

プログラミング言語 Lune

コンパイラが教えてくれる、遅延評価の関数型言語

Hiroshi Koide

Lune v0.1

目次

	序章 Lune へようこそ	3
第I部 ひとめぐり		
第1章	やさしい入門	6
第II部 言語を学ぶ		
第2章	値・型・式	18
第3章	関数	29
第4章	遅延評価 — Lune の心臓部	38
第5章	代数的データ型とパターンマッチ	53
第6章	レコード	66
第7章	null 安全	73
第8章	リストとストリーム	83
第9章	命令的に書く — var・while・for・IO	92
第10章	モジュール	100
第III部 コンパイラと道具		
第11章	エラーと対話する	108
第12章	REPL・整形・検査	117
第13章	プログラムを組み立てる — ケーススタディ	126
	付録A 言語リファレンスマニュアル	136
	付録B 標準ライブラリリファレンス	146
	付録C 診断コード一覧	153
	付録D CLI・REPL コマンドリファレンス	157
	付録E Lune の設計と、これから	161
	索引	167

章の中の節までは載せていません。HTML 版では左の目次から、PDF 版ではしおりからたどれます。

序章 Lune へようこそ

放課後の部室に、こう書いた紙が貼ってあります。

部訓: 必要になるまで、やらない。一度やったことは、忘れない。

楽そうな部だと思って入ったら、これが評価戦略の話だと知らされる —— というのがこの言語のリポジトリに書いてある入部初日の風景です。この本は、その続きです。

部室に来られなくても大丈夫なように、教わることを一冊にまとめました。先輩たちは出てきませんが、**彼らの役割は全部コンパイラが持っています**。

Lune とは何か — Lazy と Native

名前の由来から。

Lazy は評価戦略のことです。Lune の値は、必要になるまで計算されません。 `let` と書いた時点では何も起きず、その値を実際に使う瞬間まで待ちます。一度計算したら覚えていて、二度は計算しません。部訓のとおりです。

Native は.....ネイティブコードのことではありません。**native language**、**母語**のことです。Lune のコンパイラは、エラーとその解説を日本語で話します。

```
$ lune --check guide.lune
error[TYP0001]: 未定義の名前: cont
--> guide.lune:2:13
  |
2 | let total = cont + 5
  |               ^^^^ この名前は定義されていない
    = hint: もしかして `count` ですか?
    = help: 詳しくは `lune explain TYP0001 --lang ja` を実行してください
```

英語のエラーメッセージを読み解く労力を、プログラミングそのものを学ぶ労力に回してほしい —— それがこの言語の出発点です。英語で読みたければ `--lang en` で切り替わります。

この本の態度 — エラーは教材です

部長の台詞をひとつだけ借ります。

うちの部では、エラーは赤点じゃなくて教材。

これは気の利いた言い方ではなく、設計方針です。Lune の診断は、**何が起きたか、どこで起きたか、次に何をすればいいか**の3つを必ず持っています。上の例なら、`^^^^` が場所を、`= hint:` が次の一手を、`= help:` がもっと詳しい解説への入口を示しています。

全31個の診断コードには、それぞれ「意味・再現する最小の例・直し方」の解説が用意されています。`lune explain TYP0001` と打てば読めます。**エラー番号は覚えるものではなく、調べるための鍵**です。

だからこの本では、**わざとエラーを出します**。各章に「壊してみよう」というコーナーがあります。

壊してみよう — REPL で `1 + true` と打ってみてください。何が返ってきましたか。

うまくエラーが出たら成功です。**エラーが出たら、正解です**。

読み方

REPL を開きながら読んでください。これがいちばん大事なお願いです。

```
$ lune
Lune v0.1.0 REPL. Type :help or :quit.
lune>
```

インストールが面倒なら、ブラウザだけで動く [Playground](#) があります。処理系がそのままブラウザの中で動くので、何も入れずに始められます。REPL も入っています。

本書に載っている出力は、**すべて実際に処理系を動かして採ったものです**。同じものが手元でも出ます。出ないときは、この本が間違っているか、あなたが何か面白いものを見つけたかのどちらかです。

- **第I部（第1章）** — まず1周。細かい規則は後まわしにして、書いて動かして壊します。
- **第II部（第2～10章）** — 言語の中身。値と型、関数、**遅延評価（第4章）**、ADT と match、レコード、null 安全、リストとストリーム、命令的な書き方、モジュール。
- **第III部（第11～13章）** — コンパイラと道具。診断の読み方、REPL と整形、そして総合演習のケーススタディ。
- **付録A～E** — 引くための本。文法、標準ライブラリ、診断コード一覧、CLI/REPL、そして設計の話。

急いでいる人向けに: 第1章 → 第4章 → 第2・3章 → 第5章 → 第8章 の順でも、言語の芯には届きます。**第4章（遅延評価）はこの言語の看板**なので、飛ばさないでください。`:thunks` と `:trace` で、評価が起きる瞬間を自分の目で見ることになります。

演習問題には解答が付いています。各問題の直下に折りたたんで置いてあるので、考えてから開いてください。

予備知識

何かひとつのプログラミング言語で、変数と関数と条件分岐を書いたことがあれば十分です。関数型プログラミングの経験は要りません。遅延評価も知らなくて大丈夫です —— むしろ 知らないほうが、第4章は面白いはずです。

それでは、よい部活動を。

まず REPL を開いて、`1 + 2` と打ってみてください。第1章はそこから始まります。

第1章 やさしい入門

新しいプログラミング言語を身につける方法は、昔からひとつしかありません。それでプログラムを書くことです。この章では、Lune のプログラムを書いて、動かして、壊してみます。細かい規則の説明はあとの章に任せて、まず言語の全体を1周しましょう。

ひとつだけ、この本の流儀を先に伝えておきます。**エラーを恐れないでください**。Lune のコンパイラは、エラーを「失敗の通知」ではなく「教材」として設計されています。この章の後半では、わざとエラーを起こして、コンパイラに教わる練習をします。

1.1 REPL で計算する

Lune には REPL（対話環境）があります。式を打ち込むと、その場で評価して結果を返してくれます。まずはここから始めましょう。

```
$ lune
Lune v0.1.0 REPL. Type :help or :quit.
lune>
```

表記について — この本では、シェルのコマンドは `$` で、REPL への入力は `lune>` で示します。`lune` コマンドは、リポジトリの `./bin/lune` を指します（パスを通すか、`alias lune=./bin/lune` としておくとも本書のとおりに入ります）。

もうひとつ。本書は、コンパイラの診断（エラーメッセージ）をすべて**日本語**で掲載します。同じ出力を得るには、環境変数 `LUNE_LANG` を `ja` に設定してください。

```
$ export LUNE_LANG=ja
```

シェルの設定ファイル（`~/.zshrc` など）に書いておけば、以後 `lune` はいつでも日本語で答えます。英語の診断に切り替えたいときは、環境変数より優先される `--lang en` フラグを付けてください（REPL の中では `:lang en`）。

数を打ち込んでみます。

```
lune> 1 + 2
3 : Int
lune> 40 + 2
42 : Int
```

結果は `値 : 型` の形で表示されます。`3` が値、`Int` が型です。Lune ではすべての式が型を持ち、REPL はそれをいつも教えてくれます。

文字列は `"` で囲みます。`+` でつなげられます。

```
lune> "hello, " + "world"
"hello, world" : String
```

REPL を抜けるには `:quit`（または `:q`）です。

```
lune> :quit
bye
```

`:help` でコマンドの一覧が見られます。REPL は本書全体を通して最良の実験室です。疑問がわいたら、いつでも打ち込んで確かめてください。

1.2 let で名前を付ける

値に名前を付けるには `let` を使います。

```
lune> let x = 41
ok
lune> x + 1
42 : Int
```

宣言に対して REPL は `ok` とだけ答えます。値が表示されないことに注意してください。これは手抜きではありません。**Lune の `let` は、名前を付けた時点ではまだ計算をしていないのです。**

予告: 計算はまだ起きていない — REPL の `:thunks` コマンドを使うと、束縛が評価済みかどうかを（評価を起こさずに）覗けます。

```
lune> let x = 1 + 1
ok
lune> :thunks x
x : unevaluated
lune> x
2 : Int
lune> :thunks x
x : evaluated = 2
```

`1 + 1` は、`x` を使った瞬間に初めて計算されました。これが **遅延評価** — Lune の設計の心臓部です。第4章でじっくり扱います。いまは「`let` は計算の約束を書くもの」とだけ覚えて先へ進んでください。

1.3 プログラムをファイルに書く

REPL は実験室ですが、プログラムはファイルに書いて残します。エディタで `hello.lune` を作ってください。

```
module hello

def greet(name: String): String =
  "hello, " + name

let main = println(greet("world"))
```

上から順に読みます。

- `module hello` — このファイルがモジュール `hello` であることを宣言します。Lune のソースファイルは必ずこの宣言から始まります。
- `def greet(name: String): String = ...` — 関数定義です。引数 `name` の型が `String`、戻り値の型も `String`。本体はインデントして書きます。
- `let main = println(greet("world"))` — トップレベルの束縛です。 `println` は値を表示する組み込み関数です。

まず、型が合っているかだけを検査してみます。

```
$ lune --check hello.lune
type check OK
```

次に実行します。Lune v0.1 の実行モデルは「**トップレベルの束縛をひとつ選んで評価する**」です。 `--eval` に束縛の名前を渡します。

```
$ lune --eval main hello.lune
hello, world
()
```

2行表示されました。1行目は `println` が出力した文字列です。2行目の `()` は `main` 自体の値 — `println` の戻り値である `Unit` 値です（`--eval` は評価した束縛の値を最後に表示します）。

`println` は `String` を**生の内容のまま**出力します。引用符は付かず、`"a\nb"` のようなエスケープは実際の改行として出力されます。一方、`String` 以外の値は Lune の標準表示形式（`show` 形式）で表示されます。文字列を引用符付きの `show` 形式で表示したいときは `println(show(value))` と書きます。表示の規則は第2章で整理します。

1.4 if は式である

Lune の `if` は、値を返す**式**です。C や Python の「文」の感覚とは少し違います。

```
lune> if 5 > 3 then "big" else "small"
"big" : String
```

式なので、結果をそのまま `let` に束縛したり、関数の本体にしたりできます。値が要る場面では `else` を省略できません — 条件が偽のときの値がなくなってしまうからです。

本体が長くなるときは、ブロック形も使えます。REPL は行末が `=` や `:` で終わると継続入力になります（`...` が継続のプロンプトです。空行で確定します）。

```
lune> def abs(x: Int): Int =
...     if x < 0:
...         -x
...     else:
...         x
...
ok
lune> abs(-5)
5 : Int
```

1.5 リストで遊ぶ — 温度換算表

C の入門書は、伝統的に華氏-摂氏の温度換算表を最初に作ります。敬意を表して、この本でも作りましょう。ただし Lune 流に — ループの代わりに**リスト**で。

リストは `[ ]` で書きます。表示は Lisp 風の `( )` 区切りです。

```
lune> [1, 2, 3]
(1 2 3) : List[Int]
lune> range(0, 5)
(0 1 2 3 4) : List[Int]
```

`range(start, end)` は `start` 以上 `end` 未満の整数リストを作ります。リストを加工する基本の道具は 3 つ、`map`（全要素に関数を適用）、`filter`（条件を満たす要素だけ残す）、`fold`（畳み込んで 1 つの値にする）です。

```
lune> map(range(1, 6), fn x -> x * 2)
(2 4 6 8 10) : List[Int]
lune> filter(range(1, 10), fn x -> x % 2 == 0)
(2 4 6 8) : List[Int]
lune> fold([1, 2, 3, 4], 0, fn acc x -> acc + x)
10 : Int
```

`fn x -> x * 2` は名前のない小さな関数（ラムダ）です。引数の型を書いていないのに動くのは、`map` が期待する型からコンパイラが推論してくれるからです（第3章）。

ところで、`range` の幅をうんと広げてみてください。

```
lune> take(filter(range(1, 1000000000), fn x -> x % 7 == 0), 5)
(7 14 21 28 35) : List[Int]
```

1億要素のリストから 7 の倍数を 5 個。**待たされません**。Lune は 1 億個のリストを作ってから 5 個取り出したのではなく、5 個取るのに要る分だけリストを伸ばしたからです。「必要になるまで計算しない」 — この性質を Lune は**遅延評価**と呼び、言語の中心に据えています。第4章まるごとがこの話です。

道具が揃いました。温度換算表を作ります。ファイル `temperature.lune` :

```

module temperature

# 華氏 0~300 度を 20 度刻みで摂氏に換算した対応表を作る。
def toCelsius(f: Int): Double =
    (f - 32) * 5 / 9

let fahrenheit = map(range(0, 16), fn i -> i * 20)

let table = map(fahrenheit, fn f -> (f, toCelsius(f)))

```

から行末まではコメントです。toCelsius の戻り値が Double になっているのは、Lune の / が常に Double を返すからです。C の整数除算のような切り捨ての罫はありません（そのかわり Int と Double を混ぜて足すことはできません — 第2章で詳しく）。

fn f -> (f, toCelsius(f)) の (f, toCelsius(f)) は**タプル**、2つの値の組です。実行してみます。

```

$ lune --eval table temperature.lune
((0, -17.77777777777778) (20, -6.666666666666667) (40, 4.444444444444445)
(60, 15.555555555555555) (80, 26.666666666666668) (100, 37.77777777777778)
(120, 48.888888888888886) (140, 60.0) (160, 71.11111111111111) (180,
82.22222222222223) (200, 93.33333333333333) (220, 104.44444444444444)
(240, 115.55555555555556) (260, 126.66666666666667) (280,
137.77777777777777) (300, 148.88888888888889))

```

華氏と摂氏の対応表が、(華氏, 摂氏) のタプルのリストとして得られました。ループも、カウンタ変数も、途中経過を入れる変数もありません。「0 から 15 までの番号を 20 倍して華氏のリストを作り、それぞれを摂氏との組に写す」 — プログラムが仕様書のように読めます。これが Lune の基本的な書き味です。

1.6 最初のエラー — コンパイラと対話する

さて、お待ちかねの時間です。**わざと間違えましょう**。hello.lune を少し書き換えて、typo.lune を作ります。greeting と書くべきところを greting と打ち間違えたことにします。

```

module hello

let greeting = "hello, world"

let main = println(greting)

```

```

$ lune --check typo.lune

```

```
error[TYP0001]: 未定義の名前: greting
--> typo.lune:5:20
|
5 | let main = println(greting)
|                      ^^^^^^^^ この名前は定義されていない
= hint: もしかして `greeting` ですか?
= help: 詳しくは `lune explain TYP0001 --lang ja` を実行してください
```

この表示を上から解剖します。Lune の診断はすべてこの形です。

行	意味
<code>error[TYP0001]: ...</code>	重大度 (error)、診断コード (TYP0001)、要約
<code>--> typo.lune:5:20</code>	場所 — ファイル名:行:桁
<code>5 let main = ...</code> と <code>^^^^^^</code>	問題のソースと、問題の箇所を指す印
<code>= hint: ...</code>	修正の提案。ここでは「もしかして <code>greeting</code> ですか?」
<code>= help: 詳しくは \ lune explain TYP0001 ...`</code>	もっと詳しく知る方法

表記について — 診断の `-->` 行に表示されるパスは、実際にはあなたの環境での絶対パスになります。紙面では作業ディレクトリを省略しています。

`help` 行の言うとおりにしてみましょう。**Lune のすべての診断コードには、教材としての解説が付いています。**

```
$ lune explain TYP0001
```

```
error[TYP0001]: 未定義の名前
```

現在のスコープで束縛されておらず、`prelude` にも `import` にも見つからない名前が使われました。

発生する例:

```
let y = x + 1      # x はどこにも定義されていない
```

直し方:

使う前に名前を定義または `import` してください。綴りも確認してください。

何が起きたのか、それを再現する最小の例、直し方。エラーが出るたびにこの解説を読む習慣をつけると、コンパイラが家庭教師になります（help 行は `--lang ja` を付けた再実行を案内しますが、`LUNE_LANG=ja` を設定済みならフラグなしでこのとおり日本語になります）。

そして今回のようなタイプは、hint が機械的に適用できる修正なので、`lune fix` が直してくれます。

```
$ lune fix typo.lune
module hello

let greeting = "hello, world"

let main = println(greeting)
```

修正後のソースが出力されました（`--write` を付けるとファイルを直接書き換えます）。

この「壊す → 診断を読む → `explain` で理解する → 直す」というループが、本書の学び方の基本です。各章にはこのための「壊してみよう」コーナーがあります。さっそくひとつ。

壊してみよう — REPL で `Int` と `Bool` を足してみてください。

```
lune> 1 + true
error[TYP0003]: +: Int が必要ですが、Bool が見つかりました
= help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

`TYP0003`（型の不一致）は、これから最もよく出会う診断です。 `:explain` `TYP0003` で解説を読んでおきましょう（REPL の中では `lune explain` の代わりに `:explain` が使えます）。

1.7 ここから先の地図

これで1周です。あなたはもう、Lune のプログラムを書いて、型を検査して、実行して、エラーを読んで直せます。この本の残りは、この章で駆け抜けた風景をゆっくり歩き直します。

- **第2～3章** — 値・型・式、そして関数。部分適用とパイプライン `|>` という強力な道具が加わります。

- **第4章** — 遅延評価。1.2 節で予告した「計算はまだ起きていない」の全貌。Lune を学ぶ最大の理由がここにあります。
- **第5～8章** — データの形を型で表す道具たち。代数的データ型、パターンマッチ、レコード、null 安全、そして無限リスト。
- **第9～10章** — 命令的な書き方と、プログラムの分割。
- **第11～13章** — コンパイラと道具を使いこなし、少し大きなプログラムを組み立てます。

順に読むのがおすすめですが、第4章まで読めば、あとは気になる章からつまみ食いしても大丈夫です。

まとめ

書いたもの	意味
<code>let name = 式</code>	束縛（計算の約束）。使われた時に評価される
<code>def f(x: T): U = 本体</code>	関数定義
<code>fn x -> 式</code>	ラムダ（名前のない関数）
<code>if c then a else b / if c: ブロック</code>	条件分岐。値を返す式
<code>[1, 2, 3]</code>	リスト。表示は <code>(1 2 3)</code>
<code>(a, b)</code>	タプル
<code>module m</code>	モジュール宣言。ファイルの先頭に置く
コマンド	意味
<code>lune</code>	REPL を起動
<code>lune --check FILE</code>	型検査のみ
<code>lune --eval NAME FILE</code>	束縛 NAME を評価して表示
<code>lune explain CODE</code>	診断コードの解説
<code>lune fix FILE</code>	機械適用できる修正を実施
<code>export LUNE_LANG=ja</code>	診断の既定言語を日本語にする（ <code>--lang</code> フラグが優先）
<code>:quit</code> <code>:help</code> <code>:thunks</code> <code>:explain</code>	REPL コマンド

演習問題

演習 1-1 (★) `hello.lune` の挨拶を、自分の名前と好きな挨拶に変えて動かしてください。ついでに `greet` の本体の `+` を `-` に変えて、どんな診断が出るか見てみましょう。

解答

`"hello, " - name` は型エラーになります。

```
error[TYP0003]: -: 数値型が必要ですが、String が見つかりました
  = help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

`+` は文字列の連結に使えますが、`-` は数値専用です。診断が「`-` は数値型を期待したのに `String` が来た」と、演算子の側の都合を主語にして説明していることに注目してください。

ところで、呼び出しの引数を消して `greet()` にしても、実は型エラーになりません（`lune --check` は通ります）。Lune の関数は引数を途中まで渡せるからです — これは部分適用といって、第3章の主役のひとつです。

演習 1-2 (★) `temperature.lune` を逆向きにした、摂氏 → 華氏の換算表を作ってください（0～100 度を 20 度刻み）。換算式は $F = C \times 9/5 + 32$ です。

解答

```
module answers

# 演習 1-2: 摂氏 → 華氏の換算表 (0~100 度を 20 度刻み)。
def toFahrenheit(c: Int): Double =
  c * 9 / 5 + 32.0

let table = map(range(0, 6), fn i -> (i * 20, toFahrenheit(i * 20)))
```

```
$ lune --eval table ex1-2.lune
((0, 32.0) (20, 68.0) (40, 104.0) (60, 140.0) (80, 176.0) (100, 212.0))
```

ひとつ罣があります。`+ 32` と書くと `TYP0003`（`+: Double` が必要ですが、`Int` が見つかりました）になります。`c * 9 / 5` は `/` のせいで `Double` になっているので、足す方も `32.0` と書く必要があります。Lune は `Int` と `Double` を黙って混ぜません。

演習 1-3 (★★) `temperature.lune` の `fahrenheits` について、`fold` を使って合計を、さらに `length` と組み合わせて平均を求めてください。

解答

```
module answers

# 演習 1-3: fold で華氏温度の合計と平均を求める。
let fahrenheit = map(range(0, 16), fn i -> i * 20)

let total = fold(fahrenheit, 0, fn acc f -> acc + f)

let average = total / length(fahrenheit)
```

```
$ lune --eval total ex1-3.lune
2400
$ lune --eval average ex1-3.lune
150.0
```

`fold`(リスト, 初期値, `fn 途中結果 要素 -> 新しい途中結果`) の形を覚えてください。
`average` が `150.0` (`Double`) になるのは、`/` が常に `Double` を返すからです。

演習 1-4 (★★) `temperature.lune` の表から、摂氏で氷点下になる華氏温度だけを `filter` で取り出してください。

解答

```
module answers

# 演習 1-4: 摂氏で氷点下になる華氏温度だけを残す。
def toCelsius(f: Int): Double =
  (f - 32) * 5 / 9

let fahrenheit = map(range(0, 16), fn i -> i * 20)

let freezing = filter(fahrenheit, fn f -> toCelsius(f) < 0.0)
```

```
$ lune --eval freezing ex1-4.lune
(0 20)
```

華氏 0 度と 20 度だけが氷点下でした。比較の右辺が `0` ではなく `0.0` なのは、演習 1-2 と同じ理由です。

演習 1-5 (★) `lune explain` を「呼び出す前」に自分で診断を予想する練習です。
`greet("world", "again")` のように引数を 2 つ渡すと何が起きるでしょうか。予想してから実行し、`lune explain TYP0005` を読んでください。

解答

```
error[TYP0005]: 引数は最大 1 個ですが、2 個渡されました
--> arity.lune:6:20
|
6 | let main = println(greet("world", "again"))
|                      ^^^^^^ 引数の個数が違う
= help: 詳しくは `lune explain TYP0005 --lang ja` を実行してください
```

「引数は**最大** 1 個」という言い方に注目してください。Lune の関数は引数を途中まで渡すこと（部分適用、第3章）ができるので、「少ない」のは必ずしもエラーではなく、「多い」ことが確実なエラーなのです。

より正確には — この章で登場した機能の正確な仕様は次にあります: 構文とリテラルは `documents/LANGUAGE_SPEC.md §5-9`、REPL の表示は `documents/REPL_SPEC.md` と `documents/VALUE_DISPLAY_SPEC.md`、診断の形式は `documents/ERROR_DIAGNOSTICS_SPEC.md`。この章のコード例は `books/examples/ch01/` にあり、すべて実際の CLI で検証されています。

第2章 値・型・式

第1章では駆け足で言語を1周しました。ここからの第II部では、部品を一つずつ手に取って眺めます。まずは一番小さな部品 ― 値と、その型と、それらを組み合わせる式です。

この章でも REPL を開いておいてください。Lune の式はすべて REPL にそのまま打ち込めます。「この式の型は何だろう」と思ったら、打てば答えが返ります。

2.1 リテラルと基本型

Lune のリテラルと、対応する型の一覧です。

リテラル	型	例の表示
42	Int	42 : Int
3.14	Double	3.14 : Double
"hello"	String	"hello" : String
true / false	Bool	true : Bool
null	Null	null : Null
()	Unit	() : Unit
(1, "a")	Tuple[Int, String]	(1, "a") : Tuple[Int, String]

いくつか補足します。

Int は大きさを気にしなくてよい。 v0.1 の Int は任意精度で、あふれることはありません。

```
lune> 9223372036854775807 + 1
9223372036854775808 : Int
```

64ビット整数の限界を超えても平気です（C ならここで音もなく負の数になります）。

文字列のエスケープ。 " の中では \n（改行）や \" が使えます。表示（show 形式）ではエスケープされたまま出るので驚かないでください。

```
lune> "a\nb"
"a\nb" : String
```

これはあくまで REPL の `show` 形式の話です。 `println` に渡すと生の内容が出力されるので、 `\n` は実際の改行になります（1.3節で見たとおりです）。

シングルクォートは `String` になる。 `'x'` という書き方も受け付けますが、v0.1 に独立した文字型は事実上なく、 `"x"` と同じ `String` になります。

```
lune> 'x'
"x" : String
```

`null` と `()` は別物です。 `null` は「値がない」ことを表すためだけの値で、`null` 安全（第7章）の主演です。 `()` は `Unit` 型のただ一つの値で、「返すべき意味のある値がない」関数（`println` など）の戻り値に使われます。

2.2 演算子 — 算術・比較・論理

算術演算子は `+` `-` `*` `/` `//` `%` と単項の `-` です。 `*` `/` `//` `%` は同じ優先順位で、それより `+` `-` が弱い — おおむね数学の直感どおりで、 `()` でくくれば変えられます。

```
lune> 1 + 2 * 3
7 : Int
lune> (1 + 2) * 3
9 : Int
lune> 2 * -3
-6 : Int
```

割り算には注意が要ります。第1章でも触れたとおり、 `/` は常に `Double` を返します。 `Int` を `Int` で割っても `Double` です。整数の割り算の余りが欲しいときは `%`（こちらは `Int`）です。

```
lune> 7 / 2
3.5 : Double
lune> 7 % 2
1 : Int
```

では「整数を整数で割って整数が欲しい」ときは何を使うのか。 **床除算** `//` です。結果の型はオペランドの型のまま変わりません。

```
lune> 7 // 2
3 : Int
lune> 7.0 // 2.0
3.0 : Double
```

`/` と `//` は名前が似ていますが、**別の演算子**です。「割り算」と言うとき Lune には2つある、と覚えてください。`Int` のまま計算を続けたいなら `//`、実数として割りたいなら `/` です。

`//` の丸めには一つ、予想を裏切るところがあります。**負の無限大方向への切り下げ** (floor) で、ゼロ方向への切り捨てではありません。

```
lune> -7 // 2
-4 : Int
```

`-3.5` を「切り捨て」て `-3` だと思った人は、ここで一度つまずきます。なぜ `-4` なのかは、`%` と組で辻褄が合っているからです — その理由は第13章の演習 13-4 でゆっくり考えます。いまは「`//` は floor」と覚えておけば十分です。

なお `/` `//` `%` はどれも、右側が 0 なら実行時エラーになります (`RUN0006`。第11章で読み方を学びます)。

そして `Int` と `Double` は黙って混ざりません。

```
lune> 1 + 2.0
error[TYP0003]: +=: Int が必要ですが、Double が見つかりました
  = help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
lune> 1 == 1.0
error[TYP0003]: ==: Int と Double は比較できません
  = help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

暗黙の型変換で「だいたい合ってる」結果を返すより、書き手にどちらの世界で計算したいのかを決めてもらう — Lune はその立場です。`2.0` か `2` か、リテラルの側を揃えてください。

比較演算子は `==` `!=` `<` `<=` `>` `>=` です。`==` と `!=` は数値・文字列・Bool で使えます。大小比較の `<` 系は数値専用です。

```
lune> 3 < 5
true : Bool
lune> "a" == "a"
true : Bool
lune> "a" < "b"
error[TYP0003]: <: 数値型が必要ですが、String が見つかりました
  = help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

複合値 — タプル、リスト、そして第5章で学ぶ ADT の値 — も `==` で比べられます。比較は**構造的**です。別々に作った値でも、中身が等しければ等しい値です。

```
lune> (1, 2) == (1, 2)
true : Bool
lune> (1, 2) == (2, 1)
false : Bool
lune> [1, 2] == [1, 2]
true : Bool
lune> Some(1) == Some(1)
true : Bool
```

ただし、比べられるのは**同じ型**としだけです。要素数の違うタプルは中身以前に型が違
うので、比較そのものが型エラーになります。

```
lune> (1, 2) == (1, 2, 3)
error[TYP0003]: ==: Tuple[Int, Int] と Tuple[Int, Int, Int] は比較できません
= help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

論理演算子は `&&`（かつ）と `||`（または）、否定は関数 `not` です。`&&` は `||` より強く
結びつきます。そして第4章で見た遅延評価のおかげで、どちらも**短絡**します — 左だけで
答えが決まるなら、右は評価されません。

```
lune> false && crash()
false : Bool
lune> true || crash()
true : Bool
lune> not(true)
false : Bool
```

このほかに null 合体の `??`（第7章）とパイプライン `|>`（第3章）があります。全演算子
の優先順位表は付録Aにまとめます。

2.3 if は式 — then/else と elif

第1章の復習から。 `if` は値を返す式で、1行形式とブロック形式があります。

```
lune> if 5 > 3 then "big" else "small"
"big" : String
```

式である以上、守るべき規則が2つあります。**条件は `Bool`**、そして**分岐の型は一致で
す**。

```
lune> if 1 then 2 else 3
error[TYP0003]: if の条件: Bool が必要ですが、Int が見つかりました
  = help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
lune> if true then 1 else "a"
error[TYP0003]: 分岐の型が一致しません: Int と String
  = help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

Cのように「0は偽」ではありません。`if x % 2`ではなく`if x % 2 == 1`と書きます。

分岐が増えるときは、ブロック形式の `elif` が使えます。 `grade.lune` :

```
module grade

# if は式。block 形式では elif で分岐を足せる。
def grade(score: Int): String =
  if score >= 80:
    "pass"
  elif score >= 60:
    "retry"
  else:
    "fail"

let result = grade(75)
```

```
$ lune --eval result grade.lune
"retry"
```

どんなに分岐しても、`grade` は必ず `String` を一つ返します。「どの経路を通っても値がある」ことを型検査が保証してくれる — 式指向の安心感です。

2.4 let-in — 式の中の束縛

`let` はトップレベルだけのものではありません。**式の中で**一時的な名前が欲しいときは `let ... in ...` が使えます。

```
lune> let answer = let x = 40 in x + 2
ok
lune> answer
42 : Int
```

`in` の右側が式全体の値になり、`x` はその中でだけ見えます。名前を2つ以上付けたいときは、ブロック形式で `let` を並べるのがきれいです。

```
module answers
```

```
# 演習 2-3: ブロック形式の let で中間値に名前を付けて BMI を計算する。
```

```
let bmi =  
  let h = 1.7  
  let w = 62.0  
  w / (h * h)
```

```
$ lune --eval bmi ex2-3.lune  
21.453287197231838
```

