Haskell Basics

CIS 194 Week 1
14 January 2013

Suggested reading:

- Learn You a Haskell for Great Good, chapter 2
- Real World Haskell, chapters 1 and 2

What is Haskell?

Haskell is a *lazy, functional* programming language created in the late 1980's by a committee of academics. There were a plethora of lazy functional languages around, everyone had their favorite, and it was hard to communicate ideas. So a bunch of people got together and designed a new language, taking some of the best ideas from existing languages (and a few new ideas of their own). Haskell was born.

So what is Haskell like? Haskell is:

Functional

There is no precise, accepted meaning for the term "functional". But when we say that Haskell is a *functional* language, we usually have in mind two things:

- Functions are *first-class*, that is, functions are values which can be used in exactly the same ways as any other sort of value.
- The meaning of Haskell programs is centered around *evaluating expressions* rather than *executing instructions*.

Taken together, these result in an entirely different way of thinking about programming. Much of our time this semester will be spent exploring this way of thinking.

Pure

Haskell expressions are always *referentially transparent*, that is:

- No mutation! Everything (variables, data structures…) is *immutable*.
- Expressions never have "side effects" (like updating global variables or printing to the screen).
- Calling the same function with the same arguments results in the same output every time.

This may sound crazy at this point. How is it even possible to get anything done without mutation or side effects? Well, it certainly requires a shift in thinking (if you're used to an imperative or object-oriented paradigm). But once you've made the shift, there are a number of wonderful benefits:

- *Equational reasoning and refactoring*: In Haskell one can always "replace equals by equals", just like you learned in algebra class.
- *Parallelism*: Evaluating expressions in parallel is easy when they are guaranteed not to affect one another.
- *Fewer headaches*: Simply put, unrestricted effects and action-at-a-distance makes for programs that are hard to debug, maintain, and reason about.

Lazy

In Haskell, expressions are *not evaluated until their results are actually needed*. This is a simple decision with far-reaching consequences, which we will explore throughout the semester. Some of the consequences include:

- It is easy to define a new *control structure* just by defining a function.
- It is possible to define and work with *infinite data structures*.
- It enables a more compositional programming style (see *wholemeal programming* below).
- One major downside, however, is that reasoning about time and space usage becomes much more complicated!

Statically typed

Every Haskell expression has a type, and types are all checked at *compile-time*. Programs with type errors will not even compile, much less run.

Themes

Throughout this course, we will focus on three main themes.

Types

Static type systems can seem annoying. In fact, in languages like C++ and Java, they *are* annoying. But this isn't because static type systems *per se* are annoying; it's because C++ and Java's type systems are insufficiently expressive! This semester we'll take a close look at Haskell's type system, which

- *Helps clarify thinking and express program structure*

  The first step in writing a Haskell program is usually to *write down all the types*. Because Haskell's type system is so expressive, this is a non-trivial design step and is an immense help in clarifying one's thinking about the program.

- *Serves as a form of documentation*

  Given an expressive type system, just looking at a function's type tells you a lot about what the function might do and how it can be used, even before you have read a single word of written documentation.

- *Turns run-time errors into compile-time errors*

  It's much better to be able to fix errors up front than to just test a lot and hope for the best. "If it compiles, it must be correct" is mostly facetious (it's still quite possible to have errors in logic even in a type-correct program), but it happens in Haskell much more than in other languages.

Abstraction

"Don't Repeat Yourself" is a mantra often heard in the world of programming. Also known as the "Abstraction Principle", the idea is that nothing should be duplicated: every idea, algorithm, and piece of data should occur exactly once in your code. Taking similar pieces of code and factoring out their commonality is known as the process of *abstraction*.

Haskell is very good at abstraction: features like parametric polymorphism, higher-order functions, and type classes all aid in the fight against repetition. Our journey through Haskell this semester will in large part be a journey from the specific to the abstract.

