Note di Programmazione Funzionale

Appunti ad uso degli studenti di Programmazione
Corsi di Studio in Informatica
Università degli Studi di Pisa

Anno accademico 1998/99

Giuseppe Manco

Indice

Convenzioni

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 Caml. Nel seguito faremo riferimento al sistema CAML LIGHT, release 0.73. Tutti i programmi utilizzati in questa dispensa sono stati testati utilizzando tale sistema. È possibile anche utilizzare i programmi con la release 1.05 del sistema (facendo però attenzione alla sintassi utilizzata: le due versioni utilizzano una sintassi differente per il linguaggio. Nei vari capitoli delineeremo alcune significative differenze sintattiche).

Nel corso dei vari capitoli seguiremo la seguente notazione: denoteremo ogni programma Caml (e ogni interazione col sistema) in testo dattiloscitto, mentre denoteremo funzioni ed espressioni matematiche (relative, per esempio, alla definizione Caml) in italico. Ad esempio, `succ x` denota l'espressione Caml relativa alla funzione successore, mentre $succ\ x$ denota l'espressione matematica che deriva dalla definizione Caml.

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:

$$
\begin{array}{ll}
 & a \times (b + c + 0) \\
= & \quad \{\ 0 \text{ è l'elemento neutro della somma } \} \\
 & a \times (b + c) \\
= & \quad \{\ \text{proprietà distributiva della moltiplicazione sulla somma } \} \\
 & a \times b + a \times c
\end{array}
$$

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.

Molte delle funzioni descritte in queste dispense non sono disponibili automaticamente nel sistema Caml, ma devono essere caricate le relative librerie.

Gran parte del materiale presente su questa dispensa è tratta dal testo "Introduction to Functional Programming", di R. Bird e P. Wadler. In esso vengono trattati estensivamente molti degli argomenti introdotti in questa dispensa: il lettore interessato può trovarvi approfondimenti agli argomenti trattati.

Il sistema Caml e la documentazione relativa può essere reperito all'indirizzo Web
`http://pauillac.inria.fr/caml`.

Capitolo 1

Concetti Fondamentali

1.1 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 effettuata per mezzo dell'*applicazione*. 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 funzionale 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.

1.1.1 Una sessione al Terminale

Per esemplificare alcuni dei concetti sopra esposti, proveremo a descrivere una sessione al terminale, e ad interagire col sistema Caml. Immaginiamo quindi 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.

Un'espressione tipica che potremmo digitare è un numero:

```
# 42;;
- : int = 42
```

Qui il calcolatore ha risposto valutando l'espressione digitata, e quindi fornendoci delle informazioni sull'espressione. Poiché tale espressione era un numero intero, il sistema ha risposto semplicemente mostrando in output l'espressione ed il suo tipo. Il numero decimale `42` è un'espressione nella sua più semplice forma, e non è possibile applicare un ulteriore processo di valutazione.

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.

Abbiamo detto che l'aspetto interessante della programmazione funzionale è la capacità di costruire definizioni. Vediamo qualche semplice esempio: supponiamo di voler definire la funzione che associa ad un numero il suo quadrato, e la funzione *min*, che associa il minimo tra due numeri interi. Nella notazione matematica, tali funzioni sono espresse come segue:

$$\begin{aligned} square(x) &= x^2 \\ \min(x, y) &= \begin{cases} x & \text{se } x \leq y \\ y & \text{se } x > y \end{cases} \end{aligned}$$

Per permettere l'introduzione di tali definizioni nel sistema, faremo uso dello speciale costrutto `let`, che permette la definizione di identificatori:

```
# let square x = x * x;;
square : int -> int = <fun>
# let min (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`). Si noti inoltre che il sistema segnala che l'espressione valutata denota una funzione, tramite il simbolo `<fun>`.

La definizione delle funzioni `square` e `min` permette di fare delle interrogazioni più complesse di quelle viste precedentemente:

```
# square(3+4);;
- : int = 49
# min 3 4
- : int = 3
# min (square 5) (4+9);;
- : int = 13
```

In effetti, 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. Si noti anche che il sistema ha risposto in maniera differente rispetto al caso delle semplici espressioni: la riscrittura dell'identificatore `square` sta ad indicare che `square` è un identificatore, appunto, e che il suo valore è indicato nel seguito.

Alcune espressioni possono essere valutate senza il bisogno di un contesto. Ad esempio, le prime due espressioni che abbiamo visto non hanno bisogno di un contesto, poiché si basano su operazioni assunte come primitive (le regole di semplificazione sono definite nel valutatore stesso). In ogni momento, comunque, il programmatore può aggiungere o modificare definizioni al contesto attuale. Per esempio, supponiamo di voler aggiungere al contesto attuale, comprendente le definizioni di `square` e `min` viste sopra, le seguenti definizioni:

```
# let lato = 12;;
lato : int = 12
# let area = square lato;;
area : int = 144
```

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

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

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

1. è possibile interagire col sistema sottoponendo 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.

Esercizi

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

2 Si definisca una funzione `max` che calcola il più grande tra due numeri.

1.2 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[1]. Inoltre, il significato di un'espressione dipende soltanto dai significati delle sue sottoespressioni.

[1] Questo aspetto distingue i linguaggi funzionali dai linguaggi relativi ad altri paradigmi. Ad esempio, i costrutti di un linguaggio imperativo hanno l'effetto di modificare lo stato di esecuzione del programma stesso.

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.

1.2.1 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 venga effettuata, considerando la valutazione dell'espressione `square (3+4)`, vista precedentemente. Nel seguito il simbolo $\rightsquigarrow$ verrà utilizzato col significato *'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 appena vista 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. La più semplice forma equivalente ad un'espressione, qualunque essa sia, non è un valore ma una rappresentazione di esso. 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 infinite rappresentazioni. I calcolatori di

solito operano con la rappresentazione binaria dei numeri, nella quale il numero 'quarantanove' può essere rappresentato dalla stringa "0000000000110001" di 16 bit.

Diremo che un'espressione è *canonica* (o in *forma normale*) se non può essere ulteriormente ridotta. Un valore è 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 nel senso matematico. 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 d'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 d'errore o un perpetuo silenzio.

Esercizi

1 Si dica quante differenti sequenze di riduzioni possono essere applicate all'espressione

```
square (square (3+7))
```

per ridurla in forma normale.

2 Si consideri la definizione

```
let three x = 3
```

In quanti modi l'espressione `three (3+4)` può essere ridotta ad una forma normale.

1.3 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` nel sistema 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, la coppia `int * bool`, che ha il primo componente intero ed il secondo booleano); le funzioni (ad esempio le funzioni `int -> int` 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 punto)).

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.

1.4 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 $f : A \mapsto B$ o, nella notazione Caml, `f : A -> 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 un'espressione. Per esempio, esse possono essere passate come argomenti ad altre funzioni e/o restituite come risultato. Per esempio, potremmo scrivere:

```
# let g f x = f(x+1) - 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+1)`.

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 x y = x + y
```

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 tra quel numero e tre:

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

Le due definizioni descrivono differenti procedure per ottenere la corrispondenza, ma `double` e `double1` rappresentano la stessa funzione[2]. 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.

1.4.1 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 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 ed 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 x = x;;
id : 'a -> 'a = <fun>
```

Qui `'a` denota una variabile di tipo, e sta ad indicare che tale variabile può essere istanziata 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 esere sostituito a `'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:

[2]Un importante risultato della teoria della computabilità, comunque, ci dice che in generale non è possibile stabilire se due definizioni rappresentano la stessa funzione.

```
# 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 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, 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, se vogliamo definire una istanza della funzione `id` sulle coppie, possiamo fare uso della seguente definizione:

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

o, alternativamente, definire il tipo degli argomenti:

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

1.4.2 Definizioni

Abbiamo già visto in qualche esempio come sia possibile definire una funzione per casi. Ad esempio, la funzione `min` che calcola il minimo tra due elementi di un insieme `'a` su cui è definito un ordine, può essere definita nel seguente modo:

```
#let min 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, 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*.

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

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

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

Le definizioni locali possono anche riguardare funzioni:

```
#let foo x = let f y = x*y
             in f x;;
foo : int -> int = <fun>
#foo 3;;
- : int = 9
# f 3;;
Toplevel input:
>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.

Esercizi

1 Si diano esempi di funzioni con i seguenti tipi:

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

2 Si definisca il tipo appropriato per descrivere la funzione di composizione di due funzioni.

3 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.

4 Si inferiscano i tipi delle seguenti definizioni:

```
let one x = 1

let apply f x = f x
```

5 Si valutino le seguenti espressioni, riducendole in forma canonica:

```
let x =1+2 in let foo y = y+x in foo x

let x = 1+2 in let foo x = x+x in foo x

let f1 f2 x = f2 x
    in let g x = x+1
           in f1 g 2
```

1.5 Specifica e Implementazione

In programmazione, per specifica si intende una descrizione matematica dei compiti che che un programma è chiamato a svolgere, mentre per implementazione si intende la realizzazione di un programma che soddisfi la specifica data. Cè 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 coincisa e chiara il più possibile. L'implementazione, per contro, è una serie di istruzioni eseguibili da un calcolatore, ed 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 per 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*:

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

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

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

$$\begin{array}{ll} & increase\ x \\ = & \quad \{\ \text{definizione di } \texttt{increase}\ \} \\ & square(x+1) \\ = & \quad \{\ \text{definizione di } \texttt{square}\ \} \\ & (x+1) \times (x+1) \\ = & \quad \{\ \text{proprietà algebriche}\ \} \\ & x \times x + 2 \times x + 1 \\ > & \quad \{\ \text{assunzione: } x \geq 0\ \} \\ & x \times x \\ = & \quad \{\ \text{definizione di } \texttt{square}\ \} \\ & square\ x \end{array}$$

Qui noi abbiamo inventato una definizione per *increase*, e 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:

$$\forall x \geq 0.\ (square\ (x+1) > square\ x)$$

D'altra parte,

$$\forall x \geq 0.\ (square\ (x+1) > square\ x \wedge increase\ x = square\ (x+1) \Longrightarrow increase\ x > square\ x)$$

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

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.

Esercizi

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

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

```
let isSquare 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 reali, 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 reali (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[1], ed in particolare i numeri reali non possono essere rappresentati con precisione (occorre cioè ricorrere ad una approssimazione finita del numero).

Useremo le operazioni descritte nella tabella 2.1, definite su `int` (i corrispettivi operatori per `float` si ottengono aggiungendo `.` al simbolo). 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). Poiché la rappresentazione di un numero può non essere esatta, operazioni sui numeri reali 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 artimetici:

```
#2+3*4;;
```

[1] il sistema Caml utilizza 31 bits per la rappresentazione dei numeri interi, e 64 per la rappresentazione dei numeri reali.

`+`	addizione intera
`*`	moltiplicazione intera
`/`	divisione intera
`mod`	modulo intero
`**`	esponenziale reale

Tabella 2.1: Operazioni aritmetiche

```
- : 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 e 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 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 della tabella 2.1 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 di composizione funzionale "`->`", che opera su 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. Il motivo fondamentale di questa scelta consiste nel fatto che nel contesto della programmazione funzionale i valori funzione sono "cittadini di prima classe", essendo ammesso un costruttore di tipo che, alla pari di altri costruttori, permette di ottenere tipi funzione. 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[2]. 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:

[2]in alcuni linguaggi funzionali, l'utilizzo del primo tipo di definizione può portare ad una corretta valutazione rispetto al secondo tipo di valutazione.

```
#let curry f x y = f (x,y);;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
#curry foo2;;
- : 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>
```

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. È comunque possibile considerare tali operatori come operatori prefissi, utilizzando la funzione `prefix`[3]. 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 reale,

```
#let inv = ( prefix /.) 1.0;;
inv : float -> float = <fun>
#inv 4.0;;
- : float = 0.25
```

Esercizi

1 Per quali argomenti le seguenti funzioni ritornano `true`?

```
(prefix =) 9 (prefix +) 2 (prefix  *) 7
(prefix >) 3 (prefix mod) 8
```

[3]Nella release 1.05 del sistema, la parola chiave `prefix` non viene utilizzata. Un operatore infisso racchiuso tra parentesi tonde è considerato come prefisso.

`=`	uguaglianza
`<>`	disuguaglianza
`< (<=)`	minore (minore o uguale)
`> (>=)`	maggiore (maggiore o uguale)
`not`	negazione
`&`	congiunzione
`or`	disgiunzione

Tabella 2.2: Operatori logici e di confronto

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
```

La tabella 2.2 elenca alcuni tra gli operatori tipici.

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, come di consueto[4]:

```
#(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 & false;;
- : 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.

[4]Nella release 0.73 gli operatori & e `or` non possono essere utilizzati come prefissi.

```
let bisestile y = (y mod 4 = 0) & (y mod 100 <> 0 or y mod 400 = 0)
```

Esercizi

1 Si definisca la funzione *sumsqrs* che prende come argomenti tre interi e restituisce la somma dei quadrati dei due più grandi.

2 Si definisca la funzione *prime* che prende come argomenti due interi e stabilisce se i due interi sono primi tra loro.

2.3 Caratteri e Stringhe

I simboli che normalmente ci 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 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 `'7'` è sostanzialmente differente dal numero 7: 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.

Due funzioni primitive sono associate al tipo `char`, entrambe relative alla codifica ASCII dei caratteri:

```
#code;;
- : char -> int = <fun>
# chr;;
- : int -> char = <fun>
```

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

```
#code 'b';;
- : int = 98
# chr 98;;
- : char = 'b'
chr (code 'b' +1);;
- : char = 'c'
code '\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 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 delle funzioni per rendere maiuscolo/minuscolo un carattere:

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

Sfruttando il fatto che i caratteri in maiuscolo vengono immediatamente dopo i caratteri in minuscolo.

2.3.1 Stringhe

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[5] di caratteri che in questo particolare caso contiene un solo elemento.

Vediamo alcuni esempi di stringhe:

[5] Una sequenza ordinata per posizione. Le liste saranno oggetto di studio del prossimo capitolo.

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

Esercizi

1 Si definisca la funzione `nextlet` che prende come argomento una lettera dell'alfabeto e restituisce la lettera successiva (si assuma che le lettere siano chiuse circolarmente).

2 Si definisca la funzione `digitval` che converte un carattere di tipo numerico nel corrispondente valore numerico. Si estenda la funzione ad una stringa.

