CSE P 501 – Compilers

LR Parsing
Hal Perkins
Spring 2018
Agenda

• LR Parsing
• Table-driven Parsers
• Parser States
• Shift-Reduce and Reduce-Reduce conflicts
Bottom-Up Parsing

- Idea: Read the input left to right
- Whenever we’ve matched the right hand side of a production, reduce it to the appropriate non-terminal and add that non-terminal to the parse tree
- The upper edge of this partial parse tree is known as the *frontier*
Example

- Grammar

\[
S ::= aAB \ e \\
A ::= Abc \mid b \\
B ::= d
\]

- Bottom-up Parse

![Bottom-up Parse Tree]
LR(1) Parsing

• We’ll look at LR(1) parsers
  – Left to right scan, Rightmost derivation, 1 symbol lookahead
  – Almost all practical programming languages have a LR(1) grammar
  – LALR(1), SLR(1), etc. – subsets of LR(1)
    • LALR(1) can parse most real languages, tables are more compact, and is used by YACC/Bison/CUP/etc.
LR Parsing in Greek

- The bottom-up parser reconstructs a reverse rightmost derivation
- Given the rightmost derivation
  \[ S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w \]
  the parser will first discover \( \beta_{n-1} \Rightarrow \beta_n \), then \( \beta_{n-2} \Rightarrow \beta_{n-1} \), etc.
- Parsing terminates when
  - \( \beta_1 \) reduced to \( S \) (start symbol, success), or
  - No match can be found (syntax error)
How Do We Parse with This?

- Key: given what we’ve already seen and the next input symbol (the lookahead), decide what to do.
- Choices:
  - Perform a reduction
  - Look ahead further
- Can reduce $A \Rightarrow \beta$ if both of these hold:
  - $A \Rightarrow \beta$ is a valid production, and
  - $A \Rightarrow \beta$ is a step in this rightmost derivation
- This is known as a shift-reduce parser
Sentential Forms

- If $S =>^* \alpha$, the string $\alpha$ is called a sentential form of the grammar
- In the derivation
  $S => \beta_1 => \beta_2 => ... => \beta_{n-2} => \beta_{n-1} => \beta_n = w$
each of the $\beta_i$ are sentential forms
- A sentential form in a rightmost derivation is called a right-sentential form (similarly for leftmost and left-sentential)
Handles

- Informally, a production whose right hand side matches a substring of the tree frontier \textit{that is part of the rightmost derivation of the current input string} (i.e., the "correct" production)
  - Even if $A ::= \beta$ is a production, it is a handle only if $\beta$ matches the frontier at a point where $A ::= \beta$ was used in \textit{this specific} derivation
  - $\beta$ may appear in many other places in the frontier without designating a handle
- Bottom-up parsing is all about finding handles
Handle Examples

• In the derivation
  \[ S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde \]
  – abbcde is a right sentential form whose handle is \textcolor{red}{A::=b} at position 2
  – a\textcolor{red}{Abc}de is a right sentential form whose handle is \textcolor{red}{A::=Abc} at position 4

• Note: some books take the left end of the match as the position
Handles – The Dragon Book Defn.

• Formally, a handle of a right-sentential form $\gamma$ is a production $A ::= \beta$ and a position in $\gamma$ where $\beta$ may be replaced by $A$ to produce the previous right-sentential form in the rightmost derivation of $\gamma$. 
Implementing Shift-Reduce Parsers

- Key Data structures
  - A stack holding the frontier of the tree
  - A string with the remaining input (tokens)
- We also need something to encode the rules that tell us what action to take next, given the state of the stack and the lookahead symbol
  - Typically a table that encodes a finite automata
Shift-Reduce Parser Operations

- **Reduce** – if the top of the stack is the right side of a handle $A::=\beta$, pop the right side $\beta$ and push the left side $A$
- **Shift** – push the next input symbol onto the stack
- **Accept** – announce success
- **Error** – syntax error discovered
Shift-Reduce Example

\[
S ::= aABe \\
A ::= Abc \mid b \\
B ::= d
\]

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>abbcde$</td>
<td>shift</td>
</tr>
<tr>
<td>$a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aAb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aAAb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aAbc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aAAd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aAB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$aABd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How Do We Automate This?

