関数型言語 OCaml で作る
代数計算機

icchon 編　著

2026-06-28 版　　未定 発行

前書き

前書きなのだ。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

2

目次

第 1 章

代数計算機を作るとは？

本書は、四則演算と累乗根だけを使って方程式の厳密解を式として求める計算機を、OCaml の型とファンクタで実装します。この章では、本書が採用するアプローチと、その背景にある考え方を説明します。

1.1 数値的に解くか、代数的に解くか

方程式を解くには、大きく分けて 2 つのアプローチがあります。1 つは、ニュートン法のような数値計算で、近似的な数値を求める方法。もう 1 つは、四則演算と累乗根だけを使って、厳密な解を式として書き下す、代数的な方法です。

本書がつくるのは後者の計算機です。たとえば 3 次方程式の解を、$0.6180339\ldots$ のような近似値としてではなく、四則演算と累乗根だけを使った厳密な式として求めます。

1.2 「代数的に解く」プログラムにも深さがある

ここで、「代数的に解くプログラムをつくる」という言葉ひとつをとっても、その実現のしかたには深さの違いがあることに触れておきたいです。

もっとも素朴なのは、解の公式に数値を当てはめて、その結果を文字列として組み立てて表示するだけのプログラムです。もう少し頑張ると、$\sqrt{4}$ のような式を見つけたら 2 に簡約する、といった書き換えルールをたくさん用意して、式が変化しなくなるまで適用し続けるプログラムになります。世の中の数式処理システムの多くは、突き詰めるとこの「式の書き換え」を土台にしています。

本書がつくろうとしているのは、これらとは少し違う道です。「環」「体」「拡大体」といった代数構造そのものを、OCaml の**型**として構築し、四則演算を、その型に対して正しく定義された演算として実装します。こうして得られる解は、表示用の文字列ではな

く、ちゃんと**定義された代数構造の値**になります。式の書き換えによる方法では、最終的に得られるのは表示用の文字列です。本書の方法で得られるのは代数構造の値そのものなので、得られた解に対してそのまま計算を続けたり、検算したりすることが、特別な工夫なしにできます。

1.3 型が代数になる、ということ

本書では、この「代数構造を型として構築する」という方針を、整数から有限体まですべての代数構造に一貫して適用します。

引き算は、「足し算と逆元さえあれば自動的につくれる」ものでした。本書ではこの考え方をすべての演算に当てはめ、**群**（Group）・**モノイド**（Monoid）・**環**（Ring）・**体**（Field）といった代数的な性質を、「ある性質を満たす代数を受け取ると、別の演算を自動生成する」**ファンクタ**として実装していきます。この部品を積み重ねていくと、整数・有理数・複素数・ベクトル・多項式・有限体という一見バラバラに見える代数たちが、実は同じ部品の組み合わせでできていることが見えてきます。

実は、こうした発想を持つ数式処理システムは、本書が初めてではありません。**Axiom**という CAS は、**圏**（Category）と**ドメイン**（Domain）という仕組みで、代数的な性質を型システムの中に表現しています。本書がやろうとしているのは、発想としてはこのAxiom に近いことを、OCaml のモジュールとファンクタという、比較的身近な道具立てだけでゼロから組み立ててみる、という試みです。

1.4 なぜ OCaml なのか

1.4.1 シグネチャとファンクタが標準機能であること

「代数構造を型として構築する」という方針を実現するには、**シグネチャ**（型や値が満たすべきインターフェース）と、**ファンクタ**（あるシグネチャを満たすモジュールを受け取り、別のモジュールを返す関数）を、言語の標準機能として持っている必要があります。OCaml のモジュールシステムは、この 2 つをそのままの形で提供しています。

同じことを Java や Python、C++ のような言語で行おうとすると、インターフェースやジェネリクス、テンプレートを駆使した力技になりがちで、「群を受け取って環を返す」といった、モジュールそのものを変換する処理を素直には書けません。Haskell の型クラスも似た発想を実現できますが、「シグネチャ（何を満たすべきか）」と「ファンクタ（構造をどう変換するか）」が言語機能として明確に分かれているわけではなく、本書のように「ある代数から別の代数を機械的に生成する」処理を中心に据えるには、OCaml のモ

ジュール言語の方が素直に対応します。

1.4.2 必要以上の型システムを持ち込まないこと

もう 1 つの理由は、必要以上の型システムを持ち込まずに済むことです。本書が実装する代数構造は、あらかじめ決まった有限個の演算をまとめたものにすぎず、それを表現するのに OCaml の機能で十分です。Idris や Agda のような**依存型**（Dependent Type）を持つ言語であれば、「群である」という性質そのものを型として証明付きで表現することもできますが、本書の目的（方程式の解を、正しく計算できる値として求めること）にはオーバースペックです。OCaml は、モジュールとファンクタという道具だけで、この本に必要な範囲の型安全性を過不足なく提供してくれます。

1.5 前提知識

高校程度の数学（方程式、複素数、ベクトルの基本）があれば十分です。群・環・体といった抽象代数の知識は前提とせず、本書の中で必要になった順番に定義していきます。

OCaml についても、関数型プログラミングの経験は必須ではありません。`type`・`module`・`functor` といった構文は、「環境構築」の章のあとに置いた「本書で使う OCaml の構文」の章でまとめて説明します。

コードの実行結果は、コンパイル・実行の手順とあわせて本文中にそのまま示します。そのため、手元で OCaml を実行しなくても、本書は最後まで読み進められます。実際に手を動かしてコードを試したい方のために、環境構築の章でセットアップ方法も説明しますが、それは必須の準備ではなく、そうしたい方向けの選択肢です。

1.6 本書の構成

本書は、大きく 2 つの章で構成されています。

基礎的な代数をつくるでは、整数から出発し、有理数・複素数・ベクトル・多項式・有限体と、代数構造を 1 つずつ実装していきます。同じファンクタ（Add_group、Mul_monoid など）が、整数にも有理数にも複素数にも使い回されます。「群・環・体」という抽象的な分類が、具体的で再利用可能な部品であることを、実装を通して確認します。

代数を応用するでは、ここまでつくった部品を組み合わせて、1 次方程式から 3 次方程式までを解きます。

この 2 つの章に入る前に、次の「環境構築」で OCaml を動かす準備を整え、続く「本書で使う OCaml の構文」で以降のコードを読むのに必要な構文を確認します。準備がで

きたら、整数づくりから始めましょう。

9

第 2 章

環境構築

本書のコードとその実行結果は、本文中にそのまま示すので、この章を読まなくても最後まで読み進められます。この章は、実際に手元でコードを動かしてみたい方のためのものです。次の 2 つの環境を整えます。

- OCaml のコンパイラ（`ocamlopt`）— 本書のコード例を実際にコンパイル・実行するため
- **Overleaf**— コードの実行結果（LaTeX 形式の文字列）を、数式として組版された形で確認するため

2.1 opam をインストールする

OCaml のコードをコンパイルするには、OCaml 本体（コンパイラ）が必要です。OCaml 本体は、**opam**（OCaml Package Manager）というパッケージマネージャ経由でインストールするのが一般的です。opam は、OCaml のバージョンやライブラリを、**スイッチ**（Switch）と呼ばれる単位で切り替えながら管理してくれるツールです。

2.1.1 macOS の場合

Homebrew が入っていれば、次のコマンドだけで opam がインストールできます。

```
brew install opam
```

2.1.2 Linux の場合

多くのディストリビューションでは、公式のインストールスクリプトを使うのが手軽です。

```
bash -c "sh <(curl -fsSL https://opam.ocaml.org/install.sh)"
```

Debian/Ubuntu 系であれば、`apt` からインストールすることもできます。

```
sudo apt install opam
```

2.1.3 Windows の場合

WSL（Windows Subsystem for Linux）を使い、WSL 上の Linux 環境（Ubuntu など）に対して、上記の Linux 向けの手順を実行してください。

2.2 OCaml をセットアップする

opam のインストールができたら、初期化を行います。初回だけ必要な作業です。

```
opam init
```

途中でいくつか質問されますが、基本的にはデフォルト（Enter キーを押すだけ）で問題ありません。

次に、OCaml 本体をインストールした**スイッチ**（Switch）を作成します。本書のコードは、OCaml の標準ライブラリだけで書かれているので、特別なバージョン指定は必要ありません。

```
opam switch create default 5.2.1
eval $(opam env)
```

`eval $(opam env)` は、インストールした OCaml にパスを通すためのコマンドです。新しくターミナルを開いたときにも、同じコマンドを実行する必要があります（面倒であれば、シェルの設定ファイル `.zshrc` や `.bashrc` に追記しておくとよいでしょう）。

最後に、正しくインストールできたか確認しましょう。

```
ocaml -version
>>
The OCaml toplevel, version 5.2.1
```

バージョン番号が表示されれば成功です。

2.3 コード例を動かしてみる

それでは、本書のコード例を実際にコンパイルしてみましょう。本書のコードは、GitHub リポジトリの `code/fundamental-algebra/`以下にまとまっています。

試しに、整数の節でつくった `Integer` モジュールをコンパイルしてみます。`code/fundamental-algebra/`ディレクトリに移動して、次のコマンドを実行してください。

```
ocamlopt add_monoid.ml mul_monoid.ml add_group.ml mul_group.ml \
  euclidean.ml integer.ml -o /dev/null
```

何もエラーが出ずに終了すれば、コンパイルは成功です。本書では、このように `ocamlopt` に必要なファイルを列挙してコンパイルする、というスタイルを一貫して使います。ファイルの依存関係（どのファイルがどのファイルを必要とするか）によって列挙する順番が変わる点にだけ、注意してください。

2.4 Overleaf で実行結果を確認する

本書のテストコードは、計算結果を

```
\[\mathrm{add}\left(2, 3\right) = 5\]
```

のような、LaTeX 形式の文字列として出力します。このままでは数式として読みづらいので、Overleaf というサービスを使って、実際に組版された形で確認します。Overleaf は、ブラウザ上で LaTeX の文書を編集・コンパイルできる無料のサービスで、手元に LaTeX 環境を用意する必要がありません。

使い方は簡単です。

1. https://www.overleaf.com にアクセスし、アカウントを作成します
2.「New Project」→「Blank Project」で新しいプロジェクトを作成します
3. 中身を、次のような最小限のテンプレートに置き換えます

▼リスト 2.1　"main.tex"

```
\documentclass{article}
\begin{document}

(ここにOCamlの実行結果を貼り付けます)

\end{document}
```

1
3

1. OCaml のコード例を実行して出力された文字列を、\begin{document}と\end{document}の間に貼り付けます
2. 画面上部の「Recompile」を押せば、数式として組版された結果を確認できます

これで、本書のコードを実際に手を動かしながら読み進める準備が整いました。

第 3 章

本書で使う OCaml の構文

本書のコードは、OCaml の標準的な構文だけで書かれています。ただし、`module` や `functor` を多用するため、一般的な入門書ではあまり触れられない構文もいくつか登場します。この章では、以降の章で繰り返し使う構文をまとめて説明します。数学的な内容の説明のたびに構文の説明を挟まずに済むように、先にここでひととおり触れておきます。

3.1　値と関数の束縛

OCaml では、`let` で値や関数を名前に束縛します。「定義」ではなく**束縛**（Binding）と呼ぶのは、`let` が新しい名前を導入して既存の値と結びつけるだけの構文であり、変数への再代入のように同じ名前の中身をあとから書き換えることはできないためです。型注釈は `(引数名: 型)` や `: 戻り値の型` のように書きます。関数の中で自分自身を呼び出す場合は、`let` の代わりに `let rec` を使います。

▼リスト 3.1　"let / let rec"

```
let x: int = 3
let add (a: int) (b: int): int = a + b
let rec sum (n: int): int = if n = 0 then 0 else n + sum (n - 1)
```

`add 2 3` は 5 に、`sum 5` は 1+2+3+4+5 で 15 になります。

3.2　タプルとパターンマッチ

複数の値をまとめて扱うときは、`(a, b)` のようなタプルを使います。タプルの中身を取り出すには、`let (a, b) = pair in ...` のように書きます。

値によって処理を分岐させるときは `match ... with` を使います。値が存在するかどうかわからない場合の型として `option` 型があり、値があれば `Some x`、なければ `None` という形で表現します。

▼リスト 3.2　"タプルと match"

```
let swap (pair: int * int): int * int =
  let (a, b) = pair in
  (b, a)

let describe (n: int option): string =
  match n with
  | Some x -> string_of_int x
  | None -> "none"
```

本書では、逆元が存在しない場合や、方程式が特定の形の解を持たない場合などに `option` 型を使います。

3.3　モジュールとシグネチャ

OCaml の**モジュール**（module）は、型や値をひとまとめにした単位です。モジュールが満たすべき型や値の一覧を**シグネチャ**（signature）と呼び、`module type ... = sig ... end` で定義します。

▼リスト 3.3　"module と module type"

```
module type COUNTER = sig
  type t
  val zero: t
  val succ: t -> t
end

module Int_counter: COUNTER with type t = int = struct
  type t = int
  let zero = 0
  let succ x = x + 1
end
```

`module Int_counter: COUNTER with type t = int = struct ... end` の `: COUNTER with type t = int` の部分は、`Int_counter` が `COUNTER` というシグネチャを満たすこと、かつ `Int_counter.t` が具体的には `int` であることを表しています。

3.4 ファンクタ

ファンクタ（functor）は、モジュールを受け取って別のモジュールを返す、モジュールに対する関数です。`functor（M: シグネチャ）-> struct ... end` という形で定義します。本書では、ある代数構造を受け取って、別の演算を自動生成する道具としてファンクタを使います。

▼リスト 3.4　"functor"

```
module type ADD = sig
  type t
  val zero: t
  val add: t -> t -> t
end

module Pair_add (M: ADD): ADD with type t = M.t * M.t = struct
  type t = M.t * M.t
  let zero = (M.zero, M.zero)
  let add (a1, a2) (b1, b2) = (M.add a1 b1, M.add a2 b2)
end
```

`Pair_add` は、`ADD` を満たすモジュール `M` を受け取って、`M.t` を 2 つ組にした型の上で成分ごとの足し算を行うモジュールを返します。`Pair_add（Int_add）` のように、シグネチャさえ満たしていれば、どんなモジュールに対してもこのファンクタを適用できます。

--[[path = ocaml-syntax/functor_box (not exist)]]--

▲図 3.1　ファンクタ = モジュールを受け取ってモジュールを返す関数、のイメージ図（仮）

3.5 include

`include` は、他のモジュールやモジュール型に含まれる型・値の定義を、そのまま今のモジュールに取り込む構文です。

▼リスト 3.5　"include"

```
module Pair_add_with_double (M: ADD) = struct
  include Pair_add (M)
  let double (x: t): t = add x x
end
```

Pair_add_with_double は、Pair_add (M) が持つ type t・zero・add をそのまま引き継いだうえで、double を追加します。include のあとに同じ名前の定義を書くと、あとに書いたほうが外部から見える定義として優先されます。この性質は、既存のモジュールに手を加えずに機能を足す場合には便利ですが、意図せず既存の定義を上書きしてしまう原因にもなるので、本書の中でも注意が必要な箇所ではそのつど触れます。

3.6 シグネチャ制約: with type = と with type :=

include でモジュール型を組み合わせるとき、複数のシグネチャがそれぞれ独自に type t を宣言していると、type t が二重に宣言されてしまい、コンパイルエラーになります。これを解決するのが、シグネチャに対する型の制約です。制約には 2 種類あり、書き方は似ていますが意味が異なります。

等価制約 with type =

with type t = 型は、抽象的な型 t が、具体的にはこの型と等しいという制約です。type t という宣言そのものは残るので、外部からは引き続き M.t という名前で参照できます。

破壊的代入 with type :=

with type t := 型は、シグネチャの中の type t という宣言そのものを取り除き、シグネチャ内の t を指定した型で置き換えます。type t の宣言が結果から消えるため、他のシグネチャの type t と衝突しなくなります。

▼リスト 3.6　"with type :=で衝突を避ける"

```
module type MUL = sig
  type t
  val one: t
  val mul: t -> t -> t
end

module type RING = sig
  type t
  include ADD with type t := t
  include MUL with type t := t
```

```
end
```

ADD は前節で定義したシグネチャ、MUL は同じ形で乗法を表すシグネチャです。どちらも独自に type t を持っているため、RING の中でそのまま include ADD・include MUL とすると、type t が 3 回宣言されて衝突します。with type t := t によって、ADD・MUL それぞれの type t を、RING が最初に宣言した type t に置き換えているため、衝突が起きません。本書では、複数の代数的な性質を 1 つの型の上に組み合わせる場面で、この書き方を繰り返し使います。

3.7 ファーストクラスモジュール

通常、モジュールはコンパイル時に決まった名前として扱われ、値のように変数に代入したり関数の引数として渡したりすることはできません。ただし、(module M: シグネチャ) という構文を使うと、モジュールを一時的に通常の値として扱うことができます。これを**ファーストクラスモジュール**（first-class module）と呼びます。

▼リスト 3.7　"ファーストクラスモジュール"

```
module type SHOWABLE = sig
  type t
  val example: t
  val show: t -> string
end

module Int_showable: SHOWABLE with type t = int = struct
  type t = int
  let example = 0
  let show = string_of_int
end

let boxed: (module SHOWABLE) = (module Int_showable)

let describe (m: (module SHOWABLE)): string =
  let module M = (val m) in
  M.show M.example
```

(module Int_showable) で、モジュール Int_showable を (module SHOWABLE) 型の値に変換しています。逆に、値からモジュールに戻すには let module M = (val m) in ... のように書きます。実行時の条件によって使うモジュール（つまり型）を切り替えたい場合に、この構文を使います。

以上が、本書のコードを読むうえで前提となる OCaml の構文です。次の章から、これらの構文を使って実際に代数構造を実装していきます。

第 4 章

基礎的な代数をつくる

この章では、整数・有理数・複素数・ベクトル・多項式・有限体といった、見慣れた数の体系を OCaml で一から実装していきます。ここで大切にしたいのは、「2 + 3 を計算するプログラムを書く」ことそのものではありません。足し算とは何か、逆元とは何かという、演算そのものが満たすべき性質に注目し、その性質をファンクタという形でコードに落とし込んでいきます。

たとえば、整数の引き算は、足し算と符号反転さえあれば自動的に生成できます。有理数の割り算も、掛け算と逆数さえあれば自動的に生成できます。このように、「ある性質を満たす代数を受け取ったら、別の演算を自動生成する」という部品（ファンクタ）を積み重ねていくと、整数・有理数・複素数・ベクトル・多項式・有限体という一見バラバラに見える代数たちが、実は同じ部品の組み合わせでできていることが見えてきます。

これから登場する代数的な構造には、次のような大まかな階層があります。

- **モノイド**（Monoid）: 二項演算と単位元を持つ（逆元は持たない）
- **群**（Group）: モノイドに、逆元の存在が加わったもの
- **環**（Ring）: 加法については群、乗法についてはモノイドであるもの
- **体**（Field）: 環に、0 以外のすべての元に対する乗法の逆元が加わったもの

これらの正式な定義は、整数・有理数を実際につくりながら、この章の中で少しずつ登場します。今はまだ名前だけ頭の片隅に置いておいてください。

4.1 整数

4.1.1 基本的な機能

まずは整数をつくってみましょう。ひとまず整数を使うときに欲しい演算として

- 足し算
- 引き算
- 掛け算

といったものが考えられますね。ちなみに割り算はその結果が整数の世界からはみ出してしまうため今回は考えられません。

これらの演算を実現するためには、等価判定（**イコール**）と単位元（**0** や **1**）が必要になります。これらを踏まえてひとまずコードに起こしてみましょう。

▼リスト 4.1　integer.ml

```
module BASE = struct
type t = int
let equal x1 x2 = x1 = x2
let zero = 0
let add x1 x2 = x1 + x2
let sub x1 x2 = x1 - x2
let one = 1
let mul x1 x2 = x1 * x2
end
```

整数を作ろうとしているのに `type t = int` のように整数を使っているじゃないか。と思った方もいるかも知れません。
しかし、内部で整数型を使用していることは実際どうでもいいのです。自分で定義した整数型の内部がどのように実現されてもいいのです。
これから本書で行なっていく**代数**はそれよりも抽象度の高い階層であり、どのような関数を持っているかという性質やそれら代数同士の関係性を見ていくのです。

足し算や引き算、掛け算が定義できましたね。ここで、引き算に注目してみましょう。

引き算というのは、小学生の頃に足し算や掛け算のように独立した演算として教わったはずです。そして、中学生になって、引き算というのはマイナスの足し算というように表されると知ったわけですね。つまり、逆元（**neg**）を用いて引き算はこのように表されます。

```
let sub x1 x2 = add x1 (neg x2)
```

このように引き算というのは足し算と逆元を使ったアルゴリズムでしかないですね。引き算というのは自分で定義するものではなく、生成されるべきものな気がしてきました。

そこで、ファンクタを使って分離してみましょう。

▼リスト 4.2　add_group.ml

```
module type MIN = sig
  type t
  val add: t -> t -> t
  val neg: t -> t
end

module Extend (M: MIN) = struct
  let sub x1 x2 = M.add x1 (M.neg x2)
end
```

▼リスト 4.3　integer.ml

```
module BASE = struct
type t = int
let equal x1 x2 = x1 = x2
let zero = 0
let add x1 x2 = x1 + x2
let one = 1
let mul x1 x2 = x1 * x2
end
include Add_group.Extend(BASE)
```

BASE モジュールを Add_group の Extend ファンクタに渡し、そのモジュールを include することで引き算を追加しています。足し算と逆元があれば、引き算が自動で作られるようになりましたね。

