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）

『そうだなぁ、ずっと一緒にいてくれる人が欲しいな』

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）

『えへへ。わたしはηが頑張ってるところ、大好きだから！』

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。

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。

3.1　せつめいするラボ

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。

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、思ったんだ。

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。

はは、よかった。

……

そんでさ、圏論って簡単なの？

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と思う。

ほーほー。

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！　いこう！

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**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

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

集合は、多くの数学分野の基礎。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。

へえー、そうなの？

うん。圏論もそうなんだ。

じゃあ、説明を始めるね。

集合は、**もの**の**集まり**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 = { △, ▲, ◿ }
```

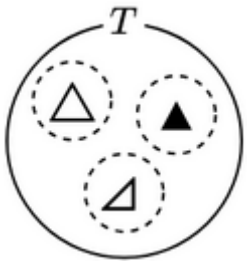
▲リスト4.1: 外延的記法での表現

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

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。

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。
```

▲リスト4.5: 自然数の集合ℕ(エヌ)

[*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。

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]

- 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）

[*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。

- 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。

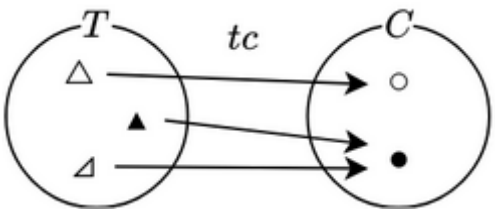
とーっても面白い事象なんだよ……ってね。

ふふ、そうなんだ。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

やじるしだ！

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。「Tの各元をCに写す」……という感じに。

うんうん。

言葉では、次のように定義できる。

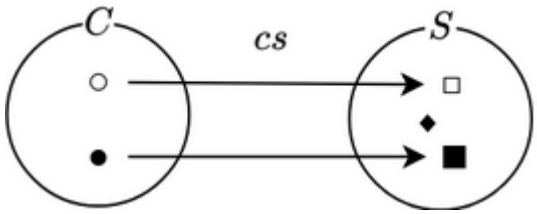
```
tc : T -> C
tc(△) = ○
tc(▲) = ●
tc(◿) = ●

または

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

▲リスト4.6: 写像tc : T -> C

へえ〜。写像、他にはどんなのがあるの？

いい質問だね。

丸から四角への写像を書いてみよう。



▲ 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**

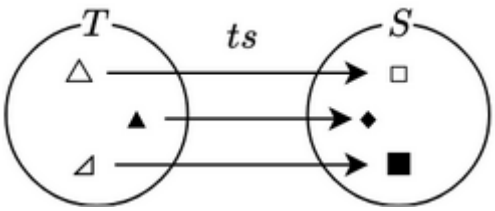
――と呼ぶ。

写像は、単射と全射にわけられるのかな。

おしい。

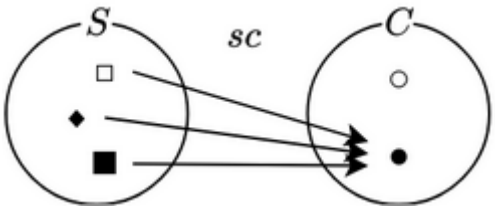
実は**全単射**という、全射でも単射でもある写像。そして全射でも単射でもない写像も考えられる。

次のtsは全単射。



▲ 図4.4: 写像ts : T -> S

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

ぜんぶで4パターン、あるんだね！

- 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　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。

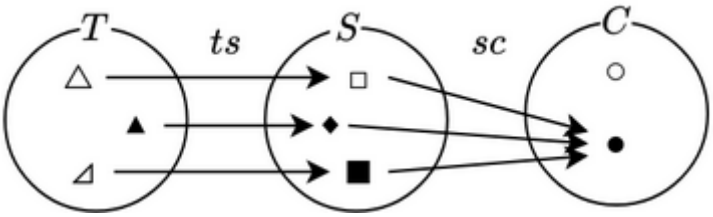
ある写像f，gに対して
　　f : X -> Y
　　g : Y -> Z
次のように定義される。

g ○ f : X -> Z
(g ○ f)(x) = g(f(x))

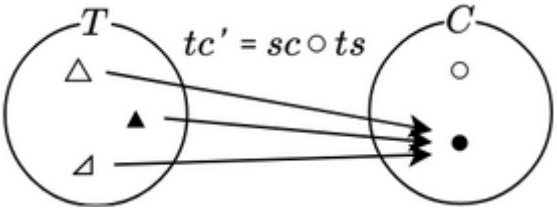
▲リスト4.8: 写像の合成○

さっき使った写像tsとscを合成して、写像tc'――ティーシープライム――を定義してみよう。

それはこのように書けるよ。



▲ 図4.6: 写像tsとscの合成イメージ



▲ 図4.7: 写像tc' : T -> C

```
tc' : T -> C
tc' = sc ○ ts

つまり

tc'(△)  =  (sc ○ ts)(△)   ―― tc' = sc ○ ts を展開
        =  sc(ts(△))       ―― (sc ○ ts)(x) = sc(ts(x)) を展開
        =  sc(□)           ―― ts(△) = □ を展開
        =  ●

tc'(▲) = (同じく展開) = ●
tc'(◿) = (同じく展開) = ●
```

▲リスト4.9: 写像tc' : T -> C

おー！　△を□に写したあとに、●に写すのか～。

そうそう。**写した元をまた写す**……っていうを各元にするのが、合成された写像だね。

そういえば、tcもTからCへの写像だったよね。tcとtc'はちがうものなの？

うん。tcが△を○に写すのに対して、tc'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。

たしかに、ほんとだ！

4.2　まとめ

これで集合の内容は終わり。お疲れ様！

おつかれさま、ありがと！

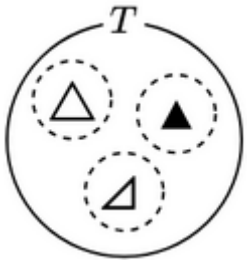
いいえ。

……理解したことを、まとめてみよう。

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4.8: 図での表現

うんうん。

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・10000。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
  - 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
  - 偶数
  - 奇数
- 非可算無限な集合
  - 実数
  - 実数のうち 0.0 〜 10.0

有限と可算無限・非可算無限だね。

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だった。

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

うんうん！

f : X -> Yとg : Y -> Zのような、Yで繋がる写像は、**合成**という概念が定義できる。つまり、合成○が、g ○ f : X -> Zを定義できる。

……以上。

ありがとう♪

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。

う、宇宙。ふふ、たのしそう……なのかな。

……

4.3　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。

4.3.1　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4.11: 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4.12: 集合の元に、順番はない

4.3.2 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。

1) T -> Tの具体的な定義を、いくつか考えてみて。

4.3.3 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。

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

▲リスト4.13: 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4.14*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4.14: 偶数から整数への全単射

◆ ◆ ◆

お疲れ様 :)

4.4 （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*

第5章　用語

さて次は、数学的な各用語を説明するよ。数学を歩くにあたって、最も基本的な各概念を。

あとは簡単に、Haskellというものの紹介。

よ、よろしくっ！

これらを理解することで、μは圏論に――十分に入門することができる。

……まずは、次の3つの用語を説明するね。

- 任意の○○
- ○○が存在する
- ○○が一意である

最後の「一意」？　以外はなんだか……普段も使いそうな言葉だね。

確かに、そうだね。

でもこれらには……数学特有の意味がついているんだ。普通の意味と混同すると、混乱しちゃうかも。

おー……。

数学のときに、数学の言葉を、適切に使えると――格好いいね。

5.1　任意

まずは「**任意**」から。

「任意の○○」っていうと、日常では「好きな○○を」とか……その人の意思に任せるような言葉だよね。

だね。

でも数学的には――**全て**の○○――という意味を持つ。むしろ、その人は意思を選べず、全ての○○について考えることになる。

……否応ない感じだ。

そんな圧迫した意味でもないかな……。

……「任意の」と「全ての」が、その時々によって使い分けられるけど――2つは同じ意味。

まずは先程おぼえた集合の知識で、用法の例を確認してみよう。

自然数の任意の元は、2で割り切れるか、割り切れない。

▲リスト5.1:「任意の」の用法

「自然数の全ての元は、2で割り切れるか、割り切れない。」……って意味だよね？　なんだか、あたりまえなことを、いってる？

うん、当たり前ではあるね。まあ言葉遊びだと思って。

ふふふ。

そして「任意の」「全ての」の意味を、数式で主張したいときは……∀という記号が用いられる。

これも用法を確認してみるよ。ちなみに集合ℕっていうのは、自然数の集合のことだったね。

```
∀n ∈ ℕ,
  n % 2 = 0 ∨
  n % 2 ≠ 0
```

▲リスト5.2: ∀の用法

x % yは、xをyで割り算した余り――を求める計算のこと。剰余算とも言うね。

ジョーヨさん……。

ぷっ。……こほん。

そしてa ∨ bは、「aもしくはb」っていう意味。

ちなみにa ∧ b、「aかつb」という記号もあるよ。

……さっきの数式を、一行ずつ読んでみると、こうなる。

```
∀n ∈ ℕ,          —— 任意の自然数nについて、
  n % 2 = 0 ∨    —— nは2で割り切れるか、もしくは
  n % 2 ≠ 0      —— 割り切れない
```

▲リスト5.3: ∀の用法――意味

おお。さっきといってること、おんなじだ！

その通り！

5.2　存在

次は「**存在する**」という言い回しについて。

これは……日常での意味と、あまり変わらない。

「ηはここに存在する」みたい、な？

……う、うん。僕もμがここに、ちゃんといてくれて……うれしい、よ。

えっ……わたしも……ありがとう、存在してくれて。

あっ、えっと……。こほん。

……今言ってくれたことと、数学的な意味の「存在」は、同じだね。

「○○が存在する」は、たまに「○○がある」とも言われるよ。

……えへへ。

数式で「存在する」を主張したいときは、∃という記号が用いられる。

書いてみよう。整数の集合をℤとするよ。[*1]

```
∃x ∈ ℤ,      —— 整数のうち、以下を満たすものが存在する
    x % 2 = 0  —— 「2で割り切れる」
```

▲リスト5.4: ∃の用法

あ、偶数のことかな。偶数は割り切れそう！

その通り。少なくとも、整数には偶数が**1つ以上ある**ことを、言っているよ。

じゃあ……「存在」については、こんなところまでかな。

……

……。

……。

5.3　一意

じゃあ、最後の用語。「○○は一意である」「一意な○○」について。

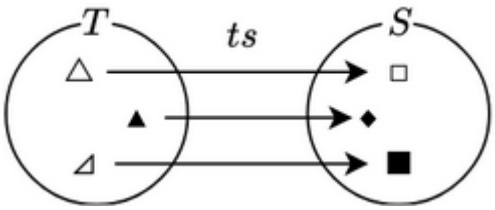
これはほとんどの場合、「存在」と一緒に使われて――その際に、**ただ1つだけ**存在することを強調する。

[*1] 整数のドイツ名 "Z"ahlen の "Z" だよ。

へえー。

全単射は――「**一方が決まれば、他方が1つだけ決まる**」っていう意味で――割り当てが一意だよ。

さっきも使った全単射tsを見てみよう。



▲ 図5.1: 全単射ts

このtsで、例えば△が決まれば、□が決まる。▲が決まれば、◆が決まる。

逆に□が決まれは、△が決まる。■が決まれば、◿が決まる。

つまり――「割り当ての一方が決まれば、もう他方も**一意に決まる**」。

数式では一意をヨ!と書く。書いてみよう。

写像ts : T -> Sが全単射だとすると

```
∀  t ∈ T,          ―― Tの任意の元tに対して、
∃! s ∈ S,          ―― 以下を満たす、Sの元sが一意に存在する。
   ⇒  ts(t) = s    ―― 「ts(t) と s が等しい」
```

つまりtsは、Tの任意の元tに対して、あるSの元sを一意的に割り当てる。

▲リスト5.5: ヨ!の用法

a ⇒ bは「aであれば、bである」と読むよ。

ふんふん、なるほどね〜。

ηが1人しかいない……ηは人類の中で一意……ってことだね。

そ、そうなるね。僕であれば、μが……一意に決まる……よ。

えへへ！

5.4　Haskell

じゃあ終わりに軽く、Haskellについて紹介するよ。

この話は少し難しいし、圏論には少ししか関わってこないので……頭を休めて、聞いてもらえればと思う。

おっけー！……ふぅ。

Haskell……プログラミング言語？　だっけ。

そう、Haskellはプログラミング言語の一つだよ。プログラミング言語っていうのは……機械に物事を頼むときに使う、便利な言葉。

今回は最も重要な――記法周りについてを、説明するね。

おねがいします。

5.4.1　型と値、そして関数

「集合と元」「写像」という概念があったよね。

うん、あったあった。さっきやったね。

Haskellにも、集合と似た――**型**（かた）と**値**（あたい）、**関数**（かんすう）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、定義できる。

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

-- 集合での記法↓
-- T = { △, ▲, ◿ }
```

▲リスト5.6: 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, Dottedを値という。

x ∈ Tっぽい書き方が提供されてるよ。

```
-- △ ∈ T
Unfilled :: T

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

▲リスト5.7: Tの値

typeというキーワードで、型の別名をつけることもできる。

