Module - 3 [Syntax Analysis]

Syntax Analysis or Parsing is the second phase, i.e., after lexical Analysis. It checks the syntactical structure of the given input, i.e, whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. If not, the error is reported by the syntax analyzer.

The pushdown automata (PDA) is used to design the syntax analysis phase.

The Role of the Parser

In our compiler model, the parser obtains a strings of tokens from the lexical analyzer, and verifies that the string of token is a valid sequence, i.e, ascertains whether its structure is syntactically correct. The whole process is known as Parsing, i.e, determining if a string of tokens can be generated by the grammar.

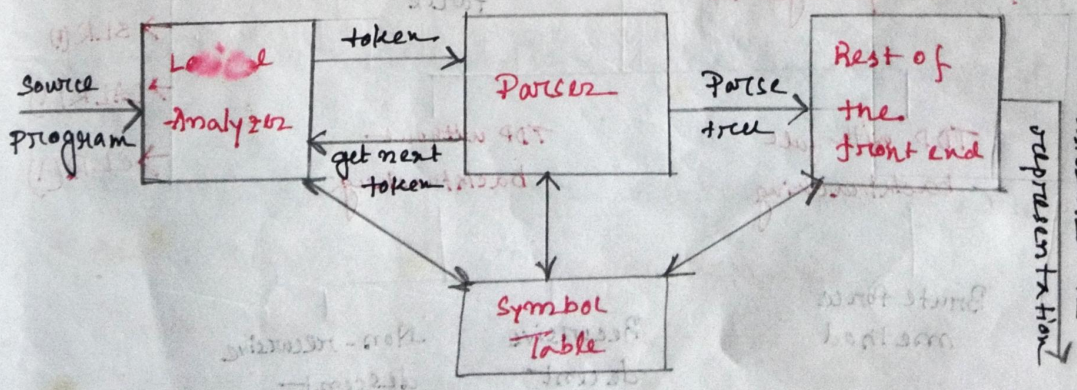


fig : Position of parser in the compiler model

other tasks of parser

→ The second task of parser is to report syntactic error, together with the line number.

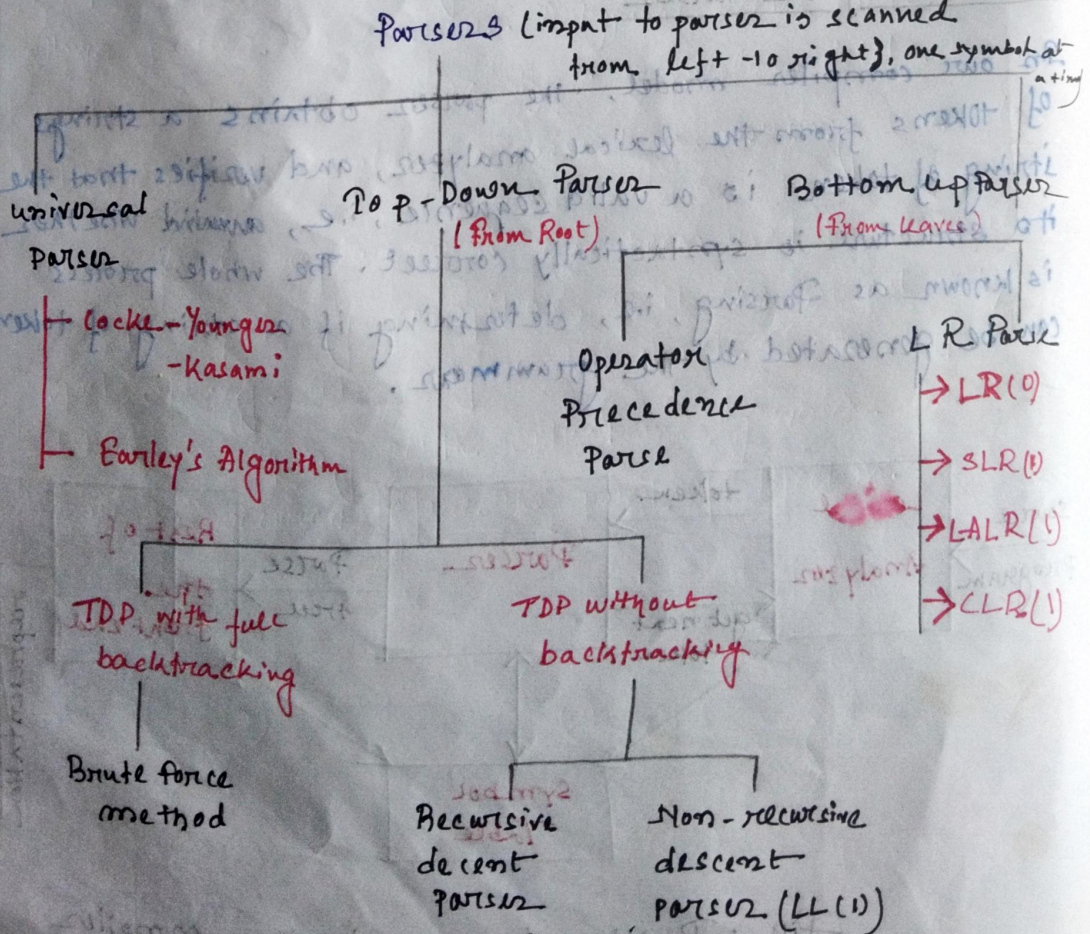
→ To recover from commonly occuring errors to continue processing the remainder of the program.

