Interactive code snippets not yet available for SoH 2.0, see our Status of of School of Haskell 2.0 blog post (https://www.fpcomplete.com/blog/2016/01/soh-status)

10. Error Handling

5 Mar 2015 Bartosz Milewski (https://www.schoolofhaskell.com/user/bartosz)

View Markdown source (https://www.schoolofhaskell.com/tutorial-raw/4/29607dda3950552237ecdc82079df1fa3c292a4b)

Sections

Previous content: 9. Evaluator (https://www.schoolofhaskell.com/user/school/starting-with-haskell/basics-of-haskell/9_Evaluator)

Next content: 11. State Monad (https://www.schoolofhaskell.com/user/school/starting-with-haskell/basics-of-haskell/12-State-Monad)

Go up to: Basics of Haskell (https://www.schoolofhaskell.com/school/starting-with-haskell/basics-of-haskell)

See all content by Bartosz Milewski (https://www.schoolofhaskell.com/user/bartosz)

Show me how you handle errors and I'll tell you what programmer you are. Error handling is fundamental to all programming. Language support for error handling varies from none whatsoever (C) to special language extensions (exceptions in C++, Java, etc.). Haskell is unique in its approach because it's expressive enough to let you build your own error handling frameworks. Haskell doesn't need built-in exception support: it implements it in libraries.

We've seen one way of dealing with errors: calling the `error` function that terminates the program. This works fine for runtime assertions, which alert us to bugs in the program. But many "errors" are actually expected. We've seen one such example: `Data.Map.lookup` fails when called with a key that's not present in the map. The possibility of failure is encoded in the `Maybe` return type of `lookup`. It's interesting to compare this with similar functions in other languages. In C++, for instance, `std::map` defines multiple accessor functions varying only in their failure behavior:

- `at` throws an exception
- `find` returns an empty iterator
- `operator[]` inserts a dummy value using a default constructor for it.

The last one is the most bizarre of the three. Since the array access operator must return a reference, even if the key is not found, it has to create a dummy value. The behavior of `at` is potentially dangerous if the client forgets to catch the exception. Of the three, `find` is the safest, since the return type suggests to the client iteration rather than straight dereference; and iteration normally starts with checking for termination.

In functional programming, failure is another way of saying that the computation is *partial*; that is, not defined for all values of arguments. In Haskell we always try to use *total* functions -- functions defined for all values of their arguments. If the domain of a computation is known at compile time, we can often define a restricted data type to be used for its arguments; for instance, an enumeration instead of an integer. This is not always possible or feasible, so the other option is to turn a partial function into a total function by changing its return type. C++ method `find` does this trick by returning an iterator (it *always* returns an iterator for *any* value of its argument); Haskell `lookup` does it by returning a `Maybe`.

This trick of returning a different type in order to turn a non-functional computation into a pure function is used extensively in Haskell and finds its full expression in monads.

"Computation" or "notion of computation" often describes what we want to do in terms that may or may not have immediate pure function implementation: that is values in, values out. Transforming computations into functions is what functional programming is all about.

Either May Be Better than Maybe

The trick with `Maybe` is a bit limited. All we know about the failure is that it occurred. In practice, we'd like to know more. So the next step in error handling is to use the `Either` data structure defined in the Prelude:

```
data Either a b = Left a | Right b
```

`Either` is parameterized by two types, not one. A value of the `Either` type either contains a value of type `a` or of type `b`. We can discriminate between the two possibilities by pattern matching on either constructor. `Either` is mostly used as a generalization of `Maybe` in which `Left` not only encodes failure but is accompanied by an error message. `Right` encodes success and the accompanying value. So `a` is often fixed to be a `String`.

Here's how we can use `Either` to encode the failure of a lookup without terminating the whole program:

```
lookUp :: String -> SymTab -> Either String (Double, SymTab)
lookUp str symTab =
    case M.lookup str symTab of
      Just v  -> Right (v, symTab)
      Nothing -> Left ("Undefined variable " ++ str)
```

Now the burden is on the caller to pattern match the result of any such call and either continue with the successful result, or handle the failure. More often than not, the error is just passed unchanged to the caller, and so on; and somewhere at the top of the call tree, displayed to the user (remember, the very top is always an `IO` function).

