Ambiguity, Precedence, Associativity & Top-Down Parsing

Lecture 9-10
(From slides by G. Necula & R. Bodik)

9/18/06  Prof. Hilfinger CS164 Lecture 9

---

**Administrivia**

- Please let me know if there are continued problems with being able to see other people’s stuff.
- Preliminary run of test data against any projects handed in by midnight Wednesday.
  - Not final data sets, but may give you an indication.
  - You can submit early and often.
  - Will not test again until midnight Friday.

---

**Remaining Issues**

- How do we find a derivation of $s$?
- Ambiguity: what if there is more than one parse tree (interpretation) for some string $s$?
- Errors: what if there is no parse tree for some string $s$?
- Given a derivation, how do we construct an abstract syntax tree from it?

Today, we’ll look at the first two.

---

**Ambiguity**

- **Grammar**

\[ E \rightarrow E + E \mid E * E \mid (E) \mid \text{int} \]

- **Strings**

\[ \text{int} + \text{int} + \text{int} \]
\[ \text{int} * \text{int} + \text{int} \]

---

**Ambiguity. Example**

The string $\text{int} + \text{int} + \text{int}$ has two parse trees

---

**Ambiguity. Example**

The string $\text{int} * \text{int} + \text{int}$ has two parse trees

---

* has higher precedence than +
Ambiguity (Cont.)

- A grammar is ambiguous if it has more than one parse tree for some string
  - Equivalently, there is more than one rightmost or leftmost derivation for some string
- Ambiguity is bad
  - Leaves meaning of some programs ill-defined
- Ambiguity is common in programming languages
  - Arithmetic expressions
  - IF-THEN-ELSE

Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite the grammar unambiguously
  $E \rightarrow E + T | T$
  $T \rightarrow T * \text{int} | \text{int} | (E)$
- Enforces precedence of $*$ over $+$
- Enforces left-associativity of $+$ and $*$

Ambiguity. Example

The $\text{int} * \text{int} + \text{int}$ has only one parse tree now

\begin{align*}
E & \rightarrow T + T \\
T & \rightarrow T * \text{int} | \text{int} | (E)
\end{align*}

Ambiguity: The Dangling Else

- Consider the grammar
  $E \rightarrow \text{if } E \text{ then } E$
  | $\text{if } E \text{ then } E \text{ else } E$
  | OTHER
- This grammar is also ambiguous

The Dangling Else: Example

- The expression
  $\text{if } E_1 \text{ then } (\text{if } E_2 \text{ then } E_3 \text{ else } E_4)$
  has two parse trees
- Typically we want the second form

The Dangling Else: A Fix

- else matches the closest unmatched then
- We can describe this in the grammar (distinguish between matched and unmatched "then")
  $E \rightarrow \text{MIF}$
  | $\text{UIF}$
  | $\text{OTHER}$
  $\text{MIF} \rightarrow \text{if } E \text{ then } \text{MIF} \text{ else } \text{MIF}$
  | $\text{OTHER}$
  $\text{UIF} \rightarrow \text{if } E \text{ then } \text{MIF} \text{ else } \text{UIF}$
- Describes the same set of strings
The Dangling Else: Example Revisited

- The expression
  \[ \text{if } E_1 \text{ then if } E_2 \text{ then } E_3 \text{ else } E_4 \]
- Not valid because the then expression is not a MIF

A valid parse tree (for a UIF)

Ambiguity

- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - But we need disambiguation mechanisms
- Instead of rewriting the grammar
  - Use the more natural (ambiguous) grammar
  - Along with disambiguating declarations
- Most tools allow precedence and associativity declarations to disambiguate grammars
- Examples …

Associativity Declarations

- Consider the grammar
  \[ E \rightarrow E + E \mid \text{int} \]
- Ambiguous: two parse trees of \( \text{int} + \text{int} + \text{int} \)
- Left-associativity declaration: \( \%\text{left} '+' \)

