LR Parsing

Quick Review:

- top-down parsers build a parse tree from root to leaves
- bottom-up parsers build a parse tree from leaves to root
- \( LR(1) \) parsers: \((bottom \ up)\)
  - scan the input from left to right
  - build a rightmost derivation in reverse
  - use a single token lookahead to disambiguate
- have a simple, table-driven, \textit{shift-reduce} skeleton
- encode grammatical knowledge in tables

\textit{LR parsers are practical, efficient, and easy to build.}
LR(1) Parsing

The skeleton parser:

```plaintext
token = next_token()
repeat forever
    s = top of stack
    if action[s, token] = "shift \(s_i\)" then
        push token
        push \(s_i\)
        token = next_token()
    else if action[s, token] = "reduce \(A := \beta\)" then
        pop \(2 \cdot |\beta|\) symbols
        s = top of stack
        push \(A\)
        push goto[s, A]
    else if action[s, token] = "accept" then
        return
    else error()
```

This takes \(k\) shifts and \(l\) reduces, where \(k\) is the length of the input string and \(l\) depends on the grammar.

Equivalent to Figure 4.30, Aho, Sethi, and Ullman.
Example Tables

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id + * $</td>
<td>&lt;expr&gt; &lt;term&gt; &lt;factor&gt;</td>
</tr>
<tr>
<td>$S_0$</td>
<td>s4 — — — 1 2 3</td>
</tr>
<tr>
<td>$S_1$</td>
<td>— — — acc — — —</td>
</tr>
<tr>
<td>$S_2$</td>
<td>— s5 — r3 — — —</td>
</tr>
<tr>
<td>$S_3$</td>
<td>— r5 s6 r5 — — —</td>
</tr>
<tr>
<td>$S_4$</td>
<td>— r6 r6 r6 — — —</td>
</tr>
<tr>
<td>$S_5$</td>
<td>s4 — — — 7 2 3</td>
</tr>
<tr>
<td>$S_6$</td>
<td>s4 — — — — 8 3</td>
</tr>
<tr>
<td>$S_7$</td>
<td>— — — r2 — — —</td>
</tr>
<tr>
<td>$S_8$</td>
<td>— r4 — r4 — — —</td>
</tr>
</tbody>
</table>

The Grammar

1. $<\text{goal}> ::= <\text{expr}>$
2. $<\text{expr}> ::= <\text{term}> + <\text{expr}>$
3. $| <\text{term}>$
4. $<\text{term}> ::= <\text{factor}> * <\text{term}>$
5. $| <\text{factor}>$
6. $<\text{factor}> ::= \text{id}$
LR(1) Parsing

There are three commonly used algorithms to build tables for an “LR” parser:

1. SLR(1)
   - smallest class of grammars
   - smallest tables (number of states)
   - simple, fast construction

2. LR(1)
   - full set of LR(1) grammars
   - largest tables (number of states)
   - slow, large construction

3. LALR(1)
   - intermediate sized set of grammars
   - same number of states as SLR(1)
   - canonical construction is slow and large
   - better construction techniques exist

An LR(1) parser for either ALGOL or PASCAL has several thousand states, while an SLR(1) or LALR(1) parser for the same language may have several hundred states.
FIRST

For a string of grammar symbols $\alpha$, define $\text{FIRST}(\alpha)$ as

- the set of terminal symbols that begin strings derived from $\alpha$
- If $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$

$\text{FIRST}(\alpha)$ contains the set of tokens that are valid in the initial position in $\alpha$

To build $\text{FIRST}(X)$:

1. if $X$ is a terminal, $\text{FIRST}(X)$ is $\{X\}$
2. if $X ::= \epsilon$, then $\epsilon \in \text{FIRST}(X)$.
3. if $X ::= Y_1 Y_2 \cdots Y_k$ then put $\text{FIRST}(Y_1)$ in $\text{FIRST}(X)$
4. if $X$ is a non-terminal and $X ::= Y_1 Y_2 \cdots Y_k$, then
   - $a \in \text{FIRST}(X)$ if $a \in \text{FIRST}(Y_i)$ and $\epsilon \in \text{FIRST}(Y_j)$ for all $1 \leq j < i$.
   - (If $\epsilon \notin \text{FIRST}(Y_1)$, then $\text{FIRST}(Y_i)$ is irrelevant, for $1 < i$.)
**FOLLOW**

For a non-terminal $A$, define $\text{FOLLOW}(A)$ as

the set of terminals that can appear immediately to the right of $A$ in some sentential form

Thus, a non-terminal’s $\text{FOLLOW}$ set specifies the tokens that can legally appear after it.
A terminal symbol has no $\text{FOLLOW}$ set.

