Top-Down Parsing

In this approach a parse tree is constructed for the input string, starting from the root and creating the nodes of the parse tree in preorder. Top-Down parsing can be viewed as finding a Left Most derivation for an input string.

* At each step of a top-down parse, the key problem is that of determining the production to be applied for a Non terminal. once an A-production is choosen, the rest of parsing process consists of matching the terminal symbols in the production body with the input string.

The general form of top-down parsing called recursive descent parsing, which may require backtracking to find the correct A-productions to be applied.

General recursive descent may require backtracking, i.e it may requires repeated scans over the input.

Lets take an example for recursive descent parsing which needs backtracking

Example: $S \rightarrow cAd$
$A \rightarrow ab|a$

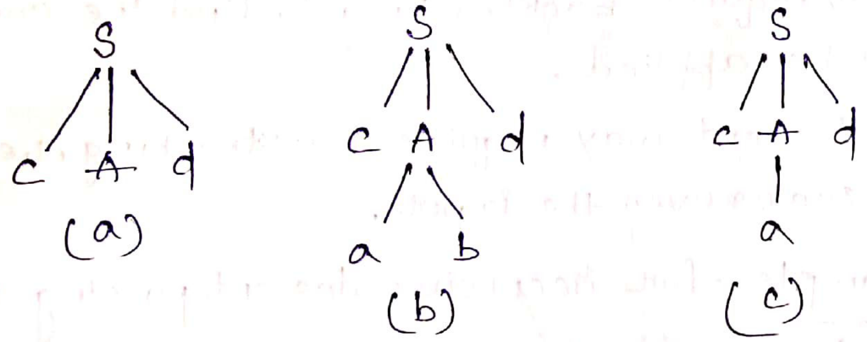
To construct a parse tree for the input string $w = cad$, begin with a tree consisting of a single node labeled S, and the i/p pointer pointing to c, the first symbol of w. S has only one production, so we use it to expand S and obtain the tree shown in fig (a) below. The leaf labeled c, matches the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A. Now we expand A using first alternative $A \rightarrow ab$ to obtain the tree of fig(b). We have a match for the second i/p symbol 'a' so we advance the i/p pointer to 'd', the third i/p symbol, and compare 'd' against the next leaf, labeled 'b'. Since 'b' does not match 'd' we report failure and go back to A and apply another production.

* In going back to A, we must reset the i/p pointer to position 2, the position it had when we first came to A, which means that procedure for A must store the input pointer in a local variable

* The second alternative for A produces the tree of Fig (c). The leaf 'a' matches the second symbol of w and the leaf d matches the third symbol. Since we have produced a parse tree for w, we halt and announces successful completion of parsing.

DRAWBACK : There are few Grammars that can cause a recursive descent parser to enter into an infinite Loop.

One of such Grammar is known as Left Recursive Grammar



DIFFICULTIES WITH TOP DOWN PARSING

There are various difficulties associated with top-down parsing. They are :

1. Backtracking is the Major difficulty with top-down parsing Choosing a wrong production for expansion neccessitates backtracking. Top-down parsing with backtracking involves exponential time Complexity with respect to the length of the input

2. Left recursive grammars cannot be parsed by top-down parsers since they may create an infinite loop.

3. Top-down parsers cannot parse the ambiguous Grammar

4. Top-down parsers are slow and debugging is very difficult.

RECURSIVE DESCENT PARSING

A Top down parser that executes a set of recursive procedures to process the input without backtracking is called Recursive-Descent parser and parsing is called Recursive Descent parsing. The recursive procedures can be easy to write and fairly efficient if written in a language that implements the procedure call efficiently. There is a procedure for each non-terminal in the Grammar. We assume a global variable, lookhead, holding the current input token and a procedure match (Expected token) is the action recognizing the next token in the parsing process and advancing the input stream pointer, such that lookahead points to the next token to be parsed. match() is effectively a call to the Lexical analyzer to get next token. For example input stream is a+b$ then

```
Lookahead ==a
  match()
lookahead==+
  match()
Lookahead ==b
```

Example : Stmt → IDENTIFIER = expr ;

```
Stmt ()
 {
   if(match(IDENTIFIER))
    {
      if(match (=))
       {
         if(expr)
          {
            if(match(;))
             {
             } return(SUCCESS);
          }
       }
   }
   return(FAILURE);
}
```

Example : Consider the following Grammar

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

Write the algorithm of recursive descent parser

```
E()
{
   T();
   E'();
}
T()
{
   F();
   T'();
}
T'()
{
 if(lookhead==*)
   {match();
    F();
    T'();
   }
}

E'()
{
 if(lookhead==+)
  {
   match();
   T();
   E'();
  }
}

F()
{
  if(lookhead==id)
   {
    match();
   }
  else if(lookhead=='(')
     {
      match();
      E();
      if(lookhead==')')
       {
        match();
       }
      else ERROR
     }
    else ERROR
}
```

Eg: Implement Recursive Descent parser for the following Grammar

$S \to AB$
$A \to c | cB$
$B \to d$

Recursive - Descent Parsing

A Recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

Recursive - Descent Parsing

```
  void A() {
1)  Choose an A-production, A → X1X2 .. Xk ;
2)  for (i=1 to K) {
3)        if( Xi is a nonterminal )
4)          call procedure Xi()
5)        else if(Xi equals the current input Symbol a)
6)        advance the input to the next symbol ;
7)        else /* an error has occurred */ ;
      }
  }
```

Fig: A typical procedure for a nonterminal in a top-down parser

General recursive - descent may require backtracking.
To allow backtracking, the above code needs to be modified first we cannot choose a unique A- production at line (1), so we must try each of several productions in some order. Then failure at Line (7) is not ultimate failure, but suggests only that we need to return to line (1) and try another A-production. Only if there are no more A-productions to try then we declare that an input error has been found. In order to retry another production we need to be able to reset the i/p pointer to where it was when we reached line (1). Thus a local variable is needed to store this input pointer for future use

PREDICTIVE PARSING

Predictive parsing is a special case of recursive descent parsing which donot need backtracking. These parsers can function as predictive parser only if the input Grammar is free from left recursion and left factoring.

FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW associated with a Grammar G. During top-down parsing FIRST and FOLLOW allow us to choose which production to apply based on the next input symbol.

FIRST SET : A first set of a Non-terminal A is the set of all the terminals that A can begin with.

Eg: C_Stmt $\rightarrow$ ID EQ_op C_Expression SEMI_COLON

In above production C_stmt begins with ID for a legal input. Hence FIRST set of C_stmt is an ID.

TO COMPUTE FIRST(X) For Grammar SymBOLS X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = $\{X\}$.
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_K$ is a production for some $K \geq 1$, then place a in first(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ... FIRST(Y_{i-1}); that is $Y_1 \cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots K$ then add ϵ to FIRST(X). Everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$ then we add FIRST(Y_2) and so on.
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

Example

$E \rightarrow E+T \mid T$
$T \rightarrow T*F \mid F$
$F \rightarrow (E) \mid id$
} Eliminate left recursion from this Grammar

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

Now let us compute FIRST SET for this Grammar

Consider production (1) $E \rightarrow TE'$ it is in form $X \rightarrow Y_1, Y_2 \cdot\cdot Y_K$

FIRST(E) = FIRST(T)
then calculate FIRST(T)

Consider T's production i.e production (3)
FIRST(T) = FIRST(F)
then calculate FIRST(F)

Consider F's production i.e production (5)
FIRST(F) = FIRST(( ) = {(}
= FIRST(id) = {id}
FIRST(F) = {(, id}

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

Then find FIRST(E') and FIRST(T')

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

FIRST(E') = FIRST(+) ∪ FIRST(ε)
= {+, ε}

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

FIRST(T') = FIRST(*) ∪ FIRST(ε)
= {*, ε}

THE FIRST FUNCTION FOR above Grammar

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

FOLLOW SET :

A follow SET of a nonterminal A is the set of all the terminals that can follow A.

C_Stmt → ID EQ_OP C_Expression SEMI_COLON

By inspecting above production we can figure out that the C_Expression is definitely followed by a SEMI_COLON for a legal input. Hence

FOLLOW(C_Expression) = {SEMI_COLON}

* The FOLLOW set would never contain ϵ.

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

1. place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
2. If there is a production $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B)

* FOLLOW(B) = FIRST(β) − ϵ (OR) $\underbrace{\text{FOLLOW}(B) = \text{FIRST}(\beta)}_{\text{RULE}(2)}$

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

* FOLLOW(B) = {FIRST(β) − ϵ} ∪ FOLLOW(A) [if $A \to \alpha B \beta$ & FIRST(β) contains ϵ]

* FOLLOW(B) = FOLLOW(A) [if $A \to \alpha B$] → Rule 4

EXAMPLE : Compute FOLLOW SET

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

Let us compute FOLLOW(E), to compute this check the production which contains E on right side [consider it as B in Right side production]

FOLLOW(E) : $\underset{\alpha}{(}\underset{B}{E}\underset{\beta}{)}$ with $F \to (E)$ $\Rightarrow$ calculate FIRST(β) i.e FIRST( ))

FIRST( )) = {) }. Since FIRST( )) doesn't contain ϵ apply rule 2 E is start symbol place \$ in FOLLOW(A) = { \$ }

FOLLOW(E) = FIRST( )) $\Rightarrow$ FOLLOW(E) = {), \$ }

FOLLOW(E') : $\underset{A}{E} \to \underset{\alpha}{T}\underset{B}{E'}$ $\Rightarrow$ FOLLOW(E') = FOLLOW(E) = {), \$ }
Apply RULE (4)

FOLLOW(T) : $E \to TE'$ i.e FOLLOW(E') = FOLLOW(E) We are not getting FOLLOW(T) because T is α in this production so consider another production having T at right side & consider it as B

$E' \to \underset{\alpha}{+}\underset{B}{T}\underset{\beta}{E'} | \epsilon$

FIRST(E') is contains ϵ productions so Apply Rule 3.

