Azzolini Riccardo 2021-03-08

Espressioni di base e valutazioni

1 Espressioni

Mentre i linguaggi imperativi sono basati su *istruzioni* che modificano lo stato della macchina astratta, i linguaggi funzionali si basano su **espressioni** la cui **valutazione** produce dei valori. In particolare, ogni linguaggio di programmazione funzionale fornisce:

- **espressioni primitive** — i letterali e le variabili (o meglio i "nomi definiti", perché non si tratta delle le variabili mutable dei linguaggi imperativi) — che costituiscono gli elementi di base nella definizione delle espressioni;
- **meccanismi di composizione** — operatori e metodi/funzioni — che permettono di costruire espressioni composte;
- **meccanismi di astrazione**, che consentono di introdurre nomi tramite i quali fare riferimento a espressioni definite.

2 Read-eval-print loop

Per i linguaggi funzionali è comune disporre di un interprete interattivo,[1] detto **read-eval-print loop** (**REPL**) perché è un programma che iterativamente (*loop*):

1. prende come input una singola espressione (*read*),
2. la valuta (*eval*),
3. mostra il risultato all'utente (*print*).

Per il linguaggio Scala è disponibile un REPL che può essere eseguito da riga di comando oppure tramite un IDE.

[1] Per linguaggi funzionali moderni esistono spesso sia un compilatore che un interprete.

3 Esempi di uso del REPL di Scala

Se si inserisce nel REPL un'espressione,[2] il REPL risponde indicandone il tipo e il valore:

```
scala> 12 + 24 * 2
res0: Int = 60
```

Ad esempio, in questo caso, `Int` rappresenta il tipo intero, e `60` è il valore dell'espressione. Inoltre, il REPL associa un nome al valore dell'espressione, qui `res0`, che in seguito può essere utilizzato per far riferimento a tale valore:

```
scala> res0 + 20
res1: Int = 80
```

All'interno del REPL è anche possibile definire dei nomi associati a delle espressioni, usando la parola riservata `def` seguita da un identificatore,[3] dal simbolo `=` e dall'espressione da associare:

```
scala> def radius = 10
radius: Int

scala> def pi = 3.14159
pi: Double
```

Si noti che questi non sono variabili (mutable), ma appunto nomi a cui sono associate delle espressioni non modificabili. Si osservi inoltre che i tipi di queste definizioni non sono specificati esplicitamente, ma piuttosto vengono dedotti dall'interprete in base ai tipi delle espressioni specificate alla destra dell'uguale; dopo la valutazione di ciascuna definizione, il REPL ne mostra il tipo.

I nomi definiti possono essere usati in un'espressione:

```
scala> radius * pi
res2: Double = 31.4159
```

Il risultato di quest'espressione ha tipo `Double` perché `radius` ha tipo `Int` e `pi` ha tipo `Double`, dunque, come in Java, l'operatore `*` applica una conversione implicita al valore di `radius` e poi esegue la moltiplicazione tra due valori `Double`.

[2] I caratteri `scala>` non sono parte dell'input da scrivere, ma piuttosto rappresentano il prompt del REPL; in questi esempi, tale prompt serve per indicare che ciò che segue sulla stessa riga è un input inserito dall'utente, e non un output del programma.

[3] Le regole sui caratteri che possono formare un identificatore sono uguali a quelle di Java.

4 Valutazione

Considerando solo gli elementi delle espressioni visti finora, la **valutazione** (**evaluation**) di un'espressione avviene nel modo seguente:

1. Si considera l'operatore più a sinistra, rispettando le regole di precedenza.
2. Si valutano ricorsivamente i suoi operandi. Un operando che è un nome viene valutato rimpiazzandolo (*rewriting*) con il lato destro della sua definizione.
3. Si applica l'operatore agli operandi per determinare il valore risultante.

Il processo di valutazione si arresta quando si ottiene un singolo valore.

La valutazione di un'espressione può essere descritta come una sequenza di passi (indicati con $\rightarrow$), ciascuno dei quali corrisponde all'esecuzione di una singola operazione: o il rewriting di un nome, o la valutazione di un operatore i cui operandi sono già stati valutati. Ad esempio, date le definizioni `radius` e `pi` di prima, l'espressione `(2 * pi) * radius` viene valutata tramite i seguenti passi:

`(2 * pi) * radius`	
$\rightarrow$ `(2 * 3.14159) * radius`	[rewriting nome `pi`]
$\rightarrow$ `6.28318 * radius`	[valutazione operatore `*`]
$\rightarrow$ `6.28318 * 10`	[rewriting nome `radius`]
$\rightarrow$ `62.8318`	[valutazione operatore `*`]

Si osservi che il processo di valutazione lavora quasi solo sulla sintassi dell'espressione, trasformando una stringa (che rappresenta un'espressione) in un'altra secondo determinate regole, tra cui il rewriting dei nomi. L'unica eccezione è la valutazione dell'operatore `*`, che per semplicità viene trattato come un'operazione elementare, eseguita in un passo solo, ma volendo si potrebbe ricondurre anch'essa a una sequenza di trasformazioni sintattiche su una determinata rappresentazione dei valori numerici, che (come già detto) è proprio ciò che fanno internamente i calcolatori.

