Uso delle liste per rappresentare tipi astratti di dati

tipi concreto: un particolare dato definito con un tipo linguaggio
tipo astratto: è una struttura matematica

Il tipo astratto di dati "dizionario"

Il tipo astratto "dizionario" è una collezione di elementi, ciascuno dei quali è costituito da una coppia **(chiave, valore)**.
In un dizionario, elementi distinti hanno chiavi distinte.

Un tipo astratto di dati viene caratterizzato dall'**insieme di operazioni** che è possibile fare su tali dati.

Le operazioni sul tipo dizionario sono:

ricerca: dato un dizionario e una chiave, trovare il valore associato alla chiave data

inserimento: dato un dizionario e una coppia `(chiave,valore)`, aggiungere tale coppia al dizionario (eventualmente sostituendo l'elemento con la stessa chiave, se esiste)

cancellazione: dato un dizionario e una chiave, cancellare dal dizionario l'elemento con la chiave data, se esiste.

I dizionari si possono implementare in OCaml mediante liste associative.

Liste associative

Una **lista associativa** è un modo di implementare un dizionario, cioè una lista di coppie (associazioni tra chiavi e valori).

Una lista associativa è un **tipo concreto**:

$$(\text{'a} \times \text{'b})\ \textbf{list}$$

Le chiavi devono essere di un tipo con uguaglianza.

Ad esempio la lista

```
[("pippo",100);("pluto",50);("topolino",30)]
```

è di tipo **(string * int) list** e rappresenta un dizionario con chiavi di tipo **string** e valori associati di tipo **int**.

qualunque tipi ma non si possono mettere funzioni

se si trova una coppia con la stessa chiave, il valore associato alla chiave e quella che nella lista si trova prima dell'altra

La ricerca

Problema: dato un dizionario, rappresentato dalla lista associativa **assoc_list**, e una chiave **k**, riportare il valore associato a **k** in **assoc_list**, se esiste, un errore altrimenti.

assoc: 'a -> ('a * 'b) list -> 'b

```
exception NotFound

(* assoc: 'a -> ('a * 'b) list -> 'b *)
let rec assoc k = function
      [] -> raise NotFound
    | (k1,v)::rest -> if k = k1 then v
                      else assoc k rest
```

Nel modulo List: **List.assoc**, solleva l'eccezione predefinita **Not_found** quando la ricerca fallisce

Esercizio

Si consideri la seguente dichiarazione:

```
let rec assoc k lista = function
      [] -> raise NotFound
    | (k1,v)::rest -> if k = k1 then v
                      else assoc k rest
```

La dichiarazione non è corretta.

Quale errore viene segnalato da OCaml e perché?

Inserimento

Problema: data una chiave **k**, un valore **v** e una lista associativa **assoc_list**, riportare la lista che si ottiene inserendo la coppia **(k,v)** in **assoc_list** – sostituendo (o sovrascrivendo) l'eventuale elemento già esistente con chiave **k**

inserisci: 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list

La proprietà fondamentale dell'inserimento è in relazione con la ricerca: si deve avere che

assoc k (inserisci k v assoclist) = v

Sostiutire effettivamente un'eventuale coppia **(k,v')** già presente in **assoc_list** è "costoso": occorre controllare tutta la lista.

Poiché la ricerca riporta il primo valore trovato associato alla chiave della ricerca, è sufficiente inserire la nuova coppia prima delle altre.

```
let inserisci k v assoc_list =
      (k,v)::assoc_list
```

Cancellazione

Problema: data una chiave **k** e una lista associativa **assoc_list**, riportare la lista che si ottiene cancellando da **assoc_list** tutte le coppie con chiave **k** (potrebbero essercene diverse). Se non c'è nessuna coppia con chiave **k**, riportare **assoc_list** stessa

cancella : 'a -> ('a * 'b) list -> ('a * 'b) list

```
let rec cancella k = function
    [] -> []
  | (k',v)::rest ->
      if k=k' then cancella k rest
      else (k',v)::cancella k rest
```

Rappresentazione di tipi astratti di dati (I)

Tio astratto: insieme di elementi in cui vengono definiti delle operazioni

Un tipo astratto di dati è costituito da un insieme di oggetti **A** e un insieme di operazioni su tali oggetti.

