

# 実例によるPureScript

---

ウェブのための関数型プログラミング

Phil Freeman, "PureScript by Example - Functional Programming for the Web"

[＜ 目次](#)

## 1 はじめに

---

### 1.1 関数型JavaScript

---

関数型プログラミングの手法は、かねてよりJavaScriptでも用いられてきました。

- [UnderscoreJS](#)などのライブラリは、`map` や `filter`、`reduce` といったよく知られた関数を活用して、小さいプログラムを組み合わせて大きなプログラムを作れるようにします。

```
var sumOfPrimes =  
  _.chain(_.range(1000))  
    .filter(isPrime)  
    .reduce(function(x, y) {  
      return x + y;  
    })  
    .value();
```

- NodeJSにおける非同期プログラミングでは、コールバックを定義するために第一級の値としての関数に大きく依存しています。

```
require('fs').readFile(sourceFile, function (error, data) {  
  if (!error) {  
    require('fs').writeFile(destFile, data, function (error) {
```

```
        if (!error) {
            console.log("File copied");
        }
    });
}
});
```

関数は単純な抽象化を可能にし、優れた生産性をもたらしてくれます。しかし、JavaScriptでの関数型プログラミングには欠点があります。JavaScriptは冗長で、型付けされず、強力な抽象化を欠いているのです。また、奔放に書かれたJavaScriptコードは、式の理解をとて難しくします。

PureScriptはこのような問題を解決すべく作られたプログラミング言語です。PureScriptは、とても表現力豊かでありながらわかりやすく読みやすいコードを書けるようにする、軽量の構文を備えています。強力な抽象化を提供する豊かな型システムも使用しています。また、JavaScriptやJavaScriptへとコンパイルされる他の言語と相互運用するときに重要な、高速で理解しやすいコードを生成します。一言で言えば、PureScriptは純粋関数型プログラミングの理論的な強力さと、JavaScriptのお手軽で緩いプログラミングスタイルとの、とても現実的なバランスを狙っているということを理解して頂けたらと思います。

## 1.2 型と型推論

動的型付けの言語と静的型付けの言語をめぐる議論についてはよく知られています。PureScriptは静的型付けの言語、つまり正しいプログラムはコンパイラによってその動作を示すような型を与えられる言語です。逆にいえば、型を与えることができないプログラムは誤ったプログラムであり、コンパイラによって拒否されます。動的型付けの言語とは異なり、PureScriptでは型はコンパイル時のみに存在し、実行時には型の表現はありません。

PureScriptの型は、これまでJavaやC#のような他の言語で見たような型とは、いろいろな意味で異なっていることに注意することが大切です。おおまかに言えばPureScriptの型はJavaやC#と同じ目的を持っているものの、PureScriptの型はMLとHaskellのような言語に影響を受けています。開発者がプログラムについての強い主張を表明できるので、PureScriptの型は表現力豊かなのです。最も重要なのは、PureScriptの型システムは型推論(type inference)をサポートしていることです。型推論は最低限の明示的な型注釈だけを必要とし、型システムを厄介者ではなく道具にしてくれます。簡単な例を示すと、次のコードは数を定義していますが、それが `Number` 型だという注釈はコードのどこにもありません。

```
iAmANumber =
    let square x = x * x
    in square 42
```

次のもっと複雑な例では、コンパイラにとって未知の型が存在しているときでさえも、型注釈なしで型の正しさを確かめることができるということが示されています。

```
iterate f 0 x = x
iterate f n x = iterate f (n - 1) (f x)
```

ここで `x` の型は不明ですが、`x` がどんな型を持っているかにかかわらず、`iterate` が型システムの規則に従っていることをコンパイラは検証することができます。

静的型はプログラムの正しさについての確信を得るためだけではなく、その正しさによって開発を助けるということをあなたに納得させる(もしくは、あなたの理解を確認する)ことをこの本では試みます。最も単純な抽象化を使わないかぎりJavaScriptでコードの大規模なリファクタリングすることは難しいですが、型検証器のある表現力豊かな型システムは、リファクタリングさえ楽しく対話的な体験にします。

加えて、型システムによって提供されたこのセーフティネットは、より高度な抽象化をも可能にします。実際に、関数型プログラミング言語Haskellによって知られるようになった、型主導の強力な抽象化である型クラスをPureScriptは備えています。

## 1.3 多言語Webプログラミング

関数型プログラミングはすでに多くの成功を収めています。特に成功している応用例をいくつか挙げれば、データ解析、構文解析、コンパイラの実装、ジェネリックプログラミング、並列処理などです。

PureScriptのような関数型言語はアプリケーション開発の最初から最後までを実施することが可能です。値や関数の型を提供することで既存のJavaScriptコードをインポートし、通常のPureScriptコードからこれらの関数を使用する機能をPureScriptは提供しています。この手法については本書で後ほど見ていくことになります。

しかしながら、PureScriptの強みのひとつはJavaScriptを対象とする他の言語との相互運用性にあります。アプリケーションの開発の一部にだけPureScriptを使用し、JavaScriptの残りの部分を記述するのに他の言語を使用するという方針もあります。

いくつかの例を示します。

- 中核となる処理はPureScriptで記述し、ユーザーインターフェースはJavaScriptで記述する
- JavaScriptや、他のJavaScriptにコンパイルされる言語でアプリケーションを書き、PureScriptでそのテストを書く

- 既存のアプリケーションのユーザインタフェースのテストを自動化するためにPureScriptを使用する

この本では、小規模な課題をPureScriptで解決することに焦点を当てます。この解決策は大規模なアプリケーションに統合することもできますが、JavaScriptからPureScriptコードを呼び出す方法、およびその逆についても見ていきます。

## 1.4 ソフトウェア要件

---

この本でのソフトウェア要件は最小限です。第1章では開発環境のセットアップを一から案内します。これから使用するツールは、ほとんどの現代のオペレーティングシステムの標準リポジトリで利用できるものです。

PureScriptコンパイラ自体は最新のHaskell Platformが稼働しているシステムならソースからビルドすることができますが、次の章ではこの手順を説明していきます。

## 1.5 読者について

---

読者はJavaScriptの基本をすでに理解しているものと仮定します。すでにNPM、Grunt、BowerのようなJavaScriptのエコシステムでの経験があれば、自身の好みに応じて標準設定をカスタマイズしたい場合などに役に立ちますが、そのような知識は必要ではありません。

関数型プログラミングの予備知識は必要ありませんが、あっても害にはならないでしょう。実例には新しいアイデアが付きものですから、これから使う関数型プログラミングからこうした概念に対する直感的理解を形成することができるはずです。

PureScriptはプログラミング言語Haskellに強く影響を受けているため、Haskellに通じている読者はこの本の中で提示された概念や構文の多くに見覚えがあるでしょう。しかしながら、読者はPureScriptとHaskellの間にはいくつか重要な違いがあることも理解しておかなければなりません。ここで紹介する概念の多くはHaskellでも同じように解釈できるとはいえ、どちらかの言語での考え方を他方の言語でそのまま応用しようとすることは、必ずしも適切ではありません。

## 1.6 本書の読み進めかた

---

本書の各章は、概ね章ごとに完結しています。しかしながら、多少の関数型プログラミングの経験がある初心者でも、まずは各章を順番に進めていくことをおすすめします。最初の数章では、本書の後半の内容を理解するために必要な基礎知識を養います。関数型プロ

プログラミングの考え方に十分通じた読者(特にMLやHaskellのよう強く型付けされた言語での経験を持つ読者)なら、本書の前半のほうの章を読まなくても後半の章のコードの大まかな理解を得ることがおそらく可能でしょう。

各章ではそれぞれひとつの実用的な例に焦点をあて、新しい考え方を導入するための動機付けとして用います。各章のコードは本書の[GitHubのリポジトリ](#)から入手できます。各章にはソースコードから抜粋したコード片が掲載されていますが、完全に理解するためには本書に掲載されたコードと平行してリポジトリのソースコードを読む必要があります。対話式環境 `psci` で実行し理解を確かめられるように、長めの節には短いコード片が含まれます。

コード例は次のように等幅フォントで示されています。

```
module Example where

main = Debug.Trace.trace "Hello, World!"
```

先頭にドル記号がついた行は、コマンドラインに入力されたコマンドです。

```
$ psc src/Main.purs
```

通常はこれらのコマンドはLinux / Mac OSの利用者には適合しますが、Windowsの利用者はファイル区切り文字を変更する、シェルの組み込み機能をWindowsの相当するものに置き換えるなどの小さな変更を加える必要があるかもしれません。

`psci` 対話式プロンプトに入力するコマンドは、行の先頭に山括弧が付けられています。

```
> 1 + 2
  3
```

各章には演習が付いており、それぞれ難易度も示されています。各章の内容を完全に理解するために、演習に取り組むことを強くお勧めします。

この本は初心者にはPureScriptへの導入を提供することを目的としており、問題についてのお決まりの解決策の一覧を提供するような種類の本ではありません。初心者にとってこの本を読むのは楽しい挑戦になるはずですし、本書の内容を読み演習に挑戦すればだいたいの利益を得られるでしょうが、なにより重要なのは、あなたが自分自身のコードを書いてみることです。

## 1.7 困ったときには

---

もしどこかでつまづいたときには、PureScriptを学ぶためのオンラインで利用可能な資料がたくさんあります。

- PureScript IRCチャンネルはあなたが抱える問題についてチャットするのに最適な場所です。IRCクライアントでirc.freenode.netをポイントし、#purescriptチャンネルに接続してください。
- [PureScriptのウェブサイト](#)にはPureScriptの開発者によって書かれたブログ記事や、初心者向けの動画、その他のリソースへのリンクがあります。
- [PureScriptコンパイラのドキュメント](#)は、言語の主要な機能についての簡単なコード例があります。
- [Try PureScript!](#) ではユーザーがWebブラウザでPureScriptコードをコンパイルすることができます。また、ウェブサイトにはコードの簡単な例がいくつか含まれています。

もしあなたが例を読んで学ぶことを好むなら、GitHubの `purescript` と `purescript-contrib` organisation にはPureScriptコードの例がたくさんあります。

## 1.8 著者について

---

私はPureScriptコンパイラの最初の開発者です。私はカリフォルニア州ロサンゼルスを拠点にしており、8ビットパーソナルコンピュータ、Amstrad CPC上のBASICでまだ幼い時にプログラミングを始めました。それ以来、私はいくつものプログラミング言語(JavaやScala、C#、F#)で業務に携わってきました。

プロとしての経歴が始まって間もなく、私は関数型プログラミングと数学の関係を理解するようになり、そしてプログラミング言語Haskellとの恋に落ちました。

JavaScriptでの経験をもとに、私はPureScriptコンパイラの開発を始めることにしました。私がHaskellのような言語から取り上げた関数型プログラミングの手法を使っていることを私自ら発見しましたが、それを応用するためのもっと理にかなった環境を求めています。そのときの解決策にはHaskellをその意味論を維持しながらJavaScriptへとコンパイルするいろいろな試み(Fay、Haste、GHCJS)が含まれていましたが、私が興味を持っていたのは、この問題への別の切り口からのアプローチ、すなわちHaskellのような言語の構文と型システムを楽しみながらJavaScriptの意味論も維持するということが、どのようにすれば可能になるのかでした。

私は[ウェブサイト](#)を運営しており、[Twitter](#)で連絡をとることもできます。

## 1.9 謝辞

---

現在の状態に到達するまでPureScriptを手伝ってくれた多くの協力者に感謝したいと思います。コンパイラやツール、ライブラリ、ドキュメント、テストでの組織的で弛まぬ努力がなかったら、プロジェクトは間違いなく失敗していたことでしょう。

この本の表紙に表示されたPureScriptのロゴはGareth Hughesによって作成されたもので、[Creative Commons Attribution 4.0 license](#)の条件の下で再利用させて頂いています。

最後に、この本の内容に関する反応や訂正をくださったすべての方に、心より感謝したいと思います。

## 2 開発環境の準備

---

### 2.1 この章の目標

---

この章の目標は、作業用のPureScript開発環境を準備し、最初のPureScriptプログラムを書くことです。

これから書く最初のコードは、NPMとBowerから依存するライブラリを使用し、ビルド自動化ツールであるGruntを使用してビルドされるライブラリの例です。このライブラリは直角三角形の対角線の長さを計算する関数ひとつだけを提供します。

### 2.2 導入

---

PureScript開発環境を準備するために、次のツールを使います。

- `psc` - PureScriptコンパイラ本体
- `npm` - 残りの開発ツールをインストールできるようにする、Nodeパッケージマネージャ
- `bower` - 必要となる様々なバージョンのPureScriptパッケージで使われているパッケージマネージャ
- `grunt` - PureScriptコードをビルドするために使用する自動化ツール

この章ではこれらのツールのインストール方法と設定を説明します。

### 2.3 PureScriptのインストール

---



PureScriptコンパイラをインストールするときに推奨される方法は、ソースからコンパイラをビルドすることです。PureScriptコンパイラは[PureScriptのウェブサイト](#)から64ビットのUbuntu用のバイナリディストリビューションとしてダウンロードすることもできますが、現在のところバイナリディストリビューションは主要なリリースについてだけ提供されています。もし最近のバグ修正や機能追加がなされた最新版に保ち、コンパイラで最新のパッケージをビルドできるようにしたいなら、最新のマイナーリリースをビルドするよう以下の指示に従ってください。

主なソフトウェア要件としては、[Haskell Platform](#)がインストールされていることです。お使いのオペレーティングシステムによっては、パッケージマネージャを使用して `ncurses` 開発パッケージもインストールする必要があるかもしれません(例えば、Ubuntuでは `libncurses5-dev` パッケージとしての利用できます)。

Cabal実行ファイルの最新版を持っているのを確認することから始めましょう。

```
$ cabal install Cabal cabal-install
```

また、Cabalのパッケージ一覧が最新であることも確認してください。

```
$ cabal update
```

PureScriptコンパイラは、グローバルもしくはローカルディレクトリ内のCabalサンドボックス内のどちらかにインストールすることができます。この節ではグローバルにPureScriptをインストールし、その実行ファイルがパス上で利用できるようにする方法を説明します。

`cabal install` コマンドを使用して、HackageからPureScriptをインストールします。

```
$ cabal install purescript
```

これでコンパイラおよび関連する実行ファイルはあなたのパス上で利用できるようになります。確認のために、コマンドラインでPureScriptコンパイラを実行してみましょう：

```
$ psc
```

## 2.4 各ツールのインストール

もし[NodeJS](#)がインストールされていないなら、NodeJSをインストールする必要があります。そうするとシステムに `npm` パッケージマネージャもインストールされるはずです。 `npm` がイン



ストールされ、パス上で利用可能であることを確認してください。

`npm` がインストールされたら、GruntとBowerもインストールする必要があります。プロジェクトがどこで作業しているかにかかわらずこれらのコマンドラインツールを利用可能にするため、通常はグローバルにインストールしておくのがいいでしょう。

```
$ npm install -g grunt-cli bower
```

これで、最初のPureScriptプロジェクトを作成するために必要なすべてのツールの用意ができたことになります。

## 2.5 Hello, PureScript!

まずはシンプルに始めましょう。PureScriptコンパイラ `psc` を直接使用して、基本的なHello World! プログラムをコンパイルします。3つの標準のコマンドですべての依存関係ライブラリを含めてゼロからアプリをビルドできるようになるまで、この章を読み進むにつれて開発手順をだんだんと自動化していきます。

まず最初に、ソースファイルのディレクトリ `src` を作成し、`src/Chapter2.purs` という名前のファイルに以下のコードを貼り付けます。

```
module Chapter2 where

import Debug.Trace

main = trace "Hello, World!"
```

これは小さなサンプルコードですが、いくつかの重要な概念を示しています。

- すべてのソースファイルはモジュールヘッダから始まります。モジュール名は、ドットで区切られた大文字で始まる1つ以上の単語から構成されています。ここではモジュール名としてひとつの単語だけが使用されていますが、`My.First.Module` というようなモジュール名も有効です。
- モジュールは、モジュール名の各部分を区切るためのドットを含めた、完全な名前を使用してインポートされます。ここでは `trace` 関数を提供する `Debug.Trace` モジュールをインポートしています。
- この `main` プログラムの定義本体は、関数適用の式になっています。PureScriptでは、関数適用は関数名の後に引数を空白で区切って書くことで示されます。

それではこのコードをビルドして実行してみましょう。次のコマンドを実行します。

```
$ psc src/Chapter2.purs
```

うまくいくと、大量のJavaScriptがコンソールに出力されるのを目にするはずです。コンソールに出力する代わりに、`--output` コマンドラインオプションで出力をファイルにリダイレクトしてみましょう。

```
$ psc src/Chapter2.purs --output dist/Main.js
```

これでNodeJSを使用してコードを実行することができるはずです。

```
$ node dist/Main.js
```

うまくいくと、NodeJSはこのコードを正常に実行し、コンソールには何も出力されないはずです。これは、メインとなるモジュールの名前をPureScriptコンパイラに教えていないためです！

```
$ psc src/Chapter2.purs --output dist/Main.js --main=Chapter2
```

再びNodeJSで実行すると、今度は“Hello, World!”という単語がコンソールに出力されるのがわかるはずです。

## 2.6 使用されていないコードを取り除く

テキストエディタで `dist/Main.js` ファイルを開くと、大量のJavaScriptコードが書かれているのがわかります。これはコンパイラがPreludeと呼ばれるモジュール群で定義されている標準関数を追加しているためです。Preludeにはコンソールに出力するのに使う

`Debug.Trace` モジュールが含まれています。

ここで生成されたコードのほとんどは実際には使用されていないので、別のコンパイラオプションを指定すると未使用のコードを削除することができます。

```
$ psc src/Chapter2.purs --output dist/Main.js --main=Chapter2 --module Chapter2
```

`Chapter2` モジュールで定義されたコードで必要とされているJavaScriptだけを含めるよう `psc` に指示する `--module Chapter2` オプションを追加しました。生成されたコードをテキストエディタで開くと、次のように出力されているのがわかるはずです。

```

var PS = PS || {};
PS.Debug_Trace = (function () {
    "use strict";
    function trace(s) {
        return function() {
            console.log(s);
            return {};
        };
    };
    return {
        trace: trace
    };
})();

var PS = PS || {};
PS.Chapter2 = (function () {
    "use strict";
    var Debug_Trace = PS.Debug_Trace;
    var main = Debug_Trace.trace("Hello, World!");
    return {
        main: main
    };
})();

PS.Chapter2.main();

```

NodeJSを使用してこのコードを実行すると、先ほどと同じ文字列がコンソールに出力されるはずです。

ここでPureScriptコンパイラがJavascriptコードを生成する方法の要点が示されています。

- すべてのモジュールはオブジェクトに変換され、そのオブジェクトにはそのモジュールのエクスポートされたメンバが含まれています。モジュールは即時関数パターンによってスコープが限定されたコードで初期化されています。
- PureScriptは可能な限り変数の名前をそのまま使おうとします
- PureScriptにおける関数適用は、そのままJavaScriptの関数適用に変換されます。
- 引数のない単純な呼び出しとしてメインメソッド呼び出しが生成され、すべてのモジュールが定義された後に実行されます。
- PureScriptコードはどんな実行時ライブラリにも依存しません。コンパイラによって生成されるすべてのコードは、あなたのコードが依存するいずれかのPureScriptモジュールをもとに出力されているものです。

PureScriptはシンプルで理解しやすいコードを生成すること重視しているので、これらの点は大切です。実際に、ほとんどのコード生成処理はごく軽い変換です。PureScriptについて

の理解が比較的浅くても、ある入力からどのようなJavaScriptコードが生成されるかを予測することは難しくありません。

## 2.7 Gruntによるビルドの自動化

今度は、PureScriptコンパイラオプションを毎回手で入力する代わりに、コードを自動でビルドできるように、Gruntを設定してみましょう。

プロジェクトディレクトリに `Gruntfile.js` という名前のファイルを作成し、次のコードを貼り付けてください。

```
module.exports = function(grunt) {

    "use strict";

    grunt.initConfig({

        srcFiles: ["src/**/*.purs"],

        psc: {
            options: {
                main: "Chapter2",
                modules: ["Chapter2"]
            },
            all: {
                src: ["<%=srcFiles%>"],
                dest: "dist/Main.js"
            }
        }
    });

    grunt.loadNpmTasks("grunt-purescript");

    grunt.registerTask("default", ["psc:all"]);
};
```

このファイルではNodeモジュールを定義しており、ビルド構成を定義するために `grunt` モジュールをライブラリとして使用しています。JSONプロパティとしてコマンドラインオプションを指定してPureScriptコンパイラを呼び出せる `grunt-purescript` プラグインを使用しています。

`grunt-purescript` プラグインは他にも便利な機能を提供しており、コードから自動的に Markdownドキュメントを生成する機能や、ライブラリから `psci` 対話式コンパイラ向けの設定

ファイルを自動生成する機能があります。興味があれば `grunt-purescript` [プロジェクトのホームページ](#)を参照してみてください。

次のように入力して、ローカルのmodulesディレクトリに `grunt` ライブラリと `grunt-purescript` プラグインをインストールしてください。

```
$ npm install grunt grunt-purescript@0.6.0
```

保存された `Gruntfile.js` ファイルを使うと、次のようにコードをコンパイルできるようになります。

```
$ grunt
>> Created file dist/Main.js.

Done, without errors.
```

## 2.8 NPMパッケージの作成

Gruntを設定したので、コンパイルするときに毎回コマンドラインにコマンドを入力する必要はなくなりましたが、もっと重要なのは、アプリケーションのエンドユーザはどちらも必要ないということです。そのためには、ビルドする前にNPMパッケージの必要なモジュールを自動的にインストールしておくという手順を追加しておきましょう。

依存関係が指定された独自のNPMパッケージを定義します。

プロジェクトディレクトリで `init` サブコマンドを指定して `npm` を実行し、新しいプロジェクトを初期化します。

```
$ npm init
```

いろいろと質問されますが、それが終わると `package.json` という名前のファイルがプロジェクトディレクトリに追加されます。このファイルではプロジェクトのプロパティを指定したり、依存するライブラリの指定を追加することができます。テキストエディタでこのファイルを開き、JSONオブジェクトに次のプロパティを追加しましょう。

```
"dependencies": {
  "grunt-purescript": "0.6.0"
}
```

このコードではインストールする `grunt-purescript` プラグインの厳密なバージョンを指定しています。

依存するライブラリを手作業でインストールするかわりに、エンドユーザーは単に `npm` コマンドを使用するだけで必要なものすべてをインストールできるようになりました。

```
$ npm install
```

## 2.9 Bowerによる依存関係の追跡

この章の目的となっている `diagonal` 関数を書くためには、平方根を計算できるようにする必要があります。 `purescript-math` パッケージにはJavaScriptの `Math` オブジェクトのプロパティとして定義されている関数の型定義が含まれていますので、 `purescript-math` パッケージをインストールしてみましょう。 `npm` の依存関係でやったのと同じように、次のようにコマンドラインに入力すると直接このパッケージをダウンロードできます。

```
$ bower install purescript-math#0.1.0
```

このコマンドは `purescript-math` ライブラリのバージョン0.1.0をそれが依存するライブラリと一緒にインストールします。

しかし、 `package.json` を作成してNPMの依存関係を制御するために `npm init` を使用したのと同じような方法で、Bowerの依存関係が含まれている `bower.json` ファイルを設定することができます。

コマンドラインに次のコマンドを入力します。

```
$ bower init
```

NPMの場合とちょうど同じように、いくつか質問をされ、それが終わると `bower.json` ファイルがプロジェクトディレクトリに配置されます。この処理の途中で、すでに存在するライブラリの依存関係をプロジェクトファイルに含めたいかどうかを尋ねられるでしょう。「はい」を選択した場合は、 `bower.json` にこのようなセクションがあるのがわかるでしょう。

```
"dependencies": {  
  "purescript-math": "0.1.0"  
}
```

エンドユーザーが手作業で依存するライブラリを指示する必要がなくなり、代わりに次のようにコマンドを呼び出すだけで依存するライブラリを取り込むことができるようになりました。

```
$ bower update
```

それでは、Bowerから取り込んだ依存先ライブラリをコンパイルに含めるように、Gruntスクリプトを更新してみましょう。 `Gruntfile.js` を編集し、ソースファイルについての行を次のように変更します。

```
srcFiles: ["src/**/*.purs", "bower_components/**/*.purs"]
```

この行では `bower_components` ディレクトリのソースファイルをコンパイルするソースファイルに含めています。独自のBower構成がある場合は、それに応じてこの行を修正する必要があるかもしれません。

## なぜNPMとBowerの両方を使うのか？

疑問に思ったかもしれませんが、なぜ2つのパッケージマネージャを使い分ける必要があるのでしょうか？ PureScriptライブラリをNPMレジストリに含めることはできないのでしょうか？

PureScriptコミュニティは、さまざまな理由でPureScriptの依存ライブラリをBowerを使用して標準化しています。

- PureScriptのライブラリパッケージがJavaScriptのソースコードを含むことはめったになく、コンパイルされないままNPMレジストリへ配置するのには適していません。
- Bowerレジストリは、直接コードをホスティングする代わりに、既存のGitリポジトリのパッケージ名とバージョンの対応関係だけを管理しています。これによりコミュニティがコードおよびリリースを管理するのにGitHubのような既存のツールを使用することができます。
- BowerはCommonJSのモジュール標準のような特定の配置に従うようパッケージに要求していません。

もちろん、任意のパッケージマネージャを自由に選択して使用することもできます。PureScriptコンパイラおよびツール群は、Bowerに（またはNPM、Gruntなどにも）依存しているわけではありません。

## 2.10 対角線の長さの計算



それでは外部ライブラリの関数を使用する例として `diagonal` 関数を書いてみましょう。

まず、`src/Chapter2.purs` ファイルの先頭に次の行を追加し、`Math` モジュールをインポートします。

```
import Math
```

そして、次のように `diagonal` 関数を定義します。

```
diagonal w h = sqrt (w * w + h * h)
```

この関数の型を定義する必要はないことに注意してください。`diagonal` は2つの数値を取り数を返す関数であるとコンパイラは推論することができます。しかし、ドキュメントとしても役立つので、通常は型注釈を提供しておくことをお勧めします。

それでは、新しい `diagonal` 関数を使うように `main` 関数も変更してみましょう。

```
main = print (diagonal 3 4)
```

Gruntを使用して、モジュールを再コンパイルします。

```
$ grunt
```

生成されたコードを再び実行すると、このコードが正常に呼び出されたことがわかるでしょう。

```
$ node dist/Main.js
```

5

## 2.11 対話式処理系を使用したコードのテスト

PureScriptコンパイラには `psci` と呼ばれる対話式のREPL(Read-eval-print loop)が付属しています。`psci` はコードをテストしたり思いついたことを試すのにとても便利です。それでは、`psci` を使って `diagonal` 関数をテストしてみましょう。

`grunt-purescript` プラグインは、ソースファイルに応じて `psci` 設定を自動で生成するように設定することができます。これにより `psci` に手作業でモジュールを読み込む手間を省くことができます。

これを設定するには、`Gruntfile.js` ファイルに以下のような `psc` や `pscMake` という新しいビルドターゲットを追加します。

```
dotPsci: ["<%=srcFiles%>"]
```

また、デフォルトのタスクにこのターゲットを追加しておきましょう。

```
grunt.registerTask("default", ["psc:all", "dotPsci"]);
```

これで `grunt` を実行すると `.psci` ファイルがプロジェクトディレクトリに自動生成されるようになります。このファイルは、`psci` の起動時に設定で使用するコマンドを指定するのに使われます。

それでは `psci` を起動してみます。

```
$ psci
>
```

コマンドの一覧を見るには、`:?` と入力します。

```
> :?
The following commands are available:

:~ Show this help menu
:i <module> Import <module> for use in PSCI
:m <file> Load <file> for importing
:q Quit PSCI
:r Reset
:t <expr> Show the type of <expr>
```

Tabキーを押すと、自分のコードで利用可能なすべての関数、及びBowerの依存関係とプレリユードモジュールのリストをすべて見ることは可能です。

幾つか数式を評価してみてください。 `psci` で評価を行うには、1行以上の式を入力し、Ctrl+ Dで入力を終了します。

```
> 1 + 2
3

> "Hello, " ++ "World!"
"Hello, World!"
```

それでは `psci` で `diagonal` 関数を試してみましょう。

```
> Chapter2.diagonal 5 12

13
```

また、`psci` で関数を定義する使こともできます。

```
> let double x = x * 2

> double 10
20
```

コード例の構文がまだよくわからなくても心配はいりません。この本を読み進めるうちにわかるようになっていきます。

最後に、`:t` コマンドを使うと式の型を確認することができます。

```
> :t true
Prim.Boolean

> :t [1, 2, 3]
[Prim.Number]
```

`psci` で試してみてください。もしどこかでつまづいた場合は、メモリ内にあるコンパイル済みのすべてのモジュールをアンロードするリセットコマンド `:r` を使用してみてください。

## 2.12 任意：CommonJSのモジュールのビルド

PureScriptコンパイラの `psc` コマンドは、ウェブブラウザでの使用に適した、単一の出力ファイルにJavaScriptコードを生成します。それとは別に、コンパイルには `psc-make` という選択肢もあります。 `psc-make` では、コンパイルされるPureScriptモジュールそれぞれについて、

個別のCommonJSモジュール生成することができます。もしCommonJSモジュール標準に対応したNodeJSのような実行環境を対象とするなら、`psc-make` のほうが望ましい場合があるでしょう。

コマンドラインで `psc-make` を実行するには、入力ファイルを指定し、`--output` オプションでCommonJSモジュールが作成されるディレクトリも指定します。

```
$ psc-make src/Chapter2.purs --output dist/
```

与えられた入力ファイルのそれぞれのモジュールについて、`dist/` ディレクトリの中にサブディレクトリが作成されるでしょう。Bowerの依存関係を使用している場合は、`bower_components/` ディレクトリ内のソースファイルを含めることを忘れないでください！

`grunt-purescript` プラグインは `psc-make` を使用したコンパイルにも対応しています。Gruntから `psc-make` を使用するには、`Gruntfile.js` ファイルを次のように変更します。

- ビルドターゲットを `psc` から `pscMake` へ変更します
- 出力先を `dist/Main.js` という単一のファイルからディレクトリ `dest: "dist /"` へ変更します
- デフォルトのタスクを `pscMake` ビルドターゲットを参照するように変更します

これで、`Chapter2` モジュールとその依存先ライブラリそれぞれについて、`grunt` コマンドラインツールが `dist/` の下にサブディレクトリを作成するようになりました。

## 2.13 Gruntプロジェクトテンプレートの使用

様々なビルドプロセスに対応するために、NPMやGrunt、Bowerはいろいろな方法でカスタマイズすることができます。しかし、簡単なプロジェクトでは、この手順はGruntプロジェクトテンプレートを使用して自動化することもできます。

`grunt-init` ツールは、テンプレートを利用して簡単なプロジェクトを開始する方法を提供します。`grunt-init-purescript` プロジェクトは、簡単なテストスイートを含むPureScriptプロジェクトのシンプルなテンプレートを提供します。

`grunt-init` を使用してプロジェクトを設定するには、最初にNPMを使用して `grunt-init` のコマンドラインツールをインストールします。

```
$ npm install -g grunt-init
```

そして、ホームディレクトリにPureScriptテンプレートを複製してください。たとえば、LinuxやMacでは、次のようにします。

```
$ mkdir ~/.grunt-init
$ git clone https://github.com/purescript-contrib/grunt-init-purescript.git
~/.grunt-init/purescript
```

これで、新しいディレクトリに簡単なプロジェクトを作成できるようになりました。

```
$ mkdir new-project
$ cd new-project/
$ grunt-init purescript
```

