March 23, 2023

Reading time: 6 minutes (1219 words)

Functional Friday - Episode 2

Learning Rescript - Expressions

Welcome back to Functional Friday, the place where functions stopped calling each other, because they had constant arguments 🥁! In the last episode we had a look on how **let bindings** and **types** work in ReScript and how they compare to JavaScript/TypeScript.
In this episode will focus mostly on expressions and we're going to use them to start introducing some differences between functional programming languages and imperative ones.

Expressions, uh? 🤔

Normally speaking, an expression is *the action of making known someone's thoughts or feelings*, whereas in programming an expression is an entity that may require further computation in order to determine its value.
Also, **in functional programming expressions are considered the basic building block** for our programs, while imperative languages relies more on *statements* (or *commands*).

It's important to fully understand this concept because, as we'll see, ReScript syntax is built around this.

Let Expressions

Previously, we saw how to use a `let` binding, and we noticed how similar was to declaring and assigning a variable in an imperative language.

```
RES
let areYouHappy = true
```

However, up to now we just talked about `let` definition. There's another usage of `let` which is an expression

```
RES
let nextYear = currentYear + 1
```

At a first look, it may seems there're no huge difference, but there's an important thing to notice: `nextYear` value is not immediately known and it depends on `currentYear` value.
Being used to read left to right here can make things counter-intuitive, so let's try to read right to left instead: *"considering the value* `1`*, add the value of* `currentYear` *and bind it to* `nextYear`*"*. This expresses (pun not intended) quite well what ReScript is going to do in order to determine `nextYear` value.

It is probably worth mentioning that ReScript will not allow us to write expressions that rely on coercion to be evaluated; things like

```
TS
const accountBalance = 2 + "1.00"
```

will result in a type error since they violate inference rules [1].

As we're going to see in incoming episodes, `let` expressions are a really powerful tool. Just to give you an idea, named functions themselves are `let` expressions, i.e.

```
RES
let isApophisHittingUs = (currentYear) =>
  if currentYear == 2029 { "Oh dayum" }
  else { "Phew! Not today" }
```

I will not go into a deep explanation about it for two main reasons:

- if you are familiar with JavaScript/TypeScript, it will probably remind you about arrow functions and ternary operator, so you can probably have a good guessing about what it does;
- we will talk in-depth about functions in a dedicated episode;

Nonetheless, keep in mind this example since it will lead us straight to the next topic.

If Expressions

If we put enough attention reading through the last lines of code, we may have probably noticed this

```
RES
if currentYear == 2029 { "Oh dayum" }
else { "Phew! Not today" }
```

which reads as follows: the expression `if exp_1 { exp_2 } else { exp_3 }` evaluates to `exp_2` if `exp_1` evaluates to `true`, otherwise it evaluates to `exp_3`. In functional programming `exp_1` is generally referred as the `if` expression *guard*.

While in imperative languages `if-else` are statements, in ReScript (and in OCaml too, for the matter) they're expressions and as such they can be used and combined with other expressions, such the let expressions we just talked about:

```
RES
let drink = if isMorning { "Coffee" } else { "Tea" }
```

which, as said previously, is similar to writing a ternary operator in JavaScript/TypeScript

```
TS
const drink = isMorning ? "Coffee" : "Tea";
```

`If` expressions can be nested and we can use as well a ternary syntactic sugar, so writing

```
RES
// Nested if-else
let drink =
  if isMorning { "Coffee " }
  else {
    if isLunchTime { "Water" }
    else { "Tea" }
  }

// Ternary like
let drink = isMorning ? "Coffee" : "Tea"
```

works just fine as well[2]. Aside from aforementioned similarities, there are few important aspects and several differences compared to JavaScript/TypeScript to keep in mind. Let's check them out.

1. Guard type should be a bool

Coming from JavaScript/TypeScript, we may be tempted to write something like this:

```
RES
let drink = if 1 { "Coffee" } else { "Tea" }
```

but in ReScript this will result into an error

```bash
This has type: int
  Somewhere wanted: bool
```

This happens because ReScript strictly requires the guard type to be a `bool`, so no truthy/falsy values are allowed.

2. If and else branches should return expressions of the same type

In JavaScript/TypeScript, we may be used to write ternary like the following one

```ts
const price = isFormattedAsString ? "0.00" : 0.0;
```

but in ReScript, this will result into an error

```bash
This has type: int
  Somewhere wanted: string

  You can convert int to string with Belt.Int.toString.
```

in such circumstances the compiler will also provide us a way to fix our expression, which is really helpful.

The reason of this rule lies in the fact that we need to give this expression an overall type, but because we're statically type-checking, we don't know which branch will be executed at runtime. Therefore both branches must return the same type `t`, which is the type of the expression [3]. Also notice how the type expectation comes from the `if` branch, and it's because ReScript evaluates the `if` branch first.

3. Braces and implicit returns

As you may have noticed, we're expected to always wrap `if-else` expressions with braces (with the ternary syntactic sugar being an exception). This is called block scoping and the value in the last line in each scope is always implicitly returned. Block scoping can be used in several ways, including let expressions

```res
let fullName = {
  let firstName = "Thomas"
  let lastName = "Anderson"
```

```
    `${firstName} A. ${lastName}` // this line is implicitly returned
  }
```

4. Else branch is mandatory

As obvious as it may sound, we always need to define an `else` branch, otherwise we're going to face an interesting type error message. The following code

```
RES
let dinner = if amIHungry { "Pizza" }
```

will in fact cause this type error

```
BASH
This has type: string
  Somewhere wanted: unit
```

the `unit` type is a special type that has a single value `()` and it compiles to `undefined`. While there are some specific cases where it may be useful or even beneficial to intentionally omit the `else` branch, for now we'll keep it simple and just consider it mandatory.

Let's wrap it up! 🫡

In this episode we saw what expressions are, how to use both `let` and `if` expressions, how they compare to JavaScript/TypeScript and how the type system interacts, being way more strict than what we're used to. We now have some very basics building blocks and I encourage you to jump into the playground and experiment yourself. Again, these arguments will recur in the future so keeping this article in mind may be useful.

As always, thank you for reading through the whole article: I really hope you enjoyed the content and you'll stick around for the next episode. Up to then, happy coding and... see you next time!
Cheers! 🤓

REFERENCES

[1] Aside from coercion, we should also keep in mind that (+) operator is not polymorphic. If you want further information about it, feel free to check ReScript documentation.
[2] As mentioned in the official documentation it's preferable to use if-else blocks instead of ternary whenever is possible.
[3] In the incoming episodes, we'll see how we can eventually work around this using variants, even if there're way more efficient way in ReScript to handle these cases.

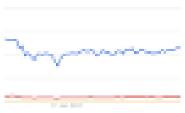
Functional Programming · Javascript · ReScript · Software Development · TypeScript

To leave a comment, click the button below to sign in with Google.

SIGN IN WITH GOOGLE

Popular posts from this blog

March 16, 2023



Functional Friday - Episode 1 Discovering ReScript - Types and Let Binding Welcome to this first episode of Functional Friday! This series is meant to be a journey of ReScript discovery and what it looks like for a JavaScript/TypeScript developer. If y …

READ MORE

March 08, 2023

About Me - Maurizio Vacca Curious by nature, driven by passion. I'm a software developer, whatever it means. Hey, thank you! My name is Maurizio, nice to meet you. I do really appreciate you are taking the time to read through this - hopefully - brief presentation of mine. Last time I had a blog it wa …

READ MORE

Powered by Blogger

Report Abuse

Wanna follow me?

@maurizio_vacca

Maurizio Vacca

Archive