Per rappresentare un tipo astratto **A**:

1. Determinare un tipo concreto **T** tale che:
   1. ogni oggetto di **A** abbia almeno un "rappresentante" in **T**;
   2. elementi distinti di **A** abbiano rappresentanti distinti in **T** (non vengono confusi);
2. Implementare ogni operazione F su **A** mediante un'operazione P su **T** in modo tale che la rappresentazione "conservi le operazioni":

   se $v_1, ..., v_n$ sono valori di **T** che rappresentano, rispettivamente, gli oggetti $a_1, ..., a_n$ di **A**, allora

   $$P(v_1, ..., v_n) \text{ è un rappresentante di } F(a_1, ..., a_n)$$

   Quindi "operare sui rappresentanti" e poi determinare l'oggetto rappresentato dal valore ottenuto fornisce lo stesso risultato che si ottiene operando sugli oggetti di **A** stessi.

Rappresentazione di tipi astratti di dati (II)

Siano $\mathcal{A} = (A, f_1, \ldots, f_n)$ e $\mathcal{B} = (B, g_1, \ldots, g_n)$ due strutture algebriche dello stesso tipo (per ogni $i = 1 \ldots n$, f_i e g_i hanno lo stesso numero di argomenti).

Una **rappresentazione** di $\mathcal{A}$ (tipo astratto) in $\mathcal{B}$ (tipo concreto) è un'applicazione (funzione parziale)

$$\varphi : \mathcal{B} \to \mathcal{A}$$
($\varphi(b) = a$: b rappresenta a,
quindi ogni $b \in B$ rappresenta al massimo un $a \in A$)

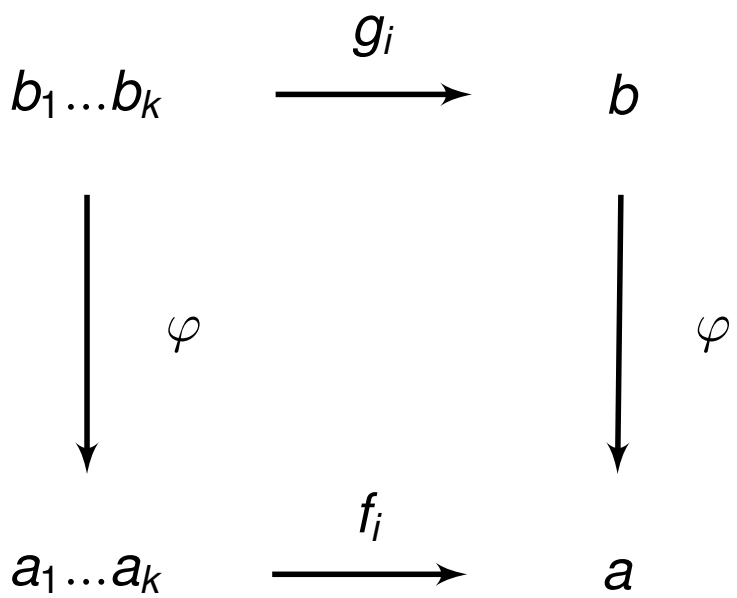
tale che:

- per ogni $a \in A$ esiste $b \in B$ tale che $\varphi(b) = a$ (φ è suriettiva);
- per ogni $i = 1 \ldots n$, e per ogni $b_1, \ldots, b_n$ di B per cui φ è definita, si ha:

$$\varphi(g_i(b_1, \ldots, b_n)) = f_i(\varphi(b_1), \ldots, \varphi(b_n))$$

(φ è un omomorfismo)

Rappresentazione di tipi astratti di dati (III)



(Leggere il paragrafo 2.3.3 delle dispense – o il 5.1.2 del libro di testo)

Rappresentazione di insiemi finiti mediante liste (I)

Tipo astratto “insieme finito di elementi dello stesso tipo”, con le operazioni:

$$\in, \cup, \cap \text{ e } \setminus$$

Tipo concreto: liste. Una lista **lst** rappresenta l’insieme che contiene esattamente gli elementi di **lst**.

- Ogni insieme finito di elementi di tipo **’a** ha almeno un “rappresentante” in **’a list**. (Quanti?)
- Insiemi distinti hanno rappresentanti distinti in **’a list**: ogni lista rappresenta un unico insieme.