Wholemeal programming

Another theme we will explore is *wholemeal programming*. A quote from Ralf Hinze:

> "Functional languages excel at wholemeal programming, a term coined by Geraint Jones. Wholemeal programming means to think big: work with an entire list, rather than a sequence of elements; develop a solution space, rather than an individual solution; imagine a graph, rather than a single path. The wholemeal approach often offers new insights or provides new perspectives on a given problem. It is nicely complemented by the idea of projective programming: first solve a more general problem, then extract the interesting bits and pieces by transforming the general program into more specialised ones."

For example, consider this pseudocode in a C/Java-ish sort of language:

```
int acc = 0;
for ( int i = 0; i < lst.length; i++ ) {
  acc = acc + 3 * lst[i];
}
```

This code suffers from what Richard Bird refers to as "indexitis": it has to worry about the low-level details of iterating over an array by keeping track of a current index. It also mixes together what can more usefully be thought of as two separate operations: multiplying every item in a list by 3, and summing the results.

In Haskell, we can just write

```
sum (map (3*) lst)
```

This semester we'll explore the shift in thinking represented by this way of programming, and examine how and why Haskell makes it possible.

Literate Haskell

This file is a "literate Haskell document": only lines preceded by > and a space (see below) are code; everything else (like this paragraph) is a comment. Your programming assignments do not have to be literate Haskell, although they may be if you like. Literate Haskell documents have an extension of `.lhs`, whereas non-literate Haskell source files use `.hs`.

Declarations and variables

Here is some Haskell code:

```
x :: Int
x = 3

-- Note that normal (non-literate) comments are preceded by two hyphens
{- or enclosed
   in curly brace/hyphen pairs. -}
```

The above code declares a variable `x` with type `Int` (`::` is pronounced "has type") and declares the value of `x` to be `3`. Note that *this will be the value of* `x` *forever* (at least, in this particular program). The value of `x` cannot be changed later.

Try uncommenting the line below; it will generate an error saying something like `Multiple declarations of `x'`.

```
-- x = 4
```

In Haskell, *variables are not mutable boxes*; they are just names for values!

Put another way, `=` does *not* denote "assignment" like it does in many other languages. Instead, `=` denotes *definition*, like it does in mathematics. That is, `x = 4` should not be read as "`x` gets `4`" or "assign `4` to `x`", but as "`x` is *defined to be* `4`".

What do you think this code means?

```
y :: Int
y = y + 1
```

Basic Types

```
-- Machine-sized integers
i :: Int
i = -78
```

`Int`s are guaranteed by the Haskell language standard to accommodate values at least up to 2^{29}, but the exact size depends on your architecture. For example, on my 64-bit machine the range is 2^{63}. You can find the range on your machine by evaluating the following:

```
biggestInt, smallestInt :: Int
biggestInt  = maxBound
smallestInt = minBound
```

(Note that idiomatic Haskell uses `camelCase` for identifier names. If you don't like it, tough luck.)

The `Integer` type, on the other hand, is limited only by the amount of memory on your machine.

```
-- Arbitrary-precision integers
n :: Integer
n = 123456789098765432198734098233498734987234987453 4

reallyBig :: Integer
reallyBig = 2^(2^(2^(2^2)))

numDigits :: Int
numDigits = length (show reallyBig)
```

For floating-point numbers, there is `Double`:

```
-- Double-precision floating point
d1, d2 :: Double
d1 = 4.5387
d2 = 6.2831e-4
```

There is also a single-precision floating point number type, `Float`.

Finally, there are booleans, characters, and strings:

```
-- Booleans
b1, b2 :: Bool
b1 = True
b2 = False

-- Unicode characters
c1, c2, c3 :: Char
c1 = 'x'
c2 = '∅'
c3 = ' '

-- Strings are lists of characters with special syntax
s :: String
s = "Hello, Haskell!"
```

GHCi

GHCi is an interactive Haskell REPL (Read-Eval-Print-Loop) that comes with GHC. At the GHCi prompt, you can evaluate expressions, load Haskell files with `:load` (`:l`) (and reload them with `:reload` (`:r`)), ask for the type of an expression with `:type` (`:t`), and many other things (try `:?` for a list of commands).