ところで、今回ファンクタのあるモジュール名[*1]を Add_group としましたが、group というのは**群**（Group）という数学用語です。群とは、集合 G と二項演算 $*$ の組 $(G, *)$ であって、次の 4 つの性質（**群の公理**（Group Axioms））を満たすもののことです。

[*1] OCaml ではファイル名の先頭を大文字にしたものがモジュール名になる

$$
\begin{aligned}
&\text{閉性} && a, b \in G \implies a * b \in G \\
&\text{結合律} && (a * b) * c = a * (b * c) \\
&\text{単位元の存在} && \exists e \in G,\ a * e = e * a = a \\
&\text{逆元の存在} && \exists a^{-1} \in G,\ a * a^{-1} = a^{-1} * a = e
\end{aligned}
\tag{4.1}
$$

$(\mathbb{Z}, +)$、つまり整数と足し算の組で確認してみましょう。閉性・結合律は、2 つの整数を足せばまた整数になり、足す順番を変えても結果が変わらないということなので、直感的に成り立っていますね。単位元は 0 です。そして、任意の整数 a に対する逆元は $-a$ であり、

$$a + (-a) = (-a) + a = 0 \tag{4.2}$$

が成り立ちます。$(\mathbb{Z}, +)$ は群の公理をすべて満たしているため、**群**（Group）です。特に演算が足し算であることを強調して、**加法群**（Additive Group）と呼びます。

群の定義には「逆元の存在」が含まれていました。逆元さえあれば、それを使って引き算を定義できるのでした。裏を返せば、引き算という演算をわざわざ個別に定義する必要はなく、群の性質（足し算と逆元）さえあれば機械的に導出できる、ということです。これが、さきほど `Add_group.Extend` ファンクタとして実装した内容です。Add_group モジュールの Extend ファンクタというのは

加法群を引数にとり引き算を生成する関数

というわけです。

もちろん、加法群であればなんでもよいので、整数でなくても構いません。有理数や複素数、あるいは多項式みたいな代数においても自動で引き算が生成されます。

ただし、自然数（$0, 1, 2, \ldots$）は、たとえば 1 に対する逆元 -1 が自然数の中に存在しないため、逆元が定義できず、Add_group ファンクタを適用できません。

さて、整数型をもう少しだけ充実させましょう。次の関数を追加します。

- compare: 比較
- div_rem: 商と余り
- to_string: 文字列化

▼リスト 4.4　integer.ml

```
module BASE = struct
...
let compare x1 x2 = x1 - x2
let div_rem x1 x2 = (x1 / x2, x1 mod x2)
let to_string x = string_of_int x
end
```

これで基本的な関数は揃いました。これらの関数を用いて新しい関数をもう少し追加してみましょう。

4.1.2　最大公約数

最大公約数を定義するには何が必要でしょうか。試しに、一度書いてみましょう。

```
let rec gcd a b =
    if b = 0 then a
    else gcd b (a mod b)
```


このアルゴリズムを**ユークリッドの互除法**（Euclidean Algorithm）といいます。なぜこれで最大公約数が求まるのか、簡単に確認しておきましょう。a を b で割った商を q、余りを r とすると、

$$a = qb + r \tag{4.3}$$

と書けます。この式を見ると、a と b を共通に割り切る数は、$r = a - qb$ も割り切ることがわかります。逆に、b と r を共通に割り切る数は、$a = qb + r$ も割り切ります。つまり a, b の公約数の集合と、b, r の公約数の集合はまったく同じであり、

$$\gcd(a, b) = \gcd(b, r) = \gcd(b, a \bmod b) \tag{4.4}$$

が成り立ちます。この関係を使えば、(a, b) の最大公約数を求める代わりに、より小さい $(b, a \bmod b)$ の最大公約数を求めればよいことになります。割った余りは必ずもとの数より小さくなるため、この置き換えを繰り返すといつか b が 0 になり、そこでの a がそのまま最大公約数です（a を 0 で割った余りは考えられないので、これ以上小さくできないところで止まります）。

必要なのは、

- 等価判定
- 加法単位元
- 商と余り

だということがわかります。商と余りは quo・rem のように別々の関数にすることもできますが、多くの処理系では 1 回の除算でまとめて求まる値なので、ここでは div_rem という 1 つの関数にまとめ、商と余りの組を返すことにします。

実は、この部品がそろうと、最大公約数だけでなく、もう 1 つ便利なアルゴリズムが同時に手に入ります。**拡張ユークリッドの互除法**（Extended Euclidean Algorithm）と呼ばれるもので、

$$g = u \cdot a + v \cdot b \tag{4.5}$$

を満たす最大公約数 g と、係数 u, v を同時に求めるアルゴリズムです。互除法を 1 ステップ進めるたびに求まる商 q を使って、u, v を再帰の戻りがけに更新していくことで計算できます。今の段階では使い道がわかりにくいと思いますが、これは有限体の節で、ある元の逆元を求めるために使います。拡張ユークリッドの互除法まで導出するには、商と余りに加えて、引き算・掛け算・乗法単位元も必要になるので、ファンクタに渡すべき部品は次のようになります。

▼リスト 4.5　euclidean.ml

```
module type MIN = sig
  type t
  val zero: t
  val one: t
  val sub: t -> t -> t
  val mul: t -> t -> t
  val div_rem: t -> t -> t * t
  val equal: t -> t -> bool
end

module Extend(M: MIN) = struct
  let rec gcd (a: M.t) (b: M.t): M.t =
    if M.equal b M.zero then a
    else gcd b (M.div_rem a b |> snd)

  let rec ext_gcd (a: M.t) (b: M.t): M.t * M.t * M.t =
    if M.equal b M.zero then
      (a, M.one, M.zero)
    else
```

```
      let q, r = M.div_rem a b in
      let (g, u1, v1) = ext_gcd b r in
      (g, v1, M.sub u1 (M.mul q v1))
end
```

これで、最大公約数 gcd と、拡張ユークリッドの互除法 ext_gcd を生成するファンクタができました。

4.1.3 累乗

累乗を定義するのには何が必要でしょうか。試しに、一度書いてみましょう。

```
let pow n x =
    let rec aux acc n =
      if n = 0 then acc
      else aux (acc * x) (n-1)
    in
    aux 1 n
```

累乗というのは、同じ数を n 回掛け合わせたものでした。これを漸化式で書くと、

$$x^n = \begin{cases} 1 & (n = 0) \\ x \cdot x^{n-1} & (n \geq 1) \end{cases} \tag{4.6}$$

となります。$n = 0$ のときは単位元 1 を返し、そうでなければ x を 1 つ掛けて、残りの $n - 1$ 乗を再帰的に計算する。さきほどのコードは、この漸化式をそのままアルゴリズムにしたものです（末尾再帰にするために、掛けた結果を acc に貯めながら計算しています)。

必要なのは、

- 掛け算
- 乗法単位元

であることがわかります。ファンクタにするにはこの部分を引数から取ればいいわけですね。

▼リスト 4.6　mul_monoid.ml

```
module type MIN = sig
  type t
  val mul: t -> t -> t
  val one: t
end

module Extend(M: MIN) = struct
  let pow n x =
    let rec aux acc n =
      if n = 0 then acc
      else aux (M.mul acc x) (n-1)
    in
    aux M.one n
end
```

`n = 0` とか `n-1` とか普通に書いてるけど、さっき引き算や 0 を定義したのにおかしくないか。と思われた方もいるかも知れません。
`let pow n x` というのは一見、整数同士の二項演算に見えますが、実は **n** を固定した単項演算です。`val pow: int -> t -> t` のように型定義されます。ここで **int** と **t** はまったく別の型です。今回は型 **t** が整数なので紛らわしいですね。
`type t = int` として OK だったのと似ていますね。ただのアルゴリズムとして型 **int** を用いているだけなのです。

掛け算と単位元を元に累乗を生成できていますね。ところで、モジュール名を Mul_monoid としましたが、monoid とは、**モノイド**（Monoid）という数学用語です。モノイドとは、集合 M と二項演算 $*$ の組であって、次の性質を満たすもののことです。

$$\begin{aligned} &\text{閉性} && a, b \in M \implies a * b \in M \\ &\text{結合律} && (a * b) * c = a * (b * c) \\ &\text{単位元の存在} && \exists e \in M,\ a * e = e * a = a \end{aligned} \tag{4.7}$$

先ほどの群の公理と見比べてみると、「逆元の存在」が抜けていることに気づくと思います。つまりモノイドとは、群から逆元の存在という条件だけを取り除いたものです。$(\mathbb{Z}, \times)$、つまり整数と掛け算の組を考えてみましょう。閉性・結合律・単位元（1）はすべて満たしていますが、たとえば 2 に対する逆元、つまり $2 \times x = 1$ を満たすような整数 x は存在しません（$\frac{1}{2}$ は整数ではありません）。そのため $(\mathbb{Z}, \times)$ は群にはならず、モ

ノイドにとどまります。Mul_monoid モジュールは、この**乗法モノイド**（Multiplicative Monoid）を示します。

Integer モジュール全体としては次のようになります。

▼リスト 4.7　integer.ml

```
module BASE = struct
  type t = int
  let equal x1 x2 = x1 = x2
  let compare x1 x2 = x1 - x2
  let zero = 0
  let one = 1
  let add x1 x2 = x1 + x2
  let neg x = -x
  let mul x1 x2 = x1 * x2
  let to_string x = string_of_int x
  let div_rem x1 x2 = (x1 / x2, x1 mod x2)
end

include BASE
include Mul_monoid.Extend(BASE)
include Add_group.Extend(BASE)
include Euclidean.Extend(struct
  type nonrec t = t
  let zero, one, sub, mul, div_rem, equal =
    zero, one, sub, mul, div_rem, equal
end)
```


最後の Euclidean.Extend だけは、BASE をそのまま渡していません。BASE 自体は sub を持っていない（引き算は Add_group.Extend が自動生成したものでしたね）ので、ここまでの include によってすでにスコープに入っている zero・one・sub・mul・div_rem・equal を、あらためて明示的に束縛して渡しています。

4.1.4　テスト

作成した Integer モジュールの動作を確認してみましょう。

▼リスト 4.8　integer_test.ml

```
let test_unary f f_name x =
  Printf.sprintf "\\[\\mathrm{%s}\\left(%s\\right) = %s\\]"
    f_name (x |> Integer.to_string) (x |> f |> Integer.to_string)
  |> print_endline
let test_biop f f_name x1 x2 =
  Printf.sprintf "\\[\\mathrm{%s}\\left(%s, %s\\right) = %s\\]"
    f_name (x1 |> Integer.to_string) (x2 |> Integer.to_string)
    (f x1 x2 |> Integer.to_string)
```

```
    |> print_endline

let () =
  let test_add = test_biop Integer.add "add" in
  test_add 2 3;
  test_add 2 (-3);
  let test_mul = test_biop Integer.mul "mul" in
  test_mul 2 3;
  test_mul 2 (-3);
  let test_gcd = test_biop Integer.gcd "gcd" in
  test_gcd 6 2;
  test_gcd 11 4;
  let test_pow = test_unary (Integer.pow 4) "pow4" in
  test_pow 4;
  test_pow 0;
```

コンパイルして、実行用バイナリを作成しましょう。

```
ocamlopt add_monoid.ml mul_monoid.ml add_group.ml mul_group.ml \
euclidean.ml integer.ml integer_test.ml
```

出力バイナリに名前を指定しない場合、a.out が作成されるので、実行してみましょう。

```
./a.out
>>
\[\mathrm{add}\left(2, 3\right) = 5\]
\[\mathrm{add}\left(2, -3\right) = -1\]
\[\mathrm{mul}\left(2, 3\right) = 6\]
\[\mathrm{mul}\left(2, -3\right) = -6\]
\[\mathrm{gcd}\left(6, 2\right) = 2\]
\[\mathrm{gcd}\left(11, 4\right) = 1\]
\[\mathrm{pow4}\left(4\right) = 256\]
\[\mathrm{pow4}\left(0\right) = 0\]
```

結果が Latex 形式で出力されましたね。これを Tex の Viwer で確認してみましょう。

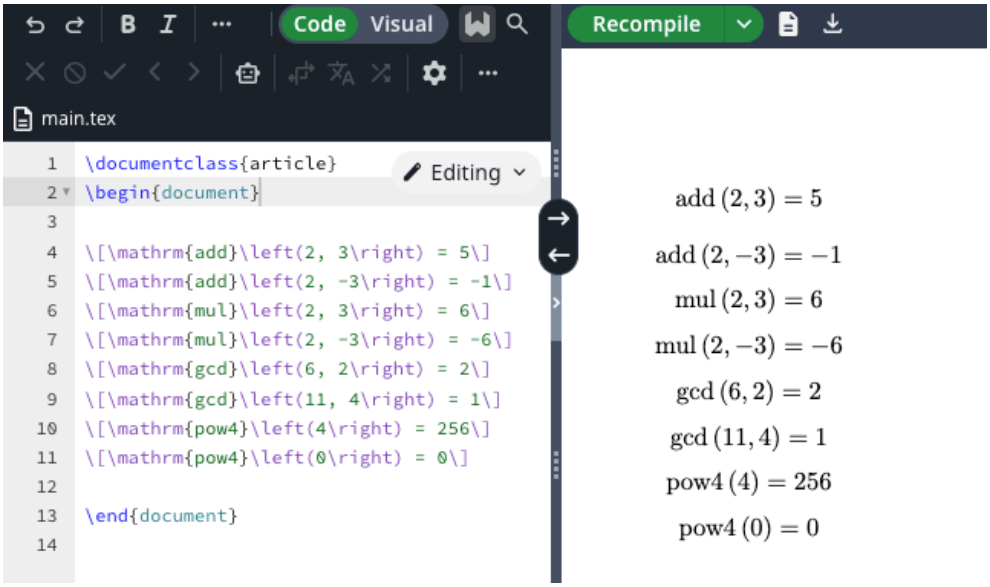
▼リスト 4.9　main.tex

```
\documentclass{article}
\begin{document}

\[\mathrm{add}\left(2, 3\right) = 5\]
\[\mathrm{add}\left(2, -3\right) = -1\]
\[\mathrm{mul}\left(2, 3\right) = 6\]
\[\mathrm{mul}\left(2, -3\right) = -6\]
\[\mathrm{gcd}\left(6, 2\right) = 2\]
\[\mathrm{gcd}\left(11, 4\right) = 1\]
```

```
\[\mathrm{pow4}\left(4\right) = 256\]
\[\mathrm{pow4}\left(0\right) = 0\]

\end{document}
```



▲図 4.1　ブラウザ上でチェック

整数が完成しましたが、中には、なぜ自然数からではなくいきなり整数からつくるのか。と思った方もいるかも知れません。実際、自然数の方がより基礎的な代数であり、自然数から一歩一歩つくり上げていくのが数学的には真摯な姿勢かと思います。しかし、本書においては、方程式を解く数式処理システムを作成することにフォーカスしています。そのため本書においては、数学的な厳密さよりもわかりやすさ、シンプルさを優先しています。

4.2 有理数

4.2.1 基本的な機能

有理数型をつくっていきましょう。有理数型は `type t = (Integer.t * Integer.t )` のように、分子と分母をそれぞれ対応させた 2 つの整数組で表すことにします。

さて、Integer モジュールにおいては内部で int 型を用い、データと一対一に対応していたので、等価判定はシンプルでした。しかしながら、有理数において、

$$\frac{n}{d} = \frac{k \cdot d}{k \cdot n} \tag{4.8}$$

のように、等価判定のために自由度がありますね。等価判定をするためには、2 つのアプローチがあります。

1. 同値変形して等価判定する
2. 正規化をしてデータを比較する

というものです。

1. 同値変形して等価判定する

有理数において次の同値変形をすることです。

$$\begin{aligned} \frac{n_1}{d_1} &\iff \frac{n_2}{d_2} \\ n_1 \cdot d_2 &\iff n_2 \cdot d_1 \end{aligned} \tag{4.9}$$

2. 正規化をしてデータを比較する

定数倍だけの自由度がある有理数に対し、一意に表される形に式変形をするということです。

$$\frac{k \cdot n'}{k \cdot d'} = \frac{n'}{d'} \quad (n' \text{ と } d' \text{ は互いに素}) \tag{4.10}$$

本書では、この **2. 正規化をしてデータを比較する**を採用します。等価判定を行うときに、内部のデータそのものを比較するだけでよくなり、シンプルだからです。また、出力

を見たいとき、どのみち正規化をしたいというモチベーションがあるためです。

正規化を実装する際、何が必要か考えてみましょう。分子と分母を互いに素になるようにするには、それぞれ最大公約数で割ればいいですね。さらに、負の数のときは、必ず分子にマイナスをつけるようにしましょう。

```
let normalize (n, d) =
  let make_positive x =
    if Integer.compare x Integer.zero >= 0 then x else Integer.neg x
  in
  let g = Integer.gcd (make_positive n) (make_positive d) in
  let n' = Integer.div_rem n g |> fst in
  let d' = Integer.div_rem d g |> fst in
  if Integer.compare d' Integer.zero < 0 then
    (Integer.neg n', Integer.neg d')
  else
    (n', d')
```

これで normalize 関数を適用させることで正規化が行われるようになります。他の関数も考えてみましょう。足し算、掛け算は次のように表されますね。

足し算

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2} \tag{4.11}$$

```
let add (n1, d1) (n2, d2) =
  let n = Integer.add (Integer.mul n1 d2) (Integer.mul n2 d1) in
  let d = Integer.mul d1 d2 in
  (n, d)
```

掛け算

$$\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 \cdot n_2}{d_1 \cdot d_2} \tag{4.12}$$

```
let mul (n1, d1) (n2, d2) =
  let n = Integer.mul n1 n2 in
  let d = Integer.mul d1 d2 in
  (n, d)
```

さて、有理数は整数とは違い、掛け算に関する逆元を考えることができますね。

乗法の逆元

$$\left(\frac{n}{d}\right)^{-1} = \frac{d}{n} \tag{4.13}$$

```
let inv (n, d) = (d, n)
```

先ほど作成した Add_group ファンクタと同様に、乗法に関する逆元を元に、割り算を生成するファンクタ: Mul_group を作成してみましょう。

▼リスト 4.10　mul_group.ml

```
module type MIN = sig
  type t
  val mul: t -> t -> t
  val inv: t -> t
end

module Extend(M: MIN) = struct
  let div x1 x2 = M.mul x1 (M.inv x2)
end
```

Add_group ファンクタ（リスト 4.2) とほとんど同じですね。

ここで、乗法の逆元が考えられることにより累乗を拡張することができます。Integer モジュールにおいては、

$$\mathrm{pow}\,(n, x) := x^n \quad (n \geq 0) \tag{4.14}$$

このように、指数部分が自然数でしたが、有理数においては負の数まで拡張ができます。

$$\mathrm{pow}\,(n, x) := \begin{cases} x^n & (n \geq 0) \\ (x^{-1})^{|n|} & (n < 0) \end{cases} \tag{4.15}$$

マイナスの指数というのは、逆数にした後に累乗をすることで定義できます。なぜ負の数の累乗まで拡張できたかというと、乗法の逆元があったからですね。

乗法の逆元がある代数を受け取るファンクタのあるモジュールは、Mul_group でしたね。Mul_group にあるファンクタに累乗の関数を追加してみましょう。

▼リスト 4.11　mul_group.ml

```
module type MIN = sig
  type t
  val one: t
  val mul: t -> t -> t
  val inv: t -> t
end

module Extend(M: MIN) = struct
  let div x1 x2 = M.mul x1 (M.inv x2)
  let pow n x =
    let base = if n >= 0 then x else M.inv x in
    let rec aux acc n =
      if n = 0 then acc
      else aux (M.mul acc base) (n - 1)
    in
    aux M.one (abs n)
end
```

n が負のときは、x そのものではなく、あらかじめ逆数にした M.inv x を base として、その $|n|$ 乗を計算しています。これは先ほどの数式

$$(x^{-1})^{|n|} \tag{4.16}$$

をそのままコードにしたものです。これで、乗法の逆元があれば、負の数までの累乗が自動で生成されるようになりましたね。

Mul_monoid のファンクタと Mul_group のファンクタのどちらにも val pow: int -> t -> t は存在するため、両方 include したい際は、Mul_monoid -> Mul_group の順に include するようにしましょう。そのようにすれば、Mul_group の pow 関数により、Mul_monoid の pow が上書きされます。

有理数全体を書いてみましょう。

▼リスト 4.12　rational.ml

```
module BASE = struct
type t = Integer.t * Integer.t
let one = (Integer.one, Integer.one)
let zero = (Integer.zero, Integer.one)
let normalize (n, d) =
  let make_positive x =
```

```
    if Integer.compare x Integer.zero >= 0 then x else Integer.neg x
  in
  let g = Integer.gcd (make_positive n) (make_positive d) in
  let n' = Integer.div_rem n g |> fst in
  let d' = Integer.div_rem d g |> fst in
  if Integer.compare d' Integer.zero < 0 then
    (Integer.neg n', Integer.neg d')
  else
    (n', d')
let neg (n, d) = (Integer.neg n, d)
let add (n1, d1) (n2, d2) =
  let n = Integer.add (Integer.mul n1 d2) (Integer.mul n2 d1) in
  let d = Integer.mul d1 d2 in
  (n, d)
let mul (n1, d1) (n2, d2) =
  let n = Integer.mul n1 n2 in
  let d = Integer.mul d1 d2 in
  (n, d)
let inv (n, d) = (d, n)
let compare (n1, d1) (n2, d2) =
  Integer.compare (Integer.mul n1 d2) (Integer.mul n2 d1)
let equal x1 x2 = (x1 |> normalize) = (x2 |> normalize)
let to_string (n, d) =
  Printf.sprintf "\\frac{%s}{%s}"
    (n |> Integer.to_string) (d |> Integer.to_string)
end
include Mul_monoid.Extend(BASE)
include Add_group.Extend(BASE)
include Mul_group.Extend(BASE)
```


