Corso di linguaggi di programmazione, Modulo 1 - Linguaggio ML

Riccardo Pavetta

Marzo 2021

1 Introduzione

ML è un linguaggio funzionale, cioè basato sulla definizione e applicazioni di funzioni. Ogni funzione può essere trattata come un valore, in modo da poterla utilizzare come parametro di altre funzioni.

Side Effect Non ha side effect, quindi un'operazione non modifica il valore di una variabile, ma restituisce un nuovo elemento con il valore ottenuto.

Ricorsione In ML la ricorsione può essere implementata facilmente ed è molto efficiente, è quindi preferibile il suo utilizzo in sostituzione dei cicli while.

Fortemente tipizzato ML è un linguaggio fortemente tipizzato, i tipi di variabili e costanti vengono determinati automaticamente durante la compilazione. Una volta stabilito il tipo di una variabile non è più possibile cambiarlo. Normalmente non è necessaria la dichiarazione delle variabili.

Commenti In ML i commenti si fanno nel seguente modo: (* commento *)

2 Terminologia

Ogni oggetto è rappresentato da un nome. Alcuni nomi vengono assegnati dall'utente come i nomi di variabili, funzioni, parametri formali, tipi creati dall'utente... Altri vengono assegnati dal sistema: tipi ed operazioni primitive e costrutti predefiniti.

Binding L'associazione tra un nome e un'oggetto.

Environment L'insieme delle associazioni tra nomi e i relativi oggetti, che esistono durante un preciso momento, in una parte del codice nel tempo di esecuzione del programma.

Dichiarazione Un meccanismo che crea un'associazione in un ambiente. Grazie ai puntatori e alle chiamate per riferimento uno stesso nome può essere associato a più oggetti.

Blocchi Nei linguaggi moderni l'environment è diviso in blocchi, cioè porzioni del programma indentificate da tag di apertura e chiusura che contengono dichiarazioni locali, cioè che valgono solamente in quello scope. Tutte le dichiarazioni all'interno di un blocco sono valide anche in eventuali blocchi nidificati, a meno che all'interno di quest'ultimi non vengano dichiarati oggetti con lo stesso nome, sovrascrivendo la dichiarazione precedente.

3 Tipi di variabili

3.1 Unit

Il tipo unit è l'equivalente del tipo void negli altri linguaggi e viene rappresentato con ().

```
> ();
val it = (): unit
```

3.2 Interi

Per rappresentare i numeri negativi si utilizza il carattere ~, ad esempio ~3 equivale a -3. è possibile rappresentare anche i numeri in esadecimale.

```
> 0x124;
val it = 292: int
> ~0x124;
val it = ~292: int
```

3.3 Reali

Si può utilizzare la lettera E(e) per rappresentare delle potenze di 10.

```
> ~123.0;
val it = ~123.0: real
> 3E~3;
val it = 0.003: real
> 3.14e12;
val it = 3.14E12: real
```

3.4 Stringhe e caratteri

Le stringhe vengono rappresentate tra i doppi apici, i caratteri sempre con i doppi apici ma preceduti da #.

```
> #"a";
val it = #"a": char
> "foo";
val it = "foo": string
```

Esistono vari caratteri di escape:

```
\n: newline
\t: tab
\\: Backslash
\": Double-quote
\xyz: ASCII character with this code
\^x: Control character. ^G is the same as \007
```

è possibile concatenare stringhe tramite il simbolo: $\wedge$

```
> print ("foo" ^ "ciao ");
foociao val it = (): unit
```

Non è possibile concatenare due tipi diversi, nemmeno un char con una stringa. La funzione explode permette di trasformare una stringa in una lista di caratteri. La funzione implode permette di trasformare una lista di caratteri in una stringa.

```
> implode (explode ("xyz"));
val it = "xyz": string
```

3.5 Tuple

In ML è definito il tipo * che corrisponde alle tuple. Possono contenere elementi di tipi diversi al loro interno.

```
> (1,2);
val it = (1, 2): int * int

> val t = (4, 5.0, "six"): int * real * string;
```

Tramite #1 è possibile accedere al primo elemento della tupla, #2 al secondo e così via. ML inoltre permette la creazione di tuple che hanno tuple come elementi.

```
> (1,(2,3,4));
val it = (1, (2, 3, 4)): int * (int * int * int)
```

3.6 Liste

ML ha il tipo di dato lista. Tramite hd otteniamo il primo elemento, invece con tl otteniamo tutti gli elementi tranne il primo.

```
> val L = [1,2,3];
val L = [1, 2, 3]: int list

> hd(L);
val it = 1: int
> tl(L);
val it = [2,3]: int

> [];
val it = []: 'a list
lista vuota
```

