Podstawowe funkcje wejścia–wyjścia

W Objective Camlu wejście i wyjście jest realizowane na sposób imperatywny. Oto spis podstawowych operacji:

```
type in_channel
type out_channel

val stdin : in_channel
val stdout : out_channel
val stderr : out_channel

val print_char    : char -> unit
val print_string  : string -> unit
val print_int     : int -> unit
val print_float   : float -> unit
val print_endline : string -> unit
val print_newline : unit -> unit

val prerr_char    : char -> unit
val prerr_string  : string -> unit
val prerr_int     : int -> unit
val prerr_float   : float -> unit
val prerr_endline : string -> unit
val prerr_newline : unit -> unit
```

```
exception End_of_file
exception Failure of string
exception Sys_error

val read_line  : unit -> string
val read_int   : unit -> int
val read_float : unit -> float

type open_flag =
  Open_rdonly | Open_wronly | Open_append |
  Open_creat  | Open_trunc  | Open_excl |
  Open_binary | Open_text   | Open_nonblock

val open_out     : string -> out_channel
val open_out_bin : string -> out_channel
val open_out_gen :
  open_flag list -> int -> string -> out_channel
val close_out    : out_channel -> unit

val output_char   : out_channel -> char -> unit
val output_string : out_channel -> string -> unit
val output_byte   : out_channel -> int -> unit
```

```
val open_in     : string -> in_channel
val open_in_bin : string -> in_channel
val open_in_gen :
  open_flag list -> int -> string -> in_channel
val close_in    : in_channel -> unit

val input_char : in_channel -> char
val input_line : in_channel -> string
val input_byte : in_channel -> int
```

Korzystając z opisanych funkcji możemy już pisać programy, które komunikują się ze światem. Zaczniemy od bardzo prostego przykładu.

```
(* fib.ml *)
let rec fib n =
  if n < 2 then
    1
  else
    fib (n - 1) + fib (n - 2) ;;

let funkcja_glowna() =
  let arg = int_of_string Sys.argv.(1) in
    print_int (fib arg);
    print_newline();
    exit 0;;

funkcja_glowna();;
```

Mając podany plik, możemy wyprodukować z niego samodzielny program poleceniem:

```
gsg@lipa:~gsg> ocamlc -c fib.ml
gsg@lipa:~gsg> ocamlc -o fib fib.cmo
gsg@lipa:~gsg> ./fib 7
21
```

Następny przykład to program, który czyta kolejne liczby z wejścia i wypisuje na wyjściu odpowiadające im liczby Fibonacciego.

```
(* fib2.ml *)
let rec fib n =
  if n < 2 then
    1
  else
    fib (n - 1) + fib (n - 2) ;;

let rec liczby ()=
  let liczba = int_of_string (read_line ()) in
    print_int (fib liczba);
    print_newline ();
    liczby () ;;

let funkcja_glowna() =
  try
    liczby ();
  with End_of_file ->
    exit 0;;

funkcja_glowna();;
```

A teraz jeszcze jeden przykład — zliczanie znaków i wierszy w pliku.

```
(* wc.ml *)
let rec wc file chars lines =
  (* funkcja odpowiadająca pętli głównej *)
  try
    let line = input_line file in
    let chars’ = String.length line in
      wc file (chars + chars’ + 1) (lines + 1)
  with End_of_file ->
    (chars, lines);;

let print_result chars lines =
  print_string "Data contains ";
  print_int chars;
  print_string " characters and ";
  print_int lines;
  print_endline " lines";;
```

```
let open_input () =
  if Array.length Sys.argv < 2 then
    stdin
  else
    open_in Sys.argv.(1) ;;

let main () =
  let input_source = open_input () in
  let (chars, lines) = wc input_source 0 0 in
    print_result chars lines;
    close_in input_source;;

main();;
```

I na dowód, że działa:

```
gsg@lipa:~gsg> ocamlc -c wc.ml
gsg@lipa:~gsg> ocamlc -o wc wc.cmo
gsg@lipa:~gsg> ./wc
ala ma
kota
^D
Data contains 12 characters and 2 lines
```

Zmienne i przypisania

W Objective Camlu istnieją też referencje, które zachowują się podobnie do tradycyjnych zmiennych, którym można przypisywać wartość (istnieją też inne, powszechnie znane modyfikowalne struktury danych, na przykład tablice).

