Principles of Programming Languages

Introduction & Chapter 1: Practical Functional Programming

Table of Contents

Introduction: What This Course is About

In One Sentence

This course studies principles underlying the design of programming languages.

Main Objectives

1. Learning principles of programming languages:
    - Elements of programming languages
    - Abstraction means
    - Formal definition
    - Concrete and abstract syntax, operational semantics
    - program correctness - type checking and type inference systems

   Key concepts in describing programming languages will emerge such as substitution, scope, reduction and structural induction. These tools will help us draft first proofs of correctness for programs and program transformations

Main Objectives

2. Describing program execution by studying evaluators: interpreters, program transformers and compilers.

Main Objectives

2. Describing program execution by studying evaluators: interpreters, program transformers and compilers.
3. Comparing programming paradigms: Functional programming, Logic programming and Imperative programming.

Main Objectives

2. Describing program execution by studying evaluators: interpreters, program transformers and compilers.
3. Comparing programming paradigms: Functional programming, Logic programming and Imperative programming.
4. Learning principles of program design: Abstraction, contracts, modular architecture, testing.

Main Objectives

The unifying underlying objective is to understand how programming languages work, why they are designed the way they are, and what good programming practices they encourage.

Main Objectives

Understanding these *principles of programming languages* will help us learn new languages, compare existing languages, choose the right language for a given task, and build our own language when needed.

How Will We Learn?

The course is a mixture of theory and practice. Theoretical topics are supported by implemented software, and course assignments involve programming.

Meta-programming

Describing and developing software tools that manipulate programs - parse them, evaluate them, transform them, translate from one language to another; reason about code to prove that it has some desirable properties (related to correctness or performance) in a predictable manner.

Meta-programming

Specifically, we will learn how to develop interpreters for a functional language, and a program transformer which infers types of an untyped program and rewrites it into a typed form. We will also develop an interpreter for a logic programming language.

Good Programming Practices

Encourage good programming practices and understand their importance through examples, and by applying them to develop meta-programming tools. The parsers, interpreters and transformers of programs we will develop will be practical examples of good programming.

Let’s Begin

Now that we know what we are getting ourselves into, let’s begin!

Chapter 1: Practical Functional Programming

Motivation

In this chapter, we introduce the functional programming paradigm and explain its benefits.

Motivation

We illustrate the recommended practices with the TypeScript programming language - which is an extension of JavaScript which supports an advanced type system. We introduce the notion of type systems, type checking and type inference on practical examples. We illustrate through examples closures, higher order functions, currying, and recursive programming.

Chapter 1: Practical Functional Programming

Programming Paradigms

Definition

A programming paradigm is a way of programming - that recommends “preferred practices” and discourages or makes impossible “risky practice.”

Imperative

Control flow is an explicit sequence of commands.

```
10 INPUT "What is your name: "; U$
20 PRINT "Hello "; U$
30 INPUT "How many stars do you want: "; N
40 S$ = ""
50 FOR I = 1 TO N
60 S$ = S$ + "*"
70 NEXT I
80 PRINT S$
90 INPUT "Do you want more stars? "; A$
100 IF LEN(A$) = 0 THEN GOTO 90
110 A$ = LEFT$(A$, 1)
120 IF A$ = "Y" OR A$ = "y" THEN GOTO 30
130 PRINT "Goodbye "; U$
140 END
```

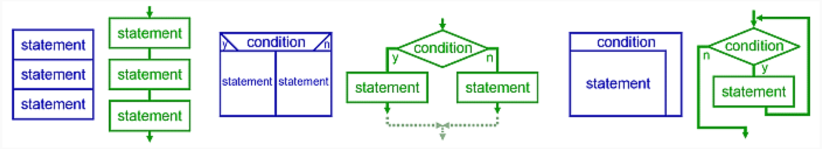
Declarative

Programs state the result you want, not how to get it - leaves flexibility to the language runtime to achieve the goal in optimized ways.

```
SELECT T1.country_name
FROM countries as T1 JOIN continents
AS T2 ON T1.continent = T2.cont_id
JOIN car_makers as T3 ON
T1.country_id = T3.country
WHERE T2.continent = 'Europe'
GROUP BY T1.country_name
HAVING COUNT(*) >= 3
```

Structured

Programs have clean, goto-free, nested control structures - arose in reaction to “goto hell” spaghetti code.