インデントされたブロックの中に `let` を並べ、最後に置いた式がブロック全体の値になります。 `lune fmt` もこの形を正準としています。

2.5 型注釈 — 書く場所と省ける場所

`let` には型注釈を付けられます。

```
let y: Int = 42
```

注釈と実際の型が食い違えば、もちろん型エラーです。 `annot.lune` :

```
module bad
```

```
let n: Int = "hello"
```

```
$ lune --check annot.lune
```

```
error[TYP0003]: let の型注釈: Int が必要ですが、String が見つかりました  
--> annot.lune:3:14  
  |  
3 | let n: Int = "hello"  
  |               ^^^^^^^^ この式の型は String  
    = help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

では、どこで注釈を書き、どこで省けるのか。v0.1 の目安はこうです。

場所	注釈
<code>let</code> の右辺が明らか	省略できる（右辺から推論される）
<code>def</code> の引数と戻り値	書く（再帰関数の戻り値型は必須 — TYP0011）
ラムダ <code>fn x -> ...</code> の引数	文脈から推論されることが多い（第3章）

注釈は機械のためだけのものではありません。 `let total: Double = ...` と書いておけば、読む人への宣言になり、右辺を書き間違えた瞬間に上のような診断が出ます。迷ったら書く、確かめたいときは `:type` — これが実務の感覚です。

2.6 タプル — 型の違う値の組

リスト（第8章）の要素はすべて同じ型ですが、**タプル**は型の違う値を決まった個数だけ束ねます。

```
lune> let pair = (1, "a")
ok
lune> pair
(1, "a") : Tuple[Int, String]
```

タプルから値を取り出すのは**分解束縛**です。 `.1` のようなアクセサはありません — 名前を付けて開けます。

```
lune> let (a, b) = (10, 20)
ok
lune> a
10 : Int
lune> b
20 : Int
```

第1章の温度換算表では（華氏、摂氏）のタプルのリストを作りました。「2つの値をとりあえず束ねて運ぶ」のがタプルの仕事です。束ねた値に名前を付けたいのなら、それはレコード（第6章）の出番です。

壊してみよう — 型検査より前の段階、字句とレイアウトのエラーも見ておきましょう。どちらも第11章で学ぶ診断コード体系の「LXL」「PRS」族です。

`let x = $1` と書くと、Lune の知らない文字でエラーになります。

```
error[LXL0001]: 予期しない文字 '$'  
--> lex.lune:3:9  
|  
3 | let x = $1  
|           ^ 予期しない文字  
= help: 詳しくは `lune explain LXL0001 --lang ja` を実行してください
```

理由なく行頭を字下げすると、レイアウト規則に引っかかります。

```
error[PRS0001]: トップレベル宣言が必要ですが、INDENT が見つかりました  
--> indent.lune:4:1  
|  
4 |   let b = 2  
|   ^ 予期しないトークン  
= help: 詳しくは `lune explain PRS0001 --lang ja` を実行してください
```

Lune のインデントは Python と同じく構文の一部です。ブロックを作るのは `=` や `:` の後だけ、と覚えてください。

まとめ

事実	一言で
基本型	<code>Int</code> （任意精度） / <code>Double</code> / <code>String</code> / <code>Bool</code> / <code>Null</code> / <code>Unit</code> / <code>Tuple[...]</code>
割り算	<code>/</code> は常に <code>Double</code> 、 <code>//</code> は床除算で型はそのまま、余りは <code>%</code>
<code>//</code> の丸め	負の無限大方向（floor）。 <code>-7 // 2</code> は <code>-4</code>
<code>Int</code> と <code>Double</code>	暗黙には混ざらない。リテラルの側を揃える
<code>==</code>	中身（構造）で比較。比べられるのは同じ型どうしだけ
<code>&&</code> 、 <code>`</code>	
<code>if</code>	条件は <code>Bool</code> 、分岐の型は一致。 <code>elif</code> で多分岐
<code>let ... in</code> / ブロック <code>let</code>	式の中の一時的な名前
型注釈	<code>let</code> では省略可、 <code>def</code> では書く。確認は <code>:type</code>
タプル	<code>(a, b)</code> で束ね、 <code>let (x, y) = ...</code> で開ける

演習問題

演習 2-1（★） 次の式の値と型を予想してから、REPL で確かめてください。

```
19 / 4
19 // 4
19 % 4
1 + 2 * 3 - 4
2 * -3
true && false || true
```

解答

`4.75 : Double`（`/` は常に `Double`）、`4 : Int`（`//` は床除算なので型は `Int` のまま）、`3 : Int`、`3 : Int`、`-6 : Int`、`true : Bool`（`&&` が先に評価されて `false`、次に `|| true`）。

演習 2-2（★） `grade.lune` に「90 点以上は“excellent”」の分岐を足してください。分岐をどこに足すかで結果が変わることを、`grade(95)` で確かめてください。

解答

`if score >= 90: "excellent"` を先頭に置きます。 `elif score >= 80: ...` より後ろに置くと、95 点は先に `score >= 80` に捕まって `"pass"` になってしまいます。
`if/elif` は上から順に試される — 条件の並び順は仕様の一部です。

演習 2-3 (★★) 身長 1.7 m、体重 62.0 kg の BMI (体重 ÷ 身長²) を、ブロック形式の `let` で中間値に名前を付けながら計算してください。

解答

```
module answers
```

```
# 演習 2-3: ブロック形式の let で中間値に名前を付けて BMI を計算する。
let bmi =
  let h = 1.7
  let w = 62.0
  w / (h * h)
```

```
$ lune --eval bmi ex2-3.lune
21.453287197231838
```

1.7 を 1 にすると `TYP0003` が出ます。 `Int` と `Double` の分離は、こういう単位の混同を早めに捕まえてくれます。

演習 2-4 (★★) 分解束縛を使って、タプル `(1, 2)` の2つの値を入れ替えた `(2, 1)` を作ってください。

解答

```
module answers
```

```
# 演習 2-4: タプルの分解束縛で 2 値を入れ替える。
let (x, y) = (1, 2)

let swapped = (y, x)
```

```
$ lune --eval swapped ex2-4.lune
(2, 1)
```

一時変数は要りません。「開けて、逆順に束ね直す」だけです。

演習 2-5 (★★) 次の3つの式の結果を予想してから確かめてください。3つ目だけ毛色が違います — 何が起きるでしょうか。

```
(1, 2) == (2, 1)
[1, 2] == [1, 2]
(1, 2) == (1, 2, 3)
```

解答

`false : Bool` (中身の並びが違う)、`true : Bool` (別々に作ったリストでも中身が同じなら等しい — 構造的等価)、そして3つ目は `true / false` ではなく**型エラー**です。

```
error[TYP0003]: ==: Tuple[Int, Int] と Tuple[Int, Int, Int] は比較できません
= help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

要素数が違うタプルは違う型なので、「等しいか」を聞くこと自体ができません。 `==` の答えが返るのは同じ型どうしだけ — 「比較できないものは実行前に弾く」という、`1 == 1.0` と同じ設計です。

より正確には — リテラルと型は `documents/LANGUAGE_SPEC.md` §5-6、演算子の全優先順位表は `documents/SYNTAX_SPEC.md` §14 (`::` `++` など将来分を含むため、実装済みかは `LANGUAGE_SPEC.md` を正とします)、`if / let-in` は `documents/LANGUAGE_SPEC.md` §9。値の表示規則は `documents/VALUE_DISPLAY_SPEC.md`。この章のコード例は `books/examples/ch02/` にあり、すべて実際の CLI で検証されています。

第3章 関数

第1章から `def` を使ってきましたが、この章で関数を正面から扱います。目標は一つ、**関数は値である**という感覚を手に入れることです。関数が値なら、変数に入れられ、引数として渡せ、戻り値として返せます。ここから Lune のプログラムは急に伸び伸びし始めます。

3.1 def — 定義の解剖

まず復習を兼ねて、部品の名前を確認します。

```
def add(x: Int, y: Int): Int =  
  x + y
```

`add` が関数名、`x: Int, y: Int` が引数とその型、2つ目の `Int` が戻り値の型、`=` の後のインデントされた部分が本体です。本体は式で、その値がそのまま戻り値になります（`return` はありません — 第2章で見たとおり、`if` も含めてすべてが式だからです）。

引数と戻り値の型は書いてください。v0.1 でも省略できる場面はありますが、書かないと推論が `Any` に落ちて警告が出たり（`TYP0010`、3.3節）、再帰関数ではエラーになったり（`TYP0011`、3.7節）します。何より、型は関数の**看板**です。

3.2 関数は値である

定義した関数は、整数や文字列と同じように扱えます。たとえば別の名前に束縛できます。

```
lune> def inc(n: Int): Int =  
...     n + 1  
...  
ok  
lune> let renamed = inc  
ok  
lune> renamed(41)  
42 : Int
```

値である以上、型があります。関数の型は `->` で書きます。

```
let inc: Int -> Int = fn x -> x + 1
let addA: Int -> Int -> Int = fn x y -> x + y
let addB: (Int, Int) -> Int = fn x y -> x + y
```

`Int -> Int` は「`Int` を受け取り `Int` を返す」。2引数は `Int -> Int -> Int` とも `(Int, Int) -> Int` とも書けます。 `->` は右結合なので、前者は `Int -> (Int -> Int)` — 「`Int` を受け取ると、`Int -> Int` を返す」と読めます。この読み方が 3.5 節の部分適用にそのままつながります。

第1章から使ってきた `map` の型を REPL に聞いてみましょう。

```
lune> :type map
map : [T, U] List[T] -> (T -> U) -> List[U]
```

先頭の `[T, U]` は「任意の型 `T`, `U` について」という宣言です（ジェネリクス、第5章）。読み下すと「`T` のリストと、`T` を `U` にする関数を受け取り、`U` のリストを返す」。型シグネチャは関数の説明文としてかなり優秀です。

3.3 ラムダ — 名前のない関数

`fn 引数 -> 式` で、その場で関数を作れます。

```
fn x -> x + 1      # 1引数
fn x y -> x + y    # 2引数（並べるだけ）
fn -> 42           # 0引数
fn x: Int -> x * 2  # 型注釈付き
```

引数の型注釈は、**文脈から分かるときは省略できます**。`map` や `applyTwice`（次節）の引数として渡すときは、受け取る側の型が期待を伝えてくれるからです。第1章の `map(range(1, 6), fn x -> x * 2)` で警告が出なかったのはこのためです。

文脈がない場所では、推論のしようがありません。

```
lune> let f = fn x -> x * 2
warning[TYP0010]: 引数 x の型を推論できません
  --> <repl:19>:1:12
    |
1 | let f = fn x -> x * 2
    |               ^ 引数の型が Any にフォールバックする
    = hint: 型注釈を追加してください。例: `fn x: Int -> ...`
    = help: 詳しくは `lune explain TYP0010 --lang ja` を実行してください
ok
```

`error` ではなく `warning` です — 束縛は成立し、`x` の型は `Any` に落ちます。動きはしますが型検査の保護が薄くなるので、`hint` のとおり `fn x: Int -> x * 2` と書くのが習慣です。

3.4 高階関数

関数を受け取る関数、返す関数を**高階関数**と呼びます。どちらも普通に書けます。

`hof.lune`:

```
module hof

def inc(n: Int): Int =
  n + 1

# 関数を受け取る関数。
def applyTwice(f: Int -> Int, x: Int): Int =
  f(f(x))

# 関数を返す関数。
def adder(n: Int): Int -> Int =
  fn x -> x + n

let fortyTwo = applyTwice(inc, 40)

let seven = adder(3)(4)
```

```
$ lune --eval fortyTwo hof.lune
42
$ lune --eval seven hof.lune
7
```

REPL で遊ぶようになります。

```
lune> applyTwice(fn x -> x * 10, 4)
400 : Int
lune> adder(3)(4)
7 : Int
```

`adder(3)` が返すラムダ `fn x -> x + n` は、作られたときの `n = 3` を覚えています。関数が定義時の環境を閉じ込めて持ち歩く — これを**クロージャ**といいます。第4章のサンクが「式と環境」の包みだったのと同じ仕掛けで、Lune の世界は「式+環境」でできています。

3.5 部分適用 — 引数は途中まで渡してよい

第1章の演習で、1引数の関数を `greet()` と0引数で呼んでも型エラーにならない、という不思議を見ました。種明かしがこの節です。

Lune では、**引数が足りない呼び出しはエラーではなく、「残りの引数を待つ関数」を返します**。

```
lune> def add(x: Int, y: Int): Int =  
...     x + y  
...  
ok  
lune> let plusTen = add(10)  
ok  
lune> :type plusTen  
plusTen : Int -> Int  
lune> plusTen(32)  
42 : Int
```

`add : Int -> Int -> Int` を「`Int` を渡すと `Int -> Int` が返る」と読んだのを思い出してください。`add(10)` はその読み方どおりのことをしただけです。まとめて `add(10, 32)` と渡しても、分けて `add(10)(32)` と渡しても同じ — この柔軟さのおかげで、「既存の関数の引数を一部固定した道具」が一行で作れます。

```
lune> map([1, 2, 3], plusTen)  
(11 12 13) : List[Int]
```

なお、渡しすぎは今までどおりエラーです（`TYP0005`、「引数は**最大** 1 個」という言い回しの理由がこれで分かりますね）。また、部分適用で渡した引数も通常の呼び出しと同じく sank として捕まります — 遅延評価は部分適用の中でも一貫しています。

3.6 パイプライン `|>` — データの流れを書く

関数適用が入れ子になると、読み順が実行順と逆になります。`double(inc(inc(5)))` は「5 を `inc` して、`inc` して、`double` する」のに、字面は逆から読まないといけません。**パイプライン演算子 `|>`** はこれを直します。`x |> f` は `f(x)` の糖衣です。

```
lune> 5 |> inc  
6 : Int  
lune> 5 |> inc |> double  
12 : Int
```

左から右へ、データが変換を通り抜けていく順に読めます。さらに、多引数関数へパイプすると**部分適用**になります。 `pipeline.lune` :

```
module pipeline

def inc(n: Int): Int =
  n + 1

def double(n: Int): Int =
  n * 2

def add(x: Int, y: Int): Int =
  x + y

# x |> f は f(x) の糖衣。左から右へ、データの流れの順に読める。
let result = 5 |> inc |> double

# 多引数関数へのパイプは部分適用になる。
let addFive = 5 |> add

let eleven = addFive(6)
```

```
$ lune --eval result pipeline.lune
12
$ lune --eval eleven pipeline.lune
11
```

パイプラインは「変換の手順書」を書くための構文です。第8章でリスト処理と組み合わせると本領を発揮します。

3.7 再帰

関数が自分自身を呼ぶ — ループの関数型的な書き方です。フィボナッチ数で見ましよう。

```
def fib(n: Int): Int =
  if n <= 1 then n else fib(n - 1) + fib(n - 2)
```

```
lune> fib(10)
55 : Int
```

再帰関数には約束が一つあります。**戻り値の型を書くこと**。書かないとどうなるか、`norettype.lune` で確かめます。

```
module bad

def fact(n: Int) =
  if n == 0 then 1 else n * fact(n - 1)
```

```
$ lune --check norettype.lune
```

```
error[TYP0011]: 再帰関数には戻り値型の注釈が必要です: fact
--> norettype.lune:3:1
|
3 | def fact(n: Int) =
|   ^^^ 型が確定する前に関数が自分自身を呼んでいる
    = hint: 戻り値型を追加してください。例: `def fact(...): T = ...`
    = help: 詳しくは `lune explain TYP0011 --lang ja` を実行してください
```

理由も診断が教えてくれています。本体の型を推論し終わる前に `fact(n - 1)` が現れるので、型検査は `fact` の型を先に知っておく必要があるのです。第4章で見たとおり、再帰していいのは関数だけ（値の再帰は `RUN0005`）、そして再帰関数には型の看板が必須 — この2つで再帰は安全に使えます。

書き方のコツを一つ: **停止条件を先に書く**。 `if n == 0 then ... else 再帰` の形をまず作ってから、再帰の側を埋めると、無限再帰を書きにくくなります。

壊してみよう — 関数でない値を呼び出すと、専用の診断が出ます。

```
lune> let x = 42
ok
lune> x(1)
error[TYP0004]: 呼び出せない値です: Int
--> <repl:21>:1:20
|
1 | x(1)
|
|               ^ この値は呼び出せない
    = help: 詳しくは `lune explain TYP0004 --lang ja` を実行してください
```

「`Int` は呼べない」 — 括弧の付けすぎ（`f(x)(y)` のつもりが `f(x, y)` だった、の逆）で出会うことが多い診断です。

まとめ

概念	一言で
<code>def f(x: T): U =</code> 本体	関数定義。型は看板、再帰なら戻り値型は必須
関数型	<code>Int -> Int</code> 。 <code>-></code> は右結合で、部分適用の読み方と一致
<code>fn x -> 式</code>	ラムダ。文脈があれば型注釈は省略可、なければ TYP0010
高階関数	関数を受け取る/返す関数。返したラムダは環境を覚える (クロージャ)
部分適用	引数が足りない呼び出しは「残りを待つ関数」
<code>x > f</code>	<code>f(x)</code> の糖衣。多引数関数へは部分適用
<code>:type f</code>	関数の型を確認する習慣

演習問題

演習 3-1 (★) `double(inc(inc(5)))` をパイプラインに書き換えてください (`inc` ・ `double` は本文のもの)。

解答

```
lune> 5 |> inc |> inc |> double
14 : Int
```

`double(inc(inc(5)))` も `14 : Int`。同じ計算ですが、パイプ版は実行の順 (`5` → `+1` → `+1` → `×2`) のまま読めます。

演習 3-2 (★★) 2つの関数を合成する `compose` を書いてください。 `compose(f, g)` は「`g` を適用してから `f` を適用する」1つの関数を返すものとします。

解答

```
module answers

# 演習 3-2: 関数合成。compose(f, g) は「g してから f」。
def compose(f: Int -> Int, g: Int -> Int): Int -> Int =
  fn x -> f(g(x))

def inc(n: Int): Int =
  n + 1

def double(n: Int): Int =
  n * 2

let incThenDouble = compose(double, inc)

let answer = incThenDouble(5)
```

```
$ lune --eval answer ex3-2.lune
12
```

返すラムダが `f` と `g` を覚えている — クロージャ (3.4節) の練習でもあります。

演習 3-3 (★★) `fib` を自分で書いて `fib(10)` を確かめてください。そのあと戻り値型 `: Int` を消して、どの診断が出るか予想してから `--check` してください。

解答

```
module answers

# 演習 3-3: 再帰でフィボナッチ数。戻り値型 Int は必須 (TYP0011)。
def fib(n: Int): Int =
  if n <= 1 then n else fib(n - 1) + fib(n - 2)

let answer = fib(10)
```

```
$ lune --eval answer ex3-3.lune
55
```

戻り値型を消すと `TYP0011` (3.7節と同じ診断) が出ます。

演習 3-4 (★★★) `applyTwice` を一般化して、関数 `f` を `n` 回適用する `applyN(f, n, x)` を書いてください。 `applyN(double, 3, 1)` が 8 になれば正解です。

解答

```
module answers

# 演習 3-4: applyTwice の一般化。f を n 回適用する。
def applyN(f: Int -> Int, n: Int, x: Int): Int =
  if n == 0 then x else applyN(f, n - 1, f(x))

def double(n: Int): Int =
  n * 2

let answer = applyN(double, 3, 1)
```

```
$ lune --eval answer ex3-4.lune
8
```

高階関数と再帰の合わせ技です。停止条件（`n == 0` なら `x` をそのまま返す）を先に書き、再帰側では「1回適用して、残り回数を減らす」と読めるようにします。

演習 3-5 (★・逆転問題) `TYP0004` (呼び出せない値です) を出す最小のコードを書いてください。

解答

最小は `42(1)` です (`error[TYP0004]: 呼び出せない値です: Int`)。数値・文字列・タプルなど、関数でない値に `(...)` を付ければ出ます。逆に `greet()` のような**引数不足**が `TYP0004` にも `TYP0005` にもならない理由を説明できれば、この章は卒業です。

より正確には — 関数定義・ラムダ・部分適用は `documents/LANGUAGE_SPEC.md` §8、関数型の構文は `documents/FUNCTION_TYPE_SPEC.md`、部分適用と遅延の関係は `documents/LAZY_EVALUATION_SPEC.md` §7、ラムダの型推論は `documents/LOCAL_TYPE_INFERENCE_SPEC.md`。この章のコード例は `books/examples/ch03/` にあり、すべて実際の CLI で検証されています。

第4章 遅延評価 — Lune の心臓部

ほとんどのプログラミング言語は、書いた順に計算します。 `let x = 重い計算` と書けば、その行で重い計算が走る — これを**正格評価**といいます。Lune は違います。**値は、必要になるまで計算されません**。これを**遅延評価**といい、Lune のあらゆる設計判断の根にある性質です。

遅延評価は「初学者の罠」と言われることがあります。いつ計算が走るのか見えないからです。Lune の答えは、隠すことではなく**見えるようにすること**でした。この章では REPL の `:thunks` と `:trace` という2つの観察道具を使って、評価が起きる瞬間を自分の目で確かめながら進みます。信じる必要はありません。すべてその場で観察できます。

4.1 let は約束である

第1章で、`let` に対して REPL が `ok` としか言わないのを見ました。値を表示しないのは、**まだ値がない**からです。確かめましょう。ここで観察用の組み込み関数 `crash()` を紹介します。評価されると必ず実行時エラーになる、いわば地雷です。

```
lune> let boom = crash()
ok
lune> boom
error[RUN0006]: crash() が評価されました
  = help: 詳しくは `lune explain RUN0006 --lang ja` を実行してください
```

地雷を束縛した行では、何も起きませんでした。爆発したのは `boom` を**使った瞬間**です。

`let name = 式` は「命令」ではなく「約束」です — 「`name` の値が必要になったら、この式で計算する」という。正格な言語の `let` が領収書だとすれば、Lune の `let` は注文票です。

4.2 サンク — 約束の中身

この「まだ計算していない式」の入れ物を**サンク** (think) と呼びます。サンクは式とその環境を閉じ込めていて、次の3つの状態のどれかにあります。

状態	意味
unevaluated	まだ一度も必要とされていない
evaluated = 値	計算済み。値を記憶している
failed = エラー	計算したら失敗した。失敗を記憶している

REPL の `:thunks` コマンドは、束縛の状態を**評価を一切起こさず**に覗きます。サンクの一生を観察しましょう。

```
lune> let x = 1 + 1
ok
lune> :thunks x
x : unevaluated
lune> x
2 : Int
lune> :thunks x
x : evaluated = 2
```

`1 + 1` は、`x` を参照した瞬間に計算されました。先ほどの `boom` はどうなっているでしょう。

```
lune> :thunks boom
boom : failed = error[RUN0006] crash() が評価されました
```

失敗も記憶されています。この意味はあとで（4.5節）はっきりします。

用語 — サンクを評価させることを「**force する**（強制する）」といいます。「`x` を force する」＝「`x` の約束を今すぐ果たさせる」です。

4.3 いつ評価されるのか — `:trace` で見る

では、force はいつ起きるのでしょうか。 `:trace on` を打つと、REPL は force が起きるたびに実況してくれます。

```
lune> :trace on
trace on
lune> let x = 1 + 1
ok
lune> x + 1
force x + 1
  force 1 + 1
  => 2
=> 3
3 : Int
```

読み方はこうです。

- `force` 式 — その式のサンクの評価に入った（インデントは入れ子の深さ）
- `=>` 値 — 対応する `force` が完了した

`let x = 1 + 1` の行ではトレースに**何も出ていない**ことに注目してください。宣言は `force` を起こさないのです。`x + 1` を打った瞬間、まず式全体が `force` され、その中で `x` の中身 `1 + 1` が `force` されました。

もう一度 `x` を使うと、3つ目のイベントが現れます。

```
lune> x
force x
  memo 1 + 1 => 2
=> 2
2 : Int
```

`memo` は「計算済みの値に当たったので、再計算せずそれを使った」という印です。

大まかには、**値が「見られる」場所で `force` が起きます**。算術や比較の引数、`if` の条件、`match` の対象、そして REPL やファイル実行での表示。正確な一覧は付録Aに譲りますが、実務上は `:trace` に聞くのが一番確実です。

同じトレースはファイル実行でも使えます。 `trace_demo.lune` :

```
module trace_demo

let x = 1 + 1

let answer = x + x
```

```
$ lune --eval answer --trace trace_demo.lune
force x + x
  force 1 + 1
  => 2
  memo 1 + 1 => 2
=> 4
4
```

`x` は2回使われていますが、計算は1回だけ。2回目は `memo` です（トレースは `stderr` に出るので、リダイレクトすれば値と分離できます）。

4.4 引数も遅延される — 制御構文が自作できる

遅延されるのは `let` だけではありません。関数の引数も、デフォルトでサンクとして渡されます。

```
lune> def first(a: Int, b: Int): Int =
...   a
...
ok
lune> first(10, crash())
10 : Int
```

`crash()` を渡したのに爆発しません。 `b` は本体で使われないので、`force` される機会がなかったのです。

これは単なる省エネ以上の意味を持ちます。制御構文が普通の関数で書けるのです。ファイル `myif.lune` :

```
module myif

# 引数はデフォルトで遅延される。制御構文を自分で定義できる。
def myIf(c: Bool, a: Int, b: Int): Int =
  if c then a else b

def myAnd(a: Bool, b: Bool): Bool =
  if a then b else false

let taken = myIf(true, 1, crash())

let skipped = myAnd(false, crash())
```

```
$ lune --eval taken myif.lune
1
$ lune --eval skipped myif.lune
false
```

`myIf` は選ばれなかった側を評価しません。`myAnd` は左が `false` なら右を見ません — 短絡評価です。正格な言語では、`&&` や `if` が短絡するのは言語が特別扱いしているからで、同じものをユーザーが関数として書くことはできません。Lune では、`if` も `&&` も原理的には「ただの関数」で再現できます。言語の芯が小さくて済むのは、遅延評価のおかげです。

4.5 メモ化 — 二度は計算しない

4.3節の `memo` をきちんと確かめましょう。ここで2つ目の観察道具、`tick()` を紹介します。呼ばれるたびに内部カウンタを 1 増やしてその値を返す組み込み関数で、相棒の `tickCount()` はカウンタを増やさずに現在値を読みます。「計算が何回走ったか」の目撃者です。

```
lune> let t = tick()
ok
lune> tickCount()
0 : Int
lune> t
1 : Int
lune> t
1 : Int
lune> tickCount()
1 : Int
```

束縛しただけではカウンタは 0 のまま。`t` を初めて使った時に 1 回だけ走り、2回目の `t` は記憶された値を返しました。**サンの評価は高々一度**です。

失敗も同じです。4.2節で `failed` 状態が「失敗を記憶している」と言いました。証拠をお見せします。

```
lune> let bad = tick() + crash()
ok
lune> tickCount()
0 : Int
lune> bad
error[RUN0006]: crash() が評価されました
  = help: 詳しくは `lune explain RUN0006 --lang ja` を実行してください
lune> tickCount()
1 : Int
lune> bad
error[RUN0006]: crash() が評価されました
  = help: 詳しくは `lune explain RUN0006 --lang ja` を実行してください
lune> tickCount()
1 : Int
```

2回目の `bad` も同じエラーを出しましたが、カウンタは1のまま — **失敗した計算は再実行されていません**。記憶していた失敗を再送しただけです。

このおかげで、Lune の変数はいつ何度読んでも同じ結果（同じ値、または同じ失敗）になります。「読むタイミングで結果が変わる」ことがないので、遅延がプログラムの意味を壊さないのです。

4.6 使う分しか作らない — 遅延がくれるもの

ここまでは遅延の**仕組み**を見てきました。この節は、その仕組みが何をくれるのかです。

第1章で使った `range(start, end)` を、少し無茶な幅で呼んでみます。

```
lune> let huge = range(1, 1000000000)
ok
lune> take(huge, 5)
(1 2 3 4 5) : List[Int]
lune> take(filter(huge, fn x -> x % 7 == 0), 3)
(7 14 21) : List[Int]
```

1億要素のリストから5個取り出し、7の倍数を3個拾いました。**どちらも一瞬で返ります**。メモリも増えません。

種は4.1節と同じです。`range` は1億個のセルを作ってから返すのではなく、**先頭のセル1つと「残りを作る約束」**を返します。`filter` も `take` も同じ形をしているので、`take(..., 3)` が3個目を受け取った時点で、その先の約束は誰にも `force` されないまま残ります。作られなかったものには、時間もメモリもかかりません。

「1億個のリストを作ってから5個取る」のではなく、「5個取るのに必要なだけ `range` が伸びる」。リストの長さは、消費者が決めているのです。

4.5 節の `tick()` に数えてもらいましょう。

```
lune> let xs = map(range(1, 100000000), fn x -> tick())
ok
lune> tickCount()
0 : Int
lune> take(xs, 3)
(1 2 3) : List[Int]
lune> tickCount()
3 : Int
```

1億要素のリストに `map` をかけて、評価は **3 回**。束縛しただけでは0回です。正格な言語なら、この式は1億回の関数呼び出しと数ギガバイトの確保を意味します。ここでは3回でした。

境界 — 全部要る計算は、やはり全部歩く

ただし、遅延は「計算しなくてよくする」ものではありません。**要求を後ろにずらすもの**です。答えを出すのに全要素が必要なら、その代金はきちんと請求されます。

```
lune> length(filter(huge, fn x -> x < 10))
```

手元では**7分ほど**かかりました（待つ気がなければ Ctrl-C で構いません）。答えは **9** です。9より大きい要素の先にもう一致するものが無いことは、`filter` には1億番目まで歩かないと分かりません。`length` は最後まで数える関数なので、全部歩くことになります。これは Lune の実装が弱いのではなく、遅延評価という考え方の**正しい帰結**です。同じ式を Haskell で書いても同じだけ歩きます。

ここでの条件 `x < 10` は昇順の列に対して単調なので、本当に言いたいことは「条件が崩れたら止めてよい」でした。それを言う関数が別にあります。

```
lune> takeWhile(huge, fn x -> x < 10)
(1 2 3 4 5 6 7 8 9) : List[Int]
```

`takeWhile` は最初の `false` で打ち切るので、リスト**全体**が一瞬で返ります。`filter` は「全部見ないと終われない」、`takeWhile` は「終われる」 — この違いが、遅延の世界では速度の差として現れます。

同じ理由で、`huge` をそのまま REPL に表示させてはいけません。表示は全要素を要求します。印字の前には `take` を挟んでください。この見分け方 — **答えを出すのに全要素**

