目次

第 1 章

登場人物

1.1　矢澤にこ

各メンバーがそれぞれのプログラミング言語を極める集団μ’ｓのメンバーで、生粋の Pure Functional Programmer。

Haskell のついでに諸数学を勉強して、楽しんでいる。

Haskell と数学回りに触れている間はポンコツが直る。ちょっとは先輩っぽいかしら？

1.2　西木野真姫

平成のにこキチ。にこちゃんの色んなところをチラ見しては、素知らぬフリをしている。

Haskell 初心者。

第 2 章

序章

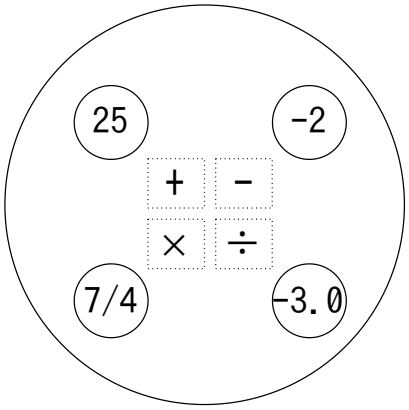

図 2.1: 代数

ｶﾞﾁｬﾝ

にこ「待たせたわね、真姫」

真姫「別に待ってないわ、にこちゃん」

閑散としていた部室に、にこちゃんが現れた。今日は私が、ある目的のためににこちゃんを呼び出したの。

にこ「今日はにこに、代数について教えて欲しいんだったわね。活動がお休みの日にまで、あんたも勤勉ねぇ」

真姫「最近はプログラミングでも代数について聞く機会も増えたし、にこちゃんなら

代数について知ってると思ってね。お願いできるかしら」

私の目的、それは……

にこ「もちろん知ってるわよ。任せてちょうだい！」

真姫「……」

共通の話題で盛り上がって、にこちゃんともっと仲良くなること！

2.1 代数とは何か

真姫「私から『代数教えて』って言ったのに聞くのもなんだけど、そもそも代数ってどんなものなの？」

にこ「いい質問ね」

にこ「代数っていうのはある種の規格よ」

にこ「例えば『半群はこのようなことができる』『モノイドはこのようなものを持つ』みたいに、『こういう構造はこういう性質を持つ』というのを調べる学問が代数なの」

真姫「半群？　モノイド？」

にこ「半群とモノイドっていうのがまさに、代数と呼ばれるもののうち一つよ」

にこ「代数にはおおまかな階層として、この 5 つが存在するわ」

1. 半群
2. モノイド
3. 群
4. 環
5. 体

真姫「へえ。代数っていくつもあるのね」

にこ「そうよ。『自然数の集合はモノイドである』みたいに、ある構造がその規格をどのように満たすか、なんてことも考えたりするわ」

にこ「これはちょうど、Haskell の `class` と `instance` の関係と同じ。モノイドが `class`、自然数の集合がその `instance` ね」

にこ「真姫はこの前 Haskell に入門してたわよね。この例えはわかるかしら」

真姫「Haskell はちょうど入門し終えたくらいだけど、わかるわ」

にこ「よかった。じゃあ今日は代数という構造がどのように何を表すのかを、ちょっとした Haskell を使って見ていきましょうか」

真姫「ええ、よろしく♪」

2.2 解説の流れ

にこ「今日はこの順に話をしていければと思うわ。また適宜、関連する概念について話させてちょうだい」

1. 半群
2. モノイド
3. 群
4. 環
5. 体

真姫「さっき教えてくれた、代数の階層ね」

にこ「そうよ。代数はこの上から下にかけて、より概念が強化されていくの」

にこ「例えば半群は簡単に言うと『掛け算ができる』っていう構造なんだけど、その下のモノイドは『掛け算ができて「単位元」というものが存在する』というものになるわ」

にこ「あ、掛け算や単位元が何のことなのかはまだおいておいてね」

真姫「ふうん。順を追っていくってことね。わかったわ」

にこ「余談だけど、プログラミングで主に応用されるのは半群・モノイドあたりね。Haskell の標準ライブラリにもあるの」

真姫「群・環・体はそんなに重要じゃないの？」

にこ「いいえ、そういうわけじゃないわ。群以上のものは性質がより特殊で、プログラミングだとデータがそれを満たしにくいってだけ。群以上は暗号方式に使用されたり、プログラミング外で重宝されてるわ」[*1]

