Session 1

Introduction

This module: a different kind of programming

"Language shapes the way we think, and determines what we can think about."
— Benjamin Lee Whorf

Procedural programming (e.g. C, Java, etc)

- closely mirrors the underlying architecture.
- focusses on *how* something is done (the steps).

Declarative programming including functional programming,

- focusses on *what* is the answer (the values).
- provides powerful abstraction (generalization) mechanisms.
- is concise, expressive, and fun.

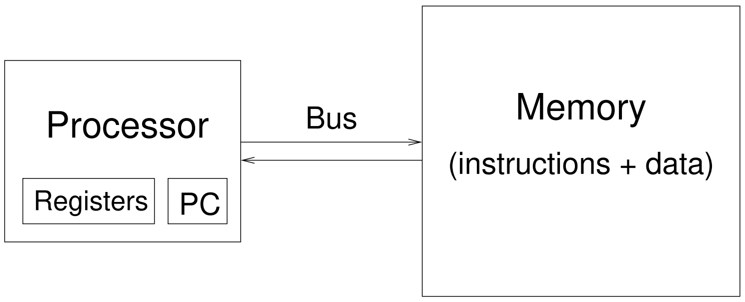
This session

- What is functional programming?
  - programming with functions
  - using and defining functions
  - types
  - a look ahead
- Getting started with the GHCi system.
  - evaluating expressions
  - errors
  - loading definitions

Background

The von Neumann architecture

As described by John von Neumann in 1945:



The von Neumann bottleneck

John Backus (the inventor of FORTRAN), used this term in his 1977 ACM Turing award lecture:

> "Surely there must be a less primitive way of making big changes in the store than by pushing vast numbers of words back and forth through the von Neumann bottleneck. Not only is this tube a literal bottleneck for the data traffic of a problem, but, more importantly, it is an intellectual bottleneck that has kept us tied to word-at-a-time thinking instead of encouraging us to think in terms of the larger conceptual units of the task at hand. Thus programming is basically planning and detailing the enormous traffic of words through the von Neumann bottleneck, and much of that traffic concerns not significant data itself, but where to find it."

Procedural programming

Problem: factorial of $n = 1 * \ldots * n$

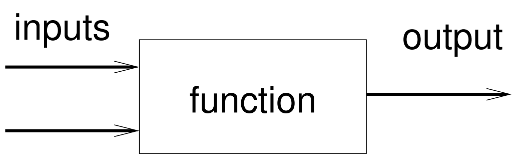
Solution: in Java:

```
public int factorial(int n) {
    int fact = 1;
    for (int i = 1; i <= n; i++)
        fact = fact * i;
    return fact;
}
```

The procedural solution reflects the "von Neumann bottleneck".

What is functional programming?

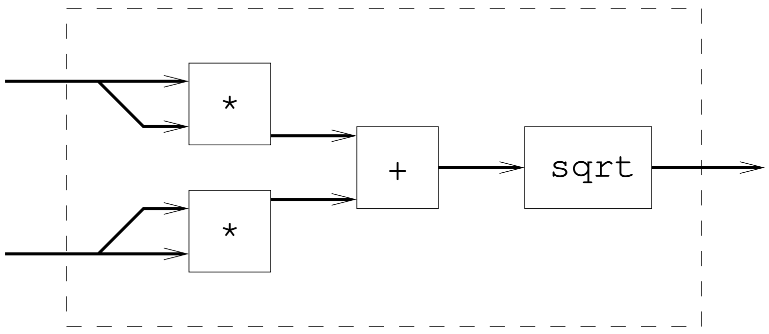
Basic component is a function:



Recall that a function determines a unique output for each combination of inputs:

inputs		*output*
`True`	**`True`**	**`True`**
`True`	**`False`**	**`True`**
`False`	**`True`**	**`True`**
`False`	**`False`**	**`False`**

Building new functions from old



Prehistory of functional programming languages

Functional programming languages have pioneered new forms of abstraction, which eventually became mainstream.

