

---

# Hymn Documentation

*Release 1.0.dev0*

**Philip Xu**

**Dec 17, 2020**



---

# Contents

---

<b>1</b>	<b>Hy Monad Notation - a monad library for Hy</b>	<b>3</b>
1.1	Introduction . . . . .	3
1.2	Requirements . . . . .	7
1.3	Installation . . . . .	8
1.4	License . . . . .	8
1.5	Links . . . . .	8
<b>2</b>	<b>Examples</b>	<b>9</b>
2.1	Calculating Pi with Monte Carlo Method . . . . .	9
2.2	Calculating Sum . . . . .	10
2.3	Dependency Handling with Lazy Monad . . . . .	10
2.4	The FizzBuzz Test . . . . .	11
2.5	Interactive Greeting . . . . .	12
2.6	Greatest Common Divisor . . . . .	13
2.7	Project Euler Problem 9 . . . . .	14
2.8	Project Euler Problem 29 . . . . .	14
2.9	Solving 24 Game . . . . .	15
<b>3</b>	<b>API Reference</b>	<b>17</b>
3.1	The Monoid Class . . . . .	17
3.2	The Monad Class . . . . .	18
3.3	The MonadPlus Class . . . . .	18
3.4	The Continuation Monad . . . . .	19
3.5	The Either Monad . . . . .	20
3.6	The Identity Monad . . . . .	24
3.7	The Lazy Monad . . . . .	24
3.8	The List Monad . . . . .	27
3.9	The Maybe Monad . . . . .	29
3.10	The Reader Monad . . . . .	33
3.11	The State Monad . . . . .	35
3.12	The Writer Monad . . . . .	38
3.13	Mixin Class . . . . .	43
3.14	Macros . . . . .	43
3.15	Monad Operations . . . . .	46
3.16	Utility Functions and Types . . . . .	48
3.17	DSL . . . . .	48

<b>4</b>	<b>Changelog</b>	<b>51</b>
<b>5</b>	<b>Indices and tables</b>	<b>53</b>
	<b>Python Module Index</b>	<b>55</b>
	<b>Index</b>	<b>57</b>

Contents:



---

 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

```
=> (import [hymn.types.continuation [cont-m call-cc]])
=> ;; computations in continuation passing style
=> (defn double [x] (cont-m.unit (* x 2)))
=> (setv 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'
=> (require [hymn.macros [do-monad-return]])
=> (.run (do-monad-return [sqr (square 42)] (.format "answer^2 = {}" sqr)))
'answer^2 = 1764'
```

The either monad

```
=> (import [hymn.types.either [Left Right either failsafe]])
=> (require [hymn.macros [do-monad-return]])
=> ;; do notation with either monad
=> (do-monad-return [a (Right 1) b (Right 2)] (/ a b))
Right(0.5)
=> (do-monad-return [a (Right 1) b (Left NaN)] (/ a b))
```

(continues on next page)

(continued from previous page)

```

Left(nan)
=> ;; failsafe is a function decorator that wraps return value into either
=> (setv 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 calls functions accordingly
=> (either print inc (safe-div 4 2))
3.0
=> (either print inc (safe-div 1 0))
division by zero

```

### The identity monad

```

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

```

### The lazy monad

```

=> (import [hymn.types.lazy [force]])
=> (require [hymn.types.lazy [lazy]])
=> ;; lazy computation implemented as monad
=> ;; macro lazy creates deferred computation
=> (setv a (lazy (print "evaluate a") 42))
=> ;; the computation is deferred, notice the value is shown as '_'
=> a
Lazy(_)
=> ;; evaluate it
=> (.evaluate a)
evaluate a
42
=> ;; now the value is cached
=> a
Lazy(42)
=> ;; calling evaluate again will not trigger the computation
=> (.evaluate a)
42
=> (setv b (lazy (print "evaluate b") 21))
=> b
Lazy(_)
=> ;; force evaluate the computation, same as calling .evaluate on the monad
=> (force b)
evaluate b
21
=> ;; force on values other than lazy return the value unchanged
=> (force 42)
42
=> (require [hymn.macros [do-monad-return]])
=> ;; do notation with lazy monad
=> (setv c (do-monad-return
... [x (lazy (print "get x") 1)

```

(continues on next page)



(continued from previous page)

```

...           y (lazy (print "get y") 2)]
...           (+ x y))
=> ;; the computation is deferred
=> c
Lazy(_)
=> ;; do it!
=> (force c)
get x
get y
3
=> ;; again
=> (force c)
3

```

### The list monad

```

=> (import [hymn.types.list [list-m]])
=> (require [hymn.macros [do-monad-return]])
=> ;; use list-m constructor to turn sequence into list monad
=> (setv xs (list-m (range 2)))
=> (setv ys (list-m (range 3)))
=> ;; do notation with list monad is list comprehension
=> (list (do-monad-return [x xs y ys :when (not (zero? y))] (/ x y)) )
[0.0, 0.0, 1.0, 0.5]
=> (require [hymn.types.list [~]])
=> ;; ~ is the tag macro for list-m
=> (list
... (do-monad-return
... [x #~ (range 2)
... y #~ (range 3)
... :when (not (zero? y))]
... (/ x y))
[0.0, 0.0, 1.0, 0.5]

```

### The maybe monad

```

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

```

### The reader monad

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

The state monad

The writer monad

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

Operations on monads

```
=> (import [hymn.operations [lift]])
=> ;; lift promotes function into monad
=> (setv 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 [<-]])
=> (require [hymn.macros [^]])
=> ;; ^ is the tag macro for lift
=> (setv p (#^ print (<- 'message) :end (<- 'end)))
=> (.run p {'message "Hello world" 'end "!\n"})
Hello world!
```

