Ambiguity, Precedence, Associativity & Top-Down Parsing

Lecture 9-10
(From slides by G. Necula & R. Bodik)
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 \texttt{int + int + int} has two parse trees

\begin{itemize}
\item \texttt{E}
\item \texttt{E + E}
\item \texttt{E + E int}
\end{itemize}

\begin{itemize}
\item \texttt{E}
\item \texttt{E + E}
\item \texttt{int E + E}
\end{itemize}

\textbf{+ is left-associative}
Ambiguity. Example

The string `int * int + 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**

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

- Enforces precedence of \( \ast \) over \(+\)
- Enforces left-associativity of \(+\) and \(\ast\)
Ambiguity. Example

The $\texttt{int} \ast \texttt{int} + \texttt{int}$ has only one parse tree now
Ambiguity: The Dangling Else

• Consider the grammar

\[
E \rightarrow \text{if } E \text{ then } E \\
\quad | \text{if } E \text{ then } E \text{ else } E \\
\quad | \text{OTHER}
\]

• This grammar is also ambiguous
The Dangling Else: Example

- The expression
  
  \[
  \text{if } E_1 \text{ then if } E_2 \text{ then } E_3 \text{ else } E_4
  \]

  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”)

\[
E \rightarrow \text{MIF} \quad /* \text{all then are matched} */ \\
| \text{UIF} \quad /* \text{some then are unmatched} */ \\
MIF \rightarrow \text{if E then MIF else MIF} \\
| \text{OTHER} \\
UIF \rightarrow \text{if E then E} \\
| \text{if E then MIF else UIF}
\]

- Describes the same set of strings
The Dangling Else: Example Revisited

• The expression \( \text{if } E_1 \text{ then if } E_2 \text{ then } E_3 \text{ else } E_4 \)

- A valid parse tree (for a UIF)

• Not valid because the then expression is not a MIF
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 int + int + int

• Left-associativity declaration: \%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:**

$\%\text{left} \ ' + '$

$\%\text{left} \ ' \ast '$
How It’s Done I: Intro to Top-Down Parsing

• Terminals are seen in order of appearance in the token stream:
  \[ t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \]

• The parse tree is constructed
  - From the top
  - From left to right

• ... As for leftmost derivation
Top-down Depth-First Parsing

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

- Start with start symbol \texttt{E}
- Try \texttt{E \rightarrow T + E} \texttt{T + E}
- Then try a rule for \texttt{T \rightarrow ( E)} \texttt{(E) + E}
  - But \texttt{(} \neq \text{input \texttt{int}}; \text{backtrack to} \texttt{T + E}
- Try \texttt{T \rightarrow int . Token matches.} \texttt{int + E}
  - But \texttt{+} \neq \text{input \texttt{*}}; \text{back to} \texttt{T + E}
- Try \texttt{T \rightarrow int * T} \texttt{int*T+E}
  - But (skipping some steps) \texttt{+} can’t be matched
- Must backtrack to \texttt{E}
Depth-First Parsing. Example $\text{int} \ast \text{int}$

- Try $E \rightarrow T$
- Follow same steps as before for $T$
  - And succeed with $T \rightarrow \text{int} \ast T$ and $T \rightarrow \text{int}$
  - With the following parse tree

```
  E
  /|
 /  |
T    
     /
int   *
     /|
     /  
    T   int
```
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.
Depth-First Parsing of $t_1 t_2 \ldots t_n$

- At a given moment, have sentential form that looks like this: $t_1 t_2 \ldots t_k A \ldots$, $0 \leq k \leq n$
- Initially, $k=0$ and $A\ldots$ is just start symbol
- Try a production for $A$: if $A \rightarrow BC$ is a production, the new form is $t_1 t_2 \ldots t_k B C \ldots$
- Backtrack when leading terminals aren’t prefix of $t_1 t_2 \ldots t_n$ and try another production
- Stop when no more non-terminals and terminals all matched (accept) or no more productions left (reject)
When Depth-First Doesn’t Work Well

- Consider productions $S \rightarrow S \alpha \mid \alpha$:
  - 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 $S$
  $$S \rightarrow^+ S\alpha \text{ for some } \alpha$$
Elimination of Left Recursion

• Consider the left-recursive grammar
  \[ S \rightarrow S \alpha \mid \beta \]

• \( S \) generates all strings starting with a \( \beta \) and followed by a number of \( \alpha \)

• Can rewrite using right-recursion
  \[ S \rightarrow \beta S' \]
  \[ S' \rightarrow \alpha S' \mid \epsilon \]
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
  \[ S \rightarrow 1 \ S' \]
  \[ S' \rightarrow 0 \ S' \mid \varepsilon \]
More Elimination of Left Recursion

