せつラボ ～圏論の基本～

原作・本文
aiya000 (@public_ai000ya)

イラスト・ヴィジュアルデザイン
碧はっさく (@HassakuTb)

しまや出版 発行

○　○　○

おはよう。まだ夜は明けてないけれど。

8年前……夢を見た。
雲の上。辺りには水色いっぱいの空。どこか完成されたような、全く風のない……空気。

それで、君は何がお望みなの？

望みなんて、あまり考えたことがなかった。
僕は……

○　○　○

目次

第1章　登場人物

1.1　η（えーた）

『……ずっと一緒にいてくれる人が、欲しいかな。』

ちびっこ気だるげ・天才ヘタレガール。
右利き。
μを大切に思うが故にμに遠慮する癖、
一息ついたとき・困ったときなどにμの方を見る癖がある。
家の縁側に庭用の靴を放置しておいたらカマキリが入っていて、
履いたときに足でつぶしてしまったのがトラウマ。

1.2　μ（みゅー）

『よーしよし、よーしよし♡　今日もよく、がんばりました♪』

やさしさ・ふわふわ・女の子。
左利き。
ηのことが大好きで、常に付き従って、世話をやいている。
人当たりがよく温和で皆に好かれる。
しかしその一方で自分の意見を主張するのが苦手。
笑ったときやちょっと困ったときなど、手を丸めて口の前に添える癖がある。
ある不思議な出来事から生まれた女の子。

第2章　（前書き）

このたびは本書をお手にとってくださり、ありがとうございます！

「最初の指南」

本書含む、数学書の前書きが「まどろっこしい！」と思ったなら、チラ見で済ませるか、飛ばしてもいいと思います。数学へのモチベーションを大事に。前書きは後で！

この本は、**圏論**という**極楽浄土**を広めるために書かれました。「数学たのしそうなんだけど、難しくて……」という人々をターゲットにしています。

圏論に熱中し、ゾーンに入ったときの――あの**「宇宙にいるような感覚」**。本書が、その導きの第一歩となれれば、嬉しいことこの上ありません。

ターゲットは**数学の未入門者**です。圏論をより厳密に理解したい方々には、物足りないかもしれません。

ですので本書を読んだ後にもし圏論に興味を持っていただけたならば、ぜひ別の専門書での「圏論」も見ていただければと思います。

圏論勉強会 @ ワークスアプリケーションズ

- *http://nineties.github.io/category-seminar*

そこそこマイルドな入門資料で、深い内容まで解説してくださっています。

本書を読んだ直後に読むなら、丁度いいはずです！

Web上の資料なので、試しに読んでみては、どうでしょうか。（本書を読んだ後にね！）

圏論の歩き方｜日本評論社

- *https://www.nippyo.co.jp/shop/book/6936.html*

圏論自体を知りたい人よりは、「圏論って何に使われてるの？　どんな使い方するの？」を、最も手っ取り早く知りたい方におすすめです。

圏論の入門書ではなく、応用事例のまとめが多い気がします。

圏論の基礎 - 丸善出版 理工・医学・人文社会科学の専門書出版社

- *https://www.maruzen-publishing.co.jp/item/b294317.html*

数学入門者向けではないかもしれません。数学に精通した人なら、入門書として最適だと思います。

密度がかなり高く、学習難易度は高いと思いますが、その分のリターンは期待できそうです。

2.1 付録・Web正誤表

本書は、読者の頭のメモリオーバーフローを避けるために、いくつかの証明が省略されています。それらは以下のURLで見ることができます。

- 付録: *https://github.com/aiya000/setulabo-basic-category-proofs*

その他。

- Web正誤表: *http://aiya000.github.io/posts/2019-03-16-setulabo-errata.html*
- Twitter等のハッシュタグ: #せつラボ