```
type Triangle = T

Unfilled :: Triangle
[Unfilled, Filled, Dotted] :: [Triangle]
```

▲リスト5.8: 型Triangle = T

Unfilled, Filled, Dotted……？

表現上の理由で、「△▲◿」みたいな記号は使えない。だから代わりに、英名を付けたんだ。

……あと、Bool・Char・Int・String・(X, Y)・Either X Yっていう、特別な型が、予め定義されているよ。

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

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

-- 整数を表すInt *2
[-10000 .. 10000] :: [Int]

-- 全ての文字列を表すString
["mu", "eta", "enjoy", "math"] :: [String]

    -- XかつYを表す(X, Y)。例えば「IntかつChar」
[(-10, 'a'), (10, 'z')] :: [(Int, Char)]

-- XもしくはYを表すEither X Y。例えば「IntもしくはChar」
[Left -10, Right 'a'] :: [Either Int Char]
```

▲リスト5.9: 型Bool, Char, Int, String

[*2] 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。

そして関数は、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
```

▲リスト5.10: 関数isZero

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5.11: 関数isZero

値を関数に与えるときは、次のように書く。

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

▲リスト5.12: 値を関数に与える（関数適用）

かっこがないんだ。

そう。Haskellでは、括弧を付けずに、値を与えるよ。
セクション記法という、関数適用の特殊な形もある。

```
(* 3) 10
-- ↑
-- (* 3)は、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 * 3

(1 +) 5
--↑
-- (1 + x)のxに5を入れて、計算するイメージ。
-- 1 + 5

(+ 10) 2    {-  2 + 10  -}
(- 1) 9     {-  9 - 1   -}
(- 1) 9     {-  9 - 1   -}
```

▲リスト5.13: セクション記法と関数適用

へぇ〜。Haskellは、集合を扱うものなの？

率直なところ、そうだね。集合……もとい型によって、命令を記述する言語だと、僕は考えているよ。

へぇ〜。

5.4.2　型クラス

型クラスというもので、指定された型への、ある種の性質を。型クラスの**インスタンス**というもので、型がその性質を持つことを――定義できるよ。

例えば次の型クラスShowは、指定された型aが、文字列として表せることを。インスタンスShow Intは、IntがShowという性質を持つことを――定義している。

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

instance Show Int where
  show = {- 実装 -}
```

▲リスト5.14: 型クラスとインスタンスの例

他には、「要素に関数を適用できる」ことを表す、Functorというのもある。これは圏論と関連があるので、紹介しておくよ。

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

-- 普通の値、もしくは「無い」を表すMaybe
instance Functor Maybe where
  fmap = {- 実装 -}

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

-- 複数の値（値のリスト）を表す[]
instance function [a] where
  fmap = {- 実装 -}
```

▲リスト5.15: Functorとインスタンス

ふぅん……？

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。

5.4.3 ghci

[*3] 表示される文字は、ghciの設定によって違うよ。

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という……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。*3

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

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

>>> show 1
"1"
>>> show 2
"2"
>>> show 10
"10"

>>> fmap (+ 1) (Just 10)  -- 10 + 1
Just 11
>>> fmap (+ 1) Nothing  -- 「無い」に(+ 1)しても、それは「無い」
Nothing
>>> fmap (+ 1) (Left "no")
Left "no"
>>> fmap (+ 1) (Right 10)
Right 11
>>> fmap (+ 1) [10, 20, 30]  -- 全ての値に(+ 1)
[11, 21, 31]
```

▲リスト5.16: ghci

Haskellについては、以上だよ。

5.5　まとめ

これでμは基本的な、数学的な会話ができるようになった。また、数学分野の具体例として、集合とHaskellを得た。

数学への入門を果たして、必要な装備も揃ったね。

わ。ワクワクするね！

どうかそのワクワクをずっと忘れずに、数学を歩いていって欲しい。

……今回のことを、まとめてみよう。

おねがいします♪

数学に必要な3つの用語。任意・存在・一意。

まず「任意の○○」と言ったときには「全ての○○」のことを表す。

次に「○○が存在する」と言ったときには……特にひねりもなく、その通りの意味を表す。

そして「一意」は「存在」と一緒に使われて、その存在がただ一つだけあることを強調する。

```
―― （ある集合Sと、その全ての元）
Sの任意の元。

―― （ある集合Sに、特別な元sがあること）
Sに元sが存在する。

―― （ある集合Sに、特別な元sがただ一つあること）
Sに元sが、一意に存在する。
```

▲リスト5.17:「任意・存在・一意」の用法

任意の人類の中で、ηは存在して一意♪

はは……μも、ね。

♪

5.6　ノートの切れ端B

以下を、数式で書いてみて。

1) 偶数の集合Evenの、任意の元xは、2で割り切れる。
2) 奇数の集合Oddに、2で割り切れない、元xが存在する。
3) n * 2 = nを満たす、自然数のある元nが、ただ一つだけ存在する。

- また、そのようなnとは何だろう？

μはとうとう、数学に足を踏み入れた。これからどんな道を進んでいくのか、楽しみだね :D

5.7　（本章での参考文献）

- 整数 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/整数*
- 一意性(いちいせい)とは - コトバンク: *https://kotobank.jp/word/一意性-31214*
- 数学記号の表 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/数学記号の表*

第6章　休憩

◆ ◆ ◆

んっ。

日差しが強くなってきた。あの寒かった冬空の空気も消えて、温かい。
この風景は、あの夢を思い出す。

ちょっと疲れた？

μを見る。目に入るのは、きれいな髪、ほさき。

……。

顔が近くて、目をそらす。
……少しだけ、深呼吸。

　そうだ、そろそろお昼ご飯にしよっか。ドーナッツとチーズタルトでもどう？　お母さんがくれたコーヒー豆も、まだあったよね〜。

とてとて、とてとて。

……

はい、コーヒー。砂糖多めのミルク無し。

僕の、いつも通りの飲み方。
コトン、とカップを置く音。2人きりの静かな空間。

いつもありがとう、μ。

ふふ。いいえ〜、だよ！

……ほっとしたから……

μは、いいお嫁さんになるね。

ぽろっと、恥ずかしい言葉が出た。

……えへへ、ありがとう。そう言ってくれて、うれしい。

じゃあηはわたしを……お嫁さんにしてくれる？

μが少しだけ僕に近づいて、そう言う。

……えと、その……

ドキ……ドキ。ドキ……ドキ。

……

ふふふ、冗談だよ。

（ほっ……。）

……

今は、ね。

――ビシャ。ああ、コーヒーをこぼしてしまった。
そんな言葉を付け足たすのは、ずるいと思う。

それなのになんだか……うれしい気持ちになった。

◆　◆　◆

第7章　圏

7.1　圏 = 対象と射の集まり

じゃあそろそろ、圏論へ入門しようか。

μはこれから……圏論を介して、数学に共通した性質について、触れていくことになる。その知識は、数学を歩くための強い武器になるよ。

わーあ、楽しみ！

そうしたらまずは、**圏**(けん)というものの説明から始めるね。圏論は、この圏を用いて、その様々な性質を述べる学問になるよ。

ほうほう……。

まずは次の、圏の定義を、眺めてみて欲しい。

まだ理解できなくても、大丈夫だから……眺めてみて。

圏Cとは、法則①②を満たす――ob(C)・ar(C)・「○」の組のことである。

ar(C)とは、**射**の集まりである。
射とは、次のような形式で書かれる――あるXからあるYへの割り当てのこと。

```
f : X -> Y
```

ob(C)とは、**対象**の集まりである。
対象とは、ar(C)の任意の射fに対する、":" の後であって "->" の前後にある**もの**のこと。

```
f : X -> Y
```

例えばこのfに対して、対象X・対象Yという。

また、"->"の前側の対象を**ドメイン**。後側の対象を**コドメイン**という。

「○」とは、射の**合成**のことである。
射の合成とは、以下のようなドメインとコドメインが同じ対象である、
圏Cの任意の射f, gに対して g ○ f を定めるものである。

```
∀ f : X -> Y,   g : Y -> Z
    ⇒  g ○ f : X -> Z
```

（法則①②については後で。）

▲リスト7.1: 圏の定義

圏の名前はDだったAだったり、あるいはFooだったりもするよ。圏による。

だからもし圏の名前がFooだったら、ob(Foo)とか、ar(Foo)とか……って書くことになるね。

もちろん射も、fという名前じゃなくてもいい。

ふむふむ。なまえは適当でいいんだね。

うん、なんでもいいんだ。

……組？　ob？　ar？

なんらかのAとBに対して「AとBの**組**」っていうのは……AとBの塊が1つの何かを表すよ、っていう意味の強調。組〈A, B〉って書いたりもするね。

obは対象の英語名、objectの略。arは射の英語名、arrowの略だよ。

圏Cは〈ob(C), ar(C), ○〉のことである……っていう感じかな。

その通りだよ。

えへへ！

――そして、法則①②について。

①
Cの任意の対象Xに対して、**恒等射**id_Xが存在する。

```
∀X ∈ ob(C)   ⇒   idX : X -> X ∈ ar(C)
```

恒等射id_Xとは A -> X と X -> B の任意の射に対しての、以下が成り立つ射のこと。

```
∀  f : A -> X,   g : X -> B
   ⇒  idX ○ f  =  f  ∧
      g ○ idX  =  g
```

②
射の合成○は、任意の射f, g, hに対しての、以下の式を満たす。

```
∀  f : W -> X,   g : X -> Y,   h : Y -> Z
   ⇒  (h ○ g) ○ f  =  h ○ (g ○ f)
```

①を圏の**恒等律**
②を圏の**結合律**　という。

▲リスト7.2: 圏の法則

うーん。AとかXとか、id_Xとかfとか……って、なんのことなんだろう？

感覚がいまいち、わからないっていうか……。

圏は抽象的だから……最初は誰もがそうなってしまうね。

えへへ〜……うーん。

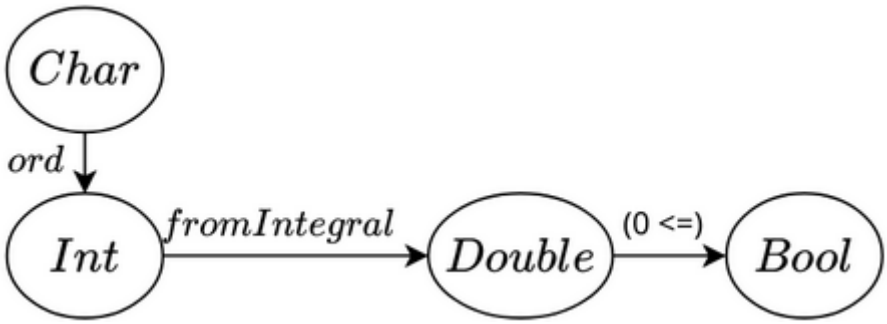
7.2　圏の具体例

7.2.1　Haskellの型と関数の圏、Hask

[*1] ここで「Haskell」とは特別に、遅延評価でなく、エラーも無限再帰も起こらないものと見做したものを指すよ。本当はこれらの性質が、Haskellが圏になることを阻害してしまうんだ。

そうしたら、圏の具体例の――**図式**で見てみよう。

Haskellの型と関数……の、ある特殊な場合……が圏になるんだ。*1



▲ 図7.1: 圏Hask

Haskellの型と関数の圏を、圏Hask――を図式にしてみた。*2

この丸*3で書かれたものが対象、矢印が射だよ。

ふふ、お団子みたいかも。

ははは、かわいらしい表現だね。

……Haskを、言葉でも定義してみよう。

```
圏Hask   =  〈ob(Hask), ar(Hask), (.)〉
ob(Hask) = {  Char,  Int,  Double,  Bool  }
ar(Hask) = {  ord : Char -> Int,    fromIntegral : Int -> Double,
                                        (0 <=) : Double -> Bool  }
```

▲リスト7.3: 圏Haskの定義

[*2] ここでの「圏Hask」は、Haskellの一部の型と、一部の関数――の圏とするよ。局所的な意味合いのない、広い意味での圏Haskは、Haskellの**全ての型**と**全ての関数**を含んだ圏のことをいうんだ。

[*3] 対象は書き方によっては四角だったり、そもそも名前を囲まなかったりするよ。

圏Haskの対象と射を定義できたところで……次はそれらが何なのか、ってところだね。

うん……対象と射が、いまいちなんなのかわからなくって。

対象とは単純に、射の矢印"->"の前側と後側の、何らかの「もの」のことだよ。「もの」が何なのかっていうのは、射が決める。

例えばord : Char -> Intの前側にはCharが、後側にはIntがあるから、CharとIntは対象だね。

対象は矢印の前側と後側。CharやIntが対象。

この前側を**ドメイン**・始域。後側を**コドメイン**・終域と呼ぶ。

これらはdom・codとも書くことがあって、例えばdom(ord)はCharのことを、cod(ord)はIntのことを指すよ。

ドム……コッド……？　むむむ。

domainのdom、**cod**omainのcodだね。

端的に言えば、こんな感じだよ。

```
任意の射f : X -> Yに対して
    dom(f) = X
    cod(f) = Y
と定義される。
```

▲リスト7.4: ドメイン・コドメイン

例えばordを例にすると、こう。

```
ord : Char -> Int
dom(ord) = Char
cod(ord) = Int
```

▲リスト7.5: ordのドメイン・コドメイン

ドメインは前側、コドメインは後側。ドメインとコドメインが対象。……うん、覚えた！

よしよし、順調だね。

次に、じゃあその「射」がなんなのか。これはちょっと抽象的な物言いになっちゃうんだけど……

射とは――「もの」から「もの」への割り当てであって、かつ法則①②を満たすもののことなんだ。

もの……？　あ、対象のことだったね。

そう。

例えばHaskなら、射は関数だったね。そして関数は型から型への割り当てなので――対象とは型のことになる。

お！　対象がなんなのかって、そうやってきまるんだ。

そうそう。じゃあもう一度、元の流れに戻って

……法則①、**恒等律**から見てみよう。まずHaskで射の合成は、関数合成と呼ばれる、次の関数(.)になる。

```
-- 関数合成関数(.)
(.) : (b -> c) -> (a -> b) -> (a -> c)
t . u = \x -> t (u x)
```

