ML for Dummies

Luca Abeni

March 9, 2017

1 Introduzione

Mentre un linguaggio di tipo imperativo rispecchia l'architettura di Von Neumann, descrivendo i programmi come sequenze di istruzioni che modificano il contenuto di locazioni di memoria (identificate da variabili), un linguaggio funzionale codifica i programmi come espressioni che vengono valutate, ritornando valori. Come conseguenza, non c'è più un riferimento diretto all'architettura di Von Neumann ed il concetto stesso di variabili con valore modificabile non esiste più.

Come detto, i programmi scritti in un linguaggio funzionale vengono eseguiti valutando espressioni. Informalmente parlando, esistono espressioni "complesse", che possono essere semplificate, ed espressioni "semplici", vale a dire non semplificabili. Un'espressione semplice e' un *valore*, mentre un'espressione complessa può essere semplificata in un valore; l'operazione che calcola tale valore è detta *valutazione* dell'espressione.

Un'espressione complessa è quindi composta da operazioni (o funzioni) applicate a valori. In caso di "eager evaluation", la valutarzione dell'espressione è effettuata valutando prima i parametri di ogni operazione e quindi applicando l'operazione ai valori ottenuti.

Per esempio, $4 * 3$ è un'espressione "complessa" composta dai valori 4 e 3 e dall'operazione di moltiplicazione. Valutando tale espressione, si semplifica a 12, che è il suo valore.

Riassumendo, un programma scritto in un linguaggio funzionale non è niente altro che un'espressione, che viene valutata quando il programma esegue. In teoria, la valutazione di questa espressione non dovrebbe avere effetti collaterali, ma alcuni effetti collaterali (per esempio, input ed output) sono spesso molto difficili da evitare. Programmi complessi sono spesso scritti definendo funzioni (nel senso matematico del termine) che vengono poi invocate dall'espressione "principale" che descrive il programma stesso.

Le varie espressioni che compongono un programma sono scritte come operazioni o funzioni che agiscono su *valori appartenenti ad un tipo*. Le funzioni stesse sono caratterizzate da un tipo (il tipo dei valori risultanti)[1]. In un linguaggio a tipizzazione statica, il tipo di un'espressione e dei suoi parametri è determinato a tempo di compilazione o comunque (in caso di interprete) prima di eseguire il codice: il compilatore/inteprete estrapola i tipi di parametri ed espressione analizzando il codice, senza eseguirlo.

Nei casi in cui il tipo dei valori e delle espressioni non possa essere facilmente estrapolato dal compilatore (o dall'interprete), linguaggi come ML possono permettere di annotare le espressioni con il loro tipo, usando un'apposita sintassi (vedi sotto).

ML utilizza un meccanismo di "eager evaluation", quindi, un'espressione aritmetica è valutata valutando prima i suoi argomenti e quindi applicando le operazioni dell'espressione ai valori ottenuti valutando gli argomenti.

2 Tipi ed Espressioni in ML

I tipi di dato base forniti da ML sono: `unit`, `bool`, `int`, `real`, `char` e `string`; oltre a fornire questi tipi di base, ML permette anche di combinarli usando delle *tuple*, di definire dei sinonimi per tipi di dato esistenti e di definire nuovi tipi di dato (i cui valori sono generati da apposite funzioi chiamate costruttori).

Il tipo `unit` è composto da un unico valore, `()` e viene utilizzato come tipo di ritorno per espressioni che non ritornano alcun valore (e che sono importanti solo per i propri effetti collaterali).

Il tipo `bool`, invece, è composto da due valori (`true` e `false`).

Il tipo `int` è composto (come il nome suggerisce) dai numeri interi, positivi e negativi. Su tali numeri è definito l'operatore `~`, che nega il segno di un numero; quindi, per esempio, `~3` rappresenta il numero

[1] formalmente, un tipo può essere definito come un insieme di valori ed il tipo di un valore indica l'insieme a cui tale valore appartiene

−3. Altri operatori definiti sul tipo `int` sono le operazioni aritmetiche di base `*`, `+` e `-`[2]. L'operatore `/` di divisione non è definito sugli interi (mentre esiste la divisione intera `div`).

Il tipo `real` è composto da un insieme di approssimazioni di numeri reali, che possono essere espressi tramite parte intera e parte frazionaria (per esempio, `3.14`) o usando la forma esponenziale (per esempio, `314e~2`). Ancora, il simbolo `~` può essere usato per negare un numero (invertirne il segno). Due valori speciali `NaN` (Not a Number) e `inf` possono essere usati per indicare valori non rappresentabili come numeri reali (per esempio, la radice quadrata di un numero negativo) o valori infiniti (il risultato della divisione di un numero reale per 0). È infine da notare che i valori di tipo `real` sono confrontabili con gli operatori <, >, <= e >= (che vengono valutati a valori di tipo `bool`), ma non si può usare il testi di uguaglianza su di essi.

Il tipo `char` è composto dall'insieme dei caratteri. Un valore di tale tipo viene rappresentato con il prefisso `#` e fra virgolette; per esempio, `#"a"`.

Il tipo `string` è composto dall'insieme delle stringhe, rappresentate fra virgolette; per esempio `"test"`. Sulle stringhe, ML definisce l'operatore di concatenazione `^`: `"Ciao, " ^"mondo" = "Ciao, Mondo"`.

Va inoltre notato che ML non effettua conversioni automatiche di tipo. Quindi, espressioni tipo `5 + 2` (che viene valutata a 7, di tipo `int`) e `5.0 + 2.0` (che viene valutata a 7.0 di tipo `real`) sono corrette, ma l'espressione `5.0 + 2` genera un errore (somma fra un valore di tipo `real` ed un valore di tipo `int`). ML fornisce però varie operazioni per convertire valori fra i vari tipi; per esempio, `ord` converte un valore di tipo `char` in un valore di tipo `int` (che rappresenta il suo codice ASCII) e `chr` effettua l'operazione inversa.

Per finire, oltre ai "classici" operatori sui vari tipi di variabili ML fornisce un operatore di selezione `if`, che permette di valutare due diverse espressioni dipendentemente dal valore di un predicato. La sintassi di un'espressione `if` in ML è:

if <p> **then** <exp1> **else** <exp2>;

dove `<p>` è un predicato (espressione di tipo booleano) ed `<exp1>` e `<exp2>` sono due espressioni aventi lo stesso tipo (notare che `<exp1>` e `<exp2>` devono avere lo stesso tipo percé il valore dell'espressione `if` risultante ha lo stesso tipo di `<exp1>` e `<exp2>`). L'espressione `if` viene valutata come `<exp1>` se `<p>` è vero, mentre è valutata come `<exp2>` se `<p>` è falso.

Sebbene l'operatore `if` di ML possa essere considerato come l'equivalente a livello di espressione dell'operazione di selezione `if` fornita dai linguaggi imperativi, è importante notare alcune differenze. Per esempio, l'operazione di selezione di un linguaggio imperativo permette di eseguire un blocco di operazioni se il predicato è vero (ramo `then`) o un diverso blocco di operazioni se il predicato è falso (ramo `else`). In teoria, ciascuno dei due blocchi (`then` o `else`) può essere vuoto, a significare che non ci sono operazioni da eseguire per un determinato valore di verità del predicato. L'operatore `if` fornito da ML, invece (come l'equivalente operatore di tutti i linguaggi funzionali), deve *sempre* essere valutabile ad un valore. Quindi, nessuna delle due espressioni `then` o `else` può essere vuota.

Un esempio di utilizzo di `if` è

if a > b **then** a **else** b;

che implementa un'espressione valutata al massimo fra `a` e `b`.

3 Associare Nomi a Valori

Le espressioni che compongono un programma ML possono fare riferimento a valori o ad identificatori definiti in un *ambiente*, (un insieme di coppie (identificatore, valore)). L'ambiente può essere modificato (in realtà, esteso) associando un valore (di qualsiasi tipo) ad un nome (identificatore) tramite la keyword `val`:

val <name> = <value>;
val <name>:<**type**> = <value>;

Questo codice (chiamato *dichiarazione* in ML) definisce un legame tra l'identificaotore `<name>` ed il valore `<value>` (di tipo `<type>`). Il valore `<value>` può essere anche il risultato della valutazione di un'espressione; in questo caso, la dichiarazione assume la forma

val <name> = <expression>;

[2] notare che in ML esiste la differenza fra l'operazione `-` (sottrazione) e l'operatore unario `~` che inverte il segno di un numero.

per esempio, `val v = 10 / 2;`.

Notare che una dichiarazione in ML (introdotta dalla keyword `val`) può essere vista come una *dichiarazione di variabile*: per esempio, `val pi = 3.14` crea una variabile identificata dal nome `pi` e la lega al valore reale 3.14. Ma in ML le variabili sono semplicemente nomi per dei valori, non sono contenitori di valori modificabili. In altre parole, una variabile ha sempre un valore costante, non modificabile, ed una successiva dichiarazione `val pi = 3.1415` non modifica il valore della variabile `pi` ma crea una nuova variabile legata al nuovo valore 3.1415 e la associa al nome `pi`, "mascherando" l'associazione precedente. ML utilizza sempre l'ultimo valore che è stato associato ad un nome. Questo significa che la keyword `val` **modifica l'ambiente, non il valore di variabili**: `val` definisce sempre una nuova variabile (inizializzata col valore specificato) e crea un nuovo legame (fra il nome specificato e la variabile creata) nell'ambiente.

4 Funzioni

Un particolare tipo di dato che non è stato precedentemente citato ma costituisce una caratteristica fondamentale dei linguaggi funzionali è il tipo di dato *funzione*. Come suggerito dal nome, un valore di questo tipo è una funzione, intesa nel senso matematico del termine: una relazione che mappa ogni elemento di insieme dominio in uno ed un solo elemento di un insieme codominio. In un linguaggio funzionale, gli insiemi dominio e codominio sono definiti dai tipi del parametro e del valore generato. Vedendo le cose da un altro punto di vista, si potrebbe dire che una funzione può essere considerata come una *espressione parametrizzata*, vale a dire un'espressione il cui valore dipende dal valore di un parametro.

