■ 連載

Haskell プログラミング

自分自身を出力するプログラム

尾上 能之（東京大学情報基盤センター）
onoue@ecc.u-tokyo.ac.jp

Quine

ハッカーの間で有名なお遊びとして，出力が自分自身のソースコードと同一になるプログラムを作成するというものがある．ただしルールとして，実行時に入力をとるものは認められていない．たとえばファイル入力を介してプログラム自身のソースコードを実行時に参照できてしまうと，簡単に処理が行えてしまうためである．また多くの言語において文字を 1 つも含まない空のプログラムも許されるが，ここではそういったトリビアルなものは扱わないものとする．

このようなプログラムは，間接自己参照に関し発展的な研究を行ったアメリカの哲学者 Willard van Orman Quine の名前をとって Quine[6] と呼ばれる．Quine は 1970 年代[1] から各種のプログラミング言語に対して作成されており，ソースコードの短さや言語固有のオリジナリティによって選りすぐられた作品が Web 上に公開されている[5), 7)]．

Haskell 版の Quine もすでに知られているが，ここではその実現方法を一から考えてみることにしよう．

Haskell 版

Haskell では文字列を出力する基本的な関数は putStr なので，最初に以下のようなプログラムを思いつくかもしれない．

```
main = putStr "main = putStr "main = putStr ..""
```

見て分かるように，下線を引いた部分が再帰的に繰り返されるためこのままではプログラムを記述できない．ちょうど自分の影を踏もうとしていつまでも追い駆け回っている状態とでも言うべきだろうか．

この処理を 1 ステップで達成するのは難しいため，最初のステップとして出力すべき文字列を変数に保存し，次のステップとしてそれを元に最初の処理を行う部分とその出力処理を表す部分をまとめて出力する 2 段階処理で実現することにする．

```
q = 文字列A ; main = putStr ( 文字列B ++ ";" ++ 文字列C )
```

まず**文字列 C** は，セミコロン以降の第 2 ステップを表すため，

```
文字列C = "main = putStr ( 文字列B ++ ";" ++ 文字列C )"
```

としたいのだが，これでは先ほどと同じ再帰の問題に陥りうまくいかない．そこで新たに導入した変数 q を用いて，**文字列 C** を q の参照で置き換えることにする．

```
q = 文字列A ; main = putStr ( 文字列B ++ ";" ++ q)
```

すると変数qは自ずとmain = putStr 文字列Dのような形になることが分かる．

```
q = "main = putStr 文字列D " ; main = putStr ( 文字列B ++ ";" ++ q)
```

次に**文字列B**であるが，これは第1ステップである変数への代入に相当するため，これをqを使って表すと"q = " ++ show qとなる．ここで，関数showを用いてqの値をダブルクォートで囲んでいることに注意されたい．

```
q = "main = putStr 文字列D " ; main = putStr ("q = " ++ show q ++ ";" ++ q)
```

最後に**文字列D**は，第2ステップのputStrの引数に対応させればよいので，"q = " ++ show q ++ ";" ++ qとなることが分かる．あとは文字列の中でダブルクォートを表す場合はバックスラッシュでエスケープする必要があるのと，空白の調整，putStrの関数適用の括弧を$で置き換えた結果，以下のHaskell版Quineプログラムを得ることができる．

```
q="main=putStr$\"q=\"++show q++\";\"++q";main=putStr$"q="++show q++";"++q
```

Haskellでは，定義は順番に評価するのではなくmainから必要に応じて評価される性質を活かし，文字列の定義と実行部の順序を逆にしてもよい．これにより，従来第2ステップで行っていた"q="と";"の連結処理を変数qの中に移動するとさらに文字数を減らすことができる．

［Haskell版Quine］☆1

```
main=putStr$q++show q;q="main=putStr$q++show q;q="
```

この50文字のプログラムが，最小文字数のHaskell版Quineとして知られている．

他の言語の場合

■ Perl, Python

Haskell版と近い考え方を用いると，Perl, Pythonなどのスクリプト言語でもQuineを作成することができる．

［Perl版Quine］

```
$q=q(print"\$q=q($q);$q");print"\$q=q($q);$q"
```

［Python版Quine］

```
q='q=%s;print q%%`q`';print q%`q`
```

Python版は式をバッククォートで囲むとrepr表現となり，文字列の場合ダブルクォートで囲まれる．qの定義において%rを用いるともう少しコードを短くすることも可能である．

