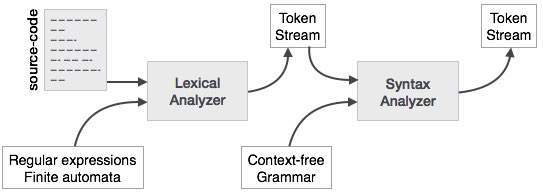
**Parsing:** Basic parsing technique, parsers, shift reduce parsing, operator-procedure parsing, top-down parsing, bottom up parsing, predictive parsing

**Syntax Analyzers**

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.



This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later in this chapter.

**Derivation**

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

* Deciding the non-terminal which is to be replaced.
* Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

**Left-most Derivation**

**If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation.** The sentential form derived by the left-most derivation is called the left-sentential form.

**Right-most Derivation**

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

**Example**

Production rules:

E → E + E

E → E \* E

E → id

Input string: id + id \* id

The left-most derivation is:

E → E \* E

E → E + E \* E

E → id + E \* E

E → id + id \* E

E → id + id \* id

Notice that the left-most side non-terminal is always processed first.

The right-most derivation is:

E → E + E

E → E + E \* E

E → E + E \* id

E → E + id \* id

E → id + id \* id

**Question: 5(a)2015 Define parse tree. Describe the construction of parse tree for a sentence ‘cad’ considering the grammer i.**

**Parse Tree**

**A parse tree is a graphical depiction of a derivation**. **It is convenient to see how strings are derived from the start symbol**.

The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic. We take the left-most derivation of a + b \* c

The left-most derivation is:

E → E \* E

E → E + E \* E

E → id + E \* E

E → id + id \* E

E → id + id \* id

Step 1:

|  |  |
| --- | --- |
| E → E \* E | Parse Tree Construction |

Step 2:

|  |  |
| --- | --- |
| E → E + E \* E | Parse Tree Construction |

Step 3:

|  |  |
| --- | --- |
| E → id + E \* E | Parse Tree Construction |

Step 4:

|  |  |
| --- | --- |
| E → id + id \* E | Parse Tree Construction |

Step 5:

|  |  |
| --- | --- |
| E → id + id \* id | Parse Tree Construction |

In a parse tree:

* All leaf nodes are terminals.
* All interior nodes are non-terminals.
* In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

**Ambiguity**

A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

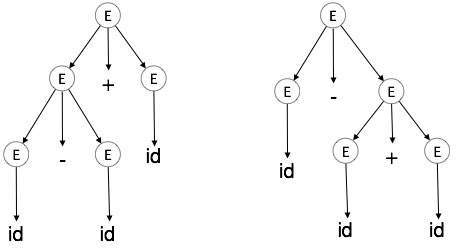
**Example**

E → E + E

E → E – Es

E → id

For the string id + id – id, the above grammar generates two parse trees:



The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

**Precedence**

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, 2+3\*4 can have two different parse trees, one corresponding to (2+3)\*4 and another corresponding to 2+(3\*4).

By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically \* (multiplication) has precedence over + (addition), so the expression 2+3\*4 will always be interpreted as:

2 + (3 \* 4)

These methods decrease the chances of ambiguity in a language or its grammar.

**Question 4(c):2016 what is left recursion of a grammer? “the top down method cannot handle left recursive grammer” Explain**

**Question 5(b):2015 Define left recursive grammer. How can you eliminate left-recursion from a context free grammer? Eliminate left recursion from**

**Left Recursion**

A grammar becomes left-recursive if it has any non-terminal ‘A’ whose derivation contains ‘A’ itself as the left-most symbol. Left-recursive grammar is considered to be a problematic situation for top-down parsers.

Top-down parsers start parsing from the Start symbol, which in itself is non-terminal. So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

**Example:**

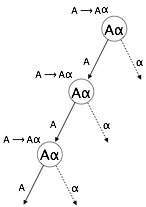
(1) A => Aα | β

(2) S => Aα | β

A => Sd

(1) is an example of immediate left recursion, where A is any non-terminal symbol and α represents a string of non-terminals.

