Programming Languages & Paradigms
PROP HT 2011

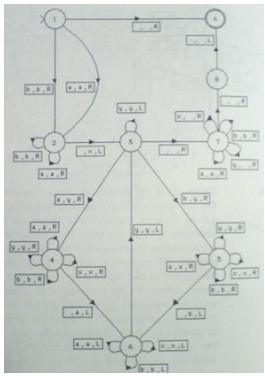
Lecture 12

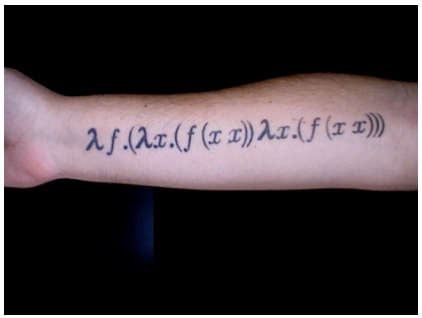
Functional Programming I: Foundations – history and lambda calculus

Beatrice Åkerblom
beatrice@dsv.su.se

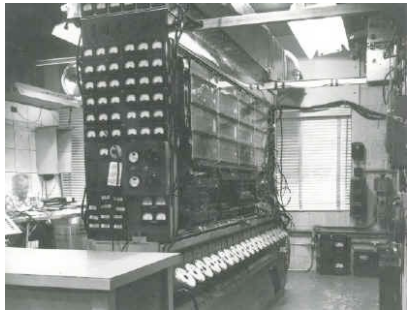
What is a Programming Language?

- The design of the **imperative** languages is based directly on the von Neumann architecture
  - *Efficiency* is the primary concern, rather than the suitability of the language for software development
- The design of **object-oriented** languages is based on cognitive psychology and philosophy
  - *Abstraction and classification* is the primary concern, rather than efficiency
- The design of the **functional languages** is based on mathematical functions
  - A *solid theoretical basis* combined with mechanisms to create *abstractions*, relatively unconcerned with the architecture of the machines on which programs will run

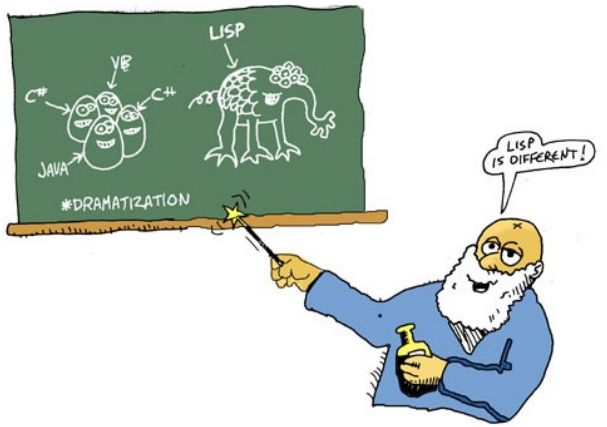




EDVAC

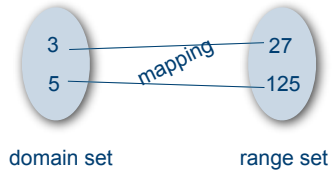


Lisp



Mathematical Functions

- A mathematical function is a mapping of members of one set, called the **domain** set, to another set, called the **range** set



- Define a function

```
real cube (real x)  { return x * x * x; }
```

```
cube (x) ≡ x * x * x
```


What is functional programming?

- At a high level: functional programming focuses on building functions.
- The programmer declares what the program does by defining a function that maps inputs to outputs.
- Complex functions are built by composing simpler functions.
  - `cube (x) ≡ x * x * x`
  - `sumCube (x,y) ≡ cube(x) + cube(y)`
- Generally this means functions in the mathematical sense:
  - In particular, variables are not modified by the code.
  - Instead variables are just names for values.

