Top-Down Parsing and Intro to Bottom-Up Parsing

Lecture 7

Dr. Aiken

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) 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 in a derivation
    - The next input symbol is t
    - There is a unique production A → α 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 the grammar
  \[ E \rightarrow T + E \mid T \]
  \[ T \rightarrow \text{int} \mid \text{int} \ast T \mid (E) \]

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

- We need to \textit{left-factor} the grammar
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
  \[ E \rightarrow T X \]
  \[ X \rightarrow + E | \varepsilon \]
  \[ T \rightarrow (E) | \text{int} Y \]
  \[ Y \rightarrow \ast T | \varepsilon \]

• 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>TX</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>+E</td>
<td></td>
<td>E</td>
<td>E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>int Y</td>
<td></td>
<td></td>
<td>E</td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>* T</td>
<td></td>
<td>E</td>
<td>E</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 int, use production \(E \rightarrow T X\)
  – This can generate an 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
    \(<X, \text{rest}>\) : if \( T[X,*\text{next}] = Y_1...Y_n \)
                         then stack \( \leftarrow <Y_1...Y_n, \text{rest}>; \)
                         else error ();
    \(<t, \text{rest}>\) : if \( t == *\text{next ++} \)
                         then stack \( \leftarrow <\text{rest}>; \)
                         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 &gt;)</td>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(TX)</td>
</tr>
<tr>
<td>(TX &gt;)</td>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(\text{int} Y)</td>
</tr>
<tr>
<td>(\text{int} Y X &gt;)</td>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(\text{terminal})</td>
</tr>
<tr>
<td>(\text{int} Y X &gt;)</td>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(\text{terminal})</td>
</tr>
<tr>
<td>(\text{int} Y &gt;)</td>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(\text{terminal})</td>
</tr>
<tr>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(\text{terminal})</td>
</tr>
<tr>
<td>(X &gt;)</td>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(\text{terminal})</td>
</tr>
<tr>
<td>($ &gt;)</td>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(\text{terminal})</td>
</tr>
<tr>
<td>($ &gt;)</td>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(\text{terminal})</td>
</tr>
<tr>
<td>($ &gt;)</td>
<td>(\text{int} \times \text{int} &gt;)</td>
<td>(\text{terminal})</td>
</tr>
</tbody>
</table>
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 $t \in \text{First}(\alpha)$

• If $A \rightarrow \alpha$ and $\alpha \rightarrow^* \varepsilon$ and $S \rightarrow^* \beta A t$
  - 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 $t \in \text{Follow}(A)$

Computing First Sets

Definition

$$\text{First}(X) = \{ t | X \rightarrow^* t \alpha \} \cup \{ \varepsilon | X \rightarrow^* \varepsilon \}$$

Algorithm sketch:
1. $\text{First}(t) = \{ t \}$
2. $\varepsilon \in \text{First}(X)$
   - 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 $1 \leq i \leq n$

First Sets, Example

• Recall the grammar

  $E \rightarrow T \ X$
  $T \rightarrow (E) \ | \ \text{int} \ Y$
  $X \rightarrow + \ E \ | \ \varepsilon$
  $Y \rightarrow \ast \ T \ | \ \varepsilon$

• First sets

  $\text{First}(()) = \{ () \}$
  $\text{First}(\text{int}) = \{ \text{int} \}$
  $\text{First}(()) = \{ () \}$
  $\text{First}(\text{int}) = \{ \text{int} \}$
  $\text{First}(\text{int}) = \{ \text{int} \}$
  $\text{First}(\ast) = \{ \ast \}$
  $\text{First}(\ast) = \{ \ast \}$
  $\text{First}(\ast) = \{ \ast \}$

Computing Follow Sets

• Definition:

  $\text{Follow}(X) = \{ t | S \rightarrow^* \beta X t \delta \}$

• Intuition

  - If $X \rightarrow A B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and $\text{Follow}(X) \subseteq \text{Follow}(B)$
  - if $B \rightarrow^* \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$
  - if $S$ is the start symbol then $\$$ \in \text{Follow}(S)$
Computing Follow Sets (Cont.)

