4.5 LR Grammars

While LL(1) grammars and top-down parsing techniques are easy to work with, they are not able to represent all of the structures found in many programming languages. For more general-purpose programming languages, we must use and LR(1) grammar and associated bottom-up parsing techniques.

LR(1) is the set of grammars that can be parsed via shift-reduce techniques with a single character of lookahead. LR(1) is a super-set of LL(1) and can accommodate left recursion and common left prefixes which are not permitted in LL(1). This enables us to express many programming constructs in a more natural way. (An LR(1) grammar must still be non-ambiguous, and it cannot have shift-reduce or reduce-reduce conflicts, which we will explain below.)

For example, Grammar \( G_{10} \) is an LR(1) grammar:

\[
\begin{align*}
1. & \quad P \rightarrow E \\
2. & \quad E \rightarrow E + T \\
3. & \quad E \rightarrow T \\
4. & \quad T \rightarrow \text{id} (E) \\
5. & \quad T \rightarrow \text{id}
\end{align*}
\]

We need to know the FIRST and FOLLOW sets of LR(1) grammars as well, so take a moment now and work out the sets for Grammar \( G_{10} \), using the same technique from section 4.3.3.
4.6 Shift-Reduce Parsing

LR(1) grammars must be parsed using the shift-reduce parsing technique. This is a bottom-up parsing strategy that begins with the tokens and looks for rules that can be applied to reduce sentential forms into non-terminals. If there is a sequence of reductions that leads to the start symbol, then the parse is successful.

A shift action consumes one token from the input stream and pushes it onto the stack. A reduce action applies one rule of the form \( A \rightarrow \alpha \) from the grammar, replacing the sentential form \( \alpha \) on the stack with the non-terminal \( A \). For example, here is a shift-reduce parse of the sentence \( \text{id(id+id)} \) using Grammar \( G_{10} \):

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id ( id + id) $</td>
<td>shift</td>
</tr>
<tr>
<td>id</td>
<td>( id + id ) $</td>
<td>shift</td>
</tr>
<tr>
<td>id ( id</td>
<td>+ id ) $</td>
<td>reduce T ( \rightarrow ) id</td>
</tr>
<tr>
<td>id ( T</td>
<td>+ id ) $</td>
<td>reduce T ( \rightarrow ) E</td>
</tr>
<tr>
<td>id ( E</td>
<td>+ id ) $</td>
<td>shift</td>
</tr>
<tr>
<td>id ( E +</td>
<td>id ) $</td>
<td>shift</td>
</tr>
<tr>
<td>id ( E + id) $</td>
<td>reduce T ( \rightarrow ) id</td>
<td></td>
</tr>
<tr>
<td>id ( E + T)</td>
<td>$</td>
<td>reduce E ( \rightarrow ) E + T</td>
</tr>
<tr>
<td>id ( E</td>
<td>$</td>
<td>shift</td>
</tr>
<tr>
<td>id ( E</td>
<td>$</td>
<td>reduce T ( \rightarrow ) id(E)</td>
</tr>
<tr>
<td>T</td>
<td>$</td>
<td>reduce E ( \rightarrow ) T</td>
</tr>
<tr>
<td>E</td>
<td>$</td>
<td>reduce E ( \rightarrow ) P</td>
</tr>
<tr>
<td>P</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

While this example shows that there exists a derivation for the sentence, it does not explain how each action was chosen at each step. For example, in the second step, we might have chosen to reduce \( \text{id} \) to \( T \) instead of shifting a left parenthesis. This would have been a bad choice, because there is no rule that begins with \( T ( \), but that was not immediately obvious without attempting to proceed further. To make these decisions, we must analyze LR(1) grammars in more detail.
4.7 The LR(0) Automaton

An LR(0) automaton represents all the possible rules that are currently under consideration by a shift-reduce parser. (The LR(0) automaton is also variously known as the canonical collection or the compact finite state machine of the grammar.) Figure 4.3 shows a complete automaton for Grammar $G_{10}$. Each box represents a state in the machine, connected by transitions for both terminals and non-terminal in the grammar.

