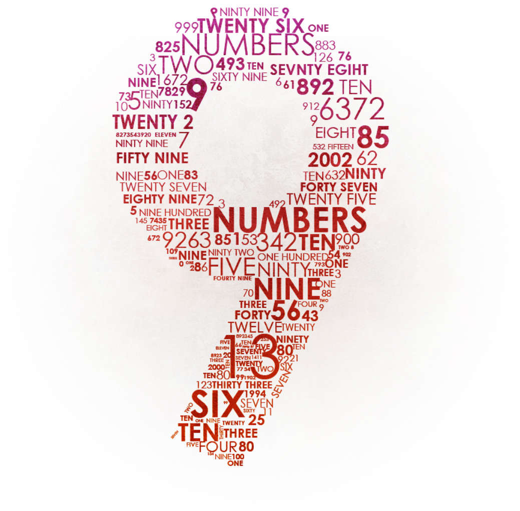
CIS 194: Homework 4

Due Wednesday, February 18, 2015



What is a Number?

This may sound like a deep, philosophical question, but the Haskell type system gives us a simple way to answer it. A number is any type that has an instance of the `Num` type class. Let's take a look at the definition of the `Num` type class as defined in the Haskell `Prelude`:

```
class  Num a  where
    (+), (-), (*)       :: a -> a -> a
    negate              :: a -> a
    abs                 :: a -> a
    signum              :: a -> a
    fromInteger         :: Integer -> a
```

So according to Haskell, a number is simply anything that can be added, subtracted, multiplied, negated, and so on[1]. The Haskell `Prelude` has a bunch of built in `Num` instances that we are already familiar with. These include the usual suspects: `Int`, `Integer`, `Float`, and `Double`. But the fun doesn't end there. We are free to define our own numbers as long as we can come up with meaningful definitions for the basic numeric operations.

[1] Notice that division is not included in the `Num` type class. Division is purposely left out because it is defined differently for integral and floating point types

Polynomials

Can a polynomial be a number? Sure! Why not? Polynomials can be added, subtracted, and multiplied just like any other *number*. In this homework, you will write a `Num` instance for a polynomial type.

Before we begin, let's define the representation of polynomials that we will be using in this assignment. A polynomial is simply a sequence of terms, and each term has a *coefficient* and a *degree*. For example, the polynomial $x^2 + 5x + 3$ has three terms, one of degree 2 with coefficient 1, one of degree 1 with coefficient 5, and one of degree 0 with coefficient 3. In Haskell, we will avoid explicitly specifying the degrees of terms by representing a polynomial as a list of coefficients, each of which has degree equal to its position in the list. We will leave the type of the coefficients to be polymorphic since we may want to have coefficients that are `Int`s, `Double`s, etc[2].

[2] There are some applications in Cryptography that use boolean polynomials. Our Haskell representation can even support that!

```
newtype Poly a = P [a]
```

In this representation, the polynomial $x^2 + 5x + 3$ would be written as `P [3, 5, 1]`.

Exercise 1 It would be really nice if GHCi could understand that `x` is the polynomial $f(x) = x$. Well, it can; we just have to give it a little help. Define the value:

```
x :: Num a => Poly a
```

To be the Haskell representation of the polynomial x, ie the degree 1 polynomial with coefficient 1.

Exercise 2 Notice that our `Poly a` type does not derive `Eq` or `Show`. There is a good reason for this! The default instances of these type classes do not really work the way we would want them to.

In this exercise you will write an instance of the `Eq` type class for `Poly a`. If we had derived this instance, Haskell would have simply compared the lists inside the `P` data constructor for equality. Think about why this is *not* good enough. When might two `Poly a`s be equivalent, but their list representations are not?

Implement the `(==)` function. Remember that we do not have to explicitly implement the `(/=)` function; it has a default implementation in terms of `(==)`.

Example: `P [1, 2, 3] == P [1, 2, 3]`

Example: `P [1, 2] /= P [1, 2, 3]`

Exercise 3 The default instance of `Show` simply displays values the way they are written in Haskell. It would be much nicer if a `Poly a` like `P [1, 2, 3]` could be displayed in a more human readable way, like `3x^2 + 2x + 1`. This will make it much easier to reason about your code throughout the rest of the assignment. A complete instance of the `Show` type class will have the following features:

- Terms are displayed as `cx^e` where `c` is the coefficient and `e` is the exponent. If `e` is `0`, then only the coefficient is displayed. If `e` is `1` then the format is simply `cx`.
- Terms are separated by the `+` sign with a single space on each side
- Terms are listed in *decreasing* order of degree
- Nothing is displayed for terms that have coefficient `0` unless the polynomial is equal to `0`.
- No coefficient is displayed for a term if the coefficient is 1, unless the degree is `0`, ie `x` instead of `1x`.
- No special treatment is necessary for terms that have negative coefficients. For example, `2x^2 + -3`, is the correct representation of $2x^2 - 3$.

Implement the `show` function according to this specification.

Example: `show (P [1, 0, 0, 2]) == "2x^3 + 1"`

Example: `show (P [0, -1, 2]) == "2x^2 + -x"`

Exercise 4 Now we will define addition for the `Poly a` type. Addition for polynomials is fairly simple, all we have to do is add the coefficients pairwise for each term in the two polynomials. For example $(x^2 + 5) + (2x^2 + x + 1) = 3x^2 + x + 6$.

It is considered good style to define significant functions outside of a type class instance. For this reason, you will write the function `plus` that adds two values of type `Poly a`:

```
plus :: Num a => Poly a -> Poly a -> Poly a
```

Notice that the type signature for `plus` has the constraint that `a` has a `Num` instance. This means that we can only add polynomials that have *numeric* coefficients[3]. Since `a` must be a `Num`, you can use all of the usual `Num` functions (ie, `(+)`) on the coefficients of the polynomial.

