TYPES OF PARSING

There are two types of parsing:

i). Top-down parsing
ii). Bottom-up parsing

I). TOP-DOWN PARSING

Top-down parsing is a parsing strategy that finds the derivation of the input string from the start symbol of the grammar.

There are two main approaches for top-down parsing:

a) Recursive descent parsing
b) Predictive parsing

a) Recursive Descent Parsing

It is a common form of top-down parsing that uses a parsing technique, which recursively parses the input to make a parse tree from the top and the input is read from left to right. It is called recursive, as it uses recursive procedures to process the input. The central idea is that each nonterminal in the grammar is implemented by a procedure in the program.
Generally, these parsers consist of a set of mutually recursive routines that may require back-tracking to create the parse tree.

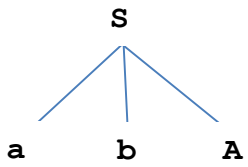
Backtracking: It means, if one derivation of a production fails, the syntax analyser restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Example

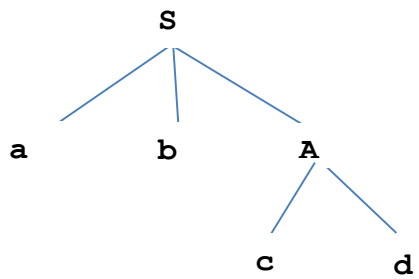
Consider the following grammar

$$S \rightarrow abA$$
$$A \rightarrow cd|c|e$$

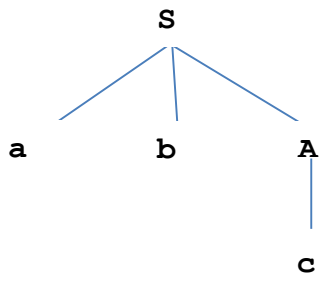
For the input stream ab, the recursive descent parser starts by constructing a parse tree representing $S \rightarrow abA$ as shown below:



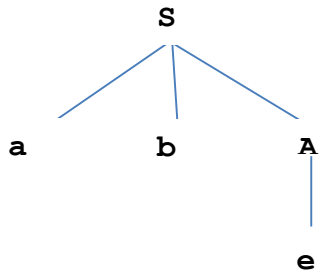
The stream is then expanded with the production $A \rightarrow cd$ as shown below



Since it does not match ab, it backtracks and tries the alternative $A \rightarrow c$ as shown below:



However, the parse tree does not match ab; the parser backtracks and tries the alternative $A \rightarrow e$ as shown below:



With this, it finds a match and thus pursing is completely successful.

In general, using recursive descent parsing, one can write a procedure for each non-terminal in the language definition:

i). Where the right hand side contains alternatives, the next symbol to input provides the selector for a switch statement, each branch of which represents one alternative.
ii). If e is an alternative, it becomes the default of the switch.
iii). Where a repetition occurs, it can be implemented iteratively.
iv). The sequence of actions within each branch, possibly one, consists of an inspection of the lexical token for each terminal, and a procedure will be assigned to a local or global variable as required by semantics of the language.

Example

The following is a recursive descent parser for the grammar shown below:

$T' \rightarrow T\$$
$T \rightarrow R$
$T \rightarrow aTc$
$R \rightarrow \varepsilon$
$R \rightarrow bR$

```
parseT'() =
     if next = 'a' or next = 'b' or next = '$' then
          parseT() ; match('$')
     else reportError()

parseT() =
     if next = 'b' or next = 'c' or next = '$' then
          parseR()
     else if next = 'a' then
          match('a') ; parseT() ; match('c')
```

```
        else reportError()

parseR() =
        if next = 'c' or next = '$' then
              (* do nothing *)
        else if next = 'b' then
              match('b') ; parseR()
        else reportError()
```

b) Predictive Parsing

Predictive parsing is a special form of recursive descent parsing in which the current input token unambiguously determines the production to be applied at each step i.e. they are backtrack free.

Non-recursive Predictive – LL(k) Parsing

This is a parsing strategy which involves building tables instead of writing code. It is also possible to automatically create tables. These parsers scan over the input stream using a prefix of tokens so as to identify production to be applied. The language accepted by these parsers is called

$LL(k)$,

where k is the length of the prefix. *LL* Stands for: *Left to right scan; left most derivation*.

