

## 第4章 モナド

モナド (monad) は、Haskell（あるいは他の関数型言語）で破壊的代入（変数の値など状態を変更すること）や入出力のような、他の言語では“副作用” (side effect) として実現される特徴を扱うための手法である。

もともとは数学のカテゴリー理論（圏論）で使われている言葉を借用したものであるが、Haskell で使用するときには、背景となるカテゴリー理論のことを知っている必要はない。

### 4.1 参照透明性と副作用

純粋な関数型言語には、式は値を表すためだけのものであり、「変数の出現はその定義（変数 = 式）の右辺の式で置き換えても全体の意味は変わらない」という性質がある。このような性質を 参照透明性 (referential transparency) と呼び、プログラムの“意味”を考察していく上でとても重要な性質である。（参照透明性のおかげで、帰納法などによる証明が容易になる。）一方、副作用は、式がその 副作用 のことである。式が副作用を持ちうるということは、その言語では参照透明性が成り立たない、ということである。Haskell の式は副作用を持っていない。

C のような言語では、入出力や破壊的代入を扱う部分では、副作用を使っているため参照透明性は成り立たない。例えば、C で、

```
1 c = getchar(); putchar(c); putchar(c); // C-code ①
```

と

```
1 putchar(getchar()); putchar(getchar()); // C-code ②
```

とは、`getchar()` が副作用を持っているために異なるプログラムである。（上のプログラムでは 1 文字、下のプログラムではキーボードから 2 文字の入力が消費される。）

一見、参照透明性と入出力や破壊的代入は相容れない性質のように見える。しかし、Haskell では次のように考える。

入出力や、変数などの状態の変更は、何らかの“アクション”である。副作用を持つ C 言語などの関数に対応する Haskell の関数は、このアクションを含めて戻り値として返す関数として考える。（つまりアクションを“副”作用ではなく、一人前の値として扱う。）この“アクション”の型は、単なる値とは異なる型を持っていて、アクションに対応している文脈でしか使用することができない。従って、アクションと値は区別され、参照透明性が保たれる。

例えば、C 言語の `getchar`, `putchar` に対応する Haskell の関数は、`getChar`, `putChar` という名前で、それぞれ次のような型を持っている。

```
1 getChar ::
2
3 putChar ::
```

この `IO` という型構成子が入出力に関するアクションの型を表す。また、`()` はユニット型と読み、意味のある値を持たない型である。（C 言語の `void` 型のようなものである。）そもそも、C 言語の `putchar(getchar())` という式に対応する `putChar getChar` のような Haskell の式は、型エラーになってしまう。`IO` 型に用意されている演算子「`>>=`」（パインド、と読む）を用いて、次のように書かなければいけない。

```
1 getChar >>= (\ c -> putChar c)
```

ここで、`(>>=)` の型は `IO a -> (a -> IO b) -> IO b` である。（後述するように、実際にはより一般的な型を持っている。）「`>>=`」は、左オペランド `IO a` の `a` の部分を抽出して、右オペランドの関数 `a -> IO b` に引数として渡す、という役割と、左オペランドのアクションと右オペランドのアクションを続けて実行する、という役割と、2 つの役割を果たす。この式では、`getChar` というアクションの結果得られる `Char` 型の値が、`c` という変数に束縛され、続いて `putChar c` というアクションが実行される。アクションの型 (`IO Char`) を持つ式から `Char` 型の値だけを抽出するには、「`>>=`」のような演算子を介する必要がある。`Char` 型の変数 `c` に対して、`c = getChar` と書けるわけではないので、型の面からも `c` が `getChar` と置き換えられないことが明白である。

**Q 4.1.1** C-code ① と C-code ② に対応する Haskell プログラム（の断片）を `getChar`, `putChar`, 「`>>=`」, 「`>>`」を使って書け。ただし「`>>`」は次のように定義される演算子である。

```
1 (>>) :: IO a -> IO b -> IO b
2 m >> n    = m >>= (\ _ -> n)
```

---

---

---

---

---

副作用を持つプログラムは実行する順番により意味が変わるので、並列に実行されるプログラムでは問題となることがある。参照透明性を保ちながらアクションを扱えることは並列実行しても意味が変わらないので理論的に有用であるだけでなく、並列コンピューティングなど実用面でも必要となってきた。

