Functional Programming in Scala

impress top gear

Scalazコントリビューターによる
関数型徹底ガイド

Scala
関数型デザイン＆プログラミング

Paul Chiusano／Rúnar Bjarnason＝著
株式会社クイープ＝訳

洗練された副作用のない手法！

再利用／並列化／一般化／テストが容易になる

◎ データ構造/入出力/エラー処理を関数型で記述
◎ ライブラリのプログラムを設計・開発
◎ 関数型プログラミングの共通パターンを見抜く

インプレス

※本文中に登場する会社名、製品名、サービス名は、各社の登録商標または商標です。
※本書の内容は原著執筆時点のものです。本書で紹介した製品／サービスなどの名前や内容は変更される可能性があります。
※本書の内容に基づく実施・運用において発生したいかなる損害も、著者、訳者、ならびに株式会社インプレスは一切の責任を負いません。
※本文中では®、TM、©マークは明記しておりません。

Original English language edition published by Manning Publications, USA.
Copyright©2014 by Manning Publications Co.
Japanese-language edition Copyright © 2015 by Impress Corporation.
All rights reserved.

Japanese translation rights arranged with Waterside Productions, Inc. through Japan UNI Agency, Inc., Tokyo

本書に寄せて

原書のタイトルである『Functional Programming in Scala』は、好奇心をそそるタイトルです。何しろ、Scala は一般に関数型プログラミング言語と呼ばれており、Scala に関する本はそれこそ何十冊と出版されているのです。そうした他の本には、この言語の関数的な側面が欠けているというのでしょうか。この質問に答えるために、もう少し掘り下げてみましょう。**関数型プログラミング**とは何でしょうか。私にとっては、「関数を使ったプログラミング」の別名にすぎません。つまり、プログラム内の関数に焦点を合わせたプログラミングスタイルです。**関数**とは何でしょうか。これについては、もっと幅広い定義が見つかります。結果を返すことに加えて副作用を発生させることがある関数を大目に見ている定義もありますが、純粋関数型プログラミングでは、数学の場合と同じように、引数を結果にマッピングする 2 項関係が関数の定義となっています。

Scala では、純粋関数だけでなく非純粋関数も認められています。これらの 2 つの方法によって異なる構文を使用したり、異なる型を与えたりといった区別をしない点で、Scala は非純粋な関数型プログラミング言語です。これは他のほとんどの言語と共通する特性です。Scala で純粋関数と非純粋関数を区別できれば便利ですが、迷うことなく Scala に追加できるほど軽量かつ柔軟に区別する方法はまだ見つかっていません。

確かに、Scala プログラマには純粋関数を使用することが一般に奨励されています。変更処理（ミューテーション：参照値の変更）、I/O、例外の使用といった副作用は禁じられていませんし、相互運用性、効率性、利便性などの理由でそれらが非常に役立つことが実際にあるわけです。しかし、副作用を乱用することは、専門家からは一般によいスタイルと見なされていません。それはそうとして、非純粋プログラムを作成することが可能で、Scala で記述するのに都合がよいということもあり、命令型プログラミングの経験が長いとそうしたスタイルを維持する方向に向かいがちで、関数型プログラミングの考え方にわざわざ順応しようとはしないものです。実際、Scala をセミコロンのない Java のように記述しようと思えば、そうすることもできるわけです。

では、関数型プログラミングを Scala できちんと学ぶためには、Haskell などの純粋関数型言語に寄り道したほうがよいのでしょうか。このアプローチに対する論拠は、本書の登場によって非常に弱くなっています。

Paul と Rúnar が本書で見せているのは、平たく言うと、Scala を純粋関数型プログラミング言語として扱う方法です。変更可能な変数、例外、従来の入出力を含め、非純粋性の形跡はすべて取り除かれています。こうした便利なものをいっさい使用せずに意味のあるプログラムを書けるのか不思議に思うなら、本書を読む必要があります。Paul と Rúnar は、最初の原理から積み上げ、増分入出力まで拡張することで、すべての概念を純粋関数だけで実際に表現できることを証明しています。そして、それが可能であるだけでなく、美しいコードや計算の本質を見抜く力につながることも示しています。

本書は深く考えさせる本です。細部への注意を促し、プログラミングに対するあなたの考え方を否定するかもしれないからです。本書を読み、推奨されている練習問題を解くことで、純粋関数型言語とは何か、その言語で何を表現できるか、それにはどのようなメリットがあるかについて理解を深めることができるでしょう。

本書に関して特に気に入っているのは、すべてがこの 1 冊に含まれていることです。可能な限り単純な表現から始まり、すべての抽象概念を詳しく説明した後、それらの上にさらに抽象概念を組み上げていきます。ある意味、表紙の向こうに Scala の別世界が展開します。その世界では、ミュータブルな状態は存在せず、すべての関数が純粋です。よく使用される Scala ライブラリは主に、関数型のインターフェイスを持ちながらも部分的に命令型の実装に基づいているため、この理想郷に少し似つかわしくありません。私の考えでは、ミュータブルな状態を関数型のインターフェイスにカプセル化できることは、Scala の長所の 1 つです。しかし、その能力がよく誤用されることも事実です。それにはまって抜けられなくなっていることに気づいた場合は、本書が強力な毒消しになります。

Martin Odersky
Scala 言語設計者

まえがき

よいソフトウェアを記述するのは容易なことではありません。他のアプローチを使って何年も格闘してきた末に、私たちは関数型プログラミングに出会い、心を奪われました。関数型プログラミングのアプローチは他とは異なるものですが、その教義がプログラムを作成するための一貫した、合成可能な、美しい方法へといかにしてつながるのかを私たちは次第に理解するようになりました。

私たちはケンブリッジで定期的に開かれる Boston Area Scala Enthusiasts に参加していました。このグループが発足したばかりの頃は、何かもっとよいものを探し求めている Java プログラマが主な参加者でした。そうしたメンバーの多くは、関数型プログラミングを Scala で活用する方法について学ぶための明白な手段がないことに不満を訴えていました。それは私たちも同感でした。2 人とも、他の Scala プログラマや Haskell プログラマに相談したり教えてもらったりしながら関数型コードを書きまくり、雑誌の記事やブログの記事、書籍の情報を寄せ集めたりしながら、関数型プログラミングを手探りで学んでいました。何かもっと簡単な方法があるはずだという気がしてなりませんでした。2010 年 4 月に、Boston Area Scala Enthusiasts のまとめ役の 1 人である Nermin Šerifović が、Scala での関数型プログラミングをテーマにした本の執筆を提案しました。自分たちの学習経験から、書きたい本のイメージは明確にあったので、すぐに簡単にできるだろうと考えていました。結局 4 年以上もかかったわけですが、よい本ができたと考えています。これは私たちが関数型プログラミングを学んでいたときにあればよかったと思える本です。

私たちが関数型プログラミングに出会ったときの興奮が少しでも伝わることを願っています。

謝辞

本書の作成に参加してくれた多くの方々に感謝したいと思います。Boston Area Scala Enthusiasts からの友人である Nermin Šerifović には、私たちの心に本書の種を最初にまいてくれたことに感謝します。

Capital IQ のすばらしいチームには、彼らの支援と、本書のカリキュラムの最初のバージョンのベーステストを勇敢にも手伝ってくれたことに感謝しています。

この旅の友であった Tony Morris には、特に感謝したいと思います。Scala での関数型プログラミングの実践に Tony が果たしたさらに大きな貢献と同じように、本書の最初の段階でいまだに計り知れないほどの働きをしてくれました。

Martin には、すばらしいまえがきと、私たちの業界の再編を後押しするこの強力な言語を作成してくれたことに感謝しています。

本書の執筆中に、関数型プログラミングに関心を持つ Scala ユーザーの力強いコミュニティからいただいた励ましに感謝しています。レビューを担当してくれた方々、MEAP の読者、そして意見を寄せてくれたり、バグレポートやプルリクエストを送ってくれたその他全員に感謝しています。あなた方の協力がなければ、本書は現在の形にはなっていなかったでしょう。