FOLLOW(T) = FIRST(E') − ϵ ∪ FOLLOW(E')
= { +, ϵ } − ϵ ∪ {), \$ }
= { +,), \$ }

FOLLOW(T') = $\underset{A}{T} \to \underset{\alpha}{F}\underset{B}{T'}$ $\Rightarrow$ FOLLOW(T') = FOLLOW(T)
$\Rightarrow$ FOLLOW(T') = { +,), \$ }

FOLLOW(F) = $T' \to \underset{\alpha}{*}\underset{B}{F}\underset{\beta}{T'}$ $\Rightarrow$ FIRST(T') contains ϵ so Apply Rule 3

FOLLOW(F) = { FIRST(T') − ϵ } ∪ FOLLOW(T')
= { *, +,), \$ }

After computing FIRST and FOLLOW FUNCTIONS construct predictive parsing table M[A, a], a two dimensional array, where A is a Non terminal and 'a' is a terminal or the symbol $, the input end Marker.

ALGORITHM: CONSTRUCTION OF A PREDICTIVE PARSING TABLE

INPUT: GRAMMAR G

OUTPUT: PARSING TABLE M

METHOD: For each production $A \rightarrow \alpha$ of the Grammar, do the following

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

   After performing the above if there is no production at all in M[A, a] then set M[A, a] to error

RECOVERY IN PREDICTIVE PARSING

NON TERMINALS	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Fig: parsing Table M

NONRECURSIVE PREDICTIVE PARSING

A nonrecursive predictive parser can be built by maintaining a stack explicitly. The parser uses Left most derivation. If ω is ~~an~~ the input that has been matched so far, then the stack holds a sequence of grammar Symbol α such that

$$S \xRightarrow[lm]{*} w\alpha$$

The table driven parser has an input buffer, a stack containing a sequence of grammar symbols, a parsing table and an output stream. The input buffer contains the string to be parsed followed by the endmarker \$. We use the same symbol to mark the bottom of the stack, which initially contains the start symbol of the Grammar on top of \$.

The parser is controlled by a program that considers X, the symbol on top of the stack, and a, the current input symbol. If X is a nonterminal, the parser chooses an X- production by consulting the entry M[X, a] of parsing table M. otherwise, it checks the match between the terminal X and current i/p symbol a.

```
    set x to the top stack symbol;
}.
```

MATCHED	STACK	INPUT	ACTION
	E\$	id+id*id \$	
	TE'\$	id+id*id\$	output E→TE'
	FT'E'\$	id+id*id\$	output T→FT'
	idT'E'\$	id+id*id\$	output F→id
id	T'E'\$	+id*id\$	match id
id	E'\$	+id*id\$	output ~~T~~E'→ε
id	+TE'\$	+id*id\$	output E'→+TE'
id+	TE'\$	id*id\$	match +
id+	FT'E'\$	id*id\$	output T→FT'
id+	idT'E'\$	id*id\$	F→id
id+id	T'E'\$	*id\$	match id
id+id	*FT'E'\$	*id\$	output T'→*FT'
id+id*	FT'E'\$	*id\$	match *
id+id*	idT'E'\$	id\$	output F→id
id+id*id	T'E'\$	\$	match id
id+id*id	E'\$	\$	output T'→ε
id+id*id	\$	\$	output E'→ε

ERROR RECOVERY IN PREDICTIVE PARSING

The Error recovery of the parser is the ability to ignore the current error and continue with the parsing for the remainder of the input.
An error is detected during predictive parsing when the terminal on the top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and M[A, a] is error.
The error recovery schemes that are commonly used in predictive parsing are

1. PANIC MODE RECOVERY
2. PHRASE LEVEL RECOVERY

PANIC MODE RECOVERY

It is based on the principle that when an error is detected, the parser should skip the input symbol until it finds the synchronizin token in the input. Usually, the synchronizing tokens are more than one; hence a set called synchronizing set. The effectiveness of panic mode recovery depends on the choice of synchronizing set. some of the guidelines for constructing the synchronizing set are as follows

1. For a Non Terminal A, all the elements of Follow set of A can be added to synchronizing set of A.
2. If we add symbols in FIRST (A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if the symbol in FIRST(A) appears in the input
3. If a terminal on the top of stack cannot be matched, then pop the terminal from the top of stack and issue a warning indicating that the terminal was inserted and continue parsing
4. If a Non-terminal can generate the empty string, then the production deriving ϵ. Can be used as default.

If the parser looks up entry M[A,a] and finds that ~~it~~ is blank, the i/p symbol is (a) is skipped.
If the entry is "synch" then nonterminal on the top of the stack is popped in attempt to resume parsing.
"synch" indicating synchronizing tokens obtain from the FOLLOW set of the NONterminal.
If a token on top of the stack does not match the i/p symbol, then we pop the token from the stack.

NON TERMINAL	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$	synch	synch	$F \to (E)$	synch	synch

on Erroneous input)id*+id

STACK	INPUT	REMARK
E$	)id*+id$	error, skip)
E$	id*+id$	id is in FIRST(E)
TE'$	id*+id$	
FT'E'$	id*+id$	
idT'E'$	id*+id$	
T'E'$	*+id$	
*FT'E'$	*+id$	
FT'E'$	+id$	ERROR, M[F,+] = synch
T'E'$	+id$	F has been poped
E'$	+id$	
+TE'$	+id$	
TE'$	id$	
FT'E'$	id$	
idT'E'$	id$	

$T'E'\$$	$\$$
$E'\$$	$\$$
$\$$	$\$$

2. **PHRASE - LEVEL RECOVERY**

phrase-level recovery in predictive parser can be implemented by filling in blank entries in the predictive parsing table with pointer to error handling (utilities) routines.

Error Handling utility can do the following

1. The error-handling routines can insert, modify or delete any symbol in the input.

2. The routines can also pop elements from the stack.

UNIT-II

2.2 Bottom up parsing: - A bottom-up parsing builds the parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). The sequence of tree snapshots shown in Fig. illustrates a bottom-up parsing of the token stream id * id, with respect to the expression grammar

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \textbf{id}
\end{aligned}
$$

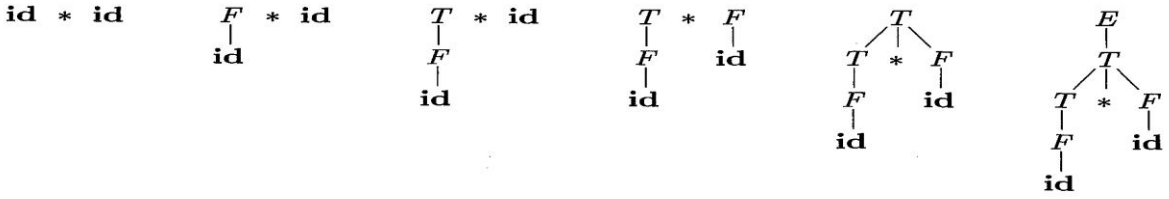


Fig:- A Bottom-up parsing for the string id * id

Reduction:

We can think of bottom-up parsing as the process of “reducing” a string w to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

The key decision during bottom-up parsing is about when to reduce and about what production to apply, as the parse proceeds. The sequence stars with the input string id*id.

First Reduction produces F*id [F->id]

Second Reduction produces T*id [F->T]

Now we have a choice between reducing E→T and the string consisting of second id, which is the body of F→id rather than reducing T to E, id is reduced to T. The parse completes with the reduction of T to the start symbol E.

The goal of bottom-up parsing is to construct a derivation in reverse order.

$$\mathbf{E \Longrightarrow T \Longrightarrow T*F \Longrightarrow T*id \Longrightarrow F*id \Longrightarrow id*id}$$

This derivation is in fact a Right Most Derivation.

The bottom-up parsing can be implemented by using the following techniques.

1. Shift-reduce parsing
2. Operator Precedence parsing
3. SLR parsing
4. Canonical LR(1) parsing
5. LALR(1) parsing

2.2.1. Shift Reduce parsing: - Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. As we shall see, the handle always appears at the top of the stack just before it is identified as the handle. A "handle" is a substring that matches the body of a production, and whose reduction represents one-step along the reverse of a rightmost derivation.

We use $ to mark the bottom of the stack and also the right end of the input. Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather than on the left as we did for top-down parsing. Initially, the stack is empty, and the string w is on the input, as follows:

STACK	INPUT
$	*w* $

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
$\$ S$	$\$$

Upon entering this configuration, the parser halts and announces successful completion of parsing.

There are actually four possible actions a shift-reduce parser can make:

(1) Shift, (2) reduce, (3) accept, and (4) error.

i. **Shift:-** Shift the next input symbol onto the top of the stack.
ii. **Reduce**:- The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
iii. **Accept:-** Announce successful completion of parsing.
iv. **Error:-** Discover a syntax error and call an error recovery routine.

Example: - List out the actions of a shift-reduce parser to parse the input string ***id*** —

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathbf{id} \end{array}$$

STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept

Example:- List out the actions of a shift-reduce parser to parse the input string **id *id+id** according to the following grammar

E → E + E | E * E | (E) | id

Stack Contents	Input Buffer	Action Taken
$	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

Conflicts During Shift-Reduce Parsing:- There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (a shift/reduce conflict), or cannot decide which of several reductions to make (a reduce/reduce conflict). That is the shift-reduce parser has two conflicts. They are,

i. **Shift/reduce conflict**
ii. **Reduce/reduce conflict**

For the following grammar

E → E + E | E * E | (E) | id

Shift/reduce conflict: **-** When the stack and the input buffer contains the contents as shown below.