■ Lisp, Scheme, OCaml

またこれとは少し違った考え方のものとしてLispにおけるQuineもよく知られている．Lispではプログラムや文字列はすべてS式で表現されるため文字列を変数に保存しなくてもよく，ラムダ式で自分自身を再帰的に呼び出すことでQuineが実現可能である．

☆1 このプログラムはhttp://www.sampou.org/haskell/ipsj/から取ることができる．

［Lisp, Scheme 版 Quine］（読みやすいよう整形済）

```
((lambda (x)
   (list x (list (quote quote) x)))
 (quote
  (lambda (x)
    (list x (list (quote quote) x)))))
```

クォートによるマクロ記法に慣れた読者には，((lambda (x) `(,x ',x)) '(lambda (x) `(,x ',x))) の方が読みやすいかもしれない[☆2]．これはちょうど遺伝子を用いて自己再製する構造とみなすことができる．説明の都合上マクロ記法による表現を用い，((lambda (x1) `(,x2 ',x3)) '(lambda (x4) `(,x5 ',x6))) のように変数 x に 1 ～ 6 の添字を振って区別することにすると，(lambda (x1) `(,x2 ',x3)) は親で，'(lambda (x4) `(,x5 ',x6)) は遺伝子に対応する．親はラムダ変数 x1 で遺伝子を取り込み，,x2 で子の体を ',x3 で遺伝子のコピーを作り結果が S 式となるよう `() で囲んで返す．親は自分から設計図を作ることができないため，遺伝子のコピーを代々引き渡していく必要があるのである．

Lisp 版と同じような発想で OCaml 版も作成できる．遺伝子のコピーの際に %S を用いることで，全体をダブルクォートで囲み必要なエスケープ処理を行っているのがミソである．

［OCaml 版 Quine］

```
(fun s -> Printf.printf "%s %S" s s) "(fun s -> Printf.printf \"%s %S\" s s)"
```

OCaml 版と同様の手法で Haskell 版 Quine も実現できる．ただ関数 main から起動できないためインタープリタで動かす必要があるので注意されたい．

```
$ hugs
Hugs.Base> (\ s -> putStr(s++" "++show s)) "(\\ s -> putStr(s++\" \"++show s))"
(\ s -> putStr(s++" "++show s)) "(\\ s -> putStr(s++\" \"++show s))"
Hugs.Base>
```

■ C

利用者が多い C 言語では，その豊富な機能を活かしてさまざまなアプローチの Quine が作成されている．ここでは ANSI などの標準規格には準拠していないが，比較的短く理解しやすいものを紹介しておこう．

［C 版 Quine］

```
main(a){printf(a,34,a="main(a){printf(a,34,a=%c%s%c,34);}",34);}
```

行っていることは以下と同等であり，文字列 a における %c をダブルクォート文字（文字コード 34）で，%s を文字列 a 自身で置き換えて出力している．

```
main() {
    char a[]="main(a){printf(a,34,a=%c%s%c,34);}";
    printf(a,34,a,34);
}
```

ポイントは，変数 a の値代入処理を printf の第 3 引数で値の参照と同時に行い，文字数を減らした点にある．また a は第 1 引数でも参照しており後方参照となるため，main(a) における a の宣言は必須となる．

■ PostScript

これまでに紹介したものとはまったく異なる例として，PostScript の Quine も紹介しよう．PostScript は Adobe が開

☆2 処理系によっては，実行時にクォートやカンマが関数 quote,unquote に置換され Quine とならない場合がある．

発したページ記述言語で，印刷の際ファイルをプリンタに送る形式の 1 つであり，現在広く用いられている PDF の原形となっている．またプログラミング言語としての特徴は，プログラムはバイナリではなくテキストファイルで記述され，スタックベースの後置記法で各コマンドが並べられている．ではさっそく PostScript 版 Quine を見てみることにしよう．

［PostScript 版 Quine］

```
(dup == =)
dup == =
```

文字列は (This is a string) のように括弧で囲んで表現され，dup はスタックの先頭を複製する命令，== はスタックの先頭を取り出しそのもの自身の文字列表現を出力する命令，= はスタックの先頭を取り出しその値の文字列表現を出力する命令となっている．これによりスタックの先頭にある文字列を s とすると，== では s がそのまま，= では s を囲む括弧を取り除いた中身が出力される．これを PostScript のインタープリタの 1 つであるツール Ghostscript で実行すると以下のようになり，Quine として動作することが分かる．

```
$ cat quine.ps
(dup == =)
dup == =
$ gs -q -dBATCH quine.ps
(dup == =)
dup == =
```

