2nd UNIT.

SYNTAX - ANALYSIS

- Introduction
- Top - Down Parsing
- Bottom - UP Parsing
- LR, SLR, LALR, CLR
- Using Ambiguous Grammar
- Parser Generators - YACC

Brute forcing Approach

Brute - Force [Eg at d End of Unit]

→ top-down parsing with full backup is a BRUTE-FORCE method of parsing.

Steps

1. Given a particular non-terminal that is to be Expanded, the first prodn for this V is applied.
2. Then, within this newly Expanded string, the next (leftmost) nonterminal is selected for Expansion & its first prodn is applied.
3. (step 2) repeats until process cannot be continued. This termination (if it ever occurs) may be due to Two Causes.

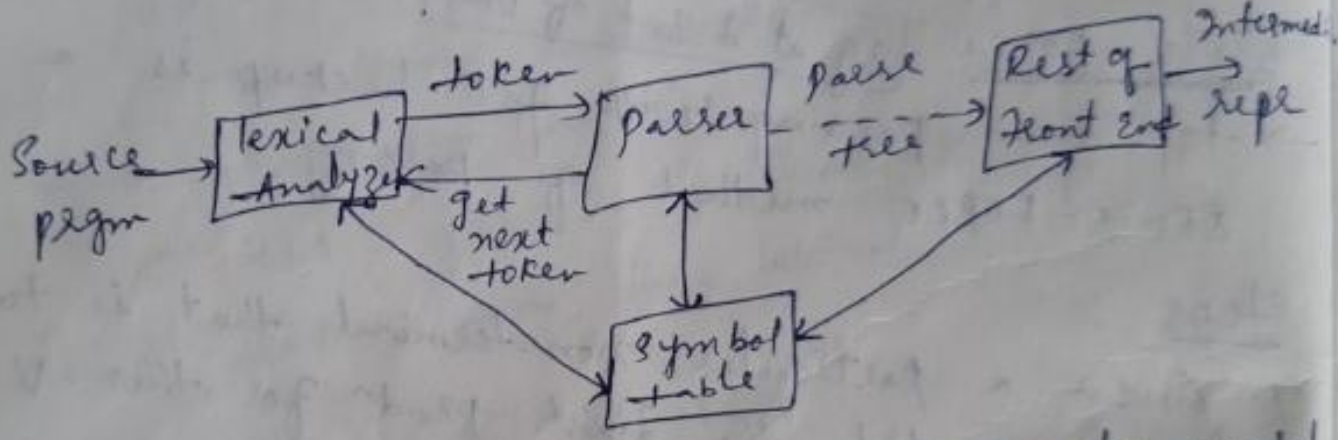
First, no more V may be present, in which case the string has been successfully parsed.

Second, Incorrect Expansion, then the process is "BACKED UP" by Undoing the most recently applied prodn. & then next prodn of this V is used as next Expansion, & then process continues as before.

Introduction

- the Role of the parser
- Representative Grammars
- Syntax Error Handling
- Error - Recovery Strategies

The Role of the Parser



Position of parser in Compiler model

→ parser obtains a string of tokens from the lexical Analyzer and verifies that the string of token names can be generated by the grammar for the source lang.

- we can expect the parser to report any syntax errors to continue in an intelligible fashion and to recover from commonly occuring

Errors to continue processing the remainder of the program.

→ the parser constructs a parse tree and passes it to the rest of the compiler for further processing

- the parse tree need not be constructed explicitly, since checking and translation actions can be ~~interspersed~~ interspersed with parsing.
- the parser and the rest of the front end could well be implemented by a single module.

There are 3 general types of parser for grammars

Universal parser
top-down "
Bottom-up "

Universal parser methods are too inefficient to use in production compilers.

Top-down parser builds parse trees from top (root) to the bottom (leaves)

Bottom-up parser builds parse tree from bottom (leaves) to the top (root).

In either case, the i/p to the parser is scanned from left to right, one symbol at a time.

Representative Grammars

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

- Expression Grammar belongs to the class of LR grammars that are suitable for Bottom-up parsing.
- It can't be used for top-down parsing because it is left recursive.

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

- Non-left-recursive variant of the Expression grammar will be used for top-down parsing.

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

- The following grammar treats + & * alike, so it is useful for illustrating techniques for handling ambiguities during parsing.

Syntax Error Handling

Common programming errors can occur at many different levels.

1. Lexical Errors include misspellings of id's, keywords or operators
   Eg:- elipseSize instead of ellipseSize & misquotes around text.

2. Syntactic Errors include misplaced semicolons or extra or missing braces;

3. Semantic Errors include type mismatches b/w operators & operands

4. Logical Errors assignment operator = instead of the comparison operator ==.

The Error Handler in a parser has goals that are simple to state but challenging to realize:

- Report the presence of errors clearly & accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct prgms.

Error-Recovery Strategies

Panic-mode
phrase-level
Error-productions
Global-correction

Panic-Mode Recovery

- on discovering an error, the parser discards i/p symbols one at a time until one of a designated set of synchronizing tokens is found.
- the synchronizing tokens are usually delimiters, such as semicolon or } whose role in the source prgm is clear & unambiguous.

- panic mode correction often skips a considerable amount of i/p w/o checking it for additional errors,
- it has the advantage of simplicity & is guaranteed not to go into an infinite loop.

phrase-level Recovery

- on discovering an error, a parser may perform local correction on the remaining i/p
- it may replace a prefix of the remaining i/p by some string that allows the parser to continue.
- A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon.
- phrase-level replacement has been used in several error-repairing compilers as it can correct any i/p string.
- its major drawback is the difficulty it has to coping with situations in which the actual error has occured before the point of detection.

Error productions

- Common errors that might be Encountered, we can augment the grammar for the lang at hand with productions that generate the erroneous Construct.
- A parser Constructed from a grammar augmented by these Error productions detects the anticipated errors when an Error production is used during parsing.
- The parser can then generate appropriate Error diagnostics about the erroneous Construct that has been recognized in the i/p.

Global Correction

- we would like a Compiler to make as few changes as possible in processing an incorrect i/p string.
- there are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction.
- unfortunately, these methods are in general too Costly to implement in terms of time & space, so these techniques are currently only of theoritical interest.

Derivation & Parse Trees

- Derivation from S means generation of string w from S.

For constructing derivation two things are imp.

(i) choice of non-terminal from several others.
(ii) choice of rule from production rules for corresponding non-terminal.

Instead of choosing the arbitrary non-terminal one can choose.

(i) either leftmost non-terminal in a sentential form then it is called leftmost derivation.
(ii) or rightmost non-terminal in a sentential form, then it is called rightmost derivation.

$S \to aB \mid bA$
$A \to a \mid aS \mid bAA$
$B \to b \mid bS \mid aBB$

derive the string aaabbabbba using above grammar

Sol leftmost derivation

$S \to aB$
$aaB.B$
$aaaBBB$
$aaabSBB$
$aaabbABB$
$aaabbaBB$
$aaabbabB$
$aaabbabbS$
$aaabbabbbA$
$aaabbabbba$

Derivation & Parse Trees

- Derivation from S means generation of string w from S.

For constructing derivation two things are imp.

(i) choice of non-terminal from several others.
(ii) choice of rule from production rules for corresponding non-terminal.

Instead of choosing the arbitrary non-terminal one can choose.

(i) either leftmost non-terminal in a sentential form then it is called leftmost derivation.
(ii) or rightmost non-terminal in a sentential form, then it is called rightmost derivation.

