Top-Down Parsing
and
Intro to Bottom-Up Parsing

Lecture 7
Predictive Parsers

• Like recursive-descent but parser can “predict” which production to use

• Predictive parsers are never wrong
  – Always able to guess correctly which production will lead to a successful parse, provided a string is in L(G).

• Two strategies allow this:
  – By looking at next few tokens
    • Lookahead
  – By restricting the form of the grammar
Predictive Parsers

• Advantage: No backtracking
  - So parsing is completely “deterministic”

• Predictive parsers accept LL(k) grammars
  - L means “left-to-right” scan of input
    • We always do this, so all our techniques would have “L” in first position
  - L means “leftmost derivation”
  - k means “predict based on k tokens of lookahead”
    • Theory is developed for arbitrary k, but...
  - In practice, LL(1) is used
LL(1) vs. Recursive Descent

• In recursive-descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices

• In LL(1),
  - At each step, only one choice of production
  - That is
    • When a non-terminal $A$ is leftmost non-terminal in a derivation...
    • And the next input symbol is token $t$
    • There is a unique production $A \rightarrow \alpha$ to use
      - Or no production to use (an error state)
      - Any other production is guaranteed to be incorrect...
      - But even the single production $A \rightarrow \alpha$ might not end up succeeding
    - Put another way, in LL(1), there is AT MOST one production to be used in a given situation
LL(1) vs. Recursive Descent

• In recursive-descent,
  - At each step, many choices of production to use
  - Backtracking used to undo bad choices

• In LL(1),
  - At each step, only one choice of production
  - That is
    • When a non-terminal $A$ is leftmost non-terminal in a derivation...
    • And the next input symbol is token $t$
    • There is a unique production $A \rightarrow \alpha$ to use
      - Or no production to use (an error state)

• LL(1) is a recursive descent variant without backtracking
Predictive Parsing and Left Factoring

• Recall our favorite grammar

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

• Hard to predict because
  - For \( T \) two productions start with \text{int}
    • With lookahead 1, can’t choose which production
  - For \( E \) it is not clear how to predict
    • What’s more \( T \) is a non-terminal so how do we even do the prediction?
    • Regardless \( T \) starts both productions of \( E \), so with single token of lookahead, not going to be easy to know what to do
Predictive Parsing and Left Factoring

• Recall our favorite grammar

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

• Hard to predict because
  - For \( T \) two productions start with \text{int}
    • With lookahead 1, can’t choose which production
  - For \( E \) it is not clear how to predict
    • What’s more \( T \) is a non-terminal so how do we even do the prediction?

• We need to \text{left-factor} the grammar

This grammar is unacceptable for LL(1) parsing
The Idea Behind Left Factoring

- Eliminate the common prefixes of multiple productions for a given non-terminal
  - In English: If for some non-terminal there are multiple productions that have the same prefix, we want to get rid of that (somehow)

- Ex:
  
  \[
  \begin{align*}
  E & \rightarrow T + E \mid T \\
  T & \rightarrow \text{int} \mid \text{int} \ast T \mid (E)
  \end{align*}
  \]

  \[
  \text{E has two productions with prefix T} \\
  \text{T has two productions with prefix int}
  \]
Left-Factoring Example

• Recall the grammar
  
  \[ E \rightarrow T + E \mid T \]
  
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

• Factor out common prefixes of productions
  
  • So the prefix appears in only one production

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

  But multiple suffixes!
  