It's a know phenomenon that error propagation spreads like a disease throughout the code. You could have seen that in C and sometimes even in C++, when exceptions are not an option. And indeed, you can create the same mess in Haskell. Here's a naive implementation of the evaluator of `SumNode`, which checks for errors that can happen in the evaluation of its children, and propagates them if necessary:

```
evaluate :: Tree -> SymTab -> Either String (Double, SymTab)

evaluate (SumNode op left right) symTab =
    case evaluate left symTab of
      Left msg -> Left msg
      Right (lft, symTab') ->
        case evaluate right symTab' of
          Left msg -> Left msg
          Right (rgt, symTab'') ->
            case op of
              Plus  -> Right (lft + rgt, symTab'')
              Minus -> Right (lft - rgt, symTab'')
```

Notice the creeping indentation. I'm showing you this code so that you know what awaits you if you refuse to learn about monads. However ugly this code is, it works, so the principle is right. We just need some monadic sugar to make it palatable.

Errors are propagated up, until they are finally dealt with in the main IO loop:

```
loop symTab = do
   str <- getLine
   if null str
   then
      return ()
   else
      let toks = tokenize str
          tree = parse toks
      in
          case evaluate tree symTab of
          Left msg -> do
              putStrLn $ "Error: " ++ msg
              loop symTab -- use old symTab
          Right (v, symTab') -> do
              print v
              loop symTab'
```

The nice thing is that, since we keep around the old copy of the symbol table, we can start the next iteration after a failure as if nothing happened. It's as if one transaction had been aborted and another started from the the same state.

```
import Data.Char
import qualified Data.Map as M

data Operator = Plus | Minus | Times | Div
    deriving (Show, Eq)

data Token = TokOp Operator
           | TokAssign
           | TokLParen
           | TokRParen
           | TokIdent String
           | TokNum Double
           | TokEnd
    deriving (Show, Eq)

operator :: Char -> Operator
operator c | c == '+' = Plus
           | c == '-' = Minus
           | c == '*' = Times
           | c == '/' = Div

tokenize :: String -> [Token]
```

Abstracting the Either Pattern

Looking at the code above, you can see a pattern arising. First of all, we have all these functions that return their results wrapped in the `Either` type. Every time we get an `Either` value, we pattern match it and fork the computation: When the result is `Right` we make the value in it available to the rest of the computation. If the result is `Left`, we skip the rest of the computation and propagate the error. We would like to capture this pattern. We'd like to isolate the boilerplate code, leaving "holes" for the client-provided variables. One such hole is to be filled by the initial `Either` value. The tricky part is the "skip the rest of the computation" part of the pattern. This can be done if the pattern has another hole for "the rest of the computation" so that it can either execute it or not.

Let's first identify this pattern in the evaluation of a unary node:

The first pattern filler is the value returned by `evaluate tree symTab`. The "rest of the calculation" filler is the code:

```
case op of
   Plus  -> Right ( x, symTab')
   Minus -> Right (-x, symTab')
```

We can abstract this code into a lambda function. Notice that this code uses `x` and `symTab'`, which were extracted from the first filler. So when we abstract it, the resulting lambda function should take this tuple as an argument:

```
\(x, symTab') -> case op of
   Plus  -> Right ( x, symTab')
   Minus -> Right (-x, symTab')
```

The parameter `op` is captured by the lambda from the outer environment -- remember, lambdas are *closures*: they can capture values from the environment in which they are defined.

Let's call this new pattern `bindE` and implement it as a function of two arguments. Here's how we would call it from the evaluator:

```
evaluate (UnaryNode op tree) symTab =
    bindE (evaluate tree symTab)
          (\(x, symTab') ->
              case op of
                Plus  -> Right ( x, symTab')
                Minus -> Right (-x, symTab'))
```

This is just a straight call to `bindE` with two arguments -- the second one being a multi-line lambda function.

