ファンクショナルプログラミングとカテゴリー論の考え方、その基礎を身につける

～ クラウド時代のソフトウェアアーキテクト、オブジェクト指向プログラマ、もしくは FP 初心者のための基礎講座 ～

第 1 回 ： カテゴリー論、ラムダ抽象、及び、タイプシステム間の相互関係

第1版　2014年3月1日

The world is everything that is the case.
Logic is not a theory but a reflection of the world.
Mathematics is a method of logic.
- Ludwig Wittgenstein (1889-1951), Tractatus Logico-Philosophics (1921), 1, 6.13, 6.234 -

【 はじめに 】

私の名前は戸﨑貴裕です。本書の目的は、主にソフトウェアアーキテクト、オブジェクト指向プログラマ、もしくはファンクショナルプログラミング（以下「FP」と表記します。）初心者といった方々を対象とし、クラウド時代のソフトウェアアーキテクチャやプログラミングに多大なる影響を与え続けている FP の概念、Lambda、Map、Closure、Lazy Evaluation、Monoid、Monad、Immutable Value、Referential Transparency、No Side Effect 等の概念を、個別の概念としてではなく、1 つの体系的な考え方として身につけていただくことにあります。重要なのはその考え方であり、FP そのものではありません。

FP は、ラムダ抽象で表現される静的なデータマップ（ファンクション）を組み合わせ、矛盾のない処理設計を可能とする考え方を基礎としており、その設計パターンを論理的に構築、展開する基礎にカテゴリー論（Category theory、圏論）があります。よって、FP だけを表面的に捉えても、カテゴリー論だけを表面的に捉えても、ソフトウェアアーキテクチャやプログラミングへの応用は十分に理解できません。とはいえ、FP もカテゴリー論もじっくり学ぶ時間がない、FP やカテゴリー論について日本語で調べてもすっきりと理解できる説明がない、機械翻訳や難解な用語で頭がパニック、といった場合も多いのではないでしょうか。本書には、そういった方の手助けになればという意図もあります（注 1）。

第 1 回では、カテゴリー論の基本的な考え方と FP 言語 Haskell におけるファンクションとの関係を見た後、Monoid と Functor の例を示し、カテゴリー論、ラムダ抽象、及び、タイプシステム間の相互関係をまとめます。そして第 2 回では、オブジェクト指向における応用を予定しています。

なお、本書では、具体的なコード例や製品例等が出てきますが、言語や製品を知らなくとも、少なくともその例が理解できる程度に解説し、細かな点は注記としてまとめます。それから、なるべく特定のベンダーに偏った説明は避けたいのですが、筆者が以前マイクロソフトでアプリケーション開発コンサルタントや開発エバンジェリストであったこともあり、Haskell 以外の部分については、多少マイクロソフト系技術の例が多くなることはご容赦ください。

0【 本書の背景 】

何事も、おおもとまで辿らんと、大事なことを見落とすものだ。

- 映画「聯合艦隊司令長官　山本五十六　太平洋戦争70年目の真実」より -

皆さんは、「クラウドとは何ですか？」と聞かれて、何と答えますか？一言でいえば、ネットワーク上の仮想マシン群ですよね。物理的なサーバーを数十台、数百台、数千台と用意し、仮想マシンを動作させ、ネットワークを通じ、様々な方法で顧客もしくは組織内の利用者にサービスを提供するのがクラウドサービスであり、提供方法により、IaaS、PaaS、SaaS 等と分類されたりするわけです。

それでは、何故、クラウドという形態がメジャーなサービスの形態となったのでしょうか？技術的に主要な原因の 1 つは、ムーアの法則の限界といわれます。コンピュータシステムの処理能力を上げる方法を大きく 2 つに分けた場合、スケールアップとスケールアウトのあることはご存知でしょう。2000 年前後までは、ムーアの法則がそれなりに現実のものとなっており、スケールアップの考え方が通用していました。しかし、熱処理やコストの問題などによって、スケールアップを追求し続けることが困難になりはじめ、多くのソリューションベンダが、スケールアウトの道を探ることになりました。これが、仮想マシン群を効率よく管理、運用するという技術の発展につながり、実用化されることで、現在のクラウドのような形態が生まれたわけです。

さて、前記の通りクラウドは、ネットワーク上の仮想マシン群です。ソフトウェアからすれば、分散して存在する CPU、メモリ、データストレージといったリソース、それから、PC、モバイルデバイス、センサーといった無数の端末がネットワークを介して生み出すビッグデータ、つまり、テラバイト、ペタバイト級のデータを相手にする可能性がある、ということになり、この状況でソフトウェアに求められるのが、分散（Distributed）、同時（Concurrent）、並列（Parallel）、非同期（Asynchronous）といった処理パターンを効率的かつ矛盾なく組み合わせることのできる設計となります。

とはいえ、ソフトウェアの処理は、やみくもに分割しても並列実行できるとは限らず、やみくもに分散配置しても同時に発生する要求やイベントに対応できるとは限りません。何故なら、現実世界の要求には、並列同時実行のできない要求、例えば共有リソースを更新する要求などが存在するからです。こうして、アムダールの法則から導き出される問題、つまるところ、インフラの並列処理性能を生かせるかどうかはソフトウェアの設計次第、という問題が一気に重要度を増し、深刻化したのです。

このような要請に対し、ラムダ抽象で表現される静的なデータマップ（ファンクション）を組み合わせ、矛盾のない処理設計を可能とする FP の考え方と、その設計パターンを論理的に構築、展開する基礎となるカテゴリー論の考え方が、FP 言語にとどまらず、オブジェクト指向言語、ダイナミック言語、データベース、そして様々なソフトウェアアーキテクチャに採用され、成果を上げています。

スケールアウトの考え方をいち早く導入して成功を収めた Google が採用し、実質的なビッグデータ処理方式のスタンダードとなった MapReduce の例をはじめ、多くのメジャーな開発言語でのラムダ抽象の採用、JVM、CLR 等のランタイム向けに FP の考え方を実現する言語の登場、Immutable Value、No Side Effect、Referential Transparency それから Monad といった考え方を取り入れたライブラリ、データベースや Web API の登場と、もはや、FP とカテゴリー論、両者の考え方を知らずにソフトウェア設計を十分に理解することはできないという状況が続き、今後も続く様相を呈しています。

それでは、このような背景を頭の隅に置いて、本題に入ることとしましょう。

1【 カテゴリー論の基本 】

第１回では、カテゴリー論の基本的な考え方を身につけることからはじめます。

ちなみに、カテゴリー論の基本的な考え方はシンプルですので、理解することは難しくありません。その一方で、およそパターンというものが重要である全ての分野において最強の思考道具となりうる考え方であり、その応用においては柔軟な発想が必要になります。

1.1　カテゴリーの定義

カテゴリー論は、カテゴリー（category）を扱う数学の理論です。数学の理論ですから、扱う対象には定義があります。なお、次項「1.2 カテゴリーの定義を理解する」を読むと、いつのまにかこの定義を理解していることになりますので、ここではざっと定義を一読し、次項を読み始めてください。

定義 ： カテゴリー（category）（注 2）

カテゴリーは、次のデータから成り、

1　１つ以上の対象（objects）
　表記例 ： A, B, C...

2　１つ以上のマップ（maps）
　表記例 ： f, g, h...

3　各マップにつき、ドメイン（domain）となる対象１つ、及び、
　コドメイン（codomain）となる対象１つ
　表記例 ： $f : A \to B$　もしくは　$A \xrightarrow{f} B$

4　各対象におけるアイデンティティマップ（identity map）
　表記例 ： 1_A もしくは id_A

5　１つ以上の合成マップ（composite map）
　表記例 ： $f \circ g$

次の規則を満たす。

6　アイデンティティ規則（identity law）

7　アソシエイティヴ規則（associative law）

1.2　カテゴリーの定義を理解する

はじめに、具体的なカテゴリーを考える場合、その対象（object）を決める必要があります。本項では、対象を集合（set）とするカテゴリーである、集合のカテゴリー（category of sets）を扱います。その理由は、集合のカテゴリーが、カテゴリー論における基本のカテゴリーであること、そしてなにより、ファンクションやタイプシステムを表現でき、よってソフトウェア設計全般にその理論が応用可能なカテゴリーであることです。集合のカテゴリーを考える場合、先の定義において、「対象」を「集合」と読み替えてかまいません。

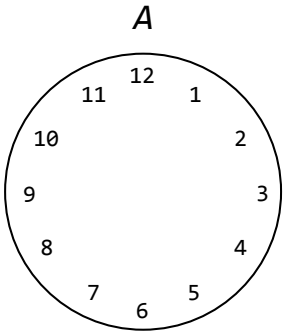
それでは、集合のカテゴリーを用いて、カテゴリーの定義を理解してしまいましょう。

はじめに、集合（set）とは、共通の性質を持ち、かつ重複しない要素の集まりです。手始めに、12時間表示の時計における時間の集合 *A* を定義してみます。

$$A = \{ 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12 \}$$

（以降、特定のプログラミング言語であることを示さない表記は、数学における一般的な表記になります。）

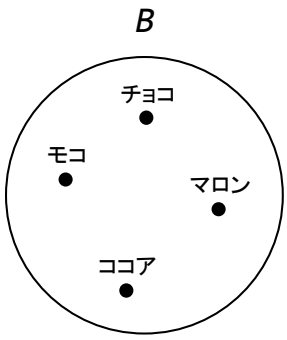
集合 *A* を絵にすると、次のようになります。要素はそのまま数字で表しています。



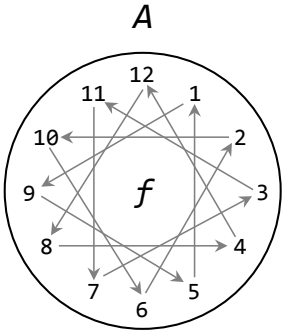
そしてもう１つ、４匹の子犬、チョコ、モコ、マロンとココアの集合として、集合 *B* を定義します。

$$B = \{ \text{チョコ}, \text{モコ}, \text{マロン}, \text{ココア} \}$$

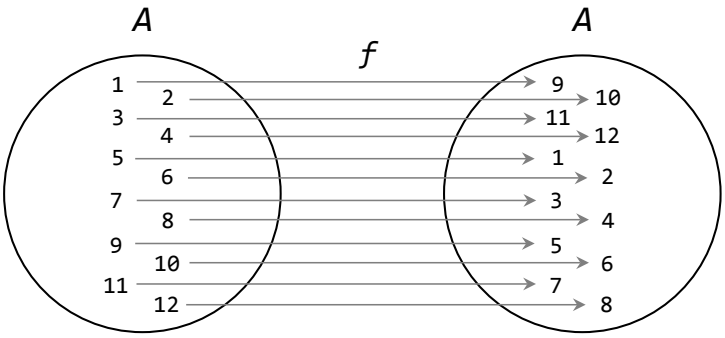
集合 *B* を絵にすると、次のようになります。要素は黒丸（●）で表し、要素名を添えてあります。



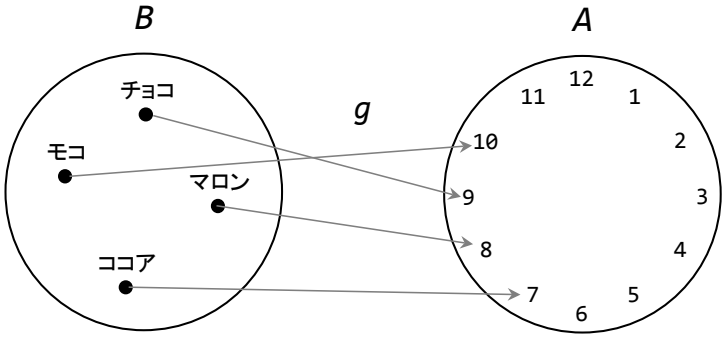
次に、集合のカテゴリーにおけるマップ（map）という考え方について、集合 *A* と集合 *B* を使った例を見てみます。はじめに、集合 *A* における「8 時間後」というマップを *f* として矢印で示すと、次の絵のようになります。例えば、9 時の 8 時間後は 5 時、といった具合です。



マップ f（「8 時間後」マップ）は、次の絵のように表現することもできます。



そしてもう一つ、集合 A と集合 B を使い、「散歩の時間」というマップ g を絵にしてみます。



このように、集合のカテゴリーにおけるマップとは、ある集合の全ての要素について、各要素から、同じ集合もしくは別の集合の 1 つの要素への静的な（変わることのない）、1 対 1、かつ 1 方向の全ての矢印の集合を意味し、矢印の先となる要素は全て 1 つの集合内になければなりません（注 3）。

さて次に、マップには 2 通りの捉え方がありますので、「8 時間後」マップである f を例として見てみましょう。

1 つ目の捉え方は、f を、集合 A から集合 A への矢印（⟶）とする捉え方（対象間の捉え方）であり、次のように表記します。

$$f : A \longrightarrow A$$

これを、「f は A から A へのマップである。」と読みます。マップは、 $A \xrightarrow{f} A$ のように表記されることもあります。

なお、$f : A \longrightarrow A$ における矢印（⟶）には別名が多く、マップ、ファンクション、トランスフォーメーション（transformation）、オペレータ（operator）、矢印（arrow）もしくは射（morphism）などと呼ばれることがありますが、全て同じです。本書ではマップと呼びます。

