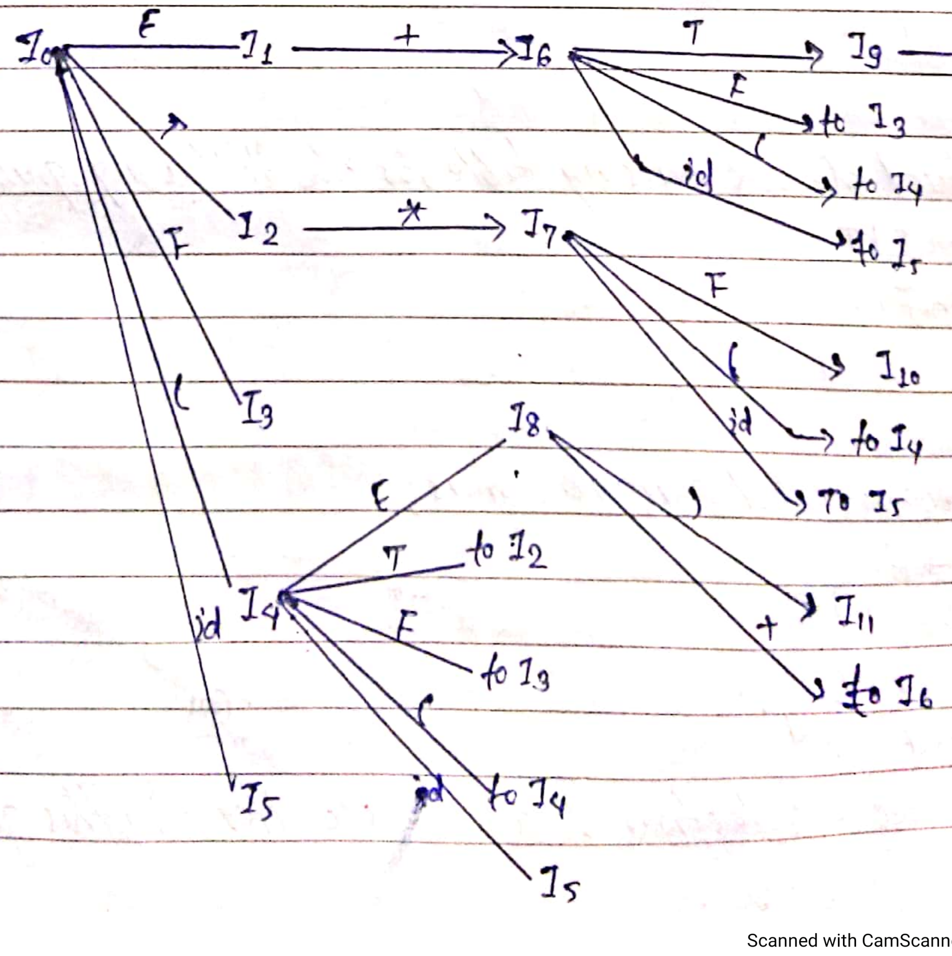
goto (I_8, +) = closure ({ E→E+.T})

= I_6

goto (I_9, *) = closure({ T→ T*.F})

= I_7

- The transition diagram of the goto function: DFA



Date ____
Page ____

Algorithm for constructing SLR parsing table:

1. Construct the canonical collection of sets of LR(0) items for augmented grammar G' starting from I_0 : closure({S'→.S})
2. Create the parsing action table as follows:
   i) If goto [I_i, a] = I_j then action [i, a] = shift j for each a ∈ T.
   ii) If A→α. is in I_i then for each b ∈ FOLLOW(A), action [i, b] = reduce A→α where A is not S'.
   iii) If S'→S. is in I_i, then action [i, $] = accept
3. Create the parsing goto table
   If goto [I_i, A] = I_j then goto [i, A] = j
4. All blank entries not defined by (2) & (3) are errors.

