Homework 9

Due Wednesday, 9 May

Handout 29
CSCI 334: Spring 2018

Notes

This homework has two types of problems:

Problems: You will turn in answers to these questions.

Pair Programming: This part involves writing both Scala and Prolog code. You are required to work with a partner for this section. I will assume that you plan to work with the same partner you worked with on the last assignment unless you email me.

Turn-In Instructions

You must turn in your work for this assignment in two parts, one for the Problems part and one for the Pair Programming part. You will be assigned two GitHub repositories.

Note that this assignment should be anonymized: neither your name nor your partner's name should appear in any of the files submitted with this assignment except the file "`collaborators.txt`" (see below).

Problems: Problem sets should be typed up using LaTeX and submitted using your `cs334_hw9_<username>` repository. For example, if your GitHub username is `dbarowy`, then your repository will be called `cs334_hw9_dbarowy`. Be sure that your work is committed and pushed to your repository by Wednesday, May 9 at 11:59pm.

If you discuss the problem set with your partner or with a study group, please be sure to include their names in a `collaborators.txt` file in your repository.

Pair Programming: Programming solutions should be typed up and submitted using your partner repository. For example, if your GitHub username is `dbarowy` and you are working with a partner whose username is `wjannen`, look for a repository called `cs334_hw9_dbarowy-wjannen` (usernames will be in alphabetical order). Be sure that your work is committed and pushed to your repository by Wednesday, May 9 at 11:59pm.

Reading

1. **(Required)** Pires, Bernardo. "Try logic programming! A gentle introduction to Prolog."
2. **(Optional)** Mitchell, Chapter 15.
3. **(As Needed)** Scala resources on course webpage

Problems

Q1. (10 points) .... Prolog Warm-Up

To do these problems, go to the SWISH Prolog interpreter website at `https://swish.swi-prolog.org/`. Alternately, download SWI-Prolog to your machine and run the `swipl` command.

In both versions of Prolog, it is customary to store program "facts" and "rules" in a separate database file which is then queried using the interpreter. For example, in SWISH, clicking the blue `Program` button where it says `Create a [Program|Notebook] here` creates a database file. Enter the following terms into the new blank Program window:

```
mood(happy,sunny).
mood(unhappy,rainy).
```

Go to `File` → `Save...` and click `Save Program` (the name does not matter). Now, on the right hand side, where you see the `?-` prompt, type:

```
mood(X,sunny).
```

and click the `Run` button. You will see `X = happy` appear in the output above.

If you use `swipl`, you will need to load your file using the `consult` command. If you edit your file be sure to reload the file using the `reconsult` command.

To submit this assignment, copy your work from the SWISH program window and save it in a file that ends in `.prolog`. You may put all of your definitions into a single file.

(a) Write a program that proves that Sawyer is jealous of Jack because Kate loves Jack instead. E.g.,

```
?- jealous(sawyer,Who).
Who = jack.
```

(b) Encode the following tree and write programs that answer the following questions.

```
                             James I
                                |
                                |
               +----------------+-----------------+
               |                                  |
            Charles I                          Elizabeth
               |                                  |
               |                                  |
    +----------+------------+                     |
    |          |            |                     |
Catherine   Charles II   James II               Sophia
                                                  |
                                                  |
                                                  |
                                               George I
```

i. Who is the grandmother of George I?
ii. Was Charles I the parent of Catherine?
iii. Who are the grandchildren of James I? (I should be able to backtrack using `;` to obtain all of the grandchildren.)

(c) Write a program that returns the last element in a list. E.g.,

```
?- myLast(X,[apple, banana, cucumber, daikon, escarole, fava]).
X = fava
```

Q2. (12 points) .... Upper and Lower Bounds on Types

Type parameters in Scala (and Java) can be given upper and lower bounds to restrict how they can be instantiated. Specifically, the type `List[_ <: C]` describes a list that stores some data of some type that is a subtype of of class `C`. In other words, the type parameter has an upper bound `C`. For example, an object of `List[_ <: Point]` is a list that contains objects which extend the `Point` class. For example, the list could be `List[Point]` or `List[ColorPoint]`, etc. Reading an element from such a list is guaranteed to return a `Point`, but writing to the list is not generally allowed. This sort of bounded type is often called an existential type because it can be interpreted as "there exists some type T such that T `<: C`."

