Definizione di nuovi tipi

Un tipo è un insieme di valori.

Per definire un nuovo tipo occorre specificare:

1. un **nome** per il tipo
2. come costruire i valori del tipo, cioè quali sono i **costruttori** del tipo.

I tipi enumerati

Sono tipi costituiti da un insieme finito di valori. Si possono definire semplicemente elencando i valori del tipo.

Ad esempio, il tipo predefinito **bool** è un tipo enumerato.

I valori di un tipo enumerato sono identificati da **costanti**, il modo più semplice di costruire un valore del tipo.

Una costante è dunque un caso particolare di **costruttore**.

Un nuovo tipo viene definito mediante una **dichiarazione di tipo**.

Esempio: il tipo delle quattro direzioni principali

$$\{Su, Giu, Destra, Sinistra\}$$

Dichiarazione del nuovo tipo **direzione**:

```
type direzione = Su | Giu | Destra | Sinistra;;
```

- **type**: parola chiave che introduce la dichiarazione di un nuovo tipo;
- **direzione**: nome del tipo;
- **Su, Giu, Destra, Sinistra**: valori del nuovo tipo.
- I nomi dei costruttori di tipi definiti dal programmatore iniziano sempre con una **lettera maiuscola.**
- Nella dichiarazione, i valori del tipo sono separati dalla barra verticale.

Una dichiarazione di tipo introduce nuovi valori

Prima della dichiarazione, OCaml non conosce i valori **Su, Giu, Destra, Sinistra**.

```
# Destra;;
Characters 0-6:
  Destra;;
  ^^^^^^
Unbound constructor Destra
```

Dopo la dichiarazione:

```
# type direzione = Su | Giu | Destra | Sinistra;;
type direzione = Su | Giu | Destra | Sinistra
# Destra;;
- : direzione = Destra
```

I nuovi valori possono essere utilizzati nella costruzione di altri valori:

```
# [Su; Destra];;
- : direzione list = [Su; Destra]
# [(Giu,1); (Sinistra,2)];;
- : (direzione * int) list = [(Giu, 1); (Sinistra, 2)]
```

Esempio: la rappresentazione delle carte napoletane

Definiamo un tipo per rappresentare i semi e uno per i valori:

```
type seme = Bastoni | Coppe | Denari | Spade;;
type valore = Asso | Due | Tre | Quattro | Cinque
              | Sei | Sette | Fante | Cavallo | Re;;
```

Rappresentiamo una carta del mazzo con una coppia di tipo **valore * seme**

```
# let settebello =(Sette,Denari);;
val settebello : valore * seme = (Sette, Denari)

# let briscola = Spade
  in [(Asso,Denari);(Due,briscola);(Tre,Bastoni)];;
  - : (valore * seme) list = [(Asso, Denari);
                              (Due, Spade); (Tre, Bastoni)]
```

I tipi enumerati sono **tipi con uguaglianza**:

```
# List.mem (Due,Spade) [(Asso,Denari);(Due,Spade);
                        (Tre,Bastoni)];;
- : bool = true
```

Le nuove costanti introdotte sono costruttori del tipo

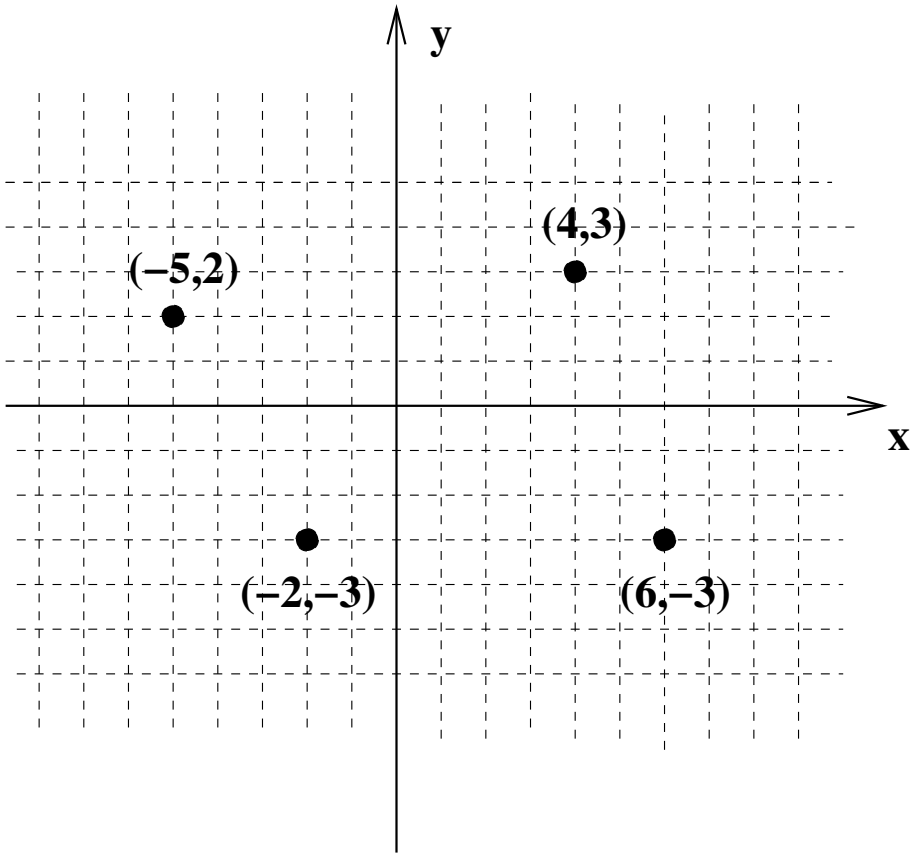
Quindi possono occorrere in un pattern.
E possiamo definire operazioni usando il pattern matching:

```
(* valore : valore -> int *)
let valore = function
     Asso -> 1
   | Due -> 2
   | Tre -> 3
   | Quattro -> 4
   | Cinque -> 5
   | Sei -> 6
   | Sette -> 7
   | Fante -> 8
   | Cavallo -> 9
   | Re -> 10;;

# valore Fante;;
- : int = 8
# (valore Due) + (valore Asso);;
- : int = 3
```