## 4.2 モナドとは

このような、さまざまな言語で副作用として実現される特徴（入出力・破壊的代入・例外処理・非決定性など）に対するアクションの型が、実は共通の構造を持っていて、同じ構造を持つ演算子で取り扱えることがわかっている。この共通の構造を持つ型（正確には型構成子）のことを            (monad) という。つまり、モナドはアクションの型である。上記の `IO` もモナドである。ただし、モナドの中にはアクションという言葉がふさわしくないものもあるので、以下では代わりに“計算” (computation) という言葉を使う。

具体的にはモナドとは

```
return :: a → M a
(>>=) :: M a → (a → M b) → M b
```

という型の関数の存在する型構成子  $M$  のことである。より厳密には、`return`, `(>>=)` が

```
(return a) >>= k      = k a
m >>= (\ a → return a) = m
(m1 >>= k1) >>= k2    = m1 >>= (\ a → (k1 a >>= k2))
```

の3つの等式 (monad law) を満たす必要がある。これらの等式は、直感的には `return` が `(>>=)` の“単位元”であること、`(>>=)` が結合則を満たすこと、を示している。つまり、数に対する、加算 (+) と加算の単位元 0 の次の等式に相当する。

```
0 + x      = x
x + 0      = x
(x + y) + z = x + (y + z)
```

直観的には  $M a$  という型は、           または“ $a$  のアクション”の型を意味する。また、`return` と `(>>=)` の直観的な意味は次の通りである。

- `return a` … 値  $a$  をそのまま返す **何もしない**（アクションが実質ない）計算を表す。
- `m >>= k` … まず、 $m :: M a$  を計算し、その結果のアクションを除いた値の部分を関数  $k :: a \rightarrow M b$  に渡して、続けて計算することを表す。また、アクションは  $m, k$  のアクションを続けて実行するアクションに相当する。

## 4.3 モナドと型クラス

モナド  $M$  の定義は模倣したい副作用により異なるし、付随する関数 `return`, `(>>=)` の定義ももちろん異なる。しかし、`return`, `(>>=)` は  $M$  をパラメータとして、決まった型を持つので、型クラスを用いてアドホック多相な関数として定義することができる。

```
1 class Functor m where
2   fmap :: (a -> b) -> m a -> m b
```

```

3
4 class Functor m => Applicative m where
5     pure :: a -> m a
6     (<*>) :: m (a -> b) -> m a -> m b
7
8 class Applicative => Monad m where
9     return :: a -> m a
10    return = pure -- return のデフォルトの実装
11    (>>=) :: m a -> (a -> m b) -> m b

```

実際は、Monad は型ではなく、型構成子に対する型クラス（型構成子クラス, type constructor class）になっている。なお、上記3つのクラスのメソッドのうち fmap, (<\*>), return については、Monad のインスタンスであれば、常に同じ定義（下記）を使うので、以降では省略し pure と (>>=) の定義のみ示すことにする。なお pure と return はモナドの場合は同じものになるが、モナドでない型構成子でも有用な場合があるため、Monad のスーパークラスの Applicative で定義されている。

```

1  -- liftM と ap は Control.Monad で定義されている
2  liftM f x = x >>= (\ a -> return (f a))
3  ap f x = f >>= (\ g ->
4             x >>= (\ a -> return (g a)))
5
6  fmap = liftM
7  f <*> x = ap f x
8
9  -- return はデフォルトの実装が pure と定義されている。
10 return = pure

```

## 4.4 IO モナド

IO は Haskell の Prelude（最初から読み込まれているライブラリー）に入っているモナドである。型クラス Monad のインスタンスである。その定義は一般のプログラマーからは見えない組込みの型となっている。ただし、直観的には次のような定義を持つ型だと考えることができる。

```

-- type IO a ≡ _____
--
-- instance Applicative IO where
-- -- 注：実際には type で定義された型の別名を instance には
-- 使えない。
--     pure a = \ w -> (a,w)
--     (<*>) = ap
--
-- instance Monad IO where
--     m >>= k = \ w -> let (a,w1) = m w in k a w1

```

ただし RealWorld は、コンピューター全体のファイルなどの状態を表す型である。IO は関数の型だが、引数の RealWorld 型のデータは隠されていて、他の計算で使われないことが保証できるため、破壊的に書き換えて、戻り値の RealWorld 型のデータのために使っても良い、ということである。

