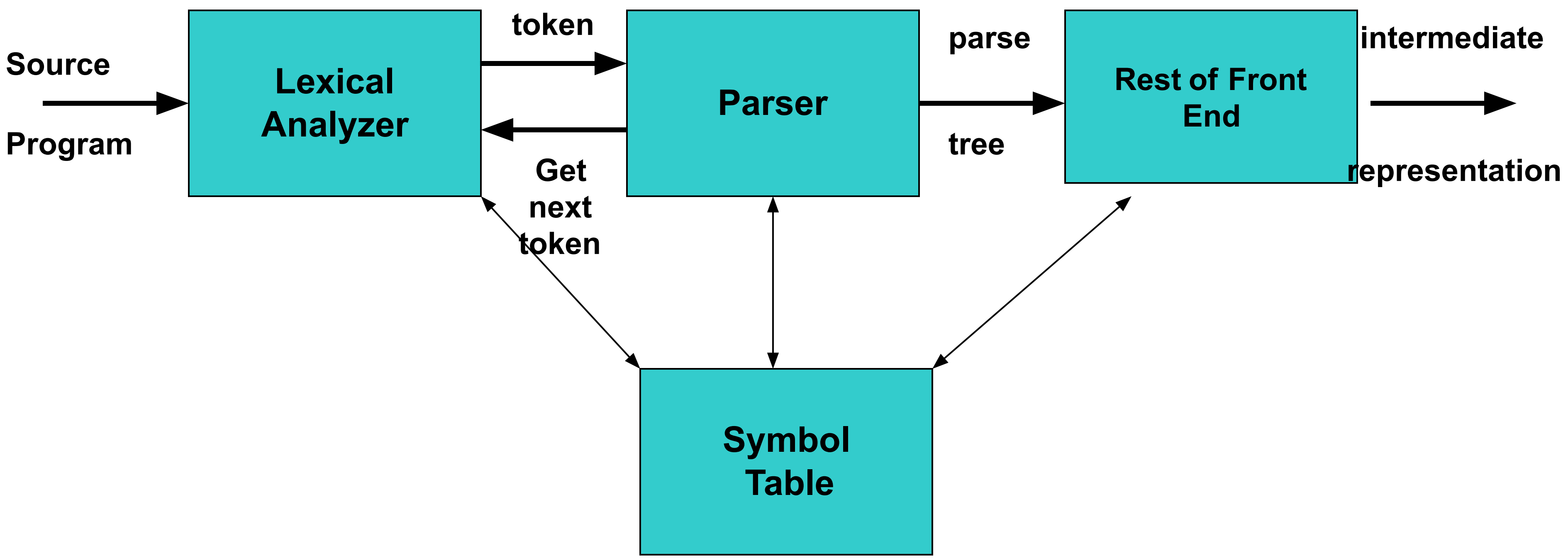
Topics covered

- Introduction
- Issues in the design of parser/syntax analyzer
- Context Free Grammar
- Writing a grammar
- Role of Parser
- Top Down Parsing
  - Basics
    - Left recursion and Left factoring
  - General strategy
  - Recursive Descent parser and its implementation
  - Difficulties of RDP
  - Predictive parser
    - Prerequisite
    - FIRST and FOLLOW computation
    - Working principle and Algorithm
    - Trace
    - Error Recovery in predictive parsing
- Conclusion

Syntax Analyser

- Introduction :
  - Syntax Analyser determines the structure of the program.
  - The tokens generated from Lexical Analyser are grouped together and checked for valid sequence defined by programming language.
  - Syntax Analyser uses context free grammar to define and validate rules for language construct.
  - Output of Syntax Analyser is parse tree or syntax tree which is a hierarchical / tree structure of the input.
- Issues in the design
  - There is a need of mechanism to describe the structure of syntactic units or syntactic constructs of programming language. Context free grammar
  - There is a need of mechanism to recognize the structure of syntactic units or syntactic constructs of programming language. Automata.

Position of Parser and role in Compiler model



Role of a parser.

- The stream of tokens is input to the syntax analyzer. The job of the parser is:
  - To identify the valid statement represented by the stream of tokens as per the syntax of the language. If it is a valid statement, it will be represented by a parse tree.
  - If it is not a valid statement, then a suitable error message is displayed, so that the programmer is able to correct the syntax error.
- Usually the semantic analysis and intermediate code generation can interspersed with parsing. Hence, in addition to the validation of the programming statements parser also performs the following tasks :
  - Type-checking and providing the semantic consistency to the source programs.
  - Execution of semantic actions that are attached with grammar and responsible for generating the required intermediate form for the source program that facilitates some kind of code optimization

Classification of Parser

Syntax Analyzer

- The syntax of a programming is described by a *context-free grammar (CFG)*. It offers the following significant benefits for both language designer and compiler writer:
  - A grammar gives a precise, yet easy to understand syntactic specification of a programming languages.
  - For certain class of grammars we can construct automatically, an efficient parser that that determines and validates the syntactic structure of a source program.
  - Properly designed grammar is useful for translating the source program into a correct object code and for detecting errors.
  - A grammar allows a language to be evolved or developed iteratively, by adding new constructs to perform new tasks. The new constructs can be integrated more easily into an implementation that follows grammatical structure of the language.

Parsers (cont.)

- As per our course syllabus We categorize the parsers into two groups:

1. **Top-Down Parser**
   - the parse tree is created top to bottom, starting from the root.
2. **Bottom-Up Parser**
   - the parse is created bottom to top; starting from the leaves

- Both top-down and bottom-up parsers scan the input from left to right (one symbol at a time).
- Efficient top-down and bottom-up parsers can be implemented only for sub-classes of context-free grammars.
  - LL for top-down parsing
  - LR for bottom-up parsing

Context-Free Grammars

- Inherently recursive structures of a programming language are defined by a context-free grammar.
- In a context-free grammar-G={ V, T, S, P },
  we have:
  - V : A finite set of non-terminals (syntactic-variables)
  - T : A finite set of terminals (in our case, this will be the set of tokens or lexical units)
  - S : A start symbol (one of the non-terminal symbol)
  - P : A finite set of productions rules in the following form
    - $A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)
- Example:

