

Concrete Abstractions

Introduzione alla Programmazione

Note ad uso degli studenti di Programmazione
corso di Laurea in Informatica
Università Ca' Foscari di Venezia
anno accademico 2009/2010
a cura di
Giulia Costantini, Giuseppe Maggiore,
Sabina Rossi

Prefazione

Lo scopo di questa dispensa è di introdurre alcuni dei concetti fondamentali della programmazione funzionale. In essa vengono presentati molti esempi di programmazione funzionale, espressi nel linguaggio ML.

Il materiale contenuto in questa dispensa è tratto dal libro di testo “Concrete Abstractions: An Introduction to Computer Science using Scheme”, M. Hailperin, B. Kaiser, K. Knight, 1999 (<http://www.gustavus.edu/+max/concrete-abstractions.html>) e dalle “Note di Programmazione Funzionale” ad uso degli studenti del corso di Programmazione per la Laurea in Informatica a.a. 1998/1999 dell’Università degli Studi di Pisa, a cura di Giuseppe Manco. Parte del materiale presente in questa dispensa è tratto anche dal testo “Introduction to Functional Programming”, R. Bird e P. Wadler.

Nel corso dei vari capitoli seguiremo la seguente notazione: denoteremo ogni programma ML (ed ogni interazione col sistema) in testo dattiloscritto, mentre denoteremo funzioni ed espressioni matematiche in italico. Ad esempio, $\text{succ } x$ denota l’espressione ML relativa alla funzione successore, mentre $\text{succ } x$ denota l’espressione matematica che deriva dalla definizione ML. Spesso utilizzeremo una notazione grafica per esprimere la dimostrazione di una proprietà. Ad esempio, per dimostrare che $a \times (b + c + 0) = a \times b + a \times c$, procediamo come segue:

$$a \times (b + c + 0)$$

= { 0 è l’elemento neutro della somma }

$$a \times (b + c)$$

= { proprietà distributiva della moltiplicazione sulla somma }

$$a \times b + a \times c$$

col significato che $a \times (b + c + 0) = a \times (b + c)$ perché lo 0 è l’elemento neutro della somma, e che a sua volta $a \times (b + c) = a \times b + a \times c$ perché la moltiplicazione distribuisce sulla somma.

Capitolo 1 – Concetti Fondamentali

1.1 – Di cosa stiamo parlando?

L'informatica ruota intorno al concetto di *processo di calcolo* (o *processo computazionale*), detto anche *processo di elaborazione delle informazioni* o semplicemente *processo*. Un processo è un susseguirsi dinamico di eventi – un avvenimento. Quando il computer è occupato a fare qualcosa, c'è un processo che sta avvenendo al suo interno. Cosa differenzia un processo di calcolo da qualche altro tipo di processo (ad esempio, un processo chimico)? Anche se originariamente il termine calcolo si riferiva al “fare conti”, non è questa la vera essenza di un processo di calcolo: per i nostri scopi, una parola, ad esempio, gode dello stesso status di un numero, e la ricerca di una parola in un dizionario è un processo di calcolo, al pari (anzi di più) della somma di due numeri. Inoltre un processo non deve per forza avvenire all'interno di un computer per essere definito un processo di calcolo – potrebbe avvenire in una vecchia biblioteca, dove un utente sfoglia a mano le pagine di un dizionario.

Un processo qualunque diventa un processo di calcolo se viene studiato ignorandone la sua natura fisica. Se scegliessimo di studiare come l'utente della biblioteca sfoglia le pagine, magari piegandole fino a un certo punto e poi lasciando che la gravità le faccia cadere, staremmo considerando un processo meccanico piuttosto che uno di calcolo. Vediamo ora, invece, una descrizione computazionale del processo “l'utente cerca la parola *fiduciario* nel dizionario”:

Poiché *fiduciario* comincia con la lettera *f*, si usa l'indice del dizionario per trovare l'inizio della sezione *f*.

Poi, visto che la seconda lettera, *i*, si trova a circa un terzo dell'alfabeto, l'utente apre le pagine a circa un terzo della sezione *f*.

Trovandosi un po' dopo la parola cercata (è arrivato a *fiorido*), l'utente cerca all'indietro sfogliando le singole pagine sequenzialmente (senza più salti) fino a che non trova la parola *fiduciario*.

Si noti che anche se apparentemente ci sono alcuni termini fisici in questa descrizione (l'indice e la sezione), la cosa interessante dell'indice ai fini di questa descrizione di processo non è il fatto di essere specificamente un indice, ma che esso permetta di recuperare facilmente quelle voci del dizionario che hanno una particolare lettera iniziale. Se il dizionario fosse stato memorizzato in un computer, potrebbe comunque avere un indice, cioè una struttura che permetta l'operazione di recuperare facilmente delle informazioni, e potremmo ancora usare sostanzialmente lo stesso processo.

Ci sono diverse domande che vi potreste porre sui processi di calcolo, come ad esempio:

Come possiamo descriverne uno o come possiamo specificare quello che vogliamo eseguire?

Come possiamo dimostrare che un processo ha un effetto particolare?

Come si fa a scegliere un processo tra diversi che producono lo stesso effetto?

Ci sono dei risultati che non è possibile raggiungere, a prescindere dal processo?

Come possiamo costruire una macchina che esegua automaticamente uno specifico processo?

Quali processi del mondo naturale è conveniente analizzare in termini di processi di calcolo?

In questa dispensa cercheremo di dare una risposta a tutte queste domande. Il nostro obiettivo principale, tuttavia, non è tanto quello di rispondere alle domande degli informatici, quanto piuttosto quello di fornire il giusto approccio circa il modo in cui formulare e affrontare tali questioni.

Poiché parleremo così tanto di processi, avremo bisogno di una notazione per descriverli. Le descrizioni sono chiamate *programmi*, e la notazione è chiamata *linguaggio di programmazione*. Per la maggior parte di questo libro utilizzeremo il linguaggio di programmazione *ML*, nelle sue implementazioni *CaML* e *F#*. Un vantaggio di ML è che la sua struttura è facile da imparare; descriveremo la sua struttura di base nelle sezioni successive. Man mano che la vostra comprensione dei processi di calcolo e dei dati su cui essi operano cresce, accrescerà anche la vostra comprensione di come questi processi e dati possono essere descritti in ML.

SCHEDA – F# vs CaML

Tutti gli esempi trattati nel corso della dispensa (se non indicato esplicitamente) funzionano sia con un qualsiasi compilatore CaML (ad esempio quello “ufficiale” INRIA), sia con F#. La differenza principale tra i due linguaggi sta nel fatto che F# è un linguaggio appartenente alla famiglia più generale dei linguaggi *Dot Net* (o .Net) di *Microsoft*. Dot Net è un vasto insieme di librerie che coprono le più svariate applicazioni, dalla grafica accelerata 3d (*DirectX/XNA*) a semplici GUI (*Windows Forms*) fino ad applicazioni per la gestione del lato server di siti web (*ASP .Net*) o database. F# può essere usato anche su *Linux/Unix/Mac* tramite *Mono*, un porting open source delle librerie .Net curato da *Novell*. F# su Microsoft Windows può sfruttare una integrazione di primo livello nell’IDE *Visual Studio* (dalla versione 2005 in poi), essendo stato ammesso nell’insieme dei linguaggi di prima classe supportati da tale ambiente di sviluppo. Questo significa che scrivere una applicazione multilinguaggio (*C++/C#/Visual Basic/F#*) e mantenerla può essere effettuato con estrema facilità nell’ambiente Visual Studio senza incontrare alcun ostacolo. La base comune .Net fa sì che ognuno di questi linguaggi possa interagire con librerie scritte in un differente linguaggio .Net come se fossero state scritte nel linguaggio di partenza. In parole povere una libreria F# aperta da un progetto C# apparirà come se fosse stata scritta in C#, e lo stesso il viceversa.

CaML d’altro canto viene mantenuto dall’INRIA come progetto *Open Source* (si veda <http://caml.inria.fr>) e dunque può essere preferibile in taluni contesti in cui l’apertura del sorgente sia fortemente desiderata o persino necessaria. Aderisce ad uno standard estremamente preciso e che viene modificato con cautela ed attenzione. Inoltre CaML, avendo molti più anni di diffusione alle spalle ha il vantaggio di avere moltissime risorse (tutorial/samples/testi ad ogni livello) già scritte e facilmente reperibili.

Un ulteriore vantaggio di ML (che vale per la maggior parte dei linguaggi di programmazione utili) è che ci permette di fare accadere i processi, poiché ci sono macchine in grado di leggere la nostra notazione e di eseguire i processi descritti con quella notazione. Il fatto che le nostre descrizioni astratte di un processo possano venire concretamente realizzate (eseguendo il processo in questione) è un aspetto positivo e gratificante dell’informatica e riflette una parte del titolo di questo libro. Questo significa anche che l’informatica è, in qualche misura, una scienza sperimentale.

Tuttavia, l’informatica non è puramente sperimentale, perché siamo in grado di applicare strumenti matematici per analizzare i processi computazionali. Fondamentale per questa analisi matematica è poter modellare questi processi; a tal scopo, nella Sezione 1.2 descriviamo il cosiddetto *modello di*

sostituzione. Questo modello astratto di un processo concreto riflette un'altra parte del titolo del libro che si fonda sul processo di calcolo vero e proprio.

SCHEDA – USO RESPONSABILE DEL COMPUTER

Se si utilizza un computer condiviso, si dovrebbe riflettere a proposito dell'accettabilità sociale del vostro comportamento. Il punto più importante da tenere presente è che la fattibilità di un'azione e la sua accettabilità sono questioni molto diverse.

Si può anche essere tecnicamente in grado di rovistare tra i files di altre persone senza la loro approvazione. Tuttavia, questo atto è generalmente considerato come andare per le strade e provare ad aprire le serrature di tutte le porte, per poi entrare se ne trovi una aperta.

In alcuni casi potresti non sapere che cosa è accettabile e cosa non lo è. Se avete dei dubbi e volete sapere se una determinata azione è legale, etico e socialmente accettabile, abbondate sul lato della prudenza. In ogni caso, chiedete ad un amministratore di sistema, ad un docente o ad un tutor.

1.2 – La programmazione funzionale

I linguaggi di programmazione sono detti *funzionali* quando la tecnica fondamentale per strutturare i programmi è quella di utilizzare delle *funzioni*, e il controllo dell'esecuzione dei programmi viene effettuato per mezzo dell'*applicazione* (di funzioni). Programmare in un linguaggio funzionale consiste nel costruire definizioni e usare il calcolatore per valutare espressioni. L'obiettivo primario di un programmatore è quello di disegnare una funzione, normalmente espressa in termini matematici e che può fare uso di un certo numero di funzioni ausiliarie, per risolvere un dato problema. Il ruolo del calcolatore è di valutare espressioni e stampare i risultati. In quest'ottica, il calcolatore agisce più che altro come una calcolatrice tascabile. Tuttavia, quello che distingue un calcolatore da una normale calcolatrice, è la possibilità per il programmatore di ampliare le capacità del calcolatore, per mezzo di definizioni di funzioni "nuove". Le espressioni che contengono le funzioni nuove, definite dal programmatore, sono valutate utilizzando le definizioni date tramite regole di semplificazione ("riduzione") per convertire le espressioni in forme semplici (ovvero, non ulteriormente riducibili).

Una tipica caratteristica della programmazione funzionale è la possibilità di guardare ad un'espressione come ad un valore ben definito, indipendentemente dal metodo di valutazione utilizzato dal calcolatore. In altre parole, il significato di un'espressione è il suo valore matematico, ed il compito del calcolatore è semplicemente quello di ottenere tale valore. Da ciò segue che le espressioni in un linguaggio funzionale possono essere costruite e manipolate come ogni altra espressione matematica, utilizzando leggi algebriche.

1.3 – Programmare con ML

Per esemplificare alcuni dei concetti sopra esposti, immaginiamo di essere di fronte al terminale, con il sistema che aspetta che un'espressione da valutare sia digitata. Sul monitor questa situazione è rappresentata dal segnale di attesa di input

#

all'inizio di una linea bianca. Noi possiamo inserire un'espressione da valutare, seguita dal simbolo “;;”, e il sistema risponderà fornendo il risultato della valutazione dell'espressione, seguito da un nuovo segnale di attesa di input su una nuova linea.

Per maggiore comodità, è possibile memorizzare il codice in un file di testo, con l'editor preferito, ed eseguirlo con il comando

```
# load "nome_del_file";;
```

Per esempio, possiamo creare il file prova.ml contenente la riga di codice

```
print_endline "Hello World";;
```

Quindi:

```
# load "prova.ml";;
Hello World
- : unit = ()
- : unit = ()
```

Il più semplice programma ML è un singolo numero. Se si chiede a ML di elaborare tale programma, esso, come risposta, vi ritornerà semplicemente il numero inserito. Chiamiamo ciò che ML fa “trovare il valore dell'espressione” da voi fornita, o più semplicemente, “valutare” l'espressione. Per esempio, usando CAML LIGHT, release 0.80, abbiamo:

```
# 12;;
- : int = 12
```

mentre con F# otteniamo:

```
# 12;;
val it : int = 12
```

In entrambi i casi, la prima linea che vedete qui sopra è stata scritta da un umano, mentre la seconda è la risposta data dal computer. In questo caso il calcolatore ha risposto valutando l'espressione digitata, e quindi fornendoci delle informazioni sull'espressione. Poiché tale espressione è un numero intero, il sistema ha risposto semplicemente mostrando in output l'espressione ed il suo tipo. Il numero decimale 12 è un'espressione nella sua forma più semplice, e non è possibile applicare un ulteriore processo di riduzione. (Si noti la differenza tra CAML e F# nella risposta data dal computer. Per semplicità, nel resto della dispensa faremo riferimento alla sintassi CAML, ed evidenzieremo le differenze con F# se rilevanti).

Lo stesso meccanismo funziona anche per altri tipi di numeri: numeri negativi, frazioni e reali:

```
# -7;;
- : int = -7
```

```
# 1/3;;
- : int = 0
```

```
# 1.0/.3.0;;
- : float = 0.3333333333

# 1/3.0;;
> ^^^
This expression has type float,
but is used with type int.

# 3.1415927;;
- : float = 3.1415927
```

Possiamo anche digitare un'espressione lievemente più complicata:

```
# 6 + 4;;
- : int = 10
```

In tal caso il calcolatore può semplificare l'espressione digitata calcolandone la somma.

Altri tipi di espressioni sono più interessanti da valutare. Ad esempio, il valore di un [nome](#) è ciò per cui quel nome sta. A breve vedremo come noi possiamo dare un nome alle cose, ma quando avviamo CAML ci sono già molti nomi attivi. La maggior parte sono nomi per le [funzioni](#), o [procedure](#); ad esempio, il nome [sqrt](#) è il nome di una funzione, così come lo è il nome [+](#). Se valutiamo una di esse, vedremo una rappresentazione stampata della funzione corrispondente:

```
# sqrt;;
- : float -> float = <fun>
```

Le funzioni prendono una serie di parametri di input, separati da una freccia (\rightarrow) e restituiscono un valore di qualche tipo. Ad esempio la funzione [sqrt](#) prende in input un numero decimale e ritorna la sua radice quadrata.

Le funzioni però non servono tanto ad essere osservate, come abbiamo appena fatto, bensì sono fatte per essere usate. Il nostro modo di usare una funzione consiste nell'applicarla a dei valori. Per esempio, la funzione [sqrt](#) può essere applicata ad un numero decimale per calcolare la sua radice quadrata. Usiamo una funzione scrivendone il nome seguito dai valori cui si vuole applicarla:

```
# sqrt 9.0;;
- : float = 3.0
```

Le funzioni il cui nome è un operatore, come ad esempio la funzione somma [+](#), possono essere applicate in due modi:

<i>notazione prefissa</i> ,	ossia	(prefix +) 2 3	in CAML
		(+) 2 3	in F#
<i>notazione infissa</i> ,	ossia	2 + 3	

Le due notazioni sono assolutamente equivalenti, nonostante la notazione infissa appaia più intuitiva mentre la notazione prefissa metta più in evidenza il fatto che si sta applicando una funzione qualunque. Usando CAML, otteniamo dunque:

```
# prefix +;;
- : int -> int -> int = <fun>

# prefix +.;;
- : float -> float -> float = <fun>
```

Mentre, usando F# si ottiene:

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

# (+.);;
val it : float -> float -> float = <fun>
```

La funzione denominata ([prefix +](#)) (in questo caso le parentesi sono facoltative) può essere applicata a due interi da aggiungere.

```
# prefix + 2 3;;
- : int = 5

# prefix +. 2.0 3.0;;
- : float = 5.0
```

SCHEDA: LA NOTAZIONE PREFISSA

La notazione prefissa degli operatori viene realizzata in CaML premettendo il nome [prefix](#) al nome dell'operatore (ad esempio, [prefix +](#)) mentre F# richiede che l'operatore venga racchiuso tra due parentesi rotonde (ad esempio [\(+\)](#)).

SCHEDA: + VS +.

Mentre CaML richiede di distinguere tra la somma di interi [+](#) e la somma di decimali [+.](#), F# permette di usare il simbolo [+](#) per sommare tutto ciò per cui ha senso effettuare una somma: interi, decimali, matrici, etc... Le versioni tradizionali di ML mantengono netta la distinzione tra la funzione che somma due numeri interi e quella che somma due numeri decimali.

Una applicazione di funzione è sempre costituita da una lista di espressioni, eventualmente circondate da parentesi e separate da spazi. Il primo valore dell'espressione è tipicamente la funzione da applicare (a parte quando si usa la notazione infissa) mentre i valori delle rimanenti espressioni sono ciò su cui la funzione va applicata. Le applicazioni di funzioni sono esse stesse espressioni, che possono così essere annidate:

```
# sqrt((prefix +.) 3.0 6.0);;
- : float = 3.0
```

oppure (notazione infissa):

```
# sqrt (3.0 +. 6.0);;
- : float = 3.0
```

Qui il valore dell'espressione ((prefix +.) 3.0 6.0) è 9.0, ed è il valore a cui la funzione denominata `sqrt` viene applicata. (In modo più succinto possiamo dire che 9.0 è l'argomento della funzione `sqrt`.)

Vi sono moltissime altre utili procedure ciascuna con il suo nome, ad esempio alcune procedure applicabili a numeri interi sono `*` per la moltiplicazione, `-` per la sottrazione e `/` per la divisione. Le funzioni equivalenti applicabili a numeri decimali sono `* .`, `- .` e `/ .`.

SCHEDA: L'OPERATORE (*)

In F# l'operatore `(*)` non può essere usato in notazione prefissa poichè `(* ... *)` è la sintassi con cui in ML si denotano i *commenti*, ossia porzioni di testo che spiegano e annotano il codice (per il solo beneficio del programmatore che li legge: il compilatore li ignora totalmente).

ESERCIZIO 1.1

Qual è il valore di ciascuna delle seguenti espressioni? Si dovrebbe essere in grado di svolgerle a mente, ma controllare le risposte utilizzando un interprete ML sarà un buon modo per cominciare a prendere dimestichezza con i meccanismi del sistema.

- a. `(3 * 4)`
- b. `((prefix +) 5 3) * ((prefix -) 5 3))`
- c. `((prefix /) ((prefix +) (((prefix -) 17 14) * 5) 6) 7)`
- d. `((prefix -) ((prefix +) ((prefix +) ((prefix -) 17 14) 5) 6) 7)`

E' uso comune suddividere le espressioni complesse, come ad esempio quella dell'Esercizio 1.1 d, in più linee usando l'indentazione per chiarire meglio la struttura dell'espressione:

```
((prefix -) ((prefix +) ((prefix +) ((prefix -) 17 14)
                           5)
                           6)
                           7)
```

Una tale organizzazione contribuisce a rendere più chiaro quale operazione viene applicata a cosa.

Ora che abbiamo maturato una certa esperienza nel maneggiare entità che hanno già un nome, dovremmo imparare ad assegnare noi stessi nomi ad entità. Tale operazione si chiama *definizione*. L'aspetto interessante della programmazione funzionale è proprio la capacità di costruire definizioni. La forma generale di una definizione è la seguente:

```
#let nome = espressione
```

Per prima cosa l'espressione viene valutata. Quindi il nome diventa un "alias" per quel valore dell'espressione. Ad esempio:

```
# let pi = 3.1415927;;
pi : float = 3.1415927
```

Valuta l'espressione 3.1415927 e ne assegna il valore al nome `pi`. Adesso il nome `pi` potrà essere usato come un'espressione, sia da solo che all'interno di espressioni più complesse:

```
# let radius = 2.0;;
radius : float = 2.0

# let area = radius *. radius *. pi;;
area : float = 12.56

# area;;
- : float = 12.56
```

In questo caso abbiamo introdotto due definizioni: abbiamo associato al nome `radius` il raggio di un cerchio e alla variabile `area` l'area del cerchio, calcolata mediante la formula $\text{raggio}^2 \times \pi$.

Finora abbiamo adoperato i nomi per memorizzare e riutilizzare il risultato di una computazione. Sarebbe però utile poter assegnare a dei nomi anche dei *metodi di computazione*, ad esempio il metodo con cui si calcola l'area di un cerchio. È infatti molto scomodo dover scrivere:

```
# let r' = 2.0;;
r' : float = 2.0

# let A' = r' *. r' *. pi;;
A' : float = 12.56

# A';;
- : float = 12.56

# let r'' = 1.75;;
r'' : float = 1.75

# let A'' = r'' *. r'' *. pi;;
A'' : float = 9.621127644

# A'';;
- : float = 9.621127644
```

Quando chiaramente vorremmo poter scrivere:

```
# let r' = 2.0;;
r' : float = 2.0

# let A' = circle_area r';;
A' : float = 12.56

# let r'' = 1.75;;
r'' : float = 1.75

# let A'' = circle_area r'';;
A'' : float = 9.621127644
```

per ottenere lo stesso effetto, definendo la formula per l'area del cerchio una volta sola. Un metodo di computazione che abbiamo già visto è il metodo `sqrt` per il calcolo della radice quadrata.

Un'espressione che denota una computazione è chiamata *λ – expression*, *espressione lambda* o *funzione*. Nel nostro esempio l'espressione lambda che denota il calcolo dell'area del cerchio sarebbe:

```
# fun radius -> radius *. radius *. pi;;
- : float -> float = <fun>
```

A parte la parola riservata `fun`, che denota la dichiarazione di una espressione lambda, osserviamo che una funzione è composta da una serie di parametri (nel nostro caso `radius`) e una espressione di ritorno (tipicamente espressa in termini di questi parametri) chiamata *corpo della funzione*. Parametri e corpo della funzione sono separati dal simbolo `->`, che rappresenta una freccia.

Possiamo già applicare la nostra funzione, ad esempio passandole come parametro `3.0`:

```
# let A = (fun radius -> radius *. radius *. pi) 3.0;;
A : float = 28.26
```

In questo caso ad `A` verrà assegnato il valore del corpo della funzione in cui al parametro `radius` viene sostituito il valore `3.0`.

Chiaramente non ne traiamo ancora granché vantaggio. Poiché però una espressione lambda è una espressione esattamente come un numero, possiamo associarla ad un nome come abbiamo fatto per `r'`, `r''` o `pi`:

```
# let circle_area = (fun radius -> radius *. radius *. pi);;
circle_area : float -> float = <fun>
```

Ora possiamo finalmente calcolare l'area di un cerchio senza doverne riscrivere ogni volta l'espressione:

```
# let A' = circle_area r';;
A' : float = 12.56

# A';;
- : float = 12.56

# let A'' = circle_area r'';;
A'' : float = 9.621127644

# A'';;
- : float = 9.621127644
```

Vediamo qualche altro semplice esempio. Supponiamo di voler definire la funzione *square* che associa ad un numero il suo quadrato, e la funzione *min* che calcola il minimo tra due numeri interi. Nella notazione matematica, tali funzioni sono espresse come segue:

$$\text{square}(x) = x^2$$

$$\min(x, y) = \begin{cases} x & \text{se } x \leq y \\ y & \text{se } x > y \end{cases}$$

Scriveremo dunque:

```
# let square = (fun x -> x * x);;
square : int -> int = <fun>

# let min = (fun (x:int) (y:int) -> if x > y then y else x);;
min : int -> int -> int = <fun>
```

Per il momento non discutiamo la sintassi utilizzata per le definizioni; è importante comunque notare come le funzioni siano definite facendo uso di altre espressioni, che possono contenere variabili (denotate con i simboli x e y). Possiamo ora scrivere:

```
# square (3+4);;
- : int = 49

# min 3 4;;
- : int = 3

# min (square 5) (4+9);;
- : int = 13
```

Lo scopo di una definizione è quello di introdurre un legame tra un nome ed un valore. Nelle definizioni viste sopra, l'identificatore `square` è associato alla funzione che calcola il quadrato del suo argomento, e l'identificatore `min` è associato alla funzione che restituisce il più piccolo tra due numeri. Un insieme di legami (*bindings*) è chiamato *ambiente* o *contesto di valutazione*. Le espressioni sono sempre valutate in qualche contesto e possono contenere occorrenze dei nomi trovati in quel contesto. Ovviamente, il valutatore userà le definizioni associate ai nomi per semplificare le espressioni da valutare. In ogni momento il programmatore può aggiungere o modificare definizioni nel contesto attuale.

Per esempio, supponiamo di voler aggiungere al contesto attuale, comprendente le definizioni di `square` e `min`, le seguenti definizioni:

```
# let lato = 12;;
lato : int = 12

# let area_quadrato = square lato;;
area_quadrato : int = 144
```

Tali definizioni introducono due costanti numeriche, `lato` e `area_quadrato`. Si noti che la definizione di `area_quadrato` dipende dalla funzione `square`, definita precedentemente. Tali costanti possono a questo punto essere utilizzate in altre espressioni:

```
# min (area_quadrato + 4) 150;;
- : int = 148
```

Per riassumere quanto visto finora, possiamo quindi evidenziare tre aspetti salienti:

1. È possibile interagire col sistema sottponendo delle espressioni da valutare;
2. È possibile utilizzare il costrutto `let` per costruire definizioni;
3. Le definizioni sono delle equazioni tra espressioni e descrivono funzioni matematiche.

ESERCIZIO 1.2

Usando la definizione di `square`, definire una funzione che calcola la quarta potenza di un numero.

SOLUZIONE

```
# let square = (fun x -> x * x);;
square : int -> int = <fun>

# let square2 = (fun x -> square(square x));;
square2 : int -> int = <fun>

#square2 2;;
- : int = 16

#square2 3;;
- : int = 81

#square2 4;;
- : int = 256

oppure

# let square2 = (fun x -> (square x) * (square x));;
square2 : int -> int = <fun>
```

ESERCIZIO 1.3

- a. Definire la funzione `tax(x)` che calcola il prezzo da assegnare ad un oggetto di valore `x`, a cui vogliamo sommare le tasse del 5% del valore.
- b. Calcolare il prezzo inclusivo delle tasse di due oggetti di valore 1.29€ e 2.40€.

SOLUZIONE

```
# let tax = (fun x -> x *. 1.05);;
tax : float -> float = <fun>

# let sommatax = (fun x y -> tax x +. tax y);;
sommatax : float -> float -> float = <fun>

# sommatax 1.29 2.40;;
- : float = 3.8745
```

Ecco un altro esempio di definizione e uso di una funzione. I parametri sono due, `radius` e `height`, ossia la funzione è definita per essere applicata a due valori. Il primo verrà sostituito a

tutte le occorrenze di `radius` nel corpo della procedura, mentre il secondo verrà sostituito dove compare `height`:

```
# let square = fun x -> x *. x;;
square : float -> float = <fun>

# let cylinder_volume =
    fun radius height -> 3.1415927 *. (square radius) *. height;;
cylinder_volume : float -> float -> float = <fun>

# cylinder_volume 5.0 4.0;;
- : float = 314.15927
```

Avendo già definito la funzione `square` la possiamo usare nel corpo di `cylinder_volume` senza dover riscriverne la definizione e ottenendo un codice più ordinato. Se però avessimo scritto prima la funzione `cylinder_volume`, allora non avremmo più potuto usare `square` in quanto la sua definizione non sarebbe stata visibile alle parti precedenti del codice.

Possiamo studiare come il computer ha prodotto il risultato 314.15927 consultando la Figura 1.1. Nel diagramma la freccia denota la conversione di un problema in uno equivalente, ossia avente la stessa risposta. Alternativamente lo stesso procedimento può essere rappresentato in modo più compatto seguendo la lista di passi che portano dall'espressione originale al suo valore:

```
cylinder-volume 5.0 4.0
3.1415927 *. (square 5.0) *. 4.0
3.1415927 *. (5.0 *. 5.0) *. 4.0
3.1415927 *. 25.0 *. 4.0
78.5398175 *. 4.0
314.15927
```

Indipendentemente dal modo in cui rappresentiamo il processo di valutazione (diagramma piuttosto che sequenza di espressioni), diciamo che stiamo usando il [sistema di valutazione per sostituzione](#).

Il nome deriva dal modo in cui gestiamo l'applicazione di funzioni: i valori degli argomenti sono sostituiti nel corpo della funzione al posto dei nomi dei parametri, e l'espressione risultante viene valutata a sua volta.

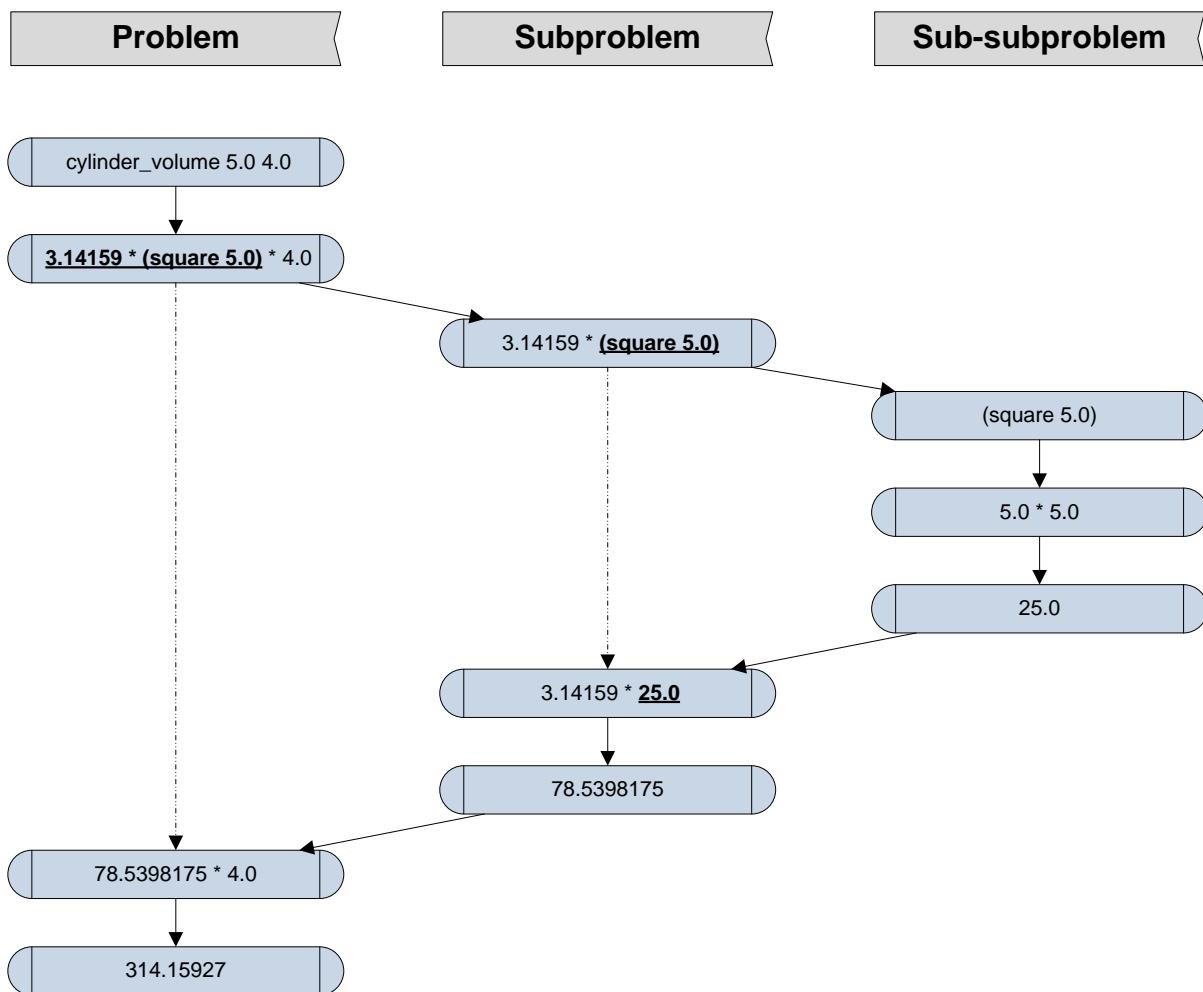


Figura 1.1

ESERCIZIO 1.4

Secondo il libro “L’Arte della Cucina”, uno sciroppo andrebbe scaldata ad un grado in meno rispetto alla temperatura indicata per ogni 150 metri di altezza sul livello del mare.

- Definire la funzione `candy_temperature` che dati due argomenti, la temperatura originale in gradi e l’altezza in metri sul livello del mare, calcola la temperatura da usare per la cottura. La ricetta per il Caramello al Cioccolato richiede una temperatura di 118 gradi; supponiamo di volerla cucinare a Enna, il capoluogo più alto d’Italia (948 metri sul livello del mare). Usare la funzione appena scritta per calcolare la temperatura da usare.
- Modificare la funzione `candy_temperature` affinché arrotondi il risultato all’intero più vicino, definendo la funzione di arrotondamento `round` (già integrata nel linguaggio F#).

Le definizioni di funzione ci danno la possibilità di effettuare la stessa computazione con valori differenti. A volte però non solo vogliamo cambiare i valori passati alla funzione, ma anche il comportamento della funzione rispetto alle circostanze. Ad esempio, consideriamo una tassa sul reddito semplice calcolata al 20% del reddito, considerando che un reddito sotto i 10000 € non verrà tassato. La funzione che calcola le tasse da pagare con un certo reddito è:

```
# let tax = fun income ->
  if income < 10000 then
    0
  else
    (income * 20) / 100;;
tax : int -> int = <fun>
```

La stessa funzione può anche essere definita usando la notazione prefissa che mette in evidenza l'operatore `<`, come:

```
# let tax = fun x ->
  if (prefix <) x 10000 then
    0
  else (x * 20) / 100;;
tax : int -> int = <fun>

# tax 12500;;
- : int = 2500

# tax 30000;;
- : int = 6000
```

Da questo esempio si possono osservare due novità. La prima è la funzione `(prefix <)` che valuta se un valore è minore di un altro. Tale funzione, avendo come nome un simbolo particolare e potendo essere usata anche in notazione infissa, viene anche chiamata *operatore*. Questa funzione non calcola un numero come tutte quelle che abbiamo visto finora, bensì calcola un valore di verità (o *booleano*): “vero” o “falso”. Nel nostro caso l’operatore calcola “vero” (`true`) se il reddito è minore di 10000 e “falso” (`false`) se il reddito è maggiore o uguale a 10000. L’altra novità è l’espressione `if`, che usa un valore di verità per decidere quale tra due espressioni valutare. (Come potreste aver già indovinato, ci sono altri operatori predefiniti che calcolano valori booleani, come `>`, `=`, `<=`, `>=`, etc. I simboli `<=` e `>=` denotano rispettivamente i simboli matematici \leq e \geq). Seguiamo i passi necessari alla valutazione dell’espressione `tax 30000`:

```
(tax 30000)
(if (30000 < 10000) then 0 else (30000 * 20 / 100))
(if false then 0 else (30000 * 20 / 100))
(30000 * 20 / 100)
6000
```

Nel passaggio dalla seconda alla terza linea, l’espressione `(30000 < 10000)` viene valutata al valore “falso”, scritto `false`. Analogamente il valore “vero” si scrive `true`. Poiché il test dell’`if` ha ritornato `false`, la seconda sottoespressione (lo `0`) viene ignorata e la terza sottoespressione `(30000 * 20 / 100)` viene valutata. Il processo di calcolo è rappresentato nella Figura 1.2.

Problem

Subproblem

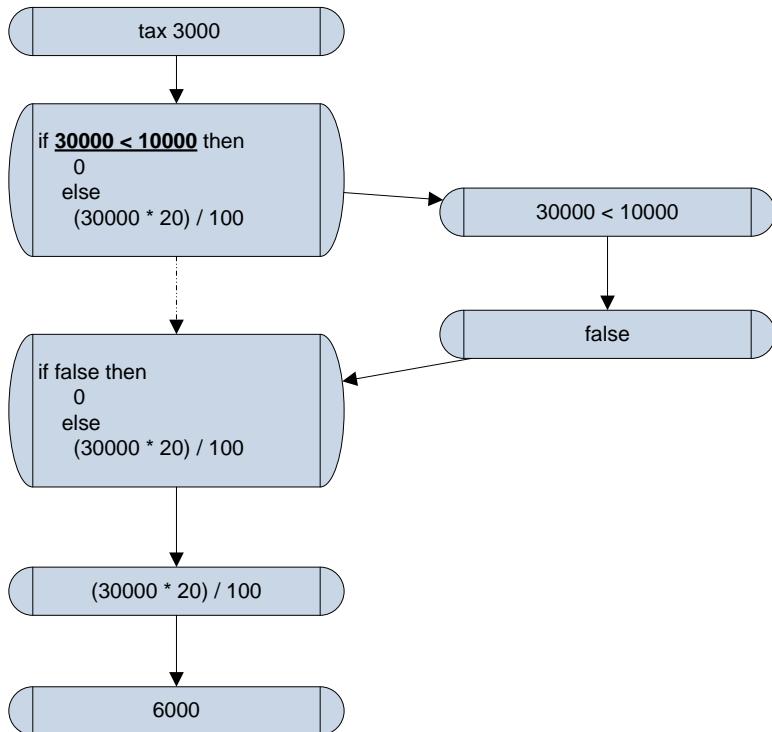


Figura 1.2

ESERCIZIO 1.5

L'esempio delle tasse visto precedentemente ha (almeno) una proprietà indesiderabile: se si guadagnano 9999€ non si pagano tasse, se si guadagnano 10000€ se ne pagano 2000 di tasse, per una perdita netta di 1999€. Questa potenziale ingiustizia viene gestita usando una cosiddetta tassa marginale, ossia ogni euro viene tassato ad un certo tasso che dipende da quale livello di reddito tale euro rappresenta. Questo significherebbe che i primi 10000€ non sono tassati, mentre tutto il reddito oltre i 10000€ è tassato al 20%. Ad esempio su un reddito di 12500€ i primi 10000€ non sono tassati, mentre i 2500€ rimanenti sono tassati al 20%, ottenendo una tassa totale di 500€.

Riscrivere la procedura di calcolo delle tasse secondo questa nuova strategia.

SOLUZIONE

```

# let tax = fun x ->
  if x < 10000 then
    0
  else
    ((x - 10000) * 20) / 100;;
tax : int -> int = <fun>

# tax 12500;;
- : int = 500

# tax 30000;;
- : int = 4000
  
```

ESERCIZIO 1.6

“L’Arte della Cucina” suggerisce un modo per calcolare per quante persone basterà un tacchino. Per tacchini fino a 5kg potranno essere fatte porzioni di 300g per persona, mentre per tacchini più grossi la dose si riduce a solo 200g. Scrivere la procedura che prende in input il peso di un tacchino e calcola il numero di persone per cui questo sarà sufficiente.

ESERCIZIO 1.7

Descrivere succintamente (in italiano!) cosa calcola ciascuna delle seguenti funzioni.

È importante esprimere *cosa* viene fatto, non *come*.

- a. let puzzle1 = fun a b c -> a + (if b > c then b else c);;
- b. let puzzle2 = fun x ->
 (if x < 0 then (prefix -) else (prefix +)) 0 x;;

ESERCIZIO 1.8

Si consideri la seguente funzione *f*. Determinare due numeri interi tali che l’applicazione di *f* a tali numeri restituisca il valore **16** come risultato.

```
let f = fun x y -> if x mod 2 = 0 then 7 else x * y
```

Nota: l’operatore *mod* ritorna il resto della divisione intera:

```
# 7 mod 3;;
- : int = 1

# 8 mod 2;;
- : int = 0
```

ESERCIZIO 1.9

Scrivere una funzione che calcola la media di due numeri interi.

SOLUZIONE

```
# let media = fun x y -> (x+y) / 2;;
media : int -> int -> int = <fun>
```

ESERCIZIO 1.10

Cosa possiamo scrivere negli spazi vuoti della definizione della seguente funzione per assicurarci che non accada mai una divisione per zero? Dare almeno due risposte differenti.

```
# let foo = fun x y -> if ____ then x + y else x / y;;
```

1.4 – Espressioni e valori

Come abbiamo visto, la nozione di espressione è centrale nella programmazione funzionale. Ci sono molti tipi di espressioni matematiche, delle quali non tutte sono permesse nella notazione che descriveremo, ma tutte con certe caratteristiche comuni. La caratteristica più importante è che un’espressione è usata per denotare un valore: il significato di un’espressione è esclusivamente un valore, e qualunque metodo per ottenere tale valore non produce altri effetti (ovvero, non viene modificato lo stato di esecuzione del programma stesso). Inoltre, il significato di un’espressione dipende soltanto dai significati delle sue sottoespressioni. Un’espressione può contenere certi identificatori (chiamati “variabili”), che rappresentano valori sconosciuti. Tali identificatori denotano

sempre la stessa quantità rispetto al contesto nel quale vengono valutati. Tale caratteristica è chiamata *integrità referenziale*.

Tra i valori che un'espressione può denotare sono inclusi i numeri, valori booleani, caratteri, ennuple, funzioni e liste. In questa dispensa vedremo tali valori, e come sia possibile costruire altri valori a partire da questi.

1.4.1 – Semantica

Abbiamo visto come un calcolatore valuti un'espressione, stampando un'espressione equivalente (che denota lo stesso valore), più semplice di quella data. Abbiamo chiamato tale processo *riduzione* (o, alternativamente, *valutazione* o *semplificazione*) di un'espressione ad un'altra. Daremo ora un breve accenno su come la riduzione viene effettuata, considerando la valutazione dell'espressione `square (3+4)`, vista precedentemente. Nel seguito il simbolo \hookrightarrow verrà utilizzato col significato di “*si riduce a*”. Una possibile sequenza di riduzioni è la seguente:

```
square (3+4)
↪ { definizione di + }
square 7
↪ { definizione di square }
7 × 7
↪ { definizione di × }
49
```

In tale sequenza, l'etichetta `+` si riferisce all'uso della regola di riduzione per l'addizione, `×` si riferisce alla regola per la moltiplicazione, e `square` si riferisce ad un uso della definizione di `square` fornita dal programmatore. L'espressione `49` è la più semplice forma equivalente all'espressione data, poiché non può essere ulteriormente ridotta.

Vale la pena notare che la sequenza di riduzioni descritta sopra non è l'unica possibile per l'espressione data. Infatti, un'altra possibile sequenza è la seguente:

```
square (3+4)
↪ { definizione di square }
(3+4) × (3+4)
↪ { definizione di + }
7 × (3+4)
↪ { definizione di + }
7 × 7
↪ { definizione di × }
49
```

In tale sequenza la regola di riduzione ottenuta dalla definizione di `square` è applicata per prima, ma il risultato è identico.

È importante puntualizzare la distinzione tra i valori e le loro rappresentazioni tramite espressioni. Ci sono molte rappresentazioni per uno stesso valore: per esempio, il numero astratto “quarantanove”

può essere rappresentato tramite il numero decimale 49, il numero romano **XLIX**, l'espressione 7×7 , ed altre molteplici rappresentazioni.

Diremo che un'espressione è *canonica* (o *in forma normale*) se non può essere ulteriormente ridotta. Un valore viene stampato nella sua rappresentazione canonica. Si noti che la nozione di espressione canonica dipende sia dalla sintassi che dalle regole di riduzione. Alcuni valori, per esempio, non hanno una rappresentazione canonica, mentre altri non hanno una rappresentazione finita: è il caso del numero π , che non ha una rappresentazione decimale finita. Alcune espressioni non possono essere ridotte, nel senso che non denotano dei valori ben definiti. Per esempio, supponendo che l'operatore “ $/$ ” denoti la divisione numerica, l'espressione $1/0$ non denota un numero ben definito. La richiesta di valutare $1/0$ causa un messaggio di errore. Nella nostra trattazione introdurremo un simbolo speciale, \perp , chiamato *bottom*, per denotare il valore indefinito. Diremo, per esempio, che il valore di $1/0$ è \perp . Ovviamente, il calcolatore non sarà sempre in grado di calcolare \perp : piuttosto, il comportamento del calcolatore su un'espressione che ha valore \perp sarà un messaggio di errore o un perpetuo silenzio.

1.4.2 – Tipi

Nella notazione che stiamo descrivendo, l'universo dei valori è partizionato in collezioni, chiamate *tipi*. I tipi possono essere distinti in due categorie. Nella prima ci sono i tipi di base, i cui valori sono dati come primitivi. Per esempio, i numeri interi costituiscono un tipo di dato di base (denotato `int` in CAML), così come i valori booleani (`bool`) e i caratteri (`char`). Nella seconda categoria ci sono i tipi derivati, i cui valori sono costruiti a partire da quelli di altri tipi. Esempi di tipi derivati includono le ennuple (ad esempio, il tipo `int * bool` rappresenta l'insieme delle coppie che hanno come prima componente un numero intero e come seconda un booleano); le funzioni (ad esempio, il tipo `int -> int` rappresenta l'insieme delle funzioni da interi a interi); infine il tipo lista (ad esempio, `char list` è il tipo delle liste di caratteri).

Ogni tipo ha un insieme di operazioni associate, che non sono significative per un altro tipo. Per esempio, non ha senso sommare un numero ad un carattere o moltiplicare tra di loro due funzioni. Come abbiamo già visto nella sezione precedente, ad ogni espressione ben formata è associato un tipo che può essere dedotto dai costituenti l'espressione stessa. In altre parole, così come il valore di un'espressione dipende esclusivamente dai valori delle sue sottoespressioni, il tipo di un'espressione dipende dai tipi delle sottoespressioni. Questo principio è chiamato *tipizzazione forte*, o *strong-typing*. La conseguenza fondamentale del principio della tipizzazione forte è che ogni espressione alla quale non può essere assegnato un tipo è considerata “mal tipata”, e non viene valutata. Tali espressioni non hanno valore, essendo considerate illegali. Ad esempio, l'espressione `1.0 * 2` è mal tipata. Il sistema riconosce l'inconsistenza e genera il seguente messaggio:

```
# 1.0 * 2;;
Toplevel input:
>1.0*2;;
>^^^
This expression has type float,
but is used with type int.
```

L'operatore `*`, infatti, è riservato esclusivamente ad operazioni su interi (il corrispettivo CAML per operazioni sui reali è `*.` (asterisco seguito da `.`)).

Ci sono due passi di analisi su un'espressione che viene sottoposta a valutazione. L'espressione viene prima controllata sintatticamente: tale passo si chiama *analisi sintattica*. Se l'espressione è conforme alla sintassi da utilizzare per definire espressioni, si cerca di assegnarle un tipo. Questo passo viene chiamato *analisi di tipo*. Se il tentativo di assegnarle un tipo fallisce, allora viene generato un errore di tipo.

Il controllo dei tipi consente di determinare se le espressioni sono usate con coerenza e di non autorizzare tentativi di calcolare espressioni senza senso, come ad esempio la sottrazione di un booleano da un intero o la somma di una stringa con un reale. Ad esempio:

```
# true + 5;;
Toplevel input:
>true + 5;;
>^^^^^
This expression has type bool,
but is used with type int.
```

1.5 – Funzioni e Definizioni

L'elemento più importante in programmazione funzionale è una funzione. Matematicamente, una funzione è una regola di corrispondenza che associa ad ogni elemento di un certo tipo A un solo elemento di un certo tipo B . Il tipo A è chiamato *dominio* della funzione, mentre il tipo B viene detto *codominio* della funzione. Tale informazione verrà espressa con la simbologia $f : A \rightarrow B$ o, nella notazione CAML, $f : A \rightarrow B$.

Se x denota un elemento di A , scriveremo $f(x)$ (o semplicemente $f\ x$) per denotare il risultato dell'applicazione della funzione f a x . Tale valore è l'unico elemento di B associato ad x dalla regola di corrispondenza per f .

Le funzioni sono dei valori e come tali possono essere manipolate in una espressione. Per esempio esse possono essere passate come argomenti ad altre funzioni e/o restituite come risultato. Ad esempio, potremmo scrivere:

```
# let g = fun f x -> f(x+2)-1;;
g : (int -> int) -> int -> int = <fun>
```

La funzione g così definita prende come parametro una funzione (denotata con f) e restituisce una funzione che, dato un argomento x , restituisce il predecessore del risultato dell'applicazione $f(x+2)$. Esempio:

```
#g abs 7;;
- : int = 8

#g abs (-7);;
- : int = 4
```

La manipolazione di espressioni permette di definire delle funzioni tramite l'applicazione "parziale" di argomenti ad una funzione. Ad esempio, si consideri la funzione

```
let f = fun x y -> x + y  
f : int -> int -> int = <fun>
```

Tale funzione prende due numeri in ingresso e restituisce la somma dei due numeri. Ora, l'espressione `f 3` denota a sua volta una funzione: la funzione che prende in ingresso un numero e restituisce la somma di quel numero e 3:

```
# f 3;;  
- : int -> int = <fun>
```

È importante distinguere tra una funzione (intesa come valore) ed una particolare definizione per essa. Ci possono essere molte possibili definizioni per una stessa funzione: ad esempio, la funzione che raddoppia il suo argomento può essere definita nei seguenti modi:

```
# let double = fun x -> x + x;;  
double : int -> int = <fun>  
  
# let double1 = fun x -> 2 * x;;  
double1: int -> int = <fun>
```

Le due definizioni descrivono differenti procedure per raddoppiare un numero, ma `double` e `double1` rappresentano la stessa funzione. Se comunque guardiamo ad esse come procedure per la valutazione, una definizione può essere più o meno "efficiente" dell'altra, nel senso che il valutatore è in grado di ridurre espressioni della forma `double x` più o meno rapidamente di espressioni della forma `double1 x`. Tuttavia, la nozione di efficienza non è direttamente legata ad una funzione, quanto invece alla sua definizione ed ai meccanismi per la sua valutazione.

1.5.1 – Informazioni di Tipo

Come abbiamo visto negli esempi precedenti, ad ogni definizione di funzione è automaticamente associato un tipo, derivato direttamente dall'equazione che la definisce. Ciò è conseguenza diretta della tipizzazione forte, che abbiamo definito in precedenza. Così, per esempio, l'operatore `*` che abbiamo utilizzato nella definizione di `square` è riservato esclusivamente ad operazioni su interi, e di conseguenza, il tipo associato a `square` è `int -> int`. Alternativamente, avremmo potuto definire `square` in termini dell'operatore `*.`, definito sui reali:

```
# let square' = fun x -> x *. x;;  
square' : float -> float = <fun>
```

Anche in questo caso i parametri della funzione non hanno bisogno di una esplicita dichiarazione del tipo, poiché il sistema è in grado di inferire tale tipo automaticamente. Tuttavia, alcune funzioni hanno un dominio e un codominio generali. Si consideri ad esempio la definizione $id(x) = x$, per la funzione identità. Tale funzione mappa ogni elemento del dominio in sé stesso. Il suo tipo è, quindi,

$A \mapsto A$, per qualsiasi tipo A . Il risultato della valutazione di `id` nel sistema, richiederà quindi l'introduzione di *variabili di tipo*:

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

Qui `'a` denota una variabile di tipo, e sta ad indicare che tale variabile può essere instanziata in maniera diversa in circostanze diverse. Per esempio:

```
# id 3;;
- : int = 3
```

indica che l'espressione `id 3` è ben formata ed ha tipo `int`, poiché `int` può essere sostituito ad `'a` nel tipo di `id`, producendo una funzione di tipo `int -> int`. Allo stesso modo anche le seguenti espressioni sono ben formate ed hanno associato un tipo:

```
# id square;;
- : int -> int = <fun>

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

Un altro esempio di funzione il cui tipo contiene variabili è il seguente:

```
# let foo = fun x -> 'a';;
foo : 'a -> char = <fun>
```

In questo caso il dominio della funzione `foo` può essere uno qualsiasi, mentre il codominio è fissato.