IO は Haskell で入出力や状態を効率的に扱うために、処理系で特別な扱いを受ける。主なプリミティブとして、putChar, getChar の他、以下のような関数

がある。

```
1 putStr    :: String -> IO () -- 文字列を出力する
2 putStrLn :: String -> IO () -- 文字列を出力して改行する
3
4 getLine   :: IO String      -- 一行を読み込む
5 getContents :: IO String    -- EOF まで読み込む
6
7 print     :: Show a => a -> IO ()
8           -- 文字列に変換して出力し、改行する
9 readLn   :: Read a => IO a
10          -- 一行を読み込んでパースする
```

特に `getContents` は遅延評価で標準入力の内容を読み込む。言い換えれば、必要にならなければ入力を求めない。だから、最後 (EOF) まで入力していなくても、必要になったところまで入力が完了していれば処理が進む。これらの他に、標準入出力ではなくファイルに対して入出力するための関数なども用意されている。

ところで、Haskell のプログラムを、GHCi のような対話環境ではなく、`ghc` コマンドで実行可能ファイルにコンパイルして実行するとき、C と同じように `main` という名前の関数から実行が開始される約束になっている。そして `main` 関数は、`IO t` という形の型 (`t` は任意の型) を持たなければいけないことになっている。

つまり次のようなプログラムでは、標準入力のすべてを読み込むことはなく、最初の 1 行のみを出力する。

```
1 main :: IO ()
2 main = getContents >>=
3       (\ all -> let ls = lines all
4                 in putStrLn (head ls))
```

ただし、`lines :: String -> [String]` は文字列を改行文字で分割する関数、`head :: [a] -> a` はリストの先頭要素を返す関数である。例えば、`lines "ab\ncdef\nxyz"` は `["ab", "cdef", "xyz"]` である。

たとえば、標準入力から読み込んだ文字列の大文字を小文字に変換したものと小文字を大文字に変換したものを出力するプログラムは以下のようなになる。

```
1 import Data.Char    -- Data.Charモジュールを import する
2
3 -- toLower, toUpper :: Char -> Char
4 --   は大文字・小文字の変換関数
5 main :: IO ()
6 main = getContents >>=
7       (\ s -> putStr (map toLower s
8                       ++ map toUpper s))
```

上記の定義は括弧を省略し、さらにレイアウトを整えて、次のように書くことが多い。

```
1 main = getContents >>= \ s ->
```

```
2 putStr (map toLower s ++ map toUpper s)
```

以降のプログラムでは、このように括弧を省略する。

実は、Haskell では Monad クラスに対して、do 記法という糖衣構文を用意していて、この関数は次のように書くこともできる。

```
1 main = do { s <- getContents;
2           putStr (map toLower s ++ map toUpper s) }
```

この **do** 式も第 A 章で紹介したレイアウトルールの対象であり、適切にレイアウトすればブレースやセミコロンを省略することができる。

```
1 main = do s <- getContents
2           putStr (map toLower s ++ map toUpper s)
```

そして **do** 式は次のルールで翻訳される。（下線部に翻訳規則を再帰的に適用していく。）ここで  $e$  は式、 $decls$  は宣言の並びである。

```
do { e }                ⇒ e
do { e; stmts }          ⇒ e >>= \ _ -> do { stmts }
do { x <- e; stmts }      ⇒ e >>= \ x -> do { stmts } (注)
do { let decls; stmts } ⇒ let decls in do { stmts }
```

実は、この翻訳規則はリストの内包表記の翻訳規則とほとんど同じである。また、（注）の  $x$  のところには、実は変数だけではなく、パターンも可能である。その場合の翻訳規則は正確には次のようになる。

```
do { p <- e; stmts } ⇒ let ok p = do { stmts }
                        ok _ = fail "..."
                        in e >>= ok
```

**Q 4.4.1** つぎのようなプログラムを上記の IO に関する関数を用いて定義せよ。

**ヒント:** 入力には `getContents` ではなく、`getLine` を使用せよ。

1. 一行だけ行を読み込んで、それをオウム返しするプログラム
2. 一行だけ行を読み込んで、それを 2 回オウム返しするプログラム
3. 整数を読み込んで、その 3 倍を出力するプログラム
4. 正の整数を読み込んで、その数を辺の長さとして、\* で三角形を表示するプログラム

