Lecture 9: LR Parsing

Carsten Schürmann

October 15, 2001

In the previous lecture we have seen how LR parsing works in general. The string that is to be parsed is passed to the parser, which tries to construct a parse tree from bottom up. To this end it uses a stack on which it may shift characters from the front of the string to be parsed. The top elements on a stack form a *handle* which may be reduced to a non-terminal element according to the rules defining the grammar.

In this lecture we explore how a machine can take a grammar and derives a parsing algorithm from it. This technology allows programmers to specify grammars easily.

As running example throughout this section, we use the following grammar $G = (T, N, R, S)$ where the set of terminal symbols is defined as

$$T ::= id \mid + \mid \$.$$

\$ is the "end of input" symbol. The set of non-terminal symbols has the form

$$N ::= S \mid E \mid T,$$

and the set of reduction rules R is given here.

$$\begin{array}{ll} (0) & S \rightarrow E\,\$ \\ (1) & E \rightarrow T + E \\ (2) & E \rightarrow T \\ (3) & T \rightarrow id \end{array}$$

S is the start symbol.

The goal of this note is three fold. First we show the reader an example of how to construct a parse table. Second we demonstrate the internal working of the parser with string $id + id + id\$$ as example, step by step. And third we define SLR grammars.

1 LR(0)

In a first experiment we construct a table assuming that this grammar is an LR(0) grammar, that is a grammar that does not require any look ahead in order to make a transition from one state to the next.

The parse table is constructed in the following way. We define a notion of state that captures the form of all top level handles that currently lie on top of the stack. Recall that a handle is the right hand side of a reduction rule in R, and in order to make sure that all of the intermediate steps are also captured by our notion of states, we define a cursor through inference rules, denoted by a dot.

The first state is computed by transitive recursive closure of all reduction rules that may potentially and eventually reduce after parsing a part (or all) of the input string.

$$\boxed{1} = \begin{array}{l} S \to .E\ \$ \\ E \to .T + E \\ E \to .T \\ T \to .id \end{array}$$

Next we consider what can happen if we parse the first element from the string. There are only two possibilities. Either it is a $+$, or it is an identifier id. The first case marks an error situation, the second case is very well possible. Therefore, we allow a transition from state $\boxed{1}$ to state $\boxed{5}$ labeled id. We remark that we name the states in accordance with Figure 3.24 in Appel's book.

$$\boxed{5} = \quad T \to id.$$

This transition clearly corresponds to a shift operation. But how do configurations of the machine that executes the parse look like? A configuration is a stack of alternating state names, and terminal (or non-terminal as we see soon) symbols; and the input string. The initial configuration of our parser is:

$$\text{Step 0:} \quad \boxed{1} \qquad id + id + id\$$

When shifting, we shift not only the terminal symbol, but also the new state of the parser on the stack.

$$\text{Step 1:} \quad \boxed{1}\, id\, \boxed{5} \qquad +id + id\$$

The next operation is a reduce operation using rule (3). The idea behind reducing is to pop off as many objects and states from the stack as the right hand side of rule (3) contains elements. For this example it is just 1 element. Then push terminal T on the stack.

$$\text{Step 1.5:} \quad \boxed{1}\, T \qquad +id + id\$$

But what state is the parser in now? To answer this question consider how the cursor moves in state 1, after parsing T. It behaves almost as during a shift operation, but this time the input symbol is a non-terminal. This is why this operation is called goto, and not shift. What happens, is that we have to define a new state.

$$\boxed{3} = \begin{array}{l} E \to T. + E \\ E \to T. \end{array}$$

and the goto operation pushes it onto the stack. In the diagram in Figure 3.24, the goto operation is labeled by T from state $\boxed{1}$ to state $\boxed{3}$. The final configuration on the stack has the form:

$$\text{Step 2:} \quad \boxed{1}\, T\, \boxed{3} \qquad +id + id\$$

The reduce and goto operation always follow each other, therefore we consider Step 1.5 only an auxiliary configuration. By continuously applying the same technique, one can easily determine all 6 states of this parser, and the appropriate transition relation.

In the parse table we write $s\,\boxed{\text{n}}$ if the parser performs a shift and transitions to state $\boxed{\text{n}}$. We write $r(i)$ if the parser is supposed to reduce using rule (i). We write $g\,\boxed{\text{n}}$ if after reducing, the parser transitions into state $\boxed{\text{n}}$, and we write a, if all of the input is succesfully consumed and a valid parse tree has been constructed. An empty entry in the parse table signals error.

