Many thanks to Tamar Pinhas, Ami Hauptman, Eran Tomer, Barak Bar-Orion, Azzam Marae, Yaron Gonen, Ehud Barnea, Rotem Mairon, Igal Khitron, Rani Etinger, Ran Anner, Tal Achimeir, Michael Elhadad, Michael Frank for their great help in preparing these notes and the associated code.
# Contents

## Introduction

### 1 Practical Functional Programming

#### 1.1 Programming Paradigms

1.1.1 Relation between Programming Paradigms and Programming Languages 
1.1.2 Dimensions of Variability across Programming Paradigms 
1.1.3 Key Programming Paradigms

#### 1.2 Functional Programming

1.2.1 Expressions and Values
1.2.2 No Side Effects
1.2.3 Higher Order Functions
1.2.4 Advantages of FP

#### 1.3 Using Javascript to Illustrate FP

1.3.1 From Imperative to Procedural
1.3.2 Structured Programming
1.3.3 Procedural Programming
1.3.4 Testing Requirements
1.3.5 Problems with the Procedural Paradigm
1.3.6 Concurrency
1.3.7 Declarative vs. Procedural
1.3.8 Functional Abstractions
1.3.9 Reasoning about Code

#### 1.4 Summary

1.4.1 Programming Paradigms
1.4.2 Functional Programming
1.4.3 Semantics of Programming Languages
Introduction

What the Course is About

This course studies principles underlying the design of programming languages. It has four main objectives:

• Learning principles of programming languages: elements of programming languages; abstraction means; formal definition; concrete syntax, 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.

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

• Comparing programming paradigms: Functional programming, Logic programming and Imperative programming.

• Learning principles of program design: Abstraction, contracts, modular architecture, testing.

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. 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.

The course is a mixture of theory and practice. Theoretical topics are supported by implemented software, and course assignments involve programming. It combines two main threads:

• **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. 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 metaprogramming tools. The parsers, interpreters and transformers of programs we will develop will be practical examples of good programming.

A tool we will use to understand the structure of programming languages will be to compare different languages. Specifically, we will compare Javascript (and some of its variants such as Typescript), Scheme and Prolog. We will also refer to Java but without using it.

**Course Outline**

1. Chapter 1: Practical functional programming in Typescript:
   (a) programming paradigms
   (b) elements of functional programming: higher-order functions, anonymous functions, closures, currying, lazy evaluation, immutable data, recursion.
   (c) advantages of functional programming
   (d) types, type checking, type inference
   (e) compound data structures, functional processing of JSON
   (f) typing functions, closures
   (g) currying, partial evaluation
2. Chapter 2: Operational Semantics and Interpreters
3. Chapter 3: Program Transformations: A Type Inference System
4. Chapter 4: Abstraction over Control: Asynchronous programming, Generators, Lazy Data Structures, Continuations
5. Chapter 5: Logic Programming; Logic Programming Interpreter
Chapter 1

Practical Functional Programming

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

1.1 Programming Paradigms

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

References:

- [http://cs.lmu.edu/ ray/notes/paradigms/](http://cs.lmu.edu/ ray/notes/paradigms/)
- [Wikipedia on Programming Paradigms](https://en.wikipedia.org/wiki/Programming_paradigm)

There exist multiple different programming paradigms:

- **Imperative** — Control flow is an explicit sequence of commands - mainly defined in contrast to “declarative”.

- **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 (for example, SQL and spreadsheets are declarative programming environments).

- **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.
Chapter 1. Practical Functional Programming

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

- **Object-Oriented** — Computation is effected by sending messages to objects; objects encapsulate state and exhibit behavior.

- **Event-Driven** — Control flow is determined by asynchronous actions in reaction to events (from humans, sensors or other computations).

- **Logic (Rule-based)** — Programmer specifies a set of facts and rules, and an engine infers the answers to questions.

### 1.1.1 Relation between Programming Paradigms and Programming 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*). In other words, programming paradigms are a way to classify programming languages (a paradigm is a family of programming languages that have similar properties).

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.

### 1.1.2 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:

- **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.

- **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).

### 1.1.3 Key Programming Paradigms

Each paradigm puts emphasis on specific programming techniques which came to help on certain aspects of code quality. For example:

• **Structured programming**: encourages the use of structured control flow tools such as if/elseif/else, while - and to avoid uncontrolled goto.