2 つ目の捉え方は、f を実装とする捉え方（要素間の捉え方）です。この場合、従来の数学の関数表記と同じく、評価結果を $f(a)$ と表記します。ここで、a は集合 A の要素を表す引数になります。例えば、$f(7) = 3$ と評価されます。なお、この捉え方の場合、$a \mapsto f(a)$ （入力 ↦ 出力）のように特殊な矢印（根元に縦線をもつ矢印。）が使われます。

捉え方にかかわらず、全てのマップについて、矢印の元となる集合をドメイン（domain）、矢印の先となる集合をコドメイン（codomain）といいます。そして、 $f : A \longrightarrow A$ のようにドメインとコドメインが同一の集合であるマップを、エンドウマップ（endomap、内部マップ）といいます。エンドウマップのうち、実装が $f : a \mapsto a$ となるマップ、別の表現をすれば、 $f(a) = a$ となるマップ、つまり、実装の矢印の元と先が必ず同じ要素になるマップを特に、アイデンティティマップ（identity map）といい、1_A もしくは id_A のように表記します。集合 A でいえば、「0 時間後」というマップがアイデンティティマップに相当します。話を f に戻すと、 f はエンドウマップであり、アイデンティティマップではない、ということになります。

さてここで、カテゴリー論の説明だけでは退屈だと思いますので、純粋な FP 言語として知られる Haskell を使い、先ほどのマップ f（「8 時間後」マップ）の適用例を見てみましょう。Haskell におけるファンクションと f の定義方法については後ほど理解することになりますので、ここではイメージだけをつかんでください。それでは、f をいくつかの数に適用する Haskell 表現の例です。

```
map f [1, 2, 2, 8, 10, 6, 4, 3]
```

この表現は、1、2、2、8、10、6、4、3 のそれぞれに対してマップ f（8 時間後マップ）を適用する表現であり、よって評価結果は次の通りとなります。全て 8 時間後ですね。

```
[9, 10, 10, 4, 6, 2, 12, 11]
```

この Haskell の例における map は、ビッグデータ処理の MapReduce にある Map と同じ考え方です。異なる点は、上記表現が 1 つの CPU で評価されるのに対し、MapReduce の場合であれば、各データに対する f の適用が分散並列処理の対象として扱われる点です。この場合の分散並列処理の最小単位については、map f [1, 2, 2, 8, 10, 6, 4, 3] を次のように書き換えるとわかりやすくなるかもしれません。これは正しい Haskell の表現であり、評価結果は同じく [9, 10, 10, 4, 6, 2, 12, 11] になります。

```
[f 1, f 2, f 2, f 8, f 10, f 6, f 4, f 3]
```

このように個々の計算（computation）が表現されれば、計算毎の分散並列実行も想像できますよね。もちろん、上記評価 1 回のために分散並列処理を行うことは通常考えられませんが、重要なのは規模ではなく、個々の計算が他の処理やリソースに依存していないという点です。現実のソフトウェアにおける並列処理設計は、この f のようにシンプルで依存関係のない、つまり、外部依存により値が変わったり、待ちや競合が発生したりしない計算を定義できるかどうかにかかっています。数十億の Web ページの分析も、単語のカウントといった単純計算の組み合わせで表現できて初めて可能となります。まさに分割統治（divide and conquer）の実践ですね。なお、MapReduce の Reduce は、分散並列処理後のデータをまとめる処理にあたり、FP 言語における fold の考え方に相当します。

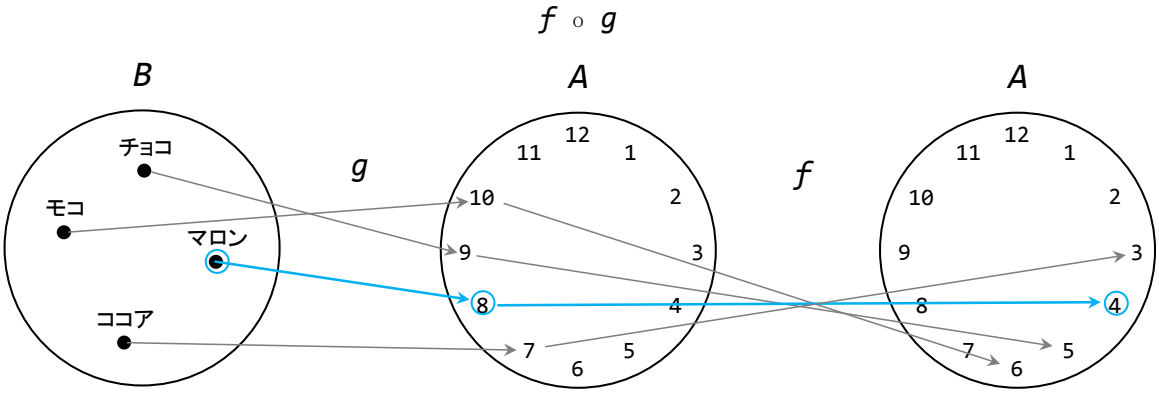
さて、マップ適用のイメージがつかめたでしょうか。それではカテゴリー論の話に戻ります。

マップ f とマップ g を定義した後で、「犬の食事は散歩の後がよく、8 時間おきの食事が理想です。」という話を耳にしました。ということは、散歩の直後に食事をしたとして、次の食事は、散歩の時間の 8 時間後ぐらいでもよいのではないでしょうか。

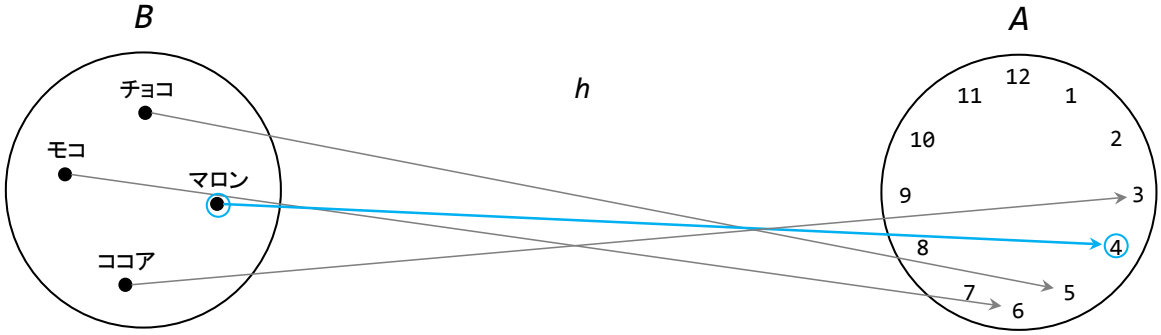
幸い手元に、「散歩の時間」というマップ g と、「8 時間後」というマップ f があります。この 2 つのマップを合成して、「次の食事の時間」というマップ h を作成してみましょう。

$$h = f \circ g$$

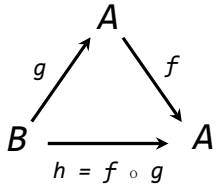
マップの合成は、白丸（∘）で表し、先に評価されるマップを右側に表記します。これは、数学の関数表記において、g を評価した後に f を評価する場合、$f(g(b))$ と表記するのと同じ順番です（b は集合 B の要素とします。）。それでは、 $f \circ g$ を絵にしてみましょう。



青い矢印は、数学の関数表記にすると、$f(g(b))$ の b にマロンを与えた矢印であり、$f(g(\text{マロン}))$ において、$g(\text{マロン})$ が先に 8 と評価され、$f(g(\text{マロン})) = f(8)$ となり、次に、$f(8)$ が 4 と評価されます。そして先ほど、$h = f \circ g$ と定義しましたので、 $h(\text{マロン}) = 4$ となります。これが、「次の食事の時間」マップ h です。



ここで、h のドメインとコドメインを確認します。$f : A \longrightarrow A$、$g : B \longrightarrow A$、ですから、$f \circ g$ である h は、そのままつなげると、 $B \xrightarrow{g} A \xrightarrow{f} A$ になります。ここでは、g が先に評価され、その結果が f のドメインになる、つまり g のコドメイン A が f のドメイン A になっています。結果として、$h : B \longrightarrow A$ と表記できます。これが、マップの合成の考え方です。そして、この合成マップを集合間（対象間）だけの概念図で示すと、次のようになります。g の矢印の後に f の矢印をたどる場合（$f \circ g$）と、h の矢印をたどる場合とで、ドメインとコドメインが同じであることが見て取れますね。

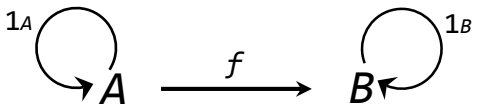


なお、上記の例から、先に評価されるマップのコドメインと後に評価されるマップのドメインが同一でないとマップの合成はできない、ということも同時に理解できると思います。このことは、ソフトウェア設計への応用において非常に重要になりますので、忘れないでください。

さて、皆さんは既に、カテゴリーの定義のうち、アイデンティティ規則とアソシエイティヴ規則を除く部分の具体例を理解していることになりますので、「対象」を「集合」と読み替えて、定義を読み返してみてください。

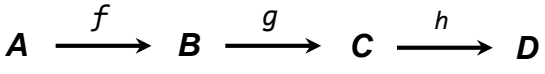
それでは、残りの 2 つの規則について説明します。この 2 つの規則は、ともにマップの合成に関する規則になります。

はじめに、アイデンティティ規則です。集合 A と集合 B のあるカテゴリーを考え、集合 A の要素を a、集合 B の要素を b で表すと、アイデンティティマップは、$1_A : A \longrightarrow A$、と $1_B : B \longrightarrow B$ の 2 つになり、それぞれの実装が、$1_A : a \mapsto a$ 、と $1_B : b \mapsto b$ になります。このカテゴリーにおいて、$f : A \longrightarrow B$ のとき、$f \circ 1_A$ という合成マップを考えると、1_A マップによって要素は変わらず、よって、$f \circ 1_A = f$ が成立しないとなりません。同様に、$1_B \circ f = f$ も成立しないとなりません。この規則をアイデンティティ規則といいます。絵にするとわかりやすいですね。



$$f \circ 1_A = f = 1_B \circ f$$

次に、アソシエイティヴ規則です。この規則は 3 つ以上のマップの合成に関する規則です。例えば、集合 A から D を扱うカテゴリーにおいて、$f : A \longrightarrow B$、$g : B \longrightarrow C$、及び、$h : C \longrightarrow D$ の 3 つのマップがあるとします。このとき、ドメインとコドメインの関係から、$h \circ g \circ f$ というマップの合成がありうるわけですが、数学における足し算において、$(1 + 2) + 3$ と $1 + (2 + 3)$ が同じ評価となるように（この性質もアソシエイティヴ法則、associative property と呼ばれます。）、$(h \circ g) \circ f = h \circ (g \circ f)$ が成立しないとならない、という規則が、アソシエイティヴ規則です。



$$(h \circ g) \circ f = h \circ (g \circ f)$$

これらの規則は、あるカテゴリーを定義して論理を展開していくうえで必要な規則になりますが、応用においてはその都度指摘しますので、今のところはこのような規則のあったことだけ頭の隅に置いておけば結構です。

さて、カテゴリー論の基本的な考え方はここまでです。あとは、矢印が逆向きのマップ（inverse map や dual におけるマップ。）を考えたり、カテゴリーを 2 つ想定してカテゴリー間のマップ（functor）を考えたりといった応用に対し、頭を柔軟にして対処すればよいだけの話です（注 4）。

それでは、ここで一旦カテゴリー論から離れ、とはいえ頭の隅に置きつつ、次項では FP におけるファンクションの考え方を身につけます。

2【 FP におけるファンクション 】

本項では、FP 言語の代表として Haskell を使用し、FP におけるファンクションの考え方を理解します。もちろん、Haskell を知らない方にも理解できるよう解説します。

2.1　ラムダ抽象

はじめに、プログラミング言語なしで、足し算を表すファンクション add を考えてみます。おそらく、次のように表記するのが一般的ではないでしょうか。

```
add(a,b) = a + b
```

念のために書きますが、*add* がファンクションの名前、 *(a,b)* はファンクション *add* の取る引数（入力変数）、右辺の *a + b* は評価方法であり、例えば、*add(3,4) = 3 + 4 = 7* と評価されます。

これを Haskell で表現した 1 例は、次のようになります。

```
1 add :: Integer -> Integer -> Integer
2 add a b = a + b
```

1 行目はファンクションの型（type、タイプ）の指定となります（注 5）。Integer とは整数のことですね。Haskell においても Integer は整数を表す型です。2 行目はファンクションの定義です（注 6）。

さて、このファンクションの型指定、 Integer -> Integer -> Integer では、矢印が使われています。この矢印は、カテゴリー論で見たマップを表しているのでしょうか。それとも、他の意味があるのでしょうか。この答えを探るには、少々昔話が必要です。