Rappresentazione di posizioni e movimenti su un piano

Vogliamo rappresentare la posizione ed il movimento di un oggetto su un piano.



L'oggetto:

- si trova in un punto del piano, identificato da una coppia (x, y) di interi;
- è rivolto in una delle quattro direzioni (Su, Giu, Destra o Sinistra);
- può spostarsi in avanti di n punti, nella direzione verso cui è rivolto;
- può girare in senso orario, modificando la sua posizione di 90^o.

Obiettivo

Problema: data la posizione (punto del piano (x, y) e una direzione) dell'oggetto e un'azione di spostamento o di cambiamento di direzione, calcolare la nuova posizione dell'oggetto.

Ad esempio:

- se l'oggetto si trova in $(4, 3)$ ed è rivolto verso Destra, quando va avanti di 3 passi si troverà in $(7, 3)$, sempre rivolto verso Destra;
- se l'oggetto si trova in $(-2, -3)$ ed è rivolto verso Giu, quando gira si troverà sempre in $(-2, -3)$, ma rivolto verso Sinistra.

Rappresentazione delle posizioni

Una posizione è identificata da una coppia di coordinate (punto nel piano) e una direzione.

Posizione: punto e direzione

Punto: coppia di interi

Direzione:

```
type direzione = Su | Giu | Destra | Sinistra
```

Possiamo rappresentare le posizioni mediante triple di tipo **int * int * direzione**:

```
type posizione = int * int * direzione
```

Nota bene: non si sta definendo un nuovo tipo, stiamo soltanto dando un altro nome a un tipo già esistente (che OCaml, normalmente, non utilizzerà).

Selettori del tipo posizione

Vai al codice

Rappresentazione delle azioni

Ci sono due tipi di azioni:

- girare di 90^o in senso orario
- andare avanti di n passi (con n intero)

Vogliamo definire un tipo di dati **azione** per rappresentare le azioni.

L'azione di girare può essere rappresentata da una costante: **Gira**.

L'azione di andare avanti di n passi costituisce in realtà tutto un insieme di azioni: andare avanti di 0 passi, andare avanti di 1 passo, andare avanti di 2 passi, ... E anche andare avanti di -1 passo (indietro di 1 passo), avanti di -2 passi (indietro di 2), ...

Per rappresentare le azioni serve una funzione: un **costruttore funzionale** che, applicato a un intero n, riporta il valore che rappresenta "andare avanti di n passi":

Avanti 0, Avanti 1, Avanti 2, Avanti 3, ...
Avanti(-1), Avanti(-2), Avanti(-3), ...

Dichiarazione del tipo azione

```
type azione =  Gira
             | Avanti of int
```

Questa dichiarazione specifica che l'insieme dei valori di tipo **azione** è costituito da:

- la costante **Gira**;
- tutti i valori della forma **Avanti n**, dove **n** è un intero.

Avanti è un **costruttore funzionale** (si comporta come una funzione):

- è di tipo **int -> azione**;
- quando è applicato a un valore di tipo **int**, restituisce (**costruisce**) un valore di tipo **azione**.

La parola chiave **of** introduce il tipo degli oggetti a cui il costruttore si applica per costruire un oggetto di tipo **azione**.

L'insieme (infinito) dei valori del tipo **azione** è costituito da: **Gira, Avanti 0, Avanti 1, Avanti 2, Avanti 3, ..., Avanti (-1), Avanti (-2), Avanti (-3), ...**

I **costruttori** sono parte della descrizione del tipo, determinano il modo canonico di descrivere i **valori** del tipo.