Procedural

Imperative programming organized around hierarchies of nested procedure calls.

```c
int main(int argc, char *argv[]) {
    int x = 5;

    int result1 = doStepOne(x);
    double result2 = doStepTwo(result1);
    printf("The result is %.2f", result2);

    return 0;
}
```

Functional

Computation proceeds by (nested) function calls that avoid any global state mutation and through the definition of function composition.

```
(define sum-even-squares
  (foldl + 0
         (filter even?
                 (map sqr '(1 2 3 4 5)))))
```

Object-oriented

Computation is effected by sending messages to objects; objects encapsulate state and exhibit behavior.

```java
public class Person {
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public void sayHello() {
    System.out.println("Hello, my name is " + name);
  }
}
```

Event-driven

Control flow is determined by asynchronous actions in reaction to events.

```
document.addEventListener("click", e => {
  console.log(`I was clicked at (${e.x}, ${e.y})`);
});
```

Logic

Programmer specifies a set of facts and rules, and an engine infers the answers to questions.

```
male(ben).
female(dor).
teacher(ben, ppl).
teacher(dor, ppl).

?- teacher(X, ppl), female(X).
X = dor.
```

Relation Between Paradigms and Languages

Programming Languages, by the way they are designed, make some programming paradigms easy to follow. When this is the case, we say that the language belongs to the paradigm.

For example, Scheme is a functional programming language.

Relation Between Paradigms and Languages

In other words, programming paradigms are a way to classify programming languages (a paradigm is a **family of programming languages** that have similar properties)

Relation Between Paradigms and Languages

A given programming language can support more than one programming paradigm. For example, C++ and Java are multi-paradigm languages, which support the Object-oriented paradigm, the Procedural paradigm, and in recent versions some aspect of the Functional paradigm.

Dimensions of Variability Across Programming Paradigms

Paradigms are distinguished by programming practices they encourage and forbid (or make difficult). Historically, new paradigms emerge in reaction to problems faced over time by standard practitioners. The motivating forces and programming practices that are concerned by programming paradigms include:

Dimensions of Variability Across Programming Paradigms

- **Control Flow**: how execution flows within the program (sequence and branches, in concurrent threads, in reactive manner, declarative).
- **Code Organization**: how code is organized into a hierarchy of units (expressions, functions, modules, packages) and how these units are organized.
- **Performance**: how code can be run fast, use less resources (RAM, disk, network), behave better (responsive, scalable) at runtime.

Dimensions of Variability Across Programming Paradigms

- **Coupling and Reuse**: how easily code can be reused in different contexts.
- **Testing**: how easy it is to test and verify that code is correct.
- **Syntax**: how natural, brief, readable is the expression of code given the syntax of the language. Can the syntax of language be extended by the programmer.
- **Domain**: to which application domain is the paradigm best applicable (server-side processing, database, GUI front-end, control system).

Chapter 1: Practical Functional Programming

Functional Programming

Introduction

We will start our investigation of programming language principles by studying a specific paradigm - called Functional Programming (FP). We will first illustrate practical applications of this paradigm using JavaScript.

Introduction

Functional Programming (FP) is a paradigm of programming that is most similar to evaluation of expressions in mathematics.

A program is viewed as an expression, which is evaluated by successive applications of functions to their arguments, and substitution of the result for the functional expression.

Its origin is in the *lambda calculus* invented by Church in the 1930s.

Expressions and Values

No state!

Expressions and Values

No state!

Computation is not a sequence of statements that modify state (variables, DB, console), but a sequence of *expressions* that result from successive *evaluation* of sub-expressions.