が要るか — は第8章 8.5 節で、無限リストを相手に本格的に扱います。 `range` の幅が十分に大きければ、それは実質的に無限リストと同じものだからです。

4.7 正格化の工具箱

デフォルトは遅延。でも「今すぐ評価したい」場面はあります。Lune は明示的なオプアウトの道具を段階別に揃えています。

道具	書く場所	意味
<code>strict let x = 式</code>	束縛	束縛した瞬間に評価する
<code>strict x: T (略記 !x)</code>	関数の引数	呼び出しの瞬間に評価する
<code>strict field: T</code>	コンストラクタのフィールド	構築の瞬間に評価する
<code>seq a b</code>	式	<code>a</code> を評価してから <code>b</code> を返す
<code>deepForce x</code>	式	<code>x</code> を中身まで残らず評価する
<code>lazy 式 / force 式</code>	式	遅延を型 <code>Lazy[T]</code> として持ち歩く

strict let — 束縛時に評価します。 `:thunks` で見ると、もはやサンクですらありません。

```
lune> strict let s = tick()
ok
lune> tickCount()
1 : Int
lune> :thunks s
s : value = 1
```

strict 引数 — 4.4節の `first` を正格引数で書き直すと、使わない引数でも呼び出し時に評価されます。

```
lune> def strictFirst(strict a: Int, strict b: Int): Int =
...     a
...
ok
lune> strictFirst(10, crash())
error[RUN0006]: crash() が評価されました
= help: 詳しくは `lune explain RUN0006 --lang ja` を実行してください
```

strict フィールド — コンストラクタのフィールドも通常は遅延です（`Box(crash())` を作っても爆発しない）が、`strict` を付ければ構築時に評価されます。

```
module point

# strict 付きの正格フィールドは、構築の時点で評価される。
type Point =
  | Point(strict x: Int, strict y: Int)

let p = Point(crash(), 0)
```

```
$ lune --eval p point.lune
error[RUN0006]: crash() が評価されました
= help: 詳しくは `lune explain RUN0006 --lang ja` を実行してください
```

「不正な値を持った `Point` は一瞬たりとも存在させない」 — データ型の設計道具として第5～6章で再登場します。

seq — `seq a b` は「`a` を評価してから `b` を返す」。評価の順序だけを制御したいときに使います。

```
lune> let a = tick()
ok
lune> let answer = seq a 42
ok
lune> answer
42 : Int
lune> :thunks a
a : evaluated = 1
```

`answer` を使った瞬間、値としては関係のない `a` が道連れで評価されました。

deepForce — `force` は普段、値の「外側」までしか評価しません。中身まで残らず評価したいときは `deepForce` です。

```
lune> let pair = (tick(), tick())
ok
lune> :thunks pair
pair : unevaluated
lune> deepForce pair
(2, 3) : Tuple[Int, Int]
lune> :thunks pair
pair : evaluated = (2, 3)
```

lazy / force — 「すべてが遅延なのに、なぜ `lazy` が要るの?」と思うかもしれません。`lazy` 式 は遅延を**型として**持ち歩くための道具です。

```
lune> let delayed = lazy (40 + 2)
ok
lune> :type delayed
delayed : Lazy[Int]
lune> force delayed
42 : Int
```

`Int` と `Lazy[Int]` は別の型です。関数の戻り値やデータ構造のフィールドに `Lazy[T]` と書けば、「これはまだ計算されていないかもしれない」という事実が型シグネチャに現れ、受け取った側は `force` で明示的に開けることになります。暗黙の遅延を、契約に格上げする道具です。

4.8 底なしの再帰 — RUN0005

遅延評価の名物トラブルを見ておきましょう。「自分自身を使って自分を定義する」とどうなるか。

```
module recursive

let x = x + 1
```

`x` の値を知るには `x` の値が要る — 待っていても答えは出ません。まず型検査は、この素直な形を入り口で弾きます（定義中の名前はまだスコープにないので、未定義名になります）。

```
$ lune --check recursive.lune
```

```
error[TYP0001]: 未定義の名前: x
--> recursive.lune:3:9
|
3 | let x = x + 1
|           ^ この名前は定義されていない
= help: 詳しくは `lune explain TYP0001 --lang ja` を実行してください
```

しかし循環は、静的検査をすり抜けて実行時に初めて現れることもあります。v0.1 の `--eval` は型検査を通さず直接実行するので、このファイルで体験できます。無限ループになるのでしょうか？

```
$ lune --eval x recursive.lune
```

```
error[RUN0005]: 再帰的なサンク評価: この値の定義が自分自身の結果に依存しています
  = hint: 再帰的な値は計算できません。再帰関数（`def`）として書くか、参照の循環を断ち切ってください
  = help: 詳しくは `lune explain RUN0005 --lang ja` を実行してください
```

なりません。**即座に検出されます**。仕組みは 4.2 節のサンク状態にあります。サンクは評価に入る時、内部的に「評価中」の印を付けます。評価の途中で同じサンクをもう一度 force しようとしたら、それは定義が自分自身に戻ってきたということ — 待つだけ無駄だと分かるので、その場で `RUN0005` を報告するのです。`lune explain RUN0005` がこの仕組みごと解説してくれます。

コラム: <<loop>> — 遅延評価の本家 Haskell (GHC) も同種の検出を持っていますが、報告は <<loop>> という素っ気ない一言で、しかも検出できず本当にループすることもあります。Lune はこの検出を診断コード付きの教材にしました。「なぜ無限ループせずに済むのか」まで `explain` で学べます。

なお、**再帰関数は問題ありません**。

```
lune> def fact(n: Int): Int =
...   if n == 0 then 1 else n * fact(n - 1)
...
ok
lune> fact(5)
120 : Int
```

`def` の本体は呼ばれるまで走らないので、`fact(n - 1)` は自分の定義を force しません。存在できないのは再帰的な**値**です。

4.9 遅延とどう付き合うか

道具は揃いました。最後に使いどころの指針です。

遅延が効く場面:

- **使わないかもしれない値** — デフォルト値、エラーメッセージ、分岐の片側。書くだけならタダです。
- **自作の制御構文・短絡するロジック** (4.4節)。
- **無限のデータ構造** — 遅延評価の白眉です。第8章をお楽しみに。

正格を選ぶべき場面:

- **順序が大事な副作用** — 出力が「表示に必要な順」に走ると混乱します。IOは正格に（第9章）。
- **計測とデバッグ** — 「この行の実行時間」を測りたいのに計算が後ろへ逃げていく、という時は `strict let` や `seq` で釘を刺します。
- **サンクの積み上がり** — ループで巨大な「未評価の計算の塔」を育ててしまうことがあります。第8章の `fold` で再訪します。

そして迷ったら、`:thunks` と `:trace` に聞く。この章であなたが使った道具は、そのまま実戦のデバッグ道具です。

壊してみよう — `0` で割る式を束縛して、サンクの一生を観察してください。

```
lune> let half = 1 / 0
ok
lune> half
error[RUN0006]: ゼロ除算です
  = hint: `/` の右オペランドが 0 に評価されました
  = help: 詳しくは `lune explain RUN0006 --lang ja` を実行してください
lune> :thunks half
half : failed = error[RUN0006] ゼロ除算です
```

束縛では何も起きず（遅延）、使うと失敗し（force）、失敗が記憶されました（メモ化）。この章の全部が3行に入っています。

まとめ

概念	一言で
遅延評価	値は必要になるまで計算されない
サンク	未評価の式の入れ物。 <code>unevaluated</code> / <code>evaluated</code> / <code>failed</code>
<code>force</code>	サンクに評価を強制すること。値が「見られる」場所で起きる
メモ化	評価は高々一度。成功も失敗も記憶される
使う分しか作らない	リストの長さは消費者が決める。 <code>range</code> の幅そのものには代価がない
全要素が要るか	要るなら歩き切る（ <code>length</code> / <code>fold</code> / 表示）。 要らないなら <code>take</code> / <code>takeWhile</code>
<code>:thunks</code> / <code>:trace</code>	状態を覗く / <code>force</code> を実況する。迷ったらこれ
<code>strict let</code> / <code>strict</code> 引数 / <code>strict</code> フィールド	遅延のオプトアウト
<code>seq</code> / <code>deepForce</code>	評価の順序と深さの制御
<code>lazy</code> / <code>force</code> と <code>Lazy[T]</code>	遅延を型で明示する
<code>RUN0005</code>	再帰的な値は存在できない。関数にせよ

演習問題

演習 4-1 (★) 次を順に打つと、それぞれの行で何が表示されるでしょう。予想してから確かめてください。

```
lune> let u = tick()
lune> let v = tick()
lune> v
lune> :thunks
lune> u
```

解答

```
lune> let u = tick()
ok
lune> let v = tick()
ok
lune> v
1 : Int
lune> :thunks
u : unevaluated
v : evaluated = 1
lune> u
2 : Int
```

先に定義した `u` が 2 で、後から定義した `v` が 1 です。カウンタは**定義した順**ではなく**使った順**に増えます。評価の順序を決めるのは、プログラムの字面ではなく需要 (demand) である — 遅延評価の核心です。

演習 4-2 (★★) `myAnd` (4.4節) にならって、左が `true` なら右を評価せずに `true` を返す `myOr` を書いてください。 `myOr(true, crash())` が爆発しないことが合格条件です。

解答

```
module answers

# 演習 4-2: 左が true なら右を評価せずに短絡する myOr。
def myOr(a: Bool, b: Bool): Bool =
  if a then true else b

let shortCircuited = myOr(true, crash())
```

```
$ lune --eval shortCircuited ex4-2.lune
true
```

`if` が評価するのは条件 `a` だけ。選ばれた枝が `true` (リテラル) なら、`b` のサンクは触られもしません。

演習 4-3 (★★) `let t = tick()` と `strict let s = tick()` を1つずつ定義し、`tickCount()` と `:thunks` で両者の違いを説明してください。

解答

定義直後に `tickCount()` を読むと 1 — `strict let` の分だけが走っています。 `:thunks` で見ると `t : unevaluated` に対し `s : value = 1`。 `s` はサンクを経由せず、束縛の時点で値になっています。 `t` が計算されるのは、この後 `t` を初めて使った瞬間です。

演習 4-4 (★★) リストの要素も遅延されます。では `let xs = [crash(), 1, 2]` に対して `length(xs)` は成功するでしょうか。予想してから確かめてください。

解答

```
lune> let xs = [crash(), 1, 2]
ok
lune> length(xs)
3 : Int
```

成功します。 `length` が数えるのはリストの**背骨**（要素の連なり）だけで、各**要素**の中身は `force` しません。「構造だけ評価して中身は評価しない」という区別は、第8章の無限リストを支える仕掛けそのものです。

演習 4-5 (★★★・逆転問題) `RUN0005` を実際に出してください。そのうえで、同じファイルが `--check` では `RUN0005` ではなく `TYP0001` になる理由を説明してください。

解答

最小の再現は `let x = x + 1` を含むファイルを `--eval x` で実行することです（4.8 節）。 `--check` で `TYP0001` になるのは、型検査が「定義の右辺を検査する時点では、定義しようとしている名前はまだスコープにない」という規則で、自己参照を**未定義名**として入り口で弾くからです。型検査は名前の循環を静的に防ぎ、評価器は実行時に現れた循環を `RUN0005` で検出する — 二段構えの防御になっています。

より正確には — サンクの状態遷移と `force` 境界の完全な一覧は

`documents/LAZY_EVALUATION_SPEC.md`、`:thunks` / `:trace` の表示仕様は `documents/REPL_SPEC.md` §5.1–5.2。 `crash` / `tick` / `tickCount` は `documents/STANDARD_LIBRARY_SPEC.md` §8.1 の観察用ビルトインです。この章のコード例は `books/examples/ch04/` にあり、すべて実際の CLI で検証されています。

第5章 代数的データ型とパターンマッチ

ここまでの型は、数・文字列・タプルといった「素材」でした。この章では、**自分のプログラムのための型**を作ります。道具は2つ。データの形を宣言する**代数的データ型**（ADT）と、形に沿って分解する `match` です。

この2つを覚えると、プログラムの書き方が変わります。「値をチェックしてフラグで分岐する」代わりに、「ありうる形をぜんぶ型に書き、コンパイラにケース漏れを見張らせる」ようになるのです。ケースが漏れたときに Lune が**反例つき**で教えてくれるところまで、この章で体験します。

5.1 「形」を型で表す

図形を扱うプログラムを考えます。図形は、円か、長方形か、そのどちらかだとしましょう。Lune ではその「どちらか」をそのまま書けます。

```
type Shape =  
  | Circle(radius: Double)  
  | Rect(width: Double, height: Double)
```

`Shape` 型の値は、`Circle` か `Rect` のどちらかの**コンストラクタ**で作られます。それ以外の形はありません。

```
lune> Circle(1.0)  
Circle(1.0) : Shape  
lune> Rect(3.0, 4.0)  
Rect(3.0, 4.0) : Shape
```

フィールドには名前を付けて宣言します（`radius: Double`）。この名前は宣言の読みやすさのためのもので、作るときは位置で渡します。

フィールドを持たないコンストラクタも作れます。信号の色のような「ただの選択肢」です。

```
type Color =  
  | Red  
  | Green  
  | Blue
```

```
lune> let g = Green()
ok
lune> g
Green : Color
```

v0.1 では、フィールドなしのコンストラクタも**呼び出して値にします**（`Green()`）。例外はプレリユードの `None` と `Nil`（リストの終端、第8章）で、これらは最初から値として登録されているため裸で使えます。

用語メモ — `Shape` のような「どれか一つ」の型を**直和型**と呼びます。タプルやレコードのような「全部持つ」型（直積型）と対になる概念で、両方を組み合わせて作る型がいわゆる代数的データ型です。設計の使い分けは第6章の終わりで整理します。

5.2 match — 形に沿って分解する

ADT の値を使う側は `match` です。第1章から予告していた本命の構文をきちんと見ます。

```
def area(s: Shape): Double =
  match s:
    | Circle(r) -> r * r * 3.14159
    | Rect(w, h) -> w * h
```

`match` は対象の値の**形**を上から順に調べ、最初に合った腕の式を評価します。

`Circle(r)` のようなパターンは、形を確かめると同時に、中身を `r` という名前で**取り出します**。検査と分解が一度に済む — これがパターンマッチの気持ちよさです。

`match` も式なので、値を返します。腕の本体が長ければ複数行にできます（インデントして書きます）。

パターンには色々な形が置けます。まとめて見ましょう。

```
lune> def sign(n: Int): String =
...     match n:
...       | 0 -> "zero"
...       | x if x < 0 -> "negative"
...       | _ -> "positive"
...
ok
lune> sign(0)
"zero" : String
lune> sign(-5)
"negative" : String
lune> sign(3)
"positive" : String
```

- **リテラルパターン** `0` — その値ちょうどに合う
- **名前パターン** `x` — 何にでも合い、値に名前を付ける
- **ガード** `x if x < 0` — パターンに追加の条件を付ける
- **ワイルドカード** `_` — 何にでも合い、名前も付けない

タプルも分解できます。

```
lune> def where(p: Tuple[Int, Int]): String =
...     match p:
...       | (0, 0) -> "origin"
...       | (x, 0) -> "x-axis"
...       | _ -> "elsewhere"
...
ok
lune> where((3, 0))
"x-axis" : String
```

ガードは `Shape` のような ADT と組み合わせると表現力を発揮します。 `shape.lune` :

```
# ガード: パターンに追加の条件を付ける。
def describe(s: Shape): String =
  match s:
    | Circle(_) -> "circle"
    | Rect(w, h) if w == h -> "square"
    | Rect(_, _) -> "rectangle"
```

```
$ lune --eval squareness shape.lune
"square"
```

5.3 網羅性 — コンパイラが反例をくれる

ここからが Lune の見せ場です。match は網羅的でなければなりません。つまり、ありうる形をすべてカバーしていなければ型エラーです。しかも Lune は「何が足りないか」を具体的な反例 (witness) で教えてくれます。

Color のケースを1つ書き忘れてみます。

```
lune> def label(c: Color): String =  
...   match c:  
...     | Red -> "warm"  
...     | Blue -> "cool"  
...  
error[TYP0007]: 網羅的でない match: Green のケースがありません  
--> <repl:2>:2:5  
  |  
2 |   match c:  
  |     ^^^^^ パターン Green がカバーされていない  
    = hint: Green のケースを追加するか、ワイルドカードケース `| _ -> ...` を追加し  
    てください  
    = help: 詳しくは `lune explain TYP0007 --lang ja` を実行してください
```

「Green が足りない」。この程度なら自分でも気づけそうですが、witness の真価は形が入れ子になったときです。missing.lune:

```
module bad  
  
# Some(false) のケースが漏れている。  
def toInt(o: Option[Bool]): Int =  
  match o:  
    | Some(true) -> 1  
    | None -> 0
```

```
$ lune --check missing.lune
```

```
error[TYP0007]: 網羅的でない match: Some(false) のケースがありません  
--> missing.lune:5:5  
  |  
5 |   match o:  
  |     ^^^^^ パターン Some(false) がカバーされていない  
    = hint: Some(false) のケースを追加するか、ワイルドカードケース `| _ -> ...` を  
    追加してください  
    = help: 詳しくは `lune explain TYP0007 --lang ja` を実行してください
```

`Some` は書いたし `None` も書いた — それでも「`Some(false)` が漏れている」とコンパイラは**反例を合成して**指摘します。ここまで来ると、網羅性検査は単なるお目付け役ではなく、**仕様の穴を見つける道具**です。「この関数、こういう入力のこと考えてた?」と、実例つきで聞いてくれるのですから。

2つ注意を。第一に、**ガード付きの腕は網羅性に数えられません**。`| x if x < 0 -> ...` がすべての負数を覆うことをコンパイラは証明できないからです。ガードを使ったら、受け皿になる無条件の腕を必ず置いてください。第二に、`| _ -> ...` は網羅性を一発で満たしますが、**型にケースを足したときに教えてもらえなくなる**という代償があります。列挙できるなら列挙する — 次の演習 5-1 で効果を体験できます。

5.4 到達しない腕と、失敗しうる束縛

網羅性の逆向きの検査もあります。**すでに覆われていて絶対に選ばれない腕**は警告になります。`unreachable.lune`：

```
def f(c: Color): Int =
  match c:
    | Red -> 1
    | Red -> 2
    | Green -> 3
```

```
$ lune --check unreachable.lune
```

```
warning[TYP0009]: 到達しない match ケース: Red
--> unreachable.lune:10:9
  |
10 |           | Red -> 2
   |           ^ このケースには決して到達しない
   = hint: このケースを削除するか、これをカバーしているケースより前に移動してください
   = help: 詳しくは `lune explain TYP0009 --lang ja` を実行してください
type check OK
```

`warning` なので検査自体は通ります（最後に `type check OK` が出ていることに注目）。CI で警告も落とした場合の作法は第12章で扱います。

もう一つ。パターンは `let` にも書けました（タプルの分解束縛、第2章）。ただし `let` に置けるのは**失敗しようがない**（反駁不能な）パターンだけです。`refutable.lune`：

```
module bad

let Some(x) = Some(1)
```

```
$ lune --check refutable.lune
```

```
error[TYP0008]: let の束縛に反駁可能パターンは使えません: Some(x)
--> refutable.lune:3:5
|
3 | let Some(x) = Some(1)
|    ^^^^ このパターンはマッチに失敗し得る
= hint: このパターンは None をカバーしていません
= hint: `match` を使って Option[Int] の全ケースを場合分けしてください
= help: 詳しくは `lune explain TYP0008 --lang ja` を実行してください
```

右辺が `Some(1)` だと分かっているけど、`Option[Int]` 型の値は `None` かもしれない、`let` では受けられません。hint が言うとおり、そういう値は `match` で開けます。タプルパターンが `let` に書けたのは、タプルに「別の形」がないからです。

5.5 ジェネリックな ADT — 入れ物を型引数で抽象化する

`Option` を自分で定義すると、こう書けます（実際、プレリユードの定義と同じ構造です）。

```
type Option[T] =
  | Some(value: T)
  | None

def getOrElse[T](option: Option[T], defaultValue: T): T =
  match option:
    | Some(value) -> value
    | None -> defaultValue
```

`[T]` は「任意の型 `T` について」 — `Option[Int]` にも `Option[String]` にも同じ定義が使えます。関数側も `def getOrElse[T](...)` と型引数を受け取り、`Option[T]` と `T` の関係（中身とデフォルト値は同じ型）を型で約束します。第3章で読んだ `map : [T, U] List[T] -> (T -> U) -> List[U]` と同じ仕組みです。

5.6 Option と Result を使う

プレリユードには `Option[T]`（値があるかないか）と `Result[T, E]`（成功か失敗か）が最初から入っています。基本の道具はこの4つです。

```
lune> getOrElse(Some(42), 0)
42 : Int
lune> getOrElse(None, 7)
7 : Int
lune> optionMap(Some(2), fn x -> x + 1)
Some(3) : Option[Int]
lune> unwrapOr(Err("boom"), 0)
0 : Int
lune> resultMap(Ok(20), fn x -> x * 2)
Ok(40) : Result[Int, E]
```

`optionMap` は「あれば変換、なければそのまま `None`」、`unwrapOr` は「成功なら中身、失敗ならデフォルト」。例外を投げる代わりに**失敗を値として運ぶ** — これが関数型の定番のエラー処理で、第13章のケーススタディで本格的に使います。

自分の関数から返してみましょう。整数の割り算は0では割れないので、結果を「失敗を値として運ぶ」型 `Option[Double]` で包みます。

```
lune> def maybeDiv(x: Int, y: Int): Option[Double] =
...     if y == 0 then None else Some(x / y)
...
ok
lune> maybeDiv(1, 2)
Some(0.5) : Option[Double]
lune> maybeDiv(1, 0)
None : Option[Double]
```

さりげないコードですが、型検査器が静かに良い仕事をしています。`None` は単体では「中身の型が何か」を知りません。裸のまま打つと、型引数が未確定で残ることが見えます。

```
lune> None
None : Option[T]
```

先ほどの `resultMap(Ok(20), ...)` の結果に `E` が残っていたのも同じ理由です — `Ok(20)` からは成功側の型しか分かりません。では `maybeDiv` の中の `None` はなぜ `Option[Double]` になれたのか。関数の戻り値注釈 `Option[Double]` が**期待型**として `if` の両分岐に配られ、未確定だった `T` をそこで確定させるからです。期待型は `let` の型注釈からも流れます。

```
lune> let nothing: Option[Double] = None
ok
lune> nothing
None : Option[Double]
```

期待型は `match` の腕にも配られます。0 除算を「理由付きの失敗」として返すなら `Result` で。 `maybediv.lune` :

```
# match の各腕にも期待型が分配される。Err の T、Ok の E も同様に確定する。
def safeDiv(x: Int, y: Int): Result[Double, String] =
  match y:
    | 0 -> Err("div by zero")
    | _ -> Ok(x / y)
```

```
$ lune --eval ratio maybediv.lune
Ok(4.5)
```

`Err("div by zero")` からは失敗側の型しか、`Ok(x / y)` からは成功側の型しか分かりませんが、期待型 `Result[Double, String]` が残りの型引数を埋めてくれます。

先取り: null 許容型でも同じ — この「期待型が分岐に配られて合流する」仕組みは、第7章で学ぶ null 許容型 `T?` でも働きます。

```
# 先取り (第7章): null 許容型でも同じ仕組みで分岐が合流する。
def nullDiv(x: Int, y: Int): Double? =
  if y == 0 then null else x / y
```

`null` の分岐と `Double` の分岐が `Double?` へ合流して、型検査を通ります。
`Option` と `T?` の使い分けは第7章で整理します。

まとめ

概念	一言で
<code>type T = A(...) B(...)</code>	直和型。値は必ずどれか一つのコンストラクタ
フィールドなしコンストラクタ	<code>Green()</code> と呼んで値にする（プレリユードの <code>None / Nil</code> だけは裸で値）
<code>match</code>	形の検査と分解を同時に行う式
パターン	リテラル / 名前 / <code>_</code> / タプル / コンストラクタ / ガード <code>if</code>
TYP0007	ケース漏れ。 反例つき で教えてくれる
TYP0009	到達しない腕（警告）
TYP0008	<code>let</code> に失敗しうるパターンは置けない
<code>type Option[T] = ...</code>	ジェネリック ADT。関数側は <code>def f[T](...)</code>
<code>Option / Result</code>	プレリユード提供。 <code>getOrElse / optionMap / unwrapOr / resultMap</code>
期待型	返り値注釈や <code>let</code> 注釈が分岐へ配られ、 <code>None / Err(...)</code> の型引数を確定

演習問題

演習 5-1 (★) 本文の `Color` に `Yellow` を追加してください（`label` はそのまま）。どの診断が出るか予想してから `--check` し、`| _ -> ...` で網羅性を満たした場合との違いを考えてください。

解答

`error[TYP0007]`: 網羅的でない `match`: `Yellow` のケースがありません が `label` に出ます。型にケースを足した瞬間、**その型を `match` しているすべての場所**をコンパイラが洗い出してくれる — これが「列挙できるなら `_` より列挙」の理由です。`| _ -> "unknown"` と書いてあったら、`Yellow` は黙って `"unknown"` に落ち、誰も気づきません。

演習 5-2 (★★) 信号機の色 `Light` (`Red / Yellow / Green`) と、「次の色」を返す `next` を書いてください。赤→青→黄→赤の順とします。

解答

```
module answers

# 演習 5-2: 信号機の次の色。網羅性検査が全色の考慮を保証する。
type Light =
  | Red
  | Yellow
  | Green

def next(light: Light): Light =
  match light:
    | Red -> Green()
    | Green -> Yellow()
    | Yellow -> Red()

let afterRed = next(Red())

let afterTwo = next(next(Red()))
```

```
$ lune --eval afterTwo ex5-2.lune
Yellow
```

パターンでは裸の `Red`、値を作るときは `Red()`。腕を1本消して `TYP0007` を出してみるのも良い復習です。

演習 5-3 (★★) 整数を検査して、0以上なら値を、負なら理由を返す仕組みを作ってください。プレリユードの `Result[Int, String]` でも書けますが、ここでは自分用の結果型（型引数なし）を定義すること。コンストラクタ名を自分のドメインの言葉にできる、という ADT の利点を味わう練習です。

解答

```
module answers

# 演習 5-3: 検査の結果を、成功と失敗の両方の情報ごと型にする。
type Checked =
  | Valid(value: Int)
  | Invalid(reason: String)

def check(n: Int): Checked =
  if n >= 0 then Valid(n) else Invalid("negative")

def report(c: Checked): String =
  match c:
    | Valid(v) -> "ok: " + show(v)
    | Invalid(reason) -> "rejected: " + reason

let good = report(check(5))

let bad = report(check(-3))
```

```
$ lune --eval bad ex5-3.lune
"rejected: negative"
```

「成功にも失敗にも運びたい情報がある」とき、Bool や特別な値 (-1 など) ではなく型で表す — ADT 設計のいちばん実用的な型紙です。5.6節の仕組みがあるので

`Result[Int, String]` を使う版もそのまま書けますが、自分用の型なら `Valid / Invalid` という**ドメインの言葉**が `match` の腕や診断にそのまま現れます。

演習 5-4 (★★★) プレリユードを使わずに `MyOption[T]` と `myGetOrElse` を自作し、`MySome(42)` と `MyNone()` の両方で動くことを確かめてください。

解答

```
module answers

# 演習 5-4: ジェネリックな入れ物を自作する。prelude の Option と同じ構造。
type MyOption[T] =
  | MySome(value: T)
  | MyNone

def myGetOrElse[T](option: MyOption[T], defaultValue: T): T =
  match option:
    | MySome(value) -> value
    | MyNone -> defaultValue

let some = myGetOrElse(MySome(42), 0)

let none = myGetOrElse(MyNone(), 0)

# 期待型による確定 (5.6節) は自作のジェネリック ADT にも働く。
let empty: MyOption[Int] = MyNone()
```

```
$ lune --eval some ex5-4.lune
42
$ lune --eval none ex5-4.lune
0
```

型引数の決まり方が2通り見えます。MySome(42) の `T = Int` は**引数から決まり**、MyNone() の `T` はもう一方の引数 `0` との単一化で決まります。また、5.6節の期待型による確定は自作のジェネリック ADT にも働くので、`let empty: MyOption[Int] = MyNone()` のように注釈から確定させることもできます。

演習 5-5 (★・逆転問題) TYP0008 (反駁可能パターン) を出す最小のコードを書き、診断の2つの hint がそれぞれ何を教えているか説明してください。

解答

`let Some(x) = Some(1)` (5.4節)。1つ目の hint 「このパターンは None をカバーしていません」は**なぜ失敗**しうるか (覆えていない形の名指し)、2つ目の「`match` を使って全ケースを場合分けしてください」は**どうすればいいか** (代わりの構文)。診断は「原因 → 対処」の順で読む、という第11章の読み方の予行演習です。

より正確には — ADT と `match` の構文は documents/LANGUAGE_SPEC.md §10-11、網羅性検査と witness の算法は documents/MATCH_EXHAUSTIVENESS_SPEC.md、Option/Result の全 API は documents/STANDARD_LIBRARY_SPEC.md §4-5、期待型がコンストラクタの型引数を確定する規則は documents/LOCAL_TYPE_INFERENCE_SPEC.md §5.3・§5.6。なお仕様書には OR パターン (`| 0 | 1 ->`) が載っていますが v0.1 では未

実装です。この章のコード例は `books/examples/ch05/` にあり、すべて実際の CLI で検証されています。

第6章 レコード

タプル（第2章）は値を「束ねる」道具でしたが、`(36, "Ada")` の 36 が年齢なのか背番号なのかは、書いた本人しか知りません。値に**名前**を付けて束ねたくなったら、**レコード**の出番です。第5章の ADT が「どれか一つ」（直和）を表すのに対し、レコードは「全部持つ」（直積）を名前付きで表します。

6.1 宣言と構築 — 名前で作る

```
record User:
  name: String
  age: Int
```

フィールドの名前と型を並べるだけです。作るときは**必ずフィールド名を指定**します。

```
lune> let ada = User(name = "Ada", age = 36)
ok
lune> ada
{ name = "Ada", age = 36 } : User
```

表示も `{ フィールド = 値 }` の形です。位置で渡すことはできません。

```
lune> User("Ada", 36)
error[REC0006]: User の構築には名前付きフィールドが必要です
--> <repl:2>:1:25
  |
1 | User("Ada", 36)
  |
      ^^^^^ フィールド = 値 の形で書く
= help: 詳しくは `lune explain REC0006 --lang ja` を実行してください
```