数学で使用される *f(x)* という形のファンクションの表記法は、1734 年、レオンハルト・オイラー（Leonhard Euler, 1707-1783）という数学者によってはじめて使用されたとのことです。一方で、アロンゾ・チャーチ（Alonzo Church, 1903-1995）というアメリカの論理学者は、1930 年代に、ギリシャ文字のラムダ（λ、lambda）を使用した次のような表記方法を考えました。この表記法をラムダ抽象（lambda abstraction）といいます。

```
λab.a+b
```

一見すると、これが足し算のファンクションを表すとは思えないかもしれません。ラムダ抽象では、λ がファンクションを表し、その右隣に引数を定義し、ドット（.）以降に出力（評価方法）を定義します。上記例では、引数が *ab*、出力が *a+b* ということになります。非常にシンプルですね。

ラムダ抽象では、先に見た add ファンクションにおける add のような名前は必要なく、よって、匿名のファンクション（anonymous function）として捉えることも、表記全体がファンクションの一意な名前であると捉えることもできます。

ちなみに、引数を指定して評価する場合には次のような表記になります。


```
λab.a+b 3 4 = 7
```

そして、この λab.a+b は、次の表記の省略形となります。

```
λa.λb.a+b
```

この表記が何を定義しているのか、わかりますか？。先ほどお話ししたラムダ抽象の規則、ラムダ（λ）の右側が引数、ドット（.）以降が出力という規則を当てはめてみましょう。そうすると、上記ファンクションは、引数が a、出力がファンクション λb.a+b となるファンクションであることになります。例えば、引数が 3（a = 3）のときの出力は、ファンクション λb.3+b になります。

```
λa.λb.a+b 3 = λb.3+b
```

そして、出力となったファンクションに 4 を入力すると、出力は 7 になります。

```
λb.3+b 4 = 7
```

そうすると、もともとの表記である λab.a+b は 2 の引数を取るファンクションに見えますが、ラムダ抽象においては、第 1 引数を入力として残りの評価に必要なファンクションを出力するファンクションを表している、ということになります。

これを、複数の引数をもつファンクションを、第 1 引数を入力として残りの評価に必要なファンクションを出力するファンクションに変換していると捉えた場合、この変換をカリー化（Currying）といいます。この名前は、PF 言語 Haskell の名の由来でもある、数学者であり論理学者であったハスケル・カリー（Haskell Curry, 1900-1982）からきています。

2.2 Haskell とラムダ

さて、昔話だけではわかりにくいと思いますので、ここで現代の Haskell にもどります。

Haskell におけるファンクションは、ラムダ抽象になります。よって、Haskell のファンクションは全てカリー化されているということになり、また、先ほどのラムダ抽象の例は、Haskell で表現できることになります。

ここで、ファンクション add を思い出してください。先ほどは add ファンクションを、add a b = a + b と定義していました。この定義をラムダ表記で書き換えると、次のようになります。なお、Haskell において、λ はその形に似せてバックスラッシュ（'\'）で代用します（注 7）。

```
add :: Integer -> Integer -> Integer
add = \a -> \b -> a + b
```

この表現方法では、\a -> \b -> a + b の部分が、先ほどの λa.λb.a+b に相当します。ドット（.）が矢印（->）に変わっただけですね。

これをファンクションの型指定にあわせると、 Integer -> Integer -> Integer は、「引数 a の型 -> 引数 b の型 -> 出力の型」を表しているということになります。そうすると、Haskell におけるファンクションは、型（タイプ）を持つラムダ抽象ということになります。

ここで、Haskell のインタラクティブ・コンパイラである GHCi を使い、add ファンクションがラムダ抽象と同じく扱えるのかどうかの実験をしてみましょう（注 8）。なお、ghci>という部分は、インタラクティブ・コンパイラのプロンプトです。

```
1 ghci> let add = \a -> \b -> a + b
2 ghci> add 3 4
3 7
```

1 行目がファンクションの定義（注 9）、2 行目がファンクションに引数を指定した表現、3 行目が評価結果です。3 + 4 = 7 ですから、足し算ファンクションとして機能していることが確認できました。

次に、add がカリー化されたファンクションであることを確認してみましょう。カリー化されたファンクションが、引数 1 つを入力として残りの評価に必要なファンクションを出力とする、ということでしたので、最初の確認では、add = \a -> \b -> a + b を add a = \b -> a + b と書き換えてみます。そうすると、引数が a、出力がファンクション \b -> a + b であることがわかりやすくなりますね。

それでは、このファンクションの引数 a に 3 を入力して（注 10）、その評価結果が何になるのか、実験してみましょう（解説はコードの後で。）。

```
1 ghci> let add a = \b -> a + b
2 ghci> let output = add 3
3 ghci> :t output
4 output :: Integer -> Integer
5 ghci> output 4
6 7
```

1 行目では、ファンクション add の定義を行っています。

2 行目では、ファンクション add の 引数 a に 3 を入力し、その出力（評価結果）を output という名前で定義しています（注 11）。そうすると、output は \b -> a + b の変数 a が 3 に特定された、\b -> 3 + b になるはずです。

3 行目では、型の表示を行うコマンド（:t）を使用して、output の型を問い合わせています。

4 行目の答えによると、output の型は、Integer -> Integer になっています。先ほどの解釈によれば、「引数が Integer 型 -> 出力が Integer 型」のファンクションということになります。

5 行目では、output に 4 を入力しています。

6 行目の結果は 7 になっています。このことから、output は、\b -> 3 + b と書けるファンクション、ラムダ表記で表すと *λb.3+b* と書けるファンクションとして機能しています。

そしてもちろん、Haskell のファンクションがラムダ抽象であれば、add = \a -> \b -> a + b と定義しても、ラムダ表記 \a -> \b -> a + b をそのまま使用した場合でも、add a b = a + b と定義した場合でも、同じ結果となるはずです。

この 3 つの場合を GHCi で試してみましょう。

```
ghci> let add = \a -> \b -> a + b
ghci> let output = add 3
ghci> :t output
output :: Integer -> Integer
ghci> output 4
7
```

```
ghci> let output = (\a -> \b -> a + b) 3
ghci> :t output
output :: Integer -> Integer
ghci> output 4
7
```

```
ghci> let add a b = a + b
ghci> let output = add 3
ghci> :t output
output :: Integer -> Integer
ghci> output 4
7
```

これで、Haskell におけるファンクションがラムダ抽象であるということ、よって、Haskell におけるファンクションはカリー化されている、つまり、1 つの引数を持つファンクションとみなされるということが実感できたと思います（注 12）。

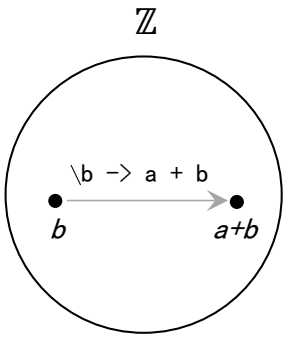
3【 ラムダ抽象とカテゴリー論との関係 】

本項では、ラムダ抽象とカテゴリー論との関係を見ることとします。

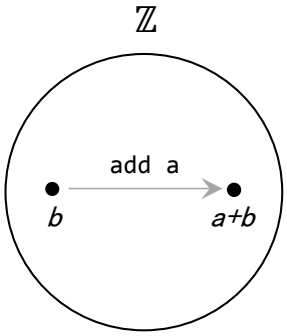
さて、add :: Integer -> Integer -> Integer という型指定が、入力の型と出力の型を表していることはわかりましたが、この矢印の組み合わせはカテゴリー論の矢印と関係があるのでしょうか。いまのところの手がかりは、add ファンクションの型が Integer -> (Integer -> Integer)と解釈されるということだけです。やはりここは、絵にして考えてみましょう。

はじめに、(Integer -> Integer) に注目します。add = \a -> \b -> a + b の型が Integer -> (Integer -> Integer) ですから、そのまま解釈すると、(Integer -> Integer) に該当する部分は、\b -> a + b になります。次に、(Integer -> Integer) という型から、Integer、つまり整数の集合を対象とする

カテゴリーと関係がありそうです。そこで、\b -> a + b という表現を、整数の集合 ℤ との関係で絵にしてみましょう。次の絵では、例えば b が 4 であれば、矢印の先は a + 4 になります。



ここで、add = \a -> \b -> a + b を変形すると add a = \b -> a + b、つまり\b -> a + b = add a ですから、先ほどの絵は次のように書くことができます（ファンクションの定義においても、数式と同じように等号の論理を扱えるのが Haskell のよいところです。）。



この絵をよく見ると、第 1 引数のみを持つ add a という 1 つの矢印、言い方を換えれば、「a を足す」という、a の値が定まれば特定できるマップだけで add ファンクションが表されています。

このように、2 つの変数を必要とするファンクションについて、1 つの変数の値を定めるだけでマップの定まるカテゴリーを、カーテシアンクローズトカテゴリー（Cartesian closed category、デカルト閉圏）といいます。add ファンクションの例では同じ集合（整数の集合）に閉じたマップ、つまり a も b も a+b も整数の集合の要素ですが、これが異なる集合の要素についてであっても、集合のカテゴリーは全て、カーテシアンクローズトカテゴリーになります（注 13）。

ということは、ここで Integer 型を集合のカテゴリーにおける対象と捉えたように、プログラミング言語における型（タイプ）の 1 つ 1 つを、その型の取りうる全ての値の集合と捉えれば、プログラミング言語のタイプシステムはカーテシアンクローズトカテゴリーとして扱えることになります。

そしてこのカーテシアンクローズトカテゴリーの特性が、カテゴリー論、ラムダ抽象（ファンクション）、及び、タイプシステム間の相互関係を体系的かつシンプルに結び付ける鍵となります。

ちなみに、既存のタイプシステムがない場合でも、例えば同じ構造を持つ JSON や XML の集合を考えて設計に生かすといったように、独自のデータ構造や DSL を考える場合においても、集合のカテゴリーさえ正しく構成できれば、本書第 1 回の内容は応用可能です。

そうすると、応用対象は、どのような表現方法におけるどのような集合でも、例えば、文字の集合でも、ショッピングカートの集合でも、ゲームのキャラクターの集合でも、マシンの集合でも、I/O の集合でも、ワークフローの集合でもよいのです。一気に応用範囲が広がりましたね。

さて、話が大きくなる前に、ここで、カーテシアンクローズドカテゴリーが引数を 2 つ取るファンクションについての話であれば、3 つ以上の引数を持つ場合は応用できないのか、という疑問をお持ちの方、いらっしゃると思います。

この場合、2 つの考え方があります。1 つめは、3 つ以上の引数を 1 つの組として考えることが妥当な場合です。例えば、3 次元座標を動かすには、x 座標、y 座標及び z 座標の移動が必要になりますが、この場合、例えば、座標を `(x,y,z)` として表し、座標の集合を対象とするカテゴリーを考えます。後にカテゴリー論とタイプシステムとの関係を見れば、座標だけでなく、ユーザー定義型を含む全てのデータ型について、集合のカテゴリーで成立する理論を等しく応用できることがわかると思います。

2 つめは、引数のそれぞれを独立させて扱うことが妥当な場合に、マップの合成を用いる考え方です。この場合には、思考順序を逆にしましょう。はじめに実現したいことをそのままマップとして考えるのです。実現したいことが「(a に) b を掛けてから c を足す」であれば、自ずと「b を掛ける」というマップと「c を足す」というマップの合成が思いつくでしょう。マップを個別の処理とするか、合成結果として得られるマップを 1 つの処理とするか、実装はどうあれ、現実的にはこの思考順序、つまりマップ（矢印）の特定、合成と分解が設計のプロセスとなることもあります。

さて、ここで、集合のカテゴリーにおいて応用可能な考え方を追加します。整数の集合を対象とした時の、アイデンティティマップは何でしょう。皆さんはおそらく、わざわざ答えるまでもない、アイデンティティマップの説明が、「実装が $f : a \mapsto a$ となるマップ」なのだから、Haskell で表現すれば、`Integer -> Integer` 型の `\a -> a` になるのでしょう（注 14）、と考えるはずです。正解です。

それでは、`add 0` という答えはどうでしょう。`add 0` は、`add a` の表す具体的マップの 1 つですが、`add a` の表す具体的マップの中で唯一、`add 0` は、整数の集合 $\mathbb{Z}$ における全ての要素 z を、$z + 0$ に対応させるマップ（$z \mapsto z + 0$）、つまり、整数の集合においては、結果として `\a -> a` と同じマップになります。そしてこの、`add 0` という答えは、ただの頭の体操的な答えでも、たまたまひらめいた答えでもありません。整数の集合を、足し算との関係において、モノイド（Monoid）として捉えた場合の考えです。というわけで、モノイドの説明も含め、再びカテゴリー論の話に戻ります。

4【 モノイドとファンクター 】

本項では、再びカテゴリー論に戻り、モノイド（Monoid）とファンクター（Functor）を理解します。本項のあとでまた FP に戻り、タイプシステムとの関係を見ることになります。

4.1 モノイド（Monoid）

モノイドとは、ソフトウェア設計やプログラミングにおいて応用範囲の広い概念であり、カテゴリー論とグループ論（群論、Group theory）において別々に定義されます。両者は密接に関係しますが別物ですので、注意してください。ここでは、話の展開上、グループ論における定義と具体例から見ることにします。