いくつかの簡単な質問を受けたあと、現在のディレクトリにプロジェクトが初期化されますので、これまで見てきたコマンドを使ってビルドの準備をしましょう。

```
$ npm install
$ bower update
$ grunt
```

最後のコマンドでは、ソースファイルをビルドし、テストスイートを実行しています。

より複雑なプロジェクトの雛形として、このプロジェクトテンプレートを使用することができます。

## 演習

1. (簡単) `Math.pi` 定数を使用し、指定された半径の円の面積を計算する関数 `circleArea` を書いてみましょう。また、`psci` を使用してその関数をテストしてください。
2. (やや難しい) `node dist/Main.js` と入力する代わりにユーザが単に `grunt run` を入力するだけで、コンパイルされたコードをNodeJSで実行できるように、`Gruntfile.js` ファイルにタスクを追加しましょう。ヒント: `grunt-execute` Gruntプラグインを使用することを検討してください。

## 2.14 まとめ

この章では、JavaScriptのエコシステムのNPMとBower、Gruntという標準的なツールを使用し、一から開発環境をセットアップしました。

また最初のPureScript関数を書き、コンパイルし、NodeJSを使用して実行することができました。

以降の章では、コードをコンパイルやデバッグ、テストするためにこの開発設定を使用しますので、これらのツールや使用手順に十分習熟しておくといよいでしょう。

## 3 関数とレコード

---

### 3.1 この章の目標

---

この章では、関数およびレコードというPureScriptプログラムのふたつの構成要素を導入します。さらに、どのようにPureScriptプログラムを構造化するのか、どのように型をプログラム開発に役立てるかを見ていきます。

電話番号の一覧を管理する簡単な電話帳アプリケーションを作成していきます。このコード例により、PureScriptの構文からいくつかの新しい概念を導入します。

このアプリケーションのフロントエンドは対話式処理系 `psci` を使うようにしますが、JavaScriptでフロントエンドを書くこともできるでしょう。

### 3.2 プロジェクトの準備

---

この章のソースコードは `src/data/ PhoneBook.purs` というファイルに含まれています。このファイルは次のようなモジュール宣言とインポート一覧から始まります。

```
module Data.PhoneBook where

import Data.List
import Data.Maybe

import Control.Plus (empty)
```

ここでは `purescript-lists` パッケージで提供されている `Data.List` モジュールをインポートしています。`purescript-lists` パッケージはbowerを使用してインストールすることができ、連結リストを使うために必要ないくつかの関数が含まれています。

`Data.Maybe` モジュールは、値が存在したりしなかったりするような、オプションな値を扱うためのデータ型と関数を定義しています。

`Control.Plus` モジュールには後ほど使う `empty` 値が定義されています。このモジュールのインポート内容は括弧内で明示的に列挙されていることに注意してください。明示的な列挙はインポート内容の衝突を避けるのに役に立つので、一般に良い習慣です。

## 3.3 単純な型

JavaScriptのプリミティブ型に対応する組み込みデータ型として、PureScriptでは数値型と文字列型、真偽型の3つが定義されています。すべてのモジュールに暗黙にインポートされる `Prim` モジュールでこれらの型は定義されています。これらの型はそれぞれ `Number`、`String`、と `Boolean` と呼ばれ、`psci` の `:t` コマンドを使用すると簡単な値の型を確認できます。

```
$ psci

> :t 1
Prim.Number

> :t "test"
Prim.String

> :t true
Prim.Boolean
```

PureScriptには他にも配列とレコード、関数の3つの組み込み型が定義されています。

配列はJavaScriptの配列に対応していますが、JavaScriptの配列とは異なり、PureScriptの配列のすべての要素は同じ型を持つ必要があります。

```
> :t [1, 2, 3]
[Prim.Number]

> :t [true, false]
[Prim.Boolean]

> :t [1, false]
Cannot unify Prim.Number with Prim.Boolean.
```

最後の例で起きているエラーは型検証器によって報告されたもので、配列の2つの要素の



型を単一化(Unification)しようとして失敗したことを示しています。

レコードはJavaScriptのオブジェクトに対応しており、レコードリテラルはJavaScriptのオブジェクトリテラルと同じ構文になっています。

```
> let author =  
    { name: "Phil"  
      , interests: ["Functional Programming", "JavaScript"]  
    }  
  
> :t author  
{ name :: Prim.String, interests :: [Prim.String] }
```

この型が示しているのは、オブジェクト `author` は、

- `String` 型のフィールド `name`
- `[String]` つまり `String` の配列の型のフィールド `interests`

というふたつのフィールド(field)を持っているということです。

レコードのフィールドは、ドットに続けて参照したいフィールドのラベルを書くと参照することができます。

```
> author.name  
"Phil"  
  
> author.interests  
["Functional Programming", "JavaScript"]
```

PureScriptの関数はJavaScriptの関数に対応しています。PureScriptの標準ライブラリは多くの関数の例を提供しており、この章ではそれらをもう少し詳しく見ていきます。

```
> :t Prelude.flip  
forall a b c. (a -> b -> c) -> b -> a -> c  
  
> :t Prelude.const  
forall a b. a -> b -> a
```

ファイルのトップレベルでは、等号の直前に引数を指定することで関数を定義することができます。

```
add :: Number -> Number -> Number
```

```
add x y = x + y
```

バックスラッシュに続けて空白文字で区切られた引数名のリストを書くと、関数をインラインで定義することもできます。

```
> let
    add :: Number -> Number -> Number
    add = \x y -> x + y
```

`psci` でこの関数が定義されていると、次のように関数の隣に2つの引数を空白で区切って書くことで、関数をこれらの引数に適用(apply)することができます。

```
> add 10 20
30
```

## 3.4 量化された型

前の節ではPreludeで定義された関数の型をいくつかの見てきました。たとえば `flip` 関数は次のような型を持っていました。

```
> :t Prelude.flip
forall a b c. (a -> b -> c) -> b -> a -> c
```

この `forall` キーワードは `flip` が全称量化された型(universally quantified type)を持っていることを示しています。これは、`a` や `b`、`c` をどの型に置き換えても、`flip` はその型でうまく動作するという意味です。

例えば、`a` を `Number`、`b` を `String`、`c` を `String` というように選んでみたとします。この場合、`flip` の型を次のように特殊化(specialize)することができます。

```
(Number -> String -> String) -> String -> Number -> String
```

量化された型を特殊化したいということをコードで示す必要はありません。特殊化は自動的に行われます。たとえば、すでにその型の `flip` を持っていたかのように、次のように単に `flip` を使用することができます。

```
> flip (\n s -> show n ++ s) "Ten" 10
```

```
"10Ten"
```

`a`、`b`、`c` の型はどんな型でも選ぶことができるといっても、型の不整合は生じないようにしなければなりません。`flip` に渡す関数の型は、他の引数の型と整合性がなくてはなりません。第2引数として文字列 `"Ten"`、第3引数として数 `10` を渡したのはそれが理由です。もし引数が逆になっているとうまくいかないでしょう。

```
> flip (\n s -> show n ++ s) 10 "Ten"
```

```
Error in value 10:  
Value does not have type Prim.String
```

## 3.5 字下げについての注意

JavaScriptとは異なり、PureScriptのコードは字下げの大きさに影響されます(indentation-sensitive)。これはHaskellと同じようになっています。コード内の空白の多寡は無意味ではなく、Cのような言語で中括弧によってコードのまとまりを示しているように、PureScriptでは空白がコードのまとまりを示すのに使われているということです。

宣言が複数行にわたる場合は、2つめの行は最初の行の字下げより深く字下げしなければなりません。

したがって、次は正しいPureScriptコードです。

```
add x y z = x +  
           y + z
```

しかし、次は正しいコードではありません。

```
add x y z = x +  
           y + z
```

後者では、PureScriptコンパイラはそれぞれの行ごとにひとつ、つまり2つの宣言であると構文解析します。

一般に、同じブロック内で定義された宣言は同じ深さで字下げする必要があります。例えば `psci` で `let` 文の宣言は同じ深さで字下げしなければなりません。次は正しいコードです。

```
> let x = 1
```

```
y = 2
```

しかし、これは正しくありません。

```
> let x = 1
      y = 2
```

PureScriptのいくつかの予約語(例えば `where` や `of`、`let`)は新たなコードのまとまりを導入しますが、そのコードのまとまり内の宣言はそれより深く字下げされている必要があります。

```
example x y z = foo + bar
  where
    foo = x * y
    bar = y * z
```

ここで `foo` や `bar` の宣言は `example` の宣言より深く字下げされていることに注意してください。

ただし、ソースファイルの先頭、最初の `module` 宣言における予約語 `where` だけは、この規則の唯一の例外になっています。

## 3.6 独自の型の定義

PureScriptで新たな問題に取り組むときは、まずはこれから扱おうとする値の型の定義を書くことから始めるのがよいでしょう。最初に、電話帳に含まれるレコードの型を定義してみます。

```
type Entry = { firstName :: String, lastName :: String, phone :: String }
```

これは `Entry` という型同義語(type synonym、型シノニム)を定義しています。型 `Entry` は等号の右辺と同じ型ということです。レコードの型はいずれも文字列である `firstName`、`lastName`、`phone` という3つのフィールドからなります。

それでは、2つめの型別名も定義してみましょう。電話帳のデータ構造として、単に項目の連結リストとして格納することにします。

```
type PhoneBook = List Entry
```

`List Entry` は `[Entry]` とは同じではないということに注意してください。 `[Entry]` は電話帳の項目の配列を意味しています。

## 3.7 型構築子と種

`List` は型構築子(type constructor、型コンストラクタ)の一例になっています。 `List` そのものは型ではなく、何らかの型 `a` があるとき `List a` が型になっています。つまり、 `List` は型引数(type argument) `a` をとり、新たな型 `List a` を構築するのです。

ちょうど関数適用と同じように、型構築子は他の型に並べることで適用されることに注意してください。型 `List Entry` は実は型構築子 `List` が型 `Entry` に適用されたものです。これは電話帳項目のリストを表しています。

(型注釈演算子 `::` を使って)もし型 `List` の値を間違えて定義しようとする、今まで見たことのないような種類のエラーが表示されるでしょう。

```
> :i Data.List
> Nil :: List
Expected type of kind *, was * -> *
```

これは種エラー(kind error)です。値がその型で区別されるのと同じように、型はその種(kind)によって区別され、間違った型の値が型エラーになるように、間違った種の型は種エラーを引き起こします。

`Number` や `String` のような、値を持つすべての型の種を表す `*` と呼ばれる特別な種があります。

型構築子にも種があります。たとえば、種 `* -> *` はちょうど `List` のような型から型への関数を表しています。ここでエラーが発生したのは、値が種 `*` であるような型を持つと期待されていたのに、`List` は種 `* -> *` を持っているためです。

`psci` で型の種を調べるには、`:k` 命令を使用します。例えば次のようになります。

```
> :k Number
*

> :i Data.List
> :k List
* -> *

> :k List String
*
```

PureScriptの種システムは他にも面白い種に対応していますが、それらについては本書の他の部分で見ていくことになるでしょう。

## 3.8 電話帳の項目の表示

それでは最初に、文字列で電話帳の項目を表現するような関数を書いてみましょう。まずは関数に型を与えることから始めます。型の定義は省略することも可能ですが、ドキュメントとしても役立つので型を書いておくようにすると良いでしょう。型宣言は関数の名前とその型を `::` 記号で区切るようにして書きます。

```
showEntry :: Entry -> String
```

`showEntry` は引数として `Entry` を取り `String` を返す関数であるということを、この型シグネチャは言っています。`showEntry` の定義は次のとおりです。

```
showEntry entry = entry.lastName ++ ", " ++  
                  entry.firstName ++ ": " ++  
                  entry.phone
```

この関数は `Entry` レコードの3つのフィールドを連結し、単一の文字列にします。

関数定義は関数の名前が始まり、引数名のリストが続きます。関数の結果は等号の後ろに定義します。フィールドはドットに続けてフィールド名を書くことで参照することができます。PureScriptでは、文字列連結はJavaScriptのような単一のプラス記号ではなく、ダブルプラス演算子 (`++`) を使用します。

## 3.9 はやめにテスト、たびたびテスト

`psci` 対話式処理系では反応を即座に得られるので、試行錯誤を繰り返したいときに向いています。それではこの最初の関数が正しく動作するかを `psci` を使用して確認してみましょう。

まず、これまで書かれたコードをビルドします。

```
$ grunt
```

次に、`psci` を起動し、この新しいモジュールをインポートするために `:i` 命令を使います。

```
$ psci
```

```
> :i Data.PhoneBook
```

レコードリテラルを使うと、電話帳の項目を作成することができます。レコードリテラルは JavaScript の無名オブジェクトと同じような構文です。これを `let` 式で名前に束縛します。

```
> let example = { firstName: "John", lastName: "Smith", phone: "555-555-5555" }
```

(Ctrl+ Dで式をを終了することを忘れないようにしましょう)。それでは、この関数を `example` に適用してみてください。

```
> showEntry example
```

```
"Smith, John: 555-555-5555"
```

おめでとうございます！PureScriptで初めて関数を書き、それを実行することができました。

## 3.10 電話帳の作成

今度は電話帳の操作を支援する関数をいくつか書いてみましょう。空の電話帳を表す値として、空のリストを使います。

```
emptyBook :: PhoneBook  
emptyBook = empty
```

既存の電話帳に値を挿入する関数も必要でしょう。この関数を `insertEntry` と呼ぶことにします。関数の型を与えることから始めましょう。

```
insertEntry :: Entry -> PhoneBook -> PhoneBook
```

`insertEntry` は、最初の引数として `Entry`、第二引数として `PhoneBook` を取り、新しい `PhoneBook` を返すということを、この型シグネチャは言っています。

既存の `PhoneBook` を直接変更することはしません。その代わりに、同じデータが含まれている新しい `PhoneBook` を返すようにします。このように、`PhoneBook` は永続データ構造 (persistent data structure) の一例となっています。これはPureScriptにおける重要な考え方



です。変更はコードの副作用であり、コードの振る舞いについての判断するのを難しくします。そのため、我々は可能な限り純粋な関数や不変のデータを好むのです。

`Data.List` の `Cons` 関数を使用すると `insertEntry` を実装できます。 `psci` を起動し `:t` コマンドを使って、この関数の型を見てみましょう。

```
$ psci

> :t Data.List.Cons

forall a. a -> List a -> List a
```

`Cons` は、なんらかの型 `a` の値と、型 `a` を要素に持つリストを引数にとり、同じ型の要素を持つ新しいリストを返すということを、この型シグネチャは言っています。 `a` を `Entry` 型として特殊化してみましょう。

```
Entry -> List Entry -> List Entry
```

しかし、 `List Entry` はまさに `PhoneBook` ですから、次と同じになります。

```
Entry -> PhoneBook -> PhoneBook
```

今回の場合、すでに適切な入力があります。 `Entry` 、と `PhoneBook` に `Cons` を適用すると、新しい `PhoneBook` を得ることができます。これこそまさに私たちが求めていた関数です！

`insertEntry` の実装は次のようになります。

```
insertEntry entry book = Cons entry book
```

等号の左側にある2つの引数 `entry` と `book` がスコープに導入されますから、これらに `Cons` 関数を適用して結果の値を作成しています。

## 3.11 カリー化された関数

PureScriptでは、関数は常にひとつの引数だけを取ります。 `insertEntry` 関数は2つの引数を取るように見えますが、これは実際にはカリー化された関数(curried function)の一例となっています。

`insertEntry` の型に含まれる `->` は右結合の演算子であり、つまりこの型はコンパイラによって次のように解釈されます。

```
Entry -> (PhoneBook -> PhoneBook)
```

すなわち、`insertEntry` は関数を返す関数である、ということです！この関数は単一の引数 `Entry` を取り、それから単一の引数 `PhoneBook` を取り新しい `PhoneBook` を返す新しい関数を返すのです。

これは例えば、最初の引数だけを与えると `insertEntry` を部分適用(partial application)できることを意味します。`psci` でこの結果の型を見てみましょう。

```
> :t insertEntry example
```

```
PhoneBook -> PhoneBook
```

期待したとおり、戻り値の型は関数になっていました。この結果の関数に、ふたつめの引数を適用することもできます。

```
> :t (insertEntry example) emptyBook
PhoneBook
```

ここで括弧は不要であることにも注意してください。次の式は同等です。

```
> :t insertEntry example emptyBook
PhoneBook
```

これは関数適用が左結合であるためで、なぜ単に空白で区切るだけで関数に引数を与えることができるのかも説明にもなっています。

本書では今後、「2引数の関数」というように表現することがあることに注意してください。これはあくまで、最初の引数を取り別の関数を返す、カーリー化された関数を意味していると考えてください。

今度は `insertEntry` の定義について考えてみます。

```
insertEntry :: Entry -> PhoneBook -> PhoneBook
insertEntry entry book = Cons entry book
```

もし式の右辺に明示的に括弧をつけるなら、`(Cons entry) book` となります。`insertEntry entry` はその引数が単に関数 `(Cons entry)` に渡されるような関数だということです。この2つの関数はどんな入力についても同じ結果を返しますから、つまりこれらは同じ関数です！よって、両辺から引数 `book` を削除できます。

```
insertEntry :: Entry -> PhoneBook -> PhoneBook
insertEntry entry = Cons entry
```

そして、同様の理由で両辺から `entry` も削除することができます。

```
insertEntry :: Entry -> PhoneBook -> PhoneBook
insertEntry = Cons
```

この処理はイータ変換(eta conversion)と呼ばれ、引数を参照することなく関数を定義するポイントフリー形式(point-free form)へと関数を書き換えるのに使うことができます。

`insertEntry` の場合には、イータ変換によって「`insertEntry` は単にリストに対する `cons` だ」と関数の定義はとても明確になりました。しかしながら、常にポイントフリー形式のほうがいいのかどうかには議論の余地があります。

## 3.12 あなたの電話番号は？

最小限の電話帳アプリケーションの実装で必要になる最後の関数は、名前で人を検索し適切な `Entry` を返すものです。これは小さな関数を組み合わせることでプログラムを構築するという、関数型プログラミングで鍵となる考え方のよい応用例になるでしょう。

まずは電話帳をフィルタリングし、該当する姓名を持つ項目だけを保持するようにするのがいいでしょう。それから、結果のリストの先頭の(head)要素を返すだけです。

この大まかな仕様に従って、この関数の型を計算することができます。まず `psci` を起動し、`filter` 関数と `head` 関数の型を見てみましょう。

```
$ psci

> :t Data.List.filter

forall a. (a -> Boolean) -> List a -> List a

:t Data.List.head
```

```
forall a. List a -> Maybe a
```

型の意味を理解するために、これらの2つの型の一部を取り出してみましょう。

`filter` はカーリー化された2引数の関数です。最初の引数は、リストの要素を取り `Boolean` 値を結果として返す関数です。第2引数は要素のリストで、返り値は別のリストです。

`head` は引数としてリストをとり、`Maybe a` という今まで見たことがないような型を返します。`Maybe a` は型 `a` のオプションな値、つまり `a` の値を持つか持たないかのどちらかの値を示しており、JavaScriptのような言語で値がないことを示すために使われる `null` の型安全な代替手段を提供します。これについては後の章で詳しく扱います。

`filter` と `head` の全称量化された型は、PureScriptコンパイラによって次のように特殊化されます。

```
filter :: (Entry -> Boolean) -> PhoneBook -> PhoneBook
```

```
head :: PhoneBook -> Maybe Entry
```

検索する関数の引数として姓と名前を渡す必要があるのもわかっています。

`filter` に渡す関数も必要になることもわかります。この関数を `filterEntry` と呼ぶことにしましょう。`filterEntry` は `Entry -> Boolean` という型を持っています。`filterEntry` という関数適用の式は、`PhoneBook -> PhoneBook` という型を持つでしょう。もしこの関数の結果を `head` 関数に渡すと、型 `Maybe Entry` の結果を得ることになります。

これまでのことをまとめると、この `findEntry` 関数の妥当な型シグネチャは次のようになります。

```
findEntry :: String -> String -> PhoneBook -> Maybe Entry
```

`findEntry` は、姓と名前の2つの文字列、および `PhoneBook` を引数にとり、`Maybe Entry` という型の値を結果として返すということを、この型シグネチャは言っています。結果の `Maybe Entry` という型は、名前が電話帳で発見された場合にのみ `Entry` の値を持ちます。

そして、`findEntry` の定義は次のようになります。

```
findEntry firstName lastName book = head $ filter filterEntry book
  where
```

```
filterEntry :: Entry -> Boolean
filterEntry entry = entry.firstName == firstName && entry.lastName == last
```

一歩ずつこのコードの動きを調べてみましょう。

`findEntry` は、どちらも文字列型である `firstName` と `lastName`、`PhoneBook` 型の `book` という3つの名前をスコープに導入します

定義の右辺では `filter` 関数と `head` 関数が組み合わされています。まず項目のリストをフィルタリングし、その結果に `head` 関数を適用しています。

真偽型を返す関数 `filterEntry` は `where` 節の内部で補助的な関数として定義されています。このため、`filterEntry` 関数はこの定義の内部では使用できますが、外部では使用することができません。また、`filterEntry` はそれを包む関数の引数に依存することができ、`filterEntry` は指定された `Entry` をフィルタリングするために引数

`firstName` と `lastName` を使用しているので、`filterEntry` が `findEntry` の内部にあることは必須になっています。

最上位での宣言と同じように、必ずしも `filterEntry` の型シグネチャを指定しなくてもよいことに注意してください。ただし、ドキュメントとしても役に立つので型シグネチャを書くことは推奨されています。

## 3.13 中置の関数適用

上でみた `findEntry` のコードでは、少し異なる形式の関数適用が使用されています。`head` 関数は中置の `$` 演算子を使って式 `filter filterEntry book` に適用されます。

これは `head (filter filterEntry book)` という通常の関数適用と同じ意味です。

`($)` はPreludeで定義されている通常の間数です。`($)` は次のように定義されています。

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

つまり、`($)` は関数と値をとり、その値にその関数を適用します。

しかし、なぜ通常の間数適用の代わりに `$` を使ったのでしょうか？その理由は `$` は右結合で優先順位の低い演算子だということにあります。これは、深い入れ子になった関数適用のための括弧を、`$` を使うと取り除くことができることを意味します。

たとえば、ある従業員の上司の住所がある道路を見つける、次の入れ子になった関数適用を考えてみましょう。

```
street (address (boss employee))
```

これは `$` を使用して表現すればずっと簡単になります。

```
street $ address $ boss employee
```

## 3.14 関数合成

イータ変換を使うと `insertEntry` 関数を簡略化できたのと同じように、引数をよく考察すると `findEntry` の定義を簡略化することができます。

引数 `book` が関数 `filter filterEntry` に渡され、この適用の結果が `head` に渡されることに注目してください。これは言いかたを変えれば、`filter filterEntry` と `head` の合成 (composition) に `book` は渡されるということです。

PureScriptの関数合成演算子は `<<<` と `>>>` です。前者は「逆方向の合成」であり、後者は「順方向の合成」です。

いずれかの演算子を使用して `findEntry` の右辺を書き換えることができます。逆順の合成を使用すると、右辺は次のようになります。

```
(head <<< filter filterEntry) book
```

この形式なら最初の定義にイータ変換の技を適用することができ、`findEntry` は最終的に次のような形式に到達します。

```
findEntry firstName lastName = head <<< filter filterEntry
  where
    ...
```

右辺を次のようににしても同じです。

```
filter filterEntry >>> head
```

どちらにしても、これは「`findEntry` はフィルタリング関数と `head` 関数の合成である」という

`findEntry` 関数のわかりやすい定義を与えます。

どちらの定義のほうがわかりやすいかの判断はお任せしますが、このように関数を部品として捉え、関数はひとつの役目だけをこなし、機能を関数合成で組み立てるというように考えると有用なことがよくあります。

## 3.15 テスト、テスト、テスト……

これでこのアプリケーションの中核部分が完成しましたので、`psci` を使って試してみましょ

```
$ psci

> :i Data.PhoneBook
```

まずは空の電話帳から項目を検索してみましょう(これは明らかに空の結果が返ってくることが期待されます)。

```
> findEntry "John" "Smith" emptyBook

Error in declaration main
No instance found for Prelude.Show (Data.Maybe.Maybe Data.PhoneBook.Entry<>)
```

エラーです！でも心配しないでください。これは単に 型 `Entry` の値を文字列として出力する方法を `psci` が知らないという意味のエラーです。

`findEntry` の返り値の型は `Maybe Entry` ですが、これは手作業で文字列に変換することができます。

`showEntry` 関数は `Entry` 型の引数を期待していますが、今あるのは `Maybe Entry` 型の値です。この関数は `Entry` 型のオプションな値を返すことを忘れないでください。行う必要があるのは、オプションな値の中に項目の値が存在すれば `showEntry` 関数を適用し、そうでなければ存在しないという値をそのまま伝播することです。

幸いなことに、`Prelude` モジュールはこれを行う方法を提供しています。`<$>` 演算子は `Maybe` のような適切な型構築子まで関数を「持ち上げる」ことができます(この本の後半で関手について説明するときに、この関数やそれに類似する他のものについて詳しく見ていきます)。

```
> showEntry <$> findEntry "John" "Smith" emptyBook
```



Nothing

今度はうまくいきました。この返り値 `Nothing` は、オプションな返り値に値が含まれていないことを示しています。期待していたとおりです。

もっと使いやすくするために、`Entry` を文字列として出力するような関数を定義し、毎回 `showEntry` を使わなくてもいいようにすることもできます。

```
> let printEntry firstName lastName book = showEntry <$> findEntry firstName lastN
```

それでは空でない電話帳を作成してもう一度試してみましょう。先ほどの項目の例を再利用します。

```
> let john = { firstName: "John", lastName: "Smith", phone: "555-555-5555" }

> let book1 = insertEntry john emptyBook

> printEntry "John" "Smith" book1

Just ("Smith, John: 555-555-5555")
```

今度は結果が正しい値を含んでいました。`book1` に別の名前で項目を挿入して、ふたつの名前がある電話帳 `book2` を定義し、それぞれの項目を名前で検索してみてください。

## 演習

1. (簡単) `findEntry` 関数の定義の主な部分式の型を書き下し、`findEntry` 関数についてよく理解しているか試してみましょう。たとえば、`findEntry` の定義のなかにある `head` 関数の型は `List Entry -> Maybe Entry` と特殊化されています。
2. (簡単) `findEntry` の既存のコードを再利用し、与えられた電話番号から `Entry` を検索する関数を書いてみましょう。また、`psci` で実装した関数をテストしてみましょう。
3. (やや難しい) 指定された名前が `PhoneBook` に存在するかどうかを調べて真偽値で返す関数を書いてみましょう。ヒント: リストが空かどうかを調べる `Data.List.null` 関数の型を `psci` で調べてみてください。
4. (難しい) 姓名が重複している項目を電話帳から削除する関数 `removeDuplicates` を書いてみましょう。ヒント: 値どうしの等価性を定義する述

語関数に基づいてリストから重複要素を削除する関数 `List.nubBy` の型を、`psci` を使用して調べてみましょう。

## 3.16 まとめ

---

この章では、関数型プログラミングの新しい概念をいくつか導入しました。

- 不変データ型と純粋な関数の重要性
- 対話的モード `psci` を使用して関数を調べたり思いついたことを試す方法
- 検証や実装の道具としての型の役割
- 多引数関数を表現する、カーリー化された関数の使用
- 関数合成で小さな部品を組み合わせてのプログラムの構築
- `where` 節を利用したコードの構造化
- `Maybe` 型を使用してnull値を回避する方法
- イータ変換や関数合成のような手法を利用した、よりわかりやすいコードへの再構成

次の章からは、これらの考えかたに基づいて進めていきます。

# 4 再帰、マップ、畳み込み

---

## 4.1 この章の目標

---

この章では再帰関数を使ってどのようにアルゴリズムを構造化するかについて見ていきましょう。再帰はこの本を通じて使用する関数型プログラミングの基本的な手法です。

また、PureScriptの標準ライブラリから標準的な関数をいくつか扱います。`map` や `fold` のような関数だけでなく、`filter` や `concatMap` といった便利で特殊なものについても見ていきます。

この章では仮想的なファイルシステムを操作する関数のライブラリを動機付けに用います。この章で学ぶ手法を応用して、模擬的なファイルシステムによって表されるファイルのプロパティを計算する関数を記述します。

## 4.2 プロジェクトの準備

---

この章のソースコードには、`src/Data/Path.purs` と `src/FileOperations.purs` という2つのファイルが含まれています。

`Data.Path` モジュールには、仮想ファイルシステムが含まれています。このモジュールの内容を変更する必要はありません。

`FileOperations` モジュールは、`Data.Path` APIを使用する関数が含まれています。演習への回答はこのファイルだけで完了することができます。

このプロジェクトには以下のBower依存関係があります。

- `purescript-maybe` : `Maybe` 型構築子が定義されています
- `purescript-arrays` : 配列を扱うための関数が定義されています
- `purescript-foldable-traversable` : 配列の畳み込みやその他のデータ構造に関する関数が定義されています

## 4.3 はじめに

---

再帰は一般のプログラミングでも重要な手法ですが、純粋関数型プログラミングでは特に当たり前のように用いられます。この章で見ていくように、再帰はプログラムの変更可能な状態を減らすために役立つからです。

再帰は分割統治(Divide and conquer)戦略と密接な関係があります。分割統治とはすなわち、いろいろな入力に対する問題を解決するために、入力を小さな部分に分割し、それぞれの部分について問題を解いて、部分ごとの答えから最終的な答えを組み立てるということです。

それでは、PureScriptにおける再帰の簡単な例をいくつか見てみましょう。

次は階乗関数(factorial function)のよくある例です。

```
fact :: Number -> Number
fact 0 = 1
fact n = n * fact (n - 1)
```

部分問題へ問題を分割することによって階乗関数がどのように計算されるかがわかります。より小さい数へと階乗を計算していくということです。ゼロに到達すると答えは直ちに求められます。

次はフィボナッチ関数(Fibonacci function)を計算するという、これもまたよくある例です。

```
fib :: Number -> Number
fib 0 = 1
fib 1 = 1
```

```
fib n = fib (n - 1) + fib (n - 2)
```

やはり部分問題の解決策を考えることで全体を解決していることがわかります。この場合、`fib (n - 1)` と `fib (n - 2)` という式に対応した2つの部分問題があります。これらの2つの部分問題が解決されているときに、部分的な答えを加算することで答えを組み立てることができます。

## 4.4 配列上での再帰

再帰関数の定義は、`Number` 型だけに限定されるものではありません！本書の後半でパターン照合(pattern matching)を扱うときに、いろいろなデータ型の上での再帰関数について見ていきますが、今は数と配列に限っておきます。

入力ゼロでないかどうかについて分岐するのと同じように、配列の場合も、配列が空でないかどうかについて分岐していきます。再帰を使用して配列の長さを計算する次の関数を考えてみます。

```
import Data.Array (null)
import Data.Array.Unsafe (tail)

length :: forall a. [a] -> Number
length arr =
  if null arr
  then 0
  else 1 + length (tail arr)
```

この関数では配列が空かどうかで分岐するために `if ... then ... else` 式を使っています。この `null` 関数は配列が空のときに `true` を返します。空の配列の長さはゼロであり、空でない配列の長さは配列の先頭を取り除いた残りの部分の長さより1大きいというわけです。

この例はJavaScriptで配列の長さを調べるのにはどうみても実用的な方法とはいえませんが、次の演習を完了するための手がかりとしては充分でしょう。

### 演習

1. (簡単) 入力が偶数であるとき、かつそのときに限り `true` に返すような再帰関数を書いてみましょう。
2. (少し難しい) 配列内の偶数の数を数える再帰関数を書いてみましょう。ヒント：

