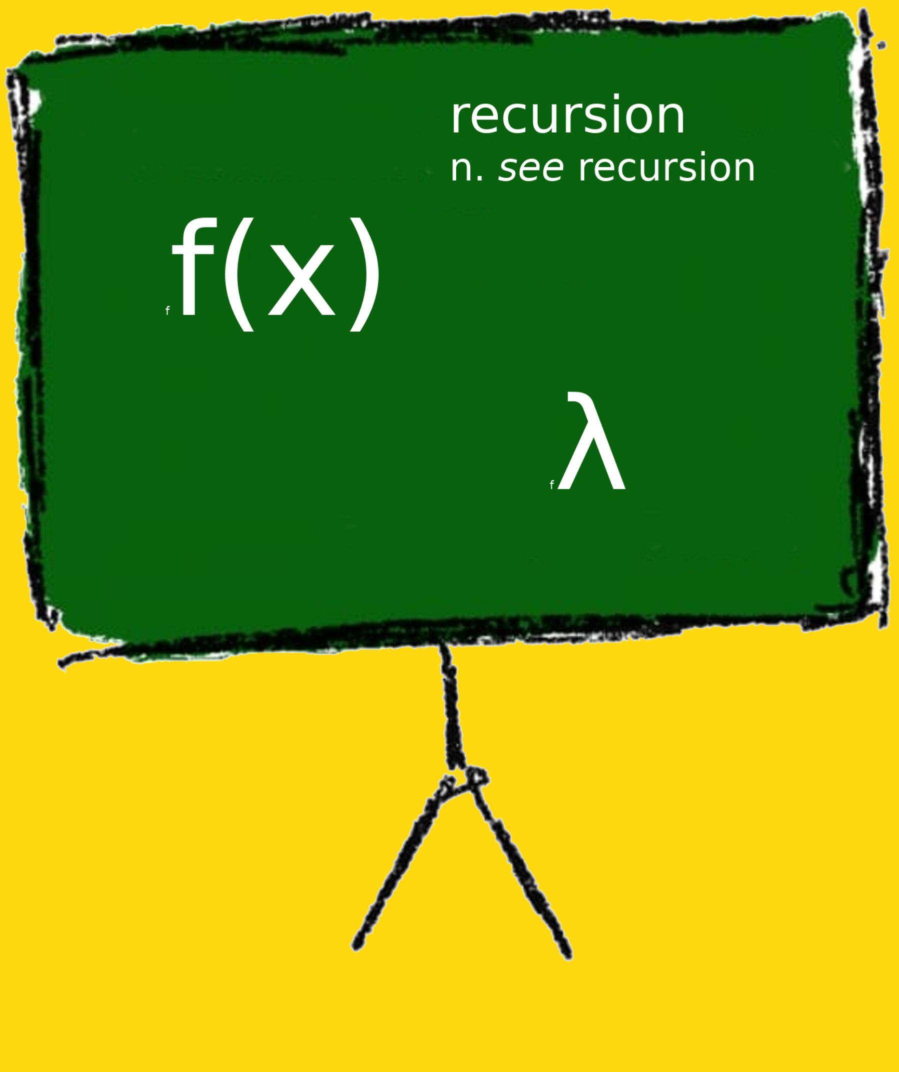
Get Started With Functional Programming



Goran Jovic

Get Started With Functional Programming

Goran Jovic

This book is for sale at http://leanpub.com/learning-functional-programming-from-scratch

This version was published on 2023-08-06

Contents

Introduction

About the book

The purpose of this book is to explain the main concepts in functional programming as simply as possible in a language agnostic manner. That is not to say that we will not use actual code - rather that the focus is not on implementation details of a specific language, compiler or platform, but on the concepts themselves.

The examples are provided as:

- drawings and diagrams
- pseudocode
- JavaScript - most of the time as simple as pseudocode, but you can actually run it!
- various other languages showcasing a specific point, e.g. HTML, CSS, SQL, Clojure and others.

Who is this book for?

Anyone who wishes to learn more about functional programming! Some working knowledge about programming basics may be helpful, but ultimately it's not a show stopper - just another thing to learn.

If you just started learning functional programming, my own personal recommendation is that you go through this book first, then master at least one concrete language and then go through it again as a refresher.

Who is this book NOT for?

Anyone who wishes to learn a specific technology. You will not learn Clojure or JavaScript or any other programming language. You will not learn a MapReduce framework or a BigData pipeline platform. You will learn the basic building blocks behind all those technologies, but that's just about it.

Anyone who wishes to use it as a reference index, academic source or anything similar. Whenever I had to choose between clarity and simplicity of explanations on one side and formal correctness on the other, I chose simplicity. So, while you can quote the examples to your friends and colleagues and get some geek cred for it, you're not getting a PhD for it (and neither am I!)

Programming paradigms

The popular wisdom says that programming is like giving directions to a very dumb person. It is a rather handy analogy and you can use to explain to your grandparents what you do for a living. However, it doesn't quite hit the mark.

I mean, even the dumbest person you could possibly imagine is still orders and orders of magnitude more complex than all your computers put together. So, it is not really like that. It's not even like teaching a monkey to ride a bicycle. That cool online app that recognizes people's expressions does not have hardware worth millions of years of evolution between its ears. All it has is ones and zeroes - that and a whole lot of layers of abstraction.

But you already know all that! If you didn't, you wouldn't be here reading a book about programming paradigms. Still, it makes a fine ice breaker, so here it is - I said it.

So, what's the point with all the paradigms, you might ask? Well, with all those layers of complexity our modern software has we need to be able to:

1. Develop it successfully or at all.
2. Maintain it, preferably in a way where it does not collapse like a house of cards.
3. If anyhow possible, keep our own sanity while we do so.

Using the right way to mold and shape our programs moves us one step closer to these noble goals, and we do that by choosing the most appropriate paradigm.

There are lots of paradigms, but these three are the most important for us at the moment:

- **Imperative programming** - programs consist of ordered lists of statements that get executed sequentially, changing the computer's state while doing so. This has been the most common programming style for a very long time and conceptually it is closest to how computers actually work. The list of example languages is very long, but some of them are: C/C++, Java, JavaScript, and so on.
- **Declarative programming** - programs are actually accurate specifications of the desired outcomes, rather than lists of instruction how to achieve them. The most common examples are HTML, CSS and SQL.
- **Functional programming** - mathematical functions are the main building blocks and programs rely on their evaluation given certain parameters. Functions are supposed to be pure - their outcome may only depend on the input parameters and they may not have any side effects. This property makes functional programming very close to the declarative paradigm. Some of the examples are Haskell, Clojure (and other Lisps), Erlang and so on.

Imperative programming

Let's say we need a cup of coffee. What we have at our disposal are one water kettler, one mini mixer, an empty cup and a teaspoon. We also have some coffee granules, sugar, water and milk. What we want as the outcome of the exercise is a cup of coffee and this is one way we can get it:

```
Make coffee

1. Put a teaspoon of sugar into the cup
2. Put a teaspoon of granules into the cup
3. Boil water in the kettler
5. Pour water to fill half the cup
6. Mix the contents of the cup
7. Add milk to fill the rest of the cup
8. Mix again
9. Serve
```

And there it is - a piece of pseudocode with an imperative program that explains how to make a cup of coffee. Yes, recipes are programs, all right!

What are some initial observations we can make:

- Order of the steps matters *a lot* - if you execute steps 5 and 7 before any other steps, you'll mix an empty cup twice and end up with unmixed sludge.
- The reason why order matters is because every step modifies the state of our cup and the next steps depend on that modification in order to work properly. Step one changes the cups state from `empty` to `contains sugar`. Step two changes it from `contains sugar` to `contains sugar and granules` and so on.
- Therefore, **mutable state** makes **order of execution** important.

What else can we see:

- Not only does our program modify the state of our cup, it also modifies the state of the outside world - certain quantities of all the ingredients are consumed, our electricity bill increased just a little bit and finally, in step 9, we got our cup of coffee and our energy level just jumped. We'll refer to all of these as **side-effects**.
- Since we do consume those resources, the next time we repeat the same program we may find that we ran out of milk. Or that it smells funny! Or the kettler just died, only a few days after warranty expiration to make things worse. So, not only does our program change the state of the outside world, it also **depends on global state** as well.

We can expand our definition and say that imperative programs consist of an ordered sequence of statements or commands that depend on current state of the world and cause changes to it in the process.

