Programmazione Funzionale in OCaml

Dispensa per il corso di Paradigmi di Programmazione

27 settembre 2021

Indice

1 Un primo sguardo a OCaml **3**
1.1 Introduzione a OCaml . . . 3
1.2 Un esecutore di OCaml, per cominciare. . . 4
1.3 Formato delle variabili . . . 5
1.4 Tutto è un'espressione . . . 6
1.5 Tipi di base (basic types) . . . 6
1.6 I tipi tupla . . . 9
1.7 Funzioni . . . 10
1.8 Il λ-calcolo in OCaml (sintatticamente) . . . 13
1.9 Altro modo (più comune) di definire funzioni: `let` . . . 14
1.10 "Scope" delle dichiarazioni e funzioni ausiliarie . . . 15
1.11 Funzioni *Curryed* . . . 17
1.12 Applicazione parziale di funzione . . . 18
1.13 Controllo del flusso di esecuzione . . . 19
1.14 Espressione di scelta condizionale `if` . . . 19
1.15 Ricorsione . . . 20
1.16 Type Inference . . . 22

2 Liste e Pattern Matching **29**
2.1 Liste . . . 29
2.2 Riflessione sul tipo di una lista . . . 30
2.3 L'operatore cons `::` . . . 31
2.4 Concatenazione di liste (append - `@`) . . . 33
2.5 Altre operazioni su liste (OCaml API) . . . 33
2.6 Pattern Matching . . . 34
2.7 Sintassi dei pattern . . . 34
2.8 Pattern matching e esaustività . . . 36
2.9 La vera potenza del Pattern Matching . . . 38
2.10 Funzioni ricorsive su liste . . . 40
2.11 Un esempio non banale: area di un poligono irregolare . . . 41
2.12 Funzioni higher-order su liste . . . 44
2.13 Un paio di esempi con `fold_right` e `fold_left` . . . 53
2.14 Le funzioni su liste nel modulo `List` . . . 54
2.15 Funzioni higher-order su liste in altri linguaggi . . . 55

3 Tipi Algebrici (Record e Variant) **57**
3.1 Definire nuovi tipi . . . 57
3.2 Tipi algebrici . . . 57
3.3 Tipi prodotto: tuple e record . . . 58
3.4 Record . . . 58
3.5 Nota: record VS oggetti . . . 61
3.6 Tipi unione (o somma): variant . . . 62
3.7 Tipi record e variant insieme . . . 66
3.8 Tipi ricorsivi . . . 67

4 Programmazione imperativa in OCaml (cenni) **73**

4.1 Introduzione 73
4.2 Array 73
4.3 Record mutabili 75
4.4 Riferimenti (refs) 75
4.5 Sequenze di comandi e cicli 77
4.6 Eccezioni 80

Capitolo 1

Un primo sguardo a OCaml

In questa dispensa saranno illustrate alcune caratteristiche di base del linguaggio OCaml, in particolare relative al paradigma di programmazione funzionale, che quello su cui OCaml si basa principalmente. Il linguaggio prevede comunque aspetti di programmazione imperativa e a oggetti che saranno brevemente illustrate.

Riferimenti Questa dispensa si basa in parte su materiale ed esempi di codice tratti da:

- Real World OCaml (Versione 2), di Yaron Minsky, Anil Madhavapeddy e Jason Hickey

La licenza per l'utilizzo del testo e degli esempi di codice è disponibile nel sito indicato.

A differenza del libro "Real World OCaml", in questa dispensa utilizzeremo la libreria standard di OCaml, e non la più moderna libreria Base utilizzata nel libro.

Inoltre, alcuni aspetti del linguaggio possono essere approfonditi consultando il manuale ufficiale di OCaml, disponibile a questo indirizzo:

- https://ocaml.org/manual/index.html

1.1 Introduzione a OCaml

Per introdurre OCaml è bene partire dai sui antenati:

ML (Meta Language) proposto da Robin Milner nel 1972 e sviluppato negli anni '80

- linguaggio funzionale *fortemente tipato* con avanzate funzionalità di "type inferece"
- prevede funzionalità caratteristiche della programmazione funzionale: funzioni come valori, variabili immutabili, polimorfismo, pattern matching, ...
- diventato linguaggio funzionale di riferimento

Caml (Categorical Abstract Machine Language) sviluppato negli anni '80/'90 all'INRIA in Francia

- versione di ML dotata di una implementazione particolarmente efficiente
- introduce la possibilità di definire moduli e fornisce una serie di librerie di programmazione

OCaml (Objective Caml) nato nel 1996 all'INRIA in Francia

- estende Caml con funzionalità di programmazione orientata agli oggetti

Perché NON studiare OCaml

OCaml *NON è* un linguaggio tra i più *popolari*

- su Google Trends si vede che le ricerche di "python", "javascript", "java" sono migliaia di volte più frequenti che quelle di "ocaml"

OCaml *NON è* un linguaggio tra i più *semplici* da imparare, usare e leggere

- i linguaggi maggiormente incentrati sui costrutti della programmazione imperativa solitamente sono più intuitivi

OCaml (probabilmente) *NON sarà* il linguaggio del *futuro*

- è in circolazione da qualche decennio
- esistono dei settori in cui ha successo (dove conta la qualità del codice)...
- ... ma è difficile immaginare che il suo utilizzo esploderà in futuro

Perché, invece, E' BENE studiare OCaml

OCaml ci consentirà di vedere la *programmazione funzionale realizzata al meglio*

- vedremo bene le caratteristiche (vantaggi e svantaggi) di questo paradigma

Molti *linguaggi "moderni"* stanno incorporando (a volte in malo modo) aspetti di programmazione funzionale presenti in OCaml da *decenni*

- studiare OCaml consentirà di comprendere questi aspetti una volta per tutte

OCaml consente di implementare in modo abbastanza naturale le semantiche statiche e dinamiche dei linguaggi di programmazione

- lo useremo per *sviluppare interpreti* di linguaggi su paradigmi diversi
- vedremo "sotto al cofano" di questi linguaggi

1.2 Un esecutore di OCaml, per cominciare...

Per vedere le varie caratteristiche del linguaggio, utilizziamo un interprete di OCaml (o **interactive toplevel**), quali:

- `ocaml`: l'interprete standard eseguibile da linea di comando dopo l'installazione di OCaml
- `utop`: un interprete con un'interfaccia più avanzata da installare separatamente
- https://try.ocamlpro.com/: un interprete online
- *questo stesso documento* (versione Jupyter Notebook): eseguendo gli esempi di codice presenti nel testo

Come si usa l'interactive toplevel di OCaml

Il toplevel di OCaml esegue un REPL (Read-Eval-Print loop)

Fornisce una linea di comando in cui è possibile scrivere espressioni (nella sintassi di OCaml)

- l'interprete *compila* e *valuta* immediatamente ogni espressione, fornendone il risultato e una informazione sul suo *tipo* (o un messaggio di errore)
- il risultato dell'espressione può essere legato a un nome, ottenendo una *dichiarazione* (di "variabili", funzioni, ...)
- un'espressione può utilizzare risorse ("variabili", funzioni,...) che siano state dichiarate precedentemente

Vediamolo all'opera

Una dichiarazione:

[1]:
```
let x = 10 ;;
```

[1]:
```
val x : int = 10
```

E un'espressione:

[2]:
```
x + 5 ;;
```

[2]:
```
- : int = 15
```

In questi primissimi esempi ci sono già alcune cose che possono essere osservate. Innanzitutto, la sintassi della dichiarazione con il costrutto `let` è piuttosto comune, e presente in altri linguaggi quali, in particolare, JavaScript. Più precisamente, `let` crea un *binding*, ossia un legame, tra un *nome* e un *valore*. Caratteristica fondamentale della programmazione funzionale è che anche le funzioni sono valori. Quindi `let` potrà essere usato anche per dichiarare funzioni.

La risposta fornita dall'interprete è simile nei due casi. Dopo il simbolo `:` abbiamo il tipo dell'espressione che OCaml ha *inferito* (ossia, OCaml ha dedotto il tipo di quello che sarà il risultato della valutazione dell'espressione stessa). Entrambe le espressioni, `10` e `x+5`, hanno tipo `int`. Vedremo in seguito che OCaml inferisce questa informazione di tipo esaminando la struttura dell'espressione, e non calcolandone il risultato. Inoltre, dopo il simbolo `=` abbiamo il risultato vero e proprio della valutazione dell'espressione.

Quello che cambia, nei due esempi di output forniti dall'interprete, è la prima parte. Nell'esempio di dicharazione l'intepete, con `val x` informa di aver creato un nuovo binding per la variabile `x` al valore `10`. Nel secondo esempio, non essendo stato usato il costrutto `let`, l'interprete informa tramite il simbolo `-` di non aver creato alcun binding. Si è limitato ad inferire il tipo dell'espressione e calcolarne il risultato.

Altra cosa da osservare è il fatto che in entrambi gli esempi mostrati si è usato il terminatore `;;` che consente al toplevel di OCaml di capire che si è terminato l'inserimento dell'espressione. Non è sempre necessario inserire questo terminatore. Ad esempio, rimuovendolo da entrambi gli esempi di codice inclusi in questo documento interattivo, il risultato che si ottiene non cambia.

Questo non è vero per altri interpreti di OCaml (ad esempio il toplevel di default `ocaml`) che non valutano l'espressione fintanto che non è stato inserito il terminatore `;;`. Inoltre, anche in questo documento interattivo, quando si volesse inserire più di una espresisone in una unica cella di codice, sarebbe necessario usare il terminatore tra un'espressione e l'altra.

1.3 Formato delle variabili

I nomi (o identificatori) delle variabili dichiarate tramite il costrutto `let`:

- possono contenere lettere, numeri e i caratteri `_` e `'`
- devono iniziare con una lettera minuscola o con `_`

Esempi corretti:

[3]:
```
let x = 3 + 4 ;;
```

[3]:
```
val x : int = 7
```

[4]:
```
let x_plus_3 = x + 3 ;;
```

[4]:
```
val x_plus_3 : int = 10
```

[5]:
```
let x' = x + 1 ;;
```

[5]:
```
val x' : int = 8
```

Esempi errati:

[6]:
```
let Seven = 3 + 4 ;;
```

```
File "[6]", line 1, characters 4-9:
1 | let Seven = 3 + 4 ;;
        ^^^^^
Error: Unbound constructor Seven
```

[7]:
```
let 7x = 7 ;;
```

```
File "[7]", line 1, characters 4-6:
1 | let 7x = 7 ;;
        ^^
Error: Unknown modifier 'x' for literal 7x
```

[8]:
```
let x-plus-3 = x + 3 ;;
```

```
File "[8]", line 1, characters 6-10:
1 | let x-plus-3 = x + 3 ;;
          ^^^^
Error: Syntax error
```

1.4 Tutto è un'espressione

Coerentemente con il paradigma funzionale, che non prevede uno "stato" del programma modificabile, i costrutti fondamentali del linguaggio non sono *comandi*, ma *espressioni*

Le espressioni possono contenere *dichiarazioni* (o meglio "bindings")

1.5 Tipi di base (basic types)

OCaml è un linguaggio *fortemente tipato*. I tipi di dato di base sono i seguenti:

- `int`: numeri interi
- `float`: numeri frazionari *floating point* a doppia precisione
- `bool`: valori di verità (booleani): `true` o `false`
- `char`: singoli caratteri (racchiusi tra apici: `'a'`)
- `string`: stringhe (racchiuse tra virgolette: `"abcd"`)
- `unit`: tipo usato in casi particolari (simile a `void` in altri linguaggi), prevede come unico valore `()`

Operazioni su `int` e `float`

Le operazioni su numeri frazionari (`float`) si scrivono diversamente da quelle su interi (`int`)

Si scrivono seguite da un punto: `+. -. *. /.`

[9]:
```
3 + 5 ;; (* operazione su interi: tipo int *)
```

[9]:
```
- : int = 8
```

[10]:
```
3.2 +. 5.2 ;; (* operazione su frazionari: tipo float *)
```

[10]: - : float = 8.4

Il tipo degli operandi deve essere *coerente* con quello dell'operazione, altrimenti il compilatore segnala un errore

I valori di tipo `float` che hanno la parte frazionaria (dopo la virgola) nulla sono rappresentati da OCaml con niente dopo il punto. Ad esempio, il numero `3.0` viene in realtà rappresentato come `3.`.

Conversione di tipo tra `int` e `float`

OCaml non prevede conversioni di tipo implicite (ossia, automatiche)

- ad esempio, il valore `3` non può essere mai usato al posto di `3.0`

Le conversioni devono essere esplicitate usando `float_of_int` e `int_of_float`

[11]:
```
float_of_int 3 ;;
int_of_float 3.0 ;;
```

[11]: - : float = 3.

[11]: - : int = 3

[12]:
```
(float_of_int 3) +. 2.4 ;;
```

[12]: - : float = 5.4

Conversioni di tipo (type cast) nella maggior parte dei linguaggi fortemente tipati

Nella maggior parte dei linguaggi di programmazione fortemente tipati (ad esempio, in C e in Java) il compilatore può effettuare delle conversioni di tipo implicite. In questi linguaggi, ad esempio, è possibile scrivere un'espressione quale `3 + 2.5` assumendo che il compilatore interpreterà automaticamente il valore `3` come se fosse `3.0`. Questa conversione avviene nel momento in cui il compilatore si accorge che l'altro operando dell'operazione di somma è un valore (o un'espressione) di tipo `float`.

Più precisamente, le conversioni implicite consentono di *promuovere* un valore o un'espressione facendole assumere un tipo più *generale*. Ad esempio, il tipo del valore `3` può essere promosso implicitamente da `int` a `float`, mentre il tipo del valore `3.0` non può essere implicitamente declassato da `float` a `int`. Il tipo `int`, infatti, è un tipo meno generale di `float`, in quanto non è in grado di rappresentare la grande maggioranza dei valori tipabili come `float`. Ad esempio, il valore `2.5`, che è tipabile come `float` non ha una rappresentazione esatta tipabile come `int`.

Sempre in linguaggi quali C e Java, è possibile forzare le conversioni da un tipo più generale a uno meno generale utilizzando un *type cast*. Questo avviene, di solito, premettendo al valore a cui applicare la forzatura, il nome del tipo di destinazione racchiuso tra parentesi. Ad esempio, l'espressione `(int) 3.2` in C o in Java forza il compilatore a considerare il valore `3.2` come se fosse di tipo `int`. In questo modo, tale valore potrà essere utilizzato ovunque sia richiesto esplicitamente un valore di tipo `int` (ad esempio, passato a una funzione che preveda un parametro di tale tipo).

Il type cast in questi linguaggi ha due effetti concreti: in primo luogo applica una forzatura nella fase di *type checking* svolta dal compilatore, consentendo, ad esempio, di completare la compilazione; in secondo luogo, fa sì che a tempo di esecuzione un valore debba concretamente essere trasformato. Ad esempio, il valore `3.2` dell'esempio dovrà, a tempo di esecuzione, essere trasformato in un valore intero. Questo solitamente causa una *perdita di precisione*. Nel caso di `(int) 3.2` il risultato concreto che si otterrà (ad esempio in Java) è che il valore `3.2` sarà trasformato nel valore intero `3` tramite un

troncamento della parte decimale del numero, che introduce quindi una approssimazione nel calcolo che si sta operando.

Conversioni di tipo in OCaml

In OCaml non sono previste conversioni di tipo implicite. E' possibile effettuare delle conversioni esplicite da `int` a `float` e viceversa utilizzando le funzioni `float_of_int` e `int_of_float` menzionate sopra. I motivi per cui in OCaml si è scelto di proibire le conversioni implicite sono essenzialmente due: il primo è che le conversioni implicite sono spesso la causa di errori di programmazione difficili da scovare; il secondo è che (come vedremo) OCaml fa un uso molto sofisticato delle informazioni sui tipi, per consentire il quale ha la necessità di determinare in modo accurato il tipo di ogni espressione contenuta nel programma. Le conversioni implicite possono introdurre delle ambiguità in tali elaborazioni e quindi non sono contemplate nel linguaggio.

Operazioni su `bool`

Vediamo la sintassi delle espressioni booleane con qualche esempio:

[13]:
```
let x = true ;;
```

[13]:
```
val x : bool = true
```

[14]:
```
x && false ;;
x || false ;;
not x;;      (* not è l'operazione di negazione -- come ! in molti altri linguaggi *)
```

[14]:
```
- : bool = false
```

[14]:
```
- : bool = true
```

[14]:
```
- : bool = false
```

[15]:
```
1 = 2 ;;    (* ATTENZIONE, il confronto si fa con =, non con == ! *)
1 <> 2 ;;   (* mentre per vedere se due espressioni sono diverse si usa <>, non != *)
```

[15]:
```
- : bool = false
```

[15]:
```
- : bool = true
```

La sintassi degli operatori logici è cambiata nel tempo ed è diversa in diversi "dialetti" del linguaggio ML. In alcuni linguaggi (e anche in alcune vecchie versioni di OCaml), al posto di `&&` e `||` capita di trovare `and` e `or`, oppure `&` e `|`. Se si riscontrano problemi con questi operatori, è bene controllare la documentazione ufficiale del linguaggio (e della versione) che si sta utilizzando.

Inoltre, come in molti altri linguaggi di programmaizone, le operazioni logiche `&&` e `||` sono *cortocircuitate*. Questo significa che se nella valutazione di una espressione `e1 && e2` la sotto espressione `e1` risulta essere `false`, dal momento che questo porterà l'intera espressione `e1 && e2` ad essere `false`, l'interprete evita di valutare (inutilmente) `e2`. Analogamente, nella valutazione di una espressione `e1 || e2` la sotto espressione `e1` risulta essere `true`, l'interprete evita di valutare (inutilmente) `e2` dando immediatamente `true` come risultato complessivo.

Questo modo di valutare le operazioni logiche migliora l'efficienza dell'esecuzione del programma e, in alcuni casi, può essere sfruttata per semplificare la scrittura dei programmi.

Operazioni su `char` e `string`

Anche in questo caso, vediamo alcuni esempi:

[16]:
```
let c = 'a' ;;
int_of_char c ;; (* conversione esplicita, esiste anche char_of_int *)
```

[16]:
```
val c : char = 'a'
```

[16]:
```
- : int = 97
```

[17]:
```
let h = "hello" ;;
let hw = h ^ "world" ;; (* concatenazione con simbolo ^ *)
```

[17]:
```
val h : string = "hello"
```

[17]:
```
val hw : string = "helloworld"
```

[18]:
```
int_of_string "10";; (* conversione esplicita, esistono anche per float, bool *)
```

[18]:
```
- : int = 10
```

Ulteriori operazioni tramite l'OCaml API (https://ocaml.org/api/index.html)

1.6 I tipi tupla

OCaml prevede diverse tipologie di *tipi strutturati*. Uno tra i più semplici sono i *tipi tupla*.

Una tupla (o ennupla) è una sequenza di valori separati da virgole.

[19]:
```
let t = (10,"hello",12.5) ;;
```

[19]:
```
val t : int * string * float = (10, "hello", 12.5)
```

Il tipo di una tupla è il *prodotto* dei tipi degli elementi che la compongono

I concetti di tupla e prodotto provengono dalla *teoria degli insiemi*.

Se immaginiamo i tipi come insiemi di valori:

- il tipo tupla corrisponde al loro *prodotto cartesiano*
- una singola tupla è un elemento di tale prodotto cartesiano

Ad esempio, possiamo immaginare che i tipi `int` e `float` corrispondano rispettivamente (in maniera approssimata) agli insiemi dei numeri interi $\mathbb{Z}$ e reali $\mathbb{R}$. Il tipo `int*float` corrisponde all'insieme $\mathbb{Z} \times \mathbb{R} = \{(n,m) | n \in \mathbb{Z}, m \in \mathbb{R}\}$, dove $\times$ è il simbolo che descrive solitamente l'operazione di prodotto cartesiano tra insiemi. Quindi dire che il valore `(3,4.2)` è di tipo `int*float` corrisponde in realtà a dire $(3, 4.2) \in \mathbb{Z} \times \mathbb{R}$.

Esempio: piano cartesiano

Ad esempio, supponiamo di voler rappresentare i punti del piano cartesiano

- sono coppie di numeri reali, ossia elementi di $\mathbb{R} \times \mathbb{R}$

Definiamo un punto in OCaml rappresentato come coppia di `float`

```
[20]: let p = ( 3.0 , 5.0 ) ;;  (* le parentesi in realtà non sono necessarie *)
```

```
[20]: val p : float * float = (3., 5.)
```

Ha tipo `float * float`

Una cosa interessante è che possiamo *destrutturare* una tupla, usando `let` in modo particolare per risalire ai singoli elementi:

```
[21]: let (x,y) = p;;
      x +. y;;
```

```
[21]: val x : float = 3.
      val y : float = 5.
```

```
[21]: - : float = 8.
```

Vedremo che questo tipo di operazioni di destrutturazione fanno parte dei potenti meccanismi di *pattern matching* di OCaml

Nello scrivere `let (x,y) = p` è importante scrivere a sinistra dell'uguale un *pattern* (in questo caso una tupla contentente variabili) che abbia la stessa struttura del valore ottenuto dall'espressione a destra dell'uguale (in questo caso `p`). In altre parole, dal momento che il tipo di `p` è `float * float`, il pattern che specifichiamo nel `let` dovrà essere una tupla con due variabili. Così facendo, `x` sarà legata al primo elemento di `p`, `y` sarà legata al secondo, e così via.

1.7 Funzioni

Sempre in coerenza con il paradigma funzionale, in OCaml le funzioni sono *valori*:

- possiamo ottenere una funzione come risultato della valutazione di un'espressione
- possiamo passare una funzione come parametro ad un'altra funzione
- una funzione può essere restituita come risultato di un'altra funzione
- ... e molto altro (vedremo) ...

Partendo dal λ-calcolo (con espressioni su interi)

Il modo "base" di definire una funzione riprende quanto visto nel λ-calcolo

Ad esempio, la funzione $\lambda x.x + 1$ può essere definita in OCaml come segue:

```
[22]: fun x -> x + 1 ;;
```

```
[22]: - : int -> int = <fun>
```

e la sua applicazione $(\lambda x.x + 1)10$ diventa, letteralmente:

```
[23]: (fun x -> x + 1) 10 ;;
```

```
[23]: - : int = 11
```

Ci sono un paio di osservazioni da fare sull'output dell'interprete OCaml. In primo luogo, nell'esempio con la definizione della funzione, l'inteprete ha restituito un output simile a quello che avevamo

visto negli esempi di espressioni mostrati in precedenza: abbiamo il simbolo `-`, che denota che non è stato effettuato alcun binding (non abbiamo dichiarato una variabile), poi abbiamo il tipo inferito dell'espressione, in questo caso `int -> int`, che descrive una funzione che prende un intero come parametro e restituisce un intero. Infine, dopo il simbolo = abbiamo il risultato della valutazione dell'espressione, che è genericamente `<fun>`, non essendo possibile per l'interprete rappresentare in generale i "valori-funzione" in formato testuale.