`Data.Array.Unsafe` モジュールの `head` 関数を使用します。

## 4.5 マップ

`map` 関数は配列に対する再帰関数のひとつです。これは、配列の各要素に順番に関数を適用することによって、配列の要素を変換するために使用されます。そのため、配列の内容は変更されますが、その形状(ここでは「長さ」)は保存されます。

本書で後ほど型クラス(type class)を扱うとき、形状を保存しながら型構築子のクラスを変換する関手(functor)と呼ばれる関数を紹介しますが、その時に `map` 関数は関手の具体例であることがわかります。

それでは `psci` で `map` 関数を試してみましょう。

```
$ psci

> :i Data.Array
> map (\n -> n + 1) [1, 2, 3, 4, 5]

[2, 3, 4, 5, 6]
```

`map` がどのように使われているかに注目してください。最初の引数には配列がどのように対応付けられるかを示す関数、第2引数には配列そのものを渡します。

## 4.6 中置演算子

バッククォート(`)で関数名を囲むと、対応関係を表す関数と配列の間に `map` 関数を書くことができます。

```
> (\n -> n + 1) `map` [1, 2, 3, 4, 5]

[2, 3, 4, 5, 6]
```

この構文は中置関数適用と呼ばれ、どんな関数でもこのように中置することができます。普通は2引数の関数に対して使うのが最も適切でしょう。

配列を扱うときは、`map` 関数と等価な `<$>` という演算子が存在します。この演算子は他の二項演算子と同じように中置で使用することができます。

```
> (\n -> n + 1) <$> [1, 2, 3, 4, 5]

[2, 3, 4, 5, 6]
```

注意: `<$>` の型は実際には `map` よりも一般的ですが、中置適用のほうが自然であるなら、`map` の代わりに `<$>` を使ってもたいいてい場合は大丈夫です。

それでは `map` の型を見てみましょう。

```
> :t map
forall a b. (a -> b) -> [a] -> [b]
```

`map` 関数に適用するときには `a` と `b` という2つの型を自由に選ぶことができると、この型は言っています。`a` は元の配列の要素の型で、`b` は目的の配列の要素の型です。もっと言えば、`map` が配列要素の型を変化させても構わないということです。例えば、数値を文字列に変換するのにマップを使用することができます。

```
> show <$> [1, 2, 3, 4, 5]

["1","2","3","4","5"]
```

中置演算子 `<$>` は特別な構文のように見えるかもしれませんが、実際には普通の PureScript 関数です。中置構文を使用した単なる適用にすぎません。実際、括弧でその名前を囲むと、この関数を通常関数のように使用することができます。これは、配列に対する `map` の代わりに、括弧で囲まれた `(<$>)` という名前を使うことができるということです。

```
> (<$>) show [1, 2, 3, 4, 5]

["1","2","3","4","5"]
```

新しい中置演算子を定義するには、関数と同じ記法を使います。演算子名を括弧で囲み、あとは普通関数のようにその中置演算子を定義します。たとえば、`Data.Array` モジュールでは次のように `range` 関数と同じ振る舞いの中置演算子 `(..)` を定義しています。

```
(..) :: Number -> Number -> [Number]
(..) = range
```

この演算子は次のように使うことができます。

```
> 1 .. 5

[1, 2, 3, 4, 5]

> show <$> (1 .. 5)

["1","2","3","4","5"]
```

注意：独自の中置演算子は自然な構文を持った領域特化言語を定義するのに優れた手段になりえます。ただし、使用には充分注意してください。初心者が読めないコードになることがありますから、新たな演算子の定義には慎重になるのが賢明です。

上記の例では、`1 .. 5` という式は括弧で囲まれていましたが、実際にはこれは必要ありません。なぜなら、`Data.Array` モジュールは、`<$>` に割り当てられた優先順位より 高い優先順位を `..` 演算子に割り当てているからです。PureScriptでは予約語 `infix` を使用して独自の演算子に優先順位を割り当てる方法が提供されています。

```
infix 5 ..
```

ここでは `<$>` の優先順位よりも高い優先順位5を `(..)` に割り当てており、これはつまり括弧を付け加える必要がないということです。

```
> show <$> 1 .. 5

["1","2","3","4","5"]
```

中置演算子に左結合性または右結合性を与えたい場合は、代わりに予約語 `infixl` と `infixr` を使います。

## 4.7 配列のフィルタリング

`Data.Array` モジュールでは他にも、`map` と同様によく使われる関数 `filter` も提供しています。この関数は、述語関数に適合する要素のみを残し、既存の配列から新しい配列を作成する機能を提供します。

たとえば、1から10までの数で、偶数であるような数の配列を計算したいとします。これは次のように行うことができます。

```
> :i Data.Array
```

```
> filter (\n -> n % 2 == 0) (1 .. 10)
[2,4,6,8,10]
```

## 演習

1. (簡単) `map` 関数や `<$>` 関数を使用して、配列に格納された数のそれぞれの平方を計算する関数を書いてみましょう。
2. (簡単) `filter` 関数を使用して、数の配列から負の数を取り除く関数を書いてみましょう。
3. (やや難しい) `filter` 関数と同じ意味の中置演算子 `<$?>` を定義してみましょう。先ほどの演習の回答を、この新しい演算子を使用して書き換えてください。また、`psci` でこの演算子の優先順位と結合性を試してみてください。

## 4.8 配列の平坦化

`Data.Array` で定義されている配列に関する標準の関数としては、`concat` 関数もあります。`concat` は配列の配列をひとつの配列へと平坦化します。

```
> :i Data.Array
> :t concat

forall a. [[a]] -> [a]

> concat [[1, 2, 3], [4, 5], [6]]

[1, 2, 3, 4, 5, 6]
```

関連する関数として、`concat` と `map` を組み合わせたような `concatMap` と呼ばれる関数もあります。`map` は(相異なる型も可能な)値からの値への関数を引数に取りますが、それに対して `concatMap` は値から値の配列の関数を取ります。

実際に動かして見てみましょう。

```
> :i Data.Array

> :t concatMap
forall a b. (a -> [b]) -> [a] -> [b]

> concatMap (\n -> [n, n * n]) (1 .. 5)
```



```
[1,1,2,4,3,9,4,16,5,25]
```

ここでは、数とその数とその数の平方の2つの要素からなる配列に写す関数 `\n -> [n, n * n]` を引数に `concatMap` を呼び出しています。結果は、1から5の数と、そのそれぞれの数の平方からなる、10個の数になります。

`concatMap` がどのように結果を連結しているのかに注目してください。渡された関数を元の配列のそれぞれの要素について一度ずつ呼び出し、その関数はそれぞれ配列を生成します。最後にそれらの配列を単一の配列に押し潰し、それが結果となります。

`map` と `filter`、`concatMap` は、「配列内包表記」(array comprehensions)と呼ばれる配列に関するあらゆる関数の基盤を形成しています。

## 4.9 配列内包表記

数 `n` の2つの因数を見つけたいとしましょう。これを行うための簡単な方法のひとつとしては、総当りで行う方法があります。つまり、`1` から `n` の数のすべての組み合わせを生成し、それを乗算してみるわけです。もしその積が `n` なら、`n` の因数の組み合わせを見つけたということになります。

配列内包表記を使用すると、この計算を実行することができます。`psci` を対話式の開発環境として使用し、ひとつずつこの手順を進めていきましょう。

`n` 以下の数の組み合わせの配列を生成する最初の手順は、`concatMap` を使えば行うことができます。

`1 .. n` のそれぞれの数を配列 `1 .. n` へとマッピングすることから始めましょう。

```
> let pairs n = concatMap (\i -> 1 .. n) (1 .. n)
```

この関数をテストしてみましょう。

```
> pairs 3
[1,2,3,1,2,3,1,2,3]
```

これは求めているものとはぜんぜん違います。単にそれぞれの組み合わせの2つ目の要素を返すのではなく、ペア全体を保持することができるように、内側の `1 .. n` の複製について関数をマッピングする必要があります。

```
> let pairs n = concatMap (\i -> map (\j -> [i, j]) (1 .. n)) (1 .. n)

> pairs 3
[[1,1],[1,2],[1,3],[2,1],[2,2],[2,3],[3,1],[3,2],[3,3]]
```

いい感じになってきました。しかし、`[1, 2]` と `[2, 1]` の両方があるように、余計な組み合わせが生成されています。`j` を `i` から `n` の範囲に限定することで、2つ目の場合を取り除くことができます。

```
> let pairs n = concatMap (\i -> map (\j -> [i, j]) (i .. n)) (1 .. n)

> pairs 3
[[1,1],[1,2],[1,3],[2,2],[2,3],[3,3]]
```

すばらしい！因数の候補のすべての組み合わせを得たので、`filter` を使って、積が与えられた `n` であるような組み合わせを選択することができます。

```
> :i Data.Foldable

> let factors n = filter (\pair -> product pair == n) (pairs n)

> factors 10
[[1,10],[2,5]]
```

このコードでは、`purescript-foldable-traversable` ライブラリの `Data.Foldable` モジュールにある `product` 関数を使っています。

うまくいきました！重複のなく、因数の組み合わせの正しい集合を見つけることができました。

## 4.10 do 記法

機能は実現できましたが、このコードの可読性は大幅に向上することができます。`map` や `concatMap` はとても基本的な関数で、`do` 記法(`do notation`)と呼ばれる特別な構文の基盤をなしています(もっと厳密に言えば、それらの一般化である `<$>` と `>>=` が基盤をなしています)。

注意: `map` と `concatMap` が配列内包表記を書けるようにしているように、もっと一般的な演算子である `<$>` と `>>=` はモナド内包表記(`monad comprehensions`)と呼ばれているものを書けるようにします。本書の後半ではモナド(`monad`)の例をたっぷり見ていくことになります

が、それはこの章ではありません。

do記法を使うと、先ほどの `factors` 関数を次のように書き直すことができます。

```
factors :: Number -> [[Number]]
factors n = filter (\xs -> product xs == n) $ do
  i <- 1 .. n
  j <- i .. n
  return [i, j]
```

予約語 `do` はdo記法を使うコードのブロックを導入します。このブロックは幾つかの型の式で構成されています。

- 配列の要素を名前に束縛する式。これは後方向きの矢印 `<-` で示されていて、その左側は名前、右側は配列の型を持つ式です。
- 名前に配列の要素を束縛しない式。最後の行の `return [i, j]` が、この種類の式の一例です。
- `let` キーワードを使用し、式に名前を与える式(ここでは使われていません)。

この新しい記法を使うとアルゴリズムの構造がわかりやすくなることがあります。心のなかで `<-` を「選ぶ」という単語に置き換えるとすると、「1からnの間の要素 `i` を選び、それから `i` からnの間の要素 `j` を選び、`[i, j]` を返す」というように読むことができるかもしれません。

最後の行の `return` 関数は予約語ではないことに注意してください。これは、他の関数と同様に `psci` で評価することができる、通常関数です。ただし、`psci` で評価するには型を明示しなければなりません。

```
> return [1, 2] :: [[Number]]
[[1, 2]]
```

配列の場合、`return` は単に1要素の配列を作成します。実際に、`return` の代わりにこの形式を使うように `factors` 関数を変更することもできます。

```
factors :: Number -> [[Number]]
factors n = filter (\xs -> product xs == n) $ do
  i <- 1 .. n
  j <- i .. n
  [[i, j]]
```

そして、結果は同じになります。

## 4.11 ガード

`factors` 関数を更に改良する方法としては、`filter`を配列内包表記の内側に移動するというものがあります。これは `purescript-control` ライブラリにある `Control.MonadPlus` モジュールの `guard` 関数を使用することで可能になります。

```
factors :: Number -> [[Number]]
factors n = do
  i <- range 1 n
  j <- range i n
  guard $ i * j == n
  return [i, j]
```

`return` と同じように、`guard` 関数は予約語ではありません。どのように動作するかを理解するために、`psci` で通常関数のように `guard` を適用してみましょう。

`guard` 関数の型は、ここで必要になる以上に一般的な型です。

```
> :i Control.MonadPlus
> :t guard

forall m. (MonadPlus m) => Boolean -> m Unit
```

今回の場合は、`psci` は次の型を報告するものと考えてください。

```
Boolean -> [Unit]
```

次の計算の結果から、配列における `guard` 関数について今知りたいことはすべてわかります。

```
> :i Data.Array
> length $ guard true

1

> length $ guard false

0
```

つまり、`guard` が `true` に評価される式を渡された場合、単一の要素を持つ配列を返すの

です。もし式が `false` と評価された場合は、その結果は空です。

ガードが失敗した場合、配列内包表記の現在の分岐は、結果なしで早めに終了されることを意味します。これは、`guard` の呼び出しが、途中の配列に対して `filter` を使用することと同じだということです。これらが同じ結果になることを確認するために、`factors` の二つの定義を試してみてください。

## 演習

1. (簡単) `factors` 関数を使用して、整数の引数が素数であるかどうかを調べる関数 `isPrime` を定義してみましょう。
2. (やや難しい) 2つの配列の直積集合を見つけるための関数を書いてみましょう。直積集合とはつまり、要素 `a`、`b` のすべての組み合わせの集合です。ここで `a` は最初の配列の要素、`b` は2つ目の配列の要素です。
3. (やや難しい) ピタゴラスの三つ組数とは、 $a^2 + b^2 = c^2$  を満たすような3つの数の配列 `[a, b, c]` のことです。配列内包表記の中で `guard` 関数を使用して、数 `n` を引数に取り、どの要素も `n` より小さいようなピタゴラスの三つ組数すべてを求める関数を書いてみましょう。その関数は `Number -> [[Number]]` という型を持っていなければなりません。
4. (難しい) `Data.Foldable` から `any` 関数を探しましょう。配列内包表記の代わりに `any` 関数を使用して `factors` 関数を書き換えてみましょう。注意: `psci` によって報告される `any` の型は、必要以上に一般的です。この演習での目的には、`any` の型は `forall a. (a -> Boolean) -> [a] -> Boolean` であると考えることができます。
5. (鬼のように難しい) `factors` 関数を使用して、数 `n` のすべての因数分解を求める関数 `factorizations` を定義してみましょう。数 `n` の因数分解とは、それらの積が `n` であるような整数の配列のことです。ヒント: 1は因数ではないと考えてください。また、無限再帰に陥らないように注意しましょう。

## 4.12 畳み込み

再帰を利用して実装される興味深い関数としては、配列に対する左畳込み(left fold)と右畳み込み(right fold)があります。

`psci` を使って、`Data.Foldable` モジュールをインポートし、`foldl` と `foldr` 関数の型を調べることから始めましょう。

```
> :i Data.Foldable
```

```
> :t foldl
```

```
forall a b f. (Foldable f) => (b -> a -> b) -> b -> f a -> b
```

```
> :t foldr
```

```
forall a b f. (Foldable f) => (a -> b -> b) -> b -> f a -> b
```

これらの型は、現在興味があるものよりも一般的です。この章の目的では、`psci` は以下の(より具体的な)答えを与えていたと考えておきましょう。

```
> :t foldl
```

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

```
> :t foldr
```

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

どちらの型でも、`a` は配列の要素の型に対応しています。型'`b`'は、配列を走査(traverse)したときの結果を蓄積する「累積器」(accumulator)の型だと考えることができます。

`foldl` 関数と `foldr` 関数の違いは走査の方向です。`foldr` が「右から」配列を畳み込むのに対して、`foldl` は「左から」配列を畳み込みます。

これらの関数の動きを見てみましょう。`foldl` を使用して数の配列の和を求めてみます。型 `a` は `Number` になり、結果の型 `b` も `Number` として選択することができます。ここでは、次の要素を累積器に加算する `Number -> Number -> Number` という型の関数、`Number` 型の累積器の初期値、和を求めたい `Number` の配列という、3つの引数を提供する必要があります。最初の引数としては、加算演算子を使用することができますし、累積器の初期値はゼロになります。

```
> foldl (+) 0 (1 .. 5)
15
```

この場合では、引数が逆になっても `(+)` 関数は同じ結果を返すので、`foldl` と `foldr` のどちらでも問題ありません。

```
> foldr (+) 0 (1 .. 5)
15
```

`foldl` と `foldr` の違いを説明するために、畳み込み関数の選択が影響する例も書いてみ

ましょう。加算関数の代わりに、文字列連結を使用して文字列を作ってみます。

```
> foldl (\acc n -> acc ++ show n) "" [1,2,3,4,5]
"12345"

> foldr (\n acc -> acc ++ show n) "" [1,2,3,4,5]
"54321"
```

これは、2つの関数の違いを示しています。左畳み込み式は、以下の関数適用と同等です。

```
((((((" " ++ show 1) ++ show 2) ++ show 3) ++ show 4) ++ show 5)
```

それに対し、右畳み込みは以下に相当します。

```
((((((" " ++ show 5) ++ show 4) ++ show 3) ++ show 2) ++ show 1)
```

## 4.13 末尾再帰

再帰はアルゴリズムを定義するための強力な手法ですが、問題も抱えています。入力が大きすぎる場合、JavaScriptで再帰関数を評価しようとするとうまくオーバーフローでエラーを起こす可能性があるのです。

`psci` で次のコードを入力すると、この問題を簡単に検証できます。

```
> let f 0 = 0
      f n = 1 + f (n - 1)

> f 10
10

> f 10000
RangeError: Maximum call stack size exceeded
```

これは問題です。関数型プログラミングの基本的な手法として再帰を採用しようとするなら、無限かもしれない再帰でも扱える方法が必要です。

PureScriptは末尾再帰最適化(tail recursion optimization)の形でこの問題に対する部分的な解決策を提供しています。

注意:この問題へのより完全な解決策としては、いわゆるトランポリン(trampolining)を使用したライブラリで実装する方法がありますが、それはこの章で扱う範囲を超えています。

末尾再帰の最適化が可能かどうかには条件があります。末尾位置(tail position)にある関数の再帰的な呼び出しは、スタックフレームが確保されないジャンプに置き換えることができます。それが関数が戻るより前の最後の呼び出しであるとき、呼び出しは末尾位置にあるといえます。なぜこの例でスタックオーバーフローを観察したのかはこれが理由です。この `fact` の再帰呼び出しは、末尾位置ではないからです。

実際には、PureScriptコンパイラは再帰呼び出しをジャンプに置き換えるのではなく、再帰的な関数全体をwhileループに置き換えます。

以下はすべての再帰呼び出しが末尾位置にある再帰関数の例です。

```
fact :: Number -> Number -> Number
fact 0 acc = acc
fact n acc = fact (n - 1) (acc * n)
```

`fact` への再帰呼び出しは、この関数の中で起こる最後のものである、つまり末尾位置にあることに注意してください。

## 4.14 累積器

末尾再帰ではない関数を末尾再帰関数に変える一般的な方法としては、累積器引数(accumulator parameter)を使用する方法があります。結果を累積するために返り値を使うと末尾再帰を妨げることがありますが、それとは対照的に累積器引数は返り値を累積する関数へ追加される付加的な引数です。

たとえば、入力配列を逆順にする、この配列の再帰を考えてみましょう。

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

この実装は末尾再帰ではないので、大きな入力配列に対して実行されると、生成されたJavaScriptはスタックオーバーフローを発生させるでしょう。しかし、代わりに、結果を蓄積するための2つ目の引数を関数に導入することで、これを末尾再帰に変えることができます。

```
reverse :: forall a. [a] -> [a]
reverse = reverse' []
```



```
where
reverse' acc [] = acc
reverse' acc (x : xs) = reverse' (x : acc) xs
```

ここでは、配列を逆転させる作業を補助関数 `reverse'` に委譲しています。関数 `reverse'` が末尾再帰的であることに注目してください。その唯一の再帰呼び出しは、最後の場合の末尾位置にあります。これは、生成されたコードがwhileループとなり、大きな入力でもスタックが溢れないことを意味します。

`reverse` のふたつめの実装を理解するためには、部分的に構築された結果を状態として扱うために、補助関数 `reverse'` で累積器引数の使用することが必須であることに注意してください。結果は空の配列で始まりますが、入力配列の要素ひとつごとに、ひとつずつ大きくなっていきます。後の要素は配列の先頭に追加されるので、結果は元の配列の逆になります！

累積器を「状態」と考えることもできますが、直接に変更がされているわけではないことにも注意してください。この累積器は不変の配列であり、計算に沿って状態受け渡すために、単に関数の引数を使います。

## 4.15 明示的な再帰より畳み込みを選ぶ

末尾再帰を使用して再帰関数を記述することができれば末尾再帰最適化の恩恵を受けることができるので、すべての関数をこの形で書こうとする誘惑にかられます。しかし、多くの関数は配列やそれに似たデータ構造に対する折り畳みとして直接書くことができることを忘れがちです。`map` や `fold` のようなコンビネータを使って直接アルゴリズムを書くことには、コードの単純さという利点があります。これらのコンビネータはよく知られており、アルゴリズムの意図をはっきりとさせるのです。

例えば、先ほどの `reverse` の例は、畳み込みとして少なくとも2つの方法で書くことができます。`foldr` を使用すると次のようになります。

```
> :i Data.Foldable
> let reverse :: forall a. [a] -> [a]
    reverse = foldr (\x xs -> xs ++ [x]) []

> reverse [1, 2, 3]

[3,2,1]
```

`foldl` を使って `reverse` を書くことは、読者への課題として残しておきます。

## 演習

1. (簡単) `foldl` を使って、真偽値の配列の要素すべてが真かどうか調べてみてください。
2. (やや難しい) 関数 `foldl (==) false xs` が真を返すような配列 `xs` とはどのようなものか説明してください。
3. (やや難しい) 累積器引数を使用して、次の関数を末尾再帰形に書きなおしてください。

```
count :: forall a. (a -> Boolean) -> [a] -> Number
count _ [] = 0
count p (x : xs) = if p x then 1 + count p xs else count p xs
```

4. (やや難しい) `foldl` を使って `reverse` を書いてみましょう。

## 4.16 仮想ファイルシステム

この節では、これまで学んだことを応用して、模擬的なファイルシステムで動作する関数を書いていきます。事前に定義されたAPIで動作するように、マップ、畳み込み、およびフィルタを使用します。

`Data.Path` モジュールでは、次のように仮想ファイルシステムのAPIが定義されています。

- ファイルシステム内のパスを表す型 `Path` があります。
- ルートディレクトリを表すパス `root` があります。
- `ls` 関数はディレクトリ内のファイルを列挙します。
- `filename` 関数は `Path` のファイル名を返します。
- `size` 関数は `Path` が示すファイルの大きさを返します。
- `isDirectory` 関数はファイルかディレクトリかを調べます。

型については、型定義は次のようになっています。

```
root :: Path

ls :: Path -> [Path]

filename :: Path -> String

size :: Path -> Maybe Number
```

```
isDirectory :: Path -> Boolean
```

`psci` でこのAPIを試してみましょう。

```
> :i Data.Path

> root
/

> isDirectory root
true

> ls root
[/bin/,/etc/,/home/]
```

`FileOperations` モジュールでは、`Data.Path` APIを操作するための関数を定義されています。`Data.Path` モジュールを変更したり実装を理解する必要はありません。すべて `FileOperations` モジュールだけで作業を行います。

## 4.17 すべてのファイルの一覧

それでは、内側のディレクトリまで、すべてのファイルを列挙する関数を書いてみましょう。この関数は以下のような型を持つでしょう。

```
allFiles :: Path -> [Path]
```

再帰を使うとこの関数を定義することができます。まずは `ls` を使用してディレクトリの直接の子を列挙します。それぞれの子について再帰的に `allFiles` を適用すると、それぞれパスの配列が返ってくるでしょう。`concatMap` を適用すると、この結果を同時に平坦化することができます。

最後に、`:` 演算子を使って現在のファイルも含めます。

```
allFiles file = file : concatMap allFiles (ls file)
```

それでは `psci` でこの関数を試してみましょう。

```
> :i FileOperations
> :i Data.Path
```

```
> allFiles root
```

```
[/,/bin/,/bin/cp,/bin/ls,/bin/mv,/etc/,/etc/hosts, ...]
```

すばらしい！do記法で配列内包表記を使ってもこの関数を書くことができるので見ていきましょう。

逆向きの矢印は配列から要素を選択するのに相当することを思い出してください。最初の手順は、引数の直接の子から要素を選択することです。それから、単にそのファイルに対してこの再帰関数を呼びします。do記法を使用しているので、再帰的な結果をすべて連結する `concatMap` が暗黙に呼び出されています。

新しいコードは次のようになります。

```
allFiles' :: Path -> [Path]
allFiles' file = file : do
  child <- ls file
  allFiles' child
```

`psci` で新しいコードを試してみてください。同じ結果が返ってくるはずです。どちらのほうかわかりやすいかの選択はお任せします。

## 演習

1. (簡単) ディレクトリのすべてのサブディレクトリの中まで、ディレクトリを除くすべてのファイルを返すような関数 `onlyFiles` を書いてみてください。
2. (やや難しい) このファイルシステムで最大と最小のファイルを決定するような畳込みを書いてください。
3. (難しい) ファイルを名前で検索する関数 `whereIs` を書いてください。この関数は型 `Maybe Path` の値を返すものとします。この値が存在するなら、そのファイルがそのディレクトリに含まれていることを表します。この関数は次のように振る舞う必要があります。

```
> whereIs "/bin/ls"
Just (/bin/)

> whereIs "/bin/cat"
Nothing
```

ヒント:do記法で配列内包表記を使用して、この関数を記述してみてください。

## 4.18 まとめ

---

この章では、アルゴリズムを簡潔に表現する手段として、PureScriptでの再帰の基本を説明しました。また、独自の中置演算子や、マップ、フィルタリングや畳み込みなどの配列に対する標準関数、およびこれらの概念を組み合わせた配列内包表記を導入しました。最後に、スタックオーバーフローエラーを回避するために末尾再帰を使用することの重要性、累積器引数を使用して末尾再帰形に関数を変換する方法を示しました。

# 5 パターン照合

---

## 5.1 この章の目標

---

この章では代数的データ型とパターン照合という2つの新しい概念を導入します。また、行多相というPureScriptの型システムの興味深い機能についても簡単に扱います。

パターン照合(Pattern matching)は関数型プログラミングでは一般的な手法で、複数の場合に実装を分解することにより、開発者は潜在的に複雑な動作の関数を簡潔に書くことができます。

代数的データ型はPureScriptの型システムの機能で、パターン照合とも密接に関連しています。

この章の目的は、代数的データ型やパターン照合を使用して、単純なベクターグラフィックスを描画し操作するためのライブラリを書くことです。

## 5.2 プロジェクトの準備

---

この章のソースコードはファイル `src/data/Picture.purs` で定義されています。

このプロジェクトはこれまで見てきたBowerパッケージを引き続き使用しますが、それに加えて次の新しい依存関係が追加されます。

- `purescript-globals` : 一般的なJavaScriptの値や関数の取り扱いを可能にします。
- `purescript-math` : JavaScriptの `Math` オブジェクトの関数群を利用可能にします。

`Data.Picture` モジュールは、簡単な図形を表すデータ型 `Shape` や、図形の集合である

型 `Picture`、及びこれらの型を扱うための関数を定義しています。

このモジュールでは、データ構造の畳込みを行う関数を提供する `Data.Foldable` モジュールもインポートします。

```
module Data.Picture where

import Data.Foldable
```

## 5.3 単純なパターン照合

それではコード例を見ることから始めましょう。パターン照合を使用して2つの整数の最大公約数を計算する関数は、次のようになります。

```
gcd :: Number -> Number -> Number
gcd n 0 = n
gcd 0 m = m
gcd n m = if n > m then gcd (n - m) m else gcd n (m - n)
```

このアルゴリズムはユークリッドの互除法と呼ばれています。その定義をオンラインで検索すると、おそらく上記のコードによく似た数学の方程式が見つかるでしょう。パターン照合の利点のひとつは、上記のようにコードを場合分けして定義することができ、数学関数の定義と似たような簡潔で宣言型なコードを書くことができることです。

パターン照合を使用して書かれた関数は、条件と結果の組み合わせによって動作します。この定義の各行は選択枝(alternative)や場合(case)と呼ばれています。等号の左辺の式はパターンと呼ばれており、それぞれの場合は空白で区切られた1つ以上のパターンで構成されています。等号の右側の式が評価され値が返される前に引数が満たさなければならない条件について、これらの場合は説明しています。それぞれの場合は上からこの順番に試されていき、最初に入力に適合した場合が返り値を決定します。

たとえば、`gcd` 関数は次の手順で評価されます。

- まず最初の場合が試されます。第2引数がゼロの場合、関数は `n` (最初の引数)を返します。
- そうでなければ、2番目の場合が試されます。最初の引数がゼロの場合、関数は `m` (第2引数)を返します。
- それ以外の場合、関数は最後の行の式を評価して返します。

パターンは値を名前に束縛することができることに注意してください。この例の各行で

は `n` という名前と `m` という名前の両方、またはどちらか一方に、入力された値を束縛しています。これより、入力の引数から名前を選ぶためのさまざまな方法に対応した、さまざまな種類のパターンを見ていくことになります。

## 5.4 単純なパターン

上記のコード例では、2種類のパターンを示しました。

- `Number` 型の値が正確に一致する場合にのみ適合する、数値リテラルパターン
- 引数を名前に束縛する、変数パターン

単純なパターンには他にも種類があります。

- 文字列リテラルと真偽リテラル
- どんな引数とも適合するが名前に束縛はしない、アンダースコア (`_`) で表されるワイルドカードパターン

ここではこれらの単純なパターンを使用した、さらに2つの例を示します。

```
fromString :: String -> Boolean
fromString "true" = true
fromString _      = false

toString :: Boolean -> String
toString true  = "true"
toString false = "false"
```

`psci` でこれらの関数を試してみてください。

## 5.5 ガード

ユークリッドの互除法の例では、`m > n` のときと `m <= n` のときの2つに分岐するために `if .. then .. else` 式を使っていました。こういうときには他にガード(guard)を使うという選択肢もあります。

ガードは真偽値の式で、パターンによる制約に加えてそのガードが満たされたときに、その場合の結果になります。ガードを使用してユークリッドの互除法を書き直すと、次のようになります。

```
gcd :: Number -> Number -> Number
```

```
gcd n 0 = n
gcd 0 n = n
gcd n m | n > m = gcd (n - m) m
gcd n m      = gcd n (m - n)
```

3行目ではガードを使用して、最初の引数が第2引数よりも厳密に大きいという条件を付け加えています。

この例が示すように、ガードは等号の左側に現れ、パイプ文字(|)でパターンのリストと区切られています。

## 演習

1. (簡単)パターン照合を使用して階乗関数を書いてみましょう。ヒント:入力がゼロのときとゼロでないときの2つの場合を考えてみましょう。
2. (やや難しい)二項係数を計算するためのパスカルの公式(Pascal's Rule、パスカルの三角形を参照のこと)について調べてみてください。パスカルの公式を利用し、パターン照合を使って二項係数を計算する関数を記述してください。

## 5.6 配列のパターン照合

パターンを使って配列を照合する方法を見ていきましょう。配列リテラルパターンとConsパターンという2種類のパターンがあります。

### 5.6.1 配列リテラルパターン

配列リテラルパターン(array literal patterns)は、固定長の配列を照合する方法を提供します。たとえば、空の配列であることを特定する関数 `isEmpty` を書きたいとします。最初の選択肢に空の配列パターン(`[]`)を用いるとこれを実現できます。

```
isEmpty :: forall a. [a] -> Boolean
isEmpty [] = true
isEmpty _ = false
```

次の関数では、長さ5の配列と適合し、配列の5つの要素をそれぞれ異なった方法で束縛しています。

```
takeFive :: [Number] -> Number
takeFive [0, 1, a, b, _] = a * b
```



```
takeFive _ = 0
```

