Session 2

Basic types and function definitions

Where we are

last session (chapters 1 and 2)

- introducing functional programming
- getting started with the GHCi environment.

this session (chapters 3 and 4.1)

- the basic types of Haskell
- defining simple functions by cases
- *Aim:* write functions in Haskell manipulating the basic types

next session (sections 5.2 and 5.3)

- built-in record types (tuples)
- user-defined record types
- polymorphism: functions on tuples and records

Revision

Source files contain modules with definitions

```haskell
module Week1 where

square :: Integer -> Integer
square n = n*n

norm :: Double -> Double -> Double
norm x y = sqrt (x*x + y*y)
```

At the GHCi prompt, we can evaluate expressions: **interpreter**

```
:load Week1
square (3+4)
:type square (3+4)
```

Semantics of a function definition

Given a function definition like

```haskell
norm :: Double -> Double -> Double
norm x y = sqrt (x*x + y*y)
```

To evaluate an expression like `norm 3.0 4.0`

- Match the formal parameter variables with the values passed as actual parameters: `x` = `3.0`, `y` = `4.0`.
- Substitute those values into the righthand side expression: `sqrt (3.0*3.0 + 4.0*4.0)`.
- Evaluate that expression:

```
norm 3.0 4.0  ⇝  sqrt (3.0*3.0 + 4.0*4.0)
              ⇝  sqrt (9.0 + 16.0)
              ⇝  sqrt 25.0
              ⇝  5.0
```

Numbers

The basic types of Haskell

Type	*Values of the type*
`Bool`	`False`, `True`
`Char`	`'a'`, `'@'`, `'0'`, `'\n'`, `'\''`, ...
`Int`	`minBound`, ..., −2, −1, 0, 1, 2, ..., `maxBound`
`Integer`	..., −2, −1, 0, 1, 2, ...
`Float`	`-3.1`, `0.005`, `12.3`, `12.3e40`, `12.3e-40`, ...
`Double`	`-3.1`, `0.005`, `12.3`, `12.3e40`, `12.3e-40`, ...

Things to try:

interpreter

```haskell
minBound::Int
maxBound::Int
3.2^50
0.3^50
```

Syntax details: infix operators

Haskell permits the usual infix notation:

interpreter

```haskell
1 + 2*5
```

Here the infix operators `+` and `*` refer to functions that take two arguments.

Operators have precedence and associativity (like in Java): more later.

To refer to such functions by themselves, we wrap them in parentheses, writing `(+)` and `(*)`:

Prelude

```haskell
(+) :: Int -> Int -> Int
(*) :: Int -> Int -> Int
```

So we could write

interpreter
```
(*) 2 5
(+) 1 ((*) 2 5)
```

Using binary functions as infix operators

If we have an ordinary identifier that refers to a binary function, e.g.

Prelude
```
div :: Int -> Int -> Int
mod :: Int -> Int -> Int
```

we can use it as an infix function by wrapping it in backquotes:

interpreter
```
(1987 `div` 100) `mod` 4
```

This is equivalent to

interpreter
```
mod (div 1987 100) 4
```

Functions on `Int`

Prelude
```
(+)    :: Int -> Int -> Int
(-)    :: Int -> Int -> Int
(*)    :: Int -> Int -> Int
(^)    :: Int -> Int -> Int
div    :: Int -> Int -> Int
mod    :: Int -> Int -> Int
abs    :: Int -> Int
negate :: Int -> Int
```

Try the last two on a few numbers to see what they do:

interpreter
```
abs 3
negate 3
abs (negate 3)
negate (negate 3)
abs 0
negate 0
```

Functions on `Double`

Prelude
```
(+)    :: Double -> Double -> Double
(-)    :: Double -> Double -> Double
(*)    :: Double -> Double -> Double
(/)    :: Double -> Double -> Double
(^)    :: Double -> Int -> Double
abs    :: Double -> Double
```

```
negate :: Double -> Double
sin    :: Double -> Double
asin   :: Double -> Double
exp    :: Double -> Double
log    :: Double -> Double
sqrt   :: Double -> Double
```

Try some experiments. Does `sin` work on degrees or radians? What base does `log` use?

