Parsing V
Introduction to LR(1) Parsers
**LR(1) Parsers**

- LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 word) for handle recognition.
- LR(1) parsers recognize languages that have an LR(1) grammar.

**Informal definition:**

A grammar is LR(1) if, given a rightmost derivation

\[ S \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow \text{sentence} \]

We can

1. *isolate the handle of each right-sentential form* \( \gamma_i \), and
2. *determine the production by which to reduce*,

by scanning \( \gamma_i \) from left-to-right, going at most 1 symbol beyond the right end of the handle of \( \gamma_i \).
LR(1) Parsers

A table-driven LR(1) parser looks like

Tables can be built by hand

It is a perfect task to automate

(homework # 2)

from Cooper & Torczon
LR(1) Parsers

(\textit{the skeleton parser})

\begin{itemize}
    \item \textbf{push INVALID}
    \item \textbf{push }s_0$
    \item \textbf{token }\leftarrow \text{next}\_\text{token()}
    \item \textbf{repeat forever}
        \begin{itemize}
            \item \textbf{s }\leftarrow \text{top of stack}
            \item \textbf{if }\text{ACTION}[s,\text{token}] = \text{"reduce }N\rightarrow\beta\text{"}
                \begin{itemize}
                    \item \textbf{pop }2^*|\beta|\text{ symbols}
                    \item \textbf{s }\leftarrow \text{top of stack}
                    \item \textbf{push }N
                    \item \textbf{push GOTO}[s,N]
                \end{itemize}
            \item \textbf{else if }\text{ACTION}[s,\text{token}] = \text{"shift }s_i\text{"}
                \begin{itemize}
                    \item \textbf{push }\text{token }\text{; push }s_i
                    \item \textbf{token }\leftarrow \text{next}\_\text{token()}
                \end{itemize}
            \item \textbf{else if }\text{ACTION}[s,\text{token}] = \text{"accept"}
                \begin{itemize}
                    \item \textbf{and }\text{token }= \text{EOF}
                \end{itemize}
        \end{itemize}
    \item \textbf{then break;}
    \item \textbf{else report a syntax error}
\end{itemize}

\textbf{report success}

\textit{The skeleton parser}

\begin{itemize}
    \item \textbf{uses }\text{ACTION }\& \text{GOTO}
    \item \textbf{does }|\text{words}| \text{shifts}
    \item \textbf{does }|\text{derivation}| \text{reductions}
    \item \textbf{does 1 accept}
    \item \textbf{detects errors by failure of 3 other cases}
\end{itemize}

\textbf{from Cooper \& Torczon}
LR(1) Parsers (parse tables)

To make a parser for $L(G)$, need a set of tables

The grammar

<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>$\rightarrow$</th>
<th>SheepNoise</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SheepNoise</td>
<td>$\rightarrow$</td>
<td>SheepNoise $baa$</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>$baa$</td>
<td></td>
</tr>
</tbody>
</table>

The tables

<table>
<thead>
<tr>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>EOF</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>1</td>
<td>accept</td>
</tr>
<tr>
<td>2</td>
<td>reduce 3</td>
</tr>
<tr>
<td>3</td>
<td>reduce 2</td>
</tr>
</tbody>
</table>

from Cooper & Torczon
Example Parses

The string “baa”

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>b aa</td>
<td>$s_0$</td>
<td>shift 2</td>
</tr>
<tr>
<td>EOF</td>
<td>$s_0$ b aa $s_2$</td>
<td>reduce 3</td>
</tr>
<tr>
<td>EOF</td>
<td>$s_0$ SN $s_1$</td>
<td>accept</td>
</tr>
</tbody>
</table>

We cannot have a syntax error with SN, because it only has 1 terminal symbol!

“baa woof” is a lexical problem, not a syntax error!

The string “baa”

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>b aa</td>
<td>$s_0$</td>
<td>shift 2</td>
</tr>
<tr>
<td>b aa</td>
<td>$s_0$ b aa $s_2$</td>
<td>reduce 3</td>
</tr>
<tr>
<td>EOF</td>
<td>$s_0$ SN $s_1$</td>
<td>shift 3</td>
</tr>
<tr>
<td>EOF</td>
<td>$s_0$ SN $s_1$ b aa $s_3$</td>
<td>reduce 2</td>
</tr>
<tr>
<td>EOF</td>
<td>$s_0$ SN $s_1$</td>
<td>accept</td>
</tr>
</tbody>
</table>
LR(1) Parsers

How does this LR(1) stuff work?