The story begins with logic:

1924 Combinatory logic: functions as data

$$S(S(KS)(S(KK)(S(KS)(S(S(KS)(S(KK)I))(KI)))))\\ (K(S(KK)(S(KK)I)))$$

1933 The λ-calculus: parameters

$$\lambda l.\ \lambda f.\ \lambda v.\ l\ v\ (\lambda x.\ \lambda u.\ f\ u)$$

1958 LISP: an implemented language, with automatic garbage collection

```
(define (length l)
  (if (null? l) 0
      (+ 1 (length (tail l)))))
```

Modern functional programming languages

1980 ML: polymorphism, algebraic data types

```
fun length [] = 0
  | length (x::xs) = 1 + length xs;
```

1985 Miranda: a pure functional language

```
length :: [*] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

1990 Haskell: constrained polymorphism

```
length :: [a] -> Int
length [] = 0
length (x:xs) = 1 + length xs
```

- A standard language, Haskell 2010
- A vehicle for language experimentation

Pure functional languages targeting JavaScript: Elm, PureScript, etc.

Haskell and the GHCi interpreter

Using the GHCi interpreter

Type expressions into the GHCi window. GHCi will respond with their values.

interpreter

```
2+3
2+3*5
(2+3)*5
2^3
2^2^3
```

Does whitespace matter? Do expressions behave the way you expect?

A function f applied to an expression e is written $f\ e$:

interpreter

```
sqrt 2
sqrt 2 + 3
sqrt (sqrt 2)
```

What are the rules with parentheses?

More things to try

Functions with two arguments:

interpreter

```
max 4 8
div 13 5
mod 13 5
```

Infix functions:

interpreter

```
13 `div` 5
13 `mod` 5
```

Make sure you use the right quotes here.

Another kind of data: lists

Lists of integers:

interpreter

```
[1, 1+3, 1+3+5]
```

The value of **[**a**..**b**]** is a list of integers, from a up to b:

interpreter

```
[1..6]
[2..2]
[6..1]
```

If e is a list of integers, **product** e is their product.

interpreter

```
product [3,4,5]
product [1..7]
product []
```

If e is a list of integers, **sum** e is their sum.

More lists

Some list functions:

interpreter

```
length [1,3,5,7]
replicate 5 2
reverse [1,3,5]
[1,2,3] ++ [4,5,6,7]
```

In Haskell, strings are just lists of characters:

interpreter

```
['H', 'e', 'l', 'l', 'o']
['a'..'z']
length "Hello"
replicate 5 '.'
reverse "Hello"
"cup" ++ "board"
```

Asking GHCi for the types of expressions

We can ask for the type of an expression with the `:type` command:

interpreter

```
:type 'a'
:t ['a', 'b', 'c']
:t "abc"
```

Some functions work for many types:

interpreter

