Programmazione Funzionale

Appunti delle Lezioni di Programmazione Funzionale

Anno Accademico: 2024/25

Giacomo Sturm

Dipartimento di Ingegneria Civile, Informatica e delle Tecnologie Aeronautiche
Università degli Studi "Roma Tre"

Sorgente del file LaTeXdisponibile al seguente link:
https://github.com/00Darxk/Programmazione-Funzionale/

Indice

1 Introduzione ad OCaml

La programmazione funzionale è un paradigma di programmazione estremamente potente, utilizza un metodo completamente diverso dagli altri paradigmi di programmazione. L'idea di base consiste nel considerare un programma una funzione, è un linguaggio di alto livello dichiarativo. Il programmatore infatti specifica che cosa deve essere calcolato, non come deve essere calcolato, a differenza di linguaggi imperativi. Il linguaggio di programmazione pià funzionale è Haskell, nel corso verrà usato il linguaggio Objective Caml, contiene aspetti di programmazione orientata agli oggetti. Objective Caml appartiene alla famiglia *Meta Language* ML sviluppata dall'INRIA in Francia dal 1984. Dietro un linguaggio c'è un modello di calcolo che determina le operazioni eseguibili ed anche lo stile di programmazione. Si basa sul lambda calcolo, un linguaggio Turing completo, estremamente semplice.

I tre costrutti fondamentali sono applicazioni, composizione e ricorsione. Su alcune distribuzioni di Linux l'interprete OCaml è preinstallato, ed è possibile interagirci digitando il comando `ocaml` su un terminale:

```
prompt> ocaml
OCaml version x.x.x
Enter #help;; for help.

#
```

Nella modalità interattiva il compilatore effettua il ciclo read-eval-print, legge un'espressione o una dichiarazione, terminata dalla sequenza `;;`, in seguito calcola il valore, deducendo i tipi delle variabili utilizzate, e restituisce la soluzione a schermo. Non c'è bisogno di dichiarazioni esplicite, OCaml può effettuare inferenze di tipo anche su espressioni estremamente complesse. È un linguaggio a tipizzazione statica, ogni tipo può essere determinato a tempo di compilazione. I tipi base di OCaml sono `bool`, `int`, `float`, `string`, `char`, `unit` e `exn`.

1.1 Dichiarazioni e Funzioni

I costrutti di base sono le espressioni, non sono comandi, hanno sempre un valore ed un tipo. Il calcolo procede valutando queste espressioni, semplificandole fino ad ottenere un'espressione non più semplificabile, cioè un valore. I costruttori di controllo principale sono applicazione e composizione di funzioni e ricorsione.

La forma generale di una dichiarazione consiste dalla parola chiave `let` seguita dall'identificatore, a cui si assegna una certa espressione:

```
let <identificatore> <parametri> = <espressione>
```

Per specificare gli argomenti o parametri di una funzione, bisogna specificarli dopo il nome della funzione, per indicare che si tratta di una funzione ricorsiva bisogna utilizzare la parola chiave `rec`, dopo `let`:

```
let rec <identificatore> <parametri> = <espressione>
```

Le espressioni funzionali sono un particolare costrutto di espressioni introdotto dalla parola chiave `function`:

```
function <parametri> -> <espressione>
```

Sono praticamente una funzione anonima, possono essere combinate con una dichiarazione con identificatore, avendo anch'essa un valore, per essere utilizzata all'interno della sua espressione.

Un ambiente è una collezione di legami tra variabili e valori, l'ambiente iniziale comprende tutte queste associazioni presenti nel modulo iniziale contenuto in `Stdlib`. Quando viene aggiunta una nuova dichiarazione viene aggiunto un nuovo legame in cima a questo ambiente. Questo ambiente viene gestito come una pila, quindi dichiarazioni future sovrascrivono dichiarazioni precedenti, poiché vengono accedute prima. Il valore delle variabili globali viene determinato a tempo di compilazione. Per cui è possibile modificare il valore di una variabile globale sovrascrivendola, ma altri oggetti possono comunque riferirsi alla vecchia definizione di questi valori, poiché si riferisce al tempo dove questa dichiarazione è stata inserita nell'ambiente:

```
# let one = 1;;
val one : int =  1
# let oneplus n = n + one;;
val oneplus : int → int = <fun>
# let one = 2;;
val one = 2
# oneplus 1;;
- : int = 2
```

Variabile	Valore
`one`	2
`oneplus`	`function n → n + one`
`one`	1
`StdLib`	

Nel corpo di `oneplus` il valore di `one` viene cercato nel suo ambiente di dichiarazione, nell'ambiente dove è stata definita la funzione `oneplus`. Questo ambiente è costituito da tutti i legami precedenti nella pila. Quando viene applicato un argomento ad una funzione, il suo argomento viene valutato nell'ambiente, viene creato un legame provvisorio del parametro formale con il valore dell'argomento. In questo nuovo ambiente viene valutato il corpo della funzione. Dopo aver determinato il valore della funzione viene eliminato il legame provvisorio.

Quando si effettua una definizione che non dipende dal tipo specificato dei suoi parametri, queste vengono sostituite da variabili di tipo `'a`, che rappresentano un tipo generico. Per indicare che si tratta di un tipo generico viene preceduto da un'apostrofo, convenzionalmente si usano le lettere greche, α in questo caso. Le funzioni così dichiarate si possono applicare a parametri di qualunque tipo.

Le funzioni sono oggetti di prima classe, hanno un proprio valore ed un tipo, quindi è possibile manipolarli, passando funzioni come argomenti ad altre funzioni.

Nelle espressioni di tipo si associa a destra:

```
int → int → int = int → (int → int)
```

Mentre nelle espressioni si associa a sinistra:

```
(func n) m = func n m
```

Per cui l'uso delle parentesi è essenziale per il buon funzionamento del codice.

Le funzioni sono oggetti di prima classe, funzioni di ordine superiore sono dei costrutti che prendono come argomento o riportano come valore una funzione. Funzioni possono essere componenti di una struttura dati, argomenti di altre funzioni o possono essere valori restituiti da altre funzioni.

In questo modo si possono realizzare semplicemente funzioni come la sommatoria, che prendono come argomento la funzione `f` di cui devono effettuare la somma:

```
# let rec sum f (lower, upper) =
    if lower > upper then 0
    else f lower + sum f (lower + 1, upper)
val sum : (int → int) → int * int → int = <fun>
```

La funzione `sum` può essere applicata anche soltanto al suo primo argomento, generando una funzione di prima classe. Invece di inserire una coppia si possono inserire due elementi, *currificando* la funzione, ovvero può funzionare anche con valutazione parziale, generando una funzione che si aspetta i rimanenti parametri, invece di generare un errore. Una funzione in forma currificata calcola gli stessi valori, consumando un solo argomento alla volta:

```
# let rec sum f lower upper =
    if lower > upper then 0
    else f lower + sum f (lower + 1) upper;;
val sum : (int → int) → int → int → int = <fun>
```

Una funzione non currificata non può essere applicata parzialmente. In generale f_c è la forma currificata di f se:

$$f : t_1 \times \cdots \times t_n \to t$$
$$f_c : t_1 \to (t_2 \to \cdots \to (t_n \to t) \cdots)$$

Currificando una funzione è possibile applicarla parzialmente.

Molte operazioni predefinite in OCaml sono in forma currificata. Le operazioni infisse predefinite sono in forma currificata, per utilizzarla in forma infissa si racchiude tra due parentesi tonde `(...)`. Per definire degli operatori infissi si racchiudono tra parentesi nella loro definizione: `let (...) = ...`. Questi operatori possono essere usati in forma infissa e specificando dopo gli argomenti oppure tra se prende due argomenti tra questi senza parentesi:

```
# let (++) x y = 2 * (x + y);;
val (++) : int → int → int = <fun>
# (++) 3 5;;
- : int 16
# 3 ++ 5
- : int 16
```

Per effettuare un'operazione di composizione il codominio della seconda funzione deve essere uguale al dominio della prima funzione. Restituisce una funzione f applicata su g: $f \circ g$, si utilizza la parola chiave `comp`:

```
let comp f g x = f (g x)
```

La composizione è definita come un'operatore infisso `(@@)`.

1.2 Tipi

Un tipo è l'insieme dei valori che può assumere, i tipi predefiniti sono:

- Booleani: `{true, false}`, su cui sono definite le operazioni booleane `not`, `&&` e `||`.
- Interi: `{min_int,...,-1,0,1,...,max_int}`, su cui sono definite le operazioni `+`, `-`, `*`, `/`, `mod`, `succ` e `pred`.
- Numeri reali a virgola mobile `float`, su cui sono definite alcune delle stesse funzioni definite sugli interi, ma per mantenere l'inferenza di tipo vengono seguiti da un punto: `+.`, `-.`, `*.` e `/.`. Oltre a queste sono definite le più importanti funzioni matematiche sui reali. Non è presente la conversione automatica dei tipi numerici.
- Caratteri `char`, definiti sempre tra due apici `'...'`, si possono convertire in codice ASCII corrispondente e viceversa con `int_of_char` e `char_of_int`.
- Stringhe tra doppi apici `"..."`, che possono essere concatenate con l'operatore di concatenazione `^`, si può accedere ai singoli caratteri con la notazione puntata specificando la posizione `.[i]`, la funzione `string_of_int` converte un numero intero in una stringa.

Il tipo `unit` ha un unico elemento `()` e può essere usato al posto di qualsiasi tipo, si comporta come un super-tipo per tutti i tipi semplici.

Se si utilizzano funzioni definite su interi su reali o viceversa, viene generato un errore, sono comunque presenti funzioni per convertire tra questi tipi, ma operazioni di casting vengono sconsigliate. Gli operatori di confronto `<`, `>` e `=` sono definiti su qualsiasi tipo, eccetto che sulle funzioni. Nella notazione di OCaml il non uguale si rappresenta come `<>`. Si possono effettuare confronti tra tuple, controllando secondo l'ordine lessicografico partendo dalla prima componente.

1.3 Espressioni Condizionali

Si possono realizzare istruzioni condizionali con il costrutto `if E then F else G`, dove `E` è un booleano, mentre `F` e `G` sono espressioni dello stesso tipo o almeno un sottotipo in comune, e nei linguaggi ML deve essere possibile determinarlo a tempo di compilazione, essendo linguaggi fortemente tipati.

Non è un costrutto di controllo come nei linguaggi imperativi, ma è un'espressione con un valore ed un tipo, il più generale tra `F` e `G`. Viene valutata in maniera pigra, se `E` è vera, non viene valutata `G`, mentre se è falsa non viene valutata `F`.

Le seguenti espressioni sono quindi equivalenti:

- `E && F`: `if E then F else false`.
- `E || F`: `if E then true else F`.

Il costrutto `else` viene sempre indentato sotto i rispettivi `if`, e deve essere presente la parte `else`, non è presente ambiguità.

Per valutare le espressioni in linguaggi ML si utilizza la regola di calcolo *call by value*, la valutazione per valore, dove si calcola il valore dell'argomento prima di applicare la funzione, invece della valutazione per nome *call by name*, questa regola viene utilizzata solamente nelle espressioni condizionali ed operatori booleani. La regola di calcolo per nome applica la funzione prima di aver calcolato il valore dell'argomento. Se non si valutassero le espressioni booleane in modo pigro, ogni funzione ricorsiva che utilizza una funzione booleana per effettuare la ricorsione, valuterebbe all'infinito il passo ricorsivo, senza poter mai fermarsi.

1.4 Coppie

Le coppie ordinate sono formate da due elementi divisi da una virgola tra parentesi tonde: `(E, F)`. Per rappresentare un costrutto di tipo si utilizza l'operatore `*` per indicarlo, il prodotto cartesiano non è associativo. Una tupla di tuple è diversa da una tripla: `int * (int * int) <> int * int * int`. Sulle coppie si utilizzano le funzioni `fst` e `snd` per restituire il primo ed il secondo elemento della coppia, queste sono polimorfe, ma se vengono usate su tuple di più di due elementi restituiscono un errore.

Le coppie come tutti i tipi di dati su OCaml sono definiti da un insieme di costruttori che creano valori di quel tipo, ed un insieme di selettori, operazioni per selezionare componenti da un valore di tipo. I tipi semplici non hanno selettori, ma solo costruttori, questi sono tutti i valori del tipo. Per le coppie il costruttore è `(,)`, l'insieme tra parentesi e virgola, applicato ad un'espressione di tipo α e β. Mentre i selettori sono `fst` e `snd`.

