

付録

「練習問題」
「この章の理解度チェック」
解答

第1章の解答

この章の理解度チェック P.047

1. Rustは、ソースコードをネイティブコードにコンパイルして、OSで直接実行可能な高速、高効率なプログラムを作成できる。高効率な言語にありがちなメモリ利用における問題点を克服しているメモリ安全な言語でもある。優れた並列性を備えており、安全にマルチスレッドで動作するプログラムを作成できる。
2. ① コメント（ラインコメントも可） ② 関数
③ 文（マクロ呼び出し文も可） ④ 変数
⑤ プレースホルダー
3. main関数で、エントリーポイントと呼ばれます。
4. ;（セミコロン）
5. 以下の3つがコメントとなります。
 - ラインコメント（//）：単一行で//以降をコメントと見なす
 - ブロックコメント（/* ~ */）：囲まれた範囲をコメントと見なす。複数行にわたって可
 - ドキュメンテーションコメント（///）：ドキュメントを生成するための特別なコメント

第2章の解答

練習問題2.1 P.062

1. ① 正しい。
② 正しい。変数名はアンダースコアではじめることができます。ただしRustでは先頭がアンダースコアである変数は特別な意味を持つので、通常は使いません。
③ 誤り。変数名を数字ではじめることはできません。
④ 誤り。予約語に完全に一致する単語は変数名になりません。
⑤ 正しい。単語間をアンダースコアで区切るスネークケース記法は変数名として一般的に使われます。

- ⑥ 正しい。typeは予約語ですが「r#」を前置することで「r#type」という変数名として使うことができます。
⑦ 誤り。アンダースコア以外の記号を含むことはできません。
⑧ 正しい。VARのような全て大文字の変数名も文法上は可能ですが、定数と紛らわしくなるので通常は使うべきではありません。

練習問題2.2 P.072

1. 以下から、それぞれ2個以上挙げていれば正解です。
 - 整数型（符号あり）：i32、i64、i8、i16、i128
 - 整数型（符号なし）：u32、u64、u8、u16、u128
 - 浮動小数点型：f32、f64（加えてf16、f128でも可）
 - その他の型：bool、str、(), !

練習問題2.3 P.078

1. 以下のようなリテラルを表現できていれば正解です。
 - ① 0xf8：プレフィクスが「0x」で、移項の数値が0～9、a～f、A～Fであること。
 - ② 987_654：アンダースコアを用いて、基本的に3桁ごとに区切る。先頭のアンダースコアは不可（末尾は可）。
 - ③ "こんにちは！\nやまうちさん。": エスケープ文字の\nが改行となる。生文字列リテラルのr#"~"#を用いても可。
 - ④ 1.41421356e-10：<仮数部>e<符号><指数部>の形式。eはEでも可。
 - ⑤ '漢': シングルクォート（'）で単一の文字をくくる。

練習問題2.5 P.092

1. 変数num_i64はi64型です。i32型である変数num_i32にはi64型の値を代入できません。以下が正しいコードです。as演算子を用いてi32型に型強制します。

```
let num_i64 = 100_i64;
let num_i32: i32 = num_i64 as i32;
あるいは
let num_i32: i32 = num_i64.try_into().unwrap();
```

2. 例えば以下のように記述します。

```
let num = i32::from_str("123").unwrap();
```

この章の理解度チェック P.092

1. 以下のポイントを挙げられれば正解です。

- main 関数定義にはfn キーワードが必要です
- 文字列リテラルはシングルクォート (') ではなくダブルクォート (") でくります
- 文の末尾にはセミコロン (;) が必要です

以上を修正したコードは以下の通りです。

▶リストA.1 practice1.rs

```
fn main() {
    let msg = "こんにちは、世界！";
    println!("{}", msg);
}
```

2. ① let ② mut ③ 初期化
④ データ型 (型でも可) ⑤ 型推論

変数はlet文で宣言し、ミュータブルな変数にはmut キーワードを付加します。変数の利用には初期化が必要です。変数にはデータ型が必要ですが、決定は基本的に型推論に任せます。

3. ① const ② 0.9 ③ net_price

定数の定義は、const キーワードからはじめます。値引率は10%であるので、正味価格は0.9を乗じたものになります。

4. (×) 浮動小数点型は、符号あり型と符号なし型に分類できる。⇒浮動小数点型は符号あり型です。符号なし型の区別があるのは整数型です。
(×) 文字列リテラルは、シングルクォートまたはダブルクォートでくくる。⇒文字列リテラルはダブルクォートでくります。シングルクォートで囲むのは文字リテラルです。
(○) 正しい記述です。
(×) 値範囲の小さな型から値範囲の大きな型へは暗黙的に変換される。⇒異なる値範囲の型は相互に変換できません。変換したい場合には型強制(型キャスト)を使用します。
(○) 正しい記述です。

5. ① let pi = 3.14; ② println!("{}", name);
③ let data = [[0; 5]; 4]; ④ let trio: (char, i32, f64);

型推論を使う場合、期待するデータ型に推論されるように初期値を記述します。f64型に推論したいなら3.14や1.0のように必ず小数点リテラルを記述します。配列の初期値指定の記法は忘れやすいのでよく押さえておきましょう。配列自身を配列の要素とすることで、多次元配列を実現できます。

第3章の解答

練習問題3.1 P.105

1. ① 誤り。算術演算子には、加算(+) 減算(-)、乗算(*)、除算(/)に剰余算(%)を加えた5種類がある。
② 誤り。他言語の一部で使えるインクリメント演算子/デクリメント演算子と呼ばれる単項演算子(++、--)はRustにおいては使えない。
③ 正しい。例えば加算では「1 + 1」のような整数型同士の演算は認められるが、「0.5 * 3」のような浮動小数点数型と整数型の演算は認められない。
④ 正しい。オーバーフローと桁あふれは同じ意味である。
⑤ 誤り。算術単項演算子は「-」のみ。「+」はコンパイルエラーとなる。

2. ① 7 ② エラー ③ エラー ④ 1.0 ⑤ 1.0

②のような、異なるデータ型同士の算術演算はできません。

③のように、ゼロで除算することが分かっている場合もコンパイル時にエラーとなります。

練習問題3.3 P.114

1. ① 誤り。等しいことを表す比較演算子は「==」と「=」を重ねます（「=」のみは代入演算子）。等しくないことを表す比較演算子は「!=」です。
- ② 誤り。配列やタプルといったシーケンス型も比較演算子の対象となります。
- ③ 正しい。比較演算子の演算結果は論理値型となります。
- ④ 正しい。辞書比較が適用されます。
- ⑤ 誤り。シーケンスの要素数が一致していなくても個々の要素の大小によって最終的な大小を決定します。
2. ① false ② true ③ false ④ true ⑤ false

⑤は一見trueになるように見えますが、非数と非数が等しいという定義はなく、falseとなります。

練習問題3.4 P.123

1. ① 正しい。算術演算と異なり、あくまでもビット単位での演算を実行するのがビット論理演算。
- ② 誤り。浮動小数点型は論理演算の対象にならない。
- ③ 誤り。「true」か「false」に評価されるのはオペランドが論理値型である場合のみ。整数型では整数型の結果となる。
- ④ 正しい。論理演算には、ビット論理演算とビットシフト演算がある。
- ⑤ 誤り。論理和はOR演算と呼ばれ、論理積はAND演算と呼ばれる。

2. ① $4 \& 4 = 0b0100 \& 0b0100 = 0b0100 = 4$
- ② $10 \mid 20 = 0b01010 \mid 0b10100 = 0b11110 = 30$
- ③ false
- ④ $20 \ll 3 = 0b00010100 \ll 3 = 0b10100000 = -96$
(符号なしと見なして160でも可)
- ⑤ エラー

⑤はシフト数が負数なので、コンパイル時か実行時にオーバーフローとなる。

練習問題3.5 P.126

1. ① 評価される。「2 == 2」がtrueになるので。
- ② 評価されない。「1 > 0」がtrueになるので。
- ③ 評価されない。「2 != 2」がfalseになるので。
- ④ 評価される。「0 != 0」がfalseになるので。

この章の理解度チェック P.129

1. ① 算術演算子（算術二項演算子でも可）
- ② 代入演算子、複合代入演算子
- ③ <、>、>=、<=から3個以上
- ④ 論理演算子
- ⑤ &&、||
2. a = 6、b = -1、c = 0、d = 1
3. 3行目でbをaで割った剰余を計算しているが、aが0であるので「ゼロで除算」エラーとなります。以下のように修正すれば正しく動作します。

```
…略…
if x != 0 && y % x == 0 {
    println!("{x} は {y} で割り切れます。");
}
```

4. ① 優先順位 ② 結合規則 ③ 高い ④ 同じ

第4章の解答

練習問題4.1 P.144

1. 以下のようなコードを書けていれば正解です。if式で複雑な条件分岐を記述する場合には、条件式を記述する順番に注意する必要があります。ifを式として使わず、個々のブロックで直接println!で表示してしまっても構いません。

▶リストA.2 practice-if.rs

```
let score = 75;
let s = if score >= 90 {
    "秀"
} else if score >= 80 {
    "優"
} else if score >= 70 {
    "良"
} else if score >= 60 {
    "可"
} else {
    "不可"
};
println!("{}", s); // 結果: 良
```

