Capitolo 5

Il sistema dei moduli

5.1 Strutture e segnature

5.1.1 Programmazione modulare

La maggior parte dei linguaggi di alto livello fornisce costrutti per la strutturazione dei programmi in unità separate (classi, moduli, packages, strutture). La presenza di un sistema di moduli è infatti fondamentale per la programmazione in grande, consentendo di suddividere il programma in parti, che possono anche essere compilate e verificate separatamente, di nascondere informazioni non necessarie alla comprensione di una descrizione astratta della soluzione del problema, e di raggruppare i dati in modo che siano invisibili i dettagli dell'implementazione.

Un modulo è un'unità di programma nel quale sono "impacchettate" definizioni collegate, come la definizione di un tipo di dati e le operazioni su quel tipo. Un modulo è separato dagli altri moduli e dal resto del programma, in modo tale che i "nomi" definiti in esso non possano essere confusi con eventuali omonimi definiti in altre parti del programma. È una scatola nera che interagisce con il resto del programma tramite un'*interfaccia*, nella quale è specificato tutto ciò che del modulo è visibile (quindi utilizzabile) dall'esterno. L'interfaccia costituisce dunque la "vista pubblica" del modulo, mentre il contenuto del modulo è la sua "vista privata".

Il compito più difficile nella progettazione di un programma di grandi dimensioni è di fatto quello di stabilire quali siano i moduli giusti e le loro interfacce.

Il sistema di moduli di un linguaggio di programmazione fornisce strumenti che consentono di attuare una metodologia di programmazione basata sull'*astrazione*: la separazione, cioè, della specifica (*cosa fa* o come si usa una determinata parte di programma) dalla sua implementazione (*come* viene realizzata). Due sono i principali tipi di astrazione.

Astrazione funzionale: la specifica di una funzione è separata dalla sua realizzazione. Di conseguenza quelli che possiamo chiamare gli "utenti" di una funzione (le parti di programma che la utilizzano) sanno cosa la funzione calcola, ma il modo specifico in cui essa svolge le sue funzioni è del tutto invisibile.

Astrazione sui dati: l'uso di un tipo di dati è separato dalla sua implementazione. L'utente sa quali sono le operazioni ammissibili per quel tipo di dato, e qual è la loro funzione, ma non sa e non può fare ipotesi su quale sia la struttura concreta che implementa il tipo di dati e quale sia il modo in cui sono implementate le operazioni.

5.1.2 Tipi astratti di dati

Un tipo di dati è un insieme di oggetti e di operazioni su di esse. È di fatto la stessa cosa di una struttura algebrica (un insieme con operazioni). Dal punto di vista informatico, un tipo di dati è una struttura che può essere definita in un linguaggio di programmazione. Quando in OCaml introduciamo un nuovo tipo di dati attraverso una dichiarazione `type`, specifichiamo qual è l'insieme dei valori del tipo, cioè tutti i valori che possono essere costruiti mediante applicazione dei costruttori (specificati nella dichiarazione `type`) a valori del tipo corretto. In questo caso si parla di tipi **concreti** di dati: il tipo è definito specificando qual è la struttura che rappresenta i suoi valori. In base a tale struttura (l'insieme dei valori del tipo), è possibile definire diverse operazioni sul tipo. Definendo la struttura, infatti, abbiamo nello stesso tempo definito quali sono le operazioni ammissibili sul tipo: le operazioni di costruzione di oggetti del tipo effettuabili mediante i costruttori, le operazioni di selezione dei componenti che si possono effettuare mediante il pattern matching, eventualmente operazioni di confronto se si tratta di un tipo con eguaglianza, ecc. Parliamo dunque di un tipo concreto di dati non soltanto perché si tratta di un tipo di dati "implementato" in un linguaggio di programmazione, ma soprattutto perché il tipo è specificato definendo in modo concreto l'insieme degli oggetti del tipo; le operazioni ammissibili sono semplicemente una conseguenza del modo in cui sono definiti i valori del tipo.

Nella fase di progetto di un programma è però generalmente conveniente *fare astrazione* dal tipo concreto di dati che si può o si vuole utilizzare per rappresentare dati da manipolare. Infatti a volte un insieme di oggetti, con il quale vogliamo "calcolare", può avere diverse rappresentazioni possibili. Ad esempio, i numeri complessi si possono rappresentare mediante parte reale e immaginaria, o anche come modulo e argomento. Un grafo può essere rappresentato in modi diversi, come lista di archi o come una funzione "successori", e in altri modi ancora. Ciascuna rappresentazione può presentare vantaggi e svantaggi. Quale scegliere? Un'idea saggia è quella di rimandare la scelta della rappresentazione e scrivere programmi che non fanno alcuna assunzione su di essa. Nel caso dei numeri complessi, ad esempio, si assume solo di avere un tipo `complex`, con valori che includono `zero`, `uno` e `i`, sul quale sono definite le operazioni `sum: complex * complex -> complex`, `times: complex * complex -> complex`, ecc. Quel che interessa e che si deve specificare è qual è l'insieme di base delle operazioni che possono essere effettuate sui dati di quel tipo, specificando, per ciascuna di esse, il tipo (a cosa si applica, quale valore riporta), la funzionalità (una specifica "dichiarativa": *che cosa* si calcola), le proprietà dell'operazione in relazione ad altre. In questo modo il tipo `complex` viene trattato in modo *astratto*: non conosciamo come è rappresentato, ma possiamo tuttavia utilizzarlo attraverso le costanti e le operazioni definite.

Secondo questo approccio, quel che viene specificato è un **tipo astratto di dati** o ADT (*Abstract Data Type*), che corrisponde in realtà a una intera classe di tipi concreti, tutti quelli che, in qualche senso, rispettano le specifiche dichiarate. La scelta del tipo concreto di dati che verrà utilizzato come rappresentante del tipo astratto è posticipata e considerata, dal punto di vista dell'utente, irrilevante. Inoltre, una tale metodologia di astrazione sui dati presuppone che i dati vengano utilizzati soltanto nei modi corretti, quelli previsti nella specifica, senza manipolazioni arbitrarie dipendenti dalla loro realizzazione concreta.

Consideriamo ad esempio il tipo "lista" (o meglio, lista di elementi di un dato tipo α). Da un punto di vista astratto, le liste sono caratterizzate dall'insieme delle operazioni costituite dai due costruttori (lista vuota e operazione di inserimento in testa), i due selettori (testa e coda della lista) e dal predicato che consente di controllare se una lista è vuota (vedi paragrafo 2.2.2). Un tale insieme di operazioni caratterizza le liste, come tipo astratto di dati.

Un altro esempio ancora è costituito dal tipo astratto "dizionario": un insieme di elementi, ciascuno dei quali è caratterizzato da una "chiave" e un "valore", dove l'insieme delle chiavi è un insieme ordinato. Le operazioni di base che si possono effettuare su un dizionario sono: ricerca di un elemento (data una chiave si ottiene il valore corrispondente), inserimento e cancellazione di un elemento.

Un tipo di dati può essere rappresentato da diversi altri tipi di dati e da diverse strutture informatiche. Per esempio, il tipo astratto dizionario può essere implementato mediante liste, ma anche mediante strutture ad albero (gli "alberi binari di ricerca"), garantendo un miglior tempo medio di esecuzione delle operazioni fondamentali. Ciascuna rappresentazione astratta può venire a sua volta realizzata concretamente da diverse strutture dati. Per un esempio, possiamo usare la realizzazione `'a tree` degli alberi binari come struttura dati per la realizzazione di un albero binario di ricerca. Questa struttura dati, insieme all'implementazione delle opportune operazioni, costituisce una realizzazione dell'ADT dizionario.