• **Procedural programming**: encourages the use of small units of codes, called procedures, which encapsulate well-defined commands. Procedures interact through well defined interfaces published by each procedure (the *contract* of the procedure, including the signature of which arguments it expects, and which return value it returns), and local variables are used inside each procedure without affecting the state of the program outside the scope of the procedures.

Key programming constructs associated to this paradigm are *procedures, local variables*. The key driving force behind procedural and modular programming was the desire to define independent *modules* as proposed in the classic article: Parnas, D.L. (1972). *On the Criteria To Be Used in Decomposing Systems into Modules* Communications of the ACM. 15 (12): 1053–58. The concepts of *high cohesion* within module and *low coupling* across modules are central to this approach. The main languages associated to this paradigm are C and C++.

• **Object-oriented programming**: models computation as the exchange of messages between independent objects. Data is encapsulated in units called objects, and can only be accessed through method calls. This paradigm evolved as a development of procedural programming, to encourage data abstraction, information hiding, polymorphism, and encapsulation. Key languages supporting OOP include Java and C++.

• **Functional programming**: models computation as the evaluation of mathematical functions, with no side-effect and no mutation of state. We will develop this definition in this course. Key languages supporting Functional programming include Scheme, Haskell and Javascript.
1.2 Functional Programming

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.

Refer to these pages:
- [wiki.haskell.org/Functional_programming](http://wiki.haskell.org/Functional_programming)
- [CMU Course 15150 - Intro to Functional Programming notes](http://cmu.edu/)

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

1.2.1 Expressions and Values

The most characteristic feature of functional programming is the lack of state during a computation. That is, a computation is not a sequence of states, created by commands (also called statements) that modify the states. Rather, a computation is a sequence of expressions, that result from the successive evaluation of sub-expressions.

For example, the following expression is computed as follows:

```
(2 * 3) + (4 * 5);
// Computation proceeds in steps:
// step 1 - the sub-expression (2 * 3) is evaluated and becomes 6
// step 2 - 6 replaces (2 * 3) in the expression: 6 + (4 * 5)
// step 3 - the sub-expression (4 * 5) is evaluated and becomes 20
// step 4 - 20 replaces (4 * 5) in the expression: 6 + 20
// step 5 - 6 + 20 is evaluated and becomes 26
// end: there are no more sub-expression to evaluate
==> 26 // the value is returned
```

The expression \((2 * 3) + (4 * 5)\) is transformed into a value by incrementally substituting sub-expressions into values.

Similarly, when we evaluate function calls:

```javascript
function f(x) {
    return x * x;
}

function g(y) {
```
return y + y;
}

f(2) + g(3);
// Computation proceeds in steps:
// step 1 - sub-expression f(2) is substituted by 2 * 2
// (2 * 2) + g(3)
// step 2 - sub-expression (2 * 2) is evaluated and becomes 4
// 4 + g(3)
// step 3 - sub-expression g(3) is substituted by 3 + 3
// 4 + (3 + 3)
// step 4 - sub-expression (3 + 3) is evaluated and becomes 6
// 4 + 6
// step 5 - 4 + 6 is evaluated and becomes 10
// end: there are no more sub-expression to evaluate

===> 10

1.2.2 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).

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. 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.

The key verbs to contrast between Imperative programming (IP) and FP are:

- In IP, commands (or statements) are executed, and through their effect, state is modified (mutated).
- In FP, expressions are evaluated, and become values.

1.2.3 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 (no names)

For example, in this program, the function square is applied to each element in the array a, returning a new array which is bound to the variable b.

```javascript
import { map } from 'ramda'

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

let a = [1, 2, 3]
let b = map(square, a)
console.log(b)
```

The expression \( x \rightarrow x \times x \times x \) is called a lambda expression - it is an unnamed function which receives \( x \) as an argument and returns \( x \times x \times x \) as its value.

1.2.4 Advantages of FP

The advantages of FP include:

• 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.

### 1.3 Using Javascript to Illustrate FP

We will use Javascript examples to illustrate these points. Javascript is not a purely functional language. It supports multiple paradigms - procedural programming, OOP and has support for FP as well. The primitives of the language sometimes contradict good practices recommended by FP (they allow easy mutation of variables or arrays - but we will learn to avoid these “bad corners” of the language). Javascript has a long and tormented history - it started as a weak language, and became standardized into a more mature language since about 2016. The EcmaScript standardization body has led this effort. We will exclusively use in the course the most recent version of Javascript - called EcmaScript 2016 or ES 7.