Arithmetic

Try evaluating each of the following expressions in GHCi:

```
ex01 = 3 + 2
ex02 = 19 - 27
ex03 = 2.35 * 8.6
ex04 = 8.7 / 3.1
ex05 = mod 19 3
```

```
ex06 = 19 `mod` 3
ex07 = 7 ^ 222
exNN = (-3) * (-7)
```

Note how ‘backticks‘ make a function name into an infix operator. Note also that negative numbers must often be surrounded by parentheses, to avoid having the negation sign parsed as subtraction. (Yes, this is ugly. I’m sorry.)

This, however, gives an error:

```
-- badArith1 = i + n
```

Addition is only between values of the same numeric type, and Haskell does not do implicit conversion. You must explicitly convert with:

- `fromIntegral`: converts from any integral type (`Int` or `Integer`) to any other numeric type.
- `round`, `floor`, `ceiling`: convert floating-point numbers to `Int` or `Integer`.

Now try this:

```
-- badArith2 = i / i
```

This is an error since `/` performs floating-point division only. For integer division we can use `div`.

```
ex08 = i `div` i
ex09 = 12 `div` 5
```

If you are used to other languages which do implicit conversion of numeric types, this can all seem rather prudish and annoying at first. However, I promise you’ll get used to it—and in time you may even come to appreciate it. Implicit numeric conversion encourages sloppy thinking about numeric code.

Boolean logic

As you would expect, Boolean values can be combined with `(&&)` (logical and), `(||)` (logical or), and `not`. For example,

```
ex10 = True && False
ex11 = not (False || True)
```

Things can be compared for equality with `(==)` and `(/=)`, or compared for order using `(<)`, `(>)`, `(<=)`, and `(>=)`.

```
ex12 = ('a' == 'a')
ex13 = (16 /= 3)
ex14 = (5 > 3) && ('p' <= 'q')
ex15 = "Haskell" > "C++"
```

Haskell also has `if`-expressions: `if b then t else f` is an expression which evaluates to `t` if the Boolean expression `b` evaluates to `True`, and `f` if `b` evaluates to `False`. Notice that `if`-*expressions* are very different than `if`-*statements*. For example, with an if-statement, the `else` part can be optional; an omitted `else` clause means "if the test evaluates to `False` then do nothing". With an `if`-expression, on the other hand, the `else` part is required, since the `if`-expression must result in some value.

Idiomatic Haskell does not use `if` expressions very much, often using pattern-matching or *guards* instead (see the next section).

Defining basic functions

We can write functions on integers by cases.

```
-- Compute the sum of the integers from 1 to n.
sumtorial :: Integer -> Integer
sumtorial 0 = 0
sumtorial n = n + sumtorial (n-1)
```

Note the syntax for the type of a function: `sumtorial :: Integer -> Integer` says that `sumtorial` is a function which takes an `Integer` as input and yields another `Integer` as output.

Each clause is checked in order from top to bottom, and the first matching clause is chosen. For example, `sumtorial 0` evaluates to `0`, since the first clause is matched. `sumtorial 3` does not match the first clause (`3` is not `0`), so the second clause is tried. A variable like `n` matches anything, so the second clause matches and `sumtorial 3` evaluates to `3 + sumtorial (3-1)` (which can then be evaluated further).

Choices can also be made based on arbitrary Boolean expressions using *guards*. For example:

```
hailstone :: Integer -> Integer
hailstone n
  | n `mod` 2 == 0 = n `div` 2
  | otherwise      = 3*n + 1
```

Any number of guards can be associated with each clause of a function definition, each of which is a Boolean expression. If the clause's patterns match, the guards are evaluated in order from top to bottom, and the first one which evaluates to `True` is chosen. If none of the guards evaluate to `True`, matching continues with the next clause.

