COMPSCI 1JC3

Introduction to Computational Thinking

Fall 2019

Assignment 3

Dr. William M. Farmer

McMaster University

Revised: October 22, 2019

The purpose of Assignment 3 is to write a module in Haskell that represents polynomials in two different ways and implements functions that map polynomials from one representation to the other. The requirements for Assignment 3 and for Assignment 3 Extra Credit are given below. You are required to do Assignment 3, but Assignment 3 Extra Credit is optional. Please submit Assignment 3 as a single `Assign_3.hs` file to the Assignment 3 folder on Avenue under Assessments/Assignments. If you choose to do Assignment 3 Extra Credit for extra marks, please submit it also as a single `Assign_3_ExtraCredit.hs` file to the Assignment 3 Extra Credit folder on Avenue in the same place. Both Assignment 3 and Assignment 3 Extra Credit are due **November 3, 2019 before midnight.** Assignment 3 is worth 4% of your final grade, while Assignment 3 Extra Credit is worth 2 extra percentage points.

Late submissions will not be accepted! So it is suggested that you submit a preliminary `Assign_3.hs` file well before the deadline so that your mark is not zero if, e.g., your computer fails at 11:50pm on November 3.

Although you are allowed to receive help from the instructional staff and other students, your submitted program must be your own work. Copying will be treated as academic dishonesty!

1 Background

Let C be a set of *coefficients* closed under addition and multiplication such as the integers $\mathbb{Z}$, the rational numbers $\mathbb{Q}$, or the real numbers $\mathbb{R}$. A *polynomial over* C is a mathematical expression that is constructed from an *indeterminant* x and members of C by applying addition ($+$) and multiplication ($*$) operators. The *zero polynomial* is the coefficient 0 by itself. Let P be the set of polynomials over some C. The value of a polynomial $p \in P$ at $c \in C$ is the result of replacing the indeterminant x with c. For example, the value of $(2 * x) + 4$ at 3 is $(2 * 3) + 4 = 10$.

Every polynomial p represents a *polynomial function* $f_p : C \to C$ that maps $c \in C$ to the value of p at c. For every $p \in P$ except the zero polynomial, there is a $q \in P$ that has the form

$$a_0 + a_1 * x^1 + a_2 * x^2 + \cdots + a_m * x^m,$$

where $a_0, a_1, \ldots, a_m \in C$, $a_m \neq 0$, x^i is an abbreviation for $x * \cdots * x$ (i times), and parentheses have been suppressed, such that f_p and f_q are the same function. q is called the *standard form* of p. The *degree* of p is m, the largest exponent appearing in q. For example, the standard form of $(x + 1) * (x + 2)$ is

$$2 + 3 * x^1 + x^2$$

and the degree of $(x + 1) * (x + 2)$ is 2. The degree of the zero polynomial is undefined.

A polynomial can be represented by the list of the coefficients in its standard form. That is, if

$$a_0 + a_1 * x^1 + a_2 * x^2 + \cdots + a_m * x^m$$

where $a_m \neq 0$ is the standard form of a nonzero polynomial p, then p can be represented by the list

$$[a_0, a_1, \ldots, a_m].$$

The zero polynomial can be represented by the empty list $[\,]$. Polynomials can be processed by manipulating their representations as lists. For example, two polynomials can be added by adding the corresponding components in their representations as lists.

We will call a list that represents a polynomial a *polynomial list*. Every list of numbers whose final value is not 0 is a polynomial list.

2 Assignment 3

The purpose of this assignment is to create a Haskell module for polynomials over an arbitrary type.

2.1 Requirements

1. Download from Avenue `Assign3_Project_Template.zip` which contains the Stack project files for this assignment. Modify the `Assign_3.hs` file in the `src` folder so that the following requirements are satisfied.

2. Your name, the date, and "Assignment 3" are in comments at the top of your file. `macid` is defined to be your MacID.

3. The file contains the algebraic data type definition

```
data Poly a =
    X
  | Coef a
  | Sum (Poly a) (Poly a)
  | Prod (Poly a) (Poly a)
  deriving Show
```

4. The file includes a function named `polyValue` of type

   ```
   Num a => Poly a -> a -> a
   ```

   such that `polyValue p n` is the value of `p` at `n`.

5. The file contains the following newtype:

   ```
   newtype PolyList a = PolyList [a]
     deriving Show
   ```

6. The file includes a function named `polyListValue` of type

   ```
   Num a => PolyList a -> a -> a
   ```

   such that, if `pl` is a polynomial list and `n` is a number, `polyListValue pl n` is the value of the polynomial function represented by `pl` at `n`. Hint: Use Horner's method[1] to do the computation:

   $$a_0 + a_1 * x^1 + a_2 * x^2 + \cdots + a_m * x^m = a_0 + x * (a_1 + x(a_2 + \cdots + x * (a_m)))$$

7. The file includes a function named `polyListSum` of type

   ```
   (Num a, Eq a) => PolyList a -> PolyList a -> PolyList a
   ```

   such that, if `pl` and `ql` are polynomial lists, `polyListSum pl ql` is the polynomial list that represents of the sum of the polynomials represented by `pl` and `ql`.