2 Ricorsione e Pattern

2.1 Definizioni Ricorsive

Nei linguaggi funzionali non esistono costrutti di controllo, ma il principale meccanismo di controllo è la ricorsione, realizzato tramite la parola chiave `rec`. In linguaggi funzionali "puri", non sono presenti costrutti di controllo per realizzare cicli. Per risolvere un problema ricorsivamente bisogna identificare i casi base, che possono essere risolti immediatamente. Inoltre bisogna identificare sotto-problemi più semplici di un generico problema complesso, che possono aiutare ad individuare la sua soluzione. Supponendo di poter risolvere questi problemi, l'ipotesi di lavoro, bisogna operare sulla soluzione di questi sotto-problemi per ottenere la soluzione del problema complesso.

2.1.1 Esempio: `split_string`

Si considera il problema di valutare un'espressione aritmetica rappresentata in una stringa. L'operazione è tra due interi non negativi, la funzione offerta come soluzione dovrebbe riportare il risultato di questa espressione. Le operazioni consentite sono la somma, differenza, prodotto e divisione intera. Si avrà una funzione: `evaluate: string -> int`. Applicata su una variabile s, di tipo stringa deve restituire il risultato dell'espressione rappresentata, o un errore se non rispetta le condizioni d'uso.

Un sotto-problema utile consiste nel suddividere la stringa in tre parti, i due interi operandi ed il carattere che identifica l'operatore: `split_string: string` $\rightarrow$ `int * char * int`. Applicata ad una stringa s restituisce una tripla (n, op, m).

Per risolvere questo problema bisogna individuare quale dei caratteri della stringa non è un numero, si può effettuare con un'altra funzione `primo_non_numerico: string` $\rightarrow$ `int`. Data una stringa s passata come input restituisce la posizione i-esima del primo carattere non numerico nella stringa.

Si può definire inoltre un'altra funzione per tagliare una porzione della stringa tra due posizioni fornite `substring: string` $\rightarrow$ `int` $\rightarrow$ `int` $\rightarrow$ `string`. Applicata ad una stringa s, restituisce la sotto-stringa dalla posizione j-esima alla posizione k-esima.

Dal moduli Pervasives si può usare la funzione `int_of_string` per restituire l'intero corrispondente ad una stringa. Dal modulo String, si possono usare la funzione `sub` e `length` per restituire una sotto-stringa la prima e la lunghezza di una stringa la seconda. Queste funzioni producono un errore se la stringa non corrisponde ad un intero la prima, ovvero se sono presenti caratteri non numerici, e se gli indici forniti non appartengono alla stringa, la seconda.

Per controllare i caratteri di una stringa si usa la funzione `get` del modulo String, abbreviata in notazione puntata come `.[i]`. Si definisce quindi la funzione per trovare il primo carattere non numerico:

```
let rec loop s i =
    if (s.[i] < '0' || s.[i] > '9') then i
    else loop s (i + 1);;
let primo_non_numerico s = loop s 0;;
```

2.2 Dichiarazioni Locali

All'interno di una funzione si possono dichiarare ulteriori espressioni, con il costrutto `let...in`, dove si dichiara l'espressione normalmente con `let`, seguito da `in`, dove vengono passati i parametri da applicare all'espressione locale. Le variabili definite nella dichiarazione locale sono variabili locali, inizializzate nel `in`, queste hanno un valore solamente all'interno dell'espressione locale. Quando viene valutata la funzione, questa variabile non ha più valore, viene quindi creato un legame temporanea nell'ambiente quando viene invocata quest'espressione locale tramite l'`in`. All'interno di una funzione locale, sono visibili i parametri globali della funzione esterna, e possono essere usati e modificati all'interno di espressioni locali.

Data una dichiarazione locale `let x = E in F` in un ambiente $\mathscr{A}$, viene calcolato il valore `v` della dichiarazione `E` di `let`, e viene provvisoriamente legata alla variabile `x` estendendo $\mathscr{A}$. In questo nuovo ambiente viene calcolato il valore dell'espressione `F` di `in`. Il valore così calcolato di `F` diventa il valore dell'intera espressione. Dopo averlo calcolato viene sciolto il legame provvisorio di `x`, e viene ripristinato l'ambiente $\mathscr{A}$. Un'espressione locale è quindi equivalente a: `let x = E in F` $\Leftrightarrow$ `(function x` $\rightarrow$ `F) E`.

Si possono usare dichiarazioni locali per evitare di dover calcolare più volte il valore di un'espressione, usata più volte. Una funzione può essere dichiarata localmente se non ha un significato autonomo, all'esterno della funzione dove viene usata e se consente di diminuire il numero di parametri.

2.2.1 Esempio: `split_string`

Utilizzando dichiarazioni locali è possibile inserire in una sola funzione tute le funzioni ausiliarie definite precedentemente:

```
let primo_non_numerico s =
    let rec loop i =
        if (s.[i] < '0' || s.[i] > '9') then i
        else loop (i+1)
    in loop 0
```

La funzione `substring` viene realizzata tramite la funzione `sub` del modulo String:

```
let substring s j k =
    String.sub s j ((k-j)+1);;
```

Si può quindi realizzare la funzione `split_string` in questo modo:

```
let split_string s =
    let i = primo_non_numerico s
    in (int_of_string (substring s 0 (i - 1)),
        s.[i],
        int_of_string (substring s (i + 1) ((String.length s) - 1)))
```

La soluzione al problema descritto, è quindi data dalla seguente funzione `evaluate`:

```
let evaluate s =
    let (n, op, m) = split_string s
    in if op = '+' then (n + m)
       else if op = '-' then (n - m)
            else if op = '*' then (n * m)
                 else if op = '/' then (n / m)
                      else ???
```

Si incontra un problema poiché la funzione deve restituire un risultato quando la stringa di input non rispetta le condizioni di utilizzo della funzione. Dovrebbe restituire un errore, oppure sollevare un messaggio di errore se il valore di *op* non coincide ad uno dei quattro operatori. Il problema è che la funzione è di tipo `string` → `int`, quindi può solamente restituire un numero intero, anche se l'espressione aritmetica fornita non è valida per poterne calcolare il valore.

2.3 Eccezioni

Si possono definire nuove eccezioni, tramite la parola chiave `exception`, tutte le eccezioni sono di tipo `exn`, permettono di scrivere programmi che segnalano errori. Invece di restituire un valore un'espressione può sollevare un'eccezione. Tutte le funzioni sono in grado di sollevare un'eccezione e terminare immediatamente la loro esecuzione, invece di restituire un valore.

Esiste un insieme di eccezioni predefinite, iniziano sempre con una lettera maiuscola. Dopo che è stata dichiarata un'eccezione è anche possibile sollevarla, utilizzando la parola chiave `raise`, seguita dall'identificativo dell'eccezione. Se durante il calcolo di un'espressione viene sollevata un'eccezione, allora il calcolo del valore termina immediatamente generando l'eccezione. Un'eccezione può essere catturata con un costrutto simile al try-catch di Java, chiamato `try-with`. Nel `try` viene inserita un'espressione da calcolare, se viene sollevata un'eccezione durante il calcolo del valore, controlla se il tipo dell'eccezione sollevata corrisponde all'eccezione presente nel costrutto `with`, seguito da `->` che indica un'espressione da eseguire in caso sia verificata l'eccezione. Un'espressione di tipo `exn` può essere il valore di qualsiasi funzione, ed argomento di qualsiasi altra funzione, questo è un eccezione per la tipizzazione forte di OCaml. Si possono usare eccezioni predefinite di OCaml per sollevare eccezioni proprie fornendo commenti più descrittivi. Il nome di tutte le eccezioni comincia con una lettera maiuscola.

Le eccezioni vengono propagate, se durante il calcolo del valore di un'espressione `E` viene sollevata un'eccezione e non viene catturata, se `E` è un'espressione locale, allora l'eccezione può essere catturata all'interno dell'espressione esterna. Se non viene catturata può continuare ad essere propagata in alto fino alla prima espressione chiamante, e se non viene catturata neanche a questo punto termina l'esecuzione dell'interno programma.

Un uso non elegante delle eccezioni consiste nell'usare un'eccezione sollevata in un costrutto `try-with` per definire dei casi base per una funzione ricorsiva. In questo modo l'interruzione dell'esecuzione causata dall'eccezione restituisce il valore del caso base dal `with`, mentre per gli altri casi non viene sollevata l'eccezione quindi l'esecuzione continua come se fosse stato implementato un caso base con un controllo.

2.3.1 Esempio: `split_string`

Utilizzando un'eccezione si può risolvere il problema proposto, definendo una nuova eccezione e sollevandola se l'operatore non corrisponde a nessuno degli operatori permessi:

```
exception BadOperation;;
let evaluate s =
    let (n, op, m) = split_string s
    in if op = '+' then (n + m)
       else if op = '-' then (n - m)
            else if op = '*' then (n * m)
                 else if op = '/' then (n / m)
                      else raise BadOperation;;
```

A questo punto utilizzando le eccezioni si possono inserire dei controlli nelle altre funzioni ausiliarie dentro al programma. Se non sono presenti caratteri non numerici, allora dovrebbe essere sollevata l'eccezione `BadOperation`:

```
let primo_non_numerico s =
    let rec aux i =
        if (s.[i] < '0' || s.[i] > '9'  ) then i
        else aux (i+1)
    in try aux 0
       with Invalid_argument "index out of bounds" -> raise BadOperation
```

L'eccezione catturata si verifica quando si esce fuori dall'indice consentito di una stringa. Si utilizza il nome `aux` per indicare una funzione ausiliaria generica.

La funzione `split_string` può fallire quando viene chiamata `int_of_string` se la sotto-stringa che le viene passata non corrisponde ad un intero, dato che solo il primo carattere non numerico viene rimosso, se ne è presente più di uno questo provoca il fallimento della funzione. Si può quindi definire una nuova eccezione:

```
exception BadInt;;
let split_string s =
    let i = primo_non_numerico s
    in try (int_of_string (substring s 0 (i - 1)),
            s.[i],
            int_of_string (substring s (i + 1) ((String.length s) - 1)))
       with Failure "int_of_string" -> raise BadInt
```

2.4 Pattern

L'uso di costrutti condizionali è molto usato e comune, e sarebbe utile per molte applicazioni poter semplificare la loro scrittura. In diversi linguaggi di programmazione sono presenti costrutti simili ad uno *switch-case*, l'equivalente in OCaml e nei linguaggi ML sono i *pattern*. Quando si effettua una dichiarazione su una variabile `x`, questo è un caso particolare di pattern. Una dichiarazione è quindi

generalmente costituita dalla parola chiave `let`, seguita da un pattern, a cui viene assegnato il valore di un'espressione. La forza principale di OCaml consiste in questa abilità di *pattern matching*. Un pattern è un'espressione costituita da variabili e/o costruttori di tipo, per i tipi introdotti fin'ora i costruttori sono tutti e soli i valori dei tipi `int`, `float`, `bool`, `char`, `string` ed `unit`, ed i costruttori di tuple: `(,)`. In un pattern però non possono esserci ripetizioni di una stessa variabile, eccetto la variabile muta `()`. Espressioni condizionali o contenenti operatori aritmetici non sono pattern, il pattern matching costituito da una sola variabile e qualunque espressione ha sempre esito positivo, non avendo restrizioni di alcun modo sui tipi.

Dato un pattern `P`, un certo valore `V` si dice conforme al pattern `P`, se è possibile sostituire le variabili in `P` con il valore di sotto-espressioni di `V`, in modo da ottenere `V` stesso. Ogni espressione `E` ha un suo valore `V`, quindi si può generalizzare la concezione di pattern matching. L'operazione di pattern matching consiste nel confrontare un espressione `E` con un pattern `P`. Il confronto ha successo se il valore ottenuto `V` è conforme al pattern `P`, se ha successo allora si determina come sostituire le variabili di `P` per ottenere il valore `V`. Quindi viene esteso l'ambiente corrente, inserendo i legami risultanti dal pattern matching.

Anche nelle dichiarazioni di funzioni i parametri sono pattern, permettendo di evitare l'uso di selettori all'interno del corpo della funzione. Le espressioni funzionali si scrivono allo stesso modo di una dichiarazione, utilizzando pattern. Definendo una funzione in maniera esplicita o utilizzando espressioni funzionali, il parametro può essere sempre un pattern.