Abbiamo quindi definito un linguaggio di espressioni che denotano tipi. Tale linguaggio contiene costanti, come `int` e `char`, variabili di tipo, come `'a` e `'b`, ed operatori, come `*` e `->`. Se un'espressione costruita su tale linguaggio contiene variabili, allora diciamo che essa denota un tipo *polimorfo*. Negli esempi precedenti, le funzioni `id` e `foo` hanno tipo polimorfo.

A partire dal linguaggio di espressioni che denotano tipi, è possibile associare i tipi direttamente in una definizione. Ad esempio, consideriamo la seguente funzione `idtuple`:

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

Se vogliamo definire una istanza della funzione `id` sulle coppie di elementi dello stesso tipo, possiamo fare uso della seguente definizione:

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

o, alternativamente, definire il tipo degli argomenti:

```
# let idtuple = fun (x : 'a * 'a) -> x;;
idtuple : 'a * 'a -> 'a * 'a = <fun>
```

Abbiamo visto come sia possibile definire una funzione per casi. La funzione `min` che calcola il minimo tra due elementi di un insieme `'a` su cui è definito un ordine, può essere definita come segue:

```
# let min = (fun x y -> if x <= y then x else y);;
min : 'a -> 'a -> 'a = <fun>
```

La funzione sopra contiene due espressioni, ognuna delle quali è scelta per mezzo di un'espressione booleana (`x <= y`) chiamata *guardia*. In base alla valutazione dell'espressione booleana si valuta la prima o la seconda espressione. In seguito vedremo come generalizzare questo meccanismo di definizione per casi, tramite i *patterns*. Si noti che la funzione `min` è definita per qualsiasi coppia di valori su cui è possibile stabilire un ordine; dato che le funzioni non possono essere confrontate, la funzione `min` non può essere applicata ad una coppia di funzioni. Infatti:

```
#min (prefix -) (prefix +);;
Uncaught exception: Invalid_argument "equal: functional value"
```

1.5.2 – Definizioni

Un nuovo nome viene introdotto nell'ambiente legandolo ad un valore (*value binding*) per mezzo della sintassi

```
let nome = espressione
```

CAML valuta prima la parte destra dell'equazione e poi ne lega il valore al nome specificato nella parte sinistra. Esempio:

```
# let n = 2 + 3;;
n : int = 5
```

Se la dichiarazione di una variabile o di una funzione contiene il riferimento ad una variabile `x`, ogni volta che tale variabile o funzione viene valutata è utilizzato il valore che `x` aveva nel momento in cui è stata introdotta la sua definizione. Ad esempio, nel codice seguente la seconda dichiarazione di `x` crea un nuovo binding che maschera il primo, ma non intacca il valore di `y`.

```
# let x = 17;;
x : int = 17

# let y = x;;
y : int = 17

# let x = true;;
x : bool = true

#y;;
- : int = 17
```

Il valore di una variabile o di una funzione è definito dal `let` più recente.

Un'altra notazione che andiamo ad introdurre è il meccanismo delle definizioni *locali*. Abbiamo visto che il costrutto `let` permette di definire delle funzioni. In realtà, il costrutto può essere utilizzato in una forma più generale, per permettere l'uso di definizioni “locali” all'espressione che definiamo. Per esempio,

```
# let piu3 = fun x -> let tre = 3 in x + tre;;
piu3 : int -> int = <fun>
```

La funzione `piu3` somma la costante denotata dall'identificatore `tre` al parametro di input. Si noti che il legame tra l'identificatore `tre` e l'espressione `3` avviene nel contesto di valutazione di `piu3`.

```
# tre;;
Toplevel input:
>tre
>^^^
The value identifier tre is unbound.
```

Le definizioni locali possono anche riguardare funzioni:

```
# let foo = (fun x -> let f = (fun y -> x * y) in f x) ;;
foo : int -> int = <fun>
# foo 3;;
- : int = 9;;
# f 3;;
> ^
The value identifier f is unbound.
```

Nella definizione precedente la funzione `f` è locale alla definizione di `foo`, ed è invisibile nel contesto generale. Il meccanismo dei contesti annidati è anche chiamato *scope*, per indicare la visibilità di una funzione in un contesto.

Le dichiarazioni locali sono utili quando un identificatore ha un uso limitato e non ha significato al di fuori dell'espressione più ampia che lo utilizza. La definizione di funzioni utilizza spesso dichiarazioni locali, allo scopo di richiamare funzioni ausiliarie che non hanno significato autonomo.

ESERCIZIO 1.11

Si definisca la funzione `segno`, di tipo `int -> int`, che restituisce 1 se l'argomento è positivo, -1 se l'argomento è negativo e 0 se l'argomento è 0.

SOLUZIONE

```
# let segno = fun x -> if x = 0 then 0 else (abs x)/x;;
segno : int -> int = <fun>
# segno 78;;
- : int = 1
# segno (-96);;
- : int = -1
```

ESERCIZIO 1.12

Si diano esempi di funzioni con i seguenti tipi:

```
(int -> int) -> int
int -> (int -> int)
(int -> int) -> (int -> int)
```

SOLUZIONE

```
# let foo = fun f -> f(1) +1;;
foo : (int -> int) -> int = <fun>

# let foo = fun x -> let f = (fun x y -> x +y ) in f x;;
foo : int -> (int -> int) = <fun>

# let foo = fun f -> let g = (fun x y -> x + y) in g(f(1));;
foo : (int -> int) -> (int -> int) = <fun>
```

ESERCIZIO 1.13

Si inferiscano i tipi delle seguenti funzioni:

```
# let one = (fun x -> 1);;
# let apply = (fun f x -> f x);;
```

1.6 – Specifica e Implementazione

In programmazione, per specifica si intende una descrizione matematica dei compiti che un programma è chiamato a svolgere, mentre per implementazione si intende la realizzazione di un programma che soddisfi la specifica data. C'è una sostanziale differenza tra specifica ed implementazione. La specifica è l'espressione degli intenti del programmatore (o delle aspettative del cliente). Una specifica dovrebbe essere concisa e chiara il più possibile. L'implementazione, per contro, è una serie di istruzioni eseguibili da un calcolatore, e il suo proposito è quello di essere abbastanza efficiente da poter essere eseguita. Il legame tra le due è la necessità che l'implementazione soddisfi la sua specifica, ed il programmatore serio è obbligato a fornire una *dimostrazione* di una corretta implementazione (nei confronti della specifica).

La specifica di una funzione è la descrizione della relazione voluta tra gli argomenti ed i risultati. Un semplice esempio è dato dalla seguente specifica della funzione *increase*:

$$\forall x \geq 0. (\text{increase } x > \text{square } x)$$

In base a ciò, il risultato di *increase* dovrebbe essere più grande del quadrato del suo argomento. Una possibile implementazione di *increase* è data dalla seguente definizione:

```
# let increase = fun x -> square (x + 1);;
increase : int -> int = <fun>
```

La dimostrazione che questa definizione di *increase* soddisfa la specifica è come segue: assumendo $x \geq 0$, otteniamo:

```

increase x
= { definizione di increase }
square (x + 1)
= { definizione di square }
(x + 1) × (x + 1)
= { proprietà algebriche }
x × x + 2 × x + 1
> { assunzione: x ≥ 0 }
x × x
= { definizione di square }
square x

```

In questo esempio, abbiamo prima inventato una definizione per *increase*, quindi abbiamo verificato che tale definizione soddisfa la specifica. Chiaramente ci sono molte altre definizioni che soddisfano la specifica.

Il problema di mostrare che una definizione formale soddisfa la sua specifica può essere affrontato in molti modi distinti. Un approccio è quello appena visto, che consiste nel disegnare la definizione e verificare, in seguito, la sua adeguatezza. Un altro approccio, che può portare a programmi più chiari e più semplici, è quello di sviluppare sistematicamente (o *sintetizzare*) le definizioni dalla specifica. Per esempio, se guardiamo alla specifica di *increase* noi possiamo argomentare che, poiché $x + 1 > x$ per tutti gli x , abbiamo:

$$\forall x. ((\text{square } x) + 1 > \text{square } x)$$

D'altra parte,

$$\forall x. ((\text{square } x) + 1 > \text{square } x \wedge \text{increase } x = (\text{square } x) + 1 \Rightarrow \text{increase } x > \text{square } x)$$

da cui otteniamo una nuova definizione che soddisfa una specifica più forte di quella data sopra, essendo la disuguaglianza valida anche per i numeri negativi.

```

# let increase = fun x -> (square x) + 1;;
increase : int -> int = <fun>

```

Le potenziali sorgenti di difficoltà di tale metodo derivano dalla possibilità che la specifica formale non rispecchi le effettive intenzioni (informali), e che la dimostrazione che l'implementazione soddisfa la specifica possa essere talmente grande o complicata da non poter essere assicurata immune da errori. Tuttavia, l'importanza di tecniche adeguate per la verifica e la correttezza dei programmi hanno reso il paradigma dello sviluppo sistematico un ramo attivo di ricerca nell'informatica.

ESERCIZIO 1.14

Si descriva la specifica e la relativa implementazione della funzione *intsqrt* che, dato un numero, restituisce l'intero più grande minore o uguale a \sqrt{x} .

ESERCIZIO 1.15

Usando una qualsiasi notazione, si esprima la specifica della funzione *isSquare* che stabilisce se un numero intero è un quadrato perfetto. La seguente definizione soddisfa la specifica ?

```
# let isSquare = fun x -> (square (intsqrt x) = x);;
```

Capitolo 2 – Tipi Elementari

In questo capitolo sono descritti i tipi di dato a partire dai quali le espressioni possono essere costruite. Come abbiamo visto, i tipi di dato di base sono gli interi, i decimali, i booleani, i caratteri e le tuple. Descriveremo come i valori di ogni tipo sono rappresentati e presenteremo alcune operazioni primitive per manipolare tali valori.

2.1 – Numeri

Nel precedente capitolo abbiamo avuto a che fare con due tipi principali di numeri: i numeri interi (denotati con `int`) ed i numeri decimali (denotati con `float`). Le costanti numeriche, come abbiamo visto, sono rappresentate con la notazione decimale. Ovviamente, essendo l’insieme delle risorse di un calcolatore finito, non tutti i numeri possono essere rappresentati su un calcolatore, ed in particolare i numeri reali non possono essere rappresentati con precisione (occorre cioè ricorrere ad una approssimazione finita del numero). (Il sistema CAML utilizza 31 bits per la rappresentazione dei numeri interi, e 64 bits per la rappresentazione dei numeri decimali).

Useremo le seguenti operazioni definite su `int`.

<code>+</code>	addizione intera
<code>-</code>	sottrazione intera
<code>*</code>	moltiplicazione intera
<code>/</code>	divisione intera
<code>mod</code>	modulo intero

Ognuno di questi operatori è usato come operatore *binario infisso* (ad esempio, per il simbolo “+”, scriviamo `x + y`). Il segno “–” comunque, può essere usato anche come operatore *unario prefisso* (per indicare la negazione).

Gli operatori sui decimali sono:

<code>+. .</code>	addizione
<code>-. .</code>	sottrazione
<code>* .</code>	moltiplicazione
<code>/ .</code>	divisione
<code>**</code>	elevamento a potenza
<code>sqrt ()</code>	radice quadrata

Da decimale a intero e viceversa:

```
# int_of_float 3.9;;
- : int = 3

# float_of_int 3;;
- : float = 3.0
```

```
- : float = 3.0
```

SCHEDA: DA INTERO A REALE E VICEVERSA

In F# le funzioni `int_of_float` e `float_of_int` sono denominate, rispettivamente, `int` e `float`.

Poiché la rappresentazione di un numero può non essere esatta, operazioni sui numeri decimali possono non produrre i risultati aspettati dall'aritmetica ordinaria (ad esempio, $(x * . y) /. y$ può essere diverso da x).

Vediamo una sessione in cui il terminale viene utilizzato per calcoli aritmetici:

```
# 2+3*4;;
- : int = 14

# 5-4-2;;
- : int = -1

# 2.0*.1.4-.12.5;;
- : float = -9.7

# 3.0**12.0/.3.0**10.0;;
- : float = 9.0

# 2.0**2.0**3.0;;
- : float = 256.0
```

Come possiamo vedere dagli esempi sopra, quando appare più di un operatore all'interno di una stessa espressione sono necessarie delle regole per stabilire la precedenza tra i vari operatori. Così, ad esempio, l'operatore esponenziale “`**`” ha precedenza sulla moltiplicazione, la divisione ed il modulo (da cui possiamo dedurre che l'espressione `3.0**10.0*.3.0` significa `(3.0**10.0)*.3.0`) e, analogamente, gli operatori di moltiplicazione e divisione hanno precedenza sugli operatori di addizione/sottrazione.

Un'altra proprietà aritmetica degli operatori riguarda l'ordine di associazione. Nel caso della definizione di una funzione, abbiamo già visto che gli argomenti della funzione vengono associati verso sinistra. Analogamente, supporremo che gli operatori descritti sopra associno verso sinistra, con l'eccezione dell'operatore esponenziale che associa verso destra (come l'ultimo esempio mostra).

Vale la pena notare, inoltre, che alcuni operatori come l'addizione e la moltiplicazione godono della proprietà associativa, in base alla quale associando a destra o a sinistra si calcola comunque lo stesso valore.

Un altro esempio di associazione a destra è dato dall'operatore “`->`” che opera sui tipi. Consideriamo la seguente definizione:

```
# let foo x y = x * y;;
foo : int -> int -> int = <fun>
```

Come si può vedere dalla risposta del sistema, la funzione ha tipo `int -> (int -> int)`. In CAML, molti operatori binari predefiniti (come “+” e “*”) hanno tipo `A -> (B -> C)` per qualche `A`, `B` e `C`. Avremmo potuto definire la funzione `foo` in maniera “simile”, ma sostanzialmente differente dal punto di vista semantico:

```
# let foo2 (x,y) = x * y;;
foo2 : int * int -> int = <fun>
```

Mentre il valore denotato dall’applicazione delle due funzioni ad una coppia di numeri è identico, la differenza è rimarcata nel meccanismo di valutazione.

Si consideri la funzione `checksum(x, y, z)` definita come segue:

```
# let checksum (x,y,z) = x + y = z;;
checksum : int * int * int -> bool = <fun>

# checksum (3,5,8);;
- : bool = true

# checksum (3,5,9);;
- : bool = false
```

La funzione `checksum` può essere definita, in modo equivalente, come segue:

```
# let checksum1 x y z = x + y = z;;
checksum1 : int -> int -> int -> bool = <fun>
```

L’associazione è a sinistra, cioè `(checksum1 x y z)` equivale a `((checksum1 x) y) z`.

```
# let checksum2 = checksum1 3;;
checksum2 : int -> int -> bool = <fun>
```

Intuitivamente, `checksum2 y z = (3 + y = z)`

```
# let checksum3 = checksum2 5;;
checksum3 : int -> bool = <fun>
```

Intuitivamente, `checksum3 z = (3 + 5 = z)`

```
# checksum3 8;;
- : bool = true

# checksum3 9;;
- : bool = false
```

Il meccanismo di utilizzare sequenze di argomenti invece che argomenti “strutturati” è chiamato *currying*, dal matematico americano H. B. Curry. Così, la funzione `foo` è detta la versione curry della

funzione `foo2`. È possibile passare da una versione non curry ad una versione curry in maniera molto semplice:

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

# curry foo;;
- : int -> int -> int = <fun>
```

e ovviamente, è possibile anche passare da una versione curry ad una non curry:

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

# uncurry foo;;
- : int * int -> int = <fun>
```

Possiamo utilizzare la versione curry della funzione somma per realizzare la funzione di incremento:

```
# let somma x y = x + y;;
somma : int -> int -> int = <fun>

# let incr = curry somma 1;;
incr : int -> int = <fun>

# incr 5;;
- : int = 6
```

Il tipo di una funzione curryficata deve essere letto associando a destra: `int -> int -> int` equivale a `int -> (int -> int)`, che corrisponde ad una funzione che prende in input un intero e restituisce una funzione da interi a interi.

Una differenza tra gli operatori predefiniti che abbiamo visto e le funzioni definite nel sistema è che tali operatori sono *operatori infissi*; sono degli operatori cioè rappresentati tra gli argomenti. Abbiamo già visto che è comunque possibile considerare tali operatori come operatori prefissi, utilizzando la funzione `prefix` (In F# e OCaml, un operatore infisso racchiuso tra parentesi tonde, esempio `(+)` è considerato come prefisso). Così, ad esempio, otteniamo:

```
# prefix +;;
- : int -> int -> int = <fun>
```

L'utilizzo di operatori normalmente infissi come operatori prefissi è di notevole utilità, ad esempio quando si definiscono delle funzioni *higher order* (delle funzioni, cioè, che prendono come parametro altre funzioni): consideriamo ad esempio la definizione di `double`:

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

A questo punto possiamo valutare le seguenti espressioni:

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

# double (prefix +) 4;;
- : int = 8
```

Ovviamente è possibile utilizzare tale notazione per definire anche altre funzioni: ad esempio, se vogliamo definire l'inverso di un numero decimale,

```
# let inv = (prefix /.) 1.0;;
inv : float -> float = <fun>

# inv 4.0;;
- : float = 0.25
```

ESERCIZIO 2.1

Per quali argomenti le seguenti funzioni ritornano true ?

- a. (prefix =) 9 (prefix +) 2 (prefix *) 7
- b. (prefix >) 3 (prefix mod) 8

2.2 – Booleani

Abbiamo già visto alcune espressioni che fanno uso di valori di verità. Tali funzioni permettono di confrontare espressioni, e restituiscono un valore detto booleano. I valori di tipo booleano sono due: `true` e `false`. Una funzione che restituisce tali valori è chiamata *predicato*. Alcuni esempi degli operatori di confronto sono i seguenti:

```
# 2 = 3;;
- : bool = false

# 2 < 1 + 3;;
- : bool = true
```

Alcuni tra gli operatori di confronto tipici sono:

=	uguaglianza
<>	disuguaglianza
< (<code><=</code>)	minore (minore o uguale)
> (<code>>=</code>)	maggiore (maggiori o uguale)

Gli operatori di confronto non sono confinati soltanto ai numeri, ma possono avere come argomento anche espressioni di tipo arbitrario (ma non funzioni). La sola restrizione è che i due argomenti abbiano lo stesso tipo. Un operatore di confronto è quindi del tipo `'a -> 'a -> bool`.

Gli operatori logici, invece, sono definiti su valori booleani. Essi sono:

not	negazione
&	congiunzione

or disgiunzione

Vediamo alcuni esempi.

```
# (prefix &);;
- : bool -> bool -> bool = <fun>

# (prefix or);;
- : bool -> bool -> bool = <fun>

# 1 < 2 or 0 > 1;;
- : bool = true

# not 2 < 1 & 2 < 1;;
- : bool = false

# not (2 < 1 & 2 < 1);;
- : bool = true

# true or false & true;;
- : bool = true

# not true or false;;
- : bool = false
```

Dagli esempi descritti si può capire la precedenza tra questi operatori.

Supponiamo di voler definire un predicato che indica se un anno è bisestile. Nel calendario gregoriano, un anno bisestile è un anno divisibile per 4, con l'eccezione di quelli divisibili per 100, che per essere bisestili devono essere divisibili per 400. In CAML, la definizione della funzione voluta è abbastanza semplice, utilizzando gli operatori logici e di confronto.

```
# let bisestile y = (y mod 4 = 0) &
                     (y mod 100 <> 0 or y mod 400 = 0);;
bisestile : int -> bool = <fun>
```

ESERCIZIO 2.2

- Si definisca la funzione `sumsqrs` che prende come argomenti tre interi e restituisce la somma dei quadrati dei due più grandi.
- Si definisca la funzione `prime` che prende come argomenti due numeri interi e stabilisce se i due interi sono primi tra loro.

2.3 – Caratteri e Stringhe

I simboli che normalmente appaiono sullo schermo di un terminale sono chiamati caratteri. I caratteri sono rappresentati su un calcolatore tramite una sequenza di bits, seguendo una certa codifica. La codifica più usata è la codifica ASCII, che permette di rappresentare 128 caratteri comprendenti i simboli visivi e i caratteri di controllo. Questi caratteri costituiscono il tipo di dato `char` e sono considerati dati primitivi. In CAML (e in molti altri sistemi), un carattere viene rappresentato racchiuso tra virgolette:

```
#'a';;
- : char = 'a'

#'5';;
- : char = '5'

#\n;;
- : char = '\n'
```

Si noti che il carattere '5' è sostanzialmente differente dal numero 5: il primo è un elemento di tipo `char`, mentre il secondo è di tipo `int` e denota un numero decimale. Come nel caso dei decimali, queste espressioni non possono essere ulteriormente valutate. Il simbolo `\n` rappresenta invece il carattere `newline`, ed è un tipico carattere di controllo.

Le seguenti due definizioni primitive sono associate al tipo `char`, entrambe relative alla codifica ASCII dei caratteri.

```
# int_of_char;;
- : char -> int = <fun>

# char_of_int;;
- : int -> char = <fun>
```

SCHEDA: DA CARATTERE A INTERO E VICEVERSA

In F# le funzioni `int_of_char` e `char_of_int` sono denominate semplicemente `code` e `chr`, rispettivamente.

La funzione `int_of_char` restituisce il valore ASCII di un carattere, mentre la funzione `char_of_int` restituisce il carattere relativo al codice passato come argomento.

```
# int_of_char 'b';;
- : int = 98

# char_of_int 98;;
- : char = 'b'

# char_of_int (int_of_char 'b' + 1);;
- : char = 'c'

# int_of_char '\n';;
- : int = 10
```

I caratteri possono essere confrontati tra di loro (abbiamo infatti accennato al fatto che gli operatori di confronto sono polimorfi); l'ordinamento tra i caratteri è quello che segue dalla corrispondente codifica ASCII:

```
# 'a' < 'z';;
- : bool = true
```

```
# ` ' < 'a';;
- : bool = true

# 'A' < 'a';;
- : bool = true
```

Sfruttando le informazioni fornite dagli esempi, possiamo provare a definire alcune funzioni di utilità generale per i caratteri. Ad esempio, possiamo definire un predicato che ci dica se un carattere rappresenta un numero, una lettera maiuscola o una minuscola:

```
# let isdigit x = '0' <= x & x <= '9';;
isdigit : char -> bool = <fun>

# let isupper x = 'A' <= x & x <= 'Z';;
isupper : char -> bool = <fun>

# let islower x = 'a' <= x & x <= 'z';;
islower : char -> bool = <fun>
```

Possiamo inoltre definire altre funzioni per rendere maiuscolo/minuscolo un carattere:

```
# let uppercase x = if islower x
    then
        let offset = int_of_char 'a' - int_of_char 'A'
        in char_of_int (int_of_char x + offset)
    else x;;
uppercase : char -> char = <fun>

# let lowercase x = if isupper x
    then
        let offset = int_of_char 'a' - int_of_char 'A'
        in char_of_int (int_of_char x - offset)
    else x;;
lowercase : char -> char = <fun>
```

Queste funzioni sfruttano il fatto che i caratteri in maiuscolo vengono immediatamente prima dei caratteri in minuscolo.

Una sequenza di caratteri è detta *stringa*. Le stringhe sono denotate utilizzando le doppie virgolette: la differenza tra 'a' e "a" risiede nel fatto che la prima è un carattere, mentre la seconda è una lista (sequenza ordinata per posizione) di caratteri che, in questo caso, contiene un solo elemento. Vediamo alcuni esempi di stringhe:

```
# "hello world";;
- : string = "hello world"

# "hello" < "hallo";;
- : bool = false
```

L'operatore ^ permette di concatenare stringhe.

```
# "piano" ^ "forte";;
- : string = "pianoforte"
```

2.4 – Ennuple

Un modo per combinare tipi in modo da formarne di nuovi è quello di accoppiarli. Per esempio, il tipo `int * char` rappresenta le coppie composte da un intero come prima componente ed un carattere come seconda componente. Poiché le coppie sono valori, possiamo valutarle fino ad ottenere delle forme canoniche:

```
# (3+5, 'a');;
- : int * char = 8, 'a'

# (2,3) = (1,2);;
- : bool = false

# (2,(3,'a'));;
- : int * (int * char) = 2, (3,'a')
```

L'ordinamento su coppie (x,y) e (z,w) è dato dalla regola " $(x,y) < (z,w)$ se e solo se $x < z$ oppure $x = z$ e $y < w$ ". Questo ordinamento è chiamato *lessicografico*.

Così come abbiamo formato coppie, possiamo formare triple, quadruple, ecc. Per esempio, il tipo `int * int * char` rappresenta triple. Si osservi che `int * (int * char)` è diverso da `int * int * char`, poiché il primo tipo rappresenta coppie di cui il secondo elemento è una coppia mentre il secondo rappresenta terne.

Definiamo due funzioni particolarmente importanti per le coppie: le funzioni di selezione.

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

# fst (1,2);;
- : int = 1

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

# snd (1,2);;
- : int = 2
```

Come si può notare, `fst` e `snd` sono funzioni polimorfe, che selezionano il primo o il secondo elemento di una coppia e lo restituiscono come risultato. Ovviamente, tali funzioni possono essere estese al caso di triple, quadruple, ecc.

2.4 .1 – Un esempio: Aritmetica Razionale

I numeri *razionali* sono generalmente rappresentati per mezzo di coppie (x,y) , o, in notazione matematica, x/y , di interi, tali che $y \neq 0$. Una frazione (x,y) è detta ridotta ai minimi termini se x e

x e y sono primi tra loro (ovvero se il massimo comun divisore tra i due è 1). Una frazione negativa è rappresentata da una coppia (x,y) tale che x è negativo. Il numero 0 è rappresentato per convenzione come $(0,1)$.

Diciamo che una frazione è in forma canonica se è ridotta ai minimi termini; due frazioni sono uguali se hanno la stessa forma canonica. Definiamo quindi l'algebra delle frazioni, definendo somma, differenza, moltiplicazione e divisione di frazioni, assicurando che i risultati siano in forma canonica. Definiamo prima la funzione `sign` che determina il segno di un numero intero.

```
# let sign x = if x = 0 then 0 else x/abs(x);;
sign : int -> int = <fun>
```

Abbiamo bisogno ora di definire la funzione `gcd` che dati due numeri naturali m e n maggiori di 0, determina il massimo comun divisore (`gcd`) di m e n .

Utilizziamo il seguente ALGORITMO DI EUCLIDE:

$$\text{gcd}(m, n) = \begin{cases} m & \text{se } n = m \\ \text{gcd}(m - n, n) & \text{se } m > n \\ \text{gcd}(m, n - m) & \text{se } n > m \end{cases}$$

```
# let rec gcd(m,n) = if n=m then m
                     else if m>n then gcd(m-n,n)
                     else gcd(m,n-m);;
gcd : int * int -> int = <fun>

# let norm (x,y) =
    let u = (sign y) * x
    and v = abs y
    in let d = gcd (abs u, v)
       in (u/d,v/d);;
norm : int * int -> int * int = <fun>

# let radd (x,y) (u,v) = norm (x*v + u*y, y*v);;
radd : int * int -> int * int -> int * int = <fun>

# let rsub (x,y) (u,v) = norm (x*v - u*y, y*v);;
rsub : int * int -> int * int -> int * int = <fun>

# let rmul (x,y) (u,v) = norm (x*u, y*v);;
rmul : int * int -> int * int -> int * int = <fun>

# let rdiv (x,y) (u,v) = norm (x*v, y*u);;
rdiv : int * int -> int * int -> int * int = <fun>
```

La funzione `norm` permette di definire la forma canonica di un numero razionale. La funzione `sign` restituisce -1 , 0 o 1 a seconda che x sia negativo, uguale a 0 o positivo. Infine la funzione `gcd` permette di calcolare il massimo comun divisore di un numero. Si noti che le funzioni `norm` e `div` non sono ben definite, poiché possono prendere valori non ammissibili. Se definiamo

```
# let compare op (x,y) (u,v) = op (x*v) (y*u);;
rsub : (int -> int -> 'a) -> int * int -> int * int -> 'a = <fun>
```

allora possiamo definire gli operatori di confronto, come ad esempio:

```
# let requals = compare (prefix =);;
requals : int * int -> int * int -> bool = <fun>

# let rless = compare (prefix <);;
rless : int * int -> int * int -> bool = <fun>
```

ESERCIZIO 2.3

Si supponga che una data sia rappresentata per mezzo di una tripla (g,m,a) di interi. Si definisca la funzione `age` che, presi come argomenti due date (la data di nascita di un individuo e la data attuale), restituisce l'età dell'individuo in anni.

ESERCIZIO 2.4

Si definisca la funzione `split` $x = (y,z)$, dove $\text{abs}(y) \leq 5$ e z è l'intero di valore assoluto più piccolo che soddisfa $x = y + 10 \times z$.

2.5 – Patterns

Abbiamo già visto come sia possibile definire una funzione per mezzo di più espressioni. Nel costrutto `if ... then ...` che abbiamo utilizzato, si valuta una condizione booleana `e`, in base al risultato della valutazione, viene effettuata la scelta tra una coppia di espressioni da valutare. È possibile anche definire una funzione per casi, facendo delle ipotesi sulla forma degli argomenti (ossia utilizzando dei patterns). I *patterns* sono degli schemi o modelli che permettono di descrivere la struttura di un dato di un certo tipo. Consideriamo ad esempio la funzione `cond`, che prende come argomenti un valore booleano e altri due valori, e restituisce il primo valore se il valore booleano è verificato, il secondo altrimenti. La funzione può essere definita per mezzo di patterns, enumerando i possibili valori che l'argomento di tipo booleano può assumere:

```
# let cond p x y = match p with
    true -> x
    | false -> y;;
cond : bool -> 'a -> 'a -> 'a = <fun>
```

La definizione di `cond` esprime la seguente nozione: una funzione può prendere in ingresso un certo “range” di valori, e restituire risultati diversi a seconda del valore che assume. Se, per esempio, il valore d'ingresso è `true`, allora il risultato sarà una data espressione; se, altrimenti, è `false`, il risultato sarà un'altra espressione.

Si noti la differenza tra questi modi di definire la funzione `cond` e l'utilizzo del costrutto `if ... then ... else ...`:

```
# let cond p x y = if p then x else y;;
cond : bool -> 'a -> 'a -> 'a = <fun>
```

Nella prima definizione, i valori che l'identificatore `p` può assumere sono enumerati esplicitamente, ed in maniera esaustiva (infatti il tipo `bool` comprende solo i valori `true` e `false`). La differenza è ancora più rimarcata nel seguente esempio, che definisce l'implicazione logica:

```
# let imply x y = match x,y with
    (true,true) -> true
    | (true,false) -> false
    | (false,true) -> true
    | (false,false) -> true;;
imply : bool -> bool -> bool = <fun>
```

Qui, il valore d'ingresso che la funzione può assumere è una coppia di booleani, enumerata a seconda della sua possibile forma. È possibile semplificare questa espressione, facendo uso di variabili nella definizione dei patterns. Si noti infatti, che la funzione assume valore `false` solo in un caso: quello in cui l'argomento è una coppia `(true, false)`. Possiamo allora provare a semplificare tali possibili patterns e raggruppare tutti gli altri:

```
# let imply x y = match x,y with
    (true,false) -> false
    | _ -> true;;
imply : bool -> bool -> bool = <fun>
```

Il simbolo `_` sta ad indicare che il pattern può assumere qualsiasi altro valore diverso del primo specificato.

I pattern possono essere specificati per qualsiasi tipo: la seguente funzione, ad esempio, specifica patterns sugli interi:

```
# let is_zero x = match x with 0 -> true | _ -> false;;
is_zero : int -> bool = <fun>
```

È possibile anche definire dei pattern che non comprendano una enumerazione esaustiva, come nel seguente esempio:

```
# let permute x = match x with
    0 -> 1
    | 1 -> 2
    | 2 -> 0;;
Toplevel input:>..... match x with 0 -> 1 | 1 -> 2 | 2 -> 0..
Warning: this matching is not exhaustive.
permute : int -> int = <fun>
```

In tal caso la funzione sarà definita solo sui valori enumerati:

```
# permute 3;;
Uncaught exception: Match_failure ("", 16, 62)

# permute 2;;
- : int = 0
```

I pattern possono avere una forma molto generale in CAML; per esempio è possibile definire dei patterns contenenti variabili, o accompagnati da condizioni che permettono la selezionabilità del pattern:

```
# let pred x = match x with
    0 -> 0
    | x when x > 0 -> x-1;;
Toplevel input:
>.....match x with
> 0 -> 0
>| x when x > 0 -> x-1..
Warning: this matching is not exhaustive.
pred : int -> int = <fun>

# pred 0;;
- : int = 0

# pred 20;;
- : int = 19

# pred (-1);;
Uncaught exception: Match_failure ("", 151, 194)
```

In fase di valutazione, si tenta di legare l'argomento della funzione da valutare ad uno dei patterns enumerati – considerati nell'ordine in cui sono specificati. Se l'associazione ha successo, allora (eventualmente valutando la condizione di guardia) viene selezionata l'espressione associata al pattern, che viene quindi sottoposta a valutazione. Questa operazione di selezione è chiamata *pattern matching*. Il pattern matching è una nozione alla base della programmazione in stile equazionale; esso porta ad un metodo di programmazione semplice e facilmente comprensibile, e semplifica il processo di ragionamento formale sulle funzioni.

Supponiamo di voler definire una funzione che applicata ad una tripla (giorno,mese,anno) dove giorno è un intero, mese è una stringa e anno è un intero, restituisca true se e solo se la terna (giorno,mese,anno) rappresenta una data corretta. Si utilizzi la funzione bisestile definita nel Paragrafo 2.2.

```
# let date (giorno,mese,anno) = match
    mese with
    "gennaio" | "marzo" | "maggio" | "luglio" |
    "agosto" | "ottobre" | "dicembre"
        -> giorno>=1 & giorno<=31
    | "aprile" | "giugno" | "settembre" | "novembre"
        -> giorno>=1 & giorno<=30
    | "febbraio" -> if bisestile anno then
        giorno>=1 & giorno<=29
        else
        giorno>=1 & giorno<=28
    | _ -> false;;
```

```

date : int * string * int -> bool = <fun>

# date (3, "novembre", 2010);;
- : bool = true

# date (29, "febbraio", 2009);;
- : bool = false

# date (29, "febbraio", 2012);;
- : bool = true

# date (31, "novembre", 2010);;
- : bool = false

# date (35, "novembre", 2010);;
- : bool = false

```

2.6 – Funzioni

Come abbiamo già visto nei precedenti paragrafi, gli argomenti (ed i risultati) delle funzioni non hanno restrizioni di tipo, ma possono assumere qualsiasi valore. Abbiamo infatti visto come si possano definire delle funzioni higher-order, che prevedono tra gli argomenti una funzione e restituiscono come risultato un'altra funzione. Vediamo alcuni esempi di funzioni higher-order.

2.6.1 – Composizione di funzioni

In CAML è abbastanza semplice esprimere la composizione di funzioni:

```

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

# let succ n = n+1
    and double n = 2*n
    in compose succ double;;
- : int -> int = <fun>

```

La funzione `compose` è ovviamente polimorfa: le uniche restrizioni sono che `g` abbia codominio dello stesso tipo del dominio di `f`. È facile vedere che la funzione `compose` gode della proprietà associativa.

2.6.2 –Funzioni inverse

Supponiamo che $f : A \rightarrow B$ sia una funzione *iniettiva*, (cioè tale che, per ogni $x, y \in A$, $f(x) = f(y)$ se e solo se $x = y$). È possibile allora associare a tale funzione una ed una sola funzione *inversa*, denotata con $f^{-1} : B \rightarrow A$, tale che $f^{-1}(f(x)) = x$ per ogni $x \in A$. Per esempio, la funzione

```
let (f : int -> int * int) x = (sign x, abs x)
```

è iniettiva ed ha come inversa la funzione

```
let (g : int * int -> int) (x, y) = x * y
```

ESERCIZIO 2.5

Si consideri la funzione

```
let h x y = f (g x y)
```

per due funzioni date f e g . Quali delle seguenti uguaglianze sono vere ?

$$h = \text{compose } f \ g$$

$$h \ x = \text{compose } f \ (g \ x)$$

$$h \ x \ y = (\text{compose } f \ g) \ x \ y$$

2.7 – Inferenza di tipo

Abbiamo detto che il sistema, nel valutare un'espressione, prova a derivare il suo tipo e, se riesce ad associare all'espressione un tipo valido, esegue i passi di riduzione. In questa sezione vedremo un esempio del metodo attuato dal sistema per derivare il tipo di un'espressione.

Supponiamo di voler valutare l'espressione $f(g x)$. Dalla sintassi dell'espressione possiamo dedurre che f deve essere una funzione e $g x$ un suo valido argomento:

$$\begin{aligned} f &: A \mapsto B \\ g x &: A \end{aligned}$$

per qualche coppia di tipi A e B . Le equazioni ci dicono che il tipo del valore d'ingresso di f e quello di $g x$ devono coincidere. Poiché anche $g x$ è una applicazione di funzione, possiamo applicare la stessa regola utilizzata sopra e ottenere:

$$\begin{aligned} g &: C \mapsto A \\ x &: C \end{aligned}$$

In questo caso, il codominio di $g x$ deve essere A , e il dominio deve essere identico a quello di x . Supponiamo di instanziare f con `min`, g con `square` e x con `2`. In tal caso, le equazioni introdotte sopra diventano:

```
min : 'a -> 'a -> 'a
square : int -> int
2 : int
```

D'altra parte, dalle equazioni sopra otteniamo che

```
square 2 : int
min square 2 : int -> int.
```

Nell'effettuare l'analisi, abbiamo applicato le seguenti regole:

Applicazione: Se $f x : A$, allora $x : B$ e $f : B \mapsto A$ per qualche B .

Uguaglianza: Se per la stessa variabile x si deduce che $x : B$ e $x : A$, allora $A = B$.

Funzione: Se $A \mapsto B = A' \mapsto B'$, allora $A = A'$ e $B = B'$.

Se al termine dell'analisi qualche parametro non viene istanziato, allora tale parametro può assumere qualsiasi valore tra i tipi ammissibili. Per esemplificare questo caso, proviamo a calcolare il

tipo di `min id`. La regola dell'applicazione ci dice che, se $\text{min id} : t$, allora $\text{min} : t' \rightarrow t$ e $\text{id} : t'$. D'altra parte, sappiamo che $\text{id} : 'a \rightarrow 'a$, e che $\text{min} : 'b \rightarrow 'b \rightarrow 'b$. Dalla regola dell'uguaglianza, otteniamo che $'b \rightarrow 'b \rightarrow 'b = t' \rightarrow t$ e $'a \rightarrow 'a = t'$. Ancora, dalla regola della funzione otteniamo: $t = 'b \rightarrow 'b$ e $'b = t'$. Mettendo tutto insieme, otteniamo $\text{min id} : 'a \rightarrow 'a \rightarrow 'a$.

ESERCIZIO 2.7

Si considerino le seguenti definizioni di funzioni:

```
let const x y = x  
let subst f g x = f x (g x)
```

Si deduca il loro tipo.

ESERCIZIO 2.8

Quali sono i tipi delle seguenti espressioni ?

```
uncurry compose  
compose curry uncurry  
compose uncurry curry
```

Capitolo 3 – Ricorsione ed Induzione

In questo capitolo studieremo le idee che stanno alla base delle definizioni **ricorsive** (cioè definizioni di funzioni espresse in termini di sé stesse), e analizzeremo una nozione correlata: l'**induzione**.

3.1 – Ricorsione

Immaginiamo di voler calcolare quanti sono i possibili ordinamenti che si possono ottenere mescolando un mazzo di carte (si tratta di calcolare il numero di permutazioni delle carte). Per risolvere questo problema possiamo ragionare nel seguente modo. Ci sono 52 scelte possibili per la prima carta. Per ciascuna di queste scelte ci sono 51 scelte possibili per la seconda carta. Quindi ci sono 52×51 scelte possibili per le prime due carte. Continuando ad applicare questo ragionamento, si ottiene che i possibili ordinamenti di un mazzo di carte sono $52 \times 51 \times 50 \times \dots \times 3 \times 2 \times 1$. Convenzionalmente questo numero viene chiamato **fattoriale di 52** ed è scritto **52!**. Per calcolare questo risultato al computer potremmo scrivere 51 moltiplicazioni a mano per tutti i numeri che vanno da 52 fino a 1, ma sarebbe molto scomodo. In alternativa, potremmo scrivere una funzione generale per calcolare il fattoriale di un numero qualsiasi, e applicare quindi questa funzione al numero 52.

Il problema consiste nel moltiplicare 52 numeri, ma sappiamo che dopo aver moltiplicato 51 numeri ce ne resta uno solo da moltiplicare. Ci sono diversi modi per scegliere quali sono i primi numeri da moltiplicare e quale numero moltiplicare per ultimo, e sono tutte scelte validissime. Alcune scelte tuttavia rendono il problema più semplice da risolvere: moltiplichiamo prima i numeri dal più piccolo fino al penultimo, e quindi moltiplichiamo il risultato così ottenuto per il numero più grande che avevamo tenuto da parte. In questo modo, la fase iniziale della strategia ricorsiva consiste nell'applicare la funzione fattoriale all'argomento decrementato di 1, e quindi moltiplicare il valore così ottenuto per l'argomento. Questo ci permette di riutilizzare la definizione di fattoriale all'interno del fattoriale stesso! In termini matematici stiamo dicendo che:

$$n! = \begin{cases} 1 & \text{se } n \leq 1 \\ n \times (n-1)! & \text{altrimenti} \end{cases}$$

È importante notare che stiamo stabilendo un **caso base** nel dire che quando $n \leq 1$ possiamo restituire direttamente il valore 1 come risultato. Senza un caso base non ci fermeremmo mai, e rischieremmo di calcolare il fattoriale all'infinito, senza mai fermarci: $3 \times 2 \times 1 \times 0 \times (-1) \times (-2) \times \dots$.

Per definire questa particolare funzione in CaML dobbiamo usare il costrutto **if** e un costrutto particolare chiamato **let rec**:

```
# let rec fact = fun n ->
    if n <= 1 then
        1
    else
        n * (fact (n-1));
fact : int -> int = <fun>
```

Si noti l'uso del costrutto `let rec`, invece del semplice `let`, per denotare la dichiarazione del nome della funzione. Il costrutto `let rec` indica al compilatore che intendiamo usare il nome della funzione all'interno del corpo della funzione stessa. Se avessimo scritto:

```
# let fact = fun n ->
    if n <= 1 then
        1
    else
        n * (fact (n-1));;
Toplevel input:
>           n * (fact (n-1));;
>                                         ^^^^
The value identifier fact is unbound.
```

allora il compilatore ci avrebbe segnalato che l'identificatore `fact`, che compare all'interno del corpo della definizione di `fact`, non è ancora stato definito (è `unbound`).

Si osservi che se si maneggiano numeri piuttosto grandi si incontrano delle strane sorprese

```
# fact 3;;
- : int = 6

#fact 12;;
- : int = 479001600

#fact 15;;
- : int = -143173632

# fact 52;;
- : int = 0
```

Le implementazioni CaML e F# di ML forniscono delle librerie che consentono di eseguire le operazioni aritmetiche sui numeri interi e decimali con maggior precisione. In particolare in CaML è possibile utilizzare il tipo numerico `num` che consente di eseguire operazioni precise sui numeri decimali: le operazioni ammesse sugli elementi di tipo `num` sono:

+ /	addizione
- /	sottrazione
* /	moltiplicazione
/ /	divisione

La funzione `num_of_int` consente di trasformare un numero intero di tipo `int` in un numero di tipo `num`. Possiamo scrivere quindi la funzione fattoriale nel seguente modo:

```
# let rec fact = fun n ->
    if n <= 1 then
        (num_of_int 1)
    else
        (num_of_int n) */ (fact (n-1));;
fact : int -> num = <fun>
```

In questo caso otteniamo

```
# fact 3;;
- : num = 6

# fact 12;;
- : num = 479001600

# fact 15;;
- : num = 1307674368000

# fact 52;;
- : num =
806581751709438785716606368564037669752895054408832778240000000000000
```

IL TIPO BIGINT IN F#

In F# possiamo utilizzare il tipo predefinito bigint nel seguente modo:

```
# let rec fact = fun n ->
    if n <= 1I then
        1I
    else
        n * (fact (n-1I));;
fact : bigint -> bigint = <fun>
```

In questo caso abbiamo scritto 1I al posto della costante 1. Il risultato è che la funzione prende come input un valore di tipo `bigint` e restituisce un valore di tipo `bigint`, interi a precisione arbitraria, piuttosto che lavorare con i numeri interi tradizionali (più efficienti dal punto di vista computazionale ma limitati e del tutto insufficienti per il calcolo di fattoriali di numeri anche ridotti). Otteniamo dunque:

```
# fact 52;;
- : bigint =
806581751709438785716606368564037669752895054408832778240000000000000I
```

Possiamo definire la funzione fattoriale utilizzando il costrutto `match ... with ...` nel seguente modo:

```
# let rec fact = fun n ->
    match n with
        0 -> 1
        | 1 -> 1
        | n -> n * (fact (n-1));;
fact : int -> int = <fun>
```

Questa funzione tuttavia è ben definita solo nel caso di numeri interi non negativi. Infatti:

```
# fact (-3);;
Uncaught exception: Out_of_memory
```

La seguente definizione mette in evidenza il fatto che la funzione `fact` non è definita sui numeri interi negativi. Usiamo la funzione `failwith` che si comporta nel seguente modo:

```
# failwith;;
- : string -> 'a = <fun>

# failwith "pippo";;
Uncought exception: Failure "pippo"

# let rec fact = fun n ->
    match n with
        0 -> 1
    | 1 -> 1
    | n when n > 1 -> n * (fact (n-1))
    | _ -> failwith "argomento negativo";;
fact : int -> int = <fun>
```

Si noti l'uso della funzione `failwith` per gestire le eccezioni. In tal caso si ha:

```
# fact (-3);;
Uncought exception: Failure "argomento negativo"
```

In tutti i casi descritti sopra, abbiamo definito la funzione fattoriale scomponendo il problema in un sottoproblema più semplice ma con la stessa forma del problema iniziale. Più precisamente, abbiamo definito una **funzione ricorsiva**. Inoltre abbiamo garantito la terminazione mediante la definizione di un **caso base**.

Quando si definiscono delle funzioni ricorsive è necessario garantire che tutte le chiamate ricorsive prima o poi finiscono in un caso base, assicurando che le computazioni terminino.

L'IMPERATIVO DEL CASO BASE:

In una procedura ricorsiva, tutte le strade devono portare al caso base.

La natura ricorsiva della funzione `fact` è evidenziata nella Figura 3.1, che mostra come la valutazione di `(fact 3)` si riduce al calcolo del sottoproblema `(fact 2)`, che infine richiede di calcolare `(fact 1)`. Se avessimo dovuto risolvere il problema `(fact 52)`, il diagramma sarebbe stato largo 52 colonne invece che solamente 3. Si noti che nel diagramma della Figura 3.1, la valutazione dell'espressione `if` e del suo test non sono mostrate esplicitamente, così come non è espansa la sottrazione di 1. Queste omissioni sono state fatte per semplificare il diagramma, concentrandosi solo sui passaggi essenziali. Senza queste semplificazioni il diagramma si espanderebbe in ben 10 passi!

Possiamo elencare i passi di valutazione. Continuando ad omettere i dettagli più ovvi, otteniamo:

```
(fact 3)
3 * (fact 2)
3 * (2 * fact 1)
3 * (2 * 1)
3 * 2
6
```

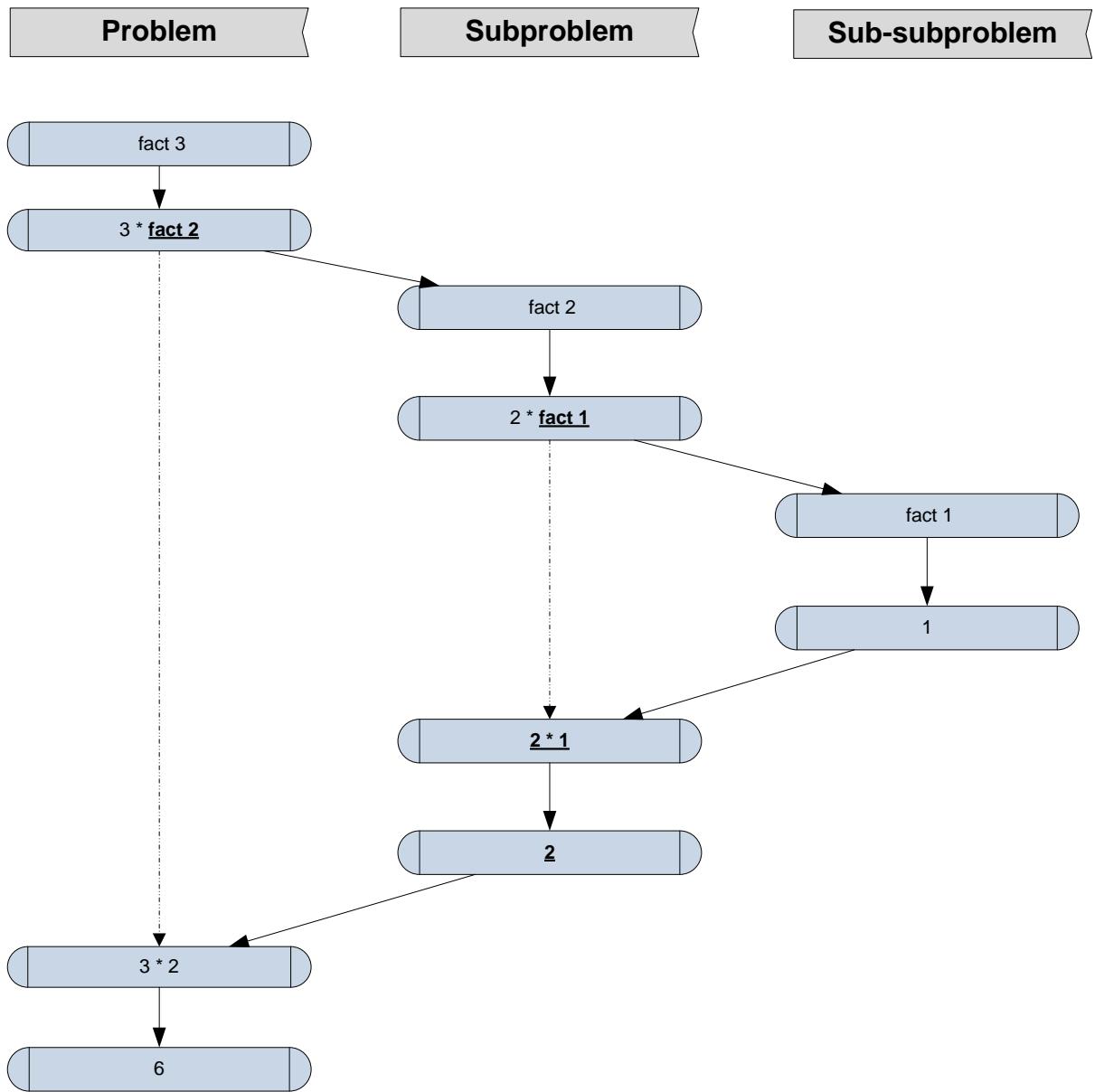


Figura 3.1

3.2 – La definizione dell'operatore esponenziale

ESPOVENTI

In questa dispensa, quando usiamo un esponente come k in x^k , significa quasi sempre che k é un intero positivo o nullo. Quando k è un intero positivo, allora x^k significa k copie di x moltiplicate tra loro:

$$x^k = \overbrace{x \times x \times \dots \times x}^{k \text{ volte}}$$

Cosa succede quando $k = 0$? Poichè avremmo potuto scrivere anche:

$$x^k = 1 \times \overbrace{x \times x \times \dots \times x}^{k \text{ volte}}$$

allora per $k = 0$ abbiamo:

$$x^0 = 1$$

che è il caso base dell'elevamento a potenza.