  New nonterminals \( X \)
  
  and \( Y \) handle suffixes
Left-Factoring Example

• Recall the grammar

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

• Factor out common prefixes of productions
  • So the prefix appears in only one production

  \[
  E \rightarrow TX \\
  X \rightarrow +E \mid \varepsilon \\
  T \rightarrow (E) \mid \text{int}Y \\
  Y \rightarrow \ast T \mid \varepsilon
  \]

  Effectively delays the decision about which production we're using
**LL(1) Parsing Table Example**

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

- **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>E</td>
<td>T X</td>
<td>X</td>
<td>X</td>
<td>TX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>+ E</td>
<td></td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
<tr>
<td>T</td>
<td>\text{int } Y</td>
<td></td>
<td></td>
<td>(E )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>* T</td>
<td></td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
<td>\varepsilon</td>
</tr>
</tbody>
</table>

- **Notes:**
  - **leftmost non-terminal**
  - **next input token**
  - **rhs of production to use**
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 \(E \rightarrow T X\)”
  – This can generate an \(\text{int}\) in the first position

• Consider the \([Y,+]\) entry
  – “When current non-terminal is \(Y\) and current token is \(+\), get rid of \(Y\)”
  – \(Y\) can be followed by \(+\) only if \(Y \rightarrow \varepsilon\)
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 the leftmost non-terminal $S$
  - We look at the next input token $a$
  - And choose the production shown at $[S,a]$

• A stack records frontier of parse tree
  - Non-terminals that have yet to be expanded
  - Terminals that have yet to matched against the input
  - Top of stack = leftmost pending terminal or non-terminal

• Reject on reaching error state
• Accept on end of input & empty stack
LL(1) Parsing Algorithm

initialize stack = <S $> and next
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 Algorithm

initialize stack = <S $> and next
repeat
    case stack of
        <X, rest> : if T[X,*next] = Y₁...Yₙ
                    then stack ← <Y₁... Yₙ rest>;
                    else error ();
        <t, rest> : if t == *next ++
                    then stack ← <rest>;
                    else error ();
    until stack == < >

$ marks bottom of stack
For non-terminal X on top of stack, lookup production
Pop X, push production rhs on stack. Note leftmost symbol of rhs is on top of the stack.
For terminal t on top of stack, check t matches next input token.
## 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>$</td>
<td>$</td>
</tr>
</tbody>
</table>

**ACCEPT**
Our Short Term Goal

• How do we construct LL(1) parse tables?

• What are the conditions necessary for constructing LL(1) parse tables?
Constructing Parsing Tables: The Intuition

- Consider non-terminal $A$, production $A \rightarrow \alpha$, & token $t$

The question: Given $A$ and $t$, under what conditions will we make the move $A \rightarrow \alpha$?

That is, under what conditions is $T[A,t] = \alpha$?
Constructing Parsing Tables: The Intuition

• Consider non-terminal \( A \), production \( A \rightarrow \alpha \), & token \( t \)
• \( T[A,t] = \alpha \) in two cases:
  - If \( \alpha \rightarrow^* t \beta \)
    - \( \alpha \) can derive a \( t \) in the first position
    - We say that \( t \in \text{First}(\alpha) \)

Note this is \( \rightarrow^* \)
Constructing Parsing Tables: The Intuition

• Consider non-terminal $A$, production $A \rightarrow \alpha$, & token $t$
• $T[A,t] = \alpha$ in two cases:

• Now, assume $t \notin \text{First}(\alpha)$
  - Doesn’t sound very promising to use $\alpha$
  - But it turns out it may not be hopeless to use $A \rightarrow \alpha$

• If $A \rightarrow \alpha$ and $\alpha \rightarrow^* \varepsilon$ and $S \rightarrow^* \beta A \dagger \delta$
  - Useful if stack has $A$, input is $t$, and $A$ cannot derive $t$
  - In this case only option is to get rid of $A$ (by deriving $\varepsilon$)
    • Can work only if $t$ can follow $A$ in at least one derivation
  - We say $t \in \text{Follow}(A)$
  - I.e., $t$ is one of the things that can come after $A$ in the grammar
Often A Point of Confusion

- We are NOT talking about A deriving $t$
  - A does not produce $t$
  - We ARE talking about $t$ appearing in a derivation directly after A
  - So this has nothing to do with what A produces
  - Has to do with where A can appear in derivations

But for right now, let’s concentrate on First sets
(We’ll get to Follow sets in a bit)
Computing First Sets

