Reminders/Announcements

- Homework 1 is due TODAY, 11:59pm
- No class or office hours on Monday (MLK day)
Agenda

- Finish discussing the “if-else” ambiguity
- Start our first parsing algorithm: LR Parsing

Reminder: “if-else” ambiguity

- Grammar for conditional statements
  
  \[
  stmt ::= \text{if } ( \text{cond} ) \text{stmt} \\
  | \text{if } ( \text{cond} ) \text{stmt} \text{ else stmt}
  \]

- This is ambiguous
  - Consider
    
    \[\text{if (a) if (b) s1 else s2}\]
Derive if(c1) if(c2) s1 else s2

\[
stmt ::= \begin{cases}
  \text{if ( cond ) } stmt \\
  \text{if ( c1 ) } stmt \\
  \text{if ( c1 ) if ( cond ) } stmt \text{ else } stmt \\
  \ldots \\
  \text{if ( c1 ) if ( c2 ) } s1 \text{ else } s2
\end{cases}
\]
Solving “if” Ambiguity

• Fix the grammar to separate if statements with else clause and if statements with no else
  – Done in Java reference grammar
  – Adds lots of non-terminals

• or, Change the language
  – But it’d better be ok to do this

• or, Use some ad-hoc rule in the parser
  – “else matches closest unpaired if”
Resolving Ambiguity with Grammar

```
Stmt ::= MatchedStmt | UnmatchedStmt
MatchedStmt ::= ...
   | if ( Expr ) MatchedStmt else MatchedStmt
UnmatchedStmt ::= if ( Expr ) Stmt |
   | if ( Expr ) MatchedStmt else UnmatchedStmt
```

– Prevents if-without-else as then clause of if-then-else, forcing else to match closest if. But, can still generate exact same language (try it!)
– formal, no additional rules beyond syntax

Check: if (c1) if (c2) stmt else stmt
Resolving Ambiguity with Grammar (2)

- If you can (re-)design the language, can avoid the problem entirely, e.g., create an end to match closest if

```
Stmt ::= ... |  
   if Expr then Stmt end |  
   if Expr then Stmt else Stmt end
```

- formal, clear, elegant
- allows sequence of Stmts in then and else branches, no {,} needed
- extra end required for every if
  (But maybe this is a good idea anyway? These ambiguities can lead to programmer bugs ...)

---

Parser Tools and Operators

- Most parser tools can cope with ambiguous grammars
  - Makes life simpler if you’re careful
- Typically one can specify operator precedence & associativity
  - Allows simpler, ambiguous grammar with fewer nonterminals as basis for generated parser, without creating problems
Parser Tools and Ambiguous Grammars

• Possible rules for resolving other problems
  – Earlier productions in the grammar preferred to later ones
  – Longest match used if there is a choice
• Parser tools normally allow for this
  – But be sure that what the tool does is really what you want

Agenda

• Finish discussing the “if-else” ambiguity
• Start our first parsing algorithm: LR Parsing
Parsing Algorithms

• The two primary style of parsing are LL and LR parsing
• LL Parsing (Left-to-right scan, Leftmost derivation)
  – Top down – start with grammar start symbol, work your way down until you get to terminals.
  – Generates a leftmost derivation (the leftmost derivation assuming unambiguous grammar)
  – The “traditional” starting point for teaching parsing.
• We’ll start with LR since you need it for your projects (and it’s the most commonly used).

LR(1) Parsing

• We’ll focus specifically on LR(1) parsers
  – Left to right scan, Rightmost derivation (reverse rightmost), 1 symbol lookahead
    • Lookahead: how far past current symbol we can look to determine which rule to apply.
  – Almost all practical programming languages have an LR(1) grammar
  – LALR(1), SLR(1), etc. – subsets of LR(1) with lower memory requirements, slightly less power
    • LALR(1) can mostly parse most real languages, and is used by YACC/Bison/CUP/etc.
Bottom-Up Parsing

• Basic Idea: Read tokens left to right, push (shift) onto a stack.
• Whenever the top of the stack matches 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.
• Process called shift-reduce parsing.

Winter 2013
UW CSE 401 (Michael Ringenburg)
Example: Parse a b b c d e (bottom up)

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

Details

• The bottom-up parser reconstructs a reverse rightmost derivation.
• Given the rightmost derivation
  \[
  S =>_{rm} \beta_1 =>_{rm} \beta_2 =>_{rm} \ldots =>_{rm} \beta_{n-2} =>_{rm} \beta_{n-1} =>_{rm} \beta_n = w
  \]
  the parser will first discover \( \beta_{n-1} =>_{rm} \beta_n \), then \( \beta_{n-2} =>_{rm} \beta_{n-1} \), etc.
• Parsing terminates when
  – \( \beta_1 \) reduced to \( S \) (start symbol, success), or
  – No match can be found (syntax error)
How Does this Work?

