What is a functional programming language?

- Basic notion of computation: the application of functions to arguments.
- Basic idea of program: writing function definitions
- Functional languages are declarative:
  more emphasis on *what* rather than *how*.

Defining Haskell values

- Function definitions are written as equations
- `double x = x + x`
  `quadruple x = double (double x)`
- compute the length of a list

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

  recursion is the natural way to describe repeated computation
- Haskell can infer types itself (Type Inference)

Type Polymorphism

- What is the type of `length`?

  ```
  > length [1,2,3]
  3
  > length ['a','b','c','d']
  4
  > length [[],[1,2],[3,2,1],[],[6,7,8]]
  5
  ```

- `length` works for lists of elements of arbitrary type
  `length :: [a] -> Int`
  Here '`a`' denotes a type variable, so the above reads as
  "`length` takes a list of (arbitrary) type `a` and returns an `Int`".
- A similar notion to "generics" in O-O languages, but builtin without fuss.

Laziness

- What's wrong with the following (recursive) definition ?

  ```
  from n =  n : (from (n+1))
  ```

  Nothing ! It just generates an infinite list of ascending numbers, starting from `n`.
- `take n list` — return first `n` elements of `list`.
- What is `take 10 (from 1)` ?

  ```
  > take 10 (from 1)
  [1,2,3,4,5,6,7,8,9,10]
  ```

- Haskell is a *lazy* language, so values are evaluated only when needed.

Program Compactness

- Sorting the empty list gives the empty list:

```
qsort [] = []
qsort (x:xs)
  = qsort [y | y <- xs, y < x]
    ++ [x]
    ++ qsort [z | z <- xs, z >= x]
```

- We have used Haskell list comprehensions
  `[y | y <- xs, y < x ]`
  "build list of `y`s, where `y` is drawn from `xs`, such that `y < x`"
- Try that in Java !

Whistle ... Stop!

- Haskell is powerful, and quite different to most mainstream languages
- It allows very powerful programs to be written in a concise manner
- These languages originally developed for theorem provers and rewrite systems
- Very popular now for:
  - software static checkers, e.g., Facebook's `infer` (fbinfer.com)
  - quantitative analysis in financial services
  - Domain-Specific Languages (DSLs)
  - Front-end language handling and transformation.

The λ-Calculus

- Invented by Alonzo Church in 1930s
- Intended as a form of logic
- Turned into a model of computation
- Not shown completely sound until early 70s !

λ-Calculus: Syntax

Infinite set *Vars*, of variables:

$$u, v, x, y, z, \ldots, x_1, x_2, \ldots \quad \in \quad Vars$$

Well-formed λ-calculus expressions *LExpr* is the smallest set of strings matching the following syntax:

$$\begin{array}{rcl} M, N, \ldots \in LExpr & ::= & v \\ & | & (\lambda x \bullet M) \\ & | & (M\ N) \end{array}$$

Read: a λ-calculus expression is either (i) a variable (v); (ii) an abstraction of a variable from an expression ($\lambda x \bullet M$); or (iii) an application of one expression to another (($M\ N$)).

λ-Calculus: Free/Bound Variables

- A variable *occurrence* is *free* in an expression if it is not mentioned in an *enclosing abstraction*.

$$x \qquad (\lambda y \bullet (yz)$$

- A variable *occurrence* is *bound* in an expression if is mentioned in an *enclosing abstraction*.

$$x \qquad (\lambda y \bullet (yz)$$

- A variable can be both free and bound in the same expression

$$(x(\lambda x \bullet (xy))$$

Think of bound variables as being like local variables in a program.

λ-Calculus: α-Renaming

We can change a binding variable and its bound instances provided we are careful not to make other free variables become bound.

$$(\lambda x \bullet (\lambda y \bullet (x\ y))) \overset{\alpha}{\to} (\lambda u \bullet \lambda v \bullet (u\ v)))$$
$$(\lambda x \bullet (x\ y)) \overset{\alpha}{\not\to} (\lambda y \bullet (y\ y))$$

formerly free y has been "captured" !

This process is called α-Renaming or α-Substitution, and leaves the meaning of a term unchanged.

It's the same as changing the name of a local variable in a program (fine, but you need to take care if there is a global variable of the same name hanging around)

λ-Calculus: Substitution

We define the notion of substituting an expression N for all free occurrences of x, in another expression M, written:

$$M[N/x]$$

$$(x\ (\lambda y \bullet (z\ y)))\ [\ (\lambda u \bullet u)\ /\ z\ ] \overset{\beta}{\to} (x\ (\lambda y \bullet ((\lambda u \bullet u)\ y)))$$

$$(x\ (\lambda y \bullet (z\ y)))\ [\ (\lambda u \bullet u)\ /\ y\ ] \overset{\beta}{\to} (x\ (\lambda y \bullet (z\ y)))$$

y was not free anywhere

λ-Calculus: Careful Substitution!

