3.1
08/02/2016

UNIT- III

SYNTAX ANALYSIS:

Parsing:

Introduction:

C-program : Block of statements
↓
group of expressions
↓
Tokens

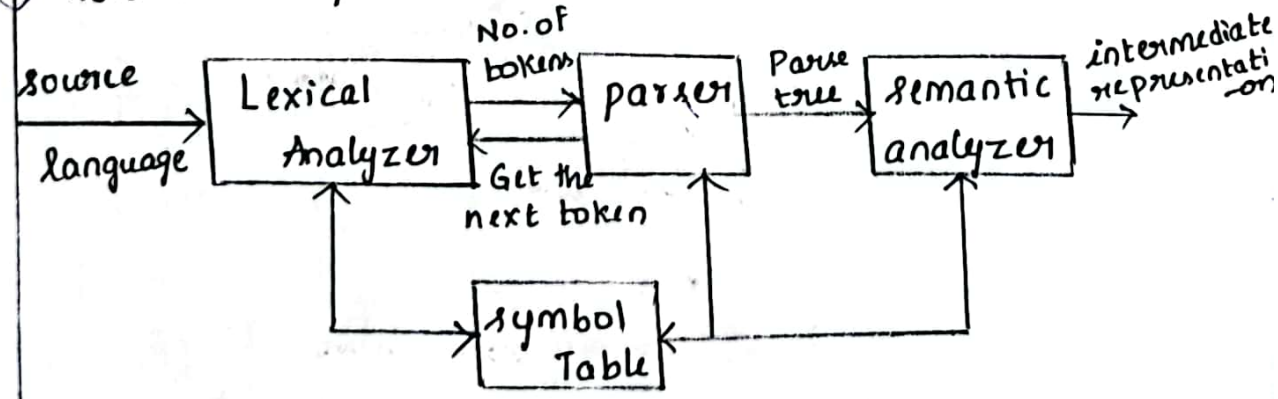
Syntatic representation of Language is explained in

* Backus Naur form (BNF)
* Context Free Grammar (CFG)

Usage of Grammar in Syntatic Language:

* Errors identified easily.
* Easy understanding
* Efficient parsers construction.

Role of the parser:



* Identifies errors and if possible correct it.

I/p : stream of tokens

O/p : Parse Tree.

Usage of Parser:

* constructs syntax tree.
* identifies error and if possible error recovery.

Issues in the parser:

* Parsing can't be done where redeclaration of variable can't be identified.
* Usage of variable before declaration can't be identified.
* Mismatch of data cannot be identified.

09/02/2016

Parser:

Accepts I/p as tokens and produces o/p as parse tree and produce syntax errors.

Types of Parser:

* Top - down parser :
  - → predictive parser.
  - → LL(1) parser
  - → Recursive descent parser.
* Bottom - up parser :
  - → shift reduce
  - → Operator precedence
  - → LR parser.
  - → SLR parser
  - → LALR parser
  - → CLR parser.

→ Building tree process from root to leaf : Top-down process

→ Building tree process from leaf to root : Bottom-up process.

* Syntatic expression (representation) are identified using two grammars
  - → LL grammar
  - → LR grammar

✓ Syntax Error Handling:.

→ Lexical error (eg: Misspelling of Keyword).

→ Syntatic error (eg: Missing of semicolon).

→ Semantic error (eg: Type mismatch).

→ Logical error (can't be identified by compilers).

* These errors should be identified and handled properly.

* To recover the error within specific time, error recovery strategies are used.

✓ Error Recovery Strategies:

(i) Panic mode [Eg: '(' scans till another ')']

* Scans left to right and discovers the errors.

* After discovering the parser discards all the input symbol one at a time until it finds the synchronising token.

* The synchronising tokens are usually delimited such as ; (or) and whose role in the source program is clear.

(ii) Phrase level [eg: Replacing ':' by ';']

Keyword – Local correction

On discovering an error, it does local correction.

(iii) Error production: [eg: try block, catch block]

Connects common (or) new errors in the production. If an error production is used by the parser, we can generate appropriate error diagnostics to indicate that the erroneous, that construct has been recognized.

(iv) Global correction:

Common steps (or) algorithms are used to identify and recover the error.

If 'x' is an invalid string with the errors, then the algorithm will produce the parse tree for the string. In this case the string can be corrected using the parse tree of the string 'y'.

10/02/2016

CONTEXT FREE GRAMMAR:

* CFG is a quadruple that consists of terminals, non-terminals, start symbol and production.

* The syntax of programming language constructs can be described by CFG or BNF.

TERMINALS:

* Basic symbols to form a set of strings
* Terminals are also called as tokens.

Non-terminals:

* A syntatic variable that denotes the set of strings.

* Made of terminals (or) non-terminals.

Start Symbol:-

One non-terminal in the grammar is selected as start symbol (or) distinguished symbol usually it is 's'.

Production:

Otherwise called as rewriting rules in which terminals and non-terminals are combined to form string.

Eg: $A \rightarrow bcd$

(or)

$A ::= bcd$

Example for CFG:

$expr \rightarrow expr\ op\ expr$

$expr \rightarrow (expr)$

$expr \rightarrow -expr$

$expr \rightarrow id$

$op \rightarrow +$

$op \rightarrow -$

$op \rightarrow *$

$op \rightarrow /$

$op \rightarrow \uparrow$

9 terminals : id, +, -, *, /, ↑, (,).

2 Non-terminals : expr, op.

∴ Expr → Start symbol because it generates op and vice versa Not possible.

Notational conventions:

Terminals: (a-z)

* Lowercase letters, special symbols (., (,) + -), digits (0-9), Keywords (if, Else), Bold Face strings such as (if, id, ...).

Non Terminals:

* Uppercase letters (A-z), Lower case italic letters (Expr).

Start Symbol:

* Usually 'S' denotes the start symbol.

Production:

$A \to \alpha$

A → Non Terminal

α ⇛ Terminal.

Here A produces a 'α' which is a terminal.

Grammar symbols:

* upper case letters (X, Y, Z)
* Lower case letters (u, v, z)
* Lower case greek letters (α, β, γ)
* $A \to \alpha_1$, $A \to \alpha_2$, $A \to \alpha_3$ ..... $A \to \alpha_K$;

$\alpha_1 | \alpha_2 | \alpha_3 \ldots\ldots | \alpha_K \rightarrow$ Alternative for 'A'.

eg:-

$E \rightarrow EAE | (E) | -E | id$

$A \rightarrow + | - | / | * | \uparrow$

* Number of productions : 9

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow E / E$

$E \rightarrow E * E$

$E \rightarrow E \uparrow E$

$E \rightarrow (E)$

$E \rightarrow - E$

$E \rightarrow id$.

Derivations, Reduction and Parse tree:

* Derivations:

Process of generating a valid string with a help of grammar is called derivations.

* Reduction:

The Process of Recognizing the valid string using grammar will be called as Reduction.

* Parse tree:

Graphical Representation of both derivation and Reduction.

Derivations:

- Derivation symbol is " $\Rightarrow$ "

Eg:

$A \rightarrow \gamma$

$\alpha A \beta \Rightarrow \alpha \gamma \beta$

Here "$\alpha\gamma\beta$" is Derived From the production

$A \rightarrow \gamma$.

$\overset{*}{\Rightarrow}$: Deriving zero (or) more times

$\overset{+}{\Rightarrow}$: Deriving one (or) more times.

11/02/2016

Eg: $E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id$

Derive: $-(id+id)$

$E \Rightarrow -E$
$\Rightarrow -(E)$
$\Rightarrow -(E+E)$
$\Rightarrow -(id+E)$
$\Rightarrow -(id+id)$

Types of Derivation:-

* Left most derivation - Replacing the string from left hand side
* Right most derivation - Replacing the string from right hand side

Example:

Leftmost

$E \Rightarrow -E$
$E \Rightarrow (E)$
$\Rightarrow -(E+E)$
$\Rightarrow -(id+E)$
$\Rightarrow -(id+id)$

Rightmost

$E \Rightarrow -E$
$\Rightarrow -(E)$
$\Rightarrow -(E+E)$
$\Rightarrow -(E+id)$
$\Rightarrow -(id+id)$

Reduction:

Reduction symbol is '$\rightarrow$' Eg: $A \rightarrow \gamma$

Replacing γ by A is called Reduction

$\xrightarrow{*}$: Reducing 0 (or) more times

$\xrightarrow{+}$: Reducing 1 (or) more times

Eg: $E \to E+E \mid E*E \mid (E) \mid -E \mid id$

(i) check $-(id+id)$ valid (or) not :

$E \to -(id+id)$
$\to -(E+id)$
$\to -(E+E)$
$\to -(E)$
$\to -E$

$\boxed{E \to E}$ valid string.

$E \to -(id/id)$
$\to -(E/id)$
$E \to -(E/E) \to$ Not a valid string Because Reduction not possible.

Types of Reduction:

* Left - Most Reduction
* Right - Most Reduction

Left most

$E \to -(id+id)$
$E \to -(E+id)$
$E \to -(E+E)$
$E \to -(E)$
$E \to -E$
$E \to E$

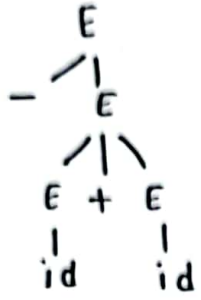
Right most

$E \to -(id+id)$
$E \to -(id+E)$
$E \to -(E+E)$
$E \to -(E)$
$E \to -E$
$E \to E$

→ Parse tree

Eg: $E \to (E) \mid E*E \mid E+E \mid -E \mid id$; construct parse tree

for (i) $-(id+id)$ (ii) $-(id-id)$

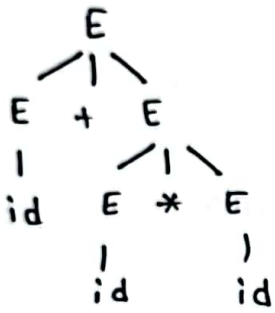


valid

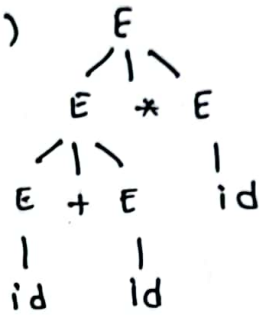
invalid.

ii) $id + id * id$ → construct parse tree:

(i)



ii)



Derivations:

(i)
$E \Rightarrow E+E$
$\Rightarrow id+E$
$\Rightarrow id+E*E$
$\Rightarrow id+id*E$
$\Rightarrow id+id*id$

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

Ambiguity:

If same statement produces two different parse tree is called Ambiguity.

Dangling else grammar:

Which parse tree we want is selected and others are registered.

Here disambiguous rules can be used to eliminate ambiguity.

Writing a grammar:

Regular Expression Vs CFG

Verifying the language generated by a Grammar

Eliminating Ambiguity

Elimination of left Recursion

Left factoring.

Regular Expression Vs CFG:

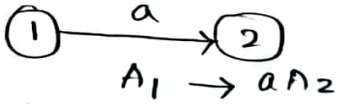
* The Algorithm for constructing the Grammar from NFA.

* For each state i of the NFA create a non-terminal A_i.

begin

* If the state 'i' has transition to state 'j' on symbol 'α' then introduce the production.

$$A_i \rightarrow \alpha A_j.$$



$A_1 \rightarrow aA_2$

* if state 'i' goes to state 'j' on input 'ε' then introduce the production

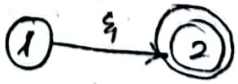
$$A_i \rightarrow A_j$$

$A_1 \rightarrow A_2$.

end

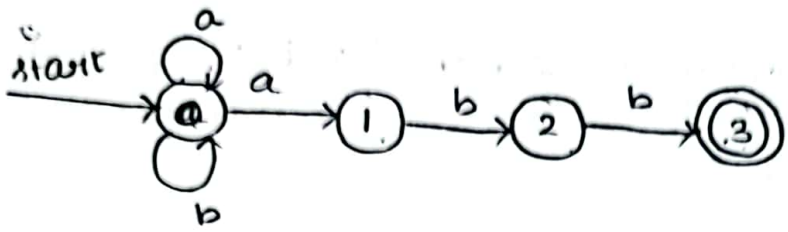
* if 'i' is an accepting state, then introduce

$$A_i \rightarrow \varepsilon$$



$A_2 \rightarrow \varepsilon$.

Construct NFA for (a/b)* abb



The productions are

$A_0 \to aA_1$
$A_0 \to aA_0$
$A_0 \to bA_0$
$A_1 \to bA_2$
$A_2 \to bA_3$
$A_3 \to \varepsilon$

(or)

$A_0 \to aA_1 | aA_0 | bA_0$
$A_1 \to bA_2$
$A_2 \to bA_3$
$A_3 \to \varepsilon$

The string "aabaabb" is obtained by using the grammar above.

$A_0 \to aA_0$
$\to aaA_0$
$\to aabA_0$
$\to aabaA_1$
$\to aababA_2$
$\to aababbA_3$
$A_0 \to aababb$

Verifying a language generated by a grammar:

Steps followed:

→ We must show that every string by 'G' is in language 'L'.

→ Every string is in language 'L' can be generated by grammar 'G'.

3.13

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

condition: every string generated by 's' is balanced

$S \rightarrow E$

$S \rightarrow (S)S$

Here 's' can be considered as 'x' and 'y'.

$S \rightarrow (x)y$

Every balanced string are derivable from 'x'.