Structured programming

Let’s expand our example a little bit. Not everyone likes their coffee the way I do, so let’s take that into account and make the number of teaspoons of sugar variable. While we’re at it, let’s also make milk optional - if it is not wanted, we’ll just put water instead.

```
Make coffee
milk : yes | no
sugar : number of teaspoons

1.  Count the number of teaspoons of sugar added - starting at 0
2.  Put a teaspoon of sugar into the cup
3.  Increase the count by 1
4.  If we counted less than the desired amount jump back to 2
5.  Put a teaspoon of granules into the cup
6.  Boil water in the kettler
7.  Pour water to fill half of the cup
8.  Mix the contents of the cup
9.  If milk is wanted jump to 10, otherwise jump to 11
10. Add milk to fill the rest of the cup
11. Add water to fill the rest of the cup
12. Mix again
13. Serve
```

This program is technically correct - go on, step through it a few times and check it yourself. However, it is well... really ugly and not the easiest thing to read and let alone maintain. The main culprits are the parts where we jump to arbitrary steps as we go along - the infamous `goto` statement.

This issue was the reason for the first real *upgrade* imperative programming got - a variation of the paradigm called **structured programming**.

By now, we may as well ditch pseudocode and continue with JavaScript:

```
let cup = {sugar : 0, coffee: 0, milk : 0, water : 0, done : false};

let sugar = 2;
let milk = true;

for(let counter = 0; counter < sugar; counter++){
        cup.sugar += 1;
}
cup.coffee += 1;
console.log('boiling water...');
cup.water += 0.5;
```

```javascript
console.log('mixing...');
if(milk){
   cup.milk += 0.5;
}else{
   cup.water += 0.5;
}
console.log('mixing...');
cup.done = true;

console.log(cup);
```

Now this is an example you can actually run! Just copy paste it into your favorite JavaScript console.

Just like in our pseudocode example, we start with an empty cup. Here it is represented with a JavaScript object with a field for each ingredient and one additional flag field that marks a "done" cup of coffee.

Then we have two variables that hold our coffee preferences. In this case it is a cup of coffee with milk and two teaspoons of sugar.

What follows is the main improvement that structural programming brought us - `for` loop and `if` conditional statement. These operations are called **control flow statements** since they do exactly that - control the flow of the program. There are two kinds of control flow statements: **conditional statements** and **loops**.

We only need to take a look back at our previous example to see that conditionals correspond to goto jumps forward, skipping some statements that do not meet the criteria. Likewise, loops correspond to jumping backwards, so that the same statements are executed more than once.

Procedural programming

Let's expand our example even further. So far we assummed that the cup is initially empty and clean. Of course, that's not how the world works and a lot of the time our coffee preparation starts with dish washing. So, let's add some code that simulates this:

```javascript
cup.water = 0;
cup.milk = 0;
cup.done = false;
```

We will need to run these few statements every time we want to make a new cup of coffee, right before all the other statements. But we also may want to clean the cup as a separate task, even if we don't want need the coffee immediately. So, these statements need to be used in two different situations.

What do we do? Do we copy-paste them and end up with two identical chunks of code in our program? There is a much better way to reuse this code - **procedural programming**.

Basically, the main innovation in procedural programming was introduction of named reusable pieces of code that capture some common functionality that needs to be invoked from different parts of the program. These modules were usually referred to as **subroutines** and there are two types of them:

- **functions** - accept zero or more parameters and return a result value. Note: this is not necessarily the same as mathematical functions or the ones from functional programming.
- **procedures** - accept zero or more parameters, change the global state and do not return a result value.

Since we need to change the state of a cup, we will need to write procedures. JavaScript only allows us to define functions, but that is not a big deal, we can just ignore the return value - if you omit a `return` statement, the result will just be a special `undefined` value. So, here we go:

```
let cup = {sugar : 0, coffee: 0, milk : 0, water : 0, done : false};

function cleanCup(){
    cup.water = 0;
    cup.milk = 0;
    cup.done = false;
}

function prepareCoffee(sugar, milk){

    cleanCup();

    for(let counter = 0; counter < sugar; counter++){
        cup.sugar += 1;
    }
    cup.coffee += 1;
    console.log('boiling water...');
    cup.water += 0.5;
    console.log('mixing...');
    if(milk){
       cup.milk += 0.5;
    }else{
       cup.water += 0.5;
    }
    console.log('mixing...');
    cup.done = true;

    console.log(cup);

}

prepareCoffee(2, true);
```

We start with the empty cup initialized. Then we have two function definitions. The second one has two parameters - one for each coffee making option. Finally, we have a function call to `prepareCoffee` with two teaspoons of sugar and milk.

What happens next is that the the very first line in our coffee making function is a call to another function `cleanCup`. That's how we ensure that the following code will always start with a clean cup.

And that's how you re-use a cup. Also code!

Feel free to play with this example in the console, e.g. try preparing coffee a few times, then try cleaning the cup and check the value of the `cup` variable and so on.

Object-oriented programming

We started with a simple example, then added some more complexity to it. We found ways to keep the structure of the program well organized and even had some level of code re-use.

Obviously, there is still *something* wrong with the current version, otherwise we wouldn't be introducing a new paradigm. Well, for starters, there is nothing to prevent anyone from running a statement like this:

```
cup.spit += 0.1;
```

The `cup` variable is completely exposed to whoever wishes to tamper with it. Ideally, we only want the cup to be changed using our procedures and not in any other way. We want to prevent anyone from spitting in our coffee. We will do that using the hallmark principle of object-oriented design - **encapsulation**.

What we need to do is bundle the data structure that holds the current state of the cup with the operations that change it. Then we need to restrict any access to it to those operations alone.

```
let Cup = function(){

   let sugar = 0;
   let coffee = 0;
   let milk = 0;
   let water = 0;
   let done = false;

   this.cleanCup = function(){
      sugar = 0;
      coffee = 0;
      milk = 0;
      water = 0;
      done = false;
   };

   this.serve = function(){
      console.log({sugar : sugar,
                   coffee : coffee,
                   milk : milk,
                   water : water,
                   done : done});
   };

   this.prepareCoffee = function(neededSugar,
                                 neededMilk){
      this.cleanCup();

       for(let i = 0; i < neededSugar; i++){
           sugar += 1;
       }
       coffee += 1;
       console.log('boiling water...');
       water += 0.5;
       console.log('mixing...');
       if(neededMilk){
          milk += 0.5;
       }else{
          water += 0.5;
       }
       console.log('mixing...');
       done = true;

```

```
    };

}
```

We created another function `Cup` whose purpose is to create and initialize an empty cup and we bundled our previous functions within it. We had to rename a few things here and there, e.g. `prepareCoffee` parameters are now called `neededSugar` and `neededMilk` to avoid the naming conflict and also to emphasize that they refer to the expressed coffee preferences not the current amount of sugar or milk in the cup. Just to make things clearer we also created a new function `serve` to well… serve the cup as it is at the moment.

The main units of object-oriented programming are **objects** - data structures that consist of:

- **fields** - variables that contain the current state of the object - in our example variables `sugar`, `coffee`, `milk`, `water`, and `done`.
- **methods** - functions that access or manipulate fields of its own object - `cleanCup`, `serve` and `prepareCoffee`.
- **constructors** - special functions that create and initialize the object - `Cup` function.
- **destructors** - special functions that destroy the object and free up any taken resources. Since JavaScript is a garbage collected language, we don't have this in our example.

Both fields and methods alike are referred to as **members**.

To achieve the encapsulation we need to make sure that some members are accessible from anywhere in the program, while others are only accessible from the object itself, specifically from its methods. Some other languages like Java or C# have explicit member access modifiers like `public` and `private`. In JavaScript we get the functionality of private members by defining the local variables with `var` within the constructor and the ones we assign with `this` are treated as public.

Please note that both fields and methods may be either public or private. It usually makes more sense to keep the fields private and methods public, but it is perfectly possible to have a private method or a public field.

We have a definition of an object, now let's use it:

```
let cup = new Cup();         //create a new cup
cup.prepareCoffee(2, true);  //prepare the coffee
cup.serve();                 //serving the cup
//=>  {sugar: 2, coffee: 1, milk: 0.5,
//=>   water: 0.5, done: true}
cup.cleanCup();              //cleaning
cup.serve();                 //now it's empty
//=> {sugar: 0, coffee: 0, milk: 0,
//=>  water: 0, done: false}
```

What if we wanted a few more cups?

```
let cup = new Cup();              //create a new cup
cup.prepareCoffee(2, true);       //sweet and creamy
cup.serve();
//=> {sugar: 2, coffee: 1, milk: 0.5,
//=>  water: 0.5, done: true}

let another = new Cup();          //create another cup
another.prepareCoffee(0, false);
another.serve();
//=> {sugar: 0, coffee: 1, milk: 0,
//=>  water: 1, done: true}
```

We just need to create another cup, prepare the coffee in it and serve. Note that if we were to just call `prepareCoffee` on the same object twice we'd get one coffee, then spill it, clean the cup and prepare the other one.

