Hymn Documentation

Release 0.2

Philip Xu

December 15, 2015

CONTENTS

Contents:

CHAPTER

ONE

HY MONAD NOTATION - A MONAD LIBRARY FOR HY

1.1 Introduction

Hymn is a monad library for Hy/Python, with do notation for monad comprehension.

Code are better than words.

The continuation monad

```
=> (require hymn.dsl)
=> (import [hymn.types.continuation [cont-m call-cc]])
=> ;; computations in continuation passing style
=> (defn double [x] (cont-m.unit (* x 2)))
=> (def length (cont-m.monadic len))
=> ;; chain with bind
=> (.run (>> (cont-m.unit [1 2 3]) length double))
6
=> (defn square [n] (call-cc (fn [k] (k (** n 2)))))
=> (.run (square 12))
144
=> (.run (square 12) inc)
145
=> (.run (square 12) str)
'144'
=> (.run (do-monad [sqr (square 42)] (.format "answer^2 = {}" sqr)))
'answer^2 = 1764'
```

The either monad

```
=> (require hymn.dsl)
=> (import [hymn.types.either [Left Right either failsafe]])
=> ;; do notation with either monad
=> (do-monad [a (Right 1) b (Right 2)] (/ a b))
Right(0.5)
=> (do-monad [a (Right 1) b (Left 'nan)] (/ a b))
Left(nan)
=> ;; failsafe is a function decorator that wraps return value into either
=> (def safe-div (failsafe /))
=> ;; returns Right if nothing wrong
=> (safe-div 4 2)
Right(2.0)
=> ;; returns Left when bad thing happened, like exception being thrown
=> (safe-div 1 0)
Left(ZeroDivisionError('division by zero',))
=> ;; function either tests the value and call functions accordingly
=> (either print inc (safe-div 4 2))
3.0
=> (either print inc (safe-div 1 0))
division by zero
```

The identity monad

```
=> (require hymn.dsl)
=> (import [hymn.types.identity [identity-m]])
=> ;; do notation with identity monad is like let binding
=> (do-monad [a (identity-m 1) b (identity-m 2)] (+ a b))
Identity(3)
```

The list monad

```
=> (require hymn.dsl)
=> (import [hymn.types.list [list-m]])
=> ;; use list-m contructor to turn sequence into list monad
=> (def xs (list-m (range 2)))
=> (def ys (list-m (range 3)))
=> ;; do notation with list monad is list comprehension
=> (list (do-monad [x xs y ys :when (not (zero? y))] (/ x y)) )
[0.0, 0.0, 1.0, 0.5]
=> ;; * is the reader macro for list-m
=> (list (do-monad [x #*(range 2) y #*(range 3) :when (not (zero? y))] (/ x y)) )
[0.0, 0.0, 1.0, 0.5]
```

The maybe monad

```
=> (require hymn.dsl)
=> (import [hymn.types.maybe [Just Nothing maybe]])
=> ;; do notation with maybe monad
=> (do-monad [a (Just 1) b (Just 1)] (/ a b))
Just(1.0)
=> ;; Nothing yields Nothing
=> (do-monad [a Nothing b (Just 1)] (/ a b))
Nothing
=> ;; maybe is a function decorator the wraps return value into maybe
=> ;; a safe-div with maybe monad
=> (def safe-div (maybe /))
=> (safe-div 42 42)
Just(1.0)
=> (safe-div 42 'answer)
Nothing
=> (safe-div 42 0)
Nothing
```

The reader monad

```
=> (require hymn.dsl)
=> (import [hymn.types.reader [lookup]])
=> ;; do notation with reader monad, lookup assumes the environment is subscriptable
=> (def r (do-monad [a (lookup 'a) b (lookup 'b)] (+ a b)))
=> ;; run reader monad r with environment
=> (.run r {'a 1 'b 2})
3
```

The state monad

```
=> (require hymn.dsl)
=> (import [hymn.types.state [lookup set-value]])
=> ;; do notation with state monad, set-value sets the value with key in the state
=> (def s (do-monad [a (lookup 'a) _ (set-value 'b (inc a))] a))
=> ;; run state monad s with initial state
=> (.run s {'a 1})
(, 1 {'a 1 'b 2})
```

The writer monad

```
=> (require hymn.dsl)
=> (import [hymn.types.writer [tell]])
=> ;; do notation with writer monad
=> (do-monad [_ (tell "hello") _ (tell " world")] nil)
StrWriter((None, 'hello world'))
=> ;; int is monoid, too
=> (.execute (do-monad [_ (tell 1) _ (tell 2) _ (tell 3)] nil))
6
```

Operations on monads

```
=> (require hymn.dsl)
=> (import [hymn.operations [lift]])
=> ;; lift promotes function into monad
=> (def m+ (lift +))
=> ;; lifted function can work on any monad
=> ;; on the maybe monad
=> (import [hymn.types.maybe [Just Nothing]])
=> (m+ (Just 1) (Just 2))
Just(3)
=> (m+ (Just 1) Nothing)
Nothing
=> ;; on the either monad
=> (import [hymn.types.either [Left Right]])
=> (m+ (Right 1) (Right 2))
Right(3)
=> (m+ (Left 1) (Right 2))
Left(1)
=> ;; on the list monad
=> (import [hymn.types.list [list-m]])
=> (list (m+ (list-m "ab") (list-m "123")))
['a1', 'a2', 'a3', 'b1', 'b2', 'b3']
=> (list (m+ (list-m "+-") (list-m "123") (list-m "xy")))
['+1x', '+1y', '+2x', '+2y', '+3x', '+3y', '-1x', '-1y', '-2x', '-2y', '-3x', '-3y']
=> ;; can be used as normal function
=> (reduce m+ [(Just 1) (Just 2) (Just 3)])
Just(6)
=> (reduce m+ [(Just 1) Nothing (Just 3)])
Nothing
=> ;; <- is an alias of lookup
=> (import [hymn.types.reader [<-]])
=> ;; ^ is the reader macro for lift
=> (def p (#^print (<- 'message) :end (<- 'end)))
=> (.run p {'message "Hello world" 'end "!\n"})
Hello world!
=> ;; random number - linear congruential generator
=> (import [hymn.types.state [get-state set-state]])
=> (def random (>> get-state (fn [s] (-> s (* 69069) inc (% (** 2 32)) set-state))))
=> (.run random 1234)
```

```
(1234, 85231147)
=> ;; random can be even shorter by using modify
=> (import [hymn.types.state [modify]])
=> (def random (modify (fn [s] (-> s (* 69069) inc (% (** 2 32))))))
=> (.run random 1234)
(1234, 85231147)
=> ;; use replicate to do computation repeatly
=> (import [hymn.operations [replicate]])
=> (.evaluate (replicate 5 random) 42)
[42, 2900899, 2793697416, 2186085609, 1171637142]
=> ;; sequence on writer monad
=> (import [hymn.operations [sequence]])
=> (import [hymn.types.writer [tell]])
=> (.execute (sequence (map tell (range 1 101))))
5050
```

Using Hymn in Python

```
>>> from hymn.dsl import *
>>> sequence(map(tell, range(1, 101))).execute()
5050
>>> msum = lift(sum)
>>> msum(sequence(map(maybe(int), "12345")))
Just(15)
>>> msum(sequence(map(maybe(int), "12345a")))
Nothing
>>> @failsafe
... def safe_div(a, b):
...     return a / b
...
>>> safe_div(1.0, 2)
Right(0.5)
>>> safe_div(1, 0)
Left(ZeroDivisionError(...))
```

1.2 Requirements

- hy >= 0.11.0

1.3 Installation

Install from PyPI:

```
pip install hymn
```

Install from source, download source package, decompress, then `cd` into source directory, run:

```
make install
```

1.4 License

BSD New, see LICENSE for details.