Una ragione importante per usare gli ADT in un programma è che tramite essi i dati che costituiscono il tipo di dato astratto sono accessibili solo attraverso le operazioni dell'ADT stesso. Tale restrizione costituisce una forma di programmazione cauta, che mette al sicuro da alterazioni accidentali dei dati provocate da qualche procedura che manipoli i dati in modo non previsto. Una seconda ragione importante per l'uso di ADT, è che gli ADT consentono di ri-progettare le strutture dati e le procedure che realizzano le operazioni su di esse (per esempio, per migliorarne l'efficienza), senza la preoccupazione di introdurre, così facendo, qualche errore nelle parti restanti del programma. Non possono infatti insorgere nuovi errori se l'unica interazione con l'ADT avviene attraverso le procedure corrispondenti alle sue operazioni (e correttamente modificate).

Supponiamo ora di voler definire un tipo di dati OCaml per *rappresentare* i numeri razionali; decidiamo di rappresentarli mediante coppie (numeratore,denominatore) e dichiariamo dunque:

```
type rat = Rat of int * int
```

Tra i valori del tipo di dati `rat` ci sono ad esempio `Rat(1,2), Rat(2,4), Rat(3,6),`

...: sono tutti valori concreti distinti, eppure nell'algebra dei razionali che si vuole rappresentare c'è un unico valore in corrispondenza di essi. In questo caso la corrispondenza tra oggetti del tipo concreto che abbiamo definito e oggetti "astratti" che si vogliono rappresentare non è uno-uno. Eppure, il tipo `rat` può servire benissimo per rappresentare i razionali. Perché?

Cosa significa "rappresentare"?

Consideriamo una struttura algebrica (un tipo di dati) $\mathcal{A} = \langle A, f_1, ..., f_n \rangle$, cioè un insieme di oggetti A, sul quale sono definite le operazioni $f_1, ..., f_n$. Sia inoltre $\mathcal{B} = \langle B, g_1, ..., g_n \rangle$ una struttura algebrica simile ad $\mathcal{A}$, cioè B è un insieme di oggetti, sul quale sono definite le operazioni $g_1, ..., g_n$, tali che il numero di argomenti di g_i è uguale al numero di argomenti di f_i, per ogni $i = 1, ..., n$.
Allora per definire una rappresentazione di $\mathcal{A}$ in $\mathcal{B}$ occorre determinare una relazione tra oggetti di A e oggetti di B ("a è rappresentato da b" o "b è un rappresentante di a") tale che:

1. ogni oggetto di A ha almeno un rappresentante in B (non si richiede che il rappresentante sia unico);
2. ogni oggetto di B rappresenta uno ed un unico elemento di A; di conseguenza, elementi distinti di A hanno rappresentanti distinti in B;
3. la rappresentazione "conserva la struttura": il risultato dell'operazione f_i eseguita in A su $a_1, ..., a_k$ è rappresentato dal risultato che si ottiene eseguendo l'operazione corrispondente g_i in B su rappresentanti degli oggetti $a_1, ..., a_k$.

Ad esempio, sia $\mathcal{A}$ l'insieme di tutti gli insiemi finiti di numeri naturali con l'operazione di unione, e $\mathcal{B}$ l'insieme delle liste di naturali con l'operazione di concatenazione. Allora $\mathcal{A}$ può essere rappresentato da $\mathcal{B}$ stabilendo che un insieme è rappresentato da qualsiasi lista contenente gli stessi elementi. Infatti: ogni insieme ha almeno un rappresentante, insiemi distinti hanno certamente rappresentanti distinti e gli elementi contenuti in $L_1@L_2$ sono esattamente quelli contenuti nell'unione degli elementi di L_1 con gli elementi di L_2. Si noti che in questo caso un insieme è rappresentato da più di una lista, dato che nelle liste è rilevante l'ordine e la ripetizione di elementi.
Formalmente, una rappresentazione di $\mathcal{A}$ in $\mathcal{B}$ è un'applicazione φ da B a A tale che:

- per ogni $a \in A$ esiste almeno un elemento $b \in B$ tale che $\varphi(b) = a$.
- per ogni operazione f_i su A a k argomenti, $\varphi(g_i(b_1, ..., b_k)) = f_i(\varphi(b_1), ..., \varphi(b_k))$.

In altri termini, φ è un *omomorfismo suriettivo*. Si noti che il fatto che ogni elemento di B rappresenta uno ed un unico elemento di A è una conseguenza del fatto che φ è una funzione (totale) da B in A.

La prima cosa da chiarire dunque è *che cosa si vuole rappresentare*: occorre specificare cos'è il tipo dei razionali da un punto di vista astratto, cioè come *tipo astratto di dati*. Per far ciò occorre specificare non tanto qual è l'insieme degli oggetti del tipo, quanto l'insieme delle operazioni di base che possono essere eseguite su questi oggetti, qual è il loro tipo e come si comportano. Infatti, se ponessimo l'accento sul-

l'insieme dei razionali, non sarebbe possibile riconoscere che il tipo `rat` è una buona rappresentazione di essi.

Per specificare un tipo astratto di dati, non viene messa in evidenza tanto la struttura dei valori, quanto il loro *comportamento logico*, ossia il comportamento dei valori rispetto alle operazioni primitive. Nel caso dei razionali, ad esempio, possiamo considerare le operazioni di base di somma, prodotto, ecc. l'uguaglianza, un'operazione di costruzione di un razionale a partire da due numeri naturali, ecc. Il comportamento logico dei razionali rispetto a queste operazioni si può specificare mediante un insieme di "assiomi" (ad esempio un insieme di equazioni) che descrivono in modo sufficientemente completo le operazioni e le relazioni (*specifica logica*). In questo caso si tratta di una *specifica formale* del tipo di dati. Spesso, tuttavia, la specifica è data senza ricorrere a un linguaggio formale.

Per tornare all'esempio dei razionali, allora, la dichiarazione di tipo non è sufficiente per avere un'implementazione dei razionali: occorrerà definire tutte le operazioni di base sui razionali, in modo tale che esse si comportino come voluto. In particolare, ad esempio, si dovrà definire un predicato di eguaglianza sui razionali che implementi correttamente l'eguaglianza sul tipo astratto, cioè ad esempio in modo che `Rat(1,2)` risulti "uguale" a `Rat(2,4)`. Una volta che si ha questa implementazione completa, si può "dimenticare" che l'insieme di supporto dell'algebra dei razionali è implementato proprio mediante il tipo `rat`, cioè che si tratta di coppie di interi, che il costruttore si chiama `Rat`, ecc. Per l'"utente" del tipo dei razionali tutto questo potrebbe essere irrilevante, mentre la cosa fondamentale è che si dispone di tale e tale operazione, di tipo così e così, che si comporta nel modo voluto.

L'astrazione sui dati, come approccio metodologico alla programmazione, consiste proprio in questo: quel che viene specificato è il tipo astratto di dati, cioè l'insieme delle operazioni ammissibili con il loro tipo e la loro descrizione, mentre nulla viene specificato e nulla l'utente può assumere sulla struttura concreta che viene utilizzata per implementare il tipo di dati. Se l'implementazione del tipo di dati è racchiusa in un modulo, l'*interfaccia* del modulo specifica quali sono le operazioni "visibili" dall'esterno (quindi le operazioni sul tipo di dati). L'interfaccia costituisce una *barriera dell'astrazione* che l'utente non può superare, cioè oltre la quale non può guardare (vedi figura 5.1).

5.1.3 Il sistema dei moduli di OCaml

Il sistema di moduli di OCaml consente di creare e manipolare *ambienti*. Un ambiente è il risultato della valutazione di una collezione di dichiarazioni. È possibile creare e modificare ambienti, conservarli separatamente, estenderli, nascondere parte del contenuto di un ambiente, e addirittura scrivere "funzioni" (*funtori*) da ambienti a ambienti.