◆ ◆ ◆

第3章　始まり

静寂な朝。白基調のログハウス、高さ5メートル、風がよく通った部屋。青くて白い、春の空。

お風呂上がったよ〜。

彼女が歩いてくる。

歩きながら揺れる、リボンでくくったふわふわの髪を、目が追う。

……。

肩や首にかかる、なだらかな曲線からする、フローラルな香り。μと一緒に暮らし始めてから何年も経つっていうのに、いつまでも慣れはしない。

ドキドキしちゃうから、少しクールぶって言葉を返す。

おかえり、μ。

ただいまη♪

3.1　せつめいするラボ

圏論？

うん。試しに一緒にやってみない？

時が経つのは早いもので、もうμも8歳になった。だからそろそろ、専門的な分野を始めてみてもいいんじゃないかと思ってね。*1

そこで圏論。数学の一分野なんだ。

へえー。勉強はじめにいい感じなのかな。

そうなんだよ、圏論は勉強はじめにいいんだ！

……なんて、ごめん。本当はなんでもいいんだ。何かを一緒にやってみたいなって、思って。

……。

圏論じゃなくてもよかったんだ。僕が圏論をやりたいだけ。μって、ものの考え方はしっかりしてるし、圏論から始めてもわかると思う。

えへへ。ηが教えてくれたから、中学校くらいの数学はわかるよ。

うん、やってみたいな。……ηがわたしに何か誘ってくれるのってあんまりもん。

あはは、よかった。

……

[*1] 8歳。英語で言うと「eight years old」

圏論の基本はシンプルなものだよ。その応用はかなり幅広くて、奥に進むには多くの数学知識が必要。だけど入り口に入るだけなら、とっても簡素。

うんうん。

とはいえ少しの事前知識は必要だから、まずはそこから始めていこう。そうしたらその後に、圏論の基本まで進んでいこう。

ふふ、なんだか楽しみだよ！　よろしくお願いします。

3.2　圏論とは何か

◆ ◆ ◆

始める前に圏論が何なのかについて、話してもいいかな。

うん、よろしく！

まずは圏論の意義から説明するね。

圏論の素晴らしさは――多様な概念を**圏**という、ただ一つの単位に落とし込むところにある。

へえー？

数学の各分野には、とっても多くの概念があるんだ。整数とか、集合とか、関数とか。

どっかで聞いたことがあるような、ないような単語だね。

うん。圏論はそれらの構造を――「圏」という定義に統一して、注視するための分野なんだ。

えーと、なんかまとめる感じなのかな。

そう。

まず各分野の概念を圏としてまとめる。次に圏を使って、事柄を述べる。

そうすることで、各分野で全く同じことを各々述べてしまったり……っていう、余計なことを回避できるんだ。

あー、なるほどね。各々の数学分野を、まとめてるんだ。

そうそう！

……そうしたらさっそく数学・圏論という、大きな宇宙に――　一緒に行こう。

……うん！

3.3 （本章での参考文献）

- 圏論 - Wikipedia（2019-04-14時点）: https://ja.wikipedia.org/wiki/圏論

第4章　集合

じゃあ数学を――始めていこう。

うん、よろしく。

これから圏論に入門するために、まずは前知識を学ぼう。

まずは**集合**というものを語る分野、**集合論**について。そしてその次に、各用語について。

ドキドキ……。

臆することは、ないと思うよ。できるだけ、やさしく教えるつもり。

それに圏論の準備と侮るなかれ。これらも、とても面白いんだ。

そっか。ηがそういうなら、そう思う！

ぜひリラックスして、楽しんで欲しい。難しく考えすぎずにね。

4.1　集合と写像

じゃあ、集合論を始めよう。

集合について、次の概念を説明していくよ。

- 集合・元（要素）・濃度
- 写像・合成・全射・単射

ねえη。そもそも集合ってどんなものなの？　何の役にたっているの？

集合は単純にいえば、ものの集まりのことだよ。

これがまた数学分野の基礎になっていてね、多くの数学が集合論を参照する。圏論もそう。

だから集合について知ることは、数学の基礎を知ることに通づる。最初の取っ掛かりとしては最適だよ。

そうなんだ。それは面白そうかも！

じゃあ、説明を始めるね。

さっきも言ったけど、集合は**もの**の**集まり**のことだよ。例えば三角形の集合の名前をTとすると、こんな感じに書いたり

```
T = { △, ▲, ⊿ }
```

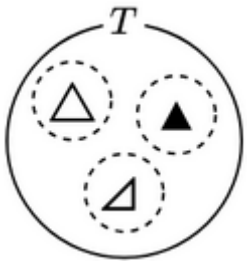
▲リスト4.1: 外延的記法での表現

こうやって書いたり

```
T = { x | xは三角形 }
```

▲リスト4.2: 内包的記法での表現

あるいは視覚的に書いたりする。



▲ 図4.1: 図での表現

ふふ、なんだか顔みたい。

はは、確かに顔に見えるかもね。……こほん。

1つ目のを**外延的記法**、2つ目のを**内包的記法**と言うよ。

3番目は？

単に**図**かな。特別な呼び名は、一般的にはないと思う。

へえー。

外延・内包・図……。書き方、いっぱいあるんだね。

まあね。

外延的記法は**具体例の列挙**。目にもわかりやすいし、テキストとして書ける。

内包的記法は**性質による記述**。ものが多すぎるときにも、わかりやすく書ける。

そして図は――**目で見て**、とってもわかりやすい。

つかいわけかな。

あ〜……。

今は「そういう3つの書き方があるんだ」っていう認識で大丈夫だよ。

えへへ、ありがと！

いえいえ。

そして集合Tの中の3つの三角形。さっきから「もの」と呼んでいるやつだね。

これを集合の**元**（げん）・または**要素**という。∈という記号を使って、こう書くよ。

△ ∈ T
▲ ∈ T
◿ ∈ T

または

T ∋ △
T ∋ ▲
T ∋ ◿

あるいは、まとめて

△, ▲, ◿ ∈ T

▲リスト4.3: 集合Tの元

▲を例に取って、「▲はTに**属する**」とも。

ちなみにその否定、「**属さない**」ことは「∉」と書くよ。

げん・ようそ。それにも名前があるんだ。

雰囲気のまま「もの」って言ってたら、意味が曖昧になってしまって……言葉を使っている間に、勘違いが起きてしまうかもしれないからね。

用語で意味をしばっておくのは、とっても大切なことなんだ。

ふふ。それ、なんかかっこいいね。

そう？　はは、そうだね。

4.1.1　濃度・有限集合・無限集合

集合Tの元の個数は3だった。じゃあ次の集合をCとすると、元の個数はなんだと思う？

```
C = { ○, ● }
```

▲リスト4.4: 丸の集合C

えーと、2？

その通り。

こうやって集合の元の個数――つまり**集合の大きさ**――を問うことは、よくある。

これを**濃度**という。

のうど……濃さ？

そう、集合の濃さだね。

そして――その集合の元の個数は、必ずしも有限じゃない。例えば「全ての自然数の集まり」は集合になるけど、元が無限にある。*1

『自然数すべての個数は、いくつ？』なんて聞かれたら……答えられないよね。

えーと、うん。1000とか10000000とか、もっと大きい数があると思う。

そう、つまり自然数には「一番大きい数」がないんだ。元が「無数」にある。

そういう、元が無数にある集合を**無限集合**というよ。

逆にそうでない、元の個数を答えられる集合を**有限集合**という。

集合TやCが有限集合。自然数とかが無限集合、ってことかな！

うんうん、そうそう。

自然数の集合の名前をℕとして書くと、こう。*2

```
ℕ = { 0, 1, 2, ... }
―― ここで「...」は、意味が明らかな、可算無限な省略。
```

[*1] 全ての自然数の集まり（**自然数全体**）とは {0, 1, 2, ... }という、物を数える数のこと。ときによって、0以上ではなく1以上を指す場合もあるよ。

[*2] 自然数の英名 "N"atural number の "N" だよ。

▲リスト4.5: 自然数の集合$\mathbb{N}$（エヌ）

おおー！……可算？

実はね、無限集合……「無限」にも、いくつか種類があるんだ。それは――**可算無限**と、**非可算無限**。

えっとー、うんうん。

圏論の基本には関わってこないから、小難しい詳細は省かせてもらうけど……

無限には――可算無限という、**自然数と同じ**濃度。そして非可算無限という、**実数と同じ**濃度。――が、あるんだ。

実数ってなんだっけ。

-10000.0とか、0.0とか、1.1とか、3.333333…とか。1/3とか、√2とか、πとか……自然数よりも、多くの数が含まれる、集合のことだね。

いろんな種類の数がはいった集合なんだ。

そう言って、差し支えないよ。

つまり濃度とは、次の代表的な3通りがある。「この3通りがある」ってことを覚えてもらえれば、十分かな。[*3]

[*3] この3通りしかないわけじゃないよ。でも常々考えられるのは、だいたいこの3通りかな。

- n個 （0以上の、有限な個数）
- 可算無限個（自然数と同じ大きさ）
- 非可算無限個（実数と同じ大きさ）

ふんふん。集合の大きさは3通り、だね！

……

ここまでで集合そのものの知識については、完了だよ。

最後に有用な知識として、いくつかの集合の具体例を見てみよう。

- 有限な集合
  - 三角形の集まり
  - 自然数のうち 0 〜 10
- 可算無限な集合
  - 自然数
  - 整数 *4
  - 偶数 *5
  - 奇数 *6
- 非可算無限な集合
  - 実数
  - 実数のうち 0.0 〜 10.0