```
:t replicate
:t reverse
:t (++)
```

Here “**a**” is a *type variable*: it can stand for any type. These functions are polymorphic (details in session 3).

Constrained polymorphism

Numeric functions also work for multiple types:

interpreter
```
1+3
1.2+3.45
```

but not just anything, as we can see from its type:

interpreter
```
:t (+)
:t mod
:t sqrt
```

Here `Num a =>` is a *constraint* on the type. The type `a` must belong to the set of types (or *class*) `Num` of numeric types, which includes including `Int`, `Integer` and `Double`.

Even numeric literals can have multiple types:

interpreter
```
:t 3
```

More numeric classes

Some numeric expressions are more tightly constrained:

interpreter
```
:t 13 `mod` 5
:t mod
```

The `Integral` types include `Int` and `Integer`.

interpreter
```
:t sqrt
:t sqrt 5
```

The `Floating` types include `Double`.

Other type classes:

Eq types that support equality testing

Show types whose values can be shown as strings

We shall return to constrained types in later sessions.

Errors

Syntax errors

interpreter
```
'x
2+3)*4
```

Type errors

interpreter
```
:t 'x'
:t (++)
'x' ++ 'y'
```

Another kind of type error:

interpreter
```
1 + 'x'
```

execution errors (rare)

interpreter
```
1 `div` 0
```

Themes

Value-oriented programming

An expression is just its value:

- Expressions that have the same value are interchangeable. (simplifies refactoring)
- Think about the value you want to produce, and how it is related to the value you are given.
- Consider what intermediate values might be useful.
- Thing big:
  - Where possible, operate on whole data structures rather than individual elements.
  - Sometimes a more general solution would be simpler.

What's missing

- modifiable variables
- loops (but we will get to recursion later)
- side effects, such as printing things from the middle of a program
- breakpoints

Static types

Every expression in Haskell has a type:

- `3` has type `Integer` – we write `3 :: Integer`, pronouncing `::` as "has type".
- `2+3 :: Integer`
- `[1,2,3] :: [Integer]` (list of `Integer`)
- `product :: [Integer] -> Integer`
- `product [1,2,3] :: Integer`

(Later we will see that these have more general types.)

Other basic types: `Int` (fixed size integers), `Double`, `Char`, `Bool`.

Basic rule of function types

$$\left.\begin{array}{l} f :: T_1 \texttt{->} T_2 \\ e :: T_1 \end{array}\right\} \Rightarrow f\ e :: T_2$$

Typeful programming

Write down the types first:

- clarify your design
- basic documentation
- more helpful error messages

Type errors are your friends:

- much better than runtime errors
- facilitate aggressive refactoring

Define your own types:

- more precisely express your design
- more type errors (see above)

Abstraction

- "Don't Repeat Yourself" (DRY) is a useful principle whatever language you're programming in.
- We achieve this through *abstraction*: factoring out the common parts of repetitious code, *e.g.* Java methods, classes and generics.
- Functional languages like Haskell are particularly good at abstraction: functions, higher-order functions, parametric polymorphism, type classes

Lazy evaluation

In Haskell (unlike Elm), expressions are not evaluated until their results are needed.
Some consequences:

- We can compose functions together without feeling guilty about large intermediate values, because they are constructed on demand.
- We can have intermediate values that are infinite (provided we only look at part of them).
- Things that must be built-in to other languages are just functions.

interpreter

```
product (take 7 [1..])
```

Drawback: working out time and (especially) space usage is harder. (We won't worry about that here.)

Small functions operating on big values

Decompose the program into lots of small, non-recursive, top-level functions, each performing a simple, meaningful transformation.

- can be independently tested
- easy to combine in different ways
- focusses attention on whole values rather than piecemeal changes
- recursive functions are harder to reason about, and you can't examine intermediate values to see what's going wrong.
- more powerful approach: pass simple functions as arguments to higher-order library functions (session 5)

Structure of this module

We will consider increasingly elaborate kinds of values, with the kinds of code used to operate on them:

	Values	**Code**
1.		simple functions
2.	scalar values	conditionals, cases
3.	composite values	pattern matching
4.	lists	list comprehensions
5.	higher-order functions	
6.	containers	
7.	lists revisited	recursion
8.	recursive datatypes	
9.	parsers	
10.	actions	**do**-expressions

Source files

GHCi commands

An addition to expressions (which must be on a single line), GHCi accepts a number of commands, each starting with a colon, including:

`:load` *filename* ...	load modules from specified files
`:reload`	repeat last load command
`:type` *exp*	print type of expression
`:?`	display a full list of commands
`:quit`	exit the interpreter

Each command may be abbreviated to a single letter.

Function syntax

We can also define functions:

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

Note: definitions go in program files; you can't type them at the GHCi command line.

Using a function in an expression:

$$f\ exp_1\ \cdots\ exp_n$$

Type signature:

$$f\ \texttt{::}\ type_1\ \texttt{->}\ \cdots\ type_n\ \texttt{->}\ type$$

Definition:

$$f\ x_1\ \cdots\ x_n\ \texttt{=}\ exp$$

`Week1.hs`: a simple Haskell module