The implementation of `bindE` is pretty straightforward. It just picks the common parts of the pattern and combines them in one function. It deals with the tedium of pattern matching the first argument and propagating the error. In case of success, it calls the continuation with the right arguments:

```
bindE :: Either String (Double, SymTab)
      -> ((Double, SymTab) -> Either String (Double, SymTab))
      -> Either String (Double, SymTab)
bindE ev k =
    case ev of
      Left msg -> Left msg
      Right (x, symTab') -> k (x, symTab')
```

Have a good look at the signature of `bindE`. Again, the second argument to `bindE` is the function that encapsulates the rest of the computation. It takes a pair `(Double, SymTab)` and returns another pair encapsulated in `Either`. This function argument is often called a *continuation*, because it continues the computation. The corresponding argument is often named `k` (think *kontinuation*; `c` would be too generic a name).

This pattern can be further abstracted by parameterizing it on the type of the contents of `Right`. In fact we need this if we want to include `addSymbol` in our scheme (see the unit in the return type):

```
addSymbol :: String -> Double -> SymTab -> Either String ((), SymTab)
```

If we want to bind the result of, say, `UnaryNode` evaluator to `addSymbol`, the input will be of type `Either String (Double, SymTab)` and the result of type `Either String ((), SymTab)`, so `bindE` really needs two type parameters, `a` and `b`:

```
bindE :: Either String a
      -> (a -> Either String b)
      -> Either String b
bindE ev k =
    case ev of
      Left msg -> Left msg
      Right v -> k v
```

Notice that as long as the client only uses `bindE` to deal with `Either` values, they don't have to deal with them directly (e.g., pattern match), except when they have to create them; and in the final unpacking, when they want to display the error message. The creation of `Either` values can also be abstracted by providing two more functions, `return` and `fail`:

```
return :: a -> Either String a
return x = Right x

fail :: String -> Either a
fail msg = Left msg
```

We can use these functions in the implementation of `lookUp`

```
lookUp :: String -> SymTab -> Either String (Double, SymTab)
lookUpb str symTab =
    case M.lookup str symTab of
      Just v  -> return (v, symTab)
      Nothing -> fail ("Undefined variable " ++ str)
```

We can also replace all `Right` constructors that are used to return values by calls to `return`. This way the client code can be written without any knowledge of the fact that values are encapsulated in the `Either` type.

Here's, for instance, an improved version of the `UnaryNode` evaluator, this time with no mention of `Either` or any of its constructors:

```
evaluate (UnaryNode op tree) symTab =
    bindE (evaluate tree symTab)
          (\(x, symTab') ->
              case op of
                Plus  -> return ( x, symTab')
                Minus -> return (-x, symTab'))
```

Maybe it doesn't seem like these transformations buy us much in code size or readability, but they are a step in the direction of hiding the details of error handling.

But we can do even better, once we realize that we have just defined a monad.

The Either Monad

A monad in Haskell is defined by a type constructor (a type parameterized by another type) and two functions, bind and return (optionally, fail). In our case, the type constructor is based on the `Either a b` type, with the first type variable fixed to `String` (yes, it's exactly like currying a type function). Let's formalize it by defining a new (paremeterized) type:

```
newtype Evaluator a = Ev (Either String a)
```

The `newtype` declaration is a compromise between `type` alias and a full-blown `data` declaration. It can be used to define data types that have only one data constructor that takes only one argument -- `Ev` in this case. (`newtype` is preferred over similar `data` declaration because of slightly better performance characteristics.)

The first change to our code will be to replace the type signature of `evaluate` from:

```
evaluate :: Tree -> SymTab -> Either String (Double, SymTab)
```

to:

```
evaluate :: Tree -> SymTab -> Evaluator (Double, SymTab)
```

We can now redefine `bindE`, `return`, and `fail` in terms of `Evaluator`. Then we have to tell Haskell that we are defining a monad. Why does Haskell need to know about our monad? Because with this knowledge it will allow us to use the `do` notation -- that's the sugar we've been craving.