यदि यो Rule ले दुईवटा conflicting action generate गर्यो भने grammar SLR हुँदैन।

Example

1) Construct the SLR parsing table for the following grammar.
E → E+T | T
T → T*F | F
F → (E) | id

Soln

The augmented grammar G' is;
E' → E
E → E+T / T
T → T*F / F
F → (E) / id

The canonical collection of set of LR(0) items for this grammar are (Computed in previous eg)

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

I_1 : $E' \to E.$
$E \to E.+T$

I_2 : $E \to T.$
$T \to T.*F$

I_3 : $T \to F.$

I_4 : $F \to (.E)$
$E \to .E+T$,
$E \to .T$,
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_5 : $F \to id.$

I_6 : $E \to E+.T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

I_7 : $T \to T*.F$
$F \to .(E)$
$F \to .id$

I_8 : $F \to (E.)$
$E \to E.+T$

I_9 : $E \to E+T.$
$T \to T.*F$

I_{10} : $T \to T*F.$

I_{11} : $F \to (E).$

Now the action table ~~for above~~ SLR parsing for above grammar is

states	terminals id	+	*	(	)	$
0	shift 5			shift 4		
1		shift 6				Accept
2		Red. $E \to T$	shift 7		Red. $E \to T$	Red. $E \to T$
3		Red. $T \to F$	Red. $T \to F$		Red. $T \to F$	Red. $T \to F$
4	shift 5			shift 4		
5		Red. $F \to id$	Red. $F \to id$		Red. $F \to id$	Red. $F \to id$
6	shift 5			shift 4		
7	shift 5			shift 4		
8	~~shift 5~~	shift 6			shift 11	
9		Red. $E \to E+T$	shift 7		Red. $E \to E+T$	Red. $E \to E+T$
10		Red. $T \to T*F$	Red. $T \to T*F$		Red. $T \to T*F$	Red. $T \to T*F$
11		Red. $F \to (E)$	Red. $F \to (E)$		Red. $F \to (E)$	Red. $F \to (E)$

Date ____
Page ____

Now goto table for SLR

	E	T	F
0	1	2	3
1			
2			
3			
4	8	2	3
5			
6		9	3
7			10
8			
9			
10			
11			

Using Above parsing table:
String : id*id+id

stack & stack symbol state

stack	Input	Action	output
0	id*id+id $	shift 5	
0 id 5	*id+id $	Reduce F→id	F→id
0 F 3	*id+id $	Reduce T→F	T→F
0 T 2	*id+id $	shift 7	
0 T 2 * 7	id+id $	shift 5	
0 T 2 * 7 id 5	+id $	Reduce F→id	F→id
0 T 2 * 7 F 10	+id $	Red. T→T*F	T→T*F
0 T 2	+id $	Red. E→T	E→T
0 E 1	+id $	shift 6	
0 E 1 + 6	id $	shift 5	
0 E 1 + 6 id 5	$	Red. F→id	
0 E 1 + 6 F 3	$	Red. T→F	
0 E 1 + 6 T 9	$	Red E→E+T	
0 E 1	$	accept	

2) G:

$S \to L = R$

$S \to R$

$L \to * R$

$L \to id$

$R \to L$

Compute the LR(0) item sets and SLR parsing table.

Sol^n

The augmented grammar G' is:

$S' \to S$

$S \to L = R$

$S \to R$

$L \to * R$

$L \to id$

$R \to L$

$I_0 = closure(\{S' \to .S\})$

$= \{ S' \to .S,$

$S \to .L = R,$

$S \to .R,$

$L \to .* R,$

$L \to .id,$

$R \to .L$

$\}$

$goto(I_0, S) = closure(\{S' \to S.\})$

$= \{S' \to S.\}$

$= I_1$

$goto(I_0, L) = closure(\{S \to L.= R, R \to L.\}) = \{S \to L.= R, R \to L.\} = I_2$

