Haskell Programs

We’re covering material from Chapters 1-2 (and maybe 3) of the textbook.

A Haskell program is a series of comments and definitions.

- Each comment begins with -- (or appears between {- and -}) and serves as documentation.
- Each definition contains a type declaration and one or more equations:

\[
name :: t_1 -> t_2 -> \cdots -> t_n -> t
\]

\[
name \ x_1 \ x_2 \ \cdots \ x_n = exp
\]

- Each \( t_i \) is a type, each \( x_i \) is a formal parameter.
- The type declaration serves as a contract:
  - What the function expects as input (\( x_i \) has type \( t_i \))
  - What the function will deliver (\( exp \) has type \( t \))

What are Types?

Type = Collection of “similar” data values

Types are very important in Haskell:

- Every expression has a type.
- Types govern what/how we can combine.
- Nothing gets evaluated unless the types make sense.

Consider the following function:

\[
isPositive :: \text{Int} \rightarrow \text{Bool}
isPositive \ \text{num} = (\text{num} > 0)
\]

- \( isPositive \) expects an \( \text{Int} \) as input.
- \( isPositive \) returns a \( \text{Bool} \) as a result.

Some Very Basic Types

Haskell has lots of built-in types, including:

- \( \text{Bool} \)
  Boolean values: \text{True} and \text{False}
- \( \text{Int} \)
  Integer values from -2147483648 to 2147483647 (i.e., 32-bit integers)
- \( \text{Integer} \)
  All integers: full-precision integers
- \( \text{Float} \)
  Floating-point numbers, such as 3.267 or -81.09 or 12345.0

We’ll discuss these (and other types) in more detail later.
### Types: Simple Examples

```
thrice :: Int -> Int
thrice n = 3*n
```

```
isPositive :: Int -> Bool
isPositive num = (num > 0)
```

```
mystery :: Int -> Int -> Int
mystery x y = (thrice x) + y
```

What are the types of these expressions?

- `thrice 12 :: Int`
- `thrice False ... Type error`
- `isPositive (thrice 12) :: Bool`
- `mystery (thrice 12) 5 :: Int`

### Evaluating Expressions in Haskell

Idea: Based on rewriting equations (just like in algebra!)

- Happens after types are checked: type errors mean no evaluation
- Lazy evaluation: expressions evaluated only when values needed

```
thrice (5+2) \Rightarrow 3 \times (5+2) \Rightarrow 3 \times 7 \Rightarrow 21
```

```
isPositive (mystery (2-3) 1) \Rightarrow (mystery (2-3) 1) > 0
\Rightarrow ((thrice (2-3)) + 1) > 0
\Rightarrow ((3 \times (2-3)) + 1) > 0
\Rightarrow ((3 \times -1) + 1) > 0
\Rightarrow (-3 + 1) > 0
\Rightarrow (-2) > 0
\Rightarrow False
```

### Terminology: Formal Parameters and Actual Parameters

Consider the following function:

```
simple :: Int -> Int -> Int
simple a b = a + 3*b
```

- In the definition above, `a` and `b` are the **formal parameters** of `simple`.
  - They are names that appear in the function definition to represent the input that may be passed into the function.
- Suppose we evaluate `simple (4+2) 5`:
  - `(4+2)` and `5` are the **actual parameters** (a.k.a. arguments) of `simple`.
  - They are the expressions that appear in a function call/invocation.

What Are the Rules for Identifiers (i.e., Names)?

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

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

Two important rules:

- Names of functions and variables **must begin** with a lowercase letter.
- Names of types, type classes, constructors, and module names **must begin** with an uppercase letter.

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

```
milesToKm, weightPerUnit
```
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.)

Integers: **Int** and **Integer**

- **Int**: integers from -2147483648 to 2147483647 (i.e., 32-bit integers)
- **Integer**: all integers (i.e., full-precision integers)
- Standard operators and functions (also work for **Integer**)

  (+), (*), (-), div, mod, (^) :: Int -> Int -> Int

  even, odd :: Int -> Bool

  (==), (/=) :: Int -> Int -> Bool

  (<), (<=), (>), (>=) :: Int -> Int -> Bool

- You cannot add (or multiply or ...) an **Int** with an **Integer**.
- However, you can convert between **Int** and **Integer**:

  toInteger :: Int -> Integer  fromInteger :: Integer -> Int

More about **Bool**

- Exactly two values: **True**, **False**
- Standard operators:

  (&&) :: Bool -> Bool -> Bool  (==) :: Bool -> Bool -> Bool  (||) :: Bool -> Bool -> Bool  (/=) :: Bool -> Bool -> Bool

