Wejście-wyjście w Lispie

Marcin Milewski

3 XI 2010

Jak to wygląda w Lispie?

- Podstawowe operacje jak w innych językach
- Wysoki poziom abstrakcji na systemem plików
- Zapis/odczyt s-wyrażeń

Podstawowe funkcje

- OPEN – zwraca strumień znakowy.
- READ-CHAR – czyta pojedynczy znak.
- READ-LINE – czyta linię, zwraca wczytany ciąg znaków, usuwa znaki końca linii.
- READ – czyta s-wyrażenie, zwraca obiekt w Lispie
- CLOSE – zamknięcie pliku

```
;; Otwieranie
(open "/home/mmi/some.txt")

;; Czytanie
(let ((in (open "/home/mmi/some.txt")))
  (format t "~a~%" (read-line in))
  (close in))
```

Co z błędami?

- Jeżeli coś pójdzie nie tak, dostaniemy błąd.
- Do OPEN możemy przekazać :if-does-not-exist
- - :error
  - :create
  - nil
- READy także przyjmują (opcjonalny) argument – czy zgłosić błąd (domyślnie true)

```
;; Otwórz plik lub zwróć nil
(let ((in (open "/home/mmi/some.txt"
                :if-does-not-exist nil)))
  (when in
    (format t "~a~%" (read-line in))
    (close in)))


;; Odczyt wszystkich linii
(let ((in (open "/home/mmi/some.txt"
                :if-does-not-exist nil)))
  (when in
    (loop for line = (read-line in nil)
          while line do (format t "~a~%" line))
    (close in)))
```

READ

- Każde wywołanie READ wczytuje s-wyrażenie
- Białe znaki i komentarze są pomijane
- Można wypisać strukturę tak, żeby dało się ją wczytać READem

```
;; Wczytanie s-wyrażeń
(defparameter s (open "/home/mmi/lisp.txt"))
(read s)
(close s)
```

Odczyt binarny

- OPEN domyślnie otwiera pliki tekstowe
- Pliki binarne – przekazać :element-type '(unsigned-byte 8)
- READ-BYTE zwraz liczbę całkowitą 0-255
- Również posiada opcjonalny argument do sygnalizacji błędu

READ-SEQUENCE

- Działa ze znakami i strumieniami binarnymi
- Można przekazać jej sekwencję oraz strumień – wypełni sekwencję strumieniem
- Zwraca indeks pierwszego niewypełnionego elementu
- Można wypełnić kawałek sekwencji – :start, :end
- Bardziej efektywna niż READ-BYTE, READ-CHAR

```
;; Wypełnianie sekwencji
(defparameter v (make-array 4))
(read-sequence v (make-string-input-stream "abcdefghi"))
```

- Kolejny argument dla OPEN – :direction :output
- OPEN zakłada że plik nie istnieje

:if-exists

- :supersede – podmień istniejący plik
- :append – dopisywanie na końcu
- :overwrite – nadpisywanie zawartości pliku
- nil – zwróci nil jeżeli plik istnieje

Funkcje do zapisu

- WRITE-CHAR
- WRITE-LINE – znak nowej linii zależny od platformy
- WRITE-STRING
- TERPRI – terminate print. Nowa linia
- FRESH-LINE – nowa linia
- WRITE-BYTE – otwieranie pliku binarnego jak przy READ-BYTE

Funkcje do zapisu

- WRITE-CHAR
- WRITE-LINE – znak nowej linii zależny od platformy
- WRITE-STRING
- TERPRI – terminate print. Nowa linia
- FRESH-LINE – nowa linia, ale tylko jeśli jej nie ma
- WRITE-BYTE – otwieranie pliku binarnego jak przy READ-BYTE

Jeszcze więcej wypisywania

- PRINT – drukuje s-wyrażenie + nowa linia + spacja
- PRIN1 – drukuje s-wyrażenie
- PPRINT – jw, ale bardziej estetycznie. Można użyć do generowanie JSONa
- PRINC – more human readable. Np. hello zamiast “hello”

Zamykanie plików

Możliwe kłopoty:

- Brak CLOSE
- Sterowanie może nie dotrzeć do CLOSE

Rozwiązania

- UNWIND-PROTECT
- WITH-OPEN-FILE

Rozwiązania

- UNWIND-PROTECT – lispowe try..finally
- WITH-OPEN-FILE – opakowanie UNWIND-PROTECT

```
;; unwind działa dobrze
(unwind-protect (open "asdf") (print "bye"))


;; with-open-file też jest git
(with-open-file (stream "/home/mmi/some.txt")
  (format t "~a~%" (read-line stream)))

;;
(with-open-file (out (ensure-directories-exist name)
                     :direction :output)
   ...
   )
```

Wstęp

- Coś więcej niż tylko Linux czy Windows
- Ścieżki reprezentowane jak pathname objects

pathname designators

namestring

Ścieżka zapisana w składni systemu operacyjnego. Np. nazwy podane przez użytkownika.

pathname

Ścieżka jako struktura. Byt abstrakcyjny. Generowane programowo.

strumień z OPEN

Nazwa, która została użyta do utworzenia strumienia.

