Introduzione alla Programmazione Funzionale con Objective Caml

Dispense del corso di Programmazione funzionale

Marta Cialdea Mayer

Anno accademico 2017/2018
(con qualche correzione successiva)

Indice

Capitolo 1

Il nucleo di un linguaggio funzionale

1.1 Programmazione funzionale e ML

La programmazione funzionale è basata sul paradigma di calcolo della *riduzione*, secondo il quale si calcola riducendo un'espressione a un'altra più semplice o più vicina a un valore, cioè a un'espressione non ulteriormente semplificabile. Ad esempio:

$$(6+3) \times (8-2) \Rightarrow 9 \times (8-2) \Rightarrow 9 \times 6 \Rightarrow 54$$

Un programma funzionale è costituito dalla definizione di un insieme di funzioni, che possono richiamarsi l'una con l'altra. Eseguire un programma funzionale consiste nel calcolare il valore di una data espressione, semplificandola fin dove possibile.

In un linguaggio funzionale, il calcolo è eseguito fondamentalmente per mezzo dell'applicazione di funzioni. Da un punto di vista astratto, un programma si può vedere come un'operazione che associa un output a ciascun input possibile. Un programma funzionale è proprio una funzione, e le strutture di controllo fondamentali sono l'applicazione di una funzione a un argomento e la composizione di funzioni. I costrutti di base sono espressioni, non comandi, costruite a partire da espressioni semplici (costanti e variabili), mediante l'applicazione e la composizione di operazioni.

Lo scheletro concettuale di molti linguaggi funzionali, come ML, Miranda, Hope e lo stesso LISP è costituito dal λ-*calcolo* (lambda-calcolo), un sistema formale sviluppato per analizzare formalmente le funzioni e il loro calcolo. Il linguaggio del λ-calcolo è molto semplice: ammette soltanto espressioni per denotare funzioni e la loro applicazione. Tuttavia, dietro questa semplicità sintattica, si cela un linguaggio di grande potenza espressiva: rispetto alla capacità di esprimere algoritmi, il λ-calcolo è altrettanto potente quanto un linguaggio di programmazione. D'altro canto, il λ-calcolo è una *teoria delle funzioni*. Esso è nato infatti prima dello sviluppo dei linguaggi di programmazione; è stato introdotto dal matematico e logico Alonzo Church negli

anni 30, allo scopo di dare una definizione formale di ciò che è calcolabile e studiare i limiti della calcolabilità.

Il λ-calcolo è costruito sulla base di una notazione specifica per esprimere funzioni. La notazione matematica ordinaria confonde spesso una funzione con il suo valore. Consideriamo, ad esempio, l'espressione $x + y$; essa può in realtà essere interpretata in modi diversi:

1. come il numero $x + y$ (assumendo che x e y abbiano un determinato valore);
2. come una funzione f con argomento x, cioè $f : \ x \mapsto x + y$ (assumendo che y abbia un determinato valore);
3. come una funzione g di y, cioè $g : \ y \mapsto x + y$ (assumendo che x abbia un determinato valore);
4. come una funzione h di due argomenti, x e y, cioè $h : \ (x, y) \mapsto x + y$.

La "notazione lambda" consente di esprimere sintatticamente la differenza tra questi concetti:

1. il numero $x + y$ si scrive sempre $x + y$;
2. la funzione f si scrive $\lambda x.(x + y)$ ("funzione di x, che riporta $x + y$");
3. la funzione g si scrive $\lambda y.(x + y)$ ("funzione di y, che riporta $x + y$");
4. la funzione h si scrive $\lambda(x, y).(x + y)$ ("funzione di x e y – o meglio, della *coppia* (x, y), che riporta $x + y$").[1]

Nei linguaggi di programmazione funzionale, come nel λ-calcolo, è possibile costruire espressioni che denotano funzioni, le quali sono trattate come **oggetti di prima classe**: esse possono essere componenti di una struttura dati, o costituire l'argomento o il valore di altre funzioni. Anche dal punto di vista notazionale i linguaggi funzionali si rifanno al λ-calcolo. Ad esempio, in Objective Caml (chiamato anche, per brevità, OCaml), il linguaggio funzionale della famiglia ML che utilizzeremo in questo testo, la notazione corrispondente alla notazione lambda rimpiazza il λ con la parola chiave `function` e il punto con `->`:

2. la funzione f si scrive `function x -> x+y`;
3. la funzione g si scrive `function y -> x+y`;
4. la funzione h si scrive `function (x,y) -> x+y`.[2]

Le espressioni 2-4 denotano tutte funzioni e possono occorrere ovunque possa occorrere il nome di una funzione.

Come in altri linguaggi, è possibile definire funzioni specificandone il codice relativo. Ad esempio, la funzione `double`, che riporta il doppio del suo argomento, si definisce in OCaml come segue:

[1] In realtà il λ-calcolo non prevede tipi strutturati come le coppie, e la funzione h sarebbe piuttosto denotata da $\lambda x.\lambda y.(x + y)$ (si veda il paragrafo 1.15.1).

[2] O, nella sua forma *currificata*, `function x -> function y -> x+y`.

```
let double = function x -> 2*x;;
```

o anche, secondo una sintassi più familiare:

```
let double(x) = 2*x;;
```

Quando una funzione è definita, è possibile applicarla a un argomento, ottenendo come risultato il valore di tale funzione per l'argomento dato. Ad esempio, quando la funzione è definita, il valore di `double(3)` è 6.

Come già accennato, in un linguaggio funzionale, a differenza di altri linguaggi, le funzioni sono oggetti di prima classe. Di conseguenza, i linguaggi funzionali consentono l'uso di **funzioni di ordine superiore**, cioè funzioni che prendono funzioni come argomento o riportano funzioni come valore, in modo assolutamente generale. Ad esempio, si può definire una funzione `comp` per la composizione di funzioni, che prende una coppia di funzioni `(f,g)` come argomento e restituisce una funzione: `comp(f,g)` è la composizione di `f` e `g`. Utilizzando la notazione `function`, ad esempio, l'espressione `comp(double,function x -> 3*x)` è la funzione che risulta dalla composizione di `double` e la funzione "triplo" (`function x -> 3*x`), in altre parole è la funzione che moltiplica per 6 il suo argomento.

La modalità fondamentale di calcolo, in un linguaggio funzionale, è la **valutazione di espressioni**, e non l'assegnazione di valori alle variabili. In un linguaggio funzionale *puro* di conseguenza **non esistono effetti collaterali**, cioè operazioni che modificano in modo permanente il valore di una variabile. Mentre in un linguaggio imperativo l'assegnazione di un valore, per esempio `a+b`, a una variabile `x` modifica il valore della variabile `x`, un linguaggio funzionale, tipicamente, calcola il valore di `a+b` ma non ne immagazzina il risultato in alcun luogo.

In realtà molti linguaggi funzionali, ed in particolare i linguaggi della famiglia ML, hanno delle caratteristiche "impure" che prevedono effetti collaterali, come la stampa di un output e la gestione di oggetti che si comportano in modo analogo alle variabili di un linguaggio imperativo, ma, dal punto di vista dello stile funzionale, queste sono considerate aberrazioni utilizzate soltanto dove necessario o quando portano a un notevole incremento di efficienza.

Infine, notiamo un ulteriore elemento di differenza rispetto ai linguaggi imperativi, che influenza in modo evidente lo stile di programmazione in un linguaggio funzionale: in un linguaggio funzionale puro non esistono strutture di controllo predefinite per la realizzazione di cicli quali `for, while, repeat`, ma il principale meccanismo di controllo è la ricorsione.

Consideriamo ad esempio l'algoritmo di Euclide per il calcolo del massimo comun divisore (gcd) di due interi non negativi n e m (e non entrambi nulli), basato sulla seguente proprietà:

$$\begin{array}{lcll} gcd(n,m) & = & n & \text{se } m = 0 \\ gcd(n,m) & = & gcd(m, n \bmod m) & \text{se } m \neq 0 \end{array}$$

In un linguaggio imperativo, un programma per il calcolo del gcd si può basare su un algoritmo iterativo, che potrebbe essere implementato in C (o Java) in questo modo:

```
int gcd(int n, int m) {
  int tmp;
  while (m != 0) {
    tmp = m;
    m = n % m;
    n = tmp;
  }
  return n;
}
```

La programmazione imperativa è basata su comandi, che operano sulla memoria (stato del programma). Per capire il programma (e provarne la correttezza) occorre tenere traccia delle modifiche dello stato.

Al contrario, in un linguaggio funzionale la funzione `gcd` può essere definita in modo dichiarativo, rispecchiando fedelmente le uguaglianze sopra enunciate. Ad esempio, nella sintassi di OCaml:

```
let rec gcd (n,m) =
  if m=0
  then n
  else gcd (m,n mod m)
```

O anche:

```
let rec gcd = function
    (n,0) -> n
  | (n,m) -> gcd(m,n mod m)
```

Ovviamente, sarebbe possibile definire il `gcd` in modo analogo anche in molti linguaggi imperativi. Ma anche quando la ricorsione è possibile, in un linguaggio imperativo è meno "in stile". Al contrario, nella programmazione funzionale la ricorsione è il costrutto di controllo fondamentale.

La generazione moderna dei linguaggi funzionali è molto evoluta rispetto ai primi linguaggi funzionali, sia in termini di concezione dei linguaggi, sia in termini di efficienza degli stessi. In particolare, la reputazione dei linguaggi funzionali di essere inefficienti non è più attuale: la qualità del codice può variare da *migliore* del C a un ordine di grandezza peggiore, con il caso tipico che è attorno a 2 volte più lenti del C, in dipendenza dal particolare linguaggio e dall'applicazione. D'altro canto, la produttività aumenta da un ordine di grandezza a un fattore 4: il codice è più corto, più veloce da scrivere, più facile da mantenere.

Per avere una rapida carrellata dei principali linguaggi funzionali moderni e delle loro applicazioni per scopi "seri", si può leggere l'articolo [7]. Citiamo da questo lavoro applicazioni alle telecomunicazioni da parte della Ericsson (in Erlang), verifica di protocolli di arbitraggio alla HP, applicazioni da parte del dipartimento della difesa di Sydney, controllo di safety conditions di transazioni di DB, un linguaggio di alto livello per la formulazione di queries per DB (CPL/Keisli, che è stato utilizzato

per rispondere con successo alle 12 queries, elencate in un precedente workshop, ritenute "impossibili"), la gestione dei voli tra gli areoporti di Parigi di Orly e Roissy, una libreria di protocolli che si può utilizzare per costruire velocemente applicazioni distribuite (Ensemble, scritto in Objective Caml), ecc.

Le osservazioni generali fatte fin qui si applicano ai linguaggi della famiglia ML ed in particolare a OCaml. Nel resto di questo paragrafo considereremo alcune caratteristiche particolari della famiglia ML. Il primo compilatore ML è stato implementato nel 1974. Il suo nome è l'acronimo di "Meta Language": ML è stato infatti progettato come meta-linguaggio per programmare un dimostratore di teoremi che potesse seguire diverse strategie di dimostrazione. La sua diffusione è da allora cresciuta moltissimo in tutto il mondo, sia per lo sviluppo di applicazioni, sia a livello didattico. Negli anni 90 è stato definito l'attuale standard, Standard ML. Il linguaggio Caml, disponibile nelle due versioni Caml Light e Objective Caml, è sviluppato e distribuito dall'INRIA (Francia) dal 1984. È disponibile al sito Web `http://caml.inria.fr/`. Caml Light, lo strumento utilizzato in [2] per introdurre la programmazione funzionale, è un sottolinguaggio di Objective Caml (documentato in [3]) ma rispetto a quest'ultimo presenta alcune differenze sintattiche anche negli elementi di base. OCaml, d'altro canto, possiede caratteristiche che esulano dal paradigma funzionale, come la possibilità di definire classi e oggetti (che non verrano considerate in questo testo).

Nel resto di questo paragrafo consideriamo le principali caratteristiche dei linguaggi della classe ML. Se il lettore non comprenderà appieno alcuni concetti ad una prima lettura, si consiglia di rileggere questo paragrafo dopo aver fatto i primi passi con un linguaggio funzionale. Cogliamo qui comunque l'occasione per citare [6], un'ottima introduzione alla programmazione funzionale mediante Standard ML, e i testi [1] e [4] per approfondire la conoscenza di OCaml. Nel seguito ci riferiremo al generico linguaggio della famiglia ML come ML, e negli esempi utilizziamo la sintassi di OCaml.

ML è un linguaggio a tipizzazione forte

I linguaggi di programmazione di alto livello trattano, normalmente, più classi di oggetti, manipolando espressioni di diverso *tipo*. Il tipo dei sottotermini di un'espressione è importante per determinare se essa è ben formata: una funzione di non si applica a qualunque tipo di argomenti. L'implementazione di qualsiasi linguaggio esegue un **controllo dei tipi** prima di valutare un'espressione, eseguire un'operazione, ecc. Il controllo dei tipi consente di determinare se le espressioni sono usate con coerenza nel programma e di non autorizzare tentativi di calcolare espressioni senza senso, come ad esempio la sottrazione di un booleano da un intero. Quando esattamente avviene tale controllo determina una classificazione dei linguaggi in due categorie:

1. linguaggi a **tipizzazione statica**, o **tipizzazione forte**, in cui il tipo di ogni espressione è determinabile a tempo di compilazione (cioè esaminando il programma, senza eseguirlo); nei linguaggi di questa classe, come il Pascal, C o

ML, ad ogni espressione corretta è associato un tipo, che determina in quale contesto l'espressione può essere utilizzata.

2. linguaggi a **tipizzazione dinamica**, in cui il tipo degli operandi di un'operazione viene controllato subito prima di eseguire l'operazione stessa, a tempo di esecuzione. Il LISP è un esempio di linguaggio a tipizzazione dinamica; in esso, non tutte le espressioni hanno associato un tipo, ma soltanto i *valori* (espressioni non ulteriormente riducibili).

Il controllo dei tipi consente di identificare un numero incredibilmente alto di errori di programmazione ed è quindi importante che sia eseguito prima possibile. Il controllo statico dei tipi delle espressioni ha il duplice vantaggio di non rallentare l'esecuzione dei programmi e di prevenire errori di esecuzione. La tipizzazione forte consente di riconoscere a tempo di compilazione uno dei tipi di errori più frequenti e garantisce che durante l'esecuzione di un programma non possano verificarsi errori di tipo.

ML è un linguaggio a tipizzazione forte: ogni espressione ha un tipo che può essere determinato a tempo di compilazione (cioè esaminando il programma, senza eseguirlo) e che restringe l'insieme delle espressioni corrette. Ad esempio, se una funzione si applica ad argomenti interi, non può essere applicata a reali.

Tuttavia, a differenza di molti altri linguaggi a tipizzazione forte, che richiedono esplicite dichiarazioni di tipo per ogni espressione, ML ha un **meccanismo di inferenza di tipi** che gli consente di dedurre qual è il tipo di un'espressione senza bisogno di dichiarazioni esplicite.

ML ha un sistema di tipi polimorfo

Il polimorfismo è la possibilità che un oggetto abbia più di un tipo; particolarmente interessanti sono le funzioni polimorfe, che possono accettare argomenti di diversi tipi, e le strutture dati polimorfe. Spesso è necessario considerare strutture di dati di tipo diverso ma con struttura molto simile, come ad esempio pila di interi, pila di reali, pila di stringhe, ecc. In molti linguaggi a tipizzazione forte, si deve definire un tipo di dati diverso per ciascuna di tali strutture e, in corrispondenza, diverse funzioni "push", "pop", ecc. In ML, al contrario, è possibile definire un'unica struttura per le pile, con un'unica funzione push e un'unica funzione pop, che funzioneranno comunque, indipendentemente dal tipo degli elementi della pila. Il meccanismo di inferenza dei tipi di ML deriva per ogni espressione il suo *tipo più generale*, che determina tutto l'insieme dei contesti in cui l'espressione può essere correttamente utilizzata.

ML è un linguaggio interattivo

I linguaggi della famiglia ML possono essere eseguiti in modalità interattiva (molte implementazioni dispongono anche di un compilatore). In tale modalità ML esegue un ciclo di *lettura, valutazione e stampa*: ogni espressione immessa dal programmatore viene analizzata, compilata, e viene calcolato e stampato il suo valore, assieme al suo tipo.

Quando si entra in OCaml (per esempio con il comando `ocaml`), appare il "prompt" del linguaggio, della forma:

```
        Objective Caml version 4.02.3

#
```

Il cancelletto è il "*prompt*" di OCaml. Possiamo immettere un'espressione, seguita da due punti e virgola (;;), e OCaml ne calcolerà il valore:

```
# 3*8;;
- : int = 24
```

Nella risposta di OCaml, è evidenziato il fatto che il valore calcolato è un intero (di tipo `int`).

Al prompt di OCaml si può immettere una *dichiarazione di valore*, introdotta dalla parola chiave `let`. Ad esempio:

```
# let three = 3;;
val three : int = 3
# three*8;;
- : int = 24
```

Qui, la dichiarazione di valore può sembrare molto simile ad un'assegnazione, ma come vedremo ci sono delle importanti differenze.

Una funzione viene definita con la stessa sintassi:

```
# let double x = 2*x;;
val double : int -> int = <fun>
```

Il valore di una funzione non è stampabile. Quindi in questo caso la risposta di OCaml specifica soltanto il tipo di `double`: si tratta di una funzione da interi a interi. Come si vede, ML ha dedotto il tipo della funzione: il prodotto (operazione predefinita) è definito solo su coppie di interi, quindi `x` deve essere un intero, e il risultato è anch'esso un intero.

Ora che la funzione double è definita, possiamo utilizzarla nelle espressioni:[3]

```
# double 5;;
- : int = 10
# double(double three);;
- : int = 12
```

Un programma funzionale consiste in un insieme di *dichiarazioni* (cioè definizioni di funzioni, di valori, dichiarazioni di tipo, ecc). Il modo interattivo consente di valutare qualsiasi espressione in cui compaiono gli identificatori definiti nel programma.

[3]Si noti che non è sempre necessario mettere tra parentesi l'argomento di una funzione.

ML è un linguaggio a scopo statico

ML determina a tempo di compilazione il valore delle variabili in una dichiarazione, con una conseguente maggiore modularità, chiarezza ed efficienza dei programmi. Ciò significa che, se la dichiarazione di una variabile o funzione contiene il riferimento ad una variabile `x` ogniqualvolta venga valutata tale variabile o funzione, sarà sempre utilizzato il valore che `x` aveva nel momento in cui è stata introdotta la dichiarazione (vedi paragrafo 1.8).

ML ha un meccanismo per la gestione di errori

In ML esitono oggetti di un tipo particolare, chiamati *eccezioni*, che consentono di gestire condizioni "eccezionali" (errori) che possono verificarsi durante l'esecuzione. Le eccezioni hanno la particolarità di poter essere argomento e valore di qualsiasi funzione e non inficiano quindi il sistema dei tipi di ML.

ML ha un sistema di moduli

Un programma di dimensioni notevoli è costruito come un insieme di moduli (o *strutture*) interdipendenti, che possono essere legate assieme tramite *funtori*. ML consente la compilazione separata dei moduli e la possibilità di esportare e importare funzioni.

Il linguaggio OCaml, in particolare, dispone di una libreria standard di moduli, la cui descrizione si può trovare nel manuale di riferimento [3].

1.2 I tipi di base

Le espressioni costituiscono la classe fondamentale di oggetti sintattici di un linguaggio funzionale. Vi sono espressioni complesse, che possono essere semplificate, e espressioni semplici, i *valori*. Ogni espressione ha un suo valore e il processo di calcolo di tale valore è la *valutazione dell'espressione*. Un'espressione viene valutata in un *ambiente*, costituito da una collezione di legami variabile-valore. Quando si inizia una sessione OCaml, l'ambiente contiene la definizione di tutti gli identificatori predefiniti. Tale ambiente (chiamato il modulo "Pervasives") si estende con l'aggiunta di nuovi legami ogniqualvolta viene valutata una dichiarazione.

Ad esempio, `3+2` è un'espressione e il suo valore è `5`. Poiché nell'espressione non compaiono variabili, il suo valore è indipendente dall'ambiente. Al contrario, il valore di un'espressione quale `if x>0 then x else -x` dipende dal valore che ha `x` nell'ambiente in cui essa viene valutata:

```
# let x = 3;;
val x : int = 3
# if x>0 then x else -x;;
- : int = 3
# let x = -14;;
val x : int = -14
```

```
# if x>0 then x else -x;;
- : int = 14
```

Un tipo è un insieme di valori. Per esempio, gli interi (`int`) costituiscono un tipo, così come i reali (`float`), le stringhe (`string`), i caratteri (`char`), i booleani (`bool`). Un'espressione di un certo tipo è un'espressione il cui valore è un oggetto di quel tipo. Cosí `3` è un intero, `3+5` è un'espressione di tipo `int`.

1.2.1 Interi e reali

Gli interi vengono scritti nel modo abituale. I reali (`float`) sono denotati da un intero seguito dal punto decimale e da zero, una o più cifre (ad esempio, `3.` oppure `3.0`); oppure possono essere scritti in notazione esponenziale (ad esempio, `3.2E-10`).

Sugli interi sono disponibili le operazioni abituali, `+, -, *, /` e `mod`; inoltre sono predefinite `succ` (successore) e `pred` (predecessore), entrambe di tipo `int -> int`. Le quattro operazioni sui reali sono denotate da `+., -., *., /.`: esse hanno un tipo diverso da quello delle corrispondenti operazioni sugli interi. Su entrambi i tipi numerici (come su stringhe, caratteri e altri tipi di dati) sono definite le relazioni `=, <, <=, >, >=, <>`, che riportano un booleano: la valutazione di un'espressione di relazione produce un valore di tipo `bool`:

```
# 3 > 9;;
- : bool = false
# 3*2 > 15/6;;
- : bool = true
```

L'operazione denotata con `**` è l'elevazione a potenza sui reali, e `sqrt` la radice quadrata. Per le altre funzioni matematiche predefinite si rimanda al manuale di OCaml [3].

1.2.2 Stringhe e caratteri

Le stringhe sono sequenze di caratteri racchiuse tra virgolette (sono disponibili anche "caratteri speciali", ad esempio per il ritorno carrello, `'\n'`, e la tabulazione, `'\t'`) e i valori di tipo `char` sono denotati da caratteri racchiusi tra apici (ad esempio, `'a'`).

Su stringhe e caratteri sono definite le operazioni di confronto e sulle stringhe è definita l'operazione di concatenazione:

```
# "programmazione " ^ "funzionale";;
- : string = "programmazione funzionale"
```

Il carattere in posizione `n` in una stringa `s` è identificato dall'espressione `s.[n]`, tenendo presente che il primo carattere è in posizione 0, il secondo in posizione 1, ecc:

```
# "ABCDEFG".[2];;
- : char = 'C'
```

Alcune funzioni predefinite operano conversione di valori: `string_of_bool`, `string_of_int` e `string_of_float` riportano la stringa corrispondente, rispettivamente, a un booleano, un intero o un reale, mentre `int_of_string` converte una stringa in un intero (quando la stringa rappresenta un intero) e in modo analogo funziona `float_of_string`.

Altre funzioni utili sulle stringhe sono definite nel modulo `String` della libreria standard di OCaml. Per accedere alle funzioni di un modulo, il nome della funzione è preceduta dal nome del modulo e un punto. Ad esempio, `String.length` riporta la lunghezza (numero di caratteri) di una stringa:

```
# String.length "123456";;
- : int = 6
```

Il modulo `Char` della libreria standard di OCaml contiene invece alcune funzioni utili sui caratteri, quali `Char.code`, che riporta il codice ASCII del suo argomento, e `Char.chr` che riporta il carattere con il codice dato:

```
# Char.code 'A';;
- : int = 65
# Char.chr 65;;
- : char = 'A'
```

1.2.3 Booleani

Il tipo `bool` è costituito unicamente dai due valori `true` e `false`. Come abbiamo visto, le espressioni costruite mediante operatori relazionali sono di tipo `bool`. Le operazioni booleane di negazione, congiunzione e disgiunzione sono `not`, `&&` (o `&`) e `||` (o `or`).

Sebbene non sia di tipo `bool`, consideriamo qui anche l'espressione condizionale, che ha la forma

$$\texttt{if } E \texttt{ then } E_1 \texttt{ else } E_2$$

in cui E è un'espressione di tipo `bool` e E_1, E_2 sono espressioni *dello stesso tipo*; il valore di `if` E `then` E_1 `else` E_2 è E_1 se il valore di E è `true`, E_2 altrimenti. Si noti che la parte "else" non è opzionale: `if` E `then` E_1 `else` E_2 è un'espressione, non un comando: se mancasse la parte "else", l'espressione non avrebbe un valore quando il valore di E è `false`. Inoltre, a causa della tipizzazione forte, non è ammesso che E_1 e E_2 abbiano tipo diverso; il tipo di `if` E `then` E_1 `else` E_2 deve poter essere determinato a tempo di compilazione ed esso è proprio il tipo di E_1 ed E_2.

Ecco qualche esempio:

```
# 3 < 5 && "pippo" < "pluto";;
- : bool = true
# false || true;;
- : bool = true
```

```
# if 3 < 5 then 0 else "ciao";;
Characters 21-27:
This expression has type string but is here used with type int
# if 3 < 5 then 0 else 1;;
- : int = 0
```

Il messaggio di errore sopra riportato da OCaml informa che nell'espressione è stato riscontrato un errore di tipo: l'espressione `"ciao"` è di tipo `string`, ma è utilizzata dove ci si aspetta un'espressione di tipo `int`, cioè dello stesso tipo di 0 (nel ramo `then`).

Si noti che il modo in cui vengono valutate le espressioni condizionali, le congiunzioni (`&&`) e le disgiunzioni (`||`) è particolare:

- nel valutare un'espressione della forma `if` E `then` E_1 `else` E_2, viene valutata innanzitutto E; se il suo valore è `true`, allora viene valutata E_1 (e non E_2), altrimenti viene valutata E_2 (e non E_1);
- per valutare un'espressione della forma E_1 `&&` E_2, viene innanzitutto valutata E_1; se il valore di E_1 è `true`, allora viene valutata E_2 e viene riportato il valore di E_2 stesso (`true`, se E_2 è anch'essa `true`, `false` altrimenti). Altrimenti, se E_1 è `false`, viene riportato `false` senza valutare E_2;
- per valutare un'espressione della forma E_1 `||` E_2, viene innanzitutto valutata E_1; se il valore di E_1 è `false`, allora viene valutata E_2 e viene riportato il valore di E_2 stesso (`true`, se E_2 è `true`, `false` altrimenti). Altrimenti, se E_1 è `true`, viene riportato `true` senza valutare E_2;

In questi casi (e solo in questi casi) il processo di valutazione è *pigro* (*lazy*): le sottoespressioni sono valutate solo quando è necessario. In tutti gli altri casi, per valutare un'espressione vengono comunque valutate tutte le sottoespressioni che la compongono (vedi paragrafo 1.8).

1.2.4 Coppie e tuple

Se t_1 e t_2 sono tipi, $t_1 \times t_2$ è il tipo delle coppie ordinate il cui primo elemento è di tipo t_1 ed il secondo di tipo t_2. Attenzione: in una espressione di tipo, il segno $\times$ non è la moltiplicazione, ma indica il prodotto cartesiano. Ad esempio, `int` $\times$ `int` è il tipo delle coppie di interi. Una coppia ordinata si scrive `(`e_1`,`e_2`)`, dove e_1 ed e_2 sono espressioni (in alcuni contesti è possibile omettere le parentesi esterne). Così l'espressione `(5*6,4*8)` è una coppia, di tipo `int` $\times$ `int`. In altri termini, $\times$ è un *costruttore di tipi*: applicato a due tipi t_1 e t_2, riporta il tipo delle coppie ordinate il cui primo elemento è di tipo t_1 ed il secondo di tipo t_2. Si noti che il costruttore di tipi $\times$ viene indicato, in OCaml, dallo stesso simbolo "`*`" utilizzato per il prodotto:

```
# (5*6,4*8);;
- : int * int = 30, 32
```

Sulle coppie sono definite le operazioni `fst` e `snd`, che riportano, rispettivamente, il primo ed il secondo elemento della coppia:

```
# fst (3*8,true);;
- : int = 24
# snd (false, sqrt 4.0);;
- : float = 2.000000
```

Possiamo generalizzare e considerare terne, quadruple o "tuple" di dimensione qualsiasi:

```
# (true,5*4,"venti");;
- : bool * int * string = (true, 20, "venti")
# ((if 3<5 then "true" else "false"),
    10.3, "hjk".[0], int_of_string "5");;
- : string * float * char * int = ("true", 10.3, 'h', 5)
```

(Attenzione: il primo elemento della quadrupla precedente va incluso tra parentesi, per evitare che OCaml legga l'espressione come: `(if 3<5 then "true" else ("false",10.3,"hjk".[0],int_of_string "5"))` e riporti un errore di tipo).

Possiamo naturalmente anche avere tuple i cui elementi sono a loro volta di tipo non semplice, come altre tuple o persino funzioni:

```
# let double(x) = 2*x;;
val double : int -> int = <fun>
# (double,(true && not false, 6*5));;
- : (int -> int) * (bool * int) = (<fun>, (true, 30))
```

Si noti che il valore dell'espressione `(double,(true && not false, 6*5))` non è stampabile, in quanto la prima componente della coppia è una funzione; al suo posto dunque OCaml scrive `<fun>` per indicare che si tratta di una funzione. Il tipo dell'espressione è quello delle coppie il cui primo elemento è una funzione di tipo `int -> int` e il secondo elemento è a sua volta una coppia, di tipo `bool` $\times$ `int`.

1.2.5 Record*

(I paragrafi contrassegnati con un asterisco, come questo, trattano argomenti dei quali non si parla nel corso.)

Un record è un oggetto composito, costituito da un numero finito di componenti di tipo non necessariamente omogeneo, ciascuna delle quali è identificata da un nome o *etichetta*, e contiene un valore di un tipo determinato. Le componenti, chiamate *campi* del record, possono contenere valori di tipo diverso dagli altri. I record sono di fatto una generalizzazione delle tuple; a differenza di queste, le componenti di un record sono distinte mediante un nome, anziché per la posizione.

In OCaml, prima di utilizzare espressioni di tipo record, è necessario definire il particolare tipo di record che si vuole utilizzare, allo scopo di introdurre le etichette dei campi. La definizione di un tipo ha la sintassi seguente

```
type <nome-del-tipo> = <specifica-del-tipo>
```

(si veda il paragrafo 3.1). Nel caso dei record, la specifica del tipo ha la forma seguente:

```
{ <label-1> : <tipo-1> ; .... ; <label-n> : <tipo-n>}
```

e un record è denotato da un'espressione della forma

```
{ <label-1> = <valore-1> ; .... ; <label-n> = <valore-n>}
```

Ad esempio:

```
# type studente = {nome: string; matricola: int * int};;
type studente = { nome: string; matricola: int * int }
# let s1 = {nome = "Emidio Bufalini"; matricola = 80129,56};;
val s1 : studente = {nome="Emidio Bufalini"; matricola=80129, 56}
```

Il valore di un campo `c` di un record `r` può essere selezionato mediante l'espressione `r.c`:

```
# s1.nome;;
- : string = "Emidio Bufalini"
# s1.matricola;;
- : int * int = (80129, 56)
```

L'ordine in cui sono specificati i valori dei campi nell'espressione che denota un record non è rilevante:

```
# s1;;
- : studente = {nome="Emidio Bufalini"; matricola=80129, 56}
# s1 = { matricola = 80129, 56; nome = "Emidio Bufalini" };;
- : bool = true
```

1.2.6 I tipi funzionali

I tipi delle funzioni hanno la forma `tipo1 -> tipo2`, dove `tipo1` è il tipo degli argomenti della funzione e `tipo2` il tipo dei valori riportati dalla funzione. I tipi `tipo1` e `tipo2` possono essere qualsiasi (tipi semplici, tuple, record, ...), ed in particolare possono essere tipi funzionali. Così ad esempio `(int -> bool) -> bool` è un tipo funzionale: il tipo delle funzioni che prendono come argomenti oggetti di tipo `int -> bool` – dunque funzioni – e riportano un booleano come valore. Anche `int -> (int -> bool)` è un tipo funzionale: il tipo delle funzioni che hanno come dominio gli interi e come codominio le funzioni da interi a booleani.

Si noti che in un'espressione di tipo della forma `tipo1 -> tipo2 -> tipo3` si associa a destra; tale espressione va dunque letta come `tipo1 -> (tipo2 -> tipo3)`.

1.2.7 Il tipo `unit`

Il tipo unit è il tipo il cui unico elemento è `()` – si può pensare che `()` sia la tupla con zero elementi, utilizzata come argomento per "funzioni" senza argomenti o come valore di "funzioni" che di fatto corrispondono a procedure: il loro ruolo consiste negli effetti collaterali che hanno e non è importante il valore riportato. Ad esempio, la funzione predefinita `print_string` è di tipo `string -> unit`: applicata a una stringa, la stampa su video (effetto collaterale) e riporta `()`.

1.3 Il polimorfismo

Come abbiamo accennato, i linguaggi della classe ML ammettono espressioni di tipo *polimorfo*, cioè oggetti con un tipo generico che può essere istanziato in diversi modi. Ad esempio, la funzione `fst` è una funzione **polimorfa**: essa ha più di un tipo, in quanto può essere applicata a coppie qualsiasi. Quando viene applicata alla coppia `(3,true)` è di tipo `int * bool -> int`, se è applicata alla coppia `(3.2,"pippo")` è di tipo `float * string -> float`, ecc. La funzione `fst` si può applicare cioè a qualsiasi espressione il cui tipo abbia la forma $t_1 \times t_2$, riportando un valore di tipo t_1.

Questo si esprime dicendo che il *tipo più generale* di `fst` è $\alpha \times \beta \to \alpha$. In questa espressione di tipo, α e β sono *variabili di tipo*. Un'espressione in cui compaiano variabili di tipo è uno **schema di tipo**; ad esempio $\alpha \times \beta \to \alpha$ è uno schema di tipo: indica un insieme infinito di tipi, tutti quelli della forma

$$t_1 \times t_2 \to t_1$$

Ogni tipo che si ottiene sostituendo α con un tipo e β con un tipo è un'**istanza** di $\alpha \times \beta \to \alpha$. Ad esempio, sono istanze dello schema $\alpha \times \beta \to \alpha$:

```
int * bool -> int
int * int -> int
(int * bool) * (int -> bool) -> (int * bool)
```

OCaml usa nomi preceduti dall'apice (come `'a` e `'b`, al posto di α e β) per indicare *variabili di tipo*. Ad esempio, osserviamo la risposta di OCaml quando al suo prompt scriviamo il nome di una funzione polimorfa:

```
# fst;;
- : 'a * 'b -> 'a = <fun>
```

o quando definiamo la funzione `first` equivalente a `fst`:

```
# let first (x,y) = x;;
val first : 'a * 'b -> 'a = <fun>
```

Come si vede, il processo di inferenza dei tipi ha lasciato alcuni tipi completamente non ristretti: `'a` e `'b` possono essere *qualsiasi*.

Un oggetto polimorfo può avere occorrenze con tipi diversi nella stessa espressione:

```
# (fst (fst,8)) (true,10);;
- : bool = true
```

Il valore di `fst (fst,8)` è la seconda occorrenza di `fst` e questa è applicata a `(true,10)`. La seconda occorrenza di `fst` è quindi di tipo `bool * int -> bool`. La prima occorrenza di `fst` si applica alla coppia `(fst,8)`, che è di tipo:

`(bool * int -> bool) * int`,

quindi la prima occorrenza di `fst` è di tipo

`(bool * int -> bool) * int -> (bool * int -> bool)`.

Un altro esempio di funzione polimorfa è la funzione identità:

```
# let id x = x;;
val id : 'a -> 'a = <fun>
```

Nell'espressione `id true`, il tipo di `id` è `bool -> bool`, mentre il tipo dell'occorrenza di `id` nell'espressione `id fst` è `('a * 'b -> 'a) -> ('a * 'b -> 'a)`.

Nel paragrafo 1.2.3 abbiamo detto che in un'espressione condizionale della forma `if` E `then` E_1 `else` E_2, le sottoespressioni E_1 e E_2 devono essere dello stesso tipo (e restrizioni analoghe saranno enunciate nel seguito per altre strutture sintattiche). Occorre qui specificare cosa si intende per "lo stesso tipo" quando E_1 e/o E_2 sono espressioni polimorfe. Consideriamo ad esempio un'espressione della forma `if` E `then fst else snd`: questa è un'espressione corretta, ad esempio avremo:

```
# if true then fst else snd;;
- : 'a * 'a -> 'a = <fun>
```

Come ha dedotto OCaml il tipo $\alpha \times \alpha \to \alpha$ per questa espressione? Ha "ragionato" così: il tipo più generale di `fst` è $\alpha \times \beta \to \alpha$ e quello di `snd` è $\alpha \times \beta \to \beta$. Ma le occorrenze di `fst` e `snd` nell'espressione `if true then fst else snd` deve essere lo stesso, dunque deve essere $\alpha = \beta$.

In generale, diremo che *due espressioni E e E' hanno lo stesso tipo* se il tipo più generale di E è T, quello di E' è T' ed è possibile sostituire *uniformemente* le variabili di tipo che occorrono in T e T' con altri tipi in modo tale che T diventi uguale a T'. In altri termini, T e T' devono avere almeno un'istanza comune. Nel caso precedente, sostituendo β con α sia nel tipo più generale di `fst` che in quello di `snd`, si ottiene la stessa espressione di tipo.

Quando un'espressione F è corretta solo se vale la restrizione "E e E' hanno lo stesso tipo", nel calcolare il tipo di F si considerano le istanze *più generali* dei tipi di E e E' che li rendono uguali (cioè quelle che "sostituiscono" le variabili di tipo imponendo meno restrizioni possibili).

1.4 Costruttori e selettori di un tipo

Consideriamo ora con più attenzione il tipo coppia. Per definire un tipo è necessario specificare qual è l'insieme dei valori del tipo. Nel caso dei tipi semplici, l'insieme dei valori è un insieme di *costanti* (ad esempio `true` e `false` per i booleani, l'insieme delle

stringhe per il tipo `string`, ecc.). Nel caso delle coppie non abbiamo costanti, ma solo un modo per *costruire* coppie a partire da due valori di altro tipo; in altri termini, definiamo l'insieme dei valori di tipo $t_1 \times t_2$ specificando come costruire un valore di tipo $t_1 \times t_2$ a partire da un valore E_1 di tipo t_1 ed un valore E_2 di tipo t_2: per ogni valore E_1 di tipo t_1 ed ogni valore E_2 di tipo t_2, l'espressione (E_1, E_2) è un valore di tipo $t_1 \times t_2$ Il tipo delle coppie è dunque definito specificando semplicemente un *costruttore*: la virgola, usata in forma infissa. L'espressione E_1, E_2 si può considerare sintatticamente simile a un'espressione della forma $E_1 + E_2$, cioè si può vedere la virgola come un'operatore. La differenza sta nel fatto che $E_1 + E_2$ non è un valore ma un'espressione che si può ridurre a una più semplice, mentre (E_1, E_2) è un valore se E_1 ed E_2 sono valori.

Analogamente, nel caso dei record, possiamo considerare che le parentesi graffe, racchiudenti l'opportuna sequenza di `<label> = ...`, costituiscano il costruttore di un tipo record. Ad esempio, per il tipo `studente` definito in 1.2.5, il costruttore è (scritto in notazione analoga a quella delle tuple): { `nome = ... ; matricola = ...` }.

In generale, anche le costanti di un tipo sono costruttori del tipo: costruttori costanti, anziché costruttori funzionali (operazioni). Un tipo viene dunque definito specificando l'insieme dei suoi costruttori. Come vedremo, ci sono tipi i cui costruttori includono sia costanti che operazioni.

Quando un tipo è definito mediante costruttori funzionali, è importante che siano definiti sul tipo anche dei *selettori* corrispondenti, cioè funzioni che, applicate a oggetti del tipo considerato, ne "selezionano" i componenti semplici. Nel caso delle coppie, i selettori sono `fst: 'a * 'b -> 'a`, che applicato a una coppia ne riporta il primo elemento, e `snd: 'a * 'b -> 'b`, che, applicato a una coppia ne riporta il secondo elemento.

1.5 Dichiarazioni

Una dichiarazione estende l'ambiente in cui sono valutate le espressioni. Un ambiente è, come si è detto, una collezione di legami tra variabili e valori. Con una dichiarazione, viene aggiunto un nuovo legame all'ambiente.