1.5 Links

Documentation: http://hymn.readthedocs.org/

Issue Tracker: https://github.com/pyx/hymn/issues/

Source Package @ PyPI: https://pypi.python.org/pypi/hymn/

Mercurial Repository @ bitbucket: https://bitbucket.org/pyx/hymn/

Git Repository @ Github: https://github.com/pyx/hymn/

CHAPTER

TWO

EXAMPLES

2.1 Calculating Pi with Monte Carlo Method

Pseudo-random number generator with `State` monad:

```
(import
  [collections [Counter]]
  [time [time]]
  [hymn.dsl [get-state replicate set-state]])

(require hymn.dsl)

;;; Knuth!
(def a 6364136223846793005)
(def c 1442695040888963407)
(def m (** 2 64))

;;; linear congruential generator
(def random
  (do-monad
    [seed get-state
     _ (set-state (-> seed (* a) (+ c) (% m)))
     new-seed get-state]
    (/ new-seed m)))

(def random-point (do-monad [x random y random] (, x y)))

(defn points [seed]
  "stream of random points"
  (while true
    ;; NOTE:
    ;; limited by the maximum recursion depth, we take 150 points each time
    (setv [random-points seed] (.run (replicate 150 random-point) seed))
    (for [point random-points]
      (yield point))))

(defn monte-carlo [number-of-points]
  "use monte carlo method to calculate value of pi"
  (def samples (take number-of-points (points (int (time)))))
  (def result
    (Counter (genexpr (>= 1.0 (+ (** x 2) (** y 2))) [[x y] samples])))
  (-> result (get true) (/ number-of-points) (* 4)))

(defmain [&rest args]
  (if (-> args len (!= 2))
    (print "usage:" (first args) "number-of-points")
    (print "the estimate for pi =" (-> args second int monte-carlo))))
```

Example output:

```
$ ./monte_carlo.hy 50000
the estimate for pi = 3.14232
```

2.2 Calculating Sum

Wicked sum function with `Writer` monad:

```
(import [hymn.dsl [sequence tell]])
(require hymn.dsl)

(defn wicked-sum [numbers]
  (.execute (sequence (map tell numbers))))

(defmain [&rest args]
  (if (-> args len (= 1))
    (print "usage:" (first args) "number1 number2 .. numberN")
    (print "sum:" (->> args rest (map int) wicked-sum))))
```

Example output:

```
$ ./sum.hy 123 456 789
sum: 1368
```

2.3 The FizzBuzz Test

The possibly over-engineered FizzBuzz solution:

```
;;; The fizzbuzz test, in the style inspired by c_wraith on Freenode #haskell

(import [hymn.dsl [<> from-maybe maybe-m]])

(require hymn.dsl)

(defn fizzbuzz [i]
  (from-maybe
    (<>
      (do-monad-with maybe-m [:when (zero? (% i 3))] "fizz")
      (do-monad-with maybe-m [:when (zero? (% i 5))] "buzz"))
    (str i)))

;;; using monoid operation, it is easy to add new case, just add one more line
;;; in the append (<>) call. e.g
(defn fizzbuzzbazz [i]
  (from-maybe
    (<>
      (do-monad-with maybe-m [:when (zero? (% i 3))] "fizz")
      (do-monad-with maybe-m [:when (zero? (% i 5))] "buzz")
      (do-monad-with maybe-m [:when (zero? (% i 7))] "bazz"))
    (str i)))
```

```
(defn format [seq]
  (.join "" (interleave seq (cycle "\t\t\t\t\n"))))

(defmain [&rest args]
  (if (-> args len (= 1))
    (print "usage:" (first args) "up-to-number")
    (print (->> args second int inc (range 1) (map fizzbuzz) format))))
```

Example output:

```
$ ./fizzbuzz.hy 100
1       2       fizz    4       buzz
fizz    7       8       fizz    buzz
11      fizz    13      14      fizzbuzz
16      17      fizz    19      buzz
fizz    22      23      fizz    buzz
26      fizz    28      29      fizzbuzz
31      32      fizz    34      buzz
fizz    37      38      fizz    buzz
41      fizz    43      44      fizzbuzz
46      47      fizz    49      buzz
fizz    52      53      fizz    buzz
56      fizz    58      59      fizzbuzz
61      62      fizz    64      buzz
fizz    67      68      fizz    buzz
71      fizz    73      74      fizzbuzz
76      77      fizz    79      buzz
fizz    82      83      fizz    buzz
86      fizz    88      89      fizzbuzz
91      92      fizz    94      buzz
fizz    97      98      fizz    buzz
```

2.4 Interactive Greeting

Greeting from `Continuation` monad:

```
(import [hymn.dsl [cont-m call-cc]])
(require hymn.dsl)

(defn validate [name exit]
  (with-monad cont-m
    (m-when (not name) (exit "Please tell me your name!"))))

(defn greeting [name]
  (.run (call-cc
          (fn [exit]
            (do-monad
              [_ (validate name exit)]
              (+ "Welcome, " name "!"))))))

(defmain [&rest args]
  (print (greeting (input "Hi, what is your name? "))))
```

Example output:

```
$ ./greeting.hy
Hi, what is your name?
Please tell me your name!
$ ./greeting.hy
Hi, what is your name? Marvin
Welcome, Marvin!
```

2.5 Greatest Common Divisor

Logging with *Writer* monad:

```
(import [hymn.dsl [tell]])

(require hymn.dsl)

(defn gcd [a b]
  (if (zero? b)
    (do-monad
      [_ (tell (.format "the result is: {}\n" (abs a)))]
      (abs a))
    (do-monad-m
      [_ (tell (.format "{} mod {} = {}\n" a b (% a b)))]
      (gcd b (% a b)))))

(defmain [&rest args]
  (if (-> args len (!= 3))
    (print "usage:" (first args) "number1 number2")
    (let [[a (int (get args 1))]
          [b (int (get args 2))]]
      (print "calculating the greatest common divisor of" a "and" b)
      (print (.execute (gcd a b))))))
```

Example output:

```
$ ./gcd.hy 24680 1352
calculating the greatest common divisor of 24680 and 1352
24680 mod 1352 = 344
1352 mod 344 = 320
344 mod 320 = 24
320 mod 24 = 8
24 mod 8 = 0
the result is: 8
```