(continues on next page)

(continued from previous page)

```

=> ;; pseudo random number - linear congruential generator
=> (import [hymn.types.state [get-state set-state]])
=> (setv 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]])
=> (setv random (modify (fn [s] (-> s (* 69069) inc (% (** 2 32)))))
=> (.run random 1234)
(1234, 85231147)
=> ;; use replicate to do computation repeatedly
=> (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  $\geq$  0.19.0

For hy version 0.14, please install hymn 0.8

For hy version 0.13, please install hymn 0.7.

For hy version 0.12, please install hymn 0.6.

For hy version 0.11 and earlier, please install hymn 0.5.

See Changelog section in documentation for details.

## 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:** <https://hymn.readthedocs.io/>

**Issue Tracker:** <https://github.com/pyx/hymn/issues/>

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

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

**Git Repository @ Gitlab:** <https://gitlab.com/pyx/hymn/>

## 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.macros [do-monad-return]])

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

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

(setv random-point (do-monad-return [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]
```

(continues on next page)

(continued from previous page)

```

        (yield point))))
(defn monte-carlo [number-of-points]
  "use monte carlo method to calculate value of pi"
  (setv
    samples (take number-of-points (points (int (time))))
    result (Counter (gfor [x y] samples (>= 1.0 (+ (** x 2) (** y 2))))))
  (-> 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]])
(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 Dependency Handling with Lazy Monad

Actions with the *Lazy* monad can be used to handle dependencies:

```

(import [hymn.dsl [force lift]])
(require [hymn.types.lazy [lazy]])
(setv depends (lift (constantly None)))
(defmacro deftask [n &rest actions]
  `(setv ~n
    (depends (lazy (print "(started" '~n))

```

(continues on next page)

(continued from previous page)

```

    ~@actions
    (lazy (print " finished " '~n " " :sep " ")))

(deftask a)
(deftask b)
(deftask c)
(deftask d)
(deftask e)
(deftask f (depends c a))
(deftask g (depends b d))
(deftask h (depends g e f))

(defmain [&rest args]
  (force h))

```

Example output:

```

$ ./deps.hy
(started h
(started g
(started b
  finished b)
(started d
  finished d)
  finished g)
(started e
  finished e)
(started f
(started c
  finished c)
(started a
  finished a)
  finished f)
  finished h)

```

## 2.4 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.macros [do-monad-with]])

(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

```

(continues on next page)

(continued from previous page)

```
(defn fizzbuzzbuzz [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
```

## 2.5 Interactive Greeting

Greeting from *Continuation* monad:

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

(require [hymn.macros [do-monad-return m-when with-monad]])

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

(defn greeting [name]
  (.run (call-cc
```

(continues on next page)



(continued from previous page)

```

    (fn [exit]
      (do-monad-return
        [_ (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!

```

## 2.6 Greatest Common Divisor

Logging with *Writer* monad:

```

(import [hymn.dsl [tell]])

(require [hymn.macros [do-monad do-monad-return]])

(defn gcd [a b]
  (if (zero? b)
    (do-monad-return
      [_ (tell (.format "the result is: {} \n" (abs a))]
        (abs a))
      (do-monad
        [_ (tell (.format "{} mod {} = {} \n" a b (% a b))]
          (gcd b (% a b)))))
    (do-monad
      [_ (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")
    (do
      (setv 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 1151130 1151330
calculating the greatest common divisor of 1151130 and 1151330
1151130 mod 1151330 = 1151130
1151330 mod 1151130 = 200
1151130 mod 200 = 130
200 mod 130 = 70
130 mod 70 = 60
70 mod 60 = 10
60 mod 10 = 0
the result is: 10

```

## 2.7 Project Euler Problem 9

Solving problem 9 with *List* monad

```
(require
 [hymn.macros [do-monad-return]]
 [hymn.types.list [~]])

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

(setv triplet
      (do-monad-return
       [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
```

## 2.8 Project Euler Problem 29

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

```
(require
 [hymn.macros [^]]
 [hymn.types.list [~]])

(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
```

## 2.9 Solving 24 Game

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

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

(require
  [hymn.macros [do-monad do-monad-return]]
  [hymn.types.list [~]]
  [hymn.types.maybe [?]])

(setv ops [+ - * /])

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

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

(defn template [numbers]
  (setv [a b c d] numbers)
  (do-monad
    [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-return
    [: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)) = 23.999999999999999, 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))
```

(continues on next page)

(continued from previous page)

```
(defn solve [numbers]
  (do-monad-return
    [[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
```

## 3.1 The Monoid Class

hymn.types.monoid - base 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

hymn.types.monad - base 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

`hymn.types.continuation` - 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 `hymn.types.continuation.Continuation`

`hymn.types.continuation.continuation_m`  
alias of `hymn.types.continuation.Continuation`

`hymn.types.continuation.run` (*self*, *k*=<*function identity*>)  
run the continuation

`hymn.types.continuation.unit` ()  
alias of `Continuation.unit` ()

`hymn.types.continuation.run` ()  
alias of `Continuation.run` ()

### 3.4.1 Hy Specific API

**cont-m**

**continuation-m**  
alias of `Continuation`

#### Tag Macro

< [**v**]  
create a `Continuation` of *v*

## Functions

### call-cc

alias of `call_cc()`

## 3.4.2 Examples

### Do Notation

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

### Operations

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

```
=> (import [hymn.types.continuation [call-cc cont-m]])
=> (require [hymn.macros [m-when do-monad-with]])
=> (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
```

### Tag Macro

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

## 3.5 The Either Monad

`hymn.types.either` - 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.

