

A Haskell Companion for “Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters”

Uk'taad B'mal

The University of Kansas - ITTC
2335 Irving Hill Rd, Lawrence, KS 66045
lambda@ittc.ku.edu

March 3, 2004

Abstract

This document is a primer to accompany the paper “Using catamorphisms, subtypes, and monad transformers for writing modular functional interpreters” by Luc Duponcheel. It attempts to re-implement in `Haskell` the interpreters written in `Gopher`. In addition to the interpreters, examples expressions are provided for each interpreter to help readers understand the abstract syntax interpreted. Each interpreter is also extended to demonstrate how well the modular interpreter achieves its modularity goal.

1 Introduction

Among them most accessible papers on composable interpreters is Duponcheel’s “Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters.” Unfortunately, the code examples are written in `Gopher` rather than `Haskell` and there are no examples of actual expressions or expression evaluation. This document attempts to address this problem by providing `Haskell` source in literal script as well as example expressions for the several interpreters. Three different interpreters are defined based on Duponcheel’s examples. In addition, examples of extension are included for the two modular interpreters.

2 First Try

Module `FirstTry` presents a Haskell encoding of Duponcheel’s First Try at building a modular interpreter:

```
module FirstTry where
```

The interpreter defined is a simple implementation of a language that supports adding and dividing integer numbers. The approach is to first define an interpreter for numbers and addition (`eval1`), then extend the interpreter to include division (`eval2`).

First, we define data structures for `Expr1` representing numbers and addition:

```
data Expr1  
  = Num1 Int  
  | Add1 Expr1 Expr1  
  deriving (Eq, Show)
```

Next, define semantic functions associated with the two operations. *num* simply returns the number encapsulated by *Num* while *add* sums the numbers associated with an *Add* term.

```
num = id      -- Num1 Int
add x y = x + y  -- Add1 Expr1 Expr1
```

The *eval₁* function associates semantics with abstract syntax. For each term defined in *Expr₁*, *eval₁* generates a semantic interpretation:

```
eval1 :: Expr1 → Int
eval1 (Num1 x) = num x      -- Num1 x
eval1 (Add1 x y) = eval1 x 'add' eval1 y  -- Add1 x y
```

We now have an interpreter defined for numbers and addition operations where both are defined as constructors in a single data type. What we are attempting to demonstrate is that such interpreters can be extended by adding further operations and types. In this case, we'll add a division operation to the initial interpreter.

Because the initial interpreter is not particularly modular, the extension requires us to touch almost all of the original code. First, the *Expr₁* data type is modified to include a division operation and becomes *Expr*:

```
data Expr
  = Num Int
  | Add Expr Expr
  | Dvd Expr Expr
  deriving (Show, Eq)
```

We can reuse the semantic definitions for *Num* and *Add*, but we need a semantic definition for *Dvd*:

```
dvd x y = if y ≡ 0 then error "Divide by zero" else x 'div' y
```

The new *eval* function extends *eval₁* by adding a case for the division operation. Note that the function must be rewritten and cannot simply be extended.

```
eval :: Expr → Int
eval (Num x) = id x
eval (Add x y) = eval x 'add' eval y
eval (Dvd x y) = eval x 'dvd' eval y
```

With this, we have an interpreter for an expression language that allows numbers, addition and division. Each new type or operation requires the same effort of rewriting expression syntax, adding semantics and modifying the evaluation function. With the exception of the semantic functions, virtually none of the existing code goes unmodified. For small interpreters, this is not a problem. For larger, more complex interpreters extension becomes more difficult.