Derive the string (x)y and ((x)y)

$S \Rightarrow (S)S$

$S \Rightarrow (x)y$

$S \Rightarrow (S)S$

$\Rightarrow ((S)S)S$

$\Rightarrow ((x)S)S$

$\Rightarrow ((x)y)S$

$\Rightarrow ((x)y)\varepsilon$

$\Rightarrow ((x)y)$.

Eliminating Ambiguity:

→ consider dangling-else grammar.

Production:

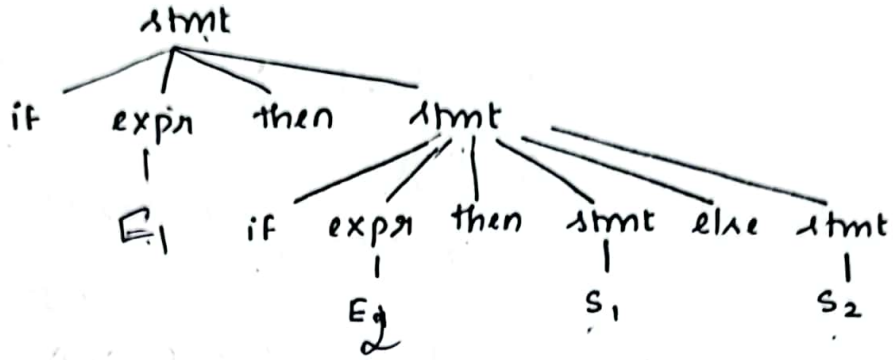
stmt → if expr then stmt

| if expr then stmt else stmt

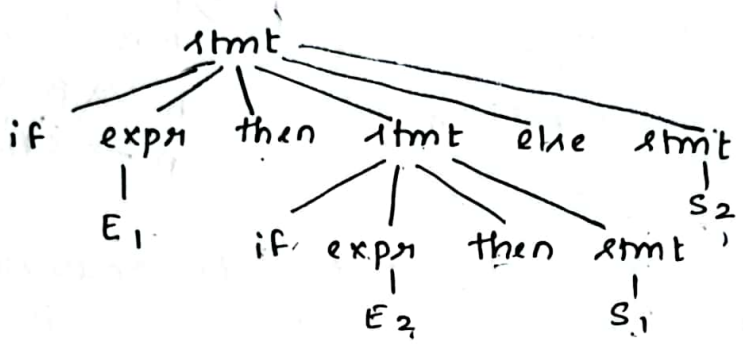
| other

if E_1 then if E_2 then S_1 else S_2, generate the parse tree for this.

TREE 1:



TREE 2:



Productions :-

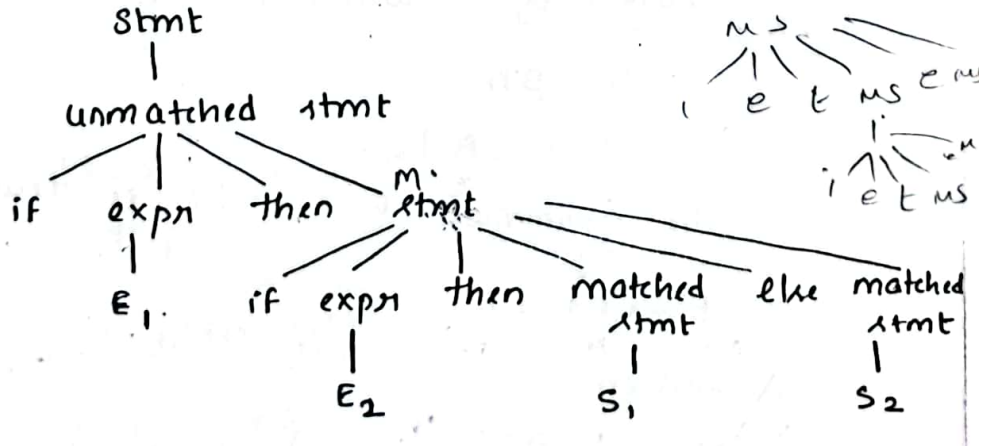
Stmt → matched stmt / unmatched stmt.

Matched stmt → if expr then Matched stmt else matched stmt.

Unmatched stmt → if expr then stmt | if expr then matched stmt else unmatched stmt.

PARSE TREE:

if E_1 then if E_2 then S_1 else S_2.



3.15 ✓ Elimination of Left Recursion:

$A \to A\alpha \mid \beta$ ($\beta \neq 0$)

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$ } rewriting as

$E \to E + T \mid T$

Here $E \to E+T$

E + T + T

E + T + T + T

⋮ [leads to left recursion]

(ii) Subdivision eg:

$S \to Aa \mid b$

$A \to Ac \mid Sd \mid e$

$A \to A\alpha_1 \mid \beta_1 \mid \beta_2$

$A \to \beta_1 Sd A' \mid A'$

$A' \to cA' \mid \varepsilon$.

Goes on so called as left recursion because replacing the 'E' by the left side.

Rules for occuring left recursion:

When non-terminal in both sides (same letter)

If any statement has the left recursion

$A \to A\alpha \mid \beta$; then the left recursion can be eliminated by rewritting the grammar as,

$A \to \beta A'$

$A' \to \alpha A' \mid \varepsilon$

The grammar $E \to E+T \mid T$ is rewritting as,

$E \to \underset{A}{E}+\underset{\alpha}{T} \mid \underset{\beta}{T}$

Rewritten

i) $A \to \beta A'$ ii) $A' \to \alpha A' \mid \varepsilon$

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

* $T \rightarrow T * F / F$

(i) $A \rightarrow \beta A'$

$T \rightarrow FT'$

(ii) $A' \rightarrow \alpha A' | \varepsilon$

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

* $F \rightarrow (E) | id$

$E \rightarrow E+T | E+t | Eu | a | b | c$

$E \rightarrow aE' | bE' | cE'$

$E' \rightarrow +TE' | tE' | uE' | \varepsilon$

$A \rightarrow A\alpha_1 | A\alpha_2 | A\alpha_n | \beta_1 | \beta_2 | \beta_n$

$A \Rightarrow \beta_1 A' | \beta_2 A' | \beta_n A'$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \alpha_3 A' | \alpha_n A' | \varepsilon$

The productions are,

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) | id$

$S \rightarrow Aa | b$ or $A \rightarrow Ac | Sd | \varepsilon$

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

The above set of productions are not having left recursion.

Left factoring:-

Rule: If any statement is of the form

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$.

$S \rightarrow Aa | b$

$A \rightarrow bdA' | A'$

$A' \rightarrow aA' | adA' | \varepsilon$

then we can choose the appropriate alternative by rewriting the grammar as,

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

$A \rightarrow \alpha\beta_1 | \alpha\beta_2$

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 | \beta_2$

Eg:- $S \rightarrow iEts \; \varepsilon \; | \; iEtses \; | \; a$

$E \rightarrow b$

(α = $iEts$, β_1 = ε, α = $iEtses$, β_2 = a)

Rewritten:

$S \rightarrow iEtss'$

$S' \rightarrow es | \varepsilon$

The production after Left factoring:

$S \rightarrow iEts | iEtSes | a$

$E \rightarrow b$

3.17

Rewritten

$S \rightarrow iEtSS' | a$

$S' \rightarrow eS | \varepsilon$

$E \rightarrow b$

eg: Produce "ibta"

$S \rightarrow iEtSS'$

$S \rightarrow ibtSS'$

$S \rightarrow ibtaS'$ $\quad E \rightarrow b$

$S \rightarrow ibta\varepsilon$ $\quad S \rightarrow a$

$S \rightarrow ibta$ $\quad S' \rightarrow \varepsilon$

Parsing:

Parsing is the process of analysing a continuous stream of input in order to determine its grammatical structure with respect to given formal grammar.

Top-down Parsing:

A parser can start with the start symbol and try to transform to the input.

Eg: LL parser

Bottom up parser:

A parser starts with I/p and to rewrite it into start symbol is called as bottom up parser.

Eg: L-R parser

Parser: A parser for grammar G is a program that takes string 'w' as i/p and produces o/p either a parse tree for 'w', if w is a sentence of 'G' (or) an error message indicating that w is not a sentence of 'G'.

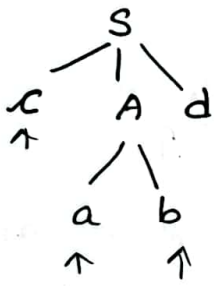
Top-Down Parsing:-

Eg: $S \rightarrow cAd$

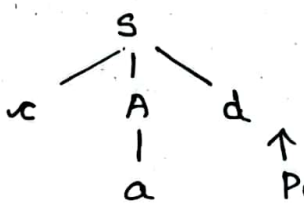
$A \rightarrow ab/a$

$w = cad$ ⇒ i/p

$S \rightarrow cAd$



* Here then the pointer is reset upto A.



Hence valid string.

I/P	Production	Left most derivation	Comment
cad ↑	$S \rightarrow cAd$	→ cAd ↑	i/p ptr matches with left most symbol of derivation and advance ptr
cad ↑	$A \rightarrow ab$	cabd ↑	i/p ptr matches with 2nd symbol of derivation and advance the ptr.
cad ↑	$A \rightarrow ab$	cabd ↑	i/p ptr does not matches the 3rd sym. So reset the ptr Hence discard the previous production.

3.19

cad	–	–	choose an alternate production
cad ↑	A→a	cad ↑	i/p ptr matches the 2nd symbol, advance ptr.
cad		cad	i/p ptr matches the 3rd symbol & string is derived.

Difficulties in top down parsing:

→ Left Recursion
→ Backtracking
→ Rejection of a valid string
→ Error reporting.

Left Recursion:

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

Procedure for E()

```
begin E()
   E()
   :
End
```

procedure T()

```
begin T()
   T()
   :
End
```

Procedure F()

```
begin F()
   F()
   :
end
```

Backtracking:-

→ If more alternatives are present, large amount of time takes place

→ More overhead takes place.

Rejection of a valid string

For eg: even though cabd is a valid string, we reject it to get the string cad.

Error Reporting:

At which point error occurs is not known (eg. by choosing (or) substituting the alternative)

less idea about errors

12/02/2016 Techniques of Top Down Parsing.

* Recursive Descend Parsing (With but Backtracking)

This is a top-down parsing methodology in which recursive procedures are used for parsing.

Let us consider the grammar 'G' for the language 'L'.

$E \rightarrow E+T / T$

$T \rightarrow T*F / F$

$F \rightarrow (E) / id$

$A \rightarrow A\alpha / \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \varepsilon$

After elimination of left recursion.

$E \rightarrow TE'$

$E' \rightarrow TE' / \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

$F \rightarrow (E) / id$

$E \rightarrow \underset{A}{E}\underset{\alpha}{+T} / \underset{\beta}{T}$

$E \rightarrow TE'$

$E' \rightarrow +TE' / \varepsilon$

$T \rightarrow \underset{A}{T}\underset{\alpha}{*F} / \underset{\beta}{F}$

$T \rightarrow FT'$

$A' \rightarrow \alpha A' / \varepsilon$

$T' \rightarrow *FT' / \varepsilon$

3.21

Procedure for each non-terminal:

① Procedure E()

```
begin
T()
EPRIME()
end;
```

② Procedure EPRIME()

```
begin
If i/p symbol = '+' then
begin
ADVANCE()
T()
EPRIME()
end
end
```

③ Procedure T()

```
begin
F()
TPRIME()
end;
```

④ Procedure TPRIME()

```
begin
If i/p symbol = '*' then
begin
ADVANCE()
F()
TPRIME()
end
end
```

⑤ Procedure F() ✓

```
begin
IF i/p symbol = 'id' then
 begin
 ADVANCE()
end
 else if i/p symbol = '(' then
 begin
 ADVANCE()
 E()
IF i/p symbol = ')' then
 ADVANCE()
 else
    ERROR() end;
else
    ERROR() end ;
```

Production	Input String	Comments.
1. E()	id + id * id ↑	Pointer pointing to id
2. T()	id + id * id ↑	Pointer pointing to id
3. F()	id + id * id ↑	Pointer pointing to id.
4. ADVANCE()	id + id * id ↑	Advance the ptr pointing to '+' return to T() and called TPRIME()
5. TPRIME()	id + id * id ↑	Pointer Pointing to '+'.
6. EPRIME()	id + id * id ↑	Pointer Pointing to '+'.
7. ADVANCE()	id + id * id ↑	Advance the ptr pointing to id.
8. T()	id + id * id ↑	Pointer pointing id

3:23

9. F()	id + id * id ↑	Pointer pointing id.
10. ADVANCE()	id + id * id ↑	Advance the pointer, and the pointer pointing the '*'
11. TPRIME()	id + id * id ↑	Pointer pointing to '*'
12. ADVANCE()	id + id * id ↑	Advance the ptr, and the ptr pointing to the id.
13. F()	id + id * id ↑	Pointer pointing to id
14. ADVANCE()	id + id * id ↑	Advance the pointer, and the ptr pointing to end of string.
15. TPRIME()	id + id * id ↑	Comments no change.
16. EPRIME()	id + id * id ↑	comments halt.

id * id

Production	Input String	Comments.
1. E()	id * id ↑	Pointer pointing to id
2. T()	id * id ↑	Pointer Pointing to id
3. F()	id * id ↑	Pointer Pointing to id
4. ADVANCE()	id * id ↑	Advance the pointer and pointer pointing to '*'
5. TPRIME()	id * id ↑	Pointer pointing to id
6. ADVANCE()	id * id ↑	Advance the pointer and Pointer Pointing to 'id'.
7. F()	id * id ↑	Pointer pointing id.
8. ADVANCE()	id * id ↑	Advance the pointer, and ptr points to end of the string

9. TPRIME()	id * id ↑	comments no change
10. EPRIME()	id * id ↑	comments halt.

Recursive descent:

```
void A()
{
choosen A Production, where A → x1, x2, ..... xk
for i=1 to k
{
if xi is a non-terminal
call procedure xi()
else
  if (xi == current i/p symbol 'a')
then
Advance the i/p to the next symbol.
else
  error
}
}
```

Elimination of Left Recursion Algorithm:

INPUT: Grammar G with no cycles or ε-production

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Arrange the non-terminals in some order $A_1, A_2, \ldots A_n$.

```
for (each i from 1 to n)
{
  for (each j from -1 to i-1)
{
  replace each production of the form Ai → Aj γ
by the productions Ai → s1γ | s2γ | ..... | skγ ; where
Aj → s1 | s2 | ..... | sk are all current Aj productions
```

3

eliminate the immediate left recursion among the Ai productions.