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

Derivations

$E \Rightarrow E+E$

- E+E derives from E
  - we can replace E by E+E
  - to able to do this, we have to have a production rule $E \rightarrow E+E$ in our grammar.

$$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+id$$

- A sequence of replacements of non-terminal symbols is called a **derivation** of id+id from E.
- In general a derivation step is

$\alpha A\beta \Rightarrow \alpha\gamma\beta$ if there is a production rule $A \rightarrow \gamma$ in our grammar
where α and β are arbitrary strings of terminal and non-terminal symbols

$\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)

$\Rightarrow$: derives in one step

$\overset{*}{\Rightarrow}$: derives in zero or more steps

$\overset{+}{\Rightarrow}$: derives in one or more steps

CFG - Terminology

- L(G) is *the language of G* (the language generated by G) which is a set of sentences.
- *A sentence of L(G)* is a string of terminal symbols of G.
- If S is the start symbol of G then

  w is a sentence of L(G) iff $S \Rightarrow w$ where w is a string of terminals of G.

- If G is a context-free grammar, L(G) is a *context-free language*.
- Two grammars are *equivalent* if they produce the same language.

- $S \stackrel{*}{\Rightarrow} \alpha$ - If α contains non-terminals, it is called as a *sentential* form of G.

  - If α does not contain non-terminals, it is called as a *sentence* of G.

Derivation Example

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

OR

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

- At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.
- If we always choose the left-most non-terminal in each derivation step, this derivation is called as **left-most derivation**.
- If we always choose the right-most non-terminal in each derivation step, this derivation is called as **right-most derivation**.

Left-Most and Right-Most Derivations

Left-Most Derivation

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(id+E) \underset{lm}{\Rightarrow} -(id+id)$$

Right-Most Derivation

$$E \underset{rm}{\Rightarrow} -E \underset{rm}{\Rightarrow} -(E) \underset{rm}{\Rightarrow} -(E+E) \underset{rm}{\Rightarrow} -(E+id) \underset{rm}{\Rightarrow} -(id+id)$$

- We will see that the top-down parsers try to find the left-most derivation of the given source program.

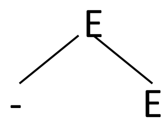
- We will see that the bottom-up parsers try to find the right-most derivation of the given source program in the reverse order.

Parse Tree

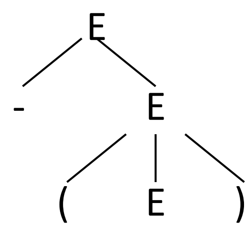
- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.

- A parse tree can be seen as a graphical representation of a derivation.

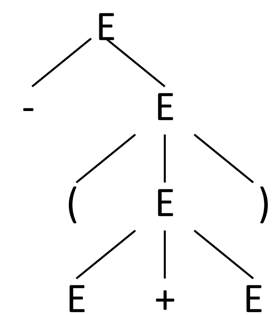
$E \Rightarrow -E$



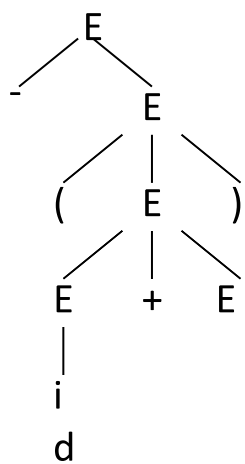
$\Rightarrow -(E)$



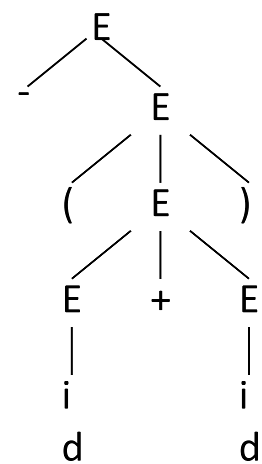
$\Rightarrow -(E+E)$



$\Rightarrow -(id+E)$



$\Rightarrow -(id+id)$



Example

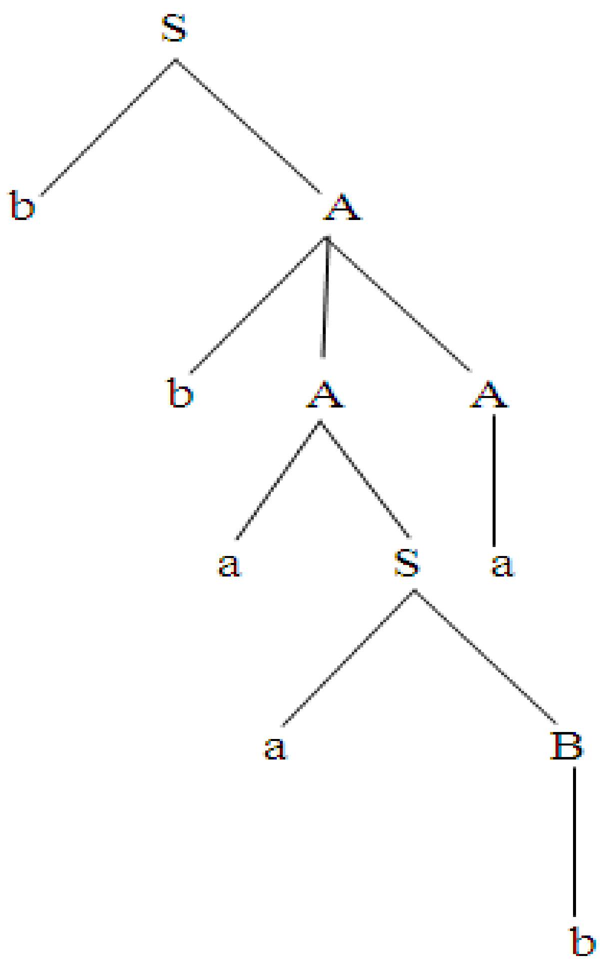
Consider the grammar

- $S \rightarrow b\ A \mid a\ B$
  $A \rightarrow b\ A\ A \mid a\ S \mid a$
  $B \rightarrow a\ B\ B \mid b\ S \mid b$

Write leftmost and rightmost derivation for the following sentences along with Parse tree.

i. bbaaba **ii. bbbaaaba**

i) bbaaba



ii) bbbaaaba (Home Work)