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

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

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 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 $(0, 1)$.

Possiamo ridefinire tutte le nozioni viste sinora con la nostra notazione standard. Diciamo che una frazione è in forma canonica se è ridotta ai minimi termini; due frazioni sono uguali se hanno la stessa rappresentazione canonica.

Definiamo quindi l'algebra delle frazioni, definendo somma, differenza, moltiplicazione e divisione di frazioni, assicurando che i risultati siano in forma canonica:

```
#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 veda il capitolo 4 per la sua definizione). 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) =
   let (x1, y1) = norm(x,y)
   and
       (u1, v1) = norm(u,v)
   in
     op (x1 * v1) (y1 * u1);;
compare : (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>
```

e così via.

Esercizi

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

2 Si definisca la funzione *split* $x = (y, z)$, dove $abs(y) \leq 5$ 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 cioè 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 secondo 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;;
```

Nella prima definizione, i valori che l'identificatore `p` della seconda definizione può assumere vengono 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 quest'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 specificare tale Possibile pattern, 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 dal 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 ("", 382, 454)
#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 ("", 670, 723)
```

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 delle nozioni 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.

Esercizi

1 Si definiscano le funzioni `and` e `or` usando patterns per il secondo argomento. Si ridefiniscano le funzioni utilizzando patterns per entrambi gli argomenti.

2 Si definisca la funzione `permute` che prende in ingresso una coppia di interi compresi tra 1 e 10 e restituisce il secondo dei due se il primo è 5, o il primo stesso nel caso contrario. Utilizzare patterns su entrambi gli argomenti e sul secondo argomento. Generalizzare la funzione al caso di coppie generali di interi (non solo compresi tra 1 e 10).

2.6 Funzioni

Come abbiamo già visto nei numerosi esempi dei 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. In realtà, ogni definizione di funzione con più di un argomento è stata definita come una funzione higher-order (in alternativa alla definizione di una funzione tramite ennuple). Vediamo altri esempi di funzioni higher-order tratte dalla letteratura matematica.

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>
```

Tale funzione 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.

Funzioni inverse

Supponiamo che $f : A \mapsto B$ 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 $f^{-1} : B \mapsto A$, 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
```

Esercizi

1 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?

$$\begin{aligned} h &= compose\ f\ g \\ h\ x &= compose\ f\ (g\ x) \\ h\ x\ y &= (compose f\ g\ )x\ y \end{aligned}$$

2 Si consideri la funzione

```
let dimezza x = x / 2
```

Si definisca la funzione $dimezza^{-1}$.

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:

$$f\ :\ \alpha \mapsto \beta \tag{2.1}$$

$$g\ x\ :\ \alpha \tag{2.2}$$

Per qualche coppia di tipi α e β. 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:

$$g\ :\ \gamma \mapsto \alpha \tag{2.3}$$

$$x\ :\ \gamma \tag{2.4}$$

In questo caso, il codominio di $g\ x$ deve essere α, e il dominio deve essere identico a quello di x. Supponiamo di istanziare 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, le equazioni 2.4 e 2.3 ci dicono che `square 2 : int`, e le equazioni 2.2 e 2.1 ci dicono che `min square 2 : int -> int`. Nell'effettuare l'analisi, abbiamo applicato le seguenti regole:

Applicazione Se $f\ x\ :\ \alpha$, allora $x\ :\ \beta$ e $f\ :\ \beta \mapsto \alpha$ per qualche β.

Uguaglianza Se per la stessa variabile x si deduce che $x\ :\ \beta$ e $x\ :\ \alpha$, allora $\alpha = \beta$.

Funzione Se $\alpha \mapsto \beta = \alpha' \mapsto \beta'$, allora $\alpha = \alpha'$ e $\beta = \beta'$.

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 `min id : t`, allora `min : t' -> t` e `id : t'`. D'altra parte, sappiamo che `id : 'a -> 'a`, e che `min : 'b -> 'b -'> 'b`. Dalla regola dell'uguaglianza, otteniamo che `'b -> 'b -> 'b = t' -> t` e `'a -> 'a = t'`. Ancora, dalla regola della funzione otteniamo `t = 'b -> 'b` e `'b = t'`. Mettendo tutto insieme, otteniamo `min id : 'a  -> 'a -> 'a`.

Esercizi

1 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.

2 Quali sono i tipi delle seguenti espressioni?

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

Capitolo 3

Liste

In questo e nei prossimi capitoli tratteremo le liste, il loro uso e le loro operazioni. Una lista è una collezione di elementi omogenei; essa 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). Le operazioni che possono essere definite sulle liste sono numerose. In questo capitolo ne vedremo alcune tra le più comuni e rilevanti.

3.1 Notazione

Una *lista* (o *sequenza*) è una collezione ordinata di valori; in una lista, cioè è possibile distinguere il primo elemento, l'ultimo e così via. Gli elementi di una lista devono avere tutti lo stesso tipo: si possono così definire liste di numeri, liste di caratteri, o liste di liste (di valori dello stesso tipo). Una lista finita è denotata utilizzando le parentesi quadre e il simbolo "`;`" come separatore degli elementi. Vediamo alcuni esempi di liste:

```
#[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. Le stringhe, che abbiamo introdotto nel precedente capitolo, 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 primo 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 di operatori di confronto; il confronto viene fatto con riferimento al primo elemento della lista.

```
#[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
```

Esercizi

1 Si dia un esempio di espressione avente due occorrenze della lista vuota: la prima occorrenza deve avere tipo `int list`, e la seconda `char list`.

3.2 Operazioni su liste

Introduciamo alcune funzioni sulle liste. Molte di queste funzioni sono definite per patterns, facendo uso della funzione ausiliaria `::`, definita nel seguente modo:

```
#(prefix ::) ;;
- : 'a * 'a list -> 'a list = <fun>
#1::[];;
- : int list = [1]
#1::[2;3];;
- : int list = [1; 2; 3]
```

Tale funzione (che chiameremo *cons*) prende come argomenti un valore ed una lista di valori dello stesso tipo, e restituisce una lista ottenuta inserendo il valore come primo elemento della lista. Per convenzione, l'operatore `::` associa a destra, per cui `1::2::[3;4]` vuol dire `1::(2::[3;4])`. Ogni lista può essere costruita inserendo i suoi elementi ad uno ad uno nella lista vuota. Possiamo infatti fare riferimento ad una lista $[x_1; x_2; \ldots; x_n]$ come ad una abbreviazione per $x_1 :: x_2 :: \ldots :: x_n : [\,]$.

Nel capitolo precedente abbiamo visto come definire una funzione per mezzo di pattern matching. Tramite il `::`, 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 single xs = match xs with
                      []       -> false
```

```
                    | [x]       -> true
                    |(x::y::xs) -> false;;
single : 'a list -> bool = <fun>
```

Si noti che qui il pattern `[x]` è usato per indicare `x::[]`.

Concatenazione. 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, $[1;2] \mathbin{@} [3] = [1] \mathbin{@} [2;3]$).

È possibile anche definire la generalizzazione `concat` della funzione di concatenazione, che prende come argomento una lista di liste e restituisce la lista risultante dalla concatenazione delle liste:

```
#concat;;
- : 'a list list -> 'a list = <fun>
#concat [[1;2];[3;4];[5;6]]
- : int list = [1;2;3;4;5;6]
```

Si noti che è possibile definire l'operatore `::` per mezzo di `@`:

$$x :: xs = [x] \mathbin{@} xs$$

Lunghezza di una lista. È possibile definire la funzione `length` che, data una lista finita, restituisce la lunghezza della lista stessa:

```
#length;;
- : 'a list -> int = <fun>
#length [1;2;3;5;7];;
- : int = 5
```

Anche in questo caso la funzione è polimorfa, essendo irrilevante il tipo degli elementi della lista. Si noti che vale

$$length(xs \mathbin{@} ys) = length\ xs\ +\ length\ ys$$

Head e Tail. La funzione `hd` seleziona il primo elemento di una lista, e `tl` seleziona la rimanente porzione.

```
#hd;;
- : 'a list -> 'a = <fun>
#tl;;
- : 'a list -> 'a list = <fun>
#tl [];;
Uncaught exception: Failure "tl"
#hd [];;
Uncaught exception: Failure "hd"
#hd [1;2;3];;
- : int = 1
#tl [1;2;3 ];;
- : int list = [2; 3]
```

Come si può vedere le funzioni 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 xs = match xs with x::xs -> x;;

let tl  xs = match xs with x::xs -> xs;;
```

Selezione su liste. È possibile selezionare l'elemento ennesimo di una lista, tramite l'operatore `nth`:

```
#nth;;
- : int -> 'a list -> 'a = <fun>
#nth 2 [1;2;3];;
- : int = 2
```

Un'altra funzione che è possibile definire è `take`, che prende come argomenti un intero n ed una lista e seleziona i primi n elementi della lista:

```
#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]
```

Si noti che la funzione `take` è una generalizzazione della funzione `hd`. È possibile anche definire una funzione `drop`, come generalizzazione della funzione `tl`:

```
#drop;;
- : int -> 'a list -> 'a list = <fun>
#drop 3 [1;2;3;4];;
- : int list = [4]
```

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 `for_all`, `exists` e `mem`. `for_all` (resp. `exists`) prende come argomenti un predicato ed una lista e verifica che tutti gli elementi (resp. almeno un elemento) della lista soddisfino il predicato. La funzione `mem` prende come argomenti un elemento ed una lista e verifica che l'argomento sia presente nella lista.

```
#let test x = x>1
 in for_all test [2;3;4];;
- : bool = true
#let test x = x>1
 in exists test [1;0;-2];;
- : bool = false
#mem 1 [1;2;3];;
- : bool = true
```

Manipolazione di 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 prende come argomento una coppia di liste e restituisce una lista di coppie dei corrispondenti elementi. Tale funzione ammette la funzione inversa, `split`:

```
#split;;
- : ('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`]
```

Vedremo come l'uso di queste funzioni combinato con funzioni higher-order permetta la definizione di un notevole numero di funzioni.

Esercizi

1 In quanti modi una lista $[1; 2; \ldots; n]$ può essere espressa come concatenazione di due liste?

2 Quali delle seguenti equazioni sono vere?

$$\begin{aligned} [\,] :: xs &= xs \\ [\,] :: xs &= [[]; xs] \\ xs :: [\,] &= xs \\ xs :: [\,] &= [xs] \\ x :: y &= [x; y] \\ (x :: xs)@ys &= x :: (xs@ys) \end{aligned}$$

3 Usando pattern matching con `::`, si definisca la funzione `rev2` che inverte una lista se e solo se essa ha lunghezza due, altrimenti la lascia invariata.

4 Si definiscano le funzioni `init` e `last` che soddisfano le seguenti relazioni:

$$init(xs \mathbin{@} [x]) = xs$$

$$last(xs \mathbin{@} [x]) = x$$

(suggerimento: si utilizzino le funzioni `rev`, `hd` e `tl`).

5 Usando `combine`, si definisca la funzione `combine4` che converte una quadrupla di liste in una lista di quadruple.

3.3 Map e filter

Introduciamo in questa sezione due utili funzioni higher-order: `map` e `filter`.

La funzione `map` ha come argomento una funzione che applica ad ogni elemento di una lista:

```
#map;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
#let f n = n*2+1
 in map f [1;2;3;4];;
- : int list = [3; 5; 7; 9]
```

La funzione `filter` filtra da una lista tutti gli elementi che soddisfano un dato predicato:

```
#filter;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
#let even n = n mod 2 = 0
 in filter even [1;2;3;4];;
- : int list = [2; 4]
```

Combinando queste due funzioni è possibile esprimerne molte altre. Supponiamo, ad esempio, di voler esprimere una funzione che traduca una lista di elementi in un'altra lista che ha la stessa lunghezza della lista data, ma i cui elementi sono tutti uguali ad una costante (ad esempio 1). Tale funzione è ottenuta utilizzando `map`:

```
# let const x = 1
  in map const [1;2;3;4;5;6];;
- : int list = [1; 1; 1; 1; 1; 1]
```

Ancora, volendo esprimere la lista i cui elementi sono il quadrato degli elementi pari di una lista data,

```
#let even x = x mod 2 = 0
 in map square (filter even [1;2;3;4;5;6]);;
- : int list = [4; 16; 36]
```

Uno dei vantaggi nell'uso delle funzioni higher-order 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:

$$\begin{aligned} map\ (compose\ f\ g) &= compose\ (map\ f)\ (map\ g) \\ map\ f\ (xs \mathbin{@} ys) &= (map\ f\ xs) \mathbin{@} (map\ f\ ys) \\ compose\ (map\ f)\ concat &= compose\ concat\ map\ (map\ f) \end{aligned}$$

La prima legge dice che se applichiamo prima g ad una lista, e poi applichiamo f al risultato, otteniamo lo stesso risultato applicando *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 *map* f ad ogni lista componente e quindi concatenare il risultato. Risultati simili possono essere espressi anche per `filter`:

$$\begin{aligned} filter\ p\ (xs \mathbin{@} ys) &= filter\ p\ xs \mathbin{@} filter\ p\ ys \\ compose\ (filter\ p)\ concat &= compose\ concat\ map(filter\ p) \\ compose\ (filter\ p)\ (filter\ q) &= compose\ (filter\ q)\ (filter\ p) \end{aligned}$$

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".

Esercizi

1 Una definizione alternativa di `filter` può essere data utilizzando `concat` e `map`:

```
let filter p = let box x = ...
               in compose concat (map box)
```

Si completi la definizione.

2 Che tipo ha `map map`?

3.4 Gli operatori di fold

Molti degli operatori che abbiamo visto sinora restituiscono come risultato delle liste. Gli operatori di fold sono più generali, in quanto convertono liste in altri valori. Considereremo gli operatori `fold_left` e `fold_right`. La definizione informale dell'operatore `fold_right` è

$$fold\_right\ f\ a\ [x_1; x_2; \ldots; x_n] = f\ x_1\ (f\ x_2\ (\ldots (f\ x_n\ a)\ldots))$$

dove f è una variabile legata ad una funzione binaria. In particolare, abbiamo:

$$\begin{aligned} fold\_right\ f\ a\ [] &= a \\ fold\_right\ f\ a\ [x_1] &= f\ x_1\ a \\ fold\_right\ f\ a\ [x_1; x_2] &= f\ x_1\ (f\ x_2\ a) \\ &\vdots \end{aligned}$$

Dalla definizione informale siamo già in grado di capire che il tipo del secondo argomento deve essere uguale al codominio di f. Infatti:

```
#fold_right;;
- : ('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 $[x_1; x_2; \ldots; x_n]$ è una lista di interi, allora

$$sumlist\ [x_1; x_2; \ldots; x_n] = \sum_{i=1}^{n} x_i$$

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 (con codominio e dominio uguali) è associativa ed ha un elemento neutro (esiste cioè un elemento a tale che $f\ x\ a = f\ a\ x = x$).