`Monad` is a typeclass -- I'll talk more about typeclasses soon. For now, it's enough to know that, in order to tell Haskell that we are defining a monad, we need to *instantiate* `Monad` with our `Evaluator`. When instantiating a `Monad` we have to provide the appropriate definitions of bind, return and, optionally, fail. The only tricky one is bind, since `Monad` defines it as an infix operator, `>>=` (pronounced bind). Operators are just functions with funny names composed of special characters. An infix operator is a function that takes two arguments, one on the left and one on the right. Without further ado, here's the instance declaration for our first monad:

```
instance Monad Evaluator where
    (Ev ev) >>= k =
        case ev of
          Left msg -> Ev (Left msg)
          Right v -> k v
    return v = Ev (Right v)
    fail msg = Ev (Left msg)
```

These are exactly the definitions we've seen before, except for the additional layer of the `Ev` constructor. Notice that I used infix notation in defining operator `>>=`. Its left argument is the pattern `(Ev ev)`, and its right argument is the continuation `k`.

Let's see how this new monad works in the evaluator of `SumNode` -- first without the `do` notation:

```
evaluate (SumNode op left right) symTab =
    evaluate left symTab >>= \(lft, symTab') ->
        evaluate right symTab' >>= \(rgt, symTab'') ->
            case op of
              Plus  -> return (lft + rgt, symTab'')
              Minus -> return (lft - rgt, symTab'')
```

Notice that the second argument to the first `>>=` is a lambda that continues up to the end of the function. Inside its body there is another `>>=` with its own lambda. Notice also that the innermost lambda has access not only to `rgt` -- which is its argument -- but also to the external `lft` and `op`, which it captures from its environment.

Here's *the same code* in `do` notation:

```
evaluate (SumNode op left right) symTab = do
    (lft, symTab')  <- evaluate left symTab
    (rgt, symTab'') <- evaluate right symTab'
    case op of
      Plus  -> return (lft + rgt, symTab'')
      Minus -> return (lft - rgt, symTab'')
```

As you can see, the `do` block hides the binding `>>=` *between* the lines, it automatically converts "the rest of the code" to continuations, and it lets you treat the arguments to lambdas as if they were local variables to be assigned to. It's this syntactic sugar that makes `do` blocks look so convincingly imperative, even though they are purely functional in nature.

Compare this with our starting point:

```
evaluate (SumNode op left right) symTab =
    case evaluate left symTab of
      Left msg -> Left msg
      Right (lft, symTab') ->
        case evaluate  right symTab' of
          Left msg -> Left msg
          Right (rgt, symTab'') ->
            case op of
              Plus  -> Right (lft + rgt, symTab'')
              Minus -> Right (lft - rgt, symTab'')
```

That's definitely progress. If we could only get rid of this noise of threading `symTab` all over the place. Oh, wait, that's the next tutorial and another monad.

Let's bask in the monadic sunshine some more. Did you notice that we have essentially implemented exceptions? `fail` behaves very much like `throw`. It shortcuts the execution of the current function, propagates the error to its caller, who will in turn shortcut its execution and so on, until the "exception is caught." Catching the exception means unpacking the `Evaluator` returned by a function, and either retrieving the result or doing something with the error.

Can you forget to "catch" the exception? Not really -- it's part of the return type. You can't even access the value of your computation without unpacking it. And if your unpacking matches all possible `Either` patterns, as it should, you'll be forced to deal with the error case anyway.

Do you remember exception specifications in Java or C++ (currently obsoleted)? In Haskell there's no need for this ad hoc feature because "exception specification" is encoded in the return type of a function. You can't ignore types in Haskell, so you essentially leave it to the compiler to enforce exception safety.