Leftmost Derivation

$$
\begin{aligned}
S &\Rightarrow b\ A \\
&\Rightarrow b\ b\ \underline{A}\ A \\
&\Rightarrow b\ b\ a\ \underline{S}\ A \\
&\Rightarrow b\ b\ a\ a\ \underline{B}\ A \\
&\Rightarrow b\ b\ a\ a\ b\ A \\
&\Rightarrow b\ b\ a\ a\ b\ a
\end{aligned}
$$

Rightmost derivation

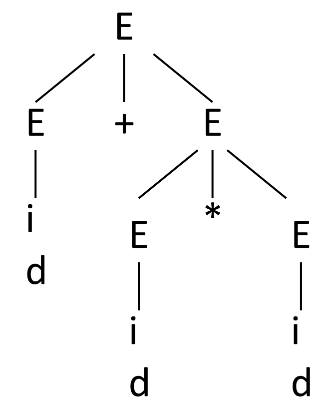
$$
\begin{aligned}
S &\Rightarrow b\ \underline{A} \\
&\Rightarrow b\ b\ A\ \underline{A} \\
&\Rightarrow b\ b\ \underline{A}\ a \\
&\Rightarrow b\ b\ a\ \underline{S}\ a \\
&\Rightarrow b\ b\ a\ a\ \underline{B}\ a \\
&\Rightarrow b\ b\ a\ a\ b\ a
\end{aligned}
$$

Fig. 3.5 Parse Tree for the string bbaaba

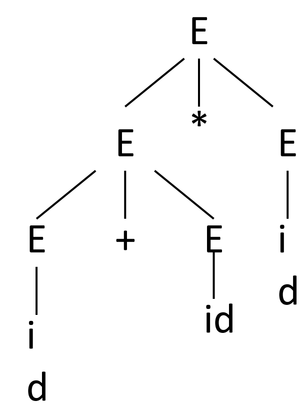
Ambiguity

- A grammar produces more than one parse tree for a sentence is called as an ***ambiguous*** grammar.

$$E \Rightarrow E{+}E \Rightarrow id{+}E \Rightarrow id{+}E{*}E$$
$$\Rightarrow id{+}id{*}E \Rightarrow id{+}id{*}id$$



$$E \Rightarrow E{*}E \Rightarrow E{+}E{*}E \Rightarrow id{+}E{*}E$$
$$\Rightarrow id{+}id{*}E \Rightarrow id{+}id{*}id$$

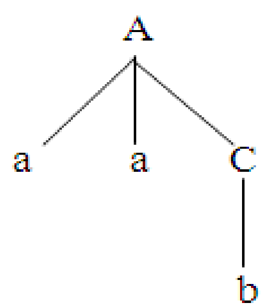


Example : $\mathbf{A \rightarrow B\ C \mid a\ a\ C}$
$\mathbf{B \rightarrow a \mid B\ a}$
$\mathbf{C \rightarrow b}$

Example: Consider the following ambiguous grammar

A → B C | a a C
B → a | B a
C → b

Tree 1

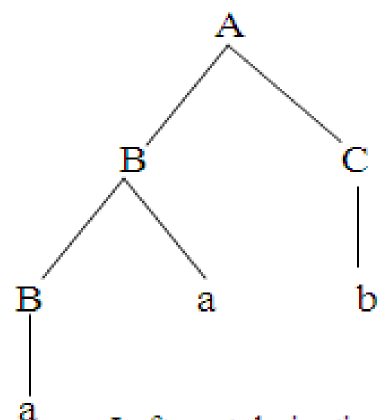


Leftmost derivation

$A \Rightarrow a \ \ a \ \ \underline{C}$

$\Rightarrow a\ a\ b$

Tree 2



Leftmost derivation

$A \ \Rightarrow \underline{B}\, C$

$\Rightarrow \underline{B}\, a \ \ c$

$\Rightarrow \ a \ \ a \ \ c \Rightarrow \ a \ \ a \ \ b$

Fig. 3.20 Two leftmost derivation for string a a b

Ambiguity (cont.)

- For the most parsers, the grammar must be unambiguous.

- unambiguous grammar

  ⍰ unique selection of the parse tree for a sentence

- We should eliminate the ambiguity in the grammar during the design phase of the compiler.
- An unambiguous grammar should be written to eliminate the ambiguity.
- We have to prefer one of the parse trees of a sentence (generated by an ambiguous grammar) to disambiguate that grammar to restrict to this choice.

Ambiguity – Operator Precedence

Let us consider the grammar

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\text{^}E \mid id \mid (E)$

- At each step we begin by introducing One Non terminal - NT for each precedence level.

Priority levels (High to low)

Exponentiation **^** - F is NT (right associative rule)

Multiplicative operator **(*, /)** - T is NT (right associative rule)

Additive operator **(+ ,-)** - E is NT (right associative rule)

- A subexp 'E' that is indivisible is either an identifier or parenthesized expression which is written as

$G \rightarrow id \mid (E)$ where G is New NT

- To write next rule we take New NT for the next highest priority level and this is connected with o Zero or more instance of next highest priority operator with previous level NT

$F \rightarrow G\text{^}F \mid G$

Ambiguity – Operator Precedence

- Ambiguous grammars (because of ambiguous operators) can be disambiguated according to the precedence and associativity rules.

$$E \rightarrow E+E \mid E^*E \mid E\verb|^|E \mid id \mid (E)$$

$\Downarrow$ disambiguate the grammar

precedence: ^ (right to left)

* (left to right)

