すごいHaskell たのしく学ぼう！

Miran Lipovača ● 著 ／ 田中英行・村主崇行 ● 共訳

no starch press
Ohmsha

Title of English-language original: Learn you a Haskell for Great Good!
ISBN 978-1-59327-283-8, published by No Starch Press, Inc.

訳者序文

初めて本書の原著（の元となった Web ページ版）を読んだときは、驚きでした。プログラミングの入門というものは、往々にして次のいずれかになりがちです。教科書めいた堅苦しい文章になるか、はたまた細かい部分を誤魔化して分かったような分からないようなものになるか。本書はいかにも後者であるかのような雰囲気でした。すなわち、まず “Learn You a Haskell for Great Good!” というよく意味の分からないタイトルに始まり、全編にわたって豊富に挿入されたファンシーなイラスト、ちりばめられた微妙に分かりづらいギャグやパロディ、いかにも初心者向け書籍のテンプレートを踏襲しているかのように見えました。しかし、読み進めるうちに、これはもしかしたらそうではない、もっとすごい本なのではないかという思いを抱き始め、読み終わるころにはそれは確信へと変わっていました。愉快なイラストに軽妙な語り口、強烈に初学者を意識した体裁とは裏腹に、しっかりと Haskell のキモが押さえられた内容になっており、難しさを感じさせないままのらりくらりとラストまで突き進んでしまいます。誤解を招くような書き方かもしれませんが、易しく書かれている割にはしっかりしている、というわけではありません。入門書の括りとして見ても、この上ない完成度なのです。ここまで Haskell でのプログラミング作法、あるいはベストプラクティスがしっかりと語られた書籍は、少なくとも日本語で読めるものでは見当たりません。特にファンクター、アプリカティブファンクター、それとモナドとの関係、非常に充実したモナドの解説、さらには Zipper にまで触れられているところは特筆に値します。Haskell に対する何やらよく分からない、難しそうだというイメージが払拭されるのではないか。読んでいるときから、これは日本語でも読めるようになるべきだと強く思っていました。その仕事を、ほかでもない自分の手によって行えたことはこの上のない幸せです。

本書はすべての Haskell 入門者に読んでいただきたい本です。あるいは、過去に Haskell を勉強しようとしたけれども挫折した、はたまた他の入門書をひととおり読んではみたものの今ひとつよく分からない、モナドが使いこなせないなどの方にもおすすめいたします。「たのしく」読み進めているうちに、いつしか Haskell の「すごい」ところが習得できていることでしょう！

最後に、翻訳にあたってお世話になった方々に感謝申し上げます。原著者との交渉、組版、校正などの多くの作業、そして遅々（延々）として進まない我々の翻訳作業を根気よく見守っていただいたオーム社の鹿野さん、並びに数多くの有益なフィードバックをいただいたレビュアーの、山本和彦さん、鈴木浩一さん、豊福親信さん、尾崎竜史さん、山下伸夫さん、石井大海さん、小久保祐介

さん、島崎清山さん、木戸崇裕さん、九岡佑介さん、竹田光孝さん、藤村大介さん、fetastein さん、情野吉紀さん、松井楽徳さん、樋村隆弘さん、川又龍一さん、@HIROCASTER さん、西村瑞希さんに改めて感謝申し上げます。

2012 年 5 月

訳者しるす

Haskell はおもしろい。以上！

この本は命令型プログラミング言語（C++ や Java、Python など）でのプログラミング経験者で、今 Haskell を勉強してみたいという人を対象にしています。でも、大げさなプログラミング経験がなくても　（この本を手に取ってくれた）あなたのような賢い人はきっとこの本を理解して Haskell を習得できることでしょう。

Haskell に対する僕の最初の印象は、何だかとても変わった言語だなというものでした。でも最初のハードルを越えた後は、スムーズに頭に入ってきました。Haskell の第一印象が奇妙なものだったとしても、あきらめないでください。Haskell を学ぶのは、もう一度はじめからプログラミングを始めるようなものです。楽しくて、今までとは違った考え方をさせてくれます。

NOTE もしも本当に行き詰まったら、freenode ネットワークの IRC チャンネル `#haskell` で質問をするとよいでしょう。素敵で寛容な、そして理解力のある人たちが答えてくれます。彼らは Haskell 初心者にとってとても良い情報源です。

で、Haskell って何なの？

Haskell は**純粋関数型**プログラミング言語です。

命令型プログラミング言語では、命令の並びをコンピュータに与えて、それを実行します。命令を実行している間、コンピュータの状態は変更されていきます。例えば、変数 `a` が 5 に設定された後に、どこか別のコードが `a` の値を変更する可能性があります。また、`for` ループや `while` ループのような繰り返し実行のための構文があります。

純粋関数型プログラミングは違います。コンピュータに**何をするか**は伝えません。**何であるか**を伝えるのです。例えば、「ある数の階乗とは 1 からその数までの積である」というような定義をコンピュータに伝えます。あるいは、「数のリストの和とは先頭の数と残りのリストの和を足し合わせたものである」などと。これらはどちらも**関数**として書くことができます。

関数型プログラミングでは、一度変数の値を設定すると、後でそれを別の値に変更することは**できません**。`a` が 5 だと言ったのなら、心変わりしてやっぱり

別の値です、と言うことはできないのです。だってあなたは a は 5 だって言ったじゃないですか（あなたは嘘をつくような人ですか？）。

純粋関数型言語では、関数は**副作用**を持ちません。関数にできることは、何かを計算してその結果を返すことだけです。これは最初のうちは制約に思うかもしれませんが、実はとてもうれしい効能があります。 関数が同じ引数で 2 回呼ばれたら、これらは同じ値を返すことが保証されます。この性質は**参照透明性**と呼ばれています。そのおかげで、プログラマは関数の正しさを簡単に推測（時には証明さえ）できます。正しいと分かっている単純な関数を組み合わせて、より複雑な正しい関数を組み立てられるのです。

Haskell は**怠け者**です。特にそうしろと言われた場合を除いては、結果が必要になるまで関数を実行しないのです。この振る舞いは、専門用語では**遅延評価**といいます。これは、Haskell が参照透明だから許されることです。関数の結果が与えられた引数だけに依存しているのなら、それがいつ計算されるかを気にする必要はありません。怠け者言語 Haskell はこの事実を利用して、計算をそれが必要になるギリギリまで引き伸ばします。いよいよ結果を表示しなければならないとなると、Haskell は要求されたものを表示するために必要な最小限の計算を行います。遅延評価は、見かけ上無限の大きさのデータを扱うことを可能にします。実際に画面に表示するデータのみが計算されるからです。

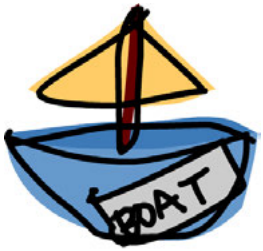
Haskell の遅延性の例を見てみましょう。数のリスト `xs = [1,2,3,4,5,6,7,8]` と、すべての要素を 2 倍にした新しいリストを返す関数 `doubleMe` があります。リストの要素を 8 倍したいなら、次のようなコードでできます。

```
doubleMe(doubleMe(doubleMe(xs)))
```

命令型言語では、まずリストを関数に渡し、コピーを作ってからそれを返すでしょう。それからさらにもう 2 回、リストを関数に渡し、コピーを作り、結果を返すというのを繰り返すでしょう。

遅延評価な言語では、リストに対して `doubleMe` を呼び出しても、結果を表示するようにとの要求がなければ、プログラムは「分かってるよ、後でやるよ！」と言うだけで実際には何もしません。いざ、結果を見せるように言われると、そ

こで初めて 1 番目の doubleMe が 2 番目の doubleMe を呼び出して、すぐに結果をくれと言います。すると 2 番目の doubleMe が同じことを 3 番目の doubleMe に対して行い、3 番目の doubleMe はしぶしぶ 1 を 2 倍したもの、つまり 2 を返します。2 番目の doubleMe はそれを受け取り、1 番目の doubleMe に 4 を返します。1 番目の doubleMe はこの結果を 2 倍して、最終的な結果として得られるリストにおける最初の要素は 8 ですよと教えてくれます。Haskell の遅延性のおかげで、 doubleMe を 3 重に呼び出しても、リストは、要求があったときにただ一度だけ走査されるのです。



Haskell は**静的型付け**言語です。これはどのコード片が数で、どれが文字列で、といったことをコンパイラがプログラムのコンパイル時にすべて知っているということです。静的型付けは、コンパイル時にたくさんのエラーの要因を捕まえてくれます。例えば、数と文字列を足し合わせようとしたら、コンパイラは、そんなことできないよって泣きごとを言うでしょう。

Haskell は**型推論**を持つとても優れた型システムを採用しています。プログラマがすべてのコード片に対して明示的に型を書かなくても、Haskell の賢い型システムは自分でそれを推論できるのです。例えば `a = 5 + 4` というコードがあったとします。ここであなたは a が数であると Haskell に教えてあげる必要はありません——コンパイラは型を自力で導き出すことができます。型推論のおかげで、汎用性の高いコードを書くのが簡単になります。 2 つの引数をとり足し算する関数を書いて、 それらの型を明示的に指定しないでおくと、数のように振る舞う任意の引数に対して動作する関数の出来上がりです。

Haskell は**エレガントで簡潔**です。高度な概念をふんだんに用いているので、通常 Haskell のプログラムは、命令型で書かれた同等のものと比べて短くなります。短いプログラムは保守しやすく、含まれるバグも少なくなります。

Haskell は本当に頭の切れる（博士号を持つ）人たちによって作られています。Haskell は、最高の言語を設計するための、研究者によって構成された委員会により、1987 年から開発されています。Haskell 言語の仕様の最新の安定版（The Haskell Report）は 2010 年に出版されました。

Haskellの世界に飛び込むのに必要なもの

Haskellを始めるのに必要なのはテキストエディタとHaskellコンパイラだけです。テキストエディタはお気に入りのものを使ってください。Haskellのコンパイラで最も広く使われているのはThe Glasgow Haskell Compiler（GHC）です。この本でもGHCを利用します。

必要なものを手っ取り早く揃えるには、**Haskell Platform**をダウンロードするのがベストです。Haskell PlatformにはGHCコンパイラだけでなく、便利なHaskellのライブラリのセットも同梱されています！ Haskell Platformを手に入れるには、http://hackage.haskell.org/platform/ に行って、利用しているOS向けの指示に従ってください。

GHCはHaskellのソースコード（通常は.hsという拡張子）をコンパイルできるのに加えて、対話モードも備えています。対話環境でスクリプトに定義された関数をロードして、それを呼び出し、結果を直接見ることができます。特に学習時には、スクリプトを変更するたびにコンパイルするより、対話モードを利用するのがお手軽です。

Haskell Platformをインストールしたなら、（LinuxかMac OS Xなら）ターミナルウインドウを開きましょう。Windowsの場合はコマンドプロンプトかPowerShellを開きます。それから**ghci**とタイプして、ENTERを押すと、対話モードが起動します（GHCiプログラムが見つからない場合は、いったん再起動してみてください）。対話モードから抜け、ターミナルに戻るには、:quit（または:q）とタイプして、ENTERを押してください。

myfunctions.hsというスクリプトに関数を定義したならば、:l myfunctionsとタイプすることで、それをGHCiにロードできます（myfunctions.hsはGHCiを起動したフォルダと同じ場所に置いてください）。

.hsスクリプトを変更した場合は、:l myfunctionsを実行して同じファイルをロードし直すか、:rを実行して現在のスクリプトをリロードします[†1]。僕の定番のワークフローは、関数を.hsファイルに定義して、GHCiにそれをロードしていじくり回し、ファイルを変更する、という繰り返しです。この本でもこのやり方を使っていきます。

謝辞

修正、提案、励ましの言葉を送ってくださった皆さんに感謝します。僕を本物の物書きのようにしてくれたKeith、Sam、Marilynにも感謝します。

[†1] ［訳注］コロン（:）で始まる行はGHCiコマンドと呼ばれ、GHCiに入力すると特別なことが起こります。GHCiコマンドの一覧を調べるには:?と入力してください。なおコマンドは全部打たずに略してもよく、複数のコマンドが該当する場合はよく使われるほうのコマンドが実行されます。

目次

第1章
はじめの第一歩

もしあなたが、イントロダクションなんて読まないというとんでもない考えの持ち主なら、今すぐ1つ前の章に戻って読んできてください。この本の使い方や、GHC に関数をロードする方法が書いてあります。

最初に、GHC の対話モードを起動して関数をいくつか呼び出し、Haskell の基本的な感覚を味わってみましょう。端末を開いて **ghci** とタイプします。次のような挨拶が表示されるでしょう[†1]。

```
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude>
```

NOTE GHCi のデフォルトのプロンプトは `Prelude>` ですが、この本の例では `ghci>` と表記します。実際の GHCi のプロンプトをこれに合わせるには、GHCi 上で **`:set prompt "ghci> "`** とタイプします。GHCi を起動するたびにタイプするのが面倒なら、ホームディレクトリに `.ghci` というファイルを作って、その中に **`:set prompt "ghci> "`** と書き込みます。

おめでとう、あなたは今 GHCi の中にいます！ それでは、簡単な計算をやってみましょうか。

```
ghci> 2 + 15
17
ghci> 49 * 100
4900
ghci> 1892 - 1472
420
```

[†1] ［訳注］原著文中での GHC のバージョンは 6.12.3 だったようですが、翻訳作業時点での Haskell Platform の最新版（2011.4.0.0）に同梱されている GHC のバージョンは 7.0.4 です。このバージョンの違いが実行結果に違いをもたらす箇所には、なるべく訳注を付けてます。GHC の開発速度には目を見張るものがあります。ぜひ最新版を入れてください！

```
ghci> 5 / 2
2.5
ghci>
```

1 つの式の中に複数の演算子がある場合、Haskell は演算子の優先順位に従って実行します。例えば、「`*`」は「`-`」よりも高い優先順位を持っているので、「`50 * 100 - 4999`」は「`(50 * 100) - 4999`」として解釈されます。

次のように、括弧を使って演算の順序を明示することもできます。

```
ghci> (50 * 100) - 4999
1
ghci> 50 * 100 - 4999
1
ghci> 50 * (100 - 4999)
-244950
```

うわーすげえかっこいい（棒読み）――うん、分かってる。すぐに本当にかっこいい例が出てくるから、今はちょっと我慢してね。

注意すべき落とし穴が 1 つあります。それは負の数の扱いです。式の途中に負の数が出てくるときは、必ず括弧で囲むようにしましょう。例えば、`5 * -3` を GHCi に入力すると文句を言ってきますが、`5 * (-3)` なら正しく動いてくれます。

ブール代数も Haskell では思いのままです。他の多くのプログラミング言語と同様に、Haskell にも真理値 `True` と `False`、それから論理積のための演算子 `&&` と論理和のための演算子 `||`、さらに `True` と `False` を反転させる論理否定演算子 `not` があります。

```
ghci> True && False
False
ghci> True && True
True
ghci> False || True
True
ghci> not False
True
ghci> not (True && True)
False
```

2 つの値が等しいか等しくないかは、それぞれ演算子 `==` と `/=` でテストできます。

```
ghci> 5 == 5
True
ghci> 1 == 0
False
```

```
ghci> 5 /= 5
False
ghci> 5 /= 4
True
ghci> "hello" == "hello"
True
```

でも！ 値の組み合わせには注意してください。例えば、`5 + "llama"` のような式を入力すると、次のようなエラーメッセージが返ってきます。

```
No instance for (Num [Char])
arising from a use of ‘+’ at <interactive>:1:0-9
Possible fix: add an instance declaration for (Num [Char])
In the expression: 5 + "llama"
In the definition of ‘it’: it = 5 + "llama"
```

ここで GHCi が言っているのは「`"llama"` は数じゃないよ、だからどうやって 5 と足し合わせればいいか分からないよ」というようなことです。`+` 演算子は両辺に数がくることを期待しているのです。

これに対して `==` 演算子は、比較可能なあらゆる型に対して動作します。ただし、両辺の値は同じ型である必要があります。例えば、`True == 5` のような式を入力すると GHCi は困ってしまいます。

NOTE `5 + 4.0` は正しい式です。`4.0` は整数ではないので、一見型が合わないように見えますが、`5` は密かにすごいやつで、浮動小数型としても整数型としても振る舞えるのです。今回の場合 `5` は、相手の浮動小数値 `4.0` に合わせた型になります。

型については、後でもう少し詳しく説明します。

1.1 関数呼び出し

気づかなかったかもしれませんが、ここまでの説明ですでに関数を使っています。例えば、`*` は 2 つの数を引数に取り、それらを掛け合わせる関数です。関数 `*` を適用する（あるいは**呼び出す**）には、掛け合わせたい 2 つの数で挟みます。このような関数は**中置**関数と呼ばれます。

でも、ほとんどの関数は**前置**関数です。Haskell で前置関数を呼び出すには、関数名を最初に書いて、それからスペース、続けて（スペースで区切られた）引数を書きます。例として、とてつもなく退屈な Haskell の関数 `succ` を呼び出してみます。

```
ghci> succ 8
9
```

succ 関数は、「後者（successor）」が明確に定義されているものを何でもいいから引数として受け取り、その値を返します。整数に対する「後者」は、単に「次に大きい数」です。

では次に、複数の引数を取る前置関数 min と max を呼び出してみましょう。

```
ghci> min 9 10
9
ghci> min 3.4 3.2
3.2
ghci> max 100 101
101
```

min と max 関数はどちらも 2 引数の関数で、何らかの順序の付いたもの（数とかね！）を受け取り、それぞれ小さいほう、または大きいほうを返します。

関数の適用は、すべての演算の中で最も高い優先度を持ちます。つまり、次の 2 つの式は等価です。

```
ghci> succ 9 + max 5 4 + 1
16
ghci> (succ 9) + (max 5 4) + 1
16
```

また、次のように書いても 9 * 10 の後者は得られません。

```
ghci> succ 9 * 10
```

演算の優先順位があるので、これは 9 の後者（つまり 10）に 10 を掛けたものとして評価されます。すなわち 100 が得られます。望みの値を得るには、次のように入力する必要があります。

```
ghci> succ (9 * 10)
```

これは 91 を返します。

関数が 2 引数のときは、その関数をバッククオート（ ` ）で囲むことで中置関数として呼び出せます[†2]。例えば、div 関数は 2 つの整数を引数に取り整数の除算を行う関数です。

```
ghci> div 92 10
9
```

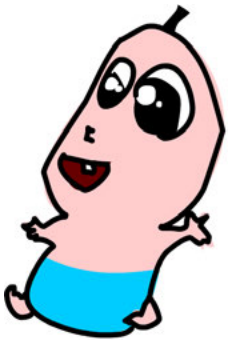
でもこのような呼び出し方では、どちらの数がどちらの数で割られるのか混乱する場合があるかもしれません。バッククオートを使うと中置関数として呼び出せるようになり、あっという間に分かりやすくなります。

[†2] ［訳注］バッククオート（ ` ）はクオート（ ' ）とは違う文字ですよ！ 日本語キーボードなら「SHIFT-@」で入力できるはずです。

```
ghci> 92 `div` 10
9
```

命令型言語に慣れているプログラマの中には、関数の適用を括弧で表すやり方に固執してしまって Haskell 流のやり方に慣れるのに苦労している人も多くいます。bar (bar 3) のようなものを見たら、これは最初に bar 関数を 3 を引数として呼び出し、それからその結果を bar 関数に再度渡している、と読めるようにしてください。これと等価な C の式は bar(bar(3)) です。

1.2 赤ちゃんの最初の関数



関数を定義する構文は、関数呼び出しに似ています。すなわち、関数名の後ろにスペースで区切った引数が続きます。ただし、引数の後ろには = 演算子が続き、関数の本体を表すコードがそれに続きます。

例として、数を 1 つ受け取り、それを 2 倍する単純な関数を書いてみましょう。お好みのエディタを開いて、次のコードをタイプしてください。

```
doubleMe x = x + x
```

このファイルを baby.hs として保存してください。それから baby.hs のあるディレクトリで ghci を実行してください。GHCi が起動したら、**:l baby** とタイプしてファイルをロードします。これで新しい関数とたわむれることができるようになります。

```
ghci> :l baby
[1 of 1] Compiling Main             ( baby.hs, interpreted )
Ok, modules loaded: Main.
ghci> doubleMe 9
18
ghci> doubleMe 8.3
16.6
```

+ は整数だけじゃなく、浮動小数点数（実際には、数とみなせるものなら何でも）に対しても動作するので、さっき定義した関数もそれらの型でも正しく動作します。

今度は 2 つの引数をそれぞれ 2 倍してから足し合わせる関数を作ってみましょう。次のコードを baby.hs に追加してください。

```
doubleUs x y = x * 2 + y * 2
```

NOTE Haskell では関数を決まった順番で定義する必要はありません。なので、コードの追加は baby.hs ファイルの先頭でもかまいません。

ではファイルを保存して、GHCi で **:l baby** とタイプして新しい関数をロードしましょう。関数が期待した値を返すかテストします。

```
ghci> doubleUs 4 9
26
ghci> doubleUs 2.3 34.2
73.0
ghci> doubleUs 28 88 + doubleMe 123
478
```

関数の定義の中で自分で定義した関数を呼び出すこともできます。それを踏まえて、 doubleUs を次のように定義することもできます。

```
doubleUs x y = doubleMe x + doubleMe y
```

これは Haskell を使っているときによく目にするパターン、「基本的で、明らかに正しい関数を組み合わせて、より大きな関数を組み立てる」のとてもシンプルな一例です。このパターンは、コードの繰り返しを避ける素晴らしい方法です。例えば、ある日数学者たちが 2 と 3 が実は等しいと証明したせいでプログラムを書き換える必要に迫られたら？ そんなときでも、ただ doubleMe の定義を x + x + x に書き換えるだけです。 doubleMe を呼び出している doubleUs は 2 と 3 が等しい、新しい奇妙な世界でも正しく動作するはずです。

次は、100 以下のときだけ数を 2 倍するような関数を書いてみましょう（100 より大きい数はもう十分に大きいので！）。

```
doubleSmallNumber x = if x > 100
                      then x
                      else x*2
```

この例には Haskell の **if** 式が出てきます。すでに他の言語で if 式にはお馴染みかもしれませんが、Haskell のユニークなところは **else** 節が必須なところです。

命令型言語のプログラムは本質的にはプログラム実行時にコンピュータが実行するステップの列です。対応する **else** 節のない **if** 文があり、その場合条件式が成立しなければ、**if** 式の中は実行されずに下に抜けます。命令型言語では、**if** 文は何もしないということがあり得るのです。

一方で、Haskell プログラムは関数の集まりです。関数はデータ値を結果の値に変換するのに使われ、すべての関数は何らかの値を返します。そしてその値はまた別の関数によって使われます。だから、すべての関数は何かを返さなければなりません。これは、すべての **if** は対応する **else** を持たなければならないことを意味します。さもなければ、ある条件を満たすときは値を返し、満たさないときは返す値がないというような関数を定義できてしまいます！ 簡単に言うと、Haskell の **if** は必ず値を返す式であって、文ではないのです。

さっきの doubleSmallNumber の返す値に 1 を加えたものを返す関数が欲しいとします。その新しい関数は次のように書けます。

```
doubleSmallNumber' x = (if x > 100 then x else x*2) + 1
```

括弧があることに注意してください。括弧がないと、1 が足されるのは x が 100 以下のときのみになってしまいます。関数名の最後に付いているアポストロフィ（'）にも注目してください。アポストロフィは Haskell の構文では特別な意味を持たない、関数名の一部として有効な文字です。慣習的に ' は、**正格**（遅延じゃない）版の関数を表したり、少し変更したバージョンの関数に似た名前をつけるために利用されます。

' は関数名として有効な文字なので、次のような関数を書くこともできます。

```
conanO'Brien = "It's a-me, Conan O'Brien!"
```

この関数には、注意すべきところが 2 つあります。1 つは、関数名の先頭の Conan を大文字で始めないこと。Haskell では関数を大文字で始められないことになっています（理由は後で見ていきましょう）。もう 1 つは、この関数は引数を何も受け取らないこと。関数が 1 つも値を取らないとき、これを**定義**とか**名前**とか呼びます。名前（もしくは関数）が何を表すかは、一度定義したら変更できないので、関数 conanO'Brien と文字列 "It's a-me, Conan O'Brien!" は互いに交換できます。

1.3 リスト入門

Haskell のリストは**一様**なデータ構造です。つまり、同じ型の要素を複数個格納できます。整数のリスト、あるいは文字のリスト、といったものを作ることはできますが、整数と文字の両方からなるリストを作ることはできません。

リストは要素をカンマ区切りで並べて、角括弧で括ったものです。

```
ghci> let lostNumbers = [4,8,15,16,23,42]
ghci> lostNumbers
[4,8,15,16,23,42]
```

NOTE GHCi の中で名前を定義するときは let キーワードを使ってください。GHCi で let a = 1 と入力するのは、スクリプトに a = 1 と書いて :l でロードするのと等価です。

連結

リストの操作の中でも最も一般的なのは、連結操作です。Haskell では ++ 演算子を用いて行います。

```
ghci> [1,2,3,4] ++ [9,10,11,12]
[1,2,3,4,9,10,11,12]
ghci> "hello" ++ " " ++ "world"
"hello world"
ghci> ['w','o'] ++ ['o','t']
"woot"
```

NOTE Haskell では、文字列は文字のリストとして表されています。例えば、文字列 `"hello"` は実際にはリスト `['h','e','l','l','o']` と同じです。このためリスト関数を文字列に用いることができ、とても便利です。

長い文字列に対して繰り返し ++ を使うときには注意が必要です。2 つのリストを連結するとき、Haskell は 1 つ目のリスト（ ++ の左側）を最後まで走査します。小さいリストを扱うときは問題になりませんが、例えば 5000 万要素のリストの最後に何かを追加するという操作には少々時間がかかるでしょう。

しかし、リストの先頭に何かを追加するのは、ほとんど一瞬で終わる軽い操作です。これには : 演算子（cons 演算子とも呼ばれます）を使います。

```
ghci> 'A':" SMALL CAT"
"A SMALL CAT"
ghci> 5:[1,2,3,4,5]
[5,1,2,3,4,5]
```

最初の例で、 : が文字と文字のリスト（文字列）を引数に取っていることに注目してください。2 つ目の例では同様に、 : は数と数のリストを取っています。 : 演算子の第一引数は、常に追加しようとしているリストの要素の型と同じ型の単一の要素です。

これに対し、 ++ 演算子は必ず 2 つのリストを引数として受け取ります。 ++ を使ってリストの最後に 1 つだけ要素を追加したい場合にも、Haskell が値をリストとして扱うように角括弧で囲む必要があります。

```
ghci> [1,2,3,4] ++ [5]
[1,2,3,4,5]
```

`[1,2,3,4] ++ 5` と書くのは間違いです。なぜなら、 ++ の引数は 2 つともリストでなければならなくて、 5 はリストではなく数だからです。

面白いことに、Haskell では `[1,2,3]` は `1:2:3:[]` の単なる構文糖衣です。`[]` は空のリストです。その先頭に 3 を追加すると `[3]` になります。そこにさらに 2 を先頭に追加すると、 `[2,3]` になります。

NOTE [] 、 [[]] 、 [[],[],[]] はそれぞれ違うものです。1 つ目は空リストで、2 つ目は 1 つの空リストを含むリストで、3 つ目は 3 つの空リストを含むリストです。

リストの要素へのアクセス

リストの要素を先頭からの位置で取得したいときには !! 演算子を用います。多くのプログラミング言語と同じく添字は 0 から数えます。

```
ghci> "Steve Buscemi" !! 6
'B'
ghci> [9.4,33.2,96.2,11.2,23.25] !! 1
33.2
```

5 個しか要素のないリストの 6 番目の要素を取り出そうとするとエラーになるので注意してください！

リスト中のリスト

リストはリストを要素として含むことができて、リストはリストを含むリストを含むことができて、……

```
ghci> let b = [[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b ++ [[1,1,1,1]]
[[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3],[1,1,1,1]]
ghci> [6,6,6]:b
[[6,6,6],[1,2,3,4],[5,3,3,3],[1,2,2,3,4],[1,2,3]]
ghci> b !! 2
[1,2,2,3,4]
```

リスト中のリストは、それぞれ違う長さでもかまいませんが、違う型は許されません。文字と数を要素として持つリストを作れないのと同じように、文字のリストと数のリストを要素として持つリストは作れません。

リストの比較

中の要素が比較可能であればリストも比較可能です。< 、<= 、>= 、> を使って 2 つのリストを比較すると、辞書順で比較されます。まず 2 つのリストの先頭の要素が比較され、それらが等しければ 2 番目の要素どうしが比較されます。2 番目の要素も等しければ 3 番目の要素が比較され、違う要素が見つかるまでこれが繰り返されます。2 つのリストの順序は、最初に見つかった異なる要素の順序で決まります。

例えば `[3,4,2] < [3,4,3]` を評価すると、Haskell はまず `3` と `3` が同じであることを見て、`4` と `4` を比較します。これら 2 つも同じなので、`2` と `3` を比較します。`2` は `3` より小さいので、1 つ目のリストは 2 つ目のリストより小さいという結論になります。`<=` 、 `>=` 、 `>` の場合も同様です。

```
ghci> [3,2,1] > [2,1,0]
True
ghci> [3,2,1] > [2,10,100]
True
ghci> [3,4,2] < [3,4,3]
True
ghci> [3,4,2] > [2,4]
True
ghci> [3,4,2] == [3,4,2]
True
```

また、空でないリストは常に空リストよりも大きいとみなされます。これにより、一方のリストが他方の先頭部分に一致するようなケースを含めて、2 つのリストの順序がすべてについて明確に定義されます。

さらなるリスト操作

いくつかの基本的なリスト関数とその使い方を紹介します。

`head` 関数はリストを受け取り、その head（先頭の要素）を返します。

```
ghci> head [5,4,3,2,1]
5
```

`tail` 関数はリストを受け取り、その tail（先頭を取り除いた残りのリスト）を返します。

```
ghci> tail [5,4,3,2,1]
[4,3,2,1]
```

`last` 関数はリストの最後の要素を返します。

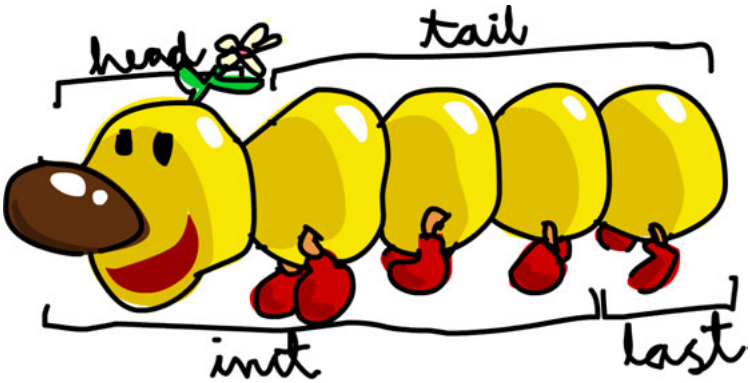
```
ghci> last [5,4,3,2,1]
1
```

`init` 関数はリストを受け取り、最後の要素を除いた残りのリストを返します。

```
ghci> init [5,4,3,2,1]
[5,4,3,2]
```

これらの関数を分かりやすく可視化するために、リストをこんな感じのモンスターと考えてみましょう。

空のリストの head を取ろうとすると何が起こるでしょう？




```
ghci> head []
*** Exception: Prelude.head: empty list
```

おっと、出鼻をくじかれました！ モンスターがいなければ頭（head）もないのです。 head 、 tail 、 last 、 init を使うときには空リストを渡さないように注意してください。このエラーはコンパイル時には捕らえられないので、空リストから何か取ってこいと Haskell にうっかり指示しないように気をつけましょう。

length 関数はリストを受け取り、その長さを返します。

```
ghci> length [5,4,3,2,1]
5
```

null 関数はリストが空かどうかを調べます。空なら True を、そうでなければ False を返します。

```
ghci> null [1,2,3]
False
ghci> null []
True
```

reverse 関数はリストを逆順にします。

```
ghci> reverse [5,4,3,2,1]
[1,2,3,4,5]
```

take 関数は数とリストを取り、先頭から指定された数の要素を取り出したリストを返します。

```
ghci> take 3 [5,4,3,2,1]
[5,4,3]
ghci> take 1 [3,9,3]
[3]
```

```
ghci> take 5 [1,2]
[1,2]
ghci> take 0 [6,6,6]
[]
```

take でリストの要素数より多い要素を取ろうとすると、Haskell はリスト全体を返します。 take で 0 個の要素を取ると、空リストが返ります。

同様に drop 関数は、指定された数の要素を先頭から削除したリストを返します。

```
ghci> drop 3 [8,4,2,1,5,6]
[1,5,6]
ghci> drop 0 [1,2,3,4]
[1,2,3,4]
ghci> drop 100 [1,2,3,4]
[]
```

maximum 関数は、何らかの順序が定義された要素からなるリストを受け取り、その中で最大の要素を返します。同様に minimum 関数は最小の要素を返します。

```
ghci> maximum [1,9,2,3,4]
9
ghci> minimum [8,4,2,1,5,6]
1
```

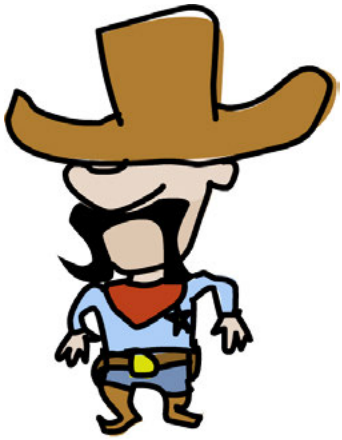
sum 関数は数のリストを受け取り、それらの和を返します。 product 関数は数のリストを受け取り、それらの積を返します。

```
ghci> sum [5,2,1,6,3,2,5,7]
31
ghci> product [6,2,1,2]
24
ghci> product [1,2,5,6,7,9,2,0]
0
```

elem 関数は要素とリストを受け取り、それがリストの要素に含まれているかどうかを返します。この関数は中置関数として使うと読みやすいので、中置関数として用いられることが多いです。

```
ghci> 4 `elem` [3,4,5,6]
True
ghci> 10 `elem` [3,4,5,6]
False
```

1.4 レンジでチン！



1 から 20 までの数からなるリストを作るには、どうすればいいでしょうか？ もちろん、手で打ち込んでもかまいませんが、それはプログラミング言語に優美さを求める紳士の調理方法ではありません。代わりに**レンジ**（range）を使いましょう。レンジは**列挙できる**要素の組み合わせでリストを作るのに使われます。

例えば、整数は 1, 2, 3, 4, ... といったふうに列挙できます。文字もアルファベットの A から Z のように列挙できます。でも例えば、名前は列挙できません（「ジョン」の次の名前は何ですか？ 知らんわ！）。

1 から 20 のすべての自然数を含むリストを作るには、`[1..20]` とタイプするだけです。Haskell ではこれは

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
```

とタイプするのとまったく同じことです。これら 2 つの書き方の唯一の違いは、長い列挙列を手で入力するのはバカげてる、ってことです。

いくつか例を挙げておきます。

```
ghci> [1..20]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]
ghci> ['a'..'z']
"abcdefghijklmnopqrstuvwxyz"
ghci> ['K'..'Z']
"KLMNOPQRSTUVWXYZ"
```

レンジには**ステップ**を指定することもできます。1 から 20 までのすべての偶数のリストが欲しいときはどうすればいいでしょうか？ あるいは、1 から 20 までのすべての 3 の倍数は？ 最初の要素と 2 つ目の要素をカンマで区切って、上限を指定するだけです。

```
ghci> [2,4..20]
[2,4,6,8,10,12,14,16,18,20]
ghci> [3,6..20]
[3,6,9,12,15,18]
```

これらはとても便利ですが、ステップ付きレンジは期待するほど賢くはありません。例えば、`[1,2,4,8,16..100]` と入力して、100 までのすべての 2 の累乗

を得ることはできません。この構文ではステップサイズを 1 つしか指定できません。それと、等差数列ではない任意の数列を、最初のいくつかの項を与えるだけで曖昧性なく特定することもできません。

NOTE 20 から 1 までの減少列を作るためには、`[20..1]` ではなくて、`[20,19..1]` と書かなければなりません。ステップなしで（`[20..1]` のように）レンジを書いた場合、Haskell は空リストから始めて、最初の要素が最後の要素より大きくなるまで要素を列挙します。20 はすでに 1 より大きいので、結果はただの空リストになります。

上限を指定しないことで、レンジを使って無限リストを生成できます。例えば、13 の倍数の最初の 24 個からなるリストを作ってみましょう。1 つのやり方として、こんな方法があります。

```
ghci> [13,26..24*13]
[13,26,39,52,65,78,91,104,117,130,143,156,169,182,195,208,221,234,
 247,260,273,286,299,312]
```

しかし、次のように無限リストを使うほうが良い方法です。

```
ghci> take 24 [13,26..]
[13,26,39,52,65,78,91,104,117,130,143,156,169,182,195,208,221,234,
 247,260,273,286,299,312]
```

Haskell は**遅延評価**なので、無限リスト全体をすぐには評価しません（無限リストの評価は終了しないので、これは望ましい動作です）。その代わりに、無限リストの中の要素が必要とされるまで待ちます。上の例では、最初の 24 個の要素しか要求しないので、Haskell は喜んでそれを返します。

すごく長い、もしくは無限の長さのリストを生成するために使える関数がいくつかあります。

- `cycle` はリストを受け取り、その要素を無限に繰り返し、無限リストを生成します。結果を表示しようとすると永遠に終わらないので、途中で切るのを忘れずに。

  ```
  ghci> take 10 (cycle [1,2,3])
  [1,2,3,1,2,3,1,2,3,1]
  ghci> take 12 (cycle "LOL ")
  "LOL LOL LOL "
  ```

- `repeat` は 1 つの要素を受け取り、その要素のみが無限に繰り返される無限リストを作ります。長さ 1 のリストを `cycle` にかけるのと同じです。

  ```
  ghci> take 10 (repeat 5)
  [5,5,5,5,5,5,5,5,5,5]
  ```

- `replicate` は単一の値からなるリストを作る簡単な方法です。リストの長さと、複製する要素を与えます。

```
ghci> replicate 3 10
[10,10,10]
```

レンジに関する最後の注意。浮動小数点数に使うときは気をつけて！ 浮動小数点数は精度に限りがあるので、レンジで使うと次のようなおかしな振る舞いをすることがあります。

```
ghci> [0.1, 0.3 .. 1]
[0.1,0.3,0.5,0.7,0.8999999999999999,1.0999999999999999]
```

1.5 リスト内包表記

リスト内包表記はリストのフィルタリング、変換、組み合わせを行う方法です。

これは数学における**集合の内包的記法**の概念に近いものです。集合の内包的記法は、他の集合から別の集合を作るときによく用いられます。単純な例として、$\{2 \cdot x \mid x \in \mathbf{N},\ x \leq 10\}$ のようなものがあります。この例で重要なのは、その厳密な構文ではなくて、「10 以下のすべての自然数を取ってきて、それぞれ 2 倍して、その結果を新しい集合としなさい」と言ってることです。

同じことを Haskell でやりたい場合、`take 10 [2,4..]` のようにリスト操作で書けるでしょう。でもリスト内包表記を使ってこんなふうに書くこともできます。

```
ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
```

この例を詳しく見てリスト内包表記の理解を深めましょう。

`[x*2 | x <- [1..10]]` では、リスト `[1..10]` から要素を**取り出す**と言っています。`[x <- [1..10]]` は、`[1..10]` から取り出した各要素の値を `x` が受け取るという意味です。これは別の表現では、`[1..10]` の各要素を `x` に**束縛**していると言えます。縦棒（`|`）より前の部分は、リスト内包表記の**出力**を表します。この出力パートでは、取り出した値を使ってどんなリストを作りたいかを指定します。この例だと、`[1..10]` から取り出した各要素を 2 倍したい、と言っています。

この例だと `take 10 [2,4..]` よりも複雑で長くなっているように見えますが、数の 2 倍なんかよりもっと複雑なことがやりたくなったらどうでしょうか？ ここからがリスト内包表記の本領発揮です。

例えば、さっきの内包表記に条件（述語とも呼びます）を追加してみましょう。述語はリスト内包表記の最後に置き、他のパートとはカンマで区切ります。2 倍した値が 12 以上のものからなるリストが欲しいとしましょう。

```
ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]
```

50 から 100 の数のうち、7 で割った余りが 3 であるすべての数が欲しかったら？ これも簡単。

```
ghci> [ x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]
```

NOTE　述語を使ってリストを間引くことをフィルタするといいます。

もっと別の例を考えましょう。10 以上のすべての奇数を "BANG!" に置き換え、10 より小さいすべての奇数を "BOOM!" に置き換える内包表記を考えます。数が奇数でなければリストから削除します。再利用のことを考えて、内包表記を関数の中に記述することにします。

```
boomBangs xs = [ if x < 10 then "BOOM!" else "BANG!" | x <- xs, odd x]
```

NOTE　GHCi の中で関数を定義しようとする場合は、関数名の前に `let` を付けるのを忘れずに。でも、この関数をスクリプトに書いてそこから GHCi にロードするなら、面倒くさい `let` と付き合う必要はありません。

odd 関数は、奇数が与えられたら True を、そうでなければ False を返します。すべての述語が True に評価された要素だけがリストに含まれます。

```
ghci> boomBangs [7..13]
["BOOM!","BOOM!","BANG!","BANG!"]
```

カンマで区切ることによって、たくさんの述語を含めることができます。例えば、10 から 20 の中で、13、15、19 でないすべての数が欲しいなら、次のようになります。

```
ghci> [ x | x <- [10..20], x /= 13, x /= 15, x /= 19]
[10,11,12,14,16,17,18,20]
```

述語を複数書くだけでなく、複数のリストから値を取り出すこともできます。複数のリストから値を取り出すと、それらのリストの要素のすべての組み合わせが結果のリストに反映されます。

```
ghci> [x+y | x <- [1,2,3], y <- [10,100,1000]]
[11,101,1001,12,102,1002,13,103,1003]
```

x はリスト [1,2,3] から取り出された値、y はリスト [10,100,1000] から取り出された値です。これら 2 つのリストが次のように組み合わされます。最初に x が 1 になり、x が 1 の間に、y が [10,100,1000] からすべての値を取ります。リスト内包表記の出力パートが x+y なので、結果のリストの先頭が 11 、101 、1001 の各値になります（10 、100 、1000 に 1 が足される）。その後、x が 2 になり、さっきと同じことが繰り返され、結果のリストに 12 、102 、1002 が追加されます。最後、x が 3 のときも同様です。

このように、リスト [1,2,3] の各要素から x とリスト [10,100,1000] の各要素から y とがすべてのあり得る組み合わせを取り、その組み合わせから x+y を使って結果のリストが作られます。

別の例を考えましょう。2 つのリスト [2,5,10] と [8,10,11] に対して、これらのリストの要素のすべての組み合わせの積を求めたいときは、次のような内包表記になります。

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

期待したとおり、新しいリストの長さは 9 です。では、すべての組み合わせの積のうちで 50 より大きいものだけ欲しいなら？ 述語を追加するだけです。

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11], x*y > 50]
[55,80,100,110]
```

今度は、形容詞と名詞のリストを組み合わせて変な言葉を作っちゃいましょう[†3]。

```
ghci> let nouns = ["hobo","frog","pope"]
ghci> let adjectives = ["lazy","grouchy","scheming"]
ghci> [adjective ++ " " ++ noun ⇨
        | adjective <- adjectives, noun <- nouns]
["lazy hobo","lazy frog","lazy pope","grouchy hobo","grouchy frog",
"grouchy pope","scheming hobo","scheming frog","scheming pope"]
```

リスト内包表記を使って length 関数を独自に定義することだってできます！これを length' と名づけましょう。この関数は、リストのすべての要素を 1 に置換してから sum で足し合わせてリストの長さを得ます。

```
length' xs = sum [1 | _ <- xs]
```

この例ではリストから取り出した値を利用しないので、それを使い捨てるために変数名アンダースコア（ _ ）を使っています。

[†3] ［訳注］本書では ghci> プロンプトに続けて入力する式がページ幅に収まらない場合、この例のように⇨マークで折り返し表示しています。実際には途中で改行せず 1 行で入力してくださいね。(22 ページの脚注も参照。)

文字列もリストなので、文字列を処理して生成するのにもリスト内包表記が使えます。次の例は文字列を受け取り、すべての小文字を取り除く（大文字だけを残す）関数です。

```
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

すべての仕事をこなしているのは、新しいリストに含まれるものはリスト ['A'..'Z'] に含まれるものだけだ、という述語です。GHCi にロードして試してみましょう。

```
ghci> removeNonUppercase "Hahaha! Ahahaha!"
"HA"
ghci> removeNonUppercase "IdontLIKEFROGS"
"ILIKEFROGS"
```

リストを含むリストを操作する場合には、入れ子になったリスト内包表記が作れます。数のリストからなるリストを受け取り、各要素はリストのままに、そこから奇数だけを取り除くという操作を考えましょう。

```
ghci> let xxs = [[1,3,5,2,3,1,2,4,5], ⇨
                 [1,2,3,4,5,6,7,8,9], ⇨
                 [1,2,4,2,1,6,3,1,3,2,3,6]]
ghci> [ [ x | x <- xs, even x ] | xs <- xxs]
[[2,2,4],[2,4,6,8],[2,4,2,6,2,6]]
```

外側のリスト内包表記の出力パートが、別のリスト内包表記になっています。リスト内包表記は必ず何らかのリストを返すので、全体の結果は数のリストのリストになることが分かります。

NOTE　可読性のためにリスト内包表記を複数行に分けることもできます。GHCi の中でコードを書くのでなければ、入れ子になった内包表記は複数行にしたほうが扱いやすいでしょう。

1.6　タプル

タプルは、複数の違う型の要素を格納して、1 つの値にするために使います。

タプルにはリストと似ている点がいくつかありますが、リストとタプルには根本的な違いがあります。1 つ目の違いはヘテロであること、つまり複数の違う型の要素が格納できるということです。2 つ目の違いは、タプルはサイズが固定だということです。格納する要素がいくつなのか事前に知っている必要があります。

タプルは括弧で囲み、要素をカンマで区切ります。

```
ghci> (1, 3)
(1,3)
ghci> (3, 'a', "hello")
(3,'a',"hello")
ghci> (50, 50.4, "hello", 'b')
(50,50.4,"hello",'b')
```

タプルを使う

どんなときにタプルが便利なのかを示す例として、Haskell で 2 次元ベクトルを表す方法を考えてみましょう。考えられる方法の 1 つは、2 要素からなるリスト `[x,y]` を用いるものです。その場合、2 次元座標上での図形の頂点を表現するベクトルのリストを作りたいとしたら、`[[1,2],[8,11],[4,5]]` のようなリストのリストになるでしょう。

この方法の問題点は、`[[1,2],[8,11,5],[4,5]]` のようなリストが作れて、これを 2 次元ベクトルのリストが期待される箇所で使えてしまうことです。このリストはベクトルのリストとしては意味を成さないにもかかわらず、型（ここでは数のリストのリスト）が同じであるという理由で、Haskell はベクトルのリストがくるべき箇所でこのリストを何の問題もなく受け入れてしまいます。これではベクトルや図形を操作する関数を書くときに厄介です。

これに対し、サイズ 2 のタプル（**ペア**とも呼ばれる）とサイズ 3 のタプル（**トリプル**とも呼ばれる）は、それぞれ違う型として扱われるので、ペアとトリプル両方を含むリストは作れません。このため、ベクトルを表すにはタプルを使ったほうがはるかに有利です。

先ほどのベクトルの角括弧を丸括弧に変えて、`[(1,2),(8,11),(4,5)]` のようにすれば、タプルになります。今度はペアとトリプルを混在させるとエラーになります。

```
ghci> [(1,2),(8,11,5),(4,5)]
Couldn't match expected type `(t, t1)'
against inferred type `(t2, t3, t4)'
In the expression: (8, 11, 5)
In the expression: [(1, 2), (8, 11, 5), (4, 5)]
In the definition of `it': it = [(1, 2), (8, 11, 5), (4, 5)]
```

Haskell は、長さが同じで違う型を含むタプルを区別できます。例えば、`[(1,2),("One",2)]` のようなタプルのリストは作れません。1 つ目のタプルは 2 つの数からなるペアですが、2 つ目のタプルは文字列と数からなるペアだからです。

タプルを使っていろいろなデータを簡単に表せます。例えば、人の名前と年齢を Haskell で表現するなら、("Christopher", "Walken", 55) のようなトリプルが使えます。

タプルは固定長だということを忘れないでください。あらかじめ必要とする要素の数が分かっている場合にだけ利用できます。タプルがこんなふうに融通が利かないのは、さっき見てきたようにそのサイズも型の一部だからです。そのため、残念ながらタプルに要素を追加する一般的な関数は書けません。ペアに要素を追加してトリプルを生成する関数、トリプルに要素を追加する 4-タプルを生成する関数、4-タプルに要素を追加する関数、さらに大きなタプルに要素を追加する関数などは、それぞれ別個に書かなければなりません。

リストと同じく、構成要素が比較可能であればそれを含むタプルも互いに比較できます。しかしリストとは違って、サイズの違うタプルを比較することはできません。

単一要素のリストは存在しますが、単一要素のタプルは存在しません。それがどういうものを表すかを考えれば理由が分かるでしょう。単一要素のタプルの性質は、それが保持している値と同じはずであり、それらを区別してもうれしいことは何もないからです。

ペアを使う

Haskell では値を格納するためにペアがとてもよく用いられるので、ペアを操作するための便利な関数がいくつか用意されています。

- fst はペアを受け取り、1 つ目の要素を返します。

```
ghci> fst (8, 11)
8
ghci> fst ("Wow", False)
"Wow"
```

- snd はペアを受け取ると、あらびっくり！ 2 つ目の構成要素を返します。

```
ghci> snd (8, 11)
11
ghci> snd ("Wow", False)
False
```

NOTE　これらの関数はペアに対してだけ働きます。トリプルや 4-タプル、5-タプルなどには使えません。トリプルから値を取り出す方法は少し後で出てきます。

zip 関数はペアのリストを作るスマートな方法です。 zip は 2 つのリストを受け取り、ジッパーみたいに 1 つのリストにします。とてもシンプルな関数です

が、2 つのリストを同時に走査するときにとても便利な関数です。こんなふうに使います。

```
ghci> zip [1,2,3,4,5] [5,5,5,5,5]
[(1,5),(2,5),(3,5),(4,5),(5,5)]
ghci> zip [1..5] ["one", "two", "three", "four", "five"]
[(1,"one"),(2,"two"),(3,"three"),(4,"four"),(5,"five")]
```

ペアは違う型を含むことができるので、 zip は違う型のリストを受け取れることに注意してください。では、リストの長さが違う場合はどうなるでしょう？

```
ghci> zip [5,3,2,6,2,7,2,5,4,6,6] ["im","a","turtle"]
[(5,"im"),(3,"a"),(2,"turtle")]
```

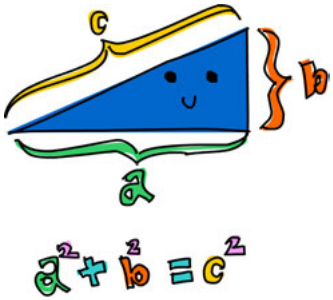
この例で分かるように、長いほうのリストは必要な分だけが使われ、余りは無視されます。Haskell は遅延評価なので、有限リストと無限リストを zip することもできます。

```
ghci> zip [1..] ["apple", "orange", "cherry", "mango"]
[(1,"apple"),(2,"orange"),(3,"cherry"),(4,"mango")]
```

直角三角形を見つける

タプルとリスト内包表記を組み合わせて 1 つ問題を解いてみましょう。Haskell を用いて次のすべての条件を満たす直角三角形を見つけるプログラムを書きます。

- 3 辺の長さはすべて整数である。
- 各辺の長さは 10 以下である。
- 周囲の長さは 24 に等しい。



1 つの角が直角（90 度）であるような三角形のことを直角三角形と呼びます。直角三角形には、直角を挟む 2 辺の長さを 2 乗して足し合わせると、その対辺を 2 乗したものに等しくなるという便利な性質があります。図では、直角の両脇の直線にラベル a と b が付けられていて、直角の対辺にラベル c が付けられています。この辺を**斜辺**と呼びます。

最初のステップとして、各要素が 10 以下であるようなトリプルをすべて生成してみましょう。

```
ghci> let triples = ⇨
        [ (a,b,c) | c <- [1..10], a <- [1..10], b <- [1..10] ]
```

内包表記の右側の部分で 3 つのリストから値を取り出し、出力の式で 3 つの値を組み合わせてトリプルのリストを作っています。 `triples` を評価すると、1000 要素のリストが表示されるはずなので、ここには示しません。

次に、ピタゴラスの定理（ `a^2 + b^2 == c^2` ）が成り立つかを調べる述語を追加して、直角三角形でないものをフィルタしましょう。また、 `a` が斜辺 `c` を超えないように、 `b` が `a` を超えないように、それぞれ変更を加えます。

```
ghci> let rightTriangles = ⇨
        [ (a,b,c) | c <- [1..10], a <- [1..c], b <- [1..a], ⇨
                    a^2 + b^2 == c^2]
```

リストのレンジをどんなふうに変更したかに注目してください。 `b` が斜辺より長いような不要なトリプルを調べないような変更をしています（正しい直角三角形は必ず斜辺が一番長いのです）。

また、辺 `b` が辺 `a` を超えないようにしています。このように変更しても問題ありません。なぜならば、 `a^2 + b^2 == c^2` および $b > a$ を満たすトリプル `(a,b,c)` は考えなくていいからです。トリプル `(b,a,c)` は検査対象に残っていて、これは同じ三角形の辺を入れ替えたものです（そうしないと、結果のリストに本質的には同じ三角形が含まれることになります）。

NOTE GHCi では式の定義を複数行にわたって書くことができません[†4]。この本には、紙面の幅に合わせるために 1 行を複数行に分割している箇所がたまにあります（そうしないと、本の幅がとんでもないことになって、普通の本棚に入らなくなって、読者の皆様はでっかい本棚を買わなければならなくなります）。

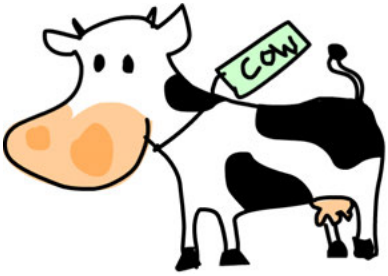
もうほとんど終わりです。関数を修正して、周囲の長さが 24 のものだけ出力しましょう。

```
ghci> let rightTriangles' = ⇨
        [ (a,b,c) | c <- [1..10], a <- [1..c], b <- [1..a], ⇨
                    a^2 + b^2 == c^2, a+b+c == 24]
ghci> rightTriangles'
[(8,6,10)]
```

答が出ました！ このように、最初に解の候補となる集合を生成し、それから 1 つ（もしくは複数）の解に辿り着くまで変換とフィルタリングを行うという手法は、関数プログラミングでよく用いられるパターンです。

[†4] ［訳注］翻訳時点の Haskell Platform に同梱されている GHCi では、 `:{` と `:}` で挟むことにより複数行にわたって式を書けます。また GHCi 7.2.1 以降では、 `:set +m` とすることで複数行にわたって式を書けるようになりました。

第2章
型を信じろ！



Haskell の強みの 1 つは、その強力な型システムです。

Haskell では、すべての式の型がコンパイル時に分かっていて、そのことがコードを安全にしています。例えば、真理値型を数値で割ろうとすると、コンパイル時にエラーになります。こういった類のエラーをコンパイル時に捕まえることができるのは、実行時にプログラムがクラッシュするよりも好ましいことです。Haskell ではすべてのものが型を持つので、コンパイラはプログラムを見るだけで実に多くのことを推論できます。

Java や Pascal とは違って、Haskell には型推論があります。例えば、数を書いたなら、「これは数の型だよ」と教えてあげなくても、Haskell はそれを自力で推論することができます。

型については、Haskell の基礎とあわせてまだほんの上っ面しか眺めていませんが、型システムをしっかり理解することは Haskell を学ぶ上でとても重要です。

2.1 明示的な型宣言

式の型は GHCi を使って調べることができます。 `:t` コマンドに続けて正しい式を入力すればその式の型を教えてくれます。試しにやってみましょう。

```
ghci> :t 'a'
'a' :: Char
```

```
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

`::` という記号は、「の型を持つ。」と読みます。明示的な型の名前の先頭は常に大文字です。`'a'` は、**文字**を意味する型 `Char` を持ちます。`True` はBool、つまり真理値型です。`"HELLO!"` は文字列ですが、型 `[Char]` を持つと表示されています。角括弧はリストのことなので、これは文字のリストと読むことができます。タプルはリストとは違い、個々の要素が型を持ちます。タプル `(True, 'a')` は型 `(Bool, Char)` を持ち、`('a','b','c')` は型 `(Char, Char, Char)` を持ちます。`4 == 5` は、常に `False` を返すので、その型は `Bool` です。

関数も型を持ちます。自分で関数を書くとき、その関数に明示的な型宣言を与えることができます。これは Haskell のプログラムを書く上で、（とても短い関数を書く場合は除いて）一般的に良い習慣だと考えられています。なので、これからはすべての関数に明示的な型宣言を与えることにします。

第１章で作った、文字列中の小文字をフィルタするリスト内包表記を思い出しましょう。これに型宣言を与えると、次のようになります[†1]。

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

`removeNonUppercase` 関数は型 `[Char] -> [Char]` を持ちます。これは、1つの文字列を引数として取り、別の文字列を結果として返すという意味です。

では、関数が複数の引数を持つ場合はどうでしょうか？ 3つの整数を受け取ってそれらを足し合わせる単純な関数はこうなります。

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

†1　［訳注］この例は、まず.hs ファイルを作って試してください。GHCi の中で、型宣言付きの関数を定義するにはちょっとした工夫が必要です。具体的な方法はいくつかありますので、探してみてください！

引数と返り値の型は `->` で区切り、返り値の型は常に宣言の最後に置きます(引数と返り値を区別せず、どれも `->` で区切っている理由は、第 5 章で明らかになります)。

関数に型宣言を与えたいけど、その型が何になるのか書いてみないとよく分からない場合は、はじめに型宣言なしで関数を書き、それから `:t` を使ってその型を調べればよいでしょう。関数は式なので、この章の冒頭で見たように `:t` が使えます。

2.2 一般的な Haskell の型

数や文字や真理値といった、よく使う Haskell の型をいくつか見てみましょう。

- `Int` は整数です。数全般に使います。`7` は `Int` になれますが、`7.2` はなれません。`Int` は**有界**、つまり最小値と最大値があります。

  NOTE GHC コンパイラでは `Int` の範囲がマシンのワードサイズによって変わります。64 ビット CPU では `Int` の最小値は -2^{63} で、最大値は $2^{63}-1$ になるでしょう。

- `Integer` も整数に使いますが、こちらは有界ではないので、とても大きい数(それも**はんぱなく**大きい数!)を表すのに使えます。とはいえ、`Int` のほうが効率的です。`Integer` の例として、次の関数をファイルに保存してみてください。

  ```
  factorial :: Integer -> Integer
  factorial n = product [1..n]
  ```

  それから `:l` で GHCi にロードして試してみましょう。

  ```
  ghci> factorial 50
  30414093201713378043612608166064768844377641568960512000000000000
  ```

- `Float` は単精度浮動小数点数です。ファイルに次の関数を追加してください。

  ```
  circumference :: Float -> Float
  circumference r = 2 * pi * r
  ```

  ロードして試してみましょう。

  ```
  ghci> circumference 4.0
  25.132742
  ```

- `Double` は倍精度浮動小数点数です。倍精度型が数を表すのに使うビット数は `Float` の 2 倍です。より多くのリソースを必要とする分、精度が高

くなります。次の関数をファイルに追加しましょう。

```
circumference' :: Double -> Double
circumference' r = 2 * pi * r
```

ロードして試してみましょう。circumference と circumference' の精度の違いに特に注目してください。

```
ghci> circumference' 4.0
25.132741228718345
```

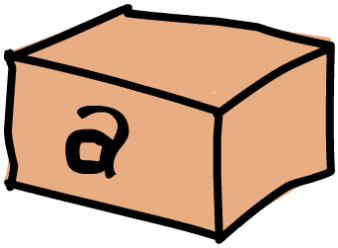
- Bool は真理値型です。真理値型が持てる値は True と False の 2 つのみです。
- Char は Unicode 文字を表します。シングルクオート（'）で括って表記します。文字のリストは文字列です。
- タプルも型ですが、その定義は要素の数とそれぞれの型によって決まります。そのため、理論的には無限の種類のタプル型が存在します（実際の処理系ではタプルの要素の最大数は 62 です。でも、そんなに必要になることはまずないでしょう）。空のタプル () も型だということに注意してください。この型はただ 1 つの値 () のみを持ちます[†2]。

2.3　型変数

いろいろな型に対して動作する関数があります。例えば head 関数は、リストを取りその先頭の要素を返します。この関数は、リストの要素が数だろうと文字だろうと、はたまたさらにネストしたリストだろうと、何も問題ないはずです！なのでこの関数はあらゆる型のリストに対して動作するべきです。

head 関数の型は何だと思いますか？　:t で調べてみましょう。

```
ghci> :t head
head :: [a] -> a
```



a とは何でしょうか？ 型の名前だとしたら大文字から始まるはずでしたね。なので、これは型ではありません。この a は**型変数**と呼ばれるもので、どんな型も取り得るということを意味します。

型変数は、型安全を保ったまま、関数を複数の型に対して動作できるようにしてくれ

[†2] ［訳注］「 () 」は「ユニット」と呼ばれます。

ます。他のプログラミング言語にあるジェネリクスにちょっとだけ似ていますが、Haskell の型変数はとても一般的な関数を簡単に書けるので、はるかにパワフルです。

型変数を用いた関数は多相的関数と呼ばれます。 head の型宣言は「任意の型のリストを引数に取り、その型の要素を 1 つ返す」と読むことができます。

NOTE 型変数には 1 文字より長い名前をつけてもかまいませんが、 a とか b とか c とか d のような名前をつけることが多いです。

ペアの 1 つ目の要素を返す関数 fst を覚えていますか？ この関数の型を調べてみましょう。

```
ghci> :t fst
fst :: (a, b) -> a
```

fst はタプルを引数に取り、その 1 つ目と同じ型の値を返すことが見て取れるでしょう。これがどんな型のペアに対しても fst が使える理由です。 a と b は違う型変数ですが、必ずしも違う型である必要はありません。ペアの 1 つ目の要素の型と返り値の型が同じであると言っているだけです。

2.4 型クラス 初級講座

型クラスは、何らかの振る舞いを定義するインターフェイスです。ある型クラスのインスタンスである型は、その型クラスが記述する振る舞いを実装します。

もっと具体的に言うと、型クラスというのは関数の集まりを定めます。ある型クラスに属する関数のことを、その型クラスのメソッドと呼ぶこともあります。ある型を型クラスのインスタンスにしようと考えたときには、それらの関数がその型ではどういう意味を成すのかを定義します。

等値性を定義する型クラスが良い例です。多くの型について、その値の等値性を == 演算子を使って比較できます。この演算子の型を調べてみましょう。

```
ghci> :t (==)
(==) :: (Eq a) => a -> a -> Bool
```

等値性演算子（ == ）は実際には関数であることに注意してください。 + 、 *、 - 、 / など、ほかのほとんどすべての演算子も同様です。関数の名前が特殊文字のみからなる場合、その関数はデフォルトで中置関数になります。その型を調べたい場合や、他の関数に渡したい場合、あるいは前置関数として呼び出したい場合には、上の例のように丸括弧で囲む必要があります。

この例には見慣れないものがありますね。 => というシンボルです。このシンボルよりも前にあるものは**型クラス制約**と呼ばれます。この例の型宣言は、「等値性関数は、同じ型の任意の 2 つの引数を取り、 Bool を返す。引数の 2 つの値の型は Eq クラスのインスタンスでなければならない」と読めます。

Eq 型クラスは、等値性をテストするためのインターフェイスを提供します。ある型の 2 つの値の等値性を比較することに意味があるなら、その型は Eq 型クラスのインスタンスにできます。Haskell のすべての標準型（I/O 型と関数を除く）は Eq のインスタンスです。

NOTE 型クラスはオブジェクト指向のクラスとは同じではないということに注意してください。これは重要です。

よく使われる Haskell の型クラスをいくつか見ていきましょう。型の値の等値性や順序を簡単に比較したり、手軽に文字列として表示したりできるのは、これらの型クラスのおかげです。

Eq 型クラス

すでに見たように、 Eq は等値性をテストできる型に使われます。 Eq のインスタンスが実装すべき関数は == と /= です。これは、関数の型変数に Eq クラスの制約が付いていたら、その関数の定義のどこかで == か /= が使われているということです。型が関数を実装しているとは、その関数がその特定の型に対して使われたときに、どういう振る舞いをするか定義するということです。 Eq のさまざまなインスタンスに対する == と /= の動作の例をいくつか挙げます。

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

Ord 型クラス

Ord は、何らかの順序を付けられる型のための型クラスです。例えば、大なり演算子（ > ）の型を見てみましょう。

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

> の型は == の型に似ています。引数を 2 つ取り、それらが関係を満たすかどうかを教えてくれる Bool を返します。

今まで見てきた型は、またもや関数は除いて、すべて Ord のインスタンスです。 Ord はすべての標準的な大小比較関数、 > 、 < 、 >= 、 <= をサポートします。

compare 関数は Ord のインスタンスの型の引数を 2 つ取り、 Ordering を返します。 Ordering は GT 、 LT または EQ のいずれかの値を取る型で、それぞれ「より大きい」、「より小さい」、「等しい」を意味します。

```
ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
ghci> 'b' > 'a'
True
```

Show 型クラス

ある値は、その型が Show 型クラスのインスタンスになっていれば、文字列として表現できます。今まで見てきた型は、関数を除けば、すべて Show のインスタンスです。この型クラスのインスタンスに対する操作で一番よく使うのは show で、これは指定した値を文字列として表示する関数です。

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

Read 型クラス

Read は Show と対をなす型クラスです。またしても、今まで見てきたすべての型はこの型のインスタンスです。 read 関数は文字列を受け取り、 Read のインスタンスの型の値を返します。

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

ここまでは順調です。しかし、 read "4" とタイプしてみるとどうなるでしょうか？

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable 'a' in the constraint:
      'Read a' arising from a use of 'read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

GHCi は、何を返せばいいか分からないよ、と言ってます。 read を使った直前の例で返すべき値の型を GHCi が推論できていたのは、結果の値に何かしらの手が加えられていたからです。例えば真理値として使われている場合には、Bool を返すべきだと GHCi は推論できます。しかし read "4" では、 Read クラスのどれかを返すことしか分からず、具体的に何を返せばいいのか GHCi には分かりません。 read の型シグネチャを見てみましょう。

```
ghci> :t read
read :: (Read a) => String -> a
```

NOTE　String は、 [Char] の単なる別名です。 String と [Char] は、それぞれまったく同じように使えますが、本書ではもっぱら String を使うことにします。 String のほうが簡単に書けるし、読みやすいですからね。

型シグネチャから、 read 関数が返す値の型は、 Read のインスタンスであることは読み取れますが、返された値の使い方によっては、それが具体的にどの型なのか判定できなくなってしまうことがあります。 この問題を解決するために、**型注釈**というものを用います。

型注釈は、式が取るべき型を Haskell に明示的に教えてあげる手段です。式の終わりに :: を追記し、それから型を指定します。

```
ghci> read "5" :: Int
5
ghci> read "5" :: Float
5.0
ghci> (read "5" :: Float) * 4
20.0
ghci> read "[1,2,3,4]" :: [Int]
[1,2,3,4]
ghci> read "(3, 'a')" :: (Int, Char)
(3, 'a')
```

コンパイラはほとんどの式の型を自力で推論できます。しかし、read "5" の型が Int か Float か分からないように、返り値の型をうまく推論できない場合もあります。read "5" を実際に評価することができれば、その型を知ることもできるかもしれませんが、Haskell は静的型付け言語なので、コードをコンパイルする（あるいは GHCi で評価される）前にすべての型が分かっている必要があります。そこで、

「ヘイ、分からないなら教えてあげるけど、この式はこの型なんだよ！」

と Haskell に伝えてあげるわけです。

Haskell に read が返すべき値の型が何なのかを教えるのは最小限でかまいません。例えば read の結果をリストの中に詰め込めば、Haskell はそのリストを通じて、「返り値の型はリストの他の要素の型である」と知ることができます。

```
ghci> [read "True", False, True, False]
[True, False, True, False]
```

read "True" は Bool 値のリストの要素なので、read "True" も Bool でなければならないと分かります。

Enum 型クラス

Enum のインスタンスは、順番に並んだ型、つまり要素の値を列挙できる型です。Enum 型クラスの主な利点は、その値をレンジの中で使えることです。また、Enum のインスタンスの型には後者関数 succ と前者関数 pred も定義されます。Enum クラスのインスタンスとしては、()、Bool、Char、Ordering、Int、Integer、Float、Double などがあります。

```
ghci> ['a'..'e']
"abcde"
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ 'B'
'C'
```

Bounded 型クラス

Bounded 型クラスのインスタンスは上限と下限を持ち、 それぞれ minBound と maxBound 関数で調べることができます。

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
'\1114111'
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
False
```

minBound と maxBound 関数は、(Bounded a) => a という面白い型を持っています。これらは、いわば多相定数です。

タプルのすべての構成要素が Bounded のインスタンスなら、そのタプル自身も Bounded になることに気をつけてください。

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

Num 型クラス

Num は数の型クラスです。このインスタンスは数のように振る舞います。数の型を調べてみましょう。

```
ghci> :t 20
20 :: (Num t) => t
```

あらゆる数もまた多相定数として表現されていて、 Num 型クラスの任意のインスタンス（ Int 、 Integer 、 Float 、 Double など）として振る舞うことができます。

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

例えば、演算子 * の型を調べることができます。

```
ghci> :t (*)
(*) :: (Num a) => a -> a -> a
```

これは、* が 2 つの数を受け取って 1 つの数を返すこと、これら 3 つの数はすべて同じ型であることを示しています。この型クラス制約により、(5 :: Int) * (6 :: Integer) は型エラーになり、5 * (6 :: Integer) は正しく動きます。5 は Integer や Int として振る舞うことができますが、同時に両方にはなれません。

ある型を Num のインスタンスにするには、その型がすでに Show と Eq のインスタンスになっている必要があります[†3]。

Floating 型クラス

Floating 型クラスには Float と Double が含まれます。この型クラスは浮動小数点数に使います。

引数と返り値が Floating 型クラスのインスタンスであるような関数は、その結果を浮動小数点数で表現できないと意味のある計算ができません。例としては、sin 、cos 、sqrt などがあります。

Integral 型クラス

Integral もまた数の型クラスです。Num が実数を含むすべての数を含む一方、Integral には整数（全体）**のみ**が含まれます。この型クラスは Int と Integer を含みます。

数を扱うとりわけ便利な関数として fromIntegral があります。この関数は次のような型を持っています。

```
fromIntegral :: (Num b, Integral a) => a -> b
```

NOTE fromIntegral の型シグネチャに複数の型クラス制約があることに注目してください。複数の型クラス制約を書くときは、このようにカンマ（,）で区切って括弧で囲みます。

この型シグネチャから分かるのは、fromIntegral は何らかの整数を引数に取り、もっと一般的な数を返すということです。この関数は、整数と浮動小数点数を一緒に扱いたいときにとても役に立ちます。例えば、length 関数はこのような型宣言を持っています。

```
length :: [a] -> Int
```

[†3] ［訳注］base-4.5（GHC 7.4.1）から、この制限は撤廃されました。

そのため、リストの長さを取得して、それに 3.2 を加えるような式はエラーになります（ Int と浮動小数点数を足し合わせようとしているため)。それをやりたい場合は fromIntegral を使ってこうします。

```
ghci> fromIntegral (length [1,2,3,4]) + 3.2
7.2
```

最後に —— 型クラスに関するいくつかの注意

型クラスは抽象的なインターフェイスとして定義されているので、1 つの型はいくつもの型クラスのインスタンスになることができるし、1 つの型クラスはいくつもの型をインスタンスとして持てます。例えば、 Char 型をインスタンスとする型クラスはたくさんあって、その中には Eq と Ord クラスがあり、これは 2 つの文字は等値性比較とアルファベット順比較の両方ができるからです。

型をある型クラスのインスタンスにするために、いったん別のインスタンスにする必要があることがあります。例えば、 Ord クラスのインスタンスになるには、先に Eq クラスのインスタンスになる必要があります。言い換えれば、 Eq クラスのインスタンスであることは Ord クラスのインスタンスになるための**必要条件**です。2 つのものが順序付けられるということは、それらが等しいかどうかも分かるはずだ、と考えれば納得できるでしょう。

第3章
関数の構文

この章では、Haskell の関数を書くための構文を見ていきます。Haskell の関数は読みやすく、理にかなった記法です。値を手軽に分解する方法、大きな `if/else` の連鎖を避ける方法、計算の中間データを一時的に保存して複数回利用する方法を見ていきます。

3.1 パターンマッチ

パターンマッチはある種のデータが従うべきパターンを指定し、そのパターンに従ってデータを分解するために使います。

Haskell では、関数を定義する際にパターンマッチを使って、関数の本体を場合分けできます。これによってシンプルで読みやすいコードが書けます。数値、文字、リスト、タプルなど、とても多くのデータ型でパターンマッチを使うことができます。例として、渡された数が 7 かどうか調べる単純な関数を書いてみましょう。

```
lucky :: Int -> String
lucky 7 = "LUCKY NUMBER SEVEN!"
lucky x = "Sorry, you're out of luck, pal!"
```

`lucky` を呼ぶと、パターンが上から下の順で試されます。渡された値が指定されたパターンに合致すると、対応する関数の本体が使われ、残りのパターンは無視されます。最初のパターンに合致するのは 7 が渡されたときだけで、その場合に関数の本体 `"LUCKY NUMBER SEVEN!"` が使われます。渡された引数が 7 でなければ 2 つ目のパターンに落っこちます。2 つ目のは何にでも合致するパターンで、引数は `x` に束縛されます。

パターンに（7 のような）具体的な値でなく小文字から始まる名前（x や y、または myNumber）を書くと、任意の値に合致するようになります。このパターンは与えられた値に常に合致し、その値をパターンに使った名前で参照できるようになります。

先ほどの例の関数であれば **if** 式を使っても簡単に実装できます。では、1 から 5 が入力されたときはそれを単語として出力し、それ以外なら "Not between 1 and 5" と出力するような関数ならばどうでしょうか？ パターンマッチなしではとてもややこしい **if**/**then**/**else** が必要になるでしょう[†1]。でもパターンマッチがあれば、とても単純に書けます。

```
sayMe :: Int -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

最後のパターン（sayMe x）を先頭に持ってくると、この関数は常に "Not between 1 and 5" を表示するようになってしまいます。あらゆる入力が最初のパターンに合致してしまって、他のパターンへ落ちていかなくなるからです。

前の章で実装した階乗を計算する関数を覚えていますか？ そこでは n の階乗を product [1..n] と定義していました。この関数を**再帰的**に定義することもできます。関数が再帰的に定義されているとは、その関数の定義の中で自分自身を呼び出しているということです。数学の階乗関数は、普通はそのように定義されます。まず、「0 の階乗は 1」と定義します。次に、「すべての正の整数の階乗は、その整数とそれから 1 を引いたものの階乗の積」と定義します。これを Haskell で実装すると次のようになります。

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

初めて再帰的な関数を定義しました！ Haskell では再帰は重要なので、第 4 章でじっくりと見ていくことにします。

パターンマッチは失敗することもあります。例えば、次のような関数を定義することができます。

---

†1　［訳注］手続き型言語だって switch 文を使えば簡潔に書けるじゃないかって？ はい、この例に関しては、そのとおりです。でも、パターンマッチのほうがずっと豊富な記述力があるってことを、おいおい見ていきますよ！

```
charName :: Char -> String
charName 'a' = "Albert"
charName 'b' = "Broseph"
charName 'c' = "Cecil"
```

この関数は一見するとうまく動くように見えますが、予期していない値が入力されるとエラーになります。

```
ghci> charName 'a'
"Albert"
ghci> charName 'b'
"Broseph"
ghci> charName 'h'
"*** Exception: tut.hs:(53,0)-(55,21): Non-exhaustive patterns in
function charName
```

「non-exhaustive patterns（パターンが網羅的でない）」と文句を言われました。おっしゃるとおりです。パターンを作るときは、すべてに合致するパターンを最後に入れておくべきです。そうすれば予期せぬ入力でプログラムがクラッシュすることを防げます[†2]。

タプルのパターンマッチ

パターンマッチはタプルでも使えます。2 つの 2 次元空間のベクトル（ペアで表す）を受け取り、それらを足し合わせる関数を書きたいとします（2 つのベクトルの加算は、x 成分と y 成分をそれぞれ足し合わせます）。パターンマッチを知らなければこう書くことになるでしょう。

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors a b = (fst a + fst b, snd a + snd b)
```

これでも動きますが、もっと良い方法があります。パターンマッチを使って書き換えてみましょう。

```
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

こっちのほうが、引数がタプルであることが分かりやすいし、タプルの要素に適切な名前がついているので読みやすくなっています。これですべてに合致するパターンになっていることに注意してください。 addVectors の型は 1 つ目のと同じく

[†2] ［訳注］ -fwarn-incomplete-patterns オプションを使えば、パターンマッチが網羅的でない場合に警告してもらえます。 -Wall オプションを使えば、そのほかにもたくさんの有用な警告が有効になります。 .hs ファイルの先頭に {-# OPTIONS -Wall -Werror #-} を付けてコーディングすれば、意図せぬ動作やクラッシュの原因になりがちな箇所を GHC が指摘してくれるので、良い Haskell コードを書く訓練になりますよ！

```
ghci> :t addVectors
addVectors :: (Double, Double) -> (Double, Double) -> (Double, Double)
```

なので、 Double のペアが 2 つ引数にくることは保証済みです。

fst と snd を用いるとペアの要素を分解できました。では、トリプルに対してはどうしましょう？ トリプルの 3 番目の要素を取り出す関数は提供されていませんが、独自に定義することはできます。

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

second :: (a, b, c) -> b
second (_, y, _) = y

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

_ はリスト内包表記のときと同じ意味です。ここの値はどうでもよいので、使い捨ての変数を表すために _ を使っています[†3]。

リストのパターンマッチとリスト内包表記

リスト内包表記でも次のようにパターンマッチが使えます。

```
ghci> let xs = [(1,3),(4,3),(2,4),(5,3),(5,6),(3,1)]
ghci> [a+b | (a, b) <- xs]
[4,7,6,8,11,4]
```

リスト内包表記のパターンマッチでは、失敗したら単に次の要素に進み、失敗した要素は結果のリストには含まれません。

```
ghci> [x*100+3 | (x, 3) <- xs]
[103,403,503]
```

普通のリストもパターンマッチで使えます。空リスト [] 、または : を含むパターンと合致させることができます（ [1,2,3] は 1:2:3:[] の構文糖衣だということを思い出してください）。 x:xs というパターンは、リストの先頭要素を x に束縛し、リストの残りを xs に束縛します。リストの要素がちょうど 1 つだった場合、 xs には空のリストが束縛されます。

NOTE Haskell プログラマは x:xs というパターンを、特に再帰関数と一緒によく使います。ただし : を含むパターンは、長さが 1 以上のリストに対してしか合致しません。

さて、たった今覚えたリストに対するパターンマッチを使って、独自の head 関数を head' という名前で実装してみましょう。

[†3] ［訳注］ _ は予約語なので通常の変数としては使えません。

```
head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x:_) = x
```

関数をロードしたら次のように試せます。

```
ghci> head' [4,5,6]
4
ghci> head' "Hello"
'H'
```

head' 関数の定義にあるように、複数の変数（そのうちの 1 つが _ の場合も）に束縛したいときは丸括弧で囲まなければシンタックスエラーになります。

error という関数を使っていることにも注目してください。error 関数は文字列を引数に取り、その文字列を使ってランタイムエラーを生成します。これは基本的にはプログラムをクラッシュさせるものなので、みだりに使ってはいけません。（でも空リストの head は取れないから仕方ないね！）[†4]

もう 1 つの例として、リストの要素を回りくどくて不便な書式で出力する関数を書いてみましょう。

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: " ++ show x ++ " and "
                ++ show y
tell (x:y:_) = "This list is long. The first two elements are: "
                ++ show x ++ " and " ++ show y
```

(x:[]) と (x:y:[]) は、それぞれ [x] および [x,y] と書くこともできます。しかし、(x:y:_) を角括弧を使って書き直すことはできません。このパターンは長さが 2 以上の任意のリストに合致するからです。

この関数を使ってみます。

```
ghci> tell [1]
"The list has one element: 1"
ghci> tell [True,False]
"The list has two elements: True and False"
ghci> tell [1,2,3,4]
"This list is long. The first two elements are: 1 and 2"
ghci> tell []
"The list is empty"
```

tell 関数は、空のリストにも、単一要素のリストにも、2 要素のリストにも、あるいはもっと多くの要素のリストにも合致するので、安全に使えます。あらゆ

[†4] ［訳注］head などのエラーを引き起こす可能性のある関数（部分関数）は、後で出てくる Maybe のような失敗を安全に扱える方法で実装すべきだ、という意見があります。実際、Prelude の部分関数を安全な方法で再実装した safe という名前の Hackage が用意されています（hackage.haskell.org/package/safe）。

る長さのリストをどう扱えばいいか分かっているので、常に意味のある値を返せるのです。

3 要素のリストを扱う方法しか知らない関数を定義したらどうなるでしょう？例えばこんな関数です。

```
badAdd :: (Num a) => [a] -> a
badAdd (x:y:z:[]) = x + y + z
```

予期しないリストが与えられたときに何が起こるでしょうか。

```
ghci> badAdd [100,20]
*** Exception: examples.hs:8:0-25: Non-exhaustive patterns in
function badAdd
```

おっと、かっこ悪い！ これがもし GHCi じゃなくてコンパイルされたプログラムの中で起こってたら、プログラムはクラッシュしていたところです。

リストに対するパターンマッチの最後の注意として、パターンマッチでは ++ 演算子を使えません（ ++ は 2 つのリストをつなげる演算子でしたね）。例えば、パターン (xs ++ ys) に対して合致させようにも、リストのどの部分を xs に合致させて、どの部分を ys に合致させればいいか、Haskell に伝えようがありません。(xs ++ [x,y,z]) や (xs ++ [x]) というパターンなら、リストの性質から論理的に合致できそうに思えますが、実際にそのような賢い合致はできません。

as パターン

Haskell には **as**(あず)**パターン**と呼ばれる特殊なパターンがあります。as パターンは、値をパターンに分解しつつ、パターンマッチの対象になった値自体も参照したいときに使います。as パターンを作るには、普通のパターンの前に名前と @ を追加します。

例えば、 xs@(x:y:ys) のような as パターンを作れます。このパターンは、x:y:ys に合致するものとまったく同じものに合致しつつ、 xs で元のリスト全体にアクセスすることもできます。 x:y:ys とタイプしなくてもいいのです。as パターンの簡単な例を示します。

```
firstLetter :: String -> String
firstLetter "" = "Empty string, whoops!"
firstLetter all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

関数をロードして試してみましょう。

```
ghci> firstLetter "Dracula"
"The first letter of Dracula is D"
```

3.2 場合分けして、きっちりガード！

関数を定義する際、引数の構造で場合分けするときにはパターンを使います。引数の値が満たす性質で場合分けするときには、**ガード**を使います[†5]。 性質で場合分けというと何だか **if** 式っぽいですし、実際 **if** とガードは似ています。

ただし、複数の条件があるときにはガードのほうが **if** より可読性が高く、パターンマッチとの相性も抜群です。

ガードを使った関数の世界に飛び込んでみましょう。肥満度指数（BMI）によって異なる叱り方をする関数を定義します。BMI とは、体重（キログラム）を身長（メートル）の 2 乗で割った値です。BMI が 18.5 以下なら痩せています。18.5 から 25 の間なら標準体重です。BMI が 25 から 30 は太りすぎで、30 以上は肥満です。（BMI を計算する関数ではなく、計算済みの BMI を受け取って忠告するだけの関数です。）

```
bmiTell :: Double -> String
bmiTell bmi
  | bmi <= 18.5 = "You're underweight, you emo, you!"
  | bmi <= 25.0 = "You're supposedly normal.\
                  \ Pffft, I bet you're ugly!"
  | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
  | otherwise   = "You're a whale, congratulations!"
```

ガードには、パイプ文字（|）とそれに続く真理値式、さらにその式が `True` に評価されたときに使われる関数の本体が続きます。式が `False` に評価されたら、その次のガードの評価に移ります。この繰り返しです。ガードは最低でも 1 つのスペースでインデントする必要があります（読みやすさのためにも、スペース 4 つでインデントするのがおすすめです[†6]）。

例えば、この関数を 24.3 という BMI で呼び出した場合、最初に BMI が 18.5 以下であるか調べます。そうではないので、次のガードに移動します。2 つ目のガードを調べ、24.3 は 25.0 より小さいので、2 つ目の文字列が返されます。

このような複雑な条件式を見ると命令型言語における巨大な **if/else** の連鎖を思い出します。ガードのほうがはるかに可読性が高いですよね。長ーい **if/else** の連鎖は顰蹙ものですが、問題の定義からしてそのような書き方が避けられない場合だってあります。そんなときは **if** よりガードを使いましょう。

[†5] ［訳注］このような場合分けの構文を**ガード**と呼ぶのは、条件を満たす場合にしか先の処理に進ませてくれない、衛兵さんみたいだからです！

[†6] ［訳注］紙面の都合からスペース 2 つでインデントしている箇所がいくつかあります。なおガードの 2 つ目では、\を使って、1 行を複数行に分けて入力しています。

たいていの場合、関数の最後のガードはすべてをキャッチする otherwise になっています。すべてのガードが False に評価されて、最後に全部をキャッチする otherwise もなかったなら、評価は失敗して次のパターンに移ります（これがパターンとガードの相性が良い理由です）。適切なパターンが見つからなければエラーが投げられます。

もちろん、ガードは複数の引数を取る関数にも使えます。bmiTell 関数を身長と体重を受け取るように変更して、BMI の計算もするように変更しましょう。

```
bmiTell :: Double -> Double -> String
bmiTell weight height
  | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
  | weight / height ^ 2 <= 25.0 = "You're supposedly normal.\
                                   \ Pffft, I bet you're ugly!"
  | weight / height ^ 2 <= 30.0 = "You're fat!\
                                   \ Lose some weight, fatty!"
  | otherwise                   = "You're a whale, congratulations!"
```

さて、僕の肥満度はどんなかな？

```
ghci> bmiTell 85 1.90
"You're supposedly normal. Pffft, I bet you're ugly!"
```

イェイ！ 肥満じゃない！ でも Haskell にブサイクに違いないと蔑まれました。知ったこっちゃない！

NOTE 初心者にありがちな間違いに、関数名と引数の後、ガードの前にイコール（ = ）を置いてしまうというものがあります。これはシンタックスエラーになります。

もう 1 つシンプルな例として、独自の max 関数を定義してみましょう。引数を 2 つ取って大きいほうを返す関数です。

```
max' :: (Ord a) => a -> a -> a
max' a b
    | a <= b    = b
    | otherwise = a
```

compare 関数もガードを使って実装できます。

```
myCompare :: (Ord a) => a -> a -> Ordering
a `myCompare` b
    | a == b    = EQ
    | a <= b    = LT
    | otherwise = GT
```

NOTE バッククオートによる中置記法は、関数呼び出しだけでなく、関数定義でも使えます。それによってコードが読みやすくなるときがあります。

```
ghci> 3 `myCompare` 2
GT
```

3.3 where?!

プログラムを書いていると、同じ値を何度も何度も計算するのを避けたいことがよくあります。何かを一度だけ計算して、その結果を格納しておくのはとても簡単です。命令型プログラミング言語なら、計算結果を変数に格納して解決するところでしょう。この節では、Haskell の **where** キーワードを使って計算の中間結果に名前をつける方法を学習します。

前の節では BMI 計算関数をこのように定義しました。

```
bmiTell :: Double -> Double -> String
bmiTell weight height
  | weight / height ^ 2 <= 18.5 = "You're underweight, you emo, you!"
  | weight / height ^ 2 <= 25.0 = "You're supposedly normal.\
                                   \ Pffft, I bet you're ugly!"
  | weight / height ^ 2 <= 30.0 = "You're fat!\
                                   \ Lose some weight, fatty!"
  | otherwise                   = "You're a whale, congratulations!"
```

上のコードでは、BMI 計算を 3 回繰り返しています。この値を **where** キーワードを使って変数に束縛し、その変数で BMI 計算をしている箇所を置き換えて、繰り返しを避けましょう。

```
bmiTell :: Double -> Double -> String
bmiTell weight height
  | bmi <= 18.5 = "You're underweight, you emo, you!"
  | bmi <= 25.0 = "You're supposedly normal.\
                   \ Pffft, I bet you're ugly!"
  | bmi <= 30.0 = "You're fat! Lose some weight, fatty!"
  | otherwise   = "You're a whale, congratulations!"
  where bmi = weight / height ^ 2
```

where キーワードをガードの後に置いて、1 つもしくは複数の変数や関数を定義できます。それら変数や関数の名前はどのガードからも見えます。BMI の計算方法を少し変えたくなっても、1 箇所を変えるだけで済みます。このテクニックにより値に名前がつくので可読性も上がります。しかも、値が一度しか計算されないようになるのでプログラムが速くなります。

なんだったら、もっとたくさん外に出してもかまいません。

```
bmiTell :: Double -> Double -> String
bmiTell weight height
  | bmi <= skinny = "You're underweight, you emo, you!"
  | bmi <= normal = "You're supposedly normal.\
                    \ Pffft, I bet you're ugly!"
  | bmi <= fat    = "You're fat! Lose some weight, fatty!"
  | otherwise     = "You're a whale, congratulations!"
  where bmi = weight / height ^ 2
        skinny = 18.5
        normal = 25.0
        fat = 30.0
```

NOTE　**where** ブロックの中のすべての変数のインデントが揃えてありますね。インデントがずれていると、Haskell は混乱してしまい、ブロックの範囲を正しく認識してくれません！

where のスコープ

where 節で定義した変数は、その関数からしか見えないので、他の関数の名前空間を汚染する心配がありません。複数の異なる関数から見える必要のある変数を定義したい場合は、グローバルに定義する必要があります。

また、**where** による束縛は関数の違うパターンの本体では共有されません。例えば、名前を引数に取り、その名前を認識できた場合には上品な挨拶を、そうでなければ下品な挨拶をする関数を考えてみます。こんな定義でどうでしょうか。

```
greet :: String -> String
greet "Juan" = niceGreeting ++ " Juan!"
greet "Fernando" = niceGreeting ++ " Fernando!"
greet name = badGreeting ++ " " ++ name
    where niceGreeting = "Hello! So very nice to see you,"
          badGreeting = "Oh! Pfft. It's you."
```

この関数は書いたとおりには動きません。**where** の束縛は違うパターンの関数本体で共有されず、**where** の束縛での名前は最後の本体からしか見えないからです。この関数を正しく動くようにするには、badGreeting と niceGreeting は次のようにグローバルに定義しなければなりません。

```
badGreeting :: String
badGreeting = "Oh! Pfft. It's you."

niceGreeting :: String
niceGreeting = "Hello! So very nice to see you,"

greet :: String -> String
greet "Juan" = niceGreeting ++ " Juan!"
greet "Fernando" = niceGreeting ++ " Fernando!"
greet name = badGreeting ++ " " ++ name
```

パターンマッチと where

where の束縛の中でもパターンマッチを使うことができます。BMI 関数の **where** 節を次のように書いてもかまいません。

```
...
    where bmi = weight / height ^ 2
          (skinny, normal, fat) = (18.5, 25.0, 30.0)
```

このテクニックの例として、ファーストネームとラストネームを受け取ってイニシャルを返す関数を書いてみましょう。

```
initials :: String -> String -> String
initials firstname lastname = [f] ++ ". " ++ [l] ++ "."
    where (f:_) = firstname
          (l:_) = lastname
```

関数の引数のところで直接パターンマッチすることもできますし、そのほうが短くて可読性も高くなると思いますが、この例のように **where** の束縛でもパターンマッチが使えます。

where ブロックの中の関数

where ブロックの中では定数だけでなく関数も定義できます。健康をテーマにしたプログラミングの続きで、体重と身長のペアのリストを受け取って BMI のリストを返す関数を作りましょう。

```
calcBmis :: [(Double, Double)] -> [Double]
calcBmis xs = [bmi w h | (w, h) <- xs]
    where bmi weight height = weight / height ^ 2
```

はい、出来上がり！ この例で `bmi` を定数ではなく関数として導入したのは、`calcBmis` 関数の引数に対して 1 つのの BMI を計算するのではなく、関数に渡されたリストの要素それぞれに対して、異なる BMI を計算する必要があるからです。

3.4 let It Be

let 式は **where** 節にとてもよく似ています。 **where** は関数の終わりで変数を束縛し、その変数はガードを含む関数全体から見えます。それに対して **let** 式は、どこでも変数を束縛でき、そして **let** 自身も式になります。しかし、 **let** 式が作る束縛は局所的

で、ガード間で共有されません。束縛を行う Haskell の他の構文と同じく、**let** 式でもパターンマッチが使えます。

let を使ってみましょう。次の関数は、円柱の表面積を高さと半径から求めます。

```
cylinder :: Double -> Double -> Double
cylinder r h =
    let sideArea = 2 * pi * r * h
        topArea = pi * r ^ 2
    in  sideArea + 2 * topArea
```

let 式は、**let** *bindings* **in** *expression* という形を取ります。**let** で定義した変数は、その **let** 式全体から見えます。

もちろん、**where** でも同じものが定義できます。では **let** と **where** の違いは何でしょう？ はじめのうちは、**let** は束縛を先に書いて式を後に書くけど **where** はその逆という違いしかないように思えるかもしれません。

本当の違いは、**let** 式はその名のとおり「式」で、**where** 節はそうじゃない、というところです。何かが「式である」というのは、それが値を持つということです。`"boo!"` は式、`3 + 5` も `head [1,2,3]` も式です。つまり、**let** 式は次のようにコード中のほとんどどんな場所でも使えるということです。

```
ghci> 4 * (let a = 9 in a + 1) + 2
42
```

let 式の便利な使い方をいくつか紹介します。

- **let** はローカルスコープに関数を作るのに使えます。

  ```
  ghci> [let square x = x * x in (square 5, square 3, square 2)]
  [(25,9,4)]
  ```

- **let** ではセミコロン（`;`）区切りを使えます。これは、複数の変数を 1 行で束縛したいときに便利です。インデントみたいに間延びした構文を使わずに済みます。

  ```
  ghci> (let a = 100; b = 200; c = 300 in a*b*c, ⇨
         let foo="Hey "; bar = "there!" in foo ++ bar)
  (6000000,"Hey there!")
  ```

- **let** 式とパターンマッチがあれば、あっという間にタプルを要素に分解してそれぞれ名前に束縛できます。

  ```
  ghci> (let (a, b, c) = (1, 2, 3) in a+b+c) * 100
  600
  ```

  ここでは、トリプル `(1,2,3)` を分解するのに **let** を使いました。最初の要素を `a`、2 つ目の要素を `b`、3 つ目の要素を `c` と呼んでいます。**in** `a+b+c`

の部分は、 **let** 式全体が値 a+b+c を持つ、と言っています。最後に、その値に 100 を掛けています。

- **let** 式はリスト内包表記の中でも使えます。この用法についてはすぐ後で詳しく見ていきます。

こんなに使い勝手が良いものなら、いつも **let** 式を使えばよいのではないでしょうか？ うーん。まず、 **let** 式は「式」であり、そのスコープに局所的なので、ガードをまたいでは使えません。また、関数の前ではなく**後ろ**で部品を定義するのが好きだから **where** を使うという人もいます。そのほうが関数の本体が名前と型宣言に近くなるので、コードが読みやすくなります。

リスト内包表記での let

体重と身長のペアのリストを計算する先ほどの例を、 **where** で関数を定義するのではなく、リスト内包表記中の **let** を使って書き換えてみましょう。

```
calcBmis :: [(Double, Double)] -> [Double]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2]
```

リスト内包表記が元のリストからタプルを受け取り、その要素を w と h に束縛するたびに、 **let** 式は w / h ^ 2 を変数 bmi に束縛します。それから bmi をリスト内包表記の出力として書き出しているだけです。

リスト内包表記の中の **let** を述語のように使っていますが、これはリストをフィルタするわけではなく、名前を束縛しているだけです。 **let** で定義された名前は、出力（ | より前の部分）とその **let** より後ろのリスト内包表記のすべてから見えます。このテクニックを使うと、次のように、肥満な人の BMI のみを返すように関数を変えることができます。

```
calcBmis :: [(Double, Double)] -> [Double]
calcBmis xs = [bmi | (w, h) <- xs, let bmi = w / h ^ 2, bmi > 25.0]
```

リスト内包表記の (w, h) <- xs の部分は**ジェネレータ**と呼ばれます。 **let** の束縛よりも前に定義されているので、変数 bmi はジェネレータからは参照できません。

GHCi での let

GHCi で直接関数や定数を定義するときは、 **let** 式の **in** の部分を省略できます。省略した場合、定義した名前はそれ以降の対話的なセッション全体から見えるようになります。

```
ghci> let zoot x y z = x * y + z
```

```
ghci> zoot 3 9 2
29
ghci> let boot x y z = x * y + z in boot 3 4 2
14
ghci> boot
<interactive>:1:0: Not in scope: `boot'
```

最初の行では **in** の部分を省略したので、GHCi は「この行では zoot は使わないんだな」と理解し、以降のセッションのために zoot を覚えておきます。しかし、2 つ目の **let** 式は **in** パートを含んでいて、boot に引数を渡して呼んでいます。**in** 部分が省略されていない **let** 式は、それ自体が値を表す式なので、GHCi はその値を表示したわけです。

3.5　case 式

case 式を使うと、変数の指定した値に対するコードブロックを評価できます。要するに **case** 式は、コード中のどこでもパターンマッチが使える構文です。多くの言語（C/C++ や Java）にも似たような case 文があるので、この概念にはお馴染みかもしれません。

Haskell の **case** 式は、その一歩先を行くものです。**case** 式という名前が示すとおり、**if**/**else** 式や **let** 式と同じく **case** も式なのです。さらに、**case** 式では変数の値に基づいて評価を分岐させるだけでなく、パターンマッチも使えます。

値を取り、パターンマッチし、その結果に基づいてコード片を評価するという点において、**case** 式は関数の引数に対するパターンマッチとかなり似ています。実のところ、あれは **case** 式の構文糖衣になっています。例えば、次の 2 つのコード片はまったく同じで、交換可能です。

```
head' :: [a] -> a
head' [] = error "No head for empty lists!"
head' (x:_) = x

head' :: [a] -> a
head' xs = case xs of [] -> error "No head for empty lists!"
                      (x:_) -> x
```

case 式の構文を示します。

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

実にシンプルです。式に合致した最初のパターンが使われます。 **case** 式に合致するパターンが見つからなかった場合、ランタイムエラーが発生します。

引数によるパターンマッチが使えるのは関数を定義するときだけですが、**case** 式はどこでも使えます。例えば、式の途中でパターンマッチをするのに使えます。

```
describeList :: [a] -> String
describeList ls = "The list is "
                  ++ case ls of [] -> "empty."
                                [x] -> "a singleton list."
                                xs -> "a longer list."
```

この例の **case** 式は次のように動作します。まず ls は空リストのパターンに対して調べます。もし ls が空なら、 **case** 式全体の値は "empty" になります。ls が空リストでなければ、1 要素のリストのパターンに対して調べます。このパターンマッチが成功したら、 **case** 式の値は "a singleton list" に評価されます。どちらでもなければ、すべてに合致するパターン xs が適用されます。最後に、 **case** 式の結果は文字列 "The list is" と連結されます。 **case** 式は値を表現しているので、文字列 "The list is" と **case** 式に対して ++ が利用できるのです。

関数定義でのパターンマッチは **case** 式と同じように使えるので、describeList は次のようにも定義できます。

```
describeList :: [a] -> String
describeList ls = "The list is " ++ what ls
    where what [] = "empty."
          what [x] = "a singleton list."
          what xs = "a longer list."
```

この関数は、構文こそ違いますが、前の例と同じように動作します。関数 what が ls を引数に呼び出されて、それからパターンマッチが行われます。関数が文字列を返すと、それが "The list is" と連結されます。

第4章
Hello 再帰!

この章では再帰について見ていくことにします。再帰が Haskell プログラミングでなぜ重要なのか、問題を再帰的に考えることによって、とても簡潔でエレガントな解法を見つける方法を学びます。

再帰とは、関数の定義の中で自分自身を呼び出す、という関数の定義の仕方です。その関数は、自分自身を呼び出すことになります。再帰が何なのかまだよく分からないなら、この段落をもう一度読んでください。(冗談ですよ！)

冗談はさておき、関数を再帰的に定義するには、解こうとする問題を同じ種類のより小さな問題に分解し、それら部分問題を解くことを考えます。必要なら、それらをさらに分解していきます。最終的には、問題の**基底部**、つまりこれ以上分解できなくて解を明示的に（再帰を使わずに）定義しなければならないケース（複数あるかも）に辿り着きます。

数学における定義には再帰がちょくちょく出てきます。例えば、**フィボナッチ数列**は次のように再帰的に定義できます。最初の 2 つのフィボナッチ数は $F(0) = 0$ と $F(1) = 1$ です。つまり、フィボナッチ数列の 0 番目と 1 番目の数はそれぞれ 0 と 1 である、と定義します。これらが基底部になります。

次に、0 と 1 以外のあらゆる自然数に対して、対応するフィボナッチ数を直前の 2 つのフィボナッチ数の和として定義します。つまり、$F(n) = F(n-1) + F(n-2)$ ということです。例えば、$F(3)$ は $F(2) + F(1)$ であり、これはさらに $(F(1) + F(0)) + F(1)$ に分解されます。再帰を使わずに定義されたフィボナッチ数だけになったので、これで無事に $F(3)$ は 2 であると言えます。

Haskell では再帰が重要です。なぜなら、Haskell では命令型言語のように計算をどうやってするかを指定するのではなく、求めるものが何であるかを宣言して計算を行うからです。Haskell の目的は、計算を実行するステップをコンピュータに示すことではなく、欲しい結果が何であるかを直接定義することであり、そのために再帰的な方法をよく使うのです。

4.1　最高に最高！

すでに Haskell にある関数を例に、どうすれば自分たちでその関数を書けるか、脳内のギアを「R」に入れて考えてみましょう。「R」はバック（reverse）ではなく再帰（recursion）ですよ。

`maximum` 関数は順序の付いた値（`Ord` 型クラスのインスタンス）のリストを受け取り、その中で一番大きな値を返します。これは、再帰を使うととてもエレガントに表現できます。

再帰的な解法の前に、命令的に定義したらどういう実装になるか考えてみることにしましょう。おそらく現在の最大値を保持するための変数を用意し、リストのすべての要素についてループして、現在の要素がこれまでの最大値よりも大きければ最大値を現在の値で更新していくことになると思います。ループが終了した時点で変数に保持されている値が結果の値になります。

では、再帰的に定義する方法を見ていきましょう。最初に基底部を定義しなければなりません。単一要素のリストに対する最大値は、その唯一の要素と同じ値です。では、もっと要素がたくさんある場合は？ その場合は、リストの先頭要素（head）と残りのリスト（tail）の最大値とでどちらが大きいかを調べます。再帰的に定義した `maximum'` 関数のコードはこうなります。

```
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list!"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

見てのとおり、パターンマッチは再帰関数の定義にもってこいです。合致と値の分解ができるので、最大値を見つけるという問題を、関連するいくつかのケースと部分問題とに簡単に分解できます。

最初のパターンでは、リストが空ならプログラムはクラッシュすると言っています。空リストの最大値なんて答えようがないので、これでつじつまが合っています。2 つ目のパターンでは、`maximum'` に単一要素のリストが渡されたらその唯一の要素を返すと言っています。

3 つ目のパターンで再帰が出てきます。リストは head と tail とに切り離されます。head を `x`、tail を `xs` と呼んでいます。それから、お馴染みの `max` 関数

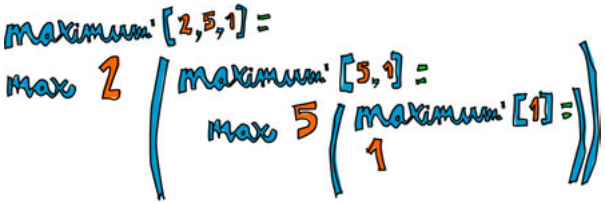
です。 max 関数は 2 つの引数を取り、大きいほうを返します。 x が xs の最大要素よりも大きければそれを返し、そうでなければ xs の最大要素を返します。でも、 maximum' はどうやって xs の最大要素を見つけるのでしょうか？ 単純です。自分自身を呼び出す、再帰です！

maximum' の動作を思い描けない人もいるでしょうから、具体的な例でやってみましょう。 maximum' を [2,5,1] で呼び出すと、最初の 2 つのパターンには合致しません。3 つ目のパターンに合致して、リストは 2 と [5,1] に分解され、maximum' が [5,1] に対して呼び出されます。

新しく呼び出した maximum' でも 3 つ目のパターンに合致して、再びリストが分解されます。今回は 5 と [1] に分解され、 maximum' が [1] に対して再帰的に呼ばれます。これは単一要素のリストなので、基底部に合致し、結果として 1 が返ります。

さて、今度は上位にさかのぼります。 5 と 1 を max 関数を使って比較します。1 は再帰的な呼び出しの返り値です。 5 のほうが大きいので、 [5,1] の最大値は 5 であると分かります。

最後に、 2 と、 [5,1] の最大値 5 とを比較します。これでもともとの問題の答が得られます。 5 は 2 よりも大きいので、 5 が [2,5,1] の最大値だと分かります。



4.2 さらにいくつかの再帰関数

再帰的な考え方が分かってきたところで、このやり方でもういくつか関数を実装してみましょう。 maximum と同じく Haskell にすでに存在する関数ですが、再帰筋群の再帰筋肉の再帰筋繊維を鍛えるべく自分たちで実装していきます。

replicate

手始めに replicate を実装します。 replicate は Int と値を取り、複製（replicate）という関数名のとおり、その値を（指定された数だけ）繰り返したリストを返します。例えば replicate 3 5 は、3 つの 5 からなるリスト

`[5,5,5]` を返します。

基底部を考えましょう。0 以下の回数だけ複製しろと要求されたときに何を返せばいいかは、すぐに分かります。何かを 0 回複製しようとしたら、空のリストが得られるべきです。負の数に対しても空のリストを返すと定義することにします。0 回よりも少ない回数の繰り返しには意味がないからです。

それ以外の一般の場合、`x` を `n` 回繰り返したリストは、`x` を head に、`x` を `n-1` 回繰り返したリストを tail にして構築したリストです。次のようなコードになります。

```
replicate' :: Int -> a -> [a]
replicate' n x
    | n <= 0    = []
    | otherwise = x : replicate' (n-1) x
```

負のケースに対応するために、ここではパターンではなくガードを使っています。

take

次は `take` を実装しましょう。この関数は指定されたリストから指定された数の要素を返します。例えば、`take 3 [5,4,3,2,1]` は `[5,4,3]` を返します。リストから 0 よりも少ない要素を取り出そうとした場合には空のリストを返します。空のリストから何かを取り出そうとした場合にも空のリストを返します。これら 2 つが基底部です。では関数を書いてみましょう。

```
take' :: Int -> [a] -> [a]
take' n _
    | n <= 0   = []
take' _ []     = []
take' n (x:xs) = x : take' (n-1) xs
```

0 以下のときに空のリストを返すという 1 つ目のパターンでは、リストの値を合致させるプレースホルダとして `_` を使っています。この条件では、リストの値が何であるかにはまったく関心がないからです。また、`otherwise` なしでガードを使っていることにも注意してください。これは `n` が 0 よりも大きいときは合致に失敗して次のパターンに移るようにするためです。

2つ目のパターンは、空のリストに対してはどんな数がきても空のリストを返すことを示しています。

3つ目のパターンは、リストを head と tail に分解します。head を x、tail を xs と呼びます。それから、「リストから n 要素取り出したものは、x を1つ目の要素にして、xs から n-1 要素を取り出したリストを残りの要素にしたリストと同じである」と定義しています。

reverse

reverse 関数はリストを取り、同じ要素で逆順のリストを返す関数です。またまた空リストが基底部になります。空のリストを逆にしたものは、やはり空のリストだからです。関数の残りの部分はどうしましょう？ 元のリストを head と tail に分解すれば、tail を逆順にして後ろに head をくっつけたものが逆順のリストになりますよね。

```
reverse' :: [a] -> [a]
reverse' [] = []
reverse' (x:xs) = reverse' xs ++ [x]
```

repeat

repeat 関数は要素を受け取り、その要素から無限リストを作る関数です。repeat の再帰的実装は実に簡単です。

```
repeat' :: a -> [a]
repeat' x = x : repeat' x
```

repeat 3 の呼び出しは、3 から始まり、無限個の 3 を tail とするリストを返します。なので、repeat 3 の呼び出しは 3 : repeat 3 に評価され、さらに 3 : (3 : (3 : repeat 3)) という具合に評価されていきます。repeat 3 の評価は終了しません。しかし、take 5 (repeat 3) は5つの 3 からなるリストに評価されます。これは本質的に、replicate 5 3 の呼び出しと似ています。

これは基底部のない再帰を利用して無限リストを作る良い例です。どこか途中で切ることだけは忘れずに。

zip

`zip` は第 1 章に出てきたリスト関数です。2 つのリストを引数に取り、これらを綴じ合わせ (zip) て返します。例えば `zip [1,2,3] [7,8]` は `[(1,7),(2,8)]` を返します (長いほうのリストは短いほうのリストの長さに切り詰められます)。

何かを空のリストと `zip` すると、単に空のリストが返ります。これが基底部になります。しかし、 `zip` は 2 つのリストを引数に取るので、実際には 2 つの基底部があります。

```
zip' :: [a] -> [b] -> [(a,b)]
zip' _ [] = []
zip' [] _ = []
zip' (x:xs) (y:ys) = (x,y) : zip' xs ys
```

最初の 2 つのパターンが基底部です。1 つ目か 2 つ目のリストが空なら、空のリストを返します。3 つ目のパターンは、「2 つのリストを `zip` したものとは、それぞれの head のペアに、それぞれの tail を `zip` したものをつなげたものである」と定義しています。

例えば、`zip'` を `[1,2,3]` と `['a','b']` で呼び出したら、この関数は `(1,'a')` を結果の最初の要素として構築し、それから `[2,3]` と `[b]` を `zip` して残りの結果を得ます。さらにもう 1 回の再帰呼び出しの後、 `[3]` と `[]` を `zip` しようとしますが、これは 1 つ目の基底部に合致します。それから最終結果が `(1,'a') : ((2,'b') : [])` と計算され、これは `[(1,'a'),(2,'b')]` と同じです。

elem

標準ライブラリにある関数をもう 1 つ実装してみましょう。 `elem` です。この関数は値とリストを受け取り、その値がリストに含まれるかを調べます。またもやこれも、空のリストが基底部です。空のリストは値を含まないので、どう考えても探している値を含みません。一般のケースでは、探したい値が幸運にも先頭にあるかもしれませんが、そうじゃない場合は tail にあるか調べる必要があります。これがそのコードです。

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' a [] = False
elem' a (x:xs)
    | a == x    = True
    | otherwise = a `elem'` xs
```

とても単純です。探しものが先頭になければ、tail を調べます。空のリストに辿り着いたなら、結果は `False` です。

4.3 クイック、ソート！

順序付けされた要素（数など）のリストをソートする問題は、自然と再帰的な解決方法へ行き着きます。リストの再帰的なソートにはいろいろなアプローチがありますが、最もクールな方法の 1 つである**クイックソート**を見ていくことにします。はじめにアルゴリズムがどのように動作するのかを見て、それから Haskell で実装していきます。

アルゴリズム

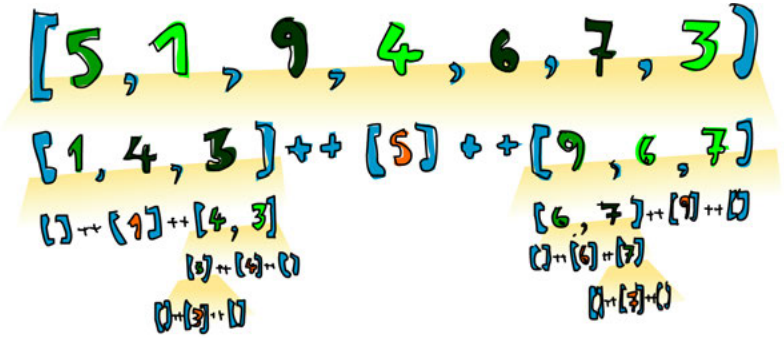
クイックソートのアルゴリズムは次のように動作します。ソートしたいリストを `[5,1,9,4,6,7,3]` としましょう。最初の要素である `5` を選択して、それから、残りのリストの中で `5` 以下の要素を左に置きます。次に、 `5` より大きい要素を右に置きます。この操作によって `[1,4,3,5,9,6,7]` というリストが得られます。

この例における `5` は**ピボット**（軸）と呼ばれます。他の要素と比較して左右に振り分けるときの軸になっているからです。ピボットに最初の要素を使う唯一の理由は、パターンマッチで簡単に取り出せるからです。実際にはどの要素をピボットにしてもかまいません[†1]。

次ページの図は、前の例に対してクイックソートがどのように動作するかを示しています。 `[5,1,9,4,6,7,3]` をソートしたいとき、まず、最初の要素をピボットとして選択します。それから、それを `[1,4,3]` と `[9,6,7]` でサンドイッチします。それが済んだら、 `[1,4,3]` と `[9,6,7]` をそれぞれ同じ方法でソートします。

`[1,4,3]` をソートするのに、最初の要素 `1` をピボットとして選択し、 `1` 以下の要素からなるリストを作ります。 `1` は `[1,4,3]` の中で一番小さい要素なので、これは空のリスト `[]` になります。ピボットの右に置くのは、 `1` より大きい要素からなる `[4,3]` です。今度は `[4,3]` をまた同じようにソートします。やはり最終的に空のリストになるまでバラバラにしてくっつけ直します。

[†1] ［訳注］所要時間の期待値を $O(N \log N)$ で抑えるためには、ピボットをランダムに選ぶなどの工夫が必要です。



こうしてアルゴリズムは 1 の右に配置するリストを返します。 1 の左は空リストなので、リスト [1,3,4] が得られました。これはソートされてます。そして 5 の左側にあります。

5 の右側も同様にソートすれば、完全なソート済みリスト [1,3,4,5,6,7,9] が得られるでしょう。

コード

クイックソートのアルゴリズムに詳しくなったところで、Haskell での実装に飛び込んでみましょう。

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerOrEqual = [a | a <- xs, a <= x]
        larger = [a | a <- xs, a > x]
    in  quicksort smallerOrEqual ++ [x] ++ quicksort larger
```

関数の型シグネチャ[†2]は quicksort :: (Ord a) => [a] -> [a] で、以前に見たように空のリストが基底部です。

x より小さいか等しい要素を左に置きたいので、その要素を取り出すのにリスト内包表記 [a | a <- xs, a <= x] を使っています。このリスト内包表記では、 xs （ピボット以外の全要素）から要素を取り出して、 a <= x の条件を満たす要素だけを残します。 x より大きい要素も同じ要領で求めています。

let 束縛を使って、2 つのリストに分かりやすい名前 smallerOrEqual と larger をつけています。最後に、リスト連結演算子（ ++ ）と quicksort 関数自身を再帰的に適用し、 smallerOrEqual をソートしたリスト、ピボット、larger をソートしたリストの順につないで最終的なリストを構築します。

[†2] ［訳注］型を表す文字列のこと。

作った関数が正しく動くか試運転してみましょう。

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the quick brown fox jumps over the lazy dog"
"        abcdeeefghhijklmnoooopqrrsttuuvwxyz"
```

ね、言ったとおりになったでしょ！

4.4 再帰的に考える

この章ではかなりたくさん再帰を使ってきました。気づいたかもしれませんが、再帰を書くには定跡があります。まず、再帰に頼らない自明な解を持つ基底部から始めます。例えば、「空のリストをソートしたものは空のリストである」。だって、ほかに考えられないもの！

それから、問題を 1 つもしくはそれ以上の部分問題に分解し、自分自身を適用することによって、それらの部分問題を再帰的に解きます。最後に、最終的な解を部分問題の解から構築します。例えばソートの場合なら、リストを 2 つのリストとピボットに分解します。それらのリストを再帰的にソートして、結果が得られたら、1 つにつなげて大きなソート済みリストにします。

再帰を使う際の定跡は、まず基底部を見極め、次に、解くべき問題をより小さな部分問題へと分割する方法を考えることです。基底部と部分問題さえ正しく選んだなら、全体として何が起こるかの詳細を考える必要はありません。部分問題の解が正しいという保証をもとに、より大きな最終問題の解を構築すればよいだけです。

第5章
高階関数

Haskell の関数は、引数として関数を取ったり返り値として関数を返したりできます。このような関数は**高階関数**と呼ばれます。高階関数は、問題を解決しプログラムを考察するための実に強力な手段であり、Haskell のような関数型プログラミング言語を使う際になくてはならないものです。

5.1 カリー化関数

Haskell のすべての関数は、公式には引数を 1 つだけ取ることになっています。しかし、これまでの章で複数の引数を取る関数を定義してきました。どうなっているんでしょうか？

実は、巧妙な仕掛けがあるんです！ 複数の引数を受け取れるかのように見えた関数は、実はすべて**カリー化**された関数だったのです。カリー化関数は、複数の引数を取る代わりに、常にちょうど 1 つの引数を取る関数です。カリー化関数が呼び出されると、その次の引数を受け取る関数を返します。その繰り返しです。

例を使って説明するのが一番でしょう。いつものように `max` 関数さんにご登場願います。 `max` 関数は、一見すると 2 つの引数をもらって大きいほうを返す関数のように見えます。例えば `max 4 5` という式を考えてみましょう。これは関数 `max` を、2 つの引数 `4` と `5` で呼び出しています。最初に、 `max` が値 `4` に適用されます。 `max` が `4` に適用されると、その実際の返り値は、 `5` に適用するための別の関数です。この関数が `5` に適用されて最終的な数値が返ります。結果として、次の 2 つの呼び出しは等価になります。

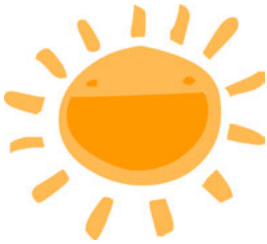
```
ghci> max 4 5
5
ghci> (max 4) 5
5
```

これがどのように動作するのか理解するために、 max 関数の型を調べてみましょう。

```
ghci> :t max
max :: (Ord a) => a -> a -> a
```

これは次のようにも書けます。

```
max :: (Ord a) => a -> (a -> a)
```



矢印 -> を型シグネチャに含むものはすべて関数です。 つまり、矢印の左側にあるものを引数に取り、右側にあるものを型とする値を返します。 a -> (a -> a) のようなものを見れば、これは a 型の値を引数に取る関数であり、「 a 型の値を引数に取り a 型の値を返す関数」を返すのだと分かります。

それで、これは何がうれしいのでしょうか？ ひらたくいうと、関数を本来より少ない引数で呼び出したときに**部分適用**された関数が得られることです。この**部分適用**された関数は、残りの変数を引数として取る関数です。例えば、 max 4 としたとしましょう。これは 1 引数の関数を返します。部分適用（関数を本来より少ない引数で呼び出すこと）を使うと、関数をその場でお手軽に作り出して、それを他の関数に渡せます。

次の小さい単純な関数を見てください。

```
multThree :: Int -> Int -> Int -> Int
multThree x y z = x * y * z
```

multThree 3 5 9 あるいは ((multThree 3) 5) 9 を呼び出すと実際には何が起こるでしょうか？ 最初に、 multThree が 3 に適用されます。スペースで区切られているからです。これにより「引数を 1 つ取って関数を返す関数」が返ります。それから、この関数が 5 に適用されます。これにより「引数を 1 つ取ってそれに 3 と 5 を掛けた数を返す関数」ができます。その関数が 9 に適用され、135 が最終的な結果になります。

関数は、何か材料を受け取って何かを作り出す小さな工場だと考えることができます。この比喩を使うと、「 multThree に数 3 を与えたら、数ではなくて、ちょっと小さくなった別の工場が出てきた」と言えます。そのちょっと小さな工

場は、数 5 を受け取り、また別の工場を作り出します。3 つ目の工場は数 9 を受け取り、最終結果である数 135 を作り出します。

この関数の型は次のようにも書けることを思い出してください。

```
multThree :: Int -> (Int -> (Int -> Int))
```

-> の前にある型（もしくは型変数）は関数が受け取る値の型で、後ろにあるのは返り値の型です。なので、この関数は Int 型の値を受け取り、(Int -> (Int -> Int)) という型を持つ関数を返します。同様に、**この**関数は Int 型の値を受け取り、Int -> Int という型の関数を返します。最後に、**この**関数は Int 型の値を受け取り、Int 型の値を返します。

本来より少ない引数で関数を呼び出すことによって新しい関数を作る例を見ていくことにしましょう。

```
ghci> let multTwoWithNine = multThree 9
ghci> multTwoWithNine 2 3
54
```

この例で、式 multThree 9 の結果は 2 引数の関数になります。この関数に multTwoWithNine という名前をつけています。 multThree 9 は 2 つの引数を取る関数だからです。もし引数が 2 つとも与えられたら、それらを掛け合わせて、それから 9 が掛けられます。 multTwoWithNine は multThree を 9 に適用したものだからです。

Int を引数に取って 100 と比較する関数を作りたいときはどうすればいいでしょうか？ これは次のように書けます。

```
compareWithHundred :: Int -> Ordering
compareWithHundred x = compare 100 x
```

例として、この関数を 99 で呼び出してみましょう。

```
ghci> compareWithHundred 99
GT
```

100 は 99 より大きいので、この関数は GT 、つまり「より大きい」を返します。

では、 compare 100 が何を返すか考えてみましょう。答は、「数を引数に取り 100 と比較する関数」です。これは上のサンプルで定義したものとまったく同じです。言い換えると、次の定義は前の定義と等価です。

```
compareWithHundred :: Int -> Ordering
compareWithHundred = compare 100
```

型の宣言は同じままです。なぜなら、 compare 100 は関数を返すからです。compare は (Ord a) => a -> (a -> Ordering) という型を持ちます。これを

100 に適用すると、「数を引数に取り Ordering を返す関数」が得られます。

セクション（という名のセクション）

中置関数に対しても、**セクション**という機能を使って部分適用することができます。中置関数をセクションするには、片側だけに値を置いて括弧で囲むだけです。これで引数を 1 つ、値を置かなかった側に取る関数ができます。つまらない例を 1 つ。

```
divideByTen :: (Floating a) => a -> a
divideByTen = (/10)
```

次のコードで分かるように、 divideByTen 200 は 200 / 10 もしくは (/10) 200 と等価です。

```
ghci> divideByTen 200
20.0
ghci> 200 / 10
20.0
ghci> (/10) 200
20.0
```

もう 1 つ例を見てみましょう。次の関数は与えられた文字が大文字かどうか調べます。

```
isUpperAlphanum :: Char -> Bool
isUpperAlphanum = (`elem` ['A'..'Z'])
```

セクションで唯一気をつけなければならないのは、 -（負の数、マイナス）演算子と同時に使うときです。(-4) は、セクションの定義から考えると、数を受け取ってそれから 4 を引く関数になるでしょう。しかし (-4) は利便性のために、マイナス 4 を意味することになっています。なので、引数から 4 を引き算する関数を作りたいときには、 subtract 関数を (subtract 4) のように部分適用します。

関数を表示する

今までのところ、部分適用した関数は何かしらの名前に束縛して、それから残りの引数を与えていました。しかし、関数そのものをターミナルに表示したことはありません。ここらでやってみませんか？ **let** で名前をつけたり、別の関数に渡したりせずに、 multThree 3 4 をそのまま GHCi に打ち込むと何が起こるでしょう？

```
ghci> multThree 3 4
<interactive>:1:0:
    No instance for (Show (a -> a))
      arising from a use of 'print' at <interactive>:1:0-12
    Possible fix: add an instance declaration for (Show (a -> a))
    In the expression: print it
    In a 'do' expression: print it
```

GHCi は、この式が a -> a の型の関数を生成したけど、それをどうやって画面に表示したらいいのか分からない、と言っています。関数は Show 型クラスのインスタンスではないので、いい感じに関数を表現する文字列が得られないのです。これは、例えば 1 + 1 のような式を入力した場合とは異なります。この場合 GHCi は、式の結果として 2 を計算し、それから 2 に対して show を呼んで、数のテキスト表現を得ます。 2 のテキスト表現は単に文字列 "2" なので、これがスクリーンに表示されます。

NOTE カリー化関数と部分適用の働きを完全に理解しておいてください。この 2 つは本当に重要です！

5.2 高階実演

すでに説明したとおり、Haskell では関数は別の関数を引数として受け取ることも、返り値として関数を返すこともできます。このコンセプトの実演として、関数を受け取り、それを 2 回適用する関数を書いてみましょう。

```
applyTwice :: (a -> a) -> a -> a
applyTwice f x = f (f x)
```

型宣言に気をつけてください。 -> は右結合なので、これまでの例では関数の型を宣言するときに括弧は必要ありませんでした。しかし、ここでの括弧は必須です。これは、関数の最初の引数が「1 つ引数を取り、同じ型の値を返す関数（ a -> a ）」だということを示しています。2 つ目の引数は何か a 型の値で、返り値も a 型です。 a は何の型、つまり Int でも String でも、あるいはもっと他の型でもかまいません。しかし、すべての a は同じ型でなければならないことに注意しましょう。

NOTE 今や読者の皆さんは、引数が複数あるように見える関数が実際には単一の引数を受け取り部分適用された関数を返すという舞台裏を知っています。でも話を単純にするために、引き続き関数が複数の引数を受け取るかのように説明することにします。

applyTwice 関数はとても単純です。単に引数 f を関数として使い、 f と x をスペースで区切り、 x にそれを適用しているだけです。それからその結果に f を再度適用します。この関数の動作例をいくつか示します。

```
ghci> applyTwice (+3) 10
16
ghci> applyTwice (++ " HAHA") "HEY"
"HEY HAHA HAHA"
ghci> applyTwice ("HAHA " ++) "HEY"
"HAHA HAHA HEY"
ghci> applyTwice (multThree 2 2) 9
144
ghci> applyTwice (3:) [1]
[3,3,1]
```

部分適用のすごさと実用性が歴然と分かります。1 引数の関数を渡す必要があるのなら、部分適用を使って 1 引数の関数を作り、それに渡せばいいのです。例えば、 + は 2 つの引数を受け取りますが、この例では、セクションを使った部分適用で 1 つだけ引数を与えています。

zipWith を実装する

高階プログラミングを使って、標準ライブラリにある zipWith という素晴らしく便利な関数を実装してみましょう。 zipWith は関数と 2 つのリストを引数に取り、2 つのリストの各要素にその関数を適用することで、2 つのリストを 1 つに結合します。これが実装です。

```
zipWith' :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = f x y : zipWith' f xs ys
```

最初に型宣言を見てみましょう。1 つ目の引数は、引数を 2 つ取り値を 1 つ返す関数です。その 2 つの引数は同じ型でも違う型でもかまいません。 zipWith' の 2 つ目と 3 つ目の引数はリストです。返り値もまたリストです。

最初のリストは a 型の値のリストでなければなりません。なぜなら、リストの結合に使う関数が第一引数として a という型を取るからです。2 つ目の引数は b 型の値のリストでなければなりません。これも同様に、結合に使う関数の第二引数が b という型だからです。結果は c 型の要素のリストになります。

NOTE　もし関数（特に高階関数）を書いていて、その型がよく分からなくなったら、型宣言を省略し、 :t を使って Haskell に推論させればいいことを思い出しましょう。

この関数は通常の zip 関数に似ています。基底部は同じですが、1 つ引数が増えています（結合に使う関数）。しかし、その引数は基底部では使わないので、 _

を使って無視しています。最後のパターンの関数の本体も zip に似ていますが、(x, y) の代わりに f x y を返します。

zipWith' 関数にはいろんな仕事をさせられます。これはほんの数例です。

```
ghci> zipWith' (+) [4,2,5,6] [2,6,2,3]
[6,8,7,9]
ghci> zipWith' max [6,3,2,1] [7,3,1,5]
[7,3,2,5]
ghci> zipWith' (++) ⇨
        ["foo ", "bar ", "baz "] ["fighters", "hoppers", "aldrin"]
["foo fighters","bar hoppers","baz aldrin"]
ghci> zipWith' (*) (replicate 5 2) [1..]
[2,4,6,8,10]
ghci> zipWith' (zipWith' (*)) ⇨
        [[1,2,3],[3,5,6],[2,3,4]] [[3,2,2],[3,4,5],[5,4,3]]
[[3,4,6],[9,20,30],[10,12,12]]
```

1 つの高階関数を実に多くの用途に使えることが分かりますね。

flip を実装する

今度は、これまた標準関数に含まれている flip を実装します。 flip 関数は、関数を引数に取り、元の関数と似ているけど最初の 2 つの引数が入れ替わった関数を返します。このようにして実装できます。

```
flip' :: (a -> b -> c) -> (b -> a -> c)
flip' f = g
    where g x y = f y x
```

型宣言から、 flip' は「a 型と b 型の値を引数に取る関数」を引数に取り、「b 型と a 型の値を引数に取る関数」を返します。なお、関数はデフォルトでカリー化されているため、2 つ目の括弧は実際には必要ありません。矢印 -> はデフォルトで右結合なので、 (a -> b -> c) -> (b -> a -> c) は (a -> b -> c) -> (b -> (a -> c)) 、あるいは (a -> b -> c) -> b -> a -> c と同じものです。上のコードでは、g x y = f y x と書きました。ということは、f y x = g x y もまた成り立ちますよね？ これを踏まえると、 flip' の実装をもっと簡潔にできます。

```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f y x = f x y
```

この新しいバージョンの flip' では、関数がカリー化されていることをうまく使っています。 flip' f を引数 y と x なしで呼び出したら、2 つの引数を取る、引数の入れ替わった f が返るでしょう。

flip された関数を、また別の関数に渡すこともよくあります。そんな場合でも、関数が完全に適用されたらどんな結果になるかをよく考えてみたり、その結

果を書き下してみたりすれば、高階関数を作るときにカリー化がうまく利用できるのです。

```
ghci> zip [1,2,3,4,5] "hello"
[(1,'h'),(2,'e'),(3,'l'),(4,'l'),(5,'o')]
ghci> flip' zip [1,2,3,4,5] "hello"
[('h',1),('e',2),('l',3),('l',4),('o',5)]
ghci> zipWith div [2,2..] [10,8,6,4,2]
[0,0,0,0,1]
ghci> zipWith (flip' div) [2,2..] [10,8,6,4,2]
[5,4,3,2,1]
```

zip 関数を flip' すると、1 つ目のリストがペアの 2 番目に、2 つ目のリストがペアの 1 番目に格納される、 zip のような関数が得られます。 flip' div 関数は 2 つ目の引数を 1 つ目の引数で割ります。なので flip' div に数 2 と 10 が渡されたら、これは div 10 2 の結果と同じになります。

5.3　関数プログラマの道具箱

関数プログラマがたった 1 つの値に対する演算をしたいことなんて、あまりありません。たいていは、数や文字や他の型のデータの集まりを受け取り、その集合を変換して結果を得たいものです。この節では、複数の値を使って仕事をするときに便利な関数を見ていくことにします。

map 関数

map 関数は、関数とリストを受け取り、その関数をリストのすべての要素に適用して新しいリストを生成します。これが定義です。

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

型シグネチャによると、map は a から b への関数と a のリストを受け取り、b のリストを返す型を持ちます。

map はいろいろな使い方のできる、とても多才な高階関数です。実例を示します。

```
ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
ghci> map (++ "!") ["BIFF", "BANG", "POW"]
["BIFF!","BANG!","POW!"]
ghci> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
```

```
ghci> map fst [(1,2),(3,5),(6,3),(2,6),(2,5)]
[1,3,6,2,2]
```

リスト内包表記を使って同じことができるのに気づいた人もいるでしょう。例えば、 map (+3) [1,5,3,1,6] は理論的には [x+3 | x <- [1,5,3,1,6]] と同じです。しかし、 map 関数を使うほうが読みやすくなります。 map の map といったものを扱うようになれば、なおさらです。

filter 関数

filter 関数は述語とリストを受け取り、そのリストの要素のうち、述語を満たすもののみからなるリストを返します（**述語**とは、何が true で何が false かを言う関数、つまり真理値を返す関数でしたね）。型シグネチャと実装を次に示します。

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

p x が True に評価されたら、その要素は新しいリストに含まれます。 False に評価された場合には含まれません。

filter の例をいくつか示します。

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
[5,6,4]
ghci> filter (==3) [1,2,3,4,5]
[3]
ghci> filter even [1..10]
[2,4,6,8,10]
ghci> let notNull x = not (null x) ⇨
        in filter notNull [[1,2,3],[],[3,4,5],[2,2],[],[],[]]
[[1,2,3],[3,4,5],[2,2]]
ghci> filter (`elem` ['a'..'z']) ⇨
        "u LaUgH aT mE BeCaUsE I aM diFfeRent"
"uagameasadifeent"
ghci> filter (`elem` ['A'..'Z']) ⇨
        "i LAuGh at you bEcause u R all the same"
"LAGER"
```

これらもすべて、 map 関数のときのように、リスト内包表記と述語を使って書けます。どんなときにリスト内包表記ではなく map と filter を使うべきか、ルールのようなものは何もありません。コードと文脈に応じて、単純に読みやすいほうを採用してください。

リスト内包表記で複数の述語を指定するようなケースを filter で書くときには、 filter を何度か適用するか、述語を論理関数 && でつないで指定します。以下は filter を 2 回適用する例です。

```
ghci> filter (<15) (filter even [1..20])
[2,4,6,8,10,12,14]
```

この例では、リスト [1..20] を受け取り、偶数だけ残すようにフィルタしています。その後、そのリストを filter (<15) に渡し、15 以上の数を取り除いています。リスト内包表記ならこうなります。

```
ghci> [x | x <- [1..20], x < 15, even x]
[2,4,6,8,10,12,14]
```

リスト内包表記で要素をリスト [1..20] から引き出し、条件を満たす数のみ結果のリストに含まれるようにしています。

第 4 章で定義した quicksort 関数を覚えていますか？ あのときは、ピボットより小さい（もしくは等しい）要素と大きい要素をフィルタするのにリスト内包表記を使っていました。 filter を使えば、同じものをもっと読みやすく書けます。

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerOrEqual = filter (<= x) xs
        larger = filter (> x) xs
    in  quicksort smallerOrEqual ++ [x] ++ quicksort larger
```

map と filter のさらなる例

別の例として、10 万以下の数のうち 3829 で割り切れる最大の数を探してみましょう。それには、解があると分かっている範囲から解になり得る集合をフィルタするだけです。

```
largestDivisible :: Integer
largestDivisible = head (filter p [100000,99999..])
    where p x = x `mod` 3829 == 0
```

はじめに、 100000 より小さいすべての数からなる降順のリストを作ります。それから、それを述語でフィルタします。数のリストは降順にソートされているので、条件を満たす最大の数は、フィルタされたリストの先頭の要素として現れます。なので、フィルタされたリストの head を取ります。残りのリストが有限か無限かはどうでもいい話です。Haskell は怠惰なので最初の条件に合う解が見つかれば評価は停止します。

次の例として、10000 より小さいすべての奇数の平方数の和を求めてみましょう。これには takeWhile という関数を使います。この関数は述語とリストを受け取り、リストの先頭から始めて述語の条件が満たされる限りリストの要素を返し続けます。条件に合わない要素が見つかったらリストを返すのをやめます。例えば、文字列の最初の単語をこんなふうにして取得できます。

```
ghci> takeWhile (/=' ') "elephants know how to party"
"elephants"
```

10000 より小さいすべての奇数の平方数の和を求めるために、まず、(^2) 関数を無限リスト [1..] に map することから始めます。それから、奇数の要素だけ残るようにフィルタします。その後で、 takeWhile を使って 10000 より小さい要素だけからなるリストを作ります。最後に、そのリストの和を求めます (sum 関数を使います)。すべては GHCi から 1 行でできるので、関数を定義する必要すらありません。

```
ghci> sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
166650
```

すごい！ 初期データ (すべての自然数を含む無限リスト) から始めて、map して、 filter して、必要な分だけ切り出して、最後にそれを足し合わせるだけ！

この例をリスト内包表記で書くと次のようになります。

```
ghci> sum (takeWhile (<10000) [m | m <- [n^2 | n <- [1..]], odd m])
166650
```

次はコラッツ列というものを扱う問題です。**コラッツの数列**は次のように定義されます。

- 任意の自然数から開始する
- 数が 1 ならば、終了
- 数が偶数なら、2 で割る
- 数が奇数なら、3 倍して 1 を足す
- 新しい値でこのアルゴリズムを繰り返す

この定義に従うと、ある数の列ができます。この列は最初の数が何であっても最終的には 1 に到達すると予想されています。例えば 13 から始めると、$13, 40, 20, 10, 5, 16, 8, 4, 2, 1$ という列が得られます ($13 \times 3 + 1 = 40$, $40/2 = 20, \ldots$)。13 から始めたコラッツ列の長さは 10 だと分かります。

解きたい問題はこれです。「1 から 100 までの数のうち、長さ 15 以上のコラッツ列の開始数になるものはいくつあるか？」

最初のステップは数列を生成する関数を書くことです。

```
chain :: Integer -> [Integer]
chain 1 = [1]
chain n
    | even n =  n : chain (n `div` 2)
    | odd n  =  n : chain (n * 3 + 1)
```

これは標準的な再帰関数の使い方です。すべての列は 1 で終わるので、基底部は 1 です。正しく動作するかテストしてみます。

```
ghci> chain 10
[10,5,16,8,4,2,1]
ghci> chain 1
[1]
ghci> chain 30
[30,15,46,23,70,35,106,53,160,80,40,20,10,5,16,8,4,2,1]
```

それでは実際に問題に答える関数 numLongChains を書きましょう。

```
numLongChains :: Int
numLongChains = length (filter isLong (map chain [1..100]))
    where isLong xs = length xs > 15
```

[1..100] を chain 関数で map し、数列を要素とするリストを得ます。その数列は、これもまたリストとして表現しています。それから、長さが 15 より大きいか調べる述語を使ってフィルタします。フィルタできたら、その結果のリストに数列がいくつあるかを数えます。

NOTE　この関数は、純粋に数学的な問題を解く関数なのに、 numLongChains :: Int というワードサイズの枠にとらわれた型を返しています。というのも、 length は Int を返すからです。もし、一般的な Num a 型を返したいなら、 length の結果に対して fromIntegral を使えばよいでしょう。

map 関数に複数の引数を与える

今まで map するのに使っていた関数は、1 引数のみを取る関数だけでした（ map （*2） [0..] みたいな）。しかし、複数の引数を取る関数で map することも可能です。例えば、 map （*） [0..] のようなことができます。この場合、関数 * は型 （Num a） => a -> a -> a を持ち、これがリストの各要素に適用されます。

すでに見たように、2 引数の関数に 1 引数だけを与えたら、1 引数を取る関数が返されるのでした。ゆえに、 [0..] を * で map したら、1 引数関数のリストが得られることになります。

例を挙げます。

```
ghci> let listOfFuns = map (*) [0..]
```

```
ghci> (listOfFuns !! 4) 5
20
```

関数のリストの 4 番目の要素を取り出すと、(4*) と等価な関数が得られます。それを 5 に適用すると、(4*) 5、つまり 4 * 5 と等しいものになります。

5.4 ラムダ式

ラムダ式とは、1 回だけ必要な関数を作るときに使う無名関数です。

通常、ラムダ式は高階関数に渡す関数を作るためだけに使われます。ラムダ式を宣言するには、バックスラッシュ (\) を書いて [†1]、それから関数の引数をスペース区切りで書きます。続けて -> 、最後に関数の本体を書きます。\ は思いっきり目を細めるとギリシャ文字のラムダ λ に見えますよね。普通、ラムダ式は括弧で囲みます。

前の節では、numLongChains の中で、filter に渡すだけの関数 isLong を定義するのに **where** 束縛を使っていました。これをラムダ式を使って書き換えるとこうなります。

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15)
                               (map chain [1..100]))
```

ラムダ式は式なので、このように関数に直接渡すことができます。式 (\xs -> length xs > 15) は、受け取ったリストの長さが 15 よりも大きいかどうかを返す関数を返します。

カリー化と部分適用の動作がよく分かってないと、必要ないところでラムダ式を使いがちです。例えば次の式は等価です。

```
ghci> map (+3) [1,6,3,2]
[4,9,6,5]
ghci> map (\x -> x + 3) [1,6,3,2]
[4,9,6,5]
```

[†1] ［訳注］日本語版 Windows 環境では\ではなく¥を使います。

(+3) と (\x -> x + 3) はどちらも引数に 3 を足す関数なので、これらは同じ結果を返します。しかし、この場合はラムダ式を作りたくありません。部分適用のほうがはるかに可読性が高いからです。

普通の関数と同じように、ラムダ式も任意の数の引数を取ることができます。

```
ghci> zipWith (\a b -> (a * 30 + 3) / b) [5,4,3,2,1] [1,2,3,4,5]
[153.0,61.5,31.0,15.75,6.6]
```

普通の関数と同じように、ラムダ式でもパターンマッチができます。唯一異なるのは、1 つの引数に対して複数のパターンを定義できないところです（同じ引数に対して [] と (x:xs) のパターンを作って合致するほうを選択する、といったもの）。

```
ghci> map (\(a,b) -> a + b) [(1,2),(3,5),(6,3),(2,6),(2,5)]
[3,8,9,8,7]
```

NOTE　もしラムダ式でパターンマッチが失敗したら、ランタイムエラーが発生します。気をつけて！

もう 1 つ興味深い例を見てみましょう。

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z

addThree' :: Int -> Int -> Int -> Int
addThree' = \x -> \y -> \z -> x + y + z
```

関数はデフォルトでカリー化されているので、これらの 2 つの関数は等価です。しかし addThree のほうが圧倒的に読みやすいです。2 つ目の実装は、カリー化を説明するための小道具程度にしかなりません。

NOTE　2 つ目の例で、ラムダ式が括弧で囲まれていないことに注意してください。ラムダ式を括弧なしで書いた場合、 -> の右側のすべてがそのラムダ式に属します。なので、このケースでは括弧を省略してタイプ数を節約しています。もちろん、括弧を付けるのがお好みなら付けてもかまいません。

とはいえ、カリー化の記法のほうが便利な場合もあります。個人的に flip 関数は次のように定義するのが一番読みやすいと思います。

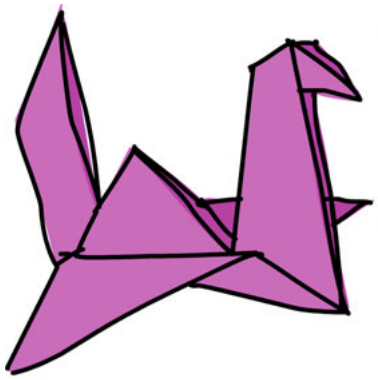
```
flip' :: (a -> b -> c) -> b -> a -> c
flip' f = \x y -> f y x
```

flip' f x y = f y x と書いてもまったく同じことですが、ラムダ式を使う書き方のほうが、新しい関数を生成するという flip の典型的な使い方をよく表しています。 flip で一番多い使い方は、引数として関数のみ、もしくは関数と引数を 1 つだけ渡し、生成された関数を map や zipWith に渡すという方法です。

```
ghci> zipWith (flip (++)) ["love you", "love me"] ["i ", "you "]
["i love you","you love me"]
ghci> map (flip subtract 20) [1,2,3,4]
[19,18,17,16]
```

自分が関数を定義するときもうまくラムダ式を使えば、その関数は部分適用され引数として別の関数に渡されるものなのだ、という意図を明確にできます。

5.5 畳み込み、見込みアリ！



第 4 章で再帰とたわむれていたころに戻りましょう。リストに対する再帰関数の多くは、ある共通のパターンに従っていました。「基底部は空リストとし、`x:xs` パターンを使ってリストを先頭要素と残りのリストに分解する」というパターンです。このパターンはとても頻繁に出てくるので、Haskell には**畳み込み**（fold）と呼ばれる便利な関数たちが用意されています。畳み込みを使うと、データ構造（例えばリスト）を単一の値にまとめることができます。

畳み込みを使えば、リストを 1 要素ずつ一回だけ走査してそれに基づいた結果を返すような関数なら何でも実装できます。リストを走査して何かを返したいときはいつでも、畳み込みを使うチャンスなのです。

畳み込み関数は、**2 引数関数**（2 つの引数を取る関数。`+` や `div` など）と、畳み込みに用いる値（**アキュムレータ**と呼ばれることが多い）の初期値、それに畳み込むリストを受け取ります。

リストは右からでも左からでも畳み込めます。畳み込み関数は、アキュムレータとリストの先頭（あるいは最後）の要素を引数として、与えられた 2 引数関数を呼び出します。その結果の値が、新しいアキュムレータになります。それから畳み込み関数は、新しいアキュムレータと新しく先頭（あるいは最後）になった要素を引数として再び 2 引数関数を呼び出し、また新しいアキュムレータを作ります。これを、リスト全体を走査しきって単一のアキュムレータの値になるまで繰り返します。

foldl で左畳み込み

最初に、 foldl 関数を見てみましょう。 foldl が「2 引数関数」「アキュムレータ」「畳み込み対象のリスト」の 3 つを引数に取ることを、型が教えてくれます。

```
ghci> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

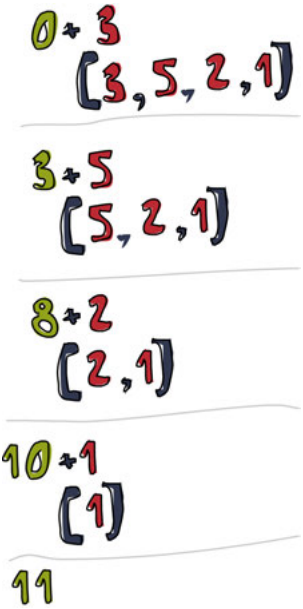
これはリストを左側から順に畳み込んでいくので、**左畳み込み**（left fold）と呼ばれます。左畳み込みでは、最初のアキュムレータとリストの先頭要素に 2 引数関数が適用されます。これが新しいアキュムレータを生成し、リストの次の要素と一緒に 2 引数関数が呼び出され、それが繰り返されます。

では、あからさまな再帰を使わず畳み込みを使って sum 関数を実装し直してみましょう。

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (\acc x -> acc + x) 0 xs
```

テストしてみます。

```
ghci> sum' [3,5,2,1]
11
```



この畳み込みで実際に何が起こっているのかを詳しく見てみましょう。 \acc x -> acc + x は 2 引数関数です。 0 はアキュムレータの初期値で、xs は畳み込みたいリストです。最初に 0 と 3 が、2 引数関数の引数 acc および x として渡されます。この場合は、2 引数関数は単なる足し算なので、この 2 つの数は足し合わされ、3 が新しいアキュムレータとして生成されます。次に、 3 と次のリストの値（ 5 ）が 2 引数関数に渡され、足し合わされ、 8 が新しいアキュムレータになります。同様にして、 8 と 2 が足し合わされて 10 になり、それから 10 と 1 が足し合わされて最終結果の 11 が生成されます。初めての畳み込み、おめでとう！

左の図は、畳み込みで何が起こるのかを、1 ステップずつ示したものです。 + の左側がアキュムレータの値です。どのようにしてリストが左側からアキュムレータによって消費されていく

のかが分かるでしょう（モグモグ、ムシャムシャ！）。関数がカリー化されていることを踏まえると、この実装はもっと簡潔に書くことができます。

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

ラムダ式 (\acc x -> acc + x) は (+) と同じです。 foldl (+) 0 はリストを取る関数を返すので、引数の xs は省略できます。一般に foo a = bar b a のような関数があった場合、カリー化のおかげで foo = bar b のように書き換えることができます。

foldr で右畳み込み

右畳み込み関数 foldr は、リストを右から順に処理する点を除いて[†2] 左畳み込みと似ています。また、右畳み込みに使う 2 引数関数では引数の順番が逆になっています。1 つ目の引数にリストの値、2 つ目にアキュムレータを渡します (これは直感的には、リストを右から畳み込むのでアキュムレータが右に位置するイメージです)。2 つを比べてみましょう。

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

畳み込みのアキュムレータの値（もしくは結果の値）は任意の型でかまいません。数でも真理値でも、はたまた新しいリストでも。例えば、右畳み込みで map 関数を実装してみましょう。アキュムレータはリストです。それに map した要素を 1 つずつくっつけていくことになります。もちろん最初の要素は空のリストです。

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (\x acc -> f x : acc) [] xs
```

[1,2,3] の各要素に (+3) を適用するときには、リストを右から走査します。最後の要素である 3 を取り、それに関数を適用して 6 を得ます。それをアキュムレータである [] の先頭に追加します。 6:[] は [6] なので、これが次のアキュムレータになります。次に、 2 に (+3) を適用して 5 が得られ、アキュムレータの先頭に追加（ : ）します。新しいアキュムレータの値は [5,6] になります。それから 1 に (+3) を適用してアキュムレータの先頭に追加し、最終的な結果として [4,5,6] を得ます。

[†2] ［訳注］リストは右からは走査できません！ foldr はあくまでリストを左から走査するけれど、結果だけ見ると右から折り畳んだように見えるということです。この点については、すぐ後で詳しく解説します。

もちろん、左畳み込みを使って次のように実装することもできなくはありません。

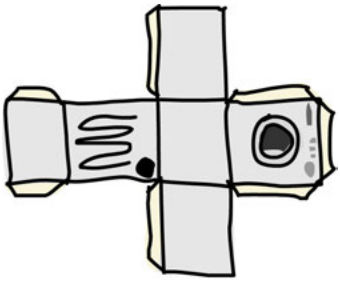
```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldl (\acc x -> acc ++ [f x]) [] xs
```

しかし、 ++ 関数は : よりもはるかに遅いので、リストから新しいリストを構築する際には普通は右畳み込みを使います。

右畳み込みと左畳み込みのもう 1 つの大きな違いは、右は無限リストに対しても動作するのに対し、左はダメだということです！

右畳み込みで関数をもう 1 つ実装してみましょう。ご存知のとおり、 elem 関数は値がリストに含まれるかどうかを調べる関数です。これを foldr で実装するには次のようにします。

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' y ys = foldr (\x acc -> if x == y then True else acc) False ys
```



この場合のアキュムレータは真理値です（畳み込みにおけるアキュムレータの型は結果の型と同じでしたね）。最初はリストに値がないものと考えるので、初期値は False になります。こうすれば空のリストに対しても正しい結果になります。空のリストに対する畳み込みは単に初期値が返るからです。

次に、現在の値が欲しい値なのか調べます。もしそうなら、アキュムレータを True にセットします。そうでなければ、アキュムレータを変更せずにそのまま返します。アキュムレータが False だったなら、現在の要素は探しているものではないからそのまま False にすべきだし、 True だったなら、それまで操作した部分にあったということなのでそのまま True にするべきだからです。

foldl1 と foldr1 関数

foldl1 と foldr1 関数は、それぞれ foldl と foldr に似ていますが、初期アキュムレータを明示的に与える必要がありません。その代わり、リストの先頭（あるいは末尾）の要素を初期アキュムレータとして使い、そこから畳み込みを始めます。これを使えば maximum 関数を次のように実装できます。

```
maximum' :: (Ord a) => [a] -> a
maximum' = foldl1 max
```

maximum を foldl1 を使って実装しました。初期アキュムレータは与えず、foldl1 がリストの最初の要素を初期アキュムレータとみなし、2 番目の要素から畳み込まれていきます。そのため、 foldl1 に必要なのは 2 引数関数と畳み込むリストだけです！ リストの先頭から始め、各要素をアキュムレータと比較します。要素がアキュムレータより大きければ、それを新しいアキュムレータにします。そうでなければ、古いアキュムレータをそのまま残します。そのために使っているのが、 foldl1 への 2 引数関数として渡している max です。 max は 2 つの値を受け取り大きいほうを返します。リストの畳み込みが終了するときには最大値だけが残ります。

foldl1 や foldr1 で定義した関数は、引数のリストに少なくとも 1 つ要素が含まれていることを前提にしているので、空リストに対して呼び出すとランタイムエラーが発生します。一方、 foldl と foldr で定義した関数は空リストに対しても正しく動きます。

NOTE 畳み込みを作る際には空リストに対する動作について考えましょう。空リストに対しては意味を成さない関数なら、 foldl1 や foldr1 を使える可能性があります。

いくつかの畳み込みの例

畳み込みがいかにパワフルかを示すために、畳み込みを使って標準ライブラリの関数をいくつか実装してみましょう。まずは reverse を書いてみることにします。

```
reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []
```

初期アキュムレータを空リストとし、元のリストを左から順にアキュムレータの先頭にくっつけていくことによって、リストを逆順にしています。

関数 \acc x -> x : acc は、引数が逆になっていることを除けば単なる : 関数です。したがって reverse' を次のように書くこともできます。

```
reverse' :: [a] -> [a]
reverse' = foldl (flip (:)) []
```

次は product を実装しましょう。

```
product' :: (Num a) => [a] -> a
product' = foldl (*) 1
```

数のリストの積を求めるには、アキュムレータを 1 から始め、 * 関数で左畳み込みをし、各要素をアキュムレータに掛け合わせていきます。

次は filter です。

```
filter' :: (a -> Bool) -> [a] -> [a]
filter' p = foldr (\x acc -> if p x then x : acc else acc) []
```

初期アキュムレータは空リストです。右から畳み込み、渡された述語 p で各要素を検査していきます。もし p x が True 、つまり現在の要素を結果の要素に残すべきなら、それをアキュムレータの先頭にくっつけます。そうでなければ古いアキュムレータをそのまま使い回します。

最後に last を実装します。

```
last' :: [a] -> a
last' = foldl1 (\_ x -> x)
```

リストの最後の要素を取得するのに foldl1 を使っています。リストの最初の要素から始めて、常に現在の値を新しいアキュムレータに置き換える 2 引数関数でリストを畳み込みます。最後まで到達したらアキュムレータを返します。これが最後の要素になっているはずです。

別の視点から見た畳み込み

右と左の畳み込みは、リストの要素に対する一連の関数適用と見ることもできます。2 引数関数 f と初期アキュムレータ z による右畳み込みがあるとします。この右畳み込みは、リスト [3,4,5,6] に対して、本質的には次のようなことを行います。

```
f 3 (f 4 (f 5 (f 6 z)))
```

f はリストの最後の要素とアキュムレータに対して呼び出され、それから最後から 2 番目の要素とアキュムレータに対して呼び出され、それが繰り返されます。

f として + 、初期アキュムレータとして 0 が与えられたとすると、これは次のようになります。

```
3 + (4 + (5 + (6 + 0)))
```

+ を前置関数として書けばこうになります。

```
(+) 3 ((+) 4 ((+) 5 ((+) 6 0)))
```

同様に、2 引数関数 g とアキュムレータ z に対する左畳み込みは次のように書けます。

```
g (g (g (g z 3) 4) 5) 6
```

2 引数関数として flip (:) 、初期アキュムレータとして [] アキュムレータが与えられたとすると、これはリストの反転であり、次と等価になります。

```
flip (:) (flip (:) (flip (:) (flip (:) [] 3) 4) 5) 6
```

そして実際、これを実行すれば [6,5,4,3] が得られます。

無限リストを畳み込む

畳み込みを一連の関数適用として考えると、foldr が無限リストに対して完璧に正しく動作する理由が見えてきます。 and を foldr で実装し、前の節のように一連の関数適用として書き下してみましょう。遅延評価の Haskell で foldr が無限リストに対して動作する仕組みが分かるはずです。

and 関数は、 Bool 値のリストを引数に取り、どれか 1 つが False ならば False を、そうでなければ True を返す関数です。リストを右から走査し、True を初期アキュムレータとします。2 引数関数としては、全部が True だったときに限り True が返ってほしいので、 && を使います。 && 関数は、2 つの引数のどちらかが False だったときに False を返すので、 False を含むリストを操作すればアキュムレータは False にセットされ、残りの要素が True だったとしても最終的な結果は False になります。

```
and' :: [Bool] -> Bool
and' xs = foldr (&&) True xs
```

foldr の動作を知っていれば、 and' [True,False,True] が次のように評価されると分かります。

```
True && (False && (True && True))
```

最後の True は初期アキュムレータで、最初の 3 つの Bool の値はリスト [True,False,True] から得られるものです。これを評価すれば False が得られます。

さて、無限リストではどうなるでしょうか？ repeat False は無限個の False からなる無限リストです。これを書き出してみると次のようになります。

```
False && (False && (False && (False ...
```

Haskell は遅延評価なので、本当に必要な部分だけを計算します。そして && 関数は、両方の引数が True だった場合に限って True を返すので、最初の引数が False であれば 2 つ目の引数を無視します。

```
(&&) :: Bool -> Bool -> Bool
True && x = x
False && _ = False
```

この場合だと、 False の終わらないリストが 2 つ目のパターンに合致します。そして、Haskell はその無限リストの残りの部分を評価することなく False を

返すのです。

```
ghci> and' (repeat False)
False
```

foldr は、2 番目の引数を常に評価するとは限らないような 2 引数関数が与えられた場合、無限リストに対してもうまく動作します。例えば && は、1 つ目の引数が False ならば 2 つ目の引数を無視します。

スキャン

scanl と scanr 関数は、foldl と foldr に似ていますが、アキュムレータの中間状態すべてをリストとして返します。 scanl1 と scanr1 関数は foldl1 と foldr1 のアナロジーです。関数の実際の使用例を示します。

```
ghci> scanl (+) 0 [3,5,2,1]
[0,3,8,10,11]
ghci> scanr (+) 0 [3,5,2,1]
[11,8,3,1,0]
ghci> scanl1 (\acc x -> if x > acc then x else acc) [3,4,5,3,7,9,2,1]
[3,4,5,5,7,9,9,9]
ghci> scanl (flip (:)) [] [3,2,1]
[[],[3],[2,3],[1,2,3]]
```

scanl は結果の最終要素に最終結果が入ります。 scanr はリストの先頭に結果が入ります。

畳み込みで実装できるような関数の途中経過をモニターしたいときにスキャンが使えます。スキャンを使った例として、次の問題に答えてみましょう。「自然数の平方根を小さいものから足していったとき、1000 を超えるのは何個目？」

すべての自然数の平方数を得るには、単純に map sqrt [1..] とするだけです。和を求めるのに畳み込みを使うこともできますが、加算の途中経過が知りたいのでスキャンを使います。スキャンを使えばどこまでが 1000 より小さいか調べられます。

```
sqrtSums :: Int
sqrtSums = length (takeWhile (<1000) (scanl1 (+) (map sqrt [1..]))) + 1
```

ここでは filter ではなく takeWhile を使っています。 filter だと 1000 以上の要素を見つけてもリストの走査をやめないからです。ここではリストが昇順だと分かっているので、 filter ではなく takeWhile を使って最初に和が 1000 より大きくなったところで走査を終了できます。

スキャンリストの最初の和は 1 です。2 番目は $1+2$ の平方根です。3 番目は、それに 3 の平方根を足したものです。1000 より小さいところに x 個の和があるなら、$x+1$ 要素の和は 1000 を超えるでしょう。

```
ghci> sqrtSums
131
ghci> sum (map sqrt [1..131])
1005.0942035344083
ghci> sum (map sqrt [1..130])
993.6486803921487
```

やった！ 正解でした。最初の 130 個の平方根を足したものは 1000 を下回り、131 個だと超えています。

5.6 $ を使った関数適用

今度は $ 関数、またの名を**関数適用演算子**について見ていきましょう。まずは定義を見てみます。

```
($) :: (a -> b) -> a -> b
f $ x = f x
```

何だこりゃ。とんだ役立たずの関数だぞ？ 単なる関数適用じゃないか！ それでだいたい合っていますが、それだけではないのです。普通の関数適用（2 つのものの間に空白を置く）は非常に高い優先順位を持っていますが、$ 関数は最も低い優先順位を持ちます。スペースを用いた関数適用は左結合（f a b c は ((f a) b) c を意味します）ですが、$ による関数適用は右結合です。

それが何の役に立つのかって？ 括弧の数を少なくしたいとき、たいていはこの関数が役立ちます。例えば sum (map sqrt [1..130]) という式を考えてみましょう。$ は優先順位が低いので、この式を sum $ map sqrt [1..130] と書き換えることができます。$ が出てきたら、その右側の式が左側の関数に引数として渡されます。

sqrt 3 + 4 + 9 はどうでしょうか？ これは 3 の平方根と 4 と 9 を足し合わせます。そうではなく、3 + 4 + 9 の平方根が欲しい場合は sqrt (3 + 4 + 9) と書かなければなりません。$ を使えば sqrt $ 3 + 4 + 9 のように書けます。$ は、はるか右に閉じ括弧のある開き括弧だと考えることもできるでしょう。

もう 1 つ例を見てみましょう。

```
ghci> sum (filter (> 10) (map (*2) [2..10]))
80
```

ホゥー、括弧がいっぱいだ！ ひどい！ この式は [2..10] の各要素に (*2) を適用し、その結果を 10 よりも大きいもののみを保持するようにフィルタして、最後に足し合わせています。

これをもう少し目に優しく書き換えるのに $ 関数を使えます。

```
ghci> sum $ filter (> 10) (map (*2) [2..10])
80
```

$ 関数は右結合なので、f $ g $ x は f $ (g $ x) と同じことです。これを踏まえて、さらに次のように書き換えることができます。

```
ghci> sum $ filter (> 10) $ map (*2) [2..10]
80
```

括弧を削除する話とは別に、$ は関数適用それ自身を関数として扱えるようにするために使えます。これにより、例えば関数適用をリストに対して map するようなことができます。

```
ghci> map ($ 3) [(4+), (10*), (^2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

この例は、関数 ($ 3) がリストに対して map されます。($ 3) という関数は、関数を引数に取って、その関数を 3 に適用する関数だと考えられます。したがって、リスト中のすべての関数が 3 に適用され、上のような結果が得られます。

5.7　関数合成

数学における**関数合成**は $(f \circ g)(x) = f(g(x))$ のように定義されます。これは、2 つの関数を合成したものは、まず 1 つの関数を呼び出し、それからもう 1 つの関数にその結果を渡して呼び出したものに等しい、という意味です。

Haskell の関数合成もこれとほとんど同じです。次のように定義される「 . 」関数を使って関数合成ができます。

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

型宣言に注意してください。f は引数として、g の返り値の型と同じ型の値を受け取らなければなりません。なので、合成された関数は、g が受け取る型と同じ型の引数を受け取り、f が出力する型と同じ型の結果を返します。例えば式 negate . (*3) は、数を受け取り、それを 3 倍して、それから符号反転する関数を返します。

関数合成の用途としては、他の関数に渡す関数をその場で作るというものがあります。もちろんラムダ式を使ってもいいですが、たいていは関数合成のほうが明快で簡潔です。

例えば、数のリストがあって、その全部を負の数にしたいとします。各要素の絶対値を取ってから符号反転するという方法が考えられます。

```
ghci> map (\x -> negate (abs x)) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

ラムダ式が関数合成の結果のように見えませんか。関数合成を使えばこう書き換えることができます。

```
ghci> map (negate . abs) [5,-3,-6,7,-3,2,-19,24]
[-5,-3,-6,-7,-3,-2,-19,-24]
```

素晴らしい！ 関数合成は右結合なので、一度にたくさんの関数を合成できます。式 f (g (z x)) は (f . g . z) x と等価です。これを踏まえて、次の汚いコードを何とかしてみましょう。

```
ghci> map (\xs -> negate (sum (tail xs))) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

次のようにとってもきれいに書き直せます。

```
ghci> map (negate . sum . tail) [[1..5],[3..6],[1..7]]
[-14,-15,-27]
```

negate . sum . tail は、リストを受け取り、それに tail を適用し、それからその結果に sum を適用し、最後にその結果に negate を適用する関数です。なので、これは前の例のラムダ式と等価です。

多引数関数の関数合成

複数の引数を取る関数を合成するときはどうすればいいでしょう？ 普通は、残り 1 引数になるまで部分適用しないと関数合成できません。次の式で考えてみましょう。

```
sum (replicate 5 (max 6.7 8.9))
```

この式は次のように書き換えられます。

```
(sum . replicate 5) (max 6.7 8.9)
```

このように書いても同じです。

```
sum . replicate 5 $ max 6.7 8.9
```

関数 replicate 5 が max 6.7 8.9 の結果に適用され、それからその結果に sum が適用されます。 replicate 関数を部分適用して 1 つだけ引数を取るようにしている点に注目してください。 max 6.7 8.9 の結果が replicate 5 に渡され、数のリストが結果の値となり、さらにそれが sum に渡されます。

たくさん括弧がある式を関数合成を使って書き直したいなら、まずは一番内側の関数とその引数を書き出すことから始めましょう。それから $ をその前に置いて、その前に置かれていた関数から最後の引数を取り除き、間にドットを置いて合成します。次の式は、

```
replicate 2 (product (map (*3) (zipWith max [1,2] [4,5])))
```

次のように書き直せます。

```
replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
```

どうやって最初の例を書き換えたのでしょうか？ まず、閉じ括弧の集まりの直前を見て、右端にある関数とその引数を見つけます。この例では zipWith max [1,2] [4,5] です。これを取り出して書きつけておきます。

```
zipWith max [1,2] [4,5]
```

それから、 zipWith max [1,2] [4,5] にどの関数が適用されているかを調べます。それは map (*3) だと分かります。そこで、これとさっき書いた式の間に $ を書きます。

```
map (*3) $ zipWith max [1,2] [4,5]
```

では、合成を開始します。どの関数が適用されるか調べます。すると product が適用されているので、 map (*3) とこれを合成します。

```
product . map (*3) $ zipWith max [1,2] [4,5]
```

それから最後に replicate 2 が適用されているので、次のように書き換えることができます。

```
replicate 2 . product . map (*3) $ zipWith max [1,2] [4,5]
```

式の終わりに 3 つも括弧があったなら、今示したような手順で関数合成を使って書き直すチャンスです。関数合成演算子を 2 つ使って書き直せるでしょう。

ポイントフリースタイル

関数合成のもう 1 つの一般的な使い道は、**ポイントフリースタイル**[†3] で関数を定義するというものです。例えばこのような関数を書いたとしましょう。

```
sum' :: (Num a) => [a] -> a
sum' xs = foldl (+) 0 xs
```

イコールの両側とも右端が xs です。関数はカリー化されているので、この両側の xs は省略できます。 `foldl (+) 0` を呼び出すとリストを受け取る関数が作り出されるからです。このようにして関数をポイントフリースタイルで書くことができます。

```
sum' :: (Num a) => [a] -> a
sum' = foldl (+) 0
```

もう 1 つの例として、次の関数をポイントフリースタイルで書いてみましょう。

```
fn x = ceiling (negate (tan (cos (max 50 x))))
```

両方の右端に x がありますが、括弧で囲まれているので取り除けません。`cos (max 50)` としてしまっては、関数のコサインを取るという、意味を成さないコードになってしまいます。ここで**できる**のは fn を関数合成で表現することです。

```
fn = ceiling . negate . tan . cos . max 50
```

素晴らしい！ ポイントフリースタイルにすると読みやすく簡潔になることが多々あります。データよりも関数に目がいくようになり、どのようにデータが移り変わっていくかではなく、どんな関数を合成して何になっているかを考えやすくなるからです。単純な関数から始め、関数合成を糊として使うことにより、より複雑な関数を作り出せばよいのです。

とはいえ関数が複雑になりすぎると、ポイントフリースタイルでは可読性が悪くなることもあります。このため、関数合成のチェインはあまり長くしないようにしましょう。 **let** を使って途中の結果にラベルを与え、問題を小さな問題に分解すれば、読む人にとって分かりやすいコードになります。

この章の前半で、奇数の平方数で 10000 より小さいものの総和を求める問題を解きました。これを関数にすると次のようになります。

```
oddSquareSum :: Integer
oddSquareSum = sum (takeWhile (<10000) (filter odd (map (^2) [1..])))
```

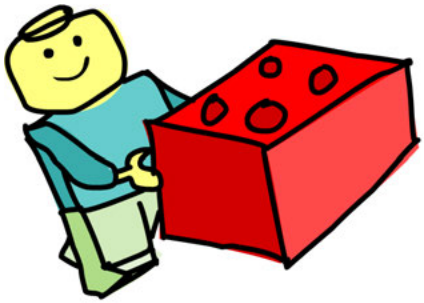
関数合成の知識を使うと次のようにも書けます。

[†3] ［訳注］**ポイント**とは、`fn x = f (g x)` のような関数定義に登場する一時変数 x のことです。このポイントを使わないで関数を定義するスタイルなので、ポイントフリースタイルというわけです。

```
oddSquareSum :: Integer
oddSquareSum = sum . takeWhile (<10000) . filter odd $ map (^2) [1..]
```

はじめのうちは少し風変わりなスタイルに見えるかもしれませんが、すぐに慣れていくことでしょう。こちらの書き方には括弧が少ないので視覚的ノイズも少なくなっています。このコードを見て「filter odd が map (^2) [1..] に適用され、それから takeWhile (<10000) がその結果に適用され、最後にその結果に sum が適用される」と読めるようになります。

第6章
モジュール



Haskell のモジュールは、いくつかの関数や型、型クラスなどを定義したファイルです。Haskell のプログラムはモジュールの集合です[†1]。

モジュールには複数の関数と型を定義でき、そのうちのいくつか、もしくはすべてをエクスポートできます。エクスポートとは、モジュールの中のものを外の世界からも見えるように、使えるようにすることです。

コードを複数のモジュールに分割することにはたくさんの利点があります。十分に汎用のモジュールなら、エクスポートする関数を多くの異なるプログラムで使えます。自分のコードを相互に強く依存しない（疎結合ともいいます）モジュールに分割しておけば、後でモジュールごとに再利用できます。コードを複数の部分に分割すれば、それだけ管理もしやすくなります。

Haskell の標準ライブラリは複数のモジュールに分割されていて、それぞれのモジュールに含まれる関数と型には何らかの関係があり、共通の目的で結び付いています。リストを操作するためのモジュール、並行プログラミングのためのモジュール、複素数を扱うためのモジュールなどがあります。これまでの章で使ってきたすべての関数、型、型クラスはデフォルトでインポートされる `Prelude` というモジュールの一部です。

[†1] ［訳注］Haskell には便利なモジュールがいっぱいあります！Hackage（hackage.haskell.org/packages）は Haskell のパッケージの巨大な動物園です。viii ページどおりに Haskell Platform をインストールしてあるなら、`cabal` コマンドが使えるはず。「`cabal install` パッケージ名」というコマンド一発で、さまざまなパッケージをインストールできますよ！

この章では便利なモジュールとその関数をいくつか見ていきます。でも、その前にモジュールをインポートする方法を覚えないと始まりません。

6.1　モジュールをインポートする

Haskell のソースコードからモジュールをインポートする構文は **import** ModuleName です。インポート構文はすべての関数定義よりも前に書く必要があります。そのため、インポート文は普通ファイルの先頭にあります。複数のモジュールをインポートしたい場合は、 **import** 文を 1 行に 1 つずつ書いてください。

便利なモジュールの例として Data.List を紹介しましょう。このモジュールはリストに対する関数をエクスポートしています。このモジュールの関数を使って、リストに一意な要素がいくつあるか数える関数を作ってみましょう。

```
import Data.List

numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
```

Data.List をインポートすると、 Data.List がエクスポートするすべての関数が使えるようになり、スクリプトのどの場所からでも呼べるようになります。 nub は Data.List がエクスポートする関数の 1 つで、リストから重複する要素を取り除く関数です。（ここで、さっき学んだポイントフリースタイルが登場しています！ length . nub は length と nub の関数合成で、\xs -> length (nub xs) と等価な関数であることを思い出してください。）

NOTE　探している関数がどこにあるのか知りたいときは、Hoogle（http://www.haskell.org/hoogle/）を使うとよいでしょう。関数名、モジュール名、はては型シグネチャからも検索できる、とても素晴らしい Haskell 検索エンジンです。

モジュールで定義された関数は、GHCi からも使えます。GHCi の中から Data.List がエクスポートしている関数を呼びたいなら、次のようにタイプします[†2]。

```
ghci> :m + Data.List
```

[†2] ［訳注］現在の GHC では、GHCi 上で import 文を利用することもできます。この import 文は、スクリプト中で利用する場合とまったく同じように使うことができます。

```
Prelude> import Data.List (nub, sort)
Prelude Data.List> import qualified Data.Map as M hiding (foldl)
Prelude Data.List M>
```

GHCi から複数のモジュールにアクセスしたい場合、 `:m +` を何回もタイプする必要はありません。複数のモジュールを一度にロードしたいなら次のようにします。

```
ghci> :m + Data.List Data.Map Data.Set
```

なお、モジュールをインポートするスクリプトをロードしている場合には、そのモジュールにアクセスするために `:m +` を使う必要はありません。モジュール中の特定の関数のみが必要な場合には、その関数のみを選んでインポートできます。例えば、 `Data.List` から `nub` と `sort` のみをインポートしたいなら、次のようにします。

```
import Data.List (nub, sort)
```

読み込みたくない関数を指定し、それ以外を全部インポートすることもできます。これは、複数のモジュールが同じ名前の関数をエクスポートしていて、不要なほうを取り除きたいときなどに使います。例えば、すでに `nub` という名前の関数が存在するので、 `Data.List` から `nub` を除くすべての関数をインポートしたい場合は、次のようにします。

```
import Data.List hiding (nub)
```

名前の競合を避けるもう 1 つの方法は、**修飾付きインポート**（qualified インポート）です。 `Data.Map` モジュールを例に見てみましょう。このモジュールは、キーに対応する値を検索するためのデータ構造を提供します。 `filter` や `null` のような、 `Prelude` にある関数と同じ名前の関数をたくさんエクスポートしています。なので、 `Data.Map` をインポートして `filter` を呼び出すと、Haskell はどちらの関数を使えばいいか分からなくなってしまいます。これを解決するには次のように修飾付きインポートします。

```
import qualified Data.Map
```

この状態で `Data.Map` の `filter` 関数を参照したいなら、 `Data.Map.filter` を使わなければなりません。単に `filter` とタイプすると、これはお馴染みの `filter` 関数のことを指します。でも、 `Data.Map` をすべての関数の前に付けるのはうんざりです。そこで、修飾付きインポートしたモジュールには何かしら短い別名をつけられるようになっています。

```
import qualified Data.Map as M
```

これで、 `Data.Map` の `filter` 関数を参照するには単に `M.filter` とタイプするだけで済むようになりました。

これまで見てきたとおり、修飾付きインポートしたモジュールの関数を参照するには、M.filter のように「.」が使われます。しかし、「.」は関数合成演算子としても使います。Haskell はどうやってこれを区別しているのでしょうか？「.」が qualified されたモジュール名と関数名の間に空白を開けずに置かれた場合には、インポートされた関数であるとみなされます。そうでなければ関数合成として扱われます[†3]。

NOTE Haskell について新しい知識を仕入れる最良の方法は、標準ライブラリのドキュメントにアクセスし、そのモジュールと関数を散策することです。各モジュールのソースコードを見ることもできます。モジュールのソースコードをいくつか読めば、Haskell に対する確かな感覚が身に付くでしょう。

6.2　標準モジュールの関数で問題を解く

標準ライブラリのモジュールには、Haskell でのコーディングライフを豊かにしてくれるたくさんの関数が用意されています。さまざまな Haskell のモジュールの関数を使っていかにして問題を解くか、実例を見ていくことにしましょう。

単語を数える

単語がたくさん含まれた文字列があって、各単語が何回現れるかを求めたいとします。最初に使うのは、Data.List モジュールに含まれる words 関数です。words 関数は、文字列を空白で区切られた文字列（単語）のリストに変換します。実行例を示します。

```
ghci> words "hey these are the words in this sentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> words "hey these           are    the words in this sentence"
["hey","these","are","the","words","in","this","sentence"]
```

それから、これも Data.List モジュールに含まれる group 関数を使います。これは同じ単語をグループ化するのに使います。この関数はリストを引数に取り、隣接する要素が同じものをまとめます。

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

では、同じ要素が隣接していない場合はどうなるでしょうか？

[†3] ［訳注］モジュール名は大文字で始まり、関数名は小文字で始まることに注意。

```
ghci> group ["boom","bip","bip","boom","boom"]
[["boom"],["bip","bip"],["boom","boom"]]
```

同じリストに含まれているのに、"boom" を含むリストが 2 つ得られました。どうすればいいのでしょうか？ 単語のリストをあらかじめソートしておきましょう！ そのためには sort 関数を使います。これは Data.List に含まれていて、順序付けされた要素からなるリストを受け取り、昇順に並び替えた新しいリストを返します。

```
ghci> sort [5,4,3,7,2,1]
[1,2,3,4,5,7]
ghci> sort ["boom","bip","bip","boom","boom"]
["bip","bip","boom","boom","boom"]
```

文字列がアルファベット順にソートされていることに気づいたでしょうか。

レシピの材料はすべて揃いました。あとは書き出すだけです。文字列を受け取り、それを単語のリストに分割し、それらの単語をソートし、それからグルーピングします。最後に、"boom" の 3 回の出現を ("boom", 3) に置き換える、魔法のようなマッピングを使います。

```
import Data.List

wordNums :: String -> [(String,Int)]
wordNums = map (\ws -> (head ws, length ws)) . group . sort . words
```

最終的な関数を作るのに関数合成を利用しました。"wa wa wee wa" のような文字列を受け取ると、words が適用され、結果として ["wa","wa","wee","wa"] が得られます。次に、sort が適用され、["wa","wa","wa","wee"] が得られます。これに group が適用されて、同じ隣接要素がグループ化されると、[["wa","wa","wa"],["wee"]] が得られます。それからこのリストに対し、リストを受け取ってタプル（1 つ目の要素はリストの head で、2 つ目の要素はその長さ）を返す関数をマップします。最終的な結果は [("wa",3),("wee",1)] になります。

これを関数合成なしで書くとこうなります。

```
wordNums xs = map (\ws -> (head ws,length ws)) (group (sort (words xs)))
```

うわあ括弧だらけ！ この関数が関数合成で読みやすくなってることがよく分かります。

干し草の山から針を探す

次のミッションは、2 つのリストを受け取り、1 つ目のリストが 2 つ目のリストのどこかに含まれているかを調べる関数を作ることです。例えば、リスト

[3,4] は [1,2,3,4,5] に含まれています。一方、 [2,5] は含まれていません。検索対象のリストを **haystack**（干し草の山）、検索したいリストを **needle**（針）と呼ぶことにします。

この難題を解くために、またもや Data.List の住人である tails 関数を使います。 tails はリストを受け取り、そのリストに対して tail 関数を繰り返し適用します。例を示します。

```
ghci> tails "party"
["party","arty","rty","ty","y",""]
ghci> tails [1,2,3]
[[1,2,3],[2,3],[3],[]]
```

これだけでは tails が必要な理由がピンとこないかもしれません。そのうち分かります。

まず、文字列 "party" から文字列 "art" を探してみましょう。最初に、tails 関数を使ってリストの tail をすべて取得します。それから各 tail を調べて、そのうちのどれかが "art" で始まっていれば、干し草の山（haystack）から針（needle）を探し当てたということです！ "party" の中で "boo" を探していても、 "boo" から始まる tail は見つからないでしょう。

ある文字列が別の文字列から始まっているかどうかを調べるために、やはり Data.List に含まれている isPrefixOf 関数を使います。これは 2 つのリストを引数に取り、2 つ目のリストが 1 つ目のリストで始まっているかどうかを教えてくれます。

```
ghci> "hawaii" `isPrefixOf` "hawaii joe"
True
ghci> "haha" `isPrefixOf` "ha"
False
ghci> "ha" `isPrefixOf` "ha"
True
```

あとは haystack の tail に needle から始まるものがあるか調べるだけです。これには Data.List にある any 関数が使えます。述語とリストを受け取り、要素のどれかが述語を満たすかどうかを返す関数です。ご覧のとおり！

```
ghci> any (> 4) [1,2,3]
False
ghci> any (=='F') "Frank Sobotka"
True
ghci> any (\x -> x > 5 && x < 10) [1,4,11]
False
```

これらの関数を組み合わせましょう。

```
import Data.List

isIn :: (Eq a) => [a] -> [a] -> Bool
needle `isIn` haystack = any (needle `isPrefixOf`) (tails haystack)
```

これでおしまい！ tails を使って haystack の tail のリストを生成し、その中に needle から始まるものがあるかを調べればよいのです。この関数をテストしてみましょう。

```
ghci> "art" `isIn` "party"
True
ghci> [1,2] `isIn` [1,3,5]
False
```

あ、ちょっと待った！ 今作った関数はすでに Data.List にあるんだった！ ちくしょう！ それは isInfixOf という名前で、我々の isIn 関数と同じ動作をします。

シーザー暗号サラダ

ガイウス・ジュリアス・シーザー君が我々に重要な仕事を依頼してきました。ガリアにいるマルクス・アントニウス君に最高機密のメッセージを届けよとのこと。仕事を確実に完遂するために、 Data.Char の関数をいくつか使って、**シーザー暗号**でメッセージを暗号化し、他人には読めないようにしましょう。

シーザー暗号は文字列を暗号化する原始的な方法で、それは各文字をアルファベット上で一定の数だけシフトするというものです。シーザー暗号みたいなものは簡単に作れます。それもアルファベットに限らず Unicode 文字全体に対するものが作れます。

文字を前方向および後ろ方向にシフトするために、 Data.Char モジュールにある文字を対応する数に変換する ord 関数と、その逆を行う chr 関数を使います。

```
ghci> ord 'a'
97
ghci> chr 97
'a'
ghci> map ord "abcdefgh"
[97,98,99,100,101,102,103,104]
```

'a' は Unicode テーブル上の 97 番目に位置するので、 ord 'a' は 97 を返します。

2 つの文字の ord の値の差は、Unicode テーブル上で文字が何文字離れているかと同じです。

文字をシフトする数と文字列を受け取り、文字列中の各文字をアルファベット上で指定された数だけ前方向にシフトする関数を書いてみましょう。

```
import Data.Char

encode :: Int -> String -> String
encode offset msg = map (\c -> chr $ ord c + offset) msg
```

文字列を暗号化するのは、「文字を受け取って対応する数に変換してからオフセットを足して文字に戻す関数」を文字列にマップするという簡単なお仕事です。関数合成野郎ならこの関数を (chr . (+ offset) . ord) のように書くでしょう。

```
ghci> encode 3 "hey mark"
"kh|#pdun"
ghci> encode 5 "please instruct your men"
"uqjfxj%nsxywzhy%~tzw%rjs"
ghci> encode 1 "to party hard"
"up!qbsuz!ibse"
```

間違いなく暗号化されました！

メッセージの復号は、基本的には元の文字をシフトした数だけ単純に逆にシフトすればいいでしょう。

```
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg
```

シーザー君のメッセージを復号してみましょう。

```
ghci> decode 3 "kh|#pdun"
"hey mark"
ghci> decode 5 "uqjfxj%nsxywzhy%~tzw%rjs"
"please instruct your men"
ghci> decode 1 "up!qbsuz!ibse"
"to party hard"
```

正格な左畳み込みにて

前の章で foldl がどのように動作するのか、それを使ってどうやってイケてる関数の数々を実装するのかを見てきました。しかし、 foldl にはまだよく調べていない問題があります。 foldl はコンピュータのメモリの特定の領域を使いすぎたときに起こる、スタックオーバーフローエラーを引き起こすことがあるのです。これを示すために、 foldl と + 関数を使って 100 個の 1 からなるリストを合計してみましょう。

```
ghci> foldl (+) 0 (replicate 100 1)
100
```

大丈夫そうです。では、ドクター・イーブルが**100万**個の1を突っ込んだリストの総和をfoldlで求めようとしたらどうなるでしょう？

```
ghci> foldl (+) 0 (replicate 1000000 1)
*** Exception: stack overflow
```

ぐぬぬ。まさに外道！ どうしてこうなったのでしょう？ Haskellは遅延評価で、だから実際の値の計算は可能な限り後まで引き伸ばされます。foldlを使うとき、Haskellは各ステップにおけるアキュムレータの計算（すなわち評価）を実際には行いません。その代わり、評価を先延ばしにします。その次のステップでもアキュムレータを評価することはなく、評価を先延ばしにします。このとき、新しい計算で前の計算結果を参照するかもしれないので、以前に先延ばしにしていた計算もメモリ上に保持し続けます。こうして畳み込みでは、それぞれバカにならない量のメモリを消費する先延ばしにした計算が積み上がっていきます。そしてついにはスタックオーバーフローエラーを引き起こしてしまうのです[†4]。

Haskellがどのように式`foldl (+) 0 [1,2,3]`を評価するのかを見てみます。

```
foldl (+) 0 [1,2,3] =
foldl (+) (0 + 1) [2,3] =
foldl (+) ((0 + 1) + 2) [3] =
foldl (+) (((0 + 1) + 2) + 3) [] =
((0 + 1) + 2) + 3 =
(1 + 2) + 3 =
3 + 3 =
6
```

見てのとおり、先延ばしにした計算による大きなスタックがまず構築され、空リストに到達したところで、先延ばしにしていたそれらの計算が実際に開始されます。これは小さなリストでは問題になりませんが、100万の要素を含むような大きなリストでは、先延ばしにしていた計算がすべて再帰的に行われるので、スタックオーバーフローという結果になってしまいます。計算を先延ばしにしない、こんな関数`foldl'`があればいいのですが。

---

†4 ［訳注］エラーメッセージはOSやGHCのバージョンによって異なります。また、致命的な結果を引き起こすために必要なリストのサイズも100万よりもずっと大きい場合があります。ドクター・イーブルの挑戦を受けてたつ勇気のある方は、各自のマシンで試してみてください。

```
foldl' (+) 0 [1,2,3] =
foldl' (+) 1 [2,3] =
foldl' (+) 3 [3] =
foldl' (+) 6 [] =
6
```

左折り畳みの各ステップ間で計算が遅延されず、すぐに評価される foldl というわけです。ここでうれしいお知らせがあります。このような foldl の正格バージョンの関数、まさしく foldl' という名前の関数が Data.List にあります。さっそくその foldl' を使って 100 万個の 1 の総和を計算してみましょう。

```
ghci> foldl' (+) 0 (replicate 1000000 1)
1000000
```

大成功です！ もし foldl を使ってスタックオーバーフローエラーになった場合には、 foldl' に切り替えてみてください。 foldl1 に対しても、やはり正格なバージョンの foldl1' という関数があります。

かっこいい数を見つけよう

あなたが通りを歩いていると、老婦人が近づいてきて言いました。「すみません。各桁の数の合計が 40 になる最初の自然数は何でしょうか？」

さて、あなたならどうする？　その数を見つけるのに Haskell の魔法を使いましょう。例えば、123 の各桁を合計すると $1+2+3=6$ から 6 が得られます。じゃあ、各桁の数を足し合わせて 40 になる数は何でしょうか？

はじめに、数を引数として受け取り、各桁の合計を求める関数を作りましょう。ここでちょっとかっこいい技を使います。まず、 show を使って数を文字列に変換します。文字列が得られたなら、それぞれの文字を数に変換して、その数のリストを合計します。文字を数に変換するには Data.Char モジュールの digitToInt というお手軽な関数を使います。これは、 Char を受け取り Int を返します。

```
ghci> digitToInt '2'
2
ghci> digitToInt 'F'
15
ghci> digitToInt 'z'
*** Exception: Char.digitToInt: not a digit 'z'
```

digitToInt は、 '0' から '9' および 'A' から 'F' （小文字でもよい）の文字に対して動作します。

数を引数に取って、その各桁の数の合計を返す関数は、次のように書けます。

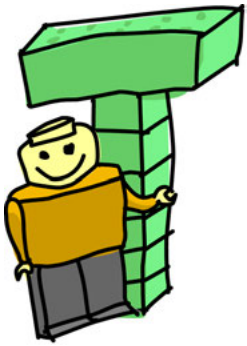
```
import Data.Char
import Data.List

digitSum :: Int -> Int
digitSum = sum . map digitToInt . show
```

文字列に変換し、その文字列に対して digitToInt をマップして、その結果の数のリストを合計します。

さて、 digitSum を適用した結果が 40 になる最初の数を探さないといけません。それには Data.List にある find 関数を使います。これは述語関数とリストを引数に取り、リストの中で条件に合致する最初の要素を返します。この関数は型がちょっと変わっています。

```
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```



最初の引数は述語で、2 つ目の引数はリストですが、これらはどうってことないですね。でも返り値は何でしょうか？ Maybe a と言っています。見たことのない型です。 Maybe a 型の値は、リスト型 [a] に似ています。リストが 0 個、1 個、あるいはもっとたくさんの要素を持てるのに対し、Maybe a 型の値は、0 個かちょうど 1 個の要素だけを持てます。この型は、失敗する可能性があることを表現するのに使います。何も持っていないという値を作るには Nothing を使います。これは空リストに似ています。何かを、例えば "hey" を保持している値を作るときは、 Just "hey" と書きます。ちょっと実演してみましょう。

```
ghci> Nothing
Nothing
ghci> Just "hey"
Just "hey"
ghci> Just 3
Just 3
ghci> :t Just "hey"
Just "hey" :: Maybe [Char]
ghci> :t Just True
Just True :: Maybe Bool
```

ご覧のとおり、 Just True の型は Maybe Bool です。これは、真理値を保持するリストの型が [Bool] であるようなものです。

もし find で述語を満たす要素が見つかったら、その要素を Just でラップしたものが返されます。見つからなければ Nothing が返されます。

```
ghci> find (> 4) [3,4,5,6,7]
Just 5
ghci> find odd [2,4,6,8,9]
Just 9
ghci> find (=='z') "mjolnir"
Nothing
```

では、目的の関数の実装に戻りましょう。すでに digitSum 関数を実装しているし、find がどのように動作するか知っているので、あとは両者を組み合わせるだけです。見つけたいものは、各桁の合計が 40 になる最初の数だということを思い出してください。

```
firstTo40 :: Maybe Int
firstTo40 = find (\x -> digitSum x == 40) [1..]
```

無限リスト [1..] を受け取り、digitSum が 40 になる最初の数を探します。

```
ghci> firstTo40
Just 49999
```

答が見つかりました！ 合計値の希望が 40 に固定されていない、引数として与えられるもっと一般的な関数が欲しければ、次のように変更できます。

```
firstTo :: Int -> Maybe Int
firstTo n = find (\x -> digitSum x == n) [1..]
```

さっと試してみましょう。

```
ghci> firstTo 27
Just 999
ghci> firstTo 1
Just 1
ghci> firstTo 13
Just 49
```

6.3　キーから値へのマッピング

集合のようなデータを扱うときは、その順序を気にしたくない場合もあります。単にキーでアクセスしたい場合などです。例えば、ある住所に誰が住んでいるか知りたいときは、住所で名前を検索したいものです。この節では、望みの値（誰かの名前）を何らかのキー（その人の住所）で検索する話をします。

だいたい大丈夫（連想リスト）

キー／値を対応付ける方法はいくつかあります。そのうちの 1 つは**連想リスト**です。連想リスト（またの名を**辞書**）は、キーと値のペアを順序を気にせずリストにしたものです。例えば電話番号を格納する連想リストなら、電話番号が値

で、人の名前がキーになるでしょう。格納されている順番は気にしません。正しい人名に対して正しい電話番号が得られればそれでいいのです。

Haskell で連想リストを表現する一番あからさまな方法は、ペアのリストでしょう。ペアの 1 つ目の要素をキーに、2 つ目の要素を値にします。電話番号の例ならこんな連想リストです。

```
phoneBook =
    [("betty", "555-2938")
    ,("bonnie", "452-2928")
    ,("patsy", "493-2928")
    ,("lucille", "205-2928")
    ,("wendy", "939-8282")
    ,("penny", "853-2492")
    ]
```

インデントが変に見えるかもしれませんが、単なる文字列のペアのリストです。

連想リストに対する一番よくある操作は、キーによる値の検索です。与えられたキーに対して値を検索する関数を作ってみましょう。

```
findKey :: (Eq k) => k -> [(k, v)] -> v
findKey key xs = snd . head . filter (\(k, v) -> key == k) $ xs
```

実にシンプルです。この関数はキーとリストを受け取り、キーに合致するもののみをフィルタで残して、最初のキー／値のペアを取り出し、値を返します。

でも、連想リストに探しているキーがなかったら何が起こるんでしょうか？うむむ。この場合、キーが連想リストになければ、空リストの head を取ろうとしてランタイムエラーが投げられるでしょう。そんなに簡単にクラッシュしてしまうプログラムを書くべきではありません。そこで Maybe 型を使うことにしましょう。キーが見つからなかったときは Nothing を返します。見つかったときは「Just　何か」（「何か」はキーに対応する値）を返します。

```
findKey :: (Eq k) => k -> [(k, v)] -> Maybe v
findKey key [] = Nothing
findKey key ((k,v):xs)
    | key == k  = Just v
    | otherwise = findKey key xs
```

型宣言を見てください。等値比較のできるキーと連想リストを引数に取り、うまくいけば値を返します。だいたいそんな感じです。

これはリストに対する再帰関数の教科書的な例です。基底部と、リストの head と tail への分割と、再帰呼び出し。全部揃っています。畳み込みの伝統的なパターンなので、畳み込みとして実装する方法を見てみましょう。

```
findKey :: (Eq k) => k -> [(k, v)] -> Maybe v
findKey key xs = foldr
                   (\(k, v) acc -> if key == k then Just v else acc)
                   Nothing xs
```

NOTE　このようなリストに対する標準的な再帰パターンでは、再帰を明示的に書くよりも、畳み込みを使うほうがよいでしょう。そのほうが読みやすく理解しやすいからです。foldr の呼び出しは誰が見ても畳み込みですが、明示的な再帰を読むには考える時間が必要です。

```
ghci> findKey "penny" phoneBook
Just "853-2492"
ghci> findKey "betty" phoneBook
Just "555-2938"
ghci> findKey "wilma" phoneBook
Nothing
```

まるで魔法みたい！ 好きな子の電話番号を持っていればその子の Just な番号が手に入り、そうでなければ Nothing で何も手に入りません。

Data.Map に潜入せよ

今実装したのは、Data.List にある lookup 関数です。キーに対応した値が欲しければ、見つかるまですべての要素を走査する必要があります。

実は、はるかに高速な連想リストと豊富なユーティリティ関数が Data.Map モジュールにあります。これからは「連想リストを使う」という代わりに「**Map** を使う」ということにします。

Data.Map は Prelude や Data.List と競合する名前をエクスポートしているので、修飾付きインポートします。

```
import qualified Data.Map as Map
```

この **import** 文をスクリプトに書いて、それからそのスクリプトを GHCi でロードします。

Data.Map の fromList 関数を使って、連想リストを Map に変換します。fromList は連想リスト（リストの形をしている）を受け取り、同じ対応関係を持つ Map を返します。まずは fromList で少し遊んでみましょう。

```
ghci> Map.fromList [(3,"shoes"),(4,"trees"),(9,"bees")]
fromList [(3,"shoes"),(4,"trees"),(9,"bees")]
ghci> Map.fromList ⇨
          [("kima","greggs"),("jimmy","mcnulty"),("jay","landsman")]
fromList [("jay","landsman"),("jimmy","mcnulty"),("kima","greggs")]
```

Data.Map の Map は、もうリストでも何でもありませんが、ターミナルに表示されるときは fromList に続けてその Map を表現する連想リストが出力されます。

元の連想リストに重複したキーがあった場合、後のほうの要素が使われます。

```
ghci> Map.fromList [("MS",1),("MS",2),("MS",3)]
fromList [("MS",3)]
```

fromList の型シグネチャは、こうです。

```
Map.fromList :: (Ord k) => [(k, v)] -> Map.Map k v
```

これは、型 k と v のペアのリストを受け取り、型 k をキー、型 v を値とする Map を返すと読めます。普通のリストによる連想リストでは、キーは等値比較だけできればよかったのですが（つまり Eq 型クラスに属する型であればよかった）、Map では順序比較が必要なことに注意してください。これは Data.Map モジュールの本質的な制約です。 Data.Map モジュールは、キーが順序付けされていることを利用して、効率よくキーを配置したりキーにアクセスしたりできるのです。

これで phoneBook を、連想リストから Map による実装へと変更できます。きちんと型宣言も追記しましょう。

```
import qualified Data.Map as Map

phoneBook :: Map.Map String String
phoneBook = Map.fromList $
    [("betty", "555-2938")
    ,("bonnie", "452-2928")
    ,("patsy", "493-2928")
    ,("lucille", "205-2928")
    ,("wendy", "939-8282")
    ,("penny", "853-2492")
    ]
```

いいね！ スクリプトを GHCi にロードして phoneBook で遊んでみましょう。まず lookup を使って電話番号を検索してみます。 lookup はキーと Map を受け取り、対応する値を Map から探します。成功したら値を Just で包んで返し、失敗したら Nothing を返します。

```
ghci> :t Map.lookup
Map.lookup :: (Ord k) => k -> Map.Map k a -> Maybe a
```

```
ghci> Map.lookup "betty" phoneBook
Just "555-2938"
ghci> Map.lookup "wendy" phoneBook
Just "939-8282"
ghci> Map.lookup "grace" phoneBook
Nothing
```

お次は、 phoneBook に電話番号を挿入して新しい Map を作りましょう。 insert はキーと値、それに Map を受け取り、その Map にキーと値を挿入した新しい Map を返します。

```
ghci> :t Map.insert
Map.insert :: (Ord k) => k -> a -> Map.Map k a -> Map.Map k a
ghci> Map.lookup "grace" phoneBook
Nothing
ghci> let newBook = Map.insert "grace" "341-9021" phoneBook
ghci> Map.lookup "grace" newBook
Just "341-9021"
```

電話番号がいくつあるか調べてみましょう。 Data.Map の size 関数は、Map を受け取り、そのサイズを返します。そのままです。

```
ghci> :t Map.size
Map.size :: Map.Map k a -> Int

ghci> Map.size phoneBook
6
ghci> Map.size newBook
7
```

この電話帳では、電話番号を文字列として表現しています。今、文字列ではなく Int のリストで電話番号を表現したいとしましょう。つまり、 "939-8282" ではなく [9,3,9,8,2,8,2] のように電話番号を持つのです。最初に、電話番号の文字列を Int のリストに変換する関数を作ります。 Data.Char の digitToInt を文字列に対してマップすればよさそうです。しかし、この関数は文字列にダッシュ（「 - 」）が出てきたらお手上げです！ そのため、事前に数字以外を取り除いておく必要があります。それには、文字が数字かどうかを教えてくれる Data.Char の isDigit 関数が使えます。文字列をフィルタしてしまえば、あとは digitToInt をマップするだけです。

```
string2digits :: String -> [Int]
string2digits = map digitToInt . filter isDigit
```

あ、もしまだなら、 **import** Data.Char をインポートするのを忘れずに！ 試してみましょう。

```
ghci> string2digits "948-9282"
[9,4,8,9,2,8,2]
```

たいへんよくできました！ では、Data.Map の map 関数を使って、phoneBook を string2digits でマップしましょう。

```
ghci> let intBook = Map.map string2digits phoneBook
ghci> :t intBook
intBook :: Map.Map String [Int]
ghci> Map.lookup "betty" intBook
Just [5,5,5,2,9,3,8]
```

Data.Map の map は、関数と Map を受け取り、その関数を Map の中の各値に適用します。

電話帳を拡張してみましょう。一人が複数の番号を持っていて、連想リストが次のように与えられたとします。

```
phoneBook =
    [("betty", "555-2938")
    ,("betty", "342-2492")
    ,("bonnie", "452-2928")
    ,("patsy", "493-2928")
    ,("patsy", "943-2929")
    ,("patsy", "827-9162")
    ,("lucille", "205-2928")
    ,("wendy", "939-8282")
    ,("penny", "853-2492")
    ,("penny", "555-2111")
    ]
```

これに対して単純に fromList を使ってしまうと、番号がいくつか失われてしまいます！ 代わりに Data.Map にある別の関数、 fromListWith を使います。この関数は fromList と似ていますが、キーの重複を削除しません。その代わり、重複時にどうするかを決める関数を受け取ります。

```
phoneBookToMap :: (Ord k) => [(k, String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith add xs
    where add number1 number2 = number1 ++ ", " ++ number2
```

fromListWith は、すでに存在するキーを見つけると、与えられた結合関数にそれら競合する 2 つの値を渡し、得られた値で古い値を置き換えます。

```
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
"827-9162, 943-2929, 493-2928"
ghci> Map.lookup "wendy" $ phoneBookToMap phoneBook
"939-8282"
```

```
ghci> Map.lookup "betty" $ phoneBookToMap phoneBook
"342-2492, 555-2938"
```

あらかじめ連想リストの値を単一要素のリストにしておけば、電話番号を連結するのに ++ が使えます。

```
phoneBookToMap :: (Ord k) => [(k, a)] -> Map.Map k [a]
phoneBookToMap xs = Map.fromListWith (++) $ map (\(k, v) -> (k, [v])) xs
```

GHCi でテストしてみましょう。

```
ghci> Map.lookup "patsy" $ phoneBookToMap phoneBook
["827-9162","943-2929","493-2928"]
```

とても簡潔ですね！

番号の連想リストから Map を作って、そのキーに対する値の中の最大値を保持したいとしましょう。このように書けます。

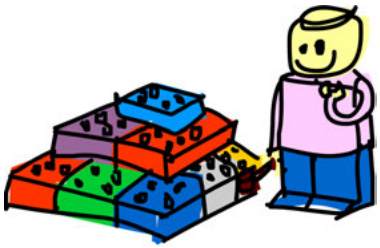
```
ghci> Map.fromListWith max ⇨
        [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,100),(3,29),(4,22)]
```

同じキーの値を足し合わせることもできます。

```
ghci> Map.fromListWith (+) ⇨
        [(2,3),(2,5),(2,100),(3,29),(3,22),(3,11),(4,22),(4,15)]
fromList [(2,108),(3,62),(4,37)]
```

Data.Map などの Haskell が提供するモジュールがとってもクールなことが分かったところで、自分でモジュールを作る方法を見ていきましょう。

6.4　モジュールを作ってみよう



この章の冒頭で言ったように、プログラムを書くときに、似たような目的の関数と型をまとめてモジュールに分けるのは良い習慣です。そうすれば、そのモジュールをインポートするだけで、それらの関数を他のプログラムから簡単に再利用できるようになります。

モジュールからは関数を**エクスポート**します。モジュールをインポートすると、そのモジュールがエクスポートする関数が使えるようになります。モジュールの内部で使う関数も定義できますが、モジュールの外で使えるのはエクスポートした関数だけです。

幾何学モジュール

幾何学オブジェクトの体積と面積を計算する小さいモジュールを例に、モジュールの作り方を見ていきます。まずは Geometry.hs という名前のファイルを作ることから始めます。

モジュールの先頭でモジュールの名前を指定します。Geometry.hs というファイル名なら、Geometry という名前にしなければなりません。そのモジュールがエクスポートする関数を指定して、それから関数を追加します。なので、モジュールはこのようなコードから始まります。

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where
```

見てのとおり、球（sphere）、立方体（cube）、直方体（cuboid）の体積、表面積を求める予定です。球はグレープフルーツのような丸い形で、立方体はサイコロのような形で、直方体はタバコの箱のような形です（子供はタバコ吸っちゃダメだよ！）。

では関数を定義しましょう。

```
module Geometry
( sphereVolume
, sphereArea
, cubeVolume
, cubeArea
, cuboidArea
, cuboidVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

sphereArea :: Float -> Float
sphereArea radius = 4 * pi * (radius ^ 2)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cubeArea :: Float -> Float
cubeArea side = cuboidArea side side side
```

```
cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectArea a b * c

cuboidArea :: Float -> Float -> Float -> Float
cuboidArea a b c = rectArea a b * 2 + rectArea a c * 2 +
rectArea c b * 2

rectArea :: Float -> Float -> Float
rectArea a b = a * b
```

とても基本的な幾何学ですが、注意点がいくつかあります。立方体は直方体の特別なケースなので、その表面積と体積は、すべての辺が同じ長さの直方体として扱うことで定義しています。また、辺の長さから長方形の面積を求めるのに、rectArea という補助関数を定義しています。単なる掛け算なので取るに足らない関数です。この関数をモジュール内（の cuboidArea と cuboidVolume ）で使っているけど、エクスポートはしていないことに気をつけてください！ これは、このモジュールは三次元のオブジェクトを扱う関数だけをエクスポートするようにしたいからです。

モジュールを作成したら、インターフェイスとしての役割をする関数のみをエクスポートするようにします。そうすれば実装は隠蔽されます。 Geometry モジュールの利用者は、エクスポートされていない関数のことを気にする必要はありません。新しいバージョンで関数を完全に書き換えたり削除したりしても（例えば rectArea を削除して代わりに * を使うとか)、エクスポートしていない関数のことなので誰も気づきません。

このモジュールを使うには次のようにするだけです。

```
import Geometry
```

ただし、Geometry.hs がインポートするモジュールと同じフォルダになければいけません。

階層的モジュール

モジュールには階層構造を与えることもできます。各モジュールは複数のサブモジュールを持つことができ、そのサブモジュールはまたサブモジュールを持つことができます。幾何学モジュール Geometry を分割して、立体の種類ごとの3つのサブモジュールを持つモジュールにしてみましょう。

最初に、Geometry という名前のフォルダを作ります。その中に3つのファイル、Sphere.hs、Cuboid.hs、Cube.hs を作ります。3つのファイルの中身をそれぞれ見てみましょう。

Sphere.hs の内容は次のとおりです。

```
module Geometry.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

Cuboid.hs は次のようになっています。

```
module Geometry.Cuboid
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectArea a b * c

area :: Float -> Float -> Float -> Float
area a b c = rectArea a b * 2 + rectArea a c * 2 + rectArea c b * 2

rectArea :: Float -> Float -> Float
rectArea a b = a * b
```

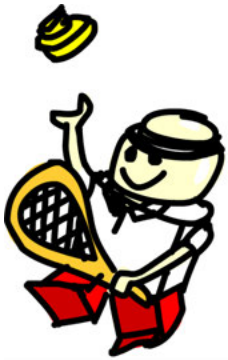
最後は次のような Cube.hs です。

```
module Geometry.Cube
( volume
, area
) where

import qualified Geometry.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```



Sphere.hs を Geometry フォルダの中に配置して、Geometry.Sphere という名前のモジュールを定義していることに注目してください。立方体と直方体でも同じようにしました。また、3 つのモジュールすべてで同じ名前の関数を定義していることにも注目してください。これが可能なのは別々のモジュールにあるからです。

これで、こんなふうにインポートできるようになりました。

```
import Geometry.Sphere
```

インポートしたら、球の表面積と体積を求める area と volume 関数を呼び出すことができます。

2つ、あるいはそれ以上のモジュールを操作したいなら、修飾付きインポートを使う必要があります。同じ名前の関数をエクスポートしているからです。次はその例です。

```
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
```

Sphere.area 、 Sphere.volume 、 Cuboid.area のようにすれば、対応する立体の表面積あるいは体積を求めることができます。

とても大きくて、とてつもない数の関数を含むファイルを書いていることに気づいたときは、共通して使える関数を見つけ出し、それをモジュールに切り出すことを検討しましょう。そうすれば、次に同じ機能が必要なプログラムを書くときは、モジュールをインポートするだけで済むようになります。

第7章
型や型クラスを自分で作ろう

これまで、 Bool 、 Int 、 Char 、 Maybe などなど、いろんなデータ型が出てきました。じゃあ、自分でデータ型を作るにはどうすればよいのでしょうか？ この章では、独自の型を作り、それを動かす方法を見ていきます！

7.1 新しいデータ型を定義する

自作のデータ型を作る方法の 1 つは **data** キーワードを使うことです。まずは、標準ライブラリで Bool 型の定義がどうなっているか見てみましょう。

```
data Bool = False | True
```

こういうふうに **data** キーワードを使うと、新しいデータ型を定義する構文になります。等号の前の部分は型の名前、ここでは Bool を表します。等号の後の部分は値コンストラクタです。値コンストラクタは、この型が取り得る値の種類を指定しています。記号 | は**または**の意味です。というわけで、この型宣言は、「 Bool 型は True または False の値を取り得る」と読めます。型名と値コンストラクタはどちらも大文字で始まる必要があります。

同様に、 Int 型はこのように定義されているとみなせます。

```
data Int = -2147483648 | -2147483647 | ... | -1 |
           0 | 1 | 2 | ... | 2147483647
```

最初と最後の値は Int が取り得る最小値と最大値です。もちろん実際の Int はこのように定義されているわけではありません（途中の数が省略されてますし

ね)。でも、概念を説明するには便利でしょう？

さて、Haskell で図形を表すにはどうしますか？ タプル（この場合はトリプル）を使うという手があります。例えば、円は `(43.1, 55.0, 10.4)` と表せます。1 つ目と 2 つ目の数字が円の中心の座標で、3 つ目が円の半径です。この表記の問題点は、三次元ベクトルみたいな、実数の 3 つ組で指定できる任意のものを表せてしまうことです。より良い解決策は、図形を表す型を自作することでしょう。

7.2　形づくる

ここでは、長方形と円という 2 種類の図形を扱うことにしましょう。こんなやり方が考えられます。

```
data Shape = Circle Float Float Float |
             Rectangle Float Float Float Float
```

これはどういう意味でしょう？ 次のように解釈してください。 `Circle` 値コンストラクタには、浮動小数を受け取るフィールドが 3 つあります。このように、値コンストラクタを書くときは後ろに型を付け足すことができ、それらは値コンストラクタに与える引数の型になります。ここでは、最初の 2 つのフィールドは円の中心の座標で、3 つ目のフィールドは円の半径です。一方、 `Rectangle` 値コンストラクタには浮動小数を受けるフィールドが 4 つあります。最初の 2 つは左下の角、後の 2 つは右上の角の座標を表します。

実のところ値コンストラクタは、最終的にそのデータ型の値を返す関数なのです。これら 2 つの値コンストラクタの型シグネチャを見てみましょう。

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

というわけで、値コンストラクタは何の変哲もない関数です。意外でしょう？ データ型にあったフィールドは、値コンストラクタ関数にとっての引数に対応します。

では、 `Shape` を引数に取って、その面積を返す関数を作ってみましょう。

```
area :: Shape -> Float
area (Circle _ _ r) = pi * r ^ 2
area (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

まず型宣言に注目です。 `area` は、 `Shape` を取って `Float` を返す関数だよ、と言っています。ここに例えば `Circle -> Float` という型宣言を書くことはできません。なぜなら `Shape` は型だけど `Circle` は型じゃなくて値コンストラクタ

だからです（ちょうど、 `True -> Int` のような型宣言を持つ関数が書けないのと同じことです）。

次に、コンストラクタはパターンマッチに使えることに注目です。これまでにも、 `[]` 、 `False` 、 `5` といった値に対するパターンマッチは行ってきました。しかし、今までの値にはフィールドはありませんでした。フィールドがある値コンストラクタでパターンマッチを行いたい場合、まず値コンストラクタ名を書き、それからフィールドを名前に束縛します。円の場合、必要なのは半径の情報だけで、円の位置を表す最初の 2 つのフィールドはいらないので、 `_` で捨てています。

```
ghci> area $ Circle 10 20 10
314.15927
ghci> area $ Rectangle 0 0 100 100
10000.0
```

お、動いた！ でも `Circle 10 20 5` をプロンプトから表示しようとするとエラーになります。これは、Haskell が今作ったデータ型を文字列にして表示する方法を（まだ）知らないからです。プロンプトから何らかの値を表示しようとした場合、Haskell はまずその値に `show` 関数を適用して文字列表記を得て、それを端末に表示します。

この `Shape` 型を `Show` 型クラスの一員にするには、型宣言をこう修正します。

```
data Shape = Circle Float Float Float |
             Rectangle Float Float Float Float
    deriving (Show)
```

今は、 **`deriving`** のことはあまり気にしないことにします。データ宣言の最後に **`deriving`** （`Show`）と書けば（同じ行でも次の行でもかまいません）、Haskell が自動的にこの型を `Show` 型クラスのインスタンスにしてくれる、とだけ覚えておきましょう。 **`deriving`** については 126 ページからの「インスタンスの自動導出」の節で詳細に見ていくことにします。

これで `Shape` 型の値を表示できます。

```
ghci> Circle 10 20 5
Circle 10.0 20.0 5.0
ghci> Rectangle 50 230 60 90
Rectangle 50.0 230.0 60.0 90.0
```

値コンストラクタは関数ですから、普通に `map` したり、部分適用したりできます。例えば、半径だけ違う同心円のリストを作りたかったら、こう書けます。

```
ghci> map (Circle 10 20) [4,5,6,6]
[Circle 10.0 20.0 4.0,Circle 10.0 20.0 5.0,Circle 10.0 20.0 6.0,
 Circle 10.0 20.0 6.0]
```

Point データ型で形を整える

いい感じにデータ型を自作できましたが、もっと良くできるんですよ。二次元空間の点を表す中間データ構造を作りましょう。そうすれば図形をもっと分かりやすくできます。

```
data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float | Rectangle Point Point deriving (Show)
```

点を定義するとき、データ型と値コンストラクタに同じ名前を使いました。ここがポイントです。同じ名前にしておくことに特別な意味はないのですが、値コンストラクタが 1 つしかないデータ型はそうするのが慣例です。というわけで、新しい Circle には 2 つのフィールドがあります。1 つは Point 型で、もう 1 つは Float 型。これで、どのフィールドが何なのか分かりやすくなりました。 Rectangle も同じことです。では、 area 関数にもこの変化を反映させましょう。

```
area :: Shape -> Float
area (Circle _ r) = pi * r ^ 2
area (Rectangle (Point x1 y1) (Point x2 y2))
                  = (abs $ x2 - x1) * (abs $ y2 - y1)
```

変更する必要があったのはパターンだけでした。 Circle パターンでは、点全体を無視しています。 Rectangle パターンでは、パターンマッチをネストすることで、一気に点が属するフィールドへアクセスしています。(もし何らかの理由で点全体も参照したくなったら、as パターンの使いどころですね。)

改良したバージョンを試してみましょう。

```
ghci> area (Rectangle (Point 0 0) (Point 100 100))
10000.0
ghci> area (Circle (Point 0 0) 24)
1809.5574
```

図形を動かす関数とかも欲しいですね。図形と、x 軸方向への移動量、y 軸方向への移動量を取って、元の図形と同じ寸法だけど違う場所にある図形を返します。

```
nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b
    = Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+b))
```

ずいぶん直感的に書けましたね。図形の位置を表す点に移動量を足し算すればいいわけです。試してみましょう。

```
ghci> nudge (Circle (Point 34 34) 10) 5 10
Circle (Point 39.0 44.0) 10.0
```

もし点を直接触りたくないというのなら、指定したサイズの図形を原点に作る補助関数を作って、それから移動させることもできます。

まず、半径を取って、座標系の原点を中心とし、与えられた半径の円を作る関数を作りましょう。

```
baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r
```

次に、幅と高さを取って、左下の頂点が原点にある長方形を作る関数を作りましょう。

```
baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
```

これらの関数を使えば、まず座標系の原点に図形を作って、それから望みの場所まで移動させることができます。これで図形を作るのが楽になりますね。

```
ghci> nudge (baseRect 40 100) 60 23
Rectangle (Point 60.0 23.0) (Point 100.0 123.0)
```

Shape をモジュールとしてエクスポートする

自作のデータ型も自作のモジュールからエクスポートできます。型をエクスポートするには、関数のエクスポートと同じところに型名を書くだけです。値コンストラクタをエクスポートしたい場合は、型名の後に括弧を追加し、その中にカンマ区切りで値コンストラクタを書きます。ある型の値コンストラクタをすべてエクスポートしたい場合は、ピリオド 2 つ (..) を書いてください。

図形を扱う関数と型をモジュールからエクスポートしたいとしたら、こう書き始めます。

```
module Shapes
( Point(..)
, Shape(..)
, area
, nudge
, baseCircle
, baseRect
) where
```

Shape(..) と書くことで、 Shape のすべての値コンストラクタがエクスポートされます。つまり、このモジュールをインポートした人は、 Rectangle と Circle の値コンストラクタを使って図形を作れるようになります。この行は Shape (Rectangle, Circle) と書いても同じことですが、 (..) を使うほうが簡単ですね。

それに、エクスポートした型に後から値コンストラクタを追加することにした場合でも、エクスポート文を変える必要がなくなります。 .. を使えば、その型のすべての値コンストラクタが自動的にエクスポートされるからです。

一方、 Shape のエクスポート文の後に括弧を付けないことで、 Shape 型をエクスポートするけれどもその値コンストラクタは一切エクスポートしない、という選択もできます。すると、このモジュールのユーザは補助関数 baseCircle と baseRect 経由でしか図形を作れなくなります。 以下のモジュールを Shapes.hs というファイル名で保存した上で、

```
module Shapes
( Point, Shape, area, nudge, baseCircle, baseRect) where

data Point = Point Float Float deriving (Show)
data Shape = Circle Point Float |
             Rectangle Point Point deriving (Show)

area :: Shape -> Float
area (Circle _ r) = pi * r ^ 2
area (Rectangle (Point x1 y1) (Point x2 y2))
  = (abs $ x2 - x1) * (abs $ y2 - y1)

nudge :: Shape -> Float -> Float -> Shape
nudge (Circle (Point x y) r) a b = Circle (Point (x+a) (y+b)) r
nudge (Rectangle (Point x1 y1) (Point x2 y2)) a b
  = Rectangle (Point (x1+a) (y1+b)) (Point (x2+a) (y2+b))

baseCircle :: Float -> Shape
baseCircle r = Circle (Point 0 0) r

baseRect :: Float -> Float -> Shape
baseRect width height = Rectangle (Point 0 0) (Point width height)
```

同じフォルダに以下のような Main.hs を作って動かしてみてください。

```
import Shapes
main = do
  print $ Circle (Point 10 20) 30
```

Haskell に「Circle に Point ？ そのようなものは存じておりませんが。」と冷たくあしらわれましたね？ このようにして Haskell は Shapes モジュールのプライバシーを完璧に守るのです。一方、Main.hs を次のように書き直せば、Shapes モジュールが許可しているものしか使っていないので、ちゃんと動きます。

```
import Shapes
main = do
  print $ nudge (baseCircle 30) 10 20
```

値コンストラクタは、フィールドを引数に取ってデータ型（例えば Shape 型）の値を返す関数にすぎないのでした。値コンストラクタをエクスポートしない、という選択をすれば、このモジュールをインポートする人が値コンストラクタを関数として直接使うことを防ぐことになります。データ型の値コンストラクタをエクスポートしないことで、データ型の実装を隠し、データ型の抽象度を上げられます。また、値コンストラクタを使ったパターンマッチもできなくなります。ユーザには必ずモジュールの提供する補助関数を使ってデータ型にアクセスしてほしい、という場合には、値コンストラクタをエクスポートしないことが望ましい選択になります。そうすれば、モジュールのユーザはモジュール内部の詳細を知る必要がなくなりますし、モジュールの作者にとってもエクスポートした関数の振る舞いが同じである限り内部の実装を自由に変えられるという利点があります[†1]。

`Data.Map` はこのアプローチを取っています。どう頑張っても、値コンストラクタから直接 Map を作ることはできません。エクスポートされていないからです。ですが、`Map.fromList` のような補助関数を使って Map を作ることは普通にできます。おかげで `Data.Map` の作者たちは、既存のプログラムを壊す心配なく、自由に Map の内部表現を変更できるのです。

でも単純なデータ型に対しては、値コンストラクタをエクスポートするのも何ら問題のない方針ですよ。

7.3 レコード構文

今度はデータ型を作る別の方法を見ていきましょう。人物を記述するデータ型を作る仕事を任されたとします。人物に関しては、その名前、苗字、年齢、身長、電話番号、好きなアイスクリームの味を記録することにしましょう（これだけ分かれば人物の記録に十分です。少なくとも僕は）。さっそくやってみましょう！

```
data Person = Person String String Int Float String String
    deriving (Show)
```

最初のフィールドは名前、次が苗字、次が年齢、という具合です。では人物を作ってみましょう。

[†1] ［訳注］このように、ユーザに必要なだけの操作法を提供するが、実装の詳細は隠されているようなデータ型のことを抽象データ型と呼びます。

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" ⇨
                       "Chocolate"
ghci> guy
Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

ちょっと読みにくいですが、それっぽいものができました。

では、特定の情報を取り出すにはどうすればいいでしょう？ 人物の名前を取り出す関数とか、苗字を取り出す関数とかが必要ですよね。こういう定義が必要そうです。

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

やれやれ。こんなの書いてても全然楽しくないっすね！ まあ、穴あきだらけで飽き飽きするようなコードですが、この方法で動くことは動きます。

```
ghci> let guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" ⇨
                       "Chocolate"
ghci> firstName guy
"Buddy"
ghci> height guy
184.2
ghci> flavor guy
"Chocolate"
```

「でも、もっとマシな方法があるんでしょ！」と思ったかもしれませんね。おあいにくさまですが、ありません。

うそだよー、ありますよ！ ハッハッハッ！

Haskell にはデータ型を書くための構文がもう１つ備わっています。それは**レコード構文**です。次のようにして同じ機能を実現できます。

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     , flavor :: String } deriving (Show)
```

このようにレコード構文では、フィールドの型名だけをスペース区切りで列挙する代わりに、中括弧を使います。まずフィールドの名前（例えば firstName ）を書き、次に :: を書き、それから型を書きます。出来上がったデータ型は以前のものとまったく同じです。この構文の利点は、データ型と同時にフィールドを取得する関数たちが作られることです。この型をレコード構文を使って定義したことで、Haskell は firstName 、 lastName 、 age 、 height 、phoneNumber、flavor という 6 つの関数を自動的に作ってくれるのです。見てください。

```
ghci> :t flavor
flavor :: Person -> String
ghci> :t firstName
firstName :: Person -> String
```

レコード構文にはもう 1 つ利点があります。 Show のインスタンスを自動導出（derive）するとき、レコード構文を使って定義しインスタンス化した型は、そうでない型とは違う表示の仕方になります。

例えば自動車を表す型があるとしましょう。メーカー、モデル名、生産された年を記録したいとします。この型は、レコード構文を使わない場合はこのように定義できます。

```
data Car = Car String String Int deriving (Show)
```

自動車は、例えばこのように表示されます。

```
ghci> Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967
```

では、レコード構文を使って定義した場合は何が起こるでしょうか。

```
data Car = Car { company :: String
               , model :: String
               , year :: Int
               } deriving (Show)
```

今度はこのようにして自動車が作れます。

```
ghci> Car {company="Ford", model="Mustang", year=1967}
Car {company = "Ford", model = "Mustang", year = 1967}
```

このときフィールドを元どおりの順番で指定する必要はありません。ただし、すべてのフィールドを埋める必要はあります。一方、レコード構文を使わなかっ

た場合は、フィールドを順番どおりに指定する必要があります。

値コンストラクタに複数のフィールドがあり、どのフィールドが何番目なのか紛らわしい場合は、レコード構文を使ってください。三次元ベクトルのデータ型を `data Vector = Vector Int Int Int` として作った場合は、フィールドがベクトルの成分であることは明らかです。一方、`Person` や `Car` の例ではフィールドがそこまで自明ではないので、レコード構文を使うほうが圧倒的に便利です。

7.4　型引数

値コンストラクタは引数を取って新しい値を生み出すのでした。例えば、`Car` という値コンストラクタは 3 つの値を取って `stang` のような自動車を表す値を作ります。同様に、型コンストラクタは型を引数に取って新しい型を作るものです。何だかとってもメタな概念のように聞こえるかもしれませんが、そんなに複雑なものじゃないですよ（C++ のテンプレートをご存知なら似てると思うかもしれません）。型引数がどう使われるか明確なイメージをつかむために、すでに知っている型がどんな実装になっているか見てみましょう。

```
data Maybe a = Nothing | Just a
```

この `a` が型引数です。そして型引数を取っているので、`Maybe` は**型コンストラクタ**と呼ばれます。`Nothing` でない場合に何をこのデータ型に保持させたいかに応じて、この型コンストラクタから `Maybe Int`、`Maybe Car`、`Maybe String` などの型を作れます。単なる `Maybe` という型の値は存在できません。なぜなら、`Maybe` は型コンストラクタであって、型ではないからです。型コンストラクタは、すべての引数を埋めて初めて何かしらの値の型になれます。

`Char` を `Maybe` の型引数に渡すと `Maybe Char` という型が得られます。例えば `Just 'a'` という値は `Maybe Char` という型を持つわけです。

たいていは型コンストラクタに型引数を明示的に渡さずに済みます。Haskell には型推論があるからです。例えば `Just 'a'` という値を作ると、Haskell はその型が `Maybe Char` であることを推論してくれます。

明示的に型を型引数として渡したければ、型の世界で行う必要があります。Haskell では、 :: 記号の右側が型の世界です。この手法は、例えば Just 3 に Maybe Int という型を持ってほしい場合に便利です。デフォルトでは、Haskell はこの値の型を (Num a) => Maybe a だと推論するでしょう。明示的な型注釈を使えば、型の制限をちょっとだけキツくできます。

```
ghci> Just 3 :: Maybe Int
Just 3
```

気づかなかったかもしれませんが、実は Maybe を使うより前に型引数を使っていました。それはリスト型です。構文糖衣があるので分かりにくいですが、リスト型は 1 つの型引数を取って具体型を生成するものなのです。[Int] 型の値、[Char] 型の値、[[String]] 型の値、などはありますが、[] 型の値というのは作れません。

NOTE 具体型とは、型引数を 1 つも取らない型か、あるいは、型引数を取るけれどもそれがすべて埋まっている型のことを指します。前者の例は Int や Bool 、後者の例は Maybe Char です。何らかの値があったら、その型は常に具体型です。

Maybe 型で遊んでみましょう。

```
ghci> Just "Haha"
Just "Haha"
ghci> Just 84
Just 84
ghci> :t Just "Haha"
Just "Haha" :: Maybe [Char]
ghci> :t Just 84
Just 84 :: (Num a) => Maybe a
ghci> :t Nothing
Nothing :: Maybe a
ghci> Just 10 :: Maybe Double
Just 10.0
```

型引数が便利なのは、さまざまな型を収納するデータ型を作れるところです。例えば Maybe を定義するのに、中身の型ごとに別々の型にすることもできなくはありません。

```
data IntMaybe = INothing | IJust Int

data StringMaybe = SNothing | SJust String

data ShapeMaybe = ShNothing | ShJust Shape
```

でも、型引数を使ってどんな型の値でも収納できる汎用の Maybe を作ったほうが便利ですよね！

Nothing の型が Maybe a であることに注意してください。この型は型引数がある、つまり Maybe a の a があるので、**多相的**です。もし、 Maybe Int を引数に取る関数があれば、それに Nothing を渡すことができます。 Nothing は値を含んでいないので、どのみち問題ないわけです。このように、 Maybe a 型の値は、 Maybe Int 型が要求される場所でも使えます。ちょうど、 5 は Int としても Double としても使えるのと同じことです。同様に、空リストの型は [a] です。これが、 [1,2,3] ++ [] という使い方も ["ha","ha","ha"] ++ [] という使い方も同時にできる理由です。

自動車は型引数を取るべきか？

型引数を使うのがよいのは、どういうときでしょうか？ 普通、型引数は Maybe a 型のような、どんな型をそこに持ってきても変わらず動作するようなデータ型に使うものです。ある型が何らかの箱のように振る舞うなら、型引数を使うとよいでしょう。

Car データ型がありましたよね。

```
data Car = Car { company :: String
               , model :: String
               , year :: Int
               } deriving (Show)
```

これをこう書き換えてみましょう。

```
data Car a b c = Car { company :: a
                     , model :: b
                     , year :: c
                     } deriving (Show)
```

でも、これって何かメリットあるの？

たぶん皆無です。どうせ Car String String Int を扱う関数しか作らないでしょうから。例えば、最初に定義した Car を使えば、自動車の情報を読みやすい形で表示する関数はこう書けます。

```
tellCar :: Car -> String
tellCar (Car {company = c, model = m, year = y}) =
    "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

試してみましょう。

```
ghci> let stang = Car {company="Ford", model="Mustang", year=1967}
ghci> tellCar stang
"This Ford Mustang was made in 1967"
```

かわいい関数ですね！ 型宣言も小さくまとまってるし、ちゃんと動きます。

では、 Car が Car a b c だったらどうなるでしょう？

```
tellCar :: (Show a) => Car String String a -> String
tellCar (Car {company = c, model = m, year = y}) =
    "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

この関数には、(Show a) => Car String String a という特殊化した Car 型を取らせる必要があります。そのため、ご覧のとおり型シグネチャはだいぶ複雑になっちゃいました。メリットといえば、Show 型クラスのインスタンスなら何でも c の箇所に取れるようになったことくらいです。

```
ghci> tellCar (Car "Ford" "Mustang" 1967)
"This Ford Mustang was made in 1967"
ghci> tellCar (Car "Ford" "Mustang" "nineteen sixty seven")
"This Ford Mustang was made in \"nineteen sixty seven\""
ghci> :t Car "Ford" "Mustang" 1967
Car "Ford" "Mustang" 1967 :: (Num t) => Car [Char] [Char] t
ghci> :t Car "Ford" "Mustang" "nineteen sixty seven"
Car "Ford" "Mustang" "nineteen sixty seven" :: Car [Char] [Char] [Char]
```

でも、実用上は Car String String Int を使うことが圧倒的に多いでしょう。だから、Car を多相型にしても労力に見合わないというものです。

普通、型引数を使うのは、データ型の値コンストラクタに収納された型が、データ型自体の動作にそこまで重要な影響を与えないときです。例えば「物」のリストは、中身の物が何であれ、とりあえずリストはリストです。数のリストの合計を求めたくなったら、特に数のリストを引数に取る総和関数を後から作ればいいわけです。Maybe についても同じことが言えます。Maybe は、ある物を 1 つ持つか持たないかの選択肢を表しているのであって、その物が何型であるかとは関係ありません。

これまでに出てきた中では、Data.Map モジュールの Map k v も多相型です。k は Map のキーの型で、v は Map の値の型です。これは型引数が大活躍できている例ですね。Map が多相化されていることで、どんな型からどんな型への Map でも作れます。もっとも、キーの型のほうは Ord に属しているという条件付きですが。ところで、Data.Map の作者には、データ型宣言にこうやって型クラス制約を加えるという選択もあったでしょう。

```
data (Ord k) => Map k v = ...
```

しかし Haskell には、データ宣言には決して型クラス制約を付けないという、とても強いコーディング規約があります[†2]。なぜかって？ それは、データ宣言に型クラス制約を付けても大した利益がない割に、型クラス制約をそこら中、必要ない箇所にまで書いてまわる羽目になるからです。もし Map k v のデータ宣

†2 ［訳注］データ宣言に型クラス制約を付ける機能は、次の Haskell 2012 で削除する予定で、現在の GHC（7.2.1 以降）ではデフォルトでは無効化され非推奨とされているほどです。

言に Ord k という型クラス制約があっても、Map のキーが順序付け可能であることを仮定する関数には、やっぱり型クラス制約を書く必要があります。これに対し、データ宣言に型クラス制約がなければ、キーの型が順序付け可能であることを仮定しない関数からは (Ord k) => を省くことができます。その一例が、Map を取って連想リストに変換する関数 toList です。こいつの型シグネチャは toList :: Map k a -> [(k, a)] です。もし Map k v のデータ宣言に型クラス制約が付いていたら、toList の型も toList :: (Ord k) => Map k a -> [(k, a)] となる必要があったでしょう。 toList 関数自体はキーの順序比較をまったく使わないというのに。

というわけで、このデータ宣言にはこの型クラス制約を付けるのが当然だろ、と思える場合でも、決して型クラス制約を付けないでください。どのみち関数の型宣言には型クラス制約を付ける必要があるんです。

三次元ベクトル

三次元ベクトルの型と、ベクトルの演算を作ってみましょう。ベクトルは多相型にします。なぜなら、普通は数値型に限るとしても、Int 、Integer 、Double など複数の型をサポートしたいからです。

```
data Vector a = Vector a a a deriving (Show)

vplus :: (Num a) => Vector a -> Vector a -> Vector a
(Vector i j k) `vplus` (Vector l m n) = Vector (i+l) (j+m) (k+n)

dotProd :: (Num a) => Vector a -> Vector a -> a
(Vector i j k) `dotProd` (Vector l m n) = i*l + j*m + k*n

vmult :: (Num a) => Vector a -> a -> Vector a
(Vector i j k) `vmult` m = Vector (i*m) (j*m) (k*m)
```

ベクトルとは空間の中の矢印のようなものだと思ってください。どこかを指している線分です。ベクトル Vector 3 4 5 は三次元空間の座標 (0,0,0) を始点とし、(3,4,5) を終点（そこを指している）とする線分です。

ベクトルを扱う関数はこのような実装になっています。

- vplus 関数は 2 つのベクトルを加算します。これは両ベクトルの対応する成分を加算することで実現できます。2 つのベクトルを加算すると、片方のベクトルを他方のベクトルの終点に継ぎ足したようなベクトルができます。こうして 2 つのベクトルを加算すると第三のベクトルになります。
- dotProd 関数は 2 つのベクトルの内積を取ります。内積の結果はただの数（スカラー）で、これはベクトルの成分を組にして乗算し、その和を取

ることで計算できます。2 つのベクトルの内積は、2 つのベクトルが成す角を求めるのに便利です。

- vmult 関数はベクトルをスカラー倍します。ベクトルと数（スカラー）の乗算は、ベクトルの各要素にスカラーを掛け算することで実現され、その結果、ベクトルは同じ方向を指したまま、長さが伸びたり縮んだりします。

これらの関数は、Vector a という形であればどんな型でも扱えます。ただし、a は Num 型クラスのインスタンスであるという条件付きです。例えば、Vector Int や Vector Integer 、Vector Float などは、Int も Integer も Float も型クラス Num のインスタンスなので扱うことができますが、Vector Char や Vector Bool は扱うことができません。

また、型宣言をよく見ると分かるように、ベクトルを扱う関数は要素型が同じベクトルどうししか演算できず、演算にからむスカラーも要素と同じ型である必要があります。Vector Int と Vector Double を加算することはできないのです。

データ宣言には Num 型クラス制約が付いていませんね。前の節で説明したとおり、データ宣言に Num を付けたところで、関数に付いている Num を省くことはできません。

念を押しておきますが、型コンストラクタと値コンストラクタを区別しておくことはとても重要です。データ型を宣言するとき、= の前にあるのが型コンストラクタであり、後ろにある（あるいは | で区切られたその後ろにある）のが値コンストラクタです。例えば、関数にこのような型を与えるのは間違いです。

```
Vector a a a -> Vector a a a -> a
```

これが動かないのは、ベクトルの型は Vector a であって、Vector a a a ではないからです。ベクトルは、型としてはあくまで引数を 1 つだけ取ります。値コンストラクタは 3 つの引数を取りますが、それは別の話です。

では、作ったベクトルで遊んでみましょう。

```
ghci> Vector 3 5 8 `vplus` Vector 9 2 8
Vector 12 7 16
ghci> Vector 3 5 8 `vplus` Vector 9 2 8 `vplus` Vector 0 2 3
Vector 12 9 19
ghci> Vector 3 9 7 `vmult` 10
Vector 30 90 70
ghci> Vector 4 9 5 `dotProd` Vector 9.0 2.0 4.0
74.0
ghci> Vector 2 9 3 `vmult` (Vector 4 9 5 `dotProd` Vector 9 2 4)
Vector 148 666 222
```

7.5　インスタンスの自動導出

27 ページの「型クラス初級講座」という節では、型クラスはある振る舞いを定義するインターフェイスであり、ある型がその振る舞いをサポートしていれば、その型クラスのインスタンスにできる、ということを学びました。例えば、Int 型は Eq 型クラスのインスタンスです。これは、Eq 型クラスは「等値性テストができる物」という振る舞いを定義しているからです。整数には等値性が定義されていますから、Int は Eq 型クラスに属することになります。これが本当に便利なのは、Eq のインターフェイスとなる、== と /= という関数があるからです。ある型が Eq 型クラスのインスタンスであるなら、その型の値には == が使えます。これこそが、`4 == 4` と `"foo" == "bar"` が両方とも型検査を通る仕組みです。

Haskell の型クラスは、Java、Python、C++ といった言語の「クラス」と紛らわしいので、多くのプログラマが引っかかりやすいところです。そういったオブジェクト指向言語でクラスといえば、何らかの動作をするオブジェクトを作るための青写真のことです。でも、Haskell のクラスは、データを作る道具ではありません。そうではなく、まずデータ型を作り、それから「このデータには何ができるだろう？」と考えるのです。もしその型が、等値性をテストできるものであれば、Eq 型クラスのインスタンスにします。その型が大小比較できるものであれば、Ord 型クラスのインスタンスにします。

Haskell は、特定の型クラスのインスタンス宣言を自動導出（derive）する能力を備えています。自動導出できる型クラスは Eq 、 Ord 、 Enum 、 Bounded 、 Show 、 Read です。自作のデータ型を作るとき、 **deriving** キーワードを使えば、Haskell がこれらの型クラスの文脈での振る舞いを自動導出してくれます。

人間の平等

このデータ型を見てください。

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     }
```

これは人間を表しています。今、この世に同姓同名で年齢まで一致する人はいない、と仮定しましょう。では、二人の人の記録があるとき、その 2 つの記録が同一人物を表しているか判定することは妥当でしょうか？ もちろん、判定でき

てしかるべきです。ですから、この型を Eq 型クラスに属させるのは妥当なことです。インスタンス宣言は自動導出してもらいましょう。

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq)
```

ある型に Eq を自動導出して == や /= で比較しようとすると、Haskell はまず値コンストラクタが一致しているかを調べます（今は 1 つしかありませんが）。それから、値コンストラクタの中に入っている各フィールドがすべて一致しているか、それぞれの組を == を使って比較します。ただし落とし穴が 1 つあります。すべてのフィールドの型が、 Eq 型クラスのインスタンスでないと自動導出は使えません。今回の場合は、 String と Int はこの条件を満たすからオッケーですね。

まずは人物を何人か作ってみましょう。スクリプトに次のように書いてください。

```
mikeD = Person {firstName = "Michael", lastName = "Diamond", age = 43}
adRock = Person {firstName = "Adam", lastName = "Horovitz", age = 41}
mca = Person {firstName = "Adam", lastName = "Yauch", age = 44}
```

Eq インスタンスを試してみましょう。

```
ghci> mca == adRock
False
ghci> mikeD == adRock
False
ghci> mikeD == mikeD
True
ghci> mikeD == Person {firstName = "Michael", ⇨
                       lastName = "Diamond", age = 43}
True
```

もちろん、今や Person は Eq ですから、 Eq a という型クラス制約の付いた関数の a のところならどこにでも使えます。例えば、 elem とか。

```
ghci> let beastieBoys = [mca, adRock, mikeD]
ghci> mikeD `elem` beastieBoys
True
```

読み方を書いてみせてよ

Show と Read はそれぞれ文字列へ変換できるものの型クラス、および文字列から変換できるものの型クラスです。 Eq のときと同じく、ある型を Show や Read のインスタンスにしたいなら、その型の値コンストラクタにフィールドがあれば、それらの型も Show や Read に属している必要があります。

では、 Person データ型を Show と Read にも属させてみましょう。

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     } deriving (Eq, Show, Read)
```

これで人物の情報をターミナルに出力できます。

```
ghci> mikeD
Person {firstName = "Michael", lastName = "Diamond", age = 43}
ghci> "mikeD is: " ++ show mikeD
"mikeD is: Person {firstName = \"Michael\", lastName = \"Diamond\",
age = 43}"
```

もし Person を Show のインスタンスにする前に、人物をターミナルに表示させようとしていたら、Haskell は「人物を文字列で表現する方法が分からない」と主張し、いうことを聞いてくれなかったでしょう。でも、あらかじめ Show のインスタンスにしておいたので文句は言われませんでした。

Read は Show のちょうど逆をする型クラスです。ただし、 read 関数を使うときは、文字列をどの型へ変換したいのか Haskell に伝えるために、明示的な型注釈を使う必要があるかもしれないことをお忘れなく。実験のため、人物を表す文字列をスクリプトに書いておいて、それを GHCi から読み込んでみます。

```
mysteryDude = "Person { firstName =\"Michael\"" ++
                      ", lastName =\"Diamond\"" ++
                      ", age = 43}"
```

読みやすいように、文字列は複数行にして書きました。この文字列を read するためには、どの型が返ることを期待しているのか、Haskell に伝える必要があります。

```
ghci> read mysteryDude :: Person
Person {firstName = "Michael", lastName = "Diamond", age = 43}
```

ただし、Haskell が「これは人物を読み取るべきだな」と型推論できるような仕方で read の結果を使っている場合には型注釈は不要です。

```
ghci> read mysteryDude == mikeD
True
```

多相型も読み取ることができますが、どの型が欲しいか Haskell が推論できるだけの情報を与える必要があります。例えば、こんなのを試すとエラーになります。

```
ghci> read "Just 3" :: Maybe a
```

これでは型引数 a にどの型が入るのか、Haskell には分かりませんね。でも、そこは Int にしたいんだよ、と教えてあげると、問題なく動きます。

```
ghci> read "Just 3" :: Maybe Int
Just 3
```

順番を守ってください！

順序付け可能な型のための型クラス、 Ord のインスタンスも自動導出できます。同じ型の値を 2 つ比較したとき、もし 2 つが異なる値コンストラクタから作られたものなら、先に定義されているほうが小さいとみなされます。例えば、Bool は False と True の値を取ります。 Bool を比較すると何が起こるかは、Bool が以下のように定義されていると考えれば分かります。

```
data Bool = False | True deriving (Ord)
```

False 値コンストラクタが先に定義されていて、その後で True が指定されているので、Ord を自動導出すると、True は False より大きいのだろうと考えられます。

```
ghci> True `compare` False
GT
ghci> True > False
True
ghci> True < False
False
```

2 つの値が同じ値コンストラクタでできている場合、フィールドがなければ 2 つは等しいとされます。フィールドがあれば、フィールドどうしが比較され、どちらが大きいか決まります（この場合、フィールドの型もまた Ord に属している必要があります)。

Maybe a データ型では、 Nothing 値コンストラクタが Just 値コンストラクタの前に定義されているので、 Nothing 値は常に Just something より小さいとされます。 something がたとえマイナス一億兆万であっても Nothing のほうが小さいのです。でも、2 つの Just 値を指定したときは、Haskell は中身を比較してくれます。

```
ghci> Nothing < Just 100
True
ghci> Nothing > Just (-49999)
False
ghci> Just 3 `compare` Just 2
GT
ghci> Just 100 > Just 50
True
```

でも、 Just (*3) > Just (*2) みたいなことはできません。 (*3) や (*2) は関数の型を持ち、関数は Ord のインスタンスではないからです。

何曜日でもいいよ

代数データ型を使えば列挙型は簡単に作ることができます。その際には Enum と Bounded 型クラスが便利です。こんなデータ型を考えてみてください。

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday |
           Saturday | Sunday
```

すべての値コンストラクタがゼロ引数（フィールドを持っていない）なので、これを Enum 型クラスに属させることができます。 Enum は、前者関数と後者関数を持つ型のための型クラスです。 Day は上限と下限を持つ型のクラスである Bounded のインスタンスにもできます。ついでに自動導出できるすべての型クラスのインスタンスにしちゃいましょう。

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday |
           Saturday | Sunday
            deriving (Eq, Ord, Show, Read, Bounded, Enum)
```

では、Day 型に何ができるのか見ていきましょう。まず Day は、Show と Read のインスタンスですから、その値を文字列にしたり文字列から変換したりできます。

```
ghci> Wednesday
Wednesday
ghci> show Wednesday
"Wednesday"
ghci> read "Saturday" :: Day
Saturday
```

また、型クラス Eq と Ord のインスタンスでもあるので、等号や不等号が使えます。

```
ghci> Saturday == Sunday
False
ghci> Saturday == Saturday
True
ghci> Saturday > Friday
True
ghci> Monday `compare` Wednesday
LT
```

さらに、 Bounded のインスタンスでもありますから、上限と下限を取ることができます。

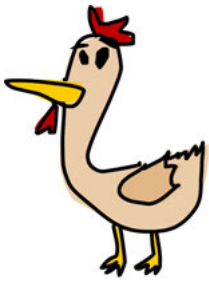
```
ghci> minBound :: Day
Monday
```

```
ghci> maxBound :: Day
Sunday
```

Enum のインスタンスもあるので、昨日の曜日や明日の曜日を知ることができるし、範囲を指定してリストを作ることもできます！

```
ghci> succ Monday
Tuesday
ghci> pred Saturday
Friday
ghci> [Thursday .. Sunday]
[Thursday,Friday,Saturday,Sunday]
ghci> [minBound .. maxBound] :: [Day]
[Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday]
```

7.6 型シノニム



前にも話したように、[Char] と String は同値で、交換可能です。これは**型シノニム**（型同義名）を使って実装されています。

型シノニムそのものは特に何もしません。ある型に別の名前を与えて、コードやドキュメントを他の人が読みやすくするだけです。標準ライブラリでの String の定義は、こんな感じで [Char] の型シノニムになっています。

```
type String = [Char]
```

type というキーワードはちょっと誤解を招きやすいかもしれません。ここでは既存の型のシノニムが定義されているのであって、新しい型が作られているわけではありません（それは **data** の役割です）。

英文字列を全部大文字に変える関数 toUpperString を作るとしたら、その型宣言はこうできます。

```
toUpperString :: [Char] -> [Char]
```

こんな型宣言も使えます。

```
toUpperString :: String -> String
```

2 つは本質的に同じものですが、後者のほうが読みやすいですね。

電話帳をかっこよくしよう

前に Data.Map モジュールを扱ったとき、電話帳をまずは連想リスト（キー／値のペアのリスト）で表し、それから Map に変換しました。これが連想リストバージョンです。

```
phoneBook :: [(String, String)]
phoneBook =
    [("betty", "555-2938")
    ,("bonnie", "452-2928")
    ,("patsy", "493-2928")
    ,("lucille", "205-2928")
    ,("wendy", "939-8282")
    ,("penny", "853-2492")
    ]
```

phoneBook の型は [(String, String)] です。この型宣言からは、これが文字列から文字列への連想リストであることが読み取れますが、それ以上の情報はありません。型宣言にもっと有益な情報を載せるため、型シノニムを作りましょう。

```
type PhoneBook = [(String,String)]
```

これで電話帳の型は phoneBook :: PhoneBook になりました。 String のシノニムも作りましょう。

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name, PhoneNumber)]
```

Haskell プログラマは、自分のプログラムの中で使っている文字列について「ただの文字列じゃなくて実際はこれを表しているんだよ」という情報を伝えたいとき、 String に型シノニムを与えます。

というわけで、名前と電話番号を取って、その名前と電話番号の組が電話帳に載っているか調べる関数を実装するときにも、何をする関数なのか一目で分かるかっこいい型宣言を書けるようになりましたよ。

```
inPhoneBook :: Name -> PhoneNumber -> PhoneBook -> Bool
inPhoneBook name pnumber pbook = (name, pnumber) `elem` pbook
```

型シノニムを使わなかったら、この関数の型はこうなっていたでしょう。

```
inPhoneBook :: String -> String -> [(String, String)] -> Bool
```

この場合は型シノニムを使った型宣言のほうが理解しやすいですね。しかし、型シノニムは使いすぎてもいけません。型シノニムを作るのは、自作の関数内で既存の型が何を表してるかを示すことで型宣言のドキュメントとしての質を高

めるためか、何度も出てくる長々しい型（[(String, String)] みたいな）が自作の関数の文脈では何か特定のものを表している場合に限ります。

型シノニムの多相化

型シノニムも型引数を取るようにできます。例えば、連想リストを表す型を作りたいけどキーや値の型は特定せず汎用にしておきたい、というのであれば、こう書けます。

```
type AssocList k v = [(k, v)]
```

これで、連想リストからキーを検索してくれる関数の型を (Eq k) => k -> AssocList k v -> Maybe v と書けるようになりました。AssocList は 2 つの型を引数に取って、AssocList Int String のような具体型を返す型コンストラクタです。

関数を部分適用して新しい関数を作れるのと同じように、型引数を部分適用すると新しい型コンストラクタが作れます。関数を呼ぶときに引数の数が足りないと、残りの引数を取る新しい関数が返ってくるのでしたね。同じように、型コンストラクタに型引数を一部しか与えないと、残りの型引数を取る型コンストラクタが返ってきます。例えば、Data.Map を使い、Int をキーとして何らかの値を返す Map の型を作りたければ、こうです。

```
type IntMap v = Map Int v
```

または、こう。

```
type IntMap = Map Int
```

どちらの書き方にしても、IntMap は引数を 1 つ取る型コンストラクタになり、その引数こそが Int が指す値になります。

これを実装するなら、たぶん Data.Map を修飾付きインポートしたくなると思いますよ。修飾付きインポートをするときは、型コンストラクタの前にもモジュール名をつける必要があります。

```
type IntMap = Map.Map Int
```

型コンストラクタと値コンストラクタの違いをちゃんと理解できてますか？IntMap や AssocList という名の型シノニムを作ったからといって、AssocList [(1,2),(4,5),(7,9)] のようなものが作れるようになるわけではありませんよ。できるのは、すでにあるものの型を別の名前で呼ぶことだけです。[(1,2),(3,5),(8,9)] :: AssocList Int Int と書くことはできます。こうすることで、中身の数が Int 型だと推論されます。このリストの型には特別な

名前がついてはいますが、整数のペアのリストが使える場所ならどこででも使うことができます。

Haskell のソースコードは、値の領域、型の領域などに分かれていると考えられます。型の領域に属するのは、データ型や型シノニムの宣言、それから型宣言や型注釈などに登場する記号 :: の右側、などです。型シノニムや、そのほか型の領域に属するものは、Haskell の一部である型の領域でしか使えないのです。AssocList [(1,2),(4,5),(7,9)] は、型の領域に属するべき AssocList を値の領域で使っているので、文法的に誤りだと言えます。

そこを左に行って、すぐ右へ

型引数を 2 つ取るデータ型といえば Either a b もあります。 Either の定義は、だいたいこんな感じです。

```
data Either a b = Left a | Right b deriving (Eq, Ord, Read, Show)
```

Either a b には値コンストラクタが 2 つあります。 Left を使うと、Either の中身は a 型になります。 Right を使ったときは、中身は b 型になります。つまり、 Either 型を使って「2 つの型のうちどちらか一方」という値を表せます。Either a b 型の値を受け取り、 Left と Right をパターンマッチして、どちらの型に合致したかに応じて異なる処理をする、という使い方をよくします。

```
ghci> Right 20
Right 20
ghci> Left "w00t"
Left "w00t"
ghci> :t Right 'a'
Right 'a' :: Either a Char
ghci> :t Left True
Left True :: Either Bool b
```

この例では、 Left True の型を評価すると Either Bool b という型になっていることが分かります。 Left 値コンストラクタで作った値なので、1 つ目の型引数は Bool に決まっていますが、2 つ目の型引数は多相のまま残っています。これは、 Nothing という値に Maybe a という型が付くのと同じ理屈です。

これまでのところ、失敗したかもしれない計算を表現するには Maybe a を使ってきました。でも、 Maybe a では足りない状況もあります。 Nothing では「何かが失敗した」以上の大した情報を持てないからです。 Data.Map の lookup であれば、失敗するのはキーが Map に入ってなかったときだけなので何が起こったかは明白であり、 Nothing で済みます。

でも、関数がなぜ失敗したのか、どのように失敗したのかを知りたいとき、普通は Either a b 型の返り値を使います。ここで、a は失敗が起こった場合に何

であるかを伝えてくれる型、b は成功した計算の型です。したがって、エラーは Left 値コンストラクタを、結果は Right 値コンストラクタを使って表します。

例として、生徒の一人一人にロッカーが割り当てられる高校を考えてみましょう。それぞれのロッカーには暗証番号が付いています。新しいロッカーの割り当てが必要な生徒は、ロッカー管理人に欲しいロッカーの番号を伝え、管理人が暗証番号を渡します。しかし、そのロッカーを誰かがすでに使っている場合、生徒は新しいロッカーを選び直す必要があります。ロッカー全体を Data.Map の Map で表しましょう。ロッカー番号から、ロッカーが使用中かどうかのフラグと、ロッカーの暗証番号への Map です。

```
import qualified Data.Map as Map

data LockerState = Taken | Free deriving (Show, Eq)

type Code = String

type LockerMap = Map.Map Int (LockerState, Code)
```

ロッカーが埋まっているか空いてるかを表す新しいデータ型 LockerState を導入しました。また、ロッカーの暗証番号には型シノニム Code を与えました。さらに、整数から「ロッカーの状態と暗証番号の組」への Map にも型シノニム LockerMap を与えました。

続いて、ロッカーを表す Map から暗証番号を検索する関数を作りましょう。この関数の結果は Either String Code 型で表すことにします。この関数が失敗するパターンは 2 通りあるからです。ロッカーを他の誰かが使っている場合に暗証番号を伝えるわけにはいきません。それから、ロッカー番号が存在しない可能性もあります。検索が失敗したときは、単に String を返して何が起こったか説明することにしましょう。

```
lockerLookup :: Int -> LockerMap -> Either String Code
lockerLookup lockerNumber map = case Map.lookup lockerNumber map of
    Nothing -> Left $ "Locker " ++ show lockerNumber
                    ++ " doesn't exist!"
    Just (state, code) -> if state /= Taken
                          then Right code
                          else Left $ "Locker " ++ show lockerNumber
                                      ++ " is already taken!"
```

まずは普通に Map を lookup します。 lookup が Nothing を返したら、Left String 値コンストラクタを使い、「ロッカーがないよ（doesn't exist）」という返事を返します。ロッカーが見つかった場合は、そのロッカーが使用中かどうか調べます。使われていれば「もう埋まっているよ（already taken!）」という Left 値を返します。空いていれば Right Code 値を返して、生徒さんに正し

い暗証番号を伝えます。これ、実は Right String なんですが（もっというと Right [Char] ですが）、ドキュメントを充実させるために型シノニム Code を作ったのでしたね。

次のような Map があったとしましょう。

```
lockers :: LockerMap
lockers = Map.fromList
    [(100,(Taken, "ZD39I"))
    ,(101,(Free, "JAH3I"))
    ,(103,(Free, "IQSA9"))
    ,(105,(Free, "QOTSA"))
    ,(109,(Taken, "893JJ"))
    ,(110,(Taken, "99292"))
    ]
```

ロッカーを検索してみます。

```
ghci> lockerLookup 101 lockers
Right "JAH3I"
ghci> lockerLookup 100 lockers
Left "Locker 100 is already taken!"
ghci> lockerLookup 102 lockers
Left "Locker number 102 doesn't exist!"
ghci> lockerLookup 110 lockers
Left "Locker 110 is already taken!"
ghci> lockerLookup 105 lockers
Right "QOTSA"
```

結果を表すのに Maybe a を使うという手もありましたが、そうしていたらロッカーの取得に失敗しても原因が分からなかったでしょう。でも、今は関数の返り値の型が失敗の情報を伝えられるようになっています。

7.7　再帰的なデータ構造

すでに見たように、代数データ型の値コンストラクタは複数のフィールドを持つこともできるし、フィールドを持たないこともできます。そして各フィールドの型は具体型である必要があります。それなら、フィールドに持つ型は自分自身でもかまわないってことですね！ つまり、再帰的なデータ型（ある型の値の一部にまた同じ型の値が入っていて、そのまた一部にまたまた同じ型の値が入っていて、……というデータ型）が作れるってことです。

[5] というリストで考えてみましょう。これは実は 5:[] の構文糖衣です。 : の左辺には値が 1 つあ

ります。右辺にはリストがあります。今の場合は空リストですね。では、[4,5] というリストはどうでしょう？ 構文糖衣を脱がすと、こいつは 4:(5:[]) です。先頭の : を見ると、やっぱり左辺には値が 1 つあり、右辺にはリスト (5:[]) があります。 3:(4:(5:6:[])) のような長いリストも仕組みは同じです。この式は、 : は右結合であることを使って 3:4:5:6:[] とも書けますし、構文糖衣を使えば [3,4,5,6] とも書けます。

リストは、「空リスト」または「要素とリスト（空でもよい）を : で結合したもの」のいずれかの値を取るデータ構造です。

では、代数データ型を使って独自のリスト型を実装してみましょう！

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```

この型は確かにリストの定義を満たしています。 List は空リストであるか、head となる値とリストの結合であるかのいずれかです。これが分かりにくいという人は、レコード構文で考えると分かりやすいかもしれません。

```
data List a = Empty | Cons { listhead :: a, listTail :: List a}
    deriving (Show, Read, Eq, Ord)
```

ここに出てくる Cons コンストラクタも分かりにくいかもですね。ぶっちゃけ、 Cons というのは : を言い換えたものです。Haskell 標準のリストにおける : も、値とリストを取ってリストを返す値コンストラクタなのです。別の言い方をすると、 : には a 型と List a 型の 2 つのフィールドがある、ということです。

```
ghci> Empty
Empty
ghci> 5 `Cons` Empty
Cons 5 Empty
ghci> 4 `Cons` (5 `Cons` Empty)
Cons 4 (Cons 5 Empty)
ghci> 3 `Cons` (4 `Cons` (5 `Cons` Empty))
Cons 3 (Cons 4 (Cons 5 Empty))
```

: との類似を明らかにするため、 Cons コンストラクタを中置記法で呼び出しています。 Empty は [] に対応し、例えば 4 `Cons` (5 `Cons` Empty) は 4:(5:[]) にあたります。

リストの改善

記号文字だけを使って関数に名前をつけると、自動的に中置関数になります。値コンストラクタもデータ型を返す関数なので、同じルールに従います。ただし 1 つだけ制限があります。中置関数にする場合、値コンストラクタの名前はコロンで始まる必要があります。これを見てください。

```
infixr 5 :-:
data List a = Empty | a :-: (List a) deriving (Show, Read, Eq, Ord)
```

まず、新しい構文要素に注目してください。データ宣言の前の行にある結合性宣言です。関数を演算子として定義した場合、その**結合性**（fixity）を宣言できます。結合性宣言は必須ではありません。結合性宣言では、演算子の結合順位や、左結合なのか右結合なのかを指定します。例えば、`*` 演算子の結合性は `infixl 7 *` で、`+` 演算子の結合性は `infixl 6` です。これは、`*` も `+` も左結合（`4 * 3 * 2` が `(4 * 3) * 2` と等しい）だけど `*` は `+` より強く結合することを意味します。結合性宣言の数字が大きいからです。つまり、`5 + 4 * 3` は `5 + (4 * 3)` と同じ意味になります[†3]。

結合性宣言を除くと、`Cons a (List a)` を `a :-: (List a)` に書き換えただけです。これでリスト型に属するリストをこう書けます。

```
ghci> 3 :-: 4 :-: 5 :-: Empty
3 :-: (4 :-: (5 :-: Empty))
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> 100 :-: a
100 :-: (3 :-: (4 :-: (5 :-: Empty)))
```

次に、2つのリストを結合する関数を作りましょう。標準のリストにおける `++` の定義はこうなっています。

```
infixr 5  ++
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

このコードを僕らのリストにも流用しましょう。`^++` という関数を作ります。

```
infixr 5  ^++
(^++) :: List a -> List a -> List a
Empty ^++ ys = ys
(x :-: xs) ^++ ys = x :-: (xs ^++ ys)
```

試してみましょう。

```
ghci> let a = 3 :-: 4 :-: 5 :-: Empty
ghci> let b = 6 :-: 7 :-: Empty
ghci> a ^++ b
3 :-: (4 :-: (5 :-: (6 :-: (7 :-: Empty))))
```

必要とあらば、標準リストを扱う関数を全部、僕らのリスト型に移植することも可能です。

---

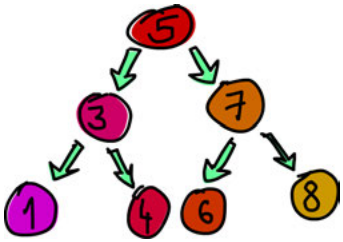
†3 ［訳注］ちなみに、結合性宣言を省略した演算子はすべて `infixl 9` になります。`` ` `` で演算子化した関数に対しても結合性を宣言でき、省略した場合は同様に `infixl 9` になります。

さりげなく (x :-: xs) というパターンマッチを使っていることに注目してください。これが可能なのは、パターンマッチとは値コンストラクタをマッチさせることにほかならないからです。 :-: は、僕らが作ったリストの値コンストラクタであり、 : は Haskell 組み込みのリストの値コンストラクタなので、どちらも当然パターンマッチできます。 [] がパターンマッチできるのも同じ理由からです。パターンマッチは値コンストラクタであれば何に対してでも使えるので、通常の前置コンストラクタに加えて 8 や 'a' といったものもパターンマッチできます。これらは数値型や文字型の値コンストラクタだからです。

木を植えよう

Haskell の再帰的データ構造にもっと慣れるために、二分探索木を実装してみましょう。

二分探索木では、1 つの要素が 2 つの子要素へのポインタを持ちます。一方は左の子で、もう一方は右の子です。左の子は親より小さく、右の子は親より大きいようにしておきます。それぞれの子要素は、さらに 2 つ（あるいは 1 つ、または 0 個）の要素を持ちます。結果として、各要素は 2 つ以下の部分木を持つことになります。



二分探索木がすごいのは、例えば 5 という要素の左部分木の全要素は 5 より小さいことが保証されているところです。右部分木の要素は、すべて 5 より大きくなります。ですから、この木に 8 が含まれているか調べたければ、5 から探索を始め、8 は 5 より大きいので、右に行きます。そこが 7 なので、また右に行きます。すると、あっ！ たったの 3 歩で探しものが見つかりました！ これが普通のリスト（あるいは、ちっとも平衡してない木）だったら、8 が含まれているか調べるのに 7 歩かかったことでしょう。

NOTE Data.Set や Data.Map が提供する Set や Map も木構造を使って実装されていますが、ただの二分探索木ではなく、平衡二分探索木を使っています。木構造の平衡とは、左右の要素の深さがだいたい等しくなっていることを指します。平衡木の探索は普通の木より速くなります。が、ここでは通常の二分探索木を実装することにしましょう。

木構造とは、空の木、もしくは何らかの値と 2 つの木を含む要素からなる構造です。それって代数データ型で表してくれといわんばかりの構造ですね！

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show)
```

手作業で木を作る代わりに、木と要素を取って要素を木に挿入する関数を作りましょう。まず、新しい値をルートの要素と比較します。その値がルートより小さければ左に、大きければ右に行きます。空の木に辿り着くまで同じことを繰り返します。空の木が見つかったなら、そこに新しい値を保持するノードを追加します。

C のような言語なら、このような操作はポインタや値の更新を使って書くでしょう。Haskell では、木を直接更新できないので、左右どちらに行くのか決めるたびに新しい部分木を作っていく必要があります。最終的に挿入関数は新しい木全体を作って返します。Haskell にはポインタという概念はなく、値しかないからです。したがって、これから作る挿入関数の型は `a -> Tree a -> Tree a` みたいになります。これは、ある要素とある木を取って、要素が挿入された新しい木を返す関数です。こう言うと効率が悪いように聞こえるかもしれませんが、Haskell には古い木と新しい木の部分構造のほとんどを共有する仕組みが備わっているので心配無用です。

これが木を作るための 2 つの関数です。

```
singleton :: a -> Tree a
singleton x = Node x EmptyTree EmptyTree

treeInsert :: (Ord a) => a -> Tree a -> Tree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)
```

`singleton` は、要素が 1 つしかない木を作るための補助関数です。ルートに何かが入っていて左右の部分木は空であるようなノードを手軽に作れるように定義しただけです。

要素 `x` を木に挿入するための関数は `treeInsert` です。まずは、再帰の基底部をパターンマッチで表現しています。挿入先が空の木である場合は、目的地に辿り着いたということなので、`x` を唯一の要素として持つ木を挿入します。挿入先が空の木でない場合はちょっと調査が必要です。まず、新しい要素とルート要素が等しければ、すでにその要素は挿入されているということなので、元の木をそのまま返します。新しい要素のほうが小さければ、「ルートの値と右部分木は元のままで、新しい要素が挿入された左部分木を持つ」木を返します。新しい要素のほうが大きかった場合は、新しい要素を右部分木に挿入した木を同様にして返します。

では次に、ある要素が木に属しているか判定する関数を作りましょう。

```
treeElem :: (Ord a) => a -> Tree a -> Bool
treeElem x EmptyTree = False
treeElem x (Node a left right)
    | x == a = True
    | x < a  = treeElem x left
    | x > a  = treeElem x right
```

まずは再帰の基底部を定義します。目の前にあるのが空の木なら、要素がそこにないことは確実です。これって、リストの中から要素を探す場合の基底部にそっくりですよね。空じゃない木から探す場合は調査が必要です。ルート要素が探しているものと等しければ大成功！ 等しくなかったら、どうしましょう？ 今こそ、左部分木のすべての要素はルートより小さいという情報を活用するときです！ 探している値がルート値より小さければ左部分木を調べ、大きければ右部分木を調べます。

それでは木で遊びましょう！ 手作業で作るのは面倒だからやめて（やろうと思えばできますが）、畳み込みを使ってリストから木を作りましょう。リストを1要素ずつ辿って値を返す操作はたいがい畳み込みで実装できるってこと、覚えておいてください！ 空の木から始め、リストを右から辿ってアキュムレータ木に要素を追加していきましょう。

```
ghci> let nums = [8,6,4,1,7,3,5]
ghci> let numsTree = foldr treeInsert EmptyTree nums
ghci> numsTree
Node 5
    (Node 3
        (Node 1 EmptyTree EmptyTree)
        (Node 4 EmptyTree EmptyTree)
    )
    (Node 7
        (Node 6 EmptyTree EmptyTree)
        (Node 8 EmptyTree EmptyTree)
    )
```

NOTE このコードを GHCi で走らすと、numsTree の結果は 1 行で表示されるはずですが、ここでは複数行に分けて書いています。さもないとページをはみ出してしまいますからね！

この foldr の畳み込みでは、treeInsert （リストの要素と木を取って新しい木を作る）が2引数関数であり、EmptyTree が初期のアキュムレータです。nums は、もちろん、畳み込み対象のリストです。

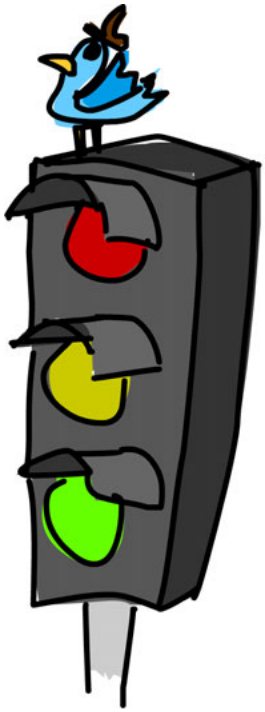
コンソールに表示される木は読みやすいものではありませんが、それでも何となく構造は分かります。ルートノードはどうやら 5 のようです。2つの部分木を持っています。それぞれのルートノードは 3 と 7 です。

ある値が木に含まれているかも調べられます。

```
ghci> 8 `treeElem` numsTree
True
ghci> 100 `treeElem` numsTree
False
ghci> 1 `treeElem` numsTree
True
ghci> 10 `treeElem` numsTree
False
```

ご覧のとおり、Haskell の代数データ型はとってもパワフルでかっこいい概念です。真理値や曜日の列挙型を手始めに、マジで何でも作れるんです！

7.8　型クラス 中級講座



ここまで、Haskell に標準で付いてくる型クラスや、それらにどの型が属しているのかを学んできました。また、Haskell に自動導出してもらうことで独自の型を標準型クラスのインスタンスにする方法も学びました。この節では、独自の型クラスを作り、そのインスタンスを手動で作る方法を学びます。

型クラスについて軽く復習しておきましょう。型クラスはインターフェイスのようなものです。型クラスは、特定の振る舞い（等値性判定だとか、順序の比較だとか、列挙だとか）を定義します。定義されたとおりに振る舞うことができる型は、その型クラスのインスタンスにされます。型クラスの振る舞いは、型クラス関数を定義することで得られます。型宣言だけして実装は後回しにしてもかまいません。というわけで、ある型 `T` がある型クラス `C` のインスタンスであるとは、型クラス `C` が定義する関数（メソッド）たちを型 `T` に対して使える、ということを意味します。

NOTE　再確認ですが、「型クラス」は Java や Python のような言語に出てくる「クラス」とは何の関係もありません。混乱しちゃう人が多いんで、手続き型言語のクラスについて知っていることは今すぐ全部忘れてほしいな！

Eq 型クラスの内部

Eq 型クラスを例に取りましょう。Eq は等値性判定ができる値の型クラスでしたね。Eq は == と /= という関数（メソッド）を定義しているのでした。今、Car という型があって、2 つの自動車を等値関数 == で比較することに意味があるのなら、Car を Eq のインスタンスにするのが理にかなっています。

これが標準ライブラリにおける Eq の定義です。

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

うおあ！ 何か変な文法とキーワードが出てきました！

まず、**class** Eq a **where** は Eq という名の新しい型クラスの定義が始まることを意味します。a は型変数で、将来 Eq のインスタンスとなるであろう型を表します。（ちなみに、名前が a である必要もなく、1 文字である必要もありません。小文字から始まっていればよろしい。）

次に、関数がいくつか定義されています。ただし、関数定義の実体を与えなくてもかまいません。必須なのは型宣言だけです。ここでは、Eq の定義する関数の実体が実装されています（デフォルト実装と呼びます）。相互再帰という形で。そのソースを読むと、「Eq に属する型を持つ 2 つの値は、それらが互いに異ならないならば等しく、互いに等しくないならば異なっている。」と書いてあります。こんなのが一体何の役に立つのか、すぐに分かりますよ。

型クラス定義に含まれる関数には、最終的にちょっと特別な型が付きます。例えば、**class** Eq a **where** と宣言して、それからクラスの中で (==) :: a -> a -> Bool みたいな型宣言をしたとします。後でその関数の型を調べると、(Eq a) => a -> a -> Bool という型になっているでしょう。

交通信号データ型

こうして作ったクラスですが、これで何ができるでしょう？ 型をこのクラスのインスタンスにして、クラスの便利な機能を使うことができます。例えばこの型を見てください。

```
data TrafficLight = Red | Yellow | Green
```

これは交通信号の状態を定義する型です。ご覧のとおり、自動導出は使っていません。これは、インスタンスを手で書いてみることが目的だからです。これが Eq のインスタンスの作り方です。

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

instance キーワードを使ってインスタンスを作りました。そう、新しい型クラスを定義するのが **class** で、型を型クラスのインスタンスにするのが **instance** なのです。 Eq を定義したときには、 **class** Eq a **where** と書き、 a は将来インスタンスになり得るあらゆる型の 1 つを表しているんだよと言いました。ここで、その言葉の意味が明らかになりましたね。なぜなら、インスタンスを作るにあたり **instance** Eq TrafficLight **where** と書いているからです。クラス定義の a がまさに実際の型で置き換えられています。

さて、クラスを宣言したときには、 == を定義するのに /= を使い、逆に /= を定義するのにも == を使っていました。そのため、インスタンス宣言ではどちらか一方だけを上書きすればよいことになります。これは、型クラスの**最小完全定義**（minimal complete definition）と呼ばれる概念です。インスタンスになろうとする型をクラスの宣伝文句のとおりに振る舞わせるために、最低限定義する必要のある関数たちがあるということです。 Eq の最小完全定義を満たすには、 == か /= のいずれかを上書きする必要があります。もし Eq の定義がこれだけだったら、

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

Eq のインスタンスを作るには、両方の関数を定義する必要があったでしょう。Haskell には、この 2 つの関数の間に何か関係があるなんて分かりませんから。この場合の最小完全定義は == と /= の両方ということになったでしょう。

見てのとおり、ここでは単純なパターンマッチを使って == のほうを定義しました。2 つの信号が**等しくない**ケースはもっといろいろあるので、まず**等しい**場合をすべて指定し、最後に必ず合致するパターンを置いて「以上の組み合わせに当てはまらない場合は、等しくないとする」と言ったのです。

Show のインスタンスにするのも手動でやってみましょう。 Show の最小完全定義を満たすには、値を取って文字列に変える show 関数を定義すれば十分です。

```
instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"
```

またしてもパターンマッチを使って実装しました。実際にどう動くか使ってみましょう。

```
ghci> Red == Red
True
ghci> Red == Yellow
False
ghci> Red `elem` [Red, Yellow, Green]
True
ghci> [Red, Yellow, Green]
[Red light,Yellow light,Green light]
```

Eq は自動導出したとしても同じ効果が得られたでしょう（手動でやったのは教育目的）。ところが Show の自動導出を使うと、値コンストラクタがそのまま文字列に変換されて出るだけです。信号が Red light とかに表示されてほしいなら、インスタンス宣言を手動で書く必要があります。

サブクラス化

別の型クラスのサブクラスである型クラスを作ることもできます。例えば Num の型クラス宣言は、全体はちょっと長いですが最初の部分はこんなふうになっています。

```
class (Eq a) => Num a where
  ...
```

前にも言ったように、型クラス制約を挟める場所というのがいろいろあります。この例は、普通に **class** Num a **where** と書くのに似ていますが a が Eq のインスタンスになっている必要があると言っています。要するに、ある型を Num のインスタンスにしたかったら、その前に Eq のインスタンスにする必要がある、と言っているのです。「ある型を数のようなものとしたいなら、等値比較くらいできないといけない」というのは、うなずける話ですね。このようなとき、 Num は Eq のサブクラスであるといいます。

サブクラス作りに必要なのはこれだけ。型クラス宣言に型クラス制約を付ければいいのです！ こうしておけば、型クラス宣言やインスタンス宣言でメソッドの実体を書くとき、 a という型は Eq に属すると推論できますから、 a 型の値に対して == が利用できます。

多相型を型クラスのインスタンスに

ところで、 Maybe とかリストといった型は、どうやって型クラスのインスタンスになれるのでしょう？ Maybe が、 TrafficLight のような普通の型と違うところは、 Maybe それ自身は具体型ではないところです。 Maybe は、型引数を 1 つ、例えば Char を取って、 Maybe Char という具体型を返す型コンストラクタです。再び Eq 型クラスを見てみましょう。

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

型宣言を見ると、 a は具体型として使われていることが分かります。なぜなら、関数に表れる型はすべて具体型である必要があるからです。ご存知のとおり、a -> Maybe という型の関数は決して作れません。が、 a -> Maybe a という型なら、あるいは Maybe Int -> Maybe String という型の関数なら作れます。このような理由から、次のようなインスタンス宣言は不可能なのです。

```
instance Eq Maybe where
    ...
```

a には具体型が入りますが、 Maybe は違います。 Maybe は型引数を 1 つ取って具体型を**生み出す**型コンストラクタなのです。

では、Maybe が型引数に取り得るすべての型ごとにインスタンス宣言をいちいち書いていく必要があるのでしょうか？ だとしたら、とても退屈な作業が待っています。 **instance** Eq (Maybe Int) **where** 、 **instance** Eq (Maybe Char) **where** のようにすべての型に対してインスタンスを作っていたら、きりがありません。そのため、型引数を単に変数として残すことが許されています。

```
instance Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

これは、「Maybe something のような格好をしている型はまとめて Eq のインスタンスにしたい」と言っているようなものです。実は型変数名は小文字で始まっていさえすれば何でもよく、 (Maybe something) と書いても文法的には正しいのですが、ここでは型変数の名前は 1 文字という Haskell の流儀に従っておきましょう。

ここでは (Maybe m) が **class** Eq a **where** という決まり文句における a の役割を果たしています。 Maybe は具体型ではありませんが、 Maybe m は具体型です。 Maybe の型引数に m という型変数を与えることで、「任意の m に対して Maybe m という姿をとる型を Eq のインスタンスにしたい」と宣言しているのです。

ただし、これには 1 つだけ問題があります。気づきましたか？ Maybe の中身に == を使いましたよね。でも、 Maybe の中身が Eq として使える保証はどこにもないはずです！ 何が必要か、もうお分かりですね。そう、型 m に対する型クラス制約が必要です！ そこで、インスタンス宣言をこう書き換えるわけです。

```
instance (Eq m) => Eq (Maybe m) where
    Just x == Just y = x == y
    Nothing == Nothing = True
    _ == _ = False
```

このインスタンス宣言なら、Maybe m の形をしている型をすべて Eq に属するようにしたい、ただし m （Maybe の中身）も Eq に属している型に限る、と伝えることができます。実はこれ、Haskell の自動導出がやってくれることと同じです。

ほとんどの場合、型クラス宣言での型クラス制約を使うのは、ある型クラスを別の型クラスのサブクラスにする場合で、インスタンス宣言での型クラス制約を使うのは、型の中身に対する必要条件を記述する場合です。例えば、ここでは Maybe の中身が型クラス Eq に属していることを要求しました。

ある型クラスのメンバ関数の型宣言において、インスタンス型が具体型（例えば a -> a -> Bool の a ）として使われているなら、その型クラスのインスタンスを宣言するにあたって、型コンストラクタに必要な数の引数を与えて、括弧で囲み、具体型を仕立てなければなりません。

これからインスタンスにしようとしている型はクラス宣言における型変数の位置を占める、と考えてください。クラス宣言 **class** Eq a **where** における a は、インスタンスを作ると実際の型で置き換えられます。ですから、頭の中でその型を関数型宣言に代入してみてください。このような型宣言は意味を成しません。

```
(==) :: Maybe -> Maybe -> Bool
```

でも、これなら大丈夫。

```
(==) :: (Eq m) => Maybe m -> Maybe m -> Bool
```

ちょっと考えたら分かります。 == は、どんなインスタンスを作ろうと、常に (==) :: (Eq a) => a -> a -> Bool という型を持つものですから。

それからもう 1 つ。型クラスのインスタンスが何者かを知りたければ、GHCi で :info YourTypeClass と打ってください[†4]。例えば、 :info Num と打てば、型クラス Num が定義している関数、および Num に属する型のリストが表示されます。 :info は型や型コンストラクタにも使えます。例えば :info Maybe とすれば、 Maybe がインスタンスとなっている型クラスの情報がすべて表示されます。例を示します。

[†4] ［訳注］省略形の :i も使えます。

```
ghci> :info Maybe
data Maybe a = Nothing | Just a -- Defined in Data.Maybe
instance (Eq a) => Eq (Maybe a) -- Defined in Data.Maybe
instance Monad Maybe -- Defined in Data.Maybe
instance Functor Maybe -- Defined in Data.Maybe
instance (Ord a) => Ord (Maybe a) -- Defined in Data.Maybe
instance (Read a) => Read (Maybe a) -- Defined in GHC.Read
instance (Show a) => Show (Maybe a) -- Defined in GHC.Show
```

7.9　Yes と No の型クラス

JavaScript をはじめ、いくつかの弱く型付けされた言語では、 `if` 式の中にほとんど何でも書くことができます。例えば、JavaScript ではこんなことができます。

```
if (0) alert("YEAH!") else alert("NO!")
```

こんなことや、

```
if ("") alert ("YEAH!") else alert("NO!")
```

こんなことも。

```
if (false) alert("YEAH!") else alert("NO!")
```

上のコードはすべて `NO!` というアラートを表示します。

一方、JavaScript では空でない文字列はすべて true 値と解釈されるため、以下のコードは `YEAH!` というアラートを出します。

```
if ("WHAT") alert ("YEAH!") else alert("NO!")
```

真理値の意味論が必要なところでは厳密に `Bool` 型を使うのが Haskell の流儀ですが、JavaScript 的な振る舞いを実装してみるのも面白そうですよね！

```
class YesNo a where
    yesno :: a -> Bool
```

いたって単純です。 `YesNo` 型クラスはメソッドを 1 つだけ定義しています。その関数は、「真理値の概念を何らかの形で含むとみなせる型」の値を取り、それが `true` であるか否かを返します。関数の中での `a` の使われ方からして、 `a` は具体型でないとダメです。

では、こいつのインスタンスを定義していきましょう。数に関しては、 `0` でない数は真理値として解釈した場合は真になり、 `0` は偽になる、という前提にしましょう（JavaScript と同じ）。

```
instance YesNo Int where
    yesno 0 = False
    yesno _ = True
```

空リストは no っぽい値で、空でないリストは yes っぽい値です。

```
instance YesNo [a] where
    yesno [] = False
    yesno _ = True
```

型引数 a を導入することで、リストの中身の型について何も想定することなくリストを具体型にするテクニックに注目です[†5]。

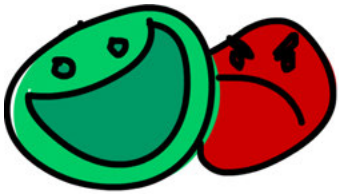
Bool 自身も真偽の概念を含んでいるはずです。どっちが真なのかは明らかですね。

```
instance YesNo Bool where
    yesno = id
```

id って何でしょう？ これは引数を 1 つ取って同じものを返すだけの標準ライブラリ関数です。

Maybe a もインスタンスにしてみましょう。

```
instance YesNo (Maybe a) where
    yesno (Just _) = True
    yesno Nothing = False
```



今回は型クラス制約は不要です。 Maybe の中身に関しては、Just 値ならば true っぽくて、Nothing なら false っぽいと定義することにしたからです。それでも、 Maybe でなく (Maybe a) と書く必要はあります。なにせ Maybe は具体型ではないので、 Maybe a -> Bool という型なら何の問題もありませんが、 Maybe -> Bool という型の関数は存在を許されません。とはいえ、これで Maybe something 型の値は中身の something が何であろうとも YesNo のインスタンスになったわけですから、すごいことです！

そういえば、ずいぶん前に Tree a という二分探索木を表す型を作りましたね。あれも、空の木は false っぽくて、空じゃない木は true っぽいということにしましょう。

```
instance YesNo (Tree a) where
    yesno EmptyTree = False
    yesno _ = True
```

[†5] ［訳注］型名 [a] は [] a とも書けます。ですからこのインスタンス宣言は、後述する Maybe a や Tree a の宣言文に合わせて、 instance YesNo ([] a) where ... と書くこともできるんですよ。

信号機も yes ／ no 値になれるでしょうか？ もちろんです。信号は、赤なら止まれ、青なら進めです（黄色ならどうするかって？ ええと、僕は「黄色は走れ」派ですね。アドレナリン出るし）。

```
instance YesNo TrafficLight where
    yesno Red = False
    yesno _ = True
```

インスタンスが出揃ったのでさっそく遊んでみましょう！

```
ghci> yesno $ length []
False
ghci> yesno "haha"
True
ghci> yesno ""
False
ghci> yesno $ Just 0
True
ghci> yesno True
True
ghci> yesno EmptyTree
False
ghci> yesno []
False
ghci> yesno [0,0,0]
True
ghci> :t yesno
yesno :: (YesNo a) => a -> Bool
```

動いてます！

では、 **if** の真似をして YesNo 値を取る関数を作ってみましょう。

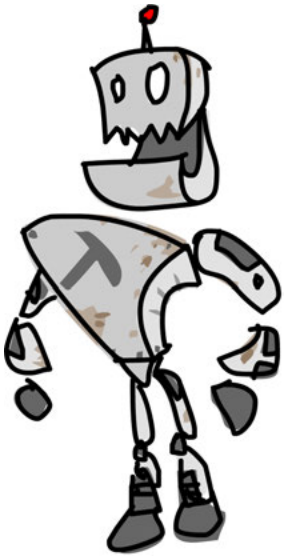
```
yesnoIf :: (YesNo y) => y -> a -> a -> a
yesnoIf yesnoVal yesResult noResult =
    if yesno yesnoVal
        then yesResult
        else noResult
```

これは、YesNo 値を 1 つと、好きな型の値を 2 つ取ります。もし yes ／ no っぽい値がどちらかというと yes なら 2 つの値のうち 1 つ目を返し、そうでなければ 2 つ目を返します[†6]。試してみましょう。

```
ghci> yesnoIf [] "YEAH!" "NO!"
"NO!"
ghci> yesnoIf [2,3,4] "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf True "YEAH!" "NO!"
"YEAH!"
```

[†6] ［訳注］ yesnoIf がこのように手軽に実装できたのは、Haskell が遅延評価を基本とする言語だからこそです。一方、正格評価では、 yesnoIf を呼び出す前に、 yesResult と noResult の両方を評価し終える必要があります！

```
ghci> yesnoIf (Just 500) "YEAH!" "NO!"
"YEAH!"
ghci> yesnoIf Nothing "YEAH!" "NO!"
"NO!"
```



7.10 Functor 型クラス

これまで標準ライブラリの型クラスをいくつも見てきました。順序が付けられる型のクラス Ord や、等値判定のできる型のクラス Eq で遊びました。それから Show という、文字列として表示できる型のインターフェイスを提供している型クラスも見ました。文字列をある型の値に変換したいときには盟友 Read がいました。今度は、 Functor （ファンクター）という型クラスを見ていきたいと思います。 Functor は、**全体を写せる**（map over）ものの型クラスです[†7]。

“map over” と聞いて、Haskell の超頻出イディオムである「リストの map 」とかを思い出しませんか？ あれも何かを写す操作の典型例です。ですから、リストは Functor 型クラスに属しています。

Functor 型クラスを知りたいと思ったら実装を見るのが一番です！ さっそく覗いてみましょう。

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

見てのとおり、 Functor は 1 つの関数 fmap を持っており、デフォルト実装は提供していません。 fmap は面白い型を持っています。これまでに出てきた型クラスでは、インスタンスになった型を表す型変数は、 (==) :: (Eq a) => a -> a -> Bool における a のようにすべて具体型でした。しかし今度の f は、具体型（ Int 、 Bool 、 Maybe String といった、実際の値が持てる型）ではなく、1 つの型引数を取る型コンストラクタです。（軽く復習。 Maybe Int は具体型で、 Maybe は 1 つの型を引数に取る型コンストラクタです。）

どうやら fmap は、「ある型 a から別の型 b への関数」と、「ある型 a に適用されたファンクター値」を取り、「別の型 b のほうに適用されたファンクター値」

[†7] ［訳注］ここで**写す**というのは、「そのままコピーをとる」**写**真のイメージではなくて、「何か変換を施す」**写**像のイメージです。“map over” をどう訳すかはずいぶん悩みましたが、簡単な話なので、なるべく平易な日本語で訳すことにしました。

を返す関数のようです。わけが分からなくても心配無用。例を見ればすぐ分かります。

ところで、fmap の型宣言は何かに似ていると思いませんか？ map 関数の型シグネチャを思い出してください。

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

おや面白い！ map は、「ある型から別の型への関数」と、「ある型のリスト」を取り、「別の型のリスト」を返す関数のようです。どうやらファンクターの正体が見えてきましたぞ！ 実は、map というのはリスト限定で動作する fmap にすぎません。リストに対する Functor インスタンス宣言はこうなります。

```
instance Functor [] where
    fmap = map
```

以上！[†8] **instance** Functor [a] **where** と書いてないのに注目。これは、f は 1 つの型を取る型コンストラクタだからです。以下の宣言を見ても明らかでしょう。

```
fmap :: (a -> b) -> f a -> f b
```

[a] は任意の型を要素とするリストを表す具体型になっているのに対し、[] はまだ具体型ではなく、1 つの型引数を取って [Int]、[String]、[[String]] といった型を生み出す型コンストラクタです。

リストにとっての fmap はただの map であるため、2 つの関数をリストに使った結果は一致します。

```
ghci> fmap (*2) [1..3]
[2,4,6]
ghci> map (*2) [1..3]
[2,4,6]
```

map や fmap を空リストに対して使ったら何が起こるでしょう？ ええ、もちろん、空リストが返ってきます。これらは、[a] 型の空リストを [b] 型の空リストに変える働きをします。

Maybe は Functor だよ、たぶん

ファンクターになれるのは、箱のような働きをする型です。リストというのは、空だったり、ものがいくつか入っていたりする箱だとみなせますね。リストの中身にまた箱が入っていて、その中身も空だったり、さらに別のものが入っていたりするかもしれません。では、ほかにも箱みたいなものってあるで

[†8] ［訳注］ここで Functor のインスタンスと宣言されている [] は、空リストのことではなく、リスト型の型コンストラクタとしての [] です。型名 [a] は [] a とも書けることを思い出してください。

しょうか？ 例えば Maybe a がそうです。そう思ってみると、 Maybe とは、何も入っていなければ Nothing 値になり、何か、例えば "HAHA" が入っていれば Just "HAHA" という値になる箱のように見えませんか？

Maybe はこんなふうにしてファンクターになっています。

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

ここでも **instance** Functor Maybe **where** と書いていることに注目してください。 YesNo のときみたいに、 (Maybe m) を使って **instance** Functor (Maybe m) **where** と書いてはいません。 Functor は具体型ではなく、型コンストラクタを要求しているからです。頭の中で、 f を Maybe に置換してみましょう。 Maybe に限って言えば fmap は (a -> b) -> Maybe a -> Maybe b になることが分かります。これは正しい。一方、 f を (Maybe m) で置換すると、 fmap は (a -> b) -> Maybe m a -> Maybe m b になりますが、これは意味が通りません。 Maybe は型引数を 1 つしか取らないからです。

fmap の実装はいたってシンプルです。もし 2 つ目の引数が空の値を示す Nothing なら、 Nothing を返します。空の箱を関数で写しても空のままというわけです。もし 2 つ目の引数が空でなく、1 つの値が詰まった Just 値だったら、関数を Just の中身に適用します。

```
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") ⇨
          (Just "Something serious.")
Just "Something serious. HEY GUYS IM INSIDE THE JUST"
ghci> fmap (++ " HEY GUYS IM INSIDE THE JUST") Nothing
Nothing
ghci> fmap (*2) (Just 200)
Just 400
ghci> fmap (*2) Nothing
Nothing
```

Tree も Functor の森に

「関数で写せるもの」であるため、 Functor に属する型はほかにもあります。われらが Tree a 型がそうです。木は、複数の値を持たせることも空にすることもできる箱のようなものですし、 Tree 型コンストラクタは引数を 1 つだけ取るからです。 fmap を Tree 専用の関数と思って眺めた場合、その型シグネチャは (a -> b) -> Tree a -> Tree b となるでしょう。

さて、ここは再帰の使いどころですよ。ある関数で空の木を写すと、空の木が生じます。一方、空でない木を写すときは、ルート値はその関数を適用したも

の、それから左右の部分木は元の部分木が写されたものになるはずです。これがそのコードです。

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x left right)
                    = Node (f x) (fmap f left) (fmap f right)
```

試してみましょう。

```
ghci> fmap (*2) EmptyTree
EmptyTree
ghci> fmap (*4) (foldr treeInsert EmptyTree [5,7,3])
Node 20 (Node 12 EmptyTree EmptyTree) (Node 28 EmptyTree EmptyTree)
```

でも、 Tree a 型を二分探索木を実装するのに使っている場合は気をつけて！二分探索木を任意の関数で写した後の木も二分探索木のままであるという保証はどこにもありません。ある木が二分探索木とみなせるためには、各ノードについて、その左部分木の要素はすべてそのノードの要素より小さく、右部分木の要素は大きくないといけません。でも、例えば negate のような関数で二分探索木を写すと、左部分木の要素はたちまちルートの要素より大きくなってしまい、二分探索木だったものはただの二分木になってしまいます。

Either は Functor であるか否か

では、 Either a b はどうでしょう？ これはファンクターにできるでしょうか？ Functor 型クラスは、型引数を 1 つだけ取る型コンストラクタを要求していますが、 Either は引数を 2 つ取ります。うーん……そうだ！ Either に引数を 1 つだけ部分適用して、自由引数を 1 つ残した状態にすればいいんじゃないでしょうか？

標準ライブラリ、具体的に言うと Control.Monad.Instances では、Either a がファンクターとして次のように定義されています。

```
instance Functor (Either a) where
    fmap f (Right x) = Right (f x)
    fmap f (Left x) = Left x
```

ふむふむ、ただの Either ではなく Either a がインスタンスにされていることがよく分かりますね。 Either a は 1 つの型引数を取る型コンストラクタであるのに対し、 Either は 2 つ取るからです。 fmap がもし Either a に特化していたら、その型シグネチャはこうなっていたでしょう。

```
(b -> c) -> Either a b -> Either a c
```

なぜなら、上のコードは以下のコードと等価だからです。

```
(b -> c) -> (Either a) b -> (Either a) c
```

Right 値コンストラクタがきた場合には関数が適用されますが、Left 値の場合には何もしません。なぜでしょう？ Either a b 型の定義に戻れば理由は見て取れます。

```
data Either a b = Left a | Right b
```

もし、左と右の両方を同じ関数で写したかったら、a と b は同じ型である必要があるでしょう。考えてもみてください。仮に文字列を取って文字列を返す関数で Either a b を写せと言われたとして、b は文字列型だけど a は数値型だったら、もう詰んでますよね。それに、fmap が Either a b に作用したときの型を見ても、2 つ目の引数は変化してもいいけど 1 つ目の引数は不変でないとまずいことが分かります。そして、1 つ目の引数は Left 値コンストラクタが使っていることが分かります。

これは箱の比喩ともうまく合致します。Right は中身の入った箱、Left は空の箱のようなもので、どうして空になってしまったのかを表すエラーメッセージが箱の側面に書かれているのです。

Data.Map の関数もファンクター値にできます。Map もまた値が入っている（入ってないかも！）箱とみなせるからです。Map k v の場合、fmap は v -> v' という関数で、Map k v を写して Map k v' という Map を返します。

NOTE ' という文字は、変数の名前に使えるように型変数の名前にも使えますが、文法上の特別な意味はありません。ただ、ある型に似ているが少し異なる型を表す記号として使うことが多いです。

Map k がどのように Functor のインスタンスになるのか自力で突き止めることは読者への練習問題とさせていただきましょう！

これまでの例で見たように、型クラスは、Functor のようなめちゃくちゃかっこいい高次の概念を表現できます。それから、型を部分適用したり、インスタンスを作る経験も積みましたね。第 11 章では、ファンクターが満たすべき法則について見ていきます。

7.11 型を司るもの、種類

型コンストラクタは、他の型を引数に取っていずれは具体型になります。この振る舞いは、関数が値を引数に取って値を生み出すのとよく似ていますね。それに、関数と同じく型コンストラクタも部分適用できます。例えば、Either String は型引数をあと 1 つとって具体型、例えば Either String Int を生み出す型コンストラクタです。

この節では、型が型コンストラクタに適用されるようすを形式的に定義してみます。この節を読み飛ばしても Haskell 魔法の大冒険は続けることができますが、読めば Haskell の型システムがどのように機能しているのかが分かります。それに、今はまだ何も分からなくても大丈夫です。

3 、 "YEAH" 、そして takeWhile といった値（そう、関数も受け取ったり渡したりできる値です。Haskell ではね）は、それぞれ固有の型を持っています。型とは、値について何らかの推論をするために付いている小さなラベルです。そして、型にも小さなラベルが付いているんです。その名は**種類**（kind）。種類は、まあ「型の型」のようなものです。奇妙でややこしい話に聞こえますが、実はとてもかっこいい概念なんです。

種類とはそもそも何者で、何の役に立つのでしょう？ さっそく GHCi の :k コマンドを使って、型の種類を調べてみましょう。

```
ghci> :k Int
Int :: *
```

この * は何でしょう？ これは、具体型を表す記号です。具体型とは型引数を取らない型のことです。値に付けられる型は具体型だけです。 * を声に出して読む必要に迫られた場合、僕なら「スター」、あるいは単に「型」と読むでしょう。

では、 Maybe の種類は何でしょうか。

```
ghci> :k Maybe
Maybe :: * -> *
```

Maybe は 1 つの具体型（例えば Int ）を取って具体型（例えば Maybe Int ）を返す型コンストラクタであることが分かります。ちょうど、 Int -> Int といえば Int を取って Int を返す関数を指すように、 * -> * は 1 つの具体型を取って 1 つの具体型を返す関数を意味するのです。 Maybe に型引数を与えて、どんな種類の型ができるか調べてみましょう。

```
ghci> :k Maybe Int
Maybe Int :: *
```

予想どおり、 Maybe に型引数を与えると具体型になっていました（これこそ、 * -> * の意味するところです）。 :t isUpper と :t isUpper 'A' を呼び出せ

ば、型コンストラクタの種類と関数の型とが対応しているのが分かります（が、まったく同じというわけではありません。型と種類は違うものです）。 isUpper 関数は Char -> Bool 型を持ち、 isUpper 'A' は Bool という型を持ちます。一方、両者の種類はどちらも * です。

:t が値の型を調べるコマンドなのと同様、 :k は型の種類を調べるコマンドです。型は値のラベルであり、種類は型のラベルである、という対応関係があるわけです。大事なことなので、二度言いましたよ。

では、 Either の種類を見てみましょう。

```
ghci> :k Either
Either :: * -> * -> *
```

Either は 2 つの具体型を取って具体型を返す関数である、と言っていますね。これもまた、2 つの引数を取って何かを返す関数の型宣言とどこか似ています。型コンストラクタは（関数と同様）カリー化されているので、ご覧のとおり部分適用できます。

```
ghci> :k Either String
Either String :: * -> *
ghci> :k Either String Int
Either String Int :: *
```

Either を Functor 型クラスのインスタンスにしようとしたときは、部分適用をして Either a の形にする必要がありました。 Functor は型引数を 1 つ取る型を要求しているのに対し、 Either は型引数を 2 つも取るからです。言い換えると、 Functor になれる型は種類が * -> * の型なので、 Either を部分適用して元の * -> * -> * という種類から * -> * という種類に変換する必要があったのです。

改めて Functor の定義を見直すと、型変数 f は 1 つの具体型を取って具体型を生み出す型として使われていることが見て取れます。

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

関数の型宣言で値の型として使われていることから、 f a や f b は具体型でなければならないことが分かります。そこから、 Functor と友達になりたい型は * -> * という種類を持つ必要があることが導けます。

第8章
入出力

この章では、キーボードからの入力を受け取ったり、画面に何かを表示したりする方法を学びます。

でもその前に、入出力（I/O）の基本をカバーしておきましょう。

- I/O アクションとは何か？
- I/O アクションがどうやって入出力を可能にするのか？
- 実際に I/O アクションが行われるのはいつか？

I/O の扱いは、Haskell の関数にできることの制約にかかわる問題なので、その制約をどうやって回避するのかを最初に見ていきます。

8.1 不純なものと純粋なものを分離する

そろそろ Haskell が純粋関数型言語であることにも慣れてきたと思います。人間からコンピュータに与えるのは、一連の実行ステップではなく、あるものが何であるかの定義です。加えて、関数は**副作用**を持つことを許されません。関数は与えられた引数のみを使って何らかの結果を返すことしかできません。関数が同じ引数で 2 回呼び出されたら、必ず同じ結果が返されます。

このことは、はじめのうちはとても大きな制限のように思えるかもしれませんが、実際には素晴らしく優れた性質なのです。命令型言語では、一見すると数を処理するだけに見

える簡単な関数が、処理の片手間にあなたの家に火をつけたり犬を誘拐したりしないことを保証できません。例えば、二分探索木を前の章で作りましたが、木そのものを変更して要素を挿入したわけではありません。その代わりに、木に新しい要素を挿入した**新しい**木を返したのでした。

関数が状態を変更できない、例えばグローバル変数を更新したりできないのは好ましいことです。なぜなら、プログラムについての推論が容易になるからです。しかし 1 つ問題があります。もし関数がこの世界の状態を何も変えられないとしたら、計算したものを僕らにどうやって伝えるのでしょうか？ そのためには出力デバイス（たいていはモニタ）の状態を変更しなければなりません。それによって光子が放たれ、僕らの脳に到達し、僕らの脳の状態を変更するのです。

でも落胆しないで。望みはあります。Haskell は、副作用を持つ関数を扱うための素晴らしく賢いシステムを持っているのです。そのシステムが、プログラムの純粋（pure）な部分と、キーボードや画面とやり取りするようなすべての汚い仕事をする不純（impure）な部分とをきっちりと分離してくれます。この 2 つの部分が隔てられているので、外の世界とやり取りしつつも、依然としてプログラムの純粋な部分を推論したり、純粋だからこそ得られる遅延評価、堅牢性、関数合成などを利用したりできます。この章ではそのようすを見ていきます。

8.2　Hello, World!

これまでは、作った関数は GHCi にロードして試していました。標準ライブラリの関数たちも、同じく GHCi から探検してきました。今ついに初めて本物の Haskell プログラムを書くときがやってきました！ イェーイ！ 最初に作るプログラムはもちろん、お馴染み「Hello, World!」です。鉄板ですね！

初心者の方は、お好みのテキストエディタに次のコードを叩き込んでください。

```
main = putStrLn "hello, world"
```

`main` を定義しただけです。その中で `putStrLn` という関数を、`"hello, world"` という引数で呼び出しています。これを `helloworld.hs` というファイルに保存しましょう。

これから今までやらなかったことをします。プログラムをコンパイルして、実行ファイルを生成するのです！ 端末を開いて helloworld.hs のあるディレクトリに移動し、次のコマンドを打ち込んでください。

```
$ ghc --make helloworld
```

これは、GHC コンパイラを起動してプログラムをコンパイルするコマンドです。実行すると、何やら次のようなレポートが表示されるでしょう。

```
[1 of 1] Compiling Main ( helloworld.hs, helloworld.o )
Linking helloworld ...
```

コンパイルが成功したら、次のように端末にタイプしてプログラムを実行できます。

```
$ ./helloworld
```

NOTE Windows を使っているなら、 ./helloworld の代わりに **helloworld.exe** とタイプしてプログラムを実行してください。

このプログラムは次のようなメッセージを表示します。

```
hello, world
```

ほら、コンパイルした初めてのプログラムが端末に何かを表示しましたよ。でもちょっとこれはありえないくらいつまらないですね！

先ほどのコードを詳しく見ていきましょう。まず、関数 putStrLn の型を見てみます。

```
ghci> :t putStrLn
putStrLn :: String -> IO ()
ghci> :t putStrLn "hello, world"
putStrLn "hello, world" :: IO ()
```

putStrLn の型は次のように読めます。「 putStrLn は文字列を引数に取り、() （空のタプル。unit 型ともいう）を結果とする **I/O アクション**を返す」[†1]

I/O アクションとは、実行されると副作用（入力を読んだり画面やファイルに何かを書き出したり）を含む動作をして結果を返すような何かです。このことを「I/O アクションが結果を**生成する**」といいます。文字列を端末に表示するアクションには実際には意味のある返り値がないので、ダミーの値として () を使います。

NOTE 空のタプルの値は () であり、その型もまた () です。

†1 ［訳注］ putStrLn "hello, world" は、評価されると "hello, world" と表示するのではなく、「 "hello, world" と表示しろ」という命令書（I/O アクション）を返します。

では I/O アクションはいつ実行されるのでしょうか？ さて、ここで main が関係してきます。I/O アクションは、僕らがそれに main という名前をつけてプログラムを起動すると実行されるのです。

8.3 I/O アクションどうしをまとめる

プログラム全体を単一の I/O アクションにしないといけないのは、ちょっと制約に思えます。そこで複数の I/O アクションを糊付けして 1 つにするのに **do** 構文が使えます。次の例を見てください。

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

新しい構文です！ しかも命令型のプログラムにかなり似ています。コンパイルして実行すれば、このコードから想像したとおりに動くでしょう。

do と書いてから、あたかも命令型のプログラムを書くように実行ステップを書き並べています。それら実行ステップのそれぞれが I/O アクションです。I/O アクションを **do** 構文を使ってまとめると、糊付けされた 1 つの I/O アクションにできます。こうして得られるアクションの型は `IO ()` になります。 **do** の中の最後の I/O アクションの型が `IO ()` だからです。そのため、 main の型シグネチャは「`main :: IO 何か`」になります。「**何か**」には具体型が入ります。 main の型宣言は普通、明示しません[†2]。

3 行目の `name <- getLine` は何でしょうか？「入力から 1 行読み込み、それを name という名前の変数に格納する」と読めそうです。本当にそうでしょうか？ では getLine の型を調べてみましょう。

```
ghci> :t getLine
getLine :: IO String
```

getLine は String を生成する I/O アクションだと分かります。ユーザが端末に何か入力するのを待って、それからその何かを文字列として返すということなので、なるほど理にかなっています。

では、 `name <- getLine` とすると何が起こるのでしょう？ このコード片は次のように読めます。「I/O アクション getLine を実行して、それからその結果の値を name に束縛せよ」 getLine は `IO String` という型を持つので、 name の型は String になります。

[†2] ［訳注］ドキュメントのため、トップレベルの定義では型宣言を書くのが一般的です。 main にも型宣言を書くことのほうが多いです。

I/O アクションは、小さな足が付いた箱だと考えることができます。実世界に出て行って、そこで何かを行い（壁に落書きしたりとか）、何か値を持って帰ってくる小さな箱です。箱がデータを取ってきたとき、それを開けて中のデータを手に入れる唯一の方法が <- です。そして、I/O アクションからデータを取り出そうとしても、それができるのは別の I/O アクションの中だけです。以上が純粋なものと不純なものをきちんと分類する Haskell の方法です。I/O アクション getLine は純粋ではありません。2 回実行したときに同じ結果を返す保証はないからです。

name <- getLine を実行すると、 name は通常の文字列になります。なぜなら、 name は箱の中にあるものを表しているからです。例えば、名前（普通の文字列）を引数として受け取り、その名前に基づいた占いの結果（fortune telling）を返すという、実に複雑な関数が書けます。

```
main = do
    putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn $ "Zis is your future: " ++ tellFortune name
```

tellFortune 関数（というか、 name を渡されるあらゆる関数）は、I/O について何も知る必要がありません。これは単なる String -> String の関数なのです！

I/O アクションのどのへんが普通の値と異なるのか見るために、次のコードを考えてみましょう。これは正しいでしょうか？

```
nameTag = "Hello, my name is " ++ getLine
```

「いいえ」と答えた方はクッキーをどうぞ。「はい」と答えた方にはタバスコのカクテルを飲んでもらいます（冗談ですよ！）。 ++ は両辺に同じ型のリストを要求するので、これは動作しません。左の引数は String（つまり [Char] ）の型を持ち、右の引数 getLine は IO String の型を持ちます。文字列と I/O アクションは連結できません。最初に I/O アクションから String 型の値を取り出す必要があります。それにはどこか別の I/O アクションの中で name <- getLine のようにするしかありません。

純粋でないデータを扱いたいなら、純粋でない環境の中で行わなければなりません。不純による汚染はゾンビのように拡散していくので、コード中の I/O の部分を可能な限り小さく抑えるようにするべきなのです。

実行された I/O アクションはどれも結果を生成します。そのため、先ほどの例は次のように書くこともできます。

```
main = do
    foo <- putStrLn "Hello, what's your name?"
    name <- getLine
    putStrLn ("Hey " ++ name ++ ", you rock!")
```

しかし、 foo は () の値しか持たないので、このように書いても意味がありません。最後の putStrLn は束縛を行っていないことに注目してください。 do ブロックの最後のアクションでは、最初の 2 つのアクションとは違って名前を束縛できないのです。なぜそうなのか、正確な話は第 13 章でモナドの世界に降り立ってから見ていくことにしましょう。

最後の行を除いて、 do ブロックのすべての行に束縛を書いてもかまいません。 putStrLn "BLAH" を _ <- putStrLn "BLAH" と書いてもいいということです。でも意味がないので、 putStrLn のような意味のある結果を生成しない I/O アクションに対しては束縛を省略します。

こんなことすると何が起こるでしょう？

```
myLine = getLine
```

入力から行を読んで、それを myLine に束縛すると思いますか？ いいえ、そうはなりません。これは、I/O アクション getLine に対して別の名前 myLine を与えているだけです。I/O アクションから値を取り出すには、他の I/O アクションの中で <- を使って名前に束縛するしかありません。

I/O アクションが実行されるのは、 main という名前を与えられたとき、あるいは do ブロックで作った別の大きな I/O アクションの中にあるときです。いくつかの I/O アクションを糊付けするのにも do ブロックが使えて、その I/O アクションをまた別の do ブロックの中で使うことができて、その I/O アクションもまた別の do ブロックで使うことができます。それらは、最終的に main の中に含まれていれば実行されることになります。

もう 1 つ I/O アクションが実行される場合があります。GHCi に I/O アクションを入力して ENTER を押したときです。

```
ghci> putStrLn "HEEY"
HEEY
```

単純に数や関数呼び出しを GHCi に入力して ENTER を押した場合も、GHCi はその結果の値に show を適用して、その結果を putStrLn を使って端末に表示します。

I/O アクションの中で let を使う

I/O アクションを do 構文を使って糊付けしているときに、let 構文を使って純粋な値を名前に束縛できます。<- が I/O を実行してその結果を名前に束縛するのに対し、let は I/O アクションの中で普通の値に名前を与えたいときに使います。これはリスト内包表記における let 構文に似ています。

<- と let による束縛を両方とも使った I/O アクションを見てみることにしましょう。

```
import Data.Char

main = do
    putStrLn "What's your first name?"
    firstName <- getLine
    putStrLn "What's your last name?"
    lastName <- getLine
    let bigFirstName = map toUpper firstName
        bigLastName = map toUpper lastName
    putStrLn $ "hey " ++ bigFirstName ++ " "
                      ++ bigLastName
                      ++ ", how are you?"
```

do ブロックの中での I/O アクションの並べ方が分かってもらえたでしょうか？ let と I/O アクション、それから let の中の名前はどうでしょう？ Haskell ではインデントが重要なので、これはいい実例です。

map toUpper firstName と書いていますが、これにより "John" のような文字列を、もっとかっこいい "JOHN" のような文字列に変換しています。そうして大文字化した文字列を名前に束縛して、それを端末に表示する文字列として使っています。

いつ <- を使って、いつ let 束縛を使えばいいのか、困惑しているかもしれませんね。<- は、I/O アクションを実行して、その結果に名前を束縛するのに使います。ところが map toUpper firstName は I/O アクションではありません。純粋な Haskell の式です。というわけで、<- は I/O アクションの結果に名前を束縛したいときに使い、let 束縛は純粋な式に名前を束縛するのに使います。let firstName = getLine のように実行しても、I/O アクション getLine を別の名前で呼んだことにしかなりません。それを実行して結果を束縛するには <- を使う必要があるのです。

逆順に表示する

Haskell における入出力の感覚をもっとつかむために、1 行ずつ読み込んでは単語をさかさまにして表示するという動作を繰り返す単純なプログラムを作っ

てみましょう。プログラムに空行を入力したら停止するようにします。これがそのプログラムです。

```
main = do
    line <- getLine
    if null line
        then return ()
        else do
            putStrLn $ reverseWords line
            main

reverseWords :: String -> String
reverseWords = unwords . map reverse . words
```

このプログラムが何をするのか、とりあえず動かしてみましょう。reverse.hs というファイルに保存し、コンパイルして実行してみます。

```
$ ghc --make reverse.hs
[1 of 1] Compiling Main             ( reverse.hs, reverse.o )
Linking reverse ...
$ ./reverse
clean up on aisle number nine
naelc pu no elsia rebmun enin
the goat of error shines a light upon your life
eht taog fo rorre senihs a thgil nopu ruoy efil
it was all a dream
ti saw lla a maerd
```

reverseWords 関数はごく普通の関数です。 "hey there man" のような文字列を受け取り、これに words を適用して ["hey","there","man"] のような単語のリストを生成します。それからそのリストに対して reverse をマップし、["yeh","ereht","nam"] を受け取り、 unwords で 1 つの文字列に戻します。最終的な結果は "yeh ereht nam" になります。

main についてはどうでしょうか？ まず getLine を実行して端末から行を読み、それに line という名前をつけます。それから条件式があります。Haskell ではすべての式が何らかの値を持つので、すべての **if** には対応する **else** が必要なのでした。このコードの **if** では、条件が真（この場合は空行が入力された）なら 1 つの I/O アクションが実行され、真でなければ **else** 以下にある I/O アクションが実行されることになります。

else 以下には正確に 1 つの I/O アクションがなければならないので、 **do** ブロックを使って 2 つの I/O アクションを 1 つに糊付けしています。この部分を次のように書くこともできます。

```
else (do
    putStrLn $ reverseWords line
    main)
```

do ブロックを 1 つの I/O アクションとみなしていることが分かりやすくなりますが、醜いです。

do ブロックの中では、 getLine で入力した行に reverseWords を適用してから端末に表示しています。それに続けて、 main を実行します。これは再帰的に実行される正しいコードです。なぜなら、 main もまた I/O アクションだからです。要するにプログラムの先頭に戻ります。

null line が True なら、 **then** 以下にある return () というコードが実行されます。他の言語でサブルーチンや関数から戻るのに return を使ったことがあるかもしれません。しかし、Haskell の return は他の言語にある return とはまったく異なるものです。

Haskell（特に I/O アクションの中）での return は、純粋な値から I/O アクションを作り出します。I/O アクションを再び箱で例えると、 return は値を受け取り、それを箱の中に入れるものだと考えることができます。作り出された I/O アクションは実際には何も行いません。単に結果を生成するだけです。ゆえに、I/O の文脈では return "haha" は IO String の型を持つでしょう。

純粋な値を何もしない I/O アクションに変換して、何のありがたみがあるのでしょうか？ 先ほどのプログラムでは、空の行を入力した場合に実行するための何らかの I/O アクションが必要なのでした。それが return () と書いて何もしないハリボテの I/O アクションを作る理由です。

他の言語と異なり、Haskell の return には I/O の **do** ブロックの実行を終わらせる働きはありません。例えば、このプログラムは最後の行まで事もなげに実行されます。

```
main = do
    return ()
    return "HAHAHA"
    line <- getLine
    return "BLAH BLAH BLAH"
    return 4
    putStrLn line
```

繰り返しになりますが、どの return も結果を生成する I/O アクションを作り出し、その結果は名前に束縛されていないので捨てられてしまいます。

return を <- を使った名前の束縛と組み合わせて使うことができます。

```
main = do
    a <- return "hell"
    b <- return "yeah!"
    putStrLn $ a ++ " " ++ b
```

見てのとおり、 return が <- の反対側にあります。 return は箱の中に値をしまい込むものなので、 <- はその箱を受け取り（そしてそれを実行し）、中の

値を取り出して名前に束縛します。でも、これは冗長なコードです。なぜなら、do ブロック中では let を使った束縛が使えるからです。

```
main = do
    let a = "hell"
        b = "yeah"
    putStrLn $ a ++ " " ++ b
```

do ブロックで I/O を行うときは、たいてい return を使うことになります。というのも、何もしない I/O アクションを作る必要があったり、do ブロックの最後のアクションで作り出された結果を I/O アクションの結果として返したくない場合があったりするからです。違う結果を I/O アクションの返り値にしたいときは、return を使って望みの結果を生成する I/O アクションを作り、それを do ブロックの最後に配置します。

8.4　いくつかの便利な I/O 関数

Haskell には便利な関数と I/O アクションがたくさん用意されています。いくつかの使い方を見ていくことにしましょう。

putStr

putStr は putStrLn とよく似ています。文字列を引数として受け取り、その文字列を端末に表示する I/O アクションを返します。しかし putStrLn とは異なり、putStr は文字列を表示した後に改行を出力しません。例えば次のコードを見てください。

```
main = do
    putStr "Hey, "
    putStr "I'm "
    putStrLn "Andy!"
```

コンパイルして実行すると次のような出力が得られるでしょう。

```
Hey, I'm Andy!
```

putChar

putChar は文字を受け取り、その文字を端末に表示する I/O アクションを返す関数です。

```
main = do
    putChar 't'
    putChar 'e'
    putChar 'h'
```

putStr は putChar を使って再帰的に定義できます。 putStr の基底部は空の文字列で、この場合は空の文字列を出力します。要するに、 return () を使って何もしない I/O を返します。空でなければ、先頭の文字を putChar で表示して、それから残りを再帰的に表示します。

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

I/O の中で、純粋なコードと同じように再帰が使えることが分かるでしょうか。まずは再帰の基底部を定義して、それから残りのケースを考えます。この場合だと、最初に先頭の文字を出力して、それから残りの文字列を出力します。

print

print は、 Show のインスタンスの型（文字列としてどう表現すればよいか知っている型という意味）の値を受け取り、それに show を適用して「文字列化」して、それからその文字列を端末に出力します。基本的にこれは putStrLn . show と同じものです。はじめに引数に対して show を呼び出し、その結果を putStrLn に与えます。これは値を表示する I/O アクションを返します。

```
main = do
    print True
    print 2
    print "haha"
    print 3.2
    print [3,4,3]
```

これをコンパイルして実行すると次の出力が得られます。

```
True
2
"haha"
3.2
[3,4,3]
```

見てのとおり、 print はとても便利な関数です。I/O アクションが実行されるのは main の中に入っているか GHCi のプロンプトで評価しようとしたときだけ、という話を覚えていますか？ 僕らが GHCi で値（ 3 や [1,2,3] ）をタイプして ENTER を押したとき、その値を端末に表示するのに GHCi が実際に使っているのは print なのです！

```
ghci> 3
3
```

```
ghci> print 3
3
ghci> map (++"!") ["hey","ho","woo"]
["hey!","ho!","woo!"]
ghci> print $ map (++"!") ["hey","ho","woo"]
["hey!","ho!","woo!"]
```

文字列を表示したいとき、普通は putStrLn を使います。ダブルクオートで囲まれてほしくないからです。でも、他の型の値を端末に表示するときには print が一番よく使われます。

when

when 関数は Control.Monad モジュールにある関数です（ **import** Control.Monad するとアクセスできます）。この関数の面白いところは、 **do** ブロックでは制御構文のように見えるのに実際には普通の関数だというところです。

when は Bool と I/O アクションを受け取り、Bool の値が True の場合には渡された I/O と同じものを返します。 False だった場合は何もしない return () を返します。

次のコードは、入力を受け取り、それが SWORDFISH だったときに限ってそのままターミナルに出力する小さなプログラムです。

```
import Control.Monad

main = do
    input <- getLine
    when (input == "SWORDFISH") $ do
        putStrLn input
```

when を使わなければこのようにプログラムを書かなければならないでしょう。

```
main = do
    input <- getLine
    if (input == "SWORDFISH")
        then putStrLn input
        else return ()
```

見てのとおり、 when は条件が満たされたときのみ何らかの I/O アクションを行いたい場合に便利な関数です。

sequence

sequence 関数は、I/O アクションのリストを受け取り、それらを順に実行する I/O アクション（シーケンス）を返します。この I/O アクションが生成する

結果は、実行したすべての I/O アクションの結果からなるリストです。例えばこのようなコードがあったとします。

```
main = do
    a <- getLine
    b <- getLine
    c <- getLine
    print [a,b,c]
```

これを次のように書くことができます。

```
main = do
    rs <- sequence [getLine, getLine, getLine]
    print rs
```

これら 2 つの結果はまったく同じになります。 `sequence [getLine, getLine, getLine]` は `getLine` を 3 回行う I/O アクションを作ります。このアクションを名前に束縛したら、結果はすべての結果のリストになります。なので、この場合の結果はユーザがプロンプトから入力した 3 つのものからなるリストになるでしょう。

`sequence` を使ったよくあるパターンは、リストに対して `print` や `putStrLn` のような関数を `map` するときです。 `map print [1,2,3,4]` は I/O アクションを作りません。代わりに I/O アクションのリストを作ります。意味的には、これは次のように書いたのと同じです。

```
[print 1, print 2, print 3, print 4]
```

I/O のリストを I/O アクションに変換したいなら、それをシーケンスにしないといけません。

```
ghci> sequence $ map print [1,2,3,4,5]
1
2
3
4
5
[(),(),(),(),()]
```

出力の最後にある `[(),(),(),(),()]` は何でしょうか？ GHCi で I/O アクションを評価すると、端末にはその結果が表示されます。ただし結果が `()` のときだけは例外です。この例外があるので、 `putStrLn "hehe"` を評価すると GHCi は `hehe` とだけ表示します。 `putStrLn "hehe"` は `()` を生成するからです。しかし GHCi に `getLine` を入力した場合には I/O アクションの結果が表示されます。 `getLine` の型は `IO String` だからです。

mapM

「リストに対して I/O アクションを返す関数をマップし、それからシーケンスにする」という操作は頻出するので、ユーティリティ関数 mapM と mapM_ が用意されています。 mapM は関数とリストを受け取り、リストに対して関数をマップして、それからそれをシーケンスにします。 mapM_ も同じことをしますが、その後で結果を捨ててしまいます。I/O アクションの結果が必要ないときは mapM_ を使います。 mapM の使用例を次に示します。

```
ghci> mapM print [1,2,3]
1
2
3
[(),(),()]
```

3 つのユニットからなるリストは不要なので、こうしたほうがいいでしょう。

```
ghci> mapM_ print [1,2,3]
1
2
3
```

forever

forever 関数は I/O アクションを受け取り、その I/O アクションを永遠に繰り返す I/O アクションを返します。 Control.Monad で定義されています。次の小さなプログラムは、無限にユーザからの入力を受け取り、それを大文字化して出力し続けます。

```
import Control.Monad
import Data.Char

main = forever $ do
    putStr "Give me some input: "
    l <- getLine
    putStrLn $ map toUpper l
```

forM

forM （ Control.Monad にあります）は、 mapM に似ていますが引数の順序が逆になっています。最初の引数がリストで、2 番目がそのリストにマップする関数です。何の役に立つのでしょう？ ラムダ式と do 記法をうまく組み合わせてこんな書き方ができるのです。

```
import Control.Monad

main = do
    colors <- forM [1,2,3,4] $ \a -> do
        putStrLn $ "Which color do you associate with the number "
                   ++ show a ++ "?"
        color <- getLine
        return color
    putStrLn "The colors that you associate with 1, 2, 3 and 4 are: "
    mapM putStrLn colors
```

これを実行してみると次のような結果が得られます。

```
Which color do you associate with the number 1?
white
Which color do you associate with the number 2?
blue
Which color do you associate with the number 3?
red
Which color do you associate with the number 4?
orange
The colors that you associate with 1, 2, 3 and 4 are:
white
blue
red
orange
```

`\a -> do ...` というラムダ式は、数を受け取り I/O アクションを返す関数です。 `do` の最後で `return color` を呼んでいることに注目してください。この `do` ブロックはユーザの選択した色を表す文字列を返すと定義しているのでこうしています。ただ、実際にこう書く必要はありません。 `getLine` はすでに選択した色を返していて、それが `do` ブロックの最後に位置しているからです。`color <- getLine` を実行した後で `return color` を行うのは、 `getLine` の結果をほどいてから再度梱包しているだけなので、単に `getLine` を呼び出すのと同じなのです。

`forM` 関数を 2 つの引数で呼び出すと、I/O アクションが生成され、その結果は `colors` に束縛されます。 `colors` は文字列を含む普通のリストです。最後にすべての色を `mapM putStrLn colors` を呼んで表示しています。

`forM` はこんなふうに考えればいいでしょう。「このリストの各要素に対応する I/O アクションを作る。それぞれの I/O アクションの動作は、アクションを作るのに使った要素に応じたものにできる。最終的には、これらのアクションが実行された結果が何かに束縛される。(結果が必要なければ丸ごと捨ててしまうこともできる)」

`forM` を使わなくても同じことはできます。しかし `forM` を使うとコードが読みやすくなります。 `do` 記法を使った何らかのアクションをマップしてシーケン

スにしたい場合、普通は forM を使います[†3]。

8.5　I/O アクションおさらい

I/O の基本をざっとおさらいしておきましょう。I/O アクションというのは値であり、Haskell の他の値とよく似ています。関数の引数として渡すことができて、関数の結果として I/O アクションを返すことができます。

I/O アクションが特別なのは、 main 関数の中に入っていると（あるいは GHCi のプロンプトで評価されると）、それが実行されるところです。画面に何かを表示したり、名曲「ヤケティ・サックス」をスピーカーから再生したりするのです。またどの I/O アクションも、実世界から取得してきたものを伝える結果を生成できます。

[†3] ［訳注］ mapM と forM は関数とリスト、どちらの引数を長く書きたいかによって使い分けるのが良いでしょう。

第9章
もっと入力、もっと出力

Haskell の入出力の背後に潜むコンセプトを理解したところで、ようやく面白いことを始められます。この章の話題は、ファイルとのやり取り、乱数生成、コマンドライン引数の扱い方などです。チャンネルはそのまま！

9.1 ファイルとストリーム

I/O アクションの仕組みの知識を手に、Haskell でファイルの読み書きをしていくことにします。とその前に、どうすれば Haskell でストリームデータを簡単に扱えるのかを見ていきましょう。**ストリーム**とは、時間をかけてプログラムに出たり入ったりする連続したデータ片のことです。例えば、キーボードからプログラムに文字を入力するとき、その文字たちをストリームと考えることができます。

入力のリダイレクト

多くの対話型プログラムはキーボードからユーザの入力を受け取ります。しかし、テキストファイルの内容をプログラムに入力として与えることができれば、さらに便利です。そのために入力のリダイレクトを使います。

Haskell プログラムで入力のリダイレクトができると便利なので、それをどのように行うのかを見てみましょう。はじめに、次のような俳句を含む小さなファ

イル haiku.txt を作ります。

```
I'm a lil' teapot
What's with that airplane food, huh?
It's so small, tasteless
```

あちゃー、これはひどい。誰か俳句を教えてくれる人がいたら連絡ください。

気を取り直して、行を読み込んで大文字化して表示を繰り返す小さなプログラムを書きましょう。

```
import Control.Monad
import Data.Char

main = forever $ do
    l <- getLine
    putStrLn $ map toUpper l
```

このプログラムを capslocker.hs として保存し、コンパイルします。

キーボードから行を入力する代わりに、haiku.txt をプログラムにリダイレクトして入力しましょう。入力をリダイレクトするには、プログラム名の後に < という文字と、それに続けて入力したいファイル名を指定します。やってみましょう。

```
$ ghc --make capslocker
[1 of 1] Compiling Main             ( capslocker.hs, capslocker.o )
Linking capslocker ...
$ ./capslocker < haiku.txt
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
capslocker <stdin>: hGetLine: end of file
```

capslocker に端末から俳句をタイプして、最後に EOF（end of file。「CTRL-D」[†1]として入力できる）を入力したときの動作とほとんど同じです。capslocker を実行して次のように言っているような感じです。「待って、キーボードから読まないで。代わりにこのファイルの中身を受け取って！」

入力ストリームから文字列を得る

入力ストリームを扱いやすい普通の文字列にしてくれる I/O アクション getContents を見ていきましょう。 getContents は、標準入力から EOF 文字に達するまですべての文字を読み込みます。その型は getContents :: IO String です。 getContents がイケてるのは遅延 I/O を行うところです。どういうことかというと、 foo <- getContents としても getContents は一度に全

[†1] ［訳注］Windows のコマンドプロンプトでは「CTRL-Z」。

部の入力をメモリに読み込んで foo に束縛しないのです。そう、 getContents は遅延するのです！「やーやー、後でちゃんと端末から入力を受け取るよ。それが本当に必要になったときに！」

capslocker.hs の例では、 forever を使って 1 行ずつ入力を読み、それを大文字にして表示していました。 getContents は I/O の細かい面倒を見てくれます。その一環として、必要なときに必要な分だけ入力を読み込んでくれるのです。何か入力を受け取り、それを変換して出力するプログラムなら、 getContents を使って簡潔に書けます。

```
import Data.Char

main = do
    contents <- getContents
    putStr $ map toUpper contents
```

I/O アクション getContents を実行して、その結果の文字列に contents という名前をつけています。それからその文字列を toUpper で写して、その結果を端末に表示します。忘れないでください。文字列は基本的にはリストなので処理は遅延されますし、 getContents は遅延 I/O なので、大文字化した文字列を表示する前にすべての中身（コンテンツ）を一度にメモリに読み込んだりはしません。必要なときに入力から行を読み込むので、読んだらすぐに大文字化した文字列を表示します。

試してみましょう。

```
$ ./capslocker < haiku.txt
I'M A LIL' TEAPOT
WHAT'S WITH THAT AIRPLANE FOOD, HUH?
IT'S SO SMALL, TASTELESS
```

正しく動作しています。じゃあ、 capslocker をリダイレクトなしで実行し、端末から入力をタイプしたらどうなるでしょうか？（プログラムを終了させるには「CTRL-D」[†2]をタイプします。）

```
$ ./capslocker
hey ho
HEY HO
lets go
LETS GO
```

とても素晴らしい！ ご覧のとおり大文字化された文字列が 1 行ごとに出力されています。

[†2] ［訳注］Windows のコマンドプロンプトでは「CTRL-Z」。

getContents の結果が contents に束縛されるとき、それは本当の文字列ではなく、最終的には文字列に評価されるプロミス（promise）としてメモリ上に置かれます。 contents に toUpper をマップするとき、それもまた入力の結果に関数をマップするというプロミスになります。最終的に putStr が呼ばれると、これがさっきのプロミスに対して「やあ、大文字化された行が必要なんだ！」と言います。そのプロミスはまだ入力の行を何も持っていないので、 contents に対し「端末からの入力の状況はどうなってる？」と問い合わせます。それでようやく getContents は実際に端末から入力して、何か入力をくれといってきたコードに生成したものを渡すのです。受け取ったコードは渡されたものに toUpper をマップし、その結果を putStr に渡して、画面に行が出力されます。さらに続けて putStr は「ヘイ、次の行をくれ！ カモン！」と言います。これが入力がなくなるまで、つまり EOF 文字が入力されるまで繰り返されます。

では、入力を受け取り、10 文字より短い行だけを出力するプログラムを作ってみましょう。

```
main = do
    contents <- getContents
    putStr (shortLinesOnly contents)

shortLinesOnly :: String -> String
shortLinesOnly = unlines . filter (\line -> length line < 10) . lines
```

プログラム中の I/O の部分を可能な限り少なくしました。このプログラムは、何かの入力に基づいて何か出力をするものだと考えられるので、入力のコンテンツを読み込み、それに対して関数を走らせ、その結果を出力する、と実装できます。

shortLinesOnly 関数は "short\nlooooooong\nbort" のような文字列を受け取ります。この例では文字列は 3 行で、そのうち 2 行は短く真ん中の 1 行は長めです。この文字列は、 lines 関数を適用すると、文字列のリスト ["short", "looooooong", "bort"] に変換されます。これに 10 文字未満の文字列をフィルタする関数が適用され、 ["short", "bort"] になります。最後に、それに unlines を適用し、改行文字で区切られた 1 つの文字列、 "short\nbort\n" になります。

実行してみましょう。次のテキストを shortlines.txt として保存します。

```
i'm short
so am i
i am a looooooooong line!!!
yeah i'm long so what hahahaha!!!!!!
short line
looooooooooooooooooooooooooong
short
```

プログラムを shortlinesonly.hs に保存して、コンパイルして実行します。

```
$ ghc --make shortlinesonly
[1 of 1] Compiling Main             ( shortlinesonly.hs, shortlinesonly.o )
Linking shortlinesonly ...
```

次のように shortlines.txt のコンテンツをリダイレクトして試します。

```
$ ./shortlinesonly < shortlines.txt
i'm short
so am i
short
```

短い行だけ端末に表示されているのが分かるでしょう。

入力を変換する

「入力を文字列として受け取り、それを関数で変換し、結果を出力する」というパターンはとてもよく出てくるので、これを簡単に済ませるための interact という関数があります。 interact は String -> String 型の関数を受け取り、入力にその関数を適用して、返ってきた結果を出力する、という I/O アクションを返します。先ほどのプログラムを interact を使って書き換えてみましょう。

```
main = interact shortLinesOnly

shortLinesOnly :: String -> String
shortLinesOnly = unlines . filter (\line -> length line < 10) . lines
```

このプログラムは、入力をファイルからリダイレクトしても、キーボードから一行一行入力しても実行できます。出力はどちらの場合も同じですが、キーボードから入力する場合は capslocker プログラムのときのように入力ごとに出力が表示されます。

入力を行ごとに読み込み、それが回文かどうかを出力するプログラムを作ってみましょう。行を読むのに getLine を使い、それが回文かどうかを出力し、それから main を呼び出して再帰してもいいですが、 interact を使うともっとシンプルに書けます。 interact によって、入力を望みの出力に変換するにはどうすればいいかだけを考えればよくなります。今回の場合だと、各行を "palindrome" か "not a palindrome" に置き換えます。

```
respondPalindromes :: String -> String
respondPalindromes =
  unlines .
  map (\xs -> if isPal xs then "palindrome" else "not a palindrome") .
  lines

isPal :: String -> Bool
isPal xs = xs == reverse xs
```

とても素直に書けています。このプログラムは、最初に次のような文字列を、

```
"elephant\nABCBA\nwhatever"
```

このようなリストに変換します。

```
["elephant", "ABCBA", "whatever"]
```

それから、リスト全体をラムダ式でマップして結果を得ます。

```
["not a palindrome", "palindrome", "not a palindrome"]
```

続いて、 unlines を使って改行文字で区切られた単一の文字列に連結します。あとは main の I/O アクションを作るだけです。

```
main = interact respondPalindromes
```

試してみましょう。

```
$ ./palindromes
hehe
not a palindrome
ABCBA
palindrome
cookie
not a palindrome
```

大きい 1 つの文字列を別の文字列に変換するプログラムを作ったのに、このプログラムは 1 行ごとに処理するプログラムを書いたかのように動作します。これは Haskell が遅延評価だからです。結果の文字列の最初の行を表示したくても、入力の最初の行がまだないので表示できません。プログラムは、入力の最初の行を取得したらすぐに出力の最初の行を表示します。プログラムから抜け出すには EOF 文字を入力します。

このプログラムにも入力をファイルからリダイレクトできます。次の内容を words.txt に保存します。

```
dogaroo
radar
rotor
madam
```

これをリダイレクトして実行します。

```
$ ./palindrome < words.txt
not a palindrome
palindrome
palindrome
palindrome
```

標準入力から単語を入力して実行したときと同じ入力が得られました。ただしプログラムはファイルから入力を受け取ったので、入力は端末には現れていません。

遅延 I/O がどのように動作し、それをどう活用するかを見てきました。プログラムを書くときは、ある入力に対してどんな出力が考えられるかという視点から考えて、その変換をする関数を書くだけです。今出力したいものは入力によって決まるので、遅延 I/O では本当に必要になるまで入力を一切消費しません。

9.2 ファイルの読み書き

ここまでは、端末への表示と端末からの読み込みに I/O を使いました。でもファイルの読み書きはどうするのでしょう？ そうですね、ある意味ではファイルの読み書きもしていたと言えます。

端末から読むということは（何か特別な）ファイルから読んでいると考えることもできます。端末への書き出しも、同様に、ある特別なファイルへの書き出しだと考えられます。これらの 2 つの特別なファイルを **stdin** および **stdout** と呼びます。それぞれ標準入力（standard input）と標準出力（standard output）を意味します。ファイルへの入出力は、標準入力からの読み込みと標準出力への書き出しにとても似ています。

まず、マザーグースの一節が書かれた `baabaa.txt` というファイルを開き、それを端末に表示するだけの簡単なプログラムを書くことから始めましょう。これが `baabaa.txt` です。

```
Baa, baa, black sheep,
Have you any wool?
Yes, sir, yes, sir,
Three bags full;
```

そしてこれがプログラムです。

```
import System.IO

main = do
    handle <- openFile "baabaa.txt" ReadMode
    contents <- hGetContents handle
    putStr contents
    hClose handle
```

コンパイルして実行すれば、期待した結果が得られます。

```
$ ./baabaa
Baa, baa, black sheep,
Have you any wool?
Yes, sir, yes, sir,
Three bags full;
```

1 行ずつ見ていきましょう。最初の行は、羊さんに呼びかけています。2 行目で歌い手は、羊に毛を刈る用意はできたか聞いています。3 行目では羊が答えていて、4 行目では 3 つの袋がいっぱいになるほどあると言っています。

プログラムのほうも 1 行ずつ見ていきましょう。プログラム全体は、複数の I/O アクションを `do` ブロックでまとめたものになっています。`do` ブロックの最初の行は、初登場の関数 `openFile` です。これは次のような型シグネチャを持っています。

```
openFile :: FilePath -> IOMode -> IO Handle
```

`openFile` は、ファイルパスと `IOMode` を受け取り、そのファイルを開いて、そのファイルに関連付けられたハンドルを返す I/O アクションを返します。`FilePath` は単なる `String` の型シノニムです。

```
type FilePath = String
```

`IOMode` は次のように定義された型です。

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

この型は、曜日を表す 7 つの値を取り得る型のように、開いたファイルに対して何をしたいのかを列挙した型です。この型は `IOMode` であって `IO Mode` ではないことに気をつけてください。`IO Mode` だと、何か `Mode` 型の結果を生成する I/O アクションを意味するでしょう。`IOMode` は単なる列挙型です。

最終的に `openFile` は、指定されたファイルを指定されたモードで開く I/O アクションを返します。そのアクションの結果を何かに束縛すれば、そのファイルに対する `Handle` が得られます。そのハンドルが読み込むファイルを示しています。

次の行では `hGetContents` という関数を使っています。これは、コンテンツをどの

ファイルから読み出すべきか知っている Handle を受け取り、そのファイルに含まれる内容を結果として返す IO String を返します。この関数は getContents にとてもよく似ています。唯一の違いは、 getContents が自動的に標準入力（つまり端末）から入力するのに対し、 hGetContents は渡されたハンドルから入力するという点です。それ以外の挙動はすべて同じです。

hGetContents は、 getContents のようにファイルの内容を一度にメモリに読み込むことはせず、必要になったときに必要な分だけコンテンツを読みます。これは、ファイル全体のコンテンツが contents として扱えるにもかかわらず実際にはメモリに読み込まれていないという、本当に素晴らしい機能です。だから、たとえとてつもなく大きなファイルを読み込んだとしても、 hGetContents はメモリを食いつぶすことがありません。

ファイルのハンドルと実際のコンテンツの違いにも注意してください。ハンドルはファイルの現在の位置を指し示すポインタにすぎません。コンテンツはファイルに実際に書かれているものです。ファイルシステム全体をとても大きな本だとすると、ハンドルは今読んでいる（もしくは書いている）ところを指し示すしおりのようなものです。

putStr contents でファイルのコンテンツを標準出力に表示し、それからハンドルを受け取って、そのハンドルを閉じる I/O アクションを返す hClose を実行します。 openFile で開いたファイルは自分で閉じる必要があるのです！ハンドルが閉じられていないファイルを開こうとすると、プログラムは強制終了するでしょう。

withFile 関数を使う

ファイルの内容を扱うもう 1 つの方法は、次のようなシグネチャを持つ withFile 関数を用いるものです。

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
```

この関数は、ファイルのパスと IOMode 、それに「ハンドルを受け取って I/O アクションを返す関数」を受け取り、「そのファイルを開いてから何かして閉じる」という I/O アクションを返します。さらに withFile は、ファイルの操作中に何かおかしなことが起こった場合にもファイルのハンドルを確実に閉じてくれます。ややこしく見えるかもしれませんが、実際はとてもシンプルな関数です。特にラムダ式と一緒に使うと便利です。

先ほどの例を withFile を使って書き換えたのが次のコードです。

```
import System.IO

main = do
    withFile "baabaa.txt" ReadMode $ \handle -> do
        contents <- hGetContents handle
        putStr contents
```

\handle -> ...（からブロックの最後まで）はハンドルを受け取り I/O アクションを返す関数です。こんなふうに withFile にはよくラムダ式で関数を渡します。実行したい I/O アクションを渡してファイルを閉じるだけではなく、どのファイルを操作するのか、そのハンドルを I/O アクションに教えてやらなければならないので、このようにして（ハンドルを受け取って）I/O アクションを返す関数を渡す必要があるのです。 withFile はファイルを開いてそのハンドルを受け取った関数に渡します。 withFile は、返ってきた I/O アクションと同じ動作をし、なおかつ何か失敗した場合でもファイルハンドルを確実に閉じてくれるような I/O アクションを作ります。

ブラケットの時間

error が呼ばれたり（空リストに対して head が呼ばれたときなど）、あるいは入出力の際にとてもまずいことが起こると、通常はプログラムが強制終了させられ、何らかのエラーメッセージが表示されます。そのような状況を、**例外**が投げられたといいます。 withFile 関数は、このような忌むべき例外が投げられたときでもファイルのハンドルを閉じてくれるのです。

この例のような、「何らかのリソース（例えばファイルのハンドル）を獲得し、それに対して何かを行う、ただしリソースが確実に解放される（例えばファイルのハンドルを閉じる）ことを保証する」というシナリオは、わりとよく登場します。そのために、Control.Exception モジュールに bracket という関数が用意されています。この関数は次のような型シグネチャを持っています。

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

最初の引数はリソース（ファイルハンドルのような）の獲得を行う I/O アクションです。2 番目の引数はリソースを解放する関数です。この関数は例外が投げられた場合でも呼ばれます。3 番目の引数はリソースを受け取り、それを使っ

て何かを行う関数です。ファイルからの読み込みやファイルへの書き出しといったメインの操作は、この3番目の引数にあたります。

リソースを獲得し、それを使って何か行い、そして確実に解放することがbracketのすべてですから、withFileを実装するのもとても簡単です。

```
withFile :: FilePath -> IOMode -> (Handle -> IO a) -> IO a
withFile name mode f = bracket (openFile name mode)
    (\handle -> hClose handle)
    (\handle -> f handle)
```

bracketに渡す最初の引数によってファイルが開き、その結果はファイルのハンドルです。2つ目の引数はハンドルを受け取り、それを閉じます。bracketは、たとえ例外が発生しようとも、確実にハンドルを閉じる処理を実行します。最後の3つ目の引数は、ハンドルを受け取り、それにfを適用します。このfは、ファイルハンドルを受け取り、そのハンドルに対してファイルからの読み書きといった操作を行う関数です。

ハンドルを握れ！

hGetContentsが指定したファイルに対して動作するgetContentsであるように、hGetLine、hPutStr、hPutStrLn、hGetCharのような関数にもhが付かないバージョンがあって、ハンドルを取らない代わりに標準入出力に対して動作します。例えば、putStrLnは文字列を受け取り、それを端末に表示してから改行文字を出力するというI/Oアクションを返します。hPutStrLnはハンドルと文字列を受け取り、ハンドルに対応したファイルに文字列を書き込み、さらに改行文字を書き込むというI/Oアクションを返します。同様に、hGetLineはハンドルを受け取り、そのファイルから1行入力するI/Oアクションを返します。

ファイルを読み込み、そのコンテンツを文字列として扱うというのは、とても一般的な操作です。そのため、これを手軽にするためのシャレた関数が3つ用意されています。readFile、writeFile、appendFileです。

readFile関数は、readFile :: FilePath -> IO Stringという型シグネチャを持つ関数です（FilePathはStringの小洒落た別名でしたね）。readFileはファイルのパスを受け取り、そのファイルを読み込み（もちろん遅延します）、その内容を表す文字列を返すI/Oアクションを返します。普通ならopenFileを呼んでからhGetContentsをそのハンドルで呼び出さなければいけないので、それよりお手軽です。前の例をreadFile関数を使って書くとこうなります。

```
import System.IO

main = do
    contents <- readFile "baabaa.txt"
    putStr contents
```

ファイルを示すハンドルを獲得しないので、それを手動で閉じることはできません。 readFile を使う場合、ハンドルを閉じるのは Haskell が自動で行います。

writeFile は writeFile :: FilePath -> String -> IO () 型の関数です。これはファイルのパスと、そのファイルに書き込みたい文字列を受け取り、その書き込みを行う I/O アクションを返します。指定されたファイルがすでに存在している場合、ファイルは上書きされます。次のコードは baabaa.txt から、それを大文字化したバージョンの baabaacaps.txt を生成するプログラムです。

```
import System.IO
import Data.Char

main = do
    contents <- readFile "baabaa.txt"
    writeFile "baabaacaps.txt" (map toUpper contents)
```

appendFile 関数は writeFile と同じ型シグネチャを持ち、同じような動作をしますが、appendFile はすでにファイルが存在していた場合に上書きするのではなくファイルの末尾に追記するという点が異なります。

9.3　ToDo リスト

ToDo リスト（やらなければならないことリスト）をテキストファイルに追加するプログラムを作るのに appendFile 関数を使ってみましょう。todo.txt という名前のファイルにタスクが 1 行ごとに書かれているものとします。プログラムは、標準入力から 1 行ずつ読み込んで、それを ToDo リストに追加します。

```
import System.IO

main = do
    todoItem <- getLine
    appendFile "todo.txt" (todoItem ++ "\n")
```

各行の最後に "\n" を追加していることに注意してください。 getLine は改行文字を除いた文字列を返してくるからです。

これを appendtodo.hs に保存して、コンパイルして、実行します。いくつか ToDo リストのアイテムを与えてみましょう。

```
$ ./appendtodo
Iron the dishes
$ ./appendtodo
Dust the dog
$ ./appendtodo
Take salad out of the oven
$ cat todo.txt
Iron the dishes
Dust the dog
Take salad out of the oven
```

NOTE cat は Unix 系のシステムでファイルの中身を端末に表示するプログラムです。Windows システムでは、お好みのエディタで todo.txt の内容を確認しましょう。

アイテムの削除

todo.txt の ToDo リストに新しいアイテムを追加するプログラムを作りました。次はアイテムを削除するプログラムを作りましょう。System.Directory の新しい関数をいくつかと、System.IO の新しい関数を 1 つ使います。解説の前に、まずはコードを見てみましょう。

```
import System.IO
import System.Directory
import Data.List

main = do
    contents <- readFile "todo.txt"
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line)
                                [0..] todoTasks
    putStrLn "These are your TO-DO items:"
    mapM_ putStrLn numberedTasks
    putStrLn "Which one do you want to delete?"
    numberString <- getLine
    let number = read numberString
        newTodoItems = unlines $ delete (todoTasks !! number) todoTasks
    (tempName, tempHandle) <- openTempFile "." "temp"
    hPutStr tempHandle newTodoItems
    hClose tempHandle
    removeFile "todo.txt"
    renameFile tempName "todo.txt"
```

最初に todo.txt を読み込み、その内容を contents に束縛します。それから、その文字列を行ごとに分割し、文字列のリストにします。ここで todoTasks は次のようになっているでしょう。

```
["Iron the dishes", "Dust the dog", "Take salad out of the oven"]
```

このリストと、0から始まるリストを、数（例えば3）と文字列(例えば"hey")を受け取り、新しい文字列（"3 - hey"）を返す関数でzipします。さて、numberedTasksは次のようになっているはずです。

```
["0 - Iron the dishes"
,"1 - Dust the dog"
,"2 - Take salad out of the oven"
]
```

それからmapM_ putStrLn numberedTasksで行ごとに表示して、どれを削除したいのかをユーザに問い合わせます。今、1（Dust the dog）を削除したいとしましょう。端末に1を打ち込みます。そうすると、numberStringは"1"になります。文字列じゃなくて数が欲しいので、この文字列に対してreadを適用します。すると1が得られ、これが**let**でnumberに束縛されます。

Data.Listのdeleteと!!関数を覚えていますか？ !!は、添字に対応するリストの要素を返します。deleteは、リストから指定した要素のうち最初に出てくるものを削除した新しいリストを返します。(todoTasks !! number)は"Dust the dog"になります。そしてtodoTasksの中から"Dust the dog"の最初の出現を削除し、unlinesを使って改行文字で区切られた1つの文字列に連結します。それがnewTodoItemsになります。

それから、System.IOにある初登場の関数openTempFileを使います。名前が示すように、この関数は一時ディレクトリ（temp directory）のパスとファイル名のテンプレートを受け取り、一時ファイルを開きます。ここでは一時ディレクトリとして"."を使っています。"."はどのOSでもカレントディレクトリを表します。一時ファイル名のテンプレートとしては"temp"を使っています。一時ファイルの名前は、"temp"の後ろにランダムな文字をいくつか付けたものになります。openTempFileが返すI/Oアクションは、一時ファイルを開き、そのファイル名とハンドルのペアを返します。todo2.txtのような名前の普通のファイルを開いても同じことはできますが、openTempFileを使うようにしてください。openTempFileを使えば、何かが入っているファイルにうっかり上書きしないことが保証されるからです。

一時ファイルを開いたので、それにnewTodoItemsを書き込みます。古いファイルは変更されず、削除すべきものを削除した新しいリストが一時ファイルに格納されます。

その後で、一時ファイルと元のファイルの両方を閉じます。それから、removeFileで元のファイルを削除します。これには削除したいファイルへのパスを渡します。古いtodo.txtを削除したら、renameFileを使って一時ファイルの名前をtodo.txtに変更します。removeFileとrenameFile（両方

とも System.Directory の関数）は、引数としてハンドルではなくファイルのパスを受け取ります。

このプログラムを deletetodo.hs に保存し、コンパイルして実行してみましょう。

```
$ ./deletetodo
These are your TO-DO items:
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
Which one do you want to delete?
1
```

どのアイテムが残っているか見てみましょう。

```
$ cat todo.txt
Iron the dishes
Take salad out of the oven
```

おー、素晴らしい！ アイテムをもっと消しましょう。

```
$ ./deletetodo
These are your TO-DO items:
0 - Iron the dishes
1 - Take salad out of the oven
Which one do you want to delete?
0
```

ファイルを調べて、残っているアイテムが 1 つになっているか確認します。

```
$ cat todo.txt
Take salad out of the oven
```

というわけで、すべてうまく動いています。しかし、このプログラムには 1 箇所だけ少し気になる点があります。一時ファイルを開いた後でプログラムが異常終了したら、一時ファイルが残ってしまいます。これを修正しましょう。

クリーンアップ

問題が起こった場合でも一時ファイルが確実に削除されるようにするために、Control.Exception にある bracketOnError 関数を使うことにします。この関数は bracket にとてもよく似ていますが、 bracket では処理が終わると常に獲得したリソースを開放するのに対し、 bracketOnError は何らかの例外が発生したときのみリソースを開放します。

```
import System.IO
import System.Directory
import Data.List
import Control.Exception

main = do
    contents <- readFile "todo.txt"
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line)
                                [0..] todoTasks
    putStrLn "These are your TO-DO items:"
    mapM_ putStrLn numberedTasks
    putStrLn "Which one do you want to delete?"
    numberString <- getLine
    let number = read numberString
        newTodoItems = unlines $ delete (todoTasks !! number) todoTasks
    bracketOnError (openTempFile "." "temp")
        (\(tempName, tempHandle) -> do
            hClose tempHandle
            removeFile tempName)

        (\(tempName, tempHandle) -> do
            hPutStr tempHandle newTodoItems
            hClose tempHandle
            removeFile "todo.txt"
            renameFile tempName "todo.txt")
```

普通に openTempFile を使うのではなく、 bracketOnError と一緒に使いました。引数として、エラーが発生したときにすべきこと、つまり、一時ハンドルを閉じてから一時ファイルを削除するというラムダ式を渡しています。最後に、一時ファイルを使って何をしたいかを記述しています。この部分は前と同じです。新しいアイテムリストを書き出し、一時ファイルのハンドルを閉じて、今のファイルを削除し、一時ファイルの名前を変更しています。

9.4　コマンドライン引数

端末で動作するスクリプトやアプリケーションを作りたいなら、コマンドライン引数を扱うことは不可欠です。Haskell の標準ライブラリには、コマンドライン引数を扱うための便利な方法が用意されています。やりましたね。

前の節で、ToDo リストにアイテムを追加するプログラムと、アイテ

ムを削除するプログラムをそれぞれ作りました。これらのプログラムの問題は、ToDo ファイルの名前がハードコードされていることです。ファイル名を `todo.txt` に決め打ちし、複数の ToDo リストを管理するというニーズがないものとしていました。

1 つの解決方法は、ToDo リストのファイル名を毎回ユーザに尋ねるというものです。このアプローチは、どのアイテムを削除するかを決めるときに使いました。それでも動作はしますが、これはユーザに「プログラムを走らせてから、プログラムが何か聞いてくるのを待って、プログラムに何か入力する」という要求をするわけで、あまり理想的な解決策ではありません。ちなみに、これは**対話的**（interactive）プログラムと呼ばれています。

対話的なコマンドラインプログラムには難点があります。スクリプトから呼び出してプログラムの実行を自動化したいとき、どうすればいいでしょう？ プログラムと対話するスクリプトを書くのは、単純にプログラム（1 つでも複数でも）を呼び出すスクリプトを書くよりも難しいものです。そのため、プログラムの実行に必要な情報は、プログラムを実行している間ではなく、プログラムを起動するときにユーザに問い合わせるようにしたいのです。プログラムに何をさせたいかをユーザからプログラムに伝える方法として、コマンドライン引数よりマシな方法はありません。

`System.Environment` モジュールは、コマンドライン引数を取得するのに便利な 2 つの素敵な I/O アクション、 `getArgs` と `getProgName` を提供しています。 `getArgs` は `getArgs :: IO [String]` という型を持ちます。これは、プログラムに与えられた引数を取得して、それを文字列のリストとして返す I/O アクションです。 `getProgName` は `getProgName :: IO String` という型を持ちます。これはプログラム名を返す I/O アクションです。これらがどのように動作するか、次の小さなプログラムで見てみましょう。

```
import System.Environment
import Data.List

main = do
   args <- getArgs
   progName <- getProgName
   putStrLn "The arguments are:"
   mapM putStrLn args
   putStrLn "The program name is:"
   putStrLn progName
```

最初にコマンドライン引数を `args` に束縛して、それからプログラム名を `progName` に束縛します。次に、 `putStrLn` を使ってプログラムの引数をすべて

表示し、それからプログラム自身の名前を表示します。このコードを arg-test としてコンパイルし、実行してみましょう。

```
$ ./arg-test first second w00t "multi word arg"
The arguments are:
first
second
w00t
multi word arg
The program name is:
arg-test
```

9.5　ToDo リストをもっと楽しむ

前の例では、タスクを追加するプログラムと削除するプログラムを完全に別々のプログラムとして作成しました。ここでは、両方のプログラムを 1 つにまとめて、追加するか削除するかをプログラムに渡すコマンドライン引数で選択できるようにしましょう。さらに、todo.txt ではなく別のファイルも操作できるようにしましょう。

このプログラムを todo という名前にすることにして、次の 3 つの異なる操作を行えるようにします。

- タスクの閲覧
- タスクの追加
- タスクの削除

タスクを todo.txt に追加するには、端末に次のように入力します。

```
$ ./todo add todo.txt "Find the magic sword of power"
```

タスクを閲覧するには view コマンドを入力します。

```
$ ./todo view todo.txt
```

タスクの削除には番号を使います。

```
$ ./todo remove todo.txt 2
```

マルチタスクタスクリスト

まず、コマンドを "add" や "view" のような文字列として受け取り、引数のリストを受け取って望みの動作を行う I/O アクションを返す関数を作ります。

```
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: String -> [String] -> IO ()
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove
```

main を次のように定義します。

```
main = do
    (command:argList) <- getArgs
    dispatch command argList
```

最初に、コマンドライン引数を取得してそれらを (command:argList) に束縛します。これは、最初の引数を command に束縛して、残りの引数を argList に束縛するという意味です。 main ブロックの次の行で dispatch 関数にコマンド（ command ）を渡し、これは add 、 view 、 remove のいずれかを返します。それから、その関数に argList を渡します。

プログラムを次のように呼び出すとしましょう。

```
$ ./todo add todo.txt "Find the magic sword of power"
```

command が "add" に、argList が ["todo.txt", "Find the magic sword of power"] になります。 dispatch 関数の 2 つ目のパターンマッチが成功して、これは add 関数を返します。最後に、それに対して argList を適用し、これは ToDo リストにアイテムを追加する I/O アクションになります。

add と view と remove 関数を実装していきましょう。 add から始めます。

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
```

これでプログラムを次のように呼び出せるようになりました。

```
./todo add todo.txt "Find the magic sword of power"
```

main ブロックの最初のパターンマッチで、"add" は command に束縛され、それから ["todo.txt", "Find the magic sword of power"] が dispatch 関数の返すものに渡されます。今のところ、正しくない入力に対する処理はまったく考慮しないことにして、引数を単に 2 要素のリストにパターンマッチしています。これは、タスクと改行文字をファイルの末尾に追記する I/O アクションを返します。

次にリスト閲覧機能を実装しましょう。ファイルのアイテムを見たいときは ./todo view todo.txt と実行します。最初のパターンマッチによって、

command は "view" になり、 argList は ["todo.txt"] になります。次のコードがこの関数の実装です。

```
view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line)
                        [0..] todoTasks
    putStr $ unlines numberedTasks
```

ToDo リストからアイテムを削除するだけの deletetodo プログラムを作ったとき、ToDo リストを閲覧する機能も付けていました。そのため、このコードは前のプログラムのその部分にとても似通っています。

最後に remove を実装します。これもタスクを削除するだけのプログラムととてもよく似ているので、コードがどのように動作するか分からなければ 187 ページの「アイテムの削除」を見返してください。大きな違いは、ファイル名 todo.txt をハードコードする代わりに引数として与えているところと、タスク番号を標準入力ではなくこれも引数から取得しているところです。

```
remove :: [String] -> IO ()
remove [fileName, numberString] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line)
                                [0..] todoTasks
    putStrLn "These are your TO-DO items:"
    mapM_ putStrLn numberedTasks
    let number = read numberString
        newTodoItems = unlines $ delete (todoTasks !! number) todoTasks
    bracketOnError (openTempFile "." "temp")
        (\(tempName, tempHandle) -> do
            hClose tempHandle
            removeFile tempName)

        (\(tempName, tempHandle) -> do
            hPutStr tempHandle newTodoItems
            hClose tempHandle
            removeFile "todo.txt"
            renameFile tempName "todo.txt")
```

fileName に基づいてファイルを開き、一時ファイルを作成して削除したい行を削除し、それを一時ファイルに書き込み、元のファイルを削除し、一時ファイルの名前を fileName に変更します。

素晴らしきこのプログラムの全貌をここに記しておきます。

```
import System.Environment
import System.Directory
import System.IO
import Data.List

dispatch :: String -> [String] -> IO ()
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove

main = do
    (command:argList) <- getArgs
    dispatch command argList

add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")

view :: [String] -> IO ()
view [fileName] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line)
                        [0..] todoTasks
    putStr $ unlines numberedTasks

remove :: [String] -> IO ()
remove [fileName, numberString] = do
    contents <- readFile fileName
    let todoTasks = lines contents
        numberedTasks = zipWith (\n line -> show n ++ " - " ++ line)
                                    [0..] todoTasks
    putStrLn "These are your TO-DO items:"
    mapM_ putStrLn numberedTasks
    let number = read numberString
        newTodoItems = unlines $ delete (todoTasks !! number) todoTasks
    bracketOnError (openTempFile "." "temp")
        (\(tempName, tempHandle) -> do
            hClose tempHandle
            removeFile tempName)
        (\(tempName, tempHandle) -> do
            hPutStr tempHandle newTodoItems
            hClose tempHandle
            removeFile "todo.txt"
            renameFile tempName "todo.txt")
```

ここでの解法をまとめると、コマンドから「リストの形でコマンドライン引数を受け取って I/O アクションを返す関数」への橋渡しをする dispatch を作りました。関数 dispatch は、 command が何かに基づいて適切な関数を返します。その関数をコマンドライン引数の残りと一緒に呼び出し、適切な操作を行う I/O アクションを取得して、それを実行します。このように、高階関数を使うこ

とで、まず dispatch 関数から適切な関数を受け取り、次にその関数にコマンドライン引数を渡して I/O アクションを得る、という設計が可能になったのです。

さっそくこのアプリを試してみましょう！

```
$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven

$ ./todo add todo.txt "Pick up children from dry cleaners"

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Take salad out of the oven
3 - Pick up children from dry cleaners

$ ./todo remove todo.txt 2

$ ./todo view todo.txt
0 - Iron the dishes
1 - Dust the dog
2 - Pick up children from dry cleaners :
```

dispatch 関数を使っていることで、簡単に機能を追加できるという利点もあります。 dispatch にパターンを追加して対応する関数を実装するだけで万事 OK ！ 練習として、ファイルとタスクの番号を受け取り、そのタスクを ToDo リストの先頭に持ってくる bump 関数を実装してみましょう。

不正な入力に対応する

不正な入力に対して Haskell からの分かりづらいエラーメッセージをただ表示するのではなく、もう少しちゃんと失敗するようにプログラムを拡張できます。まずは、すべてを拾うパターンを dispatch 関数の最後に追加して、そのようなコマンドが存在しなかった旨を表示する関数を返すようにします。

```
dispatch :: String -> [String] -> IO ()
dispatch "add" = add
dispatch "view" = view
dispatch "remove" = remove
dispatch command = doesntExist command

doesntExist :: String -> [String] -> IO ()
doesntExist command _ =
    putStrLn $ "The " ++ command ++ " command doesn't exist"
```

add 、 view 、 remove のそれぞれの関数に対してもすべて拾うパターンを追加して、与えられたコマンドに対して引数の数が違うということをユーザに伝え

ることができます。

```
add :: [String] -> IO ()
add [fileName, todoItem] = appendFile fileName (todoItem ++ "\n")
add _ = putStrLn "The add command takes exactly two arguments"
```

もし add にちょうど 2 要素のリスト以外が与えられた場合は、最初のパターンマッチは失敗し、2 番目のパターンマッチに拾われます。そして、何を間違えたのかをユーザに分かりやすく表示します。 view と remove にも同様の処理を追加できます。

まだすべての不正な入力をカバーしきれていないことに注意してください。例えば、プログラムを次のように実行するとします。

```
$ ./todo
```

この場合、プログラムはクラッシュします。なぜなら、 do ブロックで (command:argList) というパターンを使っていますが、これは引数が 1 つもない場合を考慮していないからです！ また、ファイルを開く前にそれが存在しているかの検査もしていません。これらの事前検査を追加するのは難しくありませんが、いくぶん退屈なので、このプログラムを完全に不正な入力に対処するようにするのは読者の皆さんへの練習問題としておきます。

9.6 ランダム性

プログラミングをしていると、ランダムなデータが欲しくなることがよくあります (まあ実際には**疑似**乱数であって、本当のランダム性は片手にチーズ、もう片方に風船を持って一輪車に乗ったサルからしか得られないってことはみんな知ってます)。例えば、サイコロを投げる必要のあるゲームや、プログラムのためのテストデータを生成する必要がある場合などです。この節では、Haskell でランダムなデータを生成する方法と、なぜ十分にランダムな値を生成するのに外部からの入力が必要なのかを説明していきます。

たいていのプログラミング言語は乱数を返す関数を持っています。関数を呼び出すたびに違う乱数が返ります。Haskell はどうでしょう？ ええと、Haskell

が純粋関数型言語だということを思い出してください。これはすなわち参照透明性を持つということでした。そして**参照透明性**は、関数が同じ引数で2回呼ばれたなら必ず同じ結果を生成しなければならないことを意味します。これは本当に素晴らしい性質です。なぜなら、そのおかげでプログラムが本当に必要になるまで値の評価を遅らせることができるようになるからです。ところが、これが乱数を得るのを少々厄介にしている原因でもあります。

次のような関数があると考えます。

```
randomNumber :: Int
randomNumber = 4
```

あまり役に立たない乱数関数です。常に 4 を返すからです（でも 4 は完全にランダムですよ。だって僕がサイコロで決めたんだから）。

他の言語では、どうやって一見ランダムに見える数を生成しているのでしょう？ それらはまず初期データを受け取ります。例えば現在時刻のようなものです。それに基づいてランダムに見える数を生成します。Haskell でも、何か初期データ、あるいはランダム性を受け取り、それから乱数を生成する関数を作れます。ランダム性は I/O を使って外から持ってきます。

System.Random モジュールを見てみましょう。このモジュールは乱数生成に必要なすべての関数を持っています。それでは、ここからエクスポートされている関数 random を見ていきましょう。型シグネチャは次のようになっています。

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

うわぁ！ 型宣言に新しい型クラスがありますよ！ RandomGen 型クラスはランダム性の源として扱える型を表し、 Random 型クラスはランダムな値になることのできる型を表します。例えば、真理値は True か False からランダムに選ぶことにより生成できます。同様に数も生成できます。関数はランダムな値にできるでしょうか？ できるとは思えません！ random の型宣言を日本語に翻訳してみると、次のような感じになるでしょう。「乱数ジェネレータ（ランダム性の源）を受け取り、ランダムな値と新しい乱数ジェネレータを返す」。なぜランダムな値と一緒に新しいジェネレータも返すのでしょうか？ すぐに分かりますよ。

random 関数を使うためには、何らかの乱数ジェネレータを手に入れる必要があります。 System.Random モジュールには StdGen という良い感じの型があります。これは型クラス RandomGen のインスタンスになっています。 StdGen を手動で作ったり、ある種の乱数源をもとにシステムに生成してもらったりできます。

手動で乱数ジェネレータを作るには mkStdGen 関数を使います。これは mkStdGen :: Int -> StdGen という型を持ちます。整数を引数に取り、その値

をもとに乱数ジェネレータを返します。OK、それじゃあ random と mkStdGen を連携させて、（ほぼ）ランダムな数を作ってみましょう。

```
ghci> random (mkStdGen 100)
<interactive>:1:0:
    Ambiguous type variable 'a' in the constraint:
      'Random a' arising from a use of 'random' at <interactive>:1:0-20
    Probable fix: add a type signature that fixes these type variable(s)
```

なんだこれは？ ああ、そうか。 random 関数は Random 型クラスの任意の型を返す可能性があるんでした。だから、どの型が欲しいのか Haskell に教えてやる必要があります。ランダムな値と乱数ジェネレータのペアが返ってくることも忘れずに。

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

ランダムっぽい値が得られました！ タプルの 1 番目の要素がその数で、2 番目の要素は新しい乱数ジェネレータの数値としての表現です。同じ乱数ジェネレータに対して random を再度呼び出すと何が起こるでしょうか？

```
ghci> random (mkStdGen 100) :: (Int, StdGen)
(-1352021624,651872571 1655838864)
```

もちろん、先ほどと同じ結果が得られます。では、異なる乱数ジェネレータを引数として渡してみましょう。

```
ghci> random (mkStdGen 949494) :: (Int, StdGen)
(539963926,466647808 1655838864)
```

素晴らしい、違う数です！ 別の型のランダムな値は型注釈を使って得ることができます。

```
ghci> random (mkStdGen 949488) :: (Float, StdGen)
(0.8938442,1597344447 1655838864)
ghci> random (mkStdGen 949488) :: (Bool, StdGen)
(False,1485632275 40692)
ghci> random (mkStdGen 949488) :: (Integer, StdGen)
(1691547873,1597344447 1655838864)
```

コイントス

3 回のコイントスをシミュレートする関数を書きましょう。もし random がランダムな値と一緒に新しい乱数ジェネレータを返さなければ、それぞれのコイントスの結果を生成するために、この関数に 3 つの乱数ジェネレータを渡す必要があるでしょう。しかし、1 つのジェネレータで Int 型（たくさんの異なる値を

取り得る）のランダムな値を生成できるのだから、3 回のコイントスの結果（8 通りしかない）だって生成できてしかるべきです。ランダムな値と一緒に新しいジェネレータを返す random がここで役に立ちます。

コインを単純に Bool として表すことにします。 True が裏で、 False が表です。

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

random にジェネレータを渡して、コイントスの結果と、新しいジェネレータを生成します。それから、新しいジェネレータで再度 random を呼び出し、2 つ目のコイントスの結果を得ます。3 つ目のコインも同様にします。3 回とも同じジェネレータで呼び出したら、すべてのコイントスの結果は同じになり、(False, False, False) か (True, True, True) のどちらかの結果しか得られないでしょう。

```
ghci> threeCoins (mkStdGen 21)
(True,True,True)
ghci> threeCoins (mkStdGen 22)
(True,False,True)
ghci> threeCoins (mkStdGen 943)
(True,False,True)
ghci> threeCoins (mkStdGen 944)
(True,True,True)
```

random gen :: (Bool, StdGen) として呼び出す必要がなかったことに注意してください。すでに threeCoins 関数の型シグネチャに真理値が欲しい！ と書いてあるので、 random gen でも真理値型の乱数が欲しいんだな、と Haskell は推論してくれます。

ランダムな関数をもっと

もっとたくさんのコインを投げたいときはどうすればいいでしょうか？ そのために randoms という関数があります。これは、ジェネレータを受け取って、そのジェネレータに基づく無限長のランダムな値のリストを返す関数です。

```
ghci> take 5 $ randoms (mkStdGen 11) :: [Int]
[-1807975507,545074951,-1015194702,-1622477312,-502893664]
ghci> take 5 $ randoms (mkStdGen 11) :: [Bool]
[True,True,True,True,False]
ghci> take 5 $ randoms (mkStdGen 11) :: [Float]
[7.904789e-2,0.62691015,0.26363158,0.12223756,0.38291094]
```

なぜ randoms はリストと一緒に新しいジェネレータを返さないのでしょうか？ randoms 関数は次のようにとても簡単に実装できます。

```
randoms' :: (RandomGen g, Random a) => g -> [a]
randoms' gen = let (value, newGen) = random gen in value:randoms' newGen
```

これは再帰的な定義です。ランダムな値と新しいジェネレータを受け取って、その値を head に、新しいジェネレータで作ったリストを残りの要素としてリストを作ります。この関数には無限の長さのリストを生成できてほしいので、新しい乱数ジェネレータを返してもらうことはできません。

有限のリストと新しいジェネレータを生成する関数なら作れます。

```
finiteRandoms :: (RandomGen g, Random a, Num n) => n -> g -> ([a], g)
finiteRandoms 0 gen = ([], gen)
finiteRandoms n gen =
    let (value, newGen) = random gen
        (restOfList, finalGen) = finiteRandoms (n-1) newGen
    in  (value:restOfList, finalGen)
```

これもまた再帰的定義になっています。0 個の数を生成したいとしたら、単純に空のリストと与えられたジェネレータをそのまま返します。それ以外の個数のランダムな値を返す場合は、まず乱数を 1 つとジェネレータを生成します。この数が head の要素になります。それから新しいジェネレータを使って n - 1 個の整数を生成し、リストの残りの部分とします。これらをくっつけたものと最終的なジェネレータとをペアにして結果として返します。

ある範囲の乱数を生成したい場合はどうすればいいでしょうか？ ある数より小さい整数の乱数全部、というのでは、欲しい値に対して大きすぎたり小さすぎたりします。サイコロを投げたい場合は？ これには randomR を使います。型はこうなっています。

```
randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)
```

random に似ていますが、1 つ目の引数として上限と下限のペアを受け取り、その範囲内の値を生成します。

```
ghci> randomR (1,6) (mkStdGen 359353)
(6,1494289578 40692)
ghci> randomR (1,6) (mkStdGen 35935335)
(3,1250031057 40692)
```

randomRs 関数も同じく用意されていて、指定された範囲の乱数を無限に生成します。試してみましょう。

```
ghci> take 10 $ randomRs ('a','z') (mkStdGen 3) :: [Char]
"ndkxbvmomg"
```

秘密のパスワードっぽいですね？

ランダム性と I/O

乱数の話が I/O の章で扱われているのに違和感があるかもしれません。今のところ乱数の話では何も I/O を扱ってません。乱数ジェネレータに手作業で適当な整数を与えていました。これでは現実の問題を扱うのには不十分です。プログラムが返すのは毎回同じ乱数になってしまいます。この動作は望ましくありません。これを解決するために、 System.Random は getStdGen という IO StdGen 型の I/O アクションを提供しています。このアクションは、何らかの初期データを使ってシステムのグローバル乱数ジェネレータを初期化します。getStdGen はそのグローバル乱数ジェネレータを返します。

グローバル乱数ジェネレータを使ってランダムな文字列を生成するシンプルなプログラムの例です。

```
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
```

試してみましょう。

```
$ ./random_string
pybphhzzhuepknbykxhe
$ ./random_string
eiqgcxykivpudlsvvjpg
$ ./random_string
nzdceoconysdgcyqjruo
$ ./random_string
bakzhnnuzrkgvesqplrx
```

でも注意してください。 getStdGen を 2 回実行しても、システムは同じグローバル乱数ジェネレータを 2 回返すだけです。

```
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen2 <- getStdGen
    putStr $ take 20 (randomRs ('a','z') gen2)
```

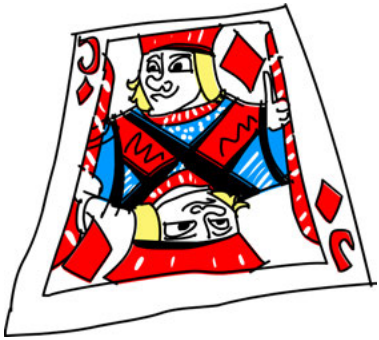
同じ文字列が 2 回表示されるはずです！

2 つの異なる文字列を得るベストな方法は、現在の乱数ジェネレータを 2 つのジェネレータに分割する、 newStdGen アクションです。このアクションは、グローバル乱数ジェネレータを分割した片方で置き換え、もう片方を結果として返します。

```
import System.Random

main = do
    gen <- getStdGen
    putStrLn $ take 20 (randomRs ('a','z') gen)
    gen' <- newStdGen
    putStr $ take 20 (randomRs ('a','z') gen')
```

newStdGen を束縛すると、新しい乱数ジェネレータが得られるだけでなく、グローバルジェネレータも更新されます。これは、getStdGen を再度実行して何かに束縛すると gen に異なるジェネレータが得られることを意味します。



次のコードは、ユーザにプログラムが考えた数を当てさせる小さなプログラムです。

```
import System.Random
import Control.Monad(when)

main = do
    gen <- getStdGen
    askForNumber gen

askForNumber :: StdGen -> IO ()
askForNumber gen = do
    let (randNumber, newGen) = randomR (1,10) gen :: (Int, StdGen)
    putStrLn "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        askForNumber newGen
```

乱数ジェネレータを受け取って、数値を端末から入力してその数値が正しいかどうかを表示する I/O アクションを返す askForNumber 関数を作ります。

askForNumber では、まず受け取った乱数ジェネレータを使って乱数と新しい乱数ジェネレータを生成しています。それぞれを randNumber および newGen と呼びます。生成された数を例えば 7 としましょうか。次に、プログラムが考えた数は何だと思うかユーザに質問します。 getLine を実行して、その結果を numberString に束縛します。ユーザが 7 を入力したなら、 numberString は "7" になります。ユーザの入力が空文字ではないか、 when を使って調べます。それから numberString を read に渡し、数に変換します。 number は 7 になります。

NOTE　read がパーズできない文字列（"haha" のような）をユーザが入力した場合、プログラムはクラッシュして読みにくいメッセージを表示するでしょう。そういう入力に対してプログラムがクラッシュしないようにしたいなら、文字列のパーズに失敗したときに空のリストを返す reads を使いましょう。成功すると望みの値と食べ残しの文字列のペアからなる単一要素のリストを返します。ぜひ！

入力された数がランダムに生成した数と一致するか調べて、ユーザに適切なメッセージを表示します。それから askForNumber を再帰的に新しい乱数ジェネレータで実行します。これは新しい乱数ジェネレータに基づいて実行される点を除いて今実行したものと同じになります。

main は、システムから乱数ジェネレータを取得し、それで askForNumber を呼び出して、1 回目のアクションを得るだけです。

プログラムの実行例です。

```
$ ./guess_the_number
Which number in the range from 1 to 10 am I thinking of?
4
Sorry, it was 3
Which number in the range from 1 to 10 am I thinking of?
10
You are correct!
Which number in the range from 1 to 10 am I thinking of?
2
Sorry, it was 4
Which number in the range from 1 to 10 am I thinking of?
5
Sorry, it was 10
Which number in the range from 1 to 10 am I thinking of?
```

次のコードは同じプログラムを実装する別の方法です。

```
import System.Random
import Control.Monad(when)

main = do
    gen <- getStdGen
    let (randNumber, _) = randomR (1,10) gen :: (Int, StdGen)
    putStrLn "Which number in the range from 1 to 10 am I thinking of? "
    numberString <- getLine
    when (not $ null numberString) $ do
        let number = read numberString
        if randNumber == number
            then putStrLn "You are correct!"
            else putStrLn $ "Sorry, it was " ++ show randNumber
        newStdGen
        main
```

前のバージョンと同じように見えますが、ジェネレータを受け取って新しいジェネレータで再帰的に呼び出す関数を作る代わりに、すべてを main でやって

しまいます。ユーザの推測が正しいかどうかを伝えた後、グローバルジェネレータを更新し、 `main` を再度呼び出します。双方のアプローチはどちらも正しいですが、最初のほうが好ましいです。 `main` が行うことが少なく、関数の再利用も簡単だからです。

9.7 bytestring

リストは確かに便利です。ここまでリストをあらゆる場面で利用してきました。リストに対して動作する関数はたくさんあるし、Haskell の遅延評価のおかげで、リストに対するフィルタやマップとして他の言語における `for` や `while` ループが書けます。本当に必要とされたときにのみ評価されるので、無限リスト（無限リストの無限リストさえも！）のようなものを扱うのも造作ありません。このため、リストをストリームとしても扱うことができます。標準入力から読み込むときも、ファイルから読み込むときも、ファイルを開き、それを文字列として読むだけです。実際のファイルアクセスは必要になってから行われます。

しかし、ファイルを文字列として処理するのことには 1 つ難点があります。実行速度が遅くなりがちなのです。リストは本当に怠け者です。 `[1,2,3,4]` のようなリストは `1:2:3:4:[]` の構文糖衣でしたね。例えばリストを出力すると、最初の要素が無理やり評価されますが、このとき残りのリスト（ `2:3:4:[]` ）はまだプロミスのままです。この未評価の部分を**サンク**（thunk）と呼びます。

サンクというのは、要するに遅延された計算のことです。Haskell の遅延評価は、前もってすべてを計算するのではなく、サンクを使ってそれを必要なときのみ計算することで実現されています。したがって、リストはプロミスだと考えられます。それも、必要になると初めて次の要素と後続のプロミスを渡してくれてくれるようなプロミスです。単なる数のリストをサンクの列として処理するのは、どう考えたって効率の面でベストとは言えないでしょう。

大きなファイルを読んだり、操作しようとしたときに、多くの場合このオーバーヘッドが問題になるのです。これが Haskell に **bytestring** が存在する理由です。bytestring はリストに似たデータ構造で、要素は 1 バイト（あるいは 8 ビット）のサイズ固定です。bytestring では遅延評価を扱う方法も異なります。

正格 bytestring と遅延 bytestring

bytestring には「正格」なものと「遅延」なものがあります。正格 bytestring は `Data.ByteString` で提供されていて、遅延性が完全に排除されています。サンクは一切ありません。正格 bytestring は配列上のバイト列として表現されます。無限の正格 bytestring のようなものは作れません。正格 bytestring の最初のバイトを評価するなら、全体を評価しなければなりません。

遅延 bytestring は `Data.ByteString.Lazy` で提供されます。これは遅延評価されますが、リストほどは遅延されません。リストには、その要素数とちょうど同じくらいの数のサンクがあり、これが場合によってリストが遅くなる原因です。遅延 bytestring では別の方法が採用されています。遅延 bytestring は、64K バイトのチャンク（chunk。thunk と混同しないようにね！）という塊に格納されるのです。そして、遅延 bytestring を評価したら（出力するなど）、最初の 64K バイトが評価されます。残りのチャンクはプロミスです。遅延 bytestring は、64K バイトの正格 bytestring からなるリストだとも考えられます。ファイルを遅延 bytestring で処理するときにはチャンク単位で読み込まれます。これが巧妙なのは、メモリ使用量の上昇を抑えつつ、しかも 64K バイトは CPU の L2 キャッシュにきちんとフィットするサイズになっているところです。

`Data.ByteString.Lazy` のドキュメントを見ると、`Data.List` の関数と同じ名前の関数がたくさん見つかるでしょう。これらは `[a]` の代わりに `ByteString` を、`a` の代わりに `Word8` を受け取ります。それらの関数はリストに対して動作する関数に似ています。名前が同じなので、スクリプトでは修飾付きインポートし、しかる後に GHCi にロードして bytestring を楽しむことにしましょう。

```
import qualified Data.ByteString.Lazy as B
import qualified Data.ByteString as S
```

こうすれば、`B` が遅延 bytestring の型と関数、`S` が正格 bytestring の型と関数になります。ここではもっぱら遅延版を使うことにします。

`pack :: [Word8] -> ByteString` という型シグネチャを持つ `pack` という関数があります。これは `Word8` のリストを受け取り、`ByteString` を返します。遅延するリストを受け取り、それよりは怠け者でない 64K バイト間隔で遅延する bytestring を作る関数だと考えればよいでしょう。

`Word8` 型は `Int` に似ていますが、これは 8 ビット符号なし整数を表します。つまり、0 から 255 という、より小さい範囲の数です。`Int` と同様に `Num` 型クラスのインスタンスです。例えば、`5` はご存知のように多相的であらゆる数値的な型として振る舞えますが、`Word8` としても振る舞えるということです。

数から bytestring に `pack` する方法を示します。

```
ghci> B.pack [99,97,110]
Chunk "can" Empty
ghci> B.pack [98..120]
Chunk "bcdefghijklmnopqrstuvwx" Empty
```

bytestring に pack した値が少なかったので、すべて 1 つのチャンク(Chunk)に収まりました。 Empty はリストにおける [] のようなもので、空の列を表します。

ご覧のとおり、数が Word8 であることを明示する必要はありません。数がその型になることは、型システムによって強いられるからです。もし 336 のような大きな数を Word8 として使おうとすると、オーバーフローして 80 になります。

bytestring を 1 バイトずつ調べる必要がある場合は unpack します。 unpack 関数は pack の逆関数です。bytestring を受け取り、バイトのリストを返します。例を示します。

```
ghci> let by = B.pack [98,111,114,116]
ghci> by
Chunk "bort" Empty
ghci> B.unpack by
[98,111,114,116]
```

正格 bytestring と遅延 bytestring を相互に変換することもできます。 toChunks 関数は、遅延 bytestring を受け取り、それを正格 bytestring のリストに変換します。 fromChunks 関数は、正格 bytestring のリストを受け取り、それを遅延 bytestring に変換します。

```
ghci> B.fromChunks [S.pack [40,41,42], S.pack [43,44,45], ⇨
                    S.pack [46,47,48]]
Chunk "()*" (Chunk "+,-" (Chunk "./0" Empty))
```

小さな正格 bytestring がたくさんあり、それらを連結せずにメモリ上で 1 つの大きな正格 bytestring として効率的に処理したい場合には、このような変換をするとよいでしょう。

bytestring 版の : は cons です。これはバイト値と bytestring を受け取り、そのバイト値を先頭にくっつけます。

```
ghci> B.cons 85 $ B.pack [80,81,82,84]
Chunk "U" (Chunk "PQRT" Empty)
```

bytestring モジュールには、Data.List などで提供されている関数によく似た関数がかなりあります。 head 、 tail 、 init 、 null 、 length 、 map 、 reverse 、 foldl 、 foldr 、 concat 、 takeWhile 、 filter 、などなど。bytestring パッケージのドキュメント http://hackage.haskell.org/package/bytestring/ にすべての関数が載っています。

bytestring モジュールには、 System.IO モジュールが提供する関数と同様の動作をする関数もあります。これらは Strings の代わりに ByteString を受け取ります。例えば、 System.IO の readFile 関数は次の型を持っています。

```
readFile :: FilePath -> IO String
```

bytestring モジュールの readFile 関数は次の型を持ちます。

```
readFile :: FilePath -> IO ByteString
```

NOTE 正格 bytestring を使ってファイルを読もうとすると、そのファイルの内容すべてがメモリ上に一度に読まれます！ 遅延 bytestring なら、こぢんまりとしたチャンクごとに読まれます。

bytestring を使ったファイルのコピー

コマンドライン引数から 2 つのファイル名を受け取って、1 つ目のファイルを 2 つ目のファイルにコピーするプログラムを作りましょう。 System.Directory には copyFile という関数がすでにありますが、とにかく独自のファイルコピー関数を実装しましょう。コードを次に示します。

```
import System.Environment
import System.Directory
import System.IO
import Control.Exception
import qualified Data.ByteString.Lazy as B

main = do
    (fileName1:fileName2:_) <- getArgs
    copy fileName1 fileName2

copy source dest = do
    contents <- B.readFile source
    bracketOnError
        (openTempFile "." "temp")
        (\(tempName, tempHandle) -> do
            hClose tempHandle
            removeFile tempName)
        (\(tempName, tempHandle) -> do
            B.hPutStr tempHandle contents
            hClose tempHandle
            renameFile tempName dest)
```

はじめに main でコマンドライン引数を取得したら、 copy 関数を呼び出すだけです。ファイルコピーの魔法は copy 関数がかけます。単にファイルを読み込み、もう片方のファイルに書き込めばコピーはできますが、何かまずいことが起こると（コピーするのにディスクの空きが足りないなど）、おかしなファイルが

残ってしまいます。そこで、いったん一時ファイルに書き込むことにします。何かおかしなことが起こっても、そのファイルを消すだけで済みます。

最初に、 B.readFile を使ってコピー元ファイルの内容を読みます。それから bracketOnError を使ってエラーハンドラをセットアップします。リソースの獲得は openTempFile "." "temp" です。一時ファイルの名前とそのハンドルのタプルが返されます。次に、エラーが起こったときに何をすべきかを書きます。まずいことが起こったなら、ハンドルを閉じ、一時ファイルを削除します。最後がコピー処理の本体です。 B.hPutStr を使って内容を一時ファイルに書き出します。一時ファイルを閉じて、それをコピー先ファイル名に変更したら、望みの処理が完了です。

readFile と hPutStr の代わりに B.readFile と B.hPutStr を使っただけ、というのがポイントです。ファイルを開いたり閉じたり、ファイル名を変更したりするのに、bytestring 用の特別な関数を使っていません。bytestring の関数を必要とするのは読み書きのときだけです。

試してみましょう。

```
$ ./bytestringcopy bart.txt bort.txt
```

bytestring を使わないプログラムも、だいたい同じような感じになるでしょう。唯一の違いは、readFile と writeFile の代わりに B.readFile と B.writeFile を使っていることだけです。

普通の文字列を使ったプログラムを bytestring を使ったものに書き換えるには、多くの場合、単に修飾付きインポートして対応する関数の前にモジュール名を付け足すだけです。文字列に対して動作する関数を bytestring で動作するように書き換える必要が出てくることはありますが、難しくはありません[†3]。

たくさんのデータを文字列として読み込むプログラムで、より良いパフォーマンスを必要とするなら、bytestring を試してみてください。とても少ない労力でパフォーマンスを向上できるチャンスです。僕はいつも、まずプログラムを普通の文字列で書いてみて、パフォーマンスが満足できるものでなければ bytestring を使ったものに書き換えるようにしています。

[†3] ［訳注］実際には留意すべき点がいくつかあります。文字列処理については巻末の付録 A に訳者による補足をまとめてあるので、そちらも参照してください。

第10章
関数型問題解決法

この章ではいくつかの面白い問題をとりあげ、それを関数型プログラミングのテクニックでできる限りエレガントに解く方法を考えます。今までの講習で身に付けた Haskell 筋力をさっそく奮って、コーディングスキルを鍛えてください。

10.1 逆ポーランド記法電卓

普通、学校で算数を習うときは中置記法で式を書きます。例えば `10 - (4 + 3) * 2` という具合に。足し算（`+`）、掛け算（`*`）、引き算（`-`）などは、Haskell の中置関数（`+` とか `` `elem` `` とか）と同じく、中置演算子です。人間は中置記法の数式をいとも簡単に脳内でパーズできます。欠点は、演算の優先順位を指定するのに括弧が必要になることです。

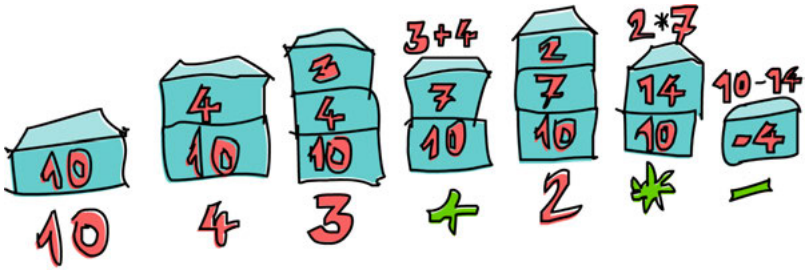
数式を書く別の方法として**逆ポーランド記法**（reverse polish notation）、略して**RPN**があります。RPN では、演算子は数に挟まれるのではなく、数の後にきます。`4 + 3` と書く代わりに `4 3 +` と書くわけです。では複数の演算子を含む数式はどのように書くのでしょう？ 例えば、まず `4` と `3` を足して、それに `10` を掛ける、という式はどう書くのでしょう？ 答は簡単です。`4 3 + 10 *` と書きます。ここで `4 3 +` は `7` なので、数式全体は `7 10 *` と等価になります。

RPN 記法の式を計算

RPN 記法を計算する方法を感覚的につかむには、数のスタックをイメージしてください。RPN 記法は、左から右へ読みます。数を読み込んだら、それをスタックのてっぺんに積みます（push）。演算子を読み込んだら、スタックのてっぺんから 2 つの数を取り出し（pop）、その 2 つの数に演算子を施して、演算結果をスタックに積み直します。式の末尾に辿り着いたら、計算結果を示す数が 1 つだけスタックに残っているはずです（数式の構文が正しければ、ですが）。

例えば、RPN 式 `10 4 3 + 2 * -` をどうやって評価するのか見ていきましょう。

1. まず `10` をスタックに積み、スタックは `10` が 1 つ入った状態になります。
2. 次のアイテムは `4` なので、これもスタックに積みます。今のスタックは `10, 4` です。
3. `3` にも同じことをして、スタックは `10, 4, 3` になります。
4. あっ、今度は演算子がきました。 `+` です。そこでスタックから 2 つの数を取り出し（スタックは `10` だけになります）、取り出した 2 つの数を加算し、計算結果をスタックに戻します。こうしてスタックは `10, 7` になります。
5. 次にスタックに `2` を積み、スタックは `10, 7, 2` になります。
6. また演算子がきました。 `7` と `2` をスタックから取り出し、掛け算して、結果をスタックに戻します。 `7` 掛ける `2` は `14` ですから、スタックは `10, 14` になります。
7. 最後に、 `-` があります。 `10` と `14` をスタックから取り出し、 `10` から `14` を引いて、スタックに戻します。
8. スタックに乗っている数は `-4` です。与えられた式には、もう数も演算子も残っていませんから、これが答です！



これが RPN 式を手で計算する方法です。では、Haskell で同じことをするにはどうすればいいか考えましょう。

RPN 関数を書く

`"10 4 3 + 2 * -"` のような RPN 式を文字列で受け取って、その式の結果を返す関数を書いていきましょう。

この関数の型はどうなるでしょうか？「文字列を引数に取って、数を結果として返す関数」ですね。割り算もできるようにしたいので、結果は倍精度浮動小数がいいでしょう。というわけで、型はこんな感じになるんじゃないでしょうか。

```
solveRPN :: String -> Double
```

NOTE 関数の実装に取り掛かる前に、まず関数の型宣言がどうなるか考えるのはとても役立つ習慣です。型システムがとても強力なおかげで、Haskell では関数の型宣言を見ればその関数について実にいろんなことが分かります。

Haskell で何かの問題を解くプログラムを実装するとき、その問題を手で解いてみた経験がヒントになることがあります。RPN 式を手計算したときは、空白で区切られた数や演算子のそれぞれを 1 つのアイテムとして扱いました。ということは、`"10 4 3 + 2 * -"` のような文字列を、まずはアイテムのリストに分割することから始めるとよさそうですね。

書き出してみますよ。

```
["10","4","3","+","2","*","-"]
```

それから、頭の中でこのリストをどう処理したんでしたっけ？ このリストを左から右へと走査して、その間スタックを更新し続けたのでした。このプロセスって何かを思い出しませんか？ 5.5 節「畳み込み、見込みアリ！」では、リストを一要素ごとに走査しながら何らかの結果を積み上げていく（**アキュムレート**する）関数というのは、どんなアキュムレータに対しても「畳み込み」を使って実装できることを見ました。数も、リストも、スタックも、そのほか何でも。

今回の場合、リストを左から右へ走査するので、左畳み込みを使いましょう。アキュムレータ値はスタックなので、畳み込みが返す結果もスタックになるはずです。ただし、すでに見たように値が 1 つしか入っていないスタックです。

もう 1 つ、スタックをどう表現するかも考える必要があります。そうですね、リストを使って、リストの先頭をスタックの先頭に対応させるのがいいでしょう。リストの先頭（`head`）に要素を追加するのは、末尾に追加するよりもずっと高速ですから。例えば `10, 4, 3` というスタックは、`[3,4,10]` というリストとして表現することになります。

これで作りたい関数の姿がおぼろげながら見えてきました。まずその関数は `"10 4 3 + 2 * -"` といった文字列を取り、それを `words` を使ってアイテムの

リストに分解します。次に、そのリストに左畳み込みを使って、単一の要素が入ったスタック（この例の場合は、 `[-4]` ）に辿り着きます。その単一要素をリストから取り出して、それがファイナルアンサーです！

これがその関数のスケッチです。

```
solveRPN :: String -> Double
solveRPN expression = head (foldl foldingFunction [] (words expression))
    where  foldingFunction stack item = ...
```

この関数は expression を取って、まずアイテムのリストに変えます。それからそのアイテムのリストを関数 foldingFunction で畳み込みます。ここで `[]` はアキュムレータの初期値です。アキュムレータはスタックなので、 `[]` は空のスタックを表しています。そして単一要素の入った最終状態のスタックを受け取ったら、 head を使ってアイテムを取り出します。

さて、あとは畳み込み処理を行う関数を書くだけです。その関数は、例えばスタック `[4,10]` とアイテム `"3"` を受け取って、新しいスタック `[3,4,10]` を返します。また、スタックが `[4,10]` で、受け取ったアイテムが `"*"` なら、関数は `[40]` を返すべきです。

あっ、畳み込み関数を書く前に、全体の関数をポイントフリースタイルで書き直しておきましょう。括弧がいっぱいあるのを見ていると精神が不安定になってきますからね。

```
solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
    where  foldingFunction stack item = ...
```

これですっきりしました。

畳み込み関数は、スタックとアイテムを取って新しいスタックを返すようにします。関数定義構文でパターンマッチを使い、スタックの上側にあるアイテムを取り出す処理と `"*"` や `"-"` のような演算子を識別する処理を一気にやりましょう。これが畳み込み関数の実装です。

```
solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
    where  foldingFunction (x:y:ys) "*" = (y * x):ys
           foldingFunction (x:y:ys) "+" = (y + x):ys
           foldingFunction (x:y:ys) "-" = (y - x):ys
           foldingFunction xs numberString = read numberString:xs
```

4 つのパターンが並んでいます。パターンは上から下へ順番に試されます。まず、畳み込み関数は現在のアイテムが `"*"` かどうか調べます。もしそうなら、`[3,4,9,3]` のようなリストを取り、その先頭の 2 つの要素をそれぞれ x と y と名づけます。この場合だと、 x が 3 で y が 4 になります。 ys は `[9,3]` になり

ます。畳み込み関数は ys の頭に「x 掛ける y」を付けて返します。こうして、スタックから先頭の 2 つの数を取り出し、掛け算して、結果をスタックに積む操作が書けました。もしアイテムが "*" でなかったら、このパターンマッチを抜け落ちて、次は "+" を調べます。以下同様。

アイテムが既知の演算子のいずれでもなかったら、それは数を表す文字列だと仮定します。もし本当に数だったら、 read を適用すれば中身が実数に変換できるはずで、その数を以前のスタックに積んで返します。

例えば ["2","3","+"] というアイテムのリストに対して、 solveRPN は左から処理を始めます。初期のスタックは [] です。 solveRPN は、スタック [] をアキュムレータとし、 "2" をアイテムとして foldingFunction を呼び出します。このアイテムは演算子ではないので、数値として解釈され、 [] の先頭に追加されます。こうしてスタックは [2] になります。再び foldingFunction が、 [2] をスタック、 "3" をアイテムとして呼び出され、新しいスタック [3,2] を生み出します。もう一度 foldingFunction が、 [3,2] をスタック、 "+" をアイテムとして呼び出されます。すると 2 つの数がスタックから出てきて、加算され、スタックに戻されます。スタックの最終状態は [5] になり、この数が返ってきます。

この関数で遊んでみましょう。

```
ghci> solveRPN "10 4 3 + 2 * -"
-4.0
ghci> solveRPN "2 3.5 +"
5.5
ghci> solveRPN "90 34 12 33 55 66 + * - +"
-3947.0
ghci> solveRPN "90 34 12 33 55 66 + * - + -"
4037.0
ghci> solveRPN "90 3.8 -"
86.2
```

動いてますね。素晴らしい！

演算子を追加しよう

この解法の何が良いかって、他のいろんな演算を簡単にサポートできるところです。二項演算子である必要もありません。例えば、数を 1 つだけ取り出してその対数を積む "ln" という演算も作れます。可変長の引数を取る演算子だって作れますよ。例えば "sum" は、スタックからすべての数を取り出してその総和を積む演算です。

RPN 電卓関数を修正してもっと多くの演算子をサポートするようにしましょう。

```
solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
    where  foldingFunction (x:y:ys) "*" = (y * x):ys
           foldingFunction (x:y:ys) "+" = (y + x):ys
           foldingFunction (x:y:ys) "-" = (y - x):ys
           foldingFunction (x:y:ys) "/" = (y / x):ys
           foldingFunction (x:y:ys) "^" = (y ** x):ys
           foldingFunction (x:xs) "ln" = log x:xs
           foldingFunction xs "sum" = [sum xs]
           foldingFunction xs numberString = read numberString:xs
```

/ はもちろん割り算で、 ** は冪乗演算です。対数演算子に関しては、スタックの先頭要素 1 つだけと残り全部、というパターンマッチを行います。自然対数を計算するのに引数は 1 つしかいらないからです。総和演算子に関しては、直前のスタックに入っていた数の総和、という数が 1 つだけ入ったスタックを返します。

```
ghci> solveRPN "2.7 ln"
0.9932517730102834
ghci> solveRPN "10 10 10 10 sum 4 /"
10.0
ghci> solveRPN "10 10 10 10 10 sum 4 /"
12.5
ghci> solveRPN "10 2 ^"
100.0
```

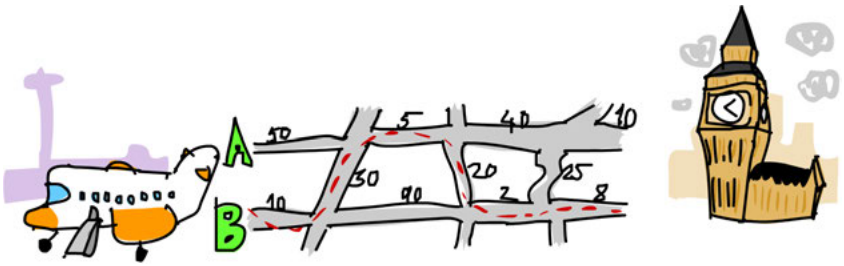
任意の浮動小数点数の RPN 式を計算できて、しかも容易に拡張できる関数がたったの 10 行で書けるというのは、かなり驚異的だと思いますよ。

NOTE この RPN 電卓ソリューションの耐障害性はまったくもってお粗末です。意味の通らない入力を受け取ると、実行時エラーになる可能性があります。でも心配ご無用。この関数をもっと頑健にする方法を第 14 章で扱います。

10.2　ヒースロー空港からロンドンへ

さて、僕らは卒業旅行に来ているとしましょう。今、飛行機でイギリスに着いた僕らはレンタカーを借りました。行きたい場所はたくさんあるので、なるべく急いでヒースロー空港からロンドンへ行かねばなりません（もちろん、安全に！）。

ヒースロー空港からロンドンへは 2 本の幹線道路が平行に走っており、その 2 つを何本もの地方道路が橋渡ししています。ある交差点から次の交差点までの所要時間は一定とします。ロンドンでの会議に間に合うかどうかは、僕らが最適な経路を見つけられるかにかかっています。僕らは図の左側からスタートし、地方道路を使って幹線道路を乗り換えるか、幹線道路を先に進むかを選べます。



ヒースロー空港からロンドンへの最短時間の経路は、この絵から分かるように、幹線道路 B からスタートし、幹線道路 A に乗り換え、A を先に進み、また幹線道路 B に乗り換え、B を先に 2 回進む経路です。この経路をとれば 75 分でロンドンに着きます。他のどんな経路をとっても、これより長い時間がかかります。

僕らの仕事は、この道路網を表現した入力を受け取り、最短経路を出力するプログラムを作ることです。これが冒頭の図に対応する入力です。

```
50
10
30
5
90
20
40
2
25
10
8
0
```

この入力ファイルは、3 つずつの数字の塊で道路網をセクションに分割している、と思って眺めてください。各セクションの 3 つの数字は、それぞれ幹線道路 `A`、幹線道路 `B`、および地方道路です。入力がちょうど三つ組の集合に当てはまるように、最後の最後に所要時間が 0 分の地方道路があることにします。「どっからでもいい、とにかくロンドンに着けばいいんだ！」ということです。

RPN 電卓の問題を解いたときと同様に、この問題も三段階で解くことにします。

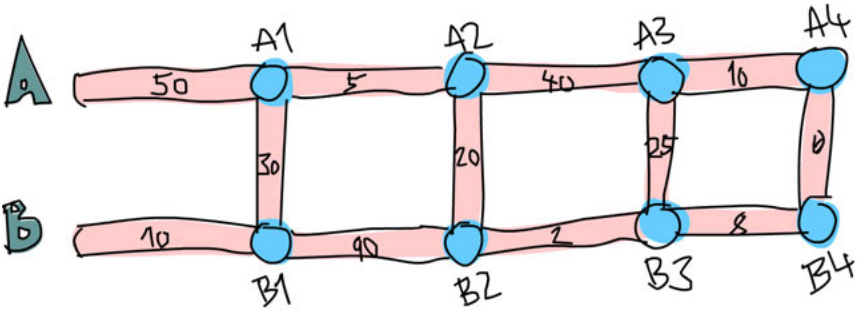
1. Haskell のことはいったん忘れて、問題を手で解く方法を考えます。RPN 電卓のときは、問題をまず手で解いてみることで、どうやら人間が RPN 式を処理するときには頭の中にスタックのようなものを思い浮かべ、数式のアイテムを 1 つずつ処理していくようだ、と気がついたのでした。

2. そのデータ構造を Haskell でどう表現しようか考えます。RPN 電卓の場合は、文字列のリストを使って数式を表現することにしたのでした。
3. そのデータ構造を Haskell で操作して、解を得る方法を考えます。電卓の場合は、左畳み込みを使って文字列のリストを走査しながら、スタックを更新していって解を得たのでした。

最速経路を計算する

さて、ヒースローからロンドンまでの最速経路を手で計算するにはどうしたらいいでしょう？ ええと、絵を眺め、心眼を働かせ、勘が当たっていることを祈る、という手もありますね。その解法は、ごく小さい入力に対してはうまくいきます。でもセクションが 1 万個あるような道路網だったら？ お手上げです！ しかも勘で決めた経路では、最短であるという保証がどこにもありませんね。まあ、あたらずとも遠からず、ぐらいでしょうか。つまり、これは良い解法ではないわけです。

これは道路網の絵をシンプルにしたものです。



幹線道路 A の第一交差点 A1 までの最短経路は分かりますか？ これは簡単ですね。ヒースローから A を真っすぐ行くのか、 B を行ってから A へ乗り換えるのか、どちらが早いかさえ比べれば分かります。明らかに B を行ってから乗り換えるほうが早いです。そっちは 40 分ですが、 A を真っすぐ行くのは 50 分かかります。では、 B1 はどうでしょう？ こちらは、 B を真っすぐ行く（所要時間 10 分）ほうがずっと早いですね。なにしろ A を走ってから乗り換えるのでは 80 分もかかります！

これで A1 までの最短経路が分かりました。 B を通って進み、それから A へ乗り換える。この経路を B, C と表すことにしましょう（ C は cross の頭文字です）。この経路のコストは 40 分です。また B1 への最短経路も分かりました。 B

を真っすぐ進むことです。これはB単体からなる経路で、そのコストは10分です。この知識は、次の交差点までの最短距離を求めるのに役立つでしょうか？もちろん、とっても役立ちます！

では、A2までの最短経路を求めてみましょう。A2に行くには、A1からA2へ直行するか、B1から前に進んで乗り換えるかの2通りがあります（僕らは前か横にしか動けませんでしたよね）。

NOTE もしかして「あれっ？ A2に行くのに、B1で乗り換えて、それから前進するのは？」と思った？ 残念！ B1からA1への経路はA1までの最短経路を探索するときにすでに考慮したので、次のステップでまた考える必要はありませんでした。

A1までとB1までのコストはもう知っているのですから、A2までの最短経路は簡単に計算できます。A1までは40分で、A1からA2までは5分なのですから、この経路B, C, Aのコストは45です。一方、B1までの所要時間はわずか10分ですが、そこからB2へ進んで乗り換えるのには110分もかかってしまいます！ これはもう明らかに、A2までの最短経路はB, C, Aです。同じようにして、B2までの最短経路は、A1から前へ進み、それから乗り換えることです。

こうしてA2とB2への最短経路が得られました。同じことを繰り返せば最後まで行けます。そうやって、A4とB4への最短経路を求めれば、そのうちで所要時間が少ないものが最も良い経路です。

2つ目のセクションでやったことは、本質的には1つ目のセクションでやったことの繰り返しですが、直前のAまたは直前のBを使う最短経路を考慮しています。1つ目のセクションでも、ある意味では、直前のAとBまでの最短経路を使ったと言えます。2つとも空の経路でありコストは0分であった、と考えればいいわけです。

まとめると、ヒースロー空港からロンドンまでの最短経路を求めるアルゴリズムはこうなります。

1. 幹線道路Aのヒースロー空港から1つ目の交差点までの最短経路は何かを求めます。選択肢は、直接Aを進むか、Bからスタートして乗り換えるかの2つです。この経路とそのコストを記録します。
2. 同じ手続きで、幹線道路Bの最初の交差点までの最短経路を求めて記録します。
3. 幹線道路Aの、さらに次の交差点までの経路は、幹線道路Aの直前の交差点から真っすぐ進むのが早いか、直前のBの交差点から真っすぐ進んで乗り換えるのが早いか調べて、早いほうを採用します。逆側の交差点についても同じことをします。
4. これを目的地まで繰り返します。

5. 目的地まで繰り返したら、2 つの経路のうち早いほうが僕らのとるべき最短経路です。

というわけで、幹線道路 A と B の最短経路を 1 つずつ覚えておけばよいわけです。そうやって目的地までいったら、2 つのうちの早いほうが求めていた経路です。

さて、最短経路を手で求める方法は分かりました。もし十分な時間と、紙と鉛筆があれば、どんなに多くのセクションがある道路網でも最短経路を求められることでしょう。

道路網を Haskell で表現する

どうすればこの道路網を Haskell のデータ型で表現できるでしょう？

手で解いたときを思い出すと、3 つの所要時間を 1 組として調べていたのでした。幹線道路 A のあるパーツ、幹線道路 B の同じパーツ、そして 2 つのパーツと接続し橋渡しをしているパーツ C です。A1 と B1 への最短経路を求めたときは、最初の 3 つのパーツの所要時間（50 分、10 分、および 30 分）だけを使いました。この 3 つ組を「Section」と呼びましょう。こうして、この例題に出てくる道路網は、4 つの Section の集まりとして簡単に表現できるようになりました。

- 50, 10, 30
- 5, 90, 20
- 40, 2, 25
- 10, 8, 0

データ型はできるだけシンプルにしましょう（でもシンプルならいいってわけじゃないですよ！）。これが僕らの道路網のデータ型です。

```
data Section = Section { getA :: Int, getB :: Int, getC :: Int }
    deriving (Show)

type RoadSystem = [Section]
```

十分にシンプルだし、僕らの解を実装するのにはうってつけな気がしますね！

Section は 3 つの道路パーツの所要時間に対応する整数を保持する単純な代数データ型です。道路網 RoadSystem は道路 Section のリストである、という型シノニムも導入することにします。

NOTE　道路 Section をトリプル（Int, Int, Int）で表すという手もあったかもしれません。独自の代数データ型を作る代わりにタプルを使う選択のほうが、影響が小さく、局所的なものを表すには優れています。しかし、より複雑なものを表現するには新し

> い型を作るほうが適切です。そうするほうが、どの値が何なのか、型システムにより多くの情報を与えられます。(Int, Int, Int) は、道路のセクションのほか三次元空間のベクトルを表すのにも使え、その両者に対する演算も可能ですが、それでは両者を混同する恐れがあります。Section と Vector というデータ型を使えば、うっかり三次元ベクトルを道路セクションに加算することはなくなります。

ヒースロー空港からロンドンまでの道路網はこう表せます。

```
heathrowToLondon :: RoadSystem
heathrowToLondon = [ Section 50 10 30
                   , Section 5 90 20
                   , Section 40 2 25
                   , Section 10 8 0
                   ]
```

あとは Haskell で解を実装するだけです！

最短経路関数を求めよ！

どんな道路網に対しても最短経路が計算できる関数は、どんな型宣言にすればいいでしょう？ その関数は、道路網を引数として取って経路を返すべきです。経路もリストとして表現しましょう。

A、B、C を列挙しただけの Label 型を導入することにします。それから、Path という名前の型シノニムを作りましょう。

```
data Label = A | B | C deriving (Show)
type Path = [(Label, Int)]
```

ということで、これから作る関数 optimalPath の型はこうなるはずです。

```
optimalPath :: RoadSystem -> Path
```

heathrowToLondon という名の道路網を引数にして呼び出したら、こういう経路を返すべきです。

```
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8)]
```

道路 A での最短経路と道路 B での最短経路を保持しながら、セクションのリストを左から右へと辿っていきましょう。リストを辿りながら、最善の経路をためていく。これって何でしょう？ 3、2、1、……はい時間です！ 正解は**左畳み込み**！

手計算で解を求めたときは、あるステップを何度も何度も繰り返しました。直前の A と B までの最適経路、それに現在の道路セクションをもとに、A と B の新しい最適経路を求めるというステップです。開始時点で A と B に対応する最適経路は、それぞれ [] と [] です。そして Section 50 10 30 というセクションを読み込み、A1 への最適経路は [(B,10),(C,30)] であり、B1 への最適経

路は [(B,10)] である、ということを求めました。この手続きを関数視点で見ると、経路のペアとセクションを引数に取って新しい経路のペアを返す関数になっています。ということは、その型はこうです。

```
roadStep :: (Path, Path) -> Section -> (Path, Path)
```

この関数を実装してみましょう。きっと役に立つはずです。

```
roadStep :: (Path, Path) -> Section -> (Path, Path)
roadStep (pathA, pathB) (Section a b c) =
    let timeA = sum (map snd pathA)
        timeB = sum (map snd pathB)
        forwardTimeToA = timeA + a
        crossTimeToA = timeB + b + c
        forwardTimeToB = timeB + b
        crossTimeToB = timeA + a + c
        newPathToA = if forwardTimeToA <= crossTimeToA
                     then (A, a):pathA
                     else (C, c):(B, b):pathB
        newPathToB = if forwardTimeToB <= crossTimeToB
                     then (B, b):pathB
                     else (C, c):(A, a):pathA
    in  (newPathToA, newPathToB)
```

何をしてるのでしょうか？ まず、そこまでの A の最適経路の所要時間を求めます。 B についても同じです。これには sum (map snd pathA) という関数を使います。例えば pathA が [(A,100),(C,20)] だったら、timeA は 120 になります。

forwardTimeToA は、幹線道路 A の直前の交差点から次の交差点まで真っすぐ進んだときの所要時間です。これは、直前の A までの所要時間と、現在のセクションの A パーツの所要時間の和に等しいはずです。

crossTimeToA は、道路 A の次の交差点まで、道路 B を直進してから乗り換える、という経路を選んだ場合の所要時間です。これは、直前の B までの所要時間に、現在のセクションの B の所要時間と C の所要時間を足したものに等しくなるはずです。

forwardTimeToB と crossTimeToB も、同様にして求めます。

これで A と B への最短経路が分かりましたから、あとは A と B への新しい経路を生成するだけです。もし、次の A まで行くのに、ただ直進するほうが早ければ、 newPathToA を (A, a):pathA に設定します。 A という Label と、 a の所

要時間を、直前の A までの最適経路に追加しています。「次の A までの最短経路は、直前の交差点 A まで最短経路で行って、それから A を直進することだよ！」と言いたいわけです。 A はただのラベルです。一方、 a は Int 型であることを忘れないでくださいね。

なぜ、 pathA ++ [(A, a)] としないで、要素を前から追加しているのでしょうか？ リストの先頭に要素を追加するのは、リストの末尾に追加するよりもずっと速いんです。前に追加しているため、この関数でリストを畳み込んで出てきた経路は逆向きになってしまいますが、最後に逆順にするのは簡単です。

もし A の次の交差点に行くのに B を直進してから乗り換えるほうが早ければ、newPathToA は、直前の B まで行き、そこから直進して A へ乗り換えるものになります。 newPathToB についても同じことを繰り返します。ただし、 A と B 、 a と b はすべて反対にします。

最後に、 newPathToA と newPathToB をペアにして返します。

では、この関数を heathrowToLondon の最初のセクションについて走らせてみましょう。最初のセクションなので、直前の A と B への最短経路に対応する引数は空リストのペアになります。

```
ghci> roadStep ([], []) (head heathrowToLondon)
([(C,30),(B,10)],[(B,10)])
```

経路は逆転しているので、右から左へ読むのでしたよね。次の交差点 A までの最短距離は、 B からスタートして A に乗り換えることだと分かります。一方、次の B までの最短経路は、単に B を直進することです。

NOTE この roadStep の実装では、 timeA = sum (map snd pathA) とするときに経路の所要時間を毎回計算しています。 A と B の最短経路だけでなく、最短の所要時間を引数に取ったり返したりする仕様にしておけば、この計算は必要ありませんでしたね。

経路のペアとセクションを取って最適経路を返す関数ができたところで、セクションのリストに左畳み込みを施すのは簡単です。 roadStep は、 ([], []) と最初のセクションを引数にして呼ばれ、そのセクションまでの最適経路を求めます。次に、その最適経路と次のセクションを引数に roadStep がまた呼ばれて、と続きます。すべてのセクションを歩き終わったら、最後に最適経路のペアが残ります。そして、2 つのうちの短いほうが答です。これを念頭に optimalPath を実装しましょう。

```
optimalPath :: RoadSystem -> Path
optimalPath roadSystem =
    let (bestAPath, bestBPath) = foldl roadStep ([], []) roadSystem
    in  if sum (map snd bestAPath) <= sum (map snd bestBPath)
            then reverse bestAPath
            else reverse bestBPath
```

この関数では、空リストのペアを初期アキュムレータとして、 roadSystem（これはセクションのリストでしたね）を左畳み込みしています。その結果は経路のペアですから、パターンマッチを使って 2 つの経路の本体を取り出します。それからどちらの所要時間が短いかを判定して返します。経路リストの後方ではなく前方へ追記していく実装を選んだことから、この時点での経路は逆転しているので、返す前に逆順にしています。

では試してみましょう！

```
ghci> optimalPath heathrowToLondon
[(B,10),(C,30),(A,5),(C,20),(B,2),(B,8),(C,0)]
```

これがまさに欲しかった結果です！ あ、期待していた結果とはちょっと違いますね。なんか、最後に (C,0) とかいうステップが付いてます。ロンドンに着いてから反対の道に乗り換えろ、と。この乗り換えには時間がかからないことになってるので、これで正解です。

入力から道路網を受け取る

最短経路を求める関数はできました。あとはテキスト形式で表現された道路網を標準入力から読み込んで RoadSystem 型に変換し、 optimalPath 関数にかけて、得られた経路を表示するだけです。

まず、リストを取り、ある要素数のグループごとに分割する関数を作りましょう。名づけて groupsOf です。

```
groupsOf :: Int -> [a] -> [[a]]
groupsOf 0 _ = undefined
groupsOf _ [] = []
groupsOf n xs = take n xs : groupsOf n (drop n xs)
```

[1..10] という引数に対して、 groupsOf 3 の結果はこうなるはずです。

```
[[1,2,3],[4,5,6],[7,8,9],[10]]
```

見てのとおり、 groupsOf は標準的な再帰関数です。 groupsOf 3 [1..10] とするのは、こうするのと同じです。

```
[1,2,3] : groupsOf 3 [4,5,6,7,8,9,10]
```

再帰が完了したとき、入力リストは3つずつの組になって得られます。いよいよこれが、標準入力を読み取って RoadSystem を作り、最短経路を出力するメイン関数です。

```
import Data.List

main = do
    contents <- getContents
    let threes = groupsOf 3 (map read $ lines contents)
        roadSystem = map (\[a,b,c] -> Section a b c) threes
        path = optimalPath roadSystem
        pathString = concat $ map (show . fst) path
        pathTime = sum $ map snd path
    putStrLn $ "The best path to take is: " ++ pathString
    putStrLn $ "Time taken: " ++ show pathTime
```

まず、標準入力からすべてのコンテンツを読み込みます。それから、このコンテンツに lines を適用してきれいな形にします。例えば "50\n10\n30\n ..." だったら ["50","10","30" ...] になります。次にそれを read で写して、数のリストに変えます。さらに groupsOf 3 を適用し、長さ3のリストのリストに変えます。さらに、その二重リストをラムダ式 (\[a,b,c] -> Section a b c) で写します。

見てのとおり、ラムダ式は長さ3のリストを取って道路セクションに変えています。この結果、 roadSystem は解きたい道路網となり、正しい型 RoadSystem (あるいは [Section]) を持ちます。これに optimalPath を適用し、経路と総所要時間を適切なテキスト形式に変えて、表示します。

以下のような文字列を paths.txt というファイルに保存してください。

```
50
10
30
5
90
20
40
2
25
10
8
0
```

そして僕らのプログラムに食わせましょう。

```
$ runhaskell heathrow.hs < paths.txt
The best path to take is: BCACBBC
Time taken: 75
```

みごとに動きました！

Data.Random モジュールの知識を使って、もっと長い道路網を生成し、今作ったばかりのコードに食わせてみてください。もしスタックオーバーフローが出たら、 foldl を foldl' に変え、 sum を foldl' (+) 0 に変えてみましょう。または、実行する前にこうやってコンパイルしてみてください。

```
$ ghc --make -O heathrow.hs
```

コンパイル時に O フラグを含めると、最適化が働いて、 foldl や sum といった関数がスタックオーバーフローしにくくなります。

第11章
ファンクターから アプリカティブファンクターへ

純粋性、高階関数、型引数を取る代数的データ型（parameterized algebraic data types）、型クラスを兼ね備える Haskell では、他のプログラミング言語よりずっと簡単に多相性を実装できます。型の巨大な階層構造に気をもむ必要はありません。その代わり、これらの型はどのように振る舞うか？ と考えて、適切な型クラスに関連付ければよいのです。例えば、`Int` はさまざまな「もの」のように振る舞います。同じかどうか判定できるもの、順序が付いたもの、列挙できる（数え上げられる）もの、などなど。

型クラスはオープンです。つまりデータ型を定義し、その型がどう振る舞うかを考えてから、その振る舞いを定義する型クラスに属させることができるのです。このオープンな型クラスと、関数の型宣言だけから多くのことを読み取れる Haskell の強力な型システムのおかげで、とても一般的で抽象的な振る舞いを定義する型クラスが作れます。

これまでに、2 つのものが等しいかどうか判定する操作や、2 つのものを何らかの順序で比較する操作を定義する型クラスを見てきました。こう言うと抽象的で高尚に聞こえますが、等号とか比較って日常的に使う操作で、とりわけ特別なことではないですよね。それから、第 7 章ではファンクターを紹介しました。ファンクターは、関数を使って全体を写せるもの、という感じでした。これもまた型クラスを使って定義できる、便利で、それでいてかなり抽象的な性質の 1 つです。この章では、ファンクターをより詳しく見ていきます。そして、ファンクターの少し強力でより便利なバージョンである**アプリカティブファンクター**も紹介します。

11.1　帰ってきたファンクター

第 7 章で学んだように、ファンクターとは関数で写せるもののことです。例えば、リスト、Maybe、木などがファンクターです。Haskell では、ファンクターは型クラス Functor で表現されます。ファンクターの型クラスメソッドは 1 つだけで、それは fmap です。 fmap の型は fmap :: (a -> b) -> f a -> f b です。この型の意味は「僕に『a を取って b を返す関数』と、a の入った箱を渡して。そしたら b の入った箱にして返すよ」という感じです。箱 f の中の要素に関数を適用してくれるのですね。

ファンクターは、**文脈**を持った値だとみなすこともできます。例えば、Maybe 値は「計算が失敗したかもしれない」という文脈を、リストは「複数の値を同時にとるかもしれない」という文脈を持ちます。 fmap は、こういった文脈を保ったまま関数を値に適用するのです。

型コンストラクタを Functor のインスタンスにするには、その型コンストラクタの種類（kind）は * -> * でないといけません。つまり、その型コンストラクタは、型変数として具体型をただ 1 つ取る必要があります。例えば Maybe は、Maybe Int や Maybe String のように 1 つの型変数を取って具体型を生むので、ファンクターになれます。これに対し、 Either のような 2 つの型変数を取る型コンストラクタをファンクターにするには、部分適用をして、あと 1 つだけ型変数を引数に取る状態にしないといけません。というわけで、**instance** Functor Either **where** は**間違い**ですが、**instance** Functor (Either a) **where** は**正しい**インスタンス宣言です。そして、fmap は Either a に働くのだと考えれば、次のような型になることが分かります。

```
fmap :: (b -> c) -> Either a b -> Either a c
```

fmap の前後で Either a の部分は不変であり、 Either a のただ 1 つの変数の部分が変化しています。

ファンクターとしての I/O アクション

これまで実に多くの型（えーと正確に言えば型コンストラクタ）が Functor のインスタンスであることを見てきました。 [] 、 Maybe 、 Either a 、それから第 7 章で作った Tree などです。 a -> b 型の関数を使って [a] 、 Maybe a 、 Either a1 a 、 Tree a などの型をも写せるのはとても便利でした。今度は IO インスタンスを見てみましょう。

例えば、 IO String という型は、実行すると、外の世界に出かけていって文字列を取ってきてくれて、それを返してくれるような I/O アクションを表して

います。取得結果は、 do 記法の中で <- 構文を使って名前に束縛できます。第8章では、I/O アクションとは小さな足のはえた箱で、外の世界に出かけていって何か値を入れて帰ってきてくれるのだ、という比喩を使いました。僕らは IO が取ってきてくれた値を調べることができますが、調べた後はまた IO の中に戻さなくてはいけません。この「足のはえた箱」の例えで、 IO がファンクターの一種であることが理解できます。

IO の Functor インスタンスがどのように定義されているか見ていきましょう。ある関数をある I/O アクションに fmap すると、「元の I/O と同じことをしつつ、その結果に指定した関数を適用して返す」I/O アクションになってほしいわけです。これが実装です。

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

I/O アクションを関数で写した結果もまた I/O アクションなので、まずは脊髄反射的に do 構文を使います。この中で 2 つのアクションを貼り付けて新しい I/O アクションを作ればいいはずです。 fmap を実装するには、まずもともとの I/O アクションを実行して結果を result と名づけます。次に return (f result) を行います。 return とは、ご存知のとおり「特に仕事を行わず、ただ何かを結果として提示する I/O アクション」を作る関数です。

do ブロックで組み立てたアクションは常に最後のアクションの結果を提示します。ですので、 return を使って仕事をしないアクションを作り、 f result を do ブロック全体の結果として提示させるわけです。このコードを見てください。

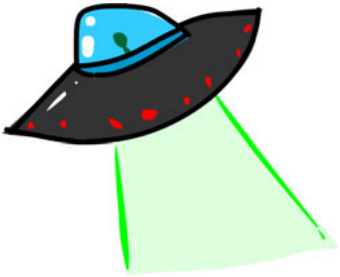
```
main = do line <- getLine
          let line' = reverse line
          putStrLn $ "You said " ++ line' ++ " backwards!"
          putStrLn $ "Yes, you said " ++ line' ++ " backwards!"
```

ユーザに文字列を入力してもらい、それを逆順にして表示しています。これは fmap を使うとこう書けます。

```
main = do line <- fmap reverse getLine
          putStrLn $ "You said " ++ line ++ " backwards!"
          putStrLn $ "Yes, you really said " ++ line ++ " backwards!"
```

Just "blah" を fmap reverse して Just "halb" を作るのと同じように、getLine を fmap reverse できるのです。 getLine は IO String の型を持つ I/O アクションですから、それを fmap reverse すると、外の世界に出かけていって文字列を 1 行入力してもらって、その結果に reverse を適用する I/O ア

クションになります。Maybe という箱に入っている値に関数を適用できるのと同様に、IO という箱の中に入っている値に関数を適用できるわけですが、fmap した結果も IO なので、そいつもやはり外の世界に出かけていって何か値を取ってくるという動作をするわけです。その後、reverse を適用済みの値を、<- を使って line という名前に束縛しています。



ほかにも、fmap (++"!") getLine という I/O アクションは getLine とほぼ同じ動作をしますが、入力された文字列の末尾に "!" を付けます!

仮に fmap が IO に限定されていたら、fmap の型は fmap :: (a -> b) -> IO a -> IO b になります。fmap は、a -> b な関数と IO a な I/O アクションを取って、「それとよく似た動作をして中身の値に関数を適用するような I/O アクション」を返す関数です。

もし自分のコードに、何らかの関数に渡すだけのために I/O の結果に名前をつけている箇所があったら、fmap を使ってみてください。そのほうがきれいに書けます。もしファンクターの中身を、1 つではなく複数の関数を使って写したいなら、そのための関数をトップレベルで宣言してもいいし、ラムダ式を使ってもいいですが、一番いいのは関数合成です。

```
import Data.Char
import Data.List

main = do line <- fmap (intersperse '-' . reverse . map toUpper)
                       getLine
          putStrLn line
```

このコードを走らせて hello there という入力を与えるとこうなります。

```
$ ./fmapping_io
hello there
E-R-E-H-T- -O-L-L-E-H
```

ここで intersperse '-' . reverse . map toUpper という関数は、文字列を取って、各文字を toUpper し、その結果に reverse を適用し、さらに intersperse '-' を適用しています。以下のように書いても同じ意味ですが、関数合成のほうがきれいに書けますね。

```
(\xs -> intersperse '-' (reverse (map toUpper xs)))
```

ファンクターとしての関数

Functor のインスタンスだとは知らずに、何気なくずっと使ってきたものがもう 1 つあります。その Functor は、(->) r です。えっ、(->) r って何のこと？ どういう意味？ 関数の型を表す r -> a は、(->) r a と書き換えることもできるのです。ちょうど 2 + 3 を (+) 2 3 と書き直せるように。むしろ便利でしょう？ 関数の型を (->) r a と表現するとき、関数 (->) には、2 つの型引数を取る型コンストラクタという新しい姿が与えられているのです。 Either と同じです。

ただし、 Functor のインスタンスにする型コンストラクタは引数が 1 つである必要があります。だから、 (->) という形のままでは Functor にはできません。部分適用して (->) r の形にすれば大丈夫です。セクションが使えれば、型コンストラクタの部分適用 (->) r を (r ->) と書きたいところですが、これは文法的にまずいです（関数の部分適用では、例えば + の部分適用を律儀に (+) 2 と書かずにセクションを使って (2+) と書けましたね）。

さて、関数がファンクターであるとはどういうことなのでしょうか？ Control.Monad.Instances にあるインスタンス宣言の実装を覗いてみましょう。

```
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

まず、 fmap の型を思い出してください。

```
fmap :: (a -> b) -> f a -> f b
```

ファンクターのインスタンスは上記の f の場所に入るので、頭の中で f を (->) r に置換してみてください。すると、関数をファンクターのインスタンスにしたとき fmap がどのような型になるかが分かります。

```
fmap :: (a -> b) -> ((->) r a) -> ((->) r b)
```

さらに、 (->) r a と (->) r b という表記を、普通の中置表記 r -> a と r -> b に直しましょう。

```
fmap :: (a -> b) -> (r -> a) -> (r -> b)
```

fmap を使うと、 Maybe は Maybe を生み出し、リストはリストを生み出しました。 Functor としての関数も、関数を取って関数を生み出すはずです。ならば、型 fmap :: (a -> b) -> (r -> a) -> (r -> b) が意味するものとは？ この型は、 a から b への関数と、 r から a への関数を引数に取り、 r から b への関数を返す、と読めます。何か思い出しませんか？ そう！ 関数合成です！

r -> a の出力を a -> b の入力につなぎ、関数 r -> b を作る。これってまさに、関数合成ですね。このインスタンス宣言はこう書いてもかまいません。

```
instance Functor ((->) r) where
    fmap = (.)
```

こう書くと、 fmap が関数ファンクターの場合には関数合成を意味することがはっきりしますね。関数ファンクターのインスタンス定義は Control.Monad.Instances から読み込めるので、試しに使ってみましょう。

```
ghci> :m + Control.Monad.Instances
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (+100) 1
"303"
```

入力の 4 行目では、fmap を中置関数として呼んでいます。こうすると、fmap と . が同じだってことがはっきりしますね。入力の 3 行目では、(*3) で (+100) を写しています。その結果は「入力を (+100) して (*3) する関数」になります。それをさらに 1 に作用させると、答は 303 になります。

他のファンクターと同様、関数ファンクターも文脈を持った値だとみなせます。 Maybe との対比で考えてみましょう。 Maybe ファンクターは、「値があるかもしれない」という文脈を表し、具体化した Maybe a は「 a 型の値があるかもしれない」という文脈でした。それに対し、関数ファンクター (->) r （あるいは r -> ）は、「 r 型の入力を適用すれば結果が返ってくる」という文脈を表し、それを具体化した (->) r a （あるいは r -> a ）は、「 r 型の入力を適用すれば a 型の結果が返ってくる」という文脈を表しています。

例えば関数 (+3) :: Int -> Int は、その関数の返り値である Int 型の値に、「結果が欲しくば Int 型に適用しろ」という文脈が付いたものとみなせます。 fmap (*3) を (+100) に使うと、 (+100) の処理をした後で結果を返す前に (*3) を適用する、新しい関数が生まれます。

fmap を関数に適用すると関数合成になるというのは、今のところあまり役に立ちませんが、興味をひかれる事実です。これについて深く考えると、 IO や (->) r のように箱というよりは計算のように振る舞うものがファンクターになれる理由が分かってきます。「ある関数」を使って「何らかの計算」を写すと、得られるものは「似たような計算」ですが、計算結果はその「関数」で修飾されたものになっているのです。

さて、 fmap が従うルールの説明に移る前に、もう一度 fmap の型について考えておきましょう。

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

第 5 章では、Haskell の関数は実はすべて 1 引数関数とみなせる、というカリー化の概念を導入しました。 a -> b -> c 型の関数は、 a 型の引数を 1 つだけ取る関数で、返り値は b -> c 型の関数です。これがまた 1 つだけ引数を取って、 c 型の値を返します。これを、多引数関数を認める立場から見れば、ある関数を一部の引数しか与えないで呼び出す（部分適用）と「残りの引数を取って最終結果を返す関数」を返す、と説明できます。そこで、a -> b -> c 型は a -> (b -> c) と書けます。こうするとカリー化がはっきり分かりますね。

同じように、 fmap :: (a -> b) -> (f a -> f b) の型を持つ fmap も、関数とファンクター値を取ってファンクター値を返す 2 引数関数とも思えますが、そうじゃなくて、関数を取って「元の関数に似ているけどファンクター値を取ってファンクター値を返す関数」を返す関数だと思うこともできます。 fmap は、関数 a -> b を取って、関数 f a -> f b を返すのです。こういう操作を、関数の**持ち上げ**（lifting）といいます。このようなファンクターの見方について、GHCi の :t コマンドを使って遊びながら覚えましょう。

```
ghci> :t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a
ghci> :t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```

式 fmap (*2) は、数が入っているファンクター f を取って、数が入っているファンクターを返す関数です。ファンクターは、リストでも Maybe でも Either String でも何でもかまいません。また、式 fmap (replicate 3) は、「何でもいい型 a 」が入っているファンクター」を取り、 a のリストが入っているファンクターを返します。このことは、例えば fmap (++"!") のような部分適用を作って、それを GHCi で名前に束縛してみるとよく分かります[†1]。

[†1] ［訳注］Haskell の困った仕様の 1 つ、単相性制限のせいで、引数の型に型クラス制約がある関数は、うまく扱えない場合があります。 NoMonomorphismRestriction オプションを使えばこの制限を解除できます。GHCi では :set -XNoMonomorphismRestriction とします。 .hs ファイルの中で関数を定義しようとしてこのエラーに出くわした場合は、関数に型クラス制約を含む型注釈をきっちり与えるか、あるいはファイルの先頭に {-# LANGUAGE NoMonomorphismRestriction #-} を追加してください。

```
ghci> :set -XNoMonomorphismRestriction
ghci> let shout = fmap (++"!")
ghci> :t shout
shout :: Functor f => f [Char] -> f [Char]
ghci> shout ["ha","ka","ta","no"]
["ha!","ka!","ta!","no!"]
```

というわけで、 fmap については 2 通りの考え方ができます。

- fmap は関数とファンクター値を取って、その関数でファンクター値を写して返すものである。
- fmap は値から値への関数を取って、それをファンクター値からファンクター値への関数に持ち上げたものを返す関数である。

そして、どちらの見方も正しいのです。「関数でファンクター値を写す」ことと、「関数を持ち上げてからファンクター値に適用する」ことは等価です。

例えば、関数 fmap (replicate 3) は、その型が fmap (replicate 3) :: (Functor f) => f a -> f [a] であることから分かるように、どんなファンクターにでも適用できます。 そして、何が起こるかはそのファンクターによって変わります。リストに対して fmap (replicate 3) すれば、リスト向けの fmap の実装が選ばれて、それは普通の map です。 Maybe a に対して fmap (replicate 3) とすれば、 Just の中の値に対して replicate 3 が適用されます。 Nothing なら Nothing です。いくつか例を示します。

```
ghci> fmap (replicate 3) [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]
ghci> fmap (replicate 3) (Just 4)
Just [4,4,4]
ghci> fmap (replicate 3) (Right "blah")
Right ["blah","blah","blah"]
ghci> fmap (replicate 3) Nothing
Nothing
ghci> fmap (replicate 3) (Left "foo")
Left "foo"
```

11.2　ファンクター則

すべてのファンクターの性質や挙動は、ある一定の法則に従うことになっています。 fmap f をファンクターに適用したら、それはファンクターの中身に f を適用するべきであって、それ以上のことをしてはいけないのです。この挙動は**ファンクター則**に記述されています。 Functor のインスタンスは、ファンクター則の 2 つの性質を満たしている必要があります。残念ながら、Haskell には自動的にファンクター則を課してくれるような機能はありませんので、ファンク

ターを自作するときは、ファンクター則を満たしているか、自前でテストしないといけません。標準ライブラリにある Functor たちは、すべてファンクター則を満たしています。

第一法則

ファンクターの第一法則は、「id でファンクター値を写した場合、ファンクター値が変化してはいけない」というものです。式で書くと fmap id = id ってことです。別の言い方をすれば、fmap id をファンクター値に適用した場合、それは id をファンクター値に適用したのと同じ結果になる、ということです。id は恒等写像、引数をそのまま返すだけの関数でしたね。id は、\x -> x とも書けます。ファンクターは中身に関数が適用されるようなものであると思えば、fmap id = id という法則は、何だか自然で当たり前のものに思えますね。

さて、第一法則が満たされているか、いくつかのファンクターで試してみましょう。

```
ghci> fmap id (Just 3)
Just 3
ghci> id (Just 3)
Just 3
ghci> fmap id [1..5]
[1,2,3,4,5]
ghci> id [1..5]
[1,2,3,4,5]
ghci> fmap id []
[]
ghci> fmap id Nothing
Nothing
```

例えば Maybe ファンクターに対する fmap の実装を見てみると、どうやって第一法則が満たされているのかが分かります。

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

さて、f に id を代入したところをイメージしてみてください。id で Maybe ファンクター値を写すとき、Maybe 値の中身は Just 値コンストラクタである場合と Nothing 値コンストラクタである場合があり得ます。

まず Just の場合、fmap id を Just x に適用すると、Just (id x) ができます。そして id は恒等写像ですから、Just (id x) は Just x です。というわけで、Just の場合は、確かに fmap id したら元と同じ値が返ってくることが分かりました。

次に、 id で Nothing を写した場合は、同じ値が返ってくることは明らかです。というわけで、 Maybe ファンクターの fmap を実装する 2 つの等式から、Maybe ファンクターは確かにファンクター第一法則 fmap id = id を満たすことが分かりました。

第二法則

第二法則は、関数合成と写す操作との間の関係です。第二法則は、2 つの関数 f と g について、「f と g の合成関数でファンクター値を写したもの」と、「まず g、次に f でファンクター値を写したもの」が等しいことを要求します。式で書くと、 fmap (f . g) = fmap f . fmap g ってことです。別の言い方をすると、すべてのファンクター値 x に対して fmap (f . g) x = fmap f (fmap g x) が成り立つべし、というのが第二法則です。

ある型が 2 つのファンクター則を満たすということは、適用に関する基本的な振る舞いが関数や他のファンクターと一致することが保証される、ということです。その型に fmap を使ったときに、関数によって写される以外の余計なことが裏で発生することはなく、あくまでも関数の素直な拡張であるファンクターとして振る舞うことが分かるのです。

ある型がファンクター第二法則を満たすかどうかは、その型の fmap の実装を見て、それから Maybe が第一法則を満たしているか調べたときと同じ方法を使えば分かります。 Maybe ファンクターが第二法則を満たしているか調べてみましょう。まず、 Nothing に対して fmap (f . g) を使うと、返り値は Nothing になります。なにしろ、 Nothing に対して何を fmap しても Nothing ですから。同じ理由から、fmap f (fmap g Nothing) を呼び出しても、結果はやはり Nothing になります。

Maybe が第二法則を満たしているか調べるのは、 Nothing 値の場合にはとても簡単でした。では、 Just 値の場合にはどうでしょう？ はい、まず fmap (f . g) (Just x) を評価してみます。これは実装から Just ((f . g) x) になりますが、これは Just (f (g x)) です。次に fmap f (fmap g (Just x)) については、これまた実装に当てはめてみると、はじめに fmap g (Just x)

の部分が評価されて Just (g x) になりますから、 fmap f (fmap g (Just x)) は fmap f (Just (g x)) になります。さらに残りを評価すると、これは Just (f (g x)) に等しいことが分かります。

この証明は難しかったかもしれませんが、心配いりません。関数合成やファンクター合成の感覚さえつかめれば大丈夫です。ある型が一種のコンテナや関数として振る舞うことからファンクター則を満たすことが直感的に明らか、という場合もよくあります。また、その型のさまざまな値でファンクター則を実験してみて、ああ、確かにファンクター則が成り立っているようだな、と納得することもできます。

法則を破る

ここで、 Functor のインスタンスなのに、ファンクター則を満たしていないような病的な例を考えてみましょう。以下のような型を用意します。

```
data CMaybe a = CNothing | CJust Int a deriving (Show)
```

C は「counter」のつもりです。 CMaybe a は、 Maybe a によく似たデータ型ですが、 Just 部分のフィールドが 1 つではなく 2 つあります。 CJust の 1 つ目のフィールドは常に Int 型で、これが何らかのカウンタになります。2 つ目のフィールドの型 a は型引数で、 a の型が何になるかは CMaybe a をどんな具体型にしたいかによって決まります。この新しい型で遊んでみましょう。

```
ghci> CNothing
CNothing
ghci> CJust 0 "haha"
CJust 0 "haha"
ghci> :t CNothing
CNothing :: CMaybe a
ghci> :t CJust 0 "haha"
CJust 0 "haha" :: CMaybe [Char]
ghci> CJust 100 [1,2,3]
CJust 100 [1,2,3]
```

CNothing コンストラクタにはフィールドがありません。 CJust コンストラクタには 2 つのフィールドがあり、最初のフィールドには整数を、2 つ目のフィールドには任意の型の値を入れられます。では、この CMaybe を Functor のインスタンスにしてみましょう。 fmap を使うたびに 2 つ目のフィールドに関数を適用しますが、その際にカウンタを増やす（つまり最初のフィールドに 1 を足す）という実装にします。

```
instance Functor CMaybe where
    fmap f CNothing = CNothing
    fmap f (CJust counter x) = CJust (counter+1) (f x)
```

Maybe のインスタンス実装に似ていますが、 fmap を空でない箱（ CJust 値）に使ったときは中身に関数を適用するだけでなくカウンタに 1 を足す、というところが違います。別に何の問題もなさそうですよね。では、使ってみましょう。

```
ghci> fmap (++"ha") (CJust 0 "ho")
CJust 1 "hoha"
ghci> fmap (++"he") (fmap (++"ha") (CJust 0 "ho"))
CJust 2 "hohahe"
ghci> fmap (++"blah") CNothing
CNothing
```

さて、 CMaybe はファンクター則を満たしているでしょうか？ 満たしていないことを証明するには 1 つでも反例を挙げればよいので、とても簡単です。

```
ghci> fmap id (CJust 0 "haha")
CJust 1 "haha"
ghci> id (CJust 0 "haha")
CJust 0 "haha"
```

ファンクター第一法則によれば、 id でファンクター値を写した結果と、単に id をファンクター値に適用した結果とは等しくなければならないのでした。上記の例から分かるとおり、 CMaybe はこの性質を満たしていません。ですので、 CMaybe は Functor のインスタンスを自称してはいるものの、ファンクターではないことになります。

CMaybe は、 Functor のインスタンスでありながらファンクター則を満たさないため、 CMaybe をファンクターとして利用するコードはバグを生む可能性があります。ファンクターを利用するとき、複数の関数を合成してから写すのと、複数の関数で次々に写すのとは、同じ結果を生まないといけないのです。しかし CMaybe の場合は、ファンクターに適用された回数を記録しているので、合成と写しの順番を入れ替えると結果が変わってしまいます。これはまずい！ CMaybe にファンクター則を満たさせようと思ったら、 fmap を使ったときに Int フィールドをいじってはいけないのです。

最初はファンクター則なんて意味が分からないし、不必要と思うかもしれません。でも、もしある型がファンクター則を両方とも満たすと分かれば、その型の挙動についてある種の信頼がおけます。 fmap を使っても、その関数でファンクター値の「中身」が写されるだけで、それ以外のことは何も起こらないと確信できるのです。この前提のおかげで、より抽象的で応用の利くコードが書けます。すべてのファンクターが満たしているはずの法則を根拠に、どんなファンクターに適用しても間違いなく動作する関数を作れるからです。

ある型を Functor のインスタンスにしようと思ったら、ちょっと時間を割いて、その実装がファンクター則を満たしているか確認してください。実装を 1 行ごとに追って法則が満たされているか確認したり、反例はないか探してみてくだ

さい。多種多様なファンクターを扱ってファンクター経験値をためれば、それらに共通する性質や振る舞いを認識できるようになって、ある型がファンクター則を満たすかどうか直感的に分かるようになるでしょう。

11.3 アプリカティブファンクターを使おう

この節では、ファンクターの強化版であるアプリカティブファンクターを紹介します。

ここまではファンクター値を写すために、もっぱら 1 引数関数を使ってきました。では、2 引数関数でファンクターを写すと何が起こるのでしょう？

例えば、`Just 3` に対して `fmap (*) (Just 3)` とすると何が起こるでしょうか？ `Maybe` の `Functor` インスタンスの実装を見ると、`fmap` の引数の関数が `Just` の中身に適用されることが分かります。ですから `fmap (*) (Just 3)` は `Just ((*) 3)` になります。これはセクション記法を使うと `Just (3 *)` とも書けます。なんと！ `Just` の中に関数が入ってしまいました！

関数がファンクター値の中に入っている例はもっとあります。

```
ghci> :t fmap (++) (Just "hey")
fmap (++) (Just "hey") :: Maybe ([Char] -> [Char])
ghci> :t fmap compare (Just 'a')
fmap compare (Just 'a') :: Maybe (Char -> Ordering)
ghci> :t fmap compare "A LIST OF CHARS"
fmap compare "A LIST OF CHARS" :: [Char -> Ordering]
ghci> :t fmap (\x y z -> x + y / z) [3,4,5,6]
fmap (\x y z -> x + y / z) [3,4,5,6] :: (Fractional a) =>
[a -> a -> a]
```

例えば、`(Ord a) => a -> a -> Ordering` の型を持つ関数 `compare` を、文字列のリストに対して `fmap` すると、`Char -> Ordering` のリストが返ってきます。`compare` が入力リストの文字列の各々に対して部分適用されたからです。返り値が `(Ord a) => a -> Ordering` のリストにならなかったのは、`compare` に第一引数が部分適用された時点で、`a` が `Char` 型に決まるので、第二引数の `a` も `Char` に限定されるからです。

このように、「多引数」関数でファンクター値を写すと、関数が入ったファンクター値が返ってきます。これって何に使えるんでしょう？ 1 つの使い道は、そ

の中身の関数を引数に取れるような型を持つ関数を fmap することです。なにしろファンクター値の中身は、ファンクターを写そうとする関数にもれなく渡されるのですから。

```
ghci> let a = fmap (*) [1,2,3,4]
ghci> :t a
a :: [Integer -> Integer]
ghci> fmap (\f -> f 9) a
[9,18,27,36]
ghci> fmap ($9) a
[9,18,27,36]
```

では、ファンクター値 Just (3 *) とファンクター値 Just 5 があったとして、Just (3 *) から関数を取り出して Just 5 の中身に適用したくなったとしたらどうしましょう？ 実は、普通のファンクターを扱う限り、これは無理なんです。普通のファンクターでできるのは、「通常の関数で」「ファンクターの中の値を」写すことだけだからです。 \f -> f 9 を fmap したときの例では、ファンクターでもあるリストの中の関数に引数を渡せてるように見えたかもしれませんが、あれはあくまでも、「 \f -> f 9 という通常の関数でファンクター値を写したところ、たまたま関数を取って引数を渡す機能を備えていた」というだけです。「ファンクターの中の関数」で「別のファンクターの中の値」を写すことは、fmap の提供する機能では無理です。確かに、 Just コンストラクタをパターンマッチして中の関数を取り出し、それを Just 5 に fmap する、という手はあります。でも、それは Maybe 限定の手段になってしまいますね。もっと、どんなファンクターにも使えるような一般的で抽象的なアプローチはないものでしょうか？

Applicative ちゃんと仲良くしてあげてね！

Control.Applicative モジュールにある型クラス Applicative に会いに行きましょう！ 型クラス Applicative は、2 つの関数 pure と <*> を定義しています。どちらもデフォルト実装は与えられていないので、ある型を Applicative のインスタンスにしたかったら両方の定義を与える必要があります。クラス定義はこんな感じです。

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

このたった 3 行の短いクラス定義から、実にたくさんのことが分かります！ 1 行目では、 Applicative クラスの定義を与えると同時に、型クラス制約を導入しています。「ある型コンストラクタを Applicative 型クラスに属させたかっ

たら、まずは Functor に属させるべし」という型クラス制約です。ですから、ある型コンストラクタが Applicative 型クラスに属していたら、それは必ず Functor でもあるので、常に fmap が使えます。

Applicative 型クラスの定義する最初のメソッドは pure と呼ばれます。その型宣言は pure :: a -> f a です。この f がアプリカティブファンクターになるものです。Haskell の型システムはとてもよくできている上に、どんな関数も引数を取って何か値を返す以外のことはできないので、型宣言からは実に多くのことが読み取れます。今回も例外ではありません。

pure は任意の型の引数を受け取り、それをアプリカティブ値の中に入れて返します。「中に入れて」という言い方はファンクターのときに使った箱の比喩を踏襲したものです。すでに見たように、この比喩が当てはまらない場合もありますが、型宣言 a -> f a を文字どおりに解釈すれば自明でしょう。すなわち pure は、値を取って、内部にその値を結果として含むアプリカティブ値にくるみます。アプリカティブ値は「箱」というよりも「文脈」と考えるほうが正確かもしれません。 pure は、値を引数に取り、その値を何らかのデフォルトの文脈（元の値を再現できるような最小限の純粋な文脈）に置くのです。

<*> 関数はすごく面白いですね。型定義はこうなっています。

```
f (a -> b) -> f a -> f b
```

これを見て何か思い出しませんか？ そう、これって fmap :: (a -> b) -> f a -> f b にそっくり！ <*> は、 fmap の強化版なのです。 fmap が普通の関数とファンクター値を引数に取って、関数をファンクター値の中の値に適用してくれるのに対し、 <*> は関数の入っているファンクター値と値の入っているファンクター値を引数に取って、1 つ目のファンクターの中身である関数を 2 つ目のファンクターの中身に適用するのです。

Maybe はアプリカティブファンクター

手始めに Maybe の Applicative インスタンスを見てみましょう。

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

ファンクターのときと同じく、アプリカティブファンクターになる型コンストラクタ f も、具体型を 1 つだけ型引数に取ることがクラス定義から分かります。なので、インスタンス宣言の書き出しは **instance** Applicative Maybe **where** とします。 **instance** Applicative (Maybe a) **where** ではダメですよ。

次の行は pure の定義です。 pure は何かを取ってそれをアプリカティブ値に包むのでしたね。 pure の定義には、ただ pure = Just とだけ書いてあります。 Just のような型コンストラクタは普通の関数でもありますから、これでいいんです。 pure x = Just x と書いてもかまいません。

残りは <*> の定義です。まず、 Nothing からは関数が取り出せません。なにしろ無ですから。そこで、 Nothing から関数を取り出そうとした場合は結果も Nothing 、ということにしましょう。

Applicative のクラス定義には Functor の型クラス制約が付いていましたから、 <*> 関数の 2 つの引数は両方ともファンクター値であるはずです。もし最初の引数が Nothing でなくて Just なら、その中身の関数を取り出して 2 つ目の引数に fmap することにします。こうすれば、2 つ目の引数が Nothing だった場合にも自動的に対応できます。 Nothing を fmap しても Nothing だからです。というわけで、Maybe ファンクターに関する <*> の挙動は、左辺が Just ならその中の値を取り出して右辺を写す、というものです。もし左右どちらかの引数が Nothing だったら答も Nothing になります。

さあ、試してみましょう。

```
ghci> Just (+3) <*> Just 9
Just 12
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just (++"hahah") <*> Nothing
Nothing
ghci> Nothing <*> Just "woot"
Nothing
```

見てのとおり、 pure （+3）と Just （+3）の効果はまったく同じです。 pure を使うのは、 Maybe をアプリカティブとして（ <*> と組み合わせて）使う場合だけに止めておいたほうが無難でしょう。そのほかの場合は素直に Just を使いましょう。

最初の 4 行は、アプリカティブから関数を取り出して適用するデモです。でも、こういう場合はわざわざアプリカティブにくるまなくても、生の関数を fmap すれば済みます。ところが最後の行はちょっと変わっています。 Nothing から関数を取り出して Just に適用しようとした結果、 Nothing を得ています。

普通のファンクターの場合、いったん関数で写してしまったら、ファンクターの中に入ってしまった関数適用の結果をファンクターの外に取り出す一般的な方法はありません。たとえその結果が部分適用された関数であったとしてもで

す。一方、アプリカティブファンクターなら、1 つの関数で複数のファンクターを続けざまに写せるのです！

アプリカティブ・スタイル

Applicative 型クラスでは、 <*> を連続して使うことができ、1 つだけでなく複数のアプリカティブ値を組み合わせて使うことができます。例えばこれを見てください。

```
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```

ここでは + 関数をアプリカティブ値の中に入れ、さらに <*> を使って 2 つの引数に適用していますが、そのどちらもアプリカティブ値です。

一体どうしてこれが可能になったのか、順を追って見ていきましょう。 <*> は左結合なので、

```
pure (+) <*> Just 3 <*> Just 5
```

この式は、

```
(pure (+) <*> Just 3) <*> Just 5
```

これと同じです。

まず、+ 関数をアプリカティブ値（今回は Maybe 値）の中に入れたいので、pure (+) を使います。これは Just (+) のことです。次に Just (+) <*> Just 3 が評価されます。その結果は Just (3+) になります。ここでは部分適用が起こっています。 + という二項演算の関数に 3 という 1 つの引数だけを与えると、「もう 1 つの引数を取ってそれに 3 を足す関数」が返ってきます。最後に Just (3+) <*> Just 5 が実行され、結果は Just 8 になります。

これってすごくない？ アプリカティブファンクターとアプリカティブ・スタイル pure f <*> x <*> y <*> ... を使えば、もともとアプリカティブなんて知らずに書かれた普通の関数にもアプリカティブ値の引数を与えることができます。しかも、 <*> が出てくるたびに部分適用が働くので、いくつでも引数を与えることができるのです。

アプリカティブ・スタイルをもっと理解して便利に使うためのポイント、それは、 pure f <*> x は fmap f x と等しい、という事実を考慮に入れることです。ちなみに、これはアプリカティブ則の 1 つです。アプリカティブ則について

はこの章の後半で詳しく見るとして、今はこの法則にどういう効果があるかを考えましょう。pure は値をデフォルトの文脈の中に入れます。ある関数をデフォルトの文脈の中に入れ、また取り出して別のアプリカティブファンクターの中の値に適用する、という操作は、単に元の関数でアプリカティブファンクターを写すのと同じことです。そこで、pure f <*> x <*> y <*> ... と書く代わりに、fmap f x <*> y <*> ... と書くことができます。このパターンはしょっちゅう使うので、Control.Applicative は fmap と等価な中置演算子 <$> をエクスポートしています。<$> の定義はこれです。

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

NOTE　f が 2 回登場しますが、1 行目と 2 行目の f は別物です。型変数の働きは引数の名前やそのほかの値の名前に依存しないことを思い出してください。1 行目の関数宣言に登場するのは型変数 f で、これには型クラス制約が付いていて、f に入る型コンストラクタは Functor 型クラスに属している必要があると言っています。2 行目の f は関数の本体で、x を写すのに使うものです。両方とも f という名前がついていますが、だからといって同じものではありませんよ。

<$> を使うとアプリカティブ・スタイルの素晴らしさが引き立ちます。例えば、関数 f を 3 つのアプリカティブ値の引数に適用したければ、f <$> x <*> y <*> z と書けるのです。引数がアプリカティブ値でなく普通の値だったら f x y z と書くところです。

この書き方でなぜうまくいくのか追ってみましょう。例えば、値 Just "johntra" と Just "volta" を結合して 1 つの Maybe String 型の値を作りたいとします。それはこう書けます。

```
ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

どうしてこうなるのか確かめる前に、上の例を次と比べてみてください。

```
ghci> (++) "johntra" "volta"
"johntravolta"
```

普通の関数をアプリカティブファンクターの関数として使うには、<$> と <*> をちりばめるだけです。すると関数は、アプリカティブ値を取ってアプリカティブ値を返すようになります。便利すぎるでしょう？

(++) <$> Just "johntra" <*> Just "volta" の例に戻りましょう。まずは (++) :: [a] -> [a] -> [a] の型を持つ関数 (++) が Just "johntra" を写します。できた値は Just ("johntra"++) と等価なもので、型は Maybe ([Char] -> [Char]) です。(++) の最初の引数が消費され、a は Char に変わりましたね。さて次は Just ("johntra"++) <*> Just "volta" の評価が発

生します。すると Just から関数が出てきて Just "volta" を写し、最終結果 Just "johntravolta" を生み出します。もし、いずれかの引数が Nothing だったとしたら、答は Nothing になります。

ここまで Maybe の例しか出てきてませんが、もしかしてアプリカティブファンクターのことを Maybe 専用だなんて思っていませんか？ Applicative のインスタンスはいっぱいあるんですよ！

リスト

リスト（正確に言えばリスト型コンストラクタ [] ）もアプリカティブファンクターです。意外ですか？ [] の Applicative インスタンス宣言はこうです。

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

pure は値を取ってデフォルトの文脈に入れるものでした。ここで言うデフォルトの文脈とは、なるべく小さな、それでいて引数を再現できるような最小限の文脈のことです。さて、リストの文脈における最小のものは空リストです。しかし、空リストは、値がないことを表しているのですから、 pure の引数に渡された値を持っておくことができません。だから、 pure は引数を単一要素のリストに入れて返すのです。これは Maybe アプリカティブファンクターのときも同じでしたね。 Maybe の最小の文脈は Nothing でしたが、 Nothing は値がないことを表しているので、 pure は Just を使って実装されていたのでした。

これが pure の動作です。

```
ghci> pure "Hey" :: [String]
["Hey"]
ghci> pure "Hey" :: Maybe String
Just "Hey"
```

<*> についてはどうでしょう？ 関数 <*> の型をリストに制限すると、 (<*>) :: [a -> b] -> [a] -> [b] になります。 <*> はリスト内包表記で実装されています。二項演算子 <*> は、何らかの方法で左辺から取り出した関数を使い、右辺を写す必要があります。ところが、左辺のリストに入っている関数は 0 個かもしれないし、1 個かもしれないし、複数個かもしれません。もちろん右辺のリストにも複数の値が格納できます。そこで、両方のリストから要素を取り出すためにリスト内包表記を使っています。 <*> は左辺のリストのそれぞれの関数を、右辺のリストのそれぞれの値に適用します。結果、 <*> が返すリストには、左辺のリストの中の関数を右辺のリストの中の値にあらゆる可能な組み合わせで適用したものが入ります。

リストの <*> を使う例です。

```
ghci> [(*0),(+100),(^2)] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]
```

左辺のリストは 3 つの関数を含んでおり、右辺リストは 3 つの値を含んでいます。そこで結果のリストのサイズは 9 になるわけです。左辺のリストのそれぞれの関数が、右辺の値のそれぞれに適用されます。

もし 2 引数関数のリストがあれば、その関数を 2 つのリストに適用できます。2 つの関数を 2 つのリストに適用する例です。

```
ghci> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

<*> は左結合なので、まず [(+),(*)] <*> [1,2] が処理されます。それぞれの関数がそれぞれの値に適用されるので、[(1+),(2+),(1*),(2*)] と等価なリストができます。次に [(1+),(2+),(1*),(2*)] <*> [3,4] が処理されて、最終結果が出てきます。

アプリカティブ・スタイルをリストで使うと楽しめますよ！

```
ghci> (++) <$> ["ha","heh","hmm"] <*> ["?","!","."]
["ha?","ha!","ha.","heh?","heh!","heh.","hmm?","hmm!","hmm."]
```

適切なアプリカティブ演算子と組み合わせるだけで、2 つの文字列を引数に取る関数を 2 つの文字列のリストに使いました。

リストは非決定性計算とみなすことができます。100 や "what" のような値は答が 1 つしかない決定性計算であるのに対し、[1,2,3] のようなリストは「どの答がいいか決められないので可能性のある答をすべて提示している」とみなせます。この見方に立てば、(+) <$> [1,2,3] <*> [4,5,6] というコードは、2 つの非決定性計算を + で足し算して、いっそう自信のない新たな非決定性計算の結果が生じた、と解釈できます。

アプリカティブ・スタイルをリストに使うと、リスト内包表記をうまく置き換えられることが多々あります。第 1 章には、[2,5,10] と [8,10,11] の積として可能な値をすべて求めたい、という例題がありました。そのときは次のようにして、

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

2 つのリストから値を取り出しては、可能な組み合わせすべてに対して掛け算関数を適用したのでした。これはアプリカティブ・スタイルではこう書けます。

```
ghci> (*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```

個人的にはこちらの記法のほうがきれいだと思います。2 つの非決定性計算に対して * を呼び出しているだけだ、と考えるほうが、リスト内包表記より分かりやすいですからね。2 つのリストの要素を掛け合わせて作れる数のうち 50 より大きなものをすべて知りたければ、このように書けます。

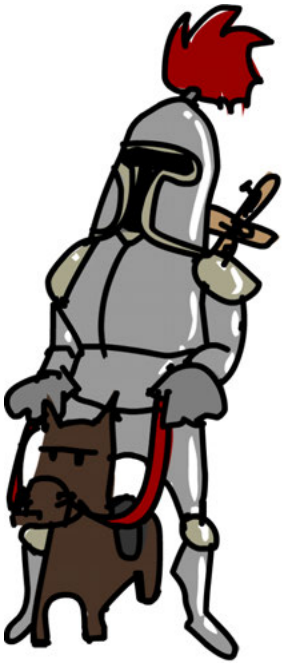
```
ghci> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

リストの場合、pure f <*> xs と fmap f xs が一致する理由は簡単です。まず、pure f は [f] です。[f] <*> xs は、左辺のリストにあるすべての関数を右辺のリストのすべての値に適用しようとしますが、左辺のリストの中には関数が 1 個しかないので、[f] <*> xs は map のような動作をするわけです。

IO もアプリカティブファンクターだよ！

今まで見てきたものの中に、ほかにも Applicative なものがあります。ずばりそれは IO です。

```
instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```



pure というのは、要するに、再現できるような形で最小の文脈に値を入れるという意味です。だから pure が単なる return なのは納得がいきますね。return は何もしない I/O アクションを作ります。ただ値を返すだけで、ターミナルに印字したりファイルを読み込んだりといった入出力操作は一切しません。

もし <*> が IO に特殊化されたら、その型は (<*>) :: IO (a -> b) -> IO a -> IO b になるはずです。IO の場合の a <*> b の実装は、まず a という名の I/O アクションを実行して関数を手に入れて、その関数を f という名前に束縛します。次に b を実行してその結果を x という名前に束縛します。最後に、関数 f を x に適用し、それを結果として返しています。実装には do 記法が使われています（do 構文はいくつかの I/O アクションを 1 つに結合するのでしたね）。

Maybe と [] に関しては、<*> 演算子は単に左辺の引数から関数を取り出しては右辺を写す操作だと解釈できました。IO に関する <*> 演算子は、取り出すところは同じですが、2 つの I/O アクションを 1 つに糊付けするにあたって**逐次実行**という意味が新たに加わっています。<*> 演算子は、まず 1 つ目の I/O アクションから関数を取り出しますが、結果を取り出したかったらその I/O を実行する必要があるというわけです。これを見てください。

```
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b
```

これはユーザに 2 行の入力を求め、その 2 行を結合したものを返す I/O アクションです。2 つの getLine I/O アクションと return を糊付けして作りました。I/O 全体が a ++ b という値を持ってほしかったからです。このプログラムは次のようにアプリカティブ・スタイルでも書けます。

```
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

このコードは、「2 つの I/O アクションの結果に関数を適用して返す I/O アクション」として実装したさっきのコードと同じものです。getLine は getLine :: IO String の型を持つ I/O アクションです。2 つのアプリカティブ値を <*> 演算子で結合したら、結果もアプリカティブ値になるので、すべて理にかなっています。

箱の比喩で言えば、getLine とは実世界に出かけていって文字列を取ってきてくれる小さな箱だと思えます。(++) <$> getLine <*> getLine を呼び出すと、この小さな箱を 2 つ送り出して端末から入力行を取得し、結合して返してくれる大きな箱が作られます。

式 (++) <$> getLine <*> getLine の型は IO String です。つまりこの式は、String を返す他の I/O アクションとまったく同様、ごく普通の I/O アクションなのです。だからこそ、こういう書き方もできるわけです。

```
main = do
    a <- (++) <$> getLine <*> getLine
    putStrLn $ "The two lines concatenated turn out to be: " ++ a
```

関数もアプリカティブだよ

Applicative のインスタンスはまだあります。(->) r、つまり関数がそうです。関数をアプリカティブとして使う機会はあまり多くありませんが、概念自体はとても面白いので、アプリカティブな関数のインスタンスがどのように実装されているか見てみましょう。

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

ある値を、pure を使ってアプリカティブ値に包んだものは、元の値を生み出せないといけません。つまり、その値を再現できるような最小限のデフォルト文脈でなければなりません。そこで、関数のアプリカティブなインスタンスの実装における pure は、値を取って「引数を無視して常にその値を返す関数」を作るようになっています。(->) r インスタンスに特殊化した pure の型は pure :: a -> (r -> a) です。

```
ghci> (pure 3) "blah"
3
```

カリー化のおかげで関数適用は左結合になっているので、この括弧は省けます。

```
ghci> pure 3 "blah"
3
```

<*> のインスタンス実装は少々暗号じみているので、関数をアプリカティブ・スタイルで使う方法から見ていきましょう。

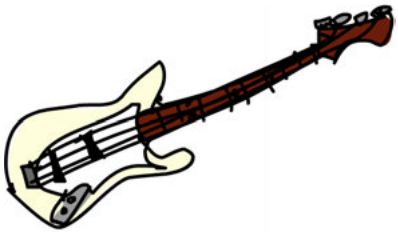
```
ghci> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
ghci> (+) <$> (+3) <*> (*100) $ 5
508
```

<*> を 2 つのアプリカティブ値に対して呼び出した結果はアプリカティブ値です。したがって、<*> を 2 つの関数に使ったら関数が返ってきます。では、このコードでは何が起こっているのでしょう？ (+) <$> (+3) <*> (*100) と書くと、「引数を (+3) と (*100) に渡し、2 つの結果に対して + を使う」関数が出来上がります。この関数に引数 5 を与え、(+) <$> (+3) <*> (*100) $ 5 という形で呼ぶと、まず (+3) と (*100) が 5 に適用され、それぞれ 8 と 500 を返します。それから + が引数 8 と 500 を取って呼ばれ、508 を生み出します。

次のコードも似たようなものです。

```
ghci> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
[8.0,10.0,2.5]
```

ここで作ったのは、3 つの関数(+3)、(*2)、(/2) からの結果をもって関数 \x y z -> [x,y,z] を呼び出す関数です。最後の引数 5 は、まず 3 つの関数にそれぞれ入り、出てきた返り値を引数にして \x y z -> [x,y,z] が呼び出されています。



NOTE　Applicative の (->) r インスタンスの仕組みを理解することは、そこまで重要ではありません。だから完全に分からなくても気にしない。アプリカティブ・スタイルと関数で遊んでみて、関数をアプリカティブとして使えることが分かれば十分です。

Zip リスト

実は、リストをアプリカティブファンクターにする方法は複数あるんです。1 つ目の方法はもうご紹介しました。 <*> を関数のリストと値のリストに対して呼び出して、左辺のリストからの関数と右辺のリストからの値のあらゆる可能な組み合わせを作り、適用したリストを返すというものです。

例えば、 [(+3),(*2)] <*> [1,2] というふうに書いたら、 (+3) は 1 と 2 の両方に適用され、 (*2) も 1 と 2 の両方に適用されるので、結果として 4 つの要素からなるリスト [4,5,2,4] ができます。ところが、[(+3),(*2)] <*> [1,2] の挙動は、左辺の 1 つ目の関数を右辺の 1 つ目の値に適用し、左辺の 2 つ目の関数を右辺の 2 つ目の値に適用する、というのではまずいのでしょうか？ それなら結果は [4,4] になるはずです。これは [1 + 3, 2 * 2] と考えることもできます。

ここで初めて登場する Applicative のインスタンスが ZipList です (Control.Applicative モジュールにあります)。

1 つの型に対して、同じ型クラスのインスタンスを複数回宣言することはできないので、 [] の代わりに ZipList a 型を導入します。これは、1 つのコンストラクタ(ZipList)と、ただ 1 つのフィールド (a のリスト) を持ちます。これがインスタンス宣言です。

```
instance Applicative ZipList where
        pure x = ZipList (repeat x)
        ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

<*> は、1 つ目の関数を 1 つ目の値に、2 つ目の関数を 2 つ目の値に、……と適用します。これをやっているのが zipWith (\f x -> f x) fs xs です。

zipWith の仕様上、出来上がるリストは 2 つのリストのうちの短いほうの長さになります。

pure の定義も面白いですね。 ZipList の pure は、引数を取って、それを永遠に繰り返すリストに突っ込みます。例えば pure "nya" は ZipList (["nya","nya","nya"... を返します。これはおかしいと思うかもしれません。 pure は引数を再現できる最小の文脈に入れるものなのに、無限リストはとても最小の文脈には見えませんよね。しかし ZipList に限っては、 pure はリストのあらゆる位置で値を再現してほしいので、この定義でよいのです。それにこの定義なら、 pure f <*> xs と fmap f xs は等価であれ、という法則を満たします。もし pure 3 がただの ZipList [3] を返す仕様だったとしたら、pure (*2) <*> ZipList [1,5,10] の結果は ZipList [2] になってしまいます。2 つのリストを zip した結果の長さは短いほうの長さになるからです。一方、無限リストと有限リストを zip した結果のリストの長さは、有限リストのほうの長さになります。

さて、 ZipList をアプリカティブ・スタイルで試してみましょう！ と言いたいところですが、 ZipList a 型は Show インスタンスをサポートしていません。getZipList 関数を使って生リストを取り出す必要があります。

```
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100,100]
[101,102,103]
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]
[101,102,103]
ghci> getZipList $ max <$> ZipList [1,2,3,4,5,3] <*> ZipList [5,3,1,2]
[5,3,3,4]
ghci> getZipList $ (,,) <$> ZipList "dog" <*> ZipList "cat" ⇨
        <*> ZipList "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

NOTE 関数 (,,) は \x y z -> (x,y,z) と同じです。同様に、 (,) と書いたら、それは \x y -> (x,y) という意味です。

zipWith 以外にも、標準ライブラリは zipWith3 、 zipWith4 、…、 zipWith7 までの関数を備えています。 zipWith は 2 引数関数を取って、2 つのリストをそれで綴じ合わせます。 zipWith3 は 3 引数関数を取って、3 つのリストを綴じ合わせます。以下同様。ところが、 ZipList をアプリカティブ・スタイルで使えば、綴じ合わせたいリストの数ごとに zip 関数を使い分ける必要はなくなります。アプリカティブ・スタイルを使えば、好きな数のリストを 1 つの関数で綴じ合わせることができます。便利でしょう？

アプリカティブ則

普通のファンクターのように、アプリカティブファンクターにもいくつかの法則が付いてきます。中でも一番重要なのは `pure f <*> x = fmap f x` です。この章で出会ったいくつかのファンクターに対してこの法則を証明するのは演習問題とします。これ以外のアプリカティブ則の一覧はこちらです。

- `pure id <*> v = v`
- `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- `pure f <*> pure x = pure (f x)`
- `u <*> pure y = pure ($ y) <*> u`

これら法則の詳細について説明しだすと長くなって退屈なので、ここでは省略します。興味のある方は、アプリカティブのインスタンスたちがこの法則をどのように満たしているか実験してみてください。

11.4　アプリカティブの便利な関数

`Control.Applicative` には `liftA2` という、以下のような型を持つ関数があります。

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
```

`liftA2` の定義はこうです。

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

`liftA2` は、これまで見てきたアプリカティブ・スタイルを省略しつつ、1 つの関数を 2 つのアプリカティブ値に適用するだけの関数です。しかし、アプリカティブファンクターが通常のファンクターに勝っている点を明瞭に示す例になっています。

通常のファンクターでは、関数を 1 つのファンクター値に適用することしかできません。アプリカティブファンクターなら、関数をいくつものファンクター値に適用できます。`liftA2` の型を `(a -> b -> c) -> (f a -> f b -> f c)` と解釈すると、これがまた面白い。この見方をすれば、`liftA2` は「通常の 2 引数関数を、2 つのアプリカティブ値を引数に取る関数に昇格させる」関数だとみなすことができます。

ここで 1 つ、面白い概念を紹介しましょう。2 つのアプリカティブ値から、それらの返り値をリストとして内包する 1 つのアプリカティブ値を組み立てるこ

とができます。例えば Just 3 と Just 4 があるとします。まずは 2 つ目の中身の 4 を単一要素リストに入れましょう。これはとっても簡単です。

```
ghci> fmap (\x -> [x]) (Just 4)
Just [4]
```

よし。今、手札には Just 3 と Just [4] があります。ここから Just [3,4] を作るにはどうしたら？ これも簡単です。

```
ghci> liftA2 (:) (Just 3) (Just [4])
Just [3,4]
ghci> (:) <$> Just 3 <*> Just [4]
Just [3,4]
```

: は、ある要素とあるリストを取って、その要素を先頭に付けた新しいリストを作る関数でしたね。さて、Just [3,4] はできました。次にこれを Just 2 と組み合わせて Just [2,3,4] を作ることはできるでしょうか？ もちろんです。どうやら、好きな数のアプリカティブ値たちから、それらの返り値をリストにしたものを持つ単一のアプリカティブ値を組み立てることは、常に可能なようです。

では、「アプリカティブ値のリスト」を取って「リストを返り値として持つ 1 つのアプリカティブ値」を返す関数を実装してみましょう。これを sequenceA と呼ぶことにします。

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

おお、再帰だ！ まずは型を見てみましょう。この関数は、アプリカティブ値のリストを、リストを持っているアプリカティブ値に変えます。この型からだけでも、再帰の基底部を書くことができます。空リストをリストを持っているアプリカティブ値にしたかったら、単にデフォルトの文脈に入れるだけです。次は再帰部です。リストの head と tail が与えられたなら（x は単一のアプリカティブ値で、xs はアプリカティブ値のリストですね）、まず sequenceA を tail に対して呼んでリストの入ったアプリカティブ値を返してもらい、pure (:) を使ってアプリカティブ値 x の中身をそのリストに結合すればいいのです。ね、簡単でしょ！

次のように実行したとしましょう。

```
sequenceA [Just 1, Just 2]
```

定義により、これはこう書くのと同値です。

```
(:) <$> Just 1 <*> sequenceA [Just 2]
```

さらに評価を進めると、こうなります。

```
(:) <$> Just 1 <*> ((:) <$> Just 2 <*> sequenceA [])
```

ここで、 sequenceA [] は Just [] になるので、

```
(:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])
```

こうなって、

```
(:) <$> Just 1 <*> Just [2]
```

Just [1,2] ができました！

sequenceA は、畳み込みを使っても実装できます。リストの要素を走査しながら何らかの結果を集計するという操作は、たいてい畳み込みを使って実装できることを覚えておいてください。

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

リストを右から処理し、アキュムレータの初期値は pure [] にしておきます。次に liftA2 (:) がアキュムレータとリストの最後の要素に適用され、単一要素リストが入ったアプリカティブ値ができます。続いて liftA2 (:) が、今の最終要素（元リストの最後から 2 番目）とアキュムレータに適用され、2 要素リストが入ったアプリカティブ値ができ、……同様に続けて、最後にすべてのアプリカティブの返り値のリストが入っているアキュムレータだけが残ります。

さっそく、この関数をアプリカティブ値に対して使ってみましょう。

```
ghci> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
ghci> sequenceA [Just 3, Nothing, Just 1]
Nothing
ghci> sequenceA [(+3),(+2),(+1)] 3
[6,5,4]
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]
[]
```

Maybe 値のリストに対して sequenceA を使うと、すべての返り値をリストとして含んだ単一の Maybe 値ができます。ただし、元のリストの中の Maybe 値のいずれかが Nothing であった場合、結果も Nothing になります。この振る舞いは、 Maybe 値のリストがあり、どれも Nothing でない場合に限って中身に興味がある、という場合に便利です。

関数に対して sequenceA を使うと、「関数のリスト」を取って「リストを返す関数」を返します。上の例では、ある数を引数に取り、リストの中の各関数を適用した結果をリストとして返すような関数を作っています。 sequenceA

`[(+3),(+2),(+1)] 3` は、関数 `(+3)` を引数 3 に、関数 `(+2)` を引数 3 に、関数 `(+1)` を引数 3 に適用し、その結果をリストとして返します。

前に `(+) <$> (+3) <*> (*2)` と書いたときには、「1 つの引数を取って `(+3)` と `(*2)` の両方を適用し、その 2 つの結果を + する」関数が作れました。これと同じノリで、`sequenceA [(+3),(*2)]` と書くと「1 つの引数を取ってそれをリストの中のすべての関数に食わせる」関数が作れるのです。ただし sequenceA では、返り値を + で結合する代わりに、`:` と `pure []` の組み合わせによって返り値たちを 1 つのリストに結合し、それが `sequenceA [(+3),(*2)]` 全体の返り値となります。

sequenceA は、関数のリストがあり、そのすべてに同じ引数を食わせて結果をリストとして眺めたい、という場合にはとても便利です。例えば、ある数がいくつかの性質をすべて満たしているかテストしたいときなどです。例えばこんなやり方があります。

```
ghci> map (\f -> f 7) [(>4),(<10),odd]
[True,True,True]
ghci> and $ map (\f -> f 7) [(>4),(<10),odd]
True
```

ここで and は、Bool のリストを取り、すべてが True である場合に限り True を返す関数でしたね。sequenceA を使うと別の書き方ができます。

```
ghci> sequenceA [(>4),(<10),odd] 7
[True,True,True]
ghci> and $ sequenceA [(>4),(<10),odd] 7
True
```

`sequenceA [(>4),(<10),odd]` が作る関数は、引数として取った数に `[(>4),(<10),odd]` のそれぞれを適用し、Bool のリストを返します。sequenceA は、`(Num a) => [a -> Bool]` の型を持ったリストを `(Num a) => a -> [Bool]` という型の関数に変えたのです。すごくないですか？

リストの中の関数はすべて同じ型じゃないといけません。例えば、`[ord, (+3)]` などというリストは作れません。ord は文字を取って数値を返すのに対し、`(+3)` は数値を取って数値を返すからです。

sequenceA を `[]` に対して使うと、リストのリストを取ってリストのリストを返す関数になります。実際の動作は、1 番目のリストから 1 つ、2 番目のリストから 1 つ、……というふうに要素を取って作れるリストをすべて列挙するというものになります。説明のために、先ほどの例を sequenceA およびリスト内包表記を使って再現してみます。

```
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
```

```
ghci> [[x,y] | x <- [1,2,3], y <- [4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2],[3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> [[x,y] | x <- [1,2], y <- [3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> sequenceA [[1,2],[3,4],[5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
ghci> [[x,y,z] | x <- [1,2], y <- [3,4], z <- [5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
```

(+) <$> [1,2] <*> [4,5,6] と書くと、x は [1,2] のいずれだか分からず、y は [4,5,6] のいずれだか分からないという文脈で、非決定性計算 x + y を行ってくれるのでした。非決定性計算の結果は、可能な答をすべて列挙したリストとして表現されるのでしたね。同様に、 sequenceA [[1,2],[3,4],[5,6]] と書くと、その結果は x は [1,2] のいずれだか分からず、 y は [3,4] のいずれだか分からず、……であるときに可能な [x,y,z] の組み合わせを列挙する、という非決定性計算になります。可能な答の 1 つひとつがリストであり、非決定性計算であることを表現するのに答のリストを使うので、全体の結果は二重リストになっているわけです。

I/O アクションに対して使ったときの sequenceA は、まさに sequence です！「I/O アクションのリスト」を引数に取って、「各 I/O アクションを実行し、返り値をリストとして返す I/O アクション」を返す関数になります。どうしてこうなるかというと、実行したら結果のリストを返すような I/O アクションを作るために [IO a] 型の値を IO [a] 型の値に変えたい、と思ったら、まずすべての I/O を直結し、評価が強制され次第 I/O を 1 つずつ実行するしかないからです。I/O アクションの結果は実行しないと取り出すことができません。

それでは 3 つの I/O アクション getLine を結合してみましょう。

```
ghci> sequenceA [getLine, getLine, getLine]
heyh
ho
woo
["heyh","ho","woo"]
```

まとめると、アプリカティブファンクターは、面白い上にとても便利なものです。アプリカティブファンクターを使えば、I/O を伴う計算、非決定性計算、失敗するかもしれない計算、……などなど、多種多様な計算をアプリカティブ・スタイルを使って組み合わせることができます。 <$> と <*> を使うだけで、普通の関数をどんなアプリカティブファンクターとも組み合わせて使うことができ、それぞれのアプリカティブファンクターの文脈の力を借りることができるのです。

第12章 モノイド

この章では、また 1 つ便利で楽しい型クラスを紹介します。それは Monoid です。モノイドは、値を二項演算子で結合できるような型を表します。この章では、モノイドとは正確には何なのか、モノイドの満たすべき法則とは何であるかを紹介します。それから、Haskell のモノイドの具体例や用途を見ていきます。

まずは **newtype** キーワードを見てみましょう。モノイドの素晴らしい世界を探検していると、**newtype** キーワードをしょっちゅう使うことになるからです。

12.1 既存の型を新しい型にくるむ

これまでに、**data** キーワードを使って独自の代数データ型を作る方法は学びました。また、**type** キーワードを使って既存の型に型シノニムを与える方法も見ました。この節では、**newtype** キーワードを使って、既存の型から新たな型を作る方法を見ていきます。そもそもなぜ **newtype** キーワードが必要なのかという疑問にも、あわせてお答えしたいと思います。

第 11 章では、リストをアプリカティブファンクターにする方法は複数あると言いました。1 つ目は、<*> が左辺のリストから関数を 1 つずつ取り出して、それぞれを右辺のリストの中の値すべてに適用し、左辺のリストの関数と右辺のリストの値のあらゆる組み合わせを作る、というものでした。

```
ghci> [(+1),(*100),(*5)] <*> [1,2,3]
[2,3,4,100,200,300,5,10,15]
```

2 つ目の方法は、<*> の左辺のリストから先頭の関数を取り出して右辺の先頭の値に適用し、左辺の次の関数を取り出して右辺の次の値に適用し、……と繰り返すものです。最終的に <*> は、2 つのリストを綴じ合わせるような働きをします。

しかし、リストはすでに Applicative のインスタンスだというのに、どうすればこの第二の Applicative インスタンスを定義できるのでしょう？ 第 11 章では、そのために ZipList a という型を導入しました。 ZipList 型の値コンストラクタは ZipList で、そのフィールドは 1 つだけです。そのフィールドにはリストを入れます。そして、綴じ合わせるタイプのアプリカティブとしてリストを使いたいときには、リストを ZipList コンストラクタでくるむだけで済むように、 ZipList を Applicative のインスタンスにしておきます。事が済んでくるまれた中身を取り出すには、 getZipList を使います。

```
ghci> getZipList $ ZipList [(+1),(*100),(*5)] <*> ZipList [1,2,3] $
[2,200,15]
```

で、これが **newtype** キーワードと何か関係があるんでしょうか？ ZipList a 型のデータ宣言をあなたならどう書くか考えてみてください。1 つの方法はこうです。

```
data ZipList a = ZipList [a]
```

この型には値コンストラクタが 1 つしかなく、その値コンストラクタにはフィールドがまた 1 つだけあって、そこにリストが入っています。 ZipList からリストを取り出す関数が自動的に手に入るように、レコード構文を使ってもいいでしょう。

```
data ZipList a = ZipList { getZipList :: [a] }
```

この方法はどこも悪くないように見えますし、実際うまく動きます。既存の型をある型クラスのインスタンスにする方法は 2 つありました（ **deriving** と **instance** ）。 **data** キーワードは既存の型を別の型でくるむためだけに使っていて、その別の型を第 11 章では **instance** によって Applicative のインスタンスにしました。

Haskell の **newtype** キーワードは、まさにこのような「1 つの型を取り、それを何かにくるんで別の型に見せかけたい」という場合のために作られたものです。 ZipList a は実際のライブラリではこう定義されています。

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

data キーワードの代わりに **newtype** キーワードが使われていますね。なぜでしょう？ まず、 **newtype** は高速です。型をくるむのに **data** キーワードを使

うと、コンストラクタに包んだりほどいたりするたびにオーバーヘッドがかかります。しかし **newtype** を使えば、「これは単に既存の型をくるんで作った新しい型だ（だから **newtype** と呼ぶのですね）、内部処理は同じまま違う型を持たせたいんだ」ということが Haskell に伝わります。Haskell は型推論を済ませた後、この点を考慮して、包んだりほどいたりする処理を省略してくれます。

では、なぜ逆に常に **data** の代わりに **newtype** を使わないのでしょうか？ それは、 **newtype** キーワードを使って既存の型から新しい型を作るときには、値コンストラクタは 1 種類しか作れず、その値コンストラクタが持てるフィールドも 1 つだけ、という制限があるからです。一方、 **data** キーワードを使えば、複数の値コンストラクタを持つデータ型を作れるし、各コンストラクタには 0 以上の任意個数のフィールドを持たせることができます。

```
data Profession = Fighter | Archer | Accountant

data Race = Human | Elf | Orc | Goblin

data PlayerCharacter = PlayerCharacter Race Profession
```

newtype で作った型に対する **deriving** キーワードも、 **data** の場合と同様にまったく問題なく使えます。 Eq 、 Ord 、 Enum 、 Bounded 、 Show 、 Read のインスタンスを導出できます[†1]。ある型クラスのインスタンスを導出したかったら、 **newtype** で包もうとしている中身の型がすでにその型クラスのインスタンスである必要があります。 **newtype** は既存の型を包んでいるだけなのですから、当然ですね。例えば、こうすれば新しい型は等号が使え、文字列に変換できるようになります。

```
newtype CharList = CharList { getCharList :: [Char] } deriving (Eq, Show)
```

使ってみましょう。

```
ghci> CharList "this will be shown!"
CharList {getCharList = "this will be shown!"}
ghci> CharList "benny" == CharList "benny"
True
ghci> CharList "benny" == CharList "oisters"
False
```

この **newtype** の例では、値コンストラクタの型はこうなります。

```
CharList :: [Char] -> CharList
```

この値コンストラクタは [Char] 型の値、例えば "my sharona" を取り、CharList 型の値を返します。さっきの例で CharList 値コンストラクタを使っ

[†1] ［訳注］GHC 言語拡張の GeneralizedNewtypeDeriving を使えば、元の型が所属していた任意の型クラスを導出できるようになります。

た箇所を見ると、実際そうなってますね。一方、 **newtype** 宣言でレコード構文を使ったことにより自動生成された getCharList 関数は、こんな型を持ちます。

```
getCharList :: CharList -> [Char]
```

getCharList は、 CharList を受け取って [Char] に変換します。これは包んだりほどいたりの処理だと考えてもいいですが、2 種類の型の間の変換だと考えることもできます。

newtype を使って型クラスのインスタンスを作る

ある型を型クラスのインスタンスにしたいのだが、型引数が一致しなくてできない、ということがよくあります。例えば、 Maybe を Functor のインスタンスにするのは簡単です。 Functor 型クラスの定義はこうだからです。

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

ですから、まずこう書いて

```
instance Functor Maybe where
```

あとは fmap を実装するだけです。

Maybe 型コンストラクタが Functor 型クラスの定義のうち f の位置を占め、すべての型引数が埋まります。 fmap は、もし Maybe に限定して動作するなら、こんな型を持つでしょう。

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

うーん、Haskell って素敵！ さて今度は、タプルを Functor のインスタンスにしたいと思ったとしましょう。 fmap をタプルに作用させて第一の要素（ fst ）を変更するようにしたい！ 例えば fmap (+3) (1, 1) と書くと (4, 1) になる、という感じです。ところが、このようなインスタンスは、いざ書こうとすると書きにくいことが分かります。 Maybe をインスタンス化するとき、 **instance** Functor Maybe **where** と簡単に書けたのは、型変数が 1 つの型コンストラクタだけが Functor のインスタンスになれるからです。型 (a, b) について同じようなことを書いて、 fmap が変更するのは型 a の部分だよ、と指定することはできそうにありません。この制限を回避するために、タプルを **newtype** して 2 つの型引数の順番を入れ替えることができます。

```
newtype Pair b a = Pair { getPair :: (a, b) }
```

これで、fmap が第一要素に作用するような形で、タプルを Functor のインスタンスにできます。

```
instance Functor (Pair c) where
    fmap f (Pair (x, y)) = Pair (f x, y)
```

ご覧のとおり、**newtype** で作った型にはパターンマッチも使えます。fmap の実装では、まずパターンマッチを使って Pair 型から中身のタプルを取り出し、f をタプルの第一要素に適用し、Pair 値コンストラクタを使ってタプルを Pair b a 型に再変換しています。Pair b a 型に限定した fmap の型はこうなります。

```
fmap :: (a -> b) -> Pair c a -> Pair c b
```

ここでは、**instance** Functor (Pair c) **where** と定義されており、Functor の型クラス定義

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

の f に Pair c が収まっているのです。

これで作りたかったものが作れました。タプルを Pair b a 型に変換すると、fmap が使えるようになり、関数をタプルの第一の要素に適用させることができます。

```
ghci> getPair $ fmap (*100) (Pair (2, 3))
(200,3)
ghci> getPair $ fmap reverse (Pair ("london calling", 3))
("gnillac nodnol",3)
```

newtype と遅延評価

newtype でできるのは、既存の型を新しい型に変えることだけです。ですから Haskell は、**newtype** で作られた型と元の型を型としては区別しつつ、同一の内部表現で扱っています。これは、**newtype** は **data** より高速に処理されるだけでなく、パターンマッチがより怠惰になることを意味します。この項ではこの点について見ていきましょう。

ご存知のとおり、Haskell はデフォルトで遅延評価な言語です。つまり、関数の結果を実際に表示しろと言われるまでは、Haskell は計算を始めません。さらに Haskell は、関数の結果を表示するのにどうしても必要な部分の計算しか行いません。さて、undefined はブッ壊れた計算を表す Haskell の値です。もし、端末に表示させるなどして、Haskell に undefined を評価しようとすると（つ

まり、どうしても undefined を計算させようとすると)、Haskell は怒りを爆発させます（専門用語では例外を投げるともいいます)。

```
ghci> undefined
*** Exception: Prelude.undefined
```

ところが、例えば undefined を要素に含むリストを作っても、その先頭要素を要求するだけなら、その先頭要素さえ undefined でなければ、すべてがうまくいっちゃいます。というのも、Haskell は先頭要素以外のリストの要素を評価する必要がないので、評価しないからです。例えばこのとおり。

```
ghci> head [3,4,5,undefined,2,undefined]
3
```

さて、ここで以下のような型を作ったとします。

```
data CoolBool = CoolBool { getCoolBool :: Bool }
```

これは **data** キーワードで作った、いかにもありふれた代数データ型です。CoolBool 型には 1 つの値コンストラクタがあり、それには 1 つのフィールドがあって、中身は Bool 型です。さて、 CoolBool 型をパターンマッチして、中身の Bool が True であるか False であるかによらず "hello" を返す関数を書いてみましょう。

```
helloMe :: CoolBool -> String
helloMe (CoolBool _) = "hello"
```

では、この関数を、正常な CoolBool 値じゃなくて undefined に適用してみましょう！

```
ghci> helloMe undefined
"*** Exception: Prelude.undefined
```

グワーッ！！ 例外だ！ どうしてこの例外が出たのでしょうか？ それは、**data** キーワードで定義された型には複数の値コンストラクタがあるかもしれず (CoolBool にはたまたま 1 つしかありませんが)、 helloMe 関数に与えられた引数が (CoolBool _) に合致するかどうかを確認するためには、どのコンストラクタが使われたのか分かるところまで引数の評価を進める必要があるからです。そして undefined を少しでも評価しようものなら……ドカーン！ 一巻の終わりです。

では、 CoolBool を作るのに、 **data** じゃなくて **newtype** を使ったらどうでしょう？

```
newtype CoolBool = CoolBool { getCoolBool :: Bool }
```

helloMe 関数を変更する必要はありません。**newtype** で定義した型も、**data** で定義した型も、パターンマッチの構文は共通だからです。さっきみたいに helloMe に undefined を適用してみましょう。

```
ghci> helloMe undefined
"hello"
```

おや、今度は動きました！ ムムムー、なんで？ これまでに学んだとおり、**newtype** を使ったときは、Haskell は新しい型の値も元の型の値も内部的には同じ表現を使えます。新しい型に対して箱を追加するような処理はしていませんし、ただ異なる型だと認識しているだけです。そして Haskell は、**newtype** キーワードはコンストラクタを 1 つしか作れないと知っているので、helloMe 関数の引数を評価することなく、引数が (CoolBool _) パターンに合致すると判定できます。だって **newtype** には値コンストラクタもフィールドも 1 つしかないのですから！

今紹介した事例は些細な違いに思えるかもしれませんが、これは実に重要な違いです。この例が示しているのは、**data** キーワードで定義した型と **newtype** キーワードで定義した型はプログラマの視点からはそっくりに見えるかもしれない（どちらにも値コンストラクタとフィールドがある）けれども、実際には 2 つの異なったメカニズムだということです。**data** はオリジナルな型を無から作り出すものです。これに対し **newtype** は、既存の型をもとに、はっきり区別される新しい型を作るものです。**data** に対するパターンマッチが箱から中身を取り出す操作なのに対し、**newtype** に対するパターンマッチは、ある型を別の型へ直接変換する操作なのです。

type vs. newtype vs. data

ここまできて、**type**、**data**、**newtype** の 3 つがどう違うのか混乱しているかもしれませんね。整理しておきましょう。

まず、**type** キーワードは型シノニムを作るためのものです。既存の型に別名をつけて、呼びやすくするのです。例えばこう書くと、

```
type IntList = [Int]
```

[Int] 型を IntList とも呼べるようになります。それだけです。2 つの呼び名は自由に交換できます。そして、IntList 型の値コンストラクタとか、そういう類のものは一切生じません。[Int] と IntList は同じものの別名にすぎないので、型注釈にどちらの名前を使うかは自由です。

```
ghci> ([1,2,3] :: IntList) ++ ([1,2,3] :: [Int])
[1,2,3,1,2,3]
```

型シノニムは、型シグネチャを整理して分かりやすくしたいときに使います。特定のものを表すために複雑な型を作ることがありますが、その型に名前をつければ、その型をどういう目的で使っているのかをコードを読む人に伝えられますよね。例えば第 7 章では、電話帳を表現するために `[(String, String)]` 型の連想リストを使いました。そして、それに `PhoneBook` という型シノニムをつけることで、電話帳を扱う関数の型シグネチャを読みやすくしました。

次に **newtype** キーワードは、既存の型を包んで新しい型を作るためのものです。 **newtype** は、もっぱら型クラスのインスタンスを作りやすくするために使われます。 **newtype** を使って既存の型を包むことにより出来上がる型は、元の型とは別物になります。次のような **newtype** を作ったとしましょう。

```
newtype CharList = CharList { getCharList :: [Char] }
```

このとき、 `CharList` と `[Char]` を `++` で連結することはできません。2 つの `CharList` を `++` で連結することすらできません。なぜなら、 `++` はリスト限定の演算子であり、 `CharList` はリストを中身に持っているとはいえ、リストそのものではないからです。しかし、 `CharList` をいったんリストに変換した上で `++` して、また `CharList` に戻すことならできます。

newtype 宣言でレコード構文を使うと、新しい型と元の型を相互変換する関数が作られます。具体的には、 **newtype** の値コンストラクタと、フィールド内の値を取り出す関数です。新しい型は、元の型の所属していた型クラスを引き継ぎがないので、 **deriving** で導出するか、インスタンス宣言を手書きする必要があります[†2]。

newtype 宣言は、値コンストラクタが 1 つだけ、フィールドも 1 つだけという制限の付いた **data** 宣言だとみなしても実用上は問題ありません。もし自分が書いたコードにそんな **data** 宣言を見つけたなら、 **newtype** で代用できないか考えてみてください。

data キーワードは、自作の新しいデータ型を作るためのものです。 **data** を使えば、フィールドとコンストラクタを山ほど備えたどんな途方もないデータ型だろうと作り出せます。リストや `Maybe` のような型も、木も、何でもです。

まとめると、3 つのキーワードはこのように使い分けてください。

- 型シグネチャを整理したいとか、型名が体を表すようにしたいだけなら、たぶん型シノニムを使うといいでしょう。

[†2] ［訳注］あるいは `GeneralizedNewtypeDeriving` を使う。

- 既存の型をある型クラスのインスタンスにしたくて、新しい型にくるむ方法を探しているのなら、 **newtype** がぴったり！
- 何かまったく新しいものを作りたい人には、きっと **data** が向いているよ！

12.2 Monoid 大集合

Haskell の型クラスは、同じ振る舞いをする型たちに共通のインターフェイスを提供するために使われています。最初に紹介したのは、等号が使える型のクラスである Eq や、順序が付けられる Ord といった単純なものでした。それから、Functor や Applicative のような、もっと面白い型クラスも紹介しました。

新しい型を作る人は、「この型には何ができるだろう？ どんな操作をサポートするだろう？」と考えて、その型に欲しい機能をもとに、どの型クラスのインスタンスを実装するかを決めます。もしその型の値どうしの等号を定義することに意味があるなら、 Eq のインスタンスにします。もしその型が何らかのファンクターになっているのなら、 Functor のインスタンスにします。などなど。

さて、こんなことを考えてみましょう。 * は 2 つの数を取って掛け算をする関数です。そして、何かと 1 を掛け算すると、結果は常に元の数です。 1 * x とやっても、 x * 1 とやっても、結果は x です。次に、関数 ++ を考えます。数の掛け算と 2 つのリストの連結は、全然別種の操作に思えますが、2 つのものを取って 1 つを返すという点では同じです。しかも、 ++ には * と同じように、演算しても相手を変えない値があります。それは空リスト [] です。

```
ghci> 4 * 1
4
ghci> 1 * 9
9
ghci> [1,2,3] ++ []
[1,2,3]
ghci> [] ++ [0.5, 2.5]
[0.5,2.5]
```

どうやら、 * に 1 という組み合わせと、 ++ に [] という組み合わせは、共通の性質を持っているようですね。

- 関数は引数を 2 つ取る。
- 2 つの引数および返り値の型はすべて等しい。
- 2 引数関数を施して相手を変えないような特殊な値が存在する。

よく観察すると、ほかにも共通の性質が見つかります。この関数を使って 3 つ以上の値を 1 つの値にまとめる計算をするとき、値の間に関数を挟む順序を変えても結果は変わらない、という性質です。例えば、`(3 * 4) * 5` も `3 * (4 * 5)` も、答は `60` です。`++` についてもこの性質は成り立ちます。

```
ghci> (3 * 2) * (8 * 5)
240
ghci> 3 * (2 * (8 * 5))
240
ghci> "la" ++ ("di" ++ "ga")
"ladiga"
ghci> ("la" ++ "di") ++ "ga"
"ladiga"
```

この性質を**結合的**（associativity）と呼びます。演算 `*` と `++` は結合的であると言います。結合的でない演算の例は `-` です。例えば `(5 - 3) - 4` と `5 - (3 - 4)` は異なる結果になります。

以上の性質に気づいたなら、……あなたはモノイドに出会ったのです！

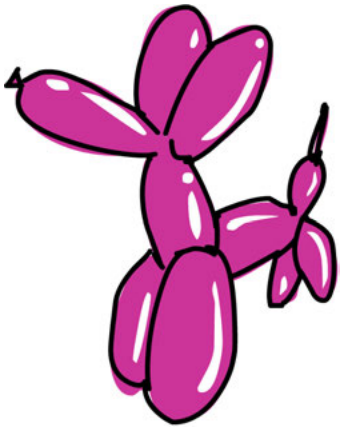
Monoid 型クラス

モノイドは、結合的な二項演算子（2 引数関数）と、その演算に関する単位元からなる構造です。ある値がある演算の単位元であるとは、その値と何か他の値を引数にしてその演算を呼び出したとき、返り値が常に他の値のほうに等しくなる、ということです。`1` は `*` の単位元であり、`[]` は `++` の単位元です。Haskell の世界では、ほかにも無数のモノイドがあるので、`Monoid` 型クラスが用意されています。`Monoid` の定義を見てみましょう。

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

`Monoid` 型クラスは `Data.Monoid` モジュールにて定義されています。今から少し時間をかけて解説するので、ぜひモノイドに慣れていってください。

まず、`Monoid` のインスタンスになれるのは具体型だけであることが分かります。型クラス定義に現れる `m` が型引数を取っていないからです。この点で `Monoid` は、`Functor` や `Applicative` のような、1 つの型引数を取る型コンストラクタがインスタンスになる型クラスとは異なっています。



最初の関数は mempty です。いや、引数を取らないので関数じゃないですね。 mempty は多相定数です。 Bounded の minBound みたいなものです。 mempty は、そのモノイドの単位元を表します。

次は mappend です。これは、お察しのとおり、モノイド固有の二項演算です。 mappend は同じ型の引数を 2 つ取り、その型の別の値を返します。この関数の名前を mappend にしたのは、ちょっと残念な判断だと思います。何かを付け足す（append）、という意味になってしまいますからね。 ++ は確かに 2 つのリストを継ぎ足す操作ですが、 * には付け足すという雰囲気が皆無ですからね。 Monoid の他のインスタンスを見ても、「付け足し」と呼べるような操作はあまりありません。ですから、 mappend を「付け足す」と考えるのはやめて、単に 2 つのモノイド値を取って第三の値を返す関数とみなすようにしてください。

最後の関数は mconcat です。これはモノイドのリストを取って mappend を間に挟んだ式を作り、単一の値を計算してくれる関数です。 mconcat には、 mempty を初期値に取り、リストを mappend で右畳み込みしていくというデフォルト実装が付いています。ほとんどのモノイドに関してはこのデフォルト実装で十分なので、 mconcat に関してはこれ以上深入りはしません。自分で Monoid 型クラスのインスタンスを作るときも、 mempty と mappend だけを実装すれば動きます。インスタンスによっては、もっと効率的な mconcat の実装があるしれませんが、多くの場合はデフォルトの実装で何ら問題ないでしょう。

モノイド則

Monoid の具体的なインスタンスを紹介する前に、まずモノイドが満たすべき法則を見ておきましょう。

これまでに、モノイドには固有の二項演算があること、その二項演算に関する単位元があること、その二項演算は結合的であること、などの条件を学びました。これらの条件を満たさない Monoid のインスタンスを作ることも可能ではありますが、そのようなインスタンスを作っても誰も得しません。なぜなら、Monoid のインスタンスを使う人は、それがモノイドのように振る舞うことを前提としているからです。でなければ、わざわざモノイドなんて持ち出す必要はな

いでしょう？ というわけで、Monoid のインスタンスを作るときは、必ず次の法則を満たすようにしておかないといけません。

- `mempty `mappend` x = x`
- `x `mappend` mempty = x`
- `(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)`

はじめの 2 つの法則は、 mempty が mappend に関して単位元として振る舞う必要があることを述べています。第三の法則は、 mappend が結合的であること、つまり複数の mappend で連結された式から 1 つの値を計算するとき、 mappend を評価する順序は最終結果に影響しないことを述べています。Haskell はこれらの法則を強制しませんので、インスタンスが実際にこれらの法則を満たすよう気をつけるのは僕たちプログラマの責任です。

12.3　モノイドとの遭遇

さて、モノイドとは何かが分かったところで、Haskell の型のうちモノイドであるものを紹介し、それらの Monoid インスタンスがどう実装されていて何に使うのかを見ていきましょう。

リストはモノイド

はい、リストはモノイドです！ すでに見たように、関数 ++ と空リスト [] がモノイドを成します。インスタンス宣言はとてもシンプルです。

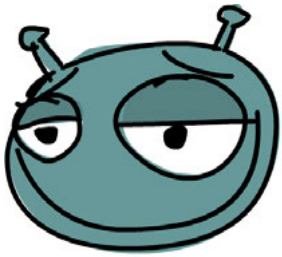
```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

リストは、中身の型が何であっても常に Monoid のインスタンスにできます。インスタンス宣言が **instance** Monoid [] ではなく **instance** Monoid [a] となっているのに気がつきましたか？ これは、 Monoid のインスタンスは具体型と決まっているからです。

動かしてみれば、何の苦もなく使えることが分かるでしょう。

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> ("one" `mappend` "two") `mappend` "tree"
"onetwotree"
ghci> "one" `mappend` ("two" `mappend` "tree")
"onetwotree"
ghci> "one" `mappend` "two" `mappend` "tree"
"onetwotree"
```

```
ghci> "pang" `mappend` mempty
"pang"
ghci> mconcat [[1,2],[3,6],[9]]
[1,2,3,6,9]
ghci> mempty :: [a]
[]
```



最後の行には型注釈を明記してありますね。単に mempty とだけ書いても、どのインスタンスを使ってよいか GHCi には分からないので、「ここではリストのインスタンスを使いたい」と伝える必要があります。ただし、[Int] や [String] ではなく、一般的な [a] という型でかまいません。空リストは任意の型を格納しているかのように振る舞えるからです。

mconcat にはデフォルト実装が指定されているので、Monoid のインスタンスを作れば mconcat は勝手に付いてきます。リストの場合、mconcat は実はただの concat です。mconcat は二重リストを取って、その要素を連結した平らなリストを返します。というのも、二重リストの中の隣接するリストをみな ++ で結ぶとそうなるからです。

リストは確かにモノイド則を満たします。複数のリストがあって、それを mappend、つまり ++ で結合していった結果は、どこから結合し始めても一緒ですよね。だって最後にはどのみちすべてが結合するのですから。空リストが単位元であることも分かります。以上です。

ところで、モノイド則は a `mappend` b と b `mappend` a が等しいことは要求していない点に気をつけてください。リストの場合、これは明らかに成り立っていませんよね。

```
ghci> "one" `mappend` "two"
"onetwo"
ghci> "two" `mappend` "one"
"twoone"
```

これでいいんです。3 * 5 と 5 * 3 は確かに同じですが、それは掛け算の性質であって、すべてのモノイドがこの性質を満たすわけではありません。実際、満たさないモノイドがほとんどです。

Product と Sum

数をモノイドにする方法の 1 つはすでに紹介しましたよね。* を二項演算にして 1 を単位元にする、という方法です。ほかにも、+ を演算にして 0 を単位元にするという方法もあります。

```
ghci> 0 + 4
4
ghci> 5 + 0
5
ghci> (1 + 3) + 5
9
ghci> 1 + (3 + 5)
9
```

0 は足し算に関する単位元ですし、足し算は結合法則も満たしますから、モノイド則が成り立っています。何の問題もありません。

さて、数をモノイドにする 2 つの方法は、どちらも素晴らしく優劣つけがたいように思えます。一体どちらを選べばよいのでしょう？ 実は、1 つだけ選ぶ必要はないのです。ある型に対して同じ型クラスのインスタンスを複数定義したかったら、 **newtype** に包んで新しい型をインスタンスにするという方法がありましたよね。二兎を追って両方捕まえることができるのです。

Data.Monoid モジュールは、この用途のために、 Product と Sum という 2 つの型をエクスポートしています。 Product の定義はこうです。

```
newtype Product a =  Product { getProduct :: a }
    deriving (Eq, Ord, Read, Show, Bounded)
```

実に単純。 **newtype** ラッパーと導出したインスタンスがいくつかあるだけです。 Product の Monoid インスタンスは、こんな感じです。

```
instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

Product モノイドの mempty は、ただの 1 を Product コンストラクタにくるんだものです。 mappend は、 Product をパターンマッチしており、中身の数を掛け算してそれをまたコンストラクタにくるんでいます。そして、見てのとおり、 Num a という型クラス制約が付いています。すでに Num のインスタンスであるようなすべての型 a について、 Product a は Monoid のインスタンスになる、ということです。 Product a をモノイドとして使うには、 **newtype** に包んだりほどいたりする必要があります。

```
ghci> getProduct $ Product 3 `mappend` Product 9
27
ghci> getProduct $ Product 3 `mappend` mempty
3
ghci> getProduct $ Product 3 `mappend` Product 4 `mappend` Product 2
24
ghci> getProduct . mconcat . map Product $ [3,4,2]
24
```

Sum は Product の近くでまったく同様に定義されていて、インスタンスもよく似ています。使い方も同じです。

```
ghci> getSum $ Sum 2 `mappend` Sum 9
11
ghci> getSum $ mempty `mappend` Sum 3
3
ghci> getSum . mconcat . map Sum $ [1,2,3]
6
```

Any と All

モノイドにする方法が2通りあって、どちらも捨てがたいような型は、Num a 以外にもあります。Bool です。1つ目の方法は || をモノイド演算とし、False を単位元とする方法です。|| は論理和を表し、2つの引数のいずれかが True ならば True を返し、そうでなければ False を返す関数です。False を単位元として使えば、確かに || は False を取れば False を返し、True を取れば True を返しますね。これを踏まえて **newtype** で Any が定義され、Monoid のインスタンスにされています。Any の定義はこうです。

```
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Ord, Read, Show, Bounded)
```

インスタンス定義はこんな感じです。

```
instance Monoid Any where
        mempty = Any False
        Any x `mappend` Any y = Any (x || y)
```

これが Any と呼ばれるのは、x `mappend` y は x か y のいずれか（**any**）が True であった場合に True になるからです。Any で包んだ Bool を 3 つ以上 mappend した場合も同様で、リストの中身がいずれか1つでも True であった場合に全体が True になります。

```
ghci> getAny $ Any True `mappend` Any False
True
ghci> getAny $ mempty `mappend` Any True
True
ghci> getAny . mconcat . map Any $ [False, False, False, True]
True
ghci> getAny $ mempty `mappend` mempty
False
```

Bool を Monoid のインスタンスにするもう1つの方法は、Any のいわば真逆です。&& をモノイド演算とし、True を単位元とする方法です。論理積は、2つの引数がともに True である場合に限り True を返します。

これがその **newtype** 宣言です。

```
newtype All = All { getAll :: Bool }
        deriving (Eq, Ord, Read, Show, Bounded)
```

そしてこれがインスタンス定義です。

```
instance Monoid All where
        mempty = All True
        All x `mappend` All y = All (x && y)
```

All 型の値を mappend した結果は、すべて（**all**）の引数が True であった場合に限り True になります。

```
ghci> getAll $ mempty `mappend` All True
True
ghci> getAll $ mempty `mappend` All False
False
ghci> getAll . mconcat . map All $ [True, True, True]
True
ghci> getAll . mconcat . map All $ [True, True, False]
False
```

うーむ、面倒です。足し算や掛け算のときもそうでしたが、こんなふうに **newtype** にくるんで mappend と mempty を使ったりするんだったら、普通は Bool の関数を直接呼び出しますよね。 mconcat を Any や All と組み合わせて使うのはまだマシに見えますが、これだって or や and の関数を使ったほうが簡単な場合が多そうです。 or は、Bool のリストを取って、リストの要素のいずれかが True ならば True を返す関数です。 and は、同じく Bool のリストを取って、すべてが True のときに True を返す関数です。

Ordering モノイド

Ordering 型を覚えていますか？ Ordering は、ものを比較した結果を表すのに使い、 LT 、 EQ 、 GT という 3 つの値のいずれかを取ります。それぞれ「小さい」「等しい」「大きい」を表しています。

```
ghci> 1 `compare` 2
LT
ghci> 2 `compare` 2
EQ
ghci> 3 `compare` 2
GT
```

これまで見てきたリスト、数、 Bool の場合、モノイドのインスタンスを見つけるのは、すでに存在するよく使われている関数の中から候補を探し、モノイドとしての性質を備えているか調べるだけの作業でした。 Ordering の場合、モノイドを見抜くのはちょっと難しいです。しかし Ordering の Monoid インスタン

スは、分かってみれば今までのモノイドと同じくごく自然な定義で、しかも便利なんです。

```
instance Monoid Ordering where
    mempty = EQ
    LT `mappend` _ = LT
    EQ `mappend` y = y
    GT `mappend` _ = GT
```

このインスタンスは次のような仕組みになっています。2 つの Ordering 値を mappend すると、左辺の値が優先されますが、左辺が EQ である場合は別です。左辺が EQ である場合、右辺が返り値になります。単位元は EQ です。最初は場当たり的なルールに見えるかもしれませんが、実はこれは文字列を辞書順で比較するときのルールに合わせた定義になっています。2 つの文字列を辞書順比較するときは、まず先頭の文字を比較し、異なっていた場合は直ちに辞書順が決まります。ところが、先頭の文字が同じだった場合は、次の文字を比較し、……と繰り返す必要がありますね。

例えば、ox と on という単語を辞書順で比較するときは、まず先頭の文字を比較し、それらは同じなので 2 文字目の比較に進みます。すると、x は n より辞書順が大きいので、単語の順序も ox は on より大きかったことが分かります。 EQ が単位元とされているのは、「2 つの単語の同じ位置に同じ文字を挿入しても辞書順は変化しない」ことに対応すると考えれば、直感的に理解できるのではないでしょうか。例えば、oix も oin より辞書順が大きいわけです。

Ordering の Monoid インスタンスでは x `mappend` y は y `mappend` x と一致しない、という点も注意が必要です。左辺が EQ でない限り左辺が優先されるので、 LT `mappend` GT は LT を返しますが、 GT `mappend` LT は GT になります。

```
ghci> LT `mappend` GT
LT
ghci> GT `mappend` LT
GT
ghci> mempty `mappend` LT
LT
ghci> mempty `mappend` GT
GT
```

では、このモノイドはどういうときに便利なのでしょう？ 例えば、2 つの文字列を引数に取り、その長さを比較して Ordering を返す関数を書きたいとしま

しょう。ただし、2 つの文字列の長さが等しいときは、直ちに EQ を返すのではなくて、2 つの文字列を辞書順比較することとします。

こう書くのが 1 つの手です。

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y = let a = length x `compare` length y
                        b = x `compare` y
                    in  if a == EQ then b else a
```

長さの比較結果に a 、辞書順の比較結果に b という名前をつけ、長さが等しければ辞書順を返しています。

しかし、 Ordering はモノイドであるという知識を使えば、この関数はずっとシンプルに書けるんです。

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
                    (x `compare` y)
```

試してみましょう。

```
ghci> lengthCompare "zen" "ants"
LT
ghci> lengthCompare "zen" "ant"
GT
```

mappend は、左辺が EQ でなければ左辺、 EQ であれば右辺を返すのでしたね。ですから、優劣をつける場合に重視したい比較条件を左辺に置けばよいのです。さて、今度は単語の中の母音の数も比較して、それを 2 番目に重要な条件にしたくなったとしましょう。ならば、こう修正すればよろしい。

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
                    (vowels x `compare` vowels y) `mappend`
                    (x `compare` y)
    where vowels = length . filter (`elem` "aeiou")
```

vowel は、文字列中の母音の数を返す補助関数です。 vowel は、まず文字列 "aeiou" に含まれる文字かどうかで引数の文字列をフィルタし、その上で length を取るという仕組みになっています。

```
ghci> lengthCompare "zen" "anna"
LT
ghci> lengthCompare "zen" "ana"
LT
ghci> lengthCompare "zen" "ann"
GT
```

1つ目の例では、"zen" のほうが "anna" より短いですから、LT が返っています。2つ目の例では、長さは等しいですが、2つ目の文字列のほうが母音が多いので、やはり LT が返っています。3つ目の例では、どちらの文字列も同じ長さで母音の数も等しいため辞書順比較が発生し、"zen" が勝っています。

このように Ordering モノイドは、さまざまな条件でものの大小を比較し、条件そのものに「最も重視すべき条件」から「どうでもいい条件」まで優先順位をつけて最終判定を出すのに使える、とても便利なものなんです！

Maybe モノイド

Maybe a も複数の方法でモノイドになれます。Maybe a がどのような Monoid インスタンスになるのか、どんなことに使えるのか、見ていきましょう。

Maybe a をモノイドにする1つ目の方法は、型引数 a がモノイドであるときに限り Maybe a もモノイドであるとし、Maybe a の mappend を、Just の中身の mappend を使って定義することです。Nothing を単位元とし、mappend される2つの値のうち片方が Nothing であれば別の片方の値を使う、とします。これがインスタンス宣言です。

```
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

型クラス制約を見てください。a が Monoid のインスタンスである場合に限り、Maybe a を Monoid のインスタンスとする、と書いてあります。「何か」と Nothing を mappend した結果は、その「何か」になります。2つの Just 値を mappend した場合は、2つの Just の中身を mappend して、また Just の中に入れます。これができるのも、型クラス制約によって Just の中身の型は Monoid のインスタンスであることが保証されているからこそです。

```
ghci> Nothing `mappend` Just "andy"
Just "andy"
ghci> Just LT `mappend` Nothing
Just LT
ghci> Just (Sum 3) `mappend` Just (Sum 4)
Just (Sum {getSum = 7})
```

これは、失敗するかもしれない計算の返り値をモノイドとして扱いたい場合に便利です。このインスタンスがあるおかげで、個々の計算が失敗したかどうかをいちいち覗き込まずに済みます。Nothing か Just かの判定などせずに、そのまま普通のモノイドとして扱ってやればよいのです。

でも、Maybe の中身が Monoid のインスタンスではなかったら？ そういえば、中身がモノイドであることを利用したのは mappend の両辺が Just である場合だけでしたね。中身がモノイドかどうか分からない状態では、mappend は使えません。どうすればいいでしょう？ 1 つの選択は、第一引数を返して第二引数は捨てる、と決めておくことです。この用途のために First a というものが存在します。これが定義です。

```
newtype First a = First { getFirst :: Maybe a }
    deriving (Eq, Ord, Read, Show)
```

Maybe a が **newtype** で包まれていますね。Monoid インスタンスはこうです。

```
instance Monoid (First a) where
    mempty = First Nothing
    First (Just x) `mappend` _ = First (Just x)
    First Nothing `mappend` x = x
```

mempty は、ただ単に Nothing を First で包んだものです。さて、mappend は、第一引数が Just 値なら第二引数を無視します。もし第一引数が Nothing なら、第二引数が Just であろうと Nothing であろうと、それが返り値になります。

```
ghci> getFirst $ First (Just 'a') `mappend` First (Just 'b')
Just 'a'
ghci> getFirst $ First Nothing `mappend` First (Just 'b')
Just 'b'
ghci> getFirst $ First (Just 'a') `mappend` First Nothing
Just 'a'
```

First は、いくつもある Maybe 値にどれか 1 つでも Just があるか調べたいときに役立ちます。mconcat 関数が便利ですよ。

```
ghci> getFirst . mconcat . map First $ [Nothing, Just 9, Just 10]
Just 9
```

逆に、2 つの Just を mappend したときに後のほうの引数を優先するような Maybe a が欲しい、という人のために、Data.Monoid には Last a 型も用意されています。これは First a とそっくりな働きをしますが、ただし mappend や mconcat を使ったときには Nothing でない最後の値が採用されるというものです。

```
ghci> getLast . mconcat . map Last $ [Nothing, Just 9, Just 10]
Just 10
ghci> getLast $ Last (Just "one") `mappend` Last (Just "two")
Just "two"
```

12.4 モノイドで畳み込む

いろんなデータ構造の上に畳み込みを定義したい……、そういうときにもモノイドが活躍します。これまでは、リストの畳み込みしか紹介しませんでした。しかし、畳み込みできるデータ構造はリストだけではありません。むしろ、ほとんどすべてのデータ構造の上に畳み込みを定義できるんです。木構造などは畳み込みしやすいデータ構造の典型です。

畳み込みと相性の良いデータ構造は実にたくさんあるので、 Foldable 型クラスが導入されました。 Functor が関数で写せるものを表すように、 Foldable は畳み込みできるものを表しています。 Foldable は Data.Foldable モジュールの中にあります。このモジュールが提供する関数群の名前は Prelude の関数と衝突するので、 Data.Foldable モジュールは修飾付きインポートをするのがおすすめです。

```
import qualified Data.Foldable as F
```

貴重なタイプ数を節約するため、修飾子の名前は F にしました。

この型クラスが定義する関数にはどんなものがあるのでしょう。どれどれ、foldr 、 foldl 、 foldr1 、それから foldl1 ……って、あれ？ どの関数も前に見ましたよ。何が新しいんでしょう？ Foldable の foldr と Prelude の foldr の型を比べてみましょう。

```
ghci> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
ghci> :t F.foldr
F.foldr :: (F.Foldable t) => (a -> b -> b) -> b -> t a -> b
```

ああ！ どうやら、 Prelude の foldr はリストを取って畳み込みを行う関数である一方、 Data.Foldable の foldr は、畳み込みができる型ならリストに限らず、何でも受け付ける関数のようですね！ 期待どおり、どちらの foldr も、リストに対してはまったく同じ動作をします。

```
ghci> foldr (*) 1 [1,2,3]
6
ghci> F.foldr (*) 1 [1,2,3]
6
```

畳み込みをサポートするデータ構造はリスト以外にもあります。例えば、みんな大好き Maybe もそうです！

```
ghci> F.foldl (+) 2 (Just 9)
11
ghci> F.foldr (||) False (Just True)
True
```

しかし、 Maybe を畳み込むのはそこまで面白いものではありません。 Maybe は単に、 Just 値だったら 1 要素のリストのように振る舞い、 Nothing 値なら空リストのように振る舞うだけです。もう少し複雑なデータ構造を試してみようじゃないか！

第 7 章で木構造を作りましたよね？ そのときの定義はこんな感じでした。

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show)
```

木は、値を持たない空の木か、1 つの値と 2 つの木を持つノードからなる再帰的構造として定義しました。木を定義した後、木を Functor のインスタンスにすることで、木に関数を fmap できるようにしました。今度は、木を Foldable のインスタンスにすることで木に畳み込みが使えるようにしましょう。

ある型コンストラクタを Foldable のインスタンスにする 1 つの方法は、直接 foldr を実装することです。けれども、通常はもっと簡単な方法があります。それは foldMap 関数を実装することです。 foldMap 関数は、他の fold 系の関数と同様に Foldable 型クラスの一員で、次のような型を持っています。

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

foldMap 関数の第一引数は、「 Foldable にしたいコンテナの中身（ a ）の型の値を取って、モノイドを返す」関数です。 foldMap 関数の第二引数は、 a 型の値を含む Foldable 構造自身です。 foldMap 関数は、まず第二引数の構造体に第一引数の関数を map して、構造体の中身をごっそりモノイド値に変えます。次にそれらモノイド値を mappend して単一のモノイド値へと結合します。この動作は何だか不自然に思えるかもしれませんが、すぐに、これは実に簡単に実装できる仕様なのだと分かりますよ。そしてこの foldMap さえあれば、ある型を Foldable にできます！ つまり、ある型の foldMap さえ定義すれば、その型の foldr や foldl は自動的に手に入るのです。

Tree を Foldable のインスタンスにする方法は、こうです。

```
instance F.Foldable Tree where
    foldMap f EmptyTree = mempty
    foldMap f (Node x l r) = F.foldMap f l `mappend`
                             f x           `mappend`
                             F.foldMap f r
```

さて、僕らの木構造の要素を取ってモノイド値を返す関数を誰かがくれたとします。その関数を使って、木構造全体を 1 つのモノイド値に帰着するにはどうすればいいのでしょう？ fmap のときは、与えられた関数をその場でノードに適用し、さらに左右 2 つある部分木にも再帰的に fmap を適用しました。今度は、関数を map するだけではなく、その結果を mappend して単一のモノイド値にする作業も必要です。まあ、やってみましょう。まず空の木の場合、つまり、値も

なければ部分木もない独りぼっちの寂しく悲しい木の場合には、モノイド作りの材料が何もないというのなら mempty になるしかないですね。

空じゃない木（Node）の場合はもうちょっと面白くなります。こちらは 2 つの部分木と 1 つの値を含みます。この場合は、まず左右の部分木に再帰的に f を foldMap しましょう。それぞれ単一のモノイド値が返ってくるはずですね。それからノードが持っている値にも f を適用します。これで 3 つのモノイド値（2 つは部分木からきたもので、1 つはノード内の値に f を適用してできたもの）が揃いました。あとはこの 3 つを 1 つの値に畳むだけです。ここは mappend の出番ですね。まず左部分木、次にノードの値、最後に右部分木という順番にさりげない配慮が光ります。

foldMap の実装自体は、値をモノイドに変換する関数を必要としなかったことに気づきましたか？ 変換関数は foldMap に引数として与えられるので、foldMap の実装に必要なのは、変換関数は所与のものとして、どこに適用して結果のモノイドをどう結合するのかを決めてやることなのです。

こうして僕らの木は Foldable インスタンスになりました。これで foldr と foldl も無料で手に入ったわけです！ 例えばこんな木を作ります。

```
testTree = Node 5
            (Node 3
                (Node 1 EmptyTree EmptyTree)
                (Node 6 EmptyTree EmptyTree)
            )
            (Node 9
                (Node 8 EmptyTree EmptyTree)
                (Node 10 EmptyTree EmptyTree)
            )
```

ルートに 5 があって、左のノードは 3 で、その左右に 1 と 6 があります。ルートの右のノードは 9 で、その左の子は 8 で、一番右に 10 があるわけです。Tree が Foldable インスタンスであるおかげで、リストに使える畳み込み関数はすべてこの Tree にも使えます。

```
ghci> F.foldl (+) 0 testTree
42
ghci> F.foldl (*) 1 testTree
64800
```

foldMap は Foldable のインスタンスを定義するためだけの存在ではありません。 Foldable な構造を単一のモノイド値に畳みたいときには実際役に立ちます。例えば、この木の中に 3 に等しい数があるか調べたかったら、こう書けばよろしい。

```
ghci> getAny $ F.foldMap (\x -> Any $ x == 3) testTree
True
```

ここで \x -> Any $ x == 3 は、数を取って Any にくるまれた Bool というモノイド値を返す関数です。 foldMap はこの関数を木のすべての要素に適用してから、 mappend を使って 1 つのモノイド値に畳んでくれるわけです。例えばこんなことをすると、

```
ghci> getAny $ F.foldMap (\x -> Any $ x > 15) testTree
False
```

木の中のすべてのノードの値は、ラムダ式の中の関数を適用された結果 Any False に変わります。複数の Any を mappend して True が返るのは、そのうち 1 つでも True が含まれる場合に限られます。あの木には 15 より大きい要素は含まれていませんから、 False が返ってくるのは当然の結果です。

木構造をいともたやすくリストに変換できる技もあります。 \x -> [x] を foldMap するのです。このラムダ式は、木の各要素を単一要素のリストに変換します。それから mappend がこれらリストモノイドの間に適用され、元の木の要素をすべて含んだリストという単一のモノイドが出来上がります。

```
ghci> F.foldMap (\x -> [x]) testTree
[1,3,6,5,8,9,10]
```

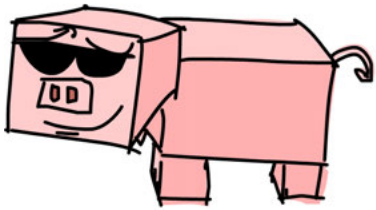
しかもこれらの技は、木構造に限らず、あらゆる Foldable なデータ構造に適用できます！

第13章
モナドがいっぱい

第7章で初めてファンクターの話をしたとき、ファンクターは関数で写せる値を表す便利な概念であることを見ました。第11章ではファンクターを一歩拡張してアプリカティブファンクターを導入し、ある種のデータ型は、文脈を持った値だと解釈できるようになりました。アプリカティブファンクターを使えば、そのような文脈を保ったまま、通常の関数をそれらの値に適用できるのでした。

この章では**モナド**を紹介します。モナドは強化されたアプリカティブファンクターです。ちょうど、アプリカティブファンクターが強化されたファンクターであったように。

13.1 アプリカティブファンクターを強化する



ファンクターを勉強し始めたとき、Functor 型クラスに属するさまざまな型は、すべて関数で写せることを見ました。ファンクターを導入した動機は、「a -> b 型の関数と、f a というデータ型があるとして、どうすれば当の関数を f a から f b への関数に変換できるだろう？」というものでした。Maybe a、リスト [a]、IO a などに対し、関数で写す方法を見てきました。a -> b 型の関数はなんと「r -> a 型の関数」を写すこともでき、その結果は「r -> b 型の関数」になることも見ました。あるデータ型を関数で写したときの挙動を指定するには、fmap の型を見てから、

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

Functor インスタンスを書き、fmap がそのデータ型を適切に処理できるようにすればよいのでした。

それから、いくつかの疑問とともに、ファンクターを改良できる可能性が浮かび上がってきました。関数 a -> b が、はじめからファンクターに包まれていたらどうする？ 例えば Just (*3) があったとして、それを Just 5 に適用するには？ その Just 5 が急に Nothing に変わったら？ Maybe の代わりにリスト [(*2),(+4)] があったとして、それを [1,2,3] に適用するには？ そもそも、そんなの動くの？ これらを解決するために、 Applicative 型クラスを導入しました。

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

Applicative 型クラスでは、通常の値をデータ型の中に入れる操作も可能になったのでした。例えば、 1 を Just 1 に変えたり、 [1] に変えたり、はたまた「何も副作用を起こさず 1 を返す I/O アクション」に変えたりできるのでした。この変換をしてくれる関数の名前は pure というのでしたね。

アプリカティブ値は、**変な**値、専門用語でいうと「文脈の付加された値」だとみなせます。例えば、文字 'a' はただの文字ですが、 Just 'a' は何らかの文脈が付いています。 Char 型の代わりに Maybe Char 型がきたとしたら、この値は文字かもしれないし文字がないことを表すのかもしれないということです。 Applicative 型クラスは、これら文脈の付いた値に、文脈を保ったまま普通の関数を適用させてくれます。例を見てみましょう。

```
ghci> (*) <$> Just 2 <*> Just 8
Just 16
ghci> (++) <$> Just "exdeath" <*> Nothing
Nothing
ghci> (-) <$> [3,4] <*> [1,2,3]
[2,1,0,3,2,1]
```

このようにアプリカティブ値として扱い始めると、 Maybe a は失敗するかもしれない計算、 [a] は複数の答があり得る計算（非決定性計算）、 IO a は副作用を伴う計算、などの意味を帯びてくるのでした。

さて、モナドはある願いを叶えるための、アプリカティブ値の自然な拡張です。その願いとは、「普通の値 a を取って文脈付きの値を返す関数に、文脈付きの値 m a を渡したい」というものです。言い換えると、 a -> m b 型の関数を m a 型の値に適用したい、ということ。ぶっちゃけると、この関数が欲しいってことです。

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

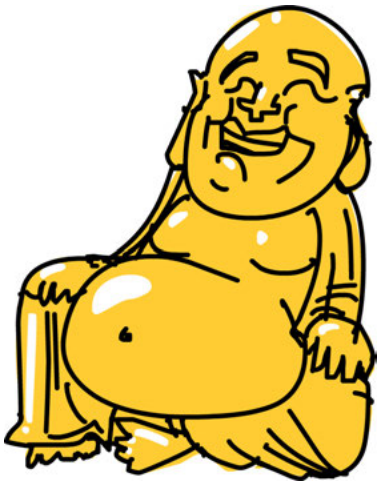
さて、変な値と、普通の値を取るけど変な値を返す関数があったとき、どうやってその値を関数に食わせればいいのでしょう？ これがモナドの一番の関心事です。これからは f a の代わりに m a と書くことにします。 m は Monad の頭

文字です。呼び方は変わりますが、モナドは >>= をサポートするアプリカティブファンクターにすぎません。関数 >>= は**バインド**（bind）と呼ばれます。

普通の値 a と普通の関数 a -> b だったら、値を関数に食わせるのは造作もないことです。単に関数を値に適用すればよろしい。ところが、特定の文脈が付きまとう値を扱うとなると、その変な値を関数に食わせるとどうなるのか、文脈を保つにはどうするのか、ちゃんと考えないといけません。でも、やってみれば簡単だということが分かりますよ！

13.2 Maybe から始めるモナド

モナドが何者なのかぼんやり分かってきたところで、具体例を見ていきましょう。実は Maybe はモナドだったんです、と言っても誰も驚かないでしょうね。このセクションでは、モナドとしての Maybe がどういう働きをするのか、詳しく見ていきましょう。



NOTE 先に進む前に、アプリカティブファンクターはちゃんと理解しておるか？ アプリカティブファンクターについては第 11 章で学んだじゃろう。さまざまな Applicative インスタンスはなぜ動くのか、それが表す計算の種類は何か、身に付けておくのじゃ。アプリカティブファンクターの知識を踏まえ、深めていくことだけがモナドの奥義に至る道なのじゃからな。

Maybe a 型の値は a 型の値を表していますが、失敗する可能性という文脈付きです。Just "dharma" という値は、文字列 "dharma" がそこに実在することを意味します。Nothing という値は、無を、あるいは文字列が何らかの計算の結果だとしたら、その計算が失敗したことを表しています。

ファンクターとして見た Maybe に対して関数を fmap すると、その Maybe が Just 値だった場合には、与えた関数が Just の中身の値に適用されるのでした。Nothing の場合は、Nothing のままでした。だって、関数を適用する相手がいないんですから！

```
ghci> fmap (++"!") (Just "wisdom")
Just "wisdom!"
```

```
ghci> fmap (++"!") Nothing
Nothing
```

アプリカティブファンクターとしての Maybe の機能も似たようなものです。ただし、アプリカティブファンクターになると、値だけでなく、値に適用する関数のほうにも文脈が付きます。 Maybe の Applicative インスタンスは、 <*> を使って Maybe の中の関数を Maybe の中の値に適用しようとすると、関数と値が両方 Just ならば結果が Just になる、そうでなければ結果は Nothing になる、というものでした。当然ですよね。関数か値のどちらかが欠けていれば、無から適用結果をでっち上げるわけにはいきませんから、失敗を伝播させる必要があります。

```
ghci> Just (+3) <*> Just 3
Just 6
ghci> Nothing <*> Just "greed"
Nothing
ghci> Just ord <*> Nothing
Nothing
```

アプリカティブ・スタイルを使って普通の関数を Maybe 値に適用するときも、同じやり方でうまくいきます。すべての引数が Just でなければ、結果は Nothing です！

```
ghci> max <$> Just 3 <*> Just 6
Just 6
ghci> max <$> Just 3 <*> Nothing
Nothing
```

それではいよいよ、 Maybe にとっての >>= をどう定義すればいいか考えていきましょう。 >>= は、「モナド値」と「普通の値を取る関数」を引数に取り、何とかしてその関数をモナド値に適用してモナド値を得ます。関数のほうは普通の値しか取れないのに、どうやってそんな芸当ができるのでしょう？ その答を出すには、モナド値の文脈に立ち入る必要があります。

この場合、 >>= は Maybe a 型の値と a -> Maybe b 型の関数を取り、この関数をどうにかして Maybe a に適用するわけです。これをどうやって実現しているのか探るには、 Maybe がアプリカティブファンクターであるという知識が役立ちます。さて、関数 \x -> Just (x+1) を考えましょう。これは数を取り、それに 1 を足して、結果を Just に包みます。

```
ghci> (\x -> Just (x+1)) 1
Just 2
ghci> (\x -> Just (x+1)) 100
Just 101
```

この関数に 1 を食わせると、Just 2 に評価されます。100 を食わせると、結果は Just 101 です。実に直感的ですね。じゃあ、この関数に Maybe 値を食わせるにはどうしたらいいでしょう？ Maybe のアプリカティブファンクターとしての振る舞いから類推すると、この問に答えるのは極めて簡単です。Just 値がきたときは Just の中身を取り出し、それを関数に食わせればよろしい。Nothing 値がきたときは、関数はあるものの、それに適用すべき値がナッシングというわけですから、結果も Nothing とせざるを得ません。

ひとまず >>= と呼ぶのはやめて applyMaybe という名前にしましょうか。これは「Maybe a 型の値」と「Maybe b を返す関数」を引数に取り、どうにかしてその関数を Maybe a に適用してくれる関数です。コードはこちら。

```
applyMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
applyMaybe Nothing f  = Nothing
applyMaybe (Just x) f = f x
```

それでは使ってみましょう。Maybe 値が左、関数が右にくるように、中置演算子表記で使ってみましょう。

```
ghci> Just 3 `applyMaybe` \x -> Just (x+1)
Just 4
ghci> Just "smile" `applyMaybe` \x -> Just (x ++ " :)")
Just "smile :)"
ghci> Nothing `applyMaybe` \x -> Just (x+1)
Nothing
ghci> Nothing `applyMaybe` \x -> Just (x ++ " :)")
Nothing
```

この例では、Just 値と関数を引数に applyMaybe を呼び出したときは、単に Just の中の値に関数が適用されていますね。Nothing 値と関数を引数に呼び出すと、全体の結果が Nothing になっています。では、関数のほうが Nothing を返す場合はどうでしょう？

```
ghci> Just 3 `applyMaybe` \x -> if x > 2 then Just x else Nothing
Just 3
ghci> Just 1 `applyMaybe` \x -> if x > 2 then Just x else Nothing
Nothing
```

確かに期待どおりの結果ですね。applyMaybe の左辺のモナド値が Nothing である場合には、全体の結果は Nothing になります。それから右辺の関数が Nothing を返した場合も、結果は Nothing です。これは Maybe をアプリカティブとして使ったときの挙動とよく似ていますね。式のどこかに Nothing があったら、答も Nothing になるという挙動です。

どうやら「変な値」に「普通の値を取って変な値を返す関数」を適用する方法が見えてきましたね。この例では「Maybe は失敗したかもしれない計算を表す」

というイメージを心に描いて設計することで、これが実現できました。

「で、これって何が便利なの？」という疑問が聞こえてきそうです。実際、アプリカティブファンクターのほうがモナドよりも強力に思えるかもしれません。だってアプリカティブファンクターは、ごく普通の関数を文脈付きの値に適用できるようにしてくれるのですから。ところがどっこい、モナドだって同じことができるんです。なにしろモナドはアプリカティブファンクターのアップグレード版ですから。さらに、モナドにはできてアプリカティブファンクターにはできないこともあるんです。この章ではそんなモナドの力の秘密を学んでいきましょう。

Maybe モナドは少し置いといて、まずは、モナドが属する型クラスを見ていきましょう。

13.3　Monad 型クラス

ファンクターには Functor 型クラスがあり、アプリカティブファンクターには Applicative 型クラスがあるように、モナドにも型クラスがあります。その名も Monad ！ って、まんまやん！

```
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y

    fail :: String -> m a
    fail msg = error msg
```

最初の行は **class** Monad m **where** となっています。あれ、でもモナドはアプリカティブファンクターの強化版のはずでは？ Monad のインスタンスは必ず Applicative のインスタンスでもあるよう、クラス宣言の前に、**class** (Applicative m) => Monad m **where** という型クラス制約があるべきではないでしょうか。うむ、確かにそうなんです。でも、Haskell の誕生当時には、アプリカティブファンクターが Haskell と相性がいいとは誰も思わなかったのです。それでも、すべてのモナドはアプ

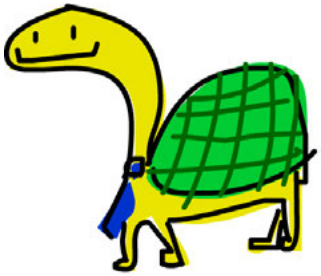
リカティブファンクターであることに間違いはありません。たとえ Monad のクラス宣言にそうは書いていないとしても。

Monad 型クラスの 1 つ目のメンバ関数は return です。これは Applicative 型クラスの pure と同じものです。だから、名前は違っていても、皆さんすでにお馴染みのはずです。 return の型は (Monad m) => a -> m a です。 return は、値を取って、その値を再現できるような最小のデフォルト文脈に入れます。return は、第 8 章で I/O を扱ったとき、すでに登場していましたね。そのときは、値を返すだけで何も I/O をしない、なんちゃって I/O アクションを作るために return を使ったのでした。 Maybe の場合は、 return は値を Just に入れて返します。

NOTE 繰り返しになりますが、Haskell の return は他の多くの言語にある return とは全然違うものです。 return は関数の実行を中断させる命令などではありません。普通の値を文脈に入れて返す、ただの関数です。

次の関数は >>= 、またの名をバインドです。これは関数適用に似ていますが、普通の値を取って通常の関数を適用するのではなく、モナド値（つまり、文脈付きの値）を取って、それに「通常の値を取るがモナド値を返す関数」を適用します。

それから、 >> があります。これにはデフォルト実装があるので、あまり詳しくは解説しません。 Monad インスタンスを実装するときも、このデフォルト実装を上書きすることは滅多にありません。 >> については 293 ページ「ロープの上のバナナ」の項で詳しく見ていきます。



Monad 型クラスの最後の関数は fail です。ユーザがコードの中から fail を呼び出すことは決してなく、もっぱら Haskell システムが呼び出します。 fail はモナド用の特別な構文において、パターンマッチに失敗してもプログラムを異常終了させず、失敗をモナドの文脈の中で扱えるようにするためのものです。これについては後で見ます。今は fail をさほど気にしなくても大丈夫です。

これで Monad 型クラスがどんなものか分かりました。では、 Maybe の Monad インスタンスの実装を見ていきましょう！

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f  = f x
    fail _ = Nothing
```

return は pure と同じですから、目をつぶっていても書けますね。Applicative 型クラスのときと同様、Just に包むだけです。関数 >>= は、さっきの applyMaybe と同じです。左辺の Maybe a を右辺の関数に食わせるとき、失敗するかもしれない計算という文脈の意味を考えれば、左辺に Nothing がきていれば Nothing を返す、Just がきていれば中身を取り出して f を適用する、という設計になります。

モナド化した Maybe で遊んでみましょう。

```
ghci> return "WHAT" :: Maybe String
Just "WHAT"
ghci> Just 9 >>= \x -> return (x*10)
Just 90
ghci> Nothing >>= \x -> return (x*10)
Nothing
```

1 行目は特に目新しい点はありません。僕らはもう Maybe の pure は使ったことがあるし、return は pure の別名にすぎないと知っているわけですから。

次の 2 つの行は >>= の用例です。Just 9 を \x -> return (x*10) に食わせたとき、x が 9 という値を取るのが分かりますか？ まるで、パターンマッチを使わずに Maybe から値を取り出せるかのようにも見えます。しかも、Maybe としての文脈は失われていません。その証拠に、左辺が Nothing に変わると、>>= の結果もちゃんと Nothing になります。

13.4　綱渡り

さて、失敗するかもしれない計算という文脈を損なうことなく、Maybe a 型の値に a -> Maybe b 型の関数を適用する方法が分かったところで、>>= を繰り返し使って Maybe a 値を返す複数の計算を扱う方法を見ていきましょう。

養魚場で働くピエールは休暇を取り、綱渡りに挑戦するようです。ピエールの綱渡りの腕前は、下手ではないという程度なのですが、1 つ悩みがありました。

バランス棒に鳥がとまりに来るんです！ 鳥たちは飛んできてはちょっと一休み。仲間とおしゃべりして、またパンくずを探しに飛んでいきます。バランス棒の左右にとまっている鳥の数が同じなら、さほど問題にはならないのですが、たまに鳥たちは片方の端に集まりたい気分になるらしく、そうなるとピエールはバランスが取れなくなって無様に転落してしまいます（ピエールは安全ネットを使っているので、怪我の心配はありません）。

さて、棒の左右にとまった鳥の数の差が 3 以内であれば、ピエールはバランスを取れるものとしましょう。例えば、右に 1 羽、左に 4 羽の鳥がとまっているなら大丈夫。だけど左に 5 羽目の鳥がとまったら、ピエールはバランスを崩して飛び降りる羽目になります。

では、鳥たちがバランス棒の左右の端に飛んできたり飛び去ったりするようすをシミュレートし、一定数の鳥たちが来たり去ったりした後も、ピエールが綱の上にいるかどうか判定するプログラムを書いてみましょう。例えば、まず左側に 1 羽の鳥が来て、それから右側に 4 羽来て、それから左側の鳥が飛び立った後、ピエールがどうなっているのかを知りたいわけです。

ひたすらコーディング

バランス棒は、単に整数のペアとして表現できます。ペアの第一成分は左側にいる鳥の数を、第二成分は右側にいる鳥の数を表すことにしましょう。

```
type Birds = Int
type Pole = (Birds, Birds)
```

まず、 `Int` の型シノニムを作り、 `Birds` と名づけます。これは、そこにいる鳥の数を表す整数です。それから `(Birds, Birds)` の型シノニムを `Pole` と名づけます（ポーランド系の苗字と混同しないでくださいね）。

では、鳥の数を取って、バランス棒の左側もしくは右側に鳥をとまらせる関数を作りましょうか。

```
landLeft :: Birds -> Pole -> Pole
landLeft n (left, right) = (left + n, right)

landRight :: Birds -> Pole -> Pole
landRight n (left, right) = (left, right + n)
```

試してみましょう。

```
ghci> landLeft 2 (0, 0)
(2,0)
ghci> landRight 1 (1, 2)
(1,3)
ghci> landRight (-1) (1, 2)
(1,1)
```

鳥を飛び立たせる処理は、負の数の鳥がとまる処理で代用します。 Pole に鳥をとまらせる関数は Pole を返すので、 landLeft と landRight は好きなだけ合成して使えます。

```
ghci> landLeft 2 (landRight 1 (landLeft 1 (0, 0)))
(3,1)
```

まず、 landLeft 1 を (0, 0) に適用すると (1, 0) になります。それから右側に鳥が 1 羽とまったら、 (1, 1) になります。さらに左側に鳥が 2 羽とまったら、 (3, 1) になります。Haskell の文法では、関数適用はまず関数を書いて、次に引数を書きますが、ここではどうも、バランス棒を先に書いて鳥をとまらせる関数が後のほうが読みやすそうですね。そこで、こんな関数を作ってみます。

```
x -: f = f x
```

これで関数を適用するのに、まず引数、次に関数を書けるようになります。

```
ghci> 100 -: (*3)
300
ghci> True -: not
False
ghci> (0, 0) -: landLeft 2
(2,0)
```

この形式を使えば、鳥を次々にとまらせる処理をさっきよりも読みやすく書けます。

```
ghci> (0, 0) -: landLeft 1 -: landRight 1 -: landLeft 2
(3,1)
```

実にスゴイ！ 先ほど普通の関数適用を使って書いたのと同じ処理ですが、より直感的になっていますね。この記法なら、まず (0, 0) から始めて、1 羽の鳥が左に、1 羽が右に、そして 2 羽が左にとまった、ということがすぐ分かります。

うわぁあああああ落ちるぅぅううううあああああ

ここまではよし。でも、片方に一気に 10 羽の鳥が来たら？

```
ghci> landLeft 10 (0, 3)
(10,3)
```

左に 10 羽もいるのに右には 3 羽しかいないですって？ 確実にピエールは宙を舞う羽目になります。この場合は明らかですが、じゃあ、こういう順番で鳥が来たらどうでしょう？

```
ghci> (0, 0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: ⇨
      landRight (-2)
(0,2)
```

どこにも問題なさそうに見えますが、気をつけて 1 ステップずつ評価してみれば、右側に 4 羽の鳥がいるけど左側には鳥がいない瞬間があることが分かるでしょう。危うく見逃すとこでした！ これを解決するためには、 landLeft 関数と landRight 関数に手を入れる必要があります。

landLeft 関数、 landRight 関数は、失敗を表現できる必要があります。ピエールがまだバランスを取れている間は新しいバランス棒の状態を返すが、鳥のとまり方が偏りすぎた場合には失敗を返すようにしたいのです。失敗を表現したいとなったら、これはもう Maybe を使うしかないですね！ さっそく書き直してみましょう。

```
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left, right)
    | abs ((left + n) - right) < 4 = Just (left + n, right)
    | otherwise                    = Nothing

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left, right)
    | abs (left - (right + n)) < 4 = Just (left, right + n)
    | otherwise                    = Nothing
```

landLeft 関数、 landRight 関数は、 Pole でなくて Maybe Pole を返すようになりました。鳥の数と、更新前のバランス棒の状態を引数に取るのは以前と同様ですが、たくさん鳥がきたときにはバランスを失ったピエールを放り出す検査が入るようになりました。このコードでは、ガード記法を使って、更新後の左右の鳥の数の差が 4 より小さいか判定しています。もし 4 より小さいなら、新しいバランス棒の状態を Just に包んで返します。差が 4 以上になった場合は、失敗を意味する Nothing を返します。

さっそく使ってみましょう。

```
ghci> landLeft 2 (0, 0)
Just (2,0)
ghci> landLeft 10 (0, 3)
Nothing
```

ピエールを落っことさずにうまく鳥が置けたら、新しいバランス棒の状態が Just に入って返ってきます。しかし、片方に鳥がとまりすぎてしまうと、 Nothing が返ってきます。素晴らしい改良です！ が、鳥をとまらせる操作を合成する能力を失ってしまったのではないでしょうか？ もう、 landLeft 1 (landRight 1 (0, 0)) とは書けなくなりました。だって、 Pole 型の (0, 0) に landRight 1 を適用すると、 Pole 型ではなく、 Maybe Pole 型が返ってくるからです。 landLeft 1 は Pole 型を取る関数なので、 Maybe Pole は受け付けてくれません。

何とかして、 Pole を取って Maybe Pole を返す関数に Maybe Pole を渡したいものです。幸運なことに、 >>= を使えば、 Maybe に対してまさにその操作ができます。やってみましょう。

```
ghci> landRight 1 (0, 0) >>= landLeft 2
Just (2,1)
```

landLeft 2 は Pole -> Maybe Pole 型の関数でしたね。 landRight 1 (0, 0) の結果である Maybe Pole は、そのままでは landLeft 2 に食わせられないので、 >>= を使って、文脈を保ったまま landLeft 2 に渡します。 >>= はまさに、 Maybe を「文脈付きの値」として扱うための道具だといえるでしょう。現に、 Nothing を landLeft 2 に渡すと、結果は Nothing となり、失敗が伝搬しているのが分かります。

```
ghci> Nothing >>= landLeft 2
Nothing
```

このように、通常の値を引数に取る関数にモナド値を渡せる >>= を使うことで、「鳥がとまる」という失敗するかもしれない操作を合成できるようになりました。鳥がとまる一連のイベントをこのように表現できます。

```
ghci> return (0, 0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

まず return を使って、バランス棒の状態を Just でくるみました。直接 (0, 0) に landRight 2 を適用するところから始めてもよかったのですが（それでも結果は同じです）、すべての関数に >>= を使うほうが統一感がありますよね。 Just (0, 0) が landRight 2 に食われて Just (0, 2) になります。それがまた landLeft 2 に食われて Just (2, 2) になり……、という具合に続きます。

そういえば、ピエール・シミュレーターに「失敗するかもしれない計算という文脈」を導入する前に出てきたこの例を覚えていますか？

```
ghci> (0, 0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: ⇨
      landRight (-2)
(0,2)
```

あのときは、ピエールと鳥たちの相互作用をうまく表現できませんでした。式の途中でピエールはバランスを崩すべきであるにもかかわらず、結果はそれを反映していません。通常の関数適用の代わりにモナド適用（ >>= ）を使うことで、これを修正しましょう。

```
ghci> return (0, 0) >>= landLeft 1 >>= landRight 4 >>= ⇨
      landLeft (-1) >>= landRight (-2)
Nothing
```

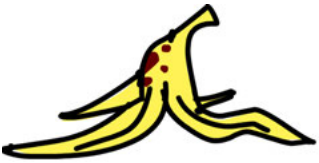
最終結果はちゃんと「失敗」になっています。なぜこの結果が得られたのか見ていきましょう。

1. まず、return が (0, 0) をデフォルトの文脈に入れることで Just (0, 0) ができます。
2. Just (0, 0) >>= landLeft 1 が評価されます。Just (0, 0) は Just 値なので、landLeft 1 が (0, 0) に適用され、Just (1, 0) を返します。この鳥の数なら、ピエールは余裕でバランスがとれます。
3. Just (1, 0) >>= landRight 4 が評価されて、Just (1, 4) を返します。ピエールはまだ何とかバランスを保っています。
4. Just (1, 4) は landLeft (-1) に食われます。これは landLeft (-1) (1, 4) が評価されるということです。もはやバランスは崩壊し、landLeft は Nothing を返します。
5. さて、この Nothing は landRight (-2) に食われますが、入力が Nothing なので、結果も自動的に Nothing になります。なにしろ landRight (-2) を適用する相手がいないからです。

Maybe をアプリカティブ値として扱うだけでは、ここまでのことは不可能だったでしょう。無理にやろうとしても、アプリカティブファンクターには、アプリカティブ値どうしをこのように深く相互作用させることはできないので、行き詰まります。アプリカティブにできることは、せいぜい通常の関数にアプリカティブ・スタイルで引数を渡すことくらいなのです。

アプリカティブ演算子たちは、アプリカティブ値からそれぞれの文脈に従って結果を取り出しては通常の関数を適用し、結果をまたアプリカティブ値に入れて返してくれます。しかし、アプリカティブ値どうしを相互作用させることは苦手なのです。ところが、今回のピエールと鳥の例では、各ステップは以前のステップの結果に依存しています。鳥たちがやってくるたびに、直前のステップを踏まえて今回の結果が検証され、バランスがとれているかどうか調べます。こうして鳥がうまくとまれたかどうかが判定されるのです。

ロープの上のバナナ



さて、今度はバランス棒にとまっている鳥の数によらず、いきなりピエールを滑らせて落っことす関数を作ってみましょう。この関数を banana と呼ぶことにします。

```
banana :: Pole -> Maybe Pole
banana _ = Nothing
```

この関数は鳥をとまらせる関数と混ぜて使えます。 banana は引数に何を渡されようと、無視して失敗を返すようにできているので、 banana を呼べば必ずピエールを落っことせます。

```
ghci> return (0, 0) >>= landLeft 1 >>= banana >>= landRight 1
Nothing
```

上の例では、 banana に渡るのは Just (1, 0) というかなり良いバランスの値ですが、banana はおかまいなしに Nothing を返すので、以降すべての結果は Nothing になってしまいます。残念でした！

ところで、入力に関係なく既定のモナド値を返す関数だったら、自作せずとも >> 関数を使うという手があります。これが >> のデフォルトの実装です。

```
(>>) :: (Monad m) => m a -> m b -> m b
m >> n = m >>= \_ -> n
```

普通の関数なら、引数を無視して既定の値を返すような関数の結果は、その既定値そのものです。ところが、モナド値を扱う場合は、モナドとしての文脈と意味を考慮する必要があります。 Maybe 版の >> の動作は、こんな感じです。

```
ghci> Nothing >> Just 3
Nothing
ghci> Just 3 >> Just 4
Just 4
ghci> Just 3 >> Nothing
Nothing
```

どうですか？ 1 番目の例では、 Maybe モナドの文脈が考慮された結果、「規定値」であるはずの Just 3 が失敗に置き換わっています。 >> を >>= _ -> で置き換えてみれば、何が起こっているのか簡単に理解できますよ。

>>= で連結した処理中での banana 関数は、 >> と Nothing という、失敗することが保証された組み合わせで置き換えられます。

```
ghci> return (0, 0) >>= landLeft 1 >> Nothing >>= landRight 1
Nothing
```

ところで、 Maybe を失敗の文脈付きの値として扱って関数に食わせるという賢明な選択をしなかったら、どうなっていたでしょう？ バランス棒に鳥をとまらせる一連の処理は、このようになったはずです。

```
routine :: Maybe Pole
routine = case landLeft 1 (0, 0) of
    Nothing -> Nothing
    Just pole1 -> case landRight 4 pole1 of
        Nothing -> Nothing
        Just pole2 -> case landLeft 2 pole2 of
            Nothing -> Nothing
            Just pole3 -> landLeft 1 pole3
```

まずバランス棒の左側に鳥を 1 羽とめ、成功したか失敗したかによって場合分けをします。失敗していた場合には、 Nothing を返します。

成功していた場合は、右側に鳥をとめる処理に進み、また場合分けをして……、という繰り返しです。この巨大で醜いコードを、 >>= による素敵なモナド適用の連鎖で書き直すのは、 Maybe モナド布教コードの定番です。 Maybe モナドを使うと、失敗するかもしれない処理が連続するコードをとても簡潔に書けるのです。

Maybe モナドの >>= の実装は、まさにこの「値が Nothing かどうかを判定し、その知識に基づいて動作を変える」というロジックになっています。もし入力値が Nothing なら、 >>= は直ちに Nothing を返します。入力が Nothing でなければ、 Just の中身を利用して計算を進めます。

この節では、関数の返り値を失敗処理をサポートする値に変えることで、関数がもっと便利になる例を見ました。関数の返り値を Maybe 値にし、関数適用を >>= に変えるだけで、ほとんど手間をかけずに失敗を処理するメカニズムを組み込むことができました。これも、 >>= が、関数を値に適用するにあたって、値が持っている文脈を保存するようにできているおかげです。 Maybe の場合、文脈は「この値は計算に失敗しているかもしれない」というものでした。ですので、Maybe 値に関数を適用すると、常に失敗の可能性が考慮されたコードが自動的に出来上がるのです。

13.5　do 記法

Haskell にとってモナドはとても便利なので、モナド専用構文まで用意されています。その名は **do** 記法。 **do** 記法は、すでに第 8 章で、複数の I/O アクションを 1 つに糊付けするときに使いましたね。実は、 **do** 記法は `IO` モナドだけじゃなくあらゆるモナドに使えます。といっても基本は同じで、 **do** 記法は複数のモナド値を糊付けするものです。

モナドを使ったお馴染みのこんなコードを見てください。

```
ghci> Just 3 >>= (\x -> Just (show x ++ "!"))
Just "3!"
```

もう見飽きたって？ 確かに、モナド値を関数に渡して、モナド値が返ってという、何の変哲もないコードです。さて、このコードを実行すると、ラムダ式の中の `x` に `3` が入りますよね。このラムダ式の中では、 `3` はモナド値でなく通常の値として扱えるわけです。では、このラムダ式の中にもう 1 つ `>>=` があったら、どうなるでしょう？ やってみましょう。

```
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Just "3!"
```

おー、 `>>=` の入れ子構造だ！ 外側のラムダ式の中では、 `Just "!"` をラムダ式 `\y -> Just (show x ++ y)` に食わせています。このラムダ式の中では、 `y` は `"!"` になります。また、 `x` は、外側のラムダ式が実行されたときに `3` が入ったままです。これを見ていると **let** 構文を思い出しますね。

```
ghci> let x = 3; y = "!" in show x ++ y
"3!"
```

この 2 つの例は似ていますが、 `>>=` のほうで使っている値はモナド値であるという大きな違いがあります。失敗の文脈付きの値なのです。ですから、好きな箇所を失敗で置き換えられます。

```
ghci> Nothing >>= (\x -> Just "!" >>= (\y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (\x -> Nothing >>= (\y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (\x -> Just "!" >>= (\y -> Nothing))
Nothing
```

最初の例では `Nothing` を関数に渡しているので、当然 `Nothing` が返っています。次の例では、 `Just 3` を関数に渡して、 `x` が `3` になっています。ところが内側のラムダ式には `Nothing` を渡していますから、そいつは `Nothing` を返し、そのせいで外側のラムダ式の結果も `Nothing` になっています。やっぱりこの操作

は **let** 式で値を変数に束縛する操作に似ていますね。ただ、操作したい値がモナド値である、という違いはあります。

この類似をもっと明確にするために、今の式をスクリプト風に書き直しましょう。 Maybe 値ごとに 1 行を使います。

```
foo :: Maybe String
foo = Just 3   >>= (\x ->
      Just "!" >>= (\y ->
      Just (show x ++ y)))
```

面倒なラムダ式をいっぱい書かなくて済むように、Haskell には **do** 記法が備わっています。 **do** 記法を使えば、さっきのコードはこう書けます。

```
foo :: Maybe String
foo = do
    x <- Just 3
    y <- Just "!"
    Just (show x ++ y)
```

いちいち Maybe 値が Just か Nothing かなんて場合分けせずとも、Maybe 値から生の値が取り出せる気分になれます。素晴らしい！ もし、中身を取り出そうとする値のいずれかが Nothing だったら、 **do** 式全体も Nothing になります。この **do** 式の中では、 Maybe モナドから（存在するとおぼしき）値を取り出しつつ、値に付きまとう文脈の処理は >>= に任せているのです。

モナド値を連鎖させた式と等価なものを、ずっと簡潔に表せる記法が **do** 式なのです。

do 自由自在

do 式は、 **let** 行を除いてすべてモナド値で構成されます。モナドの結果を調べるには <- を使います。例えば、変数を <- で Maybe String の結果に束縛すれば、その変数の型は String になります。これは >>= を使ってモナド値をラムダ式に食わせたときとまったく同じです。

do 式の最後のモナド値（上の例でいうと Just (show x ++ y)）だけは <- で結果を束縛できません。 **do** 記法は >>= とラムダ式を使う表記に書き直せるはずですが、最後のモナドで <- を使っても、それを受けるべきラムダ式がないからです。その代わり、最後のモナドの返り値は、それまでの失敗の可能性をすべて踏まえた上で、 **do** 式で糊付けしたモナド全体の結果になります。例えばこのコードを見てください。

```
ghci> Just 9 >>= (\x -> Just (x > 8))
Just True
```

>>= の左辺は Just なので、ラムダ式は 9 に適用され、結果は Just True になっています。これを do 記法で書き直すと、こうなります。

```
marySue :: Maybe Bool
marySue = do
    x <- Just 9
    Just (x > 8)
```

2 つを見比べると、do 記法で連鎖させた最後のモナドの結果が do 式全体の結果になる仕組みがよく分かると思います。

帰ってきたピエール

ピエールの綱渡りの動作も、もちろん do 記法で書けます。 landLeft と landRight は鳥の数とバランス棒を引数に取り、新しい棒の状態を Just に包んで返します。ただしピエールが滑った場合は Nothing を返します。各ステップは、それまでの結果に依存しているので、 >>= を使って連結し、各ステップからの失敗の文脈が累積するようにします。例えば、左に 2 羽、右に 2 羽、それから左に 1 羽の鳥が来る処理はこう書けます。

```
routine :: Maybe Pole
routine = do
    start <- return (0, 0)
    first <- landLeft 2 start
    second <- landRight 2 first
    landLeft 1 second
```

さて、ピエールは無事でしょうか？

```
ghci> routine
Just (3,2)
```

無事でした！

ピエールの動作を >>= を明示的に使って書いていたときには、 return (0, 0) >>= landLeft 2 >>= landRight 2 などと書くのが常でした。 landLeft 2 は Maybe 値を返す関数だからです。ところが、 do 式を使う場合は各行がモナド値である必要があるので、直前の Pole の状態に名前をつけて landLeft 関数や landRight 関数に明示的に渡さないといけません。 Maybe 値を束縛した変数の中身を調べてみれば、 start には (0, 0) 、 first には (2, 0) が入っているはずです。

1 行ごとに処理が書いてある do 式が、手続き型言語のプログラムに見える人もいるかもしれません。でも do 式では、それ以前の行の結果に依存する値の列

が、成功または失敗の文脈とともに書いてあるにすぎません。

Maybe のモナドとしての側面を利用しなかったら、このコード片がどんなふうになっていたか、もう一度見てみましょう。

```
routine :: Maybe Pole
routine =
    case Just (0, 0) of
        Nothing -> Nothing
        Just start -> case landLeft 2 start of
            Nothing -> Nothing
            Just first -> case landRight 2 first of
                Nothing -> Nothing
                Just second -> landLeft 1 second
```

成功した場合は Just (0, 0) の中のタプルが start になり、 landLeft 2 start の結果が first になり……、という流れが分かりますか？

ピエールにバナナの皮を踏ませたい場合、 do 記法ではこう書きます。

```
routine :: Maybe Pole
routine = do
    start <- return (0, 0)
    first <- landLeft 2 start
    Nothing
    second <- landRight 2 first
    landLeft 1 second
```

do 記法の中で、 <- による変数への束縛をせずにモナドを使うと、結果を無視したいモナドの後に >> を付けたのと同じことになります。モナドをつなげるけれども結果には興味がないから無視するわけです。同じ処理は _ <- Nothing とも書けますが、 <- を省略した記法のほうがきれいでしょう？

いつ do を使い、いつ >>= を使うかの選択はあなた次第です。僕はピエールの例では >>= を使うほうがいいと思います。各モナドが、もっぱら直前のモナドの結果に依存しているからです。 do 記法では、どの時点のバランス棒に鳥をとまらせるのかを明示的に書かないといけません。常に 1 つ前のステップの結果を使うのであれば、 do を使うまでもないですよね。とはいえ、この例で do 記法に対する理解が多少深まったと思います。

パターンマッチと失敗

do 記法でモナド値を変数名に束縛するときには、 let 式や関数の引数のときと同様、パターンマッチが使えます。これが do 式におけるパターンマッチの例です。

```
justH :: Maybe Char
justH = do
    (x:xs) <- Just "hello"
    return x
```

ここでは、パターンマッチを使って文字列 "hello" の先頭の文字を取り出し、それをモナド全体の結果としています。ですから justH は Just 'h' と評価されます。

もし、このパターンマッチが失敗したらどうなるのでしょう？ 関数定義では、あるパターンマッチが失敗したら、次のパターンを試すことになっていました。ある関数のすべてのパターンマッチに失敗したら、エラーが投げられ、プログラムは異常終了します。これに対し、 **let** 式でパターンマッチに失敗した場合、**let** 式には複数のパターンマッチを次々に試す仕組みはないので、直ちにエラーになります。

do 式の中でパターンマッチが失敗した場合、 Monad 型クラスの一員である fail 関数が使われるので、異常終了という形ではなく、そのモナドの文脈に合った形で失敗を処理できます。 fail のデフォルト実装はこうです。

```
fail :: (Monad m) => String -> m a
fail msg = error msg
```

あ、デフォルトでは、 fail はプログラムを異常終了させるようですね。しかし、 Maybe のような失敗を表現できる文脈を持つモナドでは、通常は独自に fail を実装しています。 Maybe の fail の実装はこんな感じです。

```
fail _ = Nothing
```

エラーメッセージを無視して Nothing を作っています。これにより、 **do** 記法で書かれた Maybe モナドの内部でパターンマッチに失敗したときは、 **do** 式全体が Nothing を返します。プログラムが異常終了するよりも、 Nothing が得られたほうがうれしいですよね。パターンマッチに失敗するような **do** 式を書いてみましょう。

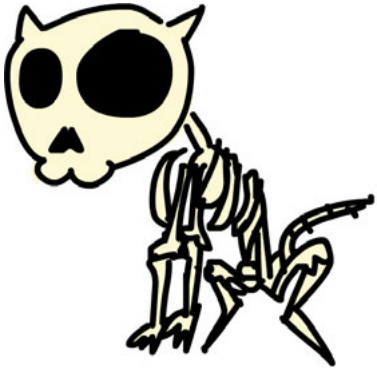
```
wopwop :: Maybe Char
wopwop = do
    (x:xs) <- Just ""
    return x
```

このパターンマッチは失敗するので、その効果は失敗するパターンマッチのある行全体を Nothing で置き換えたのと同じです。試してみましょう。

```
ghci> wopwop
Nothing
```

このように、パターンマッチが失敗しても全プログラムが巻き添えになって落ちることはなく、モナドの文脈で失敗が発生するだけで済んでいます。実に素晴らしい。

13.6 リストモナド



これまでのところ、 Maybe は失敗の文脈の付いた値として解釈できること、 >>= を使って Maybe 値を関数に食わせることで失敗処理を簡単に記述できることを見てきました。この節では、リストのモナドとしての側面を使うことで、非決定性を伴うコードをきれいに読みやすく書く方法を見ていきます。

第 11 章では、アプリカティブとしてのリストは非決定性計算を表すといいました。

例えば、 5 という値は決定的です。つまり、どう評価してもただ 1 つの決まった計算結果にしかならず、その値は既知です。一方、 [8,9,3] のような値は複数の計算結果を含んでいるとも、複数の候補値を同時に重ね合わせたような 1 つの値であるとも解釈できます。リストをアプリカティブ・スタイルで使うと、非決定性を表現していることがはっきりします。

```
ghci> (*) <$> [1,2,3] <*> [10,100,1000]
[10,100,1000,20,200,2000,30,300,3000]
```

左辺リストの要素と右辺リストの要素の、すべてのあり得る組み合わせの積が、答のリストに含まれています。非決定性計算ではたくさんの選択肢に出くわしますが、その都度すべてを試します。すると、最終結果はもっとたくさんの候補値を含んだ非決定的値になるわけです。

この非決定性計算という文脈は、うまくモナドに焼き直すことができます。リストの Monad インスタンスは、こう書けます。

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

ご存知のとおり、 return は pure と同じなので、リストの return が何なのかも分かりますね。 return は値を取って、その値を再現できるような最小限の文脈に入れます。というわけで、 return は引数の値が 1 つだけ入っているようなリストを作って返すわけです。 return は、普通の値をリストにくるんで非決定な値に混ぜたいときに便利です。

>>= は「文脈付きの値（モナディックな値）」を、「通常の値を取って文脈付きの値を返す関数」に食わせる演算でしたね。もし関数が文脈付きの値でなく通常の値を返すものだったとしたら、 >>= はここまで便利じゃなかったでしょう。一度使うと、文脈はなくなってしまいますから。

それでは、非決定的値を関数に食わせてみましょう。

```
ghci> [3,4,5] >>= \x -> [x,-x]
[3,-3,4,-4,5,-5]
```

Maybe に >>= を使うと、失敗の可能性を考慮しながら、モナド値を関数に供給できました。今度はモナドが非決定性計算を処理してくれています。

[3,4,5] は非決定的値であり、それを食わせている関数も非決定的値を返すようになっています。最終結果も非決定的であり、リスト [3,4,5] から 1 つの要素を選んで \x -> [x,-x] に食わせるすべての場合を尽くしています。関数のほうも、数を取って 2 通りの答を返します。1 つはそのまま、1 つは符号を変えて。 >>= を使ってさっきのリストをこの関数に食わせると、すべての数が両方の符号になって出てきます。ラムダ式の x は、リストの中の値を順に取ります。

この答がどうして得られるのか、実装を追いかけてみましょう。はじめに、[3,4,5] があります。それをラムダ式で写すとこうなりますね。

```
[[3,-3],[4,-4],[5,-5]]
```

ラムダ式がそれぞれの要素に作用した結果、二重リストができています。そしてこのリストは concat されて、じゃじゃーん！ 非決定性関数が非決定的値に適用できました！

非決定性計算は、失敗する可能性のある計算の上位互換になっています。空リスト [] は、返すべき値が何もない、ということなので、 Nothing とよく似ています。だから、失敗は空リストで表せばよいのです。 Maybe と同様、 fail のエラーメッセージは無視されます。では、失敗するリストで遊んでみましょう。

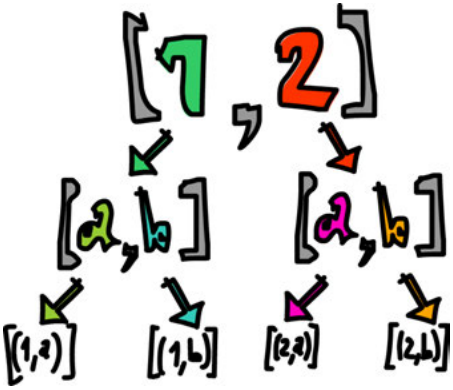
```
ghci> [] >>= \x -> ["bad","mad","rad"]
[]
ghci> [1,2,3] >>= \x -> []
[]
```

1 行目の例では、ラムダ式に空リストを食わせています。くうかい？ リストには要素がないので、次段の関数に渡すものがありません。ですから、結果は空

リストになります。これは Nothing を関数に渡したときと同じですね。2 行目の例は、3 つの要素が関数に渡されますが、関数のほうが引数を無視して空集合を返しています。とにかく関数のほうが何を与えても失敗してばかりなので、全体の結果も失敗となっています。

Maybe のときと同様、 >>= を使えばリストをいくつでも連結して非決定性を伝播させることができます。

```
ghci> [1,2] >>= \n -> ['a','b'] >>= \ch -> return (n, ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```



[1,2] から出てきた数が n に束縛され、 ['a','b'] から出てきた文字が ch に束縛されます。それから return (n, ch) をすると（ここは [(n, ch)] と書いてもよい）、ペア (n, ch) がデフォルトの文脈に入ります。この場合 return のルールは、「(n, ch) の情報を含んでいる、できる限り短いリストを返しなさい、非決定性をなるべく小さくしなさい」です。そうすれば、文脈の変化も最小限になります。このプログラムは、「[1,2] の各要素 n 、 ['a','b'] の各要素 ch に対して、タプルを作りなさい」という意味になっています。

一般的に言って、 return は値を最小限の文脈に包むので、文脈に余計な影響（Maybe だったら失敗したり、リストだったら場合の数が増えたり）を及ぼさず、ただ何らかの結果を提示するだけです。

非決定性計算を組み立てたものは、場合分けの木構造とみなすことができます。リストが表現する可能な選択肢のそれぞれが木構造の枝分かれに対応しています。これが、さっきの例を do 記法で書き直したものです。

```
listOfTuples :: [(Int, Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n, ch)
```

こう書くと、 n は [1,2] から、 ch は ['a','b'] から 1 つ選んだものだというのがはっきりしますね。 Maybe のときと同じで、モナド値から要素を取り出

して普通の値であるかのように扱えています。そして、裏では >>= が文脈の面倒を見てくれています。この場合の文脈は非決定性です。

do 記法とリスト内包表記

リストの do 記法を見てピンときたあなた。これを見てください。

```
ghci> [ (n, ch) | n <- [1,2], ch <- ['a','b'] ]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

そう、リスト内包表記です！ do 記法では、n は [1,2] の中の値を順に取り、その各々に対して ch は ['a','b'] のどれかになり、そして最後の行は (n, ch) をデフォルト文脈（単一要素リスト、非決定性が増えない）に入れます。リスト内包表記でも同じことが起こっていますが、最後の (n, ch) を return で包む操作は、リスト内包表記の出力パートがやってくれるので省かれています。

実は、リスト内包表記はリストモナドの構文糖衣にすぎないのです。リスト内包表記も do 記法も、内部では >>= を使った非決定性計算に変換されています。

MonadPlus と guard 関数

さて、リスト内包表記では、出力する要素を選別（filter）することができました。例えば 7 の付く数だけを選んで出すには、

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

というふうに数 x を show で文字に変え、それから文字 '7' が含まれているか調べます。

この選別はどんなリストモナドに翻訳されるのでしょう？ それを知るには、guard 関数と MonadPlus 型クラスを学ぶ必要があります。

MonadPlus は、モノイドの性質をあわせ持つモナドを表す型クラスです。これがその定義です。

```
class Monad m => MonadPlus m where
    mzero :: m a
    mplus :: m a -> m a -> m a
```

mzero は、Monoid 型クラスでいう mempty に対応する概念で、mplus は mappend に対応しています。リストはモノイドでもありますから、MonadPlus のインスタンスにできます。

```
instance MonadPlus [] where
    mzero = []
    mplus = (++)
```

リストに関する mzero は、候補が 1 つもない、失敗した非決定性計算を表しています。 mplus は 2 つの非決定的値を 1 つの値にくっつけます。 guard 関数の定義はこんな感じです。

```
guard :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero
```

guard は真理値を引数に取ります。引数が True なら、 guard は () を成功を表す最小限の文脈に入れます。引数が False なら、 guard は失敗したモナド値を作ります。このような動作をします。

```
ghci> guard (5 > 2) :: Maybe ()
Just ()
ghci> guard (1 > 2) :: Maybe ()
Nothing
ghci> guard (5 > 2) :: [()]
[()]
ghci> guard (1 > 2) :: [()]
[]
```

面白そうですね。でもどこが便利なのでしょう？ リストモナドでは、 guard を使って解の候補をふるい落とすことができます。

```
ghci> [1..50] >>= (\x -> guard ('7' `elem` show x) >> return x)
[7,17,27,37,47]
```

この計算結果は、前にリスト内包表記でやったのと同じです！ どうして guard を使ってできたのでしょう？ まずは guard 関数を >> につなぐと何が起こるか見てみましょう。

```
ghci> guard (5 > 2) >> return "cool" :: [String]
["cool"]
ghci> guard (1 > 2) >> return "cool" :: [String]
[]
```

guard が成功すれば、空タプルの入ったモナドが返ってきます。そこですかさず >> を使えば、その空タプルを無視し、何か別のものを返すことができます。ところが、その guard が失敗したら、後ろの return もつられて失敗します。空リストを >>= に食わせたら、答は必ず空リストになるからです。 guard は、ぶっちゃけ「引数が False なら直ちに失敗を投げよ。 True ならダミーの値 () が入っている成功を作れ」と言っているのです。 guard がやっているのは、計算を続けてよいかの判断です。

さっきの例を do 記法で書き直したものがこれです。

```
sevensOnly :: [Int]
sevensOnly = do
    x <- [1..50]
    guard ('7' `elem` show x)
    return x
```

ここでもし、最後に return を使って x を返すのを忘れたとすると、計算結果はただの空タプルのリストになってしまいます。最後に、リスト内包表記を使ったコードをもう一度挙げておきます。

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

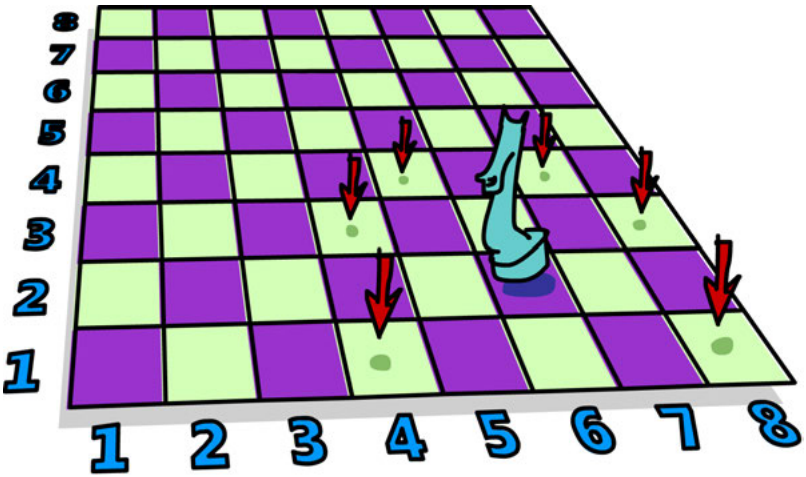
というわけで、リスト内包表記でフィルタを使うのは guard と同値なのです。

騎士の旅

ここで、非決定性計算を使って解くのにうってつけの問題をご紹介しましょう。チェス盤の上にナイトの駒が 1 つだけ乗っています。ナイトを 3 回動かして特定のマスまで移動させられるか、というのが問題です。チェス盤上でのナイトの位置は、単なる数のペアで表現することにしましょう。1 つ目の数が横軸、2 つ目が縦軸です。

ナイトの現在位置を表す型シノニムを作りましょう。

```
type KnightPos = (Int, Int)
```



さて、ナイトが (6, 2) から出発したとします。果たしてナイトは、ちょうど 3 手で (6, 1) に移動できるでしょうか？ (6, 1) に行くための最善手は？ それが分かったら、条件を満たす手順をすべて求めてみましょう！ せっかく非決定性計算を自由に使えるようになったんだから、解の候補の中から 1 つだけ選ぶのではなく、全部採用してみましょうよ。これが、ナイトの現在位置を取って次に行ける位置を列挙する関数です。

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = do
    (c', r') <- [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
                ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
                ]
    guard (c' `elem` [1..8] && r' `elem` [1..8])
    return (c', r')
```

ナイトは水平に 1 マス動いたあと垂直に 2 マス、あるいは垂直に 1 マス動いたあと水平に 2 マス、という動きが 1 手でできます。ペア (c', r') は、動きのリストにある値を順番に取り、guard は、(c', r') が盤面の範囲内であることを保証しています。盤面からはみ出していたら、guard は空リストを生み出すので、(c', r') という位置は以降の計算では使われません。

この関数はリストモナドを使わずに書くこともできます。例えば、filter を使うならこう書けます。

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c, r) = filter onBoard
    [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
    ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
    ]
    where onBoard (c, r) = c `elem` [1..8] && r `elem` [1..8]
```

どちらの実装も同じことをするので、気に入ったほうを使ってください。では、動かしてみましょう。

```
ghci> moveKnight (6, 2)
[(8,1),(8,3),(4,1),(4,3),(7,4),(5,4)]
ghci> moveKnight (8, 1)
[(6,2),(7,3)]
```

ちゃんと動きました！ 魔法みたい！ ある位置を入力に取って、可能な動きのすべてを同時に行っているようなものです。

さて、次の位置は非決定的値で得られましたから、>>= を使えば、また moveKnight にぶちこめますね。これが、初期位置を取って、3 手でいける位置をすべて返してくれる関数です。

```
in3 :: KnightPos -> [KnightPos]
in3 start = do
    first <- moveKnight start
    second <- moveKnight first
    moveKnight second
```

この関数に (6, 2) を渡して返ってくるリストはかなり大きいものです。なぜなら、同じ位置に 3 手で行く経路が複数ある場合、その位置が複数回リストに登場するからです。

先ほどのコードを do 記法を使わずに書くと、こうなります。

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

最初の >>= は、start から可能な手をすべて列挙してくれます。次の >>= は、1 手目に可能な動きのそれぞれに対して、次に可能な手を列挙。最後の >>= も同様です。

return を使って値をデフォルトの文脈に入れた上で、>>= を使ってそれを関数に食わせるという一連の記述は、単に関数を値に適用するのと等価なのですが、ここではスタイルを統一するために return と >>= を使ってみました。

では、2 つの位置を取って、1 つ目の位置から 2 つ目の位置にちょうど 3 手で到達できるか教えてくれる関数を作りましょう。

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

まず、3 手で行ける位置をすべて生成してから、目的地がその中に含まれているか調べています。例えば、(6, 2) から (6, 1) へ 3 手で行けるかどうか試すには、こうです。

```
ghci> (6, 2) `canReachIn3` (6, 1)
True
```

行けました！ では、(6, 2) から (7, 3) では？

```
ghci> (6, 2) `canReachIn3` (7, 3)
False
```

ダメでした！ 読者への演習問題として、始点と終点を与えるとどういう経路を取ればいいか教えてくれるよう、この関数を改造してみてください。なお、第 14 章では、この関数に手数をハードコードするんじゃなくて、手数の条件も引数で渡せるように改造する方法を紹介します。

13.7 モナド則

ファンクターやアプリカティブファンクターと同じく、モナドにも、すべてのモナドインスタンスが守るべき法則がいくつかあります。ある型が Monad のインスタンスになっているからといって、実際にモナドであるわけではありません。ある型が真にモナドであるためには、その型はモナド則を満たさねばなりません。モナド則があることで、モナドになっている型や、型の振る舞いについて合理的な想定ができるようになります。

Haskell では、型検査が通る限り、任意の型を任意の型クラスのインスタンスにできてしまいます。ある型がモナド則を満たしているかなどの検査はしてくれないので、 Monad 型クラスのインスタンスを自作する場合には、その型がモナド則を満たしていることを自分で保証しないといけません。標準ライブラリに含まれる型はモナド則を満たしていると信頼できますが、自分でモナドを作るときには、モナド則が成り立っているか手動で確認しないといけません。もっとも、モナド則はそんなに難しくないので、ご心配なく。

左恒等性

第一のモナド則が言っているのは、 return を使って値をデフォルトの文脈に入れたものを >>= を使って関数に食わせた結果は、単にその値にその関数を適用した結果と等しくなりなさい、ということです。形式的に言えば、 return x >>= f と f x は等価である、ということです。

モナド値を文脈付きの値とみなし、 return は値をその値が再現できるような最小限のデフォルトの文脈に入れるものだとみなす立場からすれば、このモナド則は実に自然な発想です。 return が作る文脈が本当に最小限のものだというのなら、 return で作ったモナド値を関数に食わせるという操作は、単に関数を普通の値に適用する操作とほんの少ししか食い違わないでしょう。そして実際、食い違いはゼロなのです。

Maybe モナドに関しては、 return は Just に等しいと定義されています。 Maybe モナドとは、そもそも失敗する可能性のある計算を表現するモナドでしたから、そのような文脈に値を入れたいというのなら、その値を計算の成功として扱うのは自然なことです。返すべき値が分かっているわけですからね。これは Maybe モナドの return を使っている例です。

```
ghci> return 3 >>= (\x -> Just (x+100000))
Just 100003
ghci> (\x -> Just (x+100000)) 3
Just 100003
```

リストモナドの場合、 return は引数を単一要素リストに入れる関数です。そして >>= の実装は、リストの中のすべての要素に対して関数を適用するというものでした。ところが、単一要素リストには要素が 1 つしかありませんから、関数を値に直接適用したのと同じ結果になるわけです。

```
ghci> return "WoM" >>= (\x -> [x,x,x])
["WoM","WoM","WoM"]
ghci> (\x -> [x,x,x]) "WoM"
["WoM","WoM","WoM"]
```

そして IO モナドの場合には、 return を使うと、副作用がなく単に値を返すだけの I/O アクションを作れるのでしたね。というわけでこのモナド則は IO に関しても成り立つわけです。

右恒等性

モナドの第二法則は、 >>= を使ってモナド値を return に食わせた結果は、元のモナド値と不変であると言っています。式で書くと、 m >>= return はただの m である、ということです。

この法則は、第一法則ほど自明ではないかもしれません。なぜこれが成り立つべきなのか見ていきましょう。 >>= を使ってモナド値を関数に食わせるという式があった場合、関数のほうは普通の値を取ってモナド値を返す関数のはずです。 return も、型を見れば、そういう関数の一種であることが分かります。

return は、値をその値を返せるような最小限の文脈に入れるものです。これは、例えば Maybe モナドであれば決して失敗しないことを意味し、リストモナドであれば非決定性を増やさないことを意味します。

何種類かのモナドで試してみましょう。

```
ghci> Just "move on up" >>= return
Just "move on up"
ghci> [1,2,3,4] >>= return
[1,2,3,4]
ghci> putStrLn "Wah!" >>= return
Wah!
```

リストの場合、 >>= の実装はこうなっていました。

```
xs >>= f = concat (map f xs)
```

ということは、[1,2,3,4] を return に注ぎ込むと、まず return が [1,2,3,4] を写して [[1],[2],[3],[4]] ができます。それからこのリストが concat されて、元のリストに戻ります。

左恒等性と右恒等性は、いずれも基本的に return の振る舞いに関する法則です。return はモナドシステムの中で、通常の値をモナド値に変えるという重要な役割を担っており、return が作り出すモナド値が最小限でない余計な文脈を持っていたら困ってしまいます。

結合法則

最後のモナド則は、>>= を使ったモナド関数適用の連鎖があるときに、どの順序で評価しても結果は同じであるべき、というものです。式で書くと、(m >>= f) >>= g と m >>= (\x -> f x >>= g) が等価である、というものです。

むむむ、何を言っているのか分かりませんね。えーっと、m はモナド値で、f と g はモナド関数です。(m >>= f) >>= g と書くと、m を f に食わせることになります。その結果はモナド値です。で、そのモナド値を g に食わせます。一方、m >>= (\x -> f x >>= g) と書いたときは、x を取るラムダ式があって、それはモナド値 f x を g に食わせたものを返します。どうしてこの 2 つの式が等価なのかは分かりにくいですから、例を見て理解しましょう。

さて、鳥たちと友達で、綱渡りが趣味のピエールのことを覚えていますか？ピエールのバランス棒に鳥がとまりに来るようすをシミュレートするために、失敗するかもしれない関数の連鎖を作ったのでした。

```
ghci> return (0, 0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
Just (2,4)
```

この例では、まず Just (0, 0) があって、それにモナド関数 landRight 2 を適用しています。その結果はまた別のモナドであり、それがまた次のモナド関数に束縛され、と続いています。この式にあえて括弧を付ければ、このようになります。

```
ghci> ((return (0, 0) >>= landRight 2) >>= landLeft 2) >>= landRight 2
Just (2,4)
```

しかし、このルーチンはこう書くこともできるのです。

```
return (0, 0) >>= (\x ->
landRight 2 x >>= (\y ->
landLeft 2 y >>= (\z ->
landRight 2 z)))
```

return (0, 0) は Just (0, 0) と等価であり、これをラムダ式に食わせると x が (0, 0) になります。landRight は鳥の数とバランス棒の状態（数のタプル）を取る関数ですが、それらを無事受け取って Just (0, 2) を返します。それを次のラムダ式に渡すと、y が (0, 2) になります。このような計算が、最後の鳥が処理されて Just (2, 4) が生み出されるまで続きます。これが全体の計算結果になるわけです。

このようにモナド値をモナド関数に食わせる式を評価するとき、入れ子の順序はどうでもよく、ただ関数の意味だけが重要である、というのがモナド結合法則の言うところです。この法則を別の側面から見てみましょう。今、f と g という名の 2 つの関数があるとします。

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = (\x -> f (g x))
```

g の型が a -> b で、f の型が b -> c ならば、2 つの関数を結合し、g の返り値を f の引数に渡すことで、a -> c 型の新しい関数を作れます。では、この 2 つがモナド関数であったらどうでしょう？ 2 つの関数の返り値がモナド値であったら？ a -> m b 型の関数の返り値は、そのままでは b -> m c 型の関数に渡せませんね。後者は b 型の通常の値を期待しており、モナド値は受け取れないからです。ところが、>>= を使うとこれが可能になります。

```
(<=<) :: (Monad m) => (b -> m c) -> (a -> m b) -> (a -> m c)
f <=< g = (\x -> g x >>= f)
```

こうして、モナド関数を合成することが可能になりました！

```
ghci> let f x = [x,-x]
ghci> let g x = [x*3,x*2]
ghci> let h = f <=< g
ghci> h 3
[9,-9,6,-6]
```

これは素晴らしい。でも、これと結合法則にどんな関係が？ えーと、モナド結合法則をこの関数合成の法則として見た場合、f <=< (g <=< h) と (f <=< g) <=< h が等価である、という宣言になっています。これもまた、モナドの演算子を入れ子にする順番はどうでもいい、ということの別の表現になっているのです。

ここで、<=< を使って最初の 2 つの法則を書き直してみると、左恒等性は任意のモナド関数 f に対して f <=< return は f と等価であるという宣言、右恒等性は return <=< f もまた f と等価であるという宣言になっています。こうしてみると、3 つのモナド則は、f が通常の関数であった場合に (f . g) . h

と f . (g . h) が等価であり、f . id は常に f と等しく、id . f もただの f であることと非常に似ています。

この章では、モナドの基礎を見て、Maybe モナドとリストモナドの仕組みを学びました。次の章では、ほかにもたくさんあるすごいモナドを紹介していき、独自のモナドも作ってみます。

第14章
もうちょっとだけモナド

これまで、モナドが値を文脈に入れること、文脈の中で関数適用が行えること、`>>=` や **`do`** 記法を使えば Haskell が文脈の面倒を見てくれるので、プログラマは値を扱うのに集中できることを見てきました。

最初に出会った `Maybe` モナドは失敗する可能性という文脈を値に付加するものでした。次にリストモナドを学び、実に簡単に非決定性計算を導入できることを見ました。そういえば `IO` モナドも使ってましたね、しかも `IO` がモナドだったなんて知る前から！

この章には、さらにいくつかのモナドを紹介します。それらを使うことで、普通の値をモナド値として扱えばプログラムがどんなにきれいに書けるかを体感してください。モナドの世界を探検することで、モナドを認識し、使いこなすための感覚が磨かれていくことでしょう。

この章に登場するモナドは、すべて `mtl` パッケージの一部です（Haskell の**パッケージ**はモジュールの集まりです）。`mtl` パッケージは Haskell Platform に入っているので、たぶんもうあなたの環境にもインストール済みだと思いますが、`mtl` がインストールされているか調べるには、コマンドラインから **`ghc-pkg list`** と打ってください。するとインストールされている Haskell パッケージの一覧が出てきます。その中に `mtl` とバージョン番号が記された行があるはずです。

14.1　Writer？ 中の人なんていません！

これまで、Maybe モナド、リストモナド、そして IO モナドという武器をゲットしてきました。次は Writer です。直ちに装備したまえ！

Maybe モナドが失敗の可能性という文脈付きの値を表し、リストモナドが非決定性が付いた値を表しているのに対し、Writer モナドは、もう 1 つの値がくっついた値を表し、付加された値はログのように振る舞います。Writer モナドを使えば、一連の計算を行っている間、すべてのログが単一のログ値にまとめて記録されることを保証できます。最終的なログは、モナドの返り値にくっついて出てきます。

例えば、まあデバッグ目的とかで、何が起こっているのか説明する文字列を値にくっつけたいとしましょう。盗賊団の人数を引数に取り、それが大きな盗賊団であるかどうか判定する関数を考えてみてください。とてもシンプルな関数です。

```
isBigGang :: Int -> Bool
isBigGang x = x > 9
```

さて、ただ True か False を返すだけでなく、この関数が何をしたかを示す文字列も一緒に返してほしかったら、どうすればよいでしょう？ そうですね、返したい文字列を作って、Bool と一緒に返してやればよいです。

```
isBigGang :: Int -> (Bool, String)
isBigGang x = (x > 9, "Compared gang size to 9.")
```

これで、単に Bool を返すのではなく、第一要素が元来の返り値で、第二要素がそれに添えた文字列であるタプルを返すようになりました。値に文脈が付いたわけです。さあ、動かしてみましょう。

```
ghci> isBigGang 3
(False,"Compared gang size to 9.")
ghci> isBigGang 30
(True,"Compared gang size to 9.")
```

これで問題なさそうですよね。isBigGang は、普通の値を取って、文脈の付いた値を返します。ついさっき見たように、普通の値を渡すのには何の問題もありません。では、すでに文字列が付いている値、例えば (3, "Smallish gang.") を isBigGang に食わせたかったら、どうします？ 何か前にも聞いた話ですね。普通の値を取って文脈の付いた値を返す関数があるとき、それに文脈の付いた値を食わせるには、どうしたらいいのでしょう？

前の章で Maybe モナドを探検しているとき、applyMaybe という関数を作ったのを覚えていますか。この関数は Maybe a 型の値と a -> Maybe b 型の関数を引数に取りました。関数のほうは Maybe a 型でなく a 型しか引数に取れないにもかかわらず、applyMaybe はそれに Maybe a 型の値を食わせることができました。それは、Maybe a 型の値に付いてくる文脈、つまり計算が失敗しているかもしれないという文脈を考慮することで可能になったのです。applyMaybe（のちの >>= ）が、Nothing か Just かの文脈を処理してくれるので、関数 a -> Maybe b の中では、その値を普通の値として扱うことができたのです。

同じ調子で、ログの付いた値、つまり (a, String) 型の値と、a -> (b, String) 型の関数の 2 つを取り、その値を関数のほうに食わせる関数を作りましょう。名前は applyLog にしましょう。(a, String) 値は、失敗の可能性という文脈ではなく、ログを表す値が付いているという文脈を持ちます。したがって applyLog は、元の値に付いてきたログを失うことなく、関数が新たに生み出したログと結合するように注意を払ってくれます。これが applyLog の実装です。

```
applyLog :: (a, String) -> (a -> (b, String)) -> (b, String)
applyLog (x, log) f = let (y, newLog) = f x in (y, log ++ newLog)
```

これまで文脈付きの値を関数に食わせたいと思ったら、まず実際の値を文脈から分離して、それに関数を適用し、それでもって文脈がどうなったかを見る、という手順をとってきました。例えば Maybe モナドを扱うときは、まずそれが Just x であるかを調べ、もしそうだったら x を取り出して関数を適用しました。今回も、実際の値はとても簡単に見つかりますね。だって本来の値とログのペアを扱っているのですから。というわけで、まず値のほうを取り出し、x と名づけ、関数 f を適用します。すると (y, newLog) というペアが手に入ります。ここで y は新しい値で、newLog は新しいログです。でも、ここで newLog を返してしまうと、古いログは結果から消えてしまいますから、代わりに (y, log ++ newLog) を返します。ここでは 2 つのログを結合するのに ++ を使っています。

applyLog はこんなふうに動作します。

```
ghci> (3, "Smallish gang.") `applyLog` isBigGang
(False,"Smallish gang.Compared gang size to 9.")
ghci> (30, "A freaking platoon.") `applyLog` isBigGang
(True,"A freaking platoon.Compared gang size to 9.")
```

実行結果は以前と似ていますが、今度は盗賊団の人数にはじめからログが付いていて、それが結果のログにも含まれています。

applyLog をもっと使ってみましょう。

```
ghci> ("Tobin", "Got outlaw name.") `applyLog` ⇨
      (\x -> (length x, "Applied length."))
(5,"Got outlaw name.Applied length.")
ghci> ("Bathcat", "Got outlaw name.") `applyLog` ⇨
      (\x -> (length x, "Applied length."))
(7,"Got outlaw name.Applied length.")
```

ラムダ式の中で x がタプルではなくただの文字列になっており、ログの追記は applyLog が面倒を見てくれているのが分かりますか？

モノイドが助けにきたよ

現状の applyLog は (a, String) 型の値を取るようになっていますが、ログは別に String である必要はないですよね？ ログへ追記するのには ++ を使っているんだから、文字のリストではなく、任意の型のリストが使えるんじゃないでしょうか？ もちろん、そのとおりです。ですから applyLog の型はこれに変えられます。

```
applyLog :: (a, [c]) -> (a -> (b, [c])) -> (b, [c])
```

これでログはリストになりました。ただし、最初のリストと関数が返すリストの中身の型は同じでないといけません。そうでなければ ++ を使ってくっつけられないですからね。

じゃあ、 ByteString には使えるでしょうか？ いかにもできそうですよね。ところが、現状の型ではリストに対してしか使えません。 ByteString の applyLog は別に作らないといけないのでしょうか……、待って！ リストも ByteString も Monoid 型クラスです。ということは、どちらも mappend を実装しているということです。そして mappend はまさにものをくっつける操作です。見てください。

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> B.pack [99,104,105] `mappend` B.pack [104,117,97,104,117,97]
Chunk "chi" (Chunk "huahua" Empty)
```

いいね！ これで applyLog が任意のモノイドを受け付けるようにできます。これを反映するよう、型と実装を変えないといけませんね。 ++ を mappend に取り替えて、っと。

```
applyLog :: (Monoid m) => (a, m) -> (a -> (b, m)) -> (b, m)
applyLog (x, log) f = let (y, newLog) = f x in (y, log `mappend` newLog)
```

これで、applyLog が付加する値は任意のモノイド値になったので、別にタプルを「値とログの組」と解釈する必要はなくなりました。今や、「値と、モノイド値のおまけ」とみなすことができます。例えば、商品の名前と価格（モノイド値）の組というのはどうでしょう。**newtype** の Sum を使うだけで、商品を扱うとき価格が加算されることを保証できます。例えば、これはぶっきらぼうな食事の注文を取って、飲み物も出してくれる関数です。

```
import Data.Monoid

type Food = String
type Price = Sum Int

addDrink :: Food -> (Food, Price)
addDrink "beans" = ("milk", Sum 25)
addDrink "jerky" = ("whiskey", Sum 99)
addDrink _ = ("beer", Sum 30)
```

ここでは、食べ物を文字列で表し、Sum で包んだ Int で値段を追跡しています。復習ですが、Sum を mappend すると、包まれた値は足し算されるのでしたね。

```
ghci> Sum 3 `mappend` Sum 9
Sum {getSum = 12}
```

addDrink 関数はかなりシンプルです。まず、豆を食べているときは、"milk" と一緒に Sum 25 を返します。つまり 25 セントが Sum に包まれているわけです。次に、ジャーキーを食べているときは、ウィスキーを飲むことにします。それ以外のものを食べるときはビールです。この関数を、普通に「食べ物」だけに適用しても、大して面白いことは起こりません。でも、「食べ物と値札の組」に applyLog を使って適用するのは一見の価値ありです。

```
ghci> ("beans", Sum 10) `applyLog` addDrink
("milk",Sum {getSum = 35})
ghci> ("jerky", Sum 25) `applyLog` addDrink
("whiskey",Sum {getSum = 124})
ghci> ("dogmeat", Sum 5) `applyLog` addDrink
("beer",Sum {getSum = 35})
```

ミルクは 25 セントですが、10 セントの豆と一緒に頼むと 35 セント払うことになります。

これで、おまけの値の用途はログに限らないことがはっきりしましたね。モノイド値だったら何でもよく、その値のくっつけ方はモノイド次第です。実際、ログを扱っているときは文字列結合でしたが、今は足し算になってます。

addDrink が返す値は (Food, Price) のタプルなので、それにもう一度 addDrink を適用して、飲み物を追加注文し、値段の合計を知ることができます。

やってみましょう。

```
ghci> ("dogmeat", Sum 5) `applyLog` addDrink `applyLog` addDrink
("beer",Sum {getSum = 65})
```

犬の肉に飲み物を注文すると、ビールがきて 30 セント追加されるので、結果は ("beer", Sum 35) になります。それにもう一度 applyLog を使って addDrink すると、もう 1 杯ビールがきて ("beer", Sum 65) になります。

Writer 型

これで、「モノイドのおまけの付いた値」がいかにもモナド値のように振る舞うことが分かりましたね。では、そのような値の Monad インスタンスを見ていこうじゃないですか。 Control.Monad.Writer モジュールが、 Writer w a 型とその Monad インスタンス、それに Writer w a 型を扱うための便利な関数をエクスポートしています。

値にモノイドのおまけを付けるには、タプルに入れるだけです。 Writer w a 型の実体は、そんなタプルの **newtype** ラッパーにすぎず、定義はとてもシンプルです。

```
newtype Writer w a = Writer { runWriter :: (a, w) }
```

newtype に包むことで、 Monad のインスタンスにするときに既存のタプルには影響を与えないようになっています。型引数 a が主となる値の型を表し、型引数 w がおまけのモノイド値の型を表しています。

Control.Monad.Writer モジュールの開発者は、 Writer w a 型の内部実装を変える権利を保持しておきたいそうで、 Writer 値コンストラクタをエクスポートしていません。しかし Writer コンストラクタと同じことをする writer 関数を公開しています。これを使えばタプルを Writer 値に変えられます。

Writer 値コンストラクタがエクスポートされていないので、それとパターンマッチして中身を取り出すこともできません。その代わり runWriter 関数を使います。これが **newtype** ラッパーである Writer 値を取って中身のタプルを返してくれます。

Monad インスタンスは、こんなふうに定義されています。

```
instance (Monoid w) => Monad (Writer w) where
    return x = Writer (x, mempty)
    (Writer (x, v)) >>= f = let (Writer (y, v')) = f x
                            in Writer (y, v `mappend` v')
```

まずは >>= から見ていきましょう。この実装は applyLog と基本は同じ、ただしタプルが Writer による **newtype** ラッパーの中にいる点だけが違っており、パターンマッチを使って中身を取り出しています。出てきた値 x に関数 f を適用しています。これで Writer w a 型の値が得られますから、さらに **let** 式でパターンマッチします。そして y を新しい計算結果とする一方、 mappend を使って 2 つのモノイド値を結合します。結合して得られた計算結果とモノイドをタプルに入れ、続いて Writer コンストラクタで包むことで、返り値を Writer 型にしています。生のタプルを返すわけにはいきませんからね。

じゃあ return のほうはどうでしょう？ return は、値を取って、それを再現できるような最小限のデフォルト文脈に入れる必要があります。 Writer 値にとってそのような最小限の文脈、つまりおまけのモノイド値って何でしょう？他のモノイド値になるべく影響を与えないモノイド値を選ぶとしたら、 mempty を使うのがよさそうですね。

mempty はモノイドの単位元を表現するのに使われるのでしたね。例えば "" とか Sum 0 とか、空の ByteString とか。 mempty と、何か他のモノイド値を mappend した結果は、常にその「他のモノイド値」です。そこで、 return を使って Writer 値を作り、それを >>= を使って他の関数に食わせたら、結果のモノイド値は、関数が返したものがそのまま入っているはずです。

3 という数に対して、組み合わせるモノイドを変えながら return を使ってみようじゃないですか。

```
ghci> runWriter (return 3 :: Writer String Int)
(3,"")
ghci> runWriter (return 3 :: Writer (Sum Int) Int)
(3,Sum {getSum = 0})
ghci> runWriter (return 3 :: Writer (Product Int) Int)
(3,Product {getProduct = 1})
```

Writer には Show インスタンスがないので、 runWriter を使って、 Writer 値を show が使えるタプルに変換しています。 String の単位元は空文字列になっています。 Sum に対する単位元は 0 です。0 は何と足しても相手を返しますからね。 Product を使ったら、単位元は 1 になります。

Writer インスタンスは fail の実装を与えていないので、 **do** 記法の中でパターンマッチに失敗すると error が呼ばれます。

Writer を do 記法で使う

こうして Monad インスタンスができたので、 Writer を **do** 記法で自由に扱えます。 **do** 記法は複数の Writer をまとめて何かしたいときに便利です。他のモナドの場合と同じく、プログラマにとっては普通の値のように扱える裏で、モナドが文脈の面倒を見てくれます。今回の場合は、すべてのモノイド値が mappend され、最終結果に反映されます。

Writer を **do** 記法で使い、2 つの数を掛け算する例です。

```
import Control.Monad.Writer

logNumber :: Int -> Writer [String] Int
logNumber x = writer (x, ["Got number: " ++ show x])

multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    return (a*b)
```

logNumber は、数を取って Writer 値を作り出します。 Writer 値コンストラクタを使わずに、 writer 関数を使って Writer 値を作っているところがポイントです。モノイド値としては文字列のリストを使うことにし、入ってきた数に対して「その数が通ったよ」という記録を単一要素リストとして残します。multWithLog は、全体は 1 つの Writer 値で、中では 3 と 5 を掛け算しつつ、ログをもれなく残します。 return を使って、a*b を最終結果として返しています。 return は、引数を最小の文脈に入れるものでしたから、きっとログには何も追加しないだろうと、自信をもって言えます。

このコードを実行するとこうなります。

```
ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5"])
```

時には、ある時点でモノイド値だけを追記したいことがあるかもしれません。そんなとき便利なのが tell です。 tell は MonadWriter 型クラスの一部です。Writer の場合は、モノイド値、例えば ["This is going on"] を取り、ダミー値 () を返しつつ、そのモノイド値を追記するという Writer を返します。モナドが () を返すときは、返り値を変数に束縛する必要はありません。

multWithLog の特別な報告が追加されたバージョンです。

```
multWithLog :: Writer [String] Int
multWithLog = do
    a <- logNumber 3
    b <- logNumber 5
    tell ["Gonna multiply these two"]
    return (a*b)
```

return (a*b) が最後の行になっているのは重要です。 do 式の最後のモナドの結果が、 do 式全体の結果になるというルールだからです。仮に tell を最後の行に置いたら、 do 式の結果は () になり、掛け算の結果は失われていたでしょう。ただし、ログは変更を受けません。これが動作です。

```
ghci> runWriter multWithLog
(15,["Got number: 3","Got number: 5","Gonna multiply these two"])
```

プログラムにログを追加しよう！

ユークリッドの互除法は、2 つの数を取ってその最大公約数（2 つの数をどちらも割り切れる数のうち最大のもの）を求めるアルゴリズムです。Haskell には、そのものずばり gcd 関数がすでにあるのですが、せっかくなのでログを残す機能の付いたバージョンを自前で作ってみましょう。まず、これが普通のアルゴリズムです。

```
gcd' :: Int -> Int -> Int
gcd' a b
    | b == 0    = a
    | otherwise = gcd' b (a `mod` b)
```

このアルゴリズムはとてもシンプルです。まず、第二引数がゼロかどうかを判定します。もしゼロなら、第一引数を返します。そうでないなら、第二引数と「第一引数を第二引数で割った余り」との最大公約数を返します。

例えば、8 と 3 の最大公約数を、このアルゴリズムのとおりに求めてみましょう。まず、3 は 0 じゃないので、3 と 2 の最大公約数を求めることになります（8 を 3 で割った余りは 2 ですから）。2 は、これも 0 ではないので、今度は 2 と 1 です。またしても 0 じゃないので、アルゴリズムは 1 と 0 に進み、ようやくのことで第二引数が 0 になったので、1 を返します。あのコードと答は一致するでしょうか。

```
ghci> gcd' 8 3
1
```

一致しましたね。たいへんよくできました！ さて、この結果に、ログの役割を果たすモノイド値、という文脈を付けたいわけです。さっきみたいに文字列の

リストをログとして使いましょう。ということは、新しい gcd' 関数の型はこうなるはずです。

```
gcd' :: Int -> Int -> Writer [String] Int
```

あとはこの関数にログを付けるだけです。こちらが完成したコードです。

```
import Control.Monad.Writer

gcd' :: Int -> Int -> Writer [String] Int
gcd' a b
    | b == 0 = do
        tell ["Finished with " ++ show a]
        return a
    | otherwise = do
        tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
        gcd' b (a `mod` b)
```

この関数は、普通の Int 値を 2 つ取って、Writer [String] Int 、すなわちログ取りという文脈の付いた Int です。この関数は b が 0 である場合、単純に a を返す代わりに、 do 式を使って Writer 値を結果として返します。それには、まず tell を使って終了したことを伝え、次に return を使って a を do 式の結果としています。この do 式の代わりに、こう書くこともできます。

```
writer (a, ["Finished with " ++ show a])
```

まあでも、 do 式のほうが読みやすいんじゃないですかね。

次に、 b が 0 ではない場合がきます。この場合、 mod を使って a を b で割った余りを求めたことを記録しておくことにします。それから do 式の 2 行目で gcd' を再帰的に呼び出します。ここで、gcd' は最後には Writer 値を返すことを思い出すと、 gcd' b (a `mod` b) を do 式の結果行に置いておくのは完全に正しいことだと分かります。

この新しい gcd' を試してみましょう。その返り値は Writer [String] Int であり、 newtype から中身を取り出せばタプルとして読めるはずで、タプルの第一要素は計算結果のはずです。やってみましょう。

```
ghci> fst $ runWriter (gcd' 8 3)
1
```

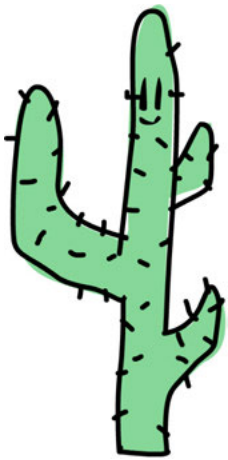
いいですね！ ログのほうはどうでしょう？ ログは文字列のリストですから、mapM_ putStrLn でも使って画面に表示させてみることにしましょう。

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcd' 8 3)
8 mod 3 = 2
3 mod 2 = 1
2 mod 1 = 0
Finished with 1
```

こんなふうに、普通のアルゴリズムを実行中に何をしてるか報告するアルゴリズムに変えられるのって、すごいと思うんですよね。しかも、普通の値をモナド値に変えるだけで、それができるんです。ログを集める作業は Writer の >>= の実装が勝手にやってくれます。

このログ機能は、ほぼどんな関数にも追加できます。ただ、普通の値を Writer 値に、関数適用を >>= に変えればよいだけです（あるいは **do** のほうが可読性が上がるかも）。

非効率なリスト構築



Writer モナドを使うときは、使うモナドに気をつけてください。リストを使うととても遅くなる場合があるからです。リストは mappend に ++ を使っていますが、++ を使ってリストの最後にものを追加する操作は、そのリストがとても長いと遅くなってしまいます。

さっき作った gcd' 関数のログ取りは速いほうでした。なぜなら、最終的に行われるリスト結合演算はこんな感じになっていたからです。

```
a ++ (b ++ (c ++ (d ++ (e ++ f))))
```

リストは左から右へ構築されるデータ構造です。これが効率的なのは、まずリストの左辺を最後まで構築し、それから初めて右辺の長いリストを結合しているからです。でも、うっかりすると、 Writer モナドを使った結果こんなコードができかねません。

```
((((a ++ b) ++ c) ++ d) ++ e) ++ f
```

さっきのが右結合だったのに対し、これは左結合です。このコードは、右辺を左辺に結合しようとするたびに、左辺をはじめから構築しないといけないので、非常に非効率です！

今からお見せする関数は gcd' と似ていますが、ログの出力が逆順になっています。再帰の各ステップは、まずプログラムの残りの部分のログを全部出力してから今のステップをログの最後に追加するようになっています。

```
import Control.Monad.Writer

gcdReverse :: Int -> Int -> Writer [String] Int
gcdReverse a b
    | b == 0 = do
        tell ["Finished with " ++ show a]
        return a
    | otherwise = do
        result <- gcdReverse b (a `mod` b)
        tell [show a ++ " mod " ++ show b ++ " = " ++ show (a `mod` b)]
        return result
```

こいつは、まず再帰を呼び出してその結果を result という変数に束縛します。それから今のステップをログに追加するので、現在のステップは再帰が生成したログの最後にきます。最後に、再帰の結果を自身の計算結果として提示しています。これを動かすと、こうなります。

```
ghci> mapM_ putStrLn $ snd $ runWriter (gcdReverse 8 3)
Finished with 1
2 mod 1 = 0
3 mod 2 = 1
8 mod 3 = 2
```

この関数は、 ++ を右結合ではなく左結合で使ってしまうので、非効率的です。

このようなやり方で結合していくとリストでは非効率になってしまう場合があるので、常に効率的な結合をサポートするデータ構造を使うのが一番でしょう。そのようなデータ構造の 1 つが差分リストです。

差分リストを使う

通常のリストに似ている**差分リスト**ですが、その実態はリストを取って別のリストを先頭に付け加える関数です。例えば、 [1,2,3] と等価な差分リストは \xs -> [1,2,3] ++ xs です。通常の空リストは [] ですが、空の差分リストは関数 \xs -> [] ++ xs として表されます。

差分リストは効率の良いリスト結合をサポートします。普通のリストを 2 つ、++ で結合するときは、左辺のリストを最後まで延々と辿っていって、そこに右辺をくっつけないといけません。でも、差分リストというアプローチをとってリストを関数として表現すると、何が起こるでしょう？

2 つの差分リストを結合する操作は、こうです。

```
f `append` g = \xs -> f (g xs)
```

f と g は、リストを取ってその前に何かを付ける関数でしたね。例えば、 f が ("dog"++) （別の書き方をすると \xs -> "dog" ++ xs ）という関数で、 g が ("meat"++) という関数なら、 f `append` g は次の関数と等価になります。

```
\xs -> "dog" ++ ("meat" ++ xs)
```

2 つの差分リストを結合した結果は、引数にまず 2 つ目の差分リスト、続いて 1 つ目の差分リストを適用する関数になるようです。

差分リストの **newtype** ラッパーを作りましょう。そうすればモノイドインスタンスを作るのが楽になります。

```
newtype DiffList a = DiffList { getDiffList :: [a] -> [a] }
```

包まれているものの型は `[a] -> [a]` です。差分リストは、リストを取って同じ型のリストを返す関数にすぎないからです。普通のリストを差分リストに変えたり、その逆をするのは簡単です。

```
toDiffList :: [a] -> DiffList a
toDiffList xs = DiffList (xs++)

fromDiffList :: DiffList a -> [a]
fromDiffList (DiffList f) = f []
```

普通のリストを差分リストにするには、さっきもやったような方法で、そのリストを引数リストの先頭に追加するような関数を作るだけです。そして差分リストはリストの前に何かを結合する関数なので、その「何か」を取り出したかったら、その関数を空リストに適用するまでです！

これが Monoid インスタンスです。

```
instance Monoid (DiffList a) where
    mempty = DiffList (\xs -> [] ++ xs)
    (DiffList f) `mappend` (DiffList g) = DiffList (\xs -> f (g xs))
```

差分リストをリストからリストへの関数と見た場合、mempty は id 関数であり mappend は関数合成になっていることが分かりますか？ これがうまく動くか試してみましょう。

```
ghci> fromDiffList (toDiffList [1,2,3,4] `mappend` toDiffList [1,2,3])
[1,2,3,4,1,2,3]
```

最高だな！ これで gcdReverse 関数の効率を上げられます。リストの代わりに差分リストを使うだけです。

```
import Control.Monad.Writer

gcd' :: Int -> Int -> Writer (DiffList String) Int
gcd' a b
    | b == 0 = do
        tell (toDiffList ["Finished with " ++ show a])
        return a
    | otherwise = do
        result <- gcd' b (a `mod` b)
        tell (toDiffList [show a ++ " mod " ++ show b ++
                          " = " ++ show (a `mod` b)])
        return result
```

必要なことは、モノイドの型を [String] から DiffList String に変えることと、tell を呼ぶ前に toDiffList で普通のリストを差分リストに変えることだけです。ログがきちんと組み立てられてるか、確かめてみましょう。

```
ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ ⇨
      gcdReverse 110 34
Finished with 2
8 mod 2 = 0
34 mod 8 = 2
110 mod 34 = 8
```

まず gcdReverse 110 34 をし、それから runWriter を使って **newtype** を剥がし、snd を使ってログだけを取り出し、fromDiffList を使って通常のリストに変換し、最後にその要素を画面に表示させています。

性能の比較

差分リストがどのくらい速度を上げてくれるのか体感するために、こんな関数を考えましょう。その関数は、自然数の引数を取って、ひたすらゼロまでカウントダウンしますが、gcdReverse のように逆向きのログを生成することで、ログの中では数がカウントアップされるようにします。

```
finalCountDown :: Int -> Writer (DiffList String) ()
finalCountDown 0 = do
    tell (toDiffList ["0"])
finalCountDown x = do
    finalCountDown (x-1)
    tell (toDiffList [show x])
```

この関数に 0 を与えると、単にログを取ります。他の数では、まずその数マイナス 1 のカウントダウンを呼び出してから、その数をログに加えます。ですから、finalCountDown を 100 に適用すると、文字列 "100" はログの最後にくるわけです。

この関数を GHCi に読み込ませて、巨大な数、そうですね 500000 とかに適用すると、素早く 0 から数えだすのが見られます。

```
ghci> mapM_ putStrLn . fromDiffList . snd . runWriter $ ⇨
      finalCountDown 500000
0
1
2
...
```

しかし、差分リストの代わりに普通のリストを使ったらどうでしょう？

```
finalCountDown :: Int -> Writer [String] ()
finalCountDown 0 = do
    tell ["0"]
finalCountDown x = do
    finalCountDown (x-1)
    tell [show x]
```

で、GHCi に数え始めなさいと言うと、

```
ghci> mapM_ putStrLn . snd . runWriter $ finalCountDown 500000
```

めっちゃ遅いです……。

もちろん、これはプログラムの速度を測る正しい科学的な方法ではありません。けれども、今回の場合、差分リスト版はすぐに結果を出し始めたのに対し、リスト版は動きだす気配すらない、という違いは体感できたのではないでしょうか。

あ、ところで北欧メタルバンド、EUROPE の “The Final Countdown” が脳内ループしてます？

14.2 Reader？ それはあなたです！

第 11 章では、関数を作る型、(->) r も、Functor のインスタンスであることを見ました。関数 f で関数 g を写すと、「g が取るのと同じ型の引数を取り、それに g を適用したものに f を適用して返す」関数ができるのでした。基本的にやってることは、「g のような関数」を作っているわけですが、ただし結果を返す前に f が適用されているわけです。これが例です。

```
ghci> let f = (*5)
ghci> let g = (+3)
ghci> (fmap f g) 8
55
```

それから、関数はアプリカティブファンクターであることも見ましたね。これにより、関数が将来返すであろう値を、すでに持っているかのように演算できるようになりました。これが例です。

```
ghci> let f = (+) <$> (*2) <*> (+10)
ghci> f 3
19
```

(+) <$> (*2) <*> (+10) という式は、「ある数を引数に取って、それに (*2) と (+10) を適用し、その結果どうしを足し算する関数」になります。例えば、この関数を 3 に適用すると (*2) と (+10) の両方が 3 に適用され、 6 と 13 ができます。それから 6 と 13 を引数に (+) が呼ばれ、結果は 19 になります。

モナドとしての関数

関数の型 (->) r はファンクターであり、アプリカティブファンクターであるばかりでなく、モナドでもあります。これまでに登場したモナド値と同様、関数もまた文脈を持った値だとみなすことができるのです。関数にとっての文脈とは、値がまだ手元になく、値が欲しければその関数を別の何かに適用しないといけない、というものです。

これまでに、ファンクターやアプリカティブファンクターとして働く関数には精通してきましたから、さっそく Monad インスタンスの設計を見にいきましょう。インスタンス宣言は Control.Monad.Instances にあり、こんな感じのソースになってます。

```
instance Monad ((->) r) where
    return x = \_ -> x
    h >>= f = \w -> f (h w) w
```

関数にとっての pure の実装は前に見ましたね。 return もほとんど pure と同じです。値を取って、その値を結果として返す最小限の文脈を常に返す。関数の場合、常に同じ値を返すようにする唯一の方法は、引数をガン無視させることです。

>>= の実装は暗号めいて見えるかもしれませんが、実際には見かけほど複雑ではありません。 >>= を使ってモナド値を関数に食わせるときは、結果は常にモナド値になります。ですからこの場合、ある関数を別の関数に食わせた結果は、また関数になるはずです。それが、結果の式がラムダ（ \ ）から始まっている理由です。

これまで見てきた >>= の実装はすべて、何らかの形で値をモナドから取り出して、それに関数 f を適用するものでした。ここでも同じことが起こっています。関数から結果を取り出すには、それを何かに適用しないといけません。なの

で、(h w) を使って、それに f を適用しているわけです。 f はモナド値（ここでは関数）を返すので、 w にもそいつを適用します。

Reader モナド

この時点で >>= がどう働くのか飲み込めなかったとしても心配なく。いくつか例を見れば、これがすごくシンプルなモナドであることが分かるでしょう。これが関数モナドを使っている do 式です。

```
import Control.Monad.Instances

addStuff :: Int -> Int
addStuff = do
    a <- (*2)
    b <- (+10)
    return (a+b)
```

これは前に見たアプリカティブ式と同じものですが、関数がモナドであることを使っているバージョンです。 do 式は常にモナド値を生み出すもので、今回も例外ではありません。この do 式の作るモナドは関数です。 addStuff は整数を 1 つ引数に取ります。まず、それに (*2) が適用され、結果は a になります。同じ整数に (+10) が適用され、その結果が b です。 return は、他のモナドと同様、文脈には何も影響せず、引数に与えられた値をそのまま提示するモナド値を作ります。これが a+b を関数全体の結果として提示します。試してみれば前と同じ結果が返ってくるはずです。

```
ghci> addStuff 3
19
```

(*2) と (+10) はどちらも 3 に適用されます。実は、return (a+b) も同じく 3 に適用されるんですが、引数を無視して常に a+b を返しています。そういうわけで、関数モナドは **Reader モナド**とも呼ばれたりします。すべての関数が共通の情報を「読む」からです。このことをもっと明確にするために、 addStuff をこんなふうに書き直すこともできます。

```
addStuff :: Int -> Int
addStuff x = let
    a = (*2) x
    b = (+10) x
    in a+b
```

このように、Reader モナドは関数を文脈付きの値として扱うことを可能にします。関数が返すであろう値をすでに知っているつもりができるのです。それができるのは、複数ある関数を貼り付けて 1 つの関数を作り、それに渡った引数を構成要素の関数すべてに配っているからです。というわけで、もし最後の引数

が届くのを待っている関数がたくさんあって、しかもそれらに渡したい値はみな同じ、という状況があれば、Reader モナドを使って未来の結果を取り出すことができます。うまく動くことは >>= が保証してくれます。

14.3　計算の状態の正体

Haskell は純粋な言語ですから、Haskell のプログラムはグローバルな状態や変数を書き換えたりできない関数だけで構成されています。関数は常に同じ計算をして値を返す運命なのです。この制限は、実のところプログラムについて考えるのを楽にしてくれます。だって、特定の時刻でのすべての変数の値を考慮に入れる必要がなくなるわけですから。

ところが、「状態」が本質的な問題、時間とともに変わっていく何らかの状態に依存している計算というのは確かにあります。Haskell はそういった計算でも問題なく扱えるんですが、モデル化するのは少し骨が折れます。そこで Haskell には State モナドが用意されています。これさえあれば、状態付きの計算などいとも簡単。しかもすべてを純粋に保ったまま扱えるんです。

第 9 章で乱数を見たときには、乱数ジェネレータを引数に取り、乱数と新しいジェネレータを返す関数を扱いました。複数の乱数が必要であれば、乱数と一緒に出てきた新しいジェネレータを常に使うよう注意する必要がありました。例えば、ジェネレータ StdGen を引数に取り、それを使ってコインを 3 回投げる関数を作るには、こうします。

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

この関数にジェネレータ gen を渡すと、まず random gen が Bool 値と新たなジェネレータを返します。2 枚目のコインを投げるには、この新しいジェネレータを使います。以下同様。

Haskell 以外のほとんどの言語では、乱数に添えて新しいジェネレータを返す必要などありません。だってジェネレータの状態を上書きすればいいじゃん！でも Haskell は純粋な言語ですから、それはできません。そこで、状態を受け取って、結果を作るとともに新しい状態も作り、その新しい状態を使って次の結果を作る必要があります。

そんなふうに状態を手動で扱うのを避けるには Haskell の純粋性をあきらめる必要がある、と考えるかもしれません。でも、その必要はないんです。なぜなら、`State` モナドという、Haskell プログラミングをこんなに素敵にしている純粋性を少しも損なうことなく状態を扱うための面倒な作業を裏でやってくれる特別にかわいいモナドがあるからです。

状態付きの計算

状態付きの計算を実演するために、まずは型を与えてみましょう。状態付きの計算とは、ある状態を取って、更新された状態と一緒に計算結果を返す関数として表現できるでしょう。そんな関数の型は、こうなるはずです。

```
s -> (a, s)
```

`s` は状態の型で、 `a` は状態付き計算の結果です。

NOTE Haskell 以外のほとんどの言語における「代入」という操作は、状態付きの計算と捉えることができます。例えば、手続き型言語で `x = 5` と書くと、普通は変数 `x` に `5` が入り、ついでに式の値も `5` になります。この挙動をじっくり観察すると、（これまでに代入されたすべての変数という）状態を取って、新しい状態と、（`5`）という結果を返す関数に見えてきませんか？ この場合の新しい状態というのは、代入前の変数のマッピングから新たに代入された部分だけが変わっている状態です。

このような状態付きの計算（状態を取って、計算結果と新しい状態を返す関数）もまた、文脈付きの値だとみなすことができます。計算の結果が「生の値」であり、その計算結果を得るためには初期状態を与える必要があること、そして、計算の結果を得るのと同時に新しい状態が得られるというのが「文脈」にあたります。

スタックと石

スタックをモデル化したいとしましょう。 **stack** とは、いくつかのデータを格納でき、次の 2 つの操作をサポートするデータ構造です。

- **Push**：スタックのてっぺんに要素を積む。
- **Pop**：スタックのてっぺんの要素を取り除く。

スタックを表現するのにはリストを使い、リストの先頭がスタックのてっぺんに対応することにします。次のような 2 つのヘルパー関数を作りましょう。

- pop ：スタックを引数に取って、要素を 1 つ取り出し、その取り出された要素を返す関数。ついでにその要素を除いた後の新しいスタックも返す。
- push ：ある要素とスタックを引数に取り、その要素をスタックに積む関数。 () を結果として返し、ついでに新しいスタックも返す。

これがその関数です。

```
type Stack = [Int]

pop :: Stack -> (Int, Stack)
pop (x:xs) = (x, xs)

push :: Int -> Stack -> ((), Stack)
push a xs = ((), a:xs)
```

push の返り値は () にしました。 push 操作の仕事はスタックを変更することで、特に重要な結果値というものはないからです。 push の第一引数だけを部分適用すると、状態付き計算が生まれます。 pop は、その型からして、すでに状態付き計算になっていますね。

これらの関数を使って、スタックをシミュレートするちょっとしたコードを書いてみましょう。とりあえず、 3 でも積んで、それから 2 つばかし値を取り出してみましょうか。こうです。

```
stackManip :: Stack -> (Int, Stack)
stackManip stack = let
    ((), newStack1) = push 3 stack
    (a , newStack2) = pop newStack1
    in pop newStack2
```

この関数は、 stack を取って、まず push 3 stack をします。その結果はタプルで、第一要素が () 、第二要素は新しいスタックです。こいつの名前は newStack1 にしましょう。次に newStack1 から新しい値を取り出します。その結果は a という数（さっき積んだ 3 が入っているはず）と、また新しいスタックです。今度は newStack2 という名前にしましょう。さらに、 newStack2 から数を取り出し、数 b とスタック newStack3 のタプルを手に入れます。 stackManip は、このタプルをそのまま返しています。使ってみましょう。

```
ghci> stackManip [5,8,2,1]
(5,[8,2,1])
```

結果は 5 で、新しいスタックは [8,2,1] になりました。 stackManip 自身も状態付き計算になっていたことに気づきましたか？ 今やったのは、状態付き計

算をいくつか取って糊付けするという操作だったわけです。それってどこかで聞いたことがあるような……。

さっきの stackManip のコードは、ちょっと長ったらしいですよね。すべての状態付き計算に、手で状態を与え、いちいち回収して名前をつけて、また次のやつに与えています。各関数にスタックを手動で与えるのではなくて、こんなふうに書けたらすごいと思いませんか？

```
stackManip = do
    push 3
    a <- pop
    pop
```

なんと、State モナドを使うとまさにこんなふうに書けちゃうんです！ State モナドがあれば、このような状態付き計算を、状態に手を触れる必要もなく扱えるのです。

State モナド

Control.Monad.State モジュールは、状態付き計算を包んだ **newtype** を提供しています。これがその定義です。

```
newtype State s a = State { runState :: s -> (a, s) }
```

State s a は、 s 型の状態を操り、 a 型の結果を返す状態付き計算です。

Control.Monad.Writer と同じく、 Control.Monad.State も値コンストラクタをエクスポートしていません。状態付き計算を State の **newtype** に包みたいときは、 state 関数を使いましょう。 state 関数は State コンストラクタとまったく同じ動作をします。

これで、状態付き計算とは何であって、それがどうして文脈付きの値とみなせるのかが分かりました。じゃあ状態付き計算の Monad インスタンスを見ていきましょう。

```
instance Monad (State s) where
    return x = State $ \s -> (x, s)
    (State h) >>= f = State $ \s -> let (a, newState) = h s
                                        (State g) = f a
                                    in  g newState
```

return は、値を取って常にその値を結果として返すような状態付き計算にしたいわけです。それが、ラムダ式 \s -> (x, s) が出てくる理由です。こいつは常に x を状態付き計算の結果として提示し、状態には一切手をつけません。 return は値を最小限の文脈に入れるという約束だからです。というわけ

で、 return はある値を提示し、状態を不変に保つような状態付き計算になるわけです。

では、 >>= はどうでしょう？ まず、状態付き計算を、 >>= を使って関数に食わせた結果もまた状態付き計算にならないといけない、ですよね？ そこでまず State の **newtype** ラッパーを書きます。それからラムダです。このラムダ式が新しい状態付き計算になるのです。では、ラムダ式の中には何を書けばいいのでしょう？ とにかく 1 つ目の状態付き計算から結果の値を取り出さないといけません。僕たちは、まさに状態付き計算の中にいますから、現在の状態 s を状態付き計算 h に渡すことならできます。すると計算結果と新しい状態のペア (a, newState) が出てきます。

これまでのところ、 >>= 演算子を実装するときは、必ずまず左辺のモナド値を使い、結果だけを取り出した後、それに右辺の関数 f を適用して、新しいモナド値を得るという手順を踏みました。 Writer を作ったときは、その手順に従って新しいモナド値を得た後、さらに 2 つのモノイド値を mappend する作業が必要でした。今回は、まず f a して、新しい状態付き計算 g を取り出しています。こうして新しい「状態付き計算」と新しい「状態」(newState という名前) が揃ったら、あとは状態付き計算 g を newState に適用するだけです。その結果は、最終結果と最終状態のタプルです！

このように、 >>= を使えば 2 つの状態付き計算を糊付けすることことができます。2 つ目の計算は、1 つ目の計算の結果を受け取る関数の中に隠れています。さて、 pop と push はすでに状態付き計算ですから、 State ラッパーに包むのは簡単です。

```
import Control.Monad.State

pop :: State Stack Int
pop = state $ \(x:xs) -> (x, xs)

push :: Int -> State Stack ()
push a = state $ \xs -> ((), a:xs)
```

関数を State の **newtype** ラッパーで包むのに、 State 値コンストラクタを直接使う代わりに state 関数を使っているのがポイントですよ。

pop はそのまま状態付き計算ですし、 push は Int を取って状態付き計算を返す関数です。これで、さっきの 3 を push してから 2 回 pop する例を、こう書け

ます。

```
import Control.Monad.State

stackManip :: State Stack Int
stackManip = do
    push 3
    a <- pop
    pop
```

みごと、1 つの push と 2 つの pop を糊付けして、1 つの状態付き計算が作れました。こいつの **newtype** ラッパーを剥がせば、初期状態を与えると動きだす関数が出てきます。

```
ghci> runState stackManip [5,8,2,1]
(5,[8,2,1])
```

ところで、2 行目の pop の結果 a は一度も使ってませんから、a に束縛する必要はなかったですね。だから、こう書いても問題ありません。

```
stackManip :: State Stack Int
stackManip = do
    push 3
    pop
    pop
```

素敵。でも、もうちょっとややこしいことはできるでしょうか？ 例えば、スタックから 1 つの数を取り出し、それが 5 だったらそっと元に戻す、でも 5 じゃなかったら、代わりに 3 と 8 を積む、とかどうでしょう。これがそのコードです。

```
stackStuff :: State Stack ()
stackStuff = do
    a <- pop
    if a == 5
        then push 5
        else do
            push 3
            push 8
```

これは直感的に書けましたね。では初期スタックを与えて走らせてみましょう。

```
ghci> runState stackStuff [9,0,2,1,0]
((),[8,3,0,2,1,0])
```

do 式はモナド値を作ること、そして State モナドに関しては **do** 式もまた状態付きの関数であることを思い出してください。stackManip と stackStuff はどちらも普通の状態付き計算ですから、この 2 つをさらに糊付けして、もっと大きな状態付き計算が作れます。

```
moreStack :: State Stack ()
moreStack = do
    a <- stackManip
    if a == 100
        then stackStuff
        else return ()
```

現在のスタックに stackManip を使った結果が 100 なら、 stackStuff を実行します。それ以外の場合は、何もしません。 return () は、状態に変更を加えず何もしないモナドです。

状態の取得と設定

Control.Monad.State モジュールは、2 つの便利な関数 get と put を備えた、MonadState という型クラスを提供しています。 State モナドに対する get の実装はこうです。

```
get = state $ \s -> (s, s)
```

現在の状態を取ってきて、それを結果として提示しているだけですね。

put 関数は、状態型の引数を取り、「その状態を、現在の状態に上書きする状態付き関数」を返します。

```
put newState = state $ \s -> ((), newState)
```

というわけで、この put と get があれば、現在のスタックを見たり、現在のスタックを丸ごとすり替えたりできます。こんなふうに。

```
stackyStack :: State Stack ()
stackyStack = do
    stackNow <- get
    if stackNow == [1,2,3]
        then put [8,3,1]
        else put [9,2,1]
```

また、 get と put を使って pop や push を実装することもできます。まず、pop はこう。

```
pop :: State Stack Int
pop = do
    (x:xs) <- get
    put xs
    return x
```

get を使ってスタック全体を取り出し、それから先頭要素を取り除いた残りを put を使って新しい状態にしています。それから return を使って x を結果として提示しています。

で、こちらは get と put を使った push の実装です。

```
push :: Int -> State Stack ()
push x = do
    xs <- get
    put (x:xs)
```

get を使って現在のスタックの状態を取得し、それに x を積んだものを put を使って新しい状態として設定するだけです。

ここで、もし >>= の型が State 値専用だったらどうなるか考えてみてください。

```
(>>=) :: State s a -> (a -> State s b) -> State s b
```

状態の型 s は常に同じで、計算結果の型は a から b に変えられます。結果の型が違う状態付き計算どうしは >>= で糊付けできますが、状態の型は同じでなければなりません。なぜだか分かりますか？ えっと、例えば Maybe モナドの >>= はこんな型でした。

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

モナド自身の型、つまり Maybe が >>= の前後で変わらないのは当然ですね。型の違う 2 つのモナドを >>= するのは無意味です。さて、 State モナドに戻ると、モナドのインスタンスになっている型は State s でした。だから s が違うってことは、 >>= を違う型のモナドの間で使おうとしてることになるんです。

乱数と State モナド

この節の冒頭で、乱数を生成する処理ってきれいに書けないよねー、という話をしました。乱数関数はジェネレータを引数に取り、乱数と一緒に新しいジェネレータを返すようにできていて、次の乱数を作るときは必ずこの新しいジェネレータを使い、古いほうを使わないよう気をつける必要があるのでした。 State モナドがあれば、こういう処理はぐっと楽になります。

System.Random モジュールの random 関数の型は、こうです。

```
random :: (RandomGen g, Random a) => g -> (a, g)
```

random は、乱数ジェネレータを引数に取り、乱数と新しいジェネレータを返す関数である、と言っていますね。どう見ても状態付き計算です。 state 関数を使って State の **newtype** に包めば、状態の扱いをモナドに任せられます。

```
import System.Random
import Control.Monad.State

randomSt :: (RandomGen g, Random a) => State g a
randomSt = state random
```

これでコインを 3 枚投げる操作はこう書けるようになりました（ True が裏で False が表ですよ）。

```
import System.Random
import Control.Monad.State

threeCoins :: State StdGen (Bool, Bool, Bool)
threeCoins = do
    a <- randomSt
    b <- randomSt
    c <- randomSt
    return (a, b, c)
```

threeCoins 関数は状態付き計算になりました。 threeCoins は、まず受け取ったジェネレータを最初の randomSt に渡します。すると randomSt が乱数と新しいジェネレータを返します。この新しいジェネレータが次に渡って、……と続きます。最後に return (a, b, c) を使って、最も新しいジェネレータを変えることなく、 (a, b, c) を結果として提示します。では使ってみましょう。

```
ghci> runState threeCoins (mkStdGen 33)
((True,False,True),680029187 2103410263)
```

これで、状態が必要な計算をするときの手間がぐっと減りましたね！

14.4　Error を壁に

Maybe モナドは「失敗するかもしれない計算」という文脈を値に与えるものでしたよね。 Maybe 値は Just something か Nothing のどちらかになれます。これは確かに便利なのですが、 Nothing を受け取ったときに分かるのは、どこかで何かが失敗したというだけです。どんな失敗があったのか、という情報を詰め込む余地はありません。

Either e a 型も失敗の文脈を与えるモナドです。しかも、失敗に値を付加できるので、何が失敗したかを説明したり、そのほか失敗にまつわる有用な情報を提供できます。 Either e a は、 Right 値であれば正解や計算の成功を、 Left 値であれば失敗を表します。これが例です。

```
ghci> :t Right 4
Right 4 :: (Num t) => Either a t
ghci> :t Left "out of cheese error"
Left "out of cheese error" :: Either [Char] b
```

Either e はおおむね Maybe の強化版ですから、モナドになっているのはごく自然なことです。 Maybe と同じく、失敗する可能性という文脈が付加された値とみなせますが、今度はエラーがあった場合にも値を付けられるんです。

Either の Monad インスタンスは Maybe のものによく似ており、Control.Monad.Error モジュールで宣言されています。

```
instance (Error e) => Monad (Either e) where
    return x = Right x
    Right x >>= f = f x
    Left err >>= f = Left err
    fail msg = Left (strMsg msg)
```

return は、いつもどおり、引数をデフォルトの最小限の文脈に入れる関数です。return は引数を Right コンストラクタに入れます。Right は、計算に成功して値があることを表すからです。これは Maybe モナドの return とよく似ていますね。

>>= は 2 つの場合に分かれます。左辺が Left である場合と Right である場合です。左辺が Right だった場合はその中の値に f を適用します。これは Maybe モナドの Just の処理とジャストそっくりです。一方、左辺がすでにエラーだった場合は、Left 値であることと、失敗を表す中身とがそのまま保たれます。

Either e の Monad インスタンスには、もう 1 つ必要条件があります。Left に入るほうの値の型（型変数 e にあたる型）は、Error 型クラスのインスタンスでなければなりません。Error 型クラスは、エラーメッセージのように振る舞える型のクラスです。Error 型クラスにはエラーを文字列として受け取って、その型に変換する strMsg 関数が定義されています。Error 型クラスの自明なインスタンスは、むろん String です！ String の場合、strMsg 関数は受け取った文字列をそのまま返すだけです。

```
ghci> :t strMsg
strMsg :: (Error a) => String -> a
ghci> strMsg "boom!" :: String
"boom!"
```

もっとも、Either を使うにあたってエラーは普通 String で表しますから、Error 型クラスについてそれほど心配はいりません。do 記法でのパターンマッチが失敗したときは、その失敗を表すのに Left 値を使ってくれます。

Either を使ってみた例です。

```
ghci> Left "boom" >>= \x -> return (x+1)
Left "boom"
ghci> Left "boom " >>= \x -> Left "no way!"
Left "boom "
ghci> Right 100 >>= \x -> Left "no way!"
Left "no way!"
```

>>= を使って Left 値を関数に食わせると、関数は無視されて Left 値がそのまま返ります。 Right 値を関数に食わせた場合は、関数が Right の中身に適用されますが、この最後の例の場合は中身が何であれ関数が Left 値を返しています。

じゃあ、 Right 値を成功する関数に渡したら？ おっと、変な型エラーが出ましたよ[†1]。うむむ。

```
ghci> Right 3 >>= \x -> return (x + 100)

<interactive>:1:0:
    Ambiguous type variable `a' in the constraints:
      `Error a' arising from a use of `it' at <interactive>:1:0-33
      `Show a' arising from a use of `print' at <interactive>:1:0-33
    Probable fix: add a type signature that fixes these type variable(s)
```

Haskell は、 Either e a という型の e の部分にどんな型を入れたらよいか分からないと言ってます。表示しようとしているのは Right 部分だけなのにね。このエラーが出るのは Monad インスタンス宣言に Error e という型クラス制約が付いていたからです。ですから、 Either モナドを使っていてさっきみたいなエラーが出たら、型シグネチャを明記しましょう。

```
ghci> Right 3 >>= \x -> return (x + 100) :: Either String Int
Right 103
```

今度は動きました！

ここでつまずいた以外は、 Either モナドの使い勝手は Maybe モナドとよく似ています。

NOTE 以前、 Maybe のモナド的側面を使って、ピエールのバランス棒に鳥がとまるようすをシミュレートしました。練習問題として、ピエールが滑って落ちた瞬間に棒の左右に何羽の鳥がとまっていたのか分かるよう、 Either モナドを使って書き直してみてください。

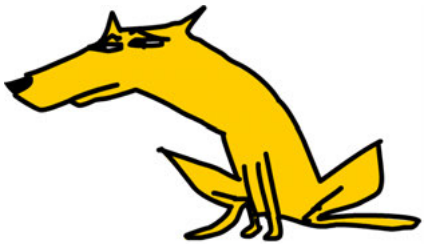
14.5　便利なモナディック関数特集

この節では、モナド値を操作したり、モナド値を返したりする関数（両方でも可！）をいくつか紹介します。そんな関数は**モナディック関数**と呼ばれます。モナディック関数にはまったく新しいものもありますが、お馴染みの関数のモ

†1　［訳注］翻訳時点での Haskell Platform 2011.4.0.0 に同梱されている GHC 7.0.4 では、 Either のモナドインスタンスは「 Control.Monad.Instances モジュールで、 Error 型クラスの文脈を付けない形で宣言する」よう変更されています。そのため、上記のサンプルはエラーを出さずに成功するようになっています。一方、 fail は Left 値でなく実行時エラーを生み出します。興味ある読者は現在の実装を確かめてみてください。

ナド版も多いですよ。 filter とか foldl とか。ここでは、 liftM 、 join 、 filterM 、そして foldM を紹介します。

liftM と愉快な仲間たち



このはてしなく遠いモナド坂を登り始めたとき、最初にファンクターを見ました。ファンクターは、「関数で写せるもの」を表していました。次に、ファンクターの強化版であるアプリカティブファンクターを見ました。これで、普通の関数を複数のアプリカティブ値に適用したり、普通の値をデフォルト文脈に入れたりできるようになりました。そしてついに、アプリカティブファンクターの強化版としてモナドを導入しました。モナドでは、文脈の中の値を何らかの手段で普通の関数に食わせることが可能になったのでした。

というわけで、すべてのモナドはアプリカティブファンクターであり、すべてのアプリカティブファンクターはファンクターでもあります。 Applicative 型クラス定義には型クラス制約が付いていて、ある型をまず Functor のインスタンスにしなければ Applicative のインスタンスにもできないようになってます。 Monad にも Applicative に対して同じような型クラス制約が付いているべきだったのですが、なにしろ Monad 型クラスが Haskell に導入されたのは Applicative よりずっと昔だったもので、そのような型クラス制約は付いてないのです。

すべてのモナドはファンクターであるべきとはいえ、モナドの Functor インスタンスに頼らなくても、 liftM 関数があれば大丈夫です。 liftM は関数とモナド値を取って、関数でモナド値を写してくれます。そう、 fmap そのものです！ liftM の型はこうです。

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
```

そしてこれが fmap の型です。

```
fmap :: (Functor f) => (a -> b) -> f a -> f b
```

もし Functor インスタンスと Monad インスタンスがそれぞれファンクター則とモナド則を満たしているなら、 fmap と liftM はまったく同じものになります（そしてこれまで出会ったモナドはちゃんと両方の法則を満たしてます）。これは pure と return が常に同じことをする、というのと似ています。ただし、

一方は Applicative 型クラス制約、他方は Monad 型クラス制約が付きます。では、 liftM を試してみましょう。

```
ghci> liftM (*3) (Just 8)
Just 24
ghci> fmap (*3) (Just 8)
Just 24
ghci> runWriter $ liftM not $ Writer (True, "chickpeas")
(False,"chickpeas")
ghci> runWriter $ fmap not $ Writer (True, "chickpeas")
(False,"chickpeas")
ghci> runState (liftM (+100) pop) [1,2,3,4]
(101,[2,3,4])
ghci> runState (fmap (+100) pop) [1,2,3,4]
(101,[2,3,4])
```

Maybe 値を fmap するとどうなるかは、よくご存知ですよね。 liftM も同じことをします。 Writer 値に関しては、関数はタプルの第一要素を写します。次に、状態付き計算に fmap や liftM を使った結果はまた状態付き計算になるんですが、その結果にくだんの関数が適用されることで値が変わります。さっきの例で、pop を走らせる前に (+100) で写しておかなかったら、結果は (1, [2,3,4]) になっていたことでしょう。

これが liftM の実装です。

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = m >>= (\x -> return (f x))
```

do 記法を使って書いてもかまいません。

```
liftM :: (Monad m) => (a -> b) -> m a -> m b
liftM f m = do
    x <- m
    return (f x)
```

モナド値 m を関数に食わせるのですが、そのモナドの結果に関数 f を適用してからデフォルトの文脈に入れています。モナド則によって、この操作は文脈をまったく変えず、モナド値が提示する結果だけを変えることが保証されています。

liftM は Functor 型クラスをまったく参照せずに実装されています。このことは、 fmap （あるいは liftM 。どちらの呼び方でも結構）はモナドが提供する機能だけを使って実装できることを意味します。このことから、モナドは少なくともファンクター以上に強い、と結論付けられます。

さて、次はアプリカティブです。 Applicative 型クラスの能力は、普通の関数を、文脈付きの値にも、あたかも普通の値であるかのように適用させてくれることでした。こんなふうに。

```
ghci> (+) <$> Just 3 <*> Just 5
Just 8
ghci> (+) <$> Just 3 <*> Nothing
Nothing
```

このアプリカティブ・スタイルは、人生をかなり楽にしてくれます。まず <$> はただの fmap 。次に <*> は Applicative 型クラスの関数で、こんな型が付いています。

```
(<*>) :: (Applicative f) => f (a -> b) -> f a -> f b
```

そう、 fmap に似ていますね。ただし、関数自身も文脈の中に入っています。どうにかして関数を文脈から取り出して f a を写し、さらに文脈を再編成する必要がありますね。Haskell の関数はデフォルトですべてカリー化されているので、 <$> と <*> を組み合わせれば、いくつ引数を取る関数だろうとアプリカティブ値に対応させられます。

とにかく、 fmap と同じように <*> も Monad 型クラスが提供する機能だけを使って実装できることが分かります。 ap という関数がありまして、本質的には <*> なのですが、 Applicative の代わりに Monad 型クラス制約が付いているのです。これが ap の定義です。

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
ap mf m = do
    f <- mf
    x <- m
    return (f x)
```

mf は、結果が関数であるようなモナド値です。関数も、それに渡したい引数も文脈の中にいるので、まず関数を文脈から取り出して f とし、次に値を取り出して x とし、最後に関数を値に適用して、その結果を提示します。さっそく実例に行きましょう。

```
ghci> Just (+3) <*> Just 4
Just 7
ghci> Just (+3) `ap` Just 4
Just 7
ghci> [(+1),(+2),(+3)] <*> [10,11]
[11,12,12,13,13,14]
ghci> [(+1),(+2),(+3)] `ap` [10,11]
[11,12,12,13,13,14]
```

これで、モナドは少なくともアプリカティブ以上に強い、ということも分かりました。なぜなら、 Monad の関数だけを使って Applicative の関数を作れるからです。実は、型がモナドであると分かったとき、 Monad のインスタンスを書き上げてしまってから、単に「 pure は return で <*> は ap だ」と書いて Applicative のインスタンスにすることがよくあります。同じように、何かが

Monad インスタンスになっていたら、そいつを Functor インスタンスにするには、ただ「fmap は liftM だ」と言うだけです。

liftA2 は、関数を 2 つのアプリカティブ値に適用するときに便利な関数です。

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f x y = f <$> x <*> y
```

liftM2 関数は型クラス制約が Monad になっているだけで同じものです。ほかにも liftM3 、 liftM4 、 liftM5 などの関数があります。

これまで、モナドの威力は少なくともアプリカティブやファンクター以上であること、そして、すべてのモナドはファンクターでもありアプリカティブファンクターであるにもかかわらず、Functor や Applicative のインスタンスになっているとは限らないということを見てきました。また、ファンクターやアプリカティブファンクターが使う関数と等価なモナド版の関数を見ました。

join 関数

ちょっと考えてみてください。あるモナド値の結果がまたモナド値だとしたら（モナドが入れ子になっていたら）、それを平らにして単一のモナド値にできるでしょうか？ 例えば、Just (Just 9) という値があったら、それを Just 9 に変えられるでしょうか？ 実は、任意の入れ子になったモナドは平らにできるんです。そして実は、これはモナド特有の性質なのです。このために、join という関数が用意されています。join の型はこうです。

```
join :: (Monad m) => m (m a) -> m a
```

ほう、join はモナド値が入ったモナド値を取って、ただのモナド値をくれる関数のようですね。いわば、モナドを平らにしてくれるわけです。Maybe 値に対して使ってみた結果はこうです。

```
ghci> join (Just (Just 9))
Just 9
ghci> join (Just Nothing)
Nothing
ghci> join Nothing
Nothing
```

1 行目は、成功する計算の結果として成功する計算を保持しています。この 2 つを join すると、単に大きな 1 つの成功する計算になります。2 行目は Just 値の結果として Nothing を返しています。これまでも、複数の Maybe 値を結合したくなったときは、その手段が <*> であれ >>= であれ、すべての入力値が Just でなければ結果は Just になりませんでした。過程のどこかが失敗したら、

計算結果は失敗になります。ここでも同じです。3 行目では最初っから失敗しているものを平らにしようとしていますが、これも同じく失敗になります。

リストを平らにする操作は、かなり直感的です。

```
ghci> join [[1,2,3],[4,5,6]]
[1,2,3,4,5,6]
```

見てのとおり、リストの join はただの concat です。 Writer 値の結果を返す Writer 値を平らにするには、モノイド値を mappend する必要があります。

```
ghci> runWriter $ join (Writer (Writer (1, "aaa"), "bbb"))
(1,"bbbaaa")
```

外側のモノイド値 "bbb" が最初にきて、それから "aaa" が追記されます。感覚的に言っても、 Writer 値の結果を調べるには、まずモノイド値をログに書き出す必要があって、そうして初めて中身を見ることができるはずですね。

Either を平らにするのは、 Maybe を平らにするのとよく似ています。

```
ghci> join (Right (Right 9)) :: Either String Int
Right 9
ghci> join (Right (Left "error")) :: Either String Int
Left "error"
ghci> join (Left "error") :: Either String Int
Left "error"
```

状態付き計算を返す状態付き計算に join を使うと、外側の状態付き計算を走らせてから出てきた計算を走らせるような状態付き計算になります。実際に動いているところを見てください。

```
ghci> runState (join (state $ \s -> (push 10, 1:2:s))) [0,0,0]
((),[10,1,2,0,0,0])
```

ここでラムダ式は、状態を取って 2 と 1 をスタックに積み、状態付き計算 push 10 を結果として返す状態付き計算です。これ全体を join で平らにして走らすと、まず 2 と 1 がスタックに積まれ、それから push 10 が実行されて、 10 がてっぺんに乗ります。

join の実装はこうなっています。

```
join :: (Monad m) => m (m a) -> m a
join mm = do
    m <- mm
    m
```

mm の結果はモナド値ですから、 mm の結果を取り出すやいなや同じ流れに乗せています。なにしろモナド値ですから。この仕掛けは、 m <- mm を呼び出したときにモナドの文脈の処理がなされることにあります。これが、 Maybe 値を

join したときに、内側と外側が両方 Just でなければ結果が Just にならなかった理由です。例えば mm が Just (Just 8) なら、join はこうなるはずです。

```
joinedMaybes :: Maybe Int
joinedMaybes = do
    m <- Just (Just 8)
    m
```

たぶん join に関して最も面白いのは、どんなモナドでも、あるモナド値とある関数があったとき、「モナド値を >>= で関数に食わせたもの」と「その関数でモナド値を写し、出てきた入れ子のモナドを join で平らにしたもの」とは常に結果が一致する！ という事実です。式でいうと、m >>= f は常に join (fmap f m) と同じということです。ちょっと考えてみれば、これが実に理にかなっていることが分かりますよ。

これまで >>= を使うときには、普通の値を取るけどモナド値を返す関数にどうすればモナド値を渡せるか、ということばかり考えてきました。そんな関数でモナド値を写したら、モナド値の中にモナド値が入ったものが出てくるでしょう。例えば、モナド値 Just 9 と関数 \x -> Just (x+1) があるとします。この関数で Just 9 を写したら結果は Just (Just 10) になってしまいます。

m >>= f が常に join (fmap f m) と等しい、という事実は、ある種の型の Monad インスタンスを自作するときにとても便利です。これは、入れ子になったモナド値を平らにする方法を導くほうが、>>= の実装を導くより簡単なことが多いからです。

もう 1 つの興味深い事実は、join はファンクターやアプリカティブファンクターが提供する関数だけでは決して実装できない、ということです。このことから、モナドはファンクターやアプリカティブと同等の強さを持つにとどまらず、より強い力を持っていると結論付けられます。モナドには、ファンクターやアプリカティブよりも多くのことができるからです。

filterM

filter 関数は Haskell プログラミングの米といっても過言ではないでしょう（map が塩です）。filter は、述語とフィルタ対象のリストを取り、述語を満たす要素だけを残してくれます。filter の型はこうです。

```
filter :: (a -> Bool) -> [a] -> [a]
```

述語とは、リストの要素を 1 つとって Bool 値を返す関数です。では、述語が返す Bool がモナド値だったらどうしましょう？ 文脈がくっついてきたらどうしましょう？ 例えば、述語が生み出した True や False に、いちいち ["Accepted the number 5"] とか ["3 is too small"] といったモノイド値が付いていたら？ そういう場合、結果のリストにも、それまでに生まれたログが全部付いてきてほしいものです。このように、述語が返した Bool に文脈が付いてくるのなら、結果のリストにも適切な文脈が付いていることが期待されるわけです。そうでなければ、せっかくそれぞれの Bool に付いてきた文脈が失われることになります。

Control.Monad モジュールの filterM こそ、まさにそのための関数です！

```
filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
```

述語は Bool を結果とするモナド値を返していますが、返ってくるモナド値には、失敗の可能性から非決定性計算まで、どんな文脈が付いているか分かりません！ それでも最終結果に文脈がちゃんと反映されることを保証するために、結果もまたモナド値になっています。

リストを取って 4 より小さい要素だけを残す関数を作りましょう。まずは普通の filter 関数からです。

```
ghci> filter (\x -> x < 4) [9,1,5,2,10,3]
[1,2,3]
```

これは簡単ですね。では、True か False かを返すだけじゃなくて、何をしたかのログも残すような述語を作ってみましょう。もちろん Writer モナドを使います。

```
keepSmall :: Int -> Writer [String] Bool
keepSmall x
    | x < 4 = do
        tell ["Keeping " ++ show x]
        return True
    | otherwise = do
        tell [show x ++ " is too large, throwing it away"]
        return False
```

ただの Bool の代わりに、この関数は Writer [String] Bool を返しています。いわばモナディック述語です！ かっこいいじゃないですか？ もし引数が 4 より小さければ、「とっておくよ」と報告した上で return True します。

では、この述語とリストを filterM に与えてみましょう。この述語は Writer 値を返すので、結果のリストも Writer 値になります。

```
ghci> fst $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
[1,2,3]
```

Writer 値の結果を見る限り、すべてうまくいっているようですね。では、ログのほうを表示して、何が起こったのか見てみましょう。

```
ghci> mapM_ putStrLn $ snd $ runWriter $ filterM keepSmall [9,1,5,2,10,3]
9 is too large, throwing it away
Keeping 1
5 is too large, throwing it away
Keeping 2
10 is too large, throwing it away
Keeping 3
```

このとおり、モナディック述語を filterM に与えるだけで、使ったモナドの文脈を活用しながらリストをフィルタできるのです。

ここで、Haskell の超かっこいい技を紹介しましょう。 filterM を使って、あるリストの冪集合を作るという技です（リストを集合とみなすことにします）。ある集合の**冪集合**とは、その集合の部分集合をすべて含んだ集合です。例えば [1,2,3] という集合があったら、その冪集合の要素は以下の集合たちです。

```
[1,2,3]
[1,2]
[1,3]
[1]
[2,3]
[2]
[3]
[]
```

言い換えれば、ある集合の冪集合を作るというのは、その集合の各要素を残すか捨てるか、そのすべての場合を尽くすのと同じです。例えば、 [2,3] はもとの集合から 1 だけを除いた集合ですし、 [1,2] は同じく 3 を除いたものです。

あるリストの冪集合を計算する関数を作るには、非決定性計算に頼るのがよいでしょう。リスト [1,2,3] を取って、まずは先頭要素 1 をつくづく眺め、自らに問います。「これを残すべきか、除くべきか。それが問題だ。」 ええっと、両方やりたいんですけど。そこで非決定性計算を使い、リストの各要素に対して残すと除くの両方の答を返す述語を作ってリストをフィルタしちゃいましょう。これが俺たちの powerset 関数だ！

```
powerset :: [a] -> [[a]]
powerset xs = filterM (\x -> [True, False]) xs
```

えっ、これだけ？ ええ、そうなんです。どんな要素がきても平等に、残すか落とすか両方のチャンスを与えます。こうして非決定性を使うと、結果のリストも非決定的値、つまりリストのリストになります。では、動かしてみましょう。

```
ghci> powerset [1,2,3]
[[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]]
```

これはちょっと考えないと理解できないかも。「非決定な値としてのリストは、何になったらいいか分からないので同時にそのすべてになろうとしている」と思えば、非決定性計算という概念が少し理解しやすくなるかもしれません。

foldM

`foldl` のモナド版が `foldM` です。第 5 章で畳み込みをやったのを覚えていますか？ `foldl` は、2 引数関数とアキュムレータの初期値、畳み込みたいリストを引数に取り、2 引数関数を使って左から順番に畳み込んで 1 つの値にしてくれる関数でした。 `foldM` は同じことをしますが、ただしモナド値を返す 2 引数関数を取り、それを使ってリストを畳み込みます。ということは、 `foldM` が返す値がモナディックだったとしても驚きませんよね。 `foldl` の型は、こうです。

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

一方、 `foldM` の型はこうです。

```
foldM :: (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
```

2 引数関数の返す値がモナディックになっているので、畳み込み全体の結果もモナディックになってます。整数のリストの和を畳み込みを使って求めてみましょう。

```
ghci> foldl (\acc x -> acc + x) 0 [2,8,3,1]
14
```

アキュムレータの初期値は `0` です。これに `2` が足され、アキュムレータは `2` になります。続いて `8` が足され、アキュムレータは `10` になります。以下同様。こうしてリストの最後まで辿り着いたときのアキュムレータの値が答です。

では、整数のリストを加算したいが、リストのいずれかの要素が `9` より大きければ計算全体を直ちに失敗させたい、という問題だったらどうでしょう？ 対象が `9` より大きいかどうかを 2 引数関数に調べさせるのが素直でしょう。もしそうなら関数は失敗します。そうでなければ、関数は楽しげに仕事を続けます。このように失敗の可能性を追加したいのですから、2 引数関数は普通のアキュム

レータじゃなくて「Maybe アキュムレータ」を返すようにしましょう。これがその 2 引数関数です。

```
binSmalls :: Int -> Int -> Maybe Int
binSmalls acc x
    | x > 9     = Nothing
    | otherwise = Just (acc + x)
```

このように 2 引数関数をモナディックにしたのですから、普通の foldl と一緒には使えなくなってしまいました。代わりに foldM を使う必要があります。それゆけ！

```
ghci> foldM binSmalls 0 [2,8,3,1]
Just 14
ghci> foldM binSmalls 0 [2,11,3,1]
Nothing
```

素晴らしい！ 2 つ目の例では、リストに 1 つだけ、 9 より大きい数が入っていたので、計算全体が Nothing になっています。 Writer を返す 2 引数関数で畳み込みをするというのもすごいですよ。畳み込みが進行している間、起こったことをすべてログにとっておけますからね。

14.6　安全な逆ポーランド記法電卓を作ろう

第 10 章で逆ポーランド記法（RPN）の電卓を実装せよという問題を解いたときには、この電卓は文法的に正しい入力が与えられる限り正しく動くよ、という注意書きがありました。ところが、何か少しでも間違うとプログラム全体が落ちてしまうのでした。しかし、既存のコードをモナディックにする方法が分かった今、 Maybe モナドを使って、ぜひとも僕らの RPN 電卓にエラー処理機能を付けようじゃないですか！

さて、僕らの RPN 電卓の実装は、 "1 3 + 2 *" みたいな文字列を引数に取り、["1","3","+","2","*"] のような単語に分解してました。それから、数をスタックに積んだりスタックの上から数を取って足し算やら割り算やらを行う 2 引数関数を使って、空のスタックから始めてリストの内容を畳み込んだのでした。

これが関数の本体でした。

```
import Data.List

solveRPN :: String -> Double
solveRPN = head . foldl foldingFunction [] . words
```

数式を文字列のリストにしてから、専用の関数で畳み込みます。それから、スタックにただ 1 つ数が残っていることを期待して、それを答として返すという実装になっていました。これがそのとき使った畳み込み関数です。

```
foldingFunction :: [Double] -> String -> [Double]
foldingFunction (x:y:ys) "*" = (y * x):ys
foldingFunction (x:y:ys) "+" = (y + x):ys
foldingFunction (x:y:ys) "-" = (y - x):ys
foldingFunction xs numberString = read numberString:xs
```

この畳み込み関数のアキュムレータは Double 値のリストとして表現されたスタックです。この畳み込み関数が RPN 式を走査していくときは、もし現在のアイテムが演算子なら 2 つのアイテムをスタックのてっぺんから取り出し、演算を施し、結果をスタックに戻します。もし現在のアイテムが実数を表す文字列なら文字列を実際の数に変換し、古いスタックとほとんど同じだけどその数がてっぺんに積まれている新しいスタックを返します。

まず、畳み込み関数に優雅に失敗（graceful failure）する能力を与えましょう。型は、このように変わるはずです。

```
foldingFunction :: [Double] -> String -> Maybe [Double]
```

つまり、新しいスタックを Just にくるんで返すか、失敗した場合は Nothing を返すわけです。

reads 関数は read に似ていますが、読み取りに成功したときは単一要素リストを返します。もし読み込みに失敗した場合は空リストを返します。しかも、読み取れた値を返すばかりでなく、消費しきれなかった文字列も返します。今回は、入力文字列を全部読み込めなかった場合も失敗とみなすことにしましょう。そして、 readMaybe という便利な関数を作りましょう。こうです。

```
readMaybe :: (Read a) => String -> Maybe a
readMaybe st = case reads st of [(x, "")] -> Just x
                                _ -> Nothing
```

試してみましょう。

```
ghci> readMaybe "1" :: Maybe Int
Just 1
ghci> readMaybe "GOTO HELL" :: Maybe Int
Nothing
```

オッケー、動いているようです。じゃあ、畳み込み関数のほうも失敗する可能性のあるモナディック関数にしてみましょう。

```
foldingFunction :: [Double] -> String -> Maybe [Double]
foldingFunction (x:y:ys) "*" = return ((y * x):ys)
foldingFunction (x:y:ys) "+" = return ((y + x):ys)
foldingFunction (x:y:ys) "-" = return ((y - x):ys)
foldingFunction xs numberString = liftM (:xs) (readMaybe numberString)
```

最初の 3 つのパターンはおおむね昔のものと同じで、ただし新しいスタックは Just に包んで返しています（ここでは return を使って包んでいますが、Just と書いてもよいですよ）。最後のパターンでは、readMaybe numberString を使った後、(:xs) で写しています。例えば、スタック xs が [1.0,2.0] で、readMaybe numberString の結果が Just 3.0 だったら、結果は Just [3.0,1.0,2.0] になるわけです。もし readMaybe numberString の結果が Nothing だったら、全体の結果も Nothing になります。

畳み込み関数を単体で試してみましょう。

```
ghci> foldingFunction [3,2] "*"
Just [6.0]

ghci> foldingFunction [3,2] "-"
Just [-1.0]
ghci> foldingFunction [] "*"
Nothing
ghci> foldingFunction [] "1"
Just [1.0]
ghci> foldingFunction [] "1 wawawawa"
Nothing
```

どうやらちゃんと動いているようですね！ これでいよいよ改良版 solveRPN が書けますよ。皆さん、ご覧あれ！

```
import Data.List

solveRPN :: String -> Maybe Double
solveRPN st = do
    [result] <- foldM foldingFunction [] (words st)
    return result
```

まずは文字列を取って単語のリストに分けるところまでは以前のバージョンと同じです。次に、空のスタックから畳み込みを始めますが、 foldl の代わりに foldM を使っています。 foldM した結果は Maybe 値で、その中身にはスタックの最終状態がリストとして入っています。そしてそのリストは単一要素のはずです。ここで、 do 記法の中でその中身を取り出し、 result という名前をつけています。もし foldM が Nothing を返していたら、全体が Nothing になってくれるはずです。それが Maybe モナドの機能ですから。あと、 do 記法の中で

さりげなくパターンマッチを使ったのに気づきました？ リストにもし 2 つ以上の要素が入っていたり空だったりしたら、パターンマッチが失敗して、やっぱり Nothing が発生するはずです。最後の行では、 return result を使って、RPN 電卓の計算結果を Maybe 値として提示しています。

じゃあ、行ってみよう！

```
ghci> solveRPN "1 2 * 4 +"
Just 6.0
ghci> solveRPN "1 2 * 4 + 5 *"
Just 30.0
ghci> solveRPN "1 2 * 4"
Nothing
ghci> solveRPN "1 8 wharglbllargh"
Nothing
```

3 例目が失敗しているのは、最終状態のスタックが単一要素でないため、 do 式の中のパターンマッチが失敗しているからです。4 例目の失敗は、 readMaybe が Nothing を返しているからです。

14.7 モナディック関数の合成

第 13 章でモナド則を紹介したとき、 <=< 関数は関数合成によく似ているけど、普通の関数 a -> b ではなくて、 a -> m b みたいなモナディック関数に作用するのだよと言いました。例を挙げます。

```
ghci> let f = (+1) . (*100)
ghci> f 4
401
ghci> let g = (\x -> return (x+1)) <=< (\x -> return (x*100))
ghci> Just 4 >>= g
Just 401
```

この例では、まず普通の関数を 2 つ合成し、できた関数を 4 に適用しています。次に、モナディック関数を 2 つ合成し、できた関数に >>= を使って Just 4 を食わせています。

複数の関数をリストに入れて持っているとき、そのすべてを合成して 1 つの巨大な関数を作るには、 id をアキュムレータ、 . を 2 引数関数として畳み込めばよいでしょう。例えばこんなふうに。

```
ghci> let f = foldr (.) id [(+8),(*100),(+1)]
ghci> f 1
208
```

関数 f は、引数にまず 1 を足し、続いて 100 倍し、最後に 8 を足す関数です。

モナディック関数も同じように合成できますが、普通の関数合成の代わりに <=< を、 id の代わりに return を使えばよいですね。 foldr を foldM とかに変えたりする必要はありません。だって関数 <=< が、関数合成がモナド風に行われることを保証してくれますから。

第 13 章で、最初にリストモナドを導入したときは、ナイトの駒がチェス盤上のある位置から別の位置までちょうど 3 手で移動できるかどうかを判定するのに使ったのでした。あのときは、 moveKnight という、ナイトの現在位置を取って可能な手のリストを返す関数を作りました。そして 3 手後にどこにいられるかを知るために、こんな関数を作ったのでした。

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

それから、ナイトが start から end まで 3 手で行けるか試しました。

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

モナディック関数合成を使えば、 in3 みたいな関数を書けますし、しかも 3 手後の位置とかじゃなくて任意の手数に対応できます。 in3 をよく見ると、 moveKnight を 3 回使っていますが、どれも直前のナイトの位置を >>= で受け取っているのが分かります。じゃあ、もっと一般化してみましょう。こうです。

```
import Data.List

inMany :: Int -> KnightPos -> [KnightPos]
inMany x start = return start >>= foldr (<=<)
    return (replicate x moveKnight)
```

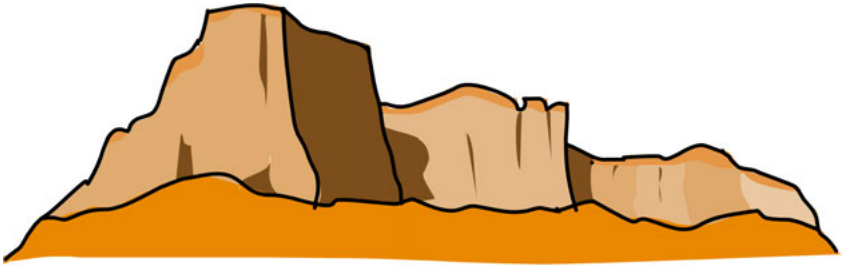
まず、 replicate を使って関数 moveKnight のコピーが x 個入ったリストを作ります。続いて、そのすべてをモナディックに合成して 1 つにすれば、初期位置を取ってナイトを非決定的に x 手動かす関数が出来上がります。それから初期位置の入った単一要素リストを return で作って、さっきの関数に渡せば完成。

となると、 canReachIn3 のほうも一般化できそうですね。

```
canReachIn :: Int -> KnightPos -> KnightPos -> Bool
canReachIn x start end = end `elem` inMany x start
```

14.8　モナドを作る

この節では、型が生まれてモナドであると確認され、適切な Monad インスタンスが与えられるまでの過程を、例題を通して学ぼうと思います。普通、モナドは作りたいと思って作るものではありません。むしろ、とある問題のある側面を



モデル化した型を作り、後からその型が文脈付きの値を表現していてモナドのように振る舞うと分かった場合に、Monad インスタンスを与える場合が多いです。

これまでにも見てきたように、リストは非決定な値を表現しています。例えば [3,5,9] のようなリストは、どの値を取るか決めかねている非決定的整数であると解釈できます。>>= を使ってリストを関数に食わせたら、リストからあらゆる可能な選択肢が取り出され、それぞれに関数が適用されて、結果がまたもリストとして提示されるわけです。

リスト [3,5,9] を、整数 3 、 5 、 9 が同時に存在している状態だと思って眺めていると、あれ、でもそれぞれの数の存在確率の情報がなくね？ と気づきませんか。[3,5,9] のような非決定的値を表現したいのだけど、さらに 3 である確率は 50 パーセント、5 と 9 はそれぞれ 25 パーセントである、ということを表したくなったらどうしましょう？ よし、やってみましょう！

リストの要素のそれぞれが、もう 1 つの値を伴っているとしましょう。その要素が実現する確率です。その値をこう表現するのはどうでしょう。

```
[(3,0.5),(5,0.25),(9,0.25)]
```

数学では、確率は普通パーセントじゃなくて、0 から 1 までの実数で表します。確率が 0 というのは絶対ありえないということで、確率が 1 というのはもう確実に起こるということです。確率を浮動小数で表してたら、すぐに精度が落ちてきて困ったことになるでしょう。ところが Haskell には分数のためのデータ型があります。その名は Rational で、Data.Ratio モジュールに入っています。Rational 型の値を作るには、分数のように書くだけです。分子と分母は % という記号で区切ります。例えば、こんな感じです。

```
ghci> 1%4
1 % 4
ghci> 1%2 + 1%2
1 % 1
ghci> 1%3 + 5%4
19 % 12
```

最初の例は四分の一（$\frac{1}{4}$）です。2 例目では、半分を 2 つ足して 1 を作っています。3 例目は $\frac{1}{3} + \frac{5}{4} = \frac{19}{12}$ という足し算です。もう浮動小数なんて使うのはやめて、確率は Rational で表すことにしましょう。

```
ghci> [(3,1%2),(5,1%4),(9,1%4)]
[(3,1 % 2),(5,1 % 4),(9,1 % 4)]
```

オッケー、これで 3 が出る確率は $\frac{1}{2}$ で、5 と 9 が出る確率はそれぞれ $\frac{1}{4}$ である、ということがきれいに表現できました。

さて、これはリストに対してさらに文脈を付加したものなので、こいつもきっと何らかの文脈付き値を表現しているのでしょう。でも先に進む前に、これを **newtype** で包んでおきませんか？ 何だか、これを何かのインスタンスにするような予感がするんです。

```
import Data.Ratio

newtype Prob a = Prob { getProb :: [(a, Rational)] } deriving Show
```

これってファンクターでしょうか？ ええ、リストはファンクターですから、リストに何かを足したものである Prob もたぶんファンクターでしょう。関数でリストを写すときは、リストの各要素に関数を適用したのでした。ここでも、各要素に関数を適用するのですが、確率はそのままにしておきましょう。では、インスタンスにしますよ。

```
instance Functor Prob where
    fmap f (Prob xs) = Prob $ map (\(x, p) -> (f x, p)) xs
```

パターンマッチを使って **newtype** から中身を取り出し、関数 f を値に適用した上で、確率はそのままに、再度 Prob で包みます。ちゃんと動くでしょうか？

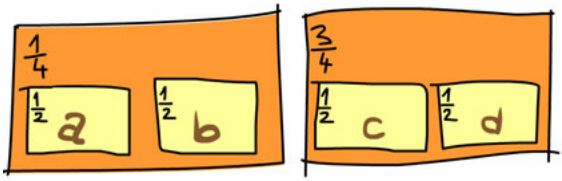
```
ghci> fmap negate (Prob [(3,1%2),(5,1%4),(9,1%4)])
Prob {getProb = [(-3,1 % 2),(-5,1 % 4),(-9,1 % 4)]}
```

確率の総和は常に 1 であることに注意しましょう。あるリストが、起こり得る場合をすべて尽くしているならば、その確率の和が 1 以外の何かになるというのは、まったく意味が通りません。例えば、投げると 75 パーセントの確率で裏が出て、50 パーセントの確率で表が出るコインというのは、ここではない、どこか不思議な宇宙でしかありえないコインです。

では、皆さんお待ちかねの大問題です。これってモナドでしょうか？ リストがモナドだったのだから、これもモナドであってほしい気がしますよね。まず、return について考えましょう。リストの return はどうでしたか？ 値を取って、単一要素リストに入れる関数でしたね。では Prob なら？ えっと、return はデフォルトの、最小限の文脈を持ってこないといけないのだから、きっと Prob

の return も単一要素リストを作るのでしょう。確率のほうはどうしましょう？ return x は、常に x を結果として提示するモナド値を作る関数でしたから、確率が 0 だったりしたら変ですよね。常にその値を示す、というのなら 1 に決まってます！

>>= のほうは？ これはちょっと難しいですよ。ここで、 m >>= f は常に join (fmap f m) と等価であることを使い、どうすれば確率リストの確率リストを平らにできるか考えましょう。例えば、 'a' か 'b' のいずれかが起こる確率はぴったり 25 パーセントであるとしましょう。そして 'a' と 'b' は同様に確からしいとします。さらに 'c' か 'd' のいずれかが起こる確率はきっかり 75 パーセントであるとしましょう。そして 'c' と 'd' もまた、同様に確からしいとしましょう。これが、このシナリオを表した絵です。



では、これらの文字が出る確率は、それぞれいくらでしょう？ これらの事象を 4 つの長方形で表したとしたら、それらの確率はいくらになるでしょう？ 確率を計算するには、それぞれの事象に寄与する確率をすべて掛け合わせればよいのです。 'a' が起こる確率は $\frac{1}{8}$ で、 'b' もそうです。なぜなら $\frac{1}{4}$ に $\frac{1}{2}$ を掛けると $\frac{1}{8}$ になるからです。 'c' が起こる確率は $\frac{3}{8}$ です。だって $\frac{3}{4} \times \frac{1}{2}$ は $\frac{3}{8}$ ですから。 'd' の確率も $\frac{3}{8}$ です。これらの確率を全部足すと、確かに 1 に戻りますね。

この状況を確率リストで表すと、こうなります。

```
thisSituation :: Prob (Prob Char)
thisSituation = Prob
    [(Prob [('a',1%2),('b',1%2)], 1%4)
    ,(Prob [('c',1%2),('d',1%2)], 3%4)
    ]
```

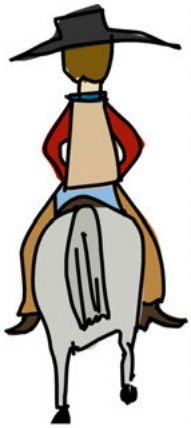
おや、こいつの型は Prob (Prob Char) ですね。では、入れ子になった確率リストを平らにする方法が分かった今、あとはどうやってコードに落とすかの問題です。そうすれば >>= は単に join (fmap f m) と書けて、モナドの一丁上がりです！ この関数を flatten と名づけましょう。 join という名前はもう使われてますからね。実装を以下に示します。

```
flatten :: Prob (Prob a) -> Prob a
flatten (Prob xs) = Prob $ concat $ map multAll xs
    where multAll (Prob innerxs, p) = map (\(x, r) -> (x, p*r)) innerxs
```

関数 multAll は、確率リストとある確率 p のタプルを取って、リストの中のすべての確率を p 倍して、事象と確率の組のリストを返す関数です。 flatten は、 multAll を入れ子確率リストの各要素に適用してまわり、得られた入れ子リストを最後に（リストとして）平らにします。

これで必要なものはすべて揃いました。今こそ Monad インスタンスを書くときです！

```
instance Monad Prob where
    return x = Prob [(x,1%1)]
    m >>= f = flatten (fmap f m)
    fail _ = Prob []
```



面倒な仕事はすでに済ませてあるので、インスタンス宣言自体はとてもシンプルです。ここでは fail 関数も追加されています。これはリストの fail 関数と同じものです。これにより、 do 記法の中でパターンマッチに失敗すると、その失敗は確率リストの文脈の中で処理されます。

できたばかりのモナドが、きちんとモナド則を満たしているかどうか試すことも、とても重要です。

1. モナド第一法則は、 return x >>= f と f x は等価であるべし、というものでした。厳密な証明はちょっと長くなるので省きますが、まずある値を return でデフォルト文脈に入れ、ある関数を fmap し、それから結果の確率リストを平らにしたとしても、関数が生み出すすべての確率に return が作った 1%1 が掛かるだけなので、文脈に影響はないことが分かると思います。
2. 第二法則は、 m >>= return は m と等価である、と言っています。この例の場合、第一法則の場合のような考察をすれば、 m >>= return が m と等しいことは明らかだと思います。
3. 第三法則は、 f <=< (g <=< h) と (f <=< g) <=< h が等価であれ、と言っています。これもまた確かに成り立っています。なぜなら確率モナドの基礎となるリストモナドは第三法則を満たしており、かつ、掛け算は結合法則を満たすからです。 1%2 * (1%3 * 1%5) は (1%2 * 1%3) * 1%5 と等しいですよね。

こうしてモナドが手に入ったからには何ができるでしょうか？ もちろん、確率的計算ができるようになったのです。確率的な事象を文脈付きの値として扱い、すべての要素の確率を反映して、確率モナドが最終結果の確率をきちんと計算してくれるのです。

さて、普通のコインが2枚と、10回投げると9回裏が出るよう細工されたコインが1枚あるとしましょう。これらのコインを全部同時に投げて、全部裏が出る確率っていくらでしょう？ まず、普通のコイン投げとイカサマコイン投げに対応する確率値を作ってみましょう。

```
data Coin = Heads | Tails deriving (Show, Eq)

coin :: Prob Coin
coin = Prob [(Heads,1%2),(Tails,1%2)]

loadedCoin :: Prob Coin
loadedCoin = Prob [(Heads,1%10),(Tails,9%10)]
```

そしてこれがコイン投げです。

```
import Data.List (all)

flipThree :: Prob Bool
flipThree = do
    a <- coin
    b <- coin
    c <- loadedCoin
    return (all (==Tails) [a,b,c])
```

こいつを動かすと、イカサマコインが紛れ込んでいるとはいえ3枚とも裏が出る確率っていうのがいうほど大きくはないってことが分かります。

```
ghci> getProb flipThree
[(False,1 % 40),(False,9 % 40),(False,1 % 40),(False,9 % 40),
 (False,1 % 40),(False,9 % 40),(False,1 % 40),(True,9 % 40)]
```

3枚とも裏が出る確率は $\frac{9}{40}$ です。これは25パーセントより少ないですね。3枚のうちのどれかが表になり、結果が `False` になるパターンは7通りあるんですが、どうやら僕らのモナドはこれらを統合する方法は知らないようです。まあでも、これは大した問題じゃありません。結果が一致する事象の確率をまとめる処理を書くのはとても簡単です（から、読者への演習問題とさせていただこうと思います）。

この節では、「リストが確率の情報も持っていたらどうなるだろう？」という疑問から出発して、型を作り、モナドを見切り、モナドのインスタンスを作って、何がしかの計算ができるようになりました。楽しんでいただけましたか？ ここまできたあなたは、モナドについて、モナドは何のためにあるのかについて、かなり理解が深まったのではないでしょうか？

第15章
Zipper

Haskell の純粋性は数多くの恵みをもたらしますが、ある種の問題については、非純粋な言語では使わないような手段で解決する必要も生じます。

Haskell は参照透明性を持つので、同じ値を表している 2 つの式の間に区別はありません。ここで例えば、要素が 5 ばっかりの木構造があり、そのどれか 1 つの要素を 6 に変えたいとします。すると、木の中の**どの** 5 を変更したいのか、明確に表現する方法が必要になります。木の中における変更点の位置を指定する必要があるのです。非純粋な言語では、該当の 5 が存在するメモリ上の位置を指定して書き換えれば済んだでしょう。ところが Haskell では、どの 5 も他の 5 と同じで、純粋な「5 という値」でしかなく、メモリ上の位置をもとにした識別は不可能です。

そもそも、Haskell では何かを**変える**ことはできません。「木を更新する」とは、実際には木を取ってそれとそっくりだけど少し異なっている木を返すことを意味します。

1 つの手は、木のルート（根元）から変更したい要素までの経路を覚えておくことです。「この木を取って、まず左に、次に右、また左に行って、そこの要素を変えてくれ」など。これは確かにちゃんと動きはするんですが、効率が悪くなりがちです。もし後で、さっき変更した要素の近くの要素を更新したくなっても、木をもう一度ルートから辿り直さなくてはなりません！

この章では、いくつかのデータ構造に、そのデータ構造の一部分に注目するための **Zipper** を備える方法を紹介します。Zipper はデータ構造の要素の更新を簡単にし、データ構造を辿るという操作を効率的にしてくれるんです。いいですね！

15.1　歩こう

生物の授業で習ったように、木にはいろんな種類があります。これから使う木の種を蒔きましょう。

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
```

この木は、空であるか、1 つの要素と 2 つの部分木を持つノードであるかのいずれかである、として定義されています。では今から、そんな木の一例を、なんと無料で読者にプレゼントしちゃいましょう。これです！

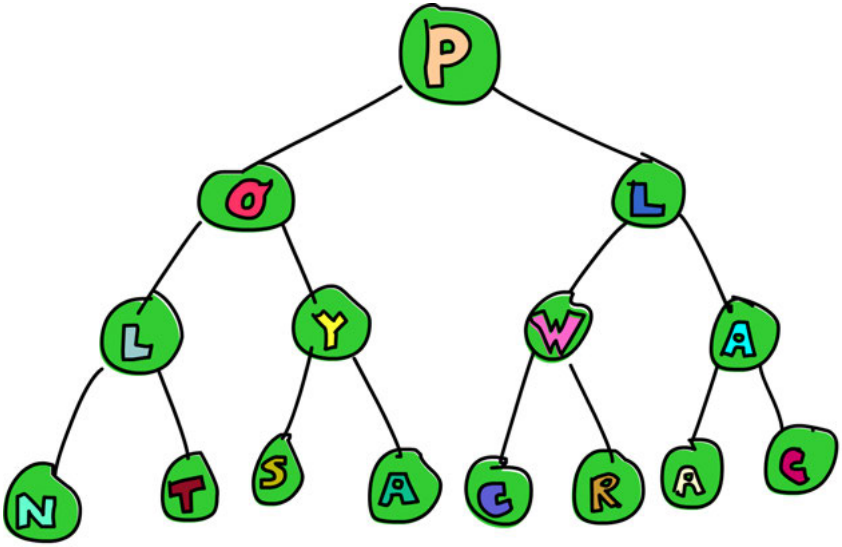
```
freeTree :: Tree Char
freeTree =
    Node 'P'
        (Node 'O'
            (Node 'L'
                (Node 'N' Empty Empty)
                (Node 'T' Empty Empty)
            )
            (Node 'Y'
                (Node 'S' Empty Empty)
                (Node 'A' Empty Empty)
            )
        )
        (Node 'L'
            (Node 'W'
                (Node 'C' Empty Empty)
                (Node 'R' Empty Empty)
            )
            (Node 'A'
                (Node 'A' Empty Empty)
                (Node 'C' Empty Empty)
            )
        )
```

そして、この木を図にしたものが次ページのイラストです。

ほら、木の中の `W` が見えますか？ あれを `P` に変えたいな。どうすればできるでしょう？ たぶん、1 つの方法は、その要素に辿り着くまでパターンマッチを繰り返すことです。まずは右に、次は左に。こうですね。

```
changeToP :: Tree Char -> Tree Char
changeToP (Node x l (Node y (Node _ m n) r)) =
    Node x l (Node y (Node 'P' m n) r)
```

ウゲェー！ このコードは醜いどころではありません。わけがわからないよ。一体全体何が起こっているのでしょう？ えええっと、まずルートの要素を `x` と名づけ（これは `freeTree` のルートにある `'P'` に合致します）、その左部分木を `l` とします。右部分木には、名前をつける代わりに、さらに深いパターンマッチを使います。こうしてパターンマッチを繰り返していって、やっと `'W'` をルー



トに持つ部分木に辿り着いたら、今度は集めた部品から木を再構成するわけですが、このときもともとルートに 'W' があった部分だけは 'P' を使います。

なんつーか、もっとマシな方法ってないんでしょうか？ 例えば、この関数が木と、方向のリストを引数に取るようにしたらどうでしょう？ 方向とは L か R のいずれかで、それぞれ左、右に対応し、方向指示に従って辿り着いた位置の値を更新するという具合です。こんなふうに。

```
data Direction = L | R deriving (Show)
type Directions = [Direction]

changeToP :: Directions -> Tree Char -> Tree Char
changeToP (L:ds) (Node x l r) = Node x (changeToP ds l) r
changeToP (R:ds) (Node x l r) = Node x l (changeToP ds r)
changeToP [] (Node _ l r) = Node 'P' l r
```

方向リストの先頭要素が L なら、元の木に似ているが、左部分木のどこかの要素が 'P' に置き換わっている木を作って返します。このとき changeToP を再帰的に呼び出すにあたって、方向リストの tail 部分を渡します。もう左には行きましたからね。先頭要素が R の場合も同様です。もし方向リストが空だったら、目的の場所に到着したってことなので、元の木のルート要素を 'P' に置き換えただけの木を返します。

木全体を出力するのも大変ですから、方向リストを取って、目的地にある要素を返す関数も作りましょう。

```
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```

この関数、よく見たら changeToP にそっくりです。違いといえば、経路に沿って出会ったものを覚えておいて木を再構築するのではなく、目的物以外のすべてを無視するようにできていることぐらいです。では、'W' を 'P' に書き換えた上で、書き換えがちゃんと動いているか確かめてみましょう。

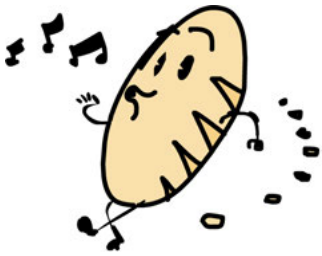
```
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

動いてるっぽいですね。この 2 つの関数では、方向リストが与えられた木の特定の部分木、いわば**注目点**を指定する役割を果たしています。例えば [R] という方向リストは、ルートの右側の部分木を表しています。空の方向リストは木全体を表します。

この技はかっこいいと思いました？ でも、これだと効率が悪い場合があるんです。特に要素の更新を何度もしたい場合は。もしも、とても巨大な木の末端のほうを長い長い方向リストで指定して、やっと書き換えたところに、また似たような位置の要素を書き換えたくなったら？ またはじめから辿り直しですよ？ 超だるい！

次の節では、部分木に注目するためのもっと良い方法を探すことにします。ある部分木から近場の部分木へと効率的に切り替えられるような方法です。

背後に残った道しるべ



部分木に注目するのに、毎回ルートから辿り直す必要のある方向リスト以外の手段を考えたほうが良さそうです。ルートから左、右、と進むときに、道しるべとなるパンくずを残したらどうでしょう？ 左へ行ったときには「さっきは左だった」、右へ行ったときには「あそこで右だった」と、往路の履歴を覚えておけば、逆方向の移動が作れそうです。やってみましょう！

パンくずを表現するのにも方向値（L と R）を使いましょう。ただし、Directions と呼ぶ代わりに Breadcrumbs という名前にしましょう。木構造を下るときにパンくず（Breadcrumbs）を残していくと、方向としては逆向きになるので。

```
type Breadcrumbs = [Direction]
```

これは、木とパンくずリストを受け取って、木を左部分木へと辿りながら L をパンくずリストの先頭に追記する関数です。

```
goLeft :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goLeft (Node _ l _, bs) = (l, L:bs)
```

ルート要素と右部分木は無視して、左部分木と、L が先頭に追加されたパンくずリストを返しています。

右に行く関数はこちら。

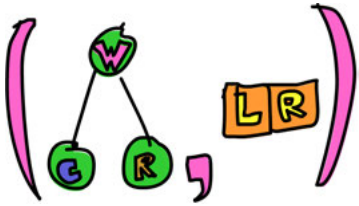
```
goRight :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goRight (Node _ _ r, bs) = (r, R:bs)
```

左に行く関数と同じ仕組みです。

これらの関数を使って、例の freeTree を右、続いて左に辿ってみましょう。

```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

これで 'W' をルートに、'C' を左部分木のルートに、'R' を右部分木のルートに持つ木が出てきました。パンくずリストは [L,R] です。まず右に、次に左に行きましたからね。



木を辿る操作をもっときれいに書くために、第 13 章で定義した関数 -: を使いましょう。関数 -: の定義はこうでした。

```
x -: f = f x
```

これで関数を値に適用するとき、「値 -: 関数」と書けるようになります。例えば、goRight (freeTree, []) と書く代わりに、(freeTree, []) -: goRight と書けます。この形式を使ってさっきの例を書き直すと、まず右、次に左に行った、というのが分かりやすくなりますよ。

```
ghci> (freeTree, []) -: goRight -: goLeft
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

来た道を戻る

木を逆方向に辿り直すにはどうすればよいのでしょう？ パンくずリストからは、今の木が親の木の左部分木だったこと、そしてその親は、親の親の木の右部分木だったことは分かります。が、それがすべてです。パンくずリストだけで

は、木構造を上向きに辿るための十分な情報がありません。1 つひとつのパンくずの中に、木を辿った方向だけじゃなくて、木構造を戻るために必要なデータをすべて蓄えておく必要があるようです。今回の場合、それは親の木のルート要素および右部分木です。

一般的に言って、1 つのパンくずには親ノードを構築するのに必要なすべてのデータを蓄えておく必要があります。つまり、辿った方向だけじゃなく、辿る可能性があった経路の情報も必要です。ところが、今注目している部分木の情報まで含んではいけません。なぜなら注目している部分木の情報はタプルの第一要素に既出だからです。これまでパンくずに含まれていると情報が重複してしまうでしょう。

情報が重複するのは避けたいです。なぜなら、注目している部分木に何らかの変更が加わった場合、パンくずリストの中にある情報との間で一貫性が壊れてしまうからです。注目点の中で少しでも変更があると、重複している情報は期限切れで無効になります。しかも、もし木の要素数が多かったらメモリもバカ食いしちゃいますよね。

では、パンくずリストを改良し、これまで左右に移動するときに無視してきた情報をすべて含むようにしましょう。 `Direction` に代わる新しいデータ型を作ります。

```
data Crumb a = LeftCrumb a (Tree a)
             | RightCrumb a (Tree a) deriving (Show)
```

今度は、ただの `L` に代わって `LeftCrumb` というコンストラクタがあり、移動元のノードに含まれていた要素と、辿らなかった右部分木をも持つようになっています。また `R` に代わって `RightCrumb` があり、やはり移動元のノードに含まれていた要素と、辿らなかった左部分木を持っています。

この新しいパンくずリストなら、辿ってきた木を再構築できるだけの情報をすべて含んでいます。もはやただのパンくずリストというより、分岐点に行きあたるたびに残してきたフロッピーディスクのようなものです。辿った方向よりもずっと多くの情報を含んでいるわけですから。

新しいパンくずは、本質的には「穴」のついた木のノードのようなものです。木を 1 階層辿るごとに、パンくずには出発点のノードの情報のうち「今注目することを選んだ部分**を除く**すべての情報」が書き込まれます。あと、穴がどこにあったのかも記録する必要がありますね。 `LeftCrumb` の場合では、左に移動したことは知っているので、穴は左部分木のところに空いているわけです。

型シノニム `Breadcrumbs` にもこの更新を反映させましょう。

```
type Breadcrumbs a = [Crumb a]
```

続いて、goLeft 関数と goRight 関数も修正し、辿らなかった経路の情報を捨てるのではなくパンくずリストに記録するようにしないといけませんね。まず、これが goLeft です。

```
goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

さっきの goLeft ととてもよく似ていますね。ただし、左に行ったことを記録するとき、パンくずリストの先頭に L ではなく LeftCrumb を追加しています。そして LeftCrumb には移動元のノードに入っていた要素 x と、選ばれなかった右部分木の情報を載せています。

この関数では、引数にきた木が Empty じゃないことを仮定している点に注意が必要です。空の木には部分木がありませんから、空の木から左に行こうとしたらエラーが出ます。これは Node に対するパターンマッチが成功せず、Empty を受理するパターンはないからです。

goRight も似たようなもんです。

```
goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goRight (Node x l r, bs) = (r, RightCrumb x l:bs)
```

さて、左や右に行くことなら、これまでも可能でした。今、さらに親ノードの情報と辿らなかった経路の情報を揃えたことで、経路を戻る能力を手に入れたはずです。これが goUp 関数です。

```
goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

木 t に注目している状態で、最新の Crumb を調べます。それが LeftCrumb であれば、現在の木である t を左部分木に使い、Node の残りの部分は、パンくずからの右部分木とルート要素の情報を使って埋めます。そして、「履歴を戻る」操作をするためにリストの先頭のパンくずを拾ってしまったわけですから、新しいパンくずリストからは先頭要素を除いておきます。

木のてっぺんにいる場合に、さらに上に戻ろうとしてこの関数を使うと、やっぱりエラーになることに注意です。でも大丈夫。後で Maybe モナドを使って、注目点を移動しようとしたときに起こり得る失敗を表現するようにしますから。

Tree a と Breadcrumbs a のペアは、元の木全体を復元するのに必要な情報に加えて、ある部分木に注目した状態というのを表現しています。このスキームなら、木の中を上、左、右へと自由自在に移動できます。

あるデータ構造の注目点、および周辺の情報を含んでいるデータ構造は **Zipper** と呼ばれます。注目点をデータ構造に沿って上下させる操作は、ズボンのジッパーを上下させる操作に似ているからです。というわけで、こんな型シノニムを定義しておくといいんじゃないかな？！

```
type Zipper a = (Tree a, Breadcrumbs a)
```

僕としては、 Focus という名前のほうが好きです。データ構造の一部分に注目しているってことをよく表す名前になりますからね。ですが、このような設計を表す名前としては Zipper のほうが広く普及しているので、ここは大勢に従っておきましょう。

注目している木を操る

さて、これで上下に動けるようになりましたから、ジッパーが注目している部分木のルート要素を書き換える関数を作りましょう。

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

注目点がノードである場合、関数 f を使ってそのノードのルート要素を書き換えます。もし注目点が空の木なら、そっとしておきます。これでどんな木でも、好きなところへ辿っていき、要素を修正しつつも常に注目点を忘れず、いつでもまた上下に移動できるように保てます。例えば、「まず左へ行き、次に右へ行き、ルート要素を 'P' で置き換えるという修正を施す」という操作なら、こう書けるわけです。

```
ghci> let newFocus = ⇨
            modify (\_ -> 'P') (goRight (goLeft (freeTree, [])))
```

-: を使えば、もっと見やすくなりますよ。

```
ghci> let newFocus = ⇨
            (freeTree, []) -: goLeft -: goRight -: modify (\_ -> 'P')
```

望みとあらば、さらに 1 つ上へ行き、そこの要素を謎めいた 'X' に置き換えることもできます。

```
ghci> let newFocus2 = modify (\_ -> 'X') (goUp newFocus)
```

もしくは -: を使ってこう書けます。

```
ghci> let newFocus2 = newFocus -: goUp -: modify (\_ -> 'X')
```

上への移動がここまで簡単になったのは、残してきたパンくずリストがデータ構造のうちの現在注目していない点を構成しており、しかも「逆向きに」、いわば裏返した靴下のようになっているからです。だからこそ上に移動するときにも、ルートから延々と下へ辿り直す必要はなく、裏返った木の先頭要素を取ってきて、表向きに直し、現在の注目点に継ぎ足せばよいのです。

どのノードも 2 つの部分木を持ちますが、それは空かもしれません。ですから、空の部分木に注目している場合、注目点を空でない部分木で置換すれば、木を別の木の先端に継ぎ足す操作が作れます。そのためのコードもシンプルです。

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_, bs) = (t, bs)
```

木とジッパーを取って、注目点を与えられた木で置き換えた新しいジッパーを返しています。こいつは、空の木を置換して木を伸ばすことだけじゃなく、既存の部分木を置き換えるのにも使えますね。それでは、お馴染みの `freeTree` の一番左に新しい木を取り付けてみましょう。

```
ghci> let farLeft = ⇨
            (freeTree, []) -: goLeft -: goLeft -: goLeft -: goLeft
ghci> let newFocus = farLeft -: attach (Node 'Z' Empty Empty)
```

これで、 `newFocus` は取り付けたばかりの新しい木に注目していて、木の残りの部分は裏返しでパンくずリストに入っている状態が作れました。ここから `goUp` で木のてっぺんまで戻ると、 `freeTree` に対して左端に `'Z'` が増えている木が作れるはずです。

真っすぐ、てっぺんまで行って、新鮮でおいしい空気を吸おう！

今どこに注目しているかによらず、木のてっぺんまで移動する関数はすごく簡単に作れます。こうです。

```
topMost :: Zipper a -> Zipper a
topMost (t, []) = (t, [])
topMost z = topMost (goUp z)
```

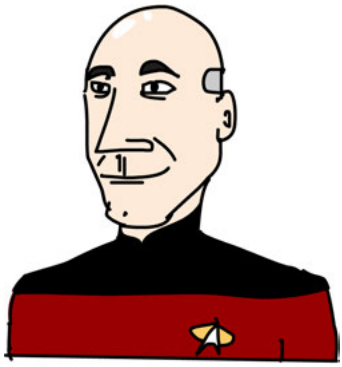
パンくずリストが空なら、すでにルートにいるということですから、現状の注目点をそのまま返せばよいですね。そうでなければ、 `goUp` で親ノードに注目した上で、再帰的に `topMost` を使います。

これで、木の中を歩き回り、左に右に、また上に行ったり、道すがら `modify` や `attach` を適用してまわったりできるようになりました。そうして修正が済んだら、 `topMost` を使ってルートまで上がり、これまで加えてきた変更を一望しましょう。

15.2　リストに注目する

ジッパーは、ほぼどんなデータ型に対しても作れるので、リストと部分リストに対して作れるといっても不思議ではないでしょう。なにしろ、リストだって木のようなもんです。ただ、一般的な木構造は各ノードに要素（のない木もありますが）と複数の部分木を持たせたものであるのに対し、リストは各ノードに 1 つの要素と 1 つの部分リストを持たせたものです。第 7 章で自作のリストを作ったときにも、そうやってデータ構造を定義しましたね。

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```



これを、先ほどの二分木の定義と比べると、まさに木は先頭要素（head）と先頭以外の要素（tail）からなる「一分木」であると察していただけるかと思います。

例えば、`[1,2,3]` というリストは、`1:2:3:[]` と書き直せます。こいつは、head である `1` と、tail である `2:3:[]` からなっています。`2:3:[]` にも head である `2` と、tail である `3:[]` があります。そして `3:[]` にとっては、`3` が head で、tail は空リスト `[]` です。

では、リストのジッパーを作ってみましょう。部分リストの間での注目点の移動は前、後方向と呼ぶことにしましょう（木の場合は上、左、右が移動可能な方向でしたね）。注目点は部分リストであり、それに加えて、前に進むたびにパンくずリストを残していくことにしましょう。

さて、リストに関するパンくずには、1 かけらあたりどんな情報を入れればよいでしょう？ 二分木を扱うパンくずのときは、親ノードのルート要素の情報に加えて、選ばれなかった部分木全体の情報も持たせる必要がありました。それから、左へ行ったのか右へ行ったのかも記録しておく必要がありました。要するに、親ノードに含まれていた情報のうち、選択して注目することにした部分木に含まれない情報はすべて保持しておく必要があったわけです。

リストは木よりも単純です。まず、左右どちらに分岐したかは記録する必要がありません。リストを深いほうに辿る選択肢は 1 つしかないからです。しかもノードの部分木も 1 つしかないので、選ばなかった経路を記録する必要もありません。どうやら、親ノードの持っていた要素だけを記録すれば済みそうです。リスト `[3,4,5]` があったときに、直前の要素は `2` だったことさえ知っていれば、その要素をリストの head に戻して `[2,3,4,5]` を復元できます。

二分木のジッパーを作ったときは、パンくずを表す Crumb というデータ型を作りましたが、リストにとって 1 かけらのパンくずは単なる要素なので、わざわざデータ型に包むまでもありません。

```
type ListZipper a = ([a], [a])
```

第一のリストは注目しているリストを表し、第二のリストはパンくずリストに対応します。では、リストを前後に移動する関数を作りましょう。

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs, x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

リストを前に進むときは、現在のリストの tail を新しい注目点とし、head をパンくずとして残します。リストを後ろに戻るときは、最新のパンくずを取り出して、現在のリストの先頭にくっつけます。この 2 つの関数を動かしてみましょう。

```
ghci> let xs = [1,2,3,4]
ghci> goForward (xs, [])
([2,3,4],[1])
ghci> goForward ([2,3,4], [1])
([3,4],[2,1])
ghci> goForward ([3,4], [2,1])
([4],[3,2,1])
ghci> goBack ([4], [3,2,1])
([3,4],[2,1])
```

このように、リスト構造に対するパンくずリストというのは、元のリストを逆順にしたものにほかなりません。注目しているリストから取り除いた要素は、パンくずリストの先頭に継ぎ足されていきます。そうしておけば、パンくずリストの先頭から要素を取り出しては注目しているリストに継ぎ足すだけで、いとも簡単にデータ構造を逆戻りできます。こうしてみると **Zipper** という名前の由来にも納得がいきますね。ジッパーのスライダーが上下に動いているようすにほんとそっくりです。

もしあなたが、テキストエディタを作ることになったら、今開いているファイルの各行を文字列のリストとして表現するのも 1 つの手です。そうしてジッパーを使えば、カーソルのある行を表現することができ、テキストの任意の箇所に新しい行を挿入したり削除したりする操作も簡単に書けますよ。

15.3　超シンプルなファイルシステム

ジッパーを使ったデモとして、ごく単純化したファイルシステムを木で表現してみましょう。そのファイルシステムに対するジッパーを作り、本物のファイルシステムみたいにフォルダ間を移動できるようにしましょう。

よくある階層的なファイルシステムは、ファイルとフォルダからなります。**ファイル**は名前のついたデータの塊です。**フォルダ**はそれらファイルを整理するためのもので、複数のファイルやフォルダを含むことができます。今回の単純な例では、ファイルシステム内のアイテムは次のいずれかであるとしましょう。

- ファイル：名前がついていて、データが入っている。
- フォルダ：名前がついていて、複数のファイルやフォルダをアイテムとして含む。

何が何だかよく分かるように、データ型と型シノニムを作りましょう。

```
type Name = String
type Data = String
data FSItem = File Name Data | Folder Name [FSItem] deriving (Show)
```

ファイルは 2 つの文字列からなります。1 つ目はファイル名で、2 つ目が中身のデータを表しています。フォルダは、フォルダ名を表す文字列と、中身のアイテムのリストからなります。中身リストが空だったら、そいつは空フォルダってことです。

いくつかのフォルダやサブフォルダが入っているフォルダの例です（実はこれ、僕のディスクの今の中身です）。

```
myDisk :: FSItem
myDisk =
    Folder "root"
        [ File "goat_yelling_like_man.wmv" "baaaaaa"
        , File "pope_time.avi" "god bless"
        , Folder "pics"
            [ File "ape_throwing_up.jpg" "bleargh"
            , File "watermelon_smash.gif" "smash!!"
            , File "skull_man(scary).bmp" "Yikes!"
            ]
        , File "dijon_poupon.doc" "best mustard"
        , Folder "programs"
            [ File "fartwizard.exe" "10gotofart"
            , File "owl_bandit.dmg" "mov eax, h00t"
            , File "not_a_virus.exe" "really not a virus"
            , Folder "source code"
                [ File "best_hs_prog.hs" "main = print (fix error)"
                , File "random.hs" "main = print 4"
                ]
            ]
        ]
```

このファイルシステムのジッパーを作ろうぜ！

これでファイルシステムはできたので、あとはジッパーさえあれば、ザッピングもズームインも自由、ファイルの追加・編集・消去も自在にできますね。二分木やリストでもやったように、パンくずリストは、ここに行く！ と決めたもの以外のすべての情報を含む必要があります。1 かけらのパンくずには、今注目している部分木以外のすべてを保存するべきです。また、「穴」の位置も覚えておかないと、上に戻ったときに直前の注目点を適切な位置に埋め戻すことができませんね。

今回の場合、パンくずはフォルダにそっくりなデータ構造になるはずです（ただし、今選択されているフォルダは含まれてません）。「あれ、じゃあファイルは？」と思いましたか？ えっと、ひとたびファイルに注目したら、ファイルシステムをそれ以上深く辿ることはできません。ですから、「ファイルからきました」というパンくずが残されることはないはずです。ファイルは、あたかも空の木のような役割なのです。

例えば、 `"root"` フォルダに注目している状態から `"dijon_poupon.doc"` に注目している状態に移るときは、どんなパンくずを残せばいいでしょうか？ ええと、まず親フォルダの名前と、注目しているアイテムの前後にくるべき他のアイテムたちですね。ですから、 `Name` が 1 つと、アイテムのリストが 2 つ必要です。現在のアイテムの前にあったアイテムのリストと、後ろにあったアイテムのリストを分けておくことで、戻るときに現在のアイテムをフォルダのどこに戻せばいいか、ちゃんと分かります。そうやって穴の位置を記録するわけです。

これが僕らのファイルシステムのパンくずの型です。

```
data FSCrumb = FSCrumb Name [FSItem] [FSItem] deriving (Show)
```

そしてこれがジッパーです。

```
type FSZipper = (FSItem, [FSCrumb])
```

階層構造を上に戻るのはとても簡単です。最新のパンくずを取って、現在の注目点とそのパンくずとから、こうやって新しい注目点を作ります。

```
fsUp :: FSZipper -> FSZipper
fsUp (item, FSCrumb name ls rs:bs) =
    (Folder name (ls ++ [item] ++ rs), bs)
```

パンくずには、フォルダの名前、フォルダの中で注目点より前にあったアイテムのリスト（その名は ls）、注目点より後ろにあったアイテムのリスト（その名は rs）が全部入っていますから、上に戻るのは簡単です。

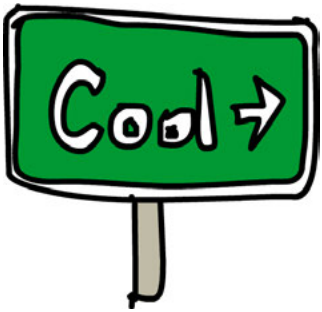
ではファイルシステムを奥に辿るのはどうでしょう？ 今 "root" にいるとして、ファイル "dijon_poupon.doc" に注目したいときは、後に残すパンくずにはフォルダ名 "root" と、"dijon_poupon.doc" よりも前にあったアイテム、後ろにあったアイテムのリストを残す必要があります。これが、アイテム名を引数に取って、現在のフォルダの中にあるファイルまたはフォルダに注目点を移す関数です。

```
import Data.List (break)

fsTo :: Name -> FSZipper -> FSZipper
fsTo name (Folder folderName items, bs) =
    let (ls, item:rs) = break (nameIs name) items
    in  (item, FSCrumb folderName ls rs:bs)

nameIs :: Name -> FSItem -> Bool
nameIs name (Folder folderName _) = name == folderName
nameIs name (File fileName _) = name == fileName
```

fsTo は、Name と FSZipper を引数に取り、その名前を持つアイテムに注目した新しい FSZipper を返します。引数に与えられた名前を持つアイテムは、フォルダ内に存在しないといけません。この関数は、ファイルシステム全体を検索してくれるわけではありません。カレントフォルダ内だけを探します。



まず break を使って、今探しているアイテムよりも前にあるものと後にあるものとにアイテムのリストを分けます。break は、述語とリストを引数に取り、リストのペアを返す関数です。ペアの第一要素はすべて、述語が False を返すような要素です。その後、述語が True を返すようなアイテムが 1 つでも見つかればそいつを含め、残りのアイテムは全部第二要素に入ります。ここでは述語として、名前とファイルシステムアイテムを引数に取り、名前が一致していれば True を返す補助関数 nameIs を作って使っています。

こうして、探していたアイテムより前にあるアイテム ls 、探していたもの item 、探していたものより後にいたもの rs が分類できました。この 3 つが break から取れたなら、あとは見つかったアイテムを注目点として提示し、パンくずに全部のデータを詰めれば完成です！

探している名前を持つアイテムがフォルダにないときは、パターン item:rs が空リストに合致しようとしてエラーが出ることに注意してください。また、注目点がフォルダでなくてファイルである場合も、同じくエラーが出てプログラムはクラッシュします。

これで、ファイルシステムの中を上下方向に移動できるようになりました。ルートから "skull_man(scary).bmp" というファイルまで辿ってみましょう。

```
ghci> let newFocus = ⇨
            (myDisk, []) -: fsTo "pics" -: fsTo "skull_man(scary).bmp"
```

こうすれば newFocus は "skull_man(scary).bmp" に注目しているはずです。本当にそうなっているか、ジッパーの第一要素（注目点そのもの）をちょっと表示してみましょう。

```
ghci> fst newFocus
File "skull_man(scary).bmp" "Yikes!"
```

じゃあ、今度は上に行って、近所のファイル "watermelon_smash.gif" を見てみましょう。

```
ghci> let newFocus2 = newFocus -: fsUp -: fsTo "watermelon_smash.gif"
ghci> fst newFocus2
File "watermelon_smash.gif" "smash!!"
```

ファイルシステムの操作

ファイルシステムの中を移動できるようになった今となっては、ファイルシステムを操作するのも朝飯前です。まずは、注目しているファイルもしくはフォルダの名前を変更する関数です。

```
fsRename :: Name -> FSZipper -> FSZipper
fsRename newName (Folder name items, bs) = (Folder newName items, bs)
fsRename newName (File name dat, bs) = (File newName dat, bs)
```

さっそく "pics" フォルダの名前を "cspi" にしてみましょう。

```
ghci> let newFocus = ⇨
            (myDisk, []) -: fsTo "pics" -: fsRename "cspi" -: fsUp
```

"pics" フォルダまで降りていって、名前を変え、再び上まで登りました。では、現在のフォルダにアイテムを新規作成する関数は？ このとおり！

```
fsNewFile :: FSItem -> FSZipper -> FSZipper
fsNewFile item (Folder folderName items, bs) =
    (Folder folderName (item:items), bs)
```

ちょろいもんだぜ。ただし、この関数はフォルダじゃなくてファイルに注目してるジッパーに使うとクラッシュするので注意が必要です。

では "pics" フォルダにファイルを追加して、ルートまで戻ってみましょう。

```
ghci> let newFocus = ⇨
          (myDisk, []) -: fsTo "pics" ⇨
          -: fsNewFile (File "heh.jpg" "lol") -: fsUp
```

これの何がすごいかって、ファイルシステムを更新したときに、データ構造自体に修正が上書きされるのではなく、関数から新しいファイルシステム全体が返ってくることです。この方式なら、古いファイルシステム（この場合はmyDisk）にも新しい版（この場合は newFocus ）にも同時にアクセスできます。

このようにジッパーを使えば、何もしなくてもバージョン管理が付いてきます。データ構造を書き換えた後でも、旧バージョンのデータに何の問題もなくアクセスできます。これは何もジッパーに限った話ではなく、Haskell の性質です。Haskell のデータ構造は immutable（一度定義すると変更できないもの）だからです。ところがジッパーがあれば、そんな immutable なデータ構造の中を楽に効率よく移動できるようになり、Haskell のデータ構造の永続性がいよいよ輝き始めます[†1]。

15.4　足下にご注意

これまで二分木、リスト、ファイルシステムといったデータ構造のジッパーを作ってきましたが、いずれも、足下を確かめずに歩き崖から落ちても気にしないという実装でした。例えば、 goLeft 関数は二分木のジッパーを取って左部分木に注目点を移します。

```
goLeft :: Zipper a -> Zipper a
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

でも、今の足場が空の木だったら？ つまり Node ではなく Empty だったら？ すると、 Node へのパターンマッチは失敗し、部分木を持たない空の木と合致するパターンはないので、実行時エラーを食らってしまいます。

今までのところは、空の木の左部分木に注目しようとする奴なんていないだろう、そんな部分木なんてないんだし、と想定していています。空の木の左部分木に行く処理なんて意味が分かりませんし、便宜のために単に無視してきたわけです。

†1　［訳注］それでも、自作のデータ構造にいちいちジッパーなんて作ってらんないよ、と思いましたか？ それこそ Haskeller にふさわしい怠け心です！ その怠け心こそが、ほぼあらゆる型に対してジッパーを自動的に作ってくれる Data.Generics.Zipper モジュールを生み出した力なのです。

また、何らかの木のルートにいてパンくずリストが空のときに、さらに上に戻ろうとしたら？ 同じエラーが発生するでしょう。どうやらジッパーを使っているときは、一歩踏み出すたび、それが人生最期の一歩になる覚悟がいるようです（ここで不吉な BGM 入る）。いかなる移動も、運良く成功するかもしれませんが、いつ失敗するかも分からないのです。これ、どこかで聞いた台詞ですよね？ そう、モナドです！ 具体的には、Maybe モナド、普通の値に失敗の可能性という文脈を追加するモナドです。

では、 Maybe モナドを使って、ジッパーの移動に失敗可能性という文脈を追加しましょう。二分木を処理するジッパーをモナディック関数に変えてみましょう。

まず、 goLeft と goRight が引き起こす可能性のある失敗をケアしましょう。これまで、失敗するかもしれないという性質は、常に関数の返り値に反映されてきました。今回も例外ではありません。

これが失敗の可能性が追加された goLeft と goRight です。

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (Node x l r, bs) = Just (l, LeftCrumb x r:bs)
goLeft (Empty, _) = Nothing

goRight :: Zipper a -> Maybe (Zipper a)
goRight (Node x l r, bs) = Just (r, RightCrumb x l:bs)
goRight (Empty, _) = Nothing
```

これで、空の木の左部分木を取ろうとしたら Nothing が返るようになりました！

```
ghci> goLeft (Empty, [])
Nothing
ghci> goLeft (Node 'A' Empty Empty, [])
Just (Empty,[LeftCrumb 'A' Empty])
```

よさげですね！ では、上に行く場合は？ 問題が起こるのは、上に行こうとしたときにパンくずが 1 つもない場合、つまりすでにルートにいた場合です。これが、木構造の境界に木(気)をつけていないとエラーを投げる goUp 関数です。

```
goUp :: Zipper a -> Zipper a
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

もっと行儀よく失敗するようにしましょう。

```
goUp :: Zipper a -> Maybe (Zipper a)
goUp (t, LeftCrumb x r:bs) = Just (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = Just (Node x l t, bs)
goUp (_, []) = Nothing
```

パンくずがあればそれでよし、新しい注目点を「成功した計算」の文脈に入れて返します。パンくずがなければ失敗を返します。

以前は、この関数はジッパーを取ってジッパーを返していましたので、こんな感じで連鎖させて木の中を歩き回ることができました。

```
ghci> let newFocus = (freeTree, []) -: goLeft -: goRight
```

ところが今は、 Zipper a を返す代わりに Maybe (Zipper a) を返すようになったので、上記のような関数適用では連鎖できなくなりました。第 13 章でピエールの綱渡りを扱ったときにも同じ問題にぶつかりましたね。ピエールも、一歩ごとに失敗するかもしれない綱渡りを強いられていました。バランス棒の片側に鳥がとまりすぎると落ちてしまうからです。

やれやれ、あの冗談を我が身で味わう羽目になるとは。今は僕ら自身が、自分で生み出したデータ構造の迷宮の中を、危険を冒して歩いているのです。幸運なことに、僕らは綱渡りのピエールから学び、彼のやったことを真似できます。関数適用の代わりに >>= を使えばよいのです。 >>= は、文脈付きの値（今の場合、Maybe (Zipper a) という失敗する可能性の文脈）を取って関数に食わせつつ、文脈が適切に処理されることを保証してくれます。ですから、ピエールがしたように、すべての -: 演算子を >>= 演算子で置き換えましょう。それだけで再び関数を連鎖できるようになります。ご覧ください。

```
ghci> let coolTree = Node 1 Empty (Node 3 Empty Empty)
ghci> return (coolTree, []) >>= goRight
Just (Node 3 Empty Empty,[RightCrumb 1 Empty])
ghci> return (coolTree, []) >>= goRight >>= goRight
Just (Empty,[RightCrumb 3 Empty,RightCrumb 1 Empty])
ghci> return (coolTree, []) >>= goRight >>= goRight >>= goRight
Nothing
```

まず、左の枝に空の木を持ち右の枝には「2 つの空の木を持つノード」を持つ木 coolTree を作ります。ジッパーを Just に包むのには return を使い、それから >>= を使って goRight 関数に食わせていきます。まず右へ 1 歩踏み出すというのは意味がある操作なので、結果は成功です。2 歩でも大丈夫です。ですがこのとき、注目点は空の部分木になっています。そして右に 3 歩というのは意味

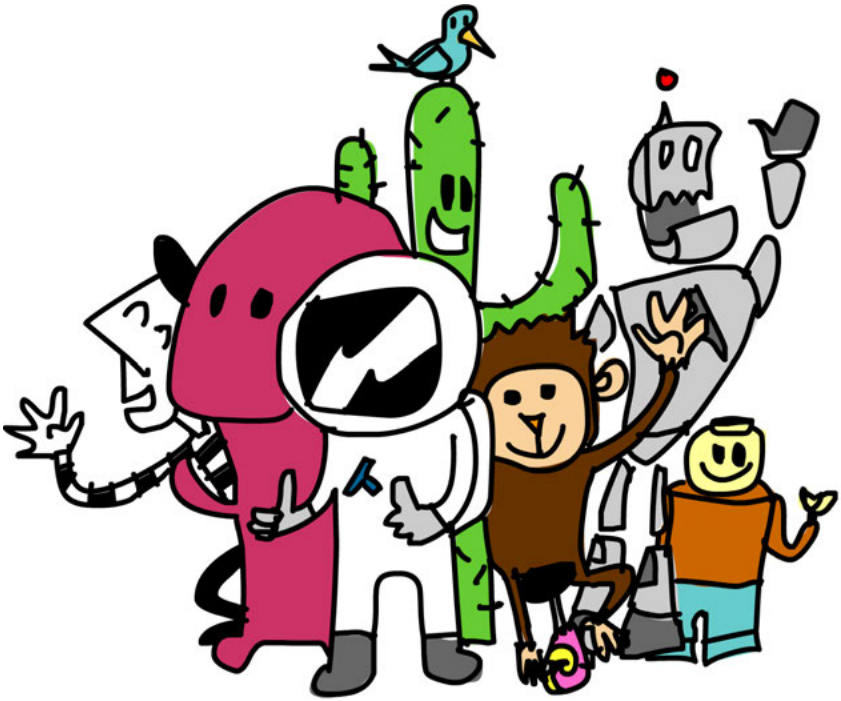
の通らない操作です。空の部分木を右に辿ることはできません。これが結果がNothingになった理由です。

こうして、ピエールの綱の下にあるような安全ネットが僕らの木にも付きました（やったね）。

NOTE さっきのファイルシステムも、存在しないファイルやフォルダに注目しようとするとか、さまざまな要因で失敗する可能性があります。ファイルシステム操作関数を、Maybeモナドを使って行儀よく失敗するように改造するのは、読者への演習問題とします。

15.5　読んでくれてありがとう！

それとも、最後のページを開いてくれてありがとう、かな？ あなたにとって楽しくて役に立つ本だったとしたらうれしいです。この本では、プログラミング言語 Haskell とそのイディオムについて、本質を見抜いた説明ができるよう努力したつもりです。Haskell にはいつも、何か新しい概念が生まれ続けており、勉強すべきことは尽きません。でも、今ならあなたには、かっこいいコードを書き、また他人のコードを読んで理解する力が備わっているはずです。ですから、さっそくコーディングを始めましょう！ そして「向こうの世界」でまた会いましょう。



付録A：マルチバイト文字列処理に関する訳者補足

第9章では、文字列を扱う処理をbytestringで書き直すのは比較的簡単だと言われていますが、とくにマルチバイト文字列を扱うにあたっては、いくつか気をつけなければならないところがあります。ここではそれを訳者による補足としてまとめます。

文字コードとtext

bytestringはその名が示すとおり、バイト列からなる文字列です。一方`String`は、`Char`のリストです。`Char`はバイトよりも広い範囲の文字、具体的にはすべてのユニコード文字を表現できます（4バイトを用いて表現されます）。そのため、日本語のような1文字が1バイトに収まらない文字を含むテキストを扱う場合、そのままbytestringに書き換えると問題が生じます。

1つは文字のエンコードの問題です。多倍長文字列をファイルに書き出す、あるいはハンドルから入出力する際には、バイト列としてエンコードされる必要がありますが、そのためのエンコーディング（Shift_JIS、EUC-JP、あるいはUTF-8など）を正しく扱う必要があります。エンコードされたバイト列をデコードせずにそのまま扱うこともできますが、その場合、多倍長文字列を正しく扱うのは非常に大変です。

`String`とほぼ同じ扱い方ができる高速なライブラリとして、**text**というパッケージがあります（`http://hackage.haskell.org/package/text`）。このパッケージは、効率の良い文字列型`Text`を提供しています。`Text`の内部表現はバイト列ではなく、（現在の実装では）UTF-16でエンコードされた16ビット列になっているので、簡単にユニコード文字列を扱えます。UTF-16でエンコードされているものの、APIとしては（オンザフライでデコードを行い）ユニコード文字を扱うものになっていますので、それを意識する必要はありません。文字列長、サロゲートペアなどは自動的に正しく扱われます。textもbytestringと同じく正格版と遅延版があります。遅延版だとチャンクサイズごとに処理されます。

OverloadedStrings 拡張

もう 1 つの問題は、（ダブルクオートで囲まれた）文字列リテラルが String の値だということです。bytestring/text とともに文字列リテラルを使おうとすると、型を合わせるために、 String を bytestring/text に変換（ Data.ByteString.pack 、 unpack という関数があります）、または逆の変換を行うコードがいたるところに入ることになります。これはとても面倒です。

実は GHC には、文字列リテラルを数値のように多相的な定数として扱う機能があります。既存のコードがコンパイルエラーにならないように、コンパイラオプションで切り替えるようになっています。利用するには ghc コマンドに -XOverloadedStrings を渡すか、

```
{-# LANGUAGE OverloadedStrings #-}
```

という行をコードの先頭に追加します。

```
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Text.Lazy as T
import qualified Data.Text.Lazy.IO as T

main = do
  txt <- T.getContents
  let out = T.unlines . map (T.append "☆") . T.lines $ txt
  T.putStr out
```

これは標準入力からテキストを入力して、各行の先頭に "☆" を追加して出力するプログラムです。"☆" がここでは String ではなく、 Text として扱われています。

パターンマッチに多相的文字列を用いることもできます。

```
wordToInt :: Text -> Int
wordToInt "ひとつ" = 1
wordToInt "ふたつ" = 2
wordToInt "みっつ" = 3
wordToInt _ = 0
```

非 ASCII 文字を含むコードをファイルに保存する場合は、UTF-8 で保存するようにしてください。

ViewPatterns 拡張

bytestring/text と一緒に用いると便利なものに、**ViewPatterns** という GHC 拡張があります。 次のコードは文字列を画面に表示するプログラムです。text の uncons 関数は、与えられた文字列が空でなければ先頭の文字と残りの文字列のペアを Just で包んだものを、空のときは Nothing を返す関数です。

```
putText :: Text -> IO ()
putText txt =
  case T.uncons txt of
    Just (x, xs) -> do
      putChar x
      putText xs
    Nothing ->
      return ()
```

String で同じようなことをするなら、(:) でのパターンマッチを使ってとてもシンプルに書けました。bytestring/text でも、ViewPatterns というものを用いれば似たようなことができるようになります。

```
{-# LANGUAGE ViewPatterns #-}

putText :: Text -> IO ()
putText (T.uncons -> Just (x, xs)) = do
  putChar x
  putStr' xs
putText _ = return ()
```

ViewPatterns はネストして使うこともできます。miran.hs は、あなたの悩みの種を入力すると明るく励ましてくれるプログラムです！

```
{-# LANGUAGE OverloadedStrings, ViewPatterns #-}

import qualified Data.Text as T
import qualified Data.Text.IO as T

main :: IO ()
main = do
  yourWorry <- T.getLine
  T.putStrLn $ encourage yourWorry
  main

isNegative :: Char -> Bool
isNegative = (`elem` "非不未無")

encourage :: T.Text -> T.Text
encourage (T.uncons -> Just ((isNegative -> True), xs))
  = T.append xs "!!"
encourage xs = xs
```

```
$ ghc --make miran
[1 of 1] Compiling Main             ( miran.hs, miran.o )
Linking miran ...
$ ./miran
不景気
景気!!
非正規雇用
正規雇用!!
無気力
気力!!
未完成
完成!!
```

ViewPatterns 拡張について十分な解説を行うための十分な余白がないので、ここで詳しく述べることはやめておきますが、興味のある方はぜひ GHC マニュアルに載っている解説を見てみてください！ 文字列以外にもさまざまなケースで便利に使える機能です。

付録B：訳語一覧

あ

アキュムレータ	accumulator
値コンストラクタ	value constructor
アプリカティブファンクター	applicative functor
インポート	import
写す	map over
エクスポート	export

か

型クラス	type class
型クラス制約	class constraint
型コンストラクタ	type constructor
型シグネチャ	type signature
型シノニム	type synonym
型注釈	type annotation
型引数	type parameter
型変数	type variable
カリー化	currying
関数合成	function composition
関数適用	function application
（再帰の）基底部	base case
具体型	concrete type
（計算）結果	result
結合性宣言	fixity declaration
結合的	associativity
後者	successor
構文糖衣	syntax-sugar
コラッツ列	Collatz sequences

さ

最小完全定義	minimal complete definition
サブクラス	subclass

差分リスト	difference list
サンク	thunk
（型の）種類	kind
真理値	Boolean
シーザー暗号	Caesar cipher
ジェネレータ	generator
（インスタンスの）自動導出	derive instance
（型の）推論	inference
（プログラムについての）推論	reasoning
生成する	yield
セクション	section
走査する	traverse
束縛	bind
疎結合	loosely coupled

た

平らにする	flatten
多相型	parameterized type
多相的	polymorphic
畳み込む	fold
単一要素	singleton
遅延 I/O	lazy I/O
中置関数	infix function
トリプル	triple

な

（集合の）内包的表記	set comprehension
二項演算子	binary operator
二分探索木	binary search tree

は

パンくずリスト	breadcrumbs
非決定性	nondeterministic
ファンクター	functor
副作用	side effect
部分適用された関数	partially applied function

（再帰の）部分問題	subproblem
文脈	context
プロミス	promise
冪集合	powerset
方向リスト	direction list

ま

無名関数	anonymous function
命令型	imperative
持ち上げ	lift

ら

乱数ジェネレータ	random generator
リスト内包表記	list comprehension
レコード構文	record syntax
レンジ	range
連想リスト	associative list

記号・ギリシア文字

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

ア

イ

ウ

エ

カ

セ

ソ

タ

チ

テ

ト

ナ

ニ

ハ

ヒ

フ

ヘ

ホ

マ

ミ

ム

メ

モ

ユ

ヨ

ラ

リ

レ

ロ

ワ

原著者について

Miran Lipovača

Miran Lipovača はスロベニアのリュブリャナで計算機科学を学んでいます。Haskell への情熱に加え、ボクシングとギター、それにお絵描きをたしなみます。踊るガイコツと数字の 71、それに自分の念力で開けたふりをして自動ドアを通るのが大好きです。

訳者について

田中 英行（たなか ひでゆき）

（株）プリファードインフラストラクチャー勤務のプログラマ。プログラミングが好きなあまりプログラミングそのものの科学を志し、その過程にて Haskell に辿り着く。プログラミングで世の中の不可能を可能にしていきたい。Haskell と、ミステリー（特に叙述モノ）、それからもふもふしてニャーと鳴く動物が好きです。

村主 崇行（むらぬし たかゆき）

京都大学白眉センター特定助教。気がつけば Haskell でゲームを作り、Haskell で就職したような人生、関係諸方には頭が上がりません。物理のレベルを上げ、Real World な問題に取り組むことでこの御恩に遍く報いたい。思い入れのある型クラスは `Traversable` で、嫌いな関数は `error` です。

- 本書の内容に関する質問は、オーム社開発部「すごい Haskell たのしく学ぼう！」係宛、E-mail（kaihatu@ohmsha.co.jp）または書状、FAX（03-3293-2825）にてお願いします。お受けできる質問は本書で紹介した内容に限らせていただきます。なお、電話での質問にはお答えできませんので、あらかじめご了承ください。

すごい Haskell たのしく学ぼう！

平成 24 年 5 月 25 日　第 1 版第 1 刷発行
平成 24 年 7 月 10 日　第 1 版第 3 刷発行（v20120619 ）

著　者　Miran Lipovača
訳　者　田中英行・村主崇行
企画編集　オーム社開発局
発行者　竹生修己
発行所　株式会社オーム社
郵便番号　101 – 8460
東京都千代田区神田錦町 3 – 1
電 話　03 – 3233 – 0641
URL　http://www.ohmsha.co.jp/

装丁デザイン　トップスタジオデザイン室（轟木亜紀子）

ISBN 978-4-274-06885-0