デベロップメントエディターの Jeff Bleiel、グラフィックエディターの Ben Kovitz、テクニカル校正者の Tom Lockney、そしてさまざまな段階の原稿を読んでくれた以下の方々を含め、本書をよりよいものにするために協力してくれた Manning のすべてのスタッフに感謝しています：Ashton Hepburn、Bennett Andrews、Chris Marshall、Chris Nauroth、Cody Koeninger、Dave Cleaver、Douglas Alan、Eric Torreborre、Erich W. Schreiner、Fernando Dobladez、Ionut、G. Stan、Jeton Bacaj、Kai Gellien、Luc Duponcheel、Mark Janssen、Michael Smolyak、Ravindra Jaju、Rintcius Blok、Rod Hilton、Sebastien Nichele、Sukant Hajra、Thad Meyer、Will Hayworth、William E. Wheeler。

最後に、Sunita と Allison には、本書を実現させるための数年にわたる冒険の間、私たちを支えてくれたことにとても感謝しています。

本書について

本書は Scala についての本ではありません。本書は**関数型プログラミング**の入門書です。関数型プログラミングは、ソフトウェアを記述するための、原則主義の先鋭的なアプローチです。本書では、その目的を達成するための手段として Scala を使用していますが、本書の内容はどの言語でのプログラミングにも応用できます。本書を読み進めていくうちに、関数型プログラミングの概念をしっかりと理解し、純粋関数型プログラムをすらすら書けるようになり、ここで説明する内容を超えて、このテーマに関する新しい内容を吸収できるようになることが、ここでの目標となります。

本書の構造

本書は 4 つの Part で構成されています。Part I では、関数型プログラミングとは何かを正確に説明し、その基本的な概念を紹介します。Part I の各章では、小さな関数型プログラムを作成し、純粋関数型のデータ構造を定義し、エラーを処理し、状態を扱うなど、基本的なテクニックを簡単に説明します。

Part II は、これを土台とした**関数型の設計**に関するチュートリアルとなっています。実用的な関数型ライブラリの例をいくつか取り上げ、それらの設計に通じる思考プロセスを明らかにします。

Part II でライブラリに取り組むうちに、これらのライブラリが特定のパターンに従っていることと、一部重複していることが明らかになります。このことから、より汎用的なライブラリを作成するには、一段上の新しい抽象化が必要であることが判明します。

Part III では、そうした抽象化について説明します。これらはコードについて論理的に考えるための非常に強力なツールです。それらをマスターすれば、並はずれた生産性を持つプログラマになることも夢ではありません。

Part IV では、データベースやファイルの操作、動画の再生といった I/O を実行し、ミュータブルな状態を利用する実際のアプリケーションを完全に関数的な方法で作成できるようになることを目指して、残っている溝を埋めていきます。

本書では、本書の内容を習得するのに役立つように注意深く配置されたプログラミング練習問題が大きな柱となります。関数型プログラミングを理解するには、机上の理論を学ぶだけでは不十分です。テキストエディタを開いてコードを実際に書いてみる必要があります。本書で学んだ理論を仕事で実践できなければ意味がありません。

また、章ごとにチャプターノートが用意されており、その章に関連する議論と、参考文献のリンクのセクションが設けられています。チャプターノートは読者のコミュニティによって拡張されることを意図したもので、編集可能な Wiki（英語）として提供されています。

https://github.com/fpinscala/fpinscala/wiki

本書の対象読者

本書では、プログラミングをある程度経験している読者を対象としています。本書を執筆するにあたって、私たちの頭の中には、関数型プログラミングにすごく興味がある平均的な Java または C プログラマという読者のイメージがありました。しかし、どの程度の経験であろうと、何らかの言語を経験しているプログラマにうってつけの本であると考えています。

専門知識がどれくらいあるかは、意欲と好奇心ほど重要ではありません。関数型プログラミングはとても楽しい反面、能力が試されるテーマでもあります。これまでとは異なる考え方が要求されるかもしれないため、プログラマとしての経験が長いほど逆に難易度が高くなるかもしれません。プログラミングの経験が何年であろうと、初心に立ち返る覚悟が必要です。

Scala を事前に経験している必要はありませんが、本書では、Scala の構文や言語機能の説明にそれほど時間や紙面を割きません。それらについては、主に他の事柄を説明する流れの中で、できるだけ形式張らずに、そのつど紹介することにします。Scala についてそれだけ理解しておけば、練習問題を解くのに十分なはずです。Scala 言語についてさらに詳しく知りたい場合は、Scala に関する別の本を読むか、Scala 言語のドキュメントを調べてみてください。

http://scala-lang.org/documentation/books.html
http://scala-lang.org/documentation/

本書の読み方

本書は最初から順番に読み進める形式になっていますが、4 つの Part の並べ方に工夫があります。これらの Part は、Part の変わり目で読むのをやめ、そこで学んだことを実際の作業に応用してみた後、次の Part に戻って来れるように配置されています。たとえば Part IV の内容を最もよく理解できるのは、Part I、II、III で関数型のプログラミングに対する理解を徐々に深めていき、それらを自分のものにしたときです。Part III を読んだ後にひと休みし、それらの章の練習問題とは別に、自分でも関数型プログラムを作成してみるとよいかもしれません。もちろん、最終的にどうするかを決めるのはあなたです。

本書のほとんどの章は同じような構造になっています。新しい発想や手法を紹介し、例を使って説明し、練習問題をいくつか解く、という流れになっています。練習問題のソースコードをダウンロードし、各章を読みながらぜひ練習問題を解いてみてください。練習問題、ヒント、解答は本書の GitHub（英語）で提供されています。

https://github.com/fpinscala/fpinscala

また、Google の scala-functional グループ[※1] や、IRC の `#fp-in-scala` チャネル[※2] での議論にも参加してみてください。

練習問題では、それぞれの難易度と重要度が示されています。難易度が高いと思われる練習問題や、本文の内容を理解するのに必ずしも必要でない練習問題については、その旨を示してあります。**難問**というマークは、心の準備を促すための本書の試みです。それは私たちの推量にすぎないため、このマークが付いていない問題を難しく感じたり、このマークが付いている問題をやさしく感じることがあるかもしれません。

◇◇ EXERCISE 1 ◇◇

難問：「難問」というマークは難しいと想定される問題を示す。

本書全体に含まれているさまざまな例は、ただ読むのではなく、実際に試すことを意図したものです。本書を読む前に、Scala インタープリタを起動して、すぐに使用できる状態にしておいてください。例の内容を自分なりに変化させて、実際に試してみることをお勧めします。何かを理解するには、それを少し変化させて、結果にどのように影響するのかを確認してみるのが効果的です。

何らかのコードを実行または評価した結果を示すために、Scala インタープリタのセッションを示すことがあります。セッションの行の先頭には、インタープリタの `scala>` プロンプトが付いています。このプロンプトに続くコードをインタープリタに入力するか、貼り付けてください。その次の行は、インタープリタの応答を示しています。

```
scala> println("Hello, World!")
Hello, World!
```

※1　https://groups.google.com/forum/#!forum/scala-functional

※2　irc.freenode.net

コードの表記とダウンロード

コードリストと本文に登場するソースコードはすべて等幅フォントで示されています。Scala のキーワードは**太字の等幅フォント**で示されています。多くのコードリストには、重要な概念を説明する注釈の番号が追加されています。

本書のサンプルのソースコード、練習問題のコード、チャプターノートは、本書の GitHub サイトからダウンロードしてください。

https://github.com/fpinscala/fpinscal

期待値の設定

関数型プログラミングはソフトウェアの記述方法にあらゆるレベルで計り知れない影響を与えますが、それを築き上げるのには時間がかかります。そのプロセスは漸進的です。関数型プログラミングのすばらしさを最初の章から堪能できるとは期待しないでください。最初に説明される原理はかなり漠然としたもので、場合によっては単なる常識に思えるかもしれません。「関数型プログラミングとは知らずにやってきたことじゃないか」と思ったとしたら、それはすばらしいことです。まさにそういうことなのです。ほとんどのプログラマは、そうとは知らずに、すでに関数型プログラミングを実践しています。関数の責任を 1 つだけにする、データをイミュータブルにするなど、ほとんどの人がベストプラクティスと考えているものの多くは、関数型プログラミングの前提を受け入れた結果です。本書では、そうしたベストプラクティスの根底にある原理を、論理的帰結に導くだけです。