Jeżeli funkcja przyjmuje nazwę pliku, to akceptuje każdy z powyższych bytów.

Z czego składa się ścieżka?

- host

Z czego składa się ścieżka?

- host
- device

Z czego składa się ścieżka?

- host
- device
- directory

Z czego składa się ścieżka?

- host
- device
- directory
- name

Z czego składa się ścieżka?

- host
- device
- directory
- name
- type

Z czego składa się ścieżka?

- host
- device
- directory
- name
- type
- version

Utworzenie ścieżki z niczego może być trudne – /home/ czy /Users/

Uzyskiwanie pathname designatora

PATHNAME foo

- pathname – nie robi nic
- namestring – zmienia na pathname. Parsowane zgodnie z SO. Standard nie precyzuje jak, ale implementacje używają tej samej konwencji (per SO).
- stream – wypakowuje nazwę ze strumienia

Mapowanie

- Unix używa directory, name i type
- Windows używa dodatkowo host lub device

```
;; Wyciąganie składowych
(pathname-directory (pathname "/foo/bar/baz.txt"))
(pathname-directory (pathname "../foo/bar/baz.txt"))
(pathname-name (pathname "/foo/bar/baz.txt"))
(pathname-type (pathname "/foo/bar/baz.txt"))

; PATHNAME-HOST
; PATHNAME-DEVICE
; PATHNAME-VERSION


(pathname "/foo/bar/baz.txt") => #P"/foo/bar/baz.txt"
; zapis/odczyt
```

Tworzenie abstrakcyjnej ścieżki

```
;; Tworzenie abstrakcyjnej ścieżki
(make-pathname
  :directory '(:absolute "foo" "bar")
  :name "baz"
  :type "txt") ==> #p"/foo/bar/baz.txt"


;; device czy host? Nieprzenośne
(make-pathname :device "c"
               :directory '(:absolute "foo" "bar")
               :name "baz")
```

Tworzenie tylko części ścieżki

```
;; Domyślne wartości
(make-pathname :type "html" :defaults input-file)

;; Zmiana katalogu na backup
(make-pathname :directory '(:relative "backup")
               :defaults input-file)
```

Łączenie ścieżek

```
;; Łączenie ścieżek (#p nie jest obowiązkowe)
(merge-pathnames
    #p"foo/bar.html"
    #p"/www/html/") ;; #p/www/html/foo/bar.html

(merge-pathnames #p"foo/bar.html" #p"html/xml/")
; #p"html/xml/foo/bar.html"

;; Względna dla podanego roota. namestring?
(enough-namestring #p"/www/html/foo/bar.html" #p"/www/")
; "html/foo/bar.html"
```

Ścieżki są uzupełniane z *DEFAULT-PATHNAME-DEFAULTS*

```
;; *DEFAULT-PATHNAME-DEFAULTS*
(merge-pathnames #p"foo.txt") ==> #p"/home/mmi/foo.txt"


;; Trzeba uważać na slash na końcu
;; /home/mmi   -- katalog: /home       plik: mmi
;; /home/mmi/  -- katalog: /home/mmi/  plik:
;;
;; Niestety trzeba sobie z tym radzić samemu
;; Practical Common Lisp, rozdz. 15.  pathname-as-directory
;; Podobnie sprawdzenie czy plik istnieje
```

read-time conditionalization

- *FEATURES* – ficzery obecne w implementacji

read-time conditionalization

- *FEATURES* – ficzery obecne w implementacji
- Działa w czasie kompilacji

```
#+feature program  -- wykonuje jeśli jest feature
#-feature program  -- wykonuje jeśli nie ma feature'a

(defun list-directory (dirname)
  (when (wild-pathname-p dirname)
    (error "Can only list concrete directory names."))
  (let ((wildcard (directory-wildcard dirname)))

    #+(or sbcl cmu lispworks)  (directory wildcard)

    #+openmcl   (directory wildcard :directories t)

    #-(or sbcl cmu lispworks openmcl allegro clisp)
    (error "list-directory not implemented")))
```

Inne operacje

- DELETE-FILE
- RENAME-FILE
- ENSURE-DIRECTORY-EXIST
- FILE-WRITE-DATE – ostatni zapis
- FILE-AUTHOR – owner
- FILE-LENGTH – bierze strumień
- FILE-POSITION

