Bottom-up Parsing

MICHAEL WOLLOWSKI

Bottom-Up Parsing: Principles

Build parse tree starting with the leaves.

Work towards root.

Leaves are terminals.

Construct a leaf for each word returned by scanner.

To build a derivation, the parser adds layers of non-terminals on top of leaves.

Structure is dictated by grammar and partially completed lower portion of parse tree.

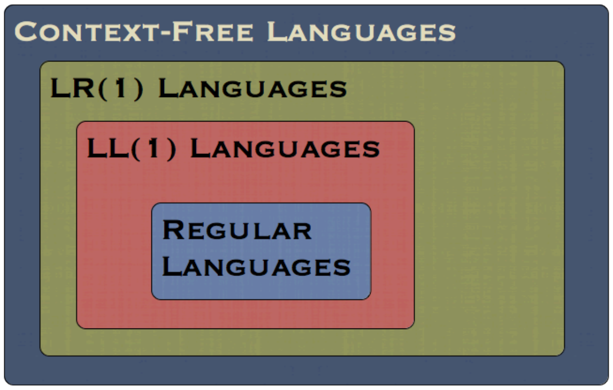
Bottom-Up Parsing: Use

Bottom-up parsing is designed for LR(1) languages

- L: Read input **L**eft-to-right
- R: Build a **R**ightmost derivation
- 1: Use **1** token of look-ahead

For bottom-up parsing, we may use a grammar that is left-recursive and is **not** left-factored.

Language Classes



Bottom-Up Parsing: Process

We call the *frontier* of a parse tree the current sentential form in the derivation.

To extend the frontier upward, the parser looks in the current frontier for a substring that matches the rhs of some production *A* → *b*.

If it finds *b* in the frontier, with its right end at *k*, it can replace *b* with *A* to create a new frontier.

If replacing *b* with *A* at position *k* is the next step in a valid derivation, then the pair <*A* → *b*, *k*> is a *handle* in the current derivation.

Finding handles is a key issue in bottom-up parsing.

Bottom-Up Parsing: Process

We call a replacement of *b* by *A* a *reduction.*

A bottom-up parser follows the follow process:

- Attempt to find a handle <*A* → *b*, *k*>
- Reduce *b* at *k* with *A*
- If the goal symbol has been derived, terminate with success.
- If not, terminate with failure

If the parser terminates with failure, it should use the context accumulated in the partial derivation to issue a meaningful error message.

In many cases, the parser can recover from an error and continue parsing.

Action and Goto Tables

An LR(1) parser uses a handle-finding automaton, encoded into two tables, called *Action* and *Goto.*

To find the next handle, a shift-reduce parser shifts symbols onto the stack until the automaton finds the rhs of a handle at the top of the stack.

Once it has a handle, the parser reduces the rhs symbols by the lhs non-terminal of the matching production A → β.

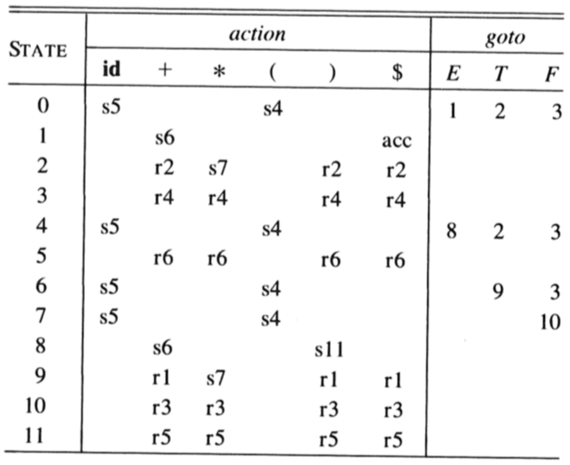
To do so, it pops the symbols in β from the stack and pushes *A* onto the stack.

Action-Goto table

Grammar:

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

Action-Goto Table:



STATE	*action*						*goto*		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Shift-Reduce Parsers

Bottom-up or LR(1) parsers are oftentimes called *shift-reduce* parsers.

This is due to their primary actions of shifting input onto a stack until a handle is found that can then be reduced to its lhs non-terminal.

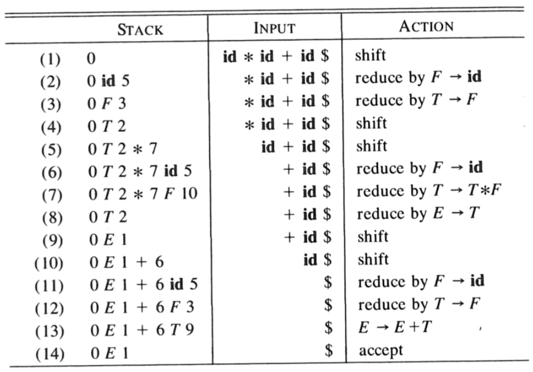
We use a stack to hold grammar symbols and we use an input buffer to hold the string w to be parsed.

In the following example (from pp 198/9 of the Dragon book), we use $ to mark the right end of the input, i.e. it indicates **eof**

Skeleton Shift-Reduce Parser

```
push $;
push start state, $s_0$;
word ← NextWord( );
while (true) do;
   state ← top of stack;
   if Action[state,word] = "reduce $A \to \beta$" then begin;
        pop 2 × |$\beta$| symbols;
        state ← top of stack;
        push A;
        push Goto[state, A];
   end;
   else if Action[state,word] = "shift $s_i$" then begin;
      push word;
      push $s_i$ ;
      word ← NextWord( );
   end;
   else if Action[state,word] = "accept"
      then break;
   else Fail( );
end;
report success;   /* executed break on "accept" case */
```

Solution to Class Exercise



	Stack	Input	Action
(1)	0	**id** $*$ **id** + **id** $	shift
(2)	0 **id** 5	$*$ **id** + **id** $	reduce by $F \to$ **id**
(3)	0 F 3	$*$ **id** + **id** $	reduce by $T \to F$
(4)	0 T 2	$*$ **id** + **id** $	shift
(5)	0 T 2 $*$ 7	**id** + **id** $	shift
(6)	0 T 2 $*$ 7 **id** 5	+ **id** $	reduce by $F \to$ **id**
(7)	0 T 2 $*$ 7 F 10	+ **id** $	reduce by $T \to T*F$
(8)	0 T 2	+ **id** $	reduce by $E \to T$
(9)	0 E 1	+ **id** $	shift
(10)	0 E 1 + 6	**id** $	shift
(11)	0 E 1 + 6 **id** 5	$	reduce by $F \to$ **id**
(12)	0 E 1 + 6 F 3	$	reduce by $T \to F$
(13)	0 E 1 + 6 T 9	$	$E \to E+T$
(14)	0 E 1	$	accept

Fig. 4.32. Moves of LR parser on **id** $*$ **id** + **id**.

Producing Parse Trees

Whenever there is a "reduce" action, the production can be used to extend the parse tree.

To the right, you see the parse tree that is produced when reading off the reduce actions from the prior slide.

```
          E
        / | \
      E   |  |
      |   |  |
      T   |  |
    /| \  |  |
   T |  | |  T
   | |  | |  |
   F |  F |  F
   | |  | |  |
  id * id + id
```

Parsing Erroneous Input

Any empty entry in the action-goto table is an error state.