List Lehrgang

[Variablen 1](#_Toc460151915)

[Konstanten 2](#_Toc460151916)

[Operationen 2](#_Toc460151917)

[Arithmetisch 2](#_Toc460151918)

[Logische Operatoren 3](#_Toc460151919)

[Bitoperations 3](#_Toc460151920)

[Entscheidungsbäume 4](#_Toc460151921)

[If else case 4](#_Toc460151922)

[When construct 4](#_Toc460151923)

[Switch 4](#_Toc460151924)

[Schleifen 4](#_Toc460151925)

[Loop for construct 5](#_Toc460151926)

[The do Construct 5](#_Toc460151927)

[The dotimes Construct 5](#_Toc460151928)

[The dolist Construct 5](#_Toc460151929)

[Blocks 5](#_Toc460151930)

[Functions 6](#_Toc460151931)

[Input from Keyboard 6](#_Toc460151932)

[Optional Parameters 6](#_Toc460151933)

[Rest Parameters 6](#_Toc460151934)

[Keyword Parameters 7](#_Toc460151935)

[Returning Values from a Function 7](#_Toc460151936)

[Lambda Funktionen 7](#_Toc460151937)

[Mapping Functions 7](#_Toc460151938)

[Pedicates 7](#_Toc460151939)

[Numbers 9](#_Toc460151940)

[Characters 10](#_Toc460151941)

[Special Characters 10](#_Toc460151942)

[Arrays 10](#_Toc460151943)

[Strings 12](#_Toc460151944)

Variablen

Variablen Definieren erfolgt über (setq name wert)

Drucken ist (print name)

„~%“ = \n

Funktionen werden als macro definiert

(defmacro setTo10(num)

(setq num 10)(print num))

defvar

sind globale variablen

variablen Zugriff erfolgt mit ~

(let ((x 'a)

(y 'b)

(z 'c))

(format t "x = ~a y = ~a z = ~a" x y z))

Mit dieser Syntax kann man arrays in arrays definieren:

(prog ((x '(a b c))

(y '(1 2 3))

(z '(p q 10)))

(format t "x = ~a y = ~a z = ~a" x y z))

terpri = new line

Bei übergabe von Parametern keine Klammern benutzen, hingegeben mit klammern, wenn es eine Funktion mit return Wert ist.

Konstanten

(defconstant PI 3.141592)

(defun area-circle(rad)

(terpri)

(format t "Radius: ~5f" rad)

(format t "~%Area: ~10f" (\* PI rad rad)))

(area-circle 10)

Flächeninhalt eines Kreises

Operationen

Arithmetisch

+ - etc schreibt man vor die eigentliche Rechnung

(incf A 1) = i++

(decf A 1) = i—

(setq a 10)

(setq b 20)

(format t "~% A + B = ~d" (+ a b))

(format t "~% A - B = ~d" (- a b))

(format t "~% A \* B = ~d" (\* a b))

(format t "~% B / A = ~d" (/ b a))

(format t "~% Increment A by 3 = ~d" (incf a 3))

(format t "~% Decrement A by 4 = ~d" (decf a 4))

Ein paar Beispiele

Vergleiche nur Zahlen!

(= A B) -> ==

(/= A B) -> !=

(setq a 10)

(setq b 20)

(format t "~% A == B is ~a" (= a b))

(format t "~% A != B is ~a" (/= a b))

(format t "~% A > B is ~a" (> a b))

(format t "~% A < B is ~a" (< a b))

(format t "~% A >= B is ~a" (>= a b))

(format t "~% A <= B is ~a" (<= a b))

(format t "~% Max of A and B is ~a" (max a b))

(format t "~% Min of A and B is ~a" (min a b))

Logische Operatoren

(setq a 10)

(setq b 20)

(format t "~% A and B is ~a" (and a b))

(format t "~% A or B is ~a" (or a b))

(format t "~% not A is ~a" (not a))

(terpri)

(setq a nil)

(print "a is nil now.")