Nel secondo esempio, con l'applicazione della funzione, l'interprete risponde nuovamente con - (nessun binding), `int` (il tipo del risultato della valutazione dell'espressione) e `11` (il risultato vero e proprio).

Entrambi questi esempi si configurano come vere e proprie espressioni: ne può essere inferito un tipo e possono essere valutate dando come risultato un valore.

Funzioni (OCaml vs JS)

Lo stesso modo "base" di definire funzioni è presente anche in *JavaScript*, usando il costrutto `function()` per definire una *funzione anonima*

Abbiamo che $\lambda x.x+1$ in JavaScript diventa (attenzione alle parentesi):

```
JAVASCRIPT: (function(x) { return(x+1) })
```

mentre $(\lambda x.x+1)10$ in JavaScript diventa (attenzione alle parentesi):

```
JAVASCRIPT: (function(x) { return(x+1) }) (10)
```

ANCHE JAVASCRIPT E' NATO COME LINGUAGGIO FUNZIONALE!

Un altro modo "base" per definire funzioni anonime in *JavaScript*

- tramite le *arrow expressions*:

$\lambda x.x+1$ corrisponde a

```
JAVASCRIPT: x => x + 1
```

$(\lambda x.x+1)10$ corrisponde a

```
JAVASCRIPT: (x => x + 1) (10)
```

E' CHIARO CHE QUESTE OPERAZIONI SONO ISPIRATE AL λ-CALCOLO...

Lo stesso accade in molti altri linguaggi "non funzionali". In Java, ad esempio, da qualche anno sono state introdotte le cosiddette "lambda expressions", che sono un'altra variante dello stesso concetto preso dal λ-calcolo.

Funzioni: altri esempi dal λ-calcolo

Continuando a tenere il λ-calcolo come riferimento, possiamo vedere qualche altro esempio riformulato in OCaml

$\lambda x.\lambda y.x+y$

```
[24]: fun x -> fun y -> x+y ;;
```

```
[24]: - : int -> int -> int = <fun>
```

$\lambda f.\lambda n.fn+1$

```
[25]: fun f -> fun n -> f n + 1 ;;
```

```
[25]: - : ('a -> int) -> 'a -> int = <fun>
```

Un primo assaggio di type inference

Questi esempi ci consentono di fare qualche osservazione su come sia definito il tipo delle funzioni.

Nel primo esempio, `fun x -> fun y -> x+y` è una funzione che prende `x` e restituisce la funzione `fun y -> x+y`. Quest'ultima è una funzione che prende `y` e restituisce il risultato di `x+y`. Essendo `x` e `y` sommati usando l'operatore `+`, che accetta solo operandi di tipo `int`, il compilatore inferisce che `x` e `y` debbano avere tipo `int`, quindi `fun y -> x+y` deve avere tipo `int -> int` (prende un intero `y` e restituisce un intero ottenuto da `x+y`). Una volta inferito il tipo di `x` e di `fun y -> x+y`, il compilatore può ora inferire anche il tipo dell'intera funzione `fun x -> fun y -> x+y` come segue: `int -> (int -> int)` (la funzione prende un intero `x` e restituisce una funzione da `int` a `int`). Dal momento che l'operatore `->` è associativo a destra, le parentesi nel tipo possono essere rimosse, ottenendo quindi `int -> int -> int`.

Nel secondo esempio, la funzione `fun f -> fun n -> f n + n` viene esaminata in modo analogo alla precedente. Questa volta però il "corpo" della funzione `f n + n` consente di dedurre che `n` è un `int` (perchè e operando di `+`), mentre `f` è una funzione, in quanto viene applicata ad `n` (in `f n`). Il fatto che `f` sia applicata al valore intero `n` e il fatto che il risultato di `f n` sia poi usato come operando di `+` consentono di inferire il tipo di `f` come `int -> int`. A questo punto, il tipo della funzione `fun f -> fun n -> f n + n` risulta essere `(int -> int) -> int -> int`. In questo caso le parentesi sono necessarie in quanto il primo `int -> int` descrive il tipo del parametro `f`.

Ancora un esempio

Vediamo ancora un esempio, sempre tenendo il λ-calcolo come riferimento

$(\lambda f.\lambda n.fn + 1)(\lambda x.2x)10$

[26]:
```
(fun f -> fun n -> f n + 1)(fun x -> 2 * x) 10 ;;
```

[26]:
```
- : int = 21
```

Questo è un esempio di funzione *higher-order* (di ordine superiore), ossia una funzione che prende un'altra funzione (`f`) come parametro e la usa nel proprio corpo

- il valore calcolato da OCaml è lo stesso che si ottiene valutando l'espressione del λ-calcolo tramite la *β-riduzione*

Richiamiamo innanzitutto le regole della β-riduzione:

$$(\lambda x.e_1)e_2 \to e_1[x/e_2] \qquad \frac{e_1 \to e'}{e_1 e_2 \to e' e_2} \qquad \frac{e_2 \to e'}{e_1 e_2 \to e_1 e'} \qquad \frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

Ne segue che l'espressione $(\lambda f.\lambda n.fn + 1)(\lambda x.2x)10$ abbia la seguente semantica:

$$\begin{array}{l} (\lambda f.\lambda n.fn + 1)(\lambda x.2x)10 \quad \text{ricordando che } e1\, e2\, e3 = (e1\, e2)\, e3 \\ \to (\lambda n.(\lambda x.2x)n + 1)10 \\ \to (\lambda n.2n + 1)10 \\ \to 20 + 1 = 21 \end{array}$$

E si ottiene esattamente il risultato calcolato da OCaml.

Funzioni con più parametri

Il costrutto `fun` può prevedere più di un parametro

[27]:
```
(fun x y -> x+y) 3 4 ;; (* equivale a (fun x -> fun y -> x+y) 3 4 *)
```

```
[27]: - : int = 7
```

Questo però è *zucchero sintattico*:

- è solo una semplificazione che evita di scrivere una sequenza di `fun`

In generale, con il termine *zucchero sintattico* (o syntactic sugar) si intende dire che un certo costrutto è stato previsto in un linguaggio di programmazione solo per semplificare la scrittura di certe operazioni. In realtà, le stesse operazioni possono essere scritte usando anche altri costrutti del linguaggio ottenendo comunque lo stesso identico risultato.

Ad esempio, molti linguaggi di programmazione (come ad esempio C, Java o Javascript) prevedono un costrutto di assegnamento con incremento += che può essere usato in questo modo:

```
JAVASCRIPT: X += 1
```

ma che è solo una semplificazione sintattica dell'operazione di incremento

```
JAVASCRIPT: X = X + 1
```

Nel caso del costrutto `fun` di OCaml abbiamo che

```
(fun x y -> x+y)
```

è zucchero sintattico per

```
(fun x -> fun y -> x+y)
```

1.8 Il λ-calcolo in OCaml (sintatticamente)

La codifica *sintattica* del λ-calcolo in OCaml si può formalizzare così:

Definizione [Encoding λ-calcolo in OCaml]. Dato un qualunque termine del λ-calcolo T, la corrispondente espressione in OCaml è $[\![T]\!]$, dove $[\![\ldots]\!]$ è la funzione definita ricorsivamente sulla struttura dei λ-termini come segue:

$$
\begin{array}{rcl}
[\![n]\!] & = & n \\
[\![x]\!] & = & x \\
[\![T' \, op \, T'']\!] & = & [\![T']\!] \, op \, [\![T'']\!] \\
[\![\lambda x.T']\!] & = & \text{fun } x \text{ -> } [\![T]\!] \\
[\![T' \, T'']\!] & = & [\![T']\!] \, [\![T'']\!]
\end{array}
$$

Dove n è un qualunque numero intero, x è un qualunque identificatore (nome) e op è un operatore artimetico $(+, -, *, /, ...)$

Stiamo considerando un λ-calcolo che consente anche di includere espressioni su numeri interi. La definizione dell'encoding quindi segue la definizione sintattica di questa variante del formalismo.

La funzione $[\![...]\!]$ ha come dominio l'insieme delle espressioni (o termini) del λ-calcolo e come codominio le espressioni OCaml. E' una funzione iniettiva (qualunque espressione del λ-calcolo può essere tradotta) ma non suriettiva (OCaml ha una sintassi molto più ricca del λ-calcolo).

In realtà $[\![...]\!]$ è una funzione molto semplice, che traduce letteralmente l'espressione del λ-calcolo trasformando i vari $\lambda x.$ in `fun x ->` e lasciando tutto il resto sostanzialmente inalterato.

E' importante notare che la funzione $[\![...]\!]$ è definita *ricorsivamente sulla struttura dei λ-termini*. Questo significa che nella defnizione di $[\![...]\!]$ abbiamo un caso per ogni termine di base (n e x) e un caso per ogni operazione ($T' \, op \, T''$, $\lambda x.T'$ e $T' \, T''$. Nei casi che corrispondono a termini di base la funzione fornisce direttamente un risultato (casi base della ricorsione), mentre nei casi corrispondenti a operazioni la funzione viene richiamata ricorsivamente sugli operandi di tale operazione (ad esempio, T' e T'' nel caso di $T' \, op \, T''$). Quindi la ricorsione in questo caso è usata per definire il risultato della funzione $[\![...]\!]$ su una certa espressione sulla base del risultato della stessa funzione $[\![...]\!]$ applicata a sottoespressioni.

Il λ-calcolo in OCaml: esempi

Tanto per vedere che la traduzione è effettivamente banale. . .

$[\![\lambda x.\lambda y.x+y]\!] =$
fun x -> $[\![\lambda y.x+y]\!] =$
fun x -> fun y -> $[\![x+y]\!] =$
fun x -> fun y -> $[\![x]\!]$ + $[\![y]\!] =$
fun x -> fun y -> x + $[\![y]\!] =$
fun x -> fun y -> x + y

$[\![\lambda f.\lambda n.fn+1]\!] =$
fun f -> $[\![\lambda n.fn+1]\!] =$
fun f -> fun n -> $[\![fn+1]\!] =$
fun f -> fun n -> $[\![fn]\!]$ + $[\![1]\!] =$
fun f -> fun n -> $[\![f]\!][\![n]\!]$ + $[\![1]\!] =$
fun f -> fun n -> f n + 1

$[\![(\lambda f.\lambda n.fn+1)(\lambda x.2x)10]\!] =$
$[\![(\lambda f.\lambda n.fn+1)(\lambda x.2x)]\!][\![10]\!] =$
$[\![(\lambda f.\lambda n.fn+1)(\lambda x.2x)]\!]10 =$
$[\![(\lambda f.\lambda n.fn+1)]\!][\![(\lambda x.2x)]\!]10 =$
$[\![(\lambda f.\lambda n.fn+1)]\!][\![(\lambda x.2x]\!]10 =$
$[\![(\lambda f.\lambda n.fn+1)]\!]$(fun x ->$[\![2x]\!]$) 10 $=$
$[\![(\lambda f.\lambda n.fn+1)]\!]$(fun x -> 2x) 10 $=$
(fun f ->$[\![\lambda n.fn+1]\!]$) (fun x -> 2x) 10 $=$
(fun f -> fun n ->$[\![fn+1]\!]$) (fun x -> 2x) 10 $=$
(fun f -> fun n ->$[\![fn]\!]+[\![1]\!]$) (fun x -> 2x) 10 $=$
(fun f -> fun n ->$[\![fn]\!]$+ 1) (fun x -> 2x) 10 $=$
(fun f -> fun n ->$[\![f]\!][\![n]\!]$+ 1) (fun x -> 2x) 10 $=$
(fun f -> fun n ->$[\![f]\!]$n + 1) (fun x -> 2x) 10 $=$
(fun f -> fun n -> f n + 1) (fun x -> 2x) 10

1.9 Altro modo (più comune) di definire funzioni: `let`

Le funzioni sono valori, e possono essere usate in una dichiarazione con `let`

[28]:
```
let somma = fun x y -> x+y ;;
somma 3 4 ;;
```

[28]:
```
val somma : int -> int -> int = <fun>
```

[28]:
```
- : int = 7
```

Più semplicemente, si può evitare di scrivere `fun` e `->` (zucchero sintattico)

[29]:
```
let somma x y = x+y ;;
somma 3 4;;
```

[29]:
```
val somma : int -> int -> int = <fun>
```

[29]:
```
- : int = 7
```

QUESTO E' IL MODO PIU' COMUNE DI DEFINIRE FUNZIONI!

Definire funzioni con `function`

Esiste in realtà un altro modo ancora di definire funzioni, che utilizza, al posto del costrutto `fun`, il costrutto `function`

[30]:
```
function x -> x + 1 ;;
```

[30]:
```
- : int -> int = <fun>
```

A differenza di `fun`, le funzioni definite con `function` possono prevedere un unico parametro. A questo livello il costrutto `function` non risultata particolarmente utile. Quando parleremo di *pattern matching* vedremo che questo costrutto può semplificare leggermente la scrittura di alcuni tipi di funzioni.

Nota: non esiste il `return`

Il corpo di una funzione in OCaml è un'*espressione*

[31]:
```
let media x y =
    x +. y /. 2.0
```

[31]:
```
val media : float -> float -> float = <fun>
```

Al momento della chiamata, l'intero corpo della funzione (l'espressione `base *. altezza /. 2.0`) viene valutato e il valore ottenuto come risultato sarà il risultato della funzione

Il *comando* `return` ha senso nella programmazione imperativa, in cui il corpo di una funzione è una sequenza di comandi

- in quel contesto `return` interrompe l'esecuzione e fornisce l'espressione da valutare nello stato raggiunto per determinare il risultato della funzione

Ad esempio, in JavaScript:

```
function area_triangolo ( x , y ) {
    let somma = x + y;
    return somma / 2;
}
```

il comando `return` interrompe la funzione e fa restituire il risultato dell'espressione `somma / 2` valutata nello stato in cui la variabile `somma` è già stata assegnata

1.10 "Scope" delle dichiarazioni e funzioni ausiliarie

Una dichiarazione di variabile aggiorna l'ambiente globale gestito dal toplevel di OCaml e diventa utilizzabile da tutte le espressioni successive

[32]:
```
let x = 10 ;;
let y = 5 * x ;;
```

[32]:
```
val x : int = 10
```

[32]:
```
val y : int = 50
```

Il costrutto `let ... in` dichiara variabili il cui *scope* è una sola espressione

[33]:
```
let z = 10 in z * z / 2 ;; (* z è utilizzabile solo qui... *)
let w = 5 * z ;;      (* mentre qui no! *)
```

[33]:
```
- : int = 50
```

```
File "[33]", line 2, characters 12-13:
2 | let w = 5 * z ;;      (* mentre qui no! *)
                ^

Error: Unbound value z
```

Le dichiarazioni di variabili con scope limitato (variabili locali) è ottenuto in molti altri linguaggi (come JavaScript, C, Java) creando blocchi di comandi `{ ... }` e dichiarando la variabile all'interno del blocco. Il costrutto `let ... in` (dall'inglese, "sia … in") sottolinea il fatto che lo scope è un'unica espressione (quella dopo `in`) e inoltre conferisce al programma un "sapore" più matematico richiamando il fatto che queste variabili sono in realtà molto simili alle incognite di una espressione algebrica… *Sia x uguale a 10 nell'espressione $x^2/2$….*

Questo meccanismo può essere usato anche per definire *funzioni ausiliarie*

[34]:
```
let f x = 2*x
in
    f (f 10) ;;
```

[34]:
```
- : int = 40
```

Esempio: soluzione equazioni di secondo grado

- $ax^2 + bx + c = 0$ con $a, b, c \in \mathbb{R}$
- formula risolutiva: $\frac{-b \pm \sqrt{\Delta}}{2a}$ dove $\Delta = b^2 - 4ac$

[78]:
```
let sol a b c =
    let delta =
        b*.b -. 4.*.a*.c
    in
        ( (-.b +. sqrt delta) /. (2.*.a) , (-.b -. sqrt delta) /. (2.*.a) ) ;;
```

[78]:
```
val sol : float -> float -> float -> float * float = <fun>
```

[36]:
```
sol 1. (-5.) 6. ;;
```

[36]:
```
- : float * float = (3., 2.)
```

La funzione `sol` prende i coefficienti a, b e c dell'equazione e restituisce una coppia con le due soluzioni

- il discriminante Δ viene calcolato una volta sola come variabile locale `delta`

Nella funzione `sol` viene utilizzata la funzione `sqrt` è che predefinita in OCaml e restituisce la radice quadrata di un numero (di tipo `float`).

Altra versione più sofisticata…

[37]:
```
let sol a b c =
    let delta =
        b*.b -. 4.*.a*.c
```

```
    in
    let sol' op =
        ( op (-.b) (sqrt delta) ) /. (2.*.a)
    in
        (sol' (+.) , sol' (-.)) ;;
```

[37]: `val sol : float -> float -> float -> float * float = <fun>`

[38]:
```
sol 1. (-.5.) 6. ;;
```

[38]: `- : float * float = (3., 2.)`

La funzione ausiliaria `sol'` viene usata per calcolare entrambe le soluzioni:

- il parametro `op` è l'operazione (somma o sottrazione) da usare nei due casi
- `(+.)` e `(-.)` sono le funzioni somma e sottrazione in forma *prefissa*

[39]:
```
(+.) 3.5 2.0 ;;
```

[39]: `- : float = 5.5`

Si usa la terminologia "forma prefissa" per indicare un operatore il cui simbolo viene messo a sinstra di entrambi (se non due) gli operadi, mentre si dice "forma infissa" per indicare un operatore il cui simbolo viene messo in mezzo agli operandi. Per gli operatori aritmetici di OCaml si possono usare entrambe le forme. Quando sono usati in forma prefissa è necessario racchiuderli tra parentesi (altrimenti il compilatore non lo capisce e assume siano usati in forma infissa).

Questa versione della funzione `sol` sfrutta la possibilità di definire funzioni higher order per evitare di scrivere due volte una espressione (per calcolare una delle due soluzioni) che deve essere ripetuta più volte con una piccola variante di esecuzione tra una volta e l'altra. Visto che le due soluzioni si calcolano nella stessa maniera se non per una operazione che una volta è una somma e l'altra una sottrazione, è possibile rendere l'espressione parametrica rispetto a quella singola operazione.

1.11 Funzioni *Curryed*

Il nome della funzione e i vari parametri sono separati da spazi, senza parentesi e virgole (diversamente da molti altri linguaggi)

[79]:
```
let somma x y = x+y ;;
somma 3 4 ;;
```

[79]: `val somma : int -> int -> int = <fun>`

[79]: `- : int = 7`

Le funzioni definite usando la notazione con gli spazi si dicono Curryed

- dal nome del matematico *Haskell Curry*, uno dei padri della programmazione funzionale

In realtà, nulla vieta di usare parentesi e virgole:

[41]:
```
let somma (x, y) = x+y ;;
somma (3,4);;
```

[41]:
```
val somma : int * int -> int = <fun>
```

[41]:
```
- : int = 7
```

ma così stiamo scrivendo una funzione con un unico parametro di tipo `int * int` (una coppia)

- alla chiamata si passa una coppia alla funzione
- la coppia viene *destrutturata* nelle variabili `x` e `y`

La notazione Curryed (cioè senza virgole) è quella comunemente usata per le funzioni in OCaml

Anche la notazione Curryed è in realtà un richiamo al λ-calcolo, in cui l'applicazione di funzione si scrive $f\,x$ senza parentesi ne virgole.

1.12 Applicazione parziale di funzione

La notazione Curryed consente di applicare una funzione *parzialmente*, passandole solo una parte dei parametri

[42]:
```
let somma x y = x+y ;;
```

[42]:
```
val somma : int -> int -> int = <fun>
```

Ad esempio, passiamo un solo argomento a `somma`:

[43]:
```
somma 10;;
```

[43]:
```
- : int -> int = <fun>
```

Il risultato è una funzione di tipo `int -> int`...

Per capire meglio, ragioniamoci con il λ-calcolo.

La funzione `somma` corrisponde a

$$\lambda x.\lambda y.x + y$$

L'applicazione parziale `somma 10` corrisponde a

$$(\lambda x.\lambda y.x + y)10$$

Applicando la semantica del λ-calcolo otteniamo

$$(\lambda x.\lambda y.x + y)10 = \lambda y.10 + y$$

Quindi il risultato è una funzione che prende un intero `y` e lo somma a `10`!

Torniamo in OCaml.

Nulla proibisce di scrivere:

[44]:
```
let somma_dieci = somma 10 ;;
```

[44]:
```
val somma_dieci : int -> int = <fun>
```

A questo punto possiamo usare (quante volte vogliamo) la funzione `somma_dieci` passandole il (secondo) valore da sommare

[45]:
```
somma_dieci 5;;
```

[45]:
```
- : int = 15
```

[46]:
```
somma_dieci 10;;
```

[46]:
```
- : int = 20
```

L'applicazione parziale di funzione è un meccanismo molto potente tipico dei linguaggi di programmazione funzionali. Consente di ottenere infinte funzioni diverse a partire da un'unica definizione. Riferendosi all'esempio della funzione `somma`, è possibile ottenere infinite funzioni diverse (`somma_uno`,`somma_cinque`,`somma_cento`,. . .) passando un diverso (singolo) parametro alla stessa funzione.

Questo meccanismo non va confuso con la possibilità (presente in molti linguaggi) di definire funzioni con parametri opzionali, o funzioni con parametri a cui è associato un valore di default. In questi casi, infatti, la funzione che applichiamo è sempre la stessa, ma avremo semplicemente la possibilità di evitare di scrivere qualche parametro il cui valore non è particolarmente rilevante.

