COMP 181

Lecture 7
More parsing

September 26, 2006

Prelude
- What is this structure?
  Ryugyong Hotel, North Korea
- Facts
  - 105 floors, 1083 ft
  - 3000 rooms, 3.9 million sq. ft.
  - Started in 1987, halted 1992
  - DPRK: it doesn’t exist

Where are we
- Last time: Top-down parsing
  - Non-termination – eliminating left recursion
  - Using FIRST and FOLLOW sets
  - The LL(1) property
  - Recursive descent parsers
- Today:
  - Building FIRST and FOLLOW sets
  - Generating top-down parsers
  - Start bottom-up parsing

Top-down parsing
- Build parse tree top down

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A → αζβ</td>
</tr>
</tbody>
</table>

Is “CD”? Consider all possible strings derivable from “CD”.
What is the set of tokens that can appear at start?

| t5 ∈ FIRST(C D) | t5 ∈ FOLLOW(B) |

Left factoring
- Problem
  - What if my grammar is not LL(1)?
  - May be able to fix it, with transformations
- Example:

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A → αζβ</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A → αζβ</td>
</tr>
</tbody>
</table>
**Left factoring**

- Graphically

```
# Production rule
1  A → a α1
2  a → A α2
3  α → A α3
```

**Expression example**

```
# Production rule
1  factor → identifier
2  ( identifier ) expr
3  identifier ( expr )
```

After left factoring:

```
# Production rule
1  factor → identifier post
2  post → ( expr )
3  ( expr )
```

In this form, it has LL(1) property

**Left factoring**

- Graphically

```
factor

---
No basis for choice

---
Next word determines choice
```

**Question**

Using left factoring and left recursion elimination, can we turn an arbitrary CFG to a form where it meets the LL(1) condition?

**Answer**

Given a CFG that does not meet LL(1) condition, it is undecidable whether or not an LL(1) grammar exists.

**Example**

\[
\{a^n b^n | n \geq 1\} \cup \{a^n b^{2n} | n \geq 1\}
\]

has no LL(1) grammar

**Limits of LL(1)**

- No LL(1) grammar for this language:

\[
\{a^n b^n | n \geq 1\} \cup \{a^n b^{2n} | n \geq 1\}
\] has no LL(1) grammar

```
# Production rule
1  G → a A b
2  a → A b
3  B → a A
4  A → G
5  b
6  a
```

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group

**Recursive descent parsing**

- Massage grammar to have LL(1) condition
  - Remove left recursion
  - Left factor, where possible
  - Build FIRST (and FOLLOW) sets
  - Define a procedure for each non-terminal
    - Implement a case for each right-hand side
    - Call procedures as needed for non-terminals
    - Add extra code, as needed
  - Can we automate this process?
FIRST and FOLLOW sets

**FIRST(α)**
For some $α \in (T \cup NT)^*$, define $FIRST(α)$ as the set of tokens that appear as the first symbol in some string that derives from $α$.
That is, $x \in FIRST(α)$ iff $α \Rightarrow x \gamma$, for some $γ$.

**FOLLOW(A)**
For some $A \in NT$, define $FOLLOW(A)$ as the set of symbols that can occur immediately after $A$ in a valid sentence.

$FOLLOW(G) = \{EOF\}$, where $G$ is the start symbol.

### Computing FIRST sets

**Idea:**
Use FIRST sets of the right side of production

**Cases:**
- $FIRST(A → B) = FIRST(B_1)$
- What does $FIRST(B_1)$ mean?
- Union of $FIRST(B_1 \rightarrow γ)$ for all $γ$
- What if $ε$ in $FIRST(B_1)$?
  - $FIRST(A → B) = \{ε\}$
  - Why +=?
  - leave $\{ε\}$ for later

**Algorithm**

- For one production: $p = A → β$
  - if ($β$ is a terminal $t$)
  - $FIRST(p) = \{t\}$
  - else if ($β = ε$)
  - $FIRST(p) = \{ε\}$
  - else
  - Given $β = B_1 B_2 B_3 ... B_k$
  - $i = 0$
  - do {
    - $i = i + 1$
    - $FIRST(p) = FIRST(p) + \{ε\}$
  } while ($ε$ in $FIRST(B_i)$ & $i < k$)
  - if ($ε$ in $FIRST(B_i)$ & $i = k$) $FIRST(p) += \{ε\}$

**Solution**

- Start with $FIRST(B)$ empty
- Compute $FIRST(A)$ using empty $FIRST(B)$
- Now go back and compute $FIRST(B)$
- What if it’s no longer empty?
- Then we recompute $FIRST(A)$
- What if new $FIRST(A)$ is different from old $FIRST(A)$?
- Then we recompute $FIRST(B)$ again...