Expressions and Values

For example:

```
(2 * 3) + (4 * 5);
```

Expressions and Values

For example:

```
6 + (4 * 5);
```

Expressions and Values

For example:

```
6 + 20;
```

Expressions and Values

For example:

```
26;
```

Expressions and Values

Similarly, when we evaluate function calls.
Given the following functions:

```
function f(x) { return x * x; }
function g(y) { return y + y; }
```

Expressions and Values

```
f(2) + g(3);
```

Expressions and Values

```
(2 * 2) + g(3);
```

Expressions and Values

```
4 + g(3);
```

Expressions and Values

```
4 + (3 + 3);
```

Expressions and Values

```
4 + 6;
```

Expressions and Values

```
10;
```

No Side Effects

Computation in functional programming has no side-effects. In the examples above, we did not ask the program to "print" a result - instead we evaluated an expression (which is the program) and the interpreter returned the value of the expression (which conveniently got printed at the end of the evaluation).

No Side Effects

The only result of a functional computation is the computed value, and there are no additional changes that can take place during computation. Functional programming does not support **variable assignment** or **state mutation**.

No Side Effects

When external side-effects are required (sending data to an output device, to disk, to the network), FP tends to delay the side-effect and push it outside of the computation.

No Side Effects

Imperative Programming uses statements which modify the state of the system, while Functional Programming evaluates expressions to return values.

Higher Order Functions

FP requires that functions be first-class, which means that they are treated like any other values:

- Functions can be passed as arguments to other functions
- Functions can be returned as a result of a function
- Functions can be anonymous

Higher Order Functions

For example:

```
import { map } from "ramda";

function square(x) { return x * x; }

let a = [1, 2, 3];
let b = map(square, a);
console.log(b);
// ==> [1, 4, 9]
```

Higher Order Functions

The function `map` receives a function as a parameter and an array, and applies the function to each element in the array, returning a new array.

Sometimes, the functions passed in as arguments are very simple, and thus don't need a name, so we can do:

```
map(x => x * x * x, a);
```

Higher Order Functions

An unnamed function is called a *lambda* expression.

Chapter 1: Practical Functional Programming

Advantages of FP

Verification

The way the program is executed is closely related to the way we prove and justify the correctness of a program mathematically. Proof by mathematical induction is closely related to the programming technique of recursion. Because functions have no side-effects, they behave like mathematical functions - each time they are called with the same parameters, they return the same values (this is called *determinism*).

Parallelism

Since expressions have no side-effects, it is natural to use parallel evaluation: the values of independent sub-expressions may be determined simultaneously, without fear of interference or conflict, and the final result is not affected by evaluation order. This enables programs to exploit available parallelism without fear of disrupting their correctness.

Abstraction

FP stresses data-centric computation, with operations that act on compound data structures as a whole, rather than via item-by-item processing. More generally, FP emphasizes the isolation of abstract types that clearly separate implementation from interface. Types are used to express and enforce abstraction boundaries, greatly enhancing maintainability of programs, and facilitating team development.

Chapter 1: Practical Functional Programming

Using JavaScript to Illustate FP

From Imperative to Procedural

To illustrate the differences among programming paradigms, we will demonstrate multiple iterations over a program that fulfills a simple requirement: We are asked to write a program to display a number value squared.

From Imperative to Procedural

We will then study how the program must be adapted when we introduce slight modifications of the requirement. All the examples are given in TypeScript.

From Imperative to Procedural

We first write a single command that performs as requested:

```
console.log(0 * 0);
```

From Imperative to Procedural

The basic programming tool we used is a **command**, also called a **statement**.

From Imperative to Procedural

Oh, no! The requirement has changed, and we are now requested to print the squares of the numbers 0 to 4.

```
console.log(0 * 0);
console.log(1 * 1);
console.log(2 * 2);
console.log(3 * 3);
console.log(4 * 4);
```

Structured Programming

This is no good!

- Code repetition
- Cannot be easily adapted to new values of the **parameters**
- The nature of the task is not reflected in the structure of the code

What can we do?

Structured Programming

- Use **variables** to capture the parameters, so that the same program can be applied to different values.
- Use an **array** to separate the data from the task.
- Use a **loop control flow structure** to express the fact that the same task is repeated multiple times on different values.

