CS107 Handout 41
Autumn 2000 November 14, 2000

RSG in Java

Julie Zelenski's handout.

As promised, a chance to view a solution to the same problem (the RSG) in yet another language, this time in Java. The RSG is a particularly good candidate program to use as our reference program since it stresses a language's facilities for string manipulation, file processing, and managing arbitrary collections, which are pretty useful features in any language.

Here is one possible object-oriented solution for the RSG in Java. We use a `Vector` of `String` objects to represent each production. We create a `Definition` class that stores a `String` nonterminal and its `Vector` of productions. Note that the `Definition` class takes responsibility for all operations that manipulate a definition— reading it from the file, expanding it out, etc. The `Grammar` object consists of a `Hashtable` of `Definition` objects and it likewise takes responsibility for the grammar’s operations for reading and expanding. The `Definitions` are stored into the table using the non-terminal `String` as the key for quick lookup of non-terminals later.

This version is practically identical in functionality to the version you wrote for `hw1c`. It parses in the very same grammar files, prints out three random expansions and exits. Designing the RSG in Java is quite nice due to the expressive power of the built-in classes (i.e. having equivalents of the `Scanner`, `DArray`, and `Hashtable` already written is great!) and writing it is much easy to debug because of all of the runtime safety features (bounds-checking, `null` pointer exceptions, runtime detection of invalid casts, etc.). It's probably not that much shorter (counting lines of code) than a C solution, but it was much less time-consuming to develop.

Getting used to objects is pretty straightforward, but at the beginning it can seem a little funky to be sending messages to do everything— even printing, getting a random number, comparing two strings, and so on are all accomplished by sending messages.

RSG.java

```
/*
 * RSG.java, the main class for this Java program
 * ----------------------------------------------
 * The static method "main" is the one that will be called to start execution.
 * We do very little here, just create and set up a new Grammar object and have
 * it generate random sentences.
 */

public class RSG {

   public static void main(String args[])
   {
      Grammar g = new Grammar("Grammars/" + args[0]);
      // grammar file is first argument

      for (int i = 1; i <= 3; i++) {
         System.out.println("Version #" + i);
         g.printRandomSentence();
      }
   }

}
```

Grammar.java

```
/* Grammar class
 * -------------
 * This class encapsulates the Grammar data.  It's mostly just a
 * Hashtable of Definition objects and not much more.  The Grammar
 * object knows how to read itself from a data file and then print
 * random sentences through a recursive expansion process.
 */

import java.util.Hashtable;   // to get access to shorthand name

public class Grammar {

   private Hashtable definitions;                     // just one instance variable

   private static final String StartDefinition = "{"; // class constant


   /* Grammar constructor
    * -------------------
    * Creates a Grammar object, reading the data from the file specified by name
    * to the constructor.
    */
   public Grammar (String grammarFile)
   {
      definitions = new Hashtable();            // create empty Hashtable
      readDefinitions(new Scanner(grammarFile));// read definitions using Scanner
   }
   /*
    * readDefinitions helper method (private, since not for outside use)
    * -----------------------------
    * Used by the Grammar constructor to repeatedly read definitions
    * from the Scanner and add to the hashtable.  We look for the
    * opening brace which signifies a new definition, and then pass
    * it off to the Definition constructor which knows how to read
    * a Definition and all its productions using the Scanner.
    * We repeatedly do this until we get to EOF (signalled by
    * a null return from getNextToken).
    */

   private void readDefinitions(Scanner scanner)
   {
      String token;

      while ((token = scanner.getNextToken()) != null) {    // read til EOF
         if (token.equals(StartDefinition)) {         // opening of definition
            Definition def = new Definition(scanner); // ctor reads def from Scanner
            definitions.put(def.getNonTerminalName(), def); // add to table
         }
      }
   }


   /*
    * printRandomSentence method
    * ---------------------------
    * The public method used to create and print a new random sentence.
    * Nothing special, just start expanding from the non-terminal "<start>"
    */

   public void printRandomSentence()
   {
      printExpansion("<start>");
      System.out.println("\n");           // end sentence with newline
   }

   /*
    * printExpansion method
    * ----------------------
    * Used during expansion phase.  Given a string, we decide whether it's
    * a terminal (in which case we just print it) or a non-terminal, in
    * which case we look it up in our table of definitions, and ask the
    * the definition to expand itself.
    */

   public void printExpansion(String s)
   {
      if (s.charAt(0) != '<') {           // this is a terminal, just print it
         System.out.print(" " + s);
      } else {
         Definition def = (Definition) definitions.get(s);  // look up in table
         def.printExpansion(this);        // tell Definition to expand itself
      }
   }
}
```

Definition.java

```
/*
 * Definition class
 * ----------------
 * This class simply gathers together the nonterminal (String) and its
 * lists of possible productions (in a Vector). Its constructor knows how to
 * read a Definition from file in correct format and the object itself takes
 * responsibility for expanding itself when generating random sentences
 * with the printExpansion method.
 */

import java.util.*;

public class Definition {

   private String nonterm; // Just two instance variables, both private
   private Vector productions;

   private static final String EndDefinition = "}";      // class constants
   private static final String EndProduction = ";";


   /* Definition constructor
    * ----------------------
    * Creates a Definition object, reading the data from the Scanner object
    * passed to the constructor.  We assume the Scanner has just passed the '{'
    * character which opens the definition and we are reading to pull off the
    * non-terminal name that follows it as the next scanner token.
    */
   public Definition(Scanner scanner)
   {
      productions = new Vector();      // create an empty list of productions
      nonterm = scanner.getNextToken();      // read non-term first
      while (readOneProduction(scanner))     // read productions until no more
         ;
   }
   /*
    * readOneProduction helper method
    * -------------------------------
    * Used by the constructor to read one single production
    * into a new Vector and then store it in the Vector of all
    * productions.  Returns a boolean result which indicates
    * whether we are at the end of processing productions.
    */

   private boolean readOneProduction(Scanner scanner)
   {
      Vector production = new Vector();

      while (true) {
         String word = scanner.getNextToken();
         if (word.equals(EndDefinition))     // if at end of entire definition
             return false;             // no more productions, we're done
         else if (word.equals(EndProduction))  { // if at end of production
            productions.addElement(production);
            return true;
         } else
            production.addElement(word);
      }
   }

   /*
    * getNonTerminalName accessor method
    * ----------------------------------
    * Accessor for the nonterm String name private variable.
    */

   public String getNonTerminalName()
   {
      return nonterm;
   }

   /*
    * printExpansion method
    * ----------------------
    * Used to expand a Definition by choosing a production at
    * random and iterating over its component strings, expanding those.
    */

   public void printExpansion(Grammar g)
   {
      int choice = (int) (Math.random() * productions.size());
      Vector chosenOne = (Vector)(productions.elementAt(choice));
      Enumeration e = chosenOne.elements();
      while (e.hasMoreElements())      // iterate over words, expand each
         g.printExpansion((String)e.nextElement());
   }
}
```