In generale quindi le espressioni `function` sono della forma:

```
function  P1 -> E1
        | P2 -> E2
        |   ...
        | Pn -> En
```

Dove ogni pattern `Pi` ha lo stesso tipo `Tp` ed una sua espressione associata di stesso tipo `Te`. Il tipo dell'espressione `function` è quindi `Tp` $\rightarrow$ `Te`. Si possono anche utilizzare pattern multipli, specificando i pattern da assegnare ad una certa espressione divisi da `|`. In questo modo si possono scrivere in modo estremamente sintetico e semplice espressioni e funzioni:

```
let F = function  P1 -> E1
                | P2 -> E2
                |   ...
                | Pn -> En
```

Per valutare questa funzione `F` applicata ad un espressione `E`, viene calcolato il valore dell'argomento `V` e si effettua pattern matching con questo valore `V`. A questo punto si effettua il processo descritto precedentemente ed il valore dell'espressione risultante dal pattern match viene riportato come il valore dell'espressione `F E`, e vengono sciolti i legami provvisori.

Si considera un espressione per il calcolo del fattoriale:

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

L'ordine è estremamente importante, poiché dopo aver calcolato il valore dell'argomento, se il valore è applicata ad un'espressione, questo valore viene confrontato in ordine dal primo pattern in poi. Se il confronto con un pattern ha successo vengono creati legami nell'ambiente con il valore di argomento, poi viene calcolata l'espressione della funzione tramite questo valore, e vengono rimossi i legami precedentemente creati, mantenendo solamente quelli tra argomento e risultato. Quindi anche se il valore potrebbe effettuare un pattern match con pattern successivi, questo viene effettuato solamente con il primo match.

Si può utilizzare il simbolo `_` come una *wildcard*, può effettuare un pattern match con qualsiasi tipo, ma non viene salvato questo legame nell'ambiente. In un costrutto try-with può essere usata per catturare qualsiasi tipo di eccezione che viene sollevata.

Un altro costrutto con pattern matching molto utile è il match-with, questo permette di confrontare un'espressione data in input con diversi pattern dello stesso tipo dell'espressione di ingresso. Questo costrutto permette quindi di scegliere quale variabile deve essere confrontata, al contrario del costrutto per `function` che prende l'argomento che gli viene passato, può essere utile in funzioni dove sono richiesti diversi argomenti. Valgono le stesse regole e condizioni generali per i pattern discusse precedentemente:

```
match E with
      P1 -> E1
    | ...
    | Pn -> En
```

Per valutare queste espressioni viene valutata l'espressione in input e ne viene computato il valore `V`, confrontandolo con ogni `P` pattern in ordine. Viene creato il legame con il primo pattern `Pi` che effettua un match ed aggiunto all'ambiente. In seguito viene calcolato il valore dell'espressione `Ei` corrispondente e sciolti i legami ausiliari creati.

Si può utilizzare per definire funzioni esplicitando i parametri presi in argomento, si considera l'esempio del fattoriale:

```
let rec fact n =
    match n with
          0 -> 1
        | _ -> n * fact(n - 1)
```

Quando il pattern matching non è esaustivo OCaml individua il problema, ma è comunque interpreta l'espressione. Se non si riesce ad effettuare alcun un match viene sollevato un errore di tipo `Match_failure`.

2.4.1 Esempio: `split_string`

Utilizzando il pattern matching si può semplificare in modo considerevole la precedente implementazione della funzione, rimuovendo i vari costrutti condizionali annidati con un solo costrutto match:

```
let evaluate s =
    let (n, op, m) = split_string s
```

```
in match op with
      '+' -> n + m
    | '-' -> n - m
    | '*' -> n * m
    | '/' -> n / m
    |  _  -> raise BadOperation
```

3 Costrutti Imperativi e I/O

Su OCaml nel modulo Pervasives sono presenti delle funzioni per la stampa sullo standard output `stdout`, `print_string` e `print_int` queste funzioni sono di tipo:

```
val print_string: string → unit
val print_int: int → unit
```

Dove il tipo `unit` contiene un solo elemento `()`. Si vuole realizzare una funzione che permette di stampare a schermo tutti gli interi compresi tra due argomenti n e m.

In OCaml si può realizzare una sequenza di comandi con una notazione simile a quella di una tupla, solamente utilizzando `;` al posto di virgole semplici `,`:

```
( E1 ; E2 ; ... ; En)
```

Se `Ei` sono espressioni, allora anche questo costrutto è un'espressione formata dalla concatenazione di queste espressioni. Il tipo ed il valore di quest'espressione è dato dall'ultima espressione `En`. Tutte queste espressioni vengono valutate da sinistra verso destra, ma i loro valori vengono ignorati, tranne quello dell'ultima espressione. Quindi si potrebbe utilizzare questo costrutto per effettuare tutte le stampe necessarie da questa funzione. Se sono presenti degli errori in una di queste espressioni vengono ignorati, e continua la valutazione delle espressioni successive.

Risulta scomodo provare ad implementare la funzione richiesta utilizzando questi costrutti, poiché dovrebbe essere dinamica rispetto al numero di stampe da effettuare. Si vuole implementare iterativamente, anche se OCaml non fornisce costrutti imperativi, è possibile simulare il loro comportamento iterativo nel modo seguente:

```
let rec ciclo n m =
    if n > m then ()
    else (print_int n;
          print_newline();
          ciclo (n + 1) m)
```

Questa funzione ricorsiva implementa un'iterazione poiché è *tail recursive*, al ritorno della chiamata ricorsiva, non deve eseguire nulla.

Si considera una funzione `conta_digits: stringa → int`, che restituisce il numero di caratteri numerici in una stringa s, realizzata in questo modo non è iterativa:

```
let conta_digits s =
    let rec loop i =
        try if (s.[i] >= '0' && s.[i] <= '9') then 1 + loop (i + 1)
            else loop (i + 1)
        with _ -> 0
    in loop 0
```

Poiché la funzione `loop` non è tail recursive, dato che alla fine di una ricorsione deve ancora effettuare l'operazione di addizione, sulla base del risultato del seguente passo ricorsivo `1 + loop (i + 1)`. Per renderla iterativa, bisogna effettuare quest'operazione prima dell'esecuzione del seguente passo

ricorsivo. Si utilizza quindi una variabile di accumulazione `acc`, che ad ogni passo ricorsivo contiene il risultato del passo. In questo modo ad ogni passo ricorsivo si sono già effettuate tutte le operazioni, prima di passare al passo seguente, quindi rappresenta un algoritmo iterativo:

```
let conta_digits s =
    let rec loop i acc =
        try if (s.[i] >= '0' && s.[i] <= '9') then loop (i + 1) (acc + 1)
            else loop (i + 1) acc
        with _ -> acc
    in loop 0 0
```

Questo accumulatore `acc` rappresenta il risultato parziale di un'iterazione, inizializzato a zero. Questo è il modo in cui vengono realizzati algoritmi ricorsivi in OCaml, dove non sono presenti costrutti imperativi.

Per leggere input da `stdin` sempre nel modulo Pervasives sono presenti le funzioni `read_line` e `read_int` di tipo `unit` → `string` o `int`. Si vuole realizzare una funzione che legga degli interi terminata da un punto, e ne restituisca la loro somma:

```
let rec somma () =
    let s = read_line
    in if s="." then 0
       else (int_of_string s) + somma ()
```

Per realizzare lo stesso algoritmo in versione iterativa, bisogna implementare un accumulatore, in una funzione ausiliaria `aux`:

```
let somma () =
    let rec aux acc =
        let s = read_line ()
        in if s="." then acc
           else aux ((int_of_string s) + acc)
    in aux 0
```

La funzione ausiliaria `aux` applicata su un argomento n restituisce n sommato agli interi letti dallo `stdin`. Si possono utilizzare le eccezioni, in modo poco elegante, per rendere più sintetico questo approccio. Invece di controllare se il carattere letto è `.` si può effettuare la conversione ad intero, in un try-with, e se la conversione non ha successo, allora termina l'esecuzione, poiché si suppone sia stato passato il carattere `.`. Dato che il comportamento della funzione è analogo al precedente, in questo modo anche se viene inserito un carattere che non è un intero non viene terminata l'esecuzione senza output:

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

Nei linguaggi iterativi non esiste l'assegnazione quindi un ciclo viene implementato da un costrutto ricorsivo. Questo è la funzione `aux`, che presenta una variabile in più, l'accumulatore `acc` per memorizzare il risultato parziale dell'iterazione. Gli argomenti di questa funzione sono le variabili che vengono modificate nel ciclo. L'operazione principale richiama quella ausiliaria, inizializzando l'accumulatore. In generale questa funzione ausiliaria può avere un corpo del tipo:

```
if <condizione-uscita> then <valore-uscita>
else <chiamata-ricorsiva> <argomenti-modificati>
```

Gli argomenti modificati sono variabili modificate durante l'iterazione, compreso l'accumulatore. Si considera quindi una possibile implementazione iterativa dell'operazione fattoriale, precedentemente definita ricorsivamente come:

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

Iterativamente diventa:

```
let fact' n =
    let rec aux acc = function
          0 -> acc
        | n -> aux (n * acc) (n - 1)
    in aux 1 n
```

Questa funzione `aux` è iterativa, poiché dopo aver raccolto il risultato per un'iterazione, non deve eseguire altre operazioni. Mentre in un processo ricorsivo, dopo aver ottenuto il risultato della ricorsione bisogna effettuare altre operazioni, nel caso del fattoriale bisogna moltiplicare il valore ottenuto, il fattoriale di $n-1$, per n.

Un processo ricorsivo esegue le operazioni al ritorno dalla ricorsione, mentre in un processo iterativo le operazioni vengono svolte prima della ricorsione l'ultima chiamata ricorsiva può riportare il suo risultato direttamente alla prima. Inoltre un processo ricorsivo ha uno spazio lineare al numero di chiamate ricorsive, mentre in un processo iterativo si usa spazio costante, poiché non c'è bisogno di salvare in memoria le altre chiamate ricorsive.

Se un problema P_1 può essere convertito in un altro problema P_2 in modo che una soluzione al primo sia anche soluzione al secondo, senza siano necessari ulteriori calcoli, si dice che il problema P_1 è stato ridotto in P_2, analogamente si dice che P_2 è riduzione di P_1. Se una funzione ricorsiva è definita in modo che tutte le sue chiamate ricorsive sono delle riduzioni, allora è una funzione ricorsiva di coda, tail recursive.

4 Liste

Le liste sono sequenze finite di elementi dello stesso tipo, dove `list` è il costruttore delle liste, preceduto dal tipo degli elementi della lista. La lista vuota è un oggetto polimorfo, indicato con `[]`, di tipo `'a list`. L'inserimento in testa, *cons*, è un'operazione fondamentale denotata da `::`, infisso. I costruttori per le liste sono:

```
[] : 'a list
:: : 'a -> 'a list -> 'a list
```

Si possono quindi realizzare liste in modo induttivo, data una lista vuota `[]`, di tipo α `list`, se `x` è di tipo α, ed `xs` è di tipo α `list`, allora `(x::xs)` è una α `list`, e nient'altro è un α `list`. In questo modo è possibile generare tutte le possibili liste per un dato tipo, per stadi, realizzando un albero radicato in `[]`, ed ogni figlio rappresenta l'aggiunta di un elemento del tipo tramite `x::xs`, dove `xs` è la lista corrispondente al nodo padre.

Le liste possono essere costruite ricorsivamente partendo da una lista vuota `[]`, supponendo di poter calcolare il valore su una lista `xs`, si determina come calcolare il valore della generica lista `x::xs`.

Nel modulo List sono definite funzioni che operano sulle liste, alcune di queste sono:

- `List.hd: 'a list → 'a`
- `List.tl: 'a list → 'a list`
- `List.length: 'a list → int`
- `List.flatten: 'a list list → 'a list`
- `List.sort: ('a → 'a → int) → 'a list → 'a list`
- `List.assoc: 'a → ('a * 'b) list → 'b`
- `List.rev: 'a list → 'a list`
- `List.mem: 'a → 'a list → bool`
- `List.split: ('a * 'b) list → 'a list * 'b list`
- `List.combine: 'a list → 'b list → ('a * 'b) list`
- `List.sort: ('a → 'a → int) → 'a list → 'a list`
- `List.merge: ('a → 'a → int) → 'a list → 'a list → 'a list`

Le funzioni *head* `hd` e *tail* `tl` sono i selettori di una lista, il primo restituisce il primo elemento ed il secondo restituisce la lista corrispondente senza il primo elemento. Non sono definiti sulla lista vuota `[]`, quindi applicati su di essa viene sollevata un'eccezione.

