# EXERCISE 17.1 - PROGRAMMING IN HASKELL - HUTTON

## 1. Background Code

Here's what has been given to us in either the statement or in the text. To reflect that computations may now fail, we must change the type of the elements of the stack from `Int` by `Maybe Int`. The parts to be completed will be surrounded by curly braces.

```
type Stack = [Maybe Int]
data Expr = Val Int
          | Add Expr Expr
          | Throw
          | Catch Expr Expr
          deriving Show
data Code = HALT
          | PUSH Int Code
          | ADD Code
          | {new THROW constructor}
          | {new CATCH constructor}

eval :: Expr -> Maybe Int
eval (Val n)      = Just n
eval (Add x y)    = case eval x of
                      Nothing -> Nothing
                      Just n  -> case eval y of
                          Just m  -> Just (n + m)
                          Nothing -> Nothing
eval Throw        = Nothing
eval (Catch x h) = case eval x of
                      Just n  -> Just n
                      Nothing -> eval h

exec :: Code -> Stack -> Stack
exec HALT        s              = s
exec (PUSH n c) s              = exec c (Just n : s)
exec (ADD c)     (my : mx : s) = exec c {complete...}
exec (THROW ...              }  = exec c {complete...}
exec (CATCH ...              }  = exec c {complete...}

comp' :: Expr -> Code -> Code
comp' (Val n)     c = PUSH n c
comp' (Add x y)   c = {complete...}
comp' Throw       c = {complete...}
comp' (Catch x h) c = {complete...}

comp :: Expr -> Code
comp e = comp' e HALT
```

The new definitions are required to satisfy the same two compiler correctness equations:

(i) `exec (comp' e c) s = exec c (eval e : s)`
(ii) `exec (comp e) s = eval e : s`

However, (ii) follows immediately from (i) together with the definition

`comp e = comp' e HALT`

As a result, we neither use nor have to worry about (ii).

## 2. Deriving the new definitions

We consider the four possibilities for an expression `e` and try to guess or deduce the formulas for the functions involved in each of these cases, using (i) and induction. The parts that should be incorporated to the code are highlighted.

**Case 1:** `e = Val n`**.** The only difference to the original version is that `eval (Val n)` yields `Just n` instead of `n` and similarly, when executing a push, we push `Just n` onto the stack. These modifications are already contained in §1. □

**Case 2:** `e = Add x y`**.**

```
  exec (comp' (Add x y) c) s      {specification (i)}
= exec c (eval (Add x y) : s)     {applying eval, using do}
= exec c (do n <- eval x
             m <- eval y
             return (n + m)) : s  {to be continued...}
```

Now we need to rewrite this as `exec c' s'`, that is, we must solve the equation

`exec c' s' = exec c (do ...) : s`

To use induction, we need to push `n` and `m` onto the stack. So we can choose

`s' = m : n : s = eval y : eval x : s`

For `c'` we use the same constructor

`ADD :: Code -> Code`

as before, but because `Maybe Int` does not directly support `+`, we need to modify `exec`:

```
exec (ADD c) (my : mx : s) = exec c (sum : s) where
    sum = do
        n <- mx
        m <- my
        return (n + m)
```

We are now able to continue our derivation:

```
= exec c (do ...) : s                  {unapplying exec}
= exec (ADD c) (eval y : eval x : s)   {induction for (i)}
= exec (comp' y (ADD c)) (eval x : s)  {induction for (i)}
= exec (comp' x (comp' y (ADD c))) s
```

Comparing the first argument of `exec` in this and in the initial expression, we are forced to define

```
comp' (Add x y) c = comp' x (comp' y (ADD c))
```

exactly as in the original version. □

**Case 3:** `e = Throw`**.** The underlying idea here is that evaluating `Throw` should result in an exception, which is represented by `Nothing`. We again start from condition (i):

```
  exec (comp' Throw c) s       {specification (i)}
= exec c (eval Throw : s)      {applying eval}
= exec c (Nothing : s)         {to be continued...}
```

We need to rewrite this as `exec c' s'`, that is, we must solve

`exec c' s' = exec c (Nothing : s)`

Clearly, we should choose `s' = (Nothing : s)`. Also, in order to bind `c`, we create a new constructor in the `Code` type:

```
THROW :: Code -> Code
```

We set `c' = THROW c` and introduce a new case for `exec`:

```
exec (THROW c) s = exec c (Nothing : s)
```

Thus, continuing, we now have:

```
  exec c (Nothing : s)         {unapplying exec}
= exec (THROW c) s
```

Comparing the first and last terms in this chain of equations, we are forced to define

```
comp' Throw c = THROW c
```

This completes all of the necessary definitions for this case. □

**Case 4:** `e = Catch x h`**.** The idea is that evaluating `Catch x h` should result in `x` unless `x` fails (evaluates to `Nothing`); in this case, we fall back to evaluating the handler `h`.

Again we begin with equation (i). To abbreviate the notation, we will use the alternative operator `<|>` from `Control.Applicative`, which tries the first value and falls back to the second if the first is `Nothing`.

```
  exec (comp' (Catch x h) c) s              {specification (i)}
= exec c (eval (Catch x h) : s)             {applying eval}
= exec c (((eval x) <|> (eval h)) : s)      {to be continued...}
```

Once again, we must find `c'` and `s'` such that

`exec c' s' =  exec c (((eval x) <|> (eval h)) : s)`

Moreover, we probably will need to use the induction hypothesis in one of these forms:

```
exec c (eval x : s)  = exec (comp' x c) s
exec c (eval h : s)  = exec (comp' h c) s
```

If `eval x` and `eval h` were at the top of the stack in `s'`, we could use them to obtain the right side. So we assume that

`s' = (eval h) : (eval x) : s`

and now we need to choose `c'` so that

`exec c' (eval h : eval x : s) = exec c ((...) : s)`

Because `c` is still unbound, we introduce

```
CATCH :: Code -> Code
```

and then set `c' = CATCH c` to deduce that:

```
exec (CATCH c) (mh : mx : s) = exec c ((mx <|> mh) : s)
```

Substituting this into our aborted derivation and continuing:

```
  exec c (((eval x) <|> (eval h)) : s)    {unapplying exec}
= exec (CATCH c) (eval h : eval x : s)    {induction for (i)}
= exec (comp' h (CATCH c)) (eval x : s)   {induction for (i)}
= exec (comp' x (comp' h (CATCH c))) s
```

Finally, comparing the first argument of `exec` in the last expression and in the expression with which we began, we conclude that we must define

```
comp' (Catch x h) c = comp' x (comp' h (CATCH c))
```

This concludes the last case and the proof. □