FUNKCIONÁLNÍ A LOGICKÉ PROGRAMOVÁNÍ

6. LISP: VSTUP A VÝSTUP, MAKRA

2011 Jan Janoušek
MI-FLP

Evropský sociální fond
Praha & EU:
Investujeme do vaší budoucnosti

INPUT AND OUTPUT

Introduction to input/output system

- A **stream** abstraction for reading and writing data (, which is similar to other programming languages).
- **Pathnames** - an abstraction for manipulating filenames.
- The ability to read and write s-expressions – unique to Lisp.

Input – input streams

- Opening a stream from a file: function `OPEN` returns a character-based input stream.
- `READ-CHAR` reads a single character.
- `READ-LINE` reads a line of text, returning it as a string with the end-of-line character(s) removed.
- `READ` reads a single s-expression, returning a Lisp object.
- the `CLOSE` function closes the stream.

Input

- Opening a stream from a file: function `OPEN` returns a character-based input stream. Standard I/O streams are denoted by T.
- `READ-CHAR` reads a single character.

  Syntax: **`(read-char [stream])`**
- `READ-LINE` reads a line of text, returning it as a string with the end-of-line character(s) removed.

  Syntax : **`(read-char [stream])`**
- `READ` reads a single s-expression, returning a Lisp object – unique to Lisp. Syntax: **`(read [stream])`**
- the `CLOSE` function closes the stream.

Examples

- `(open "/some/some2/name.txt")`

- ```
  (let ((in (open "/some/file/name.txt")))
    (format t "~a~%" (read-line in))
    (close in))
  ```

- ```
  (let ((in (open "/some/file/name.txt" :if-does-not-exist nil)))
      (when in      ; other options above: :create (default) :
          (format t "~a~%" (read-line in))
          (close in)))
  ```

Example - read

- Suppose `/some/file/name.txt` has the following contents (with 4 s-expressions):

```
(1 2 3)
456
"a string" ; this is a comment
((a b) (c d))
```

```
CL-USER> (defparameter *s* (open "/some/file/name.txt"))
*S*
CL-USER> (read *s*)
(1 2 3)
CL-USER> (read *s*)
456
CL-USER> (read *s*)
"a string"
CL-USER> (read *s*)
((A B) (C D))
CL-USER> (close *s*)
T
```

Output – output streams

- Output stream is obtained by calling `OPEN` with a `:direction` keyword argument of `:output`.

```
(open "/some/file/name.txt" :direction :output
:if-exists :supersede)

; options: :if-exists :supersede means replace the
file,  :append
```

Output

- `WRITE-CHAR` writes a single character to the stream.
- `WRITE-LINE` writes a string followed by a newline.
- `WRITE-STRING` writes a string without adding any end-of-line.
- `PRINT` prints an s-expression preceded by an end-of-line.
- `PPRINT` prints s-expressions like PRINT and PRIN1 but using the "pretty printer".
- `PRINC` also prints Lisp objects, but in a way designed for human consumption.

Output – format function

- Incredibly flexible, defines a mini-language for emitting formatted output!
- Syntax: `(format stream format-str val1 val2 ...)`

Examples:

```
CL-USER> (format t "~5$" pi)
3.14159
NIL

CL-USER>(format nil "The value is: ~a" (list 1 2 3))
 "The value is: (1 2 3)"
```

Some format function directives

- Directives without arguments:

`~nT`	skok na sloupec n
`~n%`	n-krát new-line
`~n&`	fresh line a (n-1) new-line
`~n*`	ignoruje dalších n argumentů
`~n:*`	vrací se zpět o n argumentů

```
>(format T "Navštívená města: ~5t~A v roce ~D a ~%~5t~A
v roce ~D.~%" "Londýn" 1999 "New York" 2001)

Navštívená města:
     Londýn v roce 1999 a
     New York v roce 2001.
NIL
```

MACROS

Introduction to macro system of Lisp

- Powerful feature, tight in the Lisp language (unlike macros in C implemented by commands of the C text preprocessor).
- Yet a feature that continues to set Common Lisp apart other languages.
- Macro is written in Lisp. When macro is compiled, it is expanded and it generates a Lisp code. After expanding the generated Lisp code is compiled and possibly evaluated.
- A fairly tale (fiction) of a junior programmer Mac could be said here to explain a nature of macros in Lisp

Macro Expansion Time vs. Runtime

- Only after all the macros have been fully expanded and the resulting code compiled the resulting program can actually be run.
- The time when macros run is called *macro expansion time*; this is distinct from *runtime*, when regular code, including the code generated by macros, runs.
- Running at macro expansion time runs in a very different environment than code running at runtime: during macro expansion time, there's no way to access the data that will exist at runtime.

Example

- Many existing implicit operators in Lisp are in fact macros:
- `when`: `if` and `progn` are joined together:

```
(defmacro when (condition &rest body)
   `(if ,condition (progn ,@body)))
```

- Syntax of backquote (`):
  - synonym for `list`
  - `,var` : `,var` is replaced by the actual value of `var`
  - `,@body` : `body` is assumed to be a list. `@body` is replaced by the elements of `body`.

