ITMO FP Course Homework Checker Download

Homework #0

Task 1

1. Create a module named `HW0.T1` and define the following type in it:

```
data a <-> b = Iso (a -> b) (b -> a)

flipIso :: (a <-> b) -> (b <-> a)
flipIso (Iso f g) = Iso g f

runIso :: (a <-> b) -> (a -> b)
runIso (Iso f _) = f
```

2. Implement the following functions and isomorphisms:

```
distrib :: Either a (b, c) -> (Either a b, Either a c)
assocPair :: (a, (b, c)) <-> ((a, b), c)
assocEither :: Either a (Either b c) <-> Either (Either a b) c
```

Task 2

1. Create a module named `HW0.T2` and define the following type in it:

```
type Not a = a -> Void
```

2. Implement the following functions and isomorphisms:

```
doubleNeg :: a -> Not (Not a)
reduceTripleNeg :: Not (Not (Not a)) -> Not a
```

Task 3

1. Create a module named `HW0.T3` and define the following combinators in it:

```
s :: (a -> b -> c) -> (a -> b) -> (a -> c)
s f g x = f x (g x)

k :: a -> b -> a
k x y = x
```

2. Using *only those combinators* and function application (i.e. no lambdas, pattern matching, and so on) define the following additional combinators:

```
i :: a -> a
compose :: (b -> c) -> (a -> b) -> (a -> c)
contract :: (a -> a -> b) -> (a -> b)
permute :: (a -> b -> c) -> (b -> a -> c)
```

   For example:

```
i x = x         -- No (parameters on the LHS disallowed)
i = \x -> x     -- No (lambdas disallowed)
i = Prelude.id  -- No (only use s and k)
i = s k k       -- OK
i = (s k) k     -- OK (parentheses for grouping allowed)
```

Task 4

1. Create a module named `HW0.T4`.

2. Using the `fix` combinator from the `Data.Function` module define the following functions:

```
repeat' :: a -> [a]             -- behaves like Data.List.repeat
map' :: (a -> b) -> [a] -> [b]  -- behaves like Data.List.map
fib :: Natural -> Natural       -- computes the n-th Fibonacci number
fac :: Natural -> Natural       -- computes the factorial
```

Do not use explicit recursion. For example:

```
repeat' = Data.List.repeat     -- No (obviously)
repeat' x = x : repeat' x      -- No (explicit recursion disallowed)
repeat' x = fix (x:)           -- OK
```

Task 5

1. Create a module named `HW0.T5` and define the following type in it:

```
type Nat a = (a -> a) -> a -> a
```

2. Implement the following functions:

```
nz :: Nat a
ns :: Nat a -> Nat a

nplus, nmult :: Nat a -> Nat a -> Nat a

nFromNatural :: Natural -> Nat a
nToNum :: Num a => Nat a -> a
```

3. The following equations must hold:

```
nToNum nz       ==  0
nToNum (ns x)   ==  1 + nToNum x

nToNum (nplus a b)   ==   nToNum a + nToNum b
nToNum (nmult a b)   ==   nToNum a * nToNum b
```

Task 6

1. Create a module named `HW0.T6` and define the following values in it:

```
a = distrib (Left ("AB" ++ "CD" ++ "EF"))     -- distrib from HW0.T1
b = map isSpace "Hello, World"
c = if 1 > 0 || error "X" then "Y" else "Z"
```

2. Determine the WHNF (weak head normal form) of these values:

```
a_whnf = ...
b_whnf = ...
c_whnf = ...
```

Homework #1

Task 1

1. Create a module named `HW1.T1` and define the following data type in it:

```
data Day = Monday | Tuesday | ... | Sunday
```

(Obviously, fill in the `...` with the rest of the week days).

Do not derive `Enum` for `Day`, as the derived `toEnum` is partial:

```
ghci> toEnum 42 :: Day
*** Exception: toEnum{Day}: tag (42) is outside of enumeration's range (0,6)
```

2. Implement the following functions:

```
-- | Returns the day that follows the day of the week given as input.
nextDay :: Day -> Day

-- | Returns the day of the week after a given number of days has passed.
```

```
afterDays :: Natural -> Day -> Day

-- | Checks if the day is on the weekend.
isWeekend :: Day -> Bool

-- | Computes the number of days until Friday.
daysToParty :: Day -> Natural
```

In `daysToParty`, if it is already `Friday`, the party can start immediately, we don't have to wait for the next week (i.e. return `0` rather than `7`).

Good job if you spotted that this task is a perfect fit for modular arithmetic, but that is not the point of the exercise. The functions must be implemented by operating on `Day` values directly, without conversion to a numeric representation.

Task 2

1. Create a module named `HW1.T2` and define the following data type in it:

   ```
   data N = Z | S N
   ```

2. Implement the following functions:

   ```
   nplus :: N -> N -> N        -- addition
   nmult :: N -> N -> N        -- multiplication
   nsub :: N -> N -> Maybe N   -- subtraction     (Nothing if result is negative)
   ncmp :: N -> N -> Ordering  -- comparison      (Do not derive Ord)
   ```

   The operations must be implemented without using built-in numbers (`Int`, `Integer`, `Natural`, and such).

3. Implement the following functions:

   ```
   nFromNatural :: Natural -> N
   nToNum :: Num a => N -> a
   ```

4. (Advanced) Implement the following functions:

   ```
   nEven, nOdd :: N -> Bool    -- parity checking
   ndiv :: N -> N -> N         -- integer division
   nmod :: N -> N -> N         -- modulo operation
   ```

   The operations must be implemented without using built-in numbers.

   In `ndiv` and `nmod`, the behavior in case of division by zero is not specified (you can throw an exception, go into an inifnite loop, or delete all files on the computer).

Task 3

1. Create a module named `HW1.T3` and define the following data type in it:

   ```
   data Tree a = Leaf | Branch Meta (Tree a) a (Tree a)
   ```

   The `Meta` field must store additional information about the subtree that can be accessed in constant time, for example its size. You can use `Int`, `(Int, Int)`, or a custom data structure:

   ```
   type Meta = Int         -- OK
   data Meta = M Int Int   -- OK
   ```

   Functions operating on this tree must maintain the following invariants:

   1. **Sorted**: The elements in the left subtree are less than the head element of a branch, and the elements in the right subtree are greater.
   2. **Unique**: There are no duplicate elements in the tree (follows from **Sorted**).
   3. **CachedSize**: The size of the tree is cached in the `Meta` field for constant-time access.
   4. (Advanced) **Balanced**: The tree is balanced according to one of the following strategies:

- **SizeBalanced**: For any given `Branch _ l _ r`, the ratio between the size of `l` and the size of `r` never exceeds `3`.
- **HeightBalanced**: For any given `Branch _ l _ r`, the difference between the height of `l` and the height of `r` never exceeds `1`.

