Azzolini Riccardo 2021-03-15

Blocchi e scope lessicale

1 Altro esempio di iterazione: `sqrt` con il metodo di Newton

Si vuole definire una funzione `sqrt` che calcoli la radice quadrata di un numero reale, rappresentato in formato `Double`:

```
def sqrt(x: Double): Double = ...
```

Un algoritmo classico per il calcolo della radice quadrata è quello basato sul metodo di Newton.

1.1 Metodo di Netwon

In generale, il *metodo di Newton* è un metodo per il calcolo approssimato delle radici (gli zeri) di una funzione $f : \mathbb{R} \to \mathbb{R}$, cioè delle soluzioni dell'equazione $f(x) = 0$. Per poterlo applicare, è prima necessario determinare un intervallo $[a, b]$ che contiene una sola radice. Poi, si inizia assumendo un valore iniziale ("guess") della radice, $x_0 \in [a, b]$, e da questa si calcolano iterativamente delle approssimazioni sempre migliori: per ogni $k \geq 0$, l'approssimazione x_{k+1} è calcolata a partire dall'approssimazione x_k tramite la formula

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

(dove f' è la derivata di f). Si può dimostrare che, se sono soddisfatte le condizioni per l'applicazione di questo metodo, allora $\lim_{n\to\infty} x_n = x$ tale che $f(x) = 0$.

1.2 Applicazione alla radice quadrata

Per applicare il metodo di Newton al calcolo della radice quadrata di un numero reale $z \geq 0$, bisogna innanzitutto individuare una funzione la cui radice (zero) sia $\sqrt{z}$. $\sqrt{z}$ è la radice positiva della funzione $f(x) = x^2 - z$, perché $f(\sqrt{z}) = (\sqrt{z})^2 - z = z - z = 0$ per ogni $z \geq 0$. Questa funzione ha una sola radice positiva, cioè un solo zero nell'intervallo $[0, \infty)$, quindi si può prendere come valore iniziale un qualunque valore positivo, $x_0 \geq$

0. Rimane ora da determinare la formula per il calcolo dell'approssimazione x_{k+1}; la derivata di $f(x)$ è $f'(x) = 2x$, quindi la formula è:

$$\begin{aligned} x_{k+1} &= x_k - \frac{f(x_k)}{f'(x_k)} = x_k - \frac{x_k^2 - z}{2x_k} \\ &= x_k - \frac{x_k^2}{2x_k} + \frac{z}{2x_k} = x_k - \frac{1}{2}x_k + \frac{z}{2x_k} \\ &= \frac{1}{2}x_k + \frac{z}{2x_k} = \frac{1}{2}\left(x_k + \frac{z}{x_k}\right) \\ \boxed{x_{k+1} &= \frac{1}{2}\left(x_k + \frac{z}{x_k}\right)} \end{aligned}$$

Ad esempio, per $z = 2$, scegliendo il valore iniziale $x_0 = z = 2$ il calcolo procede in questo modo:

k	Guess	Improve guess
0	$x_0 = 2$	$x_1 = \frac{1}{2}(2 + \frac{2}{2}) = 1.5$
1	$x_1 = 1.5$	$x_2 = \frac{1}{2}(1.5 + \frac{2}{1.5}) = 1.4167$
2	$x_2 = 1.4167$	$x_3 = \frac{1}{2}(1.4167 + \frac{2}{1.4167}) = 1.4142$
3	$x_3 = 1.4142$	...

Le approssimazioni successive convergono molto rapidamente, ma il risultato esatto sarebbe ottenuto solo ripetendo il processo all'infinito, quindi in pratica bisogna fermarsi quando si ritiene che la soluzione ottenuta sia un'approssimazione *abbastanza buona*. Se *errore* è il valore che stabilisce il massimo errore relativo ammesso sulla soluzione, allora una soluzione x_k è considerata abbastanza buona se

$$\frac{|x_k^2 - z|}{z} < errore$$

1.3 Implementazione

A differenza degli esempi visti in precedenza, il metodo di Newton è un algoritmo che, anche in ambito matematico, viene tipicamente definito in modo iterativo. Allora, un'implementazione iterativa in Java potrebbe essere la seguente:

```
private static double ERROR = 1e-15;

private static boolean isGoodEnough(double guess, double z) {
    return Math.abs(guess * guess - z) / z < ERROR;
}
```

```
private static double improveGuess(double guess, double z) {
    return (guess + z / guess) / 2;
}

public static double sqrt(double z) {
    double guess = z;
    while (!isGoodEnough(guess, z))
        guess = improveGuess(guess, z);
    return guess;
}
```

Qui il ciclo è stato fattorizzato in due metodi: uno che determina se la soluzione attuale è abbastanza buona, e uno che raffina la soluzione applicando un passo del metodo di Newton. La costante `ERROR`, che fissa il massimo errore relativo consentito, è stata impostata a 10^{-15} perché questo valore è una buona approssimazione della precisione del tipo `double`.