Example, contd.

```
(if (spam-p current-message)
    (progn (file-in-spam-folder current-message)
           (update-spam-database current-message)))
```

```
(when (spam-p current-message)
      (file-in-spam-folder current-message)
      (update-spam-database current-message))
```

- Macro `when` is expanded to the code above.
- Note. This macro cannot be done simply by a function!

Examples of other standard operators which are macros in fact

- `unless` is defined:

```
(defmacro unless (condition &rest body)
    `(if (not ,condition) (progn ,@body)))
```

- `cond`:

```
(cond (test-1 form*)
            . . .
      (test-N form*))
```

- `setf`: expanded to many variants such as `setq`, …

Standard looping macros

- `(dolist (`*`var list-form`*`) `*`body-form`*`*)`

  - `CL-USER> (dolist (x '(1 2 3)) (print x))`

```
1
2
3
NIL
```

- dotimes
- do
- loop

Defining your own macros

- Syntax of operator `defmacro` (stands for DEFiniton MACRO, not for DEFinition for MAC Read Only):

- `(defmacro` *`name`* `(`*`parameter`*`*)`
  `"Optional documentation string."`
  `body-form``*)`

- Writing macro:
  - Write a sample call to the macro and the code it should expand into, or vice versa.
  - Write code that generates the handwritten expansion from the arguments in the sample call.

An example of writing a macro do-primes

- Basic functions (here, brute-force approach):

```
(defun primep (number)
    (when (> number 1)
    (loop for fac from 2 to (isqrt number)
    never (zerop (mod number fac)))))

(defun next-prime (number)
    (loop for n from number
          when (primep n) return n))
```

- The desired behaviour of our macro:

```
(do-primes (p 0 19) (format t "~d " p))
```

An example of writing a macro do-primes, contd.

```
(defmacro do-primes ((var start end) &rest body)
`(do ((,var (next-prime ,start)
 (next-prime (1+ ,var)))) ((> ,var ,end)) ,@body))
```

Debugging macros

- `macroexpand` operator
- `macroexpand-n` – returns the result after the n-th run of expanding macro

```
CL-USER> (macroexpand-1 '(do-primes (p 0 19)
(format t "~d " p)))
(DO ((P (NEXT-PRIME 0) (NEXT-PRIME (1+ P))))
    ((> P 19))
    (FORMAT T "~d " P))
T
```

Note. `DO` is also macro and therefore will be expanded as well.

Another example

```
CL-USER 53 >
(defmacro from-to (from to &rest body)
  `(do ((x ,from (+ x 1)))
       ((>= x ,to))
     ,@body))
FROM-TO

CL-USER 54 >
(from-to 1 6 (princ "."))
.....
NIL
```

- Note: `x` must be an independent variable!

```
((let ((x 30)))
    (from-to 1 6 (princ ".")))
NIL
```

gensym operator

- generates a unique name of an anonymous variable

```
CL-USER 58 >
(defmacro from-to (from to &rest body)
  (let ((x (gensym)))
    `(do ((,x ,from (+ ,x 1)))
         ((>= ,x ,to))
       ,@body)))
FROM-TO

CL-USER 59 >
(let ((x 6))
  (from-to 1 x (princ ".")))
.....
NIL
```

Another using lisp macros - example

- **Example:** standard stack operations `Push, Pop`

```
> (defvar SS nil)
SS
> (Push SS 'A)
(A)
> (Push SS 'B)
(B A)
> (Pop SS)
B
> SS
(A)
```

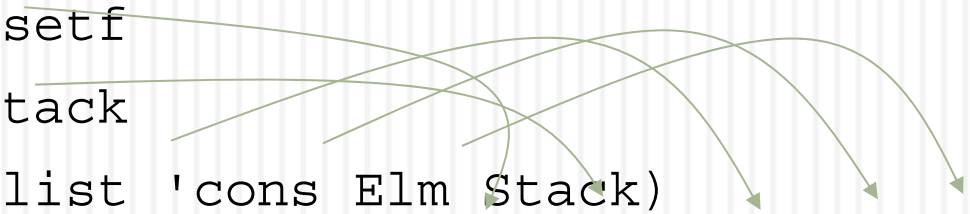
- Let's try it this way 'round ...

```
> (defun S-Push (Elm Stack) (setf Stack (cons Elm Stack)))
> (defun S-Pop (Stack)
    (prog1 (car Stack) (setf Stack (cdr Stack))))
> (defvar SS nil)
SS
> (S-Push 'A SS)
(A)
> (S-Push 'B SS)
(B)             ; ??? where is A
> SS
NIL             ; ??? B disappeared as well!!
```

? Why is it not working as expected/wanted ?

We should have used macros!

```
(defmacro Init (Stack) (list 'setf Stack nil))
                                    (setf SS nil)
(defmacro Push (Elm Stack)
  (list 'setf
        Stack
        (list 'cons Elm Stack)
) )                                 (setf SS (cons E SS))
(defmacro Pop (Stack)
  (list 'prog1
        (list 'car Stack)
        (list 'setf Stack (list 'cdr Stack))
) )                                 (prog1 (car SS)
                                    (setf SS (cdr SS)))
```



When a macro is to be used?

- Generally, when both a function and macro can be used, using function is preferred (there is no reason for using macro).

- However, there are many reasons for using macros:

  - Control of evaluating parameters : in function parameters are evaluated before performing the function and only once … and many other reasons…see next lecture…