2.6 Project Euler Problem 9

Solving problem 9 with *List* monad

```
(require hymn.dsl)

(def total 1000)
(def limit (-> total (** 0.5) int inc))

(def triplet
  (do-monad
    [m #*(range 2 limit)
     n #*(range 1 m)
     :let [[a (- (** m 2) (** n 2))]
           [b (* 2 m n)]
           [c (+ (** m 2) (** n 2))]]
     :when (-> (+ a b c) (= total))]
    [a b c]))

(defmain [&rest args]
  (print "Project Euler Problem 9 - list monad example"
         "https://projecteuler.net/problem=9"
         "There exists exactly one Pythagorean triplet"
         "for which a + b + c = 1000.  Find the product abc."
         (->> triplet first (reduce *))
         :sep "\n"))
```

Example output:

```
$ ./euler9.hy
Project Euler Problem 9 - list monad example
https://projecteuler.net/problem=9
There exists exactly one Pythagorean triplet
for which a + b + c = 1000.  Find the product abc.
31875000
```

2.7 Project Euler Problem 29

Solving problem 29 with `lift()` and `List` monad

```
(require hymn.dsl)

(defmain [&rest args]
  (print "Project Euler Problem 29 - lift and list monad example"
         "https://projecteuler.net/problem=29"
         "How many distinct terms are in the sequence generated by"
         "a to the power of b for 2 <= a <= 100 and 2 <= b <= 100?"
         (-> (#^pow #*(range 2 101) #*(range 2 101)) distinct list len)
         :sep "\n"))
```

Example output:

```
$ ./euler29.hy
Project Euler Problem 29 - lift and list monad example
https://projecteuler.net/problem=29
How many distinct terms are in the sequence generated by
a to the power of b for 2 <= a <= 100 and 2 <= b <= 100?
9183
```

2.8 Solving 24 Game

Nondeterministic computation with `List` monad and error handling with `Maybe` monad:

```
(import
  [functools [partial]]
  [itertools [permutations]])

(require hymn.dsl)

(def ops [+ - * /])

(defmacro infix-repr [fmt]
  `(.format ~fmt :a a :b b :c c :d d :op1 (. op1 --name--)
            :op2 (. op2 --name--) :op3 (. op3 --name--)))

;;; use maybe monad to handle division by zero
(defmacro safe [expr] `(#?(fn [] ~expr)))

(defn template [[a b c d]]
  (do-monad-m
    [op1 #*ops
     op2 #*ops
     op3 #*ops]
    ;; (, result infix-representation)
    [(, (safe (op1 (op2 a b) (op3 c d)))
        (infix-repr "({a} {op2} {b}) {op1} ({c} {op3} {d})"))
     (, (safe (op1 a (op2 b (op3 c d))))
        (infix-repr "{a} {op1} ({b} {op2} ({c} {op3} {d}))"))
     (, (safe (op1 (op2 (op3 a b) c) d))
        (infix-repr "(({a} {op3} {b}) {op2} {c}) {op1} {d}"))]))

(defn combinations [numbers]
  (do-monad
    [:let [[seemed (set)]]
     [a b c d] #*(permutations numbers 4)
     :when (not-in (, a b c d) seemed)]
    (do
      (.add seemed (, a b c d))
      [a b c d])))

;;; In python, 8 / (3 - (8 / 3)) = 23.99999999999999, it should be 24 in fact,
;;; so we have to use custom comparison function like this
(defn close-enough [a b] (< (abs (- a b)) 0.0001))

(defn solve [numbers]
  (do-monad
    [[result infix-repr] (<< template (combinations numbers))
     :when (>> result (partial close-enough 24))]
    infix-repr))

(defmain [&rest args]
  (if (-> args len (!= 5))
    (print "usage:" (first args) "number1 number2 number3 number4")
    (->> args rest (map int) solve (.join "\n") print)))
```

Example output:

```
$ ./solve24.hy 2 3 8 8
((2 * 8) - 8) * 3
(3 / 2) * (8 + 8)
3 / (2 / (8 + 8))
((8 - 2) - 3) * 8
((8 * 2) - 8) * 3
```

```
((8 - 3) - 2) * 8
8 * (8 - (2 + 3))
((8 + 8) / 2) * 3
(8 + 8) / (2 / 3)
(8 + 8) * (3 / 2)
8 * (8 - (3 + 2))
((8 + 8) * 3) / 2
```

CHAPTER

THREE

API REFERENCE

3.1 The Monoid Class

class `hymn.types.monoid.`**`Monoid`**

Bases: `object`

the monoid class

types with an associative binary operation that has an identity

`append``(`*other*`)`

an associative operation for monoid

classmethod **`concat`**`(`*seq*`)`

fold a list using the monoid

`empty`

the identity of `append`

`hymn.types.monoid.`**`append`**`(`**monoids*`)`

the associative operation of monoid

3.1.1 Hy Specific API

Functions

`<>`

alias of *`append()`*

3.1.2 Examples

`append()` adds up the values, while handling *`empty`* gracefully, <> is an alias of *`append()`*

```
=> (import [hymn.types.maybe [Just Nothing]])
=> (import [hymn.types.monoid [<> append]])
=> (append (Just "Cuddles ") Nothing (Just "the ") Nothing (Just "Hacker"))
Just('Cuddles the Hacker')
=> (<> (Just "Cuddles ") Nothing (Just "the ") Nothing (Just "Hacker"))
Just('Cuddles the Hacker')
```

3.2 The Monad Class

class `hymn.types.monad.`**`Monad`**(*value*)

Bases: `object`

the monad class

Implements bind operator >> and inverted bind operator << as syntactic sugar. It is equivalent to `(>>=)` and `(=<<)` in haskell, not to be confused with `(>>)` and `(<<)` in haskell.

As python treats assignments as statements, there is no way we can overload >>= as a chainable bind, be it directly overloaded through __`irshift`__, or derived by python itself through __`rshift`__.

The default implementations of `bind`, `fmap` and `join` are mutual recursive, subclasses should at least either override `bind`, or `fmap` and `join`, or all of them for better performance.

`bind`(*f*)

the bind operation

`f` is a function that maps from the underlying value to a monadic type, something like signature `f :: a -> M a` in haskell's term.

The default implementation defines `bind` in terms of `fmap` and `join`.

`fmap`(*f*)

the fmap operation

The default implementation defines `fmap` in terms of `bind` and `unit`.

`join`()

the join operation

The default implementation defines `join` in terms of `bind` and `identity` function.

classmethod **`monadic`**(*f*)

decorator that turn `f` into monadic function of the monad

classmethod **`unit`**(*value*)

the `unit` of monad

3.3 The MonadPlus Class

hymn.types.monadplus - base monadplus class

class `hymn.types.monadplus.`**`MonadPlus`**(*value*)

Bases: *`hymn.types.monad.Monad`*

the monadplus class

Monads that also support choice and failure.

`plus`(*other*)

the associative operation

`zero`

the identity of `plus`.

