Если вы видите что-то необычное, просто сообщите мне.

Purescript Part 1: Basics of Purescript

Our Haskell Web Series covers a lot of cool libraries you can use when making a web app. But frontend web development can be quite a different story! There are a number libraries and frameworks out there. Yesod and Snap come to mind. Another option is Reflex FRP, which uses GHCJS under the hood.

But in this series we'll take different approach by exploring the Purescript language. Purescript is a bit of a meld between Haskell and Javascript. Its syntax is like Haskell's, and it incorporates many elements of functional purity. But it compiles to Javascript and thus has some features that seem more at home in that language.

In this part, we'll start out by exploring the basics of Purescript. If you're already familiar with those, you can move right onto part 2 of the series! There, we'll see some of the main similarities and differences between it and Haskell. We'll culminate this series by making a web front-end with Purescript and routing between different pages.

Purescript is the tip of the iceberg when it comes to using functional languages in production! Check out our Production Checklist for some awesome Haskell libraries!

GETTING STARTED

Since Purescript is its own language, we'll need some new tools. You can follow the instructions on the Purescript website, but here are the main points.

1. Install Node.js and NPM, the Node.js package manager
2. Run npm install -g purescript

3. Run npm install -g pulp bower
4. Create your project directory and run pulp init.
5. You can then build and test code with pulp build and pulp test.
6. You can also use PSCI as a console, similar to GHCI. First, we need NPM. Purescript is its own language, but we want to compile it to Javascript we can use in the browser, so we need Node.js. Then we'll globally install the Purescript libraries. We'll also install pulp and bower. Pulp will be our build tool like Cabal.

Bower is a package repository like Hackage. To get extra libraries into our program, you would use the bower command. For instance, we need purescript-integers for our solution later in the article. To get this, run the command:

```
bower install --save purescript-integers
```

A SIMPLE EXAMPLE

Once you're set up, it's time to start dabbling with the language. While Purescript compiles to Javascript, the language itself actually looks a lot more like Haskell! We'll examine this by comparison. Suppose we want to find the all pythagorean triples whose sum is less than 100. Here's how we can write this solution in Haskell:

```
sourceList :: [Int]
sourceList = [1..100]

allTriples :: [(Int, Int, Int)]
allTriples =
  [(a, b, c) | a <- sourceList, b <- sourceList, c <- sourceList]

isPythagorean :: (Int, Int, Int) -> Bool
isPythagorean (a, b, c) = a ^ 2 + b ^ 2 == c ^ 2

isSmallEnough :: (Int, Int, Int) -> Bool
isSmallEnough (a, b, c) = a + b + c < 100

finalAnswer :: [(Int, Int, Int)]
finalAnswer = filter
```

```
  (\t -> isPythagorean t && isSmallEnough t)
    allTriples
```

Let's make a module in Purescript that will allow us to solve this same problem. We'll start by writing a module Pythagoras.purs. Here's the code we would write to match up with the Haskell above. We'll examine the specifics piece-by-piece below.

```
module Pythagoras where

import Data.List (List, range, filter)
import Data.Int (pow)
import Prelude

sourceList :: List Int
sourceList = range 1 100

data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  }

allTriples :: List Triple
allTriples = do
  a <- sourceList
  b <- sourceList
  c <- sourceList
  pure $ Triple {a: a, b: b, c: c}

isPythagorean :: Triple -> Boolean
isPythagorean (Triple triple) =
  (pow triple.a 2) + (pow triple.b 2) == (pow triple.c 2)

isSmallEnough :: Triple -> Boolean
isSmallEnough (Triple triple) =
  (triple.a) + (triple.b) + (triple.c) < 100

finalAnswer :: List Triple
finalAnswer = filter
```

```
  (\triple -> isPythagorean triple && isSmallEnough triple)
  allTriples
```

For the most part, things are very similar! We still have expressions. These expressions have type signatures. We use a lot of similar elements like lists and filters. On the whole, Purescript looks a lot more like Haskell than Javascript. But there are some key differences. Let's explore those, starting with the higher level concepts.

DIFFERENCES

One difference you can't see in code syntax is that Purescript is NOT lazily evaluated. Javascript is an eager language by nature. So it is much easier to compile to JS by starting with an eager language in the first place.

But now let's consider some of the differences we can see from the code. For starters, we have to import more things. Purescript does not import a Prelude by default. You must always explicitly bring it in. We also need imports for basic list functionality.

And speaking of lists, Purescript lacks a lot of the syntactic sugar Haskell has. For instance, we need to use List Int rather than [Int]. We can't use .. to create a range, but instead resort to the range function.