To build $\text{FOLLOW}(X)$:

1. place `eof` in $\text{FOLLOW}(<\text{goal}>)$
2. if $A ::= \alpha B \beta$, then put $\{\text{FIRST}(\beta) - \epsilon\}$ in $\text{FOLLOW}(B)$
3. if $A ::= \alpha B$ then put $\text{FOLLOW}(A)$ in $\text{FOLLOW}(B)$
4. if $A ::= \alpha B \beta$ and $\epsilon \in \text{FIRST}(\beta)$, then put $\text{FOLLOW}(A)$ in $\text{FOLLOW}(B)$
Example

For our example grammar, these sets are:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;goal&gt;</td>
<td>{ id, number }</td>
<td>{ eof }</td>
</tr>
<tr>
<td>&lt;expr&gt;</td>
<td>{ id, number }</td>
<td>{ eof }</td>
</tr>
<tr>
<td>&lt;term&gt;</td>
<td>{ id, number }</td>
<td>{ eof, +,- }</td>
</tr>
<tr>
<td>&lt;factor&gt;</td>
<td>{ id, number }</td>
<td>{ eof, +,-,*,/ }</td>
</tr>
<tr>
<td>+</td>
<td>{ + }</td>
<td>—</td>
</tr>
<tr>
<td>-</td>
<td>{ - }</td>
<td>—</td>
</tr>
<tr>
<td>*</td>
<td>{ * }</td>
<td>—</td>
</tr>
<tr>
<td>/</td>
<td>{ / }</td>
<td>—</td>
</tr>
<tr>
<td>id</td>
<td>{ id }</td>
<td>—</td>
</tr>
<tr>
<td>number</td>
<td>{ number }</td>
<td>—</td>
</tr>
</tbody>
</table>

*Computing these sets is the 1st step in building LR(1) tables*
LR(1) Grammars

Given these definitions, we can formally define an LR(1) grammar. An augmented grammar\(^\dagger\) \(G\) is LR(1) if the three conditions

1. \(\text{Start } \Rightarrow^* \alpha Aw \Rightarrow^* \alpha \beta w,\)

2. \(\text{Start } \Rightarrow^* \gamma Bx \Rightarrow^* \alpha \beta y,\)

3. \(\text{FIRST}(w) = \text{FIRST}(y)\)

imply that \(\alpha Ay = \gamma Bx.\)

(That is, \(\alpha = \gamma, A = B,\) and \(x = y.\))

To extend this to LR(\(k\)) grammars, we define \(\text{FIRST}_k(\alpha)\) as the leading \(k\) symbols that begin strings derived from \(\alpha.\)

The definition extends naturally by changing rule 3.

\(^\dagger\) An “augmented grammar” is one where the start symbol appears only on the lhs of productions.

For the rest of LR parsing, assume the grammar is augmented with a production \(S' := S.\)
LR(k) Items

The table construction algorithms use LR(k) items to represent the set of possible states in a parse.

An LR(k) item is a pair $[\alpha, \beta]$, where

- $\alpha$ is a production from $G$ with a $\bullet$ at some position in the rhs
- $\beta$ is a lookahead string containing $k$ symbols (terminals or $\text{eof}$)

Two cases of interest are $k = 0$ and $k = 1$.

$LR(0)$ items play a key role in the $SLR(1)$ table construction algorithm.

$LR(1)$ items play a key role in the $LR(1)$ and $LALR(1)$ table construction algorithms.
Viable prefix

A viable prefix is

1. a prefix of a right-sentential form that does not continue past
   the right end of the rightmost handle of that sentential form†, or
2. a prefix of a right-sentential form that can appear on the stack
   of a shift-reduce parser.

