Defining functions. Predefined functions and predicates

The purpose of this work is to explain the creation of new functions and the conditional evaluations.

In summary: the terms of program, procedure and primitive have a little bit different meanings. A program is a collection of procedures, a procedure specifies the way in which a calculation is made and a primitive is a procedure predefined by the language.

DEFUN allows the building of new procedures

The syntax:

(DEFUN

(<parameter 1> <parameter 2> …<parameter n>)

<the body of the procedure>)

DEFUN doesn't evaluate its arguments, it only establishes a procedure which can be later used. The name of the procedure needs to be a symbol. The list which follows after the name of the procedure is called a list of parameters. When a procedure will be called together with its arguments into a symbolic expression, the value of each parameter in the procedure will be determined by the value of the corresponding argument from the expression.

DEFUN

The transformation of temperature from Fahrenheit in degrees Celsius:

```
(defun f-to-c (temp)
    (/ (- temp 32) 1.8))
F-TO-C
```

DEFUN will return the name of the function which was defined, but the useful part is realised through side effects. Through calling, the value of the argument becomes the temporary value of the parameter of the procedure.

The preparation process of the memory space for the values of the parameters is called **BINDING.** The parameters are bound through the calling of procedures. The process of determining a value is called **ATRIBUTING**. LISP always assigns values immediately after binding parameters.

Example of binding

The following function returns the reversed list of a list formed of 2 elements:

```
defun exchange (pair)
    (list (cadr pair) (car pair))) ; Reverses the elements

EXCHANGE
```

Observation: Comments in LISP are made with ";" .From ";" until the end of the line is considered a comment.

Problem 1

Some are annoyed of the critical primitives CAR, CDR and CONS. Define new procedures *my-first, my-rest* and *construct* which do the same thing. You can also define *my-second, my-third,* etc.

Problem 2

Define *rotate-left* which gets a list as its argument and returns a new list in which the first element of the original list is now the last element of the returned list.

(rotate-left '(a b c))
(b c a)
(rotate-left (rotate-left '(a b c)))
(c a b)

Problem 3

Define *rotate-right*.

Problem 4

A palindrome is a list which contains the same sequence of elements which can be read both from left to right and from right to left. Define **palindrome** which gets a list as an argument and returns a palindrome twice its length.

Problem 5

Define ec2 which gets 3 parameters a, b and c and returns a list which contains the 2 roots of the quadratic a*x^2+b*x+c=0, using the formula:

$$x = \frac{-b \pm \sqrt{b*b-4*a*c}}{2*a}$$

Predicates

The **predicates** are procedures that return a truth value. More complicated definitions require the use of procedures called predicates. A predicate is a procedure which returns a value that

indicates true or false. False is always indicated with *nil.* Truth is always indicated with a special symbol *t*, but essentially anything different from *nil* means true.

Observation: *t* and *nil* are special symbols, their values being bound also at *t* and nil. Thus the value of *t* is *t* and the value of *nil* is *nil.*

ATOM is a predicate which tests if the argument is an atom.

LISTP tests if the argument is a list.

ATOM and LISTP

If we have:

(setq digits '(zero one two three four five six seven eight nine))

(ZERO ONE TWO THREE FOUR FIVE SIX SEVEN EIGHT NINE)

ATOM and *LISTP* work as follows:

(atom 'digits)

T

(atom ' five)

T

(atom digits)

Nil

(atom '(zero one two three four five six seven eight nine))

Nil

(listp 'digits)

Nil

(listp 'five)

Nil

(listp digits)

T

(listp '(zero one two three four five six seven eight nine))

T

EQUAL takes 2 arguments and returns *t* if they are equal, else it returns *nil:*

EQUAL

(equal digits digits)

T

(equal 'digits 'digits)

T

(equal digits '(zero one two three four five six seven eight nine))

T

(equal digits 'digits)

Nil

The predicate = verifies if two numbers are equal.

=

(= 3.14 2.71)

Nil

(= (* 5 5) (+ (* 3 3) (* 4 4)))

T

We now need to highlight a very important characteristic:

- *nil and the void list () are completely equivalent: nil == (), they satisfy the predicate EQUAL*
- *though convention the void list is printed as nil.*
- *there can appear programming errors because (atom '()) is t. This is because nil is considered an atom.*

The void list is the only expression that is both a list and an atom. Both (*atom nil*) and (*listp nil*) return *t*.

Nil and ()

(equal nil '())

T

The predicate NULL checks if the argument is a void list:

NULL

(null '())

t

(null t)

nil

(null digits)

nil

(null 'digits)

Nil

The predicate *MEMBER* tests if an atom is the element of a list. MEMBER checks only the membership on the first level of list.

(member <atom> <list>)

It would be natural if MEMBER would return t is the element belongs to the list. Actually MEMBER returns the fragment of the list which starts with the found atom. The idea is to return something different from nil that could be further used.

MEMBER

(member 'five digits)

(five six seven eight nine)

(member 'ten digits)

Nil

The first argument needs to be an element in the first level of the second argument, not “buried” somewhere in the list:

(member ' five '((zero two four six eight) (one three five seven)))

Nil

In the next example it is exploited that MEMBER returns something useful :

(length (cdr (member 'five digits)))

4

The predicate NUMBERP tests if the argument is a number.

