CIS 194: Homework 5

Due Friday, October 3, 2014

No template file is provided for this homework. Download the `Ring.hs` and `Parser.hs` files from the website, and make your `HW05.hs` Haskell file with your name, any sources you consulted, and any other relevant comments (just like in previous assignments). Then, say

```
module HW05 where

import Ring
import Parser
```

and off you go. Do make sure to include this module header, as it makes grading much easier for us.

Rings

A *ring* is a mathematical structure obeying certain laws. The Wikipedia page at http://en.wikipedia.org/wiki/Ring_(mathematics) is a great introduction. To help frame the concept, it is helpful to know that the integers form a ring.

A ring has a carrier set R (such as the integers), an addition operation, and a multiplication operation. In this document, we write the addition operation with $+$ and the multiplication operation with $\times$. The operations obey the following laws:

1. $+$ is associative. That is, $(a + b) + c = a + (b + c)$, for all a, b, and c in R.
2. There exists a special element $0 \in R$ such that $0 + a = a$ for all $a \in R$. 0 is the *additive identity*.
3. For every element $a \in R$, there exists an element $-a$ such that $-a + a = 0$. $-a$ is the *additive inverse* of a.
4. $+$ is commutative. That is, $a + b = b + a$ for all $a, b \in R$.
5. $\cdot$ is associative. That is, $(a \cdot b) \cdot c = a \cdot (b \cdot c)$ for all $a, b, c \in R$.
6. There exists a special element $1 \in R$ such that $1 \cdot a = a$ and $a \cdot 1 = a$ for all $a \in R$. 1 is the *multiplicative identity*.
7. $\cdot$ distributes over $+$. That is $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ and $(b + c) \cdot a = (b \cdot a) + (c \cdot a)$ for all $a, b, c \in R$.

Note that a ring does *not* require a multiplicative inverse nor does it require · to be commutative. (A ring with a multiplicative inverse for every non-0 element and with a commutative · is called a *field*, but that's a story for another day.)

The `Ring` *class*

We can represent a ring in Haskell using a typeclass. Unfortunately, we can't represent the laws (that would require dependent types, which is also a story for another day), but we can represent the operations, as follows:

```
class Ring a where
  addId  :: a            -- additive identity
  addInv :: a -> a       -- additive inverse
  mulId  :: a            -- multiplicative identity

  add :: a -> a -> a     -- addition
  mul :: a -> a -> a     -- multiplication
```

The idea here is that `a` is analogous to R, the set of elements of the ring. To help make this concrete, here is the `Ring` instance for integers:

```
instance Ring Integer where
  addId  = 0
  addInv = negate
  mulId  = 1

  add = (+)
  mul = (*)
```

(Those last two lines use operators without applying them to arguments. You can think of it almost as a section with no operands supplied. The first of the lines is equivalent to, for example, `add x y = x + y`.)

This instance declaration asserts that the integers form a ring with the given operations and identities.

Now that we have a `Ring` class, we can write operations that are generic with respect to which ring we are operating over. For examples of other rings, see the bottom of the Wikipedia page, or keep reading.

Parsing

One operation we would like to be able to write once for all rings is parsing expressions. Any ring supports the idea of expressions like

`a + (b * c) + d`, and so we should be able to write one parser (that is, function that converts from a string to a ring object) to produce them all.

Parsing is a little tricky, and we're not quite ready to write parsers yet for homework, so I've provided a parser in the `Parser.hs` module. You're welcome to read the module, but no attempt has been made to make this code readable to relative newcomers to Haskell — it's rather idiomatic advanced Haskell, though.

There's just one problem, though. The parser needs to be able to deal with so-called literals: the `a`, `b`, `c`, and `d` above. For example, if your ring is the integers, literals would look like `3` or `-2`. If your ring is 2×2 matrices, though, literals would look more like `[[1,2][8,-2]]`.