- In general
  \[ S \rightarrow S \alpha_1 | ... | S \alpha_n | \beta_1 | ... | \beta_m \]

- All strings derived from \( S \) start with one of \( \beta_1, \ldots, \beta_m \) and continue with several instances of \( \alpha_1, \ldots, \alpha_n \)

- Rewrite as
  \[ S \rightarrow \beta_1 S' | ... | \beta_m S' \]
  \[ S' \rightarrow \alpha_1 S' | ... | \alpha_n S' | \epsilon \]
General Left Recursion

• The grammar
  
  \[ S \rightarrow A \alpha | \delta \]  
  \[ A \rightarrow S \beta \]  

  is also left-recursive because
  
  \[ S \rightarrow S \beta \alpha \]  

• This left recursion can also be eliminated by first substituting (2) into (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*.

\[
S \rightarrow S \ a \ | \ a
\]

String: aaa

\[
\begin{align*}
S \\
S \ a \ & \ x \\
S \ a \ a \ & \ a \ a \\
S \ a \ a \ a \ & \ a \ a \ a \\
\end{align*}
\]

(string not all matched)
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

\[
\begin{align*}
P & \rightarrow S \, $ \\
S & \rightarrow T \, S' \\
S' & \rightarrow + \, S \mid \varepsilon \\
T & \rightarrow \text{int} \mid ( \, S \, ) \\
\end{align*}
\]

($ = \text{end marker}$)

```python
def P():  
    S(); scan($)$
def S():  
    T(); S''()
def S'():  
    if next() == '+': scan('+'); 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
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

- 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

\[ E \rightarrow T + E | T \]
\[ T \rightarrow \text{int} | \text{int} \ast T | ( E ) \]

• Factor out common prefixes of productions

\[ E \rightarrow TX \]
\[ X \rightarrow + E | \varepsilon \]
\[ T \rightarrow ( E ) | \text{int} Y \]
\[ Y \rightarrow * T | \varepsilon \]
LL(1) Parsing Table Example

- **Left-factored grammar**
  \[ E \rightarrow T X \]
  \[ T \rightarrow ( E ) | \text{int} \ Y \]
  \[ X \rightarrow + E | \varepsilon \]
  \[ Y \rightarrow * T | \varepsilon \]

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

<table>
<thead>
<tr>
<th></th>
<th>int</th>
<th>*</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>(E)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>TX</td>
<td></td>
<td></td>
<td>TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>+ E</td>
<td></td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
</tbody>
</table>
| Y |      | * T|    | \varepsilon | \varepsilon | \varepsilon |\$
LL(1) Parsing Table Example (Cont.)

- Consider the [E, int] entry
  - “When current non-terminal is E and next input is int, use production \( E \rightarrow T X \)
  - This production can generate an 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 \([E,\ast]\) entry
  - “There is no way to derive a string starting with \(\ast\) 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, rest> : if T[X,next()] == Y₁...Yₙ:
      stack ← <Y₁... Yₙ rest>;
    else: error ();
    <t, rest> : scan (t); stack ← <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>T X</td>
</tr>
<tr>
<td>T X $</td>
<td>int * int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int * int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>* int $</td>
<td>* T</td>
</tr>
<tr>
<td>* T X $</td>
<td>* int $</td>
<td>terminal</td>
</tr>
<tr>
<td>T X $</td>
<td>int $</td>
<td>int Y</td>
</tr>
<tr>
<td>int Y X $</td>
<td>int $</td>
<td>terminal</td>
</tr>
<tr>
<td>Y X $</td>
<td>$</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-driver 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

```
int * int + int
```

```
E
 /|
/ |
 /|
T + E
```

`int * int + int`
Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

```
  E
 / 
/    
T + E
 /  
/   
* T
 /  
/   
int
```

- 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 $b$
Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

E
  ├── T
  │   ├── int
  │   └── T
  │       └── int
  └── E
      └── T

- 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 $b$
Top-Down Parsing. Review

- Top-down parsing expands a parse tree from the start symbol to the leaves
  - Always expand the leftmost non-terminal