Stack Contents	**Input Buffer**	**Action Taken**
$E*E	+id$	

The parser can take the action "Shift +" or "Reduce E → E*E ". So it known as Shift/Reduce conflict. To Solve this problem it gives first preference to "Shift + ".

Reduce/reduce conflict: **-** When the stack and the input buffer contains the contents as shown below

Stack Contents	**Input Buffer**	**Action Taken**
$E*E+ E	$	

The parser can take the action "Reduce E → E+E" or "Reduce E → E*E ". So it known as Reduce/Reduce conflict. As Bottom-up parsing uses right most derivation it gives first preference to "Reduce E → E+E ".

Shift/reduce conflict or Reduce/reduce conflict will be encountered for those grammars which are not LR or they are ambiguous. However compilers use LR grammars. Thus Shift/reduce conflict or Reduce/reduce conflict will not occur during compilation process. But Shift-Reduce Parser can't be constructed for a non LR Grammar or ambiguous grammar.

2.3. LR(k) Parsing

The most important type of bottom-up parser is based on a concept called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions. The cases k = 0 or k = 1 are of practical interest, and we shall only consider LR parsers with k<= 1 here. When (k) is omitted, k is assumed to be 1. There are 3 different types of LR parsers. They are

1. **Simple LR(SLR)**
2. **Canonical LR(CLR)**
3. **Lookhead LR(LALR)**

2.3.1. ***LR Parsers:-***LR parsers are table-driven, much like the nonrecursive LL parsers. A grammar for which we can construct a parsing table is said to be an LR grammar. A grammar to be LR, it is sufficient that a left-to-right shift-reduce parser be able to recognize handles of right-sentential forms when they appear on top of the stack.

Advantages of LR parsers:-

i. LR parsers can be constructed to recognize all programming language constructs for which context-free grammars can be written.

ii. The LR-parsing method is the most general Non backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.

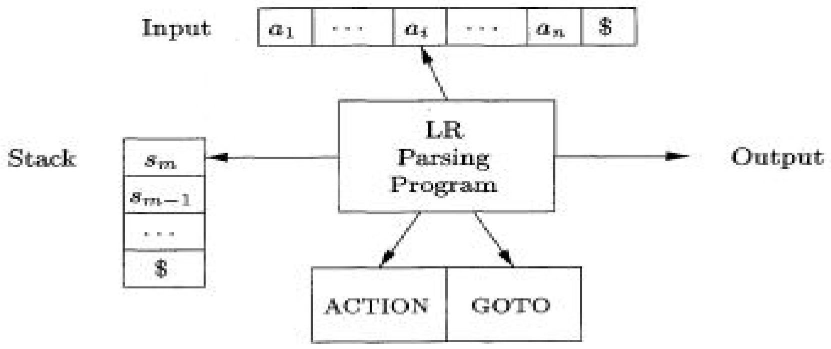
iii. The class of grammars that can be parsed using LR methods is a superset of the class of grammars that can be parsed with the predictive or LL methods.

iv. It can detect a syntactic error as soon as possible to do so on a left-to-right scan of the input.

Disadvantages of LR parsers:-

i. This method involves too much of work to construct a parser by hand: one needs a specialized tool – an LR parser Generator.

2.3.2. The LR-Parsing Algorithm LR parser consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another.



2.3.3. ***Behavior of the LR Parser: -*** The next move of the parser from the configuration above is determined by reading a_i, the current input symbol, and S_M, the state on top of the stack, and then consulting the entry ACTION[S_M, a_i]in the parsing action table. The configurations resulting after each of the four types of move are as follows

1. If ACTION[S_M, a_i]= shift s, the parser executes a shift move; it shifts the next state s onto the stack, The current input symbol is now a_{i+1}.
2. If ACTION[S_M, a_i]= reduce A → β, then the parser executes a reduce move using the production A → β. If β has r symbols, then the parser first popped 2r (only r symbols if we have only states in stack) symbols off the stack and parser then pushed s, the entry for GOTO[S_{M-r}, A], onto the stack.
3. If ACTION[S_M, a_i]= accept, parsing is completed.
4. If ACTION[S_M, a_i]= error, the parser has discovered an error and calls an error recovery routine.

2.4. SLR Parsing or SLR (1) Parsing:

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent sets of "items." An *LR(0) item (item* for short) of a grammar *G* is a production of *G* with a dot at some position of the body.

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.

2. Completing the closure.

3. Compute goto(I,X), where, I is set of items and X is grammar symbol.

- **LR(0) items:**

An *LR(0) item* of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items:
A → **.** XYZ
A → X **.** YZ
A → XY **.** Z
A → XYZ **.**
One collection of sets of LR(0) items, called the *canonical* LR(0) collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an ***LR(0) automaton***. In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection.
To construct the canonical LR(0) collection for a grammar, we define an augmented grammar and two functions, CLOSURE and GOTO.

- **Closure operation:**
  If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:
  1. Initially, every item in I is added to closure(I).
  2. If $A \rightarrow \alpha\ .\ B\beta$ is in closure (I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow .\ \gamma$ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

- **Goto operation:**
  Goto(I, X) is defined to be the closure of the set of all items ***[A→ αX . β]*** such that ***[A→ α . Xβ]*** is in I.

Steps to construct SLR parsing table for grammar G are:
1. Augment G and produce G’
2. Construct the canonical collection of set of items C for G’
3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Construction of SLR tables: - The SLR method for constructing parsing tables is a good starting point for studying LR parsing. The parsing table constructed by this method is an SLR table, and an LR parser using an SLR-parsing table is an SLR parser. The SLR method begins with LR(0) items and LR(0) automata. That is, given a grammar, G, we produce augmented grammar G’, with a new start symbol S’. From G', we construct C, the canonical collection of sets of items for G’ together with the GOT0 function.
The ACTION and GOTO entries in the parsing table are then constructed using the FOLLOW(A) for each nonterminal A of a grammar.
1. Construct C = {I_0, I_1, . . . ,I_n), the collection of sets of LR(0) items for G’.
2. State i is constructed from I_i, . The parsing actions for state i are determined as follows:
   (a) If [A →α.aβ] is in I_i, and GOTO(I_i ,a) = I_j , t hen set ACTION[i , a] to "shift j."
   Here a must be a terminal.
   (b) If [A →α.] is in I_i, then set ACTION[i, a] to "reduce A → a" for all a in FOLLOW(A) here A may not be S’.
   (c) If [S’ → S.] is in I_i ,then set ACTION[i, $1] to "accept ."
3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO(I_i , A) = I_j, then GO TO[i, A] = j.
4. All entries not defined by rules (2) and (3) are made "error."
5. The initial state of the parser is constructed from the set containing [S’→S.].

Example for SLR parsing:
Construct SLR parsing for the following grammar:
G : E → E + T | T
T → T * F | F
F → (E) | id

The given grammar is:
G : E → E + T ------ (1)
E →T ------ (2)
T → T * F ------ (3)
T → F ------ (4)
F → (E) ------ (5)
F → id ------ (6)

Step 1 : Convert given grammar into augmented grammar.
Augmented grammar :
E' → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

Step 2 : Find LR (0) items.
I0 : E' → **.** E
E → **.** E + T
E → **.** T
T → **.** T * F
T → **.** F
F → **.** (E)
F → **.** id

GOTO (I_0 , E)
I_1 : E' → E .
E → E . + T

GOTO (I_0 , T)
I_2 : E → T .
T → T . * F

GOTO (I_0 , F)
I_3 : T → F .

GOTO (I_0 , ( )
I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_0 , id)
I_5 : F → id .

GOTO (I_1 , +)
I_6 : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_2 , *)
I_7 : T → T * . F
F → . (E)
F → . id

GOTO (I_4 , E)
I_8 : F → (E .)
E → E . + T

GOTO (I_4 , T)
I_2 : E → T .
T → T . * F

GOTO (I_4 , F)
I_3 : T → F .

GOTO (I_4 , ( )
I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → id

GOTO (I_4 , id)
I_5 : F → id .

GOTO (I_6 , T)
I_9 : E → E + T .
T → T . * F

GOTO (I_6 , F)
I_3 : T → F .

GOTO (I_6 , ( )
I_4 : F → (. E)

GOTO (I_6 , id)
I_5 : F → id .

GOTO (I_7 , F)
I_{10} : T → T * F .

GOTO (I_7 , ( )
I_4 : F → (. E)
E → . E + T
E → . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_7 , id)
I_5 : F → id .

GOTO (I_8 ,))
I_{11} : F → (E) .

GOTO (I_8 , +)
I_6 : E → E + . T
T → . T * F
T → . F
F → . (E)
F → . id

GOTO (I_9 , *)
I_7 : T → T * . F
F → . (E)
F → . id

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

SLR parsing table:

	ACTION						GOTO		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
I_0	s5			s4			1	2	3
I_1		s6				ACC			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7		r1	r1			
I_{10}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

Blank entries are error entries.

Example: Check whether the input id*id+id is valid or not?.