How to make the parser generic over different literal forms? Use a typeclass!

```
class Parsable a where
  -- | If successful, 'parse' returns the thing parsed along with the
  -- "leftover" string, containing all of the input string except what
  -- was parsed
  parse :: String -> Maybe (a, String)
```

The `parse` method looks at a string and tries to extract a specific element. If it can do so, it returns the element extracted and the *remainder* of the string. This way, parsing can continue. For example, we might define the `Parsable` instance for `Bool` this way:

```
instance Parsable Bool where
  parse str = case stripPrefix "True" str of
    Just trueRest -> Just (True, trueRest)
    Nothing       -> case stripPrefix "False" str of
      Just falseRest -> Just (False, falseRest)
      Nothing        -> Nothing
```

This uses the `stripPrefix` function from `Data.List`. That function tries to strip the given prefix (the first argument) from the given list (the second argument). If that list begins with the desired prefix, the prefix is stripped and the remaining list is returned. Otherwise, `stripPrefix` returns `Nothing`. The code above first tries to strip `"True"`. If it succeeds, then `parse` succeeds, returning `True` and the leftover string. Then, it tries the same for `"False"`.

But, Haskell provides a more idiomatic way of writing this:

```
instance Parsable Bool where
  parse str
    | Just rest <- stripPrefix "True"  str = Just (True,  rest)
```

```
    | Just rest <- stripPrefix "False" str = Just (False, rest)
    | otherwise                           = Nothing
```

This version uses a feature called *pattern guards*, which allow pattern matching in a guard (as introduced by a `|`). The expression to the right of the `<-` is matched against the pattern to the left of the `<-`. If that match succeeds, the guard is successful and the expression to the right of the `=` is evaluated. Otherwise, we move down to the next guard. You can mix pattern guards and normal Boolean guards freely.

Even better than that version is this one:

```
instance Parsable Bool where
  parse = listToMaybe . reads
```

This last version uses `reads`, a datatype parser provided as part of Haskell. This will work for any type that is a member of the `Read` type class, such as `Bool` or `Integer`. Feel free to look up these functions online to learn more.

The upshot of all of this is that you will have to define `Parsable` instances for any type that you want to be parsed.

Forging your own `Ring`*s*

Exercise 1 Homeworks are starting to get more complicated! Though we haven't covered any Haskell testing framework yet (`HUnit` is probably the simplest), it's time to start testing your code. For this assignment, every exercise should be accompanied by a few definitions that show us that your definitions work. For example, to show that the definitions for `Integer` work, I could have these:

```
intParsingWorks :: Bool
intParsingWorks = (parse "3" == Just (3 :: Integer, "")) &&
                  (parseRing "1 + 2 * 5" == Just (11 :: Integer)) &&
                  (addId == (0 :: Integer))
```

Note that I needed to add type signatures to my numbers to let GHC know that I wanted to talk about `Integer` — which has a `Ring` instance — and not about other number types, like `Int` or `Double`, which do not have `Ring` instances.

Now, I can just check that `intParsingWorks` is `True` in GHCi.

Make sure to include comments explaining how to use your testing definitions!

Exercise 2 Modular arithmetic forms a ring. We will be thinking of the integers modulo 5. This ring has 5 elements: $R = \{0, 1, 2, 3, 4\}$. Addition is like normal integer addition, but it wraps around. So, $3 + 4 = 2$ and $1 + 4 = 0$. Multiplication is like normal integer multiplication, but it, too, wraps around. Note that Haskell's `mod` function is very handy here!

Define a datatype

```
data Mod5 = MkMod Integer
  deriving (Show, Eq)
```

with `Ring` and `Parsable` instances. (Your `Parsable` instance should parse just like `Integer`'s.)

Test your instances!

Exercise 3 Matrix arithmetic forms a ring. Write a datatype `Mat2x2` (you choose the representation) and `Ring` and `Parsable` instances. Your parser must be able to read something like `[[1,2][3,4]]` as a 2×2 matrix. It does not need to allow for the posssibility of spaces. Writing this idiomatically in Haskell is hard, so we will be more forgiving about style in the matrix parser.