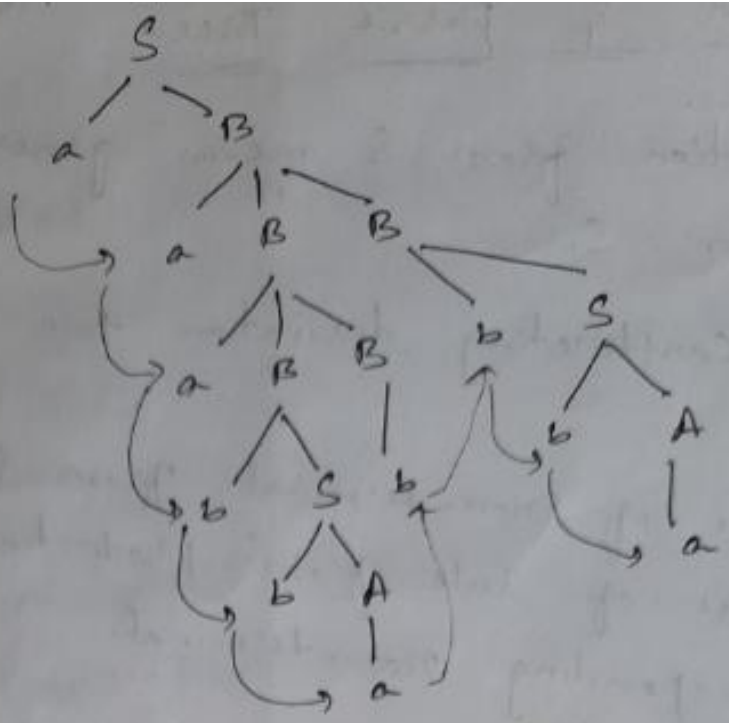
$S \to aB \mid bA$
$A \to a \mid aS \mid bAA$
$B \to b \mid bS \mid aBB$

derive the string aaabbabbba using above grammar

Sol leftmost derivation

$S \to aB$
$aaB.B$
$aaaBBB$
$aaabSBB$
$aaabbABB$
$aaabbaBB$
$aaabbabB$

$aaabbabbS$
$aaabbabbbA$
$aaabbabbba$



Parse tree

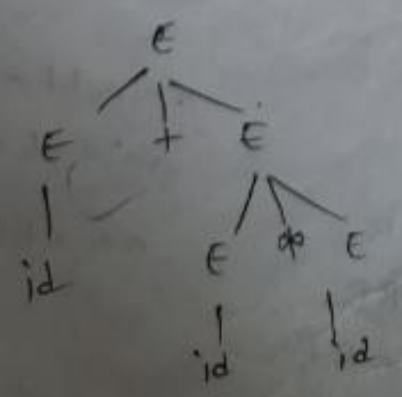
Same derivation for Rightmost derivation & Parse tree.

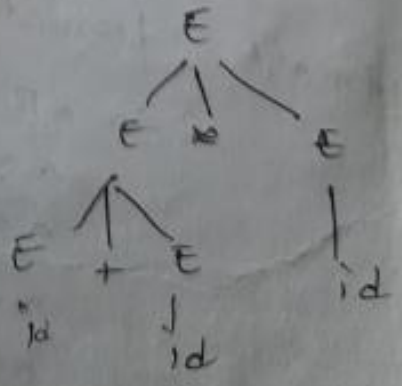
Ambiguous Grammar

A grammar G is said to be ambiguous if it generates more than one parse trees for Sentence of language L(G)

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

then for id + id * id



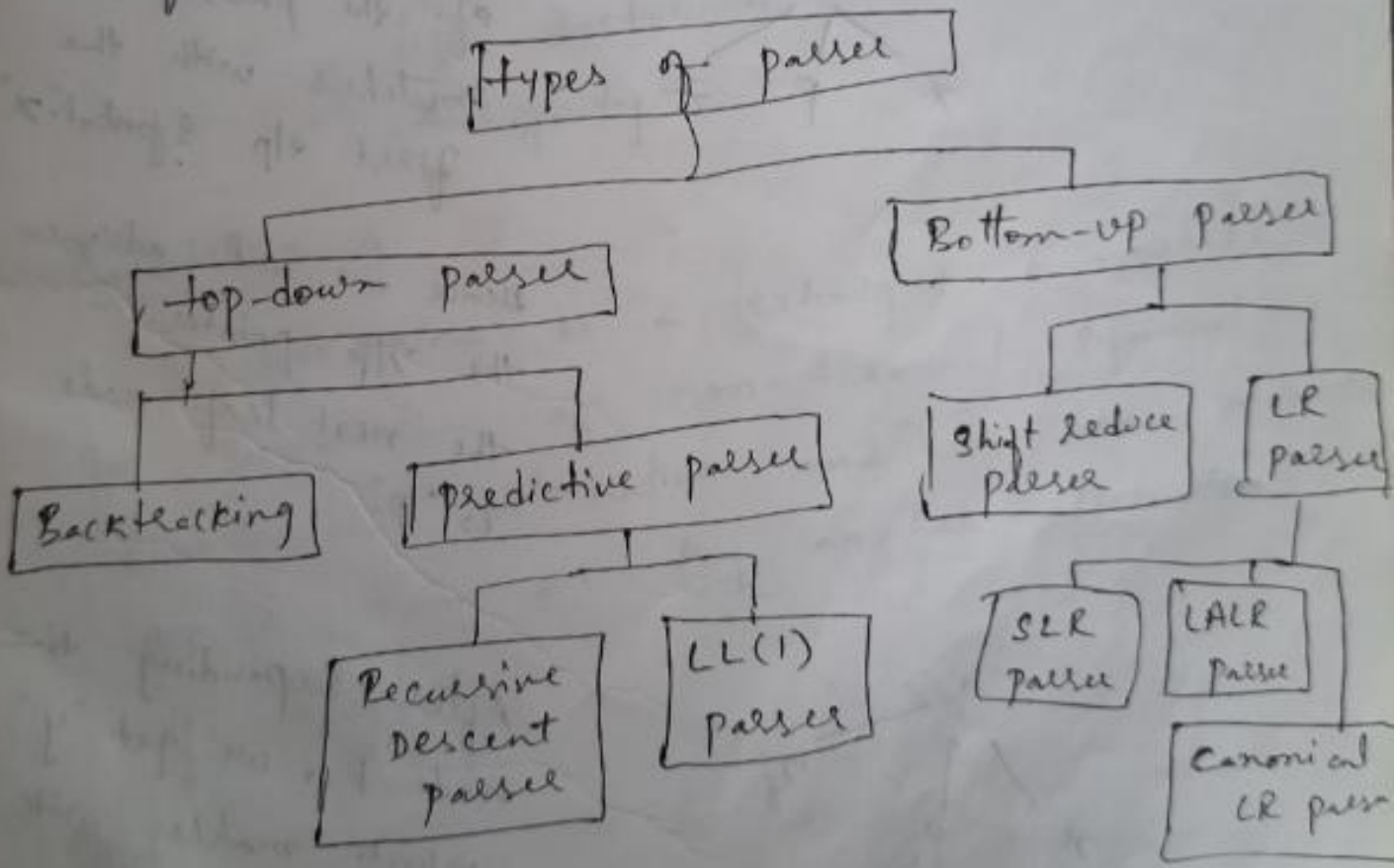


Parsing Techniques

Parsing techniques work on the following principle.

1. The parser scans the i/p string from left to right and identifies that the derivation is leftmost or rightmost.

2. the parser make use of production rules for choosing the appropriate derivation.



Top Down Parser

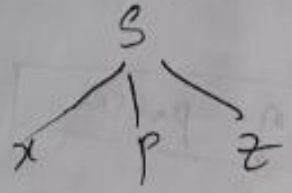
Parse tree is generated from top to bottom

Consider a Grammar

$S \rightarrow xpz$

$P \rightarrow yw \mid y$

i/p :- xyz



the leftmost leaf of the parse tree matches with the first i/p symbol "x"

Hence we will advance the i/p pointer. the next leaf node is "p".

After expanding the node "p", we get "y" which matches with the i/p symbol.

Now the next node is "w" which is not matching with the i/p symbol. Hence we go back to see

whether there is another alternative of "P".

- Now the next i/p symbol is "z" which matches the i/p symbol.
- we halt and declare that the parsing is completed successfully.

Problems with Top-down parsing

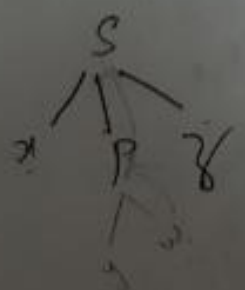
Backtracking
left Recursion
left factoring
Ambiguity

Backtracking

Backtracking is a technique in which for Expansion of non-terminal symbol we choose one alternative and if some mismatch occurs then we try another alternatives if any.

Eg:-

$S \rightarrow xPz$
$P \rightarrow yw \mid y$



or

Left Recursion

Let G be a CFG having a production rule with left Recursion

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

then we eliminate left recursion by re-writing the production rule as:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A'$$
$$A' \rightarrow \varepsilon$$

Eg:- Consider the Grammar

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

$$E \rightarrow E+T \mid T$$
$$A \rightarrow A\alpha \mid \beta$$

$A = E$, $\alpha = +T$, $\beta = T$

then

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

then rule becomes,

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

Similarly for the rule,

$$T \rightarrow T*F \mid F$$

we can eliminate left recursion as

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

So the equivalent grammar will be

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

left factoring

In general if

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

is a production then it is not possible for us to take a decision whether to choose first rule or second.

In such a situation the above grammar can be left factored as

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

For Eg:- $S \rightarrow iEtS \mid iEtSeS \mid a$ (α = $iEtS$, β = eS)

$$E \rightarrow b$$

the left factored grammar becomes

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \varepsilon$$
$$E \rightarrow b$$

(iv) Ambiguity

to remove Ambiguity we will apply one rule: if the grammar has left Associative operator (+, -, *, /) then include the left Recursion and if the grammar has right Associative operator (Exponential operator) then include the right Recursion.

the Unambiguous Grammar is

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow id$

PREDICTIVE PARSER

As the name suggests. the predictive parser tries to predict the next construction using one or more lookahead symbols from i/p string.