▲リスト7.6: 関数合成(.)の定義

なのでHaskに特化して、結合律を考えると、次のように言える。

```
任意の型Xに対して恒等関数idXが存在する。
    ∀ X ∈ ob(Hask)   ⇒   ∃ idX : X -> X ∈ ar(Hask)

つまり関数idXと、任意の A -> X と X -> B の関数に対しての、以下が成り立つ。
    ∀  f : A -> X,   g : X -> B
       ⇒ idX . f  =  f   ∧
          g . idX  =  g
```

▲リスト7.7: Haskの恒等律

まあ、最後の2つが成り立つのは、わかりやすいと思う。どうかな。

え、えーと……うん。idを合成しても、f, gの返す結果は変わらないもんね。……だよね？

うん、その通り。

もっと自信を持っていいと思うよ。もし間違ってたら、僕が訂正してあげるから。

えへへ、ありがと。

ちなみにドメインとコドメインが同じ射の一般を**自己射**って言うんだけど――恒等射でない自己射があることに注意だね。

例えば`toUpper : Char -> Char`がちょうど、自己射だけど、恒等射じゃない。

そっかぁ、ドメインとコドメインが同じでも、恒等射だとは限らないんだ。

そうなんだよね、恒等射はそう、多くあるものじゃないから。

次は**結合律**。結合律はこういうものだよ。

関数合成(.)は、任意の関数f, g, hに対しての、以下の式を満たす。

```
∀ f : W -> X,  g : X -> Y,  h : Y -> Z
    ⇒  (h . g) . f  =  h . (g . f)
```

▲リスト7.8: Haskの結合律

う、うん。確かに、関数合成に括弧は影響なさそう。

次のように置き換えていくと、結合律を満たしていることが示せるよ。*リスト7.6*を見ながらだと、わかりやすいね。

```
—— 結合律の証明
(h . g) . f
      括弧の中を展開
    = (\x -> h (g x)) . f
      (. f)を展開
    = \y -> (\x -> h (g x)) (f y)
      xに(f y)を代入
    = \y -> h (g (f y))

h . (g . f)
      括弧の中を展開
    = h . (\x -> g (f x))
      (h .)を展開
    = \y -> h ((\x -> g (f x)) y)
      xにyを代入
    = \y -> h (g (f y))

つまり

(h . g) . f  =  h . (g . f)
```

▲リスト7.9: Haskの結合律の証明

へぇ〜。……ほぇ？？

証明を見るときは、慣れるまでは——軽い姿勢でいいと思うよ。ここでは「(.)を置き換えていくと、証明ができる」程度に思って貰えれば大丈夫。

なるほど。Haskの関数と(.)で、結合律がみたされてるんだ。

ありがとう、やさしいね。

……。

7.2.2　集合と写像の圏、Sets

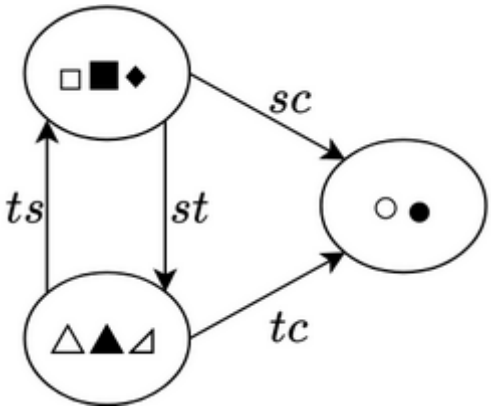
うう〜ん、たしかにHaskが圏になってるのはわかった気がするけど、それでもまだわからないかも。圏ってなんなんだろう……って。

それはきっと……特定の圏しか見ていないからだと、思う。じゃあ一度Haskを離れて、他の圏を見にいってみようか。

Hask以外の圏って、いっぱいあるの？

もちろん！　圏の多さこそが、その強みなんだ。

集合と写像の圏を見てみよう。この三角・四角・円の集合と各写像が、ちょうど圏になるよ。



▲ 図7.2: 圏Sets

写像。集合の要素から集合の要素への割り当て……だったよね。

これはさっき、集合をおしえてもらったときに、つかったやつ？

そうそう。この圏Setsも、ちゃんと法則を満たしているんだ。*4

Haskでは関数合成が、これらを満たしたね。Setsでは写像の合成が、これらを満たすよ。

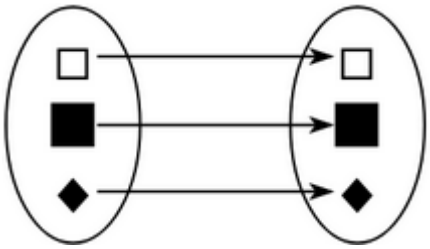
確認してみよう。

任意の集合Xに対して恒等写像idXが存在する。
　∀ X ∈ ob(Sets)
　　⇒　∃ idX : X -> X ∈ ar(Sets)

つまり写像idXと、任意の A -> X と X -> B の写像に対しての、以下が成り立つ。
　∀ f : A -> X,　g : X -> B
　　⇒　idX ○ f　　=　f　∧
　　　　g　　○ idX　=　g

▲リスト7.10: Setsの恒等律

任意の集合について、恒等写像は考えられる。例えば四角の集合についてはこんな感じだね。



▲図7.3: 集合の恒等写像

うんうん。

そしてそれはもちろん、f, gのような写像が行う、各要素の割り当てを、変更しない。

だから恒等律は成り立つ。

[*4] Hask同様に、一般的に「圏Sets」というと「全ての集合と全ての写像からなる圏」を指すことが多い。ここでのみ圏Setsといったときに、三角・四角・円の集合と各写像の圏を指すとするよ。

そう、同じ感じ。圏が多くを一般化していることの、良い現れだね。

次は結合律。

写像の合成○は、任意の写像f，g，hに対しての、以下の式を満たす。

```
∀  f : W -> X,   g : X -> Y,   h : Y -> Z
    ⇒  (h ○ g) ○ f  =  h ○ (g ○ f)
```

▲リスト7.11: Setsの結合律

これはHaskと同様に、写像合成の定義を

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7.12: 写像合成○の定義

次のように置き換えていくと、示すことができるよ。

```
((h ○ g) ○ f)(x)  =  (h ○ g)(f(x))      —— 右側の○ を 展開
                  =  h(g(f(x)))         —— 残りの○ を 展開
                  =  h((g ○ f)(x))      —— 置き換え
                  =  (h ○ (g ○ f))(x)
```

▲リスト7.13: Setsの結合律の証明

おお〜、Haskとおんなじだ。

型と集合って、とっても似てるしね。

えへへ。

でも……恒等律と結合律って、なんでそんなかたちをしてるの？

それはね――**数学に頻出するパターン**を、まとめるため、だよ。

今見てきたHaskの関数や、集合の写像。それと**似通ったもの**が、数学には多くある。

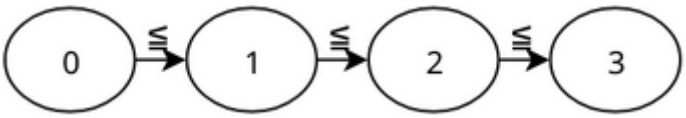
それを**まとめる**のが――圏論、だから。

数学をまとめるのが……圏論。

そっか、圏。なんだかちょっとだけ、わかってきたかも！

そう、よかった。

……最後のダメ押しとして、自然数と順序の圏を上げておこう。



▲ 図7.4: 自然数と順序の圏

任意の自然数についてx ≦ xとなるので、順序関係≦が恒等射にもなっているんだ。

へぇ～、面白いね。順序関係、なんだか関数と写像とかとはイメージ違うけど、ちゃんと射になるんだ～。

圏論の柔軟性だね。

7.3　まとめ

それじゃあ圏の感覚がわかってきたところで最後に、特定の圏によらない、圏の一般の言葉でまとめてみよう。

よろしくおねがいします。

圏は**対象**と**射**からなるもののこと。

また対象は射の前側と後側についている「もの」、**ドメイン**と**コドメイン**のこと。

そして射は**合成**というものを持ち、**恒等律**と**結合律**、という法則を満たすもののことを言う。

圏Cの恒等律は、次の通りに定義される。

任意の対象Xに対して恒等射idXが存在する。

```
∀ X ∈ ob(C)
    ⇒  idX : X -> X  ∈  ar(C)
```

つまり射idXと、任意の A -> X と X -> B の射に対しての、以下が成り立つ。

```
∀ f : A -> X,   g : X -> B
    ⇒  idX ○ f    =  f  ∧
       g   ○ idX  =  g
```

▲リスト7.14: 一般の圏の恒等律

結合律は、次のように定義される。

射の合成○は、任意の射f, g, hに対しての、以下の式を満たす。

```
∀  f : W -> X,   g : X -> Y,   h : Y -> Z
    ⇒  (h ○ g) ○ f  =  h ○ (g ○ f)
```

▲リスト7.15: 一般の圏の結合律

これらを満たすもの、圏になるものとしては、これらが挙げられる。

- Haskellの型と関数（ただしいくつかの特性を除いたもの[*1]）
- 集合と写像
- 自然数と順序≦

わー、なんか理解できた気がする〜。

よかった。まずは、理解できた気がする――という状態に、到達するのが大事。

まだわからないところがあったら、ゆっくりゆっくり……理解できた気になるまで、咀嚼してみるのがいいね。

ふふ、ありがとう。η、やさしい。

はは……ありがと。

7.4　ノートの切れ端C

適当な圏Cの対象X,Y ∈ ob(C)への、X -> Yを持つ射は、必ずしも一つとは限らない。
例えばHaskなら、以下2つのChar -> Boolが考えられる。

```
―― 文字が'a' .. 'z'の間にあるなら、Trueを返す
isLower : Char -> Bool

―― 文字が'A' .. 'Z'の間にあるなら、Trueを返す
isUpper : Char -> Bool

isLower ≠ isUpper
```

▲リスト7.16: Char -> Boolは1つだけじゃない

任意のX -> Yは、集合として考えられるってことだね。

Char -> Boolの集合{ isLower, isUpper }のように。

興味があったら、以下の問題も、説いてみて :)

任意の集合は、**離散圏**と呼ばれる――元と、元の恒等射のみからなる圏とみなすことができる。つまり集合Sの、異なる対象x,y ∈ ob(S)の間に、一切の射がない圏。

1) **適当な**集合Sを取って、その離散圏が各法則を満たしていることを、確認してみて。
2) 例1: { △, ▲, ⊿ }
3) 例2: { ★ }

適当 = 適切

「適当な」という数学用語は、「**適切な**」という意味だよ。

例えば「適当な圏C」と言ったら、「確かに圏の法則を満たしている、圏C」。

その他。

1) 日常で目にうつった様々なものが、何かしらの圏の対象、もしくは射だったりしないか。思いを馳せてみて。
2) 例:「積まれた本のうち、一番下に積まれていたものを、一番上に置き直す」は、何かしらの射になっている？　じゃあ対象は？

7.5　（本章での参考文献）

- 圏論 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/圏論*
- 圏 (数学) - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/圏_(数学)*
- 射 (圏論) - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/射_(圏論)*
- 写像の合成 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/写像の合成*

第8章　様々な対象と射

今度は対象と射を使った、いくつかの概念を見ていくね。

よろしくおねがいします。

よろしくね。

今からやろうと思う各概念は、圏論の内外で広く使われているものなんだ。

圏論を自由に歩くための、大きな手助けになってくれるはずだよ。

ふふ、たのしそう。

とても、楽しいよ。それじゃあ、始めようか。

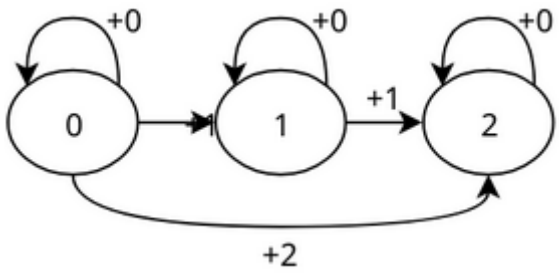
8.1　可換図式

度々あつかっている、圏を表す丸と矢印の図。あれには**図式**という名前が付いているんだ。

CharやIntとかの図や、三角や四角とかの図、だね。

うん。覚えてくれてて、ありがとう。

まずはこの図を見て欲しい。



▲ 図8.1: 0から2と足し算の圏

たすいちと、たすに？

そう、足し算。

ドメイン「0」からコドメイン「2」への、射の道が、複数あるよ。わかるかな。

え〜と……「0」から+1と+1をしたときと、+2をしたときかな。

うん、いい回答だね。

もっと細かくいうなら――圏には恒等射があるので実は、例は無限に考えられるよ。

```
(+2)
(+1) ○ (+1)
(+0) ○ (+1) ○ (+1)
(+0) ○ (+1) ○ (+0) ○ (+1)
(+0) ○ (+0) ○ (+1) ○ (+0) ○ (+1)
...
```

▲リスト8.1:「0 -> 2」の各射

無限に。

この圏には――射のたどり方が同じであれば、その全ての合成が等しくなる――という特徴がある。

つまり、こういうこと。

```
(+2)  =  (+1) ○ (+1)
      =  (+0) ○ (+1) ○ (+1)
      =  (+0) ○ (+1) ○ (+0) ○ (+1)
      =  (+0) ○ (+0) ○ (+1) ○ (+0) ○ (+1)
      =  ...
```