3 Second Try

Module *SecondTry* presents a Haskell encoding of Duponcheel's Second Try and building a modular interpreter:

module *SecondTry* **where**

Again, we define data structures for E_1 representing the abstract syntax of numbers and addition, and E_2 representing the abstract syntax of division. Unlike First Try, these data structures are separate types::

```
data  $E_1$   $x$ 
  = Num Int
  | Add x x
  deriving (Eq, Show)
```

```
data  $E_2$   $x = Dvd x x$  deriving (Eq, Show)
```

The difficult with expressions as defined in First Try is their recursive nature. Because of this, the single data structure representing expression must be modified each time the language is updated. Here, we use E_1 and E_2 to define expressions that are not directly recursive. The data constructors *In1* and *In2* prevent general recursion in the data types:

```
data  $Expr_1 = In1 (E_1 Expr_1)$  deriving (Eq, Show)
```

```
data  $Expr_2 = In2 (E_2 Expr_2)$  deriving (Eq, Show)
```

Semantic functions for each of the operations remain unchanged and will remain unchanged in all our interpreters:

```
num = id    -- Num1 Int
add x y = x + y    -- Add1 Expr1 Expr1
dvd x y = if y == 0 then error "Divide by zero" else x `div` y
```

Two helper functions are used to define evaluation functions for each of the data types. A *map* function distributes a function across an expression. Specifically, if g is a function, then mapping g onto *Add x y* is equal to *Add (g x) (g y)*. The *map* function will be used to map *eval* functions onto language terms. The ϕ functions map syntax to semantics. Each ϕ function maps a term to its semantic interpretation.

```
map1 :: ( $x \rightarrow y$ )  $\rightarrow$  ( $E_1 x \rightarrow E_1 y$ )
map1  $g$  (Num n) = Num n
map1  $g$  (Add e f) = (Add (g e) (g f))
```

```
map2 :: ( $x \rightarrow y$ )  $\rightarrow$  ( $E_2 x \rightarrow E_2 y$ )
map2  $g$  (Dvd e f) = (Dvd (g e) (g f))
```

```
 $\phi_1$  ::  $E_1 Int \rightarrow Int$ 
 $\phi_1$  (Num n) = num n
 $\phi_1$  (Add e f) = e `add` f
```

```
 $\phi_2$  ::  $E_2 Int \rightarrow Int$ 
 $\phi_2$  (Dvd e f) = e `dvd` f
```

An evaluation function becomes mapping the evaluation function onto a term and semantically interpreting the result. The functions *eval₁* and *eval₂* separately evaluate expressions $Expr_1$ and $Expr_2$ and are included as examples of writing evaluation functions. They will not be used to write the composite evaluation function for both terms. Instead, *map* and *psi* functions will be composed to form a new evaluation function.

```

eval1 :: Expr1 → Int
eval1 (In1 e1) = φ1 (map1 eval1 e1)

```

```

eval2 :: Expr2 → Int
eval2 (In2 e2) = φ2 (map2 eval2 e2)

```

The *Sum* data type is a general purpose data structure that defines the disjoint union of two other data types. The union is disjoint because the constructors *L* and *R* identify the original type any element of the sum must come from. (In a real implementation, the built-in *Either* type would be used rather than writing our own *Sum*.)

```

data Sum x y = L x | R y deriving (Show, Eq)

```

```

(< + >) :: (x → z) → (y → z) → (Sum x y → z)
l < + > r = λs → case s of
    L x → l x
    R x → r x

```

The type synonym, *E*, represents the sum of the abstract syntax data types for numbers and addition (*E*₁) and division (*E*₂). The *L* constructor will encapsulate numbers and addition while the *R* constructor will encapsulate division. The type *E* is the disjoint union and may thus be constructed with *L* or *R* and may therefore be either of the expression types. Thus, we now have a type that encapsulates both terms.

```

type E x = Sum (E1 x) (E2 x)

```

The *Expr* data type represents an individual expression encapsulated by the *InE* constructor. This is necessary to identify both *L* and *R* constructs from *Sum* as expressions.

```

data Expr = InE (E Expr)

```

With expressions defined, we can now define *map*, *φ* and *eval* functions using the same technique as for individual terms. The composite *map* function, *map_E*, uses *map1* and *map2* to define a composite map function. The construction is fascinating and is worth some investigation.