Si noti che considerare funzioni con un solo parametro non è riduttivo: il parametro può essere una n-upla di valori (quindi, l'insieme dominio è il prodotto cartesiano di n insiemi), oppure si può usare il meccanismo del *currying* (vedi Sezione 8) per ridurre funzioni con più parametri a funzioni con un solo parametro. Un'altra cosa importante da notare è che in base alla definizione data qui sopra una funzione ha l'unico effetto di calcolare un valore (risultato, o valore di ritorno) in base al valore del parametro. Non può quindi avere *effetti collaterali* di qualsiasi tipo (vale a dire, non può avere effetti che non siano nel valore di ritorno della funzione).

In ML, un valore di tipo funzione si esprime usando la keyword `fn` (equivalente a λ nel λ-calcolo). La sintassi è:

fn <param> => <expression>;

dove `<param>` è il nome del parametro formale (con eventualmente specificato il suo tipo) mentre `<expression>` è un'espressione ML valida. L'espressione **fn** x:t => exp quando valutata ha come risultato un valore di tipo funzione. In particolare, una funzione che accetta come parametro un valore di tipo `t`. Ogni volta che la funzione verrà applicata ad un valore (parametro attuale), tale valore verrà legato al nome `x` (parametro formale) per valutare l'espressione `exp`.

Per esempio,

fn n => n + 1;

è una funzione che incrementa un numero naturale (in questo caso, il parametro formale è `n` e l'espressione da valutare quando viene applicata la funzione è `n + 1`). Una funzione può essere applicata ad un valore facendo seguire il valore del parametro attuale alla funzione. Per esempio,

(**fn** n => n + 1) 5;

applica la funzione `fn n => n + 1` al valore 5 (le parentesi sono necessarie per indicare l'ordine di precedenza delle operazioni: prima si definisce la funzione e poi si applica al valore 5). Questo significa che il valore 5 (parametro attuale) viene legato al nome `n` (parametro formale) e poi viene valutata l'espressione `n + 1`, che fornisce il valore di ritorno della funzione. Il risultato di questa espressione è ovviamente 6.

Ovviamente, un valore di tipo funzione può essere associato ad un nome usando il costrutto `val` di ML. Per esempio, il seguente codice definirà una variabile (non modificabile, ricordate!) `incrementa` avente come valore una funzione da interi ad interi che somma 1 al valore passato come parametro (in altre parole, assocerà il nome `incrementa` a tale funzione):

val incrementa = **fn** n => n + 1;

Il tipo di questa variabile è `fn : int -> int` (equivalente alla dizione matematica $f : \mathcal{N} \to \mathcal{N}$). A questo punto è ovviamente possibile applicare la funzione ad un parametro attuale usandone il nome:

il risultato di `incrementa 5` è ovviamente 6. Associare nomi simbolici a funzioni (vale a dire: definire variabili di tipo funzione) è particolarmente utile quando la funzione viene usata più volte... Si consideri per esempio la differenza fra

```
(fn x => x+1) ((fn x => x+1) 2);
```

e

```
incrementa (incrementa 2);
```

Per finire, in ML esiste una sintassi semplificata per associare nomi a funzioni (questo si può anche vedere come definire variabili di tipo funzione o, in breve, definire funzioni):

```
fun <name>(<param>) = <expression>;
```

è equivalente a

```
val <name> = fn <param> => <expression>;
```

Per esempio, la funzione `incrementa` è definibile come

```
fun incrementa(n) = n + 1;
```

che appare più leggibile rispetto alla definizione basata su `val` e `fn` (la quale è più simile al λ-calcolo).

Notare infine che la keyword `fun` non introduce nuove funzionalità ma è soltanto *zucchero sintattico* per semplificare la scrittura di definizioni `val ... = fn ...`[3].

5 Definizione per Casi

Oltre a poter essere definita tramite un'espressione aritmetica che permette di calcolarne il valore (come appena visto), una funzione può anche essere definita "per casi", specificando esplicitamente il valore del risultato corrispondente con ogni valore del parametro. Questo può essere facilmente fatto utilizzando l'operatore `case`:

```
fn x => case x of
   <pattern_1> => <expression_1>
 | <pattern_2> => <expression_2>
 ...
 ...
 | <pattern_n> => <expression_n>;
```

dove l'espressione **case** x **of** p1 => e1 | p2 => e2 | ... prima valuta l'espressione `x`, poi confronta il valore ottenuto con i *pattern* specificati (`p1`, `p2`, etc...). Non appena il confronto ha successo (si verifica un *match*), viene valutata l'espressione corrispondente assegnando all'espressione `case` il valore risultante.

Un modo semplice per definire "per casi" una funzione è quindi quello di usare valori costanti come pattern per enumerare i possibili valori del parametro. Un esempio di funzione definita per casi (o per enumerazione) è:

```
val giorno = fn n => case n of
                1 => "Lunedi'"
              | 2 => "Martedi'"
              | 3 => "Mercoledi'"
              | 4 => "Giovedi'"
              | 5 => "Venerdi'"
              | 6 => "Sabato"
              | 7 => "Domenica"
              | _ => "Giorno non valido";
```

notare che lo strano pattern "_" è utilizzato per "catturare" tutti i casi non precedentemente enumerati (numeri interi minori di 1 o maggiori di 7).

Per quanto detto, l'operatore `case` sembra essere l'equivalente a livello di espressioni del comando `case` o `switch` (comando di selezione multipla) esistente in molti linguaggi imperativi. Esiste però un'importante differenza: l'espressione `case` di ML (come l'equivalente di molti linguaggi funzionali)

[3] In realtà le cose sono leggermente più complicate di questo, in caso di funzioni ricorsive o di funzioni a più parametri... Ma per il momento consideriamo `fun` come un semplice sinonimo di `val ... = fn ...`.

permette di utilizzare non solo pattern costanti (come visto nell'esempio precedente), ma anche pattern più complessi contenenti nomi o costrutti come tuple o simili. Nel confrontare il valore di un'espressione `x` con questi pattern "più complessi", ML utilizza un meccanismo di *pattern matching*. Se un pattern per esempio contiene dei nomi, quando ML confronta l'espressione con tale pattern può creare legami fra nomi e valori in modo che il confronto abbia successo. Per esempio, nel seguente codice

```
val f = fn a => case a of
                  0 => 1000.0
                | x => 1.0 / (real x);
```

se il parametro della funzione non è `0` si crea un binding fra `x` ed il valore di `a`. Pattern più complessi basati su coppie o tuple di valori possono essere invece usati per definire funzioni in più variabili:

```
val somma = fn (a, b) => a + b;
```

In questo caso, quando `somma` viene invocata il meccanismo di pattern matching viene usato per fare il binding fra il nome `a` ed il primo valore della coppia passata come parametro attuale e fra il nome `b` ed il secondo valore.

Tornando alla definizione di funzioni "per casi", si noti che l'espressione

```
fn x = case x of
   <pattern_1> => <expression_1>
 | <pattern_2> => <expression_2>
 ...
 ...
 | <pattern_n> => <expression_n>;
```

è equivalente a

```
fn <pattern_1> => <expression_1>
 | <pattern_2> => <expression_2>
 ...
 ...
 | <pattern_n> => <expression_n>;
```

Quest'ultima sintassi è talvolta più chiara, in quanto permette di specificare in modo ancora più esplicito il valore risultato della funzione per ogni valore del parametro. Più formalmente, ML cerca un match fra il valore del parametro attuale con cui la funzione viene invocata ed i vari pattern specificati nella definizione `<pattern_1>...<pattern_n>`. Se il parametro attuale matcha `<pattern_1>`, si considera la prima definizione e la funzione viene valutata come `<expression_1>`, se il parametro attuale matcha `<pattern_2>` la funzione viene valutata come `<expression_2>` e così via. Usando questa sintassi, la definizione della funzione `giorno` presentata in precedenza diventa:

```
val giorno = fn 1 => "Lunedi'"
              | 2 => "Martedi'"
              | 3 => "Mercoledi'"
              | 4 => "Giovedi'"
              | 5 => "Venerdi'"
              | 6 => "Sabato"
              | 7 => "Domenica"
              | _ => "Giorno non valido";
```

Ancora, è importante che i pattern `<expression_1>`, ... `<expression_n>` coprano tutti i possibili valori che la funzione può ricevere come parametri attuali. Per questo il simbolo speciale (_) permette di matchare tutti i valori non coperti dalle clausole precendenti.

Sebbene il meccanismo di pattern matching sia stato introdotto in questa sede per spiegare la definizione di funzioni per casi, è importante capire che questo è un meccanismo estremamente generico, che può essere utilizzato in molti altri contesti permettendo di fare molto più che definire funzioni per casi. Per esempio, il meccanismo di pattern matching può essere usato per legare nomi a valori indipendentemente dalla definizione di una funzione:

```
val (x, y) = (4, 5);
```

andrà a legare il simbolo `x` al valore `4` ed il simbolo `y` al valore `5`.

Volendo definire il concetto di pattern in modo un po' Più preciso, si può dire che un pattern può essere:

- un valore costante, che matcha solo con quello specifico valore;
- un *variable pattern* `<var>:<type>`, di tipo `<type>`, che matcha un qualsiasi valore di tipo `<type>`, dopo aver creato un binding fra `<var>` ed il valore;
- un tuple pattern `(<pattern_1>, <pattern_2>, ..., <pattern_n>)`, di tipo `<type_1> * <type_2> * ... * <type_n>`, che compone n pattern più semplici in una tupla. In questo caso si ha matching con una tupla di n valori se e solo se ognuno di questi valori matcha col corrispondente pattern della tupla;
- il wildcard pattern _, che matcha qualsiasi valore.

Notare che il valore `()` è un esempio di tuple pattern (tupla nulla).

