

---

## Reading

---

1. Mitchell, Chapter 11.

---

## Problems

---

1. (40 points) ..... Random Sentence Generator

The goals of this problem are to:

- (a) use parameterized data structures,
- (b) implement a simple class hierarchy following the *Composite Design Pattern*, and
- (c) write a fairly entertaining program.

**Random Sentence Generator** The “Random Sentence Generator” creates random sentences from a grammar. Here are a few examples of the output for generating homework extension requests:

- *Wear down the Professor’s patience:* I need an extension because I used up all my paper and then my dorm burned down and then I didn’t know I was in this class and then I lost my mind and then my karma wasn’t good last week and on top of that my dog ate my notes and as if that wasn’t enough I had to finish my doctoral thesis this week and then I had to do laundry and on top of that my karma wasn’t good last week and on top of that I just didn’t feel like working and then I skied into a tree and then I got stuck in a blizzard on Mt. Greylock and as if that wasn’t enough I thought I already graduated and as if that wasn’t enough I lost my mind and in addition I spent all weekend hung-over and then I had to go to the Winter Olympics this week and on top of that all my pencils broke.
- *Plead innocence:* I need an extension because I forgot it would require work and then I didn’t know I was in this class.
- *Honesty:* I need an extension because I just didn’t feel like working.

**Grammars** The program reads in grammars written in a form illustrated by this simple grammar file to generate poems:

```
<start> = The <object> <verb> tonight
;

<object> =
    waves
    | big yellow flowers
    | slugs
;

<verb> =
    sigh <adverb>
    | portend like <object>
```

```

| die <adverb>
;

<adverb> =
  warily
| grumpily
;

```

The strings in brackets (<>) are the non-terminals. Each non-terminal definition is followed by a sequence of productions, separated by '|' characters, and with a ';' at the end. Each production consists of a sequence of white-space separated terminals and non-terminals. A production may be empty so that a non-terminal can expand to nothing. There will always be whitespace surrounding the '|', '=', and ';' characters to make parsing easy.

Here are two possible poems generated by generating derivations for this grammar:

```
The big yellow flowers sigh warily tonight
```

```
The slugs portend like waves tonight
```

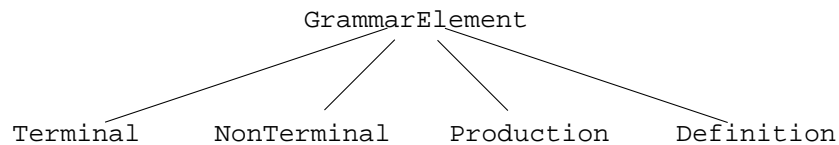
Your program will create a data structure to represent a grammar it reads in and then produce random derivations from it. Derivations will always begin with the non-terminal <start>. To expand a non-terminal, simply choose one of its productions from the grammar at random and then recursively expand each word in the production. For example:

```

<start>
-> The <object> <verb> tonight
-> The big yellow flowers <verb> tonight
-> The big yellow flowers sigh <adverb> tonight
-> The big yellow flowers sigh warily tonight

```

**System Architecture** A grammar consists of terminals, non-terminals, productions, and definitions. These four items have one thing in common: they can all be expanded into a random derivation for that part of a grammar. Thus, we will create classes organized in the following class hierarchy to store a grammar:



The abstract class GrammarElement provides the general interface to all pieces of a grammar. It is defined as follows:

```

public abstract class GrammarElement {

  /**
   * Expand the grammar element as part of a random
   * derivation. Use grammar to look up the definitions
   * of any non-terminals encountered during expansion.
   */
  public abstract void expand(Grammar grammar);
}

```

```

/**
 * Return a string representation of this grammar element.
 * This is useful for debugging. (Even though we inherit a
 * default version of toString() from the Object superclass,
 * I include it as an abstract method here to ensure that
 * all subclasses provide their own implementation.)
 */
public abstract String toString();
}

```

The Grammar object passed into `expand` is used to look up the definitions for non-terminals during the expansion process, as described next.