---



---



---



---



---

なお、**for** 文などの繰り返しのための構文はないので、繰り返しが必要なときは再帰関数を定義する必要がある。例えば、入力文字列の各行の先頭の単語だけを大文字にするプログラムは `map` を使って次のように定義することができる。

```
1 import Data.Char
2
3 capitalizeFirst [] = []
4 capitalizeFirst (xs:xss) = (map toUpper xs) : xss
5
6 capitalizeLine line
7     = unwords (capitalizeFirst (words line))
8
9 main = do all <- getContents
10         let xs = lines all
11             ys = map capitalizeLine xs
12             putStr (unlines ys)
```

**問 4.4.2** IO モナドを利用して次のようなプログラムを作成せよ。（あとの節で紹介しているライブラリー関数を使用するかもしれない。）

1. 入力文字列を行ごとに大文字と小文字に変換して出力する。（例えば、「Hello,«World«」が「hello,«HELLO,«world«WORLD」になる。ただし、「«」は改行文字を表すとする。）
2. 入力文字列中の数字 ('0' ~ '9') の出現回数をカウントして出力する。
3. 入力文字列中の '@' が出現する最初の 10 行だけを出力する。

## 4.5 参照型と可変な配列型

さらに、`Data.IORef` というモジュールを `import` することで、破壊的代入が可能な参照 (reference) 型 (`IORef`) を扱う関数も用意される。

```
1 -- import Data.IORef が必要
2
3 newIORef      :: a -> IO (IORef a)      -- 新しい参照の作成
4 readIORef    :: IORef a -> IO a        -- 参照の読出し
5 writeIORef    :: IORef a -> a -> IO ()  -- 参照への書込み
```

次に、`IORef` を利用したプログラムの例をあげる。

```
1 import Data.IORef
2
3 foo r = do i <- readIORef r
4           if i <= 0 then return ()
5           else do
6               putStrLn (show i)
7               writeIORef r (i - 1)
8               foo r
9
10 main = do r <- newIORef 10
```

```
11 | foo r
```

ただし、これは無理矢理 `IORef` を使った例であり、`IORef` を使わずに同じ動作をする次のようなプログラムのほうが自然である。

```
1 {- foo の自然な定義は以下の通り -}  
2 foo i = if i <= 0 then return ()  
3         else do putStrLn (show i)  
4                 foo (i - 1)  
5  
6 main = foo 10
```

また `Data.Array.IO` というモジュールを `import` することで、破壊的代入が可能な配列を扱う関数も用意される。

そもそも破壊的代入を使わない配列の操作は、配列をいちいちコピーすることになって大きな配列では極めて非効率的である。例えば `arr1 :: Array Int Int` のような配列があるとする。この `arr1` の 3 番目の要素を 7 にした配列を得るためには、`arr1 // [(3,7)]` のように書くが、参照透明性を保つためには元の配列 `arr1` は変化してはいけないので、この式は新しい配列を生成する必要がある。例えば、次のプログラムを考える。

```
1 import Data.Array -- Haskell の immutable な配列の型  
2  
3 -- C での int arr[5] = {1,2,3,4,5}; に相当する配列を作る  
4 arr1 :: Array Int Int  
5 arr1 = array (0, 4)  
6         [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5)]  
7 -- arr1 の 3 番目の要素を 7 にした新しい配列を作る  
8 arr2 = arr1 // [(3,7)]  
9  
10 main = do print (arr1 ! 3) -- arr1 の 3 番目を出力 (4)  
11          print (arr2 ! 3) -- arr2 の 3 番目を出力 (7)
```

この場合、`arr2` は 3 番目の要素以外は `arr1` からコピーした新しい配列である。

この例のように要素数が 5 など小さい配列であれば、これでも良いが、要素数が非常に大きい配列では、要素を 1 つ書き換えるために全要素をコピーするのは非効率である。要素数の大きな配列には破壊的代入は実質的に不可欠である。そこで、`Data.Array.IO` というモジュールを使えば、参照透明性を保ちつつ破壊的代入を行うことができる。

```
1 -- import Data.Array.IO が必要  
2  
3 newArray    :: Ix i => (i, i) -> e -> IO (IOArray i e)  
4 -- 添字の最小・最大の組と要素の初期値を受け取って  
5 -- 新しい配列を作る  
6 getBounds   :: Ix i => IOArray i e -> IO (i, i)  
7 -- 配列の添字の最小と最大を返す  
8 readArray   :: Ix i => IOArray i e -> i -> IO e  
9 -- 配列と添字を受け取って、その要素を読む  
10 writeArray  :: Ix i => IOArray i e -> i -> e -> IO ()
```