Lambda Expressions

- The central concept in λ calculus is the "expression"
- A "name", also called a "variable", is an identifier which, for our purposes, can be any of the letters `a, b, c, . . .`

```
<expression> := <name> | <function> | <application>
<function> := λ <name>.<expression>
<application> := <expression><expression>
```

- The only keywords used in the language are λ and the dot

Lambda Expressions, cont'd

- Function application associates from the left, that is, the expression but parentheses can be used to change the order

$$E_1, E_2, E_3, ...E_n$$

will be evaluated as:

$$((((E_1, E_2), E_3), ...), E_n)$$

Lambda Expressions, cont'd

- A function – a lambda abstraction:

$$\lambda x.x \qquad \lambda x.y$$

- Application:

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

- Substitution:

$$(\lambda x.x)y = [y/x]x = y$$

Lambda Expressions, cont'd

- Names don't carry any meaning and are local to definition, alpha equivalence:

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

Lambda Expressions, cont'd

- Names don't carry any meaning and are local to definition, alpha equivalence:

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

- Free and bound variables:

$(\lambda x.xy)$ — bound, free

Lambda Expressions, cont'd

- Names don't carry any meaning and are local to definition, alpha equivalence:

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

- Free and bound variables:

$$(\lambda x.xy)$$

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

Bound and free, but no connection

Lambda Expressions, cont'd

- In the expression

$$(\lambda x.xy)$$

  - y is free because there is no enclosing λy
  - the expression is undefined until y is given a definition

  - x is "defined" by its occurrence in λx
  - any meaning it had outside the expression (λx. x y) is hidden ("shadowed")
  - first x is the binding occurrence, second x is a bound occurrence
  - an expression with no free variables is closed

Lambda Expressions, cont'd

- α-conversion

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

  - the only variables renamed are those bound to the same abstraction

$$\lambda x.\lambda x.x \equiv \lambda y.\lambda x.x$$

Lambda Expressions, cont'd

- α-conversion

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

  – the only variables renamed are those bound to the same abstraction

$$\lambda x.\lambda x.x \equiv \lambda y.\lambda x.x \not\equiv \lambda x.\lambda x.x$$

Lambda Expressions, cont'd

- α-conversion

$$(\lambda z.z) \equiv (\lambda y.y) \equiv (\lambda t.t) \equiv (\lambda u.u)$$

  – the only variables renamed are those bound to the same abstraction

$$\lambda x.\lambda x.x \equiv \lambda y.\lambda x.x \not\equiv \lambda x.\lambda x.x$$

  – not possible if it results in a variable getting captured by a different abstraction

$$\lambda x.\lambda y.x$$

Can't be replaced with y

Lambda Expressions, cont'd

- Example:

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

Lambda Expressions, cont'd

- Example:

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

$$(\lambda x.x)(\lambda z.z)$$

Lambda Expressions, cont'd

- Example:

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

$$(\lambda x.x)(\lambda z.z)$$

$$[\lambda z.z/x]x$$

Lambda Expressions, cont'd

- Example:

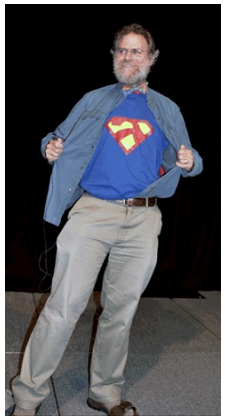
$$(\lambda x.x)(\lambda x.x)$$

$$(\lambda x.x)(\lambda z.z)$$

$$[\lambda z.z/x]x$$

$$\lambda z.z$$

Philip Wadler
a.k.a Lambda Man



Lambda Expressions, cont'd

- Be careful with substitutions:

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

Stockholm University

Lambda Expressions, cont'd

- Be careful with substitutions:

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

$$\lambda y.yy \longleftarrow \text{no!}$$

Stockholm University

Lambda Expressions, cont'd

- Be careful with substitutions:

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

$$\lambda y.yy \longleftarrow \text{no!}$$

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

Stockholm University

Lambda Expressions, cont'd

- Be careful with substitutions:

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

$$\lambda y.yy \longleftarrow \text{no!}$$

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

$$\lambda t.yt$$

Stockholm University

Lambda Expressions, cont'd

- Be careful with substitutions, again:

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

Lambda Expressions, cont'd

- Be careful with substitutions, again:

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

only free x

Lambda Expressions, cont'd

- Be careful with substitutions, again:

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

only free x

wait a second!

