Costrutti imperativi

Problema: stampa degli interi compresi tra n e m

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Stdlib.html

Output functions on standard output

val **print_string**: string -> unit
Print a string on standard output.
val **print_int**: int -> unit
Print an integer, in decimal, on standard output.
val **print_newline**: unit -> unit
Print a newline character on standard output.

Il tipo unit

Contiene un solo elemento: **()**

Sequenze di comandi

(E1 ; E2 ; ... ; En)

Se **E1, ..., En** sono espressioni, allora **(E1 ; E2 ; ... ; En)** è un'espressione.

- Il **tipo e valore** di **(E1 ; E2 ; ... ; En)** sono il tipo e valore di **En**.
- **Valutazione** di **(E1 ; E2 ; ... ; En)**: le espressioni vengono valutate tutte, da sinistra a destra, ma i valori sono ignorati, tranne quello dell'ultima espressione.

```
# (print_int 3; print_string "*"; print_int 8;
   print_string " = "; print_int(3*8); print_newline();
   3*8);;
3*8 = 24
- : int = 24
# (3*8; print_string "ciao\n"; 10);;
Characters 1-4:
  (3*8; print_string "ciao\n"; 10);;
```

Sequenze di comandi

(E1 ; E2 ; ... ; En)

Se **E1, ..., En** sono espressioni, allora **(E1 ; E2 ; ... ; En)** è un'espressione.

- Il **tipo e valore** di **(E1 ; E2 ; ... ; En)** sono il tipo e valore di **En**.
- **Valutazione** di **(E1 ; E2 ; ... ; En)**: le espressioni vengono valutate tutte, da sinistra a destra, ma i valori sono ignorati, tranne quello dell'ultima espressione.

```
# (print_int 3; print_string "*"; print_int 8;
   print_string " = "; print_int(3*8); print_newline();
   3*8);;
3*8 = 24
- : int = 24
# (3*8; print_string "ciao\n"; 10);;
Characters 1-4:
  (3*8; print_string "ciao\n"; 10);;
   ^^^
Warning 10: this expression should have type unit.
ciao
- : int = 10
```

Come si implementa un ciclo?

Problema: dato un intero m, stampare i numeri compresi tra 0 e m.

Sottoproblema: stampare gli interi compresi tra n e m.

```
(* ciclo: int -> int -> unit
ciclo n m: stampa gli interi da n a m (estremi inclusi) *)
let rec ciclo n m =
  if n>m then ()
  else (print_int n;     (* sequenza di comandi *)
        print_newline();
        ciclo (n+1) m)

(* stampa: int -> unit
   stampa m: stampa degli interi da 0 a m *)
let stampa = ciclo 0
```

ciclo è **tail recursive**: al ritorno dalla chiamata ricorsiva non si deve fare nulla. Implementa un'**iterazione**.

Ricorsione e iterazione

```
let rec fact = function 0 -> 1
                      | n -> n * fact(n-1)
```

fact non è tail recursive

```
fact 3 ==> 3 * fact 2
       ==> 3 * (2 * fact 1)
       ==> 3 * (2 * (1 * fact 0))
       ==> 3 * (2 * (1 * 1))
       ==> 3 * (2 * 1)
       ==> 3 * 2 ==> 6
```

Ma il prodotto è associativo, quindi: $3 \times (2 \times \mathit{fact}\ 1) = (3 \times 2) \times \mathit{fact}\ 1$, e:

```
fact 3 = 3 * fact 2
       = (3 * 2) * fact 1
       = (6 * 1) * fact 0
       = 6 * 1 = 6
```

Non c'è bisogno di aspettare il risultato delle chiamate ricorsive, possiamo eseguire subito il calcolo 3×2 e conservarlo in un "accumulatore" (o risultato parziale) $\Longrightarrow$ **algoritmo iterativo**

Algoritmo iterativo per il calcolo del fattoriale

```
fact(n) =
    f <- 1;
    while (n > 0)
        do f <- f*n;
           n <- n-1
        done;
    return f
```

Nei linguaggi funzionali non esiste l'assegnazione.