Let's test the encapsulation too. We shouldn't be able to see or modify the private variables directly:

```
cup.milk;
//=> undefined

cup.coffee;
//=> undefined

//the only way we can get this info
//is by calling serve

cup.serve();
//=> {sugar: 2, coffee: 1, milk: 0.5,
//=>  water: 0.5, done: true}

//let's try modifying some of the values
//e.g. we can try messing up the amount of sugar

cup.sugar = 7;
cup.serve();
//=> {sugar: 2, coffee: 1, milk: 0.5,
//=>  water: 0.5, done: true}

//still the same - it works!
//but see what happens if you thy this now:

cup.sugar
```

```
//=> 7

//a new "public" field with the same name as
//the "private" field was created automatically,
//but the private one was not affected
//It's a JavaScript gotcha
//you can ignore it for the time being.

//Let's try spitting in coffee:

cup.spit = 0.1;
cup.serve();
//=> {sugar: 2, coffee: 1, milk: 0.5,
//=>  water: 0.5, done: true}

//The coffee is untainted. Mission accomplished!
```

We've seen all three major evolution steps of imperative programming so far: structured, procedural and object-oriented. That and made a lot of coffee following step-by-step instructions. It is time to move on and try out some other ways we can write programs.

Declarative programming

Again, let's say we need a cup of coffee. But, this time we just walk into the nearest coffee shop and say:

```
One coffee with 1 teaspoon of sugar and milk, please!
```

And there's your coffee - no fuss over anything. You just declare the desired outcome and let the baristas take care of it.

The main benefit of this approach is a very high level of abstraction. Practically all the details we were so careful about in imperative programming are of no consequence anymore.

Benefits:

- There is no dependence on **mutable state**. Notice how we never even mention the cup - the coffee shop will take care of it and all its intermediate states. We only get the coffee cup when it's done.
- Without the mutable state, we no longer care about the **order of statements** in our program. Is it one with milk and sugar and one without milk or sugar, or is it the other way round? It does not matter!

- We don't care about the **global state** either - we cannot spend all the milk in the coffee shop and do not depend on its availability. The shop takes care of that in a way which is transparent to us. Our own kitchens don't work that way.

But it can't be all nice and cool, there have to be some drawbacks, too. Otherwise, we'd never do any imperative programming at all.

Drawbacks:

- We need a coffee shop to be able to get away with this, or at the very least someone willing to prepare the coffee. Try saying *One coffee with 1 teaspoon of sugar and milk, please!* out loud in your kitchen. If you do, you'll just look ridiculous and most importantly won't be getting any coffee. Incidentally, whoever gets to actually make the coffee will do so by following an imperative program.
- We are constrained by the variety of coffee shop's offer. There may be a selection of several kinds of coffee and a few different sizes, but that's it. Want coconut milk instead of the regular one? At home you can add it yourself - in a cafe it's possible only if they have it on their menu.
- We can't affect the performance in any way. There is a good chance that the baristas will be faster than you, but if they are not, you can't do anything about it. Simply, you don't get to tinker with the imperative program they use - it's completely hidden away from you.
- Different cafés all serve coffee, but in every one of them the flavor, quantity and the general experience are slightly different.
- 5$ for a cup!

Now that we went through what declarative programming is, let's see some real world uses. The most commonly mentioned example is *HyperText Markup Language*, aka HTML.

```html
<html>
  <head>
    <title>A cool page</title>
  </head>
  <body>
     <div id="main">
        <p>Hello bright and sunny world!</p>
     </div>
  </body>
</html>
```

Here we have one HTML snippet that defines a very simple web page. All it contains is a title and one paragraph. These two texts are enclosed within a structure marked with HTML tags. That's all there is to HTML - we only write the content and declare what's what.

If we want to affect the style, we'll use another declarative language *Cascading Style Sheets* or CSS.

```css
#main{
  width: 70%;
  margin-left: 15%;
  margin-right: 15%;
}

p{
  color: blue;
}
```

In this example, we say that width of the element with id `main` is 70% of its parent width. Parent is the element within which our element is enclosed, so in this case it is page `body`. We also said that the remaining 30% of the whole page width will be evenly distributed between the left and right margin, i.e. 15% each. Then, we also declared that text in all paragraph elements (denoted by `p` tags) will be blue.

Again, the difference is that in an imperative program we would instruct the computer how to draw the page and format it. With HTML and CSS we only specify the desired outcome and wait for the results. Just like in the coffee example, for this to work we need to have software that's going to actually do all the heavy lifting and draw the page - a *web browser*.

Another commonly cited declarative language is SQL. Let's see one example query:

```sql
SELECT name FROM countries
WHERE population > 10000000 AND continent = 'Europe';
```

We just say that we want the names of all the countries in Europe with population above 10 million. Do we loop through all the records in the database and check the continent and population one by one? Do we use a database index to speed up the search? Is the underlying structure of the index a B-tree or something else? Do we check the population first and then continent or the other way round? Short answer to all the questions above is - we don't care! All those issues and many more are relatively transparently handled by databases' very own coffee shops - RDMS (Relational Database Management Systems).

You must have noticed that all the major examples of declarative programming are focused on rather narrow domains. HTML, CSS and SQL are all what we call *domain specific languages* or DSLs. Not all DSLs are declarative, although most widely known declarative languages are domain specific.

So, how do we build our own declarative language?

One traditional way is to define a syntax to express the desired outcome, then use imperative programming to build a program to interpret the input and produce results. That's exactly how web pages are processed - there is a well-defined syntax (HTML and CSS) and there are programs developed using imperative style of development that process it, i.e. browsers.

Another common way is to define a syntax, then build an engine that transforms the input in that syntax into a sequence of instructions to be executed, i.e. an imperative program. This is more or less how database management systems work. You give them your SQL, they transform it into something called *execution plan*, then the plan gets executed and you get results. We can even choose to only get the plan without executing it - to see this in action just prepend `EXPLAIN` to your query and run it in your database system of choice.

But can we do better than that? Can we have the goodness and ease of mind that declarative programming offers in general purpose software development or are we condemned to the perils of imperative programming as soon as we venture beyond the well-defined borders of specific domains?

These are the questions we will attempt to answer with functional programming!

Functional programming

Liked the taste of declarative programming? Want to generalize it and write arbitrary code in a declarative fashion, not just HTML pages, database queries and overpriced coffee?

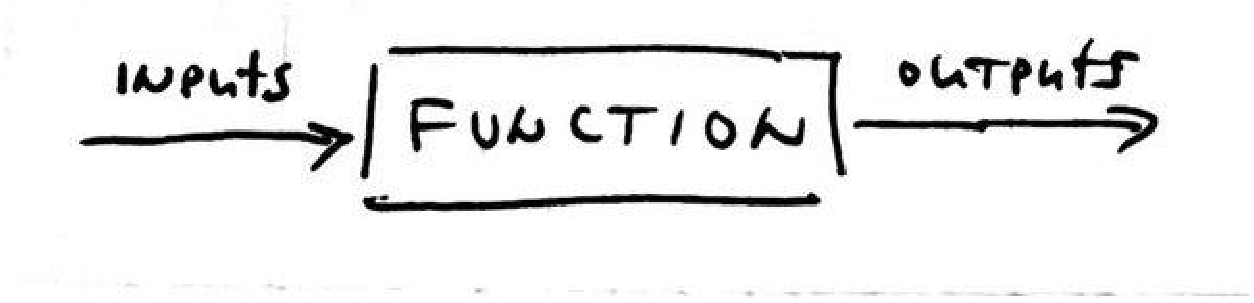
Let’s see what we can learn from all the important concepts employed by declarative programming and the evolution of imperative styles.

- Dependence on global state is bad.
- We should avoid affecting the global state as much as possible.
- If we don’t have any mutable state, the order of our code won’t matter.

So, how do we get from *here* to *there*?