6 Ricorsione

Come noto, un programma scritto secondo il praradigma di programmazione funzionale utilizza il meccanismo della ricorsione per implementare algoritmi che verrebbero implementati tramite iterazione usando un paradigma imperativo. È quidni importante capire come implementare la ricorsione usando ML.

In teoria, in una funzione ricorsiva non è una funzione simile alle altre e non dovrebbe avere alcun bisogno di essere definita in modo particolare: semplicemnte, l'espressione che implementa il corpo della funzione richiama la funzione stessa. In realtà esistono però delle complicazioni dovute alla visibiltà degli identificatori: quando si usa `val` per associare un nome ad una funzione, il nome diventa visibile solo dopo che l'intera espressione `val ...` è stata completamente valutata. Quindi, non è ancora visibile quando dovrebbe essere usato dall'espressione che costituisce il corpo della funzione. Per questo motivo, un'espressione come

```
val fact = fn n => if n = 0 then 1 else n * fact (n - 1);
```

è destinata a fallire: il nome `fact` non è ancora visibile durante la definizione della funzione `fact`.

La soluzione a questo problema è usre `val rec` invece di `val`:

```
val rec fact = fn n => if n = 0 then 1 else n * fact (n - 1);
```

`val rec` aggiunge il legame fra il nome (`fact`, in questo caso) e la funzione che si sta definendo *prima* che l'espressione sia stata completamente valutata. Quindi, rende possibile chiamare ricorsivamente il nome dalla definizione del corpo della funzione.

Le espressioni `fun` rendono invece possibile definire funzioni ricorsive senza bisogno di ricorrere alla keyword `rec` o ad altri accorgimenti:

```
fun fact n = if n = 0 then 1 else n * fact (n - 1);
```

Quindi, quando nella Sezione 4 è stato scritto che **fun** <name>(<param>) = <expression> è equivalente a **val** <name> = **fn** <param> => <expression>; è stata commessa una lieve imprecisione: **fun** <name>(<param è in realtà equivalente a **val rec** <name> = **fn** <param> => <expression>;.

Per capire meglio il significato di `val rec` e la differenza fra `val` e `val rec`, si considerino le differenze fra

```
val f = fn n => n + 1;
val f = fn n => if n = 0 then 1 else n * f(n - 1);
```

e

```
val f = fn n => n + 1;
val rec f = fn n => if n = 0 then 1 else n * f(n - 1);
```

In particolare, qual'è il valore di `f 4` nei due casi?

7 Controllare l'Ambiente

Come molti dei moderni linguaggi, ML utilizza scoping statico: in una funzione, i simboli *non locali* sono risolti (vale a dire: associati ad un valore) facendo riferimento all'ambiente del blocco di codice in cui la funzione è definita (e non all'ambiente del chiamante).

Notare che la keyword `fn` (o la "sintassi abbellita" `fun`) crea un blocco di annidamento statico (come i simboli { e } in C). All'interno di questo blocco viene aggiunto un nuovo binding fra il simbolo che identifica il parametro formale ed il valore del parametro attuale con cui la funzione verrà invocata. Questo nuovo binding può introdurre un nuovo simbolo o mascherare un binding esistente. Per esempio, una definizione "**val** v = 1" crea un legame fra il simbolo "`v`" ed il valore "`1`" nell'ambiente globale. Una definizione "**val** f = **fn** v => 2 ∗ v" crea un blocco di annidamento (contenente l'espressione "2 ∗ v") dentro il quale il simbolo "`v`" non è più associato al valore "`1`", ma al valore del parametro attuale con cui "`f`" sarà invocata. Quindi, "`f 3`" ritornerà 6, non 2. come nel seguente esempio:

I simboli non locali (notare che per quanto detto fino ad ora i simboli locali sono solo i parametri della funzione) vengono cercati nell'ambiente attivo quando la definizione della funzione viene valutata e possono essere risolti in tale momento. Per esempio, una dichiarazione**val** f = **fn** x => x + y; risulterà in un errore se quando tale dichiarazione viene processata il simbolo `y` non è legato ad alcun valore.

Standard ML mette anche a disposizione due meccanismi per creare blocchi di annidamento statici e modificare l'ambiente al loro interno (senza che l'ambiente esterno al blocco risulti modificato): uno per creare blocchi di annidamento contenenti espressioni (**let** <dichiarazioni> **in** <espressione> **end**;) ed uno per modificare l'ambiente all'interno di una dichiarazione (**local** <dichiarazioni> **in** <dichiarazione> **end**;).

In altre parole, **let** <dichiarazioni> **in** <espressione> **end**; è un'espressione valutata al valore di `<espressione>` dopo che l'ambiente è stato modificato aggiungendo i binding definiti in `<dichiarazioni>`. Tali binding sono utilizzati per valutare l'espressione e sono poi rimossi dall'ambiente immediatamente dopo la valutazione. Per esempio, il costrutto `let` è utilizzabile per implementare una versione *tail recursive* della funzione fattoriale. Si ricordi che una funzione è tail recursive se utilizza solo chiamate ricorsive *in coda*: la tradizionale funzione **val rec** fact = **fn** n => **if** n = 0 **then** 1 **else** fact (n − 1) ∗ n; non è tail recursive perché il risultato di `fact(n - 1)` non è immediatamente ritornato, ma deve essere moltiplicato per `n`. Una versione tail recursive della funzione fattoriale utilizza un secondo parametro per memorizzare il risultato parziale: **val rec** fact_tr = **fn** n => **fn** res => **if** n = 0 **then** res **else** fact1 (n − 1) (n ∗ res);. Questa funzione riceve quindi due argomenti, a differenza della funzione `fact` originale. E' quindi necessario un wrapper che invochi `fact_tr` con i giusti parametri: **val** fact = **fn** n => fact1 n 1;. Una soluzione di questo genere ha però il problema che `fact_tr` è visibile non solo a `fact` (come dovrebbe essere), ma nell'intero ambiente globale. Come detto, il costrutto `let` permette di risolvere questo problema:

```
val fact = fn n =>
        let
            val rec fact_tr = fn n => fn res =>
                if n = 0 then
                    res
                else
                    fact_tr (n - 1) (n * res)
        in
            fact_tr n 1
        end;
```

local <dichiarazioni> **in** <dichiarazione> **end**; permette invece di utilizzare i binding definiti da `<dichiarazioni>` durante la definizione compresa fra `in` e `end`, ripristinando poi l'ambiente originario.

L'utilità di `local` può essere capita meglio considerando il seguente esempio: si supponga di voler implementare in ML una funzione $f : \mathcal{N} \to \mathcal{N}$, anche se ML non supporta il tipo `unsigned int` (corrispondente ad $\mathcal{N}$) ma solo il tipo `int` (corrispondente a $\mathcal{Z}$. Per sopperire a questa limitazione, si può definire una funzione `integer_f` che implementa $f()$ usando `int` come dominio e codominio, richiamandola da una funzione `f` che controlla se il valore del parametro è positivo o no, ritornando -1 in caso di argomento negativo:

```
local
    val integer_f = fn n => ...
  in
    val f = fn n => if n < 0 then ~1 else integer_f n
  end;
```

Questa soluzione permette di evitare di esportare a tutti la funzione `integer_n`, che accetta anche argomenti negativi senza fare controlli.

Analogamente, `local` può essere usato per "nascondere" la funzione `fact_tr` a due argomenti nella definizione tail recursive del fattoriale (vedere esempio precedente riguardo a `let`):

```
local
    val rec fact_tr = fn n => fn res =>
        if n = 0 then
            res
        else
            fact_tr (n - 1) (n * res)
in
    val fact = fn n => fact_tr n 1
end;
```

Confrontando i due esempi riguardanti il fattoriale tail recursive, è facile capire come esista una stretta relazione fra il costrutto `let` ed il costrutto `local` e come possa essere sempre possibile usare `let` al posto di `local` (spostando la clausola `val` dal blocco `in`...`end` all'esterno del costrutto).

8 Funzioni che Lavorano su Funzioni

Poiché un linguaggio funzionale fornisce il tipo di dato funzione e la possibilità di vedere funzioni come valori denotabili (gestiti in modo analogo a valori di altri tipi più "tradizionali" come interi e floating point), è possibile definire funzioni che accettano valori di tipo funzione come parametro e quindi lavorano su funzioni. In modo analogo, il valore di ritorno di una funzione può essere a sua volta di tipo funzione. Da questo punto di vista, le funzioni che accettano funzioni come parametri e ritornano valori di tipo funzioni non sono dissimili da funzioni che accettano e ritornano (per esempio) valori di tipo intero. Ci sono però alcuni dettagli che meritano di essere considerati con più attenzione e che motivano l'esistenza di questa sezione.

Per analizzare alcune peculiarità interessanti delle funzioni che lavorano su funzioni (spesso citate come "high-order functions" in letteratura) consideriamo un semplice esempio basato sul calcolo della derivata di una funzione $f : \mathcal{R} \to \mathcal{R}$. Cominciando definendo una funzion `derivata1` che accetta come parametro la funzione f di cui calcolare la derivata ed un numero $x \in \mathcal{R}$. La funzione `derivata1` ritorna il valore della funzione $f'()$ derivata di $f()$ (in realtá, dell rapporto incrementale sinistro di $f()$...) calcolata nel punto x. Tale funzione, che ha un parametro di tipo funzione ed uno di tipo `real` e ritorna un valore di tipo `real`, può essere implementata (o meglio, approssimata) in questo modo:

```
val derivata1 = fn (f, x) => (f(x) - f(x - 0.001)) / 0.001;
```

(si tralasci il fatto che la derivata è stata approssimata col rapporto incrementale sinistro con passo $\Delta = 0.001$).

È da notare come ML sia in grado di inferire il tipo dei parametri `x` (che risulta essere `real` a causa dell'espressione `x - 0.001`) ed `f` (che risulta essere `real -> real` poiché `f` viene applicata ad `x` ed il suo valore di ritorno viene diviso per `0.001`). Il tipo di `derivata1` sarà quindi `((real -> real) * real) -> real`.