Vogliamo definire l'operatore di esponenziazione. Abbiamo già introdotto l'operatore con il simbolo `**`; cercheremo ora di ridefinirlo mediante una funzione ricorsiva, quando l'esponente è un numero intero non negativo. Ci poniamo dunque il problema di scrivere una funzione ricorsiva `exp` tale che `exp x n` elevi il numero intero `x` alla potenza `n`, dove `n` è un numero intero non negativo. Come spiegato nella nota sopra sugli esponenti, una implementazione corretta deve restituire 1 quando l'esponente é uguale a 0. Possiamo scrivere la seguente definizione ricorsiva:

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ x \times x^{n-1} & \text{altrimenti} \end{cases}$$

In CaML, possiamo riscrivere tali equazioni nel seguente modo:

```
# let rec exp x n = match n with
  0 -> 1
  | n when n > 0 -> x * (exp x (n-1));;
Toplevel input:
>.....match n with
>      0 -> 1
>      | n when n > 0 -> x * (exp x (n-1))..
Warning: this matching is not exhaustive.
exp : int -> int -> int = <fun>
```

Si noti come la definizione tramite pattern matching rispecchi esattamente le equazioni originali. La funzione non è definita per esponenti che siano numeri interi negativi. Possiamo utilizzare il costrutto `failwith` per trattare il caso di esponenti negativi.

```
# let rec exp x n = match n with
  0 -> 1
  | n when n > 0 -> x * (exp x (n-1))
  | _ -> failwith "esponente negativo";;
exp : int -> int -> int = <fun>
```

Per calcolare $\exp 2 3$, possiamo ridurre l'espressione nel seguente modo:

$$\begin{aligned} \exp 2 3 &= \{\text{secondo pattern di } \exp\} \\ &= 2 \times (\exp 2 2) \\ &= \{\text{secondo pattern di } \exp\} \\ &= 2 \times (2 \times (\exp 2 1)) \\ &= \{\text{secondo pattern di } \exp\} \\ &= 2 \times (2 \times (2 \times (\exp 2 0))) \\ &= \{\text{primo pattern di } \exp\} \\ &= 2 \times (2 \times (2 \times 1)) \\ &= \{\text{algebra}\} \end{aligned}$$

8

In questi tipi di definizioni c'è un problema di fondo: in base a cosa possiamo dire che la definizione data è valida (e conseguentemente che la funzione è ben definita)? Nel nostro caso è facile convincersi di ciò: $\exp x 0$ ha un valore, poiché tale valore è dato dalla prima equazione del pattern matching. Per la seconda equazione possiamo dire che se $\exp x (n - 1)$ ha un valore, allora anche $\exp x n$ è definito. Così \exp ha un valore per ogni numero naturale n , come richiesto. Nella pratica, tale forma di ragionamento, può essere anche applicata all'indietro: dati x ed n , il valore di $\exp x n$ è trovato calcolando il valore di $\exp x (n - 1)$, il quale è a sua volta trovato calcolando il valore di $\exp x (n - 2)$, e così via, fino a quando non ci si riduce a dover calcolare il valore di $\exp x 0$. Questo tipo di ragionamento è un esempio di dimostrazione tramite *induzione matematica*.

In generale, per dimostrare che una proprietà $P(n)$ è valida per ogni numero naturale n , utilizzando il metodo dell'induzione matematica bisogna dimostrare che:

(Caso 0) $P(0)$ è valida.

(Caso n+1) Se $P(n)$ è valida, allora anche $P(n + 1)$ è valida, per ogni numero naturale n .

Il Caso 0 è detto anche *Caso base* mentre il Caso n+1 è detto *Passo induttivo*. È facile convincersi che se tali proprietà sono valide, allora $P(n)$ è valida per ogni valore di n . Infatti, sappiamo che $P(0)$ è valida, per il primo caso. Allora, per il secondo caso, deve essere valida $P(1)$; ancora, per il secondo caso, è valida $P(2)$, e così via.

Proviamo, per esempio, a dimostrare che

$$\exp x (m + n) = (\exp x m) \times (\exp x n)$$

per ogni numero naturale m e n .

Dimostrazione. Dimostriamolo per induzione su m .

(Caso base) Sia $m = 0$. Abbiamo:

$$\exp x (0 + n)$$

$$= \{0 \text{ è l'elemento neutro della somma}\}$$

$$\exp x n$$

$$= \{1 \text{ è l'elemento neutro del prodotto}\}$$

$$1 \times (\exp x n)$$

$$= \{\text{primo pattern di } \exp\}$$

$$(\exp x 0) \times (\exp x n).$$

(Passo induttivo) Assumiamo che la proprietà sia vera per m , e cioè che valga la seguente ipotesi:
 $\exp x (m + n) = (\exp x m) \times (\exp x n)$ (tale ipotesi è detta anche *ipotesi induttiva*). Dimostriamo che la proprietà è vera anche per $m + 1$, e cioè che $\exp x ((m + 1) + n) = (\exp x (m + 1)) \times (\exp x n)$.

$$\exp x ((m + 1) + n)$$

$$= \{\text{proprietà della somma}\}$$

$$\exp x ((m + n) + 1)$$

$$= \{\text{secondo pattern di } \exp\}$$

$$x \times (\exp x (m + n))$$

$$= \{\text{ipotesi induttiva}\}$$

$$x \times ((\exp x m) \times (\exp x n))$$

$$= \{\text{proprietà associativa del prodotto}\}$$

$$(x \times (\exp x m)) \times (\exp x n)$$

$$= \{\text{secondo pattern di } \exp\}$$

$$(\exp x (m + 1)) \times (\exp x n).$$

3.3 – I numeri di Fibonacci

Un altro “classico” esempio di definizione ricorsiva è dato dai [numeri di Fibonacci](#). In matematica, l’ennesimo numero di Fibonacci, F_n , per $n \geq 0$, è calcolato tramite le seguenti equazioni:

$$F_0 = 0$$

$$F_1 = 1$$

$$F_n = F_{n-1} + F_{n-2} \quad \text{per } n > 1$$

Ad esempio, i numeri da F_0 a F_9 sono:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34$$

dove ogni numero della sequenza è la somma dei due che lo precedono.

Definiamo la funzione ricorsiva `fib n`, che calcola F_n :

```
# let rec fib n = match n with
    0 -> 0
    | 1 -> 1
    | n -> fib (n-1) + fib (n-2);;
fib : int -> int = <fun>
```

Anche in questo caso ogni pattern rispecchia una delle equazioni precedentemente definite. Più precisamente, possiamo scrivere:

```
# let rec fib n = match n with
    0 -> 0
    | 1 -> 1
    | n when n > 1 -> fib (n-1) + fib (n-2)
    | _ -> failwith "argomento negativo";;
fib : int -> int = <fun>
```

I numeri di Fibonacci godono di un certo numero di proprietà. Per esempio, le radici dell'equazione $x^2 - x - 1 = 0$ sono:

$$\phi = \frac{1+\sqrt{5}}{2} \quad \text{e} \quad \hat{\phi} = \frac{1-\sqrt{5}}{2}$$

e soddisfano $\phi^2 = \phi + 1$ e $\hat{\phi}^2 = \hat{\phi} + 1$. Tali valori sono correlati ai numeri di Fibonacci:

$$F_n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n)$$

per ogni n .

Dimostrazione. Dimostriamolo per induzione su n . Poniamo $c = \frac{1}{\sqrt{5}}$.

(Caso 0) Abbiamo: $F_0 = 0 = c(\phi^0 - \hat{\phi}^0)$.

(Caso 1) È facile vedere come $F_1 = 1 = c(\phi^1 - \hat{\phi}^1)$.

(Caso k+2) Assumiamo per ipotesi induttiva che $F_n = c(\phi^n - \hat{\phi}^n)$ e $F_{n+1} = c(\phi^{n+1} - \hat{\phi}^{n+1})$. Allora:

$$F_{n+2}$$

= {terza equazione della definizione}

$$F_n + F_{n+1}$$

= {ipotesi induttiva}

$$c(\phi^n - \hat{\phi}^n) + c(\phi^{n+1} - \hat{\phi}^{n+1})$$

= {aritmetica}

$$c(\phi^n(1 + \phi) - \hat{\phi}^n(1 + \hat{\phi}))$$

$$= \{ \phi^2 = \phi + 1 \text{ e } \hat{\phi}^2 = \hat{\phi} + 1 \}$$

$$c(\phi^{n+2} - \hat{\phi}^{n+2}).$$

Si noti che il principio di induzione utilizzato in questo caso è leggermente differente da quello dato nella definizione precedente. Qui, infatti, abbiamo mostrato che:

1. $P(0)$ è vera;
2. $P(1)$ è vera;
3. Fissato n , se $P(n)$ è vera e $P(n + 1)$ è vera, allora anche $P(n + 2)$ è vera.

Utilizzando un'argomentazione simile alla precedente è facile convincersi che tale principio è valido.

3.4 – Induzione

Come possiamo essere sicuri che le funzioni `fact`, `exp` e `fib` viste precedentemente siano corrette? Nel caso specifico è facile convincersene: basta provarle, ragionarci su un momento e la logica delle funzioni risulta evidente. In altri casi però è molto più difficile convincersi che una funzione ricorsiva è corretta.

Consideriamo la seguente funzione ricorsiva per elevare al quadrato un numero naturale:

```
# let rec square = fun n ->
    if n = 0 then
        0
    else
        n * n - 1 + (square (n-1));
square : int -> int = <fun>
```

Il solo fatto di chiamarsi `square` non significa che la funzione elevi effettivamente al quadrato il suo argomento: il nome della funzione potrebbe non aver alcuna relazione con la sua definizione. Niente impedisce di chiamare `square` una funzione che fa tutt'altro! Naturalmente la pretesa che una funzione si comporti in accordo con quello che il suo nome indica deve essere accompagnata da prove e spiegazioni da parte di colui che ha scritto la funzione. Dobbiamo quindi trovare un modo per garantire

la correttezza della funzione. La nostra funzione calcola correttamente il quadrato di un numero non negativo. Sappiamo infatti che certamente eleva correttamente al quadrato il numero 0, poiché restituisce 0 quando il parametro di input è 0, e $0^2 = 0$. Le nostre (ragionevoli) assunzioni sono che $(-)$ e $(+)$ sottraggano e sommino correttamente numeri interi, e che quindi $n + n - 1$ valuti a $2 \times n - 1$. Facciamo un passo avanti ed assumiamo anche che quando la funzione `square` viene applicata ad $n - 1$ ci restituisce $(n - 1)^2$. Allora possiamo dire che la funzione applicata a n restituisce in output:

$$\begin{aligned} 2 \times n - 1 + (n - 1)^2 &= \\ 2 \times n - 1 + n^2 - 2 \times n + 1 &= \\ n^2 \end{aligned}$$

che è proprio quello che ci aspettiamo. Ma attenzione: per mostrare che la funzione è corretta per un valore n generico, abbiamo dovuto assumere che fosse corretta per $n - 1$. Sembra quasi che per dimostrare che la funzione è corretta, dobbiamo assumere che la funzione è corretta! Il ragionamento è circolare solo in apparenza. Infatti per dimostrare che la funzione è corretta per 10, allora dobbiamo assumere che è corretta anche per 9. Ma per dimostrare che la funzione è corretta per 9, dobbiamo assumere che è corretta anche per 8. Questa catena ad un certo punto si interrompe, perché quando dobbiamo dimostrare che la funzione è corretta per 0, non dobbiamo andare oltre: sappiamo per definizione che la nostra funzione applicata a 0 calcola il valore corretto. La spirale si ferma al caso base, e comincia a risalire fino a dimostrare che in effetti la funzione è corretta quando viene applicata a 10.

La definizione seguente, invece, benché l'espressione algebrica $n^2 = (n + 1)^2 - 2 \times n - 1$ sia corretta, non funziona:

```
# let rec square' = (*Questa funzione non è corretta.*)
  fun n ->
    if n = 0 then
      0
    else
      square' (n+1) - (n + n + 1);;
square' : int -> int = <fun>
```

Il motivo per cui questa funzione non è corretta risiede nel fatto che essa in realtà non calcola nulla: ad ogni iterazione si allontana sempre di più dal caso base, senza mai riuscire a convergere. Ad un certo punto la memoria occupata dal processo computazionale è troppa ed il sistema riporterà un fallimento. Diciamo che una tale funzione *non termina*.

```
# square' 3;;
Uncought exception: Out_of_memory
```

La tecnica di ragionamento che abbiamo appena usato è talmente utile e diffusa da essersi meritata con gli anni un nome: *induzione matematica*. È stata anche introdotta una terminologia per rendere più agevole l'uso di dimostrazioni in questa forma. La giustificazione del funzionamento del caso base è chiamata *caso base della dimostrazione*. L'assunzione che la funzione sia corretta per valori più piccoli del parametro di input si chiama *ipotesi induttiva*, ed il ragionamento che dall'ipotesi induttiva ci porta alla correttezza dell'operazione si chiama *passo induttivo*. Si osservi che il passo induttivo va applicato solo quando non si può applicare il caso base. Per la procedura `square` abbiamo ragionato da $n - 1$ a n solo per $n > 0$, mai per $n = 0$.

NOTA: COMMENTI AL CODICE

A volte può essere utile aggiungere al listato del proprio codice delle annotazioni, note come commenti, che servono a rendere più agevole la lettura del codice e a segnalare eventuali comportamenti particolari di cui l'utente che adopera il nostro codice è bene sia a conoscenza. A questo proposito si aggiungono i cosiddetti commenti, compresi tra (* e *). Esempio:

```
(* funzione fattoriale *)
let rec fact = fun n ->
    match n with
    | 0 -> 1
    | n -> n * fact (n-1);;
```

In F# i commenti possono essere alternativamente indicati con una coppia di barre diagonali all'inizio del commento:

```
// funzione fattoriale
let rec fact = fun n ->
    match n with
    | 0 -> 1
    | n -> n * fact (n-1);;
```

Inoltre F# consente un tipo particolare di commento che usa tag XML per denotare aree specifiche del commento, in modo da avere commenti ben strutturati che descrivono tutto quello che c'è da sapere riguardo una determinata funzione. Questi commenti iniziano con una terna di barre diagonali, ed esistono compilatori che trasformano tali commenti in un formato navigabile (ad esempio una pagina web contenente tutti i commenti divisi in pagine – una per funzione – e mantenendo anche la gerarchia dei file sorgenti). Un possibile commento ben strutturato per la funzione `square` potrebbe essere:

```
/// <summary>
/// Funzione che calcola il quadrato di un numero intero non negativo.
/// </summary>
/// <remarks>
/// Se il parametro <paramref name="n"/> e' negativo la funzione non termina.
/// </remarks>
/// <param name="n">Il numero da elevare al quadrato.</param>
/// <returns>Il quadrato di <paramref name="n"/></returns>
let rec square = fun n ->
    if n = 0 then 0
    else n + n - 1 + square (n-1);;
```

Anche se per la funzione `square` un tale commento può sembrare inappropriato, nel caso di progetti consistenti (e in cui non si è i soli programmati) adottare uno standard di documentazione esaustivo e completo può migliorare notevolmente la qualità del prodotto.

Scriviamo la dimostrazione della correttezza della procedura `square` usando l'induzione matematica:

(Caso base) Sia $n = 0$. Abbiamo:

`square 0`

= {primo ramo del costrutto `if`}

0

= {per le regole dell'algebra}

0×0

= {per le regole dell'algebra}

0^2 .

(Passo induttivo) Sia $n > 0$. Assumiamo per ipotesi che `square n = n2` (tale ipotesi è chiamata *ipotesi induttiva*) e dimostriamo che `square (n + 1) = (n + 1)2`. Otteniamo:

`square (n + 1)`

= {secondo ramo del costrutto `if`}

$(n + 1) + n + (\text{square } n)$

= {per ipotesi induttiva}

$(n + 1) + n + n^2$

= {per le regole dell'algebra}

$2 \times n + 1 + n^2$

= {per le regole dell'algebra}

$(n + 1)^2$.

Conclusione: Abbiamo dimostrato per induzione matematica su n , che `square n` termina con il valore n^2 per ogni valore intero non negativo di n .

Osserviamo due cose riguardo a questa dimostrazione. La prima è che la dimostrazione è comunque relativa all'assunzione che le "operazioni elementari" (come ad esempio somma e sottrazione) si comportino come ci aspettiamo. In secondo luogo una parte importante della dimostrazione è quella in cui si mostra che effettivamente la procedura termina per i valori ammessi. La necessità di garantire la terminazione spiega la nostra attenzione sull'imperativo del caso base.

La capacità di dimostrare la correttezza può essere utile anche quando si cerca di correggere una procedura che non funziona correttamente, ossia quando si sta cercando di capire cosa non funziona e

come correggere l'errore. Come esempio consideriamo la funzione `square'` che abbiamo visto prima. Se cercassimo di dimostrarne la correttezza per induzione, il caso base e l'ipotesi induttiva sarebbero identici a quelli della dimostrazione vista sopra. Nel passo induttivo però incontreremmo delle difficoltà:

(Passo induttivo) Sia $n > 0$. Assumiamo per ipotesi induttiva che $\text{square}' n = n^2$ e dimostriamo che $\text{square}'(n + 1) = (n + 1)^2$. Otteniamo:

$$\begin{aligned} \text{square}'(n + 1) \\ = & \quad \{\text{secondo ramo del costrutto if}\} \\ \text{square}'(n + 2) - (n + 1 + n + 1 + 1) \\ = & \quad \{\text{per le regole dell'algebra}\} \\ \text{square}'(n + 2) - (2 \times n + 3) \\ = & \quad \{\text{per ipotesi induttiva}\} \end{aligned}$$

Ooops! Possiamo applicare l'ipotesi induttiva ????

ESERCIZIO 3.1

Ecco un esempio di una funzione con un errore sottile. L'errore si può trovare cercando di fare una dimostrazione induttiva di correttezza. Si provi a dimostrare che la funzione calcola il quadrato del parametro di input quando questo è non negativo. Dov'è l'errore ?

```
# let rec square = fun n ->
    if n = 0 then 0
    else 4 * n - 4 + square (n-2);;
```

L'induzione ci porta naturalmente ad un nuovo modo di ragionare e di concepire le funzioni. Si pensi alla funzione ricorsiva `square` che abbiamo definito all'inizio della sezione: ci si sarebbe potuti facilmente domandare la ragion d'essere di una funzione così strana. La risposta è che in realtà, per costruire questa funzione, abbiamo semplicemente sfruttato l'idea di calcolare il quadrato di un numero in termini dei quadrati di numeri più piccoli. Sapevamo di aver bisogno di un caso base, che presumibilmente sarebbe stato $n = 0$. Sapevamo anche che ci sarebbe servito un modo per "collegare" il quadrato di n al quadrato di qualche numero più piccolo, secondo questo modulo da riempire:

```
let rec square = fun n ->
    if n = 0 then 0
    else _____ square _____
```

Sapevamo che il parametro della chiamata ricorsiva sarebbe dovuto essere più piccolo di n per poter applicare l'ipotesi induttiva; d'altro canto, per assicurare che rimanesse un intero non negativo, non potevamo decrementarlo di troppo. Il modo più semplice è chiaramente quello di decrementare di uno:

```
let rec square = fun n ->
  if n = 0 then 0
  else _____ square (n-1)
```

A questo punto il principio di induzione ci suggerisce di non preoccuparci di cosa accade dentro la chiamata ricorsiva `square(n-1)`, e piuttosto di assumere che faccia correttamente il suo lavoro. Resta solo un po' di algebra per capire che dobbiamo sommare al valore ritornato dalla chiamata ricorsiva il valore $2 \times n - 1$:

```
let rec square = fun n ->
  if n = 0 then 0
  else (n+n-1) + square (n-1);;
```

Esercizio 3.2

Usare il ragionamento ricorsivo per completare correttamente la seguente versione della funzione `square`, sempre limitata alla gestione di numeri non negativi:

```
let rec square = fun n ->
  if n = 0 then
    0
  else if (n % 2 = 0) then
    _____ (square (n / 2))
  else
    (square (n - 1)) + (n + n - 1);;
```

3.5 – Altri esempi

La ricorsione ci permette di fare moltissime cose in più rispetto a prima. Inoltre la strategia di ricorsione sarà fondamentale in tutto il resto del corso. Nonostante ciò, specie quando si incontra questo modo di ragionare per la prima volta, lo si può trovare piuttosto indigesto. Una parte della confusione deriva proprio dall'apparente “circolarità”, che in realtà si riduce ad una spirale che converge in un caso base. Un'altra fonte di difficoltà è molto semplicemente la mancanza di familiarità. Dedichiamo quindi questa sezione a vari esempi di funzioni numeriche che usano la ricorsione.

Costruiamo la funzione `quotient` che calcola il quoziente di due numeri interi, come mostrato di seguito:

```
(quotient 9 3)
3
```

```
(quotient 10 3)
3
```

```
(quotient 11 3)
3
```

```
(quotient 12 3)
4
```

Anche se una versione di questa funzione è già presente (l'operatore `(/)`), è molto istruttivo vedere come la si può definire in termini di operazioni più elementari (in questo caso la sottrazione). Per semplificare la discussione, supponiamo di voler calcolare `(quotient n d)`, con $n \geq 0$ e $d > 0$. Se $n < d$, allora d non divide n e il risultato sarà 0. Se, invece, $n \geq d$, allora d divide n almeno una volta in più di quanto non divida $n - d$. Possiamo dunque scrivere:

```
# let rec quotient = fun n d ->
  if n < d then
    0
  else
    1 + (quotient (n-d) d);;
quotient : int -> int -> int = <fun>
```

A differenza della funzione appena vista, la versione predefinita dell'operatore quoziante `(/)` permette di calcolare il quoziante sia di numeri positivi che di numeri negativi. Quando uno o entrambi gli argomenti sono negativi possiamo definire il quoziante come segue:

```
# let rec quotient = fun n d ->
  if n < 0 then
    -(quotient (-n) d)
  else if d < 0 then
    -(quotient n (-d))
  else if n < d then
    0
  else 1 + (quotient (n-d) d);;
quotient : int -> int -> int = <fun>
```

ESERCIZIO 3.3

Definire una funzione ricorsiva che calcoli il prodotto tra due numeri interi usando solo somme.

Supponiamo ora di voler scrivere una funzione che calcola la somma dei numeri interi compresi tra 0 e n , con $n \geq 0$. Questo problema è molto simile a quello del fattoriale, con la sola differenza che dobbiamo sommare i numeri invece di moltiplicarli tra loro. Poiché il caso base $n = 0$ restituisce 0, la soluzione ha un aspetto identico a quello del fattoriale:

```
# let rec sum_first = fun n ->
  match n with
  | 0 -> 0
  | n -> n + sum_first (n-1);;
sum_first : int -> int = <fun>
```

ESERCIZIO 3.4

Consideriamo alcune funzioni aventi la stessa forma delle funzioni `fact` e `sum_first`.

a. Descrivere cosa calcola la seguente funzione in termini di n :

```
# let rec subtract_the_first = fun n ->
  if n = 0 then 0
  else subtract_the_first(n-1) - n;;
subtract_the_first : int -> int = <fun>
```

- b. Considerare cosa accade quando si scambia l'ordine della moltiplicazione nel calcolo del fattoriale:

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

Sperimentando con un po' di valori di n dovrebbe essere chiaro che questa versione calcola esattamente lo stesso valore della funzione fattoriale originale. Perché? Accade lo stesso se scambiamo l'ordine della somma nella funzione `sum_first`? E nella funzione `subtract_the_first`?

- c. Scambiando l'ordine della sottrazione in `subtract_the_first`, cambia il risultato restituito dalla funzione. Perché? Come descriveresti il valore restituito dalla nuova versione?

ESERCIZIO 3.5

Possiamo generalizzare la funzione `sum_first` considerando intervalli di interi piuttosto che un solo valore. Una implementazione possibile potrebbe essere:

```
# let rec sum_interval =
    fun min max ->
        if min > max then
            0
        else
            max + (sum_interval min (max-1));;
```

Riscrivere la funzione in modo tale da far crescere `min` piuttosto che decrescere `max`.

ESERCIZIO 3.6

- Scrivere una funzione ricorsiva che sommi i quadrati dei numeri compresi in un intervallo dato.
- Scrivere una funzione ricorsiva che sommi i cubi dei numeri compresi in un intervallo dato.
- Scrivere una funzione ricorsiva che prenda in input un intervallo e un numero intero non negativo p e sommi la potenza p -esima dei numeri compresi nell'intervallo dato.

Nella procedura per il calcolo del fattoriale il parametro viene decrementato di 1 ad ogni passo. A volte però l'argomento viene decrementato in modi diversi. Nel caso in cui volessimo calcolare quante cifre compongono la rappresentazione decimale di un numero intero n , dovremmo procedere così:

- Se $n < 10$ allora n è composto da una sola cifra;
- Se $n \geq 10$ allora n è composto da tante cifre quante sono le cifre che compongono $\frac{n}{10}$ più 1.

Cerchiamo di capire meglio perché dividendo il numero per 10, le cifre che compongono n diventano una di meno:

- $\frac{1234}{10} = 123$

- $\frac{12345678}{10} = 1234567$

e così via. Otteniamo dunque la funzione:

```
# let rec count_digits = fun n ->
    if n < 10 then
        1
    else
        1 + count_digits (n/10);;
count_digits : int -> int = <fun>
```

Cosa calcola la seguente funzione ricorsiva?

```
# let rec find_number = fun n ->
    if n < 10 then
        n
    else
        (
            let digit = n % 10
            in
                digit + 10 * (find_number (n/10))
        );
find_number : int -> int = <fun>
```

Nell'esempio precedente abbiamo sfruttato la possibilità di annidare il costrutto `let` per definire la variabile locale `digit`:

```
let digit = n % 10
in
    digit + 10 * (find_number (n/10))
```

Il costrutto `let` ci permette di specificare delle variabili ausiliarie (in questo caso la variabile `digit`) che ci serviranno per calcolare il valore della funzione (`digit + 10 * (find_number (n/10))`) usando come risultati “di appoggio” gli assegnamenti `let` appena precedenti.

ESERCIZIO 3.7

Scrivere una funzione ricorsiva che calcoli il numero di uni nella rappresentazione decimale di un intero. Generalizzare la funzione al calcolo del numero di occorrenze della cifra d , passata come secondo argomento.

ESERCIZIO 3.8

Scrivere una funzione ricorsiva che calcoli il numero di cifre dispari che compaiono nella rappresentazione decimale di un intero.

(suggerimento: `let odd = fun n -> n % 2 = 1`)

ESERCIZIO 3.9

Scrivere una funzione ricorsiva che sommi il numero delle cifre che compaiono nella rappresentazione decimale di un intero.

ESERCIZIO 3.10

Ogni numero intero positivo i può essere espresso come $i = 2^n \times k$, con k dispari. Chiamiamo n “esponente di 2 in i ”. Per esempio, l’esponente di 2 in 40 è 3, infatti $40 = 2^3 \times 5$, mentre l’esponente di 2 in 42 è 1. Se i è dispari, allora n è zero. Se, invece, i è pari, allora i può essere diviso per 2. Scrivere una funzione ricorsiva che calcoli l’esponente di 2 nel suo argomento.

ESERCIZIO 3.11

Si considerino le seguenti funzioni ricorsive:

```
# let rec exp = fun x n ->
    if n = 0 then 1
    else x * (exp x (n-1));;
exp : int -> int -> int = <fun>

# let rec foo = fun x n ->
    if n = 0 then 1
    else (exp x n) + (foo x (n - 1));;
foo : int -> int -> int = <fun>
```

Usare il principio di induzione per dimostrare che `(foo x n)` restituisce il valore

$$\frac{x^{n+1} - 1}{x - 1}$$

per ogni $x \neq 1$, $n \geq 0$. Non serve dimostrare che `(exp b m)` ritorna b^m . Attenzione: il passo induttivo richiede un po’ di algebra.

ESERCIZIO 3.12

Usando il principio di induzione, si dimostri che per qualsiasi intero non negativo n la seguente funzione calcola $2 \times n$:

```
# let rec f = fun n ->
    if n = 0 then 0
    else f (n-1) + 2;;
f : int -> int = <fun>
```

ESERCIZIO 3.13

Usando il principio di induzione, si dimostri che per qualsiasi intero non negativo n la seguente funzione calcola 2^{2^n} :

```
# let rec foo = fun n ->
    if n = 0 then 2
    else exp (foo (n - 1)) 2;;
foo : int -> int = <fun>
```

Suggerimento: servono alcune leggi dell’elevamento a potenza, in particolare $(2^a)^b = 2^{ab}$ e $2^a \times 2^b = 2^{a+b}$.

ESERCIZIO 3.14

Si dimostri che la seguente funzione calcola $\frac{n}{n+1}$ per qualsiasi intero non negativo n :

```
# let rec f = fun n ->
    if n = 0 then
        0.0
    else
        f (n-1) + (1.0 / float_of_int (n * (n+1)));;
f : int -> float = <fun>
```

Nota: la funzione `float_of_int` converte un valore intero nel corrispondente valore decimale. Quindi:

```
float_of_int 1 -> 1.0
float_of_int 2 -> 2.0
float_of_int 3 -> 3.0
float_of_int 4 -> 4.0
...
...
```

ESERCIZIO 3.15

Si consideri la seguente funzione ricorsiva:

```
# let rec foo = fun n ->
    if n = 0 then
        0.0
    else
        foo (n-1) + (1.0 / float_of_int (4 * n * n - 1));;
foo : int -> float = <fun>
```

- Cosa restituiscono le chiamate `(foo 1)`, `(foo 2)` e `(foo 3)`?
- Si dimostri per induzione che per ogni intero non negativo n , `(foo n)` calcola $\frac{n}{(2 \times n + 1)}$.

3.6 - I numeri primi

Vogliamo scrivere una funzione ricorsiva che determini se un numero intero positivo $n \geq 2$ è primo, utilizzando la seguente specifica:

- il numero 2 è primo,
- un numero $n > 2$ è primo se non è divisibile per alcun numero diverso da 1 e da n stesso.

Dunque, per verificare se un numero n è primo, è sufficiente controllare se esiste un numero compreso tra 2 e $n - 1$ che divide n . Possiamo iniziare provando a dividere n per 2. Se è divisibile, allora possiamo dire che n non è primo. Altrimenti, possiamo provare a dividere n per 3, e così via. Se non troviamo alcun numero compreso tra 2 e $n - 1$ che divide n , allora possiamo dire che n è primo.

Scriviamo prima una funzione intermedia, `trynext`, che verifica se un numero intero i divide n . Se questo è vero allora la funzione ritorna `false` (cioè n non è primo) altrimenti essa prova a verificare se $i + 1$ divide n . La funzione `trynext` è definita come segue:

```
# let rec trynext (i, n) =
    if (n mod i = 0) then
        false (* n is not prime *)
    else
        trynext ((i+1), n) (* try next number *);;
trynext : int * int -> bool = <fun>
```

Ora possiamo scrivere la funzione `prime` che invoca `trynext` con i parametri $(2, n)$:

```
# let prime n =
    if (n = 2) then
        true
    else
        trynext (2, n);;
prime : int -> bool = <fun>
```

Proviamo ora ad eseguire la funzione `prime`:

```
#prime 8;;
- : bool = false

#prime 13;;
- : bool = false

#prime 5;;
- : bool = false

#prime 17;;
- : bool = false
```

C'è qualcosa di sbagliato ! Dove sta l'errore? La funzione restituisce sempre `false` anche quando n è primo. L'errore sta nella funzione `trynext`: essa deve restituire `true` quando $i = n$. Possiamo dunque modificare la funzione `trynext` nel seguente modo:

```
# let rec trynext (i, n) =
    if (n = i) then true
    else if (n mod i = 0) then
        false (* n is not prime *)
    else
        trynext ((i+1), n) (* try next number *);;
trynext : int * int -> bool = <fun>
```

In modo più compatto possiamo scrivere:

```

# let prime n =
    let rec trynext i =
        (n = i) || ((n mod i != 0) & trynext(i+1))

            in      trynext 2;;
prime : int -> bool = <fun>

```

3.7 – Soluzione di alcuni esercizi

ESERCIZIO 3.4

La funzione `subtract_the_first` calcola la somma dei numeri interi compresi tra $-n$ e 0. Per esempio, se n è 3, la funzione calcola $-1 - 2 - 3 = -6$.

ESERCIZIO 3.11

Vediamo come è stata costruita la funzione ricorsiva che, dati $x \neq 1$ e $n \geq 0$, calcola $\frac{x^{n+1}-1}{x-1}$. Si usa la funzione `exp` tale che $\exp x n = x^n$.

```

# let rec exp = fun x n ->
    if n = 0 then 1
    else x * (exp x (n-1));;
exp : int -> int -> int = <fun>

# let rec foo = fun x n ->
    if n = 0 then 1
    else _____ (foo x (n - 1));;

```

Per completare l'ultima espressione, supponiamo che la chiamata `(foo x (n - 1))` sia corretta e restituisca il valore $\frac{x^n-1}{x-1}$. Osserviamo che

$$\frac{x^{n+1}-1}{x-1} - \frac{x^n-1}{x-1} = \frac{x^{n+1}-1-x^n+1}{x-1} = \frac{x^n(x-1)}{x-1} = x^n = \exp x n$$

Dunque, possiamo scrivere:

```

# let rec foo = fun x n ->
    if n = 0 then 1
    else (exp x n) + (foo x (n - 1));;
foo : int -> int -> int = <fun>

```

Dimostriamo ora la correttezza della funzione `foo`. Vogliamo dimostrare che $(foo x n)$ calcola il valore $\frac{x^{n+1}-1}{x-1}$. Dimostriamolo per induzione su n .

(Caso base) Abbiamo:

$$\begin{aligned}
& foo x 0 \\
&= \{ \text{per definizione di } foo \}
\end{aligned}$$

1

$$= \quad \{\text{per le regole dell'algebra}\}$$

$$\frac{x - 1}{x - 1}$$

$$= \quad \{\text{per le regole dell'algebra}\}$$

$$\frac{x^{0+1} - 1}{x - 1}.$$

(Passo induttivo) Sia $n > 0$. Assumiamo che $\exp x n = x^n$. Sia, per ipotesi induttiva, $\text{foo } x n = \frac{x^{n+1} - 1}{x - 1}$. Dimostriamo che $\text{foo } x (n + 1) = \frac{x^{n+2} - 1}{x - 1}$.

$\text{foo } x (n + 1)$

$$= \quad \{\text{per definizione di } \text{foo}\}$$

$$(n + 1) + n + (\text{square } n)$$

$$= \quad \{\text{per ipotesi induttiva}\}$$

$$\exp x (n + 1) + \text{foo } x n$$

$$= \quad \{\text{per la correttezza di } \exp \text{ e per l'ipotesi induttiva}\}$$

$$x^{n+1} + \frac{x^{n+1} - 1}{x - 1}$$

$$= \quad \{\text{per le regole dell'algebra}\}$$

$$\frac{x^{n+1} (x - 1) + x^{n+1} - 1}{x - 1}$$

$$= \quad \{\text{per le regole dell'algebra}\}$$

$$\frac{x^{n+2} - x^{n+1} + x^{n+1} - 1}{x - 1}$$

$$= \quad \{\text{per le regole dell'algebra}\}$$

$$\frac{x^{n+2} - 1}{x - 1}.$$

ESERCIZIO 3.12

Vediamo come è stata costruita la funzione ricorsiva f che, dato $n \geq 0$, calcola $2 \times n$.

```
# let rec f = fun n ->
    if n = 0 then 0
    else _____ f (n-1);;
```

Per completare l'ultima espressione, supponiamo che la chiamata $f(n - 1)$ sia corretta e restituisca il valore $2 \times (n - 1)$. Osserviamo che

$$f(n) - f(n - 1) = 2 \times n - 2 \times (n - 1) = 2 \times n - 2 \times n + 2 = 2.$$

Dunque, possiamo scrivere:

```
# let rec f = fun n ->
    if n = 0 then 0
    else 2 + f (n-1);;
f : int -> int = <fun>
```

Dimostriamo ora la correttezza della funzione f . Vogliamo dimostrare che $(f\ n)$ calcola il valore $2 \times n$. Dimostriamolo per induzione su n .

(Caso base) Sia $n = 0$. Abbiamo:

$$\begin{aligned} f 0 \\ = & \quad \{\text{per definizione di } f\} \\ 0 \\ = & \quad \{\text{per le regole dell'algebra}\} \\ 2 \times 0. \end{aligned}$$

(Passo induttivo) Sia $n > 0$. Assumiamo che $f\ n = 2 \times n$. Dimostriamo che $f\ (n + 1) = 2 \times n + 1$.

$$\begin{aligned} f (n + 1) \\ = & \quad \{\text{per definizione di } f\} \\ f (n) + 2 \\ = & \quad \{\text{per ipotesi induttiva}\} \\ 2 \times n + 2 \\ = & \quad \{\text{per le regole dell'algebra}\} \\ 2 \times (n + 1). \end{aligned}$$

ESERCIZIO 3.13

Dimostriamo che la seguente funzione ricorsiva calcola 2^{2^n} per ogni $n \geq 0$.

```
# let rec foo = fun n ->
    if n = 0 then 2
    else exp (foo (n-1)) 2;;
foo : int -> int = <fun>
```

Vogliamo dimostrare che $(\text{foo } n)$ calcola il valore 2^{2^n} . Dimostriamolo per induzione su n .

(Caso base) Sia $n = 0$. Abbiamo:

$$\begin{aligned} \text{foo } 0 &= \{\text{per definizione di } \text{foo}\} \\ &= 2 \\ &= \{\text{per le regole dell'algebra}\} \\ &= 2^1 \\ &= \{\text{per le regole dell'algebra}\} \\ &= 2^{2^0}. \end{aligned}$$

(Passo induttivo) Sia $n > 0$. Assumiamo che $\text{foo } n = 2^{2^n}$. Dimostriamo che $\text{foo } (n + 1) = 2^{2^{n+1}}$.

$$\begin{aligned} \text{foo } (n + 1) &= \{\text{per definizione di } \text{foo}\} \\ &= \exp(\text{foo } (n)) 2 \\ &= \{\text{per ipotesi induttiva}\} \\ &= \exp 2^{2^n} 2 \\ &= \{\text{per definizione di } \exp\} \\ &= (2^{2^n})^2 \\ &= \{\text{per le regole dell'algebra}\} \\ &= 2^{2^{n+1}} \\ &= \{\text{per le regole dell'algebra}\} \\ &= 2^{2^{n+1}}. \end{aligned}$$

ESERCIZIO 3.14

Dimostriamo che la seguente funzione ricorsiva calcola $\frac{n}{n+1}$ per ogni $n \geq 0$.

```
# let rec f = fun n ->
    if n = 0 then 0.0
    else f(n-1) + (1.0/ float_of_int (n * (n+1)));;
f : int -> float = <fun>
```

Vogliamo dimostrare che $(f\ n)$ calcola il valore $\frac{n}{n+1}$. Dimostriamolo per induzione su n .

(Caso base) Sia $n = 0$. Abbiamo:

$$\begin{aligned} f\ 0 \\ = & \quad \{\text{per definizione di } f\} \\ 0.0 \\ = & \quad \{\text{per le regole dell'algebra}\} \\ \frac{0}{1}. \end{aligned}$$

(Passo induttivo) Sia $n > 0$. Assumiamo che $f\ n = \frac{n}{n+1}$. Dimostriamo che $f\ (n+1) = \frac{n+1}{n+2}$.

$$\begin{aligned} f\ (n+1) \\ = & \quad \{\text{per definizione di } f\} \\ f\ n + 1.0 / \text{float_of_int} ((n+1) * (n+2)) \\ = & \quad \{\text{per ipotesi induttiva}\} \\ \frac{n}{n+1} + \frac{1}{(n+1) * (n+2)} \\ = & \quad \{\text{per le regole dell'algebra}\} \\ \frac{(n+2) * n + 1}{(n+1) * (n+2)} \\ = & \quad \{\text{per le regole dell'algebra}\} \\ \frac{n^2 + 2n + 1}{(n+1) * (n+2)} \\ = & \quad \{\text{per le regole dell'algebra}\} \\ \frac{(n+1)^2}{(n+1) * (n+2)} \end{aligned}$$

= {per le regole dell'algebra}

$$\frac{(n+1)}{(n+2)}.$$

ESERCIZIO 3.15

Dimostriamo che la seguente funzione ricorsiva calcola $\frac{n}{2^{*}n+1}$ per ogni $n \geq 0$.

```
# let rec foo = fun n ->
    if n = 0 then 0.0
    else foo(n-1) + (1.0/ float_of_int (4 * n * n - 1));
foo : int -> float = <fun>
```

Vogliamo dimostrare che $(foo\ n)$ calcola il valore $\frac{n}{2^{*}n+1}$. Dimostriamolo per induzione su n .

(Caso base) Sia $n = 0$. Abbiamo:

$foo\ 0$

= {per definizione di foo }

0.0

= {per le regole dell'algebra}

$$\frac{0}{1}$$

= {per le regole dell'algebra}

$$\frac{0}{2^{*}0+1}.$$

(Passo induttivo) Sia $n > 0$. Assumiamo $foo\ n = \frac{n}{2^{*}n+1}$. Dimostriamo che $foo\ (n+1) = \frac{n+1}{2^{*}(n+1)+1}$.

$foo\ (n+1)$

= {per definizione di foo }

$foo\ n + 1.0 / float_of_int (4 * (n+1) * (n+1) - 1)$

= {per ipotesi induttiva}

$$\frac{n}{2^{*}n+1} + \frac{1}{4 * (n+1) * (n+1) - 1}$$

= {per le regole dell'algebra}

$$\frac{n}{2^{*}n+1} + \frac{1}{(2^{*}n+1) * (2^{*}(n+1)+1)}$$

= {per le regole dell'algebra}

$$\begin{aligned} & \frac{(2 * n + 1)(n + 1)}{(2 * n + 1) * (2 * (n + 1) + 1)} \\ = & \qquad \qquad \{ \text{per le regole dell'algebra} \} \\ & \frac{(n + 1)}{(2 * (n + 1) + 1)}. \end{aligned}$$

Capitolo 4 – Tipi Derivati

Nei capitoli precedenti abbiamo visto alcuni modi fondamentali per comporre tipi elementari: tramite applicazione funzionale (\rightarrow) e costruzione di tuple (*). È possibile anche definire direttamente nuovi tipi.

Un tipo è un insieme di valori. Per definire un nuovo tipo occorre specificare:

- un *nome* per il tipo;
- come costruire i valori del tipo, cioè quali sono i *costruttori* del tipo.

4.1 – I Tipi Enumerati

I tipi *enumerati* sono costituiti da un insieme finito di valori. Si possono definire semplicemente elencando i valori del tipo.

Ad esempio, il tipo predefinito `bool` è un tipo enumerato.

Un nuovo tipo viene definito mediante una *dichiarazione di tipo*. Ad esempio, possiamo definire il tipo “segno” come positivo o negativo:

```
# type segno = Positivo | Negativo;;  
Type segno defined.
```

dove

- `type` è una parola chiave che introduce la dichiarazione di un nuovo tipo;
- `segno` è il nome del nuovo tipo;
- `Positivo` e `Negativo` sono i valori del nuovo tipo. Nella dichiarazione, i valori del tipo sono separati dalla barra verticale.
- I nomi dei costruttori di tipi definiti dal programmatore iniziano sempre con la lettera maiuscola.

Prima della dichiarazione, i valori `Positivo` e `Negativo` non sono conosciuti.

```
# Positivo;;  
Toplevel input:  
>Positivo;;  
>^^^^^  
The value identifier Positivo is unbound.
```

Dopo la dichiarazione:

```
# type segno = Positivo | Negativo;;  
Type segno defined.  
  
# Positivo;;  
- : segno = Positivo
```

Tali tipi di dato si dicono *enumerati*, poiché sono definiti enumerando esplicitamente i valori del tipo. I valori di un tipo enumerato sono identificati da *costanti*; questo è il modo più semplice per costruire un valore del tipo. Una costante è dunque un caso particolare di *costruttore*.

Possiamo definire funzioni che usano il nuovo tipo:

```
# let segnointero n = if n>=0 then Positivo else Negativo;;
segnointero : int -> segno = <fun>

# segnointero 42;;
- : segno = Positivo

# segnointero -12;;
- : segno = Negativo
```

Possiamo definire il prodotto tra i segni nel seguente modo:

```
# let prodotto x y = match (x,y) with
    (Positivo, Positivo) -> Positivo
    | (Positivo, Negativo) -> Negativo
    | (Negativo, Negativo) -> Positivo
    | (Negativo, Positivo) -> Negativo;;
- : prodotto : segno -> segno -> segno = <fun>
```

Definiamo ora il tipo “giorno” enumerando i possibili valori che può assumere:

```
# type giorno = Lunedi | Martedì | Mercoledì |
                Giovedì | Venerdì | Sabato | Domenica;;
Type giorno defined.
```

Si noti che anche con tali tipi è possibile utilizzare il pattern-matching:

```
# let workday x = match x with
    | Sabato -> false
    | Domenica -> false
    | _ -> true;;
workday : giorno -> bool = <fun>

# workday Sabato;;
- : bool = false

# workday Giovedì;;
- : bool = true
```

Supponiamo ora di voler dare una rappresentazione delle carte napoletane. Definiamo prima un tipo per rappresentare i semi:

```
# type seme = Bastoni | Coppe | Denari | Spade;;
Type seme defined.
```

Definiamo ora un tipo per rappresentare i valori:

```
# type valore = Asso | Due | Tre | Quattro | Cinque
           | Sei | Sette | Fante | Cavallo | Re;;
Type valore defined.
```

Una carta del mazzo è allora rappresentata mediante una coppia di tipo `valore * seme`.

```
# let settebello = (Sette, Denari);;
settebello : valore * seme = Sette, Denari

# let briscola = Spade
    in [(Asso, Denari); (Due, briscola); (Tre, Bastoni)];;
- : (valore * seme) list = [Asso, Denari; Due, Spade; Tre, Bastoni]
```

Usando il pattern matching possiamo definire la seguente funzione:

```
# let val x = match x with
  Asso -> 1
  | Due -> 2
  | Tre -> 3
  | Quattro -> 4
  | Cinque -> 5
  | Sei -> 6
  | Sette -> 7
  | Fante -> 8
  | Cavallo -> 9
  | Re -> 10;;
val : valore -> int = <fun>

# val Fante;;
- : int = 8

# (val Due) + (val Asso);;
- : int = 3
```

4.2 – I Tipi Unione (o Tipi Somma)

È possibile anche definire un tipo come unione (o *somma disgiunta*) di altri tipi. Ad esempio, possiamo definire il tipo `identifier` i cui valori possono essere:

- stringhe (nomi di individui), oppure
- interi (numero di carta d'identità).

Abbiamo quindi bisogno di un tipo contenente sia il tipo `int` che il tipo `string`. Tale tipo può essere definito nel seguente modo:

```
# type identifier = Name of string
                  | Code of int;;
Type identifier defined.
```

Le etichette `Name` e `Code` ci permettono di distinguere i vari casi all'interno di una somma. Tali etichette sono chiamate *costruttori*, poiché in realtà permettono di costruire il tipo a partire dai valori su cui è definito.

```
# Name;;
- : string -> identifier = <fun>

# Code;;
- : int -> identifier = <fun>

# Name "Giuseppe";;
- : identifier = Name "Giuseppe"

# let id1 = Name "Giuseppe";;
id1 : identifier = Name "Giuseppe"

# let id2 = Code 123456;;
id2 : identifier = Code 123456
```

`id1` e `id2` appartengono entrambi allo stesso tipo `identifier`, pur essendo costruiti a partire da due tipi diversi.

Si noti che due valori uguali possono essere distinti anche se sono costruiti a partire dagli stessi tipi. Si consideri ad esempio il seguente tipo, che rappresenta diversi modi per esprimere una “temperatura”:

```
# type temp = Celsius of float
    | Farhenheit of float
    | Kelvin of float;;
Type temp defined.

# Celsius 0.0 = Farhenheit 0.0;;
- : bool = false
```

Anche sui tipi somma è possibile utilizzare il pattern-matching: la seguente funzione, per esempio, permette di convertire una temperatura in gradi Celsius:

```
# let convCels x = match x with
    Celsius n -> Celsius n
    | Farhenheit n -> Celsius ((5.0 /. 9.0) *. (n -. 32.0))
    | Kelvin n -> Celsius (n +. 273.0);;
convCels : temp -> temp = <fun>

# convCels (Farhenheit 0.0);;
- : temp = Celsius -17.7777777778
```

Definiamo ora il tipo `number` che può essere sia un valore di tipo `int` che un valore di tipo `float` e le operazioni in esso definite come segue:

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

Type number defined.