2. ベン図については、P.●○の図4.2を参照してください。このような置き換えを、ド・モルガンの法則と言います。この法則を忘れてしまったとしても、ベン図を理解しておけば自分で条件式を置き換えることができますでしょう。

練習問題4.2 P.151

1. while式は条件式で有限回の繰り返しを行い、for式はコレクションの全要素に対して繰り返しを行います。loop式は条件式を持たず、基本的に無限ループとなります。
2. 以下のようなコードが書けていれば正解です。

▶リストA.3 practice-while.rs

```
let mut i = 1;
while i < 10 {
    let mut j = 1;
    while j < 10 {
        let result = i * j;
        println!("{}", result);
        j += 1;
    }
    println!();
    i += 1;
}
```

このように、while式は入れ子が可能です。その場合、カウンター変数の名前は、それぞれで異なるものを使用する必要があることに注意してください。

練習問題4.3 P.157

1. continue式、break式
2. 以下のようなコードが書けていれば正解です。ただし、loop式の多用は終了条件の見通しが悪くなるので、できるだけ避けるべきです。

▶リストA.4 practice-loop.rs

```
let mut i = 1;
'vertical': loop {
    if i >= 10 {
        break;
    }
    let mut j = 1;
    loop {
        if j >= 10 {
            break;
        }
        let result = i * j;
        if result > 50 {
            break 'vertical';
        }
        println!("{}", result);
        j += 1;
    }
    println!();
    i += 1;
}
println!();
```

この章の理解度チェック P.168

1. 以下のようなコードが書けていれば正解です。

▶リストA.5 practice-1.rs

```
let mut sum = 0;
let mut i = 100;
while i <= 200 {
    if i % 2 != 0 {
        i += 1;
        continue;
    }
    sum += i;
    i += 1;
}
println!("{}", sum);
```

この場合は「偶数でない場合continue」としましたが、比較演算子を「==」とすることで「奇数以外」とすることもできます。Rustには「(初期化、条件、繰り返し後処理)」のような構文がないので、単にcontinueしただけではwhile式が永遠に終了しなくなるので注意しましょう。「i += 1;」をif式の前に1回だけ置く書き方も考えてみてください。

2. ① for ② args ③ Ok ④ i

コマンドライン引数のデータ型は文字列 (String型) です。よって、演算に際してはfrom_strメソッドで整数型に変換する必要があります。from_strメソッドの引数は&str型である必要があるためas_strメソッドを、戻り値はResult型であるためif let式で変換結果がOk(i)であるかを判定しています。

3. 以下のようなコードが書けていれば正解です。

▶リストA.6 practice-3.rs

```
let language = "Rust";
let result = match language {
    "C++" | "C#" | "Java" | "Rust" => ①
    "コンパイラ言語",
    "Python" | "Ruby" | "PHP" => ②
    "スクリプト言語",
    _ => "不明",
};
println!("{}", result);
```

match式の結果を受け取らずに、各アームの中でprintln!マクロを使っても構いません。

4. 以下のようなコードが書けていれば正解です。

▶リストA.7 practice-4.rs

```
let language = "Rust";
let result =
    if language == "C++" || language == ③
    "C#" || language == "Java" || language ④
    == "Rust" {
        "コンパイラ言語"
    } else if language == "Python" || ⑤
    language == "Ruby" || language == "PHP" {
        "スクリプト言語"
    } else {
        "不明"
    };
println!("{}", result);
```

if式の結果を受け取らずに、各ブロックの中でprintln!マクロを使っても構いません。||演算子を使わずに、else ifブロックを条件の数だけ記述しても構いませんが、コードが冗長になるだけなので、通常は避けるべきです。また、ここでは練習のためにif式を使っていますが、このようなケースではmatch式を優先して使ってください。

第5章の解答

練習問題5.1 P.181

1. 以下の3点を指摘できていれば正解です。

- foo関数の仮引数xにデータ型の指定がない
- return式の後にセミコロン (;) がない
- foo関数の呼び出しで実引数がない

以上を修正したコードは以下の通りです。foo関数の定義がmain関数の前にあるのは問題ありません。

▶リストA.8 practice-func-use.rs

```
fn foo(x: i32) -> i32 {
    return x;
}
```

```
fn main() {
    println!("{}", foo(100));
}
```

2. 以下のようなコードを書けていれば正解です。身長 height が cm で渡されるので、それを m に変換するなどのために、mut キーワードでミュータブルな引数としています。mut キーワードを使わずに、全ての計算式を1つにまとめてしまっても構いません。

▶リストA.9 practice-func.rs

```
fn main() {
    println!("{}", calc_bmi(65.0, 175.0));㉔
    // 結果: 21.224489795918366
}

fn calc_bmi(weight: f64, mut height: ㉔
f64) -> f64 {
    height /= 100.0;
    height *= height;
    weight / height // 「weight / (height ㉔
* height / 10000.0)」などでも可
}
```

練習問題5.2 P.191

1. 以下の値を指摘できていれば正解です。

```
a in func: 10
CONST_VALUE in func: 200
a in block: 2
b in block: 123
a in main: 2
b in main: 99
CONST_VALUE in main: 100
```

スコープの基本的な考え方は、スコープ内で宣言された変数などはスコープ内でのみ有効で、変更は上位スコープに影響しないというものです。上位スコープに同名の変数などがあれば、スコープ内でのみそれを隠蔽し、スコープから抜けるとまた見えるようになることを押さえておきましょう。

練習問題5.3 P.199

1. ① type ➡ 型エイリアスの定義は type 文です。
 ② square ➡ 関数を渡すには、定義した関数名のみを指定します。
 ③ FuncType ➡ 別途定義した型エイリアスで関数ポインタのデータ型を指定します。
 ④ calc ➡ 関数ポインタ型の引数で関数の処理を呼び出せます。

練習問題5.4 P.205

1. 以下のようなコードを書けていれば正解です。クロージャの本体が単一の文や式の場合は、ブロックを省略できます。また、クロージャの戻り値は return 式によらずに記述できます。さらに、クロージャ内部で使われない引数は、アンダースコア (_) で置き換えることができます。

```
|x, _| x * x
```

この章の理解度チェック P.205

1. (×) 関数には、引数が最低1個は必要である。➡ 数学の関数とは異なり、引数（入力）のない関数も定義できます。
 (○) 関数の引数には、データ型を明示しなければならない。➡ 変数宣言と異なり、関数の引数にはデータ型を型修飾で明示する必要があります。
 (×) 関数が値を戻さないときは、戻り値のデータ型は void となる。➡ 他言語にある void というデータ型はありません。値を戻さない関数は戻り値のデータ型を省略するか、ユニット型とします。
 (○) 関数は main 関数の内部など、入れ子で定義できる。➡ 関数を入れ子で定義することと、関数内でのみ有効な関数を定義できます。
 (○) 関数は、コンパイラーが認識できる限り、クレート内のどこでも定義できる。➡ 例えば、main 関数の前でも後でも構いませんし、関数

呼び出しのコードを含む関数より前に定義されていても構いません。

2. ①: i32
- ②-> (i32, i32)
- ③ a, b

関数の引数には型注釈が必要で、戻り値のデータ型は「->」で指定し、呼び出し時には定義と同じデータ型と個数の引数を与えることを押さえておきましょう。

3. 以下のようなコードが書けていれば正解です。

▶リストA.10 practice-3.rs

```
fn fibonacci(n: u32) -> u32 {
    println!("fibonacci({n}) を計算しています。");
    if n == 0 {
        0 ①
    } else if n == 1 {
        1 ②
    } else {
        fibonacci(n - 1) + ③
    }
    fibonacci(n - 2) ③
}

fn main() {
    let n = 20;
    let fib_result = fibonacci(n);
    println!("フィボナッチ数列の {n} 番目の④
    値は {fib_result} です。");
}
```

fibonacciが、引数nに自然数を受け取って、フィボナッチ数を計算して返す関数です。フィボナッチ数が、n自身と(n-1)のフィボナッチ数の和であることを利用して、fibonacci関数内部でfibonacci関数内部を何重にも呼び出すことでフィボナッチ数を計算しています(④)。再帰呼び出しでは、同じ関数が延々と呼ばれるので、再帰呼び出しを停止する基底レベルの処理が必要になります。これが、①②のnが0か1であるかを判定している条件分岐です。return式で戻り値を記述しても構いません。

なお、factorial(100)で呼び出したとしたら、答えを導

くまでに21891回もの関数呼び出しが必要となります。階乗よりも実行効率に与える影響が大きく、アルゴリズムの検討が必要な例と言えます。

4. 以下のようなコードが書けていれば正解です。

▶リストA.11 practice-4-after.rs

```
fn main() {
    let data = [3.14152, 1.41421, 2.71828];
    array_walk(data, |n| println!("{}", ②
    n.sqrt())); ①
}

fn array_walk(data: [f64; 3], output: ②
impl Fn(f64)) { ②
    for value in data {
        output(value);
    }
}
```

変更前のarray_walk関数は、平方根を出力する関数print_sqrtを受け取っていました。このような場合には、そこをクロージャで置き換えることのできる可能性があります。ここでは、print_sqrt関数の処理内容をそのままクロージャ化し、array_walk関数に渡しています(①)。array_walk関数では、②のように2番目の引数を変更するだけです。関数アイテム型であるfnを、クロージャの型であるFnに変更していますが、クロージャの型は一意に決まらないので、implキーワードを用いる必要があることに注意してください。

