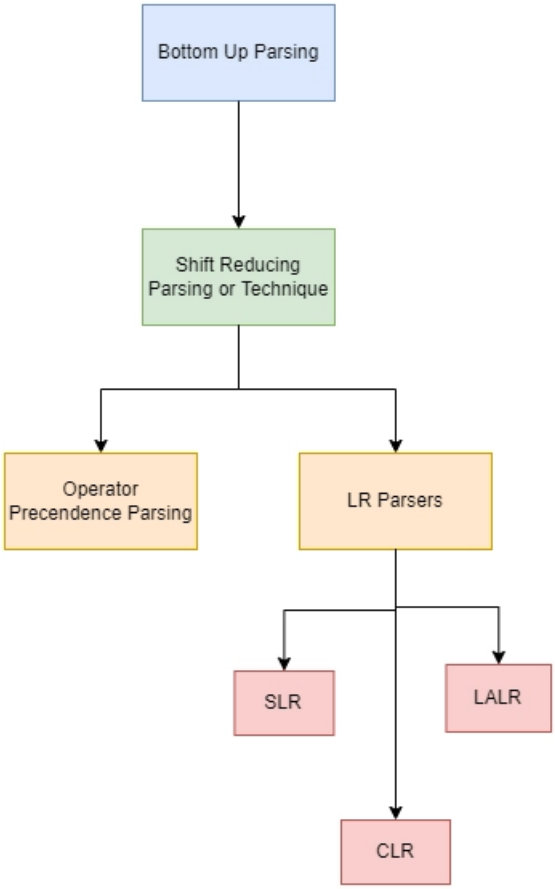
Bottom up parsing

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.

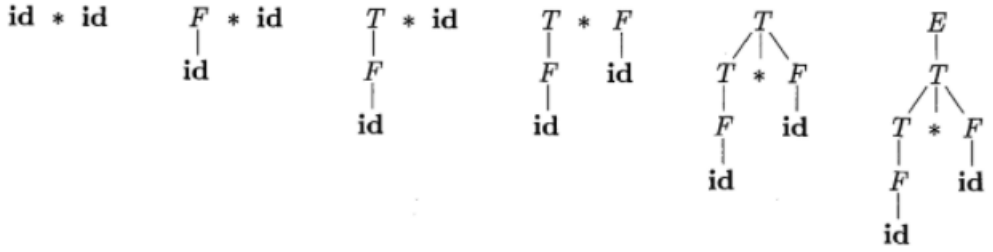


- In this technique, parsing starts from the leaf node of the parse tree to the start symbol of the parse tree in a bottom-up manner.
- Bottom-up parsing attempts to construct a parse tree by reducing input string and using **Right Most Derivation.**
- Bottom-up parsing starts with the input symbol and constructs the parse tree up to the start symbol using a production rule to reduce the string to get the starting symbol.

Example:

1. E → T
2. T → T * F
3. T → id
4. F → T
5. F → id

Parse Tree representation of input string "id * id" is as follows:



6.1 Shift-Reduce Parsing

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. Shift reduce parsing performs the two actions: **shift** and **reduce**. This is why it is known as shift-reduce parsing.

- The current symbol in the input string is pushed to a stack at the shift action. At each reduction, the symbols will be replaced by the non-terminals.
- The non-terminal is the left side of the output, and the symbol is the right side of the production.
- **$** is used to mark the bottom of the stack and also the right end of the input as follows:

STACK	INPUT
$	w $

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack.

- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.

1. *Shift.* Shift the next input symbol onto the top of the stack.
2. *Reduce.* The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. *Accept.* Announce successful completion of parsing.
4. *Error.* Discover a syntax error and call an error recovery routine.

Example:

1. E → T
2. T → T * F
3. T → id
4. F → T
5. F → id

The actions a shift-reduce parser might take in parsing the input string idl *id2 as shown below:

STACK	INPUT	ACTION
$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$ \mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \rightarrow \mathbf{id}$
$\$ F$	$* \mathbf{id}_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* \mathbf{id}_2 \$$	shift
$\$ T *$	$\mathbf{id}_2 \$$	shift
$\$ T * \mathbf{id}_2$	$\$$	reduce by $F \rightarrow \mathbf{id}$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept

Example:

Grammar:

1. S → S+S
2. S → S-S
3. S → (S)
4. S → a

Input string: a1-(a2+a3)

Parse it using shift reduce parsing:

Stack contents	Input String	Actions
$	a1-(a2+a3)$	Shift a1
$a1	-(a2+a3)$	Reduce by s->a
$S	-(a2+a3)$	Shift -
$S-	(a2+a3)$	Shift (
$S-(	a2+a3)$	Shift a2
$S-(a2	+a3)$	Reduce by S->a
$S-(S	+a3)$	Shift +
$S-(S+	a3)$	Shift a3
$S-(S+a3	)$	Reduce by S->a
$S-(S+S	)$	shift)
$S-(S+S)	$	Reduce by S->S+S
$S-(S)	$	Reduce by S->(S)
$S-S	$	Reduce by S->S-S
$S	$	Accept

6.2 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)