→ The parser constructs a parse tree and passes it to the rest of the compiler for futher processing. That means output of the syntax analyzer is a parse tree.

→ Type-checking also performed during this phase.

There are three general types of parser →

1) universal
2) Top-down
3) Bottom-up



Syntax Error Handling

Syntactic errors include misplaced semicolons or extra or missing braces, that is, "{" or "}". As another example, in C or Java, the appearance of a case statement without an enclosing switch is a syntactic error.

The precision of parsing methods allows syntactic errors to be detected very efficiently. Several parsing methods, such as the LL and LR methods, detect an error as soon as possible, that is, when the stream of tokens from the lexical analyzer cannot be parsed further according to the grammar for the language. More precisely, they have the viable-prefix property, meaning that they detect that an error has occurred as soon as they see a prefix of the input that cannot be completed to form a string in the language.

Error-Recovery Strategies

Handling syntactic errors has certain goals:

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

Here are the recovery strategies used by parsers:

- Panic-mode
- Phrase-level
- error-production
- global-correction

Panic Mode Recovery with this method, on discovering an error, the parser discards input symbols one at a time until one of a designated set of synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or }. while panic-mode correction often skips a considerable amount of input without checking it for additional errors, it has advantage of simplicity, and unlike some methods, it is guranteed not to go into an infinite loop.

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

Error Productions A parser constructed from a grammar augmented by the error productions detects the anticipated errors when an error production is used during parsing. The parser can then generate appropriate error diagnostics about the erroneous construct that has been recognized in the input.

Global Corrections The aim is to make as few changes as possible while converting an incorrect input string to a valid string. There are algorithms for choosing a minimal sequence of changes to obtain a globally least-cost correction. Given an incorrect input string x and grammar G, these algorithms will find a parse tree for a related string y, such that the number of insertions, deletions, and changes of tokens required to transform x into y is as small as possible. Unfortunately, these methods are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.

A Grammar for Arithmetic Expressions

Conditional statement

cstmt → if expr then stmt else stmt

NT T NT T NT T NT

op → + | - | * | / | ↑

d → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

expr: d op d, -d, (d),

expr op expr, -expr, (expr)

{expr → e}

grammar

e → e op e | (e) | -e | d

d → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

op → + | - | * | / | ↑

example ① -5

e → -e
→ -d
→ -5

② 7+5

e → e op e
→ d op d
→ 7+5

③ -(7+5)

e → -e
→ -(e)
→ -(e op e)
→ -(d op d)
→ -(7+5)

④ 7+5 - 9*2

e → e op e
→ e op e op e
→ e op e op e op e
→ d op d op d op d
→ 7+5-9*2

Associativity of operators

When an operand has operators on both its sides (left and right) then we need rules to decide with which operator we will associate the operands.

Left Associative → 1 + 2 + 3

(1 + 2) + 3

+, −, *, / → Left Associativity

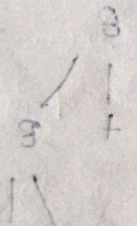
4 * 5 * 6

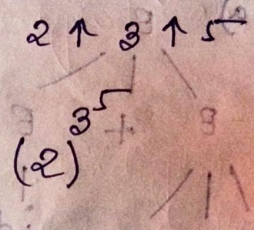
Right Associative

=, ↑

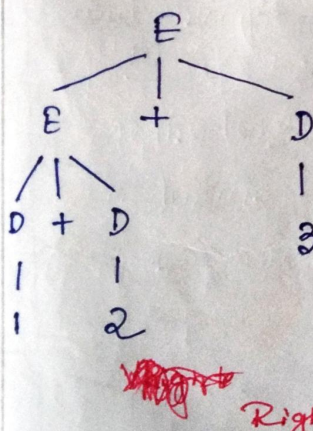
a = b = 5

2 ↑ 3 ↑ 5

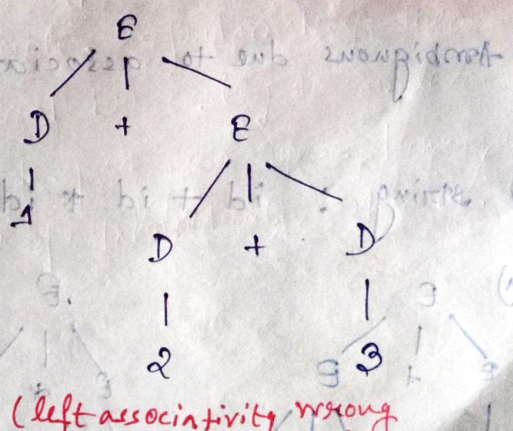




Parse tree for left associative operators are more towards the left side by length.