Javascript is used extensively to support front-end user interface programming in Web and mobile applications within browsers. For such client GUI applications, recent frameworks such as Angular2 and React rely extensively on FP to make code easier to write, reuse and test.

Javascript is increasingly used on the server-side as well, within the Node.js environment. Node is a Javascript interpreter which can be invoked standalone. It has a large set of libraries that put emphasis on asynchronous programming - an aspect we will review later in the course.

We will use a variant of Javascript called Typescript - which is Javascript (ES 7) with optional type annotations and type checking.

#### 1.3.1 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. 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.

We first write a single command that performs as requested:
```javascript
console.log(0 * 0);
```

```javascript
=>
0
```

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

The requirement is slightly changed: we are now asked to print square values for a range of integer numbers from 0 to 4. The "level 0" program that fulfills this requirement is obtained by creating a sequence of commands:

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

```javascript
=>
0
1
4
9
16
```

### 1.3.2 Structured Programming

Structured Programming starts with a critique of this "level 0" program:

- We observe code repetition
- The program cannot be easily adapted to different values of the parameters: if the requirements are slightly changed - instead of range of numbers from 0 to 4 to range of numbers from 8 to 12 then the whole program must be rewritten.
- The nature of the task is not reflected in the structure of the code: we asked to perform the same command multiple times on different values, this is not reflected in the way the program is written. The program does not reflect the intention of the programmer in a transparent manner.

To address these weaknesses, we introduce programming constructs that help us improve the program:

- Use variables to capture the parameters - so that the same program can be applied to different values.
• Use an **array data structure** to separate the data on which we want to execute the task and the task itself.

• Use a **loop control flow structure** to express the fact that the same task is repeated multiple times on different values.

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

```
0
1
4
9
16
```

The key **programming construct** that we have introduced in the programming language to support this scenario is the **for-loop**. This construct is closely associated to the array data structure. We also introduced variables - one for the array and one for the loop index which iterates over all the elements in the array (the `i` variable).

Now, if we want to apply the same program on different values (range 8 to 12 instead of 0 to 4), we need to copy the program, change the variable from `[0...4]` to `[8..12]`:

```javascript
let numbers = [8, 9, 10, 11, 12];
for(let i = 0; i < numbers.length; i++) {
  console.log(numbers[i] * numbers[i]);
}
```

```
Output:
64
81
100
121
144
```

### 1.3.3 Procedural Programming

Procedural programming starts with a critique of structured programming:

• While we have obtained a more concise program to describe the task we want to achieve, and we have separated the parameters on which we apply the task from the task itself, we still need to copy the code for each run on different parameters.
• The coupling between the code and the parameter is accidental - the usage of the variables (numbers and i) does not indicate that the code applies to these variables and no other code can manipulate these variables.

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.

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

printSquares([0, 1, 2, 3, 4]);
printSquares([8, 9, 10, 11, 12]);
```

```text
=>
0
1
4
9
16
64
81
100
121
144
```

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

**Procedures Interface**

Procedures (also called functions) have a well defined interface:

• Name

• Input parameters

• Return value
The fact that the procedure is given a name is important: it is a form of abstraction. The name replaces a complex sequence of commands - and programmers can re-use the new procedure just by knowing its name. Procedures have parameters (the numbers parameter in the example above) and local variables (the variable i is introduced by the let construct as a local variable).

Consider now a slight change in the requirements: instead of printing the square of the numbers, we want to print the cube of the numbers.

We address this change by introducing a new function, which represents what we want to do on each element in the range of numbers, and we adapt the function `printSquares` to invoke this function instead.