These invariants enable efficient processing of the tree.

2. Implement the following functions:

```
-- | Size of the tree, O(1).
tsize :: Tree a -> Int

-- | Depth of the tree.
tdepth :: Tree a -> Int

-- | Check if the element is in the tree, O(log n)
tmember :: Ord a => a -> Tree a -> Bool

-- | Insert an element into the tree, O(log n)
tinsert :: Ord a => a -> Tree a -> Tree a

-- | Build a tree from a list, O(n log n)
tFromList :: Ord a => [a] -> Tree a
```

Tip 1: in order to maintain the **CachedSize** invariant, define a helper function:

```
mkBranch :: Tree a -> a -> Tree a -> Tree a
```

Tip 2: the **Balanced** invariant is the hardest to maintain, so implement it last. Search for “tree rotation”.

Task 4

1. Create a module named `HW1.T4`.
2. Using the `Tree` data type from `HW1.T3`, define the following function:

```
tfoldr :: (a -> b -> b) -> b -> Tree a -> b
```

It must collect the elements in order:

```
treeToList :: Tree a -> [a]    -- output list is sorted
treeToList = tfoldr (:) []
```

This follows from the **Sorted** invariant.

You are encouraged to define `tfoldr` in an efficient manner, doing only a single pass over the tree and without constructing intermediate lists.

Task 5

1. Create a module named `HW1.T5`.
2. Implement the following function:

```
splitOn :: Eq a => a -> [a] -> NonEmpty [a]
```

Conceptually, it splits a list into sublists by a separator:

```
ghci> splitOn '/' "path/to/file"
["path", "to", "file"]

ghci> splitOn '/' "path/with/trailing/slash/"
["path", "with", "trailing", "slash", ""]
```

Due to the use of `NonEmpty` to enforce that there is at least one sublist in the output, the actual GHCi result will look slightly differently:

```
ghci> splitOn '/' "path/to/file"
"path" :| ["to","file"]
```

Do not let that confuse you. The first element is not in any way special.

3. Implement the following function:

```
joinWith :: a -> NonEmpty [a] -> [a]
```

It must be the inverse of `splitOn`, so that:

```
(joinWith sep . splitOn sep)  ≡  id
```

Example usage:

```
ghci> "import " ++ joinWith '.' ("Data" :| "List" : "NonEmpty" : [])
"import Data.List.NonEmpty"
```

Task 6

1. Create a module named `HW1.T6`.
2. Using `Foldable` methods *only*, implement the following function:

```
mcat :: Monoid a => [Maybe a] -> a
```

Example usage:

```
ghci> mcat [Just "mo", Nothing, Nothing, Just "no", Just "id"]
"monoid"

ghci> Data.Monoid.getSum $ mcat [Nothing, Just 2, Nothing, Just 40]
42
```

3. Using `foldMap` to consume the list, implement the following function:

```
epart :: (Monoid a, Monoid b) => [Either a b] -> (a, b)
```

Example usage:

```
ghci> epart [Left (Sum 3), Right [1,2,3], Left (Sum 5), Right [4,5]]
(Sum {getSum = 8},[1,2,3,4,5])
```

Task 7

1. Create a module named `HW1.T7`.
2. Define the following data type and a lawful `Semigroup` instance for it:

```
data ListPlus a = a :+ ListPlus a | Last a
infixr 5 :+
```

3. Define the following data type and a lawful `Semigroup` instance for it:

```
data Inclusive a b = This a | That b | Both a b
```

The instance must not discard any values:

```
This i  <>  This j  =  This (i <> j)   -- OK
This i  <>  This _  =  This i          -- This is not the Semigroup you're looking for.
```

4. Define the following data type:

```
newtype DotString = DS String
```

Implement a `Semigroup` instance for it, such that the strings are concatenated with a dot:

```
ghci> DS "person" <> DS "address" <> DS "city"
DS "person.address.city"
```

Implement a `Monoid` instance for it using `DS ""` as the identity element. Make sure that the laws hold:

```
mempty <> a  ≡  a
a <> mempty  ≡  a
```

5. Define the following data type:

```
newtype Fun a = F (a -> a)
```

Implement lawful `Semigroup` and `Monoid` instances for it.

Homework #2

Task 1

1. Create a module named `HW2.T1` and define the following data types in it:

```
data Option a = None | Some a
```

```
data Pair a = P a a
```

```
data Quad a = Q a a a a
```

```
data Annotated e a = a :# e
infix 0 :#
```

```
data Except e a = Error e | Success a
```

```
data Prioritised a = Low a | Medium a | High a
```

```
data Stream a = a :> Stream a
infixr 5 :>
```

```
data List a = Nil | a :. List a
infixr 5 :.
```

```
data Fun i a = F (i -> a)
```

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

2. For each of those types, implement a function of the following form:

```
mapF :: (a -> b) -> (F a -> F b)
```

That is, implement the following functions:

```
mapOption      :: (a -> b) -> (Option a -> Option b)
mapPair        :: (a -> b) -> (Pair a -> Pair b)
mapQuad        :: (a -> b) -> (Quad a -> Quad b)
mapAnnotated   :: (a -> b) -> (Annotated e a -> Annotated e b)
mapExcept      :: (a -> b) -> (Except e a -> Except e b)
mapPrioritised :: (a -> b) -> (Prioritised a -> Prioritised b)
mapStream      :: (a -> b) -> (Stream a -> Stream b)
mapList        :: (a -> b) -> (List a -> List b)
mapFun         :: (a -> b) -> (Fun i a -> Fun i b)
mapTree        :: (a -> b) -> (Tree a -> Tree b)
```

These functions must modify only the elements and preserve the overall structure (e.g. do not reverse the list, do not rebalance the tree, do not swap the pair).

This property is witnessed by the following laws:

```
          mapF id  ≡  id
mapF f ∘ mapF g   ≡  mapF (f ∘ g)
```

You must implement these functions by hand, without using any predefined functions (not even from `Prelude`) or deriving.