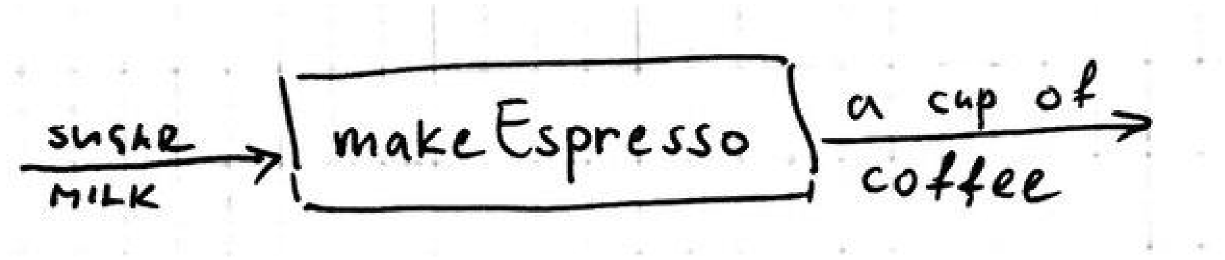
1. We’ll start with our friends **functions** from procedural programming. They represent isolated, well-defined and reusable units of work - that’s good.
2. Since we want to limit the dependence on global state, we’ll only allow the function **parameters** to affect the behaviour of the code, not any variables in the rest of the program.
3. Since we want to limit the effects our code has on the outside world, we’ll only output the results as the function **return value**. Other than that, any **side effects are prohibited**.
4. Instead of mutating variables we will produce new **immutable values** based on the old values.

The easiest way we can visualize a function like that is a pipe. It receives all its inputs on one side, does some work internally, then produces all the results on the other side.



function

Our coffee example would involve one coffee producing pipe that receives information on what kind of coffee you want on one side and outputs a hot and ready cup on the other. Kind of like an espresso machine, actually.



makeEspresso function

We will simulate this with a JavaScript function that takes the desired sugar and milk options as the input and produces a cup of coffee.

```
function makeEspresso(sugar, milk){
  if(milk){
    return {sugar: sugar, coffee : 1,
            milk : 0.5, water : 0.5};
  }
  return {sugar: sugar, coffee : 1,
          milk : 0, water : 1};
}
```

Let’s compare this function against our checklist:

- Does not depend on any variable outside its scope - the only things that affect its behavior are the parameters. Check!
- The only way the function affects the outside world is by returning a value. Check!
- While JavaScript objects are mutable by default, see that we never actually mutated the objects that represent the cup - we always created new objects. Check!
- Since we didn’t mutate any state, the order of our code does not matter as much. Instead of checking if milk is wanted, we could have checked the opposite. The following code is just as good:

```
function makeEspresso2(sugar, milk){
  if(!milk){
    return {sugar: sugar, coffee : 1,
            milk : 0, water : 1};
  }
  return {sugar: sugar, coffee : 1,
          milk : 0.5, water : 0.5};
}
```

One unsaid requirement for all this to work is unlimited supply of empty cups, sugar, coffee, water and milk. Also let's not forget the electricity the machine needs to run. We will simply hand wave all those details and smugly declare that low level resource management is not a functional way of doing things!

Anyway, let's make a few espressos:

```
makeEspresso(2, false);
//=> {sugar: 2, coffee : 1, milk : 0, water : 1}

makeEspresso(0, true);
//=> {sugar: 0, coffee: 1, milk: 0.5, water: 0.5}

makeEspresso(1, true);
//=> {sugar: 1, coffee: 1, milk: 0.5, water: 0.5}

//just for fun try the other one too
makeEspresso2(1, true);
//=> {sugar: 1, coffee: 1, milk: 0.5, water: 0.5}
```

Notice how we always get a brand new cup? Instead of reusing it, we have to dispose of it. That's not very green! Seriously, it's not a coincidence that *garbage collection* was first used in a functional programming environment.

We lost a lot of low level control over resources, but what we gained is a piece of declarative code that we can actually run. Just look at it, this snippet:

```
makeEspresso(1, true);
```

It's nothing other than our declarative pseudocode thinly disguised as JavaScript:

```
One espresso with 1 teaspoon of sugar and milk, please!
```

You declare what you want, not how you get it.

So, we ventured from an imperative program to a declarative one and thanks to functional programming we didn't even need to build a coffee shop to do so. We managed all that because we adhered to the principles enumerated at the start of this chapter. We will explore those principles and functional programming in general throughout the rest of the book.

So, stay with us - the good stuff has just started!

Theory

The very central concept in functional programming is **expression**.

Expression is a piece of code that can be evaluated, which is to say that it can be used to produce a value. In functional programming, everything is an expression and everything produces or *returns* a value.

There are two types of expressions:

- plain values
- function calls

Plain values are the elementary expressions, e.g. in JavaScript.

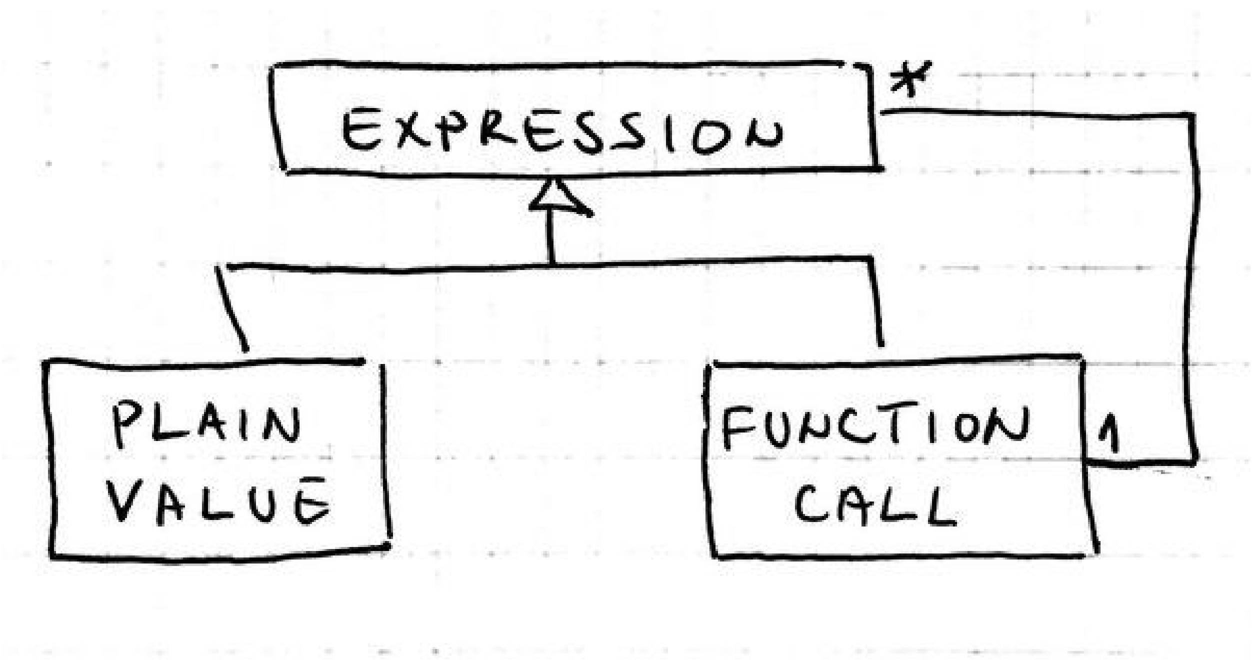
```
2
true
"hello"
{sugar: 2, coffee: 1, water: 0.5, milk: 0.5}
```

Function calls are complex expressions that receive other expressions as parameters, produce a result based on them and then return it as another value.

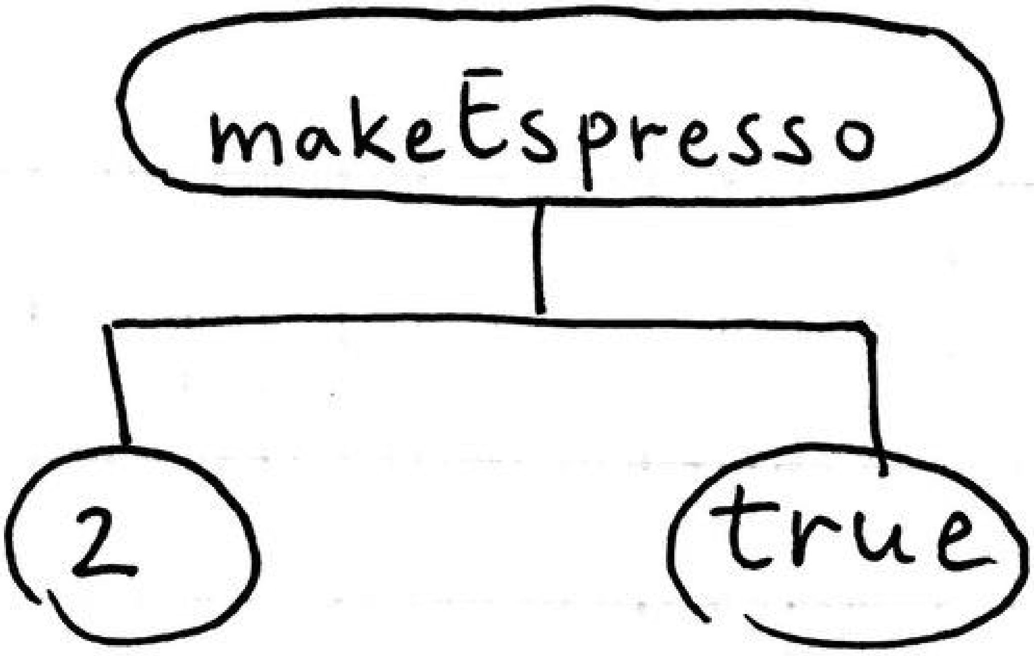
```
makeEspresso(2, true);
//=> {sugar: 2, coffee: 1, water: 0.5, milk: 0.5}
```

In this example we supply the function with two values as parameters that specify coffee options and the function gets evaluated to the value that represents the coffee cup.