L'esempio presentato non è comunque sorprendente, perché qualcosa di simile alla funzione `derivata1` presentata qui sopra si può facilmente implementare anche usando un linguaggio prevalentemente imperativo (per esempio, usando il linguaggio C si può usare un puntatore a funzione come primo parametro, invece di f). Una cosa invece notevolmente più difficile da implementare con linguaggi non funzionali è una funzione `derivata2` che riceva in ingresso solo la funzione $f()$ (e non il punto `x` in cui calcolare la derivata) e ritorni una funzione (e non un numero reale) che approssima la derivata di f. Questa funzione `derivata2` ha un unico parametro, di tipo `real -> real` e ritorna un valore di tipo `real -> real`. Il tipo di questa funzione sarà quindi `(real -> real) -> (real -> real)` Una sua possibile implementazione in ML è la seguente:

```
val derivata2 = fn f => (fn x => (f(x) - f(x - 0.001)) / 0.001);
```

Cerchiamo di capire meglio questa definizione di `derivata`: **val** derivata2 = **fn** f => sostanzialmente dice che il nome derivata è associato ad una funzione che ha come parametro "`f`". L'espressione che definisce come calcolare tale funzione è **fn** x => (f(x) − f(x − 0.001)) / 0.001 (le parentesi sono state aggiunte nell'esempio di qui sopra per aumentarne la leggibilità), che indica una funzione della variabile `x` calcolata come $\frac{f(x)-f(x-0.001)}{0.001}$. Quindi, il valore di ritorno della funzione `derivata2` è una funzione (del parametro `x`, che ML può identificare come `real` a causa dell'espressione $x - 0.001$). Tale funzione è calcolata in base alla funzione `f`, che è parametro di `derivata`. Valutando la definizione, ML può inferire il tipo di `f` (funzione da `real` a `real`: `real -> real`).

Come ulteriore considerazione, si può notare come la funzione `derivata2` possa essere vista come un'applicazione parziale della funzione `derivata1`. Sostanzialmente, invece di invocare la funzione passandole 2 argomenti (la funzione `f` di cui calcolare la derivata ed il punto `x` in cui calcolare la derivata), si invoca passando solo il primo argomento `f`... Se invocata con 2 argomenti, la funzione ritorna un numero reale (il valore della derivata di `f` nel punto `x`); quindi, se applicata ad un unico parametro `f` la funzione non può ritornare un reale, ma ritornerà un "oggetto" che può diverntare un numero reale quando applicato ad un ulteriore parametro `x` (di tipo reale)... Questo "oggetto" è quindi una funzione $f' : \mathcal{R} \to \mathcal{R}$.

Il procedimento usato per passare dalla prima versione di `derivata` alla seconda è in realtà del tutto generico ed è noto in letteratura col nome di *currying*. L'idea fondamentale del currying è (esprimendosi in modo informale) che una funzione di due parametri x e y è equivalente ad una funzione del parametro x che ritorna una funzione del parametro y. Quindi, il meccanismo del currying permette di esprimere una funzione in più variabili come una funzione in una variabile che ritorna una funzione delle altre variabili. Per esempio, una funzione $f : \mathcal{R}x\mathcal{R} \to \mathcal{R}$ che riceve due parametri reali e ritorna un numero reale può essere riscritta come $f_c : \mathcal{R} \to (\mathcal{R} \to \mathcal{R})$.

Per esempio, la funzione `somma` che somma due numeri

```
val somma => fn (x, y) => x + y;
```

può essere scritta tramite currying come

```
val sommac = fn x => (fn y => x + y);
```

(ancora una volta le parentesi sono state aggiunte nel tentativo di rendere la cosa più leggibile).

Per capire meglio tutto questo si può confrontare la funzione $f(x, y) = x^2 + y^2$ con la sua versione ottenuta attraverso il currying $f_c(x) = f_x(y) = x^2 + y^2$. Si noti che $f()$ ha dominio $\mathcal{R}^2$ e codominio $\mathcal{R}$, mentre $f_c()$ ha dominio $\mathcal{R}$ e codominio $\mathcal{R} \to \mathcal{R}$. In ML queste funzioni sono definite come

```
val f  = fn (x, y)      => x * x + y * y;
val fc = fn x => (fn y => x * x + y * y);
```

`fc` permette di fare applicazioni parziali, tipo `val g = f 5`, che definisce una funzione $g(y) = 25 + y^2$, mentre `f` non permette nulla di simile.

Il currying è direttamente supportato a livello sintattico da ML, che fornisce una sintassi semplificata per definire funzioni "curryficate" tramite l'espressione `fun`:

```
fun f a b = exp;
```

è equivalente (ricordare che `fun` è sostanzialmente zucchero sintattico) a

```
var rec f = fn a => fn b = exp;
```

Questa è un'altra piccola correzione rispetto alla definizione leggermente imprecisa di `fun` data nella Sezione 4 e corretta (sostituendo `val` con `val rec`) nella Sezione 6.

Usando la sintassi semplificata fornita da `fun`, le definizioni di `f` ed `fc` (esempio precedente) diventano quindi

```
fun f (x, y) = x * x + y * y;
fun fc x y   = x * x + y * y;
```

mentre la definizione di `derivata2` diventa

```
fun derivata2 f x = (f(x) - f(x - 0.001)) / 0.001;
```

9 Ancora sui Tipi di Dato

Fino ad ora abbiamo lavorato coi tipi di dato predefiniti di ML: `bool`, `int`, `real`, `char`, `string` e `unit` (anche se questo ultimo tipo non è stato molto utilizzato... In pratica, è utile prevalentemente per modellare funzioni "strane" che non accettano argomenti o che non ritornano alcun valore).

Tutti i tipi riconosciuti da Standard ML (siano essi "tipi semplici" come i tipi predefiniti appena citati, o tipi più complessi definiti dall'utente) possono essere aggregati in vario modo per generare tipi più complessi. Il modo più semplice per comporre tipi è formando delle tuple (definite sul prodotto cartesiano dei tipi che le compongono). Per esempio, abbiamo già visto come una funzione a più argomenti fosse definibile come una funzione che prende come unico argomento una tupla composta dai vari argomenti:

```
val sommaquadrati = fn (a,b) => a * a + b * b;
```

Questa funzione ha tipo "int * int -> int"; in altre parole, il suo parametro è di tipo "int * int", che rappresenta il prodotto cartesiano dell'insieme di definizione di `int` per se stesso (matematicamente parlando, $\mathcal{N} \times \mathcal{N}$).

Più formalmente parlando, una tupla è un insieme ordinato di più valori, ognuno dei quali avente un tipo riconosciuto da ML. L'insieme di definizione di una tupla è dato dal prodotto cartesiano degli insieme di definizione dei valori che la compongono. Notare come il tipo `unit` (avente un unico valore `()`) possa essere visto come una tupla di 0 elementi e come le tuple di 1 elemento corrispondano con l'elemento stesso. Per esempio, `(6)` è equivalente a `6` ed ha tipo `int`; quindi, l'espressione "(6)" viene valutata come "6" e l'espressione "(6) = 6" viene valutata come `true`.

Notare che il meccanismo del pattern matching applicato alle tuple permette di creare binding fra più nomi e simboli contemporaneamente: se "**val** coppia=("pi_greco", 3.14)" crea un legame fra il simbolo "`coppia`" e la coppia "("pi_greco", 3.14)" (avente tipo "`string * real`", "**val** (pigreco, pi) = coppia" crea un legame fra il simbolo "`pigreco`" e la stringa "`"pi greco"`" e fra il simbolo "`pi`" ed il numero "`3.14`" (di tipo real). Questo stesso meccanismo è stato usato finora per passare parametri multipli ad una funzione usando una tupla come unico argomento.

Standard ML permette di *definire dei sinonimi* per tipi di dati esistenti, usando il costrutto `type`: per esempio, se in un programma usiamo valori interi per rappresentare il tempo, si potrebbe usare l'identificatore `time_t` per valori riferiti ai tempi, usando una definizione del tipo "**type** time_t = int". Questa definizione permette di associare simboli a valori di tipo `time_t`, con definizioni tipo "**val** a:time_t = 5". I valori di tipo `time_t` rimangono però di fatto dei valori interi ed è quindi lecito sommare `a` a valori di tipo intero. Questo utilizzo di `type` aumenta l'espressività e rende più semplice capire il codice: nell'esempio citato sopra, è immediatamente evidente che `a` e `b` rappresentano dei tempi e non dei generici valori interi. Notare però che `type` non introduce nuovi tipi: tornando all'esempio precedente, un'espressione "`a + c`" con `c` associato ad un valore intero (quindi questa espressione somma un valore `time_t` ad un valore `int`) è valida. In altre parole, `time_t` è semplicemente un sinonimo per `int`, non un nuovo tipo.

Questo meccanismo può risultare particolarmente utile per associare nomi simbolici a tipi ottenuti come tuple:

```
type coppiadiinteri = int * int;
type nomecognome = string * string;
type coppiadireali = real * real;
...
```

Quindi per esempio una definizione "**type** coppiadireali = real * real" associa il nome `copiadireali` al tipo `real * real` e permette di definire in modo semplice e chiaro funzioni che ricevono una coppia di numeri floating point come argomento: "**val** fr = **fn** (x, y): coppiadireali => x * y" (una definizione come "**val** fr = **fn** (x, y) = > x * y" avrebbe definito una funzione da coppie di interi ad interi).