(2) is an example of indirect-left recursion.



A top-down parser will first parse the A, which in-turn will yield a string consisting of A itself and the parser may go into a loop forever.

**Removal of Left Recursion**

One way to remove left recursion is to use the following technique:

The production

A => Aα | β

is converted into following productions

A => βA'

A'=> αA' | ε

This does not impact the strings derived from the grammar, but it removes immediate left recursion.

Second method is to use the following algorithm, which should eliminate all direct and indirect left recursions.

START

Arrange non-terminals in some order like A1, A2, A3,…, An

for each i from 1 to n

{

for each j from 1 to i-1

{

replace each production of form Ai ⟹Aj𝜸

with Ai ⟹ δ1𝜸 | δ2𝜸 | δ3𝜸 |…| 𝜸

where Aj ⟹ δ1 | δ2|…| δn are current Aj productions

}

}

eliminate immediate left-recursion

END

**Example**

The production set

S => Aα | β

A => Sd

after applying the above algorithm, should become

S => Aα | β

A => Aαd | βd

and then, remove immediate left recursion using the first technique.

A => βdA'

A' => αdA' | ε

Now none of the production has either direct or indirect left recursion.

**Limitations of Syntax Analyzers**

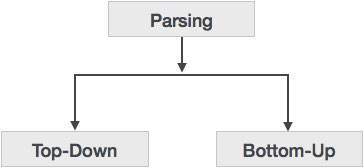
Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks -

* it cannot determine if a token is valid,
* it cannot determine if a token is declared before it is being used,
* it cannot determine if a token is initialized before it is being used,
* it cannot determine if an operation performed on a token type is valid or not.

These tasks are accomplished by the semantic analyzer, which we shall study in Semantic Analysis.

**Types of parsing:**

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : top-down parsing and bottom-up parsing.



**Top-down Parsing**

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

* **Recursive descent parsing** : It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
* **Backtracking** : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

**Bottom-up Parsing**

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

**Example:**

Input string : a + b \* c

Production rules:

S → E

E → E + T

E → E \* T

E → T

T → id

Let us start bottom-up parsing

a + b \* c

Read the input and check if any production matches with the input:

a + b \* c

T + b \* c

E + b \* c

E + T \* c

E \* c

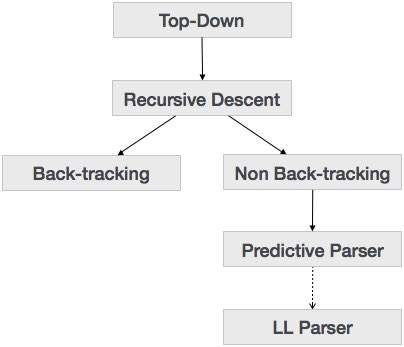
E \* T

E

S

**TopDown Parser:**

We have learnt in the last chapter that the top-down parsing technique parses the input, and starts constructing a parse tree from the root node gradually moving down to the leaf nodes. The types of top-down parsing are depicted below:



**Recursive Descent Parsing**

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every terminal and non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

**Back-tracking**

Top- down parsers start from the root node (start symbol) and match the input string against the production rules to replace them (if matched). To understand this, take the following example of CFG:

S → rXd | rZd

X → oa | ea

Z → ai

For an input string: read, a top-down parser, will behave like this:

It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e. ‘r’. The very production of S (S → rXd) matches with it. So the top-down parser advances to the next input letter (i.e. ‘e’). The parser tries to expand non-terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So the top-down parser backtracks to obtain the next production rule of X, (X → ea).

