Compiler Design

Bottom-up Parser (II)

Hwansoo Han
LR(k) Items

- A state of parser == a set of LR(k) items
- An LR(k) item is a pair \([P, \delta]\), where
  - \(P\) is a production \(A \rightarrow \beta\) with a \(\cdot\) at some position in the rhs
  - \(\delta\) is a lookahead string of length \(\leq k\) (words/tokens or EOF)
LR(k) Items

- **LR(1) items**
  - The • in an item indicates the position of the top of the stack
  
  - \([A \rightarrow \cdot \beta \gamma, a]\) means that the input seen so far is consistent with the use of \(A \rightarrow \beta \gamma\) immediately after the symbol on top of the stack. (*possibility*)
  
  - \([A \rightarrow \beta \cdot \gamma, a]\) means that the input seen so far is consistent with the use of \(A \rightarrow \beta \gamma\) at this point, *and* that the parser has already recognized \(\beta\). (*partially complete*)
  
  - \([A \rightarrow \beta \gamma \cdot, a]\) means that the parser has seen \(\beta \gamma\), *and* that a lookahead symbol of \(a\) is consistent with reducing to \(A\). (*complete*)
Computing goto()

- **goto(s,x)** computes the state that the parser would reach if it recognized an *x* while in state *s*
  - **goto**({ [A→β<Xδ,a] }, X ) produces [A→βX•δ,a]  (easy part)
  - Should also includes closure([A→βX•δ,a] )  (fill out the state)

- The algorithm

```
/* goto( s, X ) */
moved ← Ø
for each item [A→β•Xδ,a] ∈ s
    moved ← moved ∪ [A→βX•δ,a]
return closure(moved)
```

- Not a fixed-point method!
- Straightforward computation
- Uses closure( )

**goto() moves forward**
Computing closure()

- **Closure(s)** adds all the items implied by items already in s
  - Any item \([A \rightarrow \beta \cdot B \delta, a]\) implies \([B \rightarrow \cdot \tau, x]\) for each production with \(B\) on the lhs, and each \(x \in \text{FIRST}(\delta a)\)
  - Since \(\beta B \delta\) is valid, any way to derive \(\beta B \delta\) is valid, too

- **The algorithm**

```plaintext
closure( s )
while ( s is still changing )
  for each item \([A \rightarrow \beta \cdot C \delta, a]\) \in s
    for each production \(C \rightarrow \tau \in P\)
      for each \(b \in \text{FIRST}(\delta a)\) // \(\delta\) might be \(\varepsilon\)
        \(s \leftarrow s \cup [C \rightarrow \cdot \tau, b]\)
```

- Classic fixed-point method
- Halts because \(s \subseteq \text{ITEMS}\)
- Worklist version is faster

Closure “fills out” a state
High-level overview \((\text{Algorithm})\)

1. **Build the canonical collection of sets of LR(1) Items, \(I\)**
   a. Begin in an appropriate state, \(cc_0\)
      - \([S \rightarrow \cdot S, EOF]\), along with any equivalent items
      - Derive equivalent items as \(\text{closure}( cc_0 )\)
   b. Repeatedly compute, for each \(cc_k\) and each \(X\), \(\text{goto}(cc_k, X)\)
      - If the set is not already in the collection, add it
      - Record all the transitions created by \(\text{goto}( )\)
   This eventually reaches a fixed point

2. **Fill in the table from the collection of sets of LR(1) items**

   The canonical collection completely encodes the transition diagram for the handle-finding DFA
Canonical Collection

❖ Building CC: all possible states
- Start from \( cc_0 = \text{closure}( [S \rightarrow S, \text{EOF}]) \)
- Repeatedly construct new states, until all are found

❖ The algorithm

\[
cc_0 \leftarrow \text{closure}( [S \rightarrow S, \text{EOF}])
\]
\[
CC \leftarrow \{ cc_0 \}
\]
\[
k \leftarrow 1
\]

\textbf{while (CC is still changing)}

\textbf{for each } \( cc_j \in CC \) \textbf{ and for each } \( x \in (T \cup NT) \)

\[
cc_k \leftarrow \text{goto}(cc_j, x)
\]

record \( cc_j \rightarrow cc_k \) on \( x \)