3

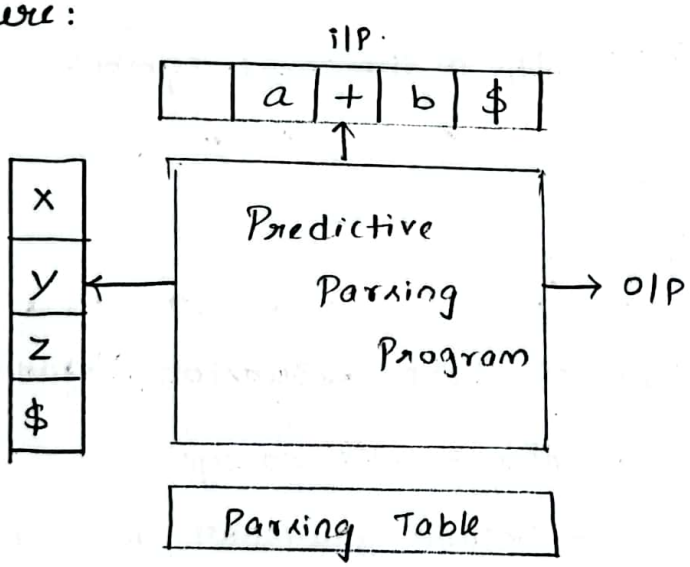
Predictive parser (or) Non Recursive predictive parser (or) Table driven predictive parser (or) Recursive descent parser with no backtracking (with no backtracking)

INPUT: Input always ends with '$' symbols ∴ (i.e) right end marker.

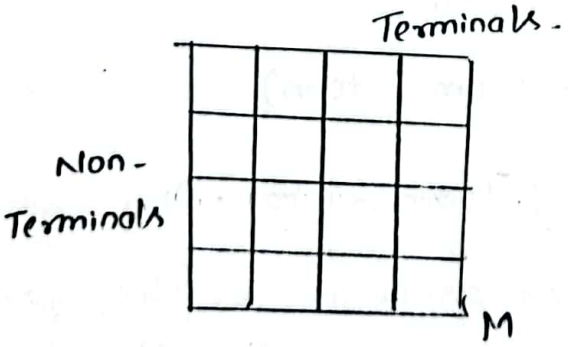
STACK: All Grammar symbols are present in the stack which is preceeded by $ symbol.

(i.e) all terminals and non-terminals.

Structure:



Parsing Table:



* The 2D array with Non-terminals at left and Terminals at top
* Named as 'M'.

HINT:

1. Consider a Grammar
2. If the grammar has left recursion, then eliminate
3. Find FIRST of all non-terminals
4. Find the follow of all non-terminals
5. Construct the parsing table 'M' using FIRST and FOLLOW
6. Parse the i/p string using predictive parsing algorithm.
7. Announce whether the given i/p is valid (or) not.

Example:

Consider the i/p string "id + id * id" and check whether it is valid (or) not using the grammar given.

1. Consider the grammar:

$E \rightarrow E+T / T$ ---- $A \rightarrow A\alpha / \beta$

$T \rightarrow T * F / F$ $\quad A \rightarrow \beta A'$

$F \rightarrow (E) / id$ $\quad A' \rightarrow \alpha A' / \varepsilon$

2. Elimination of Left Recursion

$E \rightarrow TE'$

$E' \rightarrow +TE' / \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' / \varepsilon$

3.27

$F \rightarrow (E) / id$

3. Find FIRST of all non-terminals.

FIRST (E) = { (, id }

FIRST (E') = {+, ε}

FIRST (T) = { (, id }

FIRST (T') = {*, ε}

FIRST (F) = { (, id }

4. Find FOLLOW

FOLLOW (E) = {), $ }

FOLLOW (E') = {), $ }

FOLLOW (T) = { +,), $ }

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

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

5. Construct the parsing table 'M' using FIRST & FOLLOW :

$E \rightarrow TE'$

FIRST [E] = { (, id }

M [E, (] $\Rightarrow E \rightarrow TE'$

M [E, id] $\Rightarrow E \rightarrow TE'$

$E' \rightarrow +TE'$

FIRST [E'] = {+}

M [E', +] $\Rightarrow E' \rightarrow +TE'$

$E' \rightarrow \varepsilon$ while considering and FOLLOW is taken for E'

FOLLOW [E'] = {), $ }

M [E',)] $\Rightarrow E' \rightarrow \varepsilon$

M [E', $] $\Rightarrow E' \rightarrow \varepsilon$

$T \rightarrow FT'$

FIRST [T] = {(, id}

M [T, (] ⇒ $T \rightarrow FT'$

M [T, id] ⇒ $T \rightarrow FT'$

$T' \rightarrow * FT'$

FIRST [T'] = {*}

M [T', *] ⇒ $T' \rightarrow * FT'$

$T' \rightarrow \xi$

FOLLOW [T'] = {+,), $}

M [T', +] ⇒ $T' \rightarrow \xi$

M [T',)] ⇒ $T' \rightarrow \xi$

M [T', $] ⇒ $T' \rightarrow \xi$

$F \rightarrow (E)$

FIRST [F] = {(}

M [F, (] ⇒ $F \rightarrow (E)$

$F \rightarrow id$

FIRST [F] = {id}

M [F, id] ⇒ $F \rightarrow id$

Parsing Table:

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

6. Parse the i/p string using predictive parsing algorithm:

Stack	i/p	Comment
$E	id +id * id $	Initial state
$E'T	id +id * id $	M[E, id] = E → TE'
$E'T'F	id +id * id $	M[T, id] = T → FT'
$E'T'id	id +id * id $	M [F, id] = F → id
$E'	+ id * id $	M [T', +] = T' → ε
$E'T+	+ id * id $	M [E', +] = E' → +TE'
$E'F	id * id $	M [T, id] = T → FT'
$E'T'id	id * id $	M [F, id] = F → id
$E'T'F*	* id $	M [T', *] = T' → *FT'
$E'T'id	id $	M [F, id] = F → id
$E'	$	M [T', $] = T' → ε
$	$	M[E', $] = E' → ε
$	$	HALT

Example 2:

Consider the string "id +id /id +(id + id)" and check whether it is valid (or) not using the grammar of Eg 1.

Stack	i/p	Comment.
$E	id + id /id + (id + id)	Initial state
$E'T	id + id /id + (id + id)	M [E, id] = E → TE'
$ E'T' F	id + id /id + (id + id)	M [T, id] = T → FT'
$ E'T' id	id + id / id + (id + id)	M [F, id] = F → id
$E'	+ id /id + (id + id)	M [T', +] = T' → ε

$ E'T+	+id/id+(id+id)	M[E',+] = E' → +TE'
$ T'F	id/id+(id+id)	M[T,i] = T → FT'
$ T'id	id/id+(id+id)	M[F,id] = F → id
		M[T',/] = INVALID
	INVALID.	

Example 3:

Consider the grammar

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

$T \rightarrow T, S \mid S$

and check whether the given i/p are valid (or) not

(i) (a,(a,a))

(ii) (((a,a),↑,(a),a)

1. Consider the grammar

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

$T \rightarrow T, S \mid S$

2. Elimination of Left Recursion:

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

$\frac{T}{A} \rightarrow \frac{T}{A}, \frac{S}{\alpha} \mid \frac{S}{\beta}$

$A \rightarrow A\alpha \mid \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \varepsilon$

$T \rightarrow ST'$

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

After Left Recursion

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

$T \rightarrow ST'$

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

3.31

3. Find FIRST of all non-terminals

FIRST(S) = {a, ↑, (}

FIRST(T) = {a, ↑, (}

FIRST(T') = {, , ε}

4. Find FOLLOW of all non-terminals

FOLLOW(S) = {, , follow of T} ⇒ {, ,), $}.

FOLLOW(T) = {)}

FOLLOW(T') = {)}

5. Construct the parsing table 'M' using FIRST and FOLLOW

S → a

FIRST [S] = {a}

M[S, a] ⇒ S → a

S → ↑

FIRST [S] = {↑}

M[S, ↑] ⇒ S → ↑

S → (T)

FIRST [S] = {(}

M[S, (] ⇒ S → (

T → ST'

FIRST [T] = {a, ↑, (}

M[T, a] ⇒ T → ST'

M[T, ↑] ⇒ T → ST'

M[T, (] ⇒ T → ST'

T' → , ST'

FIRST [T'] = {, }

M[T', ,] ⇒ T' → , ST'

$T' \rightarrow \varepsilon$

FOLLOW [T'] = {)}

M [T',)] $\Rightarrow T' \rightarrow \varepsilon$

Parsing Table:

	a	↑	(	)	,	$
S	$S \rightarrow a$	$S \rightarrow \uparrow$	$S \rightarrow (T)$			
T	$T \rightarrow ST'$	$T \rightarrow ST'$	$T \rightarrow ST'$			
T'				$T' \rightarrow \varepsilon$	$T' \rightarrow ,ST'$	

6. Parsing the i/p string using predictive parsing algorithm

(i) (a,(a,a))

Stack	input	Comments.
$S	(a,(a,a))$	Initial state
$)T(	(a,(a,a))$	M[S,(] = S → (T)
$)T'S	a,(a,a))$	M[T,a] = T → ST'
$)T'a	a,(a,a))$	M[S,a] = S → a
$)T'S,	,(a,a))$	M[T',;] = T' → ,ST'
$)T')T(	(a,a))$	M[S,(] = S → (T)
$)T')T'S	a,a))$	M[T,a] = T → ST'
$)T')T'a	a,a))$	M[S,a] = S → a
$)T')T'S,	,a))$	M[T',,] = T' → ,ST'
$)T')T'a	a))$	M[S,a] = S → a
$)T'))	))$	M[T',)] = T' → ε
$)	)$	M[T',)] = T' → ε
$	$	HALT

3.33

(ii) (((a,a),↑,(a),a)

Stack	Input	comment
$S	(((a,a),↑,(a),a)$	Initial state
$)T(	(((a,a),↑,(a),a)$	M[S,(] = S→(T)
$)T'S	((a,a),↑,(a),a)$	M[T,(] = T→ST'
$)T')T(	((a,a),↑,(a),a)$	M[S,(] = S→(T)
$)T')T'S	(a,a),↑,(a),a)$	M[T,(] = T→ST'
$)T')T')T(	a,a),↑,(a),a)$	M[S,(] = S→(T)
$)T')T')T'a	a,a),↑,(a),a)$	M[T,a] = T→ST'
$)T')T')T'	,a),↑,(a),a)$	M[S,a] = S→a
$)T')T')T'S,	,a),↑,(a),a)$	M[T',,] = T'→,ST'
$)T',T')T'a	a),↑,(a,a)$	M[S,a] = S→a
$)T')T')T'	),↑,(a),a)$	M[T',)] = T'→ε
$)T')T'S,	,↑,(a),a)$	M[T',,] = T'→,ST'
$)T')T'↑	↑,(a),a)$	M[S,↑] = S→↑
$)T')TS,	,(a),a)$	M[T',,] = T'→,ST
$)T')T)T(	(a),a)$	M[S,(] = S→(T)
$)T')T)T'S	a),a)$	M[T,a] = T→ST'
$)T')T)T'a	a),a)$	M[S,a] = S→a
$)T')T)	),a)$	M[T',)] = T'→ε
$)T')T'T'S,	,a)$	M[T,,] = ST'
$)T')T'T'a	a)$	M[S,a] ⇒ S→a
$)T')T'	)$	M[T',)] ⇒ T'→ε
$)T')	)$	M[T',)] ⇒ T'→ε
		Invalid

K.KARTHICKA
312013104044

1) $S \to (L) \mid a$

$L \to L, S \mid S$

* consider the Grammar

$S \to (L) \mid a$

$L \to L, S \mid S$

* Remove left Recursion

$\underset{A}{L} \to \underset{A}{L} \underset{\alpha}{, S} \mid \underset{\beta}{S}$

$A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$

$L \to SL'$, $L' \to , SL' \mid \varepsilon$.