Definition

\[
\text{First}(X) = \{ t \mid X \rightarrow^* t\alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \}
\]

• Note: X can be a single terminal, it could be a single non-terminal, or it could be a string of grammar symbols
• \( t \) however, must be a terminal
• For technical reasons, \( \varepsilon \) needs to be in \( \text{First}(X) \) if it's the case that \( X \) can go to \( \varepsilon \) in zero or more steps
  • Need to keep track of this in order to compute all of the terminals that are in the first set of a given grammar symbol
Computing First Sets

Definition

\[
\text{First}(X) = \{ \, t \mid X \rightarrow^* \, t \alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \}
\]

Algorithm sketch:

1. For \( t \) a terminal, \( \text{First}(t) = \{ \, t \, \} \)
2. For \( X \) a non-terminal \( \varepsilon \in \text{First}(X) \)
   • if \( X \rightarrow \varepsilon \)
   • if \( X \rightarrow A_1 \ldots A_n \) and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)
   • Note this can only happen if all of the \( A_i \) are non-terminals, since if there are any terminals on the R.H.S. then it can never completely go to \( \varepsilon \)
Computing First Sets

Definition

\[
\text{First}(X) = \{ t \mid X \rightarrow^* t\alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \}
\]

Algorithm sketch:
1. For \( t \) a terminal, \( \text{First}(t) = \{ t \} \)
2. For \( X \) a non-terminal \( \varepsilon \in \text{First}(X) \)
   • if \( X \rightarrow \varepsilon \)
   • if \( X \rightarrow A_1 \ldots A_n \) and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)
3. \( \text{First}(\alpha) \subseteq \text{First}(X) \) if \( X \rightarrow A_1 \ldots A_n \alpha \)
   - and \( \varepsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)

Make sure it’s clear to you why (3) is true
Computing First Sets

Definition
\[ \text{First}(X) = \{ t \mid X \rightarrow^* t \alpha \} \cup \{ \epsilon \mid X \rightarrow^* \epsilon \} \]

Algorithm sketch:
1. For \( t \) a terminal, \( \text{First}(t) = \{ t \} \)
2. For \( X \) a non-terminal \( \epsilon \in \text{First}(X) \)
   - if \( X \rightarrow \epsilon \)
   - if \( X \rightarrow A_1 \ldots A_n \) and \( \epsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)
3. \( \text{First}(\alpha) \subseteq \text{First}(X) \) if \( X \rightarrow A_1 \ldots A_n \alpha \)
   - and \( \epsilon \in \text{First}(A_i) \) for \( 1 \leq i \leq n \)

Note: Rule (1) covers terminals; (2) and (3) cover non-terminals
First Sets. Example

• Recall the grammar

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

• First sets

\[
\begin{align*}
\text{First( ( ) )} & = \{ ( ) \} \\
\text{First( ) } & = \{ ( ) \} \\
\text{First( int) } & = \{ \text{int} \} \\
\text{First( + ) } & = \{ + \} \\
\text{First( * ) } & = \{ * \}
\end{align*}
\]

\[
\begin{align*}
\text{First( T )} & = \{ \text{int, ( )} \} \\
\text{First( E )} & = \{ \text{int, ( )} \} \\
\text{First( X )} & = \{ +, \epsilon \} \\
\text{First( Y )} & = \{ *, \epsilon \}
\end{align*}
\]
Computing Follow Sets

• Definition:

\[ \text{Follow}(X) = \{ t \mid S \rightarrow^* \beta \ X \ t \ \delta \} \]

• Recall that the definition of the Follow set for a given symbol in the grammar is not about what that symbol can generate, but on where that symbol can appear.

