***LISP***

Intro

;(load "IntroductionDemo.lisp")

(list "horse" (list "cow" () (list "dog" () ()) ) (list "zebra" (list "yak" () ()) () ) )

;Cons Cells

(defvar \*fooCell\* (cons 2 4))

\*fooCell\*

(car \*fooCell\*)

(cdr \*fooCell\*)

;Lists in Common Lisp

(defvar \*list\* (list 1 2 3 4))

\*list\*

(car \*list\*)

(cdr \*list\*)

(car (cdr \*list\*))

(car (cdr (cdr (cdr \*list\*))))

(cons 0 \*list\*)

;Constructing a list from cons cells

(cons "abc" (cons 7 (cons (cons 9 (cons 10 () ) ) () ) ) )

;Showing the quote operator

;(car (2 3 4)) ;Thows exception so commented out by default

(car '(2 3 4))

;Basic Math

5

(+ 3 3)

(/ 1 9)

(/ (\* (+ 2 2) (+ 5 3)))

;Sum of list (Hardcoded indices)

(+ (car \*list\*) (car (cdr \*list\*)) (car (cdr (cdr \*list\*))) (car (cdr (cdr (cdr \*list\*)))) )

;Mention the print function

(print "Hello! You need me 😅")

Selection Statements

;is even

(print "Enter a number")

(if

(= 0 (mod (parse-integer (read-line)) 2 ))

(print "is even") ;true branch

(print "is odd") ;false branch

)

;larger string

(print "Enter two strings to be compared")

(if

;conditional

(STRING> (read-line) (read-line))

;true branch

(print "Second is lexicographically larger")

;false branch

(print "First is lexicographically larger")

)

(print "Enter two numbers to be compared")

(if

(> (parse-integer (read-line)) (parse-integer (read-line))) ;conditional

(print "First is larger") ;true branch

(print "Second is larger") ;false branch

)

(defvar \*input\* (read-char))

(cond

((CHAR-EQUAL #\M \*input\*) (print #\M))

((CHAR-EQUAL #\S \*input\*) (print #\S))

((CHAR-EQUAL #\I \*input\*) (print #\I))

(T (print "Error, incorrect char input"))

)

Functions

(defun isEven (aNumber) "Checks if a given number is even"

(if

(= 0 (mod aNumber 2 ))

T ;true branch

NIL ;false branch

)

)

(defun stringComp (&key (a "") (b "") (toConcat "" toConcat-supplied-p) ) "Compares two Strings and first one alphabetically"

(if

;conditional

(STRING> a b)

;true branch

(if

;conditional

(not (null toConcat-supplied-p))

;true branch

(concatenate 'string b toConcat)

;false branch

b

)

;false branch

(if

;conditional

(not (null toConcat-supplied-p))

;true branch

(concatenate 'string a toConcat)

;false branch

a

)

)

)

(defun realComp (a b &optional offset) "Compares two real numbers and returns the larger one"

(cond

((and (> a b) (not (null offset))) (+ a offset) )

((and (> a b) (null offset)) a )

((and (< a b) (not (null offset))) (+ b offset) )

((and (< a b) (null offset)) b )

)

)

(defvar \*input\* (read-char))

(cond

((CHAR-EQUAL #\E \*input\*)

(isEven 3)

)

((CHAR-EQUAL #\S \*input\*)

(stringComp :a "Hello" :b "World")

)

((CHAR-EQUAL #\R \*input\*)

(realComp 2/3 3/2)

)

(T (print "Error, incorrect char input"))

)

High Order Functions

(defun map- (f L)

(if (null L) (RETURN-FROM map- (break)))

(cons (funcall f (car L)) (map- f (cdr L)))

)

;tail recursive

(defun map-tail (f L &optional L2)

(if (null L) (RETURN-FROM map-tail (reverse L2)))

(map-tail f (cdr L) (cons (funcall f (car L)) L2 ))

)

(defun square (x) (\* x x))

(defun square-list (L) (map-tail #'square L))

(square-list '(3 4 5)) ;prints '(9 16 25)

;example problem

(defun isString (x)

(typep x 'String)

)

(defun myExample (f L &optional L2)

(if (null L) (RETURN-FROM myExample (reverse L2)))

(if (funcall f (car L))

(myExample f (cdr L) (cons (concatenate 'String "OR" (car L)) L2))

(myExample f (cdr L) L2)

)