$goto(I_0, R) = closure(\{S \to R.\}) = \{S \to R.\} = I_3$

$goto(I_0, *) = closure(\{L \to *.R\})$
$= \{ L \to *.R,$
$R \to .L,$
$L \to .*R,$
$L \to .id$
$\} = I_4$

$goto(I_0, id) = closure(\{L \to id.\}) = \{L \to id.\} = I_5$

$goto(I_2, =) = closure(\{S \to L=.R\})$
$= \{ S \to L=.R,$
$R \to .L,$
$L \to .*R,$
$L \to .id$
$\} = I_6$

$goto(I_4, R) = closure(\{L \to *R.\}) = \{L \to *R.\} = I_7$

$goto(I_4, L) = closure(\{R \to L.\}) = \{R \to L.\} = I_8$

$goto(I_4, *) = closure(\{L \to *.R\}) = I_4$

$goto(I_4, id) = closure(\{L \to id.\}) = I_5$

$goto(I_6, R) = closure(\{S \to L=R.\}) = \{S \to L=R.\} = I_9$

$goto(I_6, L) = closure(\{R \to L.\}) = I_8$

goto (I_6 , *) = closure ({ L → *.R }) = I_4

goto (I_6 , id) = closure ({ L → id. }) = I_5

The canonical collection of set of LR(0) items for this grammar are:

I_0 : S' → .S,
S → .L=R,
S → .R,
L → .*R,
L → .id,
R → .L

I_1 : S' → S.

I_2 : S → L.=R,
R → L.

I_3 : S → R.

I_4 : L → *.R,
R → .L,
L → .*R
L → .id

I_5 : L → id.

I_6 : S → L=.R
R → .L
L → .*R
L → .id

I_7 : L → *R.

I_8 : R → L.

I_9 : S → L=R.

Now, Follow sets of non-terminals are

Follow (S) = { $ }   Follow (L) = { =, $ },   Follow (R) = { $, = }

SLR Parsing table:

States	Action Table				Goto table		
	id	=	*	$	S	L	R
0	shift 5		shift 4		1	2	3
1				Accept			
2		shift 6 Red. R→L		Red. R→L			
3				Red. S→R			
4	shift 5		shift 4			8	7
5		Red. L→id		Red. L→id			
6	shift 5		shift 4			8	9
7		Red. L→*R		Red. L→*R			
8		Red. R→L		Red. R→L			
9				Red. S→L=R			

Here, the entry on action table for action[2, =] is multiply defined, one is shift operation and another is reduce operation.

- The grammar is not ambiguous but there is shift-reduce conflict.

LR(0) मा आउने Ambiguity लाई हटाउन LR(1) use गर्ने।

Q. Compute SLR parsing table for grammar G =

$S \rightarrow CC$

$C \rightarrow cC \mid d$

Soln

The augmented grammar G' is

$S' \rightarrow S$

$S \rightarrow CC$

$C \rightarrow cC \mid d$

$I_0 = closure(\{S' \rightarrow .S\}) = \{S' \rightarrow .S, S \rightarrow .CC, C \rightarrow .cC, C \rightarrow .d\}$

$goto(I_0, S) = closure(\{S' \rightarrow S.\}) = \{S' \rightarrow S.\} = I_1$

$goto(I_0, C) = closure(\{S \rightarrow C.C\}) = \{S \rightarrow C.C, C \rightarrow .cC, C \rightarrow .d\} = I_2$

$goto(I_0, c) = closure(\{C \rightarrow c.C\}) = \{C \rightarrow c.C, C \rightarrow .cC, C \rightarrow .d\} = I_3$