Right



(left associativity wrong not preserved)

Precedence of operators

Whenever an operator has a higher precedence than the other operator it means that the first operator will get its operands before the operator with the lower precedence.

1 + 2 * 3 → 1 + (2 * 3)

+, − < *, /

Convert Ambiguous Grammar into unambiguous grammar

*, / → left Associative Higher Precedence

+, − → left " lower "

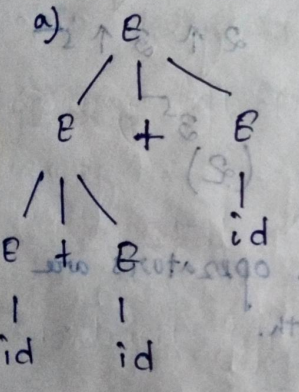
E → E + E

E → E * E

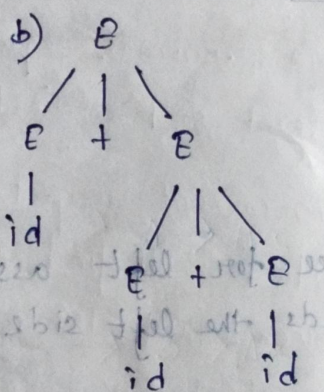
E → id

① string : id + id + id

a)



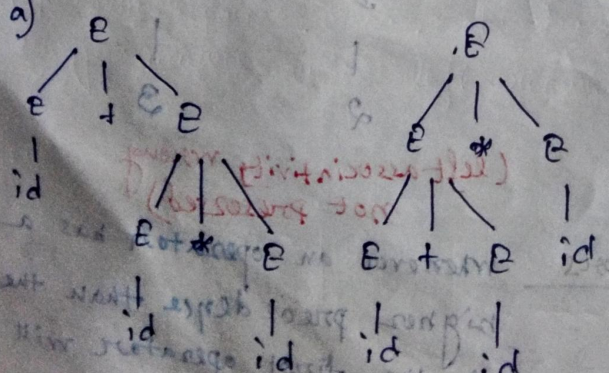
b)



→ Ambiguous due to associativity rule.

② string : id + id * id

a)



→ Ambiguous due to precedence rule.

So, as the solution to the problem is ~~pre~~ maintaining precedence and associativity, we need to introduce several different variables to give the level of "binding strength".

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * f \mid T / F \mid f$

$f \rightarrow id$

E generating the expression having operator (+, -).

T generating the expressions having operator (*, /)

f is the identifier.

Example → 1 + 2 + 3

id + id + id

• Left Recursion → If we have a non-terminal in the left ~~right~~ hand side, that directly or indirectly derives the right hand side which has the same non-terminal as the left most symbol. Then we say that left recursion exists.

If $A \xrightarrow{+} A\alpha$

$E \rightarrow E + T$

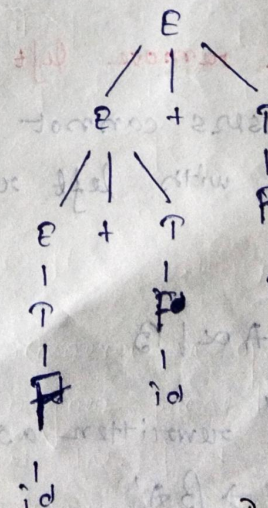
$\rightarrow E + T + T$

$\rightarrow T + T + T$

~~and + id + id~~

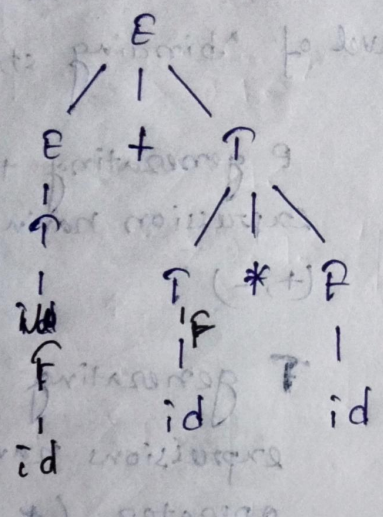
$\rightarrow f + F + f$

$\rightarrow id + id + id$



Preserves Associativity.

Expn → id + id * id



Preserves precedence

Removing left Recursion

1) Direct left recursion → $A \rightarrow Aa$

2) Indirect left recursion → $S \rightarrow Aa$

$A \rightarrow Sb$

$S \overset{+}{\Rightarrow} Sb$

Why should we remove left recursion?

Top Down parsers cannot handle left recursion or grammars with left recursion.

How?

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

↓

can be rewritten as

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' \mid \epsilon$

If there are multiple productions:

$A \to A\alpha_1 | A\alpha_2 | A\alpha_3 | \ldots | A\alpha_m | \beta_1 | \beta_2 | \ldots | \beta_n$

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

$A' \to \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \epsilon$

Advantage

→ Same language will be generated even after removing LR.

Disadvantage → The above procedure only eliminates direct left recursion but not indirect one.

Examples of direct left recursion and their removal.