1.13 Controllo del flusso di esecuzione

La maggior parte dei linguaggi di programmazione ha come costrutti fondamentali per il controllo del flusso di esecuzione del programma:

- un *comando* di scelta condizionale (ad es. `if`)
- un *comando* di iterazione (ad es. `while`)

Questi *comandi* sono legati al paradigma imperativo, la cui operazione fondamentale è *l'assegnamento*

- un `if` consente di scegliere, ad esempio, tra due assegnamenti
- un `while` consente di ripetere un assegnamento più volte

Questi costrutti hanno senso nel paradigma imperativo, in cui si assume uno stato del programma che viene aggiornato passo-passo

Controllo del flusso di esecuzione in OCaml

In OCaml (nel suo cuore funzionale) i costrutti di base per il controllo del flusso sono:

- una *espressione* di scelta condizonale (`if`)
- la RICORSIONE

Questo perchè non esistono operazioni di assegnamento: le variabili, una volta dichiarate, sono *immutabili*, ossia non possono più cambiare valore

- le variabili hanno un ruolo simile a quello delle incognite nelle espressioni algebriche (es: $y = 3x^2 + 2x - 1$)

Non potendo ri-assegnare variabili, non ha senso eseguire dei cicli

- Si può ripetere un calcolo chiamando più volte la stessa funzione (= ricorsione)

1.14 Espressione di scelta condizionale `if`

In OCaml `if` è un'espressione, e deve sempre prevedere due rami: `then` e `else`

- i due rami dovranno contenere a loro volta *espressioni dello stesso tipo*

- l'`if` potrà essere usato "a destra" di una dichiarazione

[47]:
```
let x = 23 ;;
let y = if x>10 then 1 else 0 ;;
```

[47]:
```
val x : int = 23
```

[47]:
```
val y : int = 1
```

L'`if` in OCaml assomiglia all'operatore di scelta condizionale (`_ ? _ : _`) presente in molti linguaggi di programmazione (quali JavaScript, C e Java):

```
JAVASCRIPT: var y = ( x>10 ? 1 : -1 )
```

E' bene sottolineare di nuovo che i rami `then` e `else` devono sempre essere presenti entrambi, e devono contenere espressioni dello stesso tipo. Se così non fosse, si potrebbe scrivere, ad esempio, la seguente funzione:

```
let f x =
    if x>0 then x
           else "errore"
```

e OCaml non sarebbe in grado di inferirne il tipo (`int -> ???`) in quanto la funzione restituisce a volte l'intero `x` (che è per certo di tipo `int` visto che abbiamo fatto `x>0`) e a volte la stringa `"errore"`.

Un discorso simile vale per la necessità di avere sia il ramo `then` che `else`. Se nell'`if` della funzione `f`, ad esempio, mancasse il ramo `else` la funzione non sarebbe completamente definita... Che cosa restituisce se `x` non è maggiore di zero? Va tenuto conto che non è possibile inserire altre espressioni *dopo* l'`if` (cioè non si può scrivere, prendendo come esempio JavaScript, `if (x>9) return x; ... altro ...`) in quanto in OCaml il corpo della funzione deve essere una unica espressione. Quindi, affinché la funzione sia totale è necessario che gli `if` siano completi.

1.15 Ricorsione

Per definire funzioni ricorsive è necessario utilizzare il costrutto `let rec`

[48]:
```
let fact n =
    if n<=0 then 1 else n * fact (n-1) ;;
```

```
File "[48]", line 2, characters 28-32:
2 |     if n<=0 then 1 else n * fact (n-1) ;;
                                ^^^^
Error: Unbound value fact
Hint: If this is a recursive definition,
you should add the 'rec' keyword on line 1
```

[49]:
```
let rec fact n =
    if n<=0 then 1 else n * fact (n-1);;
```

[49]:
```
val fact : int -> int = <fun>
```

Il fatto che i costrutti principali per il controllo del flusso siano `if` e ricorsione rende le definizioni di funzione in OCaml concettualmente molto simili a come le stesse funzioni sarebbero definite in termini matematici. Ad esempio, la funzione che calcola il fattoriale, in termini matematici si scrive

$$fact(n) = \begin{cases} 1 & \text{se } n \leq 0 \\ n \cdot fact(n-1) & \text{altrimenti} \end{cases}$$

che sostanzialemnte prevede una scelta tra due casi e la definizione in termini di se stessa (ricorsione) in uno dei due (come nella versione OCaml).

Perché è necessario aggiungere `rec` ?

Abbiamo visto che la definizione di funzione tramite `let`:

`let f x = x + 1` è in realtà una abbreviazione di `let f = fun x -> x+1`

Quindi l'espressione che definisce la funzione `fun x -> x + 1` non ha visibilità del nome `f` (che è in corso di dichiarazione)

Chiaramente questo crea problemi se la funzione è ricorsiva...

- e `rec` esplicita il fatto che la funzione dovrà essere ricorsiva.

Un paio di esempi di funzioni ricorsive

Successione di Fibonacci (restituisce l'n-esimo valore della successione):

[50]:
```
let rec fibonacci n =
    if n=0 || n=1 then n
    else fibonacci (n-1) + fibonacci (n-2);;
```

[50]:
```
val fibonacci : int -> int = <fun>
```

Massimo Comune Divisore con il metodo di Euclide:

[51]:
```
let rec mcd n m =
    if m=0 then n else mcd m (n mod m);;
```

[51]:
```
val mcd : int -> int -> int = <fun>
```

Vedremo esempi più interessanti quanto introdurremo strutture dati e pattern matching...

Funzioni mutuamente ricorsive (con `and`)

Ogni funzione può richiamare solo funzioni definte precedentemente.

Funzioni *mutuamente ricorsive* vanno definite una dopo l'altra separate da `and`

[1]:
```
let rec pari n =
  match n with
    | 0 -> true
    | x -> dispari (x-1)

and dispari n =
  match n with
    | 0 -> false
    | x -> pari (x-1) ;;
```

[1]:
```
val pari : int -> bool = <fun>
val dispari : int -> bool = <fun>
```

[2]:
```
pari 6;;
dispari 6;;
```

[2]:
```
- : bool = true
```

[2]:
```
- : bool = false
```

1.16 Type Inference

Abbiamo già visto che il toplevel di OCaml ci fornisce il tipo di ogni espressione che inseriamo

[52]:
```
let raggio = 5.;;
let area_cerchio = raggio**2. *. 3.14 ;;
let testo = "l'area del cerchio è " ^ string_of_float area_cerchio ;;
```

[52]:
```
val raggio : float = 5.
```

[52]:
```
val area_cerchio : float = 78.5
```

[52]:
```
val testo : string = "l'area del cerchio è 78.5"
```

Il tipo di una espressione viene inferito analizzando:

- i valori inseriti
- gli operatori usati

I valori (letterali) presenti hanno un tipo ben definito, e anche gli operatori hanno dei tipi specifici a cui possono essere applicati (ad esempio, `+` agli `int` e `+.` ai `float`). A partire da queste certezze, il compilatore di OCaml può dedurre il tipo delle variabili utilizzate nell'espressione (ad esempio, in `x+y` le variabili `x` e `y` devono necessariamente avere tipo `int` perché sono operandi di un `+`).

Esaminando l'espressione partendo dai singoli valori e dalle singole variabili e facendo dei passi di deduzione che riguardano via via sottoespressioni sempre più ampie (che corrisponde a "risalire" l'albero di sintassi astratta dell'espressione) il compilatore procede creandosi un elenco di associazioni "variabile -> tipo" che si porta dietro durante l'analisi dell'espressione.

Può accadere che analizzando sottoespressioni diverse il compilatore incontri due volte la stessa variabile. Quando, risalendo nell'albero di sintassi astratta, arriverà a considerare l'espressione completa, dovrà confrontare i tipi delle due istanze della stessa variabile incontrate. Se questi due tipi sono uguali (entrambi `int`) o *unificabili* (uno più generico dell'altro... vedremo più avanti) il compilatore può procedere prendendo il tipo meno generico dei due. Se invece i due tipi sono completamente diversi (come `int` e `float`) allora il compilatore dovrà segnalare un errore di tipo.

Ad esempio, l'espressione `(x+1)-(x+4)` supera il controllo di tipi, perchè il compilatore inferisce il tipo `int` per entrambe le istanze di `x`. Invece l'espressione `(x+1)-(x+.4.0)` non supera il controllo in quanto la prima istanza di `x` viene associata al tipo `int` mentre la seconda al tipo `float` (e anche perchè il `-` viene applicato a un secondo operando di tipo `float`).

Type Inference e Funzioni

Nel caso di espressioni che definiscano (o utilizzino) funzioni, inferire il tipo è un po' più complicato, anche se l'approccio rimane lo stesso

Si analizzano:

- i valori inseriti
- gli operatori usati
- gli argomenti/valori di ritorno delle funzioni

[53]:
```
let f x = x+1 ;;
```

[53]:
```
val f : int -> int = <fun>
```

[80]:
```
let y = f 10 ;;
```

[80]:
```
val y : int = 11
```

Vediamo un esempio un po' più articolato:

[55]:
```
let sum_if_true test first second =
    (if test first then first else 0)
    + (if test second then second else 0);;
```

[55]:
```
val sum_if_true : (int -> bool) -> int -> int -> int = <fun>
```

Il tipo inferito è `(int -> bool) -> int -> int -> int` poiché:

- i due rami dell'`if` devono avere lo stesso tipo (quindi `first` e `second` sono `int` come `0`)
- essendo usata nella guardia di un `if` la funzione `test` deve restituire un `bool`
- il risultato di + è sempre di tipo `int`

Questa è una funzione *higher order* che prende una funzione `test` che effettua un controllo sui due parametri `first` e `second` e calcola una somma. I parametri `first` e `second` prendono parte alla somma solo se superano il controllo operato dalla funzione `test` (ossia solo se il risultato di `test` è `true`) altrimenti venono sostituiti da `0`.

Una funzione che, come `test`, restituisce `true` o `false` come risultato di un controllo svolto sui suoi parametri viene comunemente detta *predicato*. Quindi, in questo esempio possiamo dire che `test` è un predicato che effettua un controllo su `first` e `second`.

Il fatto che `sum_if_true` preveda il predicato `test` come parametro fa sì che tale funzione possa essere utilizzata con predicati diversi a seconda delle necessità. Passando funzioni opportune, si potrà usare `sum_if_true` per sommare solo valori pari, oppure solo valori positivi, o altro...

Vediamo qualche esempio di uso di `sum_if_true`:

[56]:
```
sum_if_true (fun x -> x mod 2=0) 3 4;; (* somma se pari -- mod calcola il modulo *)
```

[56]:
```
- : int = 4
```

[57]:
```
sum_if_true (fun x -> x>0) 3 (-4);; (* somma se positivo *)
```

[57]:
```
- : int = 3
```

[58]:
```
let rec potenza_di_due x =
    if x = 1 then true
    else if x < 1 || x mod 2 = 1 then false
         else potenza_di_due (x/2);;

sum_if_true potenza_di_due 3 4;; (* somma se potenza di due *)
```

[58]:
```
val potenza_di_due : int -> bool = <fun>
```

[58]:
```
- : int = 4
```

L'ultimo esempio, `potenza_di_due`, mostra innazitutto che le funzioni passate come parametro non deve necessariamente essere anonime, ma possono essere qualunque funzione definibile in OCaml. Anche, come in questo caso, una funzione ricorsiva. Nell'esempio specifico, per verificare se un numero è potenza di due, lo dimezza ripetutamente andando ogni volta a vedere se è pari. Non appena si ottiene un numero dispari (`x mod 2 = 1`) o si arriva a zero, allora il numero non è una potenza di due, e si restituisce `false`. Se invece si raggiunge il valore uno sempre ottenendo numeri pari, allora il numero era una potenza di due.

Type inference e polimorfismo

In certi casi il tipo di una funzione non può essere identificato in modo *concreto* (ossia facendo riferimento ai tipi di base `int`,`float`,...)

[59]:
```
let id x = x ;; (* funzione identità *)
```

[59]:
```
val id : 'a -> 'a = <fun>
```

la funzione `id` è *polimorfa*: si può applicare ad argomenti di tipi diversi

- il risultato avrà per certo lo stesso tipo dell'argomento passato (qualunque sia)

Il costrutto `'a` è una *variabile di tipo*:

- indica che possono essere usati valori di qualunque tipo
- se presente più volte, i valori corrispondenti dovranno avere tutti lo stesso tipo

Qualche esempio d'uso:

[60]:
```
id 4 ;;
```

[60]:
```
- : int = 4
```

[61]:
```
id 'a' ;;
```

[61]:
```
- : char = 'a'
```

[62]:
```
id "ciao" ;;
```

[62]:
```
- : string = "ciao"
```

Altro esempio di funzione polimorfa

Restituisce il primo argomento se supera il test, altrimenti il secondo

[63]:
```
let first_if_true test first second =
    if test first then first else second ;;
```

[63]:
```
val first_if_true : ('a -> bool) -> 'a -> 'a -> 'a = <fun>
```

Qualche esempio d'uso:

[64]:
```
first_if_true (fun x -> x mod 2 = 0) 3 5;; (* controlla se pari *)
```

[64]:
```
- : int = 5
```

[65]:
```
let lettera c = (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') ;;

first_if_true lettera 'e' '-';; (* controlla se è lettera dell'alfabeto *)
```

[65]:
```
val lettera : char -> bool = <fun>
```

[65]:
```
- : char = 'e'
```

Ancora esempi di funzioni polimorfe

Funzione `maggiore`:

[66]:
```
let maggiore x y =
    if x > y then x else y ;;
```

[66]:
```
val maggiore : 'a -> 'a -> 'a = <fun>
```

[67]:
```
maggiore 6 5 ;;
```

[67]:
```
- : int = 6
```

[68]:
```
maggiore 3.5 9.4
```

[68]:
```
- : float = 9.4
```

[69]:
```
maggiore "albero" "gatto";;
```

[69]:
```
- : string = "gatto"
```

[70]:
```
maggiore true false ;;
```

[70]:
```
- : bool = true
```

Che succede se proviamo ad applicare `maggiore` a due funzioni?

[71]:
```
maggiore (fun x -> x) (fun x -> x+1) ;;
```

```
Exception: Invalid_argument "compare: functional value".
Raised by primitive operation at file "[66]", line 2, characters 7-12
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Viene sollevata un'*eccezione*

- in generale non è possibile confrontare funzioni (è un problema *indecidibile*)
- dal punto di vista dei tipi questo uso di maggiore è corretto, quindi questa espressione supera il controllo dei tipi a tempo di compilazione
- un'eccezione corrisponde a una situazione anomala che si verifica *a tempo di esecuzione*

Dal momento che nella funzione `maggiore` le variabili `x` e `y` non vengono mai usate se non per confrontarle fra loro con > (che può essere usato con qualunque tipo di valori), il loro tipo viene inferito come `'a` (ma deve essere lo stesso per entrambe). Quindi in generale la funzione `maggiore` supera il controllo dei tipi e può essere messa in esecuzione.

A tempo di esecuzione, però, l'operazione di confronto > solleva un'eccezione quando viene usata per confrontare funzioni. Questo perchè in generale **non è possibile confrontare funzioni**… Innanzitutto, non è chiaro che cosa possa significare che una funzione sia "maggiore" di un'altra. In secondo luogo, se anche provassimo a confrontare due funzioni per l'uguaglianza, otterremmo un'eccezione, come in questo esempio:

[72]:
```
(fun x -> x) = (fun y -> y);;
```

```
Exception: Invalid_argument "compare: functional value".
Raised by primitive operation at unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Questa impossibilità non è dovuta a una debolezza dell'interprete di OCaml, ma al fatto che determinare l'uguaglianza di due funzioni è un **problema indecidibile**. Dire che due funzioni sono uguali (o equivalenti) significa dire che per qualunque possibile combinazione di valori che passiamo alla prima, il risultato che otteniamo è sempre lo stesso che otterremmo passando gli stessi valori alla seconda. (In termini matematici, chiamando le due funzioni f e g, dovrebbe valere $\forall x. f(x) = g(x)$).

E' un risultato ben noto della teoria della calcolabilità, che se fosse possibile scrivere un algoritmo capace di determinare in generale se due funzioni sono equivalenti, allora sarebbe possibile sfruttarlo per risolvere il *problema della fermata*, che è il più classico tra gli esempi di problemi indecidibili.

Usando le *annotazioni di tipo* è possibile restringere (o eliminare) il polimorfismo:

[73]:
```
let maggiore (x : int) (y : int) : int =
    if x > y then x else y ;;
```

[73]:
```
val maggiore : int -> int -> int = <fun>
```

Così si previene che questa funzione venga usata per confrontare funzioni

E se volessimo usare la funzione `maggiore` solo su valori di tipo `int` o `string`?

Servono due funzioni distinte (con nomi diversi… *no overloading*):

[74]:
```
let maggiore_int (x : int) (y : int) : int =
    if x > y then x else y ;;
let maggiore_string (x : string) (y : string) : string =
    if x > y then x else y ;;
```

[74]:
```
val maggiore_int : int -> int -> int = <fun>
```

[74]:
```
val maggiore_string : string -> string -> string = <fun>
```

Le annotazioni di tipo, oltre ad essere utili in alcuni casi per rendere più comprensibile il codice, e oltre a far sì che il compilatore controlli che il tipo delle variabili e delle funzioni sia esattamente quello che che ci aspettiamo, consente anche di "restringere" il tipo inferito per una certa variabile.

In questi esempi, il compilatore inferirebbe tipo `'a` per `x` e `y`, ma con l'annotazione di tipo viene forzato a considerare un tipo più restrittivo (`int` e `string`, rispettivamente nei due casi).

Le due funzioni che andiamo a definire devono avere necessariamente nomi diversi. Non è infatti possibile definire (come accade ad esempio in C# e Java) funzioni che hanno lo stesso nome e che si distinguono solo per il numero o i tipi dei loro parametri (cioè, la loro firma). Vedremo che questa possibilità, presente soprattutto nei linguaggi *object-oriented*, prende il nome di *overloading* in quanto il nome della funzione viene "sovraccaricato" di significati (corrisponde a più di una implementazione).

Type Inference: OCaml vs TypeScript

Il linguaggio TypeScript ha un sistema di type inference con tipi polimorfi simile a quello di OCaml

Esistono tuttavia alcune differenze importanti…

Innanzitutto, i tipi di base sono diversi:

- In Ocaml: `int`,`float`,`bool`,…
- In TypeScript: `number`,`boolean`,`any`,`unknown`,…

In generale:

- OCaml necessita di inferire con precisione il tipo di ogni espressione
- TypeScript tende a essere *più permissivo* (vedi tipi `any` e `unknown`)

Ad esempio, la funzione identità:

[75]:
```
let id x = x ;;
```

[75]:
```
val id : 'a -> 'a = <fun>
```

In TypeScript si può scrivere usando `any`:

```
function id (x: any) : any {
    return x
}
```

però questo disabilita qualunque controllo di tipi sulla funzione `id`

- Usando `any` il compilatore accetterebbe l'espressione `12 * id("hello")`

Inoltre, la soluzione con `any` non tiene conto del fatto che il tipo del risultato è lo stesso di quello del parametro `x` (qualunque esso sia). In generale, agli occhi del compilatore di TypeScript la funzione `id` prende un parametro di qualunque tipo e restituisce un risultato di qualunque tipo (anche diverso dal precedente). In OCaml, invece, dal momento che il tipo inferito è `'a -> 'a`, il compilatore è consapevole che qualunque sia il tipo del parametro passato alla funzione, il risultato che si otterrà sarà dello stesso tipo.

In modo migliore, in TypeScript si possono usare i *Generics*:

```
function id<T> (x: T) : T {
    return x
}
```

in questo modo il compilatore può fare gli adeguati controlli sui tipi

D'altra parte:

- siamo stati costretti a *scrivere esplicitamente* la variabile di tipo `T` nel codice
- mentre OCaml è in grado di *inferire* le variabili di tipo (es. `'a`)

Un altro esempio: la funzione `maggiore` su `int` e `char`. In OCaml servono due funzioni diverse:

[76]:
```
let maggiore_int (x : int) (y : int) : int =
    if x > y then x else y ;;
let maggiore_string (x : string) (y : string) : string =
    if x > y then x else y ;;
```

[76]: `val maggiore_int : int -> int -> int = <fun>`

[76]: `val maggiore_string : string -> string -> string = <fun>`

In TypeScript possiamo scriverne una sola usando il *tipo unione* `number | string`

```
function maggiore ( x : number | string,
                    y : number | string ) : number | string {
    if (x>y) return x else return y
}
```

ma questo introduce qualche problema...

La funzione `maggiore` in TypeScript con il tipo unione `number | string`:

- può essere applicata ad un numero e una stringa:
  - `maggiore(3,"hello")`
- anche conoscendo gli argomenti passati alla funzione, il compilatore non può predire esattamente il tipo del risultato:
  - `maggiore(2,4)/2` causa un errore di tipo a tempo di compilazione

Tutto sommato forse è meglio fare due funzioni diverse...

Vedremo che anche OCaml prevede dei tipi unione, ma con un funzionamento diverso (e dei vincoli sintattici più forti) rispetto a quelli di TypeScript.

OCaml vs TypeScript: morale della favola

OCaml è più *rigoroso*:

- il suo sistema di tipi è concepito per poter determinare precisamente il tipo (anche polimorfo) di ogni espressione
- a tal fine non prevede cose come le converisioni di tipo implicite o tipi "ambigui" (come `any` o `number | string`), che semplificano la vita, ma possono indurre a errori di programmazione difficili da scovare

TypeScript, d'altra parte, cerca *compromessi*:

- per non introdurre troppe limitazioni rispetto a JavaScript, concede un po' di libertà al programmatore
- questo semplifica la programmazione, ma apre alla possibilità che il programmatore poco attento commetta qualche errore evitabile

Da questa discussione NON si vuole trarre la conclusione che il sistema di tipi di uno dei due linguaggi sia migliore rispetto a quello dell'altro. OCaml e TypeScript sono linguaggi concepiti con obiettivi profondamente diversi, ed ognuno fa un uso dei tipi appropriato per i propri scopi.

Capitolo 2

Liste e Pattern Matching

2.1 Liste

In OCaml le liste sono un tipo di dato predefinito

Lista = *sequenza finita e immutabile di valori dello stesso tipo*

[1]:
```
let numeri = [3; 5; -1; 9; 14; 21] ;;
```

[1]:
```
val numeri : int list = [3; 5; -1; 9; 14; 21]
```

Se gli elementi sono di tipo `XYZ`, il tipo della lista è `XYZ list`

Inoltre, la lista vuota si rappresenta semplicemente come `[]` e ha tipo generico:

[2]:
```
let lista_vuota = [] ;;
```

[2]:
```
val lista_vuota : 'a list = []
```

Da questi esempi è evidente che una lista si rappresenta come una sequenza di valori racchiusi tra parentesi quadre e separati da punto e virgola. Il fatto che sia una sequenza *immutabile* significa che non sarà possibile modificare (aggiungere, rimuovere o modificare) gli elementi della lista. Ogni operazione che vedremo sulle liste potrà leggere e processare gli elementi, e costuire nuove liste a partire da quelle esitenti, ma sempre senza possibilità di modificarle.