ここは ADT のコンストラクタ（位置渡し、第5章）と対照的です。レコードはフィールドが増えがちで、`User("Ada", 36)` の 36 が何かは型が同じ限り機械にも人にも判別できない — だから名前を強制する、という設計です。おかげで構築まわりの間違いは、すべて名指しで叱ってもらえます。足りなければ `REC0003`（フィールドが不足）、知らない名前なら `REC0005`（宣言されていないフィールド）、同じ名前を2回書けば `REC0004`（フィールドの二重指定）です。

6.2 フィールドアクセスと遅延

読み出しは `.` です。

```
lune> ada.name
"Ada" : String
lune> ada.age + 1
37 : Int
```

タイプにも `did-you-mean` が効きます。 `typofield.lune` :

```
$ lune --check typofield.lune
```

```
error[REC0002]: 存在しないレコードフィールド: User.nmae
--> typofield.lune:9:12
|
9 | let oops = ada.nmae
|               ^^^ このレコードにそのフィールドは宣言されていない
  = hint: もしかして `name` ですか?
  = help: 詳しくは `lune explain REC0002 --lang ja` を実行してください
```

そして第4章の住人ならもう予想がつくはずですが — **フィールドも遅延されます**。

```
lune> let ghost = User(name = "Ghost", age = crash())
ok
lune> ghost.name
"Ghost" : String
lune> :thunks ghost
ghost : evaluated = { name = "Ghost", age = <thunk> }
```

`age` に地雷を入れたまま `name` は平気で読めました。 `:thunks` のプレビューでは、触っていないフィールドが `<thunk>` のまま見えています。

構築の時点で評価してほしいフィールドには `strict` を付けます。

```
lune> record Tagged:
...     strict tag: Int
...     note: String
...
ok
lune> let t = Tagged(tag = crash(), note = "n")
ok
lune> t.note
error[RUN0006]: crash() が評価されました
  = help: 詳しくは `lune explain RUN0006 --lang ja` を実行してください
```

`note` を読んだだけなのに `tag` の地雷が爆発しました。`t` を最初に `force` した瞬間にレコードが構築され、`strict` フィールドはそのとき評価されるからです。「不正な値を抱えたレコードを存在させない」 — 第4章で予告した設計道具です。

6.3 ジェネリックなレコード

型引数も取れます。第5章の ADT と同じ書き方です。

```
lune> record Box[T]:  
...   value: T  
...  
ok  
lune> Box(value = 42)  
{ value = 42 } : Box[Int]
```

フィールドの値から `T = Int` が推論されました。関数側も型引数付きで書けます（演習 6-3 で、型が入れ替わる `swap` を作ります）。

6.4 タプル・ADT・レコードの使い分け

「値をまとめる」道具が3つ揃いました。整理しておきます。

道具	向いている場面	例
タプル	その場限りの2〜3個の組。名前を付けるほどでもない	<code>zip</code> の結果、 <code>(商, 余り)</code>
レコード	同じ形を何度も使う。フィールドに名前が要る	<code>User</code> 、設定、集計結果
ADT	「どれか一つ」の選択肢がある	<code>Shape</code> 、 <code>Option</code> 、状態

組み合わせるのが普通です。レコードのリストを第1章以来の道具で加工する例を見ましょう。`items.lune`：

```

module items

record Item:
  name: String
  price: Int

let items = [Item(name = "pen", price = 120), Item(name = "note", price = 200)]

let total = fold(map(items, fn i: Item -> i.price), 0, fn a x -> a + x)

```

```

$ lune --eval total items.lune
320

```

「レコードのリストから、フィールドを `map` で抜き、`fold` で畳む」— 実務の Lune プログラムの背骨になるパターンです。

6.5 まだできないこと — update と pattern

v0.1 のレコードには、隣の言語にある2つの機能がまだありません。

record update (`{ ada | age = 37 }` のような「一部だけ変えたコピー」) はないので、新しい値は全フィールドを書いて作ります。**record pattern** もないので、`match` でレコードは分解できません。

```

lune> match ada:
...   | { name = n } -> n
...
error[PRS0001]: パターンが必要ですが、LBRACE が見つかりました
--> <repl:3>:2:7
  |
2 |   | { name = n } -> n
  |   ^ 予期しないトークン
    = help: 詳しくは `lune explain PRS0001 --lang ja` を実行してください

```

レコードの中身が要るときは `.` で読む、条件分岐はフィールドの値に対して `if / match` する、と覚えてください。どちらも将来仕様 (付録E) には入っています。

壊してみよう — レコード構築の診断 REC0003 (フィールド不足) ・ REC0004 (同じフィールドを2回) ・ REC0005 (知らないフィールド) を、`User` でひとつずつ出してみてください。すべて構築式の上で名指しされることを確認しましょう。第11章で学ぶコード体系のうち、REC 族はこの章がホームグラウンドです。

まとめ

概念	一言で
<code>record R: + フィールド宣言</code>	直積を名前付きで。構築は <code>R(field = value)</code> 必須
<code>r.field</code>	読み出し。タイプは <code>REC0002 + did-you-mean</code>
フィールドの遅延	デフォルト遅延。 <code>strict field: T</code> で構築時評価
<code>record Box[T]:</code>	ジェネリックレコード
使い分け	その場の組→タプル / 名前つき直積→レコード / どれか一つ→ADT
未対応	record update / record pattern (読み出しは . で)

演習問題

演習 6-1 (★) `User` の構築を3通りに壊して、`REC0003` ・ `REC0005` ・ `REC0006` をそれぞれ出してください (どの壊し方がどのコードになるか、先に予想を)。

解答

`User(name = "X")` → `REC0003` (age が足りない)、`User(name = "X", years = 1)` → `REC0005` (years は知らない名前)、`User("X", 36)` → `REC0006` (名前なし構築)。ついでに `User(name = "X", name = "Y", age = 1)` で `REC0004` も出せます。

演習 6-2 (★★) `items.lune` の品物のうち、150円以上のものの**名前だけ**のリストを作ってください。

解答

```
module answers
```

```
# 演習 6-2: 150 円以上の品物の名前だけを取り出す。
```

```
record Item:  
  name: String  
  price: Int
```

```
let items = [Item(name = "pen", price = 120), Item(name = "note", price  
= 200)]
```

```
let pricey = map(filter(items, fn i: Item -> i.price >= 150), fn i: Item  
-> i.name)
```

```
$ lune --eval pricey ex6-2.lune  
("note")
```

`filter` で絞ってから `map` で抜く。逆順（先に名前を抜く）だと価格が消えて絞れなくなる — パイプラインの順序は情報の寿命で決まります。

演習 6-3 (★★) ジェネリックなレコード `Pair[A, B]` (`first: A`、`second: B`) と、前後を入れ替える `swap` を書いてください。 `swap` の戻り値の型が `Pair[B, A]` になることに注意。

解答

```
module answers
```

```
# 演習 6-3: ジェネリックなレコードと、型が入れ替わる swap。
```

```
record Pair[A, B]:  
  first: A  
  second: B
```

```
def swap[A, B](p: Pair[A, B]): Pair[B, A] =  
  Pair(first = p.second, second = p.first)
```

```
let swapped = swap(Pair(first = 1, second = "a"))
```

```
$ lune --eval swapped ex6-3.lune  
{ first = "a", second = 1 }
```

`Pair(first = p.second, ...)` の型引数は渡した値から推論されるので、構築側に型を書く必要はありません。

演習 6-4 (★★★) 6.2節の `Tagged` (`strict tag`) と、`strict` なしの同型レコードを両方作り、`crash()` を `tag` に入れて構築したときの挙動の違いを、`:thunks` も使って説

明してください。

解答

`strict` なしなら `l.note` は `"n"` を返し、`:thunks l` は `{ tag = <thunk>, note = ... }` — 地雷は触るまで無害です。`strict` 付きは `t.note` の時点で `RUN0006` — `t` の `force` がレコード構築を走らせ、構築が `strict` フィールドを評価するからです。「遅延の既定 + 明示的な正格化」という第4章の原則が、データ設計にそのまま現れています。

より正確には — レコードの構文・型付け・遅延の規則は

`documents/RECORD_FIELD_SPEC.md` (`update / pattern` が §7・§6 で将来対応と明記されています)。この章のコード例は `books/examples/ch06/` にあり、すべて実際の CLI で検証されています。

第7章 null 安全

「値がないこと」を表す `null` は、多くの言語で実行時エラーの王様です。null チェックを1箇所忘れただけで、プログラムはある日突然落ちる — 発明者自身が「10億ドルの過ち」と悔やんだ仕掛けです。

Lune は `null` を廃止する代わりに、**型で飼いならす**道を選びました。null が入りうる場所を型 `T?` で宣言させ、確かめる前に使うことをコンパイラが許さない。第5章の `Option` と似た役割ですが、こちらは専用の構文（`??`、`?.`、絞り込み）を持つ軽量版です。使い分けは章の最後に整理します。

7.1 T? — null を持てる型

型に `?` を付けると、その型の値または `null` を持てるようになります。

```
lune> let present: Int? = 42
ok
lune> let absent: Int? = null
ok
lune> present
42 : Nullable[Int]
```

表示に出てくる `Nullable[Int]` は `Int?` の内部名です（読み替えてください）。`Int` の値はそのまま `Int?` の場所に入りますが、**逆は許されません**。

```
lune> def wantsInt(n: Int): Int =
...     n + 1
...
ok
lune> wantsInt(present)
error[TYP0003]: Int が必要ですが、Nullable[Int] が見つかりました
= help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

これが null 安全のすべてです。「null かもしれない値」は、null でないことを**確かめる**まで、普通の値として使えない。確かめる方法がこの章の残りです。

7.2 match で外す — null の被覆と絞り込み

一番基本の方法は `match` です。 `orzero.lune` :

```
module orzero

# T? には T の値も null も入る。
let present: Int? = 42

let absent: Int? = null

# null を match で被覆すると、残りの腕の v は Int に絞り込まれる。
def orZero(value: Int?): Int =
  match value:
    | null -> 0
    | v -> v
```

```
$ lune --eval unwrapped orzero.lune
42
$ lune --eval defaulted orzero.lune
0
```

| `null ->` の腕が `null` を引き受けたあとなので、次の腕の `v` は `Int` に絞り込まれます (narrowing)。だから `v` をそのまま `Int` として返せるのです。

網羅性検査 (第5章) は `null` もケースとして数えます。 `missingnull.lune` :

```
module bad

# Bool? の match は true / false だけでは足りない。
def toInt(b: Bool?): Int =
  match b:
    | true -> 1
    | false -> 0
```

```
$ lune --check missingnull.lune
```

```
error[TYP0007]: 網羅的でない match: null のケースがありません
--> missingnull.lune:5:5
|
5 |     match b:
|     ^^^^^ パターン null がカバーされていない
  = hint: null のケースを追加するか、ワイルドカードケース `| _ -> ...` を追加してください
  = help: 詳しくは `lune explain TYP0007 --lang ja` を実行してください
```

「null チェックを忘れる」というあの事故が、コンパイル時の `null` のケースがありませんに変わりました。

一つ注意。名前パターンだけで受けると、網羅性は満たせますが**絞り込まれません**。

```
lune> def bad(value: Int?): Int =
...     match value:
...         | v -> v
...
error[TYP0003]: 分岐: Int が必要ですが、Nullable[Int] が見つかりました
--> <repl:17>:3:16
|
3 |         | v -> v
|           ^ この式の型は Nullable[Int]
= help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

`v` が `Int` になるのは、`null` の腕がそれより前にあるときだけです。

7.3 ?? — なければこれ

「null ならデフォルト値」は頻出なので、専用の演算子があります。**null 合体演算子** `??` です。

```
lune> absent ?? 7
7 : Int
lune> present ?? 7
42 : Int
lune> present ?? crash()
42 : Int
```

左が null でなければ左を、null なら右を返します。結果の型は `Int?` ではなく `Int` — 合体した時点で null の可能性は消えるからです。3つ目の例に注目してください。左に値があるとき、**右側は評価すらされません**。第4章の遅延評価が、ここでも当たり前のように効いています。

7.4 ?. — 安全にたどる

「`user` が null でなければ `user.name` を読みたい」 — これも専用構文があります。**安全ナビゲーション** `?.` です。 `nameof.lune` :

```

module nameOf

record User:
  name: String
  age: Int

# ?. は受け手が null なら null に短絡し、そうでなければフィールドを読む。
def nameOf(user: User?): String? =
  user?.name

let someName = nameOf(User(name = "Ada", age = 36))

let noName = nameOf(null)

let fallback = nameOf(null) ?? "(nobody)"

```

```

$ lune --eval someName nameof.lune
"Ada"
$ lune --eval noName nameof.lune
null
$ lune --eval fallback nameof.lune
"(nobody)"

```

受け手が null なら結果も null、そうでなければフィールドの値。結果は常に `String?` になるので、最後は `??` や `match` で受け止めます。?. で null を運び、?? で着地させる — この2つはセットで使う道具です。

7.5 if で絞り込む

`match` を書くまでもない場面では、`if` の条件でも絞り込めます。

```

lune> def orOne(x: Int?): Int =
...   if x != null then x else 1
...
ok
lune> orOne(absent)
1 : Int

```

`x != null` が真の分岐では、`x` は `Int` として使えます (`x == null` なら偽の分岐で絞り込まれます)。この絞り込みが効くのは `x != null` / `x == null` という単純な形だけで、`&&` で繋いだ複合条件や `elif` には及びません。凝った条件になったら `match` に切り替えてください。

7.6 null を返す関数を書く

ここまでは null を受け取る側でした。返す側は、素直に書けば通ります。

```
lune> def maybeDiv(x: Int, y: Int): Double? =  
...     if y == 0 then null else x / y  
...  
ok  
lune> maybeDiv(7, 2)  
3.5 : Nullable[Double]  
lune> maybeDiv(7, 0)  
null : Nullable[Double]
```

null を返す腕と Double を返す腕が、なぜ揉めずに合流できるのか。第5章 §5.6 の先取りボックスで予告した仕組みがここで働いています — 戻り値注釈 Double? が期待型として if の両分岐へ配られ、null の腕も x / y の腕もそこで Double? に揃うのです。match の腕でも同じことが起きます。ファイル版の maybeDiv.lune:

```
module maybeDiv  
  
# 期待型 Double? が if の両分岐へ配られ、null の腕と x / y の腕がそこで合流する。  
def maybeDiv(x: Int, y: Int): Double? =  
    if y == 0 then null else x / y  
  
let some = maybeDiv(7, 2)  
  
let none = maybeDiv(7, 0)  
  
let fallback = maybeDiv(7, 0) ?? 0.0
```

```
$ lune --eval some maybeDiv.lune  
3.5  
$ lune --eval none maybeDiv.lune  
null  
$ lune --eval fallback maybeDiv.lune  
0.0
```

maybeDiv(7, 0) で 0 除算エラーが出ないのは、if が選ばなかった腕を評価しないからです（第4章）。「失敗しうる計算」を else の腕に置いたまま、null で受け流す — T? を返す関数の基本形です。

こう書いても動く — 分岐の前に、注釈付き `let` で部品を作っておく書き方もあります。

```
# 部品を注釈付き let で先に作ってから分岐しても動く。
# quotient はサンクなので、y == 0 の側では割り算は一度も走らない。
def maybeDivLet(x: Int, y: Int): Double? =
  let quotient: Double? = x / y
  if y == 0 then null else quotient
```

`maybeDivLet(7, 0)` もちゃんと `null` を返します。 `let quotient: Double? = x / y` を「通って」いるのに 0 除算エラーが出ないのは、`quotient` がサンクのまま一度も `force` されないから — こちらも第4章の遅延評価です。どちらを選んでも安全なので、普段は分岐を先に書く素直な形で十分です。

「なくてもよい」フィールドの読み出し (`?.`)、デフォルトへの着地 (`??`)、そしてこの「なければ `null` を返す関数」。3点セットで、`null` は型に守られた普通の道具になります。

7.7 Option[T] と T? の使い分け

似た道具が2つある理由と、選び方です。

	T?	Option[T]
「なし」の表現	<code>null</code>	<code>None</code>
専用構文	<code>??</code> 、 <code>?.</code> 、 <code>if/match</code> の絞り込み	なし (<code>match</code> と関数で扱う)
相性	フィールド・引数・戻り値の「なくてもよい」	プレリユードのリスト API (<code>head / tail</code> が返す)、ジェネリック関数
入れ子の「なし」	表せない (<code>null</code> は1段)	<code>Some(None)</code> と <code>None</code> を区別できる

実用の目安はこうです。データの形として「ないかもしれない」を書くなら `T?` (レコードのフィールド、関数の引数と戻り値)。リスト処理や高階関数のパイプラインの中では `Option` (プレリユードがそちらを話すからです)。境界では `match` で詰め替えれば行き来できます。

壊してみよう — `orZero` の2つの腕を入れ替えると（`| v -> v` を先に）どうなるでしょう。予想してから試してください。

```
lune> def swapped(value: Int?): Int =
...     match value:
...         | v -> v
...         | null -> 0
...
error[TYP0003]: 分岐: Int が必要ですが、Nullable[Int] が見つかりました
--> <repl:19>:3:16
|
3 |         | v -> v
|           ^ この式の型は Nullable[Int]
= help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

到達不能（`TYP0009`）を予想した人が多いはずですが、その手前で捕まりました。先頭の `v` はまだ `null` が被覆されていないので `Int?` のままです（7.2節の注意）。そこへ戻り値注釈の `Int` が期待型として配られ（7.6節）、最初の腕がその期待に応えられないと名指しされます。絞り込みは「`null` の腕が先」のときだけ働く — 腕の順番は型にまで影響します。

まとめ

概念	一言で
<code>T?</code>	<code>T</code> または <code>null</code> 。表示名は <code>Nullable[T]</code>
型の守り	<code>T?</code> を <code>T</code> として使うと <code>TYP0003</code> 。 <code>null</code> ケース漏れは <code>TYP0007</code>
<code>match</code>	<code> null -></code> を被覆すると、以降の名前パターンは <code>T</code> に絞り込み
<code>??</code>	<code>null</code> ならデフォルト。右辺は必要になるまで評価されない
<code>?.</code>	<code>null</code> なら <code>null</code> に短絡してフィールドを読む。 <code>??</code> とセットで
<code>if x != null</code>	単純形のみ絞り込み。複雑になったら <code>match</code>
<code>null</code> を返す	期待型が分岐へ配られるので、素直に <code>if / match</code> で書ける（第5章 §5.6 と同じ仕組み）
vs <code>Option</code>	データの形は <code>T?</code> 、リスト・高階関数の世界は <code>Option</code>

演習問題

演習 7-1 (★) 結果（値と型）を予想してから確かめてください。 `absent: Int? = null`、`present: Int? = 42` とします。

```
absent ?? 7
present ?? 7
absent == null
present ?? crash()
```

解答

`7 : Int`、`42 : Int`、`true : Bool`、`42 : Int`。最後の式が爆発しないのは `??` が短絡するからです。型が `Int?` ではなく `Int` になっていることも確認してください。

演習 7-2 (★★) 7.5節の `orOne` を、`if` ではなく `match` で書き直してください。

解答

```
lune> def orOne(x: Int?): Int =
...     match x:
...         | null -> 1
...         | v -> v
...
ok
lune> orOne(null)
1 : Int
lune> orOne(41)
41 : Int
```

`orZero` と同じ骨組みです。`if x != null` 版と `match` 版はどちらも正解で、分岐が2択なら `if`、増えるなら `match`、が目安です。

演習 7-3 (★★) `nameOf` にならって `ageOf` を書き、`ageOf(null) ?? 0` が `0` になることを確かめてください。

解答

```
lune> def ageOf(user: User?): Int? =  
...     user?.age  
...  
ok  
lune> ageOf(User(name = "Ada", age = 36))  
36 : Nullable[Int]  
lune> ageOf(null) ?? 0  
0 : Int
```

「?. で運んで ?? で着地」の型 `User? → Int? → Int` の流れを、`:type` でも追ってみてください。

演習 7-4 (★★★) 整数リストの平均を返す `average(xs: List[Int]): Double?` を書いてください。空リストの平均は `null` とします。0 除算を起こさずに書けるでしょうか。

解答

```
module answers  
  
# 演習 7-4: 空リストの平均は「ない」。先に分岐すれば 0 除算には触れない。  
def average(xs: List[Int]): Double? =  
    if isEmpty(xs) then null else fold(xs, 0, fn a x -> a + x) /  
    length(xs)  
  
let some = average([2, 3, 4])  
  
let none = average([])  
  
let safe = average([]) ?? 0.0
```

```
$ lune --eval some ex7-4.lune  
3.0  
$ lune --eval none ex7-4.lune  
null  
$ lune --eval safe ex7-4.lune  
0.0
```

空かどうかを**先に**分岐してしまえば、割り算は空でないリストに対してしか評価されません。`null` の腕と計算の腕が `Double?` へ合流するのは 7.6 節のとおりです。合計と割り算を注釈付き `let` で先に定義してから分岐する書き方でも、サンのまま捨てられるので 0 除算にはなりません (7.6 節の「こう書いても動く」)。

演習 7-5 (★・逆転問題) 「`null` かもしれない値を、確かめずに使った」ことを型検査に叱られる最小のコードを書いてください。

解答

```
let present: Int? = 42

let oops = present + 1
```

とすると `error[TYP0003]: +: 数値型が必要ですが、Nullable[Int] が見つかりました` が出ます。関数に渡す形（本文 7.1 節の `wantsInt(present)`）なら `Int` が必要ですが、`Nullable[Int]` が見つかりました。他言語なら実行時の `NullPointerException` になっていた事故が、コンパイル時のこの1行に置き換わっている — それが null 安全の価値です。

より正確には — `T?` の型付け・narrowing の正確な規則は

`documents/LANGUAGE_SPEC.md` §11（match の null パターン）と §9.2（if の絞り込み）、`null` を含む網羅性は `documents/MATCH_EXHAUSTIVENESS_SPEC.md`、期待型が分岐へ配られる規則は `documents/LOCAL_TYPE_INFERENCE_SPEC.md` §5.3。実例は `samples/nullable.lune` にもあります。この章のコード例は `books/examples/ch07/` にあり、すべて実際の CLI で検証されています。

第8章 リストとストリーム

第4章の遅延評価が「データ構造の設計」に化ける章です。前半は第1章から使ってきたリストをきちんと学び直します。後半はその同じリストが**無限**になります — 新しい型は登場しません。Lune のリストは、生まれつき無限になれるのです。

8.1 リストの正体 — Cons と Nil

`[1, 2, 3]` は糖衣構文です。リストの正体は、プレリユードに定義されたただの ADT — 第5章の道具で書ける、2つのコンストラクタです。

```
lune> :type Cons
Cons : [T] T -> List[T] -> List[T]
lune> :type Nil
Nil : List[T]
lune> Cons(1, Cons(2, Nil))
(1 2) : List[Int]
```

`Nil` が空リスト、`Cons(先頭, 残り)` が「先頭の値と、残りのリスト」。`[1, 2]` は `Cons(1, Cons(2, Nil))` のことです（`Nil` はプレリユードの `None` と同じく、値として登録されています）。

ここで第4章を思い出してください。コンストラクタのフィールドはデフォルトで遅延でした。つまり `Cons` の「残り」は、必要になるまで計算されないサンスクです。この一点が、この章の後半のすべてを支えます。

8.2 基本の道具 — 開ける・数える・変換する

まず開ける道具。`head`（先頭）と `tail`（残り）は `Option` を返します。

```
lune> head([1, 2, 3])
Some(1) : Option[Int]
lune> head([])
None : Option[T]
lune> tail([1, 2, 3])
Some((2 3)) : Option[List[Int]]
lune> isEmpty([])
true : Bool
```

空リストに先頭はありません。「ないかもしれない」答えを `Option` で返すのは第5章で学んだ設計そのものです。とはいえ、リストを開けるいちばん良い方法は `Option` を剥がすことではなく、**`match` で直接分解**することです。8.6節の実例で使います。

変換の道具は第1章以来おなじみの `map` / `filter` / `fold`、それに `take` / `drop` / `range` / `length` です。復習を兼ねて型だけ整理しておきます — すべて「リストが先、道具が後」の引数順です。

関数	型 (要約)	一言
<code>map(xs, f)</code>	<code>List[T] -> (T -> U) -> List[U]</code>	全要素に <code>f</code>
<code>filter(xs, p)</code>	<code>List[T] -> (T -> Bool) -> List[T]</code>	<code>p</code> が真の要素だけ
<code>fold(xs, init, f)</code>	<code>List[T] -> U -> ((U, T) -> U) -> U</code>	畳み込み
<code>take(xs, n)</code> / <code>drop(xs, n)</code>		先頭 <code>n</code> 個を取る / 捨てる
<code>head</code> / <code>tail</code>		<code>Option</code> で返す
<code>length</code> / <code>isEmpty</code> / <code>range</code>		長さ / 空か / 整数列

8.3 tail は遅延している — リストは無限になれる

本題です。 `Cons` の `tail` はサックでした。ということは、「残りのリスト」の計算が**永遠に終わらなくても構わない**はずです。プレリユードには、まさにそういうリストを作る関数があります。

```
naturalsFrom(n)    # [n, n+1, n+2, ...]
iterate(f, x)      # [x, f(x), f(f(x)), ...]
repeat(x)          # [x, x, x, ...]
cycle(xs)           # xs を無限に繰り返す
```

`:thunks` で、無限リストの中身を覗いてみましょう（`:thunks` は評価を起こさないの
で、無限リストでも安全です）。

```
lune> let nat = naturalsFrom(1)
ok
lune> :thunks nat
nat : unevaluated
lune> head(nat)
Some(1) : Option[Int]
lune> :thunks nat
nat : evaluated = Cons(1, <thunk>)
lune> take(nat, 5)
(1 2 3 4 5) : List[Int]
lune> :thunks nat
nat : evaluated = Cons(1, Cons(2, Cons(3, Cons(...))))
```

`Cons(1, <thunk>)` — 先頭の 1 だけが計算され、残り全部は「まだ約束のまま」。 `take` で 5 個要求すると、その分だけ評価が進みました。**無限リストとは、必要な分だけ実体化する数列**です。どこまで実体化したかが `:thunks` でそのまま見える — 第4章の観察道具は、ここで最大の見せ場を迎えます。

`take` と組み合わせれば、無限リストは普通の値として扱えます。 `infinite.lune`:

```
module infinite

# List の tail は遅延している。だからリストは無限になれる。
let firstFive = take(naturalsFrom(1), 5)

let powersOfTwo = take(iterate(fn x: Int -> x * 2, 1), 6)

let threeSevens = take(repeat(7), 3)

let pattern = take(cycle([1, 2, 3]), 7)
```

```
$ lune --eval powersOfTwo infinite.lune
(1 2 4 8 16 32)
$ lune --eval pattern infinite.lune
(1 2 3 1 2 3 1)
```

8.4 遅延コンビネータ — 無限のまま加工する

`map` や `filter` も tail の遅延を保つので、無限リストを無限リストのまま加工できます。仕上げに `take` で必要な分だけ受け取ります。

```
lune> take(map(naturalsFrom(1), fn n: Int -> n * n), 5)
(1 4 9 16 25) : List[Int]
lune> takeWhile(naturalsFrom(1), fn x -> x < 4)
(1 2 3) : List[Int]
lune> take(dropWhile(naturalsFrom(1), fn x -> x < 10), 3)
(10 11 12) : List[Int]
lune> take(zip(naturalsFrom(1), cycle(["a", "b"])) , 4)
((1, "a") (2, "b") (3, "a") (4, "b")) : List[Tuple[Int, String]]
lune> take(zipWith(naturalsFrom(1), naturalsFrom(10), fn a: Int b: Int ->
a + b), 3)
(11 13 15) : List[Int]
```

- `takeWhile` / `dropWhile` — 条件が続く間だけ取る / 捨てる
- `zip(a, b)` — 2本のリストをタプルの列に束ねる（短い方で止まる）
- `zipWith(a, b, f)` — 束ねながら `f` で合成する

無限どうしを `zip` しても平気です。結果も無限リストになり、使う分しか計算されないからです。

8.5 止まらない操作 — 消費し切る関数たち

ただし、道具には「リストを最後まで消費するもの」があります。 `fold` と `length` です。無限リストに使うと、最後は永遠に来ないので**止まりません**。

```
lune> length(naturalsFrom(1))
```

これを打つとプロンプトは帰ってきません（Ctrl-C で中断してください）。エラーにならないのがポイントです — 1つずつ数え続けること自体は正しい計算で、終わらないだけなのです。無限リストをそのまま表示しようとしたときも同じなので、印字の前に `take` を挟んでください。

見分け方は単純です。**答えを出すのに全要素が要るか？**

`take` / `map` / `filter` / `takeWhile` / `zip` は要らない（遅延を保つ）。`fold` / `length` は要る（消費し切る）。`drop(xs, n)` は捨てる `n` 個分の spine しか force しないので無限リストにも使えます（`take(drop(naturalsFrom(1), 5), 3)` は `(6 7 8)`）。ただし返るのは依然として無限リストなので、表示の前に `take` を置いてください。`filter` にも罫が一つあって、条件を満たす要素が二度と現れないと、次の1個を探して走り続けます。

この見分け方は**有限リストにも効きます**。`range` も遅延なので（4.6節）、`range(1, 1000000000)` は無限ではないものの、`length` や `fold` に渡せば「止まらない」のと大差ない待ち時間になります。`take` / `takeWhile` を前に置くかどうかは、リストが無限かどうかではなく、**答えに全要素が要るかどうか**で決めてください。

8.6 実例集 — 無限リストで考える

フィボナッチ数列。「状態 (a, b) を $(b, a+b)$ に進め続ける」と読めます。 `iterate` の出番です。 `fib.lune` :

```
module fib

# 状態 (a, b) を一歩進める: (0,1) -> (1,1) -> (1,2) -> (2,3) -> ...
def step(p: Tuple[Int, Int]): Tuple[Int, Int] =
  match p:
    | (a, b) -> (b, a + b)

def fst(p: Tuple[Int, Int]): Int =
  let (a, _) = p
  a

# フィボナッチ数列そのもの (無限リスト)。
let fibs = map(iterate(step, (0, 1)), fst)

let first10 = take(fibs, 10)
```

```
$ lune --eval first10 fib.lune
(0 1 1 2 3 5 8 13 21 34)
```

`fibs` は「10個のフィボナッチ数」ではなく、**フィボナッチ数列そのもの**に付けた名前です。何個使うかは、使う側があとで決める。生成と消費の分離 — 無限リストの設計上の本当の御利益はこれです。

エラトステネスの篩。素数の無限リストを作ります。 `primes.lune` :