La forma più semplice di una dichiarazione è:

`let` *Variabile* `=` *Espressione*

Quando ad esempio si immette la dichiarazione di valore seguente:

```
# let two = 15/6;;
val two : int = 2
```

l'ambiente viene esteso con l'aggiunta di un nuovo legame tra la variabile `two` e il valore 2. Quindi:

```
# 17/two = 0;;
- : bool = false
```

L'espressione `17/two = 0` viene valutata calcolando innanzitutto il valore delle sue sottoespressioni; nell'ambiente attuale `two` ha valore 2, quindi `17/two` ha valore 8, che non è uguale a 0; di conseguenza il valore dell'espressione completa è `false`.

Si noti che le variabili non hanno un tipo fissato, ma è possibile ridichiarare una variabile con un valore di tipo diverso: ogni variabile ha il tipo del suo valore. Nonostante la terminologia, infatti, una dichiarazione di variabile in un linguaggio funzionale è più simile a una dichiarazione di costante in un linguaggio imperativo: creare un legame non è la stessa cosa di un'assegnazione. Con una dichiarazione di valore, viene "creato" un nuovo identificatore, che non ha nulla a che vedere con evenutali identificatori dichiarati in precedenza con lo stesso nome. Anche se la nuova dichiarazione può "nascondere" la visibilità della vecchia, non viene cambiato il valore del vecchio identificatore; quando un identificatore è stato legato a un valore, non è possibile in alcun modo cambiare quel valore.

L'ambiente di valutazione, cioè la collezione di legami tra variabili e valori viene gestito come uno *stack* (o pila): una nuova dichiarazione aggiunge un legame in cima alla pila, senza alterare i legami preesistenti. Possiamo rappresentare graficamente un ambiente proprio mediante una pila di coppie "variabile-valore":

var_n	val_n
var_{n-1}	val_{n-1}
...	...
var_1	val_1
ambiente del modulo Pervasives	

L'ambiente del modulo Pervasives è l'ambiente che contiene le definizioni di tutte le variabili predefinite in Ocaml. Nuove dichiarazioni aggiungono legami in cima all'ambiente: sopra è rappresentato l'ambiente che si ottiene definendo prima la variabile var_1, poi la variabile var_2, ... ed infine la variabile var_n. Quando OCaml cerca il valore di una variabile x, esamina l'ambiente a partire dall'alto: il primo legame che trova per x è quello riportato.

Ad esempio, l'ambiente creato dalle dichiarazioni seguenti:

```
# let two=2;;
val two : int = 2
# let three=two+1;;
val three : int = 3
# let two="due";;
val two : string = "due"
```

può essere rappresentato da:

two	"*due*"
three	3
two	2
ambiente del modulo Pervasives	

In questo ambiente, il valore di `two` è `"due"`: il nuovo legame nasconde il vecchio. Il ruolo di legami "nascosti" diverrà più chiaro in seguito, quando analizzeremo qualche esempio di programma.

Per dichiarare una funzione possiamo utilizzare la stessa sintassi utilizzata per la dichiarazione di variabili, in combinazione con la parola chiave `function`:

```
# let three_times = function x -> 3*x;;
val three_times : int -> int = <fun>
```

oppure utilizzare la sintassi più compatta:

`let` *Nome Argomento* `=` *Espressione*

come in:

```
# let three_times x = 3*x;;
val three_times : int -> int = <fun>
```

Si noti che non è necessario includere l'argomento tra parentesi.

Possiamo ad esempio definire i predicati `even` (pari) e `odd` (dispari) come segue:

```
# let even n = n mod 2 = 0;;
val even : int -> bool = <fun>
# let odd n = n mod 2 = 1;;
val odd : int -> bool = <fun>
```

Come evidenzia la risposta di OCaml, si tratta di funzioni da interi a booleani (predicati sugli interi).

```
# even 24;;
- : bool = true
# odd (3*8);;
- : bool = false
```

Non è necessario in OCaml includere l'argomento di una funzione tra parentesi, a meno che (come nel secondo esempio sopra riportato) l'argomento non sia un'espressione complessa; infatti normalmente nelle espressioni OCaml associa da sinistra a destra (l'espressione `odd 3 * 8` sarebbe interpretata come `(odd 3) * 8`, che non è corretta, dato che non si può moltiplicare un booleano per un intero).

Possiamo definire come segue i predicati su caratteri (funzioni di tipo `char -> bool`) che determinano se un carattere è una cifra, una lettera maiuscola o minuscola:

```
# let isdigit x = '0' <= x && x <= '9';;
val isdigit : char -> bool = <fun>
# let isupper x = 'A' <= x &&  x <= 'Z';;
val isupper : char -> bool = <fun>
# let islower x = 'a' <= x &&  x <= 'z';;
val islower : char -> bool = <fun>
```

Le operazioni di confronto `<`, `<=`, `>`, `>=` e `<>` (diverso) sono infatti definite anche sui caratteri (vedi paragrafo 1.6).

Le funzioni seguenti consentono di convertire cifre in interi (`digit: char -> int`) o caratteri minuscoli in maiuscoli (`uppercase: char -> char`), e sono ovviamente corrette soltanto se applicate ad argomenti appropriati:

```
# let digit x = Char.code x - Char.code '0';;
val digit : char -> int = <fun>
# let uppercase x =
  if islower x
      then Char.chr(Char.code 'A' + (Char.code x - Char.code 'a'))
  else x;;
val uppercase : char -> char = <fun>
```

(`uppercase` e l'inversa `lowercase` sono in realtà predefinite nel modulo `Char`).

Si è detto che la composizione di funzioni è il meccanismo di controllo fondamentale in programmazione funzionale. Esempi ancora semplicissimi sono i seguenti, in cui una funzione è applicata al suo argomento ed il valore riportato è passato come argomento a un'altra funzione:

```
# odd (digit '0');;
- : bool = false
# digit '8' + digit '9';;
- : int = 17
```

Naturalmente è possibile definire funzioni a più argomenti, nella forma:

`let` $Nome$ $(Arg_1, Arg_2, ..., Arg_n)$ `=` $Espressione$

Ad esempio:

```
# let mdiv (x,y) = if x>y then x/y else y/x;;
val mdiv : int * int -> int = <fun>
# mdiv(3*2,5*4);;
- : int = 3
```

Il tipo della funzione `mdiv` è `int` $\times$ `int -> int`: `mdiv` si applica a una coppia di interi e riporta un intero come valore. Per OCaml, di fatto, *le funzioni hanno sempre un solo argomento*; tale argomento può essere di tipo semplice, ma anche una coppia (come in questo caso), una tripla, ecc. Le parentesi che racchiudono "gli argomenti" e la virgola costituiscono in realtà l'operatore per la formazione di coppie.

1.6 Tipi con uguaglianza

Il test di uguaglianza si può effettuare su molti tipi, ma non su tutti. Ad esempio, non si può controllare l'uguaglianza tra funzioni. Infatti:

```
# fst = function (x,y) -> x;;
Exception: Invalid_argument "equal: functional value".
```

Esiste un motivo teorico per questo: è stata infatti dimostrata l'impossibilità di controllare l'uguaglianza (estensionale) tra funzioni in modo automatico. In altri termini, il problema di determinare se due funzioni sono estensionalmente uguali è *indecidibile*.

I tipi su cui si può effettuare il test di uguaglianza sono detti **tipi con uguaglianza** (o *equality types*). I tipi semplici sono tipi con uguaglianza; un tipo composito, come le tuple, è un tipo con uguaglianza se e solo se tutti i tipi componenti sono tipi con uguaglianza. Su tali tipi è sempre possibile, in OCaml, applicare gli operatori di confronto `=, <>, <, <=, >, >=`.

1.7 Uso di file

Gli esempi esposti fin qui consistono di definizioni semplici e brevi. Naturalmente, non è pensabile scrivere un programma complesso immettendo le dichiarazioni al prompt di OCaml, ma è indispensabile poter scrivere il programma su file, salvarlo, caricarlo in memoria, provarlo, modificarlo, ecc. Un programma consiste, come si è detto, di un insieme di dichiarazioni. Se è conservato in un file, si può caricare in memoria utilizzando la *direttiva* `#use`, che si applica a una stringa (con il nome di un file). Se ad esempio `prova.ml` è il nome di un file il cui contenuto consiste delle dichiarazioni delle funzione `even` e `odd`:

```
let even n = n mod 2 = 0
let odd n = n mod 2 = 1
```

l'esecuzione della direttiva

```
#use "prova.ml";;
```

avrà come effetto l'estensione dell'ambiente corrispondente a tali dichiarazioni:

```
# #use "prova.ml";;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
#
```

Si noti che nel file non è necessario terminare ogni dichiarazione con il doppio punto e virgola.

D'ora in poi, negli esempi di dichiarazioni ometteremo il "cancelletto" corrispondente al prompt di OCaml, assumendo di utilizzare un file.

Nota importante: abituatevi, da subito, a commentare bene i vostri programmi, specificando, per ogni funzione, il suo tipo e dando una breve descrizione di *cosa* calcola la funzione. Chiamiamo questo tipo di descrizione *specifica dichiarativa*. I commenti in OCaml sono racchiusi tra i caratteri `(*` e `*)`. Quindi ad esempio il nostro file con la definizione di pari e dispari dovrebbe piuttosto essere:

```
(* even: int -> int *)
(* even n = true se n e' pari, false altrimenti *)
let even n = n mod 2 = 0
```

```
(* odd: int -> int *)
(* odd n = true se n e' dispari, false altrimenti *)
let odd n = n mod 2 = 1
```

1.8 La valutazione delle funzioni

Si consideri la definizione di `even` data sopra. Per calcolare il valore della funzione applicata a un argomento, ad esempio `even (3*8)`, OCaml innanzitutto determina il valore dell'argomento, nel nostro caso di `3*8`. Il meccanismo di chiamata delle funzioni è cioè quella che si dice "chiamata per valore". L'ambiente di valutazione viene poi *provvisoriamente* esteso con l'aggiunta del legame di `n` – parametro formale nella definizione di `even` – con il valore `24` dell'argomento:

n	24
...	...
even	$function\ n \rightarrow n\ mod\ 2 = 0$
...	...
ambiente del modulo Pervasives	

In questo nuovo ambiente viene valutato il *corpo della funzione*, cioè l'espressione `n mod 2 = 0`: il valore di `n`, in tale ambiente, è `24`, quindi `n mod 2` è `0`, dunque il valore di `even (3*8)` è `true`. Nel momento in cui termina il calcolo di tale valore, il legame di `n` viene rimosso dall'ambiente che torna ad essere come era prima della chiamata di `even`:

...	...
even	$function\ n \rightarrow n\ mod\ 2 = 0$
...	...
ambiente del modulo Pervasives	

Abbiamo detto che i linguaggi della classe ML determinano a tempo di compilazione il valore delle variabili in una dichiarazione. Di conseguenza, se la dichiarazione di una funzione contiene il riferimento ad una variabile `x` (come la definizione di `sixtimes: int -> int`, sotto riportata, alla variabile `six`), quando tale funzione viene applicata a un argomento, il valore di `six` viene ricercato nell'ambiente in cui la funzione è stata definita (*ambiente di definizione*) e non in quello in cui viene applicata (*ambiente di valutazione*). Ad esempio:

```
# let six=6;;
val six : int = 6
# let sixtimes n=six*n;;
val sixtimes : int -> int = <fun>
# sixtimes 3;;
- : int = 18
# let six=50;;
```

```
val six : int = 50
# six;;
- : int = 50
# sixtimes 3;;
- : int = 18
# let six="six";;
val six : string = "six"
# sixtimes 3;;
- : int = 18
```

Quando viene definita la funzione `sixtimes` il valore di `six` nell'ambiente è 6. Di conseguenza, anche se in seguito la variabile `six` viene ridichiarata, il valore di `sixtimes` non cambia. L'ambiente in cui viene valutata l'ultima espressione `sixtimes 3` può essere rappresentato così:

six	"*six*"
six	50
sixtimes	$function\ n \rightarrow six \times n$
six	6
ambiente del modulo Pervasives	

Il valore di `six` in questo ambiente è la stringa `"six"`, ma l'ambiente in cui viene cercato il valore di `six` nel corpo di `sixtimes`, quando questa viene applicata, è quello "sotto" la definizione di `sixtimes` stessa, ed in tale ambiente il valore di `six` è 6.

L'esempio appena visto può servire anche a chiarire la differenza tra dichiarazione di un valore e assegnazione di una variabile: il primo valore di `six` (6) non viene sovrascritto dalle successive dichiarazioni; esso è ancora vivo nell'ambiente "visibile" dalla funzione `sixtimes`, anche se non è più visibile al livello principale dell'interazione.

Consideriamo ora il caso di una funzione che si applica a una coppia, come ad esempio quella seguente:

```
(* average: float * float -> float *)
(* average(x,y) = media aritmetica di x e y *)
let average (x,y) = (x+.y)/.2.0;;
```

Quando viene valutata ad esempio l'espressione `average (6.5,8.5)`, il valore dell'argomento, `(6.5,8.5)`, viene confrontato con l'espressione `(x,y)` utilizzata nella definizione per indicare il parametro formale di `average`. OCaml riconosce che le due espressioni hanno la stessa struttura: se in `(x,y)` si sostituisce `x` con `6.5` e `y` con `8.5`, si ottiene proprio `(6.5,8.5)`. L'ambiente viene allora esteso provvisoriamente con due nuovi legami: di `x` con `6.5` e di `y` con `8.5`;

x	6.5
y	8.5
...	...

in questo ambiente viene valutato il corpo della funzione `(x+.y)/.2.0` e infine viene ripristinato l'ambiente preesistente rilasciando i legami provvisori per `x` e `y`.

1.9 Pattern matching

L'operazione di confronto di un'espressione in cui possono comparire variabili (un *pattern* o espressione schematica) con un valore è chiamata *pattern matching*: se l'operazione ha successo, cioè il valore è conforme allo schema, allora si può determinare il valore che deve essere sostituito a ciascuna variabile del pattern perché lo schema sia identico al valore. Si veda il capitolo 2 del libro [1],[4] che contiene una descrizione del pattern matching in tutte le sue forme possibili.

Il caso più semplice di pattern è l'espressione costiutita da un identificatore, ad esempio `n`: ogni valore si confronta postitivamente con `n`. Nel caso del pattern `(x,y)`, ogni coppia darà esito positivo.

L'operazione di pattern matching viene eseguita anche nel caso delle dichiarazioni di valore: in generale, una dichiarazione di valore ha la forma

`let` *Pattern* `=` *Espressione*

Così, ad esempio:

```
# let (x,y) = (3*8,10<100 || false);;
val x : int = 24
val y : bool = true
```

In OCaml, un pattern è un'espressione costruita soltanto mediante variabili e *costruttori di tipo*. Per i tipi introdotti fin qui, i costruttori sono tutti e solo i valori del tipo (per `int, float, bool, char, string, unit`), i costruttori di tuple, cioè le parentesi e le virgole con la sintassi abituale, e i costruttori di record. In altri termini, ogni intero, reale, carattere o stringa può occorrere in un pattern, così come `()`, `true` e `false`, e ogni tupla di pattern è ancora un pattern. Per quel che riguarda i record, i pattern per i record hanno la forma:

$$\{\ \texttt{label}_1\ \texttt{=}\ \texttt{pattern}_1\ \texttt{;}\ \ldots.\ \texttt{;}\ \texttt{label}_n\ \texttt{=}\ \texttt{pattern}_n\ \}$$

Ad esempio, {`nome = n; matricola = m`} è un pattern di tipo `studente` (vedi paragrafo 1.2.5).

Si noti che le operazioni sui tipi numerici, sulle stringhe, booleani, ecc. non sono costruttori e non possono quindi occorrere in un pattern; così, ad esempio, `-n, not f, n + m, s1^s2` non sono pattern. Un'ulteriore osservazione in proposito: **in un pattern una stessa variabile non può occorrere più di una volta**, ad eccezione della variabile muta, di cui parleremo tra poco. Dunque, ad esempio `(n,n)` non è un pattern ammesso nel linguaggio. Se ad esempio proviamo a valutare la dichiarazione:

```
let (n,n) = (3,3)
```

si ottiene il messaggio d'errore `Variable n is bound several times in this matching`.

[4] `https://caml.inria.fr/pub/docs/oreilly-book/html/book-ora016.html`

1.9.1 Espressioni match

La definizione di una funzione può sfruttare il *pattern matching* in modo più forte. Ad esempio, possiamo definire l'OR esclusivo come segue:

```
(* xor : bool * bool -> bool *)
(* xor(p,q) = true se e solo se esattamente uno tra p e q e' vero *)
let xor (p,q) = if p then not q else q
```

La definizione di `xor` distingue due casi, a seconda del valore del primo argomento. I due casi possono essere identificati mediante l'uso di pattern, anziché esplicitamente dall'espressione condizionale:

```
let xor (p,q) = match p with
    true -> not q
  | false -> q;;
```

Il corpo della funzione contiene un'operazione esplicita di pattern matching. Si può leggere come segue: "se il valore di `p` è conforme al pattern `true`, allora il valore della funzione è il valore di `not q`, altrimenti, se il valore di `p` è conforme al pattern `false`, il valore della funzione è quello di `q`". L'espressione `match p with ...` contiene due clausole, separate dalla barretta verticale.

La forma generale delle espressioni `match` è:

```
match E with
          P1 -> E1
        | P2 -> E2
        | ...
        | Pk -> Ek
```

Trattandosi di un'espressione, una struttura di questa forma ha sempre un tipo e un valore. Dato che OCaml è a tipizzazione statica, ci sono delle restrizioni nel tipo delle sottoespressioni (come nel caso delle espressioni condizionali):

- il tipo di E e di tutti i pattern $P_1, \ldots, P_n$ deve essere lo stesso;
- il tipo delle espressioni $E_1, \ldots, E_n$ deve essere lo stesso.

Quando OCaml valuta un'espressione esamina le clausole a partire dall'alto, confrontando il valore v di E con i pattern P_1, P_2, ... fino a trovare una clausola applicabile, cioè un pattern il cui confronto con v dà esito postivo. Nel semplice caso della definizione di `xor`, la prima clausola è applicabile solo se l'argomento `p` ha valore `true`; per ogni altro valore (`false`) si applicherà la seconda clausola.

Quindi, quando OCaml valuta un'espressione della forma `xor (`E_1`,`E_2`)` calcola innanzitutto i valori v_1 e v_2 di E_1 ed E_2, rispettivamente, e confronta v_1 con il pattern `true`; se tale confronto ha esito positivo (se v_1 = `true`), allora viene riportato l'opposto del valore di E_2 (*not* v_2). Altrimenti confronta v_1 con il pattern `false`; tale confronto ha sicuramente successo e viene quindi riportato il valore v_2.

Poiché il secondo caso si verifica ogniqualvolta non si verifica il primo (l'unico altro valore booleano è `false`), possiamo utilizzare come pattern per il secondo caso la *variabile muta* (o *anonima* o *dummy*) "_" e definire `xor` così:

```
let xor (p,q) = match p with
    true -> not q
  | _ -> q;;
```

Attenzione: la variabile muta si può utilizzare soltanto nei pattern e non nelle espressioni. Un'operazione di pattern matching con la variabile muta ha sempre successo. Possiamo poi costruire pattern più complessi in cui occorre la variabile muta, come ad esempio `(_,3)`, che è conforme a tutte le coppie il cui secondo elemento è `3`. Analogamente, il pattern `(_,_,_)` è conforme a tutte le triple di oggetti (si noti che la variabile muta può occorrere più volte in un pattern, ma due occorrenze diverse sono come due variabili diverse).

In maggior dettaglio, un'espressione `match` della forma

```
match E with
            P1 -> E1
        |   P2 -> E2
        |   ...
        |   Pk -> Ek
```

viene valutata come segue. Innanzitutto viene calcolato il valore v di E e tale valore viene confrontato, nell'ordine, con i pattern $P_1, P_2, ...$ Se nessun confronto dà esito positivo, la valutazione dell'espressione risulta in un errore (l'*eccezione* `Match_failure`); altrimenti, se P_i è il primo pattern il cui confronto con v dà esito postivo, l'ambiente viene esteso legando ciascuna variabile nel pattern con il corrispondente valore in v. Ad esempio se il pattern è `(x,y)` e v = `(3,true)`, l'ambiente viene esteso legando `x` a `3` e `y` a `true`. Si noti che se la variabile muta occorre nel pattern, non viene creato alcun legame ad essa corrispondente. Nell'ambiente così esteso viene calcolato il valore di E_i, l'espressione corrispondente al pattern P_i, che sarà il valore di tutta l'espressione `match`. Quando tale valore è stato infine calcolato, viene ripristinato l'ambiente precedente.

1.9.2 Espressioni function

Un modo alternativo di definire le funzioni mediante pattern matching è quello di utilizzare la forma generale delle espressioni `function`:

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

dove $E_1, ..., E_n$ sono espressioni e $P_1, ..., P_n$ sono pattern. Quando una tale funzione viene applicata a un argomento E, la valutazione avviene confrontando il valore di E

con $P_1, ..., P_n$, nell'ordine dato. Se P_i è il primo pattern cui il valore di E è conforme, viene valutata l'espressione E_i nell'ambiente che si ottiene legando ciascuna variabile nel pattern con il corrispondente valore in E. Il valore ottenuto viene riportato come valore dell'applicazione e l'ambiente ritorna come era prima.

Analogamente al caso delle espressioni `match`, un'espressione `function` della forma precedente è corretta solo se:

- il tipo di tutti i pattern $P_1, \ldots, P_n$ deve essere lo stesso;
- il tipo delle espressioni $E_1, \ldots, E_n$ deve essere lo stesso.

Quindi possiamo definire la funzione `xor` anche come segue:

```
let xor = function
   (true,q) -> not q
  | (_,q) -> q;;
```

Quando `xor` viene applicata a un argomento E, viene innanzitutto calcolato il valore di E, che sarà necessariamente una coppia di booleani (b_1, b_2). Tale valore viene confrontato con il pattern `(true,q)` e se tale confronto dà esito positivo, l'ambiente viene esteso aggiungendo il legame di `q` con b_2; in tale ambiente viene valutata l'espressione `not q` e il valore ottenuto, che sarà uguale a *not* b_2, è il valore riportato da `xor`. Altrimenti, (b_1, b_2) viene confrontato con il pattern `(_,q)`; il confronto dà necessariamente esito positivo e viene quindi riportato il valore b_2 di `q`.

1.10 Definizioni ricorsive di funzioni

Come si è detto, la ricorsione è il principale meccanismo di controllo nei linguaggi funzionali. La definizione ricorsiva di una funzione è introdotta in Caml dalle parole chiavi `let rec` anziché soltanto `let`. Ad esempio, la funzione fattoriale si può definire in Caml come segue:

```
(* fact: int -> int *)
(* fact n = fattoriale di n, definita soltanto su
   interi non negativi *)
let rec fact n =
  if n=0
  then 1
  else n * fact(n-1)
```

oppure, usando il pattern matching, nel modo seguente:

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

Si noti che `fact` è definita soltanto per gli interi non negativi: la valutazione di `fact (-1)`, ad esempio, non termina.

Le funzioni di base sui numeri naturali (come somma, prodotto, elevamento a potenza, ecc.), se non fossero predefinite in OCaml, potrebbero essere definite ricorsivamente utilizzando soltanto le funzioni `succ` (successore) e `pred` (predecessore):

```
(* sum: int * int -> int *)
(* sum (n,m) = n+m *)
let rec sum = function
    (n,0) -> n
  | (n,m) -> succ(sum(n,pred m))

(* times: int * int -> int *)
(* times (n,m) = n*m *)
let rec times = function
    (n,0) -> n
  | (n,m) -> sum(n,times(n,pred m))

(* power: int * int -> int *)
(* times (n,m) = m-esima potenza di n *)
let rec power = function
    (n,0) -> 1
  | (n,m) -> times(n,power(n,pred m))
```

Un altro esempio è dato dalla funzione `stringcopy: string * int -> string`, che, applicata a una coppia `(s,n)`, dove `s` è una stringa e `n` un intero non negativo, riporta la stringa che risulta dalla replicazione di `s` n volte.

```
(* stringcopy : string * int -> string *)
(* stringcopy (s,n) = s ^ s ^ .... ^ s (n volte) *)
let rec stringcopy (s,n) =
   if n <= 0 then ""
   else s^stringcopy(s,n-1)

# stringcopy ("ma",5);;
- : string = "mamamamama"
```

Un ulteriore esempio di definizione ricorsiva è quella del massimo comun divisore di due interi, che abbiamo già visto:

```
(* gcd: int * int -> int *)
(* gcd (n,m) = massimo comun divisore di n e m,
   dove n e m sono interi non negativi e non entrambi nulli *)
let rec gcd = function
    (n,0) -> n
  | (n,m) -> gcd(m,n mod m)
```

1.10.1 Ricorsione e iterazione

La ricorsione in un linguaggio funzionale sostituisce l'iterazione. Consideriamo ad esempio il semplice algoritmo iterativo seguente per calcolare il fattoriale di un intero non negativo:

```
per calcolare il fattoriale di n:
        porre f = 1
        finche' n > 0:
              { porre f = f*n;
                porre n = n-1 }
        riportare f
```

Questo algoritmo consiste di un ciclo, in cui sono coinvolte due variabili, `f` e `n`; la prima è inizializzata a 1, la seconda all'argomento stesso del fattoriale. Il ciclo "while" può essere implementato da una funzione ausiliaria con due argomenti che si richiama ricorsivamente. Nel programma seguente, la funzione `fact_aux` implementa il ciclo, la funzione principale, `fact_it`, richiama `facti` con i parametri iniziali corretti, cioè compie l'inizializzazione e avvia il ciclo.

```
let rec fact_aux (n,acc) =
  if n=0 then acc             (* il ciclo termina *)
  else fact_aux (n-1,n*acc) (* si continua con l'iterazione
                                successiva *)

(* oppure, usando un'espressione match: *)
let rec fact_aux (n,acc) =
  match n with
    0 -> acc
  | _ -> fact_aux (n-1,n*acc)

(* fact_it: int -> int *)
(* fact_it n = fattoriale di n, definita soltanto su
   interi non negativi *)
let fact_it n =
  fact_aux (n,1)            (* inizializzazione del ciclo *)
```

La funzione `fact_aux` è definita ricorsivamente, ma il processo della sua esecuzione è un processo *iterativo*.

1.10.2 Processi ricorsivi e iterativi: *tail recursion*

Consideriamo la funzione `fact` definita a pagina 29 ed osserviamo il processo generato da una sua applicazione:

```
fact 3 ==> 3 * fact 2
       ==> 3 * (2 * fact 1)
```

```
==> 3 * (2 * (1 * fact 0))
==> 3 * (2 * (1 * 1))
==> 3 * (2 * 1)
==> 3 * 2
==> 6
```

Come si vede, il calcolo delle chiamate ricorsive intermedie resta in sospeso finché non si ottiene il risultato dalla chiamata successiva. Ad esempio, per calcolare `fact 2` si deve aspettare di ottenere il risultato di `fact 1`; solo a questo punto si può eseguire la moltiplicazione finale (`2 * fact 1`) e riportare il risultato di `fact 2`. Dal punto di vista dell'implementazione del processo, questo significa che è necessario conservare in memoria i "record di attivazione" di tutte le chiamate ricorsive: il primo di essi contiene il legame del parametro formale `n` con il parametro attuale `3`, richiama `fact 2` e resta in attesa del risultato riportato da questa chiamata per moltiplicarlo infine per 3. Il secondo record di attivazione riguarda la chiamata `fact 2`; questo calcolo viene avviato invocando `fact 1` e restando in attesa del risultato fornito per moltiplicarlo per 2. E così via, fino a che la chiamata di base `fact 0` non riporta il suo risultato al "chiamante", e tutti i calcoli lasciati in sospeso possono finalmente essere terminati.

Dal punto di vista dello spazio di memoria occupato da questo processo, osserviamo che esso è proporzionale a `n`:

n	0
n	1
n	2
n	3
$fact$	$function\ n \rightarrow \ldots$
$\ldots$	$\ldots$

Questa "catena" di calcoli lasciati in sospeso può essere evitata sfruttando la proprietà associativa della moltiplicazione:

```
3 * (2 * fact 1) = (3 * 2) * fact 1
```

Possiamo calcolare subito `3 * 2` e conservarlo in un "accumulatore". Da questa osservazione deriva la versione "iterativa" del fattoriale, `fact_it`, definita a pagina 31. Osserviamo il processo di esecuzione di un calcolo di `fact_it`:

```
fact_it 3 = fact_aux(3,1)
          = fact_aux(2,3)
          = fact_aux(1,6)
          = fact_aux(0,6)
          = 6
```

Come si vede, il processo è "lineare": dopo aver raccolto il risultato della chiamata ricorsiva, non si deve fare nulla. Il risultato della prima chiamata ricorsiva di `fact_aux` (`fact_aux(3,1)`) è lo stesso che viene calcolato e riportato dall'ultima chiamata (`fact_aux(0,6)`). Quest'ultimo risultato, 6, viene di fatto riportato da tutte le chiamate intermedie come risultato, senza l'esecuzione di ulteriori calcoli.

Di fatto, in questo caso, non è necessario conservare i dati relativi a tutte le chiamate intermedie della funzione. Molti compilatori riconoscono questa forma "fittizia" di ricorsione, chimata *tail recursion* (ricorsione di coda), e ottimizzano l'occupazione di memoria nel caso di processi iterativi.

In generale, una funzione `tail recursive` ha questa caratteristica: per calcolare un suo valore, se non si è in un caso di base, vengono eseguiti alcuni calcoli, poi viene effettuata una chiamata ricorsiva alla funzione ed il risultato di tale chiamata ricorsiva è il risultato riportato dalla funzione stessa senza l'esecuzione di ulteriori calcoli. In altri termini, non è necessario eseguire alcun calcolo al *ritorno dalla ricorsione*.

Possiamo caratterizzare le funzioni *tail recursive* in un modo diverso. Quando un problema P1 viene convertito in un altro P2, in modo che la soluzione di P2 è identica alla soluzione di P1 (non servono altri calcoli), allora si dice che P1 è stato *ridotto* a P2, o che P2 è una *riduzione* di P1. Quando una funzione ricorsiva è definita in modo tale che tutte le chiamate ricorsive sono riduzioni, allora la funzione è *tail recursive*.

In sintesi, possiamo distinguere due tipologie di processi di calcolo:

- Un processo ricorsivo:
  1. può eseguire calcoli al ritorno dalla ricorsione
  2. usa *spazio* proporzionale alla dimensione del suo input.
- In un processo iterativo
  1. il risultato parziale viene conservato in un "accumulatore";
  2. il processo è lineare;
  3. dopo aver raccolto il risultato della chiamata ricorsiva non si deve fare nulla;
  4. l'ultima chiamata può riportare il suo risultato direttamente alla prima.

Il calcolo di una funzione *tail recursive* è allora un processo iterativo e non ricorsivo.

1.11 Dichiarazioni locali

Supponiamo di voler calcolare il quadrato del fattoriale di 3. Avendo definito la funzione fattoriale, possiamo valutare l'espressione:

```
# fact 3 * fact 3;;
- : int = 36
```

La valutazione di questa espressione, tuttavia, richiede di calcolare due volte il fattoriale di 3. Ciò si può evitare ricorrendo alla valutazione dell'espressione seguente:

```
# let n = fact 3 in n*n;;
- : int = 36
```

L'espressione utilizza una *dichiarazione locale* e si può leggere così: "sia n il fattoriale di 3 nell'espressione $n \times n$". Essa viene valutata calcolando il valore di `fact 3`, estendendo temporaneamente l'ambiente con il legame di `n` con il valore ottenuto e valutando nel nuovo ambiente l'espressione `n*n`. Questo è il valore di tutta l'espressione. Quando esso è stato calcolato, il legame per `n` viene rimosso e l'ambiente ritorna quello che era in precedenza.

La forma generale di una dichiarazione locale è la seguente:

Dichiarazione `in` *Espressione*

Si tratta di un'espressione e come tale ha un tipo e un valore: il tipo di *Dichiarazione* `in` *Espressione* è quello di *Espressione* e il suo valore in un ambiente A è il valore che ha *Espressione* nell'ambiente che estende A secondo *Dichiarazione*. In altri termini, per valutare *Dichiarazione* `in` *Espressione*, OCaml estende provvisoriamente l'ambiente aggiungendo i legami dettati da *Dichiarazione*; in tale ambiente esteso valuta *Espressione*, poi viene ripristinato l'ambiente preesistente.

Di conseguenza, dopo la valutazione di un'espressione della forma `let n = ... in ...`, la variabile `n` torna ad assumere lo stesso valore che aveva prima (o nessun valore):

```
# let n = 3 in if n<10 then 1 else 2;;
- : int = 1
# n;;
Characters 0-1:
  n;;
  ^
Error: Unbound value n
```

Poiché una dichiarazione inizia con `let`, le espressioni della forma *Dichiarazione* `in` *Espressione* vengono chiamate "espressioni `let`".

Le dichiarazioni locali sono utili anche quando un identificatore ha un uso limitato e non ha significato al di fuori di un'espressione più ampia che lo utilizza. La definizione di funzioni utilizza infatti spesso dichiarazioni locali. Nella definizione del fattoriale iterativo, ad esempio, possiamo definire localmente la funzione ausiliaria, che non ha significato autonomo, e, dato che non è visibile all'esterno, darle un nome più breve:

```
(* fact_it: int -> int *)
(* fact_it n = fattoriale di n, definita soltanto su
   interi non negativi *)
let fact_it n =
  (* aux : int * int -> int *)
  (* aux (n,acc) = acc * fattoriale di n *)
  let rec aux (n,acc) =
    match n with
      0 -> acc
    | _ -> aux (n-1,n*acc)
  in aux (n,1)
```

Naturalmente è buona regola commentare anche le funzioni locali (come nell'esempio qui sopra).

Si noti che, mentre è importante dal punto di vista dello stile di programmazione dare nomi significativi alle funzioni visibili a *top level* (cioè al livello principale dell'interazione), il nome di una funzione definita localmente può anche essere poco significativo (come nell'esempio precedente).

1.12 Questioni di stile: quando utilizzare dichiarazioni locali

Lo stile di programmazione indotto dai linguaggi imperativi porta spesso alla definizione di un'unica "grande" procedura, all'interno della quale sono definite localmente eventuali procedure ausiliarie. In programmazione funzionale, al contrario, si preferisce definire funzioni localmente soltanto quando:

- le funzioni non hanno alcun significato autonomo, al di fuori del contesto in cui sono utilizzate,

e/o

- la definizione locale consente di "risparmiare" parametri (soprattutto quando questi corrispondo a strutture dati complesse).

Ad esempio, utilizzando la funzione `gcd` definita a pagina 7 e rappresentando le frazioni mediante coppie (numeratore,denominatore), possiamo definire come segue una funzione che, applicata a una frazione, riporti la stessa frazione ridotta ai minimi termini:

```
(* fraction : int * int -> int * int *)
(* fraction (n,d) = (n1,d1), dove (n1,d1) rappresenta la stessa
                frazione (n,d) ridotta ai minimi termini *)
 let fraction (n,d) =
    let com = gcd (n,d)
    in  (n/com, d/com)
```

In questo caso, dato che la funzione `gcd` ha un significato "autonomo", non avrebbe senso definirla localmente a `fraction`.

Anche per quel che riguarda la definizione locale di valori, in programmazione funzionale si tende a "fare più economia", rispetto all'uso che normalmente si fa dell'assegnazione in programmazione imperativa. In generale, si definisce localmente un valore soltanto quando ciò consente di evitare di valutare più volte la stessa espressione. Ad esempio, lo stile indotto dalla programmazione imperativa porterebbe a definire la funzione `fraction` in questo modo:

```
let fraction (n,d) =
    let com = gcd (n,d)
    in let n1 = n/com
    in let d1 = d/com
    in (n1,d1)
```

Ma, mentre la definizione locale di `com` consente di evitare di calcolare due volte il valore di `gcd(n,d)`, ed è quindi non solo giustificata ma anche fortemente consigliata, non c'è motivo di definire localmente le variabili `n1` e `d1`.

1.13 Programmi

Fin qui abbiamo definito funzioni diverse, apparentemente scollegate le une dalle altre. Con il prossimo esempio mostriamo come una collezione di funzioni possa costituire un programma completo, per calcolare la radice quadrata di un numero reale con il metodo di Newton (l'esempio è ripreso da [5]). Per calcolare la radice di a, si sceglie un qualsiasi numero x_0 positivo, ad esempio 1, come prima approssimazione. Poi si esegue un'iterazione in cui, se x è l'approssimazione attuale, la successiva approssimazione considerata sarà $(a/x + x)/2$ e si termina non appena la differenza $a - x^2$ diviene sufficientemente piccola (ad esempio, minore di 0.001). Il programma che segue è costituito da quattro funzioni:

- il predicato `okp: float * float -> bool` determina l'accettabilità di un risultato x come sufficiente approssimazione per la radice quadrata di a;
- la funzione `next: float * float -> float` riporta la successiva approssimazione;
- la funzione `findroot: float * float -> float` implementa l'iterazione: applicata ad a e un'approssimazione corrente x, riporta un'approssimazione accettabile della radice quadrata di a;
- infine, `sqroot: float -> float` è la funzione principale.

```
(* epsilon e' una variabile globale, che svolge il ruolo di una
   costante *)
let epsilon = 0.001

(* okp: float * float -> bool *)
(* okp(x,a) = true se la differenza tra a e il quadrato di x
              e' minore di epsilon *)
let okp(x,a) =
   abs_float(a -. x *. x) < epsilon
```

```
(* next: float * float -> float *)
(* next(x,a) = approssimazione successiva a x per la radice
               quadrata di a *)
let next(x,a) =
  (a /. x +. x) /. 2.0


(* findroot: float * float -> float *)
(* findroot(x,a) calcola iterativamente
        un'approssimazione accettabile della radice quadrata di a,
        a partire dall'approssimazione x *)
let rec findroot(a,x) =
  let newval = next(x,a) in
  if okp(newval,a)
  then newval
  else findroot(a,newval)

(* sqroot: float -> float *)
(* sqroot a = approssimazione accettabile della radice di a *)
let sqroot a =
  findroot(a,1.0)
```

Il programma viene eseguito chiamando la funzione principale:

```
# sqroot 10.0;;
- : float = 3.16227766517567499
```

(Per curiosità, provate a confrontare i valori che si ottengono con la funzione predefinita `sqrt: float -> float`). Si noti che `findroot` utilizza una dichiarazione di valore locale: per evitare che il valore di `next(x,a)` venga calcolato due volte, si utilizza la variabile locale `newval`.

Naturalmente, possiamo rendere locali le dichiarazioni delle funzioni ausiliarie. Questo caso – benché semplice – è un esempio in cui la scelta non è solo una questione di stile: il parametro `a` viene passato da una fuzione all'altra e ad ogni chiamata ricorsiva di `findroot` senza mai essere modificato. Esso può essere reso "globale" rispetto a `findroot`, guadagnando in chiarezza ed efficienza. Il codice completo si trova nella figura 1.1. Si noti la sequenza di espressioni `let` una dentro l'altra. Per non appesantire il codice, tutti i commenti, anche quelli relativi alle funzioni ausiliarie, precedono la definizione della funzione principale.

1.13.1 Le torri di Hanoi

Tutte le funzioni che abbiamo visto fin qui si possono scrivere agilmente in qualsiasi linguaggio imperativo utilizzando strutture iterative anziché la ricorsione. Concludiamo questo paragrafo presentando un problema che difficilmente si presta a una

```
(* sqroot: float -> float *)
(* sqroot a = approssimazione accettabile della radice di a *)
(* okp:  float -> bool *)
(* okp x = true se la differenza tra a e il quadrato di x
          e' minore di epsilon *)
(* next: float  -> bool *)
(* next x = approssimazione successiva a x per la radice
            quadrata di a *)
(* findroot:  float -> float *)
(* findroot x calcola iterativamente
        un'approssimazione accettabile della radice quadrata di a,
        a partire dall'approssimazione x *)
let sqroot a =
  let epsilon = 0.001 in
  let okp x =
    abs_float(a -. x *. x) < epsilon in
  let next x =
    (a /. x +. x) /. 2.0 in
  let rec findroot x =
    let newval = next x in
    if okp newval
    then newval
    else findroot newval
  in findroot 1.0
```

Figura 1.1: Radice quadrata di un reale secondo il metodo di Newton

soluzione iterativa e si risolve invece molto facilmente utilizzando la ricorsione: il gioco delle torri di Hanoi consiste in una base di legno sulla quale sono fissati tre pioli. Sui pioli si possono infilare dei dischi, tutti di diametro diverso. La sola regola del gioco è che non si può mai porre un disco sopra un altro di diametro minore. All'inizio del gioco i dischi sono tutti sul piolo di sinistra; il gioco consiste nello spostare tutti i dischi sul piolo di destra, usando quello centrale come piolo di appoggio, spostando un disco alla volta e senza violare la regola del gioco.

