## ALGORITHM FIRST(X)

Rule 1 → If $X \to a\alpha$ where 'a' is a terminal then FIRST(X) ← a.

If $X \to \epsilon$, then FIRST(X) ← $\epsilon$

Rule 2 → If $X \to Y_1 Y_2 Y_3 \dots Y_n$ and if $Y_1 Y_2 Y_3 \dots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$ then FIRST(X) ← non-$\epsilon$ symbols in FIRST($Y_i$).

Rule 3 → If $X \to Y_1 Y_2 Y_3 \dots Y_n$ and $Y_1 Y_2 Y_3 \dots Y_n \overset{*}{\Rightarrow} \epsilon$, then FIRST(X) ← $\epsilon$.

## ALGORITHM FOLLOW(S):

Rule 1 → FOLLOW(S) ← $ where S is the start symbol.

Rule 2 → If $A \to \alpha B \beta$ is a production and $\beta \neq \epsilon$ then FOLLOW(B) ← non-$\epsilon$ symbols in FIRST($\beta$).

Rule 3 → If $A \to \alpha B \beta$ is a production and $\beta \overset{*}{\Rightarrow} \epsilon$, then FOLLOW(B) ← FOLLOW(A).

## Constructing predictive parsing Table.

Now, using FIRST and FOLLOW sets, we can easily construct the predictive parsing table and the productions are entered into the table M[A,a] where

* M is a 2-dimensional array representing the predictive parsing table.
* A is a non-terminal w/c represent the row values.
* 'a' is a terminal or $ w/c is endmarker and represent

the column values.

ALGORITHM: Predictive-Parsing-Table (G, M)

Input: Grammar G.

Output: Predictive parsing table M.

Procedure: for each production $A \to \alpha$ of grammar G apply the following rules

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

2) If FIRST($\alpha$) contains $\epsilon$, for each symbol $b$ in FOLLOW($A$) add $A \to \alpha$ to $M[A, b]$.

Predictive parsing algorithm:

Input: The string w ending with $ (end of the input) and the parsing table.

Output: If $w \in L(G)$ i.e. if the input string is generated successfully from the parser, the parse tree using LMD is constructed. Otherwise the parser displays error messages.

Method: Initially the $ and S are placed on the stack and the i/p buffer contains i/p string w ending with $.

The initial configuration of the parser

| Stack | Input |
|---|---|
| $S | w$ |

⇒ Final configuration of parser, if parsing is successful.

| Stack | Input |
|---|---|
| $ | $ |

Algm: Let i/p pointer points to the first symbol of w

Let X = S be the symbol on top of the stack.

```
while (X ≠ $)                    // stack is not empty
    If (X == a)                  // stack symbol = input symbol.
        pop X from the stack.
        Advance the i/p ptr.
    else if X is a terminal
        Error()
    else if M[X,a] is blank.
        Error()
    else if M[X,a] = X → Y1 Y2 Y3 ... Yk
        output the production X → Y1 Y2 Y3 ... Yk.
        Remove X from the stack.
        Push Y1, Y2, Y3 ... Yk in reverse order
    end if
    Set X = top stack symbol.
end while.
```

• Consider the following grammar and the corresponding predictive parsing table. and show the sequence of moves made by the parser for the string id+id*id.

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

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

| | E | E' | T | T' | F |
|---|---|---|---|---|---|
| FOLLOW | {$, )} | {$, )} | {+, ), $} | {+, ), $} | {*, +, ), $} |

Parsing Table(a)

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

Sequence of moves : id+id*id.

M[E, id] = E → TE'



| Stack (x) | Input (a) | Output | Action |
|---|---|---|---|
| $ E | id + id * id $ | E → TE' | Remove E & push TE' in rev |
| $ E' T | id + id * id $ | T → FT' | Remove T & push FT' in revers |
| $ E' T' F | id + id * id $ | F → id | Remove F & push id |
| $ E' T' id | id + id * id $ | match (id) | Remove id increment i/p ptr |
| $ E' T' | + id * id $ | T' → ε | Remove T' from stack. |
| $ E' | + id * id $ | E' → + TE' | Remove E' & push + TE' in reverse. |
| $ E' T + | + id * id $ | match (+) | Remove + increment i/p ptr. |
| $ E' T | id * id $ | T → FT' | Remove T & push FT' in reverse |
| $ E' T' F | id * id $ | F → id | Remove F & push id. |
| $ E' T' id | id * id $ | match (id) | Remove id & increment i/p ptr |
| $ E' T' | * id $ | T' → * FT' | Remove T' & push * FT' in reverse |
| $ E' T' F * | * id $ | match (*) | Remove * increment i/p ptr |
| $ E' T' F | id $ | F → id | Remove F & push id. |
| $ E' T' id | id $ | match (id) | Remove id & increment i/p ptr. |
| $ E' T' | $ | T' → ε | Remove T' from stack. |
| $ E' | $ | E' → ε | Remove E' from stack |
| $ | $ | ACCEPT. | |

Since the stack contains $ & the i/p ptr points to $, the string id+id*id is parsed successfully

>

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

| | FIRST | FOLLOW |
|---|---|---|
| S | {a, b, c, d} | {$} |
| B | {a, ε} | {b} |
| C | {c, ε} | {d} |

| variables | a | b | c | d | $ |
|---|---|---|---|---|---|
| S | $S \to Bb/Cd$ | $S \to Bb/Cd$ | $S \to Bb/Cd$ | $S \to Bb/Cd$ | |
| B | $B \to aB$ | $B \to \epsilon$ | | | |
| C | | | $C \to cC$ | $C \to \epsilon$ | |