(setq b 5)

(format t "~% A and B is ~a" (and a b))

(format t "~% A or B is ~a" (or a b))

(format t "~% not A is ~a" (not a))

(terpri)

(print "a is 10 now")

(setq a 10)

(setq b 0)

(setq c 30)

(setq d 40)

(format t "~% Result of and operation on 10, 0, 30, 40 is ~a" (and a b c d))

(format t "~% Result of or operation on 10, 0, 30, 40 is ~a" (or a b c d))

(terpri)

(print "c is nil now")

(setq a 10)

(setq b 20)

(setq c nil)

(setq d 40)

(format t "~% Result of and operation on 10, 20, nil, 40 is ~a" (and a b c d))

(format t "~% Result of or operation on 10, 20, nil, 40 is ~a" (or a b c d))

Bitoperations

And or xor

Xor = false wenn beide gleich sind

(setq a 60)

(setq b 13)

(format t "~% BITWISE AND of a and b is ~a" (logand a b))

(format t "~% BITWISE INCLUSIVE OR of a and b is ~a" (logior a b))

(format t "~% BITWISE EXCLUSIVE OR of a and b is ~a" (logxor a b))

(format t "~% A NOT B is ~a" (lognor a b))

(format t "~% A EQUIVALANCE B is ~a" (logeqv a b))

(terpri)

(terpri)

(setq a 10)

(setq b 0)

(setq c 30)

(setq d 40)

(format t "~% Result of bitwise and operation on 10, 0, 30, 40 is ~a" (logand a b c d))

(format t "~% Result of bitwise or operation on 10, 0, 30, 40 is ~a" (logior a b c d))

(format t "~% Result of bitweise xor operation on 10, 0, 30, 40 is ~a" (logxor a b c d))

(format t "~% Result of bitwise equivalence operation on 10, 0, 30, 40 is ~a" (logeqv a b c d))

Entscheidungsbäume

(setq a 10)

(cond ((> a 20)

(format t " ~% a is less than 20"))

(t (format t "~% value of a is ~d " a)))

The t in the second clause ensures that the last action is performed if none other is fulfilled

* T markiert den default case

If case

(setq a 10)

(if (> a 20)

then (format t "~% a is less than 20"))

(format t "~% value of a is ~d " a)

If else case

(setq a 100)

(if (> a 20)

(format t "~% a is greater than 20")

(format t "~% a is less than 20"))

(format t "~% value of a is ~d " a)

When construct

(setq a 100)

(when (> a 20)

(format t "~% a is greater than 20"))

(format t "~% value of a is ~d " a)

Switch

(setq day 4)

(case day

(1 (format t "~% Monday"))

(2 (format t "~% Tuesday"))

(3 (format t "~% Wednesday"))

(4 (format t "~% Thursday"))

(5 (format t "~% Friday"))

(6 (format t "~% Saturday"))

(7 (format t "~% Sunday")))

Schleifen

Loop = while(true)

Läuft bis ein return statement kommt

(setq a 10)

(loop

(setq a (+ a 1))

(write a)

(terpri)

(when (> a 17) (return a)))

Loop for construct

(loop for x in '(tom dick harry)

do (format t "~s " x)

)

(loop for a from 10 to 20

do (print a)

)

Drucke nur alle durch 2 teilbaren

(loop for x from 1 to 20

if(evenp x)

do (print x)

)

The do Construct

(do ((x 0 (+ 2 x))

(y 20 ( - y 2)))

((= x y)(- x y))

(format t "~% x = ~d y = ~d" x y))

The dotimes Construct

(dotimes (n 11)

(setq temp (\* n n))

(format t "~% ~a ~b" n temp))

The dolist Construct

(setq list '(1 2 3 4 5 6 7 8 9))

(dolist (n list)

(format t "~% Number: ~d Square: ~d" n (\* n n)))

Blocks

(defun demo-function (flag)

(print 'entering-outer-block)

(block outer-block

(print 'entering-inner-block)

(print (block inner-block

(if flag

(return-from outer-block 3)

(return-from inner-block 5))

(print 'This-will-not-be-printed)))

(print 'left-inner-block)

(print 'leaving-outer-block)

t))

(demo-function t)

(terpri)

(demo-function nil)

Functions

Defun definiert eine function

(defun averagenum (n1 n2 n3 n4)

(/ ( + n1 n2 n3 n4) 4))

(write(averagenum 10 20 30 40))

Beispiel 2

(defun area-circle(rad)

(format t "~%Calculates area of a circle with given radius")

(terpri)

(format t "~%Radius : ~5f" rad)