$goto(I_0, d) = closure(\{C \rightarrow d.\}) = \{C \rightarrow d.\} = I_4$

$goto(I_2, C) = closure(\{S \rightarrow CC.\}) = \{S \rightarrow CC.\} = I_5$

$goto(I_2, c) = closure(\{C \rightarrow c.C\}) = I_3$

Date ______
Page ______

$goto(I_2, d) = closure(\{C \to d.\}) = I_4$

$goto(I_3, C) = closure(\{C \to cC.\}) = \{C \to cC.\} = I_6$

$goto(I_3, c) = closure(\{C \to c.C\}) = I_3$

$goto(I_3, d) = closure(\{C \to d.\}) = I_4$

The canonical collection of set of LR(0) items are

I_0 : $S' \to .S$
$S \to .CC$
$C \to .cC$
$C \to .d$

I_1 : $S' \to S.$

I_2 : $S \to C.C$
$C \to .cC$
$C \to .d$

I_3 : $C \to c.C$,
$C \to .cC$
$C \to .d$

I_4 : $C \to d.$

I_5 : $S \to CC.$

I_6 : $C \to cC.$

FOLLOW(S) = {\$} , FOLLOW(C) = {c, d, \$}

SLR Parsing table:

	Action table			goto table	
state	c	d	\$	S	C
0	shift 3	shift 4		1	2
1			Accept		
2	shift 3	shift 4			5
3	shift 3	shift 4			6
4	Red. $C \to d$	red. $C \to d$	Red. $C \to d$	~~Red. $S \to CC$~~	
5			Red. $S \to CC$	~~Red. $S \to CC$~~	
6	Red. $C \to cC$	Red. $C \to cC$	Red $C \to cC$		

The Core of set of LR(1) items is the set of their 1st components.

Date ____
Page ____

LR(1) Parser / Grammar

- SLR parser is so simple and only represent the small group of grammar.
- LR(1) parsing uses look-ahead to avoid unnecessary conflicts in parsing table.

Any LR(1) item is of the form
$(A \rightarrow .\alpha B\beta, a)$ where $A \rightarrow .\alpha B\beta$ is called the core and 'a' is called lookahead.

*** Computation of closure for LR(1) items:** closure(I)

For each item of the form $[A \rightarrow \alpha.B\beta, a]$ in I,
for each production $B \rightarrow D$ in G' and for each terminal 'b' in FIRST(βa) do
add $[B \rightarrow .D, b]$ to I if it is not already in I.

*** goto in LR(1):**

For each item of the form $[A \rightarrow \alpha.X\beta, a] \in I$
$goto[I, X] = closure([A \rightarrow \alpha X.\beta, a])$

→ I_0 for LR(1) item sets is computed as
$I_0 = closure([S' \rightarrow .S, \$])$

*** Construction of Canonical LR(1) collection / LR(1) DFA:**

Algorithm:

1. Augment the grammar with production $S' \to S$
2. Start with $C = \{closure(\{S' \to .S, \$\})\}$ where S is start symbol.
3. Repeat the following until no more set of LR(1) item can be added to C.

    for each I in C and each grammar symbol X
        if goto(I,X) is not empty and not in C
            add goto(I,X) to C.

Example

1) Compute the LR(1) collection of items from the following grammar.

$S \to CC$

$C \to eC \mid d$

Sol^n

The augmented grammar is

$S' \to S$	FIRST(S) = {e, d}
$S \to CC$	FIRST(C) = {e, d}
$C \to eC \mid d$	

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

$= \{ [S' \to .S, \$],$

$[S \to .CC, \$],$

$[C \to .eC, e/d],$

$[C \to .d, e/d]$

$\}$

$goto(I_0, S) = closure([S' \to S., \$])$

$= \{[S' \to S., \$]\}$

$= I_1$

$goto(I_0, C) = closure([S \to C.C, \$])$

$= \{ [S \to C.C, \$],$
$[C \to .eC, \$],$
$[C \to .d, \$]$
$\} = I_2$

$goto(I_0, e) = closure([C \to e.C, e|d])$
$= \{ [C \to e.C, e|d],$
$[C \to .eC, e|d],$
$[C \to .d, e|d]$
$\} = I_3$