Integer モジュールを使用して、有理数が完成しましたね。

このままでも十分なのですが、有理数を作る際、受け取るモジュールは Integer に限定する必要はないことに気が付くと思います。

- 加法
- 加法の逆元
- 乗法

があればなんでもいいわけです。このような性質を満たすものを**環**（Ring）と呼びます。また、これに加え、乗法の逆元が加わったものを**体**（Field）と呼びます。今回作成した Rational モジュールも**体**（Field）です。

改めて、環と体の定義を整理しておきましょう。**環**（Ring）とは、集合 R と 2 つの二項演算（加法 $+$、乗法 $\times$）の組 $(R, +, \times)$ であって、次の性質を満たすもののことです。

$$\begin{array}{ll} \text{加法について可換群} & (R,+) \text{ が群であり、} a+b=b+a \\ \text{乗法についてモノイド} & (R,\times) \text{ がモノイドである} \\ \text{分配律} & a\times(b+c)=a\times b+a\times c \end{array} \tag{4.17}$$

整数 $\mathbb{Z}$ はまさにこの環の例でした。加法については逆元（引き算）が存在しますが、乗法については逆元（割り算）が存在しません。

体（Field）は、環にさらに「0 以外のすべての元に乗法の逆元が存在する」という条件を加えたものです。

$$a\in R,\ a\neq 0 \implies \exists a^{-1}\in R,\ a\times a^{-1}=a^{-1}\times a=1 \tag{4.18}$$

有理数 $\mathbb{Q}$ は、整数を分数にすることで、この乗法の逆元を人工的に手に入れた体だと言えます。実際、$\frac{n}{d}$ の逆元は $\frac{d}{n}$ でしたね（$n\neq 0$ であれば）。このように、環から分数をつくることで体を得る構成のことを、数学では**分数体**（Field of Fractions）と呼びます。

というわけで、

Integer モジュールを受け取り、**有理数**を作る

のではなく、

環（Ring）を受け取り**有理体**（Rational Field）を返すファンクタを作る

ことにしましょう。そうすることによって今後新しい**環**（Ring）を作成した時に、ファンクタを適用するだけで有理体を作ることができます。色々と数学用語が登場してしまいましたが、先ほどのコードをただファンクタに変えるだけなのでとても簡単です。

▼リスト 4.13　fraction.ml

```
module type Ring = sig
  type t
  include Add_monoid.MIN with type t := t
  include Add_group.MIN with type t := t
  include Mul_monoid.MIN with type t := t
  val compare: t -> t -> int
  val equal: t -> t -> bool
  val gcd: t -> t -> t
  val div_rem: t -> t -> t * t
  val to_string: t -> string
end

module Make(R: Ring) = struct
  type t = R.t * R.t
```

```
  let one = (R.one, R.one)
  let zero = (R.zero, R.one)
  let normalize (n, d) =
    let make_positive x =
      if R.compare x R.zero >= 0 then x else R.neg x
    in
    let g = R.gcd (make_positive n) (make_positive d) in
    let n' = R.div_rem n g |> fst in
    let d' = R.div_rem d g |> fst in
    if R.compare d' R.zero < 0 then
      (R.neg n', R.neg d')
    else
      (n', d')
  let neg (n, d) =
    (R.neg n, d) |> normalize
  let add (n1, d1) (n2, d2) =
    let d = R.mul d1 d2 in
    let n = R.add (R.mul n1 d2) (R.mul n2 d1) in
    (n, d) |> normalize
  let mul (n1, d1) (n2, d2) =
    let n = R.mul n1 n2 in
    let d = R.mul d1 d2 in
    (n, d) |> normalize
  let inv (n, d) = (d, n) |> normalize
  let compare (n1, d1) (n2, d2) =
    R.compare (R.mul n1 d2) (R.mul n2 d1)
  let equal x1 x2 = compare x1 x2 = 0
  let to_string (n, d) =
    if R.equal d R.one then (n |> R.to_string)
    else Printf.sprintf "\\frac{%s}{%s}" (n |> R.to_string) (d |> R.to_string)

  include Mul_monoid.Extend(struct
    type nonrec t = t
    let one, mul, equal = one, mul, equal
  end)
  include Add_group.Extend(struct
    type nonrec t = t
    let add, neg = add, neg
  end)
  include Mul_group.Extend(struct
    type nonrec t = t
    let one, mul, inv = one, mul, inv
  end)
end
```

to_string では、分母が 1 のときは分数の形にせず、分子だけを表示するようにしています。整数と同じ値の有理数を、わざわざ $\frac{n}{1}$ という分数の形で表示しないための工夫です。

有理体ファンクタができました。これを用いて有理数を作成しましょう。

▼リスト 4.14　rational.ml

```
include Fraction.Make(Integer)
```

驚くほどシンプルです。有理数が整数を**環**（Ring）とする有理体だという定義そのものですね。

4.2.2 テスト

作成した Rational モジュールの動作を確認してみましょう。

▼リスト 4.15　rational_test.ml

```
let test_unary f f_name x =
  Printf.sprintf "\\[\\mathrm{%s}\\left(%s\\right) = %s\\]"
    f_name (x |> Rational.to_string) (x |> f |> Rational.to_string)
  |> print_endline
let test_biop f f_name x1 x2 =
  Printf.sprintf "\\[\\mathrm{%s}\\left(%s, %s\\right) = %s\\]"
    f_name (x1 |> Rational.to_string) (x2 |> Rational.to_string)
    (f x1 x2 |> Rational.to_string)
  |> print_endline

let () =
  let test_add = test_biop Rational.add "add" in
  test_add (1, 2) (1, 3);
  test_add (1, 2) (-1, 3);
  let test_mul = test_biop Rational.mul "mul" in
  test_mul (1, 2) (1, 3);
  test_mul (1, 2) (-1, 3);
  let test_div = test_biop Rational.div "div" in
  test_div (1, 2) (1, 3);
  let test_pow = test_unary (Rational.pow (-4)) "pow(-4)" in
  test_pow (1, 2)
```

コンパイルして、実行用バイナリを作成しましょう。

```
ocamlopt add_monoid.ml mul_monoid.ml add_group.ml mul_group.ml \
euclidean.ml fraction.ml integer.ml rational.ml rational_test.ml
```

出力バイナリに名前を指定しない場合、`a.out` が作成されるので、実行してみましょう。

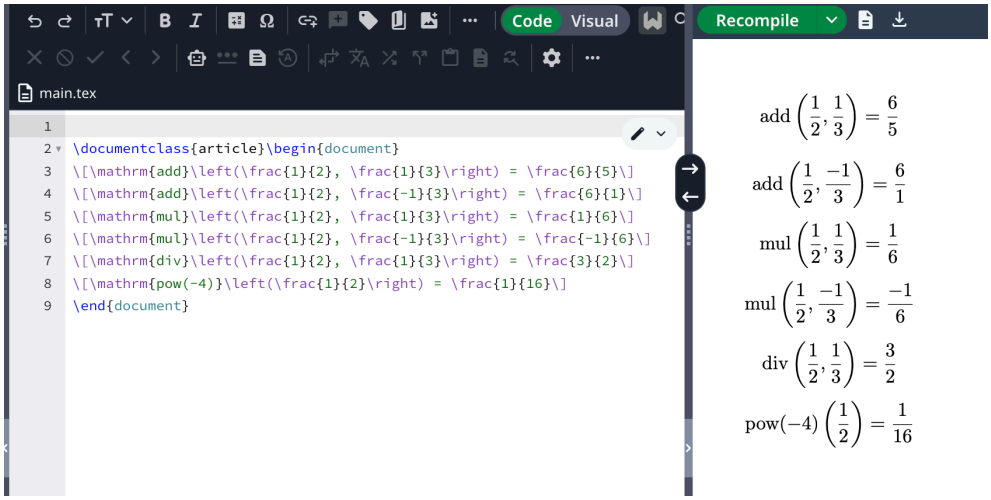
```
./a.out
>>
\[\mathrm{add}\left(\frac{1}{2}, \frac{1}{3}\right) = \frac{5}{6}\]
```

```
\[\mathrm{add}\left(\frac{1}{2}, \frac{-1}{3}\right) = \frac{1}{6}\]
\[\mathrm{mul}\left(\frac{1}{2}, \frac{1}{3}\right) = \frac{1}{6}\]
\[\mathrm{mul}\left(\frac{1}{2}, \frac{-1}{3}\right) = \frac{-1}{6}\]
\[\mathrm{div}\left(\frac{1}{2}, \frac{1}{3}\right) = \frac{3}{2}\]
\[\mathrm{pow(-4)}\left(\frac{1}{2}\right) = 16\]
```

負の指数を渡すと、逆数にしてから累乗する定義どおり、

$$(1/2)^{-4} = 2^4 = 16 \tag{4.19}$$

になっていることが確認できます。結果が Latex 形式で出力されましたね。これを Tex の Viwer で確認してみましょう。



▲図 4.2　ブラウザ上でチェック

0 で割ってみよう

我々は小学生の頃から散々と 0 では割ってはいけないと教わりました。そして、プログラミングにおいても 0 で割るとランタイムエラーが起きたりプログラムがクラッシュしてしまうため、0 で割ることに対し、とても注意してきたはずです。

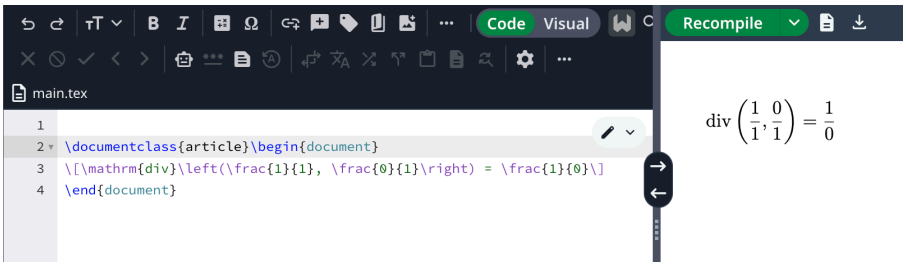
しかし、勇気を振り絞って 0 で割ってみましょう。

▼リスト 4.16　rational_test.ml

```
let () =
  let test_div = test_biop Rational.div "div" in
  test_div (1, 1) (0, 1);
```

```
>>
\[\mathrm{div}\left(1, 0\right) = \frac{1}{0}\]
```

分母が 1 の値は分数の形にせず分子だけを表示するようにしていたので、$\frac{1}{1}$ と $\frac{0}{1}$ はそれぞれ 1 と 0 と表示されています。



▲図 4.3　ブラウザ上でチェック

$$\begin{aligned} \frac{1}{1} \div \frac{0}{1} &= \frac{1}{1} \cdot \frac{1}{0} \\ &= \frac{1}{0} \end{aligned} \tag{4.20}$$

となっていますね。自分で作成した代数なのでランタイムエラーなどはもちろん起こらず出力がされました。

分母に 0 が入っていますが、分母を極限まで小さくした場合、

$$\lim_{n \to 0} \frac{1}{n} = \infty \tag{4.21}$$

無限大となりますね。
そこで、0 で割ることで自然に登場した

$$\frac{1}{0} \tag{4.22}$$

という数字を**無限大**として扱うということを考えてみましょう。また、

$$\frac{0}{0} \tag{4.23}$$

を**不定形**（NaN）としましょう。今から、無限大を含む計算を行ってみましょう。
無限大の足し算

$$\begin{aligned} x + \infty &= \frac{x}{1} + \frac{1}{0} \\ &= \frac{x \cdot 0 + 1 \cdot 1}{1 \cdot 0} \\ &= \frac{1}{0} \\ &= \infty \end{aligned} \tag{4.24}$$

$$x + \infty = \infty \tag{4.25}$$

無限大になりました。
0 と無限大の掛け算

$$\begin{aligned} 0 \cdot \infty &= \tfrac{0}{1} \cdot \tfrac{1}{0} \\ &= \tfrac{0 \cdot 1}{1 \cdot 0} \\ &= \tfrac{0}{0} \\ &= \mathrm{NaN} \end{aligned} \tag{4.26}$$

$$0 \cdot \infty = \mathrm{NaN} \tag{4.27}$$

不定形（NaN）になりました。

$$\begin{aligned} \infty - \infty &= \tfrac{1}{0} - \tfrac{1}{0} \\ &= \tfrac{1 \cdot 0 - 0 \cdot 1}{0 \cdot 0} \\ &= \tfrac{0}{0} \\ &= \mathrm{NaN} \end{aligned} \tag{4.28}$$

$$\infty - \infty = \mathrm{NaN} \tag{4.29}$$

不定形（NaN）になりました。

$$\begin{aligned} \infty + \infty &= \tfrac{1}{0} + \tfrac{1}{0} \\ &= \tfrac{1 \cdot 0 + 0 \cdot 1}{0 \cdot 0} \\ &= \tfrac{0}{0} \\ &= \mathrm{NaN} \end{aligned} \tag{4.30}$$

$$\infty + \infty = \mathrm{NaN} \tag{4.31}$$

不定形（NaN）になりました。

$$\infty + \infty = \infty \tag{4.32}$$

となりそうですが、正の無限大と負の無限大の区別が存在しないため、無限大同士の引き算の結果と一致します。

$$\begin{aligned} x \div \infty &= \tfrac{x}{1} \div \tfrac{1}{0} \\ &= \tfrac{x}{1} \cdot \tfrac{0}{1} \\ &= \tfrac{x \cdot 0}{1 \cdot 1} \\ &= \tfrac{0}{1} \\ &= 0 \end{aligned} \tag{4.33}$$

$$x \div \infty = 0 \tag{4.34}$$

0 になりました。

$$\begin{aligned} x \div 0 &= \tfrac{x}{1} \div \tfrac{0}{1} \\ &= \tfrac{x}{1} \cdot \tfrac{1}{0} \\ &= \tfrac{x \cdot 1}{1 \cdot 0} \\ &= \tfrac{x}{0} \\ &= \tfrac{1}{0} \\ &= \infty \end{aligned} \tag{4.35}$$

$$x \div 0 = \infty \tag{4.36}$$

無限大になりました。$\frac{x}{0}$ が $\frac{1}{0}$ に化けているのを不思議に思われたかもしれませんが、これは `normalize` が正規化する際、分子分母を $\gcd(x, 0) = x$ で割るために起きています。分子分母の比だけが意味を持つ以上、$\frac{x}{0}$ も $\frac{1}{0}$ も「同じ無限大」として扱われるのは自然なことですね。

まとめると、

$$\begin{aligned} x + \infty &= \infty \\ 0 \cdot \infty &= \mathrm{NaN} \\ \infty - \infty &= \mathrm{NaN} \\ \infty + \infty &= \mathrm{NaN} \\ x \div \infty &= 0 \\ x \div 0 &= \infty \end{aligned} \tag{4.37}$$

となります。

このように無限大を含む計算の結果を眺めてみると、そのまま使ってしまっても不都合はないように思えます。このような、0 で割ることを許容する代数を**リーマン球面**（Riemann Sphere）といいます。通常、複素数に無限大を追加した集合

$$\mathbf{C} \cup \{\infty\} \tag{4.38}$$

と表されます。今回は、有理数なので、

$$\mathbf{R} \cup \{\infty\} \tag{4.39}$$

となりますね。

とても便利なのでリーマン球面は有理数の上位互換なのではと思ってしまいますが、無限大を含む計算において、結合法則や分配法則、逆元の存在性という重要

な代数的性質が失われてしまいます。このように、ある機能を追加することによってある性質が失われてしまうというトレードオフの関係になることが多いです。

また、

$$\mathbf{R} \cup \{\infty\} \tag{4.40}$$

に対し、 **不定形**（NaN）を加えた、

$$\mathbf{R} \cup \{\infty, \mathrm{NaN}\} \tag{4.41}$$

を**輪**（Wheel）といいます。興味がある人は**輪論**（Wheel Theory）で調べてみてください。

不定形（NaN）を追加して便利になった代償として、何に 0 をかけても 0 になるという**零元の性質**を失っています。

さて、話が少し広がってしまいましたが、今回作成した Rational モジュールの話に戻ります。Rational モジュールにおいては、内部のデータ型に制限をしていません。具体的には、**分母に 0 が来た時の挙動を内部でハンドリングしていない**ということです。そのため、このモジュールを使用するユーザーは

- **有理数**
- **リーマン球面**
- **輪**

のいずれとしても解釈して使用することが可能です。

4.3 複素数

4.3.1 基本的な機能

有理数を作ることができたので、次は複素数を作っていきましょう。複素数は、$i^2 = -1$ を満たす新しい数 i（虚数単位）を有理数に付け加え、$a + bi$（a, b は有理数）という形の数の全体として定義されます。

複素数になったことにより、新しく共役という単項演算が登場します。

共役

$$\overline{a+bi} = a - bi \tag{4.42}$$

他の基本的な演算に関しては、次のとおりになります。加法

$$(a_1 + b_1 i) + (a_2 + b_2 i) = (a_1 + a_2) + (b_1 + b_2)i \tag{4.43}$$

乗法

$$(a_1 + b_1 i) \cdot (a_2 + b_2 i) = (a_1 a_2 - b_1 b_2) + (a_1 b_2 + b_1 a_2)i \tag{4.44}$$

共役同士の積

$$\begin{aligned} (a+bi) \cdot (a-bi) &= a^2 + b^2 \\ z \cdot \overline{z} &= |z|^2 \end{aligned} \tag{4.45}$$

4
5

乗法の逆元

$$z^{-1} = \frac{\overline{z}}{|z|^2} \tag{4.46}$$

なぜこれが逆元なのかは、直前の「共役同士の積」の式からわかります。$z \cdot \overline{z} = |z|^2$ なので、両辺を $|z|^2$（0 でない有理数）で割ると

$$z \cdot \frac{\overline{z}}{|z|^2} = 1 \tag{4.47}$$

となり、たしかに共役を $|z|^2$ で割った値が z の乗法の逆元になっていることがわかります。実装するときも、この式をそのままなぞればよく、z に共役をかけて $|z|^2$（norm2）を求め、それぞれの成分を norm2 で割るだけです。

これらを踏まえて、複素数を作っていきましょう。実数と虚数に分け、2 つの数字のペアで表現するようにしましょう。コード上で、`type t = Rational.t * Rational.t` です。

▼リスト 4.17　complex.ml

```
module BASE = struct
  type t = Rational.t * Rational.t
  let equal (a1, b1) (a2, b2) =
    (Rational.equal a1 a2) && (Rational.equal b1 b2)
  let zero = (Rational.zero, Rational.zero)
  let one = (Rational.one, Rational.zero)
  let add (a1, b1) (a2, b2) = (Rational.add a1 a2, Rational.add b1 b2)
  let neg (a, b) = (Rational.neg a, Rational.neg b)
  let conj (a, b) = (a, Rational.neg b)
  let mul (a1, b1) (a2, b2) =
    (Rational.sub (Rational.mul a1 a2) (Rational.mul b1 b2),
    Rational.add (Rational.mul a1 b2) (Rational.mul b1 a2))
  let norm2 x = mul x (conj x) |> fst
  let inv x =
    let norm = norm2 x in
    let (a, b) = conj x in
    (Rational.div a norm, Rational.div b norm)
  let to_string (a, b) =
    let impart_to_string x =
      if Rational.equal x Rational.one then "i"
      else if Rational.equal x (Rational.neg Rational.one) then "-i"
      else Printf.sprintf "%si" (Rational.to_string x)
    in
    if Rational.equal b Rational.zero then (Rational.to_string a)
    else if Rational.equal a Rational.zero then b |> impart_to_string
    else Printf.sprintf "%s + %s"
      (a |> Rational.to_string) (b |> impart_to_string)
end

include BASE
include Add_group.Extend(BASE)
include Mul_monoid.Extend(BASE)
include Mul_group.Extend(BASE)
```

Complex モジュールが完成しました。Rational モジュールを内部で使用していることと、ファンクタにより演算を生成していることにより、新しく作成する部分は非常に少なく済みます。

さて、Complex モジュールは Rational モジュールのペアを受け取っていますが、これをファンクタの引数として渡し、**複素的**な構造を返すファンクタというのは考えられないでしょうか。先ほど、有理体ファンクタを作ったのと同じ考えです。

そこまでの抽象化に意味はあるのかという疑問は残ると思いますが、後ほどその嬉しさについて説明するので、ファンクタにしてみましょう。

▼リスト 4.18　complexical.ml

```
module type Field = sig
  type t
  include Add_monoid.MIN with type t := t
  include Add_group.MIN with type t := t
  include Mul_monoid.MIN with type t := t
  include Mul_group.MIN with type t := t
  val equal: t -> t -> bool
  val to_string: t -> string
  val div: t -> t -> t
  val sub: t -> t -> t
end

module Make(F: Field) = struct
  type t = F.t * F.t
  let equal (a1, b1) (a2, b2) =
    (F.equal a1 a2) && (F.equal b1 b2)
  let zero = (F.zero, F.zero)
  let one = (F.one, F.zero)
  let add (a1, b1) (a2, b2) = (F.add a1 a2, F.add b1 b2)
  let neg (a, b) = (F.neg a, F.neg b)
  let conj (a, b) = (a, F.neg b)
  let mul (a1, b1) (a2, b2) =
    (F.sub (F.mul a1 a2) (F.mul b1 b2),
    F.add (F.mul a1 b2) (F.mul b1 a2))
  let norm2 x = mul x (conj x) |> fst
  let inv x =
    let norm = norm2 x in
    let (a, b) = conj x in
    (F.div a norm, F.div b norm)
  let to_string (a, b) =
    let impart_to_string x =
      if F.equal x F.one then "i"
      else if F.equal x (F.neg F.one) then "-i"
      else Printf.sprintf "%si" (F.to_string x)
    in
    if F.equal b F.zero then (F.to_string a)
    else if F.equal a F.zero then b |> impart_to_string
    else Printf.sprintf "%s + %s"
      (a |> F.to_string) (b |> impart_to_string)

  include Mul_monoid.Extend(struct
    type nonrec t = t
    let one, mul, equal = one, mul, equal
  end)
  include Add_group.Extend(struct
    type nonrec t = t
    let add, neg = add, neg
  end)
  include Mul_group.Extend(struct
    type nonrec t = t
    let one, mul, inv = one, mul, inv
  end)
end
```

