Bottom-Up Parsing

The term "Bottom-Up Parsing" refer to the order in which nodes in the parse tree are constructed, construction starts at the leaves and proceeds towards the root. Bottom-Up Parsing can handle a large class of grammars.

1. **Shift-Reduce Parsing:** Is a general style of Bottom-up syntax analysis , it attempts to construct a parse tree for an input string beginning at leaves and working up towards the root,(reducing a string *w* to the start symbol of grammar).At each reduction step a particular substring matching the right side of production is replaced by the symbol on the left of that production.

Example : consider the grammar

S ⟶ aABe

A ⟶ Abc | b

B ⟶ d

And the input is **abbcde**

The implementation Bottom-Up Parsing is

a <u>b</u> b c d e
a <u>A b c</u> d e
a A <u>d</u> e
<u>a A B e</u>
S
Accept

Handle : Is a substring that matches the right side of a production.

Stack Implementation of Shift-Reduce Parsing:

A convenient way to implement a shift-reduce parser is to use a *Stack* to hold a grammar symbols and an input buffer to hold the sting *w* to be parsed. We use $ to mark the bottom of *stack* and also the right end of the input string. There are actually four possible actions:

1. **Shift :** The next input symbol is Shifted onto the top of *stack*.
2. **Reduce :** Replace the handle with nonterminal.
3. **Accept :** The parser announces successful completion of parsing .
4. **Error :** The parser discovers that syntax error has occurred and calls an error recovery routine.

Example: Consider the following grammar

$$\mathbf{E \longrightarrow E+E \mid E*E \mid (E) \mid id}$$

And the input string is **id + id * id,** then the implementation is :

Stack	Input Buffer	Action
$	id+id*id$	Shift
$id	+id*id$	Reduce: E→id
$E	+id*id$	Shift
$E+	id*id$	Shift
$E+id	*id$	Reduce: E→id
$E+E	*id$	Shift(*)
$E+E*	id$	Shift
$E+E*id	$	Reduce: E→id
$E+E*E	$	Reduce: E→E*E
$E+E	$	Reduce: E→E+E
$E	$	Accept

Conflicts During Shift-Reduce Parsing:

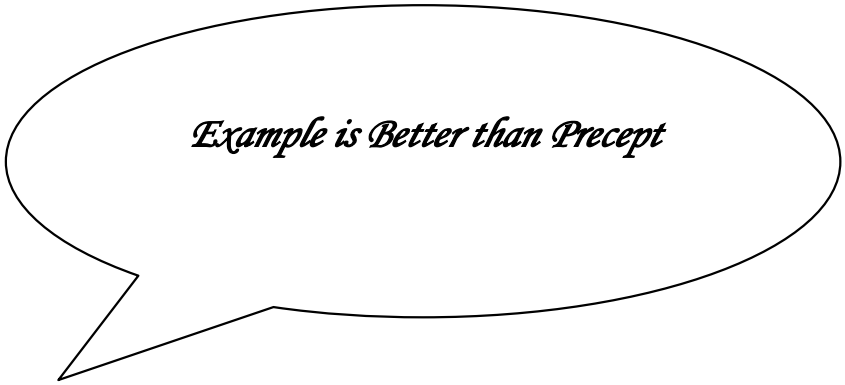
There are context free grammars for which shift-reduce parsing cannot be used. Ambiguous grammars lead to parsing conflicts. Can fix by rewriting grammar or by making appropriate choice of action during parsing. There are two type of conflicts :

1. **Shift/Reduce** conflicts: should we shift or reduce? (See previous example (*))
2. **Reduce/Reduce** conflicts: which production should we reduce with? for example:

stmt → id(param)
param → id
expr → id(expr) | id

Stack	**Input Buffer**	**Action**
$...id(id	,id)...$	Reduce by ??

Should we reduce to **param** or to **expr** ?

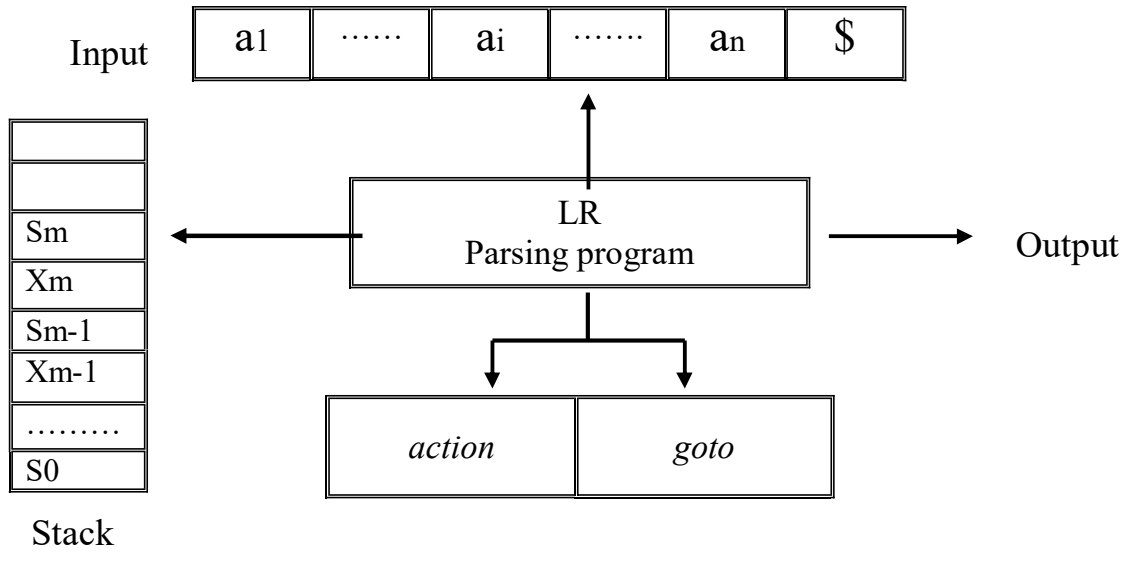


LR Parsers

This section presents an efficient Bottom-Up syntax analysis technique that can be used to parse a large class of context-free grammars. The technique is called LR(k) parsing, the "L" is for left to right scanning of input, the "R" for constructing a rightmost derivation in reverse, and "k" for the number of input symbols of lookahead that are used in making parsing decisions-when "k" is omitted , k is assumed to be 1).
LR parsing is attractive for a variety of reasons:-

1. LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
2. The LR parsing method is the most general *nonbacktracking* shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
3. The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.

The schematic form of an LR parser is shown in following figure .It consists of **an Input**,**an Output**,**a Stack**,**a Driver program**,and **a Parsing table** that has two parts (*action* and *goto*) .



Model of an LR parser