はじめに、グループ論の扱う対象は、集合と、集合の要素 2 つを引数とする二項演算子のペアであり、$(G, \bullet)$ と表記されます。ここで、G は集合を、黒丸（•）は二項演算子を表します。

次に、モノイドは、グループ論において次のように定義されます。なお、すぐに具体例を示して説明しますので、定義を一読する程度で具体例に進んでください。

定義：グループ論におけるモノイド（monoid）

モノイド $(G, \bullet)$ は、次の 3 つの要請を満たす（注 15）。

1 クロージャー（closure、閉包）
G の要素 a, b の全ての組ついて、二項演算子 $\bullet$ を適用した $a \bullet b$ の結果も、
G の要素となる。

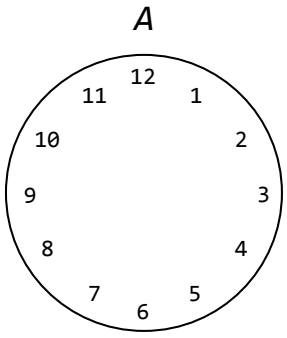
2 アソシエティビティ（associativity、結合律）
G の要素 a, b, c の全ての組について、$a \bullet (b \bullet c) = (a \bullet b) \bullet c$ が成立する。

3 アイデンティティ要素（identity element、単位元）
G には、G の全ての要素 a について、$e \bullet a = a \bullet e = a$ の成立する要素 e が存在する。

さて、具体例を考えてみます。ここでは、カテゴリー論の説明で使用した 12 時間表示の時計における時間の集合 A を、モノイド（グループ論）として再定義してみましょう。集合 A の定義は次の通りでした。

$$A = \{ 1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12 \}$$

データのイメージは理解を助けますので、集合 A の絵も思い出してみましょう。



集合 A をモノイドとするには、集合 A の要素 2 つを引数とする二項演算子が必要であり、かつ、モノイドの定義にある 3 つの要請に応える必要があります。ここでは、時間の足し算を表す二項演算子を考えてみます（注 16）。この二項演算子（集合 A における時間の足し算）を、Haskell のファンクションとして定義した 1 例は次の通りです。ここでは、ファンクションの名前を addTime とします（解説はコードの後で。）。

```
addTime :: Integer -> Integer -> Integer
addTime a b  ……  ① 引数 a 及び b をとるファンクション addTime は、
  | r == 0 = 12  ……  ③ 余りが 0 であれば（12 で割り切れれば）12 と評価され、
  | otherwise = r  ……  ④ 余りが 0 以外であれば余りの値として評価される。
  where
    r = mod (a + b) 12  ……  ② a と b とを足した値を 12 で割った余りを計算し、
```


Haskell の文法を知らなくとも、青字の解説を①から④の順に読むと、集合 `A` の要素全ての組み合わせ `a, b` について、`addTime` の評価結果が 1 から 12 の範囲となること、つまり、クロージャーが満たされていることがわかると思います。12 時間表記の時計における時間の足し算ですから、結果が 1 から 12 の範囲になければ困りますね。また、`addTime` は、他のプログラミング言語において簡単に実装できることもわかると思います。

`addTime` は、数学的にいえば、12 を法（モジュロ、modulo）とする世界の足し算において、`0` を 12 にしているだけですね。仮に、`A = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11}` であったとしたら、`addTime` の定義は、 `addTime a b = mod (a + b) 12` 、つまり、足し算の結果を 12 で割った余りとして評価されるという、より簡単な定義になります。ここでわざわざ `0` を 12 に置き換えたのは、整数の足し算においても、モノイドによっては、アイデンティティ要素が `0` にならない場合のあることを示すためです。

どういうことかといいますと、集合 `A` ではなく、整数全体の集合 ℤ と足し算（`+`）の組をモノイド `(ℤ, +)` とした場合、アイデンティティ要素は `0` になります。ℤ の要素を `z` とすれば、全ての `z` について、`z + 0 = z` となるからです。ちなみに、ℤ と掛け算の組をモノイド `(ℤ, ×)` とした場合のアイデンティティ要素は 1 になります。全ての `z` について、`z ×1 = z` になりますね。アイデンティティ要素の決定は、後に見るタイプシステムへの応用においても、非常に重要になります。それに当然、言語の型は `Integer` だけではありません。

さて、話を集合 `A` と `addTime` に戻すと、モノイド `(A, addTime)` におけるアイデンティティ要素は、12 になります。`A` の全ての要素 `a` に対して、`addTime 12 a = a` かつ `addTime a 12 = a` が成立しますね。そして、`addTime` は足し算ですから、アソシエティビティの要請にも応えています。なお、クロージャー、アソシエティビティ及びアイデンティティ要素の確認を行う Haskell ファンクションの 1 例を注 17 に記載しますので、ご興味があれば確認ください。

ここまでで、モノイド `(A, addTime)` が定義できました。グループ論で定義されるモノイドが、集合、それから集合の要素 2 つを引数とする二項演算子の組み合わせであり、クロージャー、アソシエティビティ及びアイデンティティ要素の要請を満たす構造であることがお分かりになったと思います。

グループ論におけるモノイドは、時計の例のようにそのまま実用可能な考え方です。それでは、カテゴリー論におけるモノイドも同じようなものなのでしょうか。実は、カテゴリー論におけるモノイドは、そのままでは役に立ちません。しかし、ある方法によって見事に変身することができます。

4.2 カテゴリー論におけるモノイド

本当のことなのではじめにお断りしておきますが、本項と次項の説明を理解する価値は、タイプシステムとの関係が見えてくるまでさっぱりわかりません。すっきりするまで少々辛抱してください。

さて、カテゴリー論におけるモノイドは特殊なカテゴリーであり、実用としては、集合のカテゴリーにおけるパターンの原型を表すことになります。よって、その原型を別のカテゴリーに変換することではじめてその価値が現れるカテゴリーとなります。この考えを理解するには少々発想の転換が必要です。これからの説明を読む前に、カテゴリーの定義において、対象に制限がないこと、マップにはドメインとコドメインとなる対象がありさえすればよいこと、また、マップの合成はドメインとコドメインの関係上合成可能であれば、その合成演算の定義に制限のないことを念頭に置いてください。

はじめに、定義を示します。

> 定義：カテゴリー論におけるモノイド
>
> モノイドは、単一の対象を持つカテゴリーである。

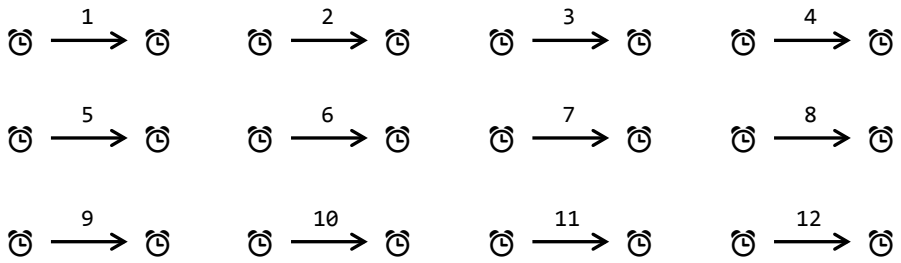
この定義から具体例をイメージするのは困難と思いますので、これから、具体例を *M* として話を進めます。ここで *M* は、グループ論におけるモノイド（*A*, addTime）を表すカテゴリーとします。

はじめに、カテゴリー論におけるモノイドは、1 つのカテゴリーです。よって、*M* には、対象、マップ、アイデンティティマップ、及びマップの合成が存在します。

次に、*M* の対象を、構造を持たない単一の物とします。例えば、この対象を ⏲ と表記します。この ⏲ は構造をもちませんので、これまでに見た集合やグループではありません。ただの物です。なお、目覚まし時計のように見えるこの表記は、書き間違いでも目の錯覚でもありません。目覚まし時計です。何でもよいのですが、対象を表す記号が必要なだけです。

次に、カテゴリー*M* には対象が ⏲ しかありませんから、全てのマップのドメインは ⏲ 、全てのマップのコドメインも ⏲ です。⏲ のアイデンティティマップは当然、⏲ から ⏲ へのマップです。

次に、*M* におけるマップを、1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 の 12 個定義します。つまり、集合 *A* の各要素をマップとみなすのです。数字で表されてはいますが、全てマップです。先にお話しした通り、この 12 個のマップ全てについて、ドメインは ⏲ 、コドメインは ⏲ になります。

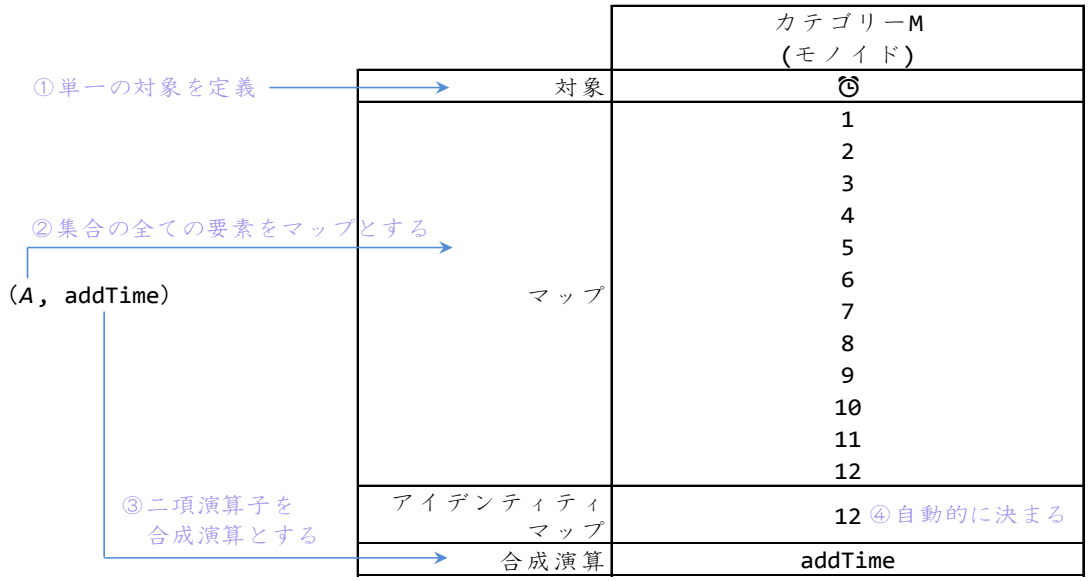


次に、マップの合成を考えます。ここで、マップの合成に用いる白丸（∘）の演算を、合成対象の 2 つのマップに対する addTime の適用とします。つまり、マップの合成を、マップを時間と見立てた足し算として評価するということです。そうすると、例えばマップ 7 とマップ 8 を順に合成すると、マップ 3 と評価されます（8∘7 = 3）。なお、この場合は足し算ですから、合成順序を逆にしても同じ結果となります（7∘8 = 3）。

M のもつ 12 個のマップは、ドメインとコドメインが同じ ⏲ ですので、その全てが互いに 2 通りの順序で合成可能、つまり、2 つのマップの合成だけでも順列計算（12! / (12 - 2)!）で 132 通りの合成が可能なわけですが、合成が addTime の適用ですから、結果として、*M* におけるマップの合成結果は全て、もともと存在する 12 個のマップのいずれかのマップになる、ということになります。

次に、対象 ⏲ のアイデンティティマップ（1⏲）を考えます。合成演算が addTime の適用ですから、答えは自動的に決まります。1⏲ は、マップ 12 になります。つまり、(*A*, addTime) のアイデンティティ要素 12 がそのままアイデンティティマップとなります。

ここまでの、グループ論におけるモノイド (*A*, addTime) から、カテゴリー論におけるモノイド M を定義した過程を絵にすると、次のようになります。



次項では、この M を実用的なカテゴリーに変身させましょう。

4.3 ファンクター（Functor）

さて、*M* の変身です。実際には、M を別のカテゴリーにマップします。ここでは、別のカテゴリーを *T* としましょう。*M* から *T* へのマップを *F* とすると、*F* は次のように表記されます。

$$F : M \longrightarrow T$$

このようなカテゴリー間のマップを、ファンクター（Functor）と言います。ファンクターは、それぞれのカテゴリーの構造を維持した、カテゴリーの構成要素間のマップを行います。対象は対象に、マップはマップに、アイデンティティマップはアイデンティティマップに、マップの合成はマップの合成に、という具合です。そして、マップの方法（実装）はファンクターの定義次第となります。

それでは、*T* が集合のカテゴリーになるように、ファンクター*F* を定義していきましょう。

はじめに、*M* の対象 ⏲ を、*T* の対象である集合 *A* にマップします。これを、次のように定義します。

$$F(⏲) = A$$

ここで *A* は、これまでどおり、集合 *A* = {1,2,3,4,5,6,7,8,9,10,11,12} とします。

次に、*M* の各マップを、*T* の各マップにマップします。これを、次のように定義します。

$$F(x) = \text{addtime } x$$

そうすると、ファンクター*F* は、*M* のマップ 1 を、*T* におけるマップ addTime 1 に、2 を addTime 2 に、3 を addTime 3 に、というようにマップすることになります。