For example, suppose we evaluate `hailstone 3`. First, `3` is matched against `n`, which succeeds (since a variable matches anything). Next, `n `mod` 2 == 0` is evaluated; it is `False` since `n = 3` does not result in a remainder of `0` when divided by `2`. `otherwise` is just an convenient synonym for `True`, so the second guard is chosen, and the result of `hailstone 3` is thus `3*3 + 1 = 10`.

As a more complex (but more contrived) example:

```
foo :: Integer -> Integer
foo 0 = 16
foo 1
  | "Haskell" > "C++" = 3
  | otherwise         = 4
foo n
  | n < 0             = 0
  | n `mod` 17 == 2   = -43
  | otherwise         = n + 3
```

What is `foo (-3)`? `foo 0`? `foo 1`? `foo 36`? `foo 38`?

As a final note about Boolean expressions and guards, suppose we wanted to abstract out the test of evenness used in defining `hailstone`. A first attempt is shown below:

```
isEven :: Integer -> Bool
isEven n
  | n `mod` 2 == 0 = True
  | otherwise      = False
```

This *works*, but it is much too complicated. Can you see why?

Pairs

We can pair things together like so:

```
p :: (Int, Char)
p = (3, 'x')
```

Notice that the `(x,y)` notation is used both for the *type* of a pair and a pair *value*.

The elements of a pair can be extracted again with *pattern matching*:

```
sumPair :: (Int,Int) -> Int
sumPair (x,y) = x + y
```

Haskell also has triples, quadruples, … but you should never use them. As we'll see next week, there are much better ways to package three or more pieces of information together.

Using functions, and multiple arguments

To apply a function to some arguments, just list the arguments after the function, separated by spaces, like this:

```
f :: Int -> Int -> Int -> Int
f x y z = x + y + z
exFF = f 3 17 8
```

The above example applies the function `f` to the three arguments `3`, `17`, and `8`. Note also the syntax for the type of a function with multiple arguments, like `Arg1Type -> Arg2Type -> ... -> ResultType`. This might seem strange to you (and it should!). Why all the arrows? Wouldn't it make more sense for the type of `f` to be something like `Int Int Int -> Int`? Actually, the syntax is no accident: it is the way it is for a very deep and beautiful reason, which we'll learn about in a few weeks; for now you just have to take my word for it!

Note that **function application has higher precedence than any infix operators**. So it would be incorrect to write

`f 3 n+1 7`

if you intend to pass `n+1` as the second argument to `f`, because this parses as

`(f 3 n) + (1 7)`.

Instead, one must write

`f 3 (n+1) 7`.

Lists

Lists are one of the most basic data types in Haskell.

```haskell
nums, range, range2 :: [Integer]
nums   = [1,2,3,19]
range  = [1..100]
range2 = [2,4..100]
```

Haskell (like Python) also has *list comprehensions*; you can read about them in LYAH.

Strings are just lists of characters. That is, `String` is just an abbreviation for `[Char]`, and string literal syntax (text surrounded by double quotes) is just an abbreviation for a list of `Char` literals.

```haskell
-- hello1 and hello2 are exactly the same.

hello1 :: [Char]
hello1 = ['h', 'e', 'l', 'l', 'o']

hello2 :: String
hello2 = "hello"

helloSame = hello1 == hello2
```

This means that all the standard library functions for processing lists can also be used to process `String`s.