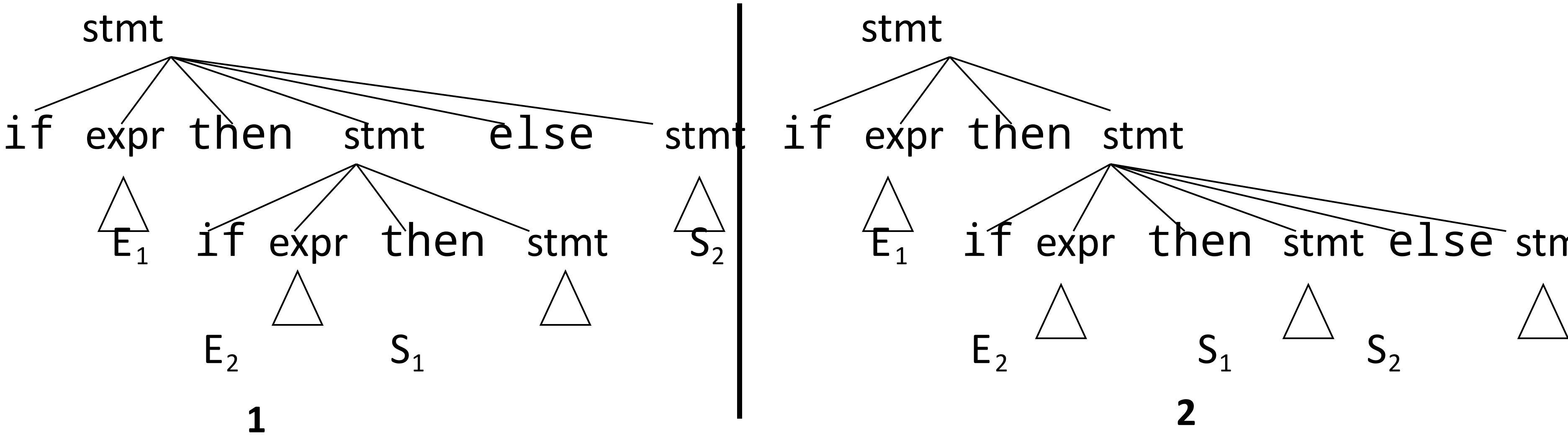
\+ (left to right)

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T^*F \mid F$$
$$F \rightarrow G\verb|^|F \mid G$$
$$G \rightarrow id \mid (E)$$

Ambiguity (cont.)

stmt → **if** expr **then** stmt |
if expr **then** stmt **else** stmt | otherstmts

if E_1 then if E_2 then S_1 else S_2



Ambiguity (cont.)

- We prefer the second parse tree (else matches with closest then).
- So, we have to disambiguate our grammar to reflect this choice.

- The unambiguous grammar will be:

stmt → matchedstmt
  | unmatchedstmt

matchedstmt → **if** expr **then** matchedstmt **else** matchedstmt
  | otherstmts

unmatchedstmt → **if** expr **then** stmt
  | **if** expr **then** matchedstmt **else** unmatchedstmt

Left Recursion

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

  $A \overset{+}{\Rightarrow} A\alpha$ for some string α

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\, A'$

$A' \rightarrow \alpha\, A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\, A' \mid \ldots \mid \beta_n\, A'$

$A' \rightarrow \alpha_1\, A' \mid \ldots \mid \alpha_m\, A' \mid \varepsilon$ an equivalent grammar

Immediate Left-Recursion -- Example

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T\,E'$

$E' \rightarrow +T\,E' \mid \varepsilon$

$T \rightarrow F\,T'$

$T' \rightarrow *F\,T' \mid \varepsilon$

$F \rightarrow id \mid (E)$

Left-Recursion -- Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$
$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or
$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

Eliminate Left-Recursion -- Algorithm

- Arrange non-terminals in some order: $A_1 \ldots A_n$

- for i **from** 1 **to** n **do** {

    - **for** j **from** 1 **to** i-1 **do** {

        replace each production

            $A_i \rightarrow A_j \gamma$

                by

            $A_i \rightarrow \alpha_1 \gamma \mid \ldots \mid \alpha_k \gamma$

            where $A_j \rightarrow \alpha_1 \mid \ldots \mid \alpha_k$

    }

    - eliminate immediate left-recursions among A_i productions

}

Eliminate Left-Recursion -- Example

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:

- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$
  So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid f$
- Eliminate the immediate left-recursion in A

$$A \rightarrow bdA' \mid fA'$$
$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

So, the resulting equivalent grammar which is not left-recursive is:

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid fA'$$
$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

Eliminate Left-Recursion – Example2

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop.
- Eliminate the immediate left-recursion in A

  $A \rightarrow SdA' \mid fA'$
  $A' \rightarrow cA' \mid \varepsilon$

for S:

- Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid fA'a$
  So, we will have $S \rightarrow SdA'a \mid fA'a \mid b$
- Eliminate the immediate left-recursion in S

  $S \rightarrow fA'aS' \mid bS'$
  $S' \rightarrow dA'aS' \mid \varepsilon$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow fA'aS' \mid bS'$
$S' \rightarrow dA'aS' \mid \varepsilon$
$A \rightarrow SdA' \mid fA'$
$A' \rightarrow cA' \mid \varepsilon$

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

  grammar a new equivalent grammar suitable for predictive parsing

stmt → `if` expr `then` stmt `else` stmt
  |`if` expr `then` stmt

- when we see `if`, we cannot immediately decide which production rule to choose to expand *stmt* in the derivation.

Left-Factoring (cont.)

- In general,

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ Here α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

- while processing if the input begins string derived from α we do not know or decide whether to expand

  A to $\alpha\beta_1$ or

  A to $\alpha\beta_2$

  However we can defer the decision by first expanding A to $\alpha A'$ and then after seeing the i/p derived from α and look ahead symbol, we expand A' to β_1 or β_2. This is left factored and the production are re-written for the grammar and is as follows:

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring -- Algorithm

- Input : Grammar G
- Output : An equivalent left factored grammar
- Method :

1. For each Non-terminal A find the longest prefix α common to two or more alternatives (production rules).

2. If α <> ε then replace all of A-productions

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

where γi represents all alternatives that do not begin with α

by

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$

Here A' is a new Non-terminal

$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

3. Step 1 and 2 are repeated until no two alternatives for a Non-terminal have a common prefix.

- $A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

Left-Factoring – Example1

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Left-Factoring – Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