**plus** (*other*)  
the associative operation

**unit**  
alias of *Right*

**class** `hymn.types.either.Left` (*value*)  
Bases: `hymn.types.either.Either`  
left of *Either*

**plus** (*other*)  
the associative operation

**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 `hymn.types.either.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 `hymn.types.either.Right`

`hymn.types.either.zero = Left('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*

## Tag 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

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

### Do Notation with :when

```
=> (import [hymn.types.either [either-m]])
=> (require [hymn.macros [do-monad-with]])
=> (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

Use `->either` to create an *Either* from a value

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

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

```
=> (import [hymn.types.either [Left Right left? right?]])
=> (right? (Right 42))
True
=> (left? (Left None))
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'",))
=> (setv safe-div (failsafe /))
=> (safe-div 1 2)
Right(0.5)
=> (safe-div 1 0)
Left(ZeroDivisionError('division by zero',))
```

## Tag Macro

```
=> (require [hymn.types.either []])
=> (#| int "42")
Right(42)
=> (#| int "nan")
Left(ValueError("invalid literal for int() with base 10: 'nan'",))
=> (setv 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

**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()`.

`hymn.types.identity.identity_m`

alias of `hymn.types.identity.Identity`

`hymn.types.identity.unit()`

alias of `Identity.unit()`

### 3.6.1 Hy Specific API

**identity-m**

alias of `Identity`

### 3.6.2 Examples

```
=> (import [hymn.types.identity [identity-m]])
=> (require [hymn.macros [do-monad-return]])
=> (do-monad-return [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.

```
=> (import [hymn.types.identity [identity-m]])
=> (> (identity-m.unit 2) (identity-m.unit 1))
True
=> (= (identity-m.unit 42) (identity-m.unit 42))
True
```

## 3.7 The Lazy Monad

`hymn.types.lazy` - the lazy monad

**class** `hymn.types.lazy.Lazy` (*value*)

Bases: `hymn.types.monad.Monad`

the lazy monad

lazy computation as monad

**bind** (*f*)

the bind operator of `Lazy`

**evaluate** ()  
evaluate the lazy monad

**evaluated**  
return True if this computation is evaluated

**classmethod unit** (*value*)  
the unit of lazy monad

`hymn.types.lazy.evaluate` (*self*)  
evaluate the lazy monad

`hymn.types.lazy.force` (*m*)  
force the deferred computation *m* if it is a *Lazy*, act as function identity otherwise, return the result

`hymn.types.lazy.is_lazy` (*v*)  
return True if *v* is a *Lazy*

`hymn.types.lazy.lazy_m`  
alias of `hymn.types.lazy.Lazy`

`hymn.types.lazy.unit` ()  
alias of `Lazy.unit` ()

`hymn.types.lazy.evaluate` ()  
alias of `Lazy.evaluate` ()

### 3.7.1 Hy Specific API

**lazy-m**  
alias of *Lazy*

#### Macro

**lazy** [&rest *exprs*]  
create a *Lazy* from expressions, the expressions will not be evaluated until being forced by `force` () or `evaluate` ()

#### Function

**lazy?**  
alias of `is_lazy` ()

### 3.7.2 Examples

#### Do Notation

```
=> (require [hymn.macros [do-monad-return]])
=> (require [hymn.types.lazy [lazy]])
=> (setv two (do-monad-return [x (lazy (print "evaluate two") 2)] x))
=> two
Lazy(_)
=> (.evaluate two)
evaluate two
2
```

## Operations

Use macro `lazy()` to create deferred computation from expressions, the computation will not be evaluated until asked explicitly

```
=> (require [hymn.types.lazy [lazy]])
=> (setv answer (lazy (print "the answer is ...") 42))
=> answer
Lazy(_)
=> (.evaluate answer)
the answer is ...
42
=> (.evaluate answer)
42
```

Deferred computation can also be created with expressions wrapped in a function

```
=> (import [hymn.types.lazy [lazy-m]])
=> (setv a (lazy-m (fn [] (print "^o^") 42)))
=> (.evaluate a)
^o^
42
```

Use `evaluate()` to evaluate the computation, the result will be cached

```
=> (require [hymn.types.lazy [lazy]])
=> (setv who (lazy (input "enter your name? ")))
=> who
Lazy(_)
=> (.evaluate who)
enter your name? Marvin
'Marvin'
=> who
Lazy('Marvin')
=> (import [hymn.operations [lift]])
=> (setv m+ (lift +))
=> (setv x (lazy (print "get x") 2))
=> x
Lazy(_)
=> (setv x3 (m+ x x x))
=> x3
Lazy(_)
=> (.evaluate x3)
get x
6
=> x
Lazy(2)
=> x3
Lazy(6)
```

Use `force()` to evaluate `Lazy` as well as other values

```
=> (import [hymn.types.lazy [force]])
=> (require [hymn.types.lazy [lazy]])
=> (force (lazy (print "yes") 1))
yes
1
=> (force 1)
```

(continues on next page)

(continued from previous page)

```

1
=> (setv a (lazy (print "Stat!") (+ 1 2 3)))
=> a
Lazy(_)
=> (force a)
Stat!
6
=> a
Lazy(6)

```

lazy? () returns True if the object is a *Lazy* value

```

=> (import [hymn.types.lazy [lazy?]])
=> (require [hymn.types.lazy [lazy]])
=> (lazy? 1)
False
=> (lazy? (lazy 1))
True

```

## 3.8 The List Monad

hymn.types.list - 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\_lists*)

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 lists

**classmethod unit** (*\*values*)

the unit, 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 *hymn.types.list.List*

hymn.types.list.**zero**

the zero of list monad, an empty list

### 3.8.1 Hy Specific API

#### `list-m`

alias of `List`

#### Tag Macro

##### ~ `[seq]`

turn iterable `seq` into a `List`

New in version 0.7.0.

---

**Note:** This is the new name of tag macro `*`. Starting from 0.12, macros in Hy are namespaced/qualified, using symbol `*` as tag macro is no longer a good idea since `(require [hymn.macros [*]])` should mean importing *all* macros.

---

### 3.8.2 Examples

#### Do Notation

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

#### Do Notation with `:when`

