Azzolini Riccardo 2021-03-11

Espressioni booleane e condizionali

1 Espressioni booleane

Il tipo `Boolean` di Scala, che corrisponde al tipo `boolean` di Java, ha due possibili valori, *true* e *false*, rappresentabili dai letterali `true` e `false`. Come in Java, le espressioni booleane sono composte:[1]

- dalle costanti, cioè i letterali, `true` e `false`;
- dall'operatore di negazione, `!b`;
- dall'operatore di congiunzione, `b && b`;
- dall'operatore di disgiunzione, `b || b`;
- dalle usuali condizioni semplici, costruite a partire dagli operatori di confronto (`e == e`, `e != e`, `e <= e`, `e >= e`, `e < e`, `e > e`), applicabili a operandi di tipi uguali o compatibili (nel caso dei tipi di base, le regole di compatibilità sono analoghe a quelle di Java).

Al fine di applicare il modello di sostituzione alle espressioni booleane, bisogna specificare le regole di riduzione (riscrittura) per gli operatori booleani, che sono le seguenti (dove `b` è un'arbitraria espressione booleana):

(1) `!true` → `false`
(2) `!false` → `true`
(3) `true && e` → `e`
(4) `false && e` → `false`
(5) `true || e` → `true`
(6) `false || e` → `e`

Si noti che nei casi (4) e (5) l'operando destro non viene valutato; questo comportamento, comune anche nei linguaggi imperativi, prende il nome di *lazy evaluation* o *short-circuit evaluation*.

[1] Qui `b` rappresenta un'arbitraria espressione booleana, ed `e` rappresenta un'arbitraria espressione di tipo confrontabile con il tipo dell'altro operando dell'operatore di confronto.

2 Espressioni condizionali

In Scala esiste un costrutto `if-else` che esprime la scelta tra due alternative; esso ha una forma simile a quella di Java, ma invece che un'istruzione è un'*espressione*, chiamata **espressione condizionale**, che viene valutata e produce un valore. In particolare, la sintassi di questo costrutto è

`if (`*predicate*`)` *then-expr*

oppure

`if (`*predicate*`)` *then-expr* `else` *else-expr*

dove:

- *predicate* è un'espressione booleana (la condizione, che in ambito di programmazione funzionale viene più spesso chiamata *predicato*);
- *then-expr* ed *else-expr* sono espressioni di tipi tra loro compatibili.

Intuitivamente, il risultato della valutazione di un'espressione condizionale è il valore di *then-expr* se *predicate* ha valore `true`, mentre è il valore di *else-expr* se *predicate* ha valore `false`. Cosa avvenga esattamente quando *predicate* è falso e non viene specificata la *else-expr* verrà spiegato più avanti, ma semplificando si potrebbe dire che l'espressione "non restituisce nulla".

Un esempio di espressione condizionale è il corpo della seguente funzione, che calcola il valore assoluto di un numero intero:

```
def abs(x: Int) = if (x >= 0) x else -x
```

Formalmente, la valutazione di un'espressione `if-else` (per ora, non verrà considerato il caso di `if` senza `else`) dipende dal valore del predicato, secondo le seguenti regole di riscrittura:

(1) `if (true)` *e1* `else` *e2* $\rightarrow$ *e1*

(2) `if (false)` *e1* `else` *e2* $\rightarrow$ *e2*

Si osservi che in entrambi i casi viene valutata solo una delle due espressioni *e1* ed *e2*.

3 Definizioni per nome e per valore

Così come la valutazione degli argomenti di una funzione può avvenire per nome o per valore, lo stesso è possibile per le definizioni di nomi senza parametri.

Quella vista finora è la **definizione per nome**,

`def` *name* `=` *expr*

il cui corpo *expr* viene valutato ogni volta che si utilizza *name*.

Invece, una **definizione per valore** viene scritta con la sintassi

```
val name = expr
```

cioè usando la parola riservata `val` invece di `def`. In questa forma, il corpo *expr* viene valutato una volta sola al momento della definizione, dopodiché *name* viene rimpiazzato direttamente dal suo valore ogni volta che viene utilizzato. Allora, una definizione per valore può essere più efficienti, soprattutto quando il corpo è un'espressione complessa e il nome che si definisce viene poi usato spesso. Ad esempio, date le definizioni

```
def square(x: Int) = x * x
val x = 2
val y = square(x)
```

in ogni utilizzo il nome `y` farà riferimento al valore `4`, e non all'espressione `square(x)`, quindi non sarà necessario ricalcolare ogni volta il valore di quest'ultima.

La differenza tra i comportamenti di `def` e `val` può essere messa in evidenza considerando la definizione di `loop`, già vista nel caso `def`:

- La definizione

  ```
  def loop: Boolean = loop
  ```

  non causa problemi immediatamente: l'interprete va in loop solo quando il nome viene effettivamente usato all'interno di un'espressione valutata.

- La definizione

  ```
  val loop: Boolean = loop
  ```

  manderebbe immediatamente in loop l'interprete (se non fosse per alcuni dettagli della specifica del linguaggio Scala e del funzionamento della JVM, secondo i quali questa definizione ricorsiva non è ammessa in alcuni contesti, e dove invece è ammessa associa al nome `loop` un valore di default, ad esempio `false` nel caso del tipo `Boolean`).