There are two types of predictive parsers:

1. Recursive Descent
2. LL(1) parser.

Recursive Descent parser

- A parser that uses collection of Recursive procedures for parsing the given i/p string is called Recursive Descent (RD) parser.

- The R.H.S of the production rule is directly converted to a program.
- For Each non-terminal a separate procedure is written and body of the procedure (code) is R.H.S of the corresponding non-terminal.

Basic steps for Construction of RD parser

- The R.H.S of the rule is directly converted into program code symbol by symbol.
- if the i/p symbol is non-terminal then a call to the procedure corresponding to the non-terminal is made.
- if the i/p symbol is terminal then it is matched with the lookahead from i/p. The lookahead pointer has to be advanced on matching of the i/p symbol.

- if the production rule has many alternates then has to be combined into single body of procedure.
- the parser should be activated by a procedure corresponding to the start sy

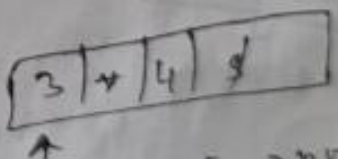
$E \rightarrow num\ T$

$T \rightarrow * num\ T \mid \varepsilon$

```
procedure E
{
    if lookahead = num then
    {
        match(num);
        T;
    }
    else error;
    if lookahead = '$'
    {
        declare success;
    }
    else error;
```

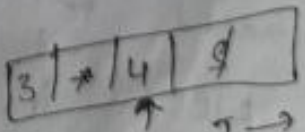
```
procedure T
{
    if lookahead = '*'
    {
        match('*');
```

```
if lookahead = 'num'
{
    match(num);
      T;
    }
    else error;
  }
else NULL;
}

procedure match (token t)
{
  if 'lookahead = t
  lookahead = next_token;
   else error
}

procedure error
{
  print("error!!");
}
```



$\epsilon \rightarrow$ num T

T → * num T

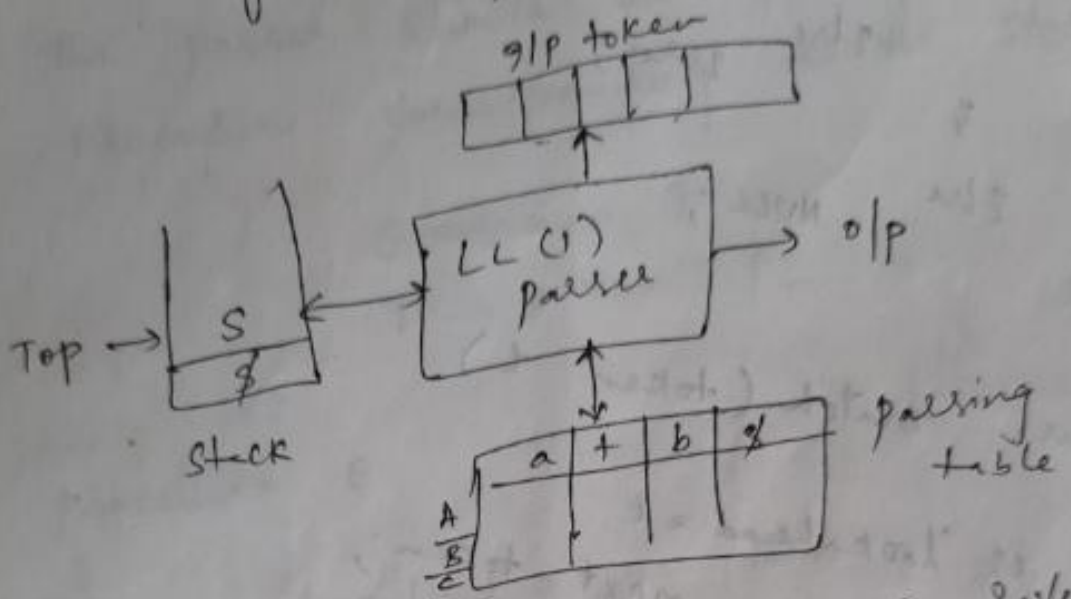


T → + num T

declare success

Predictive LL(1) parser

LL(1) → 1 lookahead
↓ ↘ leftmost derivation
left to right i/p scan



Stack :- the symbols in R.H.S of rule are pushed into the stack in Reverse order

Parsing table :- M[A, a]
↓ terminal
↓ non-terminal

Construction of Predictive LL(1) parser

The Construction of predictive LL(1) parser is based on two very imp fns and those are FIRST & FOLLOW

① Computation of FIRST & FOLLOW fn.
② Construct the predictive parsing table using FIRST & FOLLOW fns.

(3). parse the i/p string with the help of predictive parsing table.

FIRST function

FIRST(α) is a set of terminal symbols that are first symbols appearing at R.H.S in derivation of α.

if $\alpha \Rightarrow \varepsilon$ then ε is also in FIRST(α).

following are the rules used to compute the FIRST fns.

(1). if the terminal symbol "a" the FIRST(a) = {a}.

(2). if there is a rule $X \to \varepsilon$ then FIRST(X) = {ε}

(3). for the rule $A \to X_1 X_2 X_3 \cdots X_k$

FIRST(A) = FIRST(X_1) $\cup$ FIRST(X_2) $\cdots \cup$ FIRST(X_k).

where $k < X_j \leq n$ such that $1 \leq j \leq k-1$

FOLLOW fn.

FOLLOW(A) is defined as the set of terminal symbols that appear immediately to the right of A.

In other words

$FOLLOW(A) = \{a \mid S \Rightarrow \alpha A a \beta$

where α & β are some grammar Symbols may be terminal or non-terminal.

The Rules for Computing FOLLOW

1. For the start symbol S place $ in FOLLOW(S).
2. if there is a production $A \rightarrow \alpha B \beta$ then Everything in FIRST(β) w/o ε is to be placed in FOLLOW(B).
3. if there is a production $A \rightarrow \alpha B \beta$ or $A \rightarrow \alpha B$ and

FIRST(β) = {ε} then

FOLLOW(A) = FOLLOW(B) or

FOLLOW(B) = FOLLOW(A)

Problem for LL(1) Parser

Consider the Grammar

$E \rightarrow TE'$

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

$T \rightarrow FT'$

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

$F \rightarrow (E) \mid id$

Step 1 :- Computing FIRST & FOLLOW

→ $E \rightarrow TE'$,

is a rule in which the first symbol at R.H.S is **T**.

Now $T \rightarrow FT'$ in which the first symbol at R.H.S is **F** &

where $F \rightarrow (E) \mid id$

∴ FIRST(E) = FIRST(T) = FIRST(F)

As $F \rightarrow (E)$

$F \rightarrow id$

Hence FIRST(E) = FIRST(T) = FIRST(F) = {(, id}

→ $E' \rightarrow +TE' \mid \varepsilon$

FIRST(E') = {+, ε}

→ $T' \rightarrow *FT' \mid \varepsilon$

FIRST(T') = {*, ε}

Symbols	FIRST
E	{(, id}
E'	{+, ε}
T	{(, id}
T'	{*, ε}
F	{(, id}

FOLLOW(E)

(i) As there is a rule $F \rightarrow (E)$ the symbol ')' appears immediately to the right of E.

Hence ')' will be in FOLLOW(E)

Since E is a start symbol, add $ to FOLLOW of E.

