Bottom-up parsing

Recall

For a grammar $G$, with start symbol $S$, any string $\alpha$ such that $S \Rightarrow^* \alpha$ is called a sentential form

- If $\alpha \in V_t^*$, then $\alpha$ is called a sentence in $L(G)$
- Otherwise it is just a sentential form (not a sentence in $L(G)$)

A left-sentential form is a sentential form that occurs in the leftmost derivation of some sentence.

A right-sentential form is a sentential form that occurs in the rightmost derivation of some sentence.

Goal:

Given an input string $w$ and a grammar $G$, construct a parse tree by starting at the leaves and working to the root.

The parser repeatedly matches a right-sentential form from the language against the tree's upper frontier.

At each match, it applies a reduction to build on the frontier:

- each reduction matches an upper frontier of the partially built tree to the RHS of some production
- each reduction adds a node on top of the frontier

The final result is a rightmost derivation, in reverse.

Example

Consider the grammar

1. $S \rightarrow aABe$
2. $A \rightarrow Abc$
3. $| b$
4. $B \rightarrow d$

and the input string abbcde

<table>
<thead>
<tr>
<th>Prod'n.</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>a b bcde</td>
</tr>
<tr>
<td>2</td>
<td>a Abc de</td>
</tr>
<tr>
<td>4</td>
<td>aA d e</td>
</tr>
<tr>
<td>1</td>
<td>aABe</td>
</tr>
<tr>
<td></td>
<td>S</td>
</tr>
</tbody>
</table>

The trick appears to be scanning the input and finding valid sentential forms.

Handles

What are we trying to find?

A substring $\alpha$ of the tree's upper frontier that:

matches some production $A \rightarrow \alpha$ where reducing $\alpha$ to $A$ is one step in the reverse of a rightmost derivation

We call such a string a handle.

Formally:

a handle of a right-sentential form $\gamma$ is a production $A \rightarrow \beta$ and a position in $\gamma$ where $\beta$ may be found and replaced by $A$ to produce the previous right-sentential form in a rightmost derivation of $\gamma$

i.e., if $S \Rightarrow^+_r \alpha Aw \Rightarrow^+_r \alpha \beta w$ then $A \rightarrow \beta$ in the position following $\alpha$ is a handle of $\alpha \beta w$

Because $\gamma$ is a right-sentential form, the substring to the right of a handle contains only terminal symbols.
The handle $A \rightarrow \beta$ in the parse tree for $\alpha \beta w$

**Theorem:**

If $G$ is unambiguous then every right-sentential form has a unique handle.

**Proof:** (by definition)

1. $G$ is unambiguous $\Rightarrow$ rightmost derivation is unique
2. $\Rightarrow$ a unique production $A \rightarrow \beta$ applied to take $\gamma_{i-1}$ to $\gamma_i$
3. $\Rightarrow$ a unique position $k$ at which $A \rightarrow \beta$ is applied
4. $\Rightarrow$ a unique handle $A \rightarrow \beta$

**Example**

The left-recursive expression grammar (original form)

\[
\begin{align*}
1 & \quad <\text{goal}> \ ::= \ <\text{expr}> \\
2 & \quad <\text{expr}> \ ::= \ <\text{expr}> + <\text{term}> \\
3 & \quad \quad \quad \ | \quad <\text{expr}> - <\text{term}> \\
4 & \quad <\text{term}> \ ::= \ <\text{term}> * <\text{factor}> \\
5 & \quad \quad \quad \ | \quad <\text{term}> / <\text{factor}> \\
6 & \quad <\text{factor}> \ ::= \ <\text{factor}> * <\text{factor}> \\
7 & \quad \quad \quad \ \ | \quad \quad \quad \ <\text{factor}> \\
8 & \quad <\text{factor}> \ ::= \ \text{num} \\
9 & \quad \quad \quad \ \ | \quad \quad \quad \ \text{id}
\end{align*}
\]