NUMBERP

For the next examples we will establish :

(zero 0 one 1 two 2 three 3 four 4 five 5 six 6 seven 7 eight 8 nine 9)

That way:

(numberp 3.14)

t

(numberp five)

t

(numberp 'five)

nil

(numberp digits)

nil

(numberp 'digits)

nil

The predicates < and > expects that the arguments are numbers and tests if they are ordered strictly ascending or descending.

Comparison predicates

(> five 2)

t

(> 2 five)

nil

(< 2 five)

t

(< 2 2)

nil

(> five four three two one)

t

(> three one four)

nil

(> 3 1 4)

nil

Also defined are the operators <= and >= to which the strict monotony isn’t required.

The predicate ZEROP tests if the argument is the number zero.

ZEROP

(zerop zero)

t

(zerop 'zero)

error

(zerop five)

Nil

The predicate MINUSP tests if a number is negative:

(minusp one)

nil

(minusp (- one))

t

(minusp zero)

Nil

The predicate EVENP tests if a number is even.

EVENP

(evenp (* 9 7 5 3))

Nil

You can notice that the majority of predicates use **P** at the end of the name as mnemonics for the predicate. Except for ATOM.

Problem 6

Define your own version of the predicate EVENP. You can use the primitive REM which returns the remainder of the division of two whole numbers.

Problem 7

Define the predicate PALINDROMP which tests if the argument is a palindrome.

Problem 8

Define the predicate NOT-REALP which takes three parameters and returns t if b*b-4*a*c < 0.

Logical predicates

NOT returns t only if the argument is nil. Otherwise it returns nil.

NOT

(not nil)

t

(not t)

nil

(setq pets '(dog cat))

(not (member 'dog pets))

nil

(not (member tiger pets))

T

AND returns different from nil only if all arguments are different from nil. OR returns different from nil if at least one of the arguments is different from nil. Both take any number of arguments.

AND and OR

(and t t nil)

nil

(or t t nil)

t

(and (member 'dog pets) (member tiger pets))

nil

(or (member 'dingo pets) (member tiger pets))

nil

The arguments for AND and OR aren’t treated the standard way:

- AND evaluates its arguments from left to right. If it meets nil it stops, the remaining arguments won't be evaluated!
  Thus AND returns the value of the last argument.
- OR evaluates its arguments from left to right. If it meets something different from nil it stops, the remaining arguments won't be evaluated!
  Thus OR returns nil.

Both predicates return the last calculated value in case of success.

AND and OR

(and (member 'dog pets) (member 'cat pets))

(cat)

(or (member 'dog pets) (member tiger pets))

(dog cat)

Selecting alternatives. The primitive COND

The syntax of COND is:

(COND (<test 1> <result 1>)

(<test 2> <result 2>)

...

(<test n> <result n>))

The symbol COND is followed by random number of lists each containing a test and an expression to evaluate. Each list is called a clause. The idea is to go through clauses evaluating the test of each one, until the value of a test is different from nil, in that case we evaluate also the result of the clause. There exist two special cases:

- If the test of neither clause is different from nil, COND returns nil.
- If the clause contains the test it returns the value of the test.

COND

```
(cond ((null L) 'EMPTY)
      (t 'NOT-EMPTY))
```

We can also write:

```
(cond (L 'NOT-EMPTY)
      (t 'EMPTY))
```

Notice that NULL and NOT are essentially equivalent primitives:

NOT X) == (NULL X)

Problem 9

Express (abs X), (min A B) and (max A B) with the help of COND.

Problem 10

Express (not U), (or X Y Z) and (and A B C) with the help of COND.

Problem 11

Which is the value of the following form? Argument your answear.

```
(cons (car nil) (cdr nil))
```

COND and DEFUN

We want to define a procedure which gets two arguments: an atom and a list, and returns a list having the atom put at the beginning of

the list only if it isn't already in the list. Otherwise it returns the list unmodified.

```
(defun adjoin (item bag)
    (cond ((member item bag) bag) ; is it already in the list?
          (t (cons item bag)))) ; add at the beginning of the list.
```

Adjoin

Observation: t will appear several times as a test for the last clause of a COND to define what needs to be done if no clause was evaluate.

Problem 12

Define MEDIAN-OF-THREE, a procedure which gets three numerical arguments and returns the one closest to the middle as a value(meaning the one that isn't neither the smallest or the largest).

Note: in the writing of the function, a correct indentation of the code will be a big help.

Problems

Problem 1. Defun. My-first, my-rest.

Problem 2. Defun. Rotate-left .

Problem 3. Defun. Rotate-right.

Problem 4. Defun. Palindrom.

Problem 5. Defun. Quadratic ecuation.

Problem 6. Predicate. Evenp.

Problem 7. Predicate. Palindrom.

Problem 8. Predicate. Quadratic ecuation.

Problem 9. Cond. General functions.

Problem 10. Cond. Logical functions.

Problem 11. The evaluation of nil.

Problem 12. Median of three elements.