If the viable prefix is a proper prefix (that is, a handle), it is possible
to add terminals onto its end to form a right-sentential form.

As long as the prefix represented by the stack is viable, the
parser has not seen a detectable error.

† If the grammar is unambiguous, there is a unique rightmost handle.

LR(k) grammars are unambiguous. Operator grammars may be
ambiguous, but are still parsed with shift-reduce parsers.
Example

The ● indicates how much of an item we have seen at a given state in the parse.

\[ A ::= \bullet X Y Z \] indicates that the parser is looking for a string that can be derived from \( X Y Z \)

\[ A ::= X Y \bullet Z \] indicates that the parser has seen a string derived from \( X Y \) and is looking for one derivable from \( Z \)

\[ LR(0) \] Items: \hspace{1cm} \text{(no lookahead)}

\( A ::= X Y Z \) generates 4 \( LR(0) \) items.

1. \[ A ::= \bullet X Y Z \]
2. \[ A ::= X \bullet Y Z \]
3. \[ A ::= X Y \bullet Z \]
4. \[ A ::= X Y Z \bullet \]
LR(1) Items

What about LR(1) items?

- In an LR(1) item, all the lookahead strings are constrained to have length 1.
- An LR(1) item might look like \([A ::= X \bullet YZ, a]\).

Key Observations:

1. unambiguous grammar ⇒ unique rightmost derivation
2. handles appear on upper fringe of tree built by reverse rightmost derivation
   can keep fringe on the stack
3. while \(L(G)\) isn’t regular, the language of handles is, because there are only a finite number of handles
4. can recurse to match terms by leaving dfa state on the stack

*All the dfa knowledge is encoded in the Action and Goto tables*
LR(1) Items

The LR(1) table construction algorithm uses a specific set of sets of LR(1) items, called the **canonical collection of sets of LR(1) items** for a grammar $G$.

The canonical collection represents the set of valid states for the parser.

The items in each set of the canonical collection fall into two classes:

- **kernel items**: items where $\bullet$ is not at the left end of the rhs and $[S' := \bullet S, \text{eof}]$
- **non-kernel items**: all items where $\bullet$ is at the left end of rhs

Each item corresponds to a point in the parse.

To generate a **parser state** from a kernel item, we take its closure.

$\Rightarrow$ if $[A ::= \alpha \bullet B \beta, a] \in I_j$, then, in state $j$, the parser might next see a string derivable from $B \beta$

$\Rightarrow$ to form its closure, add all items of the form $[B ::= \bullet \gamma, \alpha] \in G$
LR(1) Items

What’s the point of the lookahead symbols?

- carry them along to allow us to choose correct reduction when there is any choice
- lookaheads are bookkeeping, unless item has $\bullet$ at right end.
  - in $[A ::= X \bullet YZ, a]$, $a$ has no direct use
  - in $[A ::= XYZ\bullet, a]$, $a$ is useful
- allows use of grammars that are not uniquely invertible

Recall, the $SLR(1)$ construction uses $LR(0)$ items!

The point:

For $[A ::= a\bullet,a]$ and $[B ::= a\bullet,b]$, we can decide between reducing to $A$ and to $B$ by looking at limited right context!
LR(1) Items

The canonical collection of LR(1) items:

- set of items derivable from \([S' ::= \bullet S, \text{eof}]\)
- set of all items that can derive the final configuration

The set of sets where, for each item \([A ::= X \bullet Y, u]\), there exists a rightmost derivation

\[ S' \Rightarrow^* rAst \Rightarrow rxyst \Rightarrow^* r'x'ut \]

where \(rx \Rightarrow^* r'x'\) and \(ys \Rightarrow^* u\).

To construct the canonical collection we need two functions:

- closure\((I)\)
- goto\((I, X)\)
Closure(I)

Given an item \([A ::= \alpha \cdot B\beta, a]\), its closure contains the item and any other items that can generate legal substrings to follow \(\alpha\). Thus, if the parser has viable prefix \(\alpha\) on its stack, the input should reduce to \(B\beta\) (or \(\gamma\) for some item \([B ::= \bullet\gamma, b]\) in the closure).