真姫「なんだかけっこう重要なところで使われてそうね」

にこ「そうなの。にこの知り合いの数学家に『モノイド？　何それ。群なら知ってる』っていう人がいたくらいよ。びっくりしたわ」

にこ「ってとこで、やっていきましょうか。書いていくコードはコンパイル可能・実行可能な形式で、この git リポジトリ[*2]に置いていきましょう」

真姫（ワクワク）

[*1] https://ja.wikipedia.org/wiki/ElGamal 暗号

[*2] https://github.com/aiya000/algebra-with-nico-code

第 3 章

用語、前提知識の説明

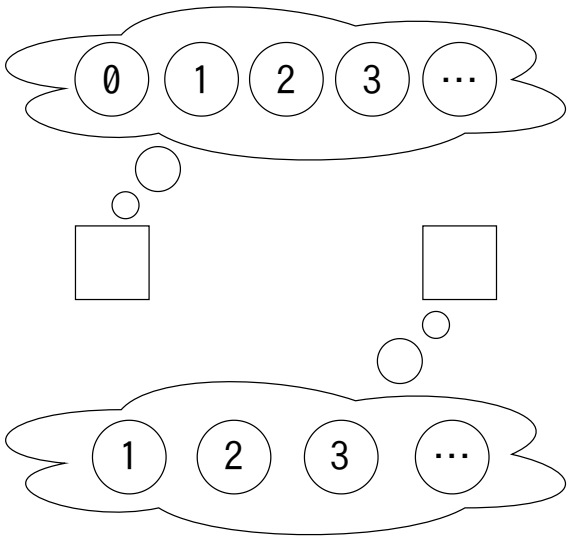

図 3.1: 認識の相違

にこ「さっそく代数の説明を……と言いたいところだけど、まずは知識合わせをしましょうか」

真姫「知識合わせ？」

にこ「予習しつつ、お互いの認識合わせをね」

にこ「これから代数について話すために、いくつか知っておいて欲しい概念があってね」

にこ「真姫はそれらを既に知ってるかもしれないけどもしかしたら、にこと認識の相違があるかもしれないじゃない。だからその解決も兼ねて、前提知識を改めて説明して

おきたいのよ」

真姫「へえ、そういうことならよろこんで。誤解があったら大変だものね」

にこ「って言っても大したことは言わないから、今は聞き流して、後で思い出してくれるくらいでもいいわ」

3.1 集合

にこ「まずは**集合**について。真姫は集合は知ってる？」

真姫「ものの集まりよね。{1, 2, 3}とか、{a, b, c}とか」

にこ「そうそう。そういう 1 と 2 と 3 とかの集まりを集合って言うのだったわね」

真姫「ええ」

にこ「また、その集合に S という名前をつけたときに」

集合 S

```
S = { 1, 2, 3 }
```

にこ「『1 は集合の**要素**である』。または『1 は集合の**元**（げん）である』と言うわ」

- 要素・元（げん）: ある集合の中の『もの』のこと

真姫「なるほど。2 や 3 も集合 S の元ってことね」

3.2 数の種類

にこ「次は**数**について」

にこ「数はいくつかの種類があるの。例えば**整数・有理数・実数**など」

にこ「これらも集合として、このように表すことができるわ」

数（集合）

```
整数   = { ..., -2, -1, 0, 1, 2, ... }
有理数 = { ..., -1/2, -3/2,
           ..., -2/1, -1/1, 0/1, 1/1, 2/1, ...,
           1/2, 3/2, ... }
実数   = { ..., -1.2, ..., -1.1,
           ..., -0.1, ..., 0.0, ..., 0.1, ...,
           1.1, ..., 1.2, ...,
           √2, ..., π , ... }
```