È possibile invece definire altre funzioni con l'ausilio di funzioni non associative. Per esempio, è molto semplice definire `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 $x_1, x_2, \ldots, x_n$ alla fine di una lista inizialmente vuota.

L'operatore `fold_left` ha un comportamento *duale* rispetto all'operatore `fold_right`. Informalmente,

$$fold\_left\ f\ a\ [x_1; x_2; \ldots; x_n] = f\ (\ldots f\ (f\ a\ x_1)\ x_2 \ldots)\ x_n$$

In particolare,

$$\begin{aligned} fold\_left\ f\ a\ [] &= a \\ fold\_left\ f\ a\ [x_1] &= f\ a\ x_1 \\ fold\_left\ f\ a\ [x_1; x_2] &= f\ (f\ a\ x_1)\ x_2 \\ &\vdots \end{aligned}$$

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).

```
#fold_left;;
- : ('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 hanno esattamente lo stesso tipo.

Un esempio di funzione definibile per mezzo di `fold_left` è la seguente:

```
#let pack xs = let decimal n x = 10*n +x
               in fold_left decimal 0 xs;;
pack : int list -> int = <fun>
```

Tale funzione codifica una sequenza di cifre in un numero singolo secondo il sistema decimale:

$$pack\ [x_0; x_1; \ldots; x_n] = \sum_{k=0}^{n-1} x_k 10^k$$

Vale la pena sottolineare alcune importanti proprietà degli operatori di fold. Se la funzione f è associativa e a è un elemento neutro di f, allora

$$fold\_right\ f\ a\ xs = fold\_left\ f\ a\ xs$$

Cioè, i due operatori definiscono la stessa funzione. Ad esempio, la funzione `sumlist` definita sopra (e tutte le altre funzioni elencate) può essere equivalentemente definita utilizzando `fold_left`.

3.4.1 Fold su liste non vuote

Si supponga di voler calcolare il massimo valore di una lista. Utilizzando un operatore di fold, la definizione dovrebbe venire abbastanza semplice:

$$maxlist = fold\_left\ (\max)\ a$$

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:

$$\begin{aligned} foldl\ f\ [x_1; x_2; \ldots; x_n] &= f(\ldots f\ (f\ x_1\ x_2)\ x_3)\ldots)\ x_n \\ foldr\ f\ [x_1; x_2; \ldots; x_n] &= f\ x_1(f\ x_2\ (\ldots(f\ x_{n-1}\ x_n)\ldots)) \end{aligned}$$

In particolare,

$$\begin{aligned} foldl\ f\ [x_1] &= x_1 \\ foldl\ f\ [x_1; x_2] &= f\ x_1\ x_2 \\ &\vdots \end{aligned}$$

Si noti come il tipo degli operatori sia definito solo a partire da un unico tipo:

```
#foldl;;
- : ('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>
```

Esercizi

1 Si definiscano le funzioni `for_all` e `exists` in termini di un operatore di fold.

2 Si consideri la seguente definizione della funzione `insert`:

```
let insert x xs = let lessthanx y = y <= x
                  in takewhile lessthanx xs @ [x] @ dropwhile lessthanx xs
```

Utilizzando tale definizione, si definisca la funzione `isort` per ordinare una lista.

3 La funzione `remdup` rimuove i duplicati adiacenti in una lista. Ad esempio,

```
remdup [1;2;2;3;3;3;1;1]=[1;2;3;1]
```

Si definisca `remdup` utilizzando un operatore di fold.

4 Si definisca una funzione f tale che:

$$length\ xs = fold\_right\ f\ 0\ xs$$

Capitolo 4

Ricorsione ed Induzione

Abbiamo già visto numerosi esempi di funzioni ricorsive (di funzioni, cioè, definite in termini di sé stesse). In questo capitolo studieremo le idee che stanno alla base delle definizioni ricorsive, e analizzeremo una nozione correlata: l'induzione. L'induzione è una particolare tecnica che permette di derivare delle leggi generali su un insieme verificando che in alcuni casi particolarmente significativi tali leggi valgono.

Cominceremo vedendo come tali concetti si applicano al caso di numeri naturali, e continueremo trattando le liste. In particolare, molte delle funzioni introdotte nel capitolo precedente verranno analizzate in questo capitolo, e l'induzione verrà utilizzata per dimostrare alcune delle proprietà viste. Alla fine del capitolo verrà trattata la nozione generale di induzione, a cui tutti i casi di induzione sui naturali e su liste possono essere ricondotti: l'induzione ben fondata.

4.1 Ricorsione ed induzione sui numeri naturali

Cominciamo esaminando la ricorsione e l'induzione in contesti più familiari: la definizione dell'operatore di esponenziazione. Abbiamo già introdotto l'operatore col simbolo `**`; cercheremo adesso di ridefinirlo in maniera ricorsiva, quando l'esponente è un numero naturale.

Nei testi matematici, l'operatore di esponenziazione `exp x n` che vogliamo definire è generalmente denotato con x^n. Le equazioni che lo definiscono sono:

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

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))
```

Si noti come la definizione tramite pattern matching rispecchi esattamente le equazioni originali. La funzione non è definita per interi negativi; al contrario, il suo significato nel caso di interi positivi dovrebbe essere abbastanza chiaro. Per esempio, per calcolare `exp 2 3`, possiamo ridurre l'espressione nel seguente modo:

$$\begin{aligned} & exp\ 2\ 3 \\ = & \quad \{ \text{ secondo pattern di } \texttt{exp} \ \} \\ & 2 \times (exp\ 2\ 2) \end{aligned}$$

$$
\begin{aligned}
&= && \{ \text{ secondo pattern di } \texttt{exp} \ \} \\
&\quad 2 \times (2 \times (\mathit{exp}\ 2\ 1)) \\
&= && \{ \text{ secondo pattern di } \texttt{exp} \ \} \\
&\quad 2 \times (2 \times (2 \times (\mathit{exp}\ 2\ 0))) \\
&= && \{ \text{ primo pattern di } \texttt{exp} \ \} \\
&\quad 2 \times (2 \times (2 \times 1)) \\
&= && \{ \text{ algebra } \} \\
&\quad 8
\end{aligned}
$$

L'argomento 3 viene legato al pattern n (poiché può essere legato e la condizione è soddisfatta) e al contrario non può essere legato a 0. Nella valutazione si prosegue così legando l'argomento o con la prima o con la seconda equazione di pattern matching (ma mai entrambe).

In questi tipi di equazioni 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: $\mathit{exp}\ x\ 0$ ha un valore, poiché tale valore è dato dalla prima equazione del pattern matching. Per la seconda equazione possiamo dire che se $\mathit{exp}\ x\ (n-1)$ ha un valore, allora anche $\mathit{exp}\ x\ n$ è definito. Così exp ha un valore per ogni numero naturale n, come richiesto.

Nella pratica, tale forma di ragionamento può anche essere applicata all'indietro: dati x ed n, il valore di $\mathit{exp}\ x\ n$ è trovato calcolando il valore di $\mathit{exp}\ x\ (n-1)$, a sua volta trovato calcolando $\mathit{exp}\ x\ (n-2)$, e così via, fino a quando non ci si riduce a dover calcolare il valore di $\mathit{exp}\ x\ 0$.

Questo tipo di ragionamento è un esempio di dimostrazione tramite *induzione matematica*. In generale, per dimostrare che una proposizione $P(n)$ è valida per ogni valore di n, 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.

È facile convincersi che se tali proprietà sono valide, allora $P(n)$ è valida per ogni valore di n. 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

$$\mathit{exp}\ x\ (m+n) = (\mathit{exp}\ x\ n) \times (\mathit{exp}\ x\ m)$$

per ogni numero naturale m ed n.

Dimostrazione. Dimostriamolo per induzione su m.

Caso 0. Abbiamo:

$$
\begin{aligned}
&\quad \mathit{exp}\ x\ (0+n) \\
&= && \{ \ 0 \text{ è l'elemento neutro della somma } \} \\
&\quad \mathit{exp}\ x\ n \\
&= && \{ \ 1 \text{ è l'elemento neutro del prodotto } \} \\
&\quad 1 \times (\mathit{exp}\ x\ n) \\
&= && \{ \text{ primo pattern di } \texttt{exp} \ \} \\
&\quad (\exp\ x\ 0) \times (\mathit{exp}\ x\ n)
\end{aligned}
$$

Caso (m+1). Assumiamo per ipotesi che $exp\ x\ (m+n) = (exp\ x\ m) \times (exp\ x\ n)$ (tale ipotesi è chiamata *ipotesi induttiva*). Allora:

$exp\ x\ ((m+1)+n)$
$=$ { proprietà della somma }
$exp\ x\ ((m+n)+1)$
$=$ { secondo pattern di `exp` }
$x \times (exp\ x\ (m+n))$
$=$ { ipotesi induttiva }
$x \times ((exp\ x\ n) \times (exp\ x\ m))$
$=$ { proprietà del prodotto }
$exp\ x\ (m+n) = (x \times (exp\ x\ n)) \times (exp\ x\ m)$
$=$ { secondo pattern di `exp` }
$(exp\ x\ (m+1)) \times (exp\ x\ n)$

□

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:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{k+2} &= F_k + F_{k+1} \end{aligned}$$

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 `fib n`, che calcola F_n:

```
let rec fib n = match n with
                  0  -> 0
                | 1  -> 1
                | x  -> fib (x-1) + fib (x-2)
```

Anche in questo caso ogni pattern rispecchia una delle equazioni precedentemente definite.

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

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

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

$$F_k = \frac{1}{\sqrt{5}}(\phi^k - \widehat{\phi}^k)$$

per ogni k.

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

Caso 0. $F_0 = 0 = c(\phi^0 - \widehat{\phi}^0)$.

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

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

$$
\begin{array}{ll}
 & F_{k+2} \\
= & \quad \{ \text{ terza equazione della definizione } \} \\
 & F_k + F_{k+1} \\
= & \quad \{ \text{ ipotesi induttiva } \} \\
 & c(\phi^k - \widehat{\phi}^k) + c(\phi^{k+1} - \widehat{\phi}^{k+1}) \\
= & \quad \{ \text{ aritmetica } \} \\
 & c(\phi^k(1+\phi) - \widehat{\phi}^k(1+\widehat{\phi})) \\
= & \quad \{ \ \phi^2 = \phi + 1 \text{ e } \widehat{\phi}^2 = \widehat{\phi} + 1 \ \} \\
 & c(\phi^{k+2} - \widehat{\phi}^{k+2})
\end{array}
$$

□

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.

Esercizi

1 Si definiscano ricorsivamente la somma e il prodotto di due numeri naturali.

2 Utilizzando le definizioni dell'esercizio precedente, si dimostrino le seguenti uguaglianze:

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

4.2 Ricorsione ed induzione su liste

I principi di induzione e ricorsione 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 *cons* 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`, che abbiamo introdotto nel precedente capitolo:

```
let rec length l = match l with
              []    -> 0
           | x::xs -> 1 + (length xs)
```

Si noti che l'utilizzo di [ ] e $(x :: xs)$ ricalca esattamente l'uso di 0 e $(n+1)$ che avevamo fatto nel precedente paragrafo.

Possiamo utilizzare la definizione sopra per calcolare la lunghezza della lista $[2;5]$:

$$
\begin{array}{ll}
 & \mathit{length}\ [2;5] \\
= & \quad \{ \text{ secondo pattern di } \texttt{length} \ \} \\
 & 1 + (\mathit{length}\ [5] \\
= & \quad \{ \text{ secondo pattern di } \texttt{length} \ \} \\
 & 1 + (1 + (\mathit{length}\ [])) \\
= & \quad \{ \text{ primo pattern di } \texttt{length} \ \} \\
 & 1 + (1 + 0) \\
= & \quad \{ \text{ algebra } \} \\
 & 2
\end{array}
$$

Vediamo come possiamo definire l'operatore di concatenazione:

```
let rec @ xs ys = match xs with
              []     -> ys
             |z::zs  -> z::(@ zs ys)
```

(si noti che in questa definizione l'operatore è definito come prefisso). È possibile definire ricorsivamente anche la sua estensione:

```
let rec concat ls = match ls with
                                []    -> []
                        | (xs::xss) -> xs @ (concat xss)
```

Il principio di induzione su liste ricalca lo stesso principio sui numeri naturali. Per dimostrare che $P(xs)$ 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:

$$xs \mathbin{@} (ys \mathbin{@} zs) = (xs \mathbin{@} ys) \mathbin{@} zs$$

Per ogni tripla xs, ys, zs di liste finite.

Dimostrazione. La dimostrazione viene data per induzione su xs.