<table>
<thead>
<tr>
<th>Prod'n.</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>$&lt;\text{goal}&gt;$</td>
</tr>
<tr>
<td>1</td>
<td>$&lt;\text{expr}&gt;$</td>
</tr>
<tr>
<td>2</td>
<td>$&lt;\text{expr}&gt; - &lt;\text{term}&gt;$</td>
</tr>
<tr>
<td>3</td>
<td>$&lt;\text{expr}&gt; - &lt;\text{term}&gt; * &lt;\text{factor}&gt;$</td>
</tr>
<tr>
<td>4</td>
<td>$&lt;\text{expr}&gt; - &lt;\text{term}&gt; * \text{id}$</td>
</tr>
<tr>
<td>5</td>
<td>$&lt;\text{expr}&gt; - &lt;\text{factor}&gt; * \text{id}$</td>
</tr>
<tr>
<td>6</td>
<td>$&lt;\text{expr}&gt; - \text{num} * \text{id}$</td>
</tr>
<tr>
<td>7</td>
<td>$&lt;\text{term}&gt; - \text{num} * \text{id}$</td>
</tr>
<tr>
<td>8</td>
<td>$\text{id} - \text{num} * \text{id}$</td>
</tr>
</tbody>
</table>

**Handle-pruning**

The process to construct a bottom-up parse is called handle-pruning.

To construct a rightmost derivation

\[ S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n \]

we set $i$ to $n$ and apply the following simple algorithm

for $i = n$ down to 1
1. find the handle $A_i \rightarrow \beta_i$ in $\gamma_i$
2. replace $\beta_i$ with $A_i$ to generate $\gamma_{i-1}$

This takes $2n$ steps, where $n$ is the length of the derivation
One scheme to implement a handle-pruning, bottom-up parser is called a *shift-reduce* parser.

Shift-reduce parsers use a *stack* and an *input buffer*

1. **initialize stack with $**

2. Repeat until the top of the stack is the goal symbol and the input token is $**

   (a) **find the handle**
   - if we don’t have a handle on top of the stack, *shift* an input symbol onto the stack

   (b) **prune the handle**
   - if we have a handle $A \rightarrow \beta$ on the stack, *reduce*
     i. pop $|\beta|$ symbols off the stack
     ii. push $A$ onto the stack

---

### Example: back to $x - 2 \ast y$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
</table>
| $\>$ | id - num * id | shift $9$
| $\langle id \rangle$ | - num * id | reduce $7$
| $\langle factor \rangle$ | - num * id | shift $4$
| $\langle expr \rangle$ | - num * id | shift $\langle factor \rangle$
| $\langle term \rangle$ | - id | reduce $8$
| $\langle expr \rangle$ | - $\langle factor \rangle$ | reduce $7$
| $\langle term \rangle$ | - $\langle term \rangle$ | reduce $5$
| $\langle expr \rangle$ | - $\langle term \rangle$ | reduce $3$
| $\langle expr \rangle$ | - $\langle term \rangle$ | reduce $1$
| $\langle goal \rangle$ | | accept $1$

---

### Shift-reduce parsing

**Shift-reduce parsers are simple to understand**

A shift-reduce parser has just four canonical actions:

1. **shift** — next input symbol is shifted onto the top of the stack

2. **reduce** — right end of handle is on top of stack; locate left end of handle within the stack; pop handle off stack and push appropriate non-terminal LHS

3. **accept** — terminate parsing and signal success

4. **error** — call an error recovery routine

Key insight: recognize handles with a DFA:

- DFA transitions shift states instead of symbols
- accepting states trigger reductions
Example tables

<table>
<thead>
<tr>
<th>state</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id + * $</td>
<td>&lt;expr&gt; &lt;term&gt; &lt;factor&gt;</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>s4</td>
<td>1 2 3</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>s5 r3</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>r5 s6 r5</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>r6 r6 r6</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>s4</td>
<td>7 2 3</td>
</tr>
<tr>
<td>6</td>
<td>s4</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>r2</td>
<td>8 3</td>
</tr>
<tr>
<td>8</td>
<td>r4 r4</td>
<td>-</td>
</tr>
</tbody>
</table>

The Grammar

1. `<goal>` ::= `<expr>`
2. `<expr>` ::= `<term>` + `<expr>`
3. | `<term>`
4. `<term>` ::= `<factor>` * `<term>`
5. | `<factor>`
6. `<factor>` ::= id

Note: This is a simple little right-recursive grammar. It is not the same grammar as in previous lectures.