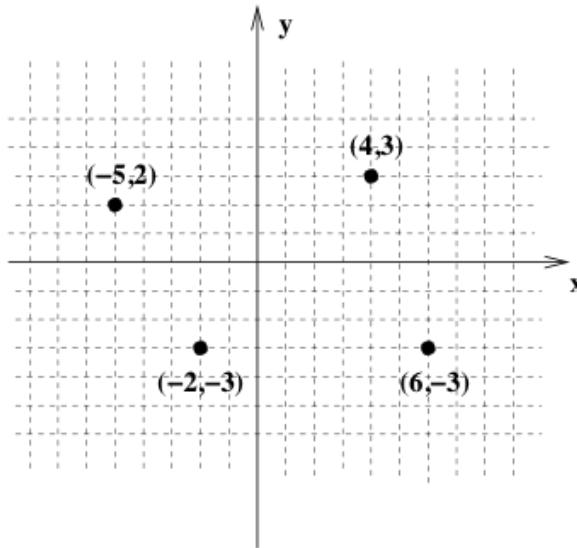
La seguente funzione definisce la somma sui valori appartenenti al tipo number:

```
# let sum (x, y) = match (x, y) with
    (Int x, Int y) -> Int (x + y)
    | (Int x, Float y) -> Float ((float_of_int x) +. y)
    | (Float x, Int y) -> Float (x +. y)
    | (Float x, Float y) -> Float (x +. (float_of_int y));;
sum : number * number -> number = <fun>

# sum (Int 3, Float 5.9);;
- : number = Float 8.9
```

4.3 – Esempio: rappresentazioni di posizioni e movimenti su un piano

Vogliamo rappresentare la posizione ed il movimento di un oggetto sul piano.



L'oggetto

- si trova in un punto del piano, identificato da una coppia (x, y) di interi;
- è rivolto in una delle quattro direzioni principali: *Su*, *Giu*, *Destra*, *Sinistra*;
- può spostarsi in avanti di n passi: in tal caso esso procede nella direzione verso cui è rivolto;
- può girare in senso orario, modificando la sua posizione di 90° .

Vogliamo risolvere il seguente problema: data la posizione (punto del piano (x, y)) e una direzione dell'oggetto e un'azione di spostamento o di cambiamento di direzione, calcolare la nuova posizione dell'oggetto. Ad esempio:

- se l'oggetto si trova nella posizione $(4, 3)$ ed è rivolto verso *Destra*, quando va avanti di 3 passi si troverà in $(7, 3)$, sempre rivolto verso *Destra*;
- se l'oggetto si trova nella posizione $(-2, -3)$ ed è rivolto verso *Giu*, quando gira si troverà sempre in $(-2, -3)$, ma rivolto verso *Sinistra*.

Un punto nel piano viene rappresentato mediante una coppia di interi.

Definiamo ora il tipo delle quattro direzioni principali

```
{Su, Giu, Destra, Sinistra}
```

La dichiarazione del nuovo tipo direzione è:

```
# type direzione = Su | Giu | Destra | Sinistra;;
Type direzione defined.
```

Una posizione è identificata da una coppia di coordinate (punto nel piano) e una direzione. Possiamo quindi rappresentare le posizioni mediante triple di tipo `int * int * direzione`, oppure definire opportunamente un nuovo tipo di dati:

```
# type posizione = Pos of int * int * direzione;;
Type posizione defined.
```

`Pos` è l'unico costruttore di valori di tipo `posizione`. Esso si comporta come una funzione:

```
# Pos;;
- : int * int * direzione -> posizione = <fun>
```

La dichiarazione del tipo `posizione` specifica che i valori di tipo `posizione` sono tutte le espressioni della forma

```
Pos (x, y, d)
```

dove `x` e `y` sono valori di tipo `int` e `d` è un valore di tipo `direzione`.

La parola chiave `of` introduce il tipo degli oggetti a cui il costruttore si applica. I costruttori sono parte della descrizione del tipo, determinano il modo canonico di descrivere i valori del tipo.

Attenzione: il tipo `posizione` non è lo stesso tipo di `int * int * direzione`:

```
# Pos(3,5,Su);;
- : posizione = Pos (3, 5, Su)

# (3,5,Su);;
- : int * int * direzione = 3, 5, Su

# (3,5,Su) = Pos(3,5,Su);;
Toplevel input:
>(3,5,Su) = Pos(3,5,Su);;
>          ^^^^^^^^^^

This expression has type posizione,
but is used with type int * int * direzione.
```

Mediante il pattern matching possiamo definire i *selettori* del tipo nel seguente modo:

```
# let xcoord (Pos(x,_,_)) = x;;
xcoord : posizione -> int = <fun>
```

```

# let ycoord (Pos(_,y,_)) = y;;
ycoord : posizione -> int = <fun>

# let dir (Pos(_,_,d)) = d;;
dir : posizione -> direzione = <fun>

# let punto (Pos(x,y,_)) = (x,y);;
punto : posizione -> int * int = <fun>

```

Ci sono due tipi di azioni:

- girare di 90° in senso orario
- andare avanti di n passi (con n intero).

Vogliamo definire un tipo di dati `azione` per rappresentare le azioni. L'azione di girare può essere rappresentata da una costante: `Gira`. L'azione di andare avanti di n passi costituisce in realtà tutto un insieme di azioni: andare avanti di 0 passi, andare avanti di 1 passo, andare avanti di 2 passi, E anche andare avanti di -1 passo (indietro di 1 passo), avanti di -2 passi (indietro di 2), Abbiamo bisogno dunque di un costruttore che, applicato ad un intero n , riporta il valore che rappresenta "andare avanti di n passi":

```

# type azione = Gira | Avanti of int;;
Type azione defined.

```

Questa dichiarazione specifica che l'insieme dei valori del tipo `azione` è costituito da:

- la costante `Gira`
- tutti i valori della forma `Avanti n`, dove n è un intero.

`Avanti` è un *costruttore*: è una funzione di tipo `int -> azione` che, quando viene applicata ad un valore di tipo `int`, restituisce (*costruisce*) un valore di tipo `azione`. I valori del tipo `azione` sono dunque:

Gira,
Avanti 0, Avanti 1, Avanti 2, Avanti 3, ...
Avanti -1, Avanti -2, Avanti -3, Avanti -4, ...

L'insieme dei valori di tipo `azione` è infinito.

Definiamo ora l'operazione principale, `sposta`, che data una posizione e una azione, determina la nuova posizione dell'oggetto sul piano. Definiamo prima la funzione `gira` che data una posizione restituisce la posizione che si ottiene girando la direzione di 90° in senso orario.

```

# let gira d = match d with
  Su -> Destra
  | Giu -> Sinistra
  | Destra -> Giu
  | Sinistra -> Su;;
gira : direzione -> direzione = <fun>

```

Si noti che il nome della funzione `gira` inizia con la lettera minuscola: esso è diverso dal nome della costante `Gira`.

Definiamo ora la funzione `avanti` che, applicata a una posizione `Pos(x, y, d)` e un intero `n`, a seconda della direzione `d`, incrementa o decrementa `x` o `y`:

- se `d` è `Su`: da `(x, y)` si passa a `(x, y+n)`
- se `d` è `Giu`: da `(x, y)` si passa a `(x, y-n)`
- se `d` è `Destra`: da `(x, y)` si passa a `(x+n, y)`
- se `d` è `Sinistra`: da `(x, y)` si passa a `(x-n, y)`

Otteniamo dunque la seguente definizione della funzione `avanti`:

```
# let avanti (Pos(x,y,d), n) = match d with
    Su -> Pos(x,y+n,d)
    | Giu -> Pos(x,y-n,d)
    | Destra -> Pos(x+n,y,d)
    | Sinistra -> Pos(x-n,y,d);;
avanti : posizione * int -> posizione = <fun>
```

Possiamo dunque definire la funzione `sposta`:

```
# let sposta (Pos(x,y,d), act) =
    match act with
        Gira -> Pos(x,y,gira d)
        | Avanti n -> avanti (Pos(x,y,d), n);;
sposta : posizione * azione -> posizione = <fun>

# sposta (Pos(4,3,Destra), Avanti 3);;
- : posizione = Pos (7, 3, Destra)

# sposta (Pos(-2,-3,Giu), Gira);;
- : posizione = Pos (-2, -3, Sinistra)
```

4.4 – I Tipi Record (o Tipi Prodotto)

Un *record* è costituito da un numero finito di *campi*, di tipo non necessariamente omogeneo, ciascuno dei quali è identificato da un'*etichetta* e contiene un valore di un tipo determinato. A differenza delle tuple, le componenti di un record (i campi) sono distinti mediante i nomi (le etichette), anziché per la posizione.

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.

Definiamo il tipo `person` nel seguente modo:

```
# type person = {name: string; age: int; job: string};;
Type person defined.
```

Possiamo dunque dichiarare espressioni di tipo record come nell'esempio seguente:

```
# let Mario = {name = "Mario"; job = "student"; age = 21};;
Mario : person = {name = "Mario"; age = 21; job = "student"}
```

Possiamo definire la funzione `age_of` che restituisce l'età di una persona:

```
# let age_of = fun person -> person.age;;
age_of : person -> int = <fun>

# age_of Mario;;
- : int = 21

# Mario.age;;
- : int = 21

# Mario.name;;
- : string = "Mario"
```

Alternativamente, possiamo definire la funzione `age_of` nel seguente modo:

```
# let age_of = fun {name=_; job=_; age = x} -> x;;
age_of : person -> int = <fun>
```

Capitolo 5 – Iterazione e Invarianti

5.1 – Iterazione

Nel capitolo precedente abbiamo introdotto una strategia generale di risoluzione dei problemi: la **ricorsione**. Essa consiste nel risolvere prima un sottoproblema uguale a quello iniziale ma più semplice e quindi eseguire il passo che resta. Ora ci occuperemo di una strategia un po' diversa, nota come **iterazione**:

LA STRATEGIA ITERATIVA: con poche operazioni iniziali trasforma il problema in uno più semplice ma con la stessa soluzione del problema iniziale. Quindi risovi il problema più semplice.

Immaginiamo di ascoltare uno studente che pensa ad alta voce mentre sviluppa una strategia alternativa per costruire la funzione fattoriale:

- Devo calcolare un fattoriale, per esempio: $6 \times 5 \times 4 \times 3 \times 2 \times 1$
- Non potrei trasformarlo in un problema con la stessa risposta ma più piccolo? Sarebbe una strategia più semplice no? Il problema è che ho sei numeri da moltiplicare. Passare a cinque numeri sarebbe meglio. Dove posso trovare cinque numeri che moltiplicati tra loro diano lo stesso risultato del fattoriale iniziale?

$$6 \times 5 \times 4 \times 3 \times 2 \times 1 = \underline{\quad} \times \underline{\quad} \times \underline{\quad} \times \underline{\quad} \times \underline{\quad}$$

- Poiché a sinistra ho un numero di troppo rispetto agli spazi da riempire a destra, sarebbe ottimo poter mettere due numeri della sequenza originale in uno spazio unico. Ad esempio potrei moltiplicarli tra loro!

$$\underline{(6 \times 5)} \times 4 \times 3 \times 2 \times 1 = \underline{30} \times 4 \times 3 \times 2 \times 1$$

- Adesso potrei ridurre la sequenza di un altro numero:

$$\dots = 120 \times 3 \times 2 \times 1$$

- E così via fino a un numero unico:

$$\dots = 360 \times 2 \times 1$$

$$\dots = 720 \times 1$$

$$\dots = 720$$

- L'ultimo passo potremmo chiamarlo "problema con zero moltiplicazioni": esso costituisce la risposta al problema originale, in quanto la sua soluzione è uguale a quella di tutti i passi precedenti compreso il fattoriale iniziale!

Supponiamo ora di voler scrivere una funzione che risolva il problema del fattoriale in questo modo. Da un lato potremmo pensare di passare alla funzione tutti i numeri da moltiplicare, ma sarebbe scomodo: e se fossero veramente tanti? Considerando che i numeri successivi al primo sono tutti consecutivi tra loro, per indicarli tutti mi basta sapere qual'è il primo. Quindi la descrizione del problema assomiglia a "*30 moltiplicato per i numeri da 4 a 1*" o "*120 moltiplicato per i numeri da 3 a 1*" e così via. In effetti l'idea di moltiplicare tra loro i numeri "*da n a 1*" è proprio la definizione del fattoriale, quindi sto cercando una funzione per moltiplicare un qualche numero con un qualche fattoriale:

```
# let rec factorial_product = (*calcola a×n!*)
  fun a n ->
    if n = 0 then a
    else
      factorial_product (a * n) (n-1) (*a×n! = (a×n)×(n-1)!*)
factorial_product : int -> int -> int = <fun>
```

NOTA - COMMENTI AL CODICE: in questa procedura vediamo che il nostro studente ha commentato sia la funzione vera e propria, descrivendo ad un ipotetico altro utente cosa questa faccia, che il corpo della funzione per chiarire come la funzione calcola il suo valore finale. Questo modo di commentare il codice è estremamente utile e consigliamo caldamente di abituarsi fin da subito ad impiegarlo.

Nella funzione sopra osserviamo che

- se n (il numero di cui si vuole calcolare il fattoriale) è uguale a 0, allora $a \times n! = a \times 0! = a \times 1 = a$,
- altrimenti $a \times n! = (a \times n) \times (n - 1)!$ è il passo ricorsivo.

Notiamo che il problema del fattoriale che abbiamo affrontato nel capitolo precedente in realtà altro non è che il problema appena visto in cui al primo parametro passiamo il valore 1:

```
# let factorial = fun n -> factorial_product 1 n;;
factorial : int -> int = <fun>
```

Spesso, nel risolvere un problema, può essere utile cercare di generalizzarlo il più possibile. Infatti risolvere la versione più generale del problema potrebbe persino rendere la soluzione del problema di partenza più semplice.

ESERCIZIO 5.1

Nel costruire la funzione del fattoriale iterativo, abbiamo scelto di moltiplicare tra loro i primi due numeri per ridurre il numero di moltiplicazioni da eseguire. Avremmo potuto fare scelte differenti come ad esempio moltiplicare i due numeri più a destra. Provare a ridefinire la funzione `factorial_product` in questo senso.

La funzione iterativa del fattoriale può sembrare non molto diversa da quella ricorsiva. In entrambe le funzioni le moltiplicazioni sono eseguite una alla volta. Una delle moltiplicazioni viene eseguita esplicitamente, mentre le altre sono “nascoste” nella chiamata della funzione a se stessa. C’è però una distinzione molto importante: la funzione ricorsiva tiene il problema corrente in attesa della soluzione del sottoproblema mentre quella iterativa riduce progressivamente il problema.

L’approccio ricorsivo è meno efficiente perché è necessario mantenere in memoria i dati dei problemi in attesa finché i rispettivi sottoproblemi non vengono risolti. Poiché un sottoproblema, la cui soluzione è attesa dal problema che lo ha generato, attende a sua volta la soluzione dei suoi sotto-sottoproblemi, la memoria richiesta dall’approccio ricorsivo aumenta man mano che si scende di livello nella risoluzione dei sotto-problemi. Questo è illustrato nel diagramma della Figura 3.1: ogni colonna rappresenta un sottoproblema; così come il diagramma cresce in larghezza al crescere del parametro passato originariamente alla procedura, nello stesso modo cresce lo spazio di memoria occupato dal processo di calcolo. Si può dire di più: la procedura di calcolo impiega al più tanta memoria quanto è il massimo numero di colonne attive contemporaneamente.

Al contrario, la funzione iterativa sta sempre e comunque risolvendo un singolo problema. Il problema viene ogni volta trasformato in uno più semplice ma con la stessa soluzione: non abbiamo mai un problema in attesa di un sottoproblema perché finita la trasformazione nel problema più semplice si passa direttamente a risolvere il nuovo problema e si abbandona il livello precedente. La quantità di memoria richiesta è esattamente la stessa per ogni livello di risoluzione indipendentemente dalla dimensione del problema originale. Nel diagramma in Figura 5.1 vediamo come nel processo iterativo di valutare (`factorial 3`) la computazione resta su una sola colonna (come al solito abbiamo omesso alcuni dettagli inutili). Anche se avessimo valutato (`factorial 52`) ci sarebbe bastata una sola colonna! Ciononostante il diagramma sarebbe

risultato più alto: si noti che la larghezza del diagramma corrisponde alla memoria richiesta, mentre la sua altezza corrisponde al tempo necessario per completare il calcolo.

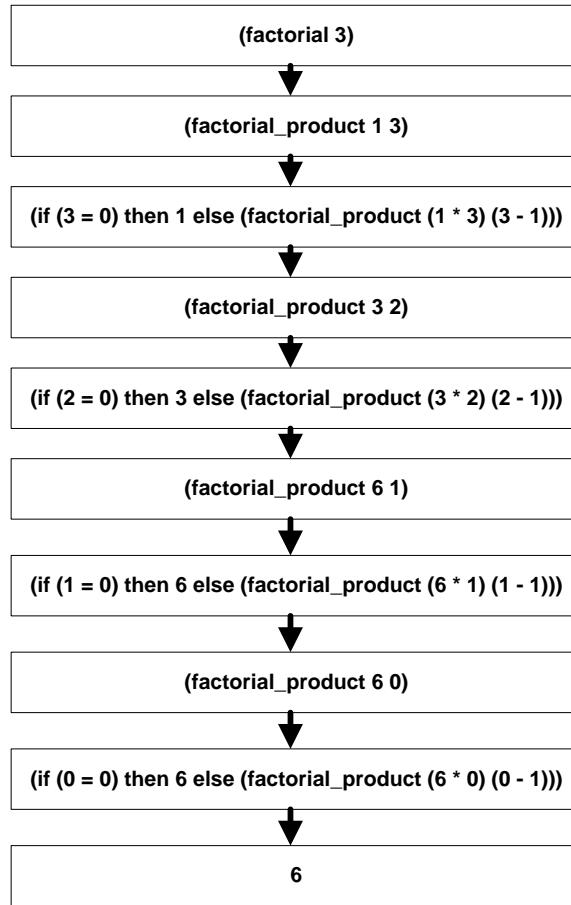


Figura 5.1

La natura iterativa del processo di calcolo si può vedere anche dalla seguente descrizione più compatta:

```

(factorial 3)
(factorial_product 1 3)
(if (3 = 0) then 1 else (factorial_product (1 * 3) (3 - 1)))
(factorial_product (1 * 3) (3 - 1))
(factorial_product 3 2)
(if (2 = 0) then 3 else (factorial_product (3 * 2) (2 - 1)))
(factorial_product (3 * 2) (2 - 1))
(factorial_product 6 1)
(if (1 = 0) then 6 else (factorial_product (6 * 1) (1 - 1)))
(factorial_product (6 * 1) (1 - 1))
(factorial_product 6 0)
(if (0 = 0) then 6 else (factorial_product (6 * 0) (0 - 1)))
6
  
```

Se calcoliamo (factorial 6), tralasciando i passi che comportano lo svolgimento degli `if` e dell'aritmetica, lo schema generale di esecuzione che otteniamo è sostanzialmente identico al calcolo di 6! fatto dal nostro ipotetico studente all'inizio del capitolo:

```
(factorial 6)           6!
(factorial_product 1 6) = 1 * 6!
(factorial_product 6 5) = 6 * 5!
(factorial_product 30 4) = 30 * 4!
(factorial_product 120 3) = 120 * 3!
(factorial_product 360 2) = 360 * 2!
(factorial_product 720 1) = 720 * 1!
(factorial_product 720 0) = 720 * 0!
```

Una piccola fonte di confusione è che le funzioni come quelle che abbiamo visto in questo capitolo continuano a chiamarsi ricorsive, poiché si auto-invocano. Pur chiamandosi ricorsive non generano processi ricorsivi. Una funzione ricorsiva è una qualsiasi funzione che invochi se stessa (anche indirettamente). Se la auto-invocazione è per risolvere un sottoproblema la cui soluzione non è la stessa del problema principale, allora il processo computazionale risultante è ricorsivo come negli esempi del Capitolo 3. Se invece l'auto-invocazione serve a risolvere una versione ridotta del problema originale (una versione che richiede meno calcoli ma che darà la stessa risposta del problema di partenza), allora il processo è iterativo come quelli che vediamo in questo capitolo.

Si ha iterazione quando la chiamata ricorsiva è l'ultima istruzione eseguita prima di terminare la funzione, per questo motivo si dice anche che si ha *tail recursion* (o *ricorsione in coda*). Anche se formalmente ricorsiva, la tail recursion dà luogo ad un processo computazionale di tipo iterativo.

5.2 – Invarianti

Il commento che troviamo in testa alla funzione `factorial_product` è molto utile perché ci descrive un'**invariante**. L'invariante di una funzione è una quantità all'interno della funzione stessa che non cambia mai. L'invariante della funzione `factorial_product` è il valore che essa ritorna: $a \times n!$ e questo sarà sempre vero, indipendentemente dai valori passati alla procedura o dal punto del processo di calcolo in cui ci troviamo. Possiamo ragionare su come questo sia vero osservando che alcuni parametri si abbassano per andare verso il caso base (il parametro n che viene decrementato via via fino a 1), mentre altri parametri cambiano per compensazione (in questo caso il parametro a che viene moltiplicato per i valori di n che "scartiamo"). In questa sezione vedremo come gli invarianti possono essere usati per scrivere funzioni iterative e dimostrare che tali funzioni sono corrette.

Cominciamo proprio con la funzione `factorial_product`: vogliamo dimostrare che tale funzione è corretta, ossia calcola effettivamente $a \times n!$, quando n è un intero positivo (da notare come fin da subito ci concentriamo sull'invariante).

Dimostrazione per induzione matematica su n :

Caso base: Se $n = 0$, per definizione di `factorial_product` otteniamo che la funzione termina restituendo il valore a , infatti $a \times 0! = a \times 1 = a$.

Ipotesi induttiva: Sia $n \geq 0$. Assumiamo che (`factorial_product a n`) termini con il valore $a \times n!$ per ogni numero intero a .

Passo induttivo: Si consideri la valutazione di (`factorial_product a n+1`). Vogliamo dimostrare che essa termina con il valore $a \times (n + 1)!$. Infatti:

```

factorial_product a n + 1
= {per definizione di factorial_product}
factorial_product (a × (n + 1)) n
= {per ipotesi induttiva}
a × (n + 1) × n!
= {per le regole dell'algebra}
a × (n + 1)!

```

Conclusione: Per induzione matematica la valutazione di (factorial_product a n) terminerà restituendo il valore $a \times n!$ per qualsiasi intero non negativo n e qualsiasi intero a .

Ora che abbiamo visto tutta la dimostrazione per induzione, è interessante tornare indietro e guardare i commenti che il nostro ipotetico studente d'inizio capitolo ha inserito nel corpo della funzione. Abbiamo già visto che il commento principale dichiara che la funzione ha $a \times n!$ come invariante, che è quello che abbiamo appena dimostrato. Nel corpo della funzione però c'è un altro commento che dice che essa calcola $a \times n! = (a \times n) \times (n - 1)!$: ciò corrisponde all'applicazione dell'ipotesi induttiva alla chiamata ricorsiva.

Adesso proviamo a scrivere una versione iterativa della funzione per l'elevamento a potenza che abbiamo visto nel Capitolo 3. Calcolare una potenza richiede varie moltiplicazioni, ed è in qualche modo simile al calcolo di un fattoriale. Per questo motivo definiremo le funzioni power e power_product in modo molto simile a come abbiamo fatto per il fattoriale e useremo anche un invariante analogo:

```

# let rec power_product =
  fun a b e -> (* calcola a per b elevato alla e*)
    if e = 0 then _____
    else power_product _____;
;

# let power = fun b e -> power_product 1 b e;;

```

Per poter dimostrare che se $e = 0$ allora la funzione restituisce $a \times b^0$, dobbiamo scrivere a nel primo spazio vuoto della funzione power_product. Per quanto riguarda gli spazi restanti: dobbiamo mettere nell'ultimo un numero intero non negativo strettamente minore di e . Sappiamo che $e > 0$, altrimenti saremmo entrati nel primo ramo del costrutto if, quindi $e - 1$ rispetta tutte le proprietà di cui abbiamo bisogno! Chiaramente poi la base b va nello spazio di mezzo: in questo modo continuiamo a calcolare potenze dello stesso numero. Per riempire l'ultimo spazio viene utile riflettere sull'invariante che abbiamo indicato nel commento. Ipotizziamo di scrivere una qualche espressione x al posto dell'ultimo spazio vuoto. Allora la chiamata ricorsiva calcolerà $x \times b^{e-1}$, dunque:

$$x \times b^{e-1} = a \times b^e$$

$$x = \frac{a \times b^e}{b^{e-1}}$$

$$x = a \times b$$

La definizione della funzione risulta quindi essere:

```
# let rec power_product =
  fun a b e -> (* calcola a per b elevato alla e *)
    if e = 0 then a
    else power_product (a*b) b (e-1);;
power_product : int -> int -> int -> int = <fun>

# let power = fun b e -> power_product 1 b e;;
power : int -> int -> int = <fun>
```

ESERCIZIO 5.2

Usare il principio di induzione matematica per dimostrare la correttezza della funzione `power_product`.

Come ultimo esempio scriveremo una funzione che un matematico del sedicesimo secolo, Pierre Fermat, credeva avrebbe restituito tutti i numeri primi. Un numero primo è un numero naturale che ha solo due divisori: 1 e sé stesso. Fermat pensava che tutti i numeri ottenuti elevando 2 al quadrato un certo numero di volte e sommando 1 fossero primi. Si può infatti vedere che i numeri $2^{2^0} + 1 = 3$, $2^{2^1} + 1 = 5$, $2^{2^2} + 1 = 17$, $2^{2^3} + 1 = 257$ e (con un po' più di sforzo) $2^{2^4} + 1 = 65537$ sono primi. Sfortunatamente il numero di Fermat $2^{2^5} + 1 = 4294967297$ non è primo, in quanto $4294967297 = 641 \times 6700417$.

Trasliamo la definizione di numero di Fermat in una funzione:

```
# let rec repeatedly_square =
  fun b n -> (* calcola b elevato al quadrato n volte, con n ≥ 0 *)
    if n = 0 then b
    else repeatedly_square ____ ____
```

Come riempiamo gli spazi vuoti? Come abbiamo fatto nel caso della funzione `power_product`, per applicare l'ipotesi induttiva l'ultimo spazio vuoto deve avere un intero non negativo minore di n , e quindi proviamo con $n - 1$. Qualsiasi cosa mettiamo nello spazio restante, sappiamo dall'ipotesi induttiva che verrà elevato al quadrato $n - 1$ volte. La domanda allora è: cosa, elevato al quadrato $n - 1$ volte, produce il risultato desiderato (b elevato al quadrato n volte)? La risposta a questa domanda è naturalmente b^2 ! La procedura finale risulta essere:

```
# let rec repeatedly_square =
  fun b n -> (* calcola b elevato al quadrato n volte, con n ≥ 0 *)
    if n = 0 then b
    else repeatedly_square (b*b) (n-1);;
repeatedly_square : int -> int -> int = <fun>
```

Si osservi che ci siamo preoccupati di elevare al quadrato b solo la prima volta, e ci siamo fidati dell'ipotesi induttiva secondo la quale il numero così ottenuto verrà elevato al quadrato le rimanenti $n - 1$ volte.

5.3 – Numeri perfetti e definizioni annidate

Sviluppiamo ora un esempio un po' più esteso che sfrutta l'iterazione. Un numero si dice **perfetto** se la somma dei suoi divisori è uguale al doppio del numero stesso, ovvero se esso vale quanto la

somma dei suoi divisori escluso lui stesso. Scriviamo una semplice funzione `perfect` che restituisce `true` se un numero è perfetto e altrimenti restituisce `false`.

```
# let perfect = fun n -> (sum_of_divisors n) = (n + n);;
```

Il modo più semplice per calcolare la somma dei divisori di n è quello di scorrere tutti i numeri da 1 a n , e sommare quelli che dividono n . Una simile computazione ha come invariante la somma dei divisori di n nell'intervallo che resta da considerare più la somma accumulata fino a quel momento. La funzione risultante è:

```
# let sum_of_divisors =
  fun n -> (* somma i divisori di n*)
    let rec sum_from_plus =
      fun low addend -> (* somma i divisori di n compresi tra low e
n più il valore di addend *)
        if low > n then addend
        else if (n mod low = 0)
          then sum_from_plus (low + 1) (addend + low)
          else sum_from_plus (low + 1) addend
    in sum_from_plus 1 0;;
sum_of_divisors : int -> int = <fun>
```

Dalla definizione precedente vediamo una caratteristica molto utile del linguaggio: è possibile annidare una definizione all'interno di una espressione. In questo modo:

- Il nome definito internamente è privato al corpo della funzione in cui appare. Quindi non si può invocare la funzione `sum_from_plus` se non all'interno di `sum_of_divisors`.
- La funzione `sum_from_plus` può accedere al valore del parametro n pur senza averlo dichiarato, in virtù del fatto che tale parametro compare nella funzione in cui essa è annidata. Le funzioni annidate possono fare uso di qualsiasi nome definito in qualsiasi funzione entro cui si trovino.

La ragione per cui non abbiamo annidato la funzione `sum_of_divisors` dentro la funzione `perfect` è che potrebbe venirci utile anche in altri contesti, mentre invece la funzione `sum_from_plus` ha uno scopo molto limitato e offrirebbe ben poco all'esterno del contesto in cui l'abbiamo collocata.

ESERCIZIO 5.3

Anche se il metodo che abbiamo usato per il calcolo della somma dei divisori è molto immediato, non è particolarmente efficiente. Infatti ogni volta che troviamo un divisore d di n , possiamo dedurre che anche $\frac{n}{d}$ è un divisore. In particolare, qualsiasi divisore maggiore di \sqrt{n} può essere dedotto dai divisori minori della radice di n . Usare questa osservazione per scrivere una versione più efficiente della procedura in questione che si ferma quando $low^2 \geq n$, ricordando che se $low^2 = n$, allora low e $\frac{n}{low}$ sono in realtà lo stesso divisore che va contato una, non due volte!

Prima di testare la nostra procedura, poiché i numeri perfetti sono rari e piuttosto distanziati tra loro, ci conviene ottimizzare ulteriormente la ricerca. La procedura seguente trova il primo numero perfetto a partire da un certo valore n :

```
# let rec first_perfect_after = fun n ->
  if perfect (n+1) then (n+1)
  else first_perfect_after (n+1);;
first_perfect_after : int -> int = <fun>
```

In questo modo possiamo cercare i numeri perfetti consecutivi usando come parametro il valore ritornato alla chiamata precedente:

```
# first_perfect_after 0;;
- : int = 6

# first_perfect_after 6;;
- : int = 28

# first_perfect_after 28;;
- : int = 496

# first_perfect_after 496;;
- : int = 8128
```

La nostra definizione di `first_perfect_after` è molto poco elegante, poiché l'espressione `(n+1)` compare ben tre volte nel codice. Possiamo usare le definizioni annidate per rendere tutto più elegante (e un pochino più efficiente) così:

```
# let rec first_perfect_after =
  fun n ->
    let next = (n+1)
    in if perfect next then
        next
      else
        first_perfect_after next;;
first_perfect_after : int -> int = <fun>
```

In questa seconda versione diamo un nome all'espressione `(n+1)` (che contestualmente precalcoliamo) e dove avevamo una ripetizione inseriamo ora il nome assegnato all'espressione invece che ripeterla.

5.4 – Altri esempi

Definire una funzione `sum` che, applicata a due interi positivi, ne determini la somma con ricorsione generica e con ricorsione in coda.

```
# let rec sum (m, n) = (* funzione ricorsiva *)
  match n with
    0 -> m
  | _ -> 1+(sum (m, n-1));;
sum : int * int -> int = <fun>

# sum (12,5);;
- : int = 17

# let rec sum (m, n) = (* funzione iterativa *)
  match n with
    0 -> m
  | _ -> sum (m+1, n-1);;
sum : int * int -> int = <fun>

# sum (3,45);;
- : int = 48
```

Definire una funzione `product` che, applicata a due interi positivi, ne determini il prodotto con ricorsione in coda.

```
# let product (a,b) =
  match (a,b) with
  (0,_) | (_,0) -> 0
  | (1,_) -> b
  | (_,1) -> a
  | _ ->
    let rec aux_product (m, n, acc) = (* calcola m×n+acc *)
      match n with
      | 0 -> acc
      | _ -> aux_product (m, n-1, acc+m)
    in aux_product (a, b, 0);;
product : int * int -> int = <fun>

#product(3,7);;
- : int = 21
```

Si consideri il problema di determinare il numero di divisori di un intero positivo n : si possono scandire i numeri da 1 a n e, per ciascuno di essi, si determina se divide n ; in caso positivo, si incrementa un contatore (inizializzato a 0). Si implementi questo algoritmo, utilizzando una funzione ausiliaria che svolga il ruolo dell'iterazione.

Soluzione 1.

```
# let rec f (n, k, acc) = (* calcola il numero dei divisori di n
maggiori o uguali a k più il valore di acc*)
  if (k > n) then acc
  else if ((n mod k) = 0) then f(n, k+1, acc+1)
    else f(n, k+1, acc);;
f : int * int * int -> int = <fun>

# let divisori n = f(n, 1, 0);;
divisori : int -> int = <fun>

#divisori (10);;
- : int = 4
```

Soluzione 2.

```
# let rec f (n, k, acc) = (* calcola il numero dei divisori di n
minori o uguali a k più il valore di acc*)
  if (k = 1) then acc+1
  else if ((n mod k) = 0) then f(n, k-1, acc+1)
    else f(n, k-1, acc);;
f : int * int * int -> int = <fun>

# let divisori n = f(n, n, 0);;
divisori : int -> int = <fun>

#divisori (12);;
- : int = 6
```

Si consideri il problema dell'esercizio precedente. In questo caso si vuol stampare la lista dei divisori.

```
# let rec f (n, k, acc) =
    if (k > n) then acc
    else if ((n mod k) = 0)
        then f(n, k+1, acc^string_of_int(k)^" ")
        else f(n, k+1, acc);;
f : int * int * string -> string = <fun>

# let divisori n = f(n, 1, " ");;
divisori : int -> string = <fun>

#divisori (20);;
- : string = " 1 2 4 5 10 20 "
```

ESERCIZIO 5.4

Un numero intero positivo può essere espresso come $2^j \times k$ dove k è dispari. La seguente funzione, dato un numero $n > 0$, calcola il valore k , cioè il fattore dispari (il più grande divisore dispari di n). Questa funzione è ricorsiva o iterativa? Perché?

```
# let rec largest_odd_divisor =
  fun n ->
    if n % 2 = 1 then n
    else largest_odd_divisor (n/2);;
```

ESERCIZIO 5.5

La seguente funzione, dati due numeri interi b e n tali che $n \geq 1$ e $b \geq 2$, calcola il più grande numero k tale che $b^k \leq n$.

```
# let rec closest_power =
  fun b n ->
    if n < b then 0
    else 1 + (closest_power b (n / b));;
```

- Spiegare perché questa funzione genera un processo ricorsivo.
- Scrivere una versione iterativa.

ESERCIZIO 5.6

Si considerino le due funzioni seguenti:

```
# let rec f =
  fun n ->
    if n = 0 then 0
    else g (n-1)
and g =
  fun n ->
    if n = 0 then 1
    else f (n - 1);;
```

- Usare il modello di sostituzione per valutare $(f\ 1)$, $(f\ 2)$, e $(f\ 3)$.
- È possibile prevedere cosa ritornerà $(f\ 4)$? E $(f\ 5)$? In generale, per quali argomenti la funzione f restituisce 0 piuttosto che 1?
- Il processo generato dalla funzione f è iterativo o ricorsivo? Perché?

ESERCIZIO 5.7

Si considerino le due funzioni seguenti:

```
# let rec f =
  fun n ->
    if n = 0 then 0
    else (1 + (g (n - 1)))
and g =
  fun n ->
    if n = 0 then 1
    else (1 + (f (n - 1));;
```

- a) Usare il modello di sostituzione per valutare $(f\ 2)$, $(f\ 3)$, e $(f\ 4)$.
- b) È possibile prevedere cosa ritornerà $(f\ 5)$? E $(f\ 6)$?
- c) Il processo generato dalla funzione f è iterativo o ricorsivo? Perché?

ESERCIZIO 5.8

Scrivere una funzione iterativa che dati due numeri interi n e k con $n > 1$ e $k \geq 0$ calcoli il numero n_k che denota la moltiplicazione dei primi k numeri consecutivi a partire da n . Ad esempio $7_3 = 7 \times 6 \times 5$ e $7_0 = 1$.

ESERCIZIO 5.9

Abbiamo già visto come si può elevare un numero ad un esponente intero purché questo non sia negativo. Possiamo estendere la nostra definizione agli esponenti negativi nel seguente modo:

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b^{n-1} \times b & \text{if } n > 0 \\ \frac{b^{n+1}}{b} & \text{if } n < 0 \end{cases}$$

- a) Sfruttando questa idea scrivere una funzione che elevi un numero ad un esponente intero.
- b) Mostrare la valutazione di $(\text{power}\ 2.0\ -3)$ tramite il modello di sostituzione.
- c) La funzione scritta genera un processo ricorsivo o iterativo?

ESERCIZIO 5.10

Si considerino le due funzioni seguenti:

```
# let rec product =
  fun low high ->
    if low > high then 1
    else low * (product (low + 1) high);;

# let factorial =
  fun n -> product 1 n;;
```

- a) Mostrare tramite il modello di sostituzione la valutazione di $(\text{factorial}\ 4)$.
- b) Il processo generato è ricorsivo o iterativo? Perché?
- c) Qual'è l'invariante della procedura product ?

ESERCIZIO 5.11

Dimostrare che, per ogni intero non-negativo n e per ogni numero a , la seguente funzione calcola $2^n \times a$.

```
# let rec foo =
  fun n a ->
    if n = 0 then a
    else foo (n - 1) (a + a);;
```

Capitolo 6 – Ordine di crescita e ricorsione ad albero

6.1 – Ordine di crescita

Quando nei capitoli precedenti abbiamo creato dei programmi, ci siamo preoccupati di un solo aspetto: che il processo generato dal programma producesse il risultato corretto. Nonostante questo sia ovviamente importante, ci sono alcune considerazioni da fare quando si scrive un nuovo programma. Se paragoniamo il nostro lavoro a quello di un aspirante designer di automobili, possiamo dire che ora sappiamo come creare macchine che ci portino dal punto A al punto B. Questo è importante, ma i clienti vogliono anche altro. In questo capitolo faremo alcune considerazioni che possiamo paragonare alle problematiche relative alla velocità e al consumo di gas delle automobili.

Confrontare la velocità di esecuzione di due programmi distinti che risolvono lo stesso problema dovrebbe essere facile. Fai partire il cronometro, vedi quanto ci mette il primo, poi quanto ci mette il secondo. Non c'è niente da dire: uno vincerà, l'altro perderà. Questo approccio ha però diverse limitazioni:

1. Non può essere usato per decidere quale dei due programmi eseguire, per il semplice fatto che richiede di eseguirli entrambi. Inoltre, non si può sapere a priori se uno dei due programmi ci metterà milioni di anni (e quindi non ha senso stare ad aspettarlo), mentre l'altro ci metterà un giorno (quindi se si è così pazienti da aspettare, almeno si otterrà il risultato).
2. Non ci fornisce informazioni su altre istanze dello stesso problema generale: magari il programma A calcola $52!$ in 1 millisecondo, mentre il programma B ci mette 5 millisecondi. Ora vogliamo però calcolare $100!$: quale dei due programmi dovremmo usare? Forse A, forse B: a volte un programma che è veloce su problemi piccoli è più lento su problemi grandi, come un velocista che ottiene risultati mediocri nelle corse a lunga distanza.
3. Non distingue tra differenze di prestazioni dovute al particolare hardware utilizzato, all'implementazione in CaML piuttosto che in F#, o ad altri dettagli di programmazione da differenze dovute a ragioni ben più profonde e radicate nella strategia di risoluzione del problema.

Gli informatici usano diverse tecniche per affrontare questi problemi, ma la principale è la seguente:

LA VISIONE ASINTOTICA: Non ci si chiede quale programma è più lento, ma piuttosto quale diventa lento più velocemente mano a mano che la dimensione del problema aumenta.

Questa idea è molto difficile da capire. Abbiamo tutti più confidenza con il concetto di “essere lenti”, piuttosto che con il concetto di qualcosa che “diventa lento sempre più velocemente”. Di seguito abbiamo sviluppato un esperimento che vi farà avere almeno un'intuizione di un programma che diventa lento sempre più velocemente.

L'idea di questo esperimento è di confrontare le velocità di due metodi diversi per ordinare un mazzo di carte numerate. Per capire bene quale dei due metodi diventa lento più velocemente, ordinerai tu stesso mazzi di diverse dimensioni. Prima di cominciare, ti servirà un mazzo di 32 carte numerate. Chiedi ad un compagno, collega, amico o gentile sconosciuto di lavorare con te, perché cronometrare l'ordinamento è più facile se si è in due. Uno dei due deve effettivamente ordinare le carte, mentre l'altro tiene traccia delle

regole del metodo di ordinamento, fornisce suggerimenti se necessario e fa notare eventuali errori. La seconda persona è anche incaricata di tenere traccia del tempo di ogni ordinamento (un cronometro viene decisamente utile).

I due metodi di ordinamento, o *algoritmi* di ordinamento, sono descritti nelle schede che seguono. (La parola *algoritmo* è essenzialmente un sinonimo di *procedura* o *metodo*, con però la fondamentale distinzione che si può chiamare algoritmo solamente una procedura che termina. Un algoritmo è un metodo di calcolo generico, indipendente dalla sua implementazione in un linguaggio di programmazione. Questo distingue gli algoritmi dai programmi.) Prima di cominciare, assicurati che sia tu che il tuo compagno abbiate capito perfettamente questi due algoritmi.

Quando avete capito i due algoritmi, create un mazzo di 4 carte mescolando tutte le 32 carte per bene e prendendo poi solo le prime 4. Ordinatele usando l'algoritmo *selection sort*, tenendo traccia di quanto tempo ci mettete. Poi ripetete il procedimento usando un mazzo di 8 carte, poi di 16, e infine di 32. Fate attenzione a rimescolarle per bene ogni volta. Infine, ripetete tutto (ordinate mazzi di 4, 8, 16, 32 carte) usando l'algoritmo di *merge sort*.

Scheda: Selection sort

Dovrai usare 3 pile di carte:



Inizialmente si mettono tutte le carte, a faccia coperta, sulla pila sorgente, con le altre due pile vuote. Poi si eseguono questi passi ripetutamente:

1. Prendere la carta più in alto nella pila sorgente e metterla a faccia scoperta nella pila destinazione.
2. Se la pila sorgente è vuota, avete finito. La pila destinazione è ordinata.
3. Altrimenti, seguite questi passi fino a che la pila sorgente non è vuota:
 - a. Prendere la carta in cima alla pila sorgente e confrontarla con la carta in cima alla pila destinazione
 - b. Se la carta sorgente è più grande...
 - i. Prendere la carta in cima alla pila destinazione e metterla a faccia coperta nella pila scarti
 - ii. Mettere la carta della pila sorgente a faccia scoperta nella pila destinazione
 - c. ...altrimenti
 - i. Mettere la carta della pila sorgente a faccia coperta nella pila scarti
4. Spostare la pila degli scarti nella posizione della pila sorgente e ripartire col punto 1.

Scheda: Merge sort

Ti servirà molto spazio per questo metodo di ordinamento, abbastanza per potere disporre tutte le carte (ci sono metodi per eseguire l'algoritmo merge sort con meno spazio, ma sono più complicati da spiegare). L'abilità principale che ti servirà per il merge sort è sapere unire due mazzi di carte (merging), quindi vai prima a leggere la scheda intitolata appunto "Merging" per sapere come fare. Una volta che hai imparato a fare il merging, il processo di *merge sort* è facile.

Per eseguire il *merge sort* vero e proprio, disponi le carte coperte in riga. Consideriamo queste come le "pile" sorgenti iniziali, anche se c'è solamente una carta in ciascuna pila. Il merge sort lavora unendo progressivamente coppie di pile di carte in modo che ci siano sempre meno pile ma sempre più grandi. Alla fine ci sarà una sola grossa pila di carte.



Ripeti i seguenti passi fino a che non c'è una unica pila di carte:

1. Unisci, con l'algoritmo di *merging*, le prime due pile di carte a faccia coperta
2. Finché ci sono almeno altre due pile di carte a faccia coperta, uniscile (sempre a due a due)
3. Gira tutte le pile in modo che siano a faccia coperta

Scheda: Merging

Partendo da due pile ordinate di carte da unire, una pila accanto all'altra, a faccia coperta, tramite il merging otterrai una unica pila, più in basso, a faccia scoperta:



Prendi la carta in cima a ciascuna sorgente, tenendo la sorgente A nella mano sinistra e la sorgente B nella destra. Ora ripeti i seguenti passi fino a che tutte le carte sono nella pila destinazione:

1. Confronta le carte che hai in mano
2. Metti la carta più grande nella pila destinazione, a faccia in su
3. Con la mano libera, prendi la prossima carta dalla pila sorgente corrispondente e torna al passo 1. Se non ci sono più carte nella pila sorgente corrispondente alla mano vuota, metti l'altra carta che stai tenendo in mano nella pila destinazione a faccia in su e, una alla volta, gira tutte le carte successive di tale pila sorgente sulla pila destinazione

La questione chiave è: e se volessimo ordinare un mazzo di 64 carte? Come ti sentiresti al pensiero di farlo usando il *selection sort*? Ci aspettiamo che ora tu stia facendo versi di disapprovazione. Questa è la sensazione che dà un processo che diventa lento velocemente.

Possiamo trarre qualche altra considerazione interessante da tutta questa fatica. Prepara una tabella con i tuoi tempi di ordinamento, o, meglio, mettili in un grafico, oppure, ancora meglio, mettili assieme a quelli dei tuoi colleghi e fanne un grafico con tempi medi e range. In Figura 6.1 c'è un grafico del genere composto grazie a 10 coppie di studenti; i segni orizzontali sono le medie, mentre le linee verticali sono i range.

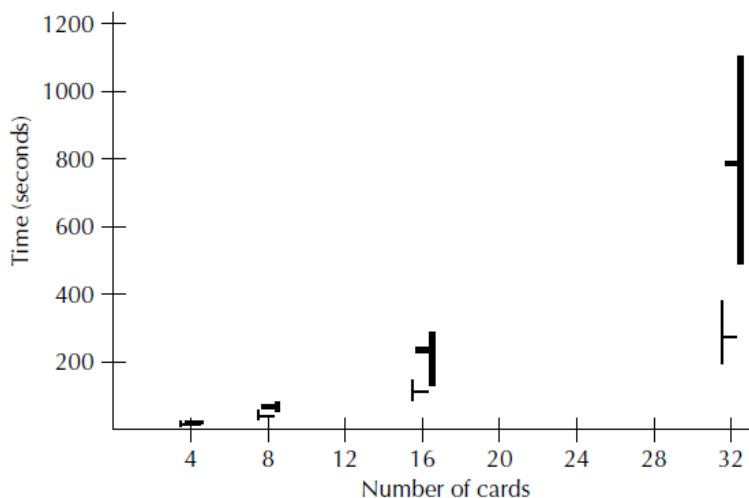


Figura 6.1 – Linee sottili: merge sort; linee marcate: selection sort

Osservando attentamente la Figura 6.1, si nota che i più veloci nel selection sort riescono a ordinare 4 carte più velocemente dei più lenti nel merge sort. Quindi, se hai solamente 4 carte da ordinare, nessuno dei due metodi è intrinsecamente superiore all'altro: qualunque dei due potrebbe risultare più veloce, a seconda delle abilità della persona che sta ordinando. D'altro canto, per 32 carte anche i più lenti nel merge sort sono riusciti a fare tempi migliori dei più veloci nel selection sort. Questo fenomeno generale avviene quando due metodi diventano lenti a tassi di crescita differenti. Una qualunque differenza iniziale in velocità, non importa quanto grande, sarà alla fine sempre superata dalla differenza intrinseca tra i due algoritmi, sempre che si arrivi a testarli su dimensioni del problema adeguate. Se dovessi gareggiare, usando le tue mani, contro un super computer velocissimo programmato per usare il selection sort, potresti sconfiggerlo usando il merge sort, sempre che il concorso comprenda l'ordinamento di un set di dati sufficientemente grande (in realtà, il data set necessario sarebbe così grande che saresti morto prima di vincere la gara: immaginatevi di fare continuare la gara a un figlio, poi a un nipote, etc.).

Un'altra cosa che puoi osservare guardando il grafico è che, se dovessimo tracciare una curva che passa sui tempi medi di ciascun algoritmo, le forme delle curve sarebbero molto diverse tra loro. Ovviamente è difficile essere precisi perché 4 punti sono troppo pochi, ma la differenza qualitativa delle due forme colpisce comunque molto. I numeri del merge sort stanno su una linea quasi orizzontale (la curva verso l'alto è molto leggera). Invece, nessuno potrebbe scambiare la curva del selection sort con una linea orizzontale: i punti del selection sort stanno su una linea con una marcata curvatura verso l'alto. Questo era ciò che intendevamo quando abbiamo detto che raddoppiando il numero di carte il lavoro di ordinamento sarebbe durato molto più del doppio.

Raccogliere altri dati empirici esaurirebbe la pazienza degli studenti più volenterosi, quindi useremo un altro modo per descrivere la forma di queste curve. Troveremo, per ciascun metodo, una funzione che descriva la relazione tra la dimensione del mazzo e il numero di passi eseguiti. Ci concentriamo sul numero di passi piuttosto che sul tempo perché non sappiamo quanto ci voglia per compiere un singolo passo. Infatti alcuni passi possono essere più lunghi di altri, e il tempo varia da persona a persona. Comunque, compiere i passi più (o meno) velocemente risulta semplicemente in una curva e non cambia la sua forma di base.

Consideriamo innanzitutto cosa avviene nel *selection sort* ordinando n carte. Contiamo il numero totale di volte in cui tocchiamo una carta, il numero di “tocchi di carte”. Nel primo passaggio del mazzo, tutte le carte vengono toccate una o due volte ciascuna. Nel secondo passaggio, vengono toccate $n - 1$ carte, nel terzo $n - 2$, e così via. Quindi il numero totale di tocchi di carte sta tra $n + (n - 1) + (n - 2) + \dots + 1$ e il doppio di tale numero. Quanto grande è il numero $n + (n - 1) + (n - 2) + \dots + 1$? È facile vedere che non è più grande di n^2 , perché vengono sommati n numeri e il più grande di questi è n . Possiamo anche vedere che è almeno $\frac{n^2}{4}$, perché ci sono $\frac{n}{2}$ numeri che sono uguali o più grandi di $\frac{n}{2}$. Quindi possiamo dire immediatamente che il numero totale di tocchi di carte è racchiuso tra $(\frac{n^2}{4}, 2n^2)$. Poiché i due estremi dell’intervallo sono entrambi multipli di n^2 , ne segue che la forma di base della curva deve essere una parabola. In simboli, possiamo dire che il numero di tocchi di carte è $\Theta(n^2)$ (la pronuncia convenzionale è “theta grande di n quadro”). Questo significa che, a parte rare eccezioni, la curva sta tra due multipli di n^2 (più precisamente tra due multipli *positivi* di n^2).