$goto(I_0, d) = closure([C \to d., e|d])$
$= \{ [C \to d., e|d] \}$
$= I_4$

$goto(I_2, C) = closure([S \to CC., \$])$
$= \{ [S \to CC., \$] \}$
$= I_5$

$goto(I_2, e) = closure([C \to e.C, \$])$
$= \{ [C \to e.C, \$],$
$[C \to .eC, \$],$
$[C \to .d, \$]$
$\} = I_6$

$goto(I_2, d) = closure([C \to d., \$])$
$= \{ [C \to d., \$] \}$
$= I_7$

$goto(I_3, C) = closure([C \to eC., e|d])$
$= \{ [C \to eC., e|d] \}$
$= I_8$

$goto(I_3, e) = closure([C \rightarrow e.C, e/d]) = I_3$

~~$= \{ [C \rightarrow e.C, e/d],$~~

~~$[C \rightarrow .eC$~~

$goto(I_3, d) = closure([C \rightarrow d., e/d]) = I_4$

$goto(I_6, C) = closure([C \rightarrow eC., \$])$

$= \{ [C \rightarrow eC., \$] \}$

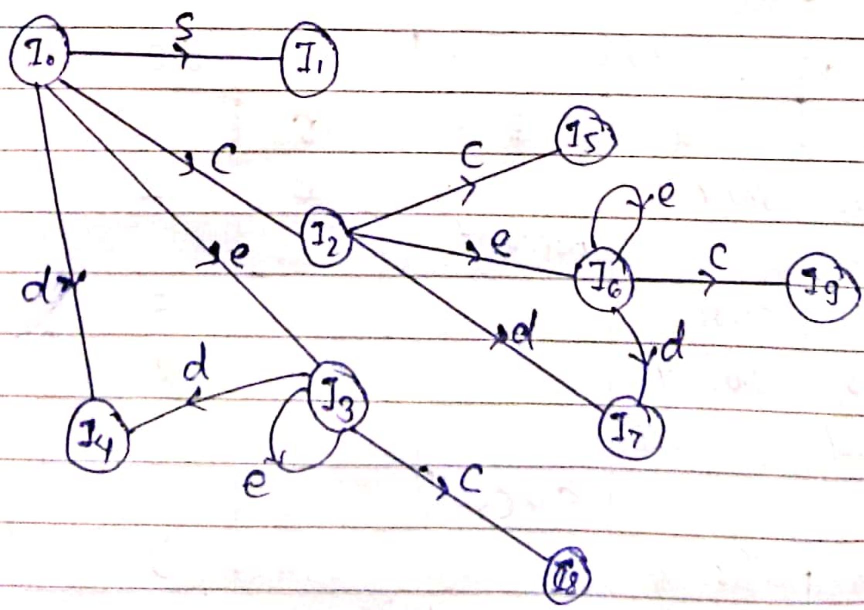
$= I_9$

$goto(I_6, e) = closure([C \rightarrow e.C, \$]) = I_6$

$goto(I_6, d) = closure([C \rightarrow d., \$]) = I_7$

The canonical ~~set~~ collection of set of LR(1) items are: write all the states $I_0 - I_9$

DFA :



Goto graph

Date
Page

Construction of the LR(1) parsing table (Algorithm)

1. Given a grammar G, make augmented grammar G' by adding production $S' \to S$ where S is start variable.
2. Construct the canonical collection of sets of LR(1) items for G'. Starting from $I_0 = closure([S' \to .S, \$])$. $C = \{I_0, I_1, I_2, .. I_n\}$
3. Create a parsing action table as follows:
   i) If $goto[I_i, a] = I_j$ then $action[i, a] = shift\ j$ for each $a \in T$
   ii) If $A \to \alpha., a$ is in I_i then $action[i, a] = reduce\ A \to \alpha$ where $A \neq S'$
   iii) If $[S' \to S., \$]$ is in I_i, then $action[i, \$] = accept$
4. Create the parsing goto table
   For any non-terminals A, If $goto[I_i, A] = I_j$ the $goto[i, A] = j$
5. All blank entries not defined by (3) & (4) are errors.