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		id * id + id $	shift
(2)	0 5	id	* id + id $	reduce by *F* —> id
(3)	0 3	*F*	* id -1- id $	reduce by *T* -+ *F*
(4)	0 2	*T*	* id + id $	shift
(5)	0 2 7	T *	id + id $	shift
(6)	0 2 7 5	T * id	+ id $	reduce by *F* -» id
(?)	0 2 7 10	T * F	+ id $	reduce by *T* ->• *T* * *F*
(8)	0 2	T	+ id $	reduce by *E* -¥ *T*
(9)	0 1	*E*	+ id $	shift
(10)	0 1 6	*E* +	id $	shift
(11)	0 1 6 5	E + id	$	reduce by *F* —> id
(12)	0 1 6 3	*E* + *F*	$	reduce by *T* -» *F*
(13)	0 1 6 9	*E* + *T*	$	reduce by *E* -> *E* + *T*
(14)	0 1	*E*	$	accept

Figure : Moves of an LR parser on id * id + id

LR Parsing Algorithm:

```
Let 'a' be the first symbol of w$
while(1) { /* Repeat Forever */
Let 's' be the state on the top of the stack;
if(ACTION[s,a]==shift j)
{
push j on to the stack;
a be the next input symbol;

else if(ACTION[s,a]==A->β)
{
pop |β| symbols from the stack;
let 't' be the top of the stack;
push GOTO[t,A] on to the stack;
Ouput the production A->β;
}
else if(ACTION[s,a]==Accept) break;
else
call error recovery routine;
}
```

Differences between LR and LL Parsers:

S.No	LR Parsers	LL Parsers
1.	These are bottom up parsers.	These are top down parsers.
2.	This is complex to implement.	This is simple to implement.
3.	LR Grammar is context-free-grammar that may not be free from ambiguity.	LL Grammar is context-free-grammar that is free from left-recursion and ambiguity.
4.	LR parser has 'k' lookahead symbol.	LL parser has only one lookahead symbol.
5.	LR parsers perform two actions ***shift*** and ***reduce*** to construct parsing table.	LL parsers performs two actions namely ***first()*** and ***follow()*** to construct parsing table.
6.	LR parsing is difficult to implement	LL parsers are easier to implement.
7.	These are efficient parsers.	These are less efficient parsers.
8.	It is applied to a large class of programming languages.	It is applied to small class of languages.

MORE POWERFUL LR PARSERS

LR PARSING Techniques can be extended to use one symbol of lookahead on the input. There are two different methods:

1. The "canonical-LR" or just LR method makes full use of the Lookahead symbol(s). This method uses a large set of items called the LR(1) items.
2. The "lookahead-LR" or "LALR" method, which is based on the LR(0) sets of items and has many fewer states than typical parsers based on the LR(1) items. By carefully introducing lookaheads into the LR(0) items we can handle many more grammars with the LALR method than with SLR method, and build parsing tables that are no bigger than the SLR tables. LALR is used in most situations.

CONSTRUCTION OF CLR(1)

In the SLR method state i calls for reduction by $A \to \alpha$ if the set of items I_i contains item $[A \to \alpha .]$ and 'a' is in FOLLOW(A). In some situations, when state i appears on top of the stack, the viable prefix $\beta\alpha$ on the stack is such that βA cannot be followed by a in any right sentential form. Thus, the reduction by $A \to \alpha$ should be invalid on input 'a'.

Example: For the Grammar $S \to L = R$ the state 2 (I_2) has
$S \to R$
$L \to .*R$
$L \to id$
$R \to L$

an item $R \to L.$ which could correspond to $A \to \alpha$, and 'a' could be '=' sign which is in FOLLOW(R). Thus the SLR calls for reduction by $R \to L$ in state 2 with = as the next input. However there is no right sentential-form of the Grammar that begins R=. Thus state 2 which is the state corresponding to viable prefix L only should not really call for reduction of that L to R.

Example id = id $

STACK	SYMBOL	INPUT	ACTION
0		id = id $	S5
0 5	id	= id $	R4
0 2	L	= id $	S6
0 2 6	L =	id $	S5
0 2	L	= id $	R5
0 3	R	= id $	NO ENTRY ERROR

It is possible to carry more information in the state that will allow us to rule out some of these invalid reductions by $A \to \alpha$. We arrange to have each state of a LR parser indicate excatly which input symbols can follow a handle 'α' for which there is a possible reduction to A.
The extra information is provided within the state by redefining items to include a terminal symbol as a second component. The general form of an item becomes $[A \to \alpha \cdot \beta, a]$ where $A \to \alpha\beta$ is a production and 'a' is a terminal or $. We call such an object as an LR(1) item. The 1 refers to the length of the second component, called Lookahead of the item.
The reduction is performed if we have an item $[A \to \alpha \cdot, a]$ we reduce α by A only if the next input symbol is a
The set of 'a's will always be a subset of FOLLOW(A).

CONSTRUCTION OF CANNONICAL SET OF ITEMS ALONG WITH THE LOOKAHEAD.

1. FOR G initially add $S' \to \cdot S$ in the set of item C
2. For each set of items I_i in C and for each Grammar symbol X 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 G. The set of items has be constructed until no more set of items that can be added to C.

3. The closure functions can be computed as follows
   For each item $A \to \alpha . X\beta , a$ and for each rule $X \to \gamma$ and $b \in FIRST(\beta a)$ then add $X \to . \gamma , b$ to I if $[X \to . \gamma, b]$ is not in I.

EXAMPLE : CONSTRUCT CLR PARSING TABLE for the Given Grammar

$S \to CC$
$C \to cC \mid d$

consider Augmented Grammar

$S' \to .S$
$S \to CC$
$C \to cC \mid d$

Start by computing the closure of $[S' \to .S, \$]$. To find the closure match $[S' \to .S, \$]$ with the item $[A \to \alpha . B\beta , a]$. That is $A = S$, $\alpha = \epsilon$, $B = S$, $\beta = \epsilon$ and $a = \$$. Function CLOSURE tells us to add $[B \to .\gamma, b]$ for each production $B \to \gamma$. and terminal b in $FIRST(\beta a)$.

I_0

$S' \to .S \$$
$S \to .CC \$$
$C \to .cC, c$
$C \to .cC, d$
$C \to .d, c$
$C \to .d, d$

$B = S$ $\beta = \epsilon$ $a = \$$
$FIRST(\epsilon \$) = \{\$\}$
$B = C$ $\beta = C$ $a = \$$
$FIRST(C\$) = FIRST(C) = \{c, d\}$

$GOTO(I_0, S) : I_1$

$S' \to S. , \$$

$GOTO(I_0, C) : I_2$

$S \to C.C, \$$
$C \to .cC, \$$
$C \to .d, \$$

$B = C$, $\beta = \epsilon$, $a = \$$
$FIRST(\epsilon \$) = \$$

$GOTO(I_0, c) : I_3$

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

$GOTO(I_0, d) : I_4$

$C \to d. \; c$
$C \to d. \; d$

$GOTO(I_2, C) : I_5$

$S \to CC. , \$$

$GOTO(I_2, c) : I_6$

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

$GOTO(I_2, d) : I_7$

$C \to d. , \$$

$GOTO(I_3, C) : I_8$

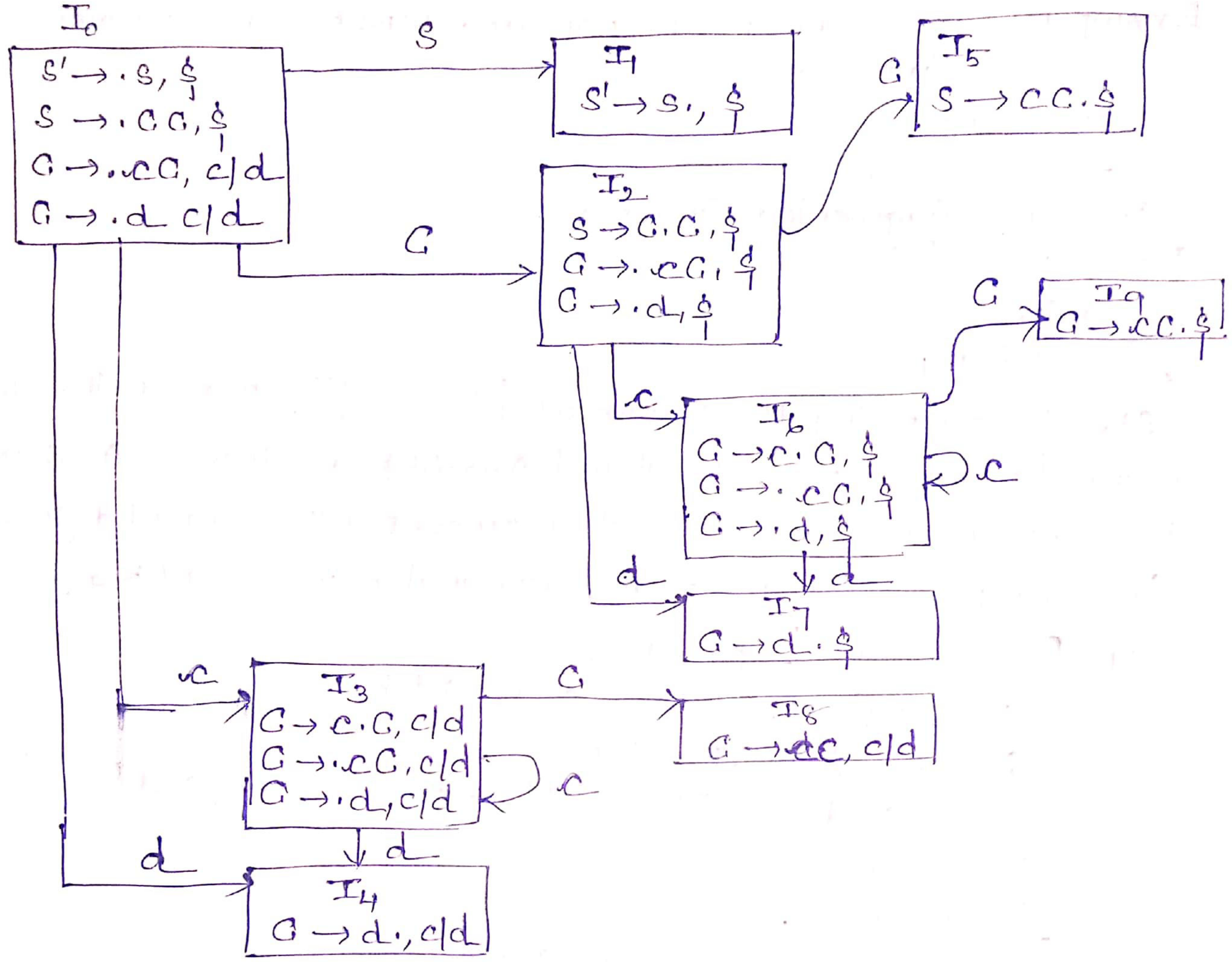
$C \to cC. , c$
$C \to cC. , d$

$GOTO(I_3, c) : I_3$
$GOTO(I_3, d) : I_4$

GOTO(I_6, C) : I_9

$\boxed{C \rightarrow cC\cdot, \$}$

The GOTO GRAPH For the above Grammar is



I_0
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d\ c/d$
S
I_1
$S' \rightarrow S\cdot, \$$
C
I_2
$S \rightarrow C\cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$
C
I_5
$S \rightarrow CC\cdot \$$
c
I_6
$C \rightarrow c\cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$
c
C
I_9
$C \rightarrow cC\cdot \$$
d
d
I_7
$C \rightarrow d\cdot \$$
c
I_3
$C \rightarrow c\cdot C, c/d$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$
c
C
I_8
$C \rightarrow cC, c/d$
d
d
I_4
$C \rightarrow d\cdot, c/d$