```
module primes

# エラトステネスの篩。先頭 p を素数として採り、
# p で割り切れるものを残りから漉して、続きは必要になったら計算する。
def sieve(xs: List[Int]): List[Int] =
  match xs:
    | Cons(p, rest) -> Cons(p, sieve(filter(rest, fn n: Int -> n % p
      != 0)))
    | Nil -> Nil

let primes = sieve(naturalsFrom(2))

let first10 = take(primes, 10)
```

```
$ lune --eval first10 primes.lune
(2 3 5 7 11 13 17 19 23 29)
```

3行に注目ポイントが詰まっています。

- リストを `match` で直接 `Cons(p, rest)` に分解しています（`head / tail` より素直です）。
- 返す `Cons(p, ...)` の第2引数 — 再帰呼び出し `sieve(...)` — は**遅延フィールドに入る**ので、無限の再帰なのに止まりません。次の素数は、誰かが要求したときに初めて漉されます。
- `| Nil -> Nil` の右辺の裸の `Nil` は、単体では型引数が未確定でした（第5章 5.6 節の `None : Option[T]` と同じ）。ここでは返り値注釈 `List[Int]` が期待型として腕へ配られ、`T = Int` を確定させています。

壊してみよう — `take` の引数順を間違えると、型がすぐに教えてくれます。

```
lune> take(5, naturalsFrom(1))
error[TYP0003]: List[T] が必要ですが、Int が見つかりました
  = help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

リストが先、個数が後。「リスト系の関数はリストが第1引数」という規約（8.2節の表）を思い出してください。

まとめ

概念	一言で
<code>Cons</code> / <code>Nil</code>	リストの正体。[1, 2] は <code>Cons(1, Cons(2, Nil))</code>
tail の遅延	「残り」はサンク。だからリストは無限になれる
<code>head</code> / <code>tail</code>	<code>Option</code> を返す。分解するなら <code>match</code> で <code>Cons(x, rest)</code>
<code>naturalsFrom</code> / <code>iterate</code> / <code>repeat</code> / <code>cycle</code>	無限リストの作り方4種
遅延を保つ道具	<code>take</code> / <code>map</code> / <code>filter</code> / <code>takeWhile</code> / <code>dropWhile</code> / <code>zip</code> / <code>zipWith</code>
消費し切る道具	<code>fold</code> / <code>length</code> — 無限リストには使わない（ <code>drop</code> は捨てる分だけ <code>force</code> するので安全）
観察	<code>:thunks</code> は評価せずに「どこまで実体化したか」を見せる

演習問題

演習 8-1 (★) 結果を予想してから確かめてください。

```
take(cycle([1, 2]), 5)
takeWhile(naturalsFrom(1), fn x -> x < 4)
take(zip(naturalsFrom(1), cycle(["a", "b"]))), 4)
```

解答

(1 2 1 2 1)、(1 2 3)、((1, "a") (2, "b") (3, "a") (4, "b"))。3つ目は「無限の番号列」と「無限の交互パターン」の `zip` — どちらも終わらないリストですが、4個しか要求していないので4個分しか計算されません。

演習 8-2 (★★) 平方数 (1, 4, 9, 16, ...) の無限リストを作り、そのうち偶数のものだけを先頭から5個取り出してください。

解答

```
module answers
```

```
# 演習 8-2: 平方数の無限リストから、偶数のものだけを 5 個。
```

```
let squares = map(naturalsFrom(1), fn n: Int -> n * n)
```

```
let evenSquares = take(filter(squares, fn s: Int -> s % 2 == 0), 5)
```

```
$ lune --eval evenSquares ex8-2.lune  
(4 16 36 64 100)
```

`map` → `filter` → `take` と、無限のまま2段加工してから収穫しています。

演習 8-3 (★★) センサーの読み `[1, 4, 7, 10]` について、隣どうしの平均（移動平均）`(2.5 5.5 8.5)` を計算してください。ヒント: リストを1つずらして自分自身と `zipWith`。

解答

```
module answers
```

```
# 演習 8-3: 隣どうしの平均（移動平均）。リストを 1 つずらして自分と zip する。
```

```
let readings = [1, 4, 7, 10]
```

```
let smoothed = zipWith(readings, drop(readings, 1), fn a: Int b: Int ->  
  (a + b) / 2)
```

```
$ lune --eval smoothed ex8-3.lune  
(2.5 5.5 8.5)
```

`zipWith` が短い方（`drop` した側）で止まるので、端の処理を書く必要がありません。

演習 8-4 (★★★) リュカ数列 (2, 1, 3, 4, 7, ... — フィボナッチと同じ漸化式で、始まりが (2, 1)) の無限リストを作り、先頭10個を取り出してください。

解答

```
module answers

# 演習 8-4: リュカ数列。フィボナッチと同じ漸化式で、種だけ (2, 1) に変える。
def step(p: Tuple[Int, Int]): Tuple[Int, Int] =
  match p:
    | (a, b) -> (b, a + b)

def fst(p: Tuple[Int, Int]): Int =
  let (a, _) = p
  a

let lucas = map(iterate(step, (2, 1)), fst)

let first10 = take(lucas, 10)
```

```
$ lune --eval first10 ex8-4.lune
(2 1 3 4 7 11 18 29 47 76)
```

`fib.lune` との差は `iterate` の種だけ。「漸化式 = `step` 関数、数列 = `iterate`」という対応が掴めれば、この形はどんな漸化式にも使えます。

演習 8-5 (★・逆転問題) 「止まらない式」を1つ書いてください。実行はしなくて構いません (するなら Ctrl-C の準備を)。なぜ止まらないのか、8.5節の言葉で説明してください。

解答

例: `length(naturalsFrom(1))`、`fold(repeat(1), 0, fn a x -> a + x)`、あるいは `takeWhile(naturalsFrom(1), fn x -> x > 0)` (条件が永遠に真)。どれも「答えを出すために全要素が要る」計算です。少し意地悪な例は `head(filter(naturalsFrom(1), fn x -> x < 0))` — `filter` 自体は遅延を保ちますが、最初の1個が永遠に見つかりません。

より正確には — `Cons` の遅延フィールドと各関数の遅延特性 (どこまで `force` するか) は `documents/STANDARD_LIBRARY_SPEC.md §6` に関数ごとに明記されています。`take(list, 0)` が `list` を評価しないことまで仕様です。この章のコード例は `books/examples/ch08/` にあり、すべて実際の CLI で検証されています。

第9章 命令的に書く — var ・ while ・ for ・ IO

気づいていたでしょうか。ここまでの8つの章で、あなたは**一度も変数を書き換えていません**。温度換算表も、無限の素数列も、じゃんけんの判定も、すべて「値に名前を付けて、式で組み合わせる」だけで書けました。

それでも、カウンタを回したい場面、途中経過を書き換えたい場面はあります。Lune の答えは現実的です — **命令的な書き方も、小さく使うぶんには良い道具**。この章では可変束縛 `var`、ループ `while` と `for`、出力の置き場所 `IO` を学びます。合言葉は「**命令はブロックに閉じ込め、外へは値を返す**」です。

9.1 ブロック慣用句 — 命令の島

まず Lune 流の「命令的コードの置き方」を見てください。 `counter.lune` :

```
module counter

# var と while はブロックの中で使い、最後の式で結果を返す。
let answer =
  var i = 0
  var total = 0
  while i < 5:
    total = total + i
    i = i + 1
  total
```

```
$ lune --eval answer counter.lune
10
```

ブロックの中では変数を書き換わり、ループが回っています。しかし**外から見れば、`answer` はただの `Int` です**。第2章のブロック `let` と同じ形で、最後の式 `total` がブロック全体の値になります。命令的な過程は島の中に閉じ、島の外は今までどおり式の世界 — この形を保つ限り、命令的なコードはプログラムの残りを汚しません。

ついでに第4章の住人として一言: この `let answer = ...` も遅延です。誰かが `answer` を使うまで、ループは一周も回りません。

9.2 var — 正格な可変束縛

書き換えられる束縛は `var` で作ります。

```
var count = 0
count = count + 1
```

`let` との違いは2つあります。**可変**であること、そして**正格**（束縛した瞬間に右辺を評価する）であることです。`tick()`（第4章）で確かめられます。

```
lune> var t = tick()
ok
lune> tickCount()
1 : Int
```

束縛しただけでカウンタが進みました — `let` なら0のままだったはずです。書き換わる変数の値が「いつ評価されるか分からない」のでは推論できないので、`var` は遅延しない、と覚えてください。

let		var
再代入	不可（不変）	可
評価	遅延（使うまで計算しない）	正格（束縛時に計算する）

代入できるのは名前だけです。`i = i + 1` と書くほかに、複合代入 `+=` `--` `*=` `/=` `//=` `%=` も使えます（`i += 1` は `i = i + 1` とまったく同じ意味です）。ひとつ落とし穴があって、`/` は常に真の除算なので `Int` の変数への `x /= 2` は型エラーになります（`Double` になってしまうため）。整数を整数のまま割りたいときは、床除算の `x //= 2` を使ってください（`//` は2.2節）。

「不可」は、心構えではなく検査です。`let` に代入すると `TYP0013` が出ます。

```
lune> let a = 1
ok
lune> a = 2
error[TYP0013]: `a` には代入できません: `let` の束縛は不変です
--> <repl:2>:1:1
  |
1 | a = 2
  | ^ この束縛には代入できない
    = hint: 書き換えるなら `var a = ...` で宣言してください
    = help: 詳しくは `lune explain TYP0013 --lang ja` を実行してください
lune> a
1 : Int
```

ファイルでも同じです。 `letassign.lune` を `lune --check` にかけると、代入している場所を指して同じ `TYP0013` が出ます。

不変なのは `let` だけではありません。**関数の仮引数**も、**for の変数**も、パターンで束縛した名前も、代入できません。書き換えてよい束縛は `var` だけです。検査が一番近い束縛を見るので、外側に `var x` があっても、ブロックの中の `let x` は不変のままです。

9.3 while — 最小のループ

`while` 条件: は、条件が `Bool` である間ブロックを繰り返します。9.1節の `counter.lune` がすべてです — 条件は**毎周**評価され、`while` 式全体の値は常に `Unit`。つまり `while` は値を作る道具ではなく、`var` を書き換えるための道具です。

`break` と `continue` はありません。途中で抜けたければ、その条件をループ条件に織り込みます (`while i < 5 && not(done):` のように)。それが煩わしくなってきたら、それはたいてい**再帰**か `fold` で書くべき合図です (第3・8章)。

9.4 for — リストを歩く

リストの全要素に何かしたいだけなら、`for` が簡潔です。 `fortotal.lune`:

```
module fortotal

let pairs = [(1, 10), (2, 20)]

# for のパターンには、let と同じく反駁不能なものが書ける。
let total =
  var result = 0
  for (left, right) in pairs:
    result = result + left + right
  result
```

```
$ lune --eval total fortotal.lune
33
```

`for` パターン `in` リスト: の形で、パターンには `let` と同じ規則 (第5章) が適用されます — タプルのような**反駁不能**なパターンは分解でき、失敗しうるパターンは弾かれます。

```
lune> for Some(x) in [Some(1), Some(2)]:
...     println(x)
...
error[TYP0008]: for の束縛に反駁可能パターンは使えません: Some(x)
--> <repl:1>:1:24
|
1 | for Some(x) in [Some(1), Some(2)]:
|           ^^^^^ このパターンはマッチに失敗し得る
= hint: このパターンは None をカバーしていません
= hint: `match` を使って Option[Int] の全ケースを場合分けしてください
= help: 詳しくは `lune explain TYP0008 --lang ja` を実行してください
```

走査できるのは `List[T]` だけです。 `badfor.lune` のようにリスト以外を渡すと、専用の診断が出ます。

```
error[TYP0006]: for の対象は List でなければなりませんが、Int でした
--> badfor.lune:5:14
|
5 |     for x in 42:
|           ^^ 走査対象は List[T] でなければならない
= help: 詳しくは `lune explain TYP0006 --lang ja` を実行してください
```

`for` もまた `Unit` です。そして背骨を1歩ずつ `force` しながら進むので、**無限リストに `for` すると帰ってきません** — 第8章の「消費し切る道具」の仲間だと思ってください。

9.5 IO — 出力と遅延の付き合い方

`println` はこれまで何度も使ってきましたが、遅延の世界には出力ならではの罫があります。見てください。

```
lune> let p = println("hi")
ok
lune> p
hi
(): Unit
```

`let` の行では**何も出力されません**。 `println("hi")` はサンクに包まれ、 `p` を使った瞬間に初めて実行されたのです。つまり遅延の世界では、「出力がいつ起きるか」は「値がいつ要るか」で決まる — 画面に順番どおり文字を出したいプログラムにとって、これは困った性質です。

だから出力は **IO: ブロックに集めます**。 `io.lune` :

```

module io

# 出力は IO ブロックに集め、上から順に実行させる。
def report(): Unit =
  IO:
    println("one")
    println("two")

let run = report()

```

```

$ lune --eval run io.lune
one
two
()

```

IO: ブロックの中の文は、ブロックが実行されるとき**上から順に**走ります。「順序が意味を持つ副作用はここにありますが」という宣言でもあり、読み手への目印でもあります（v0.1 の **IO:** は型システム上の厳密なエフェクト管理ではなく、この「置き場所」の規約です — 将来仕様では強化が予定されています）。

作法としてはこう覚えてください。**計算は純粋な関数で、出力は IO の中で**。判定ロジックを `String` を返す関数に切り出し、**IO:** の `for` で流し込む — この分離の練習が演習 9-2 (FizzBuzz) です。

9.6 raise / throw — 最後の手段

実行時エラーを自分で起こす式もあります。

```

lune> let bad = raise "failed"
ok
lune> bad
error[RUN0006]: failed
  = help: 詳しくは `lune explain RUN0006 --lang ja` を実行してください

```

raise 式（同義の **throw** もあります）は評価されると実行時エラーを送出します。例によって **let** は遅延なので、束縛しただけでは何も起きず、使った瞬間に爆発します — `crash()` の自作版だと思えばだいたい正しいです。

そして重要な注意: **try / catch はありません**。つまり **raise** したエラーをプログラムの中で受け止める方法はないのです。回復可能な失敗は、第5章の単相の結果型や `Result`、第7章の `T?` — **失敗を値として返す** 道具で表してください。**raise** の出番は「ここに来たらプログラムを止めるのが正しい」という、本当に例外的な場面だけです。

まとめ

概念	一言で
ブロック慣用句	命令的コードは <code>let 名前 =</code> のブロックに閉じ、最後の式で値を返す
<code>var</code>	可変・ 正格 。代入は名前への <code>=</code> のみ
<code>while</code> 条件:	条件は毎周評価。式全体は <code>Unit</code> 。break/continue はない
<code>for</code> パターン in リスト:	<code>List[T]</code> 専用 (TYP0006)。パターンは反駁不能 (TYP0008)
出力の順序	遅延に流されないよう <code>IO</code> : ブロックに集めて上から実行
分離の作法	計算は純粋な関数、出力は IO の中
<code>raise / throw</code>	受け止める手段はない。回復可能な失敗は値で返す

演習問題

演習 9-1 (★) `while` で 1 から 10 までの積 (10!) を計算してください。

解答

```
module answers

# 演習 9-1: while で 1 から 10 までの積 (10!)。
let product =
  var i = 1
  var acc = 1
  while i <= 10:
    acc = acc * i
    i = i + 1
  acc
```

```
$ lune --eval product ex9-1.lune
3628800
```

演習 9-2 (★★) FizzBuzz を書いてください。1~15 について、3 の倍数なら `Fizz`、5 の倍数なら `Buzz`、両方なら `FizzBuzz`、それ以外は数を1行ずつ出力します。判定と出力を分けること。

解答

```
module answers

# 演習 9-2: FizzBuzz。判定は純粋な関数に、出力は IO の for に分ける。
def fizzbuzz(n: Int): String =
  if n % 15 == 0:
    "FizzBuzz"
  elif n % 3 == 0:
    "Fizz"
  elif n % 5 == 0:
    "Buzz"
  else:
    show(n)

def main(): Unit =
  IO:
    for i in range(1, 16):
      println(fizzbuzz(i))

let run = main()
```

`fizzbuzz` は純粋な関数なので、REPL で単体テストできます（`fizzbuzz(15)` → `"FizzBuzz"`）。出力の都合と判定のロジックが混ざらない — 9.5節の分離の作法です。

演習 9-3 (★★) `let t = tick()` と `var v = tick()` を1つずつ定義して、`tickCount()` と併せて `let` と `var` の評価タイミングの違いを説明してください。

解答

定義直後の `tickCount()` は1 — 進んだのは `var` の分だけです。`let t` のサンクはまだ眠っていて、`t` を初めて使った瞬間に2になります。第4章の `strict let` と同じタイミング、と整理できます（`var` = 可変 + 正格、`strict let` = 不変 + 正格）。

演習 9-4 (★★★) 9.1節の `counter.lune` (0~4 の和) を、`var` もループも使わずに書き直してください。そのうえで、どちらが読みやすいか自分の言葉で論じてください。

解答

```
lune> fold(range(0, 5), 0, fn a x -> a + x)
10 : Int
```

1行です。「積み上げる計算」は `fold` の独壇場で、ループ版の4行（カウンタ2本と更新2式）が背負っていた「`i` と `total` を取り違えない責任」が消えます。一方、途中で条件により打ち切るような制御は `while` が素直なこともあります。目安: データを一巡して集計するなら `fold`、状態機械を回すなら `while`。

演習 9-5 (★・逆転問題) `TYP0006` と `TYP0008` を、それぞれ `for` を使った最小のコードで出してください。

解答

```
for x in 42: println(x) (TYP0006 — 42 は List ではない) と for Some(x) in  
[Some(1)]: println(x) (TYP0008 — Some(x) は失敗しうるパターン)。どちらも本  
文 9.4 節に実物があります。TYP0008 の hint が第5章の let のときと同じ言い回しで  
あることも見ておいてください — for の束縛は let の親戚です。
```

より正確には — `var` と代入は `documents/LANGUAGE_SPEC.md §7.3`、`while` は `documents/WHILE_LOOP_SPEC.md`、`for` は `documents/FOR_LOOP_SPEC.md` (`spine` の `force` 規則も)、`raise / throw` と `IO:` は `documents/LANGUAGE_SPEC.md §9.6–9.7`。この章のコード例は `books/examples/ch09/` にあり、すべて実際の CLI で検証されています。

第10章 モジュール

プログラムが育つと、1つのファイルに収まらなくなります。この章では Lune のプログラムを複数ファイルに分ける方法 — **モジュール**を学びます。

v0.1 のモジュール機構は意図的に小さいものです。パッケージ管理も、公開/非公開の制御も、名前空間アクセスありません。あるのは「ファイルを分けて、読み込む」だけ。それでも、この最小の道具立てには覚えるべき規則が4つあり、間違えたときにはちゃんと専用の診断（MOD 族、第11章）が出ます。

10.1 module 宣言 — ファイルの名札

第1章から書き写してきた1行目を、ようやく主役にします。

```
module geometry
```

Lune のソースファイルは、自分がどのモジュールなのかを宣言します。そして**宣言はファイルの位置と一致していなければなりません** — `geometry.lune` なら `module geometry`、`util/text.lune` なら `module util.text` です。ドットがディレクトリの区切りに対応します。

```
module util.text

def shout(s: String): String =
  s + "!"
```

つまりモジュール名は、ファイルへの道順そのものです。名札が中身と食い違っていると、読み込む側で叱られます（10.5節）。

なお、`--eval` や `--check` に直接渡す**入口のファイル**だけは、宣言とパスの一致を求められません（本書がこれまで `module hello` を色々な場所に置けたのはこのためです）。

10.2 import — 名前を持ってくる

読む側は `import` です。 `main.lune` :

```
module main
import geometry
import util.text

let area = circleArea(1.0)

let banner = shout("area")
```

```
$ lune --eval area main.lune
3.14159
$ lune --eval banner main.lune
"area!"
```

`import geometry` と書くと、`geometry.lune` のトップレベル宣言（`def`・`let`・`type`・`record`）が**そのまま自分の環境に入ります**。`geometry.circleArea(...)`ではなく `circleArea(...)` と、非修飾で呼べていることに注目してください。

修飾して呼ぶことは、v0.1 ではできません。

```
lune> let a = geometry.circleArea(1.0)
error[TYP0001]: 未定義の名前: geometry
  = help: 詳しくは `lune explain TYP0001 --lang ja` を実行してください
```

`geometry` という名前の値は存在しない — `import` が運んでくるのは中身の名前だけ、というわけです。裏を返せば、**別々のモジュールが同じ名前を定義していると衝突します**。v0.1 では名前を短くしすぎない（`total` より `cartTotal`）のが自衛策です。

`module` 宣言と `import` の行は、間に空行を挟まず続けて書くのが正準形です（`lune fmt` がこの形に整えます。第12章）。

10.3 分け方の作法

どう分けるか。v0.1 の道具立てで実際に効くのは、次の2つの発想です。

データとその操作をまとめる。レコードや ADT の定義と、それを扱う関数を同じモジュールに置きます。演習 10-2 で作る `shop/items.lune` がこの形です。

```

module shop.items

record Item:
  name: String
  price: Int

def priceOf(item: Item): Int =
  item.price

def total(items: List[Item]): Int =
  fold(map(items, priceOf), 0, fn a x -> a + x)

```

汎用の道具を切り出す。文字列や数値のユーティリティのように、どこからでも呼ばれるものを `util.text` のような場所へ。

逆に、v0.1 では**やらないほうがよい**分け方もあります。「実装を隠すために分ける」 — 公開/非公開の制御がないので、分けても全部見えます。「循環しそうな相互参照を分ける」 — 次節のとおり、循環は禁止です。

10.4 循環 import は検出される

`cycle_a` が `cycle_b` を import し、`cycle_b` が `cycle_a` を import したらどうなるか。

```
$ lune --check cycle_a.lune
```

```

error[MOD0002]: モジュールの循環 import を検出しました: cycle_a.lune ->
cycle_b.lune -> cycle_a.lune
  --> cycle_b.lune:2:1
  |
2 | import cycle_a
  | ^^^^^^^ この import が循環を閉じている
  = help: 詳しくは `lune explain MOD0002 --lang ja` を実行してください

```

診断が**循環の経路をそのまま** (`cycle_a → cycle_b → cycle_a`) 見せ、しかも「輪を閉じた import」を名指ししてくれます。どこを切ればよいか一目で分かる形です。

なぜ禁止なのか。Lune は依存モジュールを**先に**型検査して評価します (`main` を検査する前に `geometry` を検査する)。循環があると「先」が決められません。第4章で `let x = x + 1` が `RUN0005` になったのと同じ理屈が、ファイルの粒度で現れたものです。

直し方は3つ。共通部分を第3のモジュールに抜き出す、依存の向きを片方だけに直す、あるいは2つを1つのファイルに戻す (同じファイル内の相互参照は問題ありません)。

10.5 モジュールが見つからないとき — 探索の順番

`import foo.bar` は `foo/bar.lune` を探します。探す場所は3つ、この順です。

1. 入口ファイルのあるディレクトリ
2. カレントディレクトリ
3. `--module-path` で足したディレクトリ（複数指定可）

3番目が便利なのは、共有ライブラリを別ディレクトリに置きたいときです。

`lib/shared.lune`（`module shared`）を `usesshared.lune` から使う例:

```
$ lune --check usesshared.lune
error[MOD0001]: モジュールが見つかりません: shared
$ lune --module-path lib --check usesshared.lune
type check OK
$ lune --module-path lib --eval answer usesshared.lune
42
```

見つからないときの診断は、**探した場所を教えてください**。

```
error[MOD0001]: モジュールが見つかりません: nowhere
--> missing.lune:2:1
  |
2 | import nowhere
  | ^^^^^^^ 対応する .lune ファイルが見つからない
  = hint: 検索した場所: .
  = help: 詳しくは `lune explain MOD0001 --lang ja` を実行してください
```

（`検索した場所` は実際には絶対パスで表示されます。紙面では省略しています。）

ファイルは見つかったのに名札が違う場合は、別の診断です。 `mismatch.lune` の中身が `module totally.different` だったとき:

```
error[MOD0003]: module 宣言の不一致: mismatch のはずが totally.different でした
--> mismatch.lune:1:1
  |
1 | module totally.different
  | ^^^^^^^ モジュール名が import パスと一致しない
  = hint: 宣言を `module mismatch` に直すか、`totally.different` として import してください
  = help: 詳しくは `lune explain MOD0003 --lang ja` を実行してください
```

hint が**両方向の直し方**（名札を直す / import を直す）を示していることに注目してください。どちらが正しいかは設計判断なので、機械は決めずに両方を並べる — 第11章で学

ぶ「hint は原因と対処」の good example です。

10.6 外部 import — java.* と std.*

最後に、Lune のファイルを指さない import があります。

```
module basics
import java.time.LocalDate

def today(): String =
  LocalDate.now().toString()
```

java.* ・ javax.* ・ kotlin.* ・ std.* で始まる import は**外部 import**として扱われ、ファイル解決されません。型検査は末尾の名前（LocalDate）を Any として登録するだけなので、この定義は型検査を通ります — が、実際に呼び出せば実行時に失敗します。JVM 連携は将来仕様（付録E）で、v0.1 では「構文だけ受け付ける」段階です。

Any として通ってしまう以上、外部 import は型検査の保護が効かない領域です。v0.1 で書くプログラムでは使わないのが賢明です。

壊してみよう — この章の3つの MOD 診断のうち、MOD0001（見つからない）と MOD0003（名札不一致）を自分で出してみてください。片方から他方へ移る実験もできます: import nothere のまま nothere.lune を作り、中身を module wrong.name と書けば、MOD0001 が MOD0003 に変わります。

```
error[MOD0003]: module 宣言の不一致: nothere のはずが wrong.name でした
--> nothere.lune:1:1
  |
1 | module wrong.name
  | ^^^^^^^ モジュール名が import パスと一致しない
  = hint: 宣言を `module nothere` に直すか、`wrong.name` として import してください
  = help: 詳しくは `lune explain MOD0003 --lang ja` を実行してください
```

「ファイルを見つける」と「名札を照合する」が別の段階だと体感できます。なお module something.else と書くと else が予約語なので構文エラー（PRS0002） — モジュール名にキーワードは使えません。

まとめ

概念	一言で
<code>module a.b</code>	名札。 <code>a/b.lune</code> と一致必須（入口ファイルは例外）
<code>import a.b</code>	相手のトップレベル宣言が自分の環境に入る（非修飾で使う）
名前空間アクセス	<code>a.b.f()</code> は未対応。名前の衝突には自分で気をつける
正準形	<code>module</code> と <code>import</code> は空行を挟まず続ける
循環	<code>MOD0002</code> 。依存は先に評価されるので輪は作れない
探索順	入口ファイルの場所 → カレント → <code>--module-path</code>
診断	<code>MOD0001</code> 見つからない（探した場所つき） / <code>MOD0003</code> 名札不一致（両方向の hint）
外部 import	<code>java.*</code> <code>std.*</code> などは Any 登録のみ。v0.1 では実質使えない

演習問題

演習 10-1 (★) `import nothere` と書いたファイルを作って `MOD0001` を出し、次に `nothere.lune` を作って中身を `module wrong.name` にして `MOD0003` に変えてください。

解答

本文 10.5 節と「壊してみよう」の実例そのままです。 `MOD0001` の hint は「探した場所」、 `MOD0003` の hint は「名札を直す / import を直す」の2択。ファイル解決 → 名札照合の順に検査が進んでいることが、診断の変化から読み取れます。

演習 10-2 (★★) 第6章の `Item` レコードと集計処理を `shop/items.lune` (`module shop.items`) に切り出し、別ファイルの `main` から使って合計を求めてください。

解答

```
module shop.items

# 演習 10-2: データとその操作をモジュールに切り出す。
record Item:
  name: String
  price: Int

def priceOf(item: Item): Int =
  item.price

def total(items: List[Item]): Int =
  fold(map(items, priceOf), 0, fn a x -> a + x)
```

```
module shop_main
import shop.items

# import した名前は非修飾で使える。Item も priceOf も total も。
let cart = [Item(name = "pen", price = 120), Item(name = "note", price =
200)]

let sum = total(cart)
```

```
$ lune --eval sum shop_main.lune
320
```

`record Item` も `import` で運ばれてくるので、使う側は `Item(name = ..., price = ...)` と普通に構築できます。型・関数・値の区別なく、トップレベル宣言はすべて渡ってきます。

演習 10-3 (★★) `lib/` ディレクトリに `module shared` のファイルを置き、`--module-path` を使って読み込んでください。`--module-path` を付けない場合との違いも確認すること。

解答

本文 10.5 節のとおりです。付けなければ `MOD0001`、付ければ通ります。

`lib/shared.lune` は `module shared` (`module lib.shared` ではない) である点に注意 — `--module-path lib` は「`lib` を探索の起点にする」という指定なので、`lib` はモジュール名の一部になりません。

演習 10-4 (★★★) 2つのモジュールで循環を作って `MOD0002` を出し、そのうえで共通部分を第3のモジュールに抜き出して解消してください。

解答

`cycle_a` ⇔ `cycle_b` で循環を作れます (本文 10.4 節)。解消は、両者が必要とする定義を `common.lune` に移し、`cycle_a` と `cycle_b` の両方が `common` だけを import する形にします。依存グラフが「輪」から「木」に変わるので、評価順が決まるようになります。診断が経路 (`a -> b -> a`) を見せてくれるので、どの辺を切るかの判断はそこから始めるとよいでしょう。

演習 10-5 (★) `import java.time.LocalDate` を書いたファイルは型検査を通ります。なぜでしょうか。また、それが安全でないのはなぜでしょうか。

解答

外部 import は末尾名 `LocalDate` を `Any` として登録するだけで、実体を解決しないからです (10.6 節)。`Any` は何にでも使えてしまうので、`LocalDate.nonexistentMethod()` のような誤りも型検査を通り抜け、実行時まで問題が見つかりません。型検査の保護が効かない — v0.1 で外部 import を避ける理由です。

より正確には — 探索 root の決定、外部 import の判定、読み込み順と循環検出の規範は `documents/MODULE_LOADING_SPEC.md`。MOD 族の診断は `documents/ERROR_INDEX_JA.md` にも収録されています。この章のコード例は `books/examples/ch10/` にあり、すべて実際の CLI で検証されています。

第11章 エラーと対話する

ここまでの章で、あなたは30個近い診断に出会ってきました。タイポを名指しされ、足りない `match` のケースを反例つきで教わり、`null` の考慮漏れをコンパイル時に止められた。この章では、その体験を**体系**にします。

Lune にとって診断は「失敗の通知」ではなく、言語のユーザーインターフェースの半分です。すべての診断コードに教材としての解説が付いていること、それがテストで保証されていること — この設計に乗って、コンパイラを「うるさい門番」から「隣に座る家庭教師」に変えるのがこの章の目標です。

