The Art of Functional Programming

with examples in OCaml, Haskell, and Java

Minh Quang Tran, PhD

The Art of Functional Programming

Copyright © 2024 Minh Quang Tran
All rights reserved.

About the author:

Minh Quang Tran has over 20 years of experience studying software development and working in the software industry. He has worked in various tech startups and large software companies in Europe. He holds a Bachelor of Computer Science from Furtwangen University, Germany, an M.Sc. in Computer Science from McMaster University, Canada, and a Ph.D. in Computer Science from the Technical University of Berlin, Germany. His interest lies in understanding and mastering the fundamentals and principles that transcend programming languages, frameworks, and tools.

2 Expressions – Building Blocks of Functional Programs

2.1 Everything is an Expression

Functional programming is all about evaluating expressions to values. For someone new to functional programming, this concept can be hard to grasp at first. A great way to understand it is by contrasting it with imperative programming.

In his Turing Award lecture *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, John Backus shows that an imperative programming language – such as C and Java – splits language elements into two worlds, expressions and statements. Expressions are those constructs that evaluate to values. In C or Java, we can define arithmetic expressions like `1 + 2`, boolean expressions like `true || false`, and string expressions like `"Hello"`. They all evaluate to values. Statements, on the other hand, are commands that perform something like variable assignment `s = s + 1`, `if` statements for branching, and `for`/`while` loops for executing statements repeatedly. Statements are characterized by their side effects.

Functional programming paradigm does not have any statements, no variable assignments, no `if` statements, and no `for`/`while` loops. Therefore, instead of variable assignments, we pass values around via function arguments and return values. Conditionals like `if` are expressions rather than statements. To formulate repeated computations, functional programming relies on recursive func-

tions instead of loops. Everything is an expression in the functional paradigm.

The following diagram compares the world-view difference between imperative and functional programming languages.

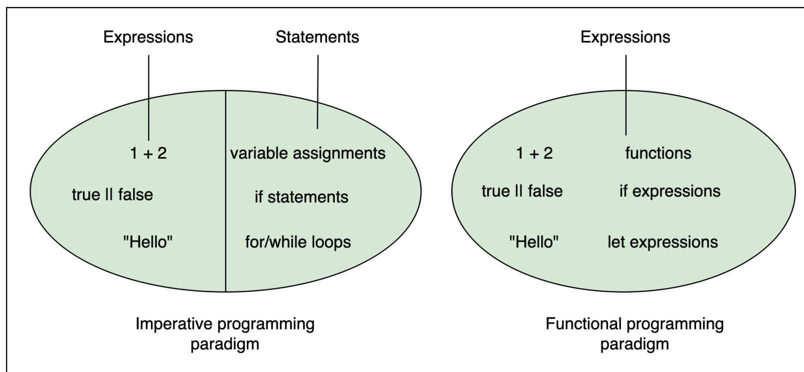

Figure 7: Language elements in imperative and functional programming

No wonder functional programming can feel odd for the uninitiated. It requires us to unlearn statements entirely. Since everything is an expression in the functional paradigm, a central part of learning a functional programming language is learning how to construct expressions and combine them to build larger ones. We'll discuss arithmetic, boolean, and string expressions in OCaml. Next, we'll discuss `if` expressions, functional programming's alternative to `if` statements in imperative programming languages.

2.1.1 Arithmetic expressions

OCaml provides built-in literal values for integer numbers, such as `1` and `2`. Those literal values are often termed primitive expressions because they represent the simplest possible expressions.

OCaml provides built-in operators: `+`, `-`, `*`, `/` and `mod`. These are used to add, subtract, multiply, divide, and calculate the remainder of two integers. We can build compound expressions with the operators.

```
1  (* OCaml *)
2  1 + 2
```

Here, `1` and `2` are called operands. The `+` operator is called a binary operator because it accepts two operands. Compound expressions, in turn, can be used as operands, allowing us to construct arbitrarily complex arithmetic expressions. For example,

```
1  (* OCaml *)
2  (1 + 2) * (3 - 4 * 5)
```

OCaml provides dedicated operators for float numbers. To add, subtract, multiply, and divide float numbers, we use `+.`, `-.`, `*.`, and `/.` .

```
1  (* OCaml *)
2  3.14 *. 2.0 *. 2.0
```