にこ「... って書いてあるのは『この間に 1 つ以上の元があるけど、省略するね』って意味よ」

真姫「例えば整数の後側の... には 3，4，5，6 そして 7 以上が入ってくるってことよね」

にこ「そうよ、さっすが真姫ね。理解が早くて助かるわ」

真姫「そ、それくらい当然よ！」

にこ「ちなみに自然数っていうのもあるわ」

自然数（集合）

```
自然数 = { 0, 1, 2, ... }
```

にこ「自然数はしばしば 1 以上の整数を表したりするけど、今回は 0 以上の整数よ」

■コラム: 実数の記法について

ここで実数についてこのように表したけど

```
実数 = { ..., -0.1, ..., 0.0, ..., 1.0, ... }
```

厳密に言うとこれは疑似表記なの。

なぜなら例えば 0.0，...，1.0 の間にはこのように無限個の実数があるけど、

```
0.0
0.01
0.011
0.0111…
0.1
```

一般的には何かに挟まれた... は『有限個の省略』を表すから。

でも今回は簡単のためにこう書かせてもらったわ。

3.2.1 有理数

真姫「そういえば、有理数ってなんだっけ」

にこ「有理数は整数と整数の比で表せる分数よ。例えば 1/2 ― 2 分の 1 ― や 1/3、25/252 が有理数ね」

にこ「ここで 1/2 の 1 を分子、2 を分母という」

にこ「有理数は単なる比だから、1/5 と 100/500 のように同一の比になるものは同じ元として扱われるわ。1/3 と 3/9 もそうね」

にこ「また 2/π のように分子または分母が整数でないもの、3/0 みたいに分母が 0 のものは有理数ではないわ」

にこ「間違いやすいものとしては……1.2/1.3 や 1/0.25 は 12/13 と 4/1 だから有理数だし、0/3 や 0/1000 のように分子ではなくて分母が 0 でも有理数よ」

真姫「x/y の x や y が整数でなかったとしても、安易に『有理数ではない』と言えないのね。ちょっと間違いやすそう」

にこ「本当にね」ﾌﾌ

にこ「まとめるとこんな感じよ」

有理数の例

```
-1/2, -1/3
0/1 = 0/2 = 0/500
1/2, 1/3 = 3/9 = 5/15
7/5, 12/15
1.2/1.3, 1/0.25
```

有理数でない例

```
3/0, 1/0
2/π , 10/√2
```

真姫「ありがとう。思い出してきたわ」

3.3 型と集合の対応

にこ「ここまでいくつかの集合を扱ってきたけど、Haskell ではこれらの集合に対する型が定義されているの」

にこ「それらはこのように、先程説明した集合と対応するわ」

- 整数 <=> Integer
- 有理数 <=> Rational

にこ「あとは Bool がこんな対応をするわね」

- 2 つの元の集合{0,1} <=> Bool

にこ「このとき整数って言えば{..., -1, 0, 1, ...}の事だし、Integer って言っても同じくそれを指す。ってな具合ね」

真姫「ふ〜ん、言い分けることには何か意味があるの？」

にこ「どちらかというと**意味を一致させておく**ことが大事なの。例えば『文字列に変換できること』を表す、Haskell の Show クラスをこのように Integer が実装できたとして」

instance

```
instance Show Integer where
  -- ここで Show の実装
```

にこ「そのとき同時に『整数は文字列に変換できる』というように、現実の意味でも同様に述べることができる。っていうことね」

真姫「ああ、なるほど。Haskell で述べたものは現実でも述べられたものとする、ってことね」

にこ「その通り！」

真姫「もう一つ疑問があるんだけど、型って集合なの？」

にこ「そういうと語弊があるけど、値を元として見ると集合になるわ」

*1

真姫「そっか、それはそうよね」

*1 ちょっと詳しい人が混乱しないために、本題から逸れないレベルで注釈をしておくわ。**でもこれを理解しなくても進行上大丈夫よ！** さて、ここで型が集合に対応するとは……型と命題・ロジックと証明が一致しているので、コードもしくは集合の側のどちらで主張しても同じことが述べられるぞ……という程度の意味を指しているわ。例えば associativeLaw :: Semigroup a => a -> a -> a -> Bool は『任意の半群 a についてのある元 a が 3 つあるならば真偽が定まる』という命題の主張に、その妥当な定義 associativeLaw x y z = .. は『.. なので命題は正しい』という証明に対応するわ（カリー＝ハワード同型）。

3.4 任意の・存在する

にこ「認識合わせはこれで最後よ」

真姫「了解よ」

にこ「論理として『**任意**の』とか『**存在**する』っていう言い回しがあるの。例えばこんな感じ」

- 任意の整数の元は、**0** 以下か **0** より上である
- 有理数の元のうち、**-1** 以下でもなく **1** 以上でもないものが、存在する

にこ「ここで前者は『**全ての元が**その条件を満たす』という意味。後者は『**1 つ以上の元が**その条件を満たす』という意味よ」

真姫「ちょっと、難しいわね」

にこ「そうねえ。例えば Haskell でこんな型と、その型を持つなんらかの関数があったとして」

a に対する条件型とその関数

```
-- 条件を表す型
type Predicate a = a -> Bool

lessEqualOrGreater :: Predicate Integer
lessEqualOrGreater x = (x <= 0) || (0 < x)

isZero :: Predicate Integer
isZero 0 = True
isZero _ = False
```