Structured Programming

```
let numbers = [0, 1, 2, 3, 4];
for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i] * numbers[i]);
}
```

Structured Programming

The key **programming construct** that we have introduced in the programming language is the **for-loop**. We also introduced variables (`numbers` and `i`).

What if we want to print the squares of the numbers 8 to 12? We have to copy the program and change the `numbers` array.

Procedural Programming

Well, we still need to copy our program for every new set of parameters.

Also, the coupling between the code and the parameters is *accidental*, meaning that we don’t know whether `numbers` and `i` apply only to our code snippet, or to other snippets of code as well.

Procedural Programming

Procedural programming improves on these weaknesses by introducing:

- **Procedures**: commands with a well defined interface of input parameters / output parameters and expected behavior.
- **Local variables**: variables which are defined only within the scope of the procedure.

Procedural Programming

```
function printSquares(numbers) {
  for (let i = 0; i < numbers.length; i++) {
    console.log(numbers[i] * numbers[i]);
  }
}
```

Procedural Programming

The programming constructs we introduced are **functions** and **let** which defines local variables for a block of statements.

Procedural Programming

Procedures (functions) have an interface:

- Name
- Input parameters
- Return value

The name of a procedure forms an **abstraction**. It hides away implementation details from the client of the procedure.

Procedural Programming

Consider the following change in requirements: we now want to print the cube of the numbers rather than their square.

Procedural Programming

```
function cube(x) {
  return Math.pow(x, 3);
}

function printCubes(numbers) {
  for (let i = 0; i < numbers.length; i++) {
    console.log(cube(numbers[i]));
  }
}
```

Procedural Programming

We see a first case of procedural abstraction in this example: the procedure `printCubes` iterates over an array, and applies the function `cube` on each element. The client of the `printCubes` procedure does not directly invoke the `cube` function - it is encapsulated inside the `printCubes` procedure.

Testing Requirements

We want to provide the capability to verify that a procedure is correct according to its specification.

Testing Requirements

We want to provide the capability to verify that a procedure is correct according to its specification.

Problem: The function `printCubes` above is difficult to test because it only prints to the console, not returning any output.

Testing Requirements

We want to provide the capability to verify that a procedure is correct according to its specification.

Problem: The function `printCubes` above is difficult to test because it only prints to the console, not returning any output.

Solution: Refactor!

Testing Requirements

```
function cubes(numbers) {
  for (let i = 0; i < numbers.length; i++) {
    numbers[i] = cube(numbers[i]);
  }
}

function printArray(a) {
  for (let i = 0; i < a.length; i++) {
    console.log(a[i]);
  }
}

function printCubes2(numbers) {
  cubes(numbers);
  printArray(numbers);
}
```

Testing Requirements

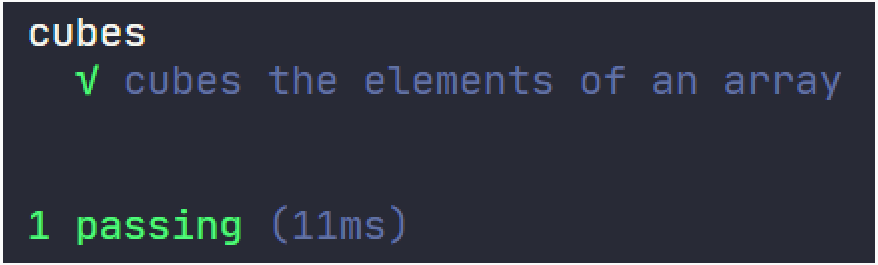
Now, let us write a unit test using the Mocha and Chai libraries:

Testing Requirements

```
import { expect } from "chai";
import { cubes } from "./lecture1";

describe("cubes", () => {
  it("cubes the elements of an array", () => {
    let numbers = [1, 2, 3];
    cubes(numbers);
    expect(numbers).to.deep.equal([1, 8, 27]);
  });
});
```

Testing Requirements



```
cubes
  √ cubes the elements of an array


1 passing (11ms)
```

Problems with the Procedural Paradigm

At this point, we have a *nice* version of our program:

- It is organized in layers of abstraction
- The procedures that operate over arrays use a structured for-loop
- It is testable

Problems with the Procedural Paradigm

These *good features* were encouraged by facilities of the programming language we use:

- It is easy to define arrays, give them names, initialize them with values, pass them as parameters, access their elements.
- It is easy to define functions.