Le liste possono contenere elementi di qualunque tipo.

Una lista di stringhe:

[3]:
```
let stringhe = ["cane"; "gatto"; "rana"; "gnu"];;
```

[3]:
```
val stringhe : string list = ["cane"; "gatto"; "rana"; "gnu"]
```

Una lista di tuple (tutte dello stesso tipo):

[4]:
```
let tuple = [ (1,"lun"); (2,"mar"); (3,"mer") ] ;;
```

[4]:
```
val tuple : (int * string) list = [(1, "lun"); (2, "mar"); (3, "mer")]
```

Una lista di funzioni (tutte dello stesso tipo)

```
[5]: let funzioni = [ (fun x -> x+1) ; (fun x -> x-1) ; (fun x -> x) ] ;;
```

```
[5]: val funzioni : (int -> int) list = [<fun>; <fun>; <fun>]
```

Nell'ultimo esempio, in realtà, gli elementi della lista (funzioni) non hanno tutti lo stesso tipo. Mentre le prime due hanno tipo `int -> int`, la terza ha tipo `'a -> 'a`. Essendo però `'a -> 'a` compatibile con `int -> int` e più generico, i tipi delle tre funzioni possono essere unificati nel tipo più specifico `int -> int`. Approfondiremo questo aspetto tra poco.

Inoltre, è anche possibile creare liste di liste:

```
[6]: let liste = [ [1; 2; 3] ; [1; 3; 2] ; [2; 1; 3] ; [2; 3; 1] ; [3; 1; 2] ; [3; 2; 1]␣
     ↪] ;;
```

```
[6]: val liste : int list list =
       [[1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1]; [3; 1; 2]; [3; 2; 1]]
```

2.2 Riflessione sul tipo di una lista

OCaml inferisce il tipo di una lista *unificando* i tipi dei suoi elementi:

```
[7]: [] ;;
```

```
[7]: - : 'a list = []
```

```
[8]: [[]; []] ;;
```

```
[8]: - : 'a list list = [[]; []]
```

```
[9]: [[]; []; [3]] ;;
```

```
[9]: - : int list list = [[]; []; [3]]
```

```
[10]: [[]; []; ["ciao"]] ;;
```

```
[10]: - : string list list = [[]; []; ["ciao"]]
```

Anche in questi esempi, come in quello della lista di funzioni visto poco sopra, si vede il meccanismo di unificazione dei tipi in opera. Le liste che contengono solo liste vuote hanno tipo (generico) `'a list list`. Non appena alle liste vuote si aggiunge una lista non vuota (di `int` e `string`, negli esempi) il tipo dell'intera lista viene reso più specifico (diventando `int list list` e `string list list`, rispettivamente).

Quello che fa il l'unificazione dei tipi è trovare il *tipo più generico possibile* che sia compatibile con tutti gli elementi della lista. Fintanto che nella lista vengono inserite solo liste vuote `[]` il tipo può rimanere generico (in quanto una lista vuota può essere di `int`, `string`, o qualunque altro tipo). Nel momento in cui nella lista venga incluso una lista non vuota (es. `[3]`), allora il tipo più generico possibile diventa il tipo di quella specifica lista, in quanto essa non può avere altri tipi se non quello dato dagli elementi che contiene (ad es. `int list`).

Con le funzioni:

[11]:
```
let f x y = 3;;
```

[11]: `val f : 'a -> 'b -> int = <fun>`

[12]:
```
let g x y = x+1 ;;
```

[12]: `val g : int -> 'a -> int = <fun>`

[13]:
```
let h x y = y+1 ;;
```

[13]: `val h : 'a -> int -> int = <fun>`

[14]:
```
let lista = [f ; g];;
```

[14]: `val lista : (int -> 'a -> int) list = [<fun>; <fun>]`

[15]:
```
let lista2 = h::lista;;
```

[15]: `val lista2 : (int -> int -> int) list = [<fun>; <fun>; <fun>]`

Questi esempi con le funzioni consentono di osservare nuovamente il funzionamento dell'unificazione dei tipi. Le tre funzioni `f`, `g` e `h` hanno tipi generici diversi. La lista `lista` ha un tipo che è il più generico possibile che sia compatibile con i tipi di `f` e `g`, ossia `(int -> 'a -> int) list`. Aggiungendo alla lista `h` (in `lista2`) il tipo deve essere ulteriormente specializzato per tenere conto del fatto che `h` ha un secondo parametro di tipo `int`.

E' interessante vedere quale sia il tipo di una lista che contenga solo `g` e `h`:

[16]:
```
let lista3 = [g ; h] ;;
```

[16]: `val lista3 : (int -> int -> int) list = [<fun>; <fun>]`

Nonostante sia `g` che `h` abbiano un tipo polimorfo (generico), il fatto che `g` preveda un primo parametro di tipo `int` e `h` un secondo parametro di tipo `int` implica che la lista abbia tipo `int -> int -> int list`, cioè che si tratti di una lista di funzioni con due parametri *entrambi interi*.

Il tipo inferito da OCaml tramite il meccanismo di unificazione descritto da questi esempi, prende il nome di *most general unifier*, ossia (come già detto) è il tipo più generico possibile che tenga conto di tutti i vincoli imposti dai tipi che sono stati unificati.

2.3 L'operatore cons `::`

Una lista può essere *costruita* a partire da un'altra usando l'operatore `::` ("cons")

Definizione (`::`). Data una lista `l` di tipo `T list` e un elemento `e` di tipo `T`, si denota con `e :: l` la lista in cui il primo elemento è `e` seguito dagli elementi in `l`

[17]:
```
let l1 = [3;2;1] ;;
let l2 = 4 :: l1 ;;
```

[17]: `val l1 : int list = [3; 2; 1]`

```
[17]: val l2 : int list = [4; 3; 2; 1]
```

La notazione `[1; 2; 3; 4]` è in realtà "zucchero sintattico" per la notazione

```
[18]: let lis = 1 :: (2 :: (3 :: (4 :: [] ))) ;;
```

```
[18]: val lis : int list = [1; 2; 3; 4]
```

che può essere scritta più semplicemente così (essendo `::` associativo a destra):

```
[19]: let lis = 1 :: 2 :: 3 :: 4 :: [] ;;
```

```
[19]: val lis : int list = [1; 2; 3; 4]
```

Questa rappresentazione evidenza la natura *induttiva* (incrementale) delle liste

- a partire dalla lista vuota `[]`, si concatena in testa `4`, poi `3`, poi `2`, poi `1`

ATTENZIONE:

E' comune dire che `::` *aggiunga* un elemento in testa alla lista, ma non è esatto:

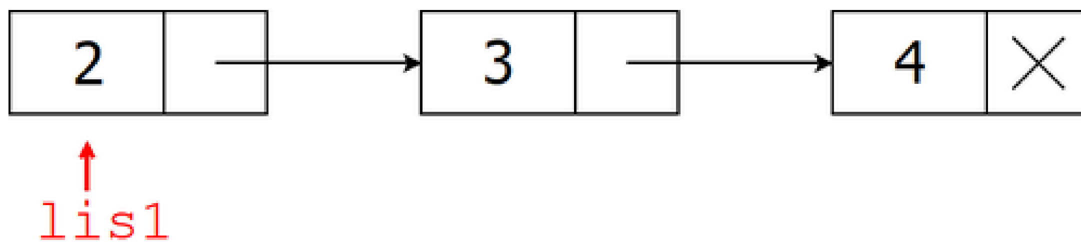
- Le liste sono immutabili, quindi `e :: l` concettualmente è una *lista diversa* con dentro `e` e gli elementi di `l`

Il fatto che le liste siano immutabili fa sì che OCaml non debba copiare gli elementi di `l` nella nuova lista

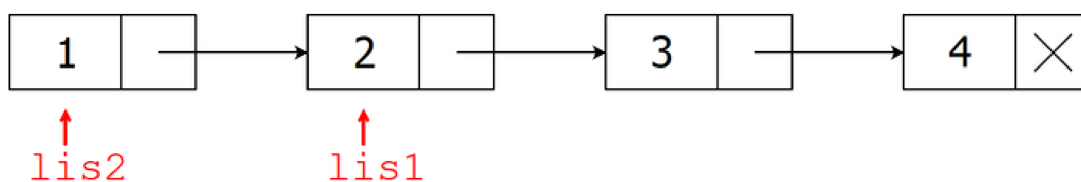
Vediamo perché...

Le liste in OCaml sono concepite come *liste concatenate singole*

La lista `lis1 = [2; 3; 4]` corrisponde a:



La lista `lis2 = 1 :: lis` può essere ottenuta concatenando in testa:



Essendo immutabili (= no modifiche ai valori) per il programmatore è come se fossero due liste diverse

- no spreco di memoria

2.4 Concatenazione di liste (append - `@`)

Date due liste `lis1` e `lis2`, l'operazione *append* `lis1 @ lis2` descrive la loro concatenazione in un'unica lista

[20]:
```
let lis1 = [1;2;3] ;;
let lis2 = [4;5;6] ;;
let lis3 = lis1 @ lis2 ;;
```

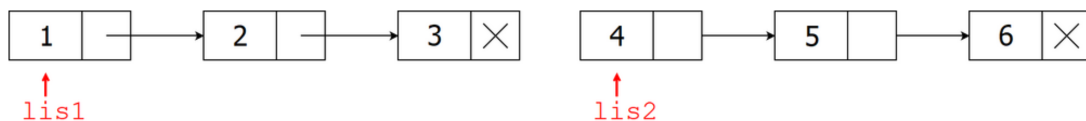
[20]:
```
val lis1 : int list = [1; 2; 3]
```

[20]:
```
val lis2 : int list = [4; 5; 6]
```

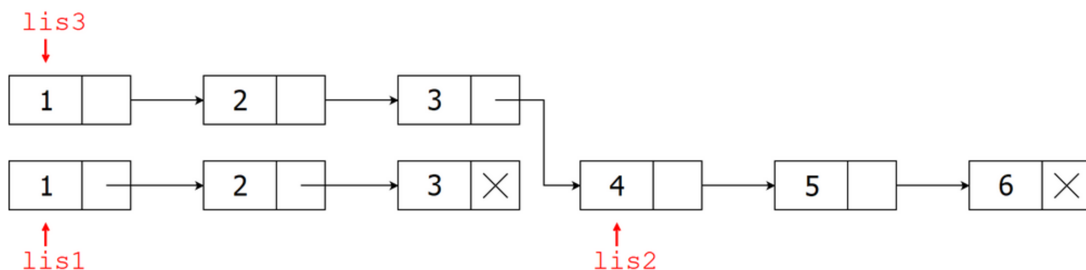
[20]:
```
val lis3 : int list = [1; 2; 3; 4; 5; 6]
```

Internamente, la concatenazione crea una copia della prima lista

Date `lis1 = [1; 2; 3]` e `lis2 = [4; 5; 6]`:



Ecco il risultato di `let lis3 = lis1 @ lis2`:



2.5 Altre operazioni su liste (OCaml API)

Il modulo `List` dell'OCaml API (https://ocaml.org/api/List.html) fornisce moltissime funzioni per l'elaborazione di liste. Ad esempio:

[21]:
```
List.length [5;2;1] ;; (* lunghezza della lista *)
```

[21]:
```
- : int = 3
```

[22]:
```
List.hd [5;2;1] ;; (* head - primo elemento della lista *)
List.tl [5;2;1] ;; (* tail - elementi successivi al primo *)
```

[22]:
```
- : int = 5
```

[22]:
```
- : int list = [2; 1]
```

```
[23]: List.rev [5;2;1] ;; (* rovescia la lista *)
```

```
[23]: - : int list = [1; 2; 5]
```

Interessante anche vedere il codice sorgente di tutte queste funzioni:

- https://github.com/ocaml/ocaml/blob/trunk/stdlib/list.ml

2.6 Pattern Matching

Per accedere agli elementi di una lista è necessaria un'operazione di *destrutturazione*

- OCaml ha un potente meccanismo di *Pattern Matching* (non solo per liste)

Sintassi:

```
match EXP with
| P_1 -> EXP_1
| P_2 -> EXP_2
...
| P_N -> EXP_N
```

Semantica informale:

- il risultato di `EXP` viene confrontato con i pattern `P_1, ..., P_n`
- se `P_i` è il primo pattern con cui *fa match*, si valuta l'espressione `EXP_i`

2.7 Sintassi dei pattern

Considerando i tipi visti fino ad ora (tipi di base, tuple e liste) la sintassi dei pattern è data da:

- valori di tipi base (non funzioni): `true`,`false`,0,1,2,2.3,4.5,`'a'`,`'b'`,`'c'`,`"abc"`
- variabili: `x`,`y`,`z`,...
- tuple: `(P_1,...,P_N)`
- liste di lunghezza fissata: `[P_1,...,P_N]`
- liste con *cons*: `P_1 :: P_2`
- wildcard: `_`

dove `P_1`,...`P_N` sono a loro volta dei pattern

Definizione (Match) Un valore `v` fa *match* con un pattern `P` se:

- `P=_`
- `P=v`
- esiste un modo di istanziare le variabili in `P` ottenendo `P'` tale che `P'=v`

Esempi:

Pattern	Fanno match	Non fanno match
1	1	0,2,3
(3,2)	(3,2)	(2,3),(4,2)
(x,y)	(3,2),(2,3),(4,2)	(1,3,2),(1,4,3,2)
(x,2)	(3,2),(4,2),("ciao",2)	(2,3), (4,2,1)
[]	[]	[3],[1;5],['a','b','c']
[3] o (3::[])	[3]	[],[1;5],['a','b','c']
3::x	[3],[3;4;5]	[],[1;5],['a','b','c']
x::y	[3],[3;4;5],[1;5],['a','b','c']	[]
_	5,true,"abc",(3,"hello"), [3;2;4], []	

Esempi con tipi di dato di base:

[24]:
```
let negazione b =
    match b with
    | true -> false
    | false -> true ;;
```

[24]: `val negazione : bool -> bool = <fun>`

[25]:
```
let rec fibonacci n =
    match n with
    | 0 -> 0
    | 1 -> 1
    | _ -> fibonacci (n-1) + fibonacci (n-2);;
```

[25]: `val fibonacci : int -> int = <fun>`

La wildcard _ messa in fondo cattura tutti gli altri casi

Qualche esempio con le tuple:

[26]:
```
let my_or x y =
    match (x,y) with
    | (false,false) -> false
    | _ -> true;;
```

[26]: `val my_or : bool -> bool -> bool = <fun>`

[27]:
```
let first_true t =
    match t with
    | (true,_,_) -> 1
    | (false,true,_) -> 2
    | (false,false,true) -> 3
    | (false,false,false) -> -1;;
```

[27]: `val first_true : bool * bool * bool -> int = <fun>`

Qualche esempio con le liste:

[28]:
```
let is_empty lis =
    match lis with
    | [] -> true
    | _ -> false
```

[28]: `val is_empty : 'a list -> bool = <fun>`

[29]:
```
let inizia_con_zero lis =
    match lis with
    | [] -> false
    | 0::lis' -> true
    | x::lis' -> false
```

[29]: val inizia_con_zero : int list -> bool = <fun>

[30]:
```
let lunghezza_uno lis =
    match lis with
    | [] -> false
    | x::[] -> true
    | x::lis' -> false
```

[30]: val lunghezza_uno : 'a list -> bool = <fun>

I casi `false` negli ultimi due esempi possono essere accorpati in _

E' opportuno sottolineare che nel pattern `x::lis'` avremo sempre che `x` è un elemento (il primo) della lista che si sta matchando, mentre `lis'` è la *lista* degli elementi che seguono. Quindi, nel caso di una lista di interi, `x` avrà tipo `int` mentre `lis'` avrà tipo `int list`.

2.8 Pattern matching e esaustività

E' opportuno (ma non obbligatorio) che i pattern siano *esaustivi*:

- per qualunque valore ci deve essere (almeno) un pattern con cui fa match

Esempio non esaustivo (warning a tempo di compilazione):

[31]:
```
let giorno n =
    match n with
    | 1 -> "lunedì"  | 2 -> "martedì"  | 3 -> "mercoledì"
    | 4 -> "giovedì" | 5 -> "venerdì"
    | 6 -> "sabato"  | 7 -> "domenica"
```

```
File "[31]", line 2, characters 4-156:
2 | ...match n with
3 |     | 1 -> "lunedì"  | 2 -> "martedì"  | 3 -> "mercoledì"
4 |     | 4 -> "giovedì" | 5 -> "venerdì"
5 |     | 6 -> "sabato"  | 7 -> "domenica"
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
0
```

[31]: val giorno : int -> string = <fun>

A tempo di esecuzione:

va tutto bene fintanto che i valori passati alla funzione sono tra quelli "matchabili"

[32]:
```
giorno 7;;
```

[32]: - : string = "domenica"

altrimenti viene sollevata un'eccezione

[33]:
```
giorno 8;;
```

```
Exception: Match_failure ("[31]", 2, 4).
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Pattern matching e costrutto `function`

Come già detto in precedenza, il costrutto `function` è alternativo a `fun` nelle definizione di funzioni. Rispetto a `fun` function ha la limitazione di prevedere un unico parametro.

[84]:
```
function x -> x + 1 ;;
```

[84]:
```
- : int -> int = <fun>
```

L'utilità di `function` sta nel fatto che include un meccanismo di pattern matching sul suo (unico) parametro. Anzichè scrivere:

[86]:
```
let isEmpty lis =
    match lis with
    | [] -> true
    | x::lis' -> false ;;
```

[86]:
```
val isEmpty : 'a list -> bool = <fun>
```

Si può invece scrivere:

[87]:
```
let isEmpty = function
    | [] -> true
    | x::lis' -> false ;;
```

[87]:
```
val isEmpty : 'a list -> bool = <fun>
```

andando quindi a definire un elenco di pattern al posto del parametro della funzione. (Infatti `lis` non è più nominato… dopo `function` ci sono direttamente i pattern.)

Se la funzione deve prevedere più di un parametro, function lavora solo sull'ultimo. Ossia la funzione:

[89]:
```
let is_first n lis =
    match lis with
    | [] -> false;
    | x::lis' -> x=n ;;
```

[89]:
```
val is_first : 'a -> 'a list -> bool = <fun>
```

Può essere riscritta (visto che il parametro su cui si fa pattern matching è l'ultimo) come segue:

[90]:
```
let is_first n = function
    | [] -> false
    | x::lis' -> x=n ;;
```

[90]:
```
val is_first : 'a -> 'a list -> bool = <fun>
```

2.9 La vera potenza del Pattern Matching

Abbiamo usato il pattern matching come strumento di *selezione condizionale*:

- l'abbiamo usato un po' come il costrutto `switch` presente in molti altri linguaggi (ad es. JavaScript)
- rispetto ad uno `switch` (o a un `if`) ci ha consentito di esprimere condizioni sulla struttura (ad es. lista non vuota, usando il pattern `x::lis`)

La vera potenza del pattern matching è che consente di "smontare" le strutture dati:

- processare i singoli elementi di una tupla
- processare i singoli elementi di una lista
- estrarre una sottolista

Questo grazie alle variabili presenti nei pattern!

Quando un valore fa match con un pattern `P_i` che contiene variabili, quelle variabili vengono istanziate e sono utilizzabili nell'espressione `EXP_i`

[34]:
```
let primo t =
    match t with
    | (x,_) -> x ;;
```

[34]:
```
val primo : 'a * 'b -> 'a = <fun>
```

[35]:
```
let somma t =
    match t with
    | (x,y) -> x+y ;;
```

[35]:
```
val somma : int * int -> int = <fun>
```

Abbiamo già visto che, sfruttando il pattern matching implicito in `let`, queste funzioni possono essere scritte più semplicemente così:

[36]:
```
let primo (x,y) = x ;;
let somma (x,y) = x+y ;;
```

[36]:
```
val primo : 'a * 'b -> 'a = <fun>
```

[36]:
```
val somma : int * int -> int = <fun>
```

Un primo esempio con le liste:

[37]:
```
let testa lis =
    match lis with
    | x::lis' -> x ;;
```

```
File "[37]", line 2, characters 4-37:
2 | ...match lis with
3 |     | x::lis' -> x...
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
```

[37]:
```
val testa : 'a list -> 'a = <fun>
```

A causa del pattern matching non esaustivo, questa è una *funzione parziale*

- funziona solo su liste non vuote

Accorpare casi del pattern matching

E' possibile accorpare casi del pattern matching per associarli alla stessa espressione

[38]:
```
let giorno_festivo n =
    match n with
    | 1 | 2 | 3 | 4 | 5 -> "feriale"
    | 6 | 7 -> "festivo"
    | _ -> "ERRORE" ;;
```

[38]:
```
val giorno_festivo : int -> string = <fun>
```

Ma se si usano variabili, bisogna che siano presenti in tutti i pattern accorpati

[39]:
```
let rec somma_coppia c =
    match c with
    | (x,0) | (0,x) -> x
    | (x,y) -> x+y ;;
```

[39]:
```
val somma_coppia : int * int -> int = <fun>
```

Altrimenti bisogna separare i casi, o sostituire le variabili (se non servono) con _

[40]:
```
let lunghezza_uno_due lis =
    match lis with
    | x::[] | x::y::[] -> true
    | _ -> false ;;
```

```
File "[40]", line 3, characters 6-22:
3 |     | x::[] | x::y::[] -> true
          ^^^^^^^^^^^^^^^^
Error: Variable y must occur on both sides of this | pattern
```

In questo esempio sono stati accorpati i pattern `x::[]` e `x::y::[]`, e il problema è che la variabile `y` è usata solo in uno dei due casi pattern. Questo non piace al compilatore, in quanto `y` potrebbe essere usata nell'espressione associata a questi due pattern, e se così fosse quando la lista ha lunghezza uno l'espressione si troverebbe ad usare una variabile a cui non è associato alcun valore.

Ad esempio, se avessimo avuto:

```
| x::[] | x::y::[] -> x+y
```

nel caso in cui la lista avesse fatto match con `x::[]`, la variabile `y` nell'espressione `x+y` non sarebbe stata definita.

Nell'esempio della funzione `lunghezza_uno_due` in realtà questo problema non accade, in quanto l'espressione associata ai pattern è semplicemente `true`, e quindi non usa nessuna variabile. Il compilatore, però, effettua i controlli sulle variabili nei pattern senza tenere conto dell'espressione a destra di `->`, e per questo segnala l'errore.

Per risolvere la situazione, una prima soluzione è di separare i pattern in due casi distinti:

[41]:
```
let lunghezza_uno_due lis =
    match lis with
    | x::[] -> true
    | x::y::[] -> true
    | _ -> false ;;
```

[41]:
```
val lunghezza_uno_due : 'a list -> bool = <fun>
```

Una seconda soluzione, visto che x e y in realtà non servono a nulla, è di utilizzare la wildcard _ al posto delle variabili, come segue:

[42]:
```
let lunghezza_uno_due lis =
    match lis with
    | _::[] | _::_::[] -> true
    | _ -> false ;;
```

[42]:
```
val lunghezza_uno_due : 'a list -> bool = <fun>
```

2.10 Funzioni ricorsive su liste

Il pattern matching ci consente ora di scrivere funzioni (ricorsive) che processano liste

[43]:
```
let rec length lis =
    match lis with
    | [] -> 0
    | x::lis' -> 1 + length lis' ;;
```

[43]:
```
val length : 'a list -> int = <fun>
```

[44]:
```
let rec somma lis =
    match lis with
    | [] -> 0
    | x::lis' -> x + somma lis' ;;
```

[44]:
```
val somma : int list -> int = <fun>
```

[45]:
```
let rec contains x lis =
    match lis with
    | [] -> false
    | y::lis' -> if y=x then true
                 else contains x lis'
```

[45]:
```
val contains : 'a -> 'a list -> bool = <fun>
```

Implementazione di @:

[46]:
```
let rec append l1 l2 =
  match l1 with
    [] -> l2
  | x :: l1' -> x :: (append l1' l2)
```

[46]: val append : 'a list -> 'a list -> 'a list = <fun>

Richiede tempo lineare nella lunghezza di `l1`

Rovesciare una lista

Soluzione poco efficiente (perché usa `@`):

```
[47]: let rec rev lis =
          match lis with
          | [] -> []
          | x::lis' -> (rev lis') @ [x] ;;
```

```
[47]: val rev : 'a list -> 'a list = <fun>
```

Soluzione più efficiente che usa un parametro `lis2` di accumulazione del risultato (*accumulatore*):

```
[48]: let rev lis =
          let rec rev_accum lis1 lis2 =
              match lis1 with
              | [] -> lis2
              | x::lis1' -> rev_accum lis1' (x::lis2)
          in
              rev_accum lis [] ;;
```

```
[48]: val rev : 'a list -> 'a list = <fun>
```

La funzione `rev_accum` ad ogni chiamata ricorsiva prende l'elemento in testa alla lista `lis1` e lo "sposta" in testa alla lista `lis2`. Quindi gli elementi vengono presi uno dopo l'altro da `lis1` e aggiunti uno prima dell'altro in `lis2`, con il risultato di rovesciare la lista.

In questo esempio il parametro `lis2` è detto *accumulatore* in quanto è un parametro che viene usato per costruire passo passo il risultato. Nell'ambito della programmazione funzionale, in cui la ricorsione è fondamentale e largamente usata, i parametri di accumulazione vengono spesso utilizzati in quanto consentono di definire funzioni con la ricorsione in coda (*tail-recursive*), ossia in cui la chiamata ricorsiva è l'ultima operazione svolta.

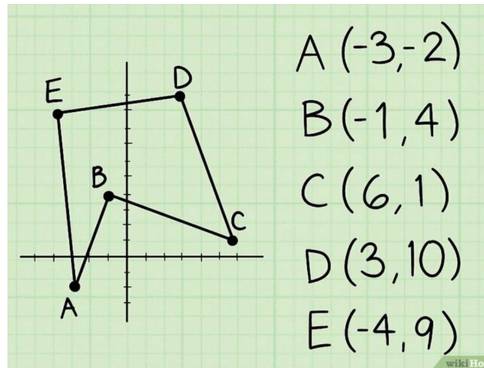
Questo è il caso anche della funzione `rev_accum`, in cui, nel caso ricorsivo, la cui chiamata ricorsiva è l'ultima operazione effettivamente svolta, dopo aver creato il risultato parziale `x::lis2` nel parametro di accumulazione.

L'utilizzo della tail-recursion è molto importante nelle funzioni che prevedono una lunga sequenza di chiamate ricorsive. Essa infatti, grazie ad ottimizzazioni che vengono svolte a tempo di esecuzione, consente di allocare nello stack un unico record di attivazione "riciclandolo" ad ogni chiamata ricorsiva. Come conseguenza, il programma viene eseguito più velocemente (si risparmia il tempo di allocare un nuovo record di attivazione), consumando meno memoria e, soprattutto, senza correre il rischio di riempire tutta la memoria a disposizione dello stack (*stack overflow*) con un conseguente arresto del programma.

2.11 Un esempio non banale: area di un poligono irregolare

Scriviamo un programma per calcolare l'area di un qualunque poligono usando il metodo descritto qui:

- https://www.wikihow.it/Calcolare-l%27Area-di-un-Poligono



In sostanza, il metodo prevede di:

- prendere i vertici *in senso antiorario* come lista di coordinate cartesiane $(x_1, y_1), \ldots, (x_n, y_n)$, copiando il primo vertice in fondo alla lista:

```
[ (-3,-2); (-1,4); (6,1); (3,10); (-4,9); (-3,-2)  ]
```

- calcolare la somma dei prodotti di ogni x_i con y_{i+1}

```
(-3*4) + (-1*1) + (6*10) + (3*9) + (-4*-2) = 82
```

- calcolare la somma dei prodotti di ogni y_i con x_{i+1}

```
(-2*-1) + (4*6) + (1*3) + (10*-4) + (9*-3) = -38
```

- sottrarre il secondo dal primo e dividere per due

```
(82-(-38))/2 = (82+38)/2 = 120/2 = 60
```

Soluzione completa:

```
[49]: let poligono = [ (-3,-2); (-1,4); (6,1); (3,10); (-4,9) ] ;;

let area pol =
    let rec passo1 (x0,y0) lis =
        match lis with
        | [] -> 1  (* non viene mai eseguito *)
        | (x,y)::[] -> x*y0
        | (x1,y1)::(x2,y2)::lis' -> x1*y2 + passo1 (x0,y0) ((x2,y2)::lis')
    in
    let rec passo2 (x0,y0) lis =
        match lis with
        | [] -> 1  (* non viene mai eseguito *)
        | (x,y)::[] -> y*x0
        | (x1,y1)::(x2,y2)::lis' -> y1*x2 + passo2 (x0,y0) ((x2,y2)::lis')
    in
    match pol with
    | [] -> 0.
    | p0::lis -> float_of_int (passo1 p0 pol - passo2 p0 pol) /. 2. ;;
(* invece che copiare il primo elemento in fondo, lo passiamo come parametro *)

area poligono;;
```

```
[49]: val poligono : (int * int) list =
        [(-3, -2); (-1, 4); (6, 1); (3, 10); (-4, 9)]
```

```
[49]: val area : (int * int) list -> float = <fun>
```

```
[49]: - : float = 60.
```

Questa funzione esegue le due sommatorie previste dal metodo tramite due funzioni ricorsive `passo1` e `passo2`. Queste due funzioni dovrebbero lavorare sulla lista in cui il primo elemento è stato copiato in fondo. Questo richiederebbe innazitutto di eseguire `pol @ [p0]`, dove `p0=(x0,y0)` è il primo elemento di `pol`. Questo però sarebbe inefficiente e consumerebbe memoria (per copiare tutto `pol`. La soluazione adottata, invece, passa `p0` come parametro a `passo1` e `passo2`. Queste due funzioni continueranno a passarsi `p0` da una chiamata all'altra mentre scorrono la lista. Arrivate in fondo alla lista utilizzerano il `p0` ricevuto come parametro come se fosse l'ultimo elemento della lista.

Le funzione `passo1` e `passo2` sono sostanzialmente identiche. L'unica cosa che cambia è che la prima ad ogni passo calcola `x1*y2`, mentre la seconda calcola `x2*y1`.

L'espressione che avvia il calcolo è quella nell'ultima riga del programma. Se `pol` non è vuota, tramite il pattern `p0::lis` viene identificato il primo elemento di tale lista e viene utilizzato per chiamare `passo1` e `passo2`. Viene eseguita la sottrazione tra i risultati ottenuti dalle due funzioni e poi viene eseguita la divisione per due.

Mentre `passo1`, `passo2` e la sottrazione vengono eseguite lavorando su dati di tipo `int` (nota: le coordinate dei punti nella lsita sono dati come valori interi) la divisione per due viene eseguita ricorrendo all'operazione su `float`. Questo perchè a differenza delle somme, moltiplicazioni e sottrazioni usate in `passo1` e `passo2`, l'operazione di divisione su interi `/` potrebbe introdurre un'approssimazione (l'area potrebbe non essere intera). Per questo motivo è più corretto usare `/.` dopo avere eseguito l'opportuna conversione da `int` a `float`.

Vediamo ora però come migliorare questa soluzione evitando di scandire due volte la lista (una volta per `passo1` e una volta per `passo2`).

Soluzione migliorata (accorpa `passo1` e `passo2` in un'unica funzione ricorsiva):

```
[50]: let poligono = [ (-3,-2); (-1,4); (6,1); (3,10); (-4,9) ] ;;

let area pol =
    let rec calcolo (x0,y0) lis =
        match lis with
        | [] -> (1,1) (* non viene mai eseguito *)
        | (x,y)::[] -> (x*y0 , y*x0)
        | (x1,y1)::(x2,y2)::lis' ->
            let (ris1,ris2) = calcolo (x0,y0) ((x2,y2)::lis')
            in (ris1 + x1*y2 , ris2 + y1*x2)
    in
    match pol with
    | [] -> 0.
    | p0::lis -> let (ris1,ris2) = calcolo p0 pol
                 in float_of_int (ris1 - ris2) /. 2. ;;

area poligono;;
```

```
[50]: val poligono : (int * int) list =
        [(-3, -2); (-1, 4); (6, 1); (3, 10); (-4, 9)]
```

```
[50]: val area : (int * int) list -> float = <fun>
```

```
[50]: - : float = 60.
```

In questa soluzione le funzioni `passo1` e `passo2` sono state sostuite da un'unica funzione ricorsiva `calcolo` che esegue i due conteggi scandendo un'unica volta la lista e portandosi dietro una coppia di valori, anzichè un solo valore. Nel primo elemento della coppia verrà calcolato il risultato che corrispondeva a `passo1`, e nel secondo quello che corrispondeva a `passo2`.

2.12 Funzioni higher-order su liste

Abbiamo visto che nella programmazione funzionale la ricorsione è l'unico modo con cui possiamo ripetere un calcolo più volte

- Per scandire o processare una lista è inevitabile ricorrere alla ricorsione
- Esistono però degli schemi ricorrenti nelle funzioni che processano liste...

Questi schemi possono essere modellati come utili funzioni higher-order

Vediamo qualche esempio:

[51]:
```
let rec contiene_zero lis =
    match lis with
    | [] -> false
    | x::lis' -> if x=0 then true
                 else contiene_zero lis' ;;
```

[51]:
```
val contiene_zero : int list -> bool = <fun>
```

[52]:
```
let rec contiene_positivo lis =
    match lis with
    | [] -> false
    | x::lis' -> if x>0 then true
                 else contiene_positivo lis' ;;
```

[52]:
```
val contiene_positivo : int list -> bool = <fun>
```

[53]:
```
let rec contiene_pari lis =
    match lis with
    | [] -> false
    | x::lis' -> if x mod 2 = 0 then true
                 else contiene_pari lis' ;;
```

[53]:
```
val contiene_pari : int list -> bool = <fun>
```

Tutte queste funzioni:

- scandiscono la lista (usando la ricorsione)
- valutano una *condizione* su ogni elemento (essere uguale a `x`, essere positivo, essere pari, ...)
- restituiscono `true` se incontrano un elemento in cui la condizione è vera
- restituiscono `false` se arrivano in fondo alla lista

L'unica differenza tra le tre funzioni `contiene_zero`, `contiene_positivo` e `contiene_pari` è nella condizione testata su ogni elemento

IDEA:

- astraiamo il test della condizione come un *predicato* (funzione `XXX -> bool`)
- scriviamo un'unica funzione che prende tale predicato come parametro e verifica se ESISTA un elemento che lo soddisfi

Exists

Ecco la funzione (higher-order) che rappresenta questo schema

[54]:
```
let rec exists p lis =
    match lis with
    | [] -> false
    | x::lis' -> if p x then true
                 else exists p lis' ;;
```

[54]:
```
val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

ora possiamo ridefinire le funzioni `contiene_zero`, `contiene_positivo` e `contiene_pari` usando la ricorsione in modo implicito (nascosta dentro a `exists`)

[55]:
```
let contiene_zero lis = exists (fun x -> x=0) lis ;;
let contiene_positivo lis = exists (fun x -> x>0) lis ;;
let contiene_pari lis = exists (fun x -> x mod 2 = 0) lis ;;
```

[55]:
```
val contiene_zero : int list -> bool = <fun>
```

[55]:
```
val contiene_positivo : int list -> bool = <fun>
```

[55]:
```
val contiene_pari : int list -> bool = <fun>
```

Anche la funzione `contains`, che verifica se un certo elemento è presente, può essere espressa in termini di `exists`:

[56]:
```
let contains x lis = exists (fun y -> x=y) lis ;;
```

[56]:
```
val contains : 'a -> 'a list -> bool = <fun>
```

In questo caso la funzione `exists` riceve la *chiusura* del predicato

- include il valore corrente di `x`

Vediamo un altro modo "creativo" di definire `contains` con il paradigma funzionale

- usando un'applicazione parziale di funzione

[57]:
```
let uguale x y = x=y;;
let contains x lis = exists (uguale x) lis;;
```

[57]:
```
val uguale : 'a -> 'a -> bool = <fun>
```

[57]:
```
val contains : 'a -> 'a list -> bool = <fun>
```

Forall

In modo simile possiamo definire una funzione higher-order che testa un predicato su *tutti* gli elementi della lista

```
[58]: let rec forall p lis =
          match lis with
          | [] -> true
          | x::lis' -> if p x then forall p lis'
                       else false ;;
```

```
[58]: val forall : ('a -> bool) -> 'a list -> bool = <fun>
```

e possiamo definire le funzioni `tutti_zeri`, `tutti_positivi` e `tutti_pari`

```
[59]: let tutti_zeri lis = forall (fun x -> x=0) lis ;;
      let tutti_positivi lis = forall (fun x -> x>0) lis ;;
      let tutti_pari lis = forall (fun x -> x mod 2 = 0) lis ;;
```

```
[59]: val tutti_zeri : int list -> bool = <fun>
```

```
[59]: val tutti_positivi : int list -> bool = <fun>
```

```
[59]: val tutti_pari : int list -> bool = <fun>
```

Filter

Un'altra operazione frequente sulle liste è quella di selezionare (filtrare) gli elementi secondo una condizione

- restituendo la lista degli elementi che la soddisfano

Anche in questo caso, possiamo definire una funzione higher-order che astrae la condizione con un predicato

```
[60]: let rec filter p lis =
          match lis with
          | [] -> []
          | x::lis' -> if p x then x::filter p lis'
                       else filter p lis' ;;
```

```
[60]: val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

Questa volta la funzione restituisce una lista

Possiamo usare `filter` per definire le funzioni `estrai_zeri`, `estrai_positivi` e `estra_pari`, sempre usando la ricorsione in modo implicito.

```
[61]: let estrai_zeri lis = filter (fun x -> x=0) lis ;;
      let estrai_positivi lis = filter (fun x -> x>0) lis ;;
      let estrai_pari lis = filter (fun x -> x mod 2 = 0) lis ;;
```

```
[61]: val estrai_zeri : int list -> int list = <fun>
```

```
[61]: val estrai_positivi : int list -> int list = <fun>
```

[61]: val estrai_pari : int list -> int list = <fun>

Map

Altra elaborazione frequente: applicare la stessa operazione a tutti gli elementi

- produce una nuova lista con tanti elementi quanto quella processata
- il tipo degli elementi può essere diverso
- per astrarre sull'operazione abbiamo bisogno di una funzione, non di un predicato

[62]:
```
let rec map f lis =
    match lis with
    | [] -> []
    | x::lis' -> f x::map f lis' ;;
```

[62]: val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

Qualche esempio d'uso:

[63]:
```
map (fun x -> x+1) [1;2;3] ;;
```

[63]: - : int list = [2; 3; 4]

[64]:
```
let primo lis = map (fun (x,y) -> x) lis ;;
primo [ (3,2); (4,7); (9,2)] ;;
```

[64]: val primo : ('a * 'b) list -> 'a list = <fun>

[64]: - : int list = [3; 4; 9]

Fold-right

Le funzioni higher-order viste fino ad ora lavoravano sui singoli elementi della lista in modo indipendente:

- `filter` valuta ogni elemento e sceglie se inserirlo nel risultato (indipendentemente dagli altri)
- `map` applica una funzione ad ogni elemento (indipendentemente dagli altri)
- ...

Spesso si vuole elaborare tutti gli elementi della lista per calcolare un unico risultato

Ad esempio:

- Calcolare la *somma* degli elementi di una lista
- Calcolare *minimo e massimo* di una lista
- *Concatenare tutti gli elementi* di una lista di stringhe

Intuitivamente, queste elaborazioni richiedono di scandire la lista portandosi dietro una o più variabili che memorizzano il *risultato parziale* via via calcolato

Ad esempio, nell'approccio imperativo (in JavaScript e con array…) useremmo un `for`:

```
function somma(a) {
    var s = 0
    for (var i in a)
        s += a[i]
```

```
    return s
}
```

Lo stesso per calcolare minimo e massimo, concatenare tutti gli elementi, ecc. . .

L'importante è capire:

- che variabili "portarsi dietro" nel ciclo
- come aggiornarle ad ogni iterazione

Anche seguendo un approccio *ricorsivo*, nella scansione di una lista è fondamentale capire come portarsi dietro il risultato parziale e come aggiornalo ad ogni passo

Il risultato parziale non sarà memorizzato in una variabile, ma passato da una chiamata all'altra come parametro o valore di ritorno

[65]:
```
let rec somma lis =
    match lis with
    | [] -> 0
    | x::lis' -> x + somma lis' ;;
somma [3;2;4] ;;
```

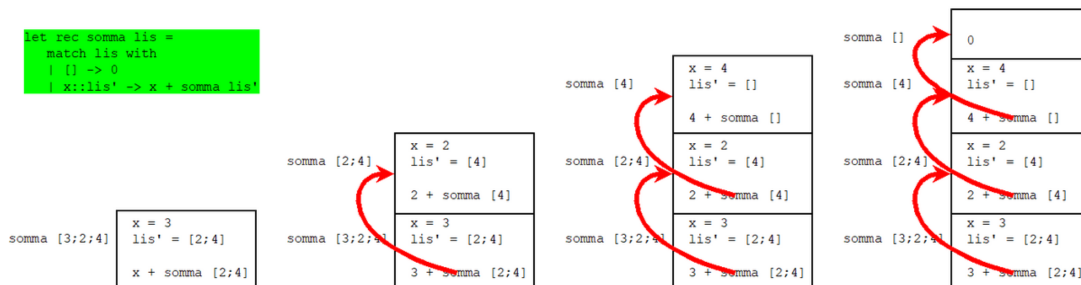
[65]:
```
val somma : int list -> int = <fun>
```

[65]:
```
- : int = 9
```

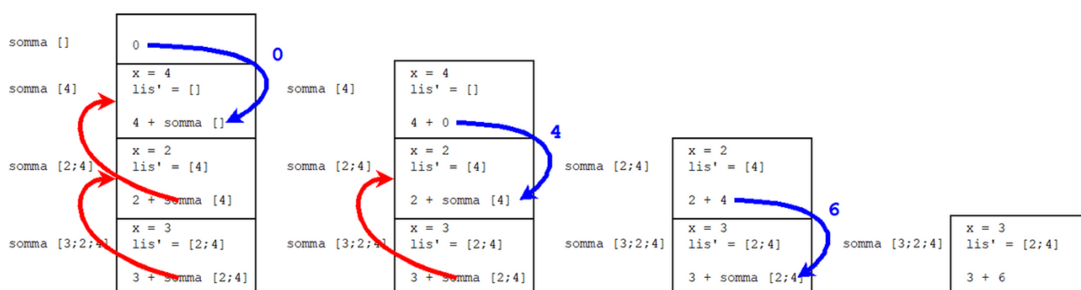
In questo esempio, ogni chiamata a `somma` restituisce la somma della porzione di lista che va dall'elemento corrente fino in fondo

- il risultato parziale è restituito dalla funzione

Vediamo che cosa succede nello *stack*:



A cui segue:



Quindi: * gli elementi della lista vengono in realtà sommati dall'ultimo al primo (da *destra*) * ad ogni passo si applica la funzione/operatore `(+)` che somma due numeri

Cerchiamo di estrarre lo schema ricorsivo su cui si basa la funzione `somma`

```
somma [3;2;4]
3 + somma [2;4]
3 + (2 + somma [4])
3 + (2 + (4 + somma []))
3 + (2 + (4 + 0))
```

Generalizziamo `(+) = f`

`f 3 (f 2 (f 4 0))`

Generalizziamo `lis = [x1;x2;...;xN]` e caso base restituisce `a` invece che `0`

`f x1 (f x2 (... (f xN a)..))`

La funzione `fold_right` realizza lo schema ricorsivo che abbiamo identificato

[66]:
```
let rec fold_right f lis a =
    match lis with
    | [] -> a
    | x::lis' -> f x (fold_right f lis' a) ;;
```

[66]:
```
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Prevede i parametri (oltre alla lista `lis`):

- `f`: la funzione da applicare ad ogni passo
- `a`: il risultato corrispondente al caso base (lista vuota `[]`)

Ad ogni passo, applica `f` all'elemento e al risultato ottenuto sul resto della lista

Usiamo `fold_right` per definire `somma` usando la ricorsione in modo implicito:

[67]:
```
let somma lis = fold_right (+) lis 0 ;;
somma [3;2;4] ;;
```

[67]:
```
val somma : int list -> int = <fun>
```

[67]:
```
- : int = 9
```

Altri esempi d'uso di `fold_right`:

[68]:
```
let concat lis = fold_right (^) lis "" ;;

concat ["abc"; "def"; "gh"] ;;
```

[68]:
```
val concat : string list -> string = <fun>
```

[68]:
```
- : string = "abcdefgh"
```

[69]:
```
let stringa_piu_lunga lis =
    let f x s =
        if String.length x > String.length s then x else s
    in
        fold_right f lis "" ;;
```

```
stringa_piu_lunga ["ciao";"hello";"hi"] ;;
```

[69]:
```
val stringa_piu_lunga : string list -> string = <fun>
```

[69]:
```
- : string = "hello"
```

Calcolare minimo e massimo di una lista

- che cosa è utile portarsi dietro durante la scansione della lista?
- che funzione si deve applicare ad ogni passo?
- inoltre: qual'è il valore del caso base `[]` ??

