Lambda Calculus

λ– Calculus History

- Developed by **Alonzo Church** during 1930's-40's
- One fundamental goal was to describe what can be computed.
- Full definition of λ-calculus is equivalent in power to a Turing machine
  - **Turing machines and λ-calculus are alternate descriptions of our understanding of what is computable**

λ– Calculus History – 2

- In the mid to late 1950's, **John McCarthy** developed **Lisp**
  - **A programming language based on λ-calculus**
  - **Implementation includes syntactic sugar**
    - **functions and forms that do not add to the power of what we can compute but make programs simpler and easier to understand**

λ– Calculus Basis

- Mathematical theory for **anonymous** functions
  - **functions that have not been bound to names**
- Present a subset of full definition to present the flavour
- Notation and interpretation scheme identifies
  - **functions and their application to operands**
    - **argument-parameter binding**
  - **Clearly indicates which variables are free and which are bound**

Bound and Free Variables

◊ Bound variables are similar to local variables in Java function (or any procedural language)

» **Changing the name of a bound variable (consistently) does not change the semantics (meaning) of a function**

◊ Free variables are similar to global variables in Java function (or any procedural language)

» **Changing the name of a free variable normally changes the semantics (meaning) of a function.**

λ–functions – 1

◊ Consider following expression

» (u + 1) (u – 1)

» is u bound or free?

◊ Disambiguate the expression with the following λ-function

» (λ u . (u + 1) (u – 1))

bound variables   defining form

» Clearly indicates that u is a bound variable

◊ Note the parallel with programming language functions

» functionName (arguments) { function definition }

– It seems obvious now but that is because programming languages developed out of these mathematical notions

λ–functions – 2

- Consider the following expression
  - » (u + a) (u + b)
- Can have any of the following functions, depending on what you mean
  - » (λ u . (u + a) (u + b))
    - > u is bound, a and b are free (defined in the enclosing context)
  - » (λ u, b . (u + a) (u + b))
    - > u and b are bound, a is free
  - » (λ u, a, b . (u + a) (u + b))
    - > u, a and b are all bound, no free variables in the expression

Function application

- Functions are applied to arguments in a list immediately following the l-function
  - » $\{ \lambda u . (u + 1) (u + 2) \} [3]$
    - > 3 ==> u then ==> (3 + 1) (3 + 2) ==> 20
  - » $\{ \lambda u . (u + a) (u + b) \} [7 – 1]$
    - > 7–1 ==> u then ==> (6 + a) (6 + b) and no further in this context
  - » $\{\lambda u, v . (u – v) (u + v) \}\ [2p + q , 2p - q]$
    - > ==> ((2p+q) – (2p - q)) ((2p + q) + (2p – q))
    - > Can pass expressions to a variable
- Can use different bracketing symbols for visual clarity; they all mean the same thing.

Using auxiliary definitions

◊ Build up longer definitions with auxiliary definitions

» **Define $u / (u + 5)$**
where $u = a (a + 1)$
where $a = 7 - 3$

$$\underbrace{\{ \lambda u \,.\, u / (u + 5) \}}\ \underbrace{[\, \underline{\{ \lambda a \,.\, a (a + 1) \}}\ \underline{[\, 7 - 3 \,]}\ \,]}$$

> **Note the nested function definition and argument application**

$$\Longrightarrow \{ \lambda u \,.\, u / (u + 5) \} [\, 4 (4 + 1) \,]$$

$$\Longrightarrow \{ 20 / (20 + 5) \}$$

$$\Longrightarrow 0.8$$

Functions are Variables

- Define $f(3) + f(5)$
  where $f(x) = a\,x\,(a + x)$
  where $a = 4$

$$\{\lambda f\ .\ f(3) + f(5)\}\,[\,\{\lambda a\ .\ \{\lambda x\ .\ a\,x\,(a + x)\}\}\,[4]\,]$$

- Arguments must be evaluated first

$$\Longrightarrow \{\lambda f\ .\ f(3) + f(5)\}\,[\,\{\lambda x\ .\ 4\,x\,(4 + x)\}\,]$$

$$\Longrightarrow \{\lambda x\ .\ 4\,x\,(4 + x)\}\ (3)\ +\ \{\lambda x\ .\ 4\,x\,(4 + x)\}\ (5)$$

$$\Longrightarrow\ 4 * 3\,(4 + 3)\ + 4 * 5\,(4 + 5)\ \ \Longrightarrow\ 264$$

Lamba notation in Lisp

- Lambda expressions are a direct analogue of λ-calculus expressions
  - **They are the basis of Lisp functions – a modified syntax to simplify the interpreter**
- For example

  (defun double (x) (+ x x))

  - **is the named version of the following unnamed lambda expression**

  (lambda (x) (+ x x)) —— **{ λ x . (x + x) }**

  - **Note the similar syntax with λ-calculus and the change to prefix, from infix, to permit a uniform syntax for functions of any number of arguments**

Anonymous functions

◊ Recall in the abstraction for sumint we defined support functions to handle each case

```
(defun double (int) (+ int int))
(defun square (int) (* int int))
(defun identity (int) int)
```

◊ This adds additional symbols we may not want, especially if the function is to be used only once.

◊ Using lambda we get the same effect without adding symbols

```
(sumint #'(lambda (int) (+ int int)) 10)
(sumint #'(lambda (int) (* int int)) 10)
(sumint #'(lambda (int) int) 10)
```

The function ‘function’

- What is the meaning of `#'` in the following

  ```
  (sumint #'(lambda (int) (+ int int)) 10)
  ```

- It is a short hand
  - » `#'(...) ==> (function (...))`

- One of its attributes is it works like quote, in that its argument is not evaluated, thus, in this simple context the following will also work

  ```
  (sumint '(lambda (int) (+ int int)) 10)
  ```

- Later we will see another attribute of **function** that makes it different from **quote**.

- Whenever a function is to be quoted use `#'` in place of `'`

Recursion

◊ Recursion with lambda functions uses labels to temporarily name a function

◊ The following is a general λ-calculus template.

> **The name is in scope within the entire body but is out of scope outside of the lambda expression.**

{ label name (lambda arguments .
body_references_name) }

◊ In Lisp can use labels to define a mutually recursive set of functions

(labels (list of named lambda expressions)
sequence of forms using the temporarily named functions
)

Example 1 of recursion

◊ A recursive multiply that uses only addition.

> The temporary function is called mult

> Use quote not function – using eval

```
(eval '(labels
          ((mult (k n)
             (cond ((zerop n) 0)
                   (t (+ k (mult k (1- n))))
          )))
       (mult 2 3)
     )
)
```

Example 2 of recursion

◊ **recTimes** computes **k * n** by supplying the paramters to a unary function that is a variation of example 1.

```
(defun recTimes (k n)
  (labels (( temp (n)
             (cond ((zerop n) 0)
                   ( t (+ k (temp (1- n))))
          )))
  (temp n)
))
```