最初のパターンは、第1要素と第2要素がそれぞれ0と1であるような、5要素の配列にのみ適合します。その場合、関数は第3要素と第4要素の積を返します。それ以外の場合は、関数は0を返します。 `psci` で試してみると、たとえば次のようになります。

```
> let takeFive [0, 1, a, b, _] = a * b
    takeFive _ = 0

> takeFive [0, 1, 2, 3, 4]
6

> takeFive [1, 2, 3, 4, 5]
0

> takeFive []
0
```

## 5.6.2 Consパターン

Consパターンは空でない配列を照合するのに使います。配列の先頭の要素(head)と、配列から先頭を取り除いた残りの配列(tail)へと、配列を分離する方法を提供します。

Consパターンは、コロン(:)で区切られた、先頭に対応するパターンと残りに対応するパターンによって構成されています。数の配列の、それぞれの要素の平方を合計する関数は次のようになります。

```
sumOfSquares :: [Number] -> Number
sumOfSquares [] = 0
sumOfSquares (n : ns) = n * n + sumOfSquares ns
```

この関数は入力を空の配列と空でない配列の2つの場合に分けて扱っています。配列が空の場合、平方の和はゼロです。そうでない場合は、Consパターンを使用して配列の先頭と残りを分離し、先頭の要素を平方し、残りの平方の和に加算しています。

別の例も見ましょう。次の関数は、数のリスト内のすべての隣接する数の積の合計を求めます。

```
sumOfProducts :: [Number] -> Number
sumOfProducts [] = 0
sumOfProducts [_] = 0
```

```
sumOfProducts (n : m : ns) = n * m + sumOfProducts (m : ns)
```

この関数は入力をゼロ要素、1要素、2要素以上の3つの場合に分けています。最後の場合は、最初の2つの要素を乗算し、残りの部分について再帰します。

## 演習

1. (簡単) 真偽値の配列のすべての要素が `true` に等しいかどうかを決定する関数 `allTrue` を書いてみてください。
2. (やや難しい) 数の配列がソートされているかどうかを調べる関数 `isSorted` を書いてください。

## 5.7 レコードパターンと行多相

レコードパターン(Record patterns)は(ご想像のとおり)レコードにマッチします。

レコードパターンはレコードリテラルに見た目が似ていますが、レコードリテラルでラベルと式をコロンで区切るのとは異なり、レコードパターンではラベルとパターンを等号で区切ります。

たとえば、次のパターンは `first` と `last` と呼ばれるフィールドが含まれた任意のレコードにマッチし、これらのフィールドの値はそれぞれ `x` と `y` という名前に束縛されます。

```
showPerson :: { first :: String, last :: String } -> String
showPerson { first = x, last = y } = y ++ ", " ++ x
```

レコードパターンはPureScriptの型システムの興味深い機能である行多相(row polymorphism)の良い例となっています。上の `showPerson` を型シグネチャなしで定義していたとします。この型はどのように推論されるのでしょうか？面白いことに、推論される型は上で与えた型とは同じではありません。

```
> let showPerson { first = x, last = y } = y ++ ", " ++ x

> :t showPerson
forall r. { first :: String, last :: String | r } -> String
```

この型変数 `r` とは何でしょうか？ `psci` で `showPerson` を使ってみると、面白いことがわかります。

```
> showPerson { first: "Phil", last: "Freeman" }  
"Freeman, Phil"  
  
> showPerson { first: "Phil", last: "Freeman", location: "Los Angeles" }  
"Freeman, Phil"
```

レコードにそれ以外のフィールドが追加されていても、`showPerson` 関数はそのまま動作するのです。型が `String` であるようなフィールド `first` と `last` がレコードに少なくとも含まれていれば、関数適用は正しく型付けされます。しかし、フィールドが不足していると、`showPerson` の呼び出しは不正となります。

```
> showPerson { first: "Phil" }  
  
Object does not have property last
```

`showPerson` の推論された型シグネチャは、`String` であるような `first` と `last` というフィールドと、それ以外の任意のフィールドを持った任意のレコードを引数に取り、`String` を返す、というように読むことができます。

この関数はレコードフィールドの行 `r` について多相的なので、行多相と呼ばれるわけです。

次のように書くことができることにも注意してください。

```
> let showPerson p = p.last ++ ", " ++ p.first
```

この場合も、`psci` は先ほどと同じ型を推論するでしょう。

後ほど拡張可能作用(Extensible effects)について議論するときに、再び行多相について見ていくことになります。

## 5.8 入れ子になったパターン

配列パターンとレコードパターンはどちらも小さなパターンを組み合わせることで大きなパターンを構成しています。これまでの例では配列パターンとレコードパターンの内部に単純なパターンを使用していましたが、パターンが自由に入れ子にすることができることも知っておくのが大切です。入れ子になったパターンを使うと、潜在的に複雑なデータ型に対して関数が条件分岐できるようになります。

たとえば、次のコードでは、レコードパターンと配列パターンを組み合わせ、レコードの配

列と照合させています。

```
type Person = { height :: Number }

totalHeight :: [Person] -> Number
totalHeight [] = 0
totalHeight ({ height = h } : ps) = h + totalHeight ps
```

## 5.9 名前付きパターン

パターンには名前を付けることができ、入れ子になったパターンを使うときにスコープに追加の名前を導入することができます。任意のパターンに名前を付けるには、`@` 記号を使います。

たとえば、次のコードは1つ以上の要素を持つ任意の配列と適合しますが、配列の先頭を `x` という名前、配列全体を `arr` という名前に束縛します。

```
dup :: forall a. [a] -> [a]
dup arr@(x : _) = x : arr
dup [] = []
```

その結果、`dup` は空でない配列の先頭の要素を複製します。

```
> dup [1, 2, 3]
[1, 1, 2, 3]
```

## 演習

1. (簡単)レコードパターンを使って、その人の市町村を探す関数 `getCity` を定義してみましょう。 `Person` は `Address` 型の `address` フィールドを含むレコードとして表現し、 `Address` は `city` フィールドを含まなければなりません。
2. (やや難しい)行多相を考慮すると、 `getCity` 関数の最も一般的な型は何でしょうか？先ほど定義した `totalHeight` 関数についてはどうでしょうか？
3. (やや難しい)パターンと連結演算子( `++` )だけを使って、配列の配列を平坦化する `flatten` 関数を書いてみましょう。ヒント: この関数は `forall a. [[a]] -> [a]` .という型を持っていなければなりません。

## 5.10 Case 式

パターンはソースコードの最上位にある関数だけに現れるわけではありません。`case` 式を使用すると計算の途中の値に対してパターン照合を使うことができます。`case` 式には無名関数に似た種類の便利さがあります。関数に名前を与えることがいつも望ましいわけではありません。パターン照合を使いたいためだけで関数に名前をつけるようなことを避けられるようになります。

例を示しましょう。次の関数は、配列の“longest zero suffix”(和がゼロであるような、最も長い配列の末尾)を計算します。

```
lzs :: [Number] -> [Number]
lzs [] = []
lzs xs@(_ : t) = case sum xs of
  0 -> xs
  _ -> lzs t
```

例えば次のようになります。

```
> lzs [1, 2, 3, 4]
[]

> lzs [1, -1, -2, 3]
[-1, -2, 3]
```

この関数は場合ごとの分析によって動作します。もし配列が空なら、唯一の選択肢は空の配列を返すことです。配列が空でない場合は、さらに2つの場合に分けるためにまず `case` 式を使用します。配列の合計がゼロであれば、配列全体を返します。そうでなければ、配列の残りに対して再帰します。

## 5.11 パターン照合の失敗

`case` 式のパターンを順番に照合していったって、もし選択肢のいずれの場合も入力が適合しなかった時は何が起こるのでしょうか？この場合、パターン照合失敗によって、`case` 式は実行時に失敗します。

簡単な例でこの動作を見てみましょう。

```
patternFailure :: Number -> Number
patternFailure 0 = 0
```

この関数はゼロの入力に対してのみ適合する単一の場合を含みます。このファイルをコンパイルして `psci` でそれ以外の値を与えてテストすると、実行時エラーが発生します。

```
> patternFailure 10
```

```
Failed pattern match
```

どんな入力の組み合わせに対しても値を返すような関数は全関数(total function)と呼ばれ、そうでない関数は部分関数(partial function)と呼ばれています。

一般的には、可能な限り全関数として定義したほうが良いと考えられています。もしその関数が正しい入力に対して値を返さないことがあるとわかっているなら、大抵は `a` に対して型 `Maybe a` の返り値にし、失敗を示すときには `Nothing` を使うようにしたほうがよいでしょう。この方法なら、型安全な方法で値の有無を示すことができます。

戻り値の型として `Maybe Number` を使うよう書き直した `patternFailure` 関数は次のようになります。

```
patternFailure :: Number -> Maybe Number
patternFailure 0 = Just 0
patternFailure _ = Nothing
```

## 5.12 代数的データ型

この節では、PureScriptの型システムでパターン照合に原理的に関係している代数的データ型(Algebraic data type, ADT)と呼ばれる機能を導入します。

しかしまずは、ベクターグラフィックスライブラリの実装というこの章の課題を解決する基礎として、簡単な例を切り口にして考えていきましょう。

直線、矩形、円、テキストなどの単純な図形の種類を表現する型を定義したいとします。オブジェクト指向言語では、おそらくインターフェイスもしくは抽象クラス `Shape` を定義し、使いたいそれぞれの図形について具体的なサブクラスを定義するでしょう。

しかしながら、この方針は大きな欠点をひとつ抱えています。 `Shape` を抽象的に扱うためには、実行したいと思う可能性のあるすべての操作を事前に把握し、 `Shape` インターフェイスに定義する必要があるのです。このため、モジュール性を壊さずに新しい操作を追加することが難しくなります。

もし図形の種類が事前にわかっているなら、代数的データ型はこうした問題を解決する型安全な方法を提供します。モジュール性のある方法で `Shape` に新たな操作を定義し、型安全なまま保守することを可能にします。

代数的データ型として表現された `Shape` がどのように記述されるかを次に示します。

```
data Shape
  = Circle Point Number
  | Rectangle Point Number Number
  | Line Point Point
  | Text Point String
```

次のように `Point` 型を代数的データ型として定義することもできます。

```
data Point = Point
  { x :: Number
  , y :: Number
  }
```

この `Point` データ型は、興味深い点をいくつか示しています。

- 代数的データ型の構築子に格納されるデータは、プリミティブ型に限定されるわけではありません。構築子はレコード、配列、あるいは他の代数的データ型を含めることもできます。
- 代数的データ型は複数の構築子があるデータを記述するのに便利ですが、構築子がひとつだけのときでも便利です。
- 代数的データ型の構築子は、代数的データ型自身と同じ名前の場合もあります。これはごく一般的であり、`Point` データ構築子と `Point` 型構築子を混同しないようにすることが大切です。これらは異なる名前空間にあります。

この宣言ではいくつかの構築子の和として `Shape` を定義しており、各構築子に含まれたデータはそれぞれ区別されます。`Shape` は、中央 `Point` と半径を持つ `Circle` か、`Rectangle`、`Line`、`Text` のいずれかです。他には `Shape` 型の値を構築する方法はありません。

代数的データ型の定義は予約語 `data` から始まり、それに新しい型の名前と任意個の型引数が続きます。その型のデータ構築子は等号の後に定義され、パイプ文字 (`|`) で区切られます。

それではPureScriptの標準ライブラリから別の例を見てみましょう。オプションな値を定義するのに使われる `Maybe` 型を本書の冒頭で扱いました。`purescript-maybe` パッケージで



は `Maybe` を次のように定義しています。

```
data Maybe a = Nothing | Just a
```

この例では型引数 `a` の使用方法を示しています。パイプ文字を「または」と読むことにすると、この定義は「`Maybe a` 型の値は、無い(`Nothing`)、またはただの(`Just`)型 `a` の値だ」と英語のように読むことができます。

データ構築子は再帰的なデータ構造を定義するために使用することもできます。更に例を挙げると、要素が型 `a` の単方向連結リストのデータ型を定義はこのようになります。

```
data List a = Nil | Cons a (List a)
```

この例は `purescript-lists` パッケージから持ってきました。ここで `Nil` 構築子は空のリストを表しており、`Cons` は先頭となる要素と他の配列から空でないリストを作成するために使われます。`Cons` の2つ目のフィールドでデータ型 `List a` を使用しており、再帰的なデータ型になっていることに注目してください。

## 5.13 代数的データ型の使用

代数的データ型の構築子を使用して値を構築するのはとても簡単です。対応する構築子に含まれるデータに応じた引数を用意し、その構築子を単に関数のように適用するだけです。

例えば、上で定義した `Line` 構築子は2つの `Point` を必要としていますので、`Line` 構築子を使って `Shape` を構築するには、型 `Point` のふたつの引数を与えなければなりません。

```
exampleLine :: Shape
exampleLine = Line origin origin
where
  origin :: Point
  origin = Point { x: 0, y: 0 }
```

`origin` を構築するため、レコードを引数として `Point` 構築子を適用しています。

代数的データ型で値を構築することは簡単ですが、これをどうやって使ったらよいのでしょうか？ここで代数的データ型とパターン照合との重要な接点が見えてきます。代数的データ型の値がどの構築子から作られたかを調べたり、代数的データ型からフィールドの値を取り出す唯一の方法は、パターン照合を使用することです。



例を見てみましょう。Shape を String に変換したいとしましょう。Shape を構築するのにどの構築子を使用したかを調べるには、パターン照合を使用しなければなりません。これには次のようにします。

```
showPoint :: Point -> String
showPoint (Point { x = x, y = y }) =
    "(" ++ show x ++ ", " ++ show y ++ ")"

showShape :: Shape -> String
showShape (Circle c r)      = ...
showShape (Rectangle c w h) = ...
showShape (Line start end)  = ...
showShape (Circle loc text) = ...
```

各構築子はパターンとして使用することができ、構築子への引数はそのパターンで束縛することができます。showShape の最初の場合を考えてみましょう。もし Shape が Circle 構築子適合した場合、2つの変数パターン c と r を使って Circle の引数(中心と半径)がスコープに導入されます。その他の場合も同様です。

showPoint は、パターン照合の別の例にもなっています。showPoint はひとつの場合しかありませんが、Point 構築子の中に含まれたレコードのフィールドに適合する、入れ子になったパターンが使われています。

## 演習

1. (簡単) 半径 10 で中心が原点にある円を表す Shape の値を構築してください。
2. (やや難しい) 引数の Shape を原点を中心として2倍に拡大する、Shape から Shape への関数を書いてみましょう。
3. (やや難しい) Shape からテキストを抽出する関数を書いてください。この関数は Maybe String を返さなければならず、もし入力が Text を使用して構築されたものでなければ、戻り値には Nothing 構築子を使ってください。

## 5.14 newtype 宣言

代数的データ型の特別な場合に、newtypeと呼ばれる重要なものがあります。newtypeは予約語 data の代わりに予約語 newtype を使用して導入します。

newtype宣言では過不足なくひとつだけの構築子を定義しなければならず、その構築子は過不足なくひとつだけの引数を取る必要があります。つまり、newtype宣言は既存の型到新

いい名前を与えるものなのです。実際、`newtype`の値は、元の型と同じ実行時表現を持っています。しかし、これらは型システムの観点から区別されます。これは型安全性の追加の層を提供するのです。

例として、ピクセルとインチのような単位を表現するために、`Number` の型レベルの別名を定義したくなる場合があるかもしれません。

```
newtype Pixels = Pixels Number
newtype Inches = Inches Number
```

こうすると `Inches` を期待している関数に `Pixels` 型の値を渡すことは不可能になりますが、実行時の効率に余計な負荷が加わることはありません。

`newtype`は次の章で型クラスを扱う際に重要になります。`newtype`は実行時の表現を変更することなく型に異なる振る舞いを与えることを可能にするからです。

## 5.15 ベクターグラフィックスライブラリ

これまで定義してきたデータ型を使って、ベクターグラフィックスを扱う簡単なライブラリを作成していきましょう。

ただの `Shape` の配列であるような、`Picture` という型同義語を定義しておきます。

```
type Picture = [Shape]
```

デバッグしていると `Picture` を `String` として表示できるようにしたくなることもあるでしょう。これはパターン照合を使用して定義された次の関数で行うことができます。

```
showPicture :: Picture -> String
showPicture picture = "[" ++ go picture ++ "]"
  where
    go :: Picture -> String
    go [] = ""
    go [x] = showShape x
    go (x : xs) = showShape x ++ ", " ++ go xs
```

再帰が `where` ブロックで定義された補助関数を使用して処理されていることに注目してください。この関数 `go` は関数 `showPicture` の内部でのみ参照可能で、モジュールの利用者が参照することはできません。

`go` は、空の配列、1要素の配列、それ以外の、3つの場合を扱います。この方針だと、文字列の末尾に余分なコンマ文字が出力されるのを避けることができます。

試してみましょう。`grunt` でこのモジュールをコンパイルし、`psci` で開きます。

```
$ grunt
$ psci

> :i Data.Picture

> showPicture [Line (Point { x: 0, y: 0 }) (Point { x: 1, y: 1 })]

"[Line [start: (0, 0), end: (1, 1)]]"
```

## 5.16 外接矩形の算出

このモジュールのコード例には、`Picture` の最小外接矩形を計算する関数 `bounds` が含まれています。

`Bounds` は外接矩形を定義するデータ型です。また、構築子をひとつだけ持つ代数的データ型として定義されています。

```
data Bounds = Bounds
  { top    :: Number
  , left   :: Number
  , bottom :: Number
  , right  :: Number
  }
```

`Picture` 内の `Shape` の配列を走査し、最小の外接矩形を累積するため、`bounds` は `Data.Foldable` の `foldl` 関数を使用しています。

```
bounds :: Picture -> Bounds
bounds = foldl combine emptyBounds
  where
    combine :: Bounds -> Shape -> Bounds
    combine b shape = shapeBounds shape \/ b
```

畳み込みの初期値として空の `Picture` の最小外接矩形を求める必要がありますが、`emptyBounds` で定義される空の外接矩形がその条件を満たしています。

累積関数 `combine` は `where` ブロックで定義されています。 `combine` は `fold1` の再帰呼び出しで計算された外接矩形と、配列内の次の `Shape` を引数にとり、ユーザ定義の演算子 `\` を使ってふたつの外接矩形の和を計算しています。 `shapeBounds` 関数は、パターン照合を使用して、単一の図形の外接矩形を計算します。

## 演習

1. (やや難しい) ベクターグラフィックライブラリを拡張し、`Shape` の面積を計算する新しい操作 `area` を追加してください。この演習では、テキストの面積は0であるものとしてください。
2. (難しい) `Shape` を拡張し、新しいデータ構築子 `Clipped` を追加してください。 `Clipped` は他の `Picture` を矩形に切り抜き出します。切り抜かれた `Picture` の境界を計算できるよう、`shapeBounds` 関数を拡張してください。これは `Shape` を再帰的なデータ型にすることに注意してください。

## 5.17 まとめ

この章では、関数型プログラミングから基本だが強力なテクニックであるパターン照合を扱いました。複雑なデータ構造の部分と照合するために、簡単なパターンだけでなく配列パターンやレコードパターンをどのように使用するかを見てきました。

またこの章では、パターン照合に密接に関連する代数的データ型を導入しました。代数的データ型がデータ構造のわかりやすい記述をどのように可能にするか、新たな操作でデータ型を拡張するためのモジュール性のある方法を提供することを見てきました。

最後に、多くの既存のJavaScript関数に型を与えるために、強力な抽象化である行多相を扱いました。この本の後半ではこれらの概念を再び扱います。

本書では今後も代数的データ型とパターン照合を使用するので、今のうちにこれらに習熟しておくで役立つでしょう。これ以外にも独自の代数的データ型を作成し、パターン照合を使用してそれらを使う関数を書くことを試してみてください。

## 6 型クラス

### 6.1 章の目標

この章では、PureScriptの型システムによって可能になる強力な抽象化の形式、型クラスを

導入します。

この章ではデータ構造をハッシュするためのライブラリを題材に説明していきます。データ自身の構造について直接考えることなく複雑なデータ構造をハッシュするために、型クラスの仕組みがどのようにして働くのかを見ていきます。

またPureScriptのPreludeと標準ライブラリに含まれる標準的な型クラスも見ていきます。PureScriptのコードは概念を簡潔に表現するために型クラスの強力さに大きく依存しているので、これらのクラスに慣れておく役に立つでしょう。

## 6.2 プロジェクトの準備

この章のソースコードは、ファイル `src/data/Hashable.purs` で定義されています。

このプロジェクトには以下のBower依存関係があります。

- `purescript-maybe` : オptionalな値を表す `Maybe` データ型が定義されています。
- `purescript-tuples` : 値の組を表す `Tuple` データ型が定義されています。
- `purescript-either` : 非交和を表す `Either` データ型が定義されています。
- `purescript-strings` : 文字列を操作する関数が定義されています。

モジュール `Data.Hashable` は、これらのBowerパッケージによって提供されるモジュールをいくつかインポートします。

```
module Data.Hashable where

import Data.Maybe
import Data.Tuple
import Data.Either
import Data.String
import Data.Function
```

## 6.3 見せてください！

型クラスの最初に扱う例は、すでに何回か見てきた関数に関係します。`show` は何らかの値を取りそれを文字列として表示する関数です。

`show` は `Prelude` モジュールの `Show` と呼ばれる型クラスで次のように定義されています。

```
class Show a where
```

```
show :: a -> String
```

このコードでは、型変数 `a` でパラメータ化された、`Show` という新しい型クラス(type class)を宣言しています。

型クラスインスタンスには、型クラスで定義された関数の、その型に特殊化された実装が含まれています。

例えば、Preludeにある `Boolean` 値に対する `Show` 型クラスインスタンスの定義は次のとおりです。

```
instance showBoolean :: Show Boolean where
  show true  = "true"
  show false = "false"
```

このコードは `showBoolean` という名前の型クラスのインスタンスを宣言します。

PureScriptでは、生成されたJavaScriptの可読性を良くするために型クラスインスタンスには名前をつけます。このとき、`Boolean` 型は `Show` 型クラスに属しているといえます。

`psci` で `Show` 型クラスについて異なる型でいくつかの値を表示してみましょう。

```
> show true

"true"

> show 1.0

"1"

> show "Hello World"

"\\"Hello World\\""
```

この例ではさまざまなプリミティブ型の値を `show` しましたが、もっと複雑な型を持つ値を `show` することもできます。

```
> i Data.Tuple
> show $ Tuple 1 true
"Tuple (1) (true)"

> :i Data.Maybe
> show $ Just "testing"
```

```
"Just (\\"testing\\")"
```

型 `Data.Either` の値を表示しようとする、興味深いエラーメッセージが表示されます。

```
> :i Data.Either
> show $ Left 10

Error in declaration it
No instance found for Prelude.Show (Data.Either.Either Prim.String u8)
```

ここでの問題は `show` しようとしている型に対する `Show` インスタンスが存在しないということではなく、`psci` がこの型を推論できなかったということです。このエラーメッセージで未知の型 `u8` と表示されているのがそれです。

`::` 演算子を使って式に対して型注釈を加えると、`psci` が正しい型クラスインスタンスを選ぶことができるようになります。

```
> show (Left 10 :: Either Number String)

"Left (10)"
```

`Show` インスタンスをまったく持っていない型もあります。関数の型 `->` がその一例です。`Number` から `Number` への関数を `show` しようすると、型検証器によってその通りのエラーメッセージが表示されます。

```
> show $ \n -> n + 1

Error in declaration it
No instance found for Prelude.Show (Prim.Number -> Prim.Number)
```

## 演習

1. (簡単)前章の `showShape` 関数を使って、`Shape` 型に対しての `Show` インスタンスを定義してみましょう。

## 6.4 標準的な型クラス

この節では、`Prelude`や標準ライブラリで定義されている標準的な型クラスをいくつか見ていきましょう。これらの型クラスはPureScript特有の抽象化の基礎としてあちこちで使われてい

るので、これらの関数の基本についてよく理解しておくことを強くお勧めします。

## 6.4.1 Eq 型クラス

Eq 型クラスは等値演算子( `==` )と不等値演算子( `/=` )を定義します。

```
class Eq a where
    (==) :: a -> a -> Boolean
    (/=) :: a -> a -> Boolean
```

異なる型の2つの値を比較しても意味がありませんから、いずれの演算子も2つの引数が同じ型を持つ必要があることに注意してください。

`psci` で Eq 型クラスを試してみましょう。

```
> 1 == 2
false

> "Test" == "Test"
true
```

## 6.4.2 Ord 型クラス

Ord 型クラスは順序付け可能な型に対して2つの値を比較する `compare` 関数を定義します。 `compare` 関数が定義されていると、比較演算子 `<`、`>` と、その仲間 `<=`、`>=` も定義されます。

```
data Ordering = LT | EQ | GT

class (Eq a) <= Ord a where
    compare :: a -> a -> Ordering
```

`compare` 関数は2つの値を比較して `Ordering` の3つの値のうちいずれかを返します。

- `LT` – 最初の引数が2番目の値より小さいとき
- `EQ` – 最初の引数が2番目の値と等しい(または比較できない)とき
- `GT` – 最初の引数が2番目の値より大きいとき

`compare` 関数についても `psci` で試してみましょう。



```
> compare 1 2
LT

> compare "A" "Z"
LT
```

### 6.4.3 Num 型クラス

`Num` 型クラスは加算、減算、乗算、除算などの数値演算子を使用可能な型を示します。必要に応じて再利用できるように、これらの演算子を抽象化するわけです。

注意: 関数呼び出しが型クラスの実装に基いて呼び出されるのとは対照的に、型クラス `Eq` や `Ord`、`Num` などはPureScriptでは特別に扱われ、`1 + 2 * 3` のような単純な式は単純なJavaScriptへと変換されます。

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  (/) :: a -> a -> a
  (%) :: a -> a -> a
  negate :: a -> a
```

### 6.4.4 半群とモノイド

`Semigroup` (半群)型クラスは、連結演算子 `<>` を提供する型を示します。

```
class Semigroup a where
  (<>) :: a -> a -> a
```

普通の文字列連結について文字列は半群をなしますし、同様に配列も半群をなします。その他の標準的なインスタンスの幾つかは、`purescript-monoid` パッケージで提供されています。

以前に見た `++` 連結演算子は、`<>` の別名として提供されています。

`purescript-monoid` パッケージで提供されている `Monoid` 型クラスは、`mempty` と呼ばれる空の値の概念で `Semigroup` 型クラスを拡張します。

```
class (Semigroup m) => Monoid m where
```

```
mempty :: m
```

文字列や配列はモノイドの簡単な例になっています。

`Monoid` 型クラスインスタンスでは、「空」の値から始めて新たな値を合成していき、その型で累積した結果を返すにはどうするかを記述する型クラスです。例えば、畳み込みを使っていくつかのモノイドの値の配列を連結する関数を書くことができます。`psci` で試すと次のようになります。

```
> :i Data.Monoid
> :i Data.Foldable

> foldl (<>) mempty ["Hello", " ", "World"]
"Hello World"

> foldl (<>) mempty [[1, 2, 3], [4, 5], [6]]
[1,2,3,4,5,6]
```

`purescript-monoid` パッケージにはモノイドと半群の多くの例を提供しており、これらを本書で扱っていきます。

## 6.4.5 Foldable 型クラス

`Monoid` 型クラスは畳み込みの結果になるような型を示しますが、`Foldable` 型クラスは、畳み込みの元のデータとして使えるような型構築子を示しています。

また、`Foldable` 型クラスは、配列や `Maybe` などのいくつかの標準的なコンテナのインスタンスを含む `purescript-foldable-traversable` パッケージで提供されています。

`Foldable` クラスに属する関数の型シグネチャは、これまで見てきたものよりも少し複雑です。

```
class Foldable f where
  foldr :: forall a b. (a -> b -> b) -> b -> f a -> b
  foldl :: forall a b. (b -> a -> b) -> b -> f a -> b
  foldMap :: forall a m. (Monoid m) => (a -> m) -> f a -> m
```

この定義は `f` を配列の型構築子だと特殊化して考えてみるとわかりやすくなります。この場合、すべての `a` について `f a` を `[a]` に置き換える事ができますが、`foldl` と `foldr` の型が、最初に見た配列に対する畳み込みの型になるとわかります。

`foldMap` についてはどうでしょうか？これは `forall m. (Monoid m) => (a -> m) -> [a] -> m` になります。この型シグネチャは、型 `m` が `Monoid` 型クラスのインスタンスであればどんな型でも返り値の型として選ぶことができると言っています。配列の要素をそのモノイドの値へと変換する関数を提供すれば、そのモノイドの構造を利用して配列を畳み込み、ひとつの値にして返すことができます。

それでは `psci` で `foldMap` を試してみましょう。

```
> :i Data.Foldable

> foldMap show [1, 2, 3, 4, 5]
"12345"
```

ここではモノイドとして文字列を選び、`Number` を文字列として表示する `show` 関数を使いました。それから、数の配列を渡し、それぞれの数を `show` してひとつの文字列へと連結した結果出力されました。

畳み込み可能な型は配列だけではありません。`purescript-foldable-traversable` では `Maybe` や `Tuple` のような型の `Foldable` インスタンスが定義されており、`purescript-lists` のような他のライブラリでは、そのライブラリのそれぞれのデータ型に対して `Foldable` インスタンスが定義されています。`Foldable` は順序付きコンテナ(ordered container)の概念を抽象化するのです。

## 6.4.6 関手と型クラス則

PureScriptで副作用を伴う関数型プログラミングのスタイルを可能にするための `Functor` と `Applicative`、`Monad` といった型クラスがPreludeでは定義されています。これらの抽象については本書で後ほど扱いますが、まずは「持ち上げ演算子」`<$>` の形ですで見えてきた `Functor` 型クラスの定義を見てみましょう。

```
class Functor f where
  (<$>) :: forall a b. (a -> b) -> f a -> f b
```

演算子 `<$>` は関数をそのデータ構造まで「持ち上げる」(lift)ことができます。ここで「持ち上げ」という言葉の具体的な定義は問題のデータ構造に依りますが、すでにいくつかの単純な型についてその動作を見てきました。

```
> :i Data.Array

> (\n -> n < 3) <$> [1, 2, 3, 4, 5]
```

```
[true, true, false, false, false]

> :i Data.Maybe
> Data.String.length <$> Just "testing"

Just (7)
```

<\$> 演算子は様々な構造の上でそれぞれ異なる振る舞いをしますが、<\$> 演算子の意味はどのように理解すればいいのでしょうか。

直感的には、<\$> 演算子はコンテナのそれぞれの要素へ関数を適用し、その結果から元のデータと同じ形状を持った新しいコンテナを構築するのだというように理解することができます。しかし、この概念を厳密にするにはどうしたらいいのでしょうか？。

**Functor** の型クラスのインスタンスは、関手則(functor laws)と呼ばれる法則を順守するものと期待されています。

- `id <$> xs = xs`
- `g <$> (f <$> xs) = (g <<< f) <$> xs`

最初の法則は恒等射律(identity law)です。これは、恒等関数をその構造まで持ち上げると、元の構造をそのまま返す恒等射になるということとっています。恒等関数は入力を変更しませんから、これは理にかなっています。

第二の法則は合成律(composition law)です。構造をひとつの関数で写してから2つめの関数で写すのは、2つの関数の合成で構造を写すのと同じだ、とっています。

「持ち上げ」の一般的な意味が何であれ、データ構造に対する持ち上げ関数の正しい定義はこれらの法則に従っていなければなりません。

標準の型クラスの多くには、このような法則が付随しています。一般に、型クラスに与えられた法則は、型クラスの関数に構造を与え、インスタンスについて調べられるようにします。興味のある読者は、すでに見てきた標準の型クラスに属する法則について調べてみてもよいでしょう。

## 演習

1. (簡単)次のnewtypeは複素数を表します。