When doing (general) substitution $M[N/x]$, we need to avoid variable "capture" of free variables in N, by bindings in M:

$$(x\ (\lambda y \bullet (z\ y)))[(y\ x)/z] \overset{\beta}{\not\to} (x\ (\lambda y \bullet ((y\ x)\ y)))$$

If N has free variables which are going to be inside an abstraction on those variables in M, then we need to α-Rename the abstractions to something else first, and then substitute:

$$\begin{aligned} & (x\ (\lambda y \bullet (z\ y)))[(y\ x)/z] \\ \overset{\alpha}{\to}\ & (x\ (\lambda w \bullet (z\ w)))[(y\ x)/z] \\ \overset{\beta}{\to}\ & (x\ (\lambda w \bullet ((y\ x)\ w))) \end{aligned}$$

The Golden Rule: A substitution should never make a free occurrence of a variable become bound, or vice-versa.

λ-Calculus: β-Reduction

We can now define the most important "move" in the λ-calculus, known as β-Reduction:

$$(\lambda x \bullet M)\ N \ \stackrel{\beta}{\to}\ M[N/x]$$

We define an expression of the form $(\lambda x \bullet M)\ N$ as a "($\beta-$)redex" (*red*ucible *ex*pression).

λ-Calculus: Normal Form

An expression is in "Normal-Form" if it contains no redexes. The object of the exercise is to reduce an expression to its normal-form (*if it exists*).

$$\begin{array}{ll} & (\underline{((\lambda x \bullet (\lambda y \bullet (y\ x)))\ u)}\ v) \\ \stackrel{\beta}{\to} & \underline{((\lambda y \bullet (y\ u))\ v)} \\ \stackrel{\beta}{\to} & (v\ u) \end{array}$$

Not all expressions have a normal form — e.g.:
$((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))$
What about:

$$(\ (\ (\lambda x \bullet (\lambda y \bullet y))\ \ ((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))\ )\ \ w\ )\quad ?$$

$$(\ (\ (\lambda x \bullet (\lambda y \bullet y))\ \ ((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))\ )\ \ w\ )$$

- Do innermost redex first

$$\begin{array}{ll} & (\ (\ (\lambda x \bullet (\lambda y \bullet y))\ \ \underline{((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x))})\ )\ \ w\ ) \\ \stackrel{\beta}{\to} & (\ (\ (\lambda x \bullet (\lambda y \bullet y))\ \ ((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))\ )\ \ w\ ) \end{array}$$

  We can keep doing this forever!
- Do outermost (leftmost) redex first

$$\begin{array}{ll} & (\ \underline{(\ (\lambda x \bullet (\lambda y \bullet y))\ \ ((\lambda x \bullet (x\ x))\ (\lambda x \bullet (x\ x)))\ )}\ \ w\ ) \\ \stackrel{\beta}{\to} & (\ (\lambda y \bullet y)\ \ w\ ) \\ \stackrel{\beta}{\to} & w \end{array}$$

λ-Calculus and Computability

- So What ? Why do we look at this weird calculus anyway?
- We can use it to encode booleans, numbers, and functions over same.
- In fact, we can encode any computable function this way!
- λ-Calculus is Turing-complete
  - or is it that Turing machines are Church-complete?
- It is one of a number of equivalent models of computation that emerged in the 1930s

And this has what to do with functions, exactly?

Consider a "conventional" function definition and application of that function to an argment:

$$f(x) = 2x + 1 \qquad f(42)$$

$$\begin{array}{ll} & f(42) \\ = & \text{substitute 42 for } x \text{ in definition r.h.s.} \\ & (2x+1)[42/x] \\ = & \text{perform substituion} \\ & 2 \times 42 + 1 \end{array}$$

This is basically β-reduction!
What the λ-calculus captures is function definition and application ($f = \lambda x \bullet 2x + 1$)

Lambda abstraction in Haskell

The Haskell notation is designed to reflect how it looks in lambda-calculus
Since these values are themselves functions, we just apply them to values to compute something

```
> (\x -> 2 * x + 1) 42
85
```

Essentially we can view Haskell as being the (typed) lambda-calculus with *LOTS* of syntactic sugar.

Haskell for CS3016 (2018)

- We shall use the GHC compiler
  https://www.haskell.org/downloads
  Version 7.10.3
- Coursework will be based on the use of the stack tool
  https://www.stackage.org (using lts-6.19).
  https://docs.haskellstack.org/en/stable/README/
- Install stack and let it install ghc, at least as far as this course is concerned (see Lab00, to come).

Course Timetable (2018–19)

- Timetable:
  - Mon 2pm LB 0.1/ICTLabI&II : Lecture/Labs
  - Thu 2pm LB 0.4/ICTLab I : Lecture/Labs
    This week will be a Lecture in LB 0.4
  - Fri 3pm LB 0.1 : Lecture/Tutorial
- Class Management: Blackboard
- Assessment
  - Exam : 75%, of which 70% is standard written paper (3Qs, do 2) and 30% is multiple choice.
  - Continuous Assessment : 25%
- Notice: there will be no class on Fri Oct 12th 2018.