1) $E \to E + T | T$

$T \to T * F | F$

$F \to (E) | id$ ✓

a) $E \to \underbrace{E + T}_{\alpha} | \underbrace{T}_{\beta}$

$E \to TE'$

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

$T \to \underbrace{T * F}_{\alpha} | \underbrace{F}_{\beta}$

$T \to FT'$

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

2) $S \to S0S1S \mid 01$

α

$S \to 01S'$

$S' \to 0S1S$

$S' \to 01SS' \mid \epsilon$

3) $L \to L,S \mid S$

$L \to SL'$

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

4) $S \to Sx \mid ssb \mid xs \mid a$

$S \to \underbrace{Sx}_{\alpha_1} \mid \underbrace{Ssb}_{\alpha_2} \mid \underbrace{xs}_{\beta_1} \mid \underbrace{a}_{\beta_2}$

$S \to xSS' \mid aS'$

$S' \to xS' \mid sbS' \mid \epsilon$

5) $A \to AA \mid Ab$

$A \to A\underbrace{A}_{\alpha_1} \mid \underbrace{Ab}_{\alpha_2}$

$A' \to AA' \mid bA' \mid \epsilon$

Eliminating Indirect Left Recursion

$A \rightarrow Ba / Aa / c$

$B \rightarrow Bb / Ab / d$

Step 1

First let us eliminate direct left recursion from

$A \rightarrow Ba / Aa / c$

↓ after elimination

$A \rightarrow BaA' / cA'$

$A' \rightarrow aA' / \epsilon$

Now, the grammar becomes,

$A \rightarrow BaA' / cA'$

$A' \rightarrow aA' / \epsilon$

$B \rightarrow Bb / Ab / d$

Step 2

Substituting the productions of A in $B \rightarrow Ab$,

$A \rightarrow BaA' / cA'$

$A \rightarrow aA' / \epsilon$

$B \rightarrow Bb / BaA'b / cA'b / d$

Step 3

Now, eliminating left recursion from the productions of B, we get the following grammar -

$A \rightarrow BaA' / cA'$

$A' \rightarrow aA' / \epsilon$

$B \rightarrow cA'bB' / dB'$

$B' \rightarrow bB' / aA'bB' / \epsilon$

Q.2 Consider the following left recursion

$$X \to Xab / Sa / b$$

$$S \to Sb / Xa / a$$

Step 1

→ Eliminate left recursion from

$$X \to Xab / Sa / b$$

↓ after elimination

$$X \to SaX' / bX'$$

$$X' \to SbX' / \epsilon$$

Now, given grammar becomes

$$S \to X \quad X \to SaX' / bX'$$

$$X' \to SbX' / \epsilon$$

$$S \to Sb / Xa / a$$

Step 2

Substituting the productions of X in $S \to Xa$, we get the following grammar —

$$X \to SaX' / bX'$$

$$X' \to SbX' / \epsilon$$

$$S \to Sb / SaX'a / bX'a / a$$

Step 3

Now, eliminating left recursion from the productions of S, we get the following grammar

$$X \to SaX' / bX' \qquad S \to bX'aS' / aS'$$

$$X' \to SbX' / \epsilon \qquad S' \to bS' / aX'aS' / \epsilon$$

8. Consider the following grammar and eliminate left recursion.

$A \rightarrow Ba/Aa/c$

$B \rightarrow Bb/Ab/d$

Step 1

$A \rightarrow Ba/Aa/c$

Eliminating left recursion,

$A \rightarrow BaA'/cA'$

$A' \rightarrow aA'/\epsilon$

Now, given grammar becomes —

$A \rightarrow BaA'/cA'$

$A' \rightarrow aA'/\epsilon$

$B \rightarrow Bb/Ab/d$

Step 2 Substituting the productions of A in $B \rightarrow Ab$, we get the following grammar—

$A \rightarrow BaA'/cA'$

$A' \rightarrow aA'/\epsilon$

$B \rightarrow Bb/BaA'b/cA'b/d$

Step 3

Now, Eliminating left recursion from the productions of B, we get the following grammar →

$A \rightarrow BaA'/cA'$ $\quad A' \rightarrow aA'/\epsilon$

$B \rightarrow cA'bB'/dB'$ $\quad B' \rightarrow bB'/aA'bB'/\epsilon$

Left Factoring — Left factoring is removing the common left factor that appears in two productions of the same non-terminal. It is done to avoid back-tracking by the parser.

Grammar with common prefixes: If RHS of ~~one~~ more than one production starts with the same symbol, then such a grammar is called as Grammar with common Prefixes.

Example: $A \rightarrow \alpha\beta_1 / \alpha\beta_2 / \alpha\beta_3$

- This kind of grammar creates a problematic situation for top down parsers.
- Top down parsers cannot decide which production must be choosen to parse the string in hand. To remove this confusion, we use left factoring.

A grammar with left factoring present is a non deterministic grammar.

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top Down parsers.

How Do We Remove Left Factoring?