Hence FOLLOW(E) = {), $}

FOLLOW(E')

(i) $E \rightarrow TE'$

(ii) $E' \rightarrow +TE'$

$E \rightarrow TE'$ the computational rule is $A \rightarrow \alpha B \beta$

$A = E$, $\alpha = T$, $B = E'$, $\beta = \varepsilon$ then by rule 3

Everything in FOLLOW(A) = FOLLOW(B)

i.e FOLLOW(E) = FOLLOW(E')

∴ FOLLOW(E') = {), $}

$E' \rightarrow +TE'$ the computational rule is $A \rightarrow \alpha B \beta$

∴ $A = E'$, $\alpha = +T$, $B = E'$, $\beta = \varepsilon$ then by rule 3

Everything in FOLLOW(A) = FOLLOW(B)

i.e FOLLOW(E') = FOLLOW(E')

∴ FOLLOW(E') = {), $}

FOLLOW(T)

(i) $E \rightarrow TE'$
(ii) $E' \rightarrow +TE'$

$E \rightarrow TE'$ the Computational rule is $A \rightarrow \alpha B\beta$
$A = E$, $\alpha = \varepsilon$, $B = T$, $\beta = E'$ then by Rule 2

$$FOLLOW(B) = \{FIRST(\beta) - \varepsilon\}$$

$$\therefore FOLLOW(T) = \{FIRST(E') - \varepsilon\}$$
$$= \{\{+, \varepsilon\} - \varepsilon\}$$
$$= \{+\}$$

$E' \rightarrow +TE'$ we will map it $A \rightarrow \alpha B\beta$
$A = E'$, $\alpha = +T$, $B = E'$, $\beta = \varepsilon$ then by Rule 3

$$FOLLOW(A) = FOLLOW(B)$$

i.e $FOLLOW(E') = FOLLOW(T)$

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

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

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

FOLLOW(T')

(i) $T \rightarrow FT'$
(ii) $T' \rightarrow *FT'$

$T \rightarrow FT'$ map it with $A \rightarrow \alpha B\beta$ then by Rule 3
$A = T$, $\alpha = F$, $B = T'$, $\beta = \varepsilon$ then by

$$FOLLOW(A) = FOLLOW(B)$$

$FOLLOW(T') = FOLLOW(T)$

$= \{+,), \$\}$

$T' \rightarrow *FT'$ we will map $A \rightarrow \alpha B\beta$ then Rule 3

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

$FOLLOW(A) = FOLLOW(B)$

$FOLLOW(T') = FOLLOW(T)$

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

FOLLOW(F)

(1) $T \rightarrow FT'$

$T' \rightarrow *FT'$

$T \rightarrow FT'$ we will map $A \rightarrow \alpha B\beta$ Rule 2

$A = T, \alpha = \varepsilon, B = F, \beta = T'$ then by

$FOLLOW(B) = \{FIRST(\beta) - \varepsilon\}$

$\therefore FOLLOW(F) = FIRST(T') - \varepsilon$

$= \{*\}$

$T' \rightarrow *FT'$ we will map $A \rightarrow \alpha B\beta$

$A = T', \alpha = *F, B = T', \beta = \varepsilon$ then by Rule 3

$FOLLOW(A) = FOLLOW(B)$

$\therefore FOLLOW(T') = FOLLOW(T)$

$= \{+,), \$\}$

Finally FOLLOW(F) = {*} ∪ {+,), $}

= {+, *,), $}

Symbols	FIRST	FOLLOW
E	{(, id}	{), $}
E'	{+, ε}	{), $}
T	{(, id}	{+,), $}
T'	{*, ε}	{+,). $}
F	{(, id}	{+, *,), $}

STEP 2:- Construction of predictive parsing table.

For the rule $A \to \alpha$ of grammar G.

① for each 'a' in FIRST(α) create entry

$M[A, a] = A \to \alpha$

where 'a' is terminal symbol.

② for ε in FIRST(α) create entry

$M[A, b] = A \to \alpha$

where 'b' is symbols from FOLLOW(A)

③ if ε is in FIRST(α) and $ is in FOLLOW(A) then create entry in the table

$M[A, \$] = A \to \alpha$

4. All the remaining Entries in the table M are marked as SYNTAX ERROR.

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

$E \to TE'$

$A \to \alpha$.

$A = E$, $\alpha = TE'$

$FIRST(TE') = \{(, id)\}$ $\{(\varepsilon)\}$ $\therefore E' = \varepsilon$

$= \{(, id\}$

$M[E, (] = E \to TE'$

$M[E, id] = E \to TE'$

$E' \to +TE'$

$A \to \alpha$

$A = E'$

$\alpha = +TE'$

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

Hence $M[E', +] = E' \to +TE'$

$E' \to \varepsilon$

$A \to \alpha$

$A = E'$, $\alpha = \varepsilon$

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

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

$M[E', \$] = E' \to \varepsilon$

$T \to FT'$

$A \to \alpha$

$A = T$, $\alpha = FT'$

$\therefore T' = \varepsilon$

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

$M[T, (] = T \to .FT'$

$M[T, id] = T \to FT'$

$T' \to *FT'$

$A \to \alpha$

$A = T'$, $\alpha = *FT'$

$FIRST(*FT') = \{*\}$

$M[T', *] = T' \to *FT'$

$T' \to \varepsilon$

$A \to \alpha$

$A = T'$, $\alpha = \varepsilon$

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

$M[T', +] = T' \to \varepsilon$

$M[T',)] = T' \to \varepsilon$

$M[T', \$] = T' \to \varepsilon$

$F \to (E)$

$A \to \alpha$

$A = F$, $\alpha = (E)$

$FIRST((E)) = \{(\}$

$M[F, (] = F \to (E)$

$F \to id$

$A \to \alpha$

$A = F$, $\alpha = id$

$FIRST(id) = \{id\}$

$M[id, id] = F \to id$

Terminals

	id	+	*	(	)	$
E	$E \to TE'$	error	error	$E \to TE'$	error	error
E'	error	$E' \to +TE'$	error	error	$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	error	error	$T \to FT'$	error	error
T'	error	$T' \to \varepsilon$	$T' \to *FT'$	error	$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to id$	error	error	$F \to (E)$	error	error

STEP 3:- parsing the i/p string

i/p :- id + id * id

$E \to TE'$
$E' \to +TE' | \varepsilon$
$T \to FT'$
$T' \to *FT' | \varepsilon$
$F \to (E) | id$

Stack	i/p	Action
$ E	id + id * id $	—
$ E' T	id + id * id $	$E \to TE'$
$ E' T' F	id + id * id $	$T \to FT'$
$ E' T' id	id + id * id $	$F \to id$
$ E' T'	+ id * id $	—
$ E'	+ id * id $	$T' \to \varepsilon$
$ E' T +	+ id * id $	$E' \to +TE'$
$ E' T	id * id $	—
$ E' T' F	id * id $	$T \to FT'$
$ E' T' id	id * id $	$F \to id$
$ E' T'	* id $	—
$ E' T' F *	* id $	$T' \to *FT'$
$ E' T' F	id $	—
$ E' T' id	id $	$F \to id$
$ E' T'	$	—
$ E'	$	$T' \to \varepsilon$
$	$	$E' \to \varepsilon$

BOTTOM-UP PARSING

→ the parse tree is constructed from bottom to up that is from leaves to root.

→ In this process, the i/p symbols are placed at the leaf nodes after successful parsing.

→ In this process, basically parser tries to identify R.H.S of production rule & replace it by corresponding L.H.S.

this activity is called REDUCTION

Eg:-

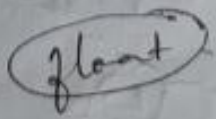
Consider Grammar

S → TL;
T → int | float
L → L, id | id

the i/p string is float id, id;