We can represent the relationship between elementary values and functions with an obligatory UML class diagram.



See the recursive link from function back to expression - that's because function parameters may be either plain values or nested function calls. Basically, an expression has a tree structure, so let's also draw it as a tree.



Expressions are practically trees, where children are parameters to the sub-expressions (i.e. functions) and plain values are leaves.

Note that in many languages you can build complex expressions not only with functions, but also with **operators**. JavaScript is one of those languages, e.g. if you wanted to get a sum of two numbers you'd do this:

```
5 + 4
//=> 9
```

You supply two numerical values, put the + operator between them and the whole expression gets evaluated to the sum of those numbers. Although operator expressions are not necessarilly implemented in the same technical way as actual functions are, logically they serve the same purpose. The main difference is in notation. JavaScript functions use *prefix* notation - function name goes first, then parameters. Operators, on the other hand use *infix* notation - parameter, operator, then the other parameter.

Some languages, like Clojure and other Lisps, make no such distinction and exclusively use prefix notation, even where other languages would have used infix operators:

```
;clojure

(+ 5 4)
;=> 9
```

In other languages, like Haskell, operators are just functions with two parameters whose names consist of symbols rather than alphanumeric characters, and may be used in either infix or prefix notation.

```
--haskell

--infix operators
4 + 5
--=> 9

--prefix operators
(+) 4 5
--=> 9
```

Likewise, Haskell functions with alphanumeric names are usually called with prefix notation, but can also be called infix-style using a special syntax:

```
--haskell

--prefix functions
mod 3 2
--=> 1

-- infix function
3 `mod` 2
--=> 1
```

We can also use the value of an evaluated expression as the input parameter of another expression, e.g. another JavaScript infix operator expression or a function:

```
10 - (5 + 4)
//=> 1

makeEspresso(10 - (5 + 4), true)
//=> {sugar: 1, coffee: 1, water: 0.5, milk: 0.5}
```

We've got a handle of the two central concepts of functional programming. However, it does not end there - there are also some constraints required for all the magic to happen. We'll show that, whenever possible, functions have to be **pure** and values have to be **immutable**. So, let's demystify that!

Pure functions

We'll start with the most commonly used definition of a pure function, then we'll provide a few examples of pure and impure functions, with different causes of impurity. Then, we will try to illustrate function purity from a few different angles. Anyway, let's start with the definition.

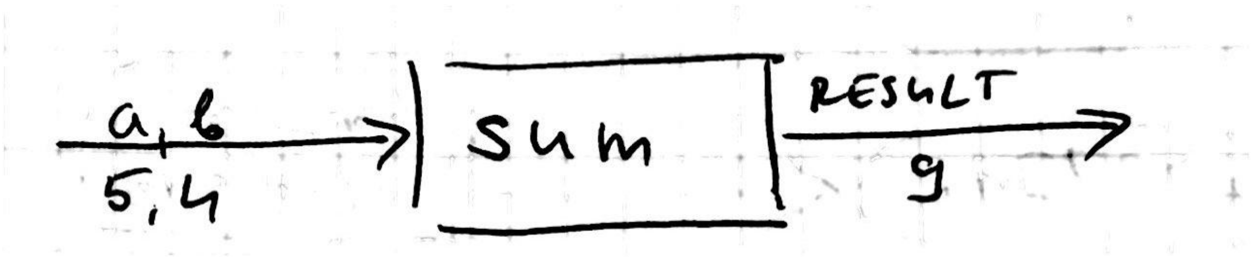
Pure function is a function where the return value is:

1. only determined by its input values,
2. without observable side effects.

If we continue with the pipe analogy, this means that if anything was to enter the pipe must do so at the entrance, as a parameter, and the only thing that leaves does so at the other end, as the return value.

For example addition is a pure function:

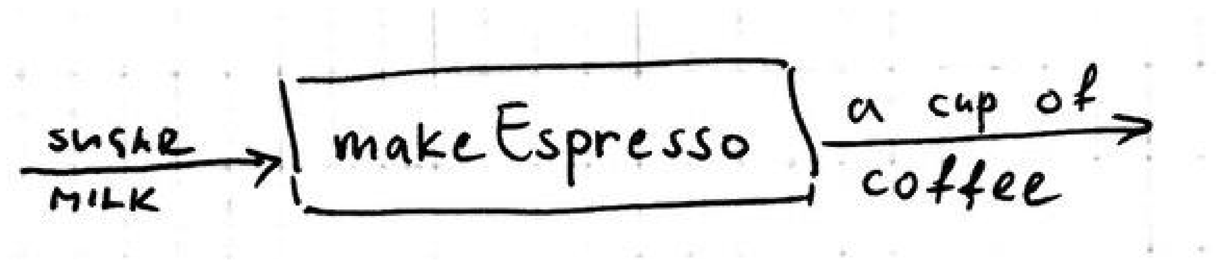
```
function sum(a, b){
   return a + b;
}

sum(3, 5);
//=> 9
//we get the result value and nothing else happens

//we call the function with same parameters
sum(3, 5);
//=> 9
//and get the same result
```



sum function

So is our espresso loving example:

```javascript
function makeEspresso2(sugar, milk){
  if(!milk){
    return {sugar: sugar, coffee : 1,
            milk : 0, water : 1};
  }
  return {sugar: sugar, coffee : 1,
          milk : 0.5, water : 0.5};
}

makeEspresso(2, false);
//=> {sugar: 2, coffee : 1, milk : 0, water : 1}
```



makeEspresso function

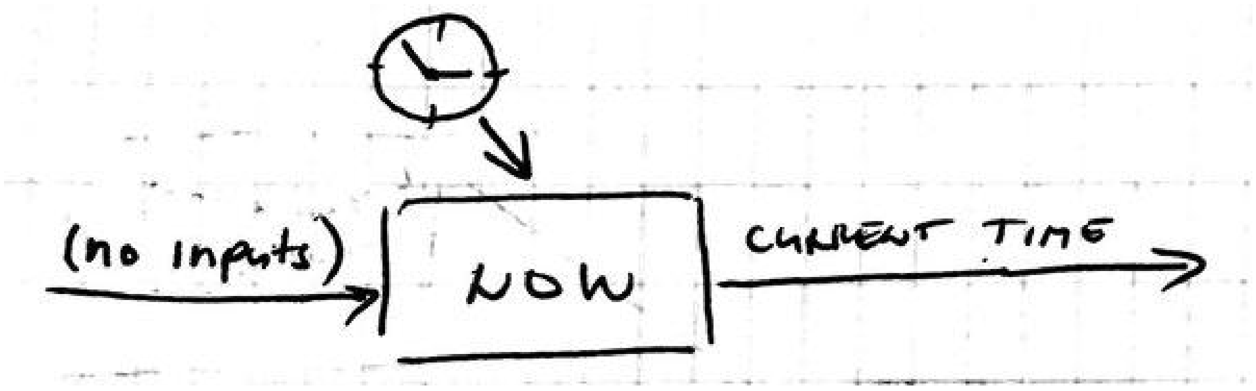
The only thing these functions depend on are their received parameters and the only effect they produce are their return values - there are no side effects. The pipes have no leaks, in the case of espresso machine quite literally.

Now, what's the easiest way to violate the first condition for purity? The offending function would need depend on something other than its input parameters. The effect would be that it would return different values with the same inputs.

```javascript
function now(){
  return new Date();
}

//we call it once
now();
//=> Wed Jun 14 2017 19:53:19 GMT+0200 (CEST)

//then again five minutes later...
now();
//=> Wed Jun 14 2017 19:58:19 GMT+0200 (CEST)
```

You'd expect a function with no parameters to always return the same value. Practically it should be equivalent to a constant.