Si devono definire funzioni che rappresentino le operazioni sul tipo astratto:

mem: applicata a un elemento x e la rappresentazione di un insieme S, determina se $x \in S$ (è un predicato).

union: applicata a due liste che rappresentano insiemi S_1 e S_2, riporta una rappresentazione di $S_1 \cup S_2$.

intersect: applicata a due liste che rappresentano insiemi S_1 e S_2, riporta una rappresentazione di $S_1 \cap S_2$.

setdiff: applicata a due liste che rappresentano insiemi S_1 e S_2, riporta una rappresentazione di $S_1 \setminus S_2$.

Rappresentazione di insiemi finiti mediante liste (II)

Ad esempio:

[ ]	rappresenta	$\emptyset$
[1;2;3;4]	rappresenta	$\{1, 2, 3, 4\}$
[4;3;2;1]	rappresenta	$\{1, 2, 3, 4\}$
[3;2;2;2;2;1;2;3;4;2]	rappresenta	$\{1, 2, 3, 4\}$

Per evitare di costruire liste eccessivamente lunghe, si stabilisce che **solo liste senza ripetizioni** rappresentano insiemi: di questo si deve tener conto quando si implementano le operazioni.

Vai al codice

Backtracking

Una delle tecniche più generali per la progettazione di algoritmi

Problemi di ricerca di una soluzione che soddisfi determinate condizioni, e che si possa costruire incrementalmente

Esempio: i candidati sono sequenze $x_1, ..., x_n$ con $x_i \in S$

Approccio "forza bruta" (enumerazione): generare ad una ad una tutte le sequenze $x_1, ..., x_n$ e controllare se soddisfano le condizioni

Backtracking: costruire la sequenza aggiungendo un elemento alla volta e controllare passo passo se la sequenza parziale ha possibilità di successo

> Ad ogni stadio i: se $x_1, ..., x_i$ ha possibilità di successo, si sceglie un elmento x_{i+1} tra le diverse alternative; se, con tale scelta, si arriva a una soluzione *sol*, quella è la soluzione.
>
> Altrimenti si sceglie un diverso x_{i+1}.
>
> Se non si trova una soluzione dopo aver esaminato tutte le possibilià: fallimento

Se si verifica che $x_1, ..., x_i$ non ha possibilità di successo, non si generano tutte le sequenze $x_1, ..., x_i, ...$

Riduzione dello spazio di ricerca utilizzando criteri di eliminazione

Somma di sottoinsiemi

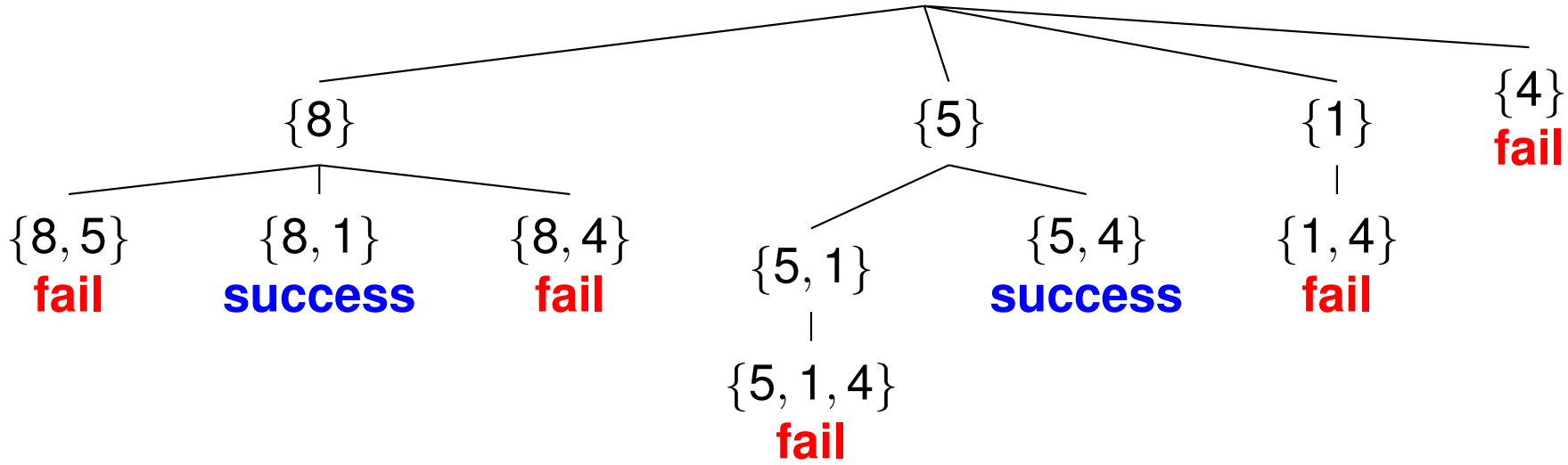
Problema: Dato un insieme S di numeri interi positivi e un intero N, determinare un sottoinsieme Y di S tale che la somma degli elementi di Y sia uguale a N.