The signature of *MapE* indicates that we will take a function between two seemingly arbitrary types, *x* and *y*, and produces a function that maps the encapsulation of *x* as a term into the encapsulation of *y* as a term. This implies that *x* and *y* are not arbitrary at all. Because *E* is defined as the *Sum* of *E*₁ and *E*₂, we know that *x* and *y* must be terms from our abstract syntax.

To form the definition of *MapE*, we simply build a function for mapping *g* onto each expression type and encapsulate the result appropriately using the *L* and *R* constructors. *map1* and *map2* provide the appropriate mapping functions for each expression type and we know that expressions of type *E*₁ should be encapsulated with *L* and those of type *E*₂ with *R*. Thus, it is simple to build a function for each element of the abstract syntax by composing encapsulation and map functions. *L* ∘ *map1* *g* is a function that maps *g* onto its argument of type *E*₁ and encapsulates the result with *L*. Similarly, *R* ∘ *map2* *g* is a function that operates on expressions of type *E*₂.

Assembling the functions is easily achieved using the *< + >* operation from *Sum*. This operation simply examines its argument and determines whether it is encapsulated with *L* or *R*. If *L*, the first function argument is selected and applied. If *R*, the second. Thus, *map_E* composes *map1* and *map2* to form a mapping function that works on all term types.

$$\begin{aligned} \text{map}_E &:: (x \rightarrow y) \rightarrow (E x \rightarrow E y) \\ \text{map}_E g &= (L \circ \text{map1 } g) < + > (R \circ \text{map2 } g) \end{aligned}$$

Building *PhiE* is analogous to building *map_E*, except that the domain argument is already encapsulated with *L* or *R*. This, the composition operator is used directly to compose ϕ_1 and ϕ_2 .

$$\begin{aligned} \phi_E &:: E \text{ Int} \rightarrow \text{Int} \\ \phi_E &= \phi_1 < + > \phi_2 \end{aligned}$$

Building *EvalE* is exactly analogous to building *Eval1* or *Eval2*. *EvalE* is first mapped onto the expression to evaluate sub-terms and ϕ_E associates the result with its operational semantics.

$$\begin{aligned} \text{eval}_E &:: \text{Expr} \rightarrow \text{Int} \\ \text{eval}_E (\text{InE } e) &= \phi_E (\text{map}_E \text{eval}_E e) \end{aligned}$$

Following are some examples of expressions being evaluated. These things aren't trivial to figure out from the paper because there are no example expressions. Note that the constructors for the sum type must be specified along with the actual value. Further, the constructor for the outermost expression must also be present as well. I didn't anticipate this when I went through the examples the first time.

```
-- Return the value 1
test0 = eval_E (InE (L (Num 1)))

-- Add 1 and 1
test1 = eval_E (InE (L (Add (InE (L (Num 1))) (InE (L (Num 1)))))

-- Divide 1 by 1
test2 = eval_E (InE
  (R (Dvd
    (InE (L (Num 1)))
    (InE (L (Num 1)))))

-- Divide 1 by 0
test3 = eval_E (InE
  (R (Dvd
    (InE (L (Num 1)))
    (InE (L (Num 0)))))

-- Divide 1 by 1 and add 1
test4 = eval_E (InE
  (L (Add
    (InE (R (Dvd
      (InE (L (Num 1)))
      (InE (L (Num 1)))))
    (InE (L (Num 1)))))
```

4 Second Try Plus Multiplication

So what's the point? Duponcheel has taken a pretty simple interpreter and obfuscated it to give us an interpreter that uses what will become fixed point types. To demonstrate the point, we'll extend the Second

Try interpreter to include a new term that performs multiplication. Although all the code is repeated here, it is surprising how little is modified.

```
module SecondTryPlus where
```

The definitions for E_1 and E_2 are unchanged. A new, separate definition is added for E_3 to represent multiplication terms:

```
data  $E_1$   $x$ 
  = Num Int
  | Add x x
  deriving (Eq, Show)

data  $E_2$   $x = Dvd$   $x$   $x$  deriving (Eq, Show)

data  $E_3$   $x = Mul$   $x$   $x$  deriving (Eq, Show)
```

