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 PF01: 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 PF02: 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.

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.

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.

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`.

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 PF03: 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` $\rightarrow$ `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 PF04: Liste

4.1 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.

I selettori di una lista sono `List.hd: 'a list` $\to$ `'a`, *head*, che restituisce il primo elemento e `List.tl: 'a list` $\to$ `'a list`, *tail*, restituisce la lista corrispondente senza il primo elemento. Non sono definiti sulla lista vuota `[]`, quindi applicati su di essa viene sollevata un'eccezione.

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`.

Si vuole determinare una funzione che determina il numero degli elementi in una lista `length: 'a list` $\to$ `int`. 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
    | x::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.

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)
```

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)
```

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 x = y then 1 + (conta x ys)
               else conta x ys
```

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)
```

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))))

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

```
let upto' m n =
    let rec aux l m' n' =
        if m > n then l
        else aux (n'::l) m' (n' - 1)
    in aux [] m n
```

L'ultima riga potrebbe essere scritta come `else aux (m'::l) (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 superiore nel primo caso e dall'estremo inferiore nel secondo caso.

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

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

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

La funzione `flatten` appiattisce una lista di liste:

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

La funzione `conta` determina le occorrenze di un elemento in una lista:

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

In forma iterativa diventa:

```
let conta v l =
    let rec aux t l' a =
        match l with
              [] -> a
            | x::xs -> if x = t then aux t xs (a + 1)
                       else aux t xs acc
    in aux v l 0
```

La funzione `contatutti: 'a list -> 'a list -> ('a * int) list` conta tutte le occorrenze di ogni elemento nella lista `flatten`:

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

Per ordinarle in ordine crescente in cui sono state estratte, bisogna determinare una funzione di ordinamento per la funzione `sort` del modulo `List` che permette di restituire una lista di coppie in output, come in input. Si definisce una funzione ausiliaria `comp`:

```
let comp (v1, n1) = function
    (_, n) -> if n1 < n then -1
              else if n1 = n then 0
              else 1
```

Questa funzione permette di comparare due coppie, quindi si può usare per ordinare una lista formata da coppie:

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

let rec primi = function
      [] -> []
    | x::xs -> (fst x)::(primi xs)

let super estrazioni dim higher =
    print(' (take' dim (sort (contatuttti (upto 1 higher) (flatten0 estrazioni)))))
```

Esempio: `merge sort`

```
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 sollo 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:

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

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

Nel modulo list sono definite le funzioni head `hd`, tail `tl` già descritte in precedenza, `length`, `flatten` che trasforma una lista di liste in un'unica grande lista, la funzione `sort` che ordina una lista, secondo un qualsiasi ordine definito da una funzione fornita come argomento.

5 PF05: Collezioni e Backtracking

5.1 Dizionari

Un tipo di dato interessante sono il tipo di dato 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.

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

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

Rappresentare un tipo astratto di dato è costituito da un insieme di oggetti ed un insieme di operazioni su tali oggetti.

5.2 Insiemi

Negli insiemi tramite liste bisogna definire le loro operazioni. Si considera una possibile implementazione della differenza tra due insiemi:

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

In maniera iterativa:

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

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.

Un problema tipico dell'informatica è la somma di sottoinsiemi. dato un insieme di interi positivi S ed un intero n, determinare un sottoinsieme 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, quindi non è serve 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 $\hat{\bar{S}}$, uno complementare dell'altro. All'inizio si ha che $\hat{S} = \emptyset$ è lo spazio di ricerca delle soluzioni e $\hat{\bar{S}} = S$. 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 $\hat{\bar{S}} = \emptyset$ allora non è una soluzione valida, se non è vuoto allora si sceglie un elemento $x \in \hat{\bar{S}}$, e si cerca una soluzione aggiungendo x alla soluzione $\hat{S}$ con $\hat{S} \cup \{x\}$ e $\hat{\bar{S}} \setminus \{x\}$ oppure senza aggiungerlo alla soluzione con $\hat{S}$ e $\hat{\bar{S}} \setminus \{x\}$.

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

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

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)
```

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 serach_all tot xs
               else mapcons
```

5.3.1 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 di una funzione 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) )
```

Esercizi

Si definisca la funzione `find`, definita nel modulo List, tale che `find p lst` restituisca il primo elemento della lista `lst` che soddisfa il predicato `p`, altrimenti solleva un'eccezione.

```
exception NotFound;;
let rec find p = function
      [] -> raise NotFound
    | x::xs -> if (p x) then x
               else find p xs
```

Si definisce la funzione `takeawhile p lst` che riporti la più lunga parte iniziale di `lst` costituita da tutti gli elementi che soddisfano il predicato `p`.

