Building LR Tables

Recall the definition of a context-free grammar

\[ G \stackrel{\text{def}}{=} (V_I, V_N, V_G, \Pi) \]

where

\[ V \stackrel{\text{def}}{=} V_I \cup V_N \]
\[ V_I \cap V_N \stackrel{\text{def}}{=} \{\} \]
\[ V_G \stackrel{\text{def}}{=} \subseteq V_N \]
\[ \Pi \stackrel{\text{def}}{=} \subseteq V_N \times V^* \]

A rewriting

\[ \alpha B \gamma \leftarrow \alpha \beta \gamma \]

is allowed if

\[ B \leftarrow \beta \in \Pi \]

and is written to show the explicit CFG rule application as

\[ \alpha B \gamma \underleftarrow{B \leftarrow \beta} \alpha \beta \gamma \]

The language defined by a CFG \( G \) is the set of strings of input symbols that can be reduced to a goal.

\[ L(G) \stackrel{\text{def}}{=} \{\alpha \mid G \leftarrow^* \alpha \land G \in V_G\} \cap V_I^* \]

Text \( \tau \in L(G) \) (is syntactically correct) if

\[ G \xrightarrow{\rho} \tau \]

where

\[ \rho \in \Pi^* \]

The following operational definition of parsing forces a left-to-right order on the rule applications

\[ G \in V_G \Rightarrow C(G, \lambda) \]
\[ B \leftarrow \beta \in \Pi \land C(\sigma B, \tau) \Rightarrow C(\sigma \beta, \tau) \]
\[ a \in V_I \land C(\sigma a, \tau) \Rightarrow C(\sigma, a \tau) \]
\[ C(\lambda, \tau) \Rightarrow \tau \in L(G) \]

Suppose we have a sequence of \( n + 1 \) left-to-right rewritings (the application of the second rule in the definition above)
\[ G \xrightarrow{\tau_0} \sigma_n A_n \tau_n \ldots \xrightarrow{\tau_2} \sigma_2 A_2 \tau_2 \xrightarrow{\tau_1} \sigma_1 A_1 \tau_1 \xrightarrow{\tau_0} \tau_0 \]

where

\[
\begin{align*}
\tau_i & \in V_I' \\
A_i & \in V_N \\
\sigma_i & \in V^* \\
r_i & \in \Pi
\end{align*}
\]

then \( \rho = r_n \ldots r_2 r_1 r_0 \) is a left-to-right parse of \( \tau_0 \), and the set of values \( \sigma_i A_i \) are the values of the parse stack just after a substitution has been applied.

Consider, for a fixed \( G \), the set of all such parse stacks. The set is a language. It can be described by a new CFG \( G' \) in which the phrase names are designated by the notation [text].

\[
\begin{align*}
V_I' & \overset{\text{def}}{=} V \\
V_N' & \overset{\text{def}}{=} \{ [A \leftarrow \alpha] \mid A \leftarrow \alpha \beta \in \Pi \} \\
V_G' & \overset{\text{def}}{=} \Pi \\
\Pi' & \overset{\text{def}}{=} \{ [A \leftarrow \alpha B] \leftarrow [A \leftarrow \alpha] B \mid B \in V \land A \leftarrow \alpha B \gamma \in \Pi \} \\
& \quad \cup \{ [A \leftarrow \alpha] \leftarrow [B \leftarrow \lambda] \mid B \in V_N \land A \leftarrow \alpha B \gamma \in \Pi \} \\
& \quad \cup \{ [A \leftarrow \alpha] \leftarrow \lambda \mid A \leftarrow \alpha \in \Pi \} \\
G' & \overset{\text{def}}{=} (V_I', V_N', G', \Pi')
\end{align*}
\]

\( G' \) is a finite automaton. The start state is \([ G \leftarrow \lambda ] \). If one applies the NFA to DFA transformation, the resulting DFA can walk the concatenation of the parse stack and input text, reaching a final state when a rule can be applied.

This is the basis of the LR technology. The LR automaton walks the parse stack, pulling new text off the as yet unprocessed input as necessary. In a final state the automaton is temporarily abandoned, the newly identified reduction is applied to the top of the parse stack, and the process starts over at the bottom of the parse stack.

LR provides an implementation of the operational definition of parsing given above. The tricky part is that more than one final state can sometimes be reached by the DFA. Resolving the ambiguous situations with k-symbol lookahead is what LR(k) signifies.

There is an LR(0) DFA for any CFG. There are no languages of interest that can be unambiguously parsed using an LR(0) machine; at least LR(1) is necessary. As it will turn out, the LR(1) DFA is large enough raise concerns for memory occupancy of the LR(1) tables. Much research has gone into reducing the table size.