Each state in the automaton consists of multiple items, which are rules augmented by a dot (.) that indicates the parser’s current position in that rule. For example, the configuration $E \rightarrow E . + T$ indicates that $E$ is currently on the stack, and $+ T$ is a possible next sequence of tokens.

The automaton is constructed as follows. State 0 is created taking the production for the start symbol ($P \rightarrow E$) and adding a dot at the beginning of the right hand side. This indicates that we expect to see a complete program, but have not yet consumed any symbols. This is known as the kernel of the state.

Kernel of State 0

\[
P \rightarrow . E
\]

Then, we compute the closure of the state as follows. For each item in the state with a non-terminal $X$ immediately to the right of the dot, we add all rules in the grammar that have $X$ as the left hand side. The newly added items have a dot at the beginning of the right hand side.

\[
\begin{align*}
P & \rightarrow . E \\
E & \rightarrow . E + T \\
E & \rightarrow . T
\end{align*}
\]

The procedure continues until no new items can be added:

Closure of State 0

\[
\begin{align*}
P & \rightarrow . E \\
E & \rightarrow . E + T \\
E & \rightarrow . T \\
T & \rightarrow . id ( E ) \\
T & \rightarrow . id
\end{align*}
\]
You can think of the state this way: It describes the initial state of the parser as expecting a complete program in the form of an \( E \). However, an \( E \) is known to begin with an \( E \) or a \( T \), and a \( T \) must begin with an \( \text{id} \). All of those symbols could represent the beginning of the program.

From this state, all of the symbols (terminals and non-terminals both) to the right of the dot are possible outgoing transitions. If the automaton takes that transition, it moves to a new state containing the matching items, with the dot moved one position to the right. The closure of the new state is computed, possibly adding new rules as described above.

For example, from state zero, \( E \), \( T \), and \( \text{id} \) are the possible transitions, because each appears to the right of the dot in some rule. Here are the states for each of those transitions:

Transition on \( E \):

\[
\begin{align*}
  P & \rightarrow E . \\
  E & \rightarrow E . + T
\end{align*}
\]

Transition on \( T \):

\[
E \rightarrow T .
\]

Transition on \( \text{id} \):

\[
\begin{align*}
  T & \rightarrow \text{id} . ( E ) \\
  T & \rightarrow \text{id} .
\end{align*}
\]

Figure 4.3 gives the complete LR(0) automaton for Grammar \( G_{10} \). Take a moment now to trace over the table and be sure that you understand how it is constructed.

No, really. Stop now and study the figure carefully before continuing.
Figure 4.3: LR(0) Automaton for Grammar $G_{10}$
The LR(0) automaton tells us the choices available at any step of bottom-up parsing. When we reach a state containing an item with a dot at the end of the rule, that indicates a possible reduction. A transition on a terminal that moves the dot one position to the right indicates a possible shift. While the LR(0) automaton tells us the available actions at each step, it does not always tell us which action to take.\footnote{The 0 in LR(0) indicates that it uses zero lookahead tokens, which is a way of saying that it does not consider the input before making a decision. In this sense, a pure LR(0) language is not very useful by itself.}

Two types of conflicts can appear in an LR grammar:

A shift-reduce conflict indicates a choice between a shift action and a reduce action. For example, state 4 offers a choice between shifting a left parenthesis and reducing by rule five:

\begin{center}
\begin{align*}
\text{Shift-Reduce Conflict:} & \\
T & \rightarrow \text{id} \cdot (E) \\
T & \rightarrow \text{id}
\end{align*}
\end{center}

A reduce-reduce conflict indicates that two distinct rules have been completely matched, and either one could apply. While Grammar $G_{10}$ does not contain any reduce-reduce conflicts, they commonly occur when a syntactic structure occurs at multiple layers in a grammar. For example, it is often the case that a function invocation can be a statement by itself, or an element within an expression. The automaton for such a grammar would contain a state like this:

\begin{center}
\begin{align*}
\text{Reduce-Reduce Conflict:} & \\
S & \rightarrow \text{id} \ (E) \\
E & \rightarrow \text{id} \ (E)
\end{align*}
\end{center}

