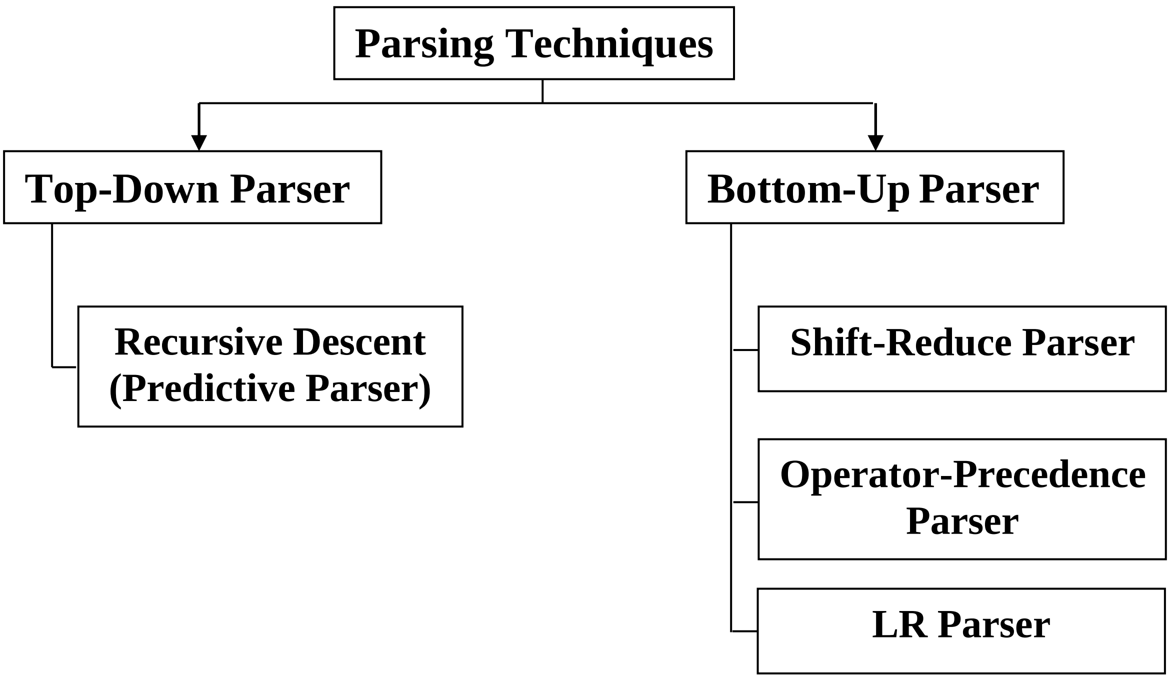
# Parsing Techniques

## Parsers

A ***parser*** for grammar ***G*** is a program that takes as input a string ***w*** and produces as output either a parse tree for ***w***, if ***w*** is a sentence of ***G***, or an error message indicating that ***w*** is not a sentence of ***G***.

There are two basic types of **parsers** for context-free grammars are **Top-Down** and **Bottom-Up**. As indicated by their names, top-down parsers start with the root and work down to the leaves, while bottom-up parsers build parse trees from the bottom (leaves) to the top (root). In both cases the input to the parser is being scanned from ***left to right***, one symbol at a time.




## Top-Down Parsing

Top-down parsing can be viewed as an attempt to find a leftmost derivation for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

## Recursive-Descent Parsing

The general form of top-down parsing, called recursive descent, the recursive descent can be divided to two cases. First case that may involve **Backtracking**, which is, making repeated scans of the input and second case, is **No Backtracking** (***Predictive Parser***).

### Backtracking

Backtracking is required in the next example, and will be keeping track of the input when backtracking takes place.

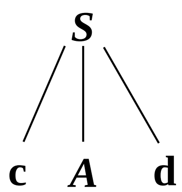
**Example:** Consider the grammar

$$S \longrightarrow cAd$$

$$A \longrightarrow ab \mid a$$

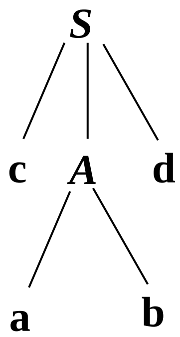
and the input string ***w*** = **cad**. To construct a parse tree for this string top-down.

1) Create a tree consisting of a single node labeled ***S***.
2) An input pointer points to **c**, the first symbol of ***w***. We then use the first production for ***S*** to expand the tree and obtain the tree of Figure below.



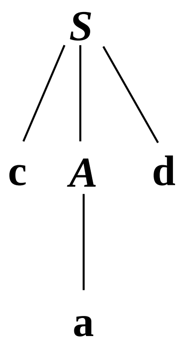

The leftmost leaf, labeled **c**, matches the first symbol of ***w***.

3) Advance the input pointer to **a**, the second symbol of ***w***, and consider the next leaf, labeled ***A***. We can then expand ***A*** using the first alternative for ***A*** to obtain the tree of Figure below. We now have a match for the second input symbol.




4) Advance the input pointer to **d**, the third input symbol, and compare **d** against the next leaf, labeled **b**. Since **b** does not match **d**, we report failure and **go back to *A*** to see whether there is another alternative for ***A*** that we have not tried but that might produce a match.

5) In going back to ***A***, we must reset the input pointer to position **2**, the position it had when we first came to ***A***, we now try the second alternative for ***A*** to obtain the tree of figure below.




The leaf **a** matches the second symbol of ***w*** and the leaf **d** matches the third symbol. Since we have produced a parse tree for ***w***, we halt and announce successful completion of parsing.

**<u>Note:</u>** A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop. That is, when we try to expand ***A***, we may eventually find ourselves again trying to expand ***A*** without having consumed any input.

## Predictive Parser

In many cases, by carefully writing a grammar, ***eliminating left recursion*** from it, and ***left factoring*** the resulting grammar, we can obtain a grammar that can be parsed by a **recursive-descent parser** that needs **no backtracking**, i.e., a **predictive parser**.

### ☒ Transition Diagrams for Predictive Parser

We can create a transition diagram as a plan for a predictive parser. Several differences between the transition diagrams for a lexical analyzer and a predictive parser are immediately apparent. In the case of the parser, there is one diagram for each **nonterminal**. The labels of edges are **tokens** (**terminal**) and **nonterminals**. A transition on a token (**terminal**) means we should take that transition if that token is the next input symbol. A transition on a nonterminal, ***A*** is a call of the procedure for ***A***.

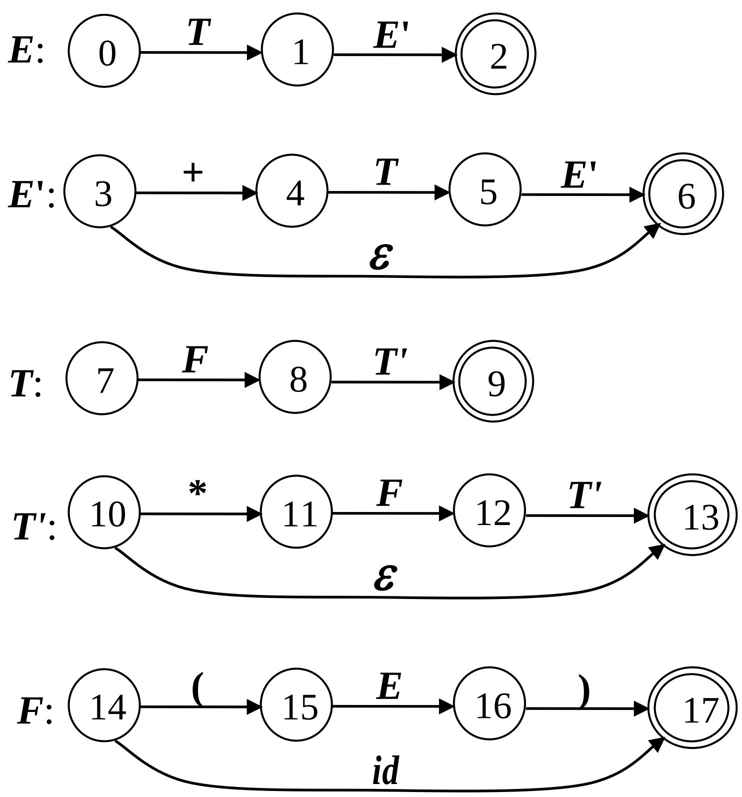
To construct the transition diagram of a predictive parser from a grammar, first **eliminate left recursion** from the grammar, and then **left factor** the grammar. Then for each **nonterminal *A*** do the following:

1) Create an **initial** and **final** (return) state.
2) For each production $A \longrightarrow X_1X_2 \ldots X_n$, create a **path** from the **initial** to the **final** state, with edges labeled $X_1X_2 \ldots X_n$.

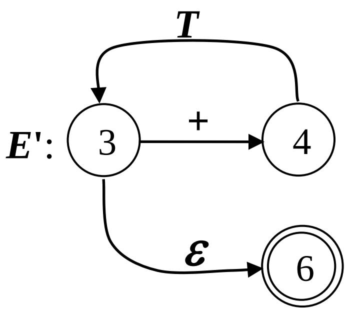
The predictive parser working off the transition diagrams behaves as follows. It begins in the **start** state for the **start symbol**. If after some actions it is in state ***s*** with an edge labeled by terminal ***a*** to state ***t***, and if the next input symbol is ***a***, then the parser moves the input cursor one position right and goes to state ***t***. If, on the other hand, the edge is labeled by a nonterminal ***A***, the parser instead goes to the **start** state for ***A***, without moving the input cursor. If it ever reaches the **final** state for ***A***, it immediately goes to state ***t***, in effect having "read" ***A*** from the input during the time it moved from state **s** to ***t***. **Finally**, if there is an edge from ***s*** to ***t*** labeled $\varepsilon$ , then from state ***s*** the parser immediately goes to state ***t***, without advancing the input.

**Example:** Design the transition diagram of predictive parser for the following grammar:

$E \longrightarrow TE'$
$E' \longrightarrow +TE' \mid \varepsilon$
$T \longrightarrow FT'$
$T' \longrightarrow *FT' \mid \varepsilon$
$F \longrightarrow (E) \mid id$



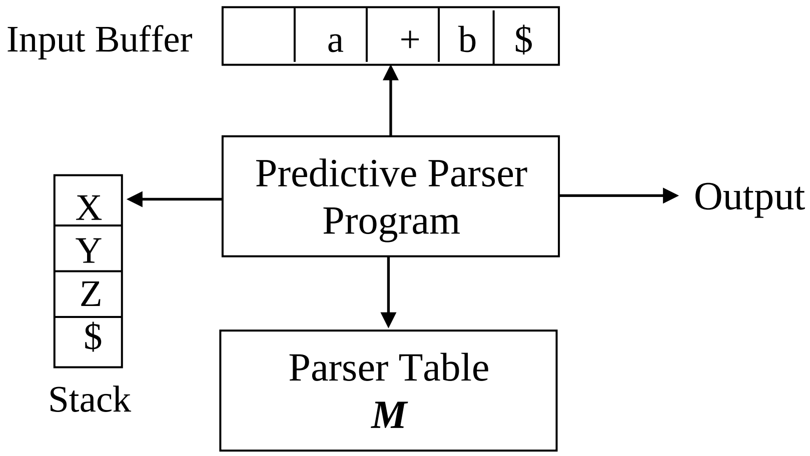
The figure in below shows an equivalent transition diagram for ***E'***.




**Simplified Transition diagram**

## ☒ Components of Predictive Parser

Predictive parser has an input buffer, a stack, a parsing table, and an output stream. The model of predictive parser is shown the in figure below:




Model of Predictive Parser

1) The input buffer contains the string to be parsed, followed by **$**, a symbol used as a right endmarker to indicate the end of the input string.

2) The stack contains a sequence of grammar symbols with **$** on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of **$**.
3) The parsing table is a **two-dimensional array *M [A, a]***, where ***A*** is a nonterminal, and ***a*** is a terminal or the symbol **$**.

The parser is controlled by a program that behaves as follows. The program considers ***X***, the symbol on top of the stack, and ***a***, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

**a)** If ***X*** **= a = $**, the parser halts and announces successful completion of parsing.

**b)** If ***X*** **= a ≠ $**, the parser pops ***X*** off the stack and advances the input pointer to the next input symbol.

**c)** If ***X*** is a nonterminal, the program consults entry ***M[X, a]*** of the parsing table ***M***. This entry will be either an ***X***-production of the grammar or an error entry. If, for example, ***M[X, a]*** = {***X*** $\longrightarrow$ ***UVW***}, the parser replaces ***X*** on top of the stack by ***WVU*** (with ***U*** on top).

4) As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X, a] = error, the parser calls error recovery routine.

| NONTER- | INPUT SYMBOL |
|---|---|

## ☒ Construction of Predictive Parsing Tables

The following algorithm can be used to construct a predictive parsing table for a grammar ***G***.

**Algorithm: Construction of a predictive parsing table.**

***Input***: Grammar ***G***.

***Output***: Parsing table ***M***.

***Method:***

1. For each production $A \longrightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal ***a*** in FIRST ($\alpha$), add $A \longrightarrow \alpha$ to ***M [A, a]***.
3. If $\varepsilon$ is in FIRST ($\alpha$), add $A \longrightarrow \alpha$ to ***M [A, b]*** for each terminal **b** in FOLLOW (***A***). If $\varepsilon$ is in FIRST ($\alpha$) and \$ is in FOLLOW (***A***), add $A \longrightarrow \alpha$ to ***M [A, \$]***.
4. Make each undefined entry of ***M*** be error.

**Example:** Parse the string **id + id * id** by using predictive parser for the following grammar:

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

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

FIRST (***E'***) = {+, $\varepsilon$ }

FIRST (***T'***) = {*, $\varepsilon$ }

FOLLOW (***E***) = FOLLOW (***E'***) = {), \$}

FOLLOW (***T***) = FOLLOW (***T'***) = {+,), \$}

FOLLOW (***F***) = {*, +,), \$}

| | id | + | * | ( | ) | $ |
|---|---|---|---|---|---|---|
| **$E$** | $E \longrightarrow TE'$ | | | $E \longrightarrow TE'$ | | |
| **$E'$** | | $E' \longrightarrow +TE'$ | | | $E' \longrightarrow \varepsilon$ | $E' \longrightarrow \varepsilon$ |
| **$T$** | $T \longrightarrow FT'$ | | | $T \longrightarrow FT'$ | | |
| **$T'$** | | $T' \longrightarrow \varepsilon$ | $T' \longrightarrow *FT'$ | | $T' \longrightarrow \varepsilon$ | $T' \longrightarrow \varepsilon$ |
| **$F$** | $F \longrightarrow \text{id}$ | | | $F \longrightarrow (E)$ | | |

**Predictive Parsing Table *M* For Above Grammar**

Blanks are error entries; non-blanks indicate a production with which to expand the top nonterminal on the stack.

| Stack | Input | Output |
|---|---|---|
| $ $E$ | **id + id * id$** | |
| $ $E'T$ | **id + id * id$** | $E \longrightarrow TE'$ |
| $ $E'T'F$ | **id + id * id$** | $T \longrightarrow FT'$ |
| $ $E'T'$ **id** | **id + id * id$** | $F \longrightarrow \text{id}$ |
| $ $E'T'$ | **+ id * id$** | |
| $ $E'$ | **+ id * id$** | $T' \longrightarrow \varepsilon$ |
| $ $E'T+$ | **+ id * id$** | $E' \longrightarrow +TE'$ |
| $ $E'T$ | **id * id$** | |
| $ $E'T'F$ | **id * id$** | $T \longrightarrow FT'$ |
| $ $E'T'$ **id** | **id * id$** | $F \longrightarrow \text{id}$ |
| $ $E'T'$ | *** id$** | |
| $ $E'T'F*$ | *** id$** | $T' \longrightarrow *FT'$ |
| $ $E'T'F$ | **id$** | |
| $ $E'T'$ **id** | **id$** | $F \longrightarrow \text{id}$ |
| $ $E'T'$ | $ | |
| $ $E'$ | $ | $T' \longrightarrow \varepsilon$ |
| $ | $ | $E' \longrightarrow \varepsilon$ |

Moves made by predictive parser on input **id + id * id**

## LL (1) Grammars

Algorithm construction of a predictive parsing table can be applied to any grammar ***G*** to produce a parsing table ***M***. For some grammars, however, ***M*** may have some entries that are **multiply-defined**. for example, if ***G*** is **left recursive** or **ambiguous**, then ***M*** will have at least one **multiply-defined** entry.

