本章では、Haskellのはじめの一歩として、言語の特徴と、ビルドツールStackを中心とした開発環境の構築を解説します。

1.1 Haskellの特徴

純粋関数型プログラミング言語であるHaskellの設計思想を紹介します。

純粋関数型プログラミング言語と書くと、難しそうに感じるかもしれません。しかし、身構える必要はありません。Haskellは普通の、実用的なプログラミング言語です。

パラダイムに若干の違いはありますが、皆さんが今まで触れてきたであろうC言語やJava、JavaScript、Python…となんら変わることはありません。

実用面で考えれば他のプログラミング言語で開発できるアプリケーションは、当然Haskellでも開発できます。Haskellが推進する関数型プログラミング*1は、上で挙げたような言語が推進するパラダイムとは少し趣が異なっていますが、パラダイムが違うからといってアプリケーション開発ができなくなるわけではありません。

例えば、JavaScriptではプロトタイプベースという他の一般的な言語が提供するオブジェクト指向とは異なるパラダイムが用いられます。現在のJavaScript（Node.js）の成功を見れば、パラダイムが違っても作れるアプリケーションに違いがでるわけではないことはわかるはずです。

Haskellは関数型、静的型付け、純粋性、型推論、遅延評価などの特徴を持ちます。それぞれの特徴について見ていきます。

Column

Haskellの歴史

Haskellは1990年、遅延評価を持つ関数型プログラミング言語のオープンな仕様（Haskell 1.0）として誕生しました。

その後もHaskell 98、Haskell 2010と仕様が公開され、広く参照されています。

Haskellは委員会で仕様を策定されました。そのためRubyのまつもとゆきひろ、PythonのGuido van Rossumに相当するような生みの親と呼ぶべき開発者は存在しません*2。

*1 Functional Programmingは関数型プログラミングの他、関数プログラミングなどと訳されることもあります。本書では関数型プログラミングをはじめ、関数型の表記を用います。

*2 GHCの開発者であるSimon Peyton JonesがHaskellの仕様、実装面での貢献からHaskellの開発者の一人として広く知られています。

1.1.1 関数型プログラミング言語

関数型プログラミング言語*3*4という言葉を字面から判断すると、関数がプログラミングの主役であることは想像がつきます。

しかし、C言語に代表される手続き型のプログラミングパラダイムでも、関数定義は重要な言語機能です。Javaのようなオブジェクト指向言語におけるメソッドも関数の一種です。これらの言語も関数型プログラミング言語と呼んでもいいのでしょうか。

関数型プログラミングの定義を知るとそうではないことがわかります。

関数型プログラミングの定義

関数型プログラミングとは、引数に対して値が決まる、数学的な関数を中心に計算を表現するプログラミングスタイルを指します*5。

数学的な関数とはなんでしょうか。次のPythonで書かれたsquare関数を見てください。この関数は2を渡すと、必ず4が返ってくる関数です。引数だけで値が決まります。

```
def square(n):
    return n ** 2
```

このように、関数を同じ値となる式に対して呼んだときに必ず同じ結果となる性質を、**参照透過性**と言います。

参照透過性を持ち合わせた関数が数学的な関数です。関数型プログラミングの主役は、このような数学的な関数を組み合わせて作った**式**です。

次のcountup関数は、同じ1を渡しても呼び出しを重ねるごとに返り値は1、2、3・・・と増えていきます。引数の値だけでは挙動が決まらない、参照透過性のない関数です。このような関数は数学的な関数とは言えません。

```
counter = 0
def countup(n):
    global counter
    counter += n
    return counter
```

一般に関数型プログラミング言語と呼ぶときは、数学的な関数の利用を推奨しているプログラミング言語を指します。

*3 略して関数型言語とも。

*4 関数型プログラミング言語としてはHaskellの他にOCamlなども広く知られています。

*5 関数型プログラミング言語については第6章も参照。

この章では変数や関数、I/Oアクション、よく使われるデータ型といったHaskellの基本的な文法事項について説明していきます。実際に手を動かしながら理解していきましょう。

2.1 文法の特色

Haskellの文法上の特徴は**式を重視し、可能な限り式だけでプログラムを構成する**ことです。この点について理解するために、まずは読者の多くが使ったことのあるだろう手続き型言語との比較から、Haskellの特徴をとらえてみましょう。

2.1.1 手続き型言語と関数型言語

C言語やJava、JavaScript、Python、Rubyなどの現在使われている言語の多くは、手続き型の文法を持っています。

これらの言語には式（expression）と文（statement）が存在します。式とは、計算を実行して結果を得るための文法要素であり、加減乗除や関数呼び出しからなります。

文とは何らかの動作を行うようにコンピュータに指示するための文法要素です。条件分岐の`if`文、ループの`for`文や`while`文などが言語仕様として含まれていることがほとんどです。言語によっては出力の`print`文などもあります。

手続き型の文法では、式で必要な計算を進めつつ、文でコンピュータにやってもらうことを指示してプログラムを記述します。これらの手続き型言語で重要なのは文です。

それに対して、Haskellはじめ関数型言語の文法の主役は式です。関数型言語のプログラムは数多くの式で構成され、プログラムそれ自体も1つの式です。Haskellでもプログラムを書くのに文は原則使いません*1*2。

関数型言語のプログラムの実行とは、この式の計算を進め、その結果として値（value）を得ることだと考えてください。式の計算のことを、**評価する**（evaluate）といいます。

手続き型言語ではコンピュータへの指示を文として上から順に並べて書くのに対し、関数型言語では細かな式をたくさん定義してそれらを組み合わせてプログラムを構築します。*3*4。

*1　多くの関数型言語では文はまったく、あるいはほとんど文法要素として存在しません。

*2　Haskellでは条件分岐の`if`は`if`式で、文ではありません（2.6.1参照）。外部モジュールを読み込む`import`宣言は、仕様で文とも表現されていますが、プログラムそのものには影響を与えられない部分でしか宣言できません。

*3　Haskellでもわかりやすさのためにソースコード上でそれぞれの式を手続き毎に改行で区切った表記はよくあります。そのため、手続き型言語と全く違う書き方を強要されることはありません。実際にどのように書かれるかは2.5のdo式を参照してください。

*4　関数型言語が式を組み合わせてプログラムを作成するといっても現段階では想像がつきづらいでしょう。本書の第9章、第10章、第11章では実際にHaskellでプログラムを作成しています。これらを参考にするとHaskellのような関数型言語におけるプログラムの構築が理解できるようになるはずです。

式と文の違いとして重要な点は他に、**型**があります。

式は`Bool`や`String -> Int`などといった型を持ちます。対して、文は型を持ちません。Haskellではプログラムがすべて式で構成されていることから、プログラムの全体が細部まで型付けされています。型付けが可能になることでプログラムがより堅固なものになります。

2.1.2 式だけで意味のあるプログラムをつくる

関数型言語では手続き型の言語のように文を使ってコンピュータに指示するという考え方はしません。式を評価するだけです。

コンピュータに指示を出さずに、コンピュータを動作させる実用的なプログラムが作成できるのかという疑問が出てくるかもしれません。もちろん、実用的なプログラムを作成するための仕組みはHaskellにも用意されています。

関数型言語では文による指示の代わりに、評価をするとなんらかのI/Oが発生する特殊な式を用います*5。Haskellにおいては、I/Oを表現する式を**I/Oアクション**と呼びます。I/Oアクションの例を見てみましょう。

- ファイルの読み書き
- コンソール上への文字出力や入力行の読み込み
- HTTPリクエストの送信とHTTPレスポンスの受信
- 日付の取得

これらの例からわかるようにI/OアクションはHaskellのプログラムの外に影響を与えたり、外から影響を受けたりするものです。I/Oアクションは頭に`IO`が付いた特別な型を持っており、I/Oを表さない型とは常に区別されます*6。

2.2 基本のデータ型

ここまでは関数型言語としてのHaskellの文法全体の特徴を紹介してきました。ここからはHaskellの個々の文法事項について解説していきます。

まずは標準的なデータ型を見ていきましょう。値を確認するためには式を評価します*7。第1章で紹介したREPLを使います。

*5　I/Oを伴う式は手続き型言語における文と、本質的には、そこまで大きく違うものではありません。しかし、言語の設計方針やプログラマーの思考に大きな影響を与えます。

*6　これらの型については2.2で紹介します。

*7　Haskellで値とは式の評価の結果得られるものを意味します。REPLで1と入力し、Enterキーを押すことも式の評価です。

本章ではHaskellの静的型付き言語としての側面、すなわち型について詳しく見ていきます。

Haskellの型は簡単に言ってしまえば、`1 :: Int`の`Int`部分に相当するものです。ここまでもコード中に書いてきましたが、あらためて学んでいきましょう。

型クラスについてもこの章で解説します。他の言語では聞かない概念ですが、Haskellの柔軟さのために欠かせない要素です。

3.1 型の記述

型とは、式をどのように組み合わせられるかを規定するものです。

例えば、`f :: Int -> Int`は引数に`Int`型を、返り値に`Int`型を必要とする関数です。

このように引数に使える式が返すべき値の型が規定され、式をどう組み合わせられるかが決まっていきます。パズルのピースをピッタリはめるようなものと考えてみてください。読者の多くが経験のあろう動的型付き言語では、このような式の組み合わせは規定できません。

型は**データの種類**を規定するものと表現すると、より直感的かもしれません。`1 :: Int`、`2.14 :: Float`などは型でデータを規定する典型的な例です。ただし、式をいくつも組み合わせていくHaskellにおいては、先述のように**型とは式をどのように組み合わせられるかを規定するもの**と定義しておくことが理解を助けます。よって本書ではこの定義を用います。

Haskellでは次のように型を記述します。式と型の間を`::`で区切り、引数間、引数と返り値の間を`->`でつなげます。型の対応関係を明示するために式と型をかっこで囲むこともあります。

```
式 :: 型
```

型を実際に書いてみましょう。

```
-- 引数を持たない式の場合
-- 式 :: 型
1 :: Int

-- 引数が1つの式の場合
-- 式 :: 引数の型 -> 返り値の型
putStr :: String -> IO ()

-- 引数が2つの式の場合
-- 式 :: 第一引数の型 -> 第二引数の型 -> 返り値の型
(||) :: Bool -> Bool -> Bool

-- 引数がn個ある式の場合
-- 式 :: 第一引数の型 -> ... 第n引数の型 -> 返り値の型
```


3.2 型システム

式に型を割り当てて、正しく組み立てられているか確かめる仕組みのことを、**型システム**（**type system**）と呼びます。Haskellの型システムには柔軟性を高める、いくつかの特徴があります。それらをコードを確認しながら見ていきましょう。

3.2.1 型チェック

JavaScriptやRubyなど、動的型付き言語では、実行時に型を確認します。そのため、型が合わずにエラーが発生する可能性があるプログラムも実行できてしまいます*1。

Haskellをはじめとした**静的型付け**を行う言語では、プログラムをコンパイルする時点ですべての式の型が決まります。型が合わなければ、コンパイルできません。そのため実行時には原則として型に起因するエラーは発生しません。

式を組み合わせるには、型に基づいた**型付け規則**（typing rules）にしたがう必要があります。型付け規則にしたがっているかを確認することを、**型チェック**（type check）と呼びます。

型付け規則とは`f :: Char -> String`という関数と`x :: Char`という式があるとき、関数適用した`f x`の型は`String`であるというようなルールです。型チェックでは式がそれら規則に則っているか否かを確認します。実際に型チェックがどのように機能するか、動的型付き言語のPythonとの比較で確認してみましょう。

```
# 指定した長さの配列を特定の数字で埋めて返す
def int_to_array(i, l):
    return [i] * l

int_to_array(3, 4) # [3, 3, 3, 3]

# 整数が入るべき引数lに小数が入ったため実行時にエラーが発生する
int_to_array(5000, 12.04)

# エラーは発生しないが想定している用途（数字の配列）に合致しない
# エラーを返すには型をチェックするコードを追記する必要がある
int_to_array('a', 2) # ['a', 'a']
```

```
-- Main.hsに書き、`stack runghc -- Main.hs`で実行
-- Int型の第一引数、Int型の第二引数、Int型のリストの返り値
intToArray :: Int -> Int -> [Int]
```

*1 動的型付き言語のこの性質は、「70%の完成度でリリースできる」と言い換えられます。また、「正しく実行できるが型付け規則に合わない式」も残念ながら存在しています。動的型付き言語では当然そのような式を実行できます。静的型付き言語では、型付け規則の表現力を日々高めることでそのような式を減らす努力をしています。

```
-- リストからl分取り出す
intToArray i l = take l (repeat i)

main :: IO ()
main = do
  -- 型に合致するため実行できる
  print (intToArray 3 4) -- [3, 3, 3, 3]
  -- 第二引数の型が一致しないためコンパイルエラーが発生する
  print (intToArray 5000 12.04)
  -- 第一引数(i :: Int)の型が一致しないためコンパイルエラーが発生する。repeat自体は引数にChar型も許容するがintToArrayの型を明示しているためここでは利用できない
  print (intToArray 'a' 10)
```

型チェックのおかげで、静的型付き言語では型に関して不備のあるプログラムが実行されることはありません。この点において、静的型付き言語は動的型付き言語より安全と言えます*2。

3.2.2 多相性

型チェックは正しいプログラムを書くために役立つ機能ですが、作った式が`Integer`や`String`などある特定の型にしか使えないとなると、コードの再利用性を著しく損ねてしまいます。

Haskellの型システムは多相性を提供してコードの再利用性を高めます。また型推論も多相性をサポートしています。

パラメータ多相

例えば、`sample x = x`と定義された`sample`関数は、任意の型に対して適用できます。

```
Prelude> sample x = x
Prelude> :t sample
sample :: t -> t
Prelude>  -- 2引数の場合はt、t1
Prelude> sample2Params f x = f x
Prelude> :t sample2Params
sample2Params :: (t1 -> t) -> t1 -> t
```

ここでの`t`型、`t1`型はその関数を扱う際に任意の具体的な型に置き換えられることを示しています。

*2 ここでは比較のためにPython側を最小の処理で書いていますが、Pythonでも引数のチェックや外部ツールの導入でHaskellの`intToArray`と同等の機能は実現できます。

このように一つの式に汎用の型を表す型変数`t`を割り当て、複数の具体的な型で利用できるようにした型システムは**パラメータ多相**(**parametric polymorphism**)を持つといいます*3*4。

アドホック多相

Haskellでは**型クラス**(type class)という仕組みを利用して、型によって全く異なる実装の式を使えます。記述だけ見ると同一の式に見えても、型によって柔軟に処理を変えられます*5。

例えば`1 == 1`と`"OCaml" == "OCaml"`の式のそれぞれの`(==)`関数では、関数名は同じですが、まったく別の実装が使われます。型クラスによる多相は、必要に応じて型ごとに場当たり的に違う実装を追加できるため、**アドホック多相**(ad-hoc polymorphism)と呼ばれます*6。アドホック多相は、オブジェクト指向言語におけるオーバーロードに相当します。

3.2.3 型推論

Haskellは強力な**型推論**(**type inference**)を備えた言語です。

型推論は型を書かなくても処理系が自動的に型を決定してくれることを指します(1.1.4参照)。これによってHaskellは静的型付き言語でありながら、ほとんど型を書かずにプログラミングできます。

静的型付き言語は実行前に型に起因するエラーを避けられるなど優れた特徴を有します。反面、動的型付き言語では記述することのない型を書くため、それが煩雑に感じられてしまうことがあります*7。Haskellでは型推論によって静的型付き言語としての特徴を生かしつつ、型を書く煩雑さを回避しています。

次のような`hello`という関数があったとします。型はまったく書かれていませんが、`"Hello, "`と`name`が結合できることから、`name`は文字列(`String`)で`hello`の返り値も文字列でなければなりません。そのため、`hello`は次のような型の関数となります*8。

```
Prelude> hello name = "Hello, " ++ name
```

```
Prelude> :t hello
hello :: [Char] -> [Char]
```

*3 言語によってはジェネリクスとも呼ばれます。

*4 多相、多相性はいずれもpolymorphismの訳語です。polymorphismはそのままにポリモーフィズムと訳されることもありますが、Haskellの文化圏では多相あるいは多相性と訳されることが多いです。多相は、1つの式が複数の型として利用されることを指します。

*5 型クラスについては3.8で解説します。

*6 オブジェクト指向の言語では、サブクラスでスーパークラスのメソッドが呼べたり、返り値を実際の返り値のクラスのスーパークラスのインスタンスとして受られたりします。これを部分型多相と呼びますが、Haskellには部分型多相の機能はありません。

*7 C言語やJavaで型を書くことになれていると、JavaScriptやRubyを書くとき型を書かずに済む軽快さを感じるのではないでしょうか。

*8 関数の型は`->`を用いて、引数 `->` 返り値という形式で表したことを思い出してください。