Parsing

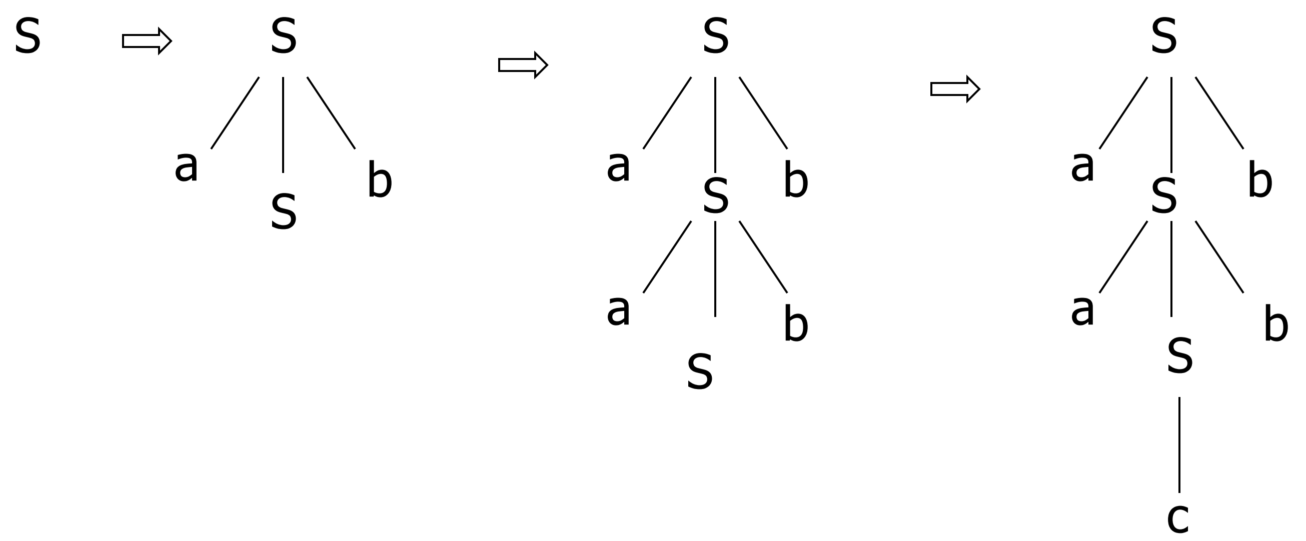
- **Top down parsing**

  In top down parsing we start from the start symbol of the grammar and by choosing the production judiciously we try to derive the given sentence.

- **Bottom-up parsing**

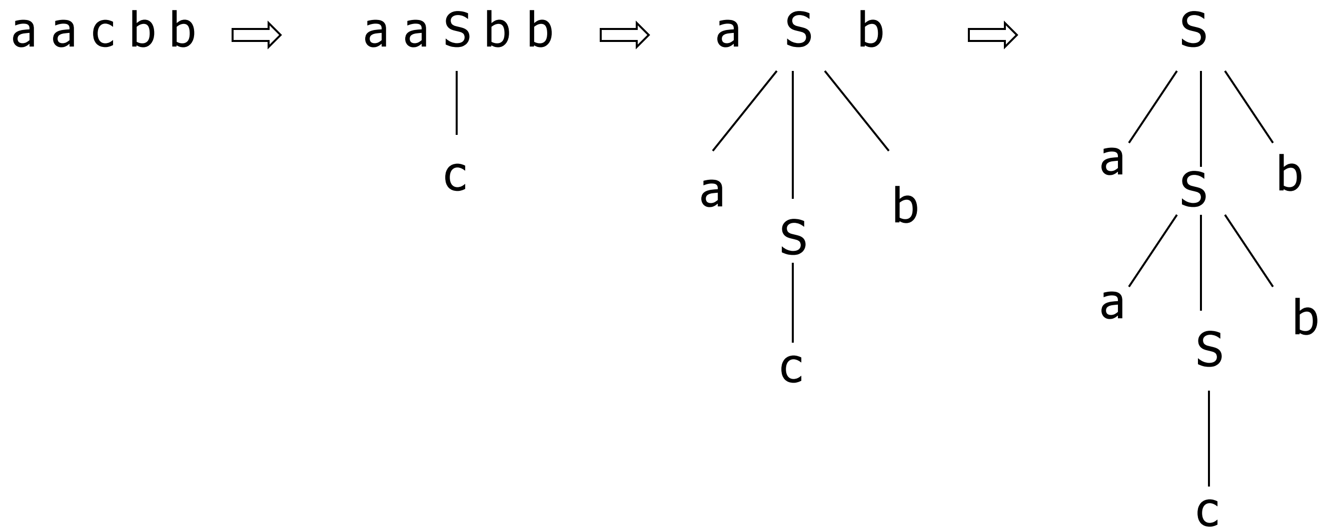
  In bottom-up parsing we start from the given sentence and using various production, we try to reach the start symbol.

Consider the grammar S -> aSb | c for the string aacbb



Top down parsing

Consider the grammar S -> aSb | c for the string aacbb



Bottom-up parsing

Design strategy for Top down Parser

- General strategy :

Basically Top down parsing can be viewed as an attempt to find leftmost derivation for an input string and constructs a parse tree from root to leaves. The following steps may be followed.

1. Given a Non-terminal (Initially Start symbol) which is to be expanded, the first alternative (production rule) is used for expansion.
2. Within the newly expanded string, the substring of terminals from left are compared with input string. If found to be match then next left most Non terminal is selected for expansion and the step 2 is repeated.
3. Otherwise the current alternative structure(production rule) selected is incorrect, hence undo the previous expansion and use the next alternative structure of the Non-terminal for expansion and step 2 is repeated.
4. In the process of step 2 and 3 if No alternative structure for a Non-terminal to be tried then process is backed up by undoing all the previous expansion. In the process of backtracking, If we reach start symbol and no alternative structure to be tried then input is invalidated. Otherwise if no Non-terminal are left for expansion then input is validated.

