Homework 8

Due 26 April

Handout 19
CSCI 334: Spring, 2012

Reading

1. **(Required)** Mitchell, Chapters 12

Problems

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

The goals of this problem are to:

(a) write more Scala code,

(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:

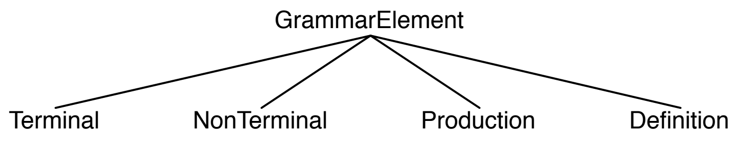
```
The big yellow flowers sigh warily tonight

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:

```
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.
   */
  def expand(grammar : Grammar) : String;

  /**
   * 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 implmementaiton.)
   */
  def toString() : String;
}
```

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:

```
class Grammar {

  // add a new non-terminal, with the given definition
  def +=(nt : String, defn : Definition)

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

  // Expand the start symbol for the grammar.
  def expand() : String

  // return a String representation of this object.
  override def toString() : String
}
```

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 list 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.

(a) Download the starter code from the handouts web page. Once compiled with scalac (or fsc), you will run the program with a command like

```
    scala RandomSentenceGenerator < Poem.g
```

You will need to use Scala's generic library classes. In particular, you will probably want to use both Lists and Maps from the standard Scala packages. 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:

i. `String next()`: Removes the next token from the input stream and returns it.
ii. `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 def readGrammar(in : Scanner): Grammar
protected def readDefinition(in : Scanner): Definition
protected def readProduction(in : Scanner): Production
```

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`. Scala provides a random number generator that can be used as follows:

```
val number = Random.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. *(30 points)* .... Assignment and Derived Classes

Mitchell, Problem 12.1

I put a working version of the code on the handouts page if you would like to experiment with it. Use `g++` to compile the program, and then run the executable `a.out` with the command "`./a.out`".

3. *(10 points)* .... Function Subtyping

Mitchell, Problem 12.3

4. *(10 points)* .... Printing

Programmers often want to print user-defined object types in a reader-friendly way to output streams (like `System.out` in Java). For example, we may wish to print a `Point` object `p` as "`(3,4)`". There are many ways to design a programming system to facilitate this.

(a) One not-so-great way is to have the stream class provide methods to print each kind of object. For example, the `OutputStream` class for whatever language we are using could be defined to have a `println(Point p)` method to facilitate writing the user-defined `Point` class to an output stream, and similar methods for other object types. What is the deficiency with this approach?

(b) In C++, this problem of writing user-defined types in a reasonable way is solved through operator overloading. C++ stream classes use the << operator to write to streams. For example, "`cout << 4`" writes the number 4 to the terminal. To define how user-defined types are written to streams, one defines a new function like the following:

```
class Point {
  int x, y;
  ...
};

ostream& operator<<(ostream& o, Point p) {
  return o << "(" << p.x << "," << p.y << ")" ;
}
```

With this definition, executing "`cout << p`", would print "`(3,4)`" if `p` were a `Point` object with those coordinates.

Java does not use operator overloading. The method

```
public void println(Object o) { ... }
```

from `java.io.PrintStream` permits one to call "`System.out.println(o)`" on any object `o` and print its representation. How does the language and library ensure that this method can print different types of objects in the appropriate way? What methods must the programmer write, how are classes structured, and what language features are involved?

(c) Could the designers have taken the same approach in C++? Would that approach fit the C++ design criteria?

5. *(30 points)* .... Generics and Traits

Consider the following trait (included in the starter code, Sets.scala):

```
trait AbstractSet[T] {
      def contains(n: T): Boolean;
      def add(n: T): AbstractSet[T];
      def remove(n: T): AbstractSet[T];