Why study LR grammars?

LR(1) grammars are often used to construct parsers.

We call these parsers LR(1) parsers.

- used to be everyone’s favourite parser (but top-down is making a comeback with JavaCC)
- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars parsable by a deterministic, bottom-up parser
- efficient parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by predictive (i.e., LL) parsers

LL($k$): recognize use of a production $A \rightarrow \beta$ seeing first $k$ symbols of $\beta$

LR($k$): recognize occurrence of $\beta$ (the handle) having seen all of what is derived from $\beta$ plus $k$ symbols of lookahead

LR parsing

Three commonly used algorithms are used 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.
The table construction algorithms use sets of LR\((k)\) items or configurations to represent the 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, marking how much of the RHS of a production has already been seen

\(\beta\) is a lookahead string containing \(k\) symbols (terminals or $)

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.

### Example

The \(\bullet\) indicates how much of an item we have seen at a given state in the parse:

\([A \rightarrow \bullet XYZ]\) indicates that the parser is looking for a string that can be derived from \(XYZ\)

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

LR(0) items: (no lookahead)

\(A \rightarrow XYZ\) generates 4 LR(0) items:

1. \([A \rightarrow \bullet XYZ]\)
2. \([A \rightarrow X \bullet YZ]\)
3. \([A \rightarrow XY \bullet Z]\)
4. \([A \rightarrow XYZ\bullet]\)

### The characteristic finite state machine (CFSM)

The CFSM for a grammar is a DFA which recognizes viable prefixes of right-sentential forms:

A viable prefix is any prefix that does not extend beyond the handle.

It accepts when a handle has been discovered and needs to be reduced.

To construct the CFSM we need two functions:

- closure\(_0\)(\(I\)) to build its states
- goto\(_0\)(\(I, X\)) to determine its transitions

#### closure\(_0\)

Given an item \([A \rightarrow \alpha \bullet B\beta]\), 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 \([B \rightarrow \bullet \gamma]\) in the closure).

function closure\(_0\)(\(I\))

repeat

if \([A \rightarrow \alpha \bullet B\beta] \in I\)

add \([B \rightarrow \bullet \gamma]\) to \(I\)

until no more items can be added to \(I\)

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

Then, $\text{GOTO}(I,X)$ is the closure of the set of all items

\[ [A \rightarrow \alpha X \cdot \beta] \text{ such that } [A \rightarrow \alpha \cdot X \beta] \in I \]

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

$\text{GOTO}(I,X)$ represents state after recognizing $X$ in state $I$.

function goto0($I,X$

let $J$ be the set of items $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha \cdot X \beta] \in I$
return closure0($J$

---

**Building the LR(0) item sets**

We start the construction with the item $[S' \rightarrow \cdot S\$

where

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

$S$ is the start symbol of $G$

$\$ represents EOF

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

function items($G'$

$s_0 = \text{closure0}([S' \rightarrow \cdot S\$])$

$S = \{s_0\}$
repeat
for each set of items $s \in S$
for each grammar symbol $X$
if goto0($s,X$) $\neq \emptyset$ and goto0($s,X$) $\notin S$
add goto0($s,X$) to $S$
until no more item sets can be added to $S$
return $S$

---

**LR(0) example**

|   | $S \rightarrow E\$
|---|---|
| 1 | $E \rightarrow E + T$
| 2 | $T \rightarrow \text{id}$
| 3 | $E \rightarrow \cdot (E)$
| 4 | $T \rightarrow \cdot (E)$
| 5 | $\$ |