コマンド	スタック	出力
(dup == =)	(dup == =)	
dup	(dup == =) (dup == =)	
==	(dup == =)	(dup == =)
=		dup == =

ここでは，オプション -q で起動メッセージの出力を抑え，-dBATCH で実行後すぐインタープリタを終了している．

メタプログラミング

見てきたように，Lisp などプログラム自分自身を計算対象のデータとして扱うことができる言語においては Quine を容易に実現することができる．このようにプログラムの中でプログラムコードを対象として扱うパラダイムをメタプログラミングと呼び，その実装手法として以下のものが知られている．

[C++, Java] ： 総称型 (generic type)
[Lisp, Scheme] ： マクロ (クォート，バッククォート，カンマ)
[Haskell] ： Template Haskell[4)]

Monad とカテゴリの関係

前章ではプログラムの中でプログラムをメタに扱う例を紹介したが，Haskell において重要な役割を果たしている Monad についてもこの見方を適用することができる．Monad はすでに本連載の中でも紹介しているが，本来カテゴリ論のモナドを語源としており，両者の関係を明らかにすることで Monad の理解がより深まることが期待される．そこでここからは Monad とカテゴリとの関係を見ていく．なおカテゴリ論のモナドと区別するため，Haskell の Monad は英字で表すことにする．

カテゴリ論

本節では，Kleisli カテゴリならびにそのために必要な用語について説明し，これがプログラムの計算モデルに対応

することを示す．始めに，集まりとその上の演算を組にし対象となる世界を表す例としてモノイドを紹介する．

定義 1　集合 M 上に 2 項演算子・と要素 e が存在し以下の性質を満たすとき，$(M, \cdot, e)$ の 3 つ組を**モノイド (monoid)** という．

結合則　$(x \cdot y) \cdot z = x \cdot (y \cdot z) \quad \forall x, y, z \in M$
単位元　$x \cdot e = e \cdot x = x \quad \forall x \in M$

たとえば，0 を含む自然数の集合を$\mathcal{N}_0$ とすると，足し算は結合則が成り立ち単位元 0 が存在するため $(\mathcal{N}_0, +, 0)$ はモノイドとなる．掛け算を元にした $(\mathcal{N}_0, \times, 1)$ も同様にモノイドである．また Haskell のリストの集合を `List` とすると，`(List, (++), [])` もモノイドとなる．

2 つのモノイド $(M, \cdot, e), (M', \cdot', e')$ に対し関数 $f : M \to M'$ が存在し，$f(e) = e'$, $f(x \cdot y) = f(x) \cdot' f(y)$ が成り立つとき，f を *monoid homomorphism* という．例として，リストの長さを求める関数 `length` は `(List, (++), [])` から $(\mathcal{N}_0, +, 0)$ への *monoid homomorphism* となっている．

次にカテゴリとは，物事を「対象」とその間の「射」の集まりとして一般化した概念である．

定義 2　カテゴリ **(category)** $\mathcal{C}$ は以下の要素から構成される．

1. **対象 (object)** の集まり．$\mathrm{Obj}(\mathcal{C})$
2. **射 (arrow, morphism)** の集まり．射 f は**ドメイン**となる対象 $dom\, f$ を**コドメイン**となる対象 $cod\, f$ に写す概念で，$dom\, f = A$, $cod\, f = B$ のとき $f : A \to B$ または $A \xrightarrow{f} B$ のように書く．またドメイン A からコドメイン B へのすべての射の集まりを $\mathcal{C}(A, B)$ と書く．
3. **合成演算**．任意の 2 つの射 f, g で $cod\, f = dom\, g$ であるものに対し，合成射 $g \circ f : dom\, f \to cod\, g$ が存在し，以下の結合則を満たす．
   任意の f, g, h ただし $A \xrightarrow{f} B, B \xrightarrow{g} C, C \xrightarrow{h} D$ に対し $(h \circ g) \circ f = h \circ (g \circ f)$
4. **恒等射**．任意の対象 A に対し，恒等射 $\mathrm{id}_A : A \to A$ が存在し，以下の恒等則を満たす．
   任意の $f : A \to B$ に対し $\mathrm{id}_B \circ f = f$ ならびに $f \circ \mathrm{id}_A = f$

