Syntax Analyzer

- Syntax Analyzer creates the syntactic structure of the given source program.
- This syntactic structure is mostly a parse tree.
- Syntax Analyzer is also known as parser.
- The syntax of a programming is described by a context-free grammar (CFG). We will use BNF (Backus-Naur Form) notation in the description of CFGs.
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - If it satisfies, the parser creates the parse tree of that program.
  - Otherwise the parser gives the error messages.
- ***A context-free grammar***
  - gives a precise syntactic specification of a programming language.
  - the design of the grammar is an initial phase of the design of a compiler.
  - a grammar can be directly converted into a parser by some tools.

PARSING

In this phase, the compiler checks for the sentence statement type, the valid statement type and the valid statement as for the rules of the language. So, It works on streams of tokens.

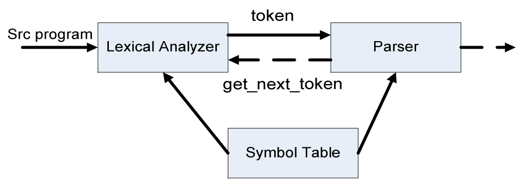


Fig: Interaction of Lexical Analyzer with Parser.

The statement types are:
- Declarative Statement
- Assignment Statement
- Conditional Statement
- Control Statement
- Procedure Call Statement

The rules of the language are specified by the grammar. They are of tow types :
- Context Free Grammar
- Context Sensitive Grammar

Mathematical Definition for Grammar:

It is defined as:

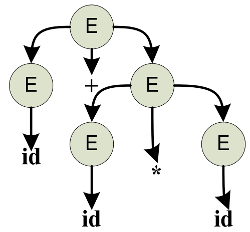
$$G = \{V, \Sigma, P, S\}$$

Where,

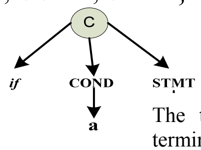
V = Set of Non-Terminal
Σ = Set of Terminal
P = Set of Production Rules
S = Starting Non-Terminal

e.g. : (i) E → E+E
E → E * E
E → **id**
Where , Σ = {+, -, **id**} N = {E} S = E

For, **id + id * id**



(ii) C → **if** COND STMT
COND → **a**
STMT → **s**
Where, Σ = **{if, a, s}** N = {C, COND, STMT} S = C



The tree terminates with terminal symbols; other are Non-Terminals (Left side of PR)

So, Parsing is a method to construct a parser tree give an input string of tokens that comes from Lexical Analysis using the predefined grammar rules or report error (if any). In case of Context Free Grammar (CFG), the left hand side of production rules contains only a single Non-Terminal. Otherwise it will be Context-Sensitive Grammar.

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
  ω is a sentence of L(G) iff $S \Rightarrow \omega$ where ω 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.
- $S \Rightarrow \alpha$ - 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:

It deals with deriving an input string starting from the starting non-terminal using production rules.

e.g. : ***id + id * id*** (Left Most Derivation)

$E \rightarrow E + E \ [E \rightarrow E+E]$
$E \rightarrow \mathbf{id} + E \ [E \rightarrow \mathbf{id}]$
$E \rightarrow \mathbf{id} + E * E \ [E \rightarrow E * E]$
$E \rightarrow \mathbf{id} + \mathbf{id} * E \ [E \rightarrow \mathbf{id}]$
$E \rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \ [E \rightarrow \mathbf{id}]$

Derivation can be :

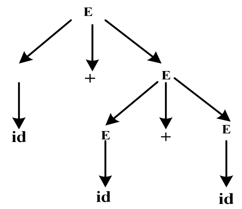
@ Left Most Derivation : The left most non-terminal is derived first.