11.1 診断の解剖学 — 部品は6つ

第1章で軽く触れた診断の構造を、今度は完全に分解します。標本には、部品が全部そろっている `TYP0008`（第5章）を使いましょう。

```
error[TYP0008]: let の束縛に反駁可能パターンは使えません: Some(x)
--> refutable.lune:3:5
|
3 | let Some(x) = Some(1)
|    ^^^^ このパターンはマッチに失敗し得る
  = hint: このパターンは None をカバーしていません
  = hint: `match` を使って Option[Int] の全ケースを場合分けしてください
  = help: 詳しくは `lune explain TYP0008 --lang ja` を実行してください
```

部品	この例では	役割
重大度	<code>error</code>	<code>error</code> （検査が失敗する）か <code>warning</code> （通るが要注意）
コード	<code>TYP0008</code>	診断の背番号。調べるための鍵（11.3節）
要約	<code>let の束縛に...</code>	一行で「何がだめか」
場所とソース断片	<code>--> refutable.lune:3:5</code> と <code>^^^^</code>	どこの、どの部分か
ラベル	<code>このパターンはマッチに失敗し得る</code>	<code>^^^^</code> が指す箇所の説明
hint / help	<code>= hint: ... ×2</code>	直し方の提案と、詳しく学ぶ道

読み順のコツは「**要約** → **ラベル** → **hint**」です。要約で何の話か掴み、ラベルでどこの話が特定し、hint で次の一手を得る。hint が複数あるときは、**原因** → **対処**の順に並んでいます（この例なら「None をカバーしていない」が原因、「match を使え」が対処）。

`warning` は検査を止めません。第5章の `TYP0009` で見たとおり、警告だけなら最後に `type check OK` が出て、検査は成功扱いです。ただし警告は「動くけれど、たぶん意図と違う」の印 — 放置しない習慣をつけましょう。

11.2 コード体系 — 7つの族と検査の段階

コードの頭3文字は**族**を表し、族はそのままコンパイラの**検査段階**に対応しています。ソースコードは上から順に、レイアウト → 字句 → 構文 → モジュール → 型の検問を通り、全部を通過したものだけが実行されます。

族	段階	例	主に出会う章
LAY	レイアウト（インデント）	不整合なインデント	第2章
LXL	字句	知らない文字、閉じていない文字列	第2章
PRS	構文解析	予期しないトークン	第2・6章
MOD	モジュール解決	見つからない import、循環	第10章
TYP	型検査	未定義名、型不一致、網羅性	全域
REC	レコード検査	フィールドの過不足・重複	第6章
RUN	実行時	0除算、再帰サンク	第4章

この表から2つのことが読み取れます。第一に、**手前の族ほど早く捕まる** — `LXL` のエラーが出ている間は、型の話はまだ始まっていません。第二に、`RUN` だけが実行時です。`--check` が通っても `RUN` は出うる（第4章の `crash()` や 0 除算） — 型検査は多くを守りますが、すべてではありません。

コードは現在全部で 30 個。一覧は付録Cに、全コードの解説カタログは `documents/ERROR_INDEX_JA.md`（英語版は `ERROR_INDEX.md`）にあります。

11.3 explain — エラー番号は調べるためにある

診断コードの解説には、3つの入り口があります。

```
$ lune explain TYP0007          # シェルから (LUNE_LANG=ja なので日本語で出ます)
```

```
lune> :explain TYP0007          # REPL の中から
```

そして全部まとめて読みたければ `lune explain --index` — これが先ほどの `ERROR_INDEX_JA.md` の正体です（このファイルは索引コマンドの出力をそのまま保存したもので、古くなるとテストが落ちる仕組みになっています）。

各解説は「何が起きたか / 再現する最小の例 / 直し方」の3部構成で、第1章の `TYP0001` や第4章の `RUN0005` で読んだとおりです。存在しないコードを聞くと、聞けるコードの一覧が返ってきます。

```
$ lune explain ZZZ9999
error: no explanation for diagnostic code 'ZZZ9999'
available codes: LAY0001, LAY0002, LXL0001, LXL0002, LXL0003, LXL0004,
MOD0001, MOD0002, MOD0003, PRS0001, PRS0002, REC0001, REC0002, REC0003,
REC0004, REC0005, REC0006, RUN0005, RUN0006, TYP0001, TYP0003, TYP0004,
TYP0005, TYP0006, TYP0007, TYP0008, TYP0009, TYP0010, TYP0011, TYP0012,
TYP0013
```

「エラー番号は覚えるものではなく、**調べるための鍵**」 — これがコード体系の使い方です。

11.4 機械が直せるエラー — did-you-mean と lune fix

診断の中には、直し方が一意に決まるものがあります。その代表がタイポです。わざと2つ仕込んだ `typos.lune` :

```
module stats

let values = [3, 1, 4, 1, 5]

let total = fold(valeus, 0, fn a x -> a + x)

let count = lenght(values)
```

```
$ lune --check typos.lune
```

```
error[TYP0001]: 未定義の名前: valeus
--> typos.lune:5:18
|
5 | let total = fold(valeus, 0, fn a x -> a + x)
|               ^^^^^^^ この名前は定義されていない
= hint: もしかして `values` ですか?
= help: 詳しくは `lune explain TYP0001 --lang ja` を実行してください
```

`--check` が報告するのは最初の1つだけですが、`lune fix` は hint を機械的に適用し、直った結果をまた検査して、直せるものがなくなるまで繰り返します。

```
$ lune fix typos.lune
module stats

let values = [3, 1, 4, 1, 5]

let total = fold(values, 0, fn a x -> a + x)

let count = length(values)
```

`valeus` も `lenght` も、一度に両方直りました。 `--write` を付ければファイルを直接書き換え、 `--check` を付ければ「直せるものが残っていれば失敗する」CI 向けの検査になります。

```
$ lune fix --check typos.lune
typos.lune: 2 auto-fixable issue(s)
```

一方、`fix` は直せないものには手を出しません。第2章の `annot.lune` (`let n: Int = "hello"`) に `fix` をかけても、ソースはそのまま出力されます。 `Int` に直すべきか `"hello"` を変えるべきか、それは設計判断であって、機械には決められないからです。この線引きを覚えてください — `hint` が一意なら機械の仕事、判断が要るなら人間の仕事。

11.5 エラー駆動開発 — witness に導かれて書く

仕上げて、この本がずっと予告してきた開発スタイルを1本通しでやります。先に骨組みを書いて、何が足りないかはコンパイラに聞くのです。

お題はじゃんけん。「`a` は `b` に勝つか」を判定します。まず型と、確実に分かっている1ケースだけ書きます。

```
lune> type Hand =
...     | Rock
...     | Paper
...     | Scissors
...
ok
lune> def beats(a: Hand, b: Hand): Bool =
...     match (a, b):
...         | (Rock, Scissors) -> true
...
error[TYP0007]: 網羅的でない match: (Paper, Rock) のケースがありません
--> <repl:2>:2:5
|
2 |     match (a, b):
|     ^^^^^ パターン (Paper, Rock) がカバーされていない
    = hint: (Paper, Rock) のケースを追加するか、ワイルドカードケース `| _ -> ...`
    を追加してください
    = help: 詳しくは `lune explain TYP0007 --lang ja` を実行してください
```

コンパイラが**次に書くべき勝ち筋**を挙げてきました。(Paper, Rock) — 紙は石に勝つ。言われたとおり、勝ち筋を全部書きます。

```
lune> def beats(a: Hand, b: Hand): Bool =
...     match (a, b):
...         | (Rock, Scissors) -> true
...         | (Paper, Rock) -> true
...         | (Scissors, Paper) -> true
...
error[TYP0007]: 網羅的でない match: (Rock, Rock) のケースがありません
--> <repl:3>:2:5
|
2 |     match (a, b):
|     ^^^^^ パターン (Rock, Rock) がカバーされていない
    = hint: (Rock, Rock) のケースを追加するか、ワイルドカードケース `| _ -> ...`
    を追加してください
    = help: 詳しくは `lune explain TYP0007 --lang ja` を実行してください
```

今度は (Rock, Rock) — **あいこの存在**を指摘されました。勝ち筋以外はすべて「勝ちではない」ので、ワイルドカードで受けます。

```
lune> def beats(a: Hand, b: Hand): Bool =
...     match (a, b):
...         | (Rock, Scissors) -> true
...         | (Paper, Rock) -> true
...         | (Scissors, Paper) -> true
...         | _ -> false
...
ok
lune> beats(Rock(), Scissors())
true : Bool
lune> beats(Scissors(), Rock())
false : Bool
```

振り返ってください。私たちは仕様の抜け（紙の勝ち筋、あいこ）を**一つも自力で思い出していません**。全部コンパイラが、具体的な反例の形で持ってきました。診断は読むものではなく、**TODO リストとして消化するもの** — これがエラー駆動開発です。

型を先に書くほど、コンパイラは良い質問をくれます。ADT で形を列挙し、match で骨組みを作り、`--check` を回す。埋め終わったら `fix` と `fmt` で仕上げる。この繰り返しが、Lune の開発の基本サイクルです。

壊してみよう — `rps.lune` の `| _ -> false` を消して `--check` し、witness を確認したら、ワイルドカードではなく**あいこ3つと負け筋3つを列挙**して網羅してみてください。どちらの書き方にも利点があります: ワイルドカードは短く、列挙は `Hand` に手を足したとき（演習 5-1 の教訓）に守られます。

まとめ

概念	一言で
診断の部品	重大度 / コード / 要約 / 場所+ラベル / hint / help。要約→ラベル→hint の順に読む
hint の並び	原因 → 対処
7つの族	LAY・LXL・PRS・MOD・TYP・REC は実行前、RUN だけ実行時
<code>lune explain / :explain / --index</code>	全31コードに解説。番号は調べるための鍵
<code>lune fix</code>	hint が一意なら機械が直す（反復適用）。 <code>--write / --check</code>
fix の線引き	機械が決められない修正には手を出さない
エラー駆動開発	骨組み → <code>--check</code> → witness を TODO として消化

演習問題

演習 11-1 (★) 11.1節の標本（TYP0008）について: この診断の「原因を述べる hint」と「対処を述べる hint」はそれぞれどれですか。また、なぜ `--check` の結果は失敗（exit 非0）になるのですか。

解答

原因は1つ目（「このパターンは None をカバーしていません」）、対処は2つ目（「`match` を使って...場合分けしてください」）。失敗になるのは重大度が `error` だからです。`warning`（TYP0009 など）だけなら検査は成功し、`type check OK` が出ます。

演習 11-2 (★★) `typos.lune` と `annot.lune`（第2章）のそれぞれに `lune fix` をかけると何が起きるか、予想してから確かめてください。

解答

`typos.lune` は2つのタイプが両方直った完全なソースが出力されます。`annot.lune` はそのまま出力されます — TYP0003 には機械的に適用できる一意の hint がないからです。`fix --check` を使うと、前者は「2 auto-fixable issue(s)」で失敗（exit 非0）、後者は「直せるものがない」ので成功します。

演習 11-3 (★★) `beats` を部品にして、`"win" / "lose" / "draw"` の3値を返す `judge(a, b)` を書いてください。

解答

```
def judge(a: Hand, b: Hand): String =  
  if beats(a, b):  
    "win"  
  elif beats(b, a):  
    "lose"  
  else:  
    "draw"
```

```
$ lune --eval tied ex11-3.lune  
"draw"
```

勝敗の規則は `beats` に1箇所だけ書き、`judge` はそれを2回聞くだけ — 規則を2箇所に書かないのが要点です。

演習 11-4 (★★★・逆転総合) `LXL`・`PRS`・`TYP`・`REC`・`RUN` の5つの族から1つずつ、診断を出す最小のコードを書いてください（この本でまだ見せていないコードが出せたらボーナス点）。

解答

例: `let x = $1` (`LXL0001`)、`let a = (1` (`PRS` 系 — 閉じていない括弧)、`1 + true` (`TYP0003`)、`User(name = "X", name = "Y", age = 1)` (`REC0004`、第6章)、`--eval` で `1 / 0` を含む束縛を評価 (`RUN0006`)。ボーナス例: `###` を閉じずにファイルを終わると `LXL0003` (閉じていないブロックコメント) が出ます。全31コードの索引 (付録C / `ERROR_INDEX_JA.md`) と照合してみてください。

演習 11-5 (★) `lune explain --index` の出力を眺めて、まだ本書に登場していない診断コードを1つ見つけ、その解説を読んでください。

解答

例えば `MOD0002` (循環 import) はまだ登場していません — 第10章のお楽しみです。
`LAY0002` (対応しない閉じ括弧) や `REC0001` (レコード宣言のフィールド重複) も、狙って出すにはひと工夫要ります。

より正確には — 診断モデルと表示形式の規範は

`documents/ERROR_DIAGNOSTICS_SPEC.md`、全コードのカタログは

`documents/ERROR_INDEX_JA.md` (`lune explain --index` で再生成)、`fix` の適用規則は

`documents/ERROR_DIAGNOSTICS_SPEC.md` §9.5。この章のコード例は
`books/examples/ch11/` にあり、すべて実際の CLI で検証されています。

第12章 REPL・整形・検査

第11章では診断との対話を学びました。この章は道具そのものの話です — REPL のコマンド一覧、正準フォーマッタ `lune fmt`、CLI の検査系フラグ、そしてそれらを CI に並べるレシピ。

一つ確認しておきたいことがあります。Lune の道具はどれも**答え合わせのための道具**です。型が分からなければ `:type`、評価の順序が怪しければ `:trace`、整形で揉めたら `fmt`。頭の中で考え込む代わりに機械に聞く癖をつけると、Lune は驚くほど素直な言語になります。

12.1 REPL コマンド全一覧

`:help` が全部教えてくれます。

```
lune> :help
commands: :help, :quit, :q, :env, :type NAME, :thunks [NAME], :trace
[on|off], :lang [en|ja], :explain CODE [en|ja]
```

コマンド	何をするか	学んだ章
<code>:help</code>	この一覧	—
<code>:quit</code> / <code>:q</code>	終了	第1章
<code>:env</code>	いま見えている名前と型を全部並べる	本章
<code>:type NAME</code>	名前の型を答える	本章
<code>:thunks [NAME]</code>	束縛の評価状態を、評価せずに見せる	第4章
<code>:trace [on off]</code>	force の実況中継を切り替える	第4章
<code>:lang [en ja]</code>	診断の言語を切り替える	第1章
<code>:explain CODE</code>	診断コードの解説	第11章

複数行の入力は、行末が `=` や `:` や `->` で終わると継続に入り (`...` プロンプト)、空行で確定します。第1章から使ってきたあの動きです。

端末で起動していれば、`readline` 風に行編集と履歴（上下矢印）も効きます。環境が対応していれば履歴は `~/.lune_history` に保存され、前のセッションの実験を呼び戻せます。

12.2 :type と :env — 型に聞く

`:type` は本書で最も使われた道具かもしれません。定義した関数にも、プレリユードの関数にも効きます。

```
lune> def add(a: Int, b: Int): Int =  
...     a + b  
...  
ok  
lune> :type add  
add : Int -> Int -> Int
```

レコードのコンストラクタに聞くと、フィールド名まで含んだ形が返ります。

```
lune> :type User  
User : (name: String, age: Int) -> User
```

第6章で「レコードは名前付き構築が必須」と学びましたが、その事実が型そのものに書かれているわけです。

`:env` はいま見えている名前を**全部**並べます。プレリユードも含むので長いリストになりますが、末尾に自分の定義が混ざって見えるのが面白いところです。

```
lune> let x = 40  
ok  
lune> :env  
Cons : [T] T -> List[T] -> List[T]  
Err : [T, E] E -> Result[T, E]  
...  
take : [T] List[T] -> Int -> List[T]  
takeWhile : [T] List[T] -> (T -> Bool) -> List[T]  
tick : () -> Int  
x : Int  
zip : [T] List[T] -> List[U] -> List[Tuple[T, U]]
```

「プレリユードに何があったか」を思い出したいときの手っ取り早い索引です（正式な一覧は付録B）。

知らない名前を聞くと、当然のように叱られます。

```
lune> :type nosuch  
error[TYP0003]: 未定義の名前: nosuch  
= help: 詳しくは `lune explain TYP0003 --lang ja` を実行してください
```

（余談ですが、未定義の名前には専用コード `TYP0001` があるので、ここが `TYP0003` なのは実装の小さな取りこぼしです。診断コードの一貫性も、本来はテストで守るべきもの

です。)

12.3 lune fmt — 議論を終わらせる整形

インデントは2つか4つか、`fn a x -> a+x` に空白を入れるか。この種の議論に費やす時間を Lune はゼロにします。**正準形はひとつ**で、`lune fmt` がそこへ揃えます。

散らかったファイル `messy.lune` :

```
module messy

let   nums = [ 1,2 ,3 ]

# 合計を出す
def total( xs : List[Int] ) : Int =
    fold( xs, 0, fn a x -> a+x )

let answer=total(nums)
```

```
$ lune fmt messy.lune
module messy

let nums = [1, 2, 3]

# 合計を出す
def total(xs: List[Int]): Int =
    fold(xs, 0, fn a x -> a + x)

let answer = total(nums)
```

余分な空行が1つに、括弧まわりの空白が整い、インデントは4つ、`#` コメントは位置ごと保たれました。

この整形には、2つの保証があります。

冪等 (idempotent) — 一度整形した結果をもう一度整形しても、何も変わりません。「整形するたびに揺れる」ことがないので、CI に置いても安全です。

意味を変えない — `fmt` は AST (構文木) を整形して印字し、**自分の出力を読み直して構文木が一致するか確かめてから**結果を返します。もし一致しなければ、整形を諦めてエラーを出します (第2章の脚注で触れた入れ子 `let-in` のバグがまさにこれで捕まりました)。整形が黙ってプログラムを壊すことはありません。

使い方は3つのモードです。

```
$ lune fmt FILE           # 整形結果を標準出力へ（元ファイルは変えない）
$ lune fmt --write FILE   # ファイルを整形して書き換える
$ lune fmt --check FILE   # 整形が必要なら失敗する（CI 用）
```

`--check` は差分があるとファイル名を挙げて終了コード 1 を返します。

```
$ lune fmt --check messy.lune
would reformat messy.lune
```

制限が一つ。 `###` で囲むブロックコメントを含むファイルは、まだ整形できません。

```
$ lune fmt bc.lune
error: bc.lune: lune fmt does not support `###` block comments yet
```

整形されない（＝壊されない）だけなので安全側の挙動ですが、 `fmt` を CI に置くなら `###` は避けて `#` を並べるのが今の作法です。

12.4 CLI の検査系フラグ

`lune` コマンドの主なフラグを、使う順に並べます。

コマンド	何をするか
<code>lune --repl</code>	REPL を起動
<code>lune --check FILE</code>	型検査だけして終わる（評価しない）
<code>lune --eval NAME FILE</code>	束縛 NAME を評価して表示
<code>lune --eval NAME --trace FILE</code>	上に加えて force のトレースを stderr へ（第4章）
<code>lune --module-path DIR</code>	モジュール探索ルートを追加（第10章、繰り返し可）
<code>lune --tokens FILE</code>	字句解析の結果（トークン列）を表示
<code>lune explain CODE</code>	診断コードの解説（第11章。 <code>--index</code> で全部）
<code>lune fmt / lune fix</code>	整形 / 自動修正（12.3節、第11章）
<code>--lang en ja</code>	診断の言語（本書は <code>ja</code> ）。既定は環境変数 <code>LUNE_LANG</code> 、フラグが優先

`--tokens` は普段使う道具ではありませんが、レイアウト（インデント）の扱いを覗くのに便利です。第2章で「インデントは構文の一部」と言ったことの実物が見られます。

```
$ lune --tokens geometry.lune
1:1      MODULE  'module'      None
1:8      IDENT   'geometry'     'geometry'
1:16     NEWLINE ''           None
3:1      DEF     'def'         None
3:5      IDENT   'circleArea'  'circleArea'
...
```

`--check` と `--eval` の関係も押さえておきましょう。 `--eval` は型検査を通しません — 第4章で `RUN0005`（再帰サンク）を出せたのはこのためです。実行する前に検査したければ、`--check` を先に走らせるのが確実です。

存在しない束縛を `--eval` すると実行時エラーになります。

```
$ lune --eval nosuch geometry.lune
error[RUN0006]: 未定義の変数: nosuch
```

12.5 小さな CI レシピ

道具を並べると、そのままプロジェクトの検査になります。終了コードは「問題があれば非0」で揃っているので、シェルの `&&` で繋がれば1本のパイプラインです。

```
lune fmt --check src/*.lune && lune --check src/main.lune && lune fix --check src/*.lune
```

3つの意味はこうです。

1. `fmt --check` — 全ファイルが正準形か（整形漏れがないか）
2. `--check` — 型検査が通るか
3. `fix --check` — 機械が直せる問題（タイポなど）が残っていないか

終了コードの規則を整理しておきます。

状況	終了コード
検査が通る	0
警告だけ （TYP0009 など）	0
エラーがある	1
fmt --check で差分あり	1
fix --check で直せる問題あり	1

注意すべきは2行目です。第5章で見たとおり、警告だけなら検査は成功します。警告も見逃したくなければ、出力に `warning` が現れたら落とす一手間が必要です。

```
lune --check src/main.lune 2>&1 | tee /tmp/out; ! grep -q warning /tmp/out
```

壊してみよう — `messy.lune` を `fmt --write` で整形してから、もう一度 `fmt --check` を走らせてください。今度は何も言われたいはず（冪等の確認）。ついでに `fmt` した結果を自分の目で読み、自分の書き癖と正準形の差を数えてみるのも良い練習です。

まとめ

道具	一言で
<code>:help</code>	REPL コマンドの一覧。迷ったらここ
<code>:type / :env</code>	型に聞く / 見えている名前を全部並べる
<code>lune fmt</code>	正準形へ整形。 冪等 で 意味を変えない （再パースで自己検査）
fmt の3モード	標準出力 / <code>--write</code> / <code>--check</code> （CI）
fmt の制限	<code>###</code> ブロックコメントは未対応（安全に諦める）
<code>--check</code> vs <code>--eval</code>	<code>--eval</code> は型検査を通さない。検査したいなら <code>--check</code> を先に
<code>--tokens</code>	字句とレイアウトの覗き窓
CI レシピ	<code>fmt --check && --check && fix --check</code> 。警告だけなら終了コードは 0

演習問題

演習 12-1 (★) `messy.lune` を `fmt` にかけた結果を予想してから確かめてください。予想と違った箇所はどこですか。

解答

本文 12.3 節のとおりです。よく外れるのは「連続した空行が1つに詰められる」「`def total( xs : List[Int] ) : Int =`」の括弧とコロンまわりの空白」「`a+x` に空白が入る」あたり。`#` コメントが消えずに残ることも確認してください。

演習 12-2 (★) `:env` を使って、プレリユードに `zipWith` があることと、その型を確かめてください。第8章で使ったときの記憶と合っていますか。

解答

`zipWith : [T, U, V] List[T] -> List[U] -> (T -> U -> V) -> List[V]`。「2本のリストと、2引数の関数を受け取り、1本のリストを返す」— 第8章の移動平均（演習 8-3）でやったことが、型に書いてあります。`:type zipWith` でも同じ答えが得られます。

演習 12-3 (★★) `:type` で確かめながら、`scale(factor, n)` から「3倍にする関数」を部分適用で作り、リストに `map` してください。途中で `:type scale(3)` のように部分適用の型も確かめること。

解答

```
module answers

# 演習 12-3: :type で確かめながら組み立てた、部分適用で作る道具。
def scale(factor: Int, n: Int): Int =
  factor * n

let double = scale(2)

let tripled = map([1, 2, 3], scale(3))

let doubled = map([1, 2, 3], double)
```

```
$ lune --eval tripled ex12-3.lune
(3 6 9)
$ lune --eval doubled ex12-3.lune
(2 4 6)
```

REPL では `:type scale` が `Int -> Int -> Int`、`let triple = scale(3)` としてから `:type triple` が `Int -> Int`。第3章で読んだ「`->` は右結合」がそのまま観察できます。

演習 12-4 (★★) 自分のプロジェクト（この本の演習ファイルでも構いません）に対して、12.5節の CI レシピを走らせてください。1つでも失敗するファイルを見つけたら、何が原因かを診断から説明してください。

解答

本書の `books/examples/` で試すなら、意図的に壊してあるファイル（`ch02/annot.lune` や `ch11/typos.lune` など）が引っかけられます。 `typos.lune` は `--check` でも `fix --check` でも落ちますが、意味が違います — 前者は「型検査が通らない」、後者は「機械が直せる問題が残っている」。同じファイルに対して**別の理由**で落ちていることを読み取れれば正解です。

演習 12-5 (★・逆転問題) `lune fmt` が「整形を諦める」場面を作ってください。

解答

`###` で囲んだブロックコメントを含むファイルを `fmt` にかければ、`lune fmt does not support ### block comments yet` で諦めます（12.3節）。もう一つの諦め方は、フォーマッタ自身のバグで意味保存の自己検査に失敗する場合です（`formatter changed the program's meaning (internal bug)`）。後者を狙って出すのは難しいですが、見つけたらそれは報告すべきバグです — 実際、本書の執筆中に1件見つかりました。

より正確には — REPL の仕様は `documents/REPL_SPEC.md` (`:thunks` / `:trace` の表示形式を含む)、フォーマッタの保証と制限は `documents/FORMATTER_SPEC.md`、CLI の全フラグは付録D。この章のコード例は `books/examples/ch12/` にあり、すべて実際の CLI で検証されています。

第13章 プログラムを組み立てる — ケーススタディ

部品はすべて揃いました。この章では3つの小さなプログラムを、設計から完成まで通して書きます。新しい構文は出てきません。代わりに、これまで別々に学んだ道具がどう噛み合うのかを見ます。

どれも同じ手順で進めます。型を決める → 骨組みを書く → コンパイラの witness に導かれて埋める → `fmt` と `--check` で仕上げる。第11章のエラー駆動開発を、実物で3回繰り返すわけです。

13.1 テキスト統計 — fold とレコードで1パス集計

K&R の入門書は文字を数えるプログラムから始まります。敬意を表して、単語の統計を作りましょう。

まず断りを一つ。 v0.1 の文字列には `length` と `+` (連結) しかなく、文字列を単語に分割する道具がありません。そこで入力は「すでに分割された単語のリスト」`List[String]` とします。分割はプログラムの外の仕事、という設計です。

型を決める。 欲しい結果は「単語数・総文字数・最長の単語」の3つ組。名前を付けて運びたいので、レコードです (第6章)。

```
record Stats:
  count: Int
  totalChars: Int
  longest: String
```

骨組みを書く。 リストを1つの値に畳むのだから `fold` (第8章)。ということは「途中の `Stats` と次の単語から、新しい `Stats` を作る」関数があればいい。 `stats.lune`:

```

module stats

record Stats:
  count: Int
  totalChars: Int
  longest: String

let empty = Stats(count = 0, totalChars = 0, longest = "")

def step(s: Stats, w: String): Stats =
  let n = length(w)
  let longer = if n > length(s.longest) then w else s.longest
  Stats(count = s.count + 1, totalChars = s.totalChars + n, longest =
longer)

def summarize(words: List[String]): Stats =
  fold(words, empty, step)

def averageLength(s: Stats): Double? =
  if s.count == 0 then null else s.totalChars / s.count

let words = ["the", "quick", "brown", "fox", "jumps"]

let summary = summarize(words)

let average = averageLength(summary)

let emptyAverage = averageLength(summarize([]))

```

```

$ lune --eval summary stats.lune
{ count = 5, totalChars = 21, longest = "quick" }
$ lune --eval average stats.lune
4.2
$ lune --eval emptyAverage stats.lune
null

```

設計上の判断が3つ入っています。

リストを1回しか歩かない。 3つの統計を別々に計算すれば `fold` を3回書くことになりませんが、レコードをアキュムレータにすれば1回で済みます。 `fold` の「途中結果」は数値でなくてよい — これが `fold` の本当の使い方です。

平均を `Double?` にした。 単語が0個のときの平均は存在しません。0を返すのは嘘なので、`null` を返します（第7章）。呼び出し側は `?? 0.0` で着地するか `match` で場合分けするかを選べます。

`step` を独立した関数にした。 `fn` で書けばラムダに埋め込めますが、名前を付けておくとREPLで単体で試せます（`step(empty, "hi")`）。テストしたいものには名前を付ける、が実務の勘所です。

13.2 家計簿 — ADT・Result・モジュール分割

次はもう少し「業務っぽい」お題です。収入と支出を記録して、残高を出します。

型を決める。項目は収入か支出のどちらか — 直和型の出番です（第5章）。データとその操作はモジュールにまとめます（第10章）。`ledger/entry.lune`:

```
module ledger.entry

type Entry =
  | Income(label: String, amount: Int)
  | Expense(label: String, amount: Int)

def signedAmount(e: Entry): Int =
  match e:
    | Income(_, amount) -> amount
    | Expense(_, amount) -> 0 - amount

def labelOf(e: Entry): String =
  match e:
    | Income(label, _) -> label
    | Expense(label, _) -> label

def validate(e: Entry): Result[Entry, String] =
  if signedAmount(e) == 0 then Err("amount must not be zero: " +
    labelOf(e)) else Ok(e)
```

ここが設計の心臓部です。**符号を型で持たない**という判断をしました。`Expense` の `amount` は正の数で持ち、符号は `signedAmount` が付けます。こうすると「支出をマイナスで記録するのか?」という混乱が起きません — 型と関数のどちらに意味を持たせるかの、小さいけれど本質的な選択です。

`validate` は失敗を**値として**返します（第5章）。例外を投げないので、呼び出し側は失敗を無視できません。`ledger_main.lune`:

```
module ledger_main
import ledger.entry

let entries = [Income("salary", 3000), Expense("rent", 1200),
Expense("food", 400)]

let balance = fold(map(entries, signedAmount), 0, fn a x -> a + x)

let expenses = filter(entries, fn e: Entry -> signedAmount(e) < 0)

let rejected = validate(Expense("zero", 0))

let accepted = validate(Income("bonus", 500))
```

```
$ lune --eval balance ledger_main.lune
1400
$ lune --eval expenses ledger_main.lune
(Expense("rent", 1200) Expense("food", 400))
$ lune --eval rejected ledger_main.lune
Err("amount must not be zero: zero")
$ lune --eval accepted ledger_main.lune
Ok(Income("bonus", 500))
```