たとえば，集合 1 つ 1 つを対象とし集合上の写像関数を射ととらえることで集合の集まりはカテゴリとみなせる．また各モノイドを対象とし *monoid homomorphism* を射ととらえることでモノイドの集まりもカテゴリとみなせる．他にもカテゴリの性質をみたす集まりは多数存在し，このような抽象化によってそれらの間に潜む共通の性質を統一的に扱おうとするのがカテゴリ論の狙いである．

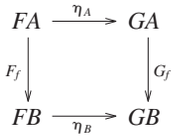
カテゴリは対象と射から構成されるが，1 つメタな立場に立ちカテゴリ自身を対象とするカテゴリも考えられ，このとき射に対応するものが関手となる．

定義 3　カテゴリ $\mathcal{C}, \mathcal{D}$ に対し，$\mathcal{C}$ から $\mathcal{D}$ への写像 F で，$\mathcal{C}$ の対象 A を $\mathcal{D}$ の対象 $F(A)$ に，$\mathcal{C}$ の射 $f : A \to B$ を $\mathcal{D}$ の射 $F(f) : F(A) \to F(B)$ に写し，以下の性質を満たすものを**関手 (functor)** F という．

1. $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$
2. $F(g \circ f) = F(g) \circ F(f)$

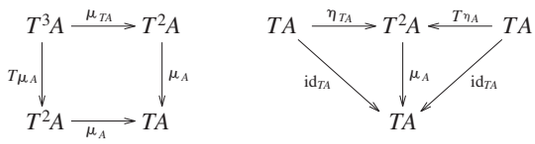
定義 4　カテゴリ $\mathcal{C}, \mathcal{D}$ ならびに $\mathcal{C}$ から $\mathcal{D}$ への関手 F, G に対し，F から G への関数 η で $\mathcal{C}$ の対象 A を $\mathcal{D}$ の射 $\eta_A : F(A) \to G(A)$ に写し，以下の図式が**可換** $(Gf \circ \eta_A = \eta_B \circ Ff)$ となるとき，η を**自然変換 (natural transformation)** とい

い $\eta : F \dot{\to} G$ と書く．



Haskell の Monad は次に紹介するカテゴリ論のモナドを由来にしており，両者は密接な関係を持つ．

定義 5　カテゴリ $\mathcal{C}$ 上に関手 $T : \mathcal{C} \to \mathcal{C}$ と 2 つの自然変換 $\eta : Id_{\mathcal{C}} \dot{\to} T, \mu : T^2 \dot{\to} T$ が存在し以下の図式が可換になるとき，(T, η, μ) の 3 つ組を $\mathcal{C}$ 上の**モナド (monad)** という．



直感的に言うと，図式が可換ということは μ が結合則，η が恒等元の役割を果たすことに相当し，モナドはモノイドとして捉えることができる．なおモナドでは恒等元に相当する η を 3 つ組の第 2 要素に書くことが多く，モノイドにおける位置（第 3 要素）と異なるため注意されたい．

定義 6　カテゴリ $\mathcal{C}$ 上の関手 $T : \mathcal{C} \to \mathcal{C}$, 自然変換 $\eta : Id_{\mathcal{C}} \dot{\to} T$, 射 $f : A \to TB$ に対し $f^* : TA \to TB$ が以下の性質を満たすとき，3 つ組 $(T, \eta, _^*)$ を **Kleisli triple** という．

- $\eta^*_A = \mathrm{id}_{TA}$
- $f^* \circ \eta_A = f$
- $g^* \circ f^* = (g^* \circ f)^*$

各 Kleisli triple$(T, \eta, _^*)$ は，$T(f : A \to B) = (\eta_B \circ f)^*, \mu_A = \mathrm{id}^*_{TA}$ とすることでモナド (T, η, μ) となることが分かる．また Kleisli triple を元に，対象を Obj$(\mathcal{C})$，射 $\mathcal{C}_T(A, B)$ を $\mathcal{C}(A, TB)$, A への恒等射を η_A, f と g の合成を $g^* \circ f$ とするカテゴリ $\mathcal{C}_T$ が存在しこれを **Kleisli カテゴリ**という．

$\mathcal{C}_T$ を用いると，非決定的な計算や副作用を持つ計算，継続計算などさまざまな計算モデルが表現できることが知られている [2]．この $\mathcal{C}_T$ が Haskell の Monad とどう対応しているか，次節で明らかにすることにしよう．

