Cflp

Laborator

Facultatea de Automatica şi Calculatoare
Departamentul de Calculatoare

Lucrarea 8

Optional parameters

When defining a function you may specify that one or more parameters are optional

OPTIONAL

```
(defun power (x &optional y)
               (cond ((not y) x) ; y == 1
                     (t (expt x y))))
```

The function from above takes one single argument, the second argument is optional and it's default value is is `1` if the caller doesn't provide any value for it.

OPTIONAL

```
(defun power (x &optional (y 1))
               (expt x y))
```

There can be any number of `&optional` parameters but only a single `&rest` parameter. The `&rest` parameter can be used when you need a variable number of arguments. When you use `&rest` you may specify as many arguments as you like, all the arguments that you specify (besides the ones that are required by the function) are translated into a list.

REST

```
(defun my-or (&rest r)
   (cond ((null r) nil)
         ((car r))
         (t (apply 'my-or (cdr r)))))
```

Macros

Macros are like functions but unlike functions macros don't evaluate their arguments.

DEFMACRO

Carefully follow this example:

```
(defmacro demo (par)
     (print par))
(setq this 'value-of-this)
(demo this)
this
value-of-this
```

When the macro is called, `this` isn't evaluated, this means that `this` becomes the value of `par` and "this"" is displayed on the screen. At the return, this is evaluated.

Problems

OUR-IF

`Our-if` macro implementation.

```
(defmacro our-if (test success failure)
     (subst test 'test
     (subst success 'success
     (subst failure 'failure
        '(cond (test success) (t failure))))))
```

The `failure` paramater may be declared as optional.

Backquote

The backquote mechanism permits creation the of mostly constant expressions. The backquote behaves like the single quote but the marker "," inside the backquote indicates that the value isn't constant.

Backquote

```
(setq variable 'example)
example
`(this is an ,variable)
(this is an example)
```

OUR-IF 2

Using the backquote mechanism `our-if` becomes:

```
(defmacro our-if (test success &optional failure)
    `(cond (,test ,success) (t ,failure)))
```

Problem 1

Define a macro that defines a function using the following syntax:

```
(define (<nume functie> <param 1> ... <param n>)
            <corp>)
```

Problem 2

Define a macro `dotimes` using the following syntax:

```
(dotimes (<var> <count> <result>) <body>)
```

`<count>` should be evaluated the first time and it should be an integer. Then `<var>` is binded successively to integers from `0` to `<count>` minus `1`. The body should be evaluated every time and `<result>` should be returned at the end.
Remark: search for a variant using `do` which is equivalent the write the macro `dotimes`.

CONS

`CONS` builds new lists using free cells
LISP keeps a list of free cells from which they get consumed on demand. (After there aren't any free cells left, the *garbage collection* algoritm is executed). `CONS` takes the fist available cell from the free cell list and it modifies the pointer which binds it to the next cell.

CONS

Assume we have:

```
(setq example '(b c))
```

It's evaluation

```
(cons 'a example)
```

will extract a cell from the free cell list a cell whose contents will be `a`. The extracted free cell's pointer that points to the next cell will be altered to point to the address of `b` from the list of `(b c)`. The address which contains the value of `a` will be returned (a new constructed list).

APPEND

`APPEND` builds a new list via copy

APPEND

Let's assume we're evaluating the following expressions:

```
(setq abc '(a b c))
(setq xyz '(x y z))
```

We have now two lists: `(a b c)` and `(x y z)` whose begining addresses are memorized by `abc` and `xyz`.
What will happen when the expression `(setq abcxyz (append abc xyz))` is evaluated ?
What we might think will happen but it **doesn't happen** might be:
The list `(a b c)` is tied to the list `(x y z)` by the pointer of c (currently pointing to `nil`) which is modified to point at the begining of `(x y z)` but this **doesn't happen!**
What actually happens: Append creates a new list and copies the contents of the list `(a b c)` to it, using available free cells. The last pointer of the newly created list is then modified to point to the begining of the `(x y z)` list, which returns the begining of the new list. Such that, the first three cells of the new list `(a b c x y z)` are actually **copies** of the cells from the list `(a b c)` while the last three cells of the new list are cells from the `(x y z)` list.

Functions with destructive effect

NCONC, RPLACA, RPLACD şi DELETE

These predicates alter the in-memory content of the list and need to be used carefully, because their actions are destructive!
`NCONC` does what we thought `APPEND` does. It concatenates two lists modifying the last pointer, not via copy.

NCONC

```
(setq abcxyz (nconc abc xyz))
(a b c x y z)
abc
(a b c x y z)
xyz
(x y z)
```

`NCONC`, just like `APPEND`, can take many arguments, it modifies the end pointer of every list, except for the last.
`RPLACA` takes two arguments from which the first must be a list. It alters the list, replacing the content of the first cell with the content of the second argument. The name comes from RePLAce CAr.
`RPLACD` is complementary, it modifies the rest of the list (the pointer which follows after the first).

DELETE removes the value of first argument from the first level of the second argument, but it doesn't alter the initial list.

DELETE

```
(delete 'a '(a b b a b))
(b b b)
```

The difference between EQUAL and EQL

For EQL two lists are the same only if they are represented by the same in-memory cells. Copies are not considered to be the same because their address is different, however, copies are considered to be the same by EQUAL.

Problem 3

REVERSE reverses the list via copy. Write a function that takes REV and reverses physically the cells from a list. The function must return a list formed by the same cells, creating a new list or returning a new list is not allowed!

```
(setq l ' (a b c d))
((a b c d)
(rev l)
(d c b a)
l
(d c b a)
```

Probleme

Problem 1. Define
Problem 2. Dotimes
Problem 3. Destructive reverse