Il ciclo viene implementato mediante un costrutto ricorsivo:

- utilizziamo una funzione ausiliaria che ha un parametro in più ("accumulatore"): i suoi argomenti sono le variabili che vengono "modificate" nel ciclo;
- l'operazione principale richiama quella ausiliaria "inizializzando" l'accumulatore

Implementazione del ciclo mediante una funzione ausiliaria

- Parametri della funzione ausiliaria: le variabili di ciclo (**n** e **f**);
- Corpo della funzione ausiliaria:
  if (condizione di uscita dal ciclo)
  then (valore da riportare in uscita dal ciclo)
  else <chiamata ricorsiva> su
  (argomenti modificati come nel ciclo stesso)
- La funzione ausiliaria viene richiamata da quella principale con argomenti uguali ai valori con cui sono inizializzate le variabili di ciclo

Implementazione del ciclo mediante una funzione ausiliaria

- Parametri della funzione ausiliaria: le variabili di ciclo (**n** e **f**);
- Corpo della funzione ausiliaria:
  if n=0 (condizione di uscita dal ciclo)
  then f (valore da riportare in uscita dal ciclo)
  else <chiamata ricorsiva> su f*n e n-1
  (argomenti modificati come nel ciclo stesso)
- La funzione ausiliaria viene richiamata da quella principale con argomenti uguali ai valori con cui sono inizializzate le variabili di ciclo (**n** stesso e **f=1**).

```
(* fact' : int -> int *)
let rec fact' n =
  (* aux: int -> int -> int
    il primo argomento e` il "risultato parziale" *)
  let rec aux f = function
      0 -> f    (* il "ciclo" termina *)
    | n -> aux (f*n)(n-1)  (* iterazione successiva *)
  in aux 1 n (* inizializzazione *)
```

Specifica della funzione ausiliaria

Cosa calcola **aux**? (Redichiariamola a top level)

```
# fact' 5;;
- : int = 120
# aux 6 5;;
- : int = 720
```

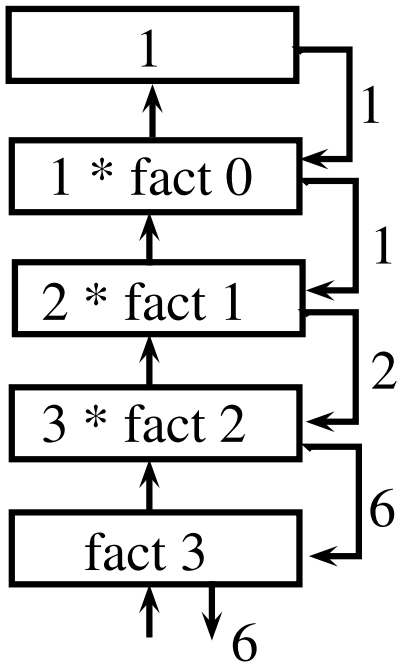
aux m n moltiplica per **m** il fattoriale di **n**

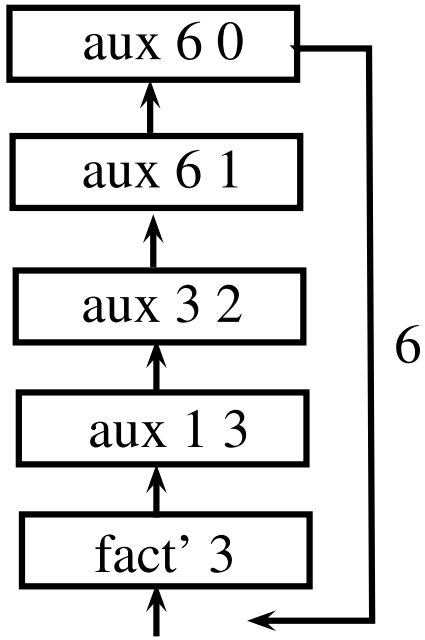
Il processo per il calcolo di fact' è iterativo

```
fact' 3 = aux 1 3
        = aux 3 2
        = aux 6 1
        = aux 6 0 = 6
```

Il processo è "lineare": dopo aver raccolto il risultato della chiamata ricorsiva, non si deve fare nulla.

Processi ricorsivi e processi iterativi





Processi ricorsivi e iterativi (II)

Un **processo ricorsivo**:

1. Esegue calcoli al ritorno dalla ricorsione
2. Usa **spazio** proporzionale al numero di chiamate ricorsive

In un **processo iterativo**:

1. I calcoli vengono tutti eseguiti prima della chiamata ricorsiva e il risultato parziale viene conservato in un **accumulatore**
2. Dopo aver raccolto il risultato della chiamata ricorsiva non si deve fare nulla
3. L'ultima chiamata può riportare il suo risultato direttamente alla prima
4. Si usa **spazio** costante

Quando un problema P1 viene convertito in un altro P2, in modo che la soluzione di P2 è identica alla soluzione di P1 (non servono altri calcoli), allora

P1 è stato ridotto a P2

P2 è una **riduzione** di P1

Quando una funzione ricorsiva è definita in modo tale che tutte le chiamate ricorsive sono riduzioni, allora la funzione è

RICORSIVA DI CODA (TAIL RECURSIVE)

