More Haskell Basics

Prof. Susan Older

9 September 2019

Let's Write Some Functions!

As time permits, let's write these functions:

- `average :: Float -> Float -> Float`
  Accepts two numbers and calculates their average
- `allPos :: Integer -> Integer -> Integer -> Bool`
  Accepts three integers and determines whether they're all positive
- `someNeg :: Integer -> Integer -> Integer -> Bool`
  Accepts three integers and determines whether at least one is negative

Details You'll Need to Know to Keep the Interpreter Happy

- Rules/requirements for names/identifiers
- Rules/requirements for indentation
- Functions versus operators
- Overloading of names/operators

What's in a Name?

Identifiers (i.e., names) begin with a letter, and can then be followed by any combination of letters, digits, underscores (_), and single quotes (').

`x` `Number` `a123_xy` `alpha'''`

Three important rules

1. Names of functions and variables must begin with a lowercase letter.
2. Names of types must begin with an uppercase letter.

   Later on, we'll see: constructors, module names, and type classes also must begin with an uppercase letter.
3. Haskell is case sensitive: `abcdef` and `abcDef` are two distinct names.

Convention: When names are built from multiple words, the second and subsequent words are capitalized.

`celsiusToFahr`, `isTooHot`

Another Gotcha: Layout (Indentation Matters!)

Layout determines where definitions start and stop.

The Rule:

A definition ends at the first piece of text that lies at the same indentation as (or to the left of) the start at that definition.

Guidelines:

- For top-level definitions, start at the leftmost column.
- When writing definitions, use the same indentation for each.

  (Emacs can help you with this task.)

Calling Functions and Using Operators

When calling a function, the name appears before its arguments:

```
div 17 4
thrice (thrice 7)
isPostive (mystery 5 (mod 18 5))
```

Operators have two arguments and appear between those arguments:

```
6 * (3+4)
(mystery 6 7) < (thrice (8-2))
```

Parentheses are needed only when the result of a function call (or operator usage) is itself being passed to a function:

- `isPositive thrice 4` will cause a type error.
- `isPositive (thrice 4)` will work.
- `mystery 6 7 < thrice (8-2)` is okay.
- `mystery 6 7 < thrice 8-2` will cause a type error.

Some Caveats on Function Application

1. Function application binds more tightly than any other operators.

   `thrice n*3` is same as `(thrice n) * 3`, **not** `thrice (n*3)`.

2. The minus sign `-` gets used for both subtraction and negative numbers:

   `thrice -12` is same as `thrice (-) 12` (type error!).

   To apply `thrice` to `-12`, use parentheses: `thrice (-12)`.

3. Different operators have different binding power:

   `*` binds more tightly than `+` (just like in math!), which means:
   `3+4 * 6` is `3 + (4*6)`

Overloading: One Name, Multiple Meanings

We've seen that `==`, `+`, and `abs` have the following types (among others):

```
(==):: Bool -> Bool -> Bool
(==):: Integer -> Integer -> Bool
(==):: Float -> Float -> Bool

abs:: Integer -> Integer
abs:: Float -> Float

(+):: Integer -> Integer -> Integer
(+):: Float -> Float -> Float
```

- These are instances of overloading:

  The same name (or symbol) is used to represent different operations/functions on different types.

- Overloading provides a way to provide common naming for similar (but ultimately different) functions/operations.
- Haskell determines from context which definition is needed.
- Overloading is handled through type classes (a topic for the future).

Dealing with Cases: What to Do?

According to SU's Bursar, tuition for main-campus undergraduate students in 2019-20 is:

Per semester (12-19 credits)	$26105
Per credit (first 11)	$2274
Per credit (20 or more)	$1568

Let's write a Haskell function that:

- Accepts as input the number of credits being taken
- Calculates the tuition cost of that number of credits

Conditional Equations

Let's look at one solution:

```
tuition :: Integer -> Integer
tuition cr
    | cr <= 0    = 0
    | cr <= 11   = cr * 2274
    | cr >= 20   = cr * 1568
    | otherwise = 26105
```

1. There are four guards (all of which must have type `Bool`) :

   `cr <= 0` `cr <= 11` `cr >= 20` `otherwise`

2. There are four possible results (all of which must have type `Integer`):

   `0` `cr * 2274` `cr * 1568` `26105`

3. Evaluation rule: Return the result associated with the first guard that evaluates to `True` (`otherwise` always evaluates to `True`)

Conditional Equations: A Quick Quiz

Consider the following:

```
puzzler :: Integer -> Integer -> Integer
puzzler m q
    | odd q || m < 10  = m       -- || is or
    | even m && m > q  = q*2     -- && is and
    | otherwise        = m+q+1
```

What are the values of the following expressions?

- `puzzler 3 8` ⇝ `3`
- `puzzler 22 100` ⇝ `123`
- `puzzler 20 14` ⇝ `28`
- `puzzler 44 6` ⇝ `12`
- `puzzler 50 7` ⇝ `50`
- `puzzler 13.0 28` Type Error!

More Types: `Char`

- Sample values: `'a'`, `'A'`, `'3'`, `' '`, `'\n'` (newline), `'\t'` (tab)
- Comparison operators:

  `(==)`, `(/=)` :: `Char -> Char -> Bool`

  `(<)`, `(<=)`, `(>)`, `(>=)` :: `Char -> Char -> Bool`

- The module `Data.Char` contains lots of useful functions, including:

  `isAlpha` :: `Char -> Bool` `isUpper` :: `Char -> Bool`

  `isDigit` :: `Char -> Bool` `isLower` :: `Char -> Bool`

  `isAlphaNum` :: `Char -> Bool` `toUpper` :: `Char -> Char`

  `isControl` :: `Char -> Bool` `toLower` :: `Char -> Char`

  To use these functions, include the following at the top of your Haskell file:

  ```
  import Data.Char
  ```

More Types: `String`

- Strings are sequences of characters, enclosed with double quotes:

```
"hello!"
"1234"
""      (empty string)
"abc\ndefg\nh\ti"
```

- String concatenation:

```
(++) :: String -> String -> String
```

  Example: `"abc" ++ "1234"` evaluates to `"abc1234"`

- Later we'll see:

  `String` = `[Char]` (i.e., strings are lists of characters)