@ Right Most Derivation : The right most non-terminal is derived frist.

e.g. : **id + id * id** (Right Most Derivation)

$E \rightarrow E + E \ [E \rightarrow E+E]$
$E \rightarrow E + E * E \ [E \rightarrow E * E\ ]$
$E \rightarrow E + E * \mathbf{id} \ [E \rightarrow \mathbf{id}]$
$E \rightarrow E + \mathbf{id} * \mathbf{id} \ [E \rightarrow \mathbf{id}]$
$E \rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \ [E \rightarrow \mathbf{id}]$

Parse Tree:



Reduction:

The process of reducing input string to the starting non-terminal using the given production rules.

e.g.:

$\mathbf{id} + \mathbf{id} * \mathbf{id}$
$\mathbf{id} + \mathbf{id} * E \ [E \rightarrow \mathbf{id}]$
$\mathbf{id} + E * E \ [E \rightarrow \mathbf{id}]$
$\mathbf{id} + E \ [E \rightarrow E * E]$
$E + E \ [E \rightarrow \mathbf{id}]$
$E \ [E \rightarrow E + E]$

Two types of Parsing Techniques:

→ Buttom – Up Parsing (Reduction Process)
→ Top – Down Parsing (Derivation Process)

Ambiguous Grammar:

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

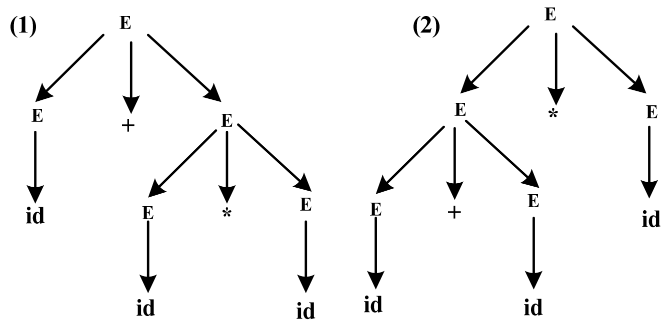


Fig. : Left Most and Right Most Derivation Tree

e. g. : Grammar for Arithmetic Expression

$E \rightarrow E\ OP\ E$
$E \rightarrow (E)$
$E \rightarrow -\ E$
$E \rightarrow \mathbf{id}$
$OP \rightarrow +\ |\ -\ |\ *\ |\ /\ |\ \%\ |\ \uparrow$

Ambiguity Removal

Given a grammar:

$E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow (E)$
$E \rightarrow \mathbf{id}$

Converted grammar without ambiguity:

$E \rightarrow E+T\ |\ T$
$T \rightarrow T*F\ |\ F$
$F \rightarrow (E)$
$F \rightarrow \mathbf{id}$

How?

We know the *multiplication* operator have higher precedence than *addition*. So, we can say that expressions are sums of one or more terms and terms are products of one or more factors.

Here, *'E+T | T'* suggests us that an expression has single term or the sum of terms. Here, we choose *'E+T'* instead of *'T+E'* because of *'+'* is left associative binary operator. Similarly, we choose *'T*F'* rather than *'F*T'*.

Notational Conventions:

1. Terminals: Lower-Case early in alphabet (a, b, c …), Operator Symbols, Parenthesis, Comma, Digits and Bold Face Strings.
2. Non-Terminals: Upper-Case early in alphabet (A, B, C…), Lower case italic names ***(expr).***

3. Grammar Symbols: Upper-Case late in alphabet (X, Y, Z) (these are either terminals or non-terminals).
4. Strings of Terminals: Lower-Case late in alphabet (x, y, z).
5. Strings of Grammar Symbols: Lower-Case Greek Letters. ($A \rightarrow \alpha$)
6. Start Symbol: Left side of first production rule.

Buttom-Up Parsing : (Reduction Process)

- A bottom-up parser creates the parse tree of the given input starting from leaves towards the root.
- A bottom-up parser tries to find the right-most derivation of the given input in the reverse order.

  $S \Rightarrow ... \Rightarrow \omega$ (the right-most derivation of ω)

  $\leftarrow$ (the bottom-up parser finds the rightmost derivation in the reverse order)
- Bottom-up parsing is also known as shift-reduce parsing because its two main actions are shift and reduce.
  - At each shift action, the current symbol in the input string is pushed to a stack.
  - At each reduction step, the symbols at the top of the stack (this symbol sequence is the right side of a production) will replaced by the non-terminal at the left side of that production.
  - There are also two more actions: accept and error.

@ Shift Reduced Parsing:

- *A shift-reduce parser tries to reduce the given input string into the starting symbol.*

a string $\xrightarrow[\text{reduced to}]{}$ the starting symbol

- *At each reduction step, a substring of the input matching to the right side of a production rule is replaced by the non-terminal at the left side of that production rule.*
- *If the substring is chosen correctly, the right most derivation of that string is created in the reverse order.*

Rightmost Derivation: $S \Rightarrow \omega$
Shift-Reduce Parser finds: $\omega \Leftarrow \ldots \Leftarrow S$

Handle

A substring is a handle which can be reduced to a single non-terminal so as to reduce the entire input string to the starting Non-Terminal.

