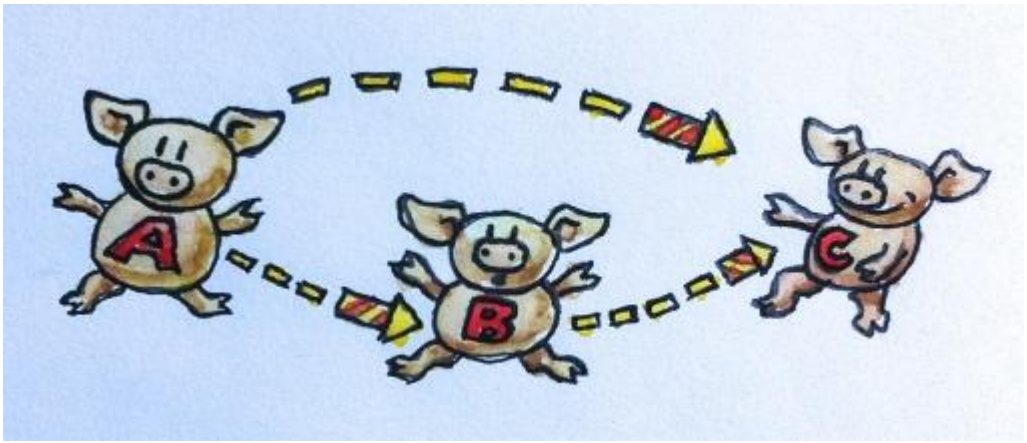
Category: The Essence of Composition

前回の記事、「プログラマのためのカテゴリー理論序説」に対する好意的な反応に圧倒されました。同時に、人々が私に寄せている期待がいかに大きいかを実感し、怖くなりました。私が何を書いても、多くの読者が失望してしまうのではないか、と。もっと実用的な本にしてほしいという読者もいれば、もっと抽象的な本にしてほしいという読者もいる。C++が嫌いですべての例を Haskell で書いてほしいという人もいれば、Haskell が嫌いで Java での例を要求する人もいる。また、説明のペースがある人には遅すぎ、ある人には速すぎることも分かっている。この本は完璧な本ではないでしょう。妥協の産物であろう。私が望むのは、読者の皆さんと私の「ハッ！」とする瞬間を共有できることです。まずは基本的なところから。

カテゴリーとは、恥ずかしいほど単純な概念である。カテゴリは、オブジェクトとその間を行き来する矢印で構成されている。だから、カテゴリは絵で表すのが簡単なのだ。モノは丸か点で描けるし、矢印は...矢印である。(たまに、モノをブタに、矢を花火に見立てることもある)。しかし、カテゴリの本質は構成である。あるいは、構図の本質がカテゴリーであるとも言える。矢印は合成だから、物体 A から物体 B に向かう矢印と、物体 B から物体 C に向かう矢印があれば、A から C に向かう矢印（その合成）があるはずだ。



Arrows as Functions

これはもう、抽象的なナンセンスすぎますか？絶望することはありません。具体的な話をしよう。モルヒズムとも呼ばれる矢印を関数として考えてみよう。A 型を引数にとり、B 型を返す関数 f と、B 型を引数にとり、C 型を返す関数 g があるとします。

数学では、このような合成は関数と関数の間に小さな丸をつけることで表現されます： g◦f。合成の順序が右から左になっていることに注意してください。人によってはこれが混乱する。Unix のパイプ記法に馴染みがあるかもしれませんね。

```
lsof | grep Chrome
```

や F#のシェブロン >>のように、どちらも左から右へと進みます。しかし、数学や Haskell では、関数は右から左へ合成される。g◦f を "f の後の g "と読むと分かりやすい。

これをさらに明確にするために、C のコードを書いてみましょう。A 型の引数を取って B 型の値を返す関数 f があります。

```
B f(A a);
```

ともう一つ。

```
C g(B b);
```

その構成は

```
C g_after_f(A a)
{
    return g(f(a));
}
```

ここでも右から左への合成が行われています： g(f(a)); 今度は C 言語です。

C++標準ライブラリに 2 つの関数を受け取ってその合成を返すテンプレートがあると言いたいところですが、そんなものはありません。そこで、気分転換に Haskell を試してみましょう。以下は、A から B への関数の宣言である。

```
f :: A -> B
```