`length` restituisce un intero che rappresenta il numero di elementi nella lista. `flatten` trasforma una lista di liste in un'unica grande lista, se gli elementi delle liste contenute nella lista interna non

coincidono produce un errore. La funzione `sort` ordina una lista, in base ad una relazione d'ordine espressa come funzione fornita come argomento della funzione. Questa funzione di ordinamento deve essere di tipo `('a → 'a → int)`, e deve restituire 1 se il primo elemento è maggiore del secondo, 0 se sono uguali, oppure -1 se il primo elemento è minore, secondo una certa relazione d'ordine.

Si vuole implementare la funzione `length`, senza utilizzare il modulo List. Per implementarla ricorsivamente si considera una lista vuota di lunghezza 0; mentre per un caso generico, la sua lunghezza è $1 + n$, dove n è la lunghezza della lista rimosso il primo elemento, si suppone si possa calcolare questo valore n. La definizione è valida poiché ripetendo l'operazione di coda si arriverà necessariamente alla lista vuota `[]`, la funzione è quindi definita utilizzando il selettore di coda:

```
let rec length l =
    if l = [] then 0
    else 1 + (length (List.tl l))
```

Alternativamente si può utilizzare il pattern matching sulla lista:

```
let rec length = function
      [] -> 0
    | _::xs -> 1 + (length xs)
```

In un pattern possono occorrere solo variabili o costruttori, `[]` e `::`. Utilizzando questi due costruttori si può individuare il caso base di una lista vuota `[]` ed il caso con una lista avente almeno un elemento `x::xs`, dove a `x` viene assegnato il valore del primo elemento della lista, ed a `xs` viene assegnato il valore della lista rimanente. Questa modalità è un'alternativa all'uso dei selettori di testa e di coda per le liste. Analogamente si può effettuare pattern matching per un certo numero di elementi specifico in una lista con `[x1;...x;n]`, questo pattern avrà successo solo se la lista su cui si vuole effettuare il match ha esattamente n elementi ed il valore di tutti i componenti `xi` assumerà il valore corrispondente dell'elemento $i-$esimo della lista. Oppure si può realizzare con `x1::...::xn::xs`, dove `xs` conterrà la lista vuota `[]` se la lista su cui si è effettuato il match ha esattamente n elementi, altrimenti sarà composta dalla coda della lista.

Per realizzare la funzione `length` ricorsivamente si considera la seguente implementazione:

```
let length' l =
    let rec aux acc = function
        | [] -> acc
        | _::xs -> aux (1 + acc) xs
    in aux 0 l
```

Un'altra funzione simile che si può implementare calcola il prodotto di tutti gli interi di una lista `prodof: int → int`. Si realizza sia in modo ricorsivo che iterativo:

```
let rec prodof = function
    | [] -> 1
    | x::xs -> x * (prodof xs)
;;
```

```
let prodof' l =
    let rec aux acc = function
        | [] -> acc
        | x::xs -> aux (x * acc) xs
    in aux 1 l
;;
```

4.1 Esempio: `super`

Si vuole creare una funzione che, date le ultime n estrazioni del superenalotto, restituisca i numeri che sono stati estratti meno volte. Questa funzione `super` prende come parametro la lista di liste `est`, rappresentanti le ultime estrazioni, il numero di interi di ogni estrazione n ed il massimo numero che può essere estratto h:

```
- super: int list list → int → int → int list
super est n h
```

Si definisce il sotto-problema di determinare quante volte un numero m compreso da 1 ad h compare in tutte le estrazioni `est`. Per poi iterare su tutti i possibili numeri che possono essere estratti. Per determinare questi numeri, si definisce la funzione `upto: int → int → int list`, applicata sugli argomenti i j produce una lista da i a j: `upto i j =[i;...;j]`:

```
let rec upto i j =
    if i > j then []
    else i::(upto (i + 1) j)
```

Si può realizzare in modo ricorsivo come il seguente:

```
let upto' i j =
    let rec aux acc i' j' =
        if i' > j' then acc
        else aux (j'::acc) i' (j' - 1)
    in aux [] i j
```

L'ultima riga potrebbe essere scritta come `else aux (m'::acc) (m'+1) n'`, ma in questo modo si inverte l'ordine della lista creata. Nelle due versioni si aggiunge sempre in testa, partendo dall'estremo inferiore nel primo caso e dall'estremo superiore nel secondo caso.

Per poter utilizzare facilmente la lista delle estrazioni, bisogna appiattirla, da una lista di liste ad una lista, quindi bisogna definire una funzione `flatten: 'a list list → 'a list`:

```
let rec flatten = function
      [] -> []
    | x::xs -> x@(flatten xs)
```

La funzione `flatten` in maniera ricorsiva ha la seguente implementazione:

```
let rec flatten ll
    let aux acc = function
          [] -> acc
        | x::xs -> flatten (acc@x) xs
    in aux [] ll
```

Si definisce quindi una funzione per contare quante volte uno solo dei numeri possibili è stato estratto, che verrà applicata su tutti i possibili numeri, `conta: 'a` → `'a list` → `int`:

```
let rec conta x = function
      [] -> 0
    | y::ys -> if y = x then 1 + (conta x ys)
               else conta x ys
```

In forma iterativa diventa:

```
let conta x l =
    let rec aux x' l' acc = match l with
              [] -> acc
            | y::ys -> if y = x' then aux x' ys (acc + 1)
                       else aux x' ys acc
    in aux x l 0
```

Si definisce ora una funzione per contare tutti questi possibili numeri, `contatutti: 'a list` → `'a list` → `('a * int) list`. Il primo argomento l_1 è la lista contenente i possibili numeri, mentre il secondo l_2 è la lista contenente tutte le estrazioni. Questa funzione restituisce una lista di coppie, dove il primo elemento rappresenta l'elemento cercato, mentre il secondo è il suo numero di occorrenze:

```
let rec contatutti l1 l2 = match l1 with
      [] -> []
    | x::xs -> (x, conta x l2)::(contatutti xs l2)
```

Utilizzando le funzioni definite si può ottenere la lista delle coppie dei possibili numeri e la loro occorrenza:

```
contatutti (upto 1 h) (flatten est)
```

Per scegliere gli elementi che sono stati estratti il numero minore di volte, si può ordinare la lista, utilizzando una funzione `sort: ('a * 'b) list` → `('a * 'b) list`. Si può usare la funzione `List.sort` fornita da OCaml nel modulo List, questa può ordinare una lista passata come parametro rispetto ad una certa relazione d'ordine espressa tramite un'espressione passata come primo argomento alla funzione. Si definisce quindi una relazione d'ordine `comp: ('a * 'b)` → `('c * 'b)` → `int` tra due elementi, o coppie, di questa lista:

```
let comp (_, x) (_, y) = if x < y then -1
                         else if x = y then 0
                         else 1
```

Si definisce ora la funzione `sort` come:

```
let sort l = List.sort comp l
```

Ottenuta la lista ordinata, per ottenere una lista dei numeri meno estratti, bisogna prima prendere un certo numero di elementi dalla lista ordinata, tramite una funzione `take: int → 'a list → 'a list`, che applicata ad un intero n ed una lista l restituisce una sottolista di lunghezza n, partendo dalla testa della lista l:

```
let rec take n = function
      [] -> []
    | x::xs -> if n <= 0 then []
               else x::(take (n - 1) xs)
```

Per realizzarla in modo ricorsivo, non si può utilizzare l'operatore cons, altrimenti ad ogni passaggio si aggiungerebbe la testa della lista all'accumulatore, invertendo l'ordine `x::acc`. Questo avviene perché in un approccio iterativo si effettuano tutte le operazioni in un passo prima di effettuare la ricorsione, quindi viene aggiunta subito l'elemento all'accumulatore, prima di passarlo all'iterazione successiva Quindi per mantenere l'ordine bisogna effettuare una concatenazione:

```
let take' n l =
    let aux acc n' = function
          [] -> acc
        | x::xs if n' > 0 then aux (acc@[x]) (n' - 1) xs
                else acc
    in aux [] n l
```

Ma la concatenazione è un'operazione molto costosa, quindi sarebbe meglio creare una lista inversa e poi invertirla alla fine. Si considera quindi una funzione `rev: 'a list → 'a list` che inverte l'ordine di una lista:

```
let rec rev = function
    | [] -> []
    | x::xs -> (rev xs)@[x]
```

Ma in modo ricorsivo utilizza comunque la concatenazione costosa, quindi si utilizza un'implementazione iterativa:

```
let rev l =
    let rec aux acc = function
        | [] -> acc
        | x::xs -> aux (x::acc) xs
    in aux [] l
```

Una funzione di inversione è comunque presente nel modulo List di OCaml, quindi si può utilizzare invece di definire una nuova funzione. L'approccio ricorsivo per la funzione `take` è quindi:

```
let take'' n l =
    let rec aux acc n' = function
        | [] -> acc
        | x::xs -> if n' > 0 then aux (x::acc) (n' - 1) xs
                   else acc
    in List.rev (aux [] n l)
```

A questo punto non è più di interesse il numero di occorrenze, quindi si definisce una funzione `primi: ('a * 'b) list` → `'a list`, per restituire una lista contenete solo il primo elemento delle coppie:

```
let rec primi = function
      [] -> []
    | (x, y)::l -> x::(primi l)
```

Si può quindi definire la funzione `super` come:

```
let super est n h = primi (take n (sort (contatutti (upto 1 h) (flatten est))))
```

4.2 Esempio: `merge sort`

Il merge sort è un algoritmo di ordinamento che applica la tecnica del *divide et impera*, consiste nel dividere un dato insieme da ordinare ricorsivamente ed ordinare i sottoinsiemi con lo stesso algoritmo di merge sort. Le due parti vengono divise in modo al più uguale. Per implementare il merge sort, bisogna definire una funzione che divida una lista in due metà, al massimo differenti di un elemento per lunghezza. Si vuole quindi definire la funzione `split: 'a list` → `'a list * 'a list`:

```
let rec split = function
      [] -> ([], [])
    | [x] -> ([x], [])
    | x::y::rest -> let (xs, ys) = split rest
                    in (x::xs, y::ys)
```

Le due parti della lista devono avere la stessa lunghezza, o al massimo variare di uno, ci sono due casi base, se la lista è vuota o ha un solo elemento. Se si hanno almeno due elementi si spacca a metà la lista, togliendo i primi due elementi `x` e `y` e si mettono in testa allo divisione successiva `xs ys`. Si può realizzare in modo più succinto, considerando solo due casi, una lista vuota o non vuota, e si alterna a quale lista viene aggiunto l'elemento successivo:

```
let rec split = function
    | [] -> ([], [])
    | x::xs -> let (ys, zs) = split xs
               in (x::zs, ys)
```

Per implementare l'algoritmo di merge sort, bisogna prima implementare la funzione merge, questa funzione deve unire due liste e mantenere il loro ordinamento. Si realizza la funzione `merge: 'a list` → `'a list` → `'a list`, considerando un ordinamento non decrescente:

```
let rec merge l1 l2 = match (l1, l2) with
    | ([], l) | (l, []) -> l
    | (x::xs, y::ys) -> if x < y then x::(merge xs (y::ys))
                        else y::(merge (x::xs) ys)
```

L'algoritmo di merge sort quindi si implementa semplicemente come:

```
let rec mergesort = function
    |  [] -> []
    | [x] -> [x]
    | l -> let (l1, l2) = split l
           in merge (mergesort l1) (mergesort l2)
```

5 Collezioni e Backtracking

Un tipo astratto di dato è costituito da un insieme di oggetti ed un insieme di operazioni che operano su questi oggetti. Per rappresentare un tipo astratto di dato bisogna almeno riferirsi ad un tipo di dato concreto. Quindi per rappresentare un tipo astratto di dato `A`, bisogna determinare un tipo concreto `T`, tale che ogni oggetto del tipo astratto di dato abbia almeno un rappresentante in `T`, ed elementi distinti del tipo `A` abbiano rappresentanti distinti in `T`. Bisogna implementare ogni operazione F su `A` con un'operazione P su `T`, in modo da conservare le operazioni. Ovvero se v_i sono valori di `T`, che rappresentano valori a_i di `A`, allora $P(a_1, \cdots, a_n)$ è un rappresentante di $F(v_1, \cdots, v_n)$. Quindi invece di operare direttamente oggetti di `A` si può operare sui loro rappresentanti in `T` ed in seguito si può determinare il corrispettivo in `A`.