にこ「その関数に型 a の全ての値が True を返さなければいけないのが『任意の』」

任意の Integer の元が lessEqualOrGreater を満たすことの示し

```
-- 以下の全てが "True" を表示すれば、示しが完了
main :: IO ()
main = do
  print $ lessEqualOrGreater ...
  print $ lessEqualOrGreater (-2)
  print $ lessEqualOrGreater (-1)
  print $ lessEqualOrGreater 0
  print $ lessEqualOrGreater 1
  print $ lessEqualOrGreater 2
```

```
  print $ lessEqualOrGreater ...
-- ... は疑似表記
```

にこ「それを満たすものを 1 つでも示せればいいのが『存在する』よ」

isZero を満たす Integer の元が存在することの示し

```
-- 以下が "True" を表示すれば、示しが完了
main :: IO ()
main = print $ isZero 0
```

真姫「なるほど。現実でも『任意の整数の元は、**0** 以下か **0** より上である』という Predicate に対して同じことを示せばいい。存在についても同じことを……。ってことね」

にこ「その通りよ♪」

真姫「ありがとう、わかりやすかったわ」

第 4 章

半群

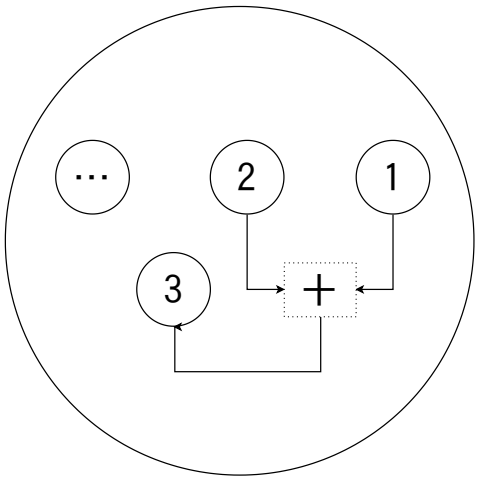

図 4.1: 集合 Integer「の上の」「二項演算」+

にこ「それじゃあ代数、始めていきましょうか」
真姫「よろしく頼むわ」
にこ「ええよろしく。まず始まりの代数、**半群**についてよ」
にこ「いきなりだけど真姫、足し算についてどう思う？」
真姫「どう？　どうって……うーん、どうとも思わないけど
にこ「じゃあ掛け算については？」
真姫「小学生で習う、わよね」
にこ「そう、小学生で習う範囲なの」

にこ「半群はその足し算や掛け算を、抽象的に定義した代数よ」
にこ「もっと言うと基本的な代数って、**四則演算を抽象化**したようなものなの」
真姫「そんなに簡単なものなの？」
にこ「油断しすぎるのはよくないけど、でもその通りよ」
にこ「代数の基礎自体はそんなに複雑なものじゃないの、抽象的すぎるだけ。大事なのは深く考えすぎないことよ」

4.1　半群 = 足し合わせることができる構造

4.1.1　解説

にこ「半群は言葉で表すと、こういうものよ」[*1][*2]

> 半群（はんぐん、英: semigroup）は集合 S とその上の結合的二項演算とをあわせて考えた代数的構造である。

にこ「ここは落ち着いて、しっかり要点をまとめましょう。ここの要点はこれよ」

- 集合 S（台集合 S）
- その（集合 S の）上の二項演算
- 結合的

真姫「集合はわかるけど、台・二項演算・結合的？」
にこ「ええ、それが半群の持つ性質よ。少しずつ解読していきましょ」
にこ「まずはこれらを Haskell で書いてみるわね……っと。半群はこんな風に `class` で定義できるわ」ｶﾀｶﾀｶﾀ

```
class Semigroup a where
  (<>) :: a -> a -> a
```

にこ「Haskell の `class` がそうであるように、半群はいくつかの約束事を定めたものなの」

[*1] https://ja.wikipedia.org/wiki/半群
[*2] 『上の』は普通に『うえの』と読むわ

にこ「というか代数自体がそういうものなのよ」
真姫「ある種の抽象ってことね」
にこ「そうよ。さすが、真姫は難しい言葉を知っているわね」
真姫「ちょっと、あんまりなめないでよね」ﾃﾞﾚﾂﾝﾂﾝｯ
にこ「この Haskell のコードをさっきの要点に照らすと、こう読めるの」

