Bottom-up Parsing

Bottom-up parsing is more general than top-down parsing. Bottom-up parsers handle a large class of grammars. It is the preferred method in practice. It is also called *LR* parsing; *L* means that tokens are read left to right and *R* means that the parser constructs a rightmost derivation. LR parsers do not need left-factored grammars. LR parsers can handle left-recursive grammars.

Consider the grammar

S → aABe
A → Abc | b
B → d

The sentence abbcde can be reduced to S:

$a\underline{b}bcde$ — b
$a\underline{A}bcde$
$a\underline{A}de$ — Abc
$aA\underline{B}e$ — d
S

These reductions, in fact, trace out the following right-most derivation in reverse:

$$
\begin{aligned}
S &\Rightarrow a\overset{2}{A}\overset{1}{B}e\\
&\Rightarrow aA\underline{d}e\\
&\Rightarrow aAbcde\\
&\Rightarrow abbcde
\end{aligned}
$$

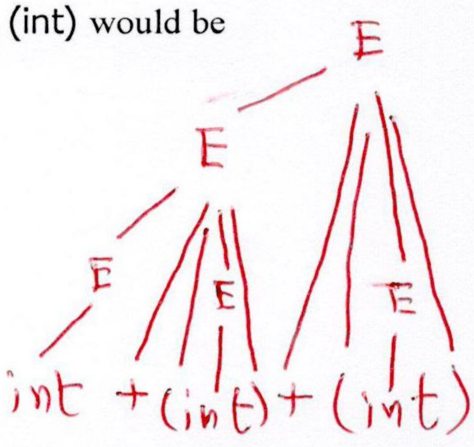
Rightmost Derivation

Consider the grammar

1. E → E + (E)
2. | int

The bottom-up parse of the string int + (int) + (int) would be

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E



Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

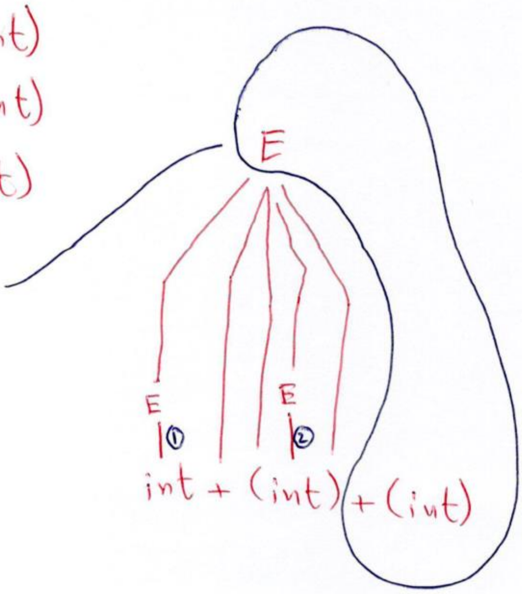
1. Shift
2. Reduce

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

E + (int)



Rightmost Derivation in reverse !

Shift moves ▶ one place to the right which shifts a terminal to the left string

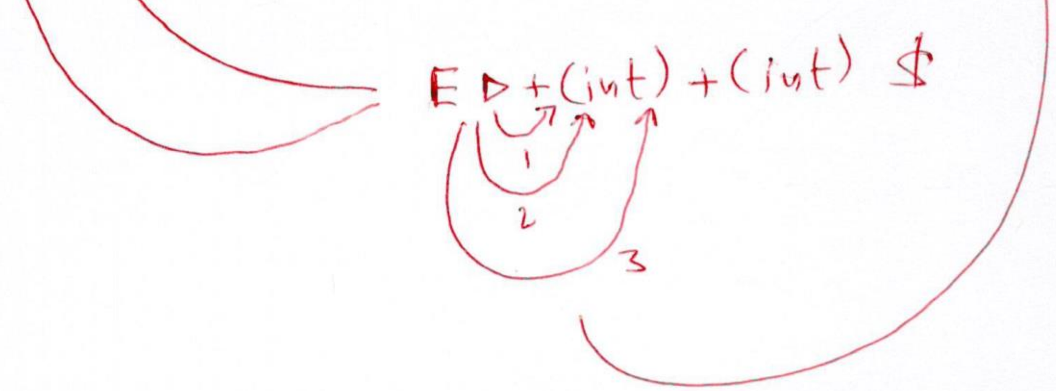
E + (▶ int) ⇒ E + (int ▶)

In the *reduce* action, the parser applies an inverse production at the right end of the left string. If E → E + (E) is a production, then

E + (E+(E)▶) ⇒ E + (E ▶)

Shift-Reduce Example

▶ int + (int) + (int) $	shift
int ▶ + (int) + (int) $	reduce E → int
E ▶ + (int) + (int) $	shift 3 times
E + (int ▶) + (int) $	reduce E → int
E + (E ▶) + (int) $	shift
E + (E) ▶ + (int) $	reduce E → E+(E)
E ▶ + (int) $	shift 3 times
E + (int ▶) $	reduce E → int
E + (E ▶) $	shift
E + (E) ▶ $	red E → E+(E)
E ▶ $	*accept*



lhs ▶ rhs

Shift-Reduce: The Stack

A stack can be used to hold the content of the left string. The Top of the stack is marked by the ► symbol. The shift action pushes a terminal on the stack. Reduce pops zero or more symbols from the stack (production *rhs*) and pushes a non-terminal on the stack (production *lhs*)

Discovering Handles

A bottom-up parser builds the parse tree starting with its leaves and working toward its root. The upper edge of this partially constructed parse tree is called its *upper frontier*. At each step, the parser looks for a section of the upper frontier that matches right-hand side of some production. When it finds a match, the parser builds a new tree node with the production's left-hand non-terminal thus extending the frontier upwards towards the root. The critical step is developing an efficient mechanism that finds matches along the tree's current frontier.

