# Simple Functional Programming in Haskell

Winter 2015-2016

# Table of Contents

# Haskell: Defining Functions

- Programs are collections of functions.
- A function is a mapping from one set (domain) to another set (range).

**Math**

$$f : \Re \to \Re$$
$$f(x) = x^2 - 1$$

**Haskell**

```haskell
f :: Float -> Float
f x = x * x  - 1
```

# Haskell: Computing with Functions

- Computation is evaluation of expressions

$$f(3) = 3 \times 3 - 1 = 8$$

- In the Haskell interactive system:

```
Main> f 3
8
```

# Type signature and definition

```haskell
-- return the square of an integer

square :: Integer -> Integer
square x = x * x
```

- -- begins a comment line
- Type signature
- Function equation

= is the equation operator

it is not assignment

- The LHS is defined to be equivalent to the expression on the RHS

# Bindings

```haskell
-- return the square of an integer

square :: Integer -> Integer
square x = x * x
```

A definition **binds** names to values.

- square is bound to a function
- x is the formal name of the argument given to the function
- A set of bindings is called an environment

# Function evaluation

```
Main> square 3
9
Main> square 5678
32239684
Main> square (square 5678)
1039397224419856
```

An expression is evaluated by

- looking up names in the environment,
- replacing the names with their values,
- carrying out the needed operations.

- Haskell is lazy: an expression is evaluated only if its value is needed.

# A function with multiple inputs

```
-- return the smaller of a pair of Integers

smaller :: (Integer, Integer) -> Integer
smaller (x,y) = if (x < y) then x else y
```

- The name smaller is bound to a function that takes a pair of Integers, and returns an Integer
- The pair notation is a container; this is one argument with 2 sub-components.
- The **if** e1 **then** e2 **else** e3 syntax:
  - Force evaluation of e1; if **True** return e2, else return e3
  - In Java/C/C++: e1 ? e2 : e3

# A function with multiple arguments

```
-- return the smaller of a pair of Integers

smaller2 :: Integer -> Integer -> Integer
smaller2 x y = if (x < y) then x else y
```

- The name smaller2 is bound to a function that takes 2 Integers, and returns an Integer
- Each parameter is handled separately; they are not a single unit.

# How to work with The Glasgow Haskell Compiler

- Start Haskell on Lab machines by typing "ghci"
- The GHC is part of "The Haskell Project"
  `http://www.haskell.org/`
- Includes a compiler (ghc), an interactive system (ghci), libraries and applications.

# How to work with The Glasgow Haskell Compiler

```
savoy[1]% ghci
GHCi, version 6.10.4: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim \ldots linking \ldots done.
Loading package integer \ldots linking \ldots done.
Loading package base \ldots linking \ldots done.
Prelude> :l functional
[1 of 1] Compiling Main             ( functional.hs, interpreted )
Ok, modules loaded: Main.
*Main> square 3
9
*Main> square (square 3)
81
```

# A C programmer's first factorial function in Haskell

```
factorial :: Integer -> Integer
factorial n =
  if (n == 0) then
    1
  else
    n * factorial (n - 1)
```

- If-then-else is familiar, but this is very ugly.
- Important: White space for indentation is significant.

# Guarded equations for multiple cases

```haskell
factorial3 :: Integer -> Integer
factorial3 n | n == 0    = 1
             | otherwise = n * factorial3 (n - 1)
```

Compare this to a definition you might find in a textbook:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times f(n-1) & \text{otherwise} \end{cases}$$

# Using patterns

```
factorial6 :: Integer -> Integer
factorial6 0 = 1
factorial6 n = n * ( factorial6 (n-1))
```

- Patterns are checked in order
- Patterns with literal values are tests of equality
- Patterns with names bind the name to the given value

# Using local definitions

```haskell
factorial6 :: Integer -> Integer
factorial6 0 = 1
factorial6 n = n * fnm1
  where fnm1 = factorial6 (n-1)
```

- The where-clause creates a local binding.
- Values and functions can be defined locally.

# Newton's method in Haskell

```haskell
-- Square root: simple recursive version

sqrt1 :: Double -> Integer -> Double
sqrt1 x n | n == 1      = 1
          | otherwise   = (y + x/y)/2
              where y   = sqrt1 x (n-1)
```

Compare to:

$$
\begin{aligned}
y_1 &= 1 \\
y_n &= \frac{y_{n-1} + \frac{x}{y_{n-1}}}{2}
\end{aligned}
$$

# Computing square roots to within given accuracy

```
-- Flexible square root

sqrt5 :: Double -> Double -> Double
sqrt5 eps x = sqh x
  where sqh y | abs (x - y*y) < eps = y
              | otherwise           = sqh ((y + x/y)/2)
```

- Local definition of recursive function with guarded equations.
- Note: The inner function uses $x$ which is bound by the outer function.

# Expressions that evaluate to functions

```haskell
sqrt_loose :: Double -> Double
sqrt_loose = sqrt5 0.5

sqrt_close :: Double -> Double
sqrt_close = sqrt5 0.0001
```

- Haskell functions are values too!
- A function call that does not have all the needed arguments creates a new function!

