Programming Paradigms

Paradigm: In science, a *paradigm* describes distinct concepts or thought patterns in some scientific discipline.

Main programming paradigms:

- imperative programming
- functional programming
- logic programming

互不相關的，正交的

Orthogonal to it:

- object-oriented programming

Welcome to my course on function programming principles in Scala. As the name implies we're going to do quite a bit of scala programming, but that's not the primary objective of the course. The primary objective is to teach you functional programming from first principles. You're going to see function programs, methods to construct them, and ways to reason about them. You're gonna find that functional programming is a paradigm that's quite different from the classical imperative(命令的) paradigm that you know from languages such as Java or C. In fact, you can combine the two paradigms, and it's one of Scala's strengths, that it provides a gradual migration path from a more concise travel-light language to full-functional programming. But in this course we're not going to be gradual, we're going to take a clean break. I'd like to, you to suspend for the time being most of what you know about programming and look at programs with fresh eyes. That way, I believe you'll be better able to absorb this new way of thinking, that's function programming, and once you've done that you'll also be able to integrate that back into your daily programming practice. In face, Scala is a great tool both to learn function programming from first principles and later on to integrate it with classical object oriented programming. So let's get started. So I have talked about functional programming as a different paradigm. If we go back to the meaning of the word, in science, a paradigm describes distant concepts or thought patterns in some scientific discipline. In programming we can distinguish three main paradigms. There is imperative programming. That's what you probably know from languages such as Java or C. There's functional programming. And there is a lesser known one called logic programming. Some people call object oriented programming a paradigm. But in my mind, it's really something that's orthogonal to the three paradigms here. In the sense that it can be combined well with both imperative programming. That's what you, essentially the current state of the art. But also with functional programming or even logic programming.

Review: Imperative programming

Imperative programming is about

- modifying mutable variables,
- using assignments
- and control structures such as if-then-else, loops, break, continue, return.

The most common informal way to understand imperative programs is as instruction sequences for a Von Neumann computer.

So let's review what imperative programming is as a paradigm. Imperative programming is really about modifying mutable variables using assignments. And those will be composed with control structures such if then L's, loops, right continue, return, and so on. The most common informal way to understand imperative programs is as instruction sequences for a Von Neumann computer. Von Neuman was one of the pioneers of the computer, built the first computer around 1945. So a Von Neumann computer, it consists essentially of a processor and memory, and then there's a bus that reads both instructions and data. From the memory into the processor. And what's important is this, that the width of that bus is about one machine word so 32 bits or 64 bits nowadays.

Imperative Programs and Computers

There's a strong correspondence between

Mutable variables	≈	memory cells
Variable dereferences	≈	load instructions
Variable assignments	≈	store instructions
Control structures	≈	jumps

Problem: Scaling up. How can we avoid conceptualizing programs word by word?

Reference: John Backus, Can Programming Be Liberated from the von. Neumann Style?, Turing Award Lecture 1978.

Now it turns out that, that model of a computer has shaped programming to no small degree. There's a very strong correspondence between the memory cells of a Von Neumann computer and the mutable variables in a programming language. Variable de-references correspond them to load instructions in the computer. Variable assignments relate to store instructions. Controlled structures all translate into a sequence of loops. So that's all very well, but the scaling is getting up. We want to avoid conceptualizing programs just word by word. We want to reason in larger structures. That was the argument made by John Backus, in his Turing award lecture, titled Can Programming Be Liberated From the Von Neumann Style. It's noteworthy that John Backus was in fact the inventor of the first high level language at all. It was Fortran in the 1950's, so more than twenty years later he found that the traditional course of action of imperative programming had run it's course, was running out of steam, and that something new was needed. And the new thing that he was proposing then was function programming.

Scaling Up

In the end, pure imperative programming is limited by the "Von Neumann" bottleneck:

此 word 應該是字節的意思 .

One tends to conceptualize data structures word-by-word.

We need other techniques for defining high-level abstractions such as collections, polynomials, geometric shapes, strings, documents.

Ideally: Develop *theories* of collections, shapes, strings, ...

So John Backus argued that in the end pure imperative program it is limited by the Von Neumann bottleneck, which means, which means that we conceptualize data structures word by word. If you want to scale up, we'll need to define higher level abstractions such as collections, polynomials, geometric shapes, strings, documents, and so on. And to be thorough, we need to have theories of these higher level abstractions, collections, shapes, and so on so that we are able to reason about them.

What is a Theory?