Caso [ ]. Abbiamo:

$$\begin{array}{ll} & [\,] \mathbin{@} (xs \mathbin{@} ys) \\ = & \quad \{ \text{ primo pattern di } \texttt{@} \ \} \\ & ys \mathbin{@} zs \\ = & \quad \{ \text{ primo pattern di } \texttt{@} \ \} \\ & ([\,] \mathbin{@} xs) \mathbin{@} ys \end{array}$$

Caso $(x :: xs)$. In questo caso,

$$\begin{array}{ll} & (x :: xs) \mathbin{@} (ys \mathbin{@} zs) \\ = & \quad \{ \text{ secondo pattern di } \texttt{@} \ \} \\ & x :: (xs \mathbin{@} (ys \mathbin{@} zs)) \\ = & \quad \{ \text{ ipotesi induttiva } \} \\ & x :: ((xs \mathbin{@} ys) \mathbin{@} zs) \\ = & \quad \{ \text{ secondo pattern di } \texttt{@} \ \} \\ & (x :: (xs \mathbin{@} ys)) \mathbin{@} zs \\ = & \quad \{ \text{ secondo pattern di } \texttt{@} \ \} \\ & ((x :: xs) \mathbin{@} ys) \mathbin{@} zs \end{array}$$

□

La dimostrazione vista ci dice ben poco sulla metodologia applicata per svilupparla. Se una presentazione elegante della dimostrazione di una proprietà aumenta la comprensibilità della dimostrazione stessa, dall'altro lato uno sviluppo più "rozzo" della dimostrazione ci permette di capire i meccanismi che ci hanno permesso di ottenerla. Quel che vogliamo dimostrare, nel caso induttivo, è la seguente proprietà:

$$(x :: xs) \mathbin{@} (ys \mathbin{@} zs) = ((x :: xs) \mathbin{@} ys) \mathbin{@} zs$$

Tale dimostrazione può essere ottenuta nel seguente modo. Innanzitutto, cominciamo a sviluppare il lato sinistro dell'identità, cercando di semplificarlo[1] il più possibile:

$$\begin{array}{ll} & (x :: xs) \mathbin{@} (ys \mathbin{@} zs) \\ = & \quad \{ \text{ secondo pattern di } \texttt{@} \ \} \\ & x :: (xs \mathbin{@} (ys \mathbin{@} zs)) \\ = & \quad \{ \text{ ipotesi induttiva } \} \\ & x :: ((xs \mathbin{@} ys) \mathbin{@} zs) \end{array}$$

In secondo luogo, possiamo provare a fare la stessa cosa con il termine destro:

[1] Qui, "semplificare" significa sostituire un termine (o una parte di esso) con la parte destra di una definizione la cui parte sinistra corrisponde al termine stesso.

$$
\begin{array}{ll}
 & ((x :: xs) \mathbin{@} ys) \mathbin{@} zs \\
= & \quad \{ \text{ secondo pattern di } \texttt{@} \ \} \\
 & (x :: (xs \mathbin{@} ys)) \mathbin{@} zs \\
= & \quad \{ \text{ secondo pattern di } \texttt{@} \ \} \\
 & x :: ((xs \mathbin{@} ys) \mathbin{@} zs)
\end{array}
$$

Poiché i due risultati sono identici, abbiamo dimostrato l'identità. La dimostrazione è ottenuta aggiungendo al primo gruppo di equazioni il secondo gruppo, in ordine inverso.

Esercizi

1 Si dimostri per induzione che

$$length\ (xs \mathbin{@} ys) = (length\ xs) + (length\ ys)$$

4.3 Operazioni su liste

Nel capitolo precedente abbiamo visto molte importanti operazioni su liste. Molte di queste operazioni possono essere facilmente definite tramite ricorsione, e molte loro proprietà dimostrate per induzione. Definiremo tali funzioni in maniera tale da poter presentare alcune interessanti variazioni ai principi introdotti nella precedente sezione.

4.3.1 *combine* e *split*

La funzione `combine` prende come argomento una coppia di liste e restituisce una lista di coppie. La sua definizione è la seguente:

```
let rec combine (l,l1) = match l,l1 with
         ([],ys)       -> []
      |  (x::xs,[])    -> []
      |  (x::xs,y::ys) -> (x,y)::(combine (xs,ys))
```

Si noti che tale 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.

È facile adattare lo stile delle dimostrazioni per induzione a tali patterns. Per esempio, è possibile vedere che

$$length\ combine\ (xs, ys) = min\ (length\ xs)\ (length\ ys)$$

per ogni coppia di liste xs, ys.

Dimostrazione. La dimostrazione è per induzione su xs e ys.

Caso $[\,], ys$**.** Allora:

$$
\begin{aligned}
& \mathit{length}\ \mathit{combine}\ ([\,], ys) \\
= & \quad \{ \text{ primo pattern di } \texttt{combine} \ \} \\
& \mathit{length}\ [\,] \\
= & \quad \{ \text{ primo pattern di } \texttt{length} \ \} \\
& 0 \\
= & \quad \{ \text{ definizione di min } \} \\
& \mathit{min}\ 0\ (\mathit{length}\ ys) \\
= & \quad \{ \text{ primo pattern di } \texttt{length} \ \} \\
& \mathit{min}\ (\mathit{lenght}\ [\,])\ (\mathit{length}\ xs)
\end{aligned}
$$

Caso $(x :: xs), [\,]$**.** In maniera analoga al caso precedente:

$$
\begin{aligned}
& \mathit{length}\ \mathit{combine}\ ((x :: xs), [\,]) \\
= & \quad \{ \text{ secondo pattern di } \texttt{combine} \ \} \\
& \mathit{length}\ [\,] \\
= & \quad \{ \text{ primo pattern di } \texttt{length} \ \} \\
& 0 \\
= & \quad \{ \text{ definizione di min } \} \\
& \text{min}\ (\mathit{length}\ x :: xs)\ 0 \\
= & \quad \{ \text{ primo pattern di } \texttt{length} \ \} \\
& \mathit{min}\ (\mathit{length}\ x :: xs)\ (\mathit{length}\ [\,])
\end{aligned}
$$

Caso $(x :: xs), (y :: ys)$**.** Supponiamo che $\mathit{length}\ \mathit{combine}\ (xs, ys) = \mathit{min}\ (\mathit{length}\ xs)\ (\mathit{length}\ ys)$. Allora:

$$
\begin{aligned}
& \mathit{length}\ \mathit{combine}\ (x :: xs, y :: ys) \\
= & \quad \{ \text{ terzo pattern di } \texttt{combine} \ \} \\
& \mathit{length}\ (x, y) :: (\mathit{combine}\ (xs, ys)) \\
= & \quad \{ \text{ secondo pattern di } \texttt{length} \ \} \\
& 1 + \mathit{length}\ (\mathit{combine}\ (xs, ys) \\
= & \quad \{ \text{ ipotesi induttiva } \} \\
& 1 + (\text{min}\ \ xs\ ys) \\
= & \quad \{ \text{ la somma distribuisce su min } \} \\
& \text{min}\ (1 + xs)\ (1 + ys) \\
= & \quad \{ \text{ secondo pattern di } \texttt{length} \ \} \\
& \text{min}\ (\mathit{length}\ x :: xs)\ (\mathit{length}\ y :: ys)
\end{aligned}
$$

□

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.

4.3.2 *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 xs = match n, xs with
                      x, []      -> []
                    | 0, ys      -> []
                    | x, (y::ys) -> y::(take (x-1) ys);;

let rec drop n xs = match n, xs with
                      0, ys      -> xs
                    | x, []      -> []
                    | x, (y::ys) -> drop (x-1) ys;;
```

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. Seguendo lo schema della precedente sezione, possiamo dimostrare abbastanza facilmente tale teorema.

Dimostrazione. Dimostriamolo per induzione su n ed xs.

Caso $0, xs$**.** Allora:

$$\begin{array}{ll} & take\ 0\ xs\ @\ drop\ 0\ xs \\ = & \quad \{ \text{ primo pattern di } \texttt{take} \text{ e } \texttt{drop} \ \} \\ & [\,]\ @\ xs \\ = & \quad \{ \text{ primo pattern di } \texttt{@} \ \} \\ & xs \end{array}$$

Caso $(n+1), [\,]$**.** In tal caso,

$$\begin{array}{ll} & take\ (n+1)\ [\,]\ @\ drop\ (n+1)\ [\,] \\ = & \quad \{ \text{ secondo pattern di } \texttt{take} \text{ e } \texttt{drop} \ \} \\ & [\,]\ @\ [\,] \\ = & \quad \{ \text{ primo pattern di } \texttt{@} \ \} \\ & [\,] \end{array}$$

Caso $(n+1), (x :: xs)$**.** In quest'ultimo caso, abbiamo:

$$\begin{array}{ll} & take\ (n+1)\ (x :: xs)\ @\ drop\ (n+1)\ (x :: xs) \\ = & \quad \{ \text{ terzo pattern di } \texttt{take} \text{ e } \texttt{drop} \ \} \\ & (x :: take\ n\ xs)\ @\ drop\ n\ xs \\ = & \quad \{ \text{ secondo pattern di } \texttt{@} \ \} \\ & x :: (take\ n\ xs\ @\ drop\ n\ xs) \\ = & \quad \{ \text{ ipotesi induttiva } \} \\ & x :: xs \end{array}$$

□

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.

4.3.3 *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 data lista. La loro definizione ricorsiva è la seguente:

```
let rec map f l =
          match l with
                []    -> []
              | x::xs -> (f x)::(map f xs)

let rec filter p xs =
          match xs with
                []    -> []
              | y::ys when p y -> y ::(filter p ys)
              | y::ys when not (p y) -> (filter p ys)
```

La novità di queste definizioni consiste nel poter definire funzioni di ordine superiore (*higher-order*) tramite definizioni ricorsive, e di poter includere nelle definizioni ricorsive espressioni condizionali.

Una proprietà di `map` e `filter` è la seguente:

$$filter\ p\ (map\ f\ xs) = map\ f\ (filter\ (compose\ p\ f)\ xs)$$

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:

$$\begin{array}{ll} & filter\ p\ (map\ f\ [\,]) \\ = & \quad \{ \text{ primo pattern di } \texttt{map} \ \} \\ & filter\ p\ [\,] \\ = & \quad \{ \text{ primo pattern di } \texttt{filter} \ \} \\ & [\,] \\ = & \quad \{ \text{ primo pattern di } \texttt{map} \ \} \\ & map\ f\ [\,] \\ = & \quad \{ \text{ primo pattern di } \texttt{filter} \ \} \\ & map\ f\ (filter\ (compose\ p\ f)\ [\,] \end{array}$$

caso $(x :: xs), p\ (f\ x) = true$**.** In tal caso,

$$\begin{array}{ll} & filter\ p\ (map\ f\ x :: xs) \\ = & \quad \{ \text{ secondo pattern di } \texttt{map} \ \} \\ & filter\ p\ (f\ x :: (map\ f\ xs)) \\ = & \quad \{ \text{ secondo pattern di } \texttt{filter}, p\ (f\ x) = true \ \} \\ & f\ x :: filter\ p\ (map\ f\ xs) \\ = & \quad \{ \text{ ipotesi induttiva } \} \\ & f\ x :: (map\ f\ (filter\ (compose\ p\ f)\ xs) \\ = & \quad \{ \text{ secondo pattern di } \texttt{map} \ \} \end{array}$$

$map\ f(x :: filter\ (compose\ p\ f)\ xs)$
$=$ { secondo pattern di `filter`, $p\ (f\ x) = true$ }
$map\ f\ (filter\ (compose\ p\ f)\ x :: xs$

caso $(x :: xs), p\ (f\ x) = false$**.** In tal caso,

$filter\ p\ (map\ f\ x :: xs)$
$=$ { secondo pattern di `map` }
$filter\ p\ (f\ x :: (map\ f\ xs))$
$=$ { secondo pattern di `filter`, $p\ (f\ x) = false$ }
$filter\ p\ (map\ f\ xs)$
$=$ { ipotesi induttiva }
$map\ f\ (filter\ (compose\ p\ f)\ xs$
$=$ { secondo pattern di `filter`, $p\ (f\ x) = false$ }
$map\ f\ (filter\ (compose\ p\ f)\ x :: xs$

□

4.3.4 Intervalli

In matematica, 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
```

In questa definizione i due casi sono distinti dal test su $m > n$ piuttosto che dal pattern matching. Il valore di `interval m n` è ben definito per ogni coppia 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:

$$map\ ((+)\ k)\ (interval\ m\ n) = interval\ (k + m)\ (k + n)$$

Dimostrazione. La dimostrazione è per induzione sulla differenza tra m ed n.

Caso $m > n$**.** Allora:

$map\ ((+)\ k)\ (interval\ m\ n)$
$=$ { prima alternativa di `interval`, poiché $m > n \Rightarrow k + m > k + n$ }
$map\ ((+)\ k)\ [\,]$
$=$ { primo pattern di `map` }
$[\,]$
$=$ { prima alternativa di `interval` }
$interval\ (k + m)\ (k + n)$

Caso $m \leq n$. In tal caso,

$$
\begin{array}{ll}
 & map\ ((+)\ k)\ (interval\ m\ n) \\
= & \quad \{ \text{ seconda alternativa di } \texttt{interval} \ \} \\
 & map\ ((+)\ k)\ (m :: interval\ (m+1)\ (n)) \\
= & \quad \{ \text{ secondo pattern di } \texttt{map} \ \} \\
 & (k+m) :: (map\ ((+)\ k)\ (interval\ (m+1)\ n)) \\
= & \quad \{ \text{ ipotesi induttiva } \} \\
 & (k+m) :: interval\ (k+m+1)\ (k+n) \\
= & \quad \{ \text{ seconda alternativa di } \texttt{interval} \ \} \\
 & interval\ (k+m)\ (k+n)
\end{array}
$$

□

Esercizi

1 Si definisca ricorsivamente la funzione *nth* in modo che $nth\ m\ l$ selezioni l'elemento di posto m della lista l e si verifichi che $nth\ m\ (drop\ n\ l) = nth\ (n+m)\ l$ per ogni l, m ed n (suggerimento: ci si lasci guidare dalla definizione).

2 Si definiscano le funzioni `takewhile` e `dropwhile` che prendono come argomenti un predicato ed una lista l e restituiscono la sottolista l_1 (resp. l_2) di l tale che $l = l_1@l_2$ e ogni elemento di l_1 soddisfa (resp. il primo elemanto di l_2 non soddisfa) il predicato. Ad esempio, `takewhile even [2;4;5;6]=[2;4]` e `dropwhile even [2;4;5;6]=[5;6]`.

3 Si definisca 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 a, se tale coppia esiste (cioè, `assoc a [...; (a,b);...] = b`).

4 Si dia una definizione ricorsiva delle funzioni *init* e *last* definite nel precedente capitolo. Si dimostri quindi che

$$
\begin{array}{c}
init\ (xs@[x]) = xs \\
last\ (xs@[x]) = x \\
xs = init\ xs@[last\ xs]
\end{array}
$$

5 Si diano le definizioni di `fold_left`, `fold_right`, `foldl` e `foldr`.

6 Si dimostrino le leggi

$$
\begin{array}{rcl}
take\ m\ (drop\ n\ xs) & = & drop\ n\ (take\ (m+n)\ xs) \\
drop\ m\ (drop\ n\ xs) & = & drop\ (m+n)\ xs \\
map\ (compose\ f\ g)\ xs & = & map\ f\ (map\ g\ xs) \\
map\ f\ (concat\ xss) & = & concat\ (map(mapf)xss)
\end{array}
$$

7 Si consideri la funzione *interval*1 n, che rappresenta l'intervallo $[1; \ldots; n]$. Si dimostri che la definizione

```
interval1 x = match x with
              0  -> []
            | n  -> 1::map ((+) 1) (interval1 (n-1))
```

segue dalla definizione di `interval` e dalla legge analizzata in questo paragrafo.