Example

LR(1) parsing table for above Eg.

States	Action table			Goto table	
	e	d	$	S	C
0	shift 3	shift 4		1	2
1			Accept		
2	shift 6	shift 7			5
3	shift 3	shift 4			8
4	Red. C→d	Red. C→d			
5			Red. S→CC		
6	shift 6	shift 7			9
7			Red. C→d		
8	Red. C→eC	Red C→eC			
9			Red. C→eC		

Date
Page

LALR(1) Grammar

→ LALR (lookahead LR) grammar is an intermediate grammar between the SLR and LR(1) grammar.

→ A typical programming language generates thousand of states for canonical LR parsers while they generate only hundreds of states for LALR parser.

→ LALR(1) parser combines two or more LR(1) sets (whose core parts are same) into a single state to reduce the table size.

e.g.

$I_1 : L \to id., =$ $I_{12} : L \to id., =/\$$

$I_2 : L \to id., \$$

Constructing LALR parsing table:

1. Given a grammar G, construct a augmented grammar G' by adding production $S' \to S$ where S is start varible.
2. Construct the canonical collection of sets of LR(1) items for G'. $C = \{I_0, I_1, I_2, \ldots, I_n\}$
3. For each core present, find all sets having the same core; & replace these by their union.
4. Create the parsing tables (action & goto tables) same as the construction of the parsing tables of LR(1) parser.
5. If J is the union of one or more sets of LR(1) items i.e. $J = I_1 \cup I_2 \cup \ldots \cup I_k$, then the cores of goto (I_1, X), goto (I_2, X), ..., goto(I_k, X) must be same as all of them have same core. Let k be the union of all sets of items having the same core as goto $[I_1, X]$. Then goto $[J, X] = k$.

→ If no conflict is introduced, the grammar is LALR(1) grammar.

Date ____
Page ____

Example

$S \to CC$

$C \to eC \mid d$

The augmented grammar is

$S' \to S$

$S \to CC$

$C \to eC \mid d$

The canonical collection of set of LR(1) items are: (Computed in previous e.g.)

I_0: { ($S' \to .S, \$$),
($S \to .CC, \$$),
($C \to .eC, e/d$),
($C \to .d, e/d$)
}

I_1: ($S' \to S., \$$)

I_2: ($S \to C.C, \$$),
($C \to .eC, \$$),
($C \to .d, \$$)

I_3: ($C \to e.C, e/d$),
($C \to .eC, e/d$),
($C \to .d, e/d$)

I_4: ($C \to d., e/d$)

I_5: ($S \to CC., \$$)

I_6: ($C \to e.C, \$$),
($C \to .eC, \$$),
($C \to .d, \$$)

I_7: ($C \to d., \$$)

I_8: $C \to eC., e/d$

I_9: ($C \to eC., \$$)

In the collection of LR(1) items, I_3 and I_6, I_4 and I_7, I_8 and I_9 have same core items. So performing union operation, the items for LALR will be as

Date
Page

I_0 : (S' → .S, $),
(S → .CC, $),
(C → .eC, c|d),
(C → .d, e|d)

I_1 : (S' → S., $)

I_2 : (S → C.C, $),
(C → .eC, $),
(C → .d, $)

I_{36} : (C → e.C, e|d|$),
(C → .eC, e|d|$),
(C → .d, e|d|$)

I_{47} : (C → d., e|d|$)

I_5 : (S → CC., $)

I_{89} : (C → eC., e|d|$)

So The LALR parsing table:

states	Action table			goto table	
	e	d	$	S	C
0	shift 36	shift 47		1	2
1			Accept		
2	shift 36	shift 47		5	
36	shift 36	shift 47			89
47	Red. C → d	Red. C → d	Red. C → d		
5			Red. S → CC		
89	Red. C → eC	Red. C → eC	Red. C → eC		

Q.//

S → L = R
S → R
L → *R
L → id
R → L

Construct LALR parsing table.