To compute closure(I)

```plaintext
function closure(I)
    add = 1
    while (add \neq 0)
        add = 0
        for each item \([A ::= \alpha \cdot B\beta, a] \in I\),
            each production \(B ::= \gamma \in G'\),
            and each terminal \(b \in \text{FIRST}(\beta a)\),
            if \([B ::= \bullet\gamma, b] \notin I\) then
                add \([B ::= \bullet\gamma, b]\) to I
                add = 1
    return I
```

Aho, Sethi, and Ullman, Figure 4.38
**Goto**($I, X$)

Let $I$ be a set of $LR(1)$ items and $X$ be a grammar symbol. Then, goto($I, X$) is the closure of the set of all items

$$[A ::= \alpha X \bullet \beta, a]$$

such that $[A ::= \alpha \bullet X \beta, a] \in I$

If $I$ is the set of valid items for some viable prefix $\gamma$, then goto($I, X$) is the set of valid items for the viable prefix $\gamma X$.

goto($I, X$) represents state after recognizing $X$ in state $I$.

To compute goto($I, X$)

```latex
function goto(I, X)
  let J be the set of items $[A ::= \alpha X \bullet \beta, a]$
  such that $[A ::= \alpha \bullet X \beta, a] \in I$
  return closure(J)
```

Aho, Sethi, and Ullman, Figure 4.38
Collection of Sets of $LR(1)$ Items

We start the construction of the collection of sets of $LR(1)$ items with the item $[S' ::= \bullet S, \text{eof}]$, where

$S'$ is the start symbol of the augmented grammar $G'$

$S$ is the start symbol of $G$, and

$\text{eof}$ is the right end of string marker

To compute the collection of sets of $LR(1)$ items

procedure items($G'$)

\[
C = \{\text{closure}([S' ::= \bullet S, \text{eof}])\}
\]

add = 1

while (add $\neq$ 0)

\[
\text{add} = 0
\]

for each set of items $I$ in $C$ and each grammar symbol $X$ such that $\text{goto}(I, X) \neq \emptyset$ and $\text{goto}(I, X) \not\in C$

\[
\text{add} \text{ goto}(I, X) \text{ to } C
\]

\[
\text{add} = 1
\]

Aho, Sethi, and Ullman, Figure 4.38
Example

Step 1

\[ I_0 \leftarrow \{ [g \rightarrow \bullet, e, \text{eof}] \} \]

\[ I_0 \leftarrow \text{closure}(I_0) \]

\{ [g \rightarrow \bullet, e, \text{eof}], [e \rightarrow \bullet, t + e, \text{eof}], [e \rightarrow \bullet, t, \text{eof}], [t \rightarrow \bullet, f * t, +], [t \rightarrow \bullet, f, t, \text{eof}], [t \rightarrow \bullet, f, +], [t \rightarrow \bullet, f, \text{eof}] \}

Iteration 1

\[ I_1 \leftarrow \text{goto}(I_0, e) \]

\[ I_2 \leftarrow \text{goto}(I_0, t) \]

\[ I_3 \leftarrow \text{goto}(I_0, f) \]

\[ I_4 \leftarrow \text{goto}(I_0, \text{id}) \]

Iteration 2

\[ I_5 \leftarrow \text{goto}(I_2, +) \]

\[ I_6 \leftarrow \text{goto}(I_3, *) \]

Iteration 3

\[ I_7 \leftarrow \text{goto}(I_5, e) \]