We also cannot use list comprehensions. Instead, to generate our original list of triples, we use the list monad. As with lists, we have to use the term Unit instead of ():

```
-- Comparable to main :: IO ()
main :: Effect Unit
main = do
  log "Hello World!"
```

In the next part, we'll discuss the distinction between Effect in Purescript and monadic constructs like IO in Haskell.

One annoyance is that polymorphic type signatures are more complicated. Whereas in Haskell, we have no issue creating a type signature [a] -> Int, this will fail in Purescript. Instead, we must always use the forall keyword:

```
myListFunction :: forall a. List a -> Int
```

Another thing that doesn't come up in this example is the Number type. We can use Int in Purescript as in Haskell. But aside from that the only important numeric type is Number. This type can also represent floating point values. Both of these get translated into the number type in Javascript.

PURESCRIPT DATA TYPES

But now let's get into one of the more glaring differences between our examples. In Purescript, we need to make a separate Triple type, rather than using a simple 3-tuple. Let's look at the reasons for this by considering data types in general.

If we want, we can make Purescript data types in the same way we would in Haskell. So we could make a data type to represent a Pythagorean triple:

```
data Triple = Triple a b c
```

This works fine in Purescript. But, it forces us to use pattern matching every time we want to pull an individual value out of this element. We can fix this in Haskell by using record syntax to give ourselves accessor functions:

```
data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  }
```

This syntax still works in Purescript, but it means something different. In Purescript a record is its own type, like a generic Javascript object. For instance, we could do this as a type synonym and not a full data type:

```
type Triple = { a :: Int, b :: Int, c :: Int}


oneTriple :: Triple
oneTriple = { a: 5, b: 12, c: 13}
```

Then, instead of using the field names like functions, we use "dot-syntax" like in Javascript. Here's what that looks like with our type synonym definition:

```
type Triple = { a :: Int, b :: Int, c :: Int}

oneTriple :: Triple
oneTriple = { a: 5, b: 12, c: 13}

sumAB :: Triple -> Int
sumAB triple = triple.a + triple.b
```

Here's where it gets confusing though. If we use a full data type with record syntax, Purescript no longer treats this as an item with 3 fields. Instead, we would have a data type that has one field, and that field is a record. So we would need to unwrap the record using pattern matching before using the accessor functions.

```
data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  }

oneTriple :: Triple
oneTriple = Triple { a: 5, b: 12, c: 13}

sumAB :: Triple -> Int
sumAB (Triple triple) = triple.a + triple.b

-- This is wrong!
sumAB :: Triple -> Int
sumAB triple = triple.a + triple.b
```

That's a pretty major gotcha. The compiler error you get from making this mistake is a bit confusing, so be careful!

PYTHAGORAS IN PURESCRIPT

With this understanding, the Purescript code above should make some more sense. But we'll go through it one more time and point out the little details.

To start out, let's make our source list. We don't have the range syntactic sugar, but we can still use the range function:

```
import Data.List (List, range, filter)

data Triple = Triple
  { a :: Int
  , b :: Int
  , c :: Int
  }

sourceList :: List Int
sourceList = range 1 100
```

We don't have list comprehensions. But we can instead use do-syntax with lists instead to get the same effect. Note that to use do-syntax in Purescript we have to import Prelude. In particular, we need the bind function for that to work. So let's generate all the possible triples now.

```
import Prelude

...

allTriples :: List Triple
allTriples = do
  a <- sourceList
  b <- sourceList
  c <- sourceList
  pure $ Triple {a: a, b: b, c: c}
```

Notice also we use pure instead of return. Now let's write our filtering functions. These will use the record pattern matching and accessing mentioned above.

```
isPythagorean :: Triple -> Boolean
isPythagorean (Triple triple) =
  (pow triple.a 2) + (pow triple.b 2) == (pow triple.c 2)

isSmallEnough :: Triple -> Boolean
isSmallEnough (Triple triple) =
  (triple.a) + (triple.b) + (triple.c) < 100
Finally, we can combine it all with filter in much the same way we did in Haskell:

finalAnswer :: List Triple
finalAnswer = filter
  (\triple -> isPythagorean triple && isSmallEnough triple)
  allTriples
```

And now our solution will work!

CONCLUSION

This conludes part 1 of our Purescript series. Syntactically, Purescript is a very near cousin of Haskell. But there are a few key differences we highlighted here about the nature of the language.

In part 2, we'll look at some other important differences in the type system. We'll see how Purescript handles type-classes and monads. After that, we'll see how we can use Purescript to build a web front-end with some of the security of a solid type system.

Download our Production Checklist for some more cool ideas of libraries you can use!

Revision #1
Created 11 March 2022 16:51:19 by gasick
Updated 11 March 2022 17:11:16 by gasick