Existential types can also have lower bound constraints. A constraint `[_ >: C]` means that the existential type must be a supertype of class `C`. For example, an object of `List[_ >: Point]` could be a `List[Point]` or `List[Any]`. (`Any` is the supertype of all types in Scala, and serves similar purposes as `Object` in Java). Reading from such a list returns objects of type `Any`, but any object of type `Point` can be added to the list.

This question asks about generic versions of a simple function that reads elements from one list and adds them to another. Here is sample non-generic code, in case this is useful reference in looking at the generic code below.

```
def addAllNonGeneric(src: MutableList, dest: MutableList) : Unit = {
  for (o <- src) {
    dest += o;
  }
}
val listOfPoints = new MutableList();
val listOfColorPoints = new MutableList();
...
addAllNonGeneric(listOfColorPoints, listOfPoints);
```

It will not compile in Scala, but gives you an idea of what we're trying to do.

(a) The simplest generic version of the `addAll` method uses an unconstrained type parameter and no bounds.

```
def addAll0[T](src: MutableList[T], dest: MutableList[T]) : Unit = {
  for (o <- src) {
    dest += o;
  }
}
```

Suppose that we declare

```
val listOfPoints : MutableList[Point] = new MutableList[Point]();
val listOfColorPoints : MutableList[ColorPoint] = new MutableList[ColorPoint]();
```

and call

```
addAll0(listOfColorPoints, listOfPoints).
```

Will this call compile or will a type error be reported at compile time? Explain briefly.

(b) With `listOfColorPoints` and `listOfPoints` defined as in the previous part of this question, will the call

```
addAll1(listOfColorPoints, listOfPoints)
```

compile, where `addAll1` is defined as follows:

```
def addAll1[T](src: MutableList[_ <: T], dest: MutableList[T]) : Unit = {
  for (o <- src) {
    dest += o;
  }
}
```

Explain briefly.

(c) With `listOfColorPoints` and `listOfPoints` defined as in the previous part of this question, will the call

```
addAll2[ColorPoint](listOfColorPoints, listOfPoints)
```

compile, where `addAll2` is defined as follows:

```
def addAll2[T](src: MutableList[T], dest: MutableList[_ >: T]) : Unit = {
  for (o <- src) {
    dest += o;
  }
}
```

Explain briefly. (The explicit instantation of `T` in the call to `addAll2` is needed because of how Scala infers types.)

(d) Suppose that your friend comes across the following general programming suggestion on the web and asks you to explain. What can you tell your friend to explain the connection between the programming advice and principles of subtyping, showing off your understanding gained from CS 334?

Get and Put Principle: If you pass a generic structure to a function:

- Use an existential type with an upper bound (ie, `[_ <: T])` when you only GET values out of the structure.
- Use an existential type with a lower bound (ie, `[_ >: T])` when you only PUT values into the structure.
- Don't use an existential type when you both GET and PUT values out of / into the structure.

Pair Programming

P1. (40 points) .... Undoable Commands

The goal of this problem is to implement the core data structures of a text editor using the Command Design Pattern.

Text Editor In essence, a text editor manages a character sequence that can be changed in response to commands issued by the user, such as inserting new text or deleting text. Typically, these commands operate on the underlying character buffer at the current position of the cursor. Thus, if the cursor is positioned at the beginning of the buffer, typing the string "moo" will cause those letters to be inserted at the start of the buffer, and so on. This question explores the internal design of a simple editor.

Most text editors involve a GUI and the user issues commands to the editor by keyboard and mouse events. For us, however, the most interesting part of a text editor's is what happens behind the scenes. Therefore, our text editor will just be a simple command line program that prompts you for edit commands. The program will print the contents of the text editor's buffer, including a "^" to indicate the current cursor position, print the prompt "?", and then wait for you to enter a command. At one point in time, this was in fact how many text editors worked — look up "ed text editor" in Wikipedia, for example (or run it on our lab machines...). The following shows one run of our editor:

```
Sample Execution                 Description
Buffer:                          Buffer is initially empty, cursor at start
        ^

? I This is a test.              Insert “This is a test.” and move cursor to
Buffer: This is a test.              immediately after inserted text
                       ^

? < 9                            Move cursor 9 characters left
Buffer: This is a test.
              ^

? >                              Move cursor 1 character right
Buffer: This is a test.
               ^

? I n't                          Insert “n't”
Buffer: This isn't a test.
                  ^

? > 3                            Move cursor 3 characters right
Buffer: This isn't a test.
                     ^

? D 4                            Delete 4 characters.
Buffer: This isn't a .
                     ^

? I cow                          Insert “cow”
Buffer: This isn't a cow.
                        ^

? Q                              Quit
```