Idea: si estrae il primo elemento e lo si usa come caso base

[70]:
```
let minimo_massimo lis =
    let f x (min,max) =
        if x<min then (x,max)
        else if x>max then (min,x)
        else (min,max)
    in
        match lis with
        | [] -> (0,0)
        | x::lis' -> fold_right f lis' (x,x) ;;

minimo_massimo [3;2;4] ;;
```

[70]:
```
val minimo_massimo : int list -> int * int = <fun>
```

[70]:
```
- : int * int = (2, 4)
```

In questa implementazione della funzione `minimo_massimo` viene restituita la coppia `(0,0)` nel caso in cui la lista sia vuota. Dal momento che il primo elemento viene estratto ed usato come primo candidato minimo e massimo, la funizone ricorsiva viene utilizzata su `lis'`, quindi la ricorsione parte a tutti gli effetti dal secondo elemento della lista.

Fold-left

Tornando all'esempio della somma, si può definire una funzione ricorsiva anche così:

[71]:
```
let rec somma a lis =
    match lis with
    | [] -> a
    | x::lis' -> somma (a+x) lis' ;;

somma 0 [3;2;4] ;;
```

[71]:
```
val somma : int -> int list -> int = <fun>
```

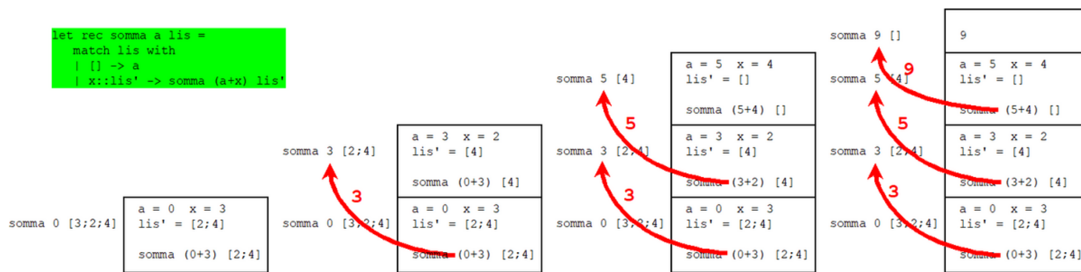
[71]:
```
- : int = 9
```

Il risultato parziale viene:

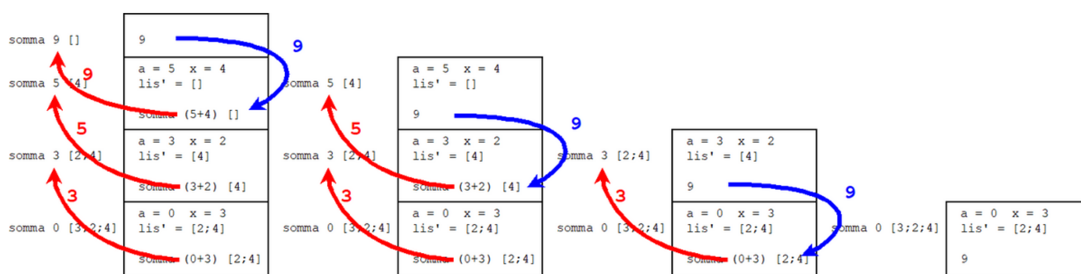
- *passato come parametro* (inizialmente zero)

- aggiornato man mano che si incontrano nuovi elementi scandendo la lista dall'inizio alla fine (da *sinistra*)

Vediamo che cosa succede nello *stack*:



A cui segue:



Quindi:

- gli elementi della lista vengono sommati dal primo all'ultimo (da *sinistra*)
- ad ogni passo si applica la funzione/operatore `(+)` che somma due numeri
- il risultato "scende" immutato lungo lo stack (la funzione è *tail recursive*, quindi l'uso dello stack si può ottimizzare...)

Come già detto nel capitolo precedente, il fatto che la funzione sia abbia la chiamata ricorsiva in coda (tail-recursive) fa si che in realtà venga utilizzato un unico record di attivazione nello stack. Questo minimizza l'uso della memoria e previene il problema dello *stack overflow*.

Estraiamo lo schema ricorsivo su cui si basa la nuova versione della funzione `somma`

```
somma 0 [3;2;4]
somma (0 + 3) [2;4]
somma ((0 + 3) + 2) [4]
somma (((0 + 3) + 2) + 4) []
(((0 + 3) + 2) + 4)
```

Generalizziamo `(+) = f`

```
f (f (f 0 3) 2) 4
```

Generalizziamo ancora `lis = [x1;x2;...;xN]` e valore iniziale `a` invece che `0`

```
f (f (... (f (f a x1) x2) ...) xN-1) xN
```

La funzione `fold_left` realizza lo schema ricorsivo che abbiamo identificato

[72]:
```
let rec fold_left f a lis =
    match lis with
    | [] -> a
    | x::lis' -> fold_left f (f a x) lis' ;;
```

[72]:
```
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Prevede i parametri (oltre alla lista `lis`):

- `f`: la funzione da applicare ad ogni passo
- `a`: il valore iniziale da cui partire con il calcolo

Ad ogni passo, richiama ricorsivamente `f` passandogli un valore via via aggiornato

Ora possiamo usare `fold_left` per definire `somma` usando la ricorsione in modo implicito

[73]:
```
let somma lis = fold_left (+) 0 lis ;;
somma [3;2;4] ;;
```

[73]:
```
val somma : int list -> int = <fun>
```

[73]:
```
- : int = 9
```

Altri esempi d'uso di `fold_left` (gli stessi già visti con `fold_right`):

[74]:
```
let concat lis = fold_left (^) "" lis ;;

concat ["abc"; "def"; "gh"] ;;
```

[74]:
```
val concat : string list -> string = <fun>
```

[74]:
```
- : string = "abcdefgh"
```

[75]:
```
let stringa_piu_lunga lis =
    let f s x =
        if String.length x > String.length s then x else s
    in
        fold_left f "" lis ;;

stringa_piu_lunga ["ciao";"hello";"hi"] ;;
```

[75]:
```
val stringa_piu_lunga : string list -> string = <fun>
```

[75]:
```
- : string = "hello"
```

Calcolare minimo e massimo di una lista:

[76]:
```
let minimo_massimo lis =
    let f (min,max) x =
        if x<min then (x,max)
        else if x>max then (min,x)
        else (min,max)
    in
        match lis with
        | [] -> (0,0)
        | x::lis' -> fold_left f (x,x) lis' ;;

minimo_massimo [3;2;4] ;;
```

[76]: val minimo_massimo : int list -> int * int = <fun>

[76]: - : int * int = (2, 4)

ATTENZIONE: rispetto alla versione con `fold_right`, i parametri di `f` sono invertiti e anche quelli di `fold_left` hanno un ordine diverso

Fold-right VS Fold-left

Ma quindi `fold_right` e `fold_left` sono intercambiabili?

- No!

Innanzitutto:

- `fold_left` è tail recursive (quindi si può ottimizzare l'uso dello stack)
- `fold_right` non lo è (anche se esistono implementazioni tail recursive)

Ma soprattutto, in alcuni casi possono portare a risultati diversi!

Esempio: funzione identità su liste (fa una "copia" della lista)

[77]:
```
let id_lista1 lis =
    let f x l = x::l
    in fold_right f lis [];;

let id_lista2 lis =
    let f l x = x::l
    in fold_left f [] lis;;

id_lista1 [1;2;3] ;;
id_lista2 [1;2;3] ;;
```

[77]: val id_lista1 : 'a list -> 'a list = <fun>

[77]: val id_lista2 : 'a list -> 'a list = <fun>

[77]: - : int list = [1; 2; 3]

[77]: - : int list = [3; 2; 1]

La versione con `fold_left` ha **ROVESCIATO** la lista

- `::` aggiunge in testa! Per copiare la lista gli elementi gli vanno dati dall'ultimo al primo

2.13 Un paio di esempi con `fold_right` e `fold_left`

- Crea una nuova lista sommando gli elementi a partire dal fondo

[78]:
```
let somma_dal_fondo lis =
    let f x l =
        match l with
        | [] -> [x]
        | x'::l' -> x+x'::l
```

```
    in
        fold_right f lis [] ;;
```

[78]: `val somma_dal_fondo : int list -> int list = <fun>`

[79]:
```
somma_dal_fondo [2;2;2;2;2] ;;
```

[79]: `- : int list = [10; 8; 6; 4; 2]`

- verifica se una lista è ordinata in modo crescente

[80]:
```
let crescente lis =
    (* f si "porta dietro" il precedente *)
    let f (prec,b) x = (x,b && prec<=x)
    in match lis with
    | [] -> true
    | x::lis' -> let (_,b) =
                    fold_left f (x,true) lis'
                 in b;;
```

[80]: `val crescente : 'a list -> bool = <fun>`

[81]:
```
crescente [1;4;4;6;9] ;;
crescente [3;5;4;7;9] ;;
```

[81]: `- : bool = true`

[81]: `- : bool = false`

Quest'ultimo esempio mostra come sia possibile processare ad ogni passo due elementi consecutivi della lista. Questo di base non è previsto da `fold_right` e `fold_left`, in quanto la funzione ausiliaria `f` da loro utilizzata legge un solo elemento della lista per volta. E' però possibile aggiungere tra i dati da portarsi dietro per il calcolo un'informazione sul precedente valore della lista incontrato (`prec`, nell'esempio) facendolo aggiornare di volta in volta alla `f`.

Nel caso di `fold_left`, che scandisce gli elementi dal primo all'ultimo, questo `prec` sarà effettivamente l'elemento che nella lista precede l'elemento corrente. Nel caso di `fold_right`, questo `prec` sarà invece l'elemento successivo a quello corrente, dal momento `fold_right` scandisce la lista in direzione inversa.

2.14 Le funzioni su liste nel modulo `List`

Le funzioni higher-order su liste che abbiamo visto sono disponibili nel modulo `List` della libreria standard di OCaml (https://ocaml.org/api/List.html)

- `exists` $\Longrightarrow$ `List.exists`
- `forall` $\Longrightarrow$ `List.for_all`
- `filter` $\Longrightarrow$ `List.filter`
- `map` $\Longrightarrow$ `List.map`
- `fold_right` $\Longrightarrow$ `List.fold_right`
- `fold_left` $\Longrightarrow$ `List.fold_left`

Oltre a molte altre funzioni utili:

- `List.find`, `List.sort`, `List.partition`, `List.merge`,...

Esempio: area di un poligono irregolare con `fold_right` e `List`

Riprendendo l'esempio del poligono irregolare:

[82]:
```
let poligono = [ (-3,-2); (-1,4); (6,1); (3,10); (-4,9) ] ;;

let area pol =
    let f (x1,y1) ((x2,y2),(ris1,ris2)) =
        ((x1,y1),(ris1+x1*y2,ris2+x2*y1))
    in
    let (p,(ris1,ris2)) = List.fold_right f pol (List.hd pol,(0,0))
    in
    (float_of_int (ris1 - ris2)) /. 2.;;

area poligono;;
```

[82]:
```
val poligono : (int * int) list =
  [(-3, -2); (-1, 4); (6, 1); (3, 10); (-4, 9)]
```

[82]:
```
val area : (int * int) list -> float = <fun>
```

[82]:
```
- : float = 60.
```

Questa implementazione della funzione `area` è molto più compatta di quelle date in precedenza, ma anche più difficile da comprendere a primo sguardo.

Si usa `fold_right` per scandire la lista dalla coda alla testa. La funzione `f` che viene richiamata ad ogni passo prende come primo parametro l'elemento corrente (la coppia `(x1,y1)`) e i dati da "portarsi dietro" che includono l'elemento successivo a quello corrente `(x2,y2)` e i risultati parziali calcolati fino a quel momento `(ris1,ris2)`. Dal momento che i dati da portarsi dietro devono essere passati come unico parametro, queste due coppie diventano una coppia di coppie `((x2,y2),(ris1,ris2))`.

La chiamata a `folr_right` inizializza queste informazioni a `List.hd` (il primo elemento della lista che, concettualmente, dovrebbe essere copiato in fondo, ma che invece sarà processato senza effettivamente aggiungerlo alla lista) e `(0,0)` in quanto il calcolo deve ancora iniziare.

Ad ogni passo, `f` prende l'elemento corrente (dall'ultimo al primo) fa le moltiplicazioni con il successivo (`x1*y2` e `x2*y1`) e somma quanto ottenuto ai risultati parziali `ris1` e `ris2`. Il risultato di `f`, oltre a contenere i valori aggiornati di `ris1` e `ris2`, conterrà come primo elemento della coppia restituita, l'elemento corrente `(x1,y1)`, in modo che diventi, nella successiva chiamata di `f` operata dalla `fold_right`, l'elemento considerato come successivo.

Il risultato finale che sarà restituito da `fold_right` è una coppia `(p,(ris1,ris2))` in cui `p` corrisponderà al primo elemento della lista (che ci stiamo portando dietro ricorsivamente), mentre `ris1` e `ris2` saranno gli effettivi risultati corrispondenti alle due fasi di processamento viste all'inizio.

Il risultato della funzione `pol` sarà quindi `(float_of_int (ris1 - ris2)) /. 2.`, come nelle versioni della funzione `area` già illustrate in precedenza.

Questa funzione è stata implementata usando `fold_right` in modo da rispecchiare l'ordine di esecuzione delle operazioni che si otteneva eseguento le precendi versioni mostrate. E' possibile definirne una versione usando invece `fold_left`.

2.15 Funzioni higher-order su liste in altri linguaggi

Le funzioni higher-order su liste che abbiamo visto sono presenti (a volte parzialmente) anche in altri linguaggi.

- In JavaScript, ad esempio, funzioni quali `map`, `filter`, `reduce` (equivale a `fold_left`) possono essere usate su array

```
JAVASCRIPT:
const array1 = [1, 2, 3, 4];
const reducer = (accum, current) => accum + current;
console.log(array1.reduce(reducer)); // 1+2+3+4=10
```

- In Java, dalla versione 8 (del 2014), sono stati introdotti elementi di programmazione funzionale (lambda expressions) e metodi higher-order per alcune strutture dati

- ...

Capitolo 3

Tipi Algebrici (Record e Variant)

3.1 Definire nuovi tipi

In OCaml è possibile definire nuovi tipi di dato usando il costrutto `type`

[1]:
```
type data = int*int*int ;;
```

[1]:
```
type data = int * int * int
```

Abbiamo definito un alias per `int*int*int`...

[2]:
```
let controlla_data (d:data) =
    match d with
    | gg,mm,aaaa -> gg>0 && gg<=31
                 && mm>0 && mm<=12
                 && aaaa>1900 && aaaa<=2030 ;;
```

[2]:
```
val controlla_data : data -> bool = <fun>
```

[3]:
```
let accoda_data (d:data) lis =
    if controlla_data(d) then d::lis else lis ;;
```

[3]:
```
val accoda_data : data -> data list -> data list = <fun>
```

L'uso degli alias può rendere il codice più leggibile, soprattutto quando abbiamo a che fare con tipi molto complicati (tuple di tuple...). Se etichettiamo il parametro di una funzione con un alias, come in questi esempi, l'algoritmo di inferenza di tipi di OCaml lo utilizzerà rendendoci la descrizione del tipo inferto più comprensibile.

3.2 Tipi algebrici

OCaml consente di *comporre* tipi di dato per creare nuovi tipi

Immaginiamo un tipo come un insieme di valori

- `int` come l'insieme dei numeri interi,
- `bool` come l'insieme {`true`,`false`},
- ecc...

Le operazioni di costruzione di nuovi tipi in OCaml sono ispirate alle operazioni insiemistiche: * *prodotto cartesiano* ($\times$), a cui corrispondono le *tuple* e i *record* * *unione* ($\cup$) (che è una *somma* di insiemi), a cui corrispondo i *variant*

3.3 Tipi prodotto: tuple e record

Il prodotto cartesiano di due insiemi (es. $\mathbb{N} \times \mathbb{N}$) è un insieme di coppie.

- $(7,4) \in \mathbb{N} \times \mathbb{N}$

Allo stesso modo il prodotto di due tipi (es. `int * int`) è un tipo di coppie

- `(7,4) : int*int`

Il discorso si estende a tuple di qualunque dimensione, come abbiamo già visto...

3.4 Record

I record sono un modo più avanzato di definire tipi prodotto * sono simili a tuple, ma possiamo dare un nome agli elementi (detti *campi*)

Definiamo, ad esempio, un tipo record da usare per descrivere i punti nel piano cartesiano (cioè, in uno spazio bidimensionale). Ogni punto sarà descritto da una coordinata `x` e una coordinata `y` entrambe di tipo `float`.

[4]:
```
type punto_2d = { x: float; y: float; } ;;
```

[4]:
```
type punto_2d = { x : float; y : float; }
```

Definito il tipo di un record, possiamo creare valori di quel tipo

[5]:
```
let p = { x = 3.; y = -4. } ;;
```

[5]:
```
val p : punto_2d = {x = 3.; y = -4.}
```

Un valore di tipo record può essere decomposto usando il *pattern matching*

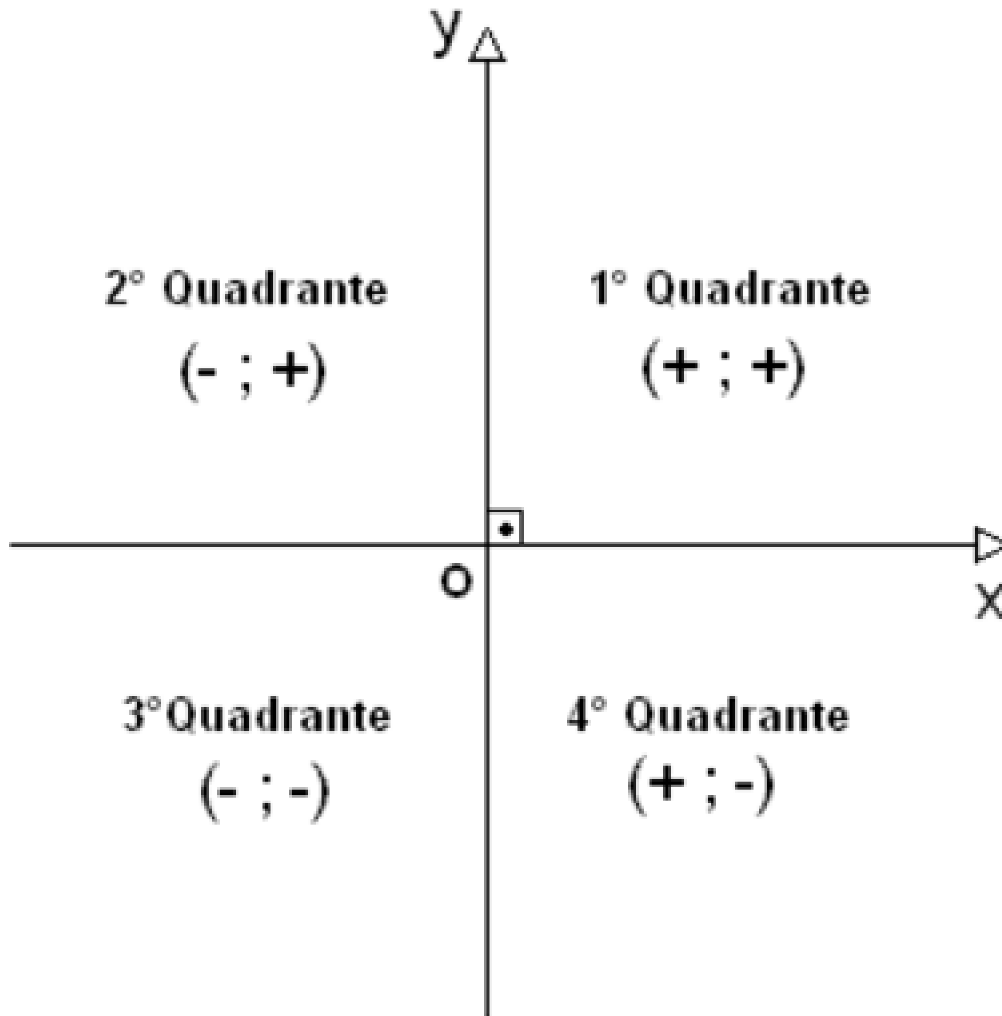
Ad esempio, calcoliamo il quadrante del piano cartesiano in cui ricade un punto

[6]:
```
let quadrante {x = x_pos; y = y_pos } =
    match x_pos>=0.,y_pos>=0. with
    | true,true -> 1
    | false,true -> 2
    | false,false -> 3
    | true,false -> 4 ;;

quadrante {x=(-3.); y=2.};;
```

[6]:
```
val quadrante : punto_2d -> int = <fun>
```

[6]:
```
- : int = 2
```



ATTENZIONE: Nell'esempio ci sono due pattern matching. Quello che decompone il record è implicito nel `let` e usa il pattern `{x = x_pos; y = y_pos }`

Infatti, analogamente a quanto accade per le tuple, anche con i record è possibile specificare un pattern direttamente nell'operazione `let`, in modo tale da destrutturare immediatamente il record nelle sue componenti.

Un po' più semplicemente, possiamo usare direttamente `x` e `y` come nomi di variabili

[7]:
```
let quadrante {x; y} =
    match x>=0.,y>=0. with
    | true,true -> 1
    | false,true -> 2
    | false,false -> 3
    | true,false -> 4 ;;
```

[7]:
```
val quadrante : punto_2d -> int = <fun>
```

Oppure possiamo accedere ai campi usando la *dot notation*: `p.x` e `p.y`

[8]:
```
let quadrante p =
    match p.x>=0.,p.y>=0. with
    | true,true -> 1
```