第6章の解答

練習問題6.1 P.220

1. 修正したコードは以下の通りです。String型を文字列リテラルで初期化する場合にはfrom関数を使います。また、代入が移動となるString型で所有権を複製するには、cloneメソッドを使います。

▶リストA.12 practice-ownership.rs

```
let s1 = String::from("こんにちは、Rust!");
let s2 = s1.clone();
println!("{s1}, {s2}");
```


2. (×) スカラー型同士の代入では、所有権は移動しません。
- (○) String型同士の代入では所有権が移動します。
- (×) &str型は文字列を参照するだけのデータ型なので所有権は移動しません。
- (○) タプル型がフィールドにString型を含むのでタプル全体で所有権が移動します。
- (×) cloneメソッドでは所有権が複製されるので移動は発生しません。

練習問題6.2 P.234

1. 以下のリストA.13のようになれば正解です。

▶リストA.13 practice-ref-1.rs

```
fn main() {
    let mut s = String::default();
    get_string(&mut s);
    println!("{}", s);
}

fn get_string(s: &mut String) {
    *s = String::from("文字列をゲットしました！");
}
```

get_string関数では&mut String型の引数を受け取るようにして、呼び出し側では&mut演算子を使ってミュータブルな参照を渡すようにします。これによって、戻り値によらずに参照経由で値を受け取ることができます。

2. 問題点は2つあります。

▶リストA.14 practice-ref-2.rs

```
fn main() {
    let mut s = String::from("こんにちは");
    let r = &mut s;
    print_string(&s);
    *r = String::from("さようなら");
    print_string(&s);

    fn print_string(m: &String) {
```

```
println!("{}", m);
}
```

- ②でミュータブルな参照を宣言しているので、参照先の①もミュータブルである必要があります(太字箇所)
- ②でミュータブルな参照を宣言して④で使用していますが、③においてイミュータブルな参照⑥を使用しています
- ③は問題となりますが、⑤は問題となりません。それは、ミュータブルな参照rは⑥の時点では使用されていないと見なされるためです。

練習問題6.3 P.240

1. ① 0.10 または 0..=9 ➡ 「..」は終点のひとつ前、「.=」は終点までとなります。
- ② 0.9 または 0..=8 ➡ ただし「9未満」なら後者の方が意味が通りやすいでしょう。
- ③ 5.. ➡ 終点を指定しないと始点以上となります。
- ④ .. ➡ 始点と終点を省略すれば全体となります。
2. ① & ➡ スライスは参照であるので「&」を前置します。
- ② 1.4 または 1..=3 ➡ 2番目から4番目に相当する数列は1.4または1..=3です。
- ③ [?..] ➡ スライスの内容をデバッグ表示するのは[?..]とるように疑問符を使います。

この章の理解度チェック P.243

1. (○) ある値を複数の変数で共有することはできません。
- (×) 所有権はCopyトレイトによって複製されたり、cloneメソッドで複製したりできます。
- (○) 所有権が移動することを特にムーブと呼びます。
- (×) 所有権のコピーは、Copyトレイトが実装されているデータ型に対して実施されます。
- (○) 変数が所有する値はスコープから抜ける時点で破棄されます。
- (○) 所有権を持たずに値を使うことは借用と呼ばれます。

- (×) ミュータブルな参照があるときイミュータブルな参照は使えません。
- (×) イミュータブルな参照は一つだけ使うことが許されています。
- (○) 参照型変数から参照先の値を取り出すことは参照外しと呼ばれます。
- (×) 参照を取得するのは&演算子、参照外しは*演算子です。

2. ① String::new()⇒String型を空文字列で初期化します。
- ② &mut ⇒受け取り可能なようにミュータブルな参照として渡します。
- ③ &mut String⇒String型へのミュータブルな参照を受け取ります。
- ④ last_name⇒姓を受け取ります。read_lineメソッドには&mut String型の変数をそのまま渡します。
- ⑤ first_name⇒名を受け取ります。あとは④と同様です。

3. 以下のようなコードが書けていれば正解です。

▶リストA.15 practice-3.rs

```
// フィボナッチ数列の次の数を生成するクロー
// ジャを返す
fn create_closure() -> impl FnMut() -> u32 {
    let mut count = 0;
    // 現在のn (キャプチャーされる)
    let mut a = 0_u32;
    // 現在のf(n-1) (キャプチャーされる)
    let mut b = 1_u32;
    // 現在のf(n-2) (キャプチャーされる)
    // ムーブクローージャとする (所有権がクロー
    // ジャ内に移動する)
    move || {
        let number = if count == 0 {
            a // n = 0のときに返る値 (0)
        } else {
            if count == 1 {
                b // n = 1のときに返る値 (1)
            } else {
                // n > 1のときはフィボナッチ数を計算
                して返す
                let c = a + b;
                a = b;
                b = c;
            }
        }
        count += 1;
        number
    }
}
```

```
c
}
};
count += 1;
number
}
}
```

フィボナッチ数では、目的の数を求めるために直前の2回分の結果を保持する必要があります。これを外部変数としてキャプチャーし、定義に従って計算した結果を返すクローージャを作成しています。

4. 以下のようなコードが書けていれば正解です。

▶リストA.16 practice-3.rs

```
fn main() {
    let mut get_fibonacci =
    create_closure();
    for count in 1..=10 {
        println!("{count}: {}",
        get_fibonacci());
    }
}
```

数列はコレクションとしても機能するので、このようにfor式で使用し、要素の値も取り出せることを押さえておきましょう。

第7章の解答

練習問題7.2 P.262

1. 以下のようなコードが書けていれば正解です。

▶リストA.17 practice-split.rs (lib-stringsパッケージ)

```
let str = "山田\t太郎\t男\t17歳\t新潟県";
let data = str.split('\t').collect::
<Vec<_>>();
for temp in data {
    println!("{temp}");
}
```

2. 以下のようなフォーマット文字列なら正解です。

▶リストA.18

practice-format.rs (lib-stringsパッケージ)

```
println!("{:-<8}:{:->4}:{:->5.2}",  
"山田花子", 50, 3.14159);
```

練習問題 7.3 P.278

1. 以下のようなコードが書けていれば正解です。

▶リストA.19 practice-match.rs (lib-regexパッケージ)

```
let s = "私の住所は〒225-0000 横浜市青葉町  
0-0-0です。\\nあなたの住所は〒273-9999  
船橋市海老川町9-9ですね。";  
let r = Regex::new(r"\\d{3}-\\d{4}").  
unwrap();  
let matched = r.find_iter(s);  
for m in matched {  
    println!("{}", m.as_str());  
}
```

マッチする文字列が1つと分かっている場合には、`find_iter` メソッドの代わりに `find` メソッドを用いても構いません。

2. 以下のようなコードが書けていれば正解です。

▶リストA.20

practice-replace.rs (lib-regexパッケージ)

```
let s = "お問い合わせはhoge@example.com  
まで";  
let r = RegexBuilder::new(r"([a-z0-9  
.!#$%&'*/=?^_{ }~-]+)@([a-z0-9-+]  
(\\.[a-z0-9-+])*)")  
    .case_insensitive(true)  
    .build()  
    .unwrap();  
println!("{}", r.replace(s, "<a href=  
'mailto:$0'>$0</a>"));
```

マッチした文字列を置換後の文字列に含めるには、特殊変数として `$0` を使います。サブマッチ文字列を使うときには、`$1`、`$2`…を使います。ここではRegex

Builder構造体を使用しましたが、Regex構造体を埋め込みフラグとともに使っても構いません。

練習問題 7.4 P.286

1. 以下のようなコードが書けていれば正解です。

▶リストA.21

practice-parse.rs (lib-datetimeパッケージ)

```
let datetime = DateTime::parse_from_str(  
    ("2025/04/29 14:20:17 +09:00",  
    "%Y/%m/%d %H:%M:%S %z").unwrap();  
println!("{}", datetime.day(),  
datetime.hour());
```

任意の日付／時刻文字列を解析したい場合には、`parse_from_str`関数を使います。この関数では、タイムゾーンの指定が必須なことに注意しましょう。

2. 以下のようなコードが書けていれば正解です。

▶リストA.22

practice-add.rs (lib-datetimeパッケージ)

```
let datetime = Local::now();  
let days = Days::new(14);  
println!("{}", datetime.checked_add_days(  
    (days).unwrap());
```

`Days` 構造体を使う代わりに `TimeDelta::days` 関数で `TimeDelta` 構造体を使っても同様です。この場合は、`checked_add_days` メソッドではなく `checked_add_signed` メソッドを使います。

練習問題 7.5 P.297

1. 以下のようなコードが書けていれば正解です。

▶リストA.23 practice-file.rs

```
use std::env;  
use std::fs::OpenOptions;  
use std::io::{BufWriter, Write};  
…略…  
let filename = "data.txt";  
let mut options = OpenOptions::new();
```

```
let file = options.create(true)
    .append(true)
    .open(filename)
    .unwrap();
let mut bw = BufWriter::new(file);
let args = env::args();
writeln!(bw, "{}", args.collect<Vec<_>>().join(",")).unwrap();
```