8. The file includes a function named `polyListDegree` of type

   ```
   (polyListDegree ::  (Num a,Eq a) => PolyList a -> Integer
   ```

   that returns the degree of the polynomial (remembering that the degree of the zero polynomial is undefined).

9. The file includes a function named `polyListProd` of type

   ```
   (Num a, Eq a) => PolyList a -> PolyList a -> PolyList a
   ```

   such that, if `pl` and `ql` are polynomial lists, `polyListProd pl ql` is the polynomial list that represents of the product of the polynomials represented by `pl` and `ql`.

[1] According to the Wikipedia article on Horner's method, the method is named after William George Horner (1786–1837), but the method was known before him by Paoli Ruffini (1765–1822) and Qin Jiushao (1202–1261).

10. The file includes a function named `polyListToPoly` of type

    ```
    Num a => PolyList a -> Poly a
    ```

    such that, if `pl` is a polynomial list, `polyListToPoly pl` is a polynomial whose standard form is represented by `pl`.

11. The file includes a function named `polyToPolyList` of type

    ```
    (Num a, Eq a) => Poly a -> PolyList a
    ```

    such that `polyToPolyList p` is the polynomial list that represents the standard form of `p`.

12. Your file can be imported into GHCi and all of your functions perform correctly.

2.2 Testing

Include in your file a test plan for all the functions mentioned above. The test plan must include at least three test cases for each function. Each test case should have following form:

`Function:` Name of the function being tested.
`Test Case Number:` The number of the test case.
`Input:` Inputs for function.
`Expected Output:` Expected output for the function.
`Actual Output:` Actual output for the function.

The test plan should be at the bottom of your file in a comment region beginning with a `{-` line and ending with a `-}` line.

3 Assignment 3 Extra Credit

The purpose of this assignment is to create a Haskell module for polynomials over an arbitrary type using an alternate algebraic data type.

3.1 Requirements

1. Modify the `Assign_3_ExtraCredit.hs` file in the `src` folder (not `Assign_3.hs`) so that the following requirements are satisfied.

2. Your name, the date, and "Assignment 3 Extra Credit" are in comments at the top of your file. `macid` is defined to be your MacID.

3. The file contains the algebraic data type definition

```
data PolyAlt a =
    Monomial a Integer
  | SumAlt (PolyAlt a) (PolyAlt a)
  deriving Show
```

`Monomial` $c\,e$ represents the polynomial $c * x^{|e|}$. For example, `Monomial 2 (-3)` represents $2 * x^3$.

4. The file includes a function named `polyAltValue` of type

   ```
   Num a => PolyAlt a -> a -> a
   ```

   such that `polyAltValue p n` is the value of `p` at `n`.

5. The file includes a function named `polyAltDegree` of type

   ```
   (Num a, Eq a) => PolyAlt a -> Integer
   ```

   such that `polyAltDegree p` is the degree of `p`.

6. The file includes a function named `polyAltDeriv` of type

   ```
   Num a => PolyAlt a -> PolyAlt a
   ```

   such that `polyAltDeriv p` represents the derivative of the polynomial function represented by `p`. `polyAltDeriv p` thus symbolically differentiates a polynomial `p`.

7. The file includes a function named `polyAltProd` of type

   ```
   Num a => PolyAlt a -> PolyAlt a -> PolyAlt a
   ```

   such that `polyAltProd p q` represents the product of the polynomial functions represented by `p` and `q`.

8. The file includes a function named `polyAltNewton` of type

   ```
   (Fractional a, Ord a) => PolyAlt a -> a -> a -> a
   ```

   such that `polyAltNewton p s t` computes, using Newton's method with the seed `s` and the tolerance `t`, a number `n` such that

   `polyValue p n` $<$ `t`.

   `polyAltNewton p s t` thus computes an approximate solution to the polynomial equation `p` $= 0$.

9. The file includes a function named `polyToPolyAlt` of type

   ```
   (Num a, Eq a) => Poly a -> PolyAlt a
   ```

   such that `polyToPolyAlt p` represents the same polynomial function as `p`.

10. The file includes a function named `polyAltToPoly` of type

    ```
    (Num a, Eq a) => PolyAlt a -> Poly a
    ```

    such that `polyAltToPoly p` represents the same polynomial function as `p`.

11. Your file successfully loads into GHCi and all of your functions perform correctly.

3.2 Testing

Include in your file a test plan for the functions `polyAltValue`, `polyAltDegree`, `polyAltDeriv`, `polyAltProd`, `polyAltNewton`, `polyToPolyAlt`, and `polyAltToPoly`. The test plan must include at least three test cases for each function.