Con un po’ di più lavoro, potremmo produrre una formula esatta per la somma $n + (n - 1) + (n - 2) + \dots + 1$. Comunque, questo non aiuterebbe granché, perché non sappiamo se in un passo una carta viene toccata una o due volte e non sappiamo quanto tempo prenda ogni tocco di carta. Quindi, dobbiamo accontentarci di una risposta imprecisa. D’altro canto, possiamo dire con sicurezza che se ci vuole almeno un centesimo di secondo per toccare ogni carta, allora ci vogliono almeno $\frac{n^2}{400}$ secondi per ordinare n carte, e allo stesso modo possiamo dire che se ci vogliono al massimo 1000 secondi per toccare una carta, allora ci vorranno al massimo $2000 \times n^2$ secondi per ordinare n carte. Dunque, l’imprecisione della notazione Θ è esattamente quello di cui abbiamo bisogno; possiamo dire non solo che il numero di “tocchi” è $\Theta(n^2)$, ma anche che il tempo che ci si mette è $\Theta(n^2)$. La nostra risposta, per quanto imprecisa, ci fornisce la forma generale dell’ordine di crescita della funzione. Se riuscissimo, come l’evidenza empirica suggerisce, a mostrare che il *merge sort* ha un ordine di crescita minore, la differenza in termini di ordine di crescita ci direbbe quale metodo è più veloce per grossi mazzi di carte.

ESERCIZIO 6.1:

Scopri quanto vale esattamente la quantità $n + (n - 1) + (n - 2) + \dots + 1$. Prova a sommare l’ultimo termine con il primo, il penultimo con il secondo, etc. Che valore risulta per ogni coppia? Quante coppie ci sono? Qual è il valore finale della somma?

Passando al *merge sort*, possiamo analizzare in modo simile quante volte si toccano le carte, spezzando il processo in passi successivi. Nel primo passaggio si uniscono le n pile iniziali in $\frac{n}{2}$ pile; questo passaggio costringe a considerare ogni singola carta. Nel secondo passaggio si uniscono le $\frac{n}{2}$ pile in $\frac{n}{4}$; quante carte si devono considerare in questo passaggio? Pensaci un attimo.

Se ci hai pensato bene, avrai realizzato che devi considerare tutte le n carte in ciascun passaggio. Rimane allora da determinare quanti passaggi vanno fatti. Il numero di pile viene ridotto della metà ad ogni passaggio, mentre il numero di carte per ciascuna pila raddoppia. Inizialmente ogni carta sta in una pila separata, mentre alla fine le n carte stanno tutte in una unica pila. Quindi, la questione può essere rifrasata come “Quante volte dobbiamo raddoppiare 1 per arrivare a n ?” o equivalentemente “Quante volte n deve essere dimezzato per raggiungere 1?”. Come viene spiegato nella scheda sul logaritmo, la risposta è il logaritmo in base 2 di n , scritto $\log_2 n$ o alcune volte solamente $\log n$. Mettendo insieme questa informazione con il fatto che ad ogni passaggio vanno considerate tutte le carte, scopriamo che abbiamo dovuto fare $n \log_2 n$ “tocchi” di carte. Questo numero non tiene conto del fatto che ad ogni passaggio giriamo le pile, e ovviamente c’è sempre il problema di quanto tempo ci si mette a compiere ogni passo. Dunque, ci conviene anche questa volta essere intenzionalmente imprecisi e dire che il tempo richiesto dal merge sort per ordinare n carte è $\Theta(n \log n)$.

Un punto interessante è che non abbiamo scritto la base del logaritmo. Questo perché il logaritmo compare all’interno di Θ , dove la base di un logaritmo è irrilevante, perché cambiare da una base ad un’altra è equivalente a moltiplicare per un fattore costante, come spiega la scheda sui logaritmi. Ricordate, dire che il tempo è Θ di una certa funzione, significa semplicemente che sta tra due multipli di quella funzione, senza specificare quali multipli esatti. Si noti che un valore sta tra due multipli di $2n^2$ se e solo se esso sta tra due multipli di n^2 , quindi non diciamo mai $\Theta(2n^2)$ ma semplicemente $\Theta(n^2)$. Questa è la stessa ragione per cui lasciamo non specificata la base del logaritmo.

In conclusione, notate che i nostri risultati analitici sono consistenti con le osservazioni empiriche. La funzione $n \log n$ cresce un po’ più velocemente che linearmente, mentre il quadrato cresce decisamente più velocemente. Questa differenza rende il merge sort un algoritmo di ordinamento decisamente superiore.

Scheda: logaritmi

Se $x^k = y$ diciamo che k è il logaritmo in base x di y . Quindi k è l’esponente a cui x deve essere elevato per produrre y . Per esempio, 3 è il logaritmo in base 2 di 8, perché devi moltiplicare tre volte 2 per sé stesso per produrre 8. In simboli, scriviamo che $\log_2 8 = 3$, perché $\log_x y$ è il simbolo per il logaritmo in base x di y . La definizione formale di logaritmo specifica il suo valore anche per casi come $\log_2 9$, che chiaramente non è un intero, perché nessuna potenza di 2 darà mai 9. Per i nostri scopi, quello che si deve sapere è che $\log_2 9$ sta tra 3 e 4, perché 9 sta tra 2^3 e 2^4 .

Poiché sappiamo che moltiplicare 3 volte 2 per sé stesso darà 8, e che due 8 moltiplicati tra loro danno 64, ne segue che moltiplicare sei volte 2 per sé stesso produrrà 64. In altre parole, $\log_2 64 = \log_2 8 \times \log_8 64 = 3 \times 2 = 6$. Questo mostra una proprietà generale dei logaritmi, ovvero che $\log_b x = \log_b c \times \log_c x$. Quindi, a prescindere da quanto vale x , i suoi logaritmi in base b e c differiscono per il fattore $\log_b c$.

6.2 – Ricorsione ad albero

Guardando qualcuno ordinare un mazzo di carte tramite l’algoritmo *merge sort* descritto nella sezione precedente, vedrai che le due metà, di sinistra e di destra, di carte non interagiscono fino all’ultimo passo di *merge*. A quel punto, ciascuna metà delle carte è già ordinata, e l’unica cosa rimasta da fare è unire le due metà. Quindi, l’algoritmo di *merge sort* può essere ristrutturato nel seguente modo. Per ordinare con il *merge sort* un mazzo di n carte:

1. Se $n = 1$, il mazzo è già in ordine quindi hai finito
2. Altrimenti:
 - a. Esegui il *merge sort* per le prime $\frac{n}{2}$ carte
 - b. Esegui il *merge sort* per le altre $\frac{n}{2}$ carte
 - c. Unisci le due metà ordinate

Formulato in questo modo, risulta chiaro che l'algoritmo è ricorsivo; tuttavia, non è la tradizionale ricorsione lineare a cui siamo abituati. Invece di risolvere una versione leggermente più piccola del problema e poi concludere il lavoro rimasto da fare, il *merge sort* risolve due versioni molto più piccole del problema (la dimensione è la metà) e poi conclude combinando i due risultati. Questa strategia di dividere il lavoro in due (o equivalentemente in tre o quattro) sottoproblemi e poi combinarne i risultati per creare la soluzione finale è conosciuta col nome di *ricorsione ad albero*. La ragione di questo nome è che il problema principale si ramifica in sottoproblemi, ciascuno dei quali si ramifica in altri sotto-sotto-problemi, e così via, proprio come le ramificazioni successive dei rami di un albero. A volte questo modo di pensare “ricorsivo ad albero” può portare ad un algoritmo con un ordine di crescita minore rispetto a quello di altre possibili soluzioni. L'ordine di crescita ridotto può essere estremamente importante se si sta scrivendo un programma che lavorerà su moli di dati molto estese.

6.3 – Esercizi di riepilogo

ESERCIZIO 6.2

Consideriamo le seguenti funzioni:

```
let rec factorial n =
  if n = 0 then 1
  else n * factorial (n-1);;

let rec factorial_sum1 n (*ritorna la somma 1! + 2! + ... + n!*) =
  if n = 0 then 0
  else (factorial n) + (factorial_sum1 (n - 1));;

let rec factorial_sum2 n (*ritorna la somma 1! + 2! + ... + n!*) =
  let rec loop k fact_k addend =
    if k > n then addend
    else loop (k+1) (fact_k * (k + 1)) (addend + fact_k)
  in
  loop 1 1 0;;
```

Nel rispondere alle seguenti domande, assumi che n sia un intero non negativo. Inoltre devi giustificare le tue risposte.

- a) Dare una formula che calcoli il numero di moltiplicazioni effettuate dalla funzione `factorial`, espressa in funzione del suo argomento n .
- b) Dare una formula che calcoli il numero di moltiplicazioni effettuate (implicitamente attraverso `factorial`) dalla funzione `factorial_sum1`, espressa in funzione del suo argomento n .
- c) Dare una formula che calcoli il numero di moltiplicazioni effettuate dalla funzione `factorial_sum2`, espressa in funzione del suo argomento n .

ESERCIZIO 6.3

Quanti modi ci sono per fattorizzare n in due o più numeri (ognuno dei quali non deve essere più piccolo di 2)? Potremmo generalizzare questo problema a quello di trovare quanti modi ci sono per fattorizzare n in due o più numeri, ognuno dei quali non è più piccolo di m . Ovvero, scriviamo:

```
let ways_to_factor n =
  ways_to_factor_using_no_smaller_than n 2;;
```

Scrivere la funzione `ways_to_factor_using_no_smaller_than`. Qui ci sono alcune considerazioni che possono essere utili:

- Se $m^2 > n$, quanti modi ci sono per fattorizzare n in due o più numeri ciascuno non più piccolo di m ?
- Altrimenti, considera il caso in cui n non sia divisibile per m . Confronta quanti modi ci sono per fattorizzare n in due o più numeri non più piccoli di m con quanti modi ci sono per fattorizzare n in due o più numeri non più piccoli di $m - 1$. Che relazione c'è tra queste due quantità?
- L'ultimo caso rimasto è quello in cui $m^2 \leq n$ e n è divisibile per m . In questo caso, c'è almeno un modo per fattorizzare n in numeri non più piccoli di m (può essere fattorizzato in m e $\frac{n}{m}$). Potrebbero però esserci anche altri modi. I modi per fattorizzare n si dividono in due categorie: quelli che usano almeno un fattore di m e quelli che non contengono alcun fattore di m . Quante fattorizzazioni ci sono in ciascuna categoria?

ESERCIZIO 6.4

Consideriamo la seguente funzione:

```
let rec bar n =
  match n with
  | 0 -> 5
  | 1 -> 7
  | _ -> n * (bar (n-2));;
```

Quante moltiplicazioni (in notazione Θ) verranno effettuate da `bar n`? Giustifica la risposta. Puoi assumere che `n` sia un intero non negativo.

ESERCIZIO 6.5

Consideriamo le seguenti funzioni:

```
let rec factorial n =
  if n = 0 then 1
  else n * factorial (n-1);;

let rec bar i j (* calcola (i!)^j *) =
  if j = 0 then 1
  else (factorial i) * (bar i (j-1));;

let rec foo n (* calcola n! + (n!)^n *) =
  (factorial n) + (bar n n);;
```

Quante moltiplicazioni (in notazione Θ) verranno effettuate da `foo n`? Giustifica la risposta.

ESERCIZIO 6.6

Supponiamo che ti siano state date n monete che sembrano (al tatto e alla vista) tutte identiche e che ti sia stato detto che una e una sola di quelle monete è falsa. La moneta falsa pesa leggermente di meno di una moneta vera. Hai una bilancia a disposizione e quindi puoi scoprire quale è la moneta falsa confrontando le varie pile di monete. Una strategia per farlo è la seguente:

- a) Se hai una sola moneta, è quella falsa;
- b) Se hai un numero pari di monete, dividi le monete in due pile (mettendo lo stesso numero di monete in ciascuna pila), confronta il peso delle due pile, scarta la pila più pesante e cerca la moneta falsa nella pila rimanente;
- c) Se hai un numero dispari di monete, estrai una moneta, dividi le rimanenti monete in due pile e confronta il peso delle due pile. Se sei fortunato, le due pile peseranno uguale e quindi la moneta che avevi estratto è quella falsa. Altrimenti, getta via la pila più pesante e la moneta che avevi estratto e cerca la moneta falsa nella pila rimanente.

Nota che se hai solo una moneta non devi fare alcuna pesata. Se hai un numero pari di monete, il massimo numero di pesate che devi fare è uno di più del massimo numero di pesate che dovresti effettuare per la metà delle monete. Se hai un numero dispari di monete, il massimo numero di pesate è lo stesso del numero di pesate che dovresti fare per una quantità di monete pari a quella corrente meno uno.

- a) Scrivere una funzione che determini il massimo numero di pesate da effettuare per trovare la moneta falsa tra n monete usando la strategia descritta sopra.
- b) Trovare una strategia più efficiente basata sul dividere la pila di monete in tre, piuttosto che in due, sotto-pile. (Suggerimento: se confronti due delle 3 pile, quali sono i possibili risultati? Cosa significa ciascuno di essi?)
- c) Scrivere una funzione che determini il massimo numero di pesate usando la strategia di divisione in 3 pile.

ESERCIZIO 6.7

Un modo per sommare gli interi da α fino a b consiste nel dividere l'intervallo a metà, sommare ricorsivamente le due metà e poi fare la somma delle due somme risultanti. Ovviamamente, potrebbe non essere possibile dividere l'intervallo esattamente a metà se l'intervallo contiene un numero dispari di interi. In questo caso, l'intervallo può essere diviso in due parti il più possibile vicine alla metà.

- a) Scrivere una funzione che implementi questa idea.
- b) Poniamo n come il numero di interi contenuti nell'intervallo compreso tra α e b . Qual è l'ordine di crescita (in notazione Θ) del numero di somme che la funzione esegue, in funzione di n ? Giustifica la risposta.

ESERCIZIO 6.8

Consideriamo il seguente problema di enumerazione: in quanti modi puoi scegliere k oggetti a partire da n oggetti diversi, assumendo ovviamente che $0 \leq k \leq n$? Per esempio, quante pizze con 3 condimenti puoi cucinare se hai a disposizione 6 condimenti tra cui scegliere?

Questo numero è la risposta al problema comunemente conosciuto come $C(n, k)$. Ecco un algoritmo per calcolare $C(n, k)$:

- a) Come detto sopra, assumiamo che $0 \leq k \leq n$, perché altrimenti il problema non avrebbe senso.
- b) I casi base sono $k = 0$ e $k = n$. Non dovrebbe essere difficile persuadersi che $C(n, n) = 1$ e che $C(n, 0) = 1$.
- c) Il caso generale è $0 < k < n$. Qui possiamo ragionare nel modo seguente: consideriamo uno degli oggetti. Le possibilità sono due: o lo selezioniamo come uno dei k oggetti, oppure no. Se lo selezioniamo, allora dobbiamo selezionare altri $k - 1$ oggetti tra i rimanenti $n - 1$, probabilmente un problema più semplice che si assume di poter risolvere ricorsivamente. Se invece non lo si seleziona, si devono selezionare k oggetti tra i rimanenti $n - 1$, e anche questo è un problema più semplice che può essere risolto ricorsivamente. Quindi il numero totale di modi per selezionare k oggetti a partire da n oggetti è la somma dei numeri che si ottengono nei due sottoproblemi considerati sopra.

Usando questo algoritmo, scrivere una funzione ricorsiva ad albero che calcola $C(n, k)$ come descritto sopra.

Capitolo 7 – Liste

7.1 – La definizione di lista

In questo capitolo tratteremo le liste. Daremo prima la loro definizione formale. Poi mostreremo come questa definizione possa essere usata per costruire liste ed esploreremo infine i diversi modi per manipolare le liste.

Cosa è esattamente una lista? Pensiamo ad un esempio concreto di liste facendo scrivere a tutti gli studenti la lista dei corsi da loro frequentati ieri. Alcuni studenti potrebbero avere una lista relativamente lunga, altri potrebbero avere frequentato solo uno o due corsi. Alcuni potrebbero non avere frequentato alcun corso (se oggi fosse lunedì probabilmente tutti si troverebbero in questa situazione). Le liste possono essere scritte in molti modi: alcuni potrebbero scrivere una lista in una colonna, altri potrebbero usare le righe, e altri potrebbero trovare modi più creativi per farlo. A prescindere da come sono scritte, le liste con almeno un elemento hanno anche un certo ordine (ad esempio, ogni lista può essere scritta a partire dal primo corso della giornata e proseguire in ordine temporale). Quindi potremmo definire una lista come *una collezione di 0 o più elementi omogenei, scritti in un certo ordine*. Una lista si distingue da un insieme in quanto uno stesso elemento può apparire in una lista più volte, e ad ogni elemento appartenente alla lista è associata una posizione (l'ordine degli elementi è significativo).

Ora immaginiamo di dover scrivere un programma per gestire liste formate da un certo tipo di elementi, diciamo interi per semplicità. Il primo compito da svolgere è definire il tipo di dato astratto “*lista di interi*”. Chiaramente, sarà un tipo di dato composto. Ma quanti elementi ci servono? Due liste diverse potrebbero avere dimensioni completamente diverse... cosa facciamo? Vogliamo che il nostro tipo di dato abbia valori di dimensioni differenti.

Il modo migliore per definire un tipo di dato consiste nel concentrarsi su come questo viene utilizzato. Possiamo illustrare come le liste vengono usate attraverso l'esempio della lista della spesa. Immaginiamo un cliente di un supermercato che costruisce la propria lista della spesa in modo che gli oggetti siano ordinati in base alla loro posizione nel carrello. La lista viene quindi usata cercando il suo primo elemento, mettendolo nel carrello, e scartandolo dalla lista. A questo punto avremo una nuova lista della spesa (contenente un elemento di meno) che si usa per continuare gli acquisti. Alla fine, ci sarà solamente un elemento nella lista (la barretta di cioccolato che sta alle casse). Dopo aver messo questo oggetto nel carrello e averlo eliminato dalla lista, la lista della spesa è vuota. Ciò significa che è arrivato il momento di andare alla cassa e pagare la merce.

L'esempio della lista della spesa ha una struttura molto ricorsiva, e quindi possiamo prenderla come modello per la definizione ricorsiva delle liste. Avremo bisogno di un “caso base” che definisce la lista più piccola possibile, e un qualche modo per definire liste più grandi in termini di liste più piccole. Il caso base è facile: c'è una lista speciale, chiamata *lista vuota*, che non contiene alcun elemento. Il caso generale è intuitivo a partire dall'esempio della lista della spesa visto prima: una lista non vuota è formata da due parti, una delle quali è un elemento (il primo della lista), e l'altro è un'altra lista, ovvero la lista di tutti gli altri elementi della lista completa. Per riassumere:

DEFINIZIONE DI LISTA: Una lista è vuota oppure è formata da due parti: il primo elemento della lista e la lista degli elementi rimanenti.

Il primo elemento di una lista non vuota è spesso chiamato la **testa** della lista, mentre la lista degli altri elementi è chiamata la **coda**. Questa visione delle liste è ciò che distingue un informatico da una persona qualsiasi. In genere le persone pensano che una lista possa avere un numero qualunque di elementi (gli oggetti della lista), mentre un informatico pensa che tutte le liste non vuote siano formate da due componenti (la testa e la coda). La coda non è un elemento della lista, ma è essa stessa una lista di elementi.

Come possiamo implementare le liste? Dato che la maggior parte delle liste è formata da due componenti, sembrerebbe naturale usare le coppie: il primo elemento della coppia potrebbe essere la testa della lista, mentre il secondo elemento potrebbe essere la coda. Tuttavia, una lista potrebbe essere vuota e quindi non avrebbe due componenti: dobbiamo tenere conto anche delle liste vuote. La soluzione è la seguente: usiamo il valore speciale `[]` per codificare la lista vuota e il costruttore `::` (detto anche **cons**) per “agganciare” un elemento ad una lista esistente (rendendo la lista esistente la coda e l’elemento aggiunto la testa della nuova lista).

- `(elt) :: (lst)`
 - dato un elemento `elt` e una lista `lst`, `::` restituisce la lista la cui testa è l’elemento `elt` e la cui coda è `lst`.

Poiché la lista più semplice di tutte è la lista vuota, la lista con un elemento si costruirà agganciando tale elemento alla lista vuota, tramite il costruttore `::`. Per costruire la lista con due elementi dovremo agganciare uno dei due elementi alla lista contenente solamente l’altro elemento (creata come appena detto), e così via per ogni ulteriore elemento che vogliamo aggiungere. La procedura risulta evidentemente ricorsiva, e la lista vuota ne è il caso base!

Varie delle procedure che scriveremo sono già definite all’interno del linguaggio. Tuttavia le scriveremo lo stesso perché forniscono ottimi esempi di tecniche di gestione delle liste. Inoltre, scrivendole, capiremo meglio cosa fanno le procedure pre-definite.

7.2 – Costruire liste

Una lista finita è denotata utilizzando le parentesi quadre e il simbolo “;” come separatore degli elementi. Scriviamo gli elementi della lista in ordine. Vediamo alcuni esempi di liste:

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

# [1;1;3];
- : int list = [1; 1; 3]

# ["hello"; "world"];
- : string list = ["hello"; "world"]

# [];
- : 'a list = []

# [[]];
- : 'a list list = []
```

```
# [(prefix +);(prefix *)];;
- : (int -> int -> int) list = [<fun>; <fun>]
```

La lista vuota è denotata da [] e una lista di un solo elemento a da [a]. Se gli elementi di una lista hanno tipo α , allora il tipo della lista sarà α list. Negli esempi si vede il tipo associato ad ogni lista. In particolare, la lista vuota ha tipo polimorfo.

Se provassimo a costruire una lista con elementi di tipo diverso, ci verrebbe restituito un errore:

```
# [1.5; 2; 1.1];;
Toplevel input:
>[1.5; 2; 1.1];;
>^^^^^^^^^^^^^^^
This expression has type float list,
but is used with type int list.
```

Vediamo un altro esempio di costruzione di lista, basato sull'esempio della lista della spesa:

```
#   ["shampoo"; "carta alluminio"; "riso"; "sugo"; "pesce";
"formaggio"];;
- : string list = ["shampoo"; "carta alluminio"; "riso"; "sugo";
"pesce"; "formaggio"]
```

Abbiamo costruito una lista di stringhe: ciascuna stringa della lista è il nome dell'oggetto da comprare.

Le stringhe sono delle liste di caratteri: in vari sistemi “hello” è semplicemente un modo più compatto per scrivere ['h'; 'e'; 'l'; 'l'; 'o'].

Una lista può contenere lo stesso valore più volte, come nel secondo esempio. Due liste sono uguali se e solo se hanno gli stessi elementi nello stesso ordine. È possibile anche confrontare due liste tra loro per mezzo degli operatori di confronto. Le liste sono ordinate secondo l'ordinamento lessicografico come si vede dai seguenti esempi:

```
# [1;2;3]=[2;1;3];;
- : bool = false

# [1;2]<[2;3];;
- : bool = true

# [1;2]<[2;1];;
- : bool = true

# [2;3]<[1;4];;
- : bool = false

# [1;3]<[1;9;15];;
- : bool = true

# [2;3;15]<[2;9];;
- : bool = true
```

```
# [2;3]<[2;3;19];;
- : bool = true

# [2;3;15]<[2;3];;
- : bool = false
```

Tutte le liste non vuote sono create usando il costruttore `::`, definito nel seguente modo:

```
# (prefix ::);;
- : 'a * 'a list -> 'a list = <fun>

# 1::[];;
- : int list = [1]
```

Qui sopra abbiamo creato una lista di interi, contenente un solo elemento di valore 1, agganciando l'elemento 1 alla lista vuota `[]` tramite il costruttore `::`.

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

L'operatore `::` (che chiameremo *cons*, costruttore) prende come argomenti un valore ed una lista di valori dello stesso tipo, e restituisce la lista ottenuta aggiungendo il primo argomento in testa.

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

Qui sopra abbiamo creato un'altra lista di interi, contenente 3 elementi. Come possiamo leggere quello che abbiamo scritto? Abbiamo chiesto una lista che avesse come testa il valore 1 e come coda la lista costruita tramite `2::3::[]`. Ma cos'è la lista così costruita? La lista avente come testa l'elemento 2 e come coda la lista formata da `3::[]`. A sua volta, questa è la lista avente come testa l'elemento 3 e come coda la lista vuota: siamo arrivati al caso base! Dunque la nostra lista finale è una lista contenente i valori interi da 1 a 3. Per convenzione, l'operatore `::` associa a destra, per cui `1::2::3::[]` vuol dire `1::(2::(3::[]))`. Ogni lista può essere costruita inserendo i suoi elementi ad uno ad uno nella lista vuota. Possiamo infatti fare riferimento ad una lista $[x_1; x_2; \dots; x_n]$ come ad una abbreviazione per $x_1 :: x_2 :: \dots :: x_n :: []$. Da notare che l'elemento scritto più a sinistra risulta essere poi il primo elemento della lista (la testa).

Vediamo ancora qualche esempio di costruzione di liste tramite il costruttore `::`:

```
# "ciao"::"mondo"::[];;
- : string list = ["ciao"; "mondo"]

# 'c'::'i'::'a'::'o'::[];;
- : char list = ['c'; 'i'; 'a'; 'o']

# 1.5::2.1::3.7::[];;
- : float list = [1.5; 2.1; 3.7]
```

Negli esempi visti sopra abbiamo costruito, rispettivamente, una lista contenente due stringhe, una lista contenente quattro caratteri e una lista contenente tre valori di tipo float. Cosa succederebbe se provassimo a inserire valori di tipo diverso nella stessa lista (ad esempio due float e un intero)?

```
# 1.5::2::3.7::[];;
Toplevel input:
>1.5::2::3.7::[];;
>      ^^^^^^
This expression has type float list,
but is used with type int list.
```

Ci viene restituito un errore: le liste sono collezioni di elementi tutti dello stesso tipo!

Finora abbiamo considerato esempi in cui il numero di elementi da inserire nella lista è molto piccolo. Come potremmo costruire una lista con molti elementi? Ipotizziamo di volere costruire la lista degli interi da 1 a 100. Invece che dovere scrivere manualmente l'intera lista, dovremmo automatizzare il processo scrivendo una procedura che costruisca la lista per noi. Come nell'esempio del fattoriale, sembra sensato scrivere una procedura di carattere generale che produca la lista degli interi compresi tra `low` e `high` e poi usarla imponendo `low = 1` e `high = 100`. La procedura è la seguente:

```
# let rec interidaa low high =
    if low > high then []
    else (low)::( interidaa (low + 1) high);;
interidaa : int -> int -> int list = <fun>
```

Per scrivere tale procedura abbiamo usato lo stesso metodo visto prima (a parole) per costruire la lista di interi compresi tra 1 e 3 tramite il costruttore `::`. Infatti la lista di interi da 1 a 3 è costruita agganciando 1 alla lista contenente gli interi compresi tra 2 e 3. La lista degli interi da 2 a 3 è sua volta costruita agganciando 2 alla lista contenente gli interi da 3 a 3 e così via, fino al caso base (`4 > 3`) in cui restituiamo la lista vuota. Proviamo ad usare la nostra procedura:

```
# interidaa 1 100;;
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19;
20; 21; 22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33; 34; 35; 36;
37; 38; 39; 40; 41; 42; 43; 44; 45; 46; 47; 48; 49; 50; 51; 52; 53;
54; 55; 56; 57; 58; 59; 60; 61; 62; 63; 64; 65; 66; 67; 68; 69; 70;
71; 72; 73; 74; 75; 76; 77; 78; 79; 80; 81; 82; 83; 84; 85; 86; 87;
88; 89; 90; 91; 92; 93; 94; 95; 96; 97; 98; 99; 100]
```

La nostra procedura funziona correttamente: ci ha restituito la lista di interi da 1 a 100.

NOTA BENE: In F# possiamo scrivere `[1..100]` per ottenere automaticamente la lista dei numeri interi da 1 a 100. Vediamo un esempio di utilizzo di tale funzionalità:

```
> [1..100];
val it : int list
= [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17;
18; 19; 20; 21; 22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33;
34; 35; 36; 37; 38; 39; 40; 41; 42; 43; 44; 45; 46; 47; 48; 49;
50; 51; 52; 53; 54; 55; 56; 57; 58; 59; 60; 61; 62; 63; 64; 65;
66; 67; 68; 69; 70; 71; 72; 73; 74; 75; 76; 77; 78; 79; 80; 81;
82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92; 93; 94; 95; 96; 97;
98; 99; 100]
```

ESERCIZIO 7.1:

Cosa si ottiene valutando l'espressione `interidaa 7 1`? Esattamente qual è il valore dell'espressione `interidaa 7 1`? Descrivere come i parametri `low` e `high` sono in relazione tra loro.

ESERCIZIO 7.2:

Scrivere una funzione che generi la lista dei numeri interi pari da `low` a `high`.

ESERCIZIO 7.3:

Potremmo riscrivere la funzione `interidaa` in modo che generi un processo iterativo. Consideriamo il seguente tentativo (sbagliato!):

```
# let interidaa low high =
    let rec aux low list =
        if low > high then list
        else (aux (low+1) (low::list))
    in aux low [];
interidaa : int -> int -> int list = <fun>
```

Cosa succede quando valutiamo l'espressione `interidaa 2 7`? Perché? Riscrivere la funzione in modo che generi la lista corretta.

SOLUZIONE

Valutando l'espressione `interidaa 2 7` otteniamo:

```
# interidaa 2 7;;
- : int list = [7;6;5;4;3;2]
```

Possiamo riscrivere la funzione nel seguente modo:

```
# let interidaa low high =
    let rec aux high list =
        if low > high then list
        else (aux (high-1) (high::list))
    in aux high [];
interidaa : int -> int -> int list = <fun>

# interidaa 2 7;;
- : int list = [2;3;4;5;6;7]
```

7.3 – Tecniche di base per la gestione delle liste

Nei capitoli precedenti abbiamo visto come definire una funzione per mezzo di pattern matching. Tramite l'operatore `::`, è possibile definire pattern matching anche sulle liste. Ad esempio, possiamo definire la funzione `null`, che ci dice se una lista è vuota:

```
# let null xs = match xs with
    [] -> true
    | _ -> false;;
null : 'a list -> bool = <fun>
```

Si noti che i due casi definiti sopra sono esaustivi (e quindi la funzione è definita su tutte le liste). Come abbiamo visto (e vedremo anche nel caso delle liste), nel caso di pattern matching è possibile anche definire un insieme di casi non esaustivi. È comunque possibile anche specializzare i casi, come nel seguente esempio:

```
# let singleton xs = match xs with
  []          -> false
  | [x]        -> true
  | (x::y::xs) -> false;;
singleton : 'a list -> bool = <fun>
```

Si noti che qui il pattern `[x]` è usato per indicare `x::[]`.

La tabella seguente riporta, in ogni riga, un pattern di esempio, un'espressione con cui il pattern viene confrontato, il modo di leggere l'espressione applicando l'operatore `::` nella stessa maniera in cui è specificato il pattern e, infine, l'esito del confronto, con gli eventuali legami stabiliti dall'operazione di pattern matching. Affiché il pattern matching abbia successo, il pattern deve essere scritto nella stessa forma dell'espressione da confrontare o in una equivalente, ottenibile con le regole di composizione delle liste. Nei pattern è preferibile usare simboli come `xs` o `ys` per indicare le liste (la "s" inglese, usata per indicare il plurale, evidenzia in questo caso una pluralità di elementi).

<i>pattern</i>	<i>espressione da confrontare</i>	<i>espressione equivalente</i>	<i>esito confronto e legami</i>
<code>[]</code>	<code>[]</code>		successo
<code>[]</code>	<code>[1]</code>		fallimento
<code>[]</code>	<code>[1; 2]</code>		fallimento
<code>[x]</code>	<code>[]</code>		fallimento
<code>[x]</code>	<code>[1]</code>		x=1
<code>[x]</code>	<code>[1; 2]</code>		fallimento
<code>x::[]</code>	<code>[]</code>		fallimento
<code>x::[]</code>	<code>[1]</code>	<code>1::[]</code>	x=1
<code>x::[]</code>	<code>[1; 2]</code>	<code>1::[2]</code>	fallimento
<code>x::y::[]</code>	<code>[]</code>		fallimento
<code>x::y::[]</code>	<code>[1]</code>	<code>1::[]</code>	fallimento
<code>x::y::[]</code>	<code>[1; 2]</code>	<code>1::2::[]</code>	x=1, y=2
<code>x::y</code>	<code>[]</code>		fallimento
<code>x::y</code>	<code>[1]</code>	<code>1::[]</code>	x=1, y=[]
<code>x::y</code>	<code>[1; 2]</code>	<code>1::[2]</code>	x=1, y=[2]
<code>x::y</code>	<code>[1; 2; 3]</code>	<code>1::[2; 3]</code>	x=1, y=[2; 3]
<code>x::xs</code>	<code>[]</code>		fallimento
<code>x::xs</code>	<code>[1]</code>	<code>1::[]</code>	x=1, xs=[]
<code>x::xs</code>	<code>[1; 2]</code>	<code>1::[2]</code>	x=1, xs=[2]
<code>x::xs</code>	<code>[1; 2; 3]</code>	<code>1::[2; 3]</code>	x=1, xs=[2; 3]
<code>x::y::xs</code>	<code>[]</code>		fallimento
<code>x::y::xs</code>	<code>[1]</code>	<code>1::[]</code>	fallimento
<code>x::y::xs</code>	<code>[1; 2]</code>	<code>1::2::[]</code>	x=1, y=2, xs=[]
<code>x::y::xs</code>	<code>[1; 2; 3]</code>	<code>1::2::[3]</code>	x=1, y=2, xs=[3]
<code>x::y::xs</code>	<code>[1; 2; 3; 4; 5]</code>	<code>1::2::[3; 4; 5]</code>	x=1, y=2, xs=[3; 4; 5]

Due liste possono essere concatenate in modo da formare un'unica lista. Tale funzione è denotata da un operatore binario infisso @.

```
# (prefix @);;
- : 'a list * 'a list -> 'a list = <fun>

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

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

Come si vede dal primo esempio, la funzione di concatenazione è polimorfa. Tale funzione prende due liste dello stesso tipo e restituisce una terza lista dello stesso tipo. Una distinzione tra @ e :: è che ogni lista può essere espressa univocamente in termini di :: e []. Questo non è vero in generale per @, poiché la concatenazione è un'operazione associativa (per esempio, [1;2]@[3]=[1]@[2;3]).

Si noti che è possibile definire l'operatore :: per mezzo di @:

$$x :: xs = [x]@xs$$

Proviamo ad implementare la funzione di concatenazione senza usare l'operatore @.

```
# let rec append (xs, ys) =
  match xs with
    [] -> ys
    | z::zs -> z::append(zs, ys);;
append : 'a list * 'a list -> 'a list = <fun>

# append ([1;3], [7]);;
- : int list = [1; 3; 7]
```

Ipotizziamo di dover scrivere una procedura che conti il numero di elementi contenuti in una lista. La definizione ricorsiva di lista ci può aiutare a definire esattamente cosa intendiamo con “numero di elementi in una lista”. Ricordiamo che una lista è vuota oppure è formata da due parti, il primo elemento e la lista degli elementi rimanenti. Quando una lista è vuota, il numero dei suoi elementi è zero. Quando non è vuota, il numero di elementi è uno di più del numero di elementi contenuti nella coda. Possiamo quindi scrivere:

```
# let rec length l =
  match l with
  | [] -> 0
  | x::xs -> 1 + (length xs);;
length : 'a list -> int = <fun>
```

Anche in questo caso la funzione è polimorfa, essendo irrilevante il tipo degli elementi della lista. Si noti che vale:

$$\text{length}(xs@ys) = \text{length } xs + \text{length } ys$$

Allo stesso modo, possiamo scrivere la funzione che calcola la somma degli elementi di una lista:

```
# let rec sum l =
  match l with
  | [] -> 0
  | x::xs -> x + (sum xs);;
sum : int list -> int = <fun>
```

Notiamo come queste due procedure siano simili alle procedure ricorsive con parametri interi, ad esempio il fattoriale. Il caso base in `length` e `sum` avviene quando la lista è vuota, esattamente come il caso base del fattoriale è quando l'intero vale 0. Nel fattoriale riducevamo il nostro intero sottraendogli 1, mentre in `length` e `sum` riduciamo la nostra lista prendendone solo la coda.

Scriviamo ora una funzione ricorsiva che calcola la somma degli elementi dispari di una lista:

```
# let rec sum_odd l =
  match l with
  | [] -> 0
  | x::xs -> if (x mod 2 <>0)
              then x + (sum_odd xs)
              else (sum_odd xs);;
sum_odd : int list -> int = <fun>
```

N.B.: Il pattern matching su liste è molto semplice: sappiamo che una lista qualunque può essere vuota, oppure può essere formata da due parti (la testa agganciata tramite il costruttore `::` alla coda). Nel pattern matching consideriamo esattamente queste due possibilità: la prima si rappresenta con `[]` (cioé proprio il simbolo della lista vuota), mentre la seconda possibilità si rappresenta con `x::xs`, dove `x` è la testa della lista e `xs` è la coda. Ovviamente i nomi `x` e `xs` sono scelti a caso; avremmo potuto ad esempio chiamarli `testa` e `coda`, scrivendo quindi `testa::coda`.

Data una lista, la funzione `hd` seleziona il primo elemento della lista, mentre la funzione `tl` seleziona la porzione rimanente:

- `hd lst`
 - Se `lst` è una lista non vuota, `hd` ne ritorna la testa.
- `tl lst`
 - Se `lst` è una lista non vuota, `tl` ne ritorna la coda.

```
# hd;;
- : 'a list -> 'a = <fun>

# tl;;
- : 'a list -> 'a list = <fun>

#hd [];
Uncaught exception: Failure "hd"

#tl [];
Uncaught exception: Failure "tl"
```

```
#hd [1;2;3];
- : int = 1

#tl [1;2;3];
- : int list = [2; 3]
```

Come si può vedere le funzioni `hd` e `tl` non sono definite sulla lista vuota, e sono polimorfe. Si noti che è possibile definire `hd` e `tl` per mezzo dell'operatore `::` poiché è soddisfatta la relazione:

$$xs = (hd\ xs) :: (tl\ xs)$$

La definizione delle due funzioni che ne deriva è molto semplice:

```
# let hd lst = match lst with x::xs -> x;;
hd : 'a list -> 'a = <fun>

# let tl lst = match lst with x::xs -> xs;;
tl : 'a list -> 'a list = <fun>
```

Vediamo altre funzioni predefinite sulle liste.

La funzione `rev` permette di invertire l'ordine degli elementi di una lista:

```
# rev;;
- : 'a list -> 'a list = <fun>

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

La funzione `diff` calcola la differenza tra due liste `xs` e `ys`. La differenza tra due liste `xs` e `ys` è definita come la lista ottenuta da `xs` rimuovendo la prima occorrenza di ogni `y` appartenente a `ys`.

```
# diff;;
- : 'a list -> 'a list -> 'a list = <fun>

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

È possibile inoltre combinare due liste tramite la funzione `combine`:

```
# combine;;
- : 'a list * 'b list -> ('a * 'b) list = <fun>

# combine ([1;2;3],[`a`;`b`;`c`]);
- : (int * char) list = [(1, `a`); (2, `b`); (3, `c`)]
```

La funzione `combine` prende come argomento una coppia di liste generiche e restituisce la lista ottenuta prendendo le coppie dei corrispondenti elementi. Proviamo a scrivere l'implementazione di questa funzione:

```
# let rec combine (lst1, lst2)=
  match (lst1, lst2) with
  ([] , []) -> []
  | (x::xs, y::ys) -> (x, y)::combine(xs, ys) ;;
combine : 'a list * 'b list -> ('a * 'b) list = <fun>
```

La funzione `combine` ammette la funzione inversa, `split`: data una lista di coppie, restituisce una coppia di liste.

```
# split;;
- : ('a * 'b) list -> 'a list * 'b list = <fun>
```

Proviamo a definirla.

```
# let rec split lst =
  match lst with
  [] -> ([] , [])
  | (x,y)::ls -> let (xs, ys) = split ls
                  in (x::xs, y::ys);;
- : ('a * 'b) list -> 'a list * 'b list = <fun>

# split [(1, `a`); (2, `b`); (3, `c`)];;
- : int list * char list = [1; 2; 3], [`a`; `b`; `c`]
```

Si ricordi che la virgola viene utilizzata per le tuple, mentre il punto e virgola viene utilizzato per le liste. Infatti si ha:

```
# (* lista di coppie i cui elementi sono liste di interi *)
[[1;2],[3;4;5]];;
- : (int list * int list) list = [[1; 2], [3; 4; 5]]

# [[1;2],[3;4;5]; [7;8],[9]];;
- : (int list * int list) list = [[1; 2], [3; 4; 5]; [7; 8], [9]]

# (* lista di liste di interi *)
[[1;2];[3;4;5];[7;8];[9]];;
- : int list list = [[1; 2]; [3; 4; 5]; [7; 8]; [9]]
```

ESERCIZIO 7.4:

In quanti modi una lista $[1; 2; \dots; n]$ può essere espressa come concatenazione di due liste?

ESERCIZIO 7.5: Quali delle seguenti equazioni sono vere?

$$\begin{aligned} [] :: xs &= xs \\ [] :: xs &= [[\] ; xs] \\ xs :: [] &= xs \\ xs :: [] &= [xs] \\ x :: y &= [x; y] \\ (x :: xs) @ ys &= x :: (xs @ ys) \end{aligned}$$

ESERCIZIO 7.6:

Usando `combine`, si definisca la funzione `combine4` che converte una quadrupla di liste in una lista di quadruple.

Scriviamo la funzione `copy`: `int * 'a -> 'a list` che, applicata ad una coppia (n, x) , restituisca la lista di lunghezza n i cui elementi sono tutti uguali a x .

```
# let rec copy (n, x) =
  if n=0 then []
  else x::copy(n-1, x);;
copy : int * 'a -> 'a list = <fun>

# copy (5, 7);;
- : int list = [7; 7; 7; 7; 7]

# copy (5, `R`);;
- : char list = [`R`; `R`; `R`; `R`; `R`]
```

Scriviamo la funzione `duplica` che applicata ad una lista $lst=[x_1; x_2; \dots; x_n]$, duplica ogni elemento della lista, cioè restituisce la lista $[x_1; x_1; x_2; x_2; \dots; x_n; x_n]$.

```
# let rec duplica lst =
  match lst with
    [] -> []
  | x::xs -> x::x::duplica(xs);;
duplica : 'a list -> 'a list = <fun>

# duplica [1;2;3];
- : int list = [1; 1; 2; 2; 3; 3]
```

La funzione `pairwith` applicata ad un oggetto y e un lista $lst=[x_1; x_2; \dots; x_n]$, restituisce la lista di coppie $[(y, x_1); (y, x_2); \dots; (y, x_n)]$:

```
# let rec pairwith y lst =
  match lst with
    [] -> []
  | x::xs -> (y,x)::(pairwith y xs);;
pairwith : 'a -> 'b list -> ('a * 'b) list = <fun>

# pairwith 0 [3;4;5];
- : (int * int) list = [0, 3; 0, 4; 0, 5]

# pairwith 1 [`a`; `b`; `c`];
- : (int * char) list = [1, `a`; 1, `b`; 1, `c`]
```

La funzione `alternate` applicata ad una lista lst restituisce la lista contenente tutti e soli gli elementi di lst che si trovano in posizione dispari. Per convenzione, il primo elemento della lista si trova in posizione 0.

```

# let rec alternate lst =
  match lst with
  [] -> []
  | [x] -> []
  | x::y::xs -> y::(alternate xs);;
alternate : 'a list -> 'a list = <fun>

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

# alternate [`a`; `b`; `c`; `d`; `e`; `f`];;
- : int list = [`b`; `d`; `f`]

```

La funzione `enumerate` applicata ad una lista `lst=[x1;x2;...;xn]`, restituisce la lista di coppie `[(0,x1);(1,x2);...;(n-1,xn)]`:

```

# let enumerate lst =
  let rec aux n lst =
    match (n, lst) with
    (_, []) -> []
    | (n, x::xs) -> (n, x)::(aux n+1 xs)
  in aux 0 lst;;
enumerate : 'a list -> (int * 'a) list = <fun>

# enumerate [1;2;3];;
- : (int * int) list = [0, 1; 1, 2; 2, 3]

# enumerate [`a`; `b`; `c`];;
- : (int * char) list = [0, `a`; 1, `b`; 2, `c`]

```

Scriviamo una funzione ricorsiva `nth` che dati una lista `lst` di lunghezza n e un numero intero k compreso tra 0 e $n - 1$, restituisca il k -esimo elemento della lista `lst`. Per essere ancora più precisi, la chiamata `a nth lst k` ritornerà il $(k+1)$ -esimo elemento della lista `lst`, perché per convenzione per $k = 0$ viene restituito il primo elemento, per $k = 1$ il secondo, e così via. La funzione `nth` può essere definita come segue:

```

# let rec nth lst k =
  match (lst, k) with
  (x::xs, 0) -> x
  | (x::xs, k) -> (nth xs k-1);;
nth : 'a list -> int -> 'a = <fun>

# nth [1;2;3] 0;;
- : int = 1

# nth [1;2;3] 2;;
- : int = 3

```

Definiamo ora la funzione `take` che è una generalizzazione della funzione `hd` come segue: la funzione `take` prende come argomenti un intero n ed una lista `lst` e seleziona i primi n elementi della lista `lst`:

```
# let rec take n lst =
  match (n, lst) with
  | (0, _) -> []
  | (_, []) -> []
  | (n, x::xs) -> x::(take n-1 xs);;
take : int * 'a list -> 'a list = <fun>

# take 3 [1;2;3;4];;
- : int list = [1;2;3]

# take 3 [1;2];;
- : int list = [1;2]

# take 7 [];;
- : 'a list = []
```

Di seguito proponiamo alcuni esercizi da svolgere.

ESERCIZIO 7.7:

Scrivere una funzione che date due liste di interi della stessa dimensione ritorni `true` se ogni elemento della prima è minore del corrispondente elemento della seconda. Ad esempio `lst_min [1;2;3] [2;3;4]` deve valutare a `true`.

Cosa dovrebbe accadere se le due liste non hanno la stessa dimensione?

ESERCIZIO 7.8:

Scrivere una funzione ricorsiva che conti il numero di volte in cui un certo elemento compare in una data lista.

ESERCIZIO 7.9:

Ecco qualche altro esercizio su liste:

- a. Scrivere una funzione che determini la posizione di un certo elemento in una lista. Ad esempio, l'espressione `position 50 [10;20;30;40;50;3;2;1]` deve valutare a 4. Da notare che abbiamo usato la stessa convenzione della funzione `nth`, cioè la prima posizione è la 0 e così via. Cosa dovremmo restituire se l'elemento cercato non è nella lista? Cosa dovremmo restituire se l'elemento compare più di una volta nella lista?
- b. Scrivere una funzione che restituisca l'elemento più grande di una lista non vuota.
- c. Scrivere una funzione che trovi la posizione dell'elemento più grande di una lista non vuota.
- d. Scrivere una funzione che restituisca l'elemento più piccolo pari di una lista non vuota.
- e. Scrivere una funzione che trovi la posizione dell'elemento più piccolo pari di una lista non vuota.

ESERCIZIO 7.10:

Definire la funzione `drop` che prende come argomenti un intero n ed una lista e seleziona gli ultimi elementi della lista esclusi i primi n :

```
# drop;;
- : int -> 'a list -> 'a list = <fun>

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

# drop 4 [1;2];;
- : int list = []
```

Abbiamo già visto come gli operatori di confronto siano definiti su una lista. Esistono inoltre altre funzioni che permettono di stabilire se un predicato è vero per gli elementi di una lista. È il caso delle funzioni predefinite `for_all`, `exists` e `mem`.

La funzione `for_all` prende come argomenti un predicato ed una lista e verifica che tutti gli elementi della lista soddisfino il predicato.

```
# for_all (fun x -> x > 1) [2;3;4];;
- : bool = true
```

o, equivalentemente, possiamo scrivere

```
# let test x = x > 1
    in for_all test [2;3;4];;
- : bool = true
```

La funzione `exists` prende come argomenti un predicato ed una lista e verifica che almeno un elemento della lista soddisfi il predicato.

```
# exists (fun x -> x > 1) [1;0;-2];;
- : bool = false
```

o, equivalentemente,

```
# let test x = x > 1
    in exists test [1;0;-2];;
- : bool = false
```

La funzione `mem` prende come argomenti un elemento ed una lista e verifica che l'argomento sia presente nella lista.

```
# mem 1 [1;2;3];;
- : bool = true
```

Proviamo a scrivere l'implementazione della funzione `mem`.

```
# let rec mem n lst =
  match lst with
    [] -> false
  | x::xs -> (n = x) or mem n xs;;
mem : 'a -> 'a list -> bool = <fun>
```

La funzione `init`, data una lista, restituisce la lista che si ottiene eliminando l'ultimo elemento:

```
# let rec init lst =
  match lst with
  [x] -> []
  | x::xs -> x::init(xs);;
init : 'a list -> 'a list = <fun>

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

7.4 – Esempio: mescoliamo un mazzo di carte

A questo punto, abbiamo visto abbastanza esempi isolati di gestione delle liste: iniziamo un percorso di più larga scala che coinvolga le liste e il loro utilizzo. Consideriamo questo fatto: dopo un certo numero (piccolo) di “mescolamenti perfetti” su un mazzo di 52 carte, il mazzo ritorna nel suo ordine originale. Per “mescolamento perfetto” intendiamo che il mazzo viene diviso in due parti uguali, che sono poi combinate in modo alternato cominciando dalla prima carta della prima metà. Quanti “mescolamenti perfetti” sono richiesti per far tornare un mazzo di 52 carte nel suo ordine iniziale? Possiamo rappresentare il nostro mazzo originale come la lista dei numeri da 1 a 52, usando la procedura `interidaa`. Come dividiamo il mazzo in due metà uguali? Possiamo scrivere due funzioni generali, una che ci restituisca i primi n elementi di una lista e l'altra che ci restituisca i rimanenti.

Scriviamo la funzione `primielementi` che costruisce la lista dei primi n elementi di una lista.

```
# let rec primielementi n lst =
  if n = 0 then []
  else (hd lst)::(primielementi (n-1) (tl lst));;
primielementi : int -> 'a list -> 'a list = <fun>
```

Scriviamo ora la funzione `ultimielementi` che costruisce la lista di tutti gli elementi della lista originale tranne i primi n .