Task 2

Create a module named `HW2.T2`. For each type from the first task except `Tree`, implement functions of the following form:

```
distF :: (F a, F b) -> F (a, b)
wrapF :: a -> F a
```

That is, implement the following functions:

```
distOption      :: (Option a, Option b) -> Option (a, b)
distPair        :: (Pair a, Pair b) -> Pair (a, b)
distQuad        :: (Quad a, Quad b) -> Quad (a, b)
distAnnotated   :: Semigroup e => (Annotated e a, Annotated e b) -> Annotated e (a, b)
distExcept      :: (Except e a, Except e b) -> Except e (a, b)
distPrioritised :: (Prioritised a, Prioritised b) -> Prioritised (a, b)
distStream      :: (Stream a, Stream b) -> Stream (a, b)
distList        :: (List a, List b) -> List (a, b)
distFun         :: (Fun i a, Fun i b) -> Fun i (a, b)
```

```
wrapOption      :: a -> Option a
wrapPair        :: a -> Pair a
wrapQuad        :: a -> Quad a
wrapAnnotated   :: Monoid e => a -> Annotated e a
wrapExcept      :: a -> Except e a
wrapPrioritised :: a -> Prioritised a
wrapStream      :: a -> Stream a
wrapList        :: a -> List a
wrapFun         :: a -> Fun i a
```

The following laws must hold:

- Homomorphism:

  ```
  distF (wrapF a, wrapF b)  ≅  wrapF (a, b)
  ```

- Associativity:

  ```
  distF (p, distF (q, r))   ≅  distF (distF (p, q), r)
  ```

- Left and right identity:

  ```
  distF (wrapF (), q)  ≅  q
  distF (p, wrapF ())  ≅  p
  ```

In the laws stated above, we reason up to the following isomorphisms:

```
((a, b), c)  ≅  (a, (b, c))  -- for associativity
    ((), b)  ≅  b            -- for left identity
    (a, ())  ≅  a            -- for right identity
```

There is more than one way to implement some of these functions. In addition to the laws, take the following expectations into account:

- `distPrioritised` must pick the higher priority out of the two.
- `distList` must associate each element of the first list with each element of the second list (i.e. the resulting list is of length `n × m`).

You must implement these functions by hand, using only:

- data types that you defined in `HW2.T1`
- `(<>)` and `mempty` for `Annotated`

Task 3

Create a module named `HW2.T3`. For `Option`, `Except`, `Annotated`, `List`, and `Fun` define a function of the following form:

```
joinF :: F (F a) -> F a
```

That is, implement the following functions:

```
joinOption    :: Option (Option a) -> Option a
joinExcept    :: Except e (Except e a) -> Except e a
joinAnnotated :: Semigroup e => Annotated e (Annotated e a) -> Annotated e a
joinList      :: List (List a) -> List a
joinFun       :: Fun i (Fun i a) -> Fun i a
```

The following laws must hold:

- Associativity:

  ```
  joinF (mapF joinF m)  ≡  joinF (joinF m)
  ```

  In other words, given `F (F (F a))`, it does not matter whether we join the outer layers or the inner layers first.

- Left and right identity:

  ```
  joinF      (wrapF m)  ≡  m
  joinF (mapF wrapF m)  ≡  m
  ```

  In other words, layers created by `wrapF` are identity elements to `joinF`.

  Given `F a`, you can add layers outside/inside to get `F (F a)`, but `joinF` flattens it back into `F a` without any other changes to the structure.

Furthermore, `joinF` is strictly more powerful than `distF` and can be used to define it:

```
distF (p, q) = joinF (mapF (\a -> mapF (\b -> (a, b)) q) p)
```

At the same time, this is only one of the possible `distF` definitions (e.g. `List` admits at least two lawful `distF`). It is common in Haskell to expect `distF` and `joinF` to agree in behavior, so the above equation must hold. (Do not redefine `distF` using `joinF`, though: it would be correct but not the point of the exercise).

Task 4

1. Create a module named `HW2.T4` and define the following data type in it:

   ```
   data State s a = S { runS :: s -> Annotated s a }
   ```

2. Implement the following functions:

   ```
   mapState :: (a -> b) -> State s a -> State s b
   wrapState :: a -> State s a
   joinState :: State s (State s a) -> State s a
   modifyState :: (s -> s) -> State s ()
   ```

   Using those functions, define `Functor`, `Applicative`, and `Monad` instances:

   ```
   instance Functor (State s) where
     fmap = mapState

   instance Applicative (State s) where
     pure = wrapState
     p <*> q = Control.Monad.ap p q

   instance Monad (State s) where
     m >>= f = joinState (fmap f m)
   ```

   These instances will enable the use of `do`-notation with `State`.

   The semantics of `State` are such that the following holds:

   ```
   runS (do modifyState f; modifyState g; return a) x
     ≡
   a :# g (f x)
   ```

   In other words, we execute stateful actions left-to-right, passing the state from one to another.

3. Define the following data type, representing a small language:

   ```
   data Prim a =
       Add a a      -- (+)
     | Sub a a      -- (-)
     | Mul a a      -- (*)
     | Div a a      -- (/)
     | Abs a        -- abs
   ```

```
  | Sgn a        -- signum

data Expr = Val Double | Op (Prim Expr)
```

For notational convenience, define the following instances:

```
instance Num Expr where
  x + y = Op (Add x y)
  x * y = Op (Mul x y)
  ...
  fromInteger x = Val (fromInteger x)

instance Fractional Expr where
  ...
```

So that `(3.14 + 1.618 :: Expr)` produces this syntax tree:

```
Op (Add (Val 3.14) (Val 1.618))
```

4. Using `do`-notation for `State` and combinators we defined for it (`pure`, `modifyState`), define the evaluation function:

   ```
   eval :: Expr -> State [Prim Double] Double
   ```

   In addition to the final result of evaluating an expression, it accumulates a trace of all individual operations:

   ```
   runS (eval (2 + 3 * 5 - 7)) []
     ≡
   10 :# [Sub 17 7, Add 2 15, Mul 3 5]
   ```

   The head of the list is the last operation, this way adding another operation to the trace is O(1).

   You can use the trace to observe the evaluation order. Consider this expression:

   ```
   (a * b) + (x * y)
   ```

   In `eval`, we choose to evaluate `(a * b)` first and `(x * y)` second, even though the opposite is also possible and would not affect the final result of the computation.

Task 5

1. Create a module named `HW2.T5` and define the following data type in it:

   ```
   data ExceptState e s a = ES { runES :: s -> Except e (Annotated s a) }
   ```

   This type is a combination of `Except` and `State`, allowing a stateful computation to abort with an error.

2. Implement the following functions:

   ```
   mapExceptState :: (a -> b) -> ExceptState e s a -> ExceptState e s b
   wrapExceptState :: a -> ExceptState e s a
   joinExceptState :: ExceptState e s (ExceptState e s a) -> ExceptState e s a
   modifyExceptState :: (s -> s) -> ExceptState e s ()
   throwExceptState :: e -> ExceptState e s a
   ```

   Using those functions, define `Functor`, `Applicative`, and `Monad` instances.