The definitions for $Expr_1$ and $Expr_2$ are unchanged. A new definition for $Expr_3$ representing multiplication is added.

```
data  $Expr_1 = In1$  ( $E_1$   $Expr_1$ ) deriving (Eq, Show)

data  $Expr_2 = In2$  ( $E_2$   $Expr_2$ ) deriving (Eq, Show)

data  $Expr_3 = In3$  ( $E_3$   $Expr_3$ ) deriving (Eq, Show)
```

Semantic functions also remain unchanged. A function for multiplication is added:

```
num = id    -- Define semantics of num
add x y = x + y    -- Define semantics of add
dvd x y = if y == 0    -- Define semantics of divide
  then error "Divide by zero"
  else  $x \text{ `div` } y$ 
mul x y = x * y    -- Define semantics of times
```

The original map and ϕ functions again remain unmodified. $map3$ and ϕ_3 are added for multiplication operations. $eval_3$ is also added to show that we can build an interpreter separately for multiplication.

```
map1 :: ( $x \rightarrow y$ )  $\rightarrow$  ( $E_1$   $x \rightarrow E_1$   $y$ )
map1  $g$  (Num  $n$ ) = Num  $n$ 
map1  $g$  (Add  $e$   $f$ ) = (Add ( $g$   $e$ ) ( $g$   $f$ ))

map2 :: ( $x \rightarrow y$ )  $\rightarrow$  ( $E_2$   $x \rightarrow E_2$   $y$ )
map2  $g$  (Dvd  $e$   $f$ ) = (Dvd ( $g$   $e$ ) ( $g$   $f$ ))

map3 :: ( $x \rightarrow y$ )  $\rightarrow$  ( $E_3$   $x \rightarrow E_3$   $y$ )
map3  $g$  (Mul  $e$   $f$ ) = (Mul ( $g$   $e$ ) ( $g$   $f$ ))

 $\phi_1$  ::  $E_1$   $Int \rightarrow Int$ 
```

$\phi_1 (Num\ n) = num\ n$
 $\phi_1 (Add\ e\ f) = e\ 'add'\ f$

$\phi_2 :: E_2\ Int \rightarrow Int$
 $\phi_2 (Dvd\ e\ f) = e\ 'dvd'\ f$

$\phi_3 :: E_3\ Int \rightarrow Int$
 $\phi_3 (Mul\ e\ f) = e\ 'mul'\ f$

$eval_1 :: Expr_1 \rightarrow Int$
 $eval_1 (In1\ e1) = \phi_1 (map1\ eval_1\ e1)$

$eval_2 :: Expr_2 \rightarrow Int$
 $eval_2 (In2\ e2) = \phi_2 (map2\ eval_2\ e2)$

$eval_3 :: Expr_3 \rightarrow Int$
 $eval_3 (In3\ e3) = \phi_3 (map3\ eval_3\ e3)$

The definition for sum does not change.

data $Sum\ x\ y = L\ x \mid R\ y$ **deriving** $(Show, Eq)$

$(< + >) :: (x \rightarrow z) \rightarrow (y \rightarrow z) \rightarrow (Sum\ x\ y \rightarrow z)$
 $l < + > r = \lambda s \rightarrow$ **case** s **of**
 $\quad L\ x \rightarrow l\ x$
 $\quad R\ x \rightarrow r\ x$

All we have done to this point is define an interpreter for multiplication operations by defining map , ϕ and $eval$ for the new abstract syntax. None of the previous definitions have been modified in any way. We will now integrate the multiplication syntax into the abstract syntax for the entire language and integrate the interpreters.