Haskell の Monad との関係

Haskell で Monad を扱うには，対象とするデータ型と関数 `return`, `(>>=)` を用意し，さらに `return`, `(>>=)` は Monad 則と呼ばれる規則を満たすよう定義する必要がある．このときデータ型に対応する関手 T ならびに `return`, `(>>=)` によって Kleisli triple が定まり☆3，Monad クラスのインスタンスは Kleisli カテゴリに対応することを示すのが，本節の狙いである．

まず復習も兼ねて Haskell における Monad クラスの定義ならびにインスタンスの例 (List Monad, Maybe Monad) を以下に挙げる．

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

☆3　正確には $_^*$ に対応するのは `(>>=)` ではなく `(=<<) = flip (>>=)`.

```
    (>>)   :: m a -> m b -> m b
    fail   :: String -> m a

    -- Minimal complete definition: (>>=), return
    p >> q  = p >>= \ _ -> q
    fail s  = error s

instance Monad [] where
    (x:xs) >>= f = f x ++ (xs >>= f)
    []     >>= f = []
    return x     = [x]
    fail s       = []

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

Haskell の処理系には有用な Monad があらかじめライブラリとして用意されているが，自分自身で Monad のインスタンスを作成してもよい．ただしすべての Monad は，以下に示す Monad 則 [3] を満たすよう (>>=), return を定義しなければならない．上に挙げた List Monad や Maybe Monad がこの法則を満たすことは，容易に確認できる．

Monad 則

```
1.   return x >>= f  ==  f x
2.    mx >>= return  ==  mx
3. (mx >>= f) >>= g  ==  mx >>= (\ x -> f x >>= g)
```

ここからは Haskell における計算の概念をカテゴリ $\mathcal{C}$ に照らし合わせて考えることにしよう．$\mathcal{C}$ の射として関数を考えるのが自然で，そうなると合成演算と恒等射はそれぞれ関数合成 (.)，恒等関数 id が対応する．また対象については，射が対象から対象への写像であることを考えると，数値や文字などの各値ではなく，型で代表される値の集合とみなすのが適切であろう．たとえば関数 length は，対象 [a] から対象 Int への射とみなせる．

次に Monad m に対し以下に定義した関数 fmap は，通常の計算の世界から Monad を用いた計算の世界への対応，すなわち関手になっていることを示す．

```
fmap :: (a -> b) -> (m a -> m b)
fmap = \ f -> \ mx -> mx >>= (\ x -> return (f x))
```

関手が満たすべき恒等射と合成に関する条件は以下のように示せる．

$F(\mathrm{id}_A)$

```
 => fmap id
 => \ mx -> mx >>= (\ x -> return (id x))        (fmap の定義)
 => \ mx -> mx >>= (\ x -> return x)             (id の定義)
 => \ mx -> mx >>= return                        (η 簡約)
 => \ mx -> mx                                   (Monad 則 2)
 => id_FA
```

$F(g) \circ F(f)$

```
 => \ mx -> fmap g (fmap f mx)
 => \ mx -> fmap f mx >>= (\ y -> return (g y))                          (fmap の定義)
 => \ mx -> (mx >>= (\ z -> return (f z))) >>= (\ y -> return (g y))     (fmap の定義)
 => \ mx -> mx >>= (\ w -> (\ z -> return (f z)) w >>=
```

```
                                  (\ y -> return (g y)))                    (Monad 則 3)
 => \ mx -> mx >>= (\ w -> return (f w) >>= (\ y -> return (g y)))          (β 簡約)
 => \ mx -> mx >>= (\ w -> return (g (f w)))                                (Monad 則 1)
 => fmap (g . f)
 => F(g∘f)
```

そこでカテゴリ $\mathcal{C}$ に対し，以下のように tee, eta, mu[☆4] を定めると (tee, eta, mu) はカテゴリのモナドになることが分かる．

```
関手 T (射について)
  tee :: (a -> a) -> (m a -> m a)
  tee f = fmap f
単位 η
  eta :: a -> m a
  eta = return
乗法 μ
  mu :: m (m a) -> m a
  mu = \ mmx -> (mmx >>= (\ mx -> mx))
     = \ mmx -> mmx >>= id
     = (>>=id)
```

これがモナドであることを示すには，モナドの定義における図式が可換であることを示せばよく，その証明は以下のようになる．

μ：左下のパス