$S \to (L) \mid a$

$L \to SL'$

$L' \to , SL' \mid \varepsilon$

* FIRST POS :-

FIRST POS (S) = {(, a}

FIRST POS (L) = {(, a}

FIRST POS (L') = {, , ε}

* FOLLOW POS (S) = {$, follow of L; first of L'} = {$,), ,}

FOLLOW POS (L) = {)}

FOLLOW POS (L') = {follow of L} = {)}

* constructing parse table.

consider $S \to (L)$

FIRST (S) = {(}

$M[S, (] = S \to (L)$

consider $S \to a$

$FIRST(S) = \{a\} \Rightarrow M[S, a] \Rightarrow S \to a$

consider $L \to SL'$

$FIRST(L) =$ FIRST of S

$= \{(, a\}$

$M[L, (] = L \to SL'$

$M[L, a] = L \to SL'$

consider $L' \to , SL'$

$FIRST(L') = FIRST(, SL') = \{,\}$

$M[L', ,] = L' \to , SL'$

consider $L' \to \varepsilon$

$FIRST(L') = \varepsilon$

$FOLLOW(L') = \{)\}$

$M[L',)] = L' \to \varepsilon$

Parsing table:

	(	)	a	,	$
S	$S \to (L)$		$S \to a$		
L	$L \to SL'$		$L \to SL'$		
L'		$L' \to \varepsilon$		$L' \to , SL'$	

9) (a,a)

Stack	input	comment
$S	(a,a)$	initial state
$)L(	(a,a)$	M[S,(] ⇒ S→(L)
$)L	a,a)$	M[L,a] = S→SL'
$)L'S	a,a)$	M[S,a] = S→a
$)L'a	a,a)$	M[L',,] = L'→,SL'
$)L'S,	,a)$	M[S,a] = S→a
$)L'a	a)$	M[L',)] = L'→ε
$)	)$	
$	$	HALT.

The given input (a,a) is accepted if it is valid input.

ii) ((a,a)

Stack	Input	comment
$S	((a,a)$	M[S,(] = S→(L)
$)L(	(a,a)$	M[L,(] = L→SL'
$)L'S	a,a)$	M[S,(] = S→(L)
$)L')L(	a,a)$	M[L,a] = L→SL'
$)L')L'S	a,a)$	M[S,a] = S→a
$)L')L'a	,a)$	M[L',,] ⇒ L'→,SL'
$)L')L'S,	a)$	M[S,a] = S→a
$)L')L'a	a,$	M[L',)] = L'→ ε.
$)L')	,$	Invalid
$)L'	$	

The given i/p is invalid

To compute FIRST()

The FIRST function determines the beginning terminals which are derived from a particular grammar symbol.

Steps:

1. If x is a terminal then FIRST(x) is x itself

   eg: FIRST(id) = id.

2. If x is a non-terminal and $x \to y_1, y_2, y_3, \ldots y_k$ is a production for some $k \geq 1$ then place 'a' in FIRST(x). If for some i where i, a is in FIRST(y_i) and ε is in all of FIRST(y_i) so on ..... upto FIRST(y_{i-1}).

   (i.e) $y_1, y_2, \ldots y_{i-1} \overset{*}{\Rightarrow} \varepsilon$

   If ε is in FIRST(y_i) for all $j = 1, 2, \ldots k$ then add ε to FIRST(x).

3. If $x \to \varepsilon$ is a production, then add ε to FIRST(x)

To compute FOLLOW(A).

1. Place $ symbol in FOLLOW(S), where S is the start symbol and $ is the input.

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

3. If $A \to \alpha B$ is a production (or) if $A \to \alpha B \beta$ where FIRST(β) contains ε, then everything in FOLLOW(A) is in FOLLOW(B).

3:35

LL(1) Grammar :

Predictive parser that is Recursive descend parser needing no backtracking can be constructed for a class of grammar called LL(1) grammars.

LL(1) grammar :

1st L – scaning of input from left to right.

2nd L – productions in leftmost derivation.

(1) – one input symbol of lookahead at each step to make parsing action decision.

A Grammar G is LL(1) if and only if $A \to \alpha | \beta$ are two distincts productions of G when the following conditions holds:

1. For no terminal a, both α and β derive strings beginning with a.
2. Almost one of α (or) β can derive an empty string
3. If β derives ε with Kleen's closure $\beta \overset{*}{\Rightarrow} \varepsilon$, then α do not derive any string beginning with any terminal in FOLLOW(A).
4. If $\alpha \overset{*}{\Rightarrow} \varepsilon$ then β do not derive any string beginning with a in FOLLOW(A).

Date: 15/02/16

- Panic mode error recovery:- [continuation]

) id * + id $

Stack	Input	Comments.
$E	id * + id $	M[E,)] = SYNCH
$ E'T	id * + id $	M[E,id] = TE'
$ E'T'F	id * + id $	M[T,id] = FT'
$ E'T'id	id * + id $	M[F,id] = id
$ E'T'F *	* + id $	M[T',*] = *FT'
$ E'T'	+ id $	M[F,+] = SYNCH
$ E'	+ id $	M[T',+] = T' → ε
$ E'T+	+ id $	M[E',+] = E' → +TE'
$ E'T'F	id $	M[T,id] = T → FT'
$ E'T'id	id $	M[F,id] = F → id
$ E'	$	M[T',$] = T' → ε
$	$	M[E',$] = E' → ε
		HALT

Construction of Predictive Parsing Table:

Algorithm:

INPUT: Grammar G

OUTPUT: Parsing Table M

METHOD: For each production $A \to \alpha$ of the grammar, do the following.

1. For each terminal a in FIRST(α), add $A \to \alpha$ to M[A,a]
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to M[A,b]. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to M[A,\$] as well.
3. If after performing the above, there is no production at all in M[A,a], the set M[A,a] to error (which we normally represent by an empty entry in the table).

Table-driven Predictive Parsing:

Algorithm:

INPUT: A string w and a parsing table M for grammar G.

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

METHOD:

let a be the first symbol of w;

let x be the top stack symbol;

```
while (x = $) {/* stack is not empty */}
else if (x is a terminal) error();
else if (M[x,a] is an error entry) error();
else if (M[x,a] = x → y1 y2 .... yk) {
output the production x → y1 y2 ... yk;
pop the stack;
push yk, yk-1, .... y1 onto the stack, with y1 on
                                top of stack;
} Let x be the stack top symbol;
}
```

Consider the grammar

S → iEts | iEtses | a

E → b

Construct the parsing table to check for ambiguity.

1. Consider the grammar.

   S → iEts | iEtses | a

   E → b

2. Elimination of left recursion and left factorising (if needed)

   S → iEts | iEtses | a

   E → b

The above grammar has no left recursion. Left factoring is performed here, since the grammar has left factoring.

S → iEts | iEtses | a

A → αβ1 | αβ2

A → αA'

A' → β1 | β2

Thus the grammar would be,

S → iEtsS'

S' → ε | es

E → b

S → a

3. Find FIRST of all non-terminals.

FIRST [S] = { a, ϕ }

FIRST [S'] = {b}

FIRST [E] = {b}

4. Find FOLLOW of all non-terminals.

FOLLOW [S] = { e, FOLLOW(S), $ }

FOLLOW [S'] = { e, $, FOLLOW(S) }

FOLLOW [E] = {t}

⇒ FOLLOW [S] = {e, $}

FOLLOW [S'] = {e, $}

FOLLOW [E] = {t}

5. Construct parsing table 'M' using FIRST and FOLLOW

$S \rightarrow iEtSS'$

FIRST [S] = {i}

M [S, i] = $S \rightarrow iEtSS'$

$S' \rightarrow eS$

FIRST [S'] = {e}

M [S', e] = $S' \rightarrow eS$

$E \rightarrow b$

FIRST [E] = {b}

M [E, b] = $E \rightarrow b$

$S \rightarrow a$

FIRST [S] = {a}

M [S, a] = $S \rightarrow a$

$S' \rightarrow \varepsilon$

FOLLOW [S'] = {e, $}

M [S', e] = $S' \rightarrow \varepsilon$

M [S', $] = $S' \rightarrow \varepsilon$

Parsing Table

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

Multiple entities are there between S' and e.

This type of grammar is known as NO LL(1) grammar.

Error Recovery in Predictive parser:

Error recovery refers to the stack of the table driven predictive parser, since it makes the terminals and non-terminals that the parser hopes to match with the remainder of the input. This technique can also be used in Recursive descent parsing.

An error is detected ~~using~~ during the predictive parsing when the terminal on the top of the stack does not match the next i/p symbol or when non-terminal A is on the top of the stack, 'a' is the next i/p symbol and M [A,a] is error (i.e) the parsing table entry is empty.

Panic mode error recovery:

It is based on the idea of skipping one symbol on i/p until a token in a selected set of synchronising tokens appears. Its effectiveness depends on the choice of synchronising set.

Non Termi-nal	Input Symbol.					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	SYNCH	SYNCH
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$	SYNCH	SYNCH
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$	SYNCH	SYNCH	$F \to (E)$	SYNCH	SYNCH

3.41

FOLLOW(E) = FOLLOW(E') = {), $}

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

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

18/02/2016. Refer 3.36 ⟸ [The problem continuation is there is written before 3 pages].

Heuristics in panic mode error recovery of Predictive parser:

1. Place all symbols in FOLLOW(A) into the synchronising set of Non-terminal A. If we skip the tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.

2. It is not enough to use FOLLOW(A) as the synchronising set of A.

3. If we add symbols in FIRST(A) to the synchronising set for Non terminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

4. If a non-terminal can generate the empty string, then the production deriving ϵ can be used as default. This reduces the no of non-terminals considered using error recovery.

5. If

Phrase level Recovery:

It is implemented by filling in the blanks entries in the predictive parsing table with pointers and error routines.

These routines may change, input or delete symbols on the i/p and issue appropriate error messages.

They may also pop from the stack.

(i) The steps carried out by the parser might then not correspond to the derivation of any word in the language at all.

(ii) Ensure that there is no possibility of infinite loop.

assignment problem:-
Show that the Grammar is in LL(1) / not LL(1).

$S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

Bottom-up Parsing:-

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

A bottom up parse for id * id

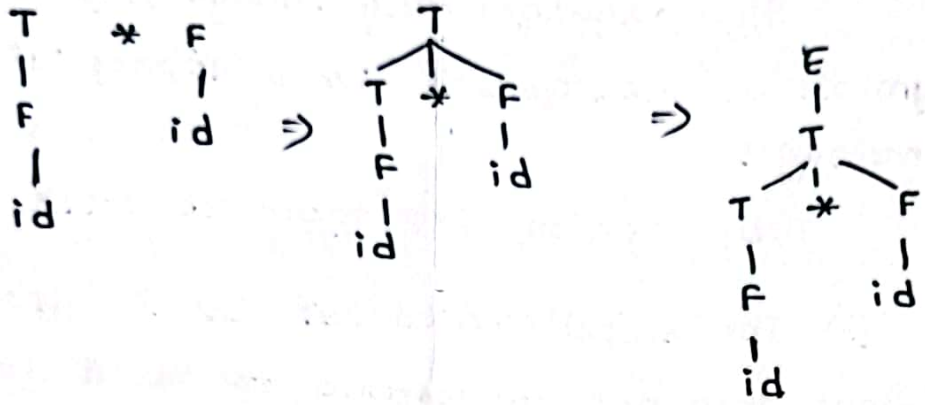
$E \to E+T / E*T / (E) / id$

$id * id \Rightarrow \begin{cases} F * id \\ | \\ id \end{cases} \Rightarrow \begin{matrix} T * id \\ | \\ F \\ | \\ id \end{matrix}$

$E \to E+T / \underline{T}$

$\underline{T} \to T*F / \underline{F}$

$\underline{F} \to (E) / \underline{id}$

$\Rightarrow$ 

$\Rightarrow$ shift Reduce parsing

$\Rightarrow$ Handle

$\Rightarrow$ Handle pruning

$\Rightarrow$ Steps to construct derivations in Reverse order [Reduction]

$\Rightarrow$ Actions [stack implementation of shift reduce parsing]

$\Rightarrow$ Actions of shift reduce parsing

$\Rightarrow$ Importance of handle pruning in shift reduce parsing

$\Rightarrow$ conflicts during shift reduce parsing

$\Rightarrow$ Viable prefixes.

⇒ Shift Reduce Parsing: [Reduction]

⇒ perform Reduction (or) right most derivation in reverse.

consider the grammar,

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

consider the i/p string "abbcde" and perform parsing

Reduction

a b b c d e

a A b c d e (A → b)

a A d e (A → Abc)

a A B e (B → d)

S (S → aABe)

[RMD]

Right Most Derivation in Reverse:

$S \rightarrow aABe$

$\rightarrow aAde$

$\rightarrow aAbcde$

$\rightarrow abbcde$

⇒ HANDLE

A handle of a string is a substring that matches the right side of a production & whose reduction to a Non terminal on the left side of the production represents one step reverse on the Right most derivation.

3.45

→ A handle of a Right sentencial form γ is a production $A \to \beta$ is a position of γ where the string β be found & replaced by A in the previous right sentencial form.

$$A \to \beta$$

$$S \xrightarrow[rm]{*} \alpha A \omega$$

$$S \xrightarrow[rm]{*} \alpha \beta \omega$$

Here $A \to \beta$ is a handle.)