本書を読むことで、Scala の構文と関数型プログラミングを一度に学べる可能性は大いにあります。このため、最初はコードがかなり異質なものに見えたり、手法が不自然に思えたり、練習問題が理解不能に思えるかもしれません。行き詰ってしまった場合は、ヒントや解答を調べるか、Google の scala-functional グループや IRC の `#fp-in-scala` チャネルで質問してみてください。

何はさておき、本書が読者にとって楽しく実りある体験となることを願っています。また、関数型プログラミングにより、私たちが経験したように、あなたの仕事がより簡単で楽しいものになることも願っています。本書の目的は、読者の仕事の生産性を高めることに尽きます。自分が作成しているソフトウェアをダーティハック（dirty hacks）の寄せ集めのように感じることが少なくなり、美しく実用的なものを作り上げているのだという実感がわいてくるはずです。

目次

本書に寄せて iii
まえがき v
謝辞 vi

Part I　関数型プログラミングの基礎

第 1 章　関数型プログラミングとは　3

1.1　関数型プログラミングの利点：単純な例 4
1.1.1　副作用を持つプログラム 4
1.1.2　関数型プログラミングのソリューション：副作用の排除 7
1.2　関数とはいったい何か 10
1.3　参照透過性、純粋性、置換モデル 11
1.4　まとめ 14

第 2 章　Scala 関数型プログラミングの準備　17

2.1　速習：Scala 言語 18
2.2　プログラムの実行 21
2.3　モジュール、オブジェクト、名前空間 23
2.4　高階関数：関数に関数を渡す 24
2.4.1　関数型のループ 24
2.4.2　高階関数の作成 27

2.5 多相関数：型の抽象化 28
2.5.1 多相関数の例 29
2.5.2 無名関数を使った高階関数の呼び出し 30
2.6 型に従う実装 32
2.7 まとめ 35

第３章 関数型プログラミングのデータ構造 37

3.1 関数型データ構造の定義 37
3.2 パターンマッチング 41
3.3 関数型データ構造でのデータ共有 44
3.3.1 データ共有の効率性 45
3.3.2 高階関数の型推論の改善 46
3.4 リストの再帰と高階関数の一般化 48
3.4.1 リストを扱うその他の関数 52
3.4.2 より単純なコンポーネントからリスト関数を組み立てるときの非効率性 54
3.5 ツリー 55
3.6 まとめ 57

第４章 例外を使わないエラー処理 59

4.1 例外の光と影 59
4.2 例外に代わる手法 62
4.3 Option データ型 64
4.3.1 Option の使用パターン 65
4.3.2 Option の合成、リフト、例外指向の API のラッピング 69
4.4 Either データ型 74
4.5 まとめ 78

第５章 正格と遅延 79

5.1 正格関数と非正格関数 80

5.2 遅延リストの例 84
5.2.1 ストリームを記憶し、再計算を回避する 85
5.2.2 ストリームを検査するためのヘルパー関数 86
5.3 プログラムの記述と評価の切り分け 87
5.4 無限ストリームと余再帰 91
5.5 まとめ 95

第 6 章 純粋関数型の状態 97

6.1 副作用を使った乱数の生成 97
6.2 純粋関数型の乱数の生成 99
6.3 ステートフル API の純粋化 101
6.4 状態の処理に適した API 104
6.4.1 状態アクションの結合 105
6.4.2 入れ子の状態アクション 106
6.5 状態アクションデータ型の一般化 109
6.6 純粋関数型の命令型プログラミング 110
6.7 まとめ 113

Part II 関数型デザインとコンビネータライブラリ

第 7 章 純粋関数型の並列処理 117

7.1 データ型と関数の選択 118
7.1.1 並列計算のためのデータ型 120
7.1.2 並列計算の結合 123
7.1.3 明示的なフォーク 125
7.2 表現の選択 127
7.3 API の改良 129
7.4 API の代数 135
7.4.1 マッピングの法則 135

7.4.2　フォークの法則 137
7.4.3　法則への違反：バグ 138
7.4.4　アクターを使った Par の完全なノンブロッキング実装 141
7.5　コンビネータを最も汎用的な形式に改良する 148
7.6　まとめ 152

第 8 章　プロパティベースのテスト　153

8.1　速習：プロパティベースのテスト 153
8.2　データ型と関数の選択 156
8.2.1　API：最初のコード 157
8.2.2　プロパティの意味と API 158
8.2.3　ジェネレータの意味と API 161
8.2.4　生成された値に依存するジェネレータ 162
8.2.5　Prop データ型の改良 163
8.3　テストケースの最小化 166
8.4　ライブラリの使用とユーザビリティの改善 168
8.4.1　単純な例 169
8.4.2　並列処理のためのテストスイートの作成 170
8.5　高階関数のテストと今後の展望 176
8.6　ジェネレータの法則 178
8.7　まとめ 179

第 9 章　パーサーコンビネータ　181

9.1　代数の設計から始める 182
9.2　代数の例 188
9.2.1　スライスと空ではない繰り返し 191
9.3　文脈依存への対処 194
9.4　JSON パーサーの作成 196
9.4.1　JSON の書式 197

9.4.2　JSON パーサー 197
9.5　エラー報告 198
9.5.1　設計の例 200
9.5.2　入れ子のエラー 201
9.5.3　分岐とバックトラックの制御 202
9.6　代数の実装 205
9.6.1　実装の例 206
9.6.2　パーサーの逐次化 207
9.6.3　パーサーのラベル付け 208
9.6.4　フェイルオーバーとバックトラック 209
9.6.5　文脈依存の解析 210
9.7　まとめ 212

Part III　関数型デザインに共通する構造

第 10 章　モノイド　215
10.1　モノイドとは 216
10.2　モノイドによるリストの畳み込み 219
10.3　結合性と並列性 220
10.4　並列解析の例 222
10.5　畳み込み可能なデータ構造 225
10.6　モノイドの合成 226
10.6.1　より複雑なモノイドの構築 227
10.6.2　合成されたモノイドを使った走査の融合 228
10.7　まとめ 229

第 11 章　モナド　231
11.1　ファンクタ：map 関数の一般化 231
11.1.1　ファンクタ則 233

11.2 モナド：flatMap と単位関数の一般化 235
11.2.1 モナドトレイト 236
11.3 モナドコンビネータ 239
11.4 モナド則 240
11.4.1 結合律 240
11.4.2 特定のモナドに対する結合律の証明 242
11.4.3 同一律 244
11.5 モナドとはいったい何か 246
11.5.1 単位元モナド 247
11.5.2 State モナドと部分的な型の適用 248
11.6 まとめ 252

第 12 章 アプリカティブファンクタとトラバーサブルファンクタ 253

12.1 モナドの一般化 253
12.2 アプリカティブトレイト 254
12.3 モナドとアプリカティブファンクタの違い 257
12.3.1 Option アプリカティブと Option モナド 257
12.3.2 Parser アプリカティブと Parser モナド 259
12.4 アプリカティブファンクタの利点 260
12.4.1 モナドではないアプリカティブファンクタ 261
12.5 アプリカティブの法則 264
12.5.1 左単位元と右単位元 265
12.5.2 結合性 265
12.5.3 積の自然性 267
12.6 トラバーサブルファンクタ 269
12.7 Traverse の使用 271
12.7.1 モノイドからアプリカティブファンクタへ 272
12.7.2 State を使った走査 273
12.7.3 トラバーサブルな構造の結合 275

12.7.4　走査の融合 277
12.7.5　入れ子の走査 277
12.7.6　モナドの合成 277
12.8　まとめ 279

Part IV　作用と I/O

第 13 章　外部作用と I/O　283

13.1　作用のリファクタリング 283
13.2　単純な IO 型 285
13.2.1　入力作用の処理 287
13.2.2　単純な IO 型の長所と短所 291
13.3　StackOverflowError の回避 293
13.3.1　制御フローをデータコンストラクタとして具体化する 293
13.3.2　トランポリン化：スタックオーバーフローに対する一般的な解決策 296
13.4　より高機能な IO 型 298
13.4.1　フリーモナド 300
13.4.2　コンソール I/O だけをサポートするモナド 301
13.4.3　純粋なインタープリタ 305
13.5　ノンブロッキングの非同期 I/O 307
13.6　汎用目的の IO 型 310
13.6.1　宇宙の果てのメインプログラム 310
13.7　なぜ IO 型では I/O のストリーミングに不十分なのか 311
13.8　まとめ 314