Un modulo in OCaml viene chiamato *struttura*. La sua interfaccia è chiamata *segnatura*. Si può evidenziare un'analogia tra il livello dei moduli e il livello del programma in OCaml: una struttura (ambiente) corrisponde a un valore, la sua segnatura al tipo di un valore, un funtore si applica ad ambiente e restituisce ambienti, così come una funzione si applica a valori e restituisce valori.

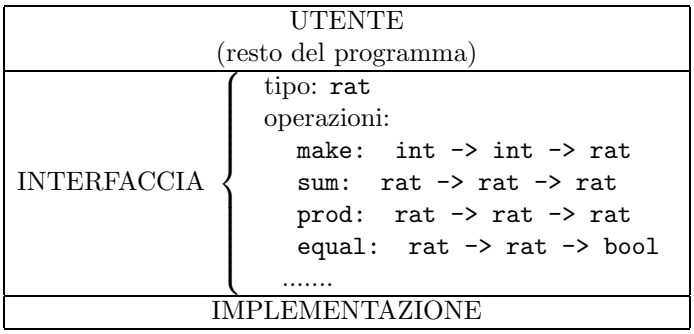

Figura 5.1: L'interfaccia del modulo come barriera dell'astrazione

Per illustrare i concetti di base e la sintassi di strutture e segnature in OCaml, iniziamo con esempi semplicissimi, ancorché poco significativi. Una struttura è un ambiente creato da una collezione di dichiarazioni associate ad un identificatore (nome della struttura). Essa raggruppa elementi che servono ad uno stesso scopo, quali tipi, eccezioni, valori e funzioni, altre strutture. Una struttura viene dichiarata con una dichiarazione `module`. Ad esempio, la dichiarazione seguente crea una struttura di nome `S` contenente un tipo `t`, un valore `x` e una funzione `f`:

```
module S =
struct
  type t = int
  let x = 3
  let rec f x = if x=0 then 1 else x*f(x-1)
end;;
```

Il risultato della valutazione di una dichiarazione di struttura è un ambiente identificato dal nome della struttura. Una struttura è un ambiente indipendente: gli oggetti dichiarati al suo interno non sono visibili direttamente a top-level. Ad essi si accede specificando il nome della struttura con la sintassi: `<nome-struttura>.<nome-elemento>`:

```
# x;;
Characters 0-1:
Unbound value x
# S.x;;
- : int = 3
# S.f S.x;;
- : int = 6
```

Possiamo dichiarare una diversa struttura, contenente anch'essa un tipo `t`, un valore `x` e una funzione `f`. Tali oggetti non verranno confusi con quelli di `S`:

```
module T =
  struct
    type t = bool
    let x = 5
    let rec f(x,y) = if y=0 then 1 else x*f(x,y-1)
  end;;

# T.x;;
- : int = 5
# S.f;;
- : int -> int = <fun>
# T.f;;
- : int * int -> int = <fun>
# T.f(S.x,S.x);;
- : int = 27
```

È possibile "aprire" una struttura: la valutazione di un'espressione della forma `open <nome-struttura>` ha come effetto l'aggiunta all'ambiente corrente dei nomi definiti nella struttura. "Aprire" una struttura è comunque un'operazione delicata, in quanto si rischia di riprodurre quei conflitti di nomi che l'organizzazione di un programma in moduli vuole evitare.

Una segnatura è una collezione di informazioni che descrivono gli elementi di una struttura, il suo "linguaggio", o la sua "interfaccia". Una segnatura è infatti costituita da una collezione di identificatori (gli identificatori dei valori e delle funzioni presenti all'interno della struttura che sono visibili all'esterno), con il tipo relativo. Quando viene dichiarata una struttura, la risposta di OCaml evidenzia l'interfaccia della struttura, così come, quando viene dichiarato un valore o una funzione, riporta il suo tipo:

```
# module S =
struct
  type t = int
  let x = 3
  let rec f x = if x=0 then 1 else x*f(x-1)
end;;
module S : sig type t = int val x : int val f : int -> int end
```

Un'espressione per una segnatura è racchiusa tra le parole chiave `sig ... end`.

È possibile associare un nome a una segnatura mediante una dichiarazione `module type`:

```
module type SIG =
  sig
    val x: int
    val b: bool
  end;;
```

Con tale dichiarazione è stato dato il nome `SIG` alla segnatura contenente un valore `x` intero e un valore `b` booleano.

Il *corpo* della segnatura (incluso tra le parole chiave `sig` e `end`) è chiamato *specifica*: esso descrive un insieme di identificatori, ma non i relativi valori. Si noti che le segnature possono essere solamente definite a top-level (non, ad esempio, all'interno di strutture).

Una segnatura definisce, in un certo senso, una classe di strutture, cioè tutte le possibili *istanze* della segnatura. Una struttura è un'istanza di una segnatura se la struttura soddisfa la specifica definita nella segnatura, cioè se essa contiene definizioni per tutti gli elementi descritti nella segnatura. Ad esempio, la struttura `A` sotto definita è un'istanza di `SIG`:

```
module A =
  struct
    let y = true
    let x = 2+1
    let b = x=7 || y
  end;;
```

La sua segnaura:

```
           sig val y :  bool val x :  int val b :  bool end
```

è infatti più generale di `SIG`.

Una segnatura si può utilizzare per nascondere alcune componenti di una struttura o per esportare delle componenti con un tipo ristretto. Ad esempio, se dichiariamo:

```
# module B : SIG =
  struct
    let y = true
    let x = 2+1
    let b = x=7 || y
  end;;
module B : SIG
```

il valore `y` non è visibile all'esterno:

```
# B.y;;
Unbound value B.y
```

Se la struttura non definisce tutti gli elementi specificati nella segnatura, si ha un errore

```
# module C : SIG =
  struct
    let b = true
  end;;

Signature mismatch:
Modules do not match: sig val b : bool end is not included in SIG
The field ‘x’ is required but not provided
```

Un uso importante delle segnature è quello che consente di realizzare l'astrazione sui dati in OCaml. Ad esempio, per realizzare il tipo astratto di dati "insieme finito di interi", possiamo definire una segnatura che nasconde il tipo concreto utilizzato per la rappresentazione. La segnatura `SET` specifica il tipo `set` senza dichiarare quale struttura concreta rappresenta gli insiemi:

```
module type SET =
sig
  type set
  exception Error
  val empty : set
  val add : int -> set -> set
  val is_empty : set -> bool
  val union : set -> set -> set
  val intersection : set -> set -> set
  val member : int -> set -> bool
  val choose : set -> int     (* funzione di scelta *)
end;;
```

La struttura `Set` rappresenta gli insiemi mediante liste:

```
module Set : SET =
struct
  type set = int list
  exception Error
  let empty = []
  let add x s = if List.mem x s then s else x::s
  let is_empty s = s=[]
  let rec union set = function
     [] -> set
   | y::ys -> add y (union set ys);;
  let rec intersection set = function
     [] -> []
   | y::ys -> if List.mem y set then y::intersection set ys
              else intersection set ys;;
  let member = List.mem
  let choose = function
     [] -> raise Error
   | x::_ -> x
end;;
```

La rappresentazione del tipo è nascosta, come evidenziato dalle risposte di OCaml:

```
# let s = Set.add 1 (Set.add 2 Set.empty);;
val s : Set.set = <abstr>
```

Di conseguenza, non possiamo applicare a insiemi operazioni su liste:

```
# 3::s;;
This expression has type Set.set but is here used with type int list
```

E, viceversa, non possiamo applicare a liste operazioni su insiemi:

```
# Set.add 1 [];;
This expression has type 'a list but is here used with type Set.set
```

Se un modulo viene definito senza specificare la sua segnatura, allora tutto ciò che in esso è definito sarà visibile dall'esterno; la sua segnatura cioè è la più ricca possibile (come è evidenziato dalla risposta di OCaml nell'esempio che segue). Quindi, se definiamo una struttura identica a `Set`, ma senza restrizione di segnatura, è possibile confondere insiemi e liste:

```
# module OpenSet =
struct
  type set = int list
  exception Error
  let empty = []
  ...
end;;
module OpenSet :
  sig
    type set = int list
    exception Error
    val empty : 'a list
    val add : 'a -> 'a list -> 'a list
    val is_empty : 'a list -> bool
    val union : 'a list -> 'a list -> 'a list
    val intersection : 'a list -> 'a list -> 'a list
    val member : 'a -> 'a list -> bool
    val choose : 'a list -> 'a
  end

# OpenSet.add 1 [];;
- : int list = [1]
# 3::OpenSet.empty;;
- : int list = [3]
```

Recpitolando, la sintassi di una dichiarazione di struttura è la seguente:

```
module <nome-struttura> : <nome-segnatura> =
  struct
    ... <DICHIARAZIONE>
    ... <DICHIARAZIONE>
  end
```

Se una struttura è dichiarata senza segnatura OCaml deduce la più generale.

