LR Parsing
Hal Perkins
Autumn 2009
Agenda

- LR Parsing
- Table-driven Parsers
- Parser States
- Shift-Reduce and Reduce-Reduce conflicts
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 an LR(1) grammar
  - LALR(1), SLR(1), etc. – subsets of LR(1)
    - LALR(1) can parse most real languages, is more compact, and is used by YACC/Bison/etc.
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

\[ S ::= aABe \]
\[ A ::= A\text{bc} | b \]
\[ B ::= d \]

- Bottom-up Parse

a b b c d e
Details

- 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, 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
  - $A \Rightarrow \beta$ is a step in this rightmost derivation

- This is known as a *shift-reduce* parser
Sentential Forms

- If $S \Rightarrow^* \alpha$, the string $\alpha$ is called a *sentential form* of the grammar.

- In the derivation
  
  $$S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-2} \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w$$

  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 substring of the tree frontier that matches the right side of a production
  - Even if $A ::= \beta$ is a production, $\beta$ is a handle only if it matches the frontier at a point where $A ::= \beta$ was used in the derivation
  - $\beta$ may appear in many other places in the frontier without being a handle for that particular production
Handles (cont.)

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 $\gamma$. 
Handle Examples

- In the derivation
  \[ S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde \]
  - abbcde is a right sentential form whose handle is \( A::=b \) at position 2
  - aAbcde is a right sentential form whose handle is \( A::=Abc \) at position 4
    - Note: some books take the left of the match as the position
Implementing Shift-Reduce Parsers

- Key Data structures
  - A stack holding the frontier of the tree
  - A string with the remaining input
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

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

<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>
</tbody>
</table>
How Do We Automate This?

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

\[
S ::= aA\tilde{B}e \\
A ::= A\text{bc} \mid b \\
B ::= d
\]

\[
S ::= aA\tilde{B}e \\
\]

![DFA Diagram]

- Start at node 1.
- Accept at node 9.
- \(a\) transitions:
  - From 1 to 2
  - From 2 to 3
- \(b\) transitions:
  - From 1 to 4
  - From 2 to 4
  - From 3 to 6
- \(c\) transitions:
  - From 6 to 7
- \(d\) transitions:
  - From 2 to 5
  - From 3 to 5
- \(e\) transitions:
  - From 8 to 9
Trace

Stack
$
$

Input
$abbcde$

$S ::= a A B e$
$A ::= A bc \mid b$
$B ::= d$

![Diagram of lexer stack trace]
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
  - We’ll defer construction details for now
Avoiding DFA Rescanning

- 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 $s_0 X_1 S_1 X_2 S_2 \ldots X_n S_n$
- State $s_0$ represents the accept state
  - (Not always added – depends on particular presentation)

- Observation: in an actual parser, only the state numbers need to be pushed, since they implicitly contain the symbol information, but for explanations, it’s clearer to use both.
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
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, \(<\text{symbol}, \text{state}>\) pairs are popped from the stack revealing a state \textit{uncovered}_s on the top of the stack.

- \texttt{goto[uncovered}_s, A] is the new state to push on the stack when reducing production \(A ::= \beta\) (after popping \(\beta\) and finding state \textit{uncovered}_s on top).
Reminder: DFA for

\[ S ::= aABe \]
\[ A ::= Abc | b \]
\[ B ::= d \]
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></td>
<td>a</td>
<td>b</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></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></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>
LR Parsing Algorithm (1)

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

Stack
$ abbcde$

Input

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

<table>
<thead>
<tr>
<th>S</th>
<th>action</th>
<th>goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>S</td>
</tr>
<tr>
<td>b</td>
<td>c</td>
<td>ac</td>
</tr>
<tr>
<td>c</td>
<td>d</td>
<td>g1</td>
</tr>
<tr>
<td>d</td>
<td>e</td>
<td>g3</td>
</tr>
<tr>
<td>e</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>s2</td>
<td>ac</td>
</tr>
<tr>
<td>2</td>
<td>s4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>s6</td>
<td>g3</td>
</tr>
<tr>
<td>4</td>
<td>r3</td>
<td>g8</td>
</tr>
<tr>
<td>5</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td></td>
</tr>
</tbody>
</table>

10/13/2009 © 2002-09 Hal Perkins & UW CSE D-28
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 $A ::= XY$
  
  $A ::= .XY$
  
  $A ::= X.Y$
  
  $A ::= XY.$

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

\[ S ::= aABe \]

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

\[ B ::= d \]
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. \( S ::= \) ifthen \( S \)
2. \( S ::= \) ifthen \( S \) else \( S \)

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

(Note: other \( S ::= \) .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
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 (LR(1) instead of SLR(1) for example)
  - Most practical tools use this information
- Fix the grammar
Another Reduce-Reduce Conflict

Suppose the grammar separates arithmetic and boolean expressions

\[ expr ::= aexp | bexp \]
\[ aexp ::= aexp * aident | aident \]
\[ bexp ::= bexp && bident | bident \]
\[ aident ::= id \]
\[ bident ::= id \]

This will create a reduce-reduce conflict
Covering Grammars

- A solution is to merge \textit{aident} and \textit{bident} into a single non-terminal (or use \textit{id} in place of \textit{aident} and \textit{bident} everywhere they appear)

- This is a \textit{covering grammar}
  - Includes some programs 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 extending it to LR(1)
- LL parsers and recursive descent
- Continue reading ch. 4