L'approccio ricorsivo consente di rendere il problema molto semplice: per spostare n dischi dal piolo X al piolo Y usando Z come appoggio:

Base: $n = 0$. In questo caso non si deve spostare nessun disco.

Caso ricorsivo: $n > 0$. Si assume (ipotesi della ricorsione) di saper spostare $n - 1$ dischi. Allora: si spostano $n - 1$ dischi da X a Z usando Y come appoggio; poi si sposta il disco rimanente da X a Y, ed infine si spostano gli $n - 1$ dischi da Z a Y usando X come appoggio.

Vogliamo scrivere una funzione che, dato un numero n, stampi la sequenza di mosse che occorre fare per spostare n dischi dal piolo chiamato "A" al piolo chiamato "B" – utilizzando un terzo piolo "C" come piolo d'appoggio.

Un'operazione di stampa è un'operazione che ha (evidentemente) effetti collaterali ed è quindi estranea ad un linguaggio funzionale puro. D'altronde è evidente che nessun linguaggio serio può fare a meno di operazioni di lettura e scrittura. In OCaml esistono diverse funzioni di stampa, per i tipi semplici `int`, `string`, `char` e `float`: rispettivamente `print_int`, `print_string`, `print_char` e `print_float`. Tali funzioni si applicano a oggetti del tipo voluto e riportano `()`, cioè l'unico oggetto di tipo `unit` (si veda il manuale di riferimento di OCaml [3] per le caratteristiche imperative di OCaml).

```
# print_string;;
- : string -> unit = <fun>
```

Come effetto collaterale delle operazioni, viene stampato il valore dell'argomento.

```
#  print_string "pippo\n";;
pippo
- : unit = ()
```

La mossa che consiste nello spostare il disco superiore del piolo X sul piolo Y è rappresentata dalla stringa `"Sposto un disco da X a Y"`. La funzione ausiliaria `aux`, dato n e il nome di tre pioli, X, Y e Z, riporta la stringa che rappresenta la sequenza di mosse per spostare n dischi da X a Y usando Z come appoggio. Infine, la funzione principale `hanoi`, applicata a n, richiama `hanoiaux` e stampa il risultato.

```
(* move : string * string -> string *)
(* move(x,y) riporta la stringa "Sposto un disco da x a y\n" *)
let move (x,y) =
  "Sposto un disco da "^x^" a "^y^"\n"

(* hanoi : int -> unit *)
(* hanoi n risolve il problema delle torri di hanoi con n dischi *)
(* aux: int * string * string * string -> string *)
(* aux (n,inizio,fine,tmp) riporta la stringa che rappresenta
         le mosse da fare per spostare n dischi dal piolo inizio al
         piolo fine, usando tmp come piolo di appoggio *)
let hanoi n =
  let rec aux (n,inizio,fine,tmp) =
    match n with
      0 -> ""
    | _ -> aux(n-1,inizio,tmp,fine)^
   (move(inizio,fine)) ^ (aux(n-1,tmp,fine,inizio))
  in print_string (aux(n,"A","B","C"))
```

Come esempio di esecuzione:

```
# hanoi 3;;
Sposto un disco da A a B
Sposto un disco da A a C
Sposto un disco da B a C
Sposto un disco da A a B
Sposto un disco da C a A
Sposto un disco da C a B
Sposto un disco da A a B
- : unit = ()
```

1.14 Eccezioni

Spesso una funzione non è definita per tutti i suoi argomenti. Un esempio è la funzione `fact: int -> int` definita a pagina 29, che è una funzione *parziale* sugli interi. Non è però desiderabile, in genere, che la valutazione di una funzione con un argomento per cui è indefinita non termini o dia un errore generico; sarebbe più conveniente che il processo di valutazione terminasse comunque, eventualmente segnalando un errore specifico, che consenta al programmatore di identificare il punto in cui si verifica.

A questo scopo OCaml dispone di un tipo di dati particolare, le *eccezioni*. Le eccezioni costituiscono un tipo di dati atipico da diversi punti di vista. Innanzitutto, un'eccezione può essere argomento e valore di qualsiasi funzione, indipendentemente dal tipo della funzione stessa. In secondo luogo, l'insieme dei valori del tipo delle eccezioni può essere esteso dal programmatore, mediante la *dichiarazione* di eccezioni. La forma più semplice di una dichiarazione di eccezione è:

```
exception <NOME>
```

Con una tale dichiarazione, un nuovo valore viene aggiunto all'insieme delle eccezioni, quello identificato dal `<NOME>`.

Per far sì che una funzione riporti un'eccezione, occorre "sollevare" l'eccezione, mediante il costrutto `raise <ECCEZIONE>`. Ad esempio, la dichiarazione di `fact` può essere riscritta in modo da tener conto di argomenti negativi, utilizzando un'eccezione:

```
exception FactError

(* fact: int -> int *)
(* fact n = fattoriale di n, se n non e' negativo,
            altrimenti viene sollevata l'eccezione FactError *)
let rec fact = function
    0 -> 1
  | n ->
      if n<0
      then raise FactError
      else n * fact(n-1)
```

O meglio, per evitare di controllare se n è negativo ad ogni chiamata ricorsiva, si può utilizzare una funzione ausiliaria ed eseguire il test inizialmente una volta sola:

```
let fact n =
  if n<0 then raise FactError
  else
    let rec aux = function
        0 -> 1
      | n -> n * aux (n-1)
    in aux n
```

La parola chiave `exception` è utilizzata per dichiarare un'eccezione; nel nostro caso introduciamo l'identificatore `FactError`,[5] che servirà a segnalare gli errori generati dalla valutazione di `fact` su argomenti negativi. La nuova dichiarazione di `fact` include un caso ulteriore: se l'argomento è negativo viene "sollevata" l'eccezione `FactError`. In parole semplici, la funzione riporta un valore di tipo "eccezionale" che serve a segnalare l'errore:

```
# fact 6;;
- : int = 720
# fact (-3);;
Exception: FactError.
```

Nella definizione data sopra di `fact` si potrebbe essere tentati di distinguere il caso in cui l'agomento è negativo mediante un ulteriore "pattern" che identifichi i numeri negativi; ma ciò non è possibile perché `-n` non è un pattern (il segno `-` non è un costruttore). Dunque questo caso va trattato esplicitamente mediante un'espressione condizionale.

Le eccezioni hanno altre particolarità. Innanzitutto si propagano automaticamente (se l'argomento di una funzione f è un'espressione il cui valore risulta "eccezionale", allora f stessa riporta tale valore "eccezionale", senza eseguire calcoli ulteriori):

```
# 6 * (15 + fact (-3));;
Exception: FactError.
```

La propagazione automatica delle eccezioni può essere interrotta da un *exception handler*, che fornisce la possibilità di "catturare" un'eccezione, specificando che, se una determinata sottoespressione E ha un valore "normale" allora si riporta tale valore, altrimenti, se la valutazione di E solleva un'eccezione, allora si deve riportare un altro valore. Per il momento, consideriamo le più semplici espressioni con *exception handler* della forma seguente:

`try` *E* `with` *P* `->` *F*

[5] Attenzione: OCaml è sensibile alla differenza tra lettere maiuscole e minuscole. In particolare, richiede che i nomi delle eccezioni inizino con lettera maiuscola. Al contrario, gli altri identificatori non devono iniziare con lettera maiuscola.

In questa espressione *E* e *F* sono espressioni dello stesso tipo e `P` è un pattern per eccezioni (come caso particolare, `P` è il nome di un'eccezione). L'espressione `try` *E* `with` *P* `->` *F* ha lo stesso tipo di *E* e *F* e viene valutata come segue. Innanzitutto viene valutata *E*. Se tale valutazione non solleva eccezioni, il valore di tutta l'espressione `try` *E* `with` *P* `->` *F* è il valore di *E*; altrimenti, se la valutazione di *E* solleva un'eccezione conforme al pattern *P* (l'eccezione *P*, se *P* è il nome di un'eccezione), allora il valore di tutta l'espressione è quello di *F*. Infine, nel caso in cui la valutazione di *E* sollevi un'eccezione *Ex* diversa da *P* (o non conforme al pattern *P*), tale eccezione viene riportata dall'intera espressione.

```
# try 6 * (15 + fact (-3)) with FactError -> 0;;
- : int = 0
# try (6 * (15 + fact 3)) with FactError -> 0;;
- : int = 126
# try fact 6 / 0 with FactError -> 0;;
Exception: Division_by_zero.
```

Nel primo caso l'eccezione `FactError` è catturata mediante l'espressione `try ... with FactError -> 0`. Nel secondo caso, poiché l'espressione `(6 * (15 + fact 3))` riporta un valore "normale", questo risulta il valore di tutta l'espressione `try ... with ...`. Nell'ultimo caso, il calcolo del valore di `fact 6 / 0` solleva un'eccezione diversa da `FactError`, che quindi si propaga come valore di tutta l'espressione `try ... with ...`.

Come si vede da questo esempio, il linguaggio contiene delle eccezioni predefinite, come `Division_by_zero`, che segnala il tentativo di effettuare una divisione per 0. Si veda il capitolo 23.1 del manuale di riferimento di OCaml [3] (`http://caml.inria.fr/pub/docs/manual-ocaml/core.html#sec525`). Un'utile eccezione predefinita è l'eccezione che sul manuale trovate descritta in questo modo:

```
exception Failure of string
```

Il modo in cui questa eccezione è definita sarà chiaro quando si vedrà come definire nuovi tipi (nel capitolo 3.1). In sostanza, `Failure` non è un'eccezione, ma un *costruttore di eccezioni*: una funzione che, applicata a una stringa, riporta un'eccezione. Per evitare di definire molte eccezioni diverse, si possono utilizzare eccezioni della forma `Failure <STRINGA>`, dove `<STRINGA>` è una stringa che fornisce informazioni sull'errore generato dalla valutazione. Ad esempio, invece di definire l'eccezione `FactError`, potremmo definire il fattoriale così:

```
let fact n =
  if n<0 then raise (Failure "fact")
  else
    let rec aux = function
        0 -> 1
      | n -> n * aux (n-1)
    in aux n
```

```
# fact(-1);;
Exception: Failure "fact".
```

Lo stesso effetto si ottiene utilizzando la funzione predefinita

```
failwith: string -> 'a
```

che trovate descritta nel capitolo sul modulo `Pervasives` del manuale di riferimento di OCaml (`http://caml.inria.fr/pub/docs/manual-ocaml/libref/Pervasives.html`): un'espressione della forma `failwith <STRINGA>` è equivalente a `raise (Failure <STRINGA>)`.

1.15 Funzioni di ordine superiore

Un'importante caratteristica che contraddistingue un linguaggio funzionale – rispetto ad altri linguaggi che consentono anch'essi di definire funzioni e applicarle ad argomenti – è il fatto che le funzioni sono *oggetti di prima classe*; esse possono cioè essere trattate come dati. In particolare:

1. una funzione può essere una componente di una struttura di dati;
2. una funzione può essere un argomento di un'altra funzione;
3. una funzione può essere un valore riportato da un'altra funzione;

Come esempio del punto 1, Ocaml ammette la definizione di coppie contenenti funzioni. Ad esempio, se sono state definite le funzioni:

```
# let double x = x * 2;;
val double : int -> int = <fun>
# let treble x = 3 * x;;
val treble : int -> int = <fun>
```

si può formare una coppia contenente le due funzioni:

```
# let coppia = (double,treble);;
val coppia : (int -> int) * (int -> int) = (<fun>, <fun>)
```

Il valore di `coppia` è una coppia (di tipo $\alpha \times \beta$), i cui elementi sono funzioni ($\alpha = \beta =$`int`$\rightarrow$`int`).

Una funzione che si applica a funzioni o che produce funzioni come valori viene detta *di ordine superiore*. Ad esempio, la funzione `apply`, sotto definita, è una funzione di ordine superiore che si applica a una coppia, il cui primo argomento è una funzione:

```
# let apply (f,x) = f x;;
val apply : ('a -> 'b) * 'a -> 'b = <fun>
# let y = (double,7);;
val y : (int -> int) * int = <fun>, 7
# apply y;;
- : int = 14
```

Una funzione di ordine superiore utilizzata comunemente in matematica è la composizione di funzioni. È possibile definire in OCaml tale funzione, denotata da `comp`, come segue:

```
(* comp : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b *)
(* comp (f,g) = composizione di f con g *)
let comp(f,g) = function x -> f(g x)
```

che è come dire:

```
let comp(f,g) =
   let h x = f(g x)
   in h
```

Il valore di `comp(f,g)` è cioè quella funzione `h` che, applicata a un argomento `x` riporta `f(g(x))`. Sebbene la forma delle due definizioni precedenti di `comp` metta ben in luce il significato della definizione, quella che verrà in realtà più comunemente utilizzata è la seguente:

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

"`comp` è quella funzione che, applicata a una coppia `(f,g)` riporta una funzione che, applicata a un elemento `x`, riporta il valore di `f(g x)`.

Il valore riportato da `comp` è dunque una funzione. Ad esempio, `comp(double,double)` è la funzione che, applicata a un argomento x, riporta il doppio del doppio di x (il quadruplo di x):

```
# comp (double,double);;
- : int -> int = <fun>
# comp (double,double) 4;;
- : int = 16
```

Si ricordi che nell'applicazione di funzioni OCaml associa a sinistra, quindi `comp (double,double) 4` è equivalente a `(comp (double,double)) 4`.

Possiamo definire la funzione `sixtimes` come composizione di `double` e `treble`:

```
# let sixtimes = comp(double,treble);;
val sixtimes : int -> int = <fun>
# sixtimes 8;;
- : int = 48
```

La funzione `comp` si applica a coppie di funzioni e fornisce come valore una funzione: il suo tipo è $(\alpha \to \beta) \times (\gamma \to \alpha) \to (\gamma \to \beta)$.

```
# comp;;
- : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>
```

Un altro esempio abituale di funzione di ordine superiore è l'operazione di sommatoria. Possiamo definire in Ocaml la funzione `sum` tale che:

$$\texttt{sum (f,n,m)} = \sum_{k=n}^{m} f(k)$$

```
(* sum : (int -> int) * int * int -> int *)
(* sum (f,n,m) = sommatoria f(n) + f(n+1) + ... + f(m) *)
let rec sum (f,n,m) =
  if n>m then 0
  else f(n) + sum (f,n+1,m)
```

Se definiamo la funzione che calcola il quadrato di un intero:

```
(* square: int -> int *)
(* square x = quadrato di x *)
let square x = x*x;;
```

abbiamo ad esempio:

```
# sum (square,1,4);;
- : int = 30
```

Le funzioni di ordine superiore permettono di generalizzare. Ad esempio, consideriamo diverse funzioni costanti:

```
(* k0: 'a -> int *)
(* k0 = funzione costante 0 *)
let k0 y = 0

(* ktrue: 'a -> bool *)
(* ktrue = funzione costante true *)
let ktrue y = true

....
```

Generalizzando, possiamo definire il funzionale costante, tradizionalmente denotato da `K`, che riporta funzioni come valori:

```
(* k: 'a -> 'b -> 'a *)
(* k x = funzione costante x *)
let k x = function y -> x
```

`k x` è quella funzione che, applicata a qualsiasi argomento `y`, riporta `x` stesso.

```
# (k 3) 0;;
- : int = 3
# let f = k true in f "pippo";;
- : bool = true
```

La definizione sopra riportata può essere data nella forma seguente, più compatta:

```
let k x y = x
```

"`k` è quella funzione che, applicata a un argomento `x`, riporta una funzione che, quando è a sua volta applicata a un argomento `y`, riporta il valore `x`".

1.15.1 Currificazione di funzioni

Consideriamo le due funzioni seguenti:

```
let sum (m,n) = m + n
let plus m n = m + n
```

La differenza fondamentale tra le due funzioni sta nel loro tipo: la prima è di tipo `int * int -> int`, mentre `plus` è di tipo `int -> int -> int`. In altri termini, `sum` è una funzione che ha un solo argomento: una coppia di interi, e un valore intero. `plus` è una funzione che ha come argomento un intero, e come valore una funzione di tipo `int -> int`: `plus` è la funzione che, quando à applicata a un intero m, riporta la funzione che aggiunge m al suo argomento.

```
# plus 3;;
- : int -> int = <fun>
# plus 3 5;;
- : int = 8
```

Avendo definito `plus`, possiamo definire la funzione `sommacento` come quella funzione che somma 100 al suo argomento:

```
# let sommacento = plus 100;;
val sommacento : int -> int = <fun>

# sommacento 3;;
- : int = 103
```

Come si vede, a differenza di `sum`, la funzione `plus` può essere applicata anche *solo parzialmente*, riportando come valore una funzione.

La stessa relazione esistente tra `sum` e `plus` vale tra le funzioni `mult` e `times` così definite:

```
# let mult (m,n) = m * n;;
val mult : int * int -> int = <fun>
# let times m n = m * n;;
val times : int -> int -> int = <fun>

# times 3;;
- : int -> int = <fun>
# let quadruplo = times 4;;
val quadruplo : int -> int = <fun>
# quadruplo 5;;
- : int = 20
```

Funzioni come `plus` e `times` si dicono le *versioni currificate* delle rispettive operazioni su coppie `sum` e `mult`:

```
mult    : int * int -> int
times   : int -> int -> int
times 5 :        int -> int
```

`times 5` è un'applicazione parziale di `times`: la funzione che moltiplica per 5 il suo argomento.

Una funzione su tuple si può sempre riscrivere in forma currificata, come una funzione che "consuma un argomento alla volta".

In generale, se f è una funzione di tipo:

$$f : \alpha_1 \times ... \times \alpha_n \to \alpha$$

allora, una funzione f_c si dice la *forma currificata* di f se e solo se:

1. il tipo di f_c è il seguente

$$f_c : \alpha_1 \to (\alpha_2 \to ... \to (\alpha_n \to \alpha)...)$$

2. per ogni $a_1, ..., a_n$, si ha che

$$f(a_1, ..., a_n) = (((f_c a_1)a_2)...a_n)$$

Si noti che le parentesi possono essere omesse, sia nell'applicazione di una funzione currificata (in tal caso si associa a sinistra), sia nel tipo della funzione (e si associa a destra). In altri termini, un'espressione della forma $f_c a_1 a_2 a_3$ sta per $((f_c a_1)a_2)a_3$ e l'espressione che indica il tipo di f_c, $\alpha_1 \to (\alpha_2 \to \alpha_3)$, si può abbreviare in $\alpha_1 \to \alpha_2 \to \alpha_3$.

Come ulteriore esempio consideriamo ancora la funzione `comp` per la composizione:

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

Possiamo definire la versione currificata di `comp`, che si applica ad una funzione alla volta. Tradizionalmente, questa funzione è chiamata **B**; poiché OCaml non ammette nomi di funzioni o variabili che inizino con lettera maiuscola, definiamo:

```
# let b f g x = f (g x);;
val b : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Come abbiamo detto, di ogni funzione a più argomenti è possibile fornire la versione currificata. Ad esempio:

```
# let pair x y = (x,y);;
val pair : 'a -> 'b -> 'a * 'b = <fun>

# let lessthan x y = y < x;;
val lessthan : 'a -> 'a -> bool = <fun>

# let greaterthan x y = y > x;;
val greaterthan : 'a -> 'a -> bool = <fun>
```

L'uso di funzioni currificate[6] permette una grande potenza espressiva, che deriva dalla possibilità di applicare le funzioni parzialmente, fornendo così espressioni che denotano funzioni. Ad esempio, se abbiamo bisogno di utilizzare una funzione che, applicata a un argomento `x` riporta la coppia `(0,x)`, possiamo utilizzare un'applicazione parziale di `pair`: `pair 0`.

Le applicazioni parziali di `lessthan` e `greaterthan` riportano predicati: `lessthan 0` è il predicato "essere minore di 0", `greaterthan 10` è il predicato "essere maggiore di 10"; per questo motivo le funzioni sono state definite "scambiando" l'ordine degli argomenti.

Come ulteriore esempio, riscriviamo in forma currificata la funzione per la sommatoria `sum`:

```
let rec sum g n m =
    if n > m then 0
    else g n + sum g (n+1) m;;
```

da cui:

```
val sum : (int -> int) -> int -> int -> int
```

Allora, avendo definito la forma currificata `times` per il prodotto, possiamo calcolare la sommatoria $\sum_{i=1}^{5} i \times 2$ mediante:

```
# sum (times 2) 1 5;;
- : int = 30
```

Utilizzeremo frequentemente le funzioni in forma currificata, in quanto esse risultano più flessibili delle corrispondenti non currificate.

1.15.2 Operatori infissi

La maggior parte delle funzioni predefinite in OCaml sono anch'esse in forma currificata. Ad esempio, il tipo delle funzioni `max` e `min`, che riportano, rispettivamente, il massimo e il minimo tra due oggetti di un *equality type*, è: `'a -> 'a -> 'a`.

Anche gli operatori infissi predefiniti in OCaml sono in realtà funzioni in forma currificata. Consideriamo ad esempio la somma. Per denotare in OCaml la funzione `+` *tout court*, e non la sua applicazione a determinati argomenti, dobbiamo racchiuderla tra parentesi tonde. Con questa accortezza possiamo conoscere il tipo di `+`:

```
# (+);;
- : int -> int -> int = <fun>
```

[6] Benché a rigor di termini abbia senso parlare di "currificazione" soltanto come relazione tra due funzioni ("f è la forma currificata di g"), parleremo in generale di funzioni currificate quando si tratta di funzioni che "consumano un argomento alla volta" (o funzioni che riportano funzioni come valori) e che quindi possono essere viste come la forma currificata di una corrispondente funzione su tuple.

Come si vede, benché la forma infissa nasconda questo fatto, la somma, come operazione predefinita, è in forma currificata. Essa corrisponde dunque alla funzione `plus`. Lo stesso vale per tutte le operazioni infisse predefinite in OCaml.

Il programmatore può definire operatori infissi: in OCaml, la distinzione tra operatori infissi e prefissi è determinata dal nome stesso della funzione: i simboli infissi sono costituiti da caratteri speciali (si veda il manuale di riferimento [3]). Ad esempio, possiamo definire la composizione (in forma currificata) utilizzando il simbolo `@@`, in notazione infissa:

```
# let (@@) f g x = f(g x);;
val @@ : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# double @@ treble;;
- : int -> int = <fun>
# @@;;
Characters 0-2:
  @@;;
  ^^
Syntax error
# (@@);;
- : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

1.15.3 Espressioni funzionali

L'uso di funzioni di ordine superiore consente di definire in modo generale operazioni parametriche rispetto ad argomenti funzionali, e l'uso di funzioni in forma currificata introduce un nuovo modo di denotare funzioni, senza la necessità di dargli un nome: mediante *espressioni funzionali* (cioè espressioni che hanno come valore una funzione).

Abbiamo cioè tre diversi modi di denotare una funzione:

1. mediante una variabile, in seguito a una dichiarazione esplicita (ad esempio `times`);
2. mediante un'astrazione funzionale, o funzione anonima (ad esempio `function x -> (0,x)`);
3. mediante un'espressione funzionale (ad esempio `double @@ (plus 5)`).

Come esempio dell'uso di espressioni funzionali, consideriamo la funzione `least` che calcola il minimo intero maggiore o uguale a n per il quale vale la proprietà p (se esiste):

```
(* least : (int -> bool) -> int -> int *)
(* least p n = minimo intero maggiore o uguale a n per il quale vale
               la proprieta' p *)
let rec least p n =
  if p n then n
  else least p (n+1)
```

In altre parole, la funzione `least`, applicata a una funzione booleana p, di tipo `int -> bool`, e un intero n, restituisce il più piccolo intero $m \geq n$ tale che $p\ m$ è uguale a `true`.

Per calcolare il minimo intero il cui quadrato sia maggiore di 10 possiamo valutare l'espressione:

```
#  least ((greaterthan 10) @@ square) 1;;
- : int = 4
```

Si noti che, in questa espressione:

```
greaterthan 10 : int -> bool
square : int -> int
@@ : (int -> bool) -> (int -> int) -> (int -> bool)
(greaterthan 10) @@ square : int -> bool
```

I seguenti passaggi mostrano la riduzione dell'espressione precedente fino ad ottenere il suo valore:

```
least ((greaterthan 10) @@ square) 3
  = if ((greaterthan 10) @@ square) 3 then 3
    else least ((greaterthan 10) @@ square) 3
  = if (greaterthan 10) (square 3) then 3
    else least ((greaterthan 10) @@ square) 3
  = if (greaterthan 10) 9 then 3
    else least ((greaterthan 10) @@ square) 3
  = if 9 > 10 then 3
    else least ((greaterthan 10) @@ square) 3
  = least ((greaterthan 10) @@ square) 3
  = if ((greaterthan 10) @@ square) 4 then 4 else ...
  = if greaterthan 10 16 then 4 else ...
  = 4
```

1.15.4 Un'operazione su predicati: `non`

Il "contrario" di un predicato è un'operazione che converte un predicato nel suo opposto, e che chiamiamo `non`. Ad esempio `non` pari è dispari, `non` bello è brutto, ecc. Attenzione a non confondere `non` con `not`: `not` è un'operazione che si applica a un booleano (`not:  bool -> bool`), mentre `non` si applica a un predicato:
`non: ('a -> bool) -> ('a -> bool)`. Può essere così definita:

```
(* non : ('a -> bool) -> 'a -> bool *)
(* non p = contrario di p *)
let non p = function x -> not (p x)
```

o, più semplicemente:

```
let non p x = not (p x)
```

In altri termini, `non p` è quella funzione che, quando è applicata a un argomento `x`, riporta la negazione di `p x`.

1.15.5 Altri esempi

1. Possiamo definire le funzioni `curry` e `uncurry` che forniscono la versione currificata (rispettivamente, non currificata) di funzioni:

   ```
   # let curry f x y = f (x,y);;
   val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
   # let uncurry f (x,y) = f x y;;
   val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
   ```

2. La funzione `c` scambia l'ordine dei primi due argomenti di una funzione currificata: `c f` è una funzione che, applicata a `x` e `y` riporta il valore di `f y x`:

   ```
   let c f x y = f y x;;
   ```

   Il tipo di `c` è: $(\alpha \to \beta \to \gamma) \to \beta \to \alpha \to \gamma$. Potremmo ad esempio controllare se 10 è maggiore di 3 valutando l'espressione `c lessthan 3 10` (senza la necessità di definire il predicato "opposto" `greaterthan`):

   ```
   # c lessthan 3 10;;
   - : bool = true
   ```

   La versione di `c` che si applica a funzioni non currificate si può definire come segue:

   ```
   # let swappair = uncurry @@ c @@ curry;;
   val swappair : ('_a * '_b -> '_c) -> '_b * '_a -> '_c = <fun>
   ```

   Se definiamo la funzione identità:

   ```
   # let id x = x;;
   val id : 'a -> 'a = <fun>
   ```

   allora:

   ```
   # (swappair id) (3,4);;
   - : int * int = (4, 3)
   ```

3. Si vuole definire una funzione `minval` che, data una funzione f di tipo $\alpha \times int \to \beta$, una funzione g di tipo $\alpha \to int$ e un valore x di tipo α, restituisca il minimo valore di $f(x, y)$ per y compreso tra 0 e $g(x)$.

   La definizione proposta utilizza una funzione ausiliaria `minsearch` che, applicata a due interi, `y` e `bound`, riporta il valore `f(x,y)` se `y` ha raggiunto `bound`, altrimenti il minimo tra `f(x,y)` e `minsearch (y+1) bound`. La funzione riporta dunque il minimo valore di `f(x,z)` per `z` compreso tra `y` e `bound`. Essa è richiamata inizialmente con argomenti `0` e `g x`.

```
(* minval : ('a * int -> 'b) -> ('a -> int) -> 'a -> 'b *)
(* minval f g x = minimo valore di f(x,y) per y compreso tra
                  0 e g(x) *)
(* minsearch: int -> int -> 'b *)
(* minsearch y bound = minimo valore di f(x,z) per z compreso
                  tra y e bound *)
let minval f g x =
  let rec minsearch y bound =
    if y = bound then f(x,y)
    else min (f(x,y)) (minsearch (y+1) bound)
  in minsearch 0 (g x)
```

4. La funzione `equalfun`, applicata a due funzioni `f` e `g` di tipo `int` $\rightarrow$ `int`, riporta `false` se esse differiscono per qualche argomento positivo, ed è indefinita altrimenti.

```
(* equalfun : (int -> 'a) -> (int -> 'a) -> bool *)
(* equalfun f g = false se esiste x positivo tale che f x
                  e' diverso da g x *)
(* equalfrom: int -> bool *)
(* equalfrom x = false se esiste y maggiore o uguale a x tale che f y
                  e' diverso da g y *)
let equalfun f g =
  let rec equalfrom x =
    f x = g x && equalfrom (x+1)
  in  equalfrom 1

# let double_bis x =  if x < 100 then 2 * x else 0;;
val double_bis : int -> int = <fun>
# equalfun double double_bis;;
- : bool = false
```

Per esercizio, si estenda la funzione `equalfun` in modo che riporti `false` anche nel caso in cui le funzioni `f` e `g` differiscano per argomenti negativi.

Pensate che sia possibile migliorare i test in modo che la funzione riporti `true` nel caso in cui le funzioni abbiano sempre gli stessi valori per gli stessi argomenti?

5. La funzione `both`, applicata a una funzione `f` e un altro argomento `x`, applica `f` a `x` ed il risultato di nuovo a `x`.

```
#  let both f x = f x x;;
val both : ('a -> 'a -> 'b) -> 'a -> 'b = <fun>
# both plus 3;;
- : int = 6
# both times 5;;
- : int = 25
```

In effetti si ha `double = both plus` e `square = both times`.

Capitolo 2

Liste

2.1 Tipi di dati induttivi e ricorsione

Diversi tipi di dati possono essere definiti induttivamente. L'esempio più elementare è costituito dall'insieme dei numeri naturali, che si può pensare come costruito a partire dall'elemento iniziale 0 mediante ripetute applicazioni dell'operazione +1; tale operazione consente di generare, a partire da un n già costruito, un altro numero, $n+1$ o $succ(n)$ (il successore di n):

$$\mathbb{N} = \{0, succ(0), succ(succ(0)), \ldots\}$$

In tal modo si stabilisce che 0 è un numero naturale, il suo successore, 1, è un numero naturale, il successore del successore, 2, è un numero naturale, ecc. Un numero naturale, dunque, o è 0 oppure ha la forma $succ(n)$, dove n è un numero naturale. In altri termini, i naturali sono *dati di un tipo induttivo*, cioè $\mathbb{N}$ si può definire come segue:

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

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

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

Questa è una **definizione induttiva** di $\mathbb{N}$: la prima clausola stabilisce un "oggetto di base", a partire dal quale si possono costruire tutti gli altri oggetti applicando l'operazione indicata nella seconda clausola (la clausola induttiva).

La definizione induttiva dei numeri naturali giustifica la **definizione ricorsiva di funzioni** sui numeri naturali. Consideriamo ad esempio la definizione ricorsiva della funzione fattoriale:

$$\begin{aligned} fact(0) &= 1 \\ fact(n+1) &= (n+1) \times fact(n) \end{aligned}$$

La funzione fattoriale è ben definita in quanto essa è definita per 0 (dalla prima clausola data sopra); è definita per 1, in quanto è uguale a $1 \times fact(0) = 1 \times 1 = 1$;

è definita per 2: $fact(2) = 2 \times fact(1) = 2 \times 1 = 2$, ecc. Ora, qualsiasi numero è "raggiungibile" in questo modo a partire dallo 0, quindi la funzione è definita per tutti i numeri naturali.

Questo ragionamento intuitivo è trattato con maggior rigore, mediante l'uso del ben noto *principio di induzione matematica*, nell'Appendice A.

L'induzione è una tecnica generale per la definizione di insiemi. Ad esempio, si può definire il sottoinsieme D di $\mathbb{N}$ come segue:

(i) $1 \in D$;

(ii) per ogni $n \in \mathbb{N}$, se $n \in D$, allora $n + 2 \in D$;

(iii) nient'altro è in D.

Anche la seguente definizione dell'insieme $P_{\{1,2,3\}}$ è induttiva, pur conformandosi ad uno schema più generale:

(i) $\{1, 2, 3\} \subseteq P_{\{1,2,3\}}$;

(ii) per ogni $n, m \in \mathbb{N}$, se $n, m \in P_{\{1,2,3\}}$, allora $n \times m \in P_{\{1,2,3\}}$;

(iii) nient'altro è in $P_{\{1,2,3\}}$.

La definizione induttiva di un insieme di oggetti A consiste di (i) una *clausola base*, che stabilisce un insieme iniziale di oggetti che appartengono a A; (ii) una *clausola induttiva*, che stabilisce quali operazioni si possono applicare ad elementi di A ottenendo ancora elementi di A, cioè quali sono le operazioni rispetto alle quali A è *chiuso*; (iii) una *clausola di chiusura*, che stabilisce che A contiene soltanto quegli oggetti che si possono ottenere dall'insieme base applicando le operazioni della clausola induttiva.

Quando A è un sottoinsieme di un insieme S precedentemente definito, anziché usare la clausola di chiusura, si può definire A come il più piccolo sottoinsieme E di S che soddisfa la clausola base e la clausola induttiva.

Ad esempio, possiamo definire un insieme di coppie di naturali, $M \subseteq \mathbb{N} \times \mathbb{N}$ come il più piccolo insieme $E \subseteq \mathbb{N} \times \mathbb{N}$ tale che:

(i) $(0, 0) \in E$

(ii) per ogni $(n, m) \in E$: $(n + 1, m + 2) \in E$

Naturalmente, la forma di una definizione induttiva può a volte essere apparentemente più complicata. Ad esempio, si può definire induttivamente la relazione binaria $<$ su $\mathbb{N}$ come il più piccolo sottoinsieme E di $\mathbb{N}\times\mathbb{N}$ tale che:

(i) $\langle m, succ(m)\rangle \in E$;

(ii) se $\langle m, n\rangle \in E$, allora $\langle m, succ(n)\rangle \in E$.

Nelle definizioni induttive che daremo nella prima forma ometteremo a volte, nel seguito, la clausola di chiusura, lasciandola sottintesa.

Per ogni insieme definito induttivamente è giustificata la definizione ricorsiva di funzioni, come si vedrà meglio in seguito.

2.2 Le liste

La definizione induttiva di insiemi è una tecnica importante in informatica. Infatti, molte strutture di dati interessanti sono definite induttivamente. Un esempio è quello delle liste.

Se `a, b, c, ...` sono elementi di uno stesso tipo α, allora una lista di elementi di tipo α è una sequenza finita di tali elementi. Le liste vengono denotate in OCaml racchiudendo gli elementi tra parentesi quadre e separandoli mediante punti e virgola. Ad esempio, `[3;6;3;10]` è una lista di interi, `[true; 3>0 && false; not true]` è una lista di booleani.

Ciascuna lista si può vedere come costruita a partire dalla lista vuota (scritta `[]`) mediante l'operazione di "inserimento in testa", indicata con `::` e chiamata operazione "cons". Ad esempio, la lista `[3;6;3;10]` si ottiene inserendo `3` in testa alla lista `[6;3;10]`: `[3;6;3;10]` = `3::[6;3;10]`; a sua volta `[6;3;10]` = `6::[3;10]`, ecc. Infine, `[10] = 10::[]`. Di fatto, `[3;6;3;10]` è un modo compatto (e più leggibile) di scrivere l'espressione `3::(6::(3::(10::[])))`.

In generale, l'insieme delle liste di elementi di un tipo `T` costituisce un tipo, denotato da `T list`. Come $\times$, anche `list` è un costruttore di tipi: "applicato" a un tipo `T` restituisce il tipo delle liste i cui elementi sono di tipo `T`. Per cui, ad esempio, `[3;6;3;10]` è un'espressione di tipo `int list`, e `[[4;5],[6;7;8],[9;10]]` è una `(int list) list`, cioè una lista di liste di interi. L'operatore `::` è di fatto una funzione polimorfa, di tipo α $\times$ α `list -> `α `list` (scritta in notazione infissa, come molti operatori aritmetici).

L'insieme delle liste di elementi di un tipo α, che chiameremo in breve α `list`, può essere definito induttivamente come segue:

(i) La lista vuota, `[]`, è una α `list`;

(ii) se `x` è di tipo α e `lst` è una α `list`, allora `(x::lst)` è una α `list`;

(iii) nient'altro è una α `list`.

Una α `list` è dunque costruita utilizzando, oltre ad elementi di tipo α, soltanto il valore `[]` e la funzione `::`. Questi sono i *costruttori* del tipo lista (così come parentesi e virgole sono i costruttori dei diversi tipi di tuple) e dalla clausola di chiusura segue che qualsiasi lista ha una delle due forme seguenti: `[]` oppure `(x::lst)`.

In una lista della forma `(x::lst)`, diciamo che `x` è la *testa* della lista, e `lst` la sua *coda*. Quindi la coda di una lista non vuota è la lista che si ottiene togliendo il primo elemento dalla lista stessa.

Si noti che alla precedente definizione, per ogni tipo α, `[]` è una lista di elementi di tipo α. Quindi `[]` è un oggetto *polimorfo*: può avere infiniti tipi diversi, a seconda dell'espressione in cui occorre. Ad esempio, la prima occorrenza di `[]` in `(1::[],true::[])` è di tipo `int list`, mentre la seconda è di tipo `bool list`.

2.2.1 Pattern matching con le liste

Nel definire funzioni sulle liste è possibile utilizzare il pattern matching. In un pattern per liste possono occorrere soltanto variabili e i due "costruttori" del tipo lista, cioè `[]` e `::` (oltre, ovviamente, ai costruttori del tipo degli elementi della lista).

La tabella seguente riporta, come esempi, alcuni pattern per liste (prima colonna), un'espressione con cui il pattern corrispondente viene confrontato, in terza colonna, eventualmente, il modo di leggere l'espressione della seconda colonna in termini di applicazioni dell'operazione `::`, e, in quarta colonna, l'esito del confronto: fallimento o successo, con gli eventuali legami che vengono stabiliti dall'operazione di pattern matching.

pattern	*espressione*		*successo e legami*
`[]`	`[]`		successo
`[]`	`[1]`		fallimento
`[]`	`[1;2]`		fallimento
`[x]`	`[]`		fallimento
`[x]`	`[1]`		x=1
`[x]`	`[1;2]`		fallimento
`x::[]`	`[]`		fallimento
`x::[]`	`[1]`	`1::[]`	x=1
`x::[]`	`[1;2]`	`1::[2]`	fallimento
`x::y::[]`	`[]`		fallimento
`x::y::[]`	`[1]`	`1::[]`	fallimento
`x::y::[]`	`[1;2]`	`1::2::[]`	x=1, y=2
`x::y`	`[]`		fallimento
`x::y`	`[1]`	`1::[]`	x=1, y=[]
`x::y`	`[1;2]`	`1::[2]`	x=1, y=[2]
`x::y`	`[1;2;3]`	`1::[2;3]`	x=1, y=[2;3]
`x::rest`	`[]`		fallimento
`x::rest`	`[1]`	`1::[]`	x=1, rest=[]
`x::rest`	`[1;2]`	`1::[2]`	x=1, rest=[2]
`x::rest`	`[1;2;3]`	`1::[2;3]`	x=1, rest=[2;3]
`x::y::rest`	`[]`		fallimento
`x::y::rest`	`[1]`	`1::[]`	fallimento
`x::y::rest`	`[1;2]`	`1::2::[]`	x=1, y=2, rest=[]
`x::y::rest`	`[1;2;3]`	`1::2::[3]`	x=1, y=2, rest=[3]
`x::y::rest`	`[1;2;3;4;5]`	`1::2::[3;4;5]`	x=1, y=2, rest=[3;4;5]

Naturalmente, in un pattern di lista può anche occorrere la variabile muta. Ad esempio `_::rest` è un pattern conforme con ogni lista non vuota. Quando un'operazione di *pattern matching* con tale pattern ha successo, viene aggiunto all'ambiente soltanto il legame di `rest` con la coda della lista considerata.

2.2.2 Costruttori, selettori e predicati per il tipo lista

I costruttori del tipo α `list` sono, come abbiamo visto, la costante `[]` e il costruttore funzionale `::`, usato in notazione infissa. In corrispondenza del costruttore `::` abbiamo i due selettori che riportano, applicati a una lista non vuota, rispettivamente la testa e la coda della lista. Infine, possiamo definire il predicato `null` che controlla se una lista è vuota.

`hd : 'a list -> 'a`
: Selettore: applicato a una lista, ne riporta la "testa" (*head*), cioè il primo elemento. Riporta un errore quando è applicato alla lista vuota.