```
 => (μA . TμA) mx
 => μA (TμA mx)
 => (TμA mx) >>= id                  (μ の定義)
 => (mx >>= (return . μA)) >>= id    (T の定義)
 => mx >>= (\ x ->
      (return . μA) x >>= id)        (Monad 則 3)
 => mx >>= (\ x -> id (μA x))        (Monad 則 1)
 => mx >>= (\ x -> μA x)
 => mx >>= (\ x -> x >>= id)         (μ の定義)
```

μ：上右のパス

```
 => (μA . μTA) mx
 => μA (μTA mx)
 => (μTA mx) >>= id                  (μ の定義)
 => (mx >>= id) >>= id               (μ の定義)
 => mx >>= (\ x -> id x >>= id)      (Monad 則 3)
 => mx >>= (\ x -> x >>= id)
                                     ∴ μ の図式は可換
```

η：右のパス

```
 => (μA . TηA) mx
 => μA (TηA mx)
 => (TηA mx) >>= id                  (μ の定義)
 => (mx >>= (return . ηA)) >>= id    (T の定義)
 => mx >>= (\ x ->
      (return . ηA) x >>= id)        (Monad 則 3)
 => mx >>= (\ x -> id (ηA x))        (Monad 則 1)
 => mx >>= (\ x -> ηA x)
 => mx >>= ηA                        (η 簡約)
 => mx >>= return                    (η の定義)
 => mx                               (Monad 則 2)
```

η：左のパス

```
 => (μA . ηTA) mx
 => μA (ηTA mx)
 => (ηTA mx) >>= id                  (μ の定義)
 => (return mx) >>= id               (η の定義)
 => id mx                            (Monad 則 1)
 => mx
                                     ∴ η の図式は可換
```

また以下に定義する関数 (=<<) に対し，(tee, eta, (=<<)) が Kleisli triple の条件を満たし，

```
(=<<) :: (a -> m b) -> m a -> m b
(=<<) = flip (>>=)
      = \ f -> \ mx -> mx >>= f
```

対応する Kleisli カテゴリが Monad m のインスタンスとなることが確認できる．

☆4 mu はライブラリの関数 Control.Monad.join と等しい．

このように，Haskell における Monad の概念はカテゴリにおけるモナドと密接に関係していることが分かる．プログラミングで Monad を用いる際こうした数学的背景について知っておく必要はないが，(>>=) や return といった演算が異なる Monad 間でオーバーロード可能で，Monad の合成をしてもプログラムの多くは書き換えなくてよいという性質はこうした背景によるものである．おさらいとして，カテゴリ論のモナドと Haskell の Monad の対応を**表 -1** にまとめておこう．

カテゴリ論	Haskell
カテゴリ $\mathcal{C}$	
対象	各型で代表される値の集まり
射	`f :: a -> b`
恒等	`id :: a -> a`
合成 $g \circ f$	`(.) :: (b -> c) -> (a -> b) -> (a -> c)`
	`(.) g f = \ x -> g (f x)`
モナド (T, η, μ)	
T	`fmap :: a -> b -> (m a -> m b)` で `a == b` の場合
η	`return :: a -> m a`
μ	`join :: m (m a) -> m a`
Kleisli triple $(T, \eta, _^*)$	
T	上に同じ
η	上に同じ
$_^*$	`(=<<) :: (a -> m b) -> m a -> m b`
Kleisli カテゴリ $\mathcal{C}_T$	
対象	$\mathcal{C}$ の対象と同じ
射	`f :: a -> m b`
恒等	`return :: a -> m a`
合成 $g^* \circ f$	`comp :: (b -> m c) -> (a -> m b) -> (a -> m c)`
	`comp g f = \ x -> ((=<<) g) (f x)`
	`= \ x -> f x >>= g`

表-1　カテゴリのモナドとHaskell Monadの対応

参考文献

1）Bratley, P. and Millo, J.: Computer Recreations: Self-Reproducing Automata, Software - Practice and Experience, 2, pp.397-400 (1972).

2）Moggi, E.: Computational Lambda-Calculus and Monads, Logic in Computer Science, pp.14-23 (1989).

3）Newbern, J.: All About Monads, - A Comprehensive Guide to the Theory and Practice of Monadic Programming in Haskell, http://www.nomaware.com/monads/html, (訳, モナドのすべて , http://www.sampou.org/haskell/a-a-monads/html/).

4）Template Haskell, http://www.haskell.org/th/

5）The Quine Page, http://www.nyx.net/~gthompso/quine.htm

6）Wikipedia:Quine, http://en.wikipedia.org/wiki/Quine

7）Wikisource:Quines, http://en.wikisource.org/wiki/Quines

（平成18年1月19日受付）