- *Informally, a handle of a string is a substring that matches the right side of a production rule.*
  - But not every substring matches the right side of a production rule is handle
- *A handle of a right sentential form* $\gamma (\equiv \alpha\beta\omega)$ *is*
  a production rule $A \rightarrow \beta$ *and a position of* γ
  where the string β *may be found and replaced by A to produce*
  the previous right-sentential form in a rightmost derivation of γ.
  $$S \Rightarrow \alpha A \omega \Rightarrow \alpha\beta\omega$$
- *If the grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.*
- *We will see that* ω *is a string of terminals.*

Handle Pruning

It is the process of identifying and reducing the handle.

- *A right-most derivation in reverse can be obtained by handle-pruning.*
  $S=\gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = \omega$
  $\omega \rightarrow$ *input string*
- *Start from* γ_n*, find a handle* $A_n \rightarrow \beta_n$ *in* γ_n*, and replace* β_n *in by* A_n *to get* γ_{n-1}.
- *Then find a handle* $A_{n-1} \rightarrow \beta_{n-1}$ *in* γ_{n-1}*, and replace* β_{n-1} *in by* A_{n-1} *to get* γ_{n-2}.
- *Repeat this, until we reach S.*

A Shift Reduce Parser :

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

*Right-Most Derivation of id+id*id*

$E \Rightarrow E+T \Rightarrow E+T*F \Rightarrow E+T*id \Rightarrow E+F*id$
$\Rightarrow E+id*id \Rightarrow T+id*id \Rightarrow F+id*id \Rightarrow id+id*id$

Right-Most Sentential Form	*Reducing Production*
$\underline{id}+id*id$	$F \rightarrow id$
$\underline{F}+id*id$	$T \rightarrow F$
$\underline{T}+id*id$	$E \rightarrow T$
$E+\underline{id}*id$	$F \rightarrow id$
$E+\underline{F}*id$	$T \rightarrow F$
$E+T*\underline{id}$	$F \rightarrow id$
$E+\underline{T*F}$	$T \rightarrow T*F$
$\underline{E+T}$	$E \rightarrow E+T$
E	

Handles are red and underlined in the right-sentential forms.

A Stack Implementation of A Shift-Reduce Parser

- *There are four possible actions of a shift-parser action:*
  - Shift : The next input symbol is shifted onto the top of the stack.
  - Reduce: Replace the handle on the top of the stack by the non-terminal.
  - Accept: Successful completion of parsing.
  - Error: Parser discovers a syntax error, and calls an error recovery routine.
- *Initial stack just contains only the end-marker $.*
- *The end of the input string is marked by the end-marker $.*

e.g. :

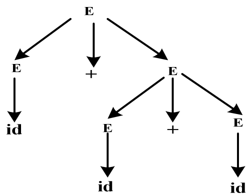
PR : $E \rightarrow E + E \mid E * E \mid$ **id**
Input String : **id + id * id**

Stack	Input String	Action
$	**id + id * id $**	Append $
$ id	**+ id * id $**	Shift
$ E	**+ id * id $**	Reduce E → id
$ E + id	*** id $**	Shift + and id
$ E + E	*** id $**	Reduce E → id
$ E + E * id	**$**	Shift * and id
$ E + E * E	**$**	Reduce E → id
$ E + E	**$**	Reduce E → E * E
$ E	**$**	Reduce E → E + E
Accept !		

Algorithm:

1. Append $ to the input string and push $ on to the stack.
2.
   a) If a substring starting from the top of the stack does not match with any of the right side of the production rules then, shift the input symbol from input string to the stack.
   b) Otherwise, 'Reduce' the substring with the left-hand side non-terminal of the matched production rule.
3. If the input string contains $ and stack contains only the starting non-terminal then Accept.
4. If no Sift, Reduce or Accept in the problem then error is reported.

Parse Tree Construction:



[This algorithm only focused on the reduction of the stack. Note: The right side of production rules must be unique.]