3. Using `do`-notation for `ExceptState` and combinators we defined for it (`pure`, `modifyExceptState`, `throwExceptState`), define the evaluation function:

   ```
   data EvaluationError = DivideByZero
   eval :: Expr -> ExceptState EvaluationError [Prim Double] Double
   ```

   It works just as `eval` from the previous task but aborts the computation if division by zero occurs:

   ```
   runES (eval (2 + 3 * 5 - 7)) []
     ≡
   Success (10 :# [Sub 17 7, Add 2 15, Mul 3 5])
   ```

```
runES (eval (1 / (10 - 5 * 2))) []
  ≡
Error DivideByZero
```

Task 6

1. Create a module named `HW2.T6` and define the following data type in it:

   ```
   data ParseError = ErrorAtPos Natural

   newtype Parser a = P (ExceptState ParseError (Natural, String) a)
     deriving newtype (Functor, Applicative, Monad)
   ```

   Here we use `ExceptState` for an entirely different purpose: to parse data from a string. Our state consists of a `Natural` representing how many characters we have already consumed (for error messages) and the `String` is the remainder of the input.

2. Implement the following function:

   ```
   runP :: Parser a -> String -> Except ParseError a
   ```

3. Let us define a parser that consumes a single character:

   ```
   pChar :: Parser Char
   pChar = P $ ES \(pos, s) ->
     case s of
       []     -> Error (ErrorAtPos pos)
       (c:cs) -> Success (c :# (pos + 1, cs))
   ```

   Study this definition:

   - What happens when the string is empty?
   - How does the parser state change when a character is consumed?

   Write a comment that explains `pChar`.

4. Implement a parser that always fails:

   ```
   parseError :: Parser a
   ```

   Define the following instance:

   ```
   instance Alternative Parser where
     empty = parseError
     (<|>) = ...

   instance MonadPlus Parser   -- No methods.
   ```

   So that `p <|> q` tries to parse the input string using `p`, but in case of failure tries `q`.

   Make sure that the laws hold:

   ```
   empty <|> p  ≡  p
   p <|> empty  ≡  p
   ```

5. Implement a parser that checks that there is no unconsumed input left (i.e. the string in the parser state is empty), and fails otherwise:

   ```
   pEof :: Parser ()
   ```

6. Study the combinators provided by `Control.Applicative` and `Control.Monad`. The following are of particular interest:

   - `msum`
   - `mfilter`
   - `optional`
   - `many`
   - `some`
   - `void`

We can use them to construct more interesting parsers. For instance, here is a parser that accepts only non-empty sequences of uppercase letters:

```
pAbbr :: Parser String
pAbbr = do
  abbr <- some (mfilter Data.Char.isUpper pChar)
  pEof
  pure abbr
```

It can be used as follows:

```
ghci> runP pAbbr "HTML"
Success "HTML"

ghci> runP pAbbr "JavaScript"
Error (ErrorAtPos 1)
```

7. Using parser combinators, define the following function:

   ```
   parseExpr :: String -> Except ParseError Expr
   ```

   It must handle floating-point literals of the form `4.09`, the operators `+ - * /` with the usual precedence (multiplication and division bind tighter than addition and subtraction), and parentheses.

   Example usage:

   - ```
     ghci> parseExpr "3.14 + 1.618 * 2"
     Success (Op (Add (Val 3.14) (Op (Mul (Val 1.618) (Val 2.0)))))
     ```
   - ```
     ghci> parseExpr "2 * (1 + 3)"
     Success (Op (Mul (Val 2.0) (Op (Add (Val 1.0) (Val 3.0)))))
     ```
   - ```
     ghci> parseExpr "24 + Hello"
     Error (ErrorAtPos 3)
     ```

   The implementation must not use the `Read` class, as it implements similar functionality (the exercise is to write your own parsers). At the same time, you are encouraged to use existing `Applicative` and `Monad` combinators, since they are not specific to parsing.

Homework #3

In this homework we will gradually develop a small programming language called Hi.

Project Structure

Create a `.cabal` file with both a library and an executable:

```
library
  exposed-modules:    ...
  build-depends:      ...
  ...

executable hi
  main-is:            Main.hs
  hs-source-dirs:     ...
  build-depends:      ...
  ...
```

In the library component, create the following modules:

```
HW3.Base
HW3.Parser
HW3.Pretty
HW3.Evaluator
```

You are allowed to add more modules to the project, but those are the required ones.

- In `HW3.Base`, define the following data types:
```

```
data HiFun     -- function names (e.g. div, sort, length, ...)
data HiValue   -- values (numbers, booleans, strings, ...)
data HiExpr    -- expressions (literals, function calls, ...)
data HiError   -- evaluation errors (invalid arguments, ...)
```

In each task, we will add constructors to these data types that are needed to implement new language features.

- In `HW3.Parser`, define the following function:

  ```
  parse :: String -> Either (ParseErrorBundle String Void) HiExpr
  ```

  The `ParseErrorBundle` type comes from the `megaparsec` package which we will use to implement our parser.

- In `HW3.Pretty`, define the following function:

  ```
  prettyValue :: HiValue -> Doc AnsiStyle
  ```

  The `Doc` and `AnsiStyle` types come from the `prettyprinter` and `prettyprinter-ansi-terminal` packages respectively. This function renders a value to a document, which in turn can be either printed to the terminal (with color highlighting) or converted to a string.

- In `HW3.Evaluator`, define the following function:

  ```
  eval :: Monad m => HiExpr -> m (Either HiError HiValue)
  ```

  One might wonder why we need the `Monad m` part. Indeed, for arithmetic operations, a simpler type would be sufficient:

  ```
  eval :: HiExpr -> Either HiError HiValue
  ```

  However, the monadic context will come into play later, when we start implementing IO actions (file system access, random number generation, and so on).

The executable component consists just of a single file, `Main.hs`.

The REPL

Using the `haskeline` package, implement a REPL in `Main.hs` that uses `parse`, `eval`, and `prettyValue` defined above. It's going to be just 15–20 lines of code, but you will use it all the time to test your implementation.

