Syntax Analysis – Part III
Top-Down Parsers

EECS 483 – Lecture 6
University of Michigan
Wednesday, January 30, 2008

Reading + Announcements

- Today
  - 4.3, 4.4 Dragon book
  - Read over stuff in 4.3 as we will go over it very fast
- Next Time
  - 4.5 Dragon book
From Last Time: Predictive Parsing

- LL(1) grammar:
  - For a given non-terminal, the lookahead symbol uniquely determines the production to apply
  - Top-down parsing = predictive parsing
  - Driven by predictive parsing table of
    - non-terminals x terminals → productions

---

From Last Time: Parsing with Table

<table>
<thead>
<tr>
<th>Partly-derived String</th>
<th>Lookahead</th>
<th>parsed part</th>
<th>unparsed part</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → ES'</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(S)S'</td>
<td>1</td>
<td>(1+2+(3+4))+5</td>
<td></td>
</tr>
<tr>
<td>(ES')S'</td>
<td>1</td>
<td>(1+2+(3+4))+5</td>
<td></td>
</tr>
<tr>
<td>(1S')S'</td>
<td>+</td>
<td>(1+2+(3+4))+5</td>
<td></td>
</tr>
<tr>
<td>(1+ES')S'</td>
<td>2</td>
<td>(1+2+(3+4))+5</td>
<td></td>
</tr>
<tr>
<td>(1+2S')S'</td>
<td>+</td>
<td>(1+2+(3+4))+5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>num</th>
<th>+</th>
<th>( )</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → ES'</td>
<td>→ ES'</td>
<td>→ ( )</td>
<td>→ $</td>
</tr>
<tr>
<td>S' → +S</td>
<td>→ E</td>
<td>→ E</td>
<td></td>
</tr>
<tr>
<td>E → num</td>
<td>→ (S)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to Construct Parsing Tables?

Needed: Algorithm for automatically generating a predictive parse table from a grammar

Constructing Parse Tables

- Can construct predictive parser if:
  - For every non-terminal, every lookahead symbol can be handled by at most 1 production
  - FIRST(β) for an arbitrary string of terminals and non-terminals β is:
    - Set of symbols that might begin the fully expanded version of β
  - FOLLOW(X) for a non-terminal X is:
    - Set of symbols that might follow the derivation of X in the input stream
Parse Table Entries

- Consider a production \( X \to \beta \)
- Add \( \to \beta \) to the \( X \) row for each symbol in \( \text{FIRST}(\beta) \)
- If \( \beta \) can derive \( \varepsilon \) (\( \beta \) is nullable), add \( \to \beta \) for each symbol in \( \text{FOLLOW}(X) \)
- Grammar is LL(1) if no conflicting entries

<table>
<thead>
<tr>
<th></th>
<th>num</th>
<th>+</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>ES'</td>
<td></td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>S'</td>
<td>$</td>
<td>ES'</td>
<td>ES'</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S'</td>
<td>+S</td>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>E</td>
<td>num</td>
<td>(</td>
<td>)</td>
<td></td>
<td>(S)</td>
</tr>
</tbody>
</table>

Computing Nullable

- \( X \) is nullable if it can derive the empty string:
  - If it derives \( \varepsilon \) directly (\( X \to \varepsilon \))
  - If it has a production \( X \to YZ \ldots \) where all RHS symbols (\( Y,Z \)) are nullable
- Algorithm: assume all non-terminals are non-nullable, apply rules repeatedly until no change

\[
\begin{align*}
S & \rightarrow ES' \\
S' & \rightarrow \varepsilon | +S \\
E & \rightarrow \text{number} | (S)
\end{align*}
\]

Only \( S' \) is nullable
Computing FIRST

- Determining FIRST(X)

1. If X is a terminal, then add X to FIRST(X)
2. If X → ε then add ε to FIRST(X)
3. If X is a nonterminal and X → Y₁Y₂...Yₖ then a is in FIRST(X) if a is in FIRST(Yᵢ) and ε is in FIRST(Yⱼ) for j = 1...i-1 (i.e., it's possible to have an empty prefix Y₁ ... Yi-1)
4. If ε is in FIRST(Y₁Y₂...Yₖ) then ε is in FIRST(X)

FIRST Example

S → ES'
S' → ε | +S
E → number | (S)