Consider the grammar:

$$E \to E+E \mid E*E \mid (E) \mid id \mid \epsilon$$

derive $id_1 + id_2 * id_3$

Right most derivation

$$E \Rightarrow E * E$$

* has higher precedence than +

$E \Rightarrow E * id_3$ $(E \to id)$

$\Rightarrow E + E * id_3$ $(E \to id)$

$\Rightarrow E + id_2 * id_3$ $(E \to E+E)$

$E \Rightarrow id_1 + id_2 * id_3$ $(E \to id)$ $(E \to E*E)$

when + has higher precedance than *

$E \Rightarrow E + E$ $(E \to id)$

$\Rightarrow E + E * E$ $(E \to id)$

$\Rightarrow E + E * id_3$ $(E \to id)$

$\Rightarrow E + id_2 * id_3$ $(E \to E*E)$

$\Rightarrow id_1 + id_2 * id_3$ $(E \to E+E)$

Here the grammar is ambiguous

Handle pruning

A rightmost derivation in reverse can be obtained by handle pruning. ie start with a string of terminals w that is to parse

If 'w' is a sentence or string of the grammar to be parsed.

$$w = \gamma_n$$

where γ_n is the right sentential form of some yet unknown rm derivation

γ_m be

$$S \underset{rm}{\Rightarrow} \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \ldots \gamma_n = w$$

example:

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

Trace id + id * id

$E \Rightarrow E+E$

$\Rightarrow E+E*E$

$\Rightarrow E+E*id$

$\Rightarrow E+id*id$

$\Rightarrow id+id*id$

Steps to Construct Derivation In Reverse Order:

→ Locate handle 'β_n' in 'γ_n'

→ Replace 'β_n' by 'A_n' using the production $A_n \rightarrow \beta_n$.

→ Locate 'β_{n-1}' in 'γ_{n-1}' & replace it by 'A_{n-1}'.

→ Continue the above process until we obtain a right sentinal form consisting only start symbol

→ Halt and announce success

3.47

→ Reverse of this production used in reduction is the right most derivation for the string.

Example:-

Input String	Handle	Reducing Production.
$id_1 + id_2 * id_3$	id_1	$E \to id$
$E + id_2 * id_3$	id_2	$E \to id$
$E + E * id_3$	id_3	$E \to id$
$E + E * E$	$E + E$	$E \to E + E$
$E * E$	$E * E$	$E \to E * E$
E	–	–

22/02/16

Stack Implementation for shift reduce parsing:

Action:

1. shift
2. Reduce
3. Reject (or) error
4. Accept.

Eg: Consider the grammar

$E \to E+E \mid E*E \mid (E) \mid id$

Stack	Ip	Comment:
$	id + id * id $	shift (id)
$id	+ id * id $	Reduce E→id
$E	+ id * id $	shift (+)
$E +	id * id $	shift (id)
$E + id	* id $	Reduce E→id

$ E+E	*id $	shift (*)
$ E+E*	id $	shift (id)
$ E+E*id	$	Reduce E → id
$ E+E*E	$	Reduce E → E*E
$ E+E	$	Reduce E → E+E
$E	$	Accept

ii) id + (id *

Stack	I/P	Comment
$	id + (id * $)	shift id
$id	+ (id * $	Reduce E → id
$E	+ (id * $	shift '+'
$E+	(id * $	shift '('
$E+(	id * $	shift 'id'
$E+(id	* $	Reduce E → id
$E+(E	* $	shift '*'
$E+(E *	$	Reject (or) error

Importance of handle pruning in shift reduce parsing:-

→ The handle always appear in the top of stack.
→ This makes stack as a suitable datastructure
→ Botton-up parsing is based on recognising handles.
→ shift reduce parser performs reduction if it finds handle on the top of stack (or) it shift zero

3.49

on more symbols on stack until a handle found in the top of the stack.

Case i)

$S \rightarrow \alpha A z$

$A \rightarrow \beta B y$

$B \rightarrow \gamma$

Rightmost Derivation:

$S \rightarrow \alpha A z$

$\rightarrow \alpha \underline{\beta B y z}$

↳ handle

Implement using stack and procedure "$\alpha\beta\gamma yz$" using stack (Shift Reduce parsing method)

Stack	i/p	Comment.
$	$\alpha\beta\gamma yz\$$	shift 'α'
$\alpha	$\beta\gamma yz\$$	shift 'β'
$\alpha\beta	$\gamma yz\$$	shift 'γ'
$\alpha\beta\gamma	$yz\$$	Reduce γ, B→γ
$\alpha\beta B	$yz\$$	shif y.
$\alpha\beta By	$z\$$	Reduce A→βBy
$\alpha A	$z\$$	shift z
$\alpha Az	$\$$	Reduce αAz, S→αAz
$S	$\$$	Accept.

(case ii) $S \to \alpha BxAz$

$A \to y$

$B \to \gamma$

Stack	I/p	Comment
$	$\alpha\gamma xyz\$$	shift 'α'
$ α	$\gamma xyz\$$	shift 'γ'
$ αγ	$xyz\$$	reduce $B \to \gamma$
$ αB	$xyz\$$	shift 'x'
$ αBx	$yz\$$	shift 'y'
$ αBxy	$z\$$	reduce $A \to y$
$ αBxA	$z\$$	shift z
$ αBxAz	$\$$	reduce $S \to \alpha BxAz$
$S	$\$$	Accept.

⇒ In both the cases, the parser had to shift 0 or more symbols to get next handle onto the stack

⇒ It never had to go into the stack to find the handle.

⇒ Hence handle pruning makes the stack, a convinent data structure for shift reduce parsing.

Conflicts during shift reduce parsing:

⇒ The general shift reducing techniques are

* perform shift action when no handle at stack top

* perform reduce action when there is handle at stack top

Conflict
- Shift reduce conflict
- Reduce reduce conflict

Shift Reduce conflict:

$E \rightarrow E+E \mid E*E \mid E/E \mid (E) \mid id$.

i/p :- id + id * id .

Stack	i/p	comment
$	id+id * id $	shift (id)
$ id	+id * id $	Reduce E→ id
$ E	+ id * id $	shift '+'.
$ E+	id * id $	shift 'id'.
$ E+id	* id $	Reduce E→ id
$ E+E.	* id $	shift '*'.
$ E+E*	id $	shift 'id'
$ E + E* id	$	Reduce E→id
$ E+E * E	$	Reduce E→ E*E
$ E+E	$	Reduce E→ E+E
$ E	$	Accepted state

Stack	I/P	Comment
$	id + id * id $	shift 'id'
$id	+ id * id $	reduce E → id
$E	+ id * id $	shift '+'
$E+	id * id $	shift 'id'
$E+id	* id $	Reduce E → id
$E+E	* id $	Reduce E → E+E
$E	* id $	shift '*'
$E*	id $	shift 'id'.
$E*id	$	Reduce E → id.
$E*E	$	Reduce E → E*E
$E	$	Accepted state.

Reduce - Reduce conflict

M → R+R | R+c | R

R → c

Stack	I/P	Comment.
$	c + c $	shift 'c'
$c	+ c $	Reduce R → c
$R	+ c $	shift '+'
$R+	c $	shift 'c'
$R+c	$	Reduce R → c
$R+R	$	Reduce M → R+R
$M	$	Accept state

3.53

Stack	i/p	Comment
$	c+c$	shift c
$c	+c$	Reduce R→c
$R	+c$	shift +
$R+	c$	shift c
$R+c	$	Reduce M→R+c
$M	$	Accept state

Reduce - reduce conflict.

M → R+R | R+c | R

R → c

i/p : c+c

Viable Prefixes:

→ If 'α' is the viable prefix of the grammar if there is a 'w' such that 'αw' is the right sential sentential form.

→ A set of prefix of right sentential form that appears on the stack of a shift reduce parser is called as Viable prefixes.

→ As long as viable prefixes on stack no parser error has been seen.

→ A set of Viable prefixes is a regular language.

LR Parser:

→ used for large set of context free grammar (CFG)

→ Efficient bottom up syntatic analytic technique that can be used to parse a large class of CFG is called as LR(K) parser.

LL(1)
- L → one i/p symbol at a time
- L → Left most derivation
- L → Left to right scan */* Top down parsing */*

LR(K)
- L → 'K' i/p symbol at a time
- R → Right most Derivation in Reverse (i.e) Reduction
- L → Left to right scan */* Bottom up parsing */*

Advantages of LR Parsing:

→ It recognizes virtually all programming language constructs for the grammar

→ It is efficient non-Backtracking shift reduce parsing

→ It is used to construct large set of CFG

→ It is easy to identify the syntatic errors as easy as possible.

Disadvantages:

→ To work in hand, it is a tedious job.

→ It is very hard to (LR parser tool) construct LR parser for all set of programming languages

∴ We need a tool for LR parsing generator

Eg: YACC ⇒ yet another compiler compiler

⇓

LR parser generator

Types of LR parser

⇒ 3 types of LR parser

SLR [SIMPLE LR]

CLR [CANONICAL LR]

LALR [LOOK AHEAD LR]

SLR:

Easy to implement, least powerful, simple low cost

CLR:

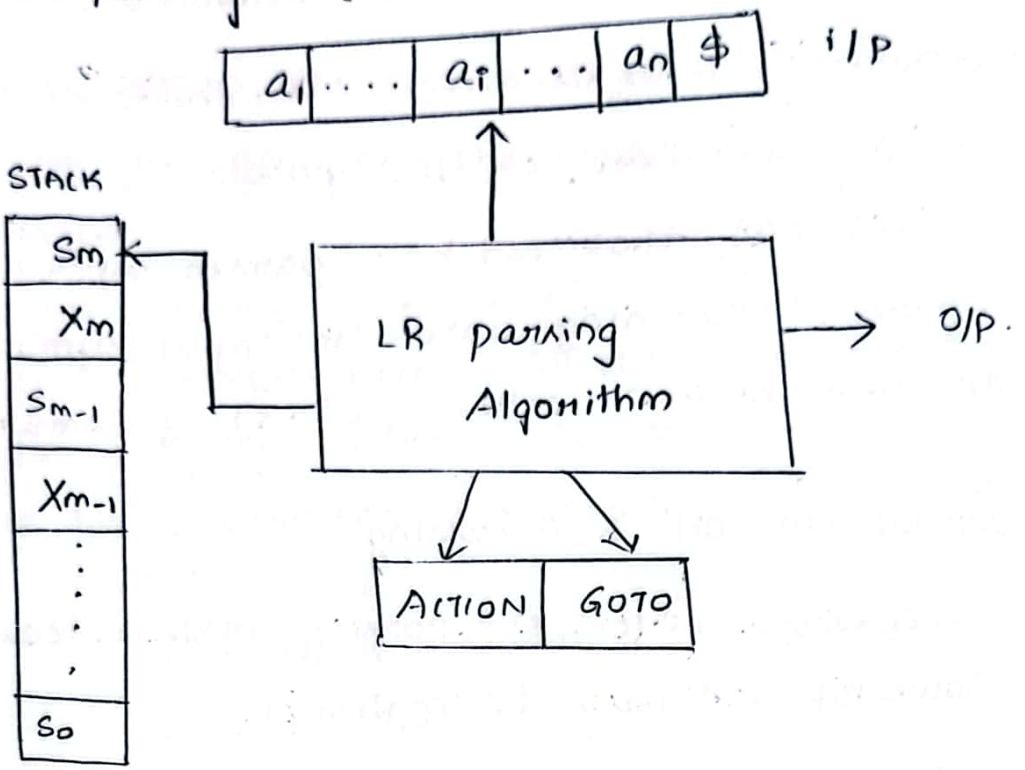
Most expensive, more powerful, tedious to implement

LALR:

Intermediate b|w SLR and CLR

Easy to implement, moderate cost and modirate powerful.

LR parsing algorithm:



* Driver program
* Parsing table
* Xφ ⇒ grammar symbol
  Sφ ⇒ state symbol

Action ⇒ terminal
goto ⇒ non terminal

LR Grammar:

→ A grammar for which we can construct a LR parsing table with action function for all the terminal and goto function for all non-terminals is said to be LR-grammar.

→ An LR parser does not have to scan the entire stack when the handle appears on the top

→ The state symbol on top of stack contains all the information it needs

3:57

→ The goto function is the finite automata

→ LR parser can be used to make shift-reduce decisions for next 'k' i/p symbols.

→ A grammar that can be parser by an LR-parser examining upto 'k' input symbols an each move is called LR(k) grammar.

Construction of SLR Parsing.

⇒ A simple LR(or) SLR parsing table is least powerful and easy to implement

⇒ A grammar for which SLR parser be constructed is said to be SLR grammar

LR(0) ITEMS (OR) ITEMS :-

An Item of a grammar is a production of G with a dot at same position of the right side.

Eg: $A \rightarrow xyz$.

The production above is replaced as .

$A \rightarrow .xyz$

$A \rightarrow x.yz$

$A \rightarrow xy.z$

$A \rightarrow xyz.$

* $A \rightarrow \epsilon$, the Item could be

$A \rightarrow .$

SET OF ITEMS.

→ A group the items together into sets, which give to rise the states of the SLR parser.

→ A collection of set of LR(0) items, we call it as canonical LR(0). collection provides the basis for the construction of SLR parser.

→ For construction SLR parsing table, 3 steps followed:

* Augumented grammar
* closure operation
* Goto operation.

Augumented Grammar:

If G ⇒ grammar, S → start symbol, then G' → augumented grammar and S' → start symbol of that augumented grammar and here the production S'→S is added with the given production.

Augumented grammar:

E→ E+T, E→T, T→T*F, T→F, F→(E), F→id.

E'→E
E→E+T
E→T
T→T*F
T→F
F→(E)
F→id

start symbol E, then augumented grammar G' whose start symbol is E' along with the production.