Precedence Declarations

- Consider the grammar
  \[ E \rightarrow E + E \mid E \ast E \mid \text{int} \]
- And the string \( \text{int} + \text{int} \ast \text{int} \)
- Precedence declarations: \( \%\text{left} '-' \)
  \( \%\text{left} '*' \)

How It’s Done I: Intro to Top-Down Parsing

- Terminals are seen in order of appearance in the token stream:
  \[ t_1 \ t_2 \ t_3 \ t_4 \ t_5 \]
- The parse tree is constructed
  - From the top
  - From left to right
- As for leftmost derivation

Top-down Depth-First Parsing

- Consider the grammar
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow ( E ) \mid \text{int} \mid \text{int} \ast T \]
- Token stream is: \( \text{int} \ast \text{int} \)
- Start with top-level non-terminal \( E \)
- Try the rules for \( E \) in order
Depth-First Parsing. Example: int * int

- Start with start symbol: E
- Try E → T + E: T + E
- Then try a rule for T → (E): (E) + E
  - But (≠ input int; backtrack to T + E
- Try T → int: Token matches: int + E
  - But + ≠ input *; backtrack to T + E
- Try T → int * T: int * T + E
  - But (skipping some steps) can’t be matched
- Must backtrack to E

Depth-First Parsing

- Parsing: given a string of tokens t_1, t_2, ..., t_n, find a leftmost derivation (and thus, parse tree)
- Depth-first parsing: Beginning with start symbol, try each production exhaustively on leftmost non-terminal in current sentential form and recurse.

When Depth-First Doesn’t Work Well

- Consider productions S → S a | α:
  - In the process of parsing S we try the above rules
  - Applied consistently in this order, get infinite loop
  - Could re-order productions, but search will have lots of backtracking and general rule for ordering is complex
- Problem here is left-recursive grammar: one that has a non-terminal S
  S → Sα for some α

Elimination of Left Recursion

- Consider the left-recursive grammar
  S → S α | β
- S generates all strings starting with a β and followed by a number of α
- Can rewrite using right-recursion
  S → β S'
  S' → α S' | ε
Elimination of left Recursion. Example

- Consider the grammar
  \[ S \rightarrow 1 \mid S \ 0 \quad (\beta = 1 \text{ and } \alpha = 0) \]

  can be rewritten as
  \[ S \rightarrow 1 S' \\
  S' \rightarrow 0 S' \mid \epsilon \]

More Elimination of Left Recursion

- In general
  \[ S \rightarrow S \ \alpha_1 \mid \ldots \mid S \ \alpha_n \mid \beta \mid \ldots \mid \beta_m \]
  All strings derived from \( S \) start with one of \( \beta \)
  and continue with several instances of \( \alpha_1 \ldots \alpha_n \)
  (1)

  Restart as
  \[ S \rightarrow \beta_1 S' \mid \ldots \mid \beta_m S' \\
  S' \rightarrow \alpha_1 S' \mid \ldots \mid \alpha_n S' \mid \epsilon \]

General Left Recursion

- The grammar
  \[ S \rightarrow A \ \alpha \mid \delta \quad (1) \\
  A \rightarrow S \ \beta \quad (2) \]
  is also left-recursive because
  \[ S \rightarrow S \ \beta \ \alpha \]
  This left recursion can also be eliminated by
  first substituting (2) into (1)
  (1)

  There is a general algorithm (e.g. Aho, Sethi, Ullman §4.3)

  But personally, I'd just do this by hand.

An Alternative Approach

- Instead of reordering or rewriting grammar,
  can use top-down breadth-first search.

  \[ S \rightarrow S \ a \mid a \]
  String: aaa

  \[ S \]
  \[ S \ a \] (string not all matched)
  \[ S \ a a \ a a \]
  \[ S \ a a a \ a a a \]

Summary of Top-Down Parsing So Far

- Simple and general parsing strategy
  - Left recursion must be eliminated first
  - But that can be done automatically
  - Or can use breadth-first search
  - But backtracking (depth-first) or maintaining
    list of possible sentential forms (breadth-
    first) can make it slow
  - Often, though, we can avoid both ...