Here's an example session that will become possible as soon as we implement arithmetic operations:

```
hi> mul(2, 10)
20

hi> sub(1000, 7)
993

hi> div(3, 5)
0.6
```

Task 1: Numbers and arithmetic

1. Extend the data types in `HW3.Base` to include the following constructors:

   ```
   data HiFun =
     ...
     | HiFunDiv
     | HiFunMul
     | HiFunAdd
     | HiFunSub

   data HiValue =
     ...
     | HiValueNumber Rational
     | HiValueFunction HiFun
   ```

```
data HiExpr =
  ...
  | HiExprValue HiValue
  | HiExprApply HiExpr [HiExpr]

data HiError =
  ...
  | HiErrorInvalidArgument
  | HiErrorInvalidFunction
  | HiErrorArityMismatch
  | HiErrorDivideByZero
```

Numbers are represented using the `Rational` type from `Data.Ratio`.

2. In the parser, add support for the following constructs:

   - Built-in names `div`, `mul`, `add`, and `sub`, that are parsed into the corresponding `HiFun` constructors (and then wrapped in `HiValueFunction`).
   - Numeric literals, such as `2`, `3.14`, `-1.618`, or `1.2e5`, that are parsed into `HiValueNumber` (tip: use `Text.Megaparsec.Char.Lexer.scientific`).
   - Function application `f(a, b, c, ...)` that is parsed into `HiExprApply`.

   For example, the expression `div(add(10, 15.1), 3)` is represented by the following syntax tree:

```
HiExprApply (HiExprValue (HiValueFunction HiFunDiv))
  [
    HiExprApply (HiExprValue (HiValueFunction HiFunAdd))
      [
        HiExprValue (HiValueNumber (10 % 1)),
        HiExprValue (HiValueNumber (151 % 10))
      ],
    HiExprValue (HiValueNumber (3 % 1))
  ]
```

3. In the evaluator, implement the arithmetic operations:

   - `add(500, 12)` evaluates to `512` (addition)
   - `sub(10, 100)` evaluates to `-90` (subtraction)
   - `mul(23, 768)` evaluates to `17664` (multiplication)
   - `div(57, 190)` evaluates to `0.3` (division)

   Nested function applications are allowed:

   - `div(add(mul(2, 5), 1), sub(11,6))` evaluates to `2.2`

   The following errors must be returned as `HiError`:

   - `HiErrorArityMismatch`: functions called with an incorrect amount of arguments, e.g. `sub(1)` or `sub(1, 2, 3)`.
   - `HiErrorDivideByZero`: the `div` function is called with `0` as its second argument, e.g. `div(1, 0)` or `div(1, sub(5, 5))`.
   - `HiErrorInvalidFunction`: numbers are used in function positions, e.g. `15(2)`.
   - `HiErrorInvalidArgument`: functions are used in numeric positions, e.g. `sub(10, add)`.

   You are advised to use the `ExceptT` monad transformer to propagate `HiError` through the evaluator.

4. In the pretty-printer, define the following special cases for rendering numbers:

   - integers: `42`, `-8`, `15`
   - finite decimal fractions: `3.14`, `-8.15`, `77.01`
   - fractions: `1/3`, `-1/7`, `3/11`
   - mixed fractions: `5 + 1/3`, `-10 - 1/7`, `24 + 3/11`

   You will find these functions useful:

- `quotRem` from `Prelude`
- `fromRationalRepetendUnlimited` from the `scientific` package

The following session in the REPL should be possible if you have implemented all of the above correctly:

```
hi> 100
100

hi> -15
-15

hi> add(100, -15)
85

hi> add(3, div(14, 100))
3.14

hi> div(10, 3)
3 + 1/3

hi> sub(mul(201, 11), 0.33)
2210.67
```

Task 2: Booleans and comparison

1. Extend the data types in `HW3.Base` to include the following constructors:

```
data HiFun =
  ...
  | HiFunNot
  | HiFunAnd
  | HiFunOr
  | HiFunLessThan
  | HiFunGreaterThan
  | HiFunEquals
  | HiFunNotLessThan
  | HiFunNotGreaterThan
  | HiFunNotEquals
  | HiFunIf

data HiValue =
  ...
  | HiValueBool Bool
```

2. In the parser, add support for the following constructs:

   - Built-in names `not`, `and`, `or`, `less-than`, `greater-than`, `equals`, `not-less-than`, `not-greater-than`, `not-equals`, `if`, that are parsed into the corresponding `HiFun` constructors.

   - Built-in names `true` and `false` that are parsed into `HiValueBool`.

3. In the evaluator, implement the new operations.

   Boolean algebra:

   - `not(true)` evaluates to `false` (negation)
   - `and(true, false)` evaluates to `false` (conjunction)
   - `or(true, false)` evaluates to `true` (disjunction)

   Equality checking:

   - `equals(10, 10)` evaluates to `true`
   - `equals(false, false)` evaluates to `true`
   - `equals(3, 10)` evaluates to `false`
   - `equals(1, true)` evaluates to `false` (no implicit cast)

   Comparisons:

   - `less-than(3, 10)` evaluates to `true`
   - `less-than(false, true)` evaluates to `true`

- `less-than(false, 0)` evaluates to `true` (`Bool` is less than `Number`)

Complements:

- for all `A B`, `greater-than(A, B) ≡ less-than(B, A)` holds
- for all `A B`, `not-equals(A, B) ≡ not(equals(A, B))` holds
- for all `A`, `B`, `not-less-than(A, B) ≡ not(less-than(A, B))` holds
- for all `A`, `B`, `not-greater-than(A, B) ≡ not(greater-than(A, B))` holds

Branching:

- for all `A B`, `if(true, A, B) ≡ A` holds
- for all `A B`, `if(false, A, B) ≡ B` holds

The following session in the REPL should be possible:

```
hi> false
false

hi> equals(add(2, 2), 4)
true

hi> less-than(mul(999, 99), 10000)
false

hi> if(greater-than(div(2, 5), div(3, 7)), 1, -1)
-1

hi> and(less-than(0, 1), less-than(1, 0))
false
```

Note also that functions are values:

```
hi> if(true, add, mul)
add

hi> if(true, add, mul)(10, 10)
20

hi> if(false, add, mul)(10, 10)
100
```

Functions can also be tested for equality:

```
hi> equals(add, add)
true

hi> equals(add, mul)
false
```

The check is trivial: a function is equal only to itself.

Ordering of function symbols is implementation-defined, that is, it's up to you whether `less-than(add, mul)` or `greater-than(add, mul)`.

Task 3: Operators

In the parser, add support for infix operators. The precedence and associativity are the same as in Haskell.