---

>

$S \to ACB \mid CbB \mid Ba$
$A \to da \mid BC$
$B \to g \mid \epsilon$
$C \to h \mid \epsilon$

| | FIRST | FOLLOW |
|---|---|---|
| S | {d, g, h, ε, b, a} | {$} |
| A | {d, g, h, ε} | {h, g, $} |
| B | {g, ε} | {$, a, h, g} |
| C | {h, ε} | {g, $, b, h} |

| variables | d | g | h | b | a | $ |
|---|---|---|---|---|---|---|
| S | $S \to ACB$ / $CbB$ / $Ba$ | $S \to ACB$ / $CbB$ / $Ba$ | $S \to ACB$ / $CbB$ / $Ba$ | $S \to ACB$ / $CbB$ / $Ba$ | $S \to ACB$ / $CbB$ / $Ba$ | $S \to \epsilon$ |
| A | $A \to da$ / $BC$ | $A \to da$ / $BC$ / $A \to \epsilon$ | $A \to da$ / $BC$ / $A \to \epsilon$ | | | $A \to \epsilon$ |
| B | | $B \to g$ / $B \to \epsilon$ | $B \to \epsilon$ | | $B \to \epsilon$ | $B \to \epsilon$ |
| C | | $C \to \epsilon$ | $C \to h$ / $C \to \epsilon$ | $C \to \epsilon$ | | $C \to \epsilon$ |

$S \to ABCd$.
$A \to \epsilon / +B$.
$B \to \epsilon / *B$.
$C \to \epsilon / \%B$.

| | FIRST | FOLLOW |
|---|---|---|
| S | {+, *, %, d} | {$} |
| A | {ε, +} | {*, %, d} |
| B | {ε, *} | {%, d} |
| C | {ε, %} | {d} |

Parsing Table

| Variables | + | * | % | $ | d |
|---|---|---|---|---|---|
| | | Terminals | | | |
| S | S→ABCd | S→ABCd | S→ABCd | | S→ABCd |
| A | A→+B | | | A→ε | |
| B | | B→*B | B→ε | | B→ε |
| C | | | C→%B | | C→ε |

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

| | S | A | B | C | D | E |
|---|---|---|---|---|---|---|
| FIRST | {a, b, c, e} | {a, ε} | {b, ε} | {c} | {d, ε} | {e, ε} |
| FOLLOW | {$} | {b, c} | {c} | {d, e, $} | {e, $} | {$} |

Parsing Table

| Variables | a | b | c | d | e | $ |
|---|---|---|---|---|---|---|
| S | S→ABCDE | S→ABCDE | S→ABCDE | | | |
| E | | | | | E→e | E→ε |
| A | A→a | A→ε | A→ε | | | |
| B | | B→b | B→ε | | | |
| C | | | C→c | | | |
| D | | | | D→d | D→ε | D→$ |

Assignment

| | FIRST | FOLLOW |
|---|---|---|
| 1) $S \to aABb$ | {a} | {$} |
| $A \to c/\epsilon$ | {c, ε} | {d, b} |
| $B \to d/\epsilon$ | {d, ε} | {b} |

Parsing Table

| Variable | a | c | d | $ | b |
|---|---|---|---|---|---|
| S | $S \to aABb$ | | | | |
| A | | $A \to c$ | $A \to \epsilon$ | | $A \to \epsilon$ |
| B | | | $B \to d$ | | $B \to \epsilon$ |

| | FIRST | FOLLOW |
|---|---|---|
| $S \to aBDh$ | {a} | {$} |
| $B \to cC$ | {c} | {g, f, h} |
| $C \to bC/\epsilon$ | {b, ε} | {g, f, h} |
| $D \to EF$ | {g, f, ε} | {h} |
| $E \to g/\epsilon$ | {g, ε} | {f, h} |
| $F \to f/\epsilon$ | {f, ε} | {h} |

Parsing Table

LL(1) ✗

| Variable | a | c | b | g | f | $ | h |
|---|---|---|---|---|---|---|---|
| S | $S \to aBDh$ | | | | | | |
| B | | $B \to cC$ | | | | | |
| C | | | $C \to bC$ | $C \to \epsilon$ | $C \to \epsilon$ | | $C \to \epsilon$ |
| D | | | | $D \to EF$ | $D \to EF$ | | $D \to \epsilon$ |
| E | | | | $E \to g$ | $E \to \epsilon$ | | $E \to \epsilon$ |
| F | | | | | $F \to f$ | | $F \to \epsilon$ |

$S \to (S) / \epsilon$ FIRST | FOLLOW.

Find sequence of moves for the string (( ))$.

Soln

| | FIRST | FOLLOW |
|---|---|---|
| $S \to (S) / \epsilon$ | {(, ε} | {), $} |

Parsing table

| Variable | ( | ) | $ |
|---|---|---|---|
| S | S→(S) | S→ε | S→ε |

| Stack | I/p | Prod | Action |
|---|---|---|---|
| $S | (( ))$ | S→(S) | Remove S, push (S) in reverse order |
| $)S( | (( ))$ | match(() | remove '(', adv i/p ptr. |
| $)S | ( ))$ | S→(S) | remove S, push (S) in reverse order. |
| $))S( | ( ))$ | match(() | remove '(' adv i/p ptr. |
| $))S | ) ))$ | S→ε | remove S, adv input ptr. |
| $)) | ) ))$ | match()) | remove ), adv i/p ptr. |
| $) | ) )$ | match()) | ——— " ——— |
| $ | $ | Accept | - |

$S \to AaAb / BbBa$

$A \to \epsilon$

$B \to \epsilon$

| | FIRST | FOLLOW |
|---|---|---|
| S | {a, b} | {$} |
| A | {ε} | {a, b} |
| B | {ε} | {b, a} |

LL(1) Grammar.

| | a | b | $ |
|---|---|---|---|
| S | S→AaAb | B→BbBa | |
| A | A→ε | A→ε | |
| B | B→ε | B→ε | |

10) $S \to aSbS/bSaS/\epsilon$ LL(1) ✗
Ambiguous.

11) $S \to aB/\epsilon$
$B \to bC/\epsilon$
$C \to cS/\epsilon$
LL(1). ✓

12) $S \to aABb$
$A \to c/\epsilon$
$B \to d/\epsilon$

3) $S \to A/a$
$A \to a$
LL(1) ✗
Ambiguous.

1) $S \to AB$.
$A \to a/\epsilon$.
$B \to b/\epsilon$.