A theory consists of

- one or more data types
- operations on these types
- laws that describe the relationships between values and operations

Normally, a theory does not describe mutations!

So what is a theory? In mathematics, a theory consists of one or more data types, operations on these types, and laws that describe the relationships between the values and the operations. So what's important is that a theory mathematics does not describe mutations. A mutation means that I have changed something while keeping the identity of the thing the same.

Theories without Mutation

For instance the theory of polynomials defines the sum of two polynomials by laws such as:

```
(a*x + b) + (c*x + d) = (a+c)*x + (b+d)
```

But it does not define an operator to change a coefficient while keeping the polynomial the same!

Whereas in an imperative program one *can* write:

```
class Polynomial { double[] coefficient; }
Polynomial p = ...;
p.coefficient[0] = 42;
```

So for instance the theory of polynomials describes the sum of two polynomials by laws such as this one here. Here we say to sum two polynomial of degree one, we Take the two coefficient of the same degree and the sum of those coefficients, and there would be laws of all the other useful operators for polynomials. But one thing the theory does not do is define an operator to change a coefficient while keeping the polynomial the same. Whereas if we look at programming, imperative programming, one can do precisely that. One can write a class polynomial that would have an array of tables containing the coefficients. One can then define a concrete polynomial p and one can set the coefficient zero of that polynomial to 42. When one does that the polynomial p is still the same. That in mathematics simply is not available. It would detract(損壞，降低) from the theory of, in mathematics and in fact would, could damage this theory by breaking laws.

Theories without Mutation

Other example:

The theory of strings defines a concatenation operator ++ which is associative:

```
(a ++ b) ++ c  =  a ++ (b ++ c)
```

But it does not define an operator to change a sequence element while keeping the sequence the same!

(This one, some languages *do* get right; e.g. Java's strings are immutable)

Let's look at another example. Strings. So most programming languages have strings. And they would also define a concatenation operator. Let's wite that ++ here. And one of the laws for concatenation is that it's associative. So, a++b with parents to the left, ++c is the same as, as a++ b ++ c, with parents to the right. But again, the theory does not define an operator to change a sequence element, while keeping the sequence the same. This one actually, some languages do get right. For instance, in Java, the type of strings is immutable. It also does not give you an operator to change a character in the string while keeping the string the same.

Consequences for Programming

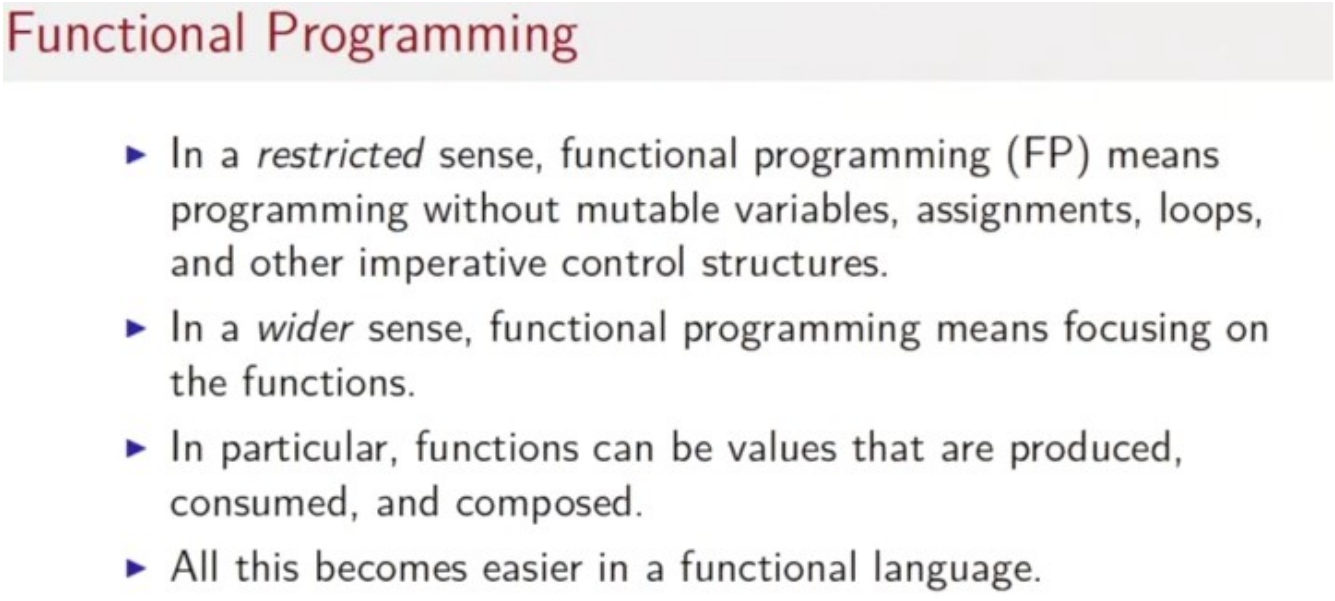
If we want to implement high-level concepts following their mathematical theories, there's no place for mutation.