For all `A B`:

- `A / B` parses to `div(A, B)`
- `A * B` parses to `mul(A, B)`
- `A + B` parses to `add(A, B)`
- `A - B` parses to `sub(A, B)`
- `A < B` parses to `less-than(A, B)`
- `A > B` parses to `greater-than(A, B)`
- `A >= B` parses to `not-less-than(A, B)`
- `A <= B` parses to `not-greater-than(A, B)`
- `A == B` parses to `equals(A, B)`

- `A /= B` parses to `not-equals(A, B)`
- `A && B` parses to `and(A, B)`
- `A || B` parses to `or(A, B)`

Tip: use `makeExprParser` from the `parser-combinators` package.

The following session in the REPL should be possible:

```
hi> 2 + 2
4

hi> 2 + 2 * 3
8

hi> (2 + 2) * 3
12

hi> 2 + 2 * 3 == (2 + 2) * 3
false

hi> 10 == 2*5 && 143 == 11*13
true
```

Task 4: Strings and slices

1. Extend the data types in `HW3.Base` to include the following constructors:

   ```
   data HiFun =
     ...
     | HiFunLength
     | HiFunToUpper
     | HiFunToLower
     | HiFunReverse
     | HiFunTrim

   data HiValue =
     ...
     | HiValueNull
     | HiValueString Text
   ```

   Strings are represented using the `Text` type from the `text` package.

2. In the parser, add support for the following constructs:

   - Bulit-in names `length`, `to-upper`, `to-lower`, `reverse`, `trim`, that are parsed into the corresponding `HiFun` constructors.
   - Built-in name `null` that is parsed into `HiValueNull`.
   - String literals, such as `"hello"`, `"42"`, or `"header\nfooter"`, that are parsed into `HiValueString` (tip: use `Text.Megaparsec.Char.Lexer.charLiteral`).

3. In the evaluator, implement the new operations:

   - `length("Hello World")` evaluates to `11`
   - `to-upper("Hello World")` evaluates to `"HELLO WORLD"`
   - `to-lower("Hello World")` evaluates to `"hello world"`
   - `reverse("stressed")` evaluates to `"desserts"`
   - `trim(" Hello World ")` evaluates to `"Hello World"`

   Then overload existing operations to work on strings:

   - `"Hello" + "World"` evaluates to `"HelloWorld"`
   - `"Cat" * 5` evaluates to `"CatCatCatCatCat"` (tip: use `stimes`)
   - `"/home/user" / "hi"` evaluates to `"/home/user/hi"`

   When a string is used as a function of one argument, perform a lookup:

   - `"Hello World"(0)` evaluates to `"H"`
   - `"Hello World"(7)` evaluates to `"o"`

Out-of-bounds indexing returns `null`:

- `"Hello World"(-1)` evaluates to `null`
- `"Hello World"(99)` evaluates to `null`

When a string is used as a function of two arguments, take a slice:

- `"Hello World"(0, 5)` evaluates to `"Hello"`
- `"Hello World"(2, 4)` evaluates to `"ll"`

4. (Advanced) When a slice index is negative, implement the Python semantics of indexing from the end of the string:

   - `"Hello World"(0, -4)` evaluates to `"Hello W"`
   - `"Hello World"(-4, -1)` evaluates to `"orl"`

   When a slice index is `null`, treat it as the start/end of the string:

   - `"Hello, World"(2, null)` evaluates to `"llo, World"`
   - `"Hello, World"(null, 5)` evaluates to `"Hello"`

The following session in the REPL should be possible:

```
hi> to-upper("what a nice language")(7, 11)
"NICE"

hi> "Hello" == "World"
false

hi> length("Hello" + "World")
10

hi> length("hehe" * 5) / 3
6 + 2/3
```

Task 5: Lists and folds

1. Extend the data types in `HW3.Base` to include the following constructors:

   ```
   data HiFun =
     ...
     | HiFunList
     | HiFunRange
     | HiFunFold

   data HiValue =
     ...
     | HiValueList (Seq HiValue)
   ```

   Lists are represented using the `Seq` type from the `containers` package.

2. In the parser, add support for the following constructs:

   - Built-in names `list`, `range`, `fold`, that are parsed into the corresponding `HiFun` constructors.

   - List literals, written as `[A, B, C, ...]`, that are parsed into function application `list(A, B, C, ...)`.

3. In the evaluator, implement the new operations:

   - `list(1, 2, 3)` constructs `HiValueList` containing `1`, `2`, `3`
   - `range(5, 10.3)` evaluates to `[5, 6, 7, 8, 9, 10]`
   - `fold(add, [11, 22, 33])` evaluates to `66`
   - `fold(mul, [11, 22, 33])` evaluates to `7986`
   - `fold(div, [11, 22, 33])` evaluates to `1/66` (left fold)

   Then overload existing operations to work on lists:

   - `length([1, true, "Hello"])` evaluates to `3`

- `reverse([1, true, "Hello"])` evaluates to `["Hello", true, 1]`
- `[1, 2] + [3, 4, 5]` evaluates to `[1, 2, 3, 4, 5]`
- `[0, "x"] * 3` evaluates to `[0, "x", 0, "x", 0, "x"]` (tip: use `stimes`)

When a list is used as a function, perform indexing/slicing:

- `["hello", true, "world"](1)` evaluates to `true`
- `["hello", true, "world"](1,3)` evaluates to `[true, "world"]`

The following session in the REPL should be possible:

```
hi> list(1, 2, 3, 4, 5)
[ 1, 2, 3, 4, 5 ]

hi> fold(add, [2, 5] * 3)
21

hi> fold(mul, range(1, 10))
3628800

hi> [0, true, false, "hello", "world"](2, 4)
[ false, "hello" ]

hi> reverse(range(0.5, 70/8))
[ 8.5, 7.5, 6.5, 5.5, 4.5, 3.5, 2.5, 1.5, 0.5 ]
```

Task 6: Bytes and serialisation

Lists of bytes (numbers from `0` to `255`) can be represented and processed more efficiently. Let us introduce a new value type for them.

1. Extend the data types in `HW3.Base` to include the following constructors:

   ```
   data HiFun =
     ...
     | HiFunPackBytes
     | HiFunUnpackBytes
     | HiFunEncodeUtf8
     | HiFunDecodeUtf8
     | HiFunZip
     | HiFunUnzip
     | HiFunSerialise
     | HiFunDeserialise

   data HiValue =
     ...
     | HiValueBytes ByteString
   ```

   Bytes are represented using the strict `ByteString` type from the `bytestring` package.

2. In the parser, add support for the following constructs:

   - Built-in names `pack-bytes`, `unpack-bytes`, `zip`, `unzip`, `encode-utf8`, `decode-utf8`, `serialise`, and `deserialise`, that are parsed into the corresponding `HiFun` constructors.