Overloading through constrained types

More overloading

Some numeric functions are overloaded:

Prelude

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

Indeed numeric literals are also overloaded:

```
123 :: Int
123 :: Float
```

In fact, these have more general types:

interpreter

```
:t (+)
:t 123
```

How to read an overloaded type

Consider a type like

Prelude

```
(+) :: Num a => a -> a -> a
```

- Ignoring the `Num a =>` part for a moment, we see this is a function that takes two arguments, both of the same type `a`, and returns something of the same type.
- The `Num a =>` part is a constraint, saying that this type `a` must belong to the `Num` class, *i.e.* that it is a numeric type. Numeric types include `Int`, `Integer`, `Float` and `Double`.
- A class like `Num` is not a type, but a *set* of types.
- Other numeric classes are `Integral` (including `Int` and `Integer`) and `Floating` (including `Float` and `Double`).

interpreter

```
:t mod
:t sqrt
```

No implicit conversions

Consider an expression like

interpreter

```
1 + 2.3
```

- In Java **`1`** has type **`int`** and **`2.3`** has type **`double`**, so **`1`** is implicitly converted from **`int`** to **`double`**, and the two are added as **`double`**s.
- In Haskell, both **`1`** and **`2.3`** have type **`Double`** (or both have type **`Float`**), and so does the result.

Conversions between types

Haskell has no implicit conversion between types.
You must use explicit functions:

$$\begin{tikzcd}
\texttt{Char} \arrow[r, shift left, "\texttt{ord}"] & \texttt{Int} \arrow[l, shift left, "\texttt{chr}"] \arrow[r, shift left, "\texttt{fromIntegral}"] & \texttt{Double} \arrow[l, shift left, "\texttt{floor, ceiling}"]
\end{tikzcd}$$

To use **`ord`** and **`chr`**, you must load or import the library module **`Data.Char`**.

interpreter

```
floor 3.7
ceiling 3.7
:m + Data.Char
ord 'a'
chr 98
```

Boolean functions

Relational operators

These are functions that return **`Bool`**:

Prelude

```
(<)  :: Int -> Int -> Bool
(<=) :: Int -> Int -> Bool
(>)  :: Int -> Int -> Bool
(>=) :: Int -> Int -> Bool
(==) :: Int -> Int -> Bool
(/=) :: Int -> Int -> Bool
```

Examples:

interpreter

```
2 < 3
2 < 2
2 /= 2
2 /= 3
```

Overloading

The `(==)` function has several types:

Prelude
```
(==) :: Int -> Int -> Bool
(==) :: Bool -> Bool -> Bool
(==) :: Char -> Char -> Bool
(==) :: Float -> Float -> Bool
```

and similarly <=, etc.

interpreter
```
True == False
'a' == 'b'
1.2 < 2.3
:t (==)
:t (<=)
```

The actual types of these functions are

Prelude
```
(==) :: Eq a => a -> a -> Bool
(<=) :: Ord a => a -> a -> Bool
```

Building Boolean expressions

The `Bool` type has the following operators

Prelude
```
(&&) :: Bool -> Bool -> Bool
(||) :: Bool -> Bool -> Bool
not  :: Bool -> Bool
```

Examples:

interpreter
```
not True
False || True
1 < 2 || 1 > 2
```

A function using relational operators:

```
small :: Int -> Bool
small n = 0 <= n && n < 10
```

Characters

Functions on `Char`

These are all in the `Data.Char` library module:

Data.Char
```
isDigit     :: Char -> Bool
isDigit c  =  '0' <= c && c <= '9'

isUpper, isLower :: Char -> Bool
isUpper c  =  'A' <= c && c <= 'Z'
isLower c  =  'a' <= c && c <= 'z'
```

```
isAlpha     :: Char -> Bool
isAlpha c   =  isUpper c || isLower c
```

