Functions over Numbers

Module 1: Part 1

Slides by Sascha Strand

Getting started

- Haskell can be interpreted or compiled.
- We’ll use the Glasgow Haskell Compiler toolset for both of these
- On the linux lab machines, this can be invoked with the command `ghc`
- To enter an interpreter session, you can type `ghci`
- To run a program through the interpreter, you can use `runghc`
- If you want to run Haskell on your own machine:
  - you can download the ghc toolset at https://www.haskell.org/ghc/ (~ 500 MB)
  - the “Haskell platform” at https://www.haskell.org/platform/ includes the same tools and more

Running code

- One-line expressions can be run in the interpreter but multi-line function definitions require special syntax
- While learning the normal syntax, I recommend writing your code in a separate file like, `eg.hs`
- And loading eg.hs in ghci by either opening it directly `ghci eg.hs`
- Or loading it in an open interpreter session
  `Prelude> :l eg.hs`

Let’s check out some code

- If you open ghci, you’ll see
  ```
  Prelude>
  ```
- The prompt will grow to show all the modules loaded in the current environment
- The Prelude is a built-in set of tools defined in the Haskell 98 standard

Function application

- Function application has the highest precedence in the language, so parenthese are optional when the number of language elements following the function match its arguments

```
Prelude> min 9 10
9
Prelude> min 8 9 + max 7 8

16
Prelude> min 8 (max 2 3)

3

Prelude> min 8 max 2 3
Error: Data constructor not in scope: ...
```

Function definitions

Functions are the mapping of an input to an output.

```
mult3 x = x * 3
```

For different inputs, they can be defined for each input.

```
fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

There is no explicit return statement, because the right-hand side of the function definition *is* what is returned. We can think of it like there is only one return value, and that's the result of evaluating the right-hand side with its arguments.

Control flow

- Haskell includes an if-then-else structure. White space is not syntactically meaningful in an if-then-else statement, but every if-then must have an else.

```
profitOrNo income expense = if income > expense
                              then "Profit"
                              else "No profit"
```

- They can be nested as well

```
profitOrLoss income expense = if income > expense
                                then "Profit"
                                else if income == expense
                                    then "Break even"
                                    else "Loss"
```

Logical Operators

`True False`	True and False values
`== /=`	equality & inequality
`not`	logical not
`|| &&`	logical or, and (short circuit)

Source

Note about if-then-else

- In module 2 (weeks 3 and 4) we’ll see syntax that can express the same semantics as nested if-then-else statements but in a cleaner and ultimately more powerful way.
- This syntax is called a guard. For now though, we can make due with if-then-else.

Constants

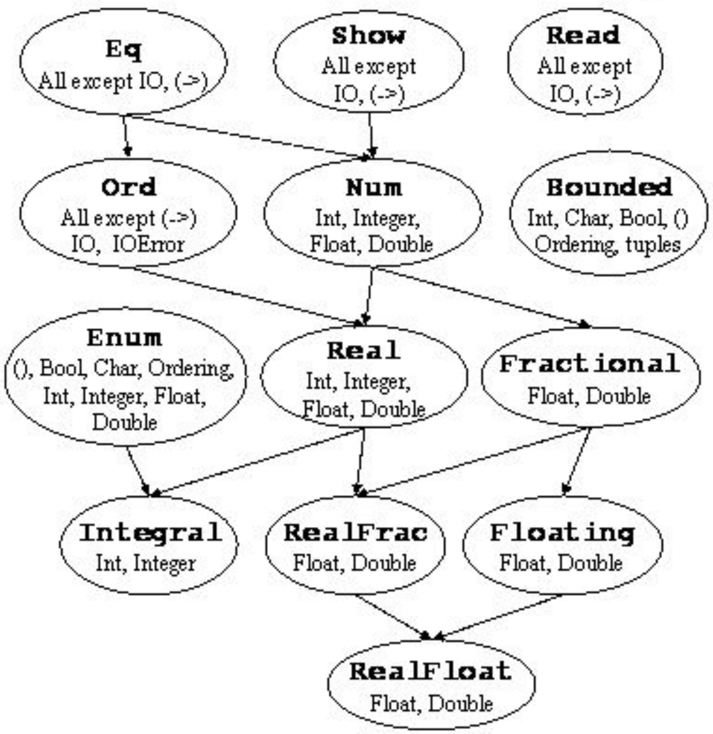
- There are no variables in a purely functional language, but there are constants.
- They can be assigned once using the `let` syntax, and they do not change.
- Constants are not strictly necessary, but they can aid in readability.
- If you use `let` in a function, it must be followed by the `in` keyword