`tl : 'a list -> 'a list`
: Selettore: applicato a una lista, ne riporta la "coda" (*tail*), cioè la lista che si ottiene eliminando il primo elemento. Riporta un errore quando è applicato alla lista vuota.

`null : 'a list -> bool`
: Predicato: determina se una lista è vuota.

Queste operazioni costituiscono l'insieme delle operazioni di base sulle liste: in qualsiasi linguaggio esse siano implementate, e con qualsiasi tecnica esse siano implementate, queste operazioni devono comunque essere definite perché l'implementazione risulti "completa". In OCaml, i costruttori sono "operazioni" predefinite. I selettori ed il predicato possono essere definiti come segue:

```
exception NullList;;

let hd = function
   (x::_) -> x
  | _ -> raise NullList;;

let tl = function
   (_::rest) -> rest
  | _ -> raise NullList;;

let null lst = lst = [];;
```

In realtà le funzioni `hd` e `tl` sono definite nel modulo `List` della libreria standard di OCaml (`http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`). Quindi è possibile utilizzare `List.hd` e `List.tl` senza bisogno di definirle. Se esse sono applicate alla lista vuota sollevano, rispettivamente, le eccezioni `Failure "hd"` e `Failure "tl"`.

Un'importante funzione definita sulle liste è l'operazione di concatenazione, denotata da `@` (utilizzata in modo infisso):

```
# [1;2;3] @ [4;5];;
- : int list = [1; 2; 3; 4; 5]
```

Sebbene sia predefinita in OCaml, come vedremo in seguito la concatenazione può essere definita sulla base delle operazioni di base.

Alcune delle funzioni che definiremo in questo capitolo sono in realtà funzioni definite nel modulo `List` della libreria standard di OCaml (si veda il manuale d'uso di OCaml [3]). In questi casi lo segnaleremo, e sarà quindi possibile utilizzare tali funzioni mediante `List.<nome-della-funzione>`.

2.3 Definizione ricorsiva di funzioni sulle liste

Anche nel caso delle liste, la definizione induttiva fornisce un metodo per generare passo passo l'insieme delle α `list`. Per esempio possiamo generare tutti i valori di tipo `bool list` come segue: al primo stadio è generata la lista vuota; al secondo stadio le liste `true::[]` (`[true]` e `false::[]` (`[false]`; al terzo stadio le liste `true::[true], true::[false], false::[true], false::[false]` (cioè tutte le liste di due elementi), e così via (naturalmente, l'insieme delle liste generate in ciascuno stadio è finito solo se l'insieme degli elementi è finito).

Questo giustifica (intuitivamente) la definizione ricorsiva di funzioni sulle liste. Ad esempio, possiamo definire ricorsivamente una funzione (polimorfa)

`length:` α `list -> int`

che, applicata a una lista L ne riporta il numero di elementi, considerando due casi: se L è la lista vuota allora la funzione riporta 0; altrimenti, per la definizone induttiva delle liste, L ha la forma $(x :: rest)$ per qualche lista $rest$ (costruita allo stadio precedente); allora si calcola il valore della funzione per $rest$ e si somma 1 al risultato. In OCaml:

```
(* length : 'a list -> int *)
let rec length = function
    [] -> 0
  | x::rest -> 1 + length rest
```

(il modulo `List` della libreria di OCaml contiene in realtà la funzione `length`). Quel che – sempre intuitivamente – garantisce che la funzione sia definita per tutte le liste è il fatto che nella chiamata ricorsiva essa viene applicata a un argomento in qualche senso "più piccolo" (una lista più corta, cioè generata ad uno stadio precedente), per cui le chiamate ricorsive prima o poi non potranno che terminare con `length []`, il cui calcolo termina subito riportando 0. Si veda l'Appendice A.

Nella forma più semplice, la definizione ricorsiva di una funzione sulle liste considera i due casi "lista vuota" (caso base) e "lista non vuota" (caso induttivo). Per formulare il caso induttivo, si assuma di saper già correttamente calcolare il valore della funzione per `rest` e si consideri quali operazioni occorre fare per ottenere da tale valore il risultato che la funzione deve riportare per `x::rest`.

Ad esempio, possiamo calcolare la lista dei quadrati di una lista di interi mediante la funzione `square_list: int list -> int list`, che richiama `square`:

```
(* square: int -> int}
(* square n = quadrato di n *)
let square n = n*n

(* square_list: int list -> int list *)
(* square_list [x1;...;xn] = [square x1;...;square xn] *)
let rec square_list = function
    [] -> []
  | n::rest -> (square n)::(square_list rest)

# square_list [2;4;6;8];;
- : int list = [4; 16; 36; 64]
```

Consideriamo ora il seguente problema: dati due interi `m` e `n`, si vuole costruire la lista di interi `[m,m+1,...,n]` (la lista vuota se `m > n`). Nel caso induttivo, possiamo ridurre il problema di costruire la lista `[m,m+1,...,n]` a quello di inserire `m` in testa alla lista `[m+1,...,n]`. La costruzione della lista `[m+1,...,n]` è un problema più semplice della costruzione della lista `[m,m+1,...,n]`: intuitivamente, si tratta di una lista più corta. Formalmente, la "misura" che diminuisce nel passaggio da `[m,m+1,...,n]` a `[m+1,...,n]` è la differenza tra l'ultimo e il primo elemento della lista. Possiamo quindi definire ricorsivamente la funzione `upto` come segue:

```
(* upto : int -> int -> int list *)
(* upto m n = [m;m+1;....;n] *)
let rec upto n m =
  if n>m then []
  else n::upto (n+1) m
```

Nel caso base ($m > n$) si ha che $n - m$ è negativo. Nel caso ricorsivo, se $n - m = k$, la chiamata ricorsiva avviene sugli argomenti $m+1$ e n, tali che $n-(m+1) = k-1 < k$. Quindi la ricorsione termina sempre.

Molte funzioni si applicano a liste ma anche ad altri argomenti. Ad esempio, se vogliamo determinare se un valore `y` è un elemento di una lista `lst`, scriveremo una funzione a due argomenti. In questo caso, la definizione sarà per ricorsione sulla lista, mantenendo fissato l'elemento `y`:

```
(*    mem: 'a -> 'a list -> bool
      mem x lst = true se lst contiene x *)
let rec mem x = function
    [] -> false
  | y::rest ->
       x=y || mem x rest
```

La funzione `mem` è predefinita nel modulo `List`.

Il tipo di `mem` è `'a -> 'a list -> bool`. Si noti tuttavia che il tipo `'a` deve essere un "tipo con uguaglianza", cioè un tipo su cui è possibile fare il test di uguaglianza. Come già visto, i tipi semplici sono tipi con uguaglianza; un tipo composito, come le

coppie o le liste, è un tipo con uguaglianza se e solo se tutti i tipi componenti sono tipi con uguaglianza. Al contrario, i tipi funzionali non sono tipi con uguaglianza, così come non lo sono tipi compositi con componenti funzionali. Quindi in generale la funzione `mem` non può essere applicata a funzioni e liste di funzioni:

```
# mem square [function n -> n*n];;
Exception: Invalid_argument "equal: functional value".
```

Una funzione può essere indefinita per alcuni argomenti, ad esempio per la lista vuota. Ne sono esempi la funzione `last`, che, applicata a una lista, ne riporta l'ultimo elemento, e di `init`, che, applicata a una lista, riporta il suo argomento senza l'ultimo elemento. In queste definizioni, il caso base della ricorsione è rappresentato dalle liste con un unico elemento, anziché dalla lista vuota.

```
(* last : 'a list -> 'a *)
let rec last = function
    [x] -> x
  | _::rest -> last rest
  | _ -> failwith "last"

(* init : 'a list -> 'a list *)
let rec init = function
    [x] -> []
  | x::rest -> x::init rest
  | _ -> failwith "last"

# last [1;2;3];;
- : int = 3
# init [1;2;3];;
- : int list = [1; 2]
# let lst = [1;2;3] in
  (init lst) @ [last lst];;
- : int list = [1; 2; 3]
```

L'ultimo caso nelle definizioni è aggiunto per evitare che il compilatore dia il *Warning* di pattern matching non esaustivo, e che in fase di esecuzione venga sollevata l'eccezione predefinita `Match_failure` (generica e poco informativa).

Quando una funzione si applica a due argomenti di tipo lista, è necessario determinare su quale argomento si effettua la ricorsione. Ad esempio, si può definire la funzione `append` (versione prefissa di `@`, che si applica a due liste `prima` e `seconda` riportando la loro concatenazione, per ricorsione sulla prima lista, lasciando invariata la seconda:

```
(* append: 'a list -> 'a list -> 'a list *)
let rec append prima seconda =
  match prima with
    [] -> seconda
  | x::prima ->  x::(prima@seconda)
```

Utilizzando la funzione `@` possiamo definire la funzione `reverse: 'a list -> 'a list`, che inverte l'ordine degli elementi del suo argomento. Ricorsivamente: se la lista è vuota (caso base), essa è il *reverse* di sé stessa; altrimenti, se la lista ha la forma `x::rest`, il suo `reverse` si ottiene inserendo `x` in coda al *reverse* di `rest`.

```
(* reverse: 'a list -> 'a list *)
(* reverse [x1;...;xn] = [xn;...;x1] *)
let rec reverse = function
    [] -> []
  | x::rest -> (reverse rest) @ [x]

# reverse [1;2;3;4;5];;
- : int list = [5; 4; 3; 2; 1]
```

(il modulo `List` della libreria di OCaml contiene la funzione `rev`, che rovescia una lista).

In alcuni casi si "mescola" la ricorsione sulle liste con la ricorsione sui naturali, come nel caso delle funzioni `take`, che "prende" i primi n elementi di una lista, e `drop`, che ne elimina i primi n:

```
(* take : int -> 'a list -> 'a list *)
(* take n lista= primi n elementi di lista,
            o lista stessa se non ce ne sono abbastanza *)
let rec take n = function
    [] -> []
  | x::rest ->
      if n<=0 then []
      else  x::take (n-1) rest

(* drop : int -> 'a list  -> 'a list *)
(* drop n lista = lista senza i primi n elementi,
             o lista vuota se n e' maggiore della lunghezza di lista *)
let rec drop n = function
    [] -> []
  | x::rest,n ->
         if n<=0 then x::rest
         else drop (n-1) rest
```

Osserviamo che, nella definizione di `drop`, l'argomento "lista" viene decomposto mediante il pattern matching, con l'uso del pattern `x::rest`, e (nel corpo della funzione) la lista viene poi ricostituita (nel ramo `then`). In casi come questo è utile una particolare forma di pattern:

```
<PATTERN> as <VARIABILE>
```

Quando un'espressione viene confrontata con un pattern di questa forma, il confronto avviene con `<PATTERN>` e dà esito positivo se tale confronto dà esito positivo. L'ambiente viene esteso mediante l'aggiunta dei legami risultanti dal pattern matching con

<PATTERN>, come al solito, ma inoltre viene anche aggiunto il legame di <VARIABILE> con l'espressione. Utilizzando il pattern matching con as possiamo allora riscrivere la definizione di drop come segue:

```
let rec drop n = function
    [] -> []
  | _::rest as lst ->
      if n>0 then drop (n-1) rest
      else lst
```

Fin qui abbiamo considerato liste di elementi di un tipo semplice. La funzione flatten si applica invece a liste di liste e riporta la loro concatenazione (la funzione è definita nel modulo List).

```
(* flatten: 'a list list -> 'a list *)
(* flatten [lista_1;...;lista_n] = lista_1 @ lista_2 @ ... @lista_n *)
let rec flatten = function
    [] -> []
  | lst::rest -> lst @ (flatten rest)

# flatten [[1;2];[];[4;5;6]];;
- : int list = [1; 2; 4; 5; 6]
```

Concludiamo questa sezione introducendo due funzioni di utilità generale, per la conversione di una stringa in una lista di caratteri (explode) e viceversa (implode). Utilizziamo qui la funzione make della libreria String (String.make: int -> char -> string), che, applicata a un intero n e un carattere c, riporta la stringa di lunghezza n costituita da tutti caratteri uguali a c:

```
# String.make 10 'A';;
- : string = "AAAAAAAAAA"
```

Nel caso in cui n è uguale a 1, la funzione converte caratteri in stringhe.

```
(* implode : char list -> string *)
(* implode lst = stringa con i caratteri in lst *)
let rec implode = function
    [] -> ""
  | x::rest -> (String.make 1 x)^implode rest

(* explode : string -> char list *)
(* explode s = lista con i caratteri di s *)
(* aux: aux : int -> char list *)
(* aux n = lista con i caratteri di str a partire da quello in
           posizione n *)
let rec explode str =
  let maxindex = (String.length str) - 1 in
  let rec aux n =
```

```
    if n > maxindex then []
    else (str.[n])::aux(n+1)
  in aux 0
```

Vedremo nel prossimo paragrafo come definire funzioni equivalenti che implementino però un algoritmo iterativo.

2.3.1 Iterazione sulle liste

La struttura di lista è lineare, ed è quindi naturale definire funzioni ricorsive di coda sulle liste (si veda il paragrafo 1.10.2). Si consideri ad esempio il calcolo della lunghezza di una lista. Un algoritmo iterativo può essere formulato in stile imperativo come segue:

```
lungezza(lst) =
   tmp <- lst
   result <- 0
   while tmp <> []
   do
     result <- result+1
     tmp <- List.tl tmp
   done
   return result
```

Questo algoritmo utilizza un ciclo, in cui sono manipolate due variabili, `tmp` e `len`. Lo stile funzionale di implementazione di questo algoritmo ricorre ad una funzione ausiliaria, con due parametri, `tmp` e `len`, inizializzati rispettivamente con la lista `lst` e `0`:

```
(* len: 'a list -> int *)
(* aux: int -> 'a list -> int *)
(* aux result tmp = result + lunghezza di tmp *)
let len lst =
  let rec aux result = function
         [] -> result
       | _::rest -> aux (result+1) rest
  in aux 0 lst
```

Ed ecco come procede, per riduzioni, il calcolo di `len [1;2;3]`:

```
len [1;2;3] = aux [1;2;3] 0
            = aux [2;3] 1
            = aux [3] 2
            = aux [] 3 = 3
```

Consideriamo ora la funzione `upto` precedentemente definita. Il processo di calcolo che essa genera è ricorsivo, come si vede dal seguente esempio:

```
upto 3 5 = if 3 > 5 then [] else 3::upto 4 5
         = 3 :: (if 4>5 then [] else 4::upto 5 5)
          = 3 :: (4 :: (if 5>5 then [] else 5::upto 6 5))
          = 3 :: (4 :: (5 :: (if 6>5 then []
                              else 6::upto 7 5))
          = 3 :: (4 :: (5 :: [])) = [3,4,5]
```

Possiamo pensare a un algoritmo iterativo equivalente. Per far ciò abbiamo bisogno di una funzione ausiliaria che esegue l'iterazione e dispone di un parametro aggiuntivo per "accumulare" il risultato, e definire:

```
(* versione iterativa di upto? *)
(* upto_forse: int -> int -> int list *)
(* aux: int list -> int -> int -> int list *)
let upto_forse m n =
  let rec aux result m n =
    if m > n then result
    else aux (m::result) (m+1) n
  in aux [] m n
```

Ma avremo:

```
# upto_forse 3 5;;
- : int list = [5; 4; 3]
```

Infatti

```
aux [] 3 5  = aux [3] 4 5 = aux [4;3] 5 5 = ...
```

e abbiamo quindi definito un `downto`! Allora, anziché incrementare il primo estremo della lista, occorre decrementare l'ultimo:

```
(* upto_it : int -> int -> int list *)
(* upto_it m n = upto m n *)
(* aux: int list -> int -> int -> int list *)
(* aux lista m n = [m;m+1;...;n] @ result *)
let upto_it m n =
  let rec aux result m n =
    if m > n then result
    else aux (n::result) m (n-1)
  in aux [] m n
```

Analogamente, se tentiamo di definire un `take` iterativo in modo *naive*, otteniamo in realtà una lista con gli elementi rovesciati:

```
(* versione iterativa di take? *)
let rtake n lst =
  let rec aux result n = function
      [] -> result
```

```
    | x::rest -> if n<=0 then result else
                 aux (x::result) (n-1) rest
  in aux [] n lst

# rtake 3 [1;2;3;4;5;6];;
- : int list = [3; 2; 1]
```

Ad ogni iterazione gli elementi non vanno inseriti in testa all'accumulatore, ma in coda:

```
(* versione iterativa con l'inserimento in coda *)
(* aux : 'a list -> int -> 'a list -> 'a list *)
(* aux result n lst = result @ (take n lst) *)
let take_it n lst =
  let rec aux result n = function
      [] ->  result
    | x::rest -> if n<=0 then result else
                 aux (result@[x]) (n-1) rest
  in aux [] n lst
```

Si noti che però eseguire un "append" ad ogni iterazione è notevolmente costoso. Vedremo tra poco una versione più efficiente di `take_it`

La funzione `rtake` ci suggerisce di definire la funzione che rovescia una lista proprio sfruttando un'ausiliaria simile a quella di `rtake`. La funzione `rev`, definita qui sotto (e predefinita nel modulo `List`, è appunto la versione iterativa di `reverse`: svolge il ruolo di *accumulatore*:

```
(* rev: 'a list -> 'a list *)
(* versione iterativa di reverse*)
(* revto : 'a list -> 'a list -> 'a list *)
(* revto result lst = (reverse lst) @ result *)
let rev lst =
   let rec revto result = function
       [] -> result
     | x::rest -> revto (x::result) rest
   in revto [] lst
```

Possiamo definire `revto` a top-level anziché localmente a `rev`, per verificare il suo comportamento:

```
let rec revto result = function
       [] -> result
     | x::rest -> revto (x::result) rest

# revto [1;2;3] [10;20;30];;
- : int list = [30; 20; 10; 1; 2; 3]
```

Usando `List.rev`, possiamo definire una diversa versione iterativa di `take`:

```
(* take iterativo: versione con il reverse *)
(* aux : 'a list -> int -> 'a list -> 'a list *)
(* aux result n list = (rev result) @ (take n lst) *)
let take_it n lst =
  let rec aux result n = function
      [] -> List.rev result
    | x::rest ->
        if n<=0 then List.rev result else
        aux (x::result) (n-1) rest
  in aux [] n lst
```

Rovesciare la lista soltanto alla fine è più efficiente, dunque preferibile, dell'applicazione di tanti "append" quanti sono gli elementi della lista.

Infine, ecco una versione iterativa delle funzioni `implode` e `explode`:

```
(* implode: char list -> string *)
(* implode lst = stringa con i caratteri in lst *)
(* aux : string -> char list -> string *)
(* aux s lista = s concatenato con la stringa costituita dai
                 caratteri in lista *)
let implode lst =
  let rec aux result = function
      [] -> result
    | c::rest -> aux (result^(String.make 1 c)) rest
  in aux "" lst

(* explode: string -> char list *)
(* explode s = lista dei caratteri che compongono s *)
(* aux : int -> char list -> char list *)
(* aux n result = lista dei caratteri della stringa s fino
                  a quello in posizione n, concatenata con
                  result *)
let explode s =
  let rec aux n result =
    if n < 0 then result
    else aux (n-1) (s.[n] :: result) in
  aux (String.length s - 1) []
```

2.3.2 Rappresentazione di insiemi finiti

Le liste possono essere utilizzate per rappresentare diversi tipi astratti di dati, tra cui gli insiemi finiti. La funzione `List.mem`[1] rappresenta allora il test di appartenenza.

[1] Abbiamo definito, per esercizio, a pagina 60 la funzione `mem`, che è comunque predefinita nel modulo `List`.

Convenendo di voler utilizzare per rappresentare insiemi soltanto liste senza ripetizioni, lasciamo per esercizio la definizione delle operazioni di unione, intersezione e differenza insiemistica, con le seguenti specifiche:

```
(* union: 'a list -> 'a list -> 'a list *)
(* union set1 set2 = lista senza ripetizioni che rappresenta l'unione
           degli insiemi rappresentati, rispettivamente, dalle liste
           set1 e set2 *)

(* intersect: 'a list -> 'a list -> 'a list *)
(* intersect set1 set2 = lista senza ripetizioni che rappresenta l'intersezione
           degli insiemi rappresentati, rispettivamente, dalle liste
           set1 e set2 *)

(* setdiff: 'a list -> 'a list -> 'a list *)
(* setdiff set1 set2 = lista senza ripetizioni che rappresenta l'insieme
           contenente tutti gli elementi di set1 che non occorrono in
                set2 *)
```

Questi sono esempi di come vogliamo che si comportino le funzioni:

```
# union [1;2;3;4] [2;4;6;8];;
- : int list = [6; 8; 1; 2; 3; 4]

# intersect [1;2;4;5;6] [1;3;5;7];;
- : int list = [1; 5]

# setdiff [1;2;3;4;5;6] [2;10;4;6];;
- : int list = [1; 3; 5]
```

Ovviamente, dato che l'ordine degli elementi in una lista utilizzata per rappresentare un insieme non è importante, le funzioni potrebbero riportare una qualsiasi permutazione di quelle mostrate sopra.

Diamo qui soltanto alcuni suggerimenti su come impostare il ragionamento per definire ricorsivamente `union` e `intersect`. In entrambi i casi, le funzioni hanno due argomenti di tipo `'a list`, ed è quindi necessario decidere su quale dei due eseguire primariamente la ricorsione. Nel caso dell'unione, che gode della proprietà commutativa ($S_1 \cup S_2 = S_2 \cup S_1$), la scelta è irrilevante. Se decidiamo di calcolare `union set1 set2` per ricorsione sul secondo argomento, dobbiamo considerare i due casi `set2=[]` e `set2` non vuoto, quindi della forma `x::rest`, e definire quindi qual è il valore della funzione nei due casi `union set1 []` e `union set1 (x::rest)`. Nel secondo caso, si assume (ipotesi della ricorsione) di saper calcolare (mediante la chiamata ricorsiva) il valore di `union set1 rest`: a questo valore si dovrà aggiungere `x` soltanto se non vi occorre già. Può essere utile definire la semplice funzione ausiliaria `setadd: 'a -> 'a list -> 'a list`, tale che `setadd x set` riporti la lista (senza ripetizioni) che rappresenta l'insieme che si ottiene aggiungendo (se non esiste già) `x` a `set`.

In alternativa (e in modo più efficiente), nell caso `union set1 (x::rest)` si può semplicemente controllare se `x` occorre in `set1`, eseguendo quindi una ricorsione *secondaria* su `set1`. Infatti, dato che si assume che `x` non occorra più volte in `set2`, occorrerà in `union set1 rest` soltanto se occorre in `set1`.

Nel caso della differenza insiemistica, il ruolo dei due argomenti è diverso. Per calcolare `intersect set1 set2` si vuole fare la ricorsione su `set1` o su `set2`? Entrambe le scelte possono andar bene:

- se la ricorsione primaria è su `set1`, si costruirà il risultato conservando soltanto gli elementi di `set1` che non occorrono in `set2`. In questo caso occorre definire qual è il valore di `setminus [] set2` (cioè $\emptyset \setminus S_2$, se S_2 è l'insieme rappresentato da `set2`), e quello di `setminus (x::rest) set2`, assumendo di saper calcolare il valore di `setminus rest set2` (chiamata ricorsiva): a questo andrà aggiunto `x` soltanto se non occorre in `set2` (ricorsione *secondaria* su `set2` utilizzando `List.mem`).

- Se invece la ricorsione primaria è su `set2`, si dovranno eliminare ad uno ad uno gli elementi di `set2` da `set1`.

  Può essere utile definire allora una funzione ausiliaria `remove: 'a -> 'a list -> 'a list`, tale che `remove x set` riporti la lista che si ottiene eliminando la prima occorrenza di `x` da `set`, se `set` contiene `x`, altrimenti riporterà `set` stesso. Se `set` rappresenta un insieme, contiene al massimo un'occorrenza di `x`, quindi, se `set` rappresenta l'insieme S, `remove x set` rappresenterà l'insieme $S \setminus \{x\}$.

  Per definire `setdiff` si deve quindi definire quindi il valore di `setdiff set1 []` e quello di `setdiff set1 (x::rest)`. Per il secondo caso, si può in realtà lavorare in due modi diversi:

  - assumendo di saper calcolare il valore di `setdiff set1 rest` (chiamata ricorsiva), si elimina `x` da tale valore.
  - Si toglie prima `x` da `set1`, e poi si effettua la chiamata ricorsiva, sfruttando la proprietà $S_1 \setminus (\{x\} \cup S_2) = (S_1 \setminus \{x\}) \setminus S_2$.

2.3.3 Cosa significa "rappresentare"?

Un tipo di dati non è altro che un insieme di oggetti sul quale sono definite delle operazioni, quindi è una struttura algebrica. Supponiamo che $\mathcal{A} = \langle A, f_1, ..., f_n \rangle$ sia una struttua algebrica e $\mathcal{B} = \langle B, g_1, ..., g_n \rangle$ una struttura algebrica simile ad $\mathcal{A}$, cioè B è un insieme di oggetti, sul quale sono definite le operazioni $g_1, ..., g_n$, tali che il numero di argomenti di g_i è uguale al numero di argomenti di f_i, per ogni $i = 1, ..., n$.

Allora per definire una rappresentazione di $\mathcal{A}$ in $\mathcal{B}$ occorre determinare una relazione tra oggetti di A e oggetti di B ("a è rappresentato da b" o "b è un rappresentante di a") che abbia le seguenti proprietà:

1. ogni oggetto di A ha almeno un rappresentante in B (non si richiede che il rappresentante sia unico);

2. ogni oggetto di B rappresenta uno ed un unico elemento di A; di conseguenza, elementi distinti di A hanno rappresentanti distinti in B;

3. la rappresentazione "conserva la struttura": il risultato dell'operazione f_i eseguita in A su $a_1, ..., a_k$ è rappresentato dal risultato che si ottiene eseguendo l'operazione corrispondente g_i in B su rappresentanti degli oggetti $a_1, ..., a_k$.

Ad esempio, sia $\mathcal{A}$ l'insieme di tutti gli insiemi finiti di numeri naturali con l'operazione di unione, e $\mathcal{B}$ l'insieme delle liste di naturali con l'operazione di concatenazione. Allora $\mathcal{A}$ può essere rappresentato da $\mathcal{B}$ stabilendo che un insieme è rappresentato da qualsiasi lista contenente gli stessi elementi (ammettendo anche ripetizioni). Infatti: ogni insieme ha almeno un rappresentante, insiemi distinti hanno certamente rappresentanti distinti e gli elementi contenuti in `lista1 @ lista2` sono esattamente quelli contenuti nell'unione degli elementi di L_1 con gli elementi di L_2. Si noti che in questo caso un insieme è rappresentato da più di una lista, dato che nelle liste è rilevante l'ordine e la ripetizione di elementi.

Formalmente, una rappresentazione di $\mathcal{A}$ in $\mathcal{B}$ è un'applicazione φ da B a A tale che:

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

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

Considerimao l'esempio degli insiemi e delle liste e definiamo φ in modo tale che, per ogni lista `lst`, $\varphi($`lst`$) = \{x \mid x$ è un elemento di `lst`$\}$. Sicuramente abbiamo che per ogni insieme finito S esiste almento una lista che contiene gli stessi elementi. Inoltre: $\varphi($`lista1@lista2`$) = \varphi($`lista1`$) \cup \varphi($`lista2`$)$, dato che gli elementi contenuti in `lista1 @ lista2` sono quelli contenuti in `lista1` o in `lista2` (unione dei due insiemi).

2.3.4 Liste associative

Consideriamo ora il tipo di dati (astratto) *dizionario*: un dizionario è costituito da un insieme di elementi (o *valori*), ciascuno dei quali è univocamente identificato da una rispettiva *chiave* (elementi distinti hanno chiavi distinte).

Le operazioni sul tipo dizionario sono:

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

inserimento: dato un dizionario D, un valore v e una chiave k, costruire il dizionario che si ottiene da D aggiungendo l'elemento v con chiave k. Se in D già esiste un elemento con chiave k, questo verrà rimosso.

cancellazione: dato un dizionario D e una chiave k, costruire il dizionario che si ottiene da D cancellando l'elemento con chiave k, se esiste (o altrimenti D stesso).

Un dizionario si può rappresentare mediante una *lista associativa*, cioè una $(\alpha \times \beta)$ `list`: gli elementi della lista saranno coppie `(chiave,valore)`. Ovviamente, per poter implementare le funzioni corrispondenti alle operazioni su dizionari, le chiavi devono essere di un tipo con uguaglianza.

Ad esempio, la lista `[("pippo",100);("pluto",50);("topolino",30)]` è una lista associativa, che associa interi a stringhe (alla stringa `"pippo"` è associato l'intero 100, a `"pluto"` l'intero 50, ecc.).

In altre parole possiamo dire che una lista associativa rappresenta una funzione parziale che è definita solo per un insieme finito di valori. Se una lista associativa contiene più di una coppia con la stessa chiave, il valore associato alla chiave è quello della coppia "più superficiale", cioè quella che nella lista si trova prima dell'altra (le liste associative sono quindi gestite come un ambiente). Ad esempio, il valore associato a "pippo" nella lista `[("pippo",100);("pluto",50);("pippo",4);("topolino",30)]` è `100` (il primo legame nasconde l'altro).

L'operazione fondamentale sulle liste associative è la ricerca del valore associato a una data chiave (che "rappresenta" l'operazione di ricerca in un dizionario). Tale operazione viene tradizionalmente chiamata `assoc` e può essere implementata come segue:

```
(* eccezione sollevata nel caso in cui alla chiave della ricerca non
   sia associato alcun valore *)
exception NotFound

(* assoc: 'a -> ('a * 'b) list -> 'b *)
(* assoc k lista = valore associato a k in lista, se esiste;
                   solleva NotFound se non esiste *)
exception NotFound
let rec assoc k = function
    [] -> raise NotFound
  | (k1,v)::rest ->
       if k = k1 then v
       else assoc k rest
```

Le liste associative sono molto utilizzate in programmazione funzionale, ed infatti la funzione `assoc` è definita nel modulo `List` della libreria standard di OCaml. In caso di fallimento, `List.assoc` solleva l'eccezione predefinita `Not_found`.

Per rappresentare l'operazione di inserimento in un dizionario, si implementerà una funzione `inserisci: 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list`, tale che `inserisci k v assoc_list` riporti la lista associativa che si ottiene inserendo la coppia `(k,v)` in `assoc_list` – sostituendo (o sovrascrivendo) l'eventuale elemento già esistente con chiave **k**.

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

```
assoc k (inserisci k v assoc_list) = v
```

Sostituire effettivamente un'eventuale coppia `(k,v')` già presente in `assoc_list` è "costoso", in quanto occorre controllare tutta la lista. Poiché la ricerca riporta il primo valore trovato associato alla chiave della ricerca, è sufficiente inserire la nuova coppia prima delle altre.

```
(* inserisci: 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list *)
(* inserisci k v assoc_list = funzione parziale che associa v a k, e
      ad ogni altra chiave k1 diversa da k, associa lo stesso valore
      associato a k1 in assoc_list *)
let inserisci k v assoc_list =
      (k,v)::assoc_list
```

La scelta di rappresentare i dizionari mediante liste associative (che possono contenere diverse coppie con la stessa chiave) facilita l'operazione di inserimento, ma rende quella di cancellazione (eseguita raramente nei dizionari) più costosa, in quanto occorre scandire tutta la lista per eliminare tutte le coppie con la chiave data:

```
(* cancella : 'a -> ('a * 'b) list -> ('a * 'b) list *)
(* cancella k assoc_list = lista che si ottiene cancellando da
      assoc\_list tutte le coppie con chiave  k.
      Se nessuna coppia ha chiave k, riporta assoc_list stessa *)
let rec cancella k = function
    [] -> []
  | (k',v)::rest ->
      if k=k'
      then cancella k rest
      else (k',v)::cancella k rest
```

2.4 La tecnica del *backtracking*

In questo paragrafo consideriamo una tecnica algoritmica generale che risulta utile per risolvere una vasta classe di problemi. Presenteremo ora esempi che considerano le liste, e l'applicheremo più avanti ad alcuni importanti problemi su alberi e grafi.

Prendiamo in considerazione problemi di ricerca di una soluzione *ammissibile* all'interno di uno *spazio di soluzioni*. Tali problemi possono essere risolti con un approccio *brute force*, mediante *algoritmi di enumerazione* o *ricerca esaustiva*: le soluzioni possibili vengono generate ad una ad una, a partire da una soluzione "iniziale", mediante un'operazione che, applicata a una soluzione corrente, genera la successiva. Via via che si genera una soluzione, essa viene sottoposta al test di ammissibilità. La ricerca termina non appena si trova una soluzione ammissibile oppure quando viene generata e controllata "l'ultima soluzione".

Un approccio più vantaggioso può essere costituito in molti casi dall'adozione di uno schema risolutivo diverso, noto come *backtracking*. I problemi che possono essere affrontati con la tecnica del *backtracking* consistono nella ricerca di una soluzione, dove la soluzione può essere costruita incrementalmente e, ad ogni stadio, è possibile scegliere diverse alternative. Perché abbia senso utilizzare il *backtracking*, il problema

deve avere la seguente caratteristica: è possibile accorgersi, in alcuni casi, che una via di soluzione esaminata è fallimentare anche prima di averla percorsa fino in fondo.

Si può immaginare di rappresentare lo spazio di ricerca delle soluzioni come un albero, in cui ogni nodo rappresenta un passo verso la soluzione, le foglie sono situazioni non ulteriormente espandibili; alcune di esse possono rappresentare una soluzione (successo) altre no (fallimento). L'esplorazione dello spazio di ricerca avviene allora in profondità: si percorre l'albero seguendo, diciamo, sempre il primo figlio, fino ad arrivare ad una foglia; se questa è una soluzione allora è il risultato della ricerca, altrimenti si ritorna ad uno dei nodi intermedi e si "ritratta" la scelta fatta di procedere lungo il primo ramo; si sceglie il secondo ramo e si prosegue fino ad una foglia, e così via, fino ad aver trovato una soluzione o ad aver esplorato tutto l'albero senza averne trovata alcuna. In quest'ultimo caso la ricerca fallisce.

Il vantaggio degli algoritmi di backtracking rispetto alla ricerca esaustiva consiste nel fatto che spesso non è necessario giungere fino ad una foglia per determinare che un ramo non può portare ad alcuna soluzione: si può riconoscere che un nodo intermedio (una soluzione parziale) è destinato a portare a fallimento, comunque si proceda nella ricerca. La ricerca lungo tutti i rami che passano per quel nodo può dunque essere interrotta, riducendo notevolmente lo spazio delle soluzioni da esplorare.

Un esempio intuitivo di ricerca con *backtracking* consiste nel problema di trovare un percorso in un labirinto. Immaginiamo di avere un labirinto rettangolare, suddiviso in caselle, come quello rappresentato qui:

	X			X					X
	X	X		X	X				
					X			X	
X		X	X		X			X	
					X	X		X	
	X	X	X	X	X	X		X	
	X				X			X	
			X		X		X	X	
X	X	X	X				X		
								X	

Una X in una casella rappresenta un "muro". Il problema consiste nel trovare un attraversamento dalla casella in alto a sinistra a quella in basso a destra. Un algoritmo di ricerca esaustiva genererebbe ad uno ad uno tutti i possibili percorsi completi nel labirinto, dalla casella iniziale a quella finale, scartando via via quelli che contengono muri.

Un algoritmo di *backtracking* costruisce invece il cammino incrementalmente. Quando si arriva ad un muro, si ritorna all'ultima scelta fatta e si tenta una via alternativa. Assumiamo che, ad ogni stadio i nella costruzione del percorso, è stato percorso il cammino $(x_1, ..., x_i)$ e si è nella posizione x_i. Se x_i contiene un muro, allora la ricerca (a questo stadio) fallisce. Altrimenti, se x_i è la casella di uscita dal labirinto, allora è stata trovata una soluzione: $(x_1, ..., x_i)$. Altrimenti, siano $y_1, ..., y_n$ le caselle adiacenti a x_i. Si sceglie un y_k e si cerca un strada verso l'uscita a partire da y_k – il cammino percorso è ora $(x_1, ..., x_i, y_k)$ e siamo in posizione y_k. Se il tentativo ha successo e riporta la soluzione $(x_1, ..., x_i, y_k, ...)$, quella è la soluzione. Altrimenti si tenta con un

diverso y_k. Se da nessuno degli y_k si può arrivare all'uscita, allora anche la ricerca a partire da x_i fallisce. Naturalmente occorre fare attenzione ai cicli: occorre tenere traccia delle strade già percorse per non rischiare di ripercorrerle all'infinito.

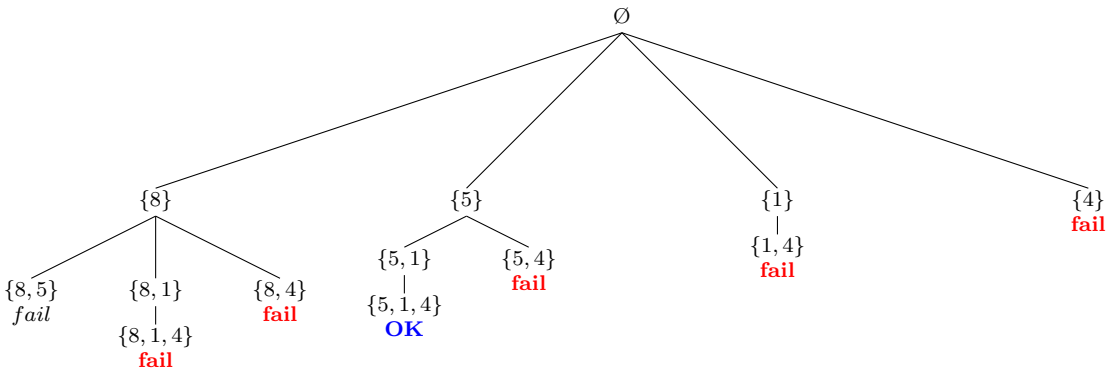
2.4.1 Ricerca di sottoinsiemi

Presentiamo qui un primo semplicissimo esempio di algoritmo di *backtracking*. Dato un insieme S di interi non negativi e un intero N, il problema consiste nel determinare un sottoinsieme Y di S tale che la somma degli elementi di Y sia uguale a N.

Un algoritmo di ricerca esaustiva per risolvere tale problema consiste nella generazione, ad uno ad uno, di tutti i sottoinsiemi dell'insieme S, e nel controllare, per ciascuno di essi, se la somma dei suoi elementi è uguale a N. L'algoritmo termina con successo appena si trova un sottoinsieme "soluzione", e fallisce quando sono stati generati tutti i sottoinsiemi senza aver trovato soluzioni. L'implementazione di tale algoritmo richiede ovviamente un metodo per generare ad uno ad uno i sottoinsiemi di un insieme, cioè si deve determinare un ordinamento dei sottoinsiemi di S e metodi per determinare qual è il primo sottoinsieme, l'ultimo sottoinsieme ed un metodo per generare il sottoinsieme "successivo" ad un altro.

Per avere un'idea di come si può affrontare invece il problema con la tecnica del *backtracking*, consideriamo ad esempio l'insieme $S = \{8, 5, 1, 4\}$ e $N = 10$ e costruiamo incrementalmente la soluzione. Inizialmente, la soluzione parziale è l'insieme vuoto, e, procedendo per stadi, aggiungiamo un elemento alla volta. Ci sono due casi di fallimento: la somma degli elementi nella soluzione parziale corrente è maggiore di N (S non contiene numeri negativi), oppure è minore di N e non ci sono altri elmenti da potervi aggiungere. Il caso di successo è invece unico: la somma degli elementi nella soluzione parziale corrente è uguale a N.

Assumiamo che gli elementi da aggiungere alla soluzione parziale corrente siano sempre considerati in quest'ordine: $8, 5, 1, 4$. Lo spazio di ricerca delle soluzioni si può rappresentare come un albero, le cui foglie sono situazioni non ulterirmente espandibili; alcune di esse possono rappresentare una soluzione (successo) altre no (fallimento):



Al livello della radice abbiamo quattro scelte possibili: includere 8 nella soluzione, includervi 5, includervi 1 o includervi 4. L'esplorazione dello spazio di ricerca avviene

in profondità: il primo nodo esaminato è $\{8\}$ che non è né un nodo di successo né di fallimento. Quindi questa foglia ha dei figli, che si ottengono aggiungendo a $\{8\}$ gli elementi di S che ancora non sono stati considerati. La foglia $\{8, 5\}$ rappresenta un fallimento, perché $8 + 5 > 10$. Allora "si torna indietro" (*backtrack*) al nodo genitore e si prosegue sull'altro suo figlio, $\{8, 1\}$, che si può ancora espandere, ma solo aggiungendo 4 (le soluzioni che contengono sia 8 che 5 sono già state esaminate), e il nodo $\{8, 1, 4\}$ rappresenta ancora un fallimento, così come il terzo figlio del nodo $\{8\}$. Allora viene ritrattata la scelta di includere 8 nella soluzione e viene esaminata l'alternativa $\{5\}$. Si noti che in tutto il sottoalbero che ha radice $\{5\}$ l'elemento 8 non viene mai considerato, in quanto le soluzioni che includono 8 sono già state esaminate nel sottoalbero alla sua sinistra. Il primo sottoalbero del nodo $\{5\}$ porta infine a una soluzione. Se si volessero produrre tutte le soluzioni, la ricerca proseguirebbe sugli altri sottoalberi in modo analogo. Si noti che, ad esempio, il nodo $\{5, 4\}$ rappresenta un fallimento, in quanto $5 + 4 < 10$ e non vi sono altri elementi da aggiungere: le soluzioni che contengono 8 sono già state esaminate, così come quelle che contengono sia 5 che 1.