「実数のうち 0.0 〜 10.0」も非可算無限なんだ。

そう。実数の一部分を切り取っても、それは実数の濃度と等しくなるんだ。

つまり全体と一部分の大きさが、等しい。……面白いね。

[*4] 整数とは、-1以下・0・1以上――の数の集まりのこと。

[*5] 偶数とは、整数のうち2で割り切れる数の、集まりのこと。

[*6] 奇数とは、整数のうち2で割り切れない数の、集まりのこと。

えっと……？？……？

今回そこは考えなくて大丈夫！　頭があふれてしまいそうなら、無理をする必要はないから。

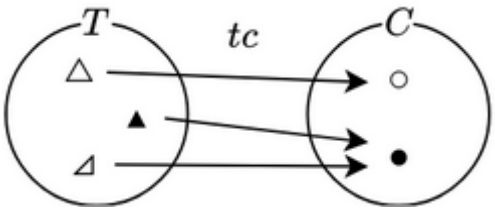
とーっても面白い事象なんだよ……ってね。

へえー、そうなんだ。ありがとう。いつか理解できたときに、思い出してみるね！

4.1.2　写像

集合はそれだけでも便利だし面白いんだけど、さらに広いことを考えるために、**写像**という概念がある。

写像というのは、集合と集合を結びつける概念だよ。例えばTからCへの写像をtcとすると……こう書ける。



▲ 図4.2: 写像tc : T -> C

うん、矢印だね。

そう、矢印。

正確には――「集合の**全ての元**」を「他方の集合のいずれかの、1つの元」へ割り当てること。

これを**写像**というよ。

またこのように各元を写像で割り当てることを、**写す**という。「写像tcは、Tの各元をCに写す」……という感じ。

言葉では、次のように定義できる。

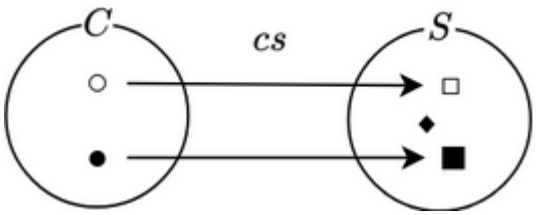
```
tc : T -> C
tc(△) = ○
tc(▲) = ●
tc(◿) = ●

もしくは以下のように書くこともある。

tc : T -> C
tc : △ |-> ○
tc : ▲ |-> ●
tc : ◿ |-> ●
```

▲リスト4.6: 写像tc : T -> C

なるほど、「写像は各元から各元への割り当て」なんだね。なんか、流れてる感じがするかも。

そうそう、その通りだよ。写像は、一方から一方への流れっていうイメージが強いね。

次は別の写像の例として、丸から四角へのものを書いてみよう。



▲ 図4.3: 写像cs : C -> S

```
cs : C -> S
cs(○) = □
cs(●) = ■
```

▲リスト4.7: 写像cs : C -> S

そしてここが大事なんだけど――

tcのように、割り当て先の集合の、**全ての元に矢印**が当たっているような写像を**全射**。

csのように、割り当て先の集合の、**各元に矢印が1つだけ当たっているか、当たっていない**か……つまり2つ以上の割り当たりがない写像を**単射**

――と呼ぶ。

えーと……tcはCの元○,●どちらにも割り当ててるから、全射って感じかな？　csは、うーんと……

tcについてはまさにその通りだよ。csが単射であることについては……ほら、例えばtcには、2つの割り当たりをもつ元あるんだ。●だね、▲と◿からの。

csはそのような元――2つ以上の割り当たりを持つ元がない。だから単射なんだ。

おー、本当だ！　わかりやすい説明、ありがと。

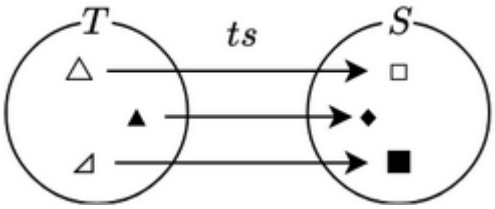
いえいえ。

写像は、単射と全射にわけられるのかな。

おしい。

実は**全単射**という、全射でも単射でもある写像。そして全射でも単射でもない写像も考えられる。

次のtsは全単射。

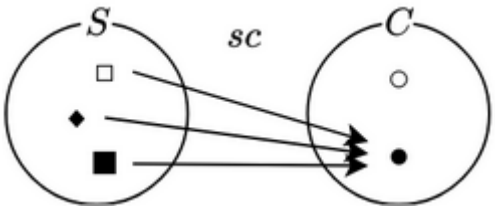


▲ 図4.4: 写像ts : T -> S

ほんとだ。2つ以上の割り当たりがある元もないし、割り当たりのない元もないね。

そうそう。

そして次のscは全射でも単射でもない、ただの写像だよ。



▲ 図4.5: 写像sc : S -> C

全部で4パターンあるんだね！

こうかな？

- 全単射
- 全射（単射でない）
- 単射（全射でない）
- 全射でも単射でもない

その通り！

4.1.3　写像の合成

最後に、**写像の合成**と呼ばれるものを見てみよう。

写像の合成とは――2つの写像を合わせて、もう1つの写像を定義するものだよ。

写像の合成は、次のように定義される。

ある写像f, gに対して
f : X -> Y
g : Y -> Z
次のように定義される。

g ○ f : X -> Z
(g ○ f)(x) = g(f(x))