It should satisfy the following law, left zero (notice the bind operator is haskell's >>= here):

```
zero >>= f = zero
```

3.4 The Continuation Monad

class `hymn.types.continuation.`**`Continuation`**(*value*)

Bases: *`hymn.types.monad.Monad`*

the continuation monad

`bind`(*f*)

the bind operation of `Continuation`

`run`(*k=<function identity>*)

run the continuation

classmethod **`unit`**(*value*)

the unit of continuation monad

`hymn.types.continuation.`**`call_cc`**(*f*)

call with current continuation

`hymn.types.continuation.`**`cont_m`**

alias of `Continuation`

`hymn.types.continuation.`**`continuation_m`**

alias of `Continuation`

`hymn.types.continuation.`**`unit`**

the unit of continuation monad

`hymn.types.continuation.`**`run`**()

alias of `Continuation.run()`

3.4.1 Hy Specific API

`cont-m`

`continuation-m`

alias of `Continuation`

Reader Macro

`< [v]`

create a `Continuation` of `v`

Functions

`call-cc`

alias of *`call_cc()`*

3.4.2 Examples

Do Notation

```
=> (require hymn.dsl)
=> (import [hymn.types.continuation [cont-m]])
=> (.run (do-monad [a (cont-m.unit 1)] (inc a)))
2
```

Operations

`call-cc()` - call with current continuation

```
=> (require hymn.dsl)
=> (import [hymn.types.continuation [call-cc cont-m]])
=> (defn search [n seq]
...   (call-cc
...     (fn [exit]
...       (do-monad-with cont-m
...         [_ (m-when (in n seq) (exit (.index seq n)))]
...         "not found."))))
=> (.run (search 0 [1 2 3 4 5]))
'not found.'
=> (.run (search 0 [1 2 3 0 5]))
3
```

Reader Macro

```
=> (require hymn.dsl)
=> (require hymn.types.continuation)
=> (#<42)
42
=> (.run (do-monad [a #<25 b #<17] (+ a b)))
42
```

3.5 The Either Monad

class `hymn.types.either.`**`Either`**(*value*)

Bases: *`hymn.types.monadplus.MonadPlus`*, *`hymn.mixins.Ord`*

the either monad

computation with two possibilities

`bind`(*f*)

the bind operation of *`Either`*

apply function to the value if and only if this is a *`Right`*.

classmethod **`from_value`**(*value*)

wrap `value` in an *`Either`* monad

return a *`Right`* if the value is evaluated as true. *`Left`* otherwise.

`unit`

alias of *`Right`*

class `hymn.types.either.`**`Left`**(*value*)

Bases: *`hymn.types.either.Either`*

left of `Either`

class `hymn.types.either.`**`Right`**`(`*value*`)`

Bases: `hymn.types.either.Either`

right of `Either`

`hymn.types.either.`**`either`**`(`*handle_left*, *handle_right*, *m*`)`

case analysis for `Either`

apply either `handle-left` or `handle-right` to `m` depending on the type of it, raise `TypeError` if `m` is not an `Either`

`hymn.types.either.`**`either_m`**

alias of `Either`

`hymn.types.either.`**`failsafe`**`(`*func*`)`

decorator to turn func into monadic function of `Either` monad

`hymn.types.either.`**`is_left`**`(`*m*`)`

return `True` if `m` is a `Left`

`hymn.types.either.`**`is_right`**`(`*m*`)`

return `True` if `m` is a `Right`

`hymn.types.either.`**`unit`**

alias of `Right`

`hymn.types.either.`**`zero`** **= Left(u'unknown error')**

left of `Either`

`hymn.types.either.`**`to_either`**`()`

alias of `from_value()`

3.5.1 Hy Specific API

`either-m`

alias of `Either`

Reader Macro

`| [f]`

turn `f` into monadic function with `failsafe()`

Functions

`->either`

`to-either`

alias of `Either.from_value()`

`left?`

alias of `is_left()`

`right?`

alias of `is_right()`

3.5.2 Examples

Comparison

Either are comparable if the wrapped values are comparable. *`Right`* is greater than *`Left`* in any case.

```
=> (import [hymn.types.either [Left Right]])
=> (> (Right 2) (Right 1))
True
=> (< (Left 2) (Left 1))
False
=> (> (Left 2) (Right 1))
False
```

Do Notation

```
=> (require hymn.dsl)
=> (import [hymn.types.either [Left Right]])
=> (do-monad [a (Right 1) b (Right 2)] (+ a b))
Right(3)
=> (do-monad [a (Left 1) b (Right 2)] (+ a b))
Left(1)
```

Do Notation with :when

```
=> (require hymn.dsl)
=> (import [hymn.types.either [either-m]])
=> (defn safe-div [a b]
...   (do-monad-with either-m [:when (not (zero? b))] (/ a b)))
=> (safe-div 1 2)
Right(0.5)
=> (safe-div 1 0)
Left('unknown error')
```

Operations

`->either()` create an *`Either`* from a value

```
=> (import [hymn.types.either [->either]])
=> (->either 42)
Right(42)
=> (->either nil)
Left(None)
```

use `left?()` and `right?()` to test the type

```
=> (import [hymn.types.either [Left Right left? right?]])
=> (right? (Right 42))
True
=> (left? (Left nil))
True
```

`either()` applies function to value in the monad depending on the type

```
=> (import [hymn.types.either [Left Right either]])
=> (either print inc (Left 1))
1
=> (either print inc (Right 1))
2
```

failsafe() turns function into monadic one

```
=> (import [hymn.types.either [failsafe]])
=> (with-decorator failsafe (defn add1 [n] (inc (int n))))
=> (add1 "41")
Right(42)
=> (add1 "nan")
Left(ValueError("invalid literal for int() with base 10: 'nan'",))
=> (def safe-div (failsafe /))
=> (safe-div 1 2)
Right(0.5)
=> (safe-div 1 0)
Left(ZeroDivisionError('division by zero',))
```

Reader Macro

```
=> (require hymn.types.either)
=> (#|int "42")
Right(42)
=> (#|int "nan")
Left(ValueError("invalid literal for int() with base 10: 'nan'",))
=> (def safe-div #|/)
=> (safe-div 1 2)
Right(0.5)
=> (safe-div 1 0)
Left(ZeroDivisionError('division by zero',))
```

3.6 The Identity Monad

hymn.types.identity - the identity monad

class `hymn.types.identity.`**`Identity`**`(`*value*`)`

Bases: *`hymn.types.monad.Monad`*, *`hymn.mixins.Ord`*

the identity monad

`hymn.types.identity.`**`identity_m`**

alias of *`Identity`*

`hymn.types.identity.`**`unit`**

the unit of identity monad

3.6.1 Hy Specific API

`identity-m`

alias of *`Identity`*

3.6.2 Examples

```
=> (require hymn.dsl)
=> (import [hymn.types.identity [identity-m]])
=> (do-monad [a (identity-m.unit 1) b (identity-m.unit 2)] (+ a b))
Identity(3)
```

Identity monad is comparable as long as what's wrapped inside are comparable.

```
=> (> (identity-m.unit 2) (identity-m.unit 1))
True
=> (= (identity-m.unit 42) (identity-m.unit 42))
True
```