• Unambiguous grammar ⇒ unique rightmost derivation
• Keep upper fringe on a stack
  > All active handles include TOS
  > Shift inputs until TOS is right end of a handle
• Language of handles is regular (finite)
  > Build a handle-recognizing DFA
  > ACTION & GOTO tables encode the DFA
• To match subterms, recurse & leave DFA’s state on stack
• Final state in DFA ⇒ a reduce action
  > New state is GOTO[ lhs, state at TOS]
  > For SN, this takes the DFA to S₁

Control DFA for SN

from Cooper & Torczon
Building LR(1) Parsers

How do we generate the ACTION and GOTO tables?

• Use the grammar to build a model of the DFA
• Use the model to build ACTION & GOTO tables
• If construction succeeds, the grammar is LR(1)

The Big Picture

• Model the state of the parser
• Use two functions \( \text{goto}(s,N) \) and \( \text{closure}(s) \)
  > \( \text{goto}() \) is analogous to \( \text{move()} \) in the subset construction
  > \( \text{closure}() \) adds information to round out a state
• Build up the states and transition functions of the DFA
• Use this information to fill in the ACTION and GOTO tables
**LR(\(k\)) items**

An \(LR(k)\) item is a pair \([A,B]\), where

- \(A\) is a production \(\alpha \rightarrow \beta \gamma \delta\) with a \(\bullet\) at some position in the \(rhs\)
- \(B\) is a lookahead string of length \(k\) \((\text{words or EOF})\)

The \(\bullet\) in an item indicates the position of the top of the stack

\([\alpha \rightarrow \bullet \beta \gamma \delta, a]\) means that the input seen so far is consistent with the use of \(\alpha \rightarrow \beta \gamma \delta\) immediately after the symbol on top of the stack

\([\alpha \rightarrow \beta \gamma \bullet \delta, a]\) means that the input seen so far is consistent with the use of \(\alpha \rightarrow \beta \gamma \delta\) at this point in the parse, *and* that the parser has already recognized \(\beta \gamma\).

\([\alpha \rightarrow \beta \gamma \delta \bullet, a]\) means that the parser has seen \(\beta \gamma \delta\), *and* that a lookahead symbol of \(a\) is consistent with reducing to \(\alpha\).

The table construction algorithm uses items to represent valid configurations of an \(LR(1)\) parser
**LR(1) Items**

The production $\alpha \rightarrow \cdot \beta \gamma \delta$, with lookahead $a$, generates 4 items

$$[\alpha \rightarrow \cdot \beta \gamma \delta, a], [\alpha \rightarrow \beta \cdot \gamma \delta, a], [\alpha \rightarrow \beta \gamma \cdot \delta, a], \& [\alpha \rightarrow \beta \gamma \delta \cdot, a]$$

The set of LR(1) items for a grammar is finite

What’s the point of all these lookahead symbols?

- Carry them along to choose correct reduction *(if a choice occurs)*
- Lookaheads are bookkeeping, unless item has $\cdot$ at right end
  > Has no direct use in $[\alpha \rightarrow \beta \gamma \cdot \delta, a]$
  > In $[\alpha \rightarrow \beta \gamma \delta \cdot, a]$, a lookahead of $a$ implies a reduction by $\alpha \rightarrow \beta \gamma \delta$
  > For $\{ [\alpha \rightarrow \gamma \cdot, a], [\beta \rightarrow \gamma \cdot \delta, b] \}$, $a \Rightarrow \text{reduce}$ to $\alpha$; $\text{FIRST}(\delta) \Rightarrow \text{shift}$

⇒ Limited right context is enough to pick the actions
LR(1) Table Construction

High-level overview

1. Build the canonical collection of sets of LR(1) Items, \( I \)
   
   a. Begin in an appropriate state, \( i_0 \)
      
      \( \rightarrow \) \( [S' \rightarrow \cdot S, EOF] \), along with any equivalent items
      
      \( \rightarrow \) Derive equivalent items as \( \text{closure}(i_0) \)

   b. Repeatedly compute, for each \( i_k \), and each \( \alpha \), \( \text{goto}(i_k, \alpha) \)
      
      \( \rightarrow \) If the set is not already in the collection, add it
      
      \( \rightarrow \) 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

   \textit{The canonical collection completely encodes the transition diagram for the handle-finding DFA}
Back to Finding Handles

Revisiting an issue from last class

Parser in a state where the stack (the fringe) was

\[ \text{Expr} \rightarrow \text{Term} \]

With lookahead of *

How did it choose to expand \text{Term} rather than reduce to \text{Expr}?