Note that in OCaml, we can write `2.` instead of `2.0` That means the expression above can be rewritten as

```
1  (* OCaml *)
```

```
2 3.14 *. 2. *. 2.
```

2.1.2 Boolean expressions

OCaml also provides comparison operators, such as =, >, >=, <, and <= for comparing two expressions. For example:

```
1 (* OCaml *)
2 1 = 2
3
4 (1 + 2) * (3 - 4 * 5) > 6 + 7
```

Comparison operators construct boolean expressions that evaluate to `true` or `false`. We combine boolean expressions with each other using logical operators `not` (negation), && (AND), and || (OR). For example:

```
1 (* OCaml *)
2 true && (1 > 2)
3
4 ((1 + 2) * (3 - 4 * 5) = -51) || (1 > 2)
5
6 not (true && (1 > 2))
```

Of course, there is no limit to how deeply those boolean expressions can be nested. Also notice that while most operators we have seen are binary, `not` exemplifies what is called a unary operator because it accepts only one operand.

2.1.3 String expressions

Like most programming languages, string literals in OCaml such as `"Hello"` are put inside quotes. We can use the `^` operator to concatenate two strings in OCaml.

```
(* OCaml *)
"Hello " ^ "FP"
(* Result: "Hello FP"*)
```

2.1.4 If expressions

One crucial aspect of any powerful programming language is testing a condition and choosing alternative computations depending on the result. Imperative programming languages provide the `if` statement used to express "if something is true, do this, or else do that." The following Java code shows a typical use of the `if` statement to calculate the maximum of two numbers, `a` and `b`.

```
// Java
if (a > b) {
   max = a;
} else {
   max = b;
}
```

As you can see, the `max` variable is updated depending on the condition. But in the functional paradigm, `if` is an expression of the form `if e1 then e2 else e3`. The right way of thinking is that if `e1` evaluates to `true`, the result is the value of the expression `e2`. Else, the result is the value of the expression `e3`. For example:

```
(* OCaml *)
if 1 > 2 then 1 else 2
```

The `else` branch in OCaml's `if` expression is mandatory because the entire `if` is treated as an expression, meaning it has to evaluate to a value regardless of the condition's outcome.

Since the condition and the two branches, `then` and `else`, may contain an expression, an `if` expression can be deeply nested that contains arbitrarily complex expressions, including other `if` expressions. For example:

```
(* OCaml *)
if 1 = 2 then if (1 + 2) * (3 - 4 * 5) = -51
    then 100 / 6 else 5 - 1 else 42
```

The mental shift from `if` statements to `if` expressions is a first step towards understanding functional programming's worldview – everything is an expression.

2.1.5 Advantages of everything-is-expression world-view

You may now be wondering, "Wait a moment! The imperative programming paradigm includes both expressions and statements, whereas the functional programming paradigm only incorporates expressions, devoid of any statements. Does this mean that the functional programming paradigm is less expressive than its imperative counterpart? Moreover, what could be the advantage of omitting such language elements??"

As it turns out, removing statements does not necessarily decrease

the expressiveness of a programming language. This is underpinned by the Church-Turing thesis, which states that lambda calculus – the foundation of functional programming – is as expressive as a Turing machine, the basis of imperative programming. We'll discuss lambda calculus in the next chapter.

As for the second question, although it might sound counterintuitive, there are enormous benefits associated with removing statements and treating everything as expressions. Such an approach allows functional programming languages to fully harness the power of combination. Before explaining why, however, allow me a short digression.

Do you play Lego? Many kids and adults enjoy spending hours building houses, robots, and human figures from Lego bricks. What makes Lego so compelling is that we can combine the same set of blocks in endless ways to build virtually anything, limited only by our imagination.

The secret of Lego's incredible flexibility lies in its adherence to what Harold Abelson and Gerald Jay Sussman, along with Julie Sussman, call the **closure property** in their seminal book *Structure and Interpretation of Computer Programs (SICP)*. They describe the closure property as follows: "An operation for combining data objects satisfies the closure property if the results of combining things with that operation can themselves be combined using the same operation."