La sintassi generale di una dichiarazione di segnatura è la seguente:

```
module type <nome-segnatura> =
  sig
    (* tipi *)
        type <nome-del-tipo>
           (* il tipo concreto e' nascosto *)
        type <nome-del-tipo> = ....
           (* in questo caso i costruttori sono visibili *)
    (* eccezioni *)
        exception <nome-dell'eccezione>
    (* valori e funzioni *)
        val <nome> : <tipo>
    (* strutture *)
        module <nome> : <segnatura>
  end
```

5.1.4 I numeri razionali

Nella progettazione di programmi di grandi dimensioni si segue un approccio top-down: la progettazione consiste fondamentalmente nella definizione delle segnature, cioè delle interfacce dei moduli. Nel caso si voglia implementare il tipo di dati razionali, si procede specificando che, indipendentemente dall'implementazione concreta, la rappresentazione deve prevedere le quattro operazioni e un test di uguaglianza. Inoltre si deve definire un costruttore per i razionali, `make`, che, dati due interi *num* e *den*, costruisca il corrispondente razionale (ridotto ai minimi termini) se *den* è diverso da zero, e dia un errore altrimenti; e si vogliono definire anche i due selettori per restituire numeratore e denominatore di un razionale.

La segnatura del modulo per i razionali è dunque la seguente:

```
module type RAT =
sig
  type rat
  exception Error
  val make : int -> int -> rat
  val num : rat -> int
  val den : rat -> int
  val sum : rat * rat -> rat
  val diff: rat * rat -> rat
  val prod :  rat * rat -> rat
  val div :  rat * rat -> rat
  val equal :  rat * rat -> bool
end;;
```

Una struttura sarà dunque un esempio della segnatura RAT se in essa sono dichiarati almeno un tipo chiamato `rat`, un'eccezione di nome `Error`, una funzione `make` di tipo `int -> int -> rat`, ecc.

Le segnature, come abbiamo visto, sono le interfacce delle strutture. Una segnatura specifica quali componenti di una struttura sono accessibili dall'esterno e con quale tipo. Quindi specificando la segnatura `RAT`, abbiamo deciso il nome del tipo di dati (ma nulla sulla struttura concreta che lo implementa), il nome delle operazioni e il loro tipo; in particolare, ad esempio, che `make` è in forma currificata e le altre operazioni invece si applicano a coppie di interi.

Nell'implementazione della struttura per i razionali sarà probabilmente necessario definire una funzione ausiliaria per il calcolo del massimo comun divisore tra due interi quindi per la riduzione di una frazione ai minimi termini. Ma tale funzione non sarà visibile all'esterno perché non è specificata nella segnatura: è una funzione di servizio, non necessaria nella descrizione del tipo di dati.

Una struttura per l'implementazione dei razionali può essere dichiarata come segue:

```
module Rat : RAT =
struct
  type rat = Rat of int*int
  exception Error
  let rec gcd m n = if m = 0 then n
                    else gcd (n mod m) m
  let reduce (Rat(n,d)) = let g = gcd n d
                          in try Rat(n / g, d / g)
                          with  Division_by_zero -> raise Error
  let make n m = if m<>0 then reduce(Rat(n,m))
                 else raise Error
  let num (Rat(n,_)) = n
  let den (Rat(_,m)) = m
  let sum (Rat(n1,d1), Rat(n2,d2)) =
       reduce (Rat(n1*d2 + n2*d1,d1*d2))
  let diff (Rat(n1,d1), Rat(n2,d2)) =
      reduce (Rat(n1*d2 - n2*d1,d1*d2))
  let prod (Rat(n1,d1), Rat(n2,d2)) =
    reduce (Rat(n1*n2,d1*d2))
  let div (Rat(n1,d1), Rat(n2,d2)) =
    reduce (Rat(n1*d2,n2*d1))
  let equal(r1,r2) = r1=r2
end;;
```

Si noti che nell'implementazione vengono prese decisioni non specificate a livelli di astrazione superiore; ad esempio, il fatto che la riduzione ai minimi termini viene effettuata immediatamente, sia alla creazione di un razionale che con l'esecuzione di un'operazione, anziché, ad esempio, soltanto quando si deve effettuare il test di uguaglianza (o mai).

Ora che la struttura è stata definita, è possibile accedere alle sue componenti mediante la sintassi `<nome-della-struttura>.<nome-della-componente>`:

```
# let r1 = Rat.make 3 8;;
```

```
val r1 : Rat.rat = <abstr>
# let r2 = Rat.make 5 6;;
val r2 : Rat.rat = <abstr>
# let r = Rat.sum(r1,r2);;
val r : Rat.rat = <abstr>
# Rat.num r;;
- : int = 29
# Rat.den r;;
- : int = 24
```

La risposta di OCaml quando vengono dichiarati valori di tipo `Rat.rat` evidenzia il fatto che il valore è un'astrazione, pertanto la sua struttura concreta non è visibile dall'esterno e il valore non è stampabile.

Dall'esterno è visibile soltanto ciò che è specificato dalla segnatura, cioè l'interfaccia nasconde tutto ciò che non è dichiarato in essa, incluso il costruttore `Rat`:

```
# Rat.gcd 20 30;;
Unbound value Rat.gcd
# Rat.Rat(3,4);;
Unbound constructor Rat.Rat
```

Se definiamo un modulo identico a `Rat`, ma senza restrizioni di segnatura, allora tutto sarà accessibile dall'esterno:

```
# module OpenRat =
  struct
    type rat = Rat of int*int
    ...
  end;;

module OpenRat :
  sig
    type rat = Rat of int * int
    exception Error
    val gcd : int -> int -> int
    val reduce : rat -> rat
    val make : int -> int -> rat
    val num : rat -> int
    val den : rat -> int
    val sum : rat * rat -> rat
    val diff : rat * rat -> rat
    val prod : rat * rat -> rat
    val div : rat * rat -> rat
    val equal : 'a * 'a -> bool
  end
```

Ovviamente l'uso improprio di alcuni costruttori o operazioni può a questo punto avere conseguenze imprevedibili se non si conoscono i dettagli dell'implementazione:

```
# OpenRat.equal (OpenRat.Rat(3,6),OpenRat.make 3 6);;
- : bool = false
```