\textbf{if } \( cc_k \notin CC \) \textbf{ then}

\[
CC \leftarrow CC \cup cc_k \quad // \text{new state in DFA}
\]
\[
k \leftarrow k + 1
\]

❖ Fixed-point computation
❖ Loop adds to \( CC \)
❖ \( CC \subseteq 2^{\text{ITEMS}} \), so \( CC \) is finite

\textit{Worklist version is faster}
Example

Simplified, right recursive expression grammar

\[
\begin{align*}
\text{Goal} & \rightarrow \text{Expr} \\
\text{Expr} & \rightarrow \text{Term} \; - \; \text{Expr} \\
\text{Expr} & \rightarrow \text{Term} \\
\text{Term} & \rightarrow \text{Factor} \; * \; \text{Term} \\
\text{Term} & \rightarrow \text{Factor} \\
\text{Factor} & \rightarrow \text{ident} \\
\end{align*}
\]

\[
\begin{array}{|c|c|}
\hline
\text{Symbol} & \text{FIRST} \\
\hline
\text{Goal} & \{ \text{ident} \} \\
\text{Expr} & \{ \text{ident} \} \\
\text{Term} & \{ \text{ident} \} \\
\text{Factor} & \{ \text{ident} \} \\
\; - & \{ - \} \\
\; * & \{ * \} \\
\text{ident} & \{ \text{ident} \} \\
\hline
\end{array}
\]
Example (building the collection)

Initialization Step

\[ cc_0 \leftarrow \text{closure}( \{ [\text{Goal} \rightarrow \cdot \text{Expr}, \text{EOF}] \} ) \]

\{ [\text{Goal} \rightarrow \cdot \text{Expr}, \text{EOF}],
    [\text{Expr} \rightarrow \cdot \text{Term} \cdot \text{Expr}, \text{EOF}],
    [\text{Expr} \rightarrow \cdot \text{Term}, \text{EOF}],
    [\text{Term} \rightarrow \cdot \text{Factor} \cdot \text{Term}, \cdot],
    [\text{Term} \rightarrow \cdot \text{Factor}, \cdot],
    [\text{Factor} \rightarrow \cdot \text{ident}, \cdot, \cdot],
    [\text{Factor} \rightarrow \cdot \text{ident}, \cdot, *] \}

Add \( cc_0 \) to a set of states, \( CC \leftarrow \{ cc_0 \} \)
Example (building the collection)

Iteration 1

\[ cc_1 \leftarrow \text{goto}(cc_0, \text{Expr}) \]
\[ cc_2 \leftarrow \text{goto}(cc_0, \text{Term}) \]
\[ cc_3 \leftarrow \text{goto}(cc_0, \text{Factor}) \]
\[ cc_4 \leftarrow \text{goto}(cc_0, \text{ident}) \]

Iteration 2

\[ cc_5 \leftarrow \text{goto}(cc_2, \text{-}) \]
\[ cc_6 \leftarrow \text{goto}(cc_3, \text{*}) \]

Iteration 3

\[ cc_7 \leftarrow \text{goto}(cc_5, \text{Expr}), \quad \# \text{ Term, Factor, ident} \Rightarrow \text{existing states} \]
\[ cc_8 \leftarrow \text{goto}(cc_6, \text{Term}) \quad \# \text{ Factor, ident} \Rightarrow \text{existing states} \]
Example

(Summary)

CC_0 : \{ [Goal \rightarrow \cdot Expr, EOF], [Expr \rightarrow \cdot Term - Expr, EOF], [Expr \rightarrow \cdot Term, EOF],
       [Term \rightarrow \cdot Factor * Term, EOF], [Term \rightarrow \cdot Factor * Term, -],
       [Term \rightarrow \cdot Factor, EOF], [Term \rightarrow \cdot Factor, -],
       [Factor \rightarrow \cdot ident, EOF], [Factor \rightarrow \cdot ident, -], [Factor \rightarrow \cdot ident, *] \}