▲リスト8.2:「0 -> 2」の全ての射が等しいこと

おお、ほんとだ。

1を足して、また1を足す。0を足したあとに、1を足して、また1を足す。2を足す。2を足したあとに、0を足す。

……とかやっても、等しいもんね！

その通り！

実は*図8.1*の、この性質は――「可換図式」の定義そのもの。

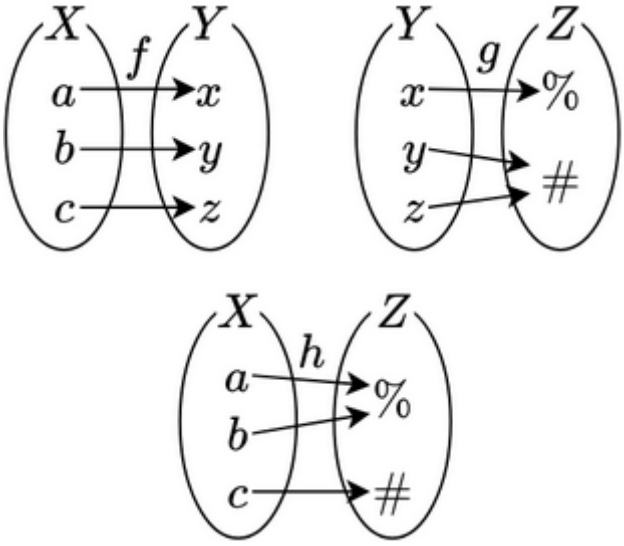
へぇ～！……？

「可換図式」とは、あるいは**「図式が可換である」**とは――その圏の、@「ドメインとコドメインが同じ全ての射が、等しいこと」なんだ。

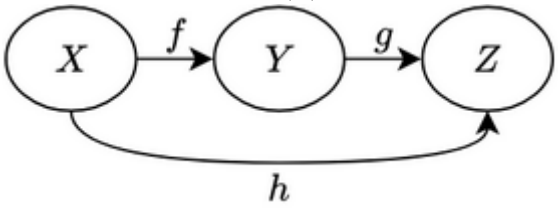
*リスト8.1*のようにね。

おおー、なるほど！……うーん？

そうだな、次は可換でない図式を見てみようか。次の図式は、可換じゃない。



▲ 図8.2: 集合X, Y, Zと各写像

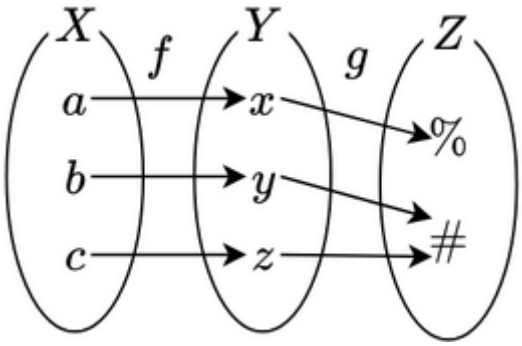


▲ 図8.3: 集合X, Y, Zと各写像の圏

可換じゃないの？

うん。g(y) = #とh(b) = %に注目して欲しい。

このg ○ fはこういう写像になるけど



▲ 図8.4: 写像g ○ f

これはhと等しくない。

あ！　g ○ fではbが#に割りあたってるけど、hでは%に割りあたってる？

あたり！

やった〜！

8.1.1　小まとめ

丸と矢印からなる圏の図を、**図式**という。

同じ「ドメインとコドメイン」を持つ全ての射が等しい場合、その図式を**可換図式**という。

「可換図式」っていわれたら、全ての射が等しい。おぼえた！

8.2　対象の直積

8.2.1　直積の定義

じゃあ次は、**対象の直積**について、説明するよ。

積は……かけ算のことだっけ。

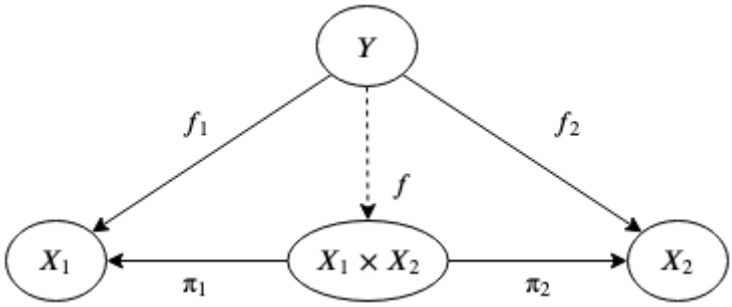
そうだね。でも今回の「直積」は、必ずしも積と一致するものじゃないんだ。紛らわしいけどね。

へぇ〜。

そしたらまずは、圏一般の、対象の直積の定義を眺めてみよう。

……

ある対象X_1とX_2の、**2対象の**直積とは――任意の対象Yと、任意の射f_1・f_2があったとき、次のような図式を可換にする対象$X_1 \times X_2$と、それをX_1とX_2に分ける射π_1・π_2――の組$\langle X_1 \times X_2, \pi_1, \pi_2 \rangle$のことをいうよ。



▲ 図8.5: 直積〈X_1×X_2, π_1, π_2〉

おお、さっそく可換図式だね。

可換図式は便利だからね。至るところで使われるよ。

言葉でも、直積を定義しておこう。

以下の命題①②③が真である、
組〈$X_1 \times X_2$，π_1，π_2〉が、ある対象X_1とX_2の直積である。

- - -

ある対象Yと、以下の射があったときに
　∀ Y,　f_1 : Y -> X_1,　f_2 : Y -> X_2

以下の等式を満たす。
　① f_1 = π_1 ○ f
　② f_2 = π_2 ○ f

また　③ 射fは一意に定まる。

▲リスト8.3: 直積〈$X_1 \times X_2$，π_1，π_2〉の定義

ちなみにπ_1とπ_2のことを指してるのが、明らかにわかるときは省略して――雑に「直積X_1×X_2」とか「X_1×X_2は直積」とか言ったりもするよ。

図式の……かける？　矢印がひとつ、てんてんになってるね。

A×Bっていうのは、大体の場合は「対象AとBの直積」を意味するんだけど……ここでは単に、$X_1 \times X_2$っていう、ひとつの対象の名前として見てくれればいいかな。

えーと。$X_1 \times X_2$も、AとかBとか、FooとかBarとか……みたいな、単なる名前でしかない……ってこと？

その通り！

ふふん。

あとは、射の点線の意味するところだけど――この射fは、そのf_1とf_2に対して、一意的に定まる射だ。っていう意味だよ。

逆に言うと、そのようなfは1つだけしかない、ってことだね。

またfのような射を**直積の仲介射**（ちゅうかいしゃ）と、π_1とπ_2のような射を**射影**という。

えーと、あわわわ……。

おっとっと、ごめんね。眺めるのはここまでにしよう。

8.2.2　圏Haskの対象の直積

次は、今眺めた直積の定義を――Haskの(X_1, X_2)・fst・sndが満たすことを確認してみよう。*1

任意の対象X_1とX_2に対する、Haskの対象(X_1, X_2)が、その直積であること――を。

まずはX_1とX_2をIntとBoolに固定して、(Int, Bool)がその直積であることを考えてみるよ。

こんな補助関数も書いておこうか。Haskellの標準ライブラリにある関数と、同等なものだね。

```
(&&&) :: (a -> b) -> (a -> c) -> a -> (b, c)
f &&& g = \x -> (f x, g x)

fst :: (a, b) -> a
fst (x, _) = x

snd :: (a, b) -> b
snd (_, y) = y
```

▲リスト8.4: 補助関数(&&&), fst, snd

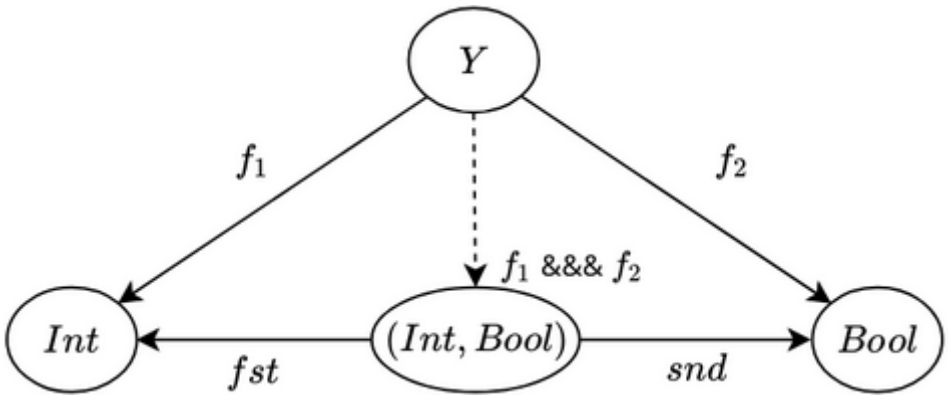
[*1] 前述の通り、直積型も対象の直積になるんだ。それ以外にも、直積の定義を満たすものはある。ここでは一例として、(X_1, X_2)とfst・sndを上げておくよ。

```
>>> import Data.Char (ord, isSpace)

>>> ord &&& isSpace $ ' '
(32,True)
>>> ord &&& isSpace $ 'a'
(97,False)

>>> fst (32, True)
32
>>> snd (32, True)
True
```

▲リスト8.5: &&&, fst, sndの動作確認

余談だけど、fstとsnd、ordとisSpaceも、標準ライブラリにある関数なんだ。

……じゃあ、さっきの可換図式をHaskで書き直してみるよ。



▲ 図8.6: 直積(Int, Bool)

Oh...

また頭が溢れてしまう前に……この可換図式の意味する、大事なところを、まとめておこうね。

以下の命題①②③が真であるので、
組〈(Int, Bool), fst, snd〉は、対象IntとBoolの直積である。

- - -

任意の対象Yと、任意の射f_1, f_2があったときに
　　∀ Y.　f_1 : Y -> X_1,　f_2 : Y -> X_2

以下の等式を満たす。
　　① f_1 = fst . (f_1 &&& f_2)
　　② f_2 = snd . (f_1 &&& f_2)

③
　　①②を満たすような射
　　　f : Char -> (Int, Bool)
　　は
　　　f_1 &&& f_2　以外にない。

　　つまり、一意に　f = f_1 &&& f_2　である。

▲リスト8.6: 〈(Int, Bool), fst, snd〉が直積であるとは

8.2.3　対象(Int, Bool)が直積であること

さて、この①②③を確認できれば、(Int, Bool)がIntとBoolの直積であることがわかる。

まずは任意のY・f_1・f_2の一例として、それぞれChar・ord・isSpaceを当てはめてみて、それから①②③を考えてみよう。

```
f1 :: Char -> Int
f1 = ord

f2 :: Char -> Bool
f2 = isSpace
```

▲リスト8.7: Y, f_1, f_2に当てはめてみる

うん！

①について。

この条件関数areCommutativeViaFstが、任意のCharの値に対してTrueを返す、という形で確認できる。

```
areCommutativeViaFst :: (Char -> Int) -> (Char -> Bool) -> Char -> Bool
areCommutativeViaFst f1 f2 x =
  (fst . (f1 &&& f2)) x
    == f1 x
```

▲リスト8.8: f_1 = fst . (f_1 &&& f_2)――の確認

```
>>> areCommutativeViaFst ord isSpace ' '
True
>>> areCommutativeViaFst ord isSpace 'a'
True
>>> areCommutativeViaFst ord isSpace 'b'
True

...
```

▲リスト8.9: f_1 と fst . (f_1 &&& f_2) が 可換であること

おお～、Trueになるね～。

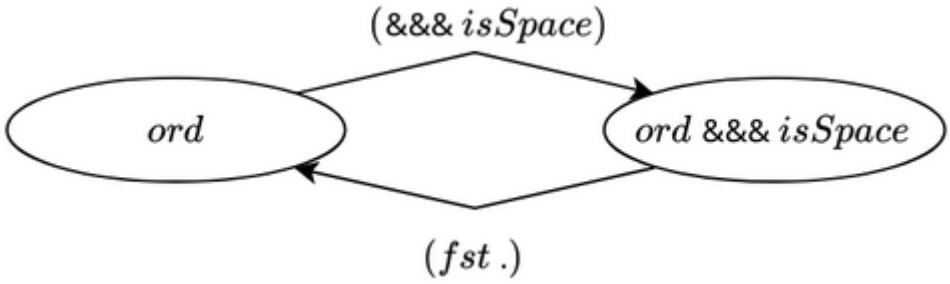
これは直感的には――

ord &&& isSpaceは値xをord xとisSpace xに分解する関数。

そしてfstはord xとisSpace xのうちord xだけを取得する関数。

つまりisSpace xを余分に作ってるだけ。なのでその合成は、単にord xするだけなのと同じ。

――と言える。



▲図8.7: (&&& isSpace) と (fst .)

