



Blocks and Lexical Scope

Principles of Functional Programming

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 auxiliary functions inside `sqrt`.

The sqrt Function, Take 2

```
def sqrt(x: Double) = {  
  def sqrtIter(guess: Double, x: Double): Double =  
    if isGoodEnough(guess, x) then 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)  
}
```

The sqrt Function, Take 2

```
def sqrt(x: Double) =  
  def sqrtIter(guess: Double, x: Double): Double =  
    if isGoodEnough(guess, x) then 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)
```

In Scala 3, braces are optional for indented code.

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.
- ▶ In Scala 3, braces are optional (i.e. implied) around a correctly indented expression that appears after =, then, else,

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
 - 0
- 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) then 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)
```

End Markers

With heavily indented code it is sometimes hard to see where a construct ends.

End markers are a tool to make this explicit.

```
def f() =
```

```
  ...
```

```
    ...
```

```
      ...
```

```
end f
```

- ▶ An end marker is followed by the name that's defined in the definition that ends at this point.
- ▶ It must align with the opening keyword (def in this case).

The sqrt Function, Take 4

```
def sqrt(x: Double) =  
  def sqrtIter(guess: Double): Double =  
    if isGoodEnough(guess) then 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)  
end sqrt
```

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
```

Summary

You have seen simple elements of functional programming in Scala.

- ▶ arithmetic and boolean expressions
- ▶ conditional expressions if-then-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.