Constructing lists

The simplest possible list is the empty list:

```haskell
emptyList = []
```

Other lists are built up from the empty list using the *cons* operator, `(:)`. Cons takes an element and a list, and produces a new list with the element prepended to the front.

```haskell
ex17 = 1 : []
ex18 = 3 : (1 : [])
ex19 = 2 : 3 : 4 : []

ex20 = [2,3,4] == 2 : 3 : 4 : []
```

We can see that `[2,3,4]` notation is just convenient shorthand for `2 : 3 : 4 : []`. Note also that these are really *singly linked lists*, NOT arrays.

```haskell
-- Generate the sequence of hailstone iterations from a starting number.
hailstoneSeq :: Integer -> [Integer]
hailstoneSeq 1 = [1]
hailstoneSeq n = n : hailstoneSeq (hailstone n)
```

We stop the hailstone sequence when we reach 1. The hailstone sequence for a general `n` consists of `n` itself, followed by the hailstone sequence for `hailstone n`, that is, the number obtained by applying the hailstone transformation once to `n`.

Functions on lists

We can write functions on lists using *pattern matching*.

```haskell
-- Compute the length of a list of Integers.
intListLength :: [Integer] -> Integer
intListLength []     = 0
intListLength (x:xs) = 1 + intListLength xs
```

The first clause says that the length of an empty list is 0. The second clause says that if the input list looks like `(x:xs)`, that is, a first element `x` consed onto a remaining list `xs`, then the length is one more than the length of `xs`.

Since we don't use `x` at all we could also replace it by an underscore: `intListLength (_:xs) = 1 + intListLength xs`.

We can also use nested patterns:

```haskell
sumEveryTwo :: [Integer] -> [Integer]
sumEveryTwo []          = []     -- Do nothing to the empty list
sumEveryTwo (x:[])      = [x]    -- Do nothing to lists with a single element
sumEveryTwo (x:(y:zs)) = (x + y) : sumEveryTwo zs
```

Note how the last clause matches a list starting with `x` and followed by… a list starting with `y` and followed by the list `zs`. We don't actually need the extra parentheses, so `sumEveryTwo (x:y:zs) = ...` would be equivalent.

Combining functions

It's good Haskell style to build up more complex functions by combining many simple ones.

```haskell
-- The number of hailstone steps needed to reach 1 from a starting
-- number.
hailstoneLen :: Integer -> Integer
hailstoneLen n = intListLength (hailstoneSeq n) - 1
```

This may seem inefficient to you: it generates the entire hailstone sequence first and then finds its length, which wastes lots of memory… doesn't it? Actually, it doesn't! Because of Haskell's lazy evaluation, each element of the sequence is only generated as needed, so the sequence generation and list length calculation are interleaved. The whole computation uses only O(1) memory, no matter how long the sequence. (Actually, this is a tiny white lie, but explaining why (and how to fix it) will have to wait a few weeks.)

We'll learn more about Haskell's lazy evaluation strategy in a few weeks. For now, the take-home message is: don't be afraid to write small functions that transform whole data structures, and combine them to produce more complex functions. It may feel unnatural at first, but it's the way to write idiomatic (and efficient) Haskell, and is actually a rather pleasant way to write programs once you get used to it.

A word about error messages

Actually, six:

Don't be scared of error messages!

GHC's error messages can be rather long and (seemingly) scary. However, usually they're long not because they are obscure, but because they contain a lot of useful information! Here's an example:

```
Prelude> 'x' ++ "foo"

<interactive>:1:1:
    Couldn't match expected type `[a0]' with actual type `Char'
    In the first argument of `(++)', namely 'x'
    In the expression: 'x' ++ "foo"
    In an equation for `it': it = 'x' ++ "foo"
```

First we are told "Couldn't match expected type `[a0]` with actual type `Char`". This means that *something* was expected to have a list type, but actually had type `Char`. What something? The next line tells us: it's the first argument of `(++)` which is at fault, namely, `'x'`. The next lines go on to give us a bit more context. Now we can see what the problem is: clearly `'x'` has type `Char`, as the first line said. Why would it be expected to have a list type? Well, because it is used as an argument to `(++)`, which takes a list as its first argument.

When you get a huge error message, resist your initial impulse to run away; take a deep breath; and read it carefully. You won't necessarily understand the entire thing, but you will probably learn a lot, and you may just get enough information to figure out what the problem is.