そっか、そうだよね。単純に……&&&とfstは打ち消し合いそうだもん。

そうそう、そういう感じ。打ち消すっていうのは……うん、今の図で、次のようになるってことだね。

```
(fst .) . (&&& isSpace)  =  \ord -> fst . (ord &&& isSpace)
                         =  \ord -> ord
                         =  ordの恒等関数
```

▲リスト8.10: (fst .) と (&&& isSpace) が 打ち消し合うこと

②について。これは①と同様に、(ord &&&)と(snd .)の合成がisSpaceの恒等関数になるよ。

次のような関数を使えば、fstと同様に確認できるね。

```
areCommutativeViaSnd :: (Char -> Int) -> (Char -> Bool) -> Char -> Bool
areCommutativeViaSnd f1 f2 x =
  (snd . (f1 &&& f2)) x
    == f2 x
```

▲リスト8.11: f_2 = snd . (f_1 &&& f_2)を確認する関数

fstがsndに、比較対象がf_2に変わったんだね。

そう。
そして最後に③。これは……

……あっ

ん、どうした？

えーとね。……ううん、別になんでもないよ。

自分の考えを持つこと、披露してみることは、大事だよ、μ。
間違ってもいいんだ。

うん……ありがと。

そのね、これ、少し考えてみたんだけど。ord &&& isSpace関数以外にも、別のこんな関数は考えられない？

```
h :: (a -> b) -> (a -> c) -> a -> (b, c)
h f g x =
  let y = f x
      z = g x
  in (y, z)

h ord isSpace :: Char -> (Int, Bool)
```

▲リスト8.12: &&&とは違う？　関数h

いいね。僕も丁度、そんな例を使って説明しようと思ったんだ。

ということは、やっぱり違う、かな。

そうだね、反例にはなっていない……でも大切な発想だよ。

数学には、発想が必要だからね。そういう発想は、どんどん挙げて欲しい。

……えへへ、ありがとう。

……。

さて、なぜ反例になっていないかだけど……このhと&&&を簡約すると、等価だからなんだ。

hと&&&が同じものなら、どちらを指しても一意ってことだね。

等価であることを、簡単に示してみるよ。

```
h  =  \f g x ->
        let y = f x
            z = g x
        in (y, z)

      let式を展開
   =  \f g x -> (f x, g x)

f &&& g  =  \x -> (f x, g x)
  fとgを移項
(&&&)    =  \f g x -> (f x, g x)

  つまり
h  =  (&&&)
```

▲リスト8.13: h == &&&

ほんとうだ、形はちがくっても、等しいコードってあるんだね。

まあ実際は、処理が最適化されたりされなかったりする場合があると思うけどね。

ここでは、Haskellの関数が等しいとは、このように簡約すると同じ形になる――と定義しておくよ。

えへへ。

あとはfをChar -> (Int, Bool)に特殊化しておけば、Charをいじわるに解釈する……次のような関数も考えられるけど

```
another :: Char -> (Int, Bool)
another x = (ord x + 252, True)
```

▲リスト8.14:「f = another」？

これは可換性を満たさない。

```
areCommutativeViaFst' :: Char -> Bool
areCommutativeViaFst' x =
  (fst . another) x
    == ord x
```

▲リスト8.15: f ≠ another

```
>>> areCommutativeViaFst' 'a'
False
```

▲リスト8.16: fst . another と ord が 可換でないこと

③まで確認できた……ってことは、(Int, Bool)はIntとBoolの直積……？

あとひとつだけ確認してみよう！

(Int, Bool)への射は任意の対象Yと、任意の射f_1,f_2からのものなので――Yが Char以外でも、①②③を満たすかどうかを考える必要があるよ。

あ、そっかぁ。

といっても&&&は、f_1とf_2のような……同一のドメインを持つ、任意の射を適用することができるので、容易に一般化できる。

任意のy, f_1, f_2 に対して
　　∀ y : Y,　 f_1 : Y -> Int,　 f_2 : Y -> Bool

① areCommutativeViaFst f_1 f_2 y　==　True
② areCommutativeViaSnd f_1 f_2 y　==　True

である。
また、その任意のf_1, f_2に対して

③ h f_1 f_2　==　f_1 &&& f_2

である。

▲リスト8.17: 任意のfst ． f_1 &&& f_2 と f_1 が 可換であること

areCommutativeViaFstとかで、関数を引数に持っておくのはなんでだろうと思ってたんだけど……これを確認するためだったんだね。

うん、そうだよ。再利用性は大事だからね！

8.2.4　任意の(X_1, X_2)が直積であること

そして最後に、もっと一般化して……

IntとBoolを任意の型X_1とX_2に一般化しても、①②③は言えるんだ。

つまり――任意の対象X_1とX_2に対して、対象(X_1, X_2)はその直積になるよ、と。

！　そこも……いっぱんか、できちゃうんだ。

そうだよ。可能なところまで、一般化してみるのは、事実を見据えるための、大切なことだね。

8.2.5　小まとめ

直積についてのまとめ。

圏の**対象の直積**は、次のように定義される。

以下の命題①②③が真である、
組〈$X_1 \times X_2$，π_1，π_2〉が、ある対象X_1とX_2の直積である。

- - -

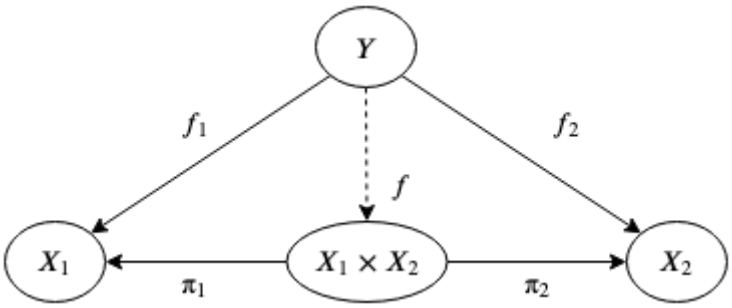
ある対象Yと、以下の射があったときに
　　∀ Y,　f_1 : Y -> X_1,　f_2 : Y -> X_2

以下の等式を満たす。
　　① f_1 = π_1 ○ f
　　② f_2 = π_2 ○ f

また　③ 射fは一意に定まる。

▲リスト8.18: 直積〈$X_1 \times X_2$，π_1，π_2〉の定義

こういうことだね♪



▲ 図8.8: 直積〈X_1×X_2, π_1, π_2〉

Good。

圏Haskにおいて、任意の対象X_1とX_2に対する(X_1, X_2)は、その直積の定義を満たすよ。

例えばYをCharに固定したり、X_1をIntに・X_2をBoolに固定したりして、確認しやすいね。

ただし命題で「任意の」と書かれているところは、最終的に固定を外して考える必要があることに、注意だね。

やったー！　今回はちょっとむずかしかったけど……ちょくせき、理解しました〜！

ふふ、よかった。

8.3　対象の直和

8.3.1　直和の定義

次は対象の**直和**について。

これをやったら、また休憩を挟もう。

いっぱいおしえてくれて、ありがとうね。

ははは、こちらこそありがとうね。

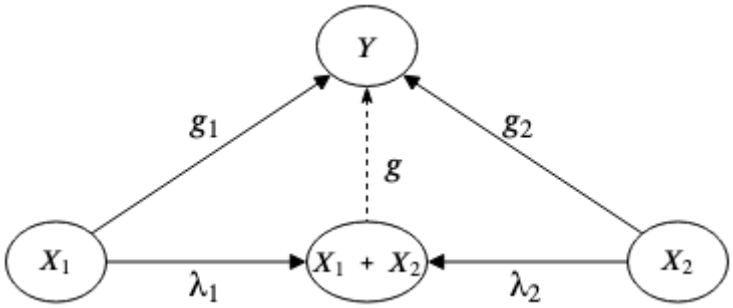
えへへ。

対象の直和は、対象の直積を**双対**（そうつい）にした定義になる。

双対っていうのは、圏の全ての射を真逆にしたもの、という意味。

へぇ〜、真逆。

さっきと同様、まずは圏一般の、可換図式での定義を見てみよう。射が真逆になっていることが、見て取れるよ。



▲ 図8.9: 直和X_1＋X_2

さっきは射がYから伸びていたのに、今度はYに伸びているね。あ、×から＋にもなってる。

そう。あとは、πがX_1＋X_2からのλになっているね。

わー、なんだか、わかりやすー。

うん、双対性は素晴らしいものだよ。

gのような射を**直和の仲介射**（ちゅうかいしゃ）。λ_1とλ_2のような射を直和の**入射**という。

登場人物が、また多くなってきたね。直積と照らし合わせて、まとめよう。

- 直和: 〈X_1＋X_2，λ_1，λ_2〉
  - 入射λ_1 ： X_1 -> X_1＋X_2
  - 入射λ_2 ： X_2 -> X_1＋X_2
  - 仲介射g ： X_1＋X_2 -> Y —— 任意のYへのg_1,g_2に対する
- 直積: 〈X_1×X_2，π_1，π_2〉
  - 射影π_1 ： X_1×X_2 -> X_1
  - 射影π_2 ： X_1×X_2 -> X_2
  - 仲介射f ： Y -> X_1×X_2 —— 任意のYからのf_1,f_2に対する

直和はにゅうしゃっ、直積はしゃえいっ。向きがはんたいで、なまえがちがうんだ。

そそ。X_1，X_2への射が入射、X_1，X_2からの射が射影だね。

じゃあさっきと同じように……言葉で直和を定義した後、圏Haskでの直和を見てみよう。

以下の命題①②③が真である、
組〈X_1＋X_2，λ_1，λ_2〉が、ある対象X_1とX_2の直和である。

- - -

ある対象Yと、以下の射があったときに
　∀ Y，　g_1 ： X_1 -> Y，　g_2 ： X_2 -> Y

以下の等式を満たす。
　① g_1 ＝ g ○ λ_1
　② g_2 ＝ g ○ λ_2

また　③ 射gは一意に定まる。

▲リスト8.19: 直和〈X_1＋X_2，λ_1，λ_2〉の定義

8.3.2　圏Haskの対象の直和

和は足し算だったよね。直和もやっぱり、和とはちがう概念なの？

積が直積に、和が直和に、絶対にならないかというと……そうではないんだ。圏による。

でも、必ずしも一致する概念ではないという意味では……うん、そうだね。

そっか。なることも……あるんだ。

その通りだよ。

……Haskのわかりやすい直積として(X_1, X_2)があったように――Haskのわかりやすい直和として`Either` X_1 X_2がある。

今回も、補助関数を書いておこう。

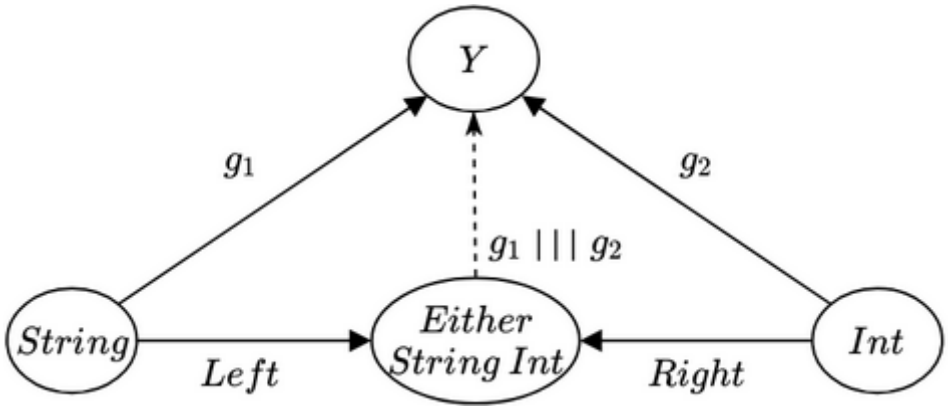
```
(|||) :: (a -> c) -> (b -> c) -> Either a b -> c
f ||| g = \x -> case x of
  Left  e -> f e
  Right a -> g a

head :: String -> Char
head "" = ' '
head (x:_) = x

lead :: Int -> Char
lead = head . show
```

▲リスト8.20: 補助関数(|||), head, lead

```
>>> head ||| lead $ Left "mu"
'm'
>>> head ||| lead $ Left "eta"
'e'
>>> head ||| lead $ Right 1000
'1'
>>> head ||| lead $ Right 0
'0'
```

▲リスト8.21: |||, head, leadの動作確認

そして次はHaskの、直和の可換図式。



▲ 図8.10: 直和Either String Int

うんうん。

言葉でも定義しておこう。

以下の命題①②③が真であるので、
組〈Either String Int, Left, Right〉は、対象StringとIntの直和である。

- - -

任意の対象Yと、任意の射g_1，g_2があったとき
　∀ Y,　g_1 : X_1 -> Y,　g_2 : X_2 -> Y

以下の等式を満たす。
　① g_1 = (g_1 ||| g_2) . Left
　② g_2 = (g_1 ||| g_2) . Right

③
　①②を満たすような射　g : Either String Int -> Char　は
　　g_1 ||| g_2　以外にない。

　つまり、一意に g = g_1 ||| g_2 である。

▲リスト8.22:〈Either String Int, Left, Right〉が直和であるとは

8.3.3　対象Either String Intが直和であること

直和のときと同じ流れだね。

鋭いね、そう。

この①②③を確認できれば、Either String IntがStringとIntの直和であることがわかる。

流れの通り、任意のY・g_1・g_2の一例として、それぞれChar・head・leadを当てはめてみて、①②③を考えてみよう。

```
g1 :: String -> Char
g1 = head

g2 :: Int -> Char
g2 = lead
```

▲リスト8.23: Y，g_1，g_2に当てはめてみる

うんうん！

次の条件関数areCommutativeViaLeftを、直和のために定義する。

これが任意のCharの値に対して、Trueを返すので、①は満たされる。

```
areCommutativeViaLeft :: (String -> Char) -> (Int -> Char) -> String -> Bool
areCommutativeViaLeft g1 g2 x =
  ((g1 ||| g2) . Left) x
    == g1 x
```

