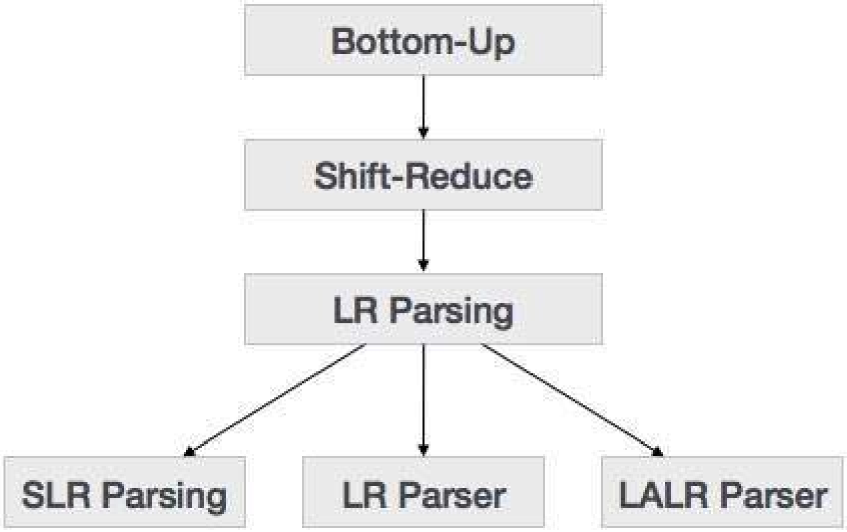
Compiler Design - Bottom-Up Parser

https://www.tutorialspoint.com/compiler_design/compiler_design_bottom_up_parser.htm

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.



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.

- **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 LRk 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:

- SLR1 – Simple LR Parser:
    - Works on smallest class of grammar
    - Few number of states, hence very small table
    - Simple and fast construction

- LR1 – LR Parser:
  - Works on complete set of LR1 Grammar
  - Generates large table and large number of states
  - Slow construction
- LALR1 – Look-Ahead LR Parser:
  - Works on intermediate size of grammar
  - Number of states are same as in SLR1

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.