▼リスト 4.19　complex.ml

```
include Complexical.Make(Rational)
```

Complex モジュールの中にあった実装を Complexical ファンクタに移動させ、Complex モジュールでは、Complexical ファンクタに有理数を渡し、複素数を生成するようにしました。

4.3.2 テスト

作成した Complex モジュールの動作を確認してみましょう。

▼リスト 4.20　complex_test.ml

```
let header = "\\documentclass{article}\\begin{document}"
let footer = "\\end{document}"

let test_unary f f_name x =
  Printf.sprintf "\\[\\mathrm{%s}\\left(%s\\right) = %s\\]"
    f_name (x |> Complex.to_string) (x |> f |> Complex.to_string)
  |> print_endline
let test_biop f f_name x1 x2 =
  Printf.sprintf "\\[\\mathrm{%s}\\left(%s, %s\\right) = %s\\]"
    f_name (x1 |> Complex.to_string) (x2 |> Complex.to_string)
    (f x1 x2 |> Complex.to_string)
  |> print_endline

let () =
  header |> print_endline;

  let x_2 = (2, 1) in
  let x_3 = (3, 1) in
  let x_neg2 = (-2, 1) in

  let test_add = test_biop Complex.add "add" in
  test_add (x_2, x_3) (x_3, x_2);
  test_add (x_2, x_3) (x_neg2, x_3);
  let test_mul = test_biop Complex.mul "mul" in
  test_mul (x_2, x_3) Complex.zero;
  test_mul (x_2, x_3) (x_neg2, x_3);
  let test_div = test_biop Complex.div "div" in
  test_div (x_2, x_3) (x_3, x_2);
  let test_pow = test_unary (Complex.pow (-4)) "pow(-4)" in
  test_pow (Rational.one, Rational.one);

  footer |> print_endline;
```

コンパイルして、実行用バイナリを作成しましょう。

```
ocamlopt add_monoid.ml mul_monoid.ml add_group.ml mul_group.ml \
euclidean.ml fraction.ml integer.ml rational.ml complexical.ml \
complex.ml complex_test.ml
```

出力バイナリに名前を指定しない場合、a.out が作成されるので、実行してみましょう。

```
./a.out
>>
\documentclass{article}\begin{document}
\[\mathrm{add}\left(2 + 3i, 3 + 2i\right) = 5 + 5i\]
\[\mathrm{add}\left(2 + 3i, -2 + 3i\right) = 6i\]
\[\mathrm{mul}\left(2 + 3i, 0\right) = 0\]
\[\mathrm{mul}\left(2 + 3i, -2 + 3i\right) = -13\]
\[\mathrm{div}\left(2 + 3i, 3 + 2i\right) = \frac{12}{13} + \frac{5}{13}i\]
\[\mathrm{pow(-4)}\left(1 + i\right) = \frac{-1}{4}\]
\end{document}
```

結果が Latex 形式で出力されましたね。これを Tex の Viewer で確認してみましょう。

--[[path = fundamental-algebra/overleaf_4 (not exist)]]--

▲図 4.4　ブラウザ上でチェック

ここで、なぜわざわざ Complexical ファンクタを作成したのかを説明します。複素数を作るときしか Complexical ファンクタを使用しない場合、別にファンクタにするメリットはありません。

しかし、複素数以外にもこの Complexical ファンクタを使用する場面があります。それは、**四元数**（Quaternion）を作りたい時です。

四元数

四元数とは、複素数を包含するような集合として定義されます。複素数は虚数部が 1 つでしたが、四元数は虚数部が 3 つあります。

$$
\begin{aligned}
& a + bi + cj + dk \\
& i^2 = j^2 = k^2 = -1 \\
& ij = k \\
& jk = i \\
& ki = j
\end{aligned}
\tag{4.48}
$$

基本的な、演算は次のようなルールとなります。和

$$
\begin{aligned}
(a_1 + b_1 i + c_1 j + d_1 k) + (a_2 + b_2 i + c_2 j + d_2 k) &= (a_1 + a_2) \\
&+ (b_1 + b_2) i \\
&+ (c_1 + c_2) j \\
&+ (d_1 + d_2) k
\end{aligned}
\tag{4.49}
$$

積

$$
\begin{aligned}
(a_1 + b_1 i + c_1 j + d_1 k) \cdot (a_2 + b_2 i + c_2 j + d_2 k) &= (a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2) \\
&+ (a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2) i \\
&+ (a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2) j \\
&+ (a_1 d_2 + b_1 c_2 - c_1 b_2 + d_1 a_2) k
\end{aligned}
\tag{4.50}
$$

四元数の積の結果はとても複雑になりました。四元数の積を普通に実装しようとすると、これらの計算ルールを愚直に記述する必要があります。

しかし、先ほど作成した Complexical ファンクタの引数に Complex モジュールを渡せば、この複雑な計算ルールを自分で書き下さなくても、四元数を生成できそうです（実際に正しく動かすには、これから説明する修正が必要です)。

▼リスト 4.21　quaternion.ml

```
include Complexical.Make(Complex)
```

このように、複素的な構造に着目した代数学の分野として、**ケーリーディクソン代数**（Cayley-Dickson Algebras）というものがあります。複素数や四元数はそれ単体で定義されるイメージが強いですが、**ケーリーディクソン代数**（Cayley-Dickson Algebra）においては、低次の複素的代数から高次の複素的代数を生成するルールによって定められます。

$$\begin{aligned}\overline{(p,q)} &= (\overline{p}, -q) \\ (p,q)\cdot(r,s) &= (pr - \overline{s}q, sp + q\overline{r})\end{aligned} \tag{4.51}$$

それぞれ、共役と積についてのルールが定められています。ここに実数を入れた場合には、複素数が作られ、複素数を入れた場合には、四元数が作られ、四元数を入れた場合には、八元数が作られるといったイメージでいくらでも**ケーリーディクソン代数**（Cayley-Dickson Algebras）が生成可能です。一般化すると、2^n 元数から $2\cdot 2^n$ 元数を生成するということです。

先ほどの Complexical ファンクタの `mul` を、上のケーリー＝ディクソンの式と見比べてみましょう。

$$(a_1, b_1)\cdot(a_2, b_2) = (a_1a_2 - b_1b_2,\ a_1b_2 + b_1a_2) \tag{4.52}$$

これは、ケーリー＝ディクソンの式で共役を全部外してしまったものになっています。F が有理数のように交換法則の成り立つ体であれば、共役は恒等写像（$\overline{x} = x$）なので、この 2 つの式は一致します。実際、複素数（F が有理数）まではこれで正しく動きます。

しかし、四元数をつくるために F に Complex モジュールを渡すと、F 自身が非可換になる場合が出てくるため、共役を省略してしまうと計算が合わなくなります。`mul` を、ケーリー＝ディクソンの式のとおりに修正しましょう。

▼リスト 4.22　complexical.ml

```
let mul (a1, b1) (a2, b2) =
    (F.sub (F.mul a1 a2) (F.mul (F.conj b2) b1),
    F.add (F.mul b2 a1) (F.mul b1 (F.conj a2)))
```

共役になっていなかった部分を共役にしました。ただし、これだけでは足りません。F.conj を呼べるようにするには、ファンクタの引数となる Field シグネチャ自体に conj を追加する必要があります。

▼リスト 4.23　complexical.ml

```
module type Field = sig
  type t
  include Add_group.S with type t := t
  include Mul_group.S with type t := t
  val equal: t -> t -> bool
  val to_string: t -> string
  val div: t -> t -> t
  val sub: t -> t -> t
  val conj: t -> t
end
```

そして、これまで Field シグネチャを満たすものとして渡してきた Rational モジュールにも、conj を実装しなければなりません。幸い、有理数は実数なので、共役をとっても自分自身のままです。

▼リスト 4.24　rational.ml

```
let conj (x: t): t = x
```

これで、Complexical.Make(Rational) で複素数を、Complexical.Make(Complex) で四元数を、Complexical.Make(Complexical.Make(Complex)) で八元数を、と再帰的に高次の複素的代数を生成できる、正しいケーリー＝ディクソン構成のファンクタが完成しました。試しに $i \cdot j$ を計算してみると、修正前は正しく k になりませんでしたが、修正後はきちんと $i \cdot j = k$、$j \cdot i = -k$ となることが確認できます。

ケーリーディクソン代数

ここで、ケーリーディクソン代数の重要な補足を行います。ケーリーディクソンの構成法は、**環**（Ring）を受け取り、**環**（Ring）を返すというものになっています。

しかしながら、今回は**環**（Ring）に乗法の逆元が追加された**体**（Field）を受け取っています。つまり、本来は積の逆元は考えられないということです。

- 一元数 （実数）
- 二元数 （複素数）

- 四元数
- 八元数

までは、乗法逆元が存在しますが、一六元数以降は、乗法の逆元が定義できません。一六元数になると、**非自明な零因子**が出てきてしまい、0 でない数をかけても 0 になってしまうケースが出てきてしまいます。そのために、乗法の逆元が一意に定まらなくなり、逆元が定義できなくなってしまいます。そのため、割り算が可能なのは八元数までとなります。

これらの性質をまとめた表が次のとおりです（表中の性質はすべて乗法についてのものです）。

▼表 4.1　ケーリー＝ディクソン代数の性質

代数	次元	順序	交換法則	結合法則	交代代数	冪結合性	非自明な零因子
実数	1	Yes	Yes	Yes	Yes	Yes	No
複素数	2	No	Yes	Yes	Yes	Yes	No
四元数	4	No	No	Yes	Yes	Yes	No
八元数	8	No	No	No	Yes	Yes	No
十六元数	16	No	No	No	No	Yes	Yes
（16 >）	>16	No	No	No	No	Yes	Yes

53

- 四元数では交換法則
- 八元数では結合法則
- 一六元数では非自明な零因子の存在

といった具合で、少しずつ代数的性質が失われていきます。

--[[path = fundamental-algebra/cayley_dickson_nesting (not exist)]]--

▲図 4.5　複素数→四元数→八元数と入れ子になっていく様子（仮）

この表の内容を、実際にコードを動かして確認してみましょう。

四元数のテスト

まずは、四元数で交換法則が失われることを確認します。`quaternion.ml` は先ほど修正した `Complexical.Make(Complex)` をそのまま使いますが、`to_string` だけは専用のものを書き直します。というのも、`Complexical` の `to_string` は虚部を常に「i」という文字列で表示するようになっており、四元数に使うと i 成分と j 成分がどちらも「i」と表

示されてしまい、区別がつかなくなるからです。

▼リスト 4.25　quaternion.ml

```
include Complexical.Make(Complex)

let term_to_string (c: Rational.t) (label: string): string option =
  if Rational.equal c Rational.zero then None
  else if label = "" then Some (Rational.to_string c)
  else if Rational.equal c Rational.one then Some label
  else if Rational.equal c (Rational.neg Rational.one) then Some ("-" ^ label)
  else Some (Printf.sprintf "%s%s" (Rational.to_string c) label)

let to_string ((a, b): t): string =
  let (r0, r1) = a in
  let (r2, r3) = b in
  let terms = List.filter_map (fun x -> x) [
    term_to_string r0 "";
    term_to_string r1 "i";
    term_to_string r2 "j";
    term_to_string r3 "k";
  ] in
  match terms with
  | [] -> "0"
  | _ -> String.concat " + " terms
```

a, b, c, d の 4 つの実数（有理数）成分を取り出し、それぞれに $1, i, j, k$ というラベルをつけて表示しています。

これで、i と j を実際に作って、積を計算してみます。

▼リスト 4.26　quaternion_test.ml

```
let header = "\\documentclass{article}\\begin{document}"
let footer = "\\end{document}"

let complex_i : Complex.t = (Rational.zero, Rational.one)
let i : Quaternion.t = (complex_i, Complex.zero)
let j : Quaternion.t = (Complex.zero, Complex.one)

let () =
  header |> print_endline;

  Printf.printf "\\[i = %s,\\quad j = %s\\]\n"
    (Quaternion.to_string i) (Quaternion.to_string j);
  Printf.printf "\\[i \\cdot j = %s\\]\n"
    (Quaternion.mul i j |> Quaternion.to_string);
  Printf.printf "\\[j \\cdot i = %s\\]\n"
    (Quaternion.mul j i |> Quaternion.to_string);

  footer |> print_endline;
```

i は複素数の虚数単位をそのまま四元数に埋め込んだもの、j はケーリー＝ディクソン構成で新しく添加された単位です。コンパイルして実行してみましょう。

```
ocamlopt add_monoid.ml mul_monoid.ml add_group.ml mul_group.ml \
euclidean.ml fraction.ml integer.ml rational.ml complexical.ml \
complex.ml quaternion.ml quaternion_test.ml
```

```
./a.out
>>
\documentclass{article}\begin{document}
\[i = i,\quad j = j\]
\[i \cdot j = k\]
\[j \cdot i = -k\]
\end{document}
```

$i \cdot j = k$ である一方、$j \cdot i = -k$ となっており、$i \cdot j \neq j \cdot i$、つまり交換法則が成り立たないことが実際に確認できました。

--[[path = fundamental-algebra/overleaf_5 (not exist)]]--

▲図 4.6　ブラウザ上でチェック

八元数のテスト

次に、八元数で結合法則が失われることを確認します。`octonion.ml` は `Complexical.Make(Quaternion)` を使い、`to_string` は四元数と同様に、8 つの実数成分を $1, e_1, \ldots, e_7$ というラベルで表示するように書き直します。

▼リスト 4.27　octonion.ml

```
include Complexical.Make(Quaternion)

let term_to_string (c: Rational.t) (label: string): string option =
  if Rational.equal c Rational.zero then None
  else if label = "" then Some (Rational.to_string c)
  else if Rational.equal c Rational.one then Some label
  else if Rational.equal c (Rational.neg Rational.one) then Some ("-" ^ label)
  else Some (Printf.sprintf "%s%s" (Rational.to_string c) label)

let to_string ((a, b): t): string =
```

```
  let ((r0, r1), (r2, r3)) = a in
  let ((r4, r5), (r6, r7)) = b in
  let terms = List.filter_map (fun x -> x) [
    term_to_string r0 "";
    term_to_string r1 "e_1";
    term_to_string r2 "e_2";
    term_to_string r3 "e_3";
    term_to_string r4 "e_4";
    term_to_string r5 "e_5";
    term_to_string r6 "e_6";
    term_to_string r7 "e_7";
  ] in
  match terms with
  | [] -> "0"
  | _ -> String.concat " + " terms
```

e_1, e_2, e_3 は四元数の i, j, k をそのまま埋め込んだもの、e_4 はケーリー＝ディクソン構成で新しく添加された単位です。$(e_1 \cdot e_2) \cdot e_4$ と $e_1 \cdot (e_2 \cdot e_4)$ を計算し、比べてみます。

▼リスト 4.28　octonion_test.ml

```
let header = "\\documentclass{article}\\begin{document}"
let footer = "\\end{document}"

let complex_i : Complex.t = (Rational.zero, Rational.one)
let qi : Quaternion.t = (complex_i, Complex.zero)
let qj : Quaternion.t = (Complex.zero, Complex.one)

let e1 : Octonion.t = (qi, Quaternion.zero)
let e2 : Octonion.t = (qj, Quaternion.zero)
let e4 : Octonion.t = (Quaternion.zero, Quaternion.one)

let () =
  header |> print_endline;

  Printf.printf "\\[e_1 = %s,\\quad e_2 = %s,\\quad e_4 = %s\\]\n"
    (Octonion.to_string e1) (Octonion.to_string e2) (Octonion.to_string e4);
  Printf.printf "\\[(e_1 \\cdot e_2) \\cdot e_4 = %s\\]\n"
    (Octonion.mul (Octonion.mul e1 e2) e4 |> Octonion.to_string);
  Printf.printf "\\[e_1 \\cdot (e_2 \\cdot e_4) = %s\\]\n"
    (Octonion.mul e1 (Octonion.mul e2 e4) |> Octonion.to_string);

  footer |> print_endline;
```

```
ocamlopt add_monoid.ml mul_monoid.ml add_group.ml mul_group.ml \
euclidean.ml fraction.ml integer.ml rational.ml complexical.ml \
complex.ml quaternion.ml octonion.ml octonion_test.ml
```

```
./a.out
>>
\documentclass{article}\begin{document}
\[e_1 = e_1,\quad e_2 = e_2,\quad e_4 = e_4\]
\[(e_1 \cdot e_2) \cdot e_4 = e_7\]
\[e_1 \cdot (e_2 \cdot e_4) = -e_7\]
\end{document}
```

$(e_1 \cdot e_2) \cdot e_4 = e_7$ である一方、$e_1 \cdot (e_2 \cdot e_4) = -e_7$ となっており、$(e_1 \cdot e_2) \cdot e_4 \neq e_1 \cdot (e_2 \cdot e_4)$、つまり結合法則が成り立たないことが実際に確認できました。

--[[path = fundamental-algebra/overleaf_6 (not exist)]]--

▲図 4.7　ブラウザ上でチェック

十六元数のテスト

最後に、十六元数で非自明な零因子が現れることを確認します。`hexadecanion.ml` は `Complexical.Make(Octonion)` を使い、`to_string` は 16 個の実数成分を次のようなラベルで表示するように書き直します。

$$1,\ e_1,\ \ldots,\ e_{15} \tag{4.53}$$

▼リスト 4.29　hexadecanion.ml

```
include Complexical.Make(Octonion)

let term_to_string (c: Rational.t) (label: string): string option =
  if Rational.equal c Rational.zero then None
  else if label = "" then Some (Rational.to_string c)
  else if Rational.equal c Rational.one then Some label
  else if Rational.equal c (Rational.neg Rational.one) then Some ("-" ^ label)
  else Some (Printf.sprintf "%s%s" (Rational.to_string c) label)

let to_string ((a, b): t): string =
  let (((r0, r1), (r2, r3)), ((r4, r5), (r6, r7))) = a in
  let (((r8, r9), (r10, r11)), ((r12, r13), (r14, r15))) = b in
  let terms = List.filter_map (fun x -> x) [
    term_to_string r0 "";
    term_to_string r1 "e_1";
    term_to_string r2 "e_2";
    term_to_string r3 "e_3";
    term_to_string r4 "e_4";
    term_to_string r5 "e_5";
    term_to_string r6 "e_6";
```

```
    term_to_string r7 "e_7";
    term_to_string r8 "e_8";
    term_to_string r9 "e_9";
    term_to_string r10 "e_{10}";
    term_to_string r11 "e_{11}";
    term_to_string r12 "e_{12}";
    term_to_string r13 "e_{13}";
    term_to_string r14 "e_{14}";
    term_to_string r15 "e_{15}";
  ] in
  match terms with
  | [] -> "0"
  | _ -> String.concat " + " terms
```

非自明な零因子とは、$a \neq 0$ かつ $b \neq 0$ であるにもかかわらず $a \cdot b = 0$ となってしまうような a, b のことです。十六元数の基底 e_1（八元数部分に埋め込まれた虚数単位）と e_4（新しく添加された単位）を使って、$a = e_1 + e_4$、$b = e_1 - e_4$ という 2 つの元をつくり、その積を計算してみます。

▼リスト 4.30　hexadecanion_test.ml

```
let header = "\\documentclass{article}\\begin{document}"
let footer = "\\end{document}"

let complex_i : Complex.t = (Rational.zero, Rational.one)
let qi : Quaternion.t = (complex_i, Complex.zero)

let oe1 : Octonion.t = (qi, Quaternion.zero)
let oe4 : Octonion.t = (Quaternion.zero, Quaternion.one)

let e1 : Hexadecanion.t = (oe1, Octonion.zero)
let e4 : Hexadecanion.t = (oe4, Octonion.zero)

let a : Hexadecanion.t = Hexadecanion.add e1 e4
let b : Hexadecanion.t = Hexadecanion.sub e1 e4

let () =
  header |> print_endline;

  Printf.printf "\\[a = e_1 + e_4 = %s\\]\n" (Hexadecanion.to_string a);
  Printf.printf "\\[b = e_1 - e_4 = %s\\]\n" (Hexadecanion.to_string b);
  Printf.printf "\\[a \\cdot b = %s\\]\n"
    (Hexadecanion.mul a b |> Hexadecanion.to_string);

  footer |> print_endline;
```

```
ocamlopt add_monoid.ml mul_monoid.ml add_group.ml mul_group.ml \
euclidean.ml fraction.ml integer.ml rational.ml complexical.ml \
complex.ml quaternion.ml octonion.ml hexadecanion.ml hexadecanion_test.ml
```

```
./a.out
>>
\documentclass{article}\begin{document}
\[a = e_1 + e_4 = e_1 + e_4\]
\[b = e_1 - e_4 = e_1 + -e_4\]
\[a \cdot b = 0\]
\end{document}
```

a も b もゼロではないのに、$a \cdot b = 0$ となってしまいました。十六元数では、0 でない数同士をかけても 0 になってしまうことがある、つまり非自明な零因子が存在することが実際に確認できました。これが、十六元数で割り算（乗法の逆元）が定義できない理由です。もし b に逆元が存在するなら、$a \cdot b = 0$ の両辺に右から逆元をかけると、次の式が成り立つはずです。

$$a \cdot b \cdot b^{-1} = 0 \cdot b^{-1} \implies a = 0 \tag{4.54}$$

5
9

しかし実際には $a \neq 0$ なので矛盾します。

--[[path = fundamental-algebra/overleaf_7 (not exist)]]--

▲図 4.8　ブラウザ上でチェック

4.4　ベクトル

この節では、ベクトルを実装していきます。

さて、ベクトルは馴染みのある代数構造かと思います。物理量や座標を表すのにしばしば使ってきたかと思います。ベクトルには、重要な 2 つの性質があります。

- スカラー倍
- ベクトル同士の和

です。

$$3 \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 3 \\ 6 \end{pmatrix} \tag{4.55}$$