5.1.5 Alberi binari

Nel paragrafo 3.2 abbiamo introdotto gli alberi binari come tipo concreto. In realtà un albero binario potrebbe essere rappresentato in modi diversi; ad esempio – nel caso più semplice – utilizzando nomi diversi per i costruttori, oppure facendo ricorso ad altre strutture di dati, come gli array o strutture realizzate mediante record e puntatori (vedi capitolo 6). Per definire il tipo albero binario indipendentemente dalla rappresentazione concreta che se ne può dare, è necessario specificare quali sono le operazioni di base sugli alberi binari. Qualunque implementazione del tipo albero binario deve consentire l'implementazione di tali operazioni. Le operazioni, includendo le costanti, che definiscono il tipo albero binario con nodi etichettati da elementi di un tipo generico `'a` sono le seguenti:

empty: `'a tree`
: Rappresenta l'albero vuoto (costruttore)

mktree: `'a * 'a tree * 'a tree -> 'a tree`
: È l'operazione che, dato un oggetto x e alberi binari t_1 e t_2, restituisce l'albero binario che ha x come radice, e t_1 e t_2 come sottoalberi sinistro e destro (costruttore)

root: `'a tree -> 'a`
: È l'operazione che, dato un albero binario non vuoto, ne restituisce la radice. Riporta un errore quando è applicata all'albero vuoto (selettore)

left: `'a tree -> 'a tree`
: È l'operazione che, dato un albero binario non vuoto, ne restituisce il sottoalbero sinistro. Riporta un errore quando è applicata all'albero vuoto (selettore)

right: `'a tree -> 'a tree`
: È l'operazione che, dato un albero binario non vuoto, ne restituisce il sottoalbero destro. Riporta un errore quando è applicata all'albero vuoto (selettore)

is_empty: `'a tree -> bool`
: È il predicato vero per l'albero vuoto, falso in tutti gli altri casi.

Una segnatura adeguata per gli alberi binari è dunque la seguente

```
module type BINTREE =
sig
  type 'a t
  exception Empty
  val empty : 'a t
  val make : 'a * 'a t * 'a t -> 'a t
  val root : 'a t -> 'a
```

```
  val left : 'a t -> 'a t
  val right : 'a t -> 'a t
  val is_empty : 'a t -> bool
end;;
```

Si noti che, dato che i nomi delle componenti di una struttura con segnatura `BINTREE` saranno sempre preceduti dal nome della struttura, i nomi di tali componenti si possono "accorciare" senza perdere la loro significatività. Ad esempio, come nome per il tipo abbiamo utilizzato il simbolo `t`, in quanto, se `Bintree` è ad esempio una struttura con segnatura `BINTREE`, dall'esterno ci si riferirà al nome del tipo mediante `Bintree.t`.

La struttura seguente, utilizzando la segnatura `BINTREE`, implementa gli alberi binari nascondendo la struttura di dati concreta che si utilizza per la loro rappresentazione.

```
module Bintree : BINTREE =
struct
  type 'a t = E | Tr of 'a * 'a t * 'a t
  exception Empty
  let empty = E
  let make(x,t1,t2) = Tr(x,t1,t2)
  let root = function
      Tr(x,_,_) -> x
    | _ -> raise Empty
  let left = function
      Tr(_,t,_) -> t
    | _ -> raise Empty
  let right = function
      Tr(_,_,t) -> t
    | _ -> raise Empty
  let is_empty t = t=E
end;;
```

Se dall'esterno vogliamo definire ad esempio una funzione che raccoglie in una lista i nodi di un albero, non possiamo utilizzare il pattern matching (i costruttori non sono visibili), ma si deve ricorrere all'uso dei selettori e del predicato `is_empty`:

```
(* preorder : 'a Bintree.t -> 'a list *)
let rec preorder t =
  if Bintree.is_empty t then []
  else ((Bintree.root t)::(preorder(Bintree.left t)))
       @ (preorder(Bintree.right t));;
```

Se invece vogliamo implementare gli alberi binari in modo tale che la struttura concreta utilizzata per rappresentarli sia visibile dall'esterno, specificheremo tale struttura nell'interfaccia. Ad esempio:

```
module type BINTREE' =
sig
  type 'a t = E
            | Tr of 'a * 'a t * 'a t
  exception Empty
  ...
end;;
```

In tal caso, se `S` è una struttura con segnatura ristretta da `BINTREE'`,

```
module S: BINTREE' = struct
  type 'a t = E | Tr of 'a * 'a t * 'a t
  exception Empty
  let empty = E
  ...
end;;
```

dall'esterno del modulo sarà possibile accedere ai costruttori del tipo `'a S.t` e definire, ad esempio, funzioni sugli alberi mediante pattern matching:

```
let rec preord = function
    S.E -> []
  | S.Tr(x,t1,t2) -> (x::(preord t1))@(preord t2);;
```

Si noti che la scelta tra l'approccio astratto e concreto è una scelta di progettazione, e nessuna delle alternative può essere considerata più "giusta" dell'altra.

5.1.6 Associazioni

Un'associazione è una collezione di legami tra coppie di valori, possibilmente di tipo diverso. Un'associazione è simile a un ambiente: un insieme finito di elementi di un tipo `'a` è posto in corrispondenza con valori di un tipo `'b`. Il tipo delle associazioni è dunque parametrico in `'a` e `'b`: `('a,'b) assoc`. Gli oggetti di tipo `'a` vengono a volte chiamati chiavi, quelli di tipo `'b` valori.

Da un punto di vista astratto, possiamo definire le associazioni come oggetti su cui sono definite le seguenti operazioni:

- `empty: ('a,'b) assoc`, l'associazione vuota (sempre indefinita);
- `lookup: 'a -> ('a,'b) assoc -> 'b`: la funzione che, applicata a un oggetto `x` e una associazione `ass` riporta il valore associato a `x` da `ass`, se `ass` è definita per `x`, un errore altrimenti (corrisponde all'operazione di ricerca del valore di una variabile in un ambiente);
- `bind: ('a*'b) list -> ('a,'b) assoc -> ('a,'b) assoc`: applicata a una lista di coppie `lst` e un'associazione `ass`, riporta l'associazione che si ottiene aggiungendo ad `ass` tutti i legami rappresentati da ciascuna delle coppie in `lst`, sovrascrivendo eventuali legami già presenti per le stesse chiavi (corrisponde all'operazione di estensione di un ambiente).

Possiamo dunque definire una segnatura per le associazioni come segue (il tipo delle associazioni viene chiamato semplicemente `t`, in quando dall'esterno vi si farà riferimento preceduto dal nome della struttura):

```
module type ASSOC =
sig
  type ('a,'b) t
  exception NotFound
  val empty: ('a,'b) t
  val lookup: 'a -> ('a,'b) t -> 'b
  val bind:  ('a*'b) list -> ('a,'b) t -> ('a,'b) t
end;;
```

Un'associazione si può rappresentare mediante una *lista associativa*, cioè una lista di coppie di tipo `('a*'b) list` (vedi paragrafo 2.3.2). L'associazione vuota è rappresentata dalla lista vuota, l'operazione `lookup` esamina ad una ad una le coppie della lista fino a che non si trova una coppia con chiave uguale a quella cercata o fino a terminare la lista; l'operazione `bind` è semplicemente un append (ma la segnatura restringe il tipo dell'operazione).

```
module Assoc : ASSOC =
struct
  type ('a,'b) t = ('a * 'b) list
  exception NotFound
  let empty = []
  let rec lookup x = function
      [] -> raise NotFound
    | (k,v)::rest ->
      if k = x then v
      else lookup x rest
  let bind = ( @ )
end;;
```

Costruiamo alcune associazioni:

```
# let inizio = Assoc.empty;;
val inizio : ('a, 'b) Assoc.t = <abstr>
# let nuovo = Assoc.bind [("a",1);("b",2)] inizio;;
val nuovo : (string, int) Assoc.t = <abstr>
# Assoc.lookup "b" nuovo;;
- : int = 2
# let altro = Assoc.bind [(1,"dotto");(3,"eolo")] inizio;;
val altro : (int, string) Assoc.t = <abstr>
# Assoc.lookup 3 altro;;
- : string = "eolo"
```

In alcuni casi può essere pesante usare sempre il nome della struttura per richiamare le funzioni al suo interno.