```
newtype Complex = Complex
  { real :: Number
  , imaginary :: Number
  }
```

`Complex` について、`Show` と `Eq` のインスタンスを定義してください。

2. (やや難しい)次は型 `a` の要素の空でない配列の型を定義しています。

```
data NonEmpty a = NonEmpty a [a]
```

`[]` についての `Semigroup` インスタンスを再利用し、空でない配列についての `Semigroup` インスタンスを書いてみてください。

3. (やや難しい) `NonEmpty` の `Functor` インスタンスを書いてみましょう。
4. (難しい) `NonEmpty` の `Foldable` インスタンスを書いてみましょう。ヒント:配列の `Foldable` インスタンスを再利用してみましょう。

## 6.5 型注釈

型クラスを使うと、関数の型に制約を加えることができます。例を示しましょう。 `Eq` 型クラスのインスタンスで定義された等値性を使って、3つの値が等しいかどうかを調べる関数を書きたいとします。

```
threeAreEqual :: forall a. (Eq a) => a -> a -> a -> Boolean
threeAreEqual a1 a2 a3 = a1 == a2 && a2 == a3
```

この型宣言は `forall` を使って定義された通常が多相型のようにも見えます。しかし、太い矢印 `=>` で型の残りの部分から区切られた、括弧内の型クラス制約があります。

インポートされたモジュールのどれかに `a` に対する `Eq` インスタンスが存在するなら、どんな型 `a` を選んでも `threeAreEqual` を呼び出すことができる、とこの型は言っています。

制約された型には複数の型クラスインスタンスを含めることができますし、インスタンスの型は単純な型変数に限定されません。 `Ord` と `Show` のインスタンスを使って2つの値を比較する例を次に示します。

```
showCompare :: forall a. (Ord a, Show a) => a -> a -> String
showCompare a1 a2 | a1 < a2 =
    show a1 ++ " is less than " ++ show a2
showCompare a1 a2 | a1 > a2 =
    show a1 ++ " is greater than " ++ show a2
showCompare a1 a2 =
```

```
show a1 ++ " is equal to " ++ show a2
```

型クラスで制約された関数を使うときには重要な制限があります。PureScriptコンパイラは制約された型を推論しません。型注釈の提供は必須になります。

`psci` で `Num` のような標準の型クラスのいずれかを使って、このことを試してみましょう。

```
> :t \x -> x + x
```

```
Error in declaration it
No instance found for Prelude.Num u2
```

ここで、数を倍にするこの関数の型を、数の型の `Num` インスタンスを使って見つけようとしませんが、`psci` は `x` の型が不明であるときはこの関数の制約された型を推論しないので、`psci` は未知の型の型クラスインスタンスを見つけることができないという旨を報告します。

たとえば `x` が数であることを表すには、次のように型検証器に指示しなければいけません。

```
> :t \x -> x + (x :: Number)
```

```
Prim.Number -> Prim.Number
```

## 6.6 インスタンスの重複

PureScriptには型クラスのインスタンスに関する重複インスタンス規則(Overlapping instances rule)という規則があります。型クラスのインスタンスが関数呼び出しのところで必要とされるときはいつでも、PureScriptは正しいインスタンスを選択するために型検証器によって推論された情報を使用します。そのとき、その型の適切なインスタンスがちょうどひとつだけ存在しなければなりません。

これを実証するために、適当な型に対して2つの異なる型クラスのインスタンスを作成してみましょう。次のコードでは、型 `T` の2つの重複する `Show` インスタンスを作成しています。

```
module Overlapped where

data T = T

instance showT1 :: Show T where
  show _ = "Instance 1"
```

```
instance showT2 :: Show T where
  show _ = "Instance 2"
```

このモジュールはエラーなくコンパイルされます。 `psci` を起動し、型 `T` の `Show` インスタンスを見つけようとする、重複インスタンス規則が適用され、エラーになります。

```
> show T
```

```
Compiling Overlapped
Error in declaration it
Overlapping instances found for Prelude.Show Overlapped.T
```

重複インスタンスルールが適用されるのは、型クラスのインスタンスの自動選択が予測可能な処理であるようにするためです。もし型に対してふたつの型クラスインスタンスを許し、モジュールインポートの順序に従ってどちらかを選ぶようにすると、実行時のプログラムの振る舞いが予測できなくなってしまう好ましくありません。

適切な法則を満たすふたつ妥当な型クラスインスタンスが存在するのなら、既存の型を包む `newtype` を定義するのが一般的な方法です。重複インスタンスのルールの下でも、異なる `newtypes` なら異なる型クラスインスタンスを持つことが許されるので、問題はなくなります。この手法は `PureScript` の標準ライブラリでも使われており、例えば `purescript-monoid` では、`Maybe a` 型は `Monoid` 型クラスの妥当なインスタンスを複数持っています。

## 6.7 インスタンスの依存関係

制約された型を使うと関数の実装が型クラスインスタンスに依存できるように、型クラスインスタンスの実装は他の型クラスインスタンスに依存することができます。これにより、型を使ってプログラムの実装を推論するという、プログラム推論の強力な形式を提供します。

`Show` 型クラスを例に考えてみましょう。要素を `show` する方法があるとき、その要素の配列を `show` する型クラスインスタンスを書くことができます。

```
instance showArray :: (Show a) => Show [a] where
  show xs = "[" ++ go xs ++ "]"
  where
    go [] = ""
    go [x] = show x
    go (x : xs) = show x ++ ", " ++ go xs
```



PureScriptのPreludeに、このコードの最適化されたものが含まれています。

ここで、関数 `show` は様々な型の入力に対して使われていることに注意してください。`[a]` つまり要素が型 `a` の配列の入力に対して動作するように `show` を定義しています。しかし、`go` 関数では、入力の先頭の要素を名前 `x` として導入し、`show x` というように呼び出しています。ここでの `show` は型 `a` の要素に適用されています。

プログラムがコンパイルされると、`Show` の正しい型クラスのインスタンスは `show` の引数の推論された型に基づいて選ばれますが、このあたりの複雑さに開発者が関与することはありません。

## 演習

1. (簡単) `Eq a` と `Eq [a]` のインスタンスを再利用して、型 `NonEmpty a` に対する `Eq` インスタンスを書いてみましょう。
2. (やや難しい) `Ord` のインスタンスを持つ任意の型 `a` について、その他のどんな値よりも大きい「無限大」の値を新たに追加することができます。

```
data Extended a = Finite a | Infinite
```

`a` の `Ord` インスタンスを再利用して、`Extended a` の `Ord` インスタンスを書いてみましょう。

3. (難しい) 順序付きコンテナを定義する(そして `Foldable` のインスタンスを持っている)ような型構築子 `f` が与えられたとき、追加の要素を先頭に含めるような新たなコンテナ型を作ることができます。

```
data OneMore f a = OneMore a (f a)
```

このコンテナ `OneMore f` もまた順序を持っています。ここで、新しい要素は任意の `f` の要素よりも前にきます。この `OneMore f` の `Foldable` インスタンスを書いてみましょう。

```
instance foldableOneMore :: (Foldable f) => Foldable (OneMore f) where
    ...
```

## 6.8 多変数型クラス



型クラスは必ずしもひとつの型だけを型変数としてとるわけではありません。型変数がひとつだけなのが最も一般的ですが、実際には型クラスはゼロ個以上の型変数を持つことができます。

それでは2つの型引数を持つ型クラスの例を見てみましょう。

```
module Stream where

import Data.Maybe
import Data.Tuple
import Data.String

class Stream list element where
    uncons :: list -> Maybe (Tuple element list)

instance streamArray :: Stream [a] a where
    uncons [] = Nothing
    uncons (x : xs) = Just (Tuple x xs)

instance streamString :: Stream String String where
    uncons "" = Nothing
    uncons s = Just (Tuple (take 1 s) (drop 1 s))
```

この `Stream` モジュールでは、`uncons` 関数を使ってストリームの先頭から要素を取り出すことができる、要素のストリームのような型を示すクラス `Stream` が定義されています。

`Stream` 型クラスは、ストリーム自身の型だけでなくその要素の型も型変数として持っていることに注意してください。これによって、ストリームの型が同じでも要素の型について異なる型クラスインスタンスを定義することができます。

このモジュールでは、`uncons` がパターン照合で配列の先頭の要素を取り除くような配列のインスタンスと、文字列から最初の文字を取り除くような文字列のインスタンスという、2つの型クラスインスタンスが定義されています。

任意のストリーム上で動作する関数を記述することができます。例えば、ストリームの要素に基づいたモノイドで結果を累積する関数は次のようになります。

```
import Data.Monoid

foldStream :: forall l e m. (Stream l e, Monoid m) => (e -> m) -> l -> m
foldStream f list =
    case uncons list of
        Nothing -> mempty
        Just (Tuple head tail) -> f head <> foldStream f tail
```

`psci` で使って、異なる `Stream` の型や異なる `Monoid` の型について `foldStream` を呼び出してみましょう。

## 6.9 型変数のない型クラス

ゼロ個の型変数を持つ型クラスを定義することもできます！型システムによってコードの大域的な性質の追跡ができるようになる、関数についてのコンパイル時表明に関係しています。

たとえば、型システムを使って部分関数の使用を追跡したいとしましょう。型引数のない型クラス `Partial` を定義し、すべての部分関数を `Partial` 制約で注釈します。

```
module Partial where

class Partial

head :: forall a. (Partial) => [a] -> a
head (x : _) = x

tail :: forall a. (Partial) => [a] -> [a]
tail (_ : xs) = xs
```

`Partial` モジュールの `Partial` 型クラスのインスタンスを定義していないことに注意してください。こうすると目的を達成できます。このままの定義では `head` 関数を使用しようとすると型エラーになるのです。

```
> Partial.head [1, 2, 3]

Error in declaration it
No instance found for Partial.Partial
```

このライブラリを使うには、2つの選択肢があります。

- このモジュールで `Partial` 型クラスのインスタンスを定義し、そのモジュールの関数が部分的であることを了承したと表明します。

```
module Main where

import Partial
```

```
instance partial :: Partial
```

- あるいは、これらの部分関数を利用するすべての関数で `Partial` 制約を再発行する方法もあります。

```
secondElement :: forall a. (Partial) => [a] -> a  
secondElement xs = head (tail xs)
```

## 6.10 上位クラス

インスタンスを別のインスタンスに依存させることによって型クラスのインスタンス間の関係を表現することができるように、いわゆる上位クラス(superclass)を使って型クラス間の関係を表現することができます。

あるクラスのどんなインスタンスも、その他のあるクラスのインスタンスで必要とされているとき、前者の型クラスは後者の型クラスの上位クラスであるといい、クラス定義で逆向きの太い矢印を使い上位クラス関係を示します。

すでに上位クラスの関係の一例について見ています。`Eq` クラスは `Ord` の上位クラスです。`Ord` クラスのすべての型クラスインスタンスについて、その同じ型に対応する `Eq` インスタンスが存在しなければなりません。`compare` 関数が2つの値が比較できないと報告した時は、それらが実は同値であるかどうかを決定するために `Eq` クラスを使いたくなることが多いでしょうから、これは理にかなっています。

一般に、下位クラスの法則が上位クラスのメンバに言及しているとき、上位クラス関係を定義するのは理にかなっています。例えば、`Ord` と `Eq` のインスタンスのどんな組についても、もしふたつの値が `Eq` インスタンスのもとで同値であるなら、`compare` 関数は `EQ` を返すはずだとみなすのは妥当です。言い換えれば、`a == b` ならば `compare a b == EQ` です。法則の階層上のこの関係は、`Eq` と `Ord` の間の上位クラス関係を説明します。

この場合に上位クラス関係を定義する別の考え方としては、この2つのクラスの間には明らかに“is-a”の関係があることです。下位クラスのすべてのメンバは、上位クラスのメンバでもあるということです。

### 演習

1. (やや難しい) 次の `Action` クラスは、ある型の動作(action)を定義する、多変数型クラスです。

```
class (Monoid m) <= Action m a where
  act :: m -> a -> a
```

actはモノイドがどうやって他の型の値を変更するのに使われるのかを説明する関数です。この動作がモノイドの連結演算子に従っていると期待しましょう。例えば、乗算を持つ自然数のモノイドは、文字列の何度かの繰り返しとして文字列に対して動作します。

```
instance repeatAction :: Action Number String where
  act 0 _ = ""
  act n s = s ++ act (n - 1) s
```

Action クラスが Monoid クラスとどのように連携するかを説明する、妥当な法則を書いてみましょう。

2. (やや難しい) インスタンス `Action m a => Action m [a]` を書いてみましょう。ここで、配列上の動作は要素の順序で実行されるように定義されるものとします。
3. (難しい) 以下のnewtypeが与えられたとき、`Action m (Self m)` のインスタンスを書いてみましょう。ここで、モノイド `m` は連結によって自身に作用するものとします。

```
newtype Self m = Self m
```

4. (やや難しい) 引数のない型クラス `Unsafe` を定義し、型安全性を欠いていることを表現する制約としてそれを使い、`Prelude.Unsafe` モジュールから `unsafeIndex` 関数の別バージョンを定義してみましょう。また、その関数を使って、`Unsafe` 制約を失わないように、配列の最後の要素を選択する `last` を定義してみましょう。

## 6.11 ハッシュの型クラス

この最後の節では、章の残りを費やしてデータ構造をハッシュするライブラリを作ります。

このライブラリの目的は説明だけであり、堅牢なハッシングの仕組みの提供を目的としていないことに注意してください。

ハッシュ関数に期待される性質とはどのようなものでしょうか？

- ハッシュ関数は決定的でなくてはなりません。つまり、同じ値には同じハッシュ値を対応させなければなりません
- ハッシュ関数はいろいろなハッシュ値の集合で結果が一様に分布しなければなりません。

最初の性質はまさに型クラスの法則のように見える一方で、2番目の性質はもっとぼんやりとした規約に従っていて、PureScriptの型システムによって確実に強制できるようなものではなさそうです。しかし、これは型クラスについて次のような直感的理解を与えるはずです。

```
type HashCode = Number

class (Eq a) <= Hashable a where
    hash :: a -> HashCode
```

これに、`a == b` ならば `hash a == hash b` という関係性の法則が付随しています。

この節の残りの部分を費やして、`Hashable` 型クラスに関連付けられているインスタンスと関数のライブラリを構築していきます。

決定的な方法でハッシュ値を結合する方法が必要になります。2つのハッシュ値を混ぜて結果を0-65535の間に分布させる次のような関数がある要求を満たしています。

```
(<#>) :: HashCode -> HashCode -> HashCode
(<#>) h1 h2 = (73 * h1 + 51 * h2) % 65536
```

この演算子を使うと、2つのハッシュ値 `h1`、`h2` を、`h1 <#> h2` .というように中置で連結することができます。

それでは、入力の種類を制限する `Hashable` 制約を使う関数を書いてみましょう。ハッシュ関数を必要とするよくある目的としては、2つの値が同じハッシュ値にハッシュされるかどうかを決定することです。 `hashEqual` 関係はそのような機能を提供します。

```
hashEqual :: forall a. (Hashable a) => a -> a -> Boolean
hashEqual = (==) `on` hash
```

この関数はハッシュ同値性を定義するために `Data.Function` の `on` 関数を使っていますが、このハッシュ同値性の定義は『それぞれの値が `hash` 関数に渡されたあとで2つの値が等しいなら、それらの値は「ハッシュ同値」である』というように宣言的に読めるはずです。

プリミティブ型の `Hashable` インスタンスをいくつか書いてみましょう。まずは文字列のインス

タンスです。 `Data.String` モジュールにある `length`、`charCodeAt` という名前の関数を使います。以下の `Hashable` インスタンスは、累積されたハッシュ値と文字コードを `<#>` 演算子を使って連結するという動作を、文字列中の文字に対して反復します。

```
instance hashString :: Hashable String where
    hash s = go 0 0
    where
        go :: Number -> HashCode -> HashCode
        go i acc | i >= length s = acc
        go i acc = go (i + 1) acc <#> charCodeAt i s
```

`Number` 型のインスタンスについてはどうでしょうか。小数や無限大が出現するせいで JavaScript の `Number` 型を扱うといろいろな問題が生じますが、ここでは説明が目的なので、単純に `show` で計算された数値の文字列表現をハッシュすることにします。

```
instance hashNumber :: Hashable Number where
    hash n = hash (show n)
```

`Number` の型クラスインスタンスは必ず `String` の型クラスインスタンスを使うことに注意してください。

`Boolean` 型のインスタンスではさらに簡単です。単に型の2つの値に2つのハッシュ値を静的に割り当てただけです。

```
instance hashBoolean :: Hashable Boolean where
    hash false = 0
    hash true  = 1
```

これらの `Hashable` インスタンスが先ほどの型クラスの法則を満たしていることを証明するにはどうしたらいいでしょうか。同じ値が等しいハッシュ値を持っていることを確認する必要があります。 `String` と `Boolean` の場合は、`Eq` の意味では同じ値でも厳密には同じではない、というような文字列や真偽値は存在しないので簡単です。

数値の場合は同じ数が同じ文字列表現を持っていることを簡単に説明しておかなければなりませんが、そうすれば後は文字列に対してすでに与えられた証明に従うことができます。

もっと面白い型についてはどうでしょうか。 `<#>` 使って入力配列の要素のハッシュ値を組み合わせた、配列の `Hashable` インスタンスは、次のようになります。

```
instance hashArray :: (Hashable a) => Hashable [a] where
    hash [] = 0
    hash (x : xs) = hash x <#> hash xs
```

この場合、配列の長さに関する帰納を使うと、型クラスの法則を証明することができます。長さゼロの唯一の配列は `[]` です。配列の `Eq` の定義により、任意の二つの空でない配列は、それらの先頭の要素が同じで配列の残りの部分が等しいとき、その時に限り等しくなります。この帰納的な仮定により、配列の残りの部分は同じハッシュ値を持ちますし、もし `Hashable a` インスタンスがこの法則を満たすなら、先頭の要素も同じハッシュ値をもつことがわかります。したがって、2つの配列は同じハッシュ値を持ち、`Hashable [a]` も同様に型クラス法則を満たしています。

この章のソースコードには、`Maybe` と `Tuple` 型のインスタンスなど、他にも `Hashable` インスタンスの例が含まれています。

## 演習

1. (簡単) `psci` を使って、各インスタンスのハッシュ関数をテストしてください。
2. (やや難しい) 同値性の近似として `hashEqual` 関数のハッシュ同値性を使い、配列が重複する要素を持っているかどうかを調べる関数を書いてください。ハッシュ値が一致したペアが見つかった場合は、`==` を使って値の同値性を厳密に検証することを忘れないようにしてください。
3. (やや難しい) 型クラスの法則を満たす、次のnewtypeの `Hashable` インスタンスを書いてください。

```
newtype Uniform = Uniform Number
```

```
instance eqUniform :: Eq Uniform where
    (==) (Uniform u1) (Uniform u2) = u1 % 1.0 == u2 % 1.0
    (/=) (Uniform u1) (Uniform u2) = u1 % 1.0 /= u2 % 1.0
```

newtypeの `Uniform` とその `Eq` インスタンスは、同じ小数部分を持っているかの同値関係を持つ数の型を表しています。そのインスタンスが型クラスの法則を満たしていることを証明してください。

4. (難しい) `Maybe`、`Either`、`Tuple` の `Hashable` インスタンスが型クラスの法則を満たしていることを証明してください。

## 6.12 まとめ

---

この章では、型に基づく抽象化で、コードの再利用のための強力な形式化を可能にする型クラスを導入しました。PureScriptの標準ライブラリから標準の型クラスを幾つか見てきました。また、ハッシュ値を計算する型クラスに基づく独自のライブラリを定義しました。

この章では型クラス法則の考え方を導入するとともに、抽象化のための型クラスを使うコードについて、その性質を証明する手法を導入しました。型クラス法則は等式推論(equational reasoning)と呼ばれる大きな分野の一部であり、プログラミング言語の性質と型システムはプログラムについて論理的な推論をできるようにするために使われています。これは重要な考え方で、本書では今後あらゆる箇所で立ち返る話題となるでしょう。

## 7 Applicativeによる検証

---

### 7.1 この章の目標

---

この章では、**Applicative** 型クラスによって表現されるApplicative関手(applicative functor)という重要な抽象化と新たに出会うことになります。名前が難しそうに思えても心配しないでください。フォームデータの検証という実用的な例を使ってこの概念を説明していきます。Applicative関手を使うと、大量の決まり文句を伴うような入力項目の内容を検証するためのコードを、簡潔で宣言的な記述へと変えることができるようになります。

また、Traversable関手(traversable functor)を表現する **Traversable** という別の型クラスにも出会います。現実の問題への解決策からこの概念が自然に生じるということがわかるでしょう。

この章では第3章から引き続き電話帳を例として扱います。今回は電話番号だけでなく住所を含む住所録のデータ型を定義し、これらの型の値を検証する関数を書きます。これらの関数は、例えばデータ入力フォームの一部で、使用者へエラーを表示するウェブユーザーインターフェイスで使われると考えてください。

### 7.2 プロジェクトの準備

---

この章のソースコード

は、`src/Data/AddressBook.purs` と `src/Data/AddressBook/Validation.purs` というファイルで定義されています。



このプロジェクトは多くのBower依存関係を持っていますが、その大半はすでに見てきたものです。新しい依存関係は2つです。

- `purescript-control` - `Applicative` のような型クラスを使用して制御フローを抽象化する関数が定義されています
- `purescript-validation` - この章の主題である `Applicative` による検証のための関手が定義されています。

`Data.AddressBook` モジュールには、このプロジェクトのデータ型とそれらの型に対する `Show` インスタンスが定義されており、`Data.AddressBook.Validation` モジュールにはそれらの型の検証規則含まれています。

## 7.3 関数適用の一般化

`Applicative` 関手の概念を理解するために、まずは以前に見た型構築子 `Maybe` について考えてみましょう。

このモジュールのソースコードでは、次のような型を持つ `address` 関数が定義されています。

```
address :: String -> String -> String -> Address
```

この関数は、通りの名前、市、州という3つの文字列から型 `Address` の値を構築するために使います。

この関数は簡単に適用できますので、`psci` でどうなるか見てみましょう。

```
> :i Data.AddressBook

> address "123 Fake St." "Faketown" "CA"
Address { street: "123 Fake St.", city: "Faketown", state: "CA" }
```

しかし、通り、市、州の3つすべてが必ずしも入力されないものとする、3つの場合それぞれで省略可能である値を示すために `Maybe` 型を使用したくなります。

考えられる場合のひとつとしては、市が省略されている場合があるかもしれません。もし `address` 関数を直接適用しようとする、型検証器からエラーが表示されます。

```
> :i Data.Maybe
> address (Just "123 Fake St.") Nothing (Just "CA")
```

```
Cannot unify Data.Maybe.Maybe u2 with Prim.String.
```

`address` は型 `Maybe String` ではなく文字列型の引数を取るのも、もちろんこれは予想どおりの型エラーです。

しかし、`Maybe` 型で示される省略可能な値を扱うために `address` 関数を「持ち上げ」ることができるはずだと期待することは理にかなっていません。実際、`Control.Apply` で提供されている関数 `lift3` がまさに求めているものです。

```
> :i Control.Apply
> lift3 address (Just "123 Fake St.") Nothing (Just "CA")

Nothing
```

このとき、引数のひとつ(市)が欠落していたので、結果は、`Nothing` です。もし3つの引数すべてが `Just` 構築子を使って与えられると、結果は値を含むことになります。

```
> lift3 address (Just "123 Fake St.") (Just "Faketown") (Just "CA")

Just (Address { street: "123 Fake St.", city: "Faketown", state: "CA" })
```

`lift3` という関数の名前は、3引数の関数を持ち上げるために使用できることを示しています。引数の数が異なる関数を持ち上げる同様の関数が `Control.Apply` で定義されています。

## 7.4 任意個の引数を持つ関数の持ち上げ

これで、`lift2` や `lift3` のような関数を使えば、引数が2個や3個の関数を持ち上げることができるのはわかりました。でも、これを任意個の引数の関数へと一般化することはできるのでしょうか。

`lift3` の型を見てみるとわかりやすいでしょう。

```
> :t Control.Apply.lift3
forall a b c d f. (Prelude.Apply f) => (a -> b -> c -> d) -> f a -> f b -> f
```

上の `Maybe` の例では型構築子 `f` は `Maybe` ですから、`lift3` は次のように特殊化されます。

```
forall a b c d. (a -> b -> c -> d) -> Maybe a -> Maybe b -> Maybe c -> Maybe d
```

この型が言っているのは、3引数の任意の関数を取り、その関数を引数と返り値が `Maybe` で包まれた新しい関数へと持ち上げる、ということです。

もちろんどんな型構築子 `f` についても持ち上げができるわけではないのですが、それでは `Maybe` 型を持ち上げができるようにしているのは何なのでしょう。さて、先ほどの型の特殊化では、`f` に対する型クラス制約から `Apply` 型クラスを取り除いていました。`Apply` は `Prelude` で次のように定義されています。

```
class Functor f where
  (<$>) :: forall a b. (a -> b) -> f a -> f b

class (Functor f) <= Apply f where
  (<*>) :: forall a b. f (a -> b) -> f a -> f b
```

`Apply` 型クラスは `Functor` の下位クラスであり、`<$>` とよく似た型を持つ追加の関数 `<*>` が定義されています。`<$>` と `<*>` の違いは、`<$>` がただの関数を引数に取るのに対し、`<*>` の最初の引数は型構築子 `f` で包まれているという点です。これをどのように使うのかはこれからすぐに見ていきますが、その前にまず `Maybe` 型について `Apply` 型クラスをどう実装するのかを見ていきましょう。

```
instance functorMaybe :: Functor Maybe where
  (<$>) f (Just a) = Just (f a)
  (<$>) f Nothing  = Nothing

instance applyMaybe :: Apply Maybe where
  (<*>) (Just f) (Just x) = Just (f x)
  (<*>) _             _   = Nothing
```

この型クラスのインスタンスが言っているのは、任意のオプションな値にオプションな関数を適用することができ、その両方が定義されている時に限り結果も定義される、ということです。

それでは、`<$>` と `<*>` を一緒に使ってどうやって引数が任意個の関数を持ち上げるのかを見ていきましょう。

1引数の関数については、`<$>` をそのまま使うだけです。

2引数の関数についても考えてみます。型 `a -> b -> c` を持つカーリー化された関数 `f` があるとしましょう。これは型 `a -> (b -> c)` と同じですから、`<$>` を `f` に適用すると型 `f a ->`

`f (b -> c)` の新たな関数を得ることになります。持ち上げられた(型 `f a` の)最初の引数にその関数を部分適用すると、型 `f (b -> c)` の新たな包まれた関数が得られます。それから、2番目の持ち上げられた(型 `f b` の)引数へ `<*>` を適用することができ、型 `f c` の最終的な値を得ます。

まとめると、もし `x :: f a` と `y :: f b` があるなら、式 `(f <$> x) <*> y` は型 `f c` を持つことがわかりました。Preludeで定義された優先順位の規則に従うと、`f <$> x <*> y` というように括弧を外すことができます。

一般的に言えば、最初の引数に `<$>` を使い、残りの引数に対しては `<*>` を使います。`lift3` で説明すると次のようになります。

```
lift3 :: forall a b c d f. (Prelude.Apply f) =>
    (a -> b -> c -> d) ->
    f a -> f b -> f c -> f d
lift3 f x y z = f <$> x <*> y <*> z
```

この式の型がちゃんと整合しているかの確認は、読者への演習として残しておきます。

例として、`<$>` と `<*>` をそのまま使うと、`Maybe` 上に `address` 関数を持ち上げることができます。

```
> address <$> Just "123 Fake St." <*> Just "Faketown" <*> Just "CA"

Just (Address { street: "123 Fake St.", city: "Faketown", state: "CA" })

> address <$> Just "123 Fake St." <*> Nothing <*> Just "CA"

Nothing
```

このように、引数が異なる他のいろいろな関数を `Maybe` 上に持ち上げてみてください。

## 7.5 Applicative型クラス

これに関連する `Applicative` という型クラスが存在しており、次のように定義されています。

```
class (Apply f) <= Applicative f where
    pure :: forall a. a -> f a
```

`Applicative` は `Apply` の下位クラスであり、`pure` 関数が定義されています。`pure` は値

を取り、その型の型構築子 `f` で包まれた値を返します。

`Maybe` についての `Applicative` インスタンスは次のようになります。

```
instance applicativeMaybe :: Applicative Maybe where
  pure x = Just x
```

`Applicative` 関手は関数を持ち上げることが可能な関手だと考えるとすると、`pure` は引数のない関数の持ち上げだというように考えることができます。

## 7.6 `Applicative` に対する直感的理解

PureScriptの関数は純粋であり、副作用は持っていません。`Applicative` 関手は、関手 `f` によって表現されたある種の副作用を提供するような、より大きな「プログラミング言語」を扱えるようにします。

たとえば、関手 `Maybe` はオプションな値の副作用を表現しています。その他の例としては、型 `err` のエラーの可能性の副作用を表す `Either err` や、大域的な構成を読み取る副作用を表す `Arrow` 関手 (arrow functor) `r ->` があります。ここでは `Maybe` 関手についてだけを考えることにします。

もし関手 `f` が作用を持つより大きなプログラミング言語を表すとする

と、`Apply` と `Applicative` インスタンスは小さなプログラミング言語 (PureScript) から新しい大きな言語へと値や関数を持ち上げることが可能にします。

`pure` は純粋な (副作用がない) 値をより大きな言語へと持ち上げますし、関数については上で述べたとおり `<$>` と `<*>` を使うことができます。

ここで新たな疑問が生まれます。もし PureScript の関数と値を新たな言語へ埋め込むのに `Applicative` が使えるなら、どうやって新たな言語は大きくなっているのでしょうか。この答えは関手 `f` に依存します。もしなんらかの `x` について `pure x` で表せないような型 `f a` の式を見つけたなら、その式はそのより大きな言語だけに存在する項を表しているということです。

`f` が `Maybe` のときの式 `Nothing` がその例になっています。`Nothing` を何らかの `x` について `pure x` というように書くことはできません。したがって、PureScript は省略可能な値を表す新しい項 `Nothing` を含むように拡大されたと考えることができます。

## 7.7 その他の作用について

それでは、他にも `Applicative` 関手へと関数を持ち上げる例をいろいろ見ていきましょう。

次は、`psci` で定義された3つの名前を結合して完全な名前を作る簡単なコード例です。

```
> let fullName first middle last = last ++ ", " ++ first ++ " " ++ middle

> fullName "Phillip" "A" "Freeman"
Freeman, Phillip A
```

この関数は、クエリパラメータとして与えられた3つの引数を持つ、(とても簡単な!)ウェブサービスの実装であるとしましょう。使用者が3つの引数すべてを与えたことを確かめたいので、引数が存在するかどうかを表す `Maybe` 型をつかうことになるでしょう。`fullName` を `Maybe` の上へ持ち上げると、省略された引数を確認するウェブサービスを実装することができます。

```
> :i Data.Maybe
> fullName <$> Just "Phillip" <*> Just "A" <*> Just "Freeman"

Just ("Freeman, Phillip A")

> fullName <$> Just "Phillip" <*> Nothing <*> Just "Freeman"

Nothing
```

この持ち上げた関数は、引数のいずれかが `Nothing` なら `Nothing` 返すことに注意してください。

これで、もし引数が不正ならWebサービスからエラー応答を送信することができるので、なかなかいい感じです。しかし、どのフィールドが間違っていたのかを応答で表示できると、もっと良くなるでしょう。

`Maybe` 上へ持ち上げる代わりに `Either String` 上へ持ち上げるようにすると、エラーメッセージを返すことができるようになります。まずは入力を `Either String` を使ってエラーを発信できる計算に変換する演算子を書きましょう。