Molti compilatori riconoscono la ricorsione di coda

conta_digits di nuovo

conta_digits: string -> int
conta_digits s = numero di caratteri numerici in s

```
let conta_digits s =
  let rec loop i =
    try if numeric s.[i] then 1 + loop (i+1)
        else loop (i+1)
    with _ -> 0
  in loop 0
```

loop non implementa un'iterazione

Versione iterativa

conta_digits di nuovo

conta_digits: string -> int
conta_digits s = numero di caratteri numerici in s

```
let conta_digits s =
  let rec loop i =
    try if numeric s.[i] then 1 + loop (i+1)
        else loop (i+1)
    with _ -> 0
  in loop 0
```

loop non implementa un'iterazione

Versione iterativa

```
let conta_digits s =
  let rec loop i result =
    try if numeric s.[i] then loop (i+1) (1+result)
        else loop (i+1) result
    with _ -> result
  in loop 0 0
```

result rappresenta il "risultato parziale" (o "accumulatore"), inizializzato a 0

Lettura da tastiera

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Stdlib.html

Input functions on standard input

val **read_line**: unit -> string
Flush standard output, then read characters from standard input until a newline character is encountered. Return the string of all characters read, without the newline character at the end.

val **read_int**: unit -> int
Flush standard output, then read one line from standard input and convert it to an integer. Raise Failure "int_of_string" if the line read is not a valid representation of an integer.

Ciclo di lettura da tastiera

Problema: leggere una sequenza di interi terminata da un punto e riportarne la somma

```
(* somma: unit -> int *)
let rec somma () =
  let s = read_line ()
  in if s="." then 0
  else (int_of_string s) + somma()
```

Versione iterativa

Ciclo di lettura da tastiera

Problema: leggere una sequenza di interi terminata da un punto e riportarne la somma

```
(* somma: unit -> int *)
let rec somma () =
  let s = read_line ()
  in if s="." then 0
  else (int_of_string s) + somma()
```

Versione iterativa

```
let somma () =
  (* loop: int -> int, funzione ausiliaria che implementa
                       il ciclo*)
  let rec loop result = (* result: accumulatore *)
    let s = read_line ()
    in if s="." then result
    else loop ((int_of_string s) + result)
  in loop 0          (* result inizializzato a 0 *)
```

Specifica dichiarativa delle funzioni ausiliarie

```
(* loop: int -> int *)
let rec loop result = (* result: accumulatore *)
    let s = read_line ()
    in if s="." then result
    else loop ((int_of_string s) + result)
```

Che cosa calcola **loop** in generale?

loop n = ???

Specifica dichiarativa delle funzioni ausiliarie

```
(* loop: int -> int *)
let rec loop result = (* result: accumulatore *)
    let s = read_line ()
    in if s="." then result
    else loop ((int_of_string s) + result)
```

Che cosa calcola **loop** in generale?

loop n = n + somma degli interi letti da tastiera

Uso "sporco" delle eccezioni

```
let somma () =
  let rec loop result =
    try let n = int_of_string (read_line())
        in loop (n+result)
    with _ -> result
  in loop 0
```

Problema: determinare numero e somma degli interi letti

```
(* numero_somma: unit -> int * int
   riporta numero e somma degli interi letti *)
let rec numero_somma () =
  let s = read_line() in
  if s="."  (* terminato? *)
  then (0,0) (* nessun numero letto, somma 0 *)
  else (* s rappresenta un int, leggi gli altri numeri *)
    let (tot,somma) = numero_somma()
    in (tot+1,somma+(int_of_string s))

(* oppure *)
let rec numero_somma () =
  try let n = int_of_string(read_line())
      in let (tot,somma) = numero_somma()
      in (tot+1,somma+n)
  with _ -> (0,0)
```

Versione iterativa di numero_somma

```
let rec numero_somma () =
  try let n = int_of_string(read_line())
      in let (tot,somma) = numero_somma()
      in (tot+1,somma+n)
  with _ -> (0,0)

let numero_somma_it () =
  (* aux: int -> int -> int * int *)
  let rec aux tot somma =  (* due "accumulatori" *)
    try
      aux (tot+1) (somma + (int_of_string(read_line())))
    with _ -> (tot,somma)
  in aux 0 0
```

General input/output functions

http://caml.inria.fr/pub/docs/manual-ocaml/libref/Stdlib.html

type **out_channel**
stdout: out_channel
open_out: string -> out_channel
output_string: out_channel -> string -> unit
close_out: out_channel -> unit

type **in_channel**
stdin: in_channel
open_in: string -> in_channel
input_line: in_channel -> string
close_in: in_channel -> unit

Esempio

media: string -> unit

media <nomefile> legge dal file con il nome dato una sequenza di numeri interi e stampa il numero degli interi letti, la loro somma e la loro media

Vai al codice