\[ I_8 \leftarrow \text{goto}(I_6, t) \]
Example

\[ \begin{align*}
I_0: & \quad [g \rightarrow \bullet e,\text{eof}], [e \rightarrow \bullet t + e,\text{eof}], [e \rightarrow \bullet t,\text{eof}], \\
     & \quad [t \rightarrow \bullet f * t,\{+,\text{eof}\}], [t \rightarrow \bullet f,\{+,\text{eof}\}], [f \rightarrow \bullet id,\{+,\text{eof}\}] \\
I_1: & \quad [g \rightarrow e \bullet,\text{eof}] \\
I_2: & \quad [e \rightarrow t \bullet,\text{eof}], [e \rightarrow t \bullet + e,\text{eof}] \\
I_3: & \quad [t \rightarrow f \bullet,\{+,\text{eof}\}], [t \rightarrow f \bullet * t,\{+,\text{eof}\}] \\
I_4: & \quad [f \rightarrow id \bullet,\{+,\star,\text{eof}\}] \\
I_5: & \quad [e \rightarrow t + \bullet e,\text{eof}], [e \rightarrow \bullet t + e,\text{eof}], [e \rightarrow \bullet t,\text{eof}], \\
     & \quad [t \rightarrow \bullet f * t,\{+,\text{eof}\}], [t \rightarrow \bullet f,\{+,\text{eof}\}], \\
     & \quad [f \rightarrow \bullet id,\{+,\star,\text{eof}\}] \\
I_6: & \quad [t \rightarrow f * t,\{+,\text{eof}\}], [t \rightarrow \bullet f * t,\{+,\text{eof}\}], \\
     & \quad [t \rightarrow \bullet f,\{+,\text{eof}\}], [f \rightarrow \bullet id,\{+,\star,\text{eof}\}] \\
I_7: & \quad [e \rightarrow t + e \bullet,\text{eof}] \\
I_8: & \quad [t \rightarrow f \star t \bullet,\{+,\text{eof}\}] \\
\end{align*} \]
LR(1) Table Construction

The Algorithm

1. construct the collection of sets of LR(1) items for $G'$.

2. State $i$ of the parser is constructed from $I_i$.

   (a) if $[A \rightarrow \alpha \cdot a\beta, b] \in I_i$ and goto($I_i, a$) = $I_j$, then set
       \text{action}[i,a] to “shift $j$”. ($a$ must be a terminal)

   (b) if $[A \rightarrow \alpha\cdot, a] \in I_i$, then set \text{action}[i,a] to
       “reduce $A \rightarrow \alpha$”.

   (c) if $[S' \rightarrow S\cdot, \text{eof}] \in I_i$, then set \text{action}[i,\text{eof}] to
       “accept”.

3. If goto($I_i, A$) = $I_j$, then set goto[$i,A$] to $j$.

4. All other entries in \text{action} and \text{goto} are set to “error”

5. The initial state of the parser is the state constructed from the
   set containing the item $[S' \rightarrow \cdot S, \text{eof}]$.

Aho, Sethi, and Ullman, Algorithm 4.10
What can go wrong?

*Rules 2a, 2b, and 2c can multiply define a position in the action table. In this case, the grammar is not LR(1).*

Two cases arise:

*shift/reduce* This is called a *shift/reduce* conflict. In general, it indicates an ambiguous construct in the grammar.

- can modify the grammar to eliminate it
- can resolve in favor of shifting

classical example: dangling else

*reduce/reduce* This is called a *reduce/reduce* conflict. Again, it indicates an ambiguous construct in the grammar.

- often, no simple resolution
- parse a nearby language

classical example: PL/I call and subscript
LALR(1) Parsing

Define the core of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.

Thus, the two sets

- \( \{ [A \Rightarrow \alpha \cdot \beta, a], [A \Rightarrow \alpha \cdot \beta, b] \} \), and
- \( \{ [A \Rightarrow \alpha \cdot \beta, c], [A \Rightarrow \alpha \cdot \beta, d] \} \)

have the same core.

Key Idea:

If two sets of LR(1) items, \( I_i \) and \( I_j \), have the same core, we can merge the states that represent them in the action and goto tables.
LALR(1) Table Construction

To construct LALR(1) parsing tables, we can insert a single step into the LR(1) algorithm.

(1.5) For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union.

The goto function must be updated to reflect the replacement sets.

The resulting algorithm has large space requirements
LALR(1) Table Construction

A more space efficient algorithm can be derived by observing that:

- we can represent $I_i$ by its kernel, those items that are either the initial item $[S' \rightarrow \bullet S, \text{eof}]$ or do not have the $\bullet$ at the left end of the rhs.

- we can compute shift, reduce, and goto actions for the state derived from $I_i$ directly from kernel($I_i$).

This method avoids building the complete collection of sets of LR(1) items.
LR($k$) Languages

Grammars

SLR(1) LALR(1) LR(1) LR($k$)

Languages

SLR(1) LALR(1) LR(1) LR($k$)