3.59

eg:-

Closure operation:

$E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

eg:- GOTO operation.

consider:-

$I : E' \rightarrow E$

$E : E+T$

GOTO (I, +)

$E \rightarrow E+.T$ advancing dot

$T \rightarrow .T*F$ } closure

$T \rightarrow .F$ }

$F \rightarrow .(E)$ }

$F \rightarrow .id$ }

24/02/16

Closure operation:

If I is the set of items for a grammar G, then closure (I) is constructed by two rules.

RULE 1:-

Every item in I is added to closure (I).

RULE 2:-

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

→ We apply the rule until no more new items can be added to closure (I).

Example:-

If $I = \{E' \rightarrow E\}$, then closure (I) = refer the eg.

GOTO operation:-

→ The function goto (I, x) where I is the set of items and x is a grammar symbol is defined to be the closure of the set of all items $[A \rightarrow \alpha X.B]$ such that $[A \rightarrow \alpha.XB]$ is in I.

if we have $A \to \alpha . X \beta$ then goto function gives closure

$[A \to \alpha X . \beta]$

refer the example

SLR(1)

⇒ Problem:-

Consider the production:

$E \to E+T / T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$.

step 1: construct the augmented grammar

$E' \to E$

$E \to E+T$

$E \to T$

$T \to T*F$

$T \to F$

$F \to (E)$

$F \to id$.

step 2: find the set of items based on closure operation and goto operation.

I0:

$E' \to .E$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

3.61

Goto (0, E)

I_1 :

$E' \to E.$

$E \to E. + T$

I2: Goto (0, T)

$E \to T.$

$T \to T. * F$

I3: Goto (0, F)

$T \to F.$

I4: Goto (0, ()

$F \to (.E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I5: Goto (0, id)

$F \to id.$

I6: Goto (1, +)

$E \to E+.T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I7: Goto (2, *)

$T \to T*.F$

$F \to .(E)$

$F \to .id$

I8: Goto (4, E)

$F \to (E.)$

$E \to E.+T$

I2: goto (4, T)

$E \to T.$

$T \to T.*F$

I3: goto (4, F)

$T \to F.$

I4: goto (4, ()

$F \to (.E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I5: Goto (4, id)

$F \to id.$

I9: Goto (6, T)

$E \to E+T.$

$T \to T.*F$

I3: Goto (6, F)

$T \to F.$

I4: Goto (6, ()

$F \to (.E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

I5: Goto (6, id)

$F \to id.$

I10: Goto (7, F)

$T \to T*F.$

I4: Goto (7, ()

$F \to .(E)$

$E \to .E+T$

$E \to .T$

$T \to .T*F$

$T \to .F$

$F \to .(E)$

$F \to .id$

Goto (7, id) : I5

$F \to id.$

I11: Goto (8,))

$F \to (E).$

Goto (8, +) : I6

$E \to E+.T$

$T \to .T*F$

$F \to .(E)$

$F \to .id$

I7: Goto (9, *)

$T \to T*.F$

$F \to .(E)$

$F \to .id$

0 represents I_0

↓

Initial state

Goto (0,E) = I_1

Goto (0,T) = I_2

Goto (0,F) = I_3

Goto (0,() = I_4

Goto (0,id) = I_5

Goto (1,+) = I_6

Goto (2,*) = I_7

Goto (4,E) = I_8

Goto (4,T) = I_2

Goto (4,F) = I_3

Goto (4,() = I_4

Goto (4,id) = I_5

Goto (6,T) = I_9

Goto (6,F) = I_3

Goto (6,() = I_4

Goto (6,id) = I_5

Goto (7,F) = I_{10}

Goto (7,() = I_4

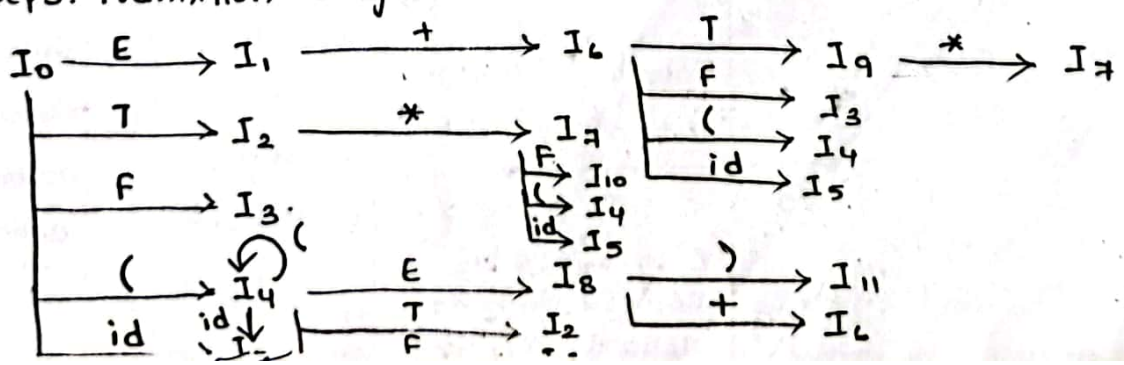
Goto (7,id) = I_5

Goto (8,)) = I_{11}

Goto (8,+) = I_6

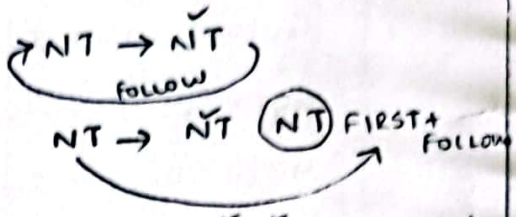
Goto (9,*) = I_7

Step3: Transition diagram.



3.63

step4:- Find the FOLLOW of non-terminal from the given grammar



⇒ Since left recursion is not eliminated in this grammar all the occurances on the right side of the production should be considered while finding FOLLOW.

① E → E+T
② E → T
③ T → T*F
④ T → F
⑤ F → (E)
⑥ F → id

FOLLOW(E) = {), +, $}
FOLLOW(T) = {*,), +, $}
FOLLOW(F) = {*,), +, $}

HINT: (i) While shifting, shift the symbol from the i/p string to the stack and put the no identified by the shift operation on the top of the stack.

(ii) For reduction, 1st reduce the non-terminal on the stack by the reduction no using that production.
eg: r4 (reduce the production no. 4) and compare the top 2 most element of the stack to get the topmost no of the stack.

step5: Construction of Parsing Table.

I_0:		I_3: {*, +,), $}	
Goto (0,E) = I_1	Goto (0, E) = 1	T → F.	Goto (3, *) = r4
Goto (0,T) = I_2	Goto (0,T) = 2		Goto (3, +) = r4
Goto (0,F) = I_3	Goto (0, F) = 3		Goto (3,)) = r4
Goto (0, () = I_4	action (0, () = S4		Goto (3, $) = r4
Goto (0, id) = I_5	action (0, id) = S5		
I_1:		I_4:	
E' → E.	Goto (1, $) = Accept	Goto (4, E) = I_8	Goto (4, E) = 8
Goto (1, +) = I_6	action (1, +) = S6.	Goto (4, T) = I_2	Goto (4, T) = 2
		Goto (4, F) = I_3	Goto (4, F) = 3
I_2: {), +, $}	Goto (2,)) = r2	Goto (4, () = I_4	action (4, () = S4
E → T.	Goto (2, +) = r2	Goto (4, id) = I_5	action (4, id) = S5
Goto (2, *) = I_7	Goto (2, $) = r2		
	action (2, *) = S7.		

I_5: {*, +,), $} F → id .	Goto(5,*) = r_6 Goto(5,+) = r_6 Goto(5,)) = r_6 Goto(5,$) = r_6
I_6: Goto(6,T) = I_9 Goto(6,F) = I_3 Goto(6,() = I_4 Goto(6,id) = I_5	Goto(6,T) = 9 Goto(6,F) = 3 action(6,() = S_4 action(6,id) = S_5
I_7: Goto(7,F) = I_{10} Goto(7,() = I_4 Goto(7,id) = I_5	Goto(7,F) = 10 action(7,() = S_4 action(7,id) = S_5
I_8: Goto(8,)) = I_{11} Goto(8,+) = I_6	action(8,)) = 11 action(8,+) = 6
I_9: {), +, $} E → E+T . Goto(9,*) = I_7	Goto(9,)) = r_1 Goto(9,+) = r_1 Goto(9,$) = r_1 action(9,*) = S_7
I_{10}: {*, +,), $} T → T* F.	Goto(10,*) = r_3 Goto(10,+) = r_3 Goto(10,)) = r_3 Goto(10,$) = r_3
I_{11}: {*, +,), $} F → (E).	Goto(11,*) = r_5 Goto(11,+) = r_5 Goto(11,)) = r_5 Goto(11,$) = r_5

State	Action						GoTo		
	id	+	*	(	)	$	E	T	F
0	S_5			S_4			1	2	3
1		S_6				Acc			
2		r_2	S_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	S_5			S_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		r_1	S_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			

3.65

Step 3: Tracing the inputs.

1) $E \to E+T$ 4) $T \to F$
2) $E \to T$ 5) $F \to (E)$
3) $T \to T*F$ 6) $F \to id$

Stack	Input	Comment
0	id + id * id $	Action[0,id] = S5
0id5	+ id * id $	Reduce F → id
0F3	+ id * id $	Reduce T → F
0T2	+ id * id $	Reduce E → T
0E1	+ id * id $	Action[1,+] = S6
0E1+6	id * id $	Action[6,id] = S5
0E1+6id5	* id $	F → id
0E1+6F3	* id $	T → F
0E1+6T9	* id $	Action[9,*] = S7
0E1+6T9*7	id $	Action[7,id] = S5
0E1+6T9*7id5	$	reduce F → id
0E1+6T9*7F10	$	reduce T → T*F
0E1+6T9	$	reduce E → E+T
0E1	$	Accept.

(ii) id+id * (id

Stack	Input	Comment
0	id+ id * (id $	Action [0, id] = S5
0id5	+id * (id $	reduce F → id
0F3	+id* (id $	reduce T → F
0T2	+id* (id $	reduce E → T
0E1	+ id *(id ($	Action [1,+] = S6.
0E1+6	Id *(id $	Action [6,id] = S5

0E1+6id5	* (id $	reduce F → id
0E1+6F3	* (id $	reduce T → F
0E1+6T9	* (id $	Action [9,*] = s7
0E1+6T9*7	(id $	Action [7,(] = s4
0E1+6T9*7(4	id $	Action [4,id] = s5
0E1+6T9*7(4id5	$	reduce F → id
0E1+6T9*7(4F3	$	reduce T → F
0E1+6T9*7(4T2	$	reduce E → T
0E1+6T9*7(4E8	$	Error.

2) Consider the grammar
S → L = R
S → R
L → *R
L → id
R → L construct LR parsing table and check whether the input id = id is valid or not.

Step1: Construct the augumented grammar

S' → S
S → L = R
L → *R
L → id
R → L.
S → R

Step2: Find the set of items based on closure operation and goto operation.

Io:
S' → .S
S → .L = R
S → .R
S → .*R

3.67

L → .id

R → .L

Goto (0,S)

I_1: S' → S.

I_2: Goto (0,L)

S → L. = R

R → L.

I_3: Goto (0,R)

S → R.

I_4: Goto (0,*)

L → *.R

R → .L

L → .*R

L → .id

I_5: Goto (0,id)

L → id

I_6: Goto (2,=)

S → L=.R

R → .L

L → .*R

L → .id

I_7: Goto (4,R)

L → *R.

I_8: Goto (4,L)

R → L.

I_4: Goto (4,*)

L → *.R

R → .L

L → .*R

L → .id

I_5: Goto (4,id)

L → id.

I_9: Goto (6,R)

S → L=R.

I_8: Goto (6,L)

R → L.

I_4: Goto (6,*)

L → *.R

R → .L

L → .*R

L → .id

I_5: Goto (6,id)

L → id.

Goto (0,S) = I_1

Goto (0,L) = I_2

Goto (0,R) = I_3

Goto (0,*) = I_4

Goto (0,id) = I_5

Goto (2,=) = I_6

Goto (4,R) = I_7

Goto (4,L) = I_8

Goto (4,*) = I_4

Goto (4,id) = I_5

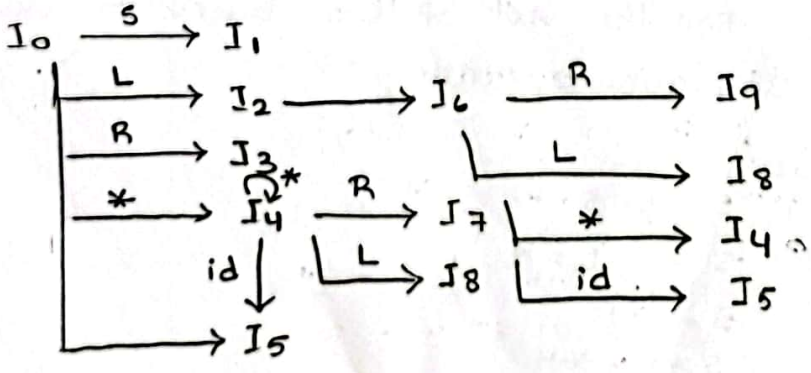
Goto (6,R) = I_9

Goto (6,L) = I_8

Goto (6,*) = I_4

Goto (6,id) = I_5

Step 3: Transition diagram:



Step 4: find the follow of NT from the given grammar

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

FOLLOW(S) = {$}

FOLLOW(L) = {=, $}

FOLLOW(R) = {=, $}

I_0:	
Goto(0,S) = I_1	Goto(0,S) = 1
Goto(0,L) = I_2	Goto(0,L) = 2
Goto(0,R) = I_3	Goto(0,R) = 3
Goto(0,*) = I_4	action(0,*) = S4
Goto(0,id) = I_5	action(0,id) = S5
I_1: {$}	action(1,$) = Accept
$S' \rightarrow S.$	
I_2: {=, $}	action(2,=) = r5
$R \rightarrow L.$	action(2,$) = r5
Goto(2,=) = I_6	action(2,=) = S6
I_3: {$}	action(3,$) = r2
$S \rightarrow R.$	
I_4:	
Goto(4,R) = I_7	Goto(4,R) = 7
Goto(4,L) = I_8	Goto(4,L) = 8
Goto(4,*) = I_4	action(4,*) = S4
Goto(4,id) = I_5	action(4,id) = S5
I_5: {=, $}	action(5,=) = r4
$L \rightarrow id.$	action(5,$) = r4

I_6:	
Goto(6,R) = I_9	Goto(6,R) = 9
Goto(6,L) = I_8	Goto(6,L) = 8
Goto(6,*) = I_4	action(6,*) = S4
Goto(6,id) = I_5	action(6,id) = S5
I_7: {=, $}	action(7,=) = r3
$L \rightarrow *R.$	action(7,$) = r3
I_8: {=, $}	action(8,=) = r5
$R \rightarrow L.$	action(8,$) = r5
I_9: {$}	action(9,$) = r1
$S \rightarrow L = R.$	

3.69

Step 5: Parsing Table

Stack	Action				Goto		
	id	*	=	$	S	L	R
0	S_5	S_4			1	2	3
1				Accept			
2			S_6 / r_5 conflict occurs				
3				r_2			
4	S_5	S_4				8	7
5			r_4	r_4			
6	S_5	S_4				8	9
7			r_3	r_3			
8			r_5	r_5			
9				r_1			

It is not a SLR(1) grammar, since the table contains multiple entries.

