Monads in Clojure

Clojure has a clojure.contrib.monads library, written by Konrad Hinsen.

It is patterned off of Haskell's monad library, but uses macros and structures instead of Haskell's type classes. In some ways this is nicer than Haskell's monad support!

This library is growing and undergoes frequent revisions, so it is worth updating it frequently to get the latest goodies!

Clojure Monad Structure

```
; create an anonymous monad structure
(defmacro monad
   [operations]
   `(let [~'m-bind   ::undefined
          ~'m-result ::undefined
          ~'m-zero   ::undefined
          ~'m-plus   ::undefined
          ~@operations]
      {:m-result ~'m-result
       :m-bind ~'m-bind
       ; these are optional
       :m-zero ~'m-zero
       :m-plus ~'m-plus}))
```

Named Monads

```
; create a named macro definition
(defmacro defmonad
   ([name doc-string operations]
    (let [doc-name (with-meta name {:doc doc-string})]
      `(defmonad ~doc-name ~operations)))

   ([name operations]
    `(def ~name (monad ~operations))))
```

Example: Identity Monad

```
; Identity monad
(defmonad identity-m
   "Monad describing plain computations. This monad
    does in fact nothing at all. It is useful for
    testing, for combination with monad transformers,
    and for code that is parameterized with a monad."

  [m-result identity
   m-bind   (fn m-result-id [mv f]
              (f mv))
  ])
```

Example: Writer Monad

```
(defn writer-m
  "Monad describing computations that accumulate
   data on the side, e.g. for logging. The monadic
   values have the structure [value log]. Any of the
   accumulators from clojure.contrib.accumulators can
   be used for storing the log data. Its empty value
   is passed as a parameter."
  [empty-accumulator]
  (monad
     [m-result  (fn m-result-writer [v]
                  [v empty-accumulator])
      m-bind    (fn m-bind-writer [mv f]
                  (let [[v1 a1] mv
                        [v2 a2] (f v1)]
                    [v2 (combine a1 a2)]))
     ]))
```

Monadic Expressions

```
(defmacro with-monad
    "Evaluates an expression after replacing the
     keywords defining the monad operations by the
     functions associated with these keywords in the
     monad definition given by name."
   [name & exprs]
   `(let [~'m-bind   (:m-bind ~name)
          ~'m-result (:m-result ~name)
          ~'m-zero   (:m-zero ~name)
          ~'m-plus   (:m-plus ~name)]
      (do ~@exprs)))
```

Monadic Functions

```
(defmacro defmonadfn
  "Like defn, but for functions that use monad
   operations and are used inside a with-monad block."
  ([name doc-string args expr]
    (let [doc-name (with-meta name {:doc doc-string})]
      `(defmonadfn ~doc-name ~args ~expr)))

  ([name args expr]
   (let [fn-name (symbol (str *ns*) (format "m+%s+m" (str name)))]
    `(do
      (defmacro ~name ~args
        (list (quote ~fn-name)
              '~'m-bind '~'m-result '~'m-zero '~'m-plus
              ~@args))
      (defn ~fn-name [~'m-bind ~'m-result ~'m-zero ~'m-plus ~@args]
        ~expr)))))
```

Example: Abstraction

```
(defn divides? [n d] (zero? (rem n d)))

(defn divisors [n]
  (let [ ds  (filter (partial divides?)
                     (range 2 (inc (int (Math/sqrt n)))))
         ds2 (map (partial / n) ds)
       ]
    (concat ds (reverse ds2))))


(defmonadfn get-divisors [n]
  (reduce m-plus m-zero (map m-result (divisors n))))
```

Example: Abstraction (cont'd)

```
user=> (with-monad maybe-m (get-divisors 23))
nil

user=> (with-monad maybe-m (get-divisors 24))
2

user=> (with-monad sequence-m (get-divisors 24))
(2 3 4 6 8 12)

user=> (with-monad sequence-m (get-divisors 25))
(5 5)

user=> (with-monad set-m (get-divisors 25))
#{5}
```

domonad

```
(fn [m rec e]
  (domonad m
           [args (m-result (rest e))
            x    (rec (first args))
            y    (rec (second args))]
           (+ x y)))
```

```
; -- Equivalent Haskell code
; \m rec e -> do let args = tail e
;                x ← rec (args!!0)
;                y ← rec (args!!1)
;                return (x + y)
```

m-lift

Clojure has a lift macro that supports any number of arguments – nice!

`(m-lift 2 fn)` is equivalent to

```
(fn [arg1 arg2] (domonad [x arg1
                          y arg2]
                         (fn x y)))
```

and `(m-lift 3 fn)` is equivalent to

```
(fn [arg1 arg2 arg3] (domonad [x arg1
                               y arg2
                               z arg3]
                              (fn x y z)))
```

*In Haskell, these are separate functions `liftM`, `liftM2`, `liftM3`, etc.

Exercise: Implement mapm

The mapm function is used to map a monadic function over a list of values. It returns its list of results in the monad. It's type would be:

(a -> b_{effect}) -> *list-of-a* -> (*list-of-b*)$_{\text{effect}}$

Hint: Clojure's `reduce` function is a left fold.

maybe-t (simplified)

```
(defn maybe-t
  “Monad transformer that transforms a monad m
   into a monad in which the base values can be
   invalid (represented by nil).”
  [m]
  (monad [m-result (with-monad m m-result)
          m-bind   (with-monad m
                     (fn m-bind-maybe-t [mv f]
                       (m-bind mv
                               (fn [x]
                                   (if (identical? x nil)
                                     (m-result nil)
                                     (f x))))))
          ]))
```

state-t (simplified)

```
(defn state-t
  "Monad transformer that transforms a monad m into a
   monad of stateful computations that have the base
   monad type as their result."
  [m]
  (monad [m-result (with-monad m
                     (fn m-result-state-t [v]
                       (fn [s]
                         (m-result [v s]))))
          m-bind   (with-monad m
                     (fn m-bind-state-t [stm f]
                       (fn [s]
                         (m-bind (stm s)
                                 (fn [[v ss]]
                                   ((f v) ss))))))
          ]))
```

Lifting Operations

Lifting is tedious and error-prone. The transformer stack must be managed carefully to avoid becoming unmaintainable.

```
; we will use a (state-t (cont-t (env-t error-m)))
; stack, abbreviated scee
(def scee (state-t (cont-t (env-t error-m))))

; helper functions for lifting from different points
; in the stack

(def lift-cont (lift-state-t
                  (cont-t (env-t error-m))))

(def lift-env (comp lift-cont
                    (lift-cont-t (env-t error-m))))

(def lift-error (comp lift-env lift-env-t))
```