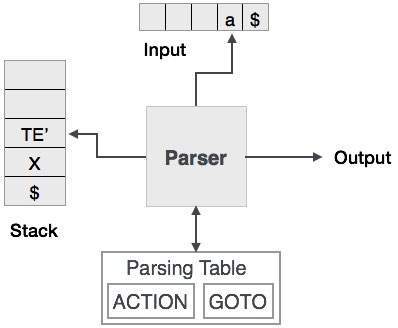
Now the parser matches all the input letters in an ordered manner. The string is accepted.

|  |  |  |  |
| --- | --- | --- | --- |
| Back Tracking | Back Tracking | Back Tracking | Back Tracking |

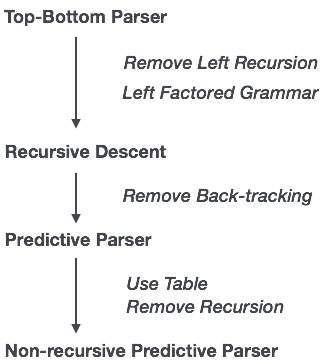
**Predictive Parser**

Predictive parser is a recursive descent parser, which has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from backtracking.

To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.



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.

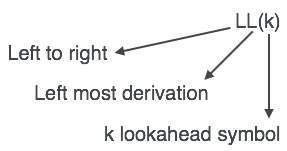


In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose. There might be instances where there is no production matching the input string, making the parsing procedure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

**LL Parser**

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look aheads. Generally k = 1, so LL(k) may also be written as LL(1).



**LL Parsing Algorithm**

We may stick to deterministic LL(1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL(1), then usually, it is not LL(k), for any given k.

Given below is an algorithm for LL(1) Parsing:

Input:

string ω

parsing table M for grammar G

Output:

If ω is in L(G) then left-most derivation of ω,

error otherwise.

Initial State : $S on stack (with S being start symbol)

ω$ in the input buffer

SET ip to point the first symbol of ω$.

repeat

let X be the top stack symbol and a the symbol pointed by ip.

if X∈ Vt or $

if X = a

POP X and advance ip.

else

error()

endif

else /\* X is non-terminal \*/

if M[X,a] = X → Y1, Y2,... Yk

POP X

PUSH Yk, Yk-1,... Y1 /\* Y1 on top \*/

Output the production X → Y1, Y2,... Yk

else

error()

endif

endif

until X = $ /\* empty stack \*/

A grammar G is LL(1) if A → α | β are two distinct productions of G:

* for no terminal, both α and β derive strings beginning with a.
* at most one of α and β can derive empty string.
* if β → t, then α does not derive any string beginning with a terminal in FOLLOW(A)

**Question 5(b)2017 what is left factoring? Why it is necessary in top-down parsing?**

**Left factoring:**

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.

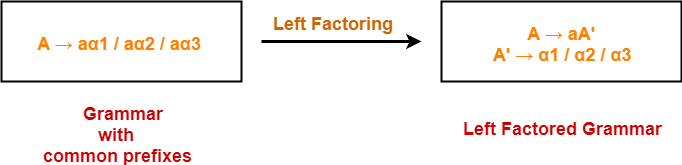
**Why this important for top-down parsing how?**

In left factoring,

* We make one production for each common prefixes.
* The common prefix may be a terminal or a non-terminal or a combination of both.
* Rest of the derivation is added by new productions.

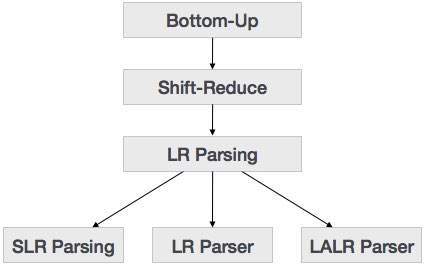
The grammar obtained after the process of left factoring is called as **Left Factored Grammar**.

**Example-**



**Bottomup Parsing:**

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The image given below depicts the bottom-up parsers available.