L'implementazione iterativa può essere convertita in una ricorsiva introducendo un nuovo parametro formale per simulare la variabile `guess`:

```
private static double sqrtRec(double guess, double z) {
    if (isGoodEnough(guess, z))
        return guess;
    else
        return sqrtRec(improveGuess(guess, z), z);
}

public static double sqrt(double z) {
    return sqrtRec(z, z);
}
```

L'implementazione ricorsiva può poi essere riscritta in Scala:

```
val ERROR = 1e-15

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

def isGoodEnough(guess: Double, z: Double): Boolean =
  abs(guess * guess - z) / z < ERROR

def improveGuess(guess: Double, z: Double): Double =
  (guess + z / guess) / 2

def sqrtRec(guess: Double, z: Double): Double =
  if (isGoodEnough(guess, z))
```

```
    guess
  else
    sqrtRec(improveGuess(guess, z), z)

def sqrt(z: Double): Double = sqrtRec(z, z)
```

Si noti che la funzione `abs` è disponibile anche nella libreria standard di Scala, così come in quella di Java, ma qui è stata definita poiché, per semplicità, si preferisce evitare per ora l'uso di librerie negli esempi.

2 Funzioni innestate

Nella programmazione è buona pratica suddividere un task (una funzione, nel caso della programmazione funzionale) in sotto-task (funzioni) "piccoli" (come è stato fatto nell'esempio precedente) per facilitare la gestione della complessità, aiutando in particolare a:

- evidenziare la logica delle varie parti che compongono il task;
- aumentare il controllo sulla correttezza del codice.

A volte, la suddivisione in sotto-task ha anche il vantaggio di rendere riutilizzabili le funzioni che implementano i sotto-task, le quali potrebbero essere utili anche in altri contesti. Spesso, invece, accade il contrario: le funzioni introdotte per realizzare i sotto-task sono funzionali allo specifico frammento di codice in cui vengono utilizzate, e non sarebbero utili al di fuori di tale contesto. Ad esempio, nel codice per il calcolo di `sqrt`, le funzioni `sqrtIter`, `improveGuess` e `isGoodEnough` sono necessarie solo per l'implementazione della funzione `sqrt`, e non per il suo utilizzo da parte dell'utente, che non ha bisogno di accedere a esse. Allora, il fatto che tali funzioni siano visibili in modo analogo a `sqrt` ha come effetto la **name-space pollution**, cioè "sporca" lo spazio dei nomi delle funzioni con definizioni che per l'utente sono inutili. Come soluzione a questo problema, Scala permette di introdurre *definizioni locali a una funzione*.[1] Ad esempio, il codice di `sqrt` potrebbe essere riscritto come segue:

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

def sqrt(z: Double): Double = {
  val ERROR = 1e-15

  def isGoodEnough(guess: Double, z: Double): Boolean =
    abs(guess * guess - z) / z < ERROR
```

[1] Questo meccanismo è in generale molto comune nei linguaggi funzionali, mentre è meno diffuso nei linguaggi imperativi. Ad esempio, in Java non è possibile definire un metodo all'interno di un altro, quindi ogni metodi di una classe è come minimo visibile in tutta la classe.

```
  def improveGuess(guess: Double, z: Double): Double =
    (guess + z / guess) / 2

  def sqrtRec(guess: Double, z: Double): Double =
    if (isGoodEnough(guess, z))
      guess
    else
      sqrtRec(improveGuess(guess, z), z)

  sqrtRec(z, z)
}
```

3 Blocchi

Un **blocco** è un frammento di codice delimitato da parentesi graffe: `{...}`. In Scala, un blocco è un'*espressione*, che può comparire ovunque siano ammesse le espressioni, cioè come corpo di una definizione, ramo di un'espressione condizionale, argomento di una funzione o di un operatore, ecc. (mentre in Java, ad esempio, un blocco è un'istruzione).

Un blocco contiene una sequenza di *definizioni* ed *espressioni*; l'ultimo elemento di un blocco è obbligatoriamente un'espressione, che definisce il *valore* restituito dalla valutazione del blocco.

4 Regole di scope nei blocchi

I blocchi hanno effetti sulla visibilità degli identificatori:

- le definizioni (`def` e `val`) scritte all'interno di un blocco sono *visibili esclusivamente all'interno del blocco*;
- le definizioni all'interno di un blocco **adombrano** (**shadow**) eventuali definizioni degli stessi nomi che compaiono all'esterno del blocco.

Ad esempio, nel codice