```haskell
module Week1 where

size :: Integer
size = 12+13

-- The square of an integer.
square :: Integer -> Integer
square n = n*n

-- Triple an integer.
triple :: Integer -> Integer
triple n = 3*n
```

Language details: layout is significant

Haskell uses indentation to decide where another definition starts, so all definitions must have the same indentation.

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

Legal: two definitions

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

Illegal: one definition?

```haskell
square
:: Integer -> Integer
```

Illegal: two definitions?

Language details: identifiers

Haskell is case-sensitive.
Moreover, identifiers are divided into two sorts:

- identifiers starting with a capital letter:
  - module names, like `Week1`.
  - type names, like `Integer`.
- identifiers starting with a lower case letter:
  - variables, like `n` or function names, like `square`.
  - type variables, like `a` (see later).

Things to try

Load `Week1.hs` into GHCi and type:

interpreter

```
square size
triple (square 2)
square (triple 2)
23 - triple (3+1)
23 - triple 3+1
```

Use the GHCi command "`:type`" ("`:t`" for short) to tell you the types of each of these, and also `triple`.

interpreter

```
:type triple
:t square size
```

What is wrong with the following?

interpreter

```
triple square 2
```

Calling functions we've defined

Edit the file `Week1.hs` to add a definition of a function that triples its input and returns the square of that:

```
squareOfTriple :: Integer -> Integer
squareOfTriple n = square (triple n)
```

Reload your script and try out the function:

interpreter

```
:r
squareOfTriple 1
squareOfTriple 3
```

Looking ahead

Functional languages provide powerful abstraction mechanisms:

- parametric polymorphism (session 3, 4, ...)

Prelude
```
length :: [a] -> Int
reverse :: [a] -> [a]
```

- constrained polymorphism

Prelude
```
elem :: Eq a => a -> [a] -> Bool
```

- functions as data (higher-order functions: session 5)

Prelude
```
map :: (a -> b) -> [a] -> [b]
```

Next session: primitive types and enumerated types in Haskell.

Exercises

Remember that definitions can't be typed into the interpreter; they must be placed in a text file (a Haskell module). You'll find it easiest to have the editor going in one window and GHCi in another. Whenever you save a change in the editor, type "`:r`" in the GHCi window.

Edit the file `Week1.hs` to add definitions of functions to do the following:

1. Define a constant for the number of seconds in a week.

2. Define a floating point constant for the "golden ratio", defined as $\varphi = \frac{1+\sqrt{5}}{2}$. That is, define

```
phi :: Double
phi = ...
```

   Evaluate φ^2, $\varphi^2 - 1$, $\frac{1}{\varphi}$ and $\frac{1}{\varphi-1}$

3. A mile is exactly 1.609344 kilometres. Define a constant of type **`Double`** holding this value, and use it to define functions

```
milesToKm :: Double -> Double
kmToMiles :: Double -> Double
```

   converting from miles to kilometres and vice versa.

4. (using the existing functions **`square`** and **`triple`**) Define a function that squares its input and returns the triple of that. Your solution should include a declaration of the type of the function.

5. Define a function that computes the the square of the square of its input (the fourth power).

6. Define a function with signature

```
factorial :: Integer -> Integer
```

   that computes the factorial of its input.

7. Try applying the function **`factorial`** from the last question to 21, and bigger numbers. Then try changing its type to operate on values of type **`Int`** instead of **`Integer`**, and try again. Try different numbers. What is going on?

8. Define a function with signature

   ```
   norm :: Double -> Double -> Double
   ```

   that computes the function $norm\ x\ y = \sqrt{x^2 + y^2}$. (Don't use the **`square`** function in this part, because it operates on **`Integer`**.)