Shift-reduce parsing

Grammars that are often used to construct shift-reduce parsers:

- operator grammars (will postpone discussion until later)
- LR(1) grammars
  - canonical LR(1) grammars
  - simple LR(1) grammars (SLR(1))
  - lookahead LR(1) grammars (LALR(1))

Grammars use different methods or levels of "context" information to detect handle.

LR(1), SLR(1) and LALR(1)) grammars use finite automata (NFAs or DFAs) to recognize viable prefixes and store "context" information.

LR(k) grammars

Informally, we say that a grammar $G$ is LR(k) if, given a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = w,$$

we can, for each right-sentential form in the derivation,

1. isolate the handle of each right-sentential form, and
2. determine the production by which to reduce by scanning $\gamma_i$ from left to right, going at most $k$ symbols beyond the right end of the handle of $\gamma_i$.

Theorem: A language $L$ has an LR(0) grammar iff

- $L$ is deterministic
- no proper prefix of a word in $L$ is in $L$ (prefix property)

What properties does a language have that can be recognized by an LL(0) grammar?

Why study LR(1) grammars?

- All context-free, deterministic languages have an LR(1) grammar. Therefore LR grammars describe a proper superset of the languages recognized by LL (predictive) parsers.
- LR grammars are the most general grammars that can be parsed by a non-backtracking, shift-reduce parser.
- Efficient shift-reduce parsers can be implemented for LR(1) grammars — time proportional to tokens + reductions.
- Easy to build since table construction can be automated.
- LR parsers detect an error as soon as possible in a left-to-right scan of the input.
- Everyone’s favorite parser (EFP) — tools widely available (example: yacc).

Table-driven LR(1) parsing

A table-driven LR(1) parser looks like

Stack two items per state: state and symbol
We’ll learn how to build these tables by hand!
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 := β" then
        pop 2 | β | 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, \( l \) reduces, and 1 accept, where \( k \) is the length of the input string and \( l \) is the length of the reverse rightmost derivation.

**Note:** Equivalent to Figure 4.30, Aho, Sethi, and Ullman

---

LR parsing

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

1. **SLR(1)** = LR(0) + FOLLOW
   - 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.

---

Example tables

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id + *</td>
<td>acc</td>
</tr>
<tr>
<td>s4</td>
<td>s5</td>
</tr>
<tr>
<td>S1</td>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
<td>r2</td>
</tr>
<tr>
<td>S4</td>
<td>r3</td>
</tr>
<tr>
<td>S5</td>
<td>r4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr</td>
<td>term</td>
</tr>
<tr>
<td>term</td>
<td>factor</td>
</tr>
<tr>
<td>factor</td>
<td>id</td>
</tr>
</tbody>
</table>

---

Viable prefix

A viable prefix is

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

If the viable prefix includes the 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.*

---

\(^1\) We assume that the grammar is unambiguous.
**SLR(1) parsing**

Viable prefix of a right-sentential form:
- contains both terminals and nonterminals
- can be recognized with NFA or DFA

Building a SLR parser
- construct DFA for recognizing viable prefixes
- augment with FOLLOW to disambiguate actions

States in the NFA are LR(0) items
States in the DFA are sets of LR(0) items (subset construction)

**Note:** An “augmented grammar” is one where the start symbol appears only on the LHS of productions. For the rest of LR parsing, we will assume the grammar is augmented with a production $S' := S$

---

**Canonical LR(0) items**

The SLR(1) table construction algorithm uses a specific set of sets of LR(0) items.

These sets are called the canonical collection of sets of LR(0) items for a grammar $G$.

The canonical collection corresponds to the set of states of the DFA that recognizes viable prefixes. Each state is the set of valid LR(0) items at a particular point in the parse.

The LR(0) item $[A := \beta_1 \bullet \beta_2]$ is valid for a viable prefix $\alpha \beta_1$ if there is a derivation $S' \Rightarrow^*_R \alpha A \omega \Rightarrow^*_R \alpha \beta_1 \beta_2 \omega$.

In general, an item will be valid for many viable prefixes.