▲リスト4.8: 写像の合成○

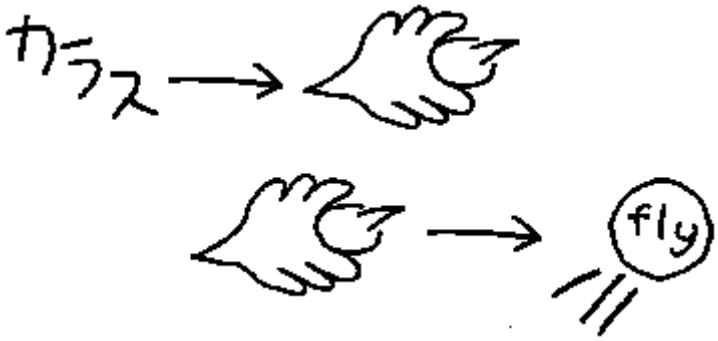
えーと、ごめん。どういうことだろ。

写像はよく、言葉の「三段論法」に比喩されるよ。「AはB、BはCである。……なので、AはCである。」というものだね。

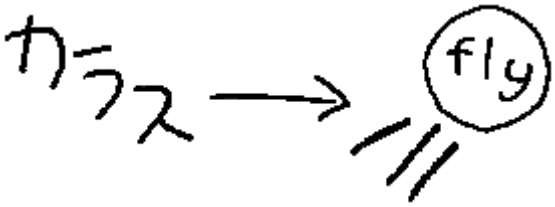
あー、それ知ってる！

うんうん。例えば「カラスは鳥。鳥は飛ぶ。なので、カラスは飛ぶ。」とかね。

（これは……鳥の絵か！　）



▲ 図4.6: カラスは鳥。鳥は飛ぶ。



▲ 図4.7: なので、カラスは飛ぶ。

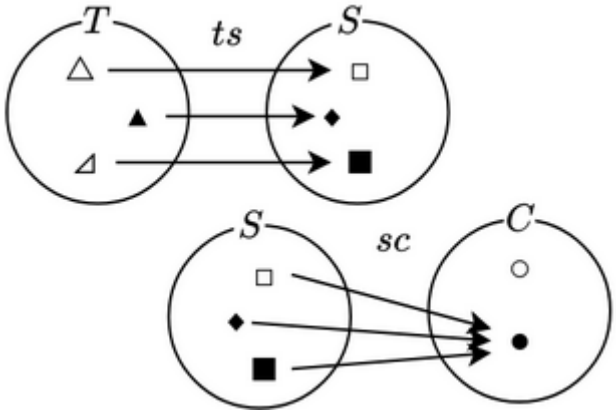
これと同じようにg ○ fを言うと

「（fが）XからY、（gが）YからZである。ならば、（g ○ fが）XからZである。」

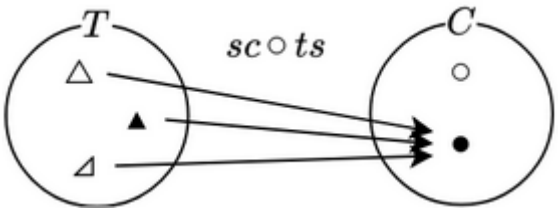
となる。さっき定義した写像tsとscを使って、写像sc ○ tsの定義を確認してみよう。つまり

「（tsが）TからS、（scが）SからCである。ならば、（sc ○ tsが）TからCである。」

を。



▲ 図4.8: （tsが）TからS、（scが）SからC。



▲図4.9: ならば、（sc ○ tsが）TからC。

写像についても、三段論法が通用する……ってことなのかな？

まあ、そうだね。三段論法と似通った性質が写像にはある、って感じかな。

わかった気がする！　ありがと♪

いえいえ、よかった。

そうしたら、今度は言葉でも書いてみよう。

```
(sc ○ ts)(△)  =  sc(ts(△))   ── (sc ○ ts)(x) = sc(ts(x)) を展開
              =  sc(□)        ── ts(△) = □ を展開
              =  ●

(sc ○ ts)(▲) = (同じく展開) = ●
(sc ○ ts)(⊿) = (同じく展開) = ●
```

▲リスト4.9: 写像sc ○ ts : T -> C

へぇ、そっか。△を□に写したあとに、●に写すってことなんだ。

そうそう、よくわかったね。**写した元をまた写す**――っていうを各元に行うのが、合成された写像だよ。

わあ、なるほどね。

写した元をまた写して、三段論法っぽいことをする。それが写像の合成なんだね！

4.2　行き「もと」と「さき」が同じ写像

そういえば、tcもTからCへの写像だったよね。それで、sc ○ tsもそうだよね。

tcとsc ○ tsはちがうものなの？

うん。tcが△を○に写すのに対して、sc ○ tsは△を●に写すよ。

TからCへの写像は、何パターンか別のものを考えられるんだ。

そうだよね、ちょっと気になってたんだ。よかった！

4.3　まとめ

これで集合については終わり。お疲れ様！

おつかれさま、ありがと！

いいえ。

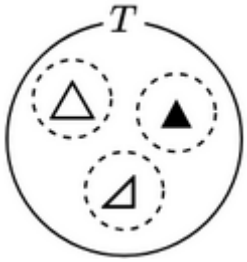
じゃあ最後に、理解したことをまとめてみよう。

はーいっ！

集合は**元**と呼ばれるものの、集まりのこと。それは**外延的記法・内包的記法**・あるいは**図**で書かれる。

```
C = { ○, ● }
S = { x | xは四角形 }
```

▲リスト4.10: 外延的記法・内包的記法、での表現



▲ 図4.10: 図での表現

うんうん。つかいわけが大切なんだったね。

そう。

そしてそれらには**濃度**と呼ばれる、集合の大きさが定義される。

濃度には種別がある。**有限**――例えば3・4・または10000など。そして**可算無限・非可算無限**。

具体的な集合として、次のものがある。

- 有限な集合
  - いくつかの三角形の集まり
  - 自然数のうち 0 〜 10
- 可算無限な集合
  - 自然数 全て
  - 整数 全て
  - 偶数 全て

    - 奇数 全て
- 非可算無限な集合
    - 実数 全て
    - 実数のうち 0.0 〜 10.0

有限と可算無限・非可算無限だね。だね。

それら集合の間には、**写像**というものが定義できる。写像とは、ある集合の全ての元から、集合の元への割り当てのこと。

写像は4種類に分けられる。

- **全単射**
- **全射**（単射でない）
- **単射**（全射でない）
- 全射でも単射でもない

全単射は、全射でも単射でもあるんだったね。

そうそう。

そしてf : X -> Yとg : Y -> Zのような、Yで繋がる写像は、**合成**という概念が使える。その合成○とは、写像g ○ f : X -> Zを定義するものである。

……以上。

ありがとう、理解できた気がするよ♪

よかった。

これでμは、数学という宇宙に、足を踏み入れたということになる。どうかこの宇宙を、楽しんでいって欲しい。──なんてね。

ははは、へんなのー。

……

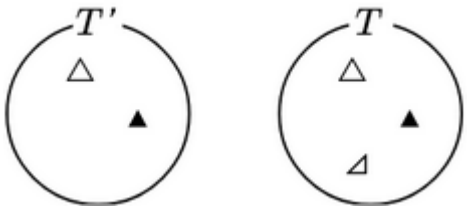
4.4　ノートの切れ端A

μへ。もし興味があれば、調べてみてね。ηより。

4.5　部分集合

4.5.1　含む

部分集合という、ある集合が他方の集合の一部分であることを表す概念がある。例えば以下の集合T'（ティープライム）は、集合Tの部分集合だよ。



▲図4.11: 集合Tと、その部分集合T'