The LR(0) automaton forms the basis of LR parsing, by telling us which actions are available in each state. But, it does not tell us which action to take or how to resolve shift-reduce and reduce-reduce conflicts. To do that, we must take into account some additional information.
4.8 SLR Parsing

Simple LR (SLR) parsing is a basic form of LR parsing in which we use FOLLOW sets to resolve conflicts in LR(0) automaton. In short, we take the reduction \( A \rightarrow \alpha \) only when the next token on the input is in FOLLOW(\( A \)). If a grammar can be parsed by this technique, we say it is an SLR grammar, which is a subset of LR(1) grammars.

For example, the shift-reduce conflict in state 4 of Figure 4.3 is resolved by consulting FOLLOW(\( T \)). If the next token is \( + \), \( , \) or $, then we reduce by rule \( T \rightarrow \text{id} \). If the next token is \( ( \), then we shift to state 5. If neither of those is true, then the input is invalid, and we emit a parse error.

These decisions are encoded in the SLR parse tables which are known historically as GOTO and ACTION. The tables are created as follows:

**SLR Parse Table Creation.**

Given a grammar \( G \) and corresponding LR(0) automaton, create tables ACTION\([s, a]\) and GOTO\([s, A]\) for all states \( s \), terminals \( a \), and non-terminals \( A \) in \( G \).

For each state \( s \):
- For each item like \( A \rightarrow \alpha . a \beta \)
  
  ACTION\([s, a]\) = **shift** to state \( t \) according to the LR(0) automaton.
- For each item like \( A \rightarrow \alpha . B \beta \)
  
  GOTO\([s, B]\) = **goto** state \( t \) according to the LR(0) automaton.
- For each item like \( A \rightarrow \alpha . a \beta \)
  
  For each terminal \( a \) in FOLLOW(\( A \)):
  
  ACTION\([s, a]\) = **reduce** by rule \( A \rightarrow \alpha \)

All remaining states are considered error states.

Naturally, each state in the table can be occupied by only one action. If following the procedure results in a table with more than one state in a given entry, then you can conclude that the grammar is not SLR. (It might still be LR(1) – more on that below.)
Here is the SLR parse table for Grammar $G_{10}$. Note carefully the states 2 and 4 where there is a choice between shifting and reducing. In state 2, a lookahead of + causes a shift, while a lookahead of $ results in a reduction $P \rightarrow E$ because $\$ is the only member of FOLLOW($P$).

<table>
<thead>
<tr>
<th>State</th>
<th>GOTO E T</th>
<th>ACTION id ( ) + $</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>G1 G8</td>
<td>S4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>G3</td>
<td>S4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>R2 R2 R2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>S5 R5 R5 R5</td>
</tr>
<tr>
<td>5</td>
<td>G6 G8</td>
<td>S4</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>S7 S2</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>R4 R4 R4</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>R3 R3 R3</td>
</tr>
</tbody>
</table>

Figure 4.4: SLR Parse Table for Grammar $G_{10}$
Now we are ready to parse an input by following the SLR parsing algorithm. The parse requires maintaining a stack of states in the LR(0) automaton, initially containing the start state $S_0$. Then, we examine the top of the stack and the lookahead token, and take the action indicated by the SLR parse table. On a shift, we consume the token and push the indicated state on the stack. On a reduce by $A \rightarrow \beta$, we pop states from stack corresponding to each of the symbols in $\beta$, then take the additional step of moving to state $\text{GOTO}[t, A]$. This process continues until we either succeed by reducing the start symbol, or fail by encountering an error state.

**SLR Parsing Algorithm.**

Let $S$ be a stack of LR(0) automaton states. Push $S_0$ onto $S$. Let $a$ be the first input token.