---

**LR(0) items**

An LR(0) item is a string $[\alpha]$, where

$\alpha$ is a production from $G$ with a $\bullet$ at some position in the RHS

The $\bullet$ indicates how much of an item we have seen at a given state in the parsing process.

$[A := \bullet XY Z]$ indicates that the parser is looking for a string that can be derived from $XY Z$

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

**LR(0) Items**  
(no lookahead)

$A := XY Z$ generates 4 LR(0) items.

1. $[A := \bullet XY Z]$
2. $[A := X \bullet Y Z]$
3. $[A := XY \bullet Z]$
4. $[A := XY Z \bullet]$

---

**Canonical Collection of LR(0) items**

To construct the canonical collection we need two functions:

- $\text{closure}(I)$

  if $[A := a \bullet B \beta] \in I_j$, 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 $[B := \bullet \gamma] \in G$

- $\text{GOTO}(I, X)$

  If $I$ is the set of items that are valid for some viable prefix $\gamma$, then $\text{GOTO}(I, X)$ is the set of items that are valid for the viable prefix $\gamma X$. 
Closure($I$)

Given an item [$A ::= \alpha \bullet B\beta$], 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 other item [$B ::= \bullet\gamma$] in the closure).

To compute closure($I$)

function closure($I$)
   repeat
      new_item ← false
      for each item [$A ::= \alpha \bullet B\beta$] ∈ $I$,
         each production $B ::= \gamma$ ∈ $G'$
         if [$B ::= \bullet\gamma$] ∉ $I$ then
            add [$B ::= \bullet\gamma$] to $I$
            new_item ← true
         endif
      until (new_item = false)
   return $I$

Goto($I, X$)

Let $I$ be a set of $LR(0)$ items and $X$ be a grammar symbol.

Then, $GOTO(I, X)$ is the closure of the set of all items

[$A ::= \alpha X \bullet \beta$] such that [$A ::= \alpha \bullet X\beta$] ∈ $I$

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($I, X$)

function goto($I, X$)
   J ← set of items [$A ::= \alpha X \bullet \beta$]
      such that [$A ::= \alpha \bullet X\beta$] ∈ $I$
   J' ← closure($J$)
   return $J'$

Collection of sets of $LR(0)$ items

We start the construction of the collection of sets of $LR(0)$ items with the item [$S' ::= \bullet S$], where

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

To compute the collection of sets of $LR(0)$ items

procedure items($G'$)
   $S_0$ ← closure([[$S' ::= \bullet S$]])
   Items ← { $S_0$ }
   ToDo ← { $S_0$ }
   while ToDo not empty do
      remove $S$ from ToDo
      for each grammar symbol $X$ do
         $S_{\text{new}}$ ← goto($S, X$)
         if $S_{\text{new}}$ is a new state then
            Items ← Items ∪ { $S_{\text{new}}$ }
            ToDo ← ToDo ∪ { $S_{\text{new}}$ }
         endif
      endfor
   endwhile
   return Items

LR(0) machines

LR(0) DFA

- states – canonical sets of LR(0) items
- edges – goto transitions
- recognizes all viable prefixes
- no lookahead

Reducing a handle (rhs of production) to a nonterminal can be viewed as:

- returning to state at beginning of handle
- making transition on nonterminal for this state

To return to state at beginning of the handle, we must use the stack to store the state!
SLR(1) tables

SLR(1) parser

- augment LR(0) machine
- add FOLLOW information using one token of lookahead
- encoded as ACTION, GOTO tables

ACTION table

- for each [state, lookahead] pair
- have we reached end of handle?
- if not, shift
- if at end of handle, reduce
- may also accept or error
- use lookahead to guide decision

GOTO table

- for each [state, nonterminal] pair
- pick state to go to after reduction

---

Example LR(0) states

\[ S_0: \quad [S' \overset{\epsilon}{=} \bullet E], \]
\[ [E \overset{\epsilon}{=} \bullet T + E], \]
\[ [E \overset{\epsilon}{=} \bullet T], \]
\[ [T \overset{\epsilon}{=} \bullet \text{id}] \]

\[ S_1: \quad [S' \overset{\epsilon}{=} \bullet E] \]

\[ S_2: \quad [E \overset{\epsilon}{=} \bullet T + E], \]
\[ [E \overset{\epsilon}{=} \bullet T], \]

\[ S_3: \quad [T \overset{\epsilon}{=} \bullet \text{id}] \]

\[ S_4: \quad [E \overset{\epsilon}{=} \bullet T + E], \]
\[ [E \overset{\epsilon}{=} \bullet T + E], \]
\[ [E \overset{\epsilon}{=} \bullet T], \]
\[ [T \overset{\epsilon}{=} \bullet \text{id}] \]

\[ S_5: \quad [E \overset{\epsilon}{=} \bullet T + E] \]