同様に

```
g :: B -> C
```

その構成は

```
g . f
```

Haskell のシンプルさを目の当たりにすると、C++でストレートな関数概念を表現できないことが少し恥ずかしくなります。実際、Haskell では Unicode 文字が使えるようになるので、composition をこう書くことができるようになる。

```
g ∘ f
```

Unicode のダブルコロンや矢印も使用可能です。

```
f ∷ A → B
```

というわけで、Haskell のレッスン第 1 回目です。ダブルコロンの意味は「...の型を持っている」です。関数の型は、2 つの型の間に矢印を挿入することで作成されます。2 つの関数の間にピリオド（または Unicode の円）を挿入すると、2 つの関数が合成されます。

Properties of Composition

どのカテゴリーにおいても、構成が満たさなければならない極めて重要な 2 つの性質がある。

1. 合成は連想的である。合成可能な（つまり、その対象が端から端まで一致する）3 つのモルヒズム、f、g、h があれば、それらを合成するのに括弧は必要ない。これは数学の表記法では次のように表される。

```
h∘(g∘f) = (h∘g)∘f = h∘g∘f
```

擬似）Haskell で。

```
f :: A -> B
g :: B -> C
h :: C -> D
h . (g . f) == (h . g) . f == h . g . f
```

(「擬似的」と言ったのは、関数には等式が定義されていないからです)。

関数を扱う場合、連想性はかなり明白ですが、他のカテゴリではそれほど明白ではないかもしれません。

2. 2. すべてのオブジェクト A に対して、合成の単位である矢印が存在する。この矢印はオブジェクトからそれ自身へとループしている。合成の単位であるということは、A で始まるか A で終わるかのいずれかの矢印とそれぞれ合成すると、同じ矢印が返ってくることを意味する。オブジェクト A に対する単位矢印を idA（identity on A）と呼ぶ。数学の表記法では、f が A から B に向かうとき

```
f∘idA = f
```

と

```
idB∘f = f
```

関数を扱う場合、identity arrow は引数を返すだけの identity 関数として実装される。この実装はどの型でも同じであり、つまりこの関数は普遍的な多相性を持つ。C++ではテンプレートとして定義することができる。

```
template<class T> T id(T x) { return x; }
```

もちろん、C++では、何を渡すかだけでなく、どのように渡すか（つまり、値で、参照で、const 参照で、移動で、など）も考慮しなければならないから、そんなに単純なものはない。

Haskell では、identity 関数は標準ライブラリ（Prelude と呼ばれる）の一部である。以下はその宣言と定義である。

```
id :: a -> a
id x = x
```

見ての通り、Haskell の多相関数は楽勝です。宣言の中で、型を型変数に置き換えるだけです。具体的な型の名前は常に大文字で始まり、型変数の名前は小文字で始まるのがミソです。つまり、ここでは a がすべての型を表している。

Haskell の関数定義は、関数名の後に正式なパラメータ（ここでは 1 つだけ、x）が続きます。この簡潔さは初心者に衝撃を与えますが、すぐに完璧な意味を持つことがわかるでしょう。関数の定義と呼び出しは、関数型プログラミングの基本であり、その構文は必要最低限に抑えられています。引数リストを括弧で囲まないだけでなく、引数の間にカンマもありません（後で複数の引数を持つ関数を定義するときにわかります）。

関数の本体は常に式であり、関数の中に文はありません。関数の結果はこの式で、ここでは単に x です。

これで Haskell の 2 回目のレッスンは終わりです。

恒等式は（これも擬似ハスケルで）次のように書ける。

```
f . id == f
id . f == f
```

あなたは自分に問いかけているかもしれません。なぜ、何もしない関数である恒等式をわざわざ使うのだろうか？では、なぜわざわざゼロという数字を使うのだろうか？ゼロは「無」を表す記号です。古代ローマ人はゼロのない数体系を持ち、優れた道路や水道橋を建設することができたが、そのいくつかは今日まで残っている。

ゼロやイドのような中立的な値は、記号変数を扱うときに非常に便利です。だからローマ人は代数学が苦手だったが、ゼロの概念に親しんだアラブ人やペルシャ人は得意だった。だから、恒等式関数は高階関数の引数として、あるいは高階関数からの戻り値として、非常に便利になる。高階の関数は、関数の記号的操作を可能にするものである。つまり、関数の代数である。