4.4 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.

4.4.1 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
```

Come si può vedere, è conveniente definire `diff` in termini di una definizione ausiliaria, `remove`:

```
let rec remove xs y =  match xs with
                         []            -> []
                       | z::zs when z=y -> zs
                       | z::zs when z<>y -> z::remove zs y
```

Una importante legge che lega concatenazione e differenza di liste è la seguente:

$$diff\ (xs \mathbin{@} ys)\ xs = ys$$

La dimostrazione è immediata, utilizzando le tecniche della sezione precedente, ed è lasciata come esercizio.

4.4.2 *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 l = match l with
                 []      -> []
               | x::xs   -> rev xs @ [x]
```

Dimostriamo che

$$rev\ (rev\ xs) = xs$$

per ogni lista xs.

Dimostrazione. Per dimostrare l'enunciato, abbiamo bisogno di un risultato ausiliario:

$$rev\ (ys \mathbin{@} [x]) = x :: rev\ ys$$

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:

$rev\ (rev\ [\,])$
$=$ { primo pattern di `rev` }
$rev\ [\,]$
$=$ { primo pattern di `rev` }
$[\,]$

caso $(x :: xs)$**.** Allora:

$rev\ (rev\ x :: xs)$
$=$ { secondo pattern di `rev` }
$rev\ ((rev\ xs) \mathbin{@} [x])$
$=$ { per il risultato di cui sopra }
$x :: rev\ (rev\ xs)$
$=$ { ipotesi induttiva }
$x :: xs$

□

Nella dimostrazione principale, abbiamo utilizzato il risultato ausiliario per dimostrare che

$$rev\ ((rev\ xs) \mathbin{@} [x]) = x :: rev\ (rev\ xs)$$

Si può verificare che non è affatto semplice cercare di dimostrare tale risultato direttamente. Sostituendo invece la sottoepressione $rev\ xs$ 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 $[x] \mathbin{@} xs$, si potrebbe provare a trarre dal secondo pattern di `rev` la seguente uguaglianza:

$$rev\ ([x] \mathbin{@} xs) = (rev\ xs) \mathbin{@} [x]$$

Da tale uguaglianza si può derivare un'uguaglianza praticamente simile

$$rev\ (xs \mathbin{@} [x]) = [x] \mathbin{@} xs$$

che è il risultato ausiliario di cui avevamo bisogno.

4.4.3 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 $[3; 2] \mathbin{@} [1]$, 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] @ [1]) @ [2]
⇝       { secondo pattern di rev }
    ((rev  [] @ [1]) @ [2]) @ [1]
⇝       { primo pattern di rev }
    (([] @ [3]) @ [2]) @ [1]
⇝       { primo pattern di @ }
    ([3] @ [2]) @ [1]
⇝       { secondo e primo pattern di @ }
    [3, 2] @ [1]
⇝       { secondo, 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 di `rev` e da $1+2+\ldots+n = n(n-1)/2$ 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:

```
let reverse xs = revit xs []
```

dove la funzione `revit` è definita come

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

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 n.

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 $fastfib\ n = fib\ n$.

Esercizi

1 Si dimostri che

$$diff\ (xs \mathbin{@} ys)\ xs = ys$$

2 Si dimostri che

$$rev\ (xs \mathbin{@} ys) = (rev\ ys) \mathbin{@} (rev\ xs)$$

3 Si dimostri che

$$fold\_right\ f\ a\ (xs \mathbin{@} ys) = fold\_right\ f\ (fold\_right\ f\ a\ ys)\ xs$$

4.5 Funzioni combinatorie

Molti problemi interessanti sono *combinatori* per natura, in quanto richiedono la selezione e/o la permutazione degli elementi di una lista in un qualche modo. Vediamo alcune funzioni che permettono di effettuare tali manipolazioni.

Segmenti iniziali. Possiamo definire una funzione `inits` che calcola la lista di tutti i segmenti iniziali di una lista, per lunghezza crescente:

```
#let rec inits ls =
              match ls with
                     []     -> [[]]
                   | x::xs  -> [[]] @ map (curry (prefix ::) x) (inits xs)
inits : 'a list -> 'a list list = <fun>
# inits [1;2;3;4];;
- : int list list = [[]; [1]; [1; 2]; [1; 2; 3]; [1; 2; 3; 4]]
```

La lista vuota ha come argomento 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.

```
#  map sumlist (inits (interval 1 10));;
- : int list = [0; 1; 3; 6; 10; 15; 21; 28; 36; 45; 55]
```

Sottosequenze. Le funzione `subs` restituisce la lista di tutte le sottosequenze di una lista:

```
#  let rec subs l =
        match l with
                []    -> [[]]
              | x::xs ->(subs xs) @ map (curry (prefix ::) 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 sottosequenza; dal secondo pattern, una lista $x :: xs$ ha come sottosequenze tutte le sottosequenze di xs, con o senza l'elemento x. Tale definizione può essere ottimizzata utilizzando le tecniche viste nel capitolo precedente. Si noti che questa definizione è differente dalla definizione di `init` solo in un punto, che tuttavia è determinante.

Interleaving. La funzione `interleave x ys` restituisce la lista di tutte le possibili liste ottenute da `ys` inserendo il termine `x`:

```
#  let rec interleave x ls =
        match ls with
           []      -> [[x]]
         | y::ys   -> [x::y::ys] @ map (curry (prefix ::) 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.

Permutazioni. La funzione `perms` calcola tutte le possibili permutazioni di una lista:

```
#let rec perms l =
    match l 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:

$$\begin{aligned} perms &: \text{'a list} -> \text{'a list list} \\ interleave &: \text{'a} -> \text{'a list} -> \text{'a list list} \end{aligned}$$

Assumendo $x :$ 'a e $xs :$ 'a list, otteniamo:

$$(map\ (interleave\ x)\ (perms\ xs)) : \alpha\ list\ list\ list.$$

Esercizi

1 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];;
- : [[1; 2]; [1; 3]; [2; 3]] = int list list;;
```

Si dia una definizione ricorsiva di `choose`.

2 Si dimostri che

$$subs\ (map\ f\ xs) = map\ (map\ f)\ (subs\ xs)$$

4.6 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 $\mathcal{R}$ su S è un insieme di coppie di elementi di S. Nel seguito useremo la notazione infissa $x\mathcal{R}y$ per denotare il fatto che x *precede* y secondo la relazione $\mathcal{R}$, e denoteremo una relazione $\mathcal{R}$ che esprime una precedenza indifferentemente con i simboli $\sqsubset$ o $\prec$.

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 nessuna coppia m ed n con $n \neq m$ tali che $m < n$ e $m < n$. Se la relazione $<$ 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 esista nessun $s \in S$ per cui $m \sqsubset s$. 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 $\ldots \sqsubset s_{m-1} \sqsubset s_m \sqsubset s_{m+1} \sqsubset \ldots$ 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 $0 \sqsubset 1 \sqsubset 2 \sqsubset \ldots$. Si noti che tale catena è infinita ascendente, ma non infinita discendente (non esiste un predecessore di 0). Se consideriamo invece l'insieme $\mathbf{Z}$ dei numeri interi con la stessa relazione $\sqsubset$, allora l'insieme dei numeri negativi è una catena infinita discendente (infatti $\ldots -2 \sqsubset -1 \sqsubset 0$). Si consideri ora l'insieme $\{1/n | n \geq 1\}$ con la relazione "$x \prec y$ se e solo se esiste un numero naturale k tale che $y = k \cdot x$". Ad esempio, $1/4 \prec 1/2$, poiché $1/2 = 2 \cdot 1/4$. Si considerino le catene $\{\frac{1}{p^k} | p \text{ è un numero primo }\}_{k \geq 1}$. Tali catene sono discendenti infinite: ad esempio, la catena infinita $\ldots 1/8 \sqsubset 1/4 \sqsubset 1/2$ è tale che $1/2 = 2 \cdot 1/4$, $1/4 = 2 \cdot 1/8$ e così via.

Si noti che se una relazione di precedenza $\sqsubset$ contiene dei *cicli*, come ad esempio in $1 \sqsubset 2$, $2 \sqsubset 5$ e $5 \sqsubset 1$, essa non è ben fondata. Nell'esempio, è possibile costruire la catena discendente infinita $\ldots 2 \sqsubset 5 \sqsubset 1 \sqsubset 2 \sqsubset 5 \sqsubset 1$.

La presenza o meno di catene infinite discendenti è una proprietà tipica dell'insieme rispetto alla relazione di precedenza che viene definita. L'insieme $\mathbb{N}$, per esempio non ha catene discententi se consideriamo la relazione $\sqsubset$. Tuttavia, se proviamo ad invertire la relazione in

"$x \sqsubset' y$ se e solo se $x = y + 1$", allora la catena discendente infinita $\ldots \sqsubset' n+1 \sqsubset' n \ldots \sqsubset' 1 \sqsubset' 0$ è propria dell'insieme con la relazione $\sqsubset'$.

Si noti come la presenza di una catena infinita discendente 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 ritrovarci 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 $n_i \sqsupset n_{i+1}$. Questo ci suggerisce di imporre l'assenza di catene infinite discendenti nell'insieme S che andiamo ad esaminare.

È importante distinguere tra catene discendenti infinite e relazioni di precedenza infinite: ovviamente se una relazione di precedenza induce una catena infinita allora la relazione stessa è infinita, ma questo non vuol dire che una relazione di precedenza che non induce catene infinite è finita. In particolare, 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 $\mathbb{N}$ con la relazione di precedenza "$x \prec y$ se e solo se $y = 0$ e $x \neq 0$". In tal caso, per ogni numero naturale n avremo $n \prec 0$, e conseguentemente tutte le catene saranno composte dai soli elementi n e 0. Tuttavia, 0 ha un numero infinito di predecessori.

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, $(\mathbb{N}, \sqsubset)$ e $(\mathbb{N}, \prec)$ sono insiemi ben fondati. Non sono insiemi ben fondati, invece, $(\mathbf{Z}, \sqsubset)$, $(\{1/n | n \geq 1\}, \prec)$ e $(\mathbb{N}, \sqsubset')$.

A questo punto abbiamo tutti gli elementi per definire la generalizzazione del principio di induzione che abbiamo visto nelle precedenti sezioni. 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, abbiamo detto, 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:

$$P(m) \text{ vale per tutti gli } m \text{ minimali rispetto a } \sqsubset \tag{4.1}$$

$$\text{con } m \text{ non minimale, se per ogni } n \sqsubset m \text{ vale } P(n), \text{ allora vale anche } P(m) \tag{4.2}$$

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 (si veda l'esercizio 2). 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 $\sqsubset$, cioè tutti gli elementi che non hanno predecessori immediati;

Caso Induttivo: detto x un generico elemento di S, non minimale rispetto a $\sqsubset$, 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 $y \sqsubset x$.

È facile verificare che l'induzione naturale è un caso particolare dell'induzione ben fondata: si noti che la relazione $<$ corrisponde alla chiusura transitiva della relazione $\sqsubset$ definita sopra, e quindi l'insieme $\mathbb{N}$ con la relazione $\sqsubset$ è ben fondato. In particolare, tale insieme ha una sola catena $0 \sqsubset 1 \sqsubset 2 \sqsubset \ldots$.

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 $\sqsubset$ definita come "$xs \sqsubset ys$ se e solo se esiste un x tale che $ys = x :: xs$". La relazione $\sqsubset$ è una relazione di precedenza, poiché non possono esistere due liste xs e ys con $xs \neq ys$ tali che $ys = x_1 :: xs$ e $xs = x_2 :: ys$, dati x_1, x_2. Inoltre, ogni catena $\{xs_i\}_{i \geq 0}$ è discendente finita, poiché $[\,] \sqsubset [x]$ per ogni x e non esiste nessuna lista xs tale che $xs \sqsubset [\,]$. Da ciò si può dedurre che l'insieme $(L, \sqsubset)$ è ben fondato, e il principio di induzione ben fondata può essere applicato.