Loop:

- Let $s$ be the top of the stack.
  - If $\text{ACTION}[s, a]$ is accept:
    - Parse complete.
  - Else if $\text{ACTION}[s, a]$ is shift $t$:
    - Push state $t$ on the stack.
    - Let $a$ be the next input token.
  - Else if $\text{ACTION}[s, a]$ is reduce $A \rightarrow \beta$:
    - Pop states corresponding to $\beta$ from the stack.
    - Let $t$ be the top of stack
    - Push $\text{GOTO}[t, A]$ onto the stack.
  - Otherwise:
    - Halt with a parse error.
Here is an example of applying the SLR parsing algorithm to the program `id ( id + id )`. The first three steps are easy: a shift is performed for each of the first three tokens `id`. The fourth step reduces `T → id`. This causes state 4 (corresponding to the right hand side `id`) to be popped from the stack. State 5 is now at the top of the stack, and `GOTO[5, T] = 8`, so state 8 is pushed, resulting in a stack of `0 4 5 8`.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Symbols</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>id ( id + id ) $</td>
<td>shift 4</td>
</tr>
<tr>
<td>0 4</td>
<td>id</td>
<td>( id + id ) $</td>
<td>shift 5</td>
</tr>
<tr>
<td>0 4 5</td>
<td>id (</td>
<td>id + id ) $</td>
<td>shift 4</td>
</tr>
<tr>
<td>0 4 5 4</td>
<td>id ( id</td>
<td>+ id ) $</td>
<td>reduce T → id</td>
</tr>
<tr>
<td>0 4 5 8</td>
<td>id ( T</td>
<td>+ id ) $</td>
<td>reduce E → T</td>
</tr>
<tr>
<td>0 4 5 6</td>
<td>id ( E</td>
<td>+ id ) $</td>
<td>shift 2</td>
</tr>
<tr>
<td>0 4 5 6 2</td>
<td>id ( E +</td>
<td>id ) $</td>
<td>shift 4</td>
</tr>
<tr>
<td>0 4 5 6 2 4</td>
<td>id ( E + id</td>
<td>) $</td>
<td>reduce T → id</td>
</tr>
<tr>
<td>0 4 5 6 2 3</td>
<td>id ( E + T</td>
<td>) $</td>
<td>reduce E → E + T</td>
</tr>
<tr>
<td>0 4 5 6</td>
<td>id ( E</td>
<td>) $</td>
<td>shift 7</td>
</tr>
<tr>
<td>0 4 5 6 7</td>
<td>id ( E )</td>
<td>$</td>
<td>reduce T → id(E)</td>
</tr>
<tr>
<td>0 8</td>
<td>T</td>
<td>$</td>
<td>reduce E → T</td>
</tr>
<tr>
<td>0 1</td>
<td>E</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

(Although we show two columns for “Stack” and “Symbols”, they are simply two representations of the same information. The stack state `0 4 5 8` represents the parse state of `id ( T` and vice versa.)

It should now be clear that SLR parsing has the same algorithmic complexity as LL(1) parsing. Both techniques require a parsing table and a stack. At each step in both algorithms, it is necessary to only consider the current state and the next token on the input. The distinction is that each LL(1) parsing state considers only a single non-terminal, while each LR(1) parsing state considers a large number of possible configurations.
4.9 Non-SLR Grammars

SLR parsing is a good starting point for understanding the general principles of bottom up parsing. However, SLR is a subset of LR(1), and not all LR(1) grammars are SLR. For example, consider Grammar $G_{11}$ which allows for a statement to be a variable assignment, or an identifier by itself. Note that $\text{FOLLOW}(S) = \{\$\}$ and $\text{FOLLOW}(V) = \{|=\$\}$.

**Grammar $G_{11}$**

1. $S \rightarrow V = E$
2. $S \rightarrow \text{id}$
3. $V \rightarrow \text{id}$
4. $V \rightarrow \text{id} [ E ]$
5. $E \rightarrow V$

We need only build part of the LR(0) automaton to see the problem:

![LR(0) Automaton](image)

Figure 4.5: Part of LR(0) Automaton for Grammar $G_{11}$

In state 1, we can reduce by $S \rightarrow \text{id}$ or $V \rightarrow \text{id}$. However, both $\text{FOLLOW}(S)$ and $\text{FOLLOW}(V)$ contain $\$$, so we cannot decide which to take when the next token is end-of-file. Even using the $\text{FOLLOW}$ sets, there is still a reduce-reduce conflict. Therefore, Grammar $G_{11}$ is not an SLR grammar.