```
# let rec ultimielementi n lst =
  if n = 0 then lst
  else (ultimielementi (n-1) (tl lst));;
ultimielementi : int -> 'a list -> 'a list = <fun>
```

Per ogni dato valore di n tale che $0 \leq n \leq (\text{length } lst)$, le procedure `primielementi` e `ultimielementi` possono essere usate per dividere la lista `lst` in due parti. Una volta che abbiamo diviso il nostro mazzo in due metà, dobbiamo combinarle per ottenere il mazzo mescolato. Le combiniamo usando la funzione `interleave`, che prende due liste e le combina in una singola lista alternandone gli elementi:

```
# let rec interleave lst1 lst2 =
  match lst1 with
  | [] -> lst2
  | x1::xs1 -> x1::(interleave lst2 xs1);;
interleave : 'a list -> 'a list -> 'a list = <fun>
```

Cerchiamo di capire perchè la procedura funziona correttamente: il primo elemento della lista risultante sarà il primo elemento di `lst1` (ovvero il primo elemento del primo argomento). Com'è il resto della lista risultante, dopo quel primo elemento? Se si alterna un mazzo di carte rosse con un mazzo di carte nere in modo che la prima carta sia rossa, come è formato il resto delle carte (escludendo la prima)? Il resto comincerà con una carta nera e poi alternerà i colori. Includerà tutte le carte nere e il resto delle rosse. In altre parole, è il risultato dell'alternanza tra le carte nere e il resto delle rosse. Questo spiega perchè nella chiamata ricorsiva alla funzione `interleave` passiamo `lst2` come primo argomento (in modo che il primo elemento di `lst2` finisce subito dopo il primo elemento di `lst1` nel risultato) e poi la coda di `lst1`.

Combinando le funzioni `interleave`, `primielementi` e `ultimielementi` possiamo definire la funzione `shuffle`, che prende come argomento il mazzo e la sua dimensione:

```
# let shuffle deck size =
  let half = (size+1)/2
in interleave (primielementi half deck) (ultimielementi half deck);;
shuffle : 'a list -> int -> 'a list = <fun>
```

Lo scopo del parametro `size` è l'efficienza (altrimenti dovremmo usare la procedura `length`). Scriviamo `(size+1)/2` invece del più naturale `size/2` per assicurarci che, quando `size` è dispari, sia la prima metà delle carte ad avere la carta in più. Da notare che quando la dimensione del mazzo è 52, `(size+1)/2=26`.

Per scoprire quanti mescolamenti sono necessari, scriviamo la seguente funzione, che automatizza diversi mescolamenti:

```
# let rec multiple_shuffle deck size times =
  if times = 0 then deck
  else multiple_shuffle (shuffle deck size) size (times - 1);;
multiple_shuffle : 'a list -> int -> int -> 'a list = <fun>
```

Possiamo scoprire quanti mescolamenti servono con chiamate di questo tipo:

```
# (multiple_shuffle (interidaa 1 52) 52 1);;
- : int list =
[1; 27; 2; 28; 3; 29; 4; 30; 5; 31; 6; 32; 7; 33; 8; 34; 9; 35; 10;
36; 11; 37; 12; 38; 13; 39; 14; 40; 15; 41; 16; 42; 17; 43; 18; 44;
19; 45; 20; 46; 21; 47; 22; 48; 23; 49; 24; 50; 25; 51; 26; 52]

# (multiple_shuffle (interidaa 1 52) 52 2);;
- : int list =
[1; 14; 27; 40; 2; 15; 28; 41; 3; 16; 29; 42; 4; 17; 30; 43; 5; 18;
31; 44; 6; 19; 32; 45; 7; 20; 33; 46; 8; 21; 34; 47; 9; 22; 35; 48;
10; 23; 36; 49; 11; 24; 37; 50; 12; 25; 38; 51; 13; 26; 39; 52]
...
```

```
# (multiple_shuffle (interidaa 1 52) 52 8);;
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19;
20; 21; 22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33; 34; 35; 36;
37; 38; 39; 40; 41; 42; 43; 44; 45; 46; 47; 48; 49; 50; 51; 52]
```

Vediamo quindi che con soli 8 “mescolamenti perfetti”, il mazzo ritorna nel suo ordine originale.

Abbiamo scritto la funzione `shuffle` in modo che possa operare su mazzi di qualunque dimensione. In effetti, la proprietà che dopo un certo numero di “mescolamenti perfetti” il mazzo ritorni al suo ordine originale vale per i mazzi di tutte le dimensioni.

Scriviamo ora una funzione che automatizza il processo di trovare il numero di “mescolamenti perfetti” richiesti per un mazzo di carte di dimensione fissata. Dato che si parte con un mazzo ordinato, abbiamo bisogno prima di tutto di un predicato (cioè di una funzione che restituisce `true` o `false`), chiamata `in_order`, che determini se una lista di interi è ordinata secondo l’ordine crescente.

```
# let rec in_order lst =
  match lst with
  | [] -> true
  | [x] -> true
  | x::y::xs -> (x<=y & in_order (y::xs));;
in_order : 'a list -> bool = <fun>
```

Usando il predicato `in_order`, scriviamo la funzione `shuffle_number` che, quando riceve in input un mazzo di carte e la sua dimensione `n`, restituisce il “numero di mescolamenti perfetti” per riportare il mazzo di carte nel suo ordine originale. Si inizia con un mazzo ordinato di dimensione `n` e lo si mescola finché non torna in ordine.

```
# let rec shuffle_number deck size =
  let shuffle_deck = (shuffle deck size)
  in
    if (in_order shuffle_deck) then 1
    else 1 + (shuffle_number shuffle_deck size);;
shuffle_number : 'a list -> int -> int = <fun>
```

7.5 – Iterazione e liste

Una parola si dice *palindroma* quando rimane invariata se letta a rovescio: ad esempio le parole *anna*, *osso*, *otto*, *madam* sono palindromi. A volte, intere frasi sono palindromi, se si ignorano gli spazi e la punteggiatura. Ad esempio: “*Madam, I’m Adam*”, “*E la sete sale*”, “*I topi non avevano nipoti*”. In questa sezione testeremo liste di simboli per vedere se sono palindrome considerandole simbolo per simbolo piuttosto che lettere per lettera. Quello che intendiamo è che invertendo l’ordine degli elementi la lista rimane invariata. Per esempio, la lista `["m"; "a"; "d"; "a"; "m"]` è palindroma, così come `["anna"; "ed"; "anna"]`.

Possiamo determinare se una lista di simboli è palindroma invertendola e controllando se il risultato è uguale alla lista originale. Dobbiamo quindi trovare un modo per invertire la lista. Esiste una

funzione predefinita del linguaggio che fa esattamente quello che ci serve, ovvero invertire la lista: la funzione `rev`:

```
# rev;;
- : 'a list -> 'a list = <fun>
```

Tuttavia, cosa impareremmo se usassimo tutto ciò che ci viene già fornito? Ben poco, quindi scriveremo da noi questa funzione. Tra l'altro, invertire una lista è molto semplice. Per invertire una lista non vuota, un approccio prevede di invertire la coda della lista e “agganciarci in fondo” la testa. L'ostacolo principale è che non abbiamo al momento alcuna funzione che ci “appenda” un elemento alla fine della lista. Vogliamo costruire una funzione ricorsiva `add_to_end` tale che la chiamata `add_to_end [1;2;3] 4` valuti alla lista `[1;2;3;4]`. Possiamo scrivere questa procedura nel solito modo ricorsivo, scorrendo la lista e agganciando l'elemento solo alla fine:

```
# let rec add_to_end lst elt =
  match lst with
  | [] -> elt::[]
  | x::xs -> x::(add_to_end xs elt);;
add_to_end : 'a list -> 'a -> 'a list = <fun>
```

Data questa funzione, possiamo scrivere la funzione che inverte una lista come segue:

```
# let rec reverse lst =
  match lst with
  | [] -> []
  | x::xs -> add_to_end (reverse xs) x;;
reverse : 'a list -> 'a list = <fun>
```

Questo modo di invertire una lista è molto poco efficiente a causa della chiamata a `add_to_end`. Un buon metodo per misurare il tempo di esecuzione della funzione è quello di contare quante volte viene chiamato il costruttore `::`.

Aggiungere un elemento in coda ad una lista di k elementi farà $(k+1)$ chiamate al costruttore `::`. Supponiamo di usare il simbolo $R(n)$ per denotare il numero di costruttori che la funzione `reverse` usa (indirettamente tramite `add_to_end`) quando inverte una lista di dimensione n . Sappiamo che $R(0) = 0$ poiché quando l'argomento è la lista vuota, la funzione `reverse` ritorna la lista vuota. Quando l'argomento è una lista non vuota, il numero di chiamate totali al costruttore saranno tutte quelle fatte per invertire la coda della lista, più tutte quelle fatte per aggiungere la testa alla fine della coda invertita. Dunque:

$$R(n) = R(n-1) + ((n-1) + 1) = R(n-1) + n$$

Ma allora allo stesso modo abbiamo che:

$$R(n-1) = R(n-2) + ((n-2) + 1) = R(n-2) + (n-1)$$

E così otteniamo:

$$\begin{aligned}
R(n) &= R(n-1) + n \\
&= R(n-2) + (n-1) + n \\
&\vdots \\
&= R(0) + 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n \\
&= 0 + 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n \\
&= \frac{n(n+1)}{2}
\end{aligned}$$

Quindi il numero di costruttori chiamati da questa versione della funzione `reverse` è $\Theta(n^2)$. Ciò significa che man mano che la lista diventa più lunga, invertirla costa sempre di più rispetto all'aumento della dimensione: una lista di 200 elementi ci metterà 4 volte tanto a invertirsi rispetto ad una lista di 100 elementi, pur essendo solo lunga il doppio.

Deve esserci un modo migliore per invertire una lista. Se vi ricordate l'esercizio 7.3, il nostro tentativo iniziale di scrivere una funzione iterativa che generasse la lista di interi da `a` a `b` produceva la lista con i numeri giusti, ma in ordine inverso. Nonostante in quell'esempio ci fosse un errore, quella funzione ci suggerisce la strategia iterativa per invertire una lista.

Prima di buttarci a scrivere la versione iterativa della funzione `reverse`, facciamo un esempio. Metti una pila di carte sul tavolo rivolte verso l'alto e invertine l'ordine lasciandole rivolte verso l'alto. Probabilmente lo hai fatto togliendo la prima carta dalla pila e appoggiandola lì vicino sul tavolo, poi spostando la carta successiva dalla cima della pila originale alla cima della nuova pila, etc, fino a che tutte le carte sono state spostate. Se ad un certo punto si chiedesse a qualcun altro di finire il lavoro, gli si potrebbe dire "inverti il resto di queste carte su quell'altra pila".

Possiamo dunque ridurre il problema di invertire la lista `[1; 2; 3; 4]` al problema più piccolo di mettere gli elementi di `[2; 3; 4]` invertiti davanti a `[1]`, che a sua volta si riduce al mettere gli elementi invertiti di `[3; 4]` davanti a `[2; 1]`, etc:

```
# let rec reverse lst =
  let rec reverse_onto lst1 lst2 =
    match lst1 with
    | [] -> lst2
    | x::xs -> reverse_onto (xs) (x::lst2)
  in reverse_onto lst [];
reverse : 'a list -> 'a list = <fun>
```

La funzione `reverse_onto` restituisce gli elementi della lista `lst1` in ordine invertito seguiti dagli elementi della lista `lst2`. Tale funzione usa un costruttore per ogni elemento della lista, mentre si muove dall'inizio di `lst1` all'inizio di `lst2`. Notate che il numero totale di costruttori usati è `n`, dove `n` è la dimensione della lista in input. Abbiamo dunque ridotto un processo $\Theta(n^2)$ a uno $\Theta(n)$. Ora abbiamo tutte le funzioni necessarie per determinare se una lista è palindroma oppure no:

```
# let palindrome l = (l = (reverse l));;
palindrome : 'a list -> bool = <fun>

# palindrome ['m'; 'a'; 'd'; 'a'; 'm'; 'i'; 'm'; 'a'; 'd'; 'a'; 'm'];;
- : bool = true
```

Definiamo ora la funzione iterativa `upto` che, dati due numeri interi `m` e `n`, restituisce la lista composta da tutti gli interi compresi tra `m` e `n`, estremi compresi. Questa funzione è la versione iterativa della funzione `interidaa` vista sopra.

```
# let upto (m,n) =
    let rec aux_upto (m,n,acc) =
        if m>n then acc
        else aux_upto (m, n-1, n::acc)
    in aux_upto (m,n,[]);
upto : int * int -> int list = <fun>

#upto (0,5);;
- : int list = [0; 1; 2; 3; 4; 5]
```

La seguente funzione `last` restituisce l'ultimo elemento di una lista.

```
# let rec last lst =
    match lst with
    [x] -> x
    | _ :: xs -> last xs;;
last : 'a list -> 'a = <fun>

# last [1;2;3];
- : int = 3
```

Definiamo ora la funzione `maxinlist` che, data una lista, restituisce l'elemento maggiore fra tutti quelli della lista. Usiamo la funzione predefinita `max` che restituisce il massimo tra due valori.

```
# let rec maxinlist lst =
    match lst with
    [x] -> x
    | x::y::xs -> maxinlist ((max x y)::xs);;
maxinlist : 'a list -> 'a = <fun>

# maxinlist [4;6;5;8];
- : int = 8
```

Alternativamente, possiamo scrivere:

```
# let rec maxinlist lst =
    match lst with
    [x] -> x
    | x::y::xs -> max x (maxinlist xs);;
maxinlist : 'a list -> 'a = <fun>
```

7.6 – Ricorsione ad albero e liste

In questa sezione, vedremo due esempi di ricorsione ad albero sulle liste. Il primo esempio è una definizione della funzione `mergesort`. L'approccio di base consiste nel separare la lista iniziale in due liste più piccole, ordinarle tramite la funzione `mergesort`, e infine riunire (`merge`) le due liste ordinate in una unica. Possiamo separare una lista in due solamente se ha almeno 2 elementi, ma fortunatamente tutte le liste vuote e quelle con un solo elemento sono già ordinate.

Quindi la nostra funzione `mergesort` apparirà così:

```
# let rec mergesort lst =
  match lst with
  | [] -> []
  | [x] -> lst
  | _ ->
    merge (mergesort (onepart lst)) (mergesort (theotherpart lst));;
```

Dobbiamo ancora fare la maggior parte del lavoro, ovvero scrivere la funzione `merge` e trovare un modo per spezzare la lista in due parti, le quali per motivi di efficienza dovrebbero essere di dimensione uguale (o il più simili possibile).

Cominciamo con la procedura `merge`. Abbiamo due liste di numeri, e ciascuna delle due è ordinata dal più piccolo al più grande. Vogliamo produrre una terza lista, che comprenda tutti gli elementi delle due liste di partenza e che sia ancora ordinata. Notiamo che ci sono essenzialmente due casi base, ovvero quando una o l'altra delle due liste originali è vuota. In ognuno dei due casi, il risultato che vogliamo ritornare è l'altra lista (possibilmente non vuota). Abbiamo poi due casi ricorsivi, a seconda della relazione tra il primo elemento della prima lista e quello della seconda lista. Vogliamo sempre agganciare l'elemento più piccolo al risultato del merging tra la coda della lista da cui tale elemento proveniva e l'altra lista.

```
# let rec merge lst1 lst2 =
  match (lst1,lst2) with
  | ([],_) -> lst2
  | (_,[]) -> lst1
  | (x1::xs1, x2::xs2) ->
    if (x1<=x2) then
      x1::(merge xs1 lst2)
    else
      x2::(merge lst1 xs2);;
merge : 'a list -> 'a list -> 'a list = <fun>
```

Vediamo un esempio di utilizzo:

```
# merge [1;3;5;6;8;19] [2;3;4;6;9;11];;
- : int list = [1; 2; 3; 4; 5; 6; 8; 9; 11; 19]
```

Ora che abbiamo risolto il primo problema, tocca al secondo: spezzare la lista in due metà. Un modo per farlo potrebbe essere quello di usare le funzioni `primielementi` e `ultimielementi` che abbiamo definito nel problema dei mescolamenti perfetti. Il problema di questo approccio è che ci servirebbe conoscere la lunghezza della lista che stiamo cercando di ordinare. È vero, potremmo usare la funzione `length` per trovarla: tuttavia vogliamo mostrare un metodo alternativo per separare la lista in due parti, che assomiglia al problema dell'*interleaving* (l'alternanza). In altre parole, se pensiamo alle nostra lista come a un mazzo di carte, possiamo separarla a metà distribuendo le carte a due persone. Una persona si prenderebbe le carte prima, terza, quinta, ..., mentre l'altro si prenderebbe la seconda, la quarta, la sesta, etc. Possiamo chiamare le due mani di carte risultanti come *odd part* (parte dispari) e *even part* (parte pari).

Per scrivere le funzioni `oddpart` e `evenpart`, pensiamo a come distribuiremmo le carte a due persone, che chiamiamo Alice e Bob. Si comincia guardando Alice, alla quale daremo la parte dispari delle carte, mentre a Bob daremo la parte pari. In altre parole, daremo le carte dispari alla persona che stiamo guardando. Daremo ad Alice la prima carta; a quel punto ci gireremo verso Bob, tenendo il resto delle carte in mano. In questo momento la situazione è esattamente uguale a quella iniziale, tranne per il fatto che siamo di fronte a Bob e abbiamo una carta di meno. Stiamo per dare a Bob la prima carta, la terza, etc delle carte rimanenti, mentre ad Alice stiamo per dare quelle numerate pari. Quindi, il mazzo di Alice (la parte dispari del mazzo) consiste della prima carta più la parte pari delle carte rimanenti. Invece il mazzo di Bob (cioè la parte pari del mazzo) consiste della parte dispari di ciò che rimane del mazzo dopo aver consegnato la prima carta ad Alice.

Le due funzioni `oddpart` e `evenpart`, ci forniscono un interessante esempio di *mutua ricorsione*, perché possiamo facilmente definire l'una in termini dell'altra:

```
# let rec
  oddpart (ls) =
    if ls = [] then []
    else hd(ls) :: evenpart(tl(ls))
and
  evenpart (ls) =
    if ls = [] then []
    else oddpart(tl(ls)) ;;
oddpart : 'a list -> 'a list = <fun>
evenpart : 'a list -> 'a list = <fun>
```

Da notare bene l'uso della keyword `and`: se non la usassimo non riusciremmo a definire queste due funzioni, perché ciascuna delle due richiama l'altra e quindi non sapremmo quale scrivere per prima. Grazie alla parola chiave `and`, le stiamo definendo contemporaneamente e questo fa sì che possano contenere riferimenti l'una verso l'altra (che è l'idea alla base della mutua ricorsione).

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

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

Ora per far funzionare la funzione `mergesort` ci basta che porre:

```
# let onepart = oddpart;;
onepart : 'a list -> 'a list = <fun>

# let theotherpart = evenpart;;
theotherpart : 'a list -> 'a list = <fun>
```

Proviamo ad usare la nostra procedura `mergesort`:

```
# mergesort [4;3;2;7;6;5;9;1;45;8;12;1;90;2];;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 12; 45; 90]
```

Vediamo un altro esempio di ricorsione ad albero. Una famiglia decide di portare i propri figli (di 3 e 4 anni) in una sala giochi a giocare ad un gioco chiamato *Whacky Gator*. Si vincono dei biglietti che possono poi essere scambiati in premi al bancone. Ogni tipo di premio ha un prezzo. Alcuni premi

valgono 10 biglietti, altri valgono 9 biglietti, e così via fino ai ragnetti di plastica che valgono solo 1 biglietto. Il bambino più grande ha vinto 10 biglietti e vuole sapere quali premi può prendere. Ovviamente può prendere solo uno dei premi che valgono 10 biglietti; in alternativa, però, può prendere un premio che vale 9 biglietti e uno che vale 1 biglietto; o ancora può prendere un premio che vale 8 biglietti e due che valgono 1 biglietto oppure un premio che vale 8 biglietti e uno che vale 2 biglietti, ecc... Mentre la madre del bambino tenta di enumerare tutte le possibili combinazioni, abbiamo deciso di risolvere il problema in modo esatto, ovvero di scrivere una funzione che calcola il numero di possibili combinazioni. Abbiamo scoperto che ci sono 1778 possibili combinazioni di premi tra cui il bambino può scegliere, dato il numero di premi che sono disponibili (vedere tabella 7.1).

Valore in biglietti	Numeri di premi diversi
10	9
9	3
8	2
7	4
6	3
5	4
4	3
3	3
2	4
1	2

Tabella 7.1 – I premi della sala giochi

Come ci siamo riusciti? Abbiamo deciso di scrivere una funzione generale per capire quante combinazioni di premi ci sono. Per capire come abbiamo sviluppato tale funzione, consideriamo un problema più piccolo. Supponiamo che ci siano solamente due tipi di premi che valgono 1 biglietto (ragni di plastica e scarafaggi di plastica). Poi ci sono i vermi di plastica che valgono ben 3 biglietti e delle piccole lenti d'ingrandimento del valore di 5 biglietti. Se il bambino avesse 5 biglietti, potrebbe prendere oppure no la lente di ingrandimento. Se non la prendesse, allora dovrebbe prendere una qualche combinazione dei premi rimanenti, che valga comunque 5 in totale. Se invece la prendesse, avrebbe usato tutti i suoi biglietti e ci sarebbe solamente una combinazione di altri premi che valgano i rimanenti 0 biglietti (ossia la combinazione vuota).

Questo approccio ci mostra come ridurre il nostro problema in problemi più piccoli. Supponiamo che i premi siano rappresentati da una lista con i loro valori; nel nostro piccolo esempio qui sopra, dovremmo usare la lista [5; 3; 1; 1] per rappresentare la lente d'ingrandimento, i vermi, i ragni e gli scarafaggi. In questo caso, se il bambino avesse un certo numero di biglietti, potrebbe prendere una combinazione che include il primo elemento della lista, o una che non lo include. Se il primo elemento viene scelto, dobbiamo contare il numero di combinazioni che può ottenere tramite il suo numero di biglietti meno il valore in biglietti del primo elemento. Se invece il primo elemento non viene scelto, allora dobbiamo contare le combinazioni possibili di oggetti che si possono ottenere per la quantità di biglietti posseduta dal bambino usando solamente il resto della lista. Quali sono i casi base? Per trovarli, notiamo che il nostro problema diventa più piccolo in uno dei due parametri: o la lista di premi diventa più piccola oppure diminuisce la quantità di biglietti,

perché tutti i premi hanno prezzi positivi. Quindi il processo dovrebbe fermarsi quando l'ammontare di biglietti è 0, meno di 0, oppure quando la lista di premi è vuota.

```
# let rec count_combos prize_list amount =
  match (prize_list, amount) with
  | (_, 0) -> 1
  | ([], _) -> 0
  | (_, amount) when amount < 0 -> 0
  | (_, _) -> (count_combos prize_list (amount - hd(prize_list)))
+ (count_combos (tl(prize_list)) amount);;
count_combos : int list -> int -> int = <fun>
```

Per vedere se davvero ci sono 1778 possibili combinazioni che valgono i 10 biglietti, dobbiamo inserire una lista di 37 numeri.

```
#count_combos[10;10;10;10;10;10;10;10;10;10;9;9;9;8;8;7;7;7;7;6;6;6;5;5
;5;5;4;4;4;3;3;3;2;2;2;2;1;1] 10;;
- : int = 1778
```

Alternativamente possiamo scrivere una funzione che ci generi questa lista, a partire dai dati della tabella. Qual'è il modo migliore per passargli tali dati? Un modo potrebbe essere una lista di coppie, in cui il primo numero di ogni coppia è il valore e il secondo è il numero di premi distinti di quel valore. Un altro modo potrebbe essere di dare il valore del premio più costoso e poi dare la lista dei numeri di premi differenti, dove il primo numero rappresenta il numero di premi che valgono di più, il secondo rappresenta il numero di premi che valgono un biglietto in meno, e così via.

ESERCIZIO 7.11:

Scrivere la funzione che generi, a partire dai dati della tabella 7.1, la lista da passare come parametro alla funzione `count_combos`. Usala per verificare che effettivamente ci sono 1778 possibili combinazioni che valgono 10 biglietti.

ESERCIZIO 7.12:

Scrivere una funzione che, data una lista di premi e un ammontare di biglietti, calcola il numero di combinazioni di premi che si possono ottenere usando esattamente quell'ammontare e assumendo che non si possa prendere più di un premio per ogni tipologia di premio.

ESERCIZIO 7.13:

Uno dei bambini ha scoperto che non deve necessariamente spendere tutti i suoi biglietti, perché li può tenere per la prossima volta. Quindi, invece di trovare i numeri di combinazioni che si ottengono con un certo ammontare di biglietti, vorrebbe sapere il numero di combinazioni che può ottenere per un qualunque ammontare di biglietti minore o uguale a quello che possiede.

Scrivere una funzione che, data una lista di premi e un ammontare massimo di biglietti, ritorna il numero di combinazioni di premi che si possono ottenere usando non più dell'ammontare di biglietti.

ESERCIZIO 7.14:

Scrivere un'altra funzione che trovi il numero di combinazioni che si possono ottenere usando non più di un certo numero di biglietti, sempre con il vincolo però di potere prendere al massimo un premio per ogni tipologia.

ESERCIZIO 7.15:

Un problema simile consiste nel considerare una quantità illimitata di monete, dove le monete sono di 4 tipi diversi: *quarters* (25 cent), *dimes* (10 cent), *nickels* (5 cent) e *pennies* (1 cent). Trovare una combinazione di queste monete per raggiungere una certa cifra. In quanti modi diversi si può raggiungere lo scopo? Scrivere una funzione che conti il numero di modi in cui si può dare il cambio di una certa cifra usando solo *quarters*, *dimes*, *nickels* e *pennies*.

7.7 – Esercizi di riepilogo

ESERCIZIO 7.16:

Dimostrare per induzione su n che la funzione seguente produce una lista di lunghezza n :

```
# let rec sevens n =
  if n = 0 then []
  else 7::(sevens (n-1));;
```

ESERCIZIO 7.17:

Scrivere una versione iterativa della seguente funzione:

```
# let rec squaresum lst =
  match lst with
  | [] -> 0
  | x::xs -> (square x) + (squaresum xs);;
```

ESERCIZIO 7.18:

Dimostrare per induzione su n che la funzione seguente produce una lista di 2^n 17:

```
# let rec seventeens n =
  if n = 0 then []
  else 17::(17::(seventeens (n-1))));;
```

ESERCIZIO 7.19:

Considera le seguenti due funzioni. La funzione `last` seleziona l'ultimo elemento da una lista non vuota. Usa la funzione `length` per trovare la lunghezza della lista.

```
# let rec length lst =
  match lst with
  | [] -> 0
  | _ -> 1 + (length (tl(lst))));;
```

```
# let rec last lst =
    if (length lst) = 1 then hd(lst)
    else last (tl(lst));;
```

- Quanti `tl` vengono applicati da `length lst` quando la lista `lst` ha `n` elementi?
- Quante chiamate a `length` vengono effettuate da `last lst` quando la lista `lst` ha `n` elementi?
- Esprimere in notazione Θ il numero totale di `tl` effettuati da `last lst`, includendo quelli effettuati da `length`, sempre assumendo che la lista `lst` abbia `n` elementi.
- Scrivere la formula esatta che esprima il numero totale di `tl` effettuati da `last lst`, includendo quelli effettuati da `length`, assumendo che la lista abbia `n` elementi.

ESERCIZIO 7.20:

Considerare la seguente funzione:

```
# let rec repeat num times =
    if times = 0 then []
    else num::(repeat num (times-1));;

# repeat 3 2;;
- : int list = [3; 3]

# repeat 17 5;;
- : int list = [17; 17; 17; 17; 17]
```

- Spiegare perché `repeat` genera un processo ricorsivo.
- Scrivere una versione iterativa di `repeat`.

ESERCIZIO 7.21

Utilizzando la funzione `sposta` definita nel paragrafo 4.3, definire una funzione

```
esegui: posizione * azione list -> posizione
```

che, applicata a una posizione e una lista di azioni `[a1; a2; ... ; an]` restituisca la posizione in cui si trova l'oggetto che, trovandosi inizialmente nella posizione data, esegue in sequenza (e in quest'ordine) le azioni `a1; a2; ... ; an`.

Ad esempio, si deve avere:

```
# let p = Pos(0,0,Su);;
p : posizione = Pos (0, 0, Su)

# esegui (p, [Avanti 3; Gira]);;
-: posizione = Pos(0,3,Destra)

# esegui (p, [Avanti 3; Gira; Avanti 2]);;
-: posizione = Pos(2,3,Destra)
```

```
# esegui (p, [Avanti 2; Gira; Avanti 3]);
:- posizione = Pos(3,2,Destra)

# esegui (p, [Avanti 2; Gira; Gra; Gira; Avanti 3]);
:- posizione = Pos(-3,2,Sinistra)
```

7.8 – Liste ed eccezioni

Se una funzione non è definita per tutti i suoi argomenti, è opportuno fare in modo che, se richiamata con valori al di fuori del suo dominio, segnali un errore, piuttosto che causare un comportamento non prevedibile o addirittura generare un loop infinito. A questo scopo sia ML che F# dispongono di un tipo particolare di dati, *le eccezioni*, che possono essere usate come argomento o valore di qualsiasi funzione. Le eccezioni vengono dichiarate nel seguente modo:

```
# exception NomeEccezione;;
```

Per far sì che una funzione riporti un'eccezione, occorre “sollevare” (*to raise*) l'eccezione, mediante la parola chiave `raise`. Quest'ultima è una funzione che data una espressione restituisce un valore (eccezione) di tipo polimorfo `'a`.

```
#raise;;
- : exn -> 'a = <fun>
```

Nel codice seguente, la funzione fattoriale è richiamata con un argomento negativo:

```
# (* Eccezione non dichiarata -> out_of_memory *)
let rec fatt n = match n with
  0 -> 1
  | n -> n * (fatt n-1);;
fatt : int -> int = <fun>

# fatt (0);;
- : int = 1

# fatt (-1);;
Uncaught exception: Out_of_memory

# (* Eccezione dichiarata -> la funzione termina *)
# exception Neg_argument;;
Exception Neg_argument defined.

# let fatt n =
  let rec p_fatt n =
    match n with
    0 -> 1
    | n -> n * (p_fatt n-1)
  in if n >= 0
    then p_fatt n
    else raise Neg_argument;;
fatt : int -> int = <fun>
```

```
# fatt (-1);;
Uncaught exception: Neg_argument
```

La valutazione di un'espressione viene immediatamente interrotta se al suo interno viene sollevata un'eccezione, come nel caso seguente:

```
# 4 * fatt (-5) +1;;
Uncaught exception: Neg_argument
```

Tale comportamento è noto come *propagazione automatica delle eccezioni* e può essere evitato tramite un *exception handler* (cattura di un'eccezione) specificando che, se una determinata espressione non solleva alcuna eccezione allora si riporta il valore dell'espressione stessa, altrimenti, se la valutazione solleva un'eccezione, allora si deve riportare un altro valore. La sintassi è:

```
try Espessione_1 with Exception -> Espessione_2
```

Le due espressioni devono essere dello stesso tipo: se la prima solleva l'eccezione specificata con il with, allora viene riportato il valore della seconda espressione, altrimenti quello della prima, a condizione che non venga sollevata un'eccezione diversa da quella specificata. Infatti, si ha:

```
# let twofatt (n, m) =
    try fatt(n) * fatt(m) with Neg_argument -> 0;;
twofatt : int * int -> int = <fun>

# twofatt (2, -3);;
- : int = 0

# twofatt (2, 3);;
- : int = 12
```

LE ECCEZIONI IN F#: In F# i nomi delle eccezioni devono contenere solo lettere maiuscole.

Quindi, nell'esempio precedente avremmo dovuto scrivere:

```
# exception NEG_ARGUMENT;;
```

Vediamo ora come si possono definire alcune funzioni sulle liste usando le eccezioni.

Le funzioni hd e tl possono essere definite anche sulla lista vuota nel seguente modo:

```
# exception EmptyList;;
Exception EmptyList defined.

# let hd lst = match lst with
    [] -> raise EmptyList
    | x::_ -> x;;
hd : 'a list -> 'a = <fun>

# let tl lst = match lst with
    [] -> EmptyList
    | _::xs -> xs;;
tl : 'a list -> 'a list = <fun>
```

Scriviamo la funzione position che data una lista lst e un valore x, restituisca la posizione della prima occorrenza di x in lst (contando da 0), se x sta in lst, oppure sollevi un'eccezione.

```
# exception NotMember;;
Exception NotMember defined.

# let position (xs,m) =
  let rec aux (n, xs)= match (n, xs) with
    (_, []) -> raise NotMember
    | (n, x::xs) -> if x=m then n else aux (n+1, xs)
  in aux(0, xs);;
position : 'a list * 'a -> int = <fun>

# position ([1;24;3;7;8;45], 8);;
- : int = 4

# position ([] , 8);;
Uncaught exception: NotMember

# position ([5;7;6], 8);;
Uncaught exception: NotMember
```

Scriviamo un programma che, data una lista di liste di interi, restituisca il valore minimo tra i massimi di ciascuna lista.

```
# exception ListaVuota;;
Exception ListaVuota defined.

# let rec max lst =
  match lst with
  [] -> raise ListaVuota
  | [x] -> x
  | x::y::ys -> if (x>y) then max (x::ys) else max (y::ys);;
max : 'a list -> 'a = <fun>

# let rec min lst =
  match lst with
  [] -> raise ListaVuota
  | [x] -> x
  | x::y::ys -> if (x<y) then min (x::ys) else min (y::ys);;
min : 'a list -> 'a = <fun>

# let rec maxmin lst =
  match lst with
  [] -> raise ListaVuota
  | [x] -> max x
  | x::y::ys -> min [(max x); (maxmin (y::ys))];;
maxmin : 'a list list -> 'a = <fun>

# maxmin [[3;100;1;9]; [2;10;20]; [80;65;4]];;
- : int = 20
```

Capitolo 8 – Funzioni di ordine superiore

8.1 – Funzioni come parametri

Nei capitoli precedenti abbiamo visto come si possono raggruppare varie espressioni specifiche, che differiscono solo per alcuni dettagli, in un'unica espressione più generale:

- Nel primo capitolo, abbiamo introdotto le funzioni. Esse ci consentono di raggruppare, in un'unica definizione generale, varie espressioni che differiscono solo per i valori specifici su cui operano, come: `3 * 3 e 4 * 4 e 5 * 5`. Per esempio, si consideri

```
# let square x = x * x;;
```

Questa unica funzione può essere usata per rappresentare tutte le espressioni specifiche citate sopra; la funzione `square` specifica l'operazione da eseguire, e il parametro ci permette di variare il valore su cui l'operazione viene eseguita.

- Nel secondo capitolo, abbiamo imparato a generare processi computazionali di dimensione variabile. In questo modo, quando ci troviamo di fronte a varie funzioni che generano processi della stessa forma, ma diversi in quanto a dimensione, come

```
# let square x = x * x;;
# let cube x = x * x * x;;,
```

abbiamo visto come definire una procedura generale che li raggruppa in un'unica definizione:

```
# let rec power b e =
  if e = 1 then b
  else b * (power b (e-1));;
```

Questa unica funzione può essere usata al posto delle funzioni più specifiche `square` e `cube`; la funzione `power` continua a specificare quali operazioni vanno effettuate, ma i parametri specificano quante operazioni fare e i valori su cui farle.

In questo capitolo vedremo un terzo modo per raggruppare varie definizioni specifiche in una singola definizione più generale.

Ipotizziamo di sostituire, nella definizione precedente di `power`, l'operazione `*` con l'operazione `+`. Facendo questo cambiamento, ci troveremmo con una funzione che calcola il prodotto di due numeri invece dell'elevamento a potenza (certo, poi sarebbe anche il caso di cambiare il nome della funzione e dei parametri per renderla più leggibile). La struttura generale delle due funzioni è la stessa: vengono combinate (con la moltiplicazione in un caso e con la somma nell'altro) e copie dell'oggetto `b`; l'unica differenza è la specifica operazione usata (moltiplicazione/somma).

La somiglianza di struttura tra le due funzioni pone un'interessante domanda: possiamo scrivere una funzione generale per tutte le computazioni di questo tipo e poi specificare non solo *quante* copie vogliamo di *quale* oggetto ma anche *come* devono essere combinate? Se così fosse, potremmo chiederle di sommare 5 copie di 2, di moltiplicare 3 copie di 8, etc... Potremmo usarla in questo modo:

```
# combina_copie_di (prefix +) 5 2;;
10

# combina_copie_di (prefix *) 8 3;;
512
```

Il primo argomento è una funzione, tramite cui specifichiamo come vogliamo combinare i valori tra loro. I nomi `(prefix +)` e `(prefix *)` sono valutati, proprio come verrebbe valutata qualunque altra espressione. Quindi, il vero e proprio valore dell'argomento è la funzione, non il suo nome.

Per cominciare a scrivere la procedura `combina_copie_di`, diamo un nome ai parametri come segue:

```
# let combina_copie_di combina valore quantita
```

Qui abbiamo 3 parametri, chiamati `combina`, `valore` e `quantita` ottenuti dalla frase “combina una certa quantità di copie di un certo valore”. Abbiamo scelto di usare un verbo per il parametro che rappresenta l'operazione con cui vengono combinati i valori e nomi per gli altri parametri per ricordarci il modo in cui sono usati. Ora possiamo finire di scrivere la funzione `combina_copie_di`, usando i nomi dei parametri nel corpo della funzione ogni volta che ci serve averne sostituito il valore specifico. Per esempio, quando vogliamo controllare se la quantità specificata è uguale a 1, scriviamo `quantita = 1`. Allo stesso modo, quando vogliamo usare la specifica operazione di combinazione richiesta, scriviamo `(combina)`. Ecco la funzione risultante:

```
# let rec combina_copie_di combina valore quantita =
  if quantita = 1 then valore
  else (combina valore (combina_copie_di combina valore (quantita-1)));;
combina_copie_di : ('a -> 'a -> 'a) -> 'a -> int -> 'a = <fun>

# combina_copie_di (prefix +) 5 7;;
- : int = 35

# combina_copie_di (prefix *) 3 2;;
- : int = 9

# combina_copie_di (prefix *) 2 3;;
- : int = 8

# combina_copie_di (prefix *) 7 5;;
- : int = 16807
```

Una volta scritta questa procedura a scopo generale, possiamo usarla per semplificare la definizione di altre procedure:

```
# let mul a b = combina_copie_di (prefix +) a b;;
# let power b e = combina_copie_di (fun a b -> a * b) b e;;
```

La funzione `combina_copie_di` è un esempio di funzione di *ordine superiore (higher order)*. Tali funzioni prendono funzioni come parametri oppure (come vedremo più avanti) restituiscono funzioni. Un vantaggio di usare la funzione di ordine superiore `combina_copie_di` è che possiamo migliorare la

tecnica usata in `combina_copie_di` e in un solo colpo miglioreremmo la performance di `power` e qualunque altra procedura che usa `combina_copie_di`.

ESERCIZIO 8.1

Scrivere una versione iterativa della funzione `combina_copie_di`.

Sviluppiamo ora un altro esempio. Si noti che contare il numero di volte in cui la cifra 6 compare in un certo numero è molto simile a contare il numero di cifre dispari presenti in un numero. Nel primo caso stiamo testando ciascuna cifra per vedere se è uguale a 6, nel secondo caso stiamo testando ciascuna cifra per vedere se è dispari. Dunque possiamo scrivere una funzione generale `num_cifre_che_soddisfano`, che possiamo poi usare per definire entrambe le funzioni citate. Il suo secondo parametro è il particolare predicato di test da usare su ciascuna cifra.

```
# let rec num_cifre_che_soddisfano n test =
  if n < 0 then num_cifre_che_soddisfano (-n) test
  else if n < 10 then
    if (test n) then 1 else 0
    else (num_cifre_che_soddisfano (n/10) test) +
      (if test (n mod 10) then 1 else 0);;
num_cifre_che_soddisfano : int -> (int -> bool) -> int = <fun>
```

Possiamo definire le due funzioni citate sopra come casi speciali della funzione più generale `num_cifre_che_soddisfano` (N.B. : in CaML il modulo si ottiene con `mod`, in F# con `%`):

```
# let num_odd_digits n =
  num_cifre_che_soddisfano n (fun n -> n mod 2 = 1);;
num_odd_digits : int -> int = <fun>

# num_odd_digits 6749;;
- : int = 2

# let num_6s n = num_cifre_che_soddisfano n (fun n -> n = 6);;
num_6s : int -> int = <fun>

# num_6s 67586;;
- : int = 2
```

ESERCIZIO 8.2

Usare la funzione `num_cifre_che_soddisfano` per definire la funzione `num_cifre`, che conta il numero di cifre che compongono la rappresentazione decimale di un numero `n`.

ESERCIZIO 8.3

Riscrivere la funzione `num_cifre_che_soddisfano` in modo iterativo.

ESERCIZIO 8.4

Scrivere una funzione generale che, dati in input due interi, `low` e `high`, e una funzione `f`, calcoli $f(\text{low}) + f(\text{low}+1) + \dots + f(\text{high})$. Mostrare come tale funzione può essere usata per sommare i quadrati degli interi compresi tra 5 e 10 e per sommare le radici quadrate degli interi compresi tra 10 e 20.

8.2 – Funzioni che restituiscono funzioni

Finora abbiamo visto funzioni che prendono altre funzioni come parametri nello stesso modo in cui prendono numeri o caratteri in input. Tuttavia, le funzioni non prendono solamente dei valori in input, ma restituiscono anche dei valori come risultato della computazione. Le funzioni di ordine superiore possono essere usate per calcolare altre funzioni. In altre parole, possiamo costruire funzioni che restituiscono funzioni. Chiaramente questo meccanismo ci può far risparmiare molto lavoro.

Come possiamo far sì che una funzione restituisca un'altra funzione? Esattamente come una funzione restituisce un numero. Ricordiamo che, per assicurarci che una funzione, quando applicata, restituisca un numero, il suo corpo deve essere una espressione che valuti ad un numero. Allo stesso modo, affinché una funzione generi una nuova funzione quando viene applicata, il suo corpo deve essere una espressione che valuta ad una funzione. Una espressione che valuta a una nuova funzione è l'espressione `fun`. Ecco un semplice esempio:

```
# let make_multiplier scaling_factor =
    fun x -> x * scaling_factor;;
make_multiplier : int -> int = <fun>

# let double = make_multiplier 2;
double : int -> int = <fun>;

# let triple = make_multiplier 3;;
triple : int -> int = <fun>

# double 7;;
- : int = 14

# triple 12;;
- : int = 36
```

Quando valutiamo l'espressione `let double = make_multiplier 2` viene valutato il corpo della funzione `make_multiplier` con il valore `2` sostituito a `scaling_factor`. In altre parole, viene valutata l'espressione `lambda fun x -> x * scaling_factor` con il `2` al posto di `scaling_factor`. Il risultato di questa valutazione è la funzione chiamata `double`, esattamente come se la definizione di `double` fosse stata `let double = fun x -> x * 2`. Quando applichiamo `double` al valore `7`, la procedura `fun x -> x * 2` viene applicata a `7`, e il risultato è ovviamente `14`.

ESERCIZIO 8.5

Scrivere una funzione `make_exponentiate` che prende in input un singolo parametro `e` (esponente) e ritorna una funzione che prende anch'essa un solo parametro, che verrà poi elevato alla potenza `e`. Come esempi, definire le funzioni `square` e `cube` come segue:

```
# let square = make_exponentiate 2;;
# let cube = make_exponentiate 3;;
# square 4;;
- : int = 16

# cube 4;;
- : int = 64
```

ESERCIZIO 8.6

Si considerino le funzioni `make_multiplier` e `make_exponentiate` definite sopra. Si noti che queste due funzioni sono molto simili. Potremmo astrarne le somiglianze creando una funzione ancora più generale `make_generator` tale da poter semplicemente scrivere:

```
# let make_multiplier = make_generator ... ;;
# let make_exponentiate = make_generator expt;;
```

(dove `expt` è una procedura che calcola l'esponenziale). Scrivere la definizione della funzione `make_generator`.

Ipotizziamo di voler automatizzare la creazione di funzioni come `repeatedly_square`. Questa funzione prende due argomenti, il numero da elevare al quadrato e quante volte elevarlo al quadrato. Vogliamo creare una funzione di ordine superiore chiamata `make_repeated_version_of` che sia in grado di creare `repeatedly_square` a partire da `square`.

```
# let make_repeated_version_of f =
  let rec the_repeated_version b n =
    if n = 0 then b
    else the_repeated_version (f b) (n - 1)
  in
  the_repeated_version;;
make_repeated_version_of : ('a -> 'a) -> 'a -> int -> 'a = <fun>

# let square x = x * x;;
square : int -> int = <fun>

# let repeatedly_square = make_repeated_version_of square;;
repeatedly_square : int -> int -> int = <fun>

# repeatedly_square 2 3;;
- : int = 256
```

Una considerazione da fare riguardo a questo esempio è che abbiamo usato una definizione interna di `the_repeated_version` per fornire un nome alla procedura generata (mentre finora restituivamo una anonima `fun`). In questo modo possiamo riferirci a tale procedura per nome quando essa viene reinvocata da sé stessa per compiere le rimanenti $n - 1$ ripetizioni. Avendo definito internamente un nome, ritorniamo la procedura con quel nome.

ESERCIZIO 8.7

Definire una funzione che possa essere usata per produrre sia la funzione `factorial` (fattoriale) che la funzione `sum_of_first` (somma dei primi n numeri interi, dove n è l'unico parametro).

ESERCIZIO 8.8

Generalizzare la soluzione dell'esercizio precedente in modo tale che possa essere usata anche per produrre la funzione `sum_of_squares` (somma dei primi n numeri interi elevati al quadrato) e `sum_of_cubes` (somma dei primi n numeri interi elevati al cubo).

8.3 – Una applicazione: verificare gli ID

Si consideri il seguente scenario:

- *Potrei avere il suo numero di carta di credito?*
- *Sì, è 6011302631452178.*
- *Mi scusi, devo averlo digitato male. Potrebbe ripetere?*

Come poteva sapere il commesso che il numero era sbagliato?

I numeri delle carte di credito sono uno degli esempi più comuni di *self-verifying numbers*. Altri esempi includono il numero ISBN dei libri, l'UPC (Universal Product Code) che è un codice sui prodotti, i numeri bancari sugli assegni, i numeri di serie sugli ordini postali, il codice di appartenenza a varie organizzazioni, l'ID degli studenti in alcune università.

I *self-verifying numbers* sono costruiti in modo tale che qualunque numero valido abbia una specifica proprietà numerica e che gli errori più comuni, come scambiare due cifre o cambiare il valore di una cifra, portino ad un numero che non soddisfa quella proprietà. In questo modo possiamo distinguere tra numeri legittimi e numeri errati, senza nemmeno dover scorrere la lista dei numeri validi.

Ciò che ci interessa a proposito dei *self-verifying numbers* è che ci sono molti sistemi diversi in uso, ma tutti hanno una forma generale comune. Dunque, sebbene abbiano bisogno di una funzione diversa per la verifica di ciascun *tipo* di numero, possiamo creare una funzione di ordine superiore che ci consenta di definire tutti i verificatori.

Ipotizziamo di chiamare d_1 la cifra più a destra di un numero, d_2 la seconda cifra più a destra, etc. Tutte le tipologie di numeri identificativi elencati prima possiedono una proprietà del seguente tipo:

$$f(1, d_1) + f(2, d_2) + f(3, d_3) + \dots \text{ è divisibile per } m$$

La differenza tra il numero di una carta di credito e il codice di un oggetto della spesa, o tra un libro e un bonifico postale, sta nella funzione f e nel divisore m .

Come possiamo definire la funzione di alto livello che costruisca per noi tutti i verificatori? Come al solito, consideriamo prima una delle funzioni che vorremmo costruire. Questo verificatore controlla se la somma delle cifre di un numero è divisibile per 17: in altre parole, il divisore è 17 e la funzione è semplicemente $f(i, d_i) = d_i$. Per scrivere questo verificatore, scriviamo prima una procedura che sommi le cifre del numero dato. Ricordiamo che possiamo individuare le cifre di un numero tramite divisioni successive per 10. Il resto della divisione per 10 è la cifra più a destra (d_1), e il quoziente è composto dalle cifre rimanenti. Per esempio, la seguente funzione calcola la somma delle cifre di un numero usando un processo iterativo:

```
# let sum_of_digits n =
  let rec sum_plus n addend =
    if n = 0 then addend
    else sum_plus (n / 10) (addend + (n mod 10))
  in sum_plus n 0;;
sum_of_digits : int -> int = <fun>
```

Il seguente verificatore controlla se la somma delle cifre di un numero n è divisibile per 17.

```
# let verifier_mod_17 n =
  if (sum_of_digits n) mod 17 = 0
  then true
  else false;;
verifier_mod_17 : int -> bool = <fun>
```

```
# verifier_mod_17 2431232;;
- : bool = true

# verifier_mod_17 2431231;;
- : bool = false
```

Scriviamo ora la funzione `make_verifier`, che prende `f` e `m` come suoi due argomenti e restituisce una funzione che controlla la validità di un numero. L'argomento `f` è anch'esso una funzione, ovviamente.

```
# let general_sum_of_digits f n =
  let rec sum_plus i n addend =
    if n = 0 then addend
    else sum_plus (i+1) (n / 10) (addend + f i (n mod 10))
  in sum_plus 1 n 0;;
general_sum_of_digits : (int -> int -> int) -> int -> int = <fun>

# let make_verifier f m =
  fun x -> if (general_sum_of_digits f x) mod m = 0
    then true
    else false;;
make_verifier : (int -> int -> int) -> int -> int -> bool = <fun>
```

Vediamo ora come si può definire la funzione `verifier_mod_17` in termini di `make_verifier`.

```
# let verifier_mod_17 =
  make_verifier (fun i x -> x) 17;;
verifier_mod_17 : int -> bool = <fun>

# verifier_mod_17 2431231;;
- : bool = false
```

Il prossimo è un esempio di verificatore per i numeri ISBN dove il divisore è 11 e la funzione è semplicemente $f(i, d_i) = i * d_i$.

```
# let check_isbn = make_verifier (fun i x -> i * x) 11;;
check_isbn : int -> bool = <fun>
```

Analogamente, possiamo scrivere:

```
# let check_isbn = make_verifier (prefix *) 11;;
check_isbn : int -> bool = <fun>

# check_isbn 0262010771;;
- : bool = true
```

Poiché viene restituito `true`, il numero passato è un numero ISBN valido. Altri tipi di numeri usano funzioni leggermente più complicate, ma potrai comunque usare `make_verifier` per costruire un verificatore molto più facilmente che dovendo ripartire da zero.

ESERCIZIO 8.9

Per i codici UPC (il codice a barre sui prodotti del supermercato) il divisore è 10 e la funzione f è definita come segue:

$$f(i, d_i) = \begin{cases} d_i & \text{se } i \text{ è dispari} \\ 3 * d_i & \text{se } i \text{ è pari} \end{cases}$$

- a) Costruire un verificatore per i codici UPC che usi la procedura `make_verifier`, e testarla su qualcuno dei tuoi prodotti (il codice UPC è formato da tutti i numeri, sia quello a sinistra delle barre, che quelli sotto le barre, che quello a destra).
- b) Prova a introdurre qualche errore, come scambiare due cifre o cambiarle. Il verificatore se ne accorge?

SOLUZIONE

```
# let check_upc =
  make_verifier (fun i x -> if i mod 2 = 1 then x else 3 * x) 10;;
check_upc : int -> bool = <fun>
```

ESERCIZIO 8.10

Il verificatore per i numeri di carte di credito usa 10 come divisore e una funzione f che restituisce d_i quando i è dispari. Tuttavia, quando i è pari, la funzione è un po' più complicata: è $2 * d_i$ se $d_i < 5$ e $2 * d_i + 1$ se $d_i \geq 5$.

- a) Costruire un verificatore per i numeri di carte di credito.
- b) Nel dialogo riportato all'inizio della sezione, il commesso aveva davvero sbagliato a scrivere il numero, oppure era il cliente ad averlo letto errato?