第 14 章　局所作用とミュータブルな状態　315

14.1　純粋関数型の変更可能な状態 315
14.2　副作用のスコープを適用するためのデータ型 317
14.2.1　変更処理にスコープを適用するための小さな言語 318

14.2.2　ミュータブルな参照の代数 320
14.2.3　ミュータブルな状態アクションの実行 321
14.2.4　ミュータブルな配列 325
14.2.5　純粋関数型の quicksort 327
14.3　純粋性とコンテキスト 328
14.3.1　副作用と見なされるのは何か 330
14.4　まとめ 331

第 15 章　ストリーム処理とインクリメンタル I/O　333

15.1　命令型 I/O の問題：例 333
15.2　単純なストリームトランスデューサ 337
15.2.1　プロセスの作成 339
15.2.2　プロセスの合成と追加 343
15.2.3　ファイルの処理 345
15.3　拡張可能な Process 型 346
15.3.1　ソース 350
15.3.2　リソースの安全性の確保 352
15.3.3　単一入力の処理 355
15.3.4　複数の入力ストリーム 358
15.3.5　シンク 361
15.3.6　エフェクトフルなチャネル 363
15.3.7　リソースの動的な割り当て 364
15.4　応用 365
15.5　まとめ 366

索引 367

Part I

関数型プログラミングの基礎

本書では、プログラムの構築に純粋関数だけを使用することにより、ファイルからの読み取りやメモリの書き換えといった副作用の発生を阻止することを根本的な前提としています。この関数型プログラミングという考え方により、プログラムの記述方法はこれまでとは大きく異なることになります。そこで本書では、最も簡単なプログラムを関数型プログラミングで記述する方法から説明していくことにします。

第1章では、関数型プログラミングの意味を正確に説明し、その利点がどのようなものであるかを理解してもらいます。Part Iの残りの章では、Scalaでの関数型プログラミングの基本的なテクニックを紹介します。第2章ではScala言語を紹介し、関数型プログラミングでループを記述する方法や、関数を「値」として操作する方法など、基礎的な部分を取り上げます。第3章では、いつ変化してもおかしくないメモリ内のデータ構造を取り上げます。第4章では、純粋関数でのエラー処理を取り上げ、第5章では、非正格性の概念を紹介します。非正格性は、関数型コードの効率性とモジュール性を向上させるための概念です。第6章では、純粋関数を使ったステートフルプログラムのモデリングを紹介します。

プログラムの入力から出力までを完全に関数として捉えてもらうことが、Part Iの目的となります。実際のコードの記述に取りかかるときに必要となるテクニックについては、Part IIで説明します。

1

関数型プログラミングとは

関数型プログラミングでは、**純粋関数**（pure function）だけを使ってプログラムを構築することが前提となります。純粋関数とは、**副作用**のない関数のことです。この前提は単純ながら大きな影響力を持ちます。副作用とは何でしょうか。たとえば以下に示すように、単に結果を返すこと以外に何かをする関数には、副作用があることになります。

- 変数を変更する
- データ構造を直接変更する
- オブジェクトのフィールドを設定する
- 例外をスローする、またはエラーで停止する
- コンソールに出力する、またはユーザー入力を読み取る
- ファイルを読み取る、またはファイルに書き込む
- 画面上に描画する

より正確な副作用の定義については後ほど説明するとして、こうしたアクションを実行する能力を持たない、あるいはこうしたアクションを実行できるタイミングや方法がかなり制限されたプログラミングがどのようなものであるか想像してみてください。想像するのは難しいかもしれませんね。副作用がなく意味のあるプログラムを記述することはそもそも可能なのでしょうか。変数の再代入が不可能であるとしたら、ループのような単純なプログラムをどのようにして記述するのでしょうか。変化するデータの扱いはどうなるのでしょうか。例外をスローせずにエラーを処理することはどうでしょうか。画面への描画やファイルからの読み取りといったI/Oを実行しなければならないプログラムはどのようにして記述するのでしょうか。

要するに、関数型プログラミングはプログラムの書き方を制約するものです。ただし、表現可能なプログラムの「種類」を制約するものではありません。本書では、すべてのプログラムを副作用なしで表現する方法について説明します。これには、I/O を実行し、エラーを処理し、データを変更するプログラムも含まれます。純粋関数を使ったプログラミングによって**モジュール性**が向上することを考えれば、関数型プログラミングのルールに従うことには大きなメリットがある、ということを具体的に示します。純粋関数にはモジュール性があるため、テスト、再利用、並列化、一般化、推論が容易になります。さらに、バグが発生しにくくなるという効果もあります。

本章では、副作用を持つ単純なプログラムを取り上げ、それらの副作用を取り除くことで、関数型プログラミングの利点を具体的に示します。また、関数型プログラミングの利点をより俯瞰的に捉え、**参照透過性**と**置換モデル**という 2 つの重要な概念を定義します。

1.1　関数型プログラミングの利点：単純な例

ここでは、純粋関数を使ったプログラミングの利点を示す例を見ながら、本書全体の原点となる概念を示すことにします。Scala の構文を見るのは初めて、という読者もいるでしょう。Scala の構文については次章で詳しく説明するので、細かい部分がよくわからなくても心配はいりません。コードが何をするものなのかが基本的に理解できていれば、それで十分です。

1.1.1　副作用を持つプログラム

コーヒーショップでの注文を処理するプログラムを実装しているとしましょう。まず、実装において副作用を利用する Scala プログラムから見ていきます。このようなプログラムは、**非純粋**プログラムとも呼ばれます。

リスト 1-1　副作用を持つ Scala プログラム

```
class Cafe {  ❶

  def buyCoffee(cc: CreditCard): Coffee = {  ❷

    val cup = new Coffee()  ❸

    cc.charge(cup.price)  ❹

    cup  ❺
  }
}
```

❶ Java の場合と同様に、`class` キーワードを使ってクラスを定義する。クラスの本体は中括弧（`{}`）で囲む。

❷ クラスのメソッドは def キーワードで定義する。cc: CreditCard は cc という名前の CreditCard 型のパラメータを定義している。buyCoffee メソッドの戻り値の型である Coffee はパラメータリストの後に指定される。メソッドの本体は = 記号に続く中括弧で囲まれたブロックで構成される。

❸ セミコロン (;) は不要。ブロック内のステートメントは改行で区切られる。

❹ 副作用：クレジットカードにチャージしている。

❺ return を指定する必要はない。cup はブロック内の最後のステートメントなので、自動的に返される。

cc.charge(cup.price) は副作用の一例です。クレジットカードにチャージするには、外部とのやり取りが必要になります。Web サービスを通じてクレジットカード会社へ問い合わせ、カード決済を承認し、カードに課金し、あとから参照するための決済記録を保存する必要があるとしましょう。ただし、リスト 1-1 の関数自体は Coffee オブジェクトを返すだけで、これらは副次的に発生するアクションです。「副作用」と呼ばれるのは、そのためです。副作用の正式な定義については、後ほど改めて説明します。

このような副作用があると、コードがテストしにくくなります。テストの際にクレジットカード会社に実際に問い合わせてクレジットカードにチャージするのは避けたいところです。テスタビリティに問題があるということは、設計を変更したほうがよいということです。クレジットカード会社に問い合わせて実際にチャージする方法を CreditCard に組み込むべきではないことはもちろん、チャージ記録を内部システムに保存する方法を CreditCard に認識させるべきではないこともほぼ間違いないでしょう。これらを CreditCard に認識させるのではなく、buyCoffee に Payments オブジェクトを渡すようにすれば、コードのモジュール性とテスタビリティを向上させることができます。

リスト 1-2　payments オブジェクトの追加

```
class Cafe {
  def buyCoffee(cc: CreditCard, p: Payments): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)
    cup
  }
}
```