## 11 -- 配列と添字と値を受け取って、要素に書き込む

次に、`Data.Array.IO` を利用したプログラムの例をあげる。ここで `randomArray` は乱数で初期化した配列を作成する関数、`printArray` は配列の要素を出力する関数、`accumArray` は配列を破壊的に書き換えて足し算をする関数である。

```
1 {-# LANGUAGE FlexibleContexts #-}
2 import Data.Array.IO
3 import System.Random
4
5 randomArray ::
6   Int -> Int -> Int -> Int -> IO (IOArray Int Int)
7 randomArray i1 i2 mn mx = do
8   arr <- newArray (i1,i2) mn -- 仮に mn で初期化する
9   loop i1 i2 mn mx arr
10  return arr
11 where
12   loop i i2 mn mx a
13     | i > i2 = return ()
14     | otherwise = do r <- randomRIO (mn, mx)
15                     writeArray a i r
16                     loop (i + 1) i2 mn mx a
17
18 printArray :: Show a => IOArray Int a -> IO ()
19 printArray arr = do
20   (i1,i2) <- getBounds arr
21   loop i1 i2 arr
22 where
23   loop i1 i2 arr | i1 > i2 = return ()
24                 | otherwise = do
25   a <- readArray arr i1
26   putStr (show a)
27   putStr ", "
28   loop (i1 + 1) i2 arr
29
30 accumArray arr = do
31   (i1, i2) <- getBounds arr
32   loop i1 i2 0 arr
33 where
34   loop i1 i2 v arr | i1 > i2 = return ()
35                   | otherwise = do
36   a <- readArray arr i1
37   let w = a + v
38   writeArray arr i1 w
39   loop (i1 + 1) i2 w arr
40
41 main = do arr <- randomArray 1 5 0 100
42          printArray arr
43          putStrLn ""
44          accumArray arr
45          printArray arr
46          putStrLn ""
```

冒頭の `{-# LANGUAGE FlexibleContexts #-}` はプラグマというコンパイラーに与える指令で、ここでは詳細には立ち入らないが、`Data.Array.IO` を使うときは付けておくことをお勧めする。

このプログラムを実行すると、例えば次のような出力が得られる。

```
56, 36, 76, 44, 10,  
56, 92, 168, 212, 222,
```

(ほぼ) 対応する C 言語のプログラムは以下のようになる。

```
1 #include <stdio.h>  
2 #include <stdlib.h>  
3 #include <time.h>  
4  
5 int* randomArray(int n, int min, int max) {  
6     srand((unsigned)time(NULL));  
7     int* arr = (int*)malloc(n * sizeof(int));  
8     for (int i = 0; i < n; i++) {  
9         arr[i] = min + rand() % (max - min + 1);  
10    }  
11    return arr;  
12 }  
13  
14 void printArray(int n, int* arr) {  
15     for (int i = 0; i < n; i++) {  
16         printf("%d, ", arr[i]);  
17     }  
18 }  
19  
20 void accumArray(int n, int* arr) {  
21     int v = 0;  
22     for (int i = 0; i < n; i++) {  
23         v += arr[i];  
24         arr[i] = v;  
25     }  
26 }  
27  
28 int main(void) {  
29     int n = 5;  
30     int* arr = randomArray(n, 0, 100);  
31     printArray(5, arr);  
32     printf("\n");  
33     accumArray(5, arr);  
34     printArray(5, arr);  
35     printf("\n");  
36  
37     return 0;  
38 }
```