```
# let check_credit_card =
  make_verifier (fun i x -> if i mod 2 = 1 then x
                      else if x < 5 then 2 * x
                      else (2 * x) + 1) 10;;
check_credit_card : int -> bool = <fun>
```

ESERCIZIO 8.11

Il numero seriale delle spedizioni monetarie postali americane è un *self-verifying number* con un divisore pari a 9 e una funzione molto semplice: $f(i, d_i) = d_i$ con una sola eccezione, ovvero $f(1, d_1) = -d_1$.

- a) Costruire un verificatore per questi numeri, e scoprire quale dei due seguenti ordini di denaro è errato: 48077469777 oppure 48077462766.
- b) In realtà, entrambi i numeri sono errati. In un caso, l'errore deriva dal fatto che uno 0 è stato rimpiazzato da un 7 e nell'altro invece due cifre sono state scambiate. Riesci a scoprire quale errore è stato scoperto e quale no? Questo ti aiuta a capire come mai gli altri tipi di numeri usano funzioni più complicate.

SOLUZIONE

```
# let check_post =
  make_verifier (fun i x -> if i = 1 then -x else x) 9;;
check_post : int -> bool = <fun>
```

8.4 – Funzioni di ordine superiore e liste

Introduciamo due utili funzioni di ordine superiore sulle liste: le funzioni `map` e `filter`.

La funzione `map` prende come argomento una funzione e una lista, e restituisce la lista che si ottiene applicando la funzione ad ogni elemento della lista.

```
# let rec map f lst =
  match lst with
  [] -> []
  | x::xs -> (f x) :: map f xs;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>

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

# map double;;
- : int list -> int list = <fun>

# map double [4;5;6];;
- : int list = [8; 10; 12]

# map double [[4;5];[6]];;
This expression has type int list list, but is used with type int list.

# map (map double);;
- : int list list -> int list list = <fun>

# map (map double) [[4;5];[6]];;
- : int list list = [[8; 10]; [12]]
```

La funzione `filter` filtra da una lista tutti gli elementi che soddisfano un dato predicato, essa rimuove cioè dalla lista tutti gli elementi che non soddisfano il predicato dato:

```
# let rec filter p lst =
  match lst with
  [] -> []
  | hd::tl -> if p hd then hd::(filter p tl)
               else (filter p tl);;
filter : ('a -> bool) -> 'a list -> 'a list = <fun>

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

# filter odd [1;6;8;11;13;14;15];;
- : int list = [1; 11; 13; 15]

# let even n = n mod 2 = 0
  in filter even [1;6;8;11;13;14;15];;
- : int list = [6; 8; 14]
```

Uno dei vantaggi nell'uso delle funzioni di ordine superiore sta nel fatto che è semplice esprimere delle proprietà algebriche, ed usare queste proprietà nella manipolazione di espressioni. Nel nostro caso, ci sono

una serie di proprietà utili per le funzioni `map` e `filter`. Ad esempio, `map` distribuisce sulla composizione di funzioni e sulla concatenazione di liste:

$$\text{map} (\text{compose } f g) = \text{compose} (\text{map } f)(\text{map } g)$$

$$\text{map } f (xs @ ys) = (\text{map } f xs) @ (\text{map } f ys)$$

$$\text{compose} (\text{map } f) \text{concat} = \text{compose concat map} (\text{map } f)$$

La prima legge dice che se applichiamo prima g ad una lista, e poi applichiamo f al risultato, otteniamo lo stesso risultato applicando $\text{compose } f g$ alla lista. La seconda legge esprime la distributività sulla concatenazione, mentre la terza legge è una generalizzazione della seconda: è la stessa cosa applicare f al risultato della concatenazione di una lista di liste e applicare $\text{map } f$ ad ogni lista componente e quindi concatenare il risultato.

Risultati simili possono essere espressi anche per `filter`:

$$\text{filter } p (xs @ ys) = \text{filter } p xs @ \text{filter } p ys$$

$$\text{compose} (\text{filter } p) \text{concat} = \text{compose concat map} (\text{filter } p)$$

$$\text{compose} (\text{filter } p)(\text{filter } q) = \text{compose} (\text{filter } q)(\text{filter } p)$$

La seconda legge generalizza la proprietà distributiva, espressa dalla prima legge, a liste di liste. La terza legge generalizza l'ordine di applicazione dei “filtri”.

ESERCIZIO 8.12

Una definizione alternativa di `filter` può essere data utilizzando `concat` e `map` nel seguente modo:

```
# let filter p = let box x = ... in compose concat (map box);;
```

Si completi la definizione.

8.5 - Gli operatori di fold

Molti degli operatori che abbiamo visto sinora restituiscono come risultato delle liste. Gli operatori di `fold` sono i più generali, in quanto convertono liste in altri valori. Considereremo gli operatori `fold_right` e `fold_left`. La definizione informale dell'operatore `fold_right` è:

$$\text{fold_right } f a [x_1; x_2; \dots; x_n] = f x_1 (f x_2 (\dots (f x_n a) \dots))$$

dove f è una variabile legata ad una funzione binaria. In particolare, abbiamo:

$$\text{fold_right } f a [] = a$$

$$\text{fold_right } f a [x_1] = f x_1 a$$

$$\text{fold_right } f a [x_1; x_2] = f x_1 (f x_2 a)$$

⋮

Dalla definizione informale siamo già in grado di capire che il tipo del secondo argomento deve essere uguale al codominio di f . Infatti:

```
# let rec fold_right f a lst =
  match lst with
  [] -> a
  | x :: xs -> f x (fold_right f a xs);;
- : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

Utilizzando `fold_right` possiamo definire una serie di funzioni di utilità.

```
# let sumlist = fold_right (prefix +) 0;;
sumlist : int list -> int = <fun>

# let product = fold_right (prefix *) 1;;
product : int list -> int = <fun>

# let concat = fold_right (prefix @) [];;
concat : '_a list list -> '_a list = <fun>

# let And = fold_right (prefix &) true;;
And : bool list -> bool = <fun>

# let Or = fold_right (prefix or) false;;
Or : bool list -> bool = <fun>
```

Ognuna di queste funzioni definisce un operatore su liste. Ad esempio, `sumlist` implementa la funzione somma sugli elementi di una lista: se $[x_1; x_2; \dots; x_n]$ è una lista di interi, allora

$$\text{sumlist } [x_1; x_2; \dots; x_n] = \sum_{i=1}^n x_i$$

Si noti che anche la funzione `concat` è definibile tramite l'operatore di `fold`. Tutte le funzioni definite sopra condividono un'importante proprietà: la funzione f è associativa e ha un elemento neutro (esiste cioè un elemento a tale che $f x a = f a x = x$).

È possibile definire altre funzioni con l'ausilio di funzioni non associative. Per esempio, è molto semplice definire le funzioni `length` e `rev`:

```
# let length = let oneplus x n = n+1 in fold_right oneplus 0;;
length : 'a list -> int = <fun>
```

In questo esempio viene utilizzata la funzione locale `oneplus`, che ignora il primo argomento, limitandosi ad incrementare il secondo argomento.

```
# let rev = let postfix x xs = xs @ [x] in fold_right postfix [];;
rev : 'a list -> 'a list = <fun>
```

L'idea è quella di appendere in successione $x_1; x_2; \dots; x_n$ alla fine di una lista inizialmente vuota.

L'operatore `fold_left` ha un comportamento *duale* rispetto all'operatore `fold_right`. Informalmente,

$$\text{fold_left } f a [x_1; x_2; \dots; x_n] = f (\dots f (f a x_1) x_2 \dots) x_n$$

In particolare:

$$\begin{aligned} \text{fold_left } f a [] &= a \\ \text{fold_left } f a [x_1] &= f a x_1 \\ \text{fold_left } f a [x_1; x_2] &= f (f a x_1) x_2 \\ &\vdots \end{aligned}$$

Si può notare che il tipo di `fold_left` è abbastanza simile a quello di `fold_right` (eccetto che per il tipo del primo argomento).

```
# let rec fold_left f a lst =
    match lst with
    [] -> a
    | x :: xs -> fold_left f (f a x) xs;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

Se f è associativa, allora ' a ' e ' b ' denotano lo stesso tipo, e quindi i due operatori `fold_right` e `fold_left` hanno esattamente lo stesso tipo.

Un esempio di funzione definibile per mezzo di `fold_left` è il seguente:

```
# let decimal_number xs = let decimal n x = 10*n + x
                           in fold_left decimal 0 xs;;
decimal_number : int list -> int = <fun>
```

Tale funzione codifica una sequenza di cifre in un numero singolo secondo il sistema decimale:

$$\text{decimal_number } [x_1; \dots; x_n] = \sum_{k=1}^n x_k * 10^{n-k}$$

Vale la pena sottolineare alcune importanti proprietà degli operatori di `fold`. Se la funzione f è associativa e a è un elemento netro di f , allora

$$\text{fold_right } f a xs = \text{fold_left } f a xs$$

Cioè, i due operatori definiscono la stessa funzione. Ad esempio, la funzione `sumlist` definita sopra (così come tutte le altre funzioni elencate sopra) può essere equivalentemente definita utilizzando `fold_left`.

Si supponga di voler calcolare il massimo valore di una lista. Utilizzando un operatore di `fold`, la definizione dovrebbe venire abbastanza semplice:

$$\text{maxlist} = \text{fold_left } (\text{max}) a$$

dove max è un operatore che calcola il massimo tra due elementi. Il problema che si pone è come scegliere a . Vorremmo che a fosse l'elemento neutro per max , poiché il suo valore non dovrebbe influenzare il risultato (dipendente esclusivamente dalla lista). Se la lista fosse composta soltanto di elementi non

negativi, allora potremmo scegliere $a = 0$. Sfortunatamente, nel caso generale non è possibile individuare un valore valido di a . Il problema può essere risolto definendo una piccola variante degli operatori di *fold*:

$$\text{foldr } f [x_1; x_2; \dots; x_n] = f x_1(f x_2 (\dots (f x_{n-1} x_n) \dots))$$

$$\text{foldl } f [x_1; x_2; \dots; x_n] = f (\dots f (f x_1 x_2) x_3) \dots) x_n$$

In particolare,

$$\text{foldl } f [x_1] = x_1$$

$$\text{foldl } f [x_1; x_2] = f x_1 x_2$$

⋮

Si noti come il tipo degli operatori sia definito solo a partire da un unico tipo:

```
# let rec foldl f lst =
    match lst with
    [x] -> x
    | x :: y :: xs -> foldl f ((f x y) :: xs);;
- : ('a -> 'a -> 'a) -> 'a list -> 'a = <fun>
```

A questo punto è semplice definire la funzione desiderata:

```
# let maxlist = foldl max;;
maxlist : 'a list -> 'a = <fun>

# maxlist [1;2;3;4];;
- : int = 4
```

Si noti infine, come sia semplice definire `foldl` in termini di `fold_left`:

```
# let foldl f xs = fold_left f (hd xs) (tl xs);;
foldl : ('a -> 'a -> 'a) -> 'a list -> 'a = <fun>
```

ESERCIZIO 8.13

Si definiscano le funzioni `for_all` e `exists` in termini di un operatore di *fold*.

ESERCIZIO 8.14

La funzione `remdup` rimuove i duplicati adiacenti in una lista. Ad esempio,

```
# remdup [1;2;2;3;3;3;1;1];;
- : int list = [1;2;3;1]
```

Si definisca la funzione `remdup` utilizzando un operatore di *fold*.

ESERCIZIO 8.15

Si definisca una funzione f tale che

$$\text{length } xs = \text{fold_right } f 0 xs.$$

8.6 – Esercizi di riepilogo

ESERCIZIO 8.16

Scrivere una funzione di ordine superiore chiamata `make_function_with_exception` che prende due numeri e una funzione come parametri e ritorna una funzione con lo stesso comportamento della funzione passata come parametro tranne quando le viene passato un argomento speciale. I due argomenti numerici di `make_function_with_exception` specificano quale è l'argomento che fa eccezione e cosa ritornare in quel caso. Per esempio, la procedura `usually_sqrt` che segue si comporta come `sqrt`, tranne quando le viene passato l'argomento 7 (in quel caso ritorna il risultato 2):

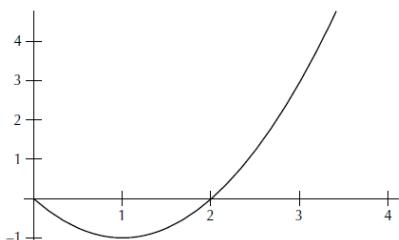
```
# let sqr x = x * x;;
# let usually_sqr = make_function_with_exception 7 2 sqr;;
# usually_sqr 5;;
- : int = 25
# usually_sqr 6;;
- : int = 36
# usually_sqr 7;;
- : int = 2
```

ESERCIZIO 8.17

Se l e h sono numeri interi, con $l < h$, diciamo che f è una funzione crescente sul range da l a h se $f(l) < f(l + 1) < \dots < f(h)$. Scrivere una funzione `increasing_on_integer_range` che prende f, l, h come suoi 3 argomenti e restituisce `true` o `false`, a seconda che f sia o meno crescente in quel range.

ESERCIZIO 8.18

Ipotizziamo di avere una funzione e di voler scoprire in quale punto intero, all'interno di un certo range, la funzione assume il valore minimo. Per esempio, guardando la figura qui sotto della funzione $f(x) = x^2 - 2x$, si può vedere che nel range tra 0 e 4 la funzione assume il valore minimo quando x vale 1.



Vogliamo scrivere una funzione che risolva questo problema. Essa, potrebbe essere usata come segue:

```
# integer_in_range_where_smallest (fun x -> x * x - 2 * x) 0 4;;
- : int = 1
```

La funzione è definita come segue; riempire gli spazi vuoti per completarla:

```
let rec integer_in_range_where_smallest f a b =
  if a = b then
    a
  else
    let smallest_place_after_a ...
      in if ... then ... else ...;;
```

ESERCIZIO 8.19

Si considerino le seguenti definizioni:

```
# let make_scaled scale f = fun x -> scale * (f x);;
make_scaled : int -> ('a -> int) -> 'a -> int = <fun>

# let add_one x = 1 + x;;
add_one : int -> int = <fun>

# let mystery = make_scaled 3 add_one;;
mystery : int -> int = <fun>
```

Rispondere alle seguenti domande, indicando anche come si è arrivati alla risposta:

- a) Qual è il valore di `mystery 4`?
- b) Qual è il valore dell'espressione

```
((make_scaled 2 (make_scaled 3 add_one)) 4)?
```

ESERCIZIO 8.20

Date le seguenti definizioni:

```
# let f m b = fun x -> (m * x) + b;;
f : int -> int -> int -> int = <fun>

# let g = f 3 2;;
g : int -> int = <fun>
```

Per ognuna delle seguenti espressioni, dire se:

- a) viene segnalato un errore (in questo caso spiegare brevemente la natura dell'errore);
- b) il valore è una funzione (in questo caso specificare quanti argomenti la funzione si aspetta);
- c) il valore è un numero (in questo caso specificare quale numero).

Le espressioni sono:

- 1) `f;;`
- 2) `g;;`
- 3) `(f 3 2) * 7;;`
- 4) `g 6;;`
- 5) `f 6;;`
- 6) `(f 4 7) 5;;`
- 7) `f (g 6);;`
- 8) `g = f 6 9;;`

ESERCIZIO 8.21

Si consideri il seguente esempio:

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

# let square x = x * x;;
square : int -> int = <fun>

# let new_function = make_averaged_function double square;;
# new_function 4;;
- : int = 12

# new_function 6;;
- : int = 24
```

Nel primo esempio, la funzione `new_function` definita in termini della funzione `make_averaged_function` ritorna 12 perché 12 è la media di 8 (il doppio di 4) e 16 (il quadrato di 4). Nel secondo esempio, la funzione `new_function` ritorna 24 perché 24 è la media tra 12 (il doppio di 6) e 36 (6 al quadrato).

In generale, la funzione `new_function` ritorna la media dei valori restituiti da `double` e `square`: queste due funzioni vengono passate come parametri a `make_averaged_function` nel corpo della definizione della funzione `new_function`.

Scrivere la definizione della funzione di ordine superiore `make_averaged_function`.

ESERCIZIO 8.22

- Scrivere una funzione che conti il numero di occorrenze di un certo elemento in una lista data.
- Generalizzare la funzione sopra in modo tale che conti il numero di elementi in una lista data che soddisfano un certo predicato.

ESERCIZIO 8.23

- Scrivere una funzione `lst_min` che date due liste di interi della stessa dimensione ritorni `true` se ogni elemento della prima è minore del corrispondente elemento della seconda. Ad esempio `lst_min [1;2;3] [2;3;4]` deve valutare a `true`.
Cosa dovrebbe accadere se le due liste non hanno la stessa dimensione?
- Generalizzare questa funzione mediante la definizione di `lists_compare` che prende 3 argomenti: il primo è un predicato che prende due argomenti (ad esempio `<`) e gli altri due sono liste. La funzione `lists_compare` ritorna `true` se e solo se il predicato ritorna sempre `true` su elementi corrispondenti delle due liste.

Potremmo ridefinire `lst_min` nella seguente maniera:

```
# let list_min l1 l2 = lists_compare (prefix <) l1 l2;;
```

ESERCIZIO 8.24

- Scrivere una funzione che, quando riceve in input un intero n , ritorna la lista dei primi n quadrati perfetti.
- Scrivere una funzione che, quando riceve in input un intero n , ritorna la lista dei primi n interi pari.
- Scrivere una funzione chiamata `sevens` che, quando riceve in input un intero n , ritorna una lista di n sette. Per esempio:

```
# sevens 5;;
- : int list = [7; 7; 7; 7; 7]
```

- Scrivere una funzione che, quando riceve in input una lista di interi positivi, ritorna una lista di liste di interi. Ognuna di queste liste contiene gli interi positivi da 1 fino al numero corrispondente nella lista originale. Ad esempio:

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

ESERCIZIO 8.25

Siano f_1, f_2, \dots, f_n tutte funzioni da interi a interi. La somma funzionale di f_1, f_2, \dots, f_n è definita come la funzione che, dato un numero x , ritorna il valore $f_1(x) + f_2(x) + \dots + f_n(x)$. Scrivere la funzione `function_sum` che, data una lista di funzioni, ritorna la somma funzionale di quelle funzioni. Ad esempio:

```
# let square x = x * x;;
# let cube x = x * x * x;;
# function_sum [square;cube] 2;;
- : int = 12
```

ESERCIZIO 8.26

Scrivere una funzione chiamata `apply_all` che, quando riceve in input una lista di funzioni e un numero intero, restituisce la lista dei valori ottenuti applicando ciascuna funzione a quel numero. Ad esempio:

```
# apply_all [sqrt;square;cube] 4;;
- : int list = [2; 16; 64]
```

ESERCIZIO 8.27

Scrivere una funzione di ordine superiore `make_list_scaler` che prende in input un singolo numero `scale` e ritorna una funzione che, quando applicata a una lista `lst` di numeri interi, ritorna la lista ottenuta dalla moltiplicazione di ciascun elemento di `lst` per `scale`. Quindi si dovrebbe ottenere:

```
# let scale_by_5 = make_list_scaler 5;;
# scale_by_5 [1;2;3;4];
- : int list = [5; 10; 15; 20]
```

ESERCIZIO 8.28

Scrivere la funzione `map2` che prende in input una funzione binaria e due liste e ritorna la lista ottenuta applicando la funzione in input agli elementi corrispondenti delle due liste. Ad esempio:

```
# map2 (prefix +) [1;2;3] [2;0;-5];;
- : int list = [3; 2; -2]

# map2 (prefix *) [1;2;3] [2;0;-5];;
- : int list = [2; 0; -15]
```

Scrivere la definizione della funzione `map2`. Si assuma che le due liste abbiano la stessa lunghezza.

ESERCIZIO 8.29

Dato un predicato che testa un singolo elemento, ad esempio `positive`, possiamo costruirne una versione che testa tutti gli elementi di una lista. Un esempio è un predicato che testa se tutti gli elementi di una lista sono positivi. Definire la funzione `all_are` che fa questo. Dovrebbe essere possibile usarla nei seguenti modi:

```
# all_are positive [1;2;3;4];;
- : bool = true

# all_are even [2;4;5;6;8];;
- : bool = false
```

(dove i predicati `positive` e `even` siano stati precedentemente definiti).

Capitolo 9 – Prove di correttezza e altri esercizi sulle liste

I principi di ricorsione ed induzione possono essere applicati anche alle liste. Abbiamo visto che, nel caso dei numeri naturali, ricorsione ed induzione sono basati sulla seguente analisi: un numero naturale è 0 oppure è il successore di un altro numero naturale (cioè ha la forma $n + 1$, per qualche n). Allo stesso modo, l'operatore `::` (costruttore di liste) ci permette di definire una lista o come la lista vuota `[]` oppure come la lista ottenuta aggiungendo un elemento in testa ad un'altra lista ($x :: xs$, per qualche x ed xs).

Un semplice esempio di definizione ricorsiva su liste è data dalla funzione `length`:

```
# let rec length lst =
  match lst with
  [] -> 0
  | x::xs -> 1 + (length xs);;
length : 'a list -> int = <fun>
```

L'utilizzo di `[]` e `(x :: xs)` ricalca esattamente l'uso di 0 e $(n + 1)$ che avevamo fatto nel caso dei numeri naturali. Possiamo utilizzare la funzione `length` per calcolare la lunghezza della lista `[2; 5]`:

$$\begin{aligned} & \text{length}[2; 5] \\ &= \{\text{secondo pattern di length}\} \\ &= 1 + (\text{length}[5]) \\ &= \{\text{secondo pattern di length}\} \\ &= 1 + (1 + \text{length}[\]) \\ &= \{\text{primo pattern di length}\} \\ &= 1 + (1 + 0) \\ &= \{\text{per le regole dell'algebra}\} \\ &= 2. \end{aligned}$$

Vediamo come possiamo definire l'operatore di concatenazione `@`:

```
# let rec concat xs ys =
  match xs with
  [] -> ys
  | z::zs -> z :: (concat zs ys);;
concat : 'a list -> 'a list -> 'a list = <fun>
```

D'ora in poi faremo riferimento alla funzione `concat` per descrivere il comportamento dell'operatore di concatenazione `@`.

Per dimostrare che una proprietà $P(xs)$ è vera per ogni lista finita xs , dobbiamo mostrare che:

Caso 1. $P([])$ è vera.

Caso 2. Se $P(xs)$ è vera, allora $P(x :: xs)$ è vera per ogni x .

È facile convincersi che il principio vale. Sappiamo per il primo caso che $P([])$ vale, e per il secondo sappiamo che $P([x])$ è vera per ogni x (infatti $[x] = x :: []$); per il secondo caso, $P([y; x])$ è vera per ogni y , e così via. A questo punto si vede che $P(xs)$ è vera per ogni lista xs .

Tramite l'induzione è possibile dimostrare molte delle proprietà degli operatori che abbiamo definito. Ad esempio, una proprietà dell'operatore di concatenazione `@` è l'associatività: per ogni tripla xs, ys, zs di liste finite, vale

$$xs @ (ys @ zs) = (xs @ ys) @ zs.$$

Dimostrazione. La dimostrazione segue per induzione su xs .

Caso []. Abbiamo:

$$[] @ (ys @ zs)$$

$$= \quad \{\text{primo pattern di concat}\}$$

$$ys @ zs$$

$$= \quad \{\text{primo pattern di concat}\}$$

$$([] @ ys) @ zs.$$

Caso ($x :: xs$). In questo caso:

$$(x :: xs) @ (ys @ zs)$$

$$= \quad \{\text{secondo pattern di concat}\}$$

$$x :: (xs @ (ys @ zs))$$

$$= \quad \{\text{ipotesi induttiva}\}$$

$$x :: ((xs @ ys) @ zs)$$

$$= \quad \{\text{secondo pattern di concat}\}$$

$$(x :: (xs @ ys)) @ zs$$

$$= \quad \{\text{secondo pattern di concat}\}$$

$$((x :: xs) @ ys) @ zs.$$

ESERCIZIO 9.1

Dimostrare per induzione che

$$\text{length} (xs @ ys) = (\text{length } xs) + (\text{length } ys).$$

9.1 – Operazioni su liste

Nel capitolo precedente abbiamo visto molte importanti operazioni sulle liste. In questo paragrafo dimostriamo alcune proprietà di tali operazioni.

9.1.1 – La funzione `combine`

La funzione `combine` prende come argomento una coppia di liste e restituisce una lista di coppie.

Possiamo generalizzare la definizione presentata nel capitolo precedente come segue:

```
# let rec combine (lst1, lst2)=
    match (lst1, lst2) with
    | ([]), ys           -> []
    | (x::xs, [])         -> []
    | (x::xs, y::ys)      -> (x, y)::combine(xs, ys) ;;
combine : 'a list * 'b list -> ('a * 'b) list = <fun>
```

Si noti che questa definizione è esaustiva. Infatti,

- il primo elemento della coppia è vuoto, oppure
- il primo è non vuoto e il secondo è vuoto, oppure
- tutti e due sono non vuoti.

Così, per ogni possibile coppia di liste, esiste un pattern (uno solo) che corrisponde a tale coppia.

Dimostriamo che vale la seguente proprietà

$$\text{length } \text{combine} (xs, ys) = \min\{\text{length } xs, \text{length } ys\}$$

per ogni coppia di liste xs, ys .

Dimostrazione. La dimostrazione segue per induzione su xs e ys .

Caso $([], ys)$. In questo caso:

$$\text{length } \text{combine} ([], ys)$$

$$= \quad \{\text{primo pattern di } \text{combine}\}$$

$$\text{length } []$$

$$= \quad \{\text{primo pattern di } \text{length}\}$$

$$0$$

$$= \quad \{\text{definizione di } \text{min}\}$$

$$\min \{0, (\text{length } ys)\}$$

= {primo pattern di `length`}
 $\min \{(length []), (length ys)\}.$

Caso ($x :: xs, []$). In maniera analoga al caso precedente:

`length combine ((x :: xs), [])`
= {secondo pattern di `combine`}
`length []`
= {primo pattern di `length`}
0
= {definizione di `min`}
 $\min \{(length x :: xs), 0\}$
= {primo pattern di `length`}
 $\min \{(length x :: xs), (length [])\}.$

Caso ($x :: xs, y :: ys$). Supponiamo $length combine (xs, ys) = \min\{(length xs), (length ys)\}$. Allora:

`length combine (x :: xs, y :: ys)`
= {terzo pattern di `combine`}
`length ((x, y) :: (combine(xs, ys)))`
= {secondo pattern di `length`}
 $1 + length (combine (xs, ys))$
= {ipotesi induttiva}
 $1 + \min\{(length xs), (length ys)\}$
= {la somma distribuisce su `min`}
 $\min\{1 + (length xs), 1 + (length ys)\}$
= {secondo pattern di `length`}
 $\min \{(length x :: xs), (length y :: ys)\}.$

La dimostrazione che abbiamo appena visto è valida poiché, come abbiamo discusso, i tre casi analizzati coprono ogni possibile combinazione di due liste. Formalmente, tale dimostrazione può essere giustificata da due induzioni annidate: la prima su xs e la seconda su ys .

9.1.2 – Le funzioni `take` e `drop`

Ricordiamo che le funzioni `take` e `drop` prendono come argomenti un numero naturale e una lista; `take n xs` restituisce la lista dei primi n elementi della lista xs , e `drop n xs` i restanti elementi. Le loro definizioni ricorsive sono le seguenti:

```
# let rec take n lst =
  match (n, lst) with
  (0, _) -> []
  | (_, []) -> []
  | (n, x::xs) -> x :: (take (n-1) xs);;
take : int -> 'a list -> 'a list = <fun>

# let rec drop n lst =
  match (n, lst) with
  (0, lst) -> lst
  | (_, []) -> []
  | (n, x::xs) -> (drop (n-1) xs);;
drop : int -> 'a list -> 'a list = <fun>
```

Ancora, queste definizioni coprono (in modo univoco) tutte le possibili combinazioni dei due argomenti. Un’importante legge relativa a `take` e `drop` è la seguente:

$$take n xs @ drop n xs = xs$$

per ogni numero naturale n e per ogni lista xs .

Dimostrazione. Dimostriamolo per induzione su n ed xs .

Caso (0, xs). Allora:

$$take 0 xs @ drop 0 xs$$

$$= \{ \text{primo pattern di } \text{take e drop} \}$$

$$[] @ xs$$

$$= \{ \text{primo pattern di concat} \}$$

$$xs.$$

Caso ($n + 1$, $[]$). In tal caso:

$$take (n + 1) [] @ drop (n + 1) []$$

$$= \{ \text{secondo pattern di } \text{take e drop} \}$$

$$[] @ []$$

```
= {primo pattern di concat}
[].
```

Caso $((n + 1), (x :: xs))$. Supponiamo che valga: $take n xs @ drop n xs = xs$. In quest'ultimo caso:

$$take (n + 1)(x :: xs)@ drop (n + 1) (x :: xs)$$

```
= {terzo pattern di take e drop}
```

$$(x :: take n xs)@ drop n xs$$

```
= {secondo pattern di concat}
```

$$x :: (take n xs @ drop n xs)$$

```
= {ipotesi induttiva}
```

$$x :: xs.$$

Si osservi che il caso analizzato è del tutto simile al caso precedente. L'unica differenza consiste nel fatto che la doppia induzione riguarda naturali e liste.

9.1.3 – Le funzioni map e filter

La funzione `map` applica una funzione ad ogni elemento di una lista data come argomento, mentre la funzione `filter` rimuove gli elementi che non soddisfano un dato predicato da una lista. La loro definizione ricorsiva è la seguente:

```
# let rec map f lst =
  match lst with
  [] -> []
  | x::xs -> (f x) :: map f xs;;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# let rec filter p lst =
  match lst with
  [] -> []
  | x::xs when p x -> x::(filter p xs)
  | x::xs when not (p x) -> (filter p xs);;
filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

La novità di queste definizioni consiste nel poter definire funzioni di ordine superiore tramite definizioni ricorsive, e di poter includere nelle definizioni ricorsive espressioni condizionali.

Una proprietà di `map` e `filter` è la seguente:

$$\text{filter } p (\text{map } f xs) = \text{map } f (\text{filter } (\text{compose } p f) xs)$$

Dimostrazione. La dimostrazione è per induzione su xs . I due casi sono: $[]$ e $x :: xs$; in quest'ultimo caso verranno considerati separatamente i casi in cui $p(f x)$ sia vero o falso.

Caso []. Allora:

$$\begin{aligned} & \text{filter } p (\text{map } f []) \\ = & \quad \{\text{primo pattern di map}\} \\ & \text{filter } p [] \\ = & \quad \{\text{primo pattern di filter}\} \\ & [] \\ = & \quad \{\text{primo pattern di map}\} \\ & \text{map } f [] \\ = & \quad \{\text{primo pattern di filter}\} \\ & \text{map } f (\text{filter} (\text{compose } p f) []). \end{aligned}$$

Caso ($x :: xs$) con $p(f x) = \text{true}$. Sia $\text{filter } p (\text{map } f xs) = \text{map } f (\text{filter} (\text{compose } p f) xs)$.
In tal caso:

$$\begin{aligned} & \text{filter } p (\text{map } f x :: xs) \\ = & \quad \{\text{secondo pattern di map}\} \\ & \text{filter } p (f x :: (\text{map } f xs)) \\ = & \quad \{\text{secondo pattern di filter, } p(f x) = \text{true}\} \\ & f x :: \text{filter } p (\text{map } f xs) \\ = & \quad \{\text{ipotesi induttiva}\} \\ & f x :: (\text{map } f (\text{filter} (\text{compose } p f) xs)) \\ = & \quad \{\text{secondo pattern di map}\} \\ & \text{map } f (x :: \text{filter} (\text{compose } p f) xs) \\ = & \quad \{\text{secondo pattern di filter, } p(f x) = \text{true}\} \\ & \text{map } f (\text{filter} (\text{compose } p f) x :: xs). \end{aligned}$$

Caso ($x :: xs$) con $p(f x) = \text{false}$. Sia $\text{filter } p (\text{map } f xs) = \text{map } f (\text{filter} (\text{compose } p f) xs)$.
In tal caso:

$$\begin{aligned} & \text{filter } p (\text{map } f x :: xs) \\ = & \quad \{\text{secondo pattern di map}\} \end{aligned}$$

$$\begin{aligned}
& \text{filter } p (f x :: (\text{map } f xs)) \\
= & \quad \{\text{secondo pattern di filter, } p(f x) = \text{false}\} \\
& \text{filter } p (\text{map } f xs) \\
= & \quad \{\text{ipotesi induttiva}\} \\
& (\text{map } f (\text{filter } (\text{compose } p f) xs)) \\
= & \quad \{\text{secondo pattern di filter, } p(f x) = \text{false}\} \\
& \text{map } f (\text{filter } (\text{compose } p f) x :: xs).
\end{aligned}$$

9.1.4 – Intervalli

In matematica, dati due numeri interi m e n , un intervallo $[m, n]$ rappresenta l'insieme di tutti i numeri interi x tali che $m \leq x \leq n$. Possiamo dare una definizione ricorsiva di un intervallo:

```

# let rec interval m n =
  if m > n then []
  else m :: interval (m+1) n;;
interval : int -> int -> int list = <fun>

# interval 6 9;;
- : int list = [6; 7; 8; 9]

```

In questa definizione i due casi sono distinti dal test $m > n$ piuttosto che dal pattern matching. Il valore di `interval m n` è ben definito per ogni coppia di numeri interi m e n , poiché la quantità $n - m$ decresce ad ogni chiamata ricorsiva.

Per poter ragionare sugli intervalli, abbiamo bisogno del seguente principio di induzione. Una proposizione $P(m, n)$ è vera per ogni coppia m e n , se:

Caso $m > n$. $P(m, n)$ è vera quando $m > n$, e

Caso $m \leq n$. Se $P(m + 1, n)$ è vera, allora $P(m, n)$ è vera quando $m \leq n$.

Questo principio può essere giustificato per induzione matematica sulla quantità $n - m$. Utilizzando tale principio di induzione, possiamo dimostrare la seguente legge:

$$\text{map } ((\text{prefix } +)k)(\text{interval } m n) = \text{interval } (k + m)(k + n)$$

Dimostrazione. La dimostrazione è per induzione sulla differenza tra n ed m .

Caso $m > n$. Allora:

$$\begin{aligned}
& \text{map } ((\text{prefix } +)k)(\text{interval } m n) \\
= & \quad \{\text{prima alternativa di } \text{interval}\}
\end{aligned}$$

$\text{map}((\text{prefix } +)k)[]$
 $= \{\text{primo pattern di map}\}$
 $[]$
 $= \{\text{prima alternativa di interval, poiché } m > n \Rightarrow k + m > k + n\}$
 $\text{interval}(k + m)(k + n).$

Caso $m \leq n$. In tal caso:

$\text{map}((\text{prefix } +)k)(\text{interval } m n)$
 $= \{\text{seconda alternativa di interval}\}$
 $\text{map}((\text{prefix } +)k)(m :: \text{interval}(m + 1)(n))$
 $= \{\text{secondo pattern di map}\}$
 $(k + m) :: (\text{map}((\text{prefix } +)k)(\text{interval}(m + 1)n))$
 $= \{\text{ipotesi induttiva}\}$
 $(k + m) :: \text{interval}(k + m + 1)(k + n)$
 $= \{\text{seconda alternativa di interval}\}$
 $\text{interval}(k + m)(k + n).$

ESERCIZIO 9.2

Definire ricorsivamente la funzione nth in modo tale che $\text{nth } m \ l$ selezioni l'elemento che si trova nella posizione m della lista l . Si verifichi inoltre la seguente proprietà:

$$\text{nth } m (\text{drop } n l) = \text{nth } (n + m)l$$

per ogni l , m ed n .

ESERCIZIO 9.3

Definire le funzioni `takewhile` e `dropwhile` che prendono come argomenti un predicato ed una lista l e restituiscono la sottolista $l1$ (risp. $l2$) di l tale che $l=l1@l2$ e ogni elemento di $l1$ soddisfa (risp. il primo elemento di $l2$ non soddisfa) il predicato. Ad esempio:

```
# takewhile even [2;4;5;6];
- : int list = [2;4]

# dropwhile even [2;4;5;6];
- : int list = [5;6]
```

ESERCIZIO 9.4

Definire la funzione `assoc` che dato un elemento `a` ed una lista di coppie, restituisce il secondo elemento della prima coppia nella lista il cui primo elemento è uguale ad `a`, se tale coppia esiste. Ad esempio,

```
# assoc 5 [(2,4);(5,6);(7,8)];;
- : int = 6
```

ESERCIZIO 9.5

Si dia una definizione ricorsiva delle funzioni `init` e `last` tali che `init(xs@[x])=xs` e `last(xs@[x])=x`. Si dimostri quindi che:

$$xs = init\ xs@[last\ xs]$$

ESERCIZIO 9.6

Si dimostrino le seguenti proprietà:

$$\begin{aligned} take\ m\ (drop\ n\ xs) &= drop\ n\ (take\ (m+n)\ xs) \\ drop\ m\ (drop\ n\ xs) &= drop\ (m+n)\ xs \\ map\ (compose\ f\ g)\ xs &= map\ f\ (map\ g\ xs) \\ map\ f\ (concat\ xss) &= concat\ (map\ (map\ f)\ xss) \end{aligned}$$

ESERCIZIO 9.7

Si consideri la funzione `intervall n`, che rappresenta l'intervallo `[1;...;n]`. Si dimostri che la seguente definizione è corretta.

```
# let rec intervall x =
  match x with
  0 -> []
  | n -> 1::map ((prefix +) 1) (intervall (n-1));;
intervall : int -> int list = <fun>
```

9.2 – Definizioni ausiliarie e generalizzazioni

A volte, invece di definire direttamente una funzione o dimostrare direttamente un teorema, è più conveniente o necessario definire altre funzioni o dimostrare altri risultati. In altre parole, possiamo aver bisogno di definizioni *ausiliarie*. In altre situazioni, possiamo rendere il nostro obiettivo più semplice da raggiungere se cerchiamo di definire un risultato più generale o cerchiamo di dimostrare un risultato più generale. In tal caso, abbiamo bisogno di *generalizzazioni* di definizioni o teoremi. In questa sezione vedremo alcuni esempi di tali tecniche.

9.2.1 – Differenza di liste

Nel capitolo precedente abbiamo visto che la differenza tra due liste `xs` e `ys` è definita come la lista ottenuta da `xs` rimuovendo la prima occorrenza di ogni `y` appartenente a `ys`. Una definizione ricorsiva di `diff` è la seguente:

```
# let rec diff xs ys =
  match ys with
  [] -> xs
  | z::zs -> diff (remove xs z) zs;;
diff : 'a list -> 'a list -> 'a list = <fun>
```

Come si può vedere, è conveniente definire `diff` in termini di una definizione ausiliaria, la funzione `remove`:

```
# let rec remove xs y =
  match xs with
  [] -> []
  | z::zs when z=y -> zs
  | z::zs when z<>y -> z::remove zs y;;
remove : 'a list -> 'a -> 'a list = <fun>
```

Una importante legge che lega concatenazione e differenza di liste è la seguente:

$$diff(xs @ ys) xs = ys$$

La dimostrazione è immediata, utilizzando le tecniche della sezione precedente, ed è lasciata come esercizio.

9.2.2 – La funzione reverse

Nel capitolo precedente abbiamo visto come definire la funzione `rev`, per mezzo degli operatori di `fold`. Una definizione diretta è la seguente:

```
# let rec rev lst =
  match lst with
  [] -> []
  | x::xs -> rev xs@[x];;
rev : 'a list -> 'a list = <fun>
```

Dimostriamo che

$$rev(rev xs) = xs$$

per ogni lista `xs`.

Dimostrazione. Per dimostrare l'enunciato, abbiamo bisogno di un risultato ausiliario:

$$rev(ys@[x]) = x :: rev ys$$

Tale risultato è facilmente dimostrabile per induzione su `ys`, ed è lasciato come esercizio.

Utilizzando il risultato di cui sopra, possiamo semplificare la dimostrazione del nostro enunciato. Tale dimostrazione viene effettuata per induzione su `xs`:

Caso []. In tal caso:

$$\begin{aligned}
 & \text{rev}(\text{rev}[\]) \\
 = & \quad \{\text{primo pattern di rev}\} \\
 & \text{rev}[\] \\
 = & \quad \{\text{primo pattern di rev}\} \\
 & [].
 \end{aligned}$$

Caso $x :: xs$. In tal caso:

$$\begin{aligned}
 & \text{rev}(\text{rev } x :: xs) \\
 = & \quad \{\text{secondo pattern di rev}\} \\
 & \text{rev}((\text{rev } xs)@[x]) \\
 = & \quad \{\text{per il risultato di cui sopra}\} \\
 & x :: \text{rev}(\text{rev } xs) \\
 = & \quad \{\text{per ipotesi induttiva}\} \\
 & x :: xs.
 \end{aligned}$$

Nella dimostrazione principale, abbiamo utilizzato il risultato ausiliario per dimostrare che

$$\text{rev}((\text{rev } xs)@[x]) = x :: \text{rev}(\text{rev } xs)$$

Si può verificare che non è affatto semplice cercare di dimostrare tale risultato direttamente. Sostituendo invece la sottoespressione $\text{rev } xs$ con una nuova variabile ys otteniamo un risultato *più generale* di quello voluto, oltretutto più facile da dimostrare.

Non esistono regole generali per stabilire quando e come un risultato ausiliario debba essere individuato. Il modo più naturale è dato dall'intuizione: poiché $x :: xs$ è equivalente a $[x]@xs$, si potrebbe provare a trarre dal secondo pattern di rev la seguente uguaglianza:

$$\text{rev}([x]@xs) = (\text{rev } xs)@[x]$$

Da tale uguaglianza si può derivare un'uguaglianza praticamente simile

$$\text{rev}(xs@[x]) = [x]@\text{rev } xs$$

che è il risultato ausiliario di cui avevamo bisogno.

9.2.3 – Ottimizzazione

Come sappiamo, possiamo pensare ad una computazione come ad una serie di passi di riduzione, ognuno dei quali consiste nell'applicare un pattern della definizione di una funzione (o di un identificatore). Per esempio, per calcolare $[3; 2] @ [1]$, abbiamo bisogno di tre passi di riduzione:

```
[3;2]@[1]
↪ { secondo pattern di @}

3::([2]@[1])

↪ { secondo pattern di @}

3::(2::([]@[1]))

↪ { primo pattern di @}

3::2::[1].
```

In generale, se xs ha lunghezza n allora il calcolo di $xs@ys$ richiederà n riduzioni derivanti dall'applicazione del secondo pattern di $@$, seguite da una riduzione derivante dall'applicazione del primo pattern di $@$; da ciò si deduce che il numero di riduzioni è proporzionale a n .

Si può notare che il calcolo di $rev [1; 2; 3]$ richiede 10 riduzioni:

```
rev [1;2;3]
↪ { secondo pattern di rev}

rev [2;3]@[1]
↪ { secondo pattern di rev}

(rev [3]@[2])@[1]
↪ { secondo pattern di rev}

((rev []@[3])@[2])@[1]
↪ { primo pattern di rev}

(([]@[3])@[2])@[1]
↪ { primo pattern di rev}

([3]@[2])@[1]
↪ { secondo e primo pattern di @}

[3;2]@[1]
↪ { secondo e primo pattern di @}

[3;2;1].
```

In generale, se xs ha lunghezza n , allora il calcolo di $rev xs$ richiederà n riduzioni ottenute applicando il secondo pattern di rev , seguite da una riduzione ottenuta applicando il primo pattern

`rev` e da $1 + 2 + \dots + n = \frac{n(n-1)}{2}$ passi di riduzione per calcolare le concatenazioni risultanti. Da ciò emerge che il numero di riduzioni è proporzionale a n^2 . Ovviamente tale metodo per calcolare `rev` è dispendioso, poiché è possibile invertire una lista di n elementi in un numero di passi di riduzione proporzionale a n .

Si consideri la funzione

```
# let reverse xs = revit xs [];;
```

dove la funzione `revit` è definita come:

```
# let rec revit lst1 lst2 =
  match lst1 with
  [] -> lst2
  | x::xs -> revit xs (x::lst2);;
```

La funzione `reverse`, che fa uso della funzione ausiliaria `revit`, è equivalente alla funzione `rev`. Infatti è possibile dimostrare che

$$rev\ xs = reverse\ xs$$

utilizzando il risultato ausiliario

$$revit\ xs\ ys = (rev\ xs)@ys$$

La dimostrazione di questi due risultati può essere ottenuta per induzione, utilizzando le tecniche note, ed è lasciata per esercizio.

Se proviamo a calcolare `reverse [1;2;3]`, otteniamo:

```
reverse [1;2;3]
↪ {definizione di reverse}
revit [1;2;3] []
↪ {secondo pattern di revit}
revit [2;3] [1]
↪ {secondo pattern di revit}
revit [3] [2;1]
↪ {secondo pattern di revit}
revit [] [3;2;1]
↪ {primo pattern di revit}
[3;2;1].
```

In generale, è facile verificare che il calcolo di `reverse xs` richiede $n + 2$ passi di riduzione, dove n è la lunghezza di xs .

Un altro esempio di ottimizzazione è dato dal programma per il calcolo dell'ennesimo numero di Fibonacci. Non ci addentreremo nella valutazione dell'efficienza del programma che abbiamo definito nei paragrafi precedenti. Si può notare, tuttavia, che il calcolo dell'ennesimo numero di Fibonacci è piuttosto dispendioso: per esempio, il calcolo di `fib 7` richiede 21 passi di riduzione. Utilizzando la tecnica delle definizioni ausiliarie, si può calcolare l'ennesimo numero di Fibonacci con un numero di passi di riduzione proporzionale a n :

```
# let fastfib n = fibgen 0 1 n;;
```

dove `fibgen` è definita come:

```
# let rec fibgen a b x =
  match x with
  0 -> a
  | n when n>0 -> fibgen b (a+b) (n-1);;
```

È facile (e lasciato per esercizio) dimostrare per induzione che $fastfib n = fib n$.

ESERCIZIO 9.8

Si dimostrino le seguenti proprietà:

$$\begin{aligned} diff(xs@ys)xs &= ys \\ rev(xs@ys) &= (rev ys)@(rev xs) \\ fold_right f a (xs@ys) &= fold_right f (fold_right f a ys) xs \end{aligned}$$

9.2.4 – Funzioni combinatorie

Molti problemi interessanti sono *combinatori* per natura, in quanto richiedono la selezione e/o permutazione degli elementi di una lista in un qualche modo. Vediamo alcune funzioni che permettono di effettuare tali manipolazioni.

Possiamo definire una funzione `inits` che calcola la lista di tutti i segmenti iniziali di una lista data, per lunghezza crescente: l'implementazione si avvale delle funzioni `map` (precedentemente analizzata) e `cons`:

```
# let cons x lst = x::lst;;
cons : 'a -> 'a list -> 'a list = <fun>

# let rec inits lst = match lst with
  [] -> []
  | x::xs -> [[]] @ map (cons x) (inits xs);;
inits : 'a list -> 'a list list = <fun>

# inits [1;2;3];;
- : int list list = [[]; [1]; [1; 2]; [1; 2; 3]]
```

Infatti si ha:

```
inits [1;2;3] ↵
[[]] @ map (cons 1) (inits [2;3]) ↵
[[]] @ map (cons 1) ([[]] @ map (cons 2) (inits [3])) ↵
[[]] @ map (cons 1) ([[]] @ map (cons 2) ([[]] @ map (cons 3)
(inits []))) ↵
[[]] @ map (cons 1) ([[]] @ map (cons 2) ([[]] @ map (cons 3
[[]]))) ↵
[[]] @ map (cons 1) ([[]] @ map (cons 2) ([[]] @ [[3]])) ↵
[[]] @ map (cons 1) ([[]] @ map (cons 2) [[], [3]])) ↵
[[]] @ map (cons 1) ([[]] @ [[2], [2;3]])) ↵
[[]] @ map (cons 1) [[], [2], [2;3]] ↵
[[]] @ [[1], [1;2], [1;2;3]] ↵
[[], [1], [1;2], [1;2;3]]
```

La funzione `inits` soddisfa la seguente proprietà: la lista vuota ha come segmento iniziale sé stessa, mentre una lista $x :: xs$ ha ancora la lista vuota come segmento iniziale, e gli altri segmenti iniziali saranno i segmenti iniziali di xs con in testa x .

La funzione `subs` restituisce la lista di tutte le sotosequenze di una lista data:

```
# let rec subs lst =
    match lst with
    [] -> [[]]
    | x::xs -> (subs xs) @ map (cons x) (subs xs);;
subs : 'a list -> 'a list list = <fun>

# subs [1;2;3];
- : int list list =
[[], [3], [2], [2; 3], [1], [1; 3], [1; 2], [1; 2; 3]]
```

Se xs ha lunghezza n , allora `subs xs` ha lunghezza 2^n . Il primo pattern indica che la lista vuota ha sé stessa come unica sotosequenza; dal secondo pattern, una lista $x :: xs$ ha come sotosequenze tutte le sotosequenze di xs , con o senza l'elemento x .

La funzione `interleave x ys` restituisce la lista di tutte le possibili liste ottenute da ys inserendo il termine x :

```
# let rec interleave x lst =
    match lst with
    [] -> [[x]]
    | y::ys -> [x::y::ys] @ map (cons y) (interleave x ys);;
interleave : 'a -> 'a list -> 'a list list = <fun>

# interleave 1 [2;3;4];
- : int list list =
[[1; 2; 3; 4], [2; 1; 3; 4], [2; 3; 1; 4], [2; 3; 4; 1]]
```

Nel secondo pattern, i modi per inserire x in una lista $y :: ys$ consistono o nel mettere x in testa alla lista $y :: ys$ oppure nell'inserire x in ys .

La funzione `perms` calcola tutte le possibili permutazioni di una lista. Essa fa uso delle seguenti funzioni `concat`:

```
# let rec concat lst =
  match lst with
  [] -> []
  | xs:::ys -> xs @ (concat ys);;
concat : 'a list list -> 'a list = <fun>

# let rec perms lst =
  match lst with
  [] -> [[]]
  | x:::xs -> concat (map (interleave x) (perms xs));;
perms : 'a list -> 'a list list = <fun>

# perms [1;2;3];;
- : int list list =
[[1; 2; 3]; [2; 1; 3]; [2; 3; 1]; [1; 3; 2]; [3; 1; 2]; [3; 2; 1]]
```

Le permutazioni di una lista non vuota $x :: xs$ sono date da tutti gli intercalamenti del termine x nelle permutazioni di xs . L'uso di `concat` nella definizione è essenziale. I tipi di `perms` e `interleave` sono:

```
perms : 'a list -> 'a list list
interleave : 'a -> 'a list -> 'a list list
```

Assumendo $x : 'a$ e $xs : 'a$ list, otteniamo:

```
(map (interleave x) (perms xs)) : 'a list list list
```

ESERCIZIO 9.9

La funzione `choose k xs` restituisce la lista di tutte le sottosequenze di xs la cui lunghezza è esattamente k .

Per esempio:

```
# choose 2 [1;2;3];;
- : int list list = [[1; 2]; [1; 3]; [2; 3]]
```

Si dia una definizione ricorsiva della funzione `choose`.

ESERCIZIO 9.10

Si dimostri la seguente proprietà:

$$\text{subs} (\text{map } f \text{ } xs) = \text{map} (\text{map } f) (\text{subs } xs).$$

9.3 – Induzione Ben Fondata

Tipicamente, le funzioni ricorsive esprimibili in un qualsiasi linguaggio funzionale trattano oggetti generici (ad esempio dei tipi di dato più complessi di quelli da noi analizzati). Inoltre, come abbiamo visto negli esempi precedenti, non è affatto detto che la ricorsione calcoli $f(n)$ in funzione di $f(m)$ esclusivamente per numeri $m < n$. Il principio di induzione, nelle forme che sinora abbiamo visto, può essere generalizzato, considerando un insieme qualsiasi ed una relazione generica che esprima una certa “precedenza”, o “ordine”, tra gli elementi dell’insieme. Essenziale, come tutti gli esempi mostrano, è il riconoscimento corretto di uno o più *casi base*, ovvero come i casi “più semplici da trattare”, ed una opportuna *ipotesi induttiva*.

Il primo problema da formalizzare è quello di generalizzare adeguatamente il concetto di *caso base* visto negli esempi precedenti. Sui naturali è ovvio ricondurre il caso base all’ordinamento sui numeri: il caso più semplice da trattare è il caso $n = 0$, mentre la dimostrazione per il caso $n > 0$ viene ricondotta alla dimostrazione per il caso $m < n$. Per una generalizzazione efficace di questo concetto, in particolare del concetto di *essere più semplice* per un elemento di un insieme rispetto ad un altro, utilizziamo la definizione di *relazione* tra gli elementi di un insieme.

Dato un insieme S , una relazione R su S è un insieme di coppie di elementi di S . Nel seguito useremo la notazione infissa $x R y$ per denotare il fatto che x precede y secondo la relazione R , e denoteremo una relazione R che esprime una precedenza indifferentemente con i simboli \prec o \sqsubset .

Nel caso dei numeri naturali, la relazione usuale di precedenza è data dall’ordine naturale imposto tra i numeri. In tale relazione, l’elemento più semplice è unicamente determinato come l’elemento più piccolo tra tutti i naturali, ovvero come l’unico elemento n tale che per ogni m non vale la relazione $m < n$. Inoltre, l’ordine naturale non contiene cicli, cioè non esiste alcuna coppia m ed n con $m \neq n$ tali che $m < n$ e $n < m$. Se la relazione \prec fosse ciclica, allora il principio di induzione che abbiamo analizzato nelle sezioni precedenti non sarebbe più applicabile: infatti, non potremmo dimostrare $P(m)$ riconducendolo alla dimostrazione di $P(n)$, poiché la dimostrazione di $P(n)$ sarebbe a sua volta riconducibile a quella di $P(m)$, ovvero alla proprietà che volevamo dimostrare originariamente.

Possiamo cercare di generalizzare tali nozioni, considerando come elemento base o *minimale* ogni elemento m di S tale che non esiste alcun $s \in S$ per cui $s \sqsubset m$. L’idea intuitiva è quella di strutturare l’insieme dato introducendo una relazione di “precedenza” non ciclica tra i suoi elementi. L’uso di relazioni di precedenza verrà considerato come la generalizzazione del concetto di ordinamento tra naturali. In base a ciò, diremo che un elemento di S è minimale se nessun elemento lo precede in \sqsubset .

Se (S, \sqsubset) è un insieme con una relazione di precedenza, dati $x, y \in S$ con $x \sqsubset y$, diremo che x è il *predecessore immediato* di y . Definiamo inoltre una *catena* come una sequenza di elementi $\dots \sqsubset s_{m-1} \sqsubset s_m \sqsubset s_{m+1} \sqsubset \dots$ tali che s_i è il predecessore immediato di s_{i+1} , per ogni i relativo ad un elemento della sequenza.

Si consideri ad esempio l’insieme \mathbb{N} dei numeri naturali, e si consideri la relazione “ $x \sqsubset y$ se e solo se $y = x + 1$ ”. Possiamo allora considerare \mathbb{N} stesso come la catena $0 \sqsubset 1 \sqsubset 2 \sqsubset 3 \sqsubset \dots$. Si noti che tale catena è infinita ascendente, ma non infinita discendente (non esiste un predecessore di 0). Se consideriamo invece l’insieme \mathbb{Z} dei numeri interi con la stessa relazione \sqsubset , allora l’insieme dei numeri negativi è una catena infinita discendente (infatti, $\dots \sqsubset -2 \sqsubset -1 \sqsubset 0$). Si consideri ora

l'insieme $S = \{1/n \mid n \geq 1\}$ con la relazione " $x < y$ se e solo se esiste un numero naturale k tale che $y = k * x$ ". Ad esempio, $1/4 < 1/2$, poiché $1/2 = 2 * 1/4$. Si considerino le catene $\{1/p^k \mid p \text{ è un numero primo e } k \geq 1\}$ dell'insieme S . Tali catene sono discendenti infinite: ad esempio, la catena $\dots < 1/8 < 1/4 < 1/2$ è tale che $1/2 = 2 * 1/4$, $1/4 = 2 * 1/8$, e così via, è dunque una catena discendente infinita.

Si noti che se una relazione di precedenza \sqsubset contiene dei *cicli*, come ad esempio $1 \sqsubset 2$, $2 \sqsubset 5$ e $5 \sqsubset 1$, essa induce catene discendenti infinite. Nell'esempio, è possibile costruire la catena discendente infinita $\dots \sqsubset 2 \sqsubset 5 \sqsubset 1 \sqsubset 2 \sqsubset 5 \sqsubset 1$.

La presenza o meno di catene discendenti infinite è una proprietà tipica dell'insieme rispetto alla relazione di precedenza che viene definita. L'insieme \mathbf{N} , per esempio, non ha catene discendenti infinite se consideriamo la relazione \sqsubset definita sopra. Tuttavia, se proviamo ad invertire la relazione in " $x \sqsubset' y$ se e solo se $x = y + 1$ ", allora la catena discendente infinita $\dots \sqsubset' 2 \sqsubset' 1 \sqsubset' 0$ è propria dell'insieme \mathbf{N} con la relazione \sqsubset' .

Si noti come la presenza di una catena discendente infinita renda impossibile l'applicazione del principio di induzione su \mathbf{Z} . Infatti, se per dimostrare una proprietà $P(n)$ per qualche n , cercassimo di ridurla alla dimostrazione di $P(m)$, per $m \sqsubset n$, rischieremmo di trovarci su una catena discendente infinita, e finiremmo quindi col tentare all'infinito di ridurre la dimostrazione di $P(n_i)$ alla dimostrazione di $P(n_{i+1})$, dove $n_{i+1} \sqsubset n_i$. Questo ci suggerisce di imporre l'assenza di catene discendenti infinite nell'insieme S che andiamo ad esaminare.

In una relazione di precedenza, un elemento può avere infiniti predecessori immediati, senza che per questo la relazione induca una catena discendente infinita. Si consideri ad esempio l'insieme \mathbf{N} con la relazione di precedenza " $x < y$ se e solo se $y = 0$ e $x \neq 0$ ". In tal caso, per ogni numero naturale n avremo $n < 0$, e conseguentemente, tutte le catene saranno composte dai soli elementi n e 0 . In questo caso 0 ha un numero infinito di predecessori ma non ci sono catene discendenti infinite.

Dato un insieme S ed una relazione di precedenza \sqsubset , diremo che l'insieme S è *ben fondato* se e solo se non ammette catene discendenti infinite in \sqsubset . Negli esempi precedenti, (\mathbf{N}, \sqsubset) e $(\mathbf{N}, <)$ sono insiemi ben fondati. Non sono insiemi ben fondati, invece, (\mathbf{Z}, \sqsubset) , $(\{1/n \mid n \geq 1\}, <)$ e (\mathbf{N}, \sqsubset') .

A questo punto abbiamo tutti gli elementi per definire la generalizzazione del principio di induzione che abbiamo visto nelle sezioni precedenti. Consideriamo il principio di induzione sui numeri naturali. Per dimostrare una proprietà $P(m)$ su un naturale m , si tratta di dimostrare che la proprietà vale per $m = 0$ e dimostrare quindi che se vale $P(n)$ per ogni $n < m$, allora vale anche $P(m)$. Il principio si basa sull'osservazione che se vale $P(0)$ allora vale $P(1)$; se vale $P(1)$ allora vale $P(2)$ e così via, per tutti i numeri naturali. La generalizzazione che vogliamo introdurre è praticamente identica: se l'insieme S che andiamo a considerare è ben fondato con la relazione \sqsubset , allora non esistono catene discendenti infinite rispetto a \sqsubset . A questo punto, se siamo in grado di dimostrare che:

- $P(m)$ vale per tutti gli m minimali rispetto a \sqsubset
- con m non minimale, se per ogni $n \sqsubset m$ vale $P(n)$, allora vale anche $P(m)$

abbiamo dimostrato che la proprietà P vale per ogni $m \in S$. Infatti, preso un qualsiasi elemento m , questo elemento apparterrà ad una catena, che, non essendo discendente infinita, conterrà un

elemento iniziale m_0 . Supponiamo che m sia l' i -esimo elemento della catena. Dimostriamo che vale $P(m_0)$. Allora vale anche $P(m_1)$. Continuando i volte con l'applicazione del principio, alla fine dimostriamo che vale $P(m)$. Tale principio di induzione è detto *principio di induzione ben fondata*, perché si basa su insiemi ben fondati.

Si noti che, in pratica, la dimostrazione di una proprietà P basata sul principio di induzione ben fondata avviene in due passi.

Caso Base: si dimostra che P vale su tutti gli elementi di S minimali rispetto a \sqsubseteq , cioè tutti gli elementi che non hanno predecessori immediati;

Caso Induttivo: detto x un generico elemento di S , non minima rispetto a \sqsubseteq , si dimostra la validità di $P(x)$ utilizzando, laddove si renda necessaria, l'ipotesi induttiva, ovvero l'ipotesi che P valga su tutti gli elementi y di S tali che $y \sqsubseteq x$.

È facile verificare che l'induzione sui numeri naturali è un caso particolare dell'induzione ben fondata: si noti che la relazione $<$ corrisponde alla chiusura transitiva della relazione \sqsubseteq definita sopra, e quindi l'insieme $(\mathbb{N}, <)$ è ben fondato. In particolare, tale insieme ha una sola catena $0 \sqsubseteq 1 \sqsubseteq 2 \sqsubseteq 3 \sqsubseteq \dots$.

Ovviamente, anche l'induzione sulle liste è un caso particolare di induzione ben fondata. Consideriamo infatti l'insieme L di tutte le liste, con la relazione \sqsubseteq definita come " $xs \sqsubseteq ys$ se e solo se esiste un x tale che $ys = x :: xs$ ". La relazione \sqsubseteq è una relazione di precedenza, poiché non possono esistere due liste xs e ys con $xs \neq ys$ tali che $xs = x_1 :: ys$ e $ys = x_2 :: xs$, dati x_1 e x_2 . Inoltre, ogni catena $\{xs_i\}_{i \geq 0}$ è discendente finita, poiché $[] \sqsubseteq [x]$ per ogni x e non esiste alcuna lista xs tale che $xs \sqsubseteq []$. Da ciò si può dedurre che l'insieme (L, \sqsubseteq) è ben fondato, e il principio di induzione ben fondata può essere applicato.

9.3.1 – Induzione Ben Fondata e Funzioni Ricorsive

Abbiamo già visto come l'induzione naturale permetta in molti casi di stabilire se la definizione di una funzione ricorsiva sia valida (ovvero se la funzione sia ben definita). In generale, l'utilizzo dell'induzione ben fondata può, in molti casi, permetterci di stabilire delle proprietà di funzioni ricorsive. Cerchiamo di dare una metodologia generale per stabilire ciò.

Un criterio generale per stabilire se la definizione ricorsiva sia valida è quello di verificare se la funzione sia definita su catene indotte da una relazione di precedenza ben fondata. Si tratta di verificare che

- la funzione sia definita per i casi base, ovvero su tutti gli elementi minimali del dominio, e
- per i casi non minimi, la funzione applicata ad un elemento x sia definita in termini dei predecessori immediati di x nel dominio.

Se queste due proprietà sono verificate, il principio di induzione ben fondata ci permette di concludere che la funzione è definita su tutto il dominio.

Tale tecnica può essere generalizzata alla dimostrazione di proprietà di funzioni ricorsive. Il procedimento da seguire può essere elencato come segue:

- si individua una relazione \sqsubseteq su D (il dominio della funzione) che esprima una relazione di precedenza e tale che (D, \sqsubseteq) sia ben fondato;
- si dimostra che la funzione è definita sulle catene di D indotte da \sqsubseteq (in tal modo si dimostra che la funzione è ben definita);
- si dimostra per induzione ben fondata che la funzione gode della proprietà data (in tal modo si dimostra la correttezza della funzione).

Si consideri, ad esempio, la funzione `pari`, definita come:

```
# let rec pari x = match x with
| 0 -> true
| 1 -> false
| n when n>1 -> pari (n-2);
pari : int -> bool = <fun>
```

Si noti che la funzione, benché abbia tipo `int -> bool`, è definita sul dominio \mathbf{N} . Consideriamo allora, su \mathbf{N} , la relazione " $x \sqsubseteq y$ se e solo se $y = x + 2$ ". La coppia $(\mathbf{N}, \sqsubseteq)$ rappresenta un insieme ben fondato poiché le uniche catene sono la catena $0 \sqsubseteq 2 \sqsubseteq 4 \sqsubseteq 6 \sqsubseteq \dots$ dei numeri pari e la catena $1 \sqsubseteq 3 \sqsubseteq 5 \sqsubseteq 7 \sqsubseteq \dots$ dei numeri dispari, ed entrambe sono discendenti finite. Si noti ora che gli elementi minimi del dominio sono 0 e 1, e che su tali elementi la funzione è definita. Inoltre, si consideri un elemento non minima x : allora $x - 2 \sqsubseteq x$, e la funzione su x è definita in termini della sua applicazione al predecessore immediato $x - 2$. Queste osservazioni ed il principio di induzione ben fondata garantiscono che la funzione è definita su tutti gli elementi di \mathbf{N} .

Consideriamo ora una possibile definizione della funzione `mod` su coppie di numeri naturali:

```
# let rec mod x y =
    if x < y then x else mod (x-y) y;;
mod : int -> int -> int = <fun>
```

Consideriamo la relazione di precedenza " $(x, y) \sqsubseteq (x', y')$ se $y = y'$ e $x' = x + y$ ". L'insieme $(\mathbf{N} \times \mathbf{N}, \sqsubseteq)$ è ben fondato: infatti le coppie del tipo (n, m) con $n < m$ sono minimi e le catene sono del tipo:

$$(0, m) \sqsubseteq (m, m) \sqsubseteq (2m, m) \sqsubseteq \dots$$

$$(1, m) \sqsubseteq (m+1, m) \sqsubseteq (2m+1, m) \sqsubseteq \dots$$

...

$$(m-1, m) \sqsubseteq (2m-1, m) \sqsubseteq (3m-1, m) \sqsubseteq \dots$$

Dimostriamo ora, che per ogni numero naturale x ,

$$pari x \equiv (mod x 2 = 0).$$

Dimostriamo l'asserto per induzione ben fondata su $(\mathbf{N}, \sqsubseteq)$.

Caso Base. I casi base sono due: il caso $x = 0$ e il caso $x = 1$.

Se $x = 0$, allora

pari 0
 \equiv {primo pattern di *pari*}

true
 \equiv {riflessività}
 $0 = 0$
 \equiv {definizione di *mod*}
 $(mod\ 0\ 2) = 0.$

Se invece $x = 1$, allora

pari 1
 \equiv {secondo pattern di *pari*}
false
 \equiv {calcolo}
 $1 = 0$
 \equiv {definizione di *mod*}
 $(mod\ 1\ 2) = 0.$

Caso Induttivo. Supponiamo che x sia diverso da 0 e da 1. Dimostriamo allora la proprietà nell'ipotesi che $pari(x - 2) \equiv (mod(x - 2)2) = 0$.

pari x
 \equiv {terzo pattern di *pari*}
pari $(x - 2)$
 \equiv {ipotesi induttiva, poiché $x - 2 \sqsubset x$ }
 $(mod(x - 2)2) = 0$
 \equiv {secondo caso di *mod*}
 $(mod x 2) = 0.$

Si noti che, nel dimostrare che entrambe le funzioni sono definite, abbiamo utilizzato due diverse relazioni su due diversi insiemi ben fondati. In effetti, per una corretta applicazione del principio di induzione ben fondata, è necessario individuare la relazione di precedenza più adeguata a dimostrare la proprietà voluta.

Consideriamo ora la seguente definizione:

```
# let rec maxl lst =
  match lst with
  | [x] -> x
  | x::y::xs when x<=y -> maxl (y::xs)
  | x::y::xs when x>y -> maxl (x::xs);;
maxl : 'a list -> 'a = <fun>
```

La funzione *maxl* calcola il massimo degli elementi di una lista non vuota. Proviamo a dimostrare che la funzione è corretta. Vogliamo dimostrare cioè che, per ogni lista non vuota *xs*, vale la proprietà

$$\text{maxl } xs \in xs \wedge (\forall z. z \in xs \Rightarrow z \leq \text{maxl } xs).$$

Possiamo provare a dimostrarlo per induzione ben fondata su $xs \in L$, dove L è l'insieme di tutte le possibili liste *non vuote*, utilizzando la relazione di precedenza \sqsubset così definita:

$$l \sqsubset l' \text{ se e solo se } tl\ l = tl\ (tl\ l').$$

Si noti che, in tale relazione, gli elementi minimali sono tutte e sole le liste singoletto.

Caso Base. Consideriamo una generica lista singoletto $[x]$.

$$\text{maxl } [x] \in [x] \wedge (\forall z. z \in [x] \Rightarrow z \leq \text{maxl } [x])$$

$$= \{ \text{primo pattern di } \text{maxl}, \text{ singoletto} \}$$

$$x \in [x] \wedge x \leq x$$

$$= \{ \text{singoletto, calcolo} \}$$

true.

Caso Induttivo. Consideriamo ora una generica lista $x :: y :: xs$ e dimostriamo la proprietà per casi, a seconda che $x \leq y$ oppure $x > y$. Sia $x \leq y$:

$$\text{maxl } x :: y :: xs \in x :: y :: xs \wedge (\forall z. z \in x :: y :: xs \Rightarrow z \leq \text{maxl } x :: y :: xs)$$

$$= \{ \text{secondo pattern di } \text{maxl}, x \leq y \}$$

$$\text{maxl } y :: xs \in x :: y :: xs \wedge (\forall z. z \in x :: y :: xs \Rightarrow z \leq \text{maxl } y :: xs)$$

$$= \{ \text{proprietà delle liste} \}$$

$$((\text{maxl } y :: xs \in y :: xs) \vee (\text{maxl } y :: xs = x)) \wedge (x \leq \text{maxl } y :: xs) \\ \wedge (\forall z. z \in y :: xs \Rightarrow z \leq \text{maxl } y :: xs)$$

$$= \{ \text{ipotesi induttiva} \}$$

$(\text{true} \vee (\text{maxl } y :: xs = x)) \wedge (x \leq \text{maxl } y :: xs) \wedge \text{true}$

= {proprietà degli operatori logici}

$(x \leq \text{maxl } y :: xs)$

= {per ipotesi induttiva $y \leq \text{maxl } y :: xs$, ipotesi $x \leq y$, transitività di \leq }

true.

Il caso $x > y$ è analogo e lasciato per esercizio.

ESERCIZIO 9.11

Si riesamino gli esempi delle sezioni precedenti e per ognuno di essi si stabiliscano formalmente il dominio S e la relazione di precedenza \sqsubset . Si dimostri che la coppia (S, \sqsubset) rappresenta un insieme ben fondato.

ESERCIZIO 9.12

Siano (D, \sqsubset) un insieme con una relazione di precedenza. Dato $X \subseteq D$, un elemento $m \in X$ è minimale per X se per ogni elemento $d \in D$ tale che $d \sqsubset m$, $d \notin X$. Dimostrare che (D, \sqsubset) è ben fondato se e solo se ogni sottoinsieme X non vuoto di D ammette un minimaile.

Suggerimento: si dimostri che, se esistesse una catena discendente infinita, l'insieme generato da tale catena non potrebbe avere un minimaile. Viceversa ...

ESERCIZIO 9.13

Si modifichi la funzione `pari` in modo che sia definita anche su interi negativi.

ESERCIZIO 9.14

Sia $\hat{}$ l'operazione di concatenazione tra stringhe. Si dimostri per induzione ben fondata che date due stringhe s_1 e s_2 , se $s_1 \neq s_2$ allora non esiste alcuna stringa s tale che $s_1 \hat{ } s = s_2 \hat{ } s$.

ESERCIZIO 9.15

Siano (S_1, \sqsubset_1) e (S_2, \sqsubset_2) due insiemi ben fondati. Si dimostri che l'insieme $(S_1 \times S_2, \sqsubset)$ con la relazione

$$(s_1, s_2) \sqsubset (s'_1, s'_2) \Leftrightarrow s_1 \sqsubset_1 s'_1 \text{ e } s_2 \sqsubset_2 s'_2$$

è ben fondato.

Capitolo 10 – Tipi Ricorsivi e Alberi binari

10.1 – I Tipi Ricorsivi

Consideriamo ora una importante generalizzazione del tipo somma: il *tipo ricorsivo*. Un tipo ricorsivo è un tipo definito in termini di sé stesso. In effetti, nel corso della trattazione dell’induzione abbiamo implicitamente sfruttato il fatto che alcuni tipi utilizzati potessero essere definiti ricorsivamente. I numeri naturali, infatti, possono essere definiti *ricorsivamente*: un numero naturale o è 0 oppure è il successore di un altro numero naturale. Allo stesso modo, una lista è `[]` oppure è il risultato della concatenazione di un elemento ad una lista. Nelle sezioni precedenti abbiamo ampiamente utilizzato tali nozioni per definire funzioni sui naturali o sulle liste, e abbiamo definito le tecniche di ragionamento per induzione basandoci sul presupposto che l’insieme dei valori possibili fosse costruito ricorsivamente (ovvero, che la ricorsività dell’insieme inducesse una relazione di precedenza).

L’insieme **N** dei numeri naturali è definito ricorsivamente nel seguente modo:

- i) $0 \in \mathbf{N}$
- ii) per ogni $n \in \mathbf{N}$, $\text{succ}(n) \in \mathbf{N}$
- iii) gli unici elementi di **N** sono quelli che si ottengono mediante i) e ii).

Il tipo **Nat** può essere introdotto esplicitamente nel seguente modo:

```
# type Nat = Zero | Succ of Nat;;  
Type Nat defined.
```

definandolo, cioè, come la somma tra un tipo enumerato (che ha il solo valore `Zero`) e il tipo costruito a partire da sé stesso. Analogamente, una lista di numeri naturali può essere definita come:

```
# type NatList = Empty | Cons of Nat * NatList;;  
Type NatList defined.
```

I valori appartenenti al tipo **Nat** possono essere enumerati nel seguente modo:

```
Zero  
Succ Zero  
Succ (Succ Zero)  
Succ (Succ (Succ Zero))  
⋮
```

Sui tipi definiti ricorsivamente, è possibile definire un ulteriore caso particolare dell’induzione ben fondata: l’*induzione strutturale*. In tale principio, la relazione di precedenza tra gli elementi di un dominio è indotta dalla struttura che appare nella definizione. Nel caso di **Nat**, la relazione indotta dalla struttura è: “ $x \sqsubset y$ se e solo se $y = \text{Succ } x$ ”. Ad esempio, `Zero ⊂ Succ Zero` poiché `Succ Zero` è strutturalmente “più ricco” di `Zero` (ossia, è ricavato da `Zero` stesso).

Sui tipi di dati definiti induttivamente si possono definire operazioni ricorsivamente:

```
# let rec int_of_nat x = match x with
    Zero -> 0
    | Succ k -> 1 + (int_of_nat k);;
int_of_nat : Nat -> int = <fun>

# int_of_nat (Succ (Succ (Succ( Succ Zero))));;
- : int = 4
```

L'operazione di somma sui numeri naturali si può definire ricorsivamente sul primo argomento:

```
# let rec somma (n, m) = match n with
    Zero -> m
    | Succ k -> Succ (somma(k, m));;
somma : Nat * Nat -> Nat = <fun>

# let k = somma(Succ(Succ Zero), Succ(Succ(Succ Zero)));}
k : Nat = Succ (Succ (Succ (Succ (Succ Zero)))))

# int_of_nat k;;
- : int = 5
```

10.2 - I Tipi Ricorsivi Polimorfi

Introduciamo, infine, i tipi ricorsivi polimorfi, ovvero tipi costruiti parametricamente ad altri tipi. Un esempio classico è dato dalle liste, che sono definite parametricamente al tipo degli elementi che le compongono. Una definizione generale di lista è data da

```
# type 'a List = Empty | Cons of 'a * 'a List;;
Type List defined.

#Empty;;
- : 'a List = Empty

#Cons (1, Empty);;
- : int List = Cons (1, Empty)
```

Proviamo a ridefinire la funzione `length` sulle liste:

```
# let rec length l = match l with
    Empty -> 0
    | Cons (x, l) -> length l + 1;;
length : 'a List -> int = <fun>
```

L'utilizzo dei tipi ricorsivi polimorfi permette di definire agevolmente entità matematiche complesse, come la prossima sezione esemplifica.

10.3 - Alberi Binari

Un albero binario è un albero in cui ogni nodo ha al più due figli, detti *figlio sinistro* e *figlio destro* del nodo. Se presente, ciascun figlio è a sua volta la radice di un albero binario, detto *sottoalbero* (destro o sinistro). Un esempio di albero binario è riportato in Figura 10.1.

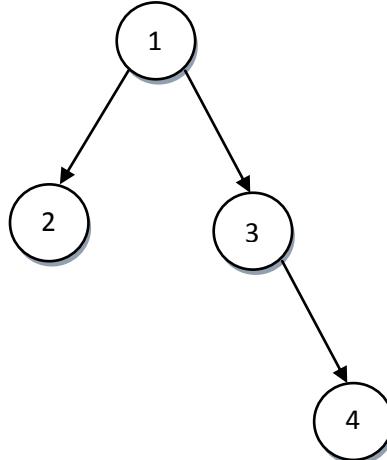


Figura 10.1

I nodi dell'albero sono etichettati con numeri interi. Il nodo etichettato con 1 è la *radice* dell'albero. A tale radice sono associati il *figlio sinistro*, composto dalla *foglia* etichettata con 2, e il *figlio destro*, composto dal nodo etichettato con 3 e con il solo *figlio destro*, la *foglia* etichettata con 4.

L'informazione che etichetta un nodo può essere di qualsiasi tipo: ad esempio, l'albero di Figura 10.1 può essere etichettato con caratteri, come in Figura 10.2.

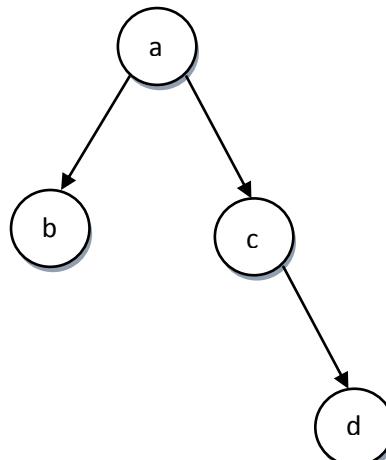


Figura 10.2

Gli alberi binari godono di molte proprietà interessanti, e possono essere utilizzati nella risoluzione di molti problemi di programmazione. In questo contesto, invece, ci interessa notare che la struttura di un albero binario può essere definita ricorsivamente sfruttando le proprietà dell'albero: un albero cioè può essere vuoto, oppure è composto da un nodo e da due sottoalberi. La definizione quindi è:

```
# type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;  
Type btree defined.
```

La variabile di tipo che viene associata indica che la definizione è parametrica al tipo degli elementi che etichettano i nodi.

Il primo albero degli esempi precedenti può essere definito nel seguente modo:

```
# Node(1,Node(2,Empty,Empty),Node(3,Empty,Node(4,Empty,Empty)));;
- : int btreet =
Node (1, Node (2, Empty, Empty), Node (3, Empty, Node (4, Empty,
Empty)))
```

Tramite il pattern-matching è possibile definire operazioni di manipolazione di alberi binari.

La seguente funzione determina se un albero è vuoto:

```
# let is_empty bt = match bt with
    Empty -> true
    | _ -> false;;
is_empty : 'a btreet -> bool = <fun>
```

Le due funzioni seguenti selezionano, rispettivamente, il sottoalbero sinistro e il sottoalbero destro di un albero binario:

```
# let left bt = match bt with
    Empty -> Empty
    | Node (x, l, r) -> l;;
left : 'a btreet -> 'a btreet = <fun>

# let right bt = match bt with
    Empty -> Empty
    | Node (x, l, r) -> r;;
right : 'a btreet -> 'a btreet = <fun>
```

La ricerca di un elemento in un albero binario può essere effettuata molto agevolmente, utilizzando il pattern-matching:

```
# let rec search x bt = match bt with
    Empty -> false
    | Node (y, lt, rt) when x=y -> true
    | Node (y, lt, rt) when x<>y -> (search x lt) or (search x rt);;
search : 'a -> 'a btreet -> bool = <fun>
```

L'albero vuoto non contiene alcun elemento, e quindi la ricerca in un albero vuoto restituisce `false`. Se, al contrario, la ricerca viene effettuata in un albero non vuoto, ci sono due casi da esaminare. Se la radice dall'albero è etichettata dall'elemento che stiamo cercando, la ricerca termina con successo. In caso contrario, la ricerca deve proseguire nei due sottoalberi dell'albero dato.

Si noti che la ricerca permette di esaminare i nodi dell'albero esattamente una volta. Un semplice algoritmo per contare i nodi di un albero è dato dalla seguente definizione:

```

# let rec count bt = match bt with
  Empty -> 0
  | Node (x, lt, rt) -> 1 + (count lt) + (count rt);;
count : 'a btree -> int = <fun>

```

È lasciato per esercizio al lettore dimostrare per induzione strutturale che la funzione `count` restituisce esattamente il numero dei nodi dell'albero.

Affrontiamo ora il problema più generale di effettuare una visita di un albero. La visita di un albero consiste nel seguire una “rotta” di viaggio che consenta di “esaminare” tutti i nodi dell’albero esattamente una volta. La visita di un albero può essere effettuata in accordo a tre diverse politiche:

- esaminando prima la radice, poi il sottoalbero sinistro e quindi quello destro (visita in preordine o visita anticipata o prefissa);
- esaminando prima il sottoalbero sinistro, quindi la radice e infine il sottoalbero destro (visita simmetrica o infissa);
- esaminando prima il sottoalbero sinistro, quindi il sottoalbero destro e infine la radice (visita in postordine o visita posticipata o postfissa).

Ad esempio, nell’albero binario di Figura 10.3 le tre visite portano ad esaminare i nodi in tre ordini diversi:

- la visita anticipata permette di visitare i nodi nell’ordine dato dalla lista [1; 3; 2; 4; 5; 6];
- la visita simmetrica permette di visitare i nodi nell’ordine dato dalla lista [2; 3; 4; 5; 1; 6];
- la visita posticipata permette di visitare i nodi nell’ordine dato dalla lista [2; 5; 4; 3; 6; 1].

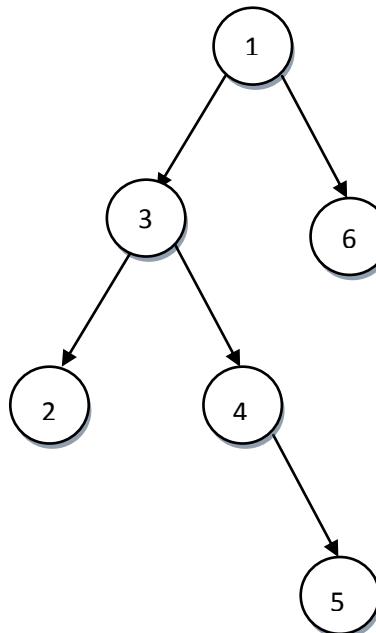


Figura 10.3

Le funzioni che permettono di definire le liste derivate dalle visite anticipata, simmetrica e posticipata, rispettivamente, sono le seguenti:

```

# let rec preorder bt = match bt with
    Empty -> []
  | Node(x, lt, rt) -> x::(preorder lt @ preorder rt);;
preorder : 'a btree -> 'a list = <fun>

# let rec inorder bt = match bt with
    Empty -> []
  | Node(x, lt, rt) -> (inorder lt) @ (x::(inorder rt));;
inorder : 'a btree -> 'a list = <fun>

# let rec postorder bt = match bt with
    Empty -> []
  | Node(x, lt, rt) -> (postorder lt) @ ((postorder rt)@[x]);;
postorder : 'a btree -> 'a list = <fun>

```

10.4 – Alberi Binari di Ricerca

Joe Franksen, uno dei proprietari di un noleggio video, riceve un sacco di lamentele da parte dei clienti riguardo alla ricerca di film per regista: è troppo lenta. Ora ci ha assunti per provare a sistemare il problema. Il problema non sembra stare nella parte di matching del sistema di query, dunque dovremo osservare le procedure usate per cercare un particolare regista o film.

Viene usata una lista di descrizioni di film ed esiste una funzione che cerca quelli di un certo regista. Questa funzione cerca attraverso tutta la lista di film, anche se quelli del regista in questione stanno all'inizio, perché non c'è modo di sapere che non ci sono altri film di quel regista più avanti nella lista. Quando il business di noleggio film di Franksen era solo una piccola parte del suo distributore di benzina, non c'erano problemi perché aveva solamente un centinaio di film. Ora che però il business si è espanso e i film sono diventati 10000, il tempo che serve per trovarne uno diventa estremamente lungo.

Ci sono modi migliori per strutturare la lista di film, in modo che sia veloce cercare quelli con un dato regista? Una idea potrebbe essere quella di ordinare la lista alfabeticamente rispetto al nome del regista. Quando cerchiamo un dato regista, possiamo fermarci non appena raggiungiamo il primo film il cui regista ha un nome “alfabeticamente più grande” di quello che cerchiamo.

Questo approccio è migliore del precedente? Dipende molto dal nome del regista cercato: se questo fosse Alfred Hitchcock, la ricerca sarebbe relativamente veloce (poiché il nome comincia con la lettera “A”). Se invece stessimo cercando film diretti da Woody Allen, dovremmo scorrere quasi l'intera lista per arrivare alla lettera “W”. Possiamo mostrare che, in media, usare una lista ordinata ci porterà ad un tempo pari alla metà di quello che ci metteremmo usando la lista disordinata. Da un punto di vista asintotico, questo non è un miglioramento significativo.

La ricerca sarebbe più veloce se usassimo l'approccio chiamato “*divide et impera*”. L'idea fondamentale consiste dividere a metà la lista ad ogni passo. Cominciamo osservando l'elemento centrale della lista: se l'elemento che stiamo cercando coincide con l'elemento centrale, abbiamo finito. Se invece è più piccolo dell'elemento centrale allora dobbiamo cercare solo nella prima metà della lista; se invece è più grande allora dobbiamo cercare nella seconda metà della lista. Questo metodo di ricerca è spesso chiamato *ricerca binaria*. Poiché ogni passo della ricerca binaria divide lo spazio di ricerca a metà, ci aspettiamo che il tempo richiesto sia alla peggio un multiplo di $\log n$,

dove n è la dimensione della lista (in simboli diciamo che il tempo è $O(\log n)$, pronunciato “o grande di $\log n$ ”, che significa che il tempo sta sotto a un multiplo costante di $\log n$. Per valori grandi di n questo è un miglioramento enorme perché, ad esempio, $\log_2(1000000) \approx 20$, ovvero guadagniamo un fattore di velocità pari a $\frac{1000000}{20} = 50000$.

Il guaio arriva quando cerchiamo di trasformare questo metodo in codice: non sappiamo come arrivare velocemente alla metà della lista. Infatti, il tempo che ci vuole per raggiungere l'elemento centrale è tale per cui la ricerca binaria diventa lenta quanto la tradizionale ricerca lineare. Possiamo fare qualcosa alla nostra lista che sia più drastico di un semplice ordinamento? In altre parole, possiamo arrangiare in qualche modo gli elementi della lista così da poter implementare efficientemente l'algoritmo di ricerca binaria? Avremmo bisogno di poter accedere facilmente all'elemento centrale (ovvero quello rispetto al quale la metà degli elementi rimanenti è più piccola e l'altra metà è più grande). Vorremmo anche poter accedere in modo efficiente agli elementi che sono più piccoli di quello centrale, così come a quelli più grandi. Inoltre, le due metà dovrebbero essere strutturate esattamente nella stessa maniera dell'intera lista di elementi, in modo da poter continuare a cercare allo stesso modo nella metà rilevante.

Come possiamo creare una tale struttura? La risposta sta nell'usare una struttura dati basata sulla descrizione che abbiamo appena fatto. Il nostro nuovo tipo di dato avrà 3 elementi: un elemento (quello centrale) e due collezioni di elementi (quelli più piccoli e quelli più grandi). In questo modo, possiamo accedere a ciascuna delle 3 parti che ci servono semplicemente usando l'apposito selettore.

Questa struttura viene chiamata *albero binario di ricerca*. Nella discussione precedente compare già un indizio della definizione ricorsiva di tale struttura: la maggior parte degli alberi binari ha un elemento centrale e due sottoalberi, che sono anch'essi alberi binari. Cerchiamo di essere più precisi. Prima di tutto, non abbiamo ancora menzionato il caso base: l'albero vuoto. Poi, dobbiamo definire cosa intendiamo per “elemento centrale”. È semplicemente un oggetto più grande di tutti quelli contenuti in uno dei due sottoalberi e più piccolo di tutti quelli contenuti nell'altro sottoalbero. Possiamo allora scrivere la seguente definizione:

ALBERO BINARIO DI RICERCA

Un albero binario di ricerca o è vuoto oppure consiste di 3 parti: la radice, il sottoalbero sinistro e il sottoalbero destro. La radice è un elemento che è maggiore o uguale rispetto a tutti quelli del sottoalbero sinistro e minore o uguale a quelli del sottoalbero destro. I sottoalberi destro e sinistro sono a loro volta alberi binari di ricerca.

Si noti che questa definizione non garantisce che la radice sia l'elemento mediano (ovvero l'elemento rispetto al quale metà degli elementi dell'albero sono più piccoli e metà sono più grandi). Quando la radice dell'albero è il mediano (e lo stesso avviene per i sotto-alberi, per i sotto-sotto-alberi, etc) allora l'albero avrà altezza minima. Vedremo che tali alberi sono gli alberi di ricerca binari più efficienti per la ricerca.

Per semplicità consideriamo ora alberi binari di ricerca i cui elementi sono dei numeri. Assumiamo inoltre che non ci siano elementi duplicati.

Un piccolo albero binario di ricerca con 7 elementi è rappresentato in Figura 10.4:

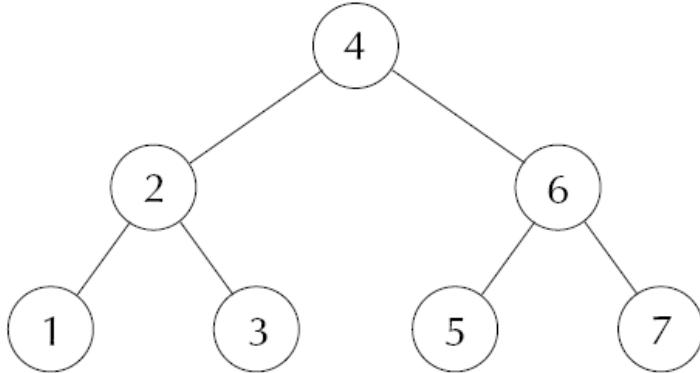


Figura 10.4

Si supponga che le funzioni $\text{left } x \text{ bt}$ e $\text{right } x \text{ bt}$ restituiscano i sottoalberi sinistro e destro del sottoalbero di bt la cui radice è etichettata con x , e si consideri la seguente proprietà di un generico albero binario di ricerca bt :

$$\forall x \in bt. (\forall y \in \text{left } x \text{ bt}. y \leq x) \wedge (\forall y \in \text{right } x \text{ bt}. x \leq y)$$

Gli alberi che godono di questa proprietà sono sempre detti *alberi binari di ricerca*. Su tali alberi la ricerca di un elemento può essere resa più efficiente, sfruttando la proprietà vista:

```

# let rec search x bt = match bt with
| Empty -> false
| Node(y, lt, rt) when x=y -> true
| Node(y, lt, rt) when x<y -> (search x lt)
| Node(y, lt, rt) when x>y -> (search x rt);;
search : 'a -> 'a btree -> bool = <fun>
  
```

In pratica, possiamo evitare di visitare tutto l'albero: se il nodo y che stiamo esaminando è maggiore dell'elemento x cercato, allora dalla proprietà $(\forall z \in \text{right } y \text{ bt}. y \leq z)$ possiamo dedurre che $(\forall z \in \text{right } y \text{ bt}. x < z)$, e quindi possiamo evitare di effettuare la ricerca nel sottoalbero destro del nodo y . Analogamente, se il nodo y che stiamo esaminando è minore dell'elemento x cercato, allora dalla proprietà $(\forall z \in \text{left } y \text{ bt}. z \leq y)$ possiamo dedurre che $(\forall z \in \text{left } y \text{ bt}. x > z)$, e quindi possiamo evitare di effettuare la ricerca nel sottoalbero sinistro del nodo y .

Vediamo ora come garantire il mantenimento di queste proprietà in un albero. Proviamo, ad esempio, a costruire un albero binario di ricerca a partire da una lista. Si noti che per ogni insieme di elementi in generale non esiste un unico albero binario di ricerca associato: ad esempio, alla lista [1; 5; 3; 2; 4], è possibile associare gli alberi binari di ricerca di Figura 10.5 e Figura 10.6.

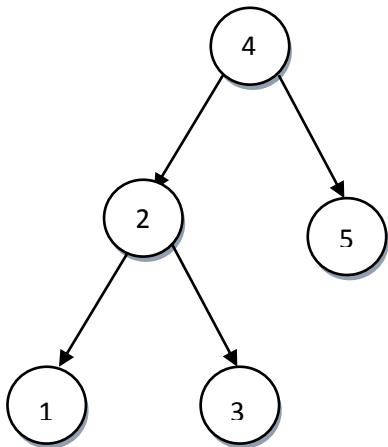


Figura 10.5

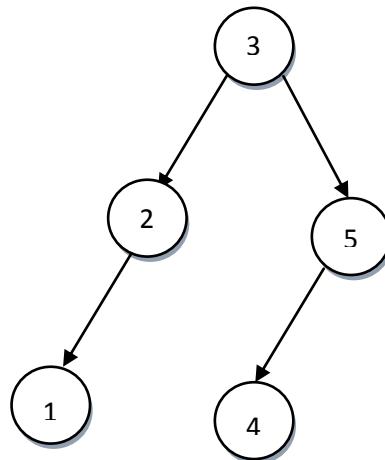


Figura 10.6

La tecnica che utilizzeremo, comunque, consiste nel cominciare a costruire l'albero ricorsivamente partendo da un elemento, chiamato perno, e partizionando gli altri elementi nell'albero in base al valore del perno. Ad esempio, se utilizziamo come perno il valore 4, allora gli elementi 1, 2 e 3 andranno a finire nel sottoalbero sinistro dell'albero con radice 4, mentre l'elemento 5 andrà a finire nel sottoalbero destro. Riapplicando lo stesso principio ai sottoalberi, otteniamo alla fine il primo dei due alberi visti sopra. Si noti che la costruzione dell'albero dipende dalla scelta del perno: il secondo albero, per esempio, è costruito scegliendo come perno l'elemento centrale della lista.

La funzione che trasforma una lista in un albero binario di ricerca, utilizzando come perno l'ultimo elemento della lista, è la seguente:

```

# let rec buildtree l =
let rec insord x bt =
  match bt with
    Empty -> Node (x, Empty, Empty)
  | Node (y, lt, rt) when x <= y -> Node (y, insord x lt, rt)
  | Node (y, lt, rt) when x > y -> Node (y, lt, insord x rt)
in match l with
  [] -> Empty
  | x :: xs -> insord x (buildtree xs);;
buildtree: 'a list -> 'a btree = <fun>

```

ESERCIZIO 10.1:

Scrivere la funzione `minimo` che restituisce l'elemento più piccolo in un albero binario di ricerca non vuoto (contenente numeri interi).

ESERCIZIO 10.2:

Scrivere una funzione che dati un albero binario di ricerca di numeri, un limite inferiore, e un limite superiore, conta quanti elementi dell'albero sono \geq del limite inferiore e quanti sono \leq del limite superiore (quindi gli elementi contenuti nell'intervallo definito dai due valori di input). Assumete che l'albero possa contenere elementi duplicati.

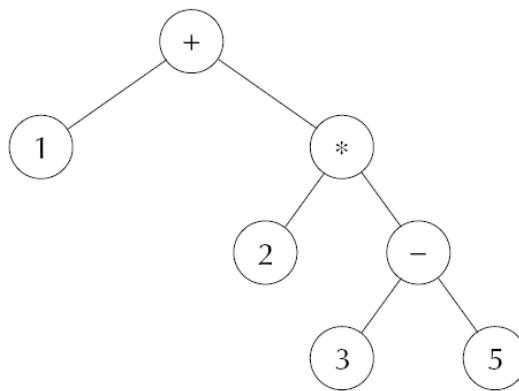
10.5 – Alberi di espressioni

Finora abbiamo usato alberi binari e alberi binari di ricerca come strumento per memorizzare collezioni di numeri o dati. Questi alberi sono diversi dalle liste per il modo in cui accediamo ai loro elementi. Una lista ha un elemento speciale, il primo elemento, e tutti gli altri elementi sono accoppiati in un'altra lista. Anche gli alberi binari hanno un elemento speciale, la radice, ma poi dividono i loro elementi rimanenti in due sottoalberi (invece che uno solo), il che produce una struttura gerarchica utile in varie circostanze. In questa sezione vediamo un altro tipo di albero che usa la struttura gerarchica per rappresentare espressioni aritmetiche. In questo caso, il modo in cui un albero è strutturato indica gli operandi per ciascuna operazione nell'espressione.

Consideriamo una espressione aritmetica, come ad esempio:

(1 + (2 * (3 - 5)))

Possiamo pensare a questa espressione come a una struttura ad albero in cui i numeri sono le foglie e gli operatori sono i nodi interni:



Tale struttura viene spesso chiamata *albero di espressioni*. Come abbiamo fatto per gli alberi binari di ricerca, possiamo definire un albero di espressioni più precisamente:

ALBERO DI ESPRESSIONI

Un albero di espressioni è o un numero oppure ha 3 parti, il nome di un operatore, un operando sinistro e un operando destro. Gli operandi, destro e sinistro, sono anch'essi alberi di espressioni.

Ci sono varie cose da notare in questa definizione:

- Ci stiamo restringendo ad espressioni che contengono operatori binari (cioè operatori che prendono esattamente due operandi).
- Ci stiamo restringendo ad avere numeri come espressioni atomiche. In generale, gli alberi di espressioni includono anche altri tipi di costanti e nomi di variabili.

Come implementiamo gli alberi di espressioni? Lo faremo come abbiamo fatto per gli alberi binari:

```
# type operator = Sum | Sub | Mul | Div;;  
Type operator defined.
```

```
# type expr_tree = Value of int |
                     Node of expr_tree * operator * expr_tree;;
Type expr_tree defined.
```

I costruttori sono `Value(x)` e `Node(left_op, operator, right_op)`. I selettori devono essere ancora definiti e sono:

```
# let is_constant tree =
  match tree with
  | Value(_)  -> true
  | _             -> false;;
is_constant : expr_tree -> bool = <fun>

# let operator expr_tree =
  match expr_tree with
  | Node(l,o,r) -> o;;
operator : expr_tree -> operator = <fun>

# let left_operand expr_tree =
  match expr_tree with
  | Node(l,o,r) -> l;;
left_operand : expr_tree -> expr_tree = <fun>

# let right_operand expr_tree =
  match expr_tree with
  | Node(l,o,r) -> r;;
right_operand : expr_tree -> expr_tree = <fun>
```

Ora che abbiamo un modo per creare espressioni, possiamo scrivere le funzioni necessarie per valutarle; sfrutteremo la definizione di albero di espressioni per decidere come strutturare il nostro codice. Per avere più flessibilità, useremo una funzione chiamata `compute_value` per trasformare un nome di operatore e due operandi nel risultato della corrispondente operazione. La procedura principale di valutazione deve solamente usare `compute_value` per trasformare le operazioni in risultati:

```
# let value tree =
  match tree with
  | Value(x)  -> x;;
value : expr_tree -> int = <fun>

# let compute_value operator operand1 operand2 =
  match operator with
  | Sum -> operand1 + operand2
  | Sub -> operand1 - operand2
  | Mul -> operand1 * operand2
  | Div -> operand1 / operand2;;
compute_value : operator -> int -> int -> int = <fun>

# let rec evaluate expr_tree =
  if is_constant expr_tree then value expr_tree
  else
    let left_value = evaluate (left_operand expr_tree)
    and
    right_value = evaluate (right_operand expr_tree)
```

```
in compute_value (operator expr_tree) left_value right_value;;
evaluate : expr_tree -> int = <fun>
```

Con queste definizioni, abbiamo

```
# let expr_tree = Node(Value(1), Sum, Node(Value(2), Mul,
Node(Value(3), Sub, Value(5))));;
expr_tree : expr_tree = Node (Value 1, Sum, Node (Value 2, Mul, Node
(Value 3, Sub, Value 5)))  
  
# evaluate expr_tree;;
- : int = -3
```

ESERCIZIO 10.3:

Scrivere una funzione che conta quante operazioni sono contenute in una espressione.

Notiamo che tutti gli operatori delle nostre espressioni sono operatori binari, e quindi abbiamo bisogno di nodi con due figli per rappresentarli; in questo caso diciamo che i nodi operatore hanno tutti grado 2. Se avessimo avuto operatori che prendono m argomenti invece di solamente 2, avremmo avuto bisogno di nodi con grado m (cioè alberi con m sottoalberi).

Il tipo di albero che abbiamo usato in questa sezione differisce leggermente dagli alberi binari che abbiamo visto precedentemente nel capitolo. In quei nodi posizionali era possibile avere un nodo con un figlio destro ma senza figlio sinistro, ad esempio. Negli alberi ordinati che usiamo per le espressioni, d'altro canto, non può esserci un secondo operando a meno che non ci sia anche il primo.