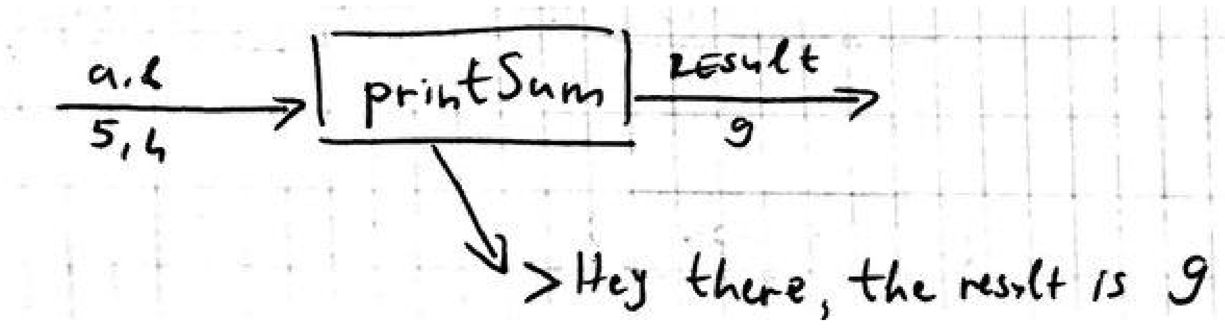
What happens here is that our function depends on something other than its non-existent parameters - the system clock:



now function

And how would we violate the second condition? Our function will need to somehow change the state of the world around it in a way other than by returning the value. E.g. it can save something to the disk, delete a row from the database, send an email or a text message, make the phone vibrate, etc. Since we want our example to be easily runnable in an average JavaScript shell we'll settle with a more modest choice - logging to the console:

```javascript
function printSum(a, b){
   let sum = a + b;
   console.log('Hey there, the result is: ' + sum);
   return sum;
}

printSum(4, 5)
//=> Hey there, the result is: 9
//=> 9

//We do get the return value, but we also get a console message!
```

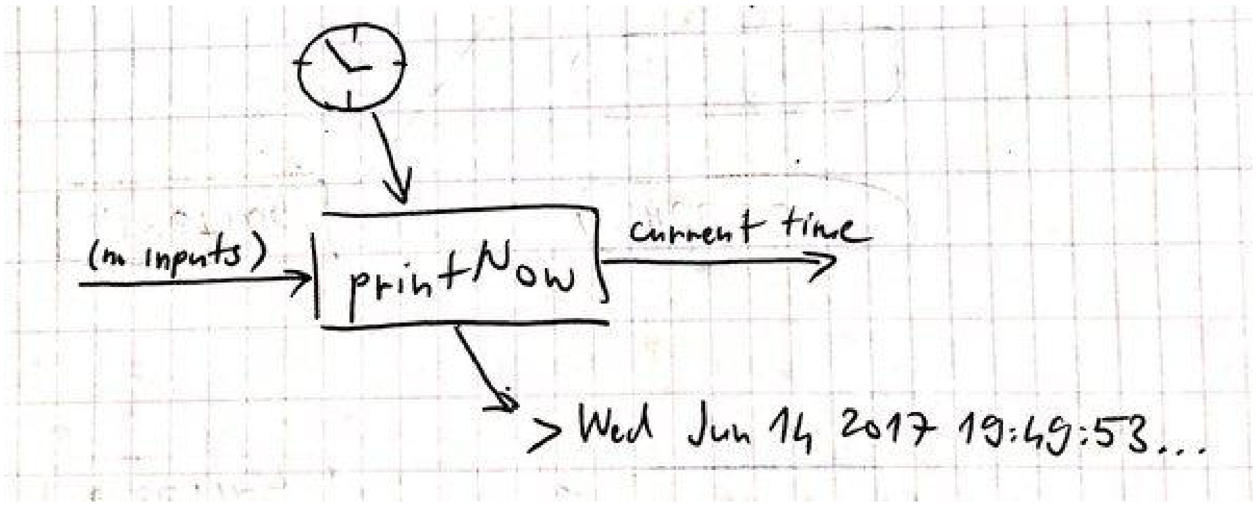


printSum function

To summarize, functions `sum` and `makeEspresso` are pure because they satisfy both purity conditions. On the other hand, functions `now` and `printSum` are not pure since each of them fails to satisfy one of the conditions.

Of course, the function will be just as impure if it violates both conditions, e.g:

```
function printNow(){
   let now = new Date();
   console.log(now);
   return now;
}

printNow();
//=> Wed Jun 14 2017 19:49:53 GMT+0200 (CEST)
//=> Wed Jun 14 2017 19:49:53 GMT+0200 (CEST)
```



printNow function

Another way to explain function purity is with some help of an equivalent concept - referential transparency. A function is said to be **referentially transparent** if any of its invocations can be replaced with the result value without any change to the program.

For example, we can call the `sum` function:

```
sum(3, 1);
```

Or we can just type in the result:

```
4
```

In the end, it's a four either way. The two expressions are equivalent.

We can do the same thing with our other pure function - either call it:

```
makeEspresso(2, false);
```

Or just type in the result:

```
{sugar: 2, coffee : 1, milk : 0, water : 1}
```

Again, the expressions are equivalent.

But, what if we tried the same thing with our impure functions? Let's try it out with `now` function:

```
now();
```

Can we replace it with a `Date` object literal, e.g. this one:

```
new Date('2017-06-14T21:06:05')
```

We can't or else the time in our application would appear frozen, like a broken clock.

Likewise, we can try the same thing with `printSum`:

```
printSum(3, 2);
```

Can we replace the function call with its result:

```
5
```

We sure can, but the outcome won't be the same. The literal number `5` does have the same value, but does not print out `Hey there, the result is: 5` on the console. If we carry out the replacement, we'll miss out any side effect that the impure function would have caused.

It is clear from the given examples that if a function is referentially transparent it is also pure.

However, we can generalize the idea of replacing function calls with its result even further. We can replace entire functions with combinations of inputs and their respective outputs.

For example, given enough memory our function `sum` could be entirely replaced by a **lookup table** like this:

a	b	sum
…	…	…
0	0	0
0	1	1
1	0	1
1	1	2
1	2	3
2	1	3
2	2	4
2	3	5
…	…	…

And then, instead of actually calculating the sum, we'd only need to look up the result in the table.

Obviously, it would be both memory intensive and impractical to do so in this particular example, as there are way too many combinations there. However, in the cases where the input parameter types are such that the number of possible values is reasonably small, it may make sense to use a lookup table.

One particularly interesting example is the case of pure functions without parameters. As mentioned already, these functions are logically equivalent to constants.

```
//e.g. this function
function numberOfSecondsInDay(){
   return 24*60*60;
}

//is equivalent to
function numberOfSecondsInDay(){
   return 86400;
}

//and the whole function can be replaced with a constant
let numberOfSecondsInDay = 86400;
```

Well-behaved imperative procedures

Now that we've mastered the concept of pure functions, we're going to deconstruct it and show how function purity is just an illusion, strictly speaking. However, one that does provide a very useful abstraction that helps us write cleaner and more manageable code.

Let's use a very simple pure function.

```
function add(a, b){
  return a + b;
}
```

As we’ve seen in the previous chapter, this JavaScript function is pure - its behavior depends only on its parameters and there are no side effects. Its only line of code returns the value of an operator expression `a + b` that also happens to be pure. Or does it?

In order to be executed on a computer, any piece of code must be either compiled to a machine-readable set of instructions or interpreted by another program that’s already machine-readable. So, if we zoom in through a couple of layers we’ll eventually get to see what the computer actually gets to run - something like these CPU instructions:

```
LOAD   12
ADD    14
STORE  17
```

This example could read as:

1. Load the value from a memory location 12 (e.g. parameter a) into CPU register.
2. Load the value from location 14 (or b) and add it to the value in the register.
3. Store the value from register into a memory location 17.

And that’s it. All the way down our purest and cleanest function actually works like a typical imperative program:

- It depends on a shared global state - the contents of your RAM at the time.
- It modifies the shared global state - the result is written back to the memory.

Note that on this level of abstraction there are no such things as parameters or return values. CPU just does stuff with data it gets from RAM and sends back any results.

All the way down all functions are just imperative procedures.

However, it would not be particularly constructive to just say: *Hey none of this stuff is actually pure, let’s just get back to good old imperative programming, it’s the same thing anyway.*

Well, that would be a bit trollish and also a tad incorrect. On one hand, the fact is that even pure functions are imperative procedures on the lowest level of abstraction. On the other hand, we really do have all the benefits we sought when we analyzed declarative programming.