Parse tree
starting from leaf node

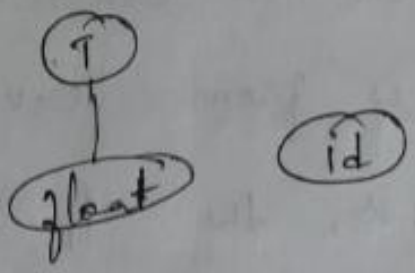
step1:-



Step2:- Reduce float to T

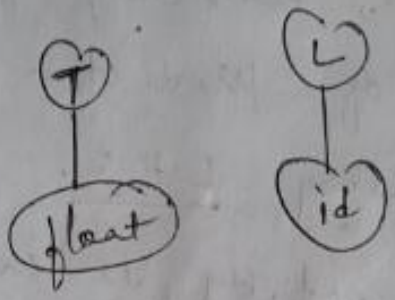
T → float

Step 3 Read next string from i/p

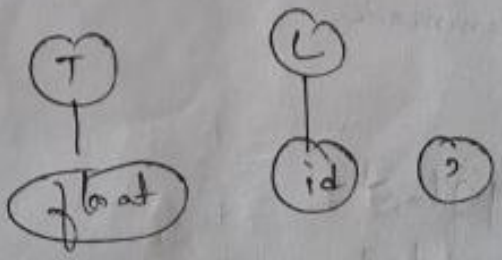


Step 4:- Reduce id to L

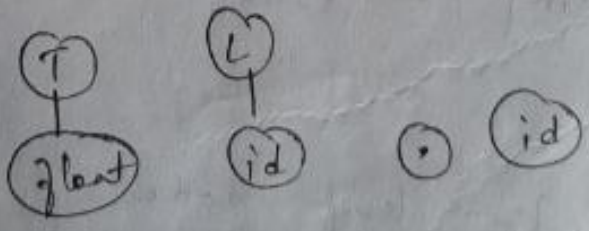
$L \rightarrow id$



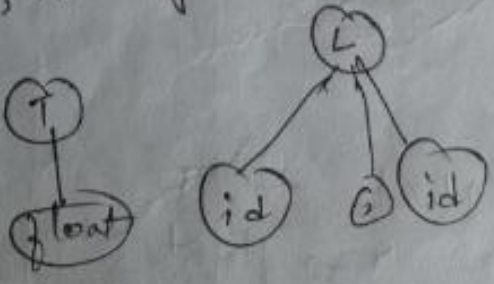
Step 5:- Read next i/p



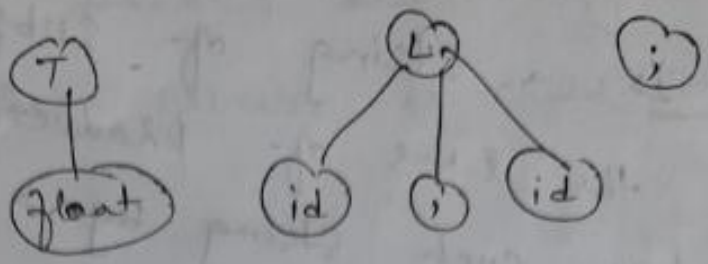
Step 6:- Read next string from i/p



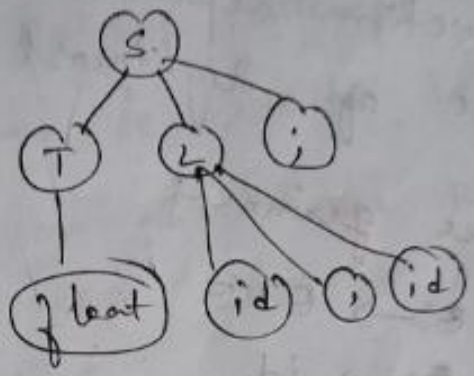
Step 7:- id, id gets Reduced to L



Step 8:- Read next steing ;



Step 9:- TL; reduced to S



Step 10:- The sentential form produced while Constructing this parse tree is

float id, id ;

T id, id ;

T L id ;

T L ;

S

HANDLE PRUNING

- Handle is a string of substring that matches the R.H.S of production and we can reduce such string by a non-terminal on L.H.S of production.

- Such reduction represents one step along the reverse of rightmost derivation.

Consider the grammar

$E \rightarrow E+E$

$E \rightarrow id$

Now the string id+id+id & rightmost derivation is

$E \rightarrow \mathbf{E+E}$

$E \rightarrow \mathbf{E+E+E}$

$E \rightarrow E+E+\mathbf{id}$

$E \rightarrow E+\mathbf{id}+id$

$E \rightarrow \mathbf{id}+id+id$

The Bold strings are called HANDLES

Right Sentential form	Handle	production
id + id + id	id	$E \rightarrow id$
E + id + id	id	$E \rightarrow id$
E + E + id	id	$E \rightarrow id$
E + E + E	E+E	$E \rightarrow E+E$
E + E	E+E	$E \rightarrow E+E$
E	—	—

Thus Bottom parser is essentially a process of detecting Handles and using them in Reduction. This process is called HANDE PRUNING.

SHIFT REDUCE PARSER

→ Shift reduce parser attempts to construct parse tree from leaves to root.

→ Thus it works on the same principle of bottom-up parser.

→ A shift reduce parser requires following Data structures.

① The i/p buffer storing the i/p string.

② A stack for storing & Accessing the L.H.S & R.H.S of rules.

→ The parser performs following basic operations.

① SHIFT : moving of the symbols from i/p buffer onto the stack.

② REDUCE : If the handle appears on the Top of the stack then reduction of it by appropriate rule is done.

that means R.H.S of rule is popped of and L.H.S is pushed in.

(3) ACCEPT :- If the stack contains START Symbol only and i/p buffer is EMPTY at the same time then that action is called ACCEPT.

(4) ERROR :- A situation in which parser cannot either shift or reduce the symbols, it cannot even perform the accept action is called as ERROR.

Ex:-

$E \rightarrow E - E$

$E \rightarrow E * E$

$E \rightarrow id$

perform shift-Reduce parsing of i/p string id - id * id

Stack	i/p buffer	Parsing Action
$	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

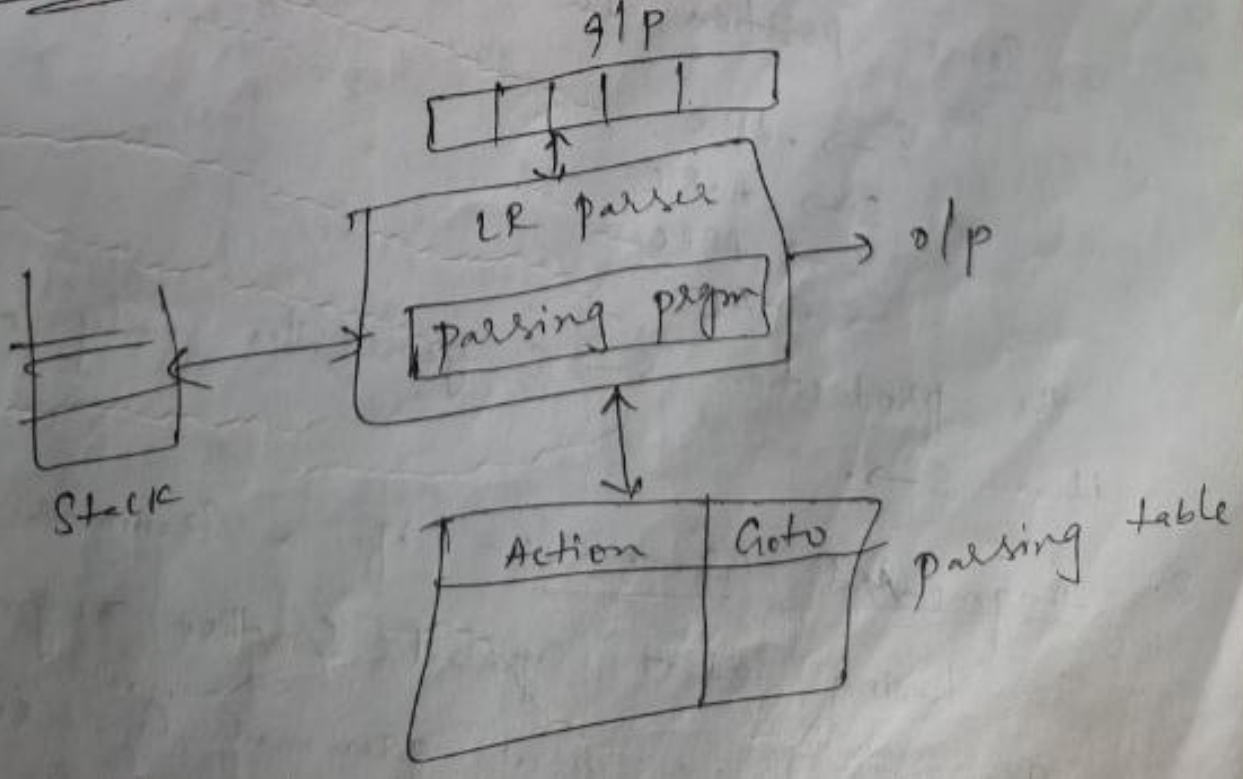
$E-E	$	Reduce E → E-E
$E	$	Accept

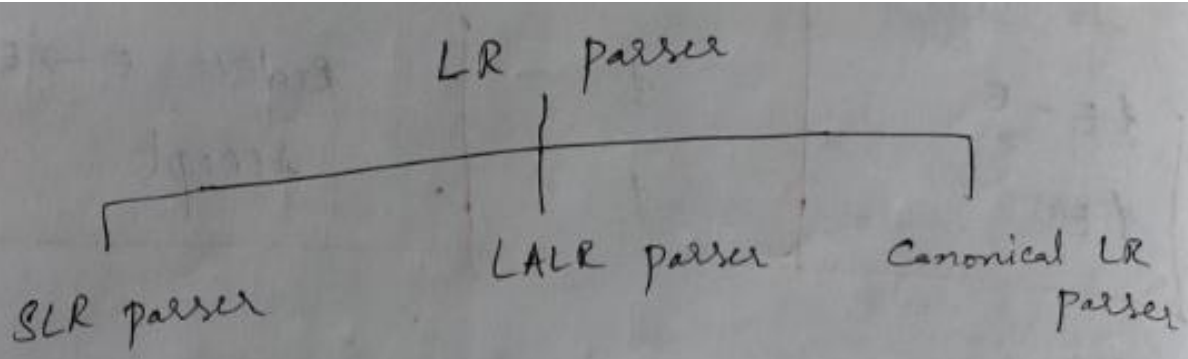
LR PARSERS

- This is the most efficient method of bottom-up parsing which can be used to parse the large class of CFG.
- this method is also called as LR(K) parsing

LR(K) → K omitted, 1 placed.
↳ I/p scan left to right
↳ right most derivation

Structures of LR Parser





SIMPLE LR Parsing SLR

Steps.