But, if we look more closely at the possible sentences allowed by the grammar, the distinction between the two becomes clear. Rule $S \rightarrow \text{id}$ would only be applied in the case where the complete sentence is $\text{id} \$$.

If any other character follows a leading $\text{id}$, then $V \rightarrow \text{id}$ applies. So, the grammar is not inherently ambiguous: we just need a more powerful parsing algorithm.
4.10 LR(1) Parsing

The LR(0) automaton is limited in power, because it does not track what tokens can actually follow a production. SLR parsing accommodates for this weakness by using FOLLOW sets to decide when to reduce. As shown above, this is not sufficiently discriminating to parse all valid LR(1) grammars.

Now we give the complete or “canonical” form of LR(1) parsing, which depends upon the LR(1) automaton. The LR(1) automaton is like the LR(0) automaton, except that each item is annotated with the set of tokens that could potentially follow it, given the current parsing state. This set is known as the lookahead of the item. The lookahead is always a subset of the FOLLOW of the relevant non-terminal.

The lookahead of the kernel of the start state is always \{\$\}. When computing the closure of an item \( A \rightarrow \alpha . B \beta \) the newly added rule \( B \rightarrow \gamma \) gets a lookahead of FIRST(\( \beta \)).

Here is an example for Grammar G_{11}. The kernel of the start state consists of the start symbol with a lookahead of $:

\[
\text{Kernel of State 0} \\
S \rightarrow . V = E (\$) \\
S \rightarrow . id (\$)
\]

The closure of the start state is computed by adding the rules for \( V \) with a lookahead of =, because = follows \( V \) in rule 1:

\[
\text{Closure of State 0} \\
S \rightarrow . V = E (\$) \\
S \rightarrow . id (\$) \\
V \rightarrow . id (=) \\
V \rightarrow . id [ E ] (=)
\]

Now suppose that we construct state 1 via a transition on the terminal id. The lookahead for each item is propagated to the new state:

\[
\text{Closure of State 1} \\
S \rightarrow \text{id . } (\$) \\
V \rightarrow \text{id . } (=) \\
V \rightarrow \text{id . [ E ] } (=)
\]
Now you can see how the lookahead solves the reduce-reduce conflict. When the next token on the input is $, we can only reduce by $S \rightarrow \text{id}$. When the next token is $=$, we can only reduce by $V \rightarrow \text{id}$. By tracking lookaheads in a more fine-grained manner than SLR, we are able to parse arbitrary LR(1) grammars.

**Exercise.**
Write out the complete LR(1) automaton for Grammar $G_{11}$. 
4.11 LALR(1) Parsing

The main downside to LR(1) parsing is that the LR(1) automaton can be much much larger than the LR(0) automaton. Any two states that have the same items but differ in lookahead sets for any items are considered to be different states. The result is enormous parse tables that consume large amounts of memory and slow down an otherwise-linear parsing algorithm.

LALR(1) parsing (lookahead-LR) is the practical answer to this problem. To construct an LALR(1) parser, we first create the LR(1) automaton, and then merge states that have the same core. The core of a state is simply the body of an item, ignoring the lookahead. When several LR(1) items are merged into one LALR(1) item, the LALR(1) lookahead is the union of the lookaheads of the LR(1) items.

LR(1) state A:
E $\rightarrow$ . E + T ($+$)
E $\rightarrow$ . T ($+$)

LR(1) state B:
E $\rightarrow$ . E + T ()+
E $\rightarrow$ . T ()+

Merged LALR state:
E $\rightarrow$ . E + T ($$)+
E $\rightarrow$ . T ($$)+

The resulting LALR automaton has the same number of states as the LR(0) automaton, but has more precise lookahead information available for each item. While this may seem a minor distinction, experience has shown this simple improvement to be highly effective at obtaining the efficiency of SLR parsing while accommodating a large number of practical grammars.

Exercise.
Starting from the LR(1) automaton, write out the LALR(1) automaton for Grammar $G_{11}$.