Le associazioni possono essere rappresentate anche in modo diverso. Dal punto di vista matematico, una associazione di tipo `('a,'b) assoc` è una funzione parziale con dominio `'a` e codominio `'b`. Possiamo dunque rappresentare le associazioni direttamente mediante funzioni. In questo caso, `empty` è la funzione indefinita ovunque, `lookup` è l'applicazione della funzione, mentre la definizione di `bind` è un pochino più complicata.

```
module Assoc1 : ASSOC =
struct
  type ('a,'b) t = 'a -> 'b
  exception NotFound
  let empty = function x -> raise NotFound
  let lookup x ass = ass x
  let rec bind lst ass = match lst with
    [] -> ass
  | (x,y)::rest ->
      function key ->
         if key=x then y else
         bind rest ass key
end;;

# let test = Assoc1.bind  [(1,"a");(2,"b")] Assoc1.empty;;
val test : (int, string) Assoc1.t = <abstr>
# Assoc1.lookup 2 test;;
- : string = "b"
```

5.1.7 Code

Un importante tipo di dato astratto basato sul modello dei dati lista è la coda, una forma ristretta di lista in cui gli elementi vengono inseriti a un estremo, il retro, e cancellati all'altro estremo, la fronte. Il termine "lista FIFO" (*first-in first-out*, letteralmente: primo entrato primo uscito) è un sinonimo di coda.

L'idea intuitiva che sta dietro alla coda è proprio quella di una coda a uno sportello: una persona entra nella coda dal retro e viene servita quando raggiunge la fronte. Le persone vengono servite nello stesso ordine in cui sono arrivate, facendo sì che, a ogni istante, venga servita la persona che ha atteso più a lungo.

Il modello astratto di una coda è lo stesso della lista (o della pila), ma le operazioni che vengono applicate sono particolari. Le due operazioni caratteristiche di una coda sono `enqueue` e `dequeue`: la prima aggiunge un elemento al retro di una coda, mentre `dequeue` cancella l'elemento che sta in fronte alla coda. Se `'a queue` è il tipo delle code con elementi di tipo `'a`, le operazioni di base sul tipo `'a queue` sono le seguenti:

`empty: 'a queue`
: la coda vuota

`dequeue: 'a queue -> 'a queue`
: funzione che, applicata ad una coda non vuota ne elimina il primo elemento

`head: 'a queue -> 'a`
: funzione che, applicata a una coda non vuota, restituisce il suo primo elemento

`enqueue: 'a queue -> 'a -> 'a list`
: funzione che, applicata a una coda `q` e un elemento `x`, riporta la coda che si ottiene inserendo `x` in `q`

`null: 'a queue -> bool`
: predicato che determina se una coda è vuota.

Quindi possiamo definire una segnatura per le code come segue:

```
module type QUEUE =
sig
  type 'a t
  exception Empty
  val empty : 'a t
  val dequeue : 'a t -> 'a t
  val head : 'a t -> 'a
  val enqueue : 'a t -> 'a -> 'a t
  val null : 'a t -> bool
end;;
```

Una coda può essere rappresentata semplicemente mediante una lista, come nella struttura `Queue1`:

```
module Queue1 : QUEUE =
struct
  type 'a t = 'a list
  exception Empty
  let empty = []
  let dequeue = function
      [] -> raise Empty
    | _::q -> q
  let head = function
      [] -> raise Empty
    | x::_ -> x
  let enqueue q x = q @ [x]
  let null q = q = []
end;;
```

L'operazione di inserimento di un elemento in una coda è però piuttosto costosa: usando la concatenazione di liste, ogni volta che si inserisce un elemento, la coda viene tutta scandita dall'inizio alla fine. Il modo più comune di implementare le code in un linguaggio funzionale è mediante una coppia di liste (vedi [10]): la prima lista contiene i primi elementi della coda, nell'ordine corretto, e la seconda contiene gli altri elementi, in ordine inverso. Cioè, la coppia `([`$x_1; \ldots; x_m$`],[`$y_1; \ldots; y_n$`])` rappresenta la coda `[`$x_1; \ldots; x_m; y_n; \ldots; y_1$`]`.

Le funzioni **enqueue** e **dequeue** sono sempre eseguite sui primi elementi delle liste `[`$y_1; \ldots; y_n$`]` e `[`$x_1; \ldots; x_m$`]`, rispettivamente. Sono dunque operazioni a costo "costante" (cioè il tempo impiegato per la loro esecuzione non dipende dalla dimensione della coda).

Quando, in seguito ad una serie di `dequeue`, la prima lista diventa vuota, allora la seconda viene rovesciata e sostituita alla prima: la coppia `([],[`$y_1; \ldots; y_n$`])` diventa cioè `([`$y_n; \ldots; y_1$`],[])`. La rappresentazione deve dunque essere mantenuta in una forma standard (o forma *normale*): o è la rappresentazione della coda vuota `([],[])` oppure ha la forma (xs, ys) con $xs \neq$ `[]`.

```
module Queue2 : QUEUE =
struct
  type 'a t = 'a list * 'a list
  exception Empty
  (* operazione di normalizzazione di una coda *)
  let norm = function
      ([],tail) -> (List.rev tail,[])
    | q -> q
  let empty = ([],[])
   (* poiche' la coda e' sempre in forma normale, se la prima
      componente e' vuota, allora tutta la coda e' vuota *)
  let dequeue = function
      ((x::h),t) -> norm (h,t)
    | _ -> raise Empty
  let head = function
      ((x::_),_) -> x
    | _ -> raise Empty
    (* se si aggiunge un elemento alla coda vuota si deve
       poi normalizzare *)
  let enqueue (h,t) x = norm (h,(x::t))
  let null q = q = ([],[])
end;;
```

5.1.8 Matrici

Se vogliamo implementare il tipo di dati matrice a due dimensioni, una possibile segnatura per esse è la seguente:

```
module type MATRIX =
sig
  type 'a t
  type coord = int * int
  exception Error
  val make: 'a -> coord -> 'a t
  val size: 'a t -> coord
  val lookup : 'a t -> coord -> 'a
```

```
  val update : 'a t -> coord -> 'a  -> 'a t
end;;
```

Qui, `'a t` è il tipo delle matrici con elementi di tipo `'a`, considerato in modo astratto; il tipo `coord` è il tipo delle coordinate ed è semplicemente un'abbreviazione per il tipo `int * int` (e questo è visibile dall'esterno); la funzione `make`, applicata a un elemento `x` e una coppia di interi `(m,n)`, riporta una matrice di dimensioni `m`×`n` i cui elementi hanno tutti valore `x`; `size` riporta la dimensione di una matrice; `lookup`, applicata a una matrice `M` e a una coppia di coordinate `(x,y)`, riporta l'elemento di `M` che si trova in posizione `(x,y)`, se tale posizione è interna alla matrice, e solleva l'eccezione `Error` altrimenti; la funzione `update`, applicata a una matrice `M`, a una coppia di coordinate `(x,y)` e a un valore `v`, solleva l'eccezione `Error` se le coordinate `(x,y)` non sono coordinate valide per `M`, altrimenti riporta una nuova matrice, che è uguale a `M` tranne che per l'elemento in posizione `(x,y)` che ha valore `v`.