**When do we stop?**
- When no more changes occur – called convergence
- $FIRST(A)$ and $FIRST(B)$ both satisfy equations
  - This is another fixpoint algorithm

**Using fixpoints:**

- $forall p \ FIRST(p) = {}$
- while ($FIRST sets are changing$)
  - pick a random $p$
  - compute $FIRST(p)$
- Can we be smarter?
  - Yes, visit in special order
  - Reverse post-order depth first search
  - Visiting all children (all right-hand sides) before visiting the left-hand side, whenever possible
Computing FOLLOW Sets

Example

**FOLLOW**

<table>
<thead>
<tr>
<th>#</th>
<th>Production rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>goal -&gt; expr</td>
</tr>
<tr>
<td>2</td>
<td>expr -&gt; term expr2</td>
</tr>
<tr>
<td>3</td>
<td>expr2 -&gt; + term expr2</td>
</tr>
<tr>
<td>4</td>
<td>+ term expr2</td>
</tr>
<tr>
<td>5</td>
<td>e</td>
</tr>
<tr>
<td>6</td>
<td>term -&gt; factor term2</td>
</tr>
<tr>
<td>7</td>
<td>term2 -&gt; * factor term2</td>
</tr>
<tr>
<td>8</td>
<td>/ factor term2</td>
</tr>
<tr>
<td>9</td>
<td>e</td>
</tr>
<tr>
<td>10</td>
<td>factor -&gt; number</td>
</tr>
<tr>
<td>11</td>
<td>identifier</td>
</tr>
</tbody>
</table>

**Computing FOLLOW sets**

- Idea: Push FOLLOW sets down, use FIRST where needed
- Cases: \(A \rightarrow B_1 B_2 B_3 B_4 \ldots B_k\)
  - What is \(\text{FOLLOW}(B_i)\)?
    - \(\text{FOLLOW}(B_i) = \text{FIRST}(B_{i+1})\)
  - What about \(\text{FOLLOW}(B_{k-1})\)?
    - \(\text{FOLLOW}(B_{k-1}) = \text{FOLLOW}(A)\)
  - What if \(e \in \text{FOLLOW}(B_i)\)?
    - \(\Rightarrow \text{FOLLOW}(B_{i-1}) \cup \text{FOLLOW}(A)\) extends to \(k-2\), etc.

**Generating a top-down parser**

- Two pieces:
  - Select the right RHS
  - Satisfy each part
- First piece:
  - \(\text{FIRST}()\) for each rule
  - Mapping:
    - \(NT \rightarrow \Sigma \rightarrow \text{rule#}\)
    - Look familiar?

\(\text{FIRST}(\{\}) = \{ +, -, EOF \}\)
\(\text{FIRST}(\{+, -, \}) = \{+, -, EOF\}\)
\(\text{FIRST}(\{\}) = \{+, -, EOF\}\)
\(\text{FIRST}(\{\}) = \{+, -, EOF\}\)
Generating a top-down parser

- Second piece
  - Keep track of progress
  - Like a depth-first search
  - Use a stack

- Idea:
  - Push Goal on stack
  - Pop stack:
    - Match terminal symbol, or
    - Apply NT mapping, push RHS on stack

Table-driven approach

- Encode mapping in a table
- Row for each non-terminal
- Column for each terminal symbol
- Table[NT, symbol] = rule#
  if symbol ∈ FIRST+(NT -> rhs(#))

 Parsing

- Where are we?
  - Top-down parsers
  - LL(1) property
  - Automatic, table-driven parsers
- Next: bottom-up parsers
- Why?
  - More powerful
  - Widely used – yacc, bison, JavaCUP

Next time

- Bottom up parsing

Left factoring

- Algorithm:
  \[ A \in NT, \]
  find the longest prefix \( \alpha \) that occurs in two
  or more right-hand sides of \( A \)
  if \( \alpha \neq \varepsilon \) then replace all of the \( A \) productions,
  \[ A \rightarrow a \beta_1 | a \beta_2 | \ldots | a \beta_n | \gamma, \]
  with
  \[ A \rightarrow \alpha Z | \gamma, \]
  \[ Z \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n, \]
  where \( Z \) is a new element of NT
  Repeat until no common prefixes remain

Code

- push the start symbol, \( \sigma \), onto Stack
- top ← top of Stack
- loop forever
  - if top = EOF and token = EOF then break & report success
  - if top is a terminal then
    - if top matches token then
      pop Stack // recognized top
      token ← next_token()
    - else   // top is a non-terminal
      if TABLE[top, token] is A
        pop Stack // get rid of A
        push Bk, Bk-1, ..., B1  // in that order
      else      // top is a non-terminal
        if TABLE[top, token] is A = B1B2...Bk
          push Bk, Bk-1, ..., B1  // in that order
        top ← top of Stack
  - Missing else’s for error conditions