Conflicts during Shift-Reduce Parsing:

- *There are context-free grammars for which shift-reduce parsers cannot be used.*
- *Stack contents and the next input symbol may not decide action:*
  – shift/reduce conflict: Whether make a shift operation or a reduction.
  – reduce/reduce conflict: The parser cannot decide which of several reductions to make.
- *If a shift-reduce parser cannot be used for a grammar, that grammar is called as non-LR(k) grammar.*
- *An ambiguous grammar can never be a LR grammar.*

Operator-Precedence Parsing:

It is a bottom-up parser which does not require the grammar at the time of parsing but uses the operator precedence relations to construct the parse tree. Operator relation/ precedence are derived from the grammar only.

Operator-Precedence Parser

- **Operator grammar**

  – Small, but an important class of grammars
  – We may have an efficient operator precedence parser (a shift-reduce parser) for an operator grammar.

- In an *operator grammar*, **no** production rules can have:
  – ε at the right side
  – two adjacent non-terminals at the right side.

- *Ex:*

$E \rightarrow AB$	$E \rightarrow EOE$	$E \rightarrow E+E$ |
$A \rightarrow a$	$E \rightarrow id$	$E*E$ |
$B \rightarrow b$	$O \rightarrow + \| * \| /$	E/E | id
not operator grammar	*not operator grammar*	*operator grammar*

Precedence Relations

- In operator-precedence parsing, we define three disjoint precedence relations between certain pairs of terminals.

  $a <\cdot b$ — b has higher precedence than a
  $a =\cdot b$ — b has same precedence as a
  $a \cdot> b$ — b has lower precedence than a

- The determination of correct precedence relations between terminals are based on the traditional notions of associativity and precedence of operators. (Unary minus causes a problem).

Using Operator-Precedence Relations

- The intention of the precedence relations is to find the handle of a right-sentential form,
  <· with marking the left end,
  =· appearing in the interior of the handle, and
  ·> marking the right hand.

- In our input string $\$a_1a_2...a_n\$$, we insert the precedence relation between the pairs of terminals (the precedence relation holds between the terminals in that pair).

Using Operator -Precedence Relations

OPG: It is an operator grammar in which the precedence relationship between terminal symbols are disjoint.

Finding Precedence Relationship:

(I). Equal Precedence ($\doteq$):- If there is a production rule of type $A \rightarrow \alpha a \beta b \gamma$ where β is ε or a single non-terminal then $a \doteq b$.

e.g.: Conditional Statement:

CS → *if* **COND** *then* **STMT** *else* **STMT**
STMT → *s*
COND → *c*

(II). Less Precedence (<.) :- For all production rules of type $A \rightarrow \alpha a B \beta$
i.e. a terminal followed by any non-terminal symbol,
(1) Establish a <· all Leading symbols of B.
(2) Establish $\$$ <· Leading symbols of S.

e.g. $E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F \quad \Rightarrow + <\cdot *$
$F \rightarrow (E) \mid \textbf{id} \quad \Rightarrow * <\cdot (, * <\cdot \textbf{id}$
$\Rightarrow (<\cdot +, (<\cdot *, (<\cdot (, (<\cdot \textbf{id}$

(III) Greater Precedence (·>):- For all production rules of type $A \rightarrow \alpha A b \beta$
i.e. a terminal symbol following a non-terminal symbol,
(1). Establish *Trailing (A)* ·> b.
(2). Establish *Trailing (S)* ·> **$**.

Example:

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

The partial operator-precedence table for this grammar

	id	+	*	$
id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

- Then the input string ***id+id*id*** with the precedence relations inserted will be:
  $\$ <\cdot id \cdot> + <\cdot id \cdot> * <\cdot id \cdot> \$$

To Find the Handles:

- Scan the string from left end until the first ·> is encountered.
- Then scan backwards (to the left) over any =· until a <· is encountered.
- The handle contains everything to left of the first ·> and to the right of the <· is encountered.