Formally, the parser must find some substring β, of the upper frontier where

1. β is the right-hand side of some production A → β, and
2. A → β is one step in right-most derivation of input stream

We can represent each potential match as a pair ⟨A→β,*k*⟩, where *k* is the position on the tree's current frontier of the right-end of β. The pair ⟨A→β,*k*⟩, is called the *handle* of the bottom-up parse.

Handle Pruning

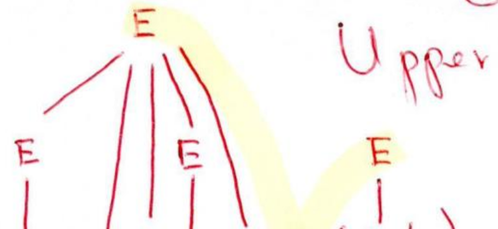
Selective Removal of parts of a plant (wikipedia)

A bottom-up parser operates by repeatedly locating handles on the frontier of the partial parse tree and performing reductions that they specify. The bottom-up parser uses a stack to hold the frontier. The stack simplifies the parsing algorithm in two ways.

①

First, the stack trivializes the problem of managing space for the frontier. To extend the frontier, the parser simply pushes the current input token onto the top of the stack.①
Second, the stack ensures that all handles occur with their right end at the top of the② stack. This eliminates the need to represent handle's position.

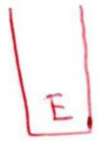
②



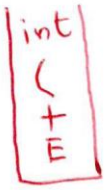
Upper frontier

E + E will be in stack

E ▷ + (int) + (int) $

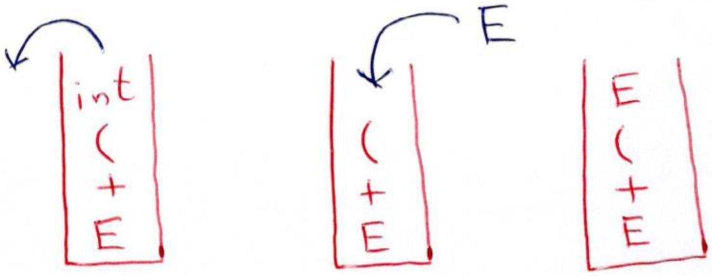


E + (int ▷) + (int) $



shift 3 times

E + (E ▷) + (int) $



reduce
E → int

Four actions of a parser:

1- Shift
2- Reduce
3- Accept
4- Error

Handles

The handle-finding mechanism is the key to efficient bottom-up parsing. As it process an input string, the parser must find and track all potential handles. For example, every legal input eventually reduces the entire frontier to grammar's goal symbol. Thus, ⟨Goal → Expr,1⟩ is a potential handle at the start of every parse. As the parser builds a derivation, it discovers other handles. At each step, the set of potential handles represent different suffixes that lead to a reduction. Each potential handle represent a string of grammar symbols that, if seen, would complete the right-hand side of some production.

For the bottom-up parse of the expression grammar string, we can represent the potential handles that the shift-reduce parser should track. Using the placeholder • to represent top of the stack, there are nine handles:

	Handles
1	⟨Factor → id •⟩
2	⟨Term → Factor•⟩
3	⟨Expr → Term •⟩
4	⟨Factor → num•⟩
5	⟨Term → Factor•⟩
6	⟨Factor → id •⟩
7	⟨Term → Term × Factor •⟩
8	⟨Expr → Expr – Term •⟩
9	⟨Goal → Expr •⟩

x – 2 × y

(tokenized as id – num ∗ id)

	word	**Stack**	**Handle**	**Action**
1	id	▶	- none -	*shift*
2	–	id ▶	⟨Factor → id,**1**⟩	*reduce*
3	–	Factor ▶	⟨Term → Factor,**1**⟩	*reduce*
4	–	Term ▶	⟨Expr → Term,**1**⟩	*reduce*
5	–	Expr ▶	- none -	*shift*
6	num	Expr – ▶	- none -	*shift*
7	×	Expr – num ▶	⟨Factor → num,**3**⟩	*reduce*
8	×	Expr – Factor ▶	⟨Term → Factor,**3**⟩	*shift*
9	×	Expr – Term ▶	- none -	*shift*
10	id	Expr – Term × ▶	- none -	*shift*
11	$	Expr – Term × id▶	⟨Factor → id,**5**⟩	*reduce*
12	$	Expr–Term × Factor▶	⟨Term→Term×Factor,**5**⟩	*reduce*
13	$	Expr – Term ▶	⟨Expr → Expr – Term,**3**⟩	*reduce*
14	$	Expr ▶	⟨Goal → Expr,**1**⟩	*reduce*
15	$	Goal	- none -	***accept***