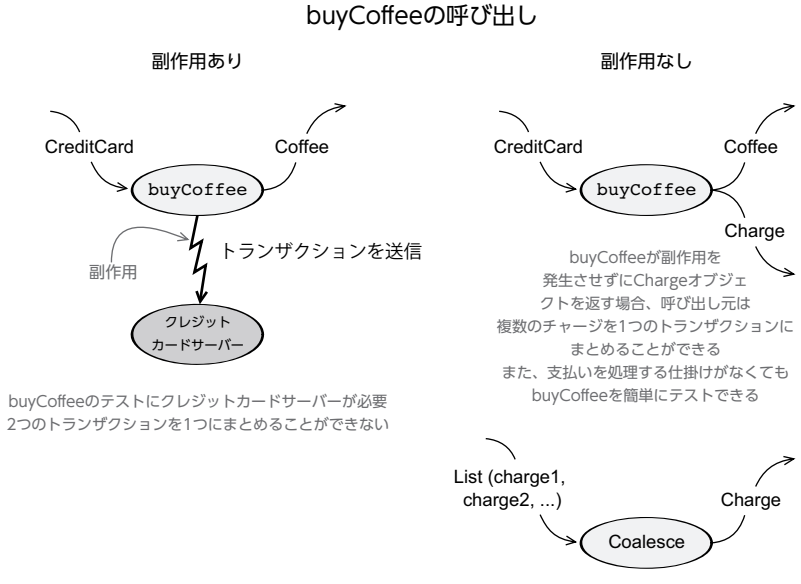
p.charge(cc, cup.price) の呼び出しには依然として副作用がありますが、少なくともテスタビリティが少し改善されたことは確かです。Payments はインターフェイスでもかまいません。その場合は、テストしやすいようにインターフェイスのモックを実装すればよいわけです。ただ

し、それも最適であるとは言えません。具体的なクラスでも問題がないようなケースであっても Payments をインターフェイスにせざるをえなくなることを考えると、モック実装は必ずしも使いやすいとは言えないからです。たとえば、buyCoffee の呼び出し後に調べなければならない内部状態が実装に含まれているとしたら、その状態が charge の呼び出しによって適切に変更されていることをテストで確認する必要があります。モックフレームワークなどを利用すれば、こうした詳細を自動的に処理できることは確かですが、buyCoffee がコーヒー 1 杯分をきっちり請求することをテストするだけなら、そこまでする必要はないでしょう。

問題はテストだけではありません。buyCoffee メソッドの再利用が難しくなることも問題です。Alice が店にやってきて、コーヒーを 12 杯注文するとしましょう。理想的には、buyCoffee をループで 12 回呼び出すなどして、このメソッドを再利用できるようにしたいところです。しかし、ここまでの実装では、決済システムに 12 回も問い合わせて、Alice のクレジットカードに対する 12 件のチャージを別々に承認する必要があります。これでは処理のコストがかさむだけで、Alice にとってもコーヒーショップにとってもいい迷惑です。

この問題にどう対処すればよいでしょうか。図 1-1 に示すように、チャージを一括処理する特別なロジックを使って、buyCoffees というまったく新しい関数を記述するという方法があります[※1]。

図 1-1：課金を一括処理するロジック



※1　Payments インターフェイスの特別な実装として BatchingPayments を作成し、同じクレジットカードへの一連のチャージを一括処理するという方法もあるが、かなりややこしいことになる。この処理の対象となるチャージの数と、それにかかる時間を考慮に入れなければならない。closeBatch を呼び出すなどして、一括処理が完了したことを buyCoffee が知らせるのだろうか。ましてや、そのタイミングをどのようにして知るのだろうか。

このケースでは、buyCoffee のロジックが非常に単純なのでたいした違いはないかもしれませんが、一筋縄ではいかないロジックが繰り返し必要になるようなケースでは、コードの再利用と合成が不可能であることがネックになるでしょう。

1.1.2　関数型プログラミングのソリューション：副作用の排除

関数型プログラミングのソリューションでは、これらの副作用を取り除き、buyCoffee が Coffee を返すことに加えて、チャージを値として返すようにします。チャージをクレジットカード会社に送信し、その記録を保存するなどの処理は、別の場所で行われます。Scala の構文についてはこの後の章で説明しますが、ここで関数型プログラミングのソリューションがどのようなものであるか見ておきましょう。

```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {  ❶
    val cup = new Coffee()
     (cup, Charge(cc, cup.price))  ❷
  }
}
```

❶ buyCoffee が戻り値として Coffee と Charge を返すことが (Coffee, Charge) という型で示されている。支払いを処理するシステムは、ここではまったく関与しない。

❷ Coffee と Charge のペアを作成するには、cup と Charge を括弧で囲み、コンマで区切る。

チャージの作成をそのチャージの処理または解釈から切り離していることがわかります。buyCoffee メソッドは、Coffee に加えて Charge も値として返すようになっています。後ほど見ていくように、このようにすると、複数のコーヒーの購入を 1 つのトランザクションで処理するときに再利用しやすくなります。しかし、Charge とは何でしょうか。Charge はここで考え出されたデータ型です。Charge クラスは CreditCard と amount を含んでおり、同じ CreditCard に対するチャージをまとめるための combine という便利な関数を備えています。

```
case class Charge(cc: CreditCard, amount: Double) {  ❶
  def combine(other: Charge): Charge =
    if (cc == other.cc)  ❷
      Charge(cc, amount + other.amount)  ❸
    else
      throw new Exception("Can't combine charges to different cards")  ❹
}
```

❶ case class はプライマリコンストラクタ（プライマリ構築子）を 1 つ持つ。プライマリコンストラクタの引数リストはクラス名（この場合は Charge）の後に指定される。引数リストのパラメータは class の変更不能（イミュータブル）なパブリックフィールドとなる。これらのフィールドには、other.cc のように、通常のオブジェクト指向のドット表記を使ってアクセスできる。

❷ if 式の構文は Java のものと同じだが、選択された分岐の結果に等しい値も返される。cc == other.cc の場合、combine は Charge(..) を返す。それ以外の場合は、else 分岐で例外がスローされる。

❸ case class は new キーワードを使わずに作成できる。クラス名に続いてプライマリコンストラクタの引数リストを指定するだけでよい。

❹ 例外をスローする構文は Java やその他多くの言語のものと同じである。関数型プログラミングでエラー状態を処理する方法については、この後の章で説明する。

n 杯のコーヒーの購入を実装するための buyCoffees はリスト 1-3 のようになります。ここまでのコードとは異なり、buyCoffee をベースとした実装が可能となっています。

リスト 1-3　buyCoffees による複数の購入

```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = ...

  def buyCoffees(cc: CreditCard, n: Int): (List[Coffee], Charge) = {  ❶
    val purchases: List[(Coffee, Charge)] = List.fill(n)(buyCoffee(cc))  ❷
    val (coffees, charges) = purchases.unzip  ❸
     (coffees, charges.reduce((c1,c2) => c1.combine(c2)))  ❹
  }
}
```

❶ List[Coffee] は Coffee 値のイミュータブルな単方向リストである。このデータ型については、第 3 章で詳しく説明する。

❷ List.fill(n)(x) は x のコピーを n 個含んだ List を作成する。このおもしろい関数呼び出し構文については、この後の章で説明する。

❸ unzip は、複数のペアからなる 1 つのリストを 2 つのリストのペアに分割する。ここでは、ペア構造を分解することにより、2 つの値（coffees、charges）を 1 行で宣言している。

❹ charges.reduce は、combine を使って 2 つのチャージを結合することで、チャージリスト全体を 1 つのチャージに変換する。reduce は**高階関数**の一種である。高階関数の正確な定義については次章で説明する。

全体的に見て、このソリューションは顕著な改善であると言えるでしょう。buyCoffees 関数を定義するにあたって buyCoffee を直接再利用できることに加えて、Payments インターフェイ

スの複雑なモック実装を定義しなくても、どちらの関数も簡単にテストできます。それどころか、Charge 値がどのように処理されるのかについて Cafe はまったく関知しません。もちろん、チャージを実際に処理するために Payments を引き続き使用することは可能ですが、Cafe がそのことについて知る必要はありません。