CC_1 : \{ [Goal \rightarrow Expr \cdot, EOF] \}

CC_2 : \{ [Expr \rightarrow Term \cdot - Expr, EOF], [Expr \rightarrow Term \cdot, EOF] \}

CC_3 : \{ [Term \rightarrow Factor \cdot * Term, EOF], [Term \rightarrow Factor \cdot * Term, -],
       [Term \rightarrow Factor \cdot, EOF], [Term \rightarrow Factor \cdot, -] \}

CC_4 : \{ [Factor \rightarrow ident \cdot, EOF], [Factor \rightarrow ident \cdot, -], [Factor \rightarrow ident \cdot, *] \}

CC_5 : \{ [Expr \rightarrow Term - \cdot Expr, EOF],
       [Expr \rightarrow \cdot Term - Expr, EOF], [Expr \rightarrow \cdot Term, EOF],
       [Term \rightarrow \cdot Factor * Term, EOF], [Term \rightarrow \cdot Factor * Term, -],
       [Term \rightarrow \cdot Factor, EOF], [Term \rightarrow \cdot Factor, -],
       [Factor \rightarrow \cdot ident, EOF], [Factor \rightarrow \cdot ident, -], [Factor \rightarrow \cdot ident, *] \}
Example

(Summary)

\[ \text{cc}_6 : \{ [\text{Term} \rightarrow \text{Factor} \cdot \text{Term} \cdot \text{EOF}], [\text{Term} \rightarrow \text{Factor} \cdot \text{Term} \cdot -], [\text{Term} \rightarrow \cdot \text{Factor} \cdot \text{Term} \cdot \text{EOF}], [\text{Term} \rightarrow \cdot \text{Factor} \cdot \text{Term} \cdot -], [\text{Term} \rightarrow \cdot \text{Factor} \cdot \text{EOF}], [\text{Term} \rightarrow \cdot \text{Factor} \cdot -], [\text{Factor} \rightarrow \cdot \text{id} \cdot \text{EOF}], [\text{Factor} \rightarrow \cdot \text{id} \cdot -], [\text{Factor} \rightarrow \cdot \text{id} \cdot *] \} \]

\[ \text{cc}_7 : \{ [\text{Expr} \rightarrow \text{Term} - \text{Expr} \cdot , \text{EOF}] \} \]

\[ \text{cc}_8 : \{ [\text{Term} \rightarrow \text{Factor} \cdot \text{Term} \cdot , \text{EOF}], [\text{Term} \rightarrow \text{Factor} \cdot \text{Term} \cdot -] \} \]
The *goto()* Relationship *(from the construction)*

<table>
<thead>
<tr>
<th>State</th>
<th>Expr</th>
<th>Term</th>
<th>Factor</th>
<th>-</th>
<th>*</th>
<th>Ident</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Filling in the ACTION and GOTO Tables

- **The algorithm**

  for each set $cc_x \in CC$
  for each item $i \in cc_x$
    if $i$ is $[A \rightarrow \beta \cdot a, b]$ and $goto(cc_x, a) = cc_k$, $a \in T$
      then $ACTION[x, a] \leftarrow \text{"shift } k\text{"}$
    else if $i$ is $[S' \rightarrow S \cdot \text{EOF}]$
      then $ACTION[x, \text{EOF}] \leftarrow \text{"accept"}$
    else if $i$ is $[A \rightarrow \beta \cdot a]$
      then $ACTION[x, a] \leftarrow \text{"reduce } A \rightarrow \beta\text{"}$
  for each $nt \in NT$
    if $goto(cc_x, nt) = cc_k$
      then $GOTO[x, nt] \leftarrow k$

- **Ignores many items where the $\cdot$ precedes non-terminal**
  - $\text{closure( )}$ instantiates items where $\cdot$ precedes FIRST($X$)
    $[A \rightarrow \beta \cdot X \gamma, a]$ forces to have $[X \rightarrow \cdot b \delta, c]$, 
    where $b \in \text{FIRST}(X)$, $c \in \text{FIRST}(\gamma a)$, $X \Rightarrow^* b \delta$
Example
(Filling in the tables)