**The Grammar Class** A Grammar object maps non-terminal names to their definitions. At a minimum, your Grammar class should implement the following:

```

public class Grammar {

    // add a new non-terminal, with the given definition
    public void add(String nt, Definition def)

    // look up a non-terminal, and return the definition, or null
    // if not def exists.
    public Definition get(String nt)

    // expand the grammar, starting with the start symbol.
    public void expand()

    // print out the grammar. Useful for debugging.
    public String toString()
}

```

**Subclasses** The four subclasses of `GrammarElement` represent the different pieces of the grammar and describe how each part is expanded:

- **Terminal:** A terminal just stores a terminal string (like “slugs”), and a terminal expands to itself.
- **NonTerminal:** A non-terminal stores a non-terminal string (like “<start>”). When a non-terminal expands, it looks up the definition for its string and recursively expands that definition.
- **Production:** A production stores a vector of `GrammarElements`. To expand, a production simply expands each one of these elements.
- **Definition:** A definition stores a vector of `Productions`. A definition is expanded by picking a random `Production` from its vector and expanding that `Production`.

This design is an example of the *Composite Design Pattern*. The hierarchy of classes leads to an extensible design where no single `expand` method is more than a few lines long.

### Implementation Steps

- (c) Download the starter code from the handouts web page. Once compiled with `javac`, you will run the program with a command like

```
java RandomSentenceGenerator < Poem.g
```

You will need to use Java's generic library classes. In particular, you will want to use the following two classes from the `java.util` package:

- `Vector<T>`: a class to store a list of `T` objects; and
- `Hashtable<K,V>`: a class to store a map from keys of type `K` to values of type `V`.

The full documentation for these classes is accessible from the `cs334` links web page.

- (b) Begin by implementing the four subclasses of `GrammarElement`. Do not write `expand` yet, but complete the rest of the classes so that you can create and call `toString()` on them.
- (c) The next step is to parse the input to your program and build the data structure representing the grammar in `RandomSentenceGenerator.java`. The grammar will be stored in the instance variable `grammar`.

I have provided a skeleton of the parsing code. The parser uses a `java.util.Scanner` object to perform lexical analysis and break the input into individual tokens. I use the following two `Scanner` methods:

- `String next()`: Removes the next token from the input stream and returns it.
- `boolean hasNext(String pattern)`: Returns `true` if and only if the next token in the input matches the given pattern. (If `pattern` is missing, this will return `true` if there are any tokens left in the input.)

When parsing the input, it is useful to keep in mind what form the input will have. In particular, we can write an EBNF grammar for the input to your program as follows:

```
<Grammar>      ::= [ non-terminal '=' <Definition> ';' ]*
<Definition>  ::= <Production> [ '|' <Production> ]*
<Production> ::= [ word ]*
```

where `non-terminal` is a non-terminal from the grammar being read and `word` is any terminal or non-terminal from the grammar being read. Recall that the syntax `[ word ]*` matches zero or more words.

The parsing code follows this definition with the following three methods:

```
protected Grammar readGrammar(Scanner in)
protected Definition readDefinition(Scanner in)
protected Production readProduction(Scanner in)
```

Modify these methods to create appropriate `Terminal`, `NonTerminal`, `Production`, `Definition`, and `Grammar` objects for the input. You may wish to print the objects you are creating as you go to ensure the grammar is being represented properly. You will need to complete the definition of `Grammar` at this point as well.

- (d) Once the grammar can be properly created and printed, implement the `expand` methods for your `GrammarElements`. Recall that a `java.util.Random` object can be used to generate random numbers:

```
Random gen = new Random();
int number = gen.nextInt(N); // number is in range [0,N-1].
```

Change `RandomSentenceGenerator` to create and print three random derivations after printing the grammar.

- (e) Write at least one additional grammar. It can be as simple or as complicated as you like. Bonus points for creativity.
- (f) Submit your `java` code and additional grammar with `turnin` as usual, and include a printout of the code with your written homework.

A few details about producing derivations:

- The grammar will always contain a `<start>` non-terminal to begin the expansion. It will not necessarily be the first definition in the file, but it will always be defined eventually. I have provided some error checking in the parsing code, but you may assume that the grammar files are otherwise syntactically correct.
- The one error condition you should catch reasonably is the case where a non-terminal is used but not defined. It is fine to catch this when expanding the grammar and encountering the undefined non-terminal rather than attempting to check the consistency of the entire grammar while reading it. The starter code contains a
 

```
RandomSentenceGenerator.fail(String msg)
```

 method that you can call to report an error and stop.
- When generating the output, just print the terminals as you expand. Each terminal should be preceded by a space when printed, except the terminals that begin with punctuation like periods, comma, dashes, etc. You can use the `Character.isLetterOrDigit` method to check whether a character is punctuation mark. This rule about leading spaces is just a rough heuristic, because some punctuation (quotes for example) might look better with spaces. Don't worry about the minor details— we're looking for something simple that is right most of the time and it's okay if is little off for some cases.