Al contrario delle tuple, in una lista gli elementi devono essere tutti dello stesso tipo. Il carattere @ permette di concatenare 2 liste che devono essere necessariamente dello stesso tipo. Il carattere :: permette di aggiungere un elemento in testa.

```
> 2 :: [3,4];
val it = [2, 3, 4]: int list

> 2 :: nil;
val it = [2]: int list

> 2 :: [];
val it = [2]: int list

> 1 :: 2 :: 3 :: nil;
val it = [1, 2, 3]: int list
```

3.6.1 I tipi nelle liste

```
> ("ab", [1,2,3], 4);
val it = ("ab", [1, 2, 3], 4): string * int list * int

> [[(1,2),(3,4)], [(5,6)], nil];
val it = [[(1, 2), (3, 4)], [(5, 6)], []]: (int * int) list list
```

4 Operatori

Le operazioni possono essere svolte solo tra gli stessi tipi.

- / : viene utilizzato per le divisioni tra interi
- div : divisioni tra numeri reali(arrotonda per difetto)
- mod : modulo
- = : uguaglianza

```
> 1 = 1;
val it = true: bool

> "abc" <= "ab";
val it = false: bool

> "abc" <= "ac";
val it = true: bool

> #"Z" < #"a";
val it = true: bool
```

L'operatore di uguaglianza non può essere utilizzato per confrontare reali. Operatori sui booleani:

- orelse : or
- andalso : and
- not : !false
- = : uguaglianza
- <> : diverso (!=)

ML supporta la lazy evaluation, infatti:

```
1<2 orelse 3>4
```

Dato che $1 < 2$ da true la seconda parte$(3 > 4)$ non viene nemmeno valutata.

5 Variabili e costanti

Nell'esempio si può vedere la dichiarazione in ML, anche se il tipo non è richiesto perché viene calcolato in automatico.

```
> val a = 3;
> val b = 98.6;
> val a = "three";
> val c = a ^ str(chr(floor(b)));
val c = "threeb": string
```

Si noti come la 3 riga crei una nuova variabile di un tipo e valore diverso ma che ha lo stesso nome.

6 Conversioni

ML fornisce delle funzioni per la conversione di tipi, per la precisione viene creata una nuova variabile di un tipo diverso, infatti il tipo della variabile vecchia non può essere modificato.

- real(4) o real 4 : conversione da intero a reale
- da reale a intero
  - floor : prende l'intero più grande tra gli interi minori di quello inserito.
  - ceil : prende l'intero più piccolo tra gli interi maggiori di quello inserito.
  - round : arrotonda.
  - trunc : tronca l'intero inserito.
- ord : da carattere a intero.
- chr : da intero a carattere.
- str : da carattere a stringa.

```
> ord #"a";
val it = 97: int

> ord #"a" - ord #"A";
val it = 32: int

> str #"a";
val it = "a": string
```

7 if-then-else

```
> if 1<2 then 3+4 else 5+6;
val it = 7: int
```

Come tutto in ML anche il costrutto if-then-else è un'espressione che deve avere un valore. Per questo motivo l'else è obbligatorio e i risultati devono essere dello stesso tipo.

```
> if 1<2 then 3+4 else 5.0+6.0;
errato
```

8 Funzioni

Definizione di una funzione in ML.

```
> fun upper(c) = chr (ord(c) - 32);
val upper = fn: char -> char
```

ML gestisce il tipo della funzione automaticamente ma se è necessario possiamo specificarlo noi.

```
> fun square (x:real) = x * x;
val square = fn: real -> real
```

In ML le funzioni accettano solo un parametro che volendo può essere una tupla.

```
fun max3(a:real,b,c) = (* maximum of reals *)
if a>b then
if a>c then a else c
else
if b>c then b else c;
val max3 = fn: real * real * real -> real
max3(5.0,4.0,7.0);
val it = 7.0: real

val x=3;
fun addx(a) = a+x;
val x=10;
addx(2);
```

Quale sarà il valore stampato?
Sarà 5 perché ML prende in considerazione il valore di x al tempo della definizione della funzione, quindi 3 e non 10.

9 Ricorsione

Nei linguaggi funzionali, la ricorsione viene utilizzata per le istruzioni iterative.

```
> fun reverse L =
if L = nil then nil
else reverse (tl L) @ [hd L];
val reverse = fn: ''a list -> ''a list
```

Distinguiamo 2 tipi di ricorsione:

- Non lineare: una funzione richiama se stessa più volte.
- Reciproca: 2 funzioni si richiamano l'una con l'altra.

Notiamo che per dichiarare 2 funzioni utilizziamo la parola chiave and.

```
fun
take(L) =
if L = nil then nil
else hd(L) :: skip(tl(L))

and

skip(L) =
if L = nil then nil
else take(tl(L));
val skip = fn: ''a list -> ''a list
val take = fn: ''a list -> ''a list
```