The algorithm produces the following table

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ident</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>s 4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>acc</td>
</tr>
<tr>
<td>s 5</td>
<td>r 3</td>
</tr>
<tr>
<td>r 5</td>
<td>s 6</td>
</tr>
<tr>
<td>r 6</td>
<td>r 6</td>
</tr>
<tr>
<td>s 4</td>
<td></td>
</tr>
<tr>
<td>s 4</td>
<td></td>
</tr>
<tr>
<td>s 4</td>
<td></td>
</tr>
<tr>
<td>r 4</td>
<td>r 4</td>
</tr>
</tbody>
</table>

Plugs into the skeleton LR(1) parser
What can go wrong?

- What if set $s$ contains $[A \rightarrow \beta \cdot a, b]$ and $[B \rightarrow \beta \cdot, a]$?
  - First item generates “shift”, second generates “reduce”
  - Both define $\text{ACTION}[s,a]$ — cannot do both actions
  - This is a fundamental ambiguity, called a $\text{shift/reduce error}$
  - Modify the grammar to eliminate it ($\text{if-then-else}$)
  - Shifting will often resolve it correctly

- What if set $s$ contains $[A \rightarrow \gamma \cdot, a]$ and $[B \rightarrow \gamma \cdot, a]$?
  - Each generates “reduce”, but with a different production
  - Both define $\text{ACTION}[s,a]$ — cannot do both reductions
  - This fundamental ambiguity is called a $\text{reduce/reduce error}$
  - Modify the grammar to eliminate it ($\text{PL/I's overloading of (})$

- In either case, the grammar is not LR(1)
Shrinking the Tables

- **Combine terminals** - `number & identifier, + & -, * & /`
  - Directly removes a column, may remove a row
  - For expression grammar, 198 (vs. 384) table entries

- **Combine rows or columns**
  - Implement identical rows once & remap states
  - Requires extra indirection on each lookup of ACTION & GOTO
  - Use separate mapping for ACTION & for GOTO

- **Use another construction algorithm**
  - Both LALR and SLR produce smaller tables with LR(0) items
  - Implementations are readily available
**SLR vs. LR(1) vs. LALR**

- **SLR parsing**
  - States are constructed from LR(0) items
  - State transitions based on symbols(X) right after •
    - \( A \rightarrow \beta \cdot X \gamma \)

- **LR(1) parsing**
  - States are constructed from LR(1) items
  - State transitions based on symbols(X) right after •
    - \( A \rightarrow \beta \cdot X \gamma, s \)

- **LALR parsing**
  - States are constructed with LR(0) items and refine states if different actions are needed depending on look-ahead symbol
  - Or states are constructed with LR(1) items and merge states if cores are the same and the same action is needed for the merged look-ahead symbols
LR(k) vs. LL(k)

Finding Reductions

- **LR(k)** ⇒ Each reduction in the parse is detectable with
  1. the completed left context,
  2. the reducible phrase, itself, and
  3. the $k$ terminal symbols to its right

- **LL(k)** ⇒ Parser must select the reduction based on
  1. The completed left context
  2. The next $k$ terminals

Thus, LR(k) examines more context

"... in practice, programming languages do not actually seem to fall in the gap between LL(1) languages and deterministic languages"

**Left Recursion vs. Right Recursion**

- **Right recursion**
  - Required for termination in top-down parsers
  - Uses (on average) more stack space
  - Produces right-associative operators

- **Left recursion**
  - Works fine in bottom-up parsers
  - Limits required stack space
  - Produces left-associative operators

- **Rule of thumb**
  - Left recursion for bottom-up parsers
  - Right recursion for top-down parsers

![Diagram of right recursion](image)

![Diagram of left recursion](image)

```
*   *
w   *     *
   x   *
   y   z

w * (x * (y * z))
```

```
*   *
*   *
*   *
w   x   y   z

((w * x) * y) * z
```
Hierarchy of Context-Free Languages

Context-Sensitive languages

Context-free languages

RG

LR(k)

LL(k)