Siano $\mathscr{A} = (A, f_1 \cdots, f_n)$ e $\mathscr{B} = (B, g_1, \cdots, g_n)$ due strutture algebriche dello stesso tipo, con ogni funzione f_i e g_i che prendano lo stesso numero di argomenti. Una rappresentazione di $\mathscr{A}$, tipo astratto, in $\mathscr{B}$, tipo concreto, è un'applicazione, funzione parziale: $\varphi : \mathscr{B} \to \mathscr{A}$, tale che è suriettiva: $\forall a \in A \, \exists b \in B \implies \varphi(b) = a$; ed è un omomorfismo: $\forall b_k \in B \implies \varphi(g_i(b_1, \cdots, b_n)) = f_i(a_1, \cdots, a_n)$.

5.1 Dizionari

Un tipo di astratto dato interessante sono dizionario, una collezione di elementi ciascuno costituito da una coppia chiave-valore. Ogni elemento ha chiave distinta. Su questa collezione si può cercare un elemento per chiave, inserire una coppia di elementi, e cancellare un elemento, data la chiave. Si possono implementare in OCaml tramite liste associative.

Una lista associativa è una lista di coppie, del tipo `('a * 'b) list`, sulle chiave deve esistere una relazione di uguaglianza. Quindi deve utilizzare un tipo che permette l'uguaglianza.

Data una lista associativa che implementa un dizionario, si vuole realizzare una funzione `assoc: 'a → ('a * 'b) list → 'b`, che data una chiave k ed una lista associativa l, restituisce, se esiste, il valore v corrispondente della chiave k nel dizionario l. Se non viene trovato l'elemento bisogna sollevare un'eccezione, dato che non può restituire alcun valore:

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

Questa dichiarazione non è corretta poiché nell'istruzione dichiarativa il tipo del costrutto `then` è il tipo di `v`, mentre nell'`else` il tipo restituito è una funzione e non coincide ad un valore. Nel modulo List è presenta una funzione `List.assoc` che effettua la stessa operazione, e solleva un'eccezione `Not_found` se non trova una chiave corrispondente. Per inserire un valore in un dizionario, bisogna prima eliminare la vecchia coppia che occupava quella chiave. Essendo troppo costoso per inserire una nuova coppia questa viene inserita in testa al dizionario, mente le vecchie coppie obsolete si trovano dopo questa nuova coppia. Quindi l'inserimento di una coppia chiave-valore k-v, viene quindi realizzato da una funzione `inserisci 'a → 'b → ('a * 'b) list → ('a * 'b) list`,

che inserisce il nuovo elemento in testa, in modo che venga scelto prima nella ricerca, e quindi "sovrascrive" il valore successivo diventato obsoleto:

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

Per cancellare una coppia chiave-valore dato che potrebbero essere presenti più coppie con la stessa chiave per l'implementazione precedente, bisogna eliminare tutte le coppie chiave-valore presenti nella lista, che coincidono alla chiave k fornita come argomento alla funzione `cancella: 'a` $\rightarrow$ `('a * 'b) list` $\rightarrow$ `('a * 'b) list`:

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

5.2 Insiemi

Un insieme finito è un tipo astratto di dato capace di contenere elementi, in questa implementazione dello stesso tipo, univoci all'interno dell'insieme. Ovvero in un'istanza di questo tipo astratto di dato può essere presente un unico elemento sulla base di una certa relazione di uguaglianza. Per gli insiemi sono definite le operazioni di ricerca $\in$, di unione $\cup$, intersezione $\cap$ e sottrazione tra insiemi $\setminus$. Il tipo concreto è una lista l che contiene esattamente gli elementi dell'insieme. Ogni lista di tipo `'a list` rappresenta un unico insieme. Si devono definire le quattro operazioni descritte sul tipo astratto:

- `mem: 'a` $\rightarrow$ `'a list` $\rightarrow$ `bool`: è un predicato che restituisce se un elemento x appartiene ad un insieme S, $x \in S$
- `union: 'a list` $\rightarrow$ `'a list` $\rightarrow$ `'a list`: applicata a due liste l_1 ed l_2 che rappresentano due insiemi S_1 e S_2 restituisce una rappresentazione di $S_1 \cup S_2$
- `intersect: 'a list` $\rightarrow$ `'a list` $\rightarrow$ `'a list`: applicata a due liste l_1 ed l_2 che rappresentano due insiemi S_1 e S_2 restituisce una rappresentazione di $S_1 \cap S_2$
- `setdiff: 'a list` $\rightarrow$ `'a list` $\rightarrow$ `'a list`: applicata a due liste l_1 ed l_2 che rappresentano due insiemi S_1 e S_2 restituisce una rappresentazione di $S_1 \setminus S_2$

Anche se una lista contenete ripetizioni è una rappresentazione contenente una sola istanza di quell'elemento, per evitare di creare liste eccessivamente lunghe si impone che le liste rappresentazioni di insiemi non abbiano ripetizioni.

Per implementare la ricerca, bisogna semplicemente iterare sulla lista e controllare se l'elemento corrente coincide a l'elemento x su cui viene applicata la funzione `mem`:

```
let rec mem x = function
    | [] -> false
    | y::ys -> y = x || (mem x ys)
```

La funzione di unione potrebbe semplicemente concatenare le due liste rappresentazioni, ma si vuole imporre la condizione che non possono esserci elementi ripetuti su di una stessa lista rappresentazione, quindi bisogna controllare la presenza prima di aggiungere un altro elemento:

```
let rec union l = function
    | [] -> l
    | x::xs -> if mem x l then union l xs
               else x::(union l xs)
```

In modo iterativo invece:

```
let union' l1 l2 =
    let rec aux l' = function
        | [] -> l'
        | x::xs -> if mem x l' then union l' xs
                   else union (x::l') xs
    in aux l1 l2
```

Poiché il caso base restituisce la prima lista, ed essenzialmente bisogna aggiungere ad ogni iterazione un elemento alla prima lista, o seconda in base all'implementazione, essenzialmente rappresenta un accumulatore, quindi non è necessario utilizzarne uno.

Per la funzione intersezione invece questo non può essere effettuato, per la sua versione iterativa; la funzione è essenzialmente la stessa, ma bisogna aggiungere un elemento solo se è presente in entrambe le liste:

```
let rec intersect l = function
    | [] -> []
    | x::xs -> if mem x l then x::(intersect l xs)
               else intersect l xs
```

Un algoritmo iterativo, invece necessita di un accumulatore:

```
let intersect' l1 l2 =
    let rec aux acc l' = function
        | [] -> acc
        | x::xs -> if mem x l' then aux (x::acc) l' xs
                   else aux acc l' xs
    in aux [] l1 l2
```

Si considera una possibile implementazione della differenza tra due insiemi:

```
let rec setdiff l1 l2 = match l1 with
    | [] -> []
    | x::xs -> if mem x l2 then setdiff xs l2
               else x::(setdiff xs l2)
```

In maniera iterativa:

```
let setdiff' l1 l2 =
    let rec aux acc l1' l2' = match l1' with
        | [] -> acc
        | x::xs -> if mem x l2' then aux acc xs l2'
                   else aux (x::acc) xs l2'
    in aux [] l1 l2
```

5.3 Backtracking

Il *backtracking* è una delle tecniche più importanti nella progettazione di algoritmi. L'idea è di costruire una soluzione in modo incrementale, in OCaml si può realizzare semplicemente. Questo algoritmo è simile ad una ricerca in ampiezza, ma è completa, poiché dopo aver trovato tutte le soluzioni scarta quelle peggiori e mantiene la soluzione migliore.

I candidati ad essere soluzioni sono una sequenza di elementi x_i appartenenti all'insieme delle possibili soluzioni S. L'approccio di "forza bruta" considera tutte le possibili combinazioni; l'algoritmo di backtracing costruisce una sequenza $x_1, \cdots, x_i$, scegliendo ad ogni passo un nuovo elemento x_{i+1} da aggiungere alla sequenza, ed analizza se questa sequenza ha possibilità di successo. Se è una soluzione mantiene la sequenza, altrimenti sceglie un altro elemento x_{i+1} da aggiungere alla sequenza. Una condizione principale è di non poter tornare indietro per elementi già visitati, e presenti nella sequenza. Questa condizione dipende dal problema su cui viene utilizzato l'algoritmo.

5.3.1 Somma di Sottoinsiemi

Un problema tipico dell'informatica è la somma di sottoinsiemi: dato un insieme di interi positivi S ed un intero n, determinare un sottoinsieme Y di S: $Y \subseteq S$, tale che la somma degli elementi di Y sia uguale al valore dell'intero n.

Si risolve mediante il backtracking, considerando come lo spazio di ricerca delle soluzioni un albero di tutti i possibili sottoinsiemi di S, che rispettano la condizione della somma. Le foglie quindi possono essere delle soluzioni valide oppure sottoinsiemi non validi, ed in caso non è necessario espanderli ulteriormente. Una volta esplorato un sotto-albero viene rimossa la sua radice e si continua la ricerca.

Ad ogni stadio della ricerca si considerano due insiemi degli elementi visitati o da visitare, $\hat{S}$ e $\hat{\hat{S}}$, uno complementare dell'altro. All'inizio si ha che $\hat{S} = \emptyset$ è lo spazio di ricerca delle soluzioni e $\hat{\hat{S}} = S$. Se la somma degli elementi di $\hat{S}$ è maggiore di n si ha una soluzione non valida, e si scarta l'ultimo elemento aggiunto, se è uguale ad n si ha identificato una soluzione. Se la somma è minore invece ed $\hat{\hat{S}} = \emptyset$ allora non è una soluzione valida, se non è vuoto allora si sceglie un elemento $x \in \hat{\hat{S}}$, e si cerca una soluzione aggiungendo x alla soluzione $\hat{S}$ con $\hat{S} \cup \{x\}$ e $\hat{\hat{S}} \setminus \{x\}$ oppure senza aggiungerlo alla soluzione con $\hat{S}$ e $\hat{\hat{S}} \setminus \{x\}$.

Per implementare un algoritmo risolutivo di questo problema si considera una funzione ausiliaria `sum` che somma tutti gli elementi di una lista:

```
let rec sum = function
    | [] -> 0
```

```
    | x::xs -> x + (sum xs)
```

In modalità iterativa:

```
let sum' l =
    let rec aux tot = function
          [] -> tot
        | x::xs -> aux (tot + x) xs
    in aux 0 l
```

Si definisce un'eccezione `NotFound` in caso la soluzione individuata non rappresenta una soluzione del problema:

```
exception NotFound
```

Si implementa ora l'algoritmo:

```
let search_subset set n =
    let rec search_aux sol others =
        let s = sum' sol
        in if s = n then sol
           else if s > n then raise NotFound
                else match others with
                    | [] -> raise NotFound
                    | x::xs -> try search_aux (x::sol) xs
                               with NotFound -> search_aux sol xs
    in search_aux [] set
```

Per creare una versione generale che restituisce tutte le possibili soluzioni, si creano delle funzioni ausiliarie:

```
let rec mapcons a = function
    | [] -> []
    | l::ls -> (a::l)::(mapcons a ls)
```

Questa funzione data una lista di liste, aggiunge in testa a tutte le liste contenute l'argomento α su cui viene applicata.

In questo modo si può definire la funzione `search_all` che cerca tutte le possibili soluzioni:

```
let rec search_all tot = function
    | [] -> if tot > 0 then []
            else [[]]
    | x::xs -> if x > tot then search_all tot xs
               else (mapcons x (search_all (tot - x) xs)) @ (search_all tot xs)
```

Questa funzione lavora cercando tutti i sottoinsiemi ricorsivamente creando un albero di sottoinsiemi, ogni volta che viene rimosso un elemento, ovvero scendendo da un ramo dell'albero, viene

diminuito il parametro `tot`, se è uguale a zero, allora la sequenza corrente corrisponde ad una soluzione, altrimenti bisogna effettuare backtracing. Ad ogni iterazione si rimuove l'elemento in testa e si determinano tutte le soluzioni nei due sottoinsiemi creati mantenendo e rimuovendo questo elemento in testa. Chiamandola ricorsivamente si scorre per intero l'albero dei sottoinsiemi possibili, ed arrivate alle foglie si scartano le soluzioni non valide e si concatenano tra di loro tutte le soluzioni individuate. Si utilizza la funzione `mapcons` per inserire l'elemento rimosso per ricreare il sottoinsieme soluzione.

5.3.2 Problema delle 8 Regine

Un problema comune chiamato problema delle 8 regine, consiste nell'individuare su una scacchiare una configurazione di otto regine, in modo che non siano mai sotto attacco. Poiché le regine possono attaccare su l'intera riga e colonna dove sono disposte, un modo per semplificare il problema, consiste nel posizionare le regine in tutte le righe e colonne, senza avere più regine sulla stessa riga e colonna, altrimenti sarebbero sicuramente sotto attacco. Il problema quindi consiste nel trovare una configurazione di otto regine, dove le diagonali non incontrano mai altre regine. Utilizzando il metodo della soluzione incrementale consiste nel piazzare una regina su una casella, e controllare se mettendo le restanti regine si trova una soluzione valida, altrimenti si ritorna a questa posizione iniziale e si sceglie un'altra casella.