Predictive Parsers

- Modification of depth-first parsing in which
  parser "predicts" which production to use
  - By looking at the next few tokens
  - No backtracking

- Predictive parsers accept LL(k) grammars
  - L means "left-to-right" scan of input
  - L means "leftmost derivation"
  - k means "predict based on k tokens of lookahead"

- In practice, LL(1) is used
LL(1) Languages

- Previously, for each non-terminal and input token there may be a choice of production
- LL(k) means that for each non-terminal and k tokens, there is only one production that could lead to success

Recursive Descent: Grammar as Program

- In recursive descent, we think of a grammar as a program.
- Each non-terminal is turned into a procedure
- Each right-hand side transliterated into part of the procedure body for its non-terminal
- First, define
  - `next()` current token of input
  - `scan(t)` check that `next() = t` (else ERROR), and then read new token.

Recursive Descent: Example

```
def P():
    S(); scan($)
def S():
    T(); S'
    elif next() in [')', '$]: pass
    else: ERROR
    def T():
        if next() == int: scan(int)
        elif next() == '(': scan('('); S(); scan(')')
        else: ERROR
```

But where do tests come from?

Predicting Right-hand Sides

- The if-tests are conditions by which parser predicts which right-hand side to use.
- In our example, used only next symbol (LL(1)); but could use more.
- Can be specified as a 2D table
  - One dimension for current non-terminal to expand
  - One dimension for next token
  - A table entry contains one production

But First: Left Factoring

- With the grammar
  
  \[
  \begin{align*}
  E &\rightarrow T \cdot E \mid E \\
  T &\rightarrow \text{int} \mid \text{int} \cdot T \mid (E)
  \end{align*}
  \]

- Impossible to predict because
  - For T two productions start with \text{int}
  - For E it is not clear how to predict

- A grammar must be left-factored before use for predictive parsing

Left-Factoring Example

- Starting with the grammar
  
  \[
  \begin{align*}
  E &\rightarrow T \cdot E \mid E \\
  T &\rightarrow \text{int} \mid \text{int} \cdot T \mid (E)
  \end{align*}
  \]

- Factor out common prefixes of productions
  
  \[
  \begin{align*}
  E &\rightarrow TX \\
  X &\rightarrow E \mid T \\
  T &\rightarrow (E) \mid \text{int} \mid \text{int} \cdot T \mid \text{int} \cdot Y \\
  Y &\rightarrow \text{int} \mid T \\
  \end{align*}
  \]

But with the grammar

\[
\begin{align*}
E &\rightarrow T \cdot E \mid E \\
T &\rightarrow \text{int} \mid \text{int} \cdot T \mid (E)
\end{align*}
\]
LL(1) Parsing Table Example

- Left-factored grammar
  \[
  E \rightarrow TX \\
  T \rightarrow (E) | \text{int } Y \\
  Y \rightarrow * T | \epsilon
  \]

- The LL(1) parsing table ($\epsilon$ is a special end marker):

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>int</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>T X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>+ E</td>
<td>X</td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>* T</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LL(1) Parsing Table Example (Cont.)

- Consider the \([E, \text{int}]\) entry
  - "When current non-terminal is \(E\) and next input is \(\text{int}\), use production \(E \rightarrow TX\)."
  - This production can generate an \(\text{int}\) in the first place

- Consider the \([Y,+]\) entry
  - "When current non-terminal is \(Y\) and current token is \(+\), get rid of \(Y\)."
  - We’ll see later why this is so

LL(1) Parsing Tables. Errors

- Blank entries indicate error situations
  - Consider the \([E,*]\) entry
  - "There is no way to derive a string starting with \(*\) from non-terminal \(E\)"