Proviamo a rivedere la dimostrazione che

$$xs \mathbin{@} (ys \mathbin{@} zs) = (xs \mathbin{@} ys) \mathbin{@} zs$$

Per ogni tripla xs, ys, zs di liste finite. Lo vogliamo dimostrare per induzione ben fondata su xs. Come abbiamo detto analizziamo due casi:

Caso Base. Come abbiamo visto, l'elemento minimale di L è $[\,]$, per cui bisogna dimostrare che vale $P([\,])$. Infatti:

$$\begin{aligned}
& [\,] \mathbin{@} (xs \mathbin{@} ys) \\
= & \quad \{ \text{ primo pattern di } \texttt{@} \ \} \\
& ys \mathbin{@} zs \\
= & \quad \{ \text{ primo pattern di } \texttt{@} \ \} \\
& ([\,] \mathbin{@} xs) \mathbin{@} ys
\end{aligned}$$

Caso Induttivo. Vogliamo dimostrare che, data una lista $\overline{xs}$, se per ogni elemento $xs \sqsubset \overline{xs}$ vale $P(xs)$, allora vale $P(\overline{xs})$. Poiché $\overline{xs}$ non è un elemento minimale, conterrà almeno un elemento x, cioè $\overline{xs} = x :: xs$ per qualche xs. In tal caso $xs \sqsubset \overline{xs}$. Ma allora,

$$\begin{aligned}
& (x :: xs) \mathbin{@} (ys \mathbin{@} zs) \\
= & \quad \{ \text{ secondo pattern di } \texttt{@} \ \} \\
& x :: (xs \mathbin{@} (ys \mathbin{@} zs)) \\
= & \quad \{ \text{ ipotesi induttiva } \} \\
& x :: ((xs \mathbin{@} ys) \mathbin{@} zs) \\
= & \quad \{ \text{ secondo pattern di } \texttt{@} \ \} \\
& (x :: (xs \mathbin{@} ys)) \mathbin{@} zs \\
= & \quad \{ \text{ secondo pattern di } \texttt{@} \ \} \\
& ((x :: xs) \mathbin{@} ys) \mathbin{@} zs
\end{aligned}$$

□

4.6.1 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 casi non minimali 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 $\sqsubset$ su D (il dominio della funzione) che esprima una relazione di precedenza e tale che $(D, \sqsubset)$ sia ben fondato;
- Si mostra che la funzione è definita sulle catene di D indotte da $\sqsubset$ (in tal modo si dimostra che la funzione è ben definita);
- Si dimostra per induzione ben fondata che la funzione in oggetto 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)
```

Si noti che la funzione, benché abbia tipo `int -> bool`, è definita sul dominio $\mathbb{N}$. Consideriamo allora, su $\mathbb{N}$, la relazione "$x \sqsubset y$ se e solo se $y = x + 2$". La coppia $(\mathbb{N}, \sqsubset)$ rappresenta un insieme ben fondato poiché le uniche catene sono la catena $0 \sqsubset 2 \sqsubset 4 \sqsubset \ldots$ dei numeri pari e la catena $1 \sqsubset 3 \sqsubset 5 \sqsubset \ldots$ dei numeri dispari, ed entrambe sono decrescenti finite. Si noti ora che gli elementi minimali del dominio sono 0 e 1, e che su tali elementi la funzione è definita. Inoltre, si consideri un elemento non minimale x: allora $x - 2 \sqsubset x$, 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 $\mathbb{N}$.

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
```

Consideriamo la relazione di precedenza "$(x, y) \sqsubset (x', y')$ se $y = y'$ e $x' = x + y$". L'insieme $(\mathbb{N} \times \mathbb{N}, \sqsubset)$ è ben fondato: infatti le coppie del tipo (n, m) con $n < m$ sono minimali e le catene sono del tipo:

$(0, m) \sqsubset (m, m) \sqsubset (2m, m) \ldots$

$(1, m) \sqsubset (m + 1, m) \sqsubset (2m + 1, m) \ldots$

$\ldots$

$(m - 1, m) \sqsubset (2m - 1, m) \sqsubset (3m - 1, m) \sqsubset \ldots.$

Dimostriamo ora che, per ogni naturale x, $pari\ x \equiv (mod\ x\ \ 2 = 0)$. Dimostriamo l'asserto per induzione ben fondata su $(\mathbb{N}, \sqsubset)$.

Caso base. I casi base sono due: il caso $x = 0$ e il caso $x = 1$. Se $x = 0$, allora

$$
\begin{array}{ll}
 & pari\ 0 \\
\equiv & \quad \{ \text{ primo pattern di } \texttt{pari} \ \} \\
 & true \\
\equiv & \quad \{ \text{ riflessività } \} \\
 & 0 = 0 \\
\equiv & \quad \{ \text{ definizione di } \texttt{mod} \ \} \\
 & (mod\ 0\ 2) = 0
\end{array}
$$

Se invece $x = 1$, allora

$$
\begin{array}{ll}
 & pari\ 1 \\
= & \quad \{ \text{ secondo pattern di } \texttt{pari} \ \} \\
 & false \\
\equiv & \quad \{ \text{ calcolo } \} \\
 & 1 = 0 \\
\equiv & \quad \{ \text{ definizione di } \texttt{mod} \ \} \\
 & (mod\ 1\ 2) = 0
\end{array}
$$

Caso induttivo. Supponiamo che x sia diverso da 0 e da 1. Dimostriamo allora la proprietà.

$$
\begin{array}{ll}
 & pari\ x \\
\equiv & \quad \{ \text{ terzo pattern di } \texttt{pari} \ \} \\
 & pari\ (x - 2) \\
\equiv & \quad \{ \text{ Ipotesi induttiva, poiché } x - 2 \sqsubset x \ \} \\
 & (mod\ (x - 2)\ 2) = 0 \\
\equiv & \quad \{ \text{ secondo caso di } \texttt{mod} \ \} \\
 & (mod\ x\ 2) = 0
\end{array}
$$

□

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 ad esempio la seguente definizione:

```
let rec maxl xs = match xs with
              [x]       -> x
            | x::y::xs when x<=y ->   maxl y::xs
            | x::y::xs when x>y  ->   maxl x::xs
```

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

$$maxl\ xs \in xs \wedge (\forall z.\ z \in xs \Rightarrow z \leq\ maxl\ xs).$$

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 $[x]$.

$$\begin{array}{ll} & maxl\ [x] \in [x] \wedge (\forall z.\ z \in [x] \Rightarrow z \leq\ maxl\ [x]) \\ \equiv & \quad \{ \text{ primo pattern di } \texttt{maxl}, \text{ Singoletto } \} \\ & x \in [x] \wedge x \leq\ x \\ \equiv & \quad \{ \text{ Singoletto, calcolo } \} \\ & \mathbf{t} \end{array}$$

Caso induttivo. Consideriamo ora una generica lista $x :: y :: xs$ e dimostriamo la proprietà per casi, a seconda che $x \leq y$ ovvero $x > y$.

$$\begin{array}{ll} & maxl\ x :: y :: xs \in x :: y :: xs \wedge (\forall z.\ z \in x :: y :: xs \Rightarrow z \leq\ maxl\ x :: y :: xs) \\ \equiv & \quad \{ \textbf{Ip: } x \leq y, \text{ secondo pattern di } \texttt{maxl} \} \\ & maxl\ y :: xs \in x :: y :: xs \wedge (\forall z.\ z \in x :: y :: xs \Rightarrow z \leq\ maxl\ y :: xs) \\ \equiv & \quad \{ \text{ proprietà delle liste } \} \\ & ((maxl\ y :: xs \in y :: xs) \vee (maxl\ y :: xs = x)) \wedge \\ & \quad (x \leq maxl\ y :: xs) \wedge (\forall z.\ z \in y :: xs \Rightarrow z \leq\ maxl\ y :: xs) \\ \equiv & \quad \{ \text{ Ipotesi induttiva } \} \\ & (\mathbf{t} \vee (maxl\ y :: xs = x))\ \wedge\ (x \leq maxl\ y :: xs) \wedge \mathbf{t}) \\ \equiv & \quad \{ \text{ Zero, Unità } \} \\ & (x \leq maxl\ y :: xs) \\ \equiv & \quad \{ \text{ per ipotesi induttiva } y \leq maxl\ y :: xs, \textbf{ Ip: } x \leq y, \text{ transitività di } \leq \} \\ & \mathbf{t} \end{array}$$

Il caso $x > y$ è analogo e lasciato per esercizio. □

Esercizi

1 Si riesaminino gli esempi delle sezioni precedenti e per ognuno di essi si stabiliscano formalmente il dominio S e la relazione di precedenza $\sqsubset$. Si mostri che la coppia $(S, \sqsubset)$ rappresenta un insieme ben fondato.

2 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 nonvuoto di D ammette un minimale. (suggerimento: si dimostri che se esistesse una catena discendente infinita l'insieme generato da tale catena non potrebbe avere un minimale. Viceversa...)

3 Si modifichi la funzione `pari` in modo che sia definita anche su interi negativi.

4 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 $s_1\hat{}s = s_2\hat{}s$.

5 Siano $(S_1, \sqsubset_1)$, $(S_2, \sqsubset_2)$ due insiemi ben fondati. Si dimostri che l'insieme $(S_1 \times S_2, \sqsubset)$, con la relazione

$$(s_1, s_2) \sqsubset (s_1', s_2') \iff s_1 \sqsubset_1 s_1' \text{ e } s_2 \sqsubset_2 s_2'$$

è ben fondato.

6 Si definisca la funzione *gcd*, che calcola il massimo comun divisore tra due interi. Suggerimento: si sfruttino le seguenti proprietà euclidee:

$$gcd(x, x) = x$$
$$gcd(x, y) = gcd(y, x)$$
$$gcd(x, y) = gcd(x, y - x)$$

4.7 Tipi Derivati

Nei capitoli precedenti abbiamo visto tre modi fondamentali per comporre tipi elementari: tramite applicazione funzionale (`->`), costruzione di tuple (`*`), o tramite costruzione di liste (`list`). È possibile anche definire direttamente nuovi tipi. Ad esempio, possiamo definire il tipo "segno" come positivo o negativo:

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

e costruire espressioni su questo nuovo tipo:

```
#let segnointero n = if n>=0 then Positivo else Negativo;;
segnointero : int -> segno = <fun>
#segnointero 1;;
- : segno = Positivo
```

Tali tipi di dato si dicono *enumerati*, poiché sono definiti enumerando esplicitamente i valori del tipo. Ad esempio, il tipo "giorno" è definito enumerando i possibili valori che un espressione può assumere:

```
#type giorno =
     Lunedi | Martedi | Mercoledi | Giovedi | Venerdi | 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 Giovedi;;
- : bool = true
```

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

- stringhe (nomi di individui), oppure
- interi (codifica del 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 identification = Name of string
                     | Code of int;;
Type identification defined.
```

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

```
#Name;;
- : string -> identification = <fun>
#Code;;
- : int -> identification = <fun>
#Name "Giuseppe";;
- : identification = Name "Giuseppe"
#let id1= Name "Giuseppe";;
id1 : identification = Name "Giuseppe"
#let id2 = Code 123456;;
id2 : identification = Code 123456
```

`id1` e `id2` appartengono entrambi allo stesso tipo `identification`, pur essendo costruiti a partire da due tipi diversi. Si noti che due valori sono distinti anche se i vari casi che li contraddistinguono sono costruiti a partire dagli stessi tipi. Si consideri ad esempio il seguente tipo, che specifica i modi possibili di 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 la definizione di 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
```

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 del *cons* 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). Il tipo `Nat` può essere introdotto esplicitamente nel seguente modo:

```
#type Nat = Zero |Succ of Nat;;
Type Nat defined.
```

definendolo, cioè, come 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 ricorsivamente:

$$\begin{aligned} &Zero \\ &Succ\ Zero \\ &Succ\ (Succ\ Zero) \\ &Succ\ (Succ\ (Succ\ Zero)) \\ &\vdots \end{aligned}$$

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 $y = Succ\ x$. Ad esempio, $Zero < Succ\ Zero$ poiché $Succ\ Zero$ è strutturalmente "più ricco" di $Zero$ (ossia, è ricavato da $Zero$ stesso).

La possibilità di definire tipi ricorsivi è un meccanismo molto potente, poiché permette di modellare molte entità matematiche. Ad esempio, posso definire le espressioni aritmetiche come tipo ricorsivo:

```
#type aop = Add | Sub | Mul | Div;;
Type aop defined.
#type aexp = Num of int | Exp of aexp * aop * aexp;;
Type aexp defined.
```

Un'espressione aritmetica è un numero, oppure è l'applicazione di un operatore aritmetico a due espressioni aritmetiche. Ad esempio, l'espressione $3 + 4$ può essere rappresentata da

```
#Exp (Num 3, Add, Num 4);;
- : aexp = Exp (Num 3, Add, Num 4)
```

A partire dalla definizione di `Aexp` possiamo definire il meccanismo di valutazione, calcolando così il valore associato ad un'espressione (la sua forma normale).

```
#let val op = match op with
              Add -> prefix +
```

```
            | Mul -> prefix *
            | Div -> prefix /
            | Sub -> prefix -;;
val : aop -> int -> int -> int = <fun>
#let rec evalint e = match e with
                     Num n -> n
                   | Exp (e1, op, e2) ->val op (evalint e1) (evalint e2);;
evalint : aexp -> int = <fun>
```

Il risultato della valutazione di un valore $Num\ n$ è n; per valutare invece un'espressione composta, bisogna innanzitutto valutare le sottoespressioni di cui è composta, e quindi applicare l'operazione associata all'operatore di composizione. Si noti che il risultato della valutazione è di tipo `int`; per restituire la forma normale dell'espressione, basta ricostruire, a partire dal risultato della valutazione, il valore *Aexp*:

```
let eval e = Num (evalint e);;
eval : aexp -> aexp = <fun>
```

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` su `list`:

```
#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.

Un Esempio: 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, ciasscun 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 4.1.

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.

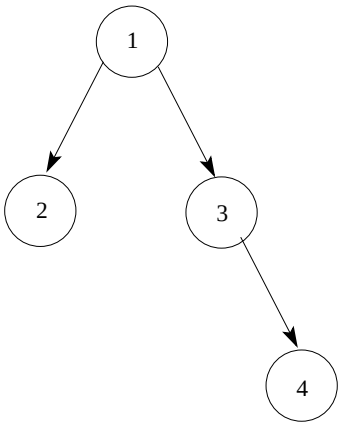

Figura 4.1: Un esempio di albero binario

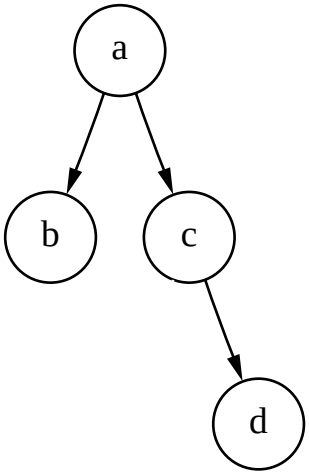

Figura 4.2: un albero binario con etichette di tipo carattere

L'informazione che etichetta un nodo può essere di qualsiasi tipo: ad esempio, l'albero di Figura 4.1 può essere etichettato con caratteri, come in Figura 4.2.

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 Caml 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 btree =
   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. Ad esempio, la seguente funzione seleziona il figlio sinistro di un albero binario:

```
#let left bt = match bt with
               Empty           -> Empty
             | Node (x, l, r)  -> l;;
left : 'a btree -> 'a btree = <fun>
```

La ricerca di un elemento in un albero binario può essere fatta molto agevolmente, utilizzando 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 btree -> bool = <fun>
```

L'albero vuoto non contiene nessun elemento, e quindi la ricerca in un albero vuoto restituisce `false`. Se, al contrario, la ricerca viene effettuata in un un albero non vuoto, ci sono due casi da esaminare. Se la radice dell'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.