Charge をファーストクラスの値にする（関数を一種の値として扱う）ことには、予想外のメリットもあります。それは、こうしたチャージを処理するためのビジネスロジックを組み立てやすくなることです。たとえば、Alice がコーヒーショップにノート PC を持ち込み、そこで数時間ほど作業しながらコーヒーを何杯か注文するとしましょう。こうした複数の注文をコーヒーショップが 1 つのチャージにまとめ、クレジットカード処理の手数料を節約できれば便利かもしれません。Charge はファーストクラスの値なので、同じクレジットカードへのチャージを List[Charge] にまとめるための関数を以下のように定義できます。

```
def coalesce(charges: List[Charge]): List[Charge] =
  charges.groupBy(_.cc).values.map(_.reduce(_ combine _)).toList
```

ここでは、groupBy、map、reduce の 3 つのメソッドに関数を値として渡しています。このようなワンライナー（1 行プログラム）を読み書きする方法については、この後の章で説明します。_.cc と _ combine _ は**無名関数**（anonymous function）です。

表記がそっけなさすぎて、このようなコードを読みにくいと感じることがあるかもしれません。ですが、本書を読み進めていくうちに、こうした Scala コードの読み書きをすぐにマスターするはずです。この関数は、チャージのリストを受け取り、使用されているクレジットカードごとにそれらをグループ化することで、請求金額をクレジットカードごとに集計します。再利用やテスタビリティに関しても、モックオブジェクトやインターフェイスを追加する必要はまったくありません。同じロジックを buyCoffee の最初の定義で実装しようとしたらどうなるか想像してみてください。

これは、関数型プログラミングに触れ込みどおりの利点があることを示す理由の 1 つにすぎず、この例は意図的に単純なものとなっています。ここで使用している一連のリファクタリングが、自然な手法、あるいはごくあたりまえの標準的な手法に思えるとしたら、それは願ってもないことです。関数型プログラミングは、多くの人がよいと考えるものをその論理的なエンドポイントに取り入れるための規則にすぎず、その適応性がそれほど明白ではなかったとしてもその規則が適用されます。これから説明していくように、関数型プログラミングの規則に一貫して従うことによる効果は絶大で、計り知れないメリットをもたらします。関数型プログラミングは、最も単純なループから高度なプログラムアーキテクチャまで、あらゆるレベルでプログラムの構成方法を根本的に変化させます。そこから生まれるスタイルはまったく異なるものとなりますが、それは美しくまとまりのあるプログラミングアプローチであり、その真価を解することを願ってやみません。

現実の世界はどうか

buyCoffee の例では、Charge の作成を Charge の解釈または処理からどのようにして切り離すのかを示しました。本書では、副作用を持つすべての関数にこの種の変換を適用することで、そうした副作用をプログラムの外側の層に追いやる方法について説明します。関数型プログラミングを使用するプログラマはよく、純粋なコアでのプログラミングと、副作用を処理する外側の薄い層でのプログラミングを引き合いに出します。

ですがそうであったとしても、Charge を処理するためには、何らかの時点で Charge を外部システムに渡さなければならないことは確実です。それに、副作用や変化が避けられないプログラムは他に存在しないのでしょうか。そのようなプログラムをどのように記述すればよいのでしょうか。本書を読み進めていくうちに、副作用を必要とするように思える、同じような機能を持つプログラムがいくつも存在することがわかるでしょう。また、副作用を持つものの、それらが見えないようにコードを構造化する方法があることもわかるでしょう。たとえば、関数の本体でローカルに宣言されたデータを関数の外側から参照できないようにすれば、そのデータは変更可能になります。また、外側の関数から見えない限り、ファイルへの書き込みを行うことも可能です。

1.2　関数とはいったい何か

先に述べたように、関数型プログラミングは純粋関数を使用するプログラミングを意味します。純粋関数とは、副作用を持たない関数のことです。コーヒーショップの例では、副作用と純粋性の概念をざっと紹介しました。ここでは、この概念の正式な定義を示し、関数型プログラミングを使用することの意味をより正確に伝えたいと思います。それにより、関数型プログラミングの利点の1つである「純粋関数が推論しやすいこと」も見抜けるようになるでしょう。

入力型が A、出力型が B の関数 f は、A 型のすべての値 a を B 型の 1 つの値 b に関連付け、b が a の値によってのみ決定されるようにする計算です。Scala では、これを A => B という単一の型として記述します[※2]。内部プロセスまたは外部プロセスの状態の変化は、計算結果 f(a) とは無関係です。たとえば、Int => String 型を使用する関数 intToString は、すべての整数を対応する文字列に変換します。さらに、それが実際に**関数**であるとしたら、それ以上のことは何も行いません。

つまり、入力に基づいて結果を計算することを除いて、関数がプログラムの実行に対して観察可能な効果を与えることはありません。要するに、副作用はありません。このことをより明白にするために、そうした関数を**純粋関数**と称することもありますが、それでは少しくどくなってしまいます。そこで特に明記しない限り、本書では副作用がないという意味で**関数**を使用します[※3]。

※2　「A から B」または「A 矢印 B」と読む。

※3　副作用があるかもしれないパラメータ化されたコードブロックを**プロシージャ**と呼ぶこともある。

純粋関数の例として、整数の加算（+）関数について考えてみましょう。加算関数は、整数値を 2 つ受け取り、整数値を 1 つ返します。この関数は、特定の 2 つの整数値に対し、常に**同じ整数値**を返します。また、文字列を変更できない —— 文字列がイミュータブルである —— Java や Scala などの言語では、`String` の `length` 関数も純粋関数の 1 つです。この関数は、与えられた文字列に対して常に同じ長さを返すだけで、それ以外は何もしません。

この純粋関数という考え方を、**参照透過性**（referential transparency）の概念を用いて正式な定義にしてみましょう。これは関数だけでなく、一般の**式**の特性でもあります。式については、さしあたり「1 つの結果として評価できるプログラムの任意の部分」として考えてください —— Scala インタープリタに入力して答えを得ることができるあらゆる部分がこれに該当します。たとえば `2 + 3` は、純粋関数 `+` を `2` と `3` の値に適用する式であり、`2` と `3` の値も式です。この場合、副作用はありません。この式の評価は常に `5` の値になります。実際には、プログラムに `2 + 3` が含まれている場合、その部分を `5` の値に置き換えてもプログラムの意味は変わりません。

式が参照透過であるとは、こういうことです。どのようなプログラムにおいても、プログラムの意味を変えることなく、式をその結果に置き換えることができる —— これが参照透過性です。また、参照透過な引数による呼び出しが参照透過になる関数を、**純粋関数**と呼びます。

> **参照透過性と純粋性**
>
> 式 `e` があり、すべてのプログラム `p` において、`p` の意味に影響を与えることなく、`p` 内のすべての `e` を `e` の評価結果と置き換えることができるとしたら、`e` は**参照透過**です。関数 `f` があり、式 `f(x)` が参照透過なすべての `x` に対して参照透過であるとしたら、`f` は**純粋関数**です ※4。

1.3　参照透過性、純粋性、置換モデル

参照透過性の定義は、たとえば最初の `buyCoffee` のコードに対してどのように適用されるでしょうか。

```
def buyCoffee(cc: CreditCard): Coffee = {
  val cup = new Coffee()
  cc.charge(cup.price)
  cup
}
```

※4　この定義には微妙な点がいくつかあるため、後ほど再定義する。詳細については、本書の GitHub サイトのチャプターノートを参照。
https://github.com/fpinscala/fpinscala/wiki

cc.charge(cup.price) の戻り値は、その内容にかかわらず、buyCoffee によって削除されます（この戻り値の型はおそらく、他の言語の void に相当する Scala の Unit になるでしょう）。このため、buyCoffee(aliceCreditCard) の評価結果は、単に new Coffee() に相当する cup になります。先の参照透過性の定義によれば、buyCoffee が純粋であるためには、p(buyCoffee(aliceCreditCard)) が任意の p に対して p(new Coffee()) と同じ振る舞いをしなければなりません。これが当てはまらないことは明らかです —— new Coffee() プログラムが何もしないのに対し、buyCoffee(aliceCreditCard) はクレジットカード会社に問い合わせてチャージを承認します。この２つのプログラムにはすでに明白な違いがあります。

