Homework for Summer School Course in Grace at ECOOP 2015

Thursday, 7/9/2015

To run Grace programs, go to `http://web.cecs.pdx.edu/~grace/minigrace/exp/` using the Chrome web browser. (Up-to-date versions of Firefox will likely work as well, but Safari will not.). Be sure that your pop-up blocker is turned off if you are going to use the objectdraw library. Otherwise you will get an internal error in objectdraw referring to the ”dom”.

When you go to the Grace Editor web site the window will be divided into three panels. On the left will be the file pane with a a list of open files, the top right is the editor pane and will show the file you are currently working on, while the bottom right is the output pane, which will show error messages and program output.

To start a program, click on the ”New” button in the Files pane (also indicated by a circled plus sign) and give your program a name.

If the ”hello world” program is not already showing, type

```
print "hello world!"
```

in the editor pane. You can then compile and execute your program by clicking on the ”Run” button at the bottom of the editor pane. The output should show up in the editor pane.

1. (10 points) **Factorial** Write Grace methods to compute factorial. One version should compute it iteratively, while the other computes it recursively. Recall that Grace does not need “return” statements. A method simply returns the last value in the block.

   Here are some quick reminders of how we can write loops in Grace:

   ```
   for(1..10) do {n->print(n)}

   var i:= 1
   while{i <= 10} do {
      print(i)
      i := i+1
   }
   ```

2. (20 points) **Rational numbers**

   Implement a class for immutable exact rational numbers in Grace. It should support the operations +, -, *, /, ==, and asString. While you don't need to include type information in the first pass at your definition of the class, eventually we would like it to generate objects with the following type:

   ```
   type Rational = {
      numerator -> Number
      denominator -> Number
      + (other: Rational) -> Rational     // infix ops as methods
      * (Other: Rational) -> Rational
      - (Other: Rational) -> Rational
   ```

```
    / (Other: Rational) -> Rational
    == (other: Rational) -> Boolean
    asString -> String
  }
```

Recall that methods for infix operators are written just like regular methods, e.g.,

```
        method +(other: Rational) -> Rational {...}
```

Rationals should be created by an invocation of the form

```
    aRational.top (2) bottom (3)
```

Don't worry about dividing by 0; just return 0 when that happens. Recall that you can create a "getter" for a definition by labeling it "is public". Thus write:

```
 def x is public = 5
```

and now if o is an object with that declaration then $o.x$ requests the value of x. Similarly, writing "is readable" generates a getter for a variable, while adding "is writable" creates a setter method `x:=`.

3. (20 points) **Lists & Trees**

   While Grace already supports lists, it is useful exercise to see how you could write your own lists in Grace.

   Here is some Grace code to create an immutable list:

```
// helper class to build immutable lists one elt at a time
class mkList.cons(value, tailList) is confidential {
    method asString {"({head}:{tail})"}
    method head {value}
    method tail {tailList}
    // perform action on all elts of list from head to tail
    method do (action) {
        action.apply (head)
        tail.do (action)
    }
    // create a new list with the results of applying function
    // to each of the elements of the list
    method map (function) {
        mkList.cons (function.apply (head),
                     tail.map (function))
    }
    method size {1 + tail.size}
```

```
}

def emptyList = object {
    method asString {"<emptyList>"}
    method do(action) {}
    method map(function) {self}
    method size {0}
}

def aList = object {
   method with(*elts) {
      var newList := emptyList
      for(elts) do {e ->
         newList := mkList.cons (e, newList)
      }
      newList
   }
}

def oneToFour = aList.with (1, 2, 3, 4)

def squares = oneToFour.map {n -> n * n}
print "oneToFour is {oneToFour}"
print "oneToFour has length {oneToFour.size}"
print "oneToFour squared is {squares}"

print "list of just 5 is {aList.with (5)}"
```

Notice that `aList` is an object with a method, `with` that creates a list from any number of arguments. The "*" on the formal parameter `elts` of `with` indicates that it represents a variable number of arguments, which are treated in the code as though they are a (built-in) list. Notice that the invocation of the method looks exactly like an invocation of a class constructor.

Please write similar code to build immutable binary search trees over `Number`. Include code to build an empty tree, determine if a tree is empty, add an element, determine if an element is a member of the tree, and to do an inorder traversal performing an action on all elements of the tree.

4. (10 points) **Objectdraw**

   Please read the directions for writing code with the objectdraw library at `http://www.cs.pomona.edu/~kim/GraceStuff/UsingObjectdrawGrace.pdf`.

   Please write a program to draw a red box that is five pixels tall and 100 pixels wide in the middle of the bottom of a program window. Also draw a blue circle that is 20 pixels in diameter in the upper left hand corner of the window. The program should allow the user to drag the circle around the window. When the circle touches the red box it should "pop" by making the circle invisible and having it be replaced by the word "pop!". When the mouse is released, the circle should be moved back to its starting position and made visible again. It should then be able to be dragged again.

If you feel adventurous, modify the program so that the ball appears where ever the user clicks and then falls slowly down toward the floor. If it hits the hot box then it explodes as before. This program would use the Animation module.