* **CONSTRUCTION OF CANONICAL LR PARSING TABLE**

1. If $[A \rightarrow \alpha\cdot a\beta, b]$ is in I_i and GOTO$(I_i, a) = I_j$ then set ACTION$[i, a]$ to "shift j" 'a' must be terminal
2. If $[A \rightarrow \alpha\cdot, a]$ is in I_i then set ACTION$[i, a]$ to "reduce $A \rightarrow \alpha$". Here A may not be S.
3. If $[S' \rightarrow S\cdot, \$]$ is in I_i then set ACTION$[i, \$]$ to "accept".
4. If GOTO(I_i, A) : I_j then GOTO$[i, A] = j$
5. All entries not defined by above rules are made "ERROR"

CANONICAL PARSING TABLE

STATE	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		

Parsing the input string "cdcd" using LR(1) parsing Table

STACK	SYMBOLS	INPUT BUFFER	ACTION
0		cdcd$	SHIFT 3
0 3	c	dcd$	SHIFT 4
0 3 4	cd	cd$	REDUCE 3 $C \to d$
0 3 8	cC	cd$	REDUCE 2 $C \to cC$
0 2	C	cd$	SHIFT 6
0 2 6	Cc	d$	SHIFT 7
0 2 6 7	Ccd	$	REDUCE 3 $C \to d$
0 2 6 9	CcC	$	REDUCE 2 $C \to cC$
0 2 5	CC	$	REDUCE 1 $S \to CC$
0 1	S	$	ACCEPT

LALR PARSER

The last parser in LR parsers is LALR (Lookahead - LR) parser. This method is often used in practice, because the tables obtained by it are considerably smaller than the CLR tables. common syntactic constructs of programming languages can be easily expressed by an LALR Grammar.

For comparision of parser size, the SLR and LALR tables for a Grammar always have the same no. of states, and this number is typically have several hundred states for the same size language. Thus it is much eaiser and more economical to construct SLR and LALR table than CLR tables.

CONSTRUCTION OF LALR :

1. Construct the collection of sets of LR(1) items
2. Merge the two states I_i and I_j if the first component are matching & replace the two states with merged state i.e $I_{ij} = I_i \cup I_j$
3. Build the LALR parse table similar to LR(1) parse table
4. parse the i/p string using LALR parse table similar to LR(1) parsing.

CONSTRUCT LALR PARSING TABLE FOR THE FOLLOWING GRAMMAR

$S \rightarrow CC$
$C \rightarrow cC|d$

the augmented Grammar is
$S' \rightarrow S$
$S \rightarrow CC$
$C \rightarrow cC|d$

Same as CLR, construct LR(1) items

$[A \rightarrow \alpha., \$ \quad]$ I_i $\qquad$ I_{ij} $[A \rightarrow \alpha., \$, a, b]$
$[A \rightarrow \alpha., a, b]$ I_j

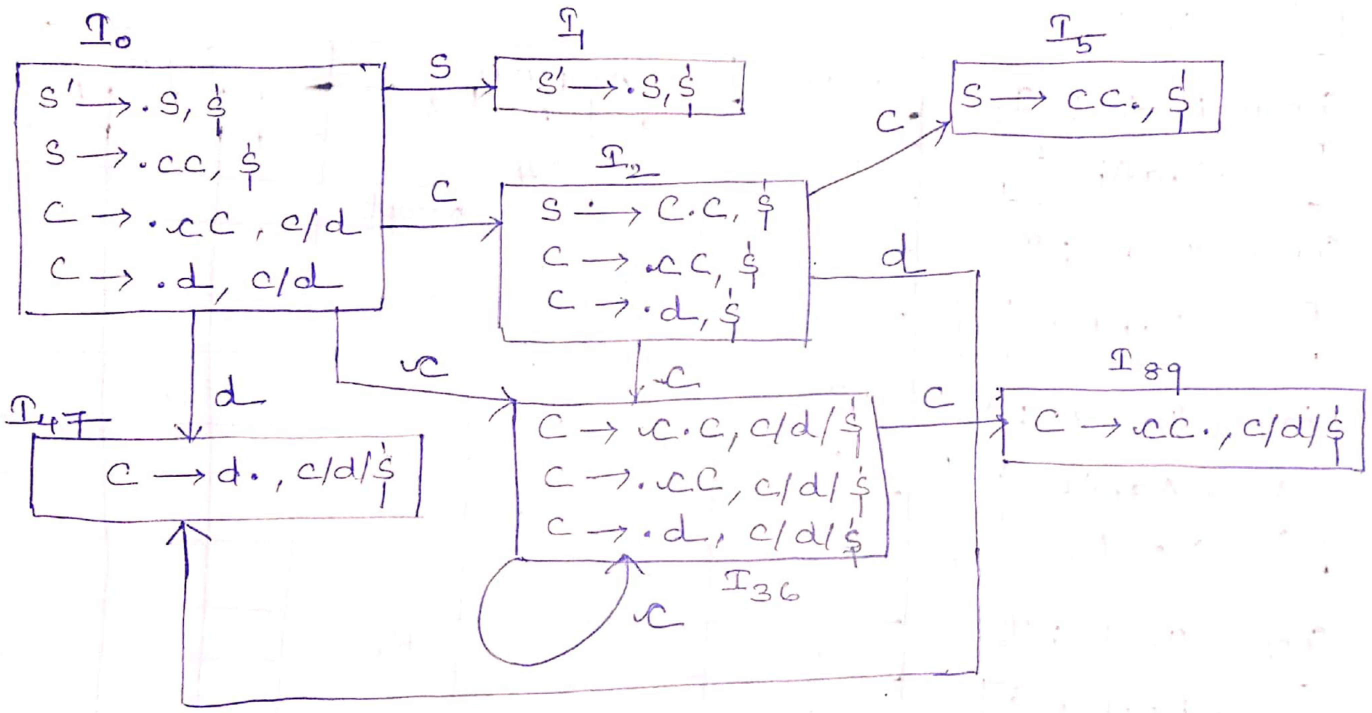
* consider I_4 $[C \rightarrow d. \; c/d]$ same as I_7 $[C \rightarrow d. \; \$]$
I_{47} $[C \rightarrow d. \; \$, c, d]$

* consider I_3 $\begin{bmatrix} C \rightarrow c.C, c \\ C \rightarrow .c.C, d \\ C \rightarrow .d, c \end{bmatrix}$ same as I_6 with Lookahead as $\$$

I_{36} , I_{89}

STATE	ACTION c	ACTION d	ACTION $	GOTO S	GOTO C
0	S36	S47		1	2
1			Acc		
2	S36	S47			5
36	S36	S47			89
47	R3	R3	R3		
89	R2	R2	R2		
5			R1		

GOTO GRAPH



PARSING THE STRING cdcd USING LALR PARSING

STACK	SYMBOLS	INPUT BUFFER	ACTION
0		cdcd$	SHIFT 36
0 36	c	dcd$	SHIFT 47
0 36 47	cd	cd$	Reduce3 (C→d)
0 36 89	cC	cd$	Reduce 2 (C→cC)
0 2	C	cd$	SHIFT 36
0 2 36	Cc	d$	SHIFT 47
0 2 36 47	Ccd	$	Reduce 3 (C→d)
0 2 36 89	CcC	$	Reduce 2 (C→cC)
0 2 5	CC	$	Reduce1 (S→CC)
0 1	S	$	Accept

Construct CLR parsing Table for the following Grammar and check if it is CLR(1).

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

0 $S' \to S$
1 $S \to AaAb$
2 $S \to BbBa$
3 $A \to \epsilon$
4 $B \to \epsilon$

The augmented Grammar

I_0

0 $S' \to .S, \$$
$S \to .AaAb, \$$
$S \to .BbBa, \$$
$A \to ., a$
$B \to ., b$

Goto (I_0, S) : I_1

$S' \to S., \$$

Goto(I_0, A) : I_2

$S \to A.aAb, \$$

Goto (I_0, B) : I_3

$S \to B.bBa, \$$

Goto (I_2, a) : I_4

$S \to Aa.Ab, \$$
$A \to ., b,$

Goto (I_3, b) : I_5

$S \to Bb.Ba, \$$
$B \to ., a$

Goto (I_4, A) : I_6

$S \to AaA.b, \$$

Goto (I_5, B) : I_7

$S \to BbB.a \$$

Goto (I_6, b) : I_8

$S \to AaAb., \$$

Goto (I_7, a) : I_9

$S \to BbBa., \$$

CLR PARSING TABLE

STATES	ACTION			GOTO		
	a	b	$	S	A	B
0	R3	R~~2~~4		1	2	3
1			Accept			
2	S4					
3		S5				
4		R3			6	
5	R4					7
6		S8				
7	S9					
8			R1			
9			R2			

* The given Grammar is CLR(1) because there are no multiple entries in the parsing Table.