**Question 7(b):2017 Draw the syntax tree of the expression:**

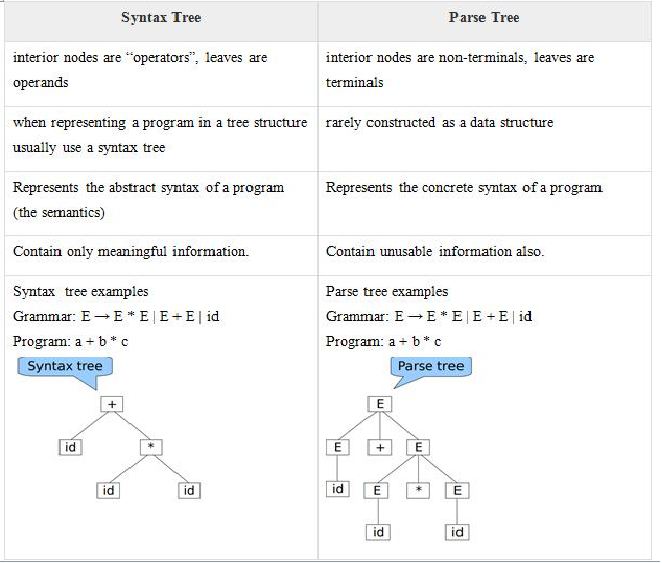
**Shift-Reduce Parsing**

Shift-reduce parsing uses two unique steps for bottom-up parsing. These steps are known as shift-step and reduce-step.

* **Shift step**: The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. This symbol is pushed onto the stack. The shifted symbol is treated as a single node of the parse tree.

**Question 4(a):2016 Define LL(1) grammer**

**Question 7(a):2017,2016 Distinguish between parse tree and syntax tree with example.**



**Question 6(b):2016 Draw the syntax tree for the for the expression : (A+B/C)/(A-C/F)\*F+(H\*Y\*Z)**

**Answer:**

* **Reduce step** : When the parser finds a complete grammar rule (RHS) and replaces it to (LHS), it is known as reduce-step. This occurs when the top of the stack contains a handle. To reduce, a POP function is performed on the stack which pops off the handle and replaces it with LHS non-terminal symbol.

**LR Parser**

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

* SLR(1) – Simple LR Parser:
  + Works on smallest class of grammar
  + Few number of states, hence very small table
  + Simple and fast construction
* LR(1) – LR Parser:
  + Works on complete set of LR(1) Grammar
  + Generates large table and large number of states
  + Slow construction
* LALR(1) – Look-Ahead LR Parser:
  + Works on intermediate size of grammar
  + Number of states are same as in SLR(1)

**LR Parsing Algorithm**

Here we describe a skeleton algorithm of an LR parser:

token = next\_token()

repeat forever

s = top of stack

if action[s, token] = “shift si” then

PUSH token

PUSH si

token = next\_token()

else if action[s, token] = “reduce A::= β“ then

POP 2 \* |β| symbols

s = top of stack

PUSH A

PUSH goto[s,A]

else if action[s, token] = “accept” then

return

else

error()

**LL vs. LR**

|  |  |
| --- | --- |
| **LL** | **LR** |
| Does a leftmost derivation. | Does a rightmost derivation in reverse. |
| Starts with the root nonterminal on the stack. | Ends with the root nonterminal on the stack. |
| Ends when the stack is empty. | Starts with an empty stack. |
| Uses the stack for designating what is still to be expected. | Uses the stack for designating what is already seen. |
| Builds the parse tree top-down. | Builds the parse tree bottom-up. |
| Continuously pops a nonterminal off the stack, and pushes the corresponding right hand side. | Tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal. |
| Expands the non-terminals. | Reduces the non-terminals. |
| Reads the terminals when it pops one off the stack. | Reads the terminals while it pushes them on the stack. |
| Pre-order traversal of the parse tree. | Post-order traversal of the parse tree |

**Question:5(c) Eliminate non-determinisom from the following grammer**

**Question:6(a) 2017 Given the grammer : i) II. T**

1. **i. construct sets of LR(1) items**
2. **ii. Construct canonical LR(1) passing table**
3. is the grammar LALR(1)? Justify answer