```
=> (import [hymn.types.list [list-m]])
=> (require [hymn.macros [do-monad-return]])
=> (list (do-monad-return
...     [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

```
=> (import [hymn.types.list [list-m]])
=> (list (list-m.unit))
[]
=> (list (list-m.unit 1))
[1]
```

(continues on next page)



(continued from previous page)

```
=> (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`

```
=> (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
```

## Tag Macro

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

## 3.9 The Maybe Monad

`hymn.types.maybe` - 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.

**plus** (*other*)  
the associative operation

**unit**  
alias of *Just*

`hymn.types.maybe.from_maybe` (*self, default*)  
return the value contained in the *Maybe*  
if the *Maybe* is *Nothing*, it returns the default values.

`hymn.types.maybe.hyx_XlessHthan_signX_maybe` (*self, default*)  
return the value contained in the *Maybe*  
if the *Maybe* is *Nothing*, it returns the default values.

`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 `hymn.types.maybe.Maybe`

`hymn.types.maybe.unit`  
alias of `hymn.types.maybe.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` ()

### 3.9.1 Hy Specific API

**maybe-m**  
alias of *Maybe*

#### Tag 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()`

## 3.9.2 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

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

### Do Notation with :when

```
=> (import [hymn.types.maybe [maybe-m]])
=> (require [hymn.macros [do-monad-with]])
=> (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

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

```
=> (import [hymn.types.maybe [->maybe]])
=> (->maybe 42)
Just (42)
=> (->maybe None)
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?]])
=> (nothing? (->maybe None))
True
=> (setv answer (->maybe 42))
=> (setv nothing (->maybe None))
=> (<-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
=> (setv safe-div (maybe /))
=> (safe-div 1 2)
Just (0.5)
=> (safe-div 1 0)
Nothing
```

## Tag Macro

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

## 3.10 The Reader Monad

`hymn.types.reader` - 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 `hymn.types.reader.Reader`

`hymn.types.reader.unit` ()

alias of `Reader.unit` ()

`hymn.types.reader.run` ()

alias of `Reader.run` ()

`hymn.types.reader.ask`

fetch the value of the environment

### 3.10.1 Hy Specific API

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

#### Function

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

### 3.10.2 Examples

#### Do Notation

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

#### Operations

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

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

Use *ask()* to fetch the environment

```
=> (import [hymn.types.reader [ask]])
=> (.run ask 42)
42
=> (require [hymn.macros [do-monad-return]])
=> (.run (do-monad-return [e ask] (inc e)) 42)
43
```

*local()* runs the reader with modified environment

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

Use *lookup()* to get the value of key in environment, **<-** is an alias of *lookup()*

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

(continues on next page)

(continued from previous page)

```

2
=> (.run (<- 1) [1 2 3])
2
=> (.run (<- 'b) {'a 1 'b 2})
2
=> (require [hymn.macros [do-monad-return]])
=> (.run (do-monad-return [a (<- 'a) b (<- 'b)] (+ a b)) {'a 25 'b 17})
42

```

## 3.11 The State Monad

`hymn.types.state` - 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*)

the unit of state monad

`hymn.types.state.state_m`

alias of `hymn.types.state.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` (*\*\*hym\_keysXsolidusXvalues*)

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()`  
alias of `State.unit()`

`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`

### 3.11.1 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()`

### 3.11.2 Examples

#### Do Notation

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



## Operations

Use `get-state()` to 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])
```

Use `lookup()` to 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()` changes 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-values () sets multiple values at once

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])
```

## 3.12 The Writer Monad

hymn.types.writer - the writer monad

```
class hymn.types.writer.ComplexWriter (value)
  Bases: hymn.types.writer.Writer
```

```
  output_type
    alias of builtins.complex
```

```
class hymn.types.writer.DecimalWriter (value)
  Bases: hymn.types.writer.Writer
```

```
  output_type
    alias of decimal.Decimal
```

```
class hymn.types.writer.FloatWriter (value)
  Bases: hymn.types.writer.Writer
```

```
  output_type
    alias of builtins.float
```

```
class hymn.types.writer.FractionWriter (value)
  Bases: hymn.types.writer.Writer
```

```
  output_type
    alias of fractions.Fraction
```

```
class hymn.types.writer.IntWriter (value)
  Bases: hymn.types.writer.Writer
```

```
  output_type
    alias of builtins.int
```

```
class hymn.types.writer.ListWriter (value)
  Bases: hymn.types.writer.Writer
```

```
  output_type
    alias of builtins.list
```

```

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

class hymn.types.writer.TupleWriter (value)
    Bases: hymn.types.writer.Writer

    output_type
        alias of builtins.tuple

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 writer

    run ()
        unwrap the writer computation

    classmethod unit (value)
        the unit of writer monad

hymn.types.writer.cancel (f, m)
    apply f to the output

hymn.types.writer.complex_writer_m
    alias of hymn.types.writer.ComplexWriter

hymn.types.writer.decimal_writer_m
    alias of hymn.types.writer.DecimalWriter

hymn.types.writer.execute (self)
    extract the output of writer

hymn.types.writer.float_writer_m
    alias of hymn.types.writer.FloatWriter

hymn.types.writer.fraction_writer_m
    alias of hymn.types.writer.FractionWriter

hymn.types.writer.int_writer_m
    alias of hymn.types.writer.IntWriter

hymn.types.writer.list_writer_m
    alias of hymn.types.writer.ListWriter

hymn.types.writer.listen (m)
    execute m and adds its output to the value of computation

hymn.types.writer.run (self)
    unwrap the writer computation

hymn.types.writer.string_writer_m
    alias of hymn.types.writer.StrWriter

hymn.types.writer.tell (message)
    log the message