- 集合 a（台集合 a）
- その（集合 a の）上の二項演算<>

にこ「そしてもう一つの要点、**結合的**とは、その二項演算<>がある法則を満たすことの言い回しよ」[*3]

```
associativeLaw :: (Semigroup a, Eq a) => a -> a -> a -> Bool
associativeLaw x y z =
  (x <> y) <> z == x <> (y <> z)
```

にこ「これは**結合法則**と呼ばれるわ」
にこ「a の任意の値 x，y，z に対して、x と y を先に合わせても、y と z を先に合わせても結果が変わらない。というものね」
真姫「a の**上の**？　二項演算は 2 つの引数を受け取る演算のことよね。結合法則って何の意味があるの？」
にこ「そこらについてまとめる必要があるわね。しっかりと、まとめましょうか」
……

4.1.2　二項演算

簡単なことだけど一応、**二項演算**について明確にしておきましょう。
二項演算は

- 何らかの x と何らかの y という **2 つの値を受け取って**
- 何らかの値 z を返す

という関数のことよ。
2 つの値、つまり二項を受け取る演算だから**二項**演算にこ♡　にこ（二項）だけにね！
+ は二項を受け取る演算だから立派な二項演算よ。

[*3] ここで Eq a が要請されるけどテストの都合のものよ。半群自体に要請される性質ではないことに注意してね。

でも succ は一項だけしか受け取らないから二項演算ではないわ。

二項演算

```
(+) :: Integer -> Integer -> Integer
(<>) :: a -> a -> a
```

二項演算ではない

```
succ :: Integer -> Integer
```

4.1.3　上の（うえの）

集合 a の**上の**という表現について。
これは二項演算が

- a の元だけしか受け取らない
- a の元を返す

という性質のことを指すわ。
+ と succ は Integer だけ受け取るから、集合 Integer の上の演算ね。
<>も『集合 a の上の演算』よ。
でもこんな (:) は

cons

```
(:) :: a -> [a] -> [a]
```

a だけじゃなくて [a] も受け取ってるから、集合 a の上の演算ではないわ。もちろん集合 [a] の上でもない。
ちなみに『の上の』という表現は『**に閉じている**』とも言い回されるわ。
例えば以下は同じ意味よ。

- + は Integer の上の演算
- + は Integer について閉じている

どっちかっていうと『閉じている』の方がイメージ付きやすいかしら？

集合 Integer の上の演算

```
succ :: Integer -> Integer
(+) :: Integer -> Integer -> Integer
```

いずれの集合にも閉じていない

```
(:) :: a -> [a] -> a
flip :: (a -> b -> c) -> b -> a -> c
```

4.1.4 結合法則

結合法則の意味について。

例えば 1 <> 2 <> 3 <> 4 という式について議論したいとき、その解釈、組み合わせは以下の通りになるわよね。

組み合わせ網羅

```
((1 <> 2) <> 3) <> 4
(1 <> (2 <> 3)) <> 4
(1 <> 2) <> (3 <> 4)
1 <> ((2 <> 3) <> 4)
1 <> (2 <> (3 <> 4))
```

ということはもしこれについて結合法則を仮定しないならにこたちは 1 <> 2 <> 3 <> 4 について、5 通りの検証を重ねないといけないの。だってその結果のどれもが別の結果になるかもしれないからね。

各パターンの結果は必ずしも一致しない

```
↓とは限らない

((1 <> 2) <> 3) <> 4
  = (1 <> (2 <> 3)) <> 4
  = (1 <> 2) <> (3 <> 4)
  = 1 <> ((2 <> 3) <> 4)
  = 1 <> (2 <> (3 <> 4))
```

1〜100 全てを含んだ式を考えるなら、パターンは 99 通りもあるわ。

このとき結合法則を満たすと仮定したならば、いずれのパターンも 1 <> 2 <> 3 <> 4 という 1 通りのものとして考えられるわ。

個別に考える必要がなくなるの。

組み合わせ（結合法則を仮定した場合）

```
1 <> 2 <> 3 <> 4
  = ((1 <> 2) <> 3) <> 4
  = (1 <> (2 <> 3)) <> 4
  = (1 <> 2) <> (3 <> 4)
  = 1 <> ((2 <> 3) <> 4)
  = 1 <> (2 <> (3 <> 4))
```