Esempio: $S = \{8, 5, 1, 4\}$, $N = 9$.

Aggiungiamo una componente alla volta.

Spazio di ricerca delle soluzioni: albero etichettato da sottoinsiemi di S
Foglie: situazioni non ulteriormente espandibili; alcune di esse possono rappresentare una soluzione (successo) altre no (fallimento).

L'esplorazione dello spazio di ricerca avviene in profondità



Somma di sottoinsiemi: algoritmo

Ad ogni stadio della ricerca:

- una soluzione parziale *Solution*
- un insieme *Altri* di elementi tra cui scegliere

Inizialmente: $Solution = \emptyset$, $Altri = S$

- Se la somma degli elementi di *Solution* è maggiore di *N*: **fallimento**.
- Se la somma degli elementi di *Solution* è uguale a *N*: **successo** con *Solution*.

Altrimenti (somma degli elementi di *Solution* minore di *N*):

- Se *Altri* è vuoto, allora **fallimento**.
- se *Altri* non è vuoto, scegliere un elemento *x* di *Altri*.
  Alternativa:
  - mettere *x* nella soluzione: cercare una soluzione con $\{x\} \cup Solution$ e $Altri - \{x\}$
  - non mettere *x* nella soluzione: cercare una soluzione con *Solution* e $Altri - \{x\}$

Vai al codice

Alcune funzioni del modulo List

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

length : 'a list -> int

nth : 'a list -> int -> 'a

assoc : 'a -> ('a * 'b) list -> 'b

rev : 'a list -> 'a list

flatten : 'a list list -> 'a list

mem : 'a -> 'a list -> bool

split : ('a * 'b) list -> 'a list * 'b list

combine : 'a list -> 'b list -> ('a * 'b) list

sort : ('a -> 'a -> int) -> 'a list -> 'a list

merge : ('a -> 'a -> int) -> 'a list -> 'a list -> 'a list

Attraversamento di un labirinto

non ho vicini se sto nella quarta colonna perchè solo i vicini possono stare nella colonna successiva

	0	1	2	3	4
0			mostro		
1		mostro		mostro	
2				mostro	
3	mostro				
4			mostro		

Rappresentazione:

- dimensione della matrice
- lista delle caselle contenenti un mostro

```
let dim = 5
let mostri = [(0,2);(1,1);(1,3);(2,3);(3,0);(4,2)]
```

Vai al codice

Mutua ricorsione

Funzioni definite in **mutua ricorsione**: si possono richiamare una con l'altra.

Esempio banale sui naturali:

- n è pari se $n = 0$ oppure $n - 1$ è dispari
- n è dispari se $n = 1$ oppure $n - 1$ è pari

Ricerca di cammino nel labirinto:

- **cerca_da_casella**: si applica a una casella **c** e può richiamare la ricerca da una delle caselle della **lista** dei vicini di **c**.

  $$\textbf{cerca\_da\_casella: (int} \times \textbf{int)} \rightarrow \textbf{(int} \times \textbf{int) list}$$

  Non si può applicare a una lista di caselle
- **cerca_da_lista**: si applica a una lista di caselle **[c1;...;ck]** e applica **cerca_da_casella** alle singole caselle **c1,...,ck** fino a che non si trova un cammino (o la lista finisce).

  $$\textbf{cerca\_da\_lista: (int} \times \textbf{int) list} \rightarrow \textbf{(int} \times \textbf{int) list}$$