3.7 The List Monad

class `hymn.types.list.`**`List`**(*value*)

Bases: *`hymn.types.monadplus.MonadPlus`*, *`hymn.types.monoid.Monoid`*

the list monad

nondeterministic computation

`append`(*other*)

the append operation of *`List`*

classmethod **`concat`**(*list_of_list*)

the concat operation of *`List`*

`fmap`(*f*)

return list obtained by applying `f` to each element of the list

`join`()

join of list monad, concatenate list of list

`plus`(*other*)

concatenate two list

classmethod **`unit`**(**values*)

create a *`List`* from *values*

`hymn.types.list.`**`fmap`**(*f*, *iterable*)

`fmap` works like the builtin `map`, but return a *`List`* instead of `list`

`hymn.types.list.`**`list_m`**

alias of *`List`*

`hymn.types.list.`**`zero`**

the zero of list monad, an empty list

3.7.1 Hy Specific API

`list-m`

alias of *`List`*

Reader Macro

*** [seq]**

turn iterable `seq` into a *List*

3.7.2 Examples

Do Notation

```
=> (require hymn.dsl)
=> (import [hymn.types.list [list-m]])
=> (list (do-monad [a (list-m [1 2 3])] (inc a)))
[2, 3, 4]
=> (list (do-monad [a (list-m [1 2 3]) b (list-m [4 5 6])] (+ a b)))
[5, 6, 7, 6, 7, 8, 7, 8, 9]
=> (list (do-monad [a (list-m "123") b (list-m "xy")] (+ a b)))
['1x', '1y', '2x', '2y', '3x', '3y']
```

Do Notation with :when

```
=> (require hymn.dsl)
=> (import [hymn.types.list [list-m]])
=> (list (do-monad
...         [a (list-m [1 2 4])
...          b (list-m [1 2 4])
...          :when (!= a b)]
...         (/ a b)))
[0.5, 0.25, 2.0, 0.5, 4.0, 2.0]
```

Operations

unit accepts any number of initial values

```
=> (list (list-m.unit))
[]
=> (list (list-m.unit 1))
[1]
=> (list (list-m.unit 1 3))
[1, 3]
=> (list (list-m.unit 1 3 5))
[1, 3, 5]
```

fmap() works like the builtin *map* function, but creates *List* instead of the builtin *list*

```
=> (require hymn.dsl)
=> (import [hymn.types.list [fmap list-m]])
=> (instance? list-m (fmap inc [0 1 2]))
True
=> (for [e (fmap inc [0 1 2])] (print e))
1
2
3
```

Reader Macro

```
=> (require hymn.types.list)
=> (import [hymn.types.list [list-m]])
=> (instance? list-m #*[0 1 2])
True
=> (list (do-monad [a #*(range 10) :when (odd? a)] (* a 2)))
[2, 6, 10, 14, 18]
```

3.8 The Maybe Monad

class `hymn.types.maybe.`**`Just`**(*value*)

Bases: *`hymn.types.maybe.Maybe`*

`Just` of the *`Maybe`*

class `hymn.types.maybe.`**`Maybe`**(*value*)

Bases: *`hymn.types.monadplus.MonadPlus`*, *`hymn.types.monoid.Monoid`*, *`hymn.mixins.Ord`*

the maybe monad

computation that may fail

`append`(*other*)

the append operation of *`Maybe`*

`bind`(*f*)

the bind operation of *`Maybe`*

apply function to the value if and only if this is a *`Just`*.

`from_maybe`(*default*)

return the value contained in the *`Maybe`*

if the *`Maybe`* is *`Nothing`*, it returns the default values.

classmethod **`from_value`**(*value*)

wrap `value` in a *`Maybe`* monad

return a *`Just`* if the value is evaluated as true. *`Nothing`* otherwise.

`unit`

alias of *`Just`*

`hymn.types.maybe.`**`is_nothing`**(*m*)

return `True` if `m` is *`Nothing`*

`hymn.types.maybe.`**`maybe`**(*func=None*, *predicate=None*, *nothing_on_exceptions=None*)

decorator to turn func into monadic function of the *`Maybe`* monad

`hymn.types.maybe.`**`maybe_m`**

alias of *`Maybe`*

`hymn.types.maybe.`**`unit`**

alias of *`Just`*

`hymn.types.maybe.`**`Nothing`** **= Nothing**

the *`Maybe`* that represents nothing, a singleton, like `None` `hymn.types.maybe.`**`zero`** = **Nothing**

the *`Maybe`* that represents nothing, a singleton, like `None`

`hymn.types.maybe.`**`from_maybe`**`()`

alias of *`from_maybe()`*

`hymn.types.maybe.`**`to_maybe`**`()`

alias of *`from_value()`*

3.8.1 Hy Specific API

`maybe-m`

alias of *`Maybe`*

Reader Macro

`? [f]`

turn `f` into monadic function with *`maybe()`*

Functions

`<-maybe`

`from-maybe`

alias of *`Maybe.from_maybe()`*

`->maybe`

`to-maybe`

alias of *`Maybe.from_value()`*

`nothing?`

alias of *`is_nothing()`*

3.8.2 Examples

Comparison

Maybes are comparable if the wrapped values are comparable. *`Just`* is greater than *`Nothing`* in any case.

```
=> (import [hymn.types.maybe [Just Nothing]])
=> (> (Just 2) (Just 1))
True
=> (= (Just 1) (Just 2))
False
=> (= (Just 2) (Just 2))
True
=> (= Nothing Nothing)
True
=> (= Nothing (Just 1))
False
=> (< (Just -1) Nothing)
False
```

Do Notation

```
=> (require hymn.dsl)
=> (import [hymn.types.maybe [Just Nothing]])
=> (do-monad [a (Just 1) b (Just 2)] (+ a b))
Just(3)
=> (do-monad [a (Just 1) b Nothing] (+ a b))
Nothing
```

Do Notation with :when

```
=> (require hymn.dsl)
=> (import [hymn.types.maybe [maybe-m]])
=> (defn safe-div [a b]
...   (do-monad-with maybe-m [:when (not (zero? b))] (/ a b)))
=> (safe-div 1 2)
Just(0.5)
=> (safe-div 1 0)
Nothing
```

Operations

`->maybe()` create a *Maybe* from value

```
=> (import [hymn.types.maybe [->maybe]])
=> (->maybe 42)
Just(42)
=> (->maybe nil)
Nothing
```

`nothing?()` returns `True` if the value is *Nothing*

```
=> (import [hymn.types.maybe [Just Nothing nothing?]])
=> (nothing? Nothing)
True
=> (nothing? (Just 1))
False
```