このとき「TはT'を**含む**」といって、次のように書くよ。

```
T' ⊆ T　または　T ⊇ T'
```

▲リスト4.11: 集合Tは集合T'を含む

また「SがS'を**含まない**」ことは、次のように書く。

```
S' ⊄ S　または　S ⊅ S'
```

▲リスト4.12: 集合Sは集合S'を含まない

1)　他の「X ⊆ Y」の形になる集合と集合を、いくつか見つけてみて。

4.5.2　空集合

集合の一例として、**空集合**と呼ばれる、全く要素を持たない集合がある。

```
φ = {}
```

▲リスト4.13: 空集合φ

あらゆる集合は、この空集合を部分集合として含むよ。例えば集合Tに対して、次のように書ける。

```
Φ ⊆ T　または　T ⊇ Φ
```

▲リスト4.14: 集合Tは空集合Φを含む

4.5.3　真に含む

実は、単に「含む」や「⊆」「⊇」といった場合には、その集合自身も含み得る。例えばTに対して「TはTを含む」や、次のようにも書ける。

```
T ⊆ T　または　T ⊇ T
```

▲リスト4.15: 集合Tは集合Tを含む

その集合自身を含まないことを強調したい場合には「**真に含む**」や「⊂」「⊃」という。例えばTに対して「TはT'を真に含む」や、次のように書くよ。[*7]

```
T' ⊂ T　または　T ⊃ T'
```

▲リスト4.16: 集合Tは集合T'を真に含む

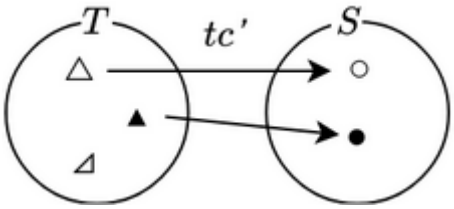
このとき、Tに対するT'のような集合のことを、**真部分集合**というよ。

「∈」「⊆」「⊂」いずれも似ているから、注意だね！

1)　Tが真に含み得る、全ての部分集合を見つけて。例えば Φ, {△}, {▲} 。残りは……？

4.5.4　部分写像

以下のtc'（ティーシープライム）を見てみよう。



▲ 図4.12: 写像？ tc' : T -> C

[*7] あるいはさらに強調して「T' ⊂ T かつ T' ≠ T」とも。

これはTの元△，▲だけをCの元に割り当てている。言い換えれば元⊿をCのいずれの元にも割り当てていない。

このtc'のように『「集合の**一部の元**」を「他方の集合のいずれかの、1つの元」に割り当てるもの』を**部分写像**という。例えばこのtc'は**部分写像tc'**と呼ばれる。

部分写像は、写像の定義である『「集合の**全ての元**」を「他方の集合のいずれかの、1つの元」に割り当てること』を満たさないので、一般的には写像とはみなされないよ。[*8]

4.5.5　写す「もと」と「さき」が同じ写像

写像は、写すもととさきが同じものも定義できる。例えばtt ： T -> Tのような形に。

1)　T -> Tの具体的な定義を、いくつか考えてみて。

4.5.6　元の重複

集合の元は、重複しない。

例えば、以下の左右の集合は、等しい。

また、これらの濃度はどちらも2になる。

```
{ 1, 2, 1 }  =  { 1, 2 }
```

▲リスト4.17: 元「1」は一つだけ

元1が二度以上あらわれても、一度あらわれるのと同じ、ってことだね。

ちなみに、元のあらわれる順番も、集合では問われないよ。

```
{ x, y }  =  { y, x }
```

▲リスト4.18: 集合の元に、順番はない

4.5.7　同じ濃度

可算無限は、自然数と同じ濃度。非可算無限は、実数と同じ濃度。――であると言った。

ある集合XとYの**同じ濃度**であるというは、実は「XとYの間に、**全単射が存在する**」と定義される。

例えばさっき見たように、自然数と整数は同じ濃度なんだ。その間に、一例として、以下のような全単射が考えられるから。

[*8] 部分集合の写像とみなすことは可能だよ。

```
f : 自然数 -> 整数
f 0 =  0
f 1 = -1
f 2 =  1
f 3 = -2
f 4 =  2
...
```

▲リスト4.19: 自然数から整数への全単射

偶数と整数の大きさも、実は同じ。偶数は整数の一部なのにね……不思議。素敵だね。

だから、次の問題も、よかったら考えてみて。

1) 偶数から整数への全単射gの、*リスト4.20*の「?」を埋めてみて。
2) 同じように、奇数から整数への全単射を、考えてみて。

```
g : 偶数 -> 整数
g 0 = 0
g 2 = 1
g ? = 2
g ? = 3
g ? = 4
```

▲リスト4.20: 偶数から整数への全単射

◆ ◆ ◆

お疲れ様 :)

4.6 （本章での参考文献）

- 集合 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/集合*
- 濃度 (数学) - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/濃度_(数学)*
- デデキント切断 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/デデキント切断*
- 連続体濃度 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/連続体濃度*
- 部分集合 - Wikipedia（2019-08-12時点）: *https://ja.wikipedia.org/wiki/部分集合*
- 部分写像 - Wikipedia（2019-08-13時点）: *https://ja.wikipedia.org/wiki/部分写像*

第5章　Haskell

次は圏論のもうひとつの題材として、Haskellというのを見てみよう。

Haskellって、どういうものなの？

Haskellは実用的なプログラミング言語だよ。プログラミング言語っていうのは、機械に頼み事をするときに使う言葉のこと。

へえー。機械のための言葉なんだ。

大まかには、そう。

しかしそれだけではなくてね。プログラミング言語はしばしば数学に関わってくるんだけど、Haskellその中でもかなり密接でね。Haskell自体が数学のひとつとして見られることも、よくあるくらいなんだ。

ほうほう。プログラミング言語であって、数学でもあるんだ。

そうなんだ。僕はプログラミング言語の中で、Haskellが一番好きなんだ。

じゃあ、始めていこうか。

ηが一番好き、かあ。

うん、よろしくお願いします♪

5.1　型と値

集合論には「集合と元」という概念があったよね。Haskellにもそれらと似た、**型**と**値**という概念がある。

型と値はdataというキーワードを使って、次のように書かれる。

```
-- Haskellでの記法↓
data T = Unfilled | Filled | Oblique

-- 集合での記法↓
-- T = { △, ▲, ◿ }
```

▲リスト5.1: 型Tの定義

ちなみに「{- foo -}」とか「-- foo」とか書いてあるのは、コメントと呼ばれる――メモみたいなものだよ。

ここでTを型。Unfilled, Filled, Obliqueを値という。

Unfilled, Filled, Oblique……？

Haskellでは表現上の理由で、「△▲◿」みたいな記号は使えないんだ。だから代わりに、英名。

ああ、なるほど。色が塗られてない△だからUnfilled、黒塗りの▲だからFilled、斜めだからObliqueってことね。

そうそう。

そして型と値について、x ∈ Tっぽい書き方が提供されてるよ。

```
-- △ ∈ T
Unfilled :: T

-- △, ▲, ◿ ∈ T
[Unfilled, Filled, Oblique] :: [T]
```