(format t "~%Area: ~10f" (\* 3.141592 rad rad)))

(area-circle 10)

Input from Keyboard

(defun area-circle(rad)

(terpri)

(format t "~%Radius : ~5f" rad)

(format t "~%Area: ~10f" (\* 3.141592 rad rad)))

(format t "~%Calculates area of a circle with given radius~%")

(format t "~%Please give a radius: ")(area-circle (read))

Optional Parameters

(defun show-members (a b &optional c d) (write (list a b c d)))

(show-members 1 2 3)

(terpri)

(show-members 'a 'b 'c 'd)

(terpri)

(show-members 'a 'b)

(terpri)

(show-members 1 2 3 4)

Rest Parameters

(defun show-members (a b &rest values) (write (list a b values)))

(show-members 1 2 3)

(terpri)

(show-members 'a 'b 'c 'd)

(terpri)

(show-members 'a 'b)

(terpri)

(show-members 1 2 3 4)

(terpri)

(show-members 1 2 3 4 5 6 7 8 9)

Keyword Parameters

(defun show-members (&key a b c d) (write (list a b c d)))

(show-members :a 1 :c 2 :d 3)

(terpri)

(show-members :a 'p :b 'q :c 'r :d 's)

(terpri)

(show-members :a 'p :d 'q)

(terpri)

(show-members :a 1 :b 2)

Returning Values from a Function

(defun add-all(a b c d)

(+ a b c d))

(setq sum (add-all 10 20 30 40))

(write sum)

(terpri)

(write (add-all 23.4 56.7 34.9 10.0))

**Sample 2**

(defun myfunc (num)

(return-from myfunc 10)

write num)

(write (myfunc 20))

Lambda Funktionen

(write ((lambda (a b c x)

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

4 2 9 3))

Mapping Functions

(write (mapcar '1+ '(23 34 45 56 67 78 89)))

Sample 2

(defun cubeMylist(lst)

(mapcar #'(lambda(x) (\* x x x x)) lst))

(setq tempList (cubeMylist '(2 3 4 5 6 7 8 9)))

(write (cubeMylist tempList))

Pedicates

Functions that test their arguments fpr some specific conditions and return nil if false else some non-nil value

(write "Check for \"atom\"")

(terpri)

(write (atom 'abcd))

(terpri)

(write "Check equal")

(terpri)

(write (equal 'a 'b))

(terpri)

(write "Check evenp with even")

(terpri)

(write (evenp 10))

(terpri)

(write "Check evenp with uneven")

(terpri)

(write (evenp 7))

(terpri)

(write "Check oddp with uneven")

(terpri)

(write (oddp 7))

(terpri)

(write "Check zerop")

(terpri)

(write (zerop 0.00000001))

(terpri)

(write "Check eq 3 and 3.0")

(terpri)

(write (eq 3 3.0))

(terpri)

(write "Check equal 3 and 3.0")

(terpri)

(write (equal 3 3.0))

(terpri)

(write "Check null nil")

(terpri)

(write (null nil))

Example 2

(defun factorial (num)

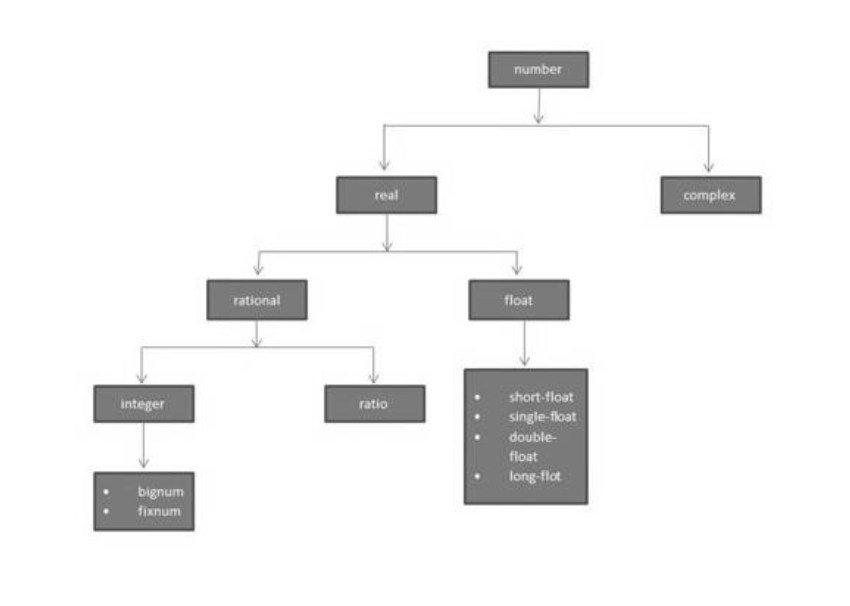
(cond ((zerop num) 1)