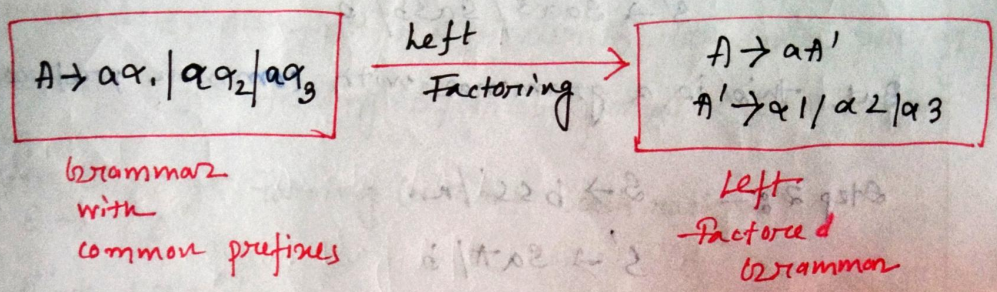
In left factoring,

1) we make one production for each common prefixes.

2) The common prefix may be a terminal or a non-terminal or a combination of both.

3) Rest of the production derivation is added by new productions.

The grammar obtained after the process of left factoring is called as Left-factored Grammar.



Practice Problems

Do the left factoring of the following grammars:

1) $S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

Soln

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

2) $A \rightarrow aAB \mid aBc \mid aAc$

Step 1 :— $A \rightarrow aA'$

$A' \rightarrow AB \mid Bc \mid Ac$

But, again this grammar is with common prefix.

Step 2:— $A \rightarrow aA'$

$A' \rightarrow AD \mid Bc$

$D \rightarrow B \mid c$

3) $S \rightarrow bSSaaS / bSSaSb / bSb / a$

Step 1 :- $S \rightarrow bSS' / a$

$S' \rightarrow SaaS / SaSb / b$

But, this is a grammar with common prefixes.

Step 2 :- $S \rightarrow bSS' / a$

$S' \rightarrow SaA / b$

$A \rightarrow aS / Sb$

4) $S \rightarrow a / ab / abc / abcd$

Step 1 :- $S \rightarrow aS'$

$S' \rightarrow b / bc / bcd / \epsilon$

Step 2 :- $S \rightarrow aS'$

$S' \rightarrow bA / \epsilon$

$A \rightarrow c / cd / \epsilon$

Step 3 :- $S \rightarrow aS'$

$S' \rightarrow bA / \epsilon$

$A \rightarrow cB / \epsilon$

$B \rightarrow d / \epsilon$.

Top-Down Parsers

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first). Equivalently, top-down parsing can be viewed as finding leftmost derivation for an input string.

Recursive Descent Parsing (without Backtracking)