```

`hymn.types.writer.tuple_writer_m`  
alias of `hymn.types.writer.TupleWriter`

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

`hymn.types.writer.writer_m`  
alias of `hymn.types.writer.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` ()

### 3.12.1 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 `hymn.types.writer.StrWriter`

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

### 3.12.2 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` ()

#### Tag 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*

## 3.12.3 Examples

### Do Notation

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

### Operations

*writer()* creates a *Writer*

```
=> (import [hymn.types.writer [writer]])
=> (writer None 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 None 98j)
ComplexWriter((None, 98j))
=> (import [decimal [Decimal]])
=> (writer None (Decimal "7.31"))
DecimalWriter((None, Decimal('7.31')))
=> (writer None 1.0)
FloatWriter((None, 1.0))
=> (writer None 7/31)
FractionWriter((None, Fraction(7, 31)))
=> (writer None [85 70 92])
ListWriter((None, [85, 70, 92]))
=> (writer None 1)
IntWriter((None, 1))
=> (writer None "a")
StrWriter((None, 'a'))
=> (writer None (, 1151130 1151330))
TupleWriter((None, (1151130, 1151330)))
=> (tell 98j)
ComplexWriter((None, 98j))
=> (tell (Decimal "7.31"))
DecimalWriter((None, Decimal('7.31')))
=> (tell 1.0)
FloatWriter((None, 1.0))
=> (tell 7/31)
FractionWriter((None, Fraction(7, 31)))
=> (tell [85 70 92])
ListWriter((None, [85, 70, 92]))
=> (tell 1)
IntWriter((None, 1))
=> (tell "a")
StrWriter((None, 'a'))
=> (tell (, 1151130 1151330))
TupleWriter((None, (1151130, 1151330)))

```

Use `listen()` to get the value of the writer

```

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

```

Use `cancel()` to apply function to the output

```

=> (import [hymn.types.writer [cancel tell]])
=> (require [hymn.macros [do-monad-return]])
=> (setv logs (do-monad-return [_ (tell [1]) _ (tell [2]) _ (tell [3])] None))
=> (.execute logs)
[1, 2, 3]
=> (.execute (cancel sum logs))
6

```

## Tag Macro

```

=> (require [hymn.types.writer [+]])
=> ;; tag macro + works like tell

```

(continues on next page)

(continued from previous page)

```

=> #+ 1
IntWriter((None, 1))
=> (.execute #+ 1)
1
=> (require [hymn.macros [do-monad-return]])
=> (do-monad-return [_ #+ 1 _ #+ 2 _ #+ 4] 42)
IntWriter((42, 7))

```

## 3.13 Mixin Class

hymn.mixins - mixin classes

```

class hymn.mixins.Ord
  Bases: object

```

mixin class that implements rich comparison ordering methods

## 3.14 Macros

hymn.macros - monad operations implemented as macros

hymn.macros provide macros for monad computations

### 3.14.1 Operation Macros

**do-monad** [**binding-forms** **expr**]

Changed in version 0.9: This macro was named `do-monad-m` before version 0.9

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

```

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

```

**do-monad-return** [**binding-forms** **expr**]

Changed in version 0.9: This macro was named `do-monad` before version 0.9

macro for sequencing monadic computations, with automatic return

```

=> (import [hymn.types.maybe [Just]])
=> (require [hymn.macros [do-monad-return]])
=> (do-monad-return [a (Just 41)] (inc 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

```
=> (import [hymn.types.maybe [maybe-m]])
=> (require [hymn.macros [do-monad-with]])
=> (do-monad-with maybe-m [:when True] 42)
Just(42)
=> (do-monad-with maybe-m [:when False] 42)
Nothing
```

All do monad macros support `:let` binding, like this:

```
=> (import [hymn.types.maybe [Just]])
=> (require [hymn.macros [do-monad-return]])
=> (defn half [x]
... (do-monad-return
... [:let [two 2]
... a x
... :let [b (/ a two)]
... b))
=> (half (Just 42))
Just(21.0)
```

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

```
=> (import [hymn.types.maybe [maybe-m]])
=> (require [hymn.macros [do-monad-with]])
=> (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
```

**monad->** `[init-value &rest actions]`  
threading macro for monadic actions

```
=> (import [hymn.types.maybe [maybe-m]])
=> (setv m-inc (maybe-m.monadic inc))
=> (setv m-div (maybe-m.monadic /))
=> (require [hymn.macros [monad->]])
=> ;; threading macro for monadic actions
=> (monad-> (maybe-m.unit 99) m-inc (m-div 5) (m-div 2))
Just(10.0)
=> ;; is equivalent to
=> (require [hymn.macros [do-monad]])
=> (do-monad [a (maybe-m.unit 99) b (m-inc a) c (m-div b 5)] (m-div c 2))
Just(10.0)
```

**monad->>** `[init-value &rest actions]`  
threading tail macro for monadic actions

```
=> (import [hymn.types.maybe [maybe-m]])
=> (setv m-inc (maybe-m.monadic inc))
=> (setv m-div (maybe-m.monadic /))
=> (require [hymn.macros [monad->>]])
=> ;; threading tail macro for monadic actions
=> (monad->> (maybe-m.unit 4) m-inc (m-div 25) (m-div 100))
Just(20.0)
=> ;; is equivalent to
=> (require [hymn.macros [do-monad]])
```

(continues on next page)



(continued from previous page)