First, define a type shorthand that adds E3 to the expression. Effectively, the abstract syntax for expressions becomes:

$(Sum\ multiplication\ (Sum\ division\ addition))$

The $Expr$ data structure does not change.

type $E\ x = Sum\ (E_3\ x)\ (Sum\ (E_1\ x)\ (E_2\ x))$

data $Expr = InE\ (E\ Expr)$

map_E uses the sum composition operator to map an operation across the entire expression abstract syntax. The signature does not change. The function body rebuilds the expression around the application of map to each possible expression type. Note the use of R and L to build what is in effect a tree containing possible expressions. Any number of expressions can be added in this way. ϕ_E is similarly modified and $eval_E$ remains unchanged.

```

map_E :: (x -> y) -> (E x -> E y)
map_E g = (L o map3 g) < + > ((R o L o map1 g) < + > (R o R o map2 g))

```

```

phi_E :: E Int -> Int
phi_E = phi3 < + > (phi1 < + > phi2)

```

```

eval_E :: Expr -> Int
eval_E (InE e) = phi_E (map_E eval_E e)

```

Following are more examples the integrate multiplication. Note how *R* and *L* must be used together to construct some terms. This is painful without a concrete syntax to compile from, but simple when writing compilers that automatically generate structures.

```

-- Return the value 1
test0 = eval_E (InE (R (L (Num 1))))

-- Add 1 and 1
test1 = eval_E (InE (R (L (Add (InE (R (L (Num 1)))) (InE (R (L (Num 1))))))))

-- Divide 1 by 1
test2 = eval_E (InE (R (R (Dvd (InE (R (L (Num 1)))) (InE (R (L (Num 1))))))))

test3 = eval_E (InE (L (Mul (InE (R (L (Num 2)))) (InE (R (L (Num 2)))))))

test4 = eval_E (InE (R (R (Dvd (InE (R (L (Num 1)))) (InE (R (L (Num 1))))))))

```

Now that we've done it the hard way, let's make things easier by defining names for operators and operands. Define some useful variables and functions that provide a kind of ad hoc syntax that we'll use to define expressions.

```

num1 = (InE (R (L (Num 1))))
num2 = (InE (R (L (Num 2))))
num3 = (InE (R (L (Num 3))))
numx x = (InE (R (L (Num x))))
divide x y = (InE (R (R (Dvd x y))))
times x y = (InE (L (Mul x y)))
plus x y = (InE (R (L (Add x y))))

```

We can now write expressions that look much more like we think they should. Use `evalE` to evaluate the following examples:

```

test5 = (num2 'times' num3)

test6 = (num1 'plus' (num2 'divide' num3))

test7 = (num1 'plus' (num3 'divide' num2))

test8 = (num2 'times' (numx 100))

test9 = (num2 'divide' (numx 0))

```

Any reasonable assessment of this interpreter reveals that to add a new term type to the interpreter, we simply need to define an interpreter for that term type and compose it with those that already exist. Composing the new evaluation function is quite systematic and achieved using features of the *Sum* type constructor. No modification is required for existing interpreter modules. In addition to dramatically simplifying interpreter construction, debugging becomes much simpler as the new interpreter can be tested and debugged in a modular fashion.

5 Third Try

The Third Try from Duponcheel attempts to abstract common structures from individual interpreters and syntax into a common form. The result is a parameterized data structure that forms fixed point forms in a common, consistent manner.

```
module ThirdTry where
```

All the *Expr* types from *SecondTry* share a common form that can be abstracted and parameterized as *Rec*:

```
data Rec f = In (f (Rec f))
```

The *Rec* type creates a fixed point data type from some element *f*. Note the constructor *In* that accomplishes the same task as *In1*, *In2*, and *InE* from previous examples.