Come ulteriore esempio, si consideri una funzione `c2t` che riceve come parametri una coppia di interi `(i1, i2)` ed un intero `i3` generando come risultato la tripletta di interi `(i1, i2, i3)`. Un primo tentativo di definizione di `c2t` può essere:

```
val c2t = fn (x, y) => fn z => (x,y,z);
```

o analogamente

```
fun c2t (x, y) z = (x, y, z);
```

ma questa funzione risulta avere tipo "'a * 'b -> 'c -> 'a * 'b * 'c" che non è proprio quel che ci attendevamo (int * int -> int -> int * int * int). Qui `'a`, `'b` e `'c` rappresentano tipi generici (tutto diventerà più chiaro parlando di polimorfismo). In altre parole, la funzione `c2t` come definita sopra può essere invocata come c2t ("ciao", "mondo") #"c" (ritornando come risultato "("ciao", "mondo", #"c")", di tipo "`string * string * char`"). Una definizione tipo

```
val c2t = fn (x:int , y:int) => fn z:int => (x, y, z);
```

chiaramente risolve il problema. La definizione può essere resa più elegante introducendo un nome per il tipo "`int * int`":

```
type coppiainteri = int * int;
val c2t = fn (x, y):coppiainteri => fn z:int => (x, y, z);
```

```
type currency = string;
type money = real * currency;

fun convert (amount, to) =
    let val toeur = fn
        (x, "eur") => x
      | (x, "usd") => x / 1.05
      | (x, "ounce gold") => x * 1113.0
    in
        ( case to of
            "eur" => toeur amount
          | "usd" => toeur amount * 1.05
          | "ounce gold" => toeur amount / 1113.0
        , to)
   end;
```

Figure 1: Conversione di denaro fra valute, usando stringhe per rappresentare le varie valute. Si noti che i pattern non sono esaustivi (non coprono tutti i possibili valori).

Ancora, è importante notare che, come precedentemente detto, `type` definisce un sinonimo per un tipo esistente, ma non definisce un nuovo tipo. Questo può comportare problemi in alcune situazioni, per esempio quando si vuole che una variabile possa assumere solo un numero finito di valori (per semplificare il pattern matching). Come esempio di questa situazione, si consideri il problema di sviluppare una funzione "`convert`" che riceve come argomento un quantitativo di denaro in una valuta nota e lo converte in una diversa valuta. Gli argomenti in ingresso sono il quantitativo di denaro da convertire (associato alla sua valuta) e la valuta in cui convertire. Usando delle stringhe (con valori, per esempio, `"eur"`, `"usd"` e `"ounce gold` per rappresentare le valute, un quantitativo di denaro in una determinata valuta può essere rappresentato con una coppia "`(real, string)`"; ovviamente, si possono usare dei sinonimi per "`string`" e per le coppie "`(real, string)`" in modo da rendere il codice più chiaro. Una possibile implementazione di `convert` usando questa tecnica è mostrata in Figura 1. Prima di tutto, è importante notare come "`type`" sia usato per definire "`currency`" e "`money`" come sinonimi per "`string`" e "`real * currency`". A questo punto, la funzione "`convert`" può ricevere come argomento una coppia il cui primo elemento è di tipo "`money`" (e rappresenta il quantitativo di denaro da convertire) ed il secondo elemento è di tipo "`currency`" (e rappresenta la valuta in cui vogliamo convertire il denaro). La funzione potrà essere quindi invocata come "convert ((1.0, "eur"), "usd")".

La funzione "`convert`" lavora utilizzando una "funzione privata" "`toeur`" (presente solo nell'ambiente locale di "`convert`", grazie al costrutto "`let ... in ... end`") che converte un quantitativo di denaro (di tipo "`money`") in euro, ritornando un numero reale. Tale funzione utilizza il pattern matching sulla coppia "`amount`" (ricordare: "`amount`" è di tipo "`real * currency`" quindi è una coppia il cui primo elemento è un numero floating point ed il secondo elemento è una stringa). Se la stringa che rappresenta la valuta in cui è espresso "`amount`" è `"eur"`, "`toeur`" può ritornare il primo elemento della coppia (il denaro è già in euro), se è `"usd"` il primo elemento della coppia va moltiplicato per 1.05 e così via. Si noti che i pattern usati sono pattern coppia composti da un variable pattern (`x`, che va a matchare il numero reale) ed un pattern costante (`"eur"`, `"usd"`, etc che va a matchare la stringa che esprime in che valuta è il denaro). Dopo aver convertito il denaro in euro (invocando "`toeur`"), "`convert`" converte questo quantitativo nella valuta desiderata (espressa da "`to`"). Ancora, la conversione è fatta usando pattern matching.

Il codice presentato in Figura 1 sembra funzionare correttamente, ma ha un grosso problema: i pattern che vengono matchati con "`to`" e con il secondo elemento di "`amount`" non coprono tutti i possibili valori di tipo stringa. Questo può causare (oltre a dei warning quando il codice viene compilato od interpretato) degli errori a tempo di esecuzione, se vengono specificate valute non esistenti o non supportate dalla funzione (per esempio, "convert ((1.0, "eur"), "usb")" - banale errore di battitura in cui si è digitato "`usb`" invece di "`usd`"). Questo problema può essere alleviato usando il wildcard pattern "_" per terminare le due espressioni di pattern matching, in modo da rendere i pattern esaustivi (ricordare che "_" matcha qualsiasi valore e può essere utilizzato per implementare qualcosa di simile alle clausole "`default:`" degli switch C o C++). Il problema è decidere quale valore ritornare quando viene specificata una valuta inesistente o non supportata): per esempio, in Figura 2 si è deciso di ritornare -1 in caso di errore. Ora,

```
type currency = string;
type money = real * currency;

fun convert (amount, to) =
    let val toeur = fn
        (x, "eur")         => x
      | (x, "usd")         => x / 1.05
      | (x, "ounce_gold") => x * 1113.0
      | (_, _)             => ~1
    in
        ( case to of
            "eur"         => toeur amount
          | "usd"         => toeur amount * 1.05
          | "ounce_gold" => toeur amount / 1113.0
          | _             => ~1
        , to)
    end;
```

Figure 2: Conversione di denaro fra valute, usando stringhe per rappresentare le varie valute. Si noti che i pattern ora coprono tutti i possibili valori grazie all'utilizzo del wildcard pattern.

invece di generare un errore a tempo di esecuzione, "`convert`" ritorna valori apparentemente senza senso in caso di input sbagliati.

Anche la soluzione di Figura 2 non è ottimale, perché in caso di input non validi invece di ottenere un errore a run-time sarebbe preferibile avere un errore di compilazione (o un errore dell'interprete basato su un'analisi statica del codice). Ma questo sarebbe possibile solo se il tipo di "`currency`" avesse solo pochi valori validi. In pratica, sarebbe necessario definire un nuovo tipo, che ha come possibili valori solo "`eur`", "`usd`" e "`ounce_gold`", in modo simile ad un tipo enumerato. Per venire incontro ad esigenze di questo genere, ML permette di definire nuovi tipi tramite la keyword `datatype`; in particolare, `datatype` permette di far riconoscere all'interprete valori non conosciuti per default. Il modo più semplice per usare `datatype` è la definizione dell'equivalente di un tipo enumerato (proprio quel che ci serve in questo caso!); per esempio

```
datatype currency = eur | usd | ounce_gold;
```

definisce tre nuovi valori ("`eur`", "`usd`" ed "`ounce_gold`") che non erano precedentemente riconosciuti: prima di questa definizione, provando ad associare il valore "`eur`" ad un nome (con "**val** c = eur") si ottiene un errore. Ma dopo la definizione, "**val** c = eur" ha successo ed a "`c`" viene associato un valore di tipo "`currency`".

In una definizione di questo tipo, il simbolo "`|`" rappresenta una "o" a significare che una variabile di tipo "`currency`" può avere valore "`eur`" oppure "`usd`" oppure "`ounce_gold`". I nomi separati da "`|`" rappresentano valori costanti che stiamo definendo. Tecnicamente, essi sono dei *costruttori di valore*, vale a dire funzioni che ritornano (costruiscono) valori del nuovo tipo che stiamo definendo. In questo caso (definizione di un tipo equivalente ad un tipo enumerativo), i costruttori non hanno alcun argomento (costruttori costanti); esiste però la possibilità di definire costruttri che generano un valore a partire da un argomento.

Per esempio, si potrebbe pensare di definire il tipo `money` non come un sinonimo di una coppia, ma come un nuovo tipo di dato:

```
datatype money = Eur of real | Usd of real | Ounce_gold of real;
```

(notare che qui si sono usati "`Eur`", "`Usd`" ed "`Ounce_gold`" come nomi per i costruttori, perché i nomi "`eur`", "`usd`" ed "`ounce_gold`" sono già usati come costruttori per un altro tipo).

Come tutte le funzioni di Standard ML, anche questi costruttori di valore hanno un solo argomento (e come al solito si può ovviare a questa limitazione usando una tupla come argomento).

In questo caso, "`Eur`" non rappresenta un valore (costante) del tipo "`money`" (come accadeva invece con "`eur`" per il tipo "`currency`"), ma è una funzione da numeri floating point a valori di tipo `money` (in Standard ML, "real -> money"). I valori di tipo `money` sono quindi "`Eur 0.5`" o simili.

La Figura 3 mosta come la funzione "`convert`" sia implementabile usando i due nuovi tipi di dato appena descritti, facendo pattern matching su valori di tipo "`currency`" in "`toeur`" e facendo pattern

```
datatype currency = eur | usd | ounce_gold;
datatype money = Eur of real | Usd of real | Ounce_gold of real;

fun convert (amount, to) =
    let val toeur = fn
        Eur x        => x
      | Usd x        => x / 1.05
      | Ounce_gold x => x * 1113.0
    in
        ( case to of
            eur        => toeur amount
          | usd        => toeur amount * 1.05
          | ounce_gold => toeur amount / 1113.0
        , to)
   end;
```

Figure 3: Conversione di denaro fra valute, usando un nuovo tipo di dato per rappresentare le varie valute. Questo esempio mostra come fare pattern matching sui tipi di dato definiti dall'utente.

```
datatype currency = eur | usd | ounce_gold;
datatype money = Eur of real | Usd of real | Ounce_gold of real;

fun convert (amount, to) =
    let val toeur = fn
        Eur x        => x
      | Usd x        => x / 1.05
      | Ounce_gold x => x * 1113.0
    in
        case to of
            eur        => Eur        (toeur amount)
          | usd        => Usd        (toeur amount * 1.05)
          | ounce_gold => Ounce_gold (toeur amount / 1113.0)
   end;
```

Figure 4: Conversione di denaro fra valute, versione finale.

matching su valori di tipo "`money`" nel corpo di "`convert`". Notare però che la conversione della funzione ai nuovi tipi di dato è ancora parziale, perché questa versione di "`convert`" riceve una coppia in cui il primo elemento è di tipo "`money`" ed il secondo è di tipo "`currency`", ma ritorna una coppia (reale, currency) invece che un valore di tipo "`money`". L'implementazione di Figura 4 risolve questo ultimo problema.