Stack	I/P	Comment
0	id = id $	a(0, id) = S5 action.
0 id 5	= id $	a(5, =) = r_4 ($r_4 \Rightarrow L \to id$)
0 L 2	= id $	a(2, =) = S6
0 L 2 = 6	id $	a(6, id) = S5
0 L 2 = 6 id 5	$	a(5, $) = r_4 ($r_4 \Rightarrow L \to id$)
0 L 2 = 6 L 8	$	a(8, $) = r_5 ($r_5 \Rightarrow R \to L$)
0 L 2 = 6 R 9	$	a(9, $) = r_1 ($r_1 \Rightarrow S \to L = R$)
0 S 1	$	a(1, $) = Accept.

Stack	I/P	Comment
0	id = id $	a(0, id) = S5
0 id 5	= id $	a(5, =) = r_4 ($r_4 \Rightarrow L \to id$)
0 L 2	= id $	a(2, =) = r_5 ($r_5 \Rightarrow R \to L$)
0 R 3	= id $	a(3, =) = reject.

Construction of SLR Parsing Algorithm:

Constructing an SLR parsing table.

Input:

An augumented grammar G'

Output:

The SLR parsing table function action and goto for G'.

Method:

Step1: Construct $C = \{I_0, I_1, \ldots I_n\}$ the collection of sets of LR(0) items for G'.

Step2: State I is constructed from I_i. The parsing action 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 a terminal.

(b) If $[A \to \alpha .]$ is in I_i, then set Action $[i, a]$ to "reduce $A \to \alpha$" for all a is in FOLLOW(A). Here A may not be in S'.

(c) If $S' \to S$ is in I_i, then set Action $[i, \$]$ to "ACCEPT". If any conflicting actions results from the above rules, we say the grammar is not SLR(1) grammar. The algorithm fails to produce a parser in this case.

Step3: The goto transition for state I are constructed for all non-terminal A using the rule:

If GOTO $(I_i, A) = I_j$, then GOTO $(i, A) = j$.

3.71

Step 4: All entries not defined by rules 2 and 3 are errors.

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

Canonical LR:

The action and goto functions from the set of LR(1) items are same as such of SLR, but with some i/p symbols.

Construction of Canonical LR Parsing Table.

Input: An augmented grammar G'

Output: The canonical LR parsing table [illegible] for ACTION and GOTO for G'

Method:

1. Construct $C' = \{I_0, I_1, \ldots I_n\}$, the collection of sets of LR(1) items for G'.

2. State i of the parser is constructed from I_i. the parsing action for the state is determined as follows.

(a) If $A \rightarrow \alpha.a\beta, b$ is in I_i and $\text{Goto}(I_i, a) = I_j$ then set action (I, a) to shift j. Here 'a' must be a terminal.

(b) If $A \rightarrow \alpha., a$ is in I_i where $A \neq S'$, ~~then~~

26/02/16 then set ACTION (I, a) to reduce $A \rightarrow \alpha$.

(c) If $S' \rightarrow S., \$$ is in I_i then set ACTION $(i, \$)$ to "ACCEPT", if any conflicting actions result from the above rules we say the grammar is not LR(1) grammar.

The Algorithm fails to produce a parser in this case.

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 $[i, A] = j$.

4. All entries not defined by rules 2 and 3 are made as "Error"

5. The initial state of the parser is the one constructed from the set of items containing $S' \rightarrow .S, \$$

HINT:

No multiple entries and then LR(1) grammar (or) SLR(1) grammar.

If multiple entries, then not LR(1) grammar (or) not SLR(1) grammar.

Canonical LR

Consider the grammar

$S \rightarrow CC$ ... 1

$C \rightarrow xC$ ... 2

$C \rightarrow d$ ... 3

for CLR & LALR

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

$B \rightarrow .\gamma, b$

$b = FIRST(\beta a)$

Step 1: Augmented grammar:

1. $S' \rightarrow S$
2. $S \rightarrow CC$
3. $C \rightarrow xC$
4. $C \rightarrow d$

Step 2: Find the set of items based on the closure and goto operation.

3.73

Consider the grammar

$S \to CC$

$C \to cC$

$C \to d$

I_0:

$A \to \alpha . B\beta, a$ ← put $ as i/p for the initial state

$S' \to .S, \$$

b = FIRST {Ba}

$S \to .CC, \$$ = FIRST {$, $}

b = $

$C \to .cC, c/d$ b = FIRST {Ba}

= FIRST {C $}

= c/d

$C \to .d, c/d$

I_1: Goto(0, S)

$S' \to S., \$$

I_2: Goto(0, C)

$S \to C.C, \$$ b = FIRST {Ba}

$C \to .cC, \$$ = FIRST {$}

$C \to .d, \$$ b = $

I_3: Goto (0, c)

$C \to c.C, c/d$

$C \to .cC, c/d$ b = FIRST {Ba}

$C \to .d, c/d$ b = FIRST {$, c/d}

I_4: Goto (0, d)

$C \to d., c/d$

I_5: Goto (2, C)

$S \to CC., \$$

I_6: Goto (2, c)

$C \to c.C, \$$ b = FIRST {B, a}

$C \to .cC, \$$ = FIRST {$}

$C \to .d, \$$ b = $

I_7: Goto (2, d)

$C \to d., \$$

I_8: Goto (3, C)

$C \to cC., c/d$

I_3: Goto (3, c)

$C \to c.C, c/d$ b = FIRST {Ba}

= FIRST {$, c/d}

$C \to .cC, c/d$ b = c/d

$C \to .d, c/d.$

I_4: Goto (3, d)

$C \to d., c/d$

I_9: Goto (6, C)

$C \to cC., \$$

I_6: Goto (6, c)

$C \to c.C, \$$ b = FIRST {B, a}

$C \to .cC, \$$ b = FIRST {$, $}

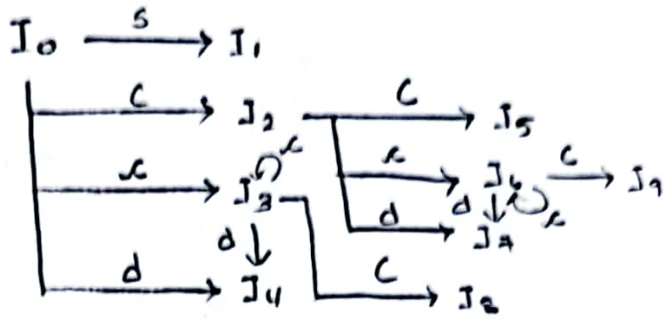
$C \to .d, \$$ b = $

I_7: Goto (6, d)

$C \to d., \$$

Goto (0, S) = I_1
Goto (0, C) = I_2
Goto (0, c) = I_3
Goto (0, d) = I_4
Goto (2, C) = I_5
Goto (2, c) = I_6
Goto (2, d) = I_7
Goto (3, C) = I_8
Goto (3, c) = I_3
Goto (3, d) = I_4
Goto (6, C) = I_9
Goto (6, c) = I_6
Goto (6, d) = I_7

Step3: Draw Transition diagram.



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

S → CC
C → cC
C → d

S → CC C is followed by NT => FOLLOW(C) = FIRST(C) ∪ FOLLOW(S)

I_0: Goto(0,S) = I_1 Goto (0,C) = I_2 Goto (0,c) = I_3 Goto (0,d) = I_4	Goto(0, S) = 1 Goto (0, C) = 2 ACTION (0, c) = S3 ACTION (0, d) = S4
I_1: S' → S. , $	ACTION [1, $] = ACC
I_2: Goto(2,C) = I_5 Goto (2,c) = I_6 Goto (2,d) = I_7	Goto (2,C) = 5 ACTION (2,c) = S6 ACTION (2,d) = S7
I_3: Goto(3,C) = I_8 Goto (3,c) = I_3 Goto (3,d) = I_4	Goto (3,C) = 8 ACTION (3,c) = S3 ACTION (3,d) = S4
I_4: FOLLOW(C) = {c,d,$} C → d. , c/d	ACTION (4,c) = r3 ACTION (4,d) = r3
I_5: {$} S → CC. , $	ACTION (5, $) = r1
I_6: Goto(6,C) = I_9 Goto (6,c) = I_6 Goto (6,d) = I_7	Goto (6,C) = 9 ACTION (6,c) = S6 ACTION (6,d) = S7
I_7: {c,d,$} C → d. , $	ACTION (7, $) = r3
I_8: {c,d,$} C → cC. , c/d	ACTION (8,C) = r2 ACTION (8,d) = r2
I_9: {c,d,$} C → cC. , $	ACTION (9, $) = r2

3.75 :-

States	Action			Goto	
	c	d	$	S	C
0	S3	S4		1	2
1			Acc		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

Step 6: Parse the i/p string dcd.

Stack	i/p	Comment
0	dcd $	A(0,d) = S4
0d4	cd $	ACTION (4,c) = r3 r3 C → d
0C2	cd $	ACTION (2,c) = S6 S6
0C2c6	d$	ACTION (6,d) = S7 S7
0C2c6d7	$	ACTION (7,$) = r3 r3 C → d
0C2c6C9	$	ACTION (9,$) = r2 r2 C → cC
0C2C5	$	ACTION (5,$) = r1 r1 S → CC
0S1	$	ACTION (1,$) = Acc. Accept

LALR :

HINT:

Solve the sum using canonical LR method, by seeing the similarities of states, combine them together so that LALR parsing table is constructed.

States	Action			Goto	
	c	d	$	S	C
0	S36	S47		1	2
1			Acc		
2	S36	S47			5
36	S36	S47			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

Parsing the i/p string dcd.

Stack	i/p	Comment
0	dcd$	A(0,d) = S47
0d47	cd$	ACTION(47,c) → r_3, r_3 C → d
0C2	cd$	ACTION [2,c] → S36 S36
0C2c36	d$	ACTION [36,d] → S47 S47
0C2c36d47	$	ACTION (47,$) → r_3 r_3 C → d
0C2c36C89	$	ACTION (89,$) → r_2 r_2 C → cC
0C2C5	$	ACTION (5,$) → r_1 r_1 S → CC
0S1	$	ACTION (1,$) → Acc Acc

29/02/16

LALR Parsing Table construction:

Input: Augmented grammar G'

Output: LALR parsing table functions ; actions and goto for G'.

3.77

Method:

1. Construct $C = \{ I_0, I_1, \ldots I_n \}$, the collection of set of LR(1) items.

2. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their names.

3. Let $C' = \{ J_0, J_1, \ldots J_n \}$ be the resulting set of LR(1) items, for the parsing actions for state are constructed from J_i in the same manner as that of canonical LR. If there is a parsing action conflict, the algorithm fails to produce a parsing and the grammar is not a LALR(1) grammar.

4. The GOTO table is constructed as follows:

(a) 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,x). (i.e) GOTO(I_1,x), GOTO(I_2,x) ..... GOTO(I_K,x) are same since $\{ I_1, I_2, \ldots I_K \}$ all have the 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

HINT:

In the above parsing Table construction I_3 and I_6 are of same (or) similar states with different i/p

similarly I_4 and I_7 are similar

Also I_8 and I_9 are similar

On combining I_3 and I_6, the state will be I_{36} with the productions,

C → c.C , c/d/$

C → .cC , c/d/$

C → .d , c/d/$

I_{47} would be :

C → d. , c/d/$

I_{89} would be :

C → cC. , c/d , $.

So the canonical parsing table can be reconstructed by combining the similar states to form a LALR parsing table.

Error Recovery in LR Parser :

Consider the grammar

E → E+E

E → E*E

E → (E)

E → id

Construct the LR parsing table.

Step 1: Augmented grammar

E' → E

E → E+E

E → E*E

E → (E)

E → id

Step 2: Find the set of items based on the closure and goto operation.