```javascript
// We use a library function Math.pow
function cube(number) {
    return Math.pow(number, 3);
}

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

printCubes([0, 1, 2, 3, 4]);
printCubes([8, 9, 10, 11, 12]);
```

Abstraction Barriers

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.

If such discipline is applied systematically, we can enforce abstraction barriers between collections of procedures: higher-level procedures only call lower-level procedures. To support such discipline, some languages introduce concepts such as modules or packages.


1.3.4 Testing Requirements

We want to provide the capability to verify that a procedure is correct according to its specification. The function printCubes above is difficult to test - because it receives a parameter, but the only result of its execution is that a series of numbers are printed on the console. It is impossible to write an automated test that will verify that the procedure behaves as expected given a specific input.

To address this limitation, we will refactor the program in 2 stages - data transformation and data output.

- The data transformation stage receives an input parameter and returns a transformed value.
- The data output stage only prints the transformed values.

This allows us to test the data transformation procedure by feeding it some test data, and then checking that the output fulfills expectations.

```javascript
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);
}
```
To run an automatic test, we can now write a unit test in the following style:

```javascript
// The assert library is used to perform tests
const assert = require('assert');

function testCubes() {
  // Empty list
  let numbers = [];
  cubes(numbers);
  assert.ok(numbers.length == 0);

  // Invariant cubes are not modified
  numbers = [0,1];
  cubes(numbers);
  assert.deepEqual([0,1], numbers, "invariant cubes");

  // Regular
  numbers = [2];
  cubes(numbers);
  assert.deepEqual(numbers, [8], "cube(2) === 8");

  return "all ok";
}

testCubes();

==>
'all ok'
```
1.3.5 Problems with the Procedural Paradigm

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

- It is organized in layers of abstraction \((\text{printCubes} > [\text{printArray, cubes}] > \text{cube})\)
- The procedures that operate over arrays (cubes and printArray) use a structured loop (for) to iterate over the items in a way that reflects the task.
- It can be tested.

These good features were encouraged by the 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.
- Functions can invoke other functions when knowing their name and the parameters they expect.
- It is easy to test functions using facilities like assert.

In other words, the language encouraged us to organize our program in a good manner. When we scale to larger and more interesting programs, we face new types of problems:

- The flow of variables across functions must be explained: are parameters passed by value (in parameters), or by reference (inout parameters) and how returned values are shared between the caller and the called function.
- The responsibilities around data structures must be clarified: which functions can only read data, and which functions can read and modify data; when is data allocated and freed.

We will return to these issues when we discuss variables and scopes. These aspects have motivated the development of the Object Oriented Paradigm.

For now, we will focus on the following issues of the procedural paradigm:

- 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.
- 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.

We turn to how these issues are addressed by FP.

### 1.3.6 Concurrency

Assume we run the procedure cubes in two concurrent threads (using an Executor as we have learned in SPL) on the same array numbers. (This is not possible in the TypeScript interpreter we are using - we will return to this aspect in the chapter discussing control flow - but variants of multi-threaded execution that can also cause safety problem are possible in TypeScript.)

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

let n89 = [8, 9];
// In Thread 1:
cubes(n89);
// In Thread 2:
cubes(n89);
```

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

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>numbers[1] = 64</td>
<td></td>
</tr>
<tr>
<td>numbers[2] = 81</td>
<td></td>
</tr>
<tr>
<td>numbers[1] = 64</td>
<td></td>
</tr>
<tr>
<td>numbers[2] = 6561</td>
<td></td>
</tr>
</tbody>
</table>

At the end of the execution, numbers may contain [64, 81], [4096, 6561], [4096, 81] or [64, 6561] or some even more unexpected values (4096 = 64 * 64, 6561 = 81 * 81).

The problem is caused by the uncontrolled access to the shared variable numbers by two independent threads. As learned in SPL, the solution to this lack of concurrency safety can be to either use immutable data structures or to enforce mutual exclusion using locks.

In the procedural paradigm, it is difficult to support immutability because shared variables and mutation are the natural way of passing information across procedures and modules. So that in the procedural programming paradigm, the default solution to enforce concurrency safety is to use locking.
But in turn, locking leads to problems of liveness - creating the possibility of deadlocks, starvation and other unpleasant phenomena.

In contrast, FP encourages immutable computation.

Note that even without threads, mutation can still lead to unsafe computation. We will see examples of this when we learn about asynchronous computation and the type of interleaved computation we can generate with generators.

### 1.3.7 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.

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

```javascript
function cubes3(numbers) {
    return numbers.map(cube);
}
cubes3([0,1,2]);
```

```plaintext
=> [ 0, 1, 8 ]
```

The example above uses the \( \text{map} \) method of the array object. It receives as a parameter a function (\( \text{cube} \)). This is an example of functions as first class citizens in FP languages (also called \textbf{higher order functions}).