   - Byte array literals, such as `[# 01 3f ec #]` that are parsed into `HiValueBytes`. Each element is a two-digit hexadecimal number.

3. In the evaluator, implement the new operations:

   - `pack-bytes([ 3, 255, 158, 32 ])` evaluates to `[# 03 ff 9e 20 #]`
   - `unpack-bytes([# 10 20 30 #])` evaluates to `[16, 32, 48]`
   - `encode-utf8("Hello!")` evaluates to `[# 48 65 6c 6c 6f 21 #]`
   - `decode-utf8([# 48 65 6c 6c 6f #])` evaluates to `"Hello"`
   - `decode-utf8([# c3 28 #])` evaluates to `null` (invalid UTF-8 byte sequence)
   - `zip` compresses the bytes using the `zlib` package (specify `bestCompression`)
   - `serialise` turns any value into bytes using the `serialise` package
   - for all `A`, `unzip(zip(A)) ≡ A` holds
   - for all `A`, `deserialise(serialise(A)) ≡ A` holds

Then overload existing operations to work on bytes:

- `[# 00 ff #] + [# 01 e3 #]` evaluates to `[# 00 ff 01 e3 #]`
- `[# 00 ff #] * 3` evaluates to `[# 00 ff 00 ff 00 ff #]` (tip: use `stimes`)

When bytes are used as a function, perform indexing/slicing as with strings and lists:

- `[# 00 ff 01 e3 #](1)` evaluates to `255`
- `[# 00 ff 01 e3 #](1,3)` evaluates to `[# ff 01 #]`

The following session in the REPL should be possible:

```
hi> pack-bytes(range(30, 40))
[# 1e 1f 20 21 22 23 24 25 26 27 28 #]

hi> zip(encode-utf8("Hello, World!" * 1000))
[# 78 da ed c7 31 0d 00 20 0c 00 30 2b f0 23 64 0e 30 00 df 92 25 f3 7f a0 82 af
   fd 1a 37 b3 d6 d8 d5 79 66 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88 88
   88 88 88 88 88 88 88 88 fc c9 03 ca 0f 3b 28 #]

hi> decode-utf8([# 68 69 #] * 5)
"hihihihihi"

hi> unzip([# 78 da 63 64 62 06 00 00 0d 00 07 #])
[# 01 02 03 #]
```

Task 7: File I/O

In this task we extend the language with I/O capabilities. We consider it to be the most important part of the homework and it is graded with extra points.

Let us start by creating a new type in `HW3.Base` that encodes the available actions:

```
data HiAction =
    HiActionRead  FilePath
  | HiActionWrite FilePath ByteString
  | HiActionMkDir FilePath
  | HiActionChDir FilePath
  | HiActionCwd
```

Now recall that the type of our `eval` function is as follows:

```
eval :: Monad m => HiExpr -> m (Either HiError HiValue)
```

We could just require `m` to be `IO` in order to execute the actions, but that would be bad design, as it would make it impossible to do evaluation in a pure, deterministic context (e.g. for tests). Instead, let us create a new class in `HW3.Base`:

```
class Monad m => HiMonad m where
  runAction :: HiAction -> m HiValue
```

One could imagine at least a few possible instances of this class:

1. `IO`, where those actions could interact with the actual file system
2. `Identity`, where the actions do nothing and return `null`
3. `State FS`, where `FS` is a pure and deterministic in-memory simulation of the file system
4. `ReaderT Permissions IO`, which can be more secure than `IO` by controlling whether the program has read-only or read-write access

While it would be useful to implement all of those, we shall limit ourselves to the last one to avoid making the task unnecessarily tedious.

In a new module `HW3.Action`, declare the following:

```
data HiPermission =
    AllowRead
  | AllowWrite
```

```
data PermissionException =
  PermissionRequired HiPermission

instance Exception PermissionException

newtype HIO a = HIO { runHIO :: Set HiPermission -> IO a }
```

Finally, let us change the type of `eval` as follows:

```
- eval ::   Monad m => HiExpr -> m (Either HiError HiValue)
+ eval :: HiMonad m => HiExpr -> m (Either HiError HiValue)
```

With those preliminaries out of the way, we can start integrating the actions into the rest of the language.

1. Extend the data types in `HW3.Base` to include the following constructors:

   ```
   data HiFun =
     ...
     | HiFunRead
     | HiFunWrite
     | HiFunMkDir
     | HiFunChDir

   data HiValue =
     ...
     | HiValueAction HiAction

   data HiExpr =
     ...
     | HiExprRun HiExpr
   ```

2. In the parser, add support for the following constructs:

   - Built-in names `read`, `write`, `mkdir`, `cd`, that are parsed into the corresponding `HiFun` constructors.
   - Built-in name `cwd` that is parsed into `HiValueAction HiActionCwd`.
   - `E!` notation that is parsed into `HiExprRun`, e.g. `read("hello.txt")!`, `mkdir("projects")!`, or `cwd!`.

3. In the evaluator, implement the new functions to return the corresponding actions:

   - `read("hi.txt")` evaluates to `read("hi.txt")`. While visually the same, internally the first one is `HiExprApply` and the second one is `HiValueAction`.
   - `write("hi.txt", "Hi!")` evaluates to `write("hi.txt", [# 48 69 21 #])`
   - `mkdir("dir")` evaluates to `mkdir("dir")`
   - `cd("dir")` evaluates to `cd("dir")`

   Then implement the `HiExprRun` construct, which should execute the action using `runAction` that we defined earlier.

4. Define the `HiMonad HIO` instance, such that:

   - `cwd!` returns the current working directory
   - `cd("mydir")!` changes the current working directory to `mydir`
   - `read("myfile")!` returns the contents of `myfile` (use `HiValueString` if the contents are valid UTF-8 and `HiValueBytes` otherwise)
   - `read("mydir")!` returns the directory listing of `mydir`
   - `write("myfile", "Hello")!` writes `"Hello"` to `myfile`
   - `mkdir("mydir")!` creates a new directory `mydir`

   Use the `directory` package to implement all of the above.

5. Implement permission control in `HiMonad HIO`, so that actions throw `PermissionException` (using `throwIO`) unless they are allowed.

