

CS 94SI Assignment #3 – Unifying OOP and FP

Due: At 2pm on Wednesday, April 23, 2008

E-mail to: cs94si-spr0708-staff@lists.stanford.edu

Exercise – Mathematical Expressions

Algebraic data types are common in functional languages like Haskell or OCaml. In Scala they can be defined with the use of abstract classes and case classes. Consider the following algebraic data type for expressions:

```
sealed abstract class Expr
case class Variable(name: Symbol) extends Expr
case class Constant(x: Double) extends Expr
case class Sum(l: Expr, r: Expr) extends Expr
case class Product(l: Expr, r: Expr) extends Expr
case class Power(b: Expr, e: Expr) extends Expr
```

This Scala code is equivalent to the following definition in Haskell:

```
data Expr = Variable Symbol
         | Constant Double
         | Sum Expr Expr
         | Product Expr Expr
         | Power Expr Expr
```

Pattern matching is a convenient way to manipulate algebraic data types. For an example of using pattern matching to deal with the `Expr` type, take a look at the `print` function we've provided. The function takes an `Expr`, turns it into a string, and prints it to the screen.

- a) Write a function that takes the derivative of an expression with respect to a given variable. Your function should have the following signature:

```
def derive(e: Expr, s: Symbol): Expr
```

Your function does not have to take the derivative of Powers with non-constant exponents. It is acceptable to throw an exception in that circumstance.

Note that in Scala, Symbols can be declared by using a single quote before the name of the symbol, as such:

```
scala> 'x
res0: Symbol = 'x

scala> 'y
res1: Symbol = 'y

scala> 'abc
res2: Symbol = 'abc
```

- b) Write a function that evaluates a given expression in a given environment. An environment is just a mapping from symbols to values for those symbols. Your function should have the following signature:

```
def eval(e: Expr, env: Map[Symbol, Double]): Double
```

If a variable in the expression is not in the given environment, you should throw an exception.

- c) Write a function that when given an expression reduces that expression to its simplest form. Your function should have the following signature:

```
def simplify(e: Expr): Expr
```

If you think there might be confusion about what “simplest form” means in this context, please e-mail the staff list.

- d) It is unlikely that the function you wrote in part (b) can simplify every expression. Write some unit tests that show what expressions your function can simplify. The CS94SI staff will be running a small contest. For every unit test that you write (and pass!) that no one else in the class passes, you will receive one point. In order to participate in the contest, please submit your assignment by **2pm at the latest** on Wednesday, April 23.

In order to make the task of writing tests easier, we’ve provided an expression parser. The expression parser takes a string and returns its corresponding `Expr`. The expression parser can be invoked on a string `str` like so: `Expr(str)`. For example, to demonstrate that your simplifier knows about the additive identity of zero, you might write the following test:

```
assertEquals(Expr("x"), simplify(Expr("x + 0")))
```

The syntax that the expression parser accepts can be expressed in BNF:

```
expr := sum
sum := product { ("+" | "-") product }
product := power { "*" power }
power := factor [ "^" factor ]
factor := "(" expr ")" | variable | constant
variable := ident
constant := floatLit
floatLit := [ "-" ] positiveFloat
positiveFloat := numericLit [ "." [ numericLit ] ]
```

- e) Think about algebraic data types and pattern matching from an object-oriented perspective. What criticism might an object-oriented programmer make about pattern matching? How would you defend pattern matching against these criticisms?

Please hand in all the code you’ve written for this assignment.