**Example:** Let us consider the following grammar:

$S \longrightarrow iEtSS' \mid a$
$S' \longrightarrow eS \mid \varepsilon$
$E \longrightarrow b$

FIRST ($S$) = {**i, a**}

FIRST ($S'$) = {**e,** $\varepsilon$ }

FIRST ($E$) = {**b**}

FOLLOW ($S$) = {**e**, $}

FOLLOW ($S'$) = {**e**, $}

FOLLOW ($E$) = {t}

| NONTER-MINALS | INPUT SYMBOL | | | | | |
|---|---|---|---|---|---|---|
| | a | b | e | i | t | $ |
| $S$ | $S \longrightarrow a$ | | | $S \longrightarrow iEtSS'$ | | |
| $S'$ | | | $S' \longrightarrow \varepsilon$<br>$S' \longrightarrow eS$ | | | $S' \longrightarrow \varepsilon$ |
| $E$ | | $E \longrightarrow b$ | | | | |

The entry for ***M[S',e]*** contains both $S' \longrightarrow eS$ and $S' \longrightarrow \varepsilon$ , since FOLLOW($S'$) = {**e, $**}. The grammar is **ambiguous** and the ambiguity is manifested by a choice in what production to use when an **e** is seen. **Therefore** this grammar is **not** LL (1).

**Definition of LL (1):**

**A grammar whose parsing table has <u>no multiply-defined</u> entries is said to be LL (1).** The **first** "L" in LL(1) stands for scanning the input from left to right, the **second** "L" for producing a leftmost derivation, and the "**1**" for using one input symbol of lookahead at each step to make parsing action decisions. LL (1) grammars have several distinctive properties. **No ambiguous** or **left- recursive** grammar can be LL (1).

**<u>The grammar is LL (1) if satisfy the following Conditions :</u>**

For all productions $A \longrightarrow \alpha_1 \mid \alpha_2 \mid \ldots\ldots \mid \alpha_n$

1. FIRST ($\alpha_i$) ∩ FIRST ($\alpha_j$) = $\phi$ for all $i \neq j$ and
2. If $\alpha_i \overset{*}{\longrightarrow} \varepsilon$ , Then FIRST ($\alpha_j$) ∩ FOLLOW($A$)= $\phi$ for all $i \neq j$

**<u>Example: Is the following grammar LL (1)?</u>**

$A \longrightarrow$ **i*B*te**
$B \longrightarrow SB \mid \varepsilon$
$S \longrightarrow$ **[ec] | •i**

***Sol:***
**<u>Rule 1:</u>**

$B \longrightarrow SB \mid \varepsilon$
FIRST ($SB$) ∩ FIRST ( $\varepsilon$ ) = {[, •} ∩ { $\varepsilon$ } = $\phi$

$S \longrightarrow$ **[ec] | •i**
FIRST (**[ec]**) ∩ FIRST (**•i**) = {[} ∩ {•} = $\phi$

**<u>Rule 2:</u>**

$B \longrightarrow SB \mid \varepsilon$
FIRST ($SB$) ∩ FOLLOW ($B$) = {[, •} ∩ {t} = $\phi$

**This grammar is LL (1).**

**<u>Example: Is the following grammar LL (1)?</u>**

$S \longrightarrow XS \mid aY$
$X \longrightarrow a \mid b$
$Y \longrightarrow (S)$

***Sol:***

**Rule 1:**

$S \longrightarrow XS \mid aY$
FIRST ($XS$) ∩ FIRST ($aY$) = {a, b} ∩ {a} = {a}

This grammar is **not LL (1)**. And it is not suitable for constructing parser table.

**Example: Is the following grammar LL (1)?**

$S \longrightarrow Aa \mid bB$
$A \longrightarrow aBmS \mid C$
$B \longrightarrow (S)$
$C \longrightarrow \varepsilon$

***Sol:***

**Rule 1:**

$S \longrightarrow Aa \mid bB$
FIRST ($Aa$) ∩ FIRST ($bB$) = {a, $\varepsilon$ } ∩ {b} = $\phi$

$A \longrightarrow aBmS \mid C$
FIRST ($aBmS$) ∩ FIRST ($C$) = {a} ∩ {$\varepsilon$} = $\phi$

**Rule 2:**

$A \longrightarrow aBmS \mid C$ Since $C \longrightarrow \varepsilon$ , Then

FIRST ($aBmS$) and FOLLOW ($A$) must be disjoint.

FIRST ($aBmS$) ∩ FOLLOW ($A$) = {a} ∩ {a} = {a}

**This grammar is not LL (1).**