Referencje są zbliżone do wskaźników, gdyż przydziela się je dynamicznie (operatorem `ref`). Oczywiście zwalniane są automatycznie przez odśmiecacz (nie ma operacji jawnego zwolnienia referencji).

```
# let current_rand = ref 0;;
val current_rand : int ref = {contents=0}

# let random () =
    current_rand := !current_rand * 25713 + 1345;
    ! current_rand;;
val random : unit -> int = <fun>
```

Referencje nie są pojęciem pierwotnym. Podstawową konstrukcją języka jest deklaracja pola rekordu jako modyfikowalnego (referencje można zdefiniować jako rekordy z jednym modyfikowalnym polem).

```
# type mutable_point =
    { mutable x : float;
      mutable y : float } ;;
type mutable_point =
        { mutable x: float; mutable y: float }

# let translate p dx dy =
    p.x <- p.x +. dx; p.y <- p.y +. dy;
val translate : mutable_point -> float
                    -> float -> unit = <fun>

# let my_point = { x = 0.0; y = 0.0 };;
val my_point : mutable_point = {x=0.0; y=0.0}

# translate my_point 1.0 2.0;;
- : unit = ()

# my_point ;;
val my_point : mutable_point = {x=1.0; y=2.0}
```

Tablice i pętle

Oto przykład korzystający z tablic (modyfikowalnych w miejscu), przypisań i pętli (można go oczywiście zapisać bez pętli, korzystając z funkcji rekurencyjnych). Zdefiniujemy imperatywną wersję funkcji sortowania tablicy przez wstawianie.

```
# let insertion_sort a =
    for i = 1 to Array.length a - 1 do
      let val_i = a.(i) in
      let j = ref i in
        while !j > 0 && val_i < a.(!j - 1) do
          a.(!j) <- a.(!j - 1);
          j := !j - 1
        done;
        a.(!j) <- val_i
    done;;
val insertion_sort : 'a array -> unit = <fun>
```

Proszę zwrócić uwagę, że powyższa funkcja modyfikuje tablicę w miejscu, a przy tym daje wynik typu `unit` (czyli żaden sensowny)!

```
# let t = [| 1; 3; 2; 5; 1; 0; 6; 1 |];;
val t : int array = [|1; 3; 2; 5; 1; 0; 6; 1|]

# insertion_sort t;;
- : unit = ()

# t;;
- : int array = [|0; 1; 1; 1; 2; 3; 5; 6|]
```

Naturalnie można pisać funkcje, których implementacja korzysta z konstrukcji imperatywnych, ale nie powoduje skutków ubocznych widocznych na zewnątrz definicji funkcji (np. funkcja, która dostając tablicę daje w wyniku nową tablicę, ale posortowaną, tworząc kopię oryginalnej tablicy i tę kopię sortując imperatywnie).

Korzystając z mechanizmów imperatywnych możemy zaimplementować program `wc` z wcześniejszego przykładu w sposób zupełnie niezgodny z duchem języka funkcyjnego, a bardziej przypominający tradycyjny styl programowania w C lub Pascalu:

```
(* wc1.ml *)
let wc file =
  let chars = ref 0 in
  let lines = ref 0 in
    try
      while true do
        let line = input_line file in
          chars := ! chars +
                      String.length line + 1;
          lines := !lines + 1
      done;
      (!chars, !lines)
    with End_of_file -> (!chars, !lines);;
```

```
let print_result chars lines =
  print_string "Data contains ";
  print_int chars;
  print_string " characters and ";
  print_int lines;
  print_endline " lines";;

let open_input () =
  if Array.length Sys.argv < 2 then
    stdin
  else
    open_in Sys.argv.(1) ;;

let main () =
  let input_source = open_input () in
  let (chars, lines) = wc input_source in
    print_result chars lines;
    close_in input_source;;

main();;
```