Scomponendo il problema in sotto-problemi è sicuramente necessario avere una funzione che indica se una regina è sotto attacco. Questa funzione deve avere come parametri due posizioni, quindi due coppie di interi, e deve restituire un booleano se queste due posizione, contenessero delle regine, sarebbero sotto attacco l'una rispetto all'altra. Anche se non si tratta di uno scacco, è più intuitivo parlare di scacco, piuttosto di attacco, per indicare che la soluzione non è valida. Date due coppie di interi i, j e m, n se le due regine sono sulla stessa diagonale ascendente, allora la distanza attraversata in diagonale deve essere uguale, questa distanza si ottiene sommando le due coordinate. Mentre per determinare se sono sulla stessa diagonale discendente bisogna controllare che scendendo dalle coordinate i, j a $i - m$ e $j - n$ ci si trova nella stessa diagonale. Mentre sono sicuramente sotto attacco se la colonna o la riga è uguale.

```
let scacco ( i, j ) ( m, n ) =
    i = m || ( i - m = j - n) || ( i + j = m + n ) || j = n
```

Poiché su ogni colonna può esserci una sola colonna, una regina viene individuata in maniera univoca solamente dalla sua colonna. Per sapere su quale riga si trova, si considera una lista, dove l'indica su quale riga si trova, mentre il valore contenuto è variabile ed indica la colonna su cui si trova attualmente nella scacchiera. Quindi la riga delle regine è sempre fissata, mentre possono traslare sulle colonne:

```
let board = [1;2;3;4;5;6;7;8]
```

In questa rappresentazione, quindi non è necessaria la condizione per il controllo sulla riga:

```
let scacco (i, j) (m, n) =
    i = m || (i - m = j - n) || (i + j = m + n)
```

Per evitare di accedere alla lista per ottenere l'indice si può realizzare direttamente una lista di coppie. Per passare a questa rappresentazione si crea una funzione ausiliaria:

```
let combine l =
    let rec aux n acc = function
          [] -> acc
        | x::xs -> aux (n + 1) ((x,n)::acc) xs
    in List.rev(aux 1 [] l)
```

Si crea quindi una funzione `safe` che controlla data una configurazione della scacchiera, se una certa riga m è libera. Questa funzione prova ad aggiungere una regina alla riga m, se non sono presenti regine sulla scacchiera allora si può aggiungere senza problemi, altrimenti bisogna controllare se si può posizionare su una delle colonne della riga m, senza che sia sotto attacco. Se ciò non è possibile allora restituisce falso.

```
let safe board m =
    let n = List.length board
    in let rec aux = function
          [] -> true
        | (i, j)::xs -> not (scacco (i, j) (m, n + 1)) && aux xs
    in aux (combine board)
```

Il problema ora consiste nell'utilizzare questa funzione `safe` per trovare una soluzione al problema. Si definisce un'eccezione `NotFound` in caso non si è trovata una soluzione:

```
exception NotFound;;
let queens n =
    let rec aux sol i j =
        if j > n then sol
        else if i > n then raise NotFound
             else if safe sol i then
                      try aux (sol@[i]) 1 (j + 1)
                      with NotFound -> aux sol (i + 1) j
                  else aux sol (i + 1) j
    in aux [] 1 1
```

6 Funzioni di Ordine Superiore

Funzioni di ordine superiore sono funzioni che prendono come argomento o restituiscono una funzione, il tipo di una funzione di ordine superiore ha più di una freccia. La funzione `sum` è una funzione di ordine superiore:

```
let rec sum f lower upper =
    if lower > upper then 0
    else f lower + sum f (lower + 1) upper
```

Il tipo di `sum` è:

```
- sum: (int → int) → (int → (int → int))
```

Questa funzione simula il comportamento della sommatoria, accetta come argomento una funzione di cui eseguire la somma, dati i limiti superiore ed inferiore.

Funzioni di ordine superiore sulle liste sono la funzione di ordinamento e per iterare sui suoi elementi `List.sort` e `List.iter`. Altre funzioni importanti sono:

- `List.map:('a → 'b) → 'a list → 'b list`
- `List.for_all: ('a → bool) → 'a list → bool`
- `List.exists: ('a → bool) → 'a list → bool`
- `List.find: ('a → bool) → 'a list → 'a'`
- `List.filter: ('a → bool) → 'a list → 'a list`

La funzione `map`, data una lista che gli viene passata crea una mappa associando ciascun elemento della lista α ad un elemento β, applicando su ognuno di essi la funzione passata, creando quindi una lista β. La funzione `for_all` prende come parametro una lista, ed un predicato e restituisce un booleano se tutti gli elementi della lista soddisfano il predicato ricevuto in argomento. Analogamente `exists` restituisce un booleano se esiste almeno un elemento della lista che soddisfa il predicato passato come parametro. La funzione `find` trova e restituisce, se esiste, il primo elemento di una lista che soddisfa un predicato passato come argomento. Analogamente la funzione `filter` restituisce tutti gli elementi della lista che soddisfano il predicato, in pratica rimuove da una lista tutti gli elementi che non soddisfano una certa condizione. Mentre se nessun elemento della lista soddisfa il predicato, solleva un'eccezione.

Un'applicazione di `map` consiste in una funzione `inits` che restituisce tutti i segmenti iniziali di una lista passata come argomento:

```
inits [1;2;3;...;n] = [[1];[1;2];...[1;2;...;n]]
```

Si considera la sua implementazione:

```
let rec inits = function
    | [] -> []
    | [x] -> [[x]]
    | x::xs -> [x]::(List.map ((@)[x]) (inits xs) )
```

Si considera una funzione che produca l'insieme delle parti di un insieme, ovvero applicata ad un insieme S restituisce tutti i possibili sottoinsiemi di S. Per implementare questa funzione data una rappresentazione di tipo lista l dell'insieme S, si rimuove dalla lista il primo elemento x, si considera il sottoinsieme contenente solo questo elemento $[x]$, e tutti gli altri sottoinsiemi, richiamando ricorsivamente la funzione, concatenati a questo elemento x rimosso. Non si può utilizzare l'operatore `cons` infisso, quindi si utilizza l'operatore di concatenazione infisso:

```
let rec powerset = function
    | [] -> [[]]
    | x::xs -> let ps = powerset xs
               in ps@(List.map ((@) [x]) ps)
```

Si determina una funzione che realizzi il prodotto cartesiano `cartprod: 'a list → 'b list → ('a * 'b) list`, applicata a due insiemi `setA` e `setB` riporta la lista di tutte le coppie (x, y) con $x \in$`setA` e $y \in$`setB`. Si definisce la funzione ausiliaria `pair: 'a → 'b → ('a * 'b)` per poter utilizzare `map`:

```
let pair x y = (x, y);;
let rec cartprod l1 l2 = match l1 with
    | [] -> []
    | x::xs -> (List.map (pair x) l2)@(cartprod xs l2)
```

Si considerano le funzioni per rappresentare e decodificare un carattere in codice morse e viceversa, utilizzando i due caratteri punto `.` e linea `-`. Data una lista di codici morse, si può usare la funzione `map`, applicata sulla funzione di decodifica per restituire una lista di caratteri. Bisogna comunque passare da una lista di caratteri ad una stringa, ed in OCaml, questi due tipi non sono coincidenti quindi sono necessarie due funzioni `implode char list → string` e `implode: string → char list`. La funzione `implode` si può realizzare semplicemente come:

```
let rec implode = function
    | [] -> []
    | x::xs -> (String.make 1 x)^(implode xs)
```

Mentre per la funzione `explode` bisogna iterare su ogni elemento della stringa, e bisogna considerare l'ordine in cui verranno aggiunti gli elementi letti:

```
let explode s =
    let rec aux acc i x =
        if i < 0 then acc
        else aux (s.[i]::acc) (i - 1) s
    in aux [] ((String.length s) - 1) s
```

Se si itera partendo dall'indice iniziale, allora bisogna invertire la stringa passata alla funzione, altrimenti restituirebbe la lista corrispondente invertita.

Altri funzioni utili nel modulo List sono:

- `List.rev_map: ('a → 'b) → 'a list → 'b list`, definita come la funzione `map` invertita, oppure la funzione `map` applicata sulla stessa lista invertita.

- `List.find: ('a → bool) → 'a list → 'a`, restituire il primo elemento della lista che soddisfa il predicato.
- `List.partition: ('a → bool) → 'a list → 'a list * 'a list` divide la lista in un prodotto di liste, dove la prima soddisfa il predicato e la seconda non lo soddisfa.
- `List.fold_left: ('a → 'b → 'a) → 'a → 'b list → 'a`
- `List.fold_left: ('a → 'b → 'a) → 'b list → 'a → 'a`

La funzione `rev_map` viene definita come:

```
List.rev_map f lst = List.rev (List.map f lst)
```

Tra le più importanti di queste sono `fold_left` e `fold_right`. Queste funzioni permettono di attraversare una lista, sostituendo al costruttore della lista `[]`, una certa funzione su cui viene applicata, fornendo il caso base a la lista su cui viene applicata. Questi due argomenti sono invertiti tra le due funzioni. Le due funzioni attraversano la lista in due direzioni opposte, `fold_right` parte dall'inizio, mentre `fold_left` parte dalla fine. Quando l'operatore di ricorsione è commutativo, utilizzare le due funzioni è equivalente.

La funzione `sumof` che somma tutti gli elementi di una lista si può quindi rappresentare come:

```
let sumof l = List.fold_right (+) l 0;;
let sumof l = List.fold_left (+) 0 l;;
```

7 Definizioni di Nuovi Tipi

Un tipo è un insieme di valori, per poter definire un nuovo tipo, è necessario un nome per il tipo e come costruire i valori del tipo, ovvero i loro costruttori.

7.1 Tipi Enumerati

Si considerano i tipi enumerati, tipi costruiti da un numero finito di valori, il tipo `bool` anch'esso un tipo enumerato, con solamente due possibili valori. I valori di tipi enumerati sono identificati da costanti, sono un tipo particolare di costruttori, che non richiedono alcun argomento.

Per definire un nuovo tipo bisogna usare la parola chiave `type`, seguita da un'assegnazione di valori del tipo separati da `|`, con la prima lettera maiuscola. Questi nuovi valori e tipi possono essere utilizzati per la creazione di nuovi valori e di nuovi tipi.

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

Quando viene definito un tipo enumerato, si ha l'uguaglianza, quindi è possibile utilizzare operazioni di confronto su questi tipi costruiti senza doverle definire, come `List.mem`. Per recuperare il valore numerico o non di uno di questi tipi si può introdurre una funzione con il pattern matching, associando ogni valore del tipo costruito ad un valore di tipo primitivo.

Si vuole rappresentare la posizione ed il movimento di un oggetto su un piano bidimensionale, in una direzione oppure al cambiamento della direzione, bisogna utilizzare il nuovo tipo definito "direzione". La posizione di un oggetto nel piano è definita da un punto ed una direzione, quindi si considera il nuovo tipo:

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

Questo non rappresenta un nuovo tipo, ma un'abbreviazione di tipo, ogni volta che si vuole utilizzare questa collezione di tipi già definiti. Si sta assegnando un altro nome ad un tipo già esistente. Si definiscono i selettori per estrarre le componenti della posizione:

```
let pos_x (x,_,_) = x
let pos_y (_,y,_) = y
let pos_dir (_,_,dir) = dir
```

Si vogliono rappresentare due tipi di azioni, girare di 90° in senso orario, oppure andare avanti di n passi, con n numero intero. Si vuole definire un tipo di dati azione per rappresentare queste azioni. Sembra che sia necessario un numero infinito di valori poiché sono presenti un numero infinito di interi, ma se si vuole realizzare un tipo enumerato, i valori devono essere finiti. Si utilizzano quindi i costruttori funzionali, che permette di rappresentare una funzione, il costruttore `Avanti` restituisce il tipo azione se viene applicato su un tipo intero:

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

Si utilizza la parola chiave `of` per definire il costruttore funzionale. Mediante il pattern matching è possibile definire selettori di dei tipi:

```
let int_of_act = function
    | Avanti n -> n
    | _ -> failwith "int_of_act"
```

Se si fosse definito il tipo posizione introducendo un nuovo tipo, con il suo costruttore, bisognerebbe modificar ele funzioni precedenti per i selettori della tripla, per essere selettori del tipo:

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

Si possono definire i selettori di posizione relativi a questo nuovo tipo:

```
let xcoord (Post(x,_,_)) = x
let ycoord (Post(_,y,_)) = y
let dir (Pos(_,_,d)) = d
let punto_di (Pos(x,y,_)) = (x,y)
```

Bisogna utilizzare le parentesi altrimenti nell'uso delle funzioni, OCaml non interpreta il costruttore come applicato sugli argomenti, dato che associa a sinistra. In questo caso il nuovo tipo `posizione` è diverso dal tipo definito precedentemente, poiché in questo caso non rappresenta un'abbreviazione di un tipo già definito, ma è un tipo a sé, avendo un unico valore `Pos` che prende la tripla, precedentemente individuata dal "tipo" `posizione`

La funzione di spostamento `sposta` prende due argomenti, la posizione corrente e l'azione da effettuare, quello che effettua questa funzione dipende dal valore dell'azione, si dovrà calcolare i due possibili modi in cui questa funzione può comportarsi.