とっても合理的になるわね！

4.2 半群の具体例

4.2.1 具体の概要

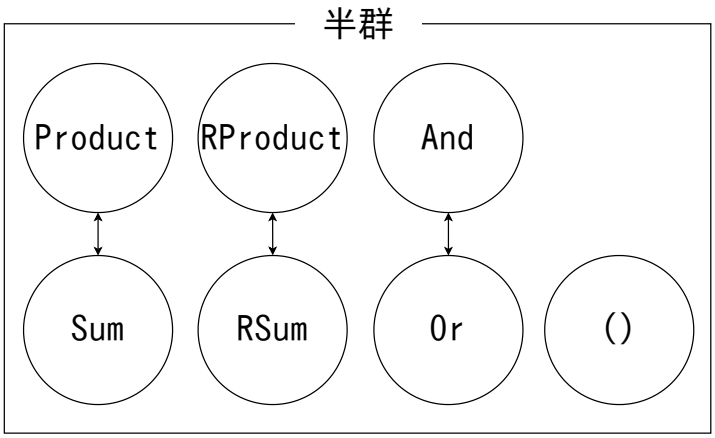

図 4.2: 半群インスタンス

真姫「おかげ様で『二項演算』と『上の』っていうのとはわかったわ、ありがとう」

真姫「でも……いまいちまだ半群っていうのがなんなのかわからないわ。なんというか、感覚に落ちてこないっていうか」

にこ「その『感覚に落ちてこない』っていうのはきっと、真姫の直感が半群を捉えるために成長しようとしている……ってことの感覚かも」

にこ「こういう抽象的な話でそういう感覚に陥るときは、具体例を見てみるのが大事ね」

にこ「`Semigroup` の `instance` を定義してみましょう」

にこ「まずは `Integer` とその足し算 `+` よ」

Integer インスタンス

```
instance Semigroup Integer where
  (<>) = (+)
```

にこ「例えば Integer の元 10，20，30 を取ったならば、このように結合法則を満たすわ」

Integer と + の結合法則

```
(10 + 20) + 30 = 10 + (20 + 30)
```

真姫「なんだかすごく当たり前のことしか言ってないように思うのだけど……」

にこ「それでいいのよ。半群は当たり前のことを改めて言ってるだけだから」

にこ「次に Bool と&&のインスタンスよ」

Bool インスタンス

```
instance Semigroup Bool where
  (<>) = (&&)
```

にこ「この Bool インスタンスは、例えば元 True，False，True を取ったときに結合法則をどう満たすと思う？」

真姫「うーん、こうかしら」ｶｷｶｷ

Bool と&&の結合法則

```
True && (False && True) = (True && False) && True
```

にこ「さすが、その通りよ！」

真姫「ベッ、ベツニアタリマエダシ……」

真姫「逆に結合法則を満たさないものなんてあるの？」

にこ「ええ、ちゃんとあるわよ。例えば引き算は結合法則を満たさないわ」

```
(30 - 20) - 50 = -40
30 - (20 - 50) = 60
```

にこ「割り算もそうね」

```
(2 / 4) / 8 = 0.0625
2 / (4 / 8) = 4.0
```

真姫「本当だ。当たり前に満たされるもの、ってわけじゃなかったのね」

4.2.2 実用例

にこ「理解の助けのために、プログラミングでの応用例を書いてみましょうか」

にこ「半群を使えば、例えばこんな concat 関数が定義できるわ」

整数の総和を求める関数

```
concat :: Semigroup a => a -> [a] -> a
concat = foldl (<>)

resultSum :: Integer
resultSum = concat 0 [1..100]

resultAll :: Bool
resultAll = concat True [True, True, True]
```

にこ「この concat 関数は、その型の性質に応じた合わせを求めるものよ」

にこ「例えば Integer と足し算 + なら『総和』を、Bool と&&なら『渡されたリストの全てが True であるか否か』を。といった感じね」

resultSum

```
5050
```

resultAll

```
True
```

真姫「ふうん、これって便利なの？」

にこ「もちろんよ。ほら、コーディングするときにその場その場で総和を求める関数

とかとかを書くことも考えられるけど、そのコストはバカにならないじゃない。コードが読みにくくもなる」

にこ「でもこれを使えば、いちいちそれらを書く必要がなくなるわ」

にこ「DRY のための基本的な技術よ」[*4]

真姫「抽象的なものを定義しておいて、幅広く使用させる。関数型プログラミングらしいアプローチね」

にこ「ええ！　よく『関数型プログラミングにおいて再帰を直に書く必要はなく、畳込みを用いることができる』っていう言論があるけれど、Semigroup を使えば畳み込みすら直に扱わなくていいってことよ」