ベクトルの要素である**スカラー**（Scalar）に対して、スカラー倍を行っています。一方、

$$\begin{pmatrix} 1 \\ 3 \end{pmatrix} + \begin{pmatrix} 2 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 7 \end{pmatrix} \tag{4.56}$$

これは、それ自身が数として振る舞っており、それぞれのスカラー同士で演算が行われることにより、全体としてベクトル同士の和が行われています。

ここで、ベクトル空間を扱う前に、**加群**（Module）について説明します。加群とは、集合 M（元は「環上のベクトル」のようなものと考えてください）と、環 R（スカラーの集合）の組 $(M, +, \cdot)$ であって、次の性質を満たすもののことです。

- $(M, +)$ が加法群である（M の元同士は足し算・引き算ができる）
- スカラー倍 $R \times M \to M$ が定義されていて、次の法則を満たす

$$\begin{aligned} &\text{スカラーに関する分配律} && (a+b) \cdot M = a \cdot M + b \cdot M \\ &\text{ベクトルに関する分配律} && a \cdot (M_1 + M_2) = a \cdot M_1 + a \cdot M_2 \\ &\text{結合律} && (a \cdot b) \cdot M = a \cdot (b \cdot M) \\ &\text{単位元} && 1 \cdot M = M \end{aligned} \tag{4.57}$$

このとき、$(M, +, \cdot)$ は、環 R 上の**加群**（Module）である、というように表現されます。スカラー R は**環**（Ring）となっています。では、加群の元 M はどうでしょうか。加群の元 M は、加法ができることから、加法群です。しかしながら、今までの代数と違い、M 同士の乗法は定義されていないため、加群は**環**ではありません（加群の元同士を掛け算する演算は、そもそも定義に含まれていません）。

一見、加群はただ単にベクトルを一般的に定義しただけのように思えます。ベクトルは、**加群**（Module）のスカラーが**環**（Ring）から**体**（Field）へと厳しくなった代数のことを指します。数学の用語で、**ベクトル空間**（Vector Space）と言います。通常、ベクトルというのは、体 F 上のベクトル空間 V のことを指します。

ベクトルを使う時、よく使う概念として、**単位ベクトル**（Unit Vector）というものがありました。これは、大きさ（ノルム）が 1 であるベクトルのことです。もとのベクトル

v から単位ベクトルをつくるには、

$$\hat{v} = v/|v| \tag{4.58}$$

のように、v をその大きさ $|v|$（スカラー）で割る必要があります。そのため、ベクトル空間では、スカラーが**環**（Ring）ではなく、**体**（Field）である必要があるのです。

単位ベクトル以外にも、スカラーに除法が考えられることによる次のような嬉しい性質があります。

- 必ず基底が存在する
- ベクトルの次元が考えられる
- スカラーの非自明な零因子が存在しない

もっともシンプルなベクトル空間は、**数ベクトル空間**（Numeric Vector Space）です。数ベクトル空間は、

$$(a_1, a_2, \ldots, a_n) \quad a_i \in \mathbf{F} \tag{4.59}$$

のように、実数の n 個のタプルで表されます。我々が、座標を表したり、物理量を表したりするとき、実数体上の数ベクトル空間を考えていたということです。

また、複素数や、四元数なども実は、それ自体がベクトル空間におけるベクトルとなっています。先ほどつくった複素数を例に出すと、複素数は、有理数体上の 2 次元数ベクトル空間となっています。

$$\text{加法 } (a+bi)+(c+di) = a+c+(b+d)i \iff (a,b)+(c,d) = (a+c, b+d) \tag{4.60}$$

$$\text{スカラー倍 } s\cdot(a+bi) = sa+sbi \iff s\cdot(a,b) = (sa, sb) \tag{4.61}$$

このように、複素数の足し算やスカラー倍は、ベクトルの足し算やスカラー倍と同じように振る舞っていることがわかります。

数ベクトル空間でないベクトル空間の例として、行列があります。

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad a_i \in \mathbf{F} \tag{4.62}$$

行列は、数ベクトル空間とは違い、二次元的なコンテナ構造を持っています。

4.4.1 ベクトル空間ファンクタ

ベクトル空間ファンクタをつくっていきましょう。ベクトル空間ファンクタの役割は、スカラー体 F からベクトル空間 V を生成することです。

では、スカラー体 F からベクトル空間 V を生成するためには、どのような情報が必要でしょうか。スカラー体だけでは、ベクトル空間を生成することはできません。なぜなら、同じ体でも、数ベクトル空間や行列が考えられるように、ベクトル空間は一意に定まらないためです。

そのため、ベクトル空間を定義するには、スカラー体 F に加えて、ベクトル空間のコンテナ構造の情報を渡してあげる必要があります。ベクトル空間ファンクタは、スカラー体 F と、ベクトル空間のコンテナ構造 $\mathbf{C}$ を受け取り、ベクトル空間 V を返すようにします。

ファンクタの引数となる、体 F のシグネチャを作成します。体は一般的なものなので、すでに作成した Field モジュールのシグネチャを使用します。

▼リスト 4.31　field.ml

```
type t
include Add_group.S with type t := t
include Mul_group.S with type t := t
val to_string: t -> string
val equal: t -> t -> bool
```

コンテナのシグネチャを作成します。コンテナは、ベクトル空間の要素を格納するための構造です。

▼リスト 4.32　vector_space.ml

```
module type Container = sig
  type 'a t
  val make: 'a -> 'a t
  val map: ('a -> 'b) -> 'a t -> 'b t
```

```
  val map2: ('a -> 'b -> 'c) -> 'a t -> 'b t -> 'c t
  val for_all2: ('a -> 'b -> bool) -> 'a t -> 'b t -> bool
end
```

- make: コンテナのコンストラクタ
- map: コンテナの要素に対して関数を適用する
- map2: 2 つのコンテナの要素に対して関数を適用する
- for_all2: 2 つのコンテナの要素に対して判定関数を適用し、すべての要素が真であるかを判定する

今回はこの 4 つの関数をコンテナのシグネチャとします。これはただ 1 つの正解があるわけではなく、ベクトル空間をつくる上で必要な関数を定義しているだけです。そのため、他の実装方法もあるかと思いますが、今回はこのような定義となっています。

では、ベクトル空間ファンクタのシグネチャを作成します。

▼リスト 4.33　vector_space.ml

```
module type S = sig
    type t
    type scalar

    val equal: t -> t -> bool
    val zero: t
    val add: t -> t -> t
    val neg: t -> t
    include Add_group.S with type t := t

    val mul_scalar: scalar -> t -> t
    val div_scalar: scalar -> t -> t
end

module type FUNCTOR_TYPE = functor (F: Field.S) (C: Container) -> S
```

Field ではなく Field.S と書いていることに注意してください。Field というのは OCaml の field.ml というファイル名から自動的に決まる**モジュール名**であって、シグネチャ（モジュール型）ではありません。ファンクタの引数の型を制約するときは、モジュール名ではなく、Field.S のようなモジュール型を指定する必要があります。

ファンクタの返り値のシグネチャは、

- 加法群
- スカラー倍

を満たすようになっています。それでは、ファンクタのシグネチャをもとに、ベクトル

空間ファンクタの実装を行います。

▼リスト 4.34　vector_space.ml

```
module Make: FUNCTOR_TYPE = functor (F: Field.S) (C: Container) -> struct
    type t = F.t C.t
    type scalar = F.t
    let equal (x1: t) (x2: t): bool = C.for_all2 F.equal x1 x2
    let zero: t = C.make F.zero
    let add (x1: t) (x2: t): t = C.map2 F.add x1 x2
    let neg (x: t): t = C.map F.neg x

    include Add_group.Extend(struct
      type nonrec t = t
      let zero, add, neg = zero, add, neg
    end)
    let mul_scalar (s: scalar) (v: t): t = C.map (fun x -> F.mul s x) v
    let div_scalar (s: scalar) (v: t): t = C.map (fun x -> F.mul (F.inv s) x) v
end
```

div_scalar は「s で割る」演算ですが、F.div v s のように書いてしまうと、mul_scalar の F.mul s x とは引数の並び順が逆になってしまいます。かわりに F.mul (F.inv s) x と書くことで、どちらも「スカラーに関する値を左に、ベクトルの成分 x を右に置いて F.mul する」という同じ形に揃えています。

これで、ベクトル空間ファンクタが完成しました。

4.4.2　数ベクトル空間

作成したベクトル空間ファンクタを使って、数ベクトル空間を作成してみましょう。スカラー体として、有理数体 **Rational** を使用し、コンテナとして、リストを使用します。次元を指定できるようにするため、数ベクトル空間ファンクタ自体は、次元の情報を受け取るようにします。

▼リスト 4.35　numeric_vector_space.ml

```
module type Dim = sig
  val dim: int
end

module Make (D: Dim) (F: Field.S) = struct
  module Container = struct
    include List
    let make (x: 'a): 'a t =
      List.init D.dim (fun _ -> x)
  end
```

```
  include Vector_space.Make(F)(Container)
end
```

4.4.3 行列

数ベクトル空間ではない、ベクトル空間の例として行列を作っていきます。なお、作成した行列モジュールを、方程式の求解や微分・積分などの、応用として使用することはありませんので、本節を読み進めるのは任意です。

では、作成したベクトル空間ファンクタを使って、行列を作成してみましょう。行列ファンクタは、n 行と m 列の次元を受け取り、n 行 m 列の行列を生成する役割です。シグネチャは次のようになります。

▼リスト 4.36　matrix.ml

```
module type MatrixDim = sig
    val rows: int
    val cols: int
end

module type S = sig
  include Vector_space.S
  val rows: int
  val cols: int
end

module type FUNCTOR_TYPE =
  functor (D: MatrixDim) (F: Field.S) ->
    (S with type scalar = F.t and type t = F.t list list)
```


ここでも Field ではなく Field.S を使っています（理由は先ほどと同じです）。また、S の中では scalar だけでなく t も具体的な型（$F.t$ のリストのリスト）に確定させています。これをせず t を抽象型のままにしてしまうと、後で Matrix.Make の結果を include したときに、外から見える抽象的な t と、内部で実際に使っているリストのリストとしての t が別物として扱われてしまい、型が合わなくなってしまいます。

それでは、行列ファンクタを実装していきます。コンテナには、数ベクトルファンクタと同様に、List を用います。今回は 2 次元のコンテナ構造であるため、2 次元 List を用います。

▼リスト 4.37　matrix.ml

```
module Make: FUNCTOR_TYPE = functor (D: MatrixDim) (F: Field.S) -> struct
  module Container = struct
    type 'a t = 'a list list
    let make (x: 'a): 'a t =
      List.init D.rows (fun _ -> List.init D.cols (fun _ -> x))
    let map (f: 'a -> 'b) (m: 'a t): 'b t =
      List.map (List.map f) m
    let map2 (f: 'a -> 'b -> 'c) (m1: 'a t) (m2: 'b t): 'c t =
      List.map2 (List.map2 f) m1 m2
    let for_all2 (f: 'a -> 'b -> bool) (m1: 'a t) (m2: 'b t): bool =
      List.for_all2 (List.for_all2 f) m1 m2
  end
  include Vector_space.Make(F)(Container)
  let rows = D.rows
  let cols = D.cols
end
```

2 次元のコンテナに相当する Container モジュールを作成し、それをスカラー体 F と共にベクトル空間ファンクタへ渡しています。S で宣言した `rows・cols` も、忘れずに `D.rows・D.cols` として定義しておきます。これで行列ファンクタが完成しました。

長方形行列はベクトル空間としてはこれで十分ですが、行列には乗法が考えられます。行列としての機能をもう少し充実させてみましょう。長方形行列における乗法は、(M 行 K 列) × (K 行 N 列) という場合でのみ考えられます。

$$\begin{pmatrix} a_{11} & \cdots & a_{1k} \\ \vdots & \ddots & \vdots \\ a_{i1} & \cdots & a_{ik} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mk} \end{pmatrix} \begin{pmatrix} b_{11} & \cdots & b_{1j} & \cdots & b_{1n} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ b_{k1} & \cdots & b_{kj} & \cdots & b_{kn} \end{pmatrix} = \begin{pmatrix} c_{11} & \cdots & \cdots & c_{1n} \\ \vdots & \ddots & \vdots & \vdots \\ \cdots & \cdots & c_{ij} & \cdots \\ \vdots & & \vdots & \ddots \\ c_{m1} & \cdots & \cdots & c_{mn} \end{pmatrix} \tag{4.63}$$

長方形行列における乗法は、別の次元同士の行列により定義されます。そのため、同じ代数の中での二項演算ではありません。また、その演算結果も別の次元となるため、$A \times B = C$ における、すべての行列の属する集合が異なることになります。そのため、乗法に関してはモノイドですらないことになります。長方形行列はツールとしては非常に便利ではありますが、少し扱いづらいです。

ここで、正方行列を考えてみましょう。正方行列は（N 行 N 列）×（N 行 N 列) の場合に情報が考えられます。そして、その結果も（N 行 N 列）になります。（N 行 N 列）の行列を考えた時、その代数は乗法に関して閉じているということがわかります。よって、乗法に関するモノイドであるといえます。

行列はベクトル空間であり、ベクトル空間は、加法群でした。つまり、行列は加法群で

す。行列が、加法群であり、乗法モノイドであることから、正方行列は**環**（Ring）となることがわかります。長方形行列では扱いづらかった代数的性質が、正方行列に限定することで、**環**（Ring）という強い制約のある代数として扱えるようになりました。

行列ファンクタを用いて、正方行列ファンクタを作成してみましょう。

正方行列

正方行列を作るには、行列ファンクタの引数に渡していた次元乗法の行と列を一致させればよいです。

シグネチャを作成します。

▼リスト 4.38　"square_matrix.ml"

```
module type Dim = sig
  val dim: int
end

module type S = sig
  include Matrix.S
  val transpose: t -> t
  include Mul_monoid.S with type t := t
end

module type FUNCTOR_TYPE =
  functor (D: Dim) (F: Field.S) ->
    (S with type scalar = F.t and type t = F.t list list)
```


乗法モノイドのシグネチャ `Mul_monoid.S` を `include` するときは、`with type t := t` を付けて、Matrix.S の持つ `t` と同じ型であることを明示しています。これを忘れると、正方行列のシグネチャ `S` の中に、由来の異なる 2 つの `t` が別々に存在することになってしまいます。

実装をします。

▼リスト 4.39　"square_matrix.ml"

```
module Make: FUNCTOR_TYPE = functor (D: Dim) (F: Field.S) -> struct
  include Matrix.Make(struct
    let cols = D.dim
    let rows = D.dim
  end)(F)
  let one =
    List.init D.dim (fun i ->
      List.init D.dim (fun j -> if i = j then F.one else F.zero)
  )
  let transpose m =
```

```
    List.init D.dim (fun i ->
      List.init D.dim (fun j ->
        List.nth (List.nth m j) i
      )
  )
  let mul a b =
    let b_trans = transpose b in
    List.map (fun row_a ->
      List.map (fun col_b ->
        List.fold_left2 (fun acc x y ->
          F.mul x y |> F.add acc
        ) F.zero row_a col_b
      ) b_trans
    ) a
  include Mul_monoid.Extend(struct
    type nonrec t = t
    let one, mul = one, mul
  end)
end
```

mul の実装を見てみましょう。行列の積の (i, j) 成分は、A の i 行目と B の j 列目の内積でした。ここでの行列はリストのリスト（行のリスト）として表現されているため、B の「列」を直接取り出すことができません。そこで、先に transpose で B を転置し、b_trans の行が B の列に対応するようにしてから、A の各行と b_trans の各行（B の各列）について、List.fold_left2 で成分同士の積の和（内積）をとっています。これで、正方行列ファンクタが完成しました。実際にあるスカラー体 F を用いて正方行列を作成してみましょう。今回は、有理数体 $\mathbb{Q}$ と、次元 $N = 3$ を考えます。


```
module Dim3 = struct
  let dim = 3
end
module Mat3 = Square_matrix.Make(Dim3)(Rational)

let of_int_list_list rows =
  List.map (List.map Rational.of_integer) rows
let a = of_int_list_list [[1;2;3]; [4;5;6]; [7;8;9]]
```

Mat3.mul a Mat3.one が a 自身と等しくなることや、単位行列 Mat3.one が対角成分だけ 1 になっていることを確認してみると良い練習になります。

4.5 多項式

この節では多項式をつくっていきます。多項式は、$2x^2 + x - 1$ や $x^4 + 1$ などのように、変数 x の累乗の和によって表されるものでした。多項式は、加法や乗法を行うことができます。

$$(x+3)+(x-2)=2x+1(x+3)\cdot(x-2)=x^2+x-6 \tag{4.64}$$

このことから、多項式は**環**（Ring）であることがわかります。これらの演算において、x という数は本質的に必要ではありません。というのも、x であろうが y であろうが計算結果の意味は変わらないためです。

そのため、多項式をある数字のペアとして考えてみましょう。たとえば、1 次多項式を考えてみます。$ax+b \iff (b,a)$ のような対応関係です。(0 次の係数, 1 次の係数）となっています。

$$\begin{aligned}
(a_1x+b_1)+(a_2x+b_2) &= (b_1,a_1)+(b_2,a_2)\\
&= (b_1+b_2,\ a_1+a_2)\\
\\
(a_1x+b_1)\cdot(a_2x+b_2) &= (b_1,a_1)\cdot(b_2,a_2)\\
&= (b_1b_2,\ a_1b_2+b_1a_2,\ a_1a_2)\\
\\
s\cdot(ax+b) &= s\cdot(b,a)\\
&= (sb,sa)
\end{aligned} \tag{4.65}$$

足し算の結果は 2 つの成分のペアのままですが、掛け算の結果は 3 つの成分になっていることに注目してください。1 次多項式同士を掛けると 2 次多項式になるので、これは自然なことです。多項式の次数は演算によって変わりうる、ということですね。このように、数字のペア同士の演算とした方が、多項式が環であることがわかりやすいですね。

ペア同士の演算や、スカラー倍も行えることから、多項式は**加群**（Module）であることがわかります。

ここで、係数スカラーについて考えます。右辺のペアの中身を見ていると、係数のスカラーは $a_1b_2+b_1a_2$ のように乗法と加法ができる必要があることわかります。そのため、係数スカラーも多項式と同様に**環**（Ring）であることがわかります。ここまでをまとめると、

- スカラー: **環**（Ring）
- 多項式自体: **環**（Ring） かつ 環 R 上の**加群**（Module）

となります。

もう一つ多項式を考えるときに重要なことがあります。それは、値の評価です。ある多項式を考えるとき、何かを代入して値を求めるという操作を行いたいです。$x^2 + 4x + 1$ に $x = 3$ を代入してみると、$3^2 + 4 \cdot 3 + 1 = 22$ といった具合です。

値を評価するとき、係数スカラーとの乗法が必ず発生します。そのため、値を評価するとき、必ず係数スカラーと同じ代数（係数スカラーと二項演算が定義されている代数）に限られます。係数スカラーが整数ならば、評価するときも整数である必要があるということです。

それでは、これまでの議論をもとに実装していきましょう。まず、係数として使える代数が満たすべき性質をシグネチャにします。次に説明する割り算のために、加法・減法・乗法に加えて除法（逆元）まで要求することにします。

▼リスト 4.40　"polynomial.ml"

```
module type Coeff = sig
  type t
  val zero: t
  val add: t -> t -> t
  val neg: t -> t
  val sub: t -> t -> t
  val one: t
  val mul: t -> t -> t
  val inv: t -> t
  val div: t -> t -> t
  val pow: int -> t -> t
  val equal: t -> t -> bool
  val to_string: t -> string
end
```

多項式そのものが満たすべきシグネチャ S も定義しておきます。

▼リスト 4.41　"polynomial.ml"

```
module type S = sig
  type coeff
  type t = coeff list
  val zero: t
  val one: t
  val normalize: t -> t
  val equal: t -> t -> bool
  val degree: t -> int
  val mul: t -> t -> t
  val add: t -> t -> t
  val neg: t -> t
  val eval: t -> coeff -> coeff
  val div_rem: t -> t -> t * t
```

```
  val pow: int -> t -> t
  val sub: t -> t -> t
  val gcd: t -> t -> t
  val ext_gcd: t -> t -> t * t * t
  val leading_coeff: t -> coeff
  val div_scalar: coeff -> t -> t
  val mul_scalar: coeff -> t -> t
  val to_string: t -> string
end
```

gcd・ext_gcd・div_scalar・mul_scalar など、まだ登場していない関数も混ざっていますが、これらは実装しながら順番に説明していきます。

多項式は、係数を低次から並べたリスト C.t list として表現します。たとえば $[c_0; c_1; c_2]$ は $c_0 + c_1 x + c_2 x^2$ を表す、という対応です。