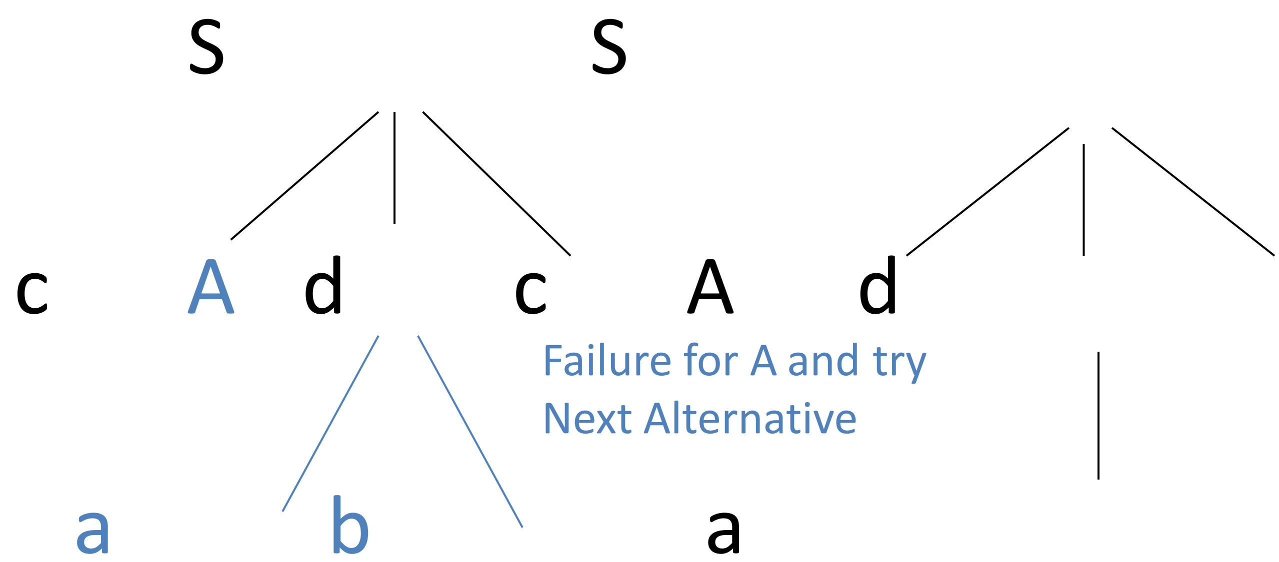
Recursive-Descent Parsing (uses Backtracking)

- Example:

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

input: cad



Pseudo Code for implementation

```
Main()
{
    i=1;   /* index pointing to input string */
    read input
     if (S() and input[i] = $)
          print(" String is valid ");
     else
          print(" String is Invalid ");
}
```

```
int S()
{
  If input[i] ='c' then
  {
     i=i+1;
      If A() then
      {
          if input[i]='d'
          {
               i=i+1;
               return 1;
          }
      }
      else
                return 0;
  }
   else
        return 0;
}
```

```
Int A()
{
  isave=i;
  if input[i] = 'a' then
  {
     i=i+1;
     if input[i] = 'b' then
     {
        i=i+1;
        return 1;
     }
   }
      i=isave;
      if input[i]='a' then
      {
          i=i+1;
          return 1;
       }
   else
   {
       return 0;
    }
}
```

Recursive-Descent Parsing (uses Backtracking)

- In order to find the correct production the general form of top down parser uses **backtracking** (via recursive calls) and is called is **Recursive Descent parser**.
- It consists of **set of procedures**, one for each Non-terminal and looks for a substring of input string and if it is found it returns **TRUE**, otherwise, it returns **FALSE.**
- Execution begins with a call to procedure for **start symbol** which **halts and announces successful parsing** if its procedure body scans the entire input string, otherwise **it announces unsuccessful parsing.**
- The pseudo-code for a typical Non-terminal may be written as follows :

Pseudo-code for Non-terminal in Recursive-descent Parser (RDP)

```
Void A()
{
   Choose an A-production A → X1X2X3....Xk
    for ( i=1 to k)
    {
       if (Xi is Non-terminal)
              call procedure Xi()
       else if (Xi equals the current input symbol 'a' )
              advance the input to the next symbol
       else
              error()   /* error and try next alternative for A-
    production */
    }
}
```

Difficulties of Recursive Descent parser

1. A left recursive grammar creates top down parser to go into an infinite loop. i.e if $A \rightarrow A\alpha$ is a A-production then, when we try to expand A, we may find ourselves again trying to expand A without having consumed any input.
2. A second problem concerns backtracking. If we make a sequence of erroneous expansion, we may have to undo the semantic action taken. This slows the process of parsing hence backtracking must be avoided.
3. The order in which the alternative structure (production rules) are selected for Non-terminal would affect the language accepted.

Prerequisites for Predictive topdown parsers

- Elimination of Left-recursion
- Left Factoring
- First Set
- Follow Set

FIRST AND FOLLOW SETS

- The implementation of both Top-down parser and bottom parser is aided by two function name **FIRST** and **FOLLOW** sets associated with **grammar G**.
- These two sets allow us choose which production to be selected based on the next input symbol

- FIRST(α) : It is defined to be the set terminals that begin string derived from α.
- How it is used ?

Consider two A-production

$A \rightarrow \alpha \mid \beta$ where FIRST(α) and FIRST(β) are disjoint sets.

Let us consider the terminal 'a' to be first symbol which is either in FIRST(α) or FIRST(β) but not in both. When choosing A-production we see the look-ahead symbol 'a' from the input. If 'a' in FIRST(α) then select A production as $A \rightarrow \alpha$ or If 'a' in FIRST(β) then select A production as $A \rightarrow \beta$

FIRST set Computation

FIRST(**X**) for all Grammar symbols **X** can be computed by applying 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 = ε or X → ε then** FIRST(**X**) = {**ε** }
3. **IF X is a Non-Terminal and X → $Y_1Y_2Y_3 \cdots Y_k$ then**

   FIRST(**X**)=FIRST($Y_1Y_2Y_3 \cdots Y_k$)

   = FIRST(Y_1) → **if FIRST(Y_1) does not derive any empty string ε**

   FIRST(**X**)=FIRST($Y_1Y_2Y_3 \cdots Y_k$)

   = FIRST(Y_1) – {ε } U FIRST($Y_2Y_3 \cdots Y_k$)

   → **if Y_1 derive an empty string ε.**

$\mathrm{FIRST}(Y_2Y_3\ldots Y_k) = \mathrm{FIRST}(Y2)$

→ **if Y_2 does not derive an empty string ε.**

$\mathrm{FIRST}(Y_2Y_3\cdots Y_k) = \mathrm{FIRST}(Y2) - \{\varepsilon\} \cup \mathrm{FIRST}(Y_3\ldots Y_k)$