1. Construction of canonical set of items.
2. Construction of SLR parsing table.
3. Parsing the i/p string.

Defination of LR(0) items & related terms:-

1. The LR(0) item for grammar G is production rule in which symbol '.' is inserted at some position in R.H.S of the rule

$S \rightarrow .ABC$
$S \rightarrow A.BC$
$S \rightarrow AB.C$
$S \rightarrow ABC.$

the production $S \rightarrow \varepsilon$ generates only one item $S \rightarrow .$

2. Augmented Grammar:- if a Grammar G is having start symbol S then augmented grammar is a new grammar G' in which S' is a new start symbol such

that $S' \rightarrow S$.

The purpose of this grammar is to indicate the acceptance of i/p.

That is when parser is about to reduce $S' \rightarrow S$ it reaches to Acceptance state.

3. Kernel items :- It is a collection of items $S' \rightarrow S$. and all the items whose dots r not at the leftmost end of R.H.S of rule.

Non-Kernel items :- the collection of all the items in which . r at the left end of R.H.S of the rule.

$S \rightarrow A.BC$

4. Closure & goto fn :- these r two imp fns required to create collection of Canonical set of items.

5. Viable prefix :- It is the set of prefixes in the right sentential form of production $A \rightarrow \alpha$. This set can appear on the stack during shift / reduce action.

Closure operation :-

For a CFG G, if I is the set of items then the fn closure(I) can be constructed using following rules.

(1.) Consider I is a set of canonical items & initially every item I is added to closure(I).

(2.) If rule $A \to \alpha \cdot B\beta$ is a rule in closure(I) and there is another rule for B such as $B \to \gamma$ then

closure(I): $A \to \alpha \cdot B\beta$
$B \to \cdot\gamma$

- This rule has to be applied until no more new items can be added to closure(I).
- The meaning of rule $A \to \alpha \cdot B\beta$ is that during derivation of the i/p string at some point we may require strings derivable from $B\beta$ as i/p.
- A non-terminal immediately to the right of · indicates that it has to be expanded shortly.

goto (I_0, b) = closure $\{(S \to bA, S \to b \cdot Ac)$

GoTo operation :-

The function goto can be define as follows.

If there is a production $A \to \alpha . B\beta$ then goto $A \to \alpha B . \beta$.

that means simply shifting of . one position ahead over the grammar symbol (may be terminal / non-terminal).

goto can be written as **goto (I, B)**.

Eg: Consider the following Grammar:

$S \to Aa \mid bAc \mid Bc \mid bBa$

$A \to d$

$B \to d$

compute closure (I) & goto (I).

Sol

closure (I)

I_0 : $S \to .Aa$
$S \to .bAc$
$S \to .Bc$
$S \to .bBc$
$A \to .d$
$B \to .d$

Now applying goto on I_0

I_1 : goto (I_0, A)
$S \to A.a$

goto (I_0, b)

I_2 :
$S \to b.Ac$
$S \to b.Bc$
$A \to .d$
$B \to .d$

goto (I_0, B)

I_3 :
$S \to B.c$

goto (I_0, d)

I_4 : $A \to d.$
$B \to d.$

Prob SLR

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Step 1:-

Construction of canonical collection of set of item:

(1). For the grammar G initially add $S' \rightarrow .S$ in the set of item C.

(2). For each set of items I_i in C and for each grammar symbol X (may be terminal or non-terminal) add closure (I_i, X). This process should be repeated by applying goto(I_i, X) for each X in I_i such that goto(I_i, X) is not Empty and not in C.

The set of items has to constructed until no more set of items can be added to C.

In this grammar we will add the Augmented grammar $E' \rightarrow .E$ in the I then we have to apply closure (I).

I_0:

$E' \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Applying goto on I_0.

goto (I_0, E)

I_1: $E' \rightarrow E.$

$E \rightarrow E.+T$

goto (I_0, T)

I_2: $E \rightarrow T.$

$T \rightarrow T.*F$

goto (I_0, F)

I_3: $T \rightarrow F.$

goto (I_0, ()

I_4: $F \rightarrow (.E)$

goto (I_0, id)

I_5: $F \rightarrow id.$

Applying goto on I_0 is completed.

→ Now applying goto on I_1

In I_1 there r two productions

$E' \to E.$

$E \to E. + T$

1st production is kernel item so ∨ can't apply goto on it.

2nd production applying goto

goto (I_1, +)

I_6:

$E \to E + .T$ → after . non-ter So, need to expand it

$T \to .T * F$

$T \to .F$ → "

$F \to .(E)$

$F \to .id$

→ Now applying goto on I_2

In I_2 there r two productions

$E \to T.$

$T \to T. * F$

1st production is kernel item so ∨ can't apply goto on it.

2nd production ∨ can apply goto

goto (I_2 , *)

I_7 :

$T \rightarrow T * \cdot F$ → After . non-term so, need to expand it

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot id$

→ Now applying goto on I_3.

the production in I_3 is a kernel item so, we can't apply goto.

→ Now applying goto on I_4.

goto (I_4 , E)

I_8 :

$F \rightarrow (E \cdot)$

$E \rightarrow E \cdot + T$

$E \rightarrow \cdot T$ } I_2

$E \rightarrow T \cdot * F$ → I_3

$T \rightarrow \cdot F$ — I_4

$F \rightarrow (\cdot E)$

$F \rightarrow id \cdot$ → I_5

(I_4, T)

(I_4, T)

(I_4, F)

(I_4, ( )

(I_4, id)

→ Repetition of states

→ Now applying goto on I_5.

the production in I_5 is a kernel item. So, no need to apply goto.

→ Now applying goto on I_6.

goto (I_6, T)

I_9:

$E \to E + T.$

$T \to T * F.$

(I_6, F)	$T \to F.$	I_3
$(I_6, ()$	$F \to (.E)$	I_4
(I_6, id)	$F \to id.$	I_5

→ repetation of states.

→ Now applying goto on I_7

goto (I_7, F)

I_{10}:

$T \to T * F.$

$(I_7, ()$	$F \to (.E)$	I_4
(I_7, id)	$F \to id.$	I_5

→ repetation of states

→ Now applying goto on I_8

goto $(I_8,))$

I_{11}:

$F \to (E).$

$(I_8, +)$	$E \to E + .T$	I_6

→ repetation of state

goto on I_9

goto $(I_9, *)$

I_7 $T \to T * .F$ →

→ Applying goto on I_9, I_{10}, I_{11} is of no use becoz either the items are kernel items or repetation of states.

STEP 2:-
Construction of SLR parsing table

(1). Initially construct set of items $C = \{I_0, I_1, I_2 \cdots I_n\}$ where C is a collection of set of LR(0) items for the i/p grammar G'.

(2). The parsing actions are based on each item I_i.

The Actions are

Shift
Reduce
Accept

$T \to F. \quad I_3$

Some answers

$Follow(T) = \{+, *, \$\}$

production no. 2

$action[I_3, +] = action[I_3, *] = action[I_3, \$] = r_2$

(a) Shift :-

If $A \to \alpha . a\beta$ is in I_i and $goto(I_i, a) = I_j$ then set $action[i, a]$ = "shift j".

Note that 'a' must be a terminal symbol.

(b) Reduce :-

If there is a rule $A \to \alpha .$ in I_i then set action$[i, a]$ = reduce $A \to \alpha$ for all symbols a, where $a \in$ FOLLOW(A).

Note that A must not be an augmented grammar S'.

(c) Accept :-

If $S' \to S.$ is in I_i then the entry in the action table action$[i, \$]$ = "accept"

4. The Goto part of the SLR table can be filled as:

the goto transitions for state i is considered for non-terminals only.

if $\boxed{\begin{array}{l} \text{goto}(I_i, A) = I_j \\ \text{goto}[I_i, A] = j \end{array}}$ then

5. All the entries not defined by rule 2 & 3 are considered to be as "error".

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

S, R, A

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				Accept			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Shift

$I_0, id \rightarrow I_5$ – shift 5

$I_0, (\rightarrow I_4$ – s4

⋮

Reduce

Kernel item

$A \rightarrow \alpha.$

↓

$A \rightarrow \alpha$ (line no. of grammar)

$T \rightarrow F.$

Follow(T) = {+, *,), $}

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

I_2:

$E \rightarrow T.$

$E \rightarrow T$

Follow(E) = {+,), $}

2 line of grammar

↓

so r2

goto(I_0, id) = I_5 goto(I_0, E) = I_1

Accept.

