```
class Monad m where
        return :: a -> m a
        (>>=) :: m a -> (a -> m b) -> m b
```

unde:

- `m` este un constructor de tipuri
`m a` — tipul computatiilor care produc rezultate de tip a
- tipul `a -> m b` este tipul continuarilor
Continuare: O functie care foloses,te un rezultat de tip `a` pentru a produce o computatie de tip `b`
- `(>>=)` este operatia de „secventiere” a computatiilor
- `return` este continuarea triviala
Pentru un `v` dat, produce computatia care va avea ca rezultat acel `v`.

Deci un monad in cod arata cam asa:

```
instance Monad Parser where
     return x = Parser (\s -> [(x,s)])
     m >>= k  = Parser (\s -> [(y,u)|
                          (x, t) <- apply m s,
                          (y, u) <- apply (k x) t ]
                       )
```

Monadul maybe

```
instance Monad Maybe where
    Just x  >>= k = k x
    Nothing >>= k = Nothing
    return x      = Just x
```

MyIO

```
module MyIO(MyIO, myPutChar, myGetChar, convert)
where

type Input     = String
type Remainder = String
type Output    = String

data MyIO a    =  MyIO (Input -> (a, Remainder,
                                  Output))

apply :: MyIO a -> Input -> (a, Remainder, Output)
apply (MyIO f) inp  =  f inp

myPutChar :: Char -> MyIO ()
myPutChar ch  =  MyIO (\inp -> ((), inp, [ch]))

myGetChar :: MyIO Char
myGetChar =  MyIO (\(ch:rem) -> (ch, rem, []))

instance Monad MyIO where
return x  =  MyIO (\inp -> (x, inp, ""))
m >>= k   =  MyIO (\inp ->
                   let (x, rem1, out1) = apply m
                     inp in
                   let (y, rem2, out2) = apply (k
                     x) rem1 in
                   (y, rem2, out1++out2))

convert :: MyIO () -> IO ()
convert m  =  interact (\inp ->
                let (x, rem, out) = apply m inp
                in out)
```

Parser

```
module Parser(Parser,apply,parse,char,spot,
  token,star,plus,parseInt) where

import Data.Char
import Control.Monad

-- The type of parsers
newtype Parser a = Parser (String -> [(a, String)])

-- Apply a parser
apply :: Parser a -> String -> [(a, String)]
apply (Parser f) s = f s

-- Return parsed value, assuming at least one
successful parse
parse :: Parser a -> String -> a
parse m s  =  one [ x | (x,t) <- apply m s, t ==
"" ]
               where
               one []                  =  error "no
parse"
               one [x]                 =  x
               one xs | length xs > 1  =  error
"ambiguous parse"

-- Parsers form a monad

--   class Monad m where
--     return :: a -> m a
--     (>>=) :: m a -> (a -> m b) -> m b

instance Monad Parser where
  return x  =  Parser (\s -> [(x,s)])
  m >>= k   =  Parser (\s ->
                 [ (y, u) |
                   (x, t) <- apply m s,
                   (y, u) <- apply (k x) t ])

-- Parsers form a monad with sums

--   class MonadPlus m where
--     mzero :: m a
--     mplus :: m a -> m a -> m a

instance MonadPlus Parser where
  mzero      =  Parser (\s -> [])
  mplus m n  =  Parser (\s -> apply m s ++ apply n
s)

-- Parse one character
char :: Parser Char
char =  Parser f
  where
  f []     =  []
  f (c:s)  =  [(c,s)]

-- guard :: MonadPlus m => Bool -> m ()
-- guard False  =  mzero
-- guard True   =  return ()

-- Parse a character satisfying a predicate (e.g.,
isDigit)
spot :: (Char -> Bool) -> Parser Char
spot p  =  do { c <- char; guard (p c); return c }
```

```
-- Match a given character
token :: Char -> Parser Char
token c  =  spot (== c)

-- Perform a list of commands, returning a list of
values
-- sequence :: Monad m => [m a] -> m [a]
-- sequence []
-- sequence (m:ms)  =  do {
--                        x <- m;
--                        xs <- sequence ms;
--                        return (x:xs)
--                      }

-- match a given string (defined two ways)
match :: String -> Parser String
match []      =  return []
match (x:xs)  =  do {
                   y <- token x;
                   ys <- match xs;
                   return (y:ys)
                 }
match' :: String -> Parser String
match' xs  =  sequence (map token xs)

-- match zero or more occurrences
star :: Parser a -> Parser [a]
star p  =  plus p `mplus` return []

-- match one or more occurrences
plus :: Parser a -> Parser [a]
plus p  =  do x <- p
              xs <- star p
              return (x:xs)

-- match a natural number
parseNat :: Parser Int
parseNat =  do s <- plus (spot isDigit)
               return (read s)

-- match a negative number
parseNeg :: Parser Int
parseNeg =  do token '-'
               n <- parseNat
               return (-n)

-- match an integer
parseInt :: Parser Int
parseInt =  parseNat `mplus` parseNeg
```

Parser folosit in evaluare de expresii

```
module Exp where

import Control.Monad
import Parser

data Exp = Lit Int
         | Exp :+: Exp
         | Exp :*: Exp
         deriving (Eq,Show)

evalExp :: Exp -> Int
evalExp (Lit n)    =  n
evalExp (e :+: f)  =  evalExp e + evalExp f
evalExp (e :*: f)  =  evalExp e * evalExp f

parseExp :: Parser Exp
parseExp   =  parseLit `mplus` parseAdd `mplus`
parseMul
  where
  parseLit = do { n <- parseInt;
                  return (Lit n) }
  parseAdd = do { token '(';
                  d <- parseExp;
                  token '+';
                  e <- parseExp;
                  token ')';
                  return (d :+: e) }
  parseMul = do { token '(';
                  d <- parseExp;
                  token '*';
                  e <- parseExp;
                  token ')';
                  return (d :*: e) }

test  :: Bool
test  =
  parse parseExp "(1+(2*3))" == (Lit 1 :+: (Lit
                                  2 :*: Lit 3)) &&
  parse parseExp "((1+2)*3)" == ((Lit 1 :+: Lit
                                  2) :*: Lit 3)
```