`<-maybe()` returns the value in the monad, or a default value if it is *Nothing*

```
=> (import [hymn.types.maybe [<-maybe ->maybe]])
=> (nothing? (->maybe nil))
True
=> (def answer (->maybe 42))
=> (def nothing (->maybe nil))
=> (<-maybe answer "not this one")
42
=> (<-maybe nothing "I got nothing")
"I got nothing"
```

append() adds up the values, handling *Nothing* gracefully

```
=> (import [hymn.types.maybe [Just Nothing]])
=> (.append (Just 42) Nothing)
Just(42)
=> (.append (Just 42) (Just 42))
```

```
Just(84)
=> (.append Nothing (Just 42))
Just(42)
```

maybe() turns a function into monadic one

```
=> (import [hymn.types.maybe [maybe]])
=> (with-decorator maybe (defn add1 [n] (inc (int n))))
=> (add1 "41")
Just(42)
=> (add1 "nan")
Nothing
=> (def safe-div (maybe /))
=> (safe-div 1 2)
Just(0.5)
=> (safe-div 1 0)
Nothing
```

Reader Macro

```
=> (require hymn.types.maybe)
=> (#?int "42")
Just(42)
=> (#?int "not a number")
Nothing
=> (def safe-div #?/)
=> (safe-div 1 2)
Just(0.5)
=> (safe-div 1 0)
Nothing
```

3.9 The Reader Monad

class `hymn.types.reader.`**`Reader`**(*value*)

Bases: *`hymn.types.monad.Monad`*

the reader monad

computations which read values from a shared environment

`bind`(*f*)

the bind operation of *`Reader`*

`local`(*f*)

return a reader that execute computation in modified environment

`run`(*e*)

run the reader and extract the final vaule

classmethod **`unit`**(*value*)

the unit of reader monad

`hymn.types.reader.`**`asks`**(*f*)

create a simple reader action from `f`

`hymn.types.reader.`**`local`**(*f*)

executes a computation in a modified environment, `f :: e -> e`

`hymn.types.reader.`**`lookup`**`(`*key*`)`
create a lookup reader of `key` in the environment

`hymn.types.reader.`**`reader`**`(`*f*`)`
create a simple reader action from `f`

`hymn.types.reader.`**`reader_m`**
alias of *`Reader`*

`hymn.types.reader.`**`unit`**
the unit of reader monad

`hymn.types.reader.`**`run`**`()`
alias of *`Reader.run()`*

`hymn.types.reader.`**`ask`**
fetch the value of the environment

3.9.1 Hy Specific API

`reader-m`
alias of *`Reader`*

Function

`<-`
alias of *`lookup()`*

3.9.2 Examples

Do Notation

```
=> (require hymn.dsl)
=> (import [hymn.types.reader [ask]])
=> (.run (do-monad [e ask] (inc e)) 41)
42
```

Operations

`asks()` create a reader with a function, *`reader()`* is an alias of *`asks()`*

```
=> (require hymn.dsl)
=> (import [hymn.types.reader [asks reader]])
=> (.run (do-monad [h (asks first)] h) [5 4 3 2 1])
5
=> (.run (do-monad [h (reader second)] h) [5 4 3 2 1])
4
```

`ask()` fetch the environment

```
=> (require hymn.dsl)
=> (import [hymn.types.reader [ask]])
=> (.run ask 42)
42
```

```
=> (.run (do-monad [e ask] (inc e)) 42)
43
```

`local()` run the reader with modified environment

```
=> (import [hymn.types.reader [ask local]])
=> (.run ask 42)
42
=> (.run ((local inc) ask) 42)
43
```

`lookup()` get the value of key in environment, <− is an alias of `lookup()`

```
=> (require hymn.dsl)
=> (import [hymn.types.reader [lookup <-]])
=> (.run (lookup 1) [1 2 3])
2
=> (.run (lookup 'b) {'a 1 'b 2})
2
=> (.run (<- 1) [1 2 3])
2
=> (.run (<- 'b) {'a 1 'b 2})
2
=> (.run (do-monad [a (<- 'a) b (<- 'b)] (+ a b)) {'a 25 'b 17})
42
```

3.10 The State Monad

class `hymn.types.state.`**`State`**(*value*)

Bases: *`hymn.types.monad.Monad`*

the state monad

computation with a shared state

`bind`(*f*)

the bind operation of `State`

use the final state of this computation as the initial state of the second

`evaluate`(*s*)

evaluate state monad with initial state and return the result

`execute`(*s*)

execute state monad with initial state, return the final state

`run`(*s*)

evaluate state monad with initial state, return result and state

classmethod **`unit`**(*a*)

unit of state monad

`hymn.types.state.`**`state_m`**

alias of `State`

`hymn.types.state.`**`lookup`**(*key*)

return a monadic function that lookup the vaule with key in the state

`hymn.types.state.`**`modify`**(*f*)

maps the current state with *f* to a new state inside a state monad

hymn.types.state.**set_state**(*s*)
: replace the current state and return the previous one

hymn.types.state.**set_value**(*key*, *value*)
: return a monadic function that set the vaule of key in the state

hymn.types.state.**set_values**(***keys/values*)
: return a monadic function that set the vaules of keys in the state

hymn.types.state.**update**(*key*, *f*)
: return a monadic function that update the vaule by f with key in the state

hymn.types.state.**update_value**(*key*, *value*)
: return a monadic function that update the vaule with key in the state

hymn.types.state.**unit**
: the unit of state monad

hymn.types.state.**evaluate**()
: alias of *State.evaluate()*

hymn.types.state.**execute**()
: alias of *State.execute()*

hymn.types.state.**run**()
: alias of *State.run()*

hymn.types.state.**get_state**
: return the current state

hymn.types.state.**gets**(*f*)
: gets specific component of the state, using a projection function `f`

3.10.1 Hy Specific API

state-m
: alias of *State*

Functions

<-
: alias of *lookup()*

<-state

get-state
: alias of *get_state()*

state<-

set-state
: alias of *set_state()*

set-value
: alias of *set_value()*

set-values
: alias of *set_values()*

update-value
: alias of *update_value()*

3.10.2 Examples

Do Notation

```
=> (require hymn.dsl)
=> (import [hymn.types.state [gets]])
=> (.run (do-monad [a (gets first)] a) [1 2 3])
(1, [1, 2, 3])
```

Operations

`get-state()` fetch the shared state, `<-state` is an alias of `get-state()`

```
=> (import [hymn.types.state [get-state <-state]])
=> (.run get-state [1 2 3])
([1, 2, 3], [1, 2, 3])
=> (.run <-state [1 2 3])
([1, 2, 3], [1, 2, 3])
```

`lookup()` get the value of key in the shared state, `<-` is an alias of `lookup()`

```
=> (import [hymn.types.state [lookup <-]])
=> (.run (lookup 1) [1 2 3])
(2, [1, 2, 3])
=> (.evaluate (lookup 1) [1 2 3])
2
=> (.evaluate (lookup 'a) {'a 1 'b 2})
1
=> (.run (<- 1) [1 2 3])
(2, [1, 2, 3])
=> (.evaluate (<- 1) [1 2 3])
2
=> (.evaluate (<- 'a) {'a 1 'b 2})
1
```

