Programming Languages
Lecture14 – Functional Languages

SUNG KYUN KWAN UNIVERSITY

남 범 석

bnam@skku.edu

“Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.”

Landin, Peter J. The next 700 programming languages. *Communications of the ACM,* 9(3):157–166, March 1966.

Imperative Programming Languages

- The design of the imperative languages is based directly on the *von Neumann architecture*
  - Efficiency is the primary concern
  - Needs to understand the machine architecture
  - Variables, conditional branching, iteration, procedures
  - Side-effects, state-based, assignment-oriented

- Imperative Languages
  - Ex. Fortran, Algol, Pascal, Ada, C, C++, Java

John von Neumann (1903-1957)
Creator of EDVAC

Historical Origins of Programming Languages

- The imperative and functional models
  - undertaken by Alan Turing, Alonzo Church, Stephen Kleene, Emil Post, etc. ~1930s
  - different formalizations of the notion of an algorithm, or *effective procedure*, based on automata, symbolic manipulation, recursive function definitions, and combinatorics

A. Turing

A. Church

S. Kleene

E. Post

A conjecture known as *Church-Turing thesis*

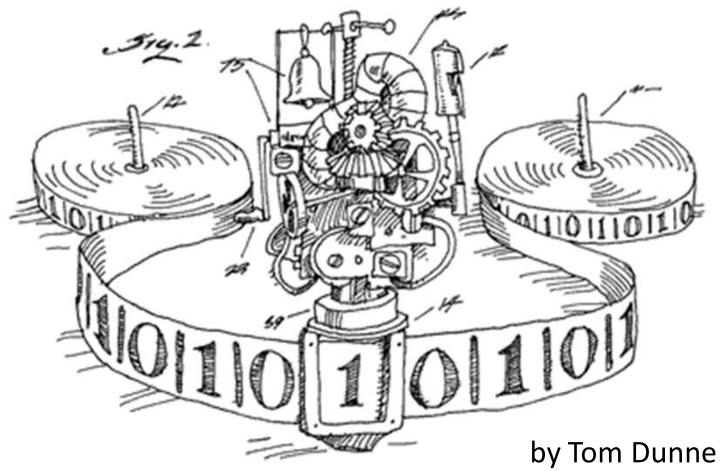
- "*Any* intuitively appealing computation model would be equally powerful as well"

- Turing Machine
  - Formal computation model for ***imperative*** programming languages

- Lambda Calculus (by Church)
  - Formal computation model for ***functional*** programming languages

Turing Machine

- Turing's model of computing
  - Pushdown automaton + an unbounded storage "tape"
  - Turing machine computes in an imperative way
    - By changing the values in cells of its tape – like variables just as a high level imperative program computes by changing the values of variables



by Tom Dunne

- Church’s model of computing
  - Notion of parameterized expressions
    - With each parameter introduced by an occurrence of the letter λ - hence the notation’s name.
  - λ-calculus was the inspiration for functional programming
  - Key idea: do everything by composing functions
    - No mutable state
    - No side effects

  - Functional languages such as Lisp, Scheme, FP, ML, Miranda, and Haskell are attempts to realize Church's λ-calculus in practical form as a programming language

Functional Programming Concepts

- Necessary features, many of which are missing in some imperative languages
  - 1st class and high-order functions
  - serious polymorphism
  - powerful list facilities
  - structured function returns
  - fully general aggregates
  - garbage collection

Recursion in Functional Languages

- How get anything done in a functional language?
  - Recursion (especially tail recursion) takes the place of iteration
  - In general, you can get the effect of a series of assignments

```
x := 0      ...
x := expr1       ...
x := expr2       ...
```

from f3(f2(f1(0))), **where each** f **expects the value of** x **as an argument,** f1 **returns** expr1**, and** f2 **returns** expr2

Recursion in FL (cont'd)

- Recursion even does a nifty job of replacing a looping

```
x := 0; i := 1; j := 100;
while i < j do
       x := x + i*j;
       i := i + 1;
       j := j - 1
end while
return x
```

becomes f(0,1,100) , where

```
f(x,i,j) == if i < j then
                f(x+i*j, i+1, j-1)
             else x
```

Fundamentals of FPL

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
  - In an imperative language, operations are done and the results are stored in variables for later use
  - Management of variables is a constant concern and source of complexity for imperative programming
- In an FPL, variables are not necessary, as is the case in mathematics

Mathematical Functions

- A mathematical function is a *mapping* of members of one set, called the *domain set*, to another set, called the *range set*
- A *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

$$\lambda x.\ x * x * x$$

for the function cube (x) = x * x * x

Lambda calculus

- Humans can give meaning to those symbols in a way that corresponds to computations.
- We write "$\lambda n. \ldots$" as a shorthand for

  "The function that, for each n, yields...,"
- **factorial** = λn. if $n=0$ then 1 else n * **factorial**($n-1$)
- The *lambda-calculus* (or λ-calculus) embodies this kind of function definition and application in the purest possible form

- ***Definition of Calculus:***
  Calculus is just a bunch of rules for manipulating symbols.