For example, let’s consider this function:

```javascript
function arraySum(array){
   let result = 0;
   for(let i = 0; i < array.length; i++){
      result += array[i];
   }
   return result;
}
```

For the sake of simplicity let's assume that parameter `array` is always an array of numbers. We can see that the function code is clearly imperative - local variables `result` and `i` are being mutated and code flow is determined by a `for` loop.

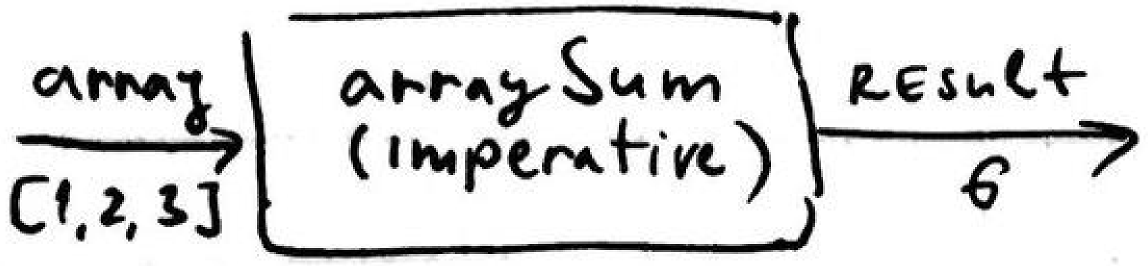
Yet, is this function pure?

```javascript
arraySum([1, 2, 3]);
//=> 6

//and again
arraySum([1, 2, 3]);
//=> 6
```

Try it out as many times as you want, it will always be a six. The outcome depends only on the input parameters and as we can see that there aren't any side effects. So, it is pure - even though it relies on plain old imperative code.

What we can learn from this is that what we call **pure functions** are in fact nothing other than exceptionally **well-behaved imperative procedures** that have the decency to take all its inputs in one clearly marked designated place and to return all its results in just as orderly manner. They have a clear boundary between what they do not mess with, but are also very protective about their own inner workings and will never be indiscreet enough to reveal their internal state and allow other programs to mess with their own variables.

Now notice that we can't really tell any of that just from looking at the function call. The imperative stuff is conveniently abstracted away and if we only observe the function as a black box we are blissfully oblivious of its inner workings.



Still, it is useful to just be aware of all this. Function purity is an abstract concept. It only exists in the world of ideas along with a point that has no dimensions and other mathematical abstractions. Programs run on actual computers, which are physical devices and as such they are subjected to physical constraints.

Hold onto that thought, we'll expand on it in the next chapter, where we explore some typical inherent impurities we can never really get away from.

Inherent impurities

No matter how hard we try, some impurities will always exist. We can abstract them away, ignore them or make sure that they won't affect normal program usage, but we can't ever get rid of them.

The two main sources of such impurities are:

- Hardware - because all programs run on actual devices.
- Time - because everything happens is trapped in the one-directional flow of time.

Hardware caused impurities are caused by physical constraints of the devices on which we run our programs. Some such devices are:

- CPU
- Memory
- Power supply or battery
- etc.

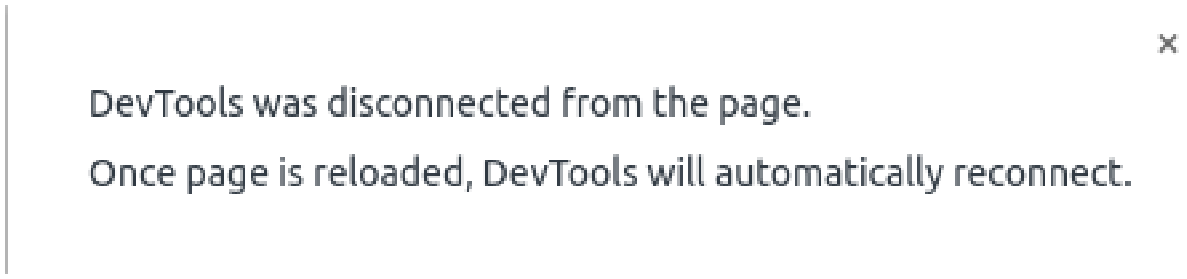
Let's illustrate this on another example - we'll write a function that receives a non-negative number and returns an array of all the numbers from 0 to that number, but not the number itself. If it receives a negative number, the function should just return an empty array.

```
function range(n){
   let array = [];
   for(let i = 0; i < n; i++){
      array.push(i);
   }
   return array;
}

range(-5);
//=> []

range(0);
//=> []

range(1);
//=> [0]

range(6);
//=> [0, 1, 2, 3, 4, 5]
```

So far, so good. But what if we give it a huge number?

```
range(1000000000);
```

In Google Chrome console instead of the result I get this error message:



DevTools was disconnected from the page.

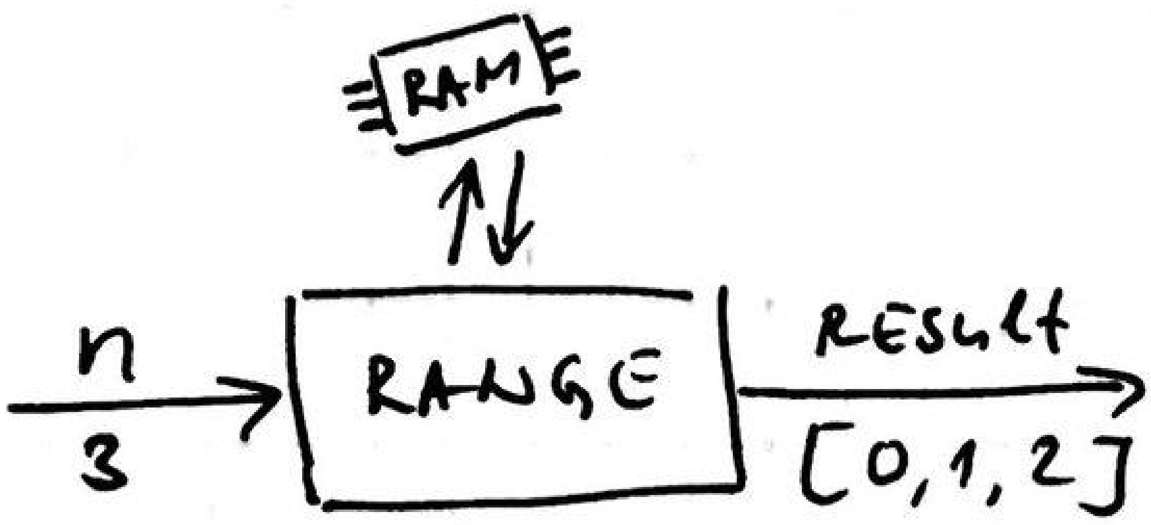
Once page is reloaded, DevTools will automatically reconnect.

Google Chrome Memory Error

When I try the same thing in a MongoDB shell (another handy JavaScript console), I get a somewhat different error message, that basically says the same thing - we ran out of memory!

```
tcmalloc: large alloc 1192198144 bytes == 0x7d768000 @
tcmalloc: large alloc 1490247680 bytes == 0x3dd2000 @
tcmalloc: large alloc 1862811648 bytes == 0x5cb08000 @
tcmalloc: large alloc 2328518656 bytes == 0x1338e4000 @
2017-06-15T15:26:02.484+0200 E QUERY    [thread1] Unknown Failure from JSInterpreter
```

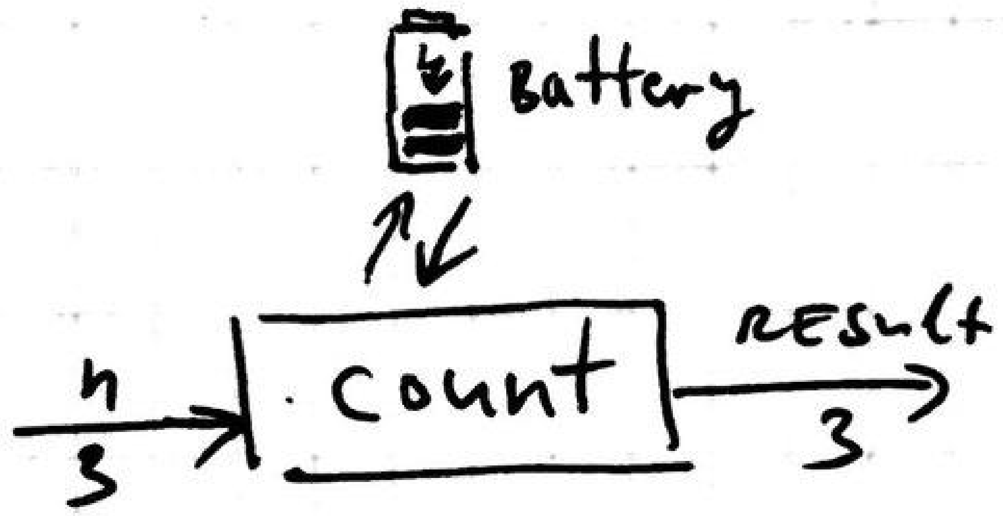
Obviously, things will be different on a different machine. That itself gives us a hint that there is an impurity. The function evaluation depends on something other than its parameters, in this case the size of the **available memory**.