`gets()` uses a function to fetch value of shared state

```
=> (import [hymn.types.state [gets]])
=> (.run (gets first) [1 2 3])
(1, [1, 2, 3])
=> (.run (gets second) [1 2 3])
(2, [1, 2, 3])
=> (.run (gets len) [1 2 3])
(3, [1, 2, 3])
```

`modify()` change the current state with a function

```
=> (import [hymn.types.state [modify]])
=> (.run (modify inc) 41)
(41, 42)
=> (.evaluate (modify inc) 41)
41
=> (.execute (modify inc) 41)
42
=> (.run (modify str) 42)
(42, '42')
```

`set-state()` replaces the current state and returns the previous one, `state<-` is an alias of `set-state()`

```
=> (import [hymn.types.state [set-state state<-]])
=> (.run (set-state 42) 1)
(1, 42)
=> (.run (state<- 42) 1)
(1, 42)
```

`set-value()` sets the value in the state with the key

```
=> (import [hymn.types.state [set-value]])
=> (.run (set-value 2 42) [1 2 3])
([1, 2, 3], [1, 2, 42])
```

`set-value()` sets multiple values at once

```
=> (import [hymn.types.state [set-values]])
=> (.run (set-values :a 1 :b 2) {})
(, {} {"b" 2 "a" 1})
```

`update()` changes the value with the key by applying a function to it

```
=> (import [hymn.types.state [update]])
=> (.run (update 0 inc) [0 1 2])
(0, [1, 1, 2])
```

`update-value()` sets the value in the state with the key

```
=> (import [hymn.types.state [update-value]])
=> (.run (update-value 0 42) [0 1 2])
(0, [42, 1, 2])
```

3.11 The Writer Monad

class `hymn.types.writer.`**`Writer`**(*value*)

Bases: *`hymn.types.monad.Monad`*

the writer monad

computation which accumulate output along with result

`bind`(*f*)

the bind operation of *`Writer`*

`execute`()

extract the output of the writer

`run`()

unwrap the writer computation

classmethod **`unit`**(*value*)

unit of writer monad

`hymn.types.writer.`**`censor`**(*f*, *m*)

apply `f` to the output

`hymn.types.writer.`**`listen`**(*m*)

execute `m` and adds its output to the value of computation

`hymn.types.writer.`**`tell`**(*message*)

log the `message`

hymn.types.writer.**writer**(*value*, *message*)
: embed a writer action with value and message

hymn.types.writer.**writer_m**
: alias of *Writer*

hymn.types.writer.**writer_with_type**(*t*)
: create a writer for type t

hymn.types.writer.**writer_with_type_of**(*message*)
: create a writer of type of message

hymn.types.writer.**execute**()
: alias of *Writer.execute()*

hymn.types.writer.**run**()
: alias of *Writer.run()*

3.11.1 Predefined Writers

class hymn.types.writer.**ComplexWriter**(*value*)

class hymn.types.writer.**DecimalWriter**(*value*)

class hymn.types.writer.**FloatWriter**(*value*)

class hymn.types.writer.**FractionWriter**(*value*)

class hymn.types.writer.**ListWriter**(*value*)

class hymn.types.writer.**IntWriter**(*value*)

hymn.types.writer.**StringWriter**
: alias of StrWriter

class hymn.types.writer.**TupleWriter**(*value*)

3.11.2 Hy Specific API

writer-m
: alias of *Writer*

Functions

writer-with-type
: alias of *writer_with_type()*

writer-with-type-of
: alias of *writer_with_type_of()*

Reader Macro

+ [w]
: create a writer that logs w

Writers

complex-writer-m
alias of `ComplexWriter`

decimal-writer-m
alias of `DecimalWriter`

float-writer-m
alias of `FloatWriter`

fraction-writer-m
alias of `FractionWriter`

list-writer-m
alias of `ListWriter`

int-writer-m
alias of `IntWriter`

string-writer-m
alias of `StringWriter`

tuple-writer-m
alias of `TupleWriter`

3.11.3 Examples

Do Notation

```
=> (require hymn.dsl)
=> (import [hymn.types.writer [tell]])
=> (do-monad [_ (tell 1) _ (tell 2)] nil)
IntWriter((None, 3))
=> (do-monad [_ (tell "hello ") _ (tell "world!")] nil)
StrWriter((None, 'hello world!'))
```

Operations

`writer()` creates a `Writer`

```
=> (import [hymn.types.writer [writer]])
=> (writer nil 1)
IntWriter((None, 1))
```

`tell()` adds message into accumulated values of writer

```
=> (import [hymn.types.writer [tell writer]])
=> (.run (tell 1))
(None, 1)
=> (.run (>> (writer 1 1) tell))
(None, 2)
```

`tell()` and `writer()` are smart enough to create writer of appropriate type

```
=> (import [hymn.types.writer [tell writer]])
=> (writer nil "a")
StrWriter((None, 'a'))
=> (writer nil 1)
IntWriter((None, 1))
=> (writer nil 1.0)
FloatWriter((None, 1.0))
=> (writer nil (, 1))
TupleWriter((None, (1,)))
=> (writer nil [1])
ListWriter((None, [1]))
=> (tell "a")
StrWriter((None, 'a'))
=> (tell 1)
IntWriter((None, 1))
=> (tell 1.0)
FloatWriter((None, 1.0))
=> (tell (, 1))
TupleWriter((None, (1,)))
=> (tell [1])
ListWriter((None, [1]))
```

`listen()` get the value of the writer

```
=> (import [hymn.types.writer [listen writer]])
=> (listen (writer "value" 42))
IntWriter((('value', 42), 42))
```

`censor()` apply function to the output

```
=> (import [hymn.types.writer [censor tell]])
=> (require hymn.dsl)
=> (def logs (do-monad [_ (tell [1]) _ (tell [2]) _ (tell [3])] nil))
=> (.execute logs)
[1, 2, 3]
=> (.execute (censor sum logs))
6
```

Reader Macro

```
=> (require hymn.dsl)
=> (require hymn.types.writer)
=> ;; reader macro + works like tell
=> #+1
IntWriter((None, 1))
=> (.execute #+1)
1
=> (do-monad [_ #+1 _ #+2 _ #+4] 42)
IntWriter((42, 7))
```

3.12 Mixin Class

class `hymn.mixins.`**`Ord`**

Bases: `object`

mixin class that implements rich comparison ordering methods

3.13 Monad Operations

`hymn.operations` provide operations and macros for monad computations

3.13.1 Macros

`do-monad [binding-forms expr]`

macro for sequencing monadic computations, with automatic return

```
=> (require hymn.operations)
=> (import [hymn.types.maybe [Just]])
=> (do-monad [a (Just 41)] (inc a))
Just(42)
```

`do-monad-m [binding-forms expr]`

macro for sequencing monadic computations, a.k.a do notation in haskell

```
=> (require hymn.operations)
=> (import [hymn.types.maybe [Just]])
=> (do-monad [a (Just 41)] (m-return a))
Just(42)
```

`do-monad-with [monad binding-forms expr]`

macro for sequencing monadic composition, with said monad as default.

