Lecture 13

Top-Down Parsing

CS 241: Foundations of Sequential Programs
Spring 2017

Troy Vasiga et al. (where Dan Holtby ∈ al)
University of Waterloo
Parsing

Given a grammar $G$ and a word $w$, find a derivation for $w$.

Two strategies:

1. Top-down: Start with the start symbol. Find a non-terminal and replace it with a right-hand side of a rule. Repeat until you get to $w$.
   - What I was doing on Tuesday

2. Bottom-up: Start with $w$. Replace a right-hand side with a non-terminal. Repeat until you get to the start symbol.
   - Weird
   - But is it really?
   - Maybe

In both of the above strategies, we have to make the correct decision at each step. That’s the real trick.
Parsing Algorithm

- There is a backtracking algorithm for parsing in any CFG
  - try each rule in turn
  - if we can move “forward”, do so
  - if we cannot move “forward”, go back a step and try the “next” rule
  - stop when we find the derivation

- Backtracking is not practical.

- We will look at two (linear-time) algorithms.
Stack-based Parsing

For top-down parsing, we use a stack to remember information about our derivations and/or processed input.
Augmenting Grammars

Empty words and empty stacks can cause hassles.

We augment our grammars by adding “beginning” and “ending” characters.

Example:

2. \( S \rightarrow AyB \)
3. \( A \rightarrow ab \)
4. \( A \rightarrow cd \)
5. \( B \rightarrow z \)
6. \( B \rightarrow wz \)

On board: abywz
Top-down parsing with a stack

Invariant:

\[ \text{derivation} = \text{input already read} + \text{stack} \]
## Stack Example

<table>
<thead>
<tr>
<th>Derivation</th>
<th>Input read</th>
<th>Input to be read</th>
<th>Stack</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Observations:

What was I doing???

- Expanding production rules: To expand the rule $A \rightarrow \beta$:
  - Pop $A$
  - Push $\beta$ (i.e. push the symbols in the string $\beta$ in reverse order)
- Matching input: To match the character $a$:
  - Remove $a$ from the front of input
  - Remove $a$ from the top of the stack

How do we know we are done?

- We can’t expand or match

What happens then?

- We accept if input = stack = $\epsilon$
- We reject otherwise

How to know which rule to use?

- Oracle
LL(1) Parsing

We need: \( \text{Predict}(A, x) = A \rightarrow \alpha \) so long as
- \( A \) is on top of the stack, and
- \( x \) is the first symbol of input to be read

Definition of an LL(1) grammar:

\[ \forall A \in N, x \in T, |\text{Predict}(A, x)| \leq 1 \]

Meaning of:
- \( L \)
  - Leftmost character (of input)
- \( L \)
  - Leftmost derivation
- \( 1 \)
  - Looking 1 token ahead (i.e. at the next token and nothing else)
Constructing a Predictor Table

CFG:

1. $S' \rightarrow \vdash S \dashv$
2. $S \rightarrow AyB$
3. $A \rightarrow ab$
4. $A \rightarrow cd$
5. $B \rightarrow z$
6. $B \rightarrow wz$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>y</th>
<th>w</th>
<th>z</th>
<th>$\vdash$</th>
<th>$\dashv$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

10
Constructing a Predictor Table (with $\epsilon$)

CFG:

1. $S' \rightarrow \vdash S \dashv$
2. $S \rightarrow AyB$
3. $A \rightarrow ab$
4. $A \rightarrow cd$
5. $B \rightarrow z$
6. $B \rightarrow wz$
7. $B \rightarrow \epsilon$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>y</th>
<th>w</th>
<th>z</th>
<th>$\vdash$</th>
<th>$\dashv$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Algorithm to construct predicator table

Below, $\alpha, \beta \in (N \cup T)^*$, $x, y \in T$, $A \in N$

$\text{Empty}(\alpha) = \text{true if } \alpha \Rightarrow^* \epsilon$

$\text{First}(\alpha) = \{x \mid \alpha \Rightarrow^* x\beta\}$

$\text{Follow}(A) = \{y \mid S' \Rightarrow^* \alpha Ay\beta\}$

$\text{Predict}(A, x) =$
$\{A \to \alpha \mid x \in \text{First}(\alpha)\} \cup \{A \to \beta \mid x \in \text{Follow}(A) \text{ and } \text{Empty}(\beta)\}$
LL(1) Parsing algorithm

Input: \( w \)
push \( S' \)
for each \( x \in w \)
    while (top of stack is some \( A \in N \)) {
        pop \( A \)
        if \( \text{Predict}(A, x) = \{A \rightarrow \alpha\} \)
            push \( \alpha \)
        else
            reject
    }
pop \( c \)
if \( c \neq x \) reject
end for
accept \( w \)
Non LL(1) Grammars

1. $S \rightarrow ab$
2. $S \rightarrow acb$
Converting non-LL(1) grammars to LL(1) grammars

Factoring
A non LL(1) language

\[ L = \{ a^n b^m | n \geq m \geq 0 \} \]

Grammar (ambiguous)

Grammar (unambiguous)