- Cannot use clairvoyance in a real parser (alas...)
- Defn. **Viable prefix** – a prefix of a right-sentential form that can appear on the stack of the shift-reduce parser
  - Equivalent: a prefix of a right-sentential form that does not continue past the rightmost handle of that sentential form
  - In Greek: γ is a **viable prefix** of G if there is some derivation \( S \Rightarrow^{*_{rm}} \alpha Aw \Rightarrow^{*_{rm}} \alpha \beta w \) and γ is a prefix of \( \alpha \beta \).
  - The occurrence of \( \beta \) in \( \alpha \beta w \) is a **handle** of \( \alpha \beta w \)
How Do We Automate This?

- Fact: the set of viable prefixes of a CFG is a regular language(!)

- Idea: Construct a DFA to recognize viable prefixes given the stack and remaining input
  - Perform reductions when we recognize them
DFA for prefixes of

\[ S ::= a A B e \]
\[ A ::= A b c \mid b \]
\[ B ::= d \]
Trace

\[ S ::= aABe \]
\[ A ::= Abc \mid b \]
\[ B ::= d \]
Observations

• Way too much backtracking
  – We want the parser to run in time proportional to the length of the input

• Where the heck did this DFA come from anyway?
  – From the underlying grammar
  – Defer construction details for now
Avoiding DFA Rescanning

- Observation: no need to restart DFA after a shift. Stay in the same state and process next token.
- Observation: after a reduction, the contents of the stack are the same as before except for the new non-terminal on top
  - ∴ Scanning the stack will take us through the same transitions as before until the last one
  - ∴ If we record state numbers on the stack, we can go directly to the appropriate state when we pop the right hand side of a production from the stack
Stack

- Change the stack to contain pairs of states and symbols from the grammar
  \[ s_0 \ X_1 \ s_1 \ X_2 \ s_2 \ ... \ X_n \ s_n \]
  - State \( s_0 \) represents the accept (start) state
    (Not always explicitly on stack – depends on particular presentation)
  - When we push a symbol on the stack, push the symbol plus the FA state
  - When we reduce, popping the handle will reveal the state of the FA just prior to reading the handle

- Observation: in an actual parser, only the state numbers are needed since they implicitly contain the symbol information. But for explanations / examples it can help to show both.
Encoding the DFA in a Table

• A shift-reduce parser’s DFA can be encoded in two tables
  — One row for each state
  — action table encodes what to do given the current state and the next input symbol
  — goto table encodes the transitions to take after a reduction
Actions (1)

• Given the current state and input symbol, the main possible actions are
  
  — $s_i$ – shift the input symbol and state $i$ onto the stack (i.e., shift and move to state $i$)
  
  — $r_j$ – reduce using grammar production $j$

• The production number tells us how many $<\text{symbol, state}>$ pairs to pop off the stack (= number of symbols on rhs of production)
Actions (2)

- Other possible *action* table entries
  - *accept*
  - *blank* – no transition – syntax error
    - A LR parser will detect an error as soon as possible on a left-to-right scan
    - A real compiler needs to produce an error message, recover, and continue parsing when this happens
Goto

• When a reduction is performed using $A ::= \beta$, we pop $|\beta| <\text{symbol, state}>$ pairs from the stack revealing a state $uncovered_s$ on the top of the stack

• $\text{goto}[uncovered_s, A]$ is the new state to push on the stack when reducing production $A ::= \beta$ (after popping handle $\beta$ and pushing $A$)
Reminder: DFA for

\[
S ::= aABe
\]

\[
A ::= Abc \mid b
\]

\[
B ::= d
\]
LR Parse Table for

1. $S ::= aABe$
2. $A ::= Abc$
3. $A ::= b$
4. $B ::= d$

<table>
<thead>
<tr>
<th>State</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>acc</td>
</tr>
<tr>
<td>1</td>
<td>s2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>s4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>s6</td>
</tr>
<tr>
<td>4</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>5</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>s7</td>
</tr>
<tr>
<td>7</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>r1</td>
</tr>
</tbody>
</table>

UW CSE P 501 Spring 2018
LR Parsing Algorithm

tok = scanner.getToken();
while (true) {
  s = top of stack;
  if (action[s, tok] = si) {
    push tok; push i (state);
    tok = scanner.getToken();
  } else if (action[s, tok] = rj) {
    pop 2 * length of right side of
    production j (2* | B|);
    uncovered_s = top of stack;
    push left side A of production j ;
    push state goto[uncovered_s, A];
  }
} else if (action[s, tok] = accept) {
  return;
} else {
  // no entry in action table
  report syntax error;
  halt or attempt recovery;
}
Example

1. $S ::= aABe$
2. $A ::= Abc$
3. $A ::= b$
4. $B ::= d$

Stack
$0$
$0a2$
$0a2b4$
$0a2A3$
$0a2A3b6$
$0a2A3b6c7$
$0a2A3$
$0a2A3b5$
$0a2A3b8$
$0a2A3b8c9$
$050$