Pattern matching e selettori

Mediante il pattern matching è possibile definire i **selettori** del tipo:

```
(* int_of_act: azione -> int *)
let int_of_act = function
   Avanti n -> n
 | _ -> failwith "int_of_act"
```

Se definissimo il tipo posizione introducendo un nuovo tipo (con il suo costruttore):

```
type posizione = Pos of int * int * direzione
```

Potremmo definirne i selettori come segue:

```
(* xcoord, ycoord : posizione -> int *)
let xcoord (Pos(x,_,_)) = x
let ycoord (Pos(_,y,_)) = y
(* dir : posizione -> direzione *)
let dir (Pos(_,_,d)) = d
```

ATTENZIONE: in questo caso il tipo **posizione** è diverso da **int * int * direzione**

Uso delle parentesi

Qual è l'errore nella definizione seguente?

```
type posizione = Pos of int * int * direzione

# let punto_di Pos(x,y,_) = (x,y);;

Characters 13-16:
  let punto_di Pos(x,y,_) = (x,y);;
               ^^^
The constructor Pos expects 3 argument(s),
but is here applied to 0 argument(s)
```

La soluzione del problema

Funzione principale:

sposta : posizione -> azione -> posizione

sposta pos act: se act ha la forma:

Gira: riporta la posizione che si ottiene girando la direzione di **pos** di 90^o in senso orario

Avanti n: riporta la posizione che si ottiene spostando avanti di **n** punti le coordinate di **pos**

Sottoproblema 1: data una direzione, riportare la direzione che si ottiene girando di 90^o in senso orario.

Sottoproblema 2: data una posizione **(x,y,d)** e un intero *n*, riportare la posizione **(x',y',d)** che si ottiene andando avanti di *n* passi nella direzione indicata da **d**.

Vai al codice

Unione di tipi

```
type number = Int of int
            | Float of float;;

(* sum : number * number -> number *)
let sum = function
    (Int x,Int y) -> Int (x + y)
  | (Int x,Float y) -> Float ((float x) +. y)
  | (Float x,Float y) -> Float (x +. y)
  | (Float x,Int y) -> Float(x +. float y);;
```

In nuovo tipo rappresenta l'**unione disgiunta** dei tipi che sono gli argomenti dei costruttori

$$A \dot{\cup} B = \{(a, 0) \mid a \in A\} \cup \{(b, 1) \mid b \in B\}$$

I costruttori marcano gli elementi, rendendo riconoscibile la loro provenienza.

Tipi definiti ricorsivamente

Definizione induttiva dei numeri naturali:

(i) $0 \in \mathbb{N}$

(ii) per ogni $n \in \mathbb{N}$, $succ(n) \in \mathbb{N}$

(iii) gli unici elementi di $\mathbb{N}$ sono quelli che si ottengono mediante (i) e (ii).

Definizione ricorsiva di un nuovo tipo per la rappresentazione dei naturali:

type nat = Zero | Succ of nat

I valori di tipo **nat** sono: **Zero, Succ Zero, Succ(Succ Zero), Succ(Succ(Succ Zero)), ...**

Sui tipi di dati definiti induttivamente si possono definire operazioni ricorsivamente:

```
(* int_of_nat : nat -> int *)
(* int_of_nat n = valore intero corrispondente a n *)
let rec int_of_nat = function
    Zero -> 0
  | Succ n -> succ(int_of_nat n)
# int_of_nat (Succ (Succ (Succ (Succ Zero))));;
- : int = 4
```

L'operazione somma di numeri naturali

Si può definire ricorsivamente sul primo argomento:

```
(* somma : nat * nat -> nat *)
let rec somma (n,m) =
  match n with
     Zero -> m
   | Succ k -> Succ(somma(k,m));;
```

```
# let k = somma(Succ(Succ Zero), Succ(Succ(Succ Zero)));;
val k : nat = Succ (Succ (Succ (Succ (Succ Zero))))

# int_of_nat k;;
- : int = 5
```

Definizione di costruttori di tipi

La definizione di un tipo può essere parametrica:

- con variabili di tipo
- con costruttori polimorfi

type 'a mylist = Nil | Cons of 'a * 'a mylist

Possiamo definire un costruttore di tipi (un tipo polimorfo) per le matrici di interi, matrici di stringhe, matrici di liste di booleani, ...

type 'a matrix = Mat of 'a list list

matrix è un **costruttore di tipi**

Il costruttore **Mat: 'a list list -> 'a matrix** è polimorfo

Mat [[true;false];[false;true]] : bool matrix
Mat [[1;2;3];[4;5]] : int matrix

Il tipo 'a option: un'alternativa all'uso delle eccezioni

type 'a option = None | Some of 'a

Una funzione parziale di tipo **'a -> 'b** può essere implementata da una funzione di tipo **'a -> 'b option**.

Utile quando non si vuole propagazione degli errori.

```
(* new_assoc : ('a * 'b) list -> 'a -> 'b option *)
let rec new_assoc lst x =
   match lst with
      [] -> None
    | (k,v)::rest -> if x=k then Some v
                     else new_assoc rest x
```

Vai al codice