```
=> (do-monad [a (maybe-m.unit 4) b (m-inc a) c (m-div 25 b)] (m-div 100 c))
Just(20.0)
```

**m-for** `[[n seq] &rest expr]`  
macro for sequencing monadic actions

```
=> (import [hymn.types.maybe [maybe-m]])
=> (require [hymn.macros [m-for]])
=> ;; with simple monad, e.g. maybe
=> (m-for [a (range 3)] (maybe-m.unit a))
Just([0, 1, 2])
=> ;; with reader monad
=> (import [hymn.types.reader [<-]])
=> (setv readers
... (m-for [a (range 5)]
... (print "create reader" a)
... (<- a)))
create reader 0
create reader 1
create reader 2
create reader 3
create reader 4
=> (.run readers [11 12 13 14 15 16])
[11, 12, 13, 14, 15]
=> (.run readers "abcdefg")
['a', 'b', 'c', 'd', 'e']
=> ;; with writer monad
=> (import [hymn.types.writer [tell]])
=> (.execute (m-for [a (range 1 101)] (tell a)))
5050
```

**m-when** `[test mexpr]`  
conditional execution of monadic expressions

**with-monad** `[monad &rest exprs]`  
provide default function m-return as the unit of the monad

```
=> (import [hymn.types.maybe [maybe-m]])
=> (require [hymn.macros [m-when with-monad]])
=> (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)
```

**monad-comp** `[expr binding-forms &optional condition]`  
different syntax for do-monad-return, in the style of list/dict/set comprehensions, the condition part is optional and can only be used with *MonadPlus* as in do-monad-return

```
=> (import [hymn.types.maybe [Just]])
=> (require [hymn.macros [monad-comp]])
=> (monad-comp (+ a b) [a (Just 1) b (Just 2)])
Just(3)
=> (monad-comp (/ a b) [a (Just 1) b (Just 0)] (not (zero? b)))
Nothing
=> (import [hymn.types.list [list-m]])
=> (list (monad-comp (/ a b) [a (list-m [1 2]) b (list-m [4 8])]))
```

(continues on next page)

(continued from previous page)

```
[0.25, 0.125, 0.5, 0.25]
=> (list (monad-comp (/ a b) [a (list-m [1 2]) b (list-m [0 1])] (not (zero? b))))
[1.0, 2.0]
```

### 3.14.2 Tag Macros

**^ [f]**

lift () tag macro, #^ f is expanded to (lift f)

```
=> (import [hymn.types.maybe [Just Nothing]])
=> (require [hymn.macros [^]])
=> (#^ + (Just 1) (Just 2))
Just(3)
=> (#^ + (Just 1) Nothing)
Nothing
```

**= [value]**

tag macro for m-return, the unit inside do-monad-return macros, #= v is expanded to (m-return v)

```
=> (import [hymn.types.maybe [Just maybe-m]])
=> (require [hymn.macros [= do-monad do-monad-with]])
=> (do-monad-with maybe-m [a #= 1 b #= 2] (+ a b))
Just(3)
=> (do-monad [a (Just 1)] #= (inc a))
Just(2)
```

## 3.15 Monad Operations

hymn.operations - operations on monads

hymn.operations provide operations for monad computations

hymn.operations.k\_compose()

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))
=> (setv +1*2 (k-compose m-double m-inc))
=> (+1*2 1)
Just(4)
=> (setv *2+1 (<=< m-inc m-double))
=> (*2+1 2)
Just(5)
=> (*2+1 "two")
Nothing
```

hymn.operations.k\_pipe()

left-to-right Kleisli composition of monads.

`>=>`alias of `k_pipe()`

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

`hymn.operations.lift()`

promote a function to a monad

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

`hymn.operations.m_map()`

map monadic function mf to a sequence, then execute that sequence of monadic values

**m-map**alias of `m_map()`

```
=> (import [hymn.operations [m-map]])
=> (import [hymn.types.maybe [maybe-m]])
=> (m-map maybe-m.unit (range 5))
Just([0, 1, 2, 3, 4])
=> (m-map (maybe-m.monadic inc) (range 5))
Just([1, 2, 3, 4, 5])
=> (import [hymn.types.writer [tell]])
=> (.execute (m-map tell (range 1 101)))
5050
```

`hymn.operations.replicate()`

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()`

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
```

## 3.16 Utility Functions and Types

### 3.16.1 Helper Classes

**class** `hymn.utils.CachedSequence` (*iterable*)  
Bases: `object`  
sequence wrapper that is lazy while keeps the items

## 3.17 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 [*]])
```

---

**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.identity [Identity identity-m]]
 [hymn.types.lazy [Lazy lazy-m evaluate :as evaluate-lazy force lazy?]]
 [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
```

(continues on next page)

(continued from previous page)