ファイルを追記モードで開くには、OpenOptions構造体を使い、create（存在しなければ作成）、append（追記モードで開く）をそれぞれ呼び出します。コマンドライン引数はargsはArgs構造体でイテレータを実装しているので、collectメソッドでベクターに変換し、joinメソッドで連結します。

このプログラムに引数を適当に付けて実行してdata.txt ファイルを出力させてみると、プログラムファイル名を含めた引数の内容が、カンマ区切りで書き出されているのを確認できます。

```
> cargo run --bin practice-file one two three
...略...
Running `target\debug\practice-file.exe one two three`
> cat data.txt
target\debug\practice-file.exe,one,two,three
```

練習問題 7.6 P.302

1. 以下のようなコードが書けていれば正解です。

▶リスト A.24 practice-path.rs

```
use std::path::Path;
...略...
let path = Path::new("sample.dir");
let path_buf = path.to_path_buf();
std::fs::create_dir(&path_buf).unwrap();
```

Path構造体のnew関数で、指定する文字列のパスを作成できます。それをPathBuf構造体とするには、to_path_bufメソッドを使います。ディレクトリの作成はcreate_dirメソッドです。このプログラムは1回

目は成功しますが、2回目はPanicでエラー終了します。create_dirメソッドは、すでにディレクトリがあればPanicを発生させます。Panicを回避したければ、create_dir_allメソッドを使います。

この章の理解度チェック P.312

1. それぞれ、以下のようなコードを書けていれば正解です。コードは、配布サンプルlibraryパッケージのpractice-1.rsに収録しています。

- ① let str = "となりのきょうはよくきょうくうきょうくだ";
println!("{}", str.rfind("きょう").unwrap());
- ② let loc = "熊谷";
let temp = 40.124;
println!("{}", 市の気温は{:1}℃です.", loc, temp);
- ③ let str = "ワタシの名前は花子です.";
println!("{}", str.replace("ワタシ", "私"));
- ④ let mut dt = Local::now();
let td = TimeDelta::days(7) + TimeDelta::hours(3);
dt = dt.checked_add_signed(td).unwrap();
println!("{}", dt.format("%c").to_string());
- ⑤ let dt1 = Local.with_ymd_and_hms(2022, 6, 2, 0, 0, 0).unwrap();
let dt2 = Local.with_ymd_and_hms(2025, 7, 20, 0, 0, 0).unwrap();
println!("{}", (dt2 - dt1).num_days());

2. ① Regex() ➡ 正規表現を使うにはRegex構造体を使います。

- ② File ➡ ファイル名を受け取ってオープンするのはFile構造体の役割です。
- ③ BufReader ➡ オープンしたファイルに対してバッファ付き入力を行うのはBufReader構造体の役割です。
- ④ read_line ➡ BufReader構造体に関連付けられたファイルから1行読み込みます。引数のString型変数に追加されていくことに注意しましょう。
- ⑤ find_iter ➡ 複数箇所のマッチ内容を取り出すにはfind_iterメソッドを使います。
- ⑥ as_str() ➡ マッチ内容から文字列を取得するのはas_strメソッドです。

File／BufReader／Regex構造体の複合問題です。分からなかった、間違ってしまったという読者は、7.5節、7.3節をもう一度見直しましょう。

3. 以下のようなコードとなっていれば正解です。

▶リストA.25 practice-4.rs

```
use std::fs::OpenOptions;
use std::io::Read;
...略...

let filename = "image.png";
let mut options = OpenOptions::new();
let mut file = options.read(true)
    .open(filename)
    .unwrap();
let mut buf: [u8; 16] = Default::default();
for i in 0..16 {
    print!("{:02x}: ", i * 16);
    let bytes = file.read(&mut buf).unwrap();
    for j in 0..bytes {
        print!("{:02x} ", buf[j]);
    }
    println();
}
```

基本は、リスト7.48と同じです。バッファの長さが16バイトとなっていること、16回×16回の二重ループで表を作成するようにしています。

4. 以下のようなコードとなっていれば正解です。

▶リストA.26 practice-4.rs

```
use rust_decimal::dec;
...略...

println!("絶対値: {}", dec!(-3.14).abs());
println!("最大値: {}", dec!(3.14).max(dec!(2.71)));
println!("最小値: {}", dec!(3.14).min(dec!(2.71)));
println!("切り上げ: {}", dec!(3.14).ceil());
println!("切り下げ: {}", dec!(3.14).floor());
println!("四捨五入: {}", dec!(3.14).round());
```

浮動小数点数をdec!マクロに置き換えるだけです。型サフィックスを指定するとコンパイル時にPanicとなることに注意しましょう。

第8章の解答

練習問題8.1 P.327

1. ① vec! ② vec[2] ③ remove ④ insert ⑤ day

最終結果からリストへの操作を想像しましょう。特に、②と④においては構文からインデックスによるアクセス、insertメソッドの使用であることのあたりを付けやすいはずです。

練習問題8.3 P.336

1. ① from ② insert ③ remove ④ (key, value)

最終結果からマップへの操作内容を類推する問題です。fromメソッドによる初期化、insert／removeメソッドによる操作、マップの要素を受け取るタブルなど、マップへの基本的な操作を再確認してください。

練習問題8.4 P.343

1. セットは、リストと異なり要素の重複を許さないコレクションで、数学の集合に似た性質を持ちます。ある値がセットに含まれているか、セット間における包含関係に関心がある場合に使用します。
- セットの実装には、HashSet構造体、BTreeSet構造体があります。双方の違いは、要素の並び順を管理するかどうかです。BTreeSet構造体は、あらかじめ決められたルールで値を並び替えます。

練習問題8.5 P.348

1. ① VecDeque ② push_front ③ front
④ get ⑤ push_back

結果からキューへの操作を想像しましょう。キューでは主な操作位置が2つあるのでリストに比べると複雑ですが、headの操作 = front、tailの操作 = backをメソッド名と関連付けると、表示と使われているメソッドの対応が分かるでしょう。

この章の理解度チェック P.354

1. (×) Vec構造体への要素の挿入／削除は、要素の移動を伴うため、先頭に近くなるほど遅くなります。
- (×) リンクの付け替えは、一般的には高速です。要素の追加／削除が頻繁に発生する用途では、Vec構造体よりLinkedListが向いています。
- (×) BTreeSet構造体に対する説明です。HashSetは要素の並び順を管理しません。
- (○) 正しい記述です。
- (○) 正しい記述です。

コレクションは、それぞれに得手不得手があります。用途そのものはいずれの構造体もほぼ共通であるので、そのときどきの用途に応じて、適切な構造体を使い分けることが重要です。

2. 以下の点を指摘できていれば正解です。

- リスト (Vec構造体) に変更を加える場合には、mut キーワードが必要です
- vec[5] はリストの範囲外であるので、出力結果から vec[2] が妥当です
- 同じく remove(6) もリストの範囲外であるので、実行時に panic となります。出力結果から remove(4) が妥当です

以上を修正した正しいコードは以下の通りです。インデックスによるアクセスや、insert メソッドや remove メソッドにおける位置指定は、0 から始まることに注意しましょう。

▶リストA.27 practice-2.rs

```
let mut vec = vec![1, 2, 3, 4];
vec[2] = 50;
vec.insert(1, 5);
vec.remove(4);
for c in &vec {
    println!("{}", c);
}
```

3. 以下のようなコードを記述できていれば正解です。

▶リストA.28 practice-3.rs

```
let mut kuku = BTreeMap::new();
for n in 1..=9 {
    for m in 1..=9 {
        kuku.insert(format!("{}", n * m), ②
n * m);
    }
}
for (k, v) in &kuku {
    println!("{}", k="{v}");
}
```

このような用途にはマップを使うことをまず押さえましょう。また、全体を出力したときにキーの順序を保持したいので、BTreeMapを使います。あとは、for式でキー文字列を生成しながら、計算結果をinsertメソッドで追加していくだけです。BTreeMapをHashMapに変えると、結果がどのように変化するか確認してもよいでしょう。

4. ① map：各要素を加工してイテレータを生成
- ② collect：イテレータからコレクションを生成
- ③ filter：要素をフィルターしてイテレータを生成
- ④ sum：総和を求める
- ⑤ all：全ての要素が条件を満たすか判定
- ⑥ any：条件を満たす要素があるか判定

第9章の解答

練習問題9.1 P.369

1. 以下のような構造体が定義できていれば正解です。

▶リストA.29 practice-struct.rs

```
struct Book {
    name: String,
    author: String,
    publisher: String,
    year: u16,
    price: u32,
    pages: u16,
}
```

2. 以下のような関数およびメソッドが定義できていれば正解です。

▶リストA.30 practice-struct.rs

```
impl Book {
    fn new(name: &str, author: &str, 
    publisher: &str, year: u16, price: 
    u32, pages: u16) -> Self {
        Book {
            name: String::from(name),
            author: String::from(author),
            publisher: String::from(publisher),
            year,
            price,
            pages,
        }
    }

    fn display(&self) {
        println!("Name: {}, Author: {}, 
        Publisher: {}, Year: {}, Price: {}, 
        Pages: {}",
            self.name, self.author, 
            self.publisher, self.year, self.price, 
```

```
self.pages);
    }
}
```