• In words, \( t \) is in \( \text{Follow}(X) \) if there is some derivation where terminal \( t \) can appear immediately after the symbol \( X \)
Computing Follow Sets

• Definition:

\[ \text{Follow}(X) = \{ t \mid S \rightarrow^* \beta X t \delta \} \]

• Intuition

- If \( X \rightarrow A B \) then \( \text{First}(B) \subseteq \text{Follow}(A) \) and \( \text{Follow}(X) \subseteq \text{Follow}(B) \)
- If \( B \rightarrow^* \epsilon \) then \( \text{Follow}(X) \subseteq \text{Follow}(A) \)
- If \( S \) is the start symbol then \( \$ \in \text{Follow}(S) \)

Recall that \( \$ \) is special symbol marking end of input
Computing Follow Sets

• Definition:
  \[ \text{Follow}(X) = \{ t \mid S \rightarrow^* \beta \ X \ t \ \delta \} \]

• Intuition
  - If \( X \rightarrow A \ B \) then \( \text{First}(B) \subseteq \text{Follow}(A) \) and \( \text{Follow}(X) \subseteq \text{Follow}(B) \)
  - if \( B \rightarrow^* \epsilon \) then \( \text{Follow}(X) \subseteq \text{Follow}(A) \)
  - If \( S \) is the start symbol then \( \$ \in \text{Follow}(S) \)
  - That is, \( \$ \) is in the Follow of the start symbol
    - Always added as an initial condition
Computing Follow Sets (Cont.)

Algorithm sketch:
1. $\$ \in \text{Follow}(S)$
2. First($\beta$) - $\{\varepsilon\} \subseteq \text{Follow}(X)$
   - For each production $A \rightarrow \alpha X \beta$
3. Follow($A$) $\subseteq$ Follow($X$)
   - For each production $A \rightarrow \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$
Follow Sets. Example

• Recall the grammar

\[
E \rightarrow TX \\
T \rightarrow (E) \mid \text{int } Y
\]

\[
X \rightarrow +E \mid \epsilon \\
Y \rightarrow *T \mid \epsilon
\]

• Follow sets

\[
\text{Follow}( + ) = \{ \text{int}, ( ) \} \\
\text{Follow}( * ) = \{ \text{int}, ( ) \} \\
\text{Follow}( ( ) ) = \{ \text{int}, ( ) \} \\
\text{Follow}( E ) = \{(), \$\} \\
\text{Follow}( X ) = \{ (), \$ \} \\
\text{Follow}( T ) = \{ +, (), \$ \} \\
\text{Follow}( Y ) = \{ +, (), \$ \} \\
\text{Follow}( \text{int} ) = \{ *, +, (), \$ \}
\]

Note, unlike with First sets, Follow sets for terminals can actually be interesting.
Follow Sets. Example

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

- Follow sets
  
  \begin{align*}
  \text{Follow}( + ) &= \{ \text{int}, ( ) \} \\
  \text{Follow}( * ) &= \{ \text{int}, ( ) \} \\
  \text{Follow}( ( ) ) &= \{ \text{int}, ( ) \} \\
  \text{Follow}( E ) &= \{ ( ) \}, \$ \\
  \text{Follow}( X ) &= \{ ( ) \} \\
  \text{Follow}( T ) &= \{ (+), ( ) \}, \$ \\
  \text{Follow}( ) &= \{ (+), ( ) \}, \$ \\
  \text{Follow}( \text{int} ) &= \{ (*), (+), ( ) \}, \$
  \end{align*}

Note First( ( )). It makes sense: what can follow an ( in the language? A nested ( or an int
Follow Sets. Example

- Recall the grammar

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

- Follow sets

\[
\begin{align*}
\text{Follow}(+) &= \{\text{int, (}\} \\
\text{Follow}(*)) &= \{\text{int, (}\} \\
\text{Follow}(()) &= \{\text{int, (}\} \\
\text{Follow}(E)) &= \{(), $\} \\
\text{Follow}(X)) &= \{(), $\} \\
\text{Follow}(T)) &= \{+, (), $\} \\
\text{Follow}(Y)) &= \{+, (), $\} \\
\text{Follow}(\text{int}) &= \{+, (), $\}
\end{align*}
\]

Similarly, First(+). What can follow + in the language? A new ( or an int. Can't be $ (end input)
So Now

- We’re going to pull together what we know about first and follow sets to construct LL(1) parsing tables.
- This is done one production at a time, eventually considering every production in the grammar.