La funzione gira, associata all'azione di valore `Gira` è di tipo `direzione` → `direzione`:

```
let gira = function
    | Su -> Destra
    | Giù -> Sinistra
    | Destra -> Giù
    | Sinistra -> Su
```

Mentre la funzione avanti associata al valore `Avanti` del tipo —azione— è di tipo `posizione` → `int` → `posizione`:

```
let avanti (x,y,d) n = match d with
    | Su -> (x,y+n,d)
    | Giù -> (x,y-n,d)
    | Destra -> (x+n,y,d)
    | Sinistra -> (x-n,y,d)
```

La funzione `sposta` quindi effettua un pattern matching sui due possibili valori di azione e chiama la funzione corrisponde:

```
let sposta (x,y,d) = function
    | Gira -> (x,y,gira d)
    | Avanti n -> avanti (x,y,d) n
```

Data la funzione sposta, definire la funzione esegui che data una posizione `pos`, ed una lista di azioni `la` restituisce la posizione corrispondente dopo aver eseguito le azioni nell'ordine in cui sono fornite:

```
let rec esegui pos la = List.fold_left sposta pos la
```

7.2 Unione di Tipi

Si possono usare i tipi enumerati per effettuare unioni di tipi, si considera un tipo fintamente enumerato, che rappresenta l'unione del tipo intero e reale:

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

Questa rappresenta l'unione disgiunta di questi due tipi, i costruttori marcano gli elementi, rendendo riconoscibile la loro provenienza. In seguito si possono definire operazioni su questo tipo, che permettono di applicare operazioni aritmetiche su tipi disgiunti:

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

Questo nuovo tipo rappresenta l'unione disgiunta dei tipi che sono argomenti dei costruttori di questo tipo enumerato. Dati due tipi A e B, la loro unione disgiunta è data da:

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

Oltre ad espandere tipi si possono restringere. Si vuole creare un tipo che contenga solamente i numeri naturali. Si considerano gli assiomi di Peano per definire i numeri naturali induttivamente:

- $0 \in \mathbb{N}$
- $\forall n \in \mathbb{N} \to \rho(n) \in \mathbb{N}$, dove ρ è una qualche funzione successore, definita su $\mathbb{N}$
- Gli unici elementi di $\mathbb{N}$ sono quelli ottenibili applicando, ricorsivamente, i primi due assiomi.

Si può definire ricorsivamente il tipo dei numeri naturali come:

```
let nat = Zero | Succ of nat
```

Dove tutti i valori di `nat` sono o `Zero` o applicazioni ricorsive di `Succ`. Su questi tipi induttivi si possono definire operazioni ricorsivamente, definendo una funzione `succ: int` $\to$ `int`, che incrementa un qualsiasi intero su cui viene applicata:

```
let rec int_of_nat = function
    | Zero -> 0
    | Succ n -> succ(int_of_nat n)
```

Si ricorda che n a cui è applicata `Succ` nel secondo match è di tipo `nat`, quindi non si può applicare direttamente `succ n`. Si possono definire operazioni aritmetiche sui numeri naturali, come è stato realizzato per l'unione degli interi e dei reali:

```
let rec sum (n, m) = match n with
    | Zero -> m
    | Succ k -> Succ(sum(k, m))
```

7.3 Costruttori di Tipo

La definizione di un tipo può essere parametrica con variabili di tipo o con costruttori polimorfi. Si vuole realizzare un tipo matrice polimorfo, utilizzando lo stereotipo `'a` prima del nome del tipo, si utilizza nella dichiarazione per renderlo polimorfo:

```
let 'a matrix = Mat of 'a list list
```

`Mat` è un costruttore di tipi, può creare matrici di qualsiasi tipo. Analogamente si possono creare liste in maniera polimorfa, definendo il costruttore `Cons`:

```
let 'a alist = Nil | Cons of 'a alist
```

Il tipo `option` è un'alternativa all'uso delle eccezioni, permette di realizzare funzioni di tipi parziali, e si può usare al posto delle eccezioni per la propagazione degli errori. Un tipo `option` può avere valore `None`, oltre ad i normali valori di un tipo definito:

```
let 'a option = None | Some of 'a
```

Si considerano anche dei selettori per il tipo optional:

```
let valore = function
    | Some n -> n
    | None -> failwith "There is no value"
```

L'esecuzione non si interrompe se viene restituito un valore `None`, dato che è accettato dal tipo `'a option`, se si volesse implementare una divisione si potrebbe utilizzare questo tipo come:

```
let div x y =
    if y = 0 then None
    else Some (x /. y)
```

Si può utilizzare per re-implementare le liste associative, nel modulo List la funzione `assoc` restituisce un'eccezione se la chiave non è presente nel dizionario, mentre utilizzando il tipo `option` si può restituire `None` invece di bloccare l'esecuzione:

```
let rec assoc l x = match l with
    | [] -> None
    | (k,v)::rest -> if x = k then Some v
                     else assoc x rest
```

In questo modo si possono creare funzioni che operano sull'intera lista associativa, senza terminare l'esecuzione in caso di un'eccezione. Dopo aver ottenuto il risultato, questo si può filtrare dato che il valore restituito `None` è valido e confrontabile. Si considera una funzione che restituisce una lista di valori v corrispondenti ad una lista di chiavi k, applicata ad una lista associativa l:

```
let assoc_all l k = List.map valore (List.filter ((<>) None) (List.map (function x -> assoc x l)
```

8 Alberi

8.1 Espressioni Aritmetiche

Si considerano espressioni costituite dalle applicazioni delle quattro operazioni aritmetiche su variabili o costanti intere. Si può definire induttivamente l'insieme delle espressioni ottenibili solamente seguendo questi tre principi:

- Se $n \in \mathbb{N}$ allora n è un'espressione
- Se x è una variabile, allora x è un'espressione
- Se E_1 e E_2 sono due espressioni, allora anche le applicazioni delle operazioni aritmetiche intere su queste due sono a loro volta delle espressioni

Questo insieme contiene definito contiene un numero infinito di oggetti di base, sono presenti quattro costruttori funzionali, somma, differenza, divisione e moltiplicazione. Forniscono metodi per realizzare oggetti complessi da oggetti più semplici. Questi costruttori funzionali si applicano su due oggetti più semplici, per crearne uno più complesso.

Essendo i costruttori funzionali applicati a più di due oggetti, si possono rappresentare tramite alberi, dove i nodi sono le operazioni applicate ai suoi due figli, e le foglie contengono gli oggetti più semplici, ovvero costanti o variabili intere. Risalendo l'albero fino alla radice, si possono calcolare queste espressioni per ottenere espressioni più complesse. Si può definire un tipo di dati induttivo, tenendo presente questi principi:

```
type expr =
    | Int of int
    | Var of string
    | Sum of expr * expr
    | Diff of expr * expr
    | Mult of expr * expr
    | Div of expr * expr
```

Le variabili sono rappresentate come stringhe, ma per ottenerne il valore bisogna definire un ambiente dove è presente. Si definisce allora il tipo ambiente:

```
type ambiente = (string * int) list
```

Si vuole allora definire una funzione `eval` che applicata ad un ambiente `env` permette di valutare un'espressione e, restituendo l'intero corrispondente. Questa funzione produce un errore se nell'ambiente non sono definiti i valori di alcune delle variabili. Essendo l'ambiente realizzato da una lista si può utilizzare la funzione `assoc` per determinare il valore delle variabili. Per le restanti operazioni è sufficiente applicare `eval` sulle due espressioni su cui è applicata l'operazione:

```
let rec eval env = function
    | Int n -> n
    | Var x -> List.assoc x env
```

```
    | Sum(e1, e2) -> eval env e1 + (eval env e2)
    | Diff(e1, e2) -> eval env e1 - (eval env e2)
    | Mult(e1, e2) -> eval env e1 * (eval env e2)
    | Div(e1, e2) -> eval env e1 / (eval env e2)
```

8.2 Alberi Binari

Un albero è un insieme di oggetti, detti nodi, dove per ogni nodo n è definita una relazione binaria $G(n, m)$ su un altro nodo m, dove m è detto genitore e n è detto figlio, tale che esiste un unico nodo radice senza genitore; ogni nodo che non sia la radice ha uno ed un solo genitore; per ogni nodo diverso dalla radice n esiste un cammino dalla radice ad n, l'albero si dice quindi connesso. Dati un qualsiasi nodo dell'albero n; esistono k nodi n_i, tali che n_1 è la radice e n_k è il nodo stesso n, inoltre per ogni nodo n_i, il nodo n_{i+1} è suo genitore. Dato un cammino tra due nodi n ed m, n viene chiamato antenato e m è detto discendente di m. Dato un cammino, il numero di nodi della sequenza viene detta lunghezza del cammino. Un nodo può essere una foglia, se non ha figli, oppure interno se ha almeno un figlio, nodi che hanno lo stesso genitore si chiamano fratelli. La profondità di un nodo è la lunghezza del cammino dal nodo alla radice, mentre l'altezza di un nodo è la lunghezza del cammino dal nodo ad una foglia. L'altezza dell'albero corrisponde alla profondità massima di un nodo dell'albero. La dimensione di un albero è il numero di nodi. Un sottoalbero è l'insieme costituito da un nodo n e tutti i suoi discendenti, dove n è la radice del sottoalbero.

Un albero binario è un albero dove ogni nodo ha al massimo due figli. Si possono definire induttivamente considerando i seguenti principi:

- Una foglia n è un albero binario, con radice sé stesso
- Dato un albero binario T_0, con radice n_0, aggiungendo un nuovo nodo n come genitore di n_0, l'albero T risultante è un albero binario
- Dati due alberi binari T_0 e T_1 con radice rispettivamente in n_0 e n_1, aggiungendo un nuovo nodo n come genitore di n_0 e n_1 si ottiene un nuovo albero binario T, con radice n
- Nient'altro è un albero binario

Il tipo di un albero binario dipende dal tipo dei nodi, quindi per definirlo si utilizza il termine `'a`, si potrebbe realizzare il nuovo tipo `tree` come:

```
type 'a tree =
    | Leaf of 'a
    | One of 'a * 'a tree
    | Two of 'a * 'a tree * 'a tree
```

La dimensione di un albero si potrebbe calcolare allora come:

```
let rec size = function
    | Leaf _ -> 1
    | One(_,t) -> 1 + size t
    | Two(_,t1,t2) -> 1 + size t1 +  size t2
```

Ma questa definizione non è compatta, includendo nell'insieme di alberi binari, l'insieme vuoto `Empty` si può definire un albero binario come avente sempre due sottoalberi, che possono essere eventualmente vuoti, ed in quel caso sarebbe una foglia:

```
type 'a tree = Empty | Tr of 'a * 'a tree * 'a tree
```

Si definiscono alcune funzioni di base e selettori per questo nuovo tipo:

```
let is_empty = function
    | Empty -> true
    | _ -> false
;;
exception EmptyTree
let root = function
    | Empty -> raise EmptyTree
    | Tr(x,_,_) -> x
;;
let is_leaf = function
    | Tr(x, Empty, Empty) -> true
    | _ -> false
;;
let leaf x = Tr(x, Empty, Empty)
let left = function
    | Empty -> raise EmptyTree
    | Tr(_,t,_) -> t
;;
let right = function
    | Empty -> raise EmptyTree
    | Tr(_,_,t) -> t
;;
```

Si può definire la funzione `size`, come precedentemente con il pattern matching, oppure usando le funzioni appena create:

```
let rec size t =
    if is_empty t then 0
    else 1 + size (left t) + size (right t)
```