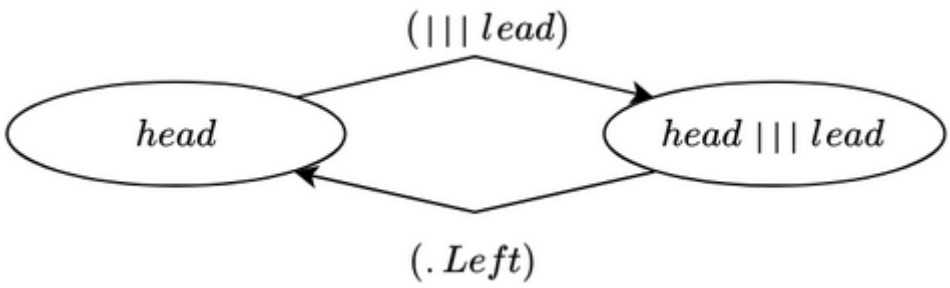
▲リスト8.24: $g_1 = (g_1 \mid\mid\mid g_2)$. Left——の確認

```
>>> areCommutativeViaLeft head lead ""
True

>>> areCommutativeViaLeft head lead "A"
True
>>> areCommutativeViaLeft head lead "B"
True
...
>>> areCommutativeViaLeft head lead "AA"
True
>>> areCommutativeViaLeft head lead "AB"
True
...
>>> areCommutativeViaLeft head lead "BA"
True
...
```

▲リスト8.25: g_1 と $(g_1 \mid\mid\mid g_2)$. Left が 可換であること

こんどは、何が何をうちけすの？

今度は(||| lead)と(. Left)の合成が、headの恒等関数になるよ。具体的には、こうだね。



▲ 図8.11: (||| lead) と (. Left)

②も同様に、(head |||)と(. Right)が、leadの恒等関数になる。

えと……うん。なるね！

やはり|||は、任意のg_1・g_2を適用できるので、③も満たせる。

これで①②③が確認できた。

8.3.4　任意のEither X_1 X_2が直和であること

そしてやっぱり、任意のEither X_1 X_2が、直和になったり……？

その通り。任意のX_1・X_2に対して、(X_1, X_2)が、その直積になったように――

任意のX_1・X_2に対して、Either X_1 X_2は、その直和になる。

やっぱり！

勘がいいのは、数学ではとても大事。さすがμだ。

えへへ……！

8.3.5　小まとめ

直和についてのあとめ。

圏の**対象の直和**は、次のように定義される。

以下の命題①②③が真である、
組〈X_1＋X_2，λ_1，λ_2〉が、ある対象X_1とX_2の直和である。

- - -

ある対象Yと、以下の射があったときに
　　∀ Y，　g_1 : X_1 -> Y，　g_2 : X_2 -> Y

以下の等式を満たす。
　　① g_1 = g ○ λ_1
　　② g_2 = g ○ λ_2

また　③ 射gは一意に定まる。

▲リスト8.26: 直和〈X_1＋X_2，λ_1，λ_2〉の定義

例えば圏Haskにおいては、任意の対象X_1・X_2に対しての〈Either X_1 X_2，Left，Right〉が、その直和の定義を満たした。

直和は、ほとんど直積と同じ内容だったから、わかりやすかったね。

うんうん。双対性のおかげだね。

……

可換性、対象の直積・直和。

これらは圏論で、数多のことを表現する。数学にも広く現れる――抽象概念だ。

これでμは、「宇宙」を見つけられるだろう。そう、あとは――関手――さえ、あれば。

8.4　ノートの切れ端D

8.4.1　対象の直積について

今回考えたのは、**2対象の**直積のみだった。実は直積は2つに限らず、n個の対象の直積が考えられるんだ。例えば3対象の直積、4対象の直積。

1)　3対象の直積〈X_1×X_2×X_3, π_1, π_2, π_3〉と、対象Yからの射f, f_1, f_2, f_3を、可換図式で書いてみて。
2)　その可換図式を、言葉での定義に直してみて。

*図8.5*と、*リスト8.3*が参考になるよ。

8.4.2　対象の直和について

直積と同様に、n個の対象の直和も考えられるよ。

1)　3対象の直和〈X_1＋X_2＋X_3, λ_1, λ_2, λ_3〉と、対象Yへの射g, g_1, g_2, g_3を、可換図式で書いてみて。
2)　その可換図式を、言葉での定義に直してみて。

*図8.9*と、*リスト8.19*が参考になるよ。

8.4.3　その他

実は以下の図はどちらとも、ある圏の図式として書いたものだったんだ。

- *図8.7*
- *図8.11*

1) その圏の対象と射は何か？

圏Setsの対象と直積となる、ずばり**集合の直積**というものがある。また――圏Setsの対象と直和となる、ずばり**集合の直和**というものがある。

余力があれば、調べてみて。

数学にも、慣れてきたかな？楽しんでもらえれば、うれしい :)

8.5 （本章での参考文献）

- 可換図式 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/可換図式*
- 図式 (圏論) - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/図式_(圏論)*
- 積 (圏論) - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/積_(圏論)*
- 極限 (圏論) - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/極限_(圏論)*
- 余積 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/余積*
- 双対 (圏論) - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/双対_(圏論)*

第9章　出会い

◆　◆　◆

じゃあ、お風呂を入れてくるね。

ありがとう。

今日もとうとう、夜、18時。そっと一息。
μは……ぴょこぴょこ、歩いてる。

思い出す、彼女との出会い。
僕がμと、初めて出会った場所。それは……

○　○　○

それで、君は何がお望みなの？

これから、よろしくね。

○　○　○

……

ピコココ・ピココ・ピココココン♪
オフロガ・ワキマシタ。

まぬけな機械音声で思考が中断される。いつの間にかμも戻ってきていた。

じゃあ、お風呂に入ろうか。
僕とμ、どっちが先に入るかはいつも通り――

さいしょは、ぐー！

じゃんけんぽんっ。

じゃんけん！

◆ ◆ ◆

第10章　関手

じゃあ今日の、最後の圏論、やっていこう。

今回は、関手という概念の定義と、その具体例を解説するよ。

お〜〜！

10.1　関手 = 圏から圏への対応

10.1.1　関手の定義

集合には写像という概念があったよね。

あったあった。集合と集合を結びつける……写すやつだ。

そうそう。

今回はそれと似たような、圏と圏を写す――**関手**（かんしゅ）というものを見ていこう。

正確には**共変関手**（きょうへんかんしゅ）というものについて。

関手……「共変」関手？

関手というと一般には、共変関手と**反変関手**（はんへんかんしゅ）に分けられるんだ。

ふぅん？

まあ、今は違いを理解しなくても大丈夫。今回は共変関手のみをやろう。

ηがそういうなら、間違いないね♪

ありがとう。

じゃあまずは、関手を、言葉で定義してみるね。

関手F : C -> Dとは、対象関数Fと射関数Fの組のことである。
（C, Dは、ある圏。）

対象関数Fとは、圏Cの任意の対象Xを、圏Dのある対象F(X)へ、割り当てるものである。
  ∀X ∈ ob(C)
    ⇒ F(X) ∈ ob(D)

射関数Fとは、圏Cの任意の射fを、圏Dのある射F(f)へ、割り当てるものである。
  ∀f : X -> Y ∈ ar(C)
    ⇒ F(f) : F(X) -> F(Y) ∈ ar(D)

- - -

射関数Fは以下の法則①②を満たす。

①
圏Cの任意の恒等射id_Xに対して、圏Dの恒等射$id_{F(X)}$が割り当たる。
  ∀ id_X : X -> X ∈ ar(C)
    ⇒ $id_{F(X)}$: F(X) -> F(X) ∈ ar(D) ∧
       $F(id_X) = id_{F(X)}$

②
圏Cの任意の射f, gの合成g ○ fに対して以下を満たす。
  ∀ f : X -> Y, g : Y -> Z, g ○ f : X -> Z ∈ ar(C)
    ⇒ F(g ○ f) = F(g) ○ F(f)

▲リスト10.1: 関手の定義

そう、関手の目的。それは――**圏の構造を保存する**――というもの。

……。

最初はわからないよね。

そうしたら、具体的な関手の定義と、そのイメージを見てみよう。その保存のイメージが、わかるかも。

おなじみ、具体例タイムだね！

10.1.2　関手C : Sets -> Tord

関手C : Sets -> Tordというのを考えてみるよ。

おー！

まずは言葉で、定義してみよう。

ここでのSetsとは集合と写像の圏。Tordとは自然数と順序の圏のこと。

[*1] 一般的には「圏Sets」とは「全ての集合と全ての写像からなる圏」だけど、ここではその、図式に書く**部分**のみを定義とするよ。

圏Sets [*1]
  ob(Sets) = { {x}, {y}, {a,b,c} }
  ar(Sets) = {
    f   : {x} -> {y}
    f^{-1} : {y} -> {x}
    g   : {y} -> {a,b,c}
    h   : {x} -> {a,b,c}
    各恒等射
  }

圏Tord
  ob(Tord) = { 1, 2, 3 }
  ar(Tord) = {
    $\leqq_{1,2}$: 1 -> 2
    $\leqq_{2,3}$: 2 -> 3
    $\leqq_{1,3}$: 1 -> 3
    各恒等射
  }

▲リスト10.2: 圏Sets, Tord

対象関数C
  C({x})   = 1
  C({y})   = 1
  C({a,b,c}) = 3

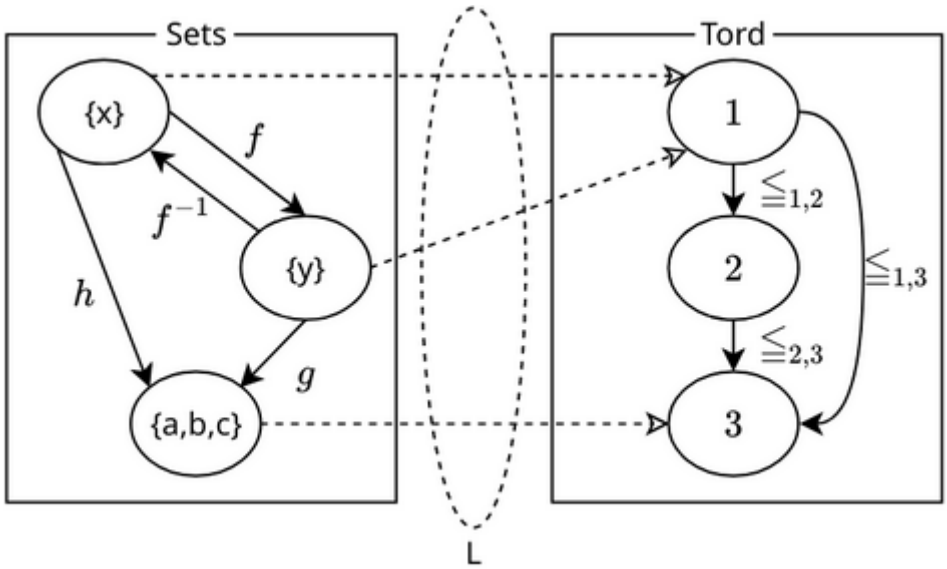
射関数C
  C(f)   = $\leqq_{1,1}$
  C(f^{-1}) = $\leqq_{1,1}$
  C(g)   = $\leqq_{1,3}$
  C(h)   = $\leqq_{1,3}$
  C($id_{\{x\}}$)   = $\leqq_{1,1}$
  C($id_{\{y\}}$)   = $\leqq_{1,1}$
  C($id_{\{a,b,c\}}$) = $\leqq_{3,3}$

▲リスト10.3: 関手C : Sets -> Tord

いっぱい……でてきたね。

ややこしくなったときは、イメージ図でまとめよう。

これは対象関数Cを表すイメージ。



▲ 図10.1: 関手の対象関数C : ∀X -> C(X)

TordとはA "t"otally "ord"er。Cは"Cardinality"を意味してるよ。

また、SetsとTordの各射は可換とする。

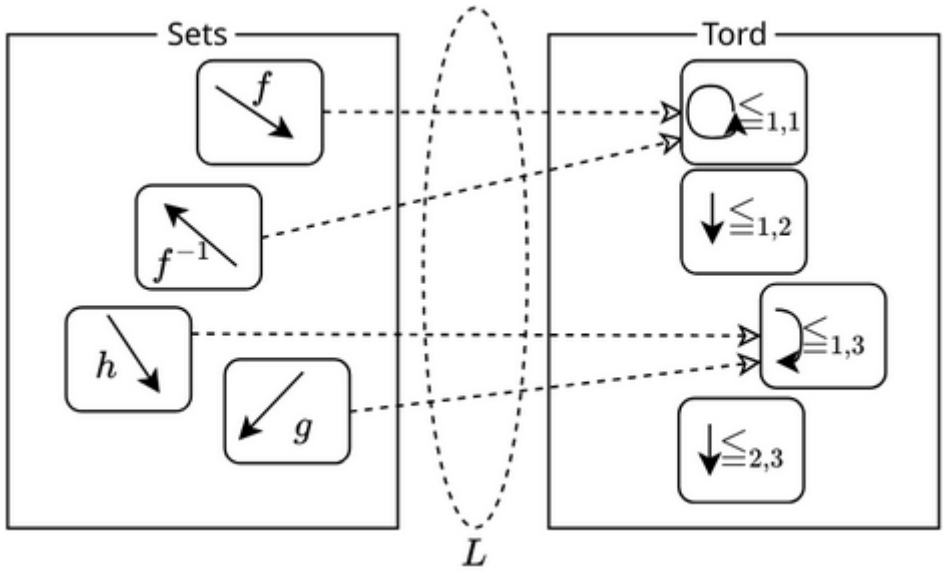
A tortally orderとCardinalityってなーに？

A totally orderは、自然数の順序関係≦のことだよ。

Cardinalityは、集合の"濃度"のこと。濃度1の集合{x}と{y}は、Tordの対象1に。濃度3の集合{a,b,c}は、3に写されてるのが、わかりやすいね。

