Lecture 6
CS 473: COMPILER DESIGN
Shift/Reduce Parsing

- Parser state:
  - Stack holds term so far (terminals and nonterminals)
  - Remaining input is a string of terminals
  - Look at stack and input to decide next step
- Parsing is a sequence of \textit{shift} and \textit{reduce} operations:
  - \textbf{Shift}: move look-ahead token to the stack
  - \textbf{Reduce}: Replace symbols $\gamma$ at top of stack with nonterminal $X$ such that $X \rightarrow \gamma$ is a production. (pop $\gamma$, push $X$)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(1 + 2 + (3 + 4)) + 5</td>
<td>shift (</td>
</tr>
<tr>
<td>(</td>
<td>1 + 2 + (3 + 4)) + 5</td>
<td>shift 1</td>
</tr>
<tr>
<td>(1</td>
<td>+ 2 + (3 + 4)) + 5</td>
<td>reduce: $E \rightarrow$ number</td>
</tr>
<tr>
<td>(E</td>
<td>+ 2 + (3 + 4)) + 5</td>
<td>reduce: $S \rightarrow E$</td>
</tr>
<tr>
<td>(S</td>
<td>+ 2 + (3 + 4)) + 5</td>
<td>shift +</td>
</tr>
<tr>
<td>(S +</td>
<td>2 + (3 + 4)) + 5</td>
<td>shift 2</td>
</tr>
<tr>
<td>(S + 2</td>
<td>+ (3 + 4)) + 5</td>
<td>reduce: $E \rightarrow$ number</td>
</tr>
</tbody>
</table>
Action Selection Problem

• Given a stack $\sigma$ and a lookahead symbol $b$, should the parser:
  – Shift $b$ onto the stack (new stack is $\sigma b$)?
  – Reduce by a production $X \rightarrow \gamma$, replacing $\gamma$ with $X$ on the stack?

• Sometimes the parser can reduce but shouldn’t
  – For example, $X \rightarrow \varepsilon$ can always be reduced

• Sometimes the stack can be reduced in multiple different ways

• Main idea: construct a DFA that tracks progress through all the productions we might be in the middle of
• If a reduce state is reached, reduce

• Otherwise, shift, and then follow the transition for the token shifted

• If no such transition, parse error

• After reducing, re-run the DFA on the stack to find the new state
Full DFA for the Example

Stack: ( id. (state 2)  
Stack after reducing: ( S. (state 7)  
Stack after reducing again: ( L. (state 5)  
Done!
Using the DFA

- Run the parser stack through the DFA
- The resulting state tells us which productions might be reduced next.
  - If not in a reduce state, then shift the next symbol and transition according to DFA
  - If in a reduce state $X \longmapsto \gamma$ with stack $\alpha \gamma$, pop $\gamma$ and push $X$
  - Then re-run the DFA again

- Optimization: No need to re-run the DFA from beginning every step
  - Store the state with each symbol on the stack: e.g. $1(3(3L5)_6$
  - On a reduction $X \longmapsto \gamma$, pop stack to reveal the state too: e.g. From stack $1(3(3L5)_6$ reduce $S \longmapsto (L)$ to reach stack $1(3$
  - Next, push the reduction symbol: $1(3S$
  - Then take the corresponding step in the DFA to find next state: $1(3S_7$
Implementing the Parsing Table

- Represent the DFA as a table with states as rows, **terminals** and **nonterminals** as columns
- For non-reduce states, if the top of the stack is a nonterminal, check the “goto table” for the next state; otherwise, shift and check the “action table” for the next state
- For reduce states, reduce according to the production listed

<table>
<thead>
<tr>
<th>State</th>
<th>Terminal Symbols</th>
<th>Nonterminal Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Action table</strong></td>
<td><strong>Goto table</strong></td>
</tr>
</tbody>
</table>
## Example Parse Table

<table>
<thead>
<tr>
<th></th>
<th>(</th>
<th>)</th>
<th>id</th>
<th>,</th>
<th>$</th>
<th>S</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td>g4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S ⟷ id</td>
<td>S ⟷ id</td>
<td>S ⟷ id</td>
<td>S ⟷ id</td>
<td>S ⟷ id</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g7</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>s6</td>
<td>s8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>S ⟷ (L)</td>
<td>S ⟷ (L)</td>
<td>S ⟷ (L)</td>
<td>S ⟷ (L)</td>
<td>S ⟷ (L)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>L ⟷ S</td>
<td>L ⟷ S</td>
<td>L ⟷ S</td>
<td>L ⟷ S</td>
<td>L ⟷ S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>s3</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>g9</td>
</tr>
<tr>
<td>9</td>
<td>L ⟷ L,S</td>
<td>L ⟷ L,S</td>
<td>L ⟷ L,S</td>
<td>L ⟷ L,S</td>
<td>L ⟷ L,S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