I_0:

E' → .E

E → .E+E

E → .E*E

E → .(E) E → .id

3.79

I_1: Goto (0, E)

$E' \to E.$
$E \to E. + E$
$E \to E. * E$

I_2: Goto (0, ()

$E \to (.E)$
$E \to .E + E$
$E \to .E * E$
$E \to .(E)$
$E \to .id$

I_3: Goto (0, id)

$E \to id.$

Continuation from the last

I_7: Goto (7, *)

$E \to E*.E$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

I_4: Goto (8, +)

$E \to E+.E$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

I_5: Goto (8, *)

$E \to E*.E$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

I_4: Goto (1, +)

$E \to E+.E$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

I_5: Goto (1, *)

$E \to E*.E$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

I_6: Goto (2, E)

$E \to (E.)$
$E \to E.+E$
$E \to E.*E$

I_2: Goto (2, ()

$E \to (.E)$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

I_3: Goto (2, id)

$E \to id.$

I_7: Goto (4, E)

$E \to E+E.$
$E \to E.+E$
$E \to E.*E$

I_2: Goto (4, ()

$E \to (.E)$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

I_3: Goto (4, id)

$E \to id.$

I_8: Goto (5, E)

$E \to E*E.$
$E \to E.+E$
$E \to E.*E$

I_2: Goto (5, ()

$E \to (.E)$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

Goto (5, id) : I_3

$E \to id.$

I_9: Goto (6,))

$E \to (E).$

I_4: Goto (6, +)

$E \to E+.E$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

I_5: Goto (6, *)

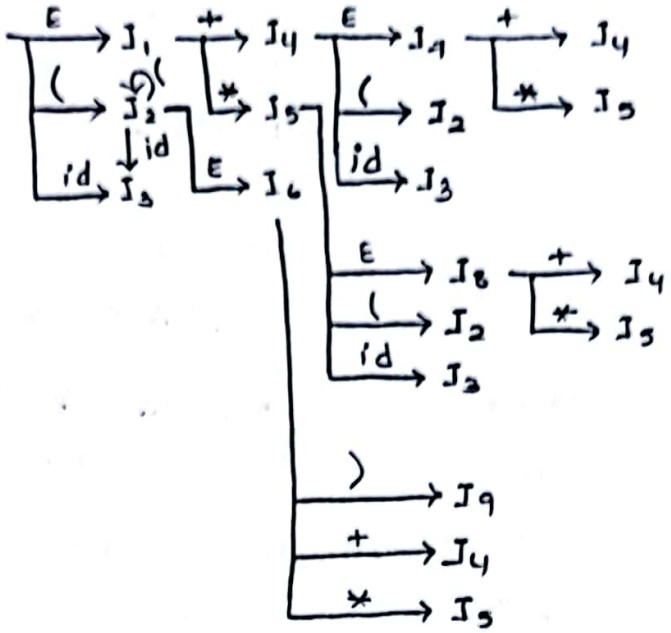
$E \to E*.E$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

I_4: Goto (7, +)

$E \to E+.E$
$E \to .E+E$
$E \to .E*E$
$E \to .(E)$
$E \to .id$

Goto (0,E) = I_1
Goto (0,() = I_2
Goto (0,id) = I_3
Goto (1,+) = I_4
Goto (1,*) = I_5
Goto (2,E) = I_6
Goto (2,() = I_2
Goto (2,id) = I_3
Goto (4,E) = I_7
Goto (4,() = I_2
Goto (4,id) = I_3
Goto (5,E) = I_8
Goto (5,() = I_2
Goto (5,id) = I_3
Goto (6,)) = I_9
Goto (6,+) = I_4
Goto (6,*) = I_5
Goto (7,+) = I_4
Goto (7,*) = I_5
Goto (8,+) = I_4
Goto (8,*) = I_5

Step 3: Transition diagram.



FOLLOW(E) = {+, *,), $}

I_0: Goto (0,E) = I_1 Goto (0,() = I_2 Goto (0,id) = I_3	Goto(0,E) = 1 Goto(0,() = S2 Goto(0,id) = S3
I_1: $E' \rightarrow E.$ Goto(1,+) = I_4 Goto(1,*) = I_5	Accept action (1,+) = S4 action (1,*) = S5
I_2: Goto(2,id) = I_3 Goto(2,E) = I_6 Goto(2,() = I_2	action(2,id) = S3 action(2,E) = S6 action(2,() = S2
I_3: {+,*,),$} $E \rightarrow id$	action (3,+) = r4 action (3,*) = r4 action (3,)) = r4 action (3,$) = r4
I_4: Goto(4,E) = I_7 Goto(4,() = I_2 Goto(4,id) = I_3	Goto(4,E) = 7 action(4,() = S2 action(4,id) = S3
I_5: Goto(5,E) = I_8 Goto(5,() = I_2 Goto(5,id) = I_3	Goto(5,E) = 8 action(5,() = S2 action(5,id) = S3
I_6: Goto(6,)) = I_9 Goto(6,+) = I_4 Goto(6,*) = I5	action(6,)) = S9 action(6,+) = S4 action(6,*) = S5
I_7: $E \rightarrow E+E.$ Goto(7,+) = I_4 Goto(7,*) = I_5	action(7,+) = S4 action(7,*) = S5 action(7,+) = r1 action(7,*) = r1 action(7,)) = r1

3.81

I_8:	action(8,+) = S4	I_9:	action(9,+) = r3
E → (E).	action(8,*) = S5	E → (E).	action(9,*) = r3
Goto(8,+) = I_4	action(8,+) = r2		action(9,)) = r3
Goto(8,*) = I_5	action(8,*) = r2		action(9,$) = r3
	action(8,)) = r2		
	action(8,$) = r2		

	Action						Goto
states	+	*	id	$	(	)	E
0			S3		S2		1
1	S4	S5		Acc			
2			S3		S2		6
3	r4	r4		r4		r4	
4			S3		S2		7
5			S3		S2		8
6	S4	S5				S9	
7	r1 S4	S5 r1		r1		r1	
8	r2 S4	S5 r2		r2		r2	
9	r3	r3		r3		r3	

Error Recovery in LR Parser:

The above grammar is modified for error detections and recovery. Each state is changed to a function call for a particular reduction on some i/p symbols by replacing error entries in that state by reduction e_1, e_2, e_3, e_4 are the error routing that are called when an error occurs.

e1: this routine is called from states 0, 2, 4 and 5 all of which expect begining of an operand either an id (or) a left paranthesis [(]. Instead +, * (or) the end of i/p ($) was found.

Solution:

Push state 3 (the Goto of states 0, 2, 4 and 5 on id)

Error: issue diagnostic "missing operand"

e2: called from states 0, 1, 2, 4 and 5 on finding a right parenthesis.

Solution:

Remove the right parenthesis from the input.

Error: issue diagnostic "unbalanced right parenthesis".

e3: called from states 1 (or) 6 when expecting an operator and an id (or) ~~right~~ left parenthesis is found.

Solution:

Push state 4 (corresponding to symbol + onto the stack)

Error: issue diagnostic "missing operator"

e4: called from state 6, when the end of the i/p is found.

Solution: Push state 9 onto the stack.

Error: issue diagnostic "missing right parenthesis".

State	Action						Goto
	id	+	*	(	)	$	E
0	s_3	e_1	e_1	s_2	e_2	e_1	1
1	e_3	s_4	s_5	e_3	e_2	acc	
2	s_3	e_1	e_1	s_2	e_2	e_1	6
3	r_4	r_4	r_4	r_4	r_4	r_4	
4	s_3	e_1	e_1	s_2	e_2	e_1	7
5	s_3	e_1	e_1	s_2	e_2	e_1	8
6	e_3	s_4	s_5	e_3	s_9	e_4	
7	r_1	r_1	s_5	r_1	r_1	r_1	
8	r_2	r_2	r_2	r_2	r_2	r_2	
9	r_2	r_3	r_2	r_2	r_2	r_2	

3.83

Trace the string "id +)"

Stack	i/p	Comment
0	id +) $	ACTION [0, id] = S_3
0id3	+) $	ACTION [3, +] = r_4
0E1	+) $	ACTION [1, +] = S_4
0E1+4	) $	ACTION [4,)] = e_2
		e_2: unbalanced right parenthesis
0E1+4id3	$	ACTION [4, $] = e_1
		e_1: push id * add
0E1+4E7	$	$E \rightarrow id$.
0E1	$	$r_1 \Rightarrow E \rightarrow E + E$
		Accept.

id id

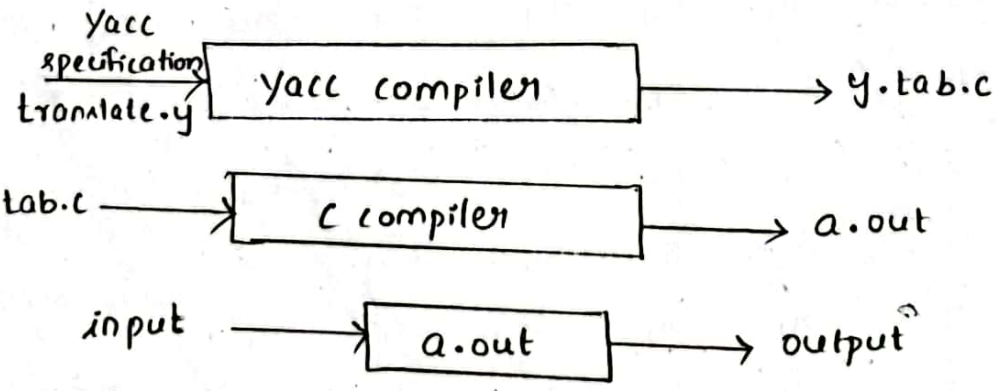
10/3/16

Yacc is meant for LR parser (bottom-up parsing)

Yacc (or) Parser generation – Design of syntax analyzer for a sample language.

1. The parser generator yacc
2. Using yacc with ambiguous grammar
3. Creating yacc lexical analyser with LEX
4. Error recovery in yacc.

1. The parser generator yacc:



Creating an i/p o/p translator with yacc:

declarations
%%
translation rules
%%
supporting C routines.

Declaration Part.

%{ ..... %} ⇒ delimeters

```
#include <ctype.h>  ⇒ C declaration
%token DIGIT ⇒ declaration of grammar tokens
%{
#include <ctype.h>
%}
%token DIGIT
%%
line : expr '\n' {printf ("%d\n", $1);
head ; body
expr : expr '+' term {$$ = $1 + $3;} E → E+T
     | term  E → T
     ;
term : term '*' factor {$$ = $1 * $3;} T → T*F
     | factor  T → F
     ;
factor : '(' expr ')' { $$ = $2;}
     | DIGIT
     ;
%%
yylex () {
  int c;
  c = getchar ();
  if (isdigit (c)) {
  yyval = c - '0';
```

$i - value associated with ith grammar of the symbol of the body

↳ attribute values associated with non-terminal of the head.

→ produce tokens consisting of token name and its attribute value.

return DIGIT; → must be declared before in yacc specification

}

return c;

}

Syntax Translation Rule

Syntax for head and body tag:

<head> : <body>1 {<semantic action>}

| <body>2 {<semantic action>}

Supporting c routine

yylex() produces tokens consisting of a token name and its associated attribute value.

DIGIT is declared in yacc before it is returned in c routine.

3. Creating yacc Lexical Analyzer with Lex.

Lex was designed to produce lexical analyzers that could be used with yacc. The lex library ll will provide the driver program named yylex(), the name required by yacc for its lexical analyzer.

#include "lex.yy.c"

yacc output file y.tab.c

lex first.l → yylex() prode.

yacc second.y

cc y.tab.c - ly - ll

Lex specification for yylex()

```
number [0-9]+.?|[0-9]*\.[0-9]+
%%
[] { /* skip blanks */}
{number} {sscanf (yytext,"%lf",&yylval);
        return NUMBER; }
\n|{return yytext[0];}
```

03/03/16

Using YACC with ambiguous grammar:

```
Lines : lines expr '\n' {printf ("%g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
```

E → E+E | E-E | E*E | E/E | -E | (E) | number

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}
%token number
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines: lines expr '\n' {printf ("%g\n", $2);}
     | lines '\n'
     | /* empty */
     ;
```

Procedure :-

1. exponent
2. bracket
3. UMINUS
4. *, /
5. +, -

```
expr : expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3; }
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     | '(' expr ')' { $$ = $2; }
     | '-' expr %prec UMINUS { $$ = - $2 }
     | NUMBER
     ;
%%
yylex() {
  int c;
  while ((c = getchar()) == ' ');
  if ((c == '.') || (isdigit(c))) {
    ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
```

yacc specification for a more advanced lex calculation.

Error Recovery in YACC:

```
%%
lines: lines expr '\n' { printf("%g\n", $2); }
     | lines '\n'
     | /* empty */
     | error '\n' { yyerror("reenter the previous
                    line"); yyerrorok; }
```

HINT:

yyerrorok starts the normal mode of operation

For any non-Terminal A, The error production can written as,

A → error α, and the state for it can written as,

A → . error α.

When an error is found,

i) Suspend normal parsing when syntax error is found on on i/p line

ii) Starts poping symbols from stack until on counter a state that has a shift action on the token error.

iii) State 0 is the state that is at the bottom of the stack which was mentioned in previous point

iv) Shift new line with happen by emitting error diagnosis " Re-enter the previous line " and thus the error recovery is done by starting from the line when yyerrok gets executed.