)

Iteration

;break

;backtrace

(defun square-list (L)

(if

;condition

(null L)

;true branch (base case)

()

;false branch (recursive case)

(cons (\* (car L) (car L)) (square-list(cdr L)))

)

)

;not tail recursive

(defun factorial (n)

(if

(= n 0)

1

(\* n (factorial(- n 1)))

)

)

;tail recursive

(defun factorialTail (n &optional (result 1))

(if

(= n 1)

(break)

(factorialTail (- n 1) (\* n result))

)

)

(defun addList (inlist)

(if

(null inlist)

0

(+ (car inlist) (addList (cdr inlist)))

)

)

(defun addListTail (inlist &optional (result 0))

(if

(null inlist)

result

(addListTail (cdr inlist) (+ (car inlist) result))

)

)

(defun myListLength (inlist)

(if (null inlist)

0

(+ 1 (myListLength (cdr inlist)))

)

)

(defun listLengthTail (inlist &optional (result 0))

(if

(null inlist)

result

(listLengthTail (cdr inlist) (+ 1 result))

)

)

(defun print-squares-cond (low high)

(cond

;if branch

((> low high) NIL)

;else branch

(T (print (\* low low)) (print-squares-cond (+ 1 low) high ))

)

)

(defun print-squares-if (low high)

;if statement with no else

(if (> low high) (RETURN-FROM print-squares-if (break)))

;statements to execute

(print (\* low low))

(print-squares-if (+ 1 low) high)

)

Let

(let( (MyList '("a" "b" ("c" "d"))) )

(print MyList)

)

(let ((a 2) (b 3))

(print (+ 1 b))

(print (+ 6 a))

)

(let ((circleArea (lambda (r) (\* 3.14 (\* r r)))))

(funcall circleArea 10) ;returns 314.0

)

;(circleArea 10) ;error, circleArea not defined

(let\* ((x 10) (y (+ x 10)))

(list x y)

)

(defun sumUserInput ()

(let (

;Could replace these read lines with function calls

(x (parse-integer (read-line)))

(y (parse-integer (read-line)))

)

(print (+ 0 y))

)

)

;Scoping

(let ((a 2) (b 3))

(let ((a (+ a b)))

(+ a b)

)

)

(let ((pi- 3.14))

(let ((circleArea (lambda (r) (\* pi- (\* r r)))))

(let ((p 3.1416))

(funcall circleArea 10)

)

)

)

(let\* ((a "a") (add-a

(lambda (x)

(if (and (numberp a) (numberp x))

(+ x a)

(format t "add-a: x or a is not a number x:~S a:~S" x a)

)

)

))

(funcall add-a 5)

)

;Closures

; (defvar x 5) ; x is a free variable in

; (defun addx (a) (+ a x)) ; function addx

; (addx 3) ; result: 8

; (let ((x 4)) (addx 3)) ; result: 7

; (let ((x 4)) (+ x 3)) ; result: 7

(let ((a 1))

(let ((f (lambda (x) (+ a x))))

(let ((a 2))

(funcall f 2)

)

)

)

(defvar a 5)

(defun f (x) (+ a x))

(f 2) ;returns 7

(let ((a 1))

(print (f 2)) ;prints 3

(let ((a 2))

(print (f 2)) ;prints 4

)

)

(defvar x 1)

(defun f (y)

(let ((x (+ y 1)))

(lambda (z) (+ x y z)))

)

(defvar z

(let ((x 3) (g (f 4)) (y 5))

(funcall g 6)

)

)

(defun f (g)

(let ((x 3))

(funcall g 2) ;replace 2 with x and the value of x

;is bound to the value of x in the let block

)

)

(defvar x 4)

(defun h (y) (+ x y))

(defvar z (f #'h))

z ;prints 6

;Currying

(defun f (a b) (+ a b))

(defun f1 (a) (lambda (b) (f a b)))

(f 2 3) ;returns 5

(funcall (f1 2) 3) ;returns 5

(defun quadratic (a b c x) (+ (\* a x x) (\* b x) c))

(quadratic 1 2 3 4) ;27

(defun q3 (a b c) (lambda (x) (quadratic a b c x)))

(defun q2 (a b) (lambda (c) (q3 a b c)))

(defun q1 (a) (lambda (b) (q2 a b)))

(funcall(funcall(funcall(q1 1) 2) 3) 4) ;27

;Partial application