Scanned with CamScanner

ERROR RECOVERY IN LR PARSING

Error is detected by LR parser after consulting the parsing ACTION table and finds an error entry. GOTO part of parsing table can not be used for detecting errors. If there exist no valid entry in ACTION part of table for the portion of input scanned so far, then an error is announced by the LR parser

CLR(1) parser detect error earlier than SLR or LALR parser

There are two modes of error recovery in, LR parsing

1. **PANIC MODE ERROR RECOVERY :-**

In this technique stack is scanned till state 'S' with a GOTO on a particular non terminal 'A' is found.. Till a symbol 'a' is found that can legitimately follow A, zero or more input symbols are discarded. The parser then stacks the state GOTO(S, A) and normal parsing is continued

2. **PHRASE LEVEL ERROR RECOVERY :-**

In this technique each error entry in LR parsing table is examined and is decided based on the language usage, the most likely programming error that would give rise to error

The possible error routines are called for recovering the errors they are explained below

→ e1 : This routine is called from states 0, 2, 4 and 5 all of which the beginning of the operand either id or left parenthesis. Instead an operator + or * or the end of input was found.

ACTION : push an imaginary 'id' on to the stack i.e state 3

PRINT : ISSUE diagnostic "MISSING OPERAND".

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

ACTION : Remove the right parenthesis from the input

PRINT : ISSUE Diagnostic "UNBALANCED PARENTHESIS".

→ e3 : This Routine is called from state 1 or 6 when expecting an operator and an 'id' or right parenthesis is found

ACTION : push + onto the stack and use state 4

PRINT : ISSUE Diagnostic "MISSING OPERATOR".

→ e_4 : This Routine is called from state 6 when end of input is found while expecting operator or a right parenthesis

ACTION: push a Right parenthesis onto the Stack and use State 9.

PRINT: Issue Diagnostic "MISSING RIGHT PARENTHESIS".

Eg:— Consider the Grammar $E \to E+E \mid E*E \mid (E) \mid id$. The LR parsing table with error Routine is shown below.

LR(0) parsing Table

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	Accept	
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_3	R_3	R_3	R_3	R_3	R_3	

STACK	SYMBOLS	INPUT	ACTION
0		id +) $	S_3 [SHIFT 3]
0 3	id	+) $	Reduce $E \to id$ (R_4)
0 1	E	+) $	S_4 [SHIFT 4]
0 1 4	E +	) $	e_2: ISSUE UNBALANCED RIGHT PARENTHESIS
0 1 4	E +	$	e_1: PUSH 3 on to stack & ISSUE MISSING OPERAND
0 1 4 3	E + id	$	R_4 $E \to id$
0 1 4 7	E + E	$	R_1 $E \to E+E$
0 1	E	$	ACCEPT

DANGLING ELSE AMBIGUITY

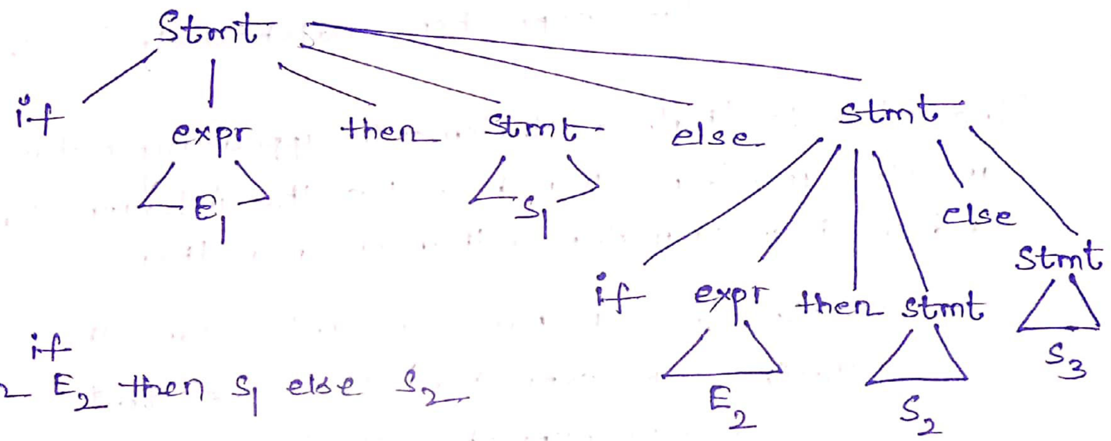
The given Grammar is ambiguous because there are more than one parse tree for the single statement which is generated from this Grammar i.e if E_1 then if E_2 then S_1 else S_2.

```
Stmt → if expr then Stmt
     | if expr then Stmt else Stmt
     | Other
```

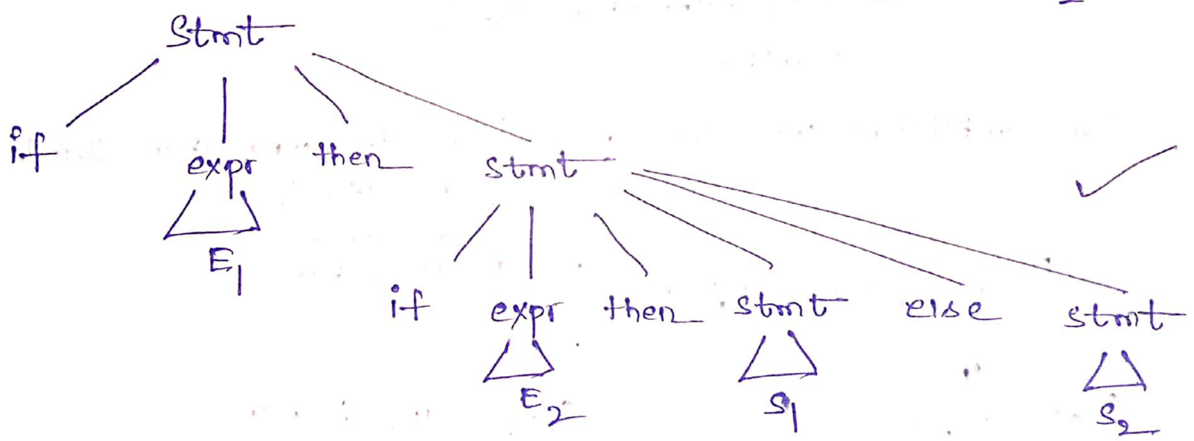
Other stands for any other statement

if E_1 then S_1 else if E_2 then S_2 else S_3

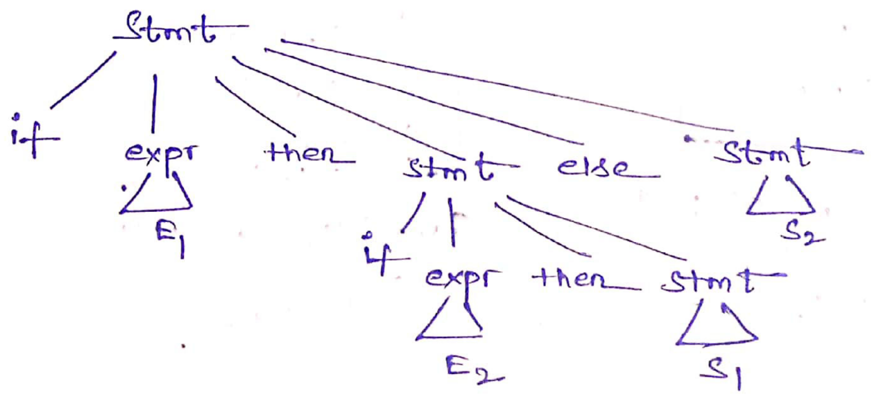
the parse tree for the above string is as follows



Eg: if E_1 then E_2 then S_1 else S_2



if E_1 then if E_2 then S_1 else S_2



In all programming languages with conditional statements of this form, the first parse tree in fig 2 is preferred.
The general rule is to match each else with the closest unmatched then.

The Grammar can be rewritten to remove the ambiguity

stmt → matchedstmt
 | openstmt
matchedstmt → if expr then matched stmt else matchedstmt
 | other
openstmt → if expr then stmt
 | if expr then matched stmt else openstmt.

The idea is that a statement appearing between a then and an else must be matched i.e the interior statement must not end with an unmatched then or open then

We construct SLR parsing table for the Grammar
stmt → if expr then stmt else stmt
 → if expr then stmt
 → other

for simplicity we consider i as if expr then ~~stmt~~.
S as stmt
e as else
a as other

$\boxed{S \rightarrow iSeS \mid iS \mid a}$

I_0

$S' \rightarrow .S$
$S \rightarrow .iSeS$
$S \rightarrow .iS$
$S \rightarrow .a$

GOTO (I_0, i) : I_2

$S \rightarrow i.SeS$
$S \rightarrow .iSeS$
$S \rightarrow .iS$
$S \rightarrow .a$
$S \rightarrow i.S$

GOTO (I_0, S) : I_1

$\boxed{S' \rightarrow S.}$

GOTO (I_0, a) : I_3

$\boxed{S \rightarrow a.}$

GOTO (I_2, S) : I_4

S → iS.eS

S → iS.