In generale, la ricerca procede per stadi. Ad ogni stadio abbiamo i seguenti dati:

- una soluzione parziale *Solution*;
- un insieme $Altri \subseteq S$ tra cui si possono scegliere elementi da aggiungere a *Solution*.

Inizialmente: $Solution = \emptyset$, $Altri = S$. Ad ogni stadio:

- Se la somma degli elementi di *Solution* è maggiore di N, allora **fallimento**.
- Se la somma degli elementi di *Solution* è uguale a N, allora **successo** con *Solution*.
- Altrimenti (somma degli elementi di *Solution* minore di N):
  - Se *Altri* è vuoto, allora **fallimento**.
  - se *Altri* non è vuoto, scegliere un elemento x di *Altri*, e considerare le alternative:
    - mettere x nella soluzione, cioè cercare una soluzione con $\{x\} \cup Solution$ e $Altri - \{x\}$
    - non mettere x nella soluzione, cioè cercare una soluzione con *Solution* e $Altri - \{x\}$

Per implementare questo algoritmo, definiamo innanzitutto la funzione di utilità `sumof`, che riporta la somma degli interi nella lista suo argomento, dato che il criterio per scartare una soluzione parziale è: la somma dei suoi elementi e' maggiore di N.

```
(* sumof: int list -> int
   sumof [x1;...xn] = x1+...+xn *)
let rec sumof = function
    [] -> 0
  | x::rest -> x+sumof rest
```

Dato che l'input del problema principale e i dati a disposizione ad ogni stadio della ricerca sono diversi, utilizzeremo una funzione ausiliaria che ha come argomenti una soluzione parziale e una lista di argomenti tra cui scegliere altri elementi da aggiungervi.

```
(* eccezione per segnalare il fallimento *)
exception NotFound

(* subset_search : int list -> int -> int list *)
  (* aux : int list ->  int list -> int list *)
  (* funzione ausiliaria che implementa il ciclo *)
  (* aux solution altri = lista@solution, dove lista contiene numeri
        scelti da altri, tali che sumof(lista@solution) = n *)
let subset_search set n =
  let rec aux solution altri =
    let somma = sumof solution in
    (* caso di fallimento *)
    if somma > n then raise NotFound
    else if somma=n then solution (* terminazione con successo *)
         else match altri with
           [] -> (* non si possono aggiungere altri elementi *)
                 raise NotFound
         | x::rest ->
         (* proviamo ad aggiungere x, se troviamo una soluzione, bene,
            altrimenti proviamo senza x *)
              try aux (x::solution) rest
              with NotFound -> aux solution rest
  in aux [] set
```

Si noti l'uso dell'espressione `try ... with ...` per implementare la parte dell'algoritmo in cui viene effettivamente eseguito il backtracking: ho due alternative, `A` e `B` e le provo in quest'ordine. Se `A` mi fornisce una soluzione, sono contento, altrimenti provo l'alternativa `B`: `try A with ... -> B`.

Possiamo raffinare questa soluzione evitando di dover ogni volta calcolare la somma degli elementi nella soluzione parziale, e conservando invece il valore complessivo che deve raggiungere la somma degli elmenti che vanno ancora aggiunti alla soluzione parziale (inizialmente N). Quando si aggiunge x a una soluzione parziale, si sottrae x stesso a tale valore. In questo caso non è necessario avere a disposizione ad ogni stadio la soluzione parziale già costruita, in quanto gli elementi possono essere aggiunti al ritorno dalla ricorsione: abbiamo un insieme S e un intero N; per trovare una soluzione, scegliamo un elemento x di S e cerchiamo una soluzione Y per $S \setminus \{x\}$ e $N - x$ (stesso tipo di problema); se esiste una tale soluzione, allora si riporta $Y \cup \{x\}$. Altrimenti, cerchiamo una soluzione senza x, cioè risolviamo lo stesso tipo di problema per $S \setminus \{x\}$ e N stesso; se si trova una soluzione, questa sarà anche la soluzione da riportare. Se entrambe le alternative falliscono, la ricerca fallisce.

Modificando l'algoritmo in questo modo, non c'è bisogno di definire una funzione ausiliaria, in quanto ad ogni stadio si dispone dello stesso tipo di dati del problema principale: un insieme di elementi che si possono inserire nella soluzione e un totale da raggiungere.

```
(* subset_search : int list -> int -> int list *)
(* seconda versione *)
let rec subset_search set n =
  match set with
    [] ->  (* non ci sono elementi da aggiungere *)
      if n>0 (* andrebbero aggiunti elementi *)
      then raise NotFound
      else []  (* n non sara' mai negativo *)
  | x::rest ->
      if x>n (* n-x sarebbe negativo *)
      then subset_search rest n (* non si puo' mettere x nella soluzione *)
      else
        try x::subset_search rest (n-x)
            (* x viene inserito nella soluzione al ritorno
               dalla ricorsione *)
        with NotFound -> subset_search rest n
            (* x non viene inserito nella soluzione *)
```

Possiamo modificare facilmente la funzione precedente, in modo che venga riportata una lista di tutte le possibili soluzioni, implementando l'algoritmo seguente: per cercare tutte le soluzioni per S e N:

- Se $S = \emptyset$ e $N > 0$, non vi sono soluzioni, quindi si riporta $\emptyset$.
- Se $S = \emptyset$ e $N = 0$, c'è un'unica soluzione, $\emptyset$, quindi si riporta l'insieme che la contiene: $\{\emptyset\}$.
- Se $S \neq \emptyset$, allora scegliere un elemento $x \in S$ e:
  - se $x > N$ allora x non si può inserire nelle soluzioni, quindi si riporta il risultato della ricerca per $S \setminus \{x\}$ e N stesso;
  - altrimenti, le soluzioni da riportare sono tutte quelle che si ottengono inserendo x in ogni soluzione trovata per $S \setminus \{x\}$ e $N - x$, oltre a quelle che risolvono il problema per $S \setminus \{x\}$ e N stesso (che non contengono x)

Per implementare questo algoritmo, definiamo una funzione di utilità che applicata a un elemento `x` e una lista di liste, riporta la lista che si ottiene inserendo `x` in testa a ogni elemento della lista:

```
(* mapcons : 'a -> 'a list list -> 'a list list *)
(* mapcons x [lst_1;...;lst_n] = [x::lst_1;...;x::lst_n] *)
let rec mapcons x = function
    [] -> []
  | lst::rest -> (x::lst)::mapcons x rest
```

Il motivo per cui abbiamo chiamato così questa funzione sarà chiaro in seguito. Possiamo allora definire come segue una funzione che riporta tutte le soluzioni al problema della ricerca di sottoinsiemi:

```
(* all_subsets: int list -> int -> int list list *)
let rec all_subsets set tot =
  match set with
    [] ->
      if tot>0
      then []
      else [[]]
  | x::rest ->
      if x>tot
      then all_subsets rest tot
      else (mapcons x (all_subsets rest (tot-x)))
           @ (all_subsets rest tot)
```

2.4.2 Uso di operatori logici per implementare il *backtracking*

Presentiamo ora un esempio che mostra come di fatto il *backtracking* possa anche essere implementato mediante gli operatori logici, e non soltanto mediante espressioni `try ... with ...`

Sia data una lista di coppie, che rappresenta la relazione "a X piace Y". Ad esempio la lista `[("maria","birra");("carlo","vino");("maria","vino");("carlo","maria");("antonio","maria")]`. Stabiliamo che X ama Y (secondo una data relazione che rappresenta cosa piace a chi) se e solo se a X e Y piace uno stesso oggetto Z. Ad esempio, se la lista è quella data sopra, Carlo ama se stesso, Maria ed Antonio.

Si vuole scrivere una funzione `loves` che, data una lista di coppie `pairs: ('a*'b) list`, e due elementi `x` e `y` di tipo `'a` stabilisca (riportando un booleano) se esiste un valore `z` tale che `pairs` contenga le coppie `(x,z)` e `(y,z)`. La soluzione consiste nel cercare in `rel` una coppia della forma `(x,z)`. Non appena viene trovata, si "sceglie" `z` come possibile secondo elemento e si verifica se `rel` contiene la coppia `(y,z)`. Se questo è falso, la scelta di `z` viene ritrattata e si prosegue la scansione di `rel`. Tale scansione viene realizzata, nella funzione `loves` sotto definita, dalla funzione ausiliaria `aux`.

```
(* loves : ('a * 'b) list -> 'a -> 'a -> bool *)
(* loves pairs x y = true se x ama y secondo la lista pairs *)
(* aux : (string * string) list -> bool *)
(* aux lista = true se lista contiene una coppia (x,z) tale che
               (y,z) e' un elemento di pairs *)
let loves pairs x y =
  let rec aux = function
      [] -> false
    | (a,z)::rest ->
        (a=x && List.mem (y,z) pairs) || aux rest
```

```
in aux pairs
```

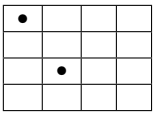
La necessità di utilizzare una funzione ausiliaria è dovuta al fatto, che una volta scelto `z`, la coppia `(y,z)` va ricercata nell'intera lista `pairs`, e non soltanto nell'argomento di `aux`.

2.4.3 Il problema delle otto regine

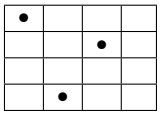
Si tratta di un puzzle classico: data una scacchiera $n \times n$ e n regine (tradizionalmente $n = 8$), si devono posizionare le regine sulla scacchiera in modo che nessuna di esse sia sotto scacco, cioè, comunque si considerino due regine, esse non devono essere né sulla stessa riga, né sulla stessa colonna, né sulla stessa diagonale.

Poiché evidentemente ogni colonna (e ogni riga) deve contenere esattamente una regina, la ricerca di una soluzione può avvenire come segue. Si posiziona una regina sulla prima colonna (scegliendo una riga). Poi si posiziona la seconda regina sulla seconda colonna, in qualsiasi riga che non sia sotto scacco da parte della prima regina. Poi si posiziona la terza regina sulla terza colonna, in qualsiasi riga che non la ponga sotto scacco da parte delle prime due regine. Si continua così fino a che non sono state sistemate tutte le regine. Può accadere, comunque, in qualsiasi stadio, che sia impossibile sistemare una regina nella colonna $m + 1$, perché tutte le righe di quella colonna sono sotto scacco. Allora si *ritorna indietro* (*backtrack*) e si riconsiderano le scelte fatte per le precedenti m regine. Si esegue una scelta diversa e si prova di nuovo.

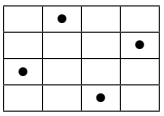
Ad esempio, per il problema delle 4 regine, si posizionerà inizialmente la prima sulla riga 1 e la seconda sulla riga 3, ma questa scelta non porta ad alcuna soluzione per la terza regina:



La scelta di porre la seconda regina sulla riga 3 viene quindi ritrattata, e si sistema sulla riga 4. In questo modo è possibile mettere la terza regina sulla seconda riga, ma non vi sono soluzioni per l'ultima:



Quindi si torna indietro di nuovo. La terza regina non può essere messa in altre posizioni che non siano sotto scacco, quindi si torna indietro ancora di un passo. Ma anche per la seconda regina non ci sono più alternative, quindi si sposta di nuovo la prima regina, ponendola in riga 2. Questa scelta porta infine a una soluzione.



Per formalizzare questo metodo di soluzione, occorre innanzitutto determinare una rappresentazione per scacchiere che possono contenere al più una regina per colonna e in cui le colonne sono riempite a partire da sinistra. Se m è la dimensione della scacchiera, utilizziamo una lista di lunghezza m, dove l'elemento in posizione i rappresenta la riga in cui è posizionata la regina in colonna i. Ad esempio, la lista $[4;6;1]$ rappresenta la scacchiera in cui sono state sistemate tre regine, nelle caselle $(4,1)$, $(6,2)$ e $(1,3)$ (la casella (n,k) è quella nella riga n della colonna k).

Iniziamo quindi a risolvere sottoproblemi utili. Innanzitutto occorrerà determinare, date due caselle, se esse rappresentano posizioni sotto scacco reciproco. Per far ciò osserviamo che gli indici delle caselle che si trovano in una diagonale parallela alla principale sono: $(1,k)$, $(2,k+1)$,...,$(n,k+n-1)$, quindi la differenza tra gli indici è costante; allora due posizioni (i,j) e (m,n) sono sulla stessa diagonale se $i-j=m-n$. Analogamente, la somma degli indici delle caselle che si trovano nella diagonale secondaria, o in una diagonale parallela alla secondaria, è costante (la riga aumenta mentre la colonna diminuisce); quindi se $i+j=m+n$ allora (i,j) e (m,n) sono sulla stessa diagonale.

La funzione `scacco` si applica a due coppie di indici e riporta `true` se le corrispondenti caselle sono sulla stessa riga o su una stessa diagonale (la colonna non viene controllata perché la rappresentazione che abbiamo adottato garantisce che questo non possa accadere). Quindi:

```
(* scacco : int * int -> int * int -> bool *)
(* scacco (i,j) (m,n) = true se due regine, posizionate sulle caselle
                        (i,j) e (m,n) sono sotto scacco reciproco
let scacco (i,j) (m,n) =
  (i=m) || (i+j=m+n) || (i-j = m-n)
```

Dobbiamo ora definire una funzione che, data la rappresentazione di una scacchiera in cui sono state posizionate m regine e un intero k, determini se la $m+1$-esima regina si può mettere nella riga k. Per determinare se $(k,m+1)$ è sotto scacco rispetto a una delle regine posizionate precedentemente, la rappresentazione della scacchiera come lista di numeri di riga `[`$\mathtt{r}_1$`,...,`$\mathtt{r}_m$`]` andrà convertita in una rappresentazione come lista di caselle: `[(`$\mathtt{r}_1$`,1),...,(`$\mathtt{r}_m$`,m)]`. La nuova posizione $(k,m+1)$ andrà poi confrontata con ogni casella di questa lista. Per eseguire la conversione di rappresentazione, utilizziamo la funzione predefinita `List.combine: 'a list -> 'b list -> ('a * 'b) list`, che, applicata a due liste di uguale lunghezza, riporta la lista delle coppie contenenti gli elementi nella stessa posizione (e solleva un'eccezione se le due liste hanno lunghezze diverse). Ad esempio, `List.combine [1;2;3] ["a";"b";"c"] = [(1,"a"); (2,"b"); (3,"c")]`. Se non fosse predefinita nel modulo List, potremmo definirla così:

```
(* combine : 'a list -> 'b list -> ('a * 'b) list *)
(* combine [x1;...;xn] [y1;...;yn] = [(x1,y1);...;(xn,yn)] *)
let rec combine lst1 lst2 =
  match (lst1,lst2) with
    ([],[]) -> []
```

```
  | (x::rest1,y::rest2) -> (x,y)::combine rest1 rest2
  | _ -> failwith "combine"
```

Si noti che controllare prima la lunghezza delle due liste sarebbe inutile e costoso: se le due lunghezze sono uguali, ogni lista andrebbe "scandita" per calcolarne la lunghezza, e poi le due liste andrebbero di nuovo scandite contemporaneamente per costruire il risultato. Questo è anche un esempio di funzione che si applica a due liste in cui la ricorsione avviene contemporaneamente sulle due liste.

`List.combine` dovrà essere applicata alla rappresentazione `board` della scacchiera e alla lista `[1;...;n]` dove n è la lunghezza di `board`. La lista `[1;...;n]` può essere generata dalla funzione `upto` definita a pagina 60.

Ora, se `(n,m+1)` è la posizione della nuova regina, si deve controllare che `not(scacco (n,m+1) (i,j))` sia vero per ogni posizione `(i,j)` delle regine già sistemate:

```
(* safe : int list -> int -> bool *)
(* safe board k = true se la prossima regina si puo' mettere alla
                  riga k
(* aux: (int * int) list -> bool *)
(* funzione ausiliaria che controlla tutte le caselle della lista *)
(* aux lista_caselle = true se nessuna casella in lista_caselle mette
                       sotto scacco una regina nella casella
                       (m+1,k), dove m e' la lunghezza di board *)
let safe board k =
  let m = List.length board in
  let rec aux = function
      [] -> true
    | (i,j)::rest ->
        not (scacco (i,j) (k,m+1)) && aux rest
  in aux (List.combine board (upto 1 m))
```

Passiamo ora a considerare il problema principale: se dobbiamo riempire una scacchiera di dimensione n, procederemo per stadi. Ad ogni stadio abbiamo i seguenti dati:

- un intero `c` (inizialmente 1) che è la colonna per la quale si deve determinare una posizione buona per la `c`-esima regina;
- un intero `r` (inizialmente 1) che è la riga a partire dalla quale si può iniziare la ricerca (le righe precedenti sono già state provate e la ricerca ha fallito);
- `board`, una lista di interi, inizialmente vuota, che rappresenta la scacchiera costruita fino a quel momento, per le colonne `1,...,c-1`.

Per implementare il backtracking, abbiamo quindi bisogno di una funzione ausiliaria, che chiamiamo `search`. La funzione solleva un'eccezione (`NotFound`) se non esistono soluzioni possibili a partire dalla scacchiera `board`. Altrimenti riporta la prima soluzione trovata. La ricerca avviene controllando prima se la posizione `r` è `safe` per la colonna `c`, data la scacchiera `board`. In tal caso viene cercata una soluzione per

la parte mancante di scacchiera, mediante la chiamata ricorsiva, che applica `search` a `c+1`, `1` e `board@[r]`. Se tale soluzione esiste, essa stessa è il valore della funzione. Altrimenti, se la chiamata ricorsiva genererà un errore, verrà cercata un'altra posizione per la `c+1`-esima regina, a partire dalla riga `r+1`. Si termina con successo quando `c` è maggiore della dimensione della scacchiera (non ci sono più regine da sistemare) e con fallimento quando `r` è maggiore della dimensione della scacchiera (non ci sono altre righe possibili per la colonna corrente).

```
exception NotFound

(* queens n = soluzione al problema delle n regine *)
(* search : int -> int -> int list -> int list *)
(* search c r board = soluzione al problema delle n regine
            che estende la scacchiera board dove sono gia' state
            sistemate c-1 regine, e con la posizione della c-esima
            regina in una riga maggiore o uguale a r *)
let queens n =
  let rec search c r board =
    if c > n then board
    else
      if r > n then raise NotFound
      else
        if safe board r
        then
          try search (c+1) 1 (board@[r])
          with NotFound -> search c (r+1) board
        else search  c (r+1) board
  in search 1 1 []
```

2.4.4 Attraversamento della palude

Presentiamo in questa sezione una versione semplificata della ricerca di un percorso in un labirinto, in cui non è necessario tenere conto delle caselle già attraversate. Sia data una matrice P, di dimensione $n \times m$ e i cui valori sono booleani, che rappresenta una zona paludosa in cui i T (true) rappresentano aree di terraferma e gli F (false) sabbie mobili (non transitabili). Per *passaggio* si intende una sequenza di aree di terraferma adiacenti che attraversano la palude da qualsiasi casella della colonna di sinistra (indice di colonna pari a 1) a una qualsiasi casella della colonna più a destra (indice di colonna pari a m), ovviamente passando solo su caselle di terraferma. I passaggi a cui si è interessati sono di lunghezza m, cioè, in un passaggio, da una casella in colonna j si va ad un'area in colonna $j + 1$. La casella in posizione (i, j) si considera quindi adiacente alle caselle in posizione $(i-1, j+1)$, $(i, j+1)$ e $(i+1, j+1)$ – se esistono. Ad esempio, la matrice 1 rappresenta una palude senza passaggi, mentre la matrice 2 ne rappresenta una con un passaggio.

Matrice 1	Matrice 2
T F F T F F	T F F T F F
T T F F F F	T F F F F F
F F T T F F	F T F F F T
T T F F F F	F F T T T F
T T T F T T	F T F F F F

Si vuole scrivere una funzione OCaml che, data una palude P, verifichi l'esistenza di almeno un passaggio in P e riporti una lista rappresentante un passaggio, se esiste, la lista vuota altrimenti. Se esiste più di un passaggio, la funzione ne riporta uno qualsiasi.

Per risolvere il problema occorre innanzitutto decidere come rappresentare la palude. Possiamo rappresentarla mediante una lista di liste di booleani, dove ciascun elemento rappresenta una riga della matrice. Ad esempio, le matrici 1 e 2 saranno rappresentate, rispettivamente, dai valori `palude1` e `palude2` così definiti:

```
let palude1 = [[true;false;false;true;false;false];
               [true;true;false;false;false;false];
               [false;false;true;true;false;false];
               [true;true;false;false;false;false];
               [true;true;true;false;true;true]];;

let palude2 = [[true;false;false;true;false;false];
               [true;false;false;false;false;false];
               [false;true;false;false;false;true];
               [false;false;true;true;true;false];
               [false;true;false;false;false;false]];;
```

Durante la ricerca del percorso dovremo certamente conoscere le dimensioni della palude. Definiamo quindi:

```
(* nrighe: 'a list -> int *)
(* nrighe p = numero di righe della palude p *)
let nrighe = List.length

(* ncol: 'a list -> int *)
(* ncol p = numero di colonne della palude p, assumendo che
            ogni riga di p abbia lo stesso numero di colonne *)
let ncol palude = List.length (List.hd palude)
```

Dovremo inoltre accedere al contenuto delle caselle. Data una palude `p = [riga`$_1$`,...,riga`$_n$`]`, il contenuto della casella in posizione `(r,c)` (riga `r` e colonna `c`) è l'elemento in posizione `c` della lista `riga`$_r$, che è a sua volta l'elemento in posizione `r` della lista `p`. Per accedere agli elementi di una lista in una posizione data, utilizziamo la funzione predefinita `List.nth: 'a list -> int -> 'a`, tale che, `List.nth lst n` = elemento della lista `lst` in posizione `n`. Come è abituale, le posizioni degli elementi in una lista sono contati a partire da 0 (definire per esercizio una funzione equivalente `nth`).

La funzione `lookup` consente di accedere quindi al contenuto di una casella in una palude:

```
(* lookup: 'a list list -> int -> int -> 'a *)
(* lookup p r c = contenuto della casella (r,c) nella palude p *)
let lookup p r c =
  try
    List.nth (List.nth p (r-1)) (c-1)
  with _ -> failwith "lookup"
```

La definizione evita che venga sollevata l'eccezione predefinita `Failure "nth"`.

Possiamo ora affrontare il problema principale. Ci sono diversi *punti di backtracking*: innanzitutto si possono scegliere diverse caselle iniziali nella prima colonna; inoltre, ad ogni stadio della ricerca, possiamo scegliere diverse caselle adiacenti a quella corrente. Nella soluzione che proponiamo, due diverse funzioni ausiliarie si occupano di questi casi: la funzione `scan_first_row` prova le diverse caselle iniziali della prima colonna come punti di partenza, e per ciascuna di esse cerca un percorso richiamando la funzione `search`. Quest'ultima cerca un percorso da una casella data `(r,c)`: fallisce se la casella è fuori matrice o è paludosa, e nel caso in cui invece la casella sia nell'ultima colonna, riporta la lista `[(r,c)]`: il percorso per uscire a partire da `(r,c)` è in questo caso costituito dalla casella stessa.

Se invece si può provare ad andare avanti, `search` cerca un percorso a partire da una delle caselle adiacenti a `(r,c)`, considerandole come diverse alternative possibili su cui eseguire il backtracking. Se una di queste ha successo e riporta un percorso di uscita `path` (che parte dunque da una casella adiacente a `(r,c)`), il percorso per uscire a partire da `(r,c)` si ottiene inserendo `(r,c)` in testa a `path`. Il percorso viene quindi ricostruito passo passo al ritorno dalla ricorsione.

```
exception NotFound

(* explore: bool list list -> (int * int) list *)
(* explore palude riporta un percorso nella palude data, se esiste,
          solleva NotFound altrimenti *)
(* search: int ->  int -> (int * int) list *)
(* search r c = percorso a partire dalla casella (r,c) se esiste,
          solleva NotFound altrimenti *)
(* scan_first_col: int -> (int * int) list *)
(* scan_first_col r = percorso a partire da una casella nella prima
          colonna con indice di riga maggiore o uguale a r *)
let explore palude =
  (* calcoliamo innanzitutto il massimo indice valido per le righe
     e le caselle della palude *)
  let rows = nrighe palude in
  let cols = ncol palude in
  let rec search r c =
    if r < 1 || r > rows        (* r e' fuori range *)
       || c < 1 || c > cols     (* c e' fuori range *)
       || not (lookup palude r c) (* la casella (r,c) e' paludosa *)
    then raise NotFound
    else (* (r,c) e' dentro la palude e in (r,c) c'e' terra *)
      if c = cols            (* siamo sulla colonna di uscita *)
      then [(r,c)]  (* percorso per uscire dalla palude a partire
                       dalla casella (r,c) *)
      else
        (* si provano le diverse alternative a partire dalle caselle
           adiacenti a (r,c)  *)
        try (r,c)::search (r-1) (c+1)
                     (* cerca in diagonale verso l'alto *)
        with _  ->  (* eccezioni catturate:
                       NotFound o Failure "lookup" *)
          try (r,c)::search r (c+1)
                       (* cerca in orizzontale *)
          with _ ->
            (r,c)::search (r+1)(c+1)
                      (* cerca in diagonale verso il basso *)
  in
  let rec scan_first_col r =
    if r>rows then raise NotFound
    else
      try search r 1
      with NotFound -> scan_first_col (r+1)
  in scan_first_col 1
```

2.5 Ordinamento di liste

In questo paragrafo consideriamo l'algoritmo di ordinamento per fusione (*merge sort*), applicandolo al caso delle liste: data una lista di elementi di un tipo ordinato, si vuole ottenere un'altra lista, ordinata e contenente gli stessi elementi.

2.5.1 La tecnica "Divide et Impera"

Alcuni dei più efficienti algoritmi di ordinamento, tra cui anche il merge sort, sono basati su una tecnica generale per la progettazione di algoritmi, denominata *divide et impera*. Questa si riferisce al modo di affrontare un problema che consiste nel cercare di suddividerlo in sottoproblemi, risolvere questi ultimi e combinare le soluzioni in modo da risolvere il problema originario. Se i sottoproblemi hanno la stessa struttura del problema originario, allora è possibile usare la stessa procedura per risolverli in modo ricorsivo.

Affinché questa tecnica funzioni, sono necessari due requisiti: il primo è che i sottoproblemi siano più semplici del problema originario; il secondo è che, dopo un numero finito di suddivisioni, si incontri un sottoproblema che possa essere risolto immediatamente.

Seguendo la tecnica divide et impera, dunque, il problema viene scomposto in opportuni sottoproblemi. Data una funzione definita per un input di misura n, questo metodo divide il problema in k sottoproblemi distinti ($1 < k \leq n$), ciascuno dei quali ha misura strettamente minore di n. Una volta che questi sottoproblemi sono risolti, si deve trovare un metodo opportuno per combinare le soluzioni ottenute e determinare la soluzione finale. Se i k sottoproblemi sono ancora troppo grandi si può applicare ancora il metodo *divide et impera* a ciascuno di essi. La riapplicazione di questo metodo ai sottoproblemi è espressa in modo naturale da algoritmi ricorsivi e permette di ottenere sottoproblemi, tutti dello stesso tipo, sempre più piccoli da risolvere, fino a che non si ottengono problemi sufficientemente piccoli per i quali esiste una soluzione specifica e che quindi non vengono suddivisi ulteriormente.

Vediamo come esempio un problema concettualmente semplice che può essere risolto in modo più efficiente con l'uso di questa tecnica. Dati due interi x e k, vogliamo calcolare x^k. Un modo ovvio è basato sulla proprietà

$$x^k = \underbrace{x \cdot x \cdot \ldots \cdot x}_{k \text{ volte}} = x \cdot (\underbrace{x \cdot \ldots \cdot x}_{k-1 \text{ volte}})$$

```
(* power: int -> int -> int *)
(* power x k = k-esima potenza di x, per k non negativo *)
let rec power x = function
    0 -> 1
  | k -> x * (power x (k-1))
```

In questo caso per calcolare x^k si eseguono $k - 1$ moltiplicazioni.

Anche questo metodo è in fondo basato sul principio "divide et impera", in quanto soddisfa entrambi i requisiti visti sopra: in primo luogo, quando l'applicazione `power`

`x k` effettua la chiamata ricorsiva, il secondo argomento è costituito da un numero minore di `k`; in secondo luogo, quando `power x` è applicata a 0, il calcolo termina immediatamente. In fondo un qualunque programma ricorsivo si può vedere come un'applicazione "limite" della tecnica *divide et impera*. Le applicazioni più efficienti e significative di questa tecnica si hanno tuttavia quando la dimensione dell'argomento decresce pi'u rapidamente, cioè quando i sottoproblemi sono notevolmente più piccoli del problema originario.

Ad esempio, calcolando x^k in un modo meno ovvio, si può ridurre il numero di moltiplicazioni da eseguire. Sfruttando la proprietà $x^{2n} = (x^2)^n$ si ottiene ad esempio che

$$x^{10} = (x^2)^5 = (x^2) \cdot (x^2)^4 = (x^2) \cdot ((x^2)^2)^2$$

In generale

$$x^1 = x$$
$$x^{2n} = (x^2)^n$$
$$x^{2n+1} = x \cdot (x^2)^n$$

La funzione che calcola l'elevamento a potenza è così definita:

```
(* power: int -> int -> int *)
let rec power x = function
    0 -> 1
  | 1 -> x
  | k ->
      if k mod 2 = 0
      then power (x*x) (k/2)
      else x * (power (x*x) (k/2))
```

Questo algoritmo è notevolemente più efficiente di quello precedente, che richiede di eseguire sempre k moltiplicazioni per calcolare la k-esima potenza di un numero. Calcoliamone la complessità, sempre in termini del numero di moltiplicazioni necessarie per ottenere la soluzione. A tale scopo distinguiamo due casi, il caso migliore e quello peggiore. Il caso migliore è quello in cui k è una potenza di due e sono necessarie $log_2 k$ moltiplicazioni. Verifichiamolo con un esempio, calcolando 2^{16}:

```
power 2 16 = power (2*2) 8 = power 4 8
           = power (4*4) 4 = power 16 4
           = power(16*16) 2 = power 256 1
           = power(256*256) 1 = power 65536 1 = 65536
```

Sono state eseguite 4 moltiplicazioni e $4 = log_2 16$

Nel caso peggiore la potenza è sempre un numero dispari, come ad esempio per calcolare 2^{15}. In questo caso il numero di moltiplicazioni eseguite è $2 \cdot \lfloor log_2 k \rfloor$. Infatti, per calcolare 2^{15}:

```
power 2 15 = 2 * power (2*2) 7 = 2 * power 4 7
           = 2 * 4 * power (4*4) 3 = 8 * power 16 3
           = 8 * 16 * power (16*16) 1 = 128 * power 256 1
           = 128 * 256 = 32768
```

vengono eseguite 6 moltiplicazioni $= 2 \cdot \lfloor log_2 15 \rfloor$.

2.5.2 Ordinamento per fusione

L'algoritmo di ordinamento per fusione (o *merge sort*), che rappresenta una potente applicazione della tecnica divide et impera, in cui una lista viene ordinata "suddividendo" il problema in due problemi analoghi, ma di misura dimezzata. Possiamo dividere la lista in due liste di dimensione pressappoco uguale e con elementi scelti arbitrariamente. Fatto questo, ciascuna delle due liste di lunghezza dimezzata viene ordinata separatamente. Per completare l'ordinamento della lista originaria di n elementi, fondiamo le due liste ordinate mediante l'algoritmo di fusione illustrato più avanti.

L'algoritmo può essere descritto semplicemente solo in modo ricorsivo: le due sottoliste vengono ordinate mediante una chiamata ricorsiva; il passo base si ha quando la lista è vuota oppure ha un solo elemento. Ad esempio, se si vuole ordinare la lista `[4;2;6;1;5;7;3]`, si ordinano (con lo stesso metodo) le due liste `[4;6;5;3]` e `[2;1;7]`, ottenendo `[3;4;5;6]` e `[1;2;7]`. La fusione di queste liste produrrà `[1;2;3;4;5;6;7]`.

La lista da ordinare deve quindi essere innanzitutto suddivisa in due liste la cui lunghezza differisce al massimo di 1. La suddivisione può seguire un qualunque criterio. Ad esempio si può prendere la prima metà degli elementi nella prima lista e il resto nella seconda lista. Nel programma che sviluppiamo, la suddivisione viene eseguita in modo diverso: nella prima lista si mettono tutti gli elementi in posizione pari, e dall'altra tutti quelli in posizione dispari.

```
(* split: 'a list -> 'a list * 'a list *)
(* split lista riporta una coppia di liste, di lunghezza
     "pressappoco" uguale, nelle quali sono suddivisi gli
     elementi di lista *)
let rec split = function
    [] -> ([],[])
  | [x] -> ([x],[])
  | x::y::rest ->
      let (prima,seconda) = split rest
      in  (x::prima,y::seconda)
```

Si noti che con questo modo di eseguire la suddivisione, la lista di input viene "scandita" una volta sola. Mentre, se mettiamo nella prima lista gli elementi della prima metà della lista, quest'ultima viene "scandita" una volta per calcolarne la lunghezza, e poi la sua prima metà deve essere "scandita" un'altra volta per prenderne gli elementi e lasciare nella seconda lista il resto.

L'algoritmo di fusione ha come input due liste ordinate e produce una lista ordinata. "Fondere" significa infatti produrre, a partire da due liste ordinate, un'unica lista ordinata contenente tutti gli elementi delle due liste date e soltanto quelli. Per esempio, date le liste [1;2;7;7;9] e [2;4;7;8], la lista prodotta è [1;2;2;4;7;7;7;8;9]. Da notare che non ha senso parlare di "fusione" di due liste che non siano già ordinate.

Un modo semplice per fondere due liste consiste nell'esaminarle a partire dal primo elemento di entrambe. A ogni passo, scegliamo il più piccolo tra i due elementi in testa alle liste, lo inseriamo nella lista risultato e lo cancelliamo dalla lista da cui è stato scelto, facendo emergere un nuovo "primo" elemento. Quando i primi elementi sono uguali, può essere inserito uno qualsiasi di essi. Il procedimento termina quando gli elementi di una delle due liste sono stati completamente trasferiti nella nuova lista. I rimanenti elementi dell'altra lista, essendo già ordinati, vengono attaccati in coda alla lista risultato.

La funzione seguente implementa la fusione.

```
(* merge: 'a list -> 'a list -> 'a list *)
(* fusione di due liste ordinate *)
let rec merge prima seconda =
  match (prima,seconda) with
    ([],_) -> seconda
  | (_,[]) -> prima
  | x::restx,y::resty ->
      if x <= y
      then x::merge restx (y::resty)
      else y::merge (x::restx) resty
```

Segue la funzione principale:

```
(* mergesort: 'a list -> 'a list *)
(* sort lst = lista ordinata contenente gli stessi elementi di lst *)
let rec sort = function
    [] -> []
  | [x] -> [x]
  | lst ->
      let (prima,seconda) = split lst
      in merge (sort prima) (sort seconda)
```

Si noti, sia in questa definizione che in quella di `split`, l'uso del pattern per coppie nelle espressioni `let`, che permette di evitare l'uso dei selettori.

2.5.3 Ordinamento con test parametrico

Utilizzando funzioni d'ordine superiore possiamo definire funzioni di ordinamento con un parametro che specifica il test da effettuare. Ad esempio, utilizzando la definizione dei predicati `leq` (minore o uguale) e `geq` (maggiore o uguale) sugli interi, si può definire un *merge sort* utilizzabile sia per ordinare una lista in ordine crescente dei suoi elementi sia in ordine decrescente:

```
(* leq: 'a -> 'a -> bool *)
(* leq x = essere minore di x *)
let leq x y =  y <= x

(* geq: 'a -> 'a -> bool *)
(* geq x = essere maggiore di x *)
let geq x y =  y >= x
```

Per definire una funzione di ordinamento per fusione con test parametrico, dobbiamo modificare la funzione `merge` e quella principale, dandogli un parametro in più che rappresenta la relazione d'ordine:

```
(* merge: ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list *)
(* merge test prima seconda = fusione delle due liste prima e seconda,
         secondo la relazione d'ordine test *)
let rec merge test prima seconda =
  match (prima,seconda) with
    ([],_) -> seconda
  | (_,[]) -> prima
  | x::restx,y::resty ->
      if test y x
      then x::merge test restx (y::resty)
      else y::merge test (x::restx) resty

(* sort: ('a -> 'a -> bool) -> 'a list -> 'a list *)
(* sort test lst = lista ordinata secondo la relazione d'ordine test
                   contenente gli stessi elementi di lst *)
let rec sort test = function
    [] -> []
  | [x] -> [x]
  | lst ->
      let (prima,seconda) = split lst
      in merge test (sort test prima) (sort test seconda)

# let lista = [13;2;78;29;35;43;28;7];;
val lista : int list = [13; 2; 78; 29; 35; 43; 28; 7]
# sort leq lista;;
- : int list = [2; 7; 13; 28; 29; 35; 43; 78]
# sort geq lista;;
- : int list = [78; 43; 35; 29; 28; 13; 7; 2]
```

Si noti che il tipo di `merge` e `sort` è polimorfo, quindi il tipo di ordinamento può essere qualisasi. Ad esempio, possiamo ordinare una lista di liste a seconda della loro lunghezza:

```
(* longer: 'a list -> 'b list -> bool *)
(* longer lst  = essere piu' lunga di lst *)
```

```
let longer lst altra =
   List.length lst <= List.length altra

# sort longer [[1;2;3;4];[1;2];[];[1];[1;2;3;4;5]];;
- : int list list = [[1; 2; 3; 4; 5]; [1; 2; 3; 4]; [1; 2]; [1]; []]
```

Oppure possiamo ordinare una lista di coppie secondo valori non crescenti dei secondi elementi delle coppie:

```
(* minore : 'a * 'b -> 'c * 'b -> bool *)
(* minore coppia = avere il secondo elemento minore del secondo
                elemento di coppia *)
let minore (_,x) (_,y) = x > y

# sort minore [('a',10);('b',3);('c',7);('d',4)];;
- : (char * int) list = [('b', 3); ('d', 4); ('c', 7); ('a', 10)]
```

2.5.4 La funzione sort del modulo List

Il modulo List della libreria standard di OCaml contiene diverse funzioni di ordinamento, tra cui: `sort`:

```
List.sort:  ('a -> 'a -> int) -> 'a list -> 'a list
```

La sua descrizione nel manuale è la seguente: "Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller. For example, `compare` is a suitable comparison function.". In altri termini, l'argomento che rappresenta la relazione d'ordine non è una funzione di tipo `'a -> 'a -> bool`, come abbiamo fatto nel paragrafo precedente, ma una funzione che riporta un `int`, che rappresenta il risultato del confronto. La funzione predefinita `compare` si comporta appunto in questo modo:

```
# compare 10 1;;
- : int = 1
# compare 1 10;;
- : int = -1
# compare "pippo" "pluto";;
- : int = -1
```

Per ordinare una lista di coppie secondo valori non decrescenti dei secondi argomenti, utilizzando `List.sort`, dobbiamo applicarla ad una funzione di confronto opportuna, che possiamo definire utilizzando la stessa `compare` predefinita:

```
(* compare_pairs: 'a * 'b -> 'c * 'b -> int *)
(* compare_pairs (a,b) (c,d) = 0 se b=d, 1 se b>d e -1 se b<d *)
let compare_pairs (_,x) (_,y) = compare x y;;
```

```
# compare_pairs ('b', 3) ('d', 4);;
- : int = -1

# List.sort compare_pairs  [('a',10);('b',3);('c',7);('d',4)];;
- : (char * int) list = [('b', 3); ('d', 4); ('c', 7); ('a', 10)]
```

Per ordinare in senso decrescente una lista di interi:

```
# List.sort (fun x y -> -(compare x y)) [13;2;78;29;35;43;28;7];;
- : int list = [78; 43; 35; 29; 28; 13; 7; 2]
```

L'espressione della forma `fun x y -> E` è un'abbreviazione per `fun x -> fun y -> E`, ed equivale a `function x -> function y -> E`. La differenza tra `fun` e `function` è che un'espressione `function` può usare il pattern matching, mentre le espressioni `fun` no. Ma le espressioni della forma `function x -> function y -> E` non si possono abbreviare in `function x y -> E`.