この、addTime 1 という表現、どこかで見たことがありませんか？そうです、ラムダ抽象とカリー化のところで見た add a と同じ形ですね。add と同様に、addTime は引数を 2 つもつファンクションですが、カリー化されているとみなせば、第 1 引数を指定して評価すると、2 つ目の引数を入力として残りの評価を行うファンクションになるわけです。そして、集合のカテゴリーはカーテシアンクローズドカテゴリーですから、ファンクター*F* のマップ先である集合のカテゴリーにおいて、addTime a は、addTime ファンクションそのものを表現するマップである、ということになります。

さて、話をファンクター*F* に戻して、*T* におけるマップの合成を考えます。ファンクター*F* の適用によって、*T* には、addTime 1 から addTime 12 までの 12 個のマップがあるわけですが、これらを合成するとどうなるでしょう。例えば、(addTime 2) ∘ (addTime 1) は何になるでしょう。答えは、addTime 3 です。1 時間足した後に 2 時間足すのですから、簡単ですね。よって、*T* におけるマップの合成では次の法則が成り立ち、*M* の合成マップも全て *T* の合成マップに変換されることになります。

(addTime y) ∘ (addTime x) = addtime *x+y*

そして、addTime の実装を考えると、アイデンティティマップは当然、 addTime 12 になります。そうすると、先ほどの、*F*(x) = addTime x という定義はすでに、*M* のアイデンティティマップ 12 を *T* のアイデンティティマップ addTime 12 に変換していることになります。

ここで、ファンクター*F* で定義したマップを表にすると、次の通りになります。

	F : *M* → *T*	
	カテゴリー*M* (モノイド)	カテゴリー*T* (集合のカテゴリー)
対象	⏰ →	*A* = {1,2,3,4,5,6,7,8,9,10,11,12}
マップ	1 2 3 4 5 6 7 8 9 10 11 12 →	addTime 1 addTime 2 addTime 3 addTime 4 addTime 5 addTime 6 addTime 7 addTime 8 addTime 9 addTime 10 addTime 11 addTime 12
アイデンティティマップ	12 →	addTime 12
マップの合成演算	addTime →	ファンクションの合成

この表を見ると、カテゴリー*T* のマップは全て、addTime a という形を持つファンクションになっています。そして全てのマップがエンドウマップであり、かつ、addTime ファンクションが集合 *A* に対して取り得る全てのマップを 12 個のマップで表現している、ということになります。そうすると、*T* は、グループ論では集合 *A* と二項演算子 addTime の組み合わせであったモノイドを、カテゴリー論において、集合 *A* を対象とし、その対象に対するエンドウマップの集まりを持つカテゴリーとして解釈できる、ということを示しています。先に見たカーテシアンクローズドカテゴリーの応用ですね。

そして、これまでに見てきた、グループ論におけるモノイドからカテゴリー論におけるモノイドへ、そして集合のカテゴリーへ、という変換は、モノイド（$\mathbb{Z}$, +）、モノイド（$\mathbb{Z}$, ×）についても成り立つことがわかると思います。しかし、応用範囲はそれだけでしょうか？

ここで、*T* を一般化し、1 つの集合とエンドウマップを持つカテゴリーを $\mathcal{S}^{\circlearrowleft}$ と表記してみます。ここで *S* は集合を、*S* から出て *S* に戻る矢印をエンドウマップとします。実は、この $\mathcal{S}^{\circlearrowleft}$ と同じ構造を持つカテゴリーは、全て、モノイドとして扱うことができます。なぜなら、テゴリーは必ず対象についてのアイデンティティマップを持ち、マップの合成はアソシエイティヴ規則を満たすからです。そしてもちろん、集合のカテゴリーですから、カーテシアンクローズドカテゴリーでもあります。

この表記を応用可能な表記にするため、 $\mathcal{S}^{\circlearrowleft}$ を書き換えて、矢印で表されるエンドウマップが集合 *A* に対するエンドウマップであることを示すために、 $\mathcal{S}^{A}$ という表記を考えてみます。そうすると、対象となる集合を区別できる表記になりますね（この表記は参考文献 1 で使用されています。）。

それでは、再び FP の世界に戻り、 $\mathcal{S}^{\circlearrowleft}$ とタイプシステムとの関係を見ることとしましょう。

5【 タイプシステムへの応用 】

本項では再び FP の世界に戻り、はじめに、Haskell におけるデータ型の考え方がどのようなものか、具体例としてリスト型を見た後で、リスト型を使用しながら、モノイドとファンクターの応用を見ます。そして最後に、カテゴリー論とタイプシステムとの関係をまとめることとします。

5.1 Haskell における型変数とリスト

昔々、C#と Java が登場した当初、リストは配列（array）として表現され、よって、全ての型に対応するリストを定義するには、全ての型の継承元である `Object` 型の配列を定義し、その配列の要素を特定の型にキャストして利用する必要がありました。しかしそうすると、型としては `Object` 型でも、メモリ上では `string` 型や `int` 型ですから、知らずに違う型にキャストしてしまうコードが現れると、コンパイル時には予測できない矛盾として、ランタイムエラーを引き起こすこともありました。

現在では、ジェネリック（generic）の考え方が導入され、`List<T>` という形で全ての型に対応できるリストの定義が可能となっています。ご存知とは思いますが、ジェネリックという考え方は、扱う型が後で指定されるアルゴリズムのパターンを実現する考え方であり、例えば、`List<T>` 型における `T` は、`List<String>`、`List<Uri>` のように、後から特定の型を指定できる変数、つまり型変数（type variable）になります。表現上便利なだけではなく、ランタイムエラーとさよならもできたわけです。

一方で、Haskell においては、明示的に型を指定しない限り、全ての表現と変数の型は、コンパイル時に、その使用方法から演繹的に推論されます。例えば、変数が足し算で使用されていれば足し算に利用可能な型として推論され、リストとして使用されていればリスト型として推論され、矛盾があればコンパイルエラーとなる、といった具合です。白々しく矛盾を押し通す表現は容赦なくかつ正確に摘発されます。そして、この推論をタイプ推論（type inference）といいます。これは F#においても同じであり、現在では C#や C++の一部にも利用されています。`var` や `auto` のことですね。

タイプ推論についてはタイプクラスと関係がありますので後回しにして、ここでは、型変数の考え方を頭において、Haskell におけるリストについて見てみましょう。少々細かい説明になりますが、FP の考え方の理解に役立ちますので、おつきあいください。

はじめに、空のリストは [] で表されます。次に、1 つの要素、例えば、数値 1 を持つリストは、リスト専用の構築演算子（リストコンストラクタ）であるコロン（:）を使用し、1:[]と表現されるデータになります。この 1:[]という表現は、[1] と表現しても同じリストを表します。1:2:[]は、[1,2] と同じリストであり、1:2:3:[]は [1,2,3] と同じリストになります。また、[1,2,3] は、1:[2,3] とも表現可能です。

つまり Haskell におけるリストは、空のリスト（[]）もしくは 1 つ以上の要素を持つリストに対し、その左側に同じ型の要素を 1 つずつリストコンストラクタ（:）でつないだデータであり、このデータは、要素を[]の中にカンマ区切りで列挙する形でも表現できるということになります。

次に、リスト型の定義を見てみましょう（注 18）。

```
data [a] = [] | a : [a]
```

この定義は、型変数 a を取るデータ型 [] は、[]もしくは a : [a] の値となる、と読みます。

なお、型変数 a を [] の中にもつ [a]という表記はリスト型固有の特別な表記であり、正式には、データ型を表す名前の右側に型変数を表記します。よって、[a] は、[] a とも表記できます。

さて、この定義では、[a] が左辺と右辺の両方に登場していますね。これを頭において以下の説明を読んでください。

はじめに、定義上データ型 [a] は右辺の [] という値、つまり空のリストになり得ます。次に、データ型 [a] が [] になり得ますから、右辺の a : [a] という表現は、a : [] 、つまり a 型の要素を 1 つだけ持つリストにもなりうる、ということになります。次に、データ型 [a] は右辺の a : [a] という値にもなり得ますから、右辺の a : [a] は、a : a : [a] にもなり得ます。この解釈を繰り返すと、データ型 [a] は、空のリスト、もしくは、空のリストに 1 つ以上の a 型の値をリストコンストラクタ(:)でつないだ値、つまり a 型の要素を 0 個以上もつリストになり得る、ということになります。

以上、論理パズルを解くような説明でしたが、オブジェクト指向言語でいうところのジェネリックリストがたった 1 行、16 文字で表現されているというのは、衝撃的ではないでしょうか。

もちろん、この定義は 1:2:3:[] といったデータ構造を定義するだけであり、[1,2,3] という表現を 1:2:3[] というデータとして解釈したり、また、1:2:3[] というデータを [1,2,3] と表示したりするファンクション、それから、リストを扱うためのファンクションは別途定義されますが、重要な点は、データとデータに対する操作をカプセル化して高い生産性を実現するオブジェクト指向とは対照的に、個別静的に定義されたデータ型とデータマップ（ファンクション）の組み合わせにより矛盾のない処理設計を可能とする FP の考え方が、このリスト型の定義に表れているということです。

さて、話が大きくなりそうですのでここで止めて、リスト型を使用しながら、モノイドとファンクターの応用を見ることとします。Haskell では、タイプクラス（type class）という考え方を利用して、モノイドとファンクターの応用を行っています。

5.2 タイプクラス

タイプクラスという考え方は、プログラミング言語におけるデータ型に、後付けの形で一定の制約を与え、特定の機能を持つクラスとして分類する考え方です。

ここでいう制約は、例えばデータ型 *D* がタイプクラス *TC* の一員として分類されるためには、タイプクラス *TC* が型と名前を指定するメンバファンクションを、*D* に合った方法で定義しなければならないという制約になり、この制約を受ける *D* を、タイプクラス *TC* のインスタンスといいます。そして、1つのデータ型は、0 個以上のタイプクラスのインスタンスとなることができます。なお、タイプクラスは、メンバファンクションの既定の定義をすることもでき、その場合には、既定の定義をそのまま使用するか新たに定義するかについて、インスタンス側で判断することになります。オブジェクト指向言語を使用している方であれば、タイプクラスを、データ型が後付けで多重継承できるインターフェイスと捉えるとわかりやすいと思います。「後付け」ですから、柔軟性は申し分ありません。

5.3 Monoid タイプクラス

言葉だけではわかりにくいと思いますので、具体例として Monoid タイプクラスを見てみましょう。Haskell における Monoid タイプクラスの宣言は次の通りです。

```
1 class Monoid a where
2   mempty  :: a
3   mappend :: a -> a -> a
```

1 行目の class Monoid a がタイプクラスの名前と宣言の開始を示し、where 以降は、Monoid タイプクラスのインスタンスとなる型が定義すべきファンクションの名前及び型を指定しています。ここで、Monoid a における a は型変数であり、タイプクラスのインスタンスとなる型を表わしています。なお、1 行目の Monoid a にある a のスコープは宣言全体となりますので、where 以降に登場する a は全てインスタンスの型を表すことになります。

2 行目からはファンクションの名前と型の指定になります。2 行目の mempty はグループ論におけるモノイドのアイデンティティ要素に、3 行目の mappend は同じくグループ論におけるモノイドの二項演算子に相当します。

次に、この Monoid タイプクラスのインスタンス宣言を 1 つ見てみましょう。ここでインスタンスとなる型は、先ほど Haskell におけるリスト型として見た [a] 型です。[a] を Monoid タイプクラスのインスタンスとする宣言は次の通りです。

```
1 instance Monoid [a] where
2     mempty  = []
3     mappend = (++)
```

1 行目は、型 [a] を、Monoid タイプクラスのインスタンスとする宣言の開始を示し、where 以降は Monoid タイプクラスのインスタンスに必要なファンクションの定義となります。

2 行目はアイデンティティ要素の定義であり、空のリスト [] と定義されています。

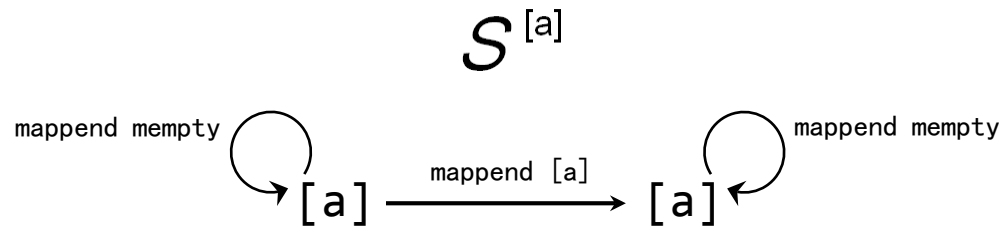
3 行目は二項演算子の指定であり、(++) と定義されています。この (++) というファンクションは、別途標準ライブラリで定義されている、リストの結合を行うファンクションになります。

なお、今後の例の理解に必要ですので 1 点解説します。(++)のように括弧を用いて定義されるファンクションは、括弧を除いた形で二項演算子（infix operator）として利用できます。例えば、足し算ファンクションである(+)は、そのまま使用した場合には前置演算子（prefix operator）として、(+) 1 2 のように使用でき、括弧を除いた場合、1 + 2 のように使用できます。