Here is a summary of all available editor commands (including some described below). The term [num] indicates an optional number.

Command	Description
`I text`	Insert text at the current cursor, moving cursor to after the new text.
`D [num]`	Delete num characters to the right of cursor position. (If num is missing, delete 1 character.)
`< [num]`	Move the cursor num characters to the left. (If num is missing, move 1 character.)
`> [num]`	Move the cursor num characters to the right. (If num is missing, move 1 character.)
`Q`	Quit
`U`	Undo the previous edit command
`P`	Print the history of edit commands
`R`	Redo an undone edit command

I have provided a working program for all but the last three commands. Your job is to change `TextEditor` to support multiple levels of undo and redo using the Command Design Pattern.

The sample execution below shows an example that uses “`U`” (undo) and “`P`” (print history). (We'll look at “`R`” (redo) at the very end of the problem.) Notice that you can undo multiple edits, not simply the last one. To support this, the text editor must keep track of an edit command history that permits you to undo as many commands from the history as you like. Undoing all commands will lead you all the way back to the original empty buffer.

The starter code for this problem is divided into two classes:

- `Buffer`: This class manages the internal state of the editor's buffer (ie, character sequence and current cursor location), and it supports commands for getting/setting the cursor location and for inserting/deleting text. Refer to the javadoc on the handouts page for more details. You should not change this class.

```
Sample Execution          Description
Buffer:
        ^
? I Hello
Buffer: Hello
             ^
? < 2
Buffer: Hello
           ^
? D 2
Buffer: Hel
           ^
? I p
Buffer: Help
            ^
? U                       Undo the previous command.
Buffer: Hel
           ^
? I ium
Buffer: Helium
              ^
? P                       Print the command history.
History:
    [Insert "Hello"]
    [Move to 3]
    [Delete 2]
    [Insert "ium"]
Buffer: Helium
              ^
? U                       Undo the last command ([Insert "ium"]).
Buffer: Hel
           ^
? U                       Undo the last command ([Delete 2]).
Buffer: Hello
           ^
? P                       Print the command history.
History:
    [Insert "Hello"]
    [Move to 3]
Buffer: Hello
           ^
? Q
```

Figure 1: Sample run of the text editor with undo.

- `TextEditor`: This class stores a `Buffer` named `buffer`. The `processOneCommand()` method reads in a command from the user and performs the appropriate operation on `buffer` by invoking one of the following methods:
  - `protected def setCursor(loc: Int): Unit`
  - `protected def insert(text: String): Unit`
  - `protected def delete(count: Int): Unit`
  - `protected def undo(): Unit`
  - `protected def redo(): Unit`
  - `protected def printHistory(): Unit`

  These methods are all quite simple. For example, the `insert` method simply inserts the text into `buffer` and repositions the cursor:

  ```
          protected def insert(text: String) = {
              buffer.insert(text);
              buffer.setCursor(buffer.getCursor() + text.length());
          }
  ```

The `EditCommand` Class To support undo, we first change the way the `TextEditor` operates on the underlying `buffer`. Rather than changing it directly, the `TextEditor` constructs `EditCommand` objects that know how to perform the desired operations and — more importantly — know how to undo those operations. All `EditCommand` objects will be derived from the `EditCommand` abstract class:

```
abstract class EditCommand(val target: Buffer) {

  /** Perform the command on the target buffer */
  def execute(): Unit;

  /** Undo the command on the target buffer */
  def undo(): Unit;

  /** Print out what this command represents */
  def toString(): String;
}
```