$\$ \lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$	$E \rightarrow id$	$\$ id + id * id \$$
$\$ \lessdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$	$E \rightarrow id$	$\$ E + id * id \$$
$\$ \lessdot + \lessdot * \lessdot id \gtrdot \$$	$E \rightarrow id$	$\$ E + E * id \$$
$\$ \lessdot + \lessdot * \gtrdot \$$	$E \rightarrow E*E$	$\$ E + E * E \$$
$\$ \lessdot + \gtrdot \$$	$E \rightarrow E+E$	$\$ E + E \$$
$\$ \$$		$\$ E \$$

Operator-Precedence Parsing Algorithm

- The input string is *w$*, the initial stack is *$* and a table holds precedence relations between certain terminals

Algorithm:

```
set p to point to the first symbol of w$ ;
repeat forever
    if($ is on top of the stack and p points to $ ) then
return
    else{
        let a be the topmost terminal symbol on the stack and
let b be the symbol pointed to by p;
        if (a ⋖ b  or  a ≐ b  ) then {     /* SHIFT */
            push b onto the stack;
            advance p to the next input symbol;
        }
        else if ( a ⋗ b) then         /* REDUCE */
            repeat pop stack
            until ( the top of stack terminal is related by ⋖
           to the terminal most recently popped );
        else  error();
    }
```


Operator-Precedence Parsing Algorithm – Example

stack	*input*	*action*	
$	id+id*id$	$\$ \lessdot id$	shift
$id	+id*id$	$id \gtrdot +$	reduce $E \rightarrow id$
$	+id*id$	shift	
$+	id*id$	shift	
$+id	*id$	$id \gtrdot *$	reduce $E \rightarrow id$
$+	*id$	shift	
$+*	id$	shift	
$+*id	$	$id \gtrdot \$$	reduce $E \rightarrow id$
$+*	$	$* \gtrdot \$$	reduce $E \rightarrow E*E$
$+	$	$+ \gtrdot \$$	reduce $E \rightarrow E+E$
$	$	accept	

How to Create Operator-Precedence Relations

- We use associativity and precedence relations among operators.

- If operator O_1 has higher precedence than operator O_2,
  ➔ $O_1 \gtrdot O_2$ and $O_2 \lessdot O_1$

- If operator O_1 and operator O_2 have equal precedence,
  they are left-associative ➔ $O_1 \gtrdot O_2$ and $O_2 \gtrdot O_1$
  they are right-associative ➔ $O_1 \lessdot O_2$ and $O_2 \lessdot O_1$

- For all operators O,
  $O \lessdot id, \quad id \gtrdot O, \quad O \lessdot (, \quad (\lessdot O, \quad O \gtrdot), \quad) \gtrdot O, \quad O \gtrdot \$, \text{ and } \quad \$ \lessdot O$

- Also, let

$(\doteq)$	$\$ \lessdot ($	$id \gtrdot)$	$) \gtrdot \$$
$(\lessdot ($	$\$ \lessdot id$	$id \gtrdot \$$	$) \gtrdot)$
$(\lessdot id$			

Handling Unary Minus

- Operator-Precedence parsing cannot handle the unary minus when we also use the binary minus in our grammar.
- The best approach to solve this problem let the lexical analyzer handle this problem.
  - The lexical analyzer will return two different operators for the unary minus and the binary minus.
  - The lexical analyzer will need a lookahead to distinguish the binary minus from the unary minus.
- Then, we make

  $O \lessdot$ unary-minus for any operator

  unary-minus $\gtrdot O$ if unary-minus has higher precedence than O

  unary-minus $\lessdot O$ if unary-minus has lower (or equal) precedence than O

- **Disadvantages:**
  - It cannot handle the unary minus (the lexical analyzer should handle the unary minus).
  - Small class of grammars.
  - Difficult to decide which language is recognized by the grammar.

- **Advantages:**
  - Simple
  - Powerful enough for expressions in programming languages

Top-Down Parsing

It can be view as constructing parse tree for any input string starting from root and creating nodes of the parse tree in preorder and attempts to find the leftmost derivation.

Top-Down parser with back tracking

Example: Construct a parse tree for the following grammar for input string $\omega = cad$.

