Cflp
Laborator

Facultatea de Automatica şi Calculatoare
Departamentul de Calculatoare

Home | Lucrarea 1 | Lucrarea 2 | Lucrarea 3 | Lucrarea 4 | Lucrarea 5 | Lucrarea 6 | Lucrarea 7 | Lucrarea 8 | Lucrarea 9 | Lucrarea 10 | Lucrarea 11 | Lucrarea 12 | Lucrarea 13 | Proiect | Orar

Lab 6

I/O Functions

PRINT and READ

Contents

PRINT gets a single argument, which is evaluated and printed on a new line.

PRINT

The example bellow prints the squared values of all integers, until we stop it.

```
(defun bore-me ()
   (do ((n 0 (+ n 1)))
       (nil)
       (print (* n n))))
BORE-ME
(bore-me)
0
1
4
9
...
```

PRINT returns the value of its argument.
When it finds (READ), the lisp interpreter stops and waits for the user to write an expression. The expression, without being evaluated, becomes the value of (READ). Since READ doesn't print anything to show that it is waiting for an expression, it is considered good practice to use a PRINT before, in order to signal that the program is waiting for user input.

Conventions used for special symbols

Sometimes it is useful to have symbols which contain special characters (like spaces, parenthesis, ...).
This can be achieved by surrounding the symbol with two vertical lines.

PRINT

```
(setq simbol1 '|(| )
|(|
(setq simbol2 '|a simbol| )
|a simbol|
(print simbol1)
|(|
|(|
(print simbol2)
```

```
|a simbol|
|a simbol|
```

If we need a currently unused symbol, we can generate one by using `GENSYM`.

GENSYM

```
(GENSYM)
G1
(GENSYM)
G2
```

TERPRI, PRIN1 si PRINC

`TERPRI` prints a line feed.
`PRIN1` is similar to `PRINT`, but doesn't append a line feed nor a space at the end.
`PRINC` is similat to `PRIN1`, but doesn't print vertical lines (if they exist).

PRIN1 and PRINC

```
(print simbol2)
|a simbol|
|a simbol|
(prin1 simbol2)|a simbol|
|a simbol|
(princ simbol2)a simbol
|a simbol|
```

Problem 1

Write the `PRETTY-PRINT` procedure, which takes one argument (a generalized list), and prints it using the following rule:

```
( <element-1>
  <element-2>
  <element-3>
  ...
  <element-n> )
```

Any element, which is a list, is going to be printed using the same algorithm, recursively.
Example:

```
(pretty-print ' ( a (b c de) fg ))
( a
  ( b
    c
    de )
  fg )
```

File I/O

In order to use files for I/O, we need to open and close them, like in other programming languages.

OPEN

Open a file for reading or writing.
Syntax:

```
    (open <string file name>
            [:direction [:input | :output | :io | :probe]]
            [:element-type [:string-char | :unsigned-byte |
:signed-byte | ...]
            [:if-does-not-exist [:error | :create | nil |
...]])
```

This function returns a file stream, which can be used for I/O operations.

Parameters

`:direction` specify if the file is being opened for reading (`:input`, default), for writing (`:output`), for both reading and writing (`:io`), or only to test its existence (`:probe`)
`:element-type` Specify what type of data is in the file. This has effect over which I/O operations are later permitted on the file. The default value is `:string-char`
`:if-does-not-exist` Actions to take in case the file does not exist. Can be used to indicate the signaling of the error (`:error`), the automatic creation of the file (`:create`), or the `nil` value being returned, without opening the file or signaling the error.
For additional details you can check out a LISP manual.

CLOSE

Close an open file. Returns `t` on success, or `nil` in case of failure.
Syntax:

```
    (close <file stream>)
```

Open - read - write

In the example bellow a file is opened, the first expression is read, and then the file is closed.

```
(let ((file         ;; this variable will contain the file
stream
     (open "file.txt" :direction :input))) ;; open the file
for reading
     (print (read file))                   ;; read and print
     (close file)                          ;; free resources
)
```

READ-CHAR

Syntax:

```
    (read-char [<file stream>
        [<error signal in case of EOF (end of file)>
              [<Value in case of EOF>] ] ])
```

Read a single character from the file stream. In the case when `<error singnal in case of EOF>` is `nil`, trying to read past the end of file will result in the function returning `<value in case of EOF>`.

WRITE-CHAR

Syntax:

```
    (write-char <character> [<file stream> ])
```

Write the character in the specified file stream.

READ-LINE

Read a line from the stream and return the value. The syntax is similar to `READ-CHAR`.

WRITE-LINE

Write the line in the stream. The syntax is similar to `WRITE-CHAR`.

READ-BYTE

Read an integer from the stream and return its value. The syntax is similar to `READ-CHAR`. The file has to be opened accordingly.

WRITE-BYTE

Write a given integer in the file stream. The syntax is similar to `WRITE-CHAR`. The file must be opened accordingly.

The functions presented in the beginning of this lab can also work with file streams, by adding an open file stream in the arguments list. Nonetheless, not specifying the stream parameter will result in reading/writing using the standard input/output.

Problem 2

Write a lisp function, which copies the content of a file into another file. The function will take as argument the name of the source file and the name of the destination file.

More information about file I/O in lisp can be found here.

Problems

Problem 1. Pretty-print.
Problem 2. Copy file.