There is a full-blown exception library `Control.Exception` (https://www.stackage.org/lts/hoogle?q=Control.Exception) in Haskell, complete with `throw`, `catch`, and a long list of predefined exception types. The Prelude also defines the `Monad` instance for `Either`, so we could have used it directly. But I thought "inventing" the monad from scratch would give you a better learning experience.

Type Classes

I said that `Monad` is a typeclass, but I haven't explained what a typeclass is. It's not exactly like a class in OO languages, but there are some similarities.

If you're familiar with Java or C++, you know that a class may define a set of virtual functions. When you have a polymorphic reference to one of the descendants of such a class, and call one of these virtual functions on it, the function that is called depends on the actual type of the object, which is determined *at runtime*. This kind of *late binding* is possible because a polymorphic object carries with it a vtable: an array of function pointers.

Similarly, in Haskell, a type class let's you define the signatures of a set of functions. You may think of a typeclass as an *interface*. (Although it is possible for a typeclass to provide default implementations for some of the functions -- the monadic `fail`, for instance, has such implementation: it calls `error` by default.) There's even an analog of a vtable that is (invisibly) passed to polymorphic functions.

The actual implementation of typeclass functions is provided by the client every time they declare some type to be an `instance` of a typeclass. In Java or C++ this would happen when the client defines concrete classes that implement an interface. In Haskell you define instances instead. The interesting thing is that, in Haskell, the client is able to connect any typeclass with any type (including built-in types) "after the fact." (In most OO languages you can't make an `int` a subclass of `IFoo` -- in Haskell you can.)

Let's work through an example:

```
class Valuable a where
    evaluate :: a -> Double

data Expr = Const Double | Add Expr Expr

instance Valuable Expr where
    evaluate (Const x) = x
    evaluate (Add lft rgt) = evaluate lft + evaluate rgt

instance Valuable Bool where
    evaluate True  = 1
    evaluate False = 0

test :: Valuable a => a -> IO ()
test v = print $ evaluate v

expr :: Expr
expr = Add (Const 2) (Add (Const 1.5) (Const 2.5))

main = do
    test expr
    test True
```

I have defined a typeclass `Valuable` with one function `evaluate`, which takes an instance of `Valuable`, `a`, and returns a `Double`. Separately I have defined a data type `Expr`, which has no knowledge of the class `Valuable`. Then I realized that `Expr` is `Valuable`, so I wrote an `instance` declaration that makes this connection and provides the "witness" -- the actual implementation of the function `evaluate`.

To make things even more exciting, I decided that the built-in type `Bool` is also `Valuable` and provided and `instance` declaration to prove it. Now I was able to write a polymorphic function `test` that calls `evaluate` on its argument. But unlike in previous examples of polymorphism, the generic argument to `test` is constrained: it has to be an instance of the class `Valuable`. The type expression before the double arrow `=>` defines class constraints for what follows. In this case we have one constraint, `Valuable a`, meaning `a` must be an instance of `Valuable`. In general there may be several such constraints listed between a set of parentheses and separated by commas.

Hadn't I provided the type signature for `test`, the compiler would have figured it out by analyzing the body of `test` and seeing that I called `evaluate` on its argument.

This is an example of a more general mechanism: If you define a function that adds its arguments, the compiler will deduce the `Num` constraint for them. If you compare arguments for (in-) equality, it will deduce `Eq` constraints. If you `print` them, it will deduce `Show`, etc.

The C++ Standards Committee almost voted in the *concepts* proposal in 2011, which would have added constrained polymorphism to C++ templates. But then they gave up because of the tremendous complexity imposed by the requirement of backwards compatibility.

Solution to the Expression Problem

In the previous tutorial I described the expression problem: How can you create a library that would be open to adding new data *and* new functions. We've seen that, in Haskell, extending a library by adding new functions was easy, but the addition of new varieties of data required modifications to the library. Clever use of typeclasses will let us have the cake and eat it too.