• 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 (shift another symbol onto the stack)
• Can reduce $A \Rightarrow \beta$ if both of these hold:
  – $A \Rightarrow \beta$ is a valid production
  – $A \Rightarrow \beta$ is a step in the rightmost derivation (e.g., don’t use the $A \Rightarrow b$ reduction for the second ‘b’ in our example).
• That’s why we call it a \textit{shift-reduce parser}

Difficulties

• Tricky parts:
  – How do we do this efficiently?
    • Prefer $O(\text{sourceLength} + \text{derivationLength})$. Can’t really do better than $O(\text{input} + \text{output})$!
    • Naïve approach (examine full stack at every step) is $O((\text{sourceLength} + \text{derivationLength}) \times \text{sourceLength})$, since stack is potentially as long as program
  – How do we know whether $A \Rightarrow \beta$ is a step in the rightmost derivation (second condition for reducing)?
• Preview: Generate DFAs encoded by tables ...

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):
  - I.e., $\alpha$ is a right-sentential form of the grammar if $S \Rightarrow_{rm}^* \alpha$.

Handles

- A substring of the tree frontier (the highest level that we’ve built) that matches the right side of a production, and is used in the rightmost derivation of the current string.
  - 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 current derivation.
  - $\beta$ may appear in other places in the frontier without being a handle for $A::=\beta$.
- Bottom-up parsing is all about finding these handles.
Handles (cont.)

• Formally, a handle of a right-sentential form $\gamma_i$ is a production $A ::= \beta$ and a position in $\gamma_i$ where $\beta$ may be replaced by $A$ to produce the previous right-sentential form $\gamma_{i-1}$ in the rightmost derivation of the current string that is being parsed.

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.
  • $A ::= b$ at position 3 is not a handle.
• (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
  - Something that encodes the rules that tell us what action to take given the state of the stack and lookahead
    - This is typically a table that encodes a finite automata

Shift-Reduce Parser Actions

- What are these actions that we may take?
  - 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>
<tr>
<td>$a</td>
<td>bcde$</td>
<td>shift</td>
</tr>
<tr>
<td>$ab</td>
<td>bcde$</td>
<td>reduce A=&gt;b</td>
</tr>
<tr>
<td>$aA</td>
<td>bcde$</td>
<td>shift</td>
</tr>
<tr>
<td>$aAb</td>
<td>cde$</td>
<td>shift</td>
</tr>
<tr>
<td>$aAbc</td>
<td>de$</td>
<td>reduce A=&gt;Abc</td>
</tr>
<tr>
<td>$aA</td>
<td>de$</td>
<td>shift</td>
</tr>
<tr>
<td>$aAd</td>
<td>e$</td>
<td>reduce B=&gt;d</td>
</tr>
<tr>
<td>$aAB</td>
<td>e$</td>
<td>shift</td>
</tr>
<tr>
<td>$aABe</td>
<td>$</td>
<td>reduce S=&gt;aABe</td>
</tr>
<tr>
<td>$S</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Winter 2013  UW CSE 401 (Michael Ringenburg)

How Do We Decide which action to take?

• 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
  – 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
  – Recall, any regular language is recognizable by a DFA
  – Perform reductions when we recognize them

Winter 2013  UW CSE 401 (Michael Ringenburg)
Viable Prefixes for our Example Grammar

<table>
<thead>
<tr>
<th>Viable Prefix</th>
<th>Handle/Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S ::= aABE</td>
<td><strong>Accept</strong></td>
</tr>
<tr>
<td>aABe</td>
<td>S ::= aABe</td>
</tr>
<tr>
<td>aAd</td>
<td>B ::= d</td>
</tr>
<tr>
<td>aAbc</td>
<td>A ::= Abc</td>
</tr>
<tr>
<td>Ab</td>
<td>A ::= b</td>
</tr>
</tbody>
</table>

Plus prefixes of above...

- The listed prefixes are those that extend all the way to the end of a handle – these correspond to reduction actions. Their prefixes are also viable prefixes.
- Why not aAbcbc? Extends past the handle (Abc).

DFA for viable prefixes of our example grammar
Trace

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

Stack | Input
--- | ---
$ | abbcde$

Observations

- Way too much backtracking (start down a path, end up having to shift and restart)
  - 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 – in this simple case we were able to intuitively see all of the viable prefixes. But how do we find them in general?
  - We’ll defer construction details for now