Recall:

\[
\text{First}(X) = \{ t \mid X \rightarrow^* t\alpha \} \cup \{ \varepsilon \mid X \rightarrow^* \varepsilon \}
\]

\[
\text{Follow}(X) = \{ t \mid S \rightarrow^* \beta X t \delta \}
\]
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 $t \in \text{First}(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    - $T[A, \$] = \alpha$
      - (note this is a special case, since $\$ is technically not a terminal symbol)
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 $t \in \text{First}(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in \text{First}(\alpha)$ and $\$$ $\in \text{Follow}(A)$ do
    - $T[A, \$$] = \alpha$
    - (note this is a special case, since $\$$ is technically not a terminal symbol)

This is the algorithm for building LL(1) tables!
LL(1) Parsing Table Example

• *Left-factored grammar*

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

• *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>E</td>
<td>T X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>+E</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td></td>
<td>(E)</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>*T</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
<td>ε</td>
</tr>
</tbody>
</table>
### First

<table>
<thead>
<tr>
<th>Syntax</th>
<th>First</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>{ ( ) }</td>
</tr>
<tr>
<td>( )</td>
<td>{ ( ) }</td>
</tr>
<tr>
<td>int</td>
<td>{ int }</td>
</tr>
<tr>
<td>+</td>
<td>{ + }</td>
</tr>
<tr>
<td>*</td>
<td>{ * }</td>
</tr>
</tbody>
</table>

### Follow

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Follow</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>{ int, ( ) }</td>
</tr>
<tr>
<td>( )</td>
<td>{ int, ( ) }</td>
</tr>
<tr>
<td>X</td>
<td>{ $, ( ) }</td>
</tr>
<tr>
<td>)</td>
<td>{ +, ) , $ }</td>
</tr>
<tr>
<td>int</td>
<td>{ * , +, ) , $ }</td>
</tr>
</tbody>
</table>
Another Example

\[ S \rightarrow Sa \mid b \]

First(S) = \{b\}, Follow(S) = \{$, a\}

• The LL(1) parsing table:

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td>b</td>
<td>ε</td>
</tr>
<tr>
<td></td>
<td>Sa</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The problem: both productions can produce a b in the first position, giving entry with multiple moves
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
  - In fact, definition of LL(1) is that a grammar is NOT LL(1) iff the built LL(1) table has a multiply defined entry

• Most programming language CFGs are not LL(1)

These amount to quick checks (for NOT LL(1)-ness)
Bottom Line on LL(1) Parsing

• **Most programming language CFGs are not LL(1)**
  - LL(1) grammars are just too weak to capture all of the interesting and important structures in real-world programming languages

• There are more powerful formalisms for describing practical grammars

• **So, why bother with LL(1)?**
  - Well, it turns out that these formalisms build on everything we’ve learned here for LL(1) grammars, so our effort is not wasted.
  - Ideas just assembled in a more sophisticated way to build more powerful parts
Bottom-Up Parsing

• Bottom-up parsing is more general than (deterministic) top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing

• Bottom-up is the preferred method
  - Used in most parser generator tools (including CUP)

• Concepts today, algorithms next time
An Introductory Example

• Bottom-up parsers don't need left-factored grammars
  - Recall what left-factored means

• Revert to the “natural” grammar for our example:
  \[
  \begin{align*}
  E &\rightarrow T + E \mid T \\
  T &\rightarrow \text{int} \ast T \mid \text{int} \mid (E)
  \end{align*}
  \]

• “natural” in quotes because we still have to deal with precedence of + and *
An Introductory Example

• Bottom-up parsers don’t need left-factored grammars
  - Recall what left-factored means

• Revert to the “natural” grammar for our example:
  \[
  E \rightarrow T + E \mid T \\
  T \rightarrow \text{int} \ast T \mid \text{int} \mid (E)
  \]