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 \( \text{map} \) does not require a counter like \( i \) to iterate over the array. There is no mutation.

An alternative way to express the same FP tool is to use the \( \text{map} \) function instead of the array \( \text{map} \) method. This is illustrated in this example, using the \textit{ramda} package which provides a large set of FP facilities for Javascript:

```javascript
import { map } from 'ramda';
map(cube, [0,1,2]);
```

```plaintext
=> [ 0, 1, 8 ]
```
1.3.8 Functional Abstractions

Consider the procedural program we have investigated:

```javascript
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);
}
```

We are given a slightly different requirement: compute the exponential value of all the elements in a list of numbers (instead of computing their cube).

To satisfy this in an easy way, we must find a way to pass the function we want to evaluate on each element as a parameter. Passing functions as parameters is possible in most paradigms, but it is made really easy in Functional Programming.

Map

The pattern we require consists of evaluating a function given as a parameter to every element of an array given as a second parameter. The result is a new array - in keeping with the principle of no side effect / no mutation of FP: the original array is left unmodified.

This pattern is immensely useful and is called map. As we have seen above, we can invoke map in two forms in Typescript: As a method of an array object. As a function with 2 parameters.

```javascript
[0,1,2,3].map(Math.exp)
```

```javascript
==>
[ 1, 2.718281828459045, 7.38905609893065, 20.0855369231876685]
```

// Similarly - using the Ramda map function:

```javascript
import { map } from 'ramda';
```
map(Math.exp, [0,1,2,3]);
==>
[ 1, 2.718281828459045, 7.38905609893065, 20.085536923187668 ]

And similarly, we can use our own cube or square function:

Compare this with the example above in procedural programming: we defined two functions printSquares and printCubes where the only difference was which function is applied on the elements of the array. Here, we have abstracted the function as a parameter.

Anonymous (Lambda) Functions

If we want to use a function that will only be used in the context of the map operation, we can use an anonymous function (also called a lambda function). The syntax in modern Javascript for lambda functions is:

(<parameters> ...) => <expression>

This is often called the fat-arrow notation. If there is only one parameter to a fat-arrow function, the parentheses are optional.

map((x)=>x*x, [0,1,2,3])
==> [ 0, 1, 4, 9 ]

The function syntax can also be used for anonymous functions - it requires the usage of the return statement:

map(function (x) {return x * x;}, [0,1,2,3])
==> [ 0, 1, 4, 9 ]

Filter

Consider yet another slight change in the requirements: we want to apply a function to a list of numbers, and then keep only the values that are even.

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

function mapAndKeepEven(f, a) {
    let fa = a.map(f);
    let res = [];
    for (let i = 0; i < fa.length; i++) {
        if (f(i) % 2 == 0) {
            res.push(f(i));
        }
    }
    return res;
}
Chapter 1. Practical Functional Programming

```javascript
for (let i = 0; i < fa.length; i++) {
  if isEven(fa[i]) {
    res = res.concat(fa[i]);
  }
}
return res;
}
```

```javascript
mapAndKeepEven((x)=>x*x, [0,1,2,3,4,5])
```

```javascript
== [ 0, 4, 16 ]
```

As the name of the function indicates, `mapAndKeepEven` is achieving two things at once - mapping a function and filtering the output according to the value of predicate (in our case, `isEven`).

The pattern of iterating over an array and keeping only elements that satisfy a condition is extremely useful.

FP languages include a function to make this operation easy to use - most often called `filter`.

Like `map`, we can invoke `filter` as a method of the Array object or as a function with 2 parameters:

```javascript
[1, 2, 3, 4].filter(isEven)
```

```javascript
== [ 2, 4 ]
```

```javascript
import { filter } from 'ramda'
filter(isEven, [1,2,3,4])
```

```javascript
== [ 2, 4 ]
```

Using filter, we can rewrite our function `mapAndKeepEven` in the following manner:

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

Or in a more concise manner:

```javascript
function mapAndKeepEven(f, a) {
  return a.map(f).filter(isEven);
}
```
We realize the function is quite general, and we can abstract away the `isEven` predicate:

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

mapAndFilter(cube, isEven, [0,1,2,3,4])
=> [ 0, 8, 64 ]
```

The `map` and `filter` functions are very convenient to chain together, when they are used as methods of the Array class:

```javascript
[0,1,2,3,4,5].map((x)=>x*x).filter((x)=>x%2==0)
=> [ 0, 4, 16 ]
```

**Compose**

Using Ramda’s `map` and `filter` functions, instead of the Array methods, we compose them as follows (as opposed to chain):

```javascript
filter(isEven, map(cube, [0,1,2,3,4,5]))
=> [ 0, 8, 64 ]
```

The operation that consists of composing functions is again a very general pattern. FP languages offer a function that receives as parameters a number of functions, and returns a new function which computes the composition of all the functions. The Ramda `compose` function works in the following manner: `compose(f,g)` returns the function that computes for all parameter `x` the value `f(g(x))`.

```javascript
import { map, filter, compose } from 'ramda'
let evenCubes = compose(filter(isEven), map(cube))
evenCubes([0,1,2,3,4])
=> [ 0, 8, 64 ]
```

This example demonstrates three important aspects of the FP approach:

- Functions can receive functions as parameters - including anonymous functions (lambda). For example, `map`, `filter` and `compose` receive functions as parameters.
- Functions can return functions as a computed value. For example, `compose` returns a new function as a computed value.
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.

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.

1.3.9 Reasoning about Code

Consider the following question about the function evenCubes: this is a function that first computes the cube of all the elements, and only then filters the results and keeps those that are even.

A friendly consultant asks us whether it would be possible to reverse the order of the operations - and first apply the filter and only then apply cube on the remaining elements.

That is, we want to compare the following two functions:

\[ \text{compose}(\text{filter}(\text{isEven}), \text{map}(\text{cube})) \]

and

\[ \text{compose}(\text{map}(\text{cube}), \text{filter}(\text{isEven})) \]

The motivation for this question is performance optimization: if we could filter first, and then apply the computation of the cube, we would save on the total number of computations performed to return the eventual result. That is, we know that \( \text{compose}(\text{map}, \text{filter}) \) is in general faster than \( \text{compose}(\text{filter}, \text{map}) \).

Naturally, we only want to perform this optimization if we can guarantee that the two functions are equivalent.

Function Equivalence

What does it mean that two functions are equivalent?

In the mathematical sense, a function maps from one set of values (the domain of definition) to another set of values (the range of the function). Under these conditions, we will say that: Given \( D \) the domain of the function \( f \), \( R \) the range of the function \( f \), \( f \) and \( g \) are equivalent, which we will write \( f \equiv g \) iff: \( g \) has the same domain \( D \), the same range \( R \) and \( \forall x \in D, f(x) = g(x) \).

In the functional programming paradigm, a pure function has no side effect (no shared variables are changed when the function is invoked). The same definition of function equivalence as for mathematical functions can be used for pure functions - except for two aspects that are specific to computation:

- a computation can throw an exception (result in an error)
- a computation (invocation of a function on parameters) can not terminate.
The definition of functional equivalence becomes then:

\[ f \text{ and } g \text{ (pure computational functions in the FP paradigm) are equivalent iff: whenever } f(x) \text{ is evaluated to a value, } g(x) \text{ is evaluated to the same value, if } f(x) \text{ throws an exception, so does } g(x), \text{ and if } f(x) \text{ does not terminate, so does } g(x). \]

**Referential Transparency**

Because this definition involves universal quantification (over all possible values of the parameter \( x \)), it is difficult to turn it into an operational process that can predict whether two functions are equivalent. It is the objective of programming languages semantic methods to provide tools to predict equivalence of programs.

The semantics of pure functional programs is much easier to develop than that of procedural programs with side effects - because it can be based on an inductive process of evaluation of expressions which only focuses on the structure of the input expression. We will develop such tools in the course, relying extensively on types and the technique of structural induction.

The property of functional programs which makes this process easy is called **referential transparency**: it means that the value of a program (called an expression in FP) depends only on its sub-expressions, and that if you substitute a sub-expression in an expression by another expression that is equivalent, then the resulting expression is equivalent to the original.

