Top-Down Parsing

ICOM 4036
Lecture 6
Review

• A parser consumes a sequence of tokens $s$ and produces a parse tree

• Issues:
  - How do we recognize that $s \in L(G)$?
  - A parse tree of $s$ describes how $s \in L(G)$
  - Ambiguity: more than one parse tree (interpretation) for some string $s$
  - Error: no parse tree for some string $s$
  - How do we construct the parse tree?
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

This string has two parse trees

+ is left-associative
Ambiguity. Example

This string 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 right-most or left-most 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 * over +
• Enforces left-associativity of + and *
Ambiguity. Example

The `int * int + int` has only one parse tree now

```
E   E + T
  |  |
T   int
  |
  T * int
  |
  int
```

```
E   E *
  |  |
int E + E
  |  |
  int  int
```
Ambiguity: The Dangling Else

• Consider the grammar
  
  $E \rightarrow \text{if } E \text{ then } E$
  
  $| \text{if } E \text{ then } E \text{ else } E$
  
  $| \text{OTHER}$

• 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")

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

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

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

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

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

- A valid parse tree (for a UIF)

- Not valid because the then expression is not a MIF
Ambiguity

• No general techniques for handling 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
  - We need disambiguation mechanisms
Precedence and Associativity Declarations

• 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 \[ E \to E + E \mid E \ast E \mid \text{int} \]
  - And the string \( \text{int} + \text{int} \ast \text{int} \)

- Precedence declarations: %left +
  %left *
Review

• We can specify language syntax using CFG
• A parser will answer whether $s \in L(G)$
• ... and will build a parse tree
• ... and pass on to the rest of the compiler

• Next:
  - How do we answer $s \in L(G)$ and build a parse tree?
Approach 1
Top-Down Parsing
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
Recursive Descent Parsing

- Consider the grammar
  \[
  E \rightarrow T + E | T \\
  T \rightarrow \text{int} | \text{int} \times T | (E)
  \]
- Token stream is: \( \text{int}_5 \times \text{int}_2 \)
- Start with top-level non-terminal \( E \)
- Try the rules for \( E \) in order
Recursive Descent Parsing. Example (Cont.)

• Try $E_0 \rightarrow T_1 + E_2$
• Then try a rule for $T_1 \rightarrow (E_3)$
  - But $(\text{does not match input token int}_5$
• Try $T_1 \rightarrow \text{int}$. Token matches.
  - But $+\text{ after } T_1 \text{ does not match input token } *$
• Try $T_1 \rightarrow \text{int} * T_2$
  - This will match but $+\text{ after } T_1 \text{ will be unmatched}$
• Have exhausted the choices for $T_1$
  - Backtrack to choice for $E_0$
Recursive Descent Parsing. Example (Cont.)

• Try $E_0 \rightarrow T_1$

• Follow same steps as before for $T_1$
  - And succeed with $T_1 \rightarrow \text{int} \ast T_2$ and $T_2 \rightarrow \text{int}$
  - With the following parse tree

```
  E_0
     |
     T_1
|
int_5 *   T_2
    |
    int_2
```
Recursive Descent Parsing. Notes.

- Easy to implement by hand
  - An example implementation is provided as a supplement “Recursive Descent Parsing”

- But does not always work ...
Recursive-Descent Parsing

• Parsing: given a string of tokens $t_1 \ t_2 \ldots \ t_n$, find its parse tree

• Recursive-descent parsing: Try all the productions exhaustively
  - At a given moment the fringe of the parse tree is: $t_1 \ t_2 \ldots \ t_k \ A \ldots$
  - Try all the productions for $A$: if $A \rightarrow BC$ is a production, the new fringe is $t_1 \ t_2 \ldots \ t_k \ B \ C \ldots$
  - Backtrack when the fringe doesn’t match the string
  - Stop when there are no more non-terminals
When Recursive Descent Does Not Work

- Consider a production $S \rightarrow S \alpha$:
  - In the process of parsing $S$ we try the above rule
  - What goes wrong?