「map で符号付き金額に変換して fold で畳む」 — 第6章の集計パターンがそのまま効いています。filter で支出だけ取り出すのも同じ発想です。

ここで **witness** の恩恵を体験してください。Entry に3つ目のケース（たとえば Transfer）を足してみると、signedAmount と labelOf の両方で TYP0007 が出ます。「振替の符号をどうするか決めていない」ことを、コンパイラが漏れなく指摘してくれる — 第11章で学んだとおりです。仕様変更に強い設計とは、こういうことです。

v0.1 の注意 — ADT のコンストラクタは位置引数で呼びます（Income("salary", 3000)）。Income(label = "salary", amount = 3000) のように名前付きで書くと TYP0012 で弾かれます。コンストラクタは部分適用できる（第3章）ので、Income(amount = 3000) は「第1引数が未充填の部分適用」を意味することになり、名前を対応させる先がないためです。名前付きが必須なのはレコードで、そこらは逆に位置引数が REC0006 になります（第6章） — この2つはちょうど裏返しの関係です。

13.3 数列の実験室 — 無限リストと評価回数の観察

最後は遅延評価の総合演習です。コラッツ数列（偶数なら半分、奇数なら3倍して1を足す。1に到達するまで続く）を無限リストで作ります。

「偶数なら半分」を書くときは割り算の種類に注意します。/ は常に Double を返すので、n / 2 では Int になりません。整数のまま割るには床除算 // です（2.2 節）。

```
lune> 7 // 2
3 : Int
lune> 7 / 2
3.5 : Double
```

```
collatz.lune:
```

```

module collatz

# `//` は整数の床除算。`/` は常に Double を返すので、Int のままにするには `//`。
def next(n: Int): Int =
  if n % 2 == 0 then n // 2 else 3 * n + 1

let fromSix = take(iterate(next, 6), 9)

let fromSeven = takeWhile(iterate(next, 7), fn n: Int => n != 1)

```

```

$ lune --eval fromSix collatz.lune
(6 3 10 5 16 8 4 2 1)
$ lune --eval fromSeven collatz.lune
(7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2)

```

`iterate(next, 6)` は「6 から始まる無限のコラッツ列」です（第8章）。何個見るかは使う側が決める — `take` で9個、あるいは `takeWhile` で「1になる直前まで」。**数列の定義と、どこまで使うかの分離**がここでも効いています。

そして最後に、遅延評価が本当に働いていることを**数値で確かめます**。`tick()`（第4章）を仕込んだ「重い」計算を用意し、5要素のリストから2個だけ取り出したとき、計算が何回走るかを測ります。`counted.lune`:

```

module counted

# tick() を仕込んだ「重い」計算。呼ばれた回数がカウンタに残る。
def costly(n: Int): Int =
  seq tick() (n * 2)

let doubled = map([1, 2, 3, 4, 5], costly)

# 先頭 2 個だけを取り、中身まで評価する。
let firstTwo = deepForce take(doubled, 2)

# そのとき costly が何回走ったか。
let cost = seq firstTwo tickCount()

```

```

$ lune --eval firstTwo counted.lune
(2 4)
$ lune --eval cost counted.lune
2

```

5要素に `map` したのに、`costly` は2回しか走りませんでした。正格な言語なら5回走ってから2個を捨てるところです。第4章から追いかけてきた遅延評価の御利益が、ついに数字で出ました。

`deepForce` が必要な理由も味わっておいてください。 `take(doubled, 2)` だけでは背骨（`Cons` の連なり）しか実体化せず、要素はサンクのままです — `:trace` で見ると `Cons(<think>, <think>)` で止まっているのが分かります（第8章の `:thunks` の観察と同じ景色）。「どこまで評価するか」を選べること自体が、Lune の設計思想の現れです。

13.4 3つのケースから学ぶこと

3本を並べると、共通の型紙が見えてきます。

1. **形を型にする**（レコードか ADT か — 直積か直和か）
2. **不変な変換で書く**（`map` / `filter` / `fold`。命令的な部品が要るならブロックに閉じ込める）
3. **「ない」「失敗した」を値で表す**（`T?` / `Result` / 単相の結果型）
4. **コンパイラに漏れを見張らせる**（`match` の網羅性、`witness` に導かれて埋める）
5. **確かめる道具を自分で作る**（`costly` と `tick()` のように。「速いはず」ではなく数字で確かめる）

そして、どのプログラムも `lune fmt` で整形し、`lune --check` を通してから完成としていきます（第12章の CI レシピ）。

演習問題

演習 13-1 (★★) `Stats` に「最短の単語」を足してください。空のリストに対しても壊れないように。

解答

初期値を `""` にすると「最短」が常に空文字列になってしまうので、「まだ無い」を `null` で表します（第7章）。

```
record Stats:
  count: Int
  totalChars: Int
  longest: String
  shortest: String?

def pickShortest(current: String?, w: String): String =
  if current == null:
    w
  elif length(w) < length(current):
    w
  else:
    current
```

```
$ lune --eval summary ex13-1.lune
{ count = 5, totalChars = 21, longest = "quick", shortest = "the" }
```

`pickShortest` の1行目で `current == null` を確かめた後は `current` が `String` に絞り込まれるので、`length(current)` が書けます（第7章の narrowing）。なお1行形式の `if ... then ... elif` は書けません — `elif` はブロック形式専用です。

演習 13-2 (★★) `Entry` に `Transfer(label: String, amount: Int)`（口座間の振替で、残高に影響しない）を足してください。コンパイラが何を要求してくるか観察すること。

解答

`type` に1行足すだけで、`signedAmount` と `labelOf` の両方に `TYP0007`（網羅的でない match）が出ます。`signedAmount` では `| Transfer(_, _) -> 0`（残高に影響しない）、`labelOf` では `| Transfer(label, _) -> label` を足せば通ります。

型を1箇所変えると、影響を受ける全箇所をコンパイラが挙げてくれる — これが `_ -> ...` のワイルドカードを避ける理由です（演習 5-1）。ワイルドカードで書いていたら、`Transfer` は黙って支出扱いになっていたかもしれません。

演習 13-3 (★★) `counted.lune` の `take(doubled, 2)` を `take(doubled, 5)` に変えると `cost` はいくつになるでしょう。`deepForce` を外したらどうなるでしょう。予想してから確かめてください。

解答

5 になります (5要素すべてを評価するので)。deepForce を外すと cost は 0 — take は背骨だけ実体化し、要素のサンクには触らないからです。「リストがある」と「中身が計算済みである」ことは別、という第8章の教訓が数字で確認できます。

演習 13-4 (★★★) $-7 // 2$ は -4 になります。 -3 ではありません。なぜそうなるのかを、 $-7 \% 2$ の値と合わせて説明してください。まず予想し、REPL で確かめてから答えること。

解答

まず両方の値を見ます。

```
lune> -7 // 2
-4 : Int
lune> -7 % 2
1 : Int
```

// の丸めは**負の無限大方向への切り下げ**（floor）で、ゼロ方向への切り捨て（truncation）ではありません。 $-7 / 2$ は -3.5 なので、floor すると -4 になります。

なぜ -3 ではないのか。// と % は無関係な2つの演算ではなく、**組で辻褄が合っていないから**です。商と余りの関係は、どんな符号でもこの等式で結ばれています。

```
a == (a // b) * b + (a % b)
```

確かめてみます。

```
lune> (-7 // 2) * 2 + (-7 % 2)
-7 : Int
```

もし // がゼロ方向切り捨てで -3 を返したとすると、% の側（1）はそのままなので:

```
lune> (-3) * 2 + (-7 % 2)
-5 : Int
```

-7 に戻りません。つまり -3 を選ぶと // と % が互いに矛盾し、「商と余り」と呼べなくなってしまう。% が除数の符号に合わせた余りを返す（ $-7 \% 2$ が -1 ではなく 1 ）と決めた時点で、// は floor でなければならないのです。片方だけを見て決められる話ではない、というのがこの演習の眼目です。

コラッツ数列には正の数しか出てこないで、next を書く限りこの違いは表に出ません。負の数を割るときに初めて効いてきます。丸め方向は

`documents/LANGUAGE_SPEC.md` の 9.1 節に規定があります。

演習 13-5 (★★★・総合) 3つのケーススタディのどれかを選んで拡張してください。例: テキスト統計に「n 文字以上の単語だけ」の絞り込みを足す / 家計簿に月ごとの集計を足す（`record Month` と `filter` の組み合わせ） / コラッツ列の「1 に到達するまでの歩数」を求める関数を書く。

解答

例として3つ目（歩数）は、`takeWhile` の結果の `length` に1を足せば求められます。

```
lune> length(takeWhile(iterate(next, 7), fn n: Int -> n != 1)) + 1
17 : Int
```

無限リストに `length` を使っていますが、`takeWhile` が先に有限化しているので止まります（第8章 8.5節）。「無限のまま加工して、有限化してから消費する」— この順序を守れているかが、遅延評価を使いこなせているかの試金石です。

より正確には — この章は新しい機能を使っていません。各節が依拠する仕様は、レコードが `documents/RECORD_FIELD_SPEC.md`、ADT と `match` が `documents/MATCH_EXHAUSTIVENESS_SPEC.md`、リストと遅延コンビネータが `documents/STANDARD_LIBRARY_SPEC.md` §6、`seq / deepForce` が `documents/LAZY_EVALUATION_SPEC.md` §9–10。この章のコード例は `books/examples/ch13/` にあり、すべて実際の CLI で検証されています。

付録A 言語リファレンスマニュアル

本文と独立に引ける、Lune v0.1 の規範的な一覧です。ここは「そういえば規則はどうだったか」を確かめる場所で、なぜそうなっているかは本文（と付録E）にあります。

規範の正は `documents/` の仕様書群です。この付録はそこから引いていますが、表の内容は実装から機械的に取り出すか、実際に処理系に通して確かめています（末尾の「この付録の出どころ」を参照）。

A.1 字句

キーワード

48 語が予約されています。このうち多くは将来のためのもので、v0.1 では構文エラーになります（A.6 を参照）。

<code>IO</code>	<code>abstract</code>	<code>as</code>	<code>catch</code>	<code>class</code>	
<code>deepForce</code>					
<code>def</code>	<code>elif</code>	<code>else</code>	<code>extends</code>	<code>false</code>	
<code>final</code>					
<code>finally</code>	<code>fn</code>	<code>for</code>	<code>force</code>	<code>if</code>	
<code>implements</code>					
<code>import</code>	<code>in</code>	<code>init</code>	<code>interface</code>	<code>internal</code>	<code>lazy</code>
<code>let</code>	<code>match</code>	<code>module</code>	<code>new</code>	<code>null</code>	
<code>override</code>					
<code>private</code>	<code>protected</code>	<code>public</code>	<code>raise</code>	<code>record</code>	<code>seq</code>
<code>static</code>	<code>strict</code>	<code>super</code>	<code>then</code>	<code>this</code>	
<code>throw</code>					
<code>throws</code>	<code>true</code>	<code>try</code>	<code>type</code>	<code>var</code>	
<code>while</code>					

`break` / `continue` / `return` / `export` / `use` は予約されていません。ただの識別子として扱われるので、書くと「未定義の名前」（TYP0001）になります。A.6 を参照。

識別子

<code>IDENT_START</code>	<code>=</code>	<code>[A-Za-z_]</code>
<code>IDENT_PART</code>	<code>=</code>	<code>[A-Za-z0-9_]</code>
<code>IDENT</code>	<code>=</code>	<code>IDENT_START IDENT_PART*</code>

`_` 単独はワイルドカードパターン、`_name` は識別子です。

リテラル

種類	例	型
整数	<code>42</code> 、 <code>1_000_000</code>	<code>Int</code> （任意精度）
浮動小数	<code>3.14</code> 、 <code>1e-3</code>	<code>Double</code>
文字列	<code>"hello"</code>	<code>String</code>
文字	<code>'x'</code>	<code>String</code> （下記）
真偽値	<code>true</code> 、 <code>false</code>	<code>Bool</code>
null	<code>null</code>	<code>Null</code>
ユニット	<code>()</code>	<code>Unit</code>
リスト	<code>[1, 2, 3]</code>	<code>List[T]</code>
タプル	<code>(1, "a")</code>	<code>Tuple[Int, String]</code>

整数リテラルの `_` は桁区切りとして無視されます。文字列のエスケープは `\n` `\r` `\t` `\\` `\"` `\'` `\0`。

文字リテラルに文字型はありません — `'x'` は字句としては専用のトークンで、中身が1文字でなければ `LXL0002` になります。しかし**型は `String` です**。v0.1 に文字型（`Char`）はなく、文字を扱う標準ライブラリ関数もありません。`let c: Char = 'x'` は、宣言されていない型名を書いたのと同じ扱いで `TYP0003` になります。`Char` は将来仕様（`LANGUAGE_FUTURE_SPEC.md`）側の型です。

コメント

```
# 行コメント
```

```
###
ブロックコメント（ネスト不可）
###
```

`//` は**コメントではありません** — 床除算の演算子です（2.2 節）。`/* ... */` もコメントではありません。

演算子表

優先順位は高い順。同じ行は同じ優先順位です。

優先度	演算子	結合
1	<code>.</code> <code>?.</code> <code>()</code> <code>[]</code>	左
2	単項 <code>!</code> 単項 <code>-</code> 単項 <code>+</code>	右
3	<code>*</code> <code>/</code> <code>//</code> <code>%</code>	左
4	<code>+</code> <code>-</code>	左
5	<code>::</code> <code>++</code>	右
6	<code>==</code> <code>!=</code> <code><</code> <code><=</code> <code>></code> <code>>=</code>	非結合
7	<code>&&</code>	左
8	<code> </code>	左
9	<code>??</code>	右
10	<code> ></code>	左
11	<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>//=</code> <code>%=</code>	右

非結合は連鎖できないという意味です。ただし止めるのはパーサではなく型検査で、`1 < 2 < 3` は構文としては通り、`<` の左辺が `Bool` になったところで `TYP0003` になります（「`<`: 数値型が必要ですが、`Bool` が見つかりました」）。

`::` と `++` はパーサにはありますが、v0.1 の型検査・評価では実用対象外です。

A.2 レイアウト規則

ブロックはインデントで表します。行頭に使えるのは**スペース**だけで、タブが混じると 字句エラー `LXL0004`（インデントにタブは使えません）になります。caret はタブそのものを指すので、どこに紛れ込んだかがすぐ分かります。

```
error[LXL0004]: インデントにタブは使えません
--> tabs.lune:2:3
  |
2 |     1
  |     ^ インデントにはスペースを使う
    = hint: 行頭のタブをスペースに置き換えてください
```

ただし**空行とコメントのみの行はインデント判定の対象外**なので（下記の規則を参照）、そこにタブがあっても叱られません。文字列の中や、コードより後ろのタブも対象外です。

字句解析のあと、レイアウト処理が `NEWLINE` / `INDENT` / `DEDENT` を生成します。

- 行頭の空白幅 `n` を、インデントスタックの先頭 `top` と比べる。
- `n == top` → `NEWLINE` だけを出す。
- `n > top` → `INDENT` を出し、`n` を push する。
- `n < top` → `n` と一致する値が出るまで `DEDENT` を出す。一致しなければインデントエラー（`LAY0001`）。
- ファイル終端では、スタックが空になるまで `DEDENT` を出してから `EOF`。

括弧の中ではレイアウトトークンを作りません。（`[` `{` で深さ +1、`)` `]` で -1 とし、深さが 0 より大きい間、物理改行はただの空白として扱われます。だから括弧の中は自由に改行できます。

```
let xs = [  
    1,  
    2,  
]
```

`:` の直後にブロックを期待する構文（`if` / `while` / `for` / `match` / `record` など）では、次の有効トークンが `NEWLINE` `INDENT` でなければなりません。`=` の後は 1 行でもブロックでも書けます。

```
def f(): Int = 1  
  
def g(): Int =  
    1
```

REPL での複数行入力は、行末が `=` `:` `->` のいずれかなら継続に入り、空行で確定します（付録D）。

A.3 文法

以下では `NL` を 1 個以上の `NEWLINE` とします。完全な EBNF は `documents/LEXER_PARSER_SPEC.md` の §5～§10 にあり、ここはそこから引いた骨格です。

ファイルと宣言

```
file      ::= NL* module_decl? import_decl* top_decl* EOF
module_decl ::= "module" qualified_name NL+
import_decl ::= "import" qualified_name ("as" IDENT)? NL+
qualified_name ::= IDENT ("." IDENT)*

top_decl   ::= function_decl | type_decl | record_decl | let_decl |
var_decl

function_decl ::= "def" IDENT type_params? "(" params? ")" (":" type)? "="
body
let_decl      ::= ("strict" "let" | "let") pattern (":" type)? "=" body
var_decl      ::= "var" IDENT (":" type)? "=" body
type_decl     ::= "type" IDENT type_params? "=" NL INDENT constructor+
DEDENT
record_decl   ::= "record" IDENT type_params? ":" NL INDENT field+ DEDENT

constructor  ::= "|" IDENT "(" ctor_fields? ")"? NL
ctor_fields  ::= ctor_field ("," ctor_field)*
ctor_field   ::= ("strict" | "!=")? IDENT ":" type
field        ::= ("strict" | "!=")? IDENT ":" type NL
params       ::= param ("," param)*
param        ::= ("strict" | "!=")? IDENT ":" type
type_params  ::= "[" IDENT ("," IDENT)* "]"
body         ::= expr | NL INDENT block DEDENT
```

式

式は Pratt パーサで解析します。優先順位は A.1 の表のとおりです。

```
expr      ::= assign
assign    ::= pipeline (assign_op assign)?
primary   ::= literal | IDENT | "(" expr ")" | list | tuple
           | if_expr | match_expr | lambda | block_expr
           | "lazy" expr | "force" expr | "deepForce" expr
           | "seq" expr expr | "while" expr ":" block | "for" pattern "in"
expr ":" block
           | "IO" ":" block | "raise" expr

if_expr   ::= "if" expr ":" NL INDENT block DEDENT elif* else?
           | "if" expr "then" expr "else" expr
match_expr ::= "match" expr ":" NL INDENT match_case+ DEDENT
match_case ::= "|" pattern ("if" expr)? "->" body NL
lambda     ::= "fn" lambda_param* "->" body
list       ::= "[" (expr ("," expr)* ","?)? "]"
tuple      ::= "(" expr ("," expr)* ")"
```

`record` の構築は呼び出し構文の特別形で、**フィールド名が必須**です（`User(name = "Ada", age = 36)`）。関数と ADT のコンストラクタは位置引数で呼びます（名前付きにすると `TYP0012`）。

パターン

```
pattern ::= "_"                # ワイルドカード
         | "null"              # null
         | IDENT               # 名前束縛
         | literal             # リテラル
         | "(" pattern ("," pattern)+ ")"      # タプル
         | IDENT "(" pattern ("," pattern)* ")" # コンストラクタ
         | pattern "|" pattern  # or パターン
         | pattern ":" type     # 型注釈付き
```

`let` のパターンは**反駁不能**でなければなりません。 `let Some(x) = ...` は `TYP0008` です（`match` を使う）。

A.4 型

基本型

```
Bool  Int  Double  String  Unit  Any  Nothing  Null
```

`Int` は任意精度です。 `Nothing` は値を持たない型（`crash()` の戻り値型）。

これが基本型の全部です。 他の言語にありそうな `Long` `Float` `Char` は v0.1 にありません。整数は `Int` ひとつ（任意精度なので `Long` を分ける理由がない）、小数は `Double` ひとつ、文字は `String` です。型注釈に書くと、綴りを間違えた型名と同じ扱いで `TYP0003` になります。

複合型

<code>Option[Int]</code>	<code>Result[Int, String]</code>	<code>List[Int]</code>
<code>Lazy[Int]</code>	<code>IO[String]</code>	<code>Tuple[Int, String]</code>
<code>String?</code>		

`T?` は AST・型表現の上では `Nullable[T]` です。

関数型

カーリー化表記が正規形で、`->` は**右結合**です。

```
Int -> Int
Int -> Int -> Int
```

`(Int, Int) -> Int` は `Int -> Int -> Int` の糖衣として扱われます。タプル 1 個を受け取る関数は `Tuple[Int, Int] -> Int` と書きます。

null 安全

`null` と非 `null` の `T` はどちらも `T?` に渡せますが、**`null` を非 `null` の型に渡すことはできず、`T?` を `T` が要る位置でそのまま使うこともできません。**アンラップの手段:

手段	書き方
<code>match</code> の <code>null</code> パターン	<code> null -> ...</code> 。被覆後の名前は非 <code>null</code> に絞られる
<code>null</code> 合体	<code>a ?? b</code> (<code>a</code> が <code>null</code> なら <code>b</code> 。短絡する)
セーフナビゲーション	<code>x?.m</code> (<code>null</code> なら <code>null</code> に短絡。結果は <code>nullable</code>)
フロー・ナローイング	<code>if x != null:</code> の分岐で <code>x</code> が <code>T</code> になる

ナローイングが働くのは単純な `if x != null / if x == null` (`x` は変数) だけです。複合条件 (`&&` など)、`elif`、`while` でのナローイングと、`!!` 断言演算子はまだありません。

局所型推論

`def` の引数には型注釈が必要です。戻り値型は省略できます (ただし**再帰関数には必須** — TYP0011)。

未注釈のラムダ引数は文脈から型を受け取ります。文脈になるのは `let / var` の型注釈、`def` の戻り値注釈、関数呼び出しの引数位置です。

```
let inc: Int -> Int = fn x -> x + 1          # x : Int
let doubled = map([1, 2, 3], fn x -> x * 2)  # x : Int
```

文脈がないと `Any` に落ち、警告 `TYP0010` が出ます。

A.5 評価

Lune はデフォルト遅延です。

遅延されるもの

- 通常の `let` の右辺
- 通常の関数引数
- 通常のコンストラクタフィールド
- 通常の `record` フィールド
- `lazy expr` の `body`

評価が起きる境界

- 変数参照
- `force` / `deepForce`
- `seq` の第 1 引数
- 条件式 (`if`) の条件、`while` の条件
- 二項演算に必要な引数
- `match` の scrutinee の外側コンストラクタ
- リテラルパターンの比較
- 正格引数 (`!x: Int` / `strict x: Int`)
- `strict let` (`!let` も同じ)
- 正格コンストラクタフィールド、正格 `record` フィールド

メモ化

サンクは成功も失敗もメモ化されます。一度失敗したサンクを再び `force` すると、同じエラーが再び出ます（計算はやり直しません）。

再入評価は実行時エラーです（`RUN0005`）。`let x = x + 1` のように自分の結果に依存する値は、`force` した瞬間に自分に戻ってくるので、待たずにその場で報告されます（第 4 章 4.8）。

`let x = x + 1` を `--check` すると、報告されるのは `TYP0001`（未定義の名前 `x`）です。右辺の `x` は、その `let` がまだ環境に入る前に解決されるからです。`RUN0005` が出るのは `--eval` で実際に force したときで、同じコードが検査段階と実行段階で別の診断を出す例になっています。

観察の道具は `:thunks`（評価せずに状態を見る）と `:trace / --trace`（force の実況）です。第4章と付録B・D。

A.6 v0.1 で未対応のもの

キーワードとして予約されているのに、まだ動かないものがあります。何が返ってくるかを併記しました — 診断コードから引けるようにするためです。

書いたもの	出る診断
<code>class</code>	<code>PRS0001</code> 予期しないトークン（CLASS）
<code>interface</code>	<code>PRS0001</code> 予期しないトークン（INTERFACE）
<code>new Foo()</code>	<code>TYP0003</code> サポートされていない式（NewExpr）
<code>this / super</code>	<code>TYP0003</code> サポートされていない式（ThisExpr）
<code>try / catch / finally</code>	<code>PRS0001</code> 式が必要（TRY）
<code>record update r { x = 2 }</code>	<code>PRS0001</code> （LBRACE）
<code>record pattern R { x }</code>	<code>PRS0002</code> ARROW が必要（LBRACE）
<code>mutable record field var x: Int</code>	<code>PRS0002</code> IDENT が必要（VAR）
<code>annotation @inline</code>	<code>PRS0001</code> （AT）
<code>throws</code>	<code>PRS0002</code> ASSIGN が必要（THROWS）
<code>import a.b.{C, D}</code>	<code>PRS0002</code> IDENT が必要（LBRACE）
ADT のフィールドアクセス <code>c.v</code>	<code>TYP0003</code> サポートされていないメンバアクセス

予約すらされていないもの — `break / continue / return / export / use` はキーワードではないので、ただの識別子と見なされ `TYP0001`（未定義の名前）になります。ループを途中で抜きたいときは条件をループ条件に織り込みます（第9章 9.3）。

言語の外側で未対応なもの: JVM バイトコード生成、実 Java ライブラリ呼び出し、`Stream / Map / Set / Promise / Iterator`、パッケージマネージャ、LSP。

将来こうしたい、という話は `documents/LANGUAGE_FUTURE_SPEC.md` にあります。なぜ `v0.1` をこの範囲で切ったのかは付録E。

この付録の出どころ

節	出どころ
A.1 キーワード	<code>lune/tokens.py</code> の <code>KEYWORDS</code> から生成 (48 語)
A.1 演算子表	<code>documents/SYNTAX_SPEC.md</code> 14 節。中置部分は <code>lune/parser.py</code> の <code>INFIX</code> と照合済み
A.1 リテラル	各リテラルを REPL に通して型を確認
A.2 レイアウト	<code>documents/LEXER_PARSER_SPEC.md</code> §3
A.3 文法	同 §5～§10 の EBNF から骨格を抜粋
A.4 型	<code>documents/LANGUAGE_SPEC.md</code> §6・§8
A.5 評価	同 §17
A.6 未対応	同 §21。各項目を実際に <code>--check</code> に通し、出る診断を記録した

書きながら仕様書と実装が食い違っている箇所を2つ見つけ、どちらも解消しました。

`LXL0004` (タブのインデント) は**実装側を直しました** — 診断は用意されていたのに、インデント幅を先頭スペースだけで数えていたため一度も出せない状態でした。

文字リテラルの型は**仕様側を直しました**。 `LANGUAGE_SPEC.md` は `Char` を基本型として挙げていましたが、実装にはこれを生む式も文字を扱う関数もありません。 `v0.1` から `Char` を降ろし、将来仕様側の型としました。

付録B 標準ライブラリリファレンス

prelude（前奏曲）に入っている全 41 個の名前です。 `import` は要りません。どのファイルでも REPL でも、最初から見えています。

いま手元の処理系が何を持っているかは、REPL に聞くのがいちばん確かです。

```
lune> :env
```

各項目は「型シグネチャ / 一行の説明 / 例」で、リストを扱うものには**無限リストに使えるか**を付けました。Lune ではこれが実用上いちばん効く区別です（第8章）。

表の「無限」列 — ○ は `naturalsFrom(1)` のような終わりのないリストに使っても返ってくるもの、× は最後まで辿ろうとして返ってこないものです。この付録の ○× は推測ではなく、実際に無限リストへ適用して確かめた結果です。

型の `[T]` は型変数（どんな型でもよい）を表します。

B.1 Option — 「あるかもしれない」

名前	型	意味
<code>Some</code>	<code>[T] T -> Option[T]</code>	値がある形を作る
<code>None</code>	<code>Option[T]</code>	値がない形
<code>isSome</code>	<code>[T] Option[T] -> Bool</code>	<code>Some</code> かどうか
<code>isNone</code>	<code>[T] Option[T] -> Bool</code>	<code>None</code> かどうか
<code>getOrElse</code>	<code>[T] Option[T] -> T -> T</code>	中身を取り出す。 <code>None</code> なら第2引数
<code>optionMap</code>	<code>[T, U] Option[T] -> (T -> U) -> Option[U]</code>	中身にだけ関数を適用する

```
lune> Some(3)
Some(3) : Option[Int]
lune> getOrElse(Some(3), 0)
3 : Int
lune> getOrElse(None, 0)
0 : Int
lune> optionMap(Some(3), fn n: Int -> n * 2)
Some(6) : Option[Int]
```

`Option` は第5章、`null` との使い分けは第7章です。

B.2 Result — 「成功か失敗か」

名前	型	意味
<code>Ok</code>	<code>[T, E] T -> Result[T, E]</code>	成功の形を作る
<code>Err</code>	<code>[T, E] E -> Result[T, E]</code>	失敗の形を作る。中身に理由を入れられる
<code>isOk</code>	<code>[T, E] Result[T, E] -> Bool</code>	成功かどうか
<code>isErr</code>	<code>[T, E] Result[T, E] -> Bool</code>	失敗かどうか
<code>unwrapOr</code>	<code>[T, E] Result[T, E] -> T -> T</code>	成功なら中身、失敗なら第2引数
<code>resultMap</code>	<code>[T, U, E] Result[T, E] -> (T -> U) -> Result[U, E]</code>	成功のときだけ関数を適用する

```
lune> Err("boom")
Err("boom") : Result[T, String]
lune> unwrapOr(Ok(3), 0)
3 : Int
lune> unwrapOr(Err("boom"), 0)
0 : Int
lune> resultMap(Ok(3), fn n: Int -> n * 2)
Ok(6) : Result[Int, E]
```

`Option` と違い、`Err` は**失敗の理由を運べます**。第5章と、第13章の家計簿の例で使います。

B.3 List — 基本

名前	型	意味	無限
<code>Nil</code>	<code>List[T]</code>	空リスト	—
<code>Cons</code>	<code>[T] T -> List[T] -> List[T]</code>	先頭に1つ足す	○
<code>head</code>	<code>[T] List[T] -> Option[T]</code>	先頭。空なら <code>None</code>	○
<code>tail</code>	<code>[T] List[T] -> Option[List[T]]</code>	先頭を除いた残り。空なら <code>None</code>	○
<code>isEmpty</code>	<code>[T] List[T] -> Bool</code>	空かどうか	○
<code>length</code>	<code>Any -> Int</code>	要素数（文字列にも使える）	×
<code>range</code>	<code>Int -> Int -> List[Int]</code>	<code>range(a, b)</code> は <code>a</code> 以上 <code>b</code> 未満	—
<code>map</code>	<code>[T, U] List[T] -> (T -> U) -> List[U]</code>	各要素に関数を適用する	○
<code>filter</code>	<code>[T] List[T] -> (T -> Bool) -> List[T]</code>	条件を満たす要素だけ残す	○
<code>fold</code>	<code>[T, U] List[T] -> U -> (U -> T -> U) -> U</code>	初期値から畳み込む	×
<code>take</code>	<code>[T] List[T] -> Int -> List[T]</code>	先頭から <code>n</code> 個	○
<code>drop</code>	<code>[T] List[T] -> Int -> List[T]</code>	先頭 <code>n</code> 個を捨てる	○
<code>takeWhile</code>	<code>[T] List[T] -> (T -> Bool) -> List[T]</code>	条件が続く間だけ取る	○
<code>dropWhile</code>	<code>[T] List[T] -> (T -> Bool) -> List[T]</code>	条件が続く間だけ捨てる	○
<code>zip</code>	<code>[T, U] List[T] -> List[U] -> List[Tuple[T, U]]</code>	2本を組にする。短いほうで終わる	○
<code>zipWith</code>	<code>[T, U, V] List[T] -> List[U] -> (T -> U -> V) -> List[V]</code>	組にしながら関数を適用する	○

```

lune> let xs = [1, 2, 3]
ok
lune> head(xs)
Some(1) : Option[Int]
lune> tail(xs)
Some((2 3)) : Option[List[Int]]
lune> range(1, 5)
(1 2 3 4) : List[Int]
lune> map(xs, fn n: Int -> n * 2)
(2 4 6) : List[Int]
lune> filter(xs, fn n: Int -> n % 2 == 1)
(1 3) : List[Int]
lune> fold(xs, 0, fn a: Int n: Int -> a + n)
6 : Int
lune> zip(xs, ["a", "b"])
((1, "a") (2, "b")) : List[Tuple[Int, String]]
lune> zipWith(xs, xs, fn a: Int b: Int -> a * b)
(1 4 9) : List[Int]