```
val x = 0
def f(y: Int) = y + 1

val result = {
  val x = f(3)
  x * x
}
```

l'invocazione di funzione `f(3)` fa riferimento alla funzione `f` definita fuori dal blocco, mentre l'uso del nome `x` nell'espressione `x * x` fa riferimento alla definizione di `x` interna al blocco, `val x = f(3)`, che adombra la definizione esterna `val x = 0`.

5 Lexical scoping

In generale, il termine **lexical scoping** (*scope lessicale*) indica che le regole che determinano la visibilità degli identificatori dipendono solo da come il codice è scritto, e non da ciò che avviene un fase di esecuzione. Nel linguaggio Scala (come anche Java) si ha appunto lo scope lessicale, mentre ci sono altri linguaggio che usano regole di scope semantiche, legate ai meccanismi di esecuzione del codice.

Tuttavia, specificare che un linguaggio ha regole di scope lessicali non è sufficiente a descrivere completamente tali regole, poiché anche solo nell'ambito dello scope lessicale il progettista di un linguaggio può effettuare diverse scelte di regole (ad esempio, può essere consentito o meno l'adombramento).

Una delle regole di scope di Scala stabilisce che le definizioni esterne a un blocco sono visibili all'interno di un blocco. Ciò è interessante soprattutto nel caso delle funzioni: i parametri formali di una funzione sono visibili nel corpo della funzione, e quindi anche *all'interno dei blocchi che compaiono nel corpo* della funzione, comprese eventuali definizioni di funzioni innestate. Questo significa che, oltre a evitare la name-space pollution, le definizioni innestate fungono anche da meccanismo di semplificazione del codice, perché si può evitare di passare come argomenti alle funzioni innestate i parametri che sono visibili nel corpo della funzione e che non devono cambiare valore in eventuali chiamate ricorsive.

Ad esempio, si consideri l'implementazione della funzione `sqrt` con le funzioni innestate:

```
def sqrt(z: Double): Double = {
  val ERROR = 1e-15

  def isGoodEnough(guess: Double, z: Double): Boolean =
    abs(guess * guess - z) / z < ERROR

  def improveGuess(guess: Double, z: Double): Double =
    (guess + z / guess) / 2

  def sqrtRec(guess: Double, z: Double): Double =
    if (isGoodEnough(guess, z))
      guess
    else
      sqrtRec(improveGuess(guess, z), z)
```

```
  sqrtRec(z, z)
}
```

Il parametro formale `z` di `sqrt` viene passato alle varie funzioni innestate, e il suo valore rimane sempre costante in tutte le chiamate (non viene mai modificato all'interno del corpo della funzione `sqrt`). Allora, dato che `z` è già visibile nel corpo di `sqrt`, si può evitare di passarlo come parametro alle funzioni innestate:

```
def sqrt(z: Double): Double = {
  val ERROR = 1e-15

  def isGoodEnough(guess: Double): Boolean =
    abs(guess * guess - z) / z < ERROR

  def improveGuess(guess: Double): Double =
    (guess + z / guess) / 2

  def sqrtRec(guess: Double): Double =
    if (isGoodEnough(guess))
      guess
    else
      sqrtRec(improveGuess(guess))

  sqrtRec(z)
}
```

6 Punto e virgola

In Scala, a differenza di quanto accade in Java, il punto e virgola (semicolon, `;`) a fine riga è in genere opzionale: è possibile scrivere, ad esempio,

```
val x = 1;
```

ma il punto e virgola è ridondante, dunque si preferisce non metterlo. Invece, esso va utilizzato se si vogliono scrivere più espressioni e/o definizioni su una stessa riga, che infatti devono essere separate da `;`, come nel seguente esempio:

```
val y = x + 1; y * y
```

Il fatto che le espressioni terminino implicitamente a fine riga ha alcune conseguenze su come devono essere scritte le espressioni con operatori infissi su più linee. In particolare, il frammento di codice

```
someLongExpression
+ someOtherExpression
```

viene interpretato come due espressioni separate:

```
someLongExpression;
+ someOtherExpression
```

Se questo è il significato desiderato dal programmatore, è bene mettere un `;` esplicito per chiarirlo. Altrimenti, se (come più spesso capita) si vuole che le due righe siano interpretate come un'unica espressione, ci sono due soluzioni:

- si può racchiudere l'espressione su più linee tra parentesi tonde (il che funziona perché `;` non viene mai inserito tra parentesi tonde):

  ```
  ( someLongExpression
    + someOtherExpression )
  ```

- si può scrivere l'operatore infisso alla fine della prima riga invece che all'inizio della seconda (così, vedendo che manca ancora l'operando destro, il compilatore Scala si accorge che l'espressione non è finita):

  ```
  someLongExpression +
  someOtherExpression
  ```