Grammar: $S \rightarrow cAd$

$A \rightarrow ab \mid a$

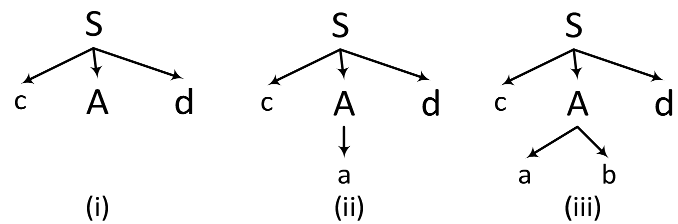
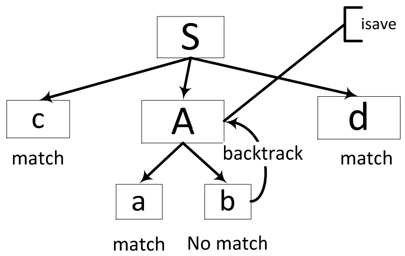


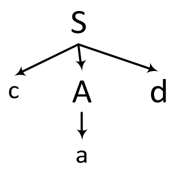
Fig. Parse tree of the given grammar.

In back tracking, if we get parse tree given in (iii) in advance, we need to back track to get proper parse tree for given input string ω.

To get back track, the node A has two options either to expand ab or a. So, we have to store the address of A in *isave* as b does not match with string $\omega = cad,$ so it must back track to *isave* and expand the other option i.e. $A \rightarrow a$.



Finally, we get the right derivation shown below.



Drawbacks of Top-Down Parser with backtracking

1. *Left recursion*: A production $A \to A\alpha$ may cause a parser to enter into an infinite loop.
2. *Backtracking*: A backtracking in semantic actions requires substantial computation overhead such as rebuilding symbol table.
3. *Choosing right production*: If an alternative is tried, it can affect the language accepted by the grammar. For example, during derivation of ω = *cabd*, rejects ω = *cad* which is also accepted by the same grammar.
4. *Difficulty to locate error point*: This parser simply reports failure but it does not locate where it is occurred.

<u>Top Down Parser without backtracking</u>

To overcome from the limitation of backtracking, we need to construct a top-down parser without backtracking through the following conversion in the given grammar.

<u>Elimination of Left Recursion</u>

If we have productions of the form $A \to A\alpha|\beta$, where β does not start with A, then left recursion can be eliminated by rewriting the productions as

$$A \to \beta A'$$
$$A' \to \alpha A' | \varepsilon$$

Example: Eliminate the left recursion of the following grammar

$$E \to E+T \mid T$$
$$T \to T*F \mid F$$
$$F \to (E) \mid \textbf{id}$$

Applying rules:

1. $E \to E+T \mid T$
   Comparing with $A \to A\alpha|\beta$, we get $A = E$, $\alpha = +T$ & $\beta = T$. So we get
   $E \to TE'$ and
   $E' \to +TE' | \varepsilon$.

2. Similarly, $T \to T*F \mid F$ is converted to:
   $T \to FT'$ and
   $T' \to *FT' | \varepsilon$.

3. $F \to (E) \mid \textbf{id}$ is not of the form $A \to A\alpha$.

Final result without left recursion:

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

<u>Left Factoring</u>

If we have production of the form $A \to \alpha\beta \mid \alpha\gamma$ and input begins with non-empty string α, we need to eliminate common left factor as shown below.

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

Example: Conditional statement:

$CS \rightarrow if\ COND\ then\ STMT$
$CS \rightarrow if\ COND\ then\ STMT\ else\ STMT$

After Conversion:

$CS \rightarrow if\ COND\ then\ STMT\ STMT'$
$STMT' \rightarrow \varepsilon \mid else\ STMT.$

The grammar with no left recursion and no left factor is called *LL(1)* grammar.

Recursive Descent Parsing

A parser that uses a set of recursive procedures to recognize its input with no backtracking is called *recursive descent parser*. Procedures shown below can accept the language understand by the grammar without left recursion and no left factor.

Grammar:

$E \rightarrow TE'$
$E' \rightarrow +TE' | \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \varepsilon$
$F \rightarrow (E) | \mathbf{id}$

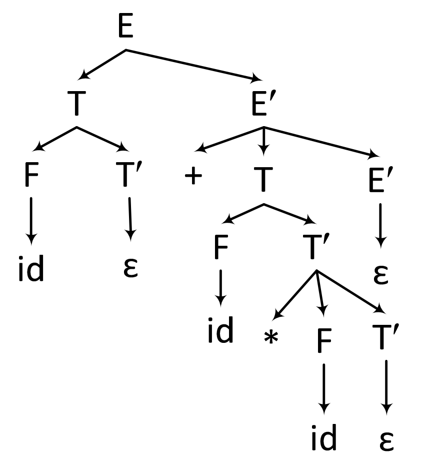
Procedures:

```
E() {
      T();
      EPRIME();
      "Success"
}
T() {
      F();
      TPRIME();
}
```



```
EPRIME() {
      If next_input_symbol = '+' {
            Advance();
            T();
            EPRIME();
      }
}
TPRIME() {
      If next_input_symbol = '*' {
            Advance();
            F();
            TPRIME();
      }
}
F() {
      If next_input_symbol = 'id' {
            Advance();
      } Else If next_input_symbol = '(' {
            Advance();
            E();
            If next_input_symbol ≠ ')'{
                  Error();
            }
      } Else {
            Error();
      }
}
```