In the context of Lego, we can consider each Lego brick as a "data object" and the act of connecting one Lego brick to another as the

"operation." When two bricks are connected, they form a new object - a composite of the two bricks. Intriguingly, this new object can be treated just like any other single LEGO brick. It can be connected to other bricks using the same operation, fully embodying the closure property. As we can see, closure property is the secret of the power of combination, as it allows us to build complex things from parts that are themselves made of parts.

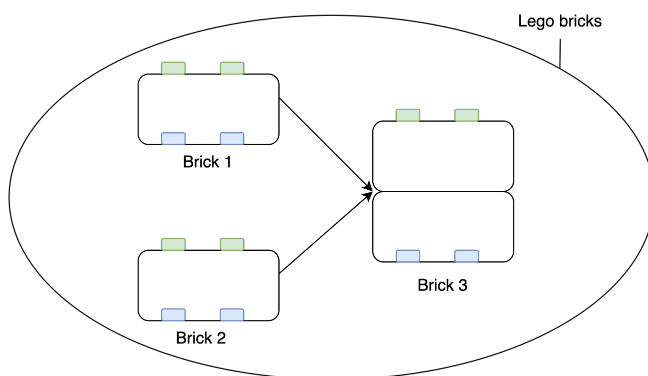

Figure 8: Lego satisfies closure property as connecting bricks results in a new brick

Why does that have anything to do with the art of functional programming? Because the same magic applies to functional programming! By treating everything as expressions, the functional programming paradigm inherently satisfies the closure property. Given two expressions, we can combine them into a new expression, which can, in turn, be further combined with other expressions, and so on.

However, the closure property is not entirely satisfied within the imperative programming paradigm due to the two divided worlds of expressions and statements discussed at the beginning of this section. Statements and expressions live in two different sets and often do not compose well with each other.

Let's look at a concrete example of this. As we saw in the previous section, functional programming languages treat conditionals like `if 1 > 2 then 0 else 42` as expressions. Because of this, we can combine it with other expressions easily. For example, `if 1 > 2 then 0 else 42` can be used as an operand of the operator `+`.

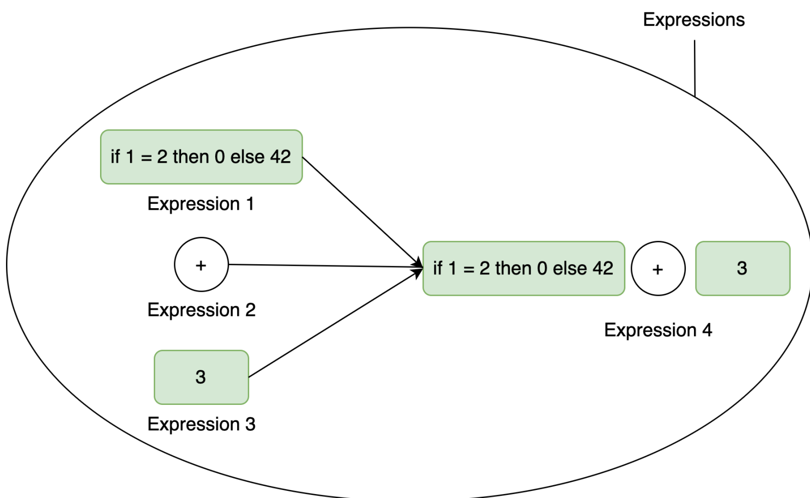

Figure 9: Combining if expression with another expression via '+' operator

However, we cannot do this in an imperative programming lan-

guage where `if` is a statement rather than an expression.

Figure 10: if statement can not be composed with another expression via '+' operator

This example is admittedly tiny, but it shows that functional programming excels at composition by treating everything as expression.

2.2 Syntax of Expressions

At its core, a program written in any textual programming language, such as OCaml, Java and Python is nothing more than a long string consisting of characters we enter with a keyboard. A string that makes up a valid program in one programming language might be considered invalid in another and vice versa. To illustrate, consider the following obscure string:

```
++++++++++[>+++++++>++++++++++>+++<<<-]>++.>+.
+++++++..+++.>++.<<+++++++++++++++.>.+++.------.
--------.>+.
```