- Lookahead symbol is the key
- With lookahead of + or -, parser should reduce to \text{Expr}
- With lookahead of * or /, parser should shift
- Parser uses lookahead to decide
- All this context from the grammar is encoded in the handle recognizing mechanism
### Stack Input Handle Action

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Handle</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id - num * id</td>
<td>non e</td>
<td>shift t</td>
</tr>
<tr>
<td>$ id</td>
<td>- num * id</td>
<td>9,1</td>
<td>red . 9</td>
</tr>
<tr>
<td>$ Fac to r</td>
<td>- num * id</td>
<td>7,1</td>
<td>red . 7</td>
</tr>
<tr>
<td>$ Te rm</td>
<td>- num * id</td>
<td>4,1</td>
<td>red . 4</td>
</tr>
<tr>
<td>$ Ex pr</td>
<td>- num * id</td>
<td>non e</td>
<td>shift t</td>
</tr>
<tr>
<td>$ Ex pr</td>
<td>- num * id</td>
<td>non e</td>
<td>shift t</td>
</tr>
<tr>
<td>$ Exp r - num</td>
<td>id</td>
<td>8,3</td>
<td>red . 8</td>
</tr>
<tr>
<td>$ Ex pr - Fac to r</td>
<td>- id</td>
<td>7,3</td>
<td>red . 7</td>
</tr>
<tr>
<td>$ Exp r - Te rm</td>
<td>- id</td>
<td>non e</td>
<td>shift t</td>
</tr>
<tr>
<td>$ Exp r - Te rm *</td>
<td>id</td>
<td>non e</td>
<td>shift t</td>
</tr>
<tr>
<td>$ Ex pr - Te rm * id</td>
<td>- id</td>
<td>9,5</td>
<td>red . 9</td>
</tr>
<tr>
<td>$ Ex pr - Te rm * Fac to r</td>
<td>- id</td>
<td>5,5</td>
<td>red . 5</td>
</tr>
<tr>
<td>$ Ex pr - Te rm</td>
<td>- id</td>
<td>3,3</td>
<td>red . 3</td>
</tr>
<tr>
<td>$ Ex pr</td>
<td>- id</td>
<td>1,1</td>
<td>red . 1</td>
</tr>
<tr>
<td>$ Goal</td>
<td></td>
<td>non e</td>
<td>accept</td>
</tr>
</tbody>
</table>

1. Shift until TOS is the right end of a handle
2. Find the left end of the handle & reduce
Next Class

- Algorithms for FIRST, goto, & closure
- Work an example — a simplified expression grammar

To prepare
- Look at the book or web pages
- Work through the SheepNoise example
Computing FIRST Sets

Define FIRST as

- If $\alpha \Rightarrow^* a\beta$, $a \in T$, $\beta \in (T \cup NT)^*$, then $a \in \text{FIRST}(\alpha)$
- If $\alpha \Rightarrow^* \epsilon$, then $\epsilon \in \text{FIRST}(\alpha)$

To compute FIRST

- Use a fixed-point method
- $\text{FIRST}(\alpha) \in 2^{(T \cup \epsilon)}$
- Loop is monotonic
  $\Rightarrow$ Algorithm halts

For each $\alpha \in T$

\begin{align*}
\text{FIRST}(\alpha) & \leftarrow \alpha \\
\text{FIRST}(\alpha) & \leftarrow \emptyset
\end{align*}

For each $\alpha \in NT$,

\begin{align*}
\text{FIRST}(\alpha) & \leftarrow \emptyset
\end{align*}

While (FIRST sets are still changing)

for each $p \in P$, of the form $\alpha \rightarrow \beta$,

- if $\beta$ is $\epsilon$ then
  \begin{align*}
  \text{FIRST}(\alpha) & \leftarrow \text{FIRST}(\alpha) \cup \{ \epsilon \} \\
  \text{FIRST}(\alpha) & \leftarrow \text{FIRST}(\alpha) \cup \text{FIRST}(\beta_1) \\
  i & \leftarrow 1 \\
  \text{while}(\epsilon \in \text{FIRST}(\beta_i) & i \leq k - 1)
  \begin{align*}
  \text{FIRST}(\alpha) & \leftarrow \text{FIRST}(\alpha) \cup \text{FIRST}(\beta_{i+1}) \\
  i & \leftarrow i + 1
  \end{align*}
\end{align*}