```
> let (<?>) Nothing err = Left err
      (<?>) (Just a) _ = Right a
```

注意: `Either err` `Applicative` 関手において、`Left` 構築子は失敗を表しており、`Right` 構築子は成功を表しています。

これで `Either String` 上へ持ち上げることで、それぞれの引数について適切なエラーメッセージを提供できるようになります。

```
> let fullNameEither first middle last =  
    fullName <$> (first <?> "First name was missing")  
    <*> (middle <?> "Middle name was missing")  
    <*> (last <?> "Last name was missing")  
  
> :t fullNameEither  
Maybe String -> Maybe String -> Maybe String -> Either String String
```

この関数は `Maybe` の3つの省略可能な引数を取り、`String` のエラーメッセージか `String` の結果のどちらかを返します。

いろいろな入力でこの関数を試してみましょう。

```
> fullNameEither (Just "Phillip") (Just "A") (Just "Freeman")  
Right ("Freeman, Phillip A")  
  
> fullNameEither (Just "Phillip") Nothing (Just "Freeman")  
Left ("Middle name was missing")  
  
> fullNameEither (Just "Phillip") (Just "A") Nothing  
Left ("Last name was missing")
```

このとき、すべてのフィールドが与えられれば成功の結果が表示され、そうでなければ省略されたフィールドのうち最初のものに対応するエラーメッセージが表示されます。しかし、もし複数の入力が省略されているとき、最初のエラーしか見ることができません。

```
> fullNameEither Nothing Nothing Nothing  
  
Left ("First name was missing")
```

これでも十分なときもありますが、エラー時にすべての省略されたフィールドの一覧がほしいときは、`Either String` よりも強力なものが必要です。この章の後半でこの解決策を見ていきます。

## 7.8 作用の結合

抽象的にApplicative関手を扱う例として、Applicative関手 `f` によって表現された副作用を

総称的に組み合わせる関数をどのように書くのかをこの節では示します。

これはどういう意味でしょうか？何らかの `a` について型 `f a` の包まれた引数の配列があるとしましょう。型 `[f a]` の配列があるということです。直感的には、これは `f` によって追跡される副作用を持つ、戻り値の型が `a` の計算の配列を表しています。これらの計算のすべてを順番に実行することができれば、`[a]` 型の結果の配列を得るでしょう。しかし、まだ `f` によって追跡される副作用が残ります。つまり、元の配列の中の作用を「結合する」ことにより、型 `[f a]` の何かを型 `f [a]` の何かへと変換することができると考えられます。

任意の固定長配列の長さ `n` について、その引数を要素に持った長さ `n` の配列を構築するような `n` 引数の関数が存在します。たとえば、もし `n` が 3 なら、関数は `\x y z -> [x, y, z]` です。この関数の型は `a -> a -> a -> [a]` です。Applicative インスタンスを使うと、この関数を `f` の上へ持ち上げて関数型 `f a -> f a -> f a -> f [a]` を得ることができます。しかし、いかなる `n` についてもこれが可能なので、いかなる引数の配列についても同じように持ち上げられることが確かめられます。

したがって、次のような関数を書くことができるはずです。

```
combineArray :: forall f a. (Applicative f) => [f a] -> f [a]
```

この関数は副作用を持つかもしれない引数の配列をとり、それぞれの副作用を適用することで、`f` に包まれた単一の配列を返します。

この関数を書くためには、引数の配列の長さについて考えます。配列が空の場合はどんな作用も実行する必要はありませんから、`pure` を使用して単に空の配列を返すことができます。

```
combineArray [] = pure []
```

実際のところ、これが可能な唯一の定義です！

入力の配列が空でないならば、型 `f a` の先頭要素と、型 `[f a]` の配列の残りについて考えます。また、再帰的に配列の残りを結合すると、型 `f [a]` の結果を得ることができます。`<$>` と `<*>` を使うと、`cons` 関数を先頭と配列の残りの上に持ち上げることができます。

```
combineArray (x : xs) = (:) <$> x <*> combineArray xs
```

繰り返しになりますが、これは与えられた型に基づいている唯一の妥当な実装です。

Maybe 型構築子を例にとりて、`psci` でこの関数を試してみましょう。



```
> combineArray [Just 1, Just 2, Just 3]
Just [1,2,3]

> combineArray [Just 1, Nothing, Just 2]
Nothing
```

`Maybe` へ特殊化して考えると、配列のすべての要素が `Just` であるとき、そのときに限りこの関数は `Just` を返します。そうでなければ、`Nothing` を返します。オプションな結果を返す計算の配列は、そのすべての計算が結果を持っていたときに全体も結果を持っているという、オプションな値に対応したより大きな言語での振る舞いに対する直感的な理解とこれは一致しています。

しかも、`combineArray` 関数はどんな `Applicative` に対しても機能します！ `Either err` を使ってエラーを発信するかもしれないかったり、`r ->` を使って大域的な状態を読み取る計算を連鎖させるときにも `combineArray` 関数を使うことができるのです。

`combineArray` 関数については、後ほど `Traversable` 関手について考えるときに再び扱います。

## 演習

1. (簡単) `lift2` を使って、オプションな引数に対して働く、数に対する演算子 `+`、`-`、`*`、`/` の持ち上げられたバージョンを書いてください。
2. (やや難しい) 上で与えられた `lift3` の定義について、`<$>` と `<*>` の型が整合していることを確認して下さい。
3. (難しい) 型 `forall a f. (Applicative f) => Maybe (f a) -> f (Maybe a)` を持つ関数 `combineMaybe` を書いてください。この関数は副作用をもつオプションな計算をとり、オプションな結果をもつ副作用のある計算を返します。

## 7.9 Applicativeによる検証

この章のソースコードでは電話帳アプリケーションで使われるいろいろなデータ型が定義されています。詳細はここでは割愛しますが、`Data.AddressBook` モジュールからエクスポートされる重要な関数は次のような型を持っています。

```
address :: String -> String -> String -> Address

phoneNumber :: PhoneType -> String -> PhoneNumber
```

```
person :: String -> String -> Address -> [PhoneNumber] -> Person
```

ここで、`PhoneType` は次のような代数的データ型として定義されています。

```
data PhoneType = HomePhone | WorkPhone | CellPhone | OtherPhone
```

これらの関数は住所録の項目を表す `Person` を構築するのに使います。例えば、`Data.AddressBook` には次のような値が定義されています。

```
examplePerson :: Person
examplePerson =
  person "John" "Smith"
    (address "123 Fake St." "FakeTown" "CA")
    [ phoneNumber HomePhone "555-555-5555"
      , phoneNumber CellPhone "555-555-0000"
    ]
```

`psci` でこれらの値使ってみましょう(結果は整形されています)。

```
> :i Data.AddressBook
> examplePerson

Person {
  firstName: "John",
  lastName: "Smith",
  address: Address {
    street: "123 Fake St.",
    city: "FakeTown",
    state: "CA"
  },
  phones: [ PhoneNumber {
    type: HomePhone,
    number: "555-555-5555"
  }, PhoneNumber {
    type: CellPhone,
    number: "555-555-0000"
  } ]
}
```

前の章では型 `Person` のデータ構造を検証するのに `Either String` 関手をどのように使うかを見ました。例えば、データ構造の2つの名前を検証する関数が与えられたとき、データ構造全体を次のように検証することができます。

```

nonEmpty :: String -> Either String Unit
nonEmpty "" = Left "Field cannot be empty"
nonEmpty _  = Right unit

validatePerson :: Person -> Either String Person
validatePerson (Person o) =
  person <$> (nonEmpty o.firstName *> pure o.firstName)
             <*> (nonEmpty o.lastName *> pure o.lastName)
             <*> pure o.address
             <*> pure o.phones

```

最初の2行では `nonEmpty` 関数を使って空文字列でないことを検証しています。もし入力为空なら `nonEmpty` はエラーを返し( `Left` 構築子で示されています)、そうでなければ `Right` 構築子を使って空の値(`unit`)を正常に返します。2つの検証を実行し、右側の検証の結果を返すことを示す連鎖演算子 `*>` を使っています。ここで、入力を変更せずに返す検証器として右側では単に `pure` を使っています。

最後の2行では何の検証も実行せず、単に `address` フィールドと `phones` フィールドを残りの引数として `person` 関数へと提供しています。

この関数は `psci` でうまく動作するように見えますが、以前見たような制限があります。

```

> validatePerson $ person "" "" (address "" "" "") []

Left ("Field cannot be empty")

```

`Either String` Applicative関手は遭遇した最初のエラーだけを返します。でもこの入力では、名前の不足と姓の不足という2つのエラーがわかるようにしたくなるでしょう。

`purescript-validation` ライブラリは別のApplicative関手も提供されています。これは単に `v` と呼ばれていて、何らかの半群(Semigroup)でエラーを返す機能があります。たとえば、`v [String]` を使うと、新しいエラーを配列の最後に連結していき、`String` の配列をエラーとして返すことができます。

`Data.Validation` モジュールは `Data.AddressBook` モジュールのデータ構造を検証するために `v [String]` Applicative関手を使っています。

`Data.AddressBook.Validation` モジュールにある検証の例としては次のようになります。

```

type Errors = [String]

nonEmpty :: String -> String -> v Errors Unit

```

```

nonEmpty field "" = invalid ["Field ' ' ++ field ++ ' ' cannot be empty"]
nonEmpty _      _ = pure unit

lengthIs :: String -> Number -> String -> V Errors Unit
lengthIs field len value | S.length value /= len =
    invalid ["Field ' ' ++ field ++ ' ' must have length " ++ show len]
lengthIs _      _      _ =
    pure unit

validateAddress :: Address -> V Errors Address
validateAddress (Address o) =
    address <$> (nonEmpty "Street" o.street *> pure o.street)
               <*> (nonEmpty "City"   o.city   *> pure o.city)
               <*> (lengthIs "State" 2 o.state *> pure o.state)

```

`validateAddress` は `Address` を検証します。`street` と `city` が空でないかどうか、`state` の文字列の長さが2であるかどうかを検証します。

`nonEmpty` と `lengthIs` の2つの検証関数はいずれも、`Data.Validation` モジュールで提供されている `invalid` 関数をエラーを示すために使っていることに注目してください。`[String]` 半群を扱っているので、`invalid` は引数として文字列の配列を取ります。

`psci` でこの関数を使ってみましょう。

```

> :i Data.AddressBook
> :i Data.AddressBook.Validation

> validateAddress $ address "" "" ""

Invalid ([ "Field 'Street' cannot be empty"
           , "Field 'City' cannot be empty"
           , "Field 'State' must have length 2"
           ])

> validateAddress $ address "" "" "CA"

Invalid ([ "Field 'Street' cannot be empty"
           , "Field 'City' cannot be empty"
           ])

```

これで、すべての検証エラーの配列を受け取ることができるようになりました。

## 7.10 正規表現検証器

`validatePhoneNumber` 関数では引数の形式を検証するために正規表現を使っています。重要なのは `matches` 検証関数で、この関数は `Data.String.Regex` モジュールにて定義されている `Regex` を使って入力を検証しています。

```
matches :: String -> R.Regex -> String -> V.Errors Unit
matches _      regex value | R.test regex value =
  pure unit
matches field _      _      =
  invalid ["Field '" ++ field ++ "' did not match the required format"]
```

繰り返しになりますが、`pure` は常に成功する検証を表しており、エラーの配列の伝達には `invalid` が使われています。

これまでと同じような感じで、`validatePhoneNumber` は `matches` 関数から構築されています。

```
validatePhoneNumber :: PhoneNumber -> V.Errors PhoneNumber
validatePhoneNumber (PhoneNumber o) =
  phoneNumber <$> pure o."type"
  <*> (matches "Number" phoneNumberRegex o.number *> pure o.number)
```

また、`psci` でいろいろな有効な入力や無効な入力に対して、この検証器を実行してみてください。

```
> validatePhoneNumber $ phoneNumber HomePhone "555-555-5555"

Valid (PhoneNumber { type: HomePhone, number: "555-555-5555" })

> validatePhoneNumber $ phoneNumber HomePhone "555.555.5555"

Invalid (["Field 'Number' did not match the required format"])
```

## 演習

1. (簡単) 正規表現の検証器を使って、`Address` 型の `state` フィールドが2文字のアルファベットであることを確かめてください。ヒント: `phoneNumberRegex` のソースコードを参照してみましょう。
2. (やや難しい) `matches` 検証器を使って、文字列に全く空白が含まれないことを検証する検証関数を書いてください。この関数を使って、適切な場合に `notEmpty` を置き換えてください。

## 7.11 Traversable 関手

残った検証器は、これまで見てきた検証器を組み合わせて `Person` 全体を検証する `validatePerson` です。

```
arrayNonEmpty :: forall a. String -> [a] -> V Errors Unit
arrayNonEmpty field [] =
  invalid ["Field '" ++ field ++ "' must contain at least one value"]
arrayNonEmpty _ _ =
  pure unit

validatePerson :: Person -> V Errors Person
validatePerson (Person o) =
  person <$> (nonEmpty "First Name" o.firstName *>
    pure o.firstName)
  <*> (nonEmpty "Last Name" o.lastName *>
    pure o.lastName)
  <*> validateAddress o.address
  <*> (arrayNonEmpty "Phone Numbers" o.phones *>
    traverse validatePhoneNumber o.phones)
```

ここに今まで見たことのない興味深い関数がひとつあります。最後の行で使われている `traverse` です。

`traverse` は `Data.Traversable` モジュールの `Traversable` 型クラスで定義されています。

```
class (Functor t, Foldable t) <= Traversable t where
  traverse :: forall a b f. (Applicative f) => (a -> f b) -> t a -> f (t b)
  sequence :: forall a f. (Applicative f) => t (f a) -> f (t a)
```

`Traversable` は `Traversable` 関手の型クラスを定義します。これらの関数の型は少し難しくうに見えるかもしれませんが、`validatePerson` は良いきっかけとなる例です。

すべての `Traversable` 関手は `Functor` と `Foldable` のどちらでもあります(`Foldable` 関手は構造をひとつの値へとまとめる、畳み込み操作を提供する型構築子であったことを思い出してください)。それ加えて、`Traversable` 関手はその構造に依存した副作用のあつまりを連結する機能を提供します。

複雑そうに聞こえるかもしれませんが、配列の場合に特殊化して簡単に考えてみましょう。配列型構築子は `Traversable` である、つまり次のような関数が存在するということです。

```
traverse :: forall a b f. (Applicative f) => (a -> f b) -> [a] -> f [b]
```

直感的には、Applicative関手 `f` と、型 `a` の値をとり型 `b` の値を返す(`f` で追跡される副作用を持つ)関数が与えられたとき、型 `[a]` の配列の要素それぞれにこの関数を適用し、型 `[b]` の(`f` で追跡される副作用を持つ)結果を得ることができます。

まだよくわからないでしょうか。それでは、更に `f` を `V Errors` Applicative関手に特殊化して考えてみましょう。 `traversable` が次のような型の関数だとしましょう。

```
traverse :: forall a b. (a -> V Errors b) -> [a] -> V Errors [b]
```

この型シグネチャは、型 `a` についての検証関数 `f` があれば、 `traverse f` は型 `[a]` の配列についての検証関数であるということを言っています。これはまさに今必要になっている `Person` データ構造体の `phones` フィールドを検証する検証器そのものです！それぞれの要素が成功するかどうかを検証する検証関数を作るために、 `validatePhoneNumber` を `traverse` へ渡しています。

一般に、 `traverse` はデータ構造の要素をひとつずつ辿っていき、副作用のある計算を実行して結果を累積します。

`Traversable` のもう一つの関数、 `sequence` の型シグネチャには見覚えがあるかもしれません。

```
sequence :: forall a f. (Applicative m) => t (f a) -> f (t a)
```

実際、先ほど書いた `combineArray` 関数は `Traversable` 型の `sequence` 関数が特殊化されたものに過ぎません。 `t` を配列型構築子として、 `combineArray` 関数の型をもう一度考えてみましょう。

```
combineArray :: forall f a. (Applicative f) => [f a] -> f [a]
```

`Traversable` 関手は、作用のある計算の集合を集めてその作用を連鎖させるという、データ構造走査の考え方を把握できるようにするものです。実際、 `sequence` と `traversable` は `Traversable` を定義するのにどちらも同じくらい重要です。これらはお互いが互いを利用して実装することができます。これについては興味ある読者への演習として残しておきます。

配列の `Traversable` インスタンスは `Data.Traversable` モジュールで与えられています。 `traverse` の定義は次のようになっています。

```
-- traverse :: forall a b f. (Applicative f) => (a -> f b) -> [a] -> f [b]
traverse _ [] = pure []
traverse f (x : xs) = (:) <$> f x <*> traverse f xs
```

入力が空の配列のときには、単に `pure` を使って空の配列を返すことができます。配列が空でないときは、関数 `f` を使うと先頭の要素から型 `f b` の計算を作成することができます。また、配列の残りに対して `traverse` を再帰的に呼び出すことができます。最後に、Applicative関手 `f` までcons演算子 `(:)` を持ち上げて、2つの結果を組み合わせます。

Traversable関手の例はただの配列以外にもあります。以前に見た `Maybe` 型構築子も `Traversable` のインスタンスを持っています。 `psci` で試してみましょう。

```
> :i Data.Maybe

> traverse (nonEmpty "Example") Nothing

Valid (Nothing)

> traverse (nonEmpty "Example") (Just "")

Invalid (["Field 'Example' cannot be empty"])

> traverse (nonEmpty "Example") (Just "Testing")

Valid (Just (Unit {}))
```

これらの例では、`Nothing` の値の走査は検証なしで `Nothing` の値を返し、`Just x` を走査すると `x` を検証するのにこの検証関数が使われるということを示しています。つまり、`traverse` は型 `a` についての検証関数を取り、`Maybe a` についての検証関数を返すのです。

他にも、何らかの型 `a` についての `Tuple a` や `Either a` や、連結リストの型構築子 `List` といったTraversable関手があります。一般的に、「コンテナ」のようなデータ型のコンストラクタは大抵は `Traversable` インスタンスを持っています。例として、演習では二分木の型の `Traversable` インスタンスを書くようになっています。

## 演習

1. (やや難しい) 左から右へと副作用を連鎖させる、次のような二分木データ構造についての `Traversable` インスタンスを書いてください。



```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

これは木の走査の順序に対応しています。行きがけ順の走査についてはどうでしょうか。帰りがけ順では？

2. (やや難しい) `Data.Maybe` を使って `Person` の `address` フィールドを省略可能になるようにコードを変更してください。ヒント: `traverse` を使って型 `Maybe a` のフィールドを検証してみましょう。
3. (難しい) `traverse` を使って `sequence` を書いてみましょう。また、`sequence` を使って `traverse` を書けるでしょうか？

## 7.12 Applicative 関手による並列処理

これまでの議論では、Applicative 関手がどのように「副作用を結合」させるかを説明するときに、「結合」(combine)という単語を選びました。しかしながら、これらのすべての例において、Applicative 関手は作用を「連鎖」(sequence)させる、というように言っても同じく妥当です。`Traverse` 関手はデータ構造に従って作用を順番に結合させる `sequence` 関数を提供する、という直感的理解とこれは一致するでしょう。

しかし一般には、Applicative 関手はこれよりももっと一般的です。Applicative 関手の規則は、その計算を実行する副作用にどんな順序付けも強制しません。実際、並列に副作用を実行するための Applicative 関手というものは妥当になりえます。

たとえば、`v` 検証関手はエラーの配列を返しますが、その代わりに `Set` 半群を選んだとしてもやはり正常に動き、このときどんな順序でそれぞれの検証器を実行しても問題はありません。データ構造に対して並列にこれを実行することさえできるのです！

非同期計算を表現する型構築子 `Async` は、並列に結果を計算する `Applicative` インスタンスを持つことができます。

```
f <$> Async computation1 <*> Async computation2
```

この計算は、`computation1` と `computation2` を非同期に使うって値を計算を始めるでしょう。そして両方の結果の計算が終わった時に、関数 `f` を使ってひとつの結果へと結合するでしょう。

この考え方の詳細は、本書の後半でコールバック地獄の問題に対して Applicative 関手を応用するときに見ていきます。

Applicative関手は並列に結合されうる副作用を捕捉する自然な方法です。

## 7.13 まとめ

---

この章では新しい考え方をたくさん扱いました。

- 関数適用の概念を副作用の考え方を表現する型構築子へと一般化する、Applicative関手の概念を導入しました。
- データ構造の検証という課題にApplicative関手がどのような解決策を与えるか、単一のエラーの報告からデータ構造を横断するすべてのエラーの報告へ変換できるApplicative関手を見てきました。
- 要素が副作用を持つ値の結合に使われることのできるコンテナであるTraversable関手の考え方を表現する、`Traversable` 型クラス導入しました。

Applicative関手は多くの問題に対して優れた解決策を与える興味深い抽象化です。本書を通じて何度も見ることになるでしょう。今回は、どうやって検証を行うかではなく、何を検証器が検証すべきなのかを定義することを可能にする、宣言的なスタイルで書く手段をApplicative関手は提供しました。一般に、Applicative関手は領域特化言語の設計のための便利な道具になります。

次の章では、これに関連するモナドという型クラスについて見ていきましょう。

## 8 Effモナド

---

### 8.1 この章の目標

---

第7章では、オプションな型やエラーメッセージ、データの検証など、副作用を扱いを抽象化するApplicative関手を導入しました。この章では、より表現力の高い方法で副作用を扱うための別の抽象化、モナドを導入します。

この章の目的は、なぜモナドが便利な抽象化なのか、do記法とどう関係するのかについて説明することです。ブラウザでユーザインターフェイスを構築する副作用を扱うためのある種のモナドを使って、前の章の住所録の例を作ることしましょう。これから扱うEffモナドは、PureScriptにおけるとても重要なモナドです。Effモナドはいわゆるネイティブな作用をカプセル化するのに使われます。

### 8.2 プロジェクトの準備

---

このプロジェクトのソースコードは前の章のソースコードの上に構築しますが、そのソースファイルを含めるようにGruntビルドスクリプトを使用しています。

コードは3つのモジュールに分かれています。

- `Main` - アプリケーションへのエントリポイントを提供します。
- `Data.AddressBook.UI` - ブラウザのユーザインターフェースをレンダリングするための関数を提供します。
- `Control.Monad.Eff.DOM` - DOMを操作する関数の簡単なライブラリを提供します。

このプロジェクトを実行するには、Gruntでビルドし、`html/index.html` ファイルをウェブブラウザで開いてください。

## 8.3 モナドとdo記法

do記法は配列内包表記を扱うときに最初に導入されました。配列内包表記は `Data.Array` モジュールの `concatMap` 関数の構文糖として提供されています。

次の例を考えてみましょう。2つのサイコロを振って出た目を数え、出た目の合計が `n` のときそれを得点とすることを考えます。次のような非決定的なアルゴリズムを使うとこれを実現することができます。

- 最初の投擲で値 `x` を選択します
- 2回目投擲で値 `y` を選択します
- もし `x` と `y` の和が `n` なら組 `{x, y}` を返し、そうでなければ失敗します

配列内包表記を使うと、この非決定的アルゴリズムを自然に書くことができます。

```
countThrows :: Number -> [[Number]]
countThrows n = do
  x <- range 1 6
  y <- range 1 6
  if x + y == n then return [x, y] else empty
```

`psci` で動作を見てみましょう。

```
> countThrows 10
[[4,6],[5,5],[6,4]]

> countThrows 12
[[6,6]]
```

前の章では、オプションな値に対応したより大きなプログラミング言語へとPureScriptの関数を埋め込む、`Maybe` Applicative関手についての直感的理解を養いました。同様に配列モナドについても、非決定選択に対応したより大きなプログラミング言語へPureScriptの関数を埋め込む、というような直感的理解を得ることができます。

一般に、ある型構築子 `m` のモナドは、型 `m a` の値を持つdo記法を使う方法を提供します。上の配列内包表記では、すべての行に何らかの型 `a` についての型 `[a]` の計算が含まれていることに注目してください。一般に、do記法ブロックのすべての行は、何らかの型 `a` とモナド `m` について、型 `m a` の計算を含んでいます。モナド `m` はすべての行で同じでなければなりません(つまり、副作用の種類は固定されます)が、型 `a` は異なることもあります(言い換えると、この計算は異なる型の結果を持つことができます)。

型構築子 `Maybe` が適用された、do記法の別の例を見てみましょう。XMLノードを表す型 `XML` と演算子があるとします。

```
(</>) :: XML -> String -> Maybe XML
```

この演算子はノードの子の要素を探し、もしそのような要素が存在しなければ `Nothing` を返します。

この場合、do記法を使うと深い入れ子になった要素を検索することができます。XML文書として符号化された利用者情報から、利用者の住んでいる市町村を読み取りたいとします。

```
userCity :: XML -> Maybe XML
userCity root = do
  prof <- root </> "profile"
  addr <- prof </> "address"
  city <- addr </> "city"
  return city
```

`userCity` 関数は子の要素である `profile` を探し、`profile` 要素の中にある `address` 要素、最後に `address` 要素から `city` 要素を探します。これらの要素のいずれかが欠落している場合は、返り値は `Nothing` になります。そうでなければ、返り値は `city` ノードから `Just` を使って構築されています。

最後の行の `return` 関数は予約語ではないことを思い出してください。`return` は実際にすべての `Applicative` 関手について定義されている `pure` 関数の別名です。JavaScriptの `return` 文を連想するかもしれませんが、関数の途中での復帰とはまったく関係がありません。最後の行を `Just city` へ変更しても同じように正しく動きます。

## 8.4 モナド型クラス

`Monad` 型クラスは次のように定義されています。

```
class (Apply m) <= Bind m where
    (>>=) :: forall a b. m a -> (a -> m b) -> m b

class (Applicative m, Bind m) <= Monad m
```

ここで鍵となる関数は `Bind` 型クラスで定義されている演算子 `=>>` で、これは「束縛」(bind)と呼ばれています。 `Monad` 型クラスは、すでに見てきた `Applicative` 型クラスの操作で `Bind` を拡張します。

`Bind` 型クラスの例をいくつか見てみるのがわかりやすいでしょう。配列についての `Bind` の妥当な定義は次のようになります。

```
instance bindArray :: Bind [] where
    (>>=) xs f = f `concatMap` xs
```

これは以前にほのめかした配列内包表記と `concatMap` 関数の関係を説明しています。

`Maybe` 型構築子についての `Bind` の実装は次のようになります。

```
instance bindMaybe :: Bind Maybe where
    (>>=) Nothing _ = Nothing
    (>>=) (Just a) f = f a
```

この定義はdo記法ブロックを通じて伝播された欠落した値についての直感的理解を補強するものです。

`Bind` 型クラスとdo記法がどのように関係しているかを見て行きましょう。最初に何らかの計算結果から値を束縛するような、簡単などdo記法ブロックについて考えてみましょう。

```
do value <- someComputation
   whatToDoNext
```

PureScriptコンパイラはこのようなパターンを見つけるたびにコードを次のように置き換えます。

```
someComputation >>= \value -> whatToDoNext
```

この計算 `whatToDoNext` は `value` に依存することができます。

連続した複数の束縛がある場合でも、この規則が先頭のほうから複数回適用されます。例えば、先ほど見た `userCity` の例では次のように構文糖が脱糖されます。

```
userCity :: XML -> Maybe XML
userCity root =
  root </> "profile" >>= \prof ->
    prof </> "address" >>= \addr ->
      addr </> "city" >>= \city ->
        return city
```

do記法を使って表現されたコードは、`>>=` 演算子を使って書かれた同じ意味のコードよりしばしば読みやすくなることも特筆すべき点です。一方で、明示的に `>>=` を使って束縛が書くと、point-free形式でコードを書く機会を増やすことになります。ただし、通常は読みやすさを優先すべきでしょう。

## 8.5 モナド則

`Monad` 型クラスはモナド則(monad laws)と呼ばれる3つの規則を持っています。これらは `Monad` 型クラスの理にかなった実装から何を期待できるかを教えてくれます。

do記法を使用してこれらの規則を説明していくのが最も簡単でしょう。

### 8.5.1 Identity 律

右単位元則(right-identity law)が3つの規則の中で最も簡単です。この規則はdo記法ブロックの最後の式であれば、`return` の呼び出しを排除することができますと言っています。

```
do
  x <- expr
  return x
```

右単位元則は、この式は単なる `expr` と同じだと言っています。

左単位元則(left-identity law)は、もしそれがdo記法ブロックの最初の式であれば、`return` の呼び出しを除去することができると述べています。

```
do
  x <- return y
  next
```

このコードの名前 `x` を式 `y` で置き換えたものと `next` は同じです。

最後の規則は結合則(associativity law)です。これは入れ子になったdo記法ブロックをどう扱うのかについて教えてくれます。

```
c1 = do
  y <- do
    x <- m1
    m2
  m3
```

上記のコード片は、次のコードと同じです。

```
c2 = do
  x <- m1
  y <- m2
  m3
```

これら計算にはそれぞれ、3つのモナドの式 `m1`、`m2`、`m3` が含まれています。どちらの場合でも `m1` の結果は名前 `x` に束縛され、`m2` の結果は名前 `y` に束縛されます。

`c1` では2つの式 `m1` と `m2` がそれぞれのdo記法ブロック内にグループ化されています。

`c2` では `m1`、`m2`、`m3` の3つすべての式が同じdo記法ブロックに現れています。

結合規則は 入れ子になったdo記法ブロックをこのように単純化しても安全であるということを言っています。

注意: do記法がどのように `>>=` の呼び出しへと脱糖されるかの定義により、`c1` と `c2` はいずれも次のコードと同じです。`

```
c3 = do
  x <- m1
  do
    y <- m2
    m3
```

## 8.6 モナドと畳み込み

抽象的にモナドを扱う例として、この節では `Monad` 型クラスの何らかの型構築子と一緒に機能するある関数を示していきます。これはモナドによるコードが副作用を伴う「より大きな言語」でのプログラミングと対応しているという直感的理解を補強しますし、モナドによるプログラミングがもたらす一般性も示しています。

これから `foldM` と呼ばれる関数を書いてみます。これは以前扱った `foldl` 関数をモナドの文脈へと一般化します。型シグネチャは次のようになっています。

```
foldM :: forall m a b. (Monad m) => (a -> b -> m a) -> a -> [b] -> m a
```

モナド `m` が現れている点を除いて、`foldl` の型と同じであることに注意しましょう。

```
foldl :: forall a b. (a -> b -> a) -> a -> [b] -> a
```

直感的には、`foldM` はさまざまな副作用の組み合わせに対応した文脈での配列の畳み込みを行うと捉えることができます。

例として `m` が `Maybe` であるとする、この畳み込みはそれぞれの段階で `Nothing` を返すことで失敗することができます。それぞれの段階ではオプションな結果を返しますから、それゆえ畳み込みの結果もオプションになります。

もし `m` として配列の型構築子 `[]` を選ぶとすると、畳み込みのそれぞれの段階で複数の結果を返すことができ、畳み込みは結果それぞれに対して次の手順を続けます。最後に、結果の集まりは、可能な経路すべての畳み込みから構成されることになります。これはグラフの走査と対応しています！

`foldM` を書くには、単に入力の配列について場合分けをするだけです。

配列が空なら、型 `a` の結果を生成するための選択肢はひとつしかありません。第2引数を返します。

```
foldM _ a [] = return a
```

`a` をモナド `m` まで持ち上げるために `return` を使わなくてはならないことも忘れないようにしてください。

配列が空でない場合はどうでしょうか？ その場合、型 `a` の値、型 `b` の値、型 `a -> b -> m`



`a` の関数があります。もしこの関数を適用すると、型 `m a` のモナドの結果を手に入れることになります。この計算の結果を逆向きの矢印 `<-` で束縛することができます。

あとは配列の残りに対して再帰するだけです。実装は簡単です。

```
foldM f a (b : bs) = do
  a' <- f a b
  foldM f a' bs
```

`do` 記法を除けば、この実装は配列に対する `foldl` の実装とほとんど同じであることにも注意してください。

`psci` でこれを定義し、試してみましょう。除算可能かどうかを調べて、失敗を示すために `Maybe` 型構築子を使う、整数の「安全な除算」関数を定義してみましょう。

```
safeDivide :: Number -> Number -> Maybe Number
safeDivide a b | a % b == 0 = Just (a / b)
safeDivide _ _ = Nothing
```

これで、`foldM` で安全な除算の繰り返しを表現することができます。

```
> foldM safeDivide 100 [5, 2, 2]
Just (5)

> foldM safeDivide 100 [2, 3, 4]
Nothing
```

もしいずれかの時点で整数にならない除算が行われようとしたら、`foldM safeDivide` 関数は `Nothing` を返します。そうでなければ、`Just` 構築子に包まれた除算の繰り返した累積の結果を返します。

## 8.7 モナドとApplicative

クラス間に上位クラス関係があるため、`Monad` 型クラスのすべてのインスタンスは `Applicative` 型クラスのインスタンスでもあります。

しかしながら、どんな `Monad` のインスタンスについても `Applicative` 型クラスの実装が、それ以上の条件なしで存在し、次のような `ap` が与えられます。

```
ap :: forall m. (Monad m) => m (a -> b) -> m a -> m b
```

```
ap mf ma = do
  f <- mf
  a <- ma
  return (f a)
```

もし `m` が `Monad` 型クラスの規則に従っているなら、`pure` が `return` で与えられ、`<*>` が `ap` で与えられるような、妥当な `Applicative` インスタンスが存在します。

興味のある読者は、これまで登場した `[]`、`Maybe`、`Either e`、`V e` といったモナドについて、この `ap` が `<*>` と一致することを確認してみてください。

もしすべてのモナドが `Applicative` 関手でもあるなら、`Applicative` 関手についての直感的理解をすべてのモナドについても適用することができるはずです。特に、更なる副作用の組み合わせで増強された「より大きな言語」でのプログラミングとモナドがいろいろな意味で一致することを当然に期待することができます。`<$>` と `<*>` を使って、引数が任意個の関数をこの新しい言語へと持ち上げることができるはずです。

しかし、モナドは `Applicative` 関手で可能な以上のことを行うことができ、重要な違いは `do` 記法の構文で強調されています。利用者情報を符号化した XML 文書から利用者の都市を検索する、`userCity` の例についてもう一度考えてみましょう。

```
userCity :: XML -> Maybe XML
userCity root = do
  prof <- root </> "profile"
  addr <- prof </> "address"
  city <- addr </> "city"
  return city
```

2番目の計算が最初の結果 `prof` に依存し、3番目の計算が2番目の計算の結果 `addr` に依存するというようなことを `do` 記法は可能にします。`Applicative` 型クラスのインターフェイスだけを使うのでは、このような以前の値への依存は不可能です。

`pure` と `<*>` だけを使って `userCity` を書こうとしてみれば、これが不可能であることがわかるでしょう。`Applicative` は関数の互いに独立した引数を持ち上げることだけを可能にしますが、モナドはもっと興味深いデータ依存関係に関わる計算を書くことを可能にします。

前の章では `Applicative` 型クラスは並列処理を表現できることを見ました。持ち上げられた関数の引数は互いに独立していますから、これはまさにその通りです。`Monad` 型クラスは計算が前の計算の結果に依存できるようにしますから、同じようにはなりません。モナドはその副作用を順番に組み合わせなければいけません。

## 演習

1. (簡単) `purescript-arrays` パッケージの `Data.Array` モジュールから `head` 関数と `tail` 関数の型を探してください。 `Maybe` モナドと `do` 記法を使い、`head` と `tail` を組み合わせて、3要素以上の配列の3番目の要素を返すような関数を作ってください。その関数は適当な `Maybe` 型を返さなければいけません。
2. (やや難しい) 与えられた幾つかの硬貨を組み合わせてできる可能性のあるすべての合計を決定する関数 `sum` を、`foldM` を使って書いてみましょう。入力の硬貨は、硬貨の価値の配列として与えられます。この関数は次のような結果にならなくてはなりません。

```
> sums []
[0]

> sums [1, 2, 10]
[0,1,2,3,10,11,12,13]
```

ヒント: `foldM` を使うと1行でこの関数を書くことが可能です。重複する要素を取り除いたり、結果を昇順に並び替えたりするのに、`nub` 関数や `sort` 関数を使いなくなるかもしれません。

3. (やや難しい) `Maybe` 型構築子について、`ap` 関数と `<*>` 演算子が一致することを確認してください。
4. (やや難しい) `purescript-maybe` パッケージで定義されている `Maybe` 型についての `Monad` インスタンスが、モナド則を満たしていることを検証してください。
5. (やや難しい) 配列上の `filter` の関数を一般化した関数 `filterM` を書いてください。この関数は次の型シグネチャを持つ必要があります。

```
filterM :: forall m a. (Monad m) => (a -> m Boolean) -> [a] -> m [a]
```

`psci` で `Maybe` と `[]` モナドを使ってその関数を試してみてください。

6. (難しい) すべてのモナドは、次で与えられるような既定の `Functor` インスタンスがあります。

```
(<$>) f a = do
  x <- a
  return (f a)
```

モナド則を使って、すべてのモナドが次を満たすことを証明してください。

```
lift2 f (return a) (return b) = return (f a b)
```

ここで、`Applicative` インスタンスは上で定義された `ap` 関数を使用しています。`lift2` が次のように定義されていたことを思い出してください。

```
lift2 :: forall f a b c. (Applicative f). (a -> b -> c) -> f a -> f b -> f c
lift2 f a b = f <$> a <*> b
```

## 8.8 ネイティブな作用

ここではPureScriptの中核となる重要なモナド、`Eff` モナドについて見ていきます。

`Eff` モナドは `Control.Monad.Eff` モジュール、およびPreludeで定義されています。これはいわゆるネイティブな作用を扱うために使います。

ネイティブな副作用とは何でしょうか。ネイティブな副作用とは、従来のJavaScriptの式が持つ副作用と、PureScript特有の式が持つ副作用を区別するものです。ネイティブな作用には次のようなものがあります。

- コンソール入出力
- 乱数生成
- 例外
- 変更可能な状態の読み書き

また、ブラウザでは次のようなものがあります。

- DOM操作
- XMLHttpRequest / AJAX呼び出し
- WebSocketによる相互作用
- Local Storageの読み書き

すでに「ネイティブでない」副作用の例については数多く見てきています。

- `Maybe` データ型で表現される省略可能な値
- `Either` データ型で表現されるエラー
- 配列やリストで表現される多価関数

これらの区別はわかりにくいので注意してください。エラーメッセージは例外の形でJavaScriptの式の副作用となることがあります。その意味では例外はネイティブな副作用を表していて、`Eff` を使用して表現することができます。しかし、`Either` を使用して実装されたエラーメッセージはJavaScriptランタイムの副作用ではなく、`Eff` を使うスタイルでエラーメッセージを実装するのは適切ではありません。そのため、ネイティブなのは作用自体というより、実行時にどのように実装されているかです。

## 8.9 副作用と純粋性

---

PureScriptのような言語が純粋であるとする、疑問が浮かんできます。副作用がないなら、どうやって役に立つ実際のコードを書くことができるのでしょうか。

その答えはPureScriptの目的は副作用を排除することではないということです。これは、純粋な計算と副作用のある計算とを型システムにおいて区別することができるような方法で、副作用を表現することを目的としているのです。この意味で、言語はあくまで純粋だということです。

副作用のある値は、純粋な値とは異なる型を持っています。このように、例えば副作用のある引数を関数に渡すことはできず、予期せず副作用持つようなことが起こらなくなります。

`Eff` モナドで管理された副作用を実行する唯一の方法は、型 `Eff eff a` の計算をJavaScriptから実行することです。

PureScriptコンパイラは、`--main` コンパイラオプションを与えることで、アプリケーションの起動時に `main` 計算を呼び出すためのJavaScriptコードを簡単に追加で生成できるようにしています。`main` は `Eff` モナドでの計算であることが要求されます。

このように、`main` によって使われる副作用が期待されることを、開発者は正確に知ることができます。加えて、`main` がどのような種類の副作用を持つかを制限するのに `Eff` モナドを使うことができるので、例えば、アプリケーションはコンソールと相互作用するが、それ以外は何もしない、ということを確実に言うことができます。

## 8.10 Effモナド

---

`Eff` モナドの目的は、副作用のある計算に型付けされたAPIを提供すると同時に、効率的なJavaScriptを生成することにあります。これは拡張可能作用(extensible effects)のモナドとも呼ばれており、これについては後述します。

例を示しましょう。次のコードでは乱数を生成するための関数が定義されている `purescript-random` モジュールを使用しています。

```
module Main where

import Control.Monad.Eff
import Control.Monad.Eff.Random

import Debug.Trace

main = do
  n <- random
  print n
```

このファイルが `Main.purs` という名前で保存されているなら、次のコマンドでコンパイルすることができます。

```
psc --main Main Main.purs
```

コンパイルされたJavaScriptを実行すると、コンソールに出力 `0` と `1` の間で無作為に選ばれた数が表示されるでしょう。

このプログラムは、乱数生成とコンソール入出力というJavaScriptランタイムが提供する2種類のネイティブな作用を、`do`記法で組み合わせて使っています。

## 8.11 拡張可能作用

`psci` でモジュールを読み込み、`main` の型を調べてみましょう。

```
> :t Main.main

forall eff. Eff (trace :: Trace, random :: Random | eff) Unit
```

この型はかなり複雑そうに見えますが、PureScriptのレコードの比喻で簡単に説明することができます。

レコード型を使った簡単な関数を考えてみましょう。

```
fullName person = person.firstName ++ " " ++ person.lastName
```

この関数は `firstName` と `lastName` というプロパティを含むレコードから完全な名前の文字列を作成します。もし `psci` でこの関数の型を同様に調べたとしても、次のように表示される

でしょう。

```
forall r. { firstName :: String, lastName :: String | r } -> String
```

この型は「少なくとも `fullName` は `firstName` と `lastName` という2つのフィールドを持つようなレコードをとり、`String` を返す。」というように読みます。

渡したレコードが `firstName` と `lastName` というプロパティさえ持っていれば、その他に余計なフィールドを持っていたとしても `fullName` は気にしません。

```
> firstName { firstName: "Phil", lastName: "Freeman", location: "Los Angeles" }  
  
Phil Freeman
```

同様に、上の `main` の型は「`main` は副作用のある計算で、乱数生成とコンソール入出力、およびそれ以外の任意の種類の副作用を備えた任意の環境で実行することができ、型 `Unit` の値を返す」というように解釈できます。

これは「拡張可能作用」という名前の由来になっています。必要な副作用さえ備えていれば、その副作用の集まりをいつでも拡張できるということです。

## 8.12 作用の混在

拡張可能作用は `Eff` モナドで異なる型の副作用を混在(interleave)させることを可能にします。

先ほど使った `random` 関数は次のような型を持っています。

```
forall eff1. Eff (random :: Random | eff1) Number
```

この作用の集まり `(random :: Random | eff1)` は `main` で見たものと同じではありません。

しかし、作用が一致するように `random` の型を特殊化できます。`eff1` に `(trace :: Trace | eff)` を選べば、これらの2つの作用の集合は同じになります。

同様に `trace` は `main` の作用に合わせて特殊化できる型を持っています。

```
forall eff2. String -> Eff (trace :: Trace | eff2) Unit
```

この場合は、`eff2` に `(random :: Random | eff)` を選ばなくてはなりません。

それが含む副作用を示す `random` と `print` の型がポイントで、より大きな副作用の集まりを持ったより大きな計算を構築するために、他の副作用を混ぜ合わせることができるのです。

`main` の型注釈を与えなくてもよいことに注意してください。`psc` は `random` と `trace` の多相的な型が与えられた `main` の最も一般的な型を見つけることができます。

## 8.13 Effの種

`main` の型は今まで見てきた他の型とは異なります。それを説明するためには、まず `Eff` の種について考える必要があります。値がその型によって分類されるように、型がその種によって分類されることを思い出してください。これまでは `*` (型の種) と `->` (型構築子のための種を構築する) だけから構築された種のみを見てきました。

`Eff` の種を見るには、`psci` で `:k` コマンドを使います。

```
> :k Control.Monad.Eff.Eff
```

```
# ! -> * -> *
```

今まで見たことのない記号が2つあります。

`!` は副作用の型についての型レベルのラベルを表す作用の種です。これを理解するためには、上の `main` で見た2つのラベルがいずれも種 `!` を持っていることに注目してください。

```
> :k Debug.Trace.Trace
```

```
!
```

```
> :k Control.Monad.Eff.Random.Random
```

```
!
```

`#` 種構築子は行の種を構築するのに使われます。行とは順序なしラベル付きの集合のことです。

そして、`Eff` は作用の行と作用の返り値の型という2つの引数を持っています。つまり、`Eff` の最初の引数は、作用の型の順序なしラベル付きの集合であり、2つめの引数は返り値の型だということです。



これで、先ほどの `main` の型を読むことができるようになりました。

```
forall eff. Eff (trace :: Trace, random :: Random | eff) Unit
```

`Eff` の最初の引数は `(trace :: Trace, random :: Random | eff)` です。これは `Trace` 作用と `Random` 作用を含む行です。パイプ記号 `|` は、ラベルが付けられた作用と、それに混ぜあわせたいそれ以外の任意の作用を表す行変数(row variable) `eff` を区切っています。

`Eff` の2番目の引数は、計算の戻り値の型 `Unit` です。

## 8.14 オブジェクトと行

拡張可能作用とレコードに深いつながりをもたらしている `Eff` の種を考えてみましょう。

上で定義した関数 `fullName` を考えます。

```
fullName :: forall r. { firstName :: String, lastName :: String | r } -> String
fullName person = person.firstName ++ " " ++ person.lastName
```

種 `*` の型だけが値を持つので、関数の矢印の左辺にある型の種は `*` でなければなりません。

中括弧は実際には構文糖であり、PureScriptコンパイラによって理解されている完全な型は次のようなものです。

```
fullName :: forall r. Object (firstName :: String, lastName :: String | r) -> Stri
```

中括弧がなくなっており、`Object` 構築子が追加されていることに注意してください。`Object` は `Prim` モジュールで定義されている組み込みの型構築子です。`Object` の種を調べてみると、次のようになっています。

```
> :k Object
```

```
# * -> *
```

つまり、`Object` は型の行をとり型を構築する型構築子なのです。これがレコードについての行多相関数を書くことを可能にしているのです。

この型システムでは、拡張可能作用を扱うのに、行多相レコード(拡張可能レコード)を使うときと同じ機構が使われています。唯一の違いは、ラベルに現れる型の種です。レコードは型の行によってパラメータ化され、`Eff` は作用の行によってパラメータ化されるのです。

これと同じ型システムの機能は、型構築子の行や、行の行でパラメータ化される型を構築するのにさえ使われることがあります！

## 8.15 きめ細かな作用

作用の行は推論されるので、大抵の場合は `Eff` を使うときに型注釈は必須ではありませんが、計算でどの作用が期待されるのかをコンパイラに示すために型注釈が使われることがあります。

先ほどの例を、作用の閉じた行で注釈すると次のようになります。

```
main :: Eff (trace :: Trace, random :: Random) Unit
main = do
  n <- random
  print n
```

行変数 `eff` がないことに注意してください。こうすると、異なった作用の型を使う計算を誤って含めることはできません。このように、コードが持つことを許される副作用を制御することができるのです。

## 8.16 ハンドラとアクション

`trace` や `random` のような関数はアクションと呼ばれます。アクションはそれらの関数の右辺に `Eff` 型を持っており、その目的は新たな効果を導入することにあります。

これは `Eff` 型が関数の引数の型として現れるハンドラとは対照的です。アクションが集合へ必要な作用を追加するのに対し、ハンドラは集合から作用を除去します。

例として、`purescript-exceptions` パッケージを考えてみます。このパッケージでは `throwException` と `catchException` という二つの関数が定義されています。

```
throwException :: forall a eff. Error -> Eff (err :: Exception | eff) a

catchException :: forall a eff. (Error -> Eff eff a) ->
  Eff (err :: Exception | eff) a ->
  Eff eff a
```

`throwException` はアクションです。 `Eff` は右辺に現れていて、新しく `Exception` 作用を導入します。

`catchException` はハンドラです。 `Eff` は関数の第2引数の型として出現しており、作用全体としては `Exception` 作用を除去します。

特定の作用を必要とするコードの部分を限定するために型システムを使うことができるので、これは便利です。作用のあるコードをハンドラで包むことにより、その作用を許さないコードブロックの中に埋め込むことができます。

例えば、`Exception` 作用を使って例外を投げるコード片を書き、それからそのコードを `catchException` で包むことによって、例外を許さないコード片の中にその計算を埋め込むことができます。

JSONドキュメントからアプリケーションの設定を読みたいとしましょう。文書を構文解析する過程で例外を投げることがあります。設定を読み構文解析するこの処理は、次のような型シグネチャを持つ関数として書くことができます。

```
readConfig :: forall eff. Eff (err :: Exception | eff) Config
```

それから、`main` 関数で `catchException` を使用して `Exception` 作用を処理することができます。

```
main = catchException printException $ do
  config <- readConfig
  runApplication config

  where
    printException e = trace (stackTrace e)
```

Preludeでも、副作用なしの計算を取り、それを純粋な値として安全に評価する `runPure` ハンドラが定義されています。

```
type Pure a = forall e. Eff e a

runPure :: forall a. Pure a -> a
```

## 8.17 可変状態

Preludeには `ST` 作用というまた別の作用も定義されています。

`ST` 作用は変更可能な状態を操作するために使われます。純粋関数プログラミングを知っているなら、共有される変更可能な状態は問題を引き起こしやすいということも知っているでしょう。しかしながら、`ST` 作用は型システムを使って安全で局所的な状態変化を可能にし、状態の共有を制限するのです。

`ST` 作用は `Control.Monad.ST` モジュールで定義されています。これがどのように動作するかを確認するには、そのアクションの型を見る必要があります。

```
newSTRef :: forall a h eff. a -> Eff (st :: ST h | eff) (STRef h a)

readSTRef :: forall a h eff. STRef h a -> Eff (st :: ST h | eff) a

writeSTRef :: forall a h eff. STRef h a -> a -> Eff (st :: ST h | eff) a

modifySTRef :: forall a h eff. STRef h a -> (a -> a) -> Eff (st :: ST h | ef
```

`newSTRef` は型 `STRef h a` の変更可能な参照領域を新しく作るのに使われます。`STRef h a` は `readSTRef` アクションを使って状態を読み取ったり、`writeSTRef` アクションや `modifySTRef` アクションで状態を変更するのに使われます。型 `a` は領域に格納された値の型で、型 `h` は型システムのメモリ領域を表しています。

例を示します。小さな時間刻みで簡単な更新関数の実行を何度も繰り返すことによって、重力に従って落下する粒子の落下の動きをシミュレートしたいとしましょう。

粒子の位置と速度を保持する変更可能な参照領域を作成し、領域に格納された値を更新するのにforループ(`Control.Monad.Eff` の `forE` アクション)を使うことでこれを実現することができます。

```
import Control.Monad.Eff
import Control.Monad.ST

simulate :: forall eff h. Number -> Number -> Number -> Eff (st :: ST h | ef
simulate x0 v0 time = do
  ref <- newSTRef { x: x0, v: v0 }
  forE 0 (time * 1000) $ \i -> do
    modifySTRef ref (\o ->
      { v: o.v - 9.81 * 0.001
      , x: o.x + o.v * 0.001
      })
  return unit
final <- readSTRef ref
return final.x
```

計算の最後では、参照領域の最終的な値を読み取り、粒子の位置を返しています。

この関数に変更可能な状態を使っている、その参照区画 `ref` がプログラムの他の部分で使われるのが許されない限り、これは純粋な関数のままであることに注意してください。 `ST` 作用が禁止するものが正確には何であるのかについては後ほど見ます。

`ST` 作用で計算を実行するには、`runST` 関数を使用する必要があります。

```
runST :: forall a eff. (forall h. Eff (st :: ST h | eff) a) -> Eff eff a
```

ここで注目して欲しいのは、領域型 `h` が関数矢印の左辺にある括弧の内側で量化されているということです。 `runST` に渡したどんなアクションでも、任意の領域 `h` がなんであれ動作するということを意味しています。

しかしながら、ひとたび参照領域が `newSTRef` によって作成されると、その領域の型はすでに固定されており、`runST` によって限定されたコードの外側で参照領域を使おうとしても型エラーになるでしょう。 `runST` が安全に `ST` 作用を除去できるのはこれが理由なのです！

実際に、`ST` はこの例の唯一の作用なので、`runPure` と `runST` を併用すると `simulate` を純粋な関数に変えることができます、

```
simulate' :: Number -> Number -> Number -> Number
simulate' x0 v0 time = runPure (runST (simulate x0 v0 time))
```

`psci` でこの関数を実行してみてください。

```
> Main.simulate' 100 0 0.0
100.00

> Main.simulate' 100 0 1.0
95.10

> Main.simulate' 100 0 2.0
80.39

> Main.simulate' 100 0 3.0
55.87

> Main.simulate' 100 0 4.0
21.54
```

もし `simulate` の定義を `runST` の呼び出しのところへ埋め込むとすると、次のようになります。

```
simulate :: Number -> Number -> Number -> Number
simulate x0 v0 time = runPure (runST (do
  ref <- newSTRef { x: x0, v: v0 }
  forE 0 (time * 1000) $ \i -> do
    modifySTRef ref (\o ->
      { v: o.v - 9.81 * 0.001
      , x: o.x + o.v * 0.001
      })
    return unit
  final <- readSTRef ref
  return final.x))
```

参照区画はそのスコープから逃れることができないことが `psc` コンパイラにわかりますし、安全に `var` に変換することができます。 `runST` の呼び出しの本体に対して生成された JavaScript は次のようになります。

```
var ref = { x: x0, v: v0 };

Control_Monad_Eff.forE(0)(time * 1000)(function (i) {
  return function __do() {
    ref = (function (o) {
      return {
        v: o.v - 9.81 * 1.0e-3,
        x: o.x + o.v * 1.0e-3
      };
    })(ref);
    return Prelude.unit;
  };
})();

return ref.x;
```

局所的な変更可能状態を扱うとき、特に `Eff` モナドで効率のよいループを生成する `forE`、`foreachE`、`whileE`、`untilE` のようなアクションを一緒に使うときには、`ST` 作用は短い JavaScript を生成できる良い方法となります。

## 演習

1. (やや難しい) もし分母で分子を割り切れないなら `throwException` を使って例外を投げるように `safeDivide` 関数を書き直してください。

2. (難しい)  $\pi$ を推定するには次のような簡単な方法があります。単位正方形内にある多数の  $N$  個の点を無作為に選び、内接する円に含まれるものの個数  $n$  を数えます。このとき  $4n/N$  が円周率  $\pi$  の概算となります。 `forE` 関数、 `Random` 作用、 `ST` 作用を使って、この方法で円周率  $\pi$  を推定する関数を書いてください。

## 8.18 DOM 作用

この章の最後の節では、 `Eff` モナドでの作用についてこれまで学んだことを、実際のDOM操作の問題に応用します。

DOMを直接扱ったり、オープンソースのDOMライブラリを扱う、自由に利用可能なPureScriptパッケージが幾つかあります。

- `purescript-simple-dom` - JavaScript DOM APIのバインディング
- `purescript-jquery` - jQueryライブラリのバインディング
- `purescript-react` - Reactライブラリへのバインディング
- `purescript-angular` - AngularJSライブラリのバインディング
- `purescript-virtual-dom` - `virtual-dom`ライブラリの最小限のラップ

しかしながら、これらのライブラリのほとんどはまだ非常に新しくAPIが安定なため、この章の内容を安定させられるように、この章のソースコードの `Control.Monad.DOM` モジュールにはDOM要素を操作するための最小限の関数群が含まれています。

DOM要素を作成や操作をするための次のようなアクションが含まれています。

```
body :: forall eff. Eff (dom :: DOM | eff) Node
createElement :: forall eff. String -> Eff (dom :: DOM | eff) Node
querySelector :: forall eff. String -> Eff (dom :: DOM | eff) (Maybe Node)
```

既存の要素の内容やスタイルを変更するためのアクションも用意されています。

```
setText :: forall eff. String -> Node -> Eff (dom :: DOM | eff) Node
setInnerHTML :: forall eff. String -> Node -> Eff (dom :: DOM | eff) Node
appendChild :: forall eff. Node -> Node -> Eff (dom :: DOM | eff) Node
addClass :: forall eff. String -> Node -> Eff (dom :: DOM | eff) Node
```

そして、DOMイベントを処理するためのアクションがあります。

```
addEventListener :: forall eff. String ->
    Eff (dom :: DOM | eff) Unit ->
```

```
Node ->
Eff (dom :: DOM | eff) Node
```

これらが住所録アプリケーションのユーザインターフェイスを作るのに必要なアクションです。

## 8.19 住所録のユーザーインターフェース

これから構築しようとしているユーザ・インターフェースは、HTMLとPureScriptファイルに分かれています。HTMLはページ上の要素の配置を定義し、PureScriptのコードはフォームの動的な振る舞いを制御する方法を定義します。

まずは利用者が住所録に新しい項目を追加できるフォームを構築することにしましょう。フォームには、さまざまなフィールド(姓、名前、都市、州など)を入力するテキストボックス、および検証エラーが表示される領域が含まれます。テキストボックスに利用者がテキストを入力すると、検証エラーが更新されます。

シンプルさを保つために、フォームは固定の形状とします。電話番号は種類(自宅、携帯電話、仕事、その他)ごとに別々のテキストボックスへ分けることにします。

`head` 要素内の次のようなコードを除いて、HTMLファイルは完全に静的です。

```
<script type="text/javascript" src="../../dist/Main.js"></script>
<script type="text/javascript">
  onload = PS.Main.main;
</script>
```

最初の行では `psc` によって生成されるJavaScriptコードを読み込み、2行目ではページがロードされたときに `PS.Main.main` 関数が確実に実行されるようにしています。

`Main` モジュールはとても単純です。`Data.AddressBook.UI` モジュールには `setupEventHandlers` 関数に処理をそのまま移譲する `main` 関数だけが定義されています。

```
main :: forall eff. Eff (trace :: Trace, dom :: DOM | eff) Unit
main = do
  trace "Attaching event handlers"
  setupEventHandlers
```

これは混在した作用の一例になっていることに注目してください。下で見るよう



に、`trace` 関数は `Trace` 作用を使い、`setupEventHandlers` 関数は `Trace` 作用と `DOM` 作用の両方を使っています(`DOM` 作用は `Control.Monad.Eff.DOM` で定義されています)。

`setupEventHandlers` 関数もとても簡単です(単一の目的を持った小さな関数それぞれに分割することによって、コードについて理解するのが簡単になっていることに注目してください)。

```
setupEventHandlers :: forall eff. Eff (trace :: Trace, dom :: DOM | eff) Unit
setupEventHandlers = do
  -- Listen for changes on form fields
  body >>= addEventListener "change" validateAndUpdateUI
```

`setupEventHandlers` はまず文書の `body` への参照を取得するために `body` アクションを使い、`>>=` を使ってその結果を `addEventListener` アクションに渡しています。`addEventListener` は `change` イベントを監視して、イベントが発生するとその度に `validateAndUpdateUI` アクションを呼び出します。

`do` 記法の定義により、これを次のようにも書けることに注意してください。

```
setupEventHandlers = do
  -- Listen for changes on form fields
  b <- body
  addEventListener "change" validateAndUpdateUI b
```

どちらが読みやすいかどうかは個人の好みの問題です。前者は名前が付けられた関数の引数がなく、`point-free`形式の一例となっています。その一方で、後者では文書の `body` の名前として `b` が使われています。

`validateAndUpdateUI` アクションの役目は、フォーム検証器を実行し、必要に応じて利用者にエラーのリストを表示することです。この場合も、他の関数へ処理を委譲することによってこれを行います。最初に、`querySelector` アクションを使用してページの `validationErrors` 要素を選択しています。それから、その要素の内容を消去するために `setInnerHTML` アクションを使用しています。

```
validateAndUpdateUI :: forall eff. Eff (trace :: Trace, dom :: DOM | eff) Unit
validateAndUpdateUI = do
  Just validationErrors <- querySelector "#validationErrors"
  setInnerHTML "" validationErrors
```

次に `validateAndUpdateUI` が `validateControls` アクションを呼び出し、フォームの検証を実行しています。

```
errorsOrResult <- validateControls
```

後ほどすぐに見るように、`errorsOrResult` はエラーのリストか `Person` レコードのどちらかを表す型 `Either [String] Person` を持っています。

最後に、もし入力の検証に失敗すると、`validateAndUpdateUI` はページ上のエラーを表示するために `displayValidationErrors` アクションに処理を委譲します。

```
case errorsOrResult of
  Left errs -> displayValidationErrors errs
  Right result -> print result

return unit
```

検証が成功した場合、コードは単にコンソールに検証結果を出力します。当然のことながら、実際のアプリケーションでは、次の手順でデータベースまたは同様のものにデータを保存することになるでしょう。

`validateControls` 関数はより興味深いものです。`validateControls` の役割は、フォームの検証を実行し、成功または失敗のいずれかを示す結果を返すことであることを思い出してください。最初に行うことは、コンソールにデバッグメッセージを出力することです。

```
validateControls :: forall eff. Eff (trace :: Trace, dom :: DOM | eff)
                  (Either [String] Person)

validateControls = do
  trace "Running validators"
```

`Data.AddressBook.UI` モジュールでは、フォームフィールドから値を読み込む関数 `valueOf` が定義されています。ここでは型シグネチャだけを示し、実装については議論しません。

```
valueOf :: forall eff. String -> Eff (dom :: DOM | eff) String
```

`valueOf` はフォーム要素のIDをとり、利用者がそのテキストボックスに入力した値を返します。

次に、`validateControls` はページ上のフォームフィールドからいろいろな文字列を読み取って `Data.AddressBook.Person` データ構造体を構築します。

```
p <- person
  <$> valueOf "#inputFirstName"
  <*> valueOf "#inputLastName"
  <*> (address <$> valueOf "#inputStreet"
      <*> valueOf "#inputCity"
      <*> valueOf "#inputState")
  <*> sequence [ phoneNumber HomePhone <$> valueOf "#inputHomePhone"
                , phoneNumber CellPhone <$> valueOf "#inputCellPhone"
                ]
```

この計算では `person`、`address`、`phoneNumber` 関数を持ち上げるために、`Eff` を `Applicative` 関手として使用していることに注意してください。また、`Person` データ構造体の電話番号配列をまとめるために必要な `Eff` の配列を連鎖させるために、`Data.Traversable` の `sequence` 関数を使っています。

最後に、`validateControls` は前の章で書いた検証関数を実行し、その結果を返します。

```
return $ validatePerson' p
```

残りのコードは `displayValidationErrors` 関数です。`displayValidationErrors` はエラーの配列をとり、ページ上にそれらの文字列を出力します。

この関数が最初に行うことは、エラーを表示するための新しい `div` 要素を作成することです。フォームのレイアウトを制御するために [Bootstrap library](#) を使っているの  
で、`addClass` アクションを使って新しい要素に適切なCSSクラスを設定しています。

```
displayValidationErrors :: forall eff. [String] -> Eff (dom :: DOM | eff) Unit
displayValidationErrors errs = do
  alert <- createElement "div"
  >>= addClass "alert"
  >>= addClass "alert-danger"
```