Quando si usa "`datatype`" per definire un nuovo tipo di dato, è importante ricordare che definire un tipo semplicemente specificandone l'insieme di definizione (insieme dei valori di tale tipo) non è abbastanza. Bisogna anche definire come operare su valori di tale tipo, definendo funzioni che operano sul tipo che si va a definire (torando all'esempio precedente, definire i tipi "`currency`" e "`money`" senza definire la funzione "`convert`" non è molto utile...).

Come ulteriore esempio, si consideri un tipo `numero` che rappresenta numeri reali o interi. Tale tipo potrebbe essere definito come

```
datatype numero = intero of int | reale of real;
```

Ma se non si definiscono delle funzioni che operano su tale tipo (implementando le operazioni aritmetiche) il tipo "`numero`" non è utilizzabile. E' quindi importante definire anche funzioni come

```
val sommanumeri = fn (intero a, intero b) => intero (a + b)
                   | (intero a, reale  b) => reale  ((real a) + b)
                   | (reale  a, intero b) => reale  (a + (real b))
                   | (reale  a, reale  b) => reale  (a + b);
val sottrainumeri = fn (intero a, intero b) => intero (a - b)
                     | (intero a, reale  b) => reale  ((real a) - b)
                     | (reale  a, intero b) => reale  (a - (real b))
                     | (reale  a, reale  b) => reale  (a - b);
...
```

Si lascia al lettore l'esercizio di definire "`moltiplicanumeri`", "`dividinumeri`" e via dicendo.

Per finire, è interessante notare come un tipo di dato definito tramite "`datatype`" possa essere definito anche basandosi sul tipo stesso: fino ad ora, abbiamo visto che un costruttore può avere 0 argomenti (costruttore costante) o 1 argomento, usando tipi di dato più semplici (`int`, `real`, coppie...) per tale argomento. Ma l'argomento di un costruttore di un tipo "`T`" può anche avere un argomento di tipo "`T`" o di un tipo derivato da "`T`". In questo caso, si parla di *tipo di dato ricorsivo*. Per esempio, una lista di interi è definibile in modo ricorsivo come

```
datatype lista = foglia of int | nodo of (int * lista);
```

usando un costruttore "`nodo`" che ha un argomento di tipo "`int * lista`" (quindi, il tipo "`lista`" è definito in termini di se stesso; per questo si chiama ricorsivo). Basandosi su questa definizione, la funzione "`Len`" che calcola la lunghezza di una lista può essere definita come

```
val rec Len = fn foglia _      => 1
              | nodo   (_, l) => 1 + Len l;
```

mentre la funzione "`Concatena`" può essere definita come

```
val Concatena = fn (foglia v, l)       => nodo (v, l)
                 | (nodo (v, l1), l2) => nodo (v, Concatena (l1, l2));
```

Notare che la precedente definizione di lista di interi non permette di rappresentare le liste vuote ed una rappresentazione tradizionalmente più usata (perché permette di rappresentare liste vuote) è la seguente:

```
datatype lista = vuota | cons of (int * lista);
```

Si rimanda alle dispense dedicate ai tipi di dato ricorsivi per maggiori dettagli al proposito delle liste in Standard ML.

Per finire, Standard ML implementa il concetto di *type variable* (una variabile i cui valori sono i tipi di dato), che permette di supportare il concetto di polimorfismo parametrico implicito (definendo "classi di funzioni" o "classi di tipi di dato" che sono parametrizzate rispetto a uno o più tipi, indicati come α, β, etc...). Per esempio, la funzione

```
fn x => x
```

ha tipo

```
'a -> 'a
```

che significa $\alpha \rightarrow \alpha$ (i programmi ML sono scritti usando caratteri ASCII, quindi le lettere greche come α non si possono usare... Si usano quindi le lettere dell'alfabeto latino con un apice davanti). Vale a dire, questa funzione (che i lettori potranno facilmente riconoscere come l'identità) mappa un valore di un generico tipo α in un altro valore **dello stesso tipo**.

I type variable possono essere anche usati per definire tipi di dati generici (o polimorfici), che in realtà non sono dei veri tipi di dati, ma delle classi di tipi di dati (parametrizzate rispetto ad una o più variabili α, β, etc...). Per esempio, la definizione

```
datatype 'a oerrore = errore | valore of 'a
```

definisce una classe di tipi di dato "α oerrore" i cui valori sono "`errore`" o valori di un generico tipo α. Potremmo quindi avere un tipo "`int oerrore`" che ha come valori "`errore`" o numeri interi (in realtà, valori generati dai numeri interi usando il costruttore "`valore`"). Ma il sistema conoscerà anche una "regola" per generare tipi "`real oerrore`", "`string oerrore`" e così via (per ogni tipo conosciuto da ML). In generale, `datatype` permette di specificare un type variable come argomento (che precede il nome del tipo) e questa variabile può essere usata per descrivere il tipo di uno (o più) argomenti di uno (o più) dei costruttori. Se sono necessari più type variable, si possono racchiudere in una tupla, come nel seguente esempio:

```
datatype ('a,'b) tipostrano = nessuno | uno of 'a | due of 'a * 'b
```

10 Cenni di Programmazione Modulare

Tutto quanto descritto fino ad ora si applica alla cosiddetta *programmazione in piccolo*, in cui un programma è generalmente composto da una sola unità di compilazione (compilation unit) e la sua bassa complessità non richiede alcuna modularizzazione del codice.

In altre parole, un programma in Standard ML appare come una sequenza di definizioni ed applicazioni di funzioni che manipolano tutte lo stesso ambiente e questo può causare una serie di problemi per quanto riguarda la riusabilità del codice. Per esempio, si consideri il seguente codice Standard ML:

```
val pi = 3.14152;
val areacerchio = fn r => r * r * pi;
```

Se si volesse riusare questo codice in un altro programma, bisognerebbe assicurarsi che il nuovo programma non usi già il nome "`pi`" per altri scopi (e non contenga quindi differenti definizioni di "`pi`"). Chiaramente, in questo specifico caso il problema può essere risolto "nascondendo" la definizione di "`pi`" tramite un blocco "`let...in...end`", ma in generale per poter riutilizzare una porzione di codice è necessario isolarla in una qualche "entità" (modulo, package, unità di compilazione o che altro) che permetta di controllare in qualche modo la visibilità delle varie definizioni. Molti dei linguaggi di programmazione moderni supportano questa *programmazione in grande* tramite costrutti linguistici di vario genere che permettono di dividere il codice in unità di compilazione (che possono avere nomi diversi in diversi linguaggi) distinte, di specificare chiaramente l'interfaccia software di ogni unità di compilazione e di separare tale interfaccia software dalla reale implementazione (che può essere in qualche modo "nascosta" all'utilizzatore finale del codice).

Il meccanismo fornito da Standard ML per supportare la programmazione in grande e la modularizzazione del codice è quello delle *structure* e *signature*. Informalmente, si può dire che una struttura (structure) SML permette di "raggruppare" una serie di definizioni in un ambiente (environment), in modo da evitare problemi con "collisioni di identificatori" (stesso identificatore usato con scopi diversi in parti diverse del codice) analoghi a quello discusso sopra. In questo senso, una struttura SML è simile ad un namespace C++. Il costrutto `signature` può essere invece utilizzato per specificare un'interfaccia software. Creando un legame fra structure e signature è quindi possibile specificare quali dei legami presenti nella struttura verranno esportati e come verranno esportati. In altre parole, una struttura contiene l'implementazione di un modulo software, mentre un signature può essere usato per specificarne l'interfaccia software (qualcuno sostiene che il signature specifichi il tipo della struttura).

Più formalmente, una struttura SML (da non essere confusa con le strutture o record presenti alcuni linguaggi imperativi come C o C++) definisce un ambiente che contiene i legami (binding) definiti fra la keyword `struct` e la keyword `end`. Una stuttura può essere quindi

```
struct
  val pi = 3.14152;
  val areacerchio = fn r => r * r * pi
end
```

ma poiché una struttura "anonima" come quella definita qui sopra è generalmente inutile (non c'è modo di accedere ai binding contenuti nel suo ambiente) si mettono assieme definizione e dichiarazione, associando un identificatore alla struttura tramite la keyword `structure`:

```
structure cerchio = struct
  val pi = 3.14152;
  val areacerchio = fn r => r * r * pi
end
```

Le definizioni contenute nella struttura non hanno effetto sull'ambiente globale, ma modificano solo l'ambiente della struttura, che può essere acceduto tramite la sintassi "`<nome struttura>.<identificatore>`" (dot notation). Il nome "`areacerchio`" non è quindi riconosciuto nell'ambiente globale:

```
> areacerchio 1.0;
poly: : error: Value or constructor (areacerchio) has not been declared
Found near areacerchio 1.0
Static Errors
```

ma è invece visibile come "`cerchio.areacerchio`":

```
> cerchio.areacerchio 1.0;
val it = 3.14152: real
```

L'utilizzo di una struttura permette quindi di riutilizzare senza problemi il codice in programmi che già utilizzano il nome `pi` per altri scopi. Purtroppo, però, ci sono situazioni in cui questo non è abbastanza. Si consideri per esempio l'implementazione di un tipo di dato astratto "`set`" che rappresenti un insieme (per la precisione, il tipo "`'a set`" rappresenta insiemi di elementi di tipo "`'a`"). Una possibile implementazione, basata sull'utilizzo di "`'a list`" per implementare un "`'a set`", è mostrata in Figura 5. Si noti come il tipo di "`emptyset`" sia "`'a list`", il tipo di "`isin`" sia "`fn: ''a -> ''a list -> bool`" e cosìvia. Anche definendo "`'a set`" come sinonimo di "`'a list`" (aggiungendo "`type 'a set = 'a list;`" prima delle altre definizioni) la situazione non migliora. Si potrebbero esplicitamente specificare i tipi di "`emptyset`" e delle varie funzioni per usare "`'a set`", ma questo migliorerebbe la situazione solo a livello formale e non pratico (sarebbe, per esempio, ancora possibile applicare "`hd`" al risultato di "`addin`", in quanto non esiste differenza fra "`'a set`" ed "`'a list`").