   - `AllowRead` enables `cwd`, `cd`, `read`

- `AllowWrite` enables `write`, `mkdir`

The following session in the REPL should be possible:

```
hi> mkdir("tmp")!
null

hi> read("tmp")!
[]

hi> mkdir("tmp/a")!
null

hi> mkdir("tmp/b")!
null

hi> read("tmp")!
[ "a", "b" ]

hi> write("tmp/hi.txt", "Hello")!
null

hi> cd("tmp")!
null

hi> read("hi.txt")!
"Hello"
```

Note that actions are just values and only `!` forces their execution:

```
hi> read
read

hi> read("hi.txt")
read("hi.txt")

hi> read("hi.txt")!
"Hello"
```

Task 8: Date and time

1. Extend the data types in `HW3.Base` to include the following constructors:

   ```
   data HiFun =
     ...
     | HiFunParseTime

   data HiValue =
     ...
     | HiValueTime UTCTime

   data HiAction =
     ...
     | HiActionNow
   ```

   Time is represented using the `UTCTime` type from the `time` package.

   Extend the data types in `HW3.Action` to include the following constructors:

   ```
   data HiPermission =
     ...
     | AllowTime
   ```

2. In the parser, add support for the following constructs:

   - Built-in name `parse-time` that is parsed into the corresponding `HiFun` constructor.
   - Built-in name `now` that is parsed into the corresponding `HiAction` constructor.

3. In the evaluator, implement `parse-time` using `readMaybe` to parse a `HiValueString` into a `HiValueTime`, or `HiValueNull` in case of failure.

4. In the `HiMonad HIO` instance, implement the `HiActionNow` to return the current system time. It requires the `AllowTime` permission.

5. In the evaluator, overload existing operations to work on time:

   - `parse-time("2021-12-15 00:00:00 UTC") + 1000` evaluates to `parse-time("2021-12-15 00:16:40 UTC")` (use `addUTCTime`)
   - `parse-time("2021-12-15 00:37:51.000890793 UTC") - parse-time("2021-12-15 00:37:47.649047038 UTC")` evaluates to `3.351843755` (use `diffUTCTime`)

The following session in the REPL should be possible:

```
hi> now!
parse-time("2021-12-15 00:42:33.02949461 UTC")

hi> parse-time("2021-01-01 00:00:00 UTC") + 365 * 24 * 60 * 60
parse-time("2022-01-01 00:00:00 UTC")
```

Task 9: Random numbers

1. Extend the data types in `HW3.Base` to include the following constructors:

```
data HiFun =
  ...
  | HiFunRand

data HiAction =
  ...
  | HiActionRand Int Int
```

2. In the parser, add support for the built-in name `rand` that is parsed into the corresponding `HiFun` constructor.

3. In the evaluator, implement the new function:

   - `rand(0, 10)` evaluates to `rand(0, 10)`. While visually the same, internally the first one is `HiExprApply` and the second one is `HiValueAction`.

4. Extend the `HiMonad HIO` instance, so that:

   - `rand(0, 5)!` evaluates to `0`, `1`, `2`, `3`, `4`, or `5`
   - the distribution of random numbers is uniform

   Tip: use the `random` package.

The following session in the REPL should be possible:

```
hi> rand
rand

hi> rand(0, 10)
rand( 0, 10 )

hi> rand(0, 10)!
8

hi> rand(0, 10)!
3
```

Task 10: Short-circuit evaluation

1. Extend the data types in `HW3.Base` to include the following constructors:

```
data HiFun =
  ...
  | HiFunEcho

data HiAction =
  ...
  | HiActionEcho Text
```

2. In the parser, add support for the built-in name `echo` that is parsed into the corresponding `HiFun` constructor.

3. In the evaluator, implement the new function:

   - `echo("Hello")` evaluates to `echo("Hello")`. While visually the same, internally the first one is `HiExprApply` and the second one is `HiValueAction`.

4. Extend the `HiMonad HIO` instance, so that `echo("Hello")!` prints `Hello` followed by a newline to `stdout`. It requires the `AllowWrite` permission.

5. In the evaluator, ensure that `if(true, A, B)` does not evaluate `B`, and `if(false, A, B)` does not evaluate `A`.

   Then generalise `A && B` as follows:

   - if `A` is `false` or `null`, return `A` without evaluating `B`
   - otherwise, evaluate and return `B`

   Generalise `A || B` as follows:

   - if `A` is `false` or `null`, evaluate and return `B`
   - otherwise, return `A` without evaluating `B`

The following session in the REPL should be possible:

```
hi> echo
echo

hi> echo("Hello")
echo("Hello")

hi> echo("Hello")!
Hello
null

hi> "Hello"(0) || "Z"
"H"

hi> "Hello"(99) || "Z"
"Z"

hi> if(2 == 2, echo("OK")!, echo("WTF")!)
OK
null

hi> true || echo("Don't do this")!
true

hi> false && echo("Don't do this")!
false

hi> [# 00 ff #] && echo("Just do it")!
Just do it
null
```

Task 11: Dictionaries

1. Extend the data types in `HW3.Base` to include the following constructors:

```
data HiFun =
  ...
  | HiFunCount
  | HiFunKeys
  | HiFunValues
  | HiFunInvert

data HiValue =
  ...
  | HiValueDict (Map HiValue HiValue)

data HiExpr =
```

```
...
| HiExprDict [(HiExpr, HiExpr)]
```

Dictionaries are represented using the `Map` type from the `containers` package.

2. In the parser, add support for the following constructs:
   - Built-in names `count`, `keys`, `values`, `invert`, that are parsed into the corresponding `HiFun` constructors.
   - Dictionary literals, written as `{ I: A, J: B, K: C }`, for example:
     - `{ "width": 120, "height": 80 }`
     - `{ 1: true, 3: true, 4: false }`
   - Dot access, written as `E.fld`, that is parsed into function application `E("fld")`. For example, the following holds:

     ```
     { "width": 120, "height": 80 }.width
       ≡
     { "width": 120, "height": 80 }("width")
     ```

3. In the evaluator, implement the new operations:
   - `{ "width": 120, "height": 80 }("width")` evaluates to `120`
   - `keys({ "width": 120, "height": 80 })` evaluates to `["height", "width"]` (sorted)
   - `values({ "width": 120, "height": 80 })` evaluates to `[80, 120]` (sorted by key)
   - `count("XXXOX")` evaluates to `{ "O": 1, "X": 4 }`
   - `count([# 58 58 58 4f 58 #])` evaluates to `{ 79: 1, 88: 4 }`
   - `count([true, true, false, true])` evaluates to `{ false: 1, true: 3 }`
   - `invert({ "x": 1, "y" : 2, "z": 1 })` evaluates to `{ 1: [ "z", "x" ], 2: ["y"] }`

The following session in the REPL should be possible:

```
hi> count("Hello World").o
2

hi> invert(count("big blue bag"))
{ 1: [ "u", "l", "i", "e", "a" ], 2: [ "g", " " ], 3: ["b"] }

hi> fold(add, values(count("Hello, World!")))
13
```