      def addAll(lst: Iterable[T]): AbstractSet[T] = {
          lst.foldLeft(this)((s, elem) => s.add(elem));
      }
      def removeAll(lst: Iterable[T]): AbstractSet[T] = {
          lst.foldLeft(this)((s, elem) => s.remove(elem));
      }
}
```

Traits are almost like Java interfaces. They can specify methods that concrete classes must implement in order to mix-in the trait. In this example, the `AbstractSet` trait has three methods that concrete set classes must implement: a method to determine membership (`contains`), a method to add a member (`add`), and a method to remove a member (`remove`). Note that traits can also be parameterized by types. In this case, the `AbstractSet` is parameterized by the type `T` of elements stored in the set.

Unlike Java interfaces, traits can also implement methods. These methods are included (for free) in any class that mixes in the trait. The `AbstractSet` trait has two such methods: a method to add an entire List of members (`addAll`) and a method to remove an entire List of members (`removeAll`). These methods are implemented in the trait itself, even though they rely on methods that have no implementation yet. (The methods `addAll` and `removeAll` are written using fold, although you need not be concerned about the details.)

(a) Write a concrete class `MutableSet` that extends the `AbstractSet` trait and implements the three methods that that trait requires. As its name suggests, your class should be mutable. That is, calling the `add` or `remove` methods on a `MutableSet` should modify the instance of the set on which the method was called, such that it now `contains` (or no longer contains), the specified element.

Note that the `AbstractSet` trait specifies that the `add` and `remove` methods must return an `AbstractSet`. In the case of `MutableSet`, it is sufficient to return the current set (`this`). This has been implemented for you in the starter code. (Remember that using the return keyword is optional; any code block automatically returns its last expression.)

There are many ways to implement `MutableSet`. This assignment is purposefully open-ended. You may want to use collections like `List[T]` or `Array[T]` from the standard library. (The standard library also includes `Set` collections, but please don't cheat and implement your `MutableSet` with a `Set` from the standard library.) Regardless of your implementation choices, the starter code includes some unit tests that your `MutableSet` should pass.

(b) Write a concrete class `ImmutableSet` that extends the `AbstractSet` trait and implements the three methods that that trait requires. As its name suggests, your class should be immutable. That is, calling the `add` or `remove` methods on an `ImmutableSet` should not modify the instance of the set on which the method was called. Instead, the method should return a new set which contains all the previous elements plus the added element (or minus the removed element). Hopefully this motivates the return type requirements of the `AbstractSet` trait.

Implementing immutable structures can be a little annoying/tricky. Here's one example that illustrates how to define multiple constructors in Scala, which may be useful to you. The following `Counter` is a "functional counter" that supports a `bump` method that returns a new counter who's value is one greater than the original counter. The class has a private default constructor that takes a parameter for the counter's initial value and a public constructor that takes no parameters. Clients may only use the public constructor, meaning all counters must start at 0. The `bump` method uses the private constructor to create a new counter object with an initial value other than 0:

```
class Counter private (val initial : Int) {

  // public constructor
  def this() = this(0);

  // return a new Counter with value one greater than mine
  def bump() : Counter = new Counter(initial + 1);

  override def toString() : String = initial.toString;
}
```

As with `MutableSet`, the starter code includes some unit tests that your `ImmutableSet` should pass.

(c) Since Scala allows method definitions in traits and sets no limit on how many traits can be mixed in, this provides for something akin to multiple inheritance. Consider the following trait:

```
trait LockingSet extends AbstractSet {
  abstract override def add(n: Int): AbstractSet = synchronized { super.add(n) }
  abstract override def remove(n: Int): AbstractSet = synchronized { super.remove(n) }
  abstract override def contains(n: Int): Boolean = synchronized { super.contains(n) }
}
```

The `LockingSet` trait can be mixed into any `AbstractSet` to create a set that can safely be used in the presence of multiple threads. The `synchronized` keyword works as you would expect from Java: each `AbstractSet` instance has a lock that must be acquired before the synchronized method can be performed. Given this definition, a thread-safe `MutableSet` can be instantiated as follows:

```
    val s = new MutableSet with LockingSet
```

Now write a trait `LoggingSet` that extends the `AbstractSet` trait and provides logging functionality. Your trait should write to standard output (using the `println` function) each time the `contains`, `add` or `remove` methods are called.

Now a `MutableSet` that is both thread-safe and logs its usage to standard output can be instantiated as follows:

```
val s = new MutableSet with LoggingSet with LockingSet
```

Note that our definitions for `LockingSet` and `LoggingSet` are agnostic to the actual implementation of our set. These traits can just as easily and without modification be mixed into our `ImmutableSet` class.

(d) Do you expect there to be a difference between `MutableSet with LoggingSet with LockingSet` and `MutableSet with LockingSet with LoggingSet`? What might that difference be?

(e) Does it make sense to mix the `LockingSet` trait into an `ImmutableSet`? Why or why not?

(f) Please turn in your `Sets.scala` code with `turnin`.

6. (*20 points*) .... (Bonus Question) Parser combinators

The parsing code I provided for the Random Sentence Generator is an example of a standard "recursive decent" parser implementation. It serves as a nice model for how you may write a parser by hand for a simple input language. However, there are also more powerful parsing techniques.

Scala provides one approach based on Parser combinators. The Expression.scala starter code from last week's homework uses a parser based on this approach. Implement a version of RandomSentenceGenerator.scala that uses a combinator parser.

The books in the lab may provide some basic details, but also consult the web — there are many examples of Scala combinator parsers out there.

7. (*15 points*) ...... (Bonus Question) C++ Multiple Inheritance and Casts

Mitchell, Problem 12.10