$E \rightarrow TE'$

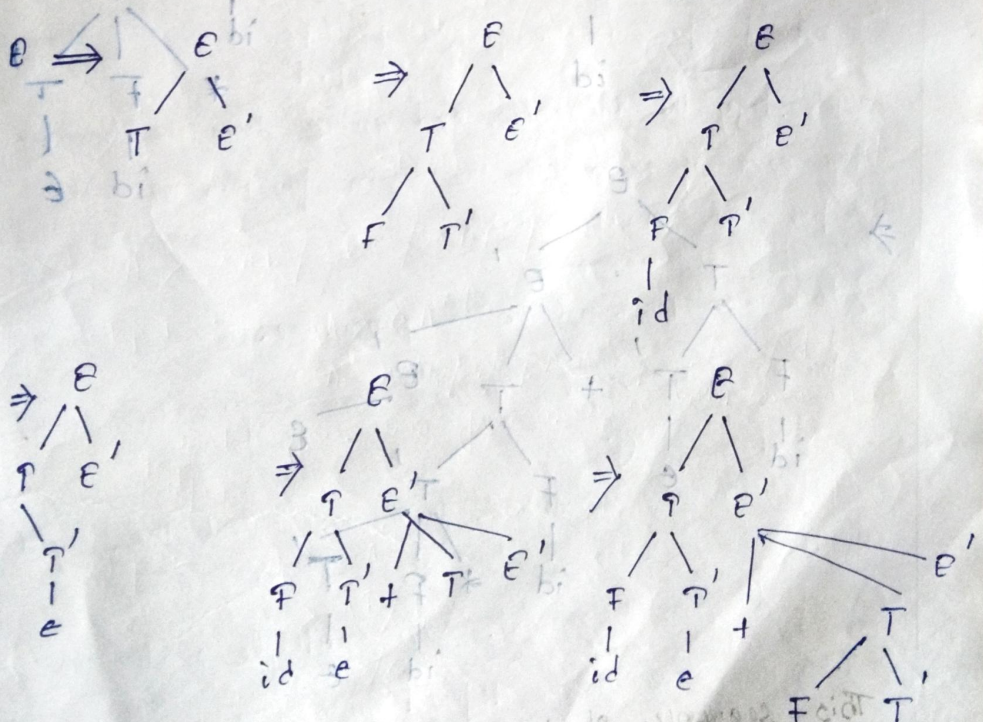
$E' \rightarrow +TE' / \epsilon$

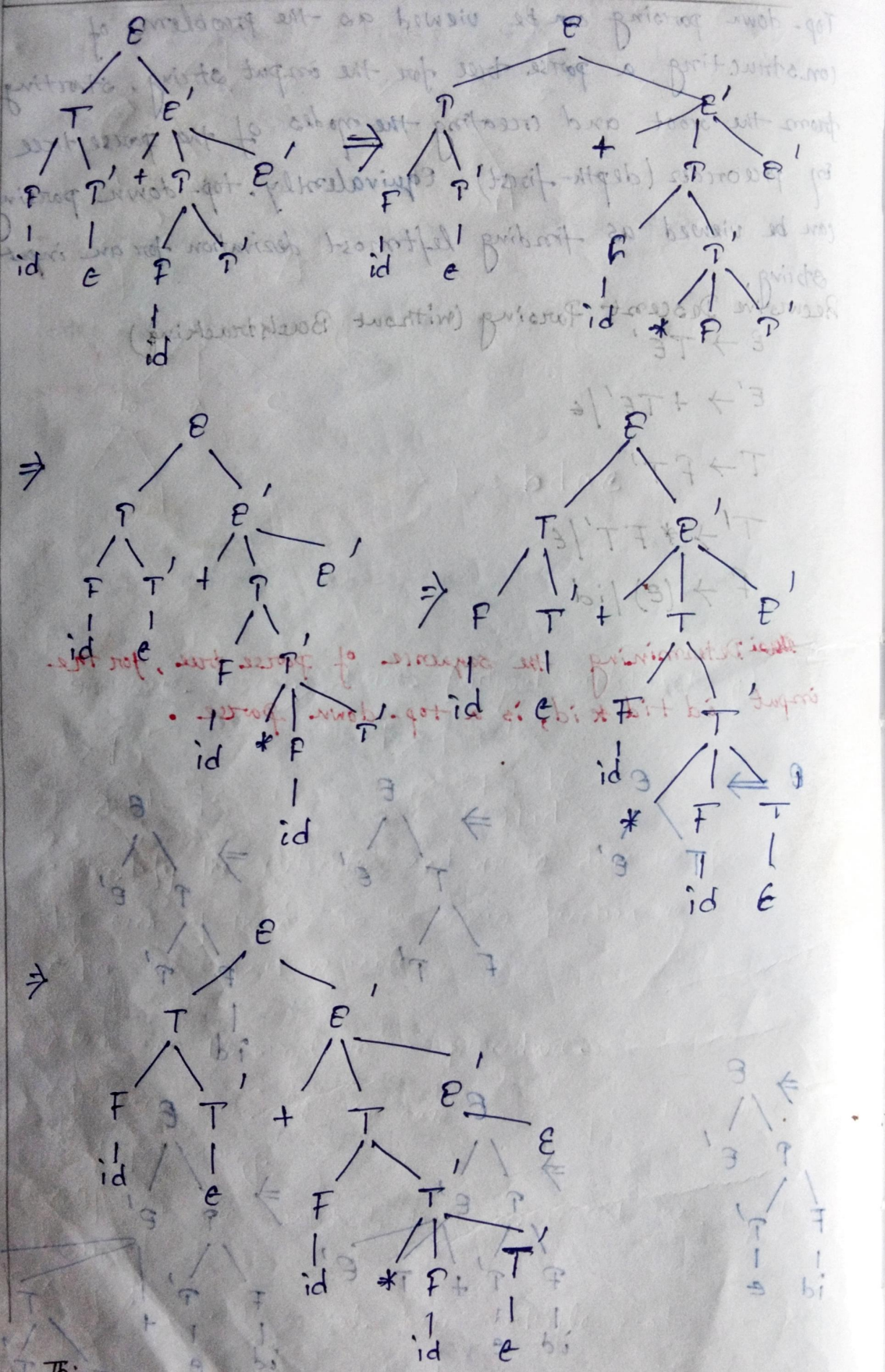
$T \rightarrow FT'$

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

$F \rightarrow (E) / id$

Determining the sequence of parse tree, for the input id + id * id, is a top-down parse.





This sequence of trees corresponds to a leftmost derivation of the input.

Recursive - Descent Parsing with Backtracking (Brute force Method)

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.

General recursive descent may require backtracking; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently.

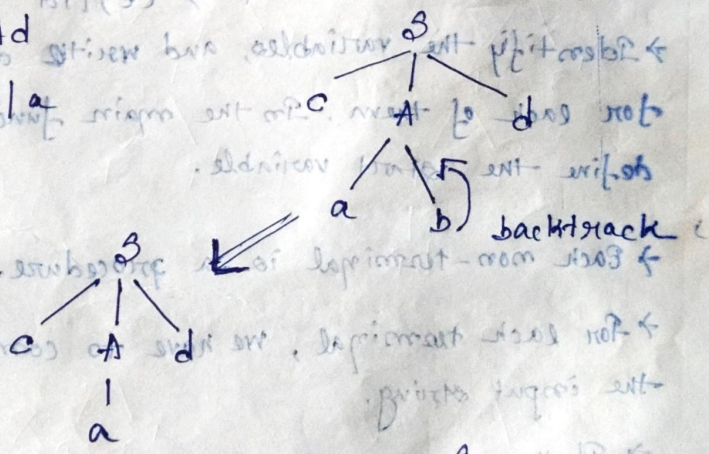
In case of backtracking, we cannot choose a unique production at each step. So, we must try each of several productions in some order. If we fail to generate the desired terminal. Then we must backtrack.