▼リスト 4.42　"polynomial.ml"

```
module Make (C: Coeff) = struct
  type t = C.t list
  type coeff = C.t
  let zero: t = []
  let one: t = [C.one]
  let normalize (x: t): t =
    let rec remove_leading_zeros = function
      | [] -> []
      | c :: cs ->
        if C.equal c C.zero then remove_leading_zeros cs
        else c :: cs
    in
    List.rev x |> remove_leading_zeros |> List.rev

  let equal (x1: t) (x2: t): bool =
    let p1 = normalize x1 in
    let p2 = normalize x2 in
    let rec aux l1 l2 =
      match l1, l2 with
      | [], [] -> true
      | [], _ | _, [] -> false
      | c1 :: cs1, c2 :: cs2 -> C.equal c1 c2 && aux cs1 cs2
    in
    aux p1 p2

  let degree (p: t) : int =
    match normalize p with
    | [] -> -1
    | p' -> List.length p' - 1

  let leading_coeff (p: t): C.t =
    match List.rev p with
    | [] -> C.zero
    | x::xs -> x
```

normalize は、リストの末尾（最高次の係数）に並ぶ 0 を取り除く関数です。たとえば $[1;0;0]$（$1+0x+0x^2$ のつもり）と $[1]$（1）は、どちらも同じ多項式 1 を表していますが、リストとしては別物です。normalize によってこの表現の自由度をなくし、equal などで正しく比較できるようにしています。degree（次数）が -1 のときは、多項式全体が 0 であることを表します。

次に乗法を実装します。

▼リスト 4.43　"polynomial.ml"

```
let mul (x1: t) (x2: t): t =
  let p1 = normalize x1 in
  let p2 = normalize x2 in
  if p1 = [] || p2 = [] then []
  else
    let res_len = List.length p1 + List.length p2 - 1 in
    let res = Array.make res_len C.zero in
    List.iteri (fun i c1 ->
      List.iteri (fun j c2 ->
        res.(i + j) <- C.add res.(i + j) (C.mul c1 c2)
      ) p2
    ) p1;
    Array.to_list res |> normalize
```

係数リスト $[c_0;c_1;\ldots]$ が $c_0+c_1x+c_2x^2+\cdots$ を表すので、2 つの多項式の積の k 次の係数は、

$$\sum_{i+j=k} c_i d_j \tag{4.66}$$

という**畳み込み**（Convolution）になります。コードでは、2 重の List.iteri によって、i 次の係数 c_i と j 次の係数 d_j の積を、結果の配列の $i+j$ 次の場所に足し込むことで、この畳み込みをそのまま計算しています。

加法・否定・評価はいたってシンプルです。

▼リスト 4.44　"polynomial.ml"

```
let rec add (x1: t) (x2: t): t =
  match x1, x2 with
  | [], l | l, [] -> l
  | c1 :: cs1, c2 :: cs2 -> C.add c1 c2 :: add cs1 cs2

let neg (x: t): t =
```

```
    List.map C.neg x

  let eval (xs: t) (x: C.t): C.t =
    List.mapi (fun i c -> C.mul c (C.pow i x)) xs
    |> List.fold_left (fun acc s -> C.add acc s) C.zero
```

eval は、係数 c_i に x^i を掛けて足し合わせる、$\sum_i c_i x^i$ という式をそのまま実装したものです。x の型が係数と同じ C.t に限られているのは、この計算に必ず係数同士の乗法（C.mul、C.pow の内部で使われます）が発生するためでしたね。

> 実は、最初に書いたコードは List.mapi (fun i s -> C.pow i s) xs となっており、引数 x を一切使わずに「係数 c_i の i 乗の和」を計算してしまっていました。後述のテストで eval を実際に実行してみたところ、想定と違う値が出てきたことでこのバグに気づきました。C.pow の引数の順序（n 乗する対象が第 2 引数）を勘違いしていたのが原因です。

最後に、多項式を画面に表示するための to_string を実装しておきます。低次から並んだ係数リストのうち、0 でない項だけを取り出し、$c_i x^i$ の形に整形して連結します。

▼リスト 4.45　"polynomial.ml"

```
let term_to_string (c: C.t) (power: int): string option =
  if C.equal c C.zero then None
  else if power = 0 then Some (C.to_string c)
  else
    let var = if power = 1 then "x" else Printf.sprintf "x^{%d}" power in
    if C.equal c C.one then Some var
    else Some (Printf.sprintf "%s %s" (C.to_string c) var)

let to_string (p: t): string =
  let terms =
    normalize p
    |> List.mapi (fun i c -> term_to_string c i)
    |> List.filter_map (fun x -> x)
    |> List.rev
  in
  match terms with
  | [] -> "0"
  | _ -> String.concat " + " terms
```

係数が 1 のときは係数部分を省略し、x^1 は x と表示するなど、複素数や四元数の to_string で使ったのと同じ考え方です。項は高次から低次の順に並べ替えてから連結して

います。

ここまでで加法・乗法がそろったので、Add_group・Mul_monoid のファンクタを適用して、引き算などを自動生成しておきましょう。

▼リスト 4.46　"polynomial.ml"

```
include Mul_monoid.Extend(struct
  type nonrec t = t
  let one, mul = one, mul
end)

include Add_group.Extend(struct
  type nonrec t = t
  let add, neg = add, neg
end)
```

続いて、多項式の割り算 div_rem を実装します。商と余りを同時に求める関数です。

▼リスト 4.47　"polynomial.ml"

```
let div_rem (num: t) (den: t) : t * t =
  let den_norm = normalize den in
  if den_norm = [] then failwith "Division by zero polynomial"
  else
    let lc_den = List.nth den_norm (List.length den_norm - 1) in
    let deg_den = degree den_norm in
    let rec aux rem quo_acc =
      let rem_norm = normalize rem in
      let deg_rem = degree rem_norm in
      if deg_rem < deg_den then
        (normalize quo_acc, rem_norm)
      else
        let lc_rem = List.nth rem_norm deg_rem in
        let factor = C.div lc_rem lc_den in
        let shift = deg_rem - deg_den in
        let term =
          List.init (shift + 1)
            (fun i -> if i = shift then factor else C.zero)
        in
        let new_rem = sub rem_norm (mul term den_norm) in
        let new_quo = add quo_acc term in
        aux new_rem new_quo
    in
    aux num []
```

やっていることは、筆算による多項式の割り算そのものです。割られる式 num（rem）の最高次の項を、割る式 den の最高次の項でちょうど打ち消せるような 1 項（term）を求め、それを引いて次数を下げる、という操作を、余りの次数が割る式の次数を下回るまで繰り返しています。term の係数 factor は、両者の最高次係数（lc_rem と lc_den）

の商です。

係数の代数に div まで要求していた理由はここにあります。整数のような環では商が割り切れるとは限りません（たとえば 3 を 2 で割り切ることはできません）が、係数が体であれば、0 でない数では必ず割り切れます。

割り算さえ手に入れば、整数のときとまったく同じ手順で最大公約数が計算できます。

▼リスト 4.48　"polynomial.ml"

```
include Euclidean.Extend(struct
  type nonrec t = t
  let zero, one, sub, mul, div_rem, equal =
    zero, one, sub, mul, div_rem, equal
end)
```

整数の節で紹介した Euclidean モジュールは、商と余りをまとめて返す div_rem さえ受け取れば、最大公約数 gcd に加えて**拡張ユークリッドの互除法**（ext_gcd）まで導出してくれるのでした。多項式の割り算はちょうど商と余りを同時に返すコードだったので、このモジュールがそのまま使えます。gcd・ext_gcd が具体的にどう役立つのかは、有限体の節で詳しく見ていきます。

最後に、多項式が係数体上の加群でもあったことを思い出しましょう。ベクトル空間ファンクタでつくった mul_scalar・div_scalar を、そのまま多項式にも使いたいところです。

▼リスト 4.49　"polynomial.ml"

```
  include (Vector_space.Make(C)(struct
    include List
    let make x = []
  end) : sig
    type t
    val mul_scalar: C.t -> t -> t
    val div_scalar: C.t -> t -> t
  end with type t := t)
end
```

Vector_space.Make を丸ごと include してしまうと、そちらの equal や add が後勝ちで上書きされてしまいます。ベクトル空間側の equal は固定長のコンテナを前提にしているため、次数の異なる多項式同士を比較すると例外になってしまうのです。欲しいのはあくまで「多項式は係数体上の加群でもある」という性質から得られる mul_scalar・div_scalar だけなので、シグネチャで絞り込んで必要な関数だけを取り込んでいます。

これで、任意の係数体 C から多項式環をつくるファンクタが完成しました。

4.5.1 テスト

係数体に有理数を使って、実際に動かしてみましょう。

▼リスト 4.50　polynomial_test.ml

```
module P = Polynomial.Make(Rational)

let header = "\\documentclass{article}\\begin{document}"
let footer = "\\end{document}"

let test_biop f f_name x1 x2 =
  Printf.sprintf "\\[\\mathrm{%s}(%s, %s) = %s\\]"
    f_name (x1 |> P.to_string) (x2 |> P.to_string) (f x1 x2 |> P.to_string)
  |> print_endline

let () =
  header |> print_endline;

  let p1 : P.t =
    [Rational.of_integer 1; Rational.of_integer 2; Rational.of_integer 3]
  in
  let p2 : P.t = [Rational.of_integer 1; Rational.of_integer 1] in

  Printf.printf "\\[p_1 = %s,\\quad p_2 = %s\\]\n"
    (P.to_string p1) (P.to_string p2);
  test_biop P.add "add" p1 p2;
  test_biop P.mul "mul" p1 p2;
  let (q, r) = P.div_rem p1 p2 in
  Printf.printf "\\[\\mathrm{div\\_rem}(%s, %s) = (%s,\\ %s)\\]\n"
    (P.to_string p1) (P.to_string p2) (P.to_string q) (P.to_string r);
  Printf.printf "\\[\\mathrm{eval}\\left(%s, 2\\right) = %s\\]\n"
    (P.to_string p1) (P.eval p1 (Rational.of_integer 2) |> Rational.to_string);
```

```
footer |> print_endline;
```

$p_1 = 1 + 2x + 3x^2$、$p_2 = 1 + x$ として、和・積・割り算・評価を確認します。コンパイルして実行してみましょう。

```
ocamlopt add_monoid.ml mul_monoid.ml add_group.ml mul_group.ml \
euclidean.ml integer.ml fraction.ml rational.ml field.ml \
vector_space.ml polynomial.ml polynomial_test.ml
```

```
./a.out
>>
\documentclass{article}\begin{document}
\[p_1 = 3 x^{2} + 2 x + 1,\quad p_2 = x + 1\]
\[\mathrm{add}(3 x^{2} + 2 x + 1, x + 1) = 3 x^{2} + 3 x + 2\]
\[\mathrm{mul}(3 x^{2} + 2 x + 1, x + 1) = 3 x^{3} + 5 x^{2} + 3 x + 1\]
\[\mathrm{div\_rem}(3 x^{2} + 2 x + 1, x + 1) = (3 x + -1,\ 2)\]
\[\mathrm{eval}\left(3 x^{2} + 2 x + 1, 2\right) = 17\]
\end{document}
```

$(1 + 2x + 3x^2)(1 + x) = 1 + 3x + 5x^2 + 3x^3$、$(1 + 2x + 3x^2) = (x + 1)(3x - 1) + 2$、$1 + 2x + 3x^2$ に $x = 2$ を代入した値は $1 + 4 + 12 = 17$ と、いずれも手計算と一致しています。先ほど見つけた `eval` のバグは、まさにこの最後の値が合わなかったことで発覚したものでした。

--[[path = fundamental-algebra/overleaf_8 (not exist)]]--

▲図 4.9　ブラウザ上でチェック

4.6　有限体

次に、有限体を作成していきます。

4.6.1　位数 p の有限体

要素の数が有限であるような体を考えてみましょう。もっとも身近な例は、整数を素数 p で割った余りの集合です。0 以上 p 未満の整数の集合に対して、加法と乗法をそれぞれ

p で割った余りとして定義します。このようにしてできる体を**有限体**（Finite Field）（あるいは**ガロア体**（Galois Field））と呼び、$\mathbb{F}_p$ と書きます。

$\mathbb{F}_p$ が本当に体であるためには、0 以外のすべての元に乗法の逆元が存在しなければなりません。ここで p が素数であることが効いてきます。仮に p が合成数、たとえば $p = 6$ だったとすると、

$$2 \times 3 = 6 \equiv 0 \pmod 6 \tag{4.67}$$

となり、どちらも 0 でない元同士の積が 0 になってしまいます。これでは 2 や 3 に逆元をもたせることはできません。p が素数であれば、0 でない任意の元 x は p と互いに素になるので、**フェルマーの小定理**（Fermat's Little Theorem）

$$x^{p-1} \equiv 1 \pmod p \quad (x \neq 0) \tag{4.68}$$

が成り立ちます。両辺を x で割れば、x を $p-2$ 乗した値が x の乗法の逆元であることがわかりますね。

うれしいことに、累乗はすでに整数の節で**乗法モノイド**としてつくってあります。逆元を求めるためにわざわざ拡張ユークリッドの互除法を持ち出す必要はなく、累乗するだけで求まってしまうわけです。

それでは実装していきましょう。位数 p をファンクタの引数として受け取れるようにします。

▼リスト 4.51　galois_field.ml

```
module type Prime = sig
  val p: int
end
```

$\mathbb{F}_p$ の元は 0 以上 p 未満の整数として表現し、加法・乗法のたびに p で正規化します。

▼リスト 4.52　galois_field.ml

```
module Make (P: Prime) = struct
  type t = int
  let zero: t = 0
  let one: t = 1
  let normalize (x: t): t =
```

```
    let r = x mod P.p in
    if r < 0 then r + P.p else r
  let add (x1: t) (x2: t): t =
    x1 + x2 |> normalize
  let mul (x1: t) (x2: t): t =
    x1 * x2 |> normalize
  let neg (x: t): t =
    -x |> normalize
  let to_string (x: t): string =
    normalize x |> string_of_int
  let equal x1 x2 = (normalize x1) = (normalize x2)

  include Add_group.Extend(struct
    type nonrec t = t
    let add, neg = add, neg
  end)
  include Mul_monoid.Extend(struct
  type nonrec t = t
    let one, mul = one, mul
  end)
  let inv (x: t): t =
    pow (P.p - 2) x
  include Mul_group.Extend(struct
    type nonrec t = t
    let one, mul, inv = one, mul, inv
  end)
end
```

7
9

`inv x = pow (P.p - 2) x` のところに注目してください。整数や有理数のときは逆元を「引き算」や「分数の反転」として素朴に定義していましたが、ここでは累乗という別の操作から逆元を導いています。同じ**乗法群**（Mul_group）のシグネチャを満たしていても、逆元の作り方は代数ごとにまったく異なってよい、というのがよくわかる例ですね。

これで、任意の素数 p に対する有限体 $\mathbb{F}_p$ ができました。

4.6.2 テスト

フェルマーの小定理から逆元を導く `inv` が実際に正しく動くか、$\mathbb{F}_5$ のすべての非零元で確認してみましょう。

▼リスト 4.53　galois_field_test.ml

```
module F5 = Galois_field.Make(struct let p = 5 end)

let header = "\\documentclass{article}\\begin{document}"
let footer = "\\end{document}"

let () =
```

```
header |> print_endline;

for x = 1 to 4 do
  let inv_x = F5.inv x in
  Printf.printf "\\[\\mathrm{inv}(%s) = %s,\\quad %s \\cdot %s = %s\\]\n"
    (F5.to_string x) (F5.to_string inv_x)
    (F5.to_string x) (F5.to_string inv_x) (F5.to_string (F5.mul x inv_x))
done;

footer |> print_endline;
```

```
ocamlopt add_monoid.ml mul_monoid.ml add_group.ml mul_group.ml \
euclidean.ml galois_field.ml galois_field_test.ml
```

```
./a.out
>>
\documentclass{article}\begin{document}
\[\mathrm{inv}(1) = 1,\quad 1 \cdot 1 = 1\]
\[\mathrm{inv}(2) = 3,\quad 2 \cdot 3 = 1\]
\[\mathrm{inv}(3) = 2,\quad 3 \cdot 2 = 1\]
\[\mathrm{inv}(4) = 4,\quad 4 \cdot 4 = 1\]
\end{document}
```

$1, 2, 3, 4$ のどの元についても、inv との積がきちんと 1 になっています。

--[[path = fundamental-algebra/overleaf_9 (not exist)]]--

▲図 4.10　ブラウザ上でチェック

4.6.3　多項式による有限体の構成

$\mathbb{F}_p$ は、整数の集合 $\mathbb{Z}$ を素数 p で割った余りの集合として構成しました。同じ発想を、数の代わりに多項式に対して使うとどうなるでしょうか。

整数における素数とは、「それ以上分解できない数」のことでした。多項式の世界でこれに対応するのが**既約多項式**（Irreducible Polynomial）です。既約多項式とは、係数体の中で 2 つ以上の低次の多項式の積に分解できない多項式のことです。整数を素数で割った余りをとると体ができたように、多項式環を既約多項式で割った余りをとっても体ができます。

具体的には、係数体 K 上の多項式環 $K[x]$ を、既約多項式 $f(x)$ で割った余りの集合として $K[x]/(f(x))$ を考えます。この操作でできる体は、もとの体 K を拡張した体になっており、これを**拡大体**（Field Extension）と呼びます。

さっそく実装していきましょう。既約多項式による拡大体ファンクタに必要なものは、係数体 C と、その係数体上の多項式環 P、そして割る対象となる既約多項式 $poly$ です。

▼リスト 4.54　galois_poly_field.ml

```
module type IrreduciblePoly = sig
  module C : Polynomial.Coeff
  module P : Polynomial.S with type coeff = C.t
  val poly : P.t
end
```

ファンクタの実装は次のようになります。

▼リスト 4.55　galois_poly_field.ml

```
module Make (I: IrreduciblePoly) = struct
  include I.P
  let normalize (p: t): t =
    div_rem p I.poly |> snd
  let equal (x: t) (y: t): bool =
    equal (normalize x) (normalize y)
  let add (x: t) (y: t): t =
    add x y |> normalize
  let neg (x: t): t =
    neg x |> normalize
  let mul (x: t) (y: t): t   =
    mul x y |> normalize
  let inv (x: t): t =
    let x_norm = normalize x in
    if equal x_norm I.P.zero then
      failwith "Division by zero in extension field"
    else
      let (g, u, _) = ext_gcd x_norm I.poly in
      let lc = I.P.leading_coeff g in
      let u_monic = div_scalar lc u in
      normalize u_monic
  let to_string (x: t): string =
    normalize x |> I.P.to_string

  include Add_group.Extend(struct
      type nonrec t = t
      let add, neg = add, neg
    end)
  include Mul_monoid.Extend(struct
    type nonrec t = t
    let one, mul = one, mul
  end)
```

```
    include Mul_group.Extend(struct
      type nonrec t = t
      let one, mul, inv = one, mul, inv
    end)
  end
```

to_string は、正規化した多項式をそのまま I.P.to_string に渡すだけです。拡大体 K の元は係数体上の多項式そのものなので、多項式の節でつくった to_string をそのまま流用できます。

加法・乗法は、多項式としての演算をしたあとに normalize（既約多項式で割った余りをとる）をかけるだけです。位数 p の有限体のときと同じ構図ですね。

注目したいのは inv です。今度はフェルマーの小定理のような近道は使えません。かわりに、整数の節で最大公約数を求めるのに使った、あの互除法が再登場します。ext_gcd は**拡張ユークリッドの互除法**（Extended Euclidean Algorithm）で、二つの元 a, b に対して

$$g = u \cdot a + v \cdot b \tag{4.69}$$

を満たす最大公約数 g と係数 u, v を同時に求めます。$f(x)$ が既約で x が 0 でなければ、x と $f(x)$ の最大公約数は（定数倍を除いて）1 になります。つまり

$$g = u(x) \cdot x + v(x) \cdot f(x) \tag{4.70}$$

の両辺を $f(x)$ で割った余りをとると、右辺の $v(x) \cdot f(x)$ の項は消えて

$$g \equiv u(x) \cdot x \pmod{f(x)} \tag{4.71}$$

となります。g は定数（0 次多項式）なので、$u(x)$ を g で割ってスケールを直してやれば、それがそのまま x の逆元です。コード中の div_scalar lc u がまさにこの「スケールを直す」操作にあたります。

これで、既約多項式さえ用意すれば、どんな係数体からでも拡大体をつくれるファンクタが手に入りました。

4.6.4 デモ: $K/\sqrt{2}$ をつくる

このファンクタの一番シンプルな使い道を試してみましょう。有理数体 $\mathbb{Q}$ 上の多項式 $x^2 - 2$ は、有理数の範囲では分解できない既約多項式です（分解できるとしたら $\sqrt{2}$ が有理数ということになってしまいます）。この既約多項式で拡大体をつくってみます。

▼リスト 4.56　root2_field.ml

```
module Root2Poly = struct
  module C = Rational
  module P = Polynomial.Make(C)
  let poly = [C.of_integer (-2); C.zero; C.one] (* x^2 - 2 *)
end

module K = Galois_poly_field.Make(Root2Poly)
```