Here, the `execute()` method carries out the desired operation on the `target` buffer, and `undo()` would perform the inverse operation. For example, to make insert undoable, the first step would be to define an `InsertCommand` class in a new file `InsertCommand.scala`:

```
class InsertCommand(b: Buffer, val text: String) extends EditCommand(b) {
  override def execute(): Unit = { ... }
  override def undo(): Unit = { ... }
  override def toString(): String = { ... }
}
```

The `TextEditor` would then perform code like the following inside `insert`:

```
protected def insert(text: String) = {
    val command = new InsertCommand(buffer, text);
    command.execute();
    ...
}
```

Assuming `InsertCommand` is implemented properly, the insertion would happen as before. However, the `TextEditor` can now remember that the last operation performed was the `InsertCommand` we created, and we can undo it simply by calling that object's `undo()` method. In essence, an `EditCommand` object describes one modification to a `Buffer`'s state and how to undo that modification. Supporting undo is then as simple as writing a new kind of `EditCommand` object for each type of buffer modification you support.

And of course, to implement multiple levels of undo, you need to keep track of more than just the last command object created...

Implementation Strategy I suggest tackling the implementation the following steps:

(a) Download the starter code from the handouts page. Compile the Scala files with the command `fsc *.scala` as usual. I have added some `assert` statements to the `Buffer` class to aid in debugging. The general form is

```
assert(condition, { "message" })
```

You may find it useful to add similar asserts to your own code as well.

(b) Implement `InsertCommand`, `DeleteCommand`, and `MoveCommand` subclasses of `EditCommand`. For each one, you must define: 1) `execute()`, (2) `undo()`, and (3) `toString()`. I recommend holding off on `undo()` for the moment. Change `TextEditor` to create and `execute` edit command objects appropriately.

(c) Extend `TextEditor` to remember the last command executed, and change `TextEditor`'s `undo()` method to undo that command. Go back and implement `undo` for each type of `EditCommand`.

(d) Once a single level of undo is working, extend `TextEditor` to support undoing multiple previous commands. Specifically, change `TextEditor` to maintain a history of commands that have been executed and not undone. Also implement the `printHistory()` method to aid in debugging. Your program should simply ignore undo requests if there are no commands are in the history. You are free to use any Scala libraries you like in your implementation (ie, any immutable or mutable collection class).

(e) The last task is to implement redo. Specifically, if you undo one or more commands but have not yet performed any new operations on the buffer, you can redo the commands you undid:

```
Sample Execution        Description
-----------------------------------------------
Buffer: hello
           ^
? D 1
Buffer: helo
           ^
? D 1
Buffer: hel
           ^
? U                     Undo delete of “o”
Buffer: helo
           ^
? U                     Undo delete of “l”
Buffer: hello
           ^
? R                     Redo delete of “l”
Buffer: helo
           ^
? R                     Redo delete of “o”
Buffer: hel
           ^
? I p
Buffer: help
            ^
? U                     Undo insert of “p”
Buffer: hel
           ^
? U                     Undo redone delete of “o”
Buffer: helo
           ^
```

Note that redoing undone commands is no longer possible if the buffer is changed in any way. For example, if you insert text after undoing some command E, you should no longer be able to redo command E:

```
Sample Execution        Description
-----------------------------------------------
Buffer:
        ^
? I moo
Buffer: moo
           ^
? U
Buffer:
        ^
? I hello               Change buffer after undo
Buffer: hello
             ^
? R                     Redo will have no effect
Buffer: hello
             ^
```

Also, redone commands should be able to be subsequently undone:

```
Sample Execution        Description
Buffer:
        ^

? I 334
Buffer: 334
           ^

? I cow
Buffer: 334cow
              ^

? I moo
Buffer: 334cowmoo
                 ^

? U                     Undo insert of “moo”
Buffer: 334cow
              ^

? U                     Undo insert of “cow”
Buffer: 334
           ^

? R                     Redo insert of “cow”
Buffer: 334cow
              ^

? R                     Redo insert of “moo”
Buffer: 334cowmoo
                 ^

? U                     Undo insert of “moo”
Buffer: 334cow
              ^

? U                     Undo insert of “cow”
Buffer: 334
           ^

? U                     Undo insert of “334”
Buffer:
        ^

? R                     Redo insert of “334”
Buffer: 334
           ^

? R                     Redo insert of “cow”
Buffer: 334cow
              ^

? U                     Undo insert of “cow”
Buffer: 334
           ^
```

Extend `TextEditor` to support multiple levels of redo. You should not need to change any class other than `TextEditor` to implement this feature.