▲リスト5.2: Tの値

下の記法は「Unfilled, Filled, Oblique :: T」じゃないんだね。

そうなんだよ。ちょっとややこしいから「Unfilled, Filled, Oblique ⊆ T」と間違えないようにね。

はーいっ！

typeというキーワードで、型の別名をつけることもできる。

```
-- Triangle = T
type Triangle = T

-- △ ∈ Triangle
Unfilled :: Triangle

-- △, ▲, ◿ ∈ Triangle
[Unfilled, Filled, Oblique] :: [Triangle]
```

▲リスト5.3: 型Triangle = T

あとは予め定義されている、基本的な5つの型を紹介しよう。

ユニット
() ・Bool・Char・Int・Stringだよ。

```
-- ただ一つの値()のみを表す型()
() :: ()
-- 値() ∈ 型()

-- 真偽を表すBool
[True, False] :: [Bool]
-- True, False ∈ Bool

-- 全ての文字を表すChar
['a', 'b' .. 'z'] :: [Char]
-- 'a', 'b', ..., 'z' ∈ Char

-- 整数を表すInt *1
[-10000, -9999 .. 10000] :: [Int]
-- -10000, -9999, ..., 10000 ∈ Int

-- 全ての文字列を表すString
["mu", "eta", "are", "happy"] :: [String]
-- "mu", "eta", "are", "happy" ∈ String
```

▲リスト5.4: 型Bool・Char・Int・String

ユニット・真偽・文字・整数・文字列。それら自体はわかった気がする。

これらはプログラミングにおいて、かなり頻出する型なんだ。僕らの講座でも、しばしば使っていこうと思うよ。

はーい♪

[*1] 実際は、Intは最小値と最大値が定義されてしまっている。Integerっていう、ほとんど整数と同じ型があるんだけど……計算が遅いんだよね。

5.2 関数

今度は「写像」に似た概念、**関数**(かんすう)についてやっていこう。

関数は、次のように定義できるよ。写像を定義するときと、似ているね。

```
isZero :: Int -> Bool
isZero 0 = True   -- 0ならばTrue
isZero _ = False  -- それ以外ならばFalse

-- isZero(0) = True
-- isZero(x) = False  (xは0以外)
```

▲リスト5.5: 関数isZero

本当だ。写像Int -> Boolに似ているね。

うんうん。

今のと全く同じ意味で、次のようにも書けるよ。ちょっとプログラミングっぽいね。

```
isZero :: Int -> Bool
isZero = \x -> case x of
  0 -> True
  _ -> False
```

▲リスト5.6: 関数isZero

case x of？？

「case x of ; a -> y ; _ -> z」は「xがaならばy、それ以外ならばz」って読むよ。

ああ、だからひとつ前の表記と同じなんだね！

そ。

そして値を関数に与えるときは、次のように書く。

```
isZero 0     {-  True   -}
isZero 10    {-  False  -}

-- isZero(0)
-- isZero(10)
```

▲リスト5.7: 値を関数に与える（関数適用）

ちなみにこのisZero 0の0・isZero 10の10など、関数に与えている値を**引数**（ひきすう）というよ。

Haskellにはかっこがないんだ。

そう。Haskellでは本的に、括弧を付けずに、値を与える。

つける必要はないけど、あえて括弧をつけることもできるよ。

```
isZero(0)     {-  True   -}
isZero(10)    {-  False  -}
```

▲リスト5.8: 括弧つき関数適用

今度は関数適用ではなく、関数定義側で括弧を使ってみよう。そうすると、次のように関数の名前に記号が使えるようになって

```
-- 掛け算
(*) :: Int -> Int -> Int
(*) x 0 = 0
(*) x y = x + (x * (y - 1))
```

▲リスト5.9: 記号を使った関数への命名

関数適用の形式が4通りになるよ。

```
-- いずれの答えも6
(*) 2 3    --  通常の関数適用
(2 *) 3    --  前置のセクション記法
(* 3) 2    --  後置のセクション記法
2 * 3      --  演算子適用
```

▲リスト5.10: 3通りの適用方法

えっと……セクション記法？

やっぱりここは難しいよね。こんな感じに順を追ってみると、わかるかな。

```
(* 3) 10
-- = 10 * 3
--
-- (* 3)は「次に受け取る値『に』3『を』掛ける」という関数。
-- (x * 3)のxに10を入れて、計算するイメージ。

(+ 10) 2    {-  2 ＋ 10  =  12  -}
(-  5) 8    {-  8 ー  5  =   3  -}
(/  2) 4    {-  4 ÷  2  =   2  -}

(1 +) 5
-- = 1 + 5
--
-- (1 +)は「次に受け取る値『を』1『に』足す」という関数。
-- (1 + x)のxに5を入れて、計算するイメージ。

(5 +) 7     {-  5 ＋  7   =  12  -}
(9 -) 1     {-  9 ー 1   =   8  -}
(8 /) 4     {-  8 ÷ 4   =   2  -}
```

▲リスト5.11: セクション記法と関数適用

ああ、そっか。(* 3) 10は左側に10を、(1 +) 5は右側に5を回すんだね。

そう！　理解が早いね。

えへへーん。

関数内部では、let-in束縛という機能使える。

```
isOdd :: Int -> Bool
isOdd x =
  let even = isEven x  -- 1個のlet-in束縛
  in not even

isEven :: Int -> Bool
isEven x =
  let isSatisfied = (x == 0) || (x == 1)
      rest = x - 2 in  -- 2個以上のlet-in束縛
  case isSatisfied of
    True -> isZero x
    False -> isEven rest
```

▲リスト5.12: 関数isOdd・isEven

```
isOdd  6    {-  False  -}
isOdd  1    {-  True   -}
isOdd  3    {-  True   -}
isOdd  6    {-  False  -}
isOdd  9    {-  True   -}

isEven 1    {-  False  -}
isEven 2    {-  True   -}
isEven 3    {-  False  -}
isEven 6    {-  True   -}
isEven 9    {-  False  -}
```

▲リスト5.13: 使用方法isOdd・isEven

let-in束縛ってなんのためにあるの？

let-inを使った方が、プログラムのコードが見やすいことがあるんだ。コードの見やすさのためだね。

へえー、なるほどねー。

Haskellは、集合を扱うものなの？

率直なところ、そうとも言えるね。
「型理論（かたりろん）」っていう数学分野があってね、これも集合論を参照しているんだ。そしてHaskellは、その型理論を参照しているから。