これを (++) でみると、(++) [1,2] [3,4] は [1,2,3,4] になり、[1,2] ++ [3,4] もまた [1,2,3,4] となります。ちなみに、括弧のない定義のファンクションについても、2 つの引数を取るファンクションであれば、ファンクションをバッククォート(`)で閉じることで二項演算子として利用できます。例えば、先に定義した add ファンクションの場合、add 1 2 は、1 `add` 2 と同じ値、つまり 3 として評価されます。

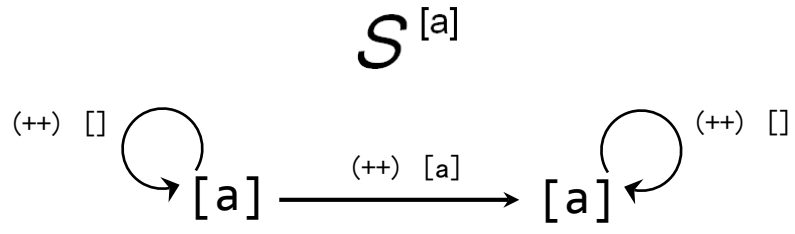
さて、話を Monoid [a] の宣言に戻します。ここで、グループ論のモノイドからカテゴリー論のモノイド、そして集合のカテゴリーへの変換を思い出せば、Monoid [a] は、[a] 型の取り得る全ての値の集合 1 つを対象とし、同対象のアイデンティティマップとして mappend mempty、エンドウマップとして mappend [a]を持つカテゴリーの定義と捉えられます。

これを、アイデンティティ規則の説明で使用した絵で表現すると次のようになり、モノイドとしてのカテゴリー $\mathcal{S}^{[a]}$ を表現することができます。



mappend[a] ∘ mappend mempty = mappend [a] = mappend mempty ∘ mappend[a]

もちろん、プログラミング言語における型が全てカーテシアンクローズトカテゴリーとして扱えられることを知っていて、[a] 型が (++) との関係でモノイドとして扱えることがわかっていれば、mempty の代わりに [] を、mappend の代わりに (++) を使用しても先ほどの絵は成り立ちます。



(++) [a] ∘ (++) [] = (++) [a] = (++) [] ∘ (++) [a]

とはいえ、[a] 型を Monoid タイプクラスのインスタンスとして宣言する価値は、Monoid 型と指定されているか、もしくは、mempty 値と比較されたり mappend の引数となったりしているために Monoid 型と推論される表現がある場合に、プログラマやコンパイラが [a] 型をその候補として認識できる点にありますから、[a] 型を Monoid タイプクラスのインスタンスとする定義は必要になります。

5.4 Functor タイプクラス

さて、次に、Functor タイプクラスを見てみましょう。宣言は次の通りです。

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

Functor タイプクラスのインスタンスに必要なメンバは fmap ファンクションだけですね。ファンクターの説明を思い出せば、ファンクターはカテゴリー間のマップを定義しますので、唯一のメンバである fmap が、カテゴリー間のマップを表すことになります。

それでは、fmap の(a -> b) -> f a -> f b という型を見てみましょう。

はじめに、1 行目にあった f がインスタンスの型ですが、f a と f b という表記から、f は型変数を取る型であることがわかります。例えば、先ほど見たリスト型である [a] のような型です。先の説明の通り、[a] は [] a の別表現ですから、リスト型で f に対応するのは [] ということになります。

次に、第 1 引数の (a -> b) は、a 型から b 型へのマップですね。ここで、宣言やファンクションの定義において、a と b のように異なる名前の型変数は、同じ型であることも、異なる型であることもあります。型変数は全ての型を表わしますから、当然ですね。よって、a と b が同じ型である場合にも、違う型である場合にも対応できなければなりません。このマップは、引数を 1 つ取り、同じ型から同じ型へのマップ、もしくは異なる型の間でのマップを行い、その結果を出力とするファンクションであるということになります。

そうすると、fmap の型は、カテゴリー間のマップとして (a->b) 型のファンクション（第 1 引数）を使い、マップ元となる f a 型の値（第 2 引数）を、f b 型の値にマップして出力するファンクションということになります。

それでは、既に知っている [a] 型について、インスタンス宣言を見てみましょう。

```
1 instance Functor [] where
2     fmap = map
```

先に見たとおり、Functor タイプクラス宣言の f に対応する型は [] ですね。次に、fmap は map と定義されています。ということは、map ファンクションの定義を見ないことには、[a] 型における fmap の定義はわかりません。また、map ファンクションの理解はカテゴリー論をソフトウェア設計やプログラミングに応用する際にとても重要ですので、その定義について少々詳しく説明します。

5.5 map ファンクション

さて、map ファンクションの定義は次の通りです。

```
1 map :: (a -> b) -> [a] -> [b]
2 map _ []     = []
3 map f (x:xs) = f x : map f xs
```


1 行目は、型の指定です。Functor タイプクラスのメンバ fmap と同じ型ですね。同じ型でなければ等号で結ぶことはできませんから、当然と言えば当然です。

2 行目と 3 行目は、ここまでで説明していないファンクションの定義方法になります。2 行目も 3 行目も map ファンクションの定義ですが、Haskell では、引数の値によって異なる評価パターンが必要な場合、そのパターンの数だけ同じ名前のファンクションの定義を行い、結果として 1 つのファンクションを表すことができます。ただし当然ながら、ファンクションの型は同じでなければならず、また、結果として引数の取り得る全ての値の組み合わせに対応できなければなりません。この引数の場合分けは、FP において利用されることの多い、パターンマッチング（pattern matching）の 1 つの方法です。そして、何故 map ファンクションで引数の場合分けが必要なのかは、すぐにわかります。

さて、2 行目では第 1 引数が _ であり、第 2 引数が [] の場合です。ここで、_（アンダースコア）は、パターンマッチングにおけるワイルドカードを示します。よって、この 2 行目は、第 1 引数の値に関係なく、第 2 引数が []（空のリスト）である場合にマッチするパターンであり、この場合に map ファンクションは [] と評価されます。

3 行目を見ると、map ファンクションの定義の右辺で map ファンクションが使用されています。リスト型の定義の時と似ていますね。このように、あるファンクションの評価方法の定義において、定義中のファンクションを再帰的に使用する方法を、リカージョン（recursion）といいます。Haskell では、C、C++、C#、Java といった命令型言語におけるループ処理（for、while、foreach 等）に相当するような繰り返しの実現に、リカージョンが使用されます。

さて、リカージョンの話を頭において 3 行目を詳しく見てみましょう。第 1 引数 f は、map ファンクションの型指定から、(a -> b) 型のファンクションであることがわかります。問題ありませんね。

問題は第 2 引数の (x:xs) です。皆さんは既に、この :（コロン）が何か知っていますね。x:xs は、引数がリストであり、かつ 1 つ以上の要素を持つパターンを示しています。例えば、第 2 引数が [1, 2, 3] であった場合に x:xs を当てはめると、1:[2, 3]、つまり、x が 1、xs が [2, 3] となります。[2, 3] の場合には x が 2、xs が [3]、[3] の場合には x が 3、xs が [] 、そして、第 2 引数が [] の場合には 3 行目のパターンにはマッチせず、2 行目のパターンにマッチすることになります。

よって、仮に第 2 引数の (x:xs) が [1, 2, 3] の場合、3 行目の表現は次のように展開できます。

```
map f (x:xs) = f x : map f xs

map f [1,2,3] = (f 1) : (map f [2,3])  ……  [1,2,3] = 1:[2,3]、よって x が 1、xs が[2,3]
              = (f 1) : (f 2) : (map f [3])
              = (f 1) : (f 2) : (f 3) : (map f [])
```

ここで map ファンクションの第 2 引数が [] の場合、定義における 3 行目のパターン x:xs にはマッチせず、2 行目のパターン map _ [] = [] にマッチし、 map f [] は [] と評価されます。よって、

```
              = (f 1) : (f 2) : (f 3) : []
              = [f 1, f 2, f 3]
```

となります。以上が、map ファンクションにおけるリカージョンの展開解説です。

こうしてみると、定義における2行目、つまり map _ [] = [] の存在価値がわかるでしょう。命令型言語でいえば、for ループや while ループにおける終了条件の役割に相当しますね。このように、リカージョンにおける展開の終了に使用されるパターンを、ベースケース（base case）といいます。

さて、結果として map ファンクションは、第2引数で指定されたリストの全ての要素に対し、第1引数で指定されたファンクション f を適用した要素のリストとして評価されるということになります。

そうすると、カテゴリー論の説明の際に使用した、「8 時間後」マップの適用例、map f [1, 2, 2, 8, 10, 6, 4, 3] を、[f 1, f 2, f 2, f 8, f 10, f 6, f 4, f 3] と表現できる論理がお分かりになったと思います。そして、既に定義した addTime を使用すると、「8時間後」マップは addTime 8 と書けますから、この適用例は map (addTime 8) [1, 2, 2, 8, 10, 6, 4, 3] と表現でき、評価結果が [9, 10, 10, 4, 6, 2, 12, 11] となることもお分かりになると思います。

以上から、カテゴリー論におけるマップと Haskell における map ファンクションの考え方が同じであることもお分かりになったと思います。なにしろ、map の第2引数が addTime 8 、つまり S^A におけるエンドウマップですからね。

さて、忘れてしまうところでしたが、リスト型を Functor タイプクラスのインスタンスとする定義の話をしていました。これですね。

```
1 instance Functor [] where
2     fmap = map
```

ここで fmap と等号で結ばれている map ファンクションの定義は理解できました。ところで、カテゴリー論におけるファンクターは、カテゴリー間のマップでしたね。そうすると、この fmap = map という定義はカテゴリー論的にどう理解すればよいのでしょうか。これを理解するためには、サブカテゴリー（subcategory）という概念の理解が必要です。そして、このサブカテゴリーを理解した時に、本書第1回の内容が頭の中で綺麗にまとまります。

5.6 タイプシステムとサブカテゴリー

サブカテゴリーとは、読んで字のごとく、カテゴリーの1部分であり、カテゴリーでもあります。カテゴリーの1部分であるということは、あるカテゴリーの対象のうち、少なくとも1つ以上の対象を持つということであり、そのものがカテゴリーであるということは、サブカテゴリーに含まれる対象についてのマップがあるということです。もちろん、それぞれの対象におけるアイデンティティマップとマップの合成も存在することになります。この考え方をプログラミング言語のタイプシステムにあてはめると、次のようになります。

はじめに、タイプシステムを1つのカテゴリーと捉えた場合、タイプシステムは、タイプシステムに含まれる全ての型（タイプ）を個別の対象（集合）とする集合のカテゴリーです。つまり、このカテゴリーを C とすると、C の対象は全ての型の数だけあり、対象毎にアイデンティティマップがあり、そしてエンドウマップも含めた対象間のマップ及びマップの合成がある、ということになります。

次に、C のサブカテゴリーを考える場合には通常、1つの型（対象）と、その型に対するアイデンティティマップ、それから1つ以上のエンドウマップを持つサブカテゴリーを考えます。例えば、

Integer 型を対象とするサブカテゴリーは、1 つのアイデンティティマップと、1 つ以上の Integer -> Integer 型のエンドウマップを持つことになります。

1 つの型のみを対象とするサブカテゴリーを考える理由は、通常のタイプシステムでは異なる型間のマップに型変換を伴うため、型変換を必要としないマップをエンドウマップとして、型変換を必要とするマップをサブカテゴリー間のマップとして、つまり、異なる型を異なるサブカテゴリーの対象と捉えることで、カテゴリー論が応用しやすくなるためです。このように 1 つの型のみを対象とするサブカテゴリーを考えると、対象とする型を T とした場合のサブカテゴリーを $\mathcal{S}^T$ として、また、異なる型間のマップをファンクターとして扱うことができます。

次に、Haskell におけるタイプクラスについては、タイプクラス型を対象とするサブカテゴリー、つまり、概念上同じ種類として扱うことのできる 2 つ以上の型を表現できる型を対象とするサブカテゴリーを考えることもできます。例えば、Haskell の標準ライブラリにおける Float 型と Double 型は、ともに分数を表す種類の型として扱うことができるため、両者ともに Fractional（分数）タイプクラスのインスタンスとなっていますが、ここで Fractional 型を対象とするサブカテゴリーを考えると、概念上 Float 型と Double 型の 2 つの型を表すことのできる対象をもつサブカテゴリーになります。