参照透過性では、関数の結果型に従い、「関数が実行するすべてのことがその戻り値によって表される」という不変条件が適用されます。この制約により、**置換モデル**と呼ばれるプログラム評価の単純かつ自然な論証が可能となります。式が参照透過である場合は、代数方程式を解く場合と同じように計算を進めることができるものと考えられます。式のすべての部分が完全に展開され、すべての変数がそれらの参照先と置き換えられた後、最も単純な形式に変換されます。その際には、手順ごとに項をそれに相当するものに置き換えること —— つまり、**等価による置換**に基づいて計算が進められます。要するに、参照透過性によりプログラムの**等式推論**（equational reasoning）が可能になります。

さらに例を２つ見てみましょう。一方の例では、すべての式が参照透過で、置換モデルを使った推論が可能です。もう一方の例には、参照透過性に違反する式が含まれています。といっても、すでに理解していると思われるものを形式化しているだけで、難しいことは何もありません。

Scala インタープリタ[※5] で以下のコードを試してみましょう。Java と Scala では、文字列がイミュータブルであることに注意してください。「変更された文字列」は、実際には新しい文字列であり、古い文字列はそのまま残されています。

```
scala> val x = "Hello, World"
x: String = Hello, World

scala> val r1 = x.reverse
r1: String = dlroW ,olleH

scala> val r2 = x.reverse
r2: String = dlroW ,olleH
```

この場合、r1 と r2 は同じです。

ここで、すべての x を、x が参照している式（その**定義**）と置き換えてみましょう。

※5　Scala インタープリタは REPL（Read-Eval-Print-Loop）とも呼ばれる。「リプル」と読む。

```
scala> val r1 = "Hello, World".reverse
r1: String = dlroW ,olleH

scala> val r2 = "Hello, World".reverse
r2: String = dlroW ,olleH
```

この変換は結果に影響を与えません。r1 と r2 の値が以前と同じであることから、x は参照透過です。さらに、r1 と r2 も参照透過であるため、これらがもっと大きなプログラムの一部に含まれていた場合、それらすべてを値と置き換えたとしてもプログラムへの影響はありません。

今度は、参照透過ではない関数として、java.lang.StringBuilder クラスの append 関数について考えてみましょう。この関数は StringBuilder を直接操作します。StringBuilder の以前の状態は、append の呼び出しの後に削除されます。以下のコードを試してみましょう。

```
scala> val x = new StringBuilder("Hello")
x: StringBuilder = Hello

scala> val y = x.append(", World")
y: StringBuilder = Hello, World

scala> val r1 = y.toString
r1: String = Hello, World

scala> val r2 = y.toString
r2: String = Hello, World
```

r1 と r2 は同じです。ここまではいいでしょう。では、この副作用が参照透過性にどのように違反するのでしょうか。append の呼び出しを先ほどと同じように書き換えて、すべての y を y が参照している式と置き換えてみましょう。

```
scala> val x = new StringBuilder("Hello")
x: StringBuilder = Hello

scala> val r1 = x.append(", World").toString
r1: String = Hello, World

scala> val r2 = x.append(", World").toString
r2: String = Hello, World, World
```

この変換では、r1 と r2 が同じではなくなります。したがって、StringBuilder.append は純粋関数ではないという結論になります。どうなっているのでしょうか。r1 と r2 は同じ式のように見えますが、実際には、同じ StringBuilder の 2 つの異なる値を参照しています。r2 が x.append を呼び出すまでに、x が参照しているオブジェクトを r1 がすでに変更しています。これが難しく思えるとしたら、実際にそうだからです。副作用はプログラムの振る舞いについての論証を難しくするのです。

逆に、置換モデルの論証は単純です。なぜなら、評価の効果は完全に局所的で、その影響がおよぶのは評価の対象となる式に限られており、コードブロックを理解するために一連の状態の更新を頭の中でシミュレートする必要がないからです。理解するために必要なのは、**局所推論**（local reasoning）だけです。関数の動作を理解するために、関数の実行の前後に発生するかもしれない状態の変化をすべて頭の中で追跡する必要はありません。関数の定義を見て、引数をその本体に代入するだけです。「置換モデル」と呼んでいたかどうかはさておき、コードについて考えるときには、このように推論してきたはずです[※6]。

純粋性の概念をこのように形式化すると、関数型プログラムのモジュール性が高い傾向にある理由が見えてきます。モジュール型のプログラムは、プログラム全体から切り離して理解したり再利用したりできるコンポーネントで構成されています。「プログラム全体」が何を意味するかは、コンポーネントの意味とそれらの合成を決定するルールの 2 つによって決まります。つまり、それらは**合成可能**（composable）です。純粋関数がモジュラーで合成可能であるのは、純粋関数により、計算そのもののロジックが「結果を処理する方法」や「入力を取得する方法」から切り離され、それらの部分がブラックボックスになるからです。入力は、「関数への引数」というただ 1 つの方法でのみ取得されます。また、出力は計算され、返されるだけです。これらの関心を切り離しておくことで、計算ロジックの再利用性が高まります。結果による副作用と入力を要求する副作用のどちらかが適切であるかをコンテキストごとに判断しなくても、必要な場所でロジックを利用できるようになります。これについては、buyCoffee の例で確認したとおりです —— 請求処理という副作用を出力から取り除くことで、（buyCoffees と coalesce を記述したときのように）テストとさらなる合成という 2 つの目的に合わせて、関数のロジックをより簡単に再利用できるようになりました。

1.4　まとめ

本章では、関数型プログラミングがどのようなものであるかを示し、関数型プログラミングを使用する理由について詳しく説明しました。本書では関数スタイルのすべての利点を明らかにしていきますが、ここでは簡単な例を使って関数型プログラミングの利点をいくつか示しました。また、参照透過性と置換モデルを取り上げ、関数型プログラミングによってプログラムの推論が容易にな

※6　現実には、コードが純粋であるかどうかを判断するために、プログラマが置換を機械的に適用するようなことはない。たいてい見ればわかるからだ。

ることと、モジュール性が向上することについても説明しました。

本書では、あらゆるレベルのプログラミングに適用される関数型プログラミングの概念と原理を取り上げ、最も単純なタスクから始めて、それを土台に話を進めていきます。この後の章では、関数型プログラミングでループを記述する方法、データ構造を実装する方法、エラーと例外を処理する方法など、基礎的な部分を取り上げます。これらの手法をマスターし、関数型プログラミングの基本的なイディオムに慣れる必要があります。Part II、III、IV では、この知識をもとに、関数型プログラミングの設計手法を探っていきます。

2

Scala関数型プログラミングの準備

純粋関数だけを使用することにした場合、当然ながら、ごく単純なプログラムでさえどのように記述すればよいのか途方に暮れてしまいます。プログラムは順番に実行される命令の集まりであり、命令はそれぞれ何らかの効果をもたらすものである —— ほとんどのプログラマはこのように考えることに慣れています。本章からは、純粋関数を組み合わせるという方法で、Scala 言語でプログラムを記述する方法を学んでいきます。

本章の内容は、Scala、関数型プログラミング、またはどちらも初めてという読者を主な対象としています。新しい言語を学ぶにあたってイマージョン[※1]は効果的な手法なので、いきなり飛び込むことにします。Scala コードに慣れるには、とにかく Scala コードを見ることに尽きます。最初の章でコードを少し見てきましたが、本章では、小さいながら完全なプログラムを見ることから始めます。続いて、このプログラムのコードを分解し、それが実際に何をするのかを詳しく調べることで、Scala 言語の基礎とその構文を理解します。本書の目標は関数型プログラミングを理解することですが、その手段として Scala を使用するため、Scala 言語とその構文を十分に理解しておく必要があります。

Scala 言語の基本的な要素を説明した後は、関数型のプログラムを記述するための基本的な方法をいくつか紹介します。ここでは、**末尾再帰関数**を使ってループを記述する方法と、**高階関数**について説明します。高階関数は、他の関数を引数として受け取り、出力として関数を返すことができる関数です。さらに、型に基づいて実装を導く**多相高階関数**の例も紹介します。

本章には新しい内容が多数含まれています。高階関数に関連する内容によっては、関数をそのように受け渡すことができない言語でのプログラミングの経験が、かえって妨げになるかもしれませ

※1　［訳注］言語を身につけるための学習法の１つで、没入法とも呼ばれる。

ん。覚えておいてほしいのは、本章の概念を 1 つ残らず習得することや練習問題をすべて解くことが重要なのではない、ということです。本書では、こうした概念に異なる角度から改めてアプローチします。まず関数型プログラミングを体験してみることが、ここでの目標となります。