2) $S \to aSA/\epsilon$
$A \to c/\epsilon$

3) $S \to A$
$A \to Bb/Cd$
$B \to aB/\epsilon$.
$C \to cC/\epsilon$.

4) $S \to aAa/\epsilon$.
$A \to abS/\epsilon$.

5) $S \to iEtSS'/a$.
$S' \to eS/\epsilon$.
$E \to b$.

Note: 1) No ambiguous grammar is LL(1)

2) If grammar has left recursion, then it

(6)

1) Since there are no multiple entries in the parsing table the given grammar is called LL(1) grammar.

2) If multiple entries are present in the parsing table the grammar is not LL(1).

3) The predictive parser accepts only the lang generated from LL(1) grammar.

## Bottom up parser.

Definition: - *) Bottom up parser reduces the given string of terminals w to the start symbol of the grammar.

*) It is an attempt to reduce the i/p string w to the start symbol S using RMD in reverse.

*) Bottom up parsing corresponds to the construction of a parse tree for an i/p string w from the leaves (bottom) and working towards the root (top) and hence the name bottom up parser.

*) Leaves represent the string of terminals and the root corresponds to the start symbol.

⇒ The process of obtaining the start symbol (root or top) from string of terminals (leaves or string of terminals) is called Bottom up parser:

Q: Consider the following grammar.

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

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

Show the tree snapshots that are constructed for the string id*id using bottom up parser.

N: In bottom up parser, we start from string of terminals and move upwards towards the root and get the start symbol. The tree snapshots that are

The RMD to get the string id * id is shown below. (7)

$E \Rightarrow T$ using $E \to T$

$\Rightarrow T * F$ using $T \to T * F$

$\Rightarrow T * id$ using $F \to id$

$\Rightarrow F * id$ using $T \to F$

$\Rightarrow id * id$ using $F \to id$

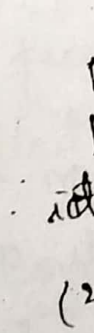
Now if we start from id*id and moving backwards by applying RMD. in reverse, we get the start symbols. and various sentential forms that we get while moving backwards are

id*id, F*id, T*id, T*F, T, E.

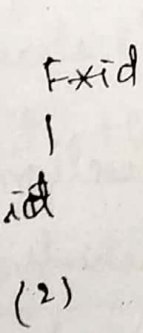
Thus, using series of reductions we are abone to get the start symbol and the tree snapshots corresponding to the above sentential forms are shown below:

id*id

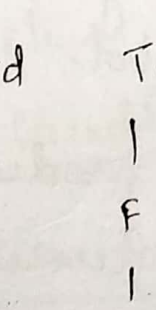
(1)



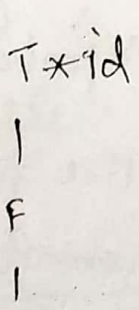

(2)



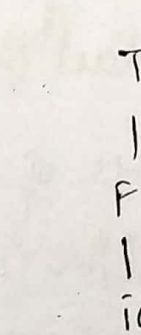

(3)




(4)

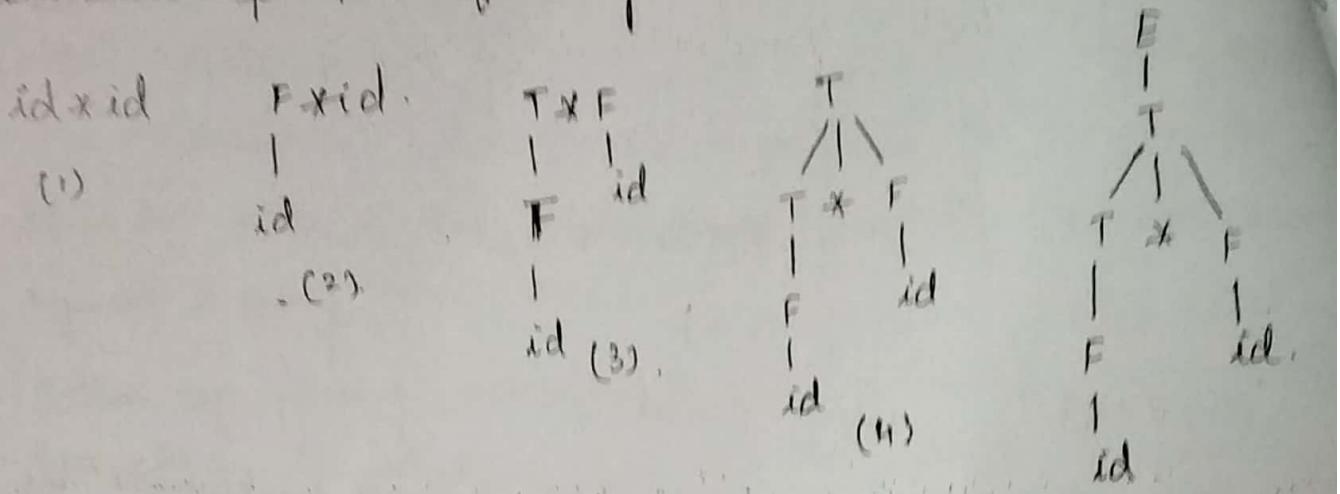



(5)




using series of reductions as shown above, we are able to get the start symbols and we say parsing is successful.

conducted for the ip string id * id are shown below:




Observe from above snapshots that we have started from the string of terminals and obtain the start symbol S hence we say parsing is successful. Thus, we can think of bottom up parser is the one that reduces the string of terminals w to the start symbol S of the given grammar.

## Reductions

Definition: During Bottom up parsing, the given string of terminals are reduced to the start symbol.

"The process of searching for RHS of the production in a string consisting of terminals and/or non-terminals and replacing it with corresponding LHS of the production is called reduction"

For example: consider the grammar.