```
verboseAbs x y =
    let outputSentence = "The absolute value is "
    in if x - y > 0
        then outputSentence ++ show (x - y)
        else outputSentence ++ show (y - x)
```

- The `show` and `++` in this example convert an integer value to a string and concatenate it.

A brief introduction to types

- Types are organized into typeclasses that have an inheritance relationship to one another.
- Each node in the tree is a typeclass containing types.
- A node below another node is an instance of its parent.
- These are a few types that are built into the Prelude.



source

Reading a type signature

- All expressions and functions have a type in Haskell
- In ghci, you can find the type of anything by using the :t command

```
Prelude> :t 3
3 :: Num p => p
Prelude> :t sqrt
sqrt :: Floating a => a -> a
```

- `3 :: Num p => p` can be read “For all Num types p, the type of 3 is p.” So the type of 3 is a member of the Num or numeric typeclass.
- `sqrt :: Floating a => a -> a` tells us “For all Floating types a, sqrt is a function that takes a type a and returns a type a.” So sqrt takes any type in the Floating typeclass and returns a type also in the Floating typeclass.

Numeric types

Type	Class	Description
`Integer`	`Integral`	Arbitrary-precision integers
`Int`	`Integral`	Fixed-precision integers
`Float`	`RealFloat`	Real floating-point, single precision
`Double`	`RealFloat`	Real floating-point, double precision

Operators on numeric types

`+, -, *, /`	addition, subtraction, multiplication, division
`logBase b`	logarithm (base b)
`** ^`	exponentiation
`rem`	C-style modulo eg. `rem (negate 3) 2 = -1`
`mod`	Distance from zero modulo eg. `mod (negate 3) 2 = 1`
`negate`	negation eg. `negate 3 = -3`
`sqrt, abs`	square root, absolute value
`<, >, <=, >=`	comparison
`min, max`	min or max of two elements

Source

Converting between numeric types

- To start working with functions over numbers and dig in to the type system, the lab today requires converting some numbers between types.
- You can find more information about how conversion function works in the source link.

Source

Converting from and between integral types

```
fromIntegral :: (Num b, Integral a) => a -> b
fromInteger :: Num a => Integer -> a
toInteger :: Integral a => a -> Integer
```

- `fromIntegral` takes a value with a type in the Integral typeclass and returns a value with a type in the Num typeclass.
  - Eg. Will convert an Int or Integer for use in a function expecting a Float or Double
- `fromInteger` takes an Integer and returns a value with a type in the Num typeclass
  - Eg. Will convert an Integer to a function expecting an Int, Float, or Double
- `toInteger` takes a value with a type belonging to the Integral typeclass and returns an Integer
  - Eg. Will convert an Int to an Integer.

Source

Converting from real types

```haskell
fromIntegral :: (Num b, Integral a) => a -> b
```

- `realToFrac` takes a value with a type in the Real typeclass and converts it to a value with type in the Fractional typeclass.
- Eg. Int or Integer to Float or Double

Converting from real-fractional numbers to integral numbers

```
ceiling  :: (RealFrac a, Integral b) => a -> b
floor    :: (RealFrac a, Integral b) => a -> b
truncate :: (RealFrac a, Integral b) => a -> b
round    :: (RealFrac a, Integral b) => a -> b
```

- All four of these functions will take a Float or a Double (the only two types we've seen in the RealFrac typeclass) and return a value with a type in the Integral typeclass.
    - Eg. Will take a float or double and convert it for use in a function that expects an Int or Integer

Some other handy syntax

`{- … -}`	block comment (nestable)
`--`	line comment
`show`	convert something to a string
`putStr`	print string
`putStrLn`	print string with nl
`printf`	print string (printf-like)

Source

Finally

- Please ask questions whenever they come up for you.
- Review lab exercises
- Review of credit options:

Without Lab:

Midterm Exam	20%
Final Exam	20%
Assignments	40%
Discussion Questions	20%

With Lab:

Midterm Exam	15%
Final Exam	15%
Assignments	30%
Discussion Questions	20%
Lab	20%