2.1　速習：Scala 言語

リスト 2-1 に示すのは、これから説明していく Scala プログラムの完全なコードです。ここでの目的は、関数型プログラミングの新しい概念を紹介することではなく、Scala 言語とその構文を知ってもらうことにあります。

リスト 2-1　gettingstarted/GettingStarted.scala

```
// コメント！
/* 別のコメント */
/** 説明コメント */
object MyModule { ❶
  def abs(n: Int): Int = ❷
    if (n < 0) -n ❸
    else n

  private def formatAbs(x: Int) = { ❹
    val msg = "The absolute value of %d is %d" ❺
    msg.format(x, abs(x)) ❻
  }

  def main(args: Array[String]): Unit = ❼
    println(formatAbs(-42))
}
```

❶ シングルトンオブジェクトを宣言することで、クラスとその唯一のインスタンスを同時に宣言。

❷ `abs` は、整数を受け取り、整数を返す。

❸ `n` が 0 よりも小さい場合に、その負の値を返す。

❹ `private` メソッドを呼び出せるのは、`MyModule` の他のメンバーのみ。

❺ `%d` で表される数値のプレースホルダを 2 つ含んだ文字列。

❻ 文字列内の 2 つの `%d` プレースホルダをそれぞれ `x` と `abs(x)` に置き換える。

❼ `Unit` は Java や C といった言語の `void` に相当する。

リスト 2-1 では、`MyModule` というオブジェクトを宣言しています。このようなオブジェクトをモジュールとも呼びます。ここでは、コードに場所と名前を与えて、あとから参照できるようにしているだけです。Scala コードは、`object` または `class` に含まれていなければならず、中括弧（`{}`）

で囲まれていなければなりません。リスト 2-1 では、より単純な object のほうを使用しています。オブジェクトとクラスについては、後ほど詳しく説明します。ひとまず、このオブジェクトを調べてみましょう。

object キーワード

object キーワードは新しいシングルトン型を作成します。シングルトン型は、名前付きのインスタンスが 1 つしか生成されない class のようなものです。Scala での object の宣言については、Java で無名クラスの新しいインスタンスを作成するのと同じように考えればよいでしょう。

Scala には、Java の static キーワードに相当するものはありません。Scala の object は、Java で static メンバーを持つクラスを使用するような場所でよく使用されます。

MyModule オブジェクトでは、def キーワードを使って abs、formatAbs、main の 3 つのメソッドが定義されています。本書では、def キーワードを使ってオブジェクトまたはクラス内で定義された関数またはフィールドをメソッドと呼びます。では、MyModule を少しずつ見ていきましょう。

abs メソッドは、整数を受け取ってその絶対値を返す純粋関数です。

```
def abs(n: Int): Int =
  if (n < 0) -n
  else n
```

def キーワードに続いてメソッドの名前を指定し、さらに括弧で囲んだパラメータリストを指定します。この場合、abs の引数は Int 型の n だけです。引数リストの閉じ括弧に続いて、オブジェクトの型アノテーション（型注釈）: Int が指定されており、結果の型が Int であることが示されています。コロン（:）は「型を持つ」という意味になります。

単一の等号（=）に続いて、メソッドの本体が始まります。等号の前にある宣言部分を**左辺**または**シグネチャ**と呼び、等号の後ろにあるコードを**右辺**または**定義**と呼ぶこともあります。return キーワードが明示的に指定されていないことに注意してください。右辺を評価した結果の値がそのままメソッドから返される値になります。if 式を含め、すべての式が結果を生成します。この場合、右辺は 1 つの式であり、式の値は n < 0 かどうかに応じて -n または n になります。

formatAbs メソッドも純粋関数です。

```
private def formatAbs(x: Int) = {
  val msg = "The absolute value of %d is %d"
  msg.format(x, abs(x))  ❶
}
```

❶ format は String で定義されている標準ライブラリのメソッド。

このメソッドは、msg オブジェクトで format メソッドを呼び出し、引数として x の値と、x に abs を適用した値を渡しています。x と abs(x) を評価した結果によって msg 内の %d がそれぞれ置き換えられ、新しい文字列が生成されます。

このメソッドは private として宣言されているため、MyModule オブジェクトの外側のコードからは呼び出せません。このメソッドは Int 型の引数を受け取って String 型の値を返しますが、戻り値の型が宣言されていないことに注意してください。Scala では、通常はメソッドの戻り値の型を推定できるため、戻り値の型が省略されても問題はありません。とはいえ一般的には、他の誰かが使用することが見込まれるメソッドについては、戻り値の型を明示的に宣言するのがよいスタイルであると考えられています。このメソッドはモジュールの private メソッドなので、型アノテーションは省略してかまいません。

このメソッドの本体には複数のステートメントが含まれているため、本体を中括弧で囲んであります。対の括弧によって囲まれているステートメントを**ブロック**と呼びます。ステートメントは改行またはセミコロン（;）で区切られます。この場合は改行を使ってステートメントを区切っているため、セミコロンは必要ありません。

ブロック内の最初のステートメントは、val キーワードを使って msg という名前の String 型の値を宣言しています。名前を付けているのは、この値をあとから参照できるようにするためです。val はイミュータブルな（不変の）変数であるため、formatAbs メソッドの本体では、msg という名前は常に同じ String 値を指しています。同じブロック内で msg に別の値を代入すると、コンパイルエラーになります。

メソッドはその右辺の値を返すだけなので、return は不要であることを思い出してください。この場合、右辺はブロックです。Scala では、中括弧で囲まれた複数のステートメントからなるブロックの値は、ブロック内の最後の式が返す値と同じになります。したがって、msg.format(x, abs(x)) の呼び出しによって返される値が formatAbs メソッドの結果になります。

最後の main メソッドは外殻であり、純粋関数型のコアを呼び出し、答えをコンソールに出力します。このようなメソッドは、副作用があることを強調する意味で、関数ではなく**プロシージャ**（procedure）または**非純粋関数**（impure function）と呼ばれることがあります。

```
def main(args: Array[String]): Unit =
  println(formatAbs(-42))
```


プログラムを実行すると、Scala 実行環境が特定のシグネチャを持つ main というメソッドを探します。つまり、main という名前には特別な意味があります。main メソッドは、引数として String 型の配列（Array）を受け取らなければならず、戻り値は Unit 型でなければなりません。

なお、ここでは使用しませんが、args 配列には、プログラムの実行時にコマンドラインで指定された引数が含まれています。

Unit は、C や Java といったプログラミング言語の void と同じような働きをします。Scala のメソッドは、クラッシュまたはハングしない限り、必ず何らかの値を返さなければなりません。そこで、意味のある値を返さない main のようなメソッドの戻り値の型として、Unit という特別な型が定義されています ※2。この型の値は 1 つしか存在せず、そのリテラル構文は空の括弧（()）です。通常、戻り値の型が Unit であることは、そのメソッドに副作用があることのヒントになります。

リスト 2-1 の main メソッドの本体は、formatAbs の呼び出しによって返された文字列（String）をコンソールに書き出します。println の戻り値の型が Unit で、main の戻り値の型と一致することに注意してください。

2.2　プログラムの実行

ここでは、Scala プログラムを実行する最も簡単な方法を紹介します。この方法は短い例を試してみるのにうってつけですが、Scala のビルドツールである sbt や、IntelliJ や Eclipse などの IDE を使って Scala コードをビルドして実行するほうが一般的です。sbt については、本書の GitHub ※3 を参照してください。

MyModule プログラムを実行するとしたら、コマンドラインから Scala コンパイラを直接起動するのが最も簡単です。まず、このプログラムのコードを MyModule.scala などの名前が付いたファイルに保存します。次に、scalac コンパイラを使って、Java バイトコードにコンパイルします。

```
> scalac MyModule.scala
```

そうすると、.class という拡張子が付いたファイルが生成されます。これらのファイルには、Java 仮想マシン（JVM）で実行可能なコンパイル済みのコードが含まれています。このコードは

※2　型と同じように「ユニット」と読む。

※3　https://github.com/fpinscala/fpinscala