$I_0 : [S \rightarrow \cdot E\$  
$I_1 : [E \rightarrow E + T \cdot$
$I_2 : [S \rightarrow E\$
$I_3 : [E \rightarrow E + T \cdot$
$I_4 : [E \rightarrow E + T \cdot$
$I_5 : [T \rightarrow \text{id} \cdot$
$I_6 : [T \rightarrow \cdot (E)$
$I_7 : [T \rightarrow \cdot (E)$
$I_8 : [T \rightarrow (E)$
$I_9 : [T \rightarrow \cdot (E)$

The corresponding CFSM:

---

**Constructing the LR(0) parsing table**

1. construct the collection of sets of LR(0) items for $S'$

2. state $i$ of the CFSM is constructed from $I_i$

   (a) $[A \rightarrow \alpha \cdot a \beta] \in I_i$ and goto0($I_i,a \rightarrow \rightarrow j$

   (b) $[A \rightarrow \alpha \cdot] \in I_i, A \neq S'$

   (c) $[S' \rightarrow S\$] \in I_i$

3. goto0($I_i,A \rightarrow \rightarrow j$

4. set undefined entries in ACTION and GOTO to "error"

5. initial state of parser $s_0$ is closure0([$S' \rightarrow \cdot S\$])
Conflicts in the ACTION table

If the LR(0) parsing table contains any multiply-defined ACTION entries then $G$ is not LR(0).

Two conflicts arise:

**shift-reduce**: both shift and reduce possible in same item set

**reduce-reduce**: more than one distinct reduce action possible in same item set

Conflicts can be resolved through lookahead in ACTION. Consider:

- $A \rightarrow \epsilon | aA$
  - $\Rightarrow$ shift-reduce conflict

- $a:=b+c*d$
  - requires lookahead to avoid shift-reduce conflict after shifting $c$ (need to see $*$ to give precedence over $+$)

---

A simple approach to adding lookahead: SLR(1)

Add lookaheads after building LR(0) item sets

Constructing the SLR(1) parsing table:

1. construct the collection of sets of LR(0) items for $G'$
2. state $i$ of the CFSM is constructed from $I_i$
   - (a) $[A \rightarrow \alpha \bullet a\beta] \in I_i$ and goto0($I_i, a$) = $I_j$
     - $\Rightarrow$ ACTION[$i, a$] = "shift $j$", $\forall a \neq $
   - (b) $[A \rightarrow \alpha \bullet] \in I_i, A \neq S'$
     - $\Rightarrow$ ACTION[$i, a$] = "reduce $A \rightarrow \alpha"$
     - $\forall a \in$ FOLLOW($A$)
   - (c) $[S' \rightarrow S \bullet S] \in I_i$
     - $\Rightarrow$ ACTION[$i, S$] = "accept"
3. goto0($I_i, A$) = $I_j$
   - $\Rightarrow$ GOTO[$i, A$] = $j$
4. set undefined entries in ACTION and GOTO to "error"
5. initial state of parser $s_0$ is closure0([$S' \rightarrow \bullet S$])

---

From previous example

- $S \rightarrow E\$
- $E \rightarrow E + T$
- $| T$
- $T \rightarrow \text{id}$
- $$(E)$