→ **if Y_2 derive an empty string ε.**

This is repeated for each Yi until no more terminals or ε can be added

1. $E \rightarrow E+T \mid T$

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

$F \rightarrow id \mid (E)$

2. $E \rightarrow T E'$

$E' \rightarrow +T E' \mid \varepsilon$

$T \rightarrow F T'$

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

$F \rightarrow id \mid (E)$

3. $S \rightarrow ACB \mid CbB \mid Ba$

$A \rightarrow da \mid BC$

$B \rightarrow g \mid \varepsilon$

$C \rightarrow h \mid \varepsilon$

4. . $S \rightarrow AaAb \mid BbBa$

$A \rightarrow \varepsilon$

$B \rightarrow \varepsilon$

FOLLOW computation

- If the grammar is **ε-free** then **FIRST** symbols are used in selecting the appropriate production for some Non-terminal and these gets added to Parsing table
- But when the grammar is **not ε-free**, the FIRST symbols cannot be used to decide the appropriate productions, as these are not added to parsing table. i.e If there is production **$A \rightarrow \varepsilon$** in the grammar then when **A** is replaced by ε cannot be decided by the FIRST symbols and hence additional information is required to decide when **$A \rightarrow \varepsilon$** is to be used so that it can be added in the table. Here we need **FOLLOW** symbols to take the decision.
- FOLLOW(A) : It is defined to be the set of terminals 'a' that can appear immediately to the right of A in some sentential form

$$S \overset{*}{\Rightarrow} \alpha A a \beta$$

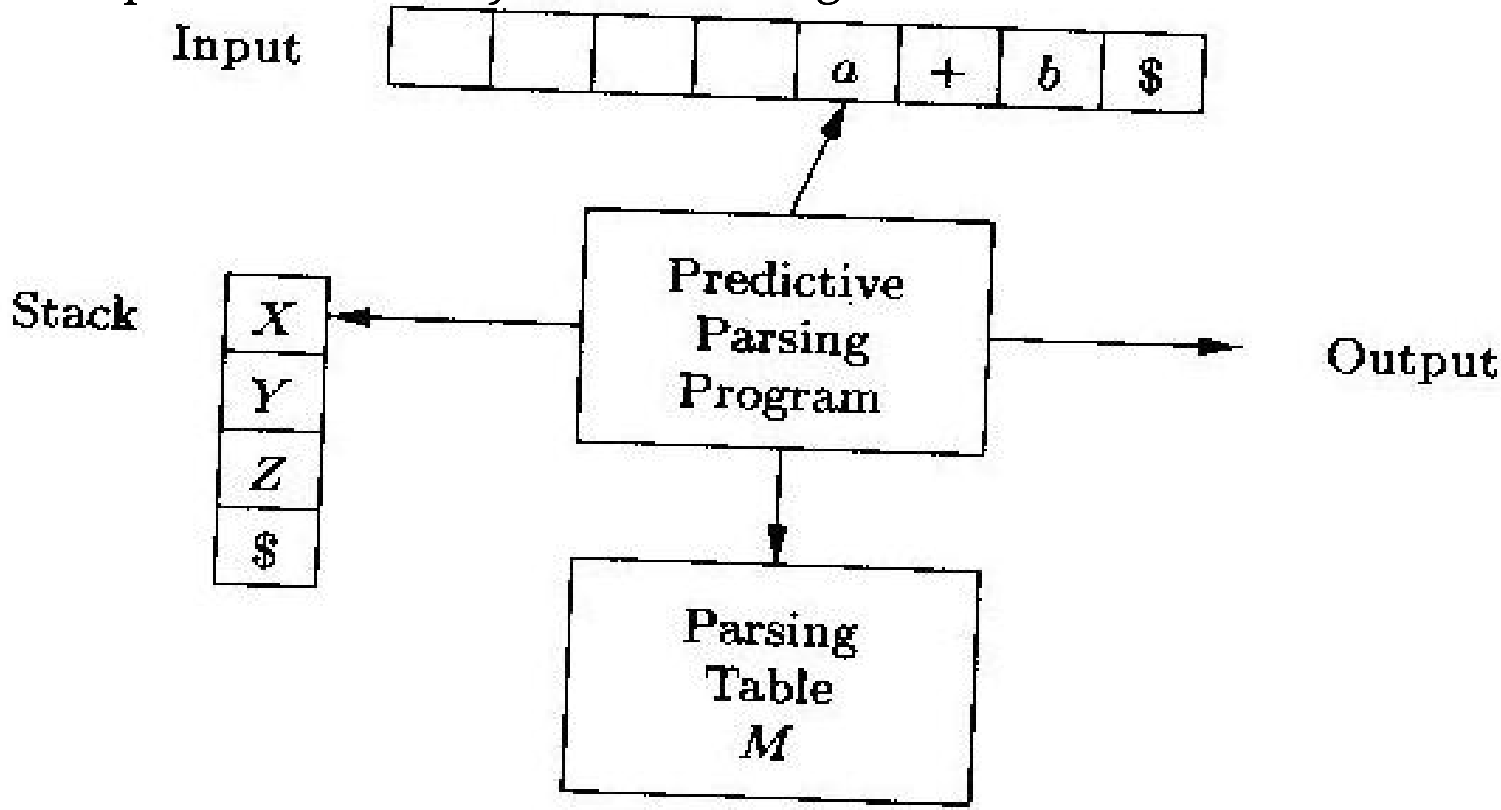
FOLLOW Set Computation

- To Compute FOLLOW(**A**) for all Non-terminals, apply the following rules until nothing can be added to any follow Set

1. Place **$** in FOLLOW(**S**), where **S** is the start symbol **$** is the input right end-marker.
2. If there is a production $A \rightarrow \alpha \mathbf{B} \beta$ then everything in FIRST(**β**) except **ε** is in FOLLOW(**B**).
3. If there is production $A \rightarrow \alpha \mathbf{B}$ **or** a production $A \rightarrow \alpha \mathbf{B} \beta$ where FIRST(β) contains **ε** then everything in FOLLOW(**A**) is FOLLOW(**B**). i.e FOLLOW(**B**) = FOLLOW(**A**)

Predictive Parser OR LL(1) parser

Predictive parser is also called **LL (1) Parser** where **First 'L'** stands for Left to right scan , **Second 'L'** stands for Leftmost derivation, which it tries to derive and '**1**' stands for use of one input symbol look-ahead at each step. It is a type of Top-down parser and implements recursive descent parser efficiently without using recursion.