We will need to define abstract syntax for each expression type. These definitions do not change from earlier examples, they will simply be packaged differently for the interpreter.

```
data E1 x
  = Num Int
  | Add x x
  deriving (Eq, Show)
```

```
data E2 x = Dvd x x deriving (Eq, Show)
```

Now use *Rec* to define fixed point types for *Expr₁* and *Expr₂*.

```
type Expr1 = Rec E1
type Expr2 = Rec E2
```

The definitions for semantic functions, *map*, *φ*, and *eval* functions for each expression do not change.

```
num = id      -- Num1 Int
add x y = x + y    -- Add1 Expr1 Expr1
dvd x y = if y ≡ 0 then error "Divide by zero" else x `div` y
```

```
map1 :: (x → y) → (E1 x → E1 y)
map1 g (Num n) = Num n
map1 g (Add e f) = (Add (g e) (g f))
```

```
map2 :: (x → y) → (E2 x → E2 y)
```

$$\text{map2 } g \text{ (Dvd } e \text{ f)} = (\text{Dvd } (g \text{ e)} (g \text{ f}))$$

$$\begin{aligned} \phi_1 &:: E_1 \text{ Int} \rightarrow \text{Int} \\ \phi_1 (\text{Num } n) &= \text{num } n \\ \phi_1 (\text{Add } e \text{ f}) &= e \text{ 'add' } f \end{aligned}$$

$$\begin{aligned} \phi_2 &:: E_2 \text{ Int} \rightarrow \text{Int} \\ \phi_2 (\text{Dvd } e \text{ f}) &= e \text{ 'dvd' } f \end{aligned}$$

$$\begin{aligned} \text{eval}_1 &:: \text{Expr}_1 \rightarrow \text{Int} \\ \text{eval}_1 (\text{In } e1) &= \phi_1 (\text{map1 } \text{eval}_1 \text{ e1}) \end{aligned}$$

$$\begin{aligned} \text{eval}_2 &:: \text{Expr}_2 \rightarrow \text{Int} \\ \text{eval}_2 (\text{In } e2) &= \phi_2 (\text{map2 } \text{eval}_2 \text{ e2}) \end{aligned}$$

The definition of *Sum* remains unchanged:

data *Sum* *x y* = *L x* | *R y* **deriving** (*Show*, *Eq*)

$$\begin{aligned} (< + >) &:: (x \rightarrow z) \rightarrow (y \rightarrow z) \rightarrow (\text{Sum } x \text{ y} \rightarrow z) \\ l < + > r &= \lambda s \rightarrow \text{case } s \text{ of} \\ &\quad L \text{ x} \rightarrow l \text{ x} \\ &\quad R \text{ x} \rightarrow r \text{ x} \end{aligned}$$

Now the fun begins. An expression in our language is the sum of the expression for addition and division. Thus, we use the *Sum* type to do our composition. In Gopher, the **type** constructor behaves differently than the **type** synonym definition operator in Haskell. Thus, the definition is altered to use **newtype** requiring the insertion of a new type constructor, *MkE*. In addition, an un-constructor is defined to pull data out of *MkE*.

newtype *E* *x* = *MkE* (*Sum* (*E*₁ *x*) (*E*₂ *x*))

unE (*MkE* *x*) = *x*

The composite *Expr* definition is the fixed point type created from *E* that we just defined.

type *Expr* = *Rec E*

The map function for the composite expression, *E*, is basically the same function with the addition of functions to accommodate the *MkE* constructor we had to introduce earlier. Specifically, *map_E* must first extract the expression from the *MkE* constructor because the individual map functions aren't aware of the need to package the expression. Once processed, the expression is repackaged. Similarly, *φ_E* must un-package its argument. Because it is making a semantic mapping, its result should not be repackaged. *eval_E* does not change.

$$\begin{aligned} \text{map}_E &:: (x \rightarrow y) \rightarrow (E \text{ x} \rightarrow E \text{ y}) \\ \text{map}_E \text{ g} &= \text{MkE} \circ ((L \circ \text{map1 } \text{g}) < + > (R \circ \text{map2 } \text{g})) \circ \text{unE} \end{aligned}$$

$$\begin{aligned} \phi_E &:: E \text{ Int} \rightarrow \text{Int} \\ \phi_E &= (\phi_1 < + > \phi_2) \circ \text{unE} \end{aligned}$$

```

eval_E :: Expr → Int
eval_E (In e) = φ_E (map_E eval_E e)