Rinchiudendo le definizioni in una struttura (Figura 6) si possono al solito eliminare i problemi di "collisione fra identificatori", ma non si riesce ancora a distinguere il tipo "`'a set`" da "`'a list`". Per rendersene conto, si provi hd (Set.addin 1 Set.emptyset); (in teoria vorremmo che questa espressione generasse un errore, in quanto "`hd`" è una funzione che agisce su "`'a list`", non su "`'a set`).

E' però possibile definire un signature (Figura 7) che specifica l'interfaccia software del tipo "`'a set`" in modo generico, senza citare dettagli implementativi (come "`'a list`", etc...). Formalmente, un signature è definito da una serie di dichiarazioni di tipo (o tipo di variabile) contenute fra le keyword "`sig`" ed "`end`". Al solito, è possibile associare un nome ad un signature anonimo (usando la keyword "`signature`"; nell'esempio si usa "**signature** SET = ...", dove "`SET`" è il nome associato al signature).

Una volta definito un signature, è possibile creare un legame fra struttura e signature usando "`:>`" come mostrato in Figura 8 (si noti che invece di associare il nome "`SET`" al signature ed usare "`:> SET`" si potrebbe utilizzare un signature anonimo). Questo risolve i problemi precedentemente discussi.

11 Lavorare con Standard ML

Una macchina astratta per il linguaggio ML (vale a dire, una macchina astratta in grado di capire ed eseguire programmi scritti in ML) può essere implementata tramite interpreti o compilatori (meglio: può essere implementata in modo prevalentemente interpretato o prevalentemente compilato). Esistono vari compilatori ed interpreti ML, ma in questa sede ci concentreremo su un'implementazione che fornisce una modalità di funzionamento interattiva che è inizialmente più intuitiva e semplice da usare. Indipendentemente dal fatto che l'implementazione della macchina astratta sia basata su un compilatore o su un interprete, interagiremo con essa tramite un ciclo REPL (Read/Evaluate/Print Loop): un programma che ciclicamente *legge* espressioni da console, le *valuta* e *stampa* su console i risultati.

```
val emptyset = [];

val rec isin =
      fn x =>
              (fn []    => false
                | y::l =>
                      if (x = y)
                          then
                              true
                          else
                              isin x l);

val addin =
      fn x =>
               fn l =>
                      if (isin x l)
                          then
                              l
                          else
                              x::l;

val rec removefrom =
      fn x =>
              (fn [] => []
                | y::l =>
                      if (x = y)
                          then
                              l
                          else
                              y::(removefrom x l));
```

Figure 5: Implementazione di insiemi (`'a set`) senza porsi problemi di riusabilità del codice.

```
structure Set = struct
  val emptyset = [];

  val rec isin =
        fn x =>
                (fn []    => false
                  | y::l =>
                        if (x = y)
                            then
                                true
                            else
                                isin x l);

  val addin =
        fn x =>
                 fn l =>
                        if (isin x l)
                            then
                                l
                            else
                                x::l;

  val rec removefrom =
        fn x =>
                (fn [] => []
                  | y::l =>
                        if (x = y)
                            then
                                l
                            else
                                y::(removefrom x l));
end;
```

Figure 6: Implementazione di insiemi (`'a set`) in una struttura.

```
signature SET = sig
  type 'a set

  val emptyset   : 'a set
  val isin       : ''a -> ''a set -> bool
  val addin      : ''a -> ''a set -> ''a set
  val removefrom : ''a -> ''a set -> ''a set
end;
```

Figure 7: Interfaccia software (signature) del tipo `'a set`. Si noti come non siano specificato alcun aspetto implementativo.

```
signature SET = sig
  type 'a set

  val emptyset   : 'a set
  val isin       : ''a -> ''a set -> bool
  val addin      : ''a -> ''a set -> ''a set
  val removefrom : ''a -> ''a set -> ''a set
end;

structure Set = struct
  val emptyset = [];

  val rec isin =
        fn x =>
                (fn []    => false
                  | y::l =>
                        if (x = y)
                            then
                                true
                            else
                                isin x l);

  val addin =
        fn x =>
                 fn l =>
                        if (isin x l)
                            then
                                l
                            else
                                x::l;

  val rec removefrom =
        fn x =>
                (fn [] => []
                  | y::l =>
                        if (x = y)
                            then
                                l
                            else
                                y::(removefrom x l));
end :> SET;
```

Figure 8: Legame fra struttura "`Set`" e signature "`SET`".

Questo significa che operativamente un utente può eseguire *interattivamente* un interprete o un compilatore (quindi si può interagire con esso tramite un prompt). In particolare, in seguito verrà usato il REPL di "Poly/ML" (`http://www.polyml.org/`) anche se tutto quanto detto si può applicare senza problemi anche ad altri interpreti o compilatori che siano compliant con Standard ML (come per esempio "Standard ML of New Jersey" - `http://www.smlnj.org/`); le uniche differenze sono nelle modalità di interazione con la macchina astratta. Il REPL di Poly/ML può essere lanciato eseguendo il comando "`poly`", che risponde come segue:

```
luca@nowhere:~$ poly
Poly/ML 5.2 Release
>
```

Notare che Poly/ML presenta il suo prompt (caratterizzato dal simbolo "`>`") e rimane in attesa che vengano immesse espressioni da valutare.

Come già spiegato, eseguire un programma funzionale significa valutare le espressioni che lo compongono (vedi Sezione 1); quindi, il REPL di una macchina astratta ML può essere inizialmente visto come un valutatore di espressioni.

Le espressioni immesse tramite il prompt del REPL vengono valutate da un interprete (o compilate eseguendo poi l'output del compilatore) man mano che l'utente le immette; ogni volta che il ciclo REPL valuta (*evaluate*) un'espressione, ne visualizza a schermo (*print*) il valore risultante. Per esempio, digitando

```
5;
```

si ottiene risposta

```
val it = 5 : int
```

indicante che l'espressione è stata valutata al valore 5 e tale valore è di tipo intero. Chiaramente, si possono valutare anche espressioni più complesse, usando gli operatori aritmetici che conosciamo e la notazione infissa (con cui siamo familiari):

```
> 5 + 2;
val it = 7 : int
> 2 * 3 * 4;
val it = 24 : int
> (2 + 3) * 4;
val it = 20 : int
```

e così via.

Notare che il simbolo "`;`" serve da *terminatore* indicando al REPL la fine di un'espressione (quindi, la fine della fase *read* e l'inizio della fase *evaluate*). Quindi, un'espressione non viene valutata fino a che il REPL non incontra un "`;`":

```
> 1 + 2
# + 3;
val it = 6 : int
```

Notare che dopo aver premuto il tasto "Enter" senza aver digitato un "`;`" (dopo "`1 + 2`"), l'interfaccia del REPL va a capo, ma non presenta l'usuale prompt ("`>`"). Il nuovo carattere di prompt che viene presentato ("`#`") significa che il REPL sta ancora attendendo il completamento dell'espressione corrente.

Da questi primi esempi si comincia a vedere come data un'espressione, la macchina astratta ML possa essere in grado di *inferire* il tipo dei parametri e del valore di ritorno. Per rendere possibile questo, ML deve usare una tipizzazione stretta e non può effettuare conversioni automatiche come fa per esempio il linguaggio C. Come conseguenza,

```
5 + 2.0;
```

genera un errore:

```
Error-Can't unify Int32.int/int with real (Overloading does not include type) Found near
+
( 5, 2.0)
Static errors (pass2)
```

Questo avviene perché per ML il numero `5` è un intero (tipo `int`), mentre il numero `2.0` è un floating point (tipo `real`). L'operatore `+` (somma) accetta come parametri due numeri interi o due numeri floating point, ma non è definito per operandi di tipo divero; per finire, come detto ML si rifiuta di

convertire automagicamente numeri interi in floating point o viceversa (come invece fa un compilatore C: l'espressione `5 + 2.0` viene valutata da un compilatore C dopo aver converito `5` in `5.0`).

Come precedentemente detto, Standard ML fornisce altri tipi di base oltre a `int` e `real`:

```
> 2;
val it = 2 : int
> 2.0;
val it = 2.0 : real
> 2 > 1;
val it = true : bool
> "abc";
val it = "abc" : string
> #"a";
val it = #"a" : char
```

Siamo ora pronti a fare la prima cosa che viene fatta quando si incontra un nuovo linguaggio di programmazione:

```
> "Hello, " ^ "world";
val it = "Hello, world" : string
```

notare che `"Hello, "` e `"world"` sono due valori di tipo stringa e "^" è l'operatore di concatenazione fra stringhe, che riceve in ingresso due parametri di tipo stringa e genera una stringa che rappresenta la loro concatenazione.