Lambda Expressions, cont'd

- Be careful with substitutions, again:

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

only free x

wait a second!

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

Lambda Expressions, cont'd

- Be careful with substitutions, again:

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

only free x

wait a second!

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

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

Confluence

$(\lambda x.xy)((\lambda z.z)u)$ $\quad$ $(\lambda x.xy)((\lambda z.z)u)$

$\hookrightarrow (\lambda x.xy)u$ $\quad$ $\hookrightarrow (\lambda z.z)u)y$

$\hookrightarrow uy$ $\quad$ $\hookrightarrow uy$

Shorter and Fewer Parentheses

$\lambda t.(\lambda x.xt)$

can be abbreviated as

$\lambda tx.xt$

Church numerals

- Logically, we define natural numbers inductively:
  - `zero` is a natural number.
  - if *n* is a natural number then (`succ` *n*) is also a natural number.

0 = `zero`
1 = `succ zero`
2 = `succ (succ zero)`
3 = `succ (succ (succ zero))`
4 = ...

- But how do we define `succ` and `zero`?

Church numerals

- A natural number *n* is represented as a higher order function, taking a value and a function *f* as arguments, where *n* should be the "starting value" and *f* will then be applied on the value *n* times

$0 \equiv \lambda fn.n$
$1 \equiv \lambda fn.fn$
$2 \equiv \lambda fn.f(f(n))$
$3 \equiv \lambda fn.f(f(f(n)))$
$\ldots$

Successor

- Successor function

number for which we want to find successor
function (same as for numbers)
value (same as for numbers)

$$S \equiv \lambda nfx.f(nfx)$$

Successor

- Successor function

number for which we want to find successor
function (same as for numbers)
value (same as for numbers)

$$S \equiv \lambda nfx.f(nfx)$$

- applied to our representation for zero

$$S \equiv (\lambda nfx.f(nfx))(\lambda fn.n)$$

Successor

- Successor function

number for which we want to find successor
function (same as before)
value (same as before)

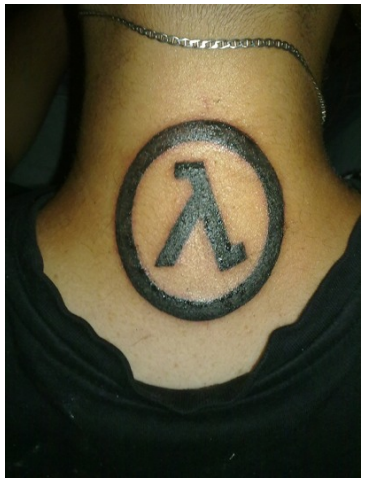
$$S \equiv \lambda nfx.f(nfx)$$

- applied to our representation for zero

$$S \equiv (\lambda nfx.f(nfx))(\lambda fn.n)$$

- needs some renaming

$$S \equiv (\lambda nfx.f(nfx))(\lambda go.o)$$



Successor, cont'd

$$S \equiv (\lambda nfx.f(nfx))(\lambda go.o)$$

- substitute all occurrences of *n* with (λgo.o)

$$\lambda fx.f((\lambda go.o)fx)$$

Successor, cont'd

$$S \equiv (\lambda nfx.f(nfx))(\lambda go.o)$$

- substitute all occurrences of *n* with (λgo.o)

$$\lambda fx.f((\lambda go.o)fx)$$

- substitute all occurrences of *g* with *f*

$$\lambda fx.f((\lambda o.o)x)$$

Successor, cont'd

$$S \equiv (\lambda nfx.f(nfx))(\lambda go.o)$$

- substitute all occurrences of *n* with (λgo.o)

$$\lambda fx.f((\lambda go.o)fx)$$

- substitute all occurrences of *g* with *f*

$$\lambda fx.f((\lambda o.o)x)$$

- substitute all occurrences of *o* with *x*

$$\lambda fx.f(x)$$

Successor, cont'd

$$S \equiv (\lambda nfx.f(nfx))(\lambda go.o)$$

- substitute all occurrences of *n* with (λgo.o)

$$\lambda fx.f((\lambda go.o)fx)$$

- substitute all occurrences of *g* with *f*

$$\lambda fx.f((\lambda o.o)x)$$

- substitute all occurrences of *o* with *x*

$\lambda fx.f(x)$ ← Looks familiar...