sx = shift and goto state x  
gx = goto state x
Example

- Parse the token stream: \((x, (y, z), w)\)$

<table>
<thead>
<tr>
<th>Stack</th>
<th>Stream</th>
<th>Action (according to table)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\epsilon_1)</td>
<td>((x, (y, z), w))$</td>
<td>s3</td>
</tr>
<tr>
<td>(\epsilon_1(3))</td>
<td>((x, (y, z), w))$</td>
<td>s2</td>
</tr>
<tr>
<td>(\epsilon_1(3x_2))</td>
<td>((x, (y, z), w))$</td>
<td>Reduce: S (\rightarrow) id</td>
</tr>
<tr>
<td>(\epsilon_1(3S))</td>
<td>((x, (y, z), w))$</td>
<td>g7 (from state 3 follow S)</td>
</tr>
<tr>
<td>(\epsilon_1(3S_7))</td>
<td>((x, (y, z), w))$</td>
<td>Reduce: L (\rightarrow) S</td>
</tr>
<tr>
<td>(\epsilon_1(3L))</td>
<td>((x, (y, z), w))$</td>
<td>g5 (from state 3 follow L)</td>
</tr>
<tr>
<td>(\epsilon_1(3L_5))</td>
<td>((x, (y, z), w))$</td>
<td>s8</td>
</tr>
<tr>
<td>(\epsilon_1(3L_5,8))</td>
<td>((x, (y, z), w))$</td>
<td>s3</td>
</tr>
<tr>
<td>(\epsilon_1(3L_5,8(3))</td>
<td>((x, (y, z), w))$</td>
<td>s2</td>
</tr>
</tbody>
</table>
LR(0) Limitations

• An LR(0) machine only works if reduce states have exactly one reduce action
  – In such states, the machine *always* reduces

• With more complex grammars, the DFA construction will yield states with *shift/reduce and reduce/reduce conflicts*:

  OK  shift/reduce  reduce/reduce

  \[
  S \rightarrow ( L ).
  \]
  \[
  S \rightarrow ( L ).
  L \rightarrow .L , S
  \]
  \[
  S \rightarrow L , S.
  S \rightarrow , S.
  \]

• Such conflicts can often be resolved by using a look-ahead symbol: LR(1)
Examples

• Consider the left associative and right associative “sum” grammars:

\[
\text{left} \\
S \rightarrow S + E \mid E \\
E \rightarrow \text{number} \mid ( S )
\]

\[
\text{right} \\
S \rightarrow E + S \mid E \\
E \rightarrow \text{number} \mid ( S )
\]

• One is LR(0), the other isn’t... which is which and why?
• What kind of conflict do you get? Shift/reduce or reduce/reduce?

• Ambiguities in associativity/precedence usually lead to shift/reduce conflicts.
LR(1) Parsing

• Algorithm is similar to LR(0) DFA construction:
  – LR(1) state = set of LR(1) items
  – An LR(1) item is an LR(0) item + a set of look-ahead symbols: 
    \[ A \rightarrow \alpha.\beta, L \]

• LR(1) closure is a little more complex:
  • Form the set of items just as for LR(0) algorithm.
  • Whenever a new item \( C \rightarrow .\gamma \) is added because \( A \rightarrow \beta.C\delta, L \) is already in the set, we need to compute its look-ahead set \( M \):
    1. The look-ahead set \( M \) includes \( \text{FIRST}(\delta) \) 
       (the set of terminals that may start strings derived from \( \delta \) )
    2. If \( \delta \) can derive \( \varepsilon \) (it is nullable), then the look-ahead \( M \) also contains \( L \)
Example Closure

- Start item: \( T \rightarrow .S$ \), {}  
- Since \( S \) is to the right of a ‘.’, add:  
  \( S \rightarrow .E + S \), {$} \quad \text{Note: {$} \text{is FIRST($) } \)  
  \( S \rightarrow .E \), {$}  
- Need to keep closing, since \( E \) appears to the right of a ‘.’ in ‘.E + S’:
  \( E \rightarrow .\text{number} \), {+} \quad \text{Because \( E \) can be followed by +}  
  \( E \rightarrow .( S ) \), {+}  
- Because \( E \) also appears to the right of ‘.’ in ‘.E’ we get:  
  \( E \rightarrow .\text{number} \), {$} \quad \text{Because \( E \) can be followed by \( \varepsilon \) } \)  
  \( E \rightarrow .( S ) \), {$}  
- No more new nonterminals after ‘,’ so we’re done

\[
\begin{align*}
T & \rightarrow S$
S \rightarrow E + S \mid E \\
E \rightarrow \text{number} \mid ( S )
\end{align*}
\]
Using the DFA

Fragment of the Action & Goto tables

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>$</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>g2</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
<td>S $E$</td>
<td></td>
</tr>
</tbody>
</table>

Choice between shift and reduce is resolved.
• The behavior is determined if:
  – There is no overlap among the look-ahead sets for each reduce item, and
  – None of the look-ahead symbols appear to the right of a ‘.’

Fragment of the Action & Goto tables

Choice between shift and reduce is resolved.
Questions
LR variants

• LR(1) gives maximal power out of a 1 look-ahead symbol parsing table
  – DFA + stack is a pushdown automaton
• In practice, LR(1) tables are very big!
  – Modern implementations (e.g. bison) directly generate code

• LALR(1) = “Look-ahead LR”
  – Merge any two LR(1) states whose items are identical except for look-ahead sets:
    
    \[
    \begin{align*}
    & E \rightarrow V. \ {\{\$\}} \\
    & E \rightarrow V. \ {\{=\}} \\
    \end{align*}
    \]
  – Can in theory lead to reduce/reduce conflicts, but:
    – Results in a much smaller parse table and works well in practice
    – This is the usual technology for automatic parser generators: yacc, bison

• GLR = “Generalized LR” parsing
  – Efficiently compute the set of all parses for a given input
  – Later passes should disambiguate based on other context
Classification of Grammars

- CFG
- LR(1)
- LALR(1)
- LL(1)
- SLR
- LR(0)

The diagram shows the classification of grammars, with LL(1) being a subset of LR(1), which is a subset of LALR(1), which is a subset of LR(1).
Questions