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

Follow Sets. Example

- Recall the grammar
  \begin{align*}
  &E \rightarrow T X, \quad X \rightarrow + E | \epsilon \\
  &T \rightarrow (E) | \text{int} Y, \quad Y \rightarrow * T | \epsilon
  \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}(Y) = \{\text{, , (}\} \\
  &\text{Follow}(\text{int}) = \{\text{, , (}\}
  \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 $t \in \text{First}(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\epsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
    - $T[A, t] = \alpha$
  - If $\epsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
    - $T[A, \$] = \alpha$

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 CFGs are not LL(1)
Bottom-Up Parsing

- Bottom-up parsing is more general than top-down parsing
  - And just as efficient
  - Builds on ideas in top-down parsing
- Bottom-up is the preferred method
- Concepts today, algorithms next time

An Introductory Example

- Bottom-up parsers don’t need left-factored grammars
- 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:

- \(\text{int} \ast \text{int} + \text{int}\)
- \(\text{int} \ast T + \text{int}\)
- \(T + \text{int}\)
- \(T + E\)
- \(E\)

Observation

- Read the productions in reverse (from bottom to top)
- This is a rightmost derivation!

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

Important Fact #1 about bottom-up parsing:

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

---

A Bottom-up Parse

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

---

A Bottom-up Parse in Detail (1)

```
int * int + int
```

---

A Bottom-up Parse in Detail (2)

```
int * int + int
int * T + int
```

---
A Bottom-up Parse in Detail (3)

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

A Bottom-up Parse in Detail (4)

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

A Bottom-up Parse in Detail (5)

int * int + int
int * T + int
T + int
T + T
T + E

A Bottom-up Parse in Detail (6)

int * int + int
int * T + int
T + int
T + T
T + E
E
A Trivial Bottom-Up Parsing Algorithm

Let $I = \text{input string}$
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 $I = \text{"S" (the start symbol)}$ 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 right-most derivation

Notation

• Idea: Split string into two substrings
  - Right substring is as yet unexamined by parsing (a string of terminals)
  - Left substring has terminals and non-terminals

• The dividing point is marked by a $|$ from the string
  - The $|$ is not part of the string

• Initially, all input is unexamined $|x_1 x_2 \ldots x_n$
Shift–Reduce Parsing

Bottom-up parsing uses only two kinds of actions:

\[ \text{Shift} \]
\[ \text{Reduce} \]

Shift

- \text{Shift: Move | one place to the right}
  - Shifts a terminal to the left string

\[ \text{ABC|xyz } \Rightarrow \text{ABCx|yz} \]

Reduce

- \text{Apply an inverse production at the right end of the left string}
  - If \( A \rightarrow xy \) is a production, then

\[ \text{Cbxy|ijk } \Rightarrow \text{CbA|ijk} \]

The Example with Reductions Only

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

| int * int + int | shift |
| int | int + int | shift |
| int | int + int | shift |
| int * T | int + int | reduce T → int |
| int * int | int + int | reduce T → int * T |
| T | int | shift |
| T | int | shift |
| T + int | reduce T → int |
| T + T | reduce E → T |
| T + E | reduce E → T + E |
| E | |

A Shift-Reduce Parse in Detail (1)

| int | int + int |
| m |

A Shift-Reduce Parse in Detail (2)

| int | int + int |
| m |

A Shift-Reduce Parse in Detail (3)

| int | int + int |
| m |
A Shift-Reduce Parse in Detail (4)

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

A Shift-Reduce Parse in Detail (5)

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

A Shift-Reduce Parse in Detail (6)

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

A Shift-Reduce Parse in Detail (7)

| int * int + int  
| int | * int + int  
| int | int + int  
| int * int | + int  
| int * T | + int  
| T | + int  
| T + | int
A Shift-Reduce Parse in Detail (8)

A Shift-Reduce Parse in Detail (9)

A Shift-Reduce Parse in Detail (10)

A Shift-Reduce Parse in Detail (11)
The Stack

- Left string can be implemented by a stack
  - Top of the stack is the $\mid$
- 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 it is legal to shift or reduce, 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 . . .