(t (\* num (factorial (- num 1))))))

(setq n 6)

(format t "~% Factorial ~d is: ~d" n (factorial n))

Numbers



Bignum = long

Float = same without f = double?

(write (/ 1 2))

(terpri)

(write (+ (/ 1 2)(/ 3 4)))

(terpri)

(write ( + #c( 1 2) #c(3 -4)))

Sample 2 Operation Types

(write (/ 45 78))

(terpri)

(write (floor 45 78))

(terpri)

(write (ceiling 3456 75))

(terpri)

(write (ceiling 3456 75))

(terpri)

(write (truncate 3456 75))

(terpri)

(write (round 3456 75))

(terpri)

(write (ffloor 3456 75))

(terpri)

(write (fceiling 3456 75))

(terpri)

(write (ftruncate 3456 75))

(terpri)

(write (mod 3456 75))

(terpri)

(setq c (complex 6 7))

(write c)

(terpri)

(write (realpart c))

(terpri)

(write (imagpart c))

Characters

(write 'a)

(terpri)

(write #\a)

(terpri)

(write-char #\a)

(terpri)

(write-char 'a)

Special Characters

(write "case-sensitive comparison")

(terpri)

(write (char= #\a #\b))

(terpri)

(write (char= #\a #\a))

(terpri)

(write (char= #\a #\A))

(terpri)

(write "case-insensitive comparision")

(terpri)

(write (char-equal #\a #\A))

(terpri)

(write (char-equal #\a #\b))

(terpri)

(write (char-lessp #\a #\b #\c))

(terpri)

(write (char-greaterp #\a #\b #\c))

Arrays

(write (setf my-array (make-array '(10))))

(terpri)

(setf (aref my-array 0) 25)

(setf (aref my-array 1) 23)

(setf (aref my-array 2) 45)

(setf (aref my-array 3) 10)

(setf (aref my-array 4) 20)

(setf (aref my-array 5) 17)

(setf (aref my-array 6) 25)

(setf (aref my-array 7) 19)

(setf (aref my-array 8) 67)

(setf (aref my-array 9) 30)

(write my-array)

Sample2

(setf x (make-array '(3 3)

:initial-contents '((0 1 2) (3 4 5) (6 7 8))))

(write x)

Sample 3

(setq a (make-array '(4 3)))

(dotimes (i 4)

(dotimes (j 3)

(setf (aref a i j) (list i 'x j '= (\* i j)))))

(dotimes (i 4)

(dotimes (j 3)

(print (aref a i j))))

Sample 4

(setq myarray (make-array '(3 2 3)

:initial-contents

'(((a b c) (1 2 3))

((d e f) (4 5 6))

((g h i) (7 8 9))

)))

+1 entweder

(setq array2 (make-array 4 :displaced-to myarray

:displaced-index-offset 2))

(write myarray)

(terpri)

(write array2)

+2 oder

(setq array2 (make-array '(3 2) :displaced-to myarray

:displaced-index-offset 2))

(write myarray)

(terpri)

(write array2)

Sample5

; a one dimensional array with 5 elements,

;initial value 5

(write (make-array 5 :initial-element 5))

(terpri)

;two dimensional arrray, with initial element a

(write (make-array '(2 3) :initial-element 'a))

(terpri)

;an array of capacity 14, but fill pointer 5, is 5

(write(length (make-array 14 :fill-pointer 5)))

(terpri)

;however its length is 14

(write(make-array 10 :element-type 'character :initial-element #\a))

(terpri)

; a two dimensional array with initial value a

(setq myarray (make-array '(2 2) :initial-element 'a :adjustable t))

(write myarray)

(terpri)

;readjusting the array

(adjust-array myarray '(1 3) :initial-element 'b)

(write myarray)

Strings