Input
abbcde$ s2$
bcde$
bcde$
bcde$
bcde$
d$ s2$
bcde$
bcde$
d$ s4$
d$ s6$
s5$

<table>
<thead>
<tr>
<th>S</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>$</th>
<th>A</th>
<th>B</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$</td>
<td></td>
<td></td>
<td>ac</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>s2</td>
<td></td>
<td></td>
<td></td>
<td>$</td>
<td></td>
<td></td>
<td>g0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>s4</td>
<td></td>
<td></td>
<td></td>
<td>$</td>
<td></td>
<td></td>
<td>g3</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>s6</td>
<td>s5</td>
<td></td>
<td></td>
<td>$</td>
<td></td>
<td></td>
<td>g8</td>
</tr>
<tr>
<td>4</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>s7</td>
<td></td>
<td></td>
<td></td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>s9</td>
<td></td>
<td></td>
<td></td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UW CSE P 501 Spring 2018
LR States

- Idea is that each state encodes
  - The set of all possible productions that we could be looking at, given the current state of the parse, and
  - *Where* we are in the right hand side of each of those productions
Items

- An *item* is a production with a dot in the right hand side.
- Example: Items for production $A ::= X Y$
  
  \[
  A ::= . X Y \\
  A ::= X . Y \\
  A ::= X Y .
  \]

- Idea: The dot represents a position in the production.
DFA for

\[ S ::= aABe \]
\[ A ::= Abc | b \]
\[ B ::= d \]
Problems with Grammars

• Grammars can cause problems when constructing a LR parser
  – Shift-reduce conflicts
  – Reduce-reduce conflicts
Shift-Reduce Conflicts

• Situation: both a shift and a reduce are possible at a given point in the parse (equivalently: in a particular state of the DFA)
• Classic example: if-else statement
  \[ S ::= \text{ifthen } S \mid \text{ifthen } S \text{ else } S \]
Parser States for

1. \( S ::= \text{ifthen } S \)
2. \( S ::= \text{ifthen } S \text{ else } S \)

- State 3 has a shift-reduce conflict
  - Can shift past else into state 4 \((s4)\)
  - Can reduce \((r1)\)
    \( S ::= \text{ifthen } S \)

(Note: other \( S ::= . \text{ifthen} \) items not included in states 2-4 to save space)
Solving Shift-Reduce Conflicts

- Fix the grammar
  - Done in Java reference grammar, others
- Use a parse tool with a “longest match” rule – i.e., if there is a conflict, choose to shift instead of reduce
  - Does exactly what we want for if-else case
  - Guideline: a few shift-reduce conflicts are fine, but be sure they do what you want (and that this behavior is guaranteed by the tool specification)
Reduce-Reduce Conflicts

- Situation: two different reductions are possible in a given state
- Contrived example
  
  \[
  S ::= A \\
  S ::= B \\
  A ::= x \\
  B ::= x
  \]
Parser States for

1. \( S ::= A \)
2. \( S ::= B \)
3. \( A ::= x \)
4. \( B ::= x \)

- State 2 has a reduce-reduce conflict (r3, r4)
Handling Reduce-Reduce Conflicts

- These normally indicate a serious problem with the grammar.
- Fixes
  - Use a different kind of parser generator that takes lookahead information into account when constructing the states
    - Most practical tools use this information
  - Fix the grammar
Another Reduce-Reduce Conflict

• Suppose the grammar tries to separate arithmetic and boolean expressions
  
  \[
  \text{expr ::= aexp | bexp} \\
  \text{aexp ::= aexp \_ aident | aident} \\
  \text{bexp ::= bexp \&\& bident | bident} \\
  \text{aident ::= id} \\
  \text{bident ::= id}
  \]

• This will create a reduce-reduce conflict after recognizing \text{id}
Covering Grammars

• A solution is to merge *aident* and *bident* into a single non-terminal like *ident* (or just use *id* in place of *aident* and *bident* everywhere they appear)

• This is a *covering grammar*
  – Will generate some programs (sentences) that are not generated by the original grammar
  – Use the type checker or other static semantic analysis to weed out illegal programs later
Coming Attractions

• Constructing LR tables
  — We’ll present a simple version (SLR(0)) in lecture, then talk about adding lookahead and then a little bit about how this relates to LALR(1) used in most parser generators

• LL parsers and recursive descent

• Continue reading ch. 3