The length of the prefix to be considered will be $k = 1$. The construction of an $LL(1)$ parser for the grammar $< V_N, V_T, P, S >$ requires first computing the following properties:

$$\textbf{FIRST}(\textbf{A}) = \{u | u \in V_T and\ A\ can\ derive\ a\ string\ starting\ with\ u\}$$

$$\textbf{FOLLOW}(\textbf{A}) = \{u | u \in V_T and\ S\ can\ derive\ a\ string\ like\ \alpha A u \beta\}$$

$$\textbf{FIRST}(\boldsymbol{\alpha}) = \{u | u \in V_T\ and\ \alpha can\ derive\ a\ string\ like\ u\beta\}$$

Generally, FIRST (A) is the set of terminals that can start a string derivable from A; FOLLOW (A) is the set of terminals that can follow an occurrence of A, FIRST (α) is the set of terminals that can start a string derivable from α.

To compute these sets of grammar, it is necessary to define another property:

nullable (A) = { true, If A can derive e in zero(0) or more steps
               { false, Otherwise

Notice that a non-terminal A is nullable if $nullable(A) = true$, and a terminal is never nullable.

The following is the algorithm for computing first and follow sets of a grammar:

for all $A \in N\ do$

$FIRST(A) = \phi;$

$FOLLOW(A) = \phi;$

$for\ all\ u \in T\ do$

$FIRST(u) = \{u\};$

do

$for\ each\ A \to X_1 \dots X_n \in P\ do$

$for\ each\ i \in [1 \dots, n]\ do$

$if\ X_1 \dots X_{i-1} are\ all\ nullable\ then$

$FIRST(A) = FIRST(A) \cup FIRST(X_i)$

$if\ X_{i+1}, \dots, X_n are\ all\ nullable\ then$

$FOLLOW(X_i) = FOLLOW(X_i) \cup FOLLOW(A)$

$for\ each\ j \in [i+1, \dots, n]\ do$

$if\ X_{i+1}, \dots, X_{j-1} are\ all\ nullable\ then$

$FOLLOW(X_i) = FOLLOW(X_i) \cup FOLLOW(X_j);$

$if\ all\ X_i's\ are\ all\ nullable\ then$

$NULLABEL(A) = true$

$until\ FIRST, FOLLOW\ and\ NULLABLE\ sets\ do\ not\ change$

We can extend the notation of FIRST set to include e and define them for strings as follows:

$$FIRST(e) = \{e\}$$

$$FIRST(X\alpha) = \begin{cases} X & if\ X \in T \\ FIRST(X) & if\ NULLABLE(X) = false \\ FIRST(X) \cup FIRST(\alpha) & if\ NULLABLE(X) = true \end{cases}$$

LOOKAHEAD

For a production rule, $A \rightarrow \alpha, LOOKAHEAD(A \rightarrow \alpha)$ is defined as the set of terminals which can appear next in the input when recognizing a production rule $A \rightarrow \alpha$. Thus, a production rule's LOOKAHEAD set specifies the tokens which should appear next in the input before a production is applied.

To build $LOOKAHEAD(A \rightarrow \alpha)$:

i). **Put** $FIRST(\alpha) - \{e\}$ **in** $LOOKAHEAD(A \rightarrow \alpha)$

ii). **If** $e \in FIRST(\alpha)$
then put $FOLLOW(A)$ **in** $LOOKAHEAD(A \rightarrow \alpha)$

A grammar G is $LL(1)$ iff, for each set of production, $A \rightarrow \alpha_1|\alpha_2| \ldots |\alpha_n$

$LOOKAHEAD(A \rightarrow \alpha_1), LOOKAHEAD(A \rightarrow \alpha_2) \ldots LOOKAHEAD(A \rightarrow \alpha_n)$ are all pairwise disjoint

The following are facts about $LL(1)$ grammar:

i). No left recursive grammar is $LL(1)$.

ii). No ambiguous grammar is $LL(1)$.

iii). Some languages have no $LL(1)$ grammar.

iv). An e-free grammar where each alternative expansion for A begins with a distinct terminal is a simple $LL(1)$ grammar.