- A left-recursive grammar has a non-terminal $S$ for some $\alpha$

- Recursive descent does not work in such cases
  - It goes into an $\infty$ loop
Elimination of Left Recursion

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

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

• Can rewrite using right-recursion
  \[ S \rightarrow \beta S' \]
  \[ S' \rightarrow \alpha S' | \varepsilon \]
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
  \[ S \rightarrow S \alpha_1 | \ldots | S \alpha_n | \beta_1 | \ldots | \beta_m \]

• 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
  \[
  S \rightarrow \beta_1 S' | \ldots | \beta_m S' \\
  S' \rightarrow \alpha_1 S' | \ldots | \alpha_n S' | \varepsilon
  \]
General Left Recursion

- The grammar
  \[ S \rightarrow A \alpha \mid \delta \]
  \[ A \rightarrow S \beta \]
  is also left-recursive because
  \[ S \rightarrow^+ S \beta \alpha \]

- This left-recursion can also be eliminated
- See Dragon Book, Section 4.3 for general algorithm
Summary of Recursive Descent

• Simple and general parsing strategy
  - Left-recursion must be eliminated first
  - ... but that can be done automatically

• Unpopular because of backtracking
  - Thought to be too inefficient

• In practice, backtracking is eliminated by restricting the grammar
Predictive Parsers

• Like recursive-descent but parser can “predict” 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

• In recursive-descent, for each non-terminal and input token there may be a choice of production
• LL(1) means that for each non-terminal and token there is only one production that could lead to success
• 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
Predictive Parsing and Left Factoring

• Recall 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

• Recall the grammar

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

• Factor out common prefixes of productions

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

- **Left-factored grammar**
  
  \[
  \begin{align*}
  E & \rightarrow TX \\
  T & \rightarrow (E) \mid \text{int } Y \\
  X & \rightarrow + E \mid \epsilon \\
  Y & \rightarrow * T \mid \epsilon
  \end{align*}
  \]

- **The LL(1) parsing table:**

<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</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td>int Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>+ E</td>
<td></td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td>* T</td>
<td></td>
<td></td>
<td>ε</td>
<td></td>
</tr>
</tbody>
</table>

Profs. Necula  CS 164  Lecture 6-7
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,*] 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 each 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 $> and next (pointer to tokens)
repeat
  case stack of
    <X, rest> : if T[X,*next] = Y_1...Y_n
                then stack ← <Y_1... Y_n rest>;
                else error ();
    <t, rest> : if t == *next ++
                then stack ← <rest>;
                else error ();
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></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>ε</td>
</tr>
<tr>
<td>X $</td>
<td>$</td>
<td>ε</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>ACCEPT</td>
</tr>
</tbody>
</table>
Constructing Parsing Tables

• LL(1) languages are those defined by a parsing table for the LL(1) algorithm

• No table entry can be multiply defined

• 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

```
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

```
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 \( b \)
Top-Down Parsing. Review

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

![](image)

- 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 + E
  / 
int T int
 /        /        / 
int * 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 $b$
Predictive Parsing. Review.

• A predictive parser is described by a table
  - For each non-terminal $A$ and for each token $b$ we specify a production $A \rightarrow \alpha$
  - When trying to expand $A$ we use $A \rightarrow \alpha$ if $b$ follows next

• Once we have the table
  - The parsing algorithm is simple and fast
  - No backtracking is necessary
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 \( \gamma \)
   - Means that \( b \) can appear after \( A \) in a derivation of the form \( S \rightarrow \ast \beta Ab\omega \)
   - 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 \( \varepsilon \)
  - We say that \( \varepsilon \in \text{First}(A) \) in this case
Computing First Sets

Definition \( \text{First}(X) = \{ b \mid X \rightarrow^* b\alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \} \)

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

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

• Recall the grammar

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

• First sets

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

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

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 \( \varepsilon \notin \text{First}(A_2) \)
Follow Sets. Example

- Recall the grammar

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

- Follow sets

\[
\begin{align*}
\text{Follow}(+) & = \{ \text{int}, ( ) \} \\
\text{Follow}(* ) & = \{ \text{int}, ( ) \} \\
\text{Follow}( ( ) ) & = \{ \text{int}, ( ) \} \\
\text{Follow}(X ) & = \{ $, , ) \} \\
\text{Follow}(T ) & = \{ +, , ) \}, $\} \\
\text{Follow}(Y ) & = \{ +, , ) \}, $\} \\
\text{Follow}(\text{int}) & = \{ *, +, , ) \}, $\}
\end{align*}
\]
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 $\alpha \rightarrow \epsilon$, for each $b \in \text{Follow}(A)$ do
    • $T[A, b] = \alpha$
  - If $\alpha \rightarrow \epsilon$ and $\$ \in \text{Follow}(A)$ do
    • $T[A, \$] = \alpha$
Constructing LL(1) Tables. Example

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

• Where in the row of \( Y \) do we put \( Y \rightarrow * T \) ?
  - In the lines of First(\( *T \)) = \{ * \}

• Where in the row of \( Y \) do we put \( Y \rightarrow \varepsilon \) ?
  - In the lines of 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
Review

• For some grammars there is a simple parsing strategy
  - Predictive parsing

• Next: a more powerful parsing strategy