構造体のインスタンス化では、スコープにフィールド名と同名の変数があれば初期化省略記法が使えることを押さえておきましょう。また、newは関連関数で引数にselfが不要なこと、displayはメソッドなのでselfが必要なこと、メソッドからフィールドを参照するにはフィールドアクセス式(self.フィールド)を使うことなども押さえておきましょう。

練習問題9.2 P.374

1. 以下のような構造体とメソッドが定義できていれば正解です。

▶リストA.31 practice-generic.rs

```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn new(x: T, y: T) -> Self {
        Point { x, y }
    }

    fn get_x(&self) -> &T {
        &self.x
    }

    fn get_y(&self) -> &T {
        &self.y
    }
}
```

2. 以下のような主旨の回答となっていれば正解です。

型パラメーターTは任意のデータ型を表すが、その全てのデータ型が「*」+とといった二項演算、sqrtメソッドを実装しているとは限らないから。

練習問題9.3 P.391

1. 以下のように実装のコードが書けていれば正解です。

▶リストA.32 practice-trait.rs

```
impl Greet for Person {
    fn greet(&self) {
        println!("こんにちは！ {} です！", &self.name);
    }
}

impl Greet for Dog {
    fn greet(&self) {
        println!("ワン！ {} だよ！", &self.name);
    }
}
```

Greetトレイトのgreetメソッドには中身がないので、必須メソッドとなります。実装側では、メソッドの中身が必要になることを押さえておきましょう。余裕があれば、インスタンス化とメソッド呼び出しのコードも書いてみましょう。

この章の理解度チェック P.401

1. (×) 構造体にも列挙体にも等しくメソッドを実装できます。
- (×) 構造体にメソッドを実装するにはimplブロックの中に定義します。
- (×) フィールドには組み込み型とユーザー定義型のあらゆるデータ型を指定できます。
- (○) 正しい記述です。
- (×) トレイトは列挙体にも実装できます。
- (○) 正しい記述です。

メソッドの実装は「impl 構造体名」、トレイトの実装は「impl トレイト名 for データ型」構文を使います。同じimplキーワードを使いますが、構文は異なるので注意しましょう。

2. 以下の点を指摘できていれば正解です。

- ジェネリック型にメソッドを実装する場合implキーワードにも型パラメーターが必要
- new関数には戻り値を表す「-> Self」または「-> Pair<T, U>」が必要
- swapメソッドには引数selfが必要

以上を修正した正しいコードは以下の通りです。

▶リストA.33 practice-2.rs

```
struct Pair<T, U> {
    first: T,
    second: U,
}

impl<T, U> Pair<T, U> {
    fn new(first: T, second: U) -> Self {
        Pair { first, second }
    }

    fn swap(&self) -> Pair<U, T> {
        Pair {
            first: &self.second,
            second: &self.first,
        }
    }
}
```

ここで、swapメソッドの引数はselfすなわちインスタンスそのものなので、型パラメーターT、Uによっては所有権の移動が発生します。このため、戻り値として新たにPair構造体を生成して返しています。

3. ① derive：derive属性を使うとトレイトを自動で実装できる
- ② enum：列挙体はenumキーワードで定義する
- ③ impl：メソッドの実装はimplキーワードを使う
- ④ Greetable：列挙子は「列挙体::列挙子」で参照する
- ⑤ greet：メソッドはメソッド呼び出し式で呼び出せる
4. 以下のようなコードが書けていれば正解です。

▶リストA.34 practice-3.rs

```
trait Printer {
    fn print(&self);
}
```



```

}

struct LaserPrinter {
}

impl Printer for LaserPrinter {
    fn print(&self) {
        println!("LaserPrinterで印刷しています。");
    }
}

struct InkjetPrinter {
}

impl Printer for InkjetPrinter {
    fn print(&self) {
        println!("InkjetPrinterで印刷しています。");
    }
}

```

Printerトレイトのprintメソッドには既定の内容がないので、必須メソッドとなります。このため、Printerトレイトを実装する2つの構造体にはprintメソッドの実装が必須になることに注意しましょう。

第10章の解答

練習問題10.1 P.418

1. 以下のような構造体が定義できていれば正解です。

▶リストA.35 practice-box.rs

```

enum Tree {
    Node(i32, Box<Tree>, Box<Tree>),
    Nil,
}

```

考え方は、リスト10.8の連結リストと同じです。リストでは続くノードを保持しますが、ツリーではぶら下がるノードが2つになります。ツリーにノードが全くない状態を表すために列挙子Nilが設けられている点も同様です。

2. 以下のようなコードが記述できていれば正解です。

▶リストA.36 practice-box.rs

```

use Tree::{Node, Nil};
...略...
let tree = Node(1,
    Box::new(Node(11,
        Box::new(Nil),
        Box::new(Nil)
    )),
    Box::new(Node(66,
        Box::new(Nil),
        Box::new(Nil)
    ))
);

```

use宣言を用いずに、Tree::Node、Tree::Nilとパス式で記述しても問題ありません。

3. 以下のようなコードが記述できていれば正解です。

▶リストA.37 practice-box.rs

```

fn display_tree(node: &Tree, depth: uize) {
    match node {
        Node(value, left, right) => {
            println!("{}", ノード: {}",
            " ".repeat(depth * 2), value);
            println!("{}", 左: ",
            " ".repeat(depth * 2));
            display_tree(left, depth + 1);
            println!("{}", 右: ",
            " ".repeat(depth * 2));
            display_tree(right, depth + 1);
        },
        Nil => {
            println!("{}", なし",
            " ".repeat(depth * 2));
        },
    }
}

fn main() {
    ...ツリーの初期化コードは省略...
    display_tree(&tree, 0);
}

```

解答例ではツリー構造を再帰的に探索していますが、繰り返し構文で記述しても問題ありません。また、子

ノードの深さに応じてインデントを加えるために引数 depth を display_tree 関数に加えています。必須ではありません。うまく左右のノードを末端までたどっていけるコードになっていれば正解です。

練習問題 10.2 P.424

1. 以下のような構造体が定義できていれば正解です。

▶リスト A.38 practice-rc.rs

```
struct LinkedList {
    value: i32,
    next: Option<Rc<LinkedList>>,
}
```

value フィールドはノードの値、next フィールドは次ノードです。Option 型となっているので、次ノードがあればその Some(Rc<LinkedList>) を、なければ None を持たせることで、リスト構造が表現できます。このとき、Option 型に直接 LinkedList 構造体を入れることはできないことを押さえてください。このような再帰的な構造体では、そのサイズをコンパイル時に決めることはできないため、Box<T> 構造体や Rc<T> 構造体を介することでサイズを固定とすることができるようになります。

2. 以下のようなコードが記述できていれば正解です。

▶リスト A.39 practice-rc.rs

```
let list = Rc::new(LinkedList {
    value: 100,
    next: Some(Rc::new(LinkedList {
        value: 200,
        next: Some(Rc::new(LinkedList {
            value: 500,
            next: None
        })),
    })),
});
```

Option<T> を使うと、解説で使った列挙体を使う場合に比較して次ノードがない場合をシンプルに記述できます。ただし、ノードがない状態（空のリスト）を表現する場合には一工夫が必要です。例えば、以下の

ようなコードで先頭ノードを持たないようにすることで、空のリストを表現できます。

```
let list: Option<Rc<LinkedList<T>>> = None;
```

この章の理解度チェック P.432

1. (○) スマートポインターでは、値をヒープに配置します。
(×) スマートポインターでは、メモリの解放は自動的に行われます。
(○) 正しい記述です。
(○) 正しい記述です。
(×) Rc<T> 構造体は、参照カウンターを用いて複数の所有権があることを疑似的に管理します。
(○) 正しい記述です。

スマートポインターは、値をヒープに配置し、そのメモリ領域は自動で確保、解放します。単純に値を格納／取得する Box<T>、参照カウンターを持ち所有権の複製に対応する Rc<T>、内部可変性を用いて参照の制約を回避する Cell<T>、RefCell<T> などがあります。

2. 以下の点を指摘できていれば正解です。

- Box<T> 構造体はジェネリック型なので型パラメーター（この場合は Stack）が必要
- new 関数には戻り値を表す「-> Self」が必要
- push メソッドの戻り値は Item(i32, Box<Stack>) である必要がある

以上を修正した、呼び出し例を含めた正しいコードは以下の通りです。

▶リスト A.40 practice-2.rs

```
enum Stack {
    Item(i32, Box<Stack>),
    Empty,
}

use Stack::{Empty, Item};
```

```
impl Stack {
    fn new() -> Self {
        Empty
    }

    fn push(self, v: i32) -> Self {
        Item(v, Box::new(self))
    }

    fn pop(self) -> (Option<i32>, Self) {
        match self {
            Item(v, next) => (Some(v), *next),
            Empty => (None, Empty),
        }
    }
}

fn main() {
    let stack = Stack::new();
    let stack = stack.push(100);
    let stack = stack.push(200);
    let (v, stack) = stack.pop();
    println!("pop: {:?}", v); ②
    // 結果: pop: Some(200)
}
```

pop メソッドについては実装例という位置付けで、特に誤りを入れていませんでした。ポップでは、スタックが空であることを想定する必要があるので、戻り値はポップした結果である Option 型と、ポップ後のスタックからなるタプルとしています。スタックが空であれば None と Stack::Empty を返しますが、データがあれば Some(値) とその次のデータを返します。