Si noti che la ricerca permette di esaminare un 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 bt) + (count rt);;
count : 'a btree -> int = <fun>
```

È lasciato per esercizio al lettore dimostrare per induzione strutturale che la funzione `count bt` restituisce esattamente il numero di nodi dell'albero.

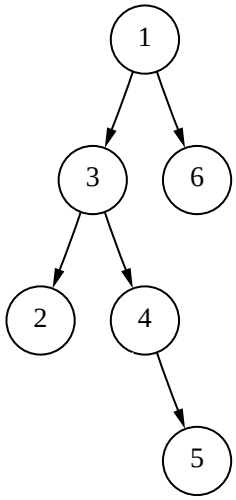

Figura 4.3: Un albero binario

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 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 posticipata o postfissa).

Ad esempio, nell'albero binario di Figura 4.3 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 $[2; 3; 4; 5; 1; 6]$;
- la visita posticipata permette di visitare i nodi nell'ordine $[2; 5; 4; 3; 6; 1]$;

Una semplice funzione che permette di definire la lista derivata dalla visita anticipata è la seguente:

```
#let rec prelim bt = match bt with
                       Void             -> []
                     | Node (x, lt, rt) -> x::(prelim lt @ prelim rt);;
prelim : 'a btree -> 'a list = <fun>
```

Si supponga che le funzioni $left\ x\ bt$ e $right\ x\ bt$ restituiscano i sottoalberi sinistro e destro del sottoalbero di bt la cui radice è etichettata con x, (da definire in Caml per esercizio), e si

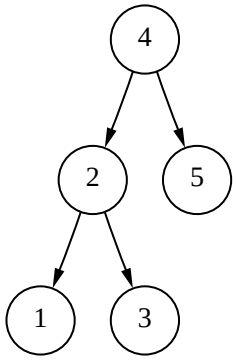

Figura 4.4: Un albero binario di ricerca

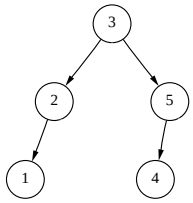

Figura 4.5: Un altro albero binario di ricerca

consideri la proprietà di un generico albero bt:

$$\forall x \in bt.\ (\forall y \in left\ x\ bt.\ y \leq x) \wedge (\forall y \in right\ x\ bt.\ x \leq y)$$

Gli alberi che godono di questa proprietà sono 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 ($\forall z \in right\ y\ bt.y \leq z$) possiamo dedurre che $\forall z \in right\ y\ bt.x < z$, e quindi possiamo evitare di effettuare la ricerca nel sottoalbero destro 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 di ricerca di Figura 4.4 e Figura 4.5.

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 specifica Caml della 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
                       Void                          -> Node (x, Void, Void)
                     | 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
                []    -> Void
              | x::xs -> insord x (buildtree xs);;
buildtree : 'a list -> 'a btree = <fun>
```

Esercizi

1 Si definisca una funzione per testare se due temperature sono uguali. Si definisca un predicato per determinare quale tra due temperature è più fredda.

2 Si definiscano la moltiplicazione e l'esponente come operazioni su Nat. Si dimostri per induzione strutturale che tali definizioni sono corrette.

3 Si esprima una rappresentazione per i numeri interi.

4 Si definiscano le funzioni che permettano di esprimere la visita posticipata e simmetrica di un albero binario.

Capitolo 5

Alcuni Esempi di Programmazione Funzionale

In questo capitolo mostreremo una serie di esempi di programmi funzionali, che riguarderanno sia applicazioni numeriche che applicazioni simboliche.

5.1 Il Calcolo della Radice Quadrata

Vediamo come definire la funzione *sqrt* che, dato un reale, restituisce la sua radice quadrata. La specifica formale è data dalla seguente asserzione:

$$\forall x \geq 0. sqrt\ x \geq 0 \wedge (sqrt\ x)^2 = x$$

Si noti che la specifica non suggerisce alcun metodo per il calcolo di *sqrt*, e non fa alcuna assunzione sulla precisione aritmetica dell'operatore. Cerchiamo quindi di indebolire la specifica, percé sia più costruttiva: costruttiva:

$$\forall x \geq 0. sqrt\ x \geq 0 \wedge abs(sqrt\ x)^2 - x) < \epsilon$$

per un sufficientemente piccolo $\epsilon > 0$, scelto considerando la precisione (limitata) del calcolatore che dovrebbe eseguire la funzione. *abs* rappresenta la funzione "valore assoluto", definibile come

```
#let abs x = if x <.0.0 then -.x else x;;
abs : float -> float = <fun>
```

Vediamo cosa diventa la nostra specifica nel caso $\epsilon = 0.0001$ e $x = 2$. In tal caso avremo:

$$sqrt\ 2 \geq 0 \wedge abs(sqrt\ 2)^2 - 2) < 0.0001$$

Ora, notiamo che

$$(1.4141)^2 = 1.99967881$$

$$(1.4142)^2 = 1.99996164$$

$$(1.4143)^2 = 2.00024449$$

Il valore $sqrt\ 2 = 1.4142$ è quindi una possibile risposta.

Senza perderci nei dettagli matematici, accenniamo al metodo di Newton per calcolare le radici di una funzione $f(a)$: se y è un'approssimazione di una radice della funzione, allora

$$\overline{y} = y - \frac{f(y)}{f'(y)}$$

è un'approssimazione migliore di y, dove $f'(y)$ è la derivata di f calcolata sul punto y. Nel nostro caso, vogliamo calcolare $sqrt\ x = a$ tale che $a^2 - x = 0$. Praticamente, vogliamo trovare le radici della funzione $f(a) = a^2 - x$, da cui otteniamo la successione

$$\begin{aligned} y_0 &= x \\ y_{n+1} &= y_n - \frac{y_n^2 - x}{2y_n} \\ &= (y_n + \frac{x}{y_n})/2 \end{aligned}$$

e avremo ottenuto il valore voluto quando avremo raggiunto un n tale che

$$abs(y_n^2 - x) < \epsilon$$

Per esempio, nel caso $x = 2$ e $\epsilon = 0.0001$, otteniamo:

$$\begin{aligned} y_0 &= 2 \\ y_1 &= 1.5 \\ y_2 &= 1.4167 \\ y_3 &= 1.4142157 \end{aligned}$$

da cui si deduce che y_3 è un valore ammissibile.

Nel nostro esempio, abbiamo utilizzato tre componenti logiche distinte. Innanzitutto, la funzione che genera nuove approssimazioni a partire da un'approssimazione y:

```
#let improve x y = (y +. x/.y)/.2.0;;
improve : float -> float -> float = <fun>
```

In secondo luogo, vi è una condizione di terminazione, che testa se un'approssimazione è accettabile:

```
#let satis x y  = abs(y**2.0 -. x) <. eps;;
satis : float -> float -> float -> bool = <fun>
```

dove `eps` è un identificatore legato al valore scelto per ϵ. Infine, c'è l'idea generale di applicare una funzione f ad un valore iniziale finché una condizione p non si avvera:

```
#let rec until p f x = if (p x) then x else (until p f (f x));;
until : ('a -> bool) -> ('a -> 'a) -> 'a -> 'a = <fun>
```

Mettendo tutto insieme, otteniamo:

```
#let sqrt x = until (satis x) (improve x) x;;
sqrt : float -> float = <fun>
```

e, poiché le funzioni precedenti sono relative a `sqrt`, possiamo compattare la definizione in

```
#let sqrt x =
                let satis y = abs(y**2.0 -. x) <. eps
                        and improve y = (y +. x/.y)/.2.0
                        and until p f y = if (p y) then y else (until p f (f y))
                in until satis improve x;;
sqrt : float -> float = <fun>
#sqrt 2.0;;
- : float = 1.41421568627
```

Si noti come, rendendo locali le definizioni di `satis` e `improve`, si sia potuta semplificare la loro definizione.

La funzione `sqrt` è stata costruita mettendo insieme altre funzioni più semplici, in uno stile di programmazione chiamato *modulare*. In tale stile di programmazione, le definizioni sono fatte per mezzo di combinazioni di definizioni più semplici, più facili da capire e da modificare. Per illustrare ciò, mostriamo come sia possibile esprimere il metodo di Newton generalizzato, e non la sua istanza `sqrt`, direttamente. Per fare ciò, abbiamo bisogno di calcolare la derivata prima di una funzione, data dalla formula matematica

$$f'(x) = \lim_{dx \to 0} \frac{f(x + dx) - f(x)}{dx}$$

In Caml,

```
#let deriv f dx x = (f (x +. dx) -. f(x)) /. dx;;
deriv : (float -> float) -> float -> float -> float = <fun>
```

Dove `dx` rappresenta un valore abbastanza piccolo (ad esempio, vicino a ϵ). Con questa definizione possiamo esprimere il metodo di Newton come:

```
#let newton f =
                let satis y = abs(f y) <. eps
                        and improve y = y -. (f y /. deriv f eps y)
                        and until p f x = if (p x) then x else (until p f (f x))
                in until satis improve;;
newton : (float -> float) -> float -> float = <fun>
```

e da qui ridefinire `sqrt` come

```
#let sqrt x = let f y = y**2.0 -. x
                in newton f x;;
sqrt : float -> float = <fun>
```

Di più, poiché ci basiamo direttamente sul metodo di Newton, possiamo calcolare la radice cubica di un numero:

```
#let cubrt x =
                let f y = y**3.0 -. x
                in newton f x;;
cubrt : float -> float = <fun>
#cubrt 3.0;;
- : float = 1.44224965666
```

Esercizi

2 Utilizzando la definizione di `sqrt`, si ridefinisca la funzione `intSqrt` accennata nel capitolo 1.

3 Si definisca `cubrt` modificando direttamente le funzioni `satis` e `improve`.

5.2 Dai numeri alle parole

Molte volte si ha bisogno di scrivere i numeri come parole. Un classico esempio è dato dagli assegni bancari, il cui ammontare deve essere scritto con parole, oltre che con cifre. In questo paragrafo cercheremo di definire una funzione *convert* che, dato n, restituisce come risultato la lista dei caratteri corrispondenti alla formulazione italiana di n.

La specifica informale assume che noi sappiamo quale è la formulazione in lingua italiana del numero: a diversi numeri, infatti, vengono associate delle parole differenti derivanti da regole differenti. Ad esempio, il numero "33" si converte in "trentatre", ma il numero "13" si converte in "tredici", ed il numero "31" si traduce in "trentuno". Allo stesso modo, il numero "1103" si converte in "millecentotre" (e non in "mille cento zero tre"), mentre il numero "1113" si converte in "millecentotredici".

Possiamo provare a porci un problema più semplice: quello di effettuare la traduzione quando $0 < n < 100$. In tal caso n avrà soltanto due cifre decimali, e la definizione di `convert` dovrebbe risultare più semplice. Si tratta infatti di combinare la prima e la seconda cifra in maniera adeguata:

```
#let digit2 n = (n/10,n mod 10);;
digit2 : int -> int * int = <fun>
```

A questo punto si tratta di combinare insieme le rappresentazioni dei due numeri. A questo proposito utilizziamo delle liste contenenti i nomi italiani dei numeri:

```
#  let unita=["uno"; "due"; "tre";
                "quattro"; "cinque"; "sei";
                "sette"; "otto"; "nove"];;
unita : string list =
  ["uno"; "due"; "tre"; "quattro"; "cinque"; "sei"; "sette"; "otto"; "nove"]
# let decina= ["dieci"; "undici"; "dodici"; "tredici"; "quattordici";
                "quindici"; "sedici"; "diciassette"; "diciotto";"diciannove"];;
decina : string list =
  ["dieci"; "undici"; "dodici"; "tredici"; "quattordici"; "quindici";
    "sedici"; "diciassette"; "diciotto"; "diciannove"]
# let decine = [("venti","vent"); ("trenta","trent"); ("quaranta","quarant");
                ("cinquanta","cinquant"); ("sessanta","sessant");
                ("settanta","settant"); ("ottanta","ottant");("novanta","novant")];;
decine : (string * string) list =
  ["venti", "vent"; "trenta", "trent"; "quaranta", "quarant";
   "cinquanta", "cinquant"; "sessanta", "sessant"; "settanta", "settant";
   "ottanta", "ottant"; "novanta", "novant"]
```

La lista relativa all'identificatore `decine` contiene una coppia (ad esempio, (venti,vent)). Il secondo elemento della coppia è necessario per un corretto accoppiamento con il numero "1": ad esempio, il numero "21" viene tradotto in `"ventuno"`, mentre tutti gli altri cominciano per `"venti"`.

Possiamo quindi definire la funzione `combine2`, che permette di effettuare la combinazione a partire da una coppia di numeri (ottenuta come risultato della funzione `digit2`):

```
#let combine2 (n,m) = match n,m with
                0,u when u>0  -> nth unita (u-1)
              | 1,u           -> nth decina u
              | t,0           -> fst (nth decine (t-2))
              | t,1           -> snd (nth decine (t-2)) ^ "uno"
              | t,u           -> fst (nth decine (t-2)) ^ (nth unita (u-1));;
combine2 : int * int -> string = <fun>
```

In questa definizione, la funzione `nth` seleziona l'ennesimo elemento di una lista, mentre l'operatore `^` corrisponde all'operatore `@` di concatenazione, nel caso di stringhe.

La definizione della funzione `convert2` a questo punto è immediata:

```
# let convert2 n  = combine (digit2 n);;
convert2 : int -> string = <fun>
# convert2 1;;
- : string = "uno"
# convert2 16;;
- : string = "sedici"
# convert2 31;;
- : string = "trentuno"
# convert2 87;;
- : string = "ottantasette"
```

Proviamo adesso a trattare il caso $0 < n < 1000$, in cui il numero ha tre cifre decimali. In tal caso, basta concatenare alla stringa ottenuta tramite `convert2` la cifra riguardante le centinaia:

```
# let convert3 n =
         if n/100 = 0 then convert2 (n mod 100)
                      else
        if n/100 = 1 then "cento" ^ convert2 (n mod 100)
                      else
        (nth unita ((n/100)-1)) ^ "cento" ^ convert2(n mod 100) ;;
convert3 : int -> string = <fun>
# convert3 114;;
- : string = "centoquattordici"
# convert3 986;;
- : string = "novecentoottantasei"
```

A questo punto abbiamo tutti gli elementi per estendere la funzione a piacere. Nel caso $0 < n < 1.000.000$:

```
# let convert6 n =
         if n/1000 = 0 then  convert3 (n mod 1000)
                 else if n/1000 = 1 then "mille" ^ (convert3 (n mod 1000))
```

```
                        else (convert3 (n/1000)) ^ "mila" ^
        (convert3 (n mod 1000));;
# convert6 12;;
- : string = "dodici"
# convert6 421;;
- : string = "quattrocentoventuno"
# convert6 895436;;
- : string = "ottocentonovantacinquemilaquattrocentotrentasei"
```

Esercizi

1 Definire ricorsivamente la funzione `convert`.

2 Si adatti la funzione presentata al trattamento di numeri negativi.

3* Si scriva la funzione inversa di *convert*. Si scriva cioè la funzione che, data la rappresentazione tramite parole di un numero, restituisca la sua rappresentazione in cifre.