2.6 Funzioni di ordine superiore sulle liste

In questo paragrafo introduciamo alcune funzioni di ordine superiore (anche chiamati *funzionali*) che rappresentano operazioni generali sulle liste. Esse sono tutte predefinite nel modulo `List` della libreria standard.

2.6.1 Map

Vi sono alcune operazioni sulle liste che possono essere definite secondo uno schema molto simile. Ad esempio, le funzioni `mapdouble` e `mapsquare`, sotto definite, applicate a una lista, riportano la lista che risulta dall'applicazione, rispettivamente, della funzione `double` e `square` agli elementi della lista.

```
(* double: int -> int *)
(* double x = doppio di x *)
let double x = 2*x

(* mapdouble : int list -> int list *)
(* mapdouble [n1;...;nk] = [2*n1;...;2*nk]
let rec mapdouble = function
    [] -> []
  | x::rest -> double x::mapdouble rest

(* square: int -> int *)
(* square x = quadrato di x *)
let double x = x*x

(* mapsquare : int list -> int list *)
(* mapsquare [n1;...;nk] = [square n1;...;square nk] *)
```

```
let rec mapsquare = function
    [] -> []
  | x::rest -> square x :: mapsquare rest
```

Analoga struttura hanno le definizioni di `mapfirst` e `mapcons`:

```
(* mapfirst : ('a * 'b) list -> 'a list *)
(* mapfirst [(a1,b1);...;(ak,bk)] = [a1;...;ak] *)
let rec mapfirst = function
    [] -> []
  | (x,_)::rest -> x::mapfirst rest

(* mapcons : 'a -> 'a list list -> 'a list list *)
(* mapcons x [x1;...;xn] = [x::x1;...;x::xn] *)
let rec mapcons x = function
    [] -> []
  | lst::rest -> (x::lst)::mapcons x rest
```

Le operazioni che in questo caso sono applicate a tutti gli elementi della lista argomento sono, rispettivamente, `fst` e l'operazione di inserimento di `x` in testa.

Possiamo generalizzare e definire la funzione di ordine superiore `map`:

```
(* map : ('a -> 'b) -> 'a list -> 'b list *)
(* map f [x1;...;xn] = [f x1;...;f xn] *)
let rec map f = function
    [] -> []
  | x::rest -> f x :: map f rest
```

Di fatto:

```
mapdouble  =  map double
mapsquare  =  map square
mapfirst   =  map fst
```

Se inoltre definiamo l'operazione di inserimento in testa in forma currificata:

```
(* cons : 'a -> 'a list -> 'a list *)
let cons x lista = x::lista
```

(`cons x` è la funzione che, applicata a una lista, riporta la lista che si ottiene inserendovi `x` in testa), allora

```
mapcons x  =  map (cons x)
```

2.6.2 Filter

Un'altra operazione generale è quella di "filtrare" una lista riportando solo gli elementi che godono di una certa proprietà. Ad esempio::

```
(* positive_sublist : int list -> int list *)
(* positive_sublist lista = elementi positivi di lista *)
let rec positive_sublist = function
    [] -> []
  | x::rest ->
       if x>0 then x::positive_sublist rest
       else positive_sublist rest

(* even_sublist : int list -> int list *)
(* even_sublist lista = elementi pari di lista *)
let rec even_sublist = function
    [] -> []
  | x::rest ->
       if x mod 2 = 0 then x::even_sublist rest
       else even_sublist rest
```

Generalizzando, possiamo definire la funzione di ordine superiore `filter` che, applicata a un predicato `p` e una lista `lst`, riporta la sottolista di `lst` costituita da tutti e soli gli elementi per i quali vale `p`:

```
(* filter : ('a -> bool) -> 'a list -> 'a list *)
(* filter p lista = elementi di lista che soddisfano p *)
let rec filter p =  function
    [] -> []
  | x::rest ->
       if p x then x::filter p rest
       else filter p rest
```

Quindi:

```
positive_sublist  =  filter (function x -> x>0)
even_sublist      =  filter (function x -> x mod 2 = 0)
```

2.6.3 For_all

Consideriamo ora il predicato che, applicato a una lista di interi, determina se i suoi elementi sono tutti positivi:

```
(* all_positive : int list -> bool *)
(* all_positive lst = tutti gli elementi di lst sono positivi *)
let rec all_positive = function
    [] -> true
  | x::rest -> x>0 && all_positive rest
```

Anche l'operazione di verificare se tutti gli elementi di una lista godono di una determinata proprietà è molto generale e si può definire in OCaml come il predicato di ordine superiore `for_all`:

```
(* for_all : ('a -> bool) -> 'a list -> bool *)
(* for_all p lst = p vale per tutti gli elementi di lst *)
let rec for_all p = function
    [] -> true
  | x::rest -> p x && for_all p rest
```

Il funzionale `for_all` converte un predicato di `'a` in un predicato di `'a list`. Si noti che, dato che l'`&&` è valutato in modo "pigro", il calcolo di un `for_all` termina appena si incontra un elemento che non gode della proprietà `p`.

Utilizzando `for_all`, si può definire `all_positive` come segue:

```
let all_positive = for_all (function x -> x>0)
```

Come ulteriore esempio, consideriamo la definizione del predicato "non appartiene":

```
(* nonmem: 'a -> 'a list -> bool *)
(* nonmem = opposto di mem
   nonmem x lst = x non appartiene a lst *)
let rec nonmem x = function
    [] -> true
  | y::rest -> x <> y && nonmem x rest
```

Il predicato può anche essere definito utilizzando il funzionale `for_all`, considerando che `nonmem x lst` vale se tutti gli elementi di `lst` hanno la proprietà di essere diversi da `x`, e:

```
nonmem x  =  for_all (function y -> x <> y)
          =  for_all (function y -> (<>) x y)
          =  for_all ((<>) x)
```

(si riguardi il paragrafo 1.15.2 per la notazione `(<>)`). Quindi:

```
let nonmem x = for_all ((<>) x)
```

Come esempio d'uso di `for_all`, consideriamo il problema di verificare se due insiemi, rappresentati mediante liste, sono disgiunti. La definizione della funzione che proponiamo è in stile dichiarativo: due insiemi X e Y sono disgiunti se ogni elemento di X è diverso da ogni elemento di Y. Possiamo definire la funzione `disjoint` senza usare funzioni di ordine superiore come segue:

```
(* disjoint : 'a list -> 'a list -> bool *)
(* x diverso da tutti gli elementi di lst = not (List.mem x lst) *)
let rec disjoint set = function
    [] -> true
  | x::rest ->
      not (List.mem x set) && disjoint set rest
```

Ma osservando che `disjoint set1 set2` vale se tutti gli elementi di `set2` hanno la proprieta‘ di non appartenere a `set1` (o viceversa), utilizzando `List.for_all`, possiamo definire:

```
let disjoint set1 set2 =
  List.for_all
    (function x -> not (List.mem x set1)) set2
```

O ancora, osservando che `not (List.mem x set1)` equivale a dire che `x` è diverso da tutti gli elementi di `set1`, cioè `List.for_all ((<>) x) set1`:

```
let disjoint set1 set2 =
    List.for_all (function x -> List.for_all (different x) set1) set2
```

Tutti gli elementi `x` di `set2` hanno la proprietà di essere diversi da tutti gli elementi di `set1`.

2.6.4 Exists

Un altra funzione che converte predicati di tipo `'a -> bool` in predicati di tipo `'a list -> bool` è `exists`, che determina se una lista contiene un elemento che gode di una data proprietà:

```
(* exists : ('a -> bool) -> 'a list -> bool *)
let rec exists p = function
    [] -> false
  | x::rest -> p x || exists p rest
```

Anche in questo caso osserviamo che il calcolo di un `exists` termina appena si incontra un elemento che gode della proprietà `p`.

Riguardando la definizione di `mem` data a pagina 60, si vede chiaramente che si tratta di un caso particolare di `exists`, e si potrebbe dunque definire come segue:

```
(* mem: 'a -> 'a list -> bool *)
(* mem x lst = esiste un elemento di lst uguale a x *)
let mem x lst = List.exists ((=) x) lst
```

o semplicemente:

```
let mem x = List.exists ((=) x)
```

2.6.5 Fold_left

Si consideri le seguenti definizioni *tail recursive* delle funzioni `sumof` e `string_concat`:

```
(* sumof : int list -> int *)
(* sumof [n1;...;nk] = n1 + ... + nk *)
(* aux:  int -> int list -> int *)
(* aux result [n1;...;nk] = result + n1 + ... + nk
```

```
let sumof lst =
  let rec aux result = function
      [] -> result
    | x::rest -> aux (result + x) rest
  in aux 0 lst

(* string_concat:  string list -> string *)
(* string_concat [s1;...;sk] = s1 ^ ... ^ sk *)
(* aux: string -> string list -> string *)
(* aux result [s1;...;sk] = result ^ s1 ^ ... ^ sk *)
let string_concat lst =
  let rec aux result = function
      [] -> result
    | s::rest -> aux (result ^ s) rest
  in aux "" lst
```

Le definizioni hanno la stessa struttura: in particolare, quel che differenzia le due funzioni ausiliarie è soltanto la funzione (somma o concatenazione di stringhe) mediante la quale si ottiene il nuovo "risultato parziale" nella chiamata ricorsiva. Possiamo allora astrarre da tale operazione e definire un funzionale che cattura la struttura comune delle due definizioni:

```
(* fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a *)
(* fold_left f a [x1;...;xk] = f(...(f (f a x1) x2) ... xk) *)
let rec fold_left f a = function
    [] -> a
  | x::rest -> fold_left f (f a x) rest
```

La funzione `fold_left` cattura la struttura di una definizione tail recursive sulle liste. La "funzione principale" si otterrà inizializzando opportunamente l'accumulatore, che, normalmente, è l'elemento neutro (a sinistra) dell'operazione `f`. Ad esempio possiamo definire:

```
let sumof = List.fold_left (+) 0
let string_concat = List.fold_left (^) ""
```

2.7 Programmare con funzioni di ordine superiore

2.7.1 Segmenti iniziali di una lista

Si vuole definire una funzione `inits: 'a list -> 'a list list`, che, applicata a una lista `lst`, riporti la lista dei segmenti iniziali di `lst`. Ad esempio, si avrà `inits [1;2;3;4] = [[1]; [1; 2]; [1; 2; 3]; [1; 2; 3; 4]]`.

Per definire la funzione, osserviamo che:

- La lista vuota non ha segmenti iniziali, quindi `inits [] = []`.

- Per il caso ricorsivo: sia `x::rest` la lista di cui si vogliono calcolare i segmenti iniziali. La chiamata ricorsiva su `rest` darà la lista dei segmenti iniziali di `rest`. Possiamo ragionare sull'esempio in cui `x::rest = [1;2;3;4]`. Il valore di `inits [2;3;4]` sarà `[[2]; [2; 3]; [2; 3; 4]]`. A ciascun elemento di questa lista di liste si dovrà inserire `1` in testa, e poi aggiungere la lista `[1]` al risultato. L'operazione di aggiungere un elemento in testa a ciascuna lista di una lista di liste si può effettuare utilizzando il funzionale `map` e la funzione `cons` già definita a pagina 93.

Quindi:

```
(* cons : 'a -> 'a list -> 'a list *)
let cons x lst = x::lst;;

(* inits : 'a list -> 'a list list *)
let rec inits = function
    [] -> []
  | x::rest -> [x]::List.map (cons x) (inits rest)
```

Utilizzando `inits` possiamo facilmente definire una funzione funzione `sublists: 'a list -> 'a list list`, tale che `sublists lst` = lista di tutte le sottoliste di `lst`. Per sottolista di una lista `lst` intendiamo una sequenza di elementi contigui in `lst`, nello stesso ordine. Ad esempio le sottoliste di `[1;2;3;4]` sono: `[1]`, `[1;2]`, `[1;2;3]`, `[1;2;3;4]`, `[2]`, `[2;3]`, `[2;3;4]`, `[3]`, `[3;4]` e `[4]`. Per il caso ricorsivo, osserviamo che se la lista ha la forma `x::rest`, le sue sottoliste sono tutti i suoi segmenti iniziali, più tutte le sottoliste di `rest`:

```
(* sublists : 'a list -> 'a list list *)
(* sublists lst = lista di tutte le sottoliste di lst *)
let rec sublists = function
    [] -> []
  | _::rest as lst -> inits lst @ sublists rest
```

2.7.2 Insieme delle parti

L'insieme delle parti (o insieme potenza) di un insieme S, $\mathcal{P}(S)$ (o 2^S) è l'iniseme di tutti i sottoinsiemi di S. Per esempio, se $S = \{1, 2, 3\}$:

$$\mathcal{P}(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

Si vuole definire una funzione `powerset: 'a list -> 'a list list`, che, applicata a una lista rappresentante un insieme S, riporti l'insieme potenza di S, cioè la lista di tutti i sottoinsiemi di S.

Per affrontare questo problema, osserviamo che:

- L'insieme vuoto ha un unico sottoinsieme: $\mathcal{P}(\emptyset) = \{\emptyset\}$. Quindi `powerset [] = [[]]` (la lista che contiene la lista vuota).

- Per il caso ricorsivo, se la lista ha la forma `x::rest`, i suoi "sottoinsiemi" sono tutti quelli di `rest` più quelli che si ottengono aggiungendo `x` a ciascuno di essi.

  Ad esempio, se `lst = [1;2;3]`, la chiamata ricorsiva su `[2;3]` riporterà `[[];[2];[3];[2;3]]` (o una sua permutazione). Per ottenere tutti i sottoinsiemi di `lst`, oltre a quelli di `rest`, si deve aggiungere `1` a ciascun elmento di `[[];[2];[3];[2;3]`, ottenendo `[[1];[1;2];[1;3];[1;2;3]`. In altri termini, l'insieme delle parti di `lst` è `[[];[2];[3];[2;3]] @ [[1];[1,2]; [1,3];[1,2,3]]`.

```
(* powerset : 'a list -> 'a list list *)
(* powerset lst = lista con tutti i sottoinsiemi di lst *)
let rec powerset = function
    [] -> [[]]
  | x::rest ->
        let powerset_rest = powerset rest
        in powerset_rest @ List.map (cons x) powerset_rest
```

L'uso dell'espressione `let` nel codice evita di calcolare due volte il risultato della chiamata ricorsiva.

2.7.3 Prodotto cartesiano

La funzione `cartprod: 'a list * 'b list -> ('a * 'b) list` , applicata a due liste (rappresentanti insiemi) `set1` e `set2`, riporta il prodotto cartesiano di `set1` e `set2`: la lista di tutte le coppie `(x,y)`, con `x` $\in$ `set1` e `y` $\in$ `set2`. Ragionando ricorsivamente sul primo argomento, osserviamo che:

- Se `set1` è vuoto, anche il prodotto cartesiano `set1` $\times$ `set2` è vuoto.
- Se `set1` ha la forma `x::rest`, allora `set1` $\times$ `set2` contiene:
  - tutte le coppie in `rest` $\times$ `set2`;
  - tutte le coppie che hanno `x` come primo elemento e un elemento di `set2` come secondo elemento.

Un sottoproblema da risolvere è dunque quello di costruire la lista `[(x,y1);...;(x,yn)]`, dato `x` e la lista `lst=[y1;...;yn]`. Ciò si può realizzare in modo molto semplice utilizzando `List.map`: ad ogni elemento `y` di `lst` si deve applicare la funzione che accoppia `x` con `y`: `function y -> (x,y)`. Quindi `List.map (function y -> (x,y))[y1;...;yn] = [(x,y1);...;(x,yn)]`.

Dunque `cartprod` può essere definita come segue:

```
(* cartprod : 'a list -> 'b list -> ('a * 'b) list *)
(* cartprod set1 set2 = prodotto cartesiano di set1 e set2 *)
let rec cartprod set1 set2 =
  match set1 with
    [] -> []
  | x::rest ->
      (List.map (function y -> (x,y)) set2) @ cartprod rest set2
```

2.7.4 Permutazioni

In questo paragrafo suggeriamo soltanto il modo di affrontare ricorsivamente il problema di definire una funzione `permut 'a list -> 'a list list` che calcoli tutte le possibili permutazioni della lista data.

Per il caso base, osserviamo che la lista vuota ha un'unica permutazione: se stessa. Quindi `permut [] = [[]]`.

Se la lista ha la forma `x::rest`, possiamo assumere di saper calcolare tutte le permutazioni di `rest`. In molti casi è utile ragionare su un esempio semplice, per capire come lavorare sul risultato della chiamata ricorsiva per ottenere quello desiderato. Supponiamo che la lista `x::rest` sia `[1;2;3]`. L'obiettivo è quello di ottenere la lista

```
[[1;2;3]; [2;1;3]; [2;3;1]; [1;3;2]; [3;1;2]; [3;2;1]]
```

(con gli elementi in qualsiasi ordine). La chiamata ricorsiva su `[2;3]` darà come risultato `[[2;3];[3;2]]`. Come possiamo da questo ottenere l'obiettivo? Possiamo osservare che i primi 3 elementi della lista obiettivo "corrispondono" a `[2;3]`: ciascuno di essi si ottiene da `[2;3]` inserendovi `1`, in tutte le posizioni possibili. E analogamente gli altri 3 elementi della lista obiettivo corrispondono alla lista `[3;2]`.

Un sottoproblema utile è dunque quello di definire una funzione

```
interleave: 'a -> 'a list -> 'a list list,
```

tale che `interleave x lst` = lista di tutte le liste che risultano dall'inserimento di `x` in `lst` (in ogni posizione).

Tornando al nostro esempio, se applichiamo la funzione `interleave 1` a tutti gli elementi (`List.map`) del risultato della chiamata ricorsiva su `[2;3]`, otterremo una lista di liste: `[[[1;2;3];[2;1;3];[2;3;1]]; [[1;3;2];[3;1;2];[3;2;1]]]`. Per ottenere l'obiettivo sarà sufficiente "appiattire" (`List.flatten`) questo risultato.

Quindi in generale `permut (x::rest)` si può ottenere applicando `List.flatten` al risultato dell'applicazione di `interleave x` a tutti gli elementi di `permut rest`.

Capitolo 3

Strutture dati fondamentali

3.1 Definizione di nuovi tipi

Il sistema dei tipi di OCaml si può estendere mediante la definizione di nuovi tipi. Per definire un tipo occorre specificare:

1. un **nome** per il tipo
2. come costruire i valori del tipo, specificando qual è l'insieme dei **costruttori** del tipo. I costruttori possono essere:
   (a) costanti (occorre specificarne il nome);
   (b) costruttori funzionali: operazioni che, applicate ad argomenti di un determinato tipo, riportano valori del nuovo tipo; occorre specificarne il nome e il tipo degli argomenti.

Il caso più semplice è costituito dai *tipi enumerati*, i cui valori sono un insieme finito di costanti. Ad esempio, con la dichiarazione:

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

vengono definiti quattro nuovi valori: `Su, Giu, Destra, Sinistra`.

La parola chiave `type` introduce una dichiarazione di tipo, ed è seguita dal nome del tipo (nell'esempio `direction`). Dopo il segno `=` seguono i nomi delle costanti, separati dalla barra verticale.

L'esempio che segue mostra che con una dichiarazione di tipo vengono introdotti nuovi valori, prima sconosciuti.

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

```
# type seme = Bastoni | Coppe | Denari | Spade;;
type seme = | Bastoni | Coppe | Denari | Spade
# Denari;;
- : seme = Denari
```

In OCaml i nomi di costruttori di un tipo devono iniziare con lettera maiuscola; questo li distingue dagli altri identificatori.

I nuovi valori possono essere utilizzati nella costruzione di tipi compositi, quali liste, tuple, ecc., ed utilizzati per formare espressioni complesse. Ad esempio:

```
# [Coppe;Denari;Spade];;
- : seme list = [Coppe; Denari; Spade]

type valore = Asso | Due | Tre | Quattro | Cinque
               | Sei | Sette | Fante | Cavallo | Re;;

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

I tipi enumerati sono tipi con eguaglianza, è cioè possibile applicare il test di eguaglianza a espressioni del tipo:

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

Le nuove costanti introdotte nella definizione del tipo, in quanto costruttori, possono essere utilizzate nei *pattern*. È quindi possibile definire una funzione per casi distinguendo i diversi pattern, come ad esempio:

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

Tipi più complessi si possono definire utilizzando *costruttori funzionali*. Si tratta di funzioni che, applicate ad opportuni argomenti, riportano valori del nuovo tipo. Oltre al nome del costruttore, nella dichiarazione di tipo, occorre specificare il tipo

dei suoi argomenti. Supponiamo ad esempio di voler rappresentare il mazzo delle carte francesi, costituite dalle carte normali, caratterizzate da seme e valore, e dai jolly. Possiamo allora definire un tipo enumerato per rappresentare i semi, come per le carte napoletane, e il tipo `carta` come segue:

```
type seme = Cuori | Quadri | Fiori | Picche
type carta =
    Jolly
  | Card of int * seme
```

Il tipo carta è costituito dal valore `Jolly` e da tutti i valori della forma `Card(n,s)`, dove `n` è un `int` e `s` un `seme`. In altri termini, `Card` è il costruttore che, applicato a una coppia di tipo `int * seme`, riporta un valore di tipo `carta`:

```
# let c = Card(2,Picche);;
val c : carta = Card (2, Picche)
# Jolly = c;;
- : bool = false
# c = (2,Picche);;
Characters 4-14:
  c = (2,Picche);;
      ^^^^^^^^^^
Error: This expression has type 'a * 'b
       but an expression was expected of type carta
```

Come si vede da questi esempi, anche il tipo `carta` è un *equality type* ed è comunque un tipo distinto dal tipo `int * seme`.

Il costruttore `Card` si comporta come una funzione, che trasforma coppie in carte. I costruttori sono parte della descrizione del tipo: essi determinano il modo canonico di descriverne i valori. A differenza delle funzioni "normali", comunque, i costruttori funzionali devono sempre essere applicati ai propri argomenti (non è possibile, ad esempio, darli come argomenti di funzioni di ordine superiore).

Anche i costruttori funzionali possono occorrere nei pattern. Ad esempio:

```
(* somma : carta -> carta -> int *)
(* somma c1 c2 = somma dei valori numerici di c1 e c2,
                 fallisce se c1 e/o c2 e' un Jolly *)
let somma c1 c2 =
  match (c1,c2) with
    (Card(n,_), Card(k,_)) -> n+k
  | _ -> failwith "somma"
```

Come ulteriore esempio, consideriamo la dichiarazione seguente, che definisce il tipo `number` come l'unione disgiunta di `int` e `float`:

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

Sul tipo `number` è possibile definire operazioni aritmetiche come la somma, utilizzando il pattern matching:

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

Nell'esempio che segue è utilizzato un costruttore costante e diversi costruttori funzionali:

```
  type egiziano = Faraone
                | Scriba of string
                | Sacerdote of string
                | Mercante of string
                | Artigiano of string
                | Contadino of string
                | Schiavo of string;;
```

Ogni valore del nuovo tipo è costruito mediante un costruttore e valori costruiti con costruttori diversi sono distinti. Ad esempio, `Scriba "Sethi"` e `Schiavo "Sethi"` sono valori distinti. In altri termini, ogni "egiziano" ha un unico *nome* semplice, il valore corrispondente.

Se tutti i tipi componenti di un nuovo tipo sono *equality types*, anche il tipo composto è un *equality type*.

La dichiarazione di un tipo può essere ricorsiva, e corrisponde alla definizione induttiva di un insieme. Ad esempio, è possibile definire il tipo `nat` dei naturali come segue:

```
type nat = Zero | Succ of nat
```

I valori del tipo sono `Zero` e tutte le espressioni della forma `Succ(Succ ...(Succ Zero)...)`. Sul tipo `nat` possiamo definire ricorsivamente la somma, il prodotto e le altre operazioni aritmetiche. Ad esempio:

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

Nuovi tipi possono essere definiti utilizzando tipi dichiarati precedentemente, ad esempio il tipo delle liste di naturali può essere definito mediante la dichiarazione:

```
type natlist = Nil | Cons of nat * natlist

# Cons(Succ Zero,Cons(Zero,Nil));;
- : natlist = Cons (Succ Zero, Cons (Zero, Nil))
```

La definizione di un tipo può essere parametrica, utilizzando variabili di tipo e costruttori polimorfi. Ad esempio, se il tipo `'a list` non fosse predefinito, potremmo definirlo come segue:

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

In questo caso, `Cons` è un costruttore polimorfo e `mylist` è un costruttore di tipi. Nella dichiarazione il nome del tipo è preceduto da una variabile di tipo (`'a`), che rappresenta il parametro del costruttore di tipi `mylist`).

È possibile che un tipo sia paramentrico rispetto a più elementi. Ad esempio, il tipo delle liste associative (vedi paragrafo 2.3.4) è `('a * 'b) list`. Se vogliamo definire per esse un nuovo tipo, esso sarà parametrico rispetto al tipo delle chiavi e al tipo dei valori. Per definire un nuovo tipo `assoc` per le liste associative, con costruttore `Assoc`, la dichiarazione avrà questa forma:

```
type ('a,'b) assoc = Assoc of ('a * 'b) list
```

3.1.1 Abbreviazioni di tipo

La parola chiave `type` si può utilizzare anche per introdurre un nuovo nome per denotare un tipo già esistente. Ad esempio, nel caso della rappresentazione delle carte napoletane, potremmo definire un nuovo nome per le coppie di tipo `valore * seme`:

```
type carta_napoletana = valore * seme
```

Con questa dichiarazione, non definiamo alcun nuovo tipo, ma semplicemente introduciamo un nuovo nome per il tipo già esistente `valore * seme`.

Al contrario, una dichiarazione della forma

```
type carta_napoletana = Card of valore * seme
```

introduce effettivamente un tipo che prima non esisteva in OCaml, cioè un nuovo insieme di valori.

3.1.2 Il tipo `'a option`

In OCaml è predefinito il tipo polimorfo `'a option`, così definito:

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

Esso praticamente rappresenta l'unione dei valori di tipo α con il valore `None`. È utile quando non si vuole far ricorso alle eccezioni per segnalare l'inapplicabilità di una funzione, perché non si vuole la propagazione automatica delle eccezioni.

Ad esempio, possiamo definire una versione di `List.assoc` che riporta `None` anziché sollevare un'eccezione nel caso in cui alla chiave data non sia associato alcun valore:

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

In questo modo è possibile utilizzare `List.map` per ottenere una lista con tutti i valori associati ad una data lista di chiavi:

```
# let alist = [(0,"pippo"); (1,"pluto"); (2,"paperino")];;
val alist : (int * string) list =
  [(0, "pippo"); (1, "pluto"); (2, "paperino")]
# List.map (new_assoc alist) [1;5;3;0;4];;
- : string option list = [Some "pluto"; None; None; Some "pippo"; None]
```

Come si vede, il comportamento è diverso da quello che si ottiene con `List.assoc`:

```
# List.map (fun x -> List.assoc x alist) [1;5;3;0;4];;
Exception: Not_found.
```

Possiamo quindi definire una funzione che riporta la lista di tutti i valori associati a una data lista di chiavi, ignorando quelle che non hanno un valore:

```
(*  valore: 'a option -> 'a *)
(* valore x : se x=Some n riporta n, altrimenti errore *)
let valore = function
    Some n -> n
  | None -> failwith "valore"

(* assoc_all : ('a * 'b) list -> 'a list -> 'b list *)
(* assoc_all lista chiavi = valori associati a chiavi in lista,
                            ignorando le chiavi indefinite *)
let assoc_all lista chiavi =
  List.map valore
    (List.filter ((<>) None)
       (List.map (new_assoc lista) chiavi))
```

3.2 Alberi binari

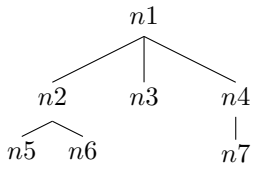
Questo paragrafo e il successivo trattano una struttura dati fondamentale, quella degli alberi, e la sua rappresentazione in OCaml.

3.2.1 Alberi: nozioni di base

Un albero è un insieme di oggetti, chiamati **nodi**, su cui è definita una relazione binaria $G(n, m)$ – che leggiamo "n è **genitore** di m" – tale che:

1. esiste un unico nodo, chiamato **radice**, che non ha genitori;
2. ogni nodo diverso dalla radice ha uno ed unico genitore;
3. per ogni nodo n diverso dalla radice esiste un **cammino** dalla radice a n (l'albero è *connesso*): esistono nodi $n_1, .., n_k$ ($k \geq 1$) tali che n_1 è la radice dell'albero, $n_k = 1$ e per ogni $i = 1, ..., k-1$ n_i è genitore di n_{i+1}.

Un albero può essere rappresentato graficamente utilizzando segmenti (*archi*) che uniscono due nodi in relazione. Un albero viene normalmente rappresentato con la radice in alto. Ad esempio:



In quest'albero, la radice è $n1$, ed è genitore di $n2$, $n3$, $n4$.

Se n è il genitore di m, allora m è un *figlio* di n. Ad esempio, nell'albero rappresentato in figura, $n5$ ed $n6$ sono figli di $n2$.

A ciascun nodo di un albero può essere associata un'*etichetta* e nodi diversi possono anche avere la stessa etichetta. Nella rappresentazione grafica di un albero etichettato, normalmente si indicano le etichette stesse e non i nodi. La loro posizione nell'albero determina il nodo a cui sono associate.

L'insieme degli alberi può essere definito per induzione, come segue:

1. L'albero costituito da un unico nodo r senza genitori è un albero, con radice r.
2. Se $t_1, t_2, ..., t_n$ sono alberi, con radici, rispettivamente, $r_1, r_2, ..., r_n$, e r è un nuovo nodo, allora la struttura che si ottiene aggiungendo il nodo r come genitore di $r_1, r_2, ..., r_n$ è un albero, la cui radice è r.
3. Nient'altro è un albero.

Introduciamo la seguente terminologia di base sugli alberi:

Cammino: sequenza di nodi $n_1, ..., n_k, n_{k+1}$ tale che, per $i = 1, ..., k$, n_i è il genitore di n_{i+1}.

Lunghezza del cammino: la lunghezza del cammino $n_1, ..., n_k, n_{k+1}$ è k, cioè uguale al numero di nodi meno 1.

Antenato e discendente: se esiste un cammino da n a m, allora n è un antenato di m e m un discendente di n.

Fratelli: nodi che hanno lo stesso genitore.

Sottoalbero: insieme costituito da un nodo n e tutti i suoi discendenti; n è la radice del sottoalbero.

Sottoalbero di un nodo n: sottoalbero la cui radice è un figlio di n.

Foglia: nodo senza figli.

Nodo interno: nodo con uno o più figli.

Altezza di un nodo: lunghezza del cammino più lungo che va dal nodo a una foglia.

Altezza dell'albero: altezza della sua radice.

Livello o profondità di un nodo: lunghezza del cammino dalla radice al nodo.

Dimensione di un albero: numero dei nodi.

3.2.2 Alberi binari

Un **albero binario** è un albero in cui ogni nodo ha al massimo due figli. Anche l"insieme degli alberi binari si può definire induttivamente, in modo indipendente dalla definizione generale di albero:

1. Un nodo n è un albero binario (una foglia), con radice n.
2. Se T_0 è un albero binario con radice n_0 e n è un nuovo nodo, allora la struttura che si ottiene aggiungendo n come genitore di n_0 è un albero binario con radice n.
3. Se T_0 e T_1 sono alberi binari con radici n_0 e n_1, rispettivamente, e n è un nuovo nodo, allora la struttura che si ottiene aggiungendo n come genitore di n_0 e di n_1 è un albero binario con radice n.
4. Nient'altro è un albero binario.

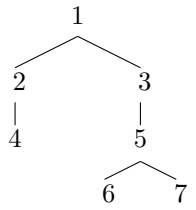
Nella rappresentazione concreta degli alberi (binari e non) si confondono a volte i nodi con le loro etichette.

non si fa generalmente distinzione tra Possiamo rappresentare gli alberi binari in OCaml mediante un tipo la cui definizione ricalca quella data sopra:

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

Un valore della forma `Leaf x` rappresenta una foglia (albero con un solo nodo), etichettata da `x`; un valore della forma `One(x,t)` rappresenta un albero la cui radice, etichettata da `x`, ha un unico figlio, che è la radice del sottoalbero `t`; mentre `Two(x,t1,t2)` rappresenta un albero la cui radice, etichettata da `x`, ha due figli.

Ad esempio, l'albero etichettato da interi:



è rappresentato dal valore di tipo `int tree`:

```
Two(1, One(2, Leaf 4),
       One(3, Two(5, Leaf 6, Leaf 7)))
```

Il tipo di un albero binario dipende dal tipo dei nodi: `Leaf, One, Two` sono costruttori polimorfi:

```
Leaf: 'a -> 'a tree
One: 'a * 'a tree -> 'a tree
Two: 'a * 'a tree * 'a tree -> 'a tree
```

Quindi `tree` è un *costruttore di tipi*

Dato che in un albero rappresentato da un valore della forma `Two(x,t1,t2)` i due sottoalberi sono ordinati, chiameremo `t1` il sottoalbero sinistro e `t2` il sottoalbero destro della radice.

La definizione induttiva di un tipo giustifica la definizione ricorsiva di funzioni su valori di quel tipo. Ad esempio, possiamo definire una funzione per calcolare la dimensione di un `'a tree`, considerando il caso foglia come caso base, e i due casi ricorsivi "albero con un unico sottoalbero della radice" e "albero con due sottoalberi della radice":

```
(* size : 'a tree -> int *)
(* size t = numero di nodi in t *)
let rec size = function
    Leaf _ -> 1
  | One(_,t) -> 1 + size t
  | Two(_,t1,t2) -> 1 + size t1 + size t2
```

(si confronti questa definizione con quella della funzione `length` per le liste, a pagina 59). L'ipotesi della ricorsione nel caso `One(_,t)` è che si sa calcolare la dimensione del sottoalbero `t`, e nel caso `Two(_,t1,t2)` si può assumere di saper calcolare la dimensione di ciascuno dei sottoalberi `t1` e `t2`.

Per evitare di dover distinguere i due casi ricorsivi nelle definizioni, gli informatici hanno inventato l'albero "vuoto" (senza nodi), in modo che ogni nodo dell'albero

abbia sempre esattamente due sottoalberi: se il nodo è una foglia, i due sottoalberi sono entrambi vuoti. Se il nodo ha un unico figlio, uno dei suoi due sottoalberi è vuoto.

Nella definizione del corrispondente tipo in OCaml, usiamo il valore `Empty` per denotare l'albero vuoto, e il costruttore funzionale `Tr`, che si applica a un nodo e due alberi:

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

Ad esempio, l'albero della figura di pagina 109 è rappresentato dal valore di tipo `int tree`:

```
  Tr(1,Tr(2,Tr(4,Empty,Empty),
           Empty),
       Tr(3,Tr(5,Tr(6,Empty,Empty),
                 Tr(7,Empty,Empty)),
           Empty))
```

E anche, ad esempio, da:

```
  Tr(1,Tr(2,Empty,
           Tr(4,Empty,Empty)),
       Tr(3,Empty,
           Tr(5,Tr(6,Empty,Empty),
                Tr(7,Empty,Empty))))
```

Con questa rappresentazione, la funzione che calcola la dimensione di un albero binario può essere definita come segue:

```
(* size : 'a tree -> int *)
(* size t = numero dei nodi di t *)
let rec size = function
    Empty -> 0
  | Tr(_,t1,t2) -> 1 + size t1 + size t2
```

3.2.3 Costruttori, selettori e predicati del tipo alberi binari

I costruttori del tipo di dati alberi binari sono dunque `Empty` e `Tr`. In corrispondenza di essi, possiamo definire il predicato `is_empty` e i tre selettori che selezionano le componenti di un albero non vuoto:

```
exception EmptyTree

(* is_empty: 'a tree -> bool *)
(* is_empty t = true se t e' vuoto, false altrimenti *)
let is_empty t =
    t = Empty
```

```
(* root: 'a tree -> 'a *)
(* root t = radice di t, se t non e' vuoto, errore altimenti *)
let root = function
    Tr(x,_,_) -> x
  | _ -> raise EmptyTree

(* left: 'a tree -> 'a tree *)
(* left t = sottoalbero sinistro di t, se t non e' vuoto, errore
            altrimenti *)
let left = function
    Tr(_,t,_) -> t
  | _ -> raise EmptyTree

(* right: 'a tree -> 'a tree *)
(* right t = sottoalbero destro di t, se t non e' vuoto, errore
            altrimenti *)
let right = function
    Tr(_,_,t) -> t
  | _ -> raise EmptyTree
```

Come al solito, i selettori possono essere utilizzati in alternativa al pattern matching. Ad esempio potremmo definire `size` come segue, trattando il tipo `'a tree` in modo astratto::

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

I due costruttori, i selettori e il predicato `is_empty` costituiscono le operazioni di base sugli alberi binari. Altre utili funzioni che possiamo considerare sono `leaf`, che costruisce un albero costituito da un solo nodo (una foglia), e il predicato `is_leaf`, che determina se un albero è una foglia oppure no.

```
(* leaf : 'a -> 'a tree *)
(* leaf x = foglia etichettata da x *)
let leaf x =
   Tr(x,Empty,Empty)

(* is_leaf : 'a tree -> bool *)
(* is_leaf t = true se t e' una foglia, false altrimenti *)
let is_leaf = function
    Tr(_,Empty,Empty) -> true
  | _ -> false
```

3.2.4 Definizioni ricorsive sugli alberi binari

Come abbiamo visto, l'insieme degli `'a tree` può essere definito induttivamente. In corrispondenza alla definizione induttiva degli alberi binari, possiamo dunque definire ricorsivamente operazioni sugli alberi binari. Nei casi più semplici, come quello che abbiamo già visto della definizione di `size`, avremo nella definizione un caso base, che definisce il risultato dell'operazione per l'albero vuoto, e un caso induttivo in cui, assumendo di saper calcolare il risultato dell'operazione per gli alberi `t1` e `t2`, si definisce il valore che si ottiene per l'albero `Tr(x,t1,t2)`.

Una struttura simile a quella di `size` ha la funzione che calcola l'altezza di un albero. L'altezza di un albero è, abbiamo detto, la massima lunghezza di un cammino che va dalla radice a una foglia. In generale, l'altezza di un albero binario non vuoto si può ottenere aggiungendo 1 al massimo delle altezze dei due sottoalberi. Perché ciò funzioni anche nel caso di alberi costituiti da un solo nodo, la cui altezza è 0, dobbiamo convenire che l'altezza di `Empty` è -1. Ricorsivamente, l'altezza di un albero binario si può definire come segue:

```
(* height : 'a tree -> int *)
let rec height = function
    Empty -> -1
  | Tr(_,t1,t2) -> 1 + max (height t1) (height t2)
```

Come ulteriore esempio, consideriamo il problema di determinare se un albero contiene un nodo la cui etichetta soddisfa una determinata proprietà `p` (un `exists` sugli alberi). Anche in questo caso, consideriamo il caso base `Empty` e il caso ricorsivo `Tr(x,t1,t2)`:

```
(* tree_exists: ('a -> bool) -> 'a tree -> bool *)
(* tree_exists p t = true sse t contiene almeno un nodo che soddisfa p *)
let rec tree_exists p = function
    Empty -> false
  | Tr(x,t1,t2) ->
      p x || tree_exists p t1 || tree_exists p t2
```

Analogamente, la funzione `raccogli` sotto definita, che riporta una lista con tutte le etichette dell'albero dato come argomento, considera gli stessi due casi:

```
(* raccogli: 'a tree -> 'a list *)
(* raccogli t = lista di tutte le etichette dei nodi di t (con ripetizioni) *)
let rec raccogli = function
    Empty -> []
  | Tr(x,t1,t2) ->
      x::((raccogli t1)@(raccogli t2))
```

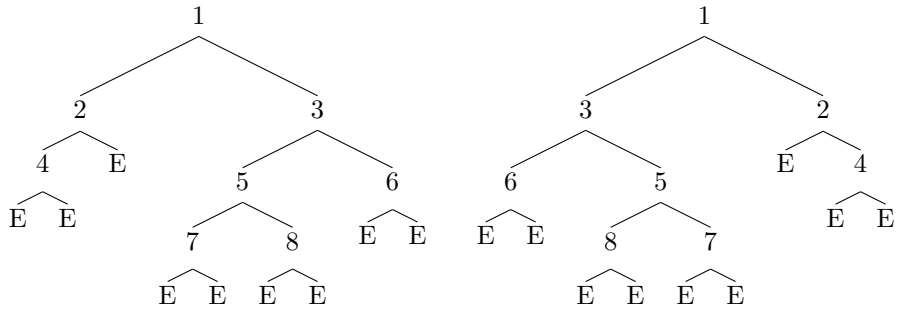
Ovviamente possono anche esserci diversi casi: se ad esempio vogliamo raccogliere soltanto le etichette delle foglie dell'albero, dovremmo considerare il caso "foglia" come caso base aggiuntivo:

```
(* foglie: 'a tree -> 'a list *)
(* foglie t = lista di tutte le etichette delle foglie di t (con ripetizioni) *)
let rec foglie = function
    Empty -> []
  | Tr(x,Empty,Empty) -> [x]
  | Tr(x,t1,t2) ->
      (foglie t1)@(foglie t2)
```

Se invece vogliamo raccogliere le etichette dei nodi che hanno esattamente un figlio, i casi da considerare saranno diversi:

```
(* nodi_con_un_figlio: 'a tree -> 'a list *)
(* nodi_con_un_figlio t =
      lista di tutte le etichette dei nodi di t che hanno esattamente un
      figlio *)
let rec nodi_con_un_figlio = function
    Empty | Tr(_,Empty,Empty) -> []
  | Tr(x,Empty,t) | Tr(x,t,Empty) ->
      x::(nodi_con_un_figlio t)
  | Tr(x,t1,t2) ->
      (nodi_con_un_figlio t1)@(nodi_con_un_figlio t2)
```

Un altro esempio di definizione ricorsiva sugli alberi binari è la funzione `reflect`: `'a tree -> 'a tree`, che calcola l'immagine riflessa (a specchio) di un albero binario. La figura qui sotto rappresenta come esempio un albero binario e la sua immagine riflessa (`E` rappresenta l'albero vuoto).