まとめると、以下のようになる。カテゴリはオブジェクトと矢印（モルヒズム）から構成される。矢印は合成可能であり、合成は連想的である。すべてのオブジェクトは、合成時の単位となる ID 矢印を持つ。

Composition is the Essence of Programming

関数型プログラマは、問題に取り組む際に独特の方法をとります。まず、禅問答のような問いかけから始めます。例えば、インタラクティブなプログラムを設計するとき、「インタラクションとは何か？コンウェイの「人生ゲーム」を実装するときは、人生の意味について考えるだろう。そこで、「プログラミングとは何か？最も基本的なレベルでは、プログラミングとは、コンピュータに何をすべきかを指示することである。例えば、「メモリアドレス x の内容をレジスタ EAX の内容に足せ」といったことだ。しかし、アセンブリでプログラミングする場合でも、私たちがコンピュータに与える命令は、より意味のあるものの表現なのです。私たちは、自明でない問題を解決しているのだ（自明であれば、コンピュータの助けを借りる必要はない）。では、どのようにして問題を解決するのだろうか。大きな問題を小さな問題に分解するのです。小さな問題がまだ大きければ、さらに小さな問題に分解し、それを繰り返す。そして最後に、小さな問題をすべて解決するコードを書きます。そして、そのコードの断片を組み合わせて、より大きな問題の解決策を生み出すというのが、プログラミングの本質なのです。分解したコードを元に戻すことができなければ、分解は意味を成しません。
この階層的な分解と再分解のプロセスは、コンピュータに押し付けられたものではありません。人間の心の限界を表しているのです。私たちの脳は、一度に扱える概念の数が少ないのです。心理学で最も引用された論文の一つ「The Magical Number Seven, Plus or Minus Two」では、人間は 7±2 個の情報の「かたまり」しか頭の中に留めておくことができないと仮定している。人間の短期記憶に関する理解の詳細は変化しているかもしれませんが、それが限定的であることは確かです。要するに、私たちは、オブジェクトのスープやコードのスパゲッティを扱うことができないのです。構造化が必要なのは、構造化されたプログラムは見ていて気持ちがいいからではなく、そうでなければ脳が効率的に処理できないからです。よく、あるコードを「エレガント」「美しい」と表現しますが、その真意は「人間の限られた頭脳で処理しやすい」ということです。エレガントなコードは、人間の消化器官が処理しやすいように、ちょうど良いサイズと個数のチャンク（塊）を作ります。

では、プログラムを構成するのに適したチャンクとは何だろうか？それは、表面積が体積よりゆっくり大きくなることです。(幾何学的な物体の表面積は、その大きさの 2 乗で大きくなり、体積はその大きさの 3 乗で大きくなるという直感から、私はこの例えが好きです)。表面積は、塊を構成するために必要な情報です。表面積はチャンクを構成するために必要な情報であり、体積はチャンクを実装するために必要な情報です。つまり、あるチャンクが実装されれば、その実装の詳細は忘れて、他のチャンクとどのように相互作用するかに集中することができる、という考え方です。オブジェクト指向プログラミングでは、表面はオブジェクトのクラス宣言やその抽象的なインターフェイスです。関数型プログラミングでは、関数の宣言がそれにあたります。(少し簡略化しているが、要はそういうことだ)。
カテゴリー理論が極端なのは、オブジェクトの内部を見ることを積極的に戒めるという意味である。カテゴリー理論における対象とは、抽象的な漠然とした存在である。そのオブジェクトについて知ることができるのは、それが他のオブジェクトとどのように関係しているか、つまり、矢印を使ってどのように接続しているかということだけである。これは、インターネット検索エンジンが、流入リンクと流出リンクを分析することによって、Web サイトをランク付けする方法です（不正をした場合を除きます）。オブジェクト指向プログラミングでは、理想化されたオブジェクトは、抽象的なインターフェース（純粋な表面、ボリュームなし）を通してのみ見ることができ、メソッドが矢印の役割を果たす。オブジェクトを他のオブジェクトとどのように組み合わせるかを理解するために、オブジェクトの実装を掘り下げる必要がある時点で、そのプログラミングパラダイムの利点が失われている。