```

length と fold だけが無限リストに使えません。 どちらも「最後まで行く」ことが仕事だからです。逆に言えば、それ以外は全部安全に使えます — `map` や `filter` を無限リストに掛けても、実際に計算されるのは後で取り出した分だけです（第4章・第8章）。

`range` は「無限」列が — ですが（リストを受け取らないため）、**返すリストは遅延**です。spine は消費された分しか作られないので、`take(range(1, 1000000000), 5)` の代価は `take(range(1, 11), 5)` と同じです。幅そのものには代価がありません（4.6節）。

範囲外や空に対しては、例外ではなく素直な値が返ります。

```

lune> take([1, 2, 3], 10)
(1 2 3) : List[Int]
lune> drop([1, 2, 3], 10)
() : List[Int]
lune> range(5, 1)
() : List[Int]
lune> head(Nil)
None : Option[T]

```

B.4 List — 終わりのないリストを作る

名前	型	意味
<code>naturalsFrom</code>	<code>Int -> List[Int]</code>	<code>n, n+1, n+2, ...</code> と続く
<code>iterate</code>	<code>[T] (T -> T) -> T -> List[T]</code>	初期値に関数を繰り返し適用し続ける
<code>repeat</code>	<code>[T] T -> List[T]</code>	同じ値が続く
<code>cycle</code>	<code>[T] List[T] -> List[T]</code>	与えたリストを繰り返し続ける

```
lune> take(naturalsFrom(1), 5)
(1 2 3 4 5) : List[Int]
lune> take(iterate(fn n: Int -> n * 2, 1), 5)
(1 2 4 8 16) : List[Int]
lune> take(repeat("x"), 3)
("x" "x" "x") : List[String]
lune> take(cycle([1, 2]), 5)
(1 2 1 2 1) : List[Int]
```

この4つはそれ自体が無限なので、必ず `take` や `takeWhile` で切って使います。第8章。

B.5 表示と出力

名前	型	意味
<code>show</code>	<code>Any -> String</code>	値を Lune の表示形式の文字列にする
<code>print</code>	<code>Any -> Unit</code>	表示する（改行なし）
<code>println</code>	<code>Any -> Unit</code>	表示して改行する

```
lune> show(42)
"42" : String
lune> show([1, 2])
"(1 2)" : String
lune> println("hi")
hi
() : Unit
```

`show` は文字列を返すだけで、画面には出しません。出すのは `print` / `println` です。`show("hi")` が `"\"hi\""` になるのは、文字列を表示形式にすると引用符が付くためです。

B.6 小さな道具

名前	型	意味
<code>id</code>	<code>[T] T -> T</code>	引数をそのまま返す
<code>const</code>	<code>[T, U] T -> U -> T</code>	第1引数を返し、第2引数は無視する
<code>not</code>	<code>Bool -> Bool</code>	真偽を反転する

```
lune> id(7)
7 : Int
lune> const(1, 2)
1 : Int
lune> not(true)
false : Bool
```

`const` の第2引数は遅延されたまま評価されません。第4章の「使わないものは計算しない」を1行で確かめられる関数です。

B.7 観察用のビルトイン

遅延評価の挙動を目で見るための道具で、**安定した標準 API ではありません**
(`STANDARD_LIBRARY_SPEC.md` 8.1 がそう明言しています)。教材とテスト用です。

名前	型	意味
<code>crash</code>	<code>() -> Nothing</code>	評価されると実行時エラーになる
<code>tick</code>	<code>() -> Int</code>	呼ばれるたびに内部カウンタを +1 して返す
<code>tickCount</code>	<code>() -> Int</code>	現在のカウンタ値。カウンタは増やさない

`crash` は「評価されなければ何も起きない」ことを示すために使います。`tick` は **何回評価されたか**を数えるためのもので、`let` が遅延することの証拠になります。

```
lune> let t1 = tick()
ok
lune> let t2 = tick()
ok
lune> tickCount()
0 : Int
```

2回 `tick()` を束縛したのにカウンタが `0` のままです。`let` は遅延するので、まだ一度も呼ばれていません。`strict let` にすると `2` になります。第4章。

B.8 seq と deepForce は関数ではありません

この2つは prelude の関数ではなく、**言語の構文**です（`lazy` や `force` と同じ仲間で、`:env` にも出てきません）。

```
seq a b          # a を評価してから b を返す
deepForce x      # x を中身まで残らず評価する
```

評価の順序と深さを制御する道具として、第4章 4.7 で扱います。

この付録の出どころ

型シグネチャは REPL の `:env`（＝処理系が持っている実物）から採りました。説明は `documents/STANDARD_LIBRARY_SPEC.md` に沿っています。「無限」列は、各関数を実際に `naturalsFrom(1)` へ適用して返ってくるかどうかを確かめた結果です。

付録C 診断コード一覧

Lune のコンパイラと評価器が発行する診断コードの全一覧です。本書はここで「地図」に徹します —— 各コードの詳しい解説（何が起きたか / 再現する最小の例 / 直し方）は本書に写さず、`lune explain CODE`（REPL では `:explain CODE`）に委ねます。同じ内容は [診断カタログ](#) と `documents/ERROR_INDEX_JA.md`（英語版は `ERROR_INDEX.md`）でも読めます。

```
$ lune explain TYP0007          # 1つのコードを読む
$ lune explain --index         # 全コードの解説を一覧する
```

コードの頭3文字は**族**で、そのままコンパイラの検査段階に対応します。族の意味と読み方は 第11章 11.2 節にあります。

「解説のないコードに出会う」ことは起きません。`tests/test_explanations.py` の `test_every_emitted_code_has_an_explanation` が、コンパイラが発行しうる全コードに解説があることをチェックしており、解説を書かずにコードを増やすとテストが落ちるようになっています。

表の読み方

- **本文で触れている章** — そのコードが本文に登場する章です。第11章は診断のしくみそのものを扱う章なので、ほとんどのコードに触れています。
- **動く例** — `books/examples/` にそのコードを実際に再現する検証済みの例がある章です。「自分で出してみたい」ときの入口として使ってください。– は、本書に狙って出す例を置いていないコードです（`LAY0002` や `REC0001` のように、意図的に出すにはひと工夫要るもの）。
- **警告（warning）** として出るものは意味の欄に「(警告)」と書いてあります。警告は `--check` を失敗させません。

一覧

コード	分類	意味	本文で触れている章	動く例
LAY0001	レイアウト	インデントが揃っていない	第 11 章	—
LAY0002	レイアウト	対応しない閉じ括弧	第 11 章	—
LXL0001	字句	予期しない文字	第 2、11 章	2
LXL0002	字句	文字列・文字リテラルが閉じていない	第 11 章	—
LXL0003	字句	ブロックコメントが閉じていない	第 11 章	—
LXL0004	字句	インデントにタブは使えない	第 11 章	—
MOD0001	モジュール解決	モジュールが見つからない・読めない	第 10、11 章	10
MOD0002	モジュール解決	モジュールの循環 import	第 10、11 章	10
MOD0003	モジュール解決	module 宣言の不一致	第 10、11 章	10
PRS0001	構文解析	予期しないトークン	第 2、6、11 章	2
PRS0002	構文解析	特定のトークンが必要	第 10、11 章	—
REC0001	レコード検査	レコードフィールドの重複宣言	第 11 章	—
REC0002	レコード検査	存在しないレコードフィールド	第 6、11 章	6
REC0003	レコード検査	レコードフィールドの不足	第 6、11 章	—

コード	分類	意味	本文で触れている章	動く例
REC0004	レコード検査	初期化でフィールドを二重指定	第 6、11 章	—
REC0005	レコード検査	宣言されていないフィールドの指定	第 6、11 章	—
REC0006	レコード検査	レコードは名前付きで構築する	第 6、11、13 章	—
RUN0005	実行時	再帰的なサンク評価	第 3、4、10、11、12 章	4
RUN0006	実行時	実行時エラー	第 2、4、6、9、11、12 章	4
TYP0001	型検査	未定義の名前	第 1、4、10、11、12 章	1、4、11
TYP0003	型検査	型の不一致	第 1、2、7、8、11、12 章	2、7
TYP0004	型検査	呼び出せない値	第 3、11 章	—
TYP0005	型検査	引数の個数が違う	第 1、3、11 章	1
TYP0006	型検査	for の対象は List でなければならない	第 9、11 章	9
TYP0007	型検査	網羅的でない match	第 5、7、11、13 章	5、7、11
TYP0008	型検査	束縛に反駁可能パターンは使えない	第 5、9、11 章	5
TYP0009	型検査	到達しない match ケース（警告）	第 5、7、11、12 章	5
TYP0010	型検査	引数の型を推論できない（警告）	第 3、11 章	—
TYP0011	型検査	再帰関数には戻り値型が必要	第 2、3、11 章	3
TYP0012	型検査	ここでは名前付き引数を使えない	第 11、13 章	—
TYP0013	型検査	不変な束縛への代入	第 9、11 章	9

番号の飛びについて

番号が連番になっていないことに気づいたかもしれません。 `TYP0002` と `RUN0001` ～ `RUN0004` が欠けています。

`RUN` 系は、設計段階では実行時エラーを種類ごとに分ける予定でした（`RUN0004` は「実行時の非網羅 match」用に用意されていました）。実装ではその方針をやめ、再帰サングの `force` だけを専用の `RUN0005` に残して、残りは汎用の `RUN0006` に集約しています。未定義変数も、失敗したサングの `force` も、標準ライブラリの型不一致も、実行時の非網羅 match も、みな `RUN0006` です。`TYP0002` は単なる欠番です。詳しくは `documents/ERROR_DIAGNOSTICS_SPEC.md` を参照してください。

欠番を埋め直すことはしません。 一度公開した番号が後から別の意味になるほうが、飛びがあるより読者にとって害が大きいからです。エラー番号は調べるための鍵であって、通し番号ではありません。

この付録の出どころ

この表の「コード」「分類」「意味」は `documents/ERROR_INDEX_JA.md` から作っています。その索引自体も手書きではなく、コンパイラのカタログから生成されたものです。

```
$ ./bin/lune explain --index --lang ja > documents/ERROR_INDEX_JA.md
```

つまりこの表の出どころは `lune/explanations_ja.py` のカタログであり、コンパイラの実装と同じ一次情報です。コードを追加・変更したときは、索引を作り直してからこの表を更新してください。

付録D CLI・REPL コマンドリファレンス

`lune` コマンドと REPL の全機能の一覧です。使い方の物語は第11章（診断）と第12章（道具）にあります。ここは「あの引数、何だったっけ」を引くための場所です。

本書の規約どおり、`lune` はリポジトリの `./bin/lune` を指し、`export LUNE_LANG=ja` が設定してあるものとします（第1章 1.1）。

D.1 起動とモード

```
$ lune [--lang en|ja] [--module-path DIR]... [MODE] [FILE]
```

モードは以下のいずれかで、**同時にはひとつだけ**指定します。

指定	何をするか
<code>lune</code>	引数なしで REPL を起動する
<code>lune --repl</code>	明示的に REPL を起動する
<code>lune --check FILE</code>	型検査だけして終わる。評価しない
<code>lune --eval NAME FILE</code>	ファイルを読み、トップレベル束縛 <code>NAME</code> を評価して表示する
<code>lune --eval NAME --trace FILE</code>	上に加えて、 <code>force</code> の実況を <code>stderr</code> に流す（第4章）
<code>lune --tokens FILE</code>	レイアウト処理後のトークン列を表示する（第12章）
<code>lune FILE</code>	モードを付けないと、構文木（AST）をそのまま印字する

最後の行は開発者向けの既定動作です。人が読むためのものではないので、ふだんは `--check` か `--eval` を付けてください。

`--eval` は型検査をせずに評価だけを行います。型の合わないコードを `--eval` で動かすと、演算子や条件の型は実行時に検査され、`lune --check` を促す hint 付きの `RUN0006` になります。まず `--check` を通す習慣をつけておくのが安全です。

D.2 グローバルオプション

オプション	効果
<code>--lang en ja</code>	診断の言語。 <code>--lang=ja</code> の形でも書ける
<code>--module-path DIR</code>	モジュールの探索ルートを追加する。 繰り返し指定できる (第10章)
<code>--version / -v</code>	版番号を表示して終わる
<code>-h / --help</code>	使い方を表示する

言語は「環境変数 `LUNE_LANG` を既定とし、`--lang` フラグがそれに優先する」という関係です。

```
$ export LUNE_LANG=ja           # 以後ずっと日本語
$ lune --lang en --check f.lune # この1回だけ英語
```

注意 — `--lang` は `lune --help` の一覧に**出てきません**。他のオプションより手前で処理される作りになっているためです。存在は忘れないでください。

版番号は、バグ報告のときに書き添えると話が早く済みます。

```
$ lune --version
lune 0.1.0
```

REPL の起動時に出る一行にも同じ番号が入っています (`Lune v0.1.0 REPL.`)。本書が対象としているのは v0.1 系です。

D.3 サブコマンド

`lune explain`

```
$ lune explain CODE [--lang en|ja]
$ lune explain --index [--lang en|ja]
```

診断コードの解説を表示します。 `--index` は全コードの解説をまとめて出力するので、`documents/ERROR_INDEX_JA.md` はこれをファイルに落としたものです (付録C)。

知らないコードを渡すと、渡せるコードの一覧を教えてください。

lune fmt / lune fix

```
$ lune fmt [--write|--check] FILE...
$ lune fix [--write|--check] FILE...
```

fmt は整形、fix は診断の hint に基づく自動修正です（第11章・第12章）。両者はフラグの体系が同じです。

フラグ	動き
なし	結果を標準出力に書く。ファイルはちょうど1つでなければならない
--write / -w	ファイルを直接書き換える。複数ファイル可
--check	書き換えず、変更が必要なら終了コード 1 で知らせる。CI 向け

--write と --check は同時に使えません。

fmt にはひとつ制限があります。### で囲むブロックコメントを含むファイルは整形できません（第12章 12.3）。

D.4 REPL コマンド

:help で出る一覧がすべてです。

コマンド	何をするか
:help	この一覧を出す
:quit / :q	終了する
:env	いま見えている名前と型を全部並べる
:type NAME	名前の型を答える
:thunks [NAME]	束縛の評価状態を、評価せずに見せる（第4章）
:trace [on off]	force の実況中継を切り替える（第4章）
:lang [en ja]	診断の言語を切り替える
:explain CODE [en ja]	診断コードの解説を読む（第11章）

引数を省いた `:thunks` は全束縛の状態を並べます。

複数行の入力は、行末が `=` `:` `->` のいずれかで終わると継続に入り、空行で確定します。端末で起動していれば行編集と履歴（上下矢印）が効き、履歴は `~/.lune_history` に残ります。

`import` も打てます（第10章）。探索は**起動したディレクトリ**が基点で、`--module-path` を 足すこともできます。読み込んだモジュールの名前はセッションに残るので、`:env` に 並びます。

```
$ lune --repl
lune> import math
ok
lune> add(1, 2)
3 : Int
```

D.5 終了コード

スクリプトや CI から呼ぶときのために。

コード	意味
0	成功。警告だけなら 0 です — <code>--check</code> は警告では失敗しません
1	検査や評価が失敗した。診断が出ている。 <code>fmt --check / fix --check</code> の「要変更」もこれ
2	使い方の誤り。知らないフラグ、引数の不足、 <code>--write</code> と <code>--check</code> の同時指定など

「診断が出た」（1）と「コマンドの呼び方を間違えた」（2）が分かれているのが要点です。 `--check` が落ちたときに、それがコードの問題なのかスクリプトの書き間違いなのかを、出力を読まずに判別できます。

付録E Lune の設計と、これから

本文はずっと「どう書くか」の話でした。この付録は「なぜそう決めたか」の話です。言語の設計には、たいてい引き受けた不便があります。それを隠さずに置いておくのが、この本のいちばん最後の仕事です。

E.1 なぜデフォルト遅延なのか

遅延評価は初学者の罠だ、というのは正しい批判です。いつ計算が走るのか見えないと、プログラムの動きが読めません。副作用の順番も直感を外れます。教育向けの言語（Racket、Pyret、Hedy）がすべて正格なのは、偶然ではありません。

それでも Lune はデフォルト遅延を選び、そしてやめないと決めています。理由は「遅延が便利だから」ではありません。

批判への答えが「隠す」ではなく「見えるようにする」だったからです。

隠す道		Lune が選んだ道
評価のタイミング	見えないので考えない	<code>:thunks</code> で状態を、 <code>:trace</code> で瞬間を見る
循環した定義	固まる（Haskell の <code><<loop>></code> ）	<code>RUN0005</code> を即座に、解説付きで報告する
正格にしたとき	言語が勝手に決める	<code>strict</code> / <code>!</code> / <code>seq</code> / <code>force</code> / <code>deepForce</code> で自分が決める

第4章がまるごとこの答えです。「信じる必要はありません。すべてその場で観察できます」と書いたのは、**可観測性が設計の代償ではなく設計の中身**だからです。

ここには狙いがもう一つあります。正格な言語で「評価戦略」を教えるのは難しい — 教える対象がプログラムの外側にあるからです。デフォルト遅延の言語では、評価戦略が**プログラムの内側の観察対象**になります。`tick()` を2回呼んでカウンタが0のままなのを見た瞬間（付録B）、遅延は概念ではなく現象になります。

設計の緊張 — 「初学者に優しい」と「評価戦略を教えられる」は、普通は両立しません。Lune はこの緊張を、診断と可観測化で解こうとしている言語です。うまく解けているかは、あなたが第4章を読み終えたときの感触が答えです。詳しい戦略は `documents/STRATEGY.md` の施策 D・E にあります。

E.2 なぜ null と Option の両方があるのか

第5章で `Option[T]`、第7章で `T?` を学びました。「ないかもしれない」を表す道具が2つある、というのは設計としては贅肉に見えます。

そう見えて、どちらも消せませんでした。

Option は消せません。 これは言語機能ではなく、ADT とパターンマッチがあれば **誰でも書ける型** です。実際 prelude の中で `type Option[T] = | Some(value: T) | None` として定義されているだけで、コンパイラは特別扱いしていません。ADT を持つ言語から `Option` を取り上げることはできない—— 禁止しない限り誰かが定義します。

T? も消せません。 Lune の長期的な目標には JVM 上で動き、Java のライブラリを直接呼ぶことが入っています (documents/LANGUAGE_FUTURE_SPEC.md)。そして Java の参照はすべて null になり得ます。 `T?` は将来仕様の Java 型対応表で **nullable reference** に対応する型で、つまり **外の世界と話すための型** です。null のない言語は、null のある世界と話せません。

だから Lune は **null を廃止せず、型で飼いならす** 道を選びました (第7章)。「null を消した言語」ではなく「null が入りうる場所を型に書かせ、確かめる前に使わせない言語」です。

役割は自然に分かれます。

	T?	Option[T]
出自	言語機能 (専用構文 <code>??</code> <code>?.</code> と絞り込み)	prelude の ADT
向いている場所	データの形。レコードのフィールド、関数の引数と戻り値	リスト処理と高階関数のパイプライン
外との境界	Java の nullable 参照に対応する	Lune の内側の話

実用の目安は第7章 7.7 節にあります。境界では `match` で詰め替えれば行き来できます。

正直に言うと — 2つあることの代償は、読者が使い分けを覚えなければならない、という一点です。これは実際に負担です。7.7 節に判断の目安を置いたのは、その負担をせめて「表を見れば済む」大きさに抑えるためです。

E.3 v0.1 が引き受けた制限

何を作らなかったかは、何を作ったかと同じくらい設計です。使えないものの一覧と、出る診断は付録AのA.6にあります。ここでは**なぜ**の側を書きます。

凍結したもの — JVM と OO

`class` / `interface` / `extends`、Java ライブラリの直接呼び出し、JVM バイトコード生成。これらは将来仕様（`LANGUAGE_FUTURE_SPEC.md` の Phase 2～4）にありますが、**当面凍結**という判断が明示されています（`documents/STRATEGY.md` §5）。

理由は能力ではなく戦線の選択です。Java 相互運用は「実務で採用してもらう」ための施策で、そこは勝ち目の薄い戦場です。一方、教育という用途には Python インタプリタの速度で足ります。だから Lune は「実務言語になる」道を降りて、「**母語で教えてくれるコンパイラを持つ関数型入門言語**」に集中しています。

`class` や `new` を書くと構文エラーになるのに、キーワードとしては予約されている — という中途半端な状態は、この凍結の跡です。将来の席は空けたまま、今は座らせていません。

まだ無いもの — try / catch

失敗を例外で飛ばす道具がありません。これは凍結というより**順番**の話です。第5章の `Result`、第7章の `T?` という「**失敗を値として返す**」道具を先に据えたので、例外はそれらと役割が重なります。第9章で「回復可能な失敗は値で表してください、`raise` の順番はプログラムを止めるのが正しい場面だけです」と書いたのは、この設計の帰結です。

小さく欠けているもの

`break` / `continue`（ループ条件に織り込む — 9.3 節）、record update と record pattern、mutable な record フィールド、ADT のフィールドアクセス、`Stream` / `Map` / `Set`。文字型ありません（付録AのA.1）。

このあたりは思想ではなく、**まだ手が回っていないだけのものが混じっています**。付録AのA.6が「何が起きるか」の一覧、`LANGUAGE_FUTURE_SPEC.md`が「どうしたいか」の一覧です。

E.4 処理系を読みたくなったら

Lune の処理系は `lune/` 以下の Pure Python、約 7,400 行、**外部依存なし**です。1日で読み切れる規模なので、言語の実装を初めて読む題材に向いています。

ソースは検査の順番に並んでいます。付録AのA.1で見た診断コードの族（`LXL` → `LAY` → `PRS` → `MOD` → `TYP` → `RUN`）が、そのままファイルの並びです。

ファイル	行数	役割
<code>lexer.py</code>	294	文字列 → トークン列。 <code>LXL</code>
<code>layout.py</code>	61	インデント → <code>INDENT</code> / <code>DEDENT</code> 。 <code>LAY</code>
<code>tokens.py</code>	214	トークン種別とキーワード表
<code>parser.py</code>	763	トークン列 → AST（Pratt パーサ）。 <code>PRS</code>
<code>nodes.py</code>	405	AST の定義。ここを読むと言語の全体像が分かる
<code>module_loader.py</code>	188	<code>import</code> の解決。 <code>MOD</code>
<code>typechecker.py</code>	1591	型検査と網羅性。 <code>TYP</code> ・ <code>REC</code> 。最大のファイル
<code>evaluator.py</code>	1259	遅延評価とサンク。 <code>RUN</code>
<code>diagnostics.py</code>	124	診断の表示形式（ <code>caret</code> や <code>hint</code> の組み立て）
<code>messages.py</code>	332	日英メッセージカタログ
<code>explanations.py</code> / <code>explanations_ja.py</code>	530 / 428	<code>lune explain</code> の教材カタログ
<code>formatter.py</code> / <code>fixer.py</code>	490 / 66	<code>lune fmt</code> / <code>lune fix</code>
<code>repl.py</code>	347	REPL。Playground もこの <code>ReplSession</code> を駆動している
<code>cli.py</code>	269	コマンドライン入口

読む順のおすすめ: `nodes.py` → `lexer.py` → `parser.py` → `evaluator.py`。最初にASTを見ておくと、あとは全部「ASTをどう作るか/どう扱うか」の話になります。遅延評価に興味があるなら `evaluator.py` の `Thunk` と `force_value` から入るのが最短です。

仕様書の読み方

`documents/` に仕様書が並んでいます。 `LANGUAGE_SPEC.md` が **v0.1 の正**で、そこから各論に降りる構造です。

知りたいこと	読む文書
v0.1 で何が使えるか	<code>LANGUAGE_SPEC.md</code> （これが正）
表面構文・EBNF	<code>SYNTAX_SPEC.md</code> / <code>LEXER_PARSER_SPEC.md</code>
遅延評価の規則	<code>LAZY_EVALUATION_SPEC.md</code>
型検査の範囲	<code>TYPE_CHECKER_SPEC.md</code> / <code>MATCH_EXHAUSTIVENESS_SPEC.md</code>
診断の設計	<code>ERROR_DIAGNOSTICS_SPEC.md</code> / <code>ERROR_INDEX_JA.md</code>
どこへ向かうのか	<code>LANGUAGE_FUTURE_SPEC.md</code> / <code>STRATEGY.md</code>

仕様書を読むときの注意 — 仕様書は「正」とされていますが、v0.1 の実装が追いついていない箇所や、逆に実装のほうが正しい箇所があります。この本の付録Aを書く途中でも2件見つかりました（`LXL0004` が出せなかった件と、文字型の件）。**疑ったら `./bin/lune` に通して確かめてください**。それが一次情報です。この本の付録は全部 そうやって書きました。

E.5 これから

`documents/STRATEGY.md` が向かう先を書いています。要約すると、汎用普及は狙わず「母語で教えてくれるコンパイラを持つ関数型入門言語」に集中し、JVM / OO 路線は凍結する。デフォルト遅延は隠すべき弱点ではなく、前面に出す差別化点として扱う。

この本を読み終えたあなたにできることが、いくつかあります。

- **つまづいた場所を教えてください**。どの診断で詰まったか、どの説明が足りなかったかは、そのまま言語の改善点です。診断が不親切だったなら、それは言語のバグだと考えています。
- **診断を増やす**。汎用の `RUN0006` に集約されている実行時エラーには、専用コードに昇格させる余地があります（`STRATEGY.md` 施策 E の作業項目）。
- **書いてみる**。v0.1 でも第13章のようなプログラムは書けます。足りない道具に気づいたら、それが次に作るものです。

部訓をもう一度。**必要になるまで、やらない。一度やったことは、忘れない。** 遅延評価とメモ化のことですが、言語の作り方の話でもあります。

索引

本文と付録に現れる主な用語です。ページ番号は PDF 版のものです（HTML 版では上の検索が使えます）。

値と型

- Int（任意精度） ... 18, 26, 137, 141
- Double ... 11, 15, 16, 18, 19, 20, 24, 26...
- String ... 7, 8, 9, 15, 18, 19, 20, 21
- Bool ... 13, 18, 20, 21, 22, 26, 28, 41...
- Unit ... 9, 18, 19, 26, 94, 95, 96, 97...
- Nothing ... 141, 151
- タプル ... 11, 14, 20, 21, 24, 26, 27, 28...
- 型注釈 ... 23, 26, 30, 35, 59, 141, 142
- 局所型推論 ... 59, 60, 61, 64, 77, 79, 82, 88...
- 型変数 ... 146

遅延評価

- 遅延評価 ... 5, 8, 10, 14, 21, 32, 38, 42...
- サンク ... 32, 38, 39, 40, 41, 42, 45, 48...
- メモ化 ... 42, 50, 143, 166
- 正格 ... 45, 49, 50, 51, 52, 98, 143, 151
- force ... 39, 40, 41, 43, 45, 46, 47, 48...
- deepForce ... 45, 46, 50, 130, 131, 132, 133, 135...
- seq ... 45, 46, 49, 50, 130, 135, 136, 140...
- 無限リスト ... 14, 45, 84, 85, 86, 87, 89, 90...
- ストリーム ... 5, 83

関数

- 部分適用 ... 13, 15, 17, 32, 33, 35, 37, 123...
- カリー化 ... 142
- 高階関数 ... 31, 35, 37, 78, 79, 162
- ラムダ ... 10, 14, 24, 30, 31, 35, 36, 37...
- パイプライン ... 13, 21, 32, 33, 35, 71, 78, 121...
- 再帰 ... 24, 29, 33, 34, 48, 142, 155

データと分岐

- 代数的データ型 ... 5, 14, 20, 53, 54, 55, 58, 61...

- パターンマッチ ... 14, 53, 54, 162
- 網羅性 ... 56, 57, 61, 62, 74, 75, 82, 109...
- 反駁不能パターン ... 57, 58, 64, 94, 95, 97, 108, 141...
- レコード ... 5, 24, 66, 67, 68, 69, 70, 71...
- Option ... 56, 58, 59, 60, 61, 63, 64, 68...
- Result ... 58, 59, 60, 61, 62, 63, 64, 96...

null 安全

- null 安全 ... 5, 14, 19, 73, 82, 142
- ナローイング ... 73, 74, 75, 76, 79, 82, 132, 134...
- null 合体演算子 ... 21, 73, 75, 76, 77, 78, 79, 80...
- セーフナビゲーション ... 73, 75, 76, 78, 79, 81, 138, 142...

演算子

- 床除算 ... 19, 20, 26, 93, 129, 130, 133, 134...
- 複合代入 ... 93
- 短絡評価 ... 21, 42, 48, 51, 76, 79, 80, 142

命令的な機能

- var ... 92, 93, 94, 97, 98, 99, 142, 144
- while ... 92, 94, 97, 98, 99, 136, 139, 140...
- for ... 92, 94, 95, 96, 97, 98, 99, 110...
- IO ... 92, 95, 96, 97, 98, 99
- モジュール ... 8, 14, 100, 101, 102, 103, 104, 106...

道具と診断

- 診断コード ... 4, 5, 12, 14, 25, 48, 108, 110...
- explain ... 3, 4, 12, 13, 14, 15, 16, 17
- fmt ... 23, 117, 119, 120, 121, 122, 124, 131...
- fix ... 13, 14, 110, 111, 114, 120, 121, 159...
- REPL ... 4, 5, 6, 7, 8, 9, 13, 14...
- :thunks ... 5, 8, 14, 38, 39, 45, 46, 49...
- :trace ... 5, 38, 39, 40, 49, 50, 52, 117...
- Playground ... 4, 164

付録B（標準ライブラリ）・付録C（診断コード）・付録D（CLI と REPL）は、それぞれの一覧そのものが索引として使えます。