# Sending functions as values to functions

```haskell
-- radical square root

squareroot :: Double -> Double -> Double

squareroot eps x = until closeEnough improveGuess 1
  where improveGuess y = (y + x/y)/2
        closeEnough u = abs (x - u*u) < eps
        until p f x = if (p x) then x else until p f (f x)
```

Haskell's compiler contains a very powerful type checking system.
This allows programmers to define very powerful functions.

# Find the root of any function $g$

```
-- radical root

root :: Double -> (Double -> Double) -> Double

root eps g = until closeEnough improveGuess 1
  where g' x = (g (x+eps) - g x)/eps
        improveGuess y = y - (g y)/(g' y)
        closeEnough u = abs (g u) < eps
```

- **until** is defined by the Haskell prelude.
- There are restrictions on $g$; can you deduce what restrictions must apply to $g$?

# Specializing the generic root function

```haskell
-- sqrt to 5 decimal places
sqrt5 a = root 0.00001 (sq a)
  where sq a x = x*x - a

-- cube root to 5 decimal places
cubert a = root 0.00001 (cu a)
  where cu a x = x*x*x - a

-- rewriting sqrt5 using function composition
sqrt6 = (root 0.00001) . sq
  where sq a x = x*x - a
```

# Haskell: Features Overview

- Purely Functional
- Strongly typed
- Polymorphic
- Lazy Evaluation
- Modern

# Haskell: Purely Functional

## Purely functional

The value of an expression depends only on the components of the expression.

- This allows programmers to use functions in powerful ways.
- Side-effects like I/O are localized and cannot have hidden effects on other functions.
- No mutation, pointers
- Functions are "first-class" values

# Haskell: Strongly typed

## Strongly typed

The type of an expression can be inferred from the expression itself.

- A Haskell program that compiles has no "silly bugs."
- Type signatures, eg `f :: Float -> Float`, are a key part of the function definition
- Haskell's type system uses a powerful inference engine

# Haskell: Polymorphic

## Polymorphic function

A function that can be applied to data of many different types

- Polymorphism is achieved through use of type variables and type conditions.
- Code reuse: write one function, use many times

# Haskell: Lazy Evaluation

## Lazy evaluation

An expression is only evaluated when the value is needed.

- If the value of an expression is not needed yet, its evaluation is delayed.
- Familiar examples from other languages:
  - **if** . . .**then**. . .**else**. . .
  - . . .**and**. . .
  - . . .**or**. . .
  - These are special cases for other languages.
- In Haskell, expressions are lazy by default. It is not a special case.

# Haskell: Purely Functional

- Functions are first-class objects
  - Can be easily passed as arguments to other functions
  - Can be easily created "on the fly"
  - Can have names, but can also be anonymous
- Benefits:
  - No bugs due to side effects, pointers
  - Functional languages may aid programmer productivity
  - Correctness is much easier to prove
- The challenge: New ways to think about putting programs together.

# Haskell: Modern

- Language features designed for programmer productivity (not the compiler's productivity)
- Good interactive compilers – easy to test functions
- Can compile to native code (stand-alone)
  The GHC compiler produces better code than C++ compilers on some benchmarks

# General Syntactic Information

- Haskell syntax was designed to be clean and sparse.
- White space used for indentation is significant; white space outside of indentation is not significant.
- Two basic components to Haskell programs:

## Types

- Types, datatypes, polymorphic data types, classes, instances, constructors
- Identifiers are always Capitalized

## Functions

- Signatures, definitions, names, variables
- Identifiers are always lower case

# Basic Haskell Types

- booleans: **Bool** ; values: **True**, **False**
- integers: **Int**, **Integer**
- floating point numbers: **Float**, **Double**
- characters: **Char**: values: eg, 'a', 'b'
- strings: **String** : eg, "abc", ""

# Every expression has a type

- You can declare a type for any value you define

```
pi :: Float
pi = 3.1415926

lucky :: Int
lucky = 7

myname :: String
myname = "Inigo Montoya"
```

The type expression operator `::` is pronounced "is of type"

- (remember, the `=` creates a binding
IT IS NOT AN ASSIGNMENT STATEMENT.)

# Function type signatures

- A function maps input type(s) to an output type
- The type operator `->` is pronounced "to"
- A function's type has at least one `->`

```haskell
square :: Integer -> Integer
square a = a * a
```

"The identifier `square` is of type **Integer** to **Integer**."

