LECTURE 3 : CONTINUE WITH SCHEME

;;; Scheme

;;; - dialect of lisp, meaning:

;;; - highly parenthesis rich syntax

;;; - datatype: list and s-expression

;;; - has lambda aka λ for making function objects

;;; - DYNAMICALLY TYPED

;;; - simple,

;;; - good for teaching, writing compilers,

;;; - easy to extend, etc

;;; - eager evaluation -> arguments of function are calculated beforehand

Eager eg.

(define k (λ (x y) x))

;; > (k 2 3)

;; 2

;; > (k 2 (/ 1 0))

;; ; /: division by zero [,bt for context] //we get an error because its eager

Does Scheme have aggregate data structures? Yes

(define kons (λ (x y) (λ (z) (if z x y))))

(define fst (λ (p) (p #t)))

(define snd (λ (p) (p #f)))

;; > (fst (kons 3 4)) //first of these two is 3

;; 3

;; > (snd (kons 3 4)) //second is 4

;; 4

;; > ((fst (kons sin sqrt)) 4)

;; -0.7568024953079282

;; > (define eh (kons 3 (kons 4 (kons (kons 5 6) 7))))

;; > (fst eh)

;; 3

;; > (snd eh)

;; #<procedure:...4/lecture-03.scm:15:22>

;; > (fst (snd eh))

;; 4

;; > (fst (snd (snd (snd eh))))

;; 7

;; > (fst (fst (snd (snd eh))))

;; 5

This particular way of doing these pairs is called church encoding pairs

Some compilers take pairs and turn them into these church encoding pairs

Every identifier we’ve used can be used as a symbol

;; > sqrt

;; #<procedure:sqrt>

;; > (quote sqrt) //this is a symbol

;; sqrt

;; > (quote ()) //creates empty list

;; ()

;; > null //creates empty list

;; ()

;; > (cons 3 null) //creates list with one element ‘3’

;; (3)

;; > (cons (quote snd) (cons (quote eh) null)) //list of two elements that contain 2 symbols [snd , eh]

;; (snd eh)

;; > (define baz (cons (quote snd) (cons (quote eh) null)))

;; > baz

;; (snd eh)

;; > (**car** baz) // car returns head of list

;; snd

;; > (**cdr** baz) //cdr retuns the remains of the list after the head.

;; (eh)

// car and cdr don’t destroy anything

;; > (define zonk (cons (quote snd) (cons (quote eh) (cons (quote aye) (cons (quote bee) (cons (quote sea) null))))))

;; > zonk

;; (snd eh aye bee sea)

;; > (car (cdr (cdr zonk)))

;; aye

;;; car, cdr

;;; (define caar (λ (x) (car (car x))))

;;; ...

;;; (define caddr (λ (x) (car (cdr (cdr x)))))

;;; ...

;;; (define cddddr (λ (x) (cdr (cdr (cdr (cdr x))))))

// can put anything after quote and that object will be returned

;; > (quote sqrt)

;; sqrt

;; > (quote ())

;; ()

;; > (quote (aye bee sea dee)) //constant list

;; (aye bee sea dee)

;; > '(aye bee sea dee) ‘ is the same as quote

;; (aye bee sea dee)

;; > '(aye (bee sea dee eee))

;; (aye (bee sea dee eee)) //this list has 2 elements, 1st elem = aye, 2nd elem = another list

;; > null? //this procedure tests whether list is empty list

;; #<procedure:null?>

;; > (null? 13)

;; #f //not empty

;; > (null? 'null)

;; #f

;; > (null? null) //global null is null

;; #t

;; > (null? '()) empty list is empty

;; #t

;; > (null? (quote ()))

;; #t

;; > eh

;; #<procedure:...4/lecture-03.scm:15:22>

;; > zonk

;; (snd eh aye bee sea)

;; > (null? zonk)

;; #f

;; > (null? (cdr (cddddr zonk)))

;; #t

;; > **list?** // used to distinguish whether something is a list

;; #<procedure:list?>

;; > (list? 3)

;; #f

;; > (list? null)

;; #t

;; > (list? '(not a list))

;; #t

;; > (**pair?)** //checks if non empty list

;; > (pair? '(not a list))

;; #t

;; > (pair? '())

;; #f

;; > (pair? 42)

;; #f

WRITE FUNCTION TO GET LENGTH OF LIST

//Some properties

;;; length of empty list is zero.

;;; length of non-empty list is one plus length of its tail.

(define my-length

(λ (xs)

(if (null? xs)

0

(+ 1 (my-length (cdr xs))))))

;; > (my-length zonk)

;; 5

;; > zonk

;; (snd eh aye bee sea)

;; > (my-length (cdr zonk))

;; 4

;; >