K.t の中身は Rational.t list、つまり有理数体 $\mathbb{Q}$ 上のベクトル（2 つの成分の組）です。x に対応する元を取り出して、$(\sqrt{2})^2$ と、$\sqrt{2}$ にその逆元をかけた値を実際に計算させてみましょう。

▼リスト 4.57　root2_field_test.ml

```
let header = "\\documentclass{article}\\begin{document}"
let footer = "\\end{document}"

let () =
  header |> print_endline;

  let root2 : Root2_field.K.t = [Rational.zero; Rational.one] in
  Printf.printf "\\[\\sqrt 2 = %s\\]\n" (Root2_field.K.to_string root2);
  Printf.printf "\\[\\left(\\sqrt 2\\right)^2 = %s\\]\n"
    (Root2_field.K.mul root2 root2 |> Root2_field.K.to_string);
  Printf.printf "\\[\\sqrt 2 \\cdot \\left(\\sqrt 2\\right)^{-1} = %s\\]\n"
    (Root2_field.K.mul root2 (Root2_field.K.inv root2)
     |> Root2_field.K.to_string);

  footer |> print_endline;
```

```
ocamlopt add_monoid.ml mul_monoid.ml add_group.ml mul_group.ml \
euclidean.ml integer.ml fraction.ml rational.ml field.ml \
vector_space.ml polynomial.ml galois_poly_field.ml \
root2_field.ml root2_field_test.ml
```

```
./a.out
>>
\documentclass{article}\begin{document}
\[\sqrt 2 = x\]
\[\left(\sqrt 2\right)^2 = 2\]
\[\sqrt 2 \cdot \left(\sqrt 2\right)^{-1} = 1\]
\end{document}
```

K.mul root2 root2 は確かに 2 になりました。

$$x^2 \equiv 2 \pmod{x^2 - 2} \tag{4.72}$$

なのですから当然ですね。同様に K.mul root2 (K.inv root2) も単位元 K.one に等しくなり、きちんと逆元が求まっていることも確認できました。

--[[path = fundamental-algebra/overleaf_10 (not exist)]]--

▲図 4.11　ブラウザ上でチェック

つまり K の元は、$a, b \in \mathbb{Q}$ を使って $a + b\sqrt{2}$ という形の数と同じ振る舞いをします。これがまさに拡大体であり、$\mathbb{Q}$ の上に $\sqrt{2}$ という新しい元を 1 つ添加してできた体という意味で $\mathbb{Q}(\sqrt{2})$、あるいは $K/\sqrt{2}$ のように書かれます。既約多項式による拡大体ファンクタに $x^2 - 2$ を渡しただけで、平方根という新しい概念をもった体を、既存の部品だけからつくれてしまったわけです。

4.6.5　タワー拡大の限界

$\sqrt{2}$ が手に入ったので、次は $\sqrt{2}$ と $\sqrt{3}$ の両方を含む体がほしくなったとします。同じやり方でいけそうです。さきほどつくった $K = \mathbb{Q}(\sqrt{2})$ を係数体として、$K[y]/(y^2 - 3)$ をつくればよいわけです。Galois_poly_field.Make の引数の係数体に、今度は Rational ではなく K を渡すだけですね。

このように、拡大体の上にさらに拡大体を重ねていくやり方を**タワー拡大**（Tower Extension）と呼びます。ですが、この調子で $\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{7}$、と扱う素数を増やしていくとどうなるでしょうか。

まず、次元が爆発します。$\mathbb{Q}$ に対して $\sqrt{2}$ を 1 つ添加すると次元は 2 倍になり（$1, \sqrt{2}$ が基底です）、さらに $\sqrt{3}$ を添加するとまた 2 倍になります。n 個の平方根を 1 つずつタワー状に積み上げると、最終的な次元は 2^n です。10 個の素数を扱いたいだけで、次元は 1024 にもなってしまいます。

型の面でも扱いにくくなります。`Galois_poly_field.Make` をタワー状に何度も適用するということは、素数の数だけモジュールをネストさせるということです。素数を 1 つ増やすたびに型そのものが変わってしまうので、「有理数に p_1 から p_n までの平方根を添加した体」を、n を実行時に決めながら 1 つの型として扱うことができません。

さらに、添加する順序にも本来意味がないはずです。数学的には $\mathbb{Q}(\sqrt{2}, \sqrt{3})$ も $\mathbb{Q}(\sqrt{3}, \sqrt{2})$ も同じ体ですが、タワー拡大では「先に $\sqrt{2}$ を添加してから $\sqrt{3}$ を添加した体」と「先に $\sqrt{3}$ を添加してから $\sqrt{2}$ を添加した体」は、コード上は別の型としてつくられてしまいます。

こうした問題を解決するには、1 つずつ根を積み上げるのではなく、扱いたい素数の集合をまとめて 1 つの体として一度に表現してしまうアプローチが必要です。次章では、この「並列な拡大」を扱う `Q_rootp` モジュールを見ていきます。

--[[path = fundamental-algebra/tower_vs_parallel (not exist)]]--

▲図 4.12　タワー拡大と並列拡大の比較（仮）

第 5 章

代数を応用する

5.1　多項式方程式

方程式を解くとは、$f(x) = 0$ を満たす x を求めることです。整数、有理数、複素数、そして拡大体まで、これまで作り上げてきた代数の力を使って、いよいよ多項式方程式を解いていきましょう。

5.1.1　数値的に解く、代数的に解く

「方程式を解く」といっても、そのアプローチにはいくつかの段階があります。まずは、もっとも素朴な方法である**数値的に解く**（Numerically Solve）方法から見ていきましょう。

ニュートン法（Newton's Method）をご存知の方も多いと思います。適当な初期値 x_0 から出発し、

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{5.1}$$

という更新式を繰り返すことで、解に近づけていく手法です。汎用性が高く（多項式に限らずほとんどどんな関数にも適用できます）、実装も比較的簡単です。

しかし、得られるのはあくまで $0.618033988\ldots$ のような近似的な数値であり、その数値がどのような式から生まれたのか、という構造の情報は失われてしまいます。

一方、本書がここまで目指してきたのは**代数的に解く**（Algebraically Solve）ことです。たとえば 2 次方程式であれば、

$$ax^2 + bx + c = 0 \implies x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5.2}$$

のように、四則演算と累乗根だけを使って、厳密な解を式として書き下すことです。近似ではなく厳密解が手に入り、しかもその解がどのような数（有理数の四則演算と平方根から作られた数、など）であるかという構造も同時にわかります。

5.1.2 「代数的に解く」の 3 段階

実は、「代数的に解く」という言葉ひとつをとっても、その実現の仕方には深さの異なる 3 つの段階があります。

1. 公式に数値を当てはめてレンダリングするだけ

もっとも素朴なのは、2 次方程式の解の公式のような既知の公式に具体的な数値を代入し、その結果を文字列として組み立てて表示するだけのアプローチです。$x^2 - 3x + 2 = 0$ であれば $a = 1, b = -3, c = 2$ を解の公式にそのまま当てはめ、

$$x = \frac{3 \pm \sqrt{9 - 8}}{2} \tag{5.3}$$

という文字列を組み立てて表示する、というイメージです。この段階では、計算結果は「表示するための文字列」でしかなく、根号の中身が完全平方数になっていても（$9 - 8 = 1$ のように）簡約すら行われません。

2. 文字列パターンマッチングによる正規化・簡約

そこで、根号の中身が完全平方になっている式を見つけたら簡約する、というルールを、文字列（あるいは**式木**（Expression Tree））へのパターンマッチングとして追加していきます。$x + 0$ を x に書き換える、といった単純なものから、平方根の中身を簡約するような少し複雑なものまで、ルールをたくさん用意し、式が変化しなくなるまで繰り返し適用するアプローチです。素朴な数式処理システムの多くは、突き詰めるとこのような書き換え規則の集合として実装されています。

実際の計算の流れを、具体的な式で見てみましょう。$\sqrt{4} + x \cdot 0$ という式は、次のような式木として保持されます。

- 根：$+$
- 左の子：平方根ノード（子：4）

- 右の子：×（左の子：x、右の子：0）

書き換えルールを、木の末端に近いところから順に適用していきます。

1. 「$a \times 0 \to 0$」というルールが、右の子の部分木 $x \times 0$ にマッチするので、そこを 0 に置き換えます。木全体は $\sqrt{4} + 0$ になります
2. 「$a + 0 \to a$」というルールが、根 + の部分木全体にマッチするので、木全体を左の子 $\sqrt{4}$ に置き換えます
3. 「$\sqrt{n} \to k$（n が完全平方数 k^2 のとき）」というルールが、残った $\sqrt{4}$ にマッチするので、2 に置き換えます

--[[path = application/expr_tree_rewrite (not exist)]]--

▲図 5.1　式木がルール適用ごとに書き換わる様子（仮）

これ以上どのルールもマッチしなくなった時点で書き換えは終了し、最終的な結果 2 が得られます。このように、式全体を 1 つの木として保持しておくことで、「木のどの部分がどのルールにマッチするか」を機械的に探索できるようになります。段階 1 の「公式に数値を代入して文字列を組み立てるだけ」のアプローチとの違いは、結果を表示用の文字列としてではなく、再帰的な構造を持つデータ（木）として保持し、その構造に対してルールを繰り返し適用できる点にあります。

実用上はかなり強力ですが、書き換えルールに漏れがあれば簡約に失敗し（今回の例で言えば、$\sqrt{n} \to k$ のルールを用意し忘れていれば $\sqrt{4}$ は簡約されないまま残ります）、逆に誤ったルールがあれば数学的に正しくない簡約をしてしまう、という危うさと常に隣り合わせです。

3. 代数の世界そのものを構築し、その中で計算する

本書がここまで一貫して取り組んできたのは、この 3 番目の道です。環・体・拡大体といった代数構造そのものを OCaml の型として構築し、四則演算（や共役、累乗根）を、その型に対して正しく定義された演算として実装してきました。この方法で方程式を解くと、得られる解は文字列ではなく、ちゃんと**定義された代数構造の値**になります。

これの何が嬉しいかというと、得られた解に対して、**そのままさらに計算を続けられる**ことです。解 x が本当に $f(x) = 0$ を満たすか検算したければ、その解を実際に多項式へ代入し、同じ体の中で計算すればよいだけです。2 の「文字列の書き換え」のアプローチでは、書き換えを繰り返した末に出てくるのは表示用の文字列であり、そこから先の計算を続けるには、もう一度その文字列を構文解析し直す必要があります。3 のアプローチ

では、解は最初から計算可能な値そのものなので、そのような手間は必要ありません。

5.1.3 実際の CAS はどうしているか

それでは、実際に広く使われている数式処理システム（**CAS**（Computer Algebra System））は、方程式をどのように解いているのでしょうか。

SymPy や **Mathematica**、**WolframAlpha** のようなシステムは、多くの場合、次数の低い方程式に対しては、本章で扱うカルダノの公式やフェラーリの公式（4 次方程式用）に相当するロジックを内部に持っていて、それを適用して厳密解を返します。

次数が 5 以上になると、アーベル・ルフィニの定理により一般には根号で解が表せないため、そのままでは根号で表現できない解を `RootOf` や `Root` といった特別なシンボル（「この多項式の i 番目の根」という意味を持つ、それ以上分解されない記号）として扱ったり、必要に応じて数値近似へ切り替えたりします。内部的には、多項式の因数分解や、グレブナー基底のような代数計算のアルゴリズムが駆使されています。

これらのシステムの多くは、式を木構造として保持し、その上でルールベースの簡約や因数分解を行うという、先ほどの段階 2 に近いアプローチを土台にしています。

その中で少し毛色が異なるのが **Axiom** です。Axiom は、**圏**（Category）と**ドメイン**（Domain）という仕組みによって、「環」「体」「ユークリッド整域」といった代数的な性質を型システムの中で表現し、その型に対して安全に演算を組み立てられるように設計された CAS です。本書がここまで OCaml のファンクタとモジュール型を使って環・体・拡大体を積み上げてきたアプローチは、発想としてはこの Axiom の設計思想にかなり近いものです。段階 3 の「代数の世界そのものを構築する」というアプローチを、型システムの力を借りて実践している、と言い換えることもできるでしょう。

本書はこのあと、1 次方程式・2 次方程式・3 次方程式までを実装します。なぜここで止めるのかについては、実際に手を動かしたあとの方がずっと納得しやすいので、理由は章の最後にまとめて説明します。

5.1.4 二次方程式の解全体がつくる体

3 次方程式の前に、まず 2 次方程式を代数的に解くとはどういうことかを確認しておきます。$ax^2 + bx + c = 0$ の解は、おなじみの解の公式

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{5.4}$$

で求まります。四則演算はすでに有理数体 $\mathbb{Q}$ の中で閉じているので、この公式を実行

するのに足りないのは、判別式 $b^2 - 4ac$ の平方根だけです。

そこで、有理数係数のあらゆる 2 次方程式の解を表現できる体 K_1 を考えます。判別式 $b^2 - 4ac$ は任意の有理数になりうるので、K_1 には

- 判別式が正の場合に備えて、あらゆる素数 p の平方根 $\sqrt{p}$
- 判別式が負の場合に備えて、虚数単位 i（$i^2 = -1$ を満たす数）

の両方が必要になりそうです。任意の有理数の平方根は素因数分解すれば素数の平方根の積として表せるので、「あらゆる素数の平方根」さえ用意しておけば十分です。両方を合わせると、

$$\mathbb{Q}(i, \sqrt{p_1}, \sqrt{p_2}, \ldots) \tag{5.5}$$

という体になります。

しかし、本書ではこの i を K_1 には含めないことにします。理由は、このあとのカルダノの公式で、判別式の符号によらず必ず 1 の原始三乗根 ω（これ自体が虚数です）を使うことになるからです。虚数単位を 2 次方程式の時点と 3 次方程式の時点とで 2 回別々に導入するのは冗長なので、

$$K_1 = \mathbb{Q}(\sqrt{p_1}, \sqrt{p_2}, \ldots) \tag{5.6}$$

という、正の素数の平方根だけを集めた「実数の範囲にとどまる拡大体」を K_1 と呼ぶことにし、虚数単位はあとで ω を添加するタイミングでまとめて導入します。

5.1.5 三次方程式を解くために必要なもの

それでは、実際に 3 次方程式を解くために必要な実装を、順を追って見ていきましょう。全体の流れは次のとおりです。

1. まず、案外多くの 3 次方程式が持っている「有理数の解」を、公式を使わずに探してみます
2. 判別式の平方根を手に入れるために、有理数体にいくつもの平方根を同時に添加した体 K_1 を構成します
3. カルダノの公式と還元不能な場合の両方に必要な、1 の原始三乗根 ω を K_1 に添加します

4. 判別式の符号に応じて 2 通りに分岐しながら、実際にカルダノの公式を適用し、3 つの解を求めます

この 4 ステップを、順番に実装していきます。

--[[path = application/field_extension_tower (not exist)]]--

▲図 5.2　有理数体から K1、K1(ω) への拡大の様子（仮）

5.1.6　有理数解を探す

まずは、拡大体の出番が来る前に、案外多くの 3 次方程式が持っている「有理数の解」を探すところから始めましょう。これは**有理数根定理**（Rational Root Theorem）と呼ばれる定理を使います。

整数係数の多項式

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = 0 \tag{5.7}$$

が有理数解 $x = \frac{p}{q}$（p, q は互いに素）を持つならば、p は a_0 の約数、q は a_n の約数でなければならない、というのが有理数根定理です。つまり、有理数解の「候補」は、定数項と最高次係数の約数の組み合わせとして、有限個に絞り込めるのです。あとは、候補を 1 つずつ多項式に代入して、実際に 0 になるかを確かめるだけです。

実装していきましょう。まず、有理数係数の多項式を整数係数に変換する関数をつくります。分母の最小公倍数をすべての係数にかければ、係数を整数化できます。

▼リスト 5.1　"q_poly.ml"

```
include Polynomial.Make(Rational)

let to_integer_poly (p : t) : Integer.t list * Integer.t * Integer.t =
  let denoms = List.map snd p in
  let common_denom = List.fold_left Integer.lcm Integer.one denoms in
  let int_coeffs =
    List.map (fun (num, den) ->
      Integer.div_rem (Integer.mul num common_denom) den |> fst
    ) p
  in
  let a0 = List.hd int_coeffs in
  let an = List.hd (List.rev int_coeffs) in
  (int_coeffs, a0, an)
```

係数を整数化した上で、定数項 a0 と最高次係数 an を取り出しています（多項式は低次から並んでいるリストだったので、List.hd が定数項、リストを逆順にした先頭が最高次係数になります)。

次に、ある整数の約数をすべて（正負とも）列挙する関数です。

▼リスト 5.2　"q_poly.ml"

```
let divisors (n : Integer.t) : Integer.t list =
  let abs_n = Integer.abs n in
  if Integer.equal abs_n Integer.zero then []
  else
    let rec aux curr acc =
      if Integer.compare (Integer.mul curr curr) abs_n > 0 then acc
      else if
        Integer.equal (Integer.div_rem abs_n curr |> snd) Integer.zero
      then
        let div = Integer.div_rem abs_n curr |> fst in
        if Integer.equal curr div then
          curr :: acc
        else
          curr :: div :: acc
      else aux (Integer.add curr Integer.one) acc
    in
    let pos_divs = aux Integer.one [] in
    List.concat_map (fun d -> [d; Integer.neg d]) pos_divs
```


n の平方根まで試し割りをして正の約数を求めたあとに、符号を反転したものも加えています。

最後に、これらを組み合わせて有理数解を探す solve_equation を実装します。

▼リスト 5.3　"q_poly.ml"

```
let rec solve_equation (p : t) : Rational.t list =
  if degree p < 1 then []
  else
    let (_, a0, an) = to_integer_poly p in
    if Integer.equal a0 Integer.zero then
      Rational.zero ::
        solve_equation (div_rem p [Rational.zero; Rational.one] |> fst)
    else
      let p_divs = divisors a0 in
      let q_divs =
        divisors an
        |> List.filter (fun d -> Integer.compare d Integer.zero > 0)
      in
      let candidates =
        List.concat_map (fun q ->
          List.map (fun p_val -> (p_val, q)) p_divs
```

```
      ) q_divs
      |> List.sort_uniq Rational.compare
  in
  List.filter
    (fun c -> Rational.equal (eval p c) Rational.zero) candidates
```

定数項が 0 であれば、$x = 0$ が明らかに解の 1 つです。その場合は、多項式を x（[Rational.zero; Rational.one]）で割った残りの多項式に対して、再帰的に同じ処理を行います。

定数項が 0 でなければ、有理数根定理どおり、a0 の約数（正負とも）を分子、an の正の約数を分母とする候補を総当たりで作ります（分母を正の約数に限定しているのは、符号は分子側の正負がすべて担っているため、重複を避けるためです）。あとは、それぞれの候補を実際に多項式へ代入し（eval）、0 になるものだけを残せば完成です。

この solve_equation は、これから何度も再利用します。3 次方程式のカルダノの公式の中でも、平方根や立方根を求める際に「この数はもしかして有理数として書けないか」を確かめるために使われることになります。

5.1.7 K1: 複数の平方根を同時に添加する

93

先ほど、正の素数の平方根だけを集めた拡大体 K_1 を考える、というところまで確認しました。これを実際に実装していきましょう。

有限体の節では、$K/\sqrt{2}$ のような単純な例をタワー拡大でつくり、素数を増やすたびに次元が 2^n で爆発したり、型がネストして扱いにくくなったりする問題を確認しました。ここでは、その反省を踏まえた「並列な拡大」を実装します。

アイデアはこうです。有限個の素数からなる集合 S（たとえば p_1 から p_k までの素数の集合）を選ぶごとに、その積の平方根、

$$\sqrt{p_1 p_2 \cdots p_k} \tag{5.8}$$

という「基底」が 1 つ決まります。K_1 の元は、こうした基底の有理数係数の線形結合、

$$\sum_{S} c_S \sqrt{\textstyle\prod_{p \in S} p} \quad (c_S \in \mathbb{Q}) \tag{5.9}$$

として表現できます。素数の集合 S を「タワーの階層」ではなく「ベクトルの基底」として扱うことで、n 個の素数があっても、1 つの型の中に収まるというわけです。

実装していきましょう。基底となる素数の集合には Set を、集合から係数への対応には Map を使います。

▼リスト 5.4　"q_rootp.ml"

```
module PrimeSet = Set.Make(Integer)
module Coeff = Rational

module BasisMap = struct
  include Map.Make(PrimeSet)
  let make (v : 'a) : 'a t =
    singleton PrimeSet.empty v
  let map = map
  let map2 (f : 'a -> 'b -> 'c) (m1 : 'a t) (m2 : 'b t) : 'c t =
    merge (fun _ o1 o2 ->
      match o1, o2 with
      | Some v1, Some v2 -> Some (f v1 v2)
      | _ -> invalid_arg "BasisMap.map2: maps have different domain/basis"
    ) m1 m2
  let for_all2 (p : 'a -> 'b -> bool) (m1 : 'a t) (m2 : 'b t) : bool =
    if cardinal m1 <> cardinal m2 then false
    else
      for_all (fun k v1 ->
        match find_opt k m2 with
        | Some v2 -> p v1 v2
        | None -> false
      ) m1
end

type t = Coeff.t BasisMap.t
let zero: t = BasisMap.empty
let one: t = BasisMap.singleton PrimeSet.empty Coeff.one
```

PrimeSet が素数の集合、BasisMap が「素数の集合」から「係数」への対応です。空の素数集合 PrimeSet.empty は、何も掛かっていない（$\sqrt{1} = 1$ の）基底なので、有理数そのものを表します。単位元 one が BasisMap.singleton PrimeSet.empty Coeff.one になっているのはそのためです。