5.3 Aritmetica in Lunghezza Variabile

Nel capitolo 2 abbiamo detto che la rappresentazione dei numeri su un calcolatore è limitata, e di conseguenza non tutti i numeri possono essere rappresentati. In questa sezione cercheremo di definire le operazioni aritmetiche di base per degli interi di dimensione variabile.

Cominciamo col considerare interi non negativi. Un intero non negativo verrà rappresentato come una lista non vuota di numeri, espressi in qualche base b (ossia, ogni elemento x della lista sarà tale che $0 \leq x < b$). Così, un intero x è rappresentato da una lista $[x_{n-1}; x_{n-2}; \ldots; x_0]$, tali che

$$x = \sum_{k=0}^{n-1} x_k b^k$$

Ad esempio, se scegliamo $b = 10000$, allora il numero 123456789 può essere rappresentato dalla lista $[1, 2345, 6789]$. Si noti che, per qualsiasi scelta di b, il numero 0 è comunque rappresentato dalla lista $[0]$.

Il primo passo è permettere una normalizzazione della rappresentazione. Infatti, il numero 123456789 può essere rappresentato anche dalla lista $[0, 0, 1, 2345, 6789]$, la cui differenza fondamentale con la rappresentazione precedente consiste nella presenza degli zeri nella parte meno significativa. Una funzione che permette l'eliminazione degli zeri non significativi è la seguente:

```
let strep l = match l with
                  [] -> [0] |
                  xs -> dropwhile ((=) 0) xs
```

Si noti che, se due numeri non sono normalizzati, non è possibile effettuare il confronto: le liste $[0, 1, 2]$ e $[1, 2]$ rappresentano lo stesso numero, ma un loro confronto darebbe esito negativo.

Addizione e Sottrazione

Le funzioni `vadd` e `vsub` possono essere definite in maniera molto semplice. I passi da seguire sono: allineare i due numeri (ovvero, rendere entrambe le liste della stessa lunghezza, aggiungendo degli zeri in testa alla lista più piccola), svolgere l'operazione richiesta su ogni cifra e "normalizzare" il risultato. Supponiamo per esempio di voler sommare i numeri rappresentati dalle liste $[4;6;5]$ e $[8;6;1]$. Il risultato della somma "cifra per cifra" è $[12;12;6]$. Tuttavia, se prendiamo come base $b = 10$, allora tale lista non rappresenta un numero valido, poiché le prime due cifre sono più grandi di 10. Normalizzando tale lista, otteniamo $[1,4,3,6]$. Tale risultato è ottenibile riducendo ogni cifra al resto della divisione per b, dopo aver aggiunto il valore avanzato dalla normalizzazione della precedente cifra. Proviamo a descrivere queste funzioni.

Innanzitutto abbiamo bisogno di descrivere il processo di mormalizzazione:

```
let align xs ys = let  m = length xs - length ys
                  in let rec copy x n = if n=0 then []
                                                else x::copy x (n-1)
                     in if m <0 then ((copy 0 (-m)) @ xs, ys)
                                else (xs, (copy 0 m) @ ys)
```

In secondo luogo abbiamo bisogno di definire il processo di normalizzazione relativo ad ogni cifra:

```
let carry x l = match l with  c::xs -> (x+c)/ b :: (x+c) mod b :: xs
```

La funzione `carry` normalizza la cifra `c` relativa ad una determinata posizione in una lista. Per normalizzare l'intera lista $[x_1; x_2; \ldots x_n]$, dobbiamo calcolare

$$carry\ x_1\ (carry\ x_2\ \ldots (carry\ x_n\ [0]))$$

e convertire il risultato. Si noti che tale operazione corrisponde ad un'operazione di fold. Possiamo quindi definire la normalizzazione come

```
let norm = compose strep (fold_right carry [0])
```

A questo punto abbiamo tutti gli elementi per calcolare le operazioni volute:

```
let vadd xs ys =  norm (map (uncurry (+)) (combine (align xs ys)))

let vsub xs ys =  norm (map (uncurry (-)) (combine (align xs ys)))
```

Si noti che il risultato di una sottrazione tra due numeri ha ulteriore bisogno di essere normalizzata, come il seguente esempio mostra:

```
# vsub [1;0;6] [3;7;5];;
- : int list = [-2; -7; 1]
```

La lista `[-2; -7; 1]` rappresenta il numero negativo -269. La situazione ideale sarebbe quella di poter rappresentare un numero negativo tramite un insieme di numeri negativi: `[-2; -6; -9]`, per esempio, può rappresentare il numero -269, e la sua rappresentazione è simmetrica alla rappresentazione di 269.

moltiplicazione

Una volta definite l'addizione e la sottrazione, possiamo definire la moltiplicazione `vmul` tra due rappresentazioni xs e ys. La soluzione più semplice è quella di applicare il metodo di moltiplicare xs per ogni cifra si ys ottenendo una serie di prodotti parziali da sommare tra loro. La lista dei prodotti parziali può essere ottenuta tramite l'operatore `map`:

```
let pprod xs ys = let bmul zs y = norm (map (( * ) y) zs)
                  in map (bmul xs) ys
```

Si noti ora che la somma tra i prodotti parziali deve essere fatta tenendo conto della "posizione" dei prodotti parziali. Ad esempio, supponendo $b = 10$,

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

Ora, il prodotto 132×12 è ottenuto sommando i due prodotti parziali 132 e 264, ma spostando 132 di una posizione: $132 \times 12 = 1320 + 264 = 1584$. Lo spostamento di una posizione nella nostra rappresentazione può essere ottenuto concatenando $[0]$ in fondo alla lista che rappresenta il numero. Infatti:

```
# vadd ([1;3;2]@[0]) [2;6;4];;
- : int list = [1; 5; 8; 4]
```

A questo punto siamo in grado di esprimere il prodotto:

```
let vmul xs ys = let addsh us vs = vadd (us @ [0]) vs
                 in foldr (addsh) (pprod xs ys)
```

Esercizi

1 Si supponga di voler rappresentare un numero negativo x con una lista di numeri $[x_{n-1}; x_{n-2}; \ldots; x_0]$ tale che $-b < x_i \leq 0$ per ogni i, e

$$x = \sum_{k=0}^{n-1} x_k b^k$$

. Si ridefinisca la funzione di normalizzazione in modo da permettere la corretta rappresentazione delle operazioni di addizione e sottrazione.

2 Si definisca una funzione `absint` che prende in ingresso la rappresentazione in base b di un numero e restituisce il numero stesso. Si dimostri poi che

$$\begin{aligned} absint\ ([0] \mathbin{@} xs) &= absint\ xs \\ absint\ (strep\ xs) &= absint\ xs \end{aligned}$$

3 Perché nella definizione di `vmul` si utilizza `foldr` invece di `fold_right`?

4 Si modifichi la definizione di `vmul` in modo che lavori con numeri sia positivi che negativi.

5.4 Rappresentazione e Manipolazione di Testi

Un testo può essere rappresentato in molti modi differenti. Ad esempio, possiamo definire un testo come una collezione di caratteri (e quindi con tipo `char list`), o, diversamente, come una collezione di linee, dove ogni linea è una collezione di caratteri che non contiene il carattere speciale '`\n`'. (e quindi con tipo `char list list`). Tali rappresentazioni sono equivalenti, e la scelta di una o dell'altra dipende dal tipo di manipolazione che si desidera fare sul testo stesso.

Supponiamo quindi di voler definire una funzione `lines` che data una lista di caratteri, restituisce una lista di linee. Tale funzione dovrebbe essere in grado di riconoscere il carattere '`\n`' come separatore tra due linee, e spezzettare un testo in base a tale carattere.

Possiamo vedere la funzione `lines` come l'inversa della funzione `unlines`, che inserisce il carattere '`\n`' tra due linee adiacenti e concatena il risultato. La definizione di quest'ultima è relativamente semplice:

```
let unlines = let  compact xs ys = xs @ [\verb"'\n'"] @ ys
              in  foldr compact
```

A questo punto la specifica di `lines` diventa

$$lines\ (unlines\ xss) = xss$$

Per ogni sequenza non vuota xss.

Come possibile definizione di *lines*, cercheremo di indagare un'espressione del tipo $fold\_rightfa$, per un'opportuna funzione f ed un opportuno valore iniziale a. Tale scelta è suggerita dal fatto che *unlines* utilizza una funzione simile a $fold\_right$.

Si noti innanzitutto che, poiché

$$\begin{aligned} foldr\ g\ [x] &= x && (5.1)\\ foldr\ g\ ([x] \mathbin{@} xs) &= g\ x(foldr\ f\ xs) && (5.2)\end{aligned}$$

$$\begin{aligned} fold\_right\ g\ a\ [] &= a && (5.3)\\ fold\_right\ g\ ([x] \mathbin{@} xs) &= g\ x\ (fold\_right\ f\ a\ xs) && (5.4)\end{aligned}$$

possiamo specificare i vari patterns nelle definizioni di *lines* e *unlines*:

$$\begin{aligned} & unlines\ [xs] \\ = & \quad \{ \text{proprietà 5.1} \} \\ & xs \end{aligned}$$

$$\begin{aligned} & unlines\ ([xs] \mathbin{@} xss) \\ = & \quad \{ \text{proprietà 5.2} \} \\ & compact\ xs\ unlinesxss \end{aligned}$$

$$\begin{aligned} & lines\ [] \\ = & \quad \{ \text{proprietà 5.3} \} \\ & a \end{aligned}$$

$$\begin{aligned} & lines\ ([x] \mathbin{@} xs) \\ = & \quad \{ \text{proprietà 5.4} \} \\ & f\ x\ lines\ xs \end{aligned}$$

D'altra parte,

$$
\begin{array}{ll}
 & a \\
= & \quad \{ \text{ proprietà 5.3 } \} \\
 & \mathit{lines}\ [] \\
= & \quad \{ \text{ proprietà 5.1 } \} \\
 & \mathit{lines}\ (\mathit{unlines}\ [[\ ]]) \\
= & \quad \{ \text{ specifica } \} \\
 & [[\ ]]
\end{array}
$$

Abbiamo quindi trovato un valore per a che fa al nostro caso. Per quanto riguarda f, si noti che

$$
\begin{array}{ll}
 & f\ x\ xss \\
= & \quad \{ \text{ specifica } \} \\
 & f\ x\ (\mathit{lines}\ (\mathit{unlines}\ xss)) \\
= & \quad \{ \text{ proprietà } 5.4 \} \\
 & \mathit{lines}\ ([x] \mathbin{@} \mathit{unlines}\ xss)
\end{array}
$$

D'altra parte, se $x =$ `'\n'`,

$$
\begin{array}{ll}
 & \mathit{lines}\ (\texttt{'\textbackslash n'} : \mathit{unlines}\ xss) \\
= & \quad \{ \text{ definizione di } \texttt{@} \} \\
 & \mathit{lines}\ ([\ ] \mathbin{@} \texttt{'\textbackslash n'} : \mathit{unlines}\ xss) \\
= & \quad \{ \text{ definizione di } \texttt{compact} \} \\
 & \mathit{lines}\ (\mathit{compact}\ []\ \mathit{unlines}\ xss) \\
= & \quad \{ \text{ proprietà } 5.2 \} \\
 & [[\ ]] \mathbin{@} xss
\end{array}
$$

Se invece $x \neq$ `'\n'`, si noti che, poiché xss non può essere vuota, possiamo scrivere supporre $xss = [ys] \mathbin{@} yss$. A questo punto otteniamo:

$$
\begin{array}{ll}
 & \mathit{lines}\ ([x] \mathbin{@} \mathit{unlines}\ ([ys] \mathbin{@} yss)) \\
= & \quad \{ \text{ proprietà } 5.2 \} \\
 & \mathit{lines}\ ([x] \mathbin{@} (ys \mathbin{@} \texttt{'\textbackslash n'} : \mathit{unlines}\ yss)) \\
= & \quad \{ \text{ @ è associativa } \} \\
 & \mathit{lines}\ (([x] \mathbin{@} ys) \mathbin{@} \texttt{'\textbackslash n'} : \mathit{unlines}\ yss \\
= & \quad \{ \text{ definizione di } \texttt{compact} \} \\
 & \mathit{lines}\ (\mathit{compact}\ ([x] \mathbin{@} ys)\ \mathit{unlines}\ yss) \\
= & \quad \{ \text{ proprietà } 5.2 \} \\
 & \mathit{lines}\ (\mathit{unlines}\ ([[x] \mathbin{@} ys] \mathbin{@} yss) \\
= & \quad \{ \text{ specifica } \} \\
 & [[x] \mathbin{@} ys] \mathbin{@} yss
\end{array}
$$

Da ciò possiamo dedurre che

$$f\ x\ xss = [[x] \mathbin{@} hd\ xss] \mathbin{@} tl\ xss$$

Mettendo tutto insieme otteniamo:

```
let lines = let f x xss = if x='\n' then [[]] @ xss
                                    else [[x] @ hd xss] @ tl xss
            in fold_right f [[]]
```

Esercizi

1 Si consideri la suddivisione di ogni linea in parole. In pratica, una linea può essere rappresentata come una lista di caratteri o, equivalentemente, come una lista di liste di caratteri tra i quali non è presente lo spazio. Si definiscano le funzioni `word` e`unword` che effettuano la trasformazione tra le due rappresentazioni. (Suggerimento: si definisca innanzitutto una funzione che effettui la normalizzazione di una linea, eliminando i caratteri di spazio superflui tra due parole. Quindi...)

2 Si estenda il problema del trattamento dei testi sviluppanodo una funzione che effettua la giustificazione di un testo (ovvero, renda tutte le linee della stessa lunghezza), aggiungendo opportunamente degli spazi tra le parole contenute nelle linee.