Date ____
Page ____

Kernel and Non-kernel items

*** Kernel items:**

The initial item $S' \to .S$ and all the other items those with no dot (.) at the begining of R.H.S are called the kernel items.

E.g.

For any production $A \to \alpha B \beta$

kernes → $A \to \alpha . B \beta$, $A \to \alpha B . \beta$, $A \to \alpha B \beta$

*** Non-kernel items:**

All items except first item $(S' \to .S)$ with dot at the begining of R.H.S are called the non-kernel items.

E.g.

$A \to . \alpha B \beta$

Q: Compute the kernel items for LR(0) for the following grammar.

$S \to CC$

$C \to bC \mid d$

Soln

First of all given grammar लाई augmented बनाउने अनि Canonical Collection of sets of LR(0) items calculate गर्ने । After that According to defn Canonical collection set of LR(0) बाट kernel choose गर्ने ।

Q) Consider the augmented grammar

$S' \rightarrow S$

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow * R$

$L \rightarrow id$

$R \rightarrow L$

The kernel of the set of LR(0) item are:

$I_0 = \{ S' \rightarrow .S \}$

$I_1 = \{ S' \rightarrow S. \}$

$I_2 = \{ S \rightarrow L. = R, R \rightarrow L. \}$

$I_3 = \{ S \rightarrow R. \}$

$I_4 = \{ L \rightarrow *.R \}$

$I_5 = \{ L \rightarrow id. \}$

$I_6 = \{ S \rightarrow L = .R \}$

$I_7 = \{ L \rightarrow *R. \}$

$I_8 = \{ R \rightarrow L. \}$

$I_9 = \{ S \rightarrow L = R. \}$

Error Recovery in Predictive Parsing (
[illegible] predictive parsing को part)

An error may occur in the predictive parsing due to following reasons

- If the terminal symbol on the top of stack does not match with the current input symbol.
- If the top of stack is non-terminal A, the current input symbol is a, and the entry for M[A, a] in parsing table is empty.

In an Error case parser should do:

- A parser should try to determine that the error has occurred as soon as possible. (Error find out गर्ने [illegible] parser को actual location of error case गर्ने [illegible])

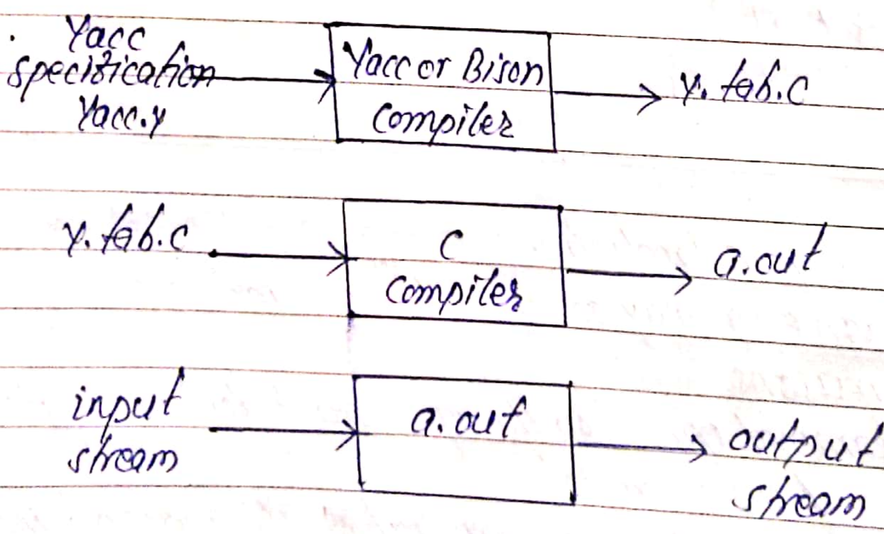
Scanned with CamScanner

- A suitable and comprehensive message should be reported. "Missing semicolon on line 36" is helpful, "unable to shift in state 425" is not.
- After an error has occurred, the parser must pick a reasonable place to resume the parse.
- A parser should avoid cascading errors.