```
<-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 m-map replicate sequence]])
```



- 0.9
  - Backward incompatible change supporting hy 0.19
  - Support python 3.7, 3.8, 3.9
  - Drop python 2.7, 3.4, 3.5 support
  - Renamed macro do-monad as do-monad-return
  - Renamed macro do-monad-m as do-monad
- 0.8
  - Backward incompatible change supporting hy 0.14
  - Drop python 3.3 support
  - Remove type-specific sharp macros in *hymn.macros*, newer hy does not support aggregating macros in a module. Type-specific sharp macros are still available in their respective type modules.
- 0.7
  - Backward incompatible change supporting hy 0.13
  - Renamed sharp macro \* to ~
  - Removed macros in *hymn.dsl*, use *hymn.macros* instead
- 0.6
  - Backward incompatible change supporting hy 0.12, using new syntax
  - Moved monad operation macros into separate module: *hymn.macros*
- 0.5
  - Version bump to indicate at least halfway done with planned features
- 0.4
  - Support python 3.5

- Remove alias of compose and pipe <| and |> to avoid confusion
- New macros: monad-> and monad->>, threading macros for monad
- 0.3
  - New operation: m-map
  - New macros: m-for, monad-comp
  - New type: deferred computation implemented as the Lazy monad
  - Improved documentation
- 0.2
  - List.unit now support any number of initial values
  - Maybe and List are instances of Monoid
- 0.1

First public release.



## CHAPTER 5

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`



## h

`hymn.macros`, 43  
`hymn.mixins`, 43  
`hymn.operations`, 46  
`hymn.types.continuation`, 19  
`hymn.types.either`, 20  
`hymn.types.identity`, 24  
`hymn.types.lazy`, 24  
`hymn.types.list`, 27  
`hymn.types.maybe`, 29  
`hymn.types.monad`, 18  
`hymn.types.monadplus`, 18  
`hymn.types.monoid`, 17  
`hymn.types.reader`, 33  
`hymn.types.state`, 35  
`hymn.types.writer`, 38



## A

append() (*hymn.types.list.List method*), 27  
 append() (*hymn.types.maybe.Maybe method*), 29  
 append() (*hymn.types.monoid.Monoid method*), 17  
 append() (*in module hymn.types.monoid*), 17  
 ask (*in module hymn.types.reader*), 33  
 asks() (*in module hymn.types.reader*), 33

## B

bind() (*hymn.types.continuation.Continuation method*), 19  
 bind() (*hymn.types.either.Either method*), 20  
 bind() (*hymn.types.identity.Identity method*), 24  
 bind() (*hymn.types.lazy.Lazy method*), 24  
 bind() (*hymn.types.maybe.Maybe method*), 29  
 bind() (*hymn.types.monad.Monad method*), 18  
 bind() (*hymn.types.reader.Reader method*), 33  
 bind() (*hymn.types.state.State method*), 35  
 bind() (*hymn.types.writer.Writer method*), 39

## C

CachedSequence (*class in hymn.utils*), 48  
 call\_cc() (*in module hymn.types.continuation*), 19  
 censor() (*in module hymn.types.writer*), 39  
 complex\_writer\_m (*in module hymn.types.writer*), 39  
 ComplexWriter (*class in hymn.types.writer*), 38  
 concat() (*hymn.types.list.List class method*), 27  
 concat() (*hymn.types.monoid.Monoid class method*), 17  
 cont\_m (*in module hymn.types.continuation*), 19  
 Continuation (*class in hymn.types.continuation*), 19  
 continuation\_m (*in module hymn.types.continuation*), 19

## D

decimal\_writer\_m (*in module hymn.types.writer*), 39  
 DecimalWriter (*class in hymn.types.writer*), 38

## E

Either (*class in hymn.types.either*), 20  
 either() (*in module hymn.types.either*), 21  
 either\_m (*in module hymn.types.either*), 21  
 empty (*hymn.types.monoid.Monoid attribute*), 17  
 evaluate() (*hymn.types.lazy.Lazy method*), 25  
 evaluate() (*hymn.types.state.State method*), 35  
 evaluate() (*in module hymn.types.lazy*), 25  
 evaluate() (*in module hymn.types.state*), 36  
 evaluated (*hymn.types.lazy.Lazy attribute*), 25  
 execute() (*hymn.types.state.State method*), 35  
 execute() (*hymn.types.writer.Writer method*), 39  
 execute() (*in module hymn.types.state*), 36  
 execute() (*in module hymn.types.writer*), 39

## F

failsafe() (*in module hymn.types.either*), 21  
 float\_writer\_m (*in module hymn.types.writer*), 39  
 FloatWriter (*class in hymn.types.writer*), 38  
 fmap() (*hymn.types.list.List method*), 27  
 fmap() (*hymn.types.monad.Monad method*), 18  
 fmap() (*in module hymn.types.list*), 27  
 force() (*in module hymn.types.lazy*), 25  
 fraction\_writer\_m (*in module hymn.types.writer*), 39  
 FractionWriter (*class in hymn.types.writer*), 38  
 from\_maybe() (*hymn.types.maybe.Maybe method*), 29  
 from\_maybe() (*in module hymn.types.maybe*), 30  
 from\_value() (*hymn.types.either.Either class method*), 21  
 from\_value() (*hymn.types.maybe.Maybe class method*), 29

## G

get\_state (*in module hymn.types.state*), 36

## H

hymn.macros (*module*), 43  
 hymn.mixins (*module*), 43

`hymn.operations` (*module*), 46  
`hymn.types.continuation` (*module*), 19  
`hymn.types.either` (*module*), 20  
`hymn.types.identity` (*module*), 24  
`hymn.types.lazy` (*module*), 24  
`hymn.types.list` (*module*), 27  
`hymn.types.maybe` (*module*), 29  
`hymn.types.monad` (*module*), 18  
`hymn.types.monadplus` (*module*), 18  
`hymn.types.monoid` (*module*), 17  
`hymn.types.reader` (*module*), 33  
`hymn.types.state` (*module*), 35  