$E \to E+T \mid T$ $\quad$ $F \to (E) \mid id$

$T \to T*F \mid F$

The RMD to get the string id*id is shown below: (8)

$E \Rightarrow T$
$\Rightarrow T*F$ using $E \to T$
$\Rightarrow T*id$ using $T \to T*F$
$\Rightarrow F*id$ $F \to id$
$\Rightarrow id*id$ using $T \to F$, $F \to id$

Note:
Main points to be considered during reduction process are
1) when to reduce and
2) what production to use for reducing as the parsing proceeds.

## Handle pruning

Defn: A substring that matches the RHS of a production (i.e that matches the body of a production) and replacing it with equivalent non-terminal on LHS of the production to get one step in RMD in reverse is called **Handle**.

Eg: Consider foll. grammar

$E \to E*T \mid T$
$T \to T*F \mid F$
$F \to id \mid (E)$

Derivation: RMD.

$E \Rightarrow T$ using $E \to T$
$\Rightarrow T*F$ using $T \to T*F$
$\Rightarrow T*id$ using $F \to id$
$\Rightarrow F*id$ using $T \to F$
$\Rightarrow id*id$ using $F \to id$

| Right sentential form in reverse | Handle | Reduction Production |
|---|---|---|
| id * id | id | F → id |
| F * id | F | T → F |
| T * id | id | F → id |
| T * F | T * F | T → T * F |
| T | T | E → T |
| E | - | E |

Note: The strings of characters underlined are the handles.

Example(1) for handle pruning.

Consider grammar $S \to 0S1 / 01$, indicates the handles in the following right sentential form: 00001111

| Right sentential form in reverse | Handle | Reduction Production |
|---|---|---|
| 00001111 | 01 | S → 01 |
| 000S111 | S → 0S1 | S → 0S1 |
| 00S11 | S → 0S1 | S → 0S1 |
| 0S1 | S → 0S1 | S → 0S1 |
| S | | |

Note: The strings of characters that are underlined are the handles. The handles are searched and are replaced with appropriate LHS of the production to get the start symbol.

Example 2:-

Grammar: "$S \rightarrow SS+ \mid SS* \mid a$" indicate the handles in the following right sentential form "SS+a*a+"

| Right Sentential form in reverse | Handle | Reduction production |
|---|---|---|
| Ss+a*a+ | SS+ | S → SS+ |
| Sa*a+ | a | S → a |
| SS*a+ | SS* | S → SS* |
| Sa+ | a | S → a |
| SS+ | SS+ | S → SS+ |
| S. | | |

Note: the strings of characters that are underlined are the handles. The handles are searched and replaced with appropriate LHS. of the prodn. to get the start symbol.

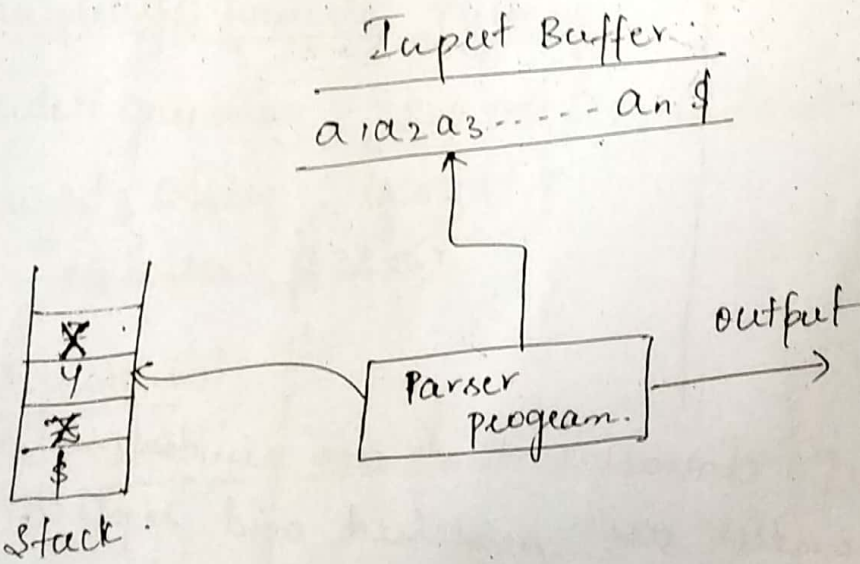
## Shift Reduce Parser

Shift reduce parser is an efficient way of implementing bottom up parser using explicit stack. The parser contains grammar symbols and i/p buffer holds the string to be parsed.

In this parser, we start from string of terminals and get the start symbol.

During parsing, always the handle will appear on top of the stack just before it is identified as the handle.

### Block diagram of various parts of - shift reduce parser.




Predictive parser has 4 components only:

1) I/p
2) Stack
3) parser program
4) o/p

Input → contains the string to be parsed and the i/p string ends with $.

$ indicates end of the i/p.

Stack → contains sequence of grammar symbols and $ is placed initially on top of the stack indicating stack is empty. (10)

3) Parser → ~~contains sequence of grammar symbols and~~

It is a program which shifts the i/p symbols on the stack or reduce the handle on the stack to appropriate LHS of the production.

Initial configuration of the parser:

| Stack | Input |
|---|---|
| $ | w$ |

Final configuration of the parser.

| Stack | Input |
|---|---|
| $ | $ |

In the final configuration the parser halts and announces successful completion of parsing.

Output: As o/p, the productions, that are used, are used to reduce are displayed till we get the start symbol on the stack.

Four possible actions of the shift-reduce parser are sta

1) Shift

Here, the next i/p symbol is shifted on to the top of stack.

2) Reduce.

By knowing the appropriate handle on the stack, the parser reduces this to the LHS of the appropriate action.

3) Accept:

The parser announces successful completion of parsing

4) Error

The parser discovers an error and the appropriate error message is displayed on the screen with the help of error handler and calls an error recovery routine.

Eg:

Show the sequence of moves made by the shift-reduce parser for the string id * id using the following grammar:

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

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

Soln: The sequence of moves made by the shift reduce parser for the string id*id is shown:

| Stack | Input | Action |
|---|---|---|
| $ | id * id $ | shift |
| $ id | * id $ | Reduce F → id |
| $ F | * id $ | Reduce T → F |
| $ T | * id $ | shift * [Instead of reducing E → T] |
| $ T * | id $ | shift id |
| $ T * id | $ | Reduce F → id |
| $ T * F | $ | Reduce T → T * F |
| $ T | $ | Reduce E → T |
| $ E | $ | ACCEPT |

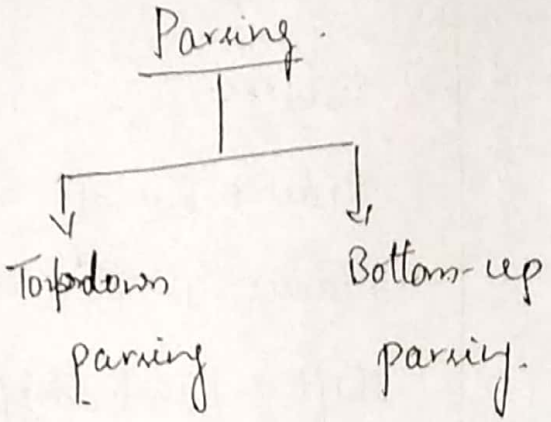
Since stack contains the start symbol and i/p is empty we say parsing is successful.

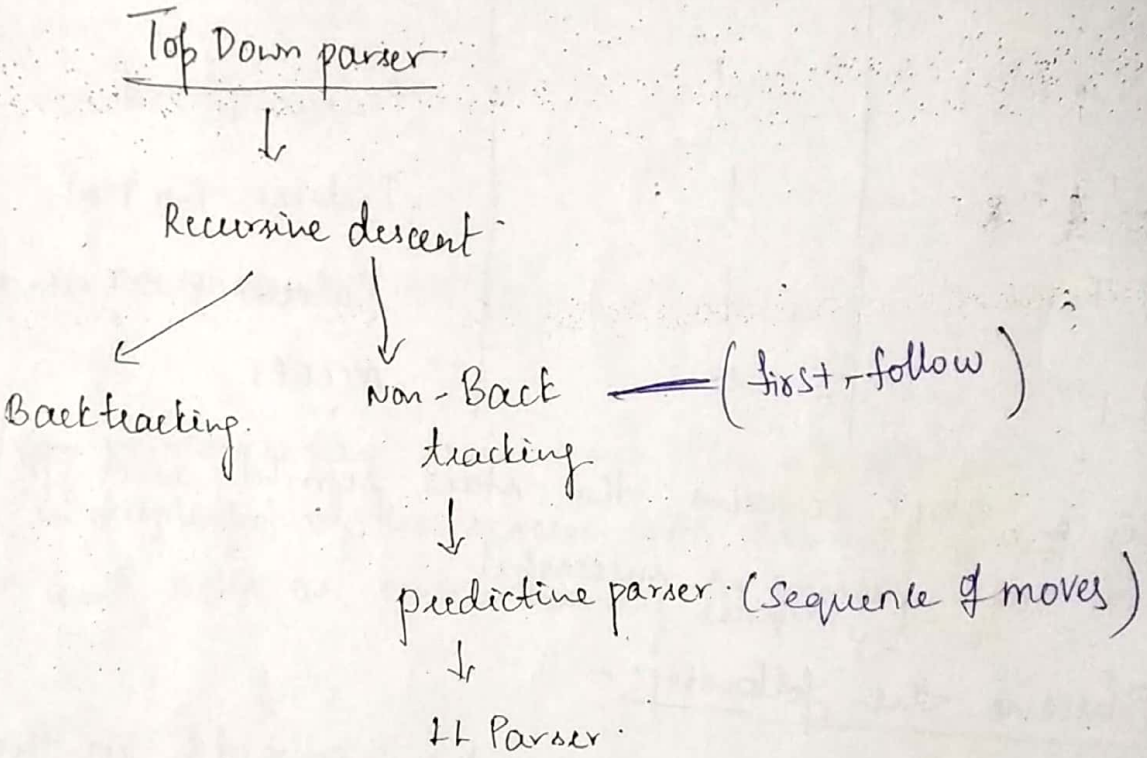
Observe the following:-

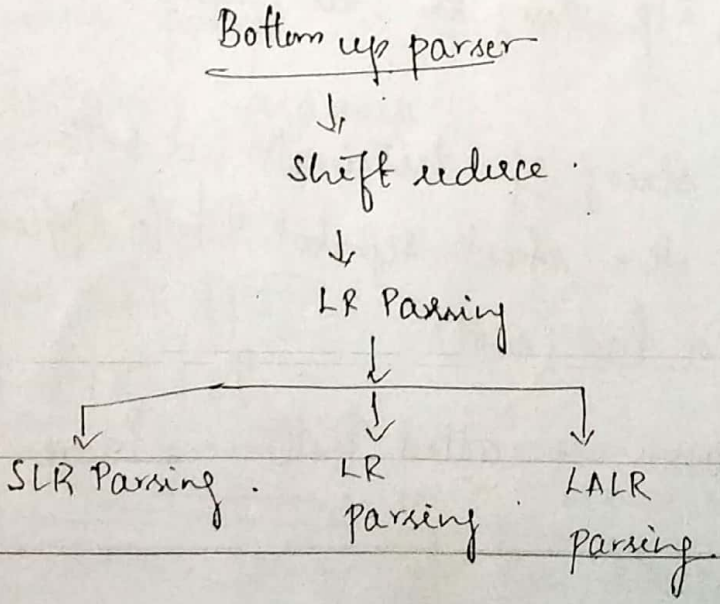
1) parser has started from the string id * id $ in the i/p, and it has reduced the i/p string to start symbol E on the stack.

2) Thus, starting from the string of terminals i.e. bottom (leaves) it has obtained the start symbol S which appears at the top of the parse tree (root).

3) So, the shift reduce parser is called Bottom up Parser.



## Conflicts during shift reduce parsing.

The shift reduce parser cannot be used for all CFG. we may get some conflicts during parsing.

The 2 types of conflicts that arise during shift reduce parsing are 1) shift-reduce conflicts
2) reduce-reduce conflicts

### shift-reduce conflicts

During parsing, by knowing the contents of the stack and the next i/p symbol, if the parser cannot decide whether to shift the i/p, symbol on to the stack or to reduce the handle on top of the stack, it results in shift-reduce conflict

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

Sequence of moves for the string id * id.

| Stack | Input | Action. |
|---|---|---|
| $ | id*id$ | Shift id. |
| $id | *id | Reduce F→id. |
| $F | *id. | |

The parser has reached a state where it cannot decide whether to shift i/p symbol * on the stack, or to reduce

using the production T→F.

## Reduce-reduce conflicts

During parsing, if shift reduce parser finds a handle on top of the stack, the parser has to reduce the handle to LHS of prodn, But, if 2 or more prodns have same handle on RHS of prodn, the parser cannot decide w/c prodn has to be selected for reducing.

Eg: E→E+T|T  String: id*id
T→T*F|F
F→(E)|id

Sequence of moves made by the parser for the string id*id

| Stack | Input | Action |
|---|---|---|
| $ | id*id$ | Shift id. |
| $id | *id$ | Reduce F→id. |
| $F | *id$ | ~~Shift *~~ Reduce F→T |
| $T | *id$ | Shift * |
| $T* | id$ | Shift id. |
| $T*id | $ | Reduce F→id. |
| $T*F | $ | ~~Reduce~~ |

The parser has reached state, where it cannot decide whether to reduce handle F to T using T→F (or) reduce handle T*F to T using T→T*F.

END

# Parsing Techniques

## Different Types of parsers.

There are 3 general types of parsers.

1) A UNIVERSAL PARSER.

- uses the algm like cocke-younger-kasami algm & Earley's algm.
- But these methods are too inefficient

2) TOP-DOWN PARSERS.

- Recursive descent parser with backtracking.
- Recursive descent parser with no backtracking. (Predictive parser)

3) BOTTOM UP PARSER.

- (Simple LR). SLR.
- LALR (Look Ahead LR)
- Canonical LR.

## Top down parsing.

Let us see "what is top down parser?"

The process of constructing a parse tree for the string of tokens (obtained from the lexical analyzer) from top i.e.

starting from the root node and creating the nodes of the parse tree in preorder in DFS (depth first search) manner is called top down parsing technique.

* The top-down parsing can be viewed as an attempt to find a left most derivation for an input string and constructing the parse tree for that derivation.

  The parser uses this approach is called top down parser.

* Since the parsing starts from top (i.e root) down to the leaves it is called top down parser.

Eg: String: id + id * id

Grammar: $E \to E+E$

$E \to E*E$

$E \to (E)$

$E \to id$

Consider: $E \to E*E$ (LMD applying)

$\to id+E$

$\to id+E*E$

$\to id+id*E$

$\to id+id*id$

$E \to id$

$E \to E*E$

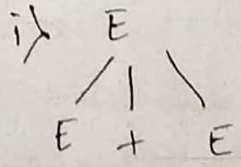
$E \to id$

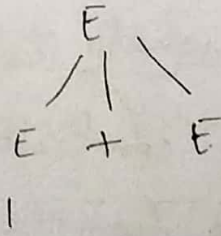
$E \to id$

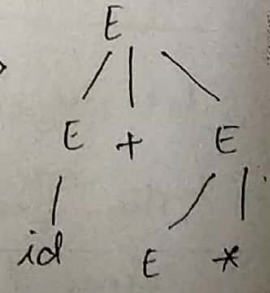
} productions applied.

LMD Parse tree

E root

i) 

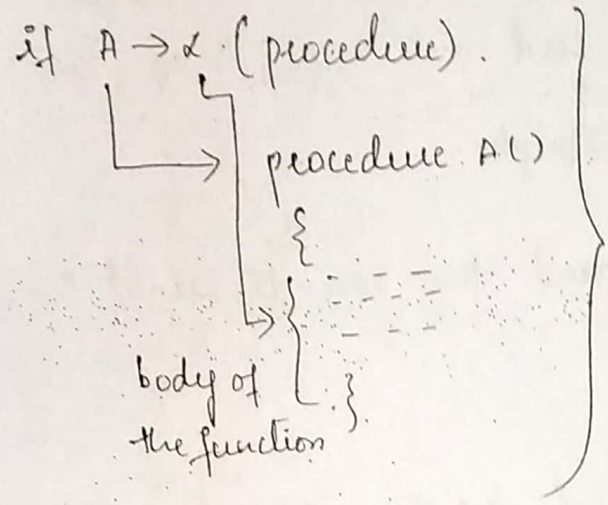

ii) 


iii) 

## Recursive descent Parser.

A recursive descent parser is a top down parser in which parse tree is constructed from the top starting from root node and selecting the productions from left to right. (if 2 or more alternate productions exists).

Eg:




General procedure for a recursive-descent parsing that uses top down parser is shown below.

Algm for recursive descent parser (backtracking is not supported)

```
procedure A()                    // A → X1 X2 --- Xk
{
    for i=1 to k do
        if (Xi is a non-terminal)
            call procedure Xi();
        else if (Xi is same as current i/p symbol a)
            advance the i/p to the next symbol.
        else error();
    endfor
```

Eg:- Show the steps involved in recursive descent parser without backtracking for the i/p string cad for the foll. grammar.

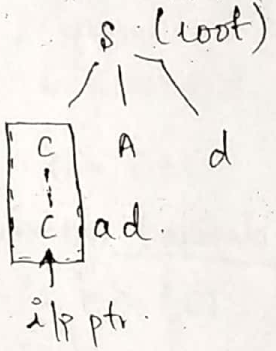
S → cAd

A → ab | a.

Soln:

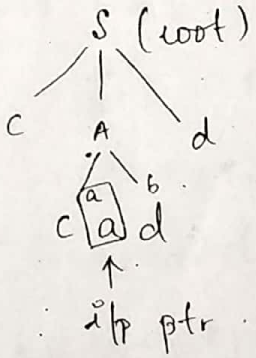
| Grammar | String to be parsed | Parse tree |
|---|---|---|
| S → c A d | cad | S (root) |
| A → ab \| a. | ↑ i/p ptr. | |

Step 1: S → cAd to expand the non-terminal S.



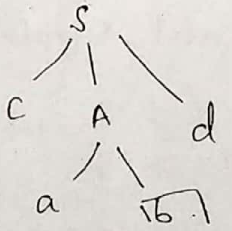

[match (c) & increment i/p ptr]

Step 2: Next node A to be expanded & i/p ptr pts to a.




[match (a) & increment i/p ptr]

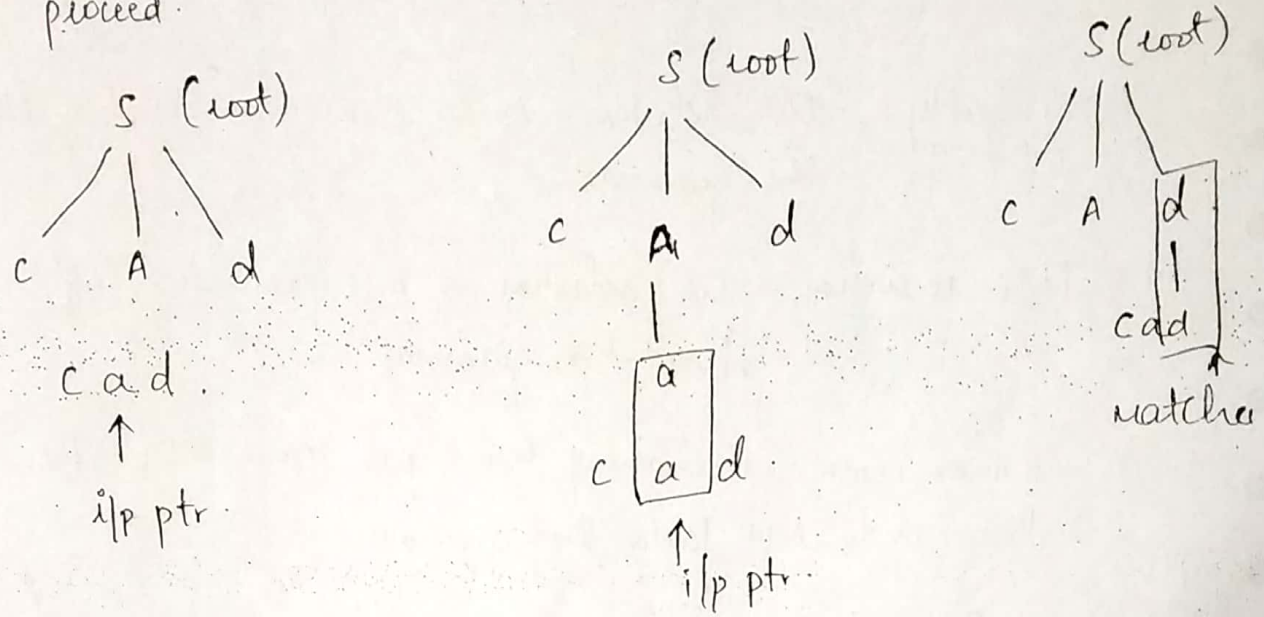
Step 3:




[b does not match with d]

Step 4: Observe that by selecting A→ab the i/p string is not matched. So we have to reset the pointer to i/p symbol a. This is done using backtracking.

After backtracking Expand A with A→a and then proceed.




finally we halt and announce successful parsing (completion)

What is the need of backtracking in recursive descent parser.

The backtracking is needed for the following reasons.

1) During parsing, the productions are applied one by one. But, if two or more alternative prodns are there, they are applied in order from left to right one at a time.

2) When a particular prodn applied fails to expand the non-terminal properly, we have to apply the alternate prodn. Before trying alternate prodn, it is necessary undo the activities done using the current prodn. This is possibly only using backtracking.

Note: The recursive descent parsers with backtracking are not frequently used. So, we just concentrate on how they work with example.

For what type of grammars recursive descent parser cannot be constructed.

1) Ambiguity - The solution is to eliminate ambiguity from the grammar.

2) Left recursion: - The solution is to eliminate left recursion from the grammar.

3) Two or more alternatives having a common prefix, The solution is to left factor the grammar.

Left Factoring: -

What is left factoring? what is the need for left factoring?

- A grammar in which two or more productions from a non terminal A do not have a common prefix of symbols on the right hand side of the A-productions is called left-factored grammar.
- The left-factored grammar is suitable for top down parser such as recursive descent parser with or without backtrack

Eg1: The grammar to generate string consisting of atleast one 'a' followed by atleast one 'b'.

$S \rightarrow aAbB$
$A \rightarrow aA \mid \varepsilon$
$B \rightarrow bB \mid \varepsilon$

} Left factored grammar.

→ S- production has only one production
It cannot have common prefix on the RHS of production
→ The 2nd A- productions do not have any common prefix on the RHS.
→ finally 3rd B- productions do not have any common prefix on the RHS.

Eg: 2) $S \rightarrow AB$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid b$

} Non-left factored grammar.

S→ production has only one production
A→ common prefix "a" on RHS
B→ —"— "b" on RHS

Note: If 2 or more productions starting from same non terminal have a common prefix the grammar is not left factored

## Use of Left factoring.

→ Left factoring is used for top down parser such as recursive descent parser with backtracking or predictive parser. which is also recursive descent parser without backtracking.

## Algorithm for left factoring:

Input: Grammar G.

Output: An equivalent left-factored grammar.

Method:

1) For each non-terminal A, find the longest prefix $\alpha$ which is common to two or more of its alternatives.

2) If there is a production of the form:

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \cdots \mid \alpha\beta_n \mid \gamma$$

where $\gamma \to$ do not start with $\alpha$,

then the above A-production can be written as.

$$A \to \alpha A' \mid \gamma$$

$$A' \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \cdots \mid \beta_n$$

Here $A' \to$ is a new non-terminal.

