ToiletPaper #159

jambit
WHERE INNOVATION WORKS

Something like Scala's for comprehension in Typescript

Author: André Petermann / Senior Software Architect / Office Leipzig

✕ Problem

Programming with monads such as `Option`, `Result`, `Either` or `Try` is a suitable way to write programs that clearly separate domain logic from handling of absence or error. However, the application of this approach to real-world programs often leads to large closures including deeply nested code blocks. Haskell's *do notation* or Scala's *for comprehension* are a nice way to solve this problem. For example, Scala's for comprehension looks like this:

```
val result: Either[String, Int] =
  for {
    dividend <- Right(42)
    divisor <- Right(2)
    divisorVerified <- if (divisor != 0) Right(divisor) else Left("Divisor must not be zero!")
  } yield dividend / divisor

println(result match {
  case Right(value) => value
  case Left(error) => error
})
```

However, there is no such thing in Typescript.

✓ Solution

The library for-comprehension-ts provides a similar notation to support *for comprehension* in Typescript:

```
const result: Result<number, string> =
    For._("dividend", success(42))
       ._("divisor", success(2))
       ._("divisorVerified", ({divisor}) => divisor != 0 ? success(divisor) : failure("Divisor must not be
       zero!"))
       .yield(({dividend, divisorVerified}) => dividend / divisorVerified)
console.log(isSuccess(result) ? result.value : result.error)
```

The library also includes an async version that allows for a seamless integration of regular and async functions using the same monads. This works without explicit awaiting of Promise inside the functions.
for-comprehension-ts can be used with all implementations of a `Monad` interface with the operations `map`, `flatMap` and `flatMapAsync`. The library already includes implementations of the following monads:

	Option<T>	**Result<T, E>**	**Try<T>**
abstraction	presence or absence	success or explicitly typed failure	success or implicitly caught exception
constructors	`some(value: T)` `none()`	`success(value: T)` `failure(error: E)`	`ok(value: T)` `error(error: any)`

In contrast to pipes this syntax allows programs to be directed acyclic graphs (DAG) whose vertices are named values (e.g., a = 3) and where edges are functions. The graph will be executed lazily, i.e., before yield is called, there is only a definition of a program. On calling yield, execution will be triggered. Thus, for-comprehension-ts programs can be duplicated, branched and repeated. Operations will only be executed until the first failure, error or absent value occurs. However, this behavioral aspect depends on the used monad.

+ Contribution

for-comprehension-ts is developed by jambit but not used in production yet. So, please feel free to contribute either directly by adding new features or indirectly by just using it:

- Source code: https://github.com/p3et/for-comprehension-ts
- For comprehension in Scala: https://www.baeldung.com/scala/for-comprehension