GOTO (I_2, i) : I_2

S → i.SeS

S → i.S

S → .iSeS

S → .iS

S → .a

GOTO (I_2, a) : I_3

GOTO (I_4, e) : I_5

S → iSe.S

S → .iSeS

S → .iS

S → .a

GOTO (I_5, S) : I_6

$\boxed{S \to iSeS.}$

~~S → iSeS~~

GOTO (I_5, i) : I_2

GOTO (I_5, a) : I_3

FOLLOW(S) : { $ }

$S \to i\underline{S}\,\overline{eS}$

$A \to \alpha B \beta$

FIRST (eS) = e

FOLLOW (S) : { e, $ }

S → iS

A . α B

FOLLOW (S) = FOLLOW (S) [same]

STATES	ACTION				GOTO
	e	i	a	$	S
0		S2	S3		1
1				Accept	
2		S2	S3		4
3	R3			R3	
4	S5/R2			R2	
5		S2	S3		6
6	R1	S2	S3	R1	

PARSING OF STRING <u>iaea</u>

STACK	SYMBOLS	INPUT	ACTION
1.) 0		iaea$	S2
2.) 0 2	i	aea$	S3
3.) 0 2 3	i a	ea$	R3 [S → a]
4.) 0 2 4	i S	ea$	R2 [S → iS]
5) 0 1	S	ea$	No entry in Parsing Table Error

TO SOLVE THIS PROBLEM : SHIFT IS GIVEN precedence over Reduce

At step 4 perform SHIFT OPERATION i.e S5

	STACK	SYMBOL	INPUT	ACTION
1)	0		iaea$	S_2
2)	0 2	i	aea$	S_3
3)	0 2 3	ia	ea$	R_3 $S \rightarrow a$
4)	0 2 4	iS	ea$	S_5
5)	0 2 4 5	iSe	a$	S_3
6)	0 2 4 5 3	iSea	$	R_3 $S \rightarrow a$
7)	0 2 4 5 6	iSeS	$	R_1 $S \rightarrow iSeS$
8)	0 1	S	$	ACCEPT

* COMPARE VARIOUS TYPES OF LR PARSERS.

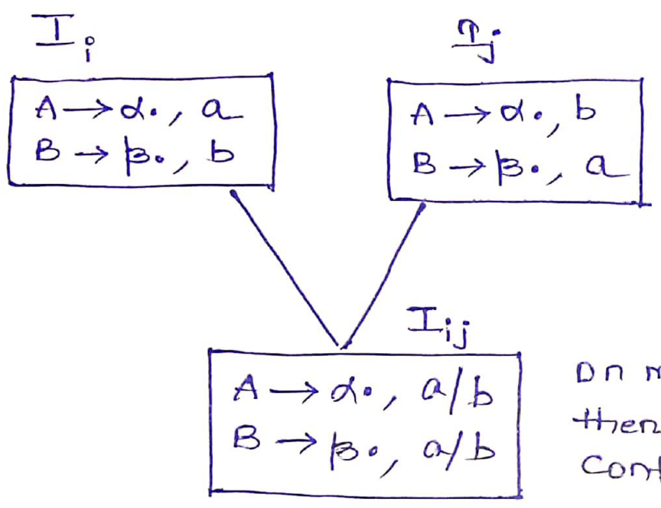
The bottom-up parsers differ in their size, capability and the class of grammars that they can handle.

S.NO.	SLR PARSER	LR PARSER	LALR PARSER
1.	SLR parser is easier to implement	LR parser is expensive in terms of time & space	LALR parser is efficient interms of space.
2.	parsing Table generated by this parser is smaller	It has Largest size parsing Table	LALR parsing Table is having same size that of a SLR parser
3.	It uses FOLLOW information for reduction	It uses the lookahead symbols.	It uses lookahead symbols
4.	It can handle only few classes of Grammar	It can handle large sub classes of CFG	It can handle a wider class of Grammars than SLR Grammars.
5.	It is least powerful	It is more powerful	It lies between SLR and ~~LALR~~ LR parser
6.	Before announcing an error, it may make several reductions. But it never shift an errorneous i/p symbol onto the stack.	It can detect a syntactic error as soon as possible	It behaves similar to SLR parser
7.	Grammar parsed by SLR parser is called SLR(1)	Grammar Parsed by CLR is called LR(1) Grammar	Grammar Parsed by LALR parser is called LALR(1) GRAMMAR

Points to REMEMBER

1. If a Grammar is not CLR(1) then it is not LALR(1). If Shift Reduce conflits arises in CLR parsing then it will be reflected in LALR parsing.

2. If a Grammar is CLR(1) then it is not necessarily LALR(1). During merging of states in LALR parsing sometimes Reduce/Reduce conflicts may arise.
Consider the example below



On merging above states I_i and I_j then it results in Reduce/Reduce Conflits.

3. power of LR PARSERS : $\boxed{\text{CLR} > \text{LALR} > \text{SLR}}$

Top-DOWN PARSING

In this approach a parse tree is constructed for the input string, starting from the root and creating the nodes of the parse tree in preorder. Top-Down parsing can be viewed as finding a Left Most derivation for an input string.

* At each step of a top-down parse, the key problem is that of determining the production to be applied for a Non terminal. Once an A-production is choosen, the rest of parsing process consists of matching the terminal symbols in the production body with the input string.

The general form of top-down parsing called recursive descent parsing, which may require backtracking to find the correct A-productions to be applied.

General recursive descent may require backtracking, i.e it may require repeated scans over the input.

Lets take an example for recursive descent parsing which needs backtracking

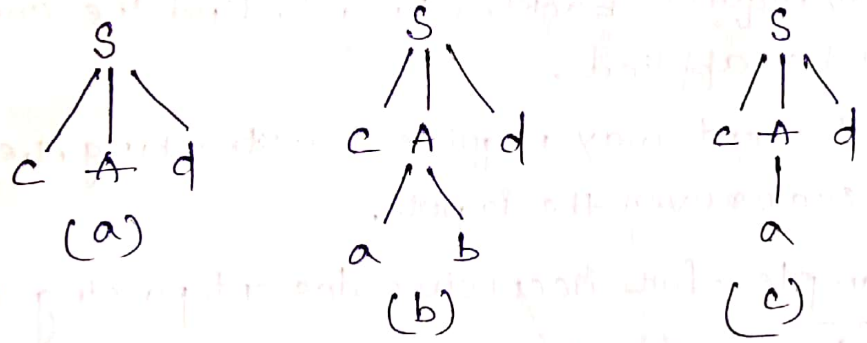
Example: $S \rightarrow cAd$
$A \rightarrow ab|a$

To construct a parse tree for the input string $w = cad$, begin with a tree consisting of a single node labeled S, and the i/p pointer pointing to c, the first symbol of w. S has only one production, so we use it to expand S and obtain the tree shown in fig (a) below. The leaf labeled c, matches the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A. Now we expand A using first alternative $A \rightarrow ab$ to obtain the tree of fig(b). We have a match for the second i/p symbol 'a' so we advance the i/p pointer to 'd', the third i/p symbol, and compare 'd' against the next leaf, labeled 'b'. Since 'b' does not match 'd' we report failure and go back to A and apply another production.

* In going back to A, we must reset the i/p pointer to position 2, the position it had when we first came to A, which means that procedure for A must store the input pointer in a local variable

* The second alternative for A produces the tree of Fig (c). The leaf 'a' matches the second symbol of ω and the leaf d matches the third symbol. Since we have produced a parse tree for ω, we halt and announces successful completion of parsing.

DRAWBACK: There are few Grammars that can cause a recursive descent parser to enter into an infinite Loop.
One of such Grammar is known as left Recursive Grammar



DIFFICULTIES WITH TOP DOWN PARSING

There are various difficulties associated with top-down parsing. They are:

1. Backtracking is the Major difficulty with top-down parsing. Choosing a wrong production for expansion neccessitates backtracking. Top-down parsing with backtracking involves exponential time complexity with respect to the length of the input
2. Left recursive grammars cannot be parsed by top-down parsers since they may create an infinite loop.
3. Top-down parsers cannot parse the ambiguous Grammar
4. Top-down parsers are slow and debugging is very difficult.

RECURSIVE DESCENT PARSING

A Top down parser that executes a set of recursive procedures to process the input without backtracking is called Recursive-Descent parser and parsing is called Recursive Descent parsing. The recursive procedures can be easy to write and fairly efficient if written in a language that implements the procedure call efficiently. There is a procedure for each non-terminal in the Grammar. We assume a global variable, lookhead, holding the current input token and a procedure match (Expected token) is the action recognizing the next token in the parsing process and advancing the input stream pointer, such that lookahead points to the next token to be parsed. match() is effectively a call to the lexical analyzer to get next token. For example input stream is a+b$ then

```
lookahead ==a
 match()
lookahead==+
 match()
Lookahead ==b
```

Example : Stmt → IDENTIFIER = expr ;

```
stmt ()
 {
   if(match(IDENTIFIER))
    {
      if(match (=))
       {
         if(expr)
          {
            if(match(;))
             {
               return(success);
             }
          }
       }
    }
   return(FAILURE);
}
```

Example: Consider the following Grammar

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

Write the algorithm of recursive descent parser

```
E()
{
   T();
   E'();
}
T()
{
   F();
   T'();
}
T'()
{
 if(lookhead==*)
   { match();
     F();
     T'();
   }
}
```

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

F()
{
  if(lookhead==id)
   {
     match();
   }
  else if(lookhead=='(')
   {
     match();
     E();
     if(lookhead==')')
      {
       match();
      }
     else ERROR
   }
  else ERROR
}
```

Eg: Implement Recursive Descent parser for the following Grammar

$S \to AB$
$A \to c \mid cB$
$B \to d$

Recursive-Descent Parsing

A Recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the Start symbol, which halts and announces success if its procedure body scans the entire input string.

Recursive-Descent Parsing