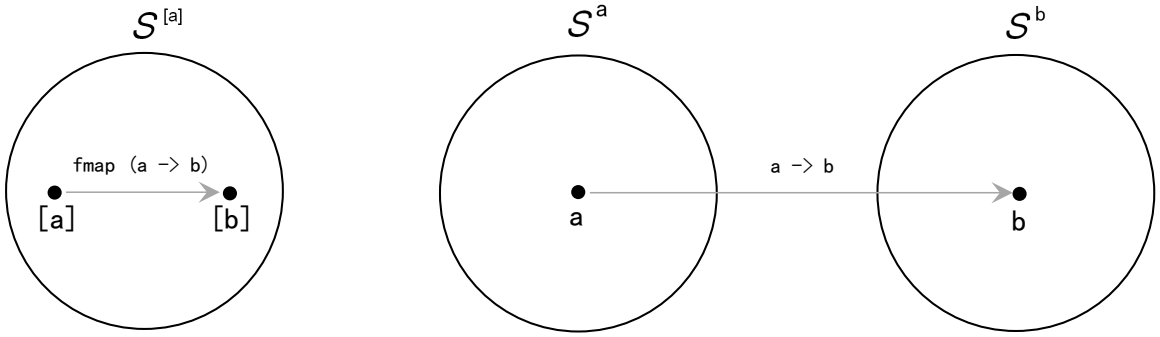
ここでタイプクラス一般を TC とすると、タイプクラスを対象とするサブカテゴリーは $\mathcal{S}^{TC}$ として扱うことができます。そしてこのサブカテゴリーにおけるエンドウマップの例が、Functor タイプクラスのメンバである fmap となります。fmap が型の指定のみで定義を伴わない、いわば仮想エンドウマップである理由は、このカテゴリーが概念上のカテゴリーであり、現実的にはタイプクラスのインスタンスの型を対象とするサブカテゴリー $\mathcal{S}^T$ として捉えることになり、インスタンス側で定義された fmap が同サブカテゴリーにおけるエンドウマップとなるためです（注 19）。

さて、タイプシステムにおけるサブカテゴリーの考え方がわかれば、リスト型を Functor タイプクラスのインスタンスとする宣言における、fmap = map という定義が自然と理解できます。

fmap ファンクションの型 (a -> b) -> f a -> f b において、(a -> b) は、サブカテゴリー $\mathcal{S}^a$ からサブカテゴリー $\mathcal{S}^b$ へのカテゴリー間のマップであり、 f a -> f b は、リスト型の場合には [a] -> [b] となりますから、サブカテゴリー $\mathcal{S}^{[a]}$ におけるエンドウマップとなりますね。

ここで、fmap ファンクションを 2 つの引数をもつファンクションと捉え、これをカリー化すると、fmap (a -> b) が (f a -> f b) とイコールになりますから、 $\mathcal{S}^{[a]}$ におけるエンドウマップは、 fmap (a -> b) となります。

これを絵にすると、次のようになります。なお、a 及び b は型変数ですから、a 型と b 型が同一の型であっても、異なる型であってもよいことになります。



以上から、もともとリスト型の要素の変換を定義している map ファンクションが、 $\mathcal{S}^{[a]}$ におけるエンドウマップを定義しているとみなされ、よって、fmap = map という定義が成立するわけです。

そして、カテゴリー論的には、ファンクターである $\mathcal{S}^{[a]}$ が、そのエンドウマップ fmap (a -> b) において、サブカテゴリー $\mathcal{S}^{a}$ と $\mathcal{S}^{b}$ の間のマップ、つまりファンクターが定義すべきカテゴリー間のマップを保持している、ということになります。

付け加えるならば、タイプシステムにおいて、サブカテゴリー $\mathcal{S}^{T}$ における対象 T が別の型にマップされるということは、その対象となる型だけではなく、アイデンティティマップ、マップ、マップの合成も全て、マップ先の型に合わせてマップされると捉えられますから、 a -> b という表現は、カテゴリー論におけるファンクターのマップを表現していることになります。

さて、ここまでで、カテゴリー論の基本的な考え方、FP におけるファンクション（ラムダ抽象）の考え方、そして、カテゴリー論、ラムダ抽象、及び、タイプシステムとの相互関係が理解できたと思います。なお、Haskell を使用したマップの合成の具体例は、複数のファンクションを組み合わせる方法の 1 つとして、第 2 回で取り扱う予定です。

6【 第 1 回の要点 】

1 カテゴリーは、1 つ以上の対象、各対象におけるアイデンティティマップ、1 つ以上のマップ及びマップの合成により構成され、アイデンティティ規則及びアソシエイティヴ規則を満たす。

2 集合のカテゴリーにおけるマップは、ある集合の全ての要素について、各要素から、同じ集合もしくは別の集合の 1 つの要素への静的な（変わることのない）1 対 1 かつ 1 方向の全ての矢印の集合を意味し、矢印の先となる要素は全て 1 つの集合内になければならない。

3 FP におけるファンクションはラムダ抽象であり、2 つ以上の引数を持つファンクションはカリー化されたファンクション、つまり、第 1 引数を入力とし、残りの評価に必要なファンクションを出力するファンクションとみなされる。

4 集合のカテゴリーは全てカーテシアンクローズドカテゴリーである。

5 カーテシアンクローズドカテゴリーでは、1 つの引数を取るファンクション、もしくは 2 つの引数を取るファンクションのカリー化された形でマップを表現でき、3 つ以上の引数を設定することが妥当な場合は、マップの合成により表現が可能となる。よって FP におけるファンクションは、カテゴリー論におけるマップと捉えられ、上記 2 の通りの静的かつ 1 方向のデータマップを表す。

6 1 つの集合を対象とする集合のカテゴリー $\mathcal{S}^{\circlearrowleft}$ は、カーテシアンクローズドカテゴリーであると同時にモノイドとして扱うことができる。

7 タイプシステムは、それぞれの型（タイプ）を、その型の取り得る全ての値の集合と捉えた集合のカテゴリーを構成し、よってカーテシアンクローズドカテゴリーとなる。

8 タイプシステムにおいて、1 つの型（タイプ）のみを対象とするサブカテゴリー $\mathcal{S}^{T}$ は、上記 6 の通り、カーテシアンクローズドカテゴリーであると同時に、モノイドとして扱うことができる。

9 タイプシステムにおいて、1 つの型（タイプ）のみを対象とするサブカテゴリー $\mathcal{S}^{T}$ 間のマップをファンクターと捉えると、カテゴリー論を応用しやすい。

10 タイプシステムが無い場合でも、集合のカテゴリーさえ正しく構成できれば、本書第 1 回の内容は応用可能となる。

7【 第１回のおわりに 】

第１回の内容は、つまるところ $\mathcal{S}^T$ という表記に集約されます。

$\mathcal{S}^T$ という表記とその組み合わせで、カテゴリー、対象、集合、マップ、矢印、アイデンティティマップ、エンドウマップ、マップの合成、アソシエティビティ、ラムダ抽象、ファンクション、カリー化、カーテシアンクローズドカテゴリー、モノイド、アイデンティティ要素、ファンクター、そしてタイプシステムの捉え方、つまり、カテゴリー論と FP の基本的な考え方、及び、タイプシステムとの相互関係が全て説明できるからです。

シンプルな表記にまとまりましたね。

$\mathcal{S}^T$ のイメージとして記憶すれば、おそらく、本書第１回の内容を忘れることはないと思います。

もし皆さんが、身の回りにあるプログラミング言語、ソフトウェア製品やサービスにおける $\mathcal{S}^T$ の応用に気が付き、その設計思想をシンプルかつ体系的に捉えることができたり、また、ソフトウェアアーキテクチャやアルゴリズム設計の評価や見直しに生かすことができたりしたら、本書第１回の目的は達成できたことになります。

シンプルであることとよい設計であることが等価とは限りませんが、シンプルに捉えられない設計やシンプルに説明できない設計があるとしたら、その応用において矛盾をきたす危険性を秘めている、その可能性を疑うべきだと思います。

とはいえ、第１回の内容だけでは応用例に乏しいこともありますので、第２回以降で補完していければと考えています。ただし、発表時期は未定です。

なお、本書の利用等につきましては、巻末の「本書の利用について」を参照ください。

以上、お読みいただき、ありがとうございました。

2014年3月1日
戸﨑　貴裕

【 注記 】

1　本書では、これまで数学にはあまり関心が無かった、という方でも理解できるように努めています。そのため、数学知識のそれなりにある方にとっては煩雑な説明や表現も多くなっていますが、その点はご了承ください。それから、英語名称に対応する日本語名称について、例えば、Functional Programming をファンクショナルプログラミングもしくは関数型プログラミングと表記したり、Category Theory をカテゴリーセオリー、カテゴリー論もしくは圏論と表記したり、そもそも英語名称が複数存在したりと、本書の題材に関しては、同じ対象を表す名称が２つ以上存在する場合があります。本書では、英語名称と日本語名称との対応がわかりやすくなること、また、情報収集を行う際に手間を生じさせないことを目的とし、なるべく英語表記が予想できるカタカナを使用したり、英語表記と日本語表記を併記したりしています。なお、それだけで目的が達成できないと思われる場合には、別途注記で解説している場合もあります。

2　この定義は、巻末の参考文献１をもとに、海外の大学関係者等が公表している資料を参考に日本語としてまとめましたが、出典により、定義の表現方法もしくは表記方法（記号）の異なる場合があります。また、出典により、本書の定義における２及び３がまとめて書かれている場合や、５の

合成マップの表記における白丸（∘）を二項演算子（binary operator）と捉えて、4、6 及び 7 を規則として表記している場合等がありますが、カテゴリーの構成要素及び規則は同じになります。

3 ここで、「静的な（変わることのない）」という部分が気になった方もおられると思います。現実的には、例えば、「散歩の時間」マップ g における散歩の時間が変更されることもあるでしょう。しかし、仮に、子犬の名前から散歩の時間への対応関係が 1 つでも g と異なる場合、そのマップは g とは異なるマップとみなされます。この考え方は、例えば、円の面積を求める関数 $Area(r) = \pi r^2$ において、半径 r の値が同じであれば、面積 $Area(r)$ の値が必ず同じ値になるように、入力と出力の関係は 1 対 1 で変わらない、という数学における関数（ファンクション）の考え方と同じです。そして、この考え方をコンピュータサイエンスの世界に反映させた考え方を、参照透過性（Referencial Transparency）といいます。参照透過性の考え方の重要性については第 1 回でも触れることがありますが、現実の問題との折り合いをつける具体的な考え方については、第 2 回以降の話題となる予定です。それから、「散歩の時間」の例について、仮に子犬それぞれの散歩の時間が 2 回以上ある場合、もしくは、仮に 2 匹の子犬が同じ時間に散歩する場合で矢印を逆にした場合、つまり、散歩の時間から子犬を特定する場合においては、要素間で 1 対 2 以上（1 対 n）の関係が出てきます。このような関係を、ソフトウェア設計においてリストや一定の規則を持つ文字列のような 2 つ以上の要素に対応できる方法で表現する場合、集合のカテゴリーについては、「**0** 匹以上の子犬のリスト」の集合や「**0** 個以上の時間のリスト」の集合を対象とするカテゴリーを考えることもできます。リストとカテゴリーの関係については、後に本文で詳しく見ることになりますが、その一方で、要素間の関係が複雑な問題に対し、1 対 n や n 対 n の関係をそのまま設計に反映させるよりも、1 対 n や n 対 n の関係を 1 対 1 の関係に分解して必要に応じてマージするといったパターン、つまり 1 対 1 の関係の応用で問題を表現する発想が、本書の背景としてお話ししたクラウド時代のデータ処理に求められる発想となります。アルゴリズム設計においてはよく使用される、分割統治（divide and conquer）の発想ですね。考えるカテゴリーが複雑になってしまった場合には、この分割統治の考え方に立ち戻るべきでしょう。ちなみに、要素間の対応が 1 対 1 の場合でも、全射（surjective function）、双射（bijective function）、単射（injective function）という分類が必要になる場合もあります。この分類についての解説は本書ではしませんので、ご興味のある方は Wikipedia 等で調べてみてください。

4 カテゴリーは、万物のパターンを探究し表現する学問である数学において、集合、グループ、トポロジーをはじめとする理論を集約できる考え方であることから、数学的森羅万象（mathematical universe）とも呼ばれます。つまり、対象、マップ、マップの合成によってパターンを表現するというこのシンプルな考え方は、およそパターンというものが重要である全ての分野において、最強の思考道具ということになります。

5 Haskell では、型の指定にはコロン（:）を二つ使い、矢印は、マイナス記号（'−'）と大なり記号（'>'）の組み合わせで表します。

6 この例では明示的にファンクションの型を指定していますが、Haskell では、ファンクションの型を指定しなくとも、ファンクションを定義することができます。その場合、ファンクションの型はコンパイラによって演繹的に推論されます。この機能をタイプ推論（type inference）といいます。タイプ推論については後程タイプクラスとともに本文で説明します。なお、**add a b = a + b** という定義と、**add (a, b) = a + b** という定義は、別の型を持つことになります。後者の場合、引数

の（a, b）は Tulpe（タプルもしくはチュープル）という型になり、引数は 1 つとみなされるからです。これについては本文において注 11 への参照が登場するまでお待ちください。

7 Haskell において、λ はその形に似せてバックスラッシュ（'\'）で代用するのですが、日本語環境ではこの文字コードが円記号（'¥'）として表示される場合が多くあります。λ に似ていないのでとても違和感がありますが、文字コードが同じであればコンパイルは可能です。ちなみに、どうしてもバックスラッシュを表示したい方は、ご使用の OS やエディタの設定で回避できる場合がありますので、"バックスラッシュ" とご自身の環境（例えば "Windows" や "Mac"）の組み合わせを Web 検索してみましょう。

8 GHCi は、Haskell の開発環境パッケージである The Haskell Platform に含まれており、同パッケージは、http://www.haskell.org/platform/ からダウンロードできます。対象 OS は Windows、Mac、Linux です。詳細は同ページでご確認ください（英語）。なお、GHCi の既定プロンプトは Prelude> になります。プロンプトは、:set コマンドで変えることができるほか、GHCi の実行パスに、:set コマンドの行を含む .ghci という名前のテキストファイルを作成することで変更できます。例えば、本書のようにプロンプトを ghci> とする場合のコマンドは、:set prompt "ghci> " となります。