`hymn.types.writer` (*module*), 38  
`hyx_XlessHthan_signX_maybe` () (*in module*  
*hymn.types.maybe*), 30

## I

`Identity` (*class in hymn.types.identity*), 24  
`identity_m` (*in module hymn.types.identity*), 24  
`int_writer_m` (*in module hymn.types.writer*), 39  
`IntWriter` (*class in hymn.types.writer*), 38  
`is_lazy` () (*in module hymn.types.lazy*), 25  
`is_left` () (*in module hymn.types.either*), 21  
`is_nothing` () (*in module hymn.types.maybe*), 30  
`is_right` () (*in module hymn.types.either*), 21

## J

`join` () (*hymn.types.list.List method*), 27  
`join` () (*hymn.types.monad.Monad method*), 18  
`Just` (*class in hymn.types.maybe*), 29

## K

`k_compose` () (*in module hymn.operations*), 46  
`k_pipe` () (*in module hymn.operations*), 46

## L

`Lazy` (*class in hymn.types.lazy*), 24  
`lazy_m` (*in module hymn.types.lazy*), 25  
`Left` (*class in hymn.types.either*), 21  
`lift` () (*in module hymn.operations*), 47  
`List` (*class in hymn.types.list*), 27  
`list_m` (*in module hymn.types.list*), 27  
`list_writer_m` (*in module hymn.types.writer*), 39  
`listen` () (*in module hymn.types.writer*), 39  
`ListWriter` (*class in hymn.types.writer*), 38  
`local` () (*hymn.types.reader.Reader method*), 33  
`local` () (*in module hymn.types.reader*), 33  
`lookup` () (*in module hymn.types.reader*), 33  
`lookup` () (*in module hymn.types.state*), 35

## M

`m_map` () (*in module hymn.operations*), 47  
`Maybe` (*class in hymn.types.maybe*), 29

`maybe` () (*in module hymn.types.maybe*), 30  
`maybe_m` (*in module hymn.types.maybe*), 30  
`modify` () (*in module hymn.types.state*), 35  
`Monad` (*class in hymn.types.monad*), 18  
`monadic` () (*hymn.types.monad.Monad class method*),  
18  
`MonadPlus` (*class in hymn.types.monadplus*), 18  
`Monoid` (*class in hymn.types.monoid*), 17

## N

`Nothing` (*in module hymn.types.maybe*), 30

## O

`Ord` (*class in hymn.mixins*), 43  
`output_type` (*hymn.types.writer.ComplexWriter attribute*), 38  
`output_type` (*hymn.types.writer.DecimalWriter attribute*), 38  
`output_type` (*hymn.types.writer.FloatWriter attribute*), 38  
`output_type` (*hymn.types.writer.FractionWriter attribute*), 38  
`output_type` (*hymn.types.writer.IntWriter attribute*),  
38  
`output_type` (*hymn.types.writer.ListWriter attribute*),  
38  
`output_type` (*hymn.types.writer.TupleWriter attribute*), 39

## P

`plus` () (*hymn.types.either.Either method*), 21  
`plus` () (*hymn.types.either.Left method*), 21  
`plus` () (*hymn.types.list.List method*), 27  
`plus` () (*hymn.types.maybe.Maybe method*), 30  
`plus` () (*hymn.types.monadplus.MonadPlus method*),  
18

## R

`Reader` (*class in hymn.types.reader*), 33  
`reader` () (*in module hymn.types.reader*), 33  
`reader_m` (*in module hymn.types.reader*), 33  
`replicate` () (*in module hymn.operations*), 47  
`Right` (*class in hymn.types.either*), 21  
`run` () (*hymn.types.continuation.Continuation method*),  
19  
`run` () (*hymn.types.reader.Reader method*), 33  
`run` () (*hymn.types.state.State method*), 35  
`run` () (*hymn.types.writer.Writer method*), 39  
`run` () (*in module hymn.types.continuation*), 19  
`run` () (*in module hymn.types.reader*), 33  
`run` () (*in module hymn.types.state*), 36  
`run` () (*in module hymn.types.writer*), 39

## S

sequence () (in module *hymn.operations*), 47  
 set\_state () (in module *hymn.types.state*), 35  
 set\_value () (in module *hymn.types.state*), 35  
 set\_values () (in module *hymn.types.state*), 35  
 State (class in *hymn.types.state*), 35  
 state\_m (in module *hymn.types.state*), 35  
 string\_writer\_m (in module *hymn.types.writer*), 39  
 StringWriter (class in *hymn.types.writer*), 38

## T

tell () (in module *hymn.types.writer*), 39  
 tuple\_writer\_m (in module *hymn.types.writer*), 39  
 TupleWriter (class in *hymn.types.writer*), 39

## U

unit (*hymn.types.either.Either* attribute), 21  
 unit (*hymn.types.maybe.Maybe* attribute), 30  
 unit (in module *hymn.types.either*), 21  
 unit (in module *hymn.types.maybe*), 30  
 unit () (*hymn.types.continuation.Continuation* class method), 19  
 unit () (*hymn.types.lazy.Lazy* class method), 25  
 unit () (*hymn.types.list.List* class method), 27  
 unit () (*hymn.types.monad.Monad* class method), 18  
 unit () (*hymn.types.reader.Reader* class method), 33  
 unit () (*hymn.types.state.State* class method), 35  
 unit () (*hymn.types.writer.Writer* class method), 39  
 unit () (in module *hymn.types.identity*), 24  
 unit () (in module *hymn.types.reader*), 33  
 unit () (in module *hymn.types.state*), 36  
 update () (in module *hymn.types.state*), 35  
 update\_value () (in module *hymn.types.state*), 35

## W

Writer (class in *hymn.types.writer*), 39  
 writer () (in module *hymn.types.writer*), 40  
 writer\_m (in module *hymn.types.writer*), 40  
 writer\_with\_type () (in module *hymn.types.writer*), 40  
 writer\_with\_type\_of () (in module *hymn.types.writer*), 40

## Z

zero (*hymn.types.monadplus.MonadPlus* attribute), 19  
 zero (in module *hymn.types.either*), 21  
 zero (in module *hymn.types.list*), 27  
 zero (in module *hymn.types.maybe*), 30