Example 1

Given the grammar

$$S \rightarrow aS|a$$

Find whether it is $LL(1)$ grammar.

Solution:

$FIRST(S) = \{a\}$

Therefore, $LOOKAHEAD(S \rightarrow aS) =$

$$LOOKAHEAD(S \rightarrow a) = \{a\}$$

Hence it is not $LL(1)$grammar

Note: *FOLLOW sets are not required since empty sets are not required in the grammar.*

Example 2

Consider the grammar

$$E \rightarrow T|E + T$$
$$T \rightarrow F|T * F$$
$$F \rightarrow a|(E)$$

Find whether the grammar is $LL(1)$.

Solution:

Calculate the FIRST sets:

$$FIRST(F) = \{a, (\}$$
$$FIRST(T) = \{a, (\}$$
$$FIRST(E) = \{a, (\}$$

Calculate the LOOKAHEAD sets:

$$LOOKAHEAD(E \rightarrow T) = \{a, (\}$$

$LOOKAHEAD(E \rightarrow E + T) = \{a, (\}$
$LOOKAHEAD(T \rightarrow F) = \{a, (\}$
$LOOKAHEAD(T \rightarrow T * F) = \{a, (\}$
$LOOKAHEAD(F \rightarrow a) = \{a\}$
$LOOKAHEAD\big(F \rightarrow (E)\big) = \{(\}$

Hence the grammar is not $LL(1)$.

Example 3

Verify whether or not the following grammar is $LL(1)$:

$S \rightarrow ABa$
$A \rightarrow bA$
$A \rightarrow e$
$B \rightarrow aB$
$B \rightarrow e$

Solution

$$LOOKAHEAD(S \rightarrow ABa) = (FIRST(A) - \{e\}) \cup (FIRST(B) - \{e\}) \cup FIRST(a)$$
$$= \{b\} \cup \{a\} = \{a, b\}$$
$$LOOKAHEAD(A \rightarrow bA) = FIRST(b) = \{b\}$$
$$LOOKAHEAD(A \rightarrow e) = (FIRST(e) - \{e\}) \cup FOLLOW(A) = \{\} \cup \{a\} = \{a\}$$
$$LOOKAHEAD(B \rightarrow aB) = FIRST(a) = \{a\}$$
$$LOOKAHEAD(B \rightarrow e) = (FIRST(e) - \{e\}) \cup FOLLOW(B) = \{\} \cup \{a\} = \{a\}$$

Hence the grammar is not $LL(1)$ since a predicts $B \rightarrow aB$ and $B \rightarrow e$.

Example 4

Consider the following grammar:

$S \rightarrow aAa$
$S \rightarrow b$
$A \rightarrow ab$
$A \rightarrow e$

Verify whether the grammar is $LL(1)$ or not.

Solution:

$LOOKAHEAD(S \rightarrow aAa) = FIRST(a) = \{a\}$
$LOOKAHEAD(S \rightarrow b) = FIRST(b) = \{b\}$
$LOOKAHEAD(A \rightarrow ab) = FIRST(a) = \{a\}$
$LOOKAHEAD(A \rightarrow e) = FOLLOW(A) = \{a\}$

Therefore the grammar is not $LL(1)$ since a is in both $A \rightarrow ab$ and $A \rightarrow e$.

LL (1) Parse Table Construction

Method - 1:

Once the first and follow states have been computed, it is possible to construct $LL(1)$ parse table M that maps pairs of non-terminals and terminals to production using the following method:

Input: Grammar G
Output: Parsing table M
Method:

1. $\forall\ production\ A \rightarrow \alpha$
   $\forall a \in LOOKAHEAD\ A \rightarrow \alpha$
   $add\ A \rightarrow \alpha\ to\ M[A, a]$
2. *Set each unidentified entry of* M *to error*

If $\exists M[A, a]$ with multiple entries then the grammar is not $LL(1)$.

