Exercise 6: More OOP, and Haskell

In this exercise, you'll get some more practice working with macros and the basic object-oriented functionality we implemented in lecture last week. We'll also start to introduce Haskell into our exercises.

The OOP portion of this week's exercise is inspired by Haskell records, which is its version of C structs (i.e., types with named fields).

Deadline: October 29, 2019 before 10:00pm

Starter code

The Racket starter code this week contains the `my-class` macro. This is the same macro that will appear in the aid sheet in the midterm.

- ex6.rkt
- Ex6.hs
- Ex6Types.hs

Task 1: Accessor Functions in `my-class`

Implement a new macro called `my-class-getter`. This macro has the same functionality as `my-class`, except that in addition to defining a class constructor function `<Class>`, it *also defines accessor functions* for each attribute of the class, using the name of the attribute as the name of its corresponding accessor. For example:

```
(my-class-getter Point (x y)
  ; Zero or more methods, omitted here
  )

(define p (Point 2 3))
(p 'x)           ; 2
(x p)            ; also 2
```

Note: this naming convention can certainly lead to name collisions down the line, a common problem with Haskell records. The Scheme convention is to name accessors `Point-x` and `Point-y` instead of `x` and `y`. However, doing so requires using a more powerful way of defining macros.

Task 2: Working with Haskell

In `Ex5.hs`, we've given function stubs for functions equivalent to the ones you wrote back in Exercise 1. Your job here is to take your implementations of functions from Racket and translate them to Haskell.

These are the functions that you'll be implementing

- celsiusToFahrenheit
- nCopies
- numEvens
- numManyEvens
- calculate

While you will need to look up built-in Haskell functions, your final code should look almost identical to the Racket code—and that's the point!

Tips:

- This section of *Learn You a Haskell* describes how to load your function definitions into the Haskell interpreter (`ghci`), for interactive testing.
- You can also use `runhaskell Ex1.hs` from the command line to run the entire module, which evaluates the `main` function, containing the provided sample property-based tests.
- The Haskell compiler error messages can be daunting. The most common beginner errors are type errors due to misunderstanding precedence (e.g., causing functions to be called on the wrong number of arguments).

When starting out, we recommend *using parentheses aggressively* to mark subexpression boundaries. Once your program compiles (and is correct!), you can start removing redundant parentheses.

The Haskell Calculator

We will not re-build the entire calculator interpreter in Haskell. However, the calculator grammar gives a good introduction to the *type definitions* in Haskell.

In Haskell, we can represent the definition of a binary arithmetic expression from Exercise 1 as follows:

```
data Expr = Number Float
          | Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          deriving (Show, Eq)
```

Here, `Expr` is the name of a new data type that we defined. There are five ways of defining this data type (five different constructors), by using one of `Number`, `Add`, `Sub`, `Mul` or `Div`. (Ignore the `deriving (Show, Eq)` for this exercise.)

We use `Float` for numbers instead of `Integer` so that division works reasonably.

Now, to create an expression equivalent to `(+ 2 (- 5 1))`, we would write

```
Add (Number 2) (Sub (Number 5) (Number 1))
```

The `calculate` function should take an `Expr`, and evaluate it to return a single number.

We've provided some starter code to guide you on how to use *pattern-matching* to deconstruct the data structure.

Note: the other provided Haskell file, `Ex5Types.hs`, contains the above `Expr` definition, as well as a few helpers used for testing. Do not make any modifications to this file. You won't be able to submit this file, and we will supply our own for testing.

Task 3: Generating Calculator Expressions

So far, we have seen how grammars can be used to define the syntax of a language, and (more powerfully) to define the *recursive structure* of terms in the language that suggest how to perform operations on them.

In this task, you'll learn about a new form of computing with grammars: using their recursive structure to *generate* expressions in a language.

(Fun fact: the Haskell testing library `QuickCheck` does something similar, though much more sophisticated, when generating inputs for property-based tests.)

Consider a subset of the calculator grammar:

```
<expr> = (Add <expr> <expr>)
       | (Mul <expr> <expr>)
       | Number 1 | Number 2 | Number 3 | Number 4
```

That is, it consists of arbitrarily-nested addition and multiplication expressions, where each numeric literal is a digit between 1 and 4.

On this exercise, we will call the `Number n` case for `<expr>` a *base case* because it is not recursive, while the `Add` and `Mul` cases *recursive* because it expands an `<expr>` in terms of other `<expr>`s.

We define the **rank** of an expression in this language as the *maximum depth* of the nesting of its subexpressions. We can define this recursively based on the grammar:

- A `Number n` has rank 0.
- An expression of the form `(Plus <expr> <expr>)` or `(Mul <expr> <expr>)` has rank one greater than the *maximum* of the rank of its two subexpressions.

Rank gives us a way of comparing the size or complexity of our generated expressions. This proves useful when generating expressions, as this notion gives us a way of ordering expressions as we generate them.

Note about efficiency

We will mainly, but not exclusively, test your work on "small" inputs (i.e., rank less than or equal to 2). For full credit, you *should* think about efficiency in your work. In particular:

- Though you should not return any duplicates, it is very inefficient to generate a (very) large list of expressions, and then remove duplicates.
- Instead, think about how to divide up the "expressions of rank k" into *mutually exclusive* sets of expressions, generate each set separately, and then combine them in the end.
- Don't repeat the same recursive call in multiple places; instead, bind the recursive call to a name (using `let`), and then simply use the name repeatedly.

Submission instructions

Submit the files `ex6.rkt` and `Ex6.hs` to MarkUs.