  - **e1** && **e2**: evaluates to **True** when both **e1** and **e2** evaluate to **True** (evaluates to **False** when either **e1** or **e2** evaluates to **False**)
  - **e1** || **e2**: evaluates to **True** when either **e1** or **e2** evaluates to **True** (evaluates to **False** when both **e1** and **e2** evaluate to **False**)
  - not **e**: evaluates to **True** when **e** evaluates to **False** (and vice versa)
  - == and /= are equality and inequality (respectively)

Floating-point Numbers: **Float**

- **Float**: single-precision floating-point numbers
  Examples include: **543.874**  **-346.2**  **12.0**
- Some standard operators and functions

  (+), (*), (-), /, (**) :: Float -> Float -> Float

  (==), (/=) :: Float -> Float -> Bool

  (<), (<=), (>), (>=) :: Float -> Float -> Bool

  ceiling, floor, round :: Float -> Integer

- More functions listed in Figure 3.2 of the textbook (page 58).
Let's Write Some Functions!

As time permits, let's write these functions:

- **average :: Float -> Float -> Float**  
  Accepts two numbers and calculates their average
- **allPositive :: Int -> Int -> Int -> Bool**  
  Accepts three integers and determines whether they're all positive
- **someNegative :: Int -> Int -> Int -> Bool**  
  Accepts three integers and determines whether at least one is negative

Dealing with Cases: What to Do?

According to SU’s Bursar, tuition for main campus undergrads is:

- Per semester (12-19 credits) $20,190
- Per credit (first 11) 1,757
- Per credit (20 or more) 1,214

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:

\[
\text{tuition :: Int -> Int} \\
\text{tuition cr} \\
\mid cr \leq 0 \quad = 0 \\
\mid cr \leq 11 \quad = cr \times 1757 \\
\mid cr \geq 20 \quad = cr \times 1214 \\
\mid \text{otherwise} \quad = 20190
\]

There are four guards (all of which must have type \texttt{Bool}):

- \texttt{cr \leq 0}
- \texttt{cr \leq 11}
- \texttt{cr \geq 20}
- \texttt{otherwise}

There are four possible results (all of which must have type \texttt{Int}):

- 0
- \texttt{cr \times 1757}
- \texttt{cr \times 1214}
- 20190

Evaluation rule: Return the result associated with the first guard that evaluates to \texttt{True}  
(Note: \texttt{otherwise} always evaluates to \texttt{True}.)

Consider the following:

\[
\text{contrived :: Int -> Int -> Int} \\
\text{contrived m n} \\
\mid \text{even m \&\& m > n} \quad = n \times 2 \\
\mid \text{odd n || n<3} \quad = m \\
\mid \text{otherwise} \quad = m+n+1
\]

What are the values of the following expressions?

- \texttt{contrived 100 7} \Rightarrow 14
- \texttt{contrived 16 100} \Rightarrow 117
- \texttt{contrived 321 7} \Rightarrow 321
- \texttt{contrived 44 0} \Rightarrow 0
- \texttt{contrived 95 0} \Rightarrow 95
- \texttt{contrived 28 15.0} Type Error!
Quick Side Discussion #1: Overloading

We’ve seen that == and + have the following types (among others):

- **(==)**: `Bool -> Bool -> Bool`
- **(==)**: `Int -> Int -> Bool`
- **(==)**: `Integer -> Integer -> Bool`
- **(==)**: `Float -> Float -> Bool`
- **(+)**: `Int -> Int -> Int`
- **(+)**: `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.

- Haskell determines from context which definition is needed.
- Overloading is handled through type classes (more on that later).

More Types: **Char**

- Sample values: 'a', 'A', '3', ', ', '
' (newline), '	' (tab)
- The module `Data.Char` contains lots of useful functions, including:

  - `isDigit :: Char -> Bool`
  - `isControl :: Char -> Bool`
  - `isAlpha :: Char -> Bool`
  - `toUpper :: Char -> Char`
  - `isAlphaNum :: Char -> Bool`
  - `toLowerCase :: Char -> Char`
  - `isUpper :: Char -> Bool`
  - `ord :: Char -> Int`
  - `isLower :: Char -> Bool`
  - `chr :: Int -> Char`

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

```haskell
import Data.Char
```

More Types: **String**

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

  "hello!"
  "1234"
  ""    (empty string)
  "abc\ndefg\n\nti"

- String concatenation:

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

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

- Later we’ll see:

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

Quick Side Discussion #2: Functions versus Operators

**When calling a function:**

The function name appears before its arguments:

```haskell```
```
  div 17 4
tuition 16
contrived 100 7
```haskell```

**When using operators (which always have two arguments):**

Operators appear between their two arguments:

```haskell```
```
  6 * (3+4)
  abc ++ "123xy"
```haskell```

If you want to treat operators like functions, put them in parentheses:

```haskell```
```
  (*) 6 (3+4)
  (+) 6 ((+) 3 4)
  (++) "abc" "123xy"
```haskell```

(CIS 252)  Haskell Fundamentals  21 January 2016  17 / 20