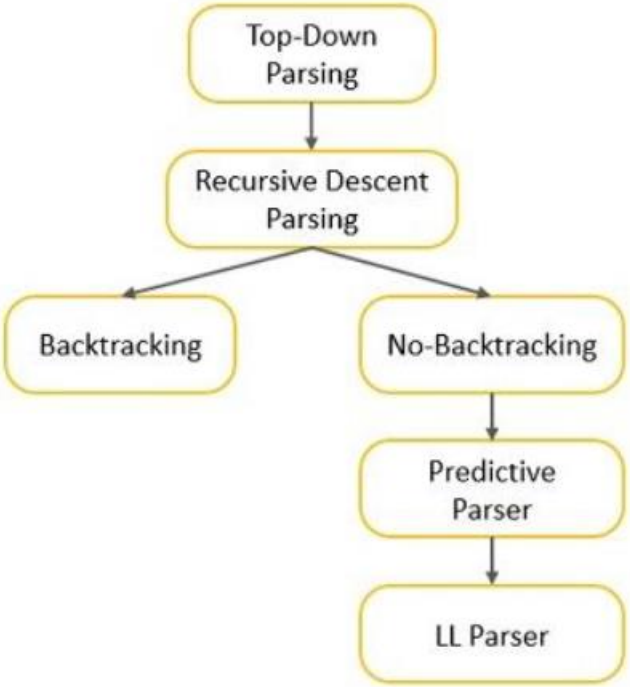
5.1 Top Down Parsing

Top-down parsing processes the **input string** provided by a lexical analyzer, and generate a parse tree for that input string using **derivation**. The top-down parsing first creates the root node of the parse tree. And it continues creating its leaf nodes.

- Remember the top-down parsing cannot handle the grammar with **left recursion** and **ambiguity.**
- At each step of a top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say A. Once an A-production is chosen, the rest of the parsing process consists of "matching7' the terminal symbols in the production body with the input string.



5.2 Recursive Descent Technique

It is a technique of top-down parsing in which the construction of a parse tree happens from the top, and input is read from left to right. It uses CFG(context-free grammar), which is why it is recursive. In this technique, procedures are used for each terminal and non-terminal entity. And, it parses the input recursively to make a parse tree with or without using back-tracking. But, if the associated grammar is not a left factor, it cannot avoid back-tracking. If this technique does not require back-tracking, it is known as predicate parsing.

5.2.1 Back-tracking

If the derivation of the input string fails in one production of a non-terminal, then the parser has to return to the place where it has picked the production. And begin deriving the string likewise using another production of the exact non-terminal. The process may require repeatedly scanning over the input string, referred to it as back-tracking.

Example

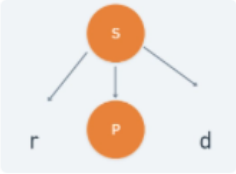
The following is a grammar rule:

S -> r P d

P -> m | m n

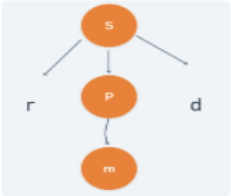
Input string: 'rmnd.'

Building the parse tree will start with the symbol S, and the input pointer is pointing to the first symbol of the given input string, 'r .'Since the start symbol 'S' has single production in its production body. So, the symbol 'S' will be expanded.



The leftmost leaf of the tree 'r' matches with the first symbol of the string given in input which is 'r .'Thus, we will extend the input pointer to the next symbol of the input string, which is 'm'.

The next leaf of the parse tree 'P' has two production rules. Let's expand the symbol 'P' with the given first production of 'P,' i.e., 'm' we get the following parse tree.



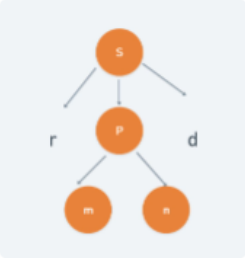
Here, the second input symbol, 'm', matches. So, we will extend the input pointer to the point next symbol of the input string, i.e., 'n.'

Here, the second input symbol, 'm', matches. So, we will extend the input pointer to the point next symbol of the input string, i.e., 'n.'

We get a mismatch when comparing the third input symbol 'n' against the next leaf labeled 'd'. Therefore, the parser must return to the symbol 'P' and look for the other alternative.

The other alternative for the production of 'P' is 'mn .'Also, the input pointer must be back-tracked and reset to the position where it was first before we expanded the symbol 'P .'It means the input pointer must point to 'm' again.

We will again expand 'P' with the following alternative production. The parse tree label 'm' leaf matches the second input symbol 'm'.



So, the input pointer is extended and points to the third input symbol, 'n .'And the next leaf of the tree labeled 'n' matches the third input symbol.

Further, the input pointer is extended to point to the fourth input symbol 'd'. And this matches the next leaf of the tree labeled 'd'.

