A Short Tutorial on Bottom-Up Parsing

Mark Seigle (seigle@cs.washington.edu)

October 16, 2000

1 Intro

We will be using the following grammar G:
0 : $S' \rightarrow S\$$
1 : $S \rightarrow (L)$
2 : $S \rightarrow x$
3 : $L \rightarrow S$
4 : $L \rightarrow L, S$

G generates strings with sequences of matching parens around one or more x's, i.e. $((x))$, $(((x)))$, x, and $(x,(x)) \in G$. The dollar sign signifies the end of input. Recall that LR(k) stands for *Left-to-right parse, Rightmost-derivation, k-token lookahead.*

We discussed the basics of bottom-up/shift-reduce parsing in class but we didn't answer the question as to how the parser knows when to shift and when to reduce! We need a criteria for the parser to choose when to shift or reduce. The criteria comes from regular expressions which serve as handles or equivalently a DFA.

2 Building the Table

LR(0) parsers operate without any look ahead—that is they operate just by looking at what is on the stack. For some string of terminals and non-terminals, we represent the stack and our position on the stack as follows: $.\alpha\beta w$, we use . to divide the stack into two halves. All elements to the left of the . are visible on the stack, all elements to the right of the . haven't been shifted onto the stack. The sequence $\alpha\beta w$ is a sequence of terminals and non-terminals. We showed in class that everything to the right of the . has to be a terminal.

At first our parser's stack is empty and the remaining input forms our S and \$. In production 0, this would look like $S' \rightarrow .S\$$. Now we can begin constructing our states of the DFA. Since we have a . followed by an S, we can expand the productions of S and precede them with a .. Basically what this corresponds to are all of the possible ways to parse the initial input.

$$\boxed{\begin{array}{l} S' \rightarrow .S\$ \\ S \rightarrow .x \\ S \rightarrow .(L) \end{array}}^{1}$$

The superscripted number represents the state number. Now let's consider the possibilities while we're in state 1. If we shift an x, we indicate that by shifting the dot past the x in the $S \rightarrow x$ production. The other rules in state 1 don't have anything to do with x's, so we ignore them.

$$\boxed{S \rightarrow x.}^{2}$$

Another possibility in state 1 is to see a left paren. We should shift the left paren $S \to (.L)$. We know that we're eventually going to see a string derived by L and then a right paren. Just as we did for state 1, when we had a **.** before a S, we had to include all of S's productions, we have to include all of L's productions.

$S \to (.L)$
$L \to .L, S$
$L \to .S$
$L \to .(L)$
$S \to .x$

[3]

The final possibility in state 1 is to somehow read an S and shift the stack to S **.** \$. This is a somewhat special condition, as we know that we've finished parsing a production. When we construct the table for the DFA, this state will have a *goto* action.

To build a parser from a grammar, we need an algorithmic method to construct the states we've previously been hand-waving through. The procedure **closure(I)** takes care of adding the necessary productions to a state. **goto (I, X)** for some set of productions in a state I, and some symbol X, goto moves the dot past X in all of the elements of I.

closure (I) =
  repeat
    for any item $A \to \alpha \,.\, \beta$ in I
      for any production $X \to \gamma$
        $I \leftarrow I \cup \{X \to .\gamma\}$
  until I does not change.
  return I

goto (I, X) **=**
  set J to the empty set
  for any item $A \to \alpha \,.\, X\beta$ in I
    add $A \to \alpha X \,.\, \beta$ to J
  return closure(J)

Now we can construct a set of states T and edges E for our DFA. We'll use the notation $\stackrel{x}{\to}$ to indicate an edge in our DFA we would follow upon reading an x.

Initialize T to {**closure**$(\{S' \to .S\$\})$
Initialize E to empty
repeat
  for each state I in T
    for each item $A \to \alpha \,.\, X\beta$ in I
      let J be **goto**$(I$,J$)$
      $T \leftarrow T \cup \{J\}$
      $E \leftarrow E \cup \{I \stackrel{X}{\to} J\}$
until E and T did not change in this iteration

So far we still haven't talked about reducing! We can now compute in which states we should reduce. Let R be a set of states $\times$ productions.

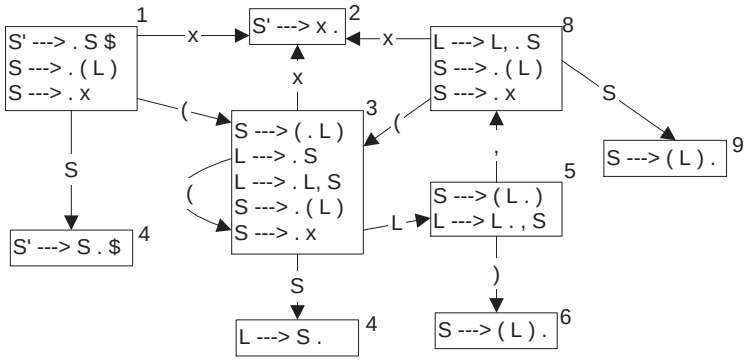

Figure 1: The completed DFA for our grammar

0	(	)	x	,	$	S	L
1	s3		s2				g4
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

Figure 2: The parsing table for our grammar

$R \leftarrow \{\}$
for each state I in T
  for each item $A \rightarrow \alpha\mathbf{.}$ in I
    $R \leftarrow R \cup \{(I, A \rightarrow \alpha)\}$

Now we can construct a table for our parse. For each edge $I \xrightarrow{X} J$ where X is a terminal, we put the action *shift* J at position (I, X) of the table, if X is a non-terminal, we put *goto* J at position (I, X). For any state containing $S' \rightarrow S\,\mathbf{.}\,\$$, we put an *accept* action at $(I, \$)$. And for any state containing $A \rightarrow \gamma\mathbf{.}$, we put a reduce action at (I, Y) for every token Y.

What if we get a shift and reduce in the same entry in the DFA? Then the grammar isn't LR(0). You can manually correct the conflict and in doing so are creating some precedence rules. For simple cases, hacking the precedence is fine (and sometimes very useful), however for large complex grammars, it is best to use a more general parsing strategy like LR(1), SLR, or LALR. See your book for more info.

3 Acknowledgments

This document borrows heavily from Andrew Appel's "Modern Compiler Implementation in C". The examples here are identical to those in the book in the hopes that fewer errors would appear.