```
int * int + int
```

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

Or...
Constructing Predictive Parsing Tables (Cont.)

2. \( b \) does not belong to an expansion of \( A \)
   - The expansion of \( A \) is empty and \( b \) belongs to an expansion of \( \gamma \) (e.g., \( b\omega \))
   - Means that \( b \) can appear after \( A \) in a derivation of the form \( S \rightarrow^{*} \beta A b\omega \)
   - We say that \( b \in \text{Follow}(A) \) in this case

- What productions can we use in this case?
  - Any \( A \rightarrow \alpha \) can be used if \( \alpha \) can expand to \( \epsilon \)
  - We say that \( \epsilon \in \text{First}(A) \) in this case
Summary of Definitions

- For $b \in T$, the set of terminals; $\alpha$ a sequence of terminal & non-terminal symbols, $S$ the start symbol, $A \in N$, the set of non-terminals:
- $\text{FIRST}(\alpha) \subseteq T \cup \{ \varepsilon \}$
  
  $b \in \text{FIRST}(\alpha)$ iff $\alpha \rightarrow^* b \ldots$
  
  $\varepsilon \in \text{FIRST}(\alpha)$ iff $\alpha \rightarrow^* \varepsilon$

- $\text{FOLLOW}(A) \subseteq T$
  
  $b \in \text{FOLLOW}(A)$ iff $S \rightarrow^* \ldots A b \ldots$
Computing First Sets

Definition \( \text{First}(X) = \{ b \mid X \rightarrow^* b\alpha \} \cup \{ \epsilon \mid X \rightarrow^* \epsilon \}, \)
\( X \) any grammar symbol.

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

2. For all productions \( X \rightarrow A_1 \ldots A_n \)
   - Add \( \text{First}(A_1) - \{ \epsilon \} \) to \( \text{First}(X) \). Stop if \( \epsilon \not\in \text{First}(A_1) \)
   - Add \( \text{First}(A_2) - \{ \epsilon \} \) to \( \text{First}(X) \). Stop if \( \epsilon \not\in \text{First}(A_2) \)
   - ... 
   - Add \( \text{First}(A_n) - \{ \epsilon \} \) to \( \text{First}(X) \). Stop if \( \epsilon \not\in \text{First}(A_n) \)
   - Add \( \epsilon \) to \( \text{First}(X) \)
Computing First Sets, Contd.

• That takes care of single-symbol case.
• In general:

\[
\text{FIRST}(X_1 X_2 \ldots X_k) = \\
\text{FIRST}(X_1) \\
\cup \text{FIRST}(X_2) \text{ if } \varepsilon \in \text{FIRST}(X_1) \\
\cup \ldots \\
\cup \text{FIRST}(X_2) \text{ if } \varepsilon \in \text{FIRST}(X_1 X_2 \ldots X_{k-1}) \\
- \{ \varepsilon \} \text{ unless } \varepsilon \in \text{FIRST}(X_i) \quad \forall \ i
\]
First Sets. Example

• For the grammar

\[ E \rightarrow T X \]
\[ T \rightarrow ( E ) \mid \text{int Y} \]
\[ X \rightarrow + E \mid \epsilon \]
\[ Y \rightarrow * T \mid \epsilon \]

• First sets

First( ( ) ) = \{ ( ) \}
First( ( ) ) = \{ ( ) \}
First( \text{int} ) = \{ \text{int} \}
First( + ) = \{ + \}
First( * ) = \{ * \}
First( T ) = \{ \text{int}, ( ) \}
First( E ) = \{ \text{int}, ( ) \}
First( X ) = \{ +, \epsilon \}
First( Y ) = \{ *, \epsilon \}
Computing Follow Sets

Definition \( \text{Follow}(X) = \{ b \mid S \rightarrow^* \beta X b \omega \} \)

1. Compute the First sets for all non-terminals first
2. Add $ to Follow(S) (if S is the start non-terminal)

3. For all productions \( Y \rightarrow \ldots X A_1 \ldots A_n \)
   - Add First\( (A_1) - \{\varepsilon\} \) to Follow\( (X) \). Stop if \( \varepsilon \notin \text{First}(A_1) \)
   - Add First\( (A_2) - \{\varepsilon\} \) to Follow\( (X) \). Stop if \( \varepsilon \notin \text{First}(A_2) \)
   - ...
   - Add First\( (A_n) - \{\varepsilon\} \) to Follow\( (X) \). Stop if \( \varepsilon \notin \text{First}(A_n) \)
   - Add Follow\( (Y) \) to Follow\( (X) \)
Follow Sets. Example

• For the grammar
  \[ E \rightarrow T \ X \]
  \[ T \rightarrow ( \ E ) \mid \text{int } Y \]
  \[ X \rightarrow + \ E \mid \varepsilon \]
  \[ Y \rightarrow * \ T \mid \varepsilon \]

• 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

\[
\begin{align*}
  E & \rightarrow T \ X \\
  T & \rightarrow ( \ E \ ) \mid \ \text{int} \ Y \\
  X & \rightarrow + \ E \mid \varepsilon \\
  Y & \rightarrow * \ T \mid \varepsilon
\end{align*}
\]

• Where in the line of \( Y \) do we put \( Y \rightarrow \ast \ T \)?
  - 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 $S \rightarrow S \ A \mid R$:
  
  ```python
  def S():
    R ()
    while next() ∈ 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)