(f) Turn in your code using `turnin` as in the previous homeworks.

There are many extensions that would make our editor more “realistic”. One idea is listed below as an extra credit problem. It should not require more than a few additional lines of code and really highlights the elegance and simplicity of adopting this design pattern.

P2. (30 points) .... Scheduling in Prolog

Scheduling TAs to TA help session slots is a surprisingly difficult problem. Here are some of the important facts:

- There are three possible help session slots per day: 4-6pm, 6-8pm, 8-10pm.
- Every day has at least one staffed help session slot except Sunday, which has zero.
- Only one TA may work a given help session at a time.

- The TAs are: Alexis, Bo, Chloe, Drake, Esmeralda, Flint, Gemma, Hans, Idris.
- Every TA must work once per week.

TAs are also engaged in a number of extracurricular activities which interfere with some possible schedules:

- Bo has clarinet practice all day Tuesday.
- Drake says that if he has to work two days in a row, he will quit.
- Esmeralda has track practice at 5am and has trouble staying awake past 8pm.
- Flint and Gemma have geology club meetings on Friday evenings from 6-10pm.

Write a Prolog program that generates a TA schedule. Note that there are a small number of additional commonsense constraints that you will need to add to the set of facts above.

Hint: I suggest that you start by writing a smaller version of the problem by hand (e.g., two TAs, two days, no other constraints). Once you understand the logic of the problem, scale the problem up. You may find it helpful to generate your Prolog program using another programming language like Scala.

P3. (10 points) .... Challenge Problem: Composable Commands

Here is one interesting extension to the basic Text Editor.

Most of the time, two consecutive commands of the same type are lumped together into a single command. Thus, if I type “hello” followed immediately by “ there” into an editor (such as emacs), the editor lumps them together into a single insertion command that removes all of “hello there” from the buffer when undone. Similarly, if I perform two cursor movement commands in a row, that is recorded in the undo history as a single command. Here is an example:

```
Sample Execution             Description
----------------------------------------------------------
Buffer:
        ^

? I hel
Buffer: hel
           ^

? I ium                      second insert composed with first
Buffer: helium
              ^

? P
History:
    [Insert "helium"]
Buffer: helium
              ^

? U
Buffer:
        ^

? R
Buffer: helium
              ^

? <
Buffer: helium
             ^

? < 2                        second move composed with first

Buffer: helium
           ^

? D 2
Buffer: helm
           ^

? D 1                        second delete composed with first
Buffer: hel
           ^

? P
History:
    [Insert "helium"]
    [Move to 3]
    [Delete 3]
Buffer: hel
           ^

? I p
Buffer: help
            ^

? U                          undo insert of “p”
Buffer: hel
           ^

? U                          undo composed delete command
Buffer: helium
           ^

? U                          undo composed move command
Buffer: helium
              ^

? U                          undo composed insert of “helium”
Buffer:
        ^
```

Implement composable commands. A good way to start is to extend the `EditCommand` class and its subclasses to define the following method:

```
    def compose(other : EditCommand) : Option[EditCommand]
```

This method either:

- returns `None` if the current command cannot be composed with `other`.
- returns a new command if the current command can be composed with `other`, because, for example, they are both insert commands. In this case, the method should also change the current command to be the composed command.

For example,

```
   val c1 = new InsertCommand(target, "hel");
   val c2 = new InsertCommand(target, "lo"));
   c1.compose(c2) match {
     case None     => // can't combine them
     case Some(c3) => c3.execute();
   }
```

would create the command `c3` that inserts "hello" into the target. If we changed `c2` to be a `DeleteCommand`, the `compose` operation would return `None`. You may find it useful to test whether an object has a certain type, which can be done in Scala with pattern matching, as in:

```
  x match {
    case i : InsertCommand  => ... // x is an InsertCommand, now bound to i
    case i : DeleteCommand  => ... // x is an DeleteCommand, now bound to i
    case i                  => ... // match all other types
  }
```