加法はいたって素直です。同じ基底同士の係数を足し、係数が 0 になった基底は取り除きます。

▼リスト 5.5　"q_rootp.ml"

```
let add (v1 : t) (v2 : t) : t =
  BasisMap.merge (fun _ key1 key2 ->
    match key1, key2 with
    | None, None -> None
    | Some c, None | None, Some c -> Some c
    | Some c1, Some c2 ->
        let sum = Coeff.add c1 c2 in
```

```
        if Coeff.equal sum Coeff.zero then None
        else Some sum
  ) v1 v2

let neg (v : t) : t =
  BasisMap.map Coeff.neg v
```

問題は乗法です。2 つの基底となる素数の集合 S_1 と S_2 をもとに、それぞれの平方根同士を掛けたとき、その結果はどの基底に乗るのでしょうか。ここで使うのが、平方根の基本的な性質、

$$\sqrt{a} \cdot \sqrt{a} = a, \qquad \sqrt{a} \cdot \sqrt{b} = \sqrt{ab}\ (a \neq b) \tag{5.10}$$

です。**両方に共通して含まれる素数**は、掛け合わさることで根号の外に出て、ただの整数（有理数）になります。**片方にしか含まれない素数**は、根号の中に残ります。つまり、

$$\sqrt{\textstyle\prod_{S_1}} \cdot \sqrt{\textstyle\prod_{S_2}} = \left(\textstyle\prod_{p \in S_1 \cap S_2} p\right) \cdot \sqrt{\textstyle\prod_{p \in S_1 \triangle S_2} p} \tag{5.11}$$

というわけです（$S_1 \triangle S_2$ は**対称差**（Symmetric Difference）、つまり「どちらか片方にだけ含まれる要素の集合」を表します）。コードにすると、次のようになります。

▼リスト 5.6　"q_rootp.ml"

```
let mul_basis (s1 : PrimeSet.t) (s2 : PrimeSet.t) (c1 : Coeff.t) (c2 : Coeff.t)
  : Coeff.t * PrimeSet.t =
  let common = PrimeSet.inter s1 s2 in
  let diff = PrimeSet.union (PrimeSet.diff s1 s2) (PrimeSet.diff s2 s1) in
  let factor_int =
    PrimeSet.fold (fun p acc -> Integer.mul acc p) common Integer.one
  in
  let new_coeff = Coeff.mul (Coeff.mul c1 c2) ((factor_int, Integer.one)) in
  (new_coeff, diff)

let add_term (c : Coeff.t) (s : PrimeSet.t) (acc : t) : t =
  if Coeff.equal c Coeff.zero then acc
  else
    let current_c =
      BasisMap.find_opt s acc |> Option.value ~default:Coeff.zero
    in
    let sum_c = Coeff.add current_c c in
    if Coeff.equal sum_c Coeff.zero then
      BasisMap.remove s acc
    else
      BasisMap.add s sum_c acc
```

```
let mul (v1 : t) (v2 : t) : t =
  BasisMap.fold (fun s1 c1 acc1 ->
    BasisMap.fold (fun s2 c2 acc2 ->
      let (new_c, new_s) = mul_basis s1 s2 c1 c2 in
      add_term new_c new_s acc2
    ) v2 acc1
  ) v1 zero
```

mul は、v_1 の各基底と v_2 の各基底のすべての組み合わせについて mul_basis を計算し、結果を足し合わせているだけです（多項式の乗法の畳み込みと同じ形ですね）。

たとえば $\sqrt{2}$ 同士を掛けるなら、共通部分は素数 2 だけ、対称差は空集合なので、係数 2・基底「空集合（有理数）」、つまりただの 2 になります。$\sqrt{2}$ と $\sqrt{3}$ を掛けるなら、共通部分が空なので、そのまま基底は 2 と 3、つまり $\sqrt{6}$ になります。

最後に、逆元です。これが少しだけ技巧的です。

▼リスト 5.7　"q_rootp.ml"

```
let collect_primes (v : t) : PrimeSet.t =
  BasisMap.fold (fun s _ acc -> PrimeSet.union s acc) v PrimeSet.empty

let split_by_prime (p : Integer.t) (v : t) : t * t =
  BasisMap.fold (fun s c (a, b) ->
    if PrimeSet.mem p s then
      let s' = PrimeSet.remove p s in
      (a, add_term c s' b)
    else
      (add_term c s a, b)
  ) v (zero, zero)

let rec inv (v : t) : t =
  if BasisMap.is_empty v then
    failwith "Division by zero"
  else
    let primes = collect_primes v in
    if PrimeSet.is_empty primes then
      let c = BasisMap.find PrimeSet.empty v in
      BasisMap.singleton PrimeSet.empty (Coeff.inv c)
    else
      let p = PrimeSet.min_elt primes in
      let a, b = split_by_prime p v in
      let sqrt_p = BasisMap.singleton (PrimeSet.singleton p) Coeff.one in
      let conj = sub a (mul b sqrt_p) in
      let norm = mul v conj in
      mul conj (inv norm)
```

発想は、有理数の分母の有理化と同じです。v に登場する素数のうち 1 つ p に注目し、$v = a + b\sqrt{p}$（a, b には p を含まない）という形に分解します。分母を有理化するときに $a - b\sqrt{p}$ を掛けたのと同じように、v に「共役」$a - b\sqrt{p}$ を掛けると、

$$v \cdot (a - b\sqrt{p}) = a^2 - b^2 p \tag{5.12}$$

となって、p がきれいに消えます。これを `norm` とすると、

$$v^{-1} = \frac{a - b\sqrt{p}}{\mathrm{norm}} = (a - b\sqrt{p}) \cdot \mathrm{norm}^{-1} \tag{5.13}$$

です。`norm` はもとの v より登場する素数が 1 つ少ないので、`inv norm` を再帰的に呼び出せば、素数の数だけ再帰が進み、いつか素数が 1 つもない（ただの有理数の）状態にたどり着いて止まります。

これで、有限個の平方根を自由に組み合わせられる体 K_1 ができました。

ちなみに、このモジュールには `cbrt`（立方根）も用意されています。中身は、「有理数根定理で立方根の候補となる有理数を絞り込み（前節の `Q_poly.solve_equation` をまさにここで再利用しています）、候補を実際に 3 乗してもとの数と一致するか検算する」という、有理数解を探したときとまったく同じ戦略でできています。カルダノの公式を実装する際にまた登場するので、覚えておいてください。

97

5.1.8 K1(ω): 1 の原始三乗根を添加する

K_1 で判別式の平方根は手に入りましたが、カルダノの公式にはもう 1 つ、**1 の原始三乗根**（Primitive Cube Root of Unity）ω が必要です。ω は $\omega^3 = 1$ かつ $\omega \neq 1$ を満たす数で、$x^2 + x + 1 = 0$ の解でもあります（$x^3 - 1 = (x-1)(x^2+x+1)$ なので）。先ほど K_1 に含めるのを保留にした虚数単位 i は、ここで ω の一部として、ようやく登場することになります。

ωをどう添加するか

ω は虚数（複素数）です。虚数が必要なら、K_1 をつくったときと同じ要領で、-1 を素数のかわりに使い、-1 の平方根、つまり虚数単位 i を並列拡大の基底に混ぜてしまえばよいのでは、と思うかもしれません。

しかし、それはできません。$\omega \cdot i \cdot \sqrt{3}$ の 3 つは、

$$\omega = -\frac{1}{2} + \frac{i\sqrt{3}}{2} \tag{5.14}$$

という関係で結ばれていて、独立ではないからです。この式から、i と $\sqrt{3}$ があれば ω が、逆に ω と $\sqrt{3}$ があれば i が、それぞれ導けます。つまり、この 3 つのうち独立に添加してよいのは 2 つまでで、残り 1 つは自動的についてくるのです。

もし先に i を K_1 へ添加してしまっていたら、そのあとで ω を独立な新しい元として添加することはできません。$\sqrt{3}$ はすでに K_1 の中にあるので、ω は i と $\sqrt{3}$ の組み合わせとして、もう表現できてしまっているからです。

そこで本書では、$\sqrt{3}$（K_1 の中の、素数 3 の平方根として）と ω の 2 つを独立な生成元として採用し、i の方は必要になったときに ω と $\sqrt{3}$ から導く、という設計にします。

カルダノの公式では、3 次方程式の 3 つの解を、1 つの立方根 α から

$$y_1 = \alpha + v, \quad y_2 = \alpha\omega + v\omega^2, \quad y_3 = \alpha\omega^2 + v\omega \tag{5.15}$$

という形でまとめて作り出します。ω を掛けるたびに解が 1 つずつ「回転」していく、というイメージです。

この ω を添加するのに、有限体の節でつくった `Galois_poly_field` がそのまま使えます。$x^2 + x + 1$ は有理数の範囲で既約な多項式なので、K_1 上でこれによる拡大体をつくれば、それがそのまま $K_1(\omega)$ です。

▼リスト 5.8　"omega_field.ml"

```
module OmegaPoly = struct
  module C = Q_rootp
  module P = Polynomial.Make(C)
  let poly = [C.one; C.one; C.one]
end

include Galois_poly_field.Make(OmegaPoly)

let omega: t = [Q_rootp.zero; Q_rootp.one]
```

既約多項式 $x^2 + x + 1$ を渡すだけで拡大体ができてしまうのは、$K/\sqrt{2}$ のデモとまったく同じ構図ですね。ω 自身は、この多項式の x に対応する元、`[Q_rootp.zero; Q_rootp.one]` です。

ω が手に入ったことで、実はうれしいことがもう 1 つあります。

$$\omega = -\frac{1}{2} + \frac{i\sqrt{3}}{2} \tag{5.16}$$

なので、ω 自身が虚部を持った数です。ここから i を

$$i = \frac{2\omega + 1}{\sqrt{3}} \tag{5.17}$$

のように ω だけで表せます。つまり、ω さえ添加しておけば、虚数単位 i を別に用意しなくても、マイナスの数の平方根が扱えるのです。

実際、判別式が負になるケース（3 つの実数解を持つ、還元不能な場合）の平方根 `sqrt_disc` は、この事実を使って実装されています。

▼リスト 5.9　"omega_field.ml"

```
let sqrt_disc (disc : Rational.t) : t =
    let num, den = disc in
    if Integer.equal num Integer.zero then
      zero
    else if Integer.compare num Integer.zero > 0 then
      let root_d = Q_rootp.of_rational_sqrt disc in
      of_ab root_d Q_rootp.zero
    else
      let abs_disc = (Integer.abs num, den) in
      let abs_disc_over_3 = Rational.div abs_disc (Rational.of_integer 3) in
      let k = Q_rootp.of_rational_sqrt abs_disc_over_3 in
      let a0 = k in
      let a1 = Q_rootp.mul (Q_rootp.of_rational (2, 1)) k in
      of_ab a0 a1
```

判別式 `disc` が正であれば、そのまま K_1 の中で平方根を取るだけです（`Q_rootp.of_rational_sqrt`）。

負であれば、D を $-|D|$（絶対値）と見て、先ほどの i の式を使い、$\sqrt{D}$ を $k + 2k\omega$ という ω の 1 次式に書き換えています。ここで k は $|D|$ を 3 で割った値の平方根です。実際に展開して確かめてみると、

$$(k + 2k\omega)^2 = k^2(1 + 4\omega + 4\omega^2) = k^2(3 + 4(\omega + \omega^2 + 1) - 4) = -3k^2 = -|D| \tag{5.18}$$

（$\omega^2 + \omega + 1 = 0$ を使いました）となって、たしかに $-|D|$ の平方根になっていることがわかりますね。

`Omega_field` にも `cbrt` が用意されていて、K_1 のときとまったく同じ「有理数根定理で候補を絞り、3 乗して検算する」という戦略で立方根を探します。カルダノの公式では、この `cbrt` が成功するかどうかが、次に説明する 2 つのパターンの分岐点になります。

5.1.9 三次方程式を解く

道具がそろいました。いよいよ、3 次方程式

$$ax^3 + bx^2 + cx + d = 0 \tag{5.19}$$

を実際に解いていきます。

チルンハウス変換

まず、

$$x = y - \frac{b}{3a} \tag{5.20}$$

という置き換えを行うと、2 次の項が消え、

$$y^3 + py + q = 0 \tag{5.21}$$

という、より単純な形（**デプレスト 3 次方程式**（Depressed Cubic））に変形できます。この変換自体は既存の多項式の演算だけで求まる、ただの代数計算です。

▼リスト 5.10　"cardano_solver.ml"

```
let compute_depressed_pq (poly : Q_poly.t) : Rational.t * Rational.t =
  let d = List.nth poly 0 in
  let c = List.nth poly 1 in
  let b = List.nth poly 2 in
  let a = List.nth poly 3 in
  let three = (3, 1) in
  let p =
    Rational.div
      (Rational.sub (Rational.mul three (Rational.mul a c)) (Rational.mul b b))
      (Rational.mul three (Rational.mul a a))
  in
  let two = (2, 1) in
  let nine = (9, 1) in
  let twenty_seven = (27, 1) in
  let num =
    Rational.add
      (Rational.sub
         (Rational.mul two (Rational.pow 3 b))
         (Rational.mul nine (Rational.mul a (Rational.mul b c))))
```

```
      (Rational.mul twenty_seven (Rational.mul (Rational.mul a a) d))
  in
  let den = Rational.mul twenty_seven (Rational.pow 3 a) in
  let q = Rational.div num den in
  (p, q)
```

カルダノの公式

デプレスト 3 次方程式 $y^3 + py + q = 0$ の解は、次の u を使って表せます。

$$u = \sqrt[3]{-\frac{q}{2} + \sqrt{\left(\frac{q}{2}\right)^2 + \left(\frac{p}{3}\right)^3}} \tag{5.22}$$

このルートの中身（判別式に相当します）の平方根を求めるのに `Omega_field.sqrt_disc` を、立方根を求めるのに `Omega_field.cbrt` を、それぞれ使います。

u が求まれば、

$$v = -\frac{p}{3u} \tag{5.23}$$

として、3 つの解が

$$y_1 = u + v, \quad y_2 = u\omega + v\omega^2, \quad y_3 = u\omega^2 + v\omega \tag{5.24}$$

と求まります（もとの方程式の解は

$$x_k = y_k - \frac{b}{3a} \tag{5.25}$$

です）。

ここで問題になるのが、`Omega_field.cbrt` が必ず成功するとは限らないことです。`cbrt` が成功するかどうかで、実装は 2 つのパターンに分かれます。

パターン A: cbrt が成功する場合

u がそのまま `Omega_field.t` の元として求まるので、上の式にそのまま代入して 3 つの解を計算するだけです。

パターン B: cbrt が失敗する場合（還元不能な場合）

これは、3 つの解がすべて実数であるにもかかわらず、$K_1(\omega)$ の中には立方根が存在しない、という状況です。仕方がないので、u の立方根を表す新しい記号 α を、形式的に添加してしまいます。

これがまさに、有限体の節でつくった `Galois_poly_field` の出番です。既約多項式 $x^3 - u$ による拡大体をつくれば、その中には（定義により）$\alpha^3 = u$ を満たす α が存在します。

▼リスト 5.11　"cubic.ml"

```
module type Field = sig
  val u : Omega_field.t
end

module Make (F : Field) = struct
  module I = struct
    module C = Omega_field
    module P = Polynomial.Make(C)
    let poly =
      [ Omega_field.neg F.u; Omega_field.zero;
        Omega_field.zero; Omega_field.one ]
  end
  include Galois_poly_field.Make(I)
end
```

`poly` が $[-u; 0; 0; 1]$、つまり低次から並べて $-u + 0x + 0x^2 + 1x^3 = x^3 - u$ になっていることに注目してください。既約多項式さえ切り替えれば、$K/\sqrt{2}$ のときと同じ仕組みで、平方根ではなく立方根を添加した拡大体がつくれてしまうのです。

この拡大体の中で、α を使ってパターン A と同じ式で 3 つの解を計算すれば、還元不能な場合でも、きちんと代数的な解が手に入ります。

2 つの型を 1 つにまとめる

ここで、少し困ったことに気づきます。パターン A の解は `Omega_field.t` 型の値として、パターン B の解は `Cubic.Make(F).t` 型の値として求まり、**両者の型が違う**のです。呼び出す側は、方程式の係数を実際に見るまでどちらのパターンになるかわからないので、返り値の型をコンパイル時に 1 つへ決め打ちすることができません。

実装では、これを**ファーストクラスモジュール**（First-Class Module）（実行時に決まる型を、値として受け渡しできるようにする OCaml の仕組み）で切り抜けています。

▼リスト 5.12　"cardano_solver.ml"

```
module type ROOT_SET = sig
  module C : sig
    type t
    val zero : t
    val one : t
    val add : t -> t -> t
    val sub : t -> t -> t
    val mul : t -> t -> t
    val inv : t -> t
    val to_string : t -> string
  end
  val roots : C.t list
end
```

解を求める関数の返り値の型を（module ROOT_SET）としておけば、パターン A でもパターン B でも、この ROOT_SET というシグネチャさえ満たしていれば、具体的な型が何であっても同じ型として返せます。

おもしろいのは、パターン A の側でも、あえてダミーの拡大（u = Omega_field.one という、最初から立方根が求まっている体）を経由させ、Cubic.Make の結果として解を包み直していることです。こうして両方のパターンを「同じ形」のモジュールとして扱えるように、実装側でひと工夫しているわけです。

どちらのパターンでも、最終的に得られる解は、文字列ではなく、**れっきとした代数構造の値**です。この解をさらに他の式に代入したり、四則演算を続けたりできることこそが、本章の冒頭で述べた「代数の世界そのものを構築する」アプローチの成果です。

5.1.10　なぜ三次方程式までなのか

さて、ここまで三次方程式を解く実装を進めてきたので、冒頭で保留にしていた「なぜ本書は三次方程式までしか実装しないのか」を説明します。

アーベル・ルフィニの定理と還元不能な場合

まず、数学的な事実を 2 つ確認しておきます。

1 つ目は**アーベル・ルフィニの定理**（Abel-Ruffini Theorem）です。5 次以上の**一般の**方程式には、四則演算と累乗根だけを使った解の公式が存在しないことが証明されています（特別な形の 5 次方程式が根号で解ける場合はありますが、**すべての** 5 次方程式に通用する公式は存在しません）。

2 つ目は、ここまでの実装でも顔を出した**還元不能な場合**（Casus Irreducibilis）です。3 つの解がすべて実数であっても、カルダノの公式を素直に適用すると、計算の途中でど

うしても複素数（1 の原始三乗根 ω）を経由しなければならない場合があります。

この 2 つは、本書がここまで積み上げてきた「拡大体を構成的に作っていく」アプローチが、数学的にどこまで届くかを決める話です。

しかし、これだけでは「なぜ 4 次方程式はやらないのか」の説明にはなりません。4 次方程式には**フェラーリの公式**（Ferrari's Formula）という、カルダノの公式と同じように根号だけで解を書き下せる公式が存在するからです。3 次方程式で止まる本当の理由は、次に説明する「静的な型でどこまで代数を表現できるか」という、実装に近いところにあります。

基底が「素数」だから安全だった

`q_rootp.ml` でつくった拡大体が、有限個の平方根をいくつでも自由に組み合わせられたのは、基底として**素数**を使ったからでした。異なる素数 $p_1, \ldots, p_n$ に対して、それぞれの平方根は必ず互いに線形独立になる（つまり、どの 1 つの平方根も、残りの平方根の有理数係数の組み合わせでは決して表せない）ことが、数論的に保証されています。だからこそ、「素数の集合をキーとする Map」というたった 1 つの静的な型だけで、新しい素数がいくつ増えても、線形独立性のチェックを一切行わずに安全に拡張し続けられたのです。

三次方程式以降は、その保証がなくなる

ところが、1 の原始三乗根 ω や、カルダノの公式で登場する立方根については、この「素数だから安全」という保証が使えません。新しく添加しようとしている数が、実はそれまでに積み上げてきた拡大体の中の元の和や積として、すでに表現できてしまっている可能性があるからです。それを確かめるには、その都度、それまで拡大してきた体すべてに対して、本当に線形独立かどうかを検証しなければなりません。これは、有限体の節の「タワー拡大の限界」で触れた、まさにタワー状の拡大の問題です。

実際、先ほどの「2 つの型を 1 つにまとめる」で見た（`module ROOT_SET`）は、まさにこの問題が顔を出した瞬間でした。パターン A とパターン B とでは、本来できあがる拡大体の「形」が違うため、コンパイル時には 1 つの型に決め打ちできず、ファーストクラスモジュールという実行時の仕組みを借りて、どうにか型の形だけを揃えていたのでした。3 次方程式を 1 つ解くだけでも、すでに静的な型の枠組みからはみ出しかけているのです。

これより先、4 次方程式や、複数の拡大体をまたいだ計算に進もうとすると、こうした工夫だけでは足りなくなり、線形独立性を実行時に判定し、拡大体の構造そのものを動的に組み立て直すような仕組みが必要になります。それは、本書がここまで大事にしてきた「代数構造を OCaml の型としてそのまま表現する」というアプローチの範囲を超えてし

まいます。3 次方程式までを実装する、というのは、この意味での限界に対応しているのです。

著者紹介

第 1 章 ひつじ / @mhidaka

ひつじだよ〜

関数型言語 OCaml で作る代数計算機

2026 年 4 月 11 日　技術書典 20 版 v1.0.0

著　者　icchon 編
編　集　icchon
発行所　未定