```
  void A( ) {
1)   Choose an A-production, A → X1X2..Xk ;
2)   for (i = 1 to k) {
3)        if ( Xi is a nonterminal )
4)           call procedure Xi( )
5)        else if (Xi equals the current input symbol a)
6)        advance the input to the next symbol ;
7)        else  /* an error has occurred */ ;
     }
  }
```

Fig: A typical procedure for a nonterminal in a top-down parser

General recursive-descent may require backtracking.
To allow backtracking, the above code needs to be modified first we cannot choose a unique A-production at line (1), so we must try each of several productions in some order. Then failure at Line (7) is not ultimate failure, but suggests only that we need to return to line (1) and try another A-production. Only if there are no more A-productions to try then we declare that an input error has been found. In order to retry another production we need to be able to reset the i/p pointer to where it was when we reached line (1). Thus a local variable is needed to store this input pointer for future use

PREDICTIVE PARSING

Predictive parsing is a special case of recursive descent parsing which do not need backtracking. These parsers can function as predictive parser only if the input Grammar is free from left recursion and left factoring.

FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW associated with a Grammar G. During top-down parsing FIRST and FOLLOW allow us to choose which production to apply based on the next input symbol.

FIRST SET : A first set of a Non-terminal A is the set of all the terminals that A can begin with.

Eg: C_Stmt $\rightarrow$ ID EQ_OP C_Expression SEMI_COLON

In above production C_stmt begins with ID for a legal input. Hence FIRST set of C_stmt is an ID.

TO COMPUTE FIRST(X) FOR Grammar SYMBOLS X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = $\{X\}$.
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_K$ is a production for some $K \geq 1$, then place a in first(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ... FIRST(Y_{i-1}); that is $Y_1 \ldots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots K$ then add ϵ to FIRST(X). Everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$ then we add FIRST(Y_2) and so on.
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

Example

$E \rightarrow E+T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$
} Eliminate left recursion from this Grammar

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

Now let us compute FIRST SET for this Grammar
consider production (1) $E \rightarrow TE'$ it is in form $X \rightarrow Y_1, Y_2 \ldots Y_K$

FIRST(E) = FIRST(T)
then calculate FIRST(T)
consider T's production i.e production (3)
FIRST(T) = FIRST(F)
then calculate FIRST(F)
consider F's production i.e production (5)
FIRST(F) = FIRST(( ) = { (}
= FIRST(id) = {id}
FIRST(F) = { (, id }
Hence FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
Then find FIRST(E') and FIRST(T')
$E' \rightarrow +TE' \mid \epsilon$
FIRST(E') = FIRST(+) ∪ FIRST(ε)
= {+, ε}

$T' \rightarrow *FT' \mid \epsilon$
FIRST(T') = FIRST(*) ∪ FIRST(ε)
= {*, ε}

THE FIRST FUNCTION FOR above Grammar
FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
FIRST(T') = {*, ε} , FIRST(E') = {+, ε}

FOLLOW SET :

A FOLLOW SET of a nonterminal A is the set of all the terminals that can follow A.

C_Stmt → ID EQ_OP C_Expression SEMI_COLON

By inspecting above production we can figure out that the C_Expression is definitely followed by a SEMI_COLON for a legal input. Hence

FOLLOW(C_Expression) = {SEMI_COLON}

* The FOLLOW set would never contain ϵ.

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

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B)

* FOLLOW(B) = FIRST(β) − ϵ (OR) $\underbrace{\text{FOLLOW(B) = FIRST}(\beta)}_{\text{RULE (2)}}$

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

* FOLLOW(B) = {FIRST(β) − ϵ} ∪ FOLLOW(A) [if $A \rightarrow \alpha B \beta$ & FIRST(β) contains ϵ]

* FOLLOW(B) = FOLLOW(A) [if $A \rightarrow \alpha B$] → Rule 4

EXAMPLE : Compute Follow SET

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

Let us compute FOLLOW(E), to compute this check the production which contains E on right side [consider it as B in Right side production]

FOLLOW(E) : $\underset{\alpha}{F \rightarrow (} \underset{B}{E} \underset{\beta}{)}$ $\Rightarrow$ calculate FIRST(β) i.e FIRST( ))

FIRST( )) = {) }. Since FIRST( )) doesn't contain ϵ apply rule 2. E is start symbol place \$ in FOLLOW(A) = { \$ }

FOLLOW(E) = FIRST( )) $\Rightarrow$ FOLLOW(E) = {), \$ }

FOLLOW(E') : $\underset{A}{E} \rightarrow \underset{\alpha\ B}{TE'}$ $\Rightarrow$ FOLLOW(E') = FOLLOW(E) = {), \$ }
Apply RULE (4)

FOLLOW(T) : E → TE' i.e FOLLOW(E') = FOLLOW(E) We are not getting FOLLOW(T) because T is α in this production so consider another production having T at right side & consider it as B

$E' \rightarrow \underset{\alpha}{+} \underset{B}{T} \underset{\beta}{E'} | \epsilon$

FIRST(E') is contains ϵ productions so Apply Rule 3.

FOLLOW(T) = FIRST(E') − ϵ ∪ FOLLOW(E')
= { +, ϵ } − ϵ ∪ {), \$ }
= { +,), \$ }

FOLLOW(T') = $\underset{A}{T} \rightarrow \underset{\alpha\ B}{FT'}$ $\Rightarrow$ FOLLOW(T') = FOLLOW(T)
$\Rightarrow$ FOLLOW(T') = { +,), \$ }

FOLLOW(F) = $T' \rightarrow \underset{\alpha}{*}\underset{B}{F}\underset{\beta}{T'}$ $\Rightarrow$ FIRST(T') contains ϵ so Apply Rule 3

FOLLOW(F) = { FIRST(T') − ϵ } ∪ FOLLOW(T')
= { *, +,), \$ }

After computing FIRST and FOLLOW FUNCTIONS construct predictive parsing table M[A, a], a two dimensional array, where A is a Non terminal and 'a' is a terminal or the symbol $, the input end Marker.

ALGORITHM: CONSTRUCTION OF A PREDICTIVE PARSING TABLE

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, $].

After performing the above if there is no production at all in M[A, a] then set M[A, a] to error

NON TERMINALS	INPUT SYMBOL					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Fig: parsing Table M

NONRECURSIVE PREDICTIVE PARSING

A nonrecursive predictive parser can be built by maintaining a stack explicitly. The parser uses Left most derivation. If ω is ~~as~~ the input that has been matched so far, then the stack holds a sequence of grammar Symbol α such that

$$S \overset{*}{\underset{lm}{\Longrightarrow}} \omega\alpha$$

The table driven parser has an input buffer, a stack containing a sequence of grammar symbols, a parsing table and an output stream. The input buffer contains the string to be parsed followed by the endmarker $. We use the same symbol to mark the bottom of the stack, which initially contains the Start symbol of the Grammar on top of $.

The parser is controlled by a program that considers X, the symbol on top of the stack, and a, the current input symbol. If X is a nonterminal, the parser chooses an X-production by consulting the entry M[X,a] of parsing table M. otherwise, it checks the match between the terminal X and current i/p symbol a.

```
    set x to the top stack symbol;
}
```

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

ERROR RECOVERY IN PREDICTIVE PARSING

The Error recovery of the parser is the ability to ignore the current error and continue with the parsing for the remainder of the input.
An error is detected during predictive parsing when the terminal on the top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and M[A, a] is error.
The error recovery schemes that are commonly used in predictive parsing are

1. PANIC MODE RECOVERY
2. PHRASE LEVEL RECOVERY

PANIC MODE RECOVERY

It is based on the principle that when an error is detected, the parser should skip the input symbol until it finds the synchronizin token in the input. Usually, the synchronizing tokens are more than one; hence a set called synchronizing set. The effectiveness of panic mode recovery depends on the choice of synchronizing set. some of the guidelines for constructing the synchronizing set are as follows

1. For a Non Terminal A, all the elements of Follow set of A can be added to synchronizing set of A.
2. If we add symbols in FIRST (A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if the Symbol in FIRST(A) appears in the input
3. If a terminal on the top of stack cannot be matched, then pop the terminal from the top of stack and issue a warning indicating that the terminal was inserted and continue parsing
4. If a Non-terminal can generate the empty string, then the production deriving ε. Can be used as default.

If the parser looks up entry M[A,a] and finds that ~~it~~ is blank, the i/p symbol is (a) is skipped.

If the entry is "synch" then nonterminal on the top of the stack is popped in attempt to resume parsing.

"synch" indicating synchronizing tokens obtain from the FOLLOW set of the NONterminal.

If a token on top of the stack does not match the i/p symbol, then we pop the token from the stack

NON TERMINAL	id	+	*	(	)	$
E	E→TE'			E→TE'	synch	synch
E'		E'→+TE'			E'→∈	E'→∈
T	T→FT'	synch		T→FT'	synch	synch
T'		T'→∈	T'→*FT'		T'→∈	T'→∈
F	F→id	synch	synch	F→(E)	synch	synch

On Erroneous input)id*+id

STACK	INPUT	REMARK
E$	)id*+id$	error, skip)
E$	id*+id$	id is in FIRST(E)
TE'$	id*+id$	
FT'E'$	id*+id$	
idT'E'$	id*+id$	
T'E'$	*+id$	
*FT'E'$	*+id$	
FT'E'$	+id$	ERROR, M[F,+] = synch F has been poped
T'E'$	+id$	
E'$	+id$	
+TE'$	+id$	
TE'$	id$	
FT'E'$	id$	
idT'E'$	id$	

Scanned by CamScanner

$T'E'\$$	$\$$
$E'\$$	$\$$
$\$$	$\$$

2 <u>PHRASE - LEVEL RECOVERY</u>

phrase-level recovery in predictive parser can be implemented by filling in blank entries in the predictive parsing table with pointer to error handling (utilities) routines.

Error Handling utility can do the following

1. The error-handling routines can insert, modify or delete any symbol in the input.

2. The routines can also pop elements from the stack.