## 2. (15 points) ..... Smalltalk and Squeak Tutorial

**Although the Squeak tutorial is not very long, you may work with a partner for this part if you wish.**

Go through the Smalltalk “BankAccount” tutorial from the handouts page on the Unix machines in lab. This tutorial guides you through defining and using a simple class in Squeak, and it should hopefully give you a feel for the designers’ vision.

A great deal more about Squeak can be found at <http://www.squeak.org>.

Here are few important details:

(a) To run Squeak, please follow these steps:

- Squeak needs to create and store a number of large files for you. To avoid filling up the student disk containing your home directory, we'll set up Squeak to use a directory on `/home/scratch`. There is already a directory on that disk with the same name as your Unix id. Change to that directory, as in:
 

```
cd /home/scratch/09abc
```
- In that directory, run the command
 

```
inisqueak
```

 Choose the “full” version (option 2) at the prompt to create an *image file* in your directory. The image file contains the environment’s code, plus any additions or modifications you have made to it. As you go through the tutorial and before you exit, you should select “save” from the “screen menu.” This will save any changes you made to the image on disk.
- You can run Squeak using your modified image in the future by going back to your Squeak directory (ie, `/home/scratch/09abc`) and running the command `squeak`.

(b) Due to differences in the various Unix window managers installed in the lab, you may need to:

- Use the middle mouse button when the tutorial refers to the right mouse button, and
- Use the right mouse button when the tutorial asks you to `Alt-Click`.

(c) If you prefer, you can download and install Squeak on a Mac and PC, using the instructions on the Squeak website.

(d) When you are done, select the “BankAccount” class in the System Browser. Once it is selected, Control-click on “BankAccount” and select “fileOut” from the pop-up menu. This will write out the file `BankAccount.st` with the contents of that class in your directory containing the image files.

You can look at its contents by opening it in Emacs. (It may look funny, because the new-line characters may appear as `^M` — this is fine.)

Use `turnin` to submit it— do not worry about including a print out. If you worked with someone, only one person needs to submit, but indicate who you worked with on your problem set.

(e) **(Include your answer to this part with your written answers)** Reflect on your experience. Do you think Squeak is the right model for interacting with your computer? Why or why not? A few sentences or short paragraph is sufficient. (Do this part individually.)

If you want to explore Smalltalk further, I highly recommend the “Morphic” tutorial at

<http://www.squeak.org/Documentation/>

It gives a nice introduction to how to extend and add graphical objects, etc. to the Squeak environment.

### 3. (15 points) ..... Loophole in Encapsulation

Mitchell, Problem 11.2

### 4. (15 points) ..... Smalltalk Run-time Structures

Mitchell, Problem 11.4

The given conversions between Cartesian and polar coordinates work for any point  $(x, y)$ , where  $x \geq 0$  and  $y > 0$ . Do not worry about points where  $x < 0$  or  $y \leq 0$ . The figure P.11.4.1 appears in page 332.

You should provide reasonable code for part (b), but you do *not* to use Squeak, although you may give it a try if you wish. A few details if you do:

- Put the classes in the “My-Stuff” category you made for the tutorial, and name the classes `MyPoint` and `MyPolarPoint` to avoid conflicts with predefined classes.
- To add Class methods, click on the “class” button to the right of the “?” button in the System Browser. Click on the “instance” button to switch back to instance methods.
- There is a missing `.` at the end of the line `x ← xCoordinate` in the book.
- Use `sqrt` and `arcTan` methods for the math operations.

### 5. (10 points) ..... Removing a Method

Mitchell, Problem 11.7

### 6. (15 points) ..... Protocol Conformance

Mitchell, Problem 11.6

(You will find it useful to answer Problem 11.7 first before working on this one.)

**7.** (10 points) ..... Subtyping and Binary Methods

Mitchell, Problem 11.8

**8.** (15 points) ..... Delegation-Based OO Languages (Bonus Question)

Mitchell, Problem 11.9