Some consequences of referential transparency are that if one evaluates an expression twice, one obtains the same result. The relative order in which one evaluates (non-overlapping) sub-expressions of a program makes no difference to the value of the program. This property enables optimization methods such as parallel evaluation of sub-expressions to speed up code.

**Proving Functions Equivalence**

In our simple example, we want to check whether the following two functions are equivalent:

\[
\begin{align*}
f &= \text{compose}(\text{filter(isEven), map(cube)}) \\
g &= \text{compose}(\text{map(cube), filter(isEven)})
\end{align*}
\]

We define typing - so that we will check this equivalence on the domain of finite arrays of integer values. That is, we want to determine, for any value of a finite array of integer values \( a = [a_1, \ldots, a_l] \) whether \( f(a) = g(a) \).
We proceed as follows:

\[
f(a) = \text{filter}(\text{isEven}, \text{map}(\text{cube}, a)) \\
= \text{filter}(\text{isEven}, [\text{cube}(a_1), \ldots, \text{cube}(a_l)])
\]

\[
g(a) = \text{map}(\text{cube}, \text{filter}(\text{isEven}, a)) \\
= \text{map}(\text{cube}, \text{filter}(\text{isEven}, [a_1, \ldots, a_l]))
\]

To continue this analysis, we need to express the effect of applying \text{filter} on an array. We write it as follows:

\[
\text{filter}(\text{pred}, [a_1, \ldots, a_l]) = [a_{i_1}, \ldots, a_{i_k}] \\
\quad | 1 \leq i_1 < i_2 < \ldots < i_k \leq l, \\
\quad \forall j \in \{i_1, \ldots, i_k\}, \text{pred}(a_j) \text{ is true} \\
\quad \text{and} \\
\quad \forall j \notin \{i_1, \ldots, i_k\}, \text{pred}(a_j) \text{ is false.}
\]

Using this definition of the way \text{filter} is computed, we obtain (we skip the false conditions):

\[
f(a) = [\text{cube}(a_{i_1}), \ldots, \text{cube}(a_{i_k})] \\
\quad | \forall j \in \{i_1, \ldots, i_k\}, \text{isEven(\text{cube}(a_j))} \\
\quad \text{and} \\
g(a) = \text{map}(\text{cube}, [a_{j_1}, \ldots, a_{j_m}]) \\
\quad = [\text{cube}(a_{j_1}), \ldots, \text{cube}(a_{j_m})] \\
\quad | \forall j \in \{j_1, \ldots, j_m\}, \text{isEven}(a_j)
\]

To prove the equivalence, we must then prove that:

\[
\forall i \in N, \text{isEven}(i) = \text{isEven}(\text{cube}(i))
\]

We leave the end of the proof to the reader.

Note that the logical axiom we use that represent the relation between the expression \text{filter}(\text{pred}, a) and its value, or between the expression \text{map}(f, a) and its value are the key steps of the logical proof of equivalence between the expressions.
Computation Steps

The key point of this example is that we can predict the equivalence of two functions by applying **reduction steps**, where the invocation of a function on arguments is described as a sequence of steps, where each step is described as a computation rule. We have seen above examples of such steps for the reduction of `map` and of `filter`.

1.4 Summary

1.4.1 Programming Paradigms

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

- Imperative, Structured, Procedural, Functional, Object-Oriented, Event-driven, Flow-driven, Logic are examples of programming paradigms.

- Programming Languages in their design make some paradigms easy to adopt. Some languages support multiple paradigms.

- Programming paradigms have evolved over time as reaction to problems observed in programming practice.

- Paradigms have evolved to reduce code repetition, facilitate testing, enforce abstraction barriers to ease code reuse, encourage safe concurrency, allow performance optimizations through declarative programs, and allow reasoning about code.

1.4.2 Functional Programming

- FP is a paradigm that encourages the usage of pure functions (with no shared state and mutation), and referential transparency (the value of an expression depends only on its sub-expressions).

- FP achieves **functional abstractions** by using **higher-order functions**: one can define anonymous functions (lambda), functions can be passed as arguments to other functions, and functions can return computed functions as values.

- `map`, `filter` and `compose` are highly reusable functional abstractions that operate over collections of values.

- FP makes it easy to perform concurrent computation because there is no shared mutable state used in computation - and thus, no need to define risky locking mechanisms.
1.4.3 Semantics of Programming Languages

- One of the objectives of the field of semantics of programming languages is to predict the equivalence of programs.

- Tools that support semantic analysis of programs include types, structural induction and the analysis of computation as a sequence of reduction steps.