Using Parsing Tables

- Method similar to recursive descent, except
  - For first non-terminal \(S\)
  - We look at the next token \(a\)
    - And choose the production shown at \([S,a]\)
  - We use a stack to keep track of pending non-terminals
  - We reject when we encounter an error state
  - We accept when we encounter end-of-input

LL(1) Parsing Algorithm

initialize stack = \(<S,$>\)
repeat
  case stack of
    \(<X, \text{rest}>\) : if \(T[X, \text{next()}] = Y_1...Y_n:\)
      stack \(<Y_1...Y_n, \text{rest}>;\)
      else: error();
    \(<t, \text{rest}>\) : scan \(t\); stack \(<\text{rest}>;\)
  until stack == <>

LL(1) Parsing Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(E $)</td>
<td>int * int $</td>
<td>TX</td>
</tr>
<tr>
<td>(TX $)</td>
<td>int * int $</td>
<td>int Y</td>
</tr>
<tr>
<td>(int Y \times $)</td>
<td>int * int $</td>
<td>terminal</td>
</tr>
<tr>
<td>(Y \times $)</td>
<td>* int $</td>
<td>* T</td>
</tr>
<tr>
<td>(* TX $)</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>(TX $)</td>
<td>int $</td>
<td>int Y</td>
</tr>
<tr>
<td>(int Y \times $)</td>
<td>int $</td>
<td>terminal</td>
</tr>
<tr>
<td>(Y \times $)</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>(X $)</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>($)</td>
<td>$</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>
Constructing Parsing Tables

• LL(1) languages are those definable by a parsing table for the LL(1) algorithm
• No table entry can be multiply defined
• Once we have the table
  - Can create table-driven or recursive-descent parser
  - The parsing algorithms are simple and fast
  - No backtracking is necessary
• We want to generate parsing tables from CFG

Top-Down Parsing. Review

• Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal
  - The leaves at any point form a string $\beta A\gamma$
    - $\beta$ contains only terminals
    - The input string is $\beta b\delta$
    - The prefix $\beta$ matches
    - The next token is $b$

Constructing Predictive Parsing Tables

• Consider the state $S \rightarrow \beta A\gamma$
  - With $b$ the next token
  - Trying to match $\beta b\delta$

There are two possibilities:
1. $b$ belongs to an expansion of $A$

  • Any $A \rightarrow a$ can be used if $b$ can start a string derived from $a$
    - In this case we say that $b \in \text{First}(a)$

Or…
Computing First Sets

**Definition** \( \text{First}(X) = \{ b \mid b \in X \text{ or } X \Rightarrow b \} \cup \{ \varepsilon \mid X \Rightarrow \varepsilon \} \) for any grammar symbol.

1. **First(b)** = \{ b \}

2. For all productions \( X \Rightarrow A_1 \ldots A_n \)
   - Add \( \text{First}(A_1) \cup \{ \varepsilon \} \) to \( \text{First}(X) \) if \( b \notin \text{First}(A_1) \)
   - Add \( \text{First}(A_2) \cup \{ \varepsilon \} \) to \( \text{First}(X) \) if \( b \notin \text{First}(A_2) \)
   - ...
   - Add \( \varepsilon \) to \( \text{First}(X) \)

**First Sets, Example**

For the grammar

- \( E \Rightarrow T \ X \)
- \( T \Rightarrow ( E ) \mid int \ Y \)
- \( X \Rightarrow + E \mid \varepsilon \)
- \( Y \Rightarrow ^* T \mid \varepsilon \)

**First sets**

- \( \text{First}(T) = \{ \text{int}, \} \)
- \( \text{First}(E) = \{ \text{int}, \} \)
- \( \text{First}(\text{int}) = \{ \text{int} \} \)
- \( \text{First}(\varepsilon) = \{ \varepsilon \} \)
- \( \text{First}(\text{int}) = \{ \varepsilon \} \)