Apply rule 1: FIRST(num) = {num}, FIRST(+) = {+}, etc.
Apply rule 2: FIRST(S') = {ε}
Apply rule 3: FIRST(S) = FIRST(E) = {}
FIRST(S') = FIRST(‘+’) + {ε} = {ε, +}
FIRST(E) = FIRST(num) + FIRST(‘(‘) = {num, (}

Rule 3 again: FIRST(S) = FIRST(E) = {num, (}
FIRST(S') = {ε, +}
FIRST(E) = {num, (}
Computing FOLLOW

- Determining FOLLOW(X)
  1. if S is the start symbol then $ is in FOLLOW(S)
  2. if A → αBβ then add all FIRST(β) != ε to FOLLOW(B)
  3. if A → αB or αBβ and ε is in FIRST(β) then add FOLLOW(A) to FOLLOW(B)

FOLLOW Example

S → ES’
S’ → ε | +S
E → number | (S)

FIRST(S) = {num, ( }
FIRST(S’) = {ε, + }
FIRST(E) = { num, ( }

Apply rule 1: FOL(S) = {$}
Apply rule 2: S → ES’ FOL(E) += {FIRST(S’) - ε} = {+}
S’ → ε | +S -
E → num | (S) FOL(S) += {FIRST(‘)’ - ε} = {$,} }
Apply rule 3: S → ES’ FOL(E) += FOL(S) = {+,$,}
(because S’ is nullable)
FOL(S’) += FOL(S) = {$,}
Putting it all Together

\[
\begin{align*}
\text{FIRST}(S) &= \{\text{num, ( }\} & \text{FOLLOW}(S) &= \{\$, )\} \\
\text{FIRST}(S'') &= \{\varepsilon, +\} & \text{FOLLOW}(S'') &= \{\$, )\} \\
\text{FIRST}(E) &= \{\text{num, ( }\} & \text{FOLLOW}(E) &= \{\varepsilon, +, $\}
\end{align*}
\]

- Consider a production \( X \rightarrow \beta \)
- Add \( \Rightarrow \beta \) to the \( X \) row for each symbol in \( \text{FIRST}(\beta) \)
- If \( \beta \) can derive \( \varepsilon \) (\( \beta \) is nullable), add \( \Rightarrow \) for each symbol in \( \text{FOLLOW}(X) \)

| \( S \rightarrow ES' \) | \( S' \rightarrow \varepsilon | +S \) | \( E \rightarrow \text{number } | (S) \) |
|-----------------|-----------------|-----------------|
| num            | +               | (               | $              |
| \( S \)         | \( ES' \)       | \( ES' \)       |
| \( S' \)        | \(+S\)          | \( \varepsilon \) |
| \( E \)         | num             | (               | \( S \)        |

Ambiguous Grammars

Construction of predictive parse table for ambiguous grammar results in conflicts in the table (i.e., 2 or more productions to apply in same cell)

\[
S \rightarrow S + S \ | \ S \ast S \ | \ \text{num}
\]

\[
\text{FIRST}(S+S) = \text{FIRST}(S*S) = \text{FIRST}(\text{num}) = \{\ \text{num}\}\]
Class Problem

\[
\begin{align*}
E & \rightarrow E + T | T \\
T & \rightarrow T * F | F \\
F & \rightarrow (E) | \text{num} | \varepsilon
\end{align*}
\]

1. Compute FIRST and FOLLOW sets for this G
2. Compute parse table entries

Top-Down Parsing Up to This Point

- Now we know
  - How to build parsing table for an LL(1) grammar (i.e. FIRST/FOLLOW)
  - How to construct recursive-descent parser from parsing table
  - Call tree = parse tree
- Open question – Can we generate the AST?
Creating the Abstract Syntax Tree

- Some class definitions to assist with AST construction
- Class Hierarchy

```java
class Expr {}

class Add extends Expr {
    Expr left, right;
    Add(Expr L, Expr R) {
        left = L; right = R;
    }
}

class Num extends Expr {
    int value;
    Num(int v) {value = v;}
}
```

Creating the AST

- We got the parse tree from the call tree
- Just add code to each parsing routine to create the appropriate nodes
- Works because parse tree and call tree are the same shape, and AST is just a compressed form of the parse tree
**AST Creation: parse_E**

```
Expr parse_E() {
  switch (token) {
    case num: // E -> number
      Expr result = Num(token.value);
      token = input.read(); return result;
    case '(': // (S)
      token = input.read();
      Expr left = parse_E();
      Expr right = parse_S();
      if (right != NULL) return left;
      else return new Add(left, right);
    default: ParseError();
  }
}
```