Let's work the magic on the previous example. First, instead of having one type `Expr` with many constructors, let's replace it with one typeclass `Expr` and many individual data types (by many I mean two in this case). We then glue these data types together by making them instances of `Expr`. `Expr` itself is empty -- its only purpose is to unify all expression types.

```
class Expr a

data Const   = Const Double
data Add a b = Add a b

instance Expr Const
instance (Expr a, Expr b) => Expr (Add a b)
```

Notice that I used the names `Const` and `Add` both to name data types and their constructors. These two namespaces are separate, so if you're running out of names, it's a common practice to reuse them this way. Also, notice the use of two class constraints in the instance definition for `Add`. Both `a` and `b` must be `Expr` for `Add a b` to be `Expr`.

Now we would like to define the function `evaluate` for both `Const` and `Add` nodes. But these are no longer just two constructors -- they are two different data types. The only way to define `evaluate` for both is to overload it. This is possible, but only if `evaluate` is part of a typeclass. The name `Valuable` for such a class seems natural. Let's make it work for `Const` and `Add`:

```
class (Expr e) => Valuable e where
    evaluate :: e -> Double

instance Valuable Const where
    evaluate (Const x) = x
instance (Valuable a, Valuable b) => Valuable (Add a b) where
    evaluate (Add lft rgt) = evaluate lft + evaluate rgt
```

This time we made sure that only expressions can be evaluated: That's the constraint `Expr e` in the definition of `Valuable`.

So that's our library. Let's now see how a client may extend it by, for instance, adding a new expression type: `Mul`.

```
data Mul a b = Mul a b

instance (Expr a, Expr b) => Expr (Mul a b)
instance (Valuable a, Valuable b) => Valuable (Mul a b) where
    evaluate (Mul lft rgt) = evaluate lft * evaluate rgt
```

Plugging `Mul` into the library required only making it the instance of `Expr` and `Valuable`. Perfect! Our library is now open to adding new data.

Let's check if it's still open to new functions. Let's add a pretty printing function to expressions, including the newly defined `Mul`. The trick is to make `pretty` a member of a new class `Pretty` and then make all the expression types its instances:

```
class (Expr e) => Pretty e where
    pretty :: e -> String

instance Pretty Const where
    pretty (Const x) = show x
instance (Pretty a, Pretty b) => Pretty (Add a b) where
    pretty (Add x y) = "(" ++ pretty x ++ " + " ++ pretty y ++ ")"
instance (Pretty a, Pretty b) => Pretty (Mul a b) where
    pretty (Mul x y) = pretty x ++ " * " ++ pretty y
```

Granted, there is a bit of syntactic noise here, but we have accomplished quite a feat: We have overcome the expression problem!

```
class Expr a

data Const   = Const Double
data Add a b = Add a b

instance Expr Const
instance (Expr a, Expr b) => Expr (Add a b)

class (Expr e) => Valuable e where
    evaluate :: e -> Double

instance Valuable Const where
    evaluate (Const x) = x
instance (Valuable a, Valuable b) => Valuable (Add a b) where
    evaluate (Add lft rgt) = evaluate lft + evaluate rgt

-- client code

data Mul a b = Mul a b

instance (Expr a, Expr b) => Expr (Mul a b)
instance (Valuable a, Valuable b) => Valuable (Mul a b) where
```

The Monad Typeclass

Let's go back to the `Monad` typeclass. This is how it's defined in the Prelude:

```
class Monad m where
    (>>=)   :: m a -> (a -> m b) -> m b
    (>>)    :: m a -> m b -> m b
    return  :: a -> m a
    fail    :: String -> m a

    mv >> k =  mv >>= \_ -> k
    fail s  = error s
```

There's something different about this typeclass: it's not a typeclass that unifies types; rather it unifies type *constructors*. The parameter `m` is a type constructor. It requires another type parameter to become a type. See how `m` always acts on either `a` or `b`, which are type variables. Going back to the `Evaluator` in our calculator, it was a type constructor too, with the type parameter `a`:

```
newtype Evaluator a = Ev (Either String a)
```

That's why we were able to make `Evaluator` an instance of `Monad`:

```
instance Monad Evaluator where
```

Specifically, notice that we used `Evaluator`, not `Evaluator a` in this definition. You always supply a type constructor to the instance definition of `Monad`.