$S' \rightarrow S.$

$I_1: E' \rightarrow E.$

$[I_1, \$] = \text{Accept}$

Goto:

$(I_0, E) - I_1$
$(I_0, T) - I_2$
$(I_0, F) - I_3$
$(I_4, E) - I_8$
$(I_4, T) - I_2$
$(I_4, F) - I_3$
$(I_6, T) - 9$
$(I_6, F) - 3$
$(I_7, F) - 10$

STEP 3 parsing the i/p string

i/p string : id * id + id

Stack	i/p buffer	parsing Action
$0	id * id + id	shift id
$0id5	* id + id	reduce F → id
$0F3	* id + id	reduce T → F
$0T2	* id + id	shift *
$0T2*7	id + id	shift id
$0T2*7id5	+ id	reduce F → id
$0T2*7F10	+ id	reduce T → T*F
$0T2	+ id	reduce E → T
$0E1	+ id	shift +
$0E1+6	id	shift id

$0E1+6id5	$	Reduce F → id
$0E1+6F3	$	Reduce T → F
$0E1+6T9	$	Reduce E → E+T
$0E1	$	Accept.

[0, id] = S5 → 5 in parsing string
[0, F] = 3
[0, T] = 2
[2, *] = S7 → 7
[1, +] = S6 → 6
[6, id] = S5 → 5

[7, id] = S5 → 5
[7, F] = 10
[0, E] = 1
[6, F] = 3
[6, T] = 9

LR(K) parser / SIMPLY LR / CANONICAL LR

→ The canonical set of items in the parsing technique in which a lookahead symbol is generated while constructing set of items.

→ Hence the collection of set of items is referred as LR(1).
↓
1 lookahead

Steps

1. Construction of canonical set of items along with the lookahead.
2. Building canonical LR parsing table.
3. parsing the i/p string.

STEP 1 :-

1. For the grammar G initially add $S' \rightarrow .S$ in the set of item C.
2. Same as SLR 2nd step.
3. The closure fn can be computed as follows.

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

$$X \rightarrow \gamma, b \in FIRST(\beta, a)$$

$X \to \cdot \gamma , b$

4. goto Same as SLR.

this process is repeated until no more set of items can be added to the collection C.

Eg:-

$S' \to S$
$S \to CC$
$C \to aC \mid d$

Construct LR(1) set of items for above grammar.

Sol

we initially add $S' \to \cdot S , \$$ as the first rule in I_0.

Now match $S' \to \cdot S , \$$
$A \to \alpha \cdot X \beta , a$

$A = S' , \alpha = \varepsilon , X = S , \beta = \varepsilon , a = \$$

$X \to \gamma , b \in FIRST(\beta , a)$
$S \to \cdot CC , \quad b \in FIRST(\varepsilon \$)$
$b = \$$

$\therefore S \to \cdot CC , \$$ will be added I_0

Now $S \to .CC, \$$ is in I_0. it match

$A \to \alpha . X\beta, a$

$A = S, \alpha = \varepsilon, X = C, \beta = C, a = \$$

$X \to \gamma, \quad b \in FIRST(\beta, a)$

$C \to .aC, \quad b \in FIRST(C, a)$

$C \to .d \qquad (a|d|\$)$

$\$$ not added coz it added only to starting symbol

$\therefore C \to .aC, a|d$
$C \to .d, a|d$ } added in I_0

Hence I_0:

$S' \to .S, \$$
$S \to .CC, \$$
$C \to .aC, a|d$
$C \to .d, a|d$

→ Applying goto on I_0

goto (I_0, S)

I_1:

$S' \to S., \$$

→ goto (I_0, C)

I_2:

$S \to C.C, \$$

$A \to \alpha . X\beta, a$

$A = S, \alpha = C, X = C, \beta = \varepsilon, a = \$$

$X \to \gamma$, $b \in FIRST(\beta, a)$

$C \to .aC$, $b \in FIRST(C, \$)$

$C \to .d$ $= \$$

goto (I_0, C)

I_2: $S \to C.C, \$$

$C \to .aC, \$$

$C \to .d, \$$

goto (I_0, a)

I_3: $C \to a.C, a/d$

$C \to .aC, a/d$

$C \to .d, a/d$

goto (I_0, d)

I_4: $C \to d., a/d$

goto (I_0, a)

$C \to .aC, a/d$

$A \to \alpha.X\beta, a$

$X \to \gamma$

$a \to$?

So rule is not satisfying for

$C \to .aC, a/d$

$C \to .d, a/d$

→ Applying goto on I_1

The production is a kernel item

So we can't apply goto on I_1.

→ Applying goto on I_2

goto (I_2, C)

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

goto (I_2, a)

I_6 :

$C \rightarrow a \cdot C$, $

$C \rightarrow \cdot aC$, $

$C \rightarrow \cdot d$, $

goto (I_2, d)

I_7 : $C \rightarrow d \cdot$, $

→ Applying goto on I_3

goto (I_3, C)

I_8 : $C \rightarrow aC \cdot$, a/d

goto (I_3, a)

$C \rightarrow a \cdot C$, a/d — I_3 → repetition of state

goto (I_3, d)

$C \rightarrow d \cdot$, a/d — I_4

→ Applying goto on I_4

production is kernel item so goto can't be applicable.

→ Applying goto on I_5.

its a kernel item. so can't apply goto

→ Applying goto on I_6

goto (I_6, C)

I_9 :

$C \to aC.\ ,\ \$$

goto (I_6, a)

$C \to a.C,\ \$ \to I_6$

goto (I_6, d)

$C \to d.,\ \$ \to I_7$

→ repeatation of states

→ I_7 kernel item

→ I_8 kernel item & repeatation states

→ I_9 kernel item & repeatation states

So can't apply goto.

States are from $I_0 - I_9$.

STEP 2:-

1. ... Same as SLR step

2. the parsing actions are
   - shift
   - Reduce
   - Accept

(2a) shift :- If $[A \to \alpha.a\beta, b]$ is in I_i and goto $(I_i, a) = I_j$ then create an Entry in the action table

action $[I_i, a] =$ shift j.

b. Reduce :-

If there is a production

$A \to \alpha .$, a then

action[Ii, a] = reduce by

$A \to \alpha$

Here A should not be S'.

c. Accept :-

$S' \to S.$, $ then

action[Ii, $] = Accept

3. goto part is same as SLR

4. Same as SLR

Grammar

① $S \to CC$

② $C \to aC$

③ $C \to d$

States	ACTION			GOTO	
	a	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	γ_3	γ_3			
5			γ_1		
6	S6	S7			9
7			γ_3		
8	γ_2	γ_2			
9			γ_2		

Shift

$[I_0, a] = I_3$ - shift 3 - S3

$[I_0, d] = I_4$ - shift 4 - S4

$[I_2, a] = I_6$ - shift 6 - S6

$[I_2, d] = I_7$

$[I_6, a] = S6$

$[I_6, d] = S7$

Accept Reduce

$A \rightarrow \alpha., a$

$A \rightarrow \alpha$

line no. of grammar

I_4: $C \rightarrow d., a|d$ - line 3

I_5: $S \rightarrow CC., \$$ - line 1

I_8: $C \rightarrow aC., a|d$ - line 2

I_9: $C \rightarrow aC., \$$ - line 2

I_7: $C \rightarrow d., \$$ - line 3

Accept

I_1: $S' \rightarrow S., \$$ - accept

Goto

$[I_0, S] - I_1 - 1$

$[I_0, C] - I_2 - 2$

$[I_2, C] - I_5 - 5$

$[I_3, C] - I_8 - 8$

$[I_6, C] - I_9 - 9$

STEP 3:- LR(0) Grammar & DPDA parsing the i/p string

i/p:- "aadd"

referring the parsing table to parse the i/p string.

stack	i/p buffer	parsing Action
$0	aadd$	shift a
$0a3	add$	shift a
$0a3a3	dd$	shift d
$0a3a3d4	d$	~~shift~~ reduce C→d
$0a3a3C8	d$	reduce C→aC
$0a3C8	d$	reduce C→aC
$0C2	d$	shift d
$0C2d7	$	reduce C→d
$0C2C5	$	reduce S→CC
$0S1	$	Accept

Thus the given string is successfully passed using LR parser (or) canonical LR parser.

LALR parser

→ In this type of parser the lookahead symbol is generated for each set of item.

→ the table obtained by this method are smaller in size than LR(K) parser.

→ In fact the states of CLR and LALR parsing are always same.

STEPS:-

1. Construction of canonical set of items along with the lookahead.
2. Building LALR parsing table.
3. parsing the i/p string using canonical LR parsing table.

STEP 1:-

→ Construction of canonical set of items will be as the LR(1) or CANONICAL LR.

→ But the only difference is that: In construction of LR(1) items for LR parser, we have differed the two states if SECOND component is different

but in this case, we will MERGE the two states by merging of 1st & 2nd Components from both the states.