useful when the only binding form is `:when`, we do not know which monad we are working with otherwise

```
=> (require hymn.operations)
=> (import [hymn.types.maybe [maybe-m]])
=> (do-monad-with maybe-m [:when true] 42)
Just(42)
=> (do-monad-with maybe-m [:when false] 42)
Nothing
```

All do monad macros support `:when` if the monad is of type *`MonadPlus`*.

```
=> (require hymn.operations)
=> (import [hymn.types.maybe [maybe-m]])
=> (defn div [a b] (do-monad-with maybe-m [:when (not (zero? b))] (/ a b)))
=> (div 1 2)
Just(0.5)
=> (div 1 0)
Nothing
```

`m-when [test mexpr]`

conditional execution of monadic expressions

`with-monad [test &rest exprs]`

provide default function m-return as the unit of the monad

```
=> (require hymn.Operation)
=> (import [hymn.types.maybe [maybe-m]])
=> (with-monad maybe-m (m-when (even? 1) (m-return 42)))
Just(None)
```

```
=> (with-monad maybe-m (m-when (even? 2) (m-return 42)))
Just(42)
```

3.13.2 Operation on Monads

hymn.operations.**k_compose**(*monadic_funcs*)

right-to-left Kleisli composition of monads.

<=<

alias of `k_compose()`

```
=> (import [hymn.operations [k-compose <=<]])
=> (import [hymn.types.maybe [Just Nothing]])
=> (defn m-double [x] (if (numeric? x) (Just (* x 2)) Nothing))
=> (defn m-inc [x] (if (numeric? x) (Just (inc x)) Nothing))
=> (def +1*2 (k-compose m-double m-inc))
=> (+1*2 1)
Just(4)
=> (def *2+1 (<=< m-inc m-double))
=> (*2+1 2)
Just(5)
=> (*2+1 "two")
Nothing
```

hymn.operations.**k_pipe**(*monadic_funcs*)

left-to-right Kleisli composition of monads.

>=>

alias of `k_compose()`

```
=> (import [hymn.operations [k-pipe >=>]])
=> (import [hymn.types.maybe [Just maybe]])
=> (def m-int (maybe int))
=> (defn m-array [n] (if (> n 0) (Just (* [0] n)) Nothing))
=> (def make-array (k-pipe m-int m-array))
=> (make-array 0)
Nothing
=> (make-array 3)
Just([0, 0, 0])
=> (def make-array (>=> m-int m-array))
=> (make-array 2)
Just([0, 0])
```

hymn.operations.**lift**(*f*)

promote a function to a monad

```
=> (import [hymn.operations [lift]])
=> (import [hymn.types.maybe [Just]])
=> (def m+ (lift +))
=> (m+ (Just 1) (Just 2))
Just(3)
```

hymn.operations.**replicate**(*n*, *m*)

perform the monadic action n times, gathering the results

```
=> (import [hymn.operations [replicate]])
=> (import [hymn.types.list [list-m]])
=> (list (replicate 2 (list-m [0 1])))
[[0, 0], [0, 1], [1, 0], [1, 1]]
```

`hymn.operations.`**`sequence`**`(`*m_values*`)`

evaluate each action in the sequence, and collect the results

```
=> (import [hymn.operations [sequence]])
=> (import [hymn.types.writer [tell]])
=> (.execute (sequence (map tell (range 1 101))))
5050
```

3.14 Utility Functions and Types

3.14.1 Helper Classes

class `hymn.utils.`**`CachedSequence`**`(`*iterable*`)`

Bases: `object`

sequence wrapper that is lazy while keeps the items

class `hymn.utils.`**`SuppressContextManager`**`(`*exceptions*`)`

Bases: `object`

context manager that suppress specified exceptions

3.14.2 Helper Functions

`hymn.utils.`**`compose`**`(`**fs*`)`

function composition

<|

alias of `compose()`

Note: `.` cannot be used as *hy* and *python* already using it, `<|` was chosen because we use `|>` as alias of `pipe()` function

`hymn.utils.`**`const`**`(`*value*`)`

constant function

`hymn.utils.`**`suppress`**`(`**exceptions*`)`

suppress specified exceptions

`hymn.utils.`**`pipe`**`(`**fs*`)`

reversed function composition

|>

alias of `pipe()`

Note: `|>` is different from the same function in *OCaml* and *F#*, which is more like the threading macro `->>` in *hy*

3.15 DSL

The module `hymn.dsl` provides types and functions from other modules of this package, so that they can be imported all at once easily.

Python

```
from hymn.dsl import *
```

Hy

```
(import [hymn.dsl [*]])
```

This module also provides all the macros defined in other modules,

```
(require hymn.dsl)
```

is all you need to use any macro defined in `Hymn`

Note: Some of the function are renamed to more descriptive one to avoid name clash, examples are *hymn.types.reader.lookup()* and *hymn.types.state.lookup()*

The entire source code of this module is listed here for reference:

```
(import
  [hymn.types.monoid [<> append]]
  [hymn.types.continuation
    [Continuation cont-m continuation-m
     call-cc
     run :as run-cont]]
  [hymn.types.either
    [Either either-m
     Left Right left? right? either failsafe]]
  [hymn.types.list [List fmap list-m]]
  [hymn.types.maybe
    [Maybe maybe-m
     Just Nothing <-maybe ->maybe from-maybe maybe nothing? to-maybe]]
  [hymn.types.reader
    [Reader reader-m
     reader
     <- :as <-r
     ask ask :as get-env
     asks asks :as get-env-with
     local local :as use-env-with
     lookup :as lookup-reader
     run :as run-reader]]
  [hymn.types.state
    [State state-m
     <-state get-state set-state state<-
     <- :as <-s
     evaluate :as evaluate-state
     execute :as execute-state
     gets gets :as get-state-with
     lookup :as lookup-state
     modify modify :as modify-state-with
     run :as run-state
     set-value set-value :as set-state-value
     set-values set-values :as set-state-values
     update update :as update-state-value-with
     update-value update-value :as update-state-value]]
  [hymn.types.writer
    [ComplexWriter complex-writer-m
     DecimalWriter decimal-writer-m
     FloatWriter float-writer-m
     FractionWriter fraction-writer-m
     ListWriter list-writer-m
     IntWriter int-writer-m
     StringWriter string-writer-m
     TupleWriter tuple-writer-m
     censor listen tell writer
     writer-with-type
     writer-with-type-of
     run :as run-writer
     execute :as execute-writer]]
  [hymn.operations
    [k-compose <=< k-pipe >=> lift replicate sequence]]
  [hymn.utils [compose <| const pipe |>]])

;;; reader macro for the continuation monad
(require hymn.types.continuation)

;;; reader macro for the either monad
(require hymn.types.either)

;;; reader macro for the list monad
(require hymn.types.list)

;;; reader macro for the maybe monad
(require hymn.types.maybe)

;;; reader macro for the writer monad
(require hymn.types.writer)

;;; macros for monad operations
(require hymn.operations)
```

CHAPTER FOUR

CHANGELOG

- 0.2
  - List.unit now support any number of initial values
  - Maybe and List are instances of Monoid
- 0.1

  First public release.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX