Haskell 実習マニュアル

第零版

大黒学

Haskell 実習マニュアル・第零版
著者——大黒学

2015 年　1 月 28 日（水）　第零版発行

目次

第 1 章　Haskell の基礎

1.1　プログラム

1.1.1　文書と言語

文字を並べることによって何かを記述したものは、「文書」(document) と呼ばれます。

文書を作るためには、記述したいことを意味として持つように、文字を並べていく必要があります。そして、そのためには、文字をどのように並べればどのような意味になるかということを定めた規則が必要になります。そのような規則は、「言語」(language) と呼ばれます。

人間に読んでもらうことを第一の目的とする文書を書く場合は、日本語や中国語やアラビア語のような、「自然言語」(natural language) と呼ばれる言語が使われます。自然言語というのは、人間の社会の中で自然発生的に形成された言語のことです。

言語には、自然言語のほかに、人間が意図的に設計することによって作られた、「人工言語」(artificial language) と呼ばれるものもあります。人間ではなくてコンピュータに読んでもらうことを第一の目的とする文書を書く場合は、通常、自然言語ではなく人工言語が使われます。

1.1.2　プログラムとプログラミング

コンピュータに何らかの動作を実行させるためには、実行してほしいことがどんな動作なのかということを記述した文書をコンピュータに与える必要があります。そのような文書は、「プログラム」(program) と呼ばれます。

プログラムを作成するためには、プログラムを書くという作業だけではなくて、プログラムの構造を設計したり、プログラムの動作をテストしたり、その不具合を修正したりするというような、さまざまな作業が必要になります。そのような、プログラムを作成するために必要となるさまざまな作業の全体は、「プログラミング」(programming) と呼ばれます。

1.1.3　プログラミング言語

プログラムというのも文書の一種ですから、それを書くためには何らかの言語が必要になります。プログラムを書く場合には、プログラムを書くことだけを目的として作られた人工言語を使うのが普通です。そのような、プログラムを書くための専用の言語は、「プログラミング言語」(programming language) と呼ばれます。

プログラミング言語には、たくさんのものがあります。例を挙げると、Fortran、COBOL、Lisp、Pascal、Basic、C、AWK、Smalltalk、ML、Prolog、Perl、PostScript、Tcl、Java、Ruby、…… というように、枚挙にいとまがないほどです。

1.1.4　この文章について

この文章（「Haskell 実習マニュアル」）は、Haskell というプログラミング言語を使って、プログラムというものの書き方について説明する、ということを目的とするチュートリアルです。

1.2　言語処理系

1.2.1　言語処理系の基礎

コンピュータというものは異質な二つの要素から構成されていて、それぞれの要素は、「ハードウェア」(hardware) と「ソフトウェア」(software) と呼ばれます。ハードウェアというのは物理的な装置のことで、ソフトウェアというのはプログラムなどのデータのことです。

コンピュータは、さまざまなプログラミング言語で書かれたプログラムを理解して実行することができます。しかし、コンピュータのハードウェアが、ソフトウェアの助力を得ないで単独で理解することのできるプログラミング言語は、ハードウェアの種類によって決まっているひとつの言語だけです。

ハードウェアが理解することのできるプログラミング言語は、そのハードウェアの「機械語」(machine language) と呼ばれます。機械語というのは、人間にとっては書くことも読むことも困難な言語ですので、人間が機械語でプログラムを書くということはめったにありません。

人間にとって書いたり読んだりすることが容易なプログラミング言語で書かれたプログラムをコンピュータに理解させるためには、そのためのプログラムが必要になります。そのような、人

間が書いたプログラムをコンピュータに理解させるためのプログラムのことを、「言語処理系」(language processor) と呼びます（「言語」を省略して、単に「処理系」と呼ぶこともあります)。

言語処理系には、「コンパイラ」(compiler) と「インタプリタ」(interpreter) と呼ばれる二つの種類があります。コンパイラというのは、人間が書いたプログラムを機械語に翻訳するプログラムのことで、インタプリタというのは、人間が書いたプログラムがあらわしている動作をコンピュータに実行させるプログラムのことです。

1.2.2 Haskell の言語処理系

Haskell の言語処理系のうちで、もっとも広く使われているのは、Glasgow Haskell Compiler (GHC) と呼ばれるコンパイラです。

GHC をインストールしたいときは、次のサイトから、Haskell Pratform と呼ばれるものをダウンロードします。

```
http://www.haskell.org/platform/
```

1.2.3 REPL

言語処理系は、「REPL」と呼ばれるものと、そうでないものとに分類することができます。REPL というのは、read-eval-print loop の略称で、次の三つの動作を延々と繰り返す、という動作をする処理系のことです。

(1) プログラムまたはその断片を読み込む (read)。
(2) 読み込んだプログラムまたはその断片を実行する (eval)。
(3) 結果を出力する (print)。

ですから、REPL を使うことによって、プログラムまたはその断片をキーボードから直接入力して、それがどのように動作するかということを即座に確かめる、ということができます。

1.2.4 GHCi

GHC は、それ自体はコンパイラですが、Haskell Pratform をコンピュータにインストールすると、GHC だけではなくて、GHCi という名前のインタプリタもインストールされます（GHCi の i は、「対話的な」(interactive) という意味です)。GHC は REPL ではありませんが、GHCi は REPL です。

GHCi は、`ghci` というコマンドを入力することによって、起動することができます。

それでは、実際に GHCi を起動してみましょう。コマンドを入力するためのアプリ（Linux や Mac OS ならばターミナル、Windows ならばコマンドプロンプト）を起動して、`ghci` というコマンドを入力してみてください。そうすると、

```
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

というように表示されるはずです（バージョンが違う GHCi は、これとは違うものを表示するかもしれません)。この最後に表示されている、

```
Prelude>
```

といのが、GHCi が出力するプロンプトです。

GHCi は、コマンドを入力することによって操作することができます。GHCi に対するコマンドは、コロン (:) で始まります。たとえば、`:help` または `:?` というコマンドを入力すると、コマンドの使い方についてのヘルプが出力されます。

それでは、ヘルプを出力させてみましょう。`:?` というコマンドを入力して、そののちエンターキーを押してみてください。

```
Prelude> :?
 Commands available from the prompt:
（以下略）
```

1.2.5 GHCi の終了

GHCi は、`:quit` または `:q` というコマンドを入力することによって終了させることができます。

それでは、GHCi を終了させてみましょう。`:q` というコマンドを入力して、そののちエンターキーを押してみてください。そうすると、GHCi が終了して、Linux や Mac OS の場合はターミナルのプロンプトが、Windows の場合はコマンドプロンプトのプロンプトが表示されるはずです。

1.2.6 式の入力

Haskell のプログラムの中には、「式」(expression) と呼ばれるものを書くことができます。式については、第 2 章で詳しく説明することになりますが、とりあえずここでは、数学で使われる式のように、何らかの計算を書きあらわしたものだと考えてください。

GHCi には、コマンドだけではなくて、式を入力することもできます。GHCi に式を入力すると、GHCi は、それがあらわしている計算を実行して、その結果を出力します。

それでは、式を GHCi に入力してみましょう。たとえば、`5+3` という式を入力して、そののちエンターキーを押してみてください。そうすると、次のように、入力した式があらわしている計算が実行されて、`8` という結果が出力されます。

```
Prelude> 5+3
8
Prelude>
```

1.2.7 エラー

プログラムの中には、何らかの間違いが含まれていることがあります。そのような、プログラムの中に含まれている間違いは、「エラー」(error) と呼ばれます。

言語処理系は、入力されたプログラムがエラーを含んでいる場合、そのエラーについてのメッセージを出力します。そのような、エラーについてのメッセージは、「エラーメッセージ」(error message) と呼ばれます。

それでは、GHCi に対して、`hoge` と入力してみてください。そうすると、おそらく、

```
Not in scope: ‘hoge’
```

というメッセージが表示されるはずです。これは、`hoge` というのが何の名前なのか分からない、ということを意味しているエラーメッセージです。

1.3 型の基礎

1.3.1 型とは何か

同じ性質を持つデータの集合は、「型」(type) と呼ばれます。

Haskell のプログラムにおいては、データはかならず、何らかの型に所属しているものとして扱われます。

データ D が型 T に所属しているとき、「D は T を持つ」という言い方をすることもあります。

1.3.2 基本型

Haskell のプログラムが扱うデータの型には、さまざまなものがあります。それらのうちでもっとも基本的な一群の型は、「基本型」(base type) と呼ばれます。すべての基本型は、「型名」(type name) と呼ばれる、英字の大文字で始まる名前が与えられています。

基本型には、次のようなものがあります。

`Bool`	真偽値（第 1.3.3 項参照）の型。
`Int`	コンピュータのハードウェアによって上限と下限が定められている整数の型。
`Integer`	上限も下限も定められていない整数の型。
`Float`	浮動小数点数（第 1.3.4 項参照）の型。
`Double`	`Float` よりも精度の高い浮動小数点数の型。
`Char`	文字の型。

1.3.3 真偽値

成り立っているか、それとも成り立っていないか、という判断の対象は、「条件」(condition) と呼ばれます。

条件が成り立っていると判断されるとき、その条件は「真」(true) であると言われます。逆に、条件が成り立っていないと判断されるとき、その条件は「偽」(false) であると言われます。

真を意味するデータと、偽を意味するデータは、総称して「真偽値」(Boolean) と呼ばれます。

1.3.4　浮動小数点数

通常、コンピュータの内部では、小数点以下の桁を持つ数値は、浮動小数点数によって表現されます。「浮動小数点数」(floating point number) というのは、小数点が仮に置かれた数字列と、その数字列の小数点を移動させる整数、という二つのものの組によってあらわされる数値のことです。

小数点が仮に置かれた数字列は「仮数部」(mantissa part) と呼ばれ、仮数部の小数点を移動させる整数は「指数部」(exponent part) と呼ばれます。プラスの指数部は小数点を左へ移動させ、マイナスの指数部は小数点を右へ移動させます。

仮数部として使うことのできる数字列の最大の桁数は、「精度」(precision) と呼ばれます。

1.3.5　複合的な型

型としては、基本型だけではなくて、基本型を組み合わせることによってできる複合的なものもあります。

複合的な型を持つデータとしては、次のようなものがあります。

- タプル (tuple)
- リスト (list)
- 関数 (function)

タプルについては第 2.5 節で、リストについては第 4 章で、関数については第 2.3 節と第 3 章で説明することにしたいと思います。

1.3.6　型式

任意の型は、「型式」(type expression) と呼ばれるものを書くことによって記述することができます。

基本型は、それの型名が、それを記述する型式になります。複合的な型を記述する型式は、その型を構成する基本型の型名を組み合わせることによって記述されます。

1.3.7　文字列の型

文字列は、基本型を持つデータではなくて、複合的な型を持つデータです。

文字列の型は、`[Char]` という型式によって記述されます。これは、「文字のリストの型」という意味です。

文字列の型を記述する `[Char]` という型式には、`String` という同義語が定義されています。ですから、文字列の型は、`[Char]` と書いてもかまいませんし、`String` と書いてもかまいません。

第 2 章　式

2.1　式の基礎

2.1.1　式と評価と値

Haskell のプログラムの中には、「式」(expression) と呼ばれるものを書くことができます。

式というのは、コンピュータによる何らかの動作をあらわしています。ですから、コンピュータは、式があらわしている動作を実行することができます。ただし、式の場合は、「実行する」(execute) とは言わずに、「評価する」(evaluate) と言うのが普通です。

式を評価すると、その結果として一つのデータが得られます。式を評価することによって得られるデータは、その式の「値」(value) と呼ばれます。

「評価する」という言葉は、「値を求める」というニュアンスを含んでいます。式を実行することを、「実行する」と言わずに「評価する」と言うのは、「式の実行というのは、単なる動作の実行ではなくて、値を求めるという志向性を持った動作の実行である」という意識が強く働いているからです。

2.1.2 式の構造

式というのは、プログラムというものを組み立てるための部品のようなものだと考えることができます。

多くの場合、式という部品は、より単純な式を組み合わせることによって作られています。たとえば、5+3 というのはひとつの式ですが、この式は、より単純な式を組み合わせることによって作られています。

5+3 という式の中にある 5 という部分は、5 という整数を求めるという動作をあらわしている式です。したがって、GHCi に対して 5 と入力すると、その式が評価されて、その値が出力されます。

```
Prelude> 5
5
```

同じように、3 という部分も、3 という整数を求めるという動作をあらわしている式です。つまり、5+3 という式は、5 という式と 3 という式を、+ というものをあいだにはさんで結びつけることによってできているわけです。

どんな式でも、それ自体を、さらに複雑な式の部品にすることができます。たとえば、5+3 という式を部品にして、5+3-7 という式を作ることができます。式というのは、組み合わせることによっていくらでも複雑なものを作ることができるという、そんな性質を持っている部品なのです。

2.2 リテラル

2.2.1 リテラルの基礎

特定のデータを生成するという動作を記述した式は、「リテラル」(literal) と呼ばれます。たとえば、481 のような、何個かの数字を並べることによってできる列は、リテラルの一種です。

リテラルを評価すると、それによって生成されたデータが、その値として得られます。たとえば、481 というリテラルを評価すると、それによって生成された 481 という整数のデータが、その値として得られます。

```
Prelude> 481
481
```

なお、この文章のこれから先の部分では、誤解のおそれがない場合、「○○のデータ」のことを単に「○○」と呼ぶことがあります。たとえば、整数そのものではなくて整数のデータのことを指しているということが文脈から明らかに分かる場合には、整数のデータのことを単に「整数」と呼ぶことがあります。

2.2.2 整数リテラル

整数のデータを生成するリテラルは、「整数リテラル」(integer literal) と呼ばれます。

整数リテラルは、次の 4 種類に分類することができます。

- 10 進数リテラル (decimal integer literal)
- 16 進数リテラル (hexadecimal integer literal)
- 8 進数リテラル (octal integer literal)

これらの整数リテラルの相違点は、その名前が示しているとおり、整数を表現するための基数です。つまり、それぞれの整数リテラルは、10、16、8、2 のそれぞれを基数として整数を表現します。

10 進数リテラルは、481 とか 3007 というような、数字だけから構成される列です。たとえば、481 という 10 進数リテラルを評価すると、481 というプラスの整数が値として得られます。

16 進数リテラルと 8 進数リテラルは、基数を示す接頭辞を先頭に書くことによって作られます。基数を示す接頭辞は、16 進数は 0x、8 進数は 0o です（x と o は大文字でもかまいません）。たとえば、0xff と 0o377 は、どちらも、255 という整数を生成します。

```
Prelude> 0xff
255
Prelude> 0o377
255
```

2.2.3 浮動小数点数リテラル

ひとつの数値を、数字の列と小数点の位置という二つの要素で表現しているデータは、「浮動小数点数」(floating point number) と呼ばれます。

浮動小数点数のデータを生成するリテラルは、「浮動小数点数リテラル」(floating point literal) と呼ばれます。

`0.003`、`41.56`、`723.0` というような、ドット (`.`) という文字の左右に数字の列を書いたものは、浮動小数点数のデータを生成するリテラルになります。この場合、ドットは小数点の位置を示します。

```
Prelude> 3.14
3.14
```

浮動小数点数を生成するリテラルとしては、

a `e` b

という形のものを書くことも可能です（ `e` は大文字でもかまいません)。a のところには、ドットを 1 個だけ含むかまたは含まない数字の列を書くことができて、b のところには、数字の列または左側にマイナスのある数字の列を書くことができます。この形のリテラルを評価すると、

$a \times 10^b$

という浮動小数点数が生成されます。

リテラル	意味
`3e8`	3×10^8
`6.022e23`	6.022×10^{23}
`6.626e-34`	6.626×10^{-34}

```
Prelude> 3e8
3.0e8
```

2.2.4 マイナスの数値を生成する式

マイナス (`-`) という文字を書いて、その右側に整数または浮動小数点数のリテラルを書くと、その全体は、マイナスの数値を生成する式になります。たとえば、`-56` という式はマイナスの 56 という整数を生成して、`-0xff` はマイナスの 255 という整数を生成して、`-8.317` はマイナスの 8.317 という浮動小数点数を生成します。

```
Prelude> -56
-56
```

マイナスの数値を生成するこのような式の先頭に書かれるマイナスという文字は、リテラルの一部分ではなくて、第 2.4 節で説明することになる「演算子」(operator) と呼ばれるものです。

2.2.5 文字リテラル

文字のデータを生成するリテラルは、「文字リテラル」(character literal) と呼ばれます。

文字リテラルは、一重引用符 (`'`) で 1 個の文字を囲むことによって作られます。たとえば、`'A'` は、文字リテラルです。

文字リテラルは、一重引用符で囲まれた中にある文字を生成します。たとえば、`'A'` という文字リテラルは、`A` という文字を生成します。

```
Prelude> 'A'
'A'
```

2.2.6 文字列リテラル

文字列のデータを生成するリテラルは、「文字列リテラル」(string literal) と呼ばれます。

文字列リテラルは、二重引用符 (`"`) で文字列を囲むことによって作られます。たとえば、`"namako"` は、文字列リテラルです。

文字列リテラルは、二重引用符で囲まれた中にある文字列を生成します。たとえば、`"namako"` という文字列リテラルは、`namako` という文字列を生成します。

```
Prelude> "namako"
```

```
"namako"
```

2.2.7 エスケープシーケンス

文字の中には、たとえばビープ音や改ページのように、そのままでは文字リテラルや文字列リテラルの中に書くことができない特殊なものがあります。

そのような特殊な文字を生成する文字リテラルや、それを含んだ文字列を生成する文字列リテラルを書きたいときには、「エスケープシーケンス」(escape sequence) と呼ばれる文字列が使われます。エスケープシーケンスは、必ず、バックスラッシュ(\) という文字で始まります[1]。

エスケープシーケンスには、次のようなものがあります。

`\a`	ビープ音	`\f`	改ページ
`\t`	水平タブ	`\b`	バックスペース
`\'`	一重引用符	`\"`	二重引用符
`\n`	改行	`\r`	キャリッジリターン
`\\`	バックスラッシュ	`\c`X	コントロール文字 (Ctrl+X)
`\`*ooo*	8 進数 *ooo* に対応する文字	`\x`*hh*	16 進数 *hh* に対応する文字

エスケープシーケンスを一重引用符で囲んだものは、そのエスケープシーケンスが意味している文字を生成する文字リテラルになります。

```
Prelude> '\n'
'\n'
```

文字列リテラルの中にエスケープシーケンスが含まれていた場合、その文字列リテラルによって生成される文字列は、そのエスケープシーケンスが意味している文字を含むものになります。

```
Prelude> "namako\numiushi\nkurage"
"namako\numiushi\nkurage"
```

2.2.8 真偽値データ構成子

真偽値に与えられた名前は、「真偽値データ構成子」(Boolean data constructor) と呼ばれます。

真偽値データ構成子は、厳密に言えばリテラルではないのですが、リテラルと同じように式として評価することができて、それがあらわしている真偽値を生成します。

真偽値データ構成子としては、次の二つのものがあります。

`True`	真。
`False`	偽。

```
Prelude> True
True
Prelude> False
False
```

2.3 関数適用

2.3.1 関数

Haskell では、何らかの動作を意味しているデータのことを「関数」(function) と呼びます。

コンピュータは、関数というデータがあらわしている動作を実行することができます。

関数があらわしている動作をコンピュータに実行させることを、関数を「適用する」(apply) と言います。

関数というのはあくまでデータであって、それがあらわしている動作を実行するのはあくまでコンピュータです。しかし、プログラムを書く人間としては、「関数自体が、自分があらわしている動作を実行する」というイメージで考えるほうが思考が単純になります。ですから、このチュートリアルでも、これからは、関数自体が動作をするというイメージで説明をしていきたいと思います。

[1] バックスラッシュは、日本語の環境では円マーク (¥) で表示されることがあります。

2.3.2 ユーザー定義関数と組み込み関数

関数を生成して、それに名前を与えることを、関数を「定義する」(define) と言います。

関数に与えられた名前は、「関数名」(function name) と呼ばれます。

Haskell では、「関数定義」(function definition) と呼ばれる記述をプログラムの中に書くことによって、関数を自由に定義することができます（関数定義の書き方については、第 3 章で説明することにしたいと思います)。プログラムの中に関数定義を書くことによって定義された関数は、「ユーザー定義関数」(user-defined function) と呼ばれます。

「ユーザー定義関数」の対義語は、「組み込み関数」です。「組み込み関数」(built-in function) というのは、Haskell の処理系に組み込まれている関数のことです。組み込み関数は、関数定義を書かなくても利用することができます。

2.3.3 引数と戻り値

関数は、何らかのデータを受け取って動作します。関数が受け取るデータは、「引数」(argument) と呼ばれます（「引数」は「ひきすう」と読みます)。

関数 f を動作させて、それに引数としてデータ d を渡すことを、「f を d に適用する」(apply f to d) と言います。

関数は、自分の動作が終了したのちに、自分を適用した者にデータを返すことができます。関数が返すデータは、「戻り値」(return value) と呼ばれます。

2.3.4 関数適用の書き方

関数を適用したいときは、関数を適用するという動作をあらわす式を書きます。そのような式は、「関数適用」(function application) と呼ばれます。

関数適用は、

$$\boxed{\text{式}_f}\ \boxed{\text{式}_1}\ \boxed{\text{式}_2}\ \cdots$$

と書きます。式 $_f$ のところには、適用したい関数を求める式を書きます。そして、式 $_1$、式 $_2$、…… のところには、関数に渡す引数を求める式を書きます。

関数適用は式ですから、評価することができます。関数適用を評価すると、それがあらわしている、関数を引数に適用するという動作が実行されます。そして、適用された関数が返した戻り値が、関数適用の値になります。

関数に名前が与えられている場合、その名前は式として評価することができて、評価したときに値として得られるのは、その関数そのものです。

すべての組み込み関数には名前が与えられています。ですから、組み込み関数の名前と、その関数に渡す引数を求める式を書くことによって、その関数を引数に適用することができます。

`div` という組み込み関数は、引数として 2 個の整数を受け取って、1 個目の整数を 2 個目の整数で除算したときの商を戻り値として返します。

`div` は、戻り値として、かならず整数を返します。除算が整数の範囲で割り切れなかった場合、小数点以下は求めません。たとえば、`div` を使って 30 を 8 で除算した場合、戻り値は、3.75 ではなくて 3 になります。

```
Prelude> div 30 8
3
```

`mod` という組み込み関数は、引数として 2 個の整数を受け取って、1 個目の整数を 2 個目の整数で除算したときのあまりを戻り値として返します。

```
Prelude> mod 30 8
6
```

2.4 演算子適用

2.4.1 演算

Haskell の組み込み関数の中には、関数適用ではなくて、「演算子適用」(operator application) と呼ばれる式によって適用されるものもあります。

演算子適用によって適用される関数は、「演算」(operation) と呼ばれます。そして、演算に与えられた名前は、「演算子」(operator) と呼ばれます。

演算には、受け取る引数の個数が 1 個のものと、2 個のものとがあります。

1 個の引数を受け取る演算は「前置演算」(prefix operation) と呼ばれ、そのような演算に与えられた名前は、「前置演算子」(prefix operator) と呼ばれます。

2 個の引数を受け取る演算は「中置演算」(infix operation) と呼ばれ、そのような演算に与えられた名前は、「中置演算子」(infix operator) と呼ばれます。

2.4.2 演算子適用の書き方

前置演算を引数に適用する演算子適用は、

演算子 式

と書きます。この形の演算子適用を評価すると、演算子の右側に書かれた式の値に対して演算が適用されて、その演算の戻り値が演算子適用の値になります。

中置演算を引数に適用する演算子適用は、

式 $_1$ 演算子 式 $_2$

と書きます。この形の演算子適用を評価すると、式 $_1$ と式 $_2$ の値に対して演算が適用されて、その演算の戻り値が演算子適用の値になります。

2.4.3 算術演算

数値に対する計算を動作とする演算は、「算術演算」(arithmetic operation) と呼ばれ、そのような演算に与えられた名前は、「算術演算子」(arithmetic operator) と呼ばれます。

中置演算でかつ算術演算であるような演算としては、次のようなものがあります。

a `+` b a と b とを足し算（加算）する。

a `-` b a から b を引き算（減算）する。

a `*` b a と b とを掛け算（乗算）する。

a `/` b a を b で割り算（除算）する。整数の範囲で割り切れない場合は小数点以下も求める。

a `**` b a の b 乗（べき乗）を求める。戻り値は常に浮動小数点数になる。

a `^` n a の n 乗（べき乗）を求める。n は整数でなければならない。a も整数だった場合は、戻り値も整数になる。

```
Prelude> 30+8
38
Prelude> 30-8
22
Prelude> 30*8
240
Prelude> 30/8
3.75
Prelude> 3**4
81.0
Prelude> 3^4
81
```

2.4.4 優先順位

ひとつの式の中に 2 個以上の演算子が含まれている場合、その式はどのように解釈されるのでしょうか。たとえば、

```
2+3*4
```

という式は、

`2+3` `*4`

という構造なのでしょうか。それとも、

`2+` `3*4`

という構造なのでしょうか。

この問題は、個々の演算子が持っている「優先順位」(precedence) と呼ばれるものによって解決されます。

優先順位というのは、演算子が左右の式と結合する強さのことだと考えることができます。優先順位が高い演算子は、それが低い演算子よりも、より強く左右の式と結合します。

`*` と `/` は、`+` と `-` よりも高い優先順位を持っています。ですから、

```
2+3*4
```

という式は、

```
2+[3*4]
```

という構造だと解釈されます。

```
Prelude> 2+3*4
14
```

さらに、`**` と `^` は、`*` と `/` よりも高い優先順位を持っています。ですから、

```
2*3^4
```

という式は、

```
2*[3^4]
```

という構造だと解釈されます。

```
Prelude> 2*3^4
162
```

2.4.5　結合規則

ひとつの式の中に同じ優先順位を持っている 2 個以上の演算子が含まれている場合、その式はどのように解釈されるのでしょうか。たとえば、

```
10-5+2
```

という式は、

```
[10-5]+2
```

という構造なのでしょうか。それとも、

```
10-[5+2]
```

という構造なのでしょうか。

この問題は、同一の優先順位を持つ演算子が共有している「結合規則」(associativity) と呼ばれる性質によって解決されます。

結合規則には、「左結合」(left-associativity) と「右結合」(right-associativity) という二つのものがあります。左結合というのは、左右の式と結合する強さが左にあるものほど強くなるという性質で、右結合というのは、それが右にあるものほど強くなるという性質です。

`+`、`-`、`*`、`/` の結合規則は、左結合です。したがって、

```
10-5+2
```

という式は、

```
[10-5]+2
```

という構造だと解釈されます。

```
Prelude> 10-5+2
7
```

`**` と `^` の結合規則は、右結合です。したがって、

```
2^3^4
```

という式は、

```
2^[3^4]
```

という構造だと解釈されます。

```
Prelude> 2^3^4
2417851639229258349412352
```

2.4.6 丸括弧

ところで、2 と 3 とを足し算して、その結果と 4 とを掛け算したい、というときは、どのような式を書けばいいのでしょうか。先ほど説明したように、+ と * とでは、* のほうが優先順位が高くなっていますので、

```
2+3*4
```

と書いたのでは、期待した結果は得られません。

演算子の優先順位や結合規則に縛られずに、自分が望んだとおりに式を解釈してほしい場合は、ひとまとまりの式だと解釈してほしい部分を、丸括弧 (()) で囲みます。そうすると、丸括弧で囲まれている部分は、演算子の優先順位や結合規則とは無関係に、ひとまとまりの式だと解釈されます。

```
Prelude> (2+3)*4
20
Prelude> (2*3)^4
1296
Prelude> 10-(5+2)
3
Prelude> (2^3)^4
4096
```

演算ではない関数の名前というのは、きわめて優先順位の高い演算子だと考えることができます。ですから、演算ではない関数に渡す引数を求める式として、関数適用または演算子適用を書く場合は、それを丸括弧で囲む必要があります。

```
Prelude> div (mod 100 40) 2
10
Prelude> div (12+18) 5
6
```

2.4.7 符号の反転

- という前置演算は、数値の符号（プラスかマイナスか）を反転させる算術演算です。

```
Prelude> -(3+5)
-8
Prelude> -(3-5)
2
```

2.4.8 中置演算子を普通の関数名にする方法

中置演算子は、普通の関数名として、普通の関数適用の中に書くことも可能です。

中置演算子を丸括弧で囲んだものは、普通の関数名と同じように、普通の関数適用の最初の式として書くことができます。この場合、1 個目の引数が中置演算の左側の引数になって、2 個目の引数が中置演算の右側の引数になります。

```
Prelude> (+) 30 8
38
```

2.4.9 関数名を中置演算子にする方法

それとは逆に、普通の関数名を中置演算子にすることも可能です。

関数名をバッククオート (‘) で囲んだものは、中置演算子として、演算子適用の中に書くことができます。この場合、中置演算子の左側に書いた式の値が 1 個目の引数になって、右側に書いた式の値が 2 個目の引数になります。

```
Prelude> 30 ‘mod‘ 8
6
```

2.5 タプル

2.5.1 タプルの基礎

任意の型を持つ 2 個以上のデータを並べることによってできる列は、「タプル」(tuple) と呼ばれます。

タプルを構成している個々のデータは、そのタプルの「要素」(element) と呼ばれます。

タプルを構成している要素の個数は、そのタプルの「長さ」(length) と呼ばれます。

2.5.2 タプル表記

タプルは、「タプル表記」(tuple notation) と呼ばれる式を書くことによって生成することができます。

タプル表記は、2 個以上の式をコンマで区切って並べて、その全体を丸括弧で囲むことによってできる、

(式 , 式 , ···)

という形の式です。タプル表記を評価すると、その中の個々の式が評価されて、それらの式の値を、式と同じ順序で並べたタプルが生成されて、そのタプルが、そのタプル表記の値になります。

たとえば、

```
(True, 68, 2.07, 'M')
```

というタプル表記を評価すると、その値として、`True` という真偽値、67 という整数、2.08 という浮動小数点数、`M` という文字、という 4 個のデータをこの順序で並べることによってできるタプルが、その値として得られます。

```
Prelude> (True, 68, 2.07, 'M')
(True,68,2.07,'M')
```

1 個のタプルは 1 個のデータですから、タプルを要素として含むタプルを作ることも可能です。たとえば、

```
(81, ('B', False), 6.57)
```

という式を評価することによって生成されるタプルは、2 個目の要素としてタプルを含んでいます。

```
Prelude> (81, ('B', False), 6.57)
(81,('B',False),6.57)
```

1 個の式を丸括弧で囲んだものは、式ではありますが、タプル表記ではありません。このような式を評価することによって得られる値は、丸括弧の中の式の値と同じものであって、タプルではありません。

```
Prelude> (81)
81
```

同じように、0 個の式を丸括弧で囲んだもの、つまり `()` というものも、式ではありますが、タプル表記ではありません。この式を評価すると、その値として、「ユニット」(unit) と呼ばれるデータが得られます。

```
Prelude> ()
()
```

ユニットが持つ型は、「ユニット型」(unit type) と呼ばれます。ユニット型の要素は、ユニットのみです。ユニット型は、`()` という型式によってあらわされます。

2.5.3 タプルの型

タプルの型は、それを構成している要素の型から構成される複合的な型です。すべてのタプルの型は、総称して「タプル型」(tuple type) と呼ばれます。

タプルの型を記述する型式は、

(型式 , 型式 , ···)

というように、それぞれの要素の型を記述する型式をコンマで区切って並べて、その全体を丸括弧で囲むことによって記述します。たとえば、

```
(True, 68, 2.07, 'M')
```

というタプルの型は、

```
(Bool, Int, Float, Char)
```

という型式によって記述することができて、

```
(81, ('B', False), 6.57)
```

というタプルの型は、

```
(Int, (Char, Bool), Float)
```

という型式によって記述することができます。

2.5.4 タプルを処理する関数

タプルを処理する Haskell の組み込み関数としては、`fst` と `snd` があります。どちらも、長さが 2 のタプルを引数として受け取って、その要素のうちのひとつを戻り値として返します。

`fst` は、長さが 2 のタプルを引数として受け取って、そのタプルの最初の要素を戻り値として返します。

```
Prelude> fst (37, 'B')
37
```

`snd` は、長さが 2 のタプルを引数として受け取って、そのタプルの 2 個目の要素を戻り値として返します。

```
Prelude> snd (37, 'B')
'B'
```

2.6 選択

2.6.1 選択の基礎

「何らかの条件にもとづいて、いくつかの動作の候補の中からひとつの動作を選んで実行する」という動作は、「選択」(selection) と呼ばれます。

選択をするためには、何らかの条件についての判断をする必要があります。Haskell の処理系には、条件の判断に使われるさまざまな演算が組み込まれています。それらの演算は、戻り値として、条件が成り立っているならば真、成り立っていないならば偽を返します。

2.6.2 比較演算

二つのデータのあいだに何らかの関係があるという条件が成り立っているかどうかを調べる、という動作をする中置演算は、「比較演算」(comparison operation) と呼ばれ、そのような演算に与えられた名前は、「比較演算子」(comparison operator) と呼ばれます。

比較演算子の優先順位は、加算や乗算などの演算子よりも低くなっています。

比較演算を二つのデータに適用すると、それらのデータのあいだに関係が成り立っているかどうかという判断が実行されます。そして、比較演算は、関係が成り立っているならば真、成り立っていないならば偽を、戻り値として返します。

比較演算としては、次のようなものがあります。

a `>` b　　a は b よりも大きい。

a `<` b　　a は b よりも小さい。

a `>=` b　　a は b よりも大きいか、または a と b とは等しい。

a `<=` b　　a は b よりも小さいか、または a と b とは等しい。

a `==` b　　a と b とは等しい。

a `/=` b　　a と b とは等しくない。

```
Prelude> 8 > 5
True
Prelude> 5 > 8
```

```
False
Prelude> 5 > 5
False
Prelude> 5 >= 5
True
Prelude> 5 == 5
True
Prelude> 5 == 8
False
Prelude> 5 /= 8
True
Prelude> 5 /= 5
False
```

大小関係があるのは、数値と数値とのあいだだけではありません。文字と文字とのあいだ、文字列と文字列とのあいだ、タプルとタプルとのあいだにも大小関係があります。

文字と文字とのあいだの大小関係は、それぞれの文字に割り当てられている文字コードの大小関係と同じです。

```
Prelude> 'B' > 'A'
True
Prelude> 'A' > 'B'
False
```

文字列と文字列とのあいだの大小関係は、それらを構成している文字を先頭から順番に比較していって、最初に発見された等しくない文字の大小関係によって決まります。このような大小関係にもとづいて、小さいものから大きいものへ文字列を並べる順序は、「辞書式順序」(lexicographical order) と呼ばれます。

```
Prelude> "stay" > "star"
True
Prelude> "star" > "stay"
False
```

タプルとタプルとのあいだの大小関係も、文字列の場合と同じように、それらを構成している要素を先頭から順番に比較していって、最初に発見された等しくない要素の大小関係によって決まります。

```
Prelude> (7, 6, 2, 8) > (7, 6, 2, 4)
True
Prelude> (7, 6, 2, 4) > (7, 6, 2, 8)
False
```

2.6.3 論理演算

引数が真偽値で、戻り値も真偽値であるような中置演算は、「論理演算」(Boolean operation) と呼ばれ、そのような演算に与えられた名前は、「論理演算子」(Boolean operator) と呼ばれます。

Haskell には、次の二つの論理演算があります。

a `&&` b　a かつ b である。

a `||` b　a または b である。

`&&` と `||` は、関係演算子よりも低い優先順位を持っています。そして、`&&` は、`||` よりも高い優先順位を持っています。

`&&` は、「論理積演算子」(logical AND operator) と呼ばれます。これは、二つの条件が両方とも成り立っているかどうかを判断したいとき、つまり、A かつ B という条件が成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
True  &&  True   ⟶  True
True  &&  False  ⟶  False
False &&  True   ⟶  False
False &&  False  ⟶  False
```

`||` は、「論理和演算子」(logical OR operator) と呼ばれます。これは、二つの条件のうちの少なくとも一つが成り立っているかどうかを判断したいとき、つまり、A または B という条件が

成り立っているかどうかを判断したいときに使われる論理演算子で、次のような動作をします。

```
True   ||  True     ⟶  True
True   ||  False    ⟶  True
False  ||  True     ⟶  True
False  ||  False    ⟶  False
```

2.6.4 論理否定関数

真偽値を反転させたいとき、つまり、A ではないという条件が成り立っているかどうかを判断したいときには、「論理否定関数」(Boolean negation function) と呼ばれる、`not` という関数が使われます。この関数は、次のような動作をします。

```
not True     ⟶  False
not False    ⟶  True
```

2.6.5 条件式

選択は、「条件式」(conditional expression) と呼ばれる式を書くことによって記述することができます。

条件式は、

`if` 式$_1$ `then` 式$_2$ `else` 式$_3$

と書きます。式$_1$ は、真偽値を求める式でないといけません。また、式$_2$ と式$_3$ は、同じ型のデータを求める式でないといけません。

条件式を評価すると、まず最初に式$_1$ が評価されます。もしもその値が真だった場合は、式$_2$ が評価されて、その値が条件式全体の値になります。式$_1$ の値が偽だった場合は、式$_3$ が評価されて、その値が条件式全体の値になります。

```
Prelude> if True then 5 else 8
5
Prelude> if False then 5 else 8
8
```

2.6.6 多肢選択

選択の対象となる動作が 3 個以上あるような選択は、「多肢選択」(multibranch selection) と呼ばれます。

多肢選択は、`else` の右側に条件式を書くことによって記述することができます。

```
Prelude> if True then 5 else if True then 8 else 3
5
Prelude> if False then 5 else if True then 8 else 3
8
Prelude> if False then 5 else if False then 8 else 3
3
```

第 3 章　関数定義

3.1　関数定義の基礎

3.1.1 関数定義についての復習

第 2.3.2 項で説明したように、関数を生成して、それに名前を与えることを、関数を「定義する」(define) と言います。

関数に与えられた名前は、「関数名」(function name) と呼ばれます。

Haskell では、「関数定義」(function definition) と呼ばれる記述をプログラムの中に書くことによって、関数を自由に定義することができます。

Haskell の処理系に組み込まれている関数のことを「組み込み関数」(built-in function) と呼ぶのに対して、プログラムの中に関数定義を書くことによって定義された関数は、「ユーザー定義関数」(user-defined function) と呼ばれます。

3.1.2　識別子

プログラムの中で何かに名前を与える場合、その名前は、名前を作るための規則にもとづいて作る必要があります。名前を作るための規則にもとづいて作られた名前は、「識別子」(identifier) と呼ばれます。

識別子は、次のような規則に従って作ることになっています。

- 識別子を作るために使うことのできる文字は、英字、数字、アンダースコア (_)、一重引用符 (’) です。
- 識別子の先頭の文字として使うことはできるのは、英字とアンダースコアです。
- 予約語 (reserved word) と同じものは識別子としては使えません。「予約語」(reserved word) というのは、用途があらかじめ予約されている単語のことで、次のようなものがあります。

```
case     deriving  import   infixr    newtype  where
class    do        in       instance  of
data     else      infix    let       then
default  if        infixl   module    type
```

- _というもの、すなわち、1 個のアンダースコアだけでできたものを識別子として使うことはできません。これは、「ワイルドカードパターン」(wild card pattern) と呼ばれるもので、特殊な用途で使われます。ワイルドカードパターンについては、第 3.3.5 項で説明することにしたいと思います。

識別子として使うことのできるものの例としては、次のようなものがあります。

```
a  A  namako  a8  a_b  a’b  doOrDoNot
```

英字の大文字と小文字は区別されますので、たとえば、`a` と `A` は、異なる識別子とみなされます。

空白を含む識別子を作ることはできませんので、複数の単語から構成される識別子を作りたいときは、通常、最後の例のように、単語と単語とのあいだの空白をすべて削除して、その代わりに単語の先頭の文字を大文字にする、という方法が使われます（先頭の単語の先頭の文字を大文字にするかどうかということについては、第 3.1.3 項で説明します)。複数の単語から構成される識別子のこのような作り方は、「キャメルケース」(camel case) と呼ばれます。

識別子として使うことのできないものの例としては、次のようなものがあります。

`nam@ko`　使うことのできない文字を含んでいる。

`8a`　先頭の文字が数字。

`else`　同じ予約語が存在する。

`_`　ワイルドカードパターンである。

3.1.3　識別子の分類

識別子は、次の 2 種類のものに分類されます。

- 変数識別子 (variable identifier)
- 構成子識別子 (constructor identifier)

これらの種類の相違点は、先頭の文字です。変数識別子は、先頭の文字が英字の小文字またはアンダースコアです。それに対して、構成子識別子は、先頭の文字が英字の大文字です。

識別子の種類は、それを名前として与える対象が何であるか、ということによって使い分ける必要があります。関数などのデータには、変数識別子を名前として与えます。

3.1.4　データ定義

Haskell では、識別子を名前としてデータに与えることを、識別子をデータに「束縛する」(bind) と言います。

変数識別子をデータに束縛する記述は、「データ定義」(data definition) と呼ばれます。

Haskell では、関数というのもデータの一種ですから、データ定義を書くことによって変数識別子を関数に束縛することができます。関数定義というのは、変数識別子を関数に束縛するデータ定義のことです。

データ定義は、「型シグネチャー宣言」(type signature declaration) と呼ばれるものと、1 個以上の「等式」(equation) と呼ばれるものから構成されます。

型シグネチャー宣言は、データ定義の先頭に書きます。等式を書くのは、その下です。

3.1.5 型シグネチャー宣言

型シグネチャー宣言は、変数識別子をどのような型のデータに束縛するのか、ということについての記述です。

Haskell の処理系は、データの型が何であるかということを推論する、「型推論」(type inference) という機能を持っています。ですから、多くの場合、型シグネチャー宣言は省略することが可能です。しかし、型シグネチャー宣言は、プログラムを読む人間にとって有益な情報を与えてくれますので、できるだけ書いておくほうがいいでしょう。

型シグネチャー宣言は、

変数識別子 :: 型式

と書きます。「変数識別子」のところには、データに束縛される変数識別子を書いて、「型式」のところには、束縛するデータの型をあらわす型式を書きます。

たとえば、`hundred` という変数識別子を `Int` のデータに束縛するデータ定義の先頭には、

```
hundred :: Int
```

という型シグネチャー宣言を書きます。

3.1.6 等式

等式は、変数識別子をどのようなデータに束縛するのか、ということについての記述です。

等式は、

変数識別子 = 式

と書きます。この中の「変数識別子」というところには何らかの変数識別子を書いて、「式」のところには何らかの式を書きます。そうすると、イコール (`=`) の左側に書かれた変数識別子が、イコールの右側に書かれた式を評価することによって得られた値に束縛されます。たとえば、

```
hundred :: Int
hundred = 100
```

というデータ定義を書くことによって、`hundred` という変数識別子を 100 という `Int` のデータに束縛することができます。

データに束縛されている変数識別子は、式として評価することができます。変数識別子を式として評価すると、その値として、その変数識別子が束縛されているデータが得られます。たとえば、`hundred` という変数識別子が 100 という `Int` のデータに束縛されているならば、その変数識別子を評価すると、100 という `Int` のデータがその値として得られます。

それでは、`hundred` という変数識別子を 100 に束縛するプログラムを GHCi に実行させてみましょう。

まず、先ほどのデータ定義を、`hundred.hs` という名前のファイルに保存してください。Haskell のプログラムを保存するファイルの名前には、このように、`.hs` という拡張子を付けることになっています。次に、コマンドを入力するためのアプリ（Linux や Mac OS ならばターミナル、Windows ならばコマンドプロンプト）に、

```
ghci hundred.hs
```

というコマンドを入力してください。そうすると、GHCi が `hundred.hs` の中のプログラムを実行して、そののち、

```
*Main>
```

というプロンプトを表示します。

この時点で、すでに `hundred` という変数識別子が 100 に束縛されているはずですので、GHCi にそれを評価させてみましょう。

```
*Main> hundred
100
```

このように、式の値として 100 が表示されます。

ファイルに格納されているプログラムを GHCi に実行させる方法としては、GHCi を起動するコマンドの引数としてファイル名を書くという方法のほかに、`:load` または `:l` というコマンドを使うという方法もあります。

それでは、`:l` というコマンドを使って、ファイルに格納されているプログラムを GHCi に実行させてみましょう。

まず、`ghci` というコマンドで、GHCi を起動してください。そして、GHCi に対して、

```
:l hundred.hs
```

というコマンドを入力してください。そうすると、`hundred.hs` の中のプログラムが実行されて、プロンプトが、

```
*Main>
```

に変わります。こののちに `hundred` という変数識別子を入力すると、その値として 100 が出力されます。

3.2　関数定義の書き方

3.2.1　関数定義の書き方の基礎

第 3.1.4 項で説明したように、Haskell では、「データ定義」(data definition) と呼ばれる記述を書くことによって、変数識別子をデータに束縛することができます。

Haskell では、関数というのもデータの一種ですから、変数識別子を関数に束縛する場合も、データ定義を書きます。「関数定義」(function definition) というのは、変数識別子をデータに束縛するデータ定義のことです。

関数定義を書くためには、関数というデータの型を記述する型式を型シグネチャー宣言の中に書くこと、そして、変数識別子を関数に束縛する等式を書くことが必要になります。そこで、この節では、関数の型を記述する型式の書き方と、変数識別子を関数に束縛する等式の書き方について説明することにしたいと思います。

3.2.2　関数型を記述する型式

それでは、まず、関数の型を記述する型式の書き方について説明しましょう。

関数の型は、引数の型と戻り値の型から構成される複合的な型です。関数の型は、総称して「関数型」(function type) と呼ばれます。

関数型を記述する型式は、

型式 $_1$ -> 型式 $_2$ -> ･･･ -> 型式 $_n$ -> 型式 $_r$ ->

と書きます。型式 $_1$ から型式 $_n$ までのところには、関数に渡す引数の型をあらわす型式を書きます。型式を書く順番は、関数に引数を渡す順番と同じでないといけません。そして、型式 $_r$ のところには、関数の戻り値の型をあらわす型式を書きます。

たとえば、引数として 1 個の整数を受け取って、戻り値として真偽値を返す関数の型は、

```
Int -> Bool
```

という型式で記述することができます。同じように、1 個目の引数として文字、2 個目の引数として整数を受け取って、戻り値として文字列を返す関数の型は、

```
Char -> Int -> String
```

という型式で記述することができます。同じように、1 個目の引数として、1 個目の要素が文字列で 2 個目の要素が整数のタプル、2 個目の引数として真偽値を受け取って、戻り値として文字を返す関数の型は、

```
(String, Int) -> Bool -> Char
```

という型式で記述することができます。

3.2.3　変数識別子を関数に束縛する等式

次に、変数識別子を関数に束縛する等式の書き方について説明しましょう。

変数識別子を関数に束縛する場合は、等式として、

変数識別子 $_f$	変数識別子 $_1$	変数識別子 $_2$	・・・	変数識別子 $_n$	=	式

という形のものを書きます。変数識別子 $_f$ のところには、名前として関数に与える変数識別子を書きます。そして、イコール (=) の右側の「式」というところには、関数がデータに適用されたときに評価されるべき式を書きます。この式を評価することによって得られた値が、関数の戻り値になります。

変数識別子 $_1$ から変数識別子 $_n$ までのところには、関数が受け取った引数に名前として与える変数識別子を書きます。これらの変数識別子は、「仮引数」(formal argument) と呼ばれます。関数を引数を適用すると、その関数の仮引数は、適用された引数に束縛されることになります。

イコール (=) の右側に書く式の中には、式として仮引数を書くことができます。仮引数を評価すると、それが束縛されているデータが値として得られます。たとえば、

```
square :: Int -> Int
square n = n*n
```

という関数定義によって定義された square という関数を、

```
square 7
```

という関数適用で、7 という整数に適用したとすると、 n という仮引数が 7 に束縛されますので、 n*n という式を評価すると、値として 49 が得られます。したがって、この場合、 square は戻り値として 49 を返すことになります。

それでは、 square を定義する関数定義を square.hs というファイルに保存して、GHCi を使って、その動作を確かめてください。

実行例

```
*Main> square 7
49
```

ちなみに、第 1.3.2 項で説明したように、 Int というのは、コンピュータのハードウェアによって上限と下限が定められている整数の型です。したがって、 square の型を、

```
Int -> Int
```

と書いたとすると、計算の結果が Int の範囲から外へ出てしまった場合、正しい結果が得られません。たとえば、このチュートリアルの筆者が使っているハードウェアでは、

```
*Main> square 100000
1410065408
```

というように、100000 の 2 乗を正しく求めることができません。

Int の代わりに Integer を使うことによって、ハードウェアによる制限を超える整数を扱うことができるようになります。

それでは、次のプログラムを入力して、動作を確かめてみましょう。

プログラムの例 square2.hs

```
square2 :: Integer -> Integer
square2 n = n*n
```

実行例

```
*Main> square2 10000000000000000000
100000000000000000000000000000000000000
```

次に、2 個以上の引数を受け取る関数を定義してみましょう。

次のプログラムの中で定義されている trapezoid という関数は、台形 (trapezoid) の上底、下底、高さを引数として受け取って、その台形の面積を戻り値として返します。

プログラムの例 trapezoid.hs

```
trapezoid :: Float -> Float -> Float -> Float
trapezoid b1 b2 h = (b1+b2)*h/2
```

実行例

```
*Main> trapezoid 3 6 5
22.5
```

実は、2 個以上の引数を受け取る関数も、実際に受け取っている引数は最初の 1 個だけです。この点に関しては、第 5.1.2 項で、もう少し詳しく説明することにしたいと思います。

3.3　パターン

3.3.1　パターンの基礎

第 3.2.3 項で、関数を定義したいときは、等式として、

変数識別子 $_f$　変数識別子 $_1$　変数識別子 $_2$　・・・　変数識別子 $_n$　=　式

という形のものを書くと説明しました。しかし、この説明は厳密には正しくありません。もう少し厳密に書くと、等式として書くものは、

変数識別子　パターン $_1$　パターン $_2$　・・・　パターン $_n$　=　式

という形のものです。この中の「パターン」というところには、「パターン」(pattern) と呼ばれるものを書くことができます。

パターンというのは、データと一致するかどうかを確かめることのできる記述のことです。等式のイコールの右側に書かれた式は、引数とパターンとが一致した場合にのみ評価されます。

パターンとして変数識別子を書くと、それは、どんな引数とも一致することになります。

3.3.2　パターンとしてのリテラル

変数識別子だけではなくて、リテラルも、パターンとして書くことができます。

パターンとしてリテラルを書くと、それは、そのリテラルがあらわしているデータのみと一致します。

次のプログラムの中で定義されている seven という関数は、7 という整数に適用すると、

```
Congratulations!
```

という文字列を戻り値として返します。

プログラムの例　seven.hs

```
seven :: Int -> String
seven 7 = "Congratulations!"
```

実行例

```
*Main> seven 7
"Congratulations!"
```

7 以外のデータに seven を適用すると、GHCi は、

```
Non-exhaustive patterns in function seven
```

というエラーメッセージを表示します。このメッセージは、「パターンが網羅的ではない」という意味です。

3.3.3　2 個以上の等式から構成される関数定義

関数定義の中には、2 個以上の等式を書いてもかまいません。

関数定義の中に 2 個以上の等式が含まれている場合、それらの等式は、上から順番に、その中のパターンと引数とが一致するかどうかを試されていきます。そして、引数と一致するパターンを持つ等式が発見された場合、その等式の中の式が評価されて、その値が戻り値になります。

引数と一致したパターンを持つ等式の下に、さらに等式が残っていたとしても、それらの等式は無視されます。

パターンと引数とが一致した場合で、そのパターンが識別子を含んでいる場合は、その識別子が引数（の一部分）に束縛されて、そののち式が評価されます。

たとえば、次のプログラムの中で定義されている numeral という関数は、1 に適用すると one、2 に適用すると two、3 に適用すると three という文字列を返して、それら以外の整数に適用した場合は many という文字列を戻り値として返します。

プログラムの例　numeral.hs

```
numeral :: Int -> String
numeral 1 = "one"
numeral 2 = "two"
numeral 3 = "three"
numeral n = "many"
```

実行例

```
*Main> numeral 1
"one"
*Main> numeral 2
"two"
*Main> numeral 3
"three"
*Main> numeral 4
"many"
```

3.3.4 特定の長さのタプルと一致するパターン

特定の長さのタプルと一致するパターンは、

(パターン , パターン , ···)

というように、要素と一致するパターンをコンマで区切って並べて、その全体を丸括弧で囲むことによって記述します。たとえば、

```
(x, y)
```

というパターンは、長さが 2 のタプルと一致して、x という変数識別子は 1 個目の要素に束縛されて、y という変数識別子は 2 個目の要素に束縛されます。

次のプログラムの中で定義されている swap という関数は、長さが 2 のタプルに適用すると、1 個目の要素と 2 個目の要素を入れ替えることによってできるタプルを戻り値として返します。

プログラムの例 swap.hs

```
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

実行例

```
*Main> swap (27, True)
(True,27)
*Main> swap ("umiushi", (68, False))
((68,False),"umiushi")
```

3.3.5 ワイルドカードパターン

第 3.1.2 節で説明したように、_というもの、すなわち、1 個のアンダースコアだけでできたものは、「ワイルドカードパターン」(wild card pattern) と呼ばれます。

ワイルドカードパターンは、変数識別子と同じように、任意のデータと一致するパターンです。しかし、ワイルドカードパターンは変数識別子ではありませんので、データには束縛されません。

ワイルドカードパターンは、任意のデータと一致することが必要だけれども、一致したデータ自体は必要がない、という場合に使われます。

次のプログラムの中で定義されている third という関数は、長さが 3 のタプルに適用すると、その 3 番目の要素を戻り値として返します。

プログラムの例 third.hs

```
third :: (a, b, c) -> c
third (_, _, x) = x
```

実行例

```
*Main> third (47, True, "tako")
"tako"
```

この関数は、タプルの 1 番目と 2 番目の要素を必要としませんので、それと一致させるパターンとしては、変数識別子ではなくてワイルドカードパターンを使っています。

3.3.6 as パターン

変数識別子を一部分として含むパターンとデータとが一致した場合、その変数識別子は、データの全体ではなくて、その一部分に束縛されることになります。たとえば、(x, y) というパターンとデータとが一致した場合、x はタプルの 1 個目の要素に、y は 2 個目の要素に束縛されます。ですから、データの全体を求めたいという場合は、(x, y) という式によって全体を復元することが必要になります。

データを分解してしまったあとで再び全体を復元する式を書くというのは面倒だ、と思われる場合に便利なのが、「as パターン」(as pattern) と呼ばれるパターンです。このパターンは、識別子をデータの一部分に束縛すると同時に、別の識別子をデータの全体に束縛します。

as パターンは、

変数識別子 @ パターン

と書きます。この中の「変数識別子」のところにはデータの全体に束縛する変数識別子を書いて、「パターン」のところには識別子をデータの一部分に束縛するためのパターンを書きます。たとえば、

```
all@(x, y)
```

という as パターンは、長さが 2 のタプルと一致して、x は 1 個目の要素に、y は 2 個目の要素に、そして all はタプルの全体に束縛されます。

次のプログラムの中で定義されている firstSecondAll という関数は、長さが 2 のタプルに適用すると、その 1 番目の要素、2 番目の要素、そしてタプルの全体から構成されるタプルを戻り値として返します。

プログラムの例 firstsecondall.hs

```
firstSecondAll :: (a, b) -> (a, b, (a, b))
firstSecondAll all@(x, y) = (x, y, all)
```

実行例

```
*Main> firstSecondAll (83, "sazae")
(83,"sazae",(83,"sazae"))
```

3.3.7 case 式

パターンは、関数が受け取った引数と一致するかどうかを調べたいときだけではなくて、式の値と一致するかどうかを調べたいときにも使うことができます。

パターンと式の値とが一致するかどうかを調べたいときは、「case 式」(case expression) と呼ばれる式を書きます。

case 式は、

case 式 of 選択肢 ・・・

と書きます。この中の「式」というところには、パターンと一致するかどうかを調べたいデータを求める式を書きます。そして、「選択肢」というところには、

パターン -> 式

という形のものを書きます。選択肢は、何個でも並べて書くことができます。

Haskell の処理系は、case 式が与えられた場合、まず最初に、case と of とのあいだに書かれた式を評価します。そして、その式の値と、選択肢の中のパターンとが一致するかどうかを、選択肢が並んでいる順番のとおりに調べていきます。そして、一致するパターンを持つ選択肢が見つかったならば、その選択肢の中の式を評価して、その式の値を、case 式全体の値にします。パターンの中に変数識別子が含まれている場合、その変数識別子は、一致したデータに束縛されます。

次のプログラムの中で定義されている evenOdd という関数は、偶数に適用すると even、奇数に適用すると odd という文字列を戻り値として返します。

プログラムの例 evenodd.hs

```
evenOdd :: Int -> String
evenOdd n =
    case mod n 2 of
        0 -> "even"
        _ -> "odd"
```

実行例

```
*Main> evenOdd 6
"even"
*Main> evenOdd 9
"odd"
```

3.4 型変数

3.4.1 型変数の基礎

第 3.1.2 項で説明したように、英字の小文字またはアンダースコアで始まる識別子は「変数識別子」(variable identifier) と呼ばれます。

変数識別子は、型式の中に書くこともできます。型式の中に書かれた変数識別子は、「型変数」(type variable) と呼ばれます。

型変数は、「任意の型」という意味を持ちます。つまり、「ここはどんな型のデータでもかまわない」という意味です。ただし、ひとつの型式の中では、同じ型変数は同じ型を意味することになります。たとえば、

```
a -> a
```

という型式は、任意の型のデータを受け取って、それと同じ型のデータを返す関数の型をあらわします。

型変数としては、どんな変数識別子を使ってもかまわないのですが、通常は、a、b、c、d、……というように、1 文字の英小文字を使います。

3.4.2 多相的関数

型変数を使わなければ記述することができない型を持つ関数は、「多相的関数」(polymorphic function) と呼ばれます。

Haskell の処理系には、受け取った引数をそのまま返す、id という関数が組み込まれています。

```
Prelude> id 81
81
Prelude> id False
False
Prelude> id "hamaguri"
"hamaguri"
```

id は、任意の型のデータを受け取ることができます。ですから、この関数の型を型式で記述するためには、型変数を使って、

```
a -> a
```

と書く必要があります。したがって、この関数は多相的関数だということになります。

第 2.5.4 項で、長さが 2 のタプルから 1 個目の要素を取り出す、fst という組み込み関数を紹介しましたが、この関数の型を型式で記述するためには、型変数を使って、

```
(a, b) -> a
```

と書く必要があります。同じように、2 個目の要素を取り出す snd という組み込み関数の型をあらわす型式を書く場合も、

```
(a, b) -> b
```

というように、型変数が必要になります。ですから、この二つの組み込み関数は、どちらも多相的関数です。

3.4.3　多相的関数の定義

それでは、多相的関数を定義する関数定義を書いてみましょう。

次のプログラムの中で定義されている duplicate という関数は、任意の型のデータを受け取って、それを二つ並べることによってできるタプルを返します。

プログラムの例　duplicate.hs

```
duplicate :: a -> (a, a)
duplicate x = (x, x)
```

実行例

```
*Main> duplicate 47
(47,47)
*Main> duplicate True
(True,True)
*Main> duplicate "namako"
("namako","namako")
```

3.5　ガード

3.5.1　ガードの基礎

これまでに出てきたすべての等式は、戻り値を求めるための式を 1 個だけしか含んでいませんでしたが、実は、戻り値を求めるための式は、ひとつの等式の中に何個でも書くことができます。

ただし、ひとつの等式の中に、戻り値を求めるための式を 2 個以上書いたとしても、それらの式のうちで実際に評価されるのは、ひとつだけです。

ひとつの等式の中に、戻り値を求めるための式が 2 個以上書かれている場合、それらのうちで実際に評価される式は、「ガード」(guard) と呼ばれるものによって選択されます。

ガードは、形の上では単なる 1 個の式です。ただしそれは、値として真偽値が得られるものでないといけません。

ひとつの等式の中に、戻り値を求めるための式を 2 個以上書く場合、それぞれの式は、ガードとのペアになっている必要があります。

ガードは、書かれている順番のとおりに評価されていきます。そして、値として真が得られたガードが発見された場合、それとペアになっている式が評価されて、その値が戻り値になります。真が得られたガードよりも後ろにあるガードは、無視されます。

3.5.2　ガードと式とのペアの記述

ガードと式とのペアは、

| ガード = 式

と書きます。たとえば、

```
| x > 0 = x*2
```

というペアは、x がゼロよりも大きい場合は x*2 を返す、という意味になります。

次のプログラムの中で定義されている greet という関数は、24 時制の時刻（時のみ）を引数として受け取って、その時刻にふさわしい挨拶の言葉を戻り値として返します。

プログラムの例　greet.hs

```
greet :: Int -> String
greet hour
  | hour < 12 = "Good morning."
  | hour < 18 = "Good afternoon."
  | hour < 24 = "Good evening."
```

実行例

```
*Main> greet 6
"Good morning."
*Main> greet 15
"Good afternoon."
*Main> greet 21
```

```
"Good evening."
```

24 以上の整数に greet を適用した場合は、どのガードも真になりませんので、GHCi は、

```
Non-exhaustive patterns in function greet
```

というエラーメッセージを表示します。

3.5.3 常に真になるガード

評価すると常に真が得られるガードを書きたい場合は、 True という真偽値データ構成子を書いてもいいのですが、通常は、 otherwise と書きます。

otherwise は、組み込み関数です。これは、引数を受け取らないで、常に True を返します。

次のプログラムの中で定義されている sign という関数は、引数として整数を受け取って、その符号をあらわす文字列を戻り値として返します。

プログラムの例 sign.hs

```
sign :: Int -> String
sign n
  | n > 0     = "plus"
  | n < 0     = "minus"
  | otherwise = "zero"
```

実行例

```
*Main> sign 5
"plus"
*Main> sign (-5)
"minus"
*Main> sign 0
"zero"
```

3.5.4 ガードと条件式

ガードというのは、選択を記述するための方法のひとつです。

選択を記述するための方法としては、第 2.6.5 項で説明した、条件式というものもあります。ガードを使った関数定義は、ガードの代わりに条件式を使って書き直すことも可能です。

次のプログラムは、先ほど紹介したプログラムの中で定義されていた sign という関数の定義を、ガードの代わりに条件式を使って書き直したものです。

プログラムの例 sign2.hs

```
sign :: Int -> String
sign n =
    if      n > 0 then "plus"
    else if n < 0 then "minus"
    else               "zero"
```

3.6 再帰

3.6.1 再帰とは何か

この節では、「再帰」(recursion) と呼ばれるものについて説明したいと思います。

再帰というのは、全体と同じものが一部分として含まれているという性質のことです。再帰という性質を持っているものは、「再帰的な」(recursive) と形容されます。

ここに、1 台のカメラと 1 台のモニターがあるとします。まず、それらを接続して、カメラで撮影した映像がモニターに映し出されるようにします。そして次に、カメラをモニターの画面に向けます。すると、モニターの画面には、そのモニター自身が映し出されることになります。そして、映し出されたモニターの画面の中には、さらにモニター自身が映し出されています。このときにモニターの画面に映し出されるのは、再帰という性質を持っている映像、つまり再帰的な映像です。

また、先祖と子孫の関係も再帰的です。なぜなら、先祖と子孫との中間にいる人々も、やはり先祖と子孫の関係で結ばれているからです。

3.6.2　基底

再帰という性質を持っているものは、全体と同じものが一部分として含まれているわけですが、その構造は、内部に向かってどこまでも続いている場合もあれば、どこかで終わっている場合もあります。

再帰的な構造がどこかで終わっている場合、その中心には、その内部に再帰的な構造を持っていない何かがあります。そのような、再帰的な構造の中心にあって、その内部に再帰的な構造を持っていないものは、その再帰的な構造の「基底」(basis) と呼ばれます。

先祖と子孫の関係では、親子関係というのが、その再帰的な構造の基底になります。

3.6.3　関数の再帰的な定義

関数は、再帰的に定義することが可能です。関数を再帰的に定義するというのは、定義される当の関数を使って関数を定義するということです。再帰的な構造を持っている概念を取り扱う関数は、再帰的に定義するほうが、再帰的ではない方法で定義するよりもすっきりした記述になります。

関数を再帰的に定義する場合は、それが循環に陥ることを防ぐために、基底について記述した選択肢を作っておくことが必要になります。

3.6.4　階乗

n が 0 またはプラスの整数だとするとき、n から 1 までの整数をすべて乗算した結果、つまり、

$$n \times (n-1) \times (n-2) \times \cdots \times 1$$

という計算の結果は、n の「階乗」(factorial) と呼ばれて、$n!$ と書きあらわされます。ただし、$0!$ は 1 だと定義します。

たとえば、$5!$ は、

$$5 \times 4 \times 3 \times 2 \times 1$$

という計算をすればいいわけですから、120 ということになります。

階乗という概念は、再帰的な構造を持っています。なぜなら、階乗は、

$$\begin{cases} & 0! = 1 \\ n \geq 1 \text{ ならば} & n! = n \times (n-1)! \end{cases}$$

というように再帰的に定義することができるからです。

階乗を求める関数も、再帰的に定義することができます。次のプログラムは、階乗を求める `factorial` という関数を再帰的に定義しています。

プログラムの例　`factorial.hs`

```
factorial :: Integer -> Integer
factorial n
  | n == 0 = 1
  | n >= 1 = n * factorial (n-1)
```

実行例

```
*Main> factorial 5
120
```

3.6.5　フィボナッチ数列

第 0 項と第 1 項が 1 で、第 2 項以降はその直前の 2 項を足し算した結果である、という数列は、「フィボナッチ数列」(Fibonacci sequence) と呼ばれます。フィボナッチ数列の第 0 項から第 12 項までを表にすると、次のようになります。

n	0	1	2	3	4	5	6	7	8	9	10	11	12
第 n 項	1	1	2	3	5	8	13	21	34	55	89	144	233

フィボナッチ数列というのは再帰的な構造を持っている概念ですので、その第 n 項 (F_n) は、

$$\begin{cases} & F_0 = 1 \\ & F_1 = 1 \\ n \geq 2 \text{ ならば} & F_n = F_{n-2} + F_{n-1} \end{cases}$$

というように再帰的に定義することができます。

フィボナッチ数列の第 n 項を求める関数も、再帰的に定義することができます。次のプログラムは、フィボナッチ数列の第 n 項を求める `fibonacci` という関数を再帰的に定義しています。

プログラムの例 `fibonacci.hs`

```
fibonacci :: Integer -> Integer
fibonacci n
  | n == 0 = 1
  | n == 1 = 1
  | n >= 2 = fibonacci (n-2) + fibonacci (n-1)
```

実行例

```
*Main> fibonacci 7
21
```

3.6.6 最大公約数

n がプラスの整数で、m が 0 またはプラスの整数だとするとき、n と m の両方に共通する約数のうちで最大のもののことを、n と m の「最大公約数」(greatest common measure, GCM) と呼びます（m が 0 の場合は、n と m の最大公約数は n だと定義します)。たとえば、54 と 36 の最大公約数は 18 です。

n と m の最大公約数は、「ユークリッドの互除法」(Euclidean algorithm) と呼ばれる次のような再帰的な手順を実行することによって求めることができます。

- m が 0 ならば、n が、n と m の最大公約数である。
- m が 1 以上ならば、n を m で除算したときのあまりを求めて、その結果を r とする。そして、m と r の最大公約数を求めれば、その結果が n と m の最大公約数である。

次のプログラムは、2 個のプラスの整数の最大公約数を求める `gcm` という関数を、ユークリッドの互除法を使って定義しています。

プログラムの例 `gcm.hs`

```
gcm :: Integer -> Integer -> Integer
gcm n m
  | m == 0 = n
  | m >= 1 = gcm m (mod n m)
```

実行例

```
*Main> gcm 54 36
18
```

3.7 ローカルな識別子

3.7.1 スコープ

プログラムの中で、識別子がデータに束縛されているという関係が有効である範囲は、その識別子の「スコープ」(scope) と呼ばれます。

プログラムの一部分に限定されているスコープは「ローカルなスコープ」(local scope) と呼ばれ、そのようなスコープを持つ識別子は「ローカルな識別子」(local identifier) と呼ばれます。

それに対して、スコープがプログラムの全域に及んでいるスコープは「グローバルなスコープ」(global scope) と呼ばれ、そのようなスコープを持つ識別子は「グローバルな識別子」(global identifier) と呼ばれます。

関数定義の中で引数に束縛された識別子は、その関数定義の中だけというローカルなスコープを持つことになります。ただし、関数に束縛された識別子、つまり関数名は、それを定義した関

数定義の外側にもスコープが及びます。ですから、関数名は、それが関数定義の中の関数定義によって定義されているのでないならば、グローバルなスコープを持つことになります。

引数に束縛される識別子はローカルなスコープを持つことになるわけですが、それ以外にも、次のようなものを書くことによって、ローカルな識別子を作ることができます。

- `where` 節 (`where` clause)
- `let` 式 (`let` expression)

3.7.2　ブロック

`where` 節や `let` 式の中には、「ブロック」(block) と呼ばれるものを書く必要があります。

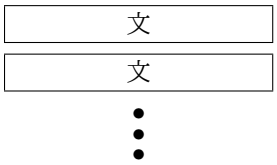
ブロックというのは、いくつかの記述をひとつにまとめた記述のことです。ブロックによってひとつにまとめられるそれぞれの記述は、「文」(statement) と呼ばれます。ブロックの中に文として何を書くことができるかということは、そのブロックが何の中に書かれているかということによって決まります。

ブロックには、二つの書き方があります。ひとつは中括弧とセミコロンを使う書き方で、もうひとつは改行を使う書き方です。

中括弧とセミコロンを使うブロックは、

`{` 文 `;` 文 `;` `···` `}`

という形の記述です。そして改行を使うブロックは、



という形の記述です。

行の先頭に空白を書くことによって、その行の本体を右にずらすことは、行の「インデント」(indentation) と呼ばれます。そして、インデントに使われた空白の個数は、そのインデントの「深さ」(depth) と呼ばれます。

改行を使ってブロックを書く場合、それを構成するそれぞれの文は、一定の深さでインデントする必要があります。インデントの深さを一定にするというのは、行の本体の先頭を縦方向に一直線に並べるということです。

3.7.3　`where` 節

ローカルな識別子を作る方法のひとつは、関数定義の末尾に「`where` 節」(`where` clause) と呼ばれる記述を書くことです。

`where` 節は、

`where` ブロック

と書きます。`where` 節のブロックの中には、データ定義を書くことができます。そのデータ定義によってデータに束縛された変数識別子は、その `while` 節を持つ関数定義の中だけというローカルなスコープを持つことになります。

収入に対する趣味の出費の比率は、「オタク係数」(otaku's coefficient) と呼ばれます。オタク係数は、

趣味の出費額 ÷ 収入 × 100

という計算によって求めることができます（単位はパーセント）。

次のプログラムの中で定義されている `otaku` という関数は、1 個目の引数として収入、2 個目の引数として趣味の出費を受け取って、オタク係数が家計にとって危険かどうかを判定するメッセージを戻り値として返します。

プログラムの例　`otaku.hs`

```
otaku :: Float -> Float -> String
otaku income outlay
  | coefficient <=  5 = "You are not an otaku."
```

```
  | coefficient <= 20 = "You are normal."
  | coefficient <= 80 = "You are dangerous."
  | otherwise         = "You are the otaku of otakus."
  where coefficient :: Float
        coefficient = outlay / income * 100
```

実行例

```
*Main> otaku 300000 10000
"You are not an otaku."
*Main> otaku 300000 50000
"You are normal."
*Main> otaku 300000 100000
"You are dangerous."
*Main> otaku 300000 250000
"You are the otaku of otakus."
```

3.7.4 let 式

ローカルな識別子を作る方法としては、where 節を書くという方法のほかに、「let 式」(let expression) と呼ばれる式を書くという方法もあります。

let 式は、

let [ブロック] in [式]

と書きます。let 式のブロックの中には、データ定義を書くことができます。そのデータ定義によってデータに束縛された変数識別子は、そのデータ定義を含む let 式の中だけというローカルなスコープを持つことになります。

while 節とは違って、let 式は式の一種です。したがって、let 式は評価することができて、評価すると、その結果として値が得られます。let 式の値は、in の右側に書かれた式の値です。

次のプログラムは、第 3.7.3 項で紹介した、オタク係数が家計にとって危険かどうかを判定する otaku という関数の定義を、let 式を使って書き直したものです。

プログラムの例 otaku2.hs

```
otaku :: Float -> Float -> String
otaku income outlay =
    let coefficient :: Float
        coefficient = outlay / income * 100
    in if coefficient <=  5 then
           "You are not an otaku."
       else if coefficient <= 20 then
           "You are normal."
       else if coefficient <= 80 then
           "You are dangerous."
       else
           "You are the otaku of otakus."
```

3.7.5 ローカルなスコープを持つ関数名

where 節や let 式のブロックの中には、関数ではないデータに識別子を束縛するデータ定義だけではなくて、関数定義を書くことも可能です。where 節や let 式のブロックの中に関数定義を書くことによって、ローカルな識別子を関数に束縛することができます。

次のプログラムの中で定義されている isPrime という関数は、2 以上の整数に適用すると、その整数が素数ならば True、素数ではないならば False を戻り値として返します。

プログラムの例 isprime.hs

```
isPrime :: Integer -> Bool
isPrime n = isPrimeSub n (n-1)
    where isPrimeSub :: Integer -> Integer -> Bool
          isPrimeSub n m
            | m < 2        = True
            | mod n m == 0 = False
            | otherwise    = isPrimeSub n (m-1)
```

実行例

```
*Main> isPrime 21
False
*Main> isPrime 23
True
```

isPrime が補助的に使っている isPrimeSub という関数を定義する関数定義は、where 節の中に書かれています。ですから、isPrimeSub という識別子は、isPrime を定義する関数定義の中だけというスコープを持つことになります。

次のプログラムは、let 式を使って isPrime の定義を書き直したものです。

プログラムの例　isprime2.hs

```
isPrime :: Integer -> Bool
isPrime n =
    let isPrimeSub :: Integer -> Integer -> Bool
        isPrimeSub n m
          | m < 2        = True
          | mod n m == 0 = False
          | otherwise    = isPrimeSub n (m-1)
    in isPrimeSub n (n-1)
```

isPrimeSub を定義する関数定義は、isPrime を定義する関数定義の中の let 式の中に書かれています。ですから、isPrimeSub という識別子は、isPrime を定義する関数定義の中の let 式の中だけというスコープを持つことになります。

3.8　空白と改行と注釈

3.8.1　空白と改行

空白という文字（スペースキーを押したときに入力される文字）と改行という文字（エンターキーを押したときに入力される文字）は、Haskell のプログラムの意味に影響を与えません。たとえば、

```
square n = n*n
```

という等式は、

```
square     n      =      n   *   n
```

と書いたとしても同じ意味になりますし、

```
square n =
    n*n
```

と書いたとしても同じ意味になります。

空白と改行は Haskell のプログラムの意味に影響を与えない、ということには例外もあります。第 3.7.2 項で説明したように、ブロックの書き方のひとつは、それぞれの文を改行で区切って、それぞれの行の先頭に空白を書く、というものです。

また、一重引用符を空白で囲むことによって作られた文字リテラル、つまり’ ’という文字リテラルは、空白という文字のデータを生成します。

同じように、文字列リテラルの中に空白を挿入した場合は、その空白を含んだ文字列のデータが生成されます。たとえば、

```
"C o n g r a t u l a t i o n s !"
```

という文字列リテラルは、

```
C o n g r a t u l a t i o n s !
```

という文字列を生成します。

3.8.2　改行を含む式を GHCi に入力する方法

GHCi は、通常、改行が入力されると、そこで式の入力が終わったと判断します。しかし、途中に改行を含む式を GHCi に入力することは、不可能ではありません。

途中に改行を含む式を GHCi に入力したい場合は、式の入力に先立って、`:{` というコマンドを入力して、式の入力が終わったのち、`:}` というコマンドを入力します。

```
Prelude> :{
Prelude| if 5>8 then
Prelude|     "namako"
Prelude| else
Prelude|     "umiushi"
Prelude| :}
"umiushi"
```

3.8.3 注釈

プログラムを書いているとき、それを読む人間（プログラムを書いた人自身もその中に含まれます）に伝えたいことを、そのプログラムの一部分として書いておきたい、ということがしばしばあります。プログラムの中に書かれたそのような文字列は、「注釈」(comment) と呼ばれます。

注釈は、処理系が、「ここからここまでは注釈だ」ということを認識することができるように、注釈を書くための文法にしたがって書く必要があります。

Haskell では、プログラムの中の注釈を処理系に無視してもらう方法が二つあります。ひとつはマイナスマイナス (`--`) を使う方法です。シーンの中に `--` を書くと、その直後から最初の改行までが無視されます。たとえば、Haskell の処理系は、

```
-- I am a comment.
```

という記述を、注釈とみなして無視します。

注釈を無視してもらう方法の二つ目は、左中括弧マイナス (`{-`) とマイナス右中括弧 (`-}`) でそれを囲むという方法です。改行を含んでいる注釈、つまり 2 行以上の注釈も、その全体を `{-` と `-}` で囲むことによって、無視してもらうことができます。たとえば、Haskell の処理系は、

```
{- I am a comment
which contain line feed. -}
```

という記述を、注釈とみなして無視します。

3.8.4 コメントアウトとアンコメント

プログラムを作成したり修正したりしているとき、その一部分を一時的に無効にしたい、ということがしばしばあります。そのような場合、無効にしたい部分を削除してしまうと、それを復活させるのに手間がかかりますので、削除するのではなくて、注釈にすることによって無効にするという手段が、しばしば使われます。記述の一部分を注釈にすることによって、それを無効にすることを、その部分を「コメントアウトする」(comment out) と言います。逆に、コメントアウトされている部分を復活させることを、その部分を「アンコメントする」(uncomment) と言います。

第 4 章　リスト

4.1　リストの基礎

4.1.1 リストとは何か

Haskell には、データの列を扱う方法が二つあります。

ひとつは、第 2.5 節で紹介したタプルを使う方法で、もうひとつは、この章で紹介することになる、「リスト」(list) と呼ばれるものを使う方法です。

タプルと同様に、リストも、データを並べることによってできる列です。リストを構成している個々のデータは、そのリストの「要素」(element) と呼ばれます。そして、リストを構成している要素の個数は、そのリストの「長さ」(length) と呼ばれます。

タプルとリストとのあいだには、次のような相違点があります。

- タプルは、長さの相違が型に反映されます。それに対して、リストは、長さの相違が型に反映されません。ですから、任意の長さの列を処理する関数を定義したいときは、タプルを使うよりもリストを使うほうが簡単にできます。

- タプルは、任意の型のデータを並べることによって作ることができます。それに対して、リストは、同じ型のデータしか並べることができません。
- タプルは、長さが 0 のものと長さが 1 のものを作ることができません。それに対して、リストは、長さが 0 のものも長さが 1 のものも作ることができます。

4.1.2 リストを生成する方法

リストを生成する方法としては、次の二つのものがあります。

- 「リスト表記」(list notation) と呼ばれる式を使う方法。
- 「コンス」(cons) と呼ばれる中置演算（演算子は : ）を使う方法。

リスト表記については第 4.1.3 項で、コンスについては第 4.2.2 項で説明することにしたいと思います。

4.1.3 リスト表記

リスト表記は、評価すると同じ型のデータが得られる 2 個以上の式をコンマで区切って並べて、その全体を角括弧で囲むことによってできる、

[式, 式, ···]

という形の式です。リスト表記を評価すると、その中の個々の式が評価されて、それらの式の値を、式と同じ順序で並べたリストが生成されて、そのリストが、そのリスト表記の値になります。

たとえば、

```
[37, 81, 24, 59]
```

というリスト表記を評価すると、その値として、37、81、24、59 という 4 個の整数をこの順序で並べることによってできるリストが得られます。

```
Prelude> [37, 81, 24, 59]
[37,81,24,59]
```

1 個のリストは 1 個のデータですから、同じ型のリストから構成されるリストを作ることも可能です。たとえば、

```
[[2, 3], [0, 4, 2, 9], [5, 1, 7]]
```

という式を評価することによって、整数のリストから構成されるリストを生成することができます。

```
Prelude> [[2, 3], [0, 4, 2, 9], [5, 1, 7]]
[[2,3],[0,4,2,9],[5,1,7]]
```

タプル表記とは違って、1 個の式を角括弧で囲んだものもリスト表記ですし、0 個の式を角括弧で囲んだものもリスト表記です。そのようなリスト表記を評価することによって、長さが 1 のリストや長さが 0 のリストを生成することができます。

```
Prelude> [81]
[81]
Prelude> []
[]
```

長さが 0 のリスト、つまり 0 個の要素から構成されるリストは、「空リスト」(empty list) と呼ばれます。

4.1.4 リスト型

リストの型は、それを構成している要素の型から構成される複合的な型です。すべてのリストの型は、総称して「リスト型」(list type) と呼ばれます。

リストの型を記述する型式は、

[型式]

というように、要素の型を記述する型式を角括弧で囲むことによって記述します。たとえば、

```
[37, 81, 24, 59]
```

というリストの型は、[Int] という型式によって記述することができて、

```
[[2, 3], [0, 4, 2, 9], [5, 1, 7]]
```

というリストの型は、[[Int]] という型式によって記述することができて、

```
[('D', True), ('H', False), ('A', True)]
```

というリストの型は、[(Char, Bool)] という型式によって記述することができます。
任意の型の要素から構成される 1 個のリストを引数として受け取って、それと同じ型のリストを戻り値として返す関数の型は、

```
[a] -> [a]
```

という型式によって記述することができます。

4.1.5 文字列

第 1.3.7 項で説明したように、文字列の型は、[Char] という型式によって記述されます（String は、それの同義語です）。
[Char] という型式から分かるとおり、文字列というのは、文字を要素とするリストです。したがって、文字列は、リスト表記を使って記述することも可能です。たとえば、

```
['n', 'a', 'm', 'a', 'k', 'o']
```

というリスト表記は、namako という文字列を生成します。

```
Prelude> ['n', 'a', 'm', 'a', 'k', 'o']
"namako"
```

4.2 リストの構造

4.2.1 リストの頭部と尾部

空リスト以外のすべてのリストは、「頭部」(head) と「尾部」(tail) という二つの部分に分解することができます。
リストの頭部というのは、そのリストの先頭の要素のことです。そしてリストの尾部というのは、そのリストから先頭の要素を取り除くことによってできるリストのことです。たとえば、

```
[37, 81, 24, 59]
```

というリストの場合、頭部は 37 という整数で、尾部は、

```
[81, 24, 59]
```

というリストです。
長さが 1 のリストも、頭部と尾部に分解することができます。長さが 1 のリストの頭部は、そのリストの唯一の要素です。尾部というのは、先頭の要素を取り除くことによってできるリストですから、長さが 1 のリストの場合、尾部は空リストになります。たとえば、[68] というリストの場合、頭部は 68 で、尾部は空リストです。

4.2.2 コンス

第 4.1.2 項で説明したように、リストを生成する方法は二つあります。ひとつは第 4.1.3 項で説明したリスト表記を使う方法で、もうひとつは、「コンス」(cons) と呼ばれる中置演算（演算子は : ）を使う方法です。
コンスに与えられた : という名前は、「コンス演算子」(cons operator) と呼ばれます。
コンスは、引数として受け取った二つのデータを結合することによってリストを生成して、そのリストを戻り値として返します。コンスが実行する結合というのは、頭部と尾部からリストを生成するという動作です。コンスは、演算子の左に書かれた式の値を頭部、右に書かれた式の値を尾部とするリストを生成します。たとえば、

```
37:[81, 24, 59]
```

という演算子適用を評価すると、

```
[37, 81, 24, 59]
```

というリストが生成されます。

```
Prelude> 37:[81, 24, 59]
[37,81,24,59]
Prelude> 68:[]
[68]
```

コンス演算子の結合規則は、右結合です。したがって、

```
83:19:[]
```

という演算子適用は、

```
83:19:[]
```

という構造だと解釈されます。

```
Prelude> 83:19:[]
[83,19]
```

4.2.3　リストと再帰

空リストではないすべてのリストは、頭部と尾部に分解することができて、尾部はさらにリストになっています。つまり、全体と同じものが一部分として含まれているわけです。したがって、リストは、再帰的な構造を持っていると考えることができます。

そして、空リストというのは、リストが持っている再帰的な構造の基底だと考えることができます。

リストは、このように再帰的な構造を持っていますので、それを処理する関数は、再帰を使うことによって簡単に定義することができます。

4.3　リストを処理する組み込み関数

4.3.1　リストの長さ

第 4.2.2 項で、「コンス」という中値演算を紹介しましたが、Haskell の処理系には、これ以外にも、リストを処理する多数の関数が組み込まれています。

この節では、リストを処理する組み込み関数を紹介したいと思います。

まず最初は、リストの長さを求める関数です。

length という組み込み関数は、リストを受け取って、その長さを返します。

```
Prelude> length [37, 81, 24, 59]
4
Prelude> length "nishinakajimaminamikata"
23
```

4.3.2　同一要素のリストの生成

replicate という組み込み関数は、1 個目の引数として整数、2 個目の引数として任意のデータを受け取って、1 個目の引数で指定された個数だけ、2 個目の引数を並べることによってできるリストを返します。

```
Prelude> replicate 10 7
[7,7,7,7,7,7,7,7,7,7]
```

4.3.3　リストの連結

この項では、リストを連結する組み込み関数を紹介します。

++ という中置演算は、2 個のリストを受け取って、それらを連結することによってできたリストを返します。

```
Prelude> [37, 81, 24, 59] ++ [47, 96, 10, 73]
[37,81,24,59,47,96,10,73]
Prelude> "kitsune" ++ "udon"
"kitsuneudon"
```

concat という組み込み関数は、リストを要素とするリストを受け取って、そのすべての要素を連結することによってできるリストを返します。

```
Prelude> concat [[3, 6, 4], [2, 7], [], [5, 9, 1, 0]]
[3,6,4,2,7,5,9,1,0]
```

4.3.4 リストからの要素の取り出し

この項では、リストから要素を取り出す組み込み関数を紹介します。

!! という中置演算は、リストと整数を受け取って、その整数を番号とする要素を返します（番号は、先頭の要素を 0 番目と数えます）。

!! を適用する演算子適用は、リストを求める式を左側に、番号を求める式を右側に書きます。

```
Prelude> [7, 2, 8, 1, 5, 4, 6] !! 3
1
```

take という組み込み関数は、1 個目の引数として整数、2 個目の引数としてリストを受け取って、リストの先頭から、整数で指定された個数の要素を取り出して、それらの要素から構成されるリストを返します。

```
Prelude> take 3 [7, 2, 8, 1, 5, 4, 6]
[7,2,8]
```

取り出す要素の個数が、リストの要素の個数よりも多い場合は、リストの全体が戻り値になります。

```
Prelude> take 7 [37, 81, 24, 59]
[37,81,24,59]
```

head という組み込み関数は、リストを受け取って、先頭の要素を返します。つまり、リストの頭部を求めるわけです。

```
Prelude> head [37, 81, 24, 59]
37
```

last という組み込み関数は、リストを受け取って、その末尾の要素を返します。

```
Prelude> last [37, 81, 24, 59]
59
```

4.3.5 リストの要素の削除

この項では、リストから要素を削除する組み込み関数を紹介します。

drop という組み込み関数は、1 個目の引数として整数、2 個目の引数としてリストを受け取って、リストの先頭から、整数で指定された個数の要素を削除することによってできたリストを返します。

```
Prelude> drop 3 [7, 2, 8, 1, 5, 4, 6]
[1,5,4,6]
```

tail という組み込み関数は、リストを受け取って、先頭の要素を削除することによってできたリストを返します。つまり、リストの尾部を求めるわけです。

```
Prelude> tail [37, 81, 24, 59]
[81,24,59]
```

init という組み込み関数は、リストを受け取って、末尾の要素を削除することによってできたリストを返します。

```
Prelude> init [37, 81, 24, 59]
[37,81,24]
```

4.3.6 リストに関する条件の判定

この項では、リストに関する条件の判定をする組み込み関数を紹介します。

elem という組み込み関数は、2 個目の引数としてリスト、1 個目の引数として、そのリストの要素と同じ型のデータを受け取って、そのデータがリストの要素ならば真、そうでないならば偽を返します。

```
Prelude> elem 24 [37, 81, 24, 59]
True
Prelude> elem 48 [37, 81, 24, 59]
```

```
False
```

null という組み込み関数は、リストを受け取って、それが空リストならば真、そうでないならば偽を返します。

```
Prelude> null []
True
Prelude> null [5, 3, 2]
False
```

4.3.7　リストの比較演算

第 2.6.2 項で紹介した比較演算は、リストとリストとの比較にも使うことができます。

```
Prelude> [3, 2, 8, 6] == [3, 2, 8, 6]
True
Prelude> [3, 2, 8, 6] == [3, 2, 8, 7]
False
Prelude> [3, 2, 8, 6] /= [3, 2, 8, 6]
False
Prelude> [3, 2, 8, 6] /= [3, 2, 8, 7]
True
```

リストとリストとのあいだの大小関係は、それらを構成している要素を先頭から順番に比較していって、最初に発見された等しくない要素の大小関係によって決まります。

```
Prelude> [3, 2, 8, 6] > [3, 2, 7, 9]
True
```

4.3.8　無限リストの生成

この項では、無限の長さを持つリストを生成する組み込み関数を紹介します。

無限の長さを持つリストは、「無限リスト」(infinite list) と呼ばれます。

cycle という組み込み関数は、リストを受け取って、それを無限に連結することによってできるリストを返します。

```
Prelude> take 15 (cycle [1, 2, 3, 4])
[1,2,3,4,1,2,3,4,1,2,3,4,1,2,3]
```

repeat という組み込み関数は、任意のデータを受け取って、それを無限に並べることによってできるリストを返します。

```
Prelude> take 10 (repeat 7)
[7,7,7,7,7,7,7,7,7,7]
```

4.3.9　リストの分割

splitAt という組み込み関数は、1 個目の引数として整数、2 個目の引数としてリストを受け取って、そのリストを、リストの先頭から整数で指定された個数の要素を取り出して並べたリストと、残りの要素から構成されるリストに分割して、それらの二つのリストから構成されるタプルを返します。

```
Prelude> splitAt 3 [7, 2, 8, 1, 5, 4, 6]
([7,2,8],[1,5,4,6])
```

4.3.10　リストの綴じ合わせ

二つのリストのそれぞれから要素をひとつずつ取り出して、長さが 2 のタプルから構成されるリストを作ることを、それらのリストを「綴じ合わせる」(zip) と言います。

zip という組み込み関数は、二つのリストを受け取って、それらのリストを綴じ合わせたリストを返します。

```
Prelude> zip [6, 3, 1, 0] ["cat", "top", "bit", "nut"]
[(6,"cat"),(3,"top"),(1,"bit"),(0,"nut")]
```

zip に渡す二つのリストは、長さが違っていてもかまいません。長さが違う場合、長いほうの余った要素は無視されます。

```
Prelude> zip [8, 4, 2, 7, 5, 9] ["six", "map", "sun", "god"]
```

```
[(8,"six"),(4,"map"),(2,"sun"),(7,"god")]
```

unzip という組み込み関数は、綴じ合わされたリストを受け取って、それを綴じ合わせる前の二つのリストに戻して、それらのリストから構成されるタプルを返します。

```
Prelude> unzip [(6, "cat"), (3, "top"), (1, "bit"), (0, "nut")]
([6,3,1,0],["cat","top","bit","nut"])
```

4.3.11 リストの逆順化

reverse という組み込み関数は、リストを受け取って、要素の順序を逆順にしたリストを返します。

```
Prelude> reverse [7, 2, 8, 1, 5, 4, 6]
[6,4,5,1,8,2,7]
```

4.3.12 リストの加算

sum という組み込み関数は、数値のリストを受け取って、そのすべての要素を加算した結果を返します。

```
Prelude> sum [60000, 5000, 800, 30, 7]
65837
```

4.3.13 リストの乗算

product という組み込み関数は、数値のリストを受け取って、そのすべての要素を乗算した結果を返します。

```
Prelude> product [2, 3, 5, 7]
210
```

4.3.14 リストの論理演算

and という組み込み関数は、真偽値のリストを受け取って、そのリストのすべての要素が真ならば真を返して、ひとつでも偽が含まれているならば偽を返します。

```
Prelude> and [True, True, True, True, True]
True
Prelude> and [True, True, True, False, True]
False
```

or という組み込み関数は、真偽値のリストを受け取って、そのリストのすべての要素が偽ならば偽を返して、ひとつでも真が含まれているならば真を返します。

```
Prelude> or [False, False, False, False, False]
False
Prelude> or [False, False, False, True, False]
True
```

4.3.15 リストの最大値と最小値

maximum という組み込み関数は、リストを受け取って、その要素のうちでもっとも大きなものを返します。

```
Prelude> maximum [5, 3, 2, 1, 8, 6, 4]
8
```

minimum という組み込み関数は、リストを受け取って、その要素のうちでもっとも小さなものを返します。

```
Prelude> minimum [5, 3, 2, 1, 8, 6, 4]
1
```

4.4　リストを処理する関数の定義

4.4.1　リストを処理する関数の定義の基礎

第 4.2.3 項で説明したように、リストは、再帰的な構造を持っていますので、それを処理する関数は、再帰を使うことによって簡単に定義することができます。

リストを再帰的に処理する関数の定義は、多くの場合、次の二つのパターンを持つ等式から構成されます。

(1) リストの基底である空リストのみと一致する、 `[]` というパターン。

(2) 空リスト以外のリストと一致する、 `(x:xs)` というパターン。

`(x:xs)` というパターンは、空リスト以外の任意のリストと一致します。そして、 `x` という変数識別子は頭部に束縛されて、 `xs` という変数識別子は尾部に束縛されます。ちなみに、 `x` と `xs` というのは、絶対にそう書かないといけないというわけではなくて、変数識別子であれば何でもかまいません。

4.4.2　リストの要素を加算する関数の定義

リストを再帰的に処理する関数を定義する例として、まず、組み込み関数の `sum` と同じような動作をする、 `mysum` という関数を定義してみましょう。

`mysum` は、引数として整数のリストを受け取って、すべての要素を加算した結果を戻り値として返します。

リストというのは、頭部と尾部から構成されています。リストを構成しているすべての要素を加算するという処理は、次のように再帰的に記述することができます。

- そのリストが空リストならば、結果は 0 になる。
- そのリストが頭部と尾部から構成されているならば、結果は、尾部を構成しているすべての要素を加算した結果と、頭部とを加算した結果である。

ですから、 `mysum` の定義は、次のように書くことができます。

プログラムの例　`mysum.hs`

```
mysum :: [Int] -> Int
mysum [] = 0
mysum (x:xs) = x + mysum xs
```

実行例

```
*Main> mysum [60000, 5000, 800, 30, 7]
65837
```

組み込み関数の `sum` は、整数と浮動小数点数とが混ざったリストも処理することができますが、 `mysum` に処理することができるのは、整数のリストだけです。整数と浮動小数点数とが混ざったリストも処理することができるようにするためには、「型クラス」(type class) と呼ばれるものを使う必要があります。型クラスについては、第 6.2 節で説明することにしたいと思います。

4.4.3　リストから要素を取り出す関数の定義

次に、リストから要素を取り出す関数を定義してみましょう。

まず、組み込み関数の `last` と同じ動作をする、 `mylast` という関数を定義してみましょう。

`mylast` は、引数としてリストを受け取って、その末尾の要素を戻り値として返します。

リストの末尾の要素を取り出すという処理は、次のように再帰的に記述することができます。

- リストの長さが 1 ならば（ `[x]` というパターンと一致するならば)、結果は頭部である。
- リストの長さが 2 以上ならば、結果は、尾部の末尾の要素である。

`mylast` の定義は、次のように書くことができます。

プログラムの例　`mylast.hs`

```
mylast :: [a] -> a
mylast [x] = x
mylast (_:xs) = mylast xs
```

実行例

```
*Main> mylast [37, 81, 24, 59]
59
```

次に、中置演算の !! と動作をする、nth という関数を定義してみましょう。

nth は、1 個目の引数としてリスト、2 個目の引数として整数を受け取って、その整数を番号とする要素を返します（番号は、先頭の要素を 0 番目と数えます）。

リストの n 番目の要素を取り出すという処理は、次のように再帰的に記述することができます。

- n が 0 ならば、結果は頭部である。
- n が 1 以上ならば、結果は、尾部の $n-1$ 番目の要素である。

nth の定義は、次のように書くことができます。

プログラムの例 nth.hs

```
nth :: [a] -> Int -> a
nth (x:xs) n
  | n == 0 = x
  | n >= 1 = nth xs (n-1)
```

実行例

```
*Main> nth [7, 2, 8, 1, 5, 4, 6] 3
1
```

4.4.4 リストを連結する関数の定義

次に、リストを連結する関数を定義してみましょう。

まず、中置演算の ++ と同じ動作をする、append という関数を定義してみましょう。

append は、引数として 2 個のリストを受け取って、それらを連結した結果を戻り値として返します。

2 個のリストを連結するという処理は、次のように再帰的に記述することができます。

- 1 個目のリストが空リストならば、結果は 2 個目のリストである。
- リストの長さが 1 以上ならば、結果は、1 個目のリストの頭部と、その尾部と 2 個目のリストとを連結した結果とをコンスした結果である。

append の定義は、次のように書くことができます。

プログラムの例 append.hs

```
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x:xs) ys = x : append xs ys
```

実行例

```
*Main> append [37, 81, 24, 59] [47, 96, 10, 73]
[37,81,24,59,47,96,10,73]
```

次に、組み込み関数の concat と同じ動作をする、myconcat という関数を定義してみましょう。

myconcat は、引数としてリストを要素とするリストを受け取って、そのすべての要素を連結することによってできるリストを戻り値として返します。

リストを構成しているリストを連結するという処理は、次のように再帰的に記述することができます。

- そのリストが空リストならば、結果も空リストである。
- リストの長さが 1 以上ならば、結果は、リストの頭部と、尾部を構成しているすべての要素を連結した結果とを連結した結果である。

myconcat の定義は、次のように書くことができます。

プログラムの例 myconcat.hs

```
myconcat :: [[a]] -> [a]
myconcat [] = []
```

```
myconcat (x:xs) = x ++ myconcat xs
```

実行例

```
*Main> myconcat [[3, 6, 4], [2, 7], [], [5, 9, 1, 0]]
[3,6,4,2,7,5,9,1,0]
```

4.4.5 無限リストを生成する関数の定義

次に、無限リストを生成する関数を定義してみましょう。

まず、組み込み関数の cycle と同じ動作をする、mycycle という関数を定義してみましょう。

mycycle は、引数としてリストを受け取って、それを無限に連結することによってできるリストを戻り値として返します。

mycycle の定義は、次のように書くことができます。

プログラムの例 mycycle.hs

```
mycycle :: [a] -> [a]
mycycle [] = []
mycycle xs = xs ++ mycycle xs
```

実行例

```
*Main> take 15 (mycycle [1, 2, 3, 4])
[1,2,3,4,1,2,3,4,1,2,3,4,1,2,3]
```

次に、組み込み関数の repeat と同じ動作をする、myrepeat という関数を定義してみましょう。

myrepeat は、任意のデータを受け取って、それを無限に並べることによってできるリストを戻り値として返します。

myrepeat の定義は、次のように書くことができます。

プログラムの例 myrepeat.hs

```
myrepeat :: a -> [a]
myrepeat xs = xs:(myrepeat xs)
```

実行例

```
*Main> take 10 (myrepeat 7)
[7,7,7,7,7,7,7,7,7,7]
```

4.5 レンジ

4.5.1 レンジの基礎

リスト表記には、「レンジ」(range) と呼ばれる特殊な書き方があります。レンジは、要素の範囲を指定することによってリストを記述する、というリスト表記の書き方です。

レンジは、基本的には、

[式 1 .. 式 2]

と書きます。この形のレンジを評価すると、その値として、式 1 の値から式 2 の値までの範囲に含まれるすべてのデータを列挙することによってできるリストが得られます。たとえば、

```
[3..8]
```

というレンジを評価すると、その値として、

```
[3, 4, 5, 6, 7, 8]
```

というリストが得られます。

```
Prelude> [3..8]
[3,4,5,6,7,8]
Prelude> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
```

4.5.2 ステップ

レンジは、指定された範囲の中に含まれるすべての要素を列挙したリストだけではなくて、指定された範囲の中に含まれるデータを飛び飛びに列挙したリストを生成することもできます。

指定された範囲の中に含まれるデータを飛び飛びに列挙したいときは、

[式$_1$, 式$_2$..式$_3$]

という形のレンジを書きます。この形のレンジを評価すると、その値として、式$_1$の値から式$_3$の値までの範囲の中にあるデータを飛び飛びに列挙することによってできるリストが得られます。

この形のレンジで、式$_1$の値から式$_2$の値とがどれだけ離れているかということは、「ステップ」(step) と呼ばれます。この形のレンジによって生成されるリストは、式$_1$から出発して、ステップの距離だけ飛び飛びにデータを列挙したものです。たとえば、

```
[3, 8..40]
```

というレンジの場合、ステップは 5 ですから、

```
[3, 8, 13, 18, 23, 28, 33, 38]
```

というリストが得られることになります。

式$_1$の値と式$_2$の値が、データの順序とは逆の順序になっている場合は、データを逆順に列挙したリストが得られます。たとえば、

```
[38, 33..0]
```

というレンジを評価すると、

```
[38, 33, 28, 23, 18, 13, 8, 3]
```

というリストが得られます。

```
Prelude> [3, 8..40]
[3,8,13,18,23,28,33,38]
Prelude> [38, 33..0]
[38,33,28,23,18,13,8,3]
Prelude> ['z', 'y'..'a']
"zyxwvutsrqponmlkjihgfedcba"
```

4.5.3 レンジによる無限リストの生成

無限リストは、4.3.8 で紹介した組み込み関数を使って生成することができるわけですが、レンジを使って生成することも可能です。

レンジは、..の右側の式を省略して、

[式$_1$..]

と書いたり、

[式$_1$, 式$_2$..]

と書いたりすることもできます。この場合、どこまで要素を列挙すればリストの生成が終了するのかという、範囲の終端が指定されていないわけですから、結果として、無限リストが生成されることになります。

```
Prelude> take 20 ['A'..]
"ABCDEFGHIJKLMNOPQRST"
```

次のプログラムの中で定義されている multiple という関数は、引数として整数 m と整数 n を受け取って、m の倍数を n 個だけ並べることによってできるリストを戻り値として返します。

プログラムの例 multiple.hs

```
multiple :: Int -> Int -> [Int]
multiple m n = take n [m, m+m..]
```

実行例

```
*Main> multiple 7 18
[7,14,21,28,35,42,49,56,63,70,77,84,91,98,105,112,119,126]
```

4.6 リスト内包表記

4.6.1 リスト内包表記の基礎

リスト表記には、レンジのほかにも、もうひとつ、特殊な書き方があります。それは、「リスト内包表記」(list comprehension) と呼ばれる書き方です。リスト内包表記は、すでに存在するリストに対する処理を記述することによってリストを記述する書き方です。

4.6.2 リスト内包表記の基本的な書き方

リスト内包表記は、基本的には、

[式$_1$ | 変数識別子 <- 式$_2$]

と書きます。この中の式$_2$のところには、値としてリストが得られる式を書きます。

リスト内包表記は、次のような手順で評価されます。

(1) 式$_2$を評価する。
(2) 式$_2$の値として得られたリストを構成しているそれぞれの要素について、縦棒 (|) と <- のあいだに書かれた変数識別子をそれに束縛して、式$_1$を評価する、ということを繰り返す。
(3) 式$_1$を評価することによって得られた値から構成されるリストを生成して、それをリスト内包表記全体の値にする。

縦棒と <- のあいだに書いた変数識別子を、そのまま式$_1$として書くと、式$_2$の値がそのままリスト内包表記の値になります。

```
Prelude> [x | x <- [0..9]]
[0,1,2,3,4,5,6,7,8,9]
```

要素に対する処理を式$_1$として書くことによって、その処理の結果から構成されるリストを生成することができます。

```
Prelude> [mod x 3 | x <- [0..9]]
[0,1,2,0,1,2,0,1,2,0]
```

次のプログラムの中で定義されている powerList という関数は、引数として整数 m と整数 n を受け取って、m の 1 乗から n 乗までを並べることによってできるリストを戻り値として返します。

プログラムの例 powerlist.hs

```
powerList :: Integer -> Integer -> [Integer]
powerList m n = [m^x | x <- [1..n]]
```

実行例

```
*Main> powerList 5 10
[5,25,125,625,3125,15625,78125,390625,1953125,9765625]
```

4.6.3 述語

リスト内包表記の中には、「述語」(predicate) と呼ばれるものを書くことができます。述語というのは、リストの要素にするデータを取捨選択する条件となる式のことです。述語は、評価すると値として真偽値が得られる式である必要があります。

述語を含むリスト内包表記は、基本的には、

[式$_1$ | 変数識別子 <- 式$_2$, 述語]

と書きます。この形のリスト内包表記は、次のような手順で評価されます。

(1) 式$_2$を評価する。
(2) 式$_2$の値として得られたリストを構成しているそれぞれの要素について、縦棒と <- のあいだに書かれた変数識別子をそれに束縛して、述語を評価する、ということを繰り返す。
(3) 述語の値が真だった場合は、式$_1$も評価する。
(4) 式$_1$を評価することによって得られた値から構成されるリストを生成して、それをリスト内包表記全体の値にする。

たとえば、

```
[x | x <- [0..9], mod x 2 == 0]
```

というリスト内包表記は、0 から 9 までの整数のうちで、2 で除算したあまりが 0 のものだけから構成されるリストを生成する、という意味ですから、値として、

```
[0, 2, 4, 6, 8]
```

というリストが得られます。

```
Prelude> [x | x <- [0..9], mod x 2 == 0]
[0,2,4,6,8]
```

次のプログラムの中で定義されている divisor という関数は、引数としてプラスの整数を受け取って、そのすべての約数から構成されるリストを戻り値として返します。

プログラムの例 divisor.py

```
divisor :: Integer -> [Integer]
divisor n = [x | x <- [1..n], mod n x == 0]
```

実行例

```
*Main> divisor 96
[1,2,3,4,6,8,12,16,24,32,48,96]
```

述語は、1 個のリスト内包表記の中に、何個でも書くことができます。2 個以上の述語を書く場合は、それらの述語をコンマで区切って並べます。

リスト内包表記の中に 2 個以上の述語が書かれている場合の取捨選択では、それらの述語をすべて真にするデータのみが選択されます。たとえば、

```
[x | x <- [0..10], mod x 2 == 0, x >= 6]
```

というリスト内包表記を評価すると、その値として、0 から 10 までの整数のうちで、2 で除算したあまりが 0 で、かつ、6 以上のものから構成されるリスト、つまり、

```
[6, 8, 10]
```

というリストが得られます。

4.6.4 リストの組み合わせ

リスト内包表記は、2 個以上のリストを組み合わせたリストを生成することもできます。

リスト内包表記を使って 2 個以上のリストを組み合わせたいときは、

変数識別子 <- 式

という形のものを、2 個以上、コンマで区切って並べます（述語を書きたい場合はそれらの右側に書きます）。そうすると、それぞれの式の値として得られたリストを構成している要素を組み合わせた要素から構成されるリストが生成されます。たとえば、

```
[(x, y) | x <- [0, 1, 2], y <- [3, 4]]
```

というリスト内包表記を評価すると、その値として、

```
[(0, 3), (0, 4), (1, 3), (1, 4), (2, 3), (2, 4)]
```

というリストが得られます。

```
Prelude> [(x, y) | x <- [0, 1, 2], y <- [3, 4]]
[(0,3),(0,4),(1,3),(1,4),(2,3),(2,4)]
```

次のプログラムの中で定義されている cartesian という関数は、2 個のリストを引数として受け取って、それらのリストの直積（それぞれのリストを構成している要素を組み合わせたタプルから構成されるリスト）を戻り値として返します。

プログラムの例 cartesian.hs

```
cartesian :: [a] -> [b] -> [(a, b)]
cartesian xs ys = [(x, y) | x <- xs, y <- ys]
```

実行例

```
*Main> cartesian [0, 1, 2] ['a', 'b']
[(0,'a'),(0,'b'),(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

第 5 章　高階関数

5.1　高階関数の基礎

5.1.1　高階関数とは何か

Haskell では、関数というのもデータの一種です。したがって、値として関数が得られる式を書くことができます。関数は、引数として関数を受け取ることもできますし、戻り値として関数を返すことも可能です。

引数として関数を受け取る関数や、戻り値として関数を返す関数は、「高階関数」(higher-order function) と呼ばれます。

5.1.2　カリー化関数

Haskell では、高階関数というのは、それほど珍しいものではありません。このチュートリアルでも、高階関数はすでに何個も登場しています。実は、Haskell では、引数として 2 個以上のデータを受け取る関数というのは、すべて高階関数なのです。

厳密に言えば、引数として 2 個以上のデータを受け取る関数というのは、Haskell には存在しません。Haskell の関数は、1 個の引数しか受け取ることができないのです。では、このチュートリアルにこれまでに登場した、2 個以上の引数を受け取る関数というのは、いったい何だったのでしょうか。

2 個以上の引数を受け取る関数の正体は、1 個目の引数だけを受け取って、戻り値として関数を返す高階関数です。2 個目の引数を受け取るのは、高階関数が戻り値として返した関数です。

戻り値として関数を返すことによって、あたかも 2 個以上の引数を受け取るかのように定義された関数は、「カリー化関数」(curried function) と呼ばれます。2 個以上の引数を受け取ると言われる Haskell の関数は、すべてカリー化関数です。それらが「n 個の引数を受け取る」と説明されるのは、あくまで便宜的なもので、実際に受け取る引数は 1 個だけです。

`replicate` という Haskell の組み込み関数も、カリー化関数の一例です。この組み込み関数は、通常、「1 個目の引数として整数、2 個目の引数として任意のデータを受け取って、1 個目の引数で指定された個数だけ、2 個目の引数を並べることによってできるリストを戻り値として返す」というように説明されます。たとえば、

```
replicate 10 True
```

という式で `replicate` を引数に適用したとすると、`replicate` は、引数として 10 と `True` を受け取って、`True` を 10 個だけ並べることによってできる、

```
[True, True, True, True, True, True, True, True, True, True]
```

というリストを戻り値として返します。

厳密に言えば、`replicate` は、1 個の引数を受け取って関数を返す高階関数です。`replicate` が返す関数は、引数として 1 個の任意のデータを受け取って、それを特定の個数だけ並べることによってできるリストを戻り値として返します。たとえば、

```
replicate 10 True
```

という式で `replicate` を引数に適用した場合、`replicate` が受け取る引数は、10 だけです。`True` を受け取るのは、`replicate` が返した関数、つまり、1 個の任意のデータを受け取って、それを 10 個だけ並べることによってできるリストを戻り値として返す関数です。

5.1.3　高階関数の型を記述する型式

関数の型は、

引数の型 -> 戻り値の型

という型式によってあらわされます。「引数の型」というところには引数の型をあらわす型式を書いて、「戻り値の型」というところには戻り値の型をあらわす型式を書きます。ただし、引数

を受け取らない関数の型は、戻り値の型をあらわす型式と同じです。

-> は、型式を作るために使われる、「型演算子」(type operator) と呼ばれるものの一種です。

高階関数というのは、引数または戻り値が関数であるような関数のことですから、高階関数の型をあらわす型式には、2 個以上の -> が含まれることになります。たとえば、replicate という組み込み関数の型は、

```
Int -> a -> [a]
```

という型式であらわされます。この型式は、

```
Int -> a -> [a]
```

という構造を持っていると解釈されます。つまり、この型式は、引数の型が Int で、戻り値の型が a -> [a] であるような関数の型をあらわしている、と解釈されるわけです。

replicate の型をあらわす型式から分かるとおり、-> という型演算子は、右結合という結合規則を持っています。したがって、引数として関数を受け取る関数の型式を記述するためには、そのように解釈してもらえるように型式を記述することが必要になります。

型式の解釈を指定するための方法は、普通の式の場合と同じように、ひとつの型式だと解釈してほしい部分を丸括弧で囲む、というものです。たとえば、引数が Int で戻り値が Char の関数を引数として受け取って、戻り値として Bool を返す関数の型をあらわす型式は、

```
(Int -> Char) -> Bool
```

というように、引数の型をあらわす型式を丸括弧で囲むことによって記述することができます。

5.2 部分適用

5.2.1 部分適用の基礎

n 個の引数に適用することのできるカリー化関数は、実際に n 個の引数に適用することもできますが、適用する対象となる引数の個数は、n 個よりも少なくてもかまいません。n 個よりも少ない引数にその関数を適用する関数適用を評価すると、その値として、まだ適用されていない引数に適用することのできる関数が得られることになります。

n 個の引数に適用することのできるカリー化関数を、n 個よりも少ない引数に適用する、という関数の適用は、「部分適用」(partial application) と呼ばれます。

5.2.2 部分適用による関数の定義

カリー化関数を部分適用することによって、そのカリー化関数を特殊化した関数を定義することができます。

たとえば、replicate というカリー化関数を部分適用することによって、受け取った引数を特定の個数だけ並べることによってできるリストを戻り値として返す関数を定義することができます。

次のプログラムは、引数として 1 個の任意のデータを受け取って、それを 10 個だけ並べることによってできるリストを戻り値として返す、replicateTen という関数を、replicate を部分適用することによって定義しています。

プログラムの例 replicateten.hs

```
replicateTen :: a -> [a]
replicateTen = replicate 10
```

実行例

```
*Main> replicateTen True
[True,True,True,True,True,True,True,True,True,True]
```

5.2.3 セクション

Haskell では、中置演算もカリー化関数です。したがって、中置演算も、部分適用が可能です。

中置演算を部分適用したいときは、「セクション」(section) と呼ばれるものを書きます。セクションというのは、

(式 中置演算子)

または、

(中置演算子 式)

という形の記述のことです。つまり、中置演算子の片側だけに式を書いて、その全体を丸括弧で囲んだものがセクションです。

セクションを評価すると、その値として、中置演算を式の値に部分適用したときの戻り値、つまり、まだ適用されていないほうのデータを受け取る関数が得られます。

たとえば、

```
(['s', 'e', 'c', 't', 'i', 'o', 'n'] !!)
```

というセクションを評価すると、引数として整数を受け取って、その整数で指定されたリストの要素を戻り値として返す関数が得られます。同じように、

```
(!! 4)
```

というセクションを評価すると、引数としてリストを受け取って、そのリストの 4 番目の要素を戻り値として返す関数が得られます。

次のプログラムは、引数として整数を受け取って、その整数で指定された英字の小文字（0 ならばa、1 ならばb、2 ならばc、……）を戻り値として返す、alphabet という関数を、!! を部分適用することによって定義しています。

プログラムの例　alphabet.hs

```
alphabet :: Int -> Char
alphabet = (['a'..'z'] !!)
```

実行例

```
*Main> alphabet 0
'a'
*Main> alphabet 22
'w'
```

5.2.4　2 個目の引数に対する部分適用

1 個目の引数ではなく、2 個目の引数に関数を部分適用したい、というときはどうすればいいのでしょうか。

普通の関数ではなくて、中置演算ならば、セクションを書くことによって、左側の引数に対しても右側の引数に対しても部分適用が可能です。ということは、中置演算ではない普通の関数も、その名前を中置演算子にしてしまえば、2 個目の引数に部分適用することが可能になるはずです。

中置演算ではない関数の名前を中置演算子にする方法は、すでに第 2.4.9 項で説明しています。すなわち、関数名をバッククオート (‘) で囲めばいいわけです。

それでは、バッククオートを使うことによって、中置演算ではない関数を 2 個目の引数に部分適用するセクションを書いてみましょう。たとえば、

```
(`replicate` True)
```

というセクションを書くことによって、replicate という関数を、True という 2 個目の引数に部分適用することができます。このセクションの値は、引数として整数 n を受け取って、True を n 個だけ並べることによってできるリストを戻り値として返す関数です。

次のプログラムは、引数として整数 n を受け取って、True を n 個だけ並べることによってできるリストを戻り値として返す、replicateTrue という関数を、replicate を 2 個目の引数に部分適用することによって定義しています。

プログラムの例　replicatetrue.hs

```
replicateTrue :: Int -> [Bool]
replicateTrue = (`replicate` True)
```

実行例

```
*Main> replicateTrue 10
```

```
[True,True,True,True,True,True,True,True,True,True]
```

5.3 map と filter

5.3.1 map と filter の基礎

この節では、map と filter という二つの組み込み関数を紹介したいと思います。

map と filter は、どちらも、引数として関数を受け取る高階関数です。Haskell の組み込み関数の中には、そのような高階関数がいくつかあるのですが、それらはものすごく便利な関数ですので、それらの使い方に習熟しておくことは、とても大切です。

5.3.2 map

map は、「写像」(mapping) と呼ばれる処理を実行する関数です。

写像というのは、リストを構成しているすべての要素に対して同じ関数を適用して、得られた戻り値から構成されるリストを作成する、という処理のことです。

map は、2 個の引数を受け取ります。1 個目の引数は、個々の要素に適用する関数で、2 個目の引数は、処理の対象となるリストです。map は、引数として受け取ったリストに対して写像を実行して、その結果として作成されたリストを戻り値として返します。

例として、引数として真偽値を受け取って、それを反転させた結果を返す、not という組み込み関数を使って、map の動作を確かめてみましょう。たとえば、

```
map not [True, False, False, False, True, True]
```

という関数適用で、not という関数と、真偽値のリストに map を適用したとすると、map は、リストを構成しているそれぞれの真偽値に not を適用して、not の戻り値から構成される、

```
[False, True, True, True, False, False]
```

というリストを作成します。

```
Prelude> map not [True, False, False, False, True, True]
[False,True,True,True,False,False]
```

関数の部分適用によってできた関数を map に引数として渡すことも可能です。

```
Prelude> map (replicate 5) ['a'..'f']
["aaaaa","bbbbb","ccccc","ddddd","eeeee","fffff"]
Prelude> map (`mod` 4) [0..20]
[0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3,0]
```

次のプログラムは、引数として整数のリストを受け取って、それを構成しているそれぞれの整数が持っている約数の個数から構成されるリストを戻り値として返す、mapCountDivisor という関数を、map を使って定義しています。

プログラムの例　mapcountdivisor.hs

```
divisor :: Integer -> [Integer]
divisor n = [x | x <- [1..n], mod n x == 0]

countDivisor :: Integer -> Int
countDivisor n = length (divisor n)

mapCountDivisor :: [Integer] -> [Int]
mapCountDivisor = map countDivisor
```

実行例

```
*Main> mapCountDivisor [1..30]
[1,2,2,3,2,4,2,4,3,4,2,6,2,4,4,5,2,6,2,6,4,4,2,8,3,4,4,6,2,8]
```

5.3.3 filter

filter は、「フィルター」(filter) と呼ばれる処理を実行する関数です。

フィルターというのは、リストを構成しているすべての要素に対して、戻り値として真偽値を返す関数を適用して、戻り値として真が得られた要素だけから構成されるリストを作成する、と

いう処理のことです。

filter は、2 個の引数を受け取ります。1 個目の引数は、個々の要素に適用する、戻り値として真偽値を返す関数で、2 個目の引数は、処理の対象となるリストです。filter は、引数として受け取ったリストに対してフィルターを実行して、その結果として作成されたリストを戻り値として返します。

例として、引数としてリストを受け取って、それが空リストならば真、そうでなければ偽を返す、null という組み込み関数を使って、filter の動作を確かめてみましょう。たとえば、

```
filter null [[3, 5], [], [], [7, 8], [4, 2], []]
```

という関数適用で、null という関数と、リストのリストに filter を適用したとすると、filter は、リストを構成しているそれぞれのリストに null を適用して、null が真を返したリストだけから構成される、

```
[[], [], []]
```

というリストを作成します。

```
Prelude> filter null [[3, 5], [], [], [7, 8], [4, 2], []]
[[],[],[]]
```

関数の部分適用によってできた関数を filter に引数として渡すことも可能です。

```
Prelude> filter (>5) [3, 8, 7, 2, 0, 6, 4, 9]
[8,7,6,9]
Prelude> filter (elem 'a') ["cat", "sun", "ant", "toy", "day"]
["cat","ant","day"]
```

次のプログラムは、引数として整数のリストを受け取って、それを構成している整数のうちで、素数であるものだけから構成されるリストを戻り値として返す、filterPrime という関数を、filter を使って定義しています。

プログラムの例　filterprime.hs

```
divisor :: Integer -> [Integer]
divisor n = [x | x <- [1..n], mod n x == 0]

prime :: Integer -> Bool
prime n = length (divisor n) == 2

filterPrime :: [Integer] -> [Integer]
filterPrime = filter prime
```

実行例

```
*Main> filterPrime [2..50]
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

5.4　ラムダ式

5.4.1　ラムダ式の基礎

第 2.3.2 項で説明したように、関数を生成して、それに名前を与えることを、関数を「定義する」(define) と言います。

関数を定義したいときは、関数定義を書けばいいわけですが、場合によっては、関数を生成することは必要だけれども、それに名前を与えることは必要ではない、ということもあります。引数として関数を受け取る高階関数を呼び出すときに、それに対して引数として渡す関数を生成する場合には、その関数に名前を与える必要はありません。

Haskell では、「ラムダ式」(lambda expression) と呼ばれる式を書くことによって、名前を与えないで関数を生成することができます。ですから、引数として関数を受け取る高階関数に渡す関数を生成したいときは、関数定義を書いて関数を定義する代わりに、ラムダ式を書くだけで済ますこともできます。

ラムダ式は、式の一種です。ラムダ式を評価すると、関数が生成されて、その関数がラムダ式の値になります。

関数を部分適用したり、ラムダ式を評価したりすることによって得られる、名前を持たない関数は、「無名関数」(anonymous function) と呼ばれます。

5.4.2 ラムダ式の書き方

ラムダ式は、

\ パターン ・・・ -> 式

と書きます。バックスラッシュ(\)[1]と -> とのあいだには、引数と一致する 1 個以上のパターンを書きます。ラムダ式の場合、引数とパターンとが一致しなかった場合はエラーになりますので、注意が必要です。

パターンの書き方は、関数定義の場合と同じです。パターンの中に書かれた変数識別子は、それと一致したデータに束縛されます。ラムダ式を評価することによって得られる値は、引数とパターンとを一致させて、 -> の右側に書かれた式を評価する、という動作をする関数です。たとえば、

```
\n -> n*n
```

というラムダ式を評価すると、その値として、引数の 2 乗を求める関数が得られます。

```
Prelude> map (\n -> n*n) [5, 3, 2, 7, 8, 4, 6]
[25,9,4,49,64,16,36]
```

ラムダ式の値として得られた関数を、引数として高階関数に渡すのではなくて、そのままデータに適用する、ということも可能です。たとえば、

```
(\x y -> x * 10000 + y) 38 74
```

という式を評価すると、ラムダ式の値として得られた関数が、38 と 74 という整数に適用されます。

```
Prelude> (\x y -> x * 10000 + y) 38 74
380074
```

5.5 foldl と foldr

5.5.1 foldl と foldr の基礎

Haskell の処理系は、foldl と foldr という組み込み関数を持っています。これらの二つの関数は、どちらも、「畳み込み」(folding) と呼ばれる処理を実行する関数です。

畳み込みというのは、引数が 2 個の関数（中置演算も含む）をリストに対して再帰的に適用する、という処理のことです。

リストを左から右に向かって畳み込みをすることは、「左畳み込み」(left folding) と呼ばれます。それとは逆に、右から左に向かって畳み込みをすることは、「右畳み込み」(right folding) と呼ばれます。 foldl は左畳み込みを実行する関数で、 foldr は右畳み込みを実行する関数です。

5.5.2 左畳み込み

foldl という組み込み関数を使うことによって、左畳み込みを実行することができます。

foldl は、次の 3 個の引数を受け取ります。

1 個目　リストに対して再帰的に適用する、2 個の引数を受け取る関数。この関数が個々の要素を処理することになりますので、「要素処理関数」と呼ぶことにします。

2 個目　「アキュムレーター」(accumulator) と呼ばれるデータ。

3 個目　左畳み込みの対象となるリスト。

foldl が受け取る要素処理関数は、1 個目の引数と戻り値がアキュムレーターと同じ型である必要があります。そして、2 個目の引数は、左畳み込みの対象となるリストの要素と同じ型でないとけません。ですから、 foldl の型は、

```
(a -> b -> a) -> a -> [b] -> a
```

[1]日本語の環境では、バックスラッシュが円マーク (¥) として表示される場合もあります。

という型式で記述されることになります。

foldl は、まず、対象となるリストを頭部と尾部に分解して、頭部とアキュムレーターに要素処理関数を適用します。そして、関数、その戻り値、尾部に対して、自身を再帰的に適用します。対象となるリストが空リストだった場合は、アキュムレーターをそのまま戻り値として返します。たとえば、

```
foldl (+) 0 [3, 8, 5, 7]
```

という関数適用で、foldl を関数とアキュムレーターとリストに適用したとしましょう。そうすると、

```
foldl (+) (3+0) [8, 5, 7]
foldl (+) (3+8) [5, 7]
foldl (+) (11+5) [7]
foldl (+) (16+7) []
```

というように、リストが空リストになるまで、foldl が再帰的に適用されていきます。そして、対象が空リストになった段階で、アキュムレーターがそのまま戻り値になって、再帰から戻っていくことになります。再帰から戻っていく過程では、再帰的に適用した foldl の戻り値が、そのまま戻り値になります。ですから、foldl は、戻り値として 23 を返すことになります。

```
Prelude> foldl (+) 0 [3, 8, 5, 7]
23
```

次のプログラムは、引数として真偽値のリストを受け取って、その中に要素として含まれている True の個数を戻り値として返す、countTrueLeft という関数を、foldl を使って定義しています。

プログラムの例　counttrueleft.hs

```
countTrueLeft :: [Bool] -> Int
countTrueLeft = foldl (\n x -> if x then n+1 else n) 0
```

実行例

```
*Main> countTrueLeft [True, True, False, True, False, True]
4
```

次のプログラムの中で定義されている myfoldl は、foldl と同じ動作をします。

プログラムの例　myfoldl.hs

```
myfoldl :: (a -> b -> a) -> a -> [b] -> a
myfoldl f z []     = z
myfoldl f z (x:xs) = myfoldl f (f z x) xs
```

実行例

```
*Main> myfoldl (+) 0 [3, 8, 5, 7]
23
```

5.5.3　右畳み込み

foldr という組み込み関数を使うことによって、右畳み込みを実行することができます。

foldl と同じように、foldr も、3 個の引数を受け取ります。すなわち、要素処理関数、アキュムレーター、そして右畳み込みの対象となるリストです。

foldr が受け取る要素処理関数は、2 個目の引数と戻り値がアキュムレーターと同じ型である必要があります。そして、1 個目の引数は、右畳み込みの対象となるリストの要素と同じ型でないとけません。ですから、foldr の型は、

```
(a -> b -> b) -> b -> [a] -> b
```

という型式で記述されることになります。

foldr は、まず、対象となるリストを頭部と尾部に分解して、関数、アキュムレーター、尾部に対して、自身を再帰的に適用します。そして、対象となるリストが空リストだった場合は、アキュムレーターをそのまま戻り値として返します。たとえば、

```
foldr (+) 0 [3, 8, 5, 7]
```

という関数適用で、foldr を関数とアキュムレーターとリストに適用したとしましょう。そうすると、

```
foldr (+) 0 [8, 5, 7]
foldr (+) 0 [5, 7]
foldr (+) 0 [7]
foldr (+) 0 []
```

というように、リストが空リストになるまで、foldr が再帰的に適用されていきます。そして、対象が空リストになった段階で、アキュムレーターがそのまま戻り値になって、再帰から戻っていくことになります。再帰から戻っていく過程では、

```
7 + 0
5 + 7
8 + 12
3 + 20
```

というように、頭部と、再帰的に適用した foldr の戻り値に対して要素処理関数が適用されて、その戻り値が foldr の戻り値になります。ですから、foldr は、戻り値として 23 を返すことになります。

```
Prelude> foldr (+) 0 [3, 8, 5, 7]
23
```

次のプログラムは、引数として真偽値のリストを受け取って、その中に要素として含まれている True の個数を戻り値として返す、countTrueRight という関数を、foldr を使って定義しています。

プログラムの例 counttrueright.hs

```
countTrueRight :: [Bool] -> Int
countTrueRight = foldr (\x n -> if x then n+1 else n) 0
```

実行例

```
*Main> countTrueRight [True, True, False, True, False, True]
4
```

次のプログラムの中で定義されている myfoldr は、foldr と同じ動作をします。

プログラムの例 myfoldr.hs

```
myfoldr :: (a -> b -> b) -> b -> [a] -> b
myfoldr f z []     = z
myfoldr f z (x:xs) = f x (myfoldr f z xs)
```

実行例

```
*Main> myfoldr (+) 0 [3, 8, 5, 7]
23
```

5.5.4 foldl と foldr の使い分け

ところで、foldl と foldr という二つの組み込み関数は、どのように使い分けをすればいいのでしょうか。

まず、foldl と foldr のどちらか一方しか使えない場合がありますので、それについて説明しておきましょう。

foldl と foldr は、それぞれ、次のような型を持っています。

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldr :: (a -> b -> b) -> b -> [a] -> b
```

このままでは少し分かりにくいので、a と b という型変数を、foldr のほうだけ入れ替えてみ

ます。

```
foldl ::  (a -> b -> a) -> a -> [b] -> a
foldr ::  (b -> a -> a) -> a -> [b] -> a
```

型式を見比べてみると分かるように、foldl のほうは、要素処理関数の 2 個目の引数が、畳み込みの対象となるリストの要素と同じでないといけないのに対して、foldr のほうは、要素処理関数の 1 個目の引数が、畳み込みの対象となるリストの要素と同じでないといけません。ですから、要素処理関数が、異なる型を持つ 2 個の引数を受け取る場合は、リストの要素と 2 個目の引数の型が同じではないならば foldl は使うことができず、リストの要素と 1 個目の引数の型が同じではないならば foldr は使うことができない、ということになります。

それでは、同じ型を持つ 2 個の引数を受け取る要素処理関数を使って畳み込みを実行する場合は、foldl と foldr のどちらを使えばいいのでしょうか。

この場合に判断のポイントとなるのは、畳み込みの結果がリストになるかどうかということです。結果がリストではない畳み込みは、どちらを使っても効率は同じですが、結果がリストになるような畳み込み、つまり、その過程でリストが成長していくような畳み込みは、foldl よりも foldr のほうが効率よく実行されます。

結果がリストになるような畳み込みは、foldl を使った場合はリストが左から右へ向かって成長するのに対して、foldr を使った場合はリストが右から左へ向かって成長します。

リストは、右から左へ向かって成長させるほうが、左から右へ向かって成長させるよりも、高い効率で実行されます。なぜなら、リストというものは、右へ行くほど再帰が深くなるという構造を持っているからです。結果がリストになるような畳み込みは、foldl よりも foldr のほうが効率よく実行されるというのは、このような理由によるものです。

5.5.5　無限リストに対する畳み込み

&& という演算は、左側の式の値が偽だった場合は、その段階で結果が偽だと分かりますので、右側の式を評価しません。同じように、|| という演算も、左側の式の値が真だった場合は、その段階で結果が真だと分かりますので、やはり右側の式を評価しません。

foldr は、このような関数が要素処理関数として与えられた場合、結果が確定した段階で再帰をストップしますので、無限リストに適用することも可能です。たとえば、

```
foldr (&&) True (repeat False)
```

という式で、無限個の False から構成されるリストに対して、foldr を使って畳み込みを実行すると、False という値が得られます。

```
Prelude> foldr (&&) True (repeat False)
False
```

なお、foldr とは違って、foldl を無限リストに適用することはできません。

5.5.6　畳み込みによる組み込み関数の定義

畳み込みを使って関数を定義する練習として、リストを処理する組み込み関数と同じ動作をする関数を、畳み込みを使って定義してみましょう。

次のプログラムは、length と同じ動作をする mylength という関数を定義しています。

プログラムの例　mylength.hs

```
mylength :: [a] -> Int
mylength = foldl (\x _ -> x+1) 0
```

実行例

```
*Main> mylength [37, 81, 24, 59]
4
```

次のプログラムは、unzip と同じ動作をする myunzip という関数を定義しています。

プログラムの例　myunzip.hs

```
myunzip :: [(a, b)] -> ([a], [b])
myunzip = foldr
    (\(x, y) (xs, ys) -> (x:xs, y:ys))
    ([], [])
```

実行例

```
*Main> myunzip [(6, "cat"), (3, "top"), (1, "bit"), (0, "nut")]
([6,3,1,0],["cat","top","bit","nut"])
```

次のプログラムは、reverseと同じ動作をするmyreverseという関数を定義しています。

プログラムの例 myreverse.hs

```
myreverse :: [a] -> [a]
myreverse = foldl (\xs x -> x:xs) []
```

実行例

```
*Main> myreverse [7, 2, 8, 1, 5, 4, 6]
[6,4,5,1,8,2,7]
```

第5.5.4項で、結果がリストになるような畳み込みはfoldlよりもfoldrのほうが効率よく実行されると書きましたが、このmyreverseの場合は例外です。

次のプログラムは、mapと同じ動作をするmymapという関数を定義しています。

プログラムの例 mymap.hs

```
mymap :: (a -> b) -> [a] -> [b]
mymap f = foldr (\x xs -> (f x) : xs) []
```

実行例

```
*Main> mymap not [True, False, False, False, True, True]
[False,True,True,True,False,False]
```

次のプログラムは、filterと同じ動作をするmyfilterという関数を定義しています。

プログラムの例 myfilter.hs

```
myfilter :: (a -> Bool) -> [a] -> [a]
myfilter f = foldr (\x xs -> if f x then x:xs else xs) []
```

実行例

```
*Main> myfilter null [[3, 5], [], [], [7, 8], [4, 2], []]
[[],[],[]]
```

5.6 関数を合成する演算と関数を適用する演算

5.6.1 関数を合成する演算

関数fとgがあるとします。このとき、データにgを適用して、その戻り値にfを適用する、という動作をする関数を生成することを、fとgを「合成する」(compose)と言います。

Haskellの処理系には、関数を合成する、「関数合成演算」(function composition operation)と呼ばれる中置演算が組み込まれていて、その演算子は、.（ドット）です。この中置演算は、

f . g

という式でfとgに適用されたとすると、データにgを適用して、その戻り値にfを適用する、という動作をする関数を、戻り値として返します。たとえば、

```
(+1) . (*100)
```

という式を評価することによって、数値を100倍してからそれに1を加算する、という動作をする関数が得られます。

```
Prelude> ((+1) . (*100)) 37
3701
```

次のプログラムは、.と同じ動作をする、mycomposeという関数を定義しています。

プログラムの例 mycompose.hs

```
mycompose :: (b -> c) -> (a -> b) -> (a -> c)
```

```
mycompose f g x = f (g x)
```

実行例

```
*Main> mycompose (+1) (*100) 37
3701
```

5.6.2　関数を適用する演算

Haskell の処理系には、関数をデータに適用するという動作をする、「関数適用演算」(function application operation) と呼ばれる中置演算が組み込まれていて、その演算子は、$（ドルマーク）です。この中置演算は、

f $ x

という式で f と x に適用されたとすると、f を x に適用して、f が戻り値として返したデータを戻り値として返します。たとえば、

```
not $ True
```

という式を評価すると、not という関数が True に適用されますので、式の値は False になります。

```
Prelude> not $ True
False
```

$ は、関数から構成されるリストを処理したい、というときに使うことができます。たとえば、

```
[(+3), (*1000), (^2), (2^), (`div` 2)]
```

というリストを構成しているそれぞれの関数を 7 に適用した結果から構成されるリストを求めたい、というときは、

```
map ($ 7) [(+3), (*1000), (^2), (2^), (`div` 2)]
```

というように $ を使えばいいわけです。

```
Prelude> map ($ 7) [(+3), (*1000), (^2), (2^), (`div` 2)]
[10,7000,49,128,3]
```

$ という演算子は、右結合で、優先順位はきわめて低く設定されています。ですから、この演算子は、式の丸括弧をできるだけ減らしたい、というときにも使うことができます。たとえば、

```
map (*1000) (replicate 7 (div 100 20))
```

という式は、$ を使うことによって、

```
map (*1000) $ replicate 7 $ div 100 20
```

というように、丸括弧を減らすことができます。

```
Prelude> map (*1000) (replicate 7 (div 100 20))
[5000,5000,5000,5000,5000,5000,5000]
Prelude> map (*1000) $ replicate 7 $ div 100 20
[5000,5000,5000,5000,5000,5000,5000]
```

次のプログラムは、$ と同じ動作をする、myapply という関数を定義しています。

プログラムの例　myapply.hs

```
myapply :: (a -> b) -> a -> b
myapply f x = f x
```

実行例

```
*Main> myapply not True
False
*Main> map (`myapply` 7) [(+3), (*1000), (^2), (2^), (`div` 2)]
[10,7000,49,128,3]
```

第 6 章　型

6.1 型についての予備知識

6.1.1 型についての復習

この章では、「型」と呼ばれるものについて説明することになるわけですが、型については、この章よりも前のところで、必要に応じて断片的に説明してきていますので、まずは、型についてこれまでに説明してきたことを復習しておくことにしましょう。

- 「型」(type) というのは、同じ性質を持つデータの集合のことである。
- Haskell のプログラムにおいては、データはかならず、何らかの型に所属しているものとして扱われる。
- データ D が型 T に所属しているとき、「D は T を持つ」という言い方をすることもある。
- 基本的な一群の型は「基本型」(base type) と呼ばれる。
- すべての基本型は、「型名」(type name) と呼ばれる、英字の大文字で始まる名前を持つ。
- 基本型には、`Bool`、`Int`、`Integer`、`Float`、`Double`、`Char` がある。
- タプル、リスト、関数は、基本型を組み合わせた複合的な型を持つ。
- 任意の型は、「型式」(type expression) と呼ばれるものを書くことによって記述することができる。
- 基本型を記述する型式は、それの型名である。
- 複合的な型を記述する型式は、基本型の型名を組み合わせたものである。
- タプルの型を記述する型式は、要素の型を記述する型式をコンマで区切って並べて、その全体を丸括弧で囲んだものである。
- リストの型を記述する型式は、要素の型を記述する型式を角括弧で囲んだものである。
- 関数の型を記述する型式は、`->` という型演算子の左側に要素の型を記述する型式、右側に戻り値の型を記述する型式を書いたものである。
- `->` という型演算子の結合規則は右結合である。
- 型式の中に書かれた変数識別子は、「型変数」(type variable) と呼ばれる。
- 型変数は、「任意の型」という意味を持つ。ただし、ひとつの型式の中では、同じ型変数は同じ型を意味する。

6.1.2 型を調べる GHCi のコマンド

GHCi は、データの型を調べるコマンドを持っています。それは、`:type` または `:t` というコマンドです。

GHCi に対して、`:type` または `:t` と入力して、その右側に空白と式を入力して、そののちエンターキーを押すと、入力された式、`::`、そして入力された式の値が持っている型を記述する型式が表示されます。

```
Prelude> :t True
True :: Bool
Prelude> :t 'A'
'A' :: Char
Prelude> :t (False, 'B')
(False, 'B') :: (Bool, Char)
Prelude> :t [False, True, False, False]
[False, True, False, False] :: [Bool]
Prelude> :t head
head :: [a] -> a
```

`:type` または `:t` というコマンドの右側に、整数リテラルや浮動小数点数リテラルを入力すると、どのような型式が表示されるのか、ということも試してみましょう。

```
Prelude> :t 48
48 :: Num a => a
Prelude> :t 2.56
2.56 :: Fractional a => a
```

そうすると、このように、なんだかよく分からない型式が表示されます。これらの型式に含まれている、`Num` とか `Fractional` というのは、「型クラス」(type class) と呼ばれるものです。型クラスについては、第 6.2 節で説明することにしたいと思います。

6.2　型クラス

6.2.1　型クラスの基礎

第 3.4.2 項で説明したように、型変数を使わなければ記述することができない型を持つ関数は、「多相的関数」(polymorphic function) と呼ばれます。多相的関数は、さまざまな型のデータを処理することができます。しかし、必ずしも、どんな型のデータでも処理することのできる関数であるとは限りません。

たとえば、特定のデータがリストの要素になっているかどうかを調べる、elem という組み込み関数は、多相型関数のひとつです。

```
Prelude> elem 24 [37, 81, 24, 59]
True
Prelude> elem 'p' "function"
False
Prelude> elem True [False, False, True, False]
True
```

このように、elem は、さまざまな型のリストについて、その中に特定のデータが含まれているかどうかを調べることができます。しかし、どんな型のリストでも大丈夫というわけではありません。たとえば、

```
elem (2^) [(+3), (*1000), (^2), (2^), (`div` 2)]
```

というように、関数から構成されるリストの中に、特定の関数が含まれているかどうかを調べることはできません。その理由は、関数というデータは、ほかの関数と等しいかどうかを比較することができないからです。

このような、多くの型のデータを処理することができるけれども、処理することのできる型には何らかの制約がある、という関数の型は、「型クラス」(type class) と呼ばれるものによる制約、という形で記述されます。

型クラスというのは、適用することのできるいくつかの関数を共有する型の集合のことです。

たとえば、elem という関数の型は、Eq という型クラスによって制約を受けています。Eq という型クラスは、等しいかどうかを比較する == と /= という二つの関数を共有する型の集合です。つまり、elem を適用することのできるデータは、等しいかどうかを比較することができるものでないといけない

型クラスを構成しているそれぞれの型は、その型クラスの「インスタンス」(instance) と呼ばれます。そして、型クラスによって共有される関数は、その型クラスの「メソッド」(method) と呼ばれます。たとえば、Bool、Int、Float、Char、(Char, Int)、[Bool] などは、Eq のインスタンスです。そして、== と /= は、Eq のメソッドです。

6.2.2　文脈

型クラスによる制約を受ける関数の型を記述するためには、型式の左側に、「文脈」(context) と呼ばれるものを書く必要があります。

文脈は、

型クラス制約 =>

と書きます。この中の「型クラス制約」というところには、型変数に対して型クラスによる制約を与える、「型クラス制約」(type class constraint) と呼ばれる記述を書きます。

型クラス制約は、

型クラス名 型変数

と書きます。そうすると、「型変数」のところに書かれた型変数が、「型クラス名」のところに書かれた名前の型クラスによって制約されることになります。たとえば、

```
Eq a
```

という型クラス制約を書くことによって、型変数の a は、Eq という型クラスによる制約を受けることになります。

ですから、elem という関数の型は、

```
Eq a => a -> [a] -> Bool
```

という型式によって記述されることになります。

ところで、2 個以上の型クラスによって制約される型は、どのような型式で記述されるのでしょうか。たとえば、fromIntegral という組み込み関数の型は、2 個の型クラスによって制約されています。この関数の型がどのような型式によって記述されているか、GHCi を使って調べてみましょう。

```
Prelude> :t fromIntegral
fromIntegral :: (Integral a, Num b) => a -> b
```

このように、2 個以上の型クラスによって制約される型を記述する型式を書きたい場合は、型クラス制約をコンマで区切って並べて、その全体を丸括弧で囲んだものを文脈の中に書けばいいわけです。

6.2.3 ユーザー定義型クラス

型クラスを作成して、それに名前を与えることを、型クラスを「定義する」(define) と言います。

型クラスに与えられた名前は、「型クラス名」(type class name) と呼ばれます。

Haskell では、「型クラス定義」(type class definition) と呼ばれる記述をプログラムの中に書くことによって、型クラスを自由に定義することができます（型クラス定義の書き方については、第 6.7 節で説明することにしたいと思います)。プログラムの中に型クラス定義を書くことによって定義された型クラスは、「ユーザー定義型クラス」(user-defined type class) と呼ばれます。

6.2.4 組み込み型クラス

「ユーザー定義型クラス」の対義語は、「組み込み型クラス」です。「組み込み型クラス」(built-in type class) というのは、Haskell の処理系に組み込まれている型クラスのことです。組み込み型クラスは、型クラス定義を書かなくても利用することができます。第 6.2.1 項で紹介した Eq は、組み込み型クラスのひとつです。

組み込み型クラスとしては、たとえば次のようなものがあります。

```
Eq    Enum      Num        Floating    Read
Ord   Bounded   Integral   Show
```

6.2.5 Eq

第 6.2.1 項で説明したように、Eq は、等しいかどうかを比較する == と /= という二つの関数をメソッドとする型クラスです。

基本型は、すべて Eq のインスタンスです。それに対して、関数の型は、Eq のインスタンスではありません。タプルの型は、すべての要素の型が Eq のインスタンスならば、その全体も Eq のインスタンスです。同じように、リストの型も、要素の型が Eq のインスタンスならば、その全体も Eq のインスタンスです。

次のプログラムは、elem と同じ動作をする myelem という関数を定義しています。

プログラムの例 myelem.hs

```
myelem :: Eq a => a -> [a] -> Bool
myelem e []  = False
myelem e (x:xs)
  | e == x    = True
  | otherwise = myelem e xs
```

実行例

```
*Main> myelem 24 [37, 81, 24, 59]
True
*Main> myelem 48 [37, 81, 24, 59]
False
```

6.2.6 Ord

Ord は、大小関係の比較をすることのできる型をインスタンスとする型クラスです。

Ord のメソッドとしては、>、<、>=、<= などがあります。

Ord のインスタンスは、Eq と同じです。すべての基本型は Ord のインスタンスで、Ord のインスタンスだけから構成されるタプルまたはリストの型も Ord のインスタンスです。

次のプログラムの中で定義されている countGreater という関数は、2 個目の引数として与えられたリストを構成している要素のうちで、1 個目の引数よりも大きいものの個数を戻り値として返します。

プログラムの例 countgreater.hs

```
countGreater :: Ord a => a -> [a] -> Int
countGreater e = foldl (\n x -> if x > e then n+1 else n) 0
```

実行例

```
*Main> countGreater 5 [3, 8, 0, 7, 6, 2, 5, 1]
3
```

6.2.7 Enum

Enum は、要素を列挙することができる型をインスタンスとする型クラスです。

レンジを使って、データを列挙したリストを生成するためには、列挙するデータの型が Enum のインスタンスである必要があります。

Enum のインスタンスは、基本型だけです。タプルやリストの型は Enum のインスタンスではありません。

Enum のインスタンスのデータは、「後者関数」(successor function) と「前者関数」(predecessor function) と呼ばれる関数を使うことによって、次のデータや前のデータを求めることができます。

後者関数は、 succ という名前の組み込み関数で、 Enum のインスタンスのデータに適用すると、そのデータの次のデータを戻り値として返します。同じように、前者関数は、 pred という名前の組み込み関数で、 Enum のインスタンスのデータに適用すると、そのデータの前のデータを戻り値として返します。

```
Prelude> succ 'C'
'D'
Prelude> pred 'C'
'B'
```

また、 Enum のインスタンスのデータは、それが列挙される順番を示す、0 から始まる番号を持っています。その番号は、 fromEnum という組み込み関数を使うことによって調べることができます。 fromEnum は、 Enum のインスタンスのデータを引数として受け取って、そのデータが持っている番号を戻り値として返します。

```
Prelude> fromEnum 'C'
67
```

ちなみに、文字のデータが持っている番号は、その文字の文字コードと等しくなります。

Enum のインスタンスのデータは、 toEnum という組み込み関数を使うことによって、特定の番号を持つものがどのデータなのかということを調べることもできます。

toEnum を使うためには、「型注釈」(type annotation) と呼ばれるものを式の右側に書くことによって、 Enum のインスタンスのうちのどの型のデータを求めるのか、ということを指定する必要があります。

型注釈は、

:: 型式

と書きます。この中の「型式」のところには、式の値の型として指定したい型を記述する型式を書きます。たとえば、

```
toEnum 67 :: Char
```

というように、 toEnum を番号に適用する式の右側に、 :: Char という型注釈を書いたとすると、 toEnum は、 Char のデータを求めることになります。

```
Prelude> toEnum 67 :: Char
'C'
```

6.2.8 Bounded

Bounded は、下限と上限を持つ型をインスタンスとする型クラスです。

Int、Char、Bool は、Bounded のインスタンスです。また、Bounded のインスタンスのデータを要素とするタプルの型も、Bounded のインスタンスです。

Bounded のインスタンスは、minBound と maxBound という組み込み関数を使うことによって、その下限と上限を求めることができます。minBound は下限を求める関数で、maxBound は上限を求める関数です。

minBound と maxBound は、どちらも、引数を受け取らない関数です。これらの関数は、型注釈で指定された型の下限または上限を戻り値として返します。

```
Prelude> minBound :: Int
-2147483648
Prelude> maxBound :: Char
'\1114111'
Prelude> maxBound :: (Int, Char, Bool)
(2147483647,'\1114111',True)
```

6.2.9 Num

Num は、数値の型をインスタンスとする型クラスです。

Int、Integer、Float、Double は、Num のインスタンスです。

第 6.1.2 項で確認したように、48 という整数リテラルの値の型は、

```
Num a => a
```

という型式であらわされるわけですが、この型式は、「Num のインスタンスであるならば、どの型のデータにもなることができる」ということを意味しています。

第 4.4.2 項で、組み込み関数の sum と同じような動作をする mysum という関数の定義を紹介しましたが、そのときは、型クラスを使わずに定義を書きましたので、mysum が合計を求めることのできるリストは、要素が Int であるものに限られていました。Num を使えば、組み込み関数の sum と同じように、整数のリストにも浮動小数点数のリストにも適用することができるように、mysum を改良することができます。

プログラムの例 mysum2.hs

```
mysum :: Num a => [a] -> a
mysum [] = 0
mysum (x:xs) = x + mysum xs
```

実行例

```
*Main> mysum [8, 3, 2, 1, 6, 4, 5]
29
*Main> mysum [4.03, 2.59, 8.16, 0.47]
15.25
```

6.2.10 Integral

Integral は、整数の型をインスタンスとする型クラスです。

Int と Integer は、Integral のインスタンスです。

第 6.2.2 項で、2 個以上の型クラスによって制約される型を記述する型式について説明したときに登場した、fromIntegral という組み込み関数は、整数を浮動小数点数に変換したいときに使われます。

たとえば、length の戻り値は、型が Int ですので、そのままでは、それと浮動小数点数とを加算するという計算ができません。それをするためには、fromIntegral を使って、length の戻り値を浮動小数点数に変換する必要があります。

```
Prelude> fromIntegral (length [3, 1, 7, 6, 9, 0, 2]) + 0.28
7.28
```

6.2.11 Floating

Floating は、浮動小数点数の型をインスタンスとする型クラスです。

Float と Double は、Floating のインスタンスです。

6.2.12 Show

Show は、文字列に変換することのできる型をインスタンスとする型クラスです。

Show のインスタンスは、Eq と同じです。すべての基本型は Show のインスタンスで、Show のインスタンスだけから構成されるタプルまたはリストも Show のインスタンスです。

Show のインスタンスのデータは、show という組み込み関数を使うことによって、文字列に変換することができます。show は、Show のインスタンスのデータに適用すると、そのデータを文字列に変換した結果を戻り値として返します。

```
Prelude> show 357
"357"
Prelude> show 4.286
"4.286"
Prelude> show (43, 639)
"(43,639)"
Prelude> show [3, 4, 1, 0, 8]
"[3,4,1,0,8]"
```

6.2.13 Read

Read は、文字列から変換することのできる型をインスタンスとする型クラスです。

Show と同じように、Read のインスタンスも、Eq と同じです。

read という組み込み関数を使うことによって、文字列を Read のインスタンスのデータに変換することができます。read は、文字列に適用すると、それを Read のインスタンスのデータに変換した結果を戻り値として返します。どの型のデータに変換するかということは、処理系が型推論によって決定します。

```
Prelude> read "38" + 24
62
Prelude> read "False" && True
False
Prelude> map (*10) (read "[3, 7, 2, 4, 8]")
[30,70,20,40,80]
```

型推論によって型を決定することができない場合は、エラーになります。そのような場合は、型注釈を書くことによって、型を指定する必要があります。たとえば、

```
read "63" :: Int
```

というように、:: Int という型注釈を書いたとすると、read は文字列を Int のデータに変換します。

```
Prelude> read "63" :: Int
63
Prelude> read "63" :: Float
63.0
```

6.3 代数データ型

6.3.1 データ構成子

構成子識別子（英字の大文字で始まる識別子）が何らかのデータを生成する式である場合、その構成子識別子は、「データ構成子」(data constructor) と呼ばれます。

たとえば、「真偽値データ構成子」(Boolean data constructor) と呼ばれる True と False は、それぞれ、真偽値の真と偽を生成する式ですから、データ構成子です。

データ構成子から生成されるデータから構成される型は、「代数データ型」(algebraic data type) と呼ばれます。

たとえば、True と False という二つのデータ構成子から生成されるデータから構成される型は、代数データ型の一例です。

データ構成子は、代数データ型のデータを生成して、それを戻り値として返す、という動作をする関数に与えられた名前です。「データ構成子」という言葉は、「データ構成子を名前として持つ関数」という意味で使われることもあります。

6.3.2 型構成子

代数データ型は、「型構成子」(type constructor) と呼ばれるものから生成されます。

たとえば、`True` と `False` によって生成されるデータから構成される型は、`Bool` という型構成子から生成されます。

型構成子というのは、代数データ型を生成して、それを戻り値として返す、という動作をする関数のようなものです（データ構成子は正真正銘の関数ですが、型構成子は、あくまで「関数のようなもの」であって関数ではありません）。

なお、「型構成子」という言葉は、「型構成子に与えられた名前」という意味で使われることもあります。たとえば、`Bool` という名前を持つもののことを「型構成子」と呼ぶこともありますし、`Bool` という名前のことを「型構成子」と呼ぶこともあります。

6.3.3 ユーザー定義型構成子

型構成子を生成して、それに名前を与えることを、型構成子を「定義する」(define) と言います。

Haskell では、「型構成子定義」(type constructor definition) と呼ばれる記述をプログラムの中に書くことによって、型構成子を自由に定義することができます（型構成子定義の書き方については、第 6.4 節で説明することにしたいと思います）。プログラムの中に型構成子定義を書くことによって定義された型構成子は、「ユーザー定義型構成子」(user-defined type constructor) と呼ばれます。

6.3.4 組み込み型構成子

「ユーザー定義型構成子」の対義語は、「組み込み型構成子」です。「組み込み型構成子」(built-in type constructor) というのは、Haskell の処理系に組み込まれている型構成子のことです。組み込み型構成子は、型構成子定義を書かなくても利用することができます。`Bool` は、組み込み型構成子のひとつです。

組み込み型構成子としては、`Bool` のほかに、次のようなものもあります。

- `Ordering`
- `Maybe`
- `Either`

6.3.5 `Ordering`

`Ordering` は、大小関係をあらわすデータ（より小さい、等しい、より大きい）から構成される代数データ型を生成する型構成子です。

`Ordering` から生成される代数データ型のデータは、次の三つのデータ構成子から生成されます。

`LT` より小さい。

`EQ` 等しい。

`GT` より大きい。

`compare` という組み込み関数は、`Ord` のインスタンスのデータを、2 個、引数として受け取って、1 個目のほうが 2 個目よりも小さいならば `LT`、1 個目と 2 個目とが等しいならば `EQ`、1 個目のほうが 2 個目よりも大きいならば `GT` を戻り値として返します。

```
Prelude> compare False True
LT
Prelude> compare 8 8
EQ
Prelude> compare 'd' 'b'
GT
```

6.3.6 `Maybe`

`Maybe` は、「何らかの意味のあるデータかもしれない (maybe) し、いかなる意味もないデータかもしれない (maybe)」というデータから構成される代数データ型を生成する型構成子です。

`Maybe` から生成される代数データ型のデータは、`Nothing` と `Just` というデータ構成子から生成されます。

Nothing というデータ構成子は、「いかなる意味も持たないデータ」を生成します。それに対して、 Just というデータ構成子は、意味のあるデータを生成します。

Just は、1 個の引数を受け取るデータ構成子です。Just を使ってデータを生成したいときは、

```
Just 式
```

というように、 Just をデータに適用する関数適用を書きます。この中の「式」というところには、どんな式を書いてもかまいません。 Just を適用する関数適用を評価すると、引数として受け取ったデータから生成された Maybe のデータが、その値として得られます。

```
Prelude> Just True
Just True
Prelude> Just 'M'
Just 'M'
Prelude> Just "namako"
Just "namako"
```

第 6.3.2 項で説明したように、型構成子は、型を生成する関数のようなものです。それは、関数と同じように、引数を受け取ることができます。ただし、型構成子が受け取る引数は、データではなくて、型です。型構成子が引数として受け取る型は、「型引数」(type argument) と呼ばれます。

型引数を受け取る型構成子によって生成された型は、

```
型構成子 型式 ...
```

という型式によって記述されます。この中の「型式」のところには、型構成子が受け取った型引数の型を記述する型式が入ります。

たとえば、 Just True という式で Maybe のデータを生成したとすると、 Maybe は、 Bool を型引数として受け取ります。ですから、その式によって生成されるデータの型は、 Maybe Bool という型式によって記述されます。

```
Prelude> :t Just True
Just True :: Maybe Bool
Prelude> :t Just 'M'
Just 'M' :: Maybe Char
Prelude> :t Just "namako"
Just "namako" :: Maybe [Char]
```

データ構成子は、そのデータ構成子によって生成されたデータと一致するパターンとして使うことができます。たとえば、 Nothing というデータ構成子は、 Nothing によって生成されたデータと一致するパターンとして使うことができます。

Nothing ではない Maybe のデータは、

```
Just 識別子
```

というパターンと一致します。このパターンの中の識別子は、 Just が引数として受け取ったデータに束縛されます。ですから、このパターンと Maybe のデータとを一致させることによって、 Maybe のデータから、それを生成するために使われた元のデータを取り出すことができます。

6.3.7 lookup

Haskell では、意味のあるデータを返すことができない可能性のある関数は、通常、 Maybe のデータを返すように定義されます。

たとえば、 lookup という組み込み関数は、 Maybe のデータを返す関数です。

lookup は、

```
Eq a => a -> [(a, b)] -> Maybe b
```

という型を持つ関数です。この型式から分かるように、 lookup は 2 個の引数を受け取ります。1 個目は Eq のデータで、2 個目は、長さが 2 のタプルから構成されるリストです。それらのタプルの 1 個目の要素は、1 個目の引数と同じ型でないといけません。

lookup は、リストの先頭から末尾に向かって、1 個目の引数を 1 個目の要素として持つタプルを探索していきます。そして、そのようなタプルが見つかったならば、そのタプルの 2 個目の

要素から構成される Maybe のデータを戻り値として返します。末尾まで探索しても、そのようなタプルが見つからなかった場合は、 Nothing を返します。

```
Prelude> lookup 35 [(21, 'B'), (35, 'D'), (84, 'M')]
Just 'D'
Prelude> lookup 62 [(21, 'B'), (35, 'D'), (84, 'M')]
Nothing
```

次のプログラムは、 lookupOrDefault という関数を定義しています。この関数は、3 個の引数を受け取ります。1 個目は Eq のデータで、2 個目は、長さが 2 のタプルから構成されるリスト（1 個目の要素は 1 個目の引数と同じ型）で、3 個目は、2 個目の引数を構成しているタプルの 2 個目の要素と同じ型のデータです。この関数は、リストの先頭から末尾に向かって、1 個目の引数を 1 個目の要素として持つタプルを探索していきます。そして、そのようなタプルが見つかったならば、そのタプルの 2 個目の要素を戻り値として返します。末尾まで探索しても、そのようなタプルが見つからなかった場合は、3 個目の引数を返します。

プログラムの例 lookupordefault.hs

```
lookupOrDefault :: Eq a => a -> [(a, b)] -> b -> b
lookupOrDefault key xs d =
    case lookup key xs of
        Just y  -> y
        Nothing -> d
```

実行例

```
*Main> lookupOrDefault 'D' [('C', 38), ('D', 27)] 100
27
*Main> lookupOrDefault 'E' [('C', 38), ('D', 27)] 100
100
```

6.3.8 Either

Either は、二つの型のうちのどちらかの型のデータから構成される代数データ型を生成する型構成子です。

Either から生成される代数データ型のデータは、 Left と Right というデータ構成子から生成されます。

Left と Right は、どちらも、1 個の引数を受け取るデータ構成子です。

Either は、引数として 2 個の型を受け取る型構成子です。したがって、 Either は、2 個の型引数を持っています。 Left は、1 個目（左側）の型引数を、受け取った引数の型に束縛します。それに対して、 Right は、2 個目（右側）の型引数を、受け取った引数の型に束縛します。

```
Prelude> :t Left True
Left True :: Either Bool b
Prelude> :t Right 'd'
Right 'd' :: Either a Char
```

Either は、関数が結果を求めることに失敗した場合に、失敗の理由を戻り値として返すようにしたい、という場合に使われます。慣習として、結果を求めることに成功した場合は、その結果に Right を適用することによって生成されたデータを返して、失敗した場合は、その理由をあらわす文字列に Left を適用することによって生成されたデータを返すことになっています（right という英単語に「正しい」という意味があることから生まれた慣習だと思われます）。

次のプログラムの中で定義されている lookupMark という関数は、学生の出席番号 (Int) と、点数表（出席番号と点数 (Int) のタプルから構成されるリスト）を引数として受け取って、出席番号で指定された学生の点数を戻り値として返します。戻り値は、

```
Either String Int
```

という型を持つデータです。指定された出席番号のタプルが存在していて、かつ、点数が 0 点以上 100 点以下ならば、その点数に Right を適用したデータが戻り値となります。正しい結果を求めることができなかった場合は、次のようなデータを返します。

`Left "not found"`	指定された出席番号のタプルが見つからなかった場合
`Left "out of range"`	点数が 0 点以上 100 点以下ではない場合

プログラムの例 lookupmark.hs

```
lookupMark :: Int -> [(Int, Int)] -> Either String Int
lookupMark id xs =
    case lookup id xs of
        Just mark  -> if mark >= 0 && mark <= 100 then
                          Right mark
                      else
                          Left "out of range"
        Nothing    -> Left "not found"
```

実行例

```
*Main> lookupMark 3 [(1, 37), (2, 108), (3, 72), (4, 84)]
Right 72
*Main> lookupMark 5 [(1, 37), (2, 108), (3, 72), (4, 84)]
Left "not found"
*Main> lookupMark 2 [(1, 37), (2, 108), (3, 72), (4, 84)]
Left "out of range"
```

6.4 型構成子定義

6.4.1 型構成子定義の基礎

第 6.3.3 項で説明したように、型構成子は、「型構成子定義」(type constructor definition) と呼ばれる記述をプログラムの中に書くことによって、自由に定義することができます。プログラムの中に型構成子定義を書くことによって定義された型構成子は、「ユーザー定義型構成子」(user-defined type constructor) と呼ばれます。

型構成子定義は、基本的には、

data 構成子識別子 = 構成子識別子 | ··· | 構成子識別子

と書きます。それぞれの「構成子識別子」のところには、何らかの構成子識別子（英字の大文字で始まる識別子）を書きます。そうすると、イコールの左側に書かれた構成子識別子が、定義される型構成子の名前になって、イコールの右側に書かれたそれぞれの構成子識別子が、定義される型構成子のデータを生成するデータ構成子になります。たとえば、

```
data Season = Spring | Summer | Autumn | Winter
```

という型構成子定義を書くことによって、Season という名前を持つ型構成子を定義することができます。この型構成子は、4 個のデータから構成される型で、それぞれのデータは、Spring、Summer、Autumn、Winter というデータ構成子によって生成されます。

6.4.2 インスタンスの自動導出

型構成子定義には、「deriving 節」(deriving clause) と呼ばれる記述を付加することができます。

deriving 節は、インスタンスの「自動導出」(automatic derivation) と呼ばれる機能を使うための記述です。インスタンスの自動導出というのは、新しく定義される型構成子から生成される代数データ型を、自動的に何らかの型クラスのインスタンスにすることができるという機能のことです。ただし、インスタンスの自動導出ができる型クラスは、次のものだけです。

```
Eq  Ord  Enum  Bounded  Show  Read
```

自動導出によって Ord のインスタンスとなった代数データ型のデータは、型構成子定義の中でデータ構成子が並んでいる順番にしたがって、大小関係が与えられます。

自動導出によって Enum のインスタンスとなった代数データ型のデータは、型構成子定義の中でデータ構成子が並んでいる順番のとおりに列挙されているとみなされます。

deriving 節は、

deriving (型クラス名 , ···)

と書きます。この中の「型クラス名」のところには、定義される型構成子をそのインスタンスにしたい型クラスの名前を書きます。たとえば、

```
data Size = S | M | L | LL deriving (Eq, Ord)
```

という型構成子定義を書くことによって、Size という型構成子から生成される代数データ型を、Eq と Ord のインスタンスにすることができます。

次のプログラムの中で定義されている Season という型構成子から生成される代数データ型は、自動導出が可能なすべての型クラスのインスタンスです。

プログラムの例 season.hs

```
data Season = Spring | Summer | Autumn | Winter
    deriving (Eq, Ord, Enum, Bounded, Show, Read)
```

実行例

```
*Main> Spring == Summer
False
*Main> Spring /= Summer
True
*Main> Spring < Summer
True
*Main> fromEnum Autumn
2
*Main> toEnum 2 :: Season
Autumn
*Main> maxBound :: Season
Winter
*Main> show Autumn
"Autumn"
*Main> read "Autumn" :: Season
Autumn
*Main> [minBound..maxBound] :: [Season]
[Spring,Summer,Autumn,Winter]
```

6.4.3 引数を受け取るデータ構成子

第 6.3.1 項で説明したように、データ構成子というのは、代数データ型のデータを生成して、それを戻り値として返す、という動作をする関数のことです（そのような関数に与えられた名前も、「データ構成子」と呼ばれます）。

Bool のデータを生成する True と False や、先ほど定義を書いた Season のデータを生成するデータ構成子（Spring や Summer など）は、引数を受け取りませんが、データ構成子というのは関数ですから、引数を受け取るデータ構成子を定義するということも可能です。

引数を受け取るデータ構成子を定義したいときは、データ構成子の名前にする構成子識別子の右側に、受け取る引数の型を指定する型式を書きます。たとえば、型構成子定義のイコールの右側に、

```
Something Int Char Bool
```

と書くことによって、1 個目の引数として Int のデータ、2 個目の引数として Char のデータ、3 個目の引数として Bool のデータを受け取る、Something というデータ構成子を定義することができます。

6.4.4 四角形

引数を受け取るデータ構成子を持つ型構成子の例として、四角形 (quadrangle) のデータから構成される代数データ型を生成する、Quadrangle という型構成子を定義してみましょう。

Quadrangle のデータを生成するデータ構成子は、四角形の辺の長さや高さを引数として受け取ります。ただし、正方形 (square) のデータを生成するデータ構成子が受け取る引数は 1 個（1 辺の長さ）、平行四辺形 (parallelogram) のデータを生成するデータ構成子が受け取る引数は 2 個（底辺の長さ、高さ）、台形 (trapezoid) のデータを生成するデータ構成子が受け取る引数は 3 個（上底の長さ、下底の長さ、高さ）です。

Quadrangle を定義する型構成子定義は、次のように書くことができます。

プログラムの例 quadrangle.hs

```
data Quadrangle = Square Float |
                  Parallelogram Float Float |
```

```
                  Trapezoid Float Float Float
    deriving (Show)
```

まず、データ構成子の型を確認してみましょう。

実行例

```
*Main> :t Square
Square :: Float -> Quadrangle
*Main> :t Parallelogram
Parallelogram :: Float -> Float -> Quadrangle
*Main> :t Trapezoid
Trapezoid :: Float -> Float -> Float -> Quadrangle
```

それぞれのデータ構成子は、次のように、引数を受け取って四角形のデータを生成します。

実行例

```
*Main> Square 5
Square 5.0
*Main> Parallelogram 4 5
Parallelogram 4.0 5.0
*Main> Trapezoid 3 4 5
Trapezoid 3.0 4.0 5.0
```

第 6.3.6 項で説明したように、データ構成子は、そのデータ構成子によって生成されたデータと一致するパターンとして使うことができます。たとえば、

```
Parallelogram b h
```

というパターンは、

```
Parallelogram 7.6 4.3
```

という関数適用によって生成されたデータと一致します。この場合、 b は 7.6 に束縛されて、 h は 4.3 に束縛されます。

次のプログラムの中で定義されている area という関数は、引数として四角形のデータを受け取って、その四角形の面積を戻り値として返します。

プログラムの例 area.hs

```
data Quadrangle = Square Float |
                  Parallelogram Float Float |
                  Trapezoid Float Float Float

area :: Quadrangle -> Float
area (Square x)          = x*x
area (Parallelogram b h) = b*h
area (Trapezoid b1 b2 h) = (b1+b2)*h/2
```

実行例

```
*Main> area (Square 5)
25.0
*Main> area (Parallelogram 4 5)
20.0
*Main> area (Trapezoid 3 4 5)
17.5
```

6.4.5 データ構成子がひとつしかない代数データ型

先ほど定義した Quadrangle という型構成子が生成する四角形の代数データ型には、三つのデータ構成子があったわけですが、データを生成するデータ構成子がひとつしかない代数データ型を生成する型構成子を定義することも可能です。

データ構成子がひとつしかない代数データ型を生成する型構成子を定義する場合、「データ構成子と型構成子とは同じ識別子を使う」ということが慣例になっています。たとえば、赤、緑、青の三原色で色をあらわす代数データ型を生成する Color という型構成子を定義する場合、次のように、データ構成子にも Color という識別子を使います。

プログラムの例 color.hs

```
data Color = Color Int Int Int deriving (Eq, Show)
```

実行例

```
*Main> Color 0 128 255
Color 0 128 255
*Main> :t Color 0 128 255
Color 0 128 255 :: Color
```

6.5 型引数

6.5.1 型引数についての復習

第 6.3.2 項で説明したように、型構成子というのは、代数データ型を生成して、それを戻り値として返す、関数のようなものです（そのようなものに与えられた名前も、「型構成子」と呼ばれます)。

第 6.3.6 項で説明したように、型構成子は、関数と同じように、引数を受け取ることができます。ただし、型構成子が受け取る引数は、データではなくて、型です。型構成子が引数として受け取る型は、「型引数」(type argument) と呼ばれます。

Maybe という組み込み型構成子は、型引数を受け取る型構成子の一例です。Maybe は、型引数として 1 個の型を受け取って、その型のデータを持つデータから構成される型を生成します。

6.5.2 型引数を受け取る型構成子の定義

型引数を受け取る型構成子を定義したいときは、型構成子定義のイコールの左側に、

data 構成子識別子 変数識別子 ··· 変数識別子

という形のものを書きます。そうすると、構成子識別子は型構成子に束縛されて、変数識別子は、型構成子が引数として受け取った型に束縛されます。つまり、それらの変数識別子は、型変数だということです。型変数は、第 3.4.1 項で説明したように、通常は、a、b、c、d、……というように、1 文字の英小文字を使います。

6.5.3 型引数を受け取る型構成子の定義の例

型引数を受け取る型構成子を定義する例として、まず、Maybe という組み込み型構成子と同じ動作をする、MyMaybe という型構成子を定義してみましょう。

プログラムの例 mymaybe.hs

```
data MyMaybe a = MyJust a | MyNothing deriving (Show)
```

実行例

```
*Main> MyJust 'M'
MyJust 'M'
*Main> MyNothing
MyNothing
*Main> :t MyJust 'M'
MyJust 'M' :: MyMaybe Char
*Main> :t MyNothing
MyNothing :: MyMaybe a
```

次に、Either という組み込み型構成子と同じ動作をする、MyEither という型構成子を定義してみましょう。

プログラムの例 myeither.hs

```
data MyEither a b = MyLeft b | MyRight a deriving (Show)
```

実行例

```
*Main> MyRight True
MyRight True
*Main> MyLeft "error"
```

```
MyLeft "error"
*Main> :t MyRight True
MyRight True :: MyEither Bool b
*Main> :t MyLeft "error"
MyLeft "error" :: MyEither a [Char]
```

6.6 再帰的な型構成子

6.6.1 再帰的な型構成子の基礎

第 6.3.2 項で説明したように、「型構成子」というのは、代数データ型を生成して、それを戻り値として返す、という動作をする関数のようなもの（またはそのようなものに与えられた名前）のことです。

ところで、関数というのは、第 3.6 節で説明したように、再帰的に定義することができます。型構成子も、「関数のようなもの」ですから、やはり再帰的に定義することが可能です。

6.6.2 リスト型を生成する型構成子の定義

リスト型というのは、実は、再帰的に定義された代数データ型です。そして、「コンス」(cons) と呼ばれる中置演算（演算子は : ）は、リストを生成するデータ構成子です。

リスト型を生成する型構成子は Haskell の処理系に組み込まれているわけですが、それとは別に、リスト型を生成する独自の型構成子を定義することも可能です。

次のプログラムの中で定義されている `List` という型構成子は、要素の型を引数として受け取って、独自のリスト型を生成します。`Nil` というのが、独自のリスト型の空リストを生成するデータ構成子で、`Cons` というのが、頭部と尾部を引数として受け取って、それらから構成される独自のリスト型のリストを生成するデータ構成子です。

プログラムの例 `list.hs`

```
data List a = Nil | Cons a (List a) deriving (Show)
```

実行例

```
*Main> Nil
Nil
*Main> Cons 59 Nil
Cons 59 Nil
*Main> Cons 24 (Cons 59 Nil)
Cons 24 (Cons 59 Nil)
*Main> Cons 81 (Cons 24 (Cons 59 Nil))
Cons 81 (Cons 24 (Cons 59 Nil))
*Main> Cons 37 (Cons 81 (Cons 24 (Cons 59 Nil)))
Cons 37 (Cons 81 (Cons 24 (Cons 59 Nil)))
```

次のプログラムの中で定義されている `listToMyList` という関数は、引数としてリストを受け取って、それを独自のリスト型のリストに変換した結果を戻り値として返します。

`myListToList` という関数は、それとは逆に、独自のリスト型のリストを引数として受け取って、それを処理系に組み込まれているリスト型のリストに変換した結果を戻り値として返します。

プログラムの例 `listtomylist.hs`

```
data List a = Nil | Cons a (List a) deriving (Show)

listToMyList :: [a] -> List a
listToMyList []     = Nil
listToMyList (x:xs) = Cons x (listToMyList xs)

myListToList :: List a -> [a]
myListToList Nil         = []
myListToList (Cons x xs) = x:(myListToList xs)
```

実行例

```
*Main> listToMyList [37, 81, 24, 59]
Cons 37 (Cons 81 (Cons 24 (Cons 59 Nil)))
```

```
*Main> myListToList (Cons 37 (Cons 81 (Cons 24 (Cons 59 Nil))))
[37,81,24,59]
```

6.6.3 二分木型を生成する型構成子の定義

リストというのが一つの方向に再帰が深くなっていく構造を持っているのに対して、二本の枝に枝分かれしながら再帰が深くなっていく構造を持つデータは、「二分木」(binary tree) と呼ばれます。

空ではない任意の二分木は、1 個の要素、左の二分木、右の二分木、という三つの部分から構成されます。

次のプログラムの中で定義されている `Tree` という型構成子は、要素の型を引数として受け取って、二分木型を生成します。`Nil` というのが空の二分木を生成するデータ構成子で、`Node` というのが、要素、左の二分木、右の二分木、という三つの引数を受け取って、それらから構成される二分木を生成するデータ構成子です。

プログラムの例 `tree.hs`

```
data Tree a = Nil | Node a (Tree a) (Tree a) deriving (Show)
```

実行例

```
*Main> Nil
Nil
*Main> Node 53 Nil Nil
Node 53 Nil Nil
*Main> Node 87 (Node 53 Nil Nil) (Node 46 Nil Nil)
Node 87 (Node 53 Nil Nil) (Node 46 Nil Nil)
```

次のプログラムの中で定義されている `treeSize` という関数は、引数として二分木を受け取って、それに含まれている要素の個数を戻り値として返します。

プログラムの例 `treesize.hs`

```
data Tree a = Nil | Node a (Tree a) (Tree a) deriving (Show)

treeSize :: Tree a -> Int
treeSize Nil = 0
treeSize (Node _ left right) =
    treeSize left + treeSize right + 1
```

実行例

```
*Main> treeSize Nil
0
*Main> treeSize (Node 53 Nil Nil)
1
*Main> treeSize (Node 87 (Node 53 Nil Nil) (Node 46 Nil Nil))
3
```

6.7 型クラス定義

6.7.1 型クラス定義の書き方

第 6.2.3 項で説明したように、Haskell では、「型クラス定義」(type class definition) と呼ばれる記述をプログラムの中に書くことによって、型クラスを自由に定義することができます。プログラムの中に型クラス定義を書くことによって定義された型クラスは、「ユーザー定義型クラス」(user-defined type class) と呼ばれます。

型クラス定義は、

```
class [構成子識別子] [変数識別子] where
    [型シグネチャー宣言]
        •
        •
        •
```

と書きます。この中の「構成子識別子」のところには、定義される型クラスに与える構成子識別子を書きます。「変数識別子」のところには、何らかの変数識別子を書きます。この変数識別子は、型に束縛される型変数ですので、通常は a と書きます。そして、「型シグネチャー宣言」のところには、定義される型クラスのインスタンスのデータに共通して適用することのできる関数の型シグネチャー宣言を、上に書いた型変数を使って書きます。

型クラスを定義する例として、循環的なデータをインスタンスとする、Circular という型クラスを定義してみましょう。型クラス定義は、次のように書くことができます。

```
class Circular a where
    next :: a -> a
    prev :: a -> a
```

next は、Circular のインスタンスのデータを引数として受け取って、そのデータの次のデータを戻り値として返す関数です。同様に、prev は、前のデータを返す関数です。

6.7.2　インスタンス定義の書き方

何らかの型を、自動導出ができない型クラスのインスタンスにするためには、「インスタンス定義」(instance definition) と呼ばれるものを書く必要があります。

インスタンス定義は、

```
instance 型クラス名 型式 where
    関数定義
      ⋮
```

と書きます。この中の「型クラス名」のところには、何らかの型クラスの名前を、「型式」のところには、その型クラスのインスタンスにしたい型をあらわす型式を書きます。そして、「関数定義」のところには、その型クラスのインスタンスのデータに共通して適用することのできる関数を定義する関数定義を書きます（型シグネチャー宣言は、すでに型クラス定義の中に書かれていますので、ここでは省略することができます）。

たとえば、四季をあらわす、

```
data Season = Spring | Summer | Autumn | Winter
```

という循環的なデータが定義されているとするときに、

```
instance Circular Season where
    next Spring = Summer
    next Summer = Autumn
    next Autumn = Winter
    next Winter = Spring

    prev Spring = Winter
    prev Summer = Spring
    prev Autumn = Summer
    prev Winter = Autumn
```

というインスタンス定義を書くと、Season は、Circular のインスタンスになります。ですから、

```
prevAndNext :: Circular a => a -> (a, a)
prevAndNext x = (prev x, next x)
```

というような、Circular のインスタンスのデータに適用することのできる関数は、Season のデータにも適用することができます。ちなみに、この関数は、引数として受け取ったデータの次のデータと前のデータから構成されるタプルを戻り値として返します。

プログラムの例　circular.hs

```
class Circular a where
    next :: a -> a
    prev :: a -> a

data Season = Spring | Summer | Autumn | Winter
    deriving (Eq, Show)
```

```
data Janken = Rock | Paper | Scissors deriving (Eq, Show)

instance Circular Season where
    next Spring = Summer
    next Summer = Autumn
    next Autumn = Winter
    next Winter = Spring

    prev Spring = Winter
    prev Summer = Spring
    prev Autumn = Summer
    prev Winter = Autumn

instance Circular Janken where
    next Rock     = Paper
    next Paper    = Scissors
    next Scissors = Rock

    prev Rock     = Scissors
    prev Paper    = Rock
    prev Scissors = Paper

prevAndNext :: Circular a => a -> (a, a)
prevAndNext x = (prev x, next x)
```

実行例

```
*Main> prevAndNext Spring
(Winter,Summer)
*Main> prevAndNext Rock
(Scissors,Paper)
```

6.8 レコード

6.8.1 レコードの基礎

代数データ型には、「レコード」(record) と呼ばれるデータから構成されるものもあります。

ひとつのレコードは、何個かのデータから構成されます。レコードを構成しているそれぞれのデータは、「フィールド」(field) と呼ばれます。

タプルやリストも何個かのデータから構成されますが、タプルやリストの要素が一列に並んでいるのに対して、フィールドは、順番というものを持っていません。

レコードを構成しているそれぞれのフィールドは、それに与えられた変数識別子によって識別されます。フィールドを識別するための変数識別子は、「フィールド名」(field name) と呼ばれます。

6.8.2 レコード型を生成する型構成子の定義

レコードから構成される代数データ型のことを、「レコード型」(record type) と呼ぶことにしましょう。

レコード型を生成する型構成子定義は、

`data` 構成子識別子 `=` 構成子識別子 `{` フィールド定義 `, ··· }`

と書きます。この中の「構成子識別子」のところには、型構成子とデータ構成子に束縛される構成子識別子を書きます。データ構成子はひとつだけですから、慣例に則って、型構成子とデータ構成子の両方に同じ識別子を使います。そして、「フィールド定義」のところには、フィールドを定義する記述を書きます。

フィールド定義は、

変数識別子 `::` 型式

と書きます。この中の「変数識別子」のところには、フィールド名にする変数識別子を書いて、

「型式」のところには、フィールドの型をあらわす型式を書きます。たとえば、

```
price :: Int
```

というフィールド定義を書くことによって、price というフィールド名によって識別される Int のフィールドを定義することができます。

次の型構成子定義は、名前 (name)、価格 (price)、数量 (quantity) という三つのフィールドを持つレコードから構成されるレコード型を生成する、Stock という型構成子を定義します。

```
data Stock = Stock { name :: String,
                     price :: Int,
                     quantity :: Int
                   }
```

6.8.3　レコードを生成する式

レコードを生成するためには、そのための式を評価する必要があります。レコードを生成する式は、

データ構成子 { フィールド生成子 , ··· }

と書きます。この中の「データ構成子」のところには、レコードを生成するデータ構成子を書きます。そして「フィールド生成子」のところには、

フィールド名 = 式

という形のものを書きます。そうすると、イコールの右側の式を評価することによって得られた値が、イコールの左側のフィールド名によって識別されるフィールドになります。たとえば、フィールド生成子として、

```
name="glass"
```

というものを書くことによって、name というフィールド名によって識別される、glass という文字列のフィールドを生成することができます。

レコードを生成する式の中に書くフィールド生成子は、そのレコード型で定義されているすべてのフィールドを網羅している必要があります。

フィールドというのはレコードの中での順序を持っていませんので、フィールド生成子も、レコードを生成する式の中に、どんな順序で書いてもかまいません。

プログラムの例　stock.hs

```
data Stock = Stock { name :: String,
                     price :: Int,
                     quantity :: Int
                   } deriving (Show)
```

実行例

```
*Main> Stock { name="spoon", price=580, quantity=37 }
Stock {name = "spoon", price = 580, quantity = 37}
*Main> Stock { quantity=61, name="fork", price=720 }
Stock {name = "fork", price = 720, quantity = 61}
```

次のプログラムの中で定義されている toStock という関数は、名前と価格と数量を引数として受け取って、それらから構成される Stock のレコードを戻り値として返します。

プログラムの例　tostock.hs

```
data Stock = Stock { name :: String,
                     price :: Int,
                     quantity :: Int
                   } deriving (Show)

toStock :: String -> Int -> Int -> Stock
toStock n p q = Stock { name=n, price=p, quantity=q }
```

実行例

```
*Main> toStock "teacup" 430 56
```

```
Stock {name = "teacup", price = 430, quantity = 56}
```

6.8.4 フィールド名

フィールド名は、レコードからフィールドを取り出す関数に束縛されています。たとえば、

```
data Stock = Stock { name :: String,
                     price :: Int,
                     quantity :: Int
                   }
```

という型構成子定義によって定義されたレコード型の price というフィールド名は、

```
Stock -> Int
```

という型を持つ関数に束縛されています。この関数を Stock のレコードに適用すると、戻り値として、 price というフィールド名によって識別されるフィールドが得られます。

次のプログラムの中で定義されている fromStock という関数は、 Stock のレコードを引数として受け取って、名前、価格、数量から構成されるタプルを戻り値として返します。

プログラムの例 fromstock.hs

```
data Stock = Stock { name :: String,
                     price :: Int,
                     quantity :: Int
                   } deriving (Show)

toStock :: String -> Int -> Int -> Stock
toStock n p q = Stock { name=n, price=p, quantity=q }

fromStock :: Stock -> (String, Int, Int)
fromStock s = (name s, price s, quantity s)
```

実行例

```
*Main> fromStock (toStock "bowl" 870 22)
("bowl",870,22)
```

6.8.5 レコードのパターン

レコードからフィールドを取得する方法としては、フィールド名がそれに束縛されている関数を適用するという方法のほかに、レコードのパターンとレコードとを一致させるという方法もあります。

レコードのパターンは、

データ構成子 { フィールドパターン , ··· }

と書きます。この中の「データ構成子」のところには、レコードを生成するデータ構成子を書きます。そして「フィールドパターン」のところには、

フィールド名 = 変数識別子

という形のものを書きます。そうすると、レコードのパターンがレコードと一致した場合、イコールの右側の変数識別子が、フィールド名によって識別されるフィールドに束縛されます。たとえば、

```
name=n
```

というフィールドパターンを書くことによって、 n という変数識別子を、 name というフィールド名によって識別されるフィールドに束縛することができます。

次のプログラムは、先ほどのプログラムの中で定義されている fromStock を、レコードのパターンを使って書き直したものです。

プログラムの例 fromstock2.hs

```
data Stock = Stock { name :: String,
                     price :: Int,
                     quantity :: Int
                   } deriving (Show)
```

```
toStock :: String -> Int -> Int -> Stock
toStock n p q = Stock { name=n, price=p, quantity=q }

fromStock :: Stock -> (String, Int, Int)
fromStock (Stock { name=n, price=p, quantity=q }) = (n, p, q)
```

実行例

```
*Main> fromStock (toStock "dish" 530 74)
("dish",530,74)
```

6.9 型シノニム

6.9.1 型シノニムの基礎

第 1.3.7 項で説明したように、文字列の型を記述する [Char] という型式には、String という同義語が定義されています。この String のような、何らかの型式の同義語であると定義された構成子識別子（英字の大文字で始まる識別子）は、「型シノニム」(type synonym) と呼ばれます。

型シノニムは、プログラムの中で自由に定義することができますので、これを有効に活用することによって、型式を分かりやすく記述することができるようになります。

6.9.2 型シノニム定義

型シノニムを定義したいときは、「型シノニム定義」(type synonym definition) と呼ばれるものを書きます。

型シノニム定義は、

type 構成子識別子 = 型式

と書きます。この中の「型式」のところには、型シノニムを定義したい型を記述する型式を書いて、「構成子識別子」のところには、その型式の型シノニムにしたい構成子識別子を書きます。たとえば、

```
type Polyline = [(Int, Int)]
```

という型シノニム定義を書くことによって、[(Int, Int)] という型式に対して、Polyline という型シノニムを定義することができます。

6.9.3 型引数を受け取る型シノニム

型構成子と同じように、型シノニムも、引数として型を受け取ることができます。型構成子の場合と同じように、型シノニムが引数として受け取る型も、「型引数」(type argument) と呼ばれます。

型引数を受け取る型シノニムを定義したいときは、型シノニム定義の中の構成子識別子とイコールのあいだに、受け取る型と同じ個数の変数識別子を書きます。そうすると、それらの変数識別子が型変数になって、型シノニムが引数として受け取った型に束縛されます。たとえば、

```
type Dictionary a b = [(a, b)]
```

という型シノニム定義を書くことによって、引数として二つの型を受け取る、Dictionary という型シノニムを定義することができます。

次のプログラムは、Dictionary という型シノニムを使って、組み込み関数の lookup と同じ動作をする mylookup という関数を定義しています。

プログラムの例 mylookup.hs

```
type Dictionary a b = [(a, b)]

mylookup :: Eq a => a -> Dictionary a b -> Maybe b
mylookup key [] = Nothing
mylookup key ((x, y):xys)
  | key == x  = Just y
  | otherwise = lookup key xys
```

実行例

```
*Main> mylookup 35 [(21, 'B'), (35, 'D'), (84, 'M')]
Just 'D'
*Main> mylookup 62 [(21, 'B'), (35, 'D'), (84, 'M')]
Nothing
```

第 7 章　入出力

7.1　入出力の基礎

7.1.1　参照透過性

Haskell というプログラミング言語は、「参照透過性」(referential transparency) と呼ばれる性質を持っています。参照透過性というのは、「式の値が、その式の構成要素のみによって定まる」という性質、言い換えれば、「式の値が、その式の外側に存在する状態に左右されない」という性質のことです。参照透過性を持つ Haskell のようなプログラミング言語は、「純粋」(pure) であると言われます。

純粋なプログラミング言語においては、関数の戻り値は、それを適用した引数のみによって決定されます。

7.1.2　副作用

コンピュータが実行した動作によって引き起こされる、何らかの状態の変化は、その動作の「副作用」(side effect) と呼ばれます。純粋なプログラミング言語においては、副作用を伴う関数というものは存在しません。なぜなら、副作用を伴う関数の存在を許すことは、参照透過性を犠牲にすることだからです。

入力装置からデータを読み込んだり、出力装置にデータを出力したりするというコンピュータの動作は、「入出力」(input/output, I/O) と呼ばれます。入出力を実行すると、何らかの状態が変化することになりますから、入出力というのは副作用の一種だということになります。

純粋ではないプログラミング言語においては、入出力は、副作用を伴う関数によって実行されます。それに対して、純粋なプログラミング言語には、入出力という副作用を伴う関数は存在しません。

ところで、Haskell というのは純粋なプログラミング言語の一種です。ということは、Haskell では入出力を実行するプログラムを書くことはできない、ということになるのでしょうか。

もちろん、そんなことはありません。Haskell でも、入出力を実行するプログラムを書くことは可能です。

7.1.3　標準出力への出力と標準入力からの読み込み

Haskell では、次のような変数識別子に束縛されているデータを使うことによって、標準出力に文字列を出力したり、標準入力から文字列を読み込んだりすることができます。

`putStr`	標準出力に文字列を出力します。
`putStrLn`	標準出力に文字列を出力して、そののち改行を出力します。
`getLine`	標準入力から 1 行の文字列を読み込みます。

```
Prelude> putStr "Quo Vadis"
Quo VadisPrelude> putStrLn "Il pendolo di Foucault"
Il pendolo di Foucault
Prelude> getLine
A Midsummer Night's Dream
"A Midsummer Night's Dream"
```

7.1.4　I/O アクション

ところで、`putStr` という変数識別子は、どのようなデータに束縛されているのでしょうか。`:t` コマンドで、その型を調べてみましょう。

```
Prelude> :t putStr
putStr :: String -> IO ()
```

このように、putStrLn という変数識別子は、引数として文字列を受け取って、IO () という型のデータを戻り値として返す関数に束縛されています。IO というのは、1 個の型引数を受け取る組み込み型構成子です。

IO という型構成子が生成する型を持つデータは、「I/O アクション」(I/O action) と呼ばれます。I/O アクションは、副作用を伴うかもしれない何らかの動作をあらわすデータです。

I/O アクションは、動作をあらわすデータですから、それを実行するということが可能です。I/O アクションを実行すると、副作用を伴うかもしれない何らかの動作が実行されます。そして、I/O アクションは、その動作の結果として、ひとつのデータを生成します。I/O アクションが生成するデータは、その I/O アクションの「結果」(result) と呼ばれます。IO という型構成子が受け取る型引数は、I/O アクションが生成する結果の型です。

入出力を実行するための Haskell の関数は、副作用を伴う動作を自分自身が実行するのではありません。そうではなくて、副作用を伴う動作をあらわす I/O アクションを戻り値として返すのです。たとえば、putStr という組み込み関数は、「引数として文字列を受け取って、その文字列を標準出力に出力する」という動作をする関数ではなくて、「引数として文字列を受け取って、「受け取った引数を標準出力に出力して、そののち改行を出力する」という動作をあらわす I/O アクションを戻り値として返す」という動作をする関数です。

putStr や putStrLn という組み込み関数は、IO () という型の I/O アクションを戻り値として返します。() というのは、「ユニット型」(unit type) と呼ばれる型をあらわす型式です。ユニット型は、() という式を評価することによって得られる、「ユニット」(unit) と呼ばれるデータのみから構成される型です。

putStr や putStrLn が返す I/O アクションは、結果を生成する必要がありません。そのような、結果を生成する必要のない I/O アクションは、通常、結果としてユニットを生成します。

次に、getLine という変数識別子が、どのようなデータに束縛されているのかということを、:t コマンドで調べてみましょう。

```
Prelude> :t getLine
getLine :: IO String
```

このように、getLine が束縛されているデータの型は、-> という型演算子を含んでいません。つまり、getLine という変数識別子は、関数ではないデータに束縛されているということです。

getLine が束縛されているデータの型は、IO String です。getLine は、この型式から分かるように、結果として文字列を生成する I/O アクションに束縛されています。この I/O アクションは、「標準入力から 1 行の文字列を読み込んで、その文字列を結果として生成する」という動作をあらわしています。

getLine が束縛されているデータは、I/O アクションであって、文字列ではありません。ですから、

```
getLine ++ "udon"
```

というように、getLine が束縛されているデータと文字列とを連結しようとすると、エラーになります。

7.1.5　GHCi による I/O アクションの実行

ところで、I/O アクションを求めることだけが putStr や getLine の仕事だとすると、その I/O アクションは、いったい誰によって実行されるのでしょうか。

値として I/O アクションが得られる式を GHCi に入力した場合、その I/O アクションを実行するのは、GHCi です。

GHCi は、入力された式の値が I/O アクションだった場合、その I/O アクションを実行して、その戻り値を出力します。ただし、I/O アクションの結果がユニットだった場合、そのユニットは出力しません。

7.2　モナド

7.2.1　モナドの基礎

Haskell には、Monad という組み込み型クラスがあります。この型クラスのインスタンスは、「モナド」(monad) と呼ばれます。たとえば、Maybe という型構成子から生成された型は、モナドです。

型としてモナドを持つデータは「モナド値」(monadic value) と呼ばれ、モナド値を値とする式は「モナド式」(monadic expression) と呼ばれます。

モナド値というのは、1 個の箱と、その中に入った 1 個のデータから構成されていると考えることができます。モナド値の箱は、何らかの文脈をあらわしています。たとえば、Maybe のモナド値の場合、箱があらわしているのは、失敗しているかもしれない可能性という文脈です。

Monad という型クラスは、次のように定義されています。

```
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a
```

この型クラス定義から分かるように、Monad という型クラスのインスタンスは、>>= という二項演算をメソッドとして共有しています。この演算は、「バインド」(bind) と呼ばれるもので、二つのモナド値を合成する (combine) という動作をします。

7.2.2 直列化

IO という型構成子が生成する型も、Monad のインスタンスです。したがって、IO が生成する型はモナドであり、I/O アクションはモナド値である、ということになります。

I/O アクションはモナド値ですから、文脈をあらわす箱と、その中のデータから構成されます。I/O アクションの箱があらわしている文脈は、入出力という副作用です。そして、I/O アクションの中のデータは、I/O アクションを実行することによって生成される結果です。

副作用を伴うプログラムを書くためには、まずこの動作をして、次にこの動作をして、それからこの動作をして、というように、副作用を伴う個々の動作が実行される順番を記述する必要があります。

Haskell では、いくつかの I/O アクションを直列化した I/O アクションを作る式を書くことによって、副作用を伴う個々の動作が実行される順番を記述します。「直列化」(sequence) というのは、実行される順番のとおりにいくつかの I/O アクションをひとつにまとめて、それらを順番に実行する I/O アクションを作る、ということです。

7.2.3 バインドによる I/O アクションの直列化

I/O アクションは、バインドを使って合成することによって、直列化することができます。「バインド」(bind) というのは、第 7.2.1 項で説明したように、>>= という二項演算のことです。

>>= の左側には、I/O アクションを求める式を書きます。そして右側には、1 個のデータを引数として受け取って I/O アクションを戻り値として返す関数を求める式を書きます。そうすると、>>= は、左側の I/O アクションを実行してから、右側の関数の戻り値として得られた I/O アクションを実行する、という動作をあらわす I/O アクションを戻り値として返します。右側の関数は、左側の式の値として得られた I/O アクションの結果に適用されます。

例として、

```
getLine >>= \line -> putStrLn line
```

という式を評価した場合、>>= はどのような I/O アクションを戻り値として返すか、ということについて考えてみましょう。

>>= の左側の式の値は、「標準入力から 1 行の文字列を読み込んで、その文字列を結果として生成する」という動作をあらわす I/O アクションです。そして右側の式の値は、「引数として文字列を受け取って、「その文字列を標準出力に出力する」という動作をあらわす I/O アクションを戻り値として返す」という動作をする関数です。この関数は、左側の式の値として得られた I/O アクションの結果、つまり読み込まれた文字列に適用されます。ですから、>>= は、「標準入力から 1 行の文字列を読み込んで、そののちその文字列を出力する」という動作をあらわす I/O アクションを戻り値として返すことになります。

```
Prelude> getLine >>= \line -> putStrLn line
Alice's Adventures in Wonderland
Alice's Adventures in Wonderland
```

左側の I/O アクションの結果が必要ではない場合は、>>= の代わりに、>> という二項演算を使います。

>> の左右には、I/O アクションを求める式を書きます。そうすると、>> は、「左側の I/O アクションを実行してから右側の I/O アクションを実行する」という動作をあらわす I/O アクションを戻り値として返します。たとえば、

```
putStrLn "namako" >> putStrLn "umiushi"
```

という式を評価すると、その値として、「namako という文字列を出力して、そののち umiushi という文字列を出力する」という動作をあらわす I/O アクションが得られます。

```
Prelude> putStrLn "namako" >> putStrLn "umiushi"
namako
umiushi
```

次のプログラムの中で定義されている strLength という I/O アクションは、「標準入力から 1 行の文字列を読み込んで、その文字列の長さを標準出力に出力する」という動作をあらわしています。

プログラムの例 bind.hs

```
strLength :: IO ()
strLength =
    putStr "enter string: " >>
    getLine >>= \line ->
    putStrLn ("length of " ++ line ++ ": " ++
        (show (length line)))
```

実行例

```
*Main> strLength
enter string: hamaguri
length of hamaguri: 8
```

7.2.4 return

Monad という型クラスのインスタンスは、return という関数をメソッドとして共有しています。この関数は、引数として任意のデータを受け取って、そのデータを箱に入れてモナド値にした結果を戻り値として返します。

return を使うことによって、モナド値ではないデータをモナド値にすることができます。

次のプログラムの中で定義されている getInteger という I/O アクションは、「標準入力から 1 行の文字列を読み込んで、その文字列を整数に変換したものを結果として生成する」という動作をあらわしています。

プログラムの例 getinteger.hs

```
getInteger :: IO Integer
getInteger =
    putStr "enter integer: " >>
    getLine >>= \line ->
    return (read line :: Integer)
```

実行例

```
*Main> getInteger
enter integer: 3704
3704
```

次のプログラムの中で定義されている doOrDoNot という関数は、引数として真偽値を受け取って、引数が真ならば、

```
Do, or do not. There is no try.
```

という文字列を標準出力に出力するという動作をあらわす I/O アクションを戻り値として返して、引数が偽ならば、何もしないという動作をあらわす I/O アクションを戻り値として返します。

プログラムの例 doordonot.hs

```
doOrDoNot :: Bool -> IO ()
doOrDoNot True = putStrLn "Do, or do not. There is no try."
doOrDoNot _    = return ()
```

実行例

```
*Main> doOrDoNot True
Do, or do not. There is no try.
*Main> doOrDoNot False
*Main>
```

7.3 do 記法

7.3.1 do 記法の基礎

モナド値を合成する式は、>>= や >> を使う代わりに、「do 記法」(do notation) と呼ばれる式を使って書くこともできます。

do 記法は、

do ブロック

と書きます。この中の「ブロック」のところには、第 3.7.2 項で説明したブロックを書きます。do 記法のブロックの中には、モナド式を書くことができます。

do 記法を評価すると、その値として、その中に書かれたすべてのモナド式の値を >> で合成したモナド値が得られます。たとえば、

```
do { putStrLn "namako"; putStrLn "umiushi" }
```

という do 記法を評価すると、その値として、namako という文字列を出力して、そののち umiushi という文字列を出力する、という動作をあらわす I/O アクションが得られます。

```
Prelude> do { putStrLn "namako"; putStrLn "umiushi" }
namako
umiushi
```

7.3.2 モナド値からデータを取り出す記述

do 記法の中で、モナド値からデータを取り出す必要があるときは、

パターン <- モナド式

という形の記述を書きます。この中の「パターン」のところには何らかのパターンを、「モナド式」のところには何らかのモナド式を書きます。そうすると、モナド式の値として得られたモナド値の中のデータとパターンとが一致する場合は、パターンの中の変数識別子がデータに束縛されます。たとえば、

```
do { line <- getLine; putStrLn line }
```

という do 記法を評価すると、その値として、標準入力から 1 行の文字列を読み込んで、そののちその文字列を出力する、という動作をあらわす I/O アクションが得られます。

```
Prelude> do { line <- getLine; putStrLn line }
The Book of the New Sun
The Book of the New Sun
```

次のプログラムは、第 7.2.3 項で紹介したプログラムを、do 記法を使って書き直したものです。

プログラムの例 donotation.hs

```
strLength :: IO ()
strLength = do
    putStr "enter string: "
    line <- getLine
    putStrLn ("length of " ++ line ++ ": " ++
        (show (length line)))
```

実行例

```
*Main> strLength
enter string: hamaguri
length of hamaguri: 8
```

7.3.3 let 式

do 記法を書くとき、その中で、モナド式ではない式の値に識別子を束縛したい、ということがしばしばあります。そのようなときは、第 3.7.4 項で紹介した let 式を do 記法の中に書きます。ただし、 do 記法の中に書く let 式は、 in 以降を省略した、

```
let ブロック
```

という形のものです。たとえば、

```
let x :: String
    x = "hitode"
```

という let 式を書くことによって、 x という変数識別子を hitode という文字列に束縛することができます。

do 記法の中の let 式によってデータに束縛された変数識別子のスコープは、その let 式から、do 記法の末尾までです。

次のプログラムは、第 7.3.2 項で紹介したプログラムを、 let 式を使って書き直したものです。

プログラムの例　let.hs

```
strLength :: IO ()
strLength = do
    putStr "enter string: "
    line <- getLine
    let s :: String
        s = "length of " ++ line ++ ": " ++
                (show (length line))
    putStrLn s
```

実行例

```
*Main> strLength
enter string: hamaguri
length of hamaguri: 8
```

7.4 コンパイル

7.4.1 この節について

第 1.2.1 項で説明したように、言語処理系には、コンパイラとインタプリタという 2 種類のものがあります。

第 1.2.4 項で説明したように、Haskell Pratform をコンピュータにインストールすると、GHC というコンパイラと GHCi というインタプリタがインストールされます。

このチュートリアルでは、これまでは GHCi ばかり使ってきたわけですが、この節では、GHC の使い方について説明したいと思います。

7.4.2 ソースコードとバイナリーコード

第 1.2.1 項で説明したように、コンパイラというのは、人間が書いたプログラムを機械語に翻訳するプログラムのことです。

コンパイラが処理の対象とする、人間によって書かれたプログラムは、「ソースコード」(source code) と呼ばれます。それに対して、コンパイラが出力した機械語のプログラムは、「バイナリーコード」(binary code) と呼ばれます。

そして、ソースコードが格納されているファイルは「ソースファイル」(source file) と呼ばれ、バイナリーコードが格納されているファイルは「バイナリーファイル」(binary file) と呼ばれます。

7.4.3 main

Haskell のコンパイラは、Haskell で書かれたソースコードを読み込んで、「main という変数識別子が束縛されている I/O アクションを実行する」という動作をするオブジェクトコードを出力します。ですから、コンパイルして実行する Haskell のプログラムを書く場合は、その中で、main という識別子を I/O アクションに束縛する必要があります。

Haskell のコンパイラは、main という変数識別子をデータに束縛していないソースコードが与えられた場合、main がないというエラーメッセージを出力します。また、main という変数識別子がデータに束縛されてはいるけれども、そのデータが I/O アクションではない場合、Haskell のコンパイラは、main の型が正しくないというエラーメッセージを出力します。

7.4.4 Hello, world

プログラミング言語の入門書の多くは、読者が最初に書いてみるべきプログラムとして、「標準出力に文字列を出力する」という動作をするものを紹介しています。それらのプログラムは、伝統的に、

```
Hello, world!
```

という文字列を出力します。このことから、プログラミング言語の初心者が最初に書いてみるべきプログラムは、そのプログラミング言語の「Hello, world」と呼ばれます。

Haskell の Hello, world は、次のようなプログラムです。

プログラムの例 hello.hs

```
main :: IO ()
main = putStrLn "Hello, world!"
```

7.4.5 GHC の使い方

GHC を使って Haskell のソースコードをコンパイルしたいときは、コマンドを入力するためのアプリ（Linux や Mac OS ならばターミナル、Windows ならばコマンドプロンプト）に、

```
ghc パス名
```

というコマンドを入力します。この中の「パス名」のところには、ソースコードが格納されているファイルを指定するパス名を書きます。たとえば、先ほど紹介した Haskell の Hello, world は、

```
ghc hello.hs
```

というコマンドを入力することによって、コンパイルすることができます。

Linux や Mac OS の上で動作する GHC は、ソースコードが格納されているファイルの名前から .hs という拡張子を取り除いた名前のファイルにバイナリーコードを出力します。Windows の上で動作する GHC は、ソースコードが格納されているファイルの拡張子を .exe に変更した名前のファイルにバイナリーコードを出力します。

それでは、Haskell の Hello, world を実行してみましょう。Linux や Mac OS を使っている場合は、

```
./hello
```

というコマンドを入力してください。Windows を使っている場合は、

```
hello
```

というコマンドを入力してください。

7.4.6 もう少し複雑なプログラム

Hello, world よりももう少し複雑なプログラムを書くためには、バインドまたは do 記法を使って、I/O アクションを直列化することが必要になります。

次のプログラムは、標準入力から読み込んだ個数だけアスタリスク (*) を出力します。

プログラムの例 stars.hs

```
stars :: Int -> String
stars n = replicate n '*'

main :: IO ()
main = do
    putStrLn "enter length:"
    length <- getLine
    putStrLn (stars (read length :: Int))
```

実行例

```
enter length:
40
****************************************
```

このプログラムが、プロンプトを出力するために、putStr ではなくて putStrLn を使っている理由は、putStr を使うと、改行が出力されるまでプロンプトが出力されないという不具合が発生するからです。

改行のないプロンプトを出力する方法については、第 7.5.7 項で説明することにしたいと思います。

7.5　モジュール

7.5.1　モジュールの基礎

処理系の中に取り込んで使うことのできる、処理系の外部にある機能は、「ライブラリー」(library) と呼ばれます。そして、処理系とともに配布されるライブラリーは、「標準ライブラリー」(standard library) と呼ばれます。

Haskell のライブラリーは、「モジュール」(module) と呼ばれるものから構成されています。処理系の中にモジュールを取り込むことを、モジュールを「インポートする」(import) と言います。

7.5.2　Prelude

実は、Haskell の処理系に組み込まれている関数や型構成子や型クラスというのは、最初から処理系に組み込まれているわけではありません。それらは、デフォルトでインポートされるモジュールの中で定義されているのです。Haskell の処理系がデフォルトでインポートするのは、Prelude という名前のモジュールです。

7.5.3　import 文

モジュールをインポートしたいときは、「import 文」(import statement) と呼ばれるものを書きます。

import 文は、基本的には、

```
import モジュール名
```

と書きます。この中の「モジュール名」というところには、インポートしたいモジュールの名前を書きます。たとえば、

```
import Data.List
```

という import 文を書くことによって、Data.List という名前のモジュールをインポートすることができます。

import 文を GHCi に入力すると、GHCi は、その中で指定されたモジュールを自分の中にインポートします。

```
Prelude> import Data.List
Prelude Data.List>
```

Data.List というモジュールは、リストを処理するさまざまな関数を定義しています。たとえば、このモジュールには、引数として 1 個のリストを受け取って、そのリストから重複する要素を取り除くことによってできるリストを戻り値として返す、nub という関数の定義が含まれています。

```
Prelude Data.List> nub [5, 3, 8, 3, 2, 8, 7, 3]
[5,3,8,2,7]
```

モジュールのインポートを必要とするプログラムを書く場合、そのための import 文は、プログラムの先頭に書く必要があります。2 個以上のモジュールをインポートする場合は、それぞれの import 文を改行で区切る必要があります。

7.5.4　修飾付きのインポート

複数のモジュールをインポートすると、しばしば、それぞれのモジュールの中で、同じ識別子が異なるものに束縛されているために、名前の競合が起きてしまうことがあります。

名前の競合は、モジュールを「修飾付き」(qualified) でインポートすることによって避けることができます。モジュールを修飾付きでインポートしたいときは、 import 文を、

```
import qualified モジュール名
```

と書きます。

モジュールを修飾付きでインポートした場合、そのモジュールの中で定義されたものは、識別子をモジュール名で修飾した名前を使わなければ指定することができません。名前にモジュール名が含まれるということは、たとえ別のモジュールで同じ識別子が使われていたとしても、名前の競合が起きる心配はないということです。

識別子をモジュール名で修飾した名前は、

```
モジュール名 . 識別子
```

と書きます。たとえば、

```
import qualified Data.List
```

という import 文で Data.List をインポートした場合、 nub を使うためには、

```
Data.List.nub
```

という名前を書く必要があります。

```
Prelude> import qualified Data.List
Prelude Data.List> Data.List.nub [5, 3, 8, 3, 2, 8, 7, 3]
[5,3,8,2,7]
```

7.5.5 モジュールの別名

モジュールを修飾付きでインポートするとき、そのモジュールに別名 (alias) を与えることも可能です。モジュールに対して、本来の名前よりも短い別名を与えることによって、修飾された名前を短くすることができます。

モジュールに別名を与えたいときは、

```
import qualified モジュール名 as 別名
```

と書きます。この中の「別名」のところには、何らかの構成子識別子を書きます。そうすると、その構成子識別子がモジュールの別名になります。

```
Prelude> import qualified Data.List as L
Prelude L> L.nub [5, 3, 8, 3, 2, 8, 7, 3]
[5,3,8,2,7]
```

7.5.6 コマンドライン引数

コンパイルされたプログラムは、それを起動するコマンドに含まれている引数を受け取ることができます。コマンドに含まれている引数は、「コマンドライン引数」(command line argument) と呼ばれます。

プログラムの中でコマンドライン引数を扱いたいときは、

```
System.Environment
```

というモジュールの中で定義されている、 getArgs という I/O アクションを使います。

getArgs は、 IO [String] という型を持つ I/O アクションです。この I/O アクションを実行すると、その結果として、コマンドライン引数を要素とするリストが得られます。

次のプログラムは、すべてのコマンドライン引数を出力します。

プログラムの例 getargs.hs

```
import System.Environment

main :: IO ()
main = do
    args <- getArgs
    putStrLn (unwords args)
```

実行例

```
$ ./getargs namako umiushi kurage hitode kamenote
namako umiushi kurage hitode kamenote
```

このプログラムの中で使われている unwords という組み込み関数は、引数として文字列のリストを受け取って、そのすべての要素を空白で区切って連結することによってできる文字列を戻り値として返します。

7.5.7 バッファのフラッシュ

入出力されるデータというのは、実際の入力元から直接読み込まれたり、実際の出力先に直接書き込まれたりすることもありますが、「バッファ」(buffer) と呼ばれる中間的な場所を経由することもあります。

第 7.4.6 項で紹介した、標準入力から読み込んだ個数だけアスタリスク (*) を出力するプログラムは、putStrLn を使ってプロンプトを出力していました。その理由は、putStr を使ってプロンプトを出力すると、バッファに書き込まれたプロンプトが、改行が出力されるまで標準出力に書き込まれないからです。

バッファに書き込まれた文字列を強制的に出力先に書き込むことを、バッファを「フラッシュする」(flush) と言います。

バッファをフラッシュしたいときは、System.IO という名前のモジュールの中で定義されている、hFlush という関数を使います。

hFlush は、「ハンドル」(handle) と呼ばれるものを引数として受け取って、「バッファをフラッシュする」という動作をあらわす I/O アクションを戻り値として返します。ハンドルというのは、プログラムの中でファイルを扱うためのデータのことです。

標準出力のバッファをフラッシュするためには、標準出力のハンドルを求める必要があります。標準出力のハンドルは、System.IO の中で定義されている、stdout という識別子を評価することによって求めることができます。

次のプログラムは、第 7.4.6 項で紹介した、標準入力から読み込んだ個数だけアスタリスク (*) を出力するプログラムを、改行のないプロンプトを出力するように改良したものです。

プログラムの例 stars2.hs

```
import System.IO

stars :: Int -> String
stars n = replicate n '*'

main :: IO ()
main = do
    putStr "enter length: "
    hFlush stdout
    length <- getLine
    putStrLn (stars (read length :: Int))
```

実行例

```
enter length: 40
****************************************
```

7.6 ファイルに対する読み書き

7.6.1 readFile

この節では、ファイルに対して読み込みや書き込みをする方法について説明したいと思います。

ファイルからデータを読み込みたいときは、readFile という組み込み関数を使います。

readFile は、引数としてパス名を受け取って、「そのパス名で指定されたファイルの内容を結果として生成する」という動作をあらわす I/O アクションを戻り値として返します。

次のプログラムは、コマンドライン引数で指定されたテキストファイルの内容を標準出力に出力します。

プログラムの例 readfile.hs

```
import System.Environment
```

```
main :: IO ()
main = do
    args <- getArgs
    case args of
        [path] -> do
            contents <- readFile path
            putStr contents
        _ -> putStrLn "Usage: readfile pathname"
```

テキストファイルの例 puellae.txt

```
Kaname Madoka
Akemi Homura
Miki Sayaka
Tomoe Mami
```

実行例

```
$ ./readfile puellae.txt
Kaname Madoka
Akemi Homura
Miki Sayaka
Tomoe Mami
```

7.6.2 writeFile

ファイルにデータを書き込みたいときは、writeFile という組み込み関数を使います。

writeFile は、1 個目の引数としてパス名、2 個目の引数として文字列を受け取って、「そのパス名で指定されたファイルにその文字列を書き込む」という動作をあらわす I/O アクションを戻り値として返します。

次のプログラムは、1 個目のコマンドライン引数で指定されたファイルに、2 個目のコマンドライン引数と改行を書き込みます。

プログラムの例 writefile.hs

```
import System.Environment

main :: IO ()
main = do
    args <- getArgs
    case args of
        [path, s] -> writeFile path (s ++ "\n")
        _         -> putStrLn "Usage: writefile pathname word"
```

実行例

```
$ ./writefile hoge.txt namako
$ cat hoge.txt
namako
```

7.6.3 appendFile

ファイルにデータを書き込むための組み込み関数としては、appendFile というものもあります。

appendFile という関数は、writeFile と同じように、1 個目の引数としてパス名、2 個目の引数として文字列を受け取って、「そのパス名で指定されたファイルにその文字列を書き込む」という動作をする I/O アクションを戻り値として返します。

writeFile と appendFile との相違点は、指定されたファイルがすでに存在していた場合にどうするかというところにあります。writeFile は、そのファイルの内容を削除したのちに文字列を書き込むのですが、appendFile は、そのファイルの内容を削除しないで、文字列をその末尾に追加します。

次のプログラムは、1 個目のコマンドライン引数で指定されたファイルに、2 個目のコマンドライン引数と改行を追加します。

プログラムの例 appendfile.hs

```
import System.Environment

main :: IO ()
main = do
    args <- getArgs
    case args of
        [path, s] -> appendFile path (s ++ "\n")
        _         -> putStrLn "Usage: appendfile pathname word"
```

実行例

```
$ ./appendfile moge.txt umiushi
$ ./appendfile moge.txt kurage
$ ./appendfile moge.txt hitode
$ cat moge.txt
umiushi
kurage
hitode
```

7.6.4 文字単位のファイル処理

文字列というのは文字を要素とするリストですから、ファイルから読み込んだ文字列をリストとして処理することによって、ファイルの内容を文字単位で処理することができます。

次のプログラムは、1 個目のコマンドライン引数で指定されたファイルから読み込んだすべての文字を大文字に変換した結果を標準出力に出力します。

プログラムの例 toupper.hs

```
import System.Environment
import Data.Char

main :: IO ()
main = do
    args <- getArgs
    case args of
        [path] -> do
            contents <- readFile path
            putStr (map toUpper contents)
        _ -> putStrLn "Usage: toupper pathname"
```

実行例

```
$ ./toupper puellae.txt
KANAME MADOKA
AKEMI HOMURA
MIKI SAYAKA
TOMOE MAMI
```

このプログラムは、toUpper という関数を使っています。この関数は、引数として文字を受け取って、それが英字の小文字ならばそれを大文字に変換したものを戻り値として返して、それが英字の小文字ではないならばそれをそのまま返します。

toUpper は、Data.Char というモジュールの中で定義されている関数です。このモジュールは、toUpper のほかにも、文字を扱うたくさんの便利な関数を定義しています。

7.6.5 単語単位のファイル処理

ファイルの内容を単語単位で処理したいときは、words と unwords という関数を使います。

words は、引数として文字列を受け取って、その文字列を、空白を区切り文字として個々の単語に分解して、それらの単語を要素とするリストを戻り値として返します。

unwords は、第 7.5.6 項で紹介したように、引数として文字列のリストを受け取って、そのすべての要素を空白で区切って連結することによってできる文字列を戻り値として返します。

次のプログラムは、1 個目のコマンドライン引数で指定されたファイルから読み込んだすべての単語の先頭の文字を空白で区切って標準出力に出力します。

プログラムの例 initial.hs

```
import System.Environment

initial :: String -> String
initial = unwords . (map (take 1)) . words

main :: IO ()
main = do
    args <- getArgs
    case args of
        [path] -> do
            contents <- readFile path
            putStrLn (initial contents)
        _ -> putStrLn "Usage: initial pathname"
```

実行例

```
$ ./initial puellae.txt
K M A H M S T M
```

7.6.6 行単位のファイル処理

ファイルの内容を行単位で処理したいときは、linesとunlinesという関数を使います。

linesは、引数として文字列を受け取って、その文字列を改行で個々の行に分解して、それらの行を要素とするリストを戻り値として返します。

unlinesは、引数として文字列のリストを受け取って、そのすべての要素を改行で区切って連結することによってできる文字列を戻り値として返します。

次のプログラムは、1個目のコマンドライン引数で指定されたファイルから読み込んだそれぞれの行の先頭に行番号を追加したものを標準出力に出力します。

プログラムの例 linenumber.hs

```
import System.Environment

number :: [String] -> [(Int, String)]
number xs = zip [1..(length xs)] xs

addNumber :: (Int, String) -> String
addNumber (n, s) = (show n) ++ ": " ++ s

addNumbers :: [String] -> [String]
addNumbers = (map addNumber) . number

lineNumber :: String -> String
lineNumber = unlines . addNumbers . lines

main :: IO ()
main = do
    args <- getArgs
    case args of
        [path] -> do
            contents <- readFile path
            putStr (lineNumber contents)
        _ -> putStrLn "Usage: linenumber pathname"
```

実行例

```
$ ./linenumber puellae.txt
1: Kaname Madoka
2: Akemi Homura
3: Miki Sayaka
4: Tomoe Mami
```

7.7　入出力のための便利な関数

7.7.1　この節について

Haskell の処理系がデフォルトでインポートする Prelude や、Control.Monad というモジュールは、入出力をするプログラムを書くときに使うと便利な関数を定義しています。この節では、そのような関数をいくつか紹介したいと思います。

7.7.2　print

print は、

```
putStrLn . show
```

という関数と同じ動作をする関数です。つまり、Show のインスタンスのデータを引数として受け取って、「それを文字列に変換したものと改行を標準出力に出力する」という動作をあらわす I/O アクションを戻り値として返す関数だということです。

次のプログラムの中で定義されている tenTimes という関数は、整数のリストを標準入力から読み込んで、そのすべての要素を 10 倍することによってできるリストを標準出力に出力します。

プログラムの例　tentimes.hs

```
tenTimes :: IO ()
tenTimes = do
    putStr "enter list of integers: "
    xs <- getLine
    print (map (*10) (read xs))
```

実行例

```
*Main> tenTimes
enter list of integers: [3, 7, 2, 4, 8]
[30,70,20,40,80]
```

7.7.3　sequence

sequence は、I/O アクションのリストを引数として受け取って、「その要素を順番に実行して、それぞれの I/O アクションの結果から構成されるリストを結果として生成する」という動作をあらわす I/O アクションを戻り値として返す関数です。

```
Prelude> sequence [getLine, getLine, getLine]
namako
umiushi
kurage
["namako","umiushi","kurage"]
```

7.7.4　mapM と mapM_

mapM は、1 個目の引数として I/O アクションを戻り値として返す関数、2 個目の引数としてリストを受け取って、「2 個目の引数を構成しているそれぞれの要素に対して 1 個目の引数を適用することによって得られた I/O アクションを実行して、それらの結果から構成されるリストを結果として生成する」という動作をあらわす I/O アクションを戻り値として返す関数です。

```
Prelude> mapM print [3, 7, 2]
3
7
2
[(),(),()]
```

mapM_ も、mapM とほとんど同じ動作をする関数です。それらの相違点は、戻り値として返す I/O アクションの結果だけです。mapM が返す I/O アクションは結果としてそれぞれの I/O アクションの結果から構成されるリストを生成しますが、mapM_ が返す I/O アクションは結果としてユニットを生成します。ですから、それぞれの I/O アクションの結果が必要な場合は mapM、必要ではない場合は mapM_ を使うようにするといいでしょう。

次のプログラムは、コマンドライン引数で指定されたすべてのファイルの内容を標準出力に出力します。

プログラムの例 cat.hs

```
import System.Environment

cat :: String -> IO ()
cat path = do
    contents <- readFile path
    putStr contents

main :: IO ()
main = do
    args <- getArgs
    mapM_ cat args
```

7.7.5 forM と forM_

forM と forM_ は、Control.Monad というモジュールの中で定義されている関数です。

forM の動作は、mapM と同じです。ただし、mapM と forM とでは、引数の順番が違います。mapM は 1 個目が関数で 2 個目がリストですが、forM は 1 個目がリストで 2 個目が関数です。

引数として渡す関数に名前が与えられている場合は mapM を使えばいいのですが、長いラムダ式の値を引数として渡す場合、mapM を使うと、その長いラムダ式のさらにうしろに 2 個目の引数を書くことになりますので、プログラムが読みにくくなってしまいます。そのような場合は、mapM の代わりに forM を使って、引数の順番を逆にするほうがベターです。

forM_ も、引数の順番が逆になっているという点を除けば、mapM_ と同じ動作をします。

次のプログラムは、第 7.7.4 項で紹介した、コマンドライン引数で指定されたすべてのファイルの内容を標準出力に出力するプログラムを、forM_ を使って書き直したものです。

プログラムの例 cat2.hs

```
import System.Environment
import Control.Monad

main :: IO ()
main = do
    args <- getArgs
    forM_ args $ \path -> do
        contents <- readFile path
        putStr contents
```

7.7.6 when

when も、Control.Monad の中で定義されている関数です。

when は、1 個目の引数として真偽値、2 個目の引数としてユニットを生成する I/O アクションを受け取って、1 個目の引数が真ならば 2 個目の引数をそのまま返して、偽ならば何もしない I/O アクションを返します。

when は、何らかの条件が成り立っているときだけ何らかの動作をする I/O アクションを作って、条件が成り立っていないときは何もしない I/O アクションを作りたい、というときに便利な関数です。

次のプログラムは、第 7.2.4 項で紹介した doOrDoNot という関数の定義を、when を使って書き直したものです。

プログラムの例 doordonot2.hs

```
import Control.Monad

doOrDoNot :: Bool -> IO ()
doOrDoNot b = when b (putStrLn "Do, or do not. There is no try.")
```

参考文献

[Lipovača,2011] Miran Lipovača, *Learn You a Haskell for Great Good!*, No Starch Press, 2011, ISBN 978-1-59327-283-8. 邦訳（田中英行、村主崇行）、『すごい Haskell たのしく学ぼう！』、

オーム社、2012、ISBN 978-274-06885-0。

[Thompson,2011] Simon Thompson, *Haskell: The Craft of Functional Programming, Third Edition*, Addison-Wesley Professional, 2011, ISBN 978-0-201-88295-7.

[大川,2014] 大川徳之、『関数プログラミング実践入門：簡潔で、正しいコードを書くために』、技術評論社、2014、ISBN 978-4-7741-6926-2。

[向井,2006] 向井淳、『入門 Haskell：はじめて学ぶ関数型言語』、毎日コミュニケーションズ、2006、ISBN 978-4-8399-1962-7。

索引