FOLLOW($E$) = FOLLOW($T$) = \{$,+,$\}

---

<table>
<thead>
<tr>
<th>state</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>id ( ) + $</td>
<td>S E T</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>s5 s6 – – –</td>
<td>– 1 9</td>
</tr>
<tr>
<td>1</td>
<td>– – – s3 s2 – – –</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>acc acc acc acc acc</td>
<td>– – –</td>
</tr>
<tr>
<td>3</td>
<td>s5 s6 – – –</td>
<td>– 4</td>
</tr>
<tr>
<td>4</td>
<td>r2 r2 r2 r2 r2</td>
<td>– – –</td>
</tr>
<tr>
<td>5</td>
<td>r4 r4 r4 r4 r4</td>
<td>– – –</td>
</tr>
<tr>
<td>6</td>
<td>s5 s6 – – –</td>
<td>– 7 9</td>
</tr>
<tr>
<td>7</td>
<td>– – s8 s3 –</td>
<td>– – –</td>
</tr>
<tr>
<td>8</td>
<td>r5 r5 r5 r5 r5</td>
<td>– – –</td>
</tr>
<tr>
<td>9</td>
<td>r3 r3 r3 r3 r3</td>
<td>– – –</td>
</tr>
</tbody>
</table>
Example: A grammar that is not LR(0)

Consider:
\[
\begin{align*}
S & \rightarrow \ E \ S \\
E & \rightarrow \ T + E \\
E & \rightarrow \ T \\
T & \rightarrow \ T \ast F \\
T & \rightarrow \ F \\
F & \rightarrow \ id \\
F & \rightarrow \ (E)
\end{align*}
\]

Its LR(0) item sets:
\[
\begin{align*}
I_0 : & \quad S \rightarrow \ E \ S \\
I_1 : & \quad S \rightarrow \ E \ S \\
I_2 : & \quad S \rightarrow \ E \ S \\
I_3 : & \quad S \rightarrow \ E \ S \\
I_4 : & \quad S \rightarrow \ E \ S
\end{align*}
\]

Consider \( I_2 \):
\[
\in \text{FOLLOW}(R) \quad (S \Rightarrow L = R \Rightarrow \ast R = R)
\]
Given an item \([A \rightarrow \alpha \bullet B\beta, a]\), 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 other
item \([B \rightarrow \gamma, b]\) in the closure).

function closure1(I)
repeat
  if \([A \rightarrow \alpha \bullet B\beta, a]\) \(\in I\)
    add \([B \rightarrow \gamma, b]\) to \(I\), where \(b \in \text{FIRST}(\beta a)\)
until no more items can be added to \(I\)
return \(I\)

Compiler Construction 1 33

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
\([A \rightarrow \alpha X \bullet \beta, a]\) such that \([A \rightarrow \alpha \bullet X\beta, a] \in I\)
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 \(I\).

function goto1(I, X)
  let \(J\) be the set of items
  \([A \rightarrow \alpha X \bullet \beta, a]\) such that \([A \rightarrow \alpha \bullet X\beta, a] \in I\)
  return closure1\((J)\)

Compiler Construction 1 34

Build lookahead into the DFA to begin with

1. construct the collection of sets of LR(1) items for
   \(G'\)
2. state \(i\) of the LR(1) machine is constructed from
   \(I_i\)
   \(\begin{align*}
   (a) & \ [A \rightarrow \alpha \bullet a\beta, b] \in I_i \text{ and goto1}(I_i, a) = I_j \\
       & \Rightarrow \text{ACTION}[i, a] = \text{"shift"}
   \\
   (b) & \ [A \rightarrow \alpha \bullet \gamma, a] \in I_i, A \neq S' \\
       & \Rightarrow \text{ACTION}[i, a] = \text{"reduce A \rightarrow \alpha"}
   \\
   (c) & \ [S' \rightarrow S \bullet, \$] \in I_i \\
       & \Rightarrow \text{ACTION}[i, \$] = \text{"accept"}
   \end{align*}\)
3. goto1\((I_i, A) = I_j \Rightarrow \text{GOTO}[i, A] = j\)
4. set undefined entries in ACTION and GOTO to
   \text{"error"}
5. initial state of parser \(s_0\) is closure1\(([S' \rightarrow \bullet S, \$])\)
Back to previous example ($\notin$ SLR(1))

$$S \rightarrow L = R$$
$$\mid R$$
$$L \rightarrow \ast R$$
$$\mid \ast id$$
$$R \rightarrow L$$