3) Repeatedly apply the transformation in step 2 as long as two alternatives for a non-terminal have a common prefix

4) Return the final grammar which is left factored.

## Problems

1) Do the left-factoring for the following grammar.

$$S \to iCtS \mid ictses \mid a$$

$$C \to b$$

Soln: Given productions.

$A \to \alpha\beta_1 | \alpha\beta_2 | \dots \alpha\beta_n | \gamma$

Left-factored productions.

$A \to \alpha A' | \gamma$

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

1) $S \to iCtSe | iCtSeS | a$

$S \to iCtSS' | a$

$S' \to e | eS$

$C \to b$

2) $C \to b$

final grammar obtained after doing left-factoring is

$S \to iCtSS' | a$

$S' \to e | eS$

$C \to b$

2) $S \to$ if E then S else S / if E then S

Soln: Given productions.

$A \to \alpha\beta_1 | \alpha\beta_2 | \dots \alpha\beta_n | \gamma$

$A \to \alpha A' | \gamma$

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

$S \to$ if E then S else S | if E then S

$S \to$ if E then S S' | $\epsilon$

$S' \to$ else S | $\epsilon$

} final grammar

3) $A \to aAB | aA$

$B \to bB | b$

4) $A \to aAB | aA | a$

5) $S \to iEtS | iEtSeS | a$

$E \to b$

## Left recursion

**Definition:** - A grammar G is said to be left recursive if has non-terminal A such that there is a derivation of the

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

Where $\alpha$ is string of terminals / non-terminals.

→ Whenever the first symbol in a partial derivation is same as symbol from which this partial derivation is obtained, then grammar is said to be left-recursive grammar.

→ A grammar may have 1) immediate left recursion
2) indirect left recursion.

## Immediate LR:

A grammar G is said to have immediate left recursion if it has a production of the form:

$$A \to A\alpha$$

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

Consider 1st 2-productions: $E \to E+T$

$T \to T*F$

The first symbol on the RHS of the production is same as the symbol on the LHS of the production.

Given grammar has immediate left recursion in 2 productions.

## Indirect left recursion:-

A left recursion involving derivations of two or more steps so that the first symbol on the RHS of the partial derivation is same as the symbol from which the derivation started is called indirect left recursion.

Eg: $E \to T$
$T \to F$
$F \to E+T \mid id$.

consider: $E \Rightarrow T$

$E \Rightarrow F$ | $T \to F$

$E \Rightarrow E+T$ | $F \to E+T$

Given grammar has indirect left recursion.

## Algorithm for left recursion:

Input: Grammar G.

output: Grammar without left-recursion.

It may have $\varepsilon$-productions

Arrange the non-terminals in the order $A_1, A_2, A_3, \ldots A_n$

for $i=1$ to $n$ do

for $j=1$ to $i-1$ do

let $A_j \to \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots \beta_k$

Replace $A_i \to A_j\alpha$ by $A_i \to \beta_1\alpha \mid \beta_2\alpha \mid \beta_3\alpha \mid \ldots \mid \beta_k\alpha$

end for

Eliminate immediate left recursion among $A_i$ productions

## Problems with top down parser

The various problems associated with top down parser are:

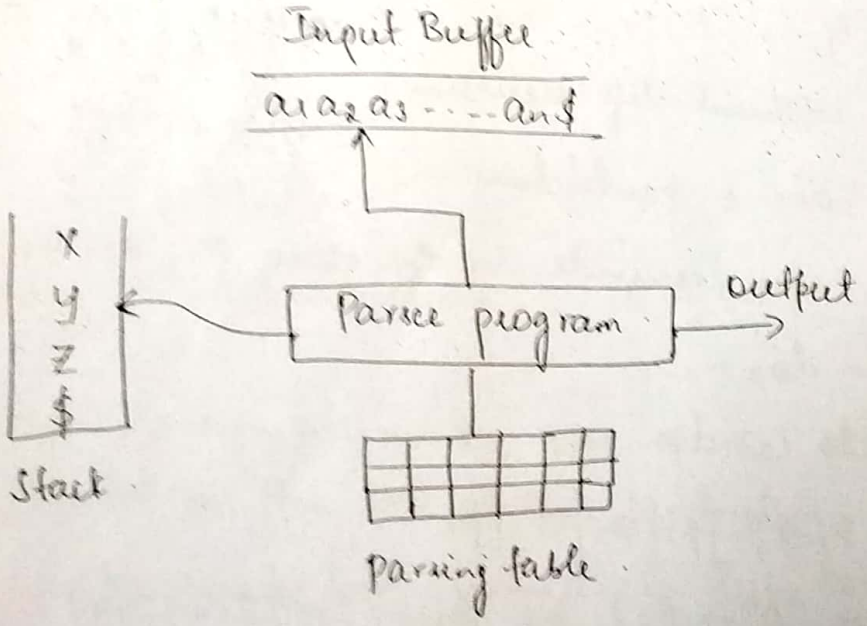
- Ambiguity in the grammar.
- left recursion.
- Non-left factored grammar.
- Backtracking.

## Recursive descent parser with no backtracking (Predictive Parse

→ Predictive parser is a top down parser. It is an efficient way of implementing a recursive descent parser by maintaining a stack explicitly rather than implicitly via recursive calls.

→ predictive parser can correctly guess or predict w/c production to use if two or more alternative productions are there.

## Various components of predictive parser




**Components**

1) Input
2) Stack
3) Parsing Table
4) Parsing Program
5) output.

Input → The i/p buffer contains the string to be parsed and the input string ends with $.
$ indicates end of i/p.

Stack → It contains sequence of symbols and
$ → placed initially on top of stack.
When $ is on top of stack, it indicates stack is empty.

Parsing table → It is a two dimensional array M[A,a]
A → non terminal [A represents row index]
a → terminal or $ [a represents column index]

M[A,a] → may contain production / blank entry.

Parser → It is a program which takes different actions based on X which is the symbol on top of the stack and the current i/p symbol a

Output → productions, that are used are displayed using which the parse tree can be constructed.