

3/10/2008

1

$$f = \lambda n. \text{IFTHENELSE} (\text{ISZERO } n) 1 \\ (\text{TIMES } n (f (\text{MINUS } n 1)))$$

$$g = \text{df. } \lambda n. \text{IFTHENELSE} \dots$$

$g h 5$ needs to be 120

h needs to expand into $g h$

what is h ?

$$h = Y g$$

$$Y = \lambda g. (\lambda x. g (x x)) (\lambda x. g (x x))$$

$$\begin{aligned} Y g &= (\lambda x. g (x x)) (\lambda x. g (x x)) \\ &= g ((\lambda x. g (x x)) (\lambda x. g (x x))) \\ &= g (Y g) \end{aligned}$$

solution is $h = Y g$

$$\begin{aligned}
 & g \ h \ 3 \\
 &= g \ (\gamma g) \ 3 \\
 &= \text{IFTHENELSE } (\text{ISZERO } 3) \ 1 \\
 &\quad (\text{TIMES } 3 ((\gamma g) \ (\text{MINUS } 3 \ 1))) \\
 &= (\text{TIMES } (g \ (\gamma g) \ 2)) \\
 &= (\text{TIMES } 3 \ (\text{TIMES } 2 \ (g \ (\gamma g) \ 1))) \\
 &= (\text{TIMES } 3 \ (\text{TIMES } 2 \ (\text{TIMES } 1 \ (g \ (\gamma g) \ \emptyset)))) \\
 &= (\text{TIMES } 3 \ (\text{TIMES } 2 \ (\text{TIMES } 1 \ 1))) \\
 &= 6
 \end{aligned}$$

Theorem : the λ calculus can compute any known function

History : John McCarthy in 1950s invented a language based on λ calculus : LISP (List Processor)

We will look at a modern version of LISP called Scheme.

Scheme is a pure functional language (actually not quite)

MoC is evaluation of expressions

An expression is a fully parenthesized prefix form:

(\uparrow 1 2)
function name arguments

Scheme has an interpreter: typing (+ 1 2) into the interpreter produces 3 as the result

$$(+ (* 2 4) 3) \Rightarrow 11$$

We can have anonymous functions with LAMBDA

$$((\text{LAMBDA } (x) (+ x 1)))$$

function \uparrow
 argument

$$= (+ 3 1)$$

$$= \text{f}$$

Functions can be first-class arguments

$$((\text{LAMBDA } (f) (f 3))) (\text{LAMBDA } (x) (+ x 1))$$

function \uparrow
 argument

$$= ((\text{LAMBDA } (x) (+ x 1)) 3)$$

$$= (+ 3 1)$$

$$= \text{f}$$

Scheme has a binding mechanism called define

e.g. (define x 1) binds x to 1

(define y (+ x 3)) binds y to 4

We can also bind functions

(define addOne (lambda (x) (+ x 1)))

this binds addOne to the function which returns a value that is one more than its argument.

Then we can do :

(addOne 3) => 4

We can redefine any number of times

(define addOne 1)

Then (addOne 1) produces an error because addOne is not bound to a function

Conditional form :

(if <test> <then> <else>)

The test evaluates to either #t (true) or #f (false)

If the test evaluates to #t then the value returned by if is the value of the <then> expression, otherwise the value returned is the value of the <else> expression.

(define n 1) [returns nothing]

(if (= n 1) 3 +)
↑ ↑
Then else

Factorial function :

(define fact

(lambda (n)

(if (= n 0)

1

(* n (fact (- n 1)))))

e.g. (fact 3) we will derive the value of this expression

$$= (\text{if } (= 3 0) 1 (* 3 (\text{fact} (- 3 1))))$$

$$= (* 3 (\text{fact} 2))$$

$$= (* 3 (* 2 (\text{fact} 1)))$$

$$= (* 3 (* 2 (* 1 (\text{fact} 0))))$$

$$= (* 3 (* 2 (* 1 1)))$$

$$= 6$$

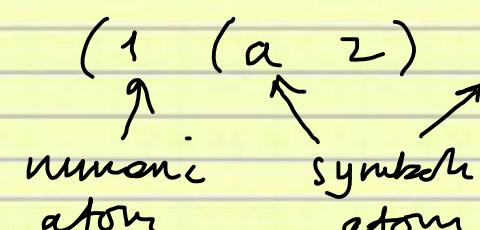
Scheme also has an important data structure called the 'list'

A list has the same syntax as an expression:

Thus $(1 \ 2 \ 3)$ is a list of three numbers

Strictly speaking a list a sequence of either atoms or lists.

e.g. $(1 \ (a \ z) \ b)$ is a list



This gave nice symbolic programming

We need a way to distinguish lists from expressions.

The method used is called 'quoting'. We prefix a list with single quote mark,

so ' $(+ \ 1 \ 2)$ ' is a list consisting of three atoms
 $(+ \ 1 \ 2)$ is an expression with a value.

Notice we only need one quote mark because of the parens.

The quote mark is actually a shorthand for the special function quote

So `'(1 2 3)` is same as `(quote (1 2 3))`
↑
function that returns its argument unevaluated

Note the difference between :

`(define x '(+ 2 3))` which binds
x to `(+ 2 3)`

and `(define x (+ 2 3))` which binds
x to 5

The ' mark stops evaluation of an expression

We can also quote symbols :

(define x 'y) binds x to y

(define x y) binds x to y's value

Numeric atoms don't need quoting

(define x 3) binds x to 3

The list has four basic operations which treat it like a singly-linked list. We can only operate on the head of the list.

To return the head of a list, we can

e.g. (car '(1 2 3)) => 1

(car '(a b c)) => a

(car '((1) (2) (3))) => (1)

To return everything except the head use cdr

e.g. (cdr '(a b c)) returns (b c)

(cdr '((a) (b) (c))) returns ((b) (c))

car = head (first)

cdr = tail (rest)

We could redefine car, cdr

(define first car)

(define rest cdr)

Historically LISP was implemented on a 509 (?)

IBM machine with an address register and a
decrement register

c - contents

a - address

r - register

c - contents

d - decrement

r - register