Functional Programming

COMP 524: Programming Language Concepts
Björn B. Brandenburg

The University of North Carolina at Chapel Hill

Based in part on slides and notes by S. Olivier, A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts.

Brief Overview

- First, some introductory remarks.
- Then we'll cover Haskell details in a tutorial style.
  - **Take notes**.
  - **Ask questions**.

What's a functional language?

Most functional languages provide:

- Functions as first-class values
- Higher-order functions
- Primitive list type (operators on lists)
- Recursion
- Structured function return (return tuples)
- Garbage collection
- Polymorphism and type inference
  - Covered next lecture.

Functional programming.

- Also possible in imperative languages.
- Applying functional style to imperative language can yield very elegant code.

So Why Functional?

- Teaches truly recursive thinking.
- Teaches **good programming style**.
  - Short, self-contained functions.
- Implicit Polymorphism.
- **Natural expressiveness** for symbolic and algebraic computations.
- Algorithms clearly map to the code that implements them.

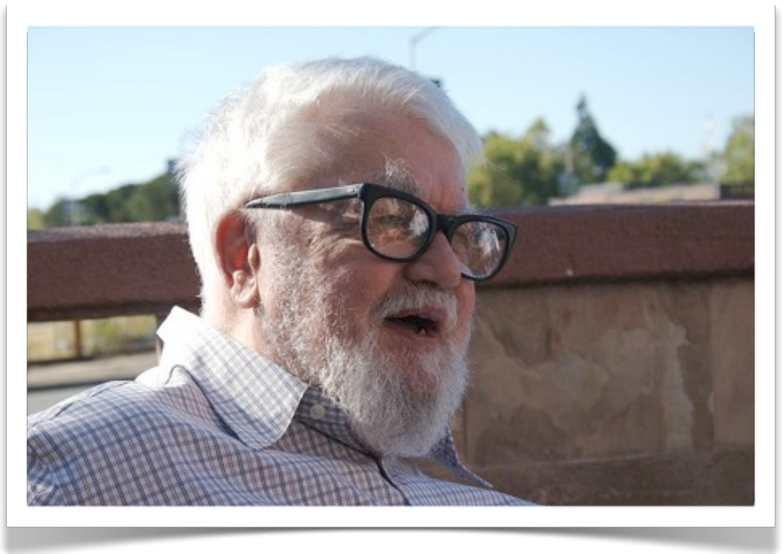
Origins

- Lambda-calculus as semantic model (Church)
- LISP (1958, MIT, McCarthy)

Alanzo Church

```
(defun fib (n)
  (if (or (= n 0) (= n 1))
      1
      (+ (fib (- n 1))
         (fib (- n 2)))))
```

Naive Fibonacci Implementation in LISP



John McCarthy

Source: Wikimedia Commons

Can Programming be Liberated from the von Neumann Style?

- This is the title of the lecture given by **John Backus** when he received the **Turing Award** in 1977.
- In this, he pointed out that a program should be an **abstract description of algorithms** rather than a sequence 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.

History

Lisp
➡ Dynamic Scoping

Common Lisp (CL), Scheme
➡ Static scoping

Common Lisp Logo (unofficial)


Scheme Logo

ML (late 1970ies)
➡ Typing, type inference, fewer parentheses


Haskell Logo

Haskell, Miranda (1980ies)
➡ purely functional
➡ Compiler disallows side effects

Referential Transparency

- Bindings are immutable.
- Any name may be substituted by the value bound to that name and **not alter the semantics** of the expression.
- **"no side effects."**
  - This means no "printf() debugging!"
- Functional programing languages **encourage referential transparency**.
- **Pure** functional programming languages **enforce** referential transparency.

Referential Transparency

If two expressions are defined to have equal values, then one can be substituted for the other in any expression without affecting the result of the computation.

```
sumOfSquares :: Int -> Int -> Int
sumOfSquares x y =
    let
        x2 = x * x
        y2 = y * y
    in
      x2 + y2


sumOfSquares' :: Int -> Int -> Int
sumOfSquares' x y = (x * x) + (y * y)
```

Semantically equivalent.

Referentially Transparent Functions

*aka **pure** functions*

- A function is called **referentially transparent** if given the same argument(s), it always **returns the same result**.
- In **mathematics, all functions** are referentially transparent.
- In **programming this is not always the case**, with use of imperative features in languages.
  - The subroutine/function called could affect some global variable that will cause a second invocation to return a different value.
  - Input/Output

Evaluation (Scott)

Advantages

- Lack of side effects makes programs easier to understand.
- Lack of explicit evaluation order (in some languages) offers possibility of **parallel evaluation** (e.g. **Haskell**).
- Programs are often **surprisingly concise**.
- Language can be extremely small and yet powerful.

Problems

- Difficult (but not impossible!) to implement efficiently on von Neumann machines.
  - Naive impl.: Lots of copying, inefficient cache use, memory use.
- Requires a different mode of thinking by the programmer.
  - **Not necessarily a bad thing!**
- Difficult to integrate I/O into purely functional model.
  - Haskell: **Monads**.