# Functions of 1 argument

```
letterGrade :: Integer -> Char
letterGrade grade
  | grade < 0      = error "negative grade"
  | grade < 50     = 'F'
  | grade < 60     = 'D'
  | grade < 70     = 'C'
  | grade < 80     = 'B'
  | grade <= 100   = 'A'
  | grade > 100    = error "grade too high"
```

# Functions of 2 arguments

- Functions of 2 arguments have 2 `->`

```
-- return the maximum
maxOf2 :: Int -> Int -> Int
maxOf2 x y
   | x > y        = x
   | otherwise    = y
```

- Reading from left to right, the two input types and the output type

# The `->` operator

- The operator `->` associates to the right:

| | | |
|---|---|---|
| **Int -> Int -> Int** | $=$ | **Int -> (Int -> Int)** |
| | $\neq$ | **(Int -> Int) -> Int** |

# Functions of 3 arguments

- Functions of 3 arguments have 3 ->

```
quad :: Float -> Float -> Float -> Float
-- calculate one root of the
-- quadratic equation:
-- a*x*x + b*x + c = 0

quad a b c | a == 0      = error "not quadratic"
           | disc < 0    = error "not real"
           | otherwise   = (-b + (sqrt disc))/(2 * a)
             where disc  = b*b - 4*a*c
```

Three input types, finally the output type.

# Strong typing

- Every syntactically correct expression has an unambiguous type.
- The type of an expression can be deduced from the expression itself.
- Any expression whose type is ambiguous or inconsistent is rejected as incorrect!
- A type error in a simple program is usually a typo
- A type error in a sophisticated program means your design has a bug: you are passing the wrong information around!

# Haskell is able to deduce type signatures

```
f x | x == 'a'    = 1
    | x == 'b'    = 1
    | otherwise   = 1
```

- The syntax tells Haskell that `f` is a function
- Let's assume `1` is an **Int**
- The guards compare input `x` to a **Char**
- The function outputs `1`
- So `f :: Char -> Int` is the type of this function.

# Inconsistent definitions

```
g x | x == 'a'  = 1
    | x == 1    = 'a'
```

- The two equations are not consistent about types
  1. The first equation suggests `g :: Char -> Integer`
  2. The second equation suggests `g :: Integer -> Char`
- Haskell rejects the definition:

```
No instance for (Num Char)
  arising from the literal '1' at badg.hs:2:15
Possible fix: add an instance declaration for (Num Char)
```

- Loose Translation: "You tried to use a Char where the compiler expected a number"
- The given "possible fix" is nonsensical for this example.

# The Gravitational Force Example

- Gravity is a force between any 2 masses separated by a distance:
$$force(M_1, M_2, r) = \frac{G \times M_1 \times M_2}{r^2}$$
- Gravitational field is the effect of a particular mass (eg., the earth) on any other mass (eg., an apple) at any distance
- Weight is the effect of a particular mass (eg., the earth) at a particular distance (eg., the surface of the earth) on any other mass

# Haskell implementation

```haskell
-- gravitational constant for our universe
gravConstant :: Float
gravConstant = 6.67e-11  -- units: N m**2/kg**2

-- the force of gravity

gravForce :: Float -> Float -> Float -> Float
gravForce mass1 radius mass2
  | radius == 0 = 0
  | otherwise   = gravConstant*mass1*mass2/square radius
```

gravForce is useful because it can be used many ways

# Giving 3 arguments to gravForce

Applied with 3 arguments, it evaluates the force of gravity between 2 masses separated by a distance

```
Main> gravForce 1 1 1
6.67e-011

Main> gravForce earthMass earthRadius 68
666.196
```

The names earthMass and earthRadius were defined but not shown.

# Giving only 2 arguments to gravForce

Applied with 2 arguments, a new function of the remaining argument is created.

```
weightOnEarth :: Float -> Float
weightOnEarth = gravForce earthMass earthRadius
```

- This name is bound to a function created by the expression on the right side of the equation.
- The value not given to gravForce is the mass of any object
- weightOnEarth calculates the weight of any object, as measured on Earth, given its mass.

# weightOnEarth is really a function

```
Main> :type weightOnEarth
weightOnEarth :: Float -> Float

Main> weightOnEarth 30
293.9101

Main> weightOnEarth 100
979.70026

Main> weightOnEarth 68
666.19617
```

# Giving only 1 argument to gravForce

Applied with 1 argument, a new function of the other 2 arguments is created.

```
sunGravity :: Float -> Float -> Float
sunGravity = gravForce sunMass
```

- This name is bound to a function created by the expression on the right side of the equation.
- The values not given to gravForce are the distance and mass of the other object.
- sunGravity is a function that represents the gravitational field created by the sun.

# sunGravity is really a function

```
Main> :type sunGravity
sunGravity :: Float -> Float -> Float

Main> sunGravity 1e10 100
132.733

Main> sunGravity 3.7e9 68
659.302
```

# Boring astronomical facts

```
earthMass :: Float
earthMass = 5.96e24  -- units: kg

earthRadius :: Float
earthRadius = 6.37e6 -- units: m

sunMass :: Float
sunMass = 1.99e30 -- units: kg
```

# Lazy evaluation

- Consider the function

```
seven x = 7
```

- Haskell is lazy: the value for $x$ is not needed, so it is not evaluated.
- Consider the following function, which is an infinite loop:

```
dumb y = 1 + (dumb y)
```

- Using lazy evaluation:

```
Main> dumb 0
^C Interrupted.
Main> seven (dumb 0)
7
```