`Monad` also defines operator `>>`, which ignores the value from its first argument (see its default implementation). You can bind two monadic functions using `>>` if you're only calling the first one for its "side effects." This will make more sense in the next tutorial, when we talk about the state monad. Operator `>>` is often used with the `IO` monad to sequence output functions, as in:

```
main :: IO ()
main = putStrLn "Hello " >> putStrLn "World"
```

which is the same as:

```
main :: IO ()
main = do
    putStrLn "Hello "
    putStrLn "World"
```

Exercises

Ex 1. Define the `WhyNot` monad:

```
data WhyNot a = Nah | Sure a
  deriving Show

instance Monad WhyNot where
   ...

safeRoot :: Double -> WhyNot Double
safeRoot x =
    if x >= 0 then
      return (sqrt x)
    else
      fail "Boo!"

test :: Double -> WhyNot Double
test x = do
   y <- safeRoot x
   z <- safeRoot (y - 4)
   w <- safeRoot z
   return w


main = do
```

```
data WhyNot a = Nah | Sure a
  deriving Show

instance Monad WhyNot where
   Sure x >>= k = k x
   Nah    >>= _ = Nah
   return x     = Sure x
   fail _       = Nah

safeRoot :: Double -> WhyNot Double
safeRoot x =
    if x >= 0 then
      return (sqrt x)
    else
      fail "Boo!"

test :: Double -> WhyNot Double
test x = do
   y <- safeRoot x
   z <- safeRoot (y - 4)
   w <- safeRoot z
   return w
```

Ex 2. Define a monad instance for `Trace` (no need to override `fail`). The idea is to create a trace of execution by sprinkling you code with calls to `put`. The result of executing this code should look something like this:

```
["fact 3","fact 2","fact 1","fact 0"]
6
```

Hint: List concatenation is done using `++` (we've seen it used for string concatenation, because `String` is just a list of `Char`).

```
newtype Trace a = Trace ([String], a)

instance Monad Trace where
    ...

put :: Show a => String -> a -> Trace ()
put msg v = Trace ([msg ++ " " ++ show v], ())

fact :: Integer -> Trace Integer
fact n = do
   put "fact" n
   if n == 0
       then return 1
       else do
           m <- fact (n - 1)
           return (n * m)

main = let Trace (lst, m) = fact 3
       in do
           print lst
           print m
```

```
newtype Trace a = Trace ([String], a)

instance Monad Trace where
    return x = Trace ([], x)
    (Trace (lst, x)) >>= k =
        let Trace (lst', y) = k x
        in Trace (lst ++ lst', y)

put :: Show a => String -> a -> Trace ()
put msg v = Trace ([msg ++ " " ++ show v], ())

fact :: Integer -> Trace Integer
fact n = do
   put "fact" n
   if n == 0
       then return 1
       else do
           m <- fact (n - 1)
           return (n * m)

main = let Trace (lst, m) = fact 3
       in do
```

Ex 3. Instead of `deriving Show`, define explicit instances of the `Show` typeclass for `Operator` and `Tree` such that `expr` is displayed as:

```
x = (13.0 - 1.0) / y
```