Example

Consider the following grammar:

$S \rightarrow E$
$E \rightarrow TE'$
$E' \rightarrow +E|-E|e$
$T \rightarrow FT'$
$T' \rightarrow * T|/T|e$
$F \rightarrow id|num$

The following is the table of FIRST and FOLLOW sets:

Symbol	FIRST	FOLLOW
S	{num , id}	{$}
E	{num , id}	{$}
E‘	{+,-,e}	{$}
T	{num, id}	{+,-,$}
T’	{*,/,e}	{+,-,$}
F	{num, id}	{+,-,*,/,$}
id	{id}	
num	{num}	
*	{*}	
/	{/}	
+	{+}	
-	{-}	

The following is the table of LOOKAHEAD sets:

Production	LOOKAHEAD
S→E	{num, id}
E→TE’	{num, id}
E’→+E	{+}
E’→-E	{-}
E’→e	{$}
T→FT’	{num, id}
T’→*T	{*}

T'→/T	{/}
T'→e	{+,-,$}
F→id	{id}
F→num	{num}

Note: The $ sign is a special symbol for 'end of input'.

The following is the parsing table:

Symbol	Id	num	+	-	*	/	$
S	S→E	S→E					
E	E→TE'	E→TE'					
E'			E'→+E	E'→-E			E'→e
T	T→FT'	T→FT'					
T'			T'→e	T'→e	T'→*T	T'→/T	T'→e
F	F→id	F→num					

Method - 2:

Alternatively, once the FIRST and FOLLOW sets have been computed, LL(1) parse table M, that maps pairs of non-terminals and terminals to productions can be constructed by using the following algorithm.

$$
\begin{aligned}
&For\ each\ A \to \alpha\ \in P\ do \\
&\quad if\ e \in First(\alpha)\ then \\
&\quad\quad for\ each\ u \in follow(A)\ do \\
&\quad\quad\quad add\ A \to\ \alpha\ to\ M[A,u] \\
&\quad for\ each\ u \in\ First\ (\alpha)\ do \\
&\quad\quad add\ A \to\ \alpha\ to\ M[A,u]
\end{aligned}
$$

If the resulting table has at most one production per (A, u) pair then the grammar is $LL(1)$:

Example

Consider the following grammar:

$E \to ME'$
$E' \to e$
$E' \to +ME'$
$M \to AM'$
$M' \to e$
$M' \to * AM'$
$A \to num$
$A \to (E)$

The following is the table of the first and follow sets:

Symbol	First	Follow
E	{num,(}	{),$}
E'	{+,e}	{),$}
M	{num,(}	{),+,$}
M'	{*,e }	{),+,$}
A	{num,(}	{),+,*,$}

The following is the parsing table M for the grammar:

	num	+	*	(	)	$
E	E→ME'			E→ME'		
E'		E'→+ME'			E'→e	E'→e
M	M→AM'			M→AM'		
M'		M'→e	M'→*AM'		M'→e	M'→e
A	A→num			A→(E)		

The table driven parser for $LL(1)$ grammar works using a parsing table and an auxiliary stack of grammar symbols. The table driven parser uses the following algorithm:

While stack is not empty do
    Let X be the top symbol on the stack
    Let U be the next input symbol
    If $X \in T$ *then*
        If $X = U$ *then*
            Pop X off the stack and advance the input
        Else
            Parsing error
    Else if $M[X, U] = A \rightarrow Y_1, \dots Y_n$ *then*
        Pop X and push $Y_n, \dots Y_1$ *onto the stack*
    Else
        Parsing error

The following is the trace of learning the algorithm on the string "$1 + 2 * 3$" using the above parsing table:

Stack	**Input**	**Action**
E	1+2*3$	Parse E→ME'
E'M	1+2*3$	Parse M→AM'
E'M'A	1+2*3$	Parse A→num
E'M'num	1+2*3$	Advance input
E'M'	+2*3$	Parse M'→e
E'	+2*3$	Parse E'→+ME'
E'M+	+2*3$	Advance input
E'M	2*3$	Parse M→AM'
E'M'A	2*3$	Parse A→num
E'M'num	2*3$	Advance input
E'M'	*3$	Parse M'→*AM'
E'M'A*	*3$	Advance input
E'M'A	3$	Parse A→num
E'M'num	3$	Advance input
E'M'	$	Parse M'→e
E'	$	Parse E'→e
	$	Accept

Note: Both the stack and the input contains an end symbol **$** to denote that the stack is empty and the input is consumed