Functional Programming

Coursework 2 - Simplify

Introduction

In this coursework assignment, you will implement a simplifier for symbolic arithmetic expressions involving addition, multiplication and integer exponentiation (x^n where n is a non-negative integer).

Downloading the skeleton code

1. Download the skeleton code zip file by going to this link.
2. Extract the contents of the zip to wherever you want to save it
   - How to extract zip files on Windows
   - How to extract zip files on Ubuntu
   - How to extract zip files on Mac

Compiling and running tests

If your code passes all the tests, you get all the marks! It's therefore essential that you can compile and run the tests. To do this:

1. In VSCode (in WSL mode if you're on Windows), open the folder containing the skeleton code
2. Open an integrated terminal in the VSCode window, either by using the menu bar ("Terminal -> New Terminal") or by using the shortcut (Ctrl + Shift + `).

3. Run `cabal update` (this will update your package list for Haskell libraries)
4. Run `cabal build` (this will build your project and the tests, which may take a little while the first time)

5. Run `cabal test` (this will run the tests)

When you run this on the skeleton code all the tests should fail, because nothing has been implemented yet.

Every time you make a change, run `cabal test` to test it! You can also load your code in GHCi/interactive mode using `cabal repl`

Write all your code in `Simplify.hs`, but don't change the name or type of the functions and don't change any other files. You are free to write any helper functions you like, as long as you write them in `Simplify.hs`.

Calculating your mark

Your mark will be based on the percentage of *tasks* you implement correctly. *This is not the same as the percentage of passing tests*. Instead, to get the marks for a task, *you must pass every test for that task*. This means all tasks are worth an equal number of marks, even if they have a different number of tests.

Submitting your coursework

Submit your work via Blackboard. You should submit *only* `Simplify.hs`. Do not rename `Simplify.hs`. Do not submit any other files. Do not submit a zip file.

You are responsible for early submission, therefore you should submit *at least* an hour before the deadline to ensure there are no upload problems. The university systems will automatically apply harsh penalties if the coursework is late by even a second.

The *Expr* Type

Expr is a data type which represents a subset of symbolic arithmetic expressions. It is defined in `Expr.hs` as:

```
data Expr
  = Op BinOp Expr Expr
  | NumLit Int
  | ExpX Int
  deriving (Eq, Show)
data BinOp = AddOp | MulOp
  deriving (Eq, Show)
```

It can represent:

- Integers, e.g.
  - *NumLit* 3 represents 3
- Addition, e.g.
  - *Op AddOp* (*NumLit* 2) (*NumLit* 3) represents $2 + 3$
- Multiplication, e.g.
  - *Op MulOp* (*NumLit* 2) (*NumLit* 3) represents $2 * 3$
- Non-negative integer powers of a single variable x, e.g.
  - *ExpX* 2 represents x^2
  - *ExpX* 9 represents x^9
  - *ExpX* 0 represents x^0
  - *ExpX* (-2) does not cause a type error, but we consider it to be invalid because it contains a negative power. The tests will not generate any negative powers in an *Expr* and your functions should not introduce any.
- Combinations of these terms, e.g.
  - *Op MulOp* (*NumLit* 1) (*Op AddOp* (*ExpX* 4) (*NumLit* 3)) represents $1 * (x^4 + 3)$

`Expr.hs` also exports two functions.

eval

eval :: *Int* → *Expr* → *Int*

eval evaluates an expression using the given a value for x, e.g.

eval 2 (*ExpX* 3) == 8

prettyExpr

prettyExpr :: *Expr* → *String*

prettyExpr takes an *Expr* and shows it in a prettier format than the default *Show* instance, e.g.

```
prettyExpr (NumLit 4) == "4"
prettyExpr (ExpX 2) == "x^2"
prettyExpr (ExpX 1) == "x"
prettyExpr (Op AddOp (NumLit 1) (ExpX 2)) == "1 + x^2"
```

This function isn't needed for the tasks or the tests, but you may find it useful while writing/debugging your code.

‘Junk’

In the following tasks you’ll have to make sure you don’t introduce ‘junk’ into your expressions. Junk is defined to be:

- addition of zero
- addition or multiplication of number literals
- $x^n + x^n$ (should be simplified to $2 * x^n$)
- multiplication by one or zero
- $x^n * x^m$ (should be simplified to x^{n+m}, e.g. $x^2 * x^3$ is junk because it should be simplified to x^5)
- x^0 (should be 1)

Anything else is not considered ‘junk’, and therefore you don’t need to worry about those cases (though you could if you wanted to). For example, $2x^2 + 3x^2$ could be simplified to $5x^2$, but you will neither be penalised nor rewarded for doing this.

Task 1

Define a function *add* :: *Expr* $\rightarrow$ *Expr* $\rightarrow$ *Expr*, which adds 2 expressions together without introducing any ‘junk’, e.g.

- adding 1 and x^4 should result in $1 + x^4$
- adding 0 and x^4 should result in x^4
- adding 1 and 2 should result in 3

Task 2

Define a function *mul* :: *Expr* $\rightarrow$ *Expr* $\rightarrow$ *Expr*, which multiplies 2 expressions together without introducing any ‘junk’, e.g.

- multiplying 2 and x^4 should result in $2 * x^4$
- multiplying 0 and x^4 should result in 0
- multiplying 1 and x^4 should result in x^4

Task 3

Define a function *addAll* :: $[$*Expr*$]$ $\rightarrow$ *Expr*, which adds a list of expressions together into a single expression without introducing any ‘junk’.

Hint: You might want to reuse your *add* function.

Task 4

Define a function *mulAll* :: $[$*Expr*$]$ $\rightarrow$ *Expr*, which multiplies a list of expressions together into a single expression without introducing any ‘junk’.

Hint: You might want to reuse your *mul* function.

The *Poly* Type

`Poly.hs` exports a type *Poly* for representing polynomials (expressions like $x^3 - 3x + 7$). Watch this short video for an introduction to `Poly.hs`. (Note: In the video *Poly* has a unicode *Show* instance, but our `Poly.hs` has a *Show* instance that's closer to what the data actually looks like. To print a *Poly* as pretty unicode, use *prettyPoly* instead.)

To represent a polynomial with the *Poly* data type, you need a list of the polynomial's coefficients, most significant power first. For example, $x^3 + 2x - 1$ corresponds to the list $[1, 0, 2, -1]$. The `Poly` module also exports a function *listToPoly*, which converts this list of coefficients into a *Poly*.

Note that the *Poly* type is an instance of the *Num* typeclass. This means that all the functions that *Num* defines can be used on *Poly*. Use Hoogle to find out what these functions are.

Note: If you are using Windows, it seems that printing Unicode characters does not work out of the box and you will get an exception when trying to *prettyPoly* a polynomial. To fix this you need to type `chcp 65001` in your terminal before running the code.

Task 5

Define the function:

exprToPoly :: *Expr* → *Poly*

which converts an expression into a polynomial.

Here it is important to think recursively: how can you convert a bigger expression by converting its sub-expressions and then combining them in a suitable way?

Hint: Since *Poly* is an instance of *Num*, you can use $+$ and $*$ to add and multiply two polynomials, respectively.

Task 6

Define the the function:

polyToExpr :: *Poly* → *Expr*

which converts a polynomial into an expression.

You can convert a polynomial to a list of its coefficients using the function *polyToList* :: *Poly* → [*Int*], which `Poly.hs` exports.

Task 7

Define a function which simplifies an expression by converting it to a polynomial and back again:

simplify :: *Expr* → *Expr*

Hint: Your previous functions can do the hard work here. Reuse them!