Example: ***id + id * id***



Non Recursive Predictive Parsing

It is a recursive descent parser which uses stack instead of recursive calls. A predictive parser has a stack, an input buffer, parsing table, program for parsing and an output stream.

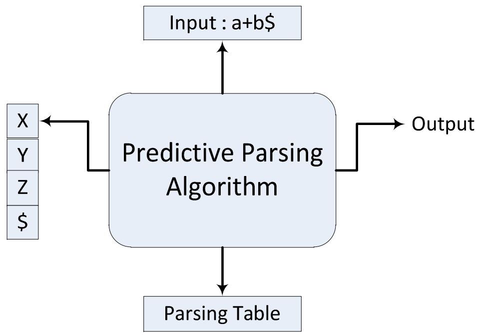


Fig. Model for predictive parser

In predictive parsing, it is required to construct a parsing table and a parsing program using their algorithm.

First and Follow

The *First* set of a string of symbol is the set of tokens that may appear when the string is expanded.

The *Follow* set of Non-Terminal is that set of tokens that can immediately follow that NT in some syntactic form.

These First and Follow allow us to fill in the entries of a predictive parsing table for the given grammar whenever possible.

First (α)

If α is any string of grammar symbols, let *First* (α) be the set of terminals that begin the strings derived from α. If $\alpha \Rightarrow \varepsilon$ then ε is also in *First* (α).
To compute *First(X)* for all grammar symbols *X*, apply the following rules until no more terminals or ε can be added to any *First* set:

1. If X is terminal, then *First(X)* is $\{X\}$.
2. If $X \rightarrow \varepsilon$ is a production, then add ε to *First(X)*.
3. If X is NT and $X \rightarrow Y_1Y_2\ldots Y_k$ is a production, then place a in *First(X)* if for some i, a is in $First(Y_i)$, and ε is in all of $First(Y_1),\ldots,First(Y_{i-1})$; i.e., $Y_1,\ldots,Y_{i-1} \Rightarrow \varepsilon$. If ε is in $First(Y_j)$ for all $j=1,2,\ldots,k$, then add ε to *First(X)*.

   For example: Everything in $First(Y_1)$ is surely in *First(X)*. If Y_1 does not derive ε, then we add no more thing to *First(X)*, but if $Y_1 \Rightarrow \varepsilon$, then we add $First(Y_2)$ and so on.

Now, we can compute $First(X_1X_2\ldots X_n)$ as follows. Add to $First(X_1X_2\ldots X_n)$ all the non-empty symbols of $First(X_1)$. Also add then non-empty symbols of $First(X_2)$ if ε is in $First(X_1)$, the non-empty symbols of $First(X_3)$ if ε is in both $First(X_1)$ and $First(X_2)$, and so on. Finally, add ε to $First(X_1X_2\ldots X_n)$ if, for all i, $First(X_i)$ contains ε.

Follow (A)

Follow(A), i.e. the set of terminals *a,* such that there exists a derivation of the form $S \rightarrow \alpha A a \beta$ for some α and β. Note that, during derivation there may be some symbols between A and a, but if so, they will derive ε and will disappear. If A is the rightmost symbol in some sentential form, then $\$$ is in *Follow(A)*.

To compute *Follow(A)* for all NTs A, apply the following rules until nothing can be added to any *Follow* set:

1. Place $\$$ in *Follow(S)*, where S is the start symbol and $\$$ is the input right end-marker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in *First(β)*, except for ε, is placed in *Follow(B)*.
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where *First(β)* contains ε (i.e. $\beta \rightarrow \varepsilon$), then everything in *Follow(A)* is in *Follow(B)*.

Example:
Consider the grammar given below.