Successor, cont'd

$$S \equiv (\lambda nfx.f(nfx))(\lambda go.o)$$

- substitute all occurrences of *n* with (λgo.o)

$$\lambda fx.f((\lambda go.o)fx)$$

- substitute all occurrences of *g* with *f*

$$\lambda fx.f((\lambda o.o)x)$$

- substitute all occurrences of *o* with *x*

$$\lambda fx.f(x) \equiv 1$$

Addition

$2 + 3$:

use the number 3 as "starting point" when "creating" number 2 by using our successor function

$$(\lambda fn.f(fn))(\lambda wyx.y(wyx))(\lambda uv.u(u(uv)))$$

"how to create 2" — successor — where to start (3)

$$(\lambda wyx.y((wy)x))((\lambda wyx.y((wy)x))(\lambda uv.u(u(uv))))$$

$$(\lambda wyx.y((wy)x))((\lambda wyx.y((wy)x))(\lambda uv.u(u(uv))))$$

$$(\lambda wyx.y((wy)x))((\lambda yx.y(((\lambda uv.u(u(uv)))y)x)))$$

$$(\lambda wyx.y((wy)x))((\lambda yx.y((\lambda v.y(y(yv)))x)))$$

$$(\lambda wyx.y((wy)x))((\lambda yx.y(y(y(yx)))))$$

$$\lambda wyx.y((wy)x)(\lambda fn.f(f(f(fn))))$$

$$\lambda yx.y(((\lambda fn.f(f(f(fn)))y)x))$$

$$\lambda yx.y(\lambda n.y(y(y(yn)x)))$$

$$\lambda yx.y(((\lambda n.y(y(y(yn))))x))$$

$$\lambda yx.y(y(y(yx)))$$

Added to presentation after the lecture, only used as preparation notes.

Booleans

- A Boolean value expresses a choice between two options:

$$true \equiv \lambda xy.x \qquad false \equiv \lambda xy.y$$

- We can then define logical operations

$$\wedge \equiv \lambda xy.xyx$$

$$\vee \equiv \lambda xy.xxy$$

$$\neg \equiv \lambda xyz.xzy$$

Stockholm University

And

$$true \equiv \lambda xy.x \qquad false \equiv \lambda xy.y$$

- And true false:

$$(\lambda xy.xyx)\ \underline{(\lambda mn.m)\ (\lambda ab.b)}$$

$$(\lambda mn.m)\ \underline{(\lambda ab.b)\ (\lambda mn.m)}$$

$$(\lambda ab.b) \equiv false$$

Stockholm University

Or

$$true \equiv \lambda xy.x \qquad false \equiv \lambda xy.y$$

- Or true false:

$$(\lambda xy.xxy)\ \underline{(\lambda mn.m)\ (\lambda ab.b)}$$

$$(\lambda mn.m)\ \underline{(\lambda mn.m)\ (\lambda ab.b)}$$

$$(\lambda mn.m) \equiv true$$

The End

Stockholm University

References

- Sebesta, R., "Concepts of Programming Languages"
- Biancuzzi, F. and Warden, S.,"Masterminds of Programming – Conversations with the Creators of Major Programming Languages", 2009

And, e.g. the below and other material:

- http://www.mactech.com/articles/mactech/Vol.07/07.06/ChurchNumerals/index.html
- http://www.utdallas.edu/~gupta/courses/apl/lambda.pdf
- http://cs.anu.edu.au/student/comp2600/lectures/Lambda-Church-2x2.pdf
- http://www.lambda-bound.com/book/lambdacalc/node1.html