へえー……。

まあ型理論を十分に語るには、時間が足りなさすぎるから……ここはHaskellに注視しておこう。

5.3　発展的な型

発展的な型と値を見てみよう。次の4つだよ。

- (X, Y)
  - XとYのペアを表す
- Either X Y
  - XもしくはYを表す
- [X]
  - 複数のXの値を表す
- Maybe X
  - Xの値が1つあるか、もしくはないことを表す

```
-- IntとCharのペア
[(-10, 'a'), (10, 'z')] :: [(Int, Char)]
--  (-10, 'a'), (10, 'z')  ∈  (Int, Char)

-- IntもしくはChar
[Left -10, Right 'a'] :: [Either Int Char]
--  -10, 'a'  ∈  Either Int Char

-- 複数のIntの値
[ [],
  [1, 2, 3],
  [2, 2, 4, 6, 10, 16] ] :: [ [Int] ]
--  [],
--  [1, 2, 3],
--  [2, 2, 4, 6, 10, 16]  ∈  [Int]

-- Xの値、もしくは「ない」
[Just 10, Just 252, Nothing] :: [Maybe Int]
--  Just 10, Just 252, Nothing  ∈  Maybe Int
```

▲リスト5.14: 型(X, Y)・Either X Y・Maybe X

「かつ」と「もしくは」がよくわかんないかも……。

ちょっと難しいところだね。

(Int, Char)の値は、例えば「あるIntの値x」と「あるCharの値y」の両方を組み合わせた(x, y)というものなんだ。

逆にいうと「-10だけ」とか「'a'だけ」とかは、(Int, Char)の値ではない。

```
-- 以下は成り立つ
(-10, 'a') :: (Int, Char)    --    (-10) ∈ (Int, Char)
(252, 'q') :: (Int, Char)    --    (252) ∈ (Int, Char)
(100, 'z') :: (Int, Char)    --    (100) ∈ (Int, Char)
```

▲リスト5.15: イイ: IntとCharの両方を組み合わせた値

```
-- 以下は『成り立たない』
-10 :: (Int, Char)    --    -10 ∈ (Int, Char)
252 :: (Int, Char)    --    252 ∈ (Int, Char)
100 :: (Int, Char)    --    100 ∈ (Int, Char)
'a' :: (Int, Char)    --    'a' ∈ (Int, Char)
'q' :: (Int, Char)    --    'q' ∈ (Int, Char)
'z' :: (Int, Char)    --    'z' ∈ (Int, Char)
```

▲リスト5.16: ダメ: Intの値だけ・Charの値だけ

またEither Int Charの値は、「あるIntの値」もしくは「あるCharの値」のどちらかというもの。

逆にいうと「(-10, 'a')」や「(252, 'q')」とかは、Either Int Charの値ではない。

(Int, Char)と対称的だね。

```
-- 以下は成り立つ
Left  (-10) :: Either Int Char    --    -10 ∈ Either Int Char
Left   252  :: Either Int Char    --    252 ∈ Either Int Char
Left   100  :: Either Int Char    --    100 ∈ Either Int Char
Right  'a'  :: Either Int Char    --    'a' ∈ Either Int Char
Right  'q'  :: Either Int Char    --    'q' ∈ Either Int Char
Right  'z'  :: Either Int Char    --    'z' ∈ Either Int Char
```

▲リスト5.17: イイ: Intの値だけ・Charの値だけ

```
-- 以下は『成り立たない』
(-10, 'a') :: (Int, Char)    --    (-10, 'a') ∈ (Int, Char)
(252, 'q') :: (Int, Char)    --    (252, 'q') ∈ (Int, Char)
(100, 'z') :: (Int, Char)    --    (100, 'z') ∈ (Int, Char)
```

▲リスト5.18: ダメ: IntとCharの両方を組み合わせた値

一度に両方持てるけど、片方だけは持てないのが(X, Y)。片方だけを持てるけど、一度に両方は持てないのがEither X Y。

かな？

いい説明だね。正しい。

よかったあ♪　わかってきたかも。
ただ、その……Left・Rightって？

それは、どっちが左右どちら側の値か判別をつけるためのものなんだ。
Either X YでXの値を使うときは「左側を表すLeft」を、Yの値を使うときは「右側を表すRight」を……って感じだね。

じゃあEither Int Charだったら、Left 'a'やRight -10はだめなんだ。

その通り。

ふふ、ありがとう♪

いいんだよ。

……。
あー……えっと、ごめん。
やっぱり[X]とMaybe Xもわかんないや。教えてくれる……？

もちろん。任せて。
まず[X]は「複数のXの値」を表すよ。

2つのXの値を表す(X, X)とは違うの？

勘がいいね、そこはよく混同される。

(X, X)は「必ず」2つの値を持つのに対して、[X]は「必ずしも」2つじゃない。

(X, X)は1つ以下・3つ以上の要素を持てないのに対して、[X]は0以上の不特定な個数を持つよ。

```
[]          ::  [Int]    --    []        ∈  [Int]
[10]        ::  [Int]    --    [10]      ∈  [Int]
[10, 20]    ::  [Int]    --    [10, 20]  ∈  [Int]
[1 .. 100]  ::  [Int]    --    [1, 2, ..., 100]  ∈  [Int]
```

▲リスト5.19: 0個・1個・2個・100個など

なるほど！　ほんとだ。

でしょ。

そしてMaybe Xは単純で、Either () Xの短縮形だよ。実際、次のように、使い方が一致する。見比べてみてね。

```
Nothing    :: Maybe Char    --    無  ∈ Maybe Char
Just (-10) :: Maybe Char    --    -10 ∈ Maybe Char
Just  252  :: Maybe Char    --    252 ∈ Maybe Char
Just  100  :: Maybe Char    --    100 ∈ Maybe Char
```

▲リスト5.20:

```
Left  ()    :: Maybe Char    --    無  ∈ Either () Char
Right (-10) :: Maybe Char    --    -10 ∈ Either () Char
Right  252  :: Maybe Char    --    252 ∈ Either () Char
Right  100  :: Maybe Char    --    100 ∈ Either () Char
```

▲リスト5.21:

ほんとだ、一致してる。じゃあMaybe Xって、なんであるの？

書きやすいからかな。

……なるほど！！

5.4　型クラス

次は**型クラス**と、型クラスの**インスタンス**について。
ひとつ、例を見てみよう。

```
class Show a where
  show :: a -> String

instance Show Int where
  show = {- 実装 -}

instance Show Bool where
  show True  = "True"
  show False = "False"
```

▲リスト5.22: Show型クラスとインスタンス

```
greet :: Show a => a -> String
greet x = "hi, " ++ show x  -- ++は文字列を結合する演算子
```