9 GHCi においては、等記号（=）による名前の定義の際に let が必要になりますが、ソースファイルにおいては必要ありません。

10 複数の引数を持つファンクションの一部の引数に値を指定して評価することを、部分適用（partial application）といいます。部分適用は、適用されない引数が残っている状態であれば、1 度に 2 つ以上の引数の値を指定する評価も含みます。なお、部分適用において、引数は左側から適用されていきます（left associative）。これは、add a b = a + b と表現した場合でも同じです。add 3 の 3 は、引数 a に対して適用されます。

11 Haskell において、等記号（=）によって定義された名前は定数を表し、その名前のスコープ内で値を変更することはできません。例えば、C#や Java における int i = 0; i = i + 1; といった、変数に対する値の「代入」という概念とは異なります。ただし、GHCi においては、let output = 1 とした後に let output = 2 とすることは可能です。これは、名前の再定義として認識されます。このような再定義はコマンドラインコンパイラならではの仕様となり、ソースコードでは通用しませんのでご注意ください。ちなみにこの、変更することのできない値（immutable value）という考え方について、本文の流れからは外れてしまいますが、第 2 回で予定している内容がどのようなものであるかわかりやすくなりますので、少々説明しておきます。さて、変更できない値、つまり不変値を採用する理由には、データの再利用によるパフォーマンス向上の可能性を理由とする場合もありますが（.NET の string のように。）、FP の考え方としては、論理学に通じていると考えることが必要です。簡単にいえば、前提が変更されてしまうと展開された論理の結果が変わってしまう、それは困る、ということです。新しい値を扱う必要があれば、それは別の定数として定義する、つまり、別の前提が必要であれば、それは別の前提として扱うという考え方です。現実のソフトウェアに当てはめれば、値は事実であり、事実は 1 つである。事実が途中で変更されてしまっては、結論が変わってしまうかもしれない、それは困る、ということです。説明がくどいように思えるかもしれませんが、処理の途中で前提、つまり結果に影響を及ぼす値が変わらないということは、矛盾の無い論理展開を実現する上で非常に重要なことなのです。この考え方は、同じ入力であ

れば同じ出力が期待できるという参照透過性（Referential Transparency）の考え方や、No Side Effect の考え方にも通ずるものがあります。一方で、オブジェクト指向におけるオブジェクトは、状態と操作をカプセル化し、操作は状態を変化させる、つまり、オブジェクトを値と捉えれば、値が変更されることが前提となっていますから、オブジェクト指向は、値を変更しないという FP の考え方とは相容れない考え方が基本になっているということになります。とはいえ、FP やカテゴリー論の考え方がオブジェクト指向の世界で実現できないということはありません。そして、オブジェクト指向がメジャーな設計思想となった理由はその生産性にありますから、オブジェクト指向の世界に FP とカテゴリー論の考え方を取り入れることで、高い生産性を保ちつつ、本書冒頭でお話ししたようなクラウド時代のデータ処理に求められるソフトウェア設計が実現できるのです。一方で、オブジェクト指向言語に慣れた方で FP の経験の無い方からすれば、そもそも、C#における const や Java における final で定義された名前しかない状態でどうやったらプログラムが書けるのかと疑問に思うかもしれません。この FP とオブジェクト指向の違い、それから、オブジェクト指向における FP とカテゴリー論の考え方の応用が第 2 回のテーマとなる予定です。

12 Haskell には、標準で uncurry というファンクションがあります。読んで字のごとく、カリー化の解除を行うファンクションです。さて、カリー化の解除とはどういうことでしょう。これまでに使用してきた add ファンクションで実験してみましょう（解説はコードの後で。）。

```
ghci> let add = \a -> \b -> a + b
ghci> :t add
add :: Integer -> Integer -> Integer
ghci> let add' = uncurry add
ghci> :t add'
add' :: (Integer, Integer) -> Integer
ghci> add' (3,4)
7
```

1 行目は、これまでに使用してきた add ファンクションの定義です。2 行目で型を問い合わせると、3 行目で Integer -> Integer -> Integer と確認できます。4 行目では、add ファンクションに対してカリー化の解除（uncurry）を適用し、結果を add' として定義しています。ちなみに、add' は、「add プライム」と読み、これもファンクションの名前として認識されます。記号プライム（'）は、数学や Haskell において、同じ評価となるファンクションを別の表現で定義する場合や、関連性のある別のファンクションを定義する場合によく使用されます。さて、5 行目で add' の型を問い合わせると、6 行目での答えは、(Integer, Integer) -> Integer となっています。この(Integer, Integer)のように括弧でくくった型を Tuple（タプルもしくはチュープル）といい、0 個以上の要素の組を表します。この場合は、2 つの Integer 型の組を表しています。例えば、7 行目の例を見ると、この Tuple 型の引数として、(3,4)を指定すると、8 行目の答えは 7 となります。そうすると、uncurry は、2 つの引数を持つ add ファンクションを、もともとの 2 つの引数を両方指定しないと評価できない形、つまり、add (a, b) = a + b の形に変換したということになります。そして、上記結果から、(a,b) は Tuple 型であり、add と add' は全く別の型を持つファンクションであることになります。

13 本文では、本文の絵の中で add a もしくは \b -> a + b で表した矢印が、カテゴリー論においてどのように解釈されるのか、何を意味しているのかについて、実は説明していません。本文においては、カーテシアンクローズドカテゴリーの性質とラムダ抽象（カリー化）との関係が理解できれば問題がないためです。そしてこの注記は、そうはいわれても釈然としない、つまり、実は説明の無かったことに気付いた方のための注記です。さて、add a もしくは \b -> a + b は、b から a+b への矢印です。そして、両者ともに、a の値が定まらないことには、マップとして機能しません。この関係を一般化し、例えば、集合 *A*、*B* 及び *C* を対象とするカテゴリーと、このカテゴリーにおける、*A* -> *B* -> *C* という型を持つファンクション *f* を考えてみます。*f* をカリー化すると、*A* -> (*B* -> *C*) と解釈されます。ここまでは問題ないと思います。そして、カテゴリー論において、このファンクションは、*A* -> C^B と解釈されます。ここで、C^B は、対象 *B* から対象 *C* へのユニバーサルマップ（ユニバーサルモーフィズム、もしくは普遍射）、すなわち、*B* から *C* への全てのマップの集合を対象として捉えた表記になり、このような対象を、指数対象（exponential object）と言います。そうすると、本文の絵の中で add a もしくは \b -> a + b で表現した矢印は、実は 1 つのマップではなく、b から a+b への全てのマップの集合を表す対象（指数対象）として解釈されることになり、この場合は全てのドメインとコドメインが $\mathbb{Z}$ ですから、add ファンクションを $\mathbb{Z} \to \mathbb{Z}^{\mathbb{Z}}$ と解釈した際の、$\mathbb{Z}^{\mathbb{Z}}$ として表されることになります。そして、a の値が定まることで、普遍的なマップではなく、1 つのマップとして定まることになります。

14 Haskell には、id というファンクションが標準ライブラリで用意されています。型は、id :: a -> a、定義は id x = x になります。

15 ここで示したモノイドが満たすべき要請は、グループ論におけるグループが満たすべき 4 つの要請（group axioms）のうちの 3 つです。4 つ目は、インバース要素（Inverse element、逆元）になります。よって、全てのグループはモノイドということになります。これら要請のうち、クロージャーについては、集合が閉じていることを前提とし、要請として数えない場合もあり、その場合には要請が 3 つとして説明されます。なお、クロージャーは $\forall a,b \in G : a \bullet b \in G$、アソシエティビティは $\forall a,b,c \in G : a \bullet (b \bullet c) = (a \bullet b) \bullet c$、アイデンティティ要素は、$\exists e \in G : \forall a \in G : e \bullet a = a \bullet e = a$、と表記したほうが、このような表記法をご存知の方にはわかりやすいと思います。ちなみに、これらの表記において、∀は「全ての」を意味し、∃は「存在する」、: は「右の評価が真となる」、∈ はご存知と思いますが、集合における包含関係を示し、A ∈ B は、集合 A の全ての要素が集合 B に含まれることを意味します。

16 念のための注記です。万が一ここで、「また足し算？」と思った方がいるとしたら、足し算に失礼です。足し算は、マイナスの数を足すことで引き算に、同じ数を足し続けることで掛け算に、同じマイナスの数を足し続けることで割り算を表すことができ、さらには、数だけではない概念、例えば後に見るようなリストの足し算や文字列の足し算といった全ての概念に応用可能なすばらしい概念なのです。あくまで念のための注記でした。

17 ここでは、二項演算子と要素のリストを引数とし、クロージャー及びアソシエティビティの確認、並びに、アイデンティティ要素の特定を行う Haskell のファンクションの 1 例を示しておきます。本書の目的からすれば、この例を理解する必要はありません。


```
checkClosure :: (Eq a) => (a -> a -> a) -> [a] -> Bool
checkClosure f [] = False
checkClosure f xs = and [elem c xs | c <- [f x y | x <- xs, y <- xs]]

checkAssocietivity :: (Eq a) => (a -> a -> a) -> [a] -> Bool
checkAssocietivity f [] = False
checkAssocietivity f xs = and [f a (f b c) == f (f a b) c | a <- xs, b <- xs, c<- xs]

findIdentityElement :: (Eq a) => (a -> a -> a) -> [a] -> Maybe a
findIdentityElement f [] = Nothing
findIdentityElement f (x:xs)
    | and [f x a == f a x && f a x == a | a <- (x:xs)] = Just x
    | otherwise = findIdentityElement f xs
```

18 データ型の定義は data 宣言によって行われ、等号（=）の左辺をタイプコンストラクタ（type constructor、型構築子。型変数を取る場合には型変数を除いた部分。例えば、リスト型の場合には [] 。）といい、右辺のパイプ（ | ）で区切られた各表現を、データコンストラクタ（data constructor、データ構築子）といいます。ここで、右辺にはデータ型の取りうる値として、データコンストラクタを全て指定することになり、データコンストラクタが2つ以上存在する場合には、パイプ（ | ）で区切ります。データコンストラクタでは、リスト型の例のように、定義対象のデータ型を再帰的に使用することができます。なお、タイプコンストラクタの名前はアルファベットの場合は大文字で始め、型変数の名前は小文字で始めます。

19 ちなみにこの考えをオブジェクト指向のジェネリックインターフェイスに当てはめてみると、.NET における LINQ と Rx の背景にある考え方が 1 つ理解できます。このお話は第 2 回で、モナド（Monad）の話とともに話題となる予定です。

【 主な参考文献等 】

1. 「Conceptual Mathematics Second Edition」TF.William Lawvere, Stephen H. Schanuel, Cambridge University Press, 2009 ISBN: 978-0-521-71916-2
2. 「Haskell 2010 Language Report」 Simon Marlow(editor), belongs to the entire Haskell community.
3. 「Lean You a Haskell for Great Good!」 Miran Lipovaca,No Starch Press, 2011 ISBN: 978-1-59327-283-8 （左記リンクに無料オンライン版あり）
4. 「The Haskell Road to Logic, Maths and Programming」 Kees Doets, Jan Van Eijck, King's College Publications, 2004 ISBN:0-9543006-9-6
   「The Haskell Road to Logic, Maths and Programming」（無料オンライン版）
5. 「Tractatus Logico-Philosophicus」 Ludwig,Wittgenstein Dover Publications, 2013 ISBN: 978-0-486-40445-5

なお、本書冒頭の引用文は上記 5 からの引用であり、代表的な日本語訳は、「世界は成立していることがらの総体である。」、「論理学は学説ではなく、世界の鏡像である。」、「数学は論理を探究するひとつの方法である。」です（岩波文庫「論理哲学論考」、ウィトゲンシュタイン著、野矢茂樹訳。引用の最後に 1、6.13、6.234 とあるのは、同書における命題の番号。）。

そうすると、数学は世界を正確に映し出して理解するためのひとつの方法であるということになります。ウィトゲンシュタインがそう記したのは既に 90 年以上も前ですが、一方で現代、カテゴリー論における Category は、「数学的森羅万象」（Mathematical Universe）とも呼ばれます。森羅万象とは、宇宙に存在する全ての物を意味しますから、ウィトゲンシュタインの言葉と、現代の数学者の言葉に共通の認識を見出すことができます。

【 本書の利用について 】

本書の商用利用を禁じます。商用利用以外の目的における配布は形態を問わず自由ですが、Haskellの仕様とみなされるソースコード部分のみを転記する場合を除き、本書の内容の一部を転記する場合には、転記の形態を問わず、著作権表示とともに本書が出典であることを明記してください。なお、本書の利用により生じた損害は、いかなるかたちにおいても補償いたしません。

その他本書についてのご意見、ご要望、間違いの指摘、お問い合わせ等は、ユーザー名 qqnn4cfp9、ドメイン juno.ocn.ne.jp のメールアドレス宛にお願いいたします。なお、返信の保証は致しませんのでご了承ください。

