Can Programming Be Liberated from the von Neumann Style?

• This is the title of a lecture given by John Backus when he received the Turing Award in 1977

• He pointed out that programs should be abstract descriptions of algorithms rather than sequences of changes in the state of the memory
  – He called for raising the level of abstraction
  – A way to realize this goal is functional programming

• Programs written in modern functional programming languages are a set of mathematical relationships between objects
  – No explicit memory management takes place
We will study functional programming in the context of the language Haskell
- [http://www.haskell.org](http://www.haskell.org)
- Download and install GHC (you can find it in the labs)

- Haskell is a purely functional language created in 87
- Haskell is the state of the art in functional programming
- I recommend the following introductory book

A very simple functional program (also known as a functional script) in Haskell
- A set of definitions

```
square :: Integer -> Integer
square x = x * x
smaller :: (Integer, Integer) -> Integer
smaller (x,y) = if x <= y then x else y
main = print (square(smaller(5, 3+4)))
```
Evaluation Order

- Functional programs are evaluated following a *reduction* (or evaluation or simplification) process
- There are two common ways of reducing expressions
  - Applicative order
    » Impatient evaluation
  - Normal order
    » Lazy evaluation

Applicative Order

- In applicative order, expressions are evaluated following the parsing tree (deeper expressions are evaluated first)

\[
\text{square } (3 + 4) \\
= \{ \text{definition of } + \} \\
\quad \text{square } 7 \\
= \{ \text{definition of square } \} \\
\quad 7 \ast 7 \\
= \{ \text{definition of } \ast \} \\
\boxed{49} \text{ Normal Form}
\]
Normal Order

• In normal order, expressions are evaluated only their value is needed

\[
\begin{align*}
square (3 + 4) &= \{ \text{definition of square } \} \\
(3 + 4) \times (3 + 4) &= \{ \text{definition of + applied to the first term } \} \\
7 \times (3 + 4) &= \{ \text{definition of + applied to the second term } \} \\
7 \times 7 &= \{ \text{definition of * } \} \\
49 &= \{ \text{definition of } \}
\end{align*}
\]

Evaluation Order and Infinity

• Normal is sometimes more efficient than applicative order

• Applicative order can handle expressions that never converge to normal forms

\[
\begin{align*}
three :: Integer -> Integer \\
three x &= 3 \\
infinity :: Integer \\
infinity &= infinity + 1
\end{align*}
\]
Evaluation Order
Example

• What is the value of three infinity?
• In applicative order

three infinity
= { definition of infinity }
  three (infinity + 1)
= { definition of infinity }
  three ((infinity + 1) + 1)
= { definition of infinity }
... This expression never reaches a normal form

Evaluation Order
Example

• What is the value of three infinity?
• In normal order

three infinity
= { definition of three }
  3
Haskell Evaluation Order

- Haskell is a *lazy* functional programming language
  - Expressions are evaluated in applicative order
  - Identical expressions are evaluated only once

```
square (3 + 4)
= { definition of square } (3 + 4) * (3 + 4)
= { definition of + applied to both terms } 7 * 7
= { definition of * } 49
```

Values

- Expressions *denote* values
  - They are *not* values, but representations of them
  - E.g. `three infinity` denotes 3
    
    ```
    three infinity = 3
    ```

- In functional programming, we denote an undefined value using the symbol ⊥
  - ⊥ is pronounced *bottom*
  - E.g. `square infinity` denotes ⊥

- A function \( f \) that satisfies \( f ⊥ = ⊥ \) is said to be *strict*
- Otherwise, \( f \) is *nonstrict*
Functions

- Functions are the most important kind of value in functional programming
  - Functions are values!
- Mathematically, a function $f$ associates an element of a set $X$ to a unique element of second set $Y$
  - We write $f: X \rightarrow Y$

\begin{align*}
\text{three} & : \text{Integer} \rightarrow \text{Integer} \\
\text{infinity} & : \text{Integer} \\
\text{square} & : \text{Integer} \rightarrow \text{Integer} \\
\text{smaller} & : (\text{Integer}, \text{Integer}) \rightarrow \text{Integer}
\end{align*}

Functions

- A function $f: X \rightarrow Y$ is said to take arguments in $X$ and return a result in $Y$
- Do not confuse a function $f$ and an application of a function $f(x)$
  - We write $f \ x$ in Haskell
- Two functions are equal if they give equal results for equal arguments
  - This is the principle of extensionality
Currying Functions

- Functional can be applied to a partially resulting in new functions with fewer arguments

\[
\text{smaller} :: (\text{Integer, Integer}) \rightarrow \text{Integer} \\
\text{smaller} (x, y) = \text{if } x \leq y \text{ then } x \text{ else } y
\]

\[
\text{smaller2} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\
\text{smaller2} x y = \text{if } x \leq y \text{ then } x \text{ else } y
\]

- The value of the application \( \text{smaller2} \ x \) is a function with type \( \text{Integer} \rightarrow \text{Integer} \)
  - This is known as currying a function

Reading Assignment

- A Short Introduction to Haskell
  - http://www.haskell.org/aboutHaskell.html