Problems with the Procedural Paradigm

- Functions can invoke other functions when knowing their name and the parameters they expect.
- It is easy to test functions using facilities like Mocha and Chai.

Problems with the Procedural Paradigm

In other words, the language *encouraged us* to organize our program in a good manner.

Problems with the Procedural Paradigm

The procedural paradigm does have its drawbacks:

- Procedural programming encourages shared state with mutation which makes concurrency difficult.
- Procedural programming commits early to a step by step way to implement operations which prevents performance optimizations.

Problems with the Procedural Paradigm

- Procedural programming makes it difficult to create functional abstractions that are highly reusable.
- Procedural programming makes it difficult to reason about code because of shared state and mutation.

Concurrency

Assume we run the procedure `squares` in two concurrent threads on the same array `numbers`.

```
function squares(numbers) {
  for (let i = 0; i < numbers.length; i++) {
    numbers[i] = numbers[i] * numbers[i];
  }
}

let n89 = [8, 9];

// In Thread 1:
squares(n89);
// In Thread 2:
squares(n89);
```

Concurrency

If the 2 threads are interleaved in an unfortunate sequence of events - the following scenario can occur:

Thread 1	Thread 2
numbers[0] = 64	
numbers[1] = 81	numbers[0] = 64
	numbers[1] = 6561

Concurrency

So how can we handle this problem?

- **Option 1**: enforce mutual exclusion using locks
- **Option 2**: immutable data structures

Declarative vs. Procedural

Consider the loop control structure as we defined it. It involves a counter variable `i` that is defined for the scope of the loop, initialized to 0, and mutated from 0 to the length of the array over which we iterate (with the `i++` operator). This is one way to iterate over the elements of an array - which is described step by step in a procedural way, as a precise recipe.

Declarative vs. Procedural

In contrast, in FP, one would prefer to use a more abstract operation, called `map`, which consists of applying a function over all the elements of a container, and returning a new container that contains the results.

```
function cubes(numbers) {
  return numbers.map(cube);
}
```

Declarative vs. Procedural

This version of the function does not change its parameter - instead, it returns a new array which contains the result. The result has the same length as the parameter. Note also that the operation `map` does not require a counter like `i` to iterate over the array. There is no mutation.

Declarative vs. Procedural

An alternative way to express the same FP tool is to use the `map` function instead of the array `map` method. This is illustrated in this example, using the Ramda package which provides a large set of FP facilities for JavaScript:

```
import { map } from "ramda";

function cubes(numbers) {
  return map(cube, numbers);
}
```

Functional Abstractions

Let's remember our procedural solution to our cubing requirement:

```
function cubes(numbers) {
  for (let i = 0; i < numbers.length; i++) {
    numbers[i] = cube(numbers[i]);
  }
}

function printArray(a) {
  for (let i = 0; i < a.length; i++) {
    console.log(a[i]);
  }
}

function printCubes2(numbers) {
  cubes(numbers);
  printArray(numbers);
}
```

Functional Abstractions

Surprise, surprise, new requirements! Compute the exponential value of all elements in a list of numbers (instead of their cube).

If we could only pass a function as a parameter...

Functional Abstractions

`map` to the rescue!

```
function exponents(numbers) {
  return numbers.map(Math.exp);
}
```

Functional Abstractions

Side-note on syntax: if we want to use an anonymous function, we can use “fat arrow notation”:

```
function squares(numbers) {
  return numbers.map(x => x * x);
}
```

or the function syntax (but then we need to use the `return` keyword):

```
function squares(numbers) {
  return numbers.map(function (x) { return x * x; });
}
```

Functional Abstractions

Consider the following requirement: we want to apply a function to a list of numbers, and then keep only the values that are even.