Example

1) $S \rightarrow cAd$

$A \rightarrow ab \mid a$

$L = cad$

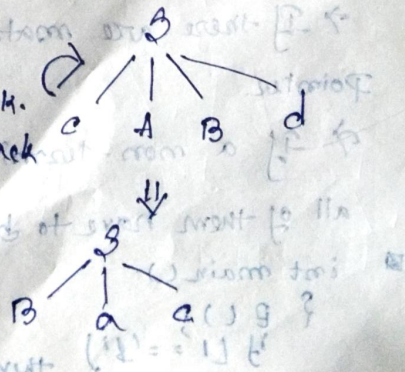


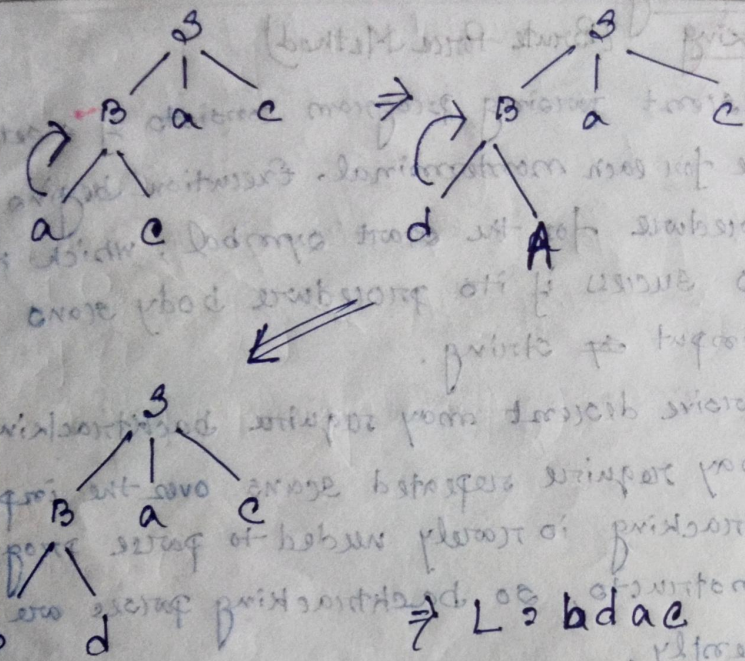
2) $S \rightarrow cABd \mid Bac$

$A \rightarrow Bcd \mid aab$

$B \rightarrow ac \mid dA \mid bd$

$L = bdac$





Recursive Descent Parser (without backtracking)

Ex

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

→ Identify the variables, and write down the function for each of them. In the main function we will define the start variable.

→ Each non-terminal is a procedure.

→ For each terminal, we have to compare them with the input string.

→ If there are matched, then increment the input pointer.

→ If a non-terminal have multiple productions, all of them have to be defined within a single procedure.

```
int main()
{ E();
  if (l'=='$')
```

Every input string is concatinated with $ at the end.

then ("parsing successful");

```
Procedure E()
{
    T();
    Eprime();
}

Procedure Eprime()
{
    if (inputSymbol == '+')
    then
        input++; / advance();
        T();
        Eprime();
    }
    else
        return();
}

Procedure T()
{
    F();
    Tprime();
}

Procedure Tprime()
{
    if (input == '*')
    then
        advance();
        F();
        Tprime();
    }
    else
        return();
}

Procedure F()
{
    if (input == '(')
    then
        advance();
        E();
    }
    else if (input == ')')
    then
        advance();
    }
    else
    {
        if (input == 'id')
        then
            advance();
        }
```

* for each non-terminal we have to call them recursively.

* for each terminal we have to perform comparison.

* for ε, return null.

Consider the string id + id $, and check if it can be parsed using RDP.

```
int main()
{
    E();
    if (l == '$')
        printf("Parsing successful");
}
```

Procedure E()

↓

Procedure T()

↓

Procedure f()

↓

if → condition failed

else if → condition failed

else → successful

advance()

↓

Procedure Tprime()

↓

if → condition failed, return null

↓

Procedure Eprime()

↓

if → condition matched → successful → advance()

↓

Procedure E() → Procedure T() → Procedure f()