• Consider the string: \text{int} \ast \text{int} + \text{int}
The Idea

Bottom-up parsing reduces a string to the start symbol by inverting productions (running them backwards)

\[
\begin{align*}
\text{int} * \text{int} + \text{int} & \quad T \rightarrow \text{int} \\
\text{int} * T + \text{int} & \quad T \rightarrow \text{int} * T \\
T + \text{int} & \quad T \rightarrow \text{int} \\
T + T & \quad E \rightarrow T \\
T + E & \quad E \rightarrow T + E
\end{align*}
\]
Observation

• Read the productions in reverse (from bottom to top)
  - Read in the order we did them, called reductions
• This is a rightmost derivation!

\[
\begin{align*}
  \text{int} \times \text{int} + \text{int} & \quad \text{T} \rightarrow \text{int} \\
  \text{int} \times \text{T} + \text{int} & \quad \text{T} \rightarrow \text{int} \times \text{T} \\
  \text{T} + \text{int} & \quad \text{T} \rightarrow \text{int} \\
  \text{T} + \text{T} & \quad \text{E} \rightarrow \text{T} \\
  \text{T} + \text{E} & \quad \text{E} \rightarrow \text{T} + \text{E} \\
  \text{E} & \\
\end{align*}
\]
Important Fact #1

Important Fact #1 about bottom-up parsing:

A bottom-up parser traces a rightmost derivation in reverse.

If you’re ever having trouble with bottom-up parsing, it’s good to come back to this basic fact.
The Idea

• A top-down parser begins with the start symbol and produces the tree incrementally by expanding some non-terminal at the frontier

• A bottom-up parser begins with all ALL the leaves of the parse tree (the entire input) and builds little trees on top of those
  - It pastes all the subtrees that it’s built so far together to create the entire parse tree
A Bottom-up Parse

\[
\text{int} \times \text{int} + \text{int} \\
\text{int} \times T + \text{int} \\
T + \text{int} \\
T + T \\
T + E \\
E
\]
A Bottom-up Parse in Detail (1)

int * int + int

int * int + int
A Bottom-up Parse in Detail (2)

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

\[ \text{int} * T + \text{int} \]
A Bottom-up Parse in Detail (3)

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

\[ \text{int} \ast T + \text{int} \]

\[ T + \text{int} \]
A Bottom-up Parse in Detail (4)

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

\[\text{int} \times T + \text{int}\]

\[T + \text{int}\]

\[T + T\]
A Bottom-up Parse in Detail (5)

\[
\begin{align*}
\text{int} \ast \text{int} & + \text{int} \\
\text{int} \ast T & + \text{int} \\
T & + \text{int} \\
T & + T \\
T & + E
\end{align*}
\]
A Bottom-up Parse in Detail (6)

\[
\text{int} \times \text{int} + \text{int} \\
\text{int} \times T + \text{int} \\
T + \text{int} \\
T + T \\
T + E \\
E
\]

\[
E \\
T \quad T \\
int \times int + int
\]
A Trivial Bottom-Up Parsing Algorithm

Let $I =$ input string
repeat
  pick a non-empty substring $\beta$ of $I$ where $X \rightarrow \beta$ is a production
  if no such $\beta$, backtrack
  replace one $\beta$ by $X$ in $I$
until $I =$ "S" (the start symbol) or all possibilities are exhausted
Questions

• Does this algorithm terminate?

• How fast is the algorithm?

• Does the algorithm handle all cases?

• How do we choose the substring to reduce at each step?
Where Do Reductions Happen?

