Blocks and Lexical Scope

August 31, 2012

Nested functions

It's good functional programming style to split up a task into many small functions.

But the names of functions like `sqrtIter`, `improve`, and `isGoodEnough` matter only for the *implementation* of `sqrt`, not for its *usage*.

Normally we would not like users to access these functions directly.

We can achieve this and at the same time avoid "name-space pollution" by putting the auxciliary functions inside `sqrt`.

The sqrt Function, Take 2

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x)

  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double, x: Double) =
    abs(square(guess) - x) < 0.001

  sqrtIter(1.0, x)
}
```

Blocks in Scala

- A block is delimited by braces `{ ... }`.

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

- It contains a sequence of definitions or expressions.
- The last element of a block is an expression that defines its value.
- This return expression can be preceded by auxiliary definitions.
- Blocks are themselves expressions; a block may appear everywhere an expression can.

Blocks and Visibility

```
val x = 0
def f(y: Int) = y + 1
val result = {
  val x = f(3)
  x * x
}
```

- The definitions inside a block are only visible from within the block.
- The definitions inside a block *shadow* definitions of the same names outside the block.

Exercise: Scope Rules

Question: What is the value of `result` in the following program?

```
val x = 0
def f(y: Int) = y + 1
val result = {
  val x = f(3)
  x * x
} + x
```

Possible answers:

- 0 `0`
- 0 `16`
- 0 `32`
- 0 `reduction does not terminate`

Lexical Scoping

Definitions of outer blocks are visible inside a block unless they are shadowed.

Therefore, we can simplify `sqrt` by eliminating redundant occurrences of the `x` parameter, which means everywhere the same thing:

The sqrt Function, Take 3

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double): Double =
    if (isGoodEnough(guess)) guess
    else sqrtIter(improve(guess))

  def improve(guess: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double) =
    abs(square(guess) - x) < 0.001

  sqrtIter(1.0)
}
```

Semicolons

In Scala, semicolons at the end of lines are in most cases optional

You could write

```
val x = 1;
```

but most people would omit the semicolon.

On the other hand, if there are more than one statements on a line, they need to be separated by semicolons:

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

Semicolons and infix operators

One issue with Scala's semicolon convention is how to write expressions that span several lines. For instance

```
someLongExpression
+ someOtherExpression
```

would be interpreted as *two* expressions:

```
someLongExpression;
+ someOtherExpression
```

Semicolons and infix operators

There are two ways to overcome this problem.

You could write the multi-line expression in parentheses, because semicolons are never inserted inside `(...)`:

```
(someLongExpression
 + someOtherExpression)
```

Or you could write the operator on the first line, because this tells the Scala compiler that the expression is not yet finished:

```
someLongExpression +
someOtherExpression
```

Summary

You have seen simple elements of functional programing in Scala.

- arithmetic and boolean expressions
- conditional expressions if-else
- functions with recursion
- nesting and lexical scope

You have learned the difference between the call-by-name and call-by-value evaluation strategies.

You have learned a way to reason about program execution: reduce expressions using the substitution model.

This model will be an important tool for the coming sessions.