---

SLR(1) table construction

The Algorithm

1. construct the collection of sets of LR(0) items for \( G' \).
2. State \( i \) of the parser is constructed from \( I_i \).
   (a) if \( [A := \alpha \bullet a \beta] \in I_i \) and \( \text{goto}(I_i, a) = I_j \), then set \( \text{ACTION}[i, a] \text{ to "shift } j\text{" (}a\text{ must be a terminal)} \)
   (b) if \( [A := \alpha \bullet] \in I_i \), then set \( \text{ACTION}[i, a] \text{ to "reduce } A := \alpha \text{" for all } a \text{ in } \text{FOLLOW}(A) \).
   (c) if \( [S' := \bullet S] \in I_i \), then set \( \text{ACTION}[i, \epsilon] \text{ to "accept"}. \)
3. If \( \text{goto}(I_i, A) = I_j \), then set \( \text{GOTO}[i, A] \text{ to } j \).
4. All other entries in ACTION and GOTO are set to "error"
5. The initial state of the parser is the state constructed from the set containing the item \([S' := \bullet S]\).
Example GOTO function

Start

\[
S_0 \leftarrow \text{closure ( } \{ [ S := \bullet \ E ] \} )
\]

Iteration 1

\[
goto(S_0, E) = S_1
\]
\[
goto(S_0, T) = S_2
\]
\[
goto(S_0, \text{id}) = S_3
\]

Iteration 2

\[
goto(S_2, +) = S_4
\]

Iteration 3

\[
goto(S_4, \text{id}) = S_3
\]
\[
goto(S_4, E) = S_5
\]
\[
goto(S_4, T) = S_2
\]

Example ACTION and GOTO tables

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{id} + \text{eof}</td>
<td>E T</td>
</tr>
<tr>
<td>\text{shift } 3</td>
<td>\text{accept } 1 \text{ 2}</td>
</tr>
<tr>
<td>\text{shift } 4</td>
<td>\text{reduce } 2</td>
</tr>
<tr>
<td>\text{reduce } 3</td>
<td>\text{reduce } 3</td>
</tr>
<tr>
<td>\text{shift } 3</td>
<td>\text{reduce } 1 \text{ 5}</td>
</tr>
</tbody>
</table>

Stack | Input | Action |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$0$</td>
<td>\text{id + id eof}</td>
<td>\text{shift } 3</td>
</tr>
<tr>
<td>$0$ $\text{id}$ $3$</td>
<td>\text{+ id eof}</td>
<td>\text{reduce } 3 (T := \text{id})</td>
</tr>
<tr>
<td>$0$ $T$ $2$</td>
<td>\text{+ id eof}</td>
<td>\text{shift } 4</td>
</tr>
<tr>
<td>$0$ $T$ $2$ $+$ $4$</td>
<td>\text{id eof}</td>
<td>\text{shift } 3</td>
</tr>
<tr>
<td>$0$ $T$ $2$ $+$ $4$ $\text{id}$ $3$</td>
<td>\text{eof}</td>
<td>\text{reduce } 3 (T := \text{id})</td>
</tr>
<tr>
<td>$0$ $T$ $2$ $+$ $4$ $T$ $2$</td>
<td>\text{eof}</td>
<td>\text{reduce } 2 (E := T)</td>
</tr>
<tr>
<td>$0$ $T$ $2$ $+$ $4$ $E$ $5$</td>
<td>\text{eof}</td>
<td>\text{reduce } 1 (E := T + E)</td>
</tr>
<tr>
<td>$0$ $E$ $1$</td>
<td>\text{eof accept}</td>
<td></td>
</tr>
</tbody>
</table>