Functional Abstractions

Our solution might look like this:

```
function isEven(x) {
  return x % 2 === 0;
}

function mapAndKeepEven(f, a) {
  let fa = a.map(f);
  let res = [];
  for (let i = 0; i < fa.length; i++) {
    if (isEven(fa[i])) {
      res.push(fa[i]);
    }
  }
  return res;
}
```

Functional Abstractions

It can't be! Mutation? Looping? Not in FP!

Introducing `filter`: given a predicate and a list, return a new list with all elements that satisfy the predicate. Example:

```
[1, 2, 3, 4, 5, 6].filter(isEven)
// ==> [2, 4, 6]
```

Functional Abstractions

So back to mapAndKeepEven, we can rewrite it as

```
function mapAndKeepEven(f, a) {
  return a.map(f).filter(isEven);
}
```

Functional Abstractions

But why stop here? We can abstract `isEven` out, and take the predicate as a parameter, giving us `mapAndFilter`:

```
function mapAndFilter(f, pred, a) {
  return a.map(f).filter(pred);
}
```

Functional Abstractions

Using Ramda, we will *compose* the functions, instead of method chain them:

```
import { map, filter } from "ramda";

function mapAndFilter(f, pred, a) {
  return filter(pred, map(f, a));
}
```

This leads us to another important pattern in FP: composition.

Functional Abstractions

Ramda offers a function called `compose`, which takes functions $f_1, \ldots, f_n$ and returns a new function, which is their composition $f_1 \circ f_2 \circ \ldots \circ f_n$. For example:

```
import { map, filter, compose } from "ramda";

compose(filter(isEven), map(cube))([0, 1, 2, 3, 4]);
// ==> [0, 8, 64]
```

Functional Abstractions

So what have we seen so far?

- Functions can receive functions as parameters - including anonymous functions (lambda). For example, `map`, `filter` and `compose` receive functions as parameters.

Functional Abstractions

So what have we seen so far?

- Functions can return functions as a computed value. For example, `compose` returns a new function as a computed value.

Functional Abstractions

So what have we seen so far?

- Ramda functions that receive two arguments, such as `map` and `filter` behave interestingly when they are passed a single argument - this is called currying. This behavior makes it much easier to compose functions.

Functional Abstractions

All of these features together encourage a style of programming in which new functions are built incrementally from smaller functions. This method is the basis of what we call functional abstractions - such as the family of operators map and filter (and more we will get to discover) or the operator compose.

Reasoning About Code

Suppose we implemented `evenCubes` as

```
const evenCubes = compose(filter(isEven), map(cube));
```

And along comes a friendly consultant that suggests we can optimize this function by first filtering, then cubing:

```
const evenCubes = compose(map(cube), filter(isEven));
```

Are these two functions equivalent?

Reasoning About Code

Definition - Function Equivalence

Two functions f, g are equivalent (which we will write $f \equiv g$) if $Domain_f = Domain_g$, $Range_f = Range_g$ and $\forall x \in Domain_f$, $f(x) = g(x)$.

Reasoning About Code

This definition holds for the mathematical sense, but computation can throw an exception, or not terminate. Let’s modify our definition to support that:

Reasoning About Code

Definition - Function Equivalence

Two functions f, g are equivalent (which we will write $f \equiv g$) if $Domain_f = Domain_g$, $Range_f = Range_g$ and $\forall x \in Domain_f$, either:

- $f(x) = g(x)$
- If f throws an exception, so does g
- If f does not terminate, so does g

Reasoning About Code

Definition - Referential Transparency

The value of an expression depends only on its sub-expressions, and if you substitute a sub-expression in an expression by another expression that is equivalent, then the resulting expression is equivalent to the original.

Reasoning About Code

Now, let's check our consultant's suggestion.
Let $f = \mathit{compose}(\mathit{filter}(\mathit{isEven}), \mathit{map}(\mathit{cube}))$ and
$g = \mathit{compose}(\mathit{map}(\mathit{cube}), \mathit{filter}(\mathit{isEven}))$.