Strumienie znaków, STRING-STREAMs

Tworzenie

- MAKE-STRING-INPUT-STREAM
- MAKE-STRING-OUTPUT-STREAM

Użycie

- READ-CHAR, READ-LINE, READ, ...
- FORMAT, PRINT, WRITE-CHAR, WRITE-LINE, ...
- GET-OUTPUT-STREAM-STRING – ! czyści strumień

- Trzeba zamykać strumienie
- WITH-INPUT-FROM-STRING, WITH-OUTPUT-TO-STRING

```
;;
(let ((s (make-string-input-stream "1.23")))
  (unwind-protect (read s)
    (close s)))

;;
(with-input-from-string (s "1.23")
  (read s))

;;
(with-output-to-string (out)
            (format out "hello, world ")
            (format out "~s" (list 1 2 3)))
```

Ciekawe strumienie

- BROADCAST-STREAM – (output) wysyła informację do wszystkich zdefiniowanych strumieni
- CONCATENATED-STREAM – (input) łańcuchowanie strumieni
- TWO-WAY-STREAM – czyta z in, pisze na out
- ECHO-STREAM – jak TWO-WAY-STREAM, ale wszystko co przeczyta ląduje także na strumieniu wyjściowym.

- Czy formatowanie stringów może być trudne?

- Czy formatowanie stringów może być trudne?
- FORMAT – wbudowany język formatowania wyjścia
- SED?

```
;;
(format stream
   ;; Are you ready for this one?

   "~:[{~;[~]~:{~S~:[->~S~;~*~]~:^ ~}~:[~; ~]~
    ~{~S->~^ ~}~:[~; ~]~[~*~;->~S~;->~*~]~:[}~;]~]"
   ;; Is that clear?

;;
(format t "~{~&~VD~}" '(5 37 10 253 15 9847 10 559 5 12))
   37
       253
           9847
       559
   12
NIL
```

Krótki przegląd

```
(format nil "~a" 3.1415926)

(format nil "~R" 376)
(format nil "~:R" 376)
(format nil "~@R" 1999)

(format nil "~[Hello~;World~]" 1)
(format nil "~#[Hello~;World~]" 1)

(format nil "~,3f" 3.1415926)   ; 3 po przecinku
(format nil "~,3$" 3.1415926)   ; 3 przed przecinkiem

(format t "~d" 1000000)    ; 1000000
(format t "~:d" 1000000)   ; 1.000.000
(format t "~@d" 1000000)   ; +1000000
```

- Dyrektywy zaczynają się tyldą. Kończą znakiem odpowiedniej dyrektywy
- Wielkość znaków (w symbolu dyrektywy) nie ma znaczenia)
- Niektóre dyrektywy przyjmują argumenty. Pisane bezpośrednio po tyldzie, oddzielone przecinkiem.
- Parametr to liczba albo quote + znak. Liczbą może być argument – `(format nil "~v$" 3 3.1415926)`

~%	~&	new line, fresh line
~|		page break
~(	~)	capitalization
~<	~>	justification
~[	~]	condition
~{	~}	iteration
~C		character
~D,		decimal integer. Także B, O, X
~R		spell an integer
~P		plural
~F		floating point
~G		~F or ~E, depending upon magnitude
~$		monetary
~A		legibly, without escapes
~S		READably, with escapes

Formatowanie liczb

```
(format nil " 12d"1000000) ==> "1000000"
(format nil " 12,'0d"1000000) ==> "000001000000"
```

Formatowanie daty (2010-05-01)

```
(format nil "????????????????????"2010 5 1)
```

Formatowanie liczb

```
(format nil " 12d"1000000) ==> "1000000"
(format nil " 12,'0d"1000000) ==> "000001000000"
```

Formatowanie daty (2010-05-01)

```
(format nil " 4,'0d- 2,'0d- 2,'0d"2010 5 1)
```

Iterowanie po liście

- Argument dyrektywy jako lista (albo użyć parametru małpy)
- Można sprawdzić ile elementów zostało do przetworzenia
- Można skakać po liście (w szczególności pominąć element lub przetworzyć go dwa razy)

```
(format nil "~{~a, ~}" (list 1 2 3))    ==> "1, 2, 3, "

(format nil "~{~a~^, ~}" (list 1 2 3))  ==> "1, 2, 3"

(format nil "~r ~:*(~d)" 1) ==> "one (1)"  ==> "one (1)"

(defparameter *english-list*
  "~{~#[~;~a~;~a and ~a~:;~@{~a~#[~; and ~:;, ~]~}~]~}")
(format nil *english-list* '())         ==> ""
(format nil *english-list* '(1))        ==> "1"
(format nil *english-list* '(1 2))      ==> "1 and 2"
(format nil *english-list* '(1 2 3))    ==> "1, 2 and 3"
(format nil *english-list* '(1 2 3 4)) ==> "1, 2, 3 and 4"
```