Class 17: Predictive Parsing (1)

Held: Monday, 3 October 2011

Summary: Today we consider how to hand-code a parser based on a grammar. The traditional technique for doing such coding is called predictive parsing.

Related Pages:

- EBoard.
- Reading: Aho et al. 4.4.
- Due: Lab: Yacc/Bison.

Notes:

- Would you prefer the midsem (in-class) on the Friday before break or the Monday after break or the Friday after break?
- There seem to be some questions on the Yacc/Bison lab.
- Coming Wednesday: Project Phase 2: Lexical Analysis.
- It sounds like there are some questions on the Yacc lab.
- EC for Thursday’s convo on the Future of the Book.
- EC for Thursday’s Thursday Extra on Computer Vision.

Overview:

- Introduction to Parsing.
- Basics of Predictive Parsing.
- An Example: Language Membership.
- Some Problems with the Technique.

Parsing: Some Basics

- We’ve seen how to use grammars to describe languages.
- Now we have a different problem: How do you build a parse tree for a string?
  - So that we can verify that it’s in the language.
  - So that we can use the parse tree for understanding the intended structure of the program.
- There are many different dimensions you can consider when writing a parser.
  - Do you hand code it or do you rely on a parser generator?
  - Do you build the parse tree by gradually expanding the start symbol (top down) or by combining parts of the string (bottom up)?
  - Do you process the nonterminals in the input string in a particular order (e.g., left-to-right or right-to-left or ...?)
Predictive Parsing

- Predictive parsing is the most common technique for hand-generated parsers.
- Predictive parsers are top-down, left-to-right parsers.
- In predictive parsing, you repeatedly expand the leftmost nonterminal.
- Predictive parsers are also called recursive-descent parsers because of the way they recursively descend through the tree.
- How do you decide how to expand the nonterminal (that is, which production to apply?). In the simplest case,
  - You know which nonterminal you are expanding.
  - You look at the next token.
  - You look at the right-hand-sides
  - If you’re lucky, only one right-hand-side begins with that token.

- Here’s a simple language for “palindromes of a’s and b’s with a c in the middle”.

S ::= a S a
    | b S b
    | c

- Here’s the corresponding set of procedures in pseudo-Java.

```java
boolean parseS(Lexer l) {
    Token t = l.nextToken(); // Consumes the token
    if (t is an a) {
        if (!parseS(l) return false);
        Token u = l.nextToken();
        return (u is an a);
    }
    else if (t is a b) {
        if (!parseS(l) return false);
        Token u = l.nextToken();
        return (u is an b);
    }
    else {
        return t is a c;
    }
} // parseS
```

- However, subtle changes to the grammar can make it much more difficult to parse this way.
- What if we didn’t have the middle c? E.g.,

S ::= a
S ::= b

- Then there are essentially two choices when we see an a
  - If it’s a “left a”, we should apply S ::= a S a
  - If it’s a “singleton a”, we should apply S ::= a
- Hence, this grammar is not immediately amenable to predictive parsing.
- There are other potential problems. For example, what if we rewrite the grammar as
S ::= A
   | B
   | c
A ::= a S a
B ::= b S b

- How do we decide which rule to apply when we see a symbol, given that two of the right-hand sides begin with a nonterminal?
- Finally, what do we do about epsilon productions, such as the following?
  
  S ::= epsilon

  ○ Since there's no symbol, how do you know when to use the rule?

Building a Parse Tree

- Note that we can use the predictive parsing technique to do more than just determine whether a string is in a language. We can extend the technique to return a parse tree.
- We may need to create different trees depending on the type of derivation used.
- Instead of returning false when we fail to parse, we'll throw an exception.
- Here's some sample code

```java
Palindrome parseS(Lexer l) throws ParseException {
    Token t = l.nextToken();
    if (t is an a) {
        Palindrome p = parseS(l);
        Token u = l.nextToken();
        if (u is an a) {
            return new Palindrome(a, p, a);
        } else {
            throw new ParseException("Failed to find matching a; saw a " + u);
        }
    } else if (t is a b) {
        Palindrome p = parseS(l);
        Token u = l.nextToken();
        if (u is an b) {
            return new Palindrome(b, p, b);
        } else {
            throw new ParseException("Failed to find matching b, saw a " + u);
        }
    } else if (t is a c) {
        return new SingletonPalindrome(c);
    } else {
        throw new ParseException("Found unexpected token: " + t);
    }
} // parseS
```
Analyzing the Grammar

- As you can tell, we need to make a number of decisions.
- Given multiple right-hand sides that start with a nonterminal, which one do you choose?
- If a nonterminal derives epsilon, how do you decide to apply that derivation?
- What if a nonterminal derives a nonterminal that derives epsilon? Consider what to do when you see a b when matching an S in

  \[
  \begin{align*}
  S &::= X b \\
  X &::= x X \\
  &\mid \text{epsilon}
  \end{align*}
  \]

- All of these analyses are aided by three tables (and corresponding functions) known as **First** (and **first**), **Follow**, and **Nullable** (and **nullable**).
- The tables and functions are recursively constructed. That is, the functions may be defined in terms of the tables and the tables may be defined in terms of the current versions of the functions.
- **First** maps nonterminals to sets of tokens that can begin strings derived from those nonterminals.
  - **first** maps sequences of nonterminals and terminals to the symbols that can begin strings derived from those sequences
- **Follow** maps nonterminals to sets of tokens that can follow those nonterminals in sequences derivable in the grammar
- **Nullable** indicates which nonterminals can derive the empty string
  - **nullable** operates on sequences of symbols (nonterminals and terminals), returning true only when the sequence can derive epsilon
- We’ll see how to construct these three tables in the next class.