**AST Creation: parse_S**

```
Expr parse_S() {
  switch (token) {
    case num:
    case '(': // S -> ES'
      Expr left = parse_E();
      Expr right = parse_S();
      if (right != NULL) return left;
      else return new Add(left, right);
    default: ParseError();
  }
}
```
Grammars

- Have been using grammar for language “sums with parentheses” \((1+2+(3+4))+5\)
- Started with simple, right-associative grammar
  - \(S \rightarrow E + S \mid E\)
  - \(E \rightarrow \text{num} \mid (S)\)
- Transformed it to an LL(1) by left factoring:
  - \(S \rightarrow ES'\)
  - \(S' \rightarrow \varepsilon \mid +S\)
  - \(E \rightarrow \text{num} \mid (S)\)
- What if we start with a left-associative grammar?
  - \(S \rightarrow S + E \mid E\)
  - \(E \rightarrow \text{num} \mid (S)\)

Reminder: Left vs Right Associativity

Consider a simpler string on a simpler grammar: “1 + 2 + 3 + 4”

**Right recursion : right associative**

\[
\begin{align*}
S & \rightarrow E + S \\
S & \rightarrow E \\
E & \rightarrow \text{num}
\end{align*}
\]

**Left recursion : left associative**

\[
\begin{align*}
S & \rightarrow S + E \\
S & \rightarrow E \\
E & \rightarrow \text{num}
\end{align*}
\]
### Left Recursion

$ S \rightarrow S + E$

$ S \rightarrow E$

$ E \rightarrow \text{num}$

#### Derived string | Lookahead | Read/Unread
--- | --- | ---
$ S $ | 1 | 1+2+3+4
$ S+E $ | 1 | 1+2+3+4
$ S+E+E $ | 1 | 1+2+3+4
$ S+E+E+E $ | 1 | 1+2+3+4
$ E+E+E+E $ | 1 | 1+2+3+4
$ 1+E+E+E $ | 2 | 1+2+3+4
$ 1+2+E+E $ | 3 | 1+2+3+4
$ 1+2+3+E $ | 4 | 1+2+3+4
$ 1+2+3+4 $ | 5 | 1+2+3+4

Is this right? If not, what’s the problem?

---

### Left-Recursive Grammars

- Left-recursive grammars don’t work with top-down parsers: we don’t know when to stop the recursion
- Left-recursive grammars are NOT LL(1)!
  - $ S \rightarrow S\alpha $
  - $ S \rightarrow \beta $
- In parse table
  - Both productions will appear in the predictive table at row $ S $ in all the columns corresponding to $ \text{FIRST}(\beta) $
Eliminate Left Recursion

- Replace
  - $X \rightarrow X\alpha_1 | ... | X\alpha_m$
  - $X \rightarrow \beta_1 | ... | \beta_n$
- With
  - $X \rightarrow \beta_1X' | ... | \beta_nX'$
  - $X' \rightarrow \alpha_1X' | ... | \alpha_mX' | \epsilon$
- See complete algorithm in Dragon book

Class Problem

Transform the following grammar to eliminate left recursion:

- $E \rightarrow E + T | T$
- $T \rightarrow T * F | F$
- $F \rightarrow (E) | \text{num}$
Creating an LL(1) Grammar

- Start with a left-recursive grammar
  - S → S + E
  - S → E
  - and apply left-recursion elimination algorithm
    - S → ES'
    - S' → +ES' | ε
- Start with a right-recursive grammar
  - S → E + S
  - S → E
  - and apply left-factoring to eliminate common prefixes
    - S → ES'
    - S' → +S | ε

Top-Down Parsing Summary

Language grammar → Left-recursion elimination
  → Left factoring
  → LL(1) grammar
  → predictive parsing table
  → FIRST, FOLLOW
  → recursive-descent parser
  → parser with AST gen
Next Topic: Bottom-Up Parsing

- A more powerful parsing technology
- LR grammars – more expressive than LL
  - Construct right-most derivation of program
  - Left-recursive grammars, virtually all programming languages are left-recursive
    - Easier to express syntax
- Shift-reduce parsers
  - Parsers for LR grammars
  - Automatic parser generators (yacc, bison)