Ex:-

→ Consider the SAME grammar as LR(1) or Canonical

→ Consider the SAME Canonical set of items as LR(1) / Canonical

$S \to CC$
$C \to aC$
$C \to d$

I_0:
$S' \to .S, \$$
$S \to .CC, \$$
$C \to .aC, a/d$
$C \to .d, a/d$

I_1: goto(I_0, S)
$S' \to S., \$$

I_2: goto(I_0, C)
$S' \to C.C, \$$
$C \to .aC, \$$
$C \to .d, \$$

I_3: goto(I_0, a)
$C \to a.C, a/d$
$C \to .aC, a/d$
$C \to .d, a/d$

I_4: goto(I_0, d)
$C \to d., a/d$

I_5: goto(I_2, C)
$S \to CC., \$$

I_6: goto (I_2, a)

$C \to a.C, \$$

$C \to .aC, \$$

$C \to .d, \$$

I_8: goto (I_3, C)

$C \to aC., a|d$

I_9: goto (I_6, C)

$C \to aC., \$$

I_7: goto (I_2, d)

$C \to d., \$$

Now we will merge 3, 6

4, 7

8, 9

Because they are having 1st component Same.

I_{36}: goto (I_0, a)

$C \to a.C, a|d|\$$

$C \to .aC, a|d|\$$

$C \to .d, a|d|\$$

I_{47}: goto (I_0, d)

$C \to d., a|d|\$$

I_{89}: goto (I_3, C)

$C \to aC., a|d|\$$

STEP 2:-
Construction of parsing table

step 1 :- Construct the LR(1) set of items.

step 2 :- Merge the two states I_i & I_j if the 1st component are matching & create a new state replacing one of the older state such as

$$I_{ij} = I_i \cup I_j$$

Step 3 :- Action table SAME as LR(1) & CANONICAL LR

Step 4 :- GOTO table SAME as LR(1) / CANONICAL LR

Step 5 :- If the parsing action CONFLICT then the algorithm fails to produce LALR parser & grammar is not LALR(1).

All the Entries not defined by Rule 3 & Rule 4 are considered to be "Error".

States	Action			Goto	
	a	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	γ3	γ3	γ3		
5			γ1		
89	γ2	γ2	γ2		

Shift

$[I_0, a]$ – shift 36 – S36

$[I_0, d]$ – S47

$[I_2, a]$ – S36

$[I_2, d]$ – S47

$[I_3, a]$, $[I_6, a]$ – S36

$[I_3, d]$, $[I_6, d]$ – S47

REDUCE

$[I_4, ald\$]$, $[I_7, ald\$]$ — $A \to \alpha.$, $C \to d.$, ald — line 3

$[I_5, \$]$ — $S \to CC.\,,\$$ — line 1 — r_1

$[I_8, a|d|\$]$ } — $C \to aC.\,, a|d|\$$ — line 2 — r_2
$[I_9, a|d|\$]$ }

ACCEPT:

$[I_1, \$]$ $S' \to S.\,, \$$ — Accept

GOTO

$[I_0, S]$ — I_1 - 1

$[I_0, C]$ - I_2 - 2

$[I_2, C]$ - I_5 - 5

$[I_3, C]$ } — I_8 } - 89
$[I_6, C]$ } I_9 }

LR(1)/LR(k) G

STEP 3:- Parsing the i/p string "aadd"

Stack	i/p buffer	parsing action
$0	aadd$	shift a
$0a36	add$	shift a
$0a36a36	dd$	shift d
$0a36a36d47	d$	reduce c → d
$0a36a36C89	d$	reduce c → aC
$0a36C89	d$	reduce c → aC
$0C2	d$	shift d
$0C2d47	$	reduce c → d
$0C2C5	$	reduce S → CC
$0S1	$	Accept

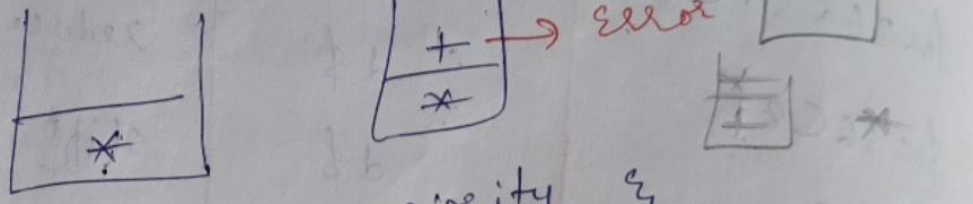
Thus the LALR & LR parser will mimic one another or the same i/p.

HANDLING AMBIGUOUS GRAMMAR

→ Ambiguous Grammar creates CONFLICTS & we cannot parse the i/p string.

→ Conflict can be avoided by using PRECEDENCE & ASSOCIATIVITY.

{ * , + }



* is having priority & + is having less priority. So, we can't push + onto the stack.

→ if S/R conflict then it can be avoided by using SHIFT only.

→ if R/R conflict then it can be avoided by using first Reduce.

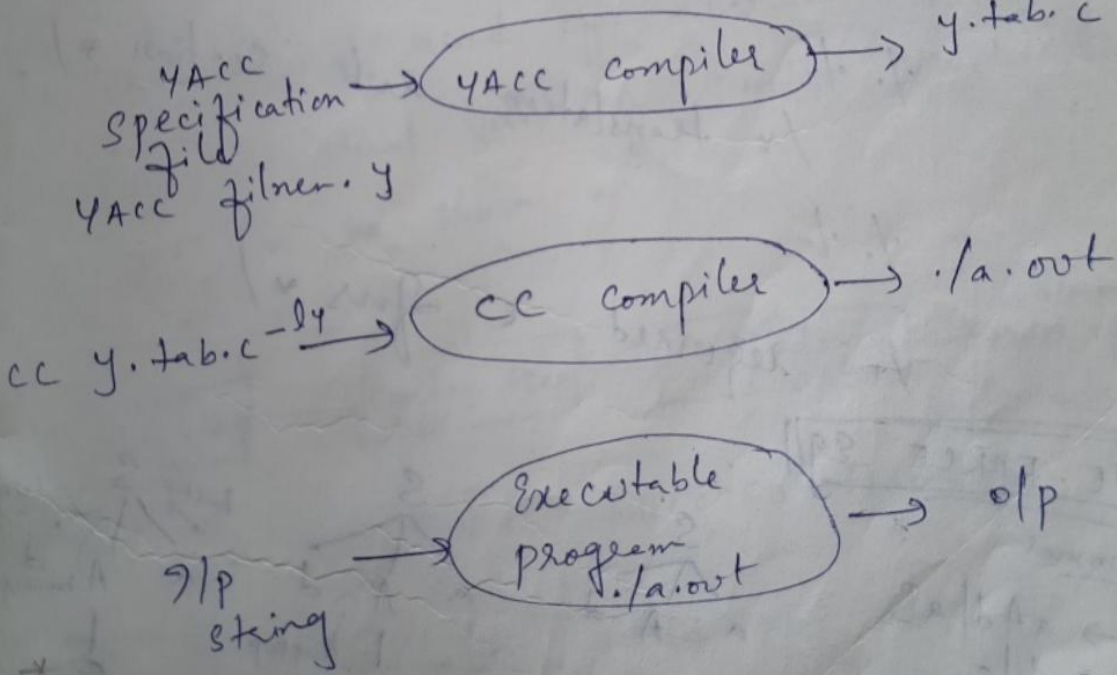
AUTOMATIC PARSER GENERATOR / YACC

→ YACC stands for YET ANOTHER COMPILER COMPILER which is basically the utility available from UNIX.

→ Basically YACC is LALR parser generator.

→ YACC can report conflicts or ambiguities (if at all) in the form of error messages.

The typical YACC translator can be represented as:-



YACC: PARSER GENERATOR MODEL

YACC Specification

→ YACC specification file consists of three parts

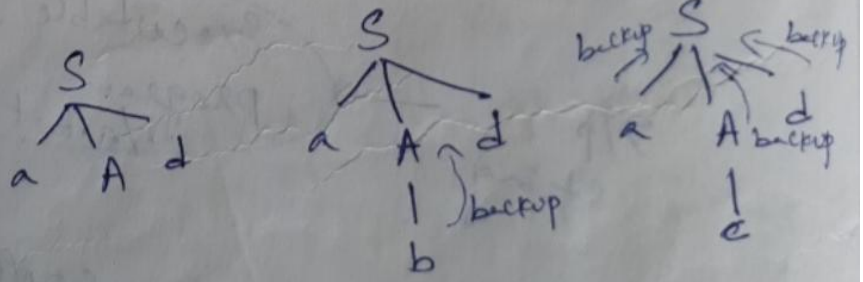
Declarative section
TRANSLATION Rule
C fns

```
%{
    /* declaration section */
%}
%%
    /* translation rule section */
%%
/* required C fns */
```

BRUTE-FORCE Eg

Grammar

$S \rightarrow aAd \mid aB$
$A \rightarrow b \mid c$
$B \rightarrow ccd \mid ddc$



I/p:- accd