* **Parser Generator** Yacc / Bison

Yacc - Yet another compiler compiler
Bison - A newer version of Yacc

Yacc/Bison is a general (ALR(1)) parser generator where grammar rules can be encoded into yacc syntax and can be compiled to generate equivalent c-code.



- Source file extension .y.

Date
Page

Stages in writing Bison program :

1. Formally specify the grammar in a form recognized by Bison.
2. Write a lexical analyzer to process input and pass tokens to the parser.
3. Write a controlling function that calls the Bison produced parser.
4. Write error-reporting routines.

Syntax / specification for Yacc / Bison

A bison specification consists of four parts :

```
%{
        C-declarations
%}
    Yacc/Bison declaration
%%
        Grammar rules and associated actions
%%
        Additional C-codes
```

For Yacc / bison specification general conventions

- Terminal are represented by upper case words. E.g. NUM, DIGIT or ~~only~~ any single character lik '+', ')', '(', 'x' etc.
- Non terminals word in lowercase e.g expr, oper
- Token type names are declared with % token
  E.g. % token NUM
       % token '+'

- For precedence with left or right association -
  - %left '+' '-' - Declares as left association
  - %right '^' - Right association
  - %union declares the collection of data type for any semantic value.

```
%union { double val;
         int val1;
       }
```

- For Grammar rules

```
non-terminals : <alternative 1>   {action 1}
              | <alternative 2>   {action 2}
              | <alternative 3>   {action 3}
                  ⋮                   ⋮
              | <alternative n>   {action n}
```

- For main() function in last section, the equivalent c-function for grammar rules and actions, yyparse() is called that is the actual c-code for above specification.

E.g.
Grammar

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid DIGIT$

(Bracket मा हुने expr छ भने त्यो गर्नेको त्यसको value हुन्छ (expr))

$E \rightarrow E + T$
$\$\$ \quad \$_1 \quad \$_2 \quad \$_3$

Action
→ L.H.S लाई double $ ले represent &
→ R.H.S लाई $_1, $_2 आदि ले
If function
$$ = add($1, $3);

Grammar rules with semantic action for this grammar

```
expr : expr '+' term        {$$ = $1 + $3;}
     | term
     ;
term : term '*' factor      {$$ = $1 * $3;}
     | factor
     ;
```

```
factor : '('expr')'      {$$ = $2;}
        | DIGIT
        ;
```

simple calculator using Yacc:

$P \rightarrow PE \mid \epsilon$
$E \rightarrow Num \mid OE_1E_2$
$op \rightarrow + \mid - \mid * \mid /$

```
%{
  #include <stdio.h>
%}
%token NUM
%%
Prog : /*empty*/
     | prog expr   {printf("%d\n", $2);}
     ;
expr : NUM     { $$ = $1; }
     | '(' op expr expr ')'  {$$ = evaluate ($2,$3,$4);
                               printf("%d = (%c%d%d)\n", $$,$2,$3,$4)
                              ;}
     ;
op : '+'    {$$ = $1;}
   | '-'    {$$ = $1;}
   | '*'    {$$ = $1;}
   | '/'    {$$ = $1;}
%%
int evaluate p(char op, int first, int second)
{
  switch (op)
  {
     Case '+' : return (first+second);
                break;
     Case '-' : return (first-second);
                break;
```

Date ____
Page ____

```
        case'x' : return (first * second);
                   break;
        case'/' : return (first /second);
                   break;
     }
  }
  yyerror()
  {
     fprint (stderr, "An error accured!");
     return 0;
  }
  main()
  {
     yyparse();
  }
%{
     # include <prefix.tab.h>
     # include <ctype.h>
%}
 NUM    [0-9][0-9]*
%%
 NUM      {yyval=atoi(yytext); return NUM;}
 [\t]+
   .        {yyval = yytext[0], return yyval; }
%%
```