Si vuole realizzare una funzione `count: 'a tree` $\rightarrow$ `('a * int) list` che restituisca una lista di coppie, contenente per ogni etichetta, unica, x di un nodo dell'albero, il suo numero di occorrenze all'interno dell'albero stesso. Nel caso base di un albero vuoto, questo restituisce una lista vuota `[]`, mentre se l'albero non è vuoto viene applicata la funzione su entrambi i figli ed ottenute le due sottoliste si fondono insieme, aggiungendo anche la radice. Questo algoritmo tuttavia non è molto efficiente, quindi si utilizza il principio dell'accumulatore come se fosse un approccio iterativo, si utilizza il risultato parziale che rappresenta il risultato dei nodi già visitati. Se un etichetta a è già presente nel risultato parziale, allora si incrementa il suo contatore, mentre se non è esistente allora

si aggiunge la coppia `(a,1)`. Quindi si parte da una lista vuota `[]` e si visita l'albero in pre-ordine. Si considera una funzione ausiliaria per aggiungere o incrementare la lista risultato:

```
let rec add x = function
    | [] -> [(x,1)]
    | (y,n)::ys -> if x = y then (y,n+1)::ys
                   else (y,n)::(add x ys)
```

Si definisce quindi la funzione `count` come:

```
let rec count t =
    let rec aux acc = function
        | Empty -> acc
        | Tr(x, l, r) -> aux (aux (add x acc) l) r
    in aux [] t
```

Si possono utilizzare alberi binari per cifrare e decifrare codici composti da due elementi, che siano numeri binari o codici morse. Nella scorsa implementazione del codice morse, si utilizzava un patter match tra tutti possibili codici morse per rappresentare le lettere dell'alfabeto. Questo può essere realizzato tramite un albero binario, dove andare a destra o sinistra indica l'aggiunta di un punto o una linea al codice risultante. In questo modo tenendo traccia del cammino effettuato si può decifrare qualsiasi sequenza morse. Scendendo a sinistra, si aggiunge un punto, a destra una linea. Supponendo di avere a disposizione un tale albero `alphabet`, si vuole definire una funzione `morse` che dato un carattere restituisce il codice morse cifrata correttamente:

```
let morse c =
    let rec aux code = function
        | Empty -> raise NotFound
        | Tr(c', l, r) -> if c' = c then code
                          else try aux (code^".") l
                               with _ -> aux (code^"-") r
    in aux "" alphabet
```

9 Alberi n-ari

Mentre gli alberi binari hanno sempre uno o due figli, gli alberi n-ari hanno un numero finito, arbitrario di figli. Dove ogni nodo dell'albero ha come figli una lista di alberi n-ari. Si possono definire induttivamente dai seguenti principi:

- Un albero costituito da un nodo r senza genitore è un albero, con radice il nodo stesso
- Dati n alberi t_i, con radice r_i, dato un nuovo nodo r, allora la struttura ottenuta aggiungendo questo nodo come genitore di tutte le radici r_i è un albero t
- Nient'altro è un albero

Il tipo albero n-ario si può definire come:

```
type 'a ntree = Tr of 'a * 'a ntree list
```

Una foglia etichettata da n è quindi rappresentata da `Tr(n,[])`.

```
let leaf x = Tr(x, [])
```

9.1 Alberi di Min-Max

Gli alberi di min-max sono degli alberi dove ad ogni foglia è assegnato un valore di utilità, il giocatore Max vuole massimizzare questo valore, mentre il giocatore Min cerca di minimizzarlo. I valori vengono assegnati solamente alle foglie, e si propagano verso l'alto, in base all'etichetta del genitore, che può essere appunto Max o Min. Se l'etichetta è Max, allora prende come valore il massimo tra i suoi figli, se è Min, il minimo. Questa propagazione si estende fino alla radice dove viene registrato il valore corrispondente.

Questi alberi sono etichettati da una coppia costituita da un tipo di giocatore, Min o Max, e da un intero, il valore numerico del nodo.

```
type player = Min | Max
type minmaxtree = Leaf of int | Node of (player * int) * minmaxtree list
```

Per ogni nodo intermedio etichettato dalla coppia Max e n, quest'ultimo rappresenta il massimo valore numerico dei figli del nodo, analogamente se viene etichettato da Min e m questo rappresenta il minimo valore numerico dei figli del nodo. Questi valori sono be assegnati, rappresentano sempre il massimo o il minimo dei valori dei figli. Dato un albero Min-Max male assegnato scrivere una funzione `propagate` che restituisce il corrispondente albero ben assegnato, mantenendo invariato il valore delle foglie. Dato un sottoalbero T, questo assegna i valori agli alberi Min-Max figli, e confronta il loro valore per definire il valore del'etichetta della radice dell'albero T. Questo è un processo ricorsivo, il caso base è una foglia, dove il valore è quello etichettato dalla foglia. Si definiscono delle funzioni ausiliarie:

```
let calcola p n m = match p with
    | Min -> min n m
```

```
    | Max -> max n m
let rec mossa p = function
    | [] -> raise Not_found
    | x::xs -> match x with
        | Leaf n -> (try calcola p n (mossa p xs)
                     with _ -> n)
        | Node((_,v),_) -> (try calcola p v (mossa p xs)
                            with _ -> v)
```

Si definisce quindi la funzione `propagate` come:

```
let rec propagate = function
    | Leaf n -> Leaf n
    | Node((p,_),l) -> let lt = List.map propagate l
                       in Node((p, mossa p lt), lt)
```

10 Grafi

I grafi possono essere considerati una generalizzazione degli alberi. Una delle caratteristiche degli alberi è di avere una sola radice, ed è possibile che esista un unico cammino che connetta due nodi. Nei grafi non è presente il concetto di radice, e sono possibili percorsi attraverso il grafo per raggiungere un certo nodo n, partendo da un altro nodo m. I nodi possono essere orientati, ovvero un arco che connette due nodi può essere caratterizzato da una direzione di navigabilità permessa.

Per realizzare un grafo in OCaml, bisogna definire il tipo grafo, una lista associativa formata da coppie di chiavi-valore, dove la chiave è il nodo dell'albero ed il valore è la lista dei nodi direttamente accessibili a quel nodo.

```
type 'a graph = Node of 'a * 'a graph list
```

Si può definire una funzione successore che restituisce i nodi successori di un nodo dato x, considerando la funzione `List.assoc`, poiché si tratta di una lista associativa:

```
exception NodeNotFound
let successore x l =
    try List.assoc x l
    with Not_found -> raise NodeNotFound
```

Questa non è l'unica rappresentazione possibile dei grafi, si potrebbe definire il tipo grafo con un altro tipo, memorizzando gli archi orientati, come coppie di nodi, dove il primo nodo è il nodo uscente, ed il secondo è il nodo entrante:

```
type 'a graph = ('a * 'a) list
```

Questa è una rappresentazione di un grafo orientato, per un grafo non orientato, la posizione dei nodi all'interno dell'arco è indifferente, gli archi (n, m) e (m, n) sono quindi equivalenti su grafi non orientati. Quindi un grafo non orientato ha la stessa rappresentazione, solamente bisogna implementare le funzioni opportunamente per considerare gli archi orientati o meno, in modo che sia conforme alle specifiche richieste. Altrimenti si possono considerare tutti gli archi ed i loro inversi, poiché possono essere attraversati sia in un verso che nell'altro.

Utilizzando questa rappresentazione per trovare i successori bisogna cercare nella lista:

```
let rec successori x = function
    | [] -> []
    | (y,z)::xs -> if x = y then z::(successori x xs)
                   else successori x xs
```

Più semplicemente si può realizzare utilizzando la funzione `filter` del modulo `List`:

```
let successori x g = List.map snd (List.filter (function (y,_) -> x = y) g)
```

Invece per determinare tutti i nodi vicini ad un nodo dato x in un grafo g, si può iterare come coni successori, e controllare anche il secondo componente della coppia. Per renderla più semplice si riutilizza la funzione `filter`:

```
let successori x g = List.map (function (y,z) -> if y = x then z else y) (List.filter (function
```

Analizzando gli alberi si è visto che esistono diversi algoritmi per visitare gli alberi, allo stesso modo sono possibili diversi modi di visitare un grafo. Potrebbero essere presenti dei cicli, e quindi il processo di visita potrebbe rimanere bloccato su uno di questi cicli. Il problema viene risolto tenendo traccia dei nodi già visitati e da visitare. I metodi più comuni sono DFS, *Depth First Search*, o visita in profondità, dove dato un nodo x, si analizza questo, prima di analizzare i successori. Mentre il contrario è la visita in ampiezza BFS, *Breath First Search*, dove prima di analizzare il nodo corrente x, si analizzano i successivi. Per implementare questi due metodi, è semplicemente utilizzare funzioni di aggiunta diverse alla collezione che contiene i nodi da visitare chiamate dei nodi pendenti, in DFS questa viene gestita come una pila, con filosofia LIFO, *Last In First Out*, ovvero il primo elemento ad essere aggiunto, sarà il primo ad essere rimosso, e quindi analizzato. Si può realizzare come una lista, dove si applica aggiunta e rimozione in testa. La visita in ampiezza BFS invece realizzare questa collezione come una coda, seguendo la filosofia *FIFO*, *Firs In First Out*, dove il primo elemento ad essere aggiunta è l'ultimo elemento ad essere rimosso.

Si considera una possibile implementazione della visita DFS, utilizzando una funzione ausiliaria che tiene traccia dei nodi visitati, e prende come parametro la lista dei nodi pendenti:

```
let depth_first_search g s =
    let rec search visited = function
        | [] -> visited
        | x::xs -> if (List.mem x visited) then search visited xs
                   else search (x::visited) ((successori x g)@xs)
    in search [] [s]
```

Nella visita in ampiezza, l'unica differenza è che i successori del nodo corrente vengono inseriti in coda, e non più in testa:

```
let breath_first_search g s =
    let rec search visited = function
        | [] -> visited
        | x::xs -> if (List.mem x visited) then search visited xs
                   else search (x::visited) (xs::(successori x g))
    in search [] [s]
```

Scrivere una funzione `test_connessi` che dato un grafo orientato g, e due nodi n e m, determini se esiste un cammino dal nodo n al nodo m, restituendo un booleano. Questa ricerca può essere effettuata sia con una visita in ampiezza che in profondità. Si considera il primo nodo come il nodo di partenza, e se viene raggiunto il nodo di arrivo allora restituisce `true`, mentre il caso base è `false`:

```
let test_connessi g n m =
    let rec aux visited = function
        | [] -> false
        | x::xs -> if x = m then true
                   else if (List.mem x visited) then aux visited xs
```

```
                        else aux (x::visited) ((successori x g)@xs)
    in aux [] [n]
```

La differenza tra i due approcci consiste nel visitare prima o meno i successori del nodo corrente. Entrambe le visite hanno la stessa complessità e sono entrambi algoritmi completi, quindi non hanno una differenza nelle prestazioni.

Per controllare se in un grafo è presente un ciclo, si può modificare la funzione precedente, modificando opportunamente il primo passo:

```
let test_ciclo g n =
    let rec aux visited = function
        | [] -> false
        | x::xs -> if x = n then true
                   else if (List.mem x visited) then aux visited xs
                        else aux (x::visited) ((successori x g)@xs)
    in aux [] (List.filter (function x -> x <> n) (successori n g))
```

Si consideri ora una funzione che restituisca un ciclo, ovvero un cammino che parte e finisce con n, se esiste, partendo dal nodo n, in un grafo orientato g. Non si considera un ciclo se un arco esce ed entra nello stesso nodo.

```
let ciclo g n =
    let rec from_node visited x =
        if List.mem x visited
        then raise NotFound
        else if x = n then [x]
        else x::from_list (x::visited) (successori x g)
    and from_list visited = function
        | [] -> raise NotFound
        | x::xs ->
            try from_node visited x
            with NotFound -> from_list (x::visited) xs
    in n::from_list [] (successori x g)
```

Dato un grafo g ed un nodo di partenza n, determinare se dal nodo n è raggiungibile un nodo che soddisfa un certo predicato p, si tratta di una variante di `test_connessi`. Se è raggiungibile un nodo he soddisfa questo predicato, allora viene restituito, altrimenti viene restituita un'eccezione. È sufficiente modificare leggermente l'implementazione delle funzioni precedenti:

```
exception NotFound
let search_nodo g n p =
    let rec aux visited = function
        | [] -> raise NotFound
        | x::xs -> if (p x) then x
                   else if (List.mem x visited) then aux visited xs
                        else aux (x::visited) ((successori x g)@xs)
    in search [] [n]
```