## 4.6 続・有用なリスト処理関数

モナドとは直接関係ないが、以前紹介した以外に文字列処理プログラムで有用と思われるリスト処理関数を、以下でいくつか紹介する。これらの関数は `sort` を除いて `Prelude` (標準ライブラリー) に定義済みである。

```
1 -- lines は文字列を改行文字のところで分割して、文字列 (単語)  
2 -- のリストにする  
3 -- 結果の文字列には改行文字は含まれない  
4 -- > lines "ab\ncdef\nxyz" == ["ab","cdef","xyz"]  
5 lines      :: String  -> [String]  
6  
7 -- words は文字列を空白文字で分割して、文字列のリストにする  
8 -- > words "ab cdef xyz" == ["ab","cdef","xyz"]  
9 words      :: String  -> [String]
```

```

10
11 -- unlines は lines の逆操作である
12 -- 各文字列の末尾に改行文字を追加し、一つに結合する
13 -- > unlines ["ab","cdef","xyz"] == "ab\ncdef\nxyz\n"
14 unlines      :: [String] -> String
15
16 -- unwords は words の逆操作である
17 -- 各文字列を空白で区切って結合する
18 -- > unwords ["ab","cdef","xyz"] == "ab cdef xyz"
19 unwords      :: [String] -> String
20
21 -- dropWhile p xs は takeWhile p xs の残りの末尾部分
22 -- を返す
23 -- > dropWhile (< 3) [1,2,3,4,5,1,2,3]
24 ---          == [3,4,5,1,2,3]
25 -- > dropWhile (< 9) [1,2,3] == []
26 -- > dropWhile (< 0) [1,2,3] == [1,2,3]
27 dropWhile    :: (a -> Bool) -> [a] -> [a]
28
29 -- any は述語とリストを受け取り、リストのなかのどれかの要素が
30 -- 述語を満たすかどうか判定する
31 -- > any (< 9) [1,2,3] == True
32 -- > any (< 2) [1,2,3] == True
33 -- > any (< 0) [1,2,3] == False
34 any          :: (a -> Bool) -> [a] -> Bool
35
36 -- all は述語とリストを受け取り、リストのなかのすべての要素が
37 -- 述語を満たすかどうか判定する
38 -- > all (< 9) [1,2,3] == True
39 -- > all (< 2) [1,2,3] == False
40 -- > all (< 0) [1,2,3] == False
41 all          :: (a -> Bool) -> [a] -> Bool
42
43 -- head はリスト（空ではいけない）の先頭要素を取り出す
44 -- > head [1,2,3] == 1
45 head         :: [a] -> a
46
47 -- last は（非空かつ有限な）リストの最後の要素を取り出す
48 -- > last [1,2,3] == 3
49 last         :: [a] -> a
50
51 -- tail はリスト（空ではいけない）の先頭要素を除いた末尾部分
52 -- を取り出す
53 -- > tail [1,2,3] == [2,3]
54 tail         :: [a] -> [a]
55
56 -- elem はリストの要素（element）として含まれているか否か
57 -- を判定する
58 -- 通常 x `elem` xs のように中置記法で用いることが多い
59 -- > 1 `elem` [1,2,3] == True
60 -- > 0 `elem` [1,2,3] == False
61 elem         :: (Eq a) => a -> [a] -> Bool
62
63 -- import Data.List が必要である
64 -- sort は安定なソートアルゴリズムの実装である
65 -- 比較関数を引数にとる sortBy 関数もある
66 -- > sort [5,2,3,8] == [2,3,5,8]
67 sort         :: (Ord a) => [a] -> [a]
68
69 -- sequence はリスト中のアクションを順に実行する。
70 sequence     :: Monad m => [m a] -> m [a]
71 sequence []   = return []
72 sequence (m:ms) = m          >>= \ a ->

```

```
73         sequence ms >>= \ as ->
74         return (a:as)
75
76 -- mapM はアクションを返す関数をリスト中の全ての要素に
77 -- 適用する
78 -- ex.) mapM putStrLn ["ab", "xyzw", "12345"]
79 mapM      :: Monad m => (a -> m b) -> [a] -> m [b]
80 mapM f as  = sequence (map f as)
```

**Q 4.6.1** 次の式の値は何か?

1. `dropWhile odd [1,4,5]`

2. `any (> 0) [-1,-3,-5]`

3. `3 `elem` [2,5,3,1]`

## 4.7 さらに詳しく知りたい人のために...

文献 (Wadler 1990) は、モナドとリストの内包表記の関係について解説している。

[1] **(Wadler 1990)** Philip Wadler, "Comprehending Monads" ACM Conference on Lisp and Functional Programming, Nice (France), 1990 年 6 月