CSE 582 – Compilers

LR Parsing
Hal Perkins
Autumn 2002

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
  - Most practical programming languages have an LR(1) grammar
  - LALR(1), SLR(1), etc. – subsets of LR(1)
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

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

Bottom-up Parse

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
</tbody>
</table>

Details

The bottom-up parser reconstructs a reverse rightmost derivation
Given the rightmost derivation
\[ S \Rightarrow \beta_1 \Rightarrow \beta_2 \Rightarrow \ldots \Rightarrow \beta_{n-1} \Rightarrow \beta_n = w \] 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 \( S \) (success), or
- No match can be found (syntax error)
How Do We Automate 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 => β if both of these hold:
  - A => β is a valid production
  - A => β is a step in this rightmost derivation
- This is known as a shift-reduce parser

Sentential Forms

- If S =>* α, the string α is called a sentential form of the grammar.
- In the derivation
  \[ S => \beta_1 => \beta_2 => \ldots => \beta_n = w \]
  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 ::= β is a production, β is a handle only if it matches the frontier at a point where A ::= β was used in the derivation
- β may appear in many other places in the frontier without being a handle for that 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

$S \Rightarrow aAde \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$

$abbcde$ 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 (e.g., Dragon Book).

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

<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

\[
S ::= aA\text{b}e
\]

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

\[
B ::= d
\]

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
  - 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
  \[S_0 \ X_1 \ S_1 \ X_1 \ S_1 \ldots X_n \ S_n\]
  - 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.

Encoding the DFA in a Table

- A shift-reduce parser’s DFA can be encoded in two tables
  - One row for each state
  - \textit{action} table encodes what to do given the current state and the next input symbol
  - \textit{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/ – shift the input symbol and state i onto the stack (i.e., shift and move to state i)
  - rj – reduce using grammar production j
    - The production number tells us how many <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 done, <symbol, state> pairs are popped from the stack revealing a state uncovered_s on the top of the stack
- goto[uncovered_s, A] is the new state to push on the stack when reducing production A ::= β (after popping β)
Reminder: DFA for

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

LR Parse Table for

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

LR Parsing Algorithm (1)

```java
word = scanner.getToken();
while (true) {
    s = top of stack;
    if (action[s, word] = si) {
        push word; push i (state);
        word = scanner.getToken();
    } else if (action[s, word] = r j) {
        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

```
S ::= aABe
A ::= Abc
A ::= b
B ::= d
```

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>abbcde$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>r2</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>r4</td>
<td>a</td>
</tr>
<tr>
<td>3</td>
<td>s5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>r1</td>
<td>s6</td>
</tr>
<tr>
<td>5</td>
<td>r2</td>
<td>s5</td>
</tr>
<tr>
<td>6</td>
<td>s7</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>r1</td>
<td>s6</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>r1</td>
<td>s5</td>
</tr>
</tbody>
</table>

### 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

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

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**

- 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
- 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
- Moral: a few shift-reduce conflicts are fine, but be sure that 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. \[ S ::= A \\
    S ::= B \\
    A ::= x \\
    B ::= x \]

2. \[ x ::= A \\
    B ::= x \]

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 already use this information
  - Fix the grammar

Another Reduce-Reduce Conflict

- Suppose the grammar separates arithmetic and boolean expressions
  
  \[
  \begin{align*}
  expr &::= aexp \mid bexp \\
  aexp &::= aexp \ast \text{id} \mid \text{id} \\
  bexp &::= \text{bexp} \&\& \text{bident} \mid \text{bident} \\
  \text{id} &::= \text{id} \\
  \text{bident} &::= \text{id}
  \end{align*}
  \]

- This will create a reduce-reduce conflict
Covering Grammars

- A solution is to merge `aident` and `bident` into a single non-terminal (or use `id` in place of `aident` and `bident` everywhere they appear).
- This is a 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. 3