```

Again, some examples to try out.

```

-- Return the value 1
test_0 = eval_E (In (MkE (L (Num 1))))

-- Add 1 and 1
test_1 = eval_E (In (MkE
  (L (Add
    (In (MkE (L (Num 1))))
    (In (MkE (L (Num 1))))))))))

-- Divide 1 by 1
test_2 = eval_E (In (MkE
  (R (Dvd
    (In (MkE (L (Num 1))))
    (In (MkE (L (Num 1))))))))))

```

6 Third Try Plus Multiplication

```

module ThirdTryPlus where

```

The *Rec* type remains unchanged.

```

data Rec f = In (f (Rec f))

```

Abstract syntax is added for multiplication.

```

data E1 x
  = Num Int
  | Add x x
  deriving (Eq, Show)

data E2 x = Dvd x x deriving (Eq, Show)

data E3 x = Mul x x deriving (Eq, Show)

```

An expression type is added for multiplication using *Rec*.

```

type Expr1 = Rec E1
type Expr2 = Rec E2
type Expr3 = Rec E3

```

The definitions for semantic functions, *map*, *φ*, and *eval* functions for each expression do not change. New instances are added for multiplication. These are identical to those added for Second Try Plus Multiplication.

```

num = id      -- Num1 Int
add x y = x + y    -- Add1 Expr1 Expr1
dvd x y = if y == 0 then error "Divide by zero" else x `div` y
mul x y = x * y    -- Define semantics of times

```

```

map1 :: (x -> y) -> (E1 x -> E1 y)
map1 g (Num n) = Num n
map1 g (Add e f) = (Add (g e) (g f))

```

```

map2 :: (x -> y) -> (E2 x -> E2 y)
map2 g (Dvd e f) = (Dvd (g e) (g f))

```

```

map3 :: (x -> y) -> (E3 x -> E3 y)
map3 g (Mul e f) = (Mul (g e) (g f))

```

```

phi1 :: E1 Int -> Int
phi1 (Num n) = num n
phi1 (Add e f) = e `add` f

```

```

phi2 :: E2 Int -> Int
phi2 (Dvd e f) = e `dvd` f

```

```

phi3 :: E3 Int -> Int
phi3 (Mul e f) = e `mul` f

```

```

eval1 :: Expr1 -> Int
eval1 (In e1) = phi1 (map1 eval1 e1)

```

```

eval2 :: Expr2 -> Int
eval2 (In e2) = phi2 (map2 eval2 e2)

```

```

eval3 :: Expr3 -> Int
eval3 (In e3) = phi3 (map3 eval3 e3)

```

The definition of *Sum* remains unchanged:

```

data Sum x y = L x | R y deriving (Show, Eq)

```

```

(< + >) :: (x -> z) -> (y -> z) -> (Sum x y -> z)
l < + > r = \s -> case s of
    L x -> l x
    R x -> r x

```

The changes to the composite structure mimic those done to extend try. Nothing else changes.

```

newtype E x = MkE (Sum (E3 x) (Sum (E1 x) (E2 x)))

```

```

unE (MkE x) = x

```

```

type Expr = Rec E

```

```

map_E :: (x -> y) -> (E x -> E y)
map_E g
  = MkE ◦ ((L ◦ map3 g) < + > ((R ◦ L ◦ map1 g) < + > (R ◦ R ◦ map2 g))) ◦ unE

```

```

phi_E :: E Int -> Int
phi_E = (phi3 < + > (phi1 < + > phi2)) ◦ unE

```

```

eval_E :: Expr -> Int
eval_E (In e) = phi_E (map_E eval_E e)