```
let rec takeawhile p = function
      [] -> []
    | x::xs -> if (p x) then x::(takeawhile p xs)
               else []
```

Definire ora un predicato `p`, tale che la funzione `takeawhile` restituisca la parte iniziale contenente solo numeri pari:

```
takeawhile (function x -> x mod 2 = 0) l
```

Si definisca la funzione `partition: ('a → bool) → 'a list → ('a list * 'a list)`, tale che applicata ad un predicato `p` ed una lista `lst` restituisca una coppia di liste, dove la prima contiene tutti gli elementi che soddisfano il predicato, mentre la seconda lista contiene tutti gli elementi che non lo soddisfano:

```
let partition p l =
    let rec aux (yes, no) = function
          [] -> (yes, no)
        | x::xs -> if (p x) then aux ((yes@[x]), no) xs
                   else aux (yes, (no@[x])) xs
    in aux ([], []) l
```

Definire la funzione `pairwith`, che dato un elemento ed una lista restituisce una lista di coppie formate dall'elemento passato e l'elemento corrispondente della lista passata, utilizzando `List.map`

```
let pairwith x l = List.map (function y -> (x, y) ) l
```

Definire la funzione `setdiff` per la differenza insiemistica, utilizzando `List.filter`:

```
let mem l x = List.exists ((=) x) l;;
let setdiff l1 l2 = List.filter (not (mem l2)) l1
```

Un'altra soluzione possibile è la seguente:

```
list rec setdiff l1 = function
      [] -> l1
    | x::xs -> setdiff(List.fiter (function x -> x <> n) l1) xs

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

7 Esercitazione 2

Definire una funzione `ultime_cifre: -> int * int` che riporti il valore intero delle due ultime cifre di un `int`. Dato un numero, il suo modulo base 10 restituisce l'ultima cifra. Mentre per ottenere la penultima si effettua una divisione intera per 10 per eliminare l'ultima cifra e si riapplica il modulo base 10 per ottenere questa cifra:

```
let ultime_cifre x = (abs(x) mod 10, abs(x/10) mod 10);;
```

Una cifra è bella se è 0, 3 o 7; un numero è bello se la sua ultima cifra è bella e la penultima (se esiste) non lo è. Definire un predicato `bello: int -> bool`, che determini se un numero è bello, la funzione non deve mai sollevare eccezioni, ma riportare sempre un `bool`.

```
let bello x =
    match abs(x mod 10) with
        0 | 3 | 7 -> (match (abs(x mod 10))/10 with
            0 | 3 | 7 -> x < 10
                3 | 7 -> false
                | 0 -> x < 100
                | _ -> true)
        | _ -> false;;
```

Scrivere una funzione `data: int * string -> bool`, che applicata ad una coppia `(d, m)` di un intero `d` e una stringa `m`, determini se la coppia rappresenta una data corretta, assumendo che l'anno non sia bisestile, si assuma che i mesi siano rappresentati da stringhe con caratteri minuscoli. La funzione on deve sollevare eccezioni, ma riportare sempre un `bool`. È utile scrivere una funzione `gdm` per determinare i giorni di un mese:

```
let gdm m =
    match with
    "gennaio"  | "marzo" | "maggio" | "luglio" | "agosto" | "ottobre" | "dicembre" -> 31
    | "aprile" | "giugno" | "settembre" | "novembre" -> 30
    | "febbraio" -> 28
    | _ -> 0
```

La funzione principale è quindi:

```
let data (d,m) = d <= gdm m && d > 0;;
```

8 Esercitazione 3

```
exception NoElement ;;
let maxlist = function
      [] -> raise NoElement
    | x::xs -> let rec aux m = function
          [] -> m
        | x::xs -> if x > m then aux x xs
                   else aux m xs
        in aux x xs
```

Si ipotizza di non avere a disposizione l'operatore `@`, definire la concatenazione:

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

In modo iterativo diventa:

```
let append l1 l2 =
    let rec aux a = function
          [] -> a
        | x::xs -> aux (x::a) xs
    in aux l2 List.rev(l1)

let rec nth n = function
      [] -> raise NoElement
    | x::xs -> if n < 0 then raise NoElement
               else if n = 0 then x
                    else nth (n-1) xs;;

let rec nondec = function
      [] | [a] -> true
    | x::y::xs -> if x > y then false
                  else nondec (y::xs)

let min_dei_max ls =
    let rec listmax acc = function
          [] -> acc
        | x::xs -> listmax ((maxlist x)::acc) xs
    in let rec minlist = function
          [] -> raise NoElement
        | x::xs -> try let y = minlist(xs)
                       in min x y
                   with _ -> x
    in minlist(listmax [] ls);;
```

Concatena l'accumulatore con i massimi di ciascuna lista del secondo argomento, l'argomento implicito è una lista di liste, essendo l'argomento della funzione esterna `ls`. Si può inferire il tipo dato che ad un elemento della lista `x` viene passato come argomento a `maxlist`, quindi deve essere una lista.

`split2` agisce in modo simile alla funzione `split` definita precedentemente, divide in due parti ...