このコードが `point-free` 形式であることに改めて注意してください。興味のある読者は、これを `>>=` を使わないように書き換えてみることをおすすめします。

次のコードは `ul` 要素を作成し先ほどの `div` に追加します。

```
ul <- createElement "ul"
ul `appendChild` alert
```

配列内の各エラーそれぞれについて `li` 要素を作成して、リストに追加します。`setText` ア

クシオンは、エラーメッセージを `li` 要素のテキストコンテンツを設定するために使用されています。

```
foreachE errs $ \err -> do
  li <- createElement "li" >>= setText err
  li `appendChild` ul
  return unit
```

配列の要素について繰り返しを行うために、このコードでは `foreachE` アクションを使っています。これは以前に見た `traverse` 関数に似ていますが、`Eff` モナドだけで使うように特殊化されています。

最後に、`querySelector` アクションを使って `validationErrors` 要素を検索し、それに先ほどの `div` を追加します。

```
Just validationErrors <- querySelector "#validationErrors"
  alert `appendChild` validationErrors

  return unit
```

以上です！ `grunt` を実行して、それからWebブラウザで `html/index.html` を開き、ユーザインターフェイスを試してみてください。

フォームフィールドにいろいろな値を入力すると、ページ上に出力された検証エラーを見ることができるでしょう。検証エラーをすべて修正すると、ブラウザのコンソール上に検証の結果が表示されるはずです。

このユーザインタフェースには明らかに改善すべき点がたくさんあります。演習ではアプリケーションがより使いやすくなるような方法を追究していきます。

## 演習

1. (簡単) このアプリケーションを変更し、職場の電話番号のテキストボックスを追加してください。
2. (やや難しい) 検証エラーを `ul` 要素を使ってリストで表示するかわりに、それぞれのエラーについてひとつずつ `alert` スタイルで `div` を作成するように、コードを変更してください。
3. (やや難しい) `>>=` の明示的な呼び出しを使わないように、`Data.AddressBook.UI` モジュールのコードを書き直してください。

4. (難しい、拡張) このユーザーインターフェイスの問題のひとつは、検証エラーがその発生源であるフォームフィールドの隣に表示されていないことです。コードを変更してこの問題を解決してください。

ヒント: 検証器によって返されるエラーの型は、エラーの原因となっているフィールドを示すために拡張する必要があります。次のようなエラー型を使用したくなるかもしれません。

```
data ValidationError = ValidationError String Field

data Field = FirstNameField
           | LastNameField
           | StreetField
           | CityField
           | StateField
           | PhoneField PhoneType
```

適切なフォーム要素を選択するように、`Field` を `querySelector` アクションの呼び出しに変更する関数を書く必要があるでしょう。

## 8.20 まとめ

この章ではPureScriptでの副作用の扱いについての多くの考え方を導入しました。

- `Monad` 型クラスと、それに関連する`do`記法の導入しました。
- モナド則を導入し、`do`記法使って書かれたコードを変換する方法を説明しました
- 異なる副作用で動作するコードを書くために、モナドを抽象的に扱う方法を説明しました。
- モナドが`Applicative`関手の一例であること、両者がどのように副作用のある計算を可能にするのか、2つの手法の違いを説明しました。
- ネイティブな作用の概念を定義し、ネイティブな副作用を処理するために使用する `Eff` モナドを導入しました。
- どのように `Eff` モナドが拡張可能作用を提供するか、複数の種類のネイティブな作用を同じ計算に混在させる方法を説明しました。
- 作用やレコードが種システムでどのように扱われるか、拡張可能なレコードと拡張可能作用の関連を見ました。
- 乱数生成、例外、コンソール入出力、変更可能な状態、およびDOM操作といった、さまざまな作用を扱うために `Eff` モナドを使いました。

`Eff` モナドは現実のPureScriptコードにおける基本的なツールです。本書ではこのあとも、

様々な場面で副作用を処理するために `Eff` モナドを使っています。

## 9 キャンバスグラフィックス

---

### 9.1 この章の目標

---

この章のコード例では、PureScriptでHTML5のCanvas APIを使用して2Dグラフィックスを生成する `purescript-canvas` パッケージに焦点をあててコードを拡張していきます。

### 9.2 プロジェクトの準備

---

このモジュールのプロジェクトでは、以下のBowerの依存関係が新しく追加されています。

- `purescript-canvas` - HTML5のCanvas APIのメソッドの型が定義されています。
- `purescript-refs` - 大域的な変更可能領域への参照を扱うための副作用を提供しています。

この章のソースコードは、それぞれに `main` メソッドが定義されている複数のモジュールへと分割されています。この章の節の内容はそれぞれ異なるファイルで実装されており、それぞれの節で対応するファイルの `main` メソッドを実行できるように、Gruntビルドターゲットを変更することで `Main` モジュールが変更できるようになっています。

HTMLファイル `html/index.html` には、各例で使用する単一の `canvas` 要素、およびコンパイルされたPureScriptコードを読み込む `script` 要素が含まれています。各節のコードをテストするには、ブラウザでこのHTMLファイルを開いてください。

### 9.3 単純な図形

---

`Rectangle.purs` ファイルにはキャンバスの中心に青い四角形をひとつ描画するという簡単な例が含まれています。このモジュールは、`Control.Monad.Eff` モジュールと、Canvas APIを扱うための `Eff` モナドのアクションが定義されている `Graphics.Canvas` モジュールをインポートします。

他のモジュールでも同様ですが、`main` アクションは最初に `getCanvasElementById` アクションを使ってCanvasオブジェクトへの参照を取得しています。また、`getContext2D` アクションを使ってキャンバスの2Dレンダリングコンテキストを参照しています。

```
main = do
  canvas <- getCanvasElementById "canvas"
  ctx <- getContext2D canvas
```

これらのアクションの型は `psci` を使うかドキュメントを見ると確認できます。

```
getCanvasElementById :: forall eff. String -> Eff (canvas :: Canvas | eff) CanvasE

getContext2D :: forall eff. CanvasElement -> Eff (canvas :: Canvas | eff) Co
```

`CanvasElement` と `Context2D` は `Graphics.Canvas` モジュールで定義されている型です。このモジュールでは、モジュール内のすべてのアクションで使用されている `Canvas` 作用も定義されています。

グラフィックスコンテキスト `ctx` は、キャンバスの状態を管理し、プリミティブな図形を描画したり、スタイルや色を設定したり、座標変換を適用するためのメソッドを提供しています。

`ctx` の取得に続けて、`setFillStyle` アクションを使って塗りのスタイルを青一色の塗りつぶしに設定しています。

```
setFillStyle "#0000FF" ctx
```

`setFillStyle` アクションがグラフィックスコンテキストを引数として取っていることに注意してください。これは `Graphics.Canvas` で共通のパターンです。

最後に、`fillPath` アクションを使用して矩形を塗りつぶしています。`fillPath` は次のような型を持っています。

```
fillPath :: forall eff a. Context2D ->
  Eff (canvas :: Canvas | eff) a ->
  Eff (canvas :: Canvas | eff) a
```

`fillPath` はグラフィックスコンテキストとレンダリングするパスを構築する別のアクションを引数にとります。パスは `rect` アクションを使うと構築することができます。`rect` はグラフィックスコンテキストと矩形の位置及びサイズを格納するレコードを引数にとります。

```
fillPath ctx $ rect ctx
  { x: 250
  , y: 250
  , w: 100
```

```
    , h: 100
  }
```

この長方形のコード例をビルドしましょう。

```
$ grunt rectangle
```

それでは `html/index.html` ファイルを開き、このコードによってキャンバスの中央に青い四角形が描画されていることを確認してみましょう。

## 9.4 行多相を利用する

パスを描画する方法は他にもあります。`arc` 関数は円弧を描画します。`moveTo` 関数、`lineTo` 関数、`closePath` 関数は細かい線分を組み合わせることでパスを描画します。

`Shapes.purs` ファイルでは長方形と円弧セグメント、三角形の、3つの図形を描画しています。

`rect` 関数は引数としてレコードをとることを見てきました。実際には、長方形のプロパティは型同義語で定義されています。

```
type Rectangle = { x :: Number
                  , y :: Number
                  , w :: Number
                  , h :: Number
                  }
```

`x` と `y` プロパティは左上隅の位置を表しており、`w` と `h` のプロパティはそれぞれ幅と高さを表しています。

`arc` 関数に以下のような型を持つレコードを渡して呼び出すと、円弧を描画することができます。

```
type Arc = { x      :: Number
            , y      :: Number
            , r      :: Number
            , start  :: Number
            , end    :: Number
            }
```



ここで、`x`と`y`プロパティは弧の中心、`r`は半径、`start`と`end`は弧の両端の角度を弧度法で表しています。

たとえば、次のコードは中心 `(300, 300)`、半径 `50` の円弧を塗りつぶします。

```
fillPath ctx $ arc ctx
  { x      : 300
  , y      : 300
  , r      : 50
  , start  : Math.pi * 5 / 8
  , end    : Math.pi * 2
  }
```

`Number` 型の `x` と `y` というプロパティが `Rectangle` レコード型と `Arc` レコード型の両方に含まれていることに注意してください。どちらの場合でもこの組は点を表しています。これは、いずれのレコード型にも適用できる、行多相な関数を書くことができることを意味します。

たとえば、`Shapes` モジュールでは `x` と `y` のプロパティを変更し図形を並行移動する `translate` 関数を定義されています。

```
translate :: forall r. Number -> Number ->
  { x :: Number, y :: Number | r } ->
  { x :: Number, y :: Number | r }
translate dx dy shape = shape
  { x = shape.x + dx
  , y = shape.y + dy
  }
```

この行多相型に注目してください。これは `triangle` が `x` と `y` というプロパティと、それに加えて他の任意のプロパティを持ったどんなレコードでも受け入れるということを言っています。`x` フィールドと `y` フィールドは更新されますが、残りのフィールドは変更されません。

これはレコード更新構文の例です。`shape { ... }` という式は、`shape` を元にして、括弧の中で指定されたように値が更新されたフィールドを持つ新たなレコードを作ります。波括弧の中の式はレコードリテラルのようなコロンではなく、等号でラベルと式を区切って書くことに意してください。

`Shapes` の例からわかるように、`translate` 関数は `Rectangle` レコードと `Arc` レコード双方に対して使うことができます。

`Shape` の例で描画される3つめの型は線分ごとのパスです。対応するコードは次のようになります。

```
setFillStyle "#FF0000" ctx
```

```
fillPath ctx $ do
  moveTo ctx 300 260
  lineTo ctx 260 340
  lineTo ctx 340 340
  closePath ctx
```

ここでは3つの関数が使われています。

- `moveTo` はパスの現在位置を指定された座標へ移動させます。
- `lineTo` は現在の位置と指定された座標の間に線分を描画し、現在の位置を更新します。
- `closePath` は開始位置と現在位置を結ぶ線分を描画し、パスを閉じます。

このコード片を実行すると、二等辺三角形を塗りつぶされます。

`shapes` ターゲットを使ってこの例をビルドしましょう。

```
$ grunt shapes
```

そしてもう一度 `html/index.html` を開き、結果を確認して下さい。キャンバスに3つの異なる型が描画されるはずです。

## 演習

1. (簡単) これまでの例のそれぞれについて、`strokePath` 関数や `setStrokeStyle` 関数を使ってみましょう。
2. (簡単) 関数の引数の内部でdo記法ブロックを使うと、`fillPath` 関数と `strokePath` 関数で共通のスタイルを持つ複雑なパスを描画することができます。同じ `fillPath` 呼び出しで隣り合った2つの矩形を描画するように、`Rectangle` のコード例を変更してみてください。線分と円弧を組み合わせて、円の扇形を描画してみてください。
3. (やや難しい) 次のような2次元の点を表すレコードが与えられたとします。

```
type Point = { x :: Number, y :: Number }
```

多数の点からなる閉じたパスを描く関数 `renderPath` 書いてください。

```
renderPath :: forall eff. Context2D -> [Point] ->
    Eff (canvas :: Canvas | eff) C
```

次のような関数を考えます。

```
f :: Number -> Point
```

この関数は引数として `1` から `0` の間の `Number` をとり、`Point` を返します。`renderPath` 関数を利用して関数 `f` のグラフを描くアクションを書いてください。そのアクションは有限個の点を `f` からサンプリングすることによって近似しなければなりません。

関数 `f` を変更し、異なるパスが描画されることを確かめてください。

## 9.5 無作為に円を描く

`Random.purs` ファイルには2種類の異なる副作用が混在した `Eff` モナドを使う例が含まれています。この例では無作為に生成された円をキャンバスに100個描画します。

`main` アクションはこれまでのようにグラフィックスコンテキストへの参照を取得し、ストロークと塗りつぶしスタイルを設定します。

```
setFillStyle "#FF0000" ctx
setStrokeStyle "#000000" ctx
```

次のコードでは `forE` アクションを使って `0` から `100` までの整数について繰り返しをしています。

```
forE 1 100 $ \_ -> do
```

それぞれの繰り返しでは、`do` 記法ブロックは3つの乱数を生成することから始まっています。

```
x <- random
y <- random
r <- random
```

これらの数は `0` から `1` の間に無作為に分布しています。これらはそれぞれ `x` 座標、`y` 座標、半径 `r` を表しています。

次のコードでこれらの変数に基づいて `Arc` を作成します。

```
let path = arc ctx
  { x      : x * 600
  , y      : y * 600
  , r      : r * 50
  , start  : 0
  , end    : Math.pi * 2
  }
```

そして最後に、現在のスタイルに従って円弧の塗りつぶしと線描が行われます。

```
fillPath ctx path
strokePath ctx path

return unit
```

`forE` に渡された関数が正しい型を持つようにするため、最後の行は必要であることに注意してください。

Gruntfileの `random` ターゲットを使用して、この例をビルドします。

```
$ grunt random
```

`html/index.html` を開いて、結果を確認してみましょう。

## 9.6 座標変換

キャンバスは簡単な図形を描画するだけのものではありません。キャンバスは変換行列を扱うことができ、図形は描画の前に形状を変形してから描画されます。図形は平行移動、回転、拡大縮小、および斜め変形することができます。

`purescript-canvas` ライブラリではこれらの変換を以下の関数で提供しています。

```
translate :: forall eff. TranslateTransform -> Context2D
--> Eff (canvas :: Canvas | eff)

rotate    :: forall eff. Number
--> Context2D

scale     :: forall eff. ScaleTransform
--> Context2D

transform :: forall eff. Transform
--> Eff (canvas :: Canvas | eff)
--> Context2D
```

```
-> Eff (canvas :: Canvas | eff)
```

`translate` アクションは `TranslateTransform` レコードのプロパティで指定した大きさだけ平行移動を行います。

`rotate` アクションは最初の引数で指定されたラジアンに応じて原点を中心とした回転を行います。

`scale` アクションは原点を中心として拡大縮小します。 `ScaleTransform` レコードは `x` 軸と `y` 軸に沿った拡大率を指定するのに使います。

最後の `transform` はこの4つのうちで最も一般的なアクションです。このアクションは行列に従ってアフィン変換を行います。

これらのアクションが呼び出された後に描画される図形は、自動的に適切な座標変換が適用されます。

実際には、これらの関数のそれぞれの作用は、コンテキストの現在の変換行列に対して変換行列を右から乗算していきます。つまり、もしある作用の変換をしていくと、その作用は実際には逆順に適用されていきます。次のような座標変換のアクションを考えてみましょう。

```
transformations ctx = do
  translate { translateX: 10, translateY } ctx
  scale { scaleX: 2, scaleY: 2 } ctx
  rotate (Math.pi / 2) ctx

  renderScene
```

このアクションの作用では、まずシーンが回転され、それから拡大縮小され、最後に平行移動されます。

## 9.7 コンテキストの保存

一般的な使い方としては、変換を適用してシーンの一部をレンダリングし、それからその変換を元に戻します。

Canvas APIにはキャンバスの状態のスタックを操作する `save` と `restore` メソッドが備わっています。 `purescript-canvas` ではこの機能を次のような関数でラップしています。

```
save    :: forall eff. Context2D -> Eff (canvas :: Canvas | eff) Context2D
restore :: forall eff. Context2D -> Eff (canvas :: Canvas | eff) Context2D
```

`save` アクションは現在のコンテキストの状態(現在の変換行列や描画スタイル)をスタックにプッシュし、`restore` アクションはスタックの一番上の状態をポップし、コンテキストの状態を復元します。

これらのアクションにより、現在の状態を保存し、いろいろなスタイルや変換を適用し、プリミティブを描画し、最後に元の変換と状態を復元することが可能になります。例えば、次の関数はいくつかのキャンバスアクションを実行しますが、その前に回転を適用し、そのあとに変換を復元します。

```
rotated ctx render = do
  save ctx
  rotate Math.pi ctx
  render
  restore ctx
```

こういったよくある使いかたの高階関数を利用した抽象化として、`purescript-canvas` ライブラリでは元のコンテキスト状態を維持しながらいくつかのキャンバスアクションを実行する `withContext` 関数が提供されています。

```
withContext :: forall eff a. Context2D ->
  Eff (canvas :: Canvas | eff) a ->
  Eff (canvas :: Canvas | eff) a
```

`withContext` を使うと、先ほどの `rotated` 関数を次のように書き換えることができます。

```
rotated ctx render = withContext ctx $ do
  rotate Math.pi ctx
  render
```

## 9.8 大域的な変更可能状態

この節では `purescript-refs` パッケージを使って `Eff` モナドの別の作用について実演してみます。

`Control.Monad.Eff.Ref` モジュールでは大域的に変更可能な参照のための型構築子、および関連する作用を提供します。

```
> :i Control.Monad.Eff.Ref
```

```
> :k RefVal
* -> *

> :k Ref
!
```

型 `RefVal a` の値は型 `a` 値を保持する変更可能な領域への参照で、前の章で見た `STRef h a` によく似ています。その違いは、`ST` 作用は `runST` を用いて除去することができますが、`Ref` 作用はハンドラを提供しないということです。`ST` は安全に局所的な状態変更を追跡するために使用されますが、`Ref` は大域的な状態変更を追跡するために使用されます。そのため、`Ref` は慎重に使用する必要があります。

`Refs.purs` ファイルには `canvas` 要素上のマウスクリックを追跡するのに `Ref` 作用を使用する例が含まれています。

このコードでは最初に `newRef` アクションを使って値 `0` で初期化された領域への新しい参照を作成しています。

```
clickCount <- newRef 0
```

クリックイベントハンドラの内部では、`modifyRef` アクションを使用してクリック数を更新しています。

```
modifyRef clickCount (\count -> count + 1)
```

`readRef` アクションは新しいクリック数を読み取るために使われています。

```
count <- readRef clickCount
```

`render` 関数では、クリック数に応じて変換を矩形に適用しています。

```
withContext ctx $ do
  let scaleX = Math.sin (count * Math.pi / 4) + 1.5
  let scaleY = Math.sin (count * Math.pi / 6) + 1.5

  translate { translateX: 300, translateY: 300 } ctx
  rotate (count * Math.pi / 18) ctx
  scale { scaleX: scaleX, scaleY: scaleY } ctx
  translate { translateX: -100, translateY: -100 } ctx
```

```
fillPath ctx $ rect ctx
  { x: 0
    , y: 0
    , w: 200
    , h: 200
  }
```

このアクションでは元の変換を維持するために `withContext` を使用しており、それから続く変換を順に適用しています(変換が下から上に適用されることを思い出してください)。

- 中心が原点に来るように、矩形を `(-100, -100)` 平行移動します。
- 矩形を原点を中心に拡大縮小します。
- 矩形を原点を中心に `10` 度の倍数だけ回転します。
- 中心がキャンバスの中心に位置するように長方形を `(300, 300)` だけ平行移動します。

このコード例をビルドしてみましょう。

```
$ grunt refs
```

`html/index.html` ファイルを開いてみましょう。何度かキャンバスをクリックすると、キャンバスの中心の周りを回転する緑の四角形が表示されるはずです。

## 演習

1. (簡単) パスの線描と塗りつぶしを同時に行う高階関数を書いてください。その関数を使用して `Random.purs` 例を書きなおしてください。
2. (やや難しい) `Random` 作用と `DOM` 作用を使用して、マウスがクリックされたときにキャンバスに無作為な位置、色、半径の円を描画するアプリケーションを作成してください。
3. (やや難しい) シーンを指定された座標を中心に回転する関数を書いてください。  
ヒント: 最初にシーンを 原点まで平行移動しましょう。

## 9.9 L-Systems

この章の最後の例として、`purescript-canvas` パッケージを使用してL-systems(Lindenmayer systems)を描画する関数を記述します。

L-Systemsはアルファベット、つまり初期状態となるアルファベットの文字列と、生成規則の



集合で定義されています。各生成規則は、アルファベットの文字をとり、それを置き換える文字の配列を返します。この処理は文字の初期配列から始まり、複数回繰り返されます。

もしアルファベットの各文字がキャンバス上で実行される命令と対応付けられていれば、その指示に順番に従うことでL-Systemsを描画することができます。

たとえば、アルファベットが文字 **L** (左回転)、**R** (右回転)、**F** (前進)で構成されていたとします。また、次のような生成規則を定義します。

```
L -> L
    R -> R
    F -> FLFRRFLF
```

配列 "FRRFRRFRR" から始めて処理を繰り返すと、次のような経過を辿ります。

```
FRRFRRFRR
    FLFRRFLFRRFLFRRFLFRRFLFRR
    FLFRRFLFLFLFRRFLFRRFLFRRFLFLFLFRRFLFRRFLFRRFLF...
```

この命令群に対応する線分パスをプロットすると、コッホ曲線と呼ばれる曲線に近似します。反復回数を増やすと、曲線の解像度が増加していきます。

それでは型と関数の言語へとこれを翻訳してみましょう。

アルファベットの選択肢は型の選択肢によって表すことができます。今回の例では、以下のような型で定義することができます。

```
data Alphabet = L | R | F
```

このデータ型では、アルファベットの各文字ごとに1つずつデータ構築子が定義が定義されています。

文字の初期配列はどのように表したらいいのでしょうか。単なるアルファベットの配列でもいいでしょう。これを **Sentence** と呼ぶことにします。

```
type Sentence = [Alphabet]

initial :: Sentence
initial = [F, R, R, F, R, R, F, R, R]
```

生成規則は `Alphabet` から `Sentence` への関数として表すことができます。

```
productions :: Alphabet -> Sentence
productions L = [L]
productions R = [R]
productions F = [F, L, F, R, R, F, L, F]
```

これはまさに上記の仕様をそのまま書き写したものです。

これで、この形式の仕様を受け取りキャンバスに描画する関数 `lsystem` を実装することができます。 `lsystem` はどのような型を持っているべきでしょうか。この関数は初期状態 `initial` と生成規則 `productions` のような値だけでなく、アルファベットの文字をキャンバスに描画する関数を引数に取る必要があります。

`lsystem` の型の最初の大まかな設計としては、次のようになるかもしれません。

```
forall eff. Sentence ->
    (Alphabet -> Sentence) ->
    (Alphabet -> Eff (canvas :: Canvas | eff) Unit) ->
    Number ->
    Eff (canvas :: Canvas | eff) Unit
```

最初の2つの引数の型は、値 `initial` と `productions` に対応しています。

3番目の引数は、アルファベットの文字を取り、キャンバス上のいくつかのアクションを実行することによって翻訳する関数を表します。この例では、文字 `L` は左回転、文字 `R` で右回転、文字 `F` は前進を意味します。

最後の引数は、実行したい生成規則の繰り返し回数を表す数です。

最初に気づくことは、現在の `lsystem` 関数は `Alphabet` 型だけで機能しますが、どんなアルファベットについても機能すべきですから、この型はもっと一般化されるべきです。それでは、量子化された型変数 `a` について、`Alphabet` と `Sentence` を `a` で置き換えましょう。

```
forall a eff. [a] ->
    (a -> [a]) ->
    (a -> Eff (canvas :: Canvas | eff) Unit) ->
    Number ->
    Eff (canvas :: Canvas | eff) Unit
```

次に気付くこととしては、「左回転」と「右回転」のような命令を実装するためには、いくつかの状態を管理する必要があります。具体的に言えば、その時点でパスが向いている方向を

状態として持たなければなりません。計算を通じて状態を関数に渡すように変更する必要があります。ここでも `lsystem` 関数は状態がどんな型でも動作しなければなりませんから、型変数 `s` を使用してそれを表しています。

型 `s` を追加する必要があるのは3箇所、次のようになります。

```
forall a s eff. [a] ->
    (a -> [a]) ->
    (s -> a -> Eff (canvas :: Canvas | eff) s) ->
    Number ->
    s ->
    Eff (canvas :: Canvas | eff) s
```

まず追加の引数の型として `lsystem` に型 `s` が追加されています。この引数はL-Systemの初期状態を表しています。

型 `s` は引数にも現れますが、翻訳関数(`lsystem` の第3引数)の戻り値の型としても現れます。翻訳関数は今のところ、引数としてL-Systemの現在の状態を受け取り、戻り値として更新された新しい状態を返します。

この例の場合では、次のような型を使って状態を表す型を定義することができます。

```
type State =
    { x :: Number
    , y :: Number
    , theta :: Number
    }
```

プロパティ `x` と `y` はパスの現在の位置を表しており、プロパティ `theta` は現在の向きを表しており、ラジアンで表された水平線に対するパスの角度です。

システムの初期状態としては次のようなものが考えられます。

```
initialState :: State
initialState = { x: 120, y: 200, theta: 0 }
```

それでは、`lsystem` 関数を実装してみます。定義はとても単純であることがわかるでしょう。

`lsystem` は第4引数の値(型 `Number`)に応じて再帰するのが良さそうです。再帰の各ステップでは、生成規則に従って状態が更新され、現在の文が変化していきます。このことを念頭に置きつつ、まずは関数の引数の名前を導入して、補助関数に処理を移譲することから

始めましょう。

```
lsystem :: forall a s eff. [a] ->
    (a -> [a]) ->
    (s -> a -> Eff (canvas :: Canvas | eff) s) ->
    Number ->
    s ->
    Eff (canvas :: Canvas | eff) s
lsystem init prod interpret n state = go init n
where
```

`go` 関数は第2引数に応じて再帰することで動きます。`n` がゼロであるときと `n` がゼロでないときの2つの場合で分岐します。

`n` がゼロの場合では再帰は完了し、解釈関数に応じて現在の文を解釈します。ここでは引数として与えられている型 `[a]` の文、型 `s` の状態、型 `s -> a -> Eff (canvas :: Canvas | eff) s` の関数を参照することができます。これらの引数の型を考えると、以前定義した `foldM` の呼び出しにちょうど対応していることがわかります。`foldM` は `purescript-control` パッケージでも定義されています。

```
go s 0 = foldM interpret state s
```

ゼロでない場合ではどうでしょうか。その場合は、単に生成規則を現在の文のそれぞれの文字に適用して、その結果を連結し、そしてこの処理を再帰します。

```
go s n = go (concatMap prod s) (n - 1)
```

これだけです！ `foldM` や `concatMap` のような高階関数を使うと、このようにアイデアを簡潔に表現することができるのです。

しかし、まだ完全に終わったわけではありません。ここで与えた型は、実際はまだ特殊化されすぎています。この定義ではキャンバスの操作が実装のどこにも使われていないことに注目してください。それに、まったく `Eff` モナドの構造を利用していません。実際には、この関数はどんなモナド `m` についても動作するのです！

この章に添付されたソースコードで定義されている、`lsystem` のもっと一般的な型は次のようになっています。

```
lsystem :: forall a m s. (Monad m) =>
    [a] ->
```

```
(a -> [a]) ->
(s -> a -> m s) ->
Number ->
s ->
m s
```

この型が言っているのは、この翻訳関数はモナド `m` で追跡される任意の副作用をまったく自由に持つことができる、ということだと理解することができます。キャンバスに描画したり、またはコンソールに情報を出力するかもしれませんし、失敗や複数の戻り値に対応しているかもしれません。こういった様々な型の副作用を使ったL-Systemを記述してみることを読者にお勧めします。

この関数は実装からデータを分離することの威力を示す良い例となっています。この手法の利点は、複数の異なる方法でデータを解釈する自由が得られることです。`lssystem` は2つの小さな関数へと分解することができるかもしれません。ひとつめは `concatMap` の適用の繰り返しを使って文を構築するもので、ふたつめは `foldM` を使って文を翻訳するものです。これは読者の演習として残しておきます。

それでは翻訳関数を実装して、この章の例を完成させましょう。`lssystem` の型は型シグネチャが言っているのは、翻訳関数の型は、何らかの型 `a` と `s`、型構築子 `m` について、`s -> a -> m s` でなければならないということです。今回は `a` を `Alphabet`、`s` を `State`、モナド `m` を `Eff (canvas :: Canvas)` というように選びたいということがわかっています。これにより次のような型になります。

```
interpret :: State -> Alphabet -> Eff (canvas :: Canvas) State
```

この関数を実装するには、`Alphabet` 型の3つのデータ構築子それぞれについて処理する必要があります。文字 `L` (左回転)と `R` (右回転)の解釈では、`theta` を適切な角度へ変更するように状態を更新するだけです。

```
interpret state L = return $ state { theta = state.theta - Math.pi / 3 }
interpret state R = return $ state { theta = state.theta + Math.pi / 3 }
```

文字 `F` (前進)を解釈するには、パスの新しい位置を計算し、線分を描画し、状態を次のように更新します。

```
interpret state F = do
  let x' = state.x + Math.cos state.theta * 1.5
      y' = state.y + Math.sin state.theta * 1.5
  moveTo ctx state.x state.y
```

```
lineTo ctx x' y'
return { x: x', y: y', theta: state.theta }
```

この章のソースコードでは、名前 `ctx` を参照できるようにするために、`interpret` 関数は `main` 関数内で `let` 束縛を使用して定義されていることに注意してください。`State` 型がコンテキストを持つように変更することは可能ですが、それはこのシステムの状態の変化部分ではないので不適切でしょう。

このL-Systemsを描画するには、次のような `strokePath` アクションを使用するだけです。

```
strokePath ctx $ lsystem initial productions interpret 5 initialState
```

`grunt lsystem` を使ってL-Systemをコンパイルし、`html/index.html` を開いてみましょう。キャンバスにコッホ曲線が描画されるのがわかると思います。

## 演習

1. (簡単) `strokePath` の代わりに `fillPath` を使用するように、上のL-Systemsの例を変更してください。ヒント: `closePath` の呼び出しを含め、`moveTo` の呼び出しを `interpret` 関数の外側に移動する必要があります。
2. (簡単) 描画システムへの影響を理解するために、コード中の様々な数値の定数を変更してみてください。
3. (やや難しい) `lsystem` 関数を2つの小さな関数に分割してください。ひとつめは `concatMap` の適用の繰り返しを使用して最終的な結果を構築するもので、ふたつめは `foldM` を使用して結果を解釈するものでなくてはなりません。
4. (やや難しい) `setShadowOffsetX` アクション、`setShadowOffsetY` アクション、`setShadowBlur` アクション、`setShadowColor` アクションを使い、塗りつぶされた図形にドロップシャドウを追加してください。ヒント: `psci` を使って、これらの関数の型を調べてみましょう。
5. (やや難しい) 向きを変えるときに角度の大きさは今のところ一定(`pi/3`)です。その代わりに、`Alphabet` データ型の中に角度の大きさを追加して、生成規則によって角度を変更できるようにしてください。

```
type Angle = Number
```

```
data Alphabet = L Angle | R Angle | F Angle
```

生成規則でこの新しい情報を使うと、どんな面白い図形を作ることができるでしょうか。

6. (難しい) **L** (60度左回転)、**R** (60度右回転)、**F** (前進)、**M** (これも前進)という4つの文字