3. ① RefCell : RefCell<T> 構造体を Rc<T> でラップすることで可変の値を共有できる
- ② borrow_mut : RefCell<T> 構造体が内包する値を書き換えるには borrow_mut メソッドを使う
- ③ clone : Rc<T> 構造体で所有権を複製するには clone 関数を使う

この問題は、連結リストを列挙体ではなく構造体で定義し、次ノードの有無を Option 型で表現する例となっていますが、練習問題 10.2 におけるものと異なり、RefCell<T> 構造体を用いて値すなわちリンク先の書き換えができるようになっています。Rc<T> 構

造体だけでは困難であった、動的な連結リストを実現できます。

4. 以下のようなコードが記述できていれば正解です。

▶リスト A.41 practice-4.rs

```
use std::cell::RefCell;
use std::rc::Rc;

// Widget 列挙体の定義
#[derive(Debug)]
enum Widget {
    // Window 列挙子は、タイトル文字列と ②
    // 子ウィジェットのリストを持つ
    Window {
        title: String,
        children: RefCell<Vec<Rc<Widget>>>,
    },
    // Button 列挙子は、ラベル文字列を持つ
    Button {
        label: String,
    },
}

// 子ウィジェットを追加する add_child メソッド ②
// を列挙体の実装する
impl Widget {
    // 引数は、追加先のウィジェットと、追加する ②
    // ウィジェット
    fn add_child(parent: &Rc<Widget>, ②
    child: Rc<Widget>) {
        // 追加先が Window 列挙子であるときのみ追加
        if let Widget::Window { children, ②
        .. } = parent.as_ref() {
            children.borrow_mut().push(child);
        }
    }
}

// 略...

// それぞれのウィジェットを生成する
let window = Rc::new(Widget::Window {
    title: "メイン".to_string(),
    children: RefCell::new(vec![]),
});
let ok = Rc::new(Widget::Button {
    label: "OK".to_string(),
});
let cancel = Rc::new(Widget::Button {
    label: "キャンセル".to_string(),
});
```

```
// Buttonを2つ、Windowに追加する
Widget::add_child(&window, ok);
Widget::add_child(&window, cancel);
// 構造をデバッグ出力する
println!("{:?}", window);
```

ポイントは、それぞれのウィジェットを `Re<Widget>` 構造体としておくことと、子ウィジェットを保持するリストを `RefCell<Vec<Re<Widget>>>` 構造体とすることです。それぞれ、ウィジェットを複数の場所から参照できるようにすること、Window 列挙子の持つ子ウィジェットのリストにあとから追加できるようにするためです。特に `RefCell<T>` 構造体を使うことで、ウィジェット自身を可変にすることなく変更できるので、参照の制約を回避しながら、柔軟なツリー操作が可能になります。

第 11 章の解答

練習問題 11.1 P.447

1. 以下のようなコードが記述できていれば正解です。

▶リスト A.42 practice-panic.rs

```
let args: Vec<String> = env::args().collect();
if args.len() < 2 {
    panic!("ファイル名を指定してください。");
}
let filename = &args[1];
...略...
```



```
thread 'main' (4184) panicked at
src\bin\practice-panic.rs:8:9:
ファイル名を指定してください。
```

これは `panic!` マクロの利用方法としての例なので正解ですが、`panic!` マクロによるプログラムの終了は正当な理由がある場合にとどめるべきです。基本的には、`args.len()` が 2 以上の場合にのみ後続の実行する、`args.len()` が 2 未満のときは `main` 関数から脱出するといった、通常の制御構文の範囲でコードを記述すべきです。

2. 以下のようなコードとできていて、エラーメッセージが出力されていれば正解です。

▶リスト A.43 practice-unwrap.rs

```
println!("{}", i32::from_str("ABC").unwrap());
```



```
thread 'main' (12144) panicked at
src\bin\practice-unwrap.rs:6:41:
called `Result::unwrap()` on an `Err`
value: ParseIntError {kind: InvalidDigit}
```

`from_str` メソッドの引数は "ABC" でなくてもよく、とにかく符号付き整数と解釈できるものでなければ何でも構いません。`unwrap` メソッドが `Err` を返すとき、その値はメソッドの発生させる可能性のあるエラーとなることにも注目してください（この場合は整数解析エラーであることを示す `ParseIntError` 構造体）。

この章の理解度チェック P.458

1. (×) Rustには例外という仕組みはありません。主に、`Panic` と `Result` 列挙体でエラーを処理します。
 (○) 正しい記述です。
 (×) 逆です。回復不能なエラー処理は `Panic` により、回復可能なエラー処理は `Result` 列挙体によって実装されます。
 (○) 正しい記述です。バックトレースともいいます。
 (×) `unwrap` メソッドでは、インスタンスが `Ok` の場合は安全に値を取り出すことができますが、`Err` の場合は `Panic` となります。
 (×) `Result` 列挙体によるエラー処理を簡潔にするのは `?` 演算子です。

Rustは、他言語で見られる例外のようなエラー処理システムを備えていません。`Panic` や、`Result` 列挙体などによってエラーをハンドリングし、必要があればプログラムを異常終了させます。`Panic` によって異常終了する仕組みは回復不能なエラー処理、`Result` 列挙体によってエラーをハンドリングする仕組みは回復可能なエラー処理と呼ばれます。

2. 以下の点を指摘できていれば正解です。

- 変数input1、input2は文字列を受け取るので宣言時にmut キーワードが必要
- read_line メソッドの戻り値にはunwrap メソッドの呼び出しが必要（2箇所）
- Panic の発生はpanic! マクロ。expect! というマクロは存在しない

以上を修正した正しいコードは以下の通りです。

▶リストA.44 practice-2.rs

```
let mut input1 = String::new();
println!("1個目の文字列を入力してください。");
std::io::stdin().read_line(&mut input1).unwrap();
if input1.trim_end().is_empty() {
    panic!("1個目の文字列が空です。");
}

let mut input2 = String::new();
println!("2個目の文字列を入力してください。");
std::io::stdin().read_line(&mut input2).unwrap();
if input2.trim_end().is_empty() {
    panic!("2個目の文字列が空です。");
}

println!("{}", input1.trim_end(), input2.trim_end());
```

3. 以下のようなコードが書けていれば正解です。

▶リストA.45 practice-3-ans.rs

```
let mut input = String::new();
println!("1個目の数値を入力してください。");
std::io::stdin().read_line(&mut input).unwrap();
let num1 = match i32::from_str(input.trim()) {
    Ok(n) => n,
    Err(_) => {
        eprintln!("数値ではありません。");
        return;
    }
};
```

```
input.clear();
println!("2個目の数値を入力してください。");
std::io::stdin().read_line(&mut input).unwrap();
let num2 = match i32::from_str(input.trim()) {
    Ok(n) => n,
    Err(_) => {
        eprintln!("数値ではありません。");
        return;
    }
};

let sum = num1 + num2;
println!("和: {sum}");
let diff = num1 - num2;
println!("差: {diff}");
let prod = num1 * num2;
println!("積: {prod}");
if num2 == 0 {
    eprintln!("0で割ることはできません。");
    return;
}
let quot = num1 / num2;
println!("商: {quot}");
```

問題文のコードで、Panicが発生する可能性があるのは、分かりやすいところでunwrapメソッドを呼び出している箇所です。ここをmatch式などで置き換えて適切に仕分ければ、Panicとせずにプログラムを終了させることができます。解答では、from_strメソッドについてのみmatch式を適用しましたが、できればread_lineメソッドにもmatch式を適用して万全を期すのがよいでしょう。

なお、Panicが発生させるコードには、除算の式もあります。除数が0の場合には整数間の除算時にPanicとなるので、あらかじめ除数が0であるかのチェックを入れると安全なコードになります。

4. 以下のようなコードとなっていれば正解です。

▶リストA.46 practice-4.rs

```
fn main() -> Result<(), Box<dyn Error>> {
    let mut input = String::new();
    println!("1個目の数値を入力してください。");
    std::io::stdin().read_line()
```

```

(&mut input)?; ————— ❷
    let num1 = i32::from_str(
(input.trim()))?; ————— ❸

    input.clear();
    println!("2個目の数値を入力してください。");
    std::io::stdin().read_line(
(&mut input)?; —————
        let num2 = i32::from_str(
(input.trim()))?;

    let sum = num1 + num2;
    println!("和: {sum}");
    let diff = num1 - num2;
    println!("差: {diff}");
    let prod = num1 * num2;
    println!("積: {prod}");
    if num2 == 0 {
        eprintln!("0で割ることはできません。");
        Ok(()) —————
    } else {
        let quot = num1 / num2;
        println!("商: {quot}");
        Ok(()) ————— ❹
    }
}

```

main関数がエラーの委譲を受けるには、❶のように戻り値がResult列挙体である必要があります。これにより、❷❸で演算子を使ってエラー発生時にmain関数にエラー処理を委ねることができます。このとき、read_lineメソッドとfrom_strメソッドの返すエラーのデータ型は異なるので、それらが実装するトレイトであるErrorを動的に受けるボックスとして、Result型を使用しています。❹は、main関数にエラーがないときの戻り値として必要です。

第12章の解答

練習問題 12.1 P.466

1. 以下のようなコマンドとなります。

```

cargo new test_program
または
cargo new --bin test_program

```

--binオプションはバイナリクレートを含めるためのもので、既定のオプションです。省略しても意味は同じです。以下の手順で作成、実行ができていれば問題ありません。

```

> cargo new test_program
    Creating binary (application)
`test_program` package
> cd .\test_program\
> cargo run
    Compiling test_program v0.1.0
(…略…)\test_program)
    Finished `dev` profile [unoptimized
+ debuginfo] target(s) in 2.23s
    Running `target\debug\test_program.
exe`
Hello, world!