Il modo più immediato di implementare le matrici in OCaml è mediante liste di liste: ciascuna lista rappresenta una riga della matrice. Nell'implementazione che segue, una matrice è in realtà rappresentata da una coppia di valori: il primo è una coppia di interi, che rappresenta la dimensione della matrice, il secondo è una lista di liste che rappresenta i valori della matrice. Inoltre, la funzione `mklist` (utilizzata da `make`) è una funzione che, applicata a un intero non negativo `n` e a un valore `v` riporta una lista di lunghezza `n` con gli elementi tutti uguali a `v`. Le funzioni `lookup` e `update` utlizzano `List.nth: 'a list -> int -> 'a` che, applicata a una lista `lst` e a un intero `n` riporta l'elemento di `lst` che si trova in posizione `n`, dove il primo elemento si trova in posizione 0, il secondo in posizione 1, ecc. Entrambe le funzioni controllano che l'applicazione di `List.nth` non sollevi un'eccezione, cioè che le coordinate siano valide. La funzione `subst`, richiamata da `update`, applicata a un intero `n`, una lista `lst` e un valore `a`, riporta la lista che si ottiene da `lst` sostituendo il suo `n`-esimo elemento con `a`.

```
module Matrix : MATRIX =
struct
  type coord  = int * int
  type 'a t = Mat of coord * 'a list list
  exception Error
  let rec mklist n v =
    if n=0 then []
    else v::mklist (n-1) v
  let make v (r,c) = Mat((r,c),mklist r (mklist c v))
  let size (Mat((r,c),v)) = (r,c)
  let lookup (Mat((r,c),v)) (i,j) =
    if i<1 || j<1 || i> r || j> c
    then raise Error
    else List.nth  (List.nth v (i-1)) (j-1)
  let rec subst n (x::rest) a =
    if n=1 then a::rest
    else x::(subst (n-1) rest a)
```

```
  let update (Mat((r,c),v)) (i,j) n =
    if i<1 || j<1 || i> r || j> c
    then raise Error
    else Mat((r,c),subst i v (subst j (List.nth v (i-1)) n))
  end;;

# let m = Matrix.make 0 (2,3);;
val m : int Matrix.t = <abstr>
# let m1 = Matrix.update m (2,2) 4;;
val m1 : int Matrix.t = <abstr>
# Matrix.lookup m1 (2,2);;
- : int = 4
```

In un senso più astratto, una matrice può essere considerata come una funzione parziale, che si applica a coppie di interi (coordinate) e riporta come valore l'"elemento della matrice" che si trova nella posizione indicata. Ecco dunque un'implementazione alternativa (*funzionale*) delle matrici, dove una matrice è rappresentata da una coppia: le sue coordinate e una funzione.

```
module Matrix1 : MATRIX =
struct
  type coord = int * int
  type 'a t = Mat of coord * (coord -> 'a)
  exception Error
  let make a (r,c) = Mat((r,c), function x -> a)
  let size (Mat((r,c),_)) = (r,c)
  let is_on (Mat((r,c),_)) (i,j) =
      0 <= i && i <= r && 0 <= j && j <= c
  let lookup (Mat(c,f)) c' =
    if is_on (Mat(c,f)) c'
    then f c'
    else raise Error
  let update (Mat(c,f)) c' v =
    if is_on (Mat(c,f)) c'
    then Mat(c,(function x ->
                    if x = c' then v
                    else f x))
    else raise Error
end;;

# let m = Matrix1.make 0 (2,3);;
val m : int Matrix1.t = <abstr>
# let m = Matrix1.update m (2,2) 4;;
val m : int Matrix1.t = <abstr>
# Matrix1.lookup m (2,2);;
- : int = 4
```

5.1.9 Alberi n-ari

Anziché manipolare direttamente la struttura concreta che implementa gli alberi n-ari, possiamo definire un insieme di operazioni di base su di essa e trattare gli alberi come tipo astratto. Le operazioni di base sugli alberi n-ari, che ne consentono la rappresentazione come tipo astratto di dati sono le seguenti:

`node : 'a ->'a ntree`; la funzione `node`, applicata ad un valore N di tipo `'a`, restituisce l'albero n-ario costituito da un unico nodo etichettato da N;

`is_node : 'a ntree -> bool` è il predicato che, applicato a un albero n-ario, restituisce `true` se l'albero è costituito da un unico nodo, `false` altrimenti;

`join : : 'a ntree -> 'a ntree -> 'a ntree` è la funzione che, applicata ad alberi T_1 e T_2 restituisce l'albero che si ottiene aggiungendo a T_1 un nuovo sottoalbero: T_2. Quest'ultimo è, nel risultato della funzione, il primo sottoalbero.

`root : : 'a ntree -> 'a` è la funzione che, applicata a un albero, restituisce la sua radice;

`left : : 'a ntree -> 'a ntree` è la funzione che, applicata a un albero, riporta il suo primo sottoalbero, se esso esiste, un errore altrimenti;

`rest : : 'a ntree -> 'a ntree` è la funzione che, applicata a un albero, riporta l'albero che si ottiene da esso eliminando il suo primo sottoalbero, se questo esiste, un errore altrimenti.

Gli alberi n-ari possono essere implementati come tipo astratto mediante il modulo `Ntree` con segnatura `NTREE`, così definiti:

```
module type NTREE =
  sig
    type 'a t
    exception Error
    val node: 'a -> 'a t
    val join: 'a t -> 'a t -> 'a t
    val root: 'a t -> 'a
    val left: 'a t -> 'a t
    val rest: 'a t -> 'a t
    val is_node: 'a t -> bool
end;;

module Ntree : NTREE =
struct
  type 'a t = Tr of 'a * 'a t list
  exception Error
  let node a = Tr(a,[])
  let join (Tr(a,tlist)) t1 = Tr(a,t1::tlist)
```

```
  let root (Tr(a,_)) = a
  let left = function
      (Tr(_,t1::_)) -> t1
    | _ -> raise Error
  let rest = function
      Tr(a,_::tlist) -> Tr(a,tlist)
    | _ -> raise Error
  let is_node (Tr(_,tlist)) = tlist = []
end;;
```

Come esempio di utilizzazione degli alberi n-ari così implementati, si può considerare la seguente visita in preordine di un albero, che non utilizza i costruttori del tipo `'a ntree`, ma soltanto le funzioni che definiscono il tipo astratto.

```
(* preorder : 'a Ntree.t -> 'a list *)

let rec preorder t =
    Ntree.root t ::
      (if Ntree.is_node t then []
       else (preorder (Ntree.left t))
             @ List.tl (preorder (Ntree.rest t)));;
```

5.2 Funtori

Un funtore è una "funzione" da strutture a strutture. È utile per esprimere una struttura parametrica rispetto a un'altra struttura. In questo testo, considereremo soltanto un esempio molto semplice: un'operazione di ordinamento di liste è parametrica rispetto al tipo degli elementi della lista e alla relazione d'ordine che su di essi si considera. In OCaml non esiste un tipo più generale per gli "insiemi ordinati", ma possiamo definire una segnatura per le strutture rappresentanti insiemi ordinati. Ad esempio:

```
module type ORDER =
    sig
      type t
      val less: t -> t -> bool
    end;;
```

Ecco due esempi di strutture con segnatura `ORDER`: gli interi con l'ordinamento inverso a quello abituale, e le coppie di tipo `int*string`, dove l'ordinamento è determinato dalla componente intera:

```
module RevInt =
  struct
    type t = int
    let less x y = x > y
```

```
  end;;

module DictElem =
    struct
      type t = (int * string)
      let less (x,_) (y,_) = x<y
    end;;
```

Supponiamo ora che `Elt` sia una struttura con segnatura `ORDER`. Potremmo definire un'altra struttura, che fa riferimento a `Elt`, contenente un'operazione di ordinamento per liste di elementi del tipo `Elt.t` (utilizziamo qui l'algoritmo di ordinamento per inserimento, vedi paragrafo 2.6):

```
module EltSort =
  struct
        (* ins: 'a -> 'a list -> 'a list *)
        let rec ins x = function
            [] -> [x]
          | y::ys -> if Elt.less x y then x::y::ys
                     else y::ins x ys
         (* insort: 'a list -> 'a list *)
        let rec insort = function
            [] -> []
          | x::xs -> ins x (insort xs)
  end;;
```

Se ora vogliamo definire una struttura che contenga l'implementazione dello stesso algoritmo di ordinamento ma su liste di elementi di un tipo diverso (o rispetto a un ordine diverso), dobbiamo dichiarare una struttura nuova: la sua dichiarazione sarà del tutto identica a quella di `EltSort` tranne che per il nome della struttura cui si fa riferimento contenente la funzione `less`.