Lambda calculus

- Syntax:
  - f(x) = x + 3: λx. x+3
  - f(3) (λx. x+3) 3
  - f(x) = x^2 λx. x*x

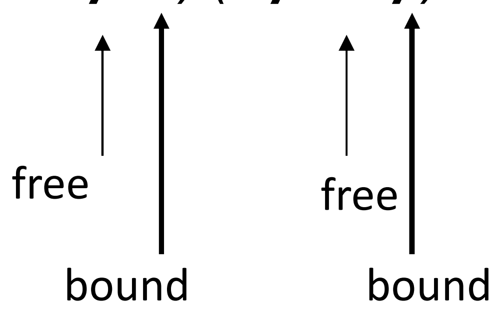
- The only keywords used in the language are λ and dot
- An expression can be surrounded with parenthesis for clarity

- In pure lambda calculus, EVERY function takes one (and only one) argument

Scope in Lambda calculus

- In λ calculus all names are local to definitions
- $\lambda x.t$: The scope of *x* is the term *t*
- An occurrence of the variable *x* is said to be *bound* when it occurs in the body *t* of an abstraction $\lambda x.t$

$$(\lambda x.\ y\ x)(\lambda y.\ x\ y)$$



Lambda calculus

- So how to do functions with multiple arguments?
  - f(x,y) = x-y   λ x. λ y. x-y
    - This is really a function of a function
    - (λ x. λ y. x-y) 7 2   yields f(7,2) = 7-2 = 5
    - (λ x. λ y. x-y) 7   yields f(7,y) = 7-y

- Note that when supplying only one parameter, we end up with a function
  - Where the other parameter is supplied
- This is called *currying*
  - A function can return a value *OR* a function

Lambda Expressions

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

  e.g., `(λ(x) x * x * x)(2)`

  which evaluates to `8`

Functions

- No side effects (this is how we would like our Scheme functions to behave also)
  - *Referential Transparency* - In an FPL, the evaluation of a function always produces the same result given the same parameters

- Can be composed
  - The result of one function can be the input to another function. (+ 3 (* 4 5))

Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

  Form: `h ≡ f ° g`

  which means `h (x) ≡ f ( g ( x))`

  For `f (x) ≡ x + 2` and `g (x) ≡ 3 * x`,

  `h ≡ f ° g` yields `(3 * x)+ 2`

Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

  Form: α

  For `h (x) ≡ x * x`

  α`( h, (2, 3, 4))` yields `(4, 9, 16)`

Data Types and Structures

- *Data object types*: originally only atoms and lists
- *List form*: parenthesized collections of sublists and/or atoms

  e.g., `(A B (C D) E)`
- Lists are stored internally as single-linked lists
- Originally, LISP was a typeless language

List Interpretation

- Lambda notation is used to specify functions and function definitions. Function applications and data have the same form.
- e.g., If the list `(A B C)` is interpreted as data. It is a simple list of three atoms, `A`, `B`, and `C`
- If it is interpreted as a function application, it means that the function named `A` is applied to the two parameters, `B` and `C`

Evaluation

- Parameters are evaluated, in no particular order
- The values of the parameters are substituted into the function body
- The function body is evaluated
- The value of the last expression in the body is the value of the function

LISP

- Lisp is an old language with many variants
- Lisp is alive and well today
  - Most modern versions are based on Common Lisp

- Variants of LISP
  - Pure Lisp
  - Interlisp, MacLisp, Emacs Lisp
  - Common Lisp
  - Scheme

Functional Programming Concepts

- Pure Lisp is purely functional
  - All other Lisps have imperative features
- All early Lisps dynamically scoped
  - Not clear whether this was deliberate or if it happened by accident
- Scheme and Common Lisp statically scoped
  - Common Lisp provides dynamic scope as an option for explicitly-declared special functions
  - Common Lisp is now THE standard Lisp