Unit-2: Syntax Analysis-1 :

Top Down Parsing : Predictive Parsing.

ALGORITHM:

Consider the Grammar- G:

For each production **A → α** do the following:

a) Find **FIRST (α)** – **call set as { S1 }** and **FOLLOW (A)** - **call set as { S2 }**

b) For all symbols in **{S1}** make entries in the table as TABLE[**A, a**] = **A → α** , where **a is S1**

c) if **ε** is in **{S1}** then make the entries in the table as TABLE**[A, b] = A → α.** where **b is S2**

E → T E’
E’ → + T E’ | **ε**
T → F T’
T’ → * F T’ | **ε**
F → (E) | id

	FIRST	FOLLOW
E	(id	) $
E’	+ , ε	) $
T	(id	+) $
T’	* , ε	+) $
F	(id	+ *) $

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

NON-TERMINAL	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	$F \rightarrow \text{id}$			$F \rightarrow (E)$		

Predictive Parser driver program

set *ip* to point to the first symbol of w;
set X to the top stack symbol;
while ($X \neq \$$) { /* stack is not empty */
    if (X is a) pop the stack and advance *ip*;
    else if (X is a terminal) *error*();
    else if ($M[X, a]$ is an error entry) *error*();
    else if ($M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
        output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        pop the stack;
        push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack, with Y_1 on top;
    }
    set X to the top stack symbol;
}

Trace of predictive parsing program

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	
	$TE'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $T \to FT'$
	$\textbf{id}\ T'E'\$$	$\textbf{id} + \textbf{id} * \textbf{id}\$$	output $F \to \textbf{id}$
$\textbf{id}$	$T'E'\$$	$+ \textbf{id} * \textbf{id}\$$	match $\textbf{id}$
$\textbf{id}$	$E'\$$	$+ \textbf{id} * \textbf{id}\$$	output $T' \to \epsilon$
$\textbf{id}$	$+\ TE'\$$	$+ \textbf{id} * \textbf{id}\$$	output $E' \to +\ TE'$
$\textbf{id}\ +$	$TE'\$$	$\textbf{id} * \textbf{id}\$$	match $+$
$\textbf{id}\ +$	$FT'E'\$$	$\textbf{id} * \textbf{id}\$$	output $T \to FT'$
$\textbf{id}\ +$	$\textbf{id}\ T'E'\$$	$\textbf{id} * \textbf{id}\$$	output $F \to \textbf{id}$
$\textbf{id} + \textbf{id}$	$T'E'\$$	$* \textbf{id}\$$	match $\textbf{id}$
$\textbf{id} + \textbf{id}$	$* FT'E'\$$	$* \textbf{id}\$$	output $T' \to * FT'$
$\textbf{id} + \textbf{id}\ *$	$FT'E'\$$	$\textbf{id}\$$	match $*$
$\textbf{id} + \textbf{id}\ *$	$\textbf{id}\ T'E'\$$	$\textbf{id}\$$	output $F \to \textbf{id}$
$\textbf{id} + \textbf{id} * \textbf{id}$	$T'E'\$$	$\$$	match $\textbf{id}$
$\textbf{id} + \textbf{id} * \textbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\textbf{id} + \textbf{id} * \textbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Unit-2: Syntax Analysis-1 :

Top Down Parsing : Predictive Parsing - Error Recovery

An Error is detected when :

- TRM on top of stack does not match with next i/p Symbol.
- TABLE[A, a] is error i.e. table entry is empty.

1. PANIC MODE OF ERROR RECOVERY:

Skipping the symbol on the i/p until a token in selected set of synchronizing tokens appears and popping the current Non-terminal from the stack

SYNC- TOKEN (A) = FOLLOW (A)

Unit-2: Syntax Analysis-1 :

Top Down Parsing : Predictive Parsing- Error Recovery

- If we add symbols in FIRST (A) to Synchronizing set of non TRM A, then it may be possible to resume parsing according to A if a symbol in FIRST (A) appears in the i/p.
- If $A \rightarrow \varepsilon$; this can be used so that some error detection may be postponed, but cannot cause error to be missed.
- If TRM cannot be matched, pop the terminal and issue an message saying that TRM was inserted and continue parsing.

Unit-2: Syntax Analysis-1 :

Top Down Parsing : Predictive Parsing- Error Recovery

NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E \rightarrow +TE'$			$E \rightarrow \epsilon$	$E \rightarrow \epsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow \text{id}$	synch	synch	$F \rightarrow (E)$	synch	synch

Unit-2: Syntax Analysis-1 :

Top Down Parsing : Predictive Parsing- Error Recovery

STACK	INPUT	REMARK
$E\ \$$	$)\ \mathbf{id} * + \mathbf{id}\ \$$	error, skip)
$E\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	id is in FIRST(E)
$TE'\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$FT'E'\ \$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$\mathbf{id}\ T'E'\$$	$\mathbf{id} * + \mathbf{id}\ \$$	
$T'E'\ \$$	$* + \mathbf{id}\ \$$	
$* FT'E'\ \$$	$* + \mathbf{id}\ \$$	
$FT'E'\ \$$	$+ \mathbf{id}\ \$$	error, $M[F, +] =$ synch
$T'E'\ \$$	$+ \mathbf{id}\ \$$	F has been popped
$E'\ \$$	$+ \mathbf{id}\ \$$	
$+ TE'\ \$$	$+ \mathbf{id}\ \$$	
$TE'\ \$$	$\mathbf{id}\ \$$	
$FT'E'\ \$$	$\mathbf{id}\ \$$	
$\mathbf{id}\ T'E'\ \$$	$\mathbf{id}\ \$$	
$T'E'\ \$$	$\$$	
$E'\ \$$	$\$$	
$\$$	$\$$	

M[A,a] is blank then i/p symbol 'a' is skipped. If the entry is 'synch' the Non- TRM (not start Non-TRM) on top of the stack is popped other wise i/p symbol is skipped . If token on top of stack does not match the i/p symbol, then pop token from the stack.

2. PHRASE-LEVEL ERROR RECOVERY:

- This is implemented by filling the blank entries in the parsing table with pointers to error routines. These routines may change, insert or delete symbols in the input or STACK and issue appropriate error messages.