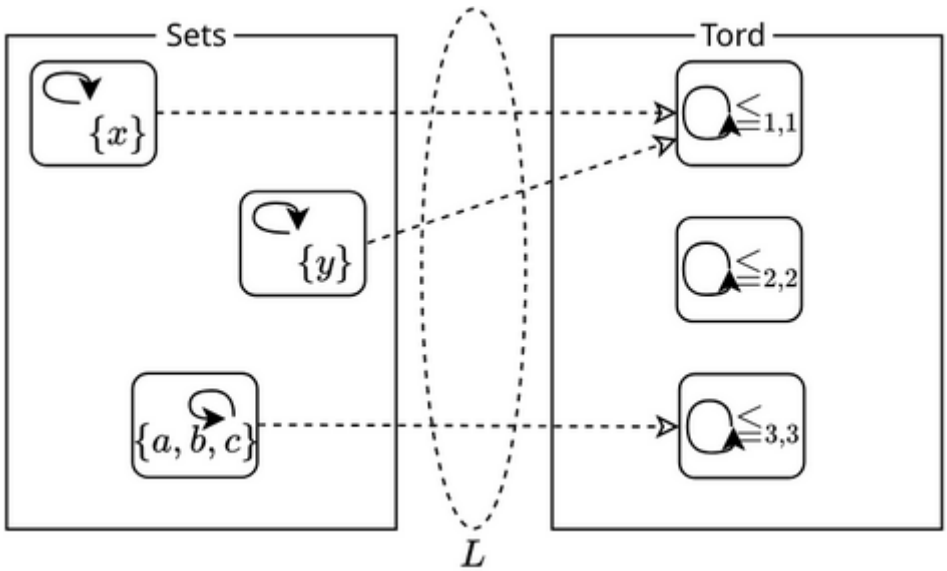
おお。確かに、元の個数が割り当たってる。

ふふ、やっぱりCardinalityは綺麗な関手だよね。
……そして射関数は、次のような割り当たりを持つよ。



▲ 図10.2: 関手の射関数C : ∀f -> C(f)

ちょっと書ききれなかったから、恒等射はこっち。



▲ 図10.3: 関手の射関数C——そのうちCの恒等射

$\leqq_{1,1}$とかって、1とかの恒等射ってことかな。

そうだよ。1 ≦ 1だからね。2同士と3同士も。

ほんとだ。

うんうん、関手C。あたまに入ってきたかも。

よかった。……そしたら、本題。

このCが、確かに関手の法則①②を満たしていること――を確認してみよう。

これがわかったなら、関手を理解できたと言えるはずだよ。

おっおー！

まず①、ここでは「Setsの各恒等射が、Tordの各恒等射に写されていること」だったね。これは定義により、直ちに証明できる。

*図10.2*がいってるの、そのままかな？

その通り。理解が早くて助かるよ。

じゃあ②。これは……パズルのように、=で結ばれた射を置き換えたり、射を合成したりすることで、示すことができる。プログラミングに近いね。

$$
\begin{aligned}
C(f \bigcirc f^{-1}) &= C(\mathrm{id}_{\{x\}}) && \text{—— 可換性による } f \bigcirc f^{-1} = \mathrm{id}_{\{x\}} \\
&= \leqq_{1,1} \\
&= \leqq_{1,1} \bigcirc \leqq_{1,1} && \text{—— 任意の射は恒等射を合成しても等しい（圏の公理）} \\
&= C(f) \bigcirc C(f^{-1})
\end{aligned}
$$

$$
\begin{aligned}
C(g \bigcirc f) &= C(h) \\
&= \leqq_{1,3} \\
&= \leqq_{1,3} \bigcirc \leqq_{1,1} && \text{—— 任意の射は恒等射を合成しても等しい} \\
&= C(g) \bigcirc C(f)
\end{aligned}
$$

$$
\begin{aligned}
C(h \bigcirc f^{-1}) &= C(g) && \text{—— 可換性による } h \bigcirc f^{-1} = g \\
&= \leqq_{1,3} \\
&= \leqq_{1,3} \bigcirc \leqq_{1,1} && \text{—— 任意の射は恒等射を合成しても等しい} \\
&= C(h) \bigcirc C(f^{-1})
\end{aligned}
$$

▲リスト10.4: 各射の合成が保存されること

*図10.2*を見ながらだと、置き換えがわかりやすいよ。

うんーと……これは何を……表しているんだっけ。

②は任意のa,b∈ar(Sets)に対する、C(a ○ b)みたいな任意の合成が、C(a) ○ C(b)になる——っていう法則だよ。

これは、その確認だね。

例えば……C(f ○ f^{-1})を取って、組み替えて、最終的にC(f) ○ C(f^{-1})を得る。……そういう各操作。

おお〜、そっかぁ。こうやって示せばいいんだ。

……

というところで……早くも今回の大筋は終わりだね。どうかな、関手は理解できた？

関手は、うーんと……特別な写像なんだね。対象と、射を写す。

そうそう。圏から圏への、対象関数と射関数のこと……って感じだね。

なんで関手って、こんな法則してるんだろう。そういえば、関手は「圏の構造を保存する」っていってたっけ。あれって……？

例えるなら……『ワンッ』と鳴く犬(けん)が、関手によって写されたなら、その先でも『ワンッ』と鳴いて欲しいんだ。犬を写したはずが、キメラが出てきちゃったら……僕らはどう対応すればいいか、わからなくなってしまうからね。

わかるような、わからないような……。

具体的には、例えば――任意の関手F : C -> Dは、Cの各射が可換であれば、Dの各射も可換にするんだ。可換性を保つ、ってことだね。

おおー。『ワンッ』と鳴くこと、可換なことを保存するんだ。

そう。『キャキャンッ』と鳴いたら、その先の犬も『キャキャンッ』だよ。

わ……。

なんちゃって……ご、ごめん。

ふふ、ありがとう。なんとなく、感覚は掴めたかも！

そう。ならよかった。

10.2　様々な関手

じゃあ最後に、2つの蛇足をたどってみよう。

- 自己関手とは
- 恒等関手とは
- 関手になりそうでならない例

うん、よろしく。

自己関手というものがある。これは関手のうち、ドメインとコドメインが同じもの、として定義される。

```
S : Sets -> Sets
T : Tord -> Tord
H : Hask -> Hask
```

▲リスト10.5: 自己関手の例

自己射に似てるね……？

いい視点だね、その通りだよ。

実際、「ある見方」をすると、自己関手は自己射になるんだ。

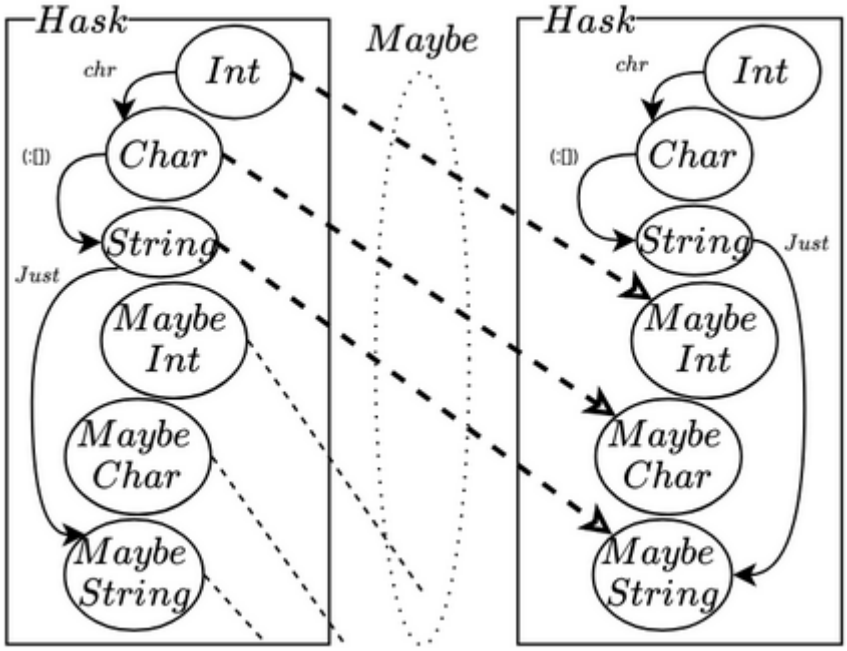
ふぅん？

まあ、それらの詳しい解説は今度にして、今回は自己関手そのままを理解するのに専念しよう。

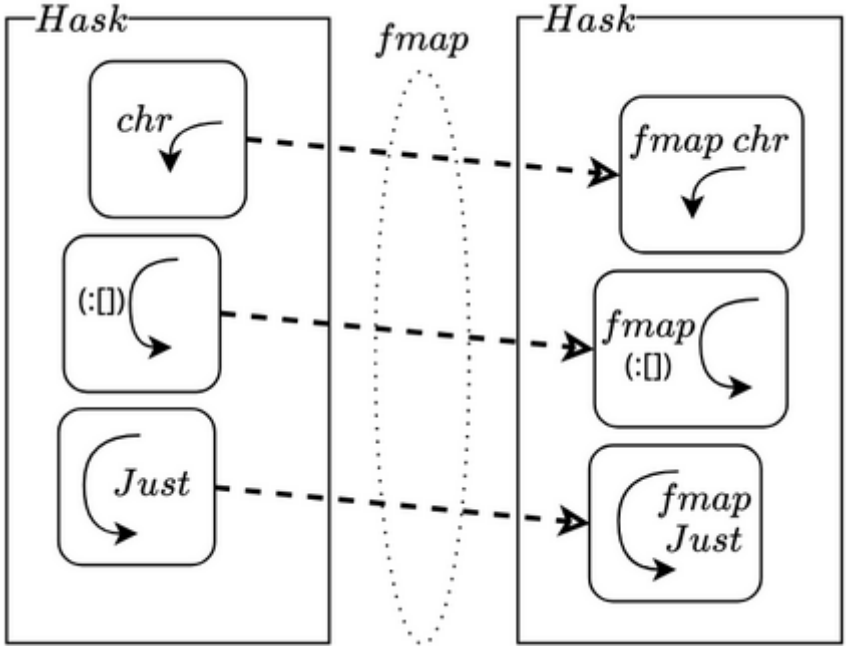
実際の、自己関手の例を。実はHaskellのFunctorのインスタンスは、全てHaskの自己関手になっている。

Maybeとか、Either Stringとか……だっけ。

そうそう。イメージを見てみよう。



▲ 図10.4: 自己関手<Maybe, fmap> : Hask -> Hask ── 対象関数Maybe



▲ 図10.5: 自己関手<Maybe, fmap> : Hask -> Hask ── 射関数fmap

ここでMaybe Int・Maybe Charなどは、実際はMaybe (Maybe Int)やMaybe (Maybe Char)に写される。

全部XがMaybe Xに写されるんだね。

そう。

fmap chr・fmap (:[])なども、実際はfmap (fmap chr)・fmap (fmap (:[]))などに写される。

わあ。

また次のようなIdとidは……

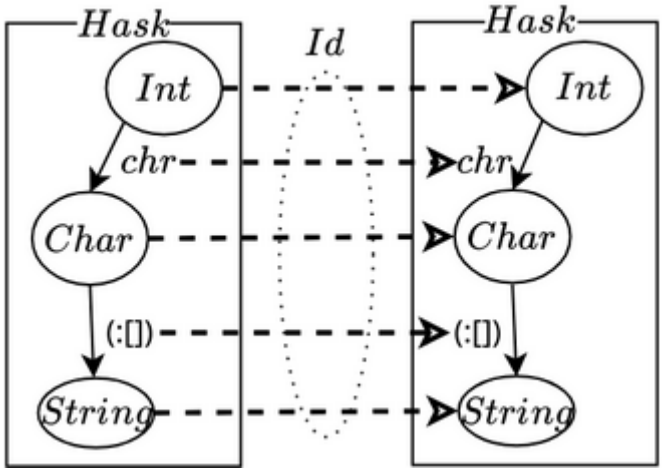
```
type Id a = a

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

▲リスト10.6: Id

恒等関手と呼ばれる、任意の対象と射を、そのまま写す関手になるよ。