Test your instances!

Exercise 4 Boolean arithmetic forms a ring. Boolean-and (conjunction) is the multiplication operation, but Boolean-or is *not* the addition operation. What is? (There aren't too many choices here!) Write `Ring` and `Parsable` instances for `Bool`.

Test your instances!

One Ring to Rule Them All

Now that we can parse rings of all shapes and sizes, we want to start taking advantage of the ring laws. To do this, we will parse ring expressions into a custom datatype designed for manipulating ring expressions:

```
data RingExpr a = Lit a
                | AddId
                | AddInv (RingExpr a)
                | MulId
                | Add (RingExpr a) (RingExpr a)
                | Mul (RingExpr a) (RingExpr a)
  deriving (Show, Eq)

instance Ring (RingExpr a) where
```

```
  addId  = AddId
  addInv = AddInv
  mulId  = MulId

  add = Add
  mul = Mul

instance Parsable a => Parsable (RingExpr a) where
  parse = fmap (first Lit) . parse

  -- to understand this last function, here are types for 'fmap' and 'first':
  -- fmap  :: (a -> b) -> Maybe a -> Maybe b
  -- first :: (a -> b) -> (a, c) -> (b, c)
```

A `RingExpr a` holds ring expressions over a given ring `a`. (So, `RingExpr Integer` stores ring expressions over integers.) Because we have `Ring` and `Parsable` instances for `RingExpr a`, we can parse these expressions using our trusty `parseRing` function. Yay!

Having built an expression, we can then evaluate it using the underlying ring:

```
-- | Evaluate a 'RingExpr a' using the ring algebra of 'a'.
eval :: Ring a => RingExpr a -> a
eval (Lit a)    = a
eval AddId      = addId
eval (AddInv x) = addInv (eval x)
eval MulId      = mulId
eval (Add x y)  = add (eval x) (eval y)
eval (Mul x y)  = mul (eval x) (eval y)
```

Why bother with `RingExpr` at all? Because we can use it to *simplify* ring expressions according to the ring laws. Provided that `Ring` instances really obey the laws, these simplifications won't change the value retrieved by evaluating the ring expression.

As an example (bogus) simplification, we can write a function that swaps all additive identities with multiplicative identities. This surely changes the value of the expression, but it demonstrates the idea of traversing a `RingExpr a` and performing a transformation:

```
swapIdentities :: RingExpr a -> RingExpr a
swapIdentities AddId = MulId
swapIdentities MulId = AddId

-- need other cases to do this *everywhere* in the expression:
swapIdentities (Lit a)    = Lit a
swapIdentities (AddInv x) = AddInv (swapIdentities x)
```

```
swapIdentities (Add x y)  = Add (swapIdentities x) (swapIdentities y)
swapIdentities (Mul x y)  = Mul (swapIdentities x) (swapIdentities y)
```

Note the need for the cases that do not match against `AddId` or `MulId`. These are necessary because other forms of expression might contain `AddId` or `MulId` internally.

Exercise 5 Write `distribute` that distributes any use of multiplication over addition. Make sure to handle both left-distribution and right-distribution.

Test your function!

Exercise 6 Write `squashMulId` that detects whenever you are multiplying (on either side) by the multiplicative identity, and remove the multiplication. To get this working over parsed expressions is a little tricky, because the parser does not produce `MulId`. For example, in a `RingExpr Integer`, the multiplicative identity would look like `Lit 0`. Bonus brownie points[1] if you avoid using `eval`.

Test your function!

[1] These are not *real* points, but it would make us happy!

Exercise 7 *(Optional)* The `distribute` and `squashMulId` functions are quite similar, in that they traverse over the whole expression to make changes to specific nodes. Generalize this notion, so that the two functions can concentrate on just the bit that they need to transform.