Its LR(1) item sets:

1. $I_0: S' \rightarrow \ast S$, $\mid$
   $S \rightarrow \ast L = R, \mid$
   $S \rightarrow \ast R, \mid$
   $L \rightarrow \ast \ast R, =$
   $L \rightarrow \ast \ast id, =$
   $R \rightarrow \ast L, =$
   $L \rightarrow \ast R, =$
   $L \rightarrow \ast id, =$

2. $I_1: S' \rightarrow \ast S$, $\mid$
   $I_{10}: R \rightarrow \ast L$, $\mid$

3. $I_2: S \rightarrow \ast L = R, \mid$

4. $I_3: S \rightarrow \ast R, \mid$

5. $I_4: L \rightarrow \ast \ast R, =$

6. $I_5: L \rightarrow \ast \ast id, =$

$\mid$ $I_6: L \rightarrow \ast \ast L, =$

$I_7: L \rightarrow \ast \ast R$, $\mid$

$I_8: L \rightarrow \ast \ast id, =$

$I_9: S \rightarrow \ast L = \ast R$, $\mid$

$I_{10}: R \rightarrow \ast L$, $\mid$

$I_{11}: L \rightarrow \ast \ast R$, $\mid$

$I_{12}: L \rightarrow \ast \ast id, =$

$I_{13}: L \rightarrow \ast \ast L$, $\mid$

$I_2$ no longer has shift-reduce conflict:

reduce on $\$, shift on $=$.

Compiler Construction 1 37

Consider:

$$S' \rightarrow S$$
$$1$$
$$S \rightarrow CC$$
$$2$$
$$C \rightarrow cC$$
$$3$$

<table>
<thead>
<tr>
<th>$S' \rightarrow S$</th>
<th>$S \rightarrow CC$</th>
<th>$C \rightarrow cC$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_0: S' \rightarrow \ast S, \mid$</td>
<td>$S \rightarrow \ast CC, \mid$</td>
<td>$C \rightarrow \ast cC, \ast cd$</td>
</tr>
<tr>
<td>$I_1: S' \rightarrow \ast S, \mid$</td>
<td>$I_{10}: R \rightarrow \ast \ast L, =$</td>
<td>$I_{11}: L \rightarrow \ast \ast R, =$</td>
</tr>
<tr>
<td>$I_{12}: L \rightarrow \ast \ast id, =$</td>
<td>$I_{13}: L \rightarrow \ast \ast L, =$</td>
<td></td>
</tr>
</tbody>
</table>

Another example

LALR(1) parsing

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

- $\{ [A \rightarrow \alpha \ast \beta, \alpha], [A' \rightarrow \alpha' \ast \beta', b], \}$
- $\{ [A \rightarrow \alpha \ast \beta, c], [A' \rightarrow \alpha' \ast \beta', d], \}$

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.
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.

The revised (and renumbered) algorithm

1. construct the collection of sets of LR(1) items for $G'$
2. for each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union. (Update the goto function incrementally)
3. state $i$ of the LALR(1) machine is constructed from

   - $[A \rightarrow \alpha \cdot a \beta, b] \in I_i$ and $\text{goto}1(I_i, a) = I_j \Rightarrow \text{ACTION}[i, a] = "\text{shift} j"
   - $[A \rightarrow \alpha \cdot a] \in I_i, A \neq S'$
   - $\Rightarrow \text{ACTION}[i, a] = "\text{reduce} A \rightarrow \alpha"

4. $\text{goto}1(I_i, A) = I_j \Rightarrow \text{GOTO}[i, A] = j$
5. set undefined entries in ACTION and GOTO to "error"
6. initial state of parser $s_0$ is closure1($[S' \rightarrow \cdot S]$)

---

**Example**

Reconsider:

<table>
<thead>
<tr>
<th></th>
<th>$S'$</th>
<th>$S$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$S'$</td>
<td>$S$</td>
<td>$C$</td>
</tr>
<tr>
<td>1</td>
<td>$S$</td>
<td>$CC$</td>
<td>$cC$</td>
</tr>
<tr>
<td>2</td>
<td>$C$</td>
<td>$cC$</td>
<td>$d$</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>$cC$</td>
<td>$d$</td>
</tr>
<tr>
<td>4</td>
<td>$C$</td>
<td>$cC$</td>
<td>$d$</td>
</tr>
</tbody>
</table>

**LR(1) item sets:**

- $I_0: S' \rightarrow \cdot S$, $\$  
- $S \rightarrow \cdot CC$, $\$  
- $C \rightarrow \cdot cC$, $\$  
- $C \rightarrow \cdot d$, $\$  
- $I_1: S' \rightarrow \cdot S$, $\$  
- $C \rightarrow \cdot cC$, $\$  
- $C \rightarrow \cdot d$, $\$  
- $I_2: S \rightarrow \cdot C$, $\$  
- $C \rightarrow \cdot cC$, $\$  
- $C \rightarrow \cdot d$, $\$  
- $I_3: C \rightarrow \cdot C$, $\$  
- $C \rightarrow \cdot cC$, $\$  
- $C \rightarrow \cdot d$, $\$  
- $I_4: C \rightarrow \cdot cC$, $\$  
- $C \rightarrow \cdot d$, $\$  
- $I_5: S \rightarrow \cdot CC$, $\$  
- $C \rightarrow \cdot cC$, $\$  
- $C \rightarrow \cdot d$, $\$  
- $I_6: C \rightarrow \cdot cC$, $\$  
- $C \rightarrow \cdot d$, $\$  
- $I_7: C \rightarrow \cdot d$, $\$  
- $I_8: C \rightarrow \cdot cC$, $\$  
- $I_9: C \rightarrow \cdot d$, $\$  

**Merged states:**

<table>
<thead>
<tr>
<th>state</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>$d$</td>
<td>$$</td>
</tr>
<tr>
<td>0</td>
<td>s36</td>
<td>s47</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>s36</td>
<td>s47</td>
</tr>
<tr>
<td>36</td>
<td>s36</td>
<td>s47</td>
</tr>
<tr>
<td>47</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>r1</td>
</tr>
<tr>
<td>89</td>
<td>r2</td>
<td>r2</td>
</tr>
</tbody>
</table>

**The role of precedence**

Precedence and associativity can be used to resolve shift/reduce conflicts in ambiguous grammars.

- lookahead with higher precedence $\Rightarrow$ shift
- same precedence, left associative $\Rightarrow$ reduce

**Advantages:**

- more concise, albeit ambiguous, grammars
- shallower parse trees $\Rightarrow$ fewer reductions

**Classic application: expression grammars**

With precedence and associativity, we can use:

- $E \rightarrow E \ast E$
- $E \rightarrow E + E$
- $E \rightarrow E - E$
- $E \rightarrow (E)$
- $E \rightarrow -E$
- $E \rightarrow \text{id}$
- $E \rightarrow \text{num}$
Error recovery in shift-reduce parsers

The problem

- encounter an invalid token
- bad pieces of tree hanging from stack
- incorrect entries in symbol table

We want to parse the rest of the file

Restarting the parser

- find a restartable state on the stack
- move to a consistent place in the input
- print an informative message (including line number)

Compiler Construction 1 45

Left versus right recursion

Right Recursion:

- needed for termination in predictive parsers
- requires more stack space
- right associative operators

Left Recursion:

- works fine in bottom-up parsers
- limits required stack space
- left associative operators

Rule of thumb:

- right recursion for top-down parsers
- left recursion for bottom-up parsers

Compiler Construction 1 46

Parsing review

Recursive descent

A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).

LL(k)

An LL(k) parser must be able to recognize the use of a production after seeing only the first k symbols of its right hand side.

LR(k)

An LR(k) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with k symbols of lookahead.

Compiler Construction 1 47