```
    | false,true -> 2
    | false,false -> 3
    | true,false -> 4 ;;
```

[8]:
```
val quadrante : punto_2d -> int = <fun>
```

La dot notation consente di usare più variabili di tipo record in modo semplice

[9]:
```
let p = {x=3.; y=3.} ;;
let q = {x=6.; y=7.} ;;
p.x ;;
```

[9]:
```
val p : punto_2d = {x = 3.; y = 3.}
```

[9]:
```
val q : punto_2d = {x = 6.; y = 7.}
```

[9]:
```
- : float = 3.
```

Esempio: Distanza tra due punti nel piano ($d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$)

[10]:
```
let distanza p1 p2 =
    sqrt ( (p2.x -. p1.x)**2. +. (p2.y -. p1.y)**2. ) ;;
```

[10]:
```
val distanza : punto_2d -> punto_2d -> float = <fun>
```

[11]:
```
distanza p q ;;
```

[11]:
```
- : float = 5.
```

Nella funzione `distanza` si è usata la funzione built-in `sqrt` che calcola la radice quadrata (già vista) e anche l'operatore `**` che calcola l'elevamento a potenza: dati `a` e `b` di tipo `float`, l'operazione `a**b` calcola `a` elevato alla `b`.

Quando si usano record con molti campi, le operazioni possono diventare verbose

[12]:
```
type persona = {nome: string; cognome: string; eta: int; indirizzo: string; citta:␣
↪string} ;;
```

[12]:
```
type persona = {
  nome : string;
  cognome : string;
  eta : int;
  indirizzo : string;
  citta : string;
}
```

[13]:
```
let mario = {nome="Mario"; cognome="Rossi"; eta=45; indirizzo="Via Roma 10";␣
↪citta="Pisa"} ;;
```

[13]:
```
val mario : persona =
  {nome = "Mario"; cognome = "Rossi"; eta = 45; indirizzo = "Via Roma 10";
   citta = "Pisa"}
```

Una piccola semplificazione è che possiamo definire una variabile in funzione di un'altra usando il costrutto di *functional update* `with`

[14]:
```
let bianca = { mario with nome = "Bianca"; eta=13 } ;;
```

[14]:
```
val bianca : persona =
  {nome = "Bianca"; cognome = "Rossi"; eta = 13; indirizzo = "Via Roma 10";
   citta = "Pisa"}
```

ATTENZIONE: Stiamo sempre lavorando con strutture dati *immutabili*. Quindi, anche utilizzando la functional update quello che si otterrà è un nuovo record distinto dal precedente (che rimane come era prima).

3.5 Nota: record VS oggetti

I record assomigliano ai dizionari di JavaScript, e si potrebbe pensare che "assegnando" funzioni ai loro elementi quello che si ottiene siano oggetti (come accadrebbe in JavaScript).

[15]:
```
type finto_oggetto = { n: int; update: int -> int } ;;
```

[15]:
```
type finto_oggetto = { n : int; update : int -> int; }
```

La differenza con gli oggetti è che *i record sono immutabili*

- le variabili non rappresentano uno "stato interno"... (base dell'object-orientation)

Inoltre i campi funzione (metodi) *non possono accedere* agli altri campi (variabili d'istanza)

Ad esempio:

[16]:
```
let obj = { n = 10; update = fun x -> x } ;;
obj.n = 11 ;;   (* non posso assegnare... *)
```

[16]:
```
val obj : finto_oggetto = {n = 10; update = <fun>}
```

[16]:
```
- : bool = false
```

In questo esempio si è cercato di usare = per assegnare un nuovo valore al campo `n` del (finto) oggetto, come si farebbe in altri linguaggi. Qui chiaramente l'interprete ha risposto `false` in quanto l'operazione = è un confronto e con `obj.n = 11` non si è fatto altro che confrontare il valore di `n` con `11`.

Vedremo più avanti che nella parte imperativa di OCaml esistono alcuni operatori di assegnamento, tra cui `<-`. Se proviamo ad utilizzarlo in questo caso otteniamo come risposta perentoria che il campo del record è immutabile.

[17]:
```
obj.n <- 11 ;;
```

```
File "[17]", line 1, characters 0-11:
1 | obj.n <- 11 ;;
    ^^^^^^^^^^^
```

```
Error: The record field n is not mutable
```

Nell'esempio successivo, invece, vediamo che i campi funzione non possono accedere agli altri campi del record, come invece potrebbe fare un metodo di un (vero) oggetto.

[18]:
```
let obj = { n = 10; update = fun x -> n + x } ;;
```

```
File "[18]", line 1, characters 38-39:
1 | let obj = { n = 10; update = fun x -> n + x } ;;
                                          ^
Error: Unbound value n
```

3.6 Tipi unione (o somma): variant

L'unione di due insiemi è... l'unione di due insiemi!

- $C = A \cup B$
- $x \in C$ se e solo se $x \in A$ oppure $x \in B$

L'unione di due tipi `t1` e `t2` è un nuovo tipo `t` che idealmente "include" tutti i valori di tipo `t1` e `t2`

Mentre in alcuni linguaggi (ad es. TypeScript) si possono unire direttamente due tipi, scrivendo ad esempio:

`TYPESCRIPT: type t = string | int`

l'operazione di unione prevista da OCaml prevede di *etichettare* i valori dei tipi da unire

Ad esempio, per unire i tipi `string` e `int`, dobbiamo scegliere due etichette (ad esempio `Txt` e `Num`) e usarle in questo modo:

[19]:
```
type numero_testo =
    | Txt of string
    | Num of int ;;
```

[19]:
```
type numero_testo = Txt of string | Num of int
```

In sostanza, ogni riga della definizione corrisponde a un caso possibile di tipo di valore, ed è caratterizzato da una etichetta distinta (detta *costruttore*)

ATTENZIONE: I costruttori devono necessariamente iniziare con la lettera maiuscola

Anche i valori del nuovo tipo dovranno essere etichettati nello stesso modo:

[20]:
```
let x = Txt "34" ;;
let y = Num 26 ;;
```

[20]:
```
val x : numero_testo = Txt "34"
```

[20]:
```
val y : numero_testo = Num 26
```

Dato un valore, è possibile ricostruire a quale dei tipi appartenga, tra quelli che sono stati uniti, usando i costruttori nel pattern matching

[21]:
```
match x with
| Txt s -> "testo"
| Num n -> "numero" ;;
```

[21]:
```
- : string = "testo"
```

L'utilizzo dei costruttori `Txt` e `Num` nei pattern fa innanzitutto capire all'interprete di OCaml che la variabile `x` ha tipo `numero_testo`. A questo punto i vari pattern consentiranno di fare cose diverse in funzione del costruttore trovato in `x`.

In questo esempio, visto che `x` era associato al valore `Txt "34"`, il risultato ottenuto è la stringa`"testo"`.

Qualche esempio di funzione che usa il tipo `numero_testo`:

[22]:
```
let int_of_nt x =
    match x with
    | Txt s -> int_of_string s
    | Num n -> n ;;

let string_of_nt x =
    match x with
    | Txt s -> s
    | Num n -> string_of_int n ;;

int_of_nt (Txt "4") ;;
```

[22]:
```
val int_of_nt : numero_testo -> int = <fun>
```

[22]:
```
val string_of_nt : numero_testo -> string = <fun>
```

[22]:
```
- : int = 4
```

Queste funzioni consentono semplicemente di estrarre una rappresentazione intera o testuale da un qualunque valore di tipo `numero_testo`, indipendentemente dal fatto che al suo interno sia usato il costruttore `Txt` seguito da una string, o il costruttore `Num` seguito da un numero.

Nella dichiarazione di un tipo unione, la parte `of <tipo>` è facoltativa

- si possono definire casi che consistono solo del costruttore
- sono come "singoletti" nell'insiemistica

Usando i soli costruttori, si possono definire *tipi enumerazione*

- In molti linguaggi chiamati *enum*

[23]:
```
type giorno = Lun | Mar | Mer | Gio | Ven | Sab | Dom ;;
```

[23]:
```
type giorno = Lun | Mar | Mer | Gio | Ven | Sab | Dom
```

[24]:
```
Gio ;;
```

[24]:
```
- : giorno = Gio
```

[25]:
```
let is_weekend g =
    match g with
    | Sab | Dom -> true
    | _ -> false ;;
```

[25]:
```
val is_weekend : giorno -> bool = <fun>
```

Un esempio più complesso: * tipo `int` esteso con `NaN`, $+\infty$ e $-\infty$

[26]:
```
type int_ext =
    | Num of int
    | NaN
    | Plus_inf
    | Minus_inf ;;
```

[26]:
```
type int_ext = Num of int | NaN | Plus_inf | Minus_inf
```

In questo caso abbiamo combinato costruttori associati a un tipo (`Num of int`) con costruttori "singoletti" che descrivono casi particolari:

- `NaN` (sta per *Not a Number*) è il "numero" che si ottiene da un'operazione tra interi che restituisce un risultato non definito... ad esempio, la divisione $\frac{0}{0}$.
- `Plus_inf` rappresenta $+\infty$
- `Minus_inf` rappresenta $-\infty$

Definiamo un po' di operazioni artimetiche sul nuovo tipo

[27]:
```
let sum x y =
    match x,y with
    | NaN,_ | _,NaN -> NaN
    | Plus_inf,Minus_inf | Minus_inf,Plus_inf -> NaN
    | Plus_inf,_ | _,Plus_inf -> Plus_inf
    | Minus_inf,_ | _,Minus_inf -> Minus_inf
    | Num n1,Num n2 -> Num (n1+n2) ;;
```

[27]:
```
val sum : int_ext -> int_ext -> int_ext = <fun>
```

La somma di due `int_ext` deve tenere conto che uno o entrabi gli operandi potrebbero essere `NaN`, `Plus_inf` o `Minus_inf`. Nella funzione `sum`, i casi del pattern matching sono ordinati in modo da gestire prima tutti questi casi particolari, e lasciare come ultima opzione il caso "standard" in cui i due operandi sono in realtà degli interi normali.

ATTENZIONE: nessun warning: il pattern matching è esausivo

- non abbiamo scordato casi!

Il controllo di esaustività dei pattern operato dal compilatore è utile per prevenire molti bug. E' infatti impossibile scordarsi di considerare qualche caso (a meno che non ignoriamo il warning che otterremmo). Inoltro, il controllo fa sì che ci venga anche segnalato se abbiamo inserito dei casi ridondanti, ossia due pattern che possono matchare lo stesso valore. Anche questo è spesso segnale di un errore di programmazione.

Ancora:

[28]:
```
let div x y =
    match x,y with
    | NaN,_ | _,NaN -> NaN
    | Plus_inf,Plus_inf | Minus_inf,Minus_inf -> NaN
```

```
    | Plus_inf,Minus_inf | Minus_inf,Plus_inf -> NaN
    | Plus_inf,_ -> Plus_inf
    | Minus_inf,_ -> Minus_inf
    | _,Plus_inf | _,Minus_inf -> Num 0
    | Num 0,Num 0 -> NaN
    | Num n,Num 0 -> if n>0 then Plus_inf else Minus_inf
    | Num n1,Num n2 -> Num (n1/n2) ;;
```

[28]: val div : int_ext -> int_ext -> int_ext = <fun>

L'implementazione dell'operazione di divisione è più complicata e critica dell'implementazione della somma. I casi da considerare sono di più, e soprattutto la divisione è un'operazione che può generare `NaN`,`Plus_inf` e `Minus_inf` a partire da operandi interi "normali"

[29]:
```
div (Num 3) (Num 0) ;;
```

[29]: - : int_ext = Plus_inf

3/0

[30]:
```
sum (Num 5) (div (Num 3) (Plus_inf)) ;;
```

[30]: - : int_ext = Num 5

$5 + (3 / +\infty)$

Tipi opzione e tipi polimorfi

Un esempio di tipo variant molto utilizzato (e *built-in* in OCaml) è il tipo opzione

- consente di specificare un valore che può essere assente
- usa i costruttori `Some` e `None`

[31]:
```
Some 4;;
```

[31]: - : int option = Some 4

[32]:
```
None ;;
```

[32]: - : 'a option = None

E' imporante osservare il tipo di un valore di tipo opzione. Nel primo caso abbiamo `int option` (che in qualche modo significa "intero opzionale"), mentre nel secondo caso abbiamo `'a option`, in quanto il costruttore `None` è lo stesso qualunque sia il tipo che stiamo estendendo.

E' lo stesso principio di funzionamento già visto per le liste... la lista `[4]` ha tipo `int list`, mentre la lista vuota `[]` ha tipo `'a list`.

I tipi opzione consentono di definire funzioni che restituiscono `None` in casi particolari

[33]:
```
let rec massimo lis =
    match lis with
    | [] -> None
    | x::lis' ->  match massimo lis' with
                    | None -> Some x
```

```
                        | Some max -> if x>max then Some x
                                        else Some max ;;
```

[33]: `val massimo : 'a list -> 'a option = <fun>`

La funzione `massimo` è polimorfa, in quanto l'unica operazione che viene eseguita sugli elementi della lista che prende come parametro è l'operazione di confronto > (che si applica a valori di qualunque tipo).

Questa funzione restituisce `None` in caso di lista vuota, in quanto il massimo di una lista priva di elementi non è definito.

[34]:
```
massimo [3;4;2] ;;
massimo ["albero";"cane";"bambino"] ;;
massimo [] ;;
```

[34]: `- : int option = Some 4`

[34]: `- : string option = Some "cane"`

[34]: `- : 'a option = None`

I tipi opzione sono (pre-)definiti in questo modo:

[35]:
```
type 'a option =
    | Some of 'a
    | None ;;
```

[35]: `type 'a option = Some of 'a | None`

questo esempio mostra come sia possibile definire *tipi polimorfi*

- usando le variabili di tipo `'a`, `'b`, ...

Le variabili di tipo sono utilizzate come veri e propri parametri nella definizione del tipo (*polimorfismo parametrico*).

3.7 Tipi record e variant insieme

Ovviamente, si possono creare tipi strutturati combinando a piacere record, variant, tuple, ecc... mostriamo giusto un esempio, per completezza, di punto multidimensionale che può rappresentare un punto nel piano (2 dimensioni) o nello spazio (3 dimensioni) usando diversi costruttori. Le coordinate del punto, in ognuno dei due casi, sono raccolte in un record:

[36]:
```
type punto_multidimensionale =
    | DueDim of {x: float; y: float; }
    | TreDim of {x: float; y: float; z: float; }
```

[36]:
```
type punto_multidimensionale =
    DueDim of { x : float; y : float; }
  | TreDim of { x : float; y : float; z : float; }
```

[37]:
```
let p1 = DueDim {x=10.; y=10.} ;;
let p2 = TreDim {x=4.; y=6.; z=8.} ;;
```

[37]:
```
val p1 : punto_multidimensionale = DueDim {x = 10.; y = 10.}
```

[37]:
```
val p2 : punto_multidimensionale = TreDim {x = 4.; y = 6.; z = 8.}
```

3.8 Tipi ricorsivi

Un'importante generalizzazione dei tipi unione sono i *tipi ricorsivi*

- sono tipi variant definiti in termini di se stessi

Vediamoli usati per definire il tipo di una lista di interi:

[38]:
```
type lista_di_int =
    | Nil
    | Elem of int * lista_di_int ;;

let lst = Elem (3 ,Elem (4, Elem (6,Nil))) ;;
```

[38]:
```
type lista_di_int = Nil | Elem of int * lista_di_int
```

[38]:
```
val lst : lista_di_int = Elem (3, Elem (4, Elem (6, Nil)))
```

Una lista di interi è fatta di elementi di tipo `int` concatenati l'uno all'altro. Ogni elemento, quindi, deve prevedere anche un "riferimento" all'elemento successivo. L'ultimo elemento della lista avrà riferimento nullo (o meglio, a un elemento nullo), in quanto non esiste un successivo.

Il tipo `lista_di_int` deve quindi essere il tipo di una qualunque lista di interi (indipendentemente dalla sua lunghezza). Per questo è definito ricorsivamente...

Abbiamo due casi:

- la lista è vuota: possiamo rappresentarla con un singolo elemento nullo che rappresenteremo con il costruttore `Nil`
- la lista non è vuota: allora esiste un primo elemento che rappresenteremo con il costruttore `Elem`. Tale elemento prevederà un intero e un riferimento all'elemento successivo. L'elemento successivo, però, altri non è che il primo elemento di un'altra lista: la lista di tutti gli elementi che seguono l'elemento corrente. Qundi possiamo vedere il riferimento al prossimo elemento come un riferimento ad un'altra lista dello stesso tipo di quella che stiamo descrivendo. Per questo motivo al costruttore `Elem` è associato il tipo `int * lista_di_int`.

In un valore di tipo `lista_di_interi` in realtà non abbiamo dei veri e propri riferimenti tra gli elementi. Dal momento che la ricorsione è usata all'interno di una coppia `int * lista_di_int`, la concatenazione degli elementi sarà realizzata tramite *annidamento* di coppie, come negli esempi mostrati sopra.

Ora possiamo definire una funzione che corrisponde all'operatore `::` delle liste di OCaml

[39]:
```
let cons x lis =
    Elem (x,lis) ;;

cons 5 lst;;
```

[39]:
```
val cons : int -> lista_di_int -> lista_di_int = <fun>
```

[39]: - : lista_di_int = Elem (5, Elem (3, Elem (4, Elem (6, Nil))))

E possiamo definire altre operazioni su `lista_di_int`:

[40]:
```
let rec somma_lista lis =
    match lis with
    | Nil -> 0
    | Elem (x,lis') -> x + somma_lista lis' ;;

somma_lista ( Elem (3 ,Elem (4, Elem (6,Nil))));;
```

[40]: val somma_lista : lista_di_int -> int = <fun>

[40]: - : int = 13

Il tipo built-in `'a list`, in effetti, è un caso particolare di variant ricorsivo polimorfo

[41]:
```
type 'a my_list =
   | Empty                        (* corrisponde a [] *)
   | Cons of 'a * 'a my_list ;;   (* corrisponde a _::_ *)
```

[41]: type 'a my_list = Empty | Cons of 'a * 'a my_list

Il tipo `'a my_list` qui presentato è una reimplementazione del tipo `'a list` di OCaml. La mostriamo solo per far vedere che `'a list` è effettivamente a tutti gli effetti un tipo variant ricorsivo polimorfo.

[42]:
```
Empty ;;
Cons (3, Cons (4, Cons (1,Empty))) ;; (* corrisponde a 3::4::1 *)
Cons ("a", Cons ("b",Empty)) ;;       (* corrisponde a "a"::"b"::[] *)
```

[42]: - : 'a my_list = Empty

[42]: - : int my_list = Cons (3, Cons (4, Cons (1, Empty)))

[42]: - : string my_list = Cons ("a", Cons ("b", Empty))

Per semplificare la lettura, OCaml usa una sintassi ad-hoc per i valori di tipo `'a list`, che conosciamo bene: `[x1;x2;...]`

Tipi ricorsivi e… Alberi

Grazie ai tipi ricorsivi è semplice definire alberi

L'idea è esattamente quella già illustrata nell'esempio precedente delle liste di interi. Descriviamo ad esempio il tipo degli alberi binari con valori interi, `albero_bin`. Andiamo a descrivere le varie tipologie di elementi (in questo caso nodi intermedi o foglie) tramite costruttori diversi, e nel caso dei nodi intermedi (che sono collegati a sottoalberi) andiamo a riferirci ricorsivamente al tipo `albero_bin`.

[43]:
```
type albero_bin =
  | Nodo of int*albero_bin*albero_bin
  | Foglia of int ;;
```

[43]: type albero_bin = Nodo of int * albero_bin * albero_bin | Foglia of int

Definiamo un valore di tipo `albero_bin` un passo alla volta… dalle foglie alla radice:

[44]:
```
let n1 = Foglia 4 ;;
let n2 = Foglia 6 ;;
let n3 = Nodo (2,n1,n2) ;;
let n4 = Foglia 8 ;;
let n5 = Nodo (5,n3,n4) ;;
```

[44]: val n1 : albero_bin = Foglia 4

[44]: val n2 : albero_bin = Foglia 6

[44]: val n3 : albero_bin = Nodo (2, Foglia 4, Foglia 6)

[44]: val n4 : albero_bin = Foglia 8

[44]: val n5 : albero_bin = Nodo (5, Nodo (2, Foglia 4, Foglia 6), Foglia 8)

Vediamo alcune operazioni su alberi:

[45]:
```
let rec visita_ant a =
    match a with
    | Foglia v -> [v]
    | Nodo (v,sx,dx) -> v::((visita_ant sx)@(visita_ant dx)) ;;

visita_ant n5 ;;
```

[45]: val visita_ant : albero_bin -> int list = <fun>

[45]: - : int list = [5; 2; 4; 6; 8]

[46]:
```
let rec somma_albero a =
    match a with
    | Foglia v -> v
    | Nodo (v,sx,dx) -> v + somma_albero sx + somma_albero dx ;;

somma_albero n5 ;;
```

[46]: val somma_albero : albero_bin -> int = <fun>

[46]: - : int = 25

Tipi ricorsivi e… Abstract Syntax Tree (AST)

Tipo di albero utile per questo corso:

- gli alberi di sintassi astratta (*Abstract Syntax Tree - AST*)

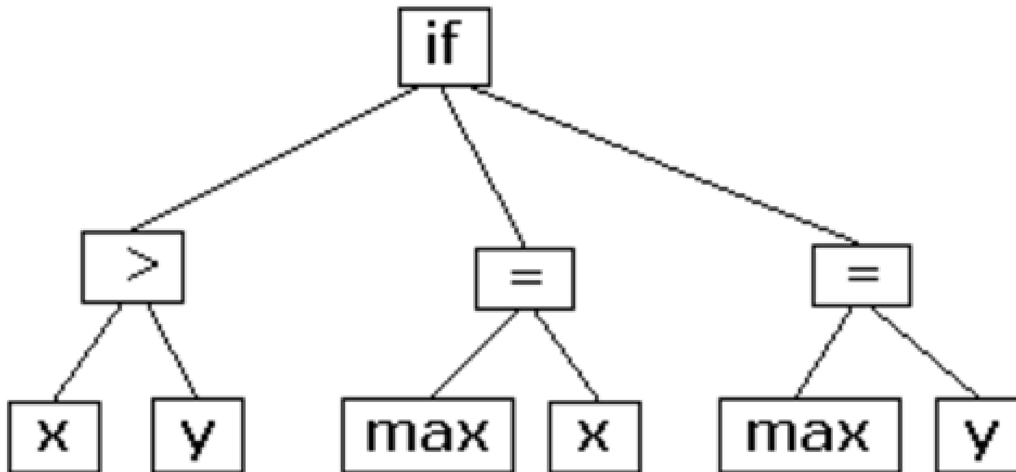
Risultato del *parsing* di un linguaggio formale:

- rappresentano la struttura sintattica di un programma/espressione/termine appartenente a un certo linguaggio
- linguaggio definito tramite una grammatica (es. formato BNF)

In JavaScript:

```
C::= ... | if (E) C [else C] | ...
```

```
if (x>y) max=x else max=y
```



Esempio: le espressioni aritmetiche (su interi)

```
Exp ::= n | Exp op Exp | -Exp | (Exp)
op ::= + | - | * | / | %
```

La rappresentazione come albero di sintassi astratta si basa sulla definizione di un tipo variant con un costruttore per ogni produzione (caso) della grammatica in formato BNF. Possiamo non rappresentare le parentesi: le parentesi servono infatti solo per capire quale operatore si applichi per primo, e questo viene rappresentato dalla posizione dei corrispondenti nodi nell'AST.

[47]:
```
type op = Add | Sub | Mul | Div | Mod ;;
type exp =
    | Val of int
    | Op of op*exp*exp
    | UMin of exp ;;
```

[47]:
```
type op = Add | Sub | Mul | Div | Mod
```

[47]:
```
type exp = Val of int | Op of op * exp * exp | UMin of exp
```

Ogni costruttore è associato a un tipo che rappresenta le informazioni espresse dalla corrispondente produzione nella grammatica. Ad esempio, il costruttore `Val`, che rappresenta la produzione `Exp ::= n` è associato al tipo `int` (per poter memorizzare `n`), mentre il costruttore `Op`, che rappresenta la produzione `Exp ::= Exp op Exp`, è associato al tipo `op*exp*exp` in modo da poter memorizzare il simbolo dell'operatore e le due espressioni a cui tale operatore si applica (che sono quindi sottoalberi dell'AST).

Sintassi astratta:

```
3 * 7 - 5
```

[48]:
```
let exp1 = Op (Sub, (Op (Mul, Val 3, Val 7)), Val 5) ;;
```

[48]:
```
val exp1 : exp = Op (Sub, Op (Mul, Val 3, Val 7), Val 5)
```

```
-3 * (7-5)
```

[49]:
```
let exp2 = Op (Mul, UMin (Val 3), (Op (Sub, Val 7, Val 5))) ;;
```

[49]:
```
val exp2 : exp = Op (Mul, UMin (Val 3), Op (Sub, Val 7, Val 5))
```

Per fare un paio di esempi di uso degli AST rappresentati come tipi algebrici in OCaml, vediamo una funziona che trasforma l'AST di una espressione in una stringa, e una funzione che calcola il risultato (valuta) una espressione.

Rappresentazione come stringa:

[50]:
```
let symbol o =
    match o with
    | Add -> "+" | Sub -> "-" | Mul -> "*" | Div -> "/" | Mod -> "%" ;;
let rec to_string e =
    match e with
    | Val n -> string_of_int n
    | Op (o,e1,e2) -> "("^ (to_string e1) ^ (symbol o) ^ (to_string e2) ^")"
    | UMin e' -> "(-"^(to_string e')^")" ;;
```

[50]:
```
val symbol : op -> string = <fun>
```

[50]:
```
val to_string : exp -> string = <fun>
```

[51]:
```
to_string exp1 ;;
to_string exp2 ;;
```

[51]:
```
- : string = "((3*7)-5)"
```

[51]:
```
- : string = "((-3)*(7-5))"
```

Valutazione delle espressioni:

[52]:
```
let rec eval e =
    match e with
    | Val n -> n
    | Op (o,e1,e2) ->
        (match o with
        | Add -> (eval e1) + (eval e2)
        | Sub -> (eval e1) - (eval e2)
        | Mul -> (eval e1) * (eval e2)
        | Div -> (eval e1) / (eval e2)
        | Mod -> (eval e1) mod (eval e2)
        )
    | UMin e' -> - (eval e') ;;
```

[52]: val eval : exp -> int = <fun>

[53]:
```
eval exp1 ;;
eval exp2 ;;
```

[53]: - : int = 16

[53]: - : int = -6

Capitolo 4

Programmazione imperativa in OCaml (cenni)

4.1 Introduzione

Fino ad ora abbiamo scritto codice secondo il paradigma funzionale:

Abbiamo principalmente definito *funzioni pure*

- Calcolano il risultato sulla base dei propri parametri e delle variabili disponibili nella chiusura

e variabili/strutture dati *immutabili*

- Non è possibile modificarle assegnando nuovi valori

Nella *programmazione imperativa* invece:

- le variabili sono *mutabili* (si possono assegnare a piacimento)
- le funzioni causano *side effect* (possono modificare l'ambiente globale)

Esempi di side effects:

- lettura/assegnamento di una varibile globale (o non locale)
- stampa di un messaggio di output / lettura di un input
- sollevamento di una eccezione

Tutto questo rende le funzioni *impure*

- il loro effetto non è più solo calcolare un risultato
- il risultato può essere influenzato da fattori esterni (variabili globali/input/. . .)

OCaml è concepito principalmente per la programmazione funzionale. . .

- .. ma include anche tutti i *costrutti imperativi* principali

Ne accenneremo brevemente alcuni:

- *array* e *record mutabili*
- *riferimenti*
- *cicli*
- *eccezioni*

Inoltre, OCaml prevede la possibilità di definire *classi e oggetti* in pieno stile object-oriented

- questo per il momento non lo vediamo

4.2 Array

Gli array in OCaml:

- contengono tutti elementi dello stesso tipo
- hanno dimensione fissata al momento della creazione (come in C, Java, ...)
- possono essere acceduti tramite indici numerici (non sono associativi)

Definizione di un nuovo array:

- letterale (elencandone gli elementi tra `[|` e `|]`

[1]:
```
let a = [|3;5;2|] ;;
```

[1]:
```
val a : int array = [|3; 5; 2|]
```

- tramite `Array.make` (con parametri la dimensione e il valore iniziale)

[2]:
```
Array.make 10 0;;
```

[2]:
```
- : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

La lunghezza di un array può essere ottenuta tramite `Array.length`

[3]:
```
let n = Array.length a;;
```

[3]:
```
val n : int = 3
```

Accedere agli elementi di un array (con indici in `0,...,(n-1)`):

- in lettura, con la sintassi `.(i)`

[4]:
```
let e = a.(1) ;;
```

[4]:
```
val e : int = 5
```

- in scrittura, con la sintassi `<-`

[5]:
```
a.(1) <- 6 ;;
a;;
```

[5]:
```
- : unit = ()
```

[5]:
```
- : int array = [|3; 6; 2|]
```

Un commento su `<-`

Il costrutto `<-` è un *COMANDO di ASSEGNAMENTO*

- è la prima volta che *modifichiamo* qualcosa!

Non è l'unico costrutto di assegnamento

- ne vedremo altri con sintassi diversa

Il *tipo dei comandi* in OCaml è `unit`

- in realtà non sono comandi, ma *espressioni con side effect*
- simile a `void` in altri linguaggi di programmazione
- descrive le cose che non possono essere valutate a valori

- l'unico valore del tipo `unit` è `()`

4.3 Record mutabili

E' possibile rendere mutabili uno o più campi tramite il modificatore `mutable`

[6]:
```
type persona =
{
    nome: string;
    cognome: string;
    mutable eta: int;
}
```

[6]:
```
type persona = { nome : string; cognome : string; mutable eta : int; }
```

[7]:
```
let mario = {nome="mario"; cognome="rossi"; eta=30 } ;;
```

[7]:
```
val mario : persona = {nome = "mario"; cognome = "rossi"; eta = 30}
```

Anche in questo caso per l'assegnamento si usa `<-`

[8]:
```
mario.eta <- 31 ;;
mario ;;
```

[8]:
```
- : unit = ()
```

[8]:
```
- : persona = {nome = "mario"; cognome = "rossi"; eta = 31}
```

[9]:
```
let invecchia p =
    p.eta <- p.eta + 1;;

invecchia mario;;
mario ;;
```

[9]:
```
val invecchia : persona -> unit = <fun>
```

[9]:
```
- : unit = ()
```

[9]:
```
- : persona = {nome = "mario"; cognome = "rossi"; eta = 32}
```

4.4 Riferimenti (refs)

Oltre ad array e record mutabili, è possibile definire anche singole variabili mutabili usando il tipo `ref`

Ad esempio:

[10]:
```
let x = ref 12 ;;
```

```
[10]: val x : int ref = {contents = 12}
```

Si definisce la variabile x che contiene un *riferimento*

- il riferimento punta a un record con un solo campo mutabile `contents` inizializzato con il valore passato

E' possibile accedere alla variabile `ref` come record

```
[11]: x.contents ;;
      x.contents <- 13;;
```

```
[11]: - : int = 12
```

```
[11]: - : unit = ()
```

Ma in realtà c'è una *sintassi specifica*, più semplice:

```
[12]: !x ;;            (* corrisponde a x.contents *)
      x := 14 ;;       (* corrisponde a x.contents <- 14 *)
```

```
[12]: - : int = 13
```

```
[12]: - : unit = ()
```

L'operazione `!` è detta *dereferenziazione*

Con `ref` si creano effettivamente dei riferimenti

- dei puntatori ad aree di memoria

Copiando il riferimento in una nuova variabile, entrambe le variabili punteranno alla stessa area di memoria

```
[13]: let x = ref 0;;
      let y = x;;
      y:=100;;
      !x;;
```

```
[13]: val x : int ref = {contents = 0}
```

```
[13]: val y : int ref = {contents = 0}
```

```
[13]: - : unit = ()
```

```
[13]: - : int = 100
```

Modificando y anche x risulta modificata

4.5 Sequenze di comandi e cicli

Una sequenza di comandi/espressioni può essere definita tramite il separatore ;

[14]:
```
let x = ref 0;;
let y = x;;
x := !x+1 ; y := !y+1 ; (!x,!y) ;; (* sequenza di espressioni *)
```

[14]:
```
val x : int ref = {contents = 0}
```

[14]:
```
val y : int ref = {contents = 0}
```

[14]:
```
- : int * int = (2, 2)
```

Le espressioni vengono valutate una dopo l'altra e l'ultima fornisce il risultato finale

In una sequenza di espressioni `E1 ; E2 ; .... ; En`:

- tutte le espressioni precedenti all'ultima devono (dovrebbero...) avere tipo `unit`
- le espressioni `E1 ; E2 ; .... ; E(n-1)` causano solo side effects (ad esempio, modificando le strutture dati mutabili)
- l'espressione `En` calcola il risultato vero e proprio

[15]:
```
x := !x+1 ; y := !y+1 ; (!x,!y) ;;
```

[15]:
```
- : int * int = (4, 4)
```

[16]:
```
4+3 ; 5+2 ;;
```

```
File "[16]", line 1, characters 0-3:
1 | 4+3 ; 5+2 ;;
    ^^^
Warning 10: this expression should have type unit.
File "[16]", line 1, characters 0-3:
1 | 4+3 ; 5+2 ;;
    ^^^
Warning 10: this expression should have type unit.
```

[16]:
```
- : int = 7
```

Sfruttando la funzione `print_endline` che stampa una stringa sulla console, il sequenziamento ; può essere utile per stampare *messaggi di debug*

[17]:
```
let abs x =
    if x>0 then (
        print_endline "ramo THEN" ;
        x
    )
    else (
        print_endline "ramo ELSE" ;
        -x
    );;

abs (-5) ;;
```

[17]: val abs : int -> int = <fun>

```
ramo ELSE
```

[17]: - : int = 5

[18]:
```
let abs x =
    if x>0 then begin
        print_endline "ramo THEN" ;
        x
    end
    else begin
        print_endline "ramo ELSE" ;
        -x
    end;;

abs (-5) ;;
```

[18]: val abs : int -> int = <fun>

```
ramo ELSE
```

[18]: - : int = 5

NOTA: le parentesi tonde e `begin...end` sono modi alternativi per creare un "blocco"

NOTA: `;` vs `in`

Viene la tentazione di usare `;` per separare dichiarazioni di variabili dal loro uso...

[46]:
```
let f n = n+1 ;; (* funzione giocattolo, giusto per l'esempio *)
let m = 10 ; f(m) ;;
```

[46]: val f : int -> int = <fun>

```
File "[46]", line 2, characters 8-10:
2 | let m = 10 ; f(m) ;;
            ^^
Warning 10: this expression should have type unit.
```

```
File "[46]", line 2, characters 15-16:
2 | let m = 10 ; f(m) ;;
                   ^
Error: Unbound value m
```

invece bisogna usare `in`

[20]:
```
let m = 10 in f(m) ;;
```

[20]: - : int = 11

Cicli `for`

La sintassi del comando `for` è la seguente:

```
for <variable> = <start> to <end> do
  ...
done
```

oppure

```
for <variable> = <start> downto <end> do
  ...
done
```

dove `<start>` e `<end>` sono valori interi entro cui far variare la `<variabile>`

Esempio giocattolo:

[21]:
```
for i = 1 to 10 do
    print_endline (string_of_int i)
done ;;
```

```
1
2
3
4
5
6
7
8
9
10
```

[21]:
```
- : unit = ()
```

Esempio: media dei valori di un array

[22]:
```
let media arr =
    let s = ref 0 in
    for i = 0 to (Array.length arr)-1 do
        s := !s + arr.(i)
    done ;
    !s / Array.length arr ;;

media [|4;6;8|] ;;
```

[22]:
```
val media : int array -> int = <fun>
```

[22]:
```
- : int = 6
```

Cicli `while`

La sintassi del comando `while` è la seguente:

```
while <condizione> do
  ...
done
```

dove <condizione> è una espressione di tipo `boolean`

[23]:
```
let x=ref 10 in
while !x>0 do
    print_endline (string_of_int !x) ;
    x := !x/2
done
```

```
10
5
2
1
```

[23]:
```
- : unit = ()
```

Esempio: restituisce la posizione del primo valore negativo in un array, se presente

[24]:
```
let primo_negativo arr =
    let pos = ref 0 in
    let trovato = ref false in
    while !pos < Array.length arr && not !trovato do
        if arr.(!pos)<0
            then trovato := true
            else pos := !pos + 1
    done;
    if (!trovato)
        then Some !pos
        else None ;;

primo_negativo [| 3; 4; -1; 6; -3; -6 |] ;;
```

[24]:
```
val primo_negativo : int array -> int option = <fun>
```

[24]:
```
- : int option = Some 2
```

Commenti sui cicli…

OCaml *non incoraggia* l'uso dei cicli:

- la sintassi non è molto *friendly*
- non è possibile interromperli brutalmente (non esiste `break` o `return`)
- il `for` non è flessibile sull'uso degli indici (no incremento di 2 per volta)

D'altra parte *incoragga* fortemente l'uso di un approccio funzionale

- ottimizzazione delle funzioni *tail-recursive*
- funzioni `iter`, `for_all`,`exists`,`map`,`filter`,`fold_right`,`fold_left` in `List` e `Array` per "ciclare" su liste e array

4.6 Eccezioni

Le eccezioni sono un modo usato in molti linguaggi di programmazione per gestire le situazioni anomale e di errore nei programmi

[25]:
```
3 / 0 ;;
```

```
Exception: Division_by_zero.
Raised by primitive operation at unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Esistono vari tipi di eccezioni predefinite (es. `Division_by_zero`) e si possono definire le proprie * specificando un costruttore e, facoltativamente, un tipo (come nei tipi variant)

[26]:
```
exception Lista_vuota ;;
exception Stringa_errata of string ;;
```

[26]:
```
exception Lista_vuota
```

[26]:
```
exception Stringa_errata of string
```

Le eccezioni si sollevano con `raise`…

[27]:
```
raise Lista_vuota;;
```

```
Exception: Lista_vuota.
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

[28]:
```
raise (Stringa_errata "test") ;;
```

```
Exception: Stringa_errata "test".
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

… e possono essere usate, ad esempio, per interrompere una funzione in caso di errore

[29]:
```
let minimo lis =
    let rec minimo_ric m lis' =
        match lis' with
        | [] -> m
        | x::lis' -> minimo_ric (if x<m then x else m) lis'
    in
    match lis with
    | [] -> raise Lista_vuota
    | x::lis' -> minimo_ric x lis'
;;
```

[29]:
```
val minimo : 'a list -> 'a = <fun>
```

[30]:
```
minimo [4;3;5;6;2;9] ;;
```

[30]:
```
- : int = 2
```

```
[31]: minimo [] ;;
```

```
Exception: Lista_vuota.
Raised at file "[29]", line 8, characters 18-29
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Altro esempio: giorni della settimana

```
[32]: let giorno_num g =
          match g with
          | "lun" -> 1 | "mar" -> 2 | "mer" -> 3 | "gio" -> 4
          | "ven" -> 5 | "sab" -> 6 | "dom" -> 7
          | _ -> raise (Stringa_errata g) ;;
```

```
[32]: val giorno_num : string -> int = <fun>
```

```
[33]: giorno_num "gio" ;;
```

```
[33]: - : int = 4
```

```
[34]: giorno_num "lunedì" ;;
```

```
Exception: Stringa_errata "lunedì".
Raised at file "[32]", line 5, characters 17-35
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Inoltre, è possibile intercettare e gestire eccezioni con il costrutto `try ... with`.

La sintassi (simile a quella del pattern matching) è la seguente:

```
try <espressione> with
| <pat1> -> <esp1>
| <pat2> -> <esp2>
...
| <patN> -> <espN>
```

e la semantica è che se `<espressione>` causa un'eccezione che fa match con il pattern `<patI>`, viene valutata l'espressione `<espI>`

- *NOTA:* non c'è controllo di esaustività dei pattern

Nel costrutto `try ... with` non avrebbe senso testare l'esaustività dei pattern, in quanto, essendo le eccezioni personalizzabili, le possibili eccezioni che possono essere sollevate sono infinite. L'unico modo per raggiungere l'esaustività sarebbe di includere sempre il pattern wildcard `_` come ultimo caso possibile.

Inoltre, spesso i gestori di eccezioni definiti con il costrutto `try ... with` si preoccupano di catturare una o un gruppo specifico di eccezioni, lasciando che eventuali altre "passino" venendo poi catturate da un controllo più esterno, oppure portando all'interruzione del programma, senza essere gestite internamente.

Per tutti questi motivi un controllo di esaustività di questi pattern sarebbe privo di senso.

Esempio (un po' forzato, si farebbe `List.for_all (fun x -> x>=0) lis`):

[35]:
```
let tutti_positivi lis =
  try
    minimo lis >= 0
  with
  | Lista_vuota -> true ;;
```

[35]:
```
val tutti_positivi : int list -> bool = <fun>
```

[36]:
```
tutti_positivi [3;2;-5];;
```

[36]:
```
- : bool = false
```

[37]:
```
tutti_positivi [];;
```

[37]:
```
- : bool = true
```

La funzione `tutti_positivi` verifica se gli elementi di un array sono tutti positivi andandone a calcolare il minimo con la funzione `minimo` definita in precedenza. Qualora la lista sia vuota, l'eccezione sollevata dalla funzione `minimo` viene catturata e il risultato sarà `true` (Nota: qualunque proprietà è soddisfatta da tutti gli elementi di una lista vuota...)

Altro esempio: trova il minimo (come numero) in una lista di giorni in formato testuale

[47]:
```
let primo_giorno lis =
    try
        Some (minimo (List.map giorno_num lis))    (* giorno_num definita in␣
↪precedenza *)
    with
    | Stringa_errata s -> print_endline ("Stringa errata: "^s) ; None
    | Lista_vuota -> print_endline "Lista vuota" ; None
    | _ -> print_endline "Errore" ; None     (* caso _ aggiunto cattura eventuali␣
↪altre eccezioni... *)
```

[47]:
```
val primo_giorno : string list -> int option = <fun>
```

[39]:
```
primo_giorno ["sab"; "gio"; "ven"] ;;
```

[39]:
```
- : int option = Some 4
```

[40]:
```
primo_giorno ["sabato"; "giovedì"; "venerdì"] ;;
```

```
Stringa errata: sabato
```

[40]:
```
- : int option = None
```

[41]:
```
primo_giorno [] ;;
```

```
Lista vuota
```

[41]:
```
- : int option = None
```

Anche in questo esempio sfruttiamo l'eccezione sollevata da `minimo`. Innanzitutto, la lista di stringhe viene data in pasto alla funzione `List.map` che, applicando la funzione `giorno_num` ad ogni elemento, produce una lista di numeri che rappresentano i vari giorni. A questo punto `minimo` calcola il numero minimo, che viene restituito come tipo `int option`, quindi preceduto dal costruttore `Some`.

Le funzioni `giorno_map` e `minimo` possono sollevare eccezioni. La prima nel caso in cui una stringa sia scritta in modo errato, e la seconda nel caso in cui la lista sia vuota. Entrambe le eccezioni vengono gestite tramite il costrutto `try ... with` visualizzando un messaggio di errore e restituendo `None`. In questo caso, sebbene non sia essenziale, è stato aggiungo anche il pattern _ per cattuare eventuali altre eccezioni (che comunque non dovrebbero verificarsi).

Ocaml fornisce inoltre alcuni altri costrutti per sollevare eccezioni alternativi a `raise`

[42]:
```
failwith "messaggio di errore" ;;
```

```
Exception: Failure "messaggio di errore".
Raised at file "stdlib.ml", line 29, characters 22-33
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

corrisponde a `raise (Failure "messaggio di errore")`, ma più breve

Inoltre, `assert <condizione>` solleva un'eccezione se `<condizione>` è falsa

[43]:
```
let dividi n m = assert (m<>0) ; n/m ;;
```

[43]:
```
val dividi : int -> int -> int = <fun>
```

[44]:
```
dividi 4 2;;
```

[44]:
```
- : int = 2
```

[45]:
```
dividi 4 0;;
```

```
Exception: Assert_failure ("[43]", 1, 17).
Raised at file "[43]", line 1, characters 17-30
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

L'eccezione sollevata da `assert` fornisce indicazioni sulla riga di codice in cui si trova la condizione che è stata violata.