**Computing First Sets, Contd.**

- That takes care of single-symbol case.
- In general:
  \[
  \text{FIRST}(X_1 X_2 X_3) = \\
  \text{FIRST}(X_1) \cup \text{FIRST}(X_2) \quad \text{if } \varepsilon \in \text{FIRST}(X_3) \\
  \quad \cup \ldots \\
  \quad \cup \text{FIRST}(X_2) \quad \text{if } \varepsilon \in \text{FIRST}(X_2 X_2 X_3) \\
  \quad \quad \quad \quad \cup \{ \varepsilon \} \quad \text{unless } \varepsilon \in \text{FIRST}(X_2) \quad \forall \ i
  \]

Computing Follow Sets

**Definition** \( \text{Follow}(X) = \{ b \mid S \Rightarrow \ast b X X \} \)

1. Compute the First sets for all non-terminals first
2. Add \( \varepsilon \) to \( \text{Follow}(S) \) if \( S \) is the start non-terminal
3. For all productions \( Y \Rightarrow \ldots X A_1 \ldots A_n \)
   - Add \( \text{First}(A_1) \cup \{ \varepsilon \} \) to \( \text{Follow}(X) \) if \( b \notin \text{First}(A_1) \)
   - Add \( \text{First}(A_2) \cup \{ \varepsilon \} \) to \( \text{Follow}(X) \) if \( b \notin \text{First}(A_2) \)
   - ...
   - Add \( \text{First}(A_n) \cup \{ \varepsilon \} \) to \( \text{Follow}(X) \) if \( b \notin \text{First}(A_n) \)
   - Add \( \text{Follow}(Y) \) to \( \text{Follow}(X) \)
Follow Sets. Example

- For the grammar
  \[ E \rightarrow TX \]
  \[ X \rightarrow *E | \varepsilon \]
  \[ T \rightarrow (E) | \text{int} Y \]
  \[ Y \rightarrow *T | \varepsilon \]
- Follow sets
  \[ \text{Follow}(E) = \{ (, \}) \]  
  \[ \text{Follow}(X) = \{ (, \} \]  
  \[ \text{Follow}(Y) = \{ *, \} \]  
  \[ \text{Follow}(T) = \{ *, \} \]

Constructing LL(1) Parsing Tables

- Construct a parsing table \( T \) for CFG \( G \)
- For each production \( A \rightarrow \alpha \) in \( G \) do:
  - For each terminal \( b \in \text{First}(\alpha) \) do
    - \( T[A, b] = \alpha \)
  - If \( \alpha \rightarrow \varepsilon \), for each \( b \in \text{Follow}(A) \) do
    - \( T[A, b] = \alpha \)

Constructing LL(1) Tables. Example

- For the grammar
  \[ E \rightarrow TX \]
  \[ X \rightarrow *E | \varepsilon \]
  \[ T \rightarrow (E) | \text{int} Y \]
  \[ Y \rightarrow *T | \varepsilon \]
- Where in the line of \( Y \) do we put \( Y \rightarrow *T \)?
  - In the lines of \( \text{First}(\ast T) = \{ \ast \} \)
- Where in the line of \( Y \) do we put \( Y \rightarrow \varepsilon \)?
  - In the lines of \( \text{Follow}(Y) = \{ (, *, ) \} \)

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then \( G \) is not LL(1)
  - If \( G \) is ambiguous
  - If \( G \) is left recursive
  - If \( G \) is not left-factored
  - And in other cases as well
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables

Recursive Descent for Real

- So far, have presented a purist view.
- In fact, use of recursive descent makes life simpler in many ways if we "cheat" a bit.
- Here's how you really handle left recursion in recursive descent, for \( S \rightarrow S A | R \):
  ```python
  def S():
    R()
    while next() \in \text{FIRST}(A):
      A()
  ```
- It's a program: all kinds of shortcuts possible.

Review

- For some grammars there is a simple parsing strategy
  - Predictive parsing (LL(1))
  - Once you build the LL(1) table, you can write the parser by hand
- Next: a more powerful parsing strategy for grammars that are not LL(1)