It's enough that you provide the implementation of the `show` function in the instance declaration. This function should take an `Operator` (or a `Tree`) and return a string.

```
data Operator = Plus | Minus | Times | Div

data Tree = SumNode Operator Tree Tree
          | ProdNode Operator Tree Tree
          | AssignNode String Tree
          | UnaryNode Operator Tree
          | NumNode Double
          | VarNode String

instance Show Operator where
    show Plus  = " + "
    ...

instance Show Tree where
    show = undefined

expr = AssignNode "x" (ProdNode Div (SumNode Minus (NumNode 13) (NumNode 1)) (VarNode "y"))

main = print expr
```

```
data Operator = Plus | Minus | Times | Div

data Tree = SumNode Operator Tree Tree
          | ProdNode Operator Tree Tree
          | AssignNode String Tree
          | UnaryNode Operator Tree
          | NumNode Double
          | VarNode String

instance Show Operator where
    show Plus  = " + "
    show Minus = " - "
    show Times = " * "
    show Div   = " / "

instance Show Tree where
    show (SumNode op lft rgt) = "(" ++ show lft ++ show op ++ show rgt ++ ")"
    show (ProdNode op lft rgt) = show lft ++ show op ++ show rgt
    show (AssignNode str tree) = str ++ " = " ++ show tree
    show (UnaryNode op tree) = show op ++ show tree
    show (NumNode x) = show x
    show (VarNode str) = str
```

Ex 4 This is an example that mimics elements of OO programming. Chess pieces are implemented as separate data types: here, for simplicity, just one, `Pawn`. The constructor of `Pawn` takes the `Color` of the piece and its position on the board (`0-7` in both dimensions). Pieces are instances of the class `Piece`, which declares the following functions: `color`, `pos`, and `moves`. The `moves` function takes a piece and returns a list of possible future positions after one move (without regard to other pieces, but respecting the boundaries of the board). Define both the typeclass and the instance, so that the following program works:

```
data Color = White | Black
    deriving (Show, Eq)

data Pawn = Pawn Color (Int, Int)

class Piece a where
   ...

instance Piece Pawn where
   ...

pieces = [Pawn White (3, 1), Pawn Black (4, 1), Pawn White (0, 7), Pawn Black (5, 0)]

main = print $ map moves pieces
```

```
data Color = White | Black
    deriving (Show, Eq)

data Pawn = Pawn Color (Int, Int)

class Piece a where
   color :: a -> Color
   pos   :: a -> (Int, Int)
   moves :: a -> [(Int, Int)]

instance Piece Pawn where
   color (Pawn c _) = c
   pos (Pawn _ pos) = pos
   moves pwn = if color pwn == White
               then mvs (pos pwn)
               else map refl (mvs $ refl (pos pwn))
     where
       refl (x, y) = (x, 7 - y)
       mvs (x, y) = if y == 1
                    then [(x, y + 1), (x, y + 2)]
                    else if y == 7
                         then []
```

The Symbolic Calculator So Far

Below is the source code for the current state of the project. Notice how `lookUp` and `addSymbol` nicely fit into the monadic scheme. I haven't made any changes to the lexer and parser files.

```
{-# START_FILE Main.hs #-}
module Main where

import qualified Data.Map as M
import Lexer (tokenize)
import Parser (parse)
import Evaluator

main = do
   loop (M.fromList [("pi", pi), ("e", exp 1.0)])

loop symTab = do
   str <- getLine
   if null str
   then
      return ()
   else
      let toks  = tokenize str
          tree  = parse toks
          Ev ev = evaluate tree symTab
      in
          case ev of
```

jphedley • 7 months ago

To compile custom Evaluator Monad post ghc 7.10, for which Functor & Applicative are Monad superclasses, the following will work

```
import Control.Applicative
import Control.Monad (liftM, ap)
...
instance Functor Evaluator where
fmap = liftM

instance Applicative Evaluator where
pure = return
(<*>) = ap
....
```

although ghc release notes recommends not using pure=return in production code.

Alexander Vorobiev • 2 years ago

Should the examples and exercises be updated for GHC 7.10 (https://ghc.haskell.org/tra... Without making all monads instances of both Applicative and Functor the code produces No instance for (Applicative ...) errors.

Syver Enstad • 3 years ago

This chapter was very useful in understanding Monads, I've appreciated your way of explaining since the Relisoft windows tutorials back in the days

Rob Grainger • 3 years ago

"Error handling is fundamenta to all programming."
But not to proof-reading apparently!