```

Again, some examples to try out.

```

-- Return the value 1
test0 = eval_E (In (MkE (R (L (Num 1)))))

-- Add 1 and 1
test1 = eval_E (In
  (MkE (R (L (Add
    (In (MkE (R (L (Num 1)))))
    (In (MkE (R (L (Num 1)))))
  )))))

-- Divide 1 by 1
test2 = eval_E (In
  (MkE (R (R (Dvd
    (In (MkE (R (L (Num 1)))))
    (In (MkE (R (L (Num 1)))))
  )))))

test3 = eval_E (In (MkE (L (Mul
  (In (MkE (R (L (Num 2)))))
  (In (MkE (R (L (Num 2)))))
)))

test4 = eval_E (In (MkE (R (R (Dvd
  (In (MkE (R (L (Num 1)))))
  (In (MkE (R (L (Num 1)))))
))))))

```

Again, adding some trivial syntax:

```

num1 = (In (MkE (R (L (Num 1)))))
num2 = (In (MkE (R (L (Num 2)))))
num3 = (In (MkE (R (L (Num 3)))))
numx x = (In (MkE (R (L (Num x)))))
divide x y = (In (MkE (R (R (Dvd x y)))))
times x y = (In (MkE (L (Mul x y))))
plus x y = (In (MkE (R (L (Add x y)))))

test5 = (num2 'times' num3)

test6 = (num1 'plus' (num2 'divide' num3))

test7 = (num1 'plus' (num3 'divide' num2))

```

```
test8 = (num2 'times' (numx 100))
```

```
test9 = (num2 'divide' (numx 0))
```

Once again, we find that extending the interpreter is reasonably easy by defining a new interpreter for the new abstract syntax constructs. The *Rec* type simplifies this process somewhat and makes the common structure of terms clear. Furthermore, it unequivocally shows that the expression data types are in fact fixed point data types.

7 Functors and Algebras

The structure of *map* and ϕ functions have a common form that can be abstracted into **class** definitions. A *Functor* must define a *map* function that takes a function *f* and maps it over a function definition. Definitions for *E*₁, *E*₂, *E*₃, and *E* are all functors whose *map* functions are already defined as *map*₁, *map*₂, *map*₃, and *map*_{*E*}.

```
class Functor f where
```

```
  map :: (x → y) → (f x → f y)
```

```
instance ThirdTryPlus.Functor E1 where
```

```
  map = map1
```

```
instance ThirdTryPlus.Functor E2 where
```

```
  map = map2
```

```
instance ThirdTryPlus.Functor E3 where
```

```
  map = map3
```

```
instance ThirdTryPlus.Functor E where
```

```
  map = mapE
```

The common structure for ϕ functions is abstracted as an *Algebra* that defines a function, ϕ , that maps a *Functor*, *f*, and argument of type *a* to a value of type *a*. The ϕ functions define algebras for each expression type.

```
class ThirdTryPlus.Functor f ⇒ Algebra f a where
```

```
  φ :: f a → a
```

```
instance Algebra E1 Int where
```

```
  φ = φ1
```

```
instance Algebra E2 Int where
```

```
  φ = φ2
```

```
instance Algebra E3 Int where
```

```
  φ = φ3
```

```
instance Algebra E Int where
```

```
  φ = φE
```

We can now define a general *eval* function for all algebras. Given *Algebra f a*, an *eval* function is directly defined as follows:

```
eval :: Algebra f a => Rec f -> a
eval (In e) = phi (ThirdTryPlus.map eval e)
```

Unfortunately, the general *eval* function doesn't enable enough type inference to find a type value for *a*. Thus, the following *evalInt* function is generally defined for evaluation functions that return *Int* types:

```
evalInt :: Algebra f Int => Rec f -> Int
evalInt (In e) = phi (ThirdTryPlus.map eval e)
```

8 Usage

This file is a template for transforming literate script into \LaTeX and is not actually a `Haskell` interpreter implementation. Each section in this file is a separate module that can be loaded individually for experimentation.

Note that the interpreters have been developed under `GHC` and some require turning on the Glasgow Extensions. Your mileage may vary if you're using `HUGS`.

To build a \LaTeX document from the interpreter files, use:

```
lhs2TeX --math Duponcheel.lhs > Duponcheel.tex
```

and run \LaTeX on the result. The individual interpreters cannot be transformed to \LaTeX directly.