Section: LR Parsing

LR PARSING

LR(k) Parser

- bottom-up parser
- shift-reduce parser
- L means: reads input left to right
- R means: produces a rightmost derivation
- k - number of lookahead symbols

LR parsing process

- convert CFG to PDA
- Use the PDA and lookahead symbols
Convert CFG to PDA

The constructed NPDA:

- three states: s, q, f
  start in state s, assume z on stack
- all rewrite rules in state s, backwards
  rules pop rhs, then push lhs
  \((s, \text{lhs}) \in \delta(s, \lambda, \text{rhs})\)
  This is called a reduce operation.
- additional rules in s to recognize terminals
  For each \(x \in \Sigma, g \in \Gamma, (s, xg) \in \delta(s, x, g)\)
  This is called a shift operation.
- pop S from stack and move into state q
- pop z from stack, move into f, accept.
Example: Construct a PDA.

\[ S \rightarrow aSb \]
\[ S \rightarrow b \]
LR Parsing Actions

1. shift
   transfer the lookahead to the stack

2. reduce
   For $X \rightarrow w$, replace $w$ by $X$ on the stack

3. accept
   input string is in language

4. error
   input string is not in language

LR(1) Parse Table

• Columns:
  terminals, $\$ and variables

• Rows:
  state numbers: represent patterns in a derivation
LR(1) Parse Table Example

1) $S \rightarrow aSb$
2) $S \rightarrow b$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s2</td>
<td>s3</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>acc</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s2</td>
<td>s3</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>r2</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>s5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>r1</td>
<td>r1</td>
<td></td>
</tr>
</tbody>
</table>

Definition of entries:

- sN - shift terminal and move to state N
- N - move to state N
- rN - reduce by rule number N
- acc - accept
- blank - error
state = 0
push(state)
read(symbol)
entry = T[state,symbol]
while entry.action ≠ accept do
    if entry.action == shift then
        push(symbol)
        state = entry.state
        push(state)
        read(symbol)
    else if entry.action == reduce then
        do 2*size_rhs times {pop()}
        state := top-of-stack()
        push(entry.rule.lhs)
        state = T[state,entry.rule.lhs]
        push(state)
    else if entry.action == blank then
        error
        entry = T[state, symbol]
end while
if symbol ≠ $ then error
Example:
Trace aabbb

5
b
3 4 4 5
b S S b
2 2 2 2 4 4
a a a a S S
2 2 2 2 2 2 2 2 1
a a a a a a a a S
0 0 0 0 0 0 0 0 0
S: z z z z z z z z z z
L: a a b b b b b b b $ $
To construct the LR(1) parse table:

- Construct a dfa to model the top of the stack
- Using the dfa, construct an LR(1) parse table

To Construct the DFA

- Add $S' \rightarrow S$
- place a marker "_" on the rhs $S' \rightarrow _S$
- Compute $\text{closure}(S' \rightarrow _S)$.

  Def. of closure:

  1. $\text{closure}(A \rightarrow v_{xy}) = \{A \rightarrow v_{xy}\}$ if $x$ is a terminal.

  2. $\text{closure}(A \rightarrow v_{xy}) = \{A \rightarrow v_{xy}\} \cup \text{closure}(x \rightarrow _w)$ for all $w$ if $x$ is a variable.
• The closure(S’ → _S) is state 0 and “unprocessed”.
• Repeat until all states have been processed
  – unproc = any unprocessed state
  – For each x that appears in A→uxv do
    * Add a transition labeled “x” from state “unproc” to a new state with production A→uxv
    * The set of productions for the new state are: closure(A→uxv)
    * If the new state is identical to another state, combine the states Otherwise, mark the new state as “unprocessed”
• Identify final states.
Example: Construct DFA

(0) $S' \rightarrow S$
(1) $S \rightarrow aSb$
(2) $S \rightarrow b$
Backtracking through the DFA

Consider aabbb

• Start in state 0.
• Shift “a” and move to state 2.
• Shift “a” and move to state 2.
• Shift “b” and move to state 3.
  Reduce by “S → b”
  Pop “b” and Backtrack to state 2.
  Shift “S” and move to state 4.
• Shift “b” and move to state 5.
  Reduce by “S → aSb”
  Pop “aSb” and Backtrack to state 2.
  Shift “S” and move to state 4.
• Shift “b” and move to state 5.
  Reduce by “S → aSb”
  Pop “aSb” and Backtrack to state 0.
Shift “S” and move to state 1.

- Accept. aabbb is in the language.
To construct LR(1) table from diagram:

1. If there is an arc from state1 to state2
   (a) arc labeled x is terminal or $\ \ \ \ T[\text{state1, x}] = \text{sh state2}$
   (b) arc labeled X is nonterminal $\ \ \ \ T[\text{state1, X}] = \text{state2}$

2. If state1 is a final state with $X \rightarrow w$
   For all a in FOLLOW(X), $\ \ \ \ T[\text{state1,a}] = \text{reduce by X} \rightarrow w$

3. If state1 is a final state with $S' \rightarrow S$
   $\ \ \ \ T[\text{state1,\$}] = \text{accept}$

4. All other entries are error
Example: LR(1) Parse Table

(0) $S' \rightarrow S$
(1) $S \rightarrow aSb$
(2) $S \rightarrow b$

Here is the LR(1) Parse Table with extra information about the stack contents of each state.

<table>
<thead>
<tr>
<th>Stack contents</th>
<th>State number</th>
<th>Terminals</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>(empty)</td>
<td>0</td>
<td>a b $</td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Actions for entries in LR(1) Parse table $T[state,symbol]$

Let entry $= T[state,symbol]$.

- **If symbol is a terminal or $\$$**
  - If entry is “shift state$i$”
    push lookahead and state$i$ on the stack
  - If entry is “reduce by rule $X \rightarrow w$”
    pop $w$ and $k$ states ($k$ is the size of $w$) from the stack.
  - If entry is “accept”
    Halt. The string is in the language.
  - If entry is “error”
    Halt. The string is not in the language.
• If symbol is nonterminal
  We have just reduced the rhs of a production $X \rightarrow w$ to a symbol. The entry is a state number, call it state$_i$. Push $T[\text{state}_i, X]$ on the stack.
Constructing Parse Tables for CFG’s with \( \lambda \)-rules

\[ A \rightarrow \lambda \] written as \( A \rightarrow \lambda \_ \)

Example

\[
\begin{align*}
S & \rightarrow \text{ddX} \\
X & \rightarrow aX \\
X & \rightarrow \lambda
\end{align*}
\]

Add a new start symbol and number the rules:

\[
(0) \quad S' \rightarrow S \\
(1) \quad S \rightarrow \text{ddX} \\
(2) \quad X \rightarrow aX \\
(3) \quad X \rightarrow \lambda
\]

Construct the DFA:
Construct the LR(1) Parse Table

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>d</th>
<th>$</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</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></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Possible Conflicts:

1. Shift/Reduce Conflict
   Example:
   
   \[
   \begin{align*}
   & A \rightarrow ab \\
   & A \rightarrow abcd
   \end{align*}
   \]

   In the DFA:
   
   \[
   \begin{align*}
   & A \rightarrow ab_ \_ \\
   & A \rightarrow ab_ \_ cd
   \end{align*}
   \]

2. Reduce/Reduce Conflict
   Example:
   
   \[
   \begin{align*}
   & A \rightarrow ab \\
   & B \rightarrow ab
   \end{align*}
   \]

   In the DFA:
   
   \[
   \begin{align*}
   & A \rightarrow ab_ \_ \\
   & B \rightarrow ab_ \_
   \end{align*}
   \]

3. Shift/Shift Conflict