↓

if → failed

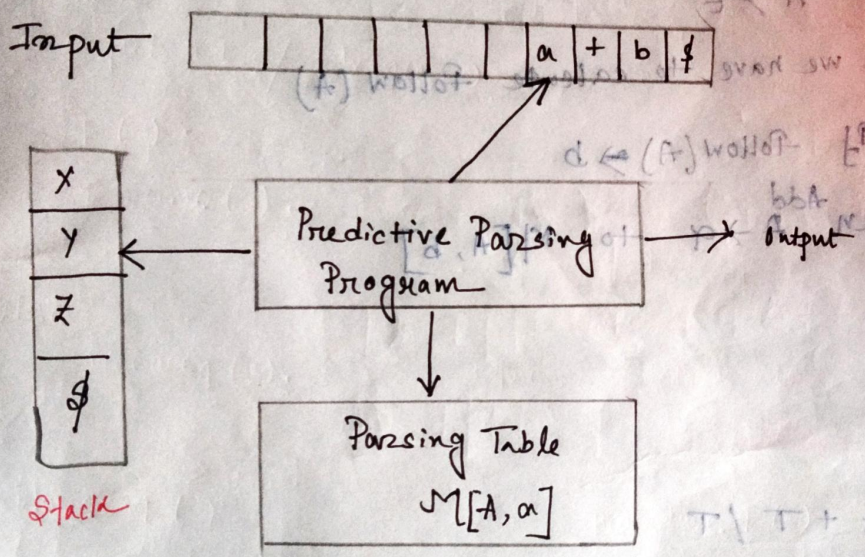
else → successful → advance() → main → parsing successful.

Non-Recursive Descent Parsing / Predictive Parsing / LL(1)

LL(1) → first L specifies, we are scanning the i/p string from left-to right.

Second L specifies, we are using left most derivation.

1 specifies we are only scanning one i/p symbol.



Model for table-driven predictive parsing

Steps to design LL(1) parsing

1. Elimination of Left Recursion.
2. " of " factoring.
3. calculation of first() and follow().
4. Construction of Parsing table.
5. Check whether the input string is accepted.

Construction of Parse Table

Rules.

1) If $A \to \alpha$

and first(α) $\to$ a (terminal symbol)

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

2) If first(α) $\to \epsilon$

or $A \to \epsilon$

then we have to calcate follow(A)

If follow(A) $\Rightarrow$ b

then Add $A \to \alpha$ to $M[A, b]$

Q. $E \to E + T / T$

$T \to T * F / F$

$F \to (E) / id$

Step 1

Elimination of left recursion

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) / id$

Step 2

elimination of left factoring

There is no ~~left fact~~ common prefix ~~prod~~ production is present.

Step 3

Calculate first and follow

first (E) = { (, id }

first (E') = { +, ε }

first (T) = { (, id }

first (T') = { *, ε }

first (F) = { (, id }

follow (E) = { $,) }

follow (E') = { $,) }

follow (T) = first (E') contains ε = follow (E)

= { +, $,) }

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

follow (F) = first (T')

= { *, $,) }

Step 4

Construction of Parsing Table

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

1. E→TE'

A α

first (TE') = { (, id }

M[E, (]

M[E, id]

Rules for Calculating follow function

Rule 1

For the start symbol S,
Place $ in follow (S).

Rule 2

For any production rule

$A \rightarrow \alpha B$

follow (B) = follow (A)

Rule 3

For any production rule $A \rightarrow \alpha B \beta$

- If $\epsilon \notin$ first (β), then follow (B) = first (β)
- If $\epsilon \in$ first (β), then follow (B) =

$$\{\text{First}(\beta) - \epsilon\} \cup \text{follow}(A)$$

Notes

- ϵ may appear in the first function of a non-terminal
- ϵ will never " " " follow " " " "
- we are calculating the follow function of a non-terminal by looking where it is present on the RHS of a production rule.

Calculation of first and follow

First function - first(α) is a set of terminal symbols that begin in strings derived from α.

consider the production rule -

$A \rightarrow abc / def / ghi$, then first(A) = {a, d, g}

Rules for calculating first function

Rule 1 For a production rule $X \rightarrow \epsilon$

first(X) = {ε}

Rule 2 If $A \rightarrow a\alpha$

For any terminal symbol 'a'.

first(A) = {a}

Rule 3 For a production rule

$X \rightarrow Y_1 Y_2 Y_3$

first(X) → 1) If $\epsilon \notin$ first(Y_1)

then first(X) = first(Y_1)

2) If $\epsilon \in$ first(Y_1)

then first(X) = {first(Y_1) - ε} ∪ first($Y_2 Y_3$)

first($Y_2 Y_3$)

→ If $\epsilon \notin$ first(Y_2)

then first($Y_2 Y_3$) = first(Y_2)

→ If $\epsilon \in$ first(Y_2),

then first($Y_2 Y_3$) = {first(Y_2) - ε} ∪ first(Y_3)

2. a) $\frac{E'}{A} \rightarrow \frac{+TE'}{\alpha}$

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

Add $E' \rightarrow +TE'$ to $M[E', +]$

b) $\frac{E'}{A} \rightarrow \frac{\epsilon}{\alpha}$

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

Add $E' \rightarrow \epsilon$ to $M[E', \$]$

Add $E' \rightarrow \epsilon$ to $M[E',)]$

3. $T \rightarrow FT'$

First $(FT') = \{(, id\}$

Add $T \rightarrow FT'$ to $M[F, (]$

$M[F, id]$

4. a) $T' \rightarrow *FT'$

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

Add $T' \rightarrow *FT'$ to $M[T', *]$

b) $T' \rightarrow \epsilon$

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

Add $T' \rightarrow \epsilon$ to $M[T', +]$

$M[T', \$]$

$M[T',)]$