Interestingly, this example also demonstrates side-effects. Whenever one process reserves a piece of memory, the same memory cannot be used for other programs at the same time. So, by being memory intensive, our function may actually prevent other programs from running properly or at all. Most of the time operating systems are pretty good at policing the resource allocation, but this still remains a possibility.

On the off chance that you have more memory available in your environment and the example actually runs fine for you, just add a few zeroes and try again. If it still works, keep adding those zeroes and you'll run into the limit eventually. You have to, it's not infinite.

Another obvious hardware dependency is **power supply**. A computer cannot work without electricity. If someone pulls the plug or the battery runs out during a function evaluation, it will be interrupted. This is also an example of a side-effect as functions with a higher strain on resources will drain the battery faster.



Let's see another example with side effects. We'll write a function that counts to `n` and then returns the count.

```
function count(n){
   let sum = 0;
   for(let i = 0; i < n; i++){
      sum += 1;
   }
   return sum;
}
```

Yes, I could have just said `return n`, but I needed an example that would intentionally put a strain on the CPU. Again, let's try it out:

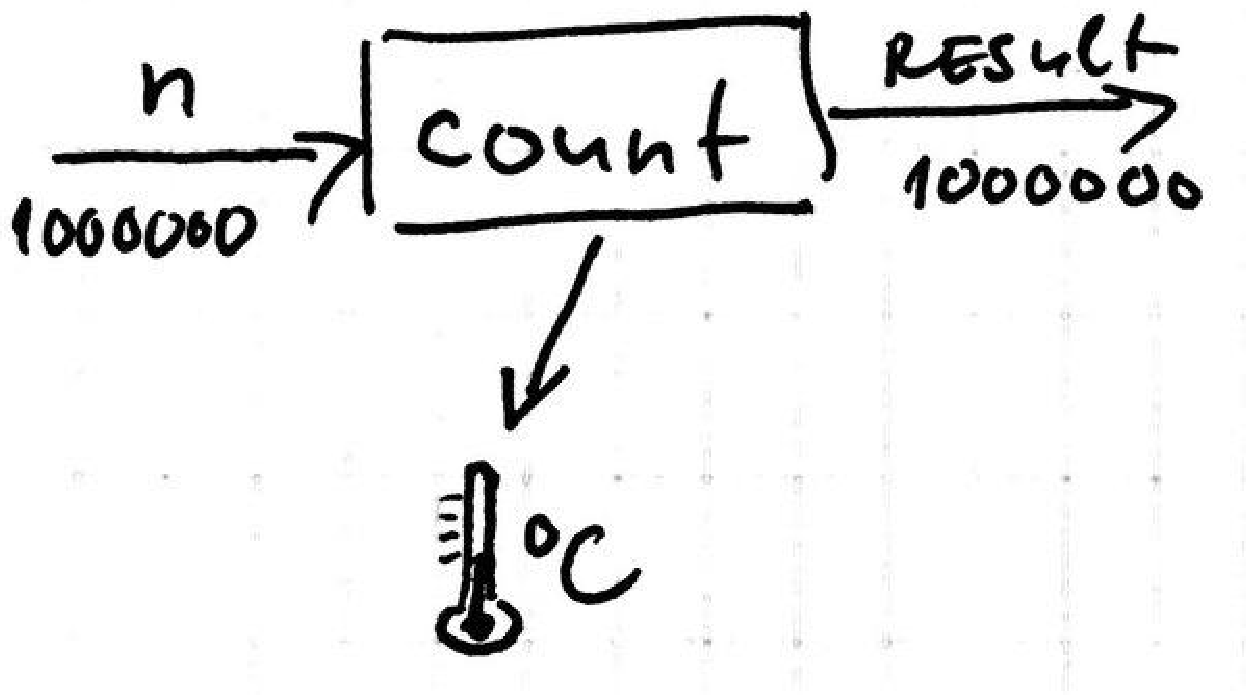
```
count(0);
//=> 0

count(4);
//=> 4

count(10);
//=> 10
```

So far, so good. It's a pure function, supposedly doesn't have side effects and unlike the previous one it doesn't really need much memory - there are just three variables involved: `n`, `i` and `sum`.

Then what would happen if we were to call it with a very large number?

```
count(1000000000000);
//=> 1000000000000
```

It actually did get the result, but when I ran it on my laptop it did something else, too. The CPU temperature jumped for a moment and the fan speed increased. That, in turn, slightly increased the temperature of my room.



If your code warms up the room in which its computer is, it is safe to claim that it produces a side effect.

Time based impurities are caused by basic laws of nature. Things simply take time to happen and nothing less than time-travel can change that.

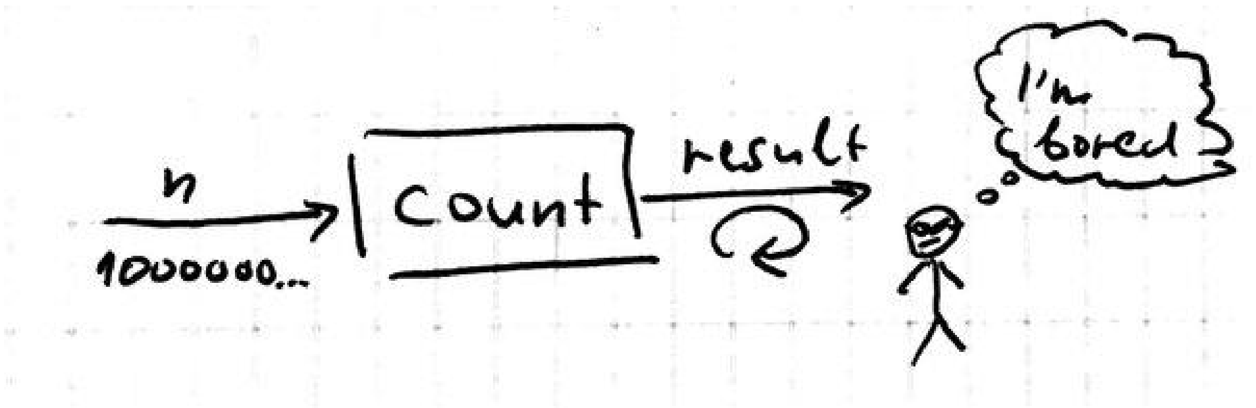
Let’s try running `count` with an even larger parameter. Be warned, you may have to kill the JavaScript console after the next example.

```
count(10000000000000000000);

//So, where is the result?
//still waiting...

//after some time...
//still waiting...

//a minute goes by...
//yep, still waiting...

//in the mean time, you remember that lame knock-knock Java joke

//but, we're still waiting...
```

While you could actually wait for the result, I recommend killing the console. It's not like we do not know the result and the point was already made - the passage of time while the function is being evaluated is a side effect in its own right.

How did this side effect manifest? Well, we can count at least two ways:

- We got bored while we waited for it to finish.
- We stopped the function from further evaluation. It is a user action, but one that has been directly caused by the long evaluation time.



So, these are the necessary evils we have to live with. Functional programming allows us to write clean and elegant code, but it is by no means magic. Fundamentally, we still have to conform with the same physical constraints as if we were doing imperative programming.

Intentional impurities

Ok, we get it - pure functions are good, side effects and mutable state are not. If we all only ever write pure code we'll easily develop top quality code and live long and prosper.

Well, actually… there's a catch. We actually need both side effects and the global state. You know, like, to get stuff done!

This is just a brief list of some impure things a typical software system needs to do:

- Get the user's inputs from the UI.
- Show the outputs on the UI.
- Store some data in the database, retrieve it and maybe update later.
- Tell the current time and date.
- Do something random, e.g. generate a session token.
- Send an email, notification, SMS or another kind of message.
- Talk to a different system, e.g. post a tweet via Twitter API.

If we stick to pure functions alone, we'll never do any of this. So, while purity is good, we actually need impure functions as well.

In the last chapter, we will explore some ways we can reconcile the parts of our code that involve only pure functions with the parts that contain inevitable side effects.