Important Fact #1 has an interesting consequence:
- Let $\alpha\beta\omega$ be a step of a bottom-up parse
- Assume the next reduction is by $X \rightarrow \beta$
- Then $\omega$ is a string of terminals

Why? Because $\alpha X \omega \rightarrow \alpha \beta \omega$ is a step in a rightmost derivation

Recall Fact #1: Bottom-up parser traces a rightmost derivation in reverse.
Notation

- **Idea:** Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
    - Turns out terminal symbols to right of right most non-terminal are exactly the unexamined input in bottom-up parsing
  - Left substring has terminals and non-terminals
  - E.g., if we have $aX\omega$, and $X$ is the rightmost non-terminal, then $\omega$ is the input we have not read yet
Notation

• The dividing point is marked by a |
  - The | is not part of the string
• Initially, all input is unexamined |$x_1x_2 \ldots x_n$

• After some input has been examined:

  $x_1x_2x_3|x_4 \ldots x_n$

  \begin{align*}
  \text{processed} & \quad \text{unprocessed}
  \end{align*}
Shift-Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

- **Shift moves**
- **Reduce moves**
Shift

- **Shift**: Move one place to the right
  - Shifts a terminal to the left string
  - Equivalently reads one token of input
  - In example below, the shift indicates that token $x$ can now be considered as part of processing
  - $y$ and $z$ remain unprocessed and unread at this point

\[ ABC|xyz \Rightarrow ABCx|yz \]
Reduce Move

• Apply an inverse production at the right end of the left string
  - If $A \rightarrow xy$ is a production, then

$$Cbxy|ijk \Rightarrow CbA|ijk$$
**Previously Seen Example with Reductions Only**

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

- \(\text{int} \ast \text{int} \mid + \text{int}\) reduce \(T \rightarrow \text{int}\)
- \(\text{int} \ast T \mid + \text{int}\) reduce \(T \rightarrow \text{int} \ast T\)

- \(T + \text{int}\) reduce \(T \rightarrow \text{int}\)
- \(T + T\) reduce \(E \rightarrow T\)
- \(T + E\) reduce \(E \rightarrow T + E\)
- \(E\)
### The Example with Shift-Reduce Parsing

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Action</th>
<th>Parse Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>`</td>
<td>int * int + int`</td>
<td>shift</td>
</tr>
<tr>
<td>`int</td>
<td>* int + int`</td>
<td>shift</td>
</tr>
<tr>
<td>`int *</td>
<td>int + int`</td>
<td>shift</td>
</tr>
<tr>
<td>`int * int</td>
<td>+ int`</td>
<td>reduce</td>
</tr>
<tr>
<td>`int * T</td>
<td>+ int`</td>
<td>reduce</td>
</tr>
<tr>
<td>`T</td>
<td>+ int`</td>
<td>shift</td>
</tr>
<tr>
<td>`T +</td>
<td>int`</td>
<td>shift</td>
</tr>
<tr>
<td>`T + int</td>
<td>`</td>
<td>reduce</td>
</tr>
<tr>
<td>`T + T</td>
<td>`</td>
<td>reduce</td>
</tr>
<tr>
<td>`T + E</td>
<td>`</td>
<td>reduce</td>
</tr>
</tbody>
</table>

E |
• Note: In this derivation, and in the details of it that follows, all I’m showing is that there exists a sequence of shift and reduce moves that can successfully parse the input string.

• I do not (yet) explain how we choose whether to perform a shift or reduce move.
A Shift-Reduce Parse in Detail (1)

\[ \text{int} \times \text{int} + \text{int} \]
A Shift-Reduce Parse in Detail (2)

<table>
<thead>
<tr>
<th>int * int + int</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
</tbody>
</table>

int * int + int

↑
A Shift-Reduce Parse in Detail (3)

\[
\begin{align*}
| \text{int} & \ast \text{int} + \text{int} \\
\text{int} & \mid \ast \text{int} + \text{int} \\
\text{int} & \ast \mid \text{int} + \text{int}
\end{align*}
\]
A Shift-Reduce Parse in Detail (4)