```
let rec reflect = function
    Empty -> Empty
  | Tr(x,t1,t2) -> Tr(x,reflect t2,reflect t1)
```

Stampa di un albero binario

Consideriamo qui una funzione di utilità per eseguire una stampa leggibile di un albero binario. Utilizzeremo la funzione `print_string` già introdotta a pagina 39. Le

funzioni di stampa si utilizzano normalmente all'interno di una *sequenza di comandi*, cioè un'espressione della forma:

```
E1 ; E2 ; ...  ; En
```

Il tipo e il valore di un'espressione di questa forma sono quelli di di `En`. Ma per valutare l'espressione, vengono valutate tutte le sottoespressioni, da sinistra a destra; i valori intermedi sono ignorati (e quindi interessano solo per i loro effetti collaterali), tranne quello dell'ultima espressione. Per rendere il codice più leggibile, le sequenze di comandi sono a volte incluse tra le parole chiave `begin` e `end`.

La funzione `treeprint`, il cui tipo più generale è `('a -> 'b) -> 'a tree -> unit`, viene utilizzata per stampare un albero binario di tipo `'a tree` quando il suo primo argomento è una funzione che abbia come effetto collaterale quello di stampare valori di tipo `'a`.

```
(* treeprint : ('a -> 'b) -> 'a tree -> unit *)
(* treeprint print t = stampa, con opportuna indentazione, l'albero t,
          utilizzando la funzione print per la stampa dei nodi *)
(* aux: string -> 'a tree -> unit *)
(* aux ind t stampa l'albero t, premettendo ad ogni riga un certo
           numero di spazi seguiti dalla stringa ind.
           Il numero di spazi viene incrementato per la stampa di
           ciascun sottoalbero *)
let treeprint print t =
  let rec aux ind = function
      Empty -> print_string (ind ^ "Empty")
    | Tr(x,Empty,Empty) ->
        begin
          print_string (ind^"Tr(");
          print x;
          print_string ",Empty,Empty)"
        end
    | Tr(x,t1,t2) ->
        begin
          print_string (ind^"Tr(");
          print x;
          print_string ",\n";
          aux ("   "^ind) t1;
          print_string ",\n";
          aux ("   "^ind) t2;
          print_string ")"
        end
  in  aux "" t ;
      print_string "\n"
```

Per stampare un albero etichettato da interi, la funzione viene richiamata con primo argomento `print_int`, che, applicata a un intero, lo stampa e riporta `unit` (OCaml dispone di analoghe funzioni per la stampa di caratteri e reali: `print_char` e `print_float`). Ad esempio, se abbiamo definito:

```
let albero =
    Tr(1, Tr(2, Tr(4,Empty,Empty), Empty),
  Tr(3, Tr(5, Tr(6,Empty,Empty), Tr(7,Empty,Empty)),
Empty))
```

Allora:

```
# treeprint print_int albero;;
Tr(1,
   Tr(2,
      Tr(4,Empty,Empty),
      Empty),
   Tr(3,
      Tr(5,
         Tr(6,Empty,Empty),
         Tr(7,Empty,Empty)),
      Empty))
- : unit = ()
```

Test albero bilanciato

Un albero binario si dice bilanciato se per ogni nodo n nell'albero vale la seguente proprietà: la differenza tra l'altezza del sottoalbero sinistro e quella del sottoalbero destro è al massimo 1. La funzione `balanced`: `'a tree -> bool` restituisce `true` se e solo se il valore del suo argomento è un albero bilanciato.

```
(* balanced : 'a tree -> bool *)
let rec balanced = function
    Empty -> true
  | Tr(_,t1,t2) ->
       balanced t1 && balanced t2
       && abs(height t1 - height t2) <= 1
```

L'operazione può essere resa più efficiente: non è infatti necessario che, per ciascun nodo n, i sottoalberi vengano esaminati due volte: una per controllarne il bilanciamento e una per calcolarne l'altezza. Nella versione che segue si fa uso di una funzione ausiliaria che esegue i due compiti contemporaneamente: essa riporta l'altezza del sottoalbero, se esso è bilanciato, un errore altrimenti. Nella valutazione del corpo della funzione principale viene valutata l'espressione

```
try let _ = aux t in true
with NotBalanced -> false
```

viene cioè valutata l'espressione `aux t`; se questa non solleva alcuna eccezione, il valore riportato (la dimensione dell'albero) viene di fatto ignorato e viene riportato il valore `true`; altrimenti l'eccezione `NotBalanced` viene catturata e viene riportato `false`.

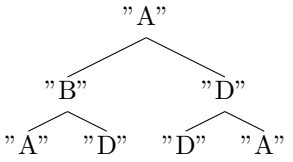
```
exception NotBalanced

(* balanced : 'a tree -> bool *)
(* balanced t = l'albero t e' bilanciato *)
(* aux: 'a tree -> int *)
(* aux t solleva NotBalanced se t non e' bilanciato, altrimenti
         riporta l'altezza di t *)
let balanced t =
  let rec aux = function
      Empty -> -1  (* l'albero vuoto e' bilanciato e ha altezza -1 *)
    | Tr(_,t1,t2) ->
        let k1 = aux t1 in
        let k2 = aux t2   (* se uno dei due sottoalberi non e'
                             bilanciato, viene sollevata l'eccezione,
                             che si propaga *)
        in if abs(k1 - k2) <= 1
           then 1 + max k1 k2
           else raise NotBalanced
  in try let _ = aux t in true
     with NotBalanced -> false
```

3.2.5 Visita di un albero binario con risultato parziale

Consideriamo il problema di "contare" quante volte occorre in un albero ciascuna etichetta. A questo scopo si vuole definre una funzione `count: 'a tree -> ('a * int) list` che, applicata a un albero t, riporti una lista di coppie contenente, per ogni etichetta x di qualche nodo dell'albero, una (unica) coppia (x, n), dove n è il numero di nodi etichettati da x.

Consideriamo come esempio l'albero rappresentato qui sotto:



Ragionando ricorsivamente nel modo standard, nel caso generale di un albero della forma `Tr(x,left,right)`, possiamo assumere di saper calcolare `count left` e `count right`. Nel caso del nostro esempio, le due chiamate ricorsive, riporteranno:

```
count left = [("B",1);("A",1);("D",1)]
count right=[("D",2);("A",1)]
```

Queste due liste vanno "fuse" addizionando gli interi associati alla stessa etichetta, per ottenere la lista `[("B",1);("A",2);("D",3)]`. A questa lista si deve poi "aggiungere" la radice, incrementando il contatore di `"A"`, per ottenere infine `[("B",1);("A",3);("D",3)]`.

Quindi le operazioni da fare ad ogni chiamata ricorsiva sono due: effettuare la somma dei contatori di due liste di coppie e incrementare il contatore associato a un'etichetta.

Ma ciò che si farebbe in un linguaggio imperativo è diverso: si utilizzerebbe un "risultato parziale" (inizializzato alla lista vuota) e si visiterebbe l'albero, modificando il risultato parziale per ogni nodo visitato. La modifica da effettuare sull'accumulatore è la seconda delle due operazioni: incrementare il contatore associato a un'etichetta.

Possiamo implementare lo stesso algoritmo in OCaml: visitiamo l'albero conservando un risultato parziale, cioè una lista di coppie contenente le informazioni relative ai nodi già visitati. Visitare un nodo a significa scandire il risultato parziale, aggiungendo 1 al secondo elemento della coppia (a, n), se una tale coppia esiste, e aggiungendo la coppia $(a, 1)$ se invece non esiste.

Questo esempio illustra un metodo generale: non essendo gli alberi una struttura lineare, come le liste, la definizione di una funzione ricorsiva con un risultato parziale che viene "accumulato" durante la visita è un pochino più complessa (e in ogni caso non si produce un processo iterativo!).

- nel caso base (albero vuoto), la visita è completa e viene riportato il risultato parziale;
- nel caso ricorsivo, assumiamo che l'albero sia `Tr(x,t1,t2)` e che `result` sia il risultato parziale accumulato fino a quel momento. Allora visitiamo la radice modificando di conseguenza `result` (nel nostro caso incrementiamo il contatore della radice), ottenendo un nuovo risultato parziale `result'`; questo viene utilizzato come "accumulatore" per la visita di uno dei due sottoalberi, ad esempio `t1`. La visita di `t1` con risultato parziale `result'` produrrà un secondo risultato parziale, `result''`, che verrà utilizzato per la visita del sottoalbero `t2`. Il risultato riportato da questa seconda visita è il risultato voluto.

Per implementare questo algoritmo di visita, definiamo innanzitutto la funzione che rappresenta la visita di un nodo:

```
(* add : 'a -> ('a * int) list -> ('a * int) list *)
(* add y lst = lista che si ottiene da lst sostituendo la
               coppia (y,n) con (y,n+1), se una tale coppia esiste,
               altrimenti aggiungendo la coppia (y,1) *)
let rec add y = function
    [] -> [(y,1)]
  | (x,n)::rest ->
      if x=y then (x,n+1)::rest
      else (x,n)::add y rest
```

Questa funzione viene utilizzata per modificare l'accumulatore nella funzione principale:

```
(* count : 'a tree -> ('a * int) list *)
(* aux:  ('a * int) list -> 'a tree ->  ('a * int) list *)
(* aux result t = lista che si ottiene da result "aggiungendo"
                          (con add) i nodi di t *)
 let count t =
  let rec aux result = function
      Empty -> result
    | Tr(a,t1,t2) ->
        aux (aux (add a result) t1) t2
  in aux [] t
```

Come ulteriore esempio, consideriamo una versione più efficiente di `raccogli`: la definizione di pagina 112 esegue un "append" ad ogni chiamata ricorsiva, ma questa è un'operazione costosa. La versione che proponiamo ora non esegue alcun append: essa utilizza una funzione ausiliaria che "accumula" in una lista i nodi visitati.

```
(* raccogli: 'a tree -> 'a list *)
(* raccogli t = lista con le etichette dei nodi di t *)
(* aux: 'a list -> 'a tree -> 'a list *)
(* aux result t = (lista con le etichette dei nodi di t) @ result *)
let raccogli t =
  let rec aux result = function
    Empty -> result
  | Tr(x,t1,t2) ->
      x:: aux (aux result t1) t2
  in aux [] t
```

3.2.6 Ricerca di un ramo in un albero

Fin qui abbiamo implementato operazioni che sono di fatto istanze di algoritmi generali di visita. Consideriamo invece ora il problema della ricerca di un ramo, ad esempio un ramo dell'albero dalla sua radice fino a una foglia etichettata da un valore dato. Rappresentando i rami come liste di nodi, il problema consiste nel definire una funzione `path_to: 'a -> 'a tree -> 'a list` tale che `path_to x t` riporti una lista rappresentante un ramo dalla radice di `t` fino a una foglia qualsiasi etichettata da `x`, se una tale foglia esiste, altrimenti sollevi un'eccezione.

Si tratta di un problema molto simile a quello dell'attraversamento della palude (paragrafo 2.4.4), che possiamo risolvere mediante la tecnica del *backtracking*: nel caso generale di un albero (che non sia una foglia) della forma `Tr(y,t1,t2)`, abbiamo due possibilità alternative: cercare il cammino in `t1` o cercarlo in `t2`. Se una delle due richerche ha successo, riporterà un ramo `path` fino a una foglia etichettata da `x` in uno dei due sottoalberi. Alla lista `path` dovremo quindi aggiungere `x` in testa. I casi base da considerare a parte sono i casi albero vuoto e foglia.

```
(* path_to : 'a -> 'a tree -> 'a list *)
(* path_to x t = ramo in t dalla radice a una foglia
                 etichettata da x *)
let rec path_to x = function
    Empty -> raise NotFound
  | Tr(y,Empty,Empty) ->
      if y=x then [x]
      else raise NotFound
  | Tr(y,t1,t2) ->     (* backtracking *)
      y::(try path_to x t1
          with NotFound -> path_to x t2)
```

3.3 Alberi n-ari

3.3.1 Rappresentazione di alberi n-ari

In un albero n-ario, a differenza che in un albero binario, un nodo può avere un numero qualsiasi di figli (purché finito). Per rappresentare i figli di un nodo, le liste sono dunque le strutture più adatte. Quindi un albero n-ario può essere rappresentato da una coppia, costituita dall'etichetta della radice e una lista di alberi n-ari.

In OCaml, possiamo utilizzare la struttura dati così definita:

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

La forma generale di un valore di tipo `'a ntree` è dunque sempre `Tr(x,tlist)`, dove `tlist` è una lista di `'a ntree`.

Si riguardi la definizione induttiva degli alberi data a pagina 107: la definizione del tipo `'a ntree` ricalca il caso generale (caso 2 della definizione). Le foglie saranno rappresentate da valori della forma `Tr(x,[])`. Possiamo definire quindi una funzione di utilità per costruire alberi costituiti da un unico nodo:

```
(* leaf: 'a -> 'a ntree *)
(* leaf x = foglia etichettata da x *)
let leaf x = Tr(x,[])
```

Ad esempio, l'albero di interi `t` rappresentato nella figura 3.1 è rappresentato dal valore della variabile `t` così definita (utilizzeremo a volte questo albero negli esempi):

```
let t = Tr(1,[Tr(2,[Tr(3,[leaf 4;
                          leaf 5]);
                    Tr(6,[leaf 7]);
                    leaf 8]);
              leaf 9;
              Tr(10,[Tr(11,[leaf 12;
                            leaf 13;
                            leaf 14]);
                     leaf 15;
                     Tr(16,[leaf 17;
                            Tr(18,[leaf 19;
                                   leaf 20])])])])
```

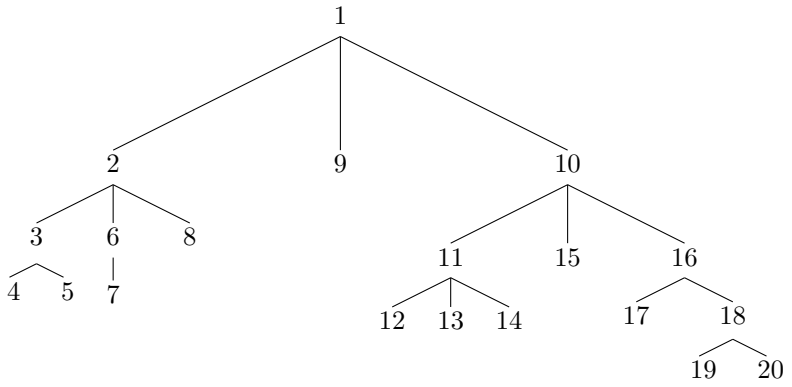


Figura 3.1: Un albero di esempio

3.3.2 La mutua ricorsione

Riguardando la definizione del tipo `'a ntree` possiamo osservare che essa presenta un'apparente circolarità: un `'a ntree` è definito in termini di `'a ntree list`, ma una lista di alberi presuppone di aver definito cos'è un `'a ntree`. Questa osservazione suggerisce un modo di lavorare sugli alberi n-ari: si definisce una funzione che si applica ad un albero in termini di un'altra funzione che si applica a liste di alberi. La seconda, a sua volta, richiamerà la prima.

Normalmente, OCaml non consente di utilizzare funzioni che non siano già state definite, ma questa regola può essere violata da funzioni definite in *mutua ricorsione*. Le definizioni di funzioni mutuamente ricorsive utilizza costrutti della forma:

```
let rec f1 ... = ...<qui si puo' richiamare f2>
    and f2 ... = ...<qui si puo' richiamare f1>
```

(dove `and` è una parola chiave).

Un esempio banale di definizione mutuamente ricorsiva (assolutamente inefficiente) è quella dei numeri naturali pari e dispari: un naturale maggiore di 1 è pari se il suo predecessore è dispari, e un naturale maggiore di 0 è dispari se il suo predecessore è pari:

```
(* pari/dispari : int -> bool
      definite solo sui naturali *)
let rec pari n =
  n<>1 && (n=0 || dispari (n-1))
and dispari n =
  n<>0 && (n=1 || pari (n-1))
```

Un esempio meno banale è dato dalle sequenze maschio-femmina di Hofstadter,[1] così definite:

$$\begin{array}{rll} & F(0) = 1 & M(0) = 0 \\ \text{per } n > 0\text{:} & F(n) = n - M(F(n-1)) & M(n) = n - F(M(n-1)) \end{array}$$

I primi elmenti di queste sequenze sono,rispettivamente:

$$\begin{array}{ll} F: & 1, 1, 2, 2, 3, 3, 4, 5, 5, 6, 6, 7, 8, 8, 9, 9, 10, 11, 11, 12, 13, \ldots \\ M: & 0, 0, 1, 2, 2, 3, 4, 4, 5, 6, 6, 7, 7, 8, 9, 9, 10, 11, 11, 12, 12, \ldots \end{array}$$

Le due funzioni possono essere definite in OCaml come segue:

```
let rec f = function
    0 -> 1
  | n -> n - m(f(n-1))
and m = function
    0 -> 0
  | n -> n-f(m(n-1))
```

Come si vedrà in seguito, per definire funzioni sugli alberi n-ari, si potrà in alcuni casi ricorrere a funzioni di ordine superiore sulle liste. Ma, quando non è possibile (o non è consigliabile), le funzioni possono spesso essere definite in mutua ricorsione con funzioni corrispondenti su liste di alberi.

3.3.3 Alcune semplici funzioni su alberi n-ari

Come primo esempio, consideriamo la definizione di una funzione che, applicata a un albero t, ne riporti il numero dei nodi. Dato che per calcolare il numero dei nodi di un albero `Tr(x,[`t_1`;...;`t_n`]` è necessario calcolare la dimensione di ciascuno dei sottoalberi t_1`;...;`t_n, possiamo usare a questo scopo la funzione `List.map`:

[1] Douglas Hofstadter, Gödel, Escher, Bach: an Eternal Golden Braid, Penguin Books (1980). Traduzione italiana: Gödel, Escher, Bach: un'Eterna Ghirlanda Brillante, Adelphi (1990).

```
(* size : 'a ntree -> int *)
(* size t = numero di nodi di t *)
let rec size (Tr(_,tlist)) =
  1 + sumof (List.map size tlist)
```

In alternativa all'uso della funzione di ordine superiore `List.map`, si può utilizzare la mutua ricorsione: la funzione `size`, che si applica ad un singolo albero, si definisce in termini della funzione `sumofsizes`, che si applica a liste di alberi ed è tale che `sumofisizes [t1;...;tn] = size t1 + ...+ size tn`:

```
(* size : 'a ntree -> int
   size t = numero di nodi di t
   sumofsizes : 'a ntree list -> int
   sumofsizes tlist = somma del numero dei nodi degli alberi
                      in tlist *)
let rec size (Tr(_,tlist)) =
  1 + sumofsizes tlist
and sumofsizes = function
    [] -> 0
  | t::rest -> (size t) + sumofsizes rest
```

Gli algoritmi di visita di un albero n-ario sono generalizzazioni dei corrispondenti algoritmi sugli alberi binari:

visita in preordine: viene visitata la radice, poi i sottoalberi;

visita in postordine: vengono visitati i sottoalberi, poi la radice;

visita simmetrica: viene visitato il primo sottoalbero (se esiste), poi la radice, poi gli altri sottoalberi.

La definizione di `size` implementa un'algoritmo di visita in postordine: prima vengono visitati i sottoalberi (per calcolarne la dimensione) e poi la radice ("visitare" un nodo significa aggiungere 1 al risultato).

Diamo qui come esempio la definizione della funzione `preorder`, che costruisce la lista dei nodi di un albero n-ario, secondo l'ordine in cui sarebbero visitati mediante una visita in preordine.

```
(* preorder : 'a ntree -> 'a list
   preorder t = lista dei nodi di t nell'ordine in cui sarebbero
   visitati secondo la visita in preordine *)
(* preorder_tlist : 'a ntree list -> 'a list
   preorder_tlist [t1;...;tn] = (preorder t1) @ ... @ (preorder tn)
*)
let rec preorder (Tr(x,tlist)) =
    x::preorder_tlist tlist
and preorder_tlist = function
    [] -> []
  | t::ts -> preorder t @ preorder_tlist ts
```

Alternativamente, possiamo utilizzare `List.map` per costruire la lista di liste `[preorder` t_1`;...;preorder` t_n`]` e poi "appiattire" tale lista:

```
let rec preorder (Tr(x,tlist)) =
    x::List.flatten (List.map preorder tlist)
```

Avremo dunque, per l'albero della figura 3.1:

```
# preorder t;;
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19; 20]
```

In generale, quando il calcolo del valore di una funzione su un albero n-ario richiede di eseguire una stessa operazione su *tutti* i sottoalberi, si può utilizzare la funzione `List.map`. Come ulteriore esempio definiamo la funzione `height`, che calcola l'altezza di un albero n-ario, utilizzando la funzione `maxl`, che riporta il massimo elemento di una lista:

```
(* maxl: 'a list -> 'a
   maxl lst = elemento massimo di lst, fallisce per la lista vuota *)
let rec maxl = function
    [x] -> x
  | x::rest -> max x (maxl rest)
  | _ -> failwith "maxl"

(* height : 'a ntree -> int
   height t = altezza di t *)
let rec height (Tr(x,tlist)) = match tlist with
  [] -> 0
| _ ->  1 + maxl (List.map height tlist)
```

In questo caso, dato che `maxl` fallisce quando è applicata alla lista vuota, nella funzione `height` è necessario distinguere il caso foglia dal caso generale.

La stessa funzione si può definire utilizzando la mutua ricorsione come segue:

```
(* height : 'a ntree -> int
   height t = altezza di t
   maxheight: 'a ntree list -> int
   maxheight tlist = se tlist <>[] allora il massimo tra le altezze
                     degli alberi in tlist,  altrimenti errore *)
let rec height (Tr(x,tlist)) = match tlist with
  [] -> 0
| _ -> 1+ maxheight tlist
and maxheight = function
    [] -> failwith "maxheight"
  | [t] -> height t
  | t::rest -> max (height t) (maxheight rest)
```

Come ultimo esempio di uso della funzione `List.map` per lavorare su alberi n-ari, definiamo una funzione che, applicata a un albero, riporta il suo *fattore di ramificazione*, cioè il massimo numero di sottoalberi di un nodo in `t`:

```
(* branching_factor :  'a ntree -> int *)
(* branching_factor t = fattore di ramificazione di t *)
let rec branching_factor (Tr(x,tlist)) =
  match tlist with
    [] -> 0
  | _ -> max (List.length tlist)
             (maxl (List.map branching_factor tlist))
```

Consideriamo ora il problema di determinare se un albero contiene un nodo etichettato da un valore dato. In questo caso, naturalmente, si può (ed è opportuno) interrompere la ricerca non appena esso viene trovato. Utilizzando la mutua ricorsione, definiamo:

```
(* occurs_in : 'a ntree -> 'a -> bool
   occurs_in t x = true se x occorre in t
   occurs_in_tlist : 'a ntree list -> 'a -> bool
   occurs_in_tlist tlist x = true se x occorre in almeno uno degli
                             alberi in tlist *)
let rec occurs_in (Tr(x,tlist)) y =
    x=y || occurs_in_tlist tlist y
and occurs_in_tlist tlist y =
  match tlist with
    [] -> false
  | t::ts -> occurs_in t y || occurs_in_tlist ts y
```

In questo caso, possiamo, in alternativa, utilizzare `List.exists`:

```
let rec occurs_in (Tr(x,tlist)) y =
    x=y || List.exists (fun t -> occurs_in t y) tlist
```

3.3.4 Ricerca di un ramo

L'uso della mutua ricorsione è necessario ogniqualvolta si possa (e si voglia) evitare di visitare *tutti* i sottoalberi di un albero, perché la visita di alcuni di essi può già essere sufficiente per determinare la soluzione, e non si hanno a disposizione funzioni di ordine superiore sulle liste che possano essere d'aiuto.

Presentiamo qui la soluzione dello stesso problema della sezione 3.2.6, per il caso degli alberi n-ari. Anche in questo caso utilizziamo la tecnica del *backtracking*: dal nodo di partenza si cerca la soluzione lungo il primo ramo che parte da esso; se viene trovata, essa è la soluzione cercata, altrimenti "si torna indietro" e si prova un cammino alternativo, e così via fino a che non si trova una soluzione o si esauriscono tutte le possibilità. La stessa strategia viene adottata, ricorsivamente, per ciascun nodo di ciascun cammino tentato.

La funzione principale `path`, applicata a un albero n-ario t e a un elemento y, riporta un ramo in t dalla radice a una foglia etichettata da y, se esiste, un errore altrimenti. Essa è definita in mutua ricorsione con la funzione `path_tlist` che, applicata a una lista di alberi $[t_1, ..., t_n]$ e a un elemento y, riporta un cammino in uno degli alberi $t_1, ..., t_n$ dalla rispettiva radice fino a una foglia etichettata da y, un errore se y non occorre in alcuna foglia di alcun t_i.

```
exception NotFound

(* path : 'a ntree -> 'a -> 'a list
   path t y = cammino fino a una foglia etichettata da y in t
   path_tlist : 'a ntree list -> 'a -> 'a list
   path_tlist tlist y = cammino fino a una foglia etichettata da y in
                        uno degli alberi in tlist *)
let rec path (Tr(x,tlist)) y =
  match tlist with
    [] -> if x=y then [x] else raise NotFound
  | _ -> x::path_tlist tlist y
and path_tlist tlist y = match tlist with
  [] -> raise NotFound
| t::ts -> try path t y
           with NotFound -> path_tlist ts y
```

È possibile dare un'implementazione alternativa della ricerca di un cammino in un albero n-ario, senza utilizzare la mutua ricorsione. In questo caso si utilizza una funzione ausiliaria `aux`, che si applica a liste di alberi, tale che `aux` $[t_1;...;t_n]$ riporta un cammino con la proprietà voluta in t_1, se esiste, altrimenti in t_2, altrimenti ... altrimenti in t_n, altrimenti riporta un errore. Ad ogni chiamata di aux, la lista di alberi corrisponde a un insieme di sottoalberi di uno stesso nodo. La funzione `aux` è richiamata inizialmente con argomento `[t]`.

```
(* path: 'a ntree -> 'a -> 'a list *)
(* aux: 'a ntree list -> 'a list
   aux tlist = cammino desiderato in uno degli alberi in tlist *)
let path t y =
  let rec aux = function
      [] -> raise NotFound
    | Tr(x,[])::rest ->
        if x=y then [x]
        else aux rest
    | Tr(x,tlist)::rest ->
          try x::aux tlist
          with NotFound -> aux rest
  in aux [t]
```

3.3.5 Cancellazione di un nodo in un albero n-ario

La funzione `delete: 'a ntree -> 'a -> 'a ntree`, applicata a un albero t e un elemento y, diverso dalla radice di t, riporta l'albero che si ottiene da t cancellando uno ed un unico nodo diverso dalla radice etichettato da y e aggiungendo i sottoalberi di y ai sottoalberi del genitore di y. Se y non occorre in t oppure se vi occorre solo nella radice, riporta un errore. La funzione è definita in mutua ricorsione con `delete_tlist: 'a -> 'a ntree list -> 'a ntree list` tale che `delete_tlist` `y` `[`$\mathtt{t}_1$`,...,`$\mathtt{t}_n$`]` che riporta, quando ha successo, i nuovi sottoalberi: se la radice di uno di essi, diciamo t_i, è uguale all'elemento y da cancellare, riporta la lista che si ottiene sostituendo t_i con i suoi sottoalberi in $[t_1, ..., t_n]$. Altrimenti cancella y dal primo sottoalbero t_i in cui occorre, ottenendo t'_i, e riporta la lista $[t_1, ..., t'_i, ..., t_n]$.

```
exception NotFound
(* delete : 'a ntree -> 'a -> 'a ntree
   delete_tlist : 'a -> 'a ntree list -> 'a ntree list *)
let rec delete (Tr(x,tlist)) y =
    Tr(x,delete_tlist y tlist)
and delete_tlist y = function
    [] -> raise NotFound
  | (Tr(x,ts))::tlist ->
      if x=y then ts@tlist
      else
        try (delete (Tr(x,ts)) y)::tlist
        with NotFound -> (Tr(x,ts))::(delete_tlist y tlist)
```

3.4 Grafi

3.4.1 Nozioni preliminari

Un grafo orientato è una coppia (V, A) dove V è un insieme (l'insieme dei *nodi* o *vertici* del grafo) e A è una relazione binaria su V (l'insieme degli *archi* del grafo). Un grafo può essere rappresentato graficamente mediante un insieme di punti (i nodi) collegati da frecce (gli archi): quando esiste una freccia dal nodo x al nodo y significa che la coppia (x, y) è un arco del grafo (cioè che x è in relazione con y).

Ad esempio, la figura 3.2 rappresenta il grafo (V, A) con $V = \{a, b, c, d\}$ e $A = \{(a, b), (a, c), (c, a), (d, d), (b, d)\}$.

Quando la relazione A è simmetrica, cioè quando per ogni coppia X, Y di nodi del grafo si ha:

$$(X, Y) \in A \Rightarrow (Y, X) \in A,$$

allora il grafo si dice *non orientato*. Nella rappresentazione di un grafo non orientato, gli archi sono rappresentati da linee anziché frecce.

Se $G = (V, A)$ è un grafo orientato, allora:

(successori) Se $x \in V$, l'insieme dei successori di x è: $\{y \in V : (x, y) \in A\}$;

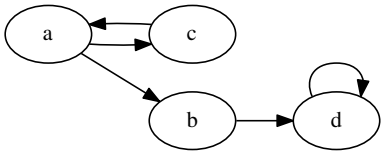


Figura 3.2: Un grafo orientato.

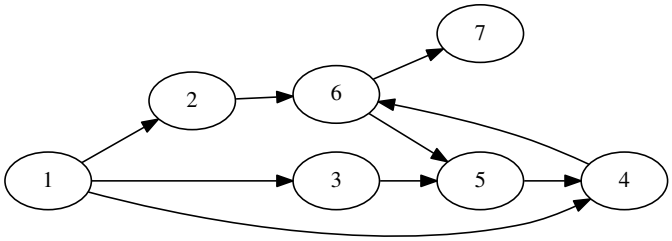


Figura 3.3: V = {1,2,3,4,5,6,7}, A = {(1,2),(1,3),(1,4),(2,6),(3,5),(4,6),(6,5),(6,7),(5,4)}

(predecessori) Se $x \in V$, l'insieme dei predecessori di x è: $\{y \in V : (y, x) \in A\}$;

(sorgente) Un nodo è detto sorgente se non ha predecessori;

(pozzo) Un nodo è detto pozzo se non ha successori.

Se G è un grafo non orientato, anziché parlare di "successori" e "predecessori" di un nodo, si parla normalmente dei suoi "vicini".

Inoltre:

Definizione 3.4.1 (CAMMINO) *Dato un grafo $G = (V, A)$, e dati $X, Y \in V$, un cammino da X a Y è una sequenza di nodi $(a_0 = X, a_1, \ldots, a_n = Y)$ tale che*

$$\textit{per ogni } i = 0, \ldots, n-1, \ (a_i, a_{i+1}) \in A$$

Un nodo Y si dice accessibile *dal nodo X se esiste un cammino da X a Y.*

Ad esempio, nel grafo rappresentato nella figura 3.3, possiamo individuare i seguenti cammini dal nodo 1 al nodo 6: (1, 2, 6), (1, 4, 6) e (1, 3, 5, 4, 6), ed anche (1, 4, 6, 5, 4, 6), ecc.

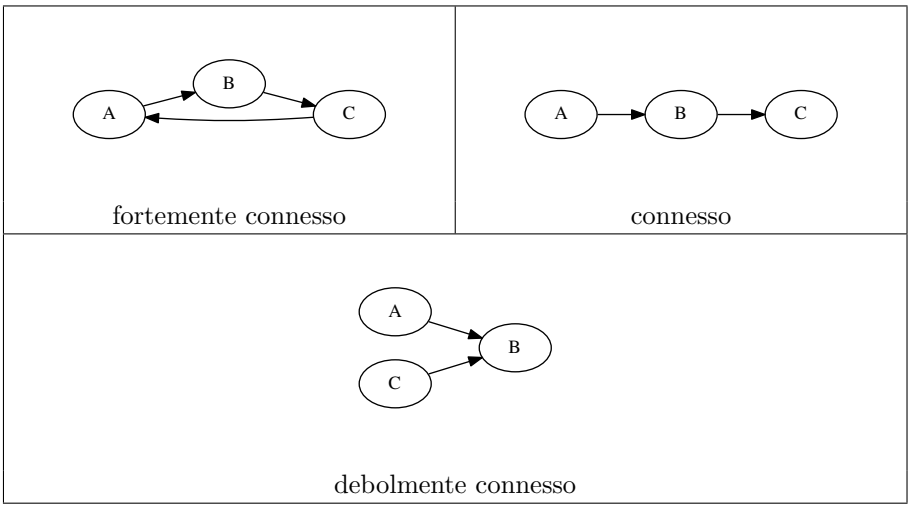


Figura 3.4: Diversi tipi di connessione.

Definizione 3.4.2 (Semicammino) *Dato un grafo $G = (V, A)$, e dati $X, Y \in V$, un semicammino da X a Y è una sequenza di nodi $(a_0 = X, a_2, \ldots, a_n = Y)$ tale che*

$$\text{per ogni } i = 0, \ldots, n-1,\ (a_i, a_{i+1}) \in A \text{ oppure } (a_{i+1}, a_i) \in A$$

Ad esempio, nel grafo della figura 3.3 un semicammino da 1 a 6 è (1, 3, 5, 6).

Un grafo non costituisce necessariamente una struttura connessa come quelli delle figure 3.2 e 3.3. Ad esempio, se consideriamo la struttura costituita da tutti i nodi e tutti gli archi dei due grafi di tali figure, questo è ancora un grafo, sebbene *non connesso*. È possibile identificare diverse varianti della nozione di connessione per un grafo, come definite qui di seguito ed illustrate nella figura 3.4.

Definizione 3.4.3 *Un grafo $G = (V, A)$ è*

fortemente connesso *se per ogni $X, Y \in V$ esiste un cammino da X a Y;*

connesso *se per ogni $X, Y \in V$ esiste un cammino da X a Y oppure esiste un cammino da Y a X;*

debolmente connesso *se per ogni $X, Y \in V$ esiste un semicammino da X a Y.*

Definizione 3.4.4 (Ciclo in un grafo orientato) *Un ciclo su X in un grafo orientato è un cammino da X a X di lunghezza ≥ 1 (cioè costituito da almeno un arco).*

Nel caso di un grafo non orientato, per evitare di considerare cicli tutti i cammini (X, Y, X), la definizione di ciclo è diversa.

Definizione 3.4.5 (Ciclo in un grafo non orientato) *Un ciclo in un grafo orientato è un cammino su X di lunghezza ≥ 3 (con almeno due archi) da X a X.*

3.4.2 Rappresentazione dei grafi mediante liste di archi

In un grafo, la relazione A è un insieme di coppie di nodi, cioè un arco è una coppia di nodi. Possiamo quindi rappresentare un grafo esplicitamente mediante una coppia: (*insieme di nodi*, *insieme di archi*). Se rappresentiamo gli insiemi mediante liste, un modo di rappresentare un grafo può essere mediante una struttura (*lista di nodi*, *lista di archi*).

Per molte applicazioni, tuttavia, non è necessario avere esplicitamente rappresentato l'insieme dei nodi; ad esempio, quando si assume che ogni nodo del grafo occorra (come primo o secondo elemento) almeno in un arco. In questo caso infatti l'insieme dei nodi si può calcolare sulla base dell'insieme degli archi. Per tutte queste applicazioni possiamo rappresentare un grafo direttamente mediante una lista di coppie di nodi.

Il tipo grafo è un tipo polimorfo: dipende dal tipo dei nodi. Possiamo quindi definire

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

Ad esempio, la seguente dichiarazione di valore lega la variabile `grafo1` alla rappresentazione della struttura illustrata nella figura 3.3.

```
let grafo1 = [(1,2);(1,3);(1,4);(2,6);
              (3,5);(4,6);(6,5);(6,7);(5,4)];;
```

La funzione definita in seguito riporta, a partire dalla rappresentazione di un grafo come lista di archi, l'insieme dei nodi non isolati del grafo stesso.

```
(* setadd: 'a -> 'a list -> 'a list *)
(* setadd x lst = aggiunge x a lst, se non c'e' gia' *)
let setadd x set =
  if List.mem x set then set else x::set

(* nodes: 'a graph -> 'a list *)
(* nodes g = lista dei nodi del grafo g *)
let rec nodes = function
    [] -> []
  | (x,y)::rest ->
      setadd x (setadd y (nodes rest))
```

I successori e i vicini di un nodo

L'operazione fondamentale sui grafi, necessaria per molte applicazioni, è quella di determinare tutti i successori di un nodo dato. Data la rappresentazione di un grafo *orientato* mediante lista di archi, per determinare i successori del nodo N si raccolgono i secondi elementi delle coppie che hanno N come primo elemento. La funzione seguente svolge tale compito.

```
(* successori : 'a -> 'a graph -> 'a list  *)
(* successori x g = lista dei successori di x in g (orientato) *)
let rec successori nodo = function
      [] -> []
    | (x,y)::rest ->
        if x = nodo then y::successori nodo rest
        else successori nodo rest
```

Oppure, utilizzando le funzioni di ordine superiore sulle liste, si può filtrare la lista degli archi rispetto alla proprietà di avere il primo elemento uguale al nodo dato, e poi prendendo tutti i secondi elementi della lista di coppie ottenuta:

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

Un grafo non orientato è ugualmente rappresentabile mediante il tipo `'a graph`. Si conviene normalmente che, se due nodi X e Y sono collegati da un arco, la lista degli archi che rappresenta il grafo conterrà soltanto una delle coppie (X, Y) o (Y, X). La funzione che determina i vicini di un nodo per grafi non orientati deve tener conto di questa convenzione.