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' | \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' | \varepsilon$$
$$F \rightarrow (E) | \mathbf{id}$$

Computing First:

$First(E) = First(T) = First(F) = \{(,\mathbf{id}\}$
$First(E') = \{+,\varepsilon\}$
$First(T') = \{+,\varepsilon\}$

Computing Follow:

1. $E \rightarrow TE'$ So, $Follow(E) = \{\$\}$ i.e. start symbol.
   Comparing with $A \rightarrow \alpha B \beta$, *First(β)* except ε are added to *Follow(B)*. i.e.
   $Follow(T) = \{First(E') - \varepsilon\} = \{+\}$.
   Also, $E' \rightarrow \varepsilon$ so, $Follow(T) = Follow(E)$.
   Again, $Follow(E') = Follow(E)$

2. $E' \rightarrow +TE'$
   Comparing with $A \rightarrow \alpha B \beta$, *First(β)* except ε are added to *Follow(B)*. i.e.
   $Follow(T) = \{First(E') - \varepsilon\} = \{+\}$.

3. $T \rightarrow FT'$ gives $Follow(T') = Follow(T)$.
   Again,
   $Follow(F) = \{First(T') - \varepsilon\} = \{*\}$
   Also, $T' \rightarrow \varepsilon$, So, $Follow(F) = Follow(T)$.

4. $T \rightarrow FT'$ gives $Follow(F) = \{First(T') - \varepsilon\} = \{*\}$
5. $F \rightarrow (E)$ gives $Follow(E) = \{)\}$

Summarizing the Follow results:
$Follow(E) = \{), \$\}$
$Follow(E') = \{), \$\}$
$Follow(T) = \{+,), \$\}$
$Follow(T') = \{+,), \$\}$
$Follow(F) = \{+, *,), \$\}$

Algorithm for Predictive Parsing Table construction

1. Compare each production with $A \rightarrow \alpha$ and apply (2) & (3) for each of them.
2. Find *First(α)*. For each '*a*' in *First(α)*, add $M[A,a] = A \rightarrow \alpha$, where *M[A,a]* is an entry for parsing table of NT A and Terminal a.

3. If ε is in *First(α)* then, add $M[A,b] = A \rightarrow \alpha$, for each *b* in *Follow(A)*.
   If ε is in *First(α)* and *$* is in *Follow(A)* then add $M[A,\$] = A \rightarrow \alpha$.
4. Make error in each undefined entry of M.

Parsing table for the grammar

T→ NT↓	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E′		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow F\,T'$			$T \rightarrow F\,T'$		
T′		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Algorithm for Predictive Parsing

```
Repeat
        Begin
                Make X to be the top of stack and 'a' is next i/p symbol.
                If X is a terminal or $ then
                        If X = a then
                                POP X from stack & remove 'a' from i/p.
                        Else
                                Error().
                Else /* i.e. X is NT */
                        If M[X,a] = X → Y1Y2...Yk then
                                Begin
                                        POP X from the stack.
```



```
                                        PUSH YkYk-1...Y1 onto the stack.
                                End
                        Else
                                Error().
        End
Until X = $. /* i.e. stack empty */
```

Example: Predictive Parser

Stack	Input (a)	Output
$E	id + id * id $	
$E′T	id + id * id $	E → TE′ from M[E, id].
$E′T′F	id + id * id $	T → FT′ from M[T, id].
$E′T′id	**id** + id * id $	F → id from M[F, id].
$E′T′	+ id * id $	Matches 'id' and remove.
$E′	+ id * id $	T′ → ε from M[T′, +].
$ E′T+	+ id * id $	E′ → +TE′ from M[E′, +].
$ E′T	id * id $	Matches '+' and remove.
$ E′T′F	id * id $	T → FT′ from M[T, id].
$E′T′id	**id** * id $	F → id from M[F, id].
$E′T′	* id $	Matches 'id' and remove.
$E′T′F*	* id $	T′ → *FT′ from M[T′, *].
$E′T′F	id $	Matches '*' and remove.
$E′T′id	**id** $	F → id from M[F, id].
$E′T′	**$**	Matches 'id' and remove.
$E′	**$**	T′ → ε from M[T′, $].
$	**$**	E′ → ε from M[E′, $].

So, $ matches and the given i/p is accepted.