COMPSCI 1JC3 C01

Introduction to Computational Thinking

Fall 2018

Assignment 4

Dr. William M. Farmer

McMaster University

Revised: November 12, 2018

Assignment 4 is a continuation of Assignment 4. Its purpose is to write a module in Haskell that involves I/O and manipulation of lists. The requirements for Assignment 4 and for Assignment 4 Extra Credit are given below. You are required to do Assignment 4, but Assignment 4 Extra Credit is optional. Please submit Assignment 4 as two files, `Assign_4.hs` and `Assign_4_Test.hs`, to the Assignment 4 folder on Avenue under Assessments/Assignments. If you choose to do Assignment 4 Extra Credit for extra marks, please submit it also as two files, `Assign_4_ExtraCredit.hs` and `Assign_4_Test_ExtraCredit.hs`, to the Assignment 4 Extra Credit folder on Avenue in the same place. Both Assignment 4 and Assignment 4 Extra Credit are due **November 18, 2018 before midnight.** Assignment 4 is worth 4% of your final grade, while Assignment 4 Extra Credit is worth 2 extra percentage points.

Late submissions will not be accepted! So it is suggested that you submit a preliminary `Assign_4.hs` file well before the deadline so that your mark is not zero if, e.g., your computer fails at 11:50pm on November 18.

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 Assignment 4

The purpose of this assignment is to create a Haskell module for polynomials represented as lists of coefficients.

1.1 Background

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.

1.2 Requirements

1. Download from Avenue `Assign4_Project_Template.zip` which contains the Stack project files for this assignment. Modify the `Assign_4.hs` file in the `src` folder so that the following requirements are satisfied. Also put your testing code for this assignment in the `Assign_4_Test.hs` file in the `test` folder.

2. Your name, the date, and "Assignment 4" are in comments at the top of your file. `macid` is defined to be your MacID.

3. The file contains the following algebraic data type definition from Assignment 3:

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

4. The file contains the following synonym type:

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

5. The file includes a function `getPolyList` of type

```
FilePath -> IO (PolyList Integer)
```

   that reads the coefficients of the standard form of a polynomial from a file in which there is one integer per line where the first integer is a_0, the second a_1, etc. and then returns the inputted integers as a polynomial list.

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 `polyListDegree` of type

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

   such that, if `pl` is a polynomial list, `polyDegree pl` is the degree of the polynomial represented by `pl`. The degree of the polynomial list `[]` should be undefined, since the degree of the zero polynomial is undefined.

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

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

   such that, if `pl` is a polynomial list, `polyListDeriv pl` is the polynomial list that represents the derivative of the polynomial represented by `pl`. `polyListDeriv pl` thus symbolically differentiates a polynomial list `pl`.

9. 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`.

10. 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`.

11. 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`.

[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).

12. 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`.

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

1.3 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.

In addition, your test plan must include at least one QuickCheck case for each of the functions `polyListValue`, `polyListDegree`, `polyListDeriv`, `polyListSum`, and `polyListProd`. Each QuickCheck case should have following form:

`Function:` Name of the function being tested.
`Property:` Code defining the property to be tested by QuickCheck.
`Actual Test Result:` Pass or Fail.

The test plan should be at the bottom of your file in a comment region beginning with a `{-` line and ending with a `-}` line. Put your testing code for this assignment in the `Assign_4_Test.hs` file in the `test` folder.

2 Assignment 4 Extra Credit

The purpose of this extra credit assignment is to implement both unary and binary addition and multiplication and compare their execution times.

2.1 Background

Natural numbers can be represented in a *unary* format in which, for example, $S(S(S(S(S(0)))))$ represents 5. S is interpreted as the successor function: $\lambda\, n \in \mathbb{N} \,.\, n+1$. Addition and multiplication can easily be defined on this unary representation, but unary addition and multiplication are extremely inefficient.

Natural numbers can also be represented in a *binary* format in which, for example, 101 represents 5. A string of 0s and 1s is stitched together with a binary function h that is interpreted as

$$\lambda\, n \in \mathbb{N} \,.\, \lambda\, d \in \{0, 1\} \,.\, (2 * n) + d.$$

Thus

$$101 = h(h(1, 0), 1) = (2 * (2 * 1 + 0)) + 1 = 5.$$

It takes some work to define addition and multiplication on this binary representation, but binary addition and multiplication are much more efficient than unary addition and multiplication.

2.2 Requirements

1. Modify the `Assign_4.hs` file in the `src` folder so that the following requirements are satisfied. Also put your testing code for this assignment in the `Assign_4_Test.hs` file in the `test` folder.

2. Your name, the date, and "Assignment 4 Extra Credit" are in comments at the top of your file. `macid` is defined to be your MacID.

3. The file contains the following three algebraic data type definitions:

```
data Nat =
    Z
  | S Nat

data Digit = Zero | One
    deriving Show

data BinNat =
    Atom Digit
  | Compound BinNat Digit
```

4. The file includes a function named `natPrint` of type `Nat -> String` that takes a member of `Nat` (e.g., `S (S (S (S (S Z))))`) as input and returns a string representing it as output (e.g., `"Nat:SSSSS0"`). Using `natPrint`, the file defines `Nat` as a instance of the type class `Show`.

5. The file includes a function named `binNatPrint` of type `BinNat -> String` that takes a member of `BinNat` (e.g.,

   ```
   Compound (Compound (Atom One) (Atom Zero)) (Atom One))
   ```

   as input and returns a string representing it as output (e.g., `"BinNat:101"`). Using `binNatPrint`, the file defines `BinNat` as a instance of the type class `Show`.

6. The file includes a function named `natParse` of type `String -> Nat` that takes a string as input (e.g., `"Nat:SSSSS0"`) as input and returns the member of `Nat` (e.g., `S (S (S (S (S Z)))))`) that represents the string as output. Using `natParse`, the file defines `Nat` as a instance of the type class `Read`.

7. The file includes a function named `binNatParse` of type `String -> Nat` that takes a string as input (e.g., `"BinNat:101"`) as input and returns the member of `BinNat` (e.g.,

   ```
   Compound (Compound (Atom One) (Atom Zero)) (Atom One))
   ```

   that represents the string as output. Using `binNatParse`, the file defines `BinNat` as a instance of the type class `Read`.

8. The file includes the following functions:

   a. `plus :: Nat -> Nat -> Nat`.

   b. `time :: Nat -> Nat -> Nat`.

   c. `binPlus :: BinNat -> BinNat -> BinNat`.

   d. `binTimes :: BinNat -> BinNat -> BinNat`.

   `plus` and `times` implement unary addition and multiplication, while `binPlus` and `binTimes` implement binary addition and multiplication.

9. The file includes an `IO` function that compares the execution times of `plus` and `times` with `binPlus` and `binTimes` on inputs that represent the same natural numbers. Call this function from the `main` in `test/Assign_4_test.sh`. (DO NOT run this in `ghci`, it will give you unreliable timings.) `NOTE`: do not use `System.Time` to do this; use the `defaultMain` function from `Criterion.Main`.

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

2.3 Testing

Include in your file a test plan for each of the functions mentioned above. The test plan must include at least three test cases and one QuickCheck case for each 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. Put your testing code for this assignment in the `Assign_4_Test_ExtraCredit.hs` file in the `test` folder.