```

2. 以下のように実行できていれば正解です。フォルダーの作成やファイルのコピーは、エクスプローラーなどを使っても構いません。

```

> mkdir src/bin
> copy src/main.rs src/bin/another.rs
> cargo run --bin another
    Compiling test_program v0.1.0
(…略…)\test_program)
    Finished `dev` profile [unoptimized
+ debuginfo] target(s) in 0.69s
    Running `target\debug\another.exe`
Hello, world!

```

cargo runコマンドにおいて、--binオプションとともにクレート名（ここではanother）を指定する必要があります。あることを押さえておいてください。

練習問題 12.2 P.479

1. 以下のようなコードとなっていれば正解です。

▶リストA.47 practice-module.rs

```

mod my_module {
    pub fn my_func() {}
}

fn main() {

```

```
my_module::my_func();
}
```

忘れやすいのは、モジュール内のmy_func関数定義に付けるpubキーワードです。モジュールmy_moduleは同一階層なのでpubキーワードは不要ですが、その内部のアイテムについてはpubキーワードが必要なことに注意しましょう。

2. 以下のようなコードとなっていれば正解です。

▶リストA.48 practice-struct.rs

```
mod my_module {
    pub struct MyStruct {
        f: i32,
    }

    impl MyStruct {
        pub fn new(f: i32) -> Self {
            Self { f }
        }

        pub fn get(self: &Self) -> i32 {
            self.f
        }
    }
}

fn main() {
    use my_module::MyStruct;

    let mut s = MyStruct::new(100);
    println!("{}", s.get());
}
```

my_moduleモジュールの外部から構造体やその関数を利用するには、構造体定義と関数定義の冒頭にpubキーワードが必要です。また、use宣言で名前を取り込む際には、現在のモジュール階層からたどれるパスでなくてはなりません。フィールドには直接アクセスできないということなのでフィールドfにはpubキーワードは不要で、その代わりにアクセスサメソッドgetでfにアクセスする必要があります。

3. それぞれ、以下のようなコードとなっていれば正解です。

▶リストA.49 my_module.rs

```
pub struct MyStruct {
    f: i32,
}

impl MyStruct {
    ...メソッドの実装はpractice-struct.rsと☑
    同じなので省略...
}
```

▶リストA.50 practice-devide.rs

```
mod my_module;
...以降はpractice-struct.rsと同じなので省略...
```

モジュール側（モジュールと同名のmy_module.rsである必要あり）には、モジュール定義の中身だけを記述し、利用側（practice-devide.rs）にはmodキーワードのみを置き、中身は記述しないのがポイントです。

この章の理解度チェック P.487

- （×）クレートとはコンパイルの単位で、複数のソースファイルから構成されます。
（○）正しい記述です。
（○）正しい記述です。
（○）正しい記述です。
（×）モジュール内のアイテムを外部から利用可能にするにはpubキーワードが必要です。
（×）まとめてビルドできますが、ビルド結果はワークグループでまとめられます。
- ① pub⇒外部に公開したいアイテムにはpubキーワードを付与します。
② mod⇒モジュールの宣言はmodキーワードを使います。
③ super⇒親モジュール名はsuperで代替できます。
④ crate⇒クレートルートはcrateで代替できます。
⑤ network⇒現モジュールからの相対指定はモジュール名を記述します。

モジュールシステムではたくさんのキーワードが登場するので迷いがちです。個々のキーワードの意味を理解して使い分けられるようにしましょう。

3. 以下の点を指摘できていれば正解です。

- 別ファイルにあるモジュールを使うには、mod 宣言を使う（useではない）
- 別ファイルにあるモジュール定義には、mod 宣言は不要
- モジュール外部から呼び出す関数にはpubキーワードが必要

以上を修正した、正しいコードは以下の通りです。太字が修正箇所、薄字が削除箇所です。

▶リストA.51 practice-3.rs (モジュール利用側)

```
mod my_module;  
  
fn main() {  
    my_module::func();  
}
```

▶リストA.52 my_module.rs (モジュール提供側)

```
mod my_module {  
    pub fn func() {}  
}
```

4. ① workspace ➡ ワークスペース共通の設定は [workspace] セクションです。
- ② members ➡ メンバーのリストは members キーに列挙します。
- ③ package ➡ ワークスペースのメンバーで共有したいパッケージ情報を [workspace.package] セクションに記述します。

ワークスペース共通の設定はワークスペースの Cargo.toml ファイルで、個々のメンバーパッケージの Cargo.toml ファイルではそれらを継承するか、独自の設定とすることが使分けられます。

第13章の解答

練習問題 13.1 P.505

1. 以下のような関数が定義できていれば正解です。

▶リストA.53 lib.rs

```
pub fn sub(left: i64, right: i64) -> i64 {  
    left - right  
}  
…略…  
#[cfg(test)]  
mod tests {  
    use super::*;  
    …略…  
    #[test]  
    fn sub_works() {  
        let result = sub(2, 2);  
        assert_eq!(result, 0);  
    }  
}
```

テスト関数を指定して実行するには、関数名をフルパスで指定します。

```
> cargo test --lib tests::sub_works  
…略…  
  
running 1 test  
test tests::sub_works ... ok  
  
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 5 filtered out;  
finished in 0.00s
```

2. 以下のような関数が定義できていれば正解です。

▶リストA.54 lib.rs

```
#[test]  
fn sub_works_custom() {  
    let result = sub(2, 2);  
    assert!(result == 4, "2 - 2 の結果は 2  
} ではありませんでした。", result);  
}
```

カスタムメッセージを指定するには assert! マクロを使う必要があります。このため、比較のための式を記

述し、そのあとにformat!マクロと同じ形式で書式指定文字列と引数を記述します。

練習問題 13.2 P.507

1. 以下のような関数が定義できていれば正解です。

▶リストA.55 tests/practice_integration.rs

```
#[test]
fn add_sub_success() {
    let result_add = testing::add(5, 4)
    as i64;
    let result_sub = testing::sub(10, 1);
    assert_eq!(result_add * result_sub, 81);
}
```

この結合テストは、`--test`オプションに`practice_integration`を指定して実行します。

```
> cargo test --lib --test practice_integration
...略...

running 1 test
test add_sub_success ... ok

test result: ok. 1 passed; 0 failed;
0 ignored; 0 measured; 0 filtered out;
finished in 0.00s
```

この章の理解度チェック P.509

1. (×) 単体テストとは、単一の関数の動作をテストします。
- (○) 正しい記述です。
- (×) ドックテストとは、ドキュメンテーションコメント中のコードをテストします。
- (○) 正しい記述です。
- (×) 単体テストのコードは、testsモジュールに置かれます。
- (×) 結合テストのコードは、パッケージルートのtestsフォルダーに配置します。

2. ① <⇒負数であることをチェックしたいので<演算子です。
- ② `cfg`⇒条件コンパイルは`cfg`属性です。
- ③ `tests`⇒テスト関数はtestsモジュールに配置します。
- ④ `should_panic`⇒panicを期待するのは`should_panic`属性です。
- ⑤ `-1.0`⇒負数を与えてpanicを発生させます（負数であればOK）。

テスト関数に関連する構文はほぼ定型です。似たようなキーワードが多いので混乱しがちですが、形として覚えておきましょう。

3. 以下の点を指摘できていれば正解です。

- テスト関数に付与するのはtest属性（testsではない）
- `result_add`関数の戻り値はu64型なので、`square_root`関数に渡すにはキャストが必要
- 値が等しいことを確認するアサーションは`assert_eq!`マクロ

以上を修正した、正しいコードは以下の通りです。

▶リストA.56 practice-3.rs

```
#[test]
fn add_square_root_success() {
    let result_add = testing::add(5, 4);
    let result_sqrt = testing::square_root(result_add as f64);
    assert_eq!(result_sqrt, 3.0);
}
```

4. 以下のようなコードとなっていれば正解です。実際にドックテストを実行して、コメント中のコードが正しく動作するかどうか確認してください。

▶リストA.57 lib.rs

```
/// Subtracts two number given.
///
/// # Arguments
/// * `left` - left side number
/// * `right` - right side number
///
```

```

/// # Examples
///
/// ```
/// let left = 5;
/// let right = 4;
/// assert_eq!(1, testing::☐)
sub(left, right));
/// ```

```

```

temp_vec.push(2);
temp_vec.push(3);
temp_vec.push(4);
temp_vec.push(5);
temp_vec
};
{
    ::std::io::_print(format_args!(☐)
("{0:?}\n", v));
};
}

```

第14章の解答

練習問題 14.1 P.524

1. 以下のような宣言的マクロが定義できていれば正解です。

▶リスト A.58 practice-decl-macro.rs

```

macro_rules! my_vec {
    ($($x:expr),*) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}