```
(* vicini: 'a -> 'a graph -> 'a list  *)
(* vicini x g = lista dei vicini di x in g (non orientato) *)
let rec vicini nodo = function
    [] -> []
  | (x,y)::rest ->
      if x = nodo then y::vicini nodo rest
      else if y = nodo then x::vicini nodo rest
      else vicini nodo rest
```

3.4.3 Visita di un grafo

Come per gli alberi, anche per i grafi è necessario definire algoritmi di visita, che consentano di analizzare tutti i nodi, in un dato ordine. Naturalmente, un albero è una struttura con un "punto d'ingresso" fissato (la radice), mentre in un grafo la visita può iniziare da un nodo qualsiasi. Il nodo di partenza sarà dunque un parametro dell'algoritmo di visita. Inoltre, a differenza degli alberi, durante la visita di un grafo è possibile tornare, seguendo gli archi, ad un nodo già visitato (se esistono cicli nel grafo). Per evitare che in tal caso l'algoritmo di visita entri in un ciclo infinito, è necessario controllare di volta in volta che il nodo considerato non sia stato già analizzato.

Due sono i più comuni metodi per visitare un grafo:

Visita in profondità (o *depth first search*): se il nodo di partenza *start* non è stato già visitato, si analizza *start* e, per ogni successore x di *start*, si visita, con lo stesso metodo, il grafo a partire da x, ricordando che *start* è già stato visitato. È un algoritmo analogo alla visita in preordine di un albero.

Visita in ampiezza (o *breadth first search*): se il nodo di partenza *start* non è già stato visitato allora si analizza *start*, si visitano tutti i suoi successori (ricordando che *start* è già stato considerato), poi tutti i successori dei successori di *start*, e così via.

L'idea fondamentale per l'implementazione degli algoritmi di visita è quella di mantenere una struttura, che chiameremo *Pending*, contenente i nodi *in attesa di essere visitati*. Inizialmente tale struttura contiene solo il nodo di ingresso. Ad ogni passo, si estrae un nodo dal *Pending*, si visita, e si inseriscono i suoi successori (o vicini) nella struttura stessa. La differenza tra i due algoritmi di visita consiste nel modo in cui viene gestita la struttura *Pending*:

- In una visita in profondità, *Pending* è gestita come una *pila*: gli elementi vengono inseriti in testa (come in una lista) ed estratti sempre dalla testa della lista. La disciplina di inserimento ed estrazione degli elementi segue cioè il principio: l'ultimo ad entrare è il primo ad uscire (*LIFO: last in, first out*).

- In una visita in ampiezza invece, *Pending* viene gestito come una *coda*: gli elementi vengono inseriti in coda ed estratti dalla testa. La disciplina di inserimento ed estrazione degli elementi segue cioè il principio: il primo ad entrare è il primo ad uscire (*FIFO: first in, first out*).

Per esemplificare gli algoritmi di visita sopra descritti, vediamo come visitare un grafo a partire da un nodo X, allo scopo di riportare la lista dei nodi visitati, cioè dei nodi raggiungibili da X: si vuole ottenere una lista che rappresenti l'insieme $\{Y :$ esiste un cammino da X a $Y\}$.

Poiché in generale un grafo può contenere cicli, occorre tener traccia dei nodi già visitati, ed evitare di visitarli più volte, altrimenti è possibile che la visita non termini. Per memorizzare i nodi già visitati (che non si possono "marcare" come quando si utilizzano strutture imperative per rappresentare i grafi), si utilizzerà una lista.

Visita in Profondità

La funzione seguente visita un grafo G in profondità (*depth first*), a partire da un nodo N, e riporta la lista dei nodi visitati, in ordine inverso rispetto a quello in cui sono stati visitati. Essa dunque riporta una lista dei nodi accessibili dal nodo N (si veda la Definizione 3.4.1).

I parametri della funzione principale sono il grafo e il nodo di partenza. Ma durante la visita si avranno invece due strutture su cui operare: la lista contenente i nodi in attesa di essere visitati e quella con i nodi già visitati. Utilizzeremo quindi una funzione ausiliaria, `search`, che ha come parametri queste due liste (inutile – e pesante dal punto di vista computazionale – passarle anche il grafo, che non viene mai modificato). Come si è detto, la lista dei nodi in attesa di essere visitati è gestita come una pila.

```
(* depth_first_collect : 'a graph -> 'a -> 'a list *)
(* depth_first_collect g x = lista dei nodi raggiungibili da x in g *)
(* search: 'a list -> 'a list -> 'a list *)
(* search visited pending =
         (nodi raggiungibili da qualche nodo in pending mediante
          un cammino che non passa per visited) @ visited *)
let depth_first_collect graph start =
    let rec search visited = function
        [] -> visited
      | n::rest ->
            if List.mem n visited
            then search visited rest
            else search (n::visited)
                        ((successori n graph) @ rest)
                          (* oppure: (vicini n graph) @ rest
                             se il grafo non e' orientato *)
                  (* i nuovi nodi sono inseriti in testa *)
    in search [] [start]
```

In questo caso, il parametro *visited* della funzione ausiliaria svolge il doppio ruolo di struttura per memorizzare i nodi già visitati e di "accumulatore": alla fine della visita la lista dei nodi visitati sarà proprio la lista dei nodi accessibili dal nodo di ingresso.

La lista dei nodi *pending* viene inizializzata con l'unico nodo di ingresso e *visited* con la lista vuota (inizialmente non è stato visitato alcun nodo). In generale, ad ogni stadio della visita di un grafo, se non si termina – quando *pending* è vuota, cioè non ci sono nodi in attesa di essere visitati (e, in questo caso il risultato è *visited*) –, si visita il primo nodo di *pending* (visitarlo in questo caso significa semplicemente inserirlo in *visited*, che, come si è detto svolge anche il ruolo di "accumulatore"), si aggiunge alla lista dei nodi già visitati, e si aggiungono a *pending* i suoi successori (o vicini). Nella visita in profondità questi sono aggiunti in testa a *pending*.

Visita in Ampiezza

L'algoritmo di visita in ampiezza differisce da quello della visita in profondità semplicemente per il fatto che i successori del nodo visitato vengono inseriti in coda alla lista contenente i nodi in attesa di essere visitati:

.

```
(* breadth_first_collect : 'a graph -> 'a -> 'a list *)
(* breadth_first_collect g x = lista dei nodi raggiungibili da x in g *)
(* search: 'a list -> 'a list -> 'a list *)
(* search visited pending =
         (nodi raggiungibili da qualche nodo in pending mediante
          un cammino che non passa per visited) @ visited *)
let breadth_first_collect graph start =
    let rec search visited = function
        [] -> visited
      | n::rest ->
            if List.mem n visited
            then search visited rest
            else search (n::visited)
                        (rest @ (successori n graph))
                             (* oppure: (vicini n graph) @ rest
                             se il grafo non e' orientato *)
                (* i nuovi nodi sono inseriti in coda *)
    in search [] [start]
```

Consideriamo come esempio il grafo della figura 3.3. Ovviamente, fissato un algoritmo di visita e un nodo iniziale, non esiste un unico ordine in cui possono essere visitati i nodi, ma ciò dipende anche dall'ordine in cui vengono considerati i successori di ciascun nodo (nell'implementazione che abbiamo, quindi, dipende dall'ordine in cui sono date le coppie nella rappresentazione del grafo). Ad esempio, nel caso della visita in profondità a partire dal nodo 1, i nodi possono essere visitati nell'ordine 1,4,6,7,5,3,2, ma anche 1,2,6,5,4,7,3. Con la rappresentazione del grafo `grafo1` data sopra, si avrà:

```
# depth_first_collect grafo1 1;;
- : int list = [3; 7; 4; 5; 6; 2; 1]

# breadth_first_collect grafo1 1;;
- : int list = [7; 5; 6; 4; 3; 2; 1]
```

Ricerca di un nodo raggiungibile dal nodo di ingresso

Ovviamente, la visita di un grafo a partire da un nodo di ingresso `start` può avere obiettivi diversi da quello di collezionare i nodi raggiungibili da `start`. Consideriamo, ad esempio, il problema di determinare se dal nodo di ingresso è raggiungibile un nodo che soddisfa un dato predicato. Se la ricerca ha successo, si vuole riportare un tale nodo, altrimenti si avrà un fallimento. In questo caso, si hanno due casi di terminazione: con fallimento (quando *pending* è vuota) o con successo, quando si visita ul nodo che soddisfa il predicato. In questo caso viene riportato tale nodo.

```
(* search_node: 'a graph -> 'a -> ('a -> bool) -> 'a *)
(* search_node g x p = nodo che soddisfa p e raggiungibile da x in g *)
(* search: 'a list -> 'a list -> 'a *)
(* search visited pending = nodo che soddisfa p, raggiungibile da
         uno dei nodi in pending senza passare per nodi in visited *)

exception NotFound

let search_node graph start p =
    let rec search visited = function
        [] -> raise NotFound
      | n::rest ->
            if List.mem n visited
            then search visited rest
            else if p n then n
                 else search (n::visited)
                             ((successori n graph) @ rest)
                           (* oppure: (vicini n graph) @ rest
                             se il grafo non e' orientato *)
    in search [] [start]
```

La funzione sopra definita può essere utilizzata per scopi diversi, applicandola ad opportuni predicati. Ad esempio, se si vuole sapere se esiste un cammino nel grafo `g` dal nodo `start` a un nodo che sia un pozzo, e si vuole sapere qual è un tale nodo, la funzione si può richiamare con `search_node g start (fun x -> successori x g = [])`.

```
# search_node grafo1 1 (fun x -> successori x grafo1  = []);;
- : int = 7
```

Se si vuole determinare se esiste un cammino da un nodo `start` a un nodo `m` nel grafo `g` (cioè se `m` è raggiungibile da `start`), si utilizzerà il predicato "essere uguale a `m`": `search_node g start (fun x -> x=m)`. Ovviamente, utilizzando `search_node`, possiamo definire una funzione che riporti un booleano e non fallisca mai:

```
(* raggiungibile: 'a graph -> 'a -> 'a -> bool *)
(* raggiungibile g start m = m e' raggiungibile da start in g *)
let raggiungibile g start m =
  try m = search_node g start ((=) m)
  with NotFound -> false

(* o anche *)
let raggiungibile g start m =
  try
    let _ = search_node g start ((=) m) in true
  with NotFound -> false
```

Si osservi che la funzione `raggiungibile` non si può utilizzare per verificare se esiste un ciclo su un nodo dato, perché `raggiungibile g start start` riporta sempre `true`. In questo caso occorre modificare l'algoritmo: si deve partire dai successori del nodo di ingresso, senza inserire quest'ultimo inizialmente tra i nodi visitati:

```
(* esiste_ciclo: 'a graph -> 'a -> bool *)
(* esiste_ciclo g start = true se esiste un ciclo su start in g *)
(* cerca: 'a list -> 'a list -> bool
   cerca visited listanodi = true se da uno dei nodi in listanodi
     si puo' raggiungere start senza passare per nodi in visited *)
let esiste_ciclo grafo start =
  let rec cerca visited = function
      [] -> false
    | n::rest ->
        if List.mem n visited
        then cerca visited rest
        else n=start ||
          cerca (n::visited)
            ((successori n grafo)@rest)
  in cerca [] (successori start grafo)
```

3.4.4 Ricerca di un cammino in un grafo

Affrontiamo in questo paragrafo il problema della ricerca di un cammino in un grafo, che può considerarsi la versione più generale del problema dell'attraversamento di un labirinto (questa volta rappresentato mediante un grafo). Il problema consiste nel determinare un cammino aciclico (una lista di nodi, senza ripetizioni) da un nodo iniziale ad un nodo che soddisfa una data proprietà. Non si richiede che il cammino sia tra i più brevi.

La differenza fondamentale, rispetto alla ricerca di un cammino in un albero n-ario, consiste nel fatto che un grafo può contenere cicli ed è quindi necessario tenere traccia dei nodi già visitati. Come nel caso degli alberi n-ari, si utilizzeranno due

funzioni ausiliarie, mutuamente ricorsive: una che esegue la ricerca a partire da un singolo nodo, e l'altra che ricerca a partire da un qualsiasi nodo di una lista. Entrambe avranno come argomento anche la lista di nodi già visitati.

Nel caso della ricerca a partire da un singolo nodo `n`, avremo due casi di terminazione: uno con fallimento (quando il nodo è già stato visitato), e uno con successo (quando il nodo soddisfa la proprietà data). Se non si termina, si proseguirà la ricerca a partire da uno dei successori (o vicini) di `n` (inserendo `n` tra i nodi visitati), utilizzando la funzione che si applica a liste di nodi. Al cammino eventualmente riportato da quest'ultima si dovrà inserire in testa il nodo `n`. Nel codice che segue assumiamo che il grafo sia non orientato (e utilizziamo quindi la funzione `vicini`).

```
exception NotFound

(* search_path : 'a graph -> 'a -> ('a -> bool) -> 'a list *)
(* search_path g start p = cammino in g da start fino a un nodo che
                           soddisfa p *)
(* funzioni ausiliarie: *)
   (* ricerca a partire da un singolo nodo *)
   (* from_node: 'a list -> 'a -> 'a list
      from_node visited n = cammino che non passa per nodi in visited,
                            dal nodo n fino a un nodo che soddisfa p *)
   (* ricerca a partire da una lista di nodi, tutti vicini di uno
      stesso nodo *)
   (* from_list: 'a list -> 'a list -> 'a list
      from_list visited nodes = cammino che non passa per nodi in visited,
                     e che parte da un nodo in nodes e arriva a un nodo che
                     soddisfa p *)
let search_path graph start p =
  let rec from_node visited n =
    if List.mem n visited
    then raise NotFound
    else if p n then [n]   (* il cammino e' trovato *)
         else n::from_list (n::visited) (vicini n graph)
  and from_list visited = function
      [] -> raise NotFound
    | n::rest -> (* provo a passare per n, ma se fallisco
                    cerco ancora passando per uno dei nodi
                    di rest *)
       try from_node visited n
       with NotFound -> from_list visited rest
  in from_node [] start
```

Si può osservare che la lista `visited` contiene alla fine il cammino rovesciato, e sfruttare questo fatto in un'implementazione alternativa:

```
let search_path graph start p =
  let rec from_node visited n =
    if List.mem n visited
    then raise NotFound
    else if p n then List.rev (n::visited)
         else from_list (n::visited) (vicini n graph)
  and from_list visited = function
      [] -> raise NotFound
    | n::rest ->
        try from_node visited n
        with NotFound -> from_list visited rest
   in from_node [] start
```

Si noti che la ricerca di un cammino mediante *backtracking* avviene necessariamente in profondità. Tuttavia, a differenza dell'algoritmo di visita in profondità, la lista dei nodi da visitare non "mescola" i vicini di un nodo con i suoi fratelli, ma `from_list` viene sempre applicata a una lista di nodi che sono tutti vicini di uno stesso nodo.

Consideriamo ora il problema di generare, se esiste, un cammino da un nodo di ingresso `start` a un nodo finale `goal` di lunghezza (numero di nodi) non superiore a un limite fissato. Si può facilmente modificare il codice di `search_path`, aggiungendo un parametro alle funzioni locali che rappresenta il numero massimo di nodi che ancora si possono aggiungere al cammino:

```
(* path_max : 'a graph -> 'a -> goal -> int -> 'a list *)
(* path_max g start goal maxlen = cammino in g di lunghezza massima
      maxlen da start a goal *)
(* from_node: 'a list -> 'a -> int -> 'a list
   from_node visited n len = cammino da n a goal di lunghezza non
                             superiore a len
   from_list: 'a list -> int -> 'a list -> 'a list
   from_list visited len nodes = cammino da uno dei nodi in nodes a goal,
              di lunghezza non superiore a len *)
let path_max graph start goal maxlen =
  let rec from_node visited n len =
    if List.mem n visited || len <= 0
    then raise NotFound
    else if n=goal then [n]
    else n::from_list (n::visited) (len-1) (vicini n graph)
  and from_list visited len = function
      [] -> raise NotFound
    | n::rest ->
        try from_node visited n len
        with NotFound -> from_list visited len rest
  in from_node [] start maxlen
```

Si osservi che, nel caso della prima implementazione della funzione `search_path`, si potrebbe anche aggiungere il nodo `n` ai nodi visitati quando, catturando l'eccezione `NotFound` in `from_list`, si passa a considerare i suoi fratelli, cioè modificando la funzione `from_list` come segue:

```
.....
  and from_list visited = function
      [] -> raise NotFound
    | n::rest ->
        try from_node visited n
        with NotFound -> from_list (n::visited) rest
  in from_node [] start
```

Ma ci sono dei casi in cui ciò non va fatto. Consideriamo, ad esempio, il problema di trovare un cammino non ciclico, a partire da un nodo di ingresso `start`, che passi per tutti i nodi di una lista data, ed eventualmente anche altri nodi. Se il grafo è rappresentato dalla variabile `grafo2` così definita:

```
let grafo2 = [(1,2);(1,3);(3,4);(4,2)]
```

il nodo di ingresso `start` è 1 e la lista è [2;3;1], l'algoritmo cercherà, partendo da 1, prima un cammino che passa per 2, che non ha successori, quindi fallisce. Il backtracking lo porterà a considerare il cammino che passa per 3, prosegue con 4 e torna a 2. Questo è il risultato voluto, ma se quando si considera il cammino che passa per 3 si è aggiunto 2 ai nodi già visitati, si avrà un fallimento. Quindi la definizione della funzione deve essere come segue:

```
(* cammino_con_nodi : 'a graph -> 'a  -> 'a list -> 'a list *)
(* cammino_con_nodi g start lista = cammino in g che parte da start
       e contiene tutti i nodi di lista (oltre eventualmente ad altri) *)
(* from_node: 'a list -> 'a list -> 'a -> 'a list
   from_node visited lst x = cammino senza cicli che parte da x
      e passa per tutti i nodi di lst, senza passare per nodi di
      visited.
   from_list: 'a list -> 'a list -> 'a list-> 'a list
   from_node visited lst listanodi = cammino senza cicli che parte da
      un nodo di listanodi e passa per tutti i nodi di lst,
      senza passare per nodi di visited. *)
exception NotFound
let cammino_con_nodi g start lista =
  let rec from_node visited lst x =
    if List.mem x visited
    then raise NotFound
    else
      if lst=[x] then [x]
         (* lista esaurita => cammino trovato *)
      else x::from_list (x::visited)
                (List.filter ((<>) x) lst)
                (successori x g)
  and from_list visited lst = function
      [] -> raise NotFound
    | x::rest ->
        try from_node visited lst x
        with NotFound -> from_list visited lst rest
  in from_node [] lista start

# cammino_con_nodi grafo2 1 [2;3;1];;
- : int list = [1; 3; 4; 2]
```

Se invece di modifica la definizione di `from_list` come riportato sotto, la stessa applicazione di `cammino_con_nodi` solleverà l'eccezione `NotFound`.

```
...
  and from_list visited lst = function
      [] -> raise NotFound
    | x::rest ->
        try from_node visited lst x
        with NotFound -> from_list (x::visited) lst rest
                                (* ============ *)
```

Una situazione analoga si avrebbe quando si cerca un cammino, in un grafo etichettato da interi (anche negativi), da un nodo di ingresso `start` a `goal`, il cui peso

(somma dei nodi che lo compongono) non superi un limite dato. Si modifichi, per esercizio, la funzione `path_max` in modo da implementare una soluzione a questo problema.

Ricerca di un cammino ciclico

Vogliamo ora scrivere un programma che, mediante lo stesso tipo di algoritmo appena visto, dato un grafo e un nodo `start`, riporti un cammino ciclico su `start`, se esiste, un errore altrimenti. L'algoritmo è sostanzialmente lo stesso: quel che cambia è l'inizializzazione dei dati. La ricerca parte infatti dai successori del nodo di ingresso, che non viene incluso tra i nodi visitati (altrimenti la ricerca terminerebbe subito riportando il cammino `[start]`). Infine, esso viene aggiunto in testa al cammino trovato.

```
(* ciclo : 'a graph -> 'a -> 'a list *)
(* ciclo g start = ciclo su start in g *)
(* from_node: 'a list -> 'a -> 'a list
   from_node visited n = cammino non ciclico da n a start che non passa per
         alcun nodo in visited
   from_list: 'a list -> 'a list -> 'a list
   from_node visited listanodi = cammino non ciclico da uno dei nodi in
        listanodi fino a start che non passa per alcun nodo in visited *)
let ciclo graph start =
    let rec from_node visited n =
      if List.mem n visited
      then raise NotFound
      else if n=start then [n]
      else n::from_list (n::visited) (successori n graph)
    and from_list visited = function
        [] -> raise NotFound
      | n::rest ->
          try from_node visited n
          with NotFound -> from_list (n::visited) rest
    in start::from_list [] (successori start graph)
```

Una versione alternativa si può formulare includendo `start` inizialmente tra i nodi visitati, ma eseguendo il test `n=start` prima del test `List.mem n visited`.

Appendice A

Definizioni induttive e dimostrazioni per induzione

A.1 Induzione matematica e ricorsione

Come si è visto nel paragrafo 2.1, l'insieme $\mathbb{N}$ dei numeri naturali si può definire induttivamente:

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

(ii) per ogni $n \in \mathbb{N}$, $succ(n) \in \mathbb{N}$ (dove $succ(n)$ è il successore di n);

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

La terza clausola di questa definizione è la "clausola di chiusura". Essa stabilisce che nessun altro elemento appartiene a $\mathbb{N}$, ed è necessaria perché quella data sopra definisca $\mathbb{N}$. Infatti, qualsiasi insieme contenente $\mathbb{N}$ soddisfa (i) e (ii), e le due prime clausole non bastano quindi a *definire* un insieme. Perché una descrizione possa essere considerata una definizione deve esistere uno ed un unico oggetto che la soddisfa. Se ad esempio si afferma che "un oggetto è nell'insieme A se e solo se esso è in A", questo non *definisce* A: ogni insieme A soddisfa infatti tale proprietà.

Sui numeri naturali è ben noto il *ragionamento per induzione*: per dimostrare che tutti i numeri naturali godono di una certa proprietà P, si prova che P vale per il numero 0 e che, se essa vale per un numero qualsiasi n, allora vale anche per $n + 1$. L'uso del *principio di induzione* è molto frequente. La dimostrazione esplicita di molte semplici proprietà dei numeri naturali, che normalmente vengono accettate come evidenti, richiede un ragionamento induttivo. In altri ragionamenti l'uso dell'induzione è nascosto da un "e così via". Il principio di induzione si può formulare come segue:

Principio di induzione matematica

Sia P una proprietà dei naturali. Se:

(B) *(base dell'induzione)* 0 ha la proprietà P e

(PI) *(passo induttivo)* per ogni numero naturale n, se n ha la proprietà P, allora anche $succ(n)$ ha la proprietà P,

allora P vale per ogni numero naturale.

Possiamo riformulare il principio di induzione matematica in maniera più compatta, scrivendo "$P(n)$" per "n ha la proprietà P" e "$\cdots \Rightarrow \cdots$" per "se $\cdots$ allora $\cdots$":

Se P è una proprietà dei numeri naturali tale che:

(B) $P(0)$ e

(PI) per ogni numero naturale n, $P(n) \Rightarrow P(succ(n))$,

allora per ogni numero naturale n, $P(n)$.

Quando si dimostra per induzione che una proprietà P vale per ogni $n \in \mathbb{N}$, si mostra che valgono (B) e (PI); il principio di induzione consente di concludere la tesi.

Se il principio di induzione vale per gli interi è perché essi sono definiti in modo che funzioni: il procedimento di costruzione di $\mathbb{N}$ è allo stesso tempo un procedimento di prova: intuitivamente, dimostrando (B) e (PI), si mostra che P vale per 0, poi per 1, poi per 2, e così via. In altri termini, (B) e (PI) sono strumenti che consentono, mentre si generano i numeri naturali mediante le clausole (i) e (ii) della definzione induttiva, di verificare contemporaneamente che ciascun numero generato ha P.

Per meglio giustificare la validità del principio di induzione, riformuliamo la definizione di $\mathbb{N}$ come segue:

Definizione A.1.1 $\mathbb{N}$ *è l'intersezione di tutti gli insiemi A tali che:*

(i) $0 \in A$

(ii) per ogni $n \in A$, $succ(n) \in A$

In altri termini, $\mathbb{N}$ è il più piccolo insieme A che soddisfa (i) e (ii) – "più piccolo" rispetto alla relazione di inclusione insiemistica. Richiedere che $\mathbb{N}$ sia minimo in tal senso equivale alla clausola di chiusura (iii) della definizione precedentemente considerata.

Notiamo, per inciso, che dalla definizione A.1.1 segue che per ogni numero naturale $n \neq 0$ esiste $m \in \mathbb{N}$ di cui n è il successore.

Riformuliamo inoltre il principio di induzione in termini di insiemi anziché di proprietà:

Sia $A \subseteq \mathbb{N}$. Se

(B) $0 \in A$ e

(PI) per ogni $n \in A$, $succ(n) \in A$,

allora $A = \mathbb{N}$.

Questa formulazione è basata sul principio che ogni proprietà P dei numeri naturali identifica un insieme $A = \{n \in \mathbb{N} \mid P(n)\}$ e, viceversa, ogni sottoinsieme A di $\mathbb{N}$ identifica una proprietà dei naturali, quella di "appartenere a A". La conclusione $A = \mathbb{N}$ equivale dunque a dire che $P(n)$ vale per ogni $n \in \mathbb{N}$.

Ora, (B) è uguale alla clausola (i) e (PI) alla clausola (ii) della definizione A.1.1. Quindi se A è un insieme che soddisfa (B) e (PI), allora $\mathbb{N} \subseteq A$ perché $\mathbb{N}$ è il più piccolo insieme che soddisfa (i) e (ii). Se inoltre abbiamo che $A \subseteq \mathbb{N}$, allora evidentemente $A = \mathbb{N}$.

In una dimostrazione per induzione si usa la terminologia seguente. La parte della dimostrazione nella quale si dimostra $P(0)$ (B) è chiamata la *base dell'induzione*. La prova del fatto che se vale $P(n)$ allora vale anche $P(succ(n))$ (PI) è il *passo induttivo*; all'interno del passo induttivo, l'ipotesi $P(n)$, dalla quale si deduce $P(succ(n))$ è *l'ipotesi induttiva*.

Esempio A.1.2 (Prova per induzione) *Sia $P(n)$ la seguente proprietà dei naturali:*

$$0 + 1 + 2 + ... + n = \frac{n(n+1)}{2}$$

Mostriamo che $P(n)$ vale per tutti i naturali.

(B) *$P(0)$ vale, infatti:*

$$0 = \frac{0(0+1)}{2}$$

(PI) *Supponiamo che valga $P(n)$ (ipotesi induttiva), cioè*

$$0 + 1 + 2 + ... + n = \frac{n(n+1)}{2}$$

Allora:

$$0 + 1 + 2 + 3 + ... + n + (n+1) = \frac{n(n+1)}{2} + (n+1) =$$

$$= \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2} =$$

$$= \frac{(n+1)((n+1)+1)}{2}$$

Dunque vale $P(n+1)$.

Quindi, applicando il principio di induzione, si conclude che per ogni $n \in \mathbb{N}$, $P(n)$ vale.

I postulati di Peano

Un oggetto matematico può, in generale, essere definito seguendo due diversi approcci: mediante una definizione esplicita o secondo il metodo assiomatico. Quando diciamo, ad esempio, che "$\mathbb{N}$ è l'intersezione di tutti gli insiemi tali che ..." definiamo $\mathbb{N}$ esplicitamente, identificandolo con uno specifico oggetto matematico. Di fatto, stiamo definendo $\mathbb{N}$ sulla base della teoria degli insiemi. Come è noto, lo stesso numero 0 e l'operazione *succ* sono definibili in termini insiemistici: $0 = \emptyset$, $succ(n) = n \cup \{n\}$. Sfruttando poi sempre le proprietà degli insiemi, possiamo dimostrare le proprietà di cui gode $\mathbb{N}$ (ad esempio la validità del principio di induzione).
Per contro, la definizione di oggetti matematici mediante il metodo assiomatico non dice nulla su "che cosa" siano tali oggetti, ma soltanto quali sono le proprietà di cui godono e le relazioni che tra di essi intercorrono; tali proprietà e relazioni sono enunciate mediante un sistema di *assiomi*. Gli assiomi definiscono *implicitamente* i concetti di cui essi parlano.
Il matematico italiano Giuseppe Peano propose per la prima volta, nel 1889, una sistemazione assiomatica dell'aritmetica dei numeri naturali. Tale sistema è costituito da assiomi relativi all'eguaglianza (che la descrivono come una relazione riflessiva, simmetrica e transitiva) e dai seguenti assiomi specifici per i numeri naturali:

1. $0 \in \mathbb{N}$;
2. se $n \in \mathbb{N}$ allora $succ(n) \in \mathbb{N}$;
3. $0 \neq succ(n)$ per ogni n (0 non è il successore di nessun numero);
4. se $succ(n) = succ(m)$, allora $n = m$;
5. se A è un insieme tale che $0 \in A$ e per ogni numero n, se $n \in A$ allora $succ(n) \in A$, allora $\mathbb{N} \subseteq A$ (assioma di induzione).

Come si vede, dunque, il principio di induzione matematica è un assioma del sistema di Peano. Questi assiomi definiscono implicitamente i termini $0, succ, \mathbb{N}$, per i quali non viene data alcuna definizione esplicita.

Come già osservato nel paragrafo 2.1, la definizione induttiva dei numeri naturali (e, conseguentemente, il principio di induzione) giustifica la definizione ricorsiva di funzioni sui numeri naturali. Il principio di induzione matematica è di fatto uno strumento fondamentale per dimostrare proprietà di funzioni definite ricorsivamente, dunque di programmi ricorsivi.

Consideriamo ancora, ad esempio, la definizione della funzione fattoriale:

$$\begin{aligned} fact(0) &= 1 \\ fact(n+1) &= (n+1) \times fact(n) \end{aligned}$$

La funzione fattoriale è definita per tutti i naturali e per ogni n, $factn = n!$. Ciò si dimostra per induzione come segue:

(B) $fact(0) = 1 = 0!$ è definito;

(PI) $fact(n+1) = (n+1) \times fact(n)$; per ipotesi induttiva, $fact(n) = n!$ è definito, quindi, poiché anche la somma e il prodotto sono funzioni definite per tutti i naturali, $fact(n+1)$ è definito; inoltre $fact(n+1) = (n+1) \times fact(n) = (n+1) \times n! = (n+1)!$.

Applicando il principio di induzione si conclude che per ogni naturale n, $fact(n)$ è definita e $fact(n) = n!$.

A.2 Induzione strutturale sulle liste

Come abbiamo visto, anche il tipo lista è definito induttivamente. In corrispondenza, per ragionare sulle liste si può utilizzare il principio di **induzione strutturale sulle liste**:

Sia P una proprietà delle liste. Se:

(B) Vale $P($`[]`$)$.

(PI) Per ogni lista `lst` ed elemento `x`, se vale $P($`lst`$)$ allora vale $P($`x` :: `lst`$)$.

Allora P vale per ogni lista.

L'induzione sulle liste si può vedere come una specializzazione dell'induzione matematica sulla lunghezza della lista:

$$P(n) = P \text{ vale per liste di lunghezza } n$$

(B) P vale per liste di lunghezza 0

(PI) Se P vale per liste di lunghezza n, allora P vale per liste di lunghezza $n+1$

Quindi per ogni n, P vale per liste di lunghezza n.

Una giustificazione intuitiva dell'induzione strutturale sulle liste, analoga a quella che abbiamo dato per l'induzione matematica, si basa sulla simmetria con la definizione induttiva delle liste data a pagina 56:

(i) La lista vuota, `[]`, è una α `list`;

(ii) se `x` è di tipo α e `lst` è una α `list`, allora `(x::lst)` è una α `list`;

(iii) nient'altro è una α `list`.

Data una qualsiasi lista `lst`, questa definizione consente di "dimostrare" che `lst` è appunto una lista. Ad esempio, se `lst = [5;1;3]`, allora:

(a) `[]` è una `int list` per (i);

(b) per (ii) e (a), `3::[] = [3]` è una `int list`;

(c) per (ii) e (b), `1::[3] = [1;3]` è una `int list`;

(d) per (ii) e (c), `5::[1;3] = [5;1;3]` è una `int list`.

L'induzione strutturale sulle liste ricalca di fatto le due regole (i) e (ii) che definiscono le liste. E, se valgono le due premesse del principio di induzione sulle liste, per ogni lista `lst` possiamo costruire una dimostrazione che essa gode della proprietà considerata, ricalcando la dimostrazione che essa è una lista.

Ad esempio, se la proprietà P soddisfa le condizioni (B) e (PI):

(a) $P($`[]`$)$ vale per la base dell'induzione (B);

(b) per (PI) e (a), vale $P($`[3]`$)$;

(c) per (PI) e (b), vale $P($`[1;3]`$)$;

(d) per (PI) e (c), vale $P($`[5;1;3]`$)$.

La definizione induttiva delle liste induce una relazione

$$\texttt{lst} \prec \texttt{x} :: \texttt{lst}$$

analoga a $n + 1 = succ(n)$. Anche se questa relazione non è un ordine totale, essa è una relazione *ben fondata*: non esistono catene discendenti infinite. Questo garantisce che la transizione da una lista alla sua coda, alla coda della coda, ecc. prima o poi termina con la lista vuota: la base dell'induzione.

A.2.1 La lista vuota è elemento neutro a destra di `@`

Come primo esempio di dimostrazione di una proprietà per induzione strutturale sulle liste, consideriamo la definizione ricorsiva della funzione di concatenazione `@` (che si può definire come la funzione `append` di pagina 61):

```
let rec ( @ ) prima seconda =
  match prima with
    [] -> seconda
  | x::rest ->  x::(rest@seconda)
```

Dalla stessa definizione di `@` segue immediatamente che la lista vuota è elemento neutro a sinistra di `@`: per ogni lista `lst`, `[] @ lst = lst`.

Dimostriamo che `[]` è anche elemento neutro a destra di `@`:

per ogni lista `lst`: `(lst @ []) = lst`.

(B) Se `lst = []`, allora `lst @ [] = [] @ [] = [] = lst` per definizione.

(PI) Assumiamo che (ipotesi induttiva): `(lst @ []) = lst`. Allora

`(x::lst) @ [] = x::(lst @ []) = x @ lst`

La tesi segue dunque per il principio di induzione strutturale sulle liste.

A.2.2 Complessità di `@`

Dalla definizione di `@` segue immediatamente la verità delle seguenti eguaglianze:

```
          [] @ seconda = seconda
(x::rest) @ seconda = x::(rest@seconda)
```

Se `prima` e `seconda` sono liste, indichiamo con `T(prima @ seconda)` il numero di operazioni "cons" (`::`) eseguite nella valutazione di `prima @ seconda`. Vogliamo dimostrare che, per ogni coppia di liste `prima` e `seconda`, `T(prima @ seconda) = length(prima)`, dove `length(lst)` è il numero di elementi in `lst`. La dimostrazione è per induzione strutturale sulla lista `prima`, e la proprietà P(`prima`) è "per ogni lista `seconda`, `T(prima @ seconda) = length(prima)`.

(B) Se `prima = []`, allora la valutazione di `prima @ seconda` non richiede l'esecuzione di alcuna operazione "cons", qualsiasi sia la lista `seconda`. Quindi in questo caso `T(prima @ seconda) = 0 = lenght(prima)`.

(PI) Sia `prima` una lista qualsiasi della forma `x::rest`. Per ipotesi induttiva, per ogni lista `seconda`,

`T(rest @ seconda) = lenght(rest)`.

La valutazione di `prima @ seconda` esegue tanti "cons" quanti ne esegue la valutazione di `x::(rest @ seconda)`, cioè

```
T(prima @ seconda)  = T((x::rest) @ seconda)
                    = 1 + T(rest @ seconda)
                    = 1 + length(rest)
                    = length(x::rest) = length(prima)
```

Quindi, per il principio di induzione sulle liste, per ogni lista `prima`, vale che per ogni lista `seconda`, `T(prima @ seconda) = length(prima)`.

A.2.3 Complessità di `reverse`

Consideriamo ora la funzione `reverse` definita a pagina 62. Dalla definizione segue immediatamente che:

```
        reverse [] = []
reverse (x::rest) = (reverse rest) @ [x]
```

Sia `lst` una lista e `T(reverse lst)` il numero di operazioni "cons" eseguite nella valutazione di `reverse lst`. Si vuole dimostrare che per ogni lista `lst`, se `length(lst) = n`, allora

$$\texttt{T(reverse lst)} = \frac{n^2 + n}{2}$$

(B) Sia `lst = []`, quindi $n =$ `length([])` $= 0$. Allora:

$$\texttt{T(reverse lst)} = 0 = \frac{n^2+n}{2}$$

(PI) Sia $length(\texttt{lst}) = n \geq 0$ e assumiamo che (ipotesi induttiva):

$$\texttt{T(reverse lst)} = \frac{n^2+n}{2}$$

`T(reverse (x::lst))`= numero di "cons" per eseguire `reverse(lst)`
+ numero di "cons" per eseguire l'append
+ numero di "cons" per costruire `[x]`

Dato che `length lst = length(reverse lst)`, il numero di "cons" necessari per eseguire l'append è uguale a n (per quanto dimostrato nel paragrafo precedente). Quindi:

$$\begin{aligned}
\texttt{T(reverse(x :: lst))} &= \texttt{T(reverse lst)} + n + 1 \\
&= \frac{n^2+n}{2} + n + 1 \\
&= \frac{n^2+n+2n+2}{2} \\
&= \frac{(n+1)^2+(n+1)}{2}
\end{aligned}$$

La tesi segue per il principio di induzione sulle liste.

A.2.4 Associatività di `@`

L'operazione `@` è associativa, cioè per ogni lista `uno`, `due` e `tre`:

```
(uno @ due) @ tre = uno @ (due @ tre)
```

La dimostrazione è per induzione su `uno`, cioè la proprietà P`(uno)` che si vuole dimostrare per ogni `uno` è:

per ogni lista `due` e `tre`: `(uno @ due) @ tre = uno @ (due @ tre)`

(B) La proprietà vale per `[]`, cioè per ogni lista `due` e `tre`: `([] @ due) @ tre = [] @ (due @ tre)`. Infatti:

`([] @ due) @ tre`	`= due @ tre`	per definizione di `@`
	`= [] @ (due @ tre)`	per definizione di `@`

(PI) Assumiamo, per ipotesi induttiva, che per ogni lista `due` e `tre`:
`(uno @ due) @ tre = uno @ (due @ tre)`. Allora:

```
((x::uno) @ due) @ tre
      = (x::(uno @ due)) @ tre   per definizione di @
      = x::((uno @ due) @ tre)   per definizione di @
      = x::(uno @ (due @ tre))   per ipotesi induttiva
      = (x::uno) @ (due @ tre)   per definizione di @
```

Quindi la proprietà vale per ogni lista `uno`.

A.2.5 Equivalenza di `reverse` e `rev`

Spesso è più semplice dimostrare proprietà di funzioni definite ricorsivamente piuttosto che delle corrispondenti versioni iterative. Nell'esempio che segue dimostriamo l'equivalenza della versione ricorsiva di `reverse` e quella iterativa (`rev`), definita a pagina 66. Ci basiamo sulle seguenti uguaglianze, conseguenze dirette delle definizioni delle funzioni considerate:

```
          reverse [] = []
  reverse (x::lst) = (reverse lst) @ [x]

    revto result [] = result
revto result (x::rest) = revto (x::result) rest

           rev lst = revto [] lst
```

Vogliamo dimostrare che: `rev lst = reverse lst`. Per far ciò, conviene dimostrare qualcosa più generale:

per ogni lista `result`: `revto result lst = (reverse lst) @ result`

(si veda la specifica dichiarativa di `revto` data a pagina 66).

La dimostrazione è per induzione su `lst`, e la proprietà P(`lst`) è "per ogni lista `result`, `revto result lst = (reverse lst) @ result`".

(B) Per ogni lista `result`:

```
revto result []  = result                  per def. di revto
                 = [] @ result             per def. di @
                 = (reverse []) @ result   per def. di reverse
```

(PI) Assumiamo che (ipotesi induttiva):

per ogni lista `tmp`: `revto tmp lst = (reverse lst) @ tmp`.

Allora:

```
revto result (x::lst)
    = revto (x::result) lst                 per def. di revto
    = (reverse lst) @ (x::result)           per ipotesi induttiva
                                            (con tmp = x::result)
    = (reverse lst) @ (x::([] @ result))    per def. di @
    = (reverse lst) @ ([x] @ result)        per def. di @
    = ((reverse lst) @ [x]) @ result        per l'associatività di @
    = (reverse (x::lst)) @ result           per def. di reverse
```

La tesi segue per il principio di induzione sulle liste.

Bibliografia

[1] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000. Traduzione inglese disponibile su `http://caml.inria.fr/pub/docs/oreilly-book/index.html`.

[2] G. Cousineau and M. Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1998.

[3] X. Leroy. The OCaml System, release 4.05. manual. Disponibile su `http://caml.inria.fr/pub/docs/manual-ocaml/`, 2017.

[4] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml. Functional programming for the masses*. O'Reilly France, 2014. Disponibile su `https://realworldocaml.org/`.

[5] L.C. Paulson. *ML for the Working Programmer. Second Edition*. Cambridge University Press, 1996.

[6] C. Reade. *Elements of Functional Programming*. Addison-Wesley, 1989.

[7] Wadler. Functional programming: an angry half dozen. *ACM SIGPLAN Notices*, 33(2), 1998. Disponibile anche sul sito Internet `http://homepages.inf.ed.ac.uk/wadler/topics/functional-programming.html`.

Indice delle funzioni definite