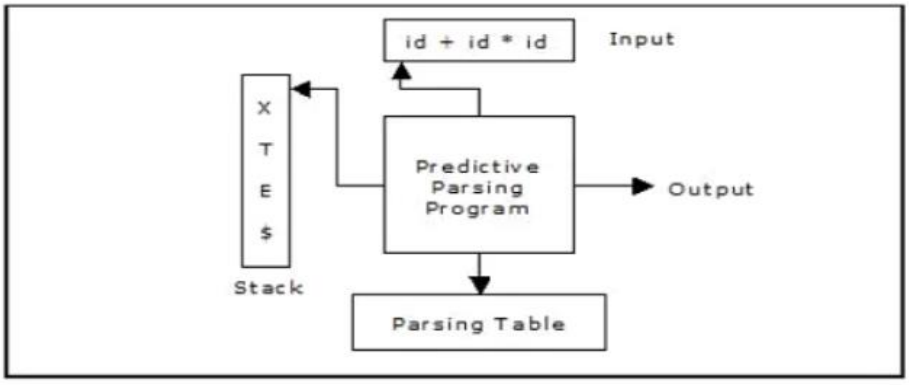
Drawbacks of Top Down Parsing with Backtracking

1. **Left recursion:** A production of the form A→Aα may cause a parset to enter into an infinite loop.
2. **Backtracking:** if we have more than one option for the same production, so backtracking is done. This means requires substantial overhead, as the entries made into symbol table may be need to be erased.
3. **Difficulty to locate error point:** In this case the compiler reports errors without describing it.

In **recursive descent parsing**, the parser may have more than one production to choose from for a single instance of input; whereas in **predictive parser**, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure to fail.

5.2.2 Predictive parsing (no backtracking)

Predictive parser is a top down parser without backtracking.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contains an end symbol $ to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.

Steps to be involved in Parsing Method:

1. Stack is pushed with $.

2. Construction of parsing table T.

- Computation of FIRST set.
- Computation of FOLLOW set.
- Making entries into the parsing table.

3. Parsing by parsing routine.

Construction of parsing table

An important part of parser table construction is to create **First** and **Follow** sets. These sets can provide the actual position of any terminal in the derivation.

Essential conditions to check first are as follows:

1. The grammar is free from left recursion.
2. The grammar should not be ambiguous.
3. The grammar has to be left factored in so that the grammar is deterministic grammar.

FIRST () − It is a function that gives the set of terminals that begin the strings derived from the production rule.

Algorithm for Calculating First Set

If X is Grammar Symbol, then First (X) will be −

- If X is a terminal symbol, then FIRST(X) = {X}
- If $X \rightarrow \varepsilon$, then FIRST(X) = {ε}
- If X is non-terminal & $X \rightarrow a\ \alpha$, then FIRST (X) = {a}
- If $X \rightarrow Y_1, Y_2, Y_3$, then FIRST (X) will be

(a) If Y is terminal, then

$FIRST(X) = FIRST(Y_1, Y_2, Y_3) = \{Y_1\}$

(b) If Y_1 is Non-terminal and

If Y_1 does not derive to an empty string i.e., If FIRST (Y_1) does not contain ε then, $FIRST(X) = FIRST(Y_1, Y_2, Y_3) = FIRST(Y_1)$

(c) If FIRST (Y_1) contains ε, then.

FIRST (X) = FIRST (Y_1, Y_2, Y_3) = FIRST(Y_1) − {ε} ∪ FIRST(Y_2, Y_3)

Similarly, FIRST (Y_2, Y_3) = {Y_2}, If Y_2 is terminal otherwise if Y_2 is Non-terminal then

- FIRST (Y_2, Y_3) = FIRST (Y_2), if FIRST (Y_2) does not contain ε.
- If FIRST (Y_2) contain ε, then
- FIRST (Y_2, Y_3) = FIRST (Y_2) − {ε} ∪ FIRST (Y_3)

Similarly, this method will be repeated for further Grammar symbols, i.e., for Y_4, Y_5, Y_6 …. Y_K.

Algorithm for Calculating Follow Set

Follow (A) is defined as the collection of terminal symbols that occur directly to the right of A.

FOLLOW(A) = {a|S ⇒* αAaβ where α, β can be any strings}

Rules to find FOLLOW

- If S is the start symbol, FOLLOW (S) ={$}
- If production is of form A → α B β, β ≠ ε.

(a) If FIRST (β) does not contain ε then, FOLLOW (B) = {FIRST (β)}

Or

(b) If FIRST (β) contains ε (i. e. , β ⇒* ε), then

FOLLOW (B) = FIRST (β) − {ε} ∪ FOLLOW (A)

∵ when β derives ε, then terminal after A will follow B.

- If production is of form A → αB, then Follow (B) ={FOLLOW (A)}.

Algorithm for Constructing Parse Table

METHOD: For each production $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

Problem − Consider the following grammar

E → TE′
E′ → +TE′|ε
T′ → FT′
T′ → FT′|ε
F → (E)|id

Solution −

Step1− Elimination of Left Recursion & perform Left Factoring

As there is no left recursion in Grammar so, we will proceed as it is. Also, there is no need for Left Factoring.

Step2− Computation of FIRST and Follow

	First	Follow
E -> TE'	{ id, (}	{ $,) }
E' -> +TE'/ε	{ +, ε }	{ $,) }
T -> FT'	{ id, (}	{ +, $,) }
T' -> *FT'/ε	{ *, ε }	{ +, $,) }
F -> id/(E)	{ id, (}	{ *, +, $,) }

Step3- Make a parser 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 –> id			F –> (E)		

Blanks are error entries; non blanks indicate a production with which to expand a nonterminal.

Note:

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of **grammars called 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.