Fino ad ora abbiamo visto come usare un REPL ML per valutare espressioni in cui tutti i valori sono espressi esplicitamente. Ricordiamo però che uno dei primi tipi di *astrazione* che abbiamo visto consiste nell'assegnare **nomi** ad "entità" (termine informale) riconosciute da un linguaggio. Vediamo quindi come sia possibile associare nomi ad "entità" in ML e quali siano le "entità" denotabili in ML (in un linguaggio imperativo, tali "entità" denotabili erano variabili, funzioni, tipi di dato, ...). Come in tutti i linguaggi funzionali, anche in ML esiste il concetto di *ambiente/environment* (una funzione che associa nomi a valori denotabili) ma non esiste il concetto di memoria (funzione che associa ad ogni variabile il valore che essa contiene)[4]; quindi, è possibile associare nomi a valori, ma non è possibile creare variabili modificabili:

```
> val n = 5;
val n = 5 : int
> val x = 2.0;
val x = 2.0 : real
```

Questi comandi associano il valore intero `5` al nome `n` ed il valore floating point `2.0` al nome `x`. Chiaramente, è possibile usare qualsiasi tipo di espressione per calcolare il valore a cui associare un nome:

```
> val x = 5.0 + 2.0;
val x = 7.0 : real
> val n = 2 * 3 * 4;
val n = 24 : int
```

Dopo aver associato un nome ad un valore, è possibile usare tale nome (invece del valore) nelle successive espressioni:

```
> val x = 5.0 + 2.0;
val x = 7.0 : real
> val y = x * 2.0;
val y = 14.0 : real
> x > y;
val it = false : bool
```

notare che "**val** y = x * 2;" avrebbe generato un errore... Perché?

Un secondo livello di astrazione consiste nel definire *funzioni*, associando nomi a blocchi di codice. Mentre nei linguaggi imperativi questo è un concetto diverso rispetto alla definizione di variabili, nei linguaggi funzionali come ML esiste un tipo di dato "funzione":

[4] Teoricamente, il concetto di memoria / variabile modificabile esiste anche in Standard ML, ma noi non ce ne occuperemo.

```
> fn x => x + 1;
val it = fn : int -> int
```

in questo caso il valore risultante dalla valutazione dell'espressione è una funzione (nel senso matematico del termine) $\mathcal{N} \to \mathcal{N}$. Poiché non si è usata un'espressione "`val`", a tale funzione non è stato dato un nome. Una funzione può però essere applicata a dei dati anche senza darle un nome:

```
>  (fn x => x +  1) 5;
val it = 6 : int
```

Anche per le funzioni, la macchina astratta ML è generalmente in grado di inferire correttamente i tipi di dato:

```
> fn x => x + 1;
val it = fn : int -> int
> fn x => x + 1.0;
val it = fn : real -> real
```

A questo punto, dovrebbe essere chiaro come associare un nome ad una funzione, tramite quella che in altri linguaggi verrebbe chiamata definizione di funzione e che in ML corrisponde ad una definizione di variabile (tecnicamente, il seguente codice definisce una variabile "`doppio`" di tipo funzione da interi ad interi):

```
> val doppio = fn n => n * 2;
val doppio = fn : int -> int
> doppio 9;
val it = 18 : int
> doppio 4.0;
Error-Can't unify int with real (Different type constructors) Found near doppio(4.0
)
Static errors (pass2)
...
```

ML fornisce anche una sintassi semplificata, basata sull'espressione `fun`, per definire funzioni:

```
> fun doppio n = 2 * n;
val doppio = fn : int -> int
> doppio 9;
val it = 18 : int
```

La sintassi di `fun` appare più intuitiva della definizione esplicita `val ... = fn ...`, ma è equivalente ad essa.

Componendo quanto visto fin'ora con l'espressione `if`, è possibile definire funzioni anche complesse (equivalenti ad algoritmi iterativi implementati con un linguaggio imperativo) tramite ricorsione. Per esempio,

```
> val rec fact = fn n => if n = 0 then 1 else n * fact(n - 1);
val fact = fn : int -> int
> fact 5;
val it = 120 : int
```

Notare l'utilizzo di `val rec` invece che il semplice `val`. Una semplice definizione con `val` avrebbe generato un errore:

```
> val fact = fn n => if n = 0 then 1 else n * fact(n - 1);
Error-Value or constructor (fact) has not been declared  Found near
if =( n, 0) then 1 else *( n, fact(......))
Static errors (pass2)
```

Come si può vedere, senza usare la keyword `rec` il nome `fact` non è visibile nella definizione della funzione (il binding fra `fact` e la funzione non fa ancora parte dell'ambiente). Usando `fun` si possono invece definire funzioni ricorsive senza usare particolari accortezze:

```
> fun fact n = if n = 0 then 1 else n * fact(n - 1);
val fact = fn : int -> int
> fact 4;
val it = 24 : int
```

```
val rec fact_tr = fn n =>
                  fn res =>
                    if n = 0 then
                                 res
                               else
                                 fact_tr (n - 1) (n * res);
val fact = fn n => fact_tr n 1;

fun fact_tr1 n res = if n = 0 then
                                 res
                               else
                                 fact_tr (n - 1) (n * res);
fun fact1 n = fact_tr1 n 1;
```

Figure 9: Fattoriale con ricorsione in coda.

```
val rec gcd = fn a => fn b => if b = 0 then
                                       a
                                     else
                                       gcd b (a mod b);


fun gcd1 a b = if b = 0 then a else gcd b (a mod b);
```

Figure 10: Massimo Comun Divisore.

Per capire meglio la differenza fra `val` e `val rec` è utile considerare il seguente esempio:

```
> val f = fn n => n + 1;
val f = fn : int -> int
> val f = fn n => if n = 0 then 1 else n * f(n - 1);
val f = fn : int -> int
> f 4;
val it = 16 : int
```

da cosa deriva il risultato $f(4) = 16$? Dalla definizione "val f = fn n => if n = 0 then 1 else n * f(n − 1);" un utente inesperto potrebbe aspettarsi che il simbolo `f` venga associato alla funzione fattoriale e quindi $f(4) = 1 * 2 * 3 * 4 = 24$. Il risultato ottenuto è invece totalmente diverso, perché nel momento in cui viene valutata la seconda espressione "val f ... ", il simbolo `f` è sempre associato alla prima definizione ("fn n => n + 1"). Quindi, `f 4` viene valutata come "if 4 = 0 then 1 else 4 * f(n − 1);" dove `f` è valutata come "fn n => n + 1". Quindi, `f 4` è valutata come `4 * ((4 - 1) + 1)` vale a dire 16. L'esempio

```
> val f = fn n => n + 1;
val f = fn : int -> int
> val rec f = fn n => if n = 0 then 1 else n * f(n - 1);
val f = fn : int -> int
> f 4;
val it = 24 : int
```

avrebbe invece dato il risultato atteso $f(4) = 24$.

Con quanto visto fino ad ora, dovrebbe a questo punto essere semplice implementare in ML funzioni ricorsive che effettuano le seguenti operazioni:

- Calcolo del fattoriale di un numero *usando ricorsione in coda* (Figura 9)
- Calcolo del massimo comun divisore fra due numeri (Figura 10)
- Soluzione del problema della torre di Hanoi (Figure 11 e 12)

```
fun move n from to via =
      if n = 0 then
                  "\n"
                else
                  (move (n - 1) from via to) ^
                  "Move disk from " ^ from ^ " to " ^ to ^
                  (move (n - 1) via to from);
```

Figure 11: Torre di Hanoi.

```
fun move n from to via =
      if n = 1 then
                  "Move disk from "^from^" to "^to^"\n"
                else
                  (move (n - 1) from via to) ^
                   move 1       from to  via ^
                  (move (n - 1) via to from);
```

Figure 12: Torre di Hanoi, versione alternativa.

Come spiegato precedentemente in questo documento, oltre a permettere di definire e valutare espressioni (associando eventualmente espressioni o valori a nomi, tramite il concetto di ambiente), ML permette di definire ed utilizzare nuovi *tipi di dato*. Il costrutto `type` permette di definire sinonimi per tipi di dato esistenti, mentre il costrutto `datatype` permette di definire nuovi tipi di dato (specificandone i costruttori che ne generano le varianti). Per esempio, si può definire un tipo `colore`, che rappresenta una componente di rosso, verde o blu (con l'intensità espressa come numero reale):

```
> rosso;
Error-Value or constructor (rosso) has not been declared   Found near rosso
Static errors (pass2)
> rosso 0.5;
Error-Value or constructor (rosso) has not been declared   Found near rosso(0.5)
Static errors (pass2)
> datatype colore = rosso of real | blu of real | verde of real;
datatype colore = blu of real | rosso of real | verde of real
> rosso;
val it = fn : real -> colore
> rosso 0.5;
val it = rosso 0.5 : colore
```

Prima della definizione "**datatype** colore = rosso **of** real blu of real — verde of real;—" la parola "`rosso`" non viene riconosciuta da Poly/ML, che lamenta l'utilizzo di un costruttore (`rosso`) che non è stato definito (vedere i primi due errori); dopo la definizione la parola "`rosso`" viene correttamente riconosciuta come un costruttore del tipo "`colore`" (funzione da `real` a `colore`).

Per finire, va notato come molti REPL di standard ML non siano particolarmente pratici da usare, perche' per esempio non supportano i tasti freccia e/o la storia delle espressioni immesse (funzionalità a cui siamo abituati lavorando con i moderni terminali). Questi problemi possono essere aggirati in due modi:

1. Usando il comando `rlwrap` per abilitare i tasti freccia e la storia dei comandi (forniti dalla libreria `readline`) nel prompt del REPL ML. Per esempio, Poly/ML può essere invocato tramite il comando "`rlwrap poly`"

2. Editando programmi ML complessi in file di testo (generalmente con estensione `.sml`), in modo da poter utilizzare le funzionalità di editor avanzati come `emacs`, `vi`, `gedit` o simili. Un programma contenuto in un file di testo può essere caricato dalla macchina astratta usando la direttiva `use` di ML: "use " file .sml";"

Come esempio di utilizzo di `use`, si può inserire il programma di Figura 12 nel file `hanoi.sml`, usando il poprio editor di testo preferito (per esempio `vi`). Il programma può poi essere caricato in Poly/ML con

```
> use "hanoi.sml";
val move = fn : int -> string -> string -> string -> string
val it = () : unit
>
```

La funzione `move` è adesso definita come se fosse stata inserita direttamente da tastiera.

L'utilizzo di `use` è anche utile per il debugging, perché segnala con più precisione gli errori sintattici, indicando la linea del programma in cui si è verificato l'errore.