```

マクロ呼び出しのコードは、以下のようになります。

▶リスト A.59 practice-decl-macro.rs

```

let v = my_vec![1, 2, 3, 4, 5];
println!("{:?}", v);☐
// 結果: [1, 2, 3, 4, 5]

```

3. 以下のようにコマンドを実行して、出力が得られていれば正解です。

```

> cargo expand --bin practice-decl-macro
...略...
fn main() {
    let v = {
        let mut temp_vec = Vec::new();
        temp_vec.push(1);
    };
}

```

練習問題 14.2 P.535

1. TokenStream は、Rust のプログラムコードをトークンの列として持ち、手続き的マクロの入出力となるデータ型です。AST は抽象構文木と言い、TokenStream をパースして得られる構文の木構造を表します。木構造なので、例えば関数名を取り出すといった操作が容易です。

2. 以下のような役割分担となります。

- proc_macro2: TokenStream などの基本的な構造体を備える proc_macro のラッパー
- syn: TokenStream を AST に変換する
- quote: AST や識別子を埋め込んで TokenStream を構築する

この章の理解度チェック P.543

1. (○) 正しい記述です。
(○) 正しい記述です。
(×) 関数風マクロなどに分類されるのは手続き的マクロです。
(×) macro_rules! マクロを使うのは宣言的マクロです。
(○) 正しい記述です。
(×) マクロの展開結果を見るには cargo expand コマンドを実行します。
2. ① macro_rules! ➡ 宣言的マクロの定義は、macro_rules! ではじめます。

- ② `ident` ➡ 識別子の指定は `ident` です。
- ③ `expr` ➡ 式の指定は `expr` です。
- ④ `*` ➡ 繰り返しにマッチしたパターンの展開は `$(...)*` です。
- ⑤ `with_args!` ➡ 宣言のマクロはマクロ名に `!` を付けて呼び出します。

宣言的マクロの構文は限定的です。最も複雑なのはフラグメント指定子と言えるので、どのようなパターンでどのフラグメント指定子を使うか復習しておきましょう。

3. 以下の点を指摘できていれば正解です。

- 関数風マクロの定義は `proc_macro` 属性 (`procedure_macro` ではない)
- 生成する関数名を指定するので式の `Expr` ではなく識別子の `Ident` としてパースする
- 参照する AST は `#` を前置する

以上を修正した、正しいコードは以下の通りです。

▶リストA.60 lib.rs

```
#[proc_macro]
pub fn generate_func(input: ☞
    TokenStream) -> TokenStream {
    let ast = parse_macro_input!(input ☞
as Ident);
    let res = quote! {
        fn #ast() {
            println!("これはマクロで生成された関数☞
です: {}", stringify!(#ast));
        }
    };
    res.into()
}
```

マクロを呼び出すコードも示しておきます。

▶リストA.61 main.rs

```
macros::generate_func!(some_generated_☞
function);
some_generated_function();
// 結果: これはマクロで生成された関数です: ☞
some_generated_function
```

4. 以下のようなコードが書けていれば正解です。

▶リストA.62 traits.rs

```
pub trait DeriveItem {
    fn get_type(&self) -> String;
}
```

▶リストA.63 lib.rs

```
// deriveマクロMyItemを定義
#[proc_macro_derive(MyItem)]
pub fn my_item_derive(input: ☞
    TokenStream) -> TokenStream {
    // 入力となるアイテムのASTを取得
    let ast = parse_macro_input!(input as ☞
DeriveInput);
    // 識別子とアイテム本体を取得
    let name = &ast.ident;
    let data = &ast.data;
    // アイテム本体の種類で振り分け
    let res = match data {
        // 構造体の場合
        Data::Struct(_) => {
            quote! {
                impl DeriveItem for #name {
                    fn get_type(&self) -> String {
                        "構造体".to_string()
                    }
                }
            },
        // 列挙体の場合
        Data::Enum(_) => {
            quote! {
                impl DeriveItem for #name {
                    fn get_type(&self) -> String {
                        "列挙体".to_string()
                    }
                }
            },
        // 共用体の場合
        Data::Union(_) => {
            quote! {
                impl DeriveItem for #name {
                    fn get_type(&self) -> String {
                        "共用体".to_string()
                    }
                }
            },
        },
    };
    res.into()
}
```

```
};
// TokenStreamに変換
res.into()
}
```

回りくどいコードになっていますが、quote!マクロのブロック内部では、match式を使ってもアイテムの種類を判別できないためです。

第15章の解答

練習問題 15.1 P.567

1. 以下のようなコードが記述できていれば正解です。

▶リストA.64 practice-thread.rs

```
let list = Arc::new(Mutex::new(Vec::new()));
let mut handles = vec![];
for i in 1..=10 {
    let list_clone = Arc::clone(&list);
    let handle = thread::spawn(move || {
        let mut num = list_clone.lock().unwrap();
        num.push(i);
    });
    handles.push(handle);
}
for handle in handles {
    handle.join().unwrap();
}
println!("list: {:?}", *list.lock().unwrap());
```

Mutex<T>構造体とArc<T>構造体を使い、内包するデータをベクター (Vec<T>) とします。スレッドをfor式で起動していくので、JoinHandle<T>格納用のベクターを用意し、spawn関数の戻り値を追加していきます。Mutex<T>構造体の代わりにRwLock<T>構造体を使用し、lockメソッドの代わりにwriteメソッドとreadメソッドを使っても同様の結果が得られます。

練習問題 15.2 P.576

1. マルチスレッドは、処理を分散・並行して実行する仕組みで、複数のCPUコアを利用するなど処理の高速化が可能です。非同期処理は、ある処理の終了を待たずに別の処理を開始できる仕組みで、I/Oバウンドな処理がある場合などに有用です。

2. ① main ② async ③ await

asyncキーワードは非同期処理に付けるもの、awaitキーワードは非同期処理の終了を待機するものと覚えておきましょう。

練習問題 15.3 P.583

1. 以下のようなコードが記述できていれば正解です。

▶リストA.65 practice-cfg-attr.rs

```
#[cfg(target_os = "windows")]
{
    println!("Windows用にコンパイルされています。");
}
#[cfg(target_os = "macos")]
{
    println!("Mac用にコンパイルされています。");
}
#[cfg(target_os = "linux")]
{
    println!("Linux用にコンパイルされています。");
}
```

ここで、ブロック ({}) は必須ではありませんが、一般的にはコードは複数になることが多いので、そのような場合の例としてあります。

2. 以下のようなコードが記述できていれば正解です。

▶リストA.66 practice-cfg-macro.rs

```
if cfg!(target_os = "windows") {
    println!("Windows用にコンパイルされています。");
} else if cfg!(target_os = "macos") {
```

```
println!("Mac用にコンパイルされています。");
} else if cfg!(target_os = "linux") {
    println!("Linux用にコンパイルされていま
す。");
} else {
    println!("その他のOS用にコンパイルされてい
ます。");
}
```

cfg! マクロではif式を使えるため、else ifを使って条件を絞り込んでいくことができます。

この章の理解度チェック P.589

1. (×) スレッドはプログラムの実行単位の一つですが、メモリ空間などのリソースを共有します。
(○) 正しい記述です。
(×) スレッドで実行するクロージャはFnOnceトレイトの実装が必須です。
(×) スレッドが返す値を取得するにはjoinメソッドを使います。
(○) 正しい記述です。
(×) スレッド間の通信には多対1のmpsc (multi producer, single consumer) という仕組みを使います。

2. 以下の点を指摘できていれば正解です。

- 関数fetch_dataは非同期関数であるのでasyncキーワードが必要
- request::get関数とtextメソッドのあとにはawaitキーワードが必要
- main関数は非同期関数になる必要があるためtokio::main属性が必要

以上を修正したコードは以下の通りです。

▶リストA.67 practice-2.rs

```
async fn fetch_data() -> Result<String,
request::Error> {
    let response = request::get("https://
rust-lang.org").await?;
    let body = response.text().await?;
    Ok(body)
}
```

```
}

#[tokio::main]
async fn main() {
    println!("データを読み込んでいます...");
    match fetch_data().await {
        Ok(data) => println!("データを読み込み
ました: {data}"),
        Err(e) => eprintln!("データの読み込み
中にエラーが発生しました: {e}"),
    }
}
```

3. ① 宣言されているのに使われていない変数yがある
② 戻り値が使われていない関数awesome_functionがある
③ 定義されているのに使われていない関数great_functionがある
④ 絶対に到達しないコードがある

それぞれ、allow属性を削除すると以下のような警告が発生します。カッコ内は対応するリント項目のallow属性です。

- ① unused variable: `y` (#[warn(unused_variables)])
- ② unused return value of `awesome\_function` that must be used (#[warn(unused_must_use)])
- ③ function `great\_function` is never used (#[warn(dead_code)])
- ④ unreachable expression (#[warn(unreachable_code)])

4. 関数generate_strの戻り値は&str型です。関数の処理内容として、文字列リテラル"Hello, world!"をto_stringメソッドでString型に変換し、&str型を返す関数の仕様にに基づき、さらにas_strメソッドを呼び出しています。つまり、関数の戻り値は関数内で生成された文字列への参照となります。

しかしながら、この文字列は関数終了とともに消滅するので、関数の戻り値はダングリングポインター（不正なポインター）となります。これにより、コンパイラは許可できない参照としてエラーとするのです。