Try them: **interpreter**

```
:m + Data.Char
isDigit '2'
isDigit 'a'
isLower 'B'
isLower 'b'
isAlpha 'c'
```

More functions on `Char`

These are also in the `Data.Char` library module: `Data.Char`

```
ord :: Char -> Int
chr :: Int -> Char
```

Things to try: **interpreter**

```
:m + Data.Char
ord 'a'
ord '1'
chr 65
chr 48
chr 32
chr (ord 'b')
chr (ord 'b' + 3)
```

Conditionals

Conditional expressions

Haskell has an expression form

`if` *boolexp* `then` exp_1 `else` exp_2

For example, **interpreter**

```
if 1 >= 3 then 1 else 3
if 3 >= 1 then 3 else 1
```

So the function `max`, defined in the `Prelude`, could be written: `Prelude`

```
max :: Int -> Int -> Int
max x y = if x >= y then x else y
```

(Actually `max` has a more general type than this.) **interpreter**

```
max 4 9
max 9 4
4 `max` 9
```

Definition by cases: guarded definitions

In the common case where the whole of the definition is conditional, Haskell has an alternative syntax inspired by mathematical notation, in which we might write

$$max(x, y) = \begin{cases} x, & \text{if } x \geq y \\ y, & \text{otherwise} \end{cases}$$

In Haskell, we can write

Prelude

```
max :: Int -> Int -> Int
max x y
  | x >= y    = x
  | otherwise = y
```

The Boolean expression `x >= y` is called a *guard*.

Guards are tested in order

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
  | x >= y && x >= z  = x
  | y >= x && y >= z  = y
  | otherwise         = z
```

could be simplified to

```
maxThree :: Int -> Int -> Int -> Int
maxThree x y z
  | x >= y && x >= z  = x
  | y >= z            = y
  | otherwise         = z
```

Defining functions using existing functions

Consider

```
maxThree :: Int -> Int -> Int -> Int
```

Instead of defining

```
maxThree x y z
  | x >= y && x >= z  = x
  | y >= x && y >= z  = y
  | otherwise         = z
```

we can reuse **max**, writing

```
maxThree x y z = max x (max y z)
```

or equivalently

```
maxThree x y z = x `max` y `max` z
```

Example: developing a function

Problem:

A function returning the middle of three numbers.

We'll design a solution top-down.
First step: write the type signature.

```
middleNumber :: Int -> Int -> Int -> Int
```

Next: write the left-hand side

```
middleNumber x y z
```

Now ask: How is the desired result related to **x**, **y** and **z**?

Case analysis

In this particular problem, we know that the result will be either **x**, **y** or **z**, so we can write

```
middleNumber x y z
  | ???           = x
  | ???           = y
  | otherwise     = z
```

Now ask: When is **x** the answer? When is **x** the answer?

- If it's trivial, write it down.
- If not, ask: Is there a function that would help, if it existed?

Assuming an auxiliary function

Suppose we had

```
-- between x y z is True if the value of y
-- lies between the values of x and z
between :: Int -> Int -> Int -> Bool
between x y z = undefined
```

The standard constant **undefined** will give a runtime error, so we can't run this, but for now it's a convenient placeholder so we can at least check types. We've also said what the function should do.

Then we could write

```
middleNumber x y z
  | between y x z  = x
  | between x y z  = y
  | otherwise      = z
```

Finishing the design

It remains to give a definition of `between`.

We could define introduce another auxiliary function, but perhaps it is simple enough to define directly:

```
between :: Int -> Int -> Int -> Bool
between x y z =
    (x <= y && y <= z) || (z <= y && y <= x)
```

(The parentheses aren't necessary here, as `&&` binds more tightly than `||`, as in Java, C++, etc.)

Now we can test `between`, and then `middleNumber`.

Local definitions

Local definitions

It's often convenient (and always correct) to introduce a name for an expression that occurs more than once.

The following are equivalent:

```
f x = let y = x*x + 1 in y*y
```

and

```
f x = y*y
  where
    y = x*x + 1
```

but `let`-`in` can occur as a subexpression, while `where` can only be used at the top level.

We usually prefer the `where` form.

You can declare local functions too, but we'll usually avoid that.

`where` and guards

Variables defined in a `where` clause can also be used in guards, as in this example:

```
-- number of real roots of a quadratic equation
-- a*x^2 + b*x + c = 0
numberOfRoots :: Double -> Double -> Double -> Int
numberOfRoots a b c
  | d < 0 = 0
  | d == 0 = 1
  | otherwise = 2
  where
    d = b*b - 4*a*c
```

Enumerated types

Enumerated types

We can define our own types using `data` declarations, to more precisely describe the values we are operating on.

We'll start with enumerated types. In later sessions we'll see more complex definitions.
An enumerated type representing a limited set of colours:

```
data Colour
    = Red | Green | Blue | Yellow
    | Cyan | Magenta | Black | White
```

This defines:

- a new type `Colour`
- eight new values of that type.

The `Show` class

By default, the interpreter won't display the values of a new type we define, which might be what we want if we want to keep the type abstract.

But often we do want to see values, so we need to explicitly add the new type to the standard `Show` class, which the interpreter uses to render values:

```
data Colour
    = Red | Green | Blue | Yellow
    | Cyan | Magenta | Black | White
    deriving (Show)
```

The other important example of types the interpreter can't show are functions.

Defining functions on enumerated types

We can use `case` to inspect values of an enumerated type, with a case for each alternative. Thus an inversion function has 8 cases:

```
invert :: Colour -> Colour
invert c = case c of
    Red -> Cyan
    Green -> Magenta
    Blue -> Yellow
    Yellow -> Blue
    Cyan -> Red
    Magenta -> Green
    Black -> White
    White -> Black
```

Defining a function with multiple equations

Alternatively, we can define the function with multiple equations, one for each possible value:

```
invert :: Colour -> Colour
invert Red = Cyan
invert Green = Magenta
invert Blue = Yellow
invert Yellow = Blue
invert Cyan = Red
invert Magenta = Green
invert Black = White
invert White = Black
```

All the equations for a particular function must occur together; they can't be mixed in with other functions.

We generally prefer the multi-clause form.

A function to invert colours

Sometimes we can avoid listing all the alternatives by using a variable for the default case:

```
-- Is this a primary colour?
primary :: Colour -> Bool
primary Red = True
primary Green = True
primary Blue = True
primary c = False
```

This relies on the fact that the first matching case is used.

When the variable isn't used on the righthand side, we can use underscore instead:

```
primary _ = False
```

Booleans

The built-in `Prelude` defines

Prelude

```
data Bool = False | True
    deriving (Show, Eq, Ord)

not :: Bool -> Bool
not False = True
not True = False

(||) :: Bool -> Bool -> Bool
False || b = b
True  || b = True

(&&) :: Bool -> Bool -> Bool
False && b = False
True  && b = b
```

Another example from the `Prelude`

The result of a comparison:

Prelude

```
data Ordering = LT | EQ | GT
    deriving (Show)
```

Things to try:

interpreter

```
compare 1.2 3
compare 1.2 1
compare 1 1
:t compare
```

Summary

Function definitions

Declaring the type of an n-argument function:

f `::` $type_1$ `->` $\cdots$ $type_n$ `->` $type$

Simple equation:

f arg_1 $\cdots$ arg_n `=` exp
[**where**
    $decls$]

Guarded definition:

f arg_1 $\cdots$ arg_n
  `|` $boolexp_1$ `=` exp_1
    ⋮
  `|` $boolexp_k$ `=` exp_k
[**where**
    $decls$]

Each arg could be

- a variable, and acts as its declaration, or
- a constant in an enumerated type, like `Blue`.

Later we will generalize arguments to *patterns*.
The **where** part is optional.

There is more than one way to do it

Haskell provides different ways of expressing conditionals, local definitions and case analysis:

Expression form	**Clause form**
if $boolexp$ **then** exp **else** exp	guards
let $decls$ **in** exp	**where** $decls$
case exp **of** arg_1 `->` exp_1 ⋮ arg_n `->` exp_n	multiple clauses, with pattern matching

Because we're aiming for small functions, the clause forms will be more useful.

Next session: structured data types

- Composite types:
  - tuples (pairs, triples, etc)
  - user-defined record types
- Polymorphism: functions that do the same thing for all types, e.g. **`fst`** does the same thing for pairs of anything.
- Reading: Thompson, sections 5.2–3

Exercises

Exercises 1–7 deal with writing functions that operate on the primitive types of Haskell, such as **`Int`**, **`Double`** and **`Char`**. The last two exercises deal with defining enumerated types and writing functions that manipulate them.

1. Write a function **`threeDifferent`** that takes three integer arguments and returns **`True`** if its arguments are all different from each other.

2. Work out what the following function does:

```
mystery :: Int -> Int -> Int -> Bool
mystery x y z = not (x == y && y == z)
```

   (If stuck, try the function on example inputs in the interpreter.)

3. Rewrite **`mystery`** so that it implements the same function, but without using **`not`**.

4. Evaluate the expression

   interpreter

```
(0.1 + 1.1) - 1.2
```

   Is the answer what you expected? Try some others.

   interpreter

```
1.2 == 1.2
0.1 + 1.1 == 1.2
```

   What is going on here?

   Hint: this *not* a peculiarity of Haskell; you can get the same effect in C, Java, C++, etc.

5. Write a function that computes the fractional part of a **`Double`**, e.g. mapping **`23.75`** to **`0.75`**. *Hint:* use the **`floor`** function (and another one – see the diagram of the slide *Conversions between types*). Note that, because it uses **`Double`**, this function will be susceptible to the same issues as seen in the last question.

6. Write a function **`clamp`** that takes three arguments, **`lo`**, **`hi`** and **`x`** (all of type **`Double`**) and returns **`x`** if it is between **`lo`** and **`hi`** (inclusive), **`lo`** if **`x`** is less than **`lo`**, and **`hi`** if **`x`** is greater than **`hi`**. You may assume that **`lo`** $\leq$ **`hi`**.

   Consider two versions: one using guards, and another using **`max`** and **`min`**.

7. Write a function

   ```
   charToNum :: Char -> Int
   ```

   that converts a character that represents a digit, like `'3'`, to the corresponding integer (`3`), or 0 if the character does not represent a digit. You may assume that the digit characters are contiguous and in ascending order. You will need to place "`import Data.Char`" in your module to gain access to the character functions.

8. Put the following in a module **`Geometry`** (in a source file `Geometry.hs`):

   (a) Define an enumerated type **`Direction`** with four values, the cardinal compass points.

   (b) Define a function

   ```
   turnLeft :: Direction -> Direction
   ```

   that maps each direction to the one immediately to its left. (Try to do this by matching the argument, e.g. like **`invert`** in the lecture, instead of using **`==`**.)

   (c) Define a function **`turnRight`** that does the reverse.

   We'll be adding more to this module in later weeks.

9. Put the following in a module **`Calendar`** (in a source file `Calendar.hs`):

   (a) In the Gregorian calendar, a year is a leap year if it is divisible by 4, except that that centuries are leap years only if divisible by 400. Thus 2000 was a leap year, but 2100 won't be. Write a function

   ```
   isLeapYear :: Int -> Bool
   ```

   to test whether a year is a leap year. To test whether a number is divisible by another, use **`mod`** and compare the remainder with 0, e.g. `` y `mod` 4 == 0 `` is **`True`** if **`y`** is divisible by 4.

   (b) Write a function

   ```
   daysInYear :: Int -> Int
   ```

   that returns the number of days in a year given as argument.

   (c) Define an enumerated type **`Month`** for representing the months of the year.

   (d) Define a function

   ```
   daysInMonth :: Month -> Int -> Int
   ```

   that returns the number of days in a month in a given year. (For most months, the answer is the same whatever the year, but for one month it does vary. Again, try to avoid using **`==`**.)