```
∀X ∈ ob(Hask)
   ⇒  id(X) ∈ ob(Hask)  ∧
      id(X) = X

∀f : X -> Y  ∈  ar(Hask)
   ⇒  id(f) : X -> Y  ∈  ar(Hask)  ∧
      id(f) = f
```

▲リスト10.7: 恒等関手〈id, id〉 : Hask -> Hask



▲ 図10.6: 恒等関手〈id, id〉 : Hask -> Hask

あれっ。対象関数も、射関数も、どっちもidってこと？

実は、そう。Haskの持つ、**強い**性質だね。

Haskって、結構おもしろい圏だよ。

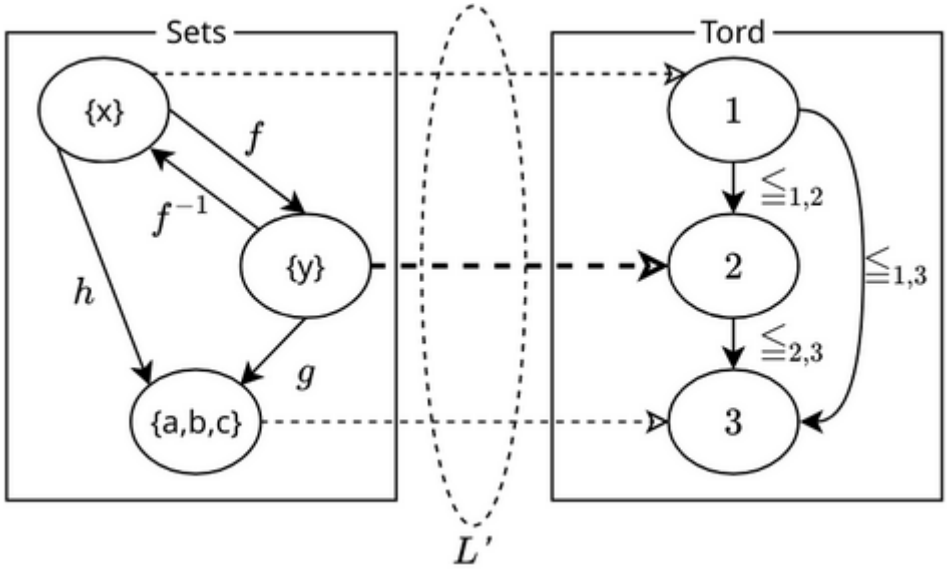
ほんとうだね〜。

10.2.1　誰でも簡単に、関手になれる？

じゃあ今度こそ、本当に最後。

将来、足を踏み外さないために……関手になりそうだけど、ならない例を見てみよう。

先程見た関手`C : Sets -> Tord`の、`C({y})`を2に変えてみたよ。これを`C'`と名付けておこう。



▲ 図10.7: 写像C' : Sets -> Tord――関手ではない

あ、太い点線のところだね。

そう。

μ、これの射関数を考えてみてくれるかな。……予め言っておくと実は、これの射関数は考えられないんだ。

なぜ考えられないかを、考えてみてくれるかな。ヒントはここ。

```
対象関数の割り当てが変わったので
  C({y})  = 1
    ⇓
  C'({y}) = 2
射関数の割り当ても変わる
  C(f)  : 1 -> 1
    ⇓
  C'(f) : 1 -> 2

じゃあ他の、Setsの各射は？
```

▲リスト10.8: ヒント

*図10.7*を見ながら、考えてみて。

ふぅん？　えーと……うーん、何からすればいいんだろう……。

まずはSetsの各射を、羅列してみるといいよ。

ありがと〜、やってみるね。

```
f    : {x} -> {y}
f-1  : {y} -> {x}
h    : {x} -> {a,b,c}
g    : {y} -> {a,b,c}
```

▲リスト10.9: Setsの各射

こうかな。

Goodだよ。

次にこの各射の、射関数C'による割り当てを考えてみよう。途中まで書いてみるね。

```
C'(f)    : ? -> ?
C'(f-1)  : ? -> ?
C'(h)    : ? -> ?
C'(g)    : ? -> ?
```

▲リスト10.10: 射の写像C'による、各射のドメインとコドメイン――を考える

「?」と書いたところを考えてみると、わかるかもしれない。やってみてくれるかな。

うん！　えーと……、こうかな。

```
C'(f)    : C'({x}) -> C'({y})
C'(f^-1) : C'({y}) -> C'({x})
C'(h)    : C'({x}) -> C'({a,b,c})
C'(g)    : C'({y}) -> C'({a,b,c})
           ⇓
C'(f)    : 1 -> 2
C'(f^-1) : 2 -> 1
C'(h)    : 1 -> 3
C'(g)    : 2 -> 3
```

▲リスト10.11: 射の写像C'による、各射のドメインとコドメイン

うん、いいね！

じゃあ最後に、実際の割り当てを考えてみよう。

```
C'(f)    = ?
C'(f^-1) = ?
C'(h)    = ?
C'(g)    = ?
```

▲リスト10.12: 射の写像C'の定義――を考える

「?」には圏Tordの射が当てはまるはずなので、考えうるのは「$\leqq_{1,2}$・$\leqq_{2,3}$・$\leqq_{1,3}$」もしくは各恒等射だけだね。どうだろう？

えーと……

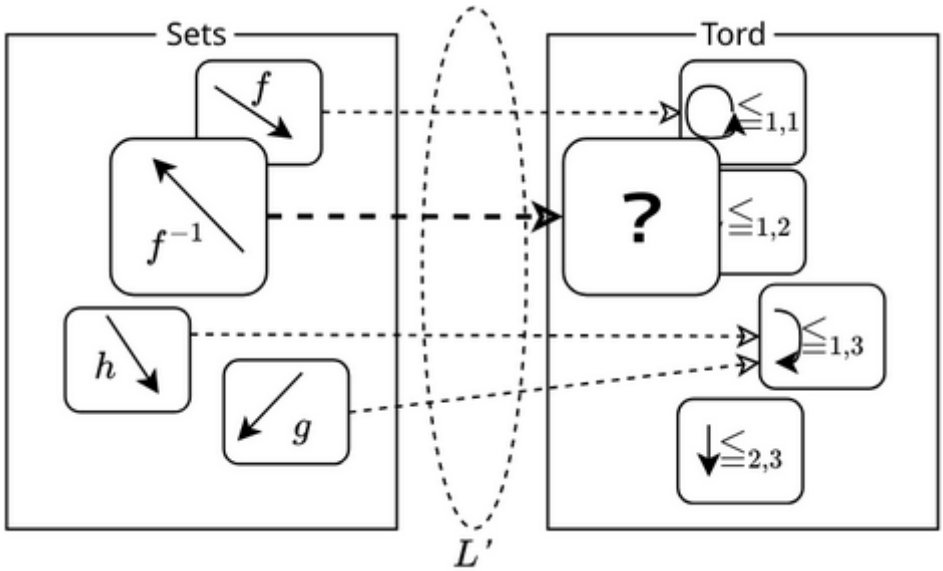
```
C'(f)    = ≦1,2
C'(f-1) = ?
C'(h)    = ≦1,3
C'(g)    = ≦2,3
```

▲リスト10.13: 射の写像C'？

あ！　C'(f^{-1}) : 2 -> 1を考えたけど、Tordに2 -> 1の射がない！

C'(f^{-1})への割り当てが、考えられないんだ。

その通り。さすがμだね！



▲ 図10.8: 射の写像C' : Sets -> Tord――関手ではなかった

えへへー、ありがとー♪

へえ〜。関手って簡単になれそうなのに、そうでもないんだね〜。

うん、そうなんだ。だから、考えたものがちゃんと関手になるかどうか…証明してみるのは、大事だね。

ふふふ。問題がとけたときの、この感覚。なんだかうれしいな。

うんうん、わかるよ。なんだか宇宙を……感じるよね。

そこまでじゃないけど、……うん♪

10.3　まとめ

関手について、まとめよう。

おねがい。

関手――**共変関手**――とは、対象関数と射関数の組。それらは圏から圏への写像であり、**圏の構造を保存する**ものである。

例えば、ある圏CからDへの関手`F : C -> D`は、以下の法則①②を満たす。

```
対象関数F
  ∀X ∈ ob(C)
    ⇒  F(X) ∈ ob(D)

射関数F
  ∀f : X -> Y  ∈  ar(C)
    ⇒  F(f) : F(X) -> F(Y)  ∈  ar(D)

①
∀id_X : X -> X  ∈  ar(C)   ―― id_Xは恒等射
    ⇒  id_F(X) : F(X) -> F(X)  ∧
       F(id_X) = id_F(X)

②
∀g ○ f : X -> Z  ∈  ar(C)
    ⇒  F(g ○ f)  =  F(g) ○ F(f)
```

▲リスト10.14: 関手の定義

①②は圏を保存する法則……だっけ。

そう。その最もたる性質として、ドメインの圏の可換性を、コドメインにも写す――というのがあったね。

うんうん。ちょっとだけ、わかったかも。

ふふ、よかった。

……関手の種類として――

関手G : C -> Cのような、ドメインの圏とコドメインの圏が同じ、**自己関手**というもの。

自己関手のうち、任意の対象と射をそのまま写す、**恒等関手**というもの。

――がある。

```
∀X ∈ ob(C)
   ⇒  G(X) ∈ ob(C)
      G(X) = X

∀f : X -> Y  ∈  ar(C)
   ⇒  G(f) : X -> Y  ∈  ar(C)  ∧
      G(f) = f
```

▲リスト10.15: 恒等関手G : C -> C

そして証明は大事、と。

お疲れ様。これで全ての内容が終わったよ。

えへへ〜、おつかれさま。

10.4　ノートの切れ端E

1)　適当な離散圏Sから、他の適当な圏Cへの関手Fを考えて、それが確かに法則を満たしていることを、確認してみて。
2)　例1: S = { x, y, z }からのF : S -> Tord
3)　例2: T = { ★ }からのF : T -> Sets

「ある見方」をすると、自己関手は自己射となるんだ。それは、圏Catというものを考えた場合のこと。

Catでは、関手が射になる。

1)　圏Catの対象は何になるのか、考えてみて。
2)　その対象の恒等射は何か、考えてみて。

圏Cの各射が可換である場合、関手F : C -> Dで写された圏Dも可換となる。これは、とっても面白い！

1) 適当な圏Cの可換図式を書いてみて。
2) 適当な圏Dの図式を書いてみて。
3) 適当な関手F : C -> Dの定義を書いて、Dが確かに可換になることを確認してみて。

10.5 （本章での参考文献）

- 関手 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/関手*
- 反対圏 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/反対圏*
- 順序集合 - Wikipedia（2019-04-14時点）: *https://ja.wikipedia.org/wiki/順序集合*

第11章　終わり

◆ ◆ ◆

もう夜だね。夜は少し……寂しい。

朝おきたときは、朝だったのに……ふしぎだね。

そうつぶやきながら２人で、同じベッドの上にのる。
2人分の体重、ベッドの少しだけ、きしむ音。

ギシ……。

……

よーしよし、よーしよし♡

ありがとうね、今日はうれしかったよ〜。ηが、わたしと、なにか一緒にやろうって……言ってくれるなんてさ。

すぐ横から、息づかいが聞こえる。

（ふぁ……。）

細い指先に撫でられて……くすぐったくて、声が出そう。

……

思い出すのは……圏論。
μは趣味がなかった。どこか曖昧で、消えてしまいそうな。だって、μは……。

消えてしまうなんてことない。わかっているけど。
だから数学という現実を、μに植え付けようとして……。

……

何か、言葉を返そうと……思ったのに、口が開かない。目も、開かない。

おやすみ、η。

おやすみなさい。
おやすみ、また明日も一緒に。

◆ ◆ ◆

11.0.1　8年前

○ ○ ○

おきていますか。おーい、おきてる？

……ん……。

おはよう。まだ夜は開けてないけれど。

ううー〜ん……。あれ、ここは？

ここは夢の中。わけあって流れ着いた人の、願いを叶えるための空間なの。

夢……？
夢の中のはずなのに、まだ眠い。

それで……

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。考えたことがなかったことを聞かれる……夢。

（ギュッ）

拳を握る。少し、寂しい記憶を思い出した。
だから、かな……

……ずっと一緒にいてくれる人が欲しいかな。

弱い言葉を吐いてしまう。
別れは寂しい。
別れは……寂しい。

それだけでいいの？　それくらい、ここで願わなくても叶いそうなものだけど。

ええと……うん。

眠って見る夢の中なのに、なんだか眠い。

……わかった。

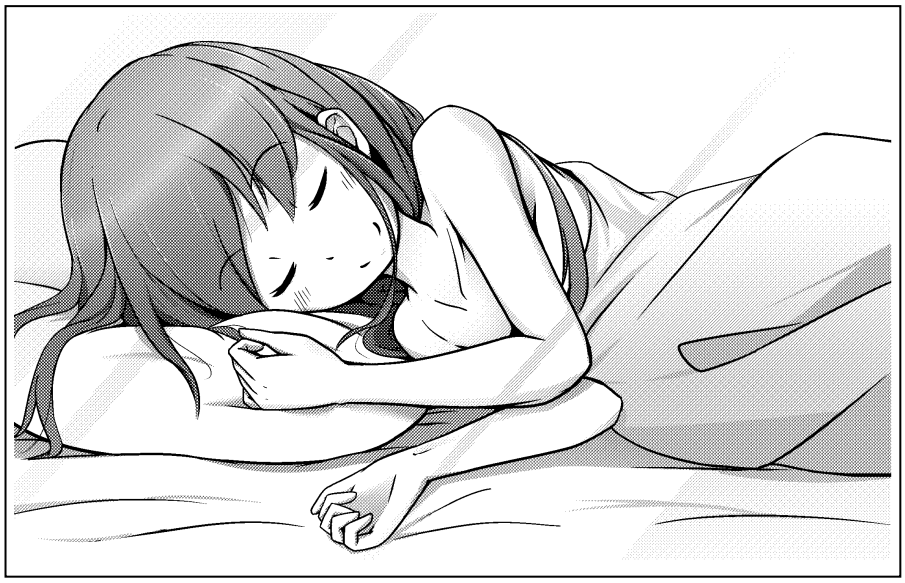
辺りが明ける。
白い空間が、さらに白く染まっていく。

それじゃあ……

夢から醒めていく。

これから、よろしくね。

○　○　○



（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

電子版ダウンロードURL: *https://aiya000.booth.pm/items/1298611*
ダウンロードパスワード: they-are-endomorphisms
※個人利用の範疇を超えた、ダウンロードパスワードの共有はご遠慮ください。