[3] So we will be able to add polynomials of polynomials once our `Num` instance is defined.

Example: `P [5, 0, 1] + P [1, 1, 2] == P [6, 1, 3]`

Example: `P [1, 0, 1] + P [1, 1] == P [2, 1, 1]`

Exercise 5 Remember FOIL[4] from high school algebra class? It is coming back to haunt you.

In this exercise you will implement polynomial multiplication. To multiply two polynomials, each term in the first polynomial must be multiplied by each term in the second polynomial. The easiest way to achieve this is to build up a `[Poly a]` where each element is the polynomial resulting from multiplying a single coefficient in the first polynomial by each coefficient in the second polynomial. Since the terms do not explicitly state their exponents, you will have to shift the output before multiplying it by each consecutive coefficient. For example `P [1, 1, 1] * P [2, 2]` will yield the list `[P [2, 2], P [0, 2, 2], P [0, 0, 2, 2]]`. You can then simply sum this list.

[4] FOIL (First Outer Inner Last) is a mnemonic for how to multiply binomials

Haskell's built in `sum` function is written in terms of `(+)`, but also uses the numeric literal `0`. If you would like to use `sum` then you will have to implement the `fromInteger` function in the `Num` type class instance for `Poly a` first (you will do this in the next exercise anyway). If you want, you can also use `foldr (+) (P [0])` in place of `sum` until you implement `fromInteger`.

Implement the function:

```
times :: Num a => Poly a -> Poly a -> Poly a
```

Example: `P [1, 1, 1] * P [2, 2] == P [2, 4, 4, 2]`

Exercise 6 Now it is time to complete our definition of the `Num` instance for polynomials. The `(+)` and `(*)` functions are already filled in for you using your implementations from the previous exercises. All you need to do is implement two more functions.

The first function you will implement is `negate`. This function should return the negation of a `Poly a`. In other words, the result of negating all of its terms. Notice that `(-)` is missing from our `Num` instance declaration. This is because the `Num` type class has a default implementation of `(-)` in terms of `(+)` and `negate`, so we don't have to implement it from scratch!

```
negate :: Num a => Poly a -> Poly a
```

Example: `negate (P [1, 2, 3]) == P [-1, -2, -3]`

Next, implement `fromInteger`. This function should take in an `Integer` and return a `Poly a`. An integer (or any other standard number for that matter) can simply be thought of as a degree 0 polynomial. Remember that you also have convert the `Integer` to a value of type `a` before you can use it as a coefficient in a polynomial!

```
fromInteger :: Num a => Integer -> Poly a
```

The `Num` type class has two more functions that do not really make sense for polynomials[5]. These functions are `abs` and `signum`. We will leave these as `undefined` since the absolute value of a polynomial is *not* a polynomial and polynomials do not really have have a sign.

[5] Maybe polynomials aren't numbers after all

Using Polynomials

Now that we have defined the `Num` instance for polynomials, we can stop using coefficient list syntax. In Haskell, the polynomial $x^2 + 5x + 3$ can be written as `x^2 + 5*x + 3`. This is because the values `x`, `5`, and `3` are all valid values of type `Poly Int` and we defined `(+)` and `(*)` as part of the `Num` instance for polynomials. Note how we can use `(^)` for exponentiation even though we did not define it. In Haskell, `(^)` is defined in terms of `(*)`, so we get it for free after implementing the `Num` instance.

Exercise 7 All this time we have been talking about adding and multiplying polynomials, but we have never said anything about evaluating them! Define the function `applyP` that applies a `Poly a` to a value of type `a`.

Example: `applyP (x^2 + 2*x + 1) 1 == 4`

Example: `applyP (x^2 + 2*x + 1) 2 == 9`

Exercise 8 We have already seen that we can write sensible instances of `Eq`, `Show`, and `Num` for polynomials. Another useful operation that we can perform on polynomials is *differentiation*. However, polynomials are not the only type of mathematical function that we can take derivatives of. For this reason we will define a new `Differentiable` type class.

```
class Num a => Differentiable a where
    deriv  :: a -> a
    nderiv :: Int -> a -> a
```

The two functions in the `Differentiable` type class are `deriv` (the first derivative) and `nderiv` (the n^{th} derivative) of the input. Instances of this type class should obey the following laws:

- $\forall$n>0, `nderiv (n - 1) (deriv f) == nderiv n f`
- $\forall$n>0, `deriv (nderiv (n - 1) f) == nderiv n f`

We can provide a default implementation of `nderiv` in the type class definition for `Differentiable` that is written in terms of `deriv`. The value of `nderiv n f` should be equal the result of applying the `deriv` function `n` times[6]. Implement the `nderiv` function.

```
nderiv :: Differentiable a => Int -> a -> a
```

Note: you will probably not be able to test this until you complete the next exercise.

[6] Is there a more efficient way to do this?

Exercise 9 Now that we have defined the `Differentiable` type class, you can create an instance of `Differentiable` for the type `Poly a`. All you need to do is fill in the definition of the `deriv` function since a default implementation of `nderiv` has already been supplied.

```
deriv :: (Num a, Enum a) => Poly a -> Poly a
```

In case you are rusty on calculus, here is a recap of the differentiation rules for polynomials. The derivative of a term cx^e is simply $ce \cdot x^{e-1}$. The derivative of a sequence of terms is simply the sum of their individual derivatives.

Example: `deriv (x^2 + 3*x + 5) == 2*x + 3`