In summary, there are three operations. Shift shifts a symbol and memorizes the new state on top of the stack, reduce pops off as many symbols and and states as required, followed immediately by a goto operation. Let's look at the complete parse of the example above.

2 Example

This is the complete parse run of a LR(0) parser, assuming that resolves the ambiguity of what to do in state $\boxed{3}$ under input $+$.

Step 0:	$\boxed{1}$	$id + id + id\$$	
Step 1:	$\boxed{1}\, id\, \boxed{5}$	$+id + id\$$	by $s5$
Step 2 :	$\boxed{1}\, T\, \boxed{3}$	$+id + id\$$	by $r3, g3$
Step 3:	$\boxed{1}\, T\, \boxed{3} + \boxed{4}$	$id + id\$$	by $s4$
Step 4:	$\boxed{1}\, T\, \boxed{3} + \boxed{4}\, id\, \boxed{5}$	$+id\$$	by $s5$
Step 5:	$\boxed{1}\, T\, \boxed{3} + \boxed{4}\, T\, \boxed{3}$	$+id\$$	by $r3, g3$
Step 6:	$\boxed{1}\, T\, \boxed{3} + \boxed{4}\, T\, \boxed{3} + \boxed{4}$	$id\$$	by $s4$
Step 7:	$\boxed{1}\, T\, \boxed{3} + \boxed{4}\, T\, \boxed{3} + \boxed{4}\, id\, \boxed{5}$	$\$$	by $s5$
Step 8:	$\boxed{1}\, T\, \boxed{3} + \boxed{4}\, T\, \boxed{3} + \boxed{4}\, T\, \boxed{3}$	$\$$	by $r3, g3$
Step 9:	$\boxed{1}\, T\, \boxed{3} + \boxed{4}\, T\, \boxed{3} + \boxed{4}\, E\, \boxed{6}$	$\$$	by $r2, g6$
Step 10:	$\boxed{1}\, T\, \boxed{3} + \boxed{4}\, E\, \boxed{6}$	$\$$	by $r1, g6$
Step 11:	$\boxed{1}\, E\, \boxed{2}$	$\$$	by $r1, g2$
Step 12:	accept		by a

3 SLR

The grammar G described in this paper is unfortunately not LR(0), because not all shift/reduce conflicts are decided. For example, in state

$$\boxed{3} = \begin{array}{l} E \to T. + E \\ E \to T. \end{array}$$

where + is the next input symbol, the parser may decide in favor for shifting even thought it needs to reduce. More precisely, with the shift operation the parser transitions into state

$$\boxed{4} = \begin{array}{l} E \to T + .E \\ E \to .T + E \\ E \to .T \\ T \to .id \end{array}$$

and with the reduce operation into state

$$\boxed{2} = \quad S \to E.\$.$$

	id	$+$	\$	E	T
$\boxed{1}$	$s\,\boxed{5}$			$g\,\boxed{2}$	$g\,\boxed{3}$
$\boxed{2}$			a		
$\boxed{3}$	$r(2)$	$s\,\boxed{4}, r(2)$	$r(2)$		
$\boxed{4}$	$s\,\boxed{5}$			$g\,\boxed{6}$	$g\,\boxed{3}$
$\boxed{5}$	$r(3)$	$r(3)$	$r(3)$		
$\boxed{6}$	$r(1)$	$r(1)$	$r(1)$		

This is intolerable, because back-tracking may be required in this situation. But how can we control this issue. The bad news is that $G \in LR(0)$, and therefore an $LR(0)$ parser will not do its job. The good news is that for this particular grammar the conflict can always be resolved. The crucial observation is that an transition into $\boxed{2}$ can only then be possible, if the terminal symbol considered is \$.

In fact we can compute the set of terminal symbols possibly following each non-terminal symbol. The algorithm for this operation is given on page 48 in Appel's book. Applied to this setting, we compute that

$$\begin{array}{rcl} \text{Follow}(S) & = & \{\} \\ \text{Follow}(E) & = & \{\$\} \\ \text{Follow}(T) & = & \{+, \$\} \end{array}$$

This information can be presented in form of a table where each $\times$ stands for the possibility to reduce in this state.

	id	$+$	\$
1			
2			
3			$\times$
4			
5		$\times$	$\times$
6			$\times$

If we merge this information with the parse table, we arrive at one that works. The resulting parsing table is also given in Figure 3.25 in Appel's book.

	id	$+$	\$	E	T
1	s 5			g 2	g 3
2			a		
3		s 4	$r(2)$		
4	s 5			g 6	g 3
5		$r(3)$	$r(3)$		
6			$r(1)$		