- The theories do not admit it.
- Mutation can destroy useful laws in the theories.

Therefore, let's

- concentrate on defining theories for operators expressed as functions,
- avoid mutations,
- have powerful ways to abstract and compose functions.

So these were observations about theories in mathematics. What are the consequences for programming? Well, if you want to implement high level concepts following the mathematical theories, you find that there's really no place for mutation. First the theories do not admit they don't have a mutation operator, and second if you add it, then it could, in fact, destroy useful laws in the theory. And that leads to a new style of programming, where we say, well, we want to concentrate in defining these theories. So, operators, which quote in the mp, expressed as functions. We want to avoid mutations. And if we are going to do without something, we want to gain something else. The things we gain is to get powerful race traps, tracked and compose functions. So, a start of function programming means avoid mutations. Get new ways to abstract and compose functions.



上圖才是對 functional programming 最好的解釋.

In fact, there are two ways to look at functional programming, a restricted one and a more general one. In the restricted sense, functional programming just means programming without mutable variables, without assignments to those variables, without loops and the other imperative control structures. So it takes a lot of things away. In the more general sense, functional programming means focusing on the functions in the program. So, in a sense, it gives you new capabilities to work with these functions. In particular, functions can be values that are produced, consumed and composed. All this can be done in any programming language. But it becomes much easier in a functional language.

Functional Programming Languages

- In a *restricted* sense, a functional programming language is one which does not have mutable variables, assignments, or imperative control structures.
- In a *wider* sense, a functional programming language enables the construction of elegant programs that focus on functions.
- In particular, functions in a FP language are first-class citizens. This means
  - they can be defined anywhere, including inside other functions
  - like any other value, they can be passed as parameters to functions and returned as results
  - as for other values, there exists a set of operators to compose functions

So then we can also look at functual languages in a restricted sense or in a more general sense. In the restricted sense then a functual programming language is one which does not have an immutable variables assignments or imperative control structure and in the wider sense the functual programming language is one with that enables the construction of elegant programs that focus on the functions. In particular functions in a functual programming language or first class citizens. What does it mean? It means that essentially you can do with a function that you could do with any other piece of data so. You can define a string anywhere you should be able to define a function anywhere including inside other functions. Like any other value, you should be able to pass a function as a parameter to another function and return it f-, as a result from a function. And as for other values, there will be a set of operators to compose functions into greater functions.

Some functional programming languages

In the restricted sense:

- Pure Lisp, XSLT, XPath, XQuery, FP
- Haskell (without I/O Monad or UnsafePerformIO)

In the wider sense:

- Lisp, Scheme, Racket, Clojure
- SML, Ocaml, F#
- Haskell (full language)
- Scala
- Smalltalk, Ruby (!)

So what are some functional programming languages? In the restricted sense, there are not many. There are some subsets, such as Pure Lisp or Haskell without the IO monad, or unsafe perform IO. There's an experimental language FP. That was the one that Backus proposed. And there are some domain specific languages, mostly in the XML domain. So examples are XSLT, XPath, or XQuery. In the wider sense, we will see the Lisp family of languages, starting with Lisp with prominent dialect Scheme, Record and Clojure. The ML family that has SML or Caml, F#, has the most popular variants, Haskell. The full language, Scala. And also Smalltalk or Ruby, so you might be surprised to see the last ones in the list of, of functional languages because they generally count as object oriented languages. But since both of these languages have a construct of blocks, which are essentially first class function values that we can pass around, I think it's fair to also adopt them in the functional family. Another example of that would be Java Script, which has, is similar capabilities.