5 Esempi di definizione di nomi con parametri

È possibile definire nomi che dipendono da uno o più parametri. Ad esempio:

```
scala> def square(x: Double) = x * x
square: (x: Double)Double
```

In questa definizione, il nome `x` costituisce un **parametro formale**,[4] che serve a identificare nel "corpo" della definizione (l'espressione a destra dell'uguale) il valore che passato come argomento al momento dell'uso di questo nome. Come al solito, quando si inserisce la definizione il REPL risponde indicandone il tipo; in questo caso, in particolare, viene indicato il tipo dei parametri, scritto tra parentesi, `(x : Double)`, seguito poi dal tipo del risultato, `Double`.[5]

Un nome con parametri corrisponde al concetto matematico di **funzione**: il prodotto cartesiano dei tipi dei parametri è il dominio, e il tipo del risultato è il codominio. Allora, `square: (x: Double)Double` corrisponde a una funzione `square` : `Double` → `Double`.

Per usare una funzione in un'espressione, bisogna specificare i **parametri attuali** tramite delle espressioni indicate come argomenti:

```
scala> square(2)
res4: Double = 4.0
```

I parametri attuali possono essere espressioni arbitrariamente complesse:

```
scala> square(2 + 3)
res5: Double = 25.0

scala> square(square(4))
res6: Double = 256.0
```

Un esempio di definizione di una funzione con due parametri è il seguente:

```
scala> def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares: (x: Double, y: Double)Double
```

Tale definizione corrisponde a una funzione

$$\texttt{sumOfSquares} : \texttt{Double} \times \texttt{Double} \to \texttt{Double}$$

Si osservi che nel corpo della funzione possono essere usate le funzioni precedentemente definite, come appunto `square`.

Una volta definita, `sumOfSquares` può ovviamente essere utilizzata:

```
scala> sumOfSquares(2, 3)
res7: Double = 13.0
```

[4] Il termine *formale* si riferisce al fatto che `x` è solo un nome "di comodo", è pura *forma*: scegliere un nome diverso non cambierebbe in alcun modo il comportamento del programma.

[5] A partire dalla versione 2.13.2 di Scala, la notazione usata nell'output dell'interprete è stata sostituita con una più simile alla sintassi del codice sorgente: ad esempio, in questo caso l'interprete stampa `def square(x: Double): Double`.

6 Forma generale della definizione di funzioni

In generale, la definizione di una funzione ha la forma

```
def name(p1: Type1, ..., pN: TypeN) = expr
```

oppure

```
def name(p1: Type1, ..., pN: TypeN): ReturnType = expr
```

- *name* è il nome della funzione.
- I parametri formali della funzione vengono specificati tra le parentesi mediante la sintassi *pi*: *Typei*, dove *pi* è il nome di un parametro e *Typei* è il suo tipo.
- Il tipo del valore restituito, *ReturnType*, può essere omesso nei casi in cui il compilatore/interprete Scala è in grado di dedurlo (in base ai tipi dei parametri formali e al corpo della funzione).
- L'espressione *expr* costituisce il corpo della funzione; se *ReturnType* è specificato esplicitamente, *expr* deve essere un'espressione il cui tipo coincide con (o è promivibile a) *ReturnType*.

Più avanti si vedrà quali siano i possibili tipi dei parametri formali e del valore restituito, ma per ora verranno considerati solo i tipi base.

7 Tipi base di Scala

I **tipi base** (*basic types*) di Scala corrispondono all'insieme dei tipi primitivi di Java (indicati però con la lettera iniziale maiuscola, perché Scala li gestisce come oggetti veri e propri, piuttosto che come tipi primitivi), più il tipo `String`. In particolare, essi sono: `Boolean`, `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Char` e `String`. Di questi, si definiscono *tipi interi* i tipi `Byte`, `Short`, `Int` e `Long`, mentre i *tipi numerici* sono i tipi interi più `Float` e `Double`.

I valori di tutti i tipi base possono essere rappresentati per mezzo di appositi **letterali**, scritti con la stessa sintassi dei corrispondenti tipi Java.

I tipi base sono sempre disponibili, perché appartengono a dei package che vengono automaticamente importati in ogni sorgente Scala e nel REPL:

- `String` appartiene al package `java.lang` (è proprio lo stesso tipo `String` usato in Java);
- gli altri tipi base appartengono al package `scala`.