Possiamo allora generalizzare e definire una volta per tutte un funtore che implementa l'algoritmo di ordinamento per inserimento:

```
module Sort =
    functor (Elt: ORDER) ->
      struct
        (* ins: 'a -> 'a list -> 'a list *)
        let rec ins x = function
            [] -> [x]
          | y::ys -> if Elt.less x y then x::y::ys
                     else y::ins x ys
         (* insort: 'a list -> 'a list *)
        let rec insort = function
            [] -> []
          | x::xs -> ins x (insort xs)
      end;;
```

Quando il funtore è "applicato" a una struttura `S` con segnatura `ORDER`, produce una struttura contenente una funzione `insort` per l'ordinamento di liste di elementi di tipo `S.t`, secondo l'ordine rappresentato da `S.less`:

```
# module RevIntSort = Sort(RevInt);;
module RevIntSort :
  sig
    val ins : RevInt.t -> RevInt.t list -> RevInt.t list
    val insort : RevInt.t list -> RevInt.t list
  end

# RevIntSort.insort [39;28;12;48;53;60];;
- : RevInt.t list = [60; 53; 48; 39; 28; 12]

# module DictSort = Sort(DictElem);;
module DictSort :
  sig
    val ins : DictElem.t -> DictElem.t list -> DictElem.t list
    val insort : DictElem.t list -> DictElem.t list
  end

# DictSort.insort [(24,"dotto");(13,"eolo");(5,"pisolo");
                   (120,"brontolo");(32,"biancaneve");
                   (57,"cucciolo")];;
- : DictElem.t list =
[5, "pisolo"; 13, "eolo"; 24, "dotto"; 32, "biancaneve"; 57, "cucciolo";
 120, "brontolo"]
```

5.3 Compilazione separata e codice eseguibile

Quando si scrivono programmi di medie e grandi dimensioni, è utile suddividere il programma in files che possono essere compilati separatamente. OCaml consente l'organizzazione di moduli in files separati, che possono essere compilati in bytecode e poi collegati (*link*) per produrre un file eseguibile.

Nell'organizzazione dei files sorgente, Objective Caml utilizza le seguenti convenzioni sui nomi dei files:

- `<nome-file>.mli`: codice sorgente per la compilazione di interfacce. La compilazione produce un'interfaccia compilata nel file `<nome-file>.cmi`
- `<nome-file>.ml`: codice sorgente per la compilazione di unità di compilazione. La compilazione produce bytecode nel file `<nome-file>.cmo`

Per compilare separatamente un'unità o un'interfaccia si utilizzano i seguenti comandi:

```
ocamlc -c <nome-file>.mli
ocamlc -c <nome-file>.ml
```

Mediante *linking* del bytecode (file `.cmo`) si ottiene un programma eseguibile indipendente.

```
ocamlc -o <nome-programma> <file_1.cmo> ... <file_n.cmo>
ocamlc -o <nome-programma> <f_1.mli> <f_1.ml> ... <f_n.ml>
```

Tra gli argomenti del *linker* ci possono essere anche librerie (`.cma`).

Per eseguire il programma, si invoca l'interprete di bytecode:

```
ocamlrun <nome-programma>
```

(su sistemi Linux, non è necessario invocare esplicitamente `ocmalrun`, ma è sufficiente dare come comando il nome del programma).

Per la compilazione di programmi stand-alone, che possono essere eseguiti anche su un computer che non disponga dell'interprete di bytecode, si utilizza il comando `ocamlc` con l'opzione `-custom`:

```
ocamlc -custom -o <nome-programma> ........
```

Su sistemi Windows occorre il compilatore Microsoft Visual C++ (vedere comunque il manuale).

5.3.1 Moduli e files

Consideriamo, come esempio, l'implementazione di un modulo per le code, organizzata in files separati. L'interfaccia è contenuta nel file `queue.mli`, il cui contenuto è il seguente:

```
(* interfaccia per le code *)
type  'a t
exception Empty
val empty : 'a t
val dequeue : 'a t -> 'a t
val head : 'a t -> 'a
val enqueue : 'a t -> 'a -> 'a t
val null : 'a t -> bool
```

Si noti che nel file non è presente (in quanto "sottintesa") l'intestazione `module type QUEUE = sig ...  end`.

Il modulo Queue.ml è definito nel file `queue.ml`, il cui contenuto è il seguente:

```
type 'a t = 'a list * 'a list
exception Empty
let norm = function ...
let empty = ([],[])
let dequeue = ...
```

```
  let head = ...
  let enqueue (h,t) x = ...
  let null q = q = ([],[])
```

Si noti, anche qui, che il file non contiene (in quanto "sottintesa") l'intestazione `module Queue = struct ... end`.

Il compilatore riconosce un modulo `Queue`, la cui segnatura è quella specificata in `queue.mli`. In altri termini, `queue.mli` definisce l'interfaccia del file di nome `queue.ml`.

5.3.2 Accesso al contenuto di una unità di compilazione

Per le altre unità di compilazione, è come se il contenuto del file `queue.mli` fosse racchiuso tra

```
module type QUEUE =
sig
     ........
end;;
```

e il contenuto del file `queue.ml` fosse racchiuso tra

```
module Queue : QUEUE =
struct
  ...
end;;
```

Possono far riferimento al contenuto del modulo `Queue` nel modo standard, ad esempio:

```
  if Queue.null (Queue.enqueue Queue.empty 3)
  then ...
```

Se si vuole controllare un programma che utilizza altri moduli in modalità interattiva, è possibile caricare bytecode in tale modalità, mediante la direttiva `#load`:

```
  #load "queue.cmo";;
```

In tal modo, il modulo `Queue` viene caricato in memoria, ed è possibile utilizzarlo come se la struttura `Queue` fosse stata definita in modalità interattiva.

5.4 Esercizi

1. Definire una segnatura per il tipo di dato astratto polinomio sugli interi e definire una struttura adeguata con tale segnatura.

2. Definire una segnatura per il tipo di dato astratto polinomio sui razionali e definire una struttura adeguata con tale segnatura.

3. Una pila (*stack*) è un tipo di dato astratto, basato sul modello dei dati delle liste, in cui tutte le operazioni vengono effettuate a un estremo della lista, chiamato la testa (*top*, letteralmente: cima) della pila. Il termine lista "LIFO" (*last-in first-out*, letteralmente: primo entrato primo uscito) è un sinonimo di pila. Le operazioni fondamentali sulle pile sono: `push` (letteralmente: spingere) e `pop` (letteralmente: stappare), dove `push x` mette l'elemento `x` sulla testa della pila e `pop` cancella l'elemento in testa alla pila. Inoltre, è necessaria una costante per la pila vuota e un'operazione di "test pila vuota". Definire una segnatura e implementare un modulo per la realizzazione del tipo di dati astratto pila mediante liste.

4. Un albero genealogico si può rappresentare mediante un albero n-ario di stringhe. Si scrivano alcune funzioni fondamentali a valori booleani che, applicate a un albero genealogico (in cui si suppone che i nodi abbiano tutti etichette diverse) e a due stringhe S e S', determinino se:

   - S è un genitore di S';
   - S è un nonno o una nonna di S';
   - S è un antenato di S';
   - S è un fratello o una sorella di S';

   Scrivere due versioni del programma: utlizzando il tipo concreto `'a ntree` ed il modulo `Ntree`.

5. Definire funtori appropriati che, applicati a strutture con segnatura `ORDER`, generino strutture contenenti un'operazione di ordinamento che implementa l'algoritmo di ordinamento veloce e, rispettivamente, ordinamento per fusione (vedi paragrafo 5.2).