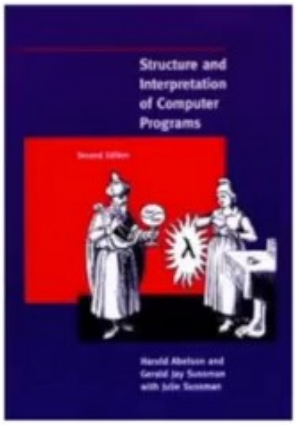
History of FP languages

1959	Lisp
1975-77	ML, FP, Scheme
1978	Smalltalk
1986	Standard ML
1990	Haskell, Erlang
1999	XSLT
2000	OCaml
2003	Scala, XQuery
2005	F#
2007	Clojure

So if you look at the history of functional programming languages, we find that they date back to almost the beginning of programming languages in general. The first functional language was LISP, invented by McCarthy at the end of the 1950s. Then there was a lot of activities in the 1970's and'80s with, ML, FP, Scheme, Small Talk, Standard ML. Reaching into the'90s with Haskell and Erlang. Later languages include OCML, 2000, Scala, 2003. F sharp 2005 and Clojure, 2007.

Recommended Book (1)

Structure and Interpretation of Computer Programs. Harold Abelson and Gerald J. Sussman. 2nd edition. MIT Press 1996.



A classic. Many parts of the course and quizzes are based on it, but we change the language from Scheme to Scala.

The full text can be downloaded here.

So, to find out more about function programming, I recommend this book, Structure and Interpretation of Computer Programs, by Harold Ibelson and Gerald Susman. The second edition appeared at MIT Press in'96. That book is an absolute classic when it comes to function programming. Many parts of our course and also the quizzes are based on it. But we change the language from Scheme to Scala.

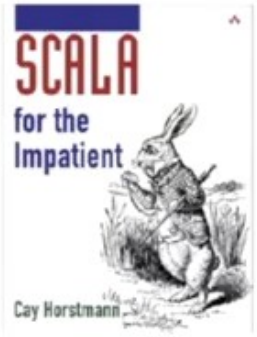
Recommended Book (2)

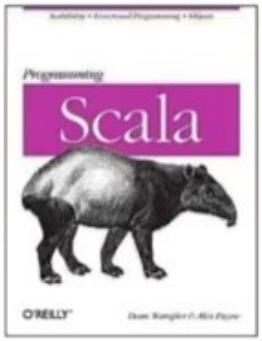
Programming in Scala. Martin Odersky, Lex Spoon, and Bill Venners. 2nd edition. Artima 2010.

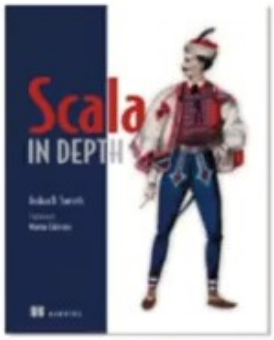
The standard language introduction and reference.

If you want to find out more about Scala then I have to recommend my own book of course. It's called Programming in Scala by myself, Lex Spoon, and Bill Venners. Its also appears in second edition at Artima Press 2010. And that's in a sense the standard language introduction for Scala and standard language reference.

Other Recommended Books







The first part of "Scala for the Impatient" is available for free download.

So then, many other books published about functional programming in general, and scala in particular. I only pick out three to recommend. But there would be many other choices as well. First recommendation is Scala For The Impatient, by Cay Horstmann. That gives a fast paced introduction for people who already know Java well. The second one is the O'Reilly book, Programming in Scala. And the third one is called Scullah in Depth by Josureth. That's the name. Implies that book goes quite a bit further than the other introductory text. You can find links to all of these books on the course site. So if I open up the course site, and go to additional resources. And I see the text, the links for all the books here, and quite a few of them are actually available online, either as the full text or in parts.

Why Functional Programming?

Functional Programming is becoming increasingly popular, because it offers the following benefits.

- simpler reasoning principles
- better modularity
- good for exploiting parallelism for multicore and cloud computing.

To find out more, see the video of my 2011 Oscon Java keynote

Working Hard to Keep it Simple

(16.30 minutes).

The slides for the video are available separately.

Now I've given you a quick introduction to functional programming, and why it matters. I've stressed that there are simpler reasoning principles, and better modularity. But there's actually a third reason why functional programming is becoming increasingly popular. And that's because it's very good for exploiting parallelism on multi call and cloud computing. To find out more about this third aspect, there's actually a video that I gave last year at the OSCON Java conference. The video's called, Working Hard to Keep it Simple. And you get that also from the site. So, you find the slides of the video here. And, and to talk. You find here. I recommend that you go now to that talk to find out a little bit more about Scala what its role is, what its purpose is, and why functional programming is important for parallelism and concurrency. After you've done that, you can come to the next lecture, where we are going to do some Scala programming.