<table>
<thead>
<tr>
<th>int * int + int</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
<tr>
<td>int *</td>
</tr>
<tr>
<td>int * int</td>
</tr>
</tbody>
</table>

```
int * int + int
```
A Shift-Reduce Parse in Detail (5)

\[
\begin{align*}
| & \text{int} \times \text{int} + \text{int} \\
\text{int} & | \ast \text{int} + \text{int} \\
\text{int} \times & | \text{int} + \text{int} \\
\text{int} \times \text{int} & | + \text{int} \\
\text{int} \times \text{T} & | + \text{int}
\end{align*}
\]
A Shift-Reduce Parse in Detail (6)

\[
\begin{align*}
| & \text{int} \times \text{int} + \text{int} \\
\text{int} & | \times \text{int} + \text{int} \\
\text{int} & | \times \text{int} + \text{int} \\
\text{int} \times \text{int} & | + \text{int} \\
\text{int} \times \text{T} & | + \text{int} \\
\text{T} & | + \text{int}
\end{align*}
\]
A Shift-Reduce Parse in Detail (7)

<table>
<thead>
<tr>
<th>int * int + int</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int + int</td>
</tr>
<tr>
<td>int *</td>
</tr>
<tr>
<td>int * int</td>
</tr>
<tr>
<td>int * T</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>T +</td>
</tr>
</tbody>
</table>
A Shift-Reduce Parse in Detail (8)

<table>
<thead>
<tr>
<th>int * int + int</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * int + int</td>
</tr>
<tr>
<td>int * int + int</td>
</tr>
<tr>
<td>int * int + int</td>
</tr>
<tr>
<td>int * T + int</td>
</tr>
<tr>
<td>T + int</td>
</tr>
<tr>
<td>T + int</td>
</tr>
</tbody>
</table>

```
T

int * int + int
```

↑
A Shift-Reduce Parse in Detail (9)

| int * int + int
int | * int + int
int * | int + int
int * int | + int
int * T | + int
T | + int
T + | int
T + int |
T + T |

\[
\begin{align*}
T \\
\text{int} & \quad * \quad \text{int} & \quad + \quad \text{int} & \quad \text{int} \\
\end{align*}
\]
A Shift-Reduce Parse in Detail (10)

\[
\begin{align*}
\text{int} & \mid \text{int} \times \text{int} + \text{int} \\
\text{int} & \mid \text{int} \times \text{int} + \text{int} \\
\text{int} & \mid \text{int} \times \text{int} + \text{int} \\
\text{int} & \mid \text{int} \times \text{int} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\end{align*}
\]

![Parse Tree]

\[
\begin{align*}
\text{int} & \mid \text{int} \times \text{int} + \text{int} \\
\text{int} & \mid \text{int} \times \text{int} + \text{int} \\
\text{int} & \mid \text{int} \times \text{int} + \text{int} \\
\text{int} & \mid \text{int} \times \text{int} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\text{T} & \mid \text{T} + \text{int} \\
\end{align*}
\]
A Shift-Reduce Parse in Detail (11)

<table>
<thead>
<tr>
<th>int * int + int</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
</tr>
<tr>
<td>int *</td>
</tr>
<tr>
<td>int * int</td>
</tr>
<tr>
<td>int * T</td>
</tr>
<tr>
<td>T</td>
</tr>
<tr>
<td>T +</td>
</tr>
<tr>
<td>T + int</td>
</tr>
<tr>
<td>T + T</td>
</tr>
<tr>
<td>T + E</td>
</tr>
<tr>
<td>E</td>
</tr>
</tbody>
</table>

```
E
  +
  |
  T
  *
  int

int
```
The Stack

• Left string can be implemented by a stack
  - Top of the stack is the |

• Shift pushes a terminal on the stack

• Reduce pops 0 or more symbols off of the stack (production rhs) and pushes a non-terminal on the stack (production lhs)
Conflicts

- In a given state, more than one action (shift or reduce) may lead to a valid parse.

- If both a shift and a reduce are possible at some juncture, there is a shift-reduce conflict.

- If it is legal to reduce by two different productions, there is a reduce-reduce conflict.

- You will see such conflicts in your project!
  - More next time...