▲リスト5.23: 確認用関数

```
greet 10     --  "hi, 10"
greet True   --  "hi, True"
```

▲リスト5.24: 確認

関数定義は普通、特定の型を受け取るんだ。isZeroや(*)のようにね。

でもgreetはShow a => aと書くことによって、IntやBoolといった不特定の型を受け取っている。greet 10、greet Trueというように、IntとBoolどちらの値も渡せていることからわかる。

これはプログラミングでは重要でね。これがなければ例えばgreetを書く代わりに、2つの関数を別々に定義する必要が出てくる。

内容は全く同じなのにね。これはプログラミング上、大きな手間になる。

```
greetInt :: Int -> String
greetInt x = "hi, " ++ show x

greetBool :: Bool -> String
greetBool x = "hi, " ++ show x
```

▲リスト5.25: 型クラスを使わない場合のgreet

```
greetInt 10     --  "hi, 10"
greetBool True  --  "hi, True"
```

▲リスト5.26: 確認

型クラスを使えば、手間をかけないですむんだね。

そう、その通り。もっと理屈っぽく言ってみようか。

型クラスはね、ある種の性質というものを表すんだ。例えばShowは「文字列にできる」っていう性質だね。

そしてそのインスタンスは、指定された型がその性質を持つこと表す。例えばShow Intは「IntはShowの性質を持つ」ということを。Show Boolも同じ。

えっとー……？

ごめんね。理屈っぽく言った部分は大切なことだけど、今はわからなくても大丈夫だよ。

……おっけー！

もうひとつ、圏論との関連が深い型クラスを見てみよう。

Functor。これは「要素に関数を適用できる」ことを表すよ。

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

instance Functor Maybe where
  fmap = {- 実装 -}

instance Functor (Either a) where
  fmap = {- 実装 -}

instance function [a] where
  fmap = {- 実装 -}
```

▲リスト5.27: Functor型クラスとインスタンス

```
fmap (+1) Nothing          --  Nothing
fmap (+1) (Just 10)        --  Just 11

fmap (+1) (Left "wrong")   --  Left "wrong"
fmap (+1) (Right 10)       --  Right 11

fmap (+1) []               --  []
fmap (+1) [10, 20, 30]     --  [11, 21, 31]
```

▲リスト5.28: 確認

Nothing・Left・[]なら何もせず、Just・Right・[]以外の[X]値ならば+1するよ。

これはプログラミング上、かなり便利なんだ。

えっと、ほー……？

……頭があふれてしまうね。圏論に関係ない話はやめておこう。

ここではFunctorとfmapというものがあったことを、頭の片隅に入れてくれれば大丈夫だよ。

えへへ。うん！

5.5 ghci

今見てきたいくつかのことを手元で動かしてみたいときは、ghciっていうHaskell用の電卓みたいなものが役に立つよ。

ghciを起動したら>>>っていう文字が表示されるから、それに続いてキーボードで入力すると……こんな風に。[*2]

```
>>> 'a'
'a'
>>> 10
10

>>> [1 .. 10] :: [Int]
[1,2,3,4,5,6,7,8,9,10]
>>> ['a' .. 'z'] :: [Char]
"abcdefghijklmnopqrstuvwxyz"

>>> show 10
"10"
>>> show True
"True"

>>> fmap (+ 1) (Just 10)
Just 11
>>> fmap (+ 1) (Right 10)
Right 11
>>> fmap (+ 1) [10, 20, 30]
[11, 21, 31]
```

▲リスト5.29: ghci

あ、さっきやってたやつだ。

[*2] 表示される文字は、ghciの設定によって違うよ。

そうそう。

5.6　まとめ

Haskellについては、以上だよ。

まとめ、しようか。

ありがとうございました！　おねがいします♪

Haskellには集合と値に似た、**型**と**値**というものがあった。これらは *リスト5.1* のように定義できて、 *リスト5.3* のように使うことができる。

内容が前後するけど、ghciを使えば確認できるね。

```
>>> data T = Unfilled | Filled | Oblique deriving (Show)

>>> Unfilled :: T
Unfilled

>>> [Unfilled, Filled, Oblique] :: [T]
[Unfilled,Filled,Oblique]
```

▲リスト5.30: 型と値（ghciでの確認）

わ、確認できたね。

うんうん。

型クラスとそのインスタンスは *リスト5.22* という風に定義できて、*リスト5.23* のように扱える。

型クラスを使うと、同じ関数をいくつも定義しなくてよくなるんだったね。

その通り！

……

これでμは、数学分野の具体例として集合に加え、Haskellを得た。必要な装備が揃ってきたね。

ηのおかげ。ありがとうね。

……どういたしまして！

5.7　ノートの切れ端B

お疲れ様 :)

毎度の問題集だよ。

1)　次の型クラスMonの、IntとBoolのインスタンスを定義してみて。（定義内容はなんでもいいよ！）

```
class Mon a where
  add :: a -> a -> a
```

▲リスト5.31: Monoid型クラス

5.8　（本章での参考文献）

- 型理論 - Wikipedia（2019-08-17時点）: *https://ja.wikipedia.org/wiki/型理論*

（後書き）

ηとμには**モデル**になった人がいます。まずは、おふたかたに、感謝。

次に、僕が執筆を始める時間になると、気を使ってくれる。生活を支えてくれる**家族**のみんなに、感謝。

そして、ηとμ、本の**ヴィジュアルデザイン**をしてくださった、 @HassakTb さんにも感謝を。自分たちの産んだ子と再び出会えた、あの瞬間は……何事にも代え難く、忘れられない経験になりました。

さらに、ここに書ききれない多くの方々にも、助けていただきました。

どうか**η**と**μ**を「**こんな子いたなぁ**」と、頭の片隅にでも置いていただければ、本望です。

もちろん、**ファンアート**や**二次創作**も大歓迎です！

ありがとう！

Special Thanks

- 温かい暮らしをくれる母父
- いつも助けてくれる愛犬ペロくん・愛犬ララちゃん
- いつも相談にのってくれるコノコさん
- 自分
- ηとμ
- ηとμを現実化してくれた・世界観を現してくれた @HassakuTb さん
- 数学的内容の監修をしてくれた @yassu0320 さん

- サークル活動を手伝ってくれた方々
    - @blac_k_ey さん
    - @tadsan さん
    - @310_ktr さん
    - @Raimu0474 さん

- 校正・校閲をしてくれた方々
    - @dif_engine さん
    - @bra_cat_ket さん
    - @yagitch さん
    - @DKamogawa カワイシン（オルガノン・クラス） さん

- これを読んでくれた貴方！

せつラボ ～圏論の基本～

2019-04-14 初版 技術書典6
2019-09-22 第二版 技術書典7

原作 aiya000 (@public_ai000ya)
ヴィジュアル 碧はっさく (@HassakuTb)
発行所 しまや出版