Section 9 - Double Dispatch, Mixins, Visitors

This handout was composed by Porter Jones. There are probably plenty of typos/incorrect solutions/etc for you to catch! Please email me with any issues, comments, or feedback at pbjones@cs.washington.edu. All thoughts are welcome :)

Double Dispatch

Fill in the following class definitions that simulate a game of rock-paper-scissors using double dispatch. In case you aren't familiar with rock-paper-scissors, rock should beat scissors, scissors should beat paper, and paper should beat rock. Everything should tie with itself.

```
class Rock
  #TODO
  def fight other

  end
  def fightWithRock other
    "Tie"
  end
  #TODO
  def fightWithPaper other

  end
  #TODO
  def fightWithScissors other

  end
end
```

```
class Paper
  #TODO
  def fight other

  end
  def fightWithRock other
    "Paper wins"
  end
  #TODO
  def fightWithPaper other

  end
  #TODO
  def fightWithScissors other

  end
end
```

```
class Scissors
  #TODO
  def fight other

  end
  def fightWithRock other
    "Rock wins"
  end
  #TODO
  def fightWithPaper other

  end
  #TODO
  def fightWithScissors other

  end
end
```

Visitor Patterns

Below are definitions of three Ruby classes, each with its own `accept` method to accept a visitor. Assuming these classes, implement the visitors described after their definitions.

```
class Int
  attr_reader :i
  def initialize i
    @i = i
  end
  def accept(visitor, arg=nil)
    visitor.visitInt(self, arg)
  end
end
```

```
class Neg
  attr_reader :e
  def initialize e
    @e = e
  end
  def accept(visitor, arg=nil)
    visitor.visitNeg(self, arg)
  end
end
```

```
class Add
  attr_reader :e1, :e2
  def initialize(e1,e2)
    @e1 = e1
    @e2 = e2
  end
  def accept(visitor, arg=nil)
    visitor.visitAdd(self, arg)
  end
end

# A sample expression
SAMPLE =
  Neg.new(Add.new(Add.new(Add.new(Int.new(3),
                                  Neg.new(Int.new 9)),
                          Int.new(-42)),
                  Add.new(Int.new(73),
                          Neg.new(Int.new(14)))))
```

1) Implement a visitor that returns a count of the number of negations in a expression tree.

2) Implement a visitor that returns a string version of an expression tree. Instances of `Neg` should look like – `(e)` and instances of `Add` should look like `(e1 + e2)`.

3) Implement a visitor that evaluates an expression tree and returns the result as an `Int`.

4) Given the following datatype binding in SML, determine what SML construct would achieve the same behavior as the Ruby visitors we wrote above. Then implement that construct for each of the visitors we wrote above.

```
datatype exp = Int of int
             | Neg of exp
             | Add of exp * exp
```

Section 9 - Solutions

This handout was composed by Porter Jones. There are probably plenty of typos/incorrect solutions/etc for you to catch! Please email me with any issues, comments, or feedback at pbjones@cs.washington.edu. All thoughts are welcome :)

Double Dispatch

```
class Rock
  def fight other
    other.fightWithRock self
  end
  def fightWithRock other
    "Tie"
  end
  def fightWithPaper other
    "Paper wins"
  end
  def fightWithScissors other
    "Rock wins"
  end
end
```

```
class Paper
  def fight other
    other.fightWithPaper self
  end
  def fightWithRock other
    "Paper wins"
  end
  def fightWithPaper other
    "Tie"
  end
  def fightWithScissors other
    "Scissors wins"
  end
end
```

```
class Scissors
  def fight other
    other.fightWithScissors self
  end
  def fightWithRock other
    "Rock wins"
  end
  def fightWithPaper other
    "Scissors wins"
  end
  def fightWithScissors other
    "Tie"
  end
end
```

Visitor Patterns

```
1) class NegCounter
     def visitInt(int, arg)
       0
     end
     def visitNeg(neg, arg)
       1 + neg.e.accept(self)
     end
     def visitAdd(add, arg)
       add.e1.accept(self) + add.e2.accept(self)
     end
   end
```

```
2) class Stringer
     def visitInt(int, arg)
       int.i.to_s
     end
     def visitNeg(neg, arg)
       "-(" + neg.e.accept(self) + ")"
     end
     def visitAdd(add, arg)
       "(" + add.e1.accept(self) + " + " + add.e2.accept(self) + ")"
     end
   end
```

```
3) class Evaluator
     def visitInt(int, arg)
       int
     end
     def visitNeg(neg, arg)
       Int.new(- neg.e.accept(self).i)
     end
     def visitAdd(add, arg)
       Int.new(add.e1.accept(self).i + add.e2.accept(self).i)
     end
   end
```

4) Functions in SML that pattern match on the datatype will achieve similar behavior to the visitor patterns shown above. Below are implementations of the three analogous functions.

```
fun neg_counter e =
    case e of
        Int _ => 0
      | Neg e => 1 + neg_counter e
      | Add (e1,e2) => neg_counter e1 + neg_counter e2

fun stringer e =
    case e of
        Int i => Int.toString i
      | Neg e => "-(" ^ (stringer e) ^ ")"
      | Add (e1,e2) => "(" ^ (stringer e1) ^ " + " ^ (stringer e2) ^ ")"

fun evaluator e =
    case e of
        Int i => Int i
      | Neg e => (case evaluator e of Int i => Int (~i))
      | Add (e1,e2) => (case (evaluator e1, evaluator e2) of
                            (Int i, Int j) => Int (i + j))
```