にこ「書かなくていいものは無駄に書かない、というのが一番正しいと思うわ」

4.2.3 半群と台集合

にこ「まだ**台集合**について話してなかったわよね」

真姫「ああ、そういえばそうね」

にこ「台集合については、単なる用語よ」

にこ「まず半群の言い回しとして、Integer と + のインスタンスをとって『半群 (Integer, +)』) と言ったりするの」

真姫「半群 (Bool, &&) とか？」

にこ「そうそう。その二項演算が話の流れで明らかなときは半群 Integer や半群 Bool と省略したりもするわ」

にこ「そしてこの半群のベースとなる集合 Integer を、半群 (Integer, +) の台集合 Integer と言うの」

真姫「半群 (Bool, &&) の台集合は、台集合 Bool？」

にこ「Excellent!」

4.2.4 台集合に対し半群は必ずしも 1 つじゃない

にこ「ところで実は、集合 Integer は半群 (Integer, *) にもなるの。Bool は半群 (Bool, ||) にもなるわ」

にこ「つまり 1 つの台集合に対して、半群は 1 つとは限らない」

真姫「確かに*は Integer に閉じていて、結合法則も満たしそう。||もそうね」

にこ「だからせっかくだし newtype を使って、それを表現してみましょう。アイドルはしっかりが大事！」

newtype

[*4] DRY 原則: Don't Repeat Yourself（同じ内容のコードを各所で書き散らすな）

```
newtype Sum = Sum
  { unSum :: Integer
  } deriving (Show, Eq, Num, Enum)

newtype Product = Product
  { unProduct :: Integer
  } deriving (Show, Eq, Num, Enum)

newtype RSum = RSum
  { unRSum :: Rational
  } deriving (Show, Eq, Num, Enum)

newtype RProduct = RProduct
  { unRProduct :: Rational
  } deriving (Show, Eq, Num, Enum)

newtype And = And
  { unAnd :: Bool
  } deriving (Show, Eq)

newtype Or = Or
  { unOr :: Bool
  } deriving (Show, Eq)

xor :: Bool -> Bool -> Bool
xor True  True  = False
xor True  False = True
xor False True  = True
xor False False = False

newtype Xor = Xor
  { unXor :: Bool
  } deriving (Show, Eq)
```

instance

```
instance Semigroup Sum where
  (<>) = (+)

instance Semigroup Product where
  (<>) = (*)

instance Semigroup And where
  And x <> And y = And $ x && y

instance Semigroup Or where
  Or x <> Or y = Or $ x || y
```

```
instance Semigroup Xor where
  Xor x <> Xor y = Xor $ x `xor` y
```

にこ「これは例えばこんな感じで使うわ」

にこ「同じ Integer の 10 <> 20 でも、足し算 Sum と掛け算 Product で結果が違うのがわかるわね」

Sum と Product の例

```
aSumInt :: Sum
aSumInt = Sum 10 <> Sum 20

aProductInt :: Product
aProductInt = Product 10 <> Product 20
```

aSumInt

```
Sum {unSum = 30}
```

aProductInt

```
Product {unProduct = 200}
```

真姫「型の名前によってインスタンスの形、というか二項演算の種類が……変わるってことね」

4.2.5 もっと半群の具体

にこ「例示をレイジーに怠っていたら理解は難しいってことで、他のインスタンスになりそうなものも列挙してみましょう」

もっとインスタンス

```
instance Semigroup RSum where
  (<>) = (+)
```

```
instance Semigroup RProduct where
  (<>) = (*)

instance Semigroup [a] where
  (<>) = (++)

instance Semigroup () where
  () <> () = ()
```

真姫「にこちゃんのダジャレセンスにはつっこんでいいのかしら？」

にこ「そして最後の確認。全てのインスタンスに対して associativeLaw テストを実行してみましょう」

各インスタンスが法則を満たすことの確認

```
main :: IO ()
main = do
  smallCheck 2 $ associativeLaw @Sum
  smallCheck 2 $ associativeLaw @Product
  smallCheck 2 $ associativeLaw @RSum
  smallCheck 2 $ associativeLaw @RProduct
  smallCheck 2 $ associativeLaw @[Double]
  smallCheck 2 $ associativeLaw @And
  smallCheck 2 $ associativeLaw @Or
  smallCheck 2 $ associativeLaw @Xor
  smallCheck 2 $ associativeLaw @()
```

にこ「smallCheck っていうのがテストを実行してくれる関数で、2 はテストの深度、つまりどれくらいテストを行うかよ。ここらはあんまり気にしなくてもいいわ」

出力

```
Completed 125 tests without failure.
Completed 125 tests without failure.
Completed 27 tests without failure.
Completed 27 tests without failure.
Completed 8 tests without failure.
Completed 8 tests without failure.
Completed 8 tests without failure.
Completed 8 tests without failure.
Completed 1 tests without failure.
```

にこ「『without failure』つまり『失敗なし』。オッケーね！」

真姫「Double はインスタンスにはならないの？」

にこ「ええ。例えばこんな感じの式があったときに、丸め誤差というのが起きて結合法則を満たさない場合があるの」

丸め誤差が起きるような式

```
({すごく大きい数} + {けっこう小さい数}) + {けっこう小さい数} ≠
{すごく大きい数} + ({けっこう小さい数} + {けっこう小さい数})
```

真姫「ああ。そういえばコンピューターの浮動小数点数は、実数とは違うものだったわね」[*5]

4.2.6 補足: TypeApplications

真姫「@Foo みたいなのは……as パターン？」

にこ「それは TypeApplications っていう、コンパイラ独自の Haskell の拡張よ。和訳すると『型の関数適用』なんだけど」

にこ「例えば今回の associativeLaw @Sum は『associativeLaw の型パラメータ a は Sum だよ』っていう注釈なの。associativeLaw はこのような型を持っていたわね」

```
associativeLaw :: (Semigroup a, Eq a) => a -> a -> a -> Bool
```

にこ「それに対して TypeApplications による a = Sum という指定がされたから、associativeLaw :: Sum -> Sum -> Sum -> Bool になるわ」

真姫「へえー、便利そうだけど……単純に Sum -> Sum -> Sum -> Bool を型指定するんじゃだめなの？」

にこ「それでもいいんだけど、TypeApplications を使った方が可読性がいいからね」

真姫「@Sum だけで済む……なるほど。可読性は正義だものね」

4.3 まとめ

- 半群 (a, <>) とは以下の様なものである
  - 二項演算（<>）を持ち

[*5] 実数と浮動小数点数が別物であることの例として、式 10 /3 が存在するわ。実数はそれを 3.333… として表すけど、浮動小数点数では結果が 3.3333333333333335 などの有限な桁の値になるのよ。

- a 上で閉じていて
- 結合結合法則を満たす

```
class Semigroup a where
  (<>) :: a -> a -> a

associativeLaw :: (Semigroup a, Eq a) => a -> a -> a -> Bool
associativeLaw x y z =
  (x <> y) <> z == x <> (y <> z)
```

にこ「以上が半群よ。最初だったからちょっと量が多くなっちゃったわね、どうだったかしら」

真姫「これくらい、私にかかれば余裕よ」

にこ「それにしては時々、目がキョロキョロしてたけど？」ﾌﾌ

真姫「バッ！！　ちが、それは」

真姫（に、にこちゃんの細くてきれいな二の腕をチラ見してただけ……）

参考文献

- Wikipedia（各章中に明記）
- 『d.y.d. 計算のきまり』: http://www.kmonos.net/wlog/121.html
- 『浮動小数点数では結合則が成り立たない - fortran66 のブログ』http://fortran66.hatenablog.com/entry/2017/05/20/001554

Special Thanks

- blacky さん (@blac_k_ey)
- kotaro さん (@310_ktr)
- よ子 さん (@hoxot)
- yassu さん (@yassu0320)
- ゆ さん (@yuche13)
- dif_engine さん (@dif_engine)
- MYAO さん (@myao_s_moking)
- bra_ca_ket さん (@bra_ca_ket)
- tadsan さん (@tadsan)
- Leo さん (@reotasoda)
- ノエルさん (@nf_shirosawa)
- Twitter で相談に乗ってくださった方々
- いつも色んな相談にのってくれるコノコさん
- いつも助けてくれる愛犬ペロくん
- いつも助けてくれる愛犬ララちゃん

索引

にこ先輩といっしょに代数！

印刷: しまや出版 (http://www.shimaya.net)

発行: 大銀河宇宙 No.1_Haskeller (https://aiya000.github.io)
「君こそが No.1 Haskeller だ！」

著者: aiya000 (Twitter: @public_ai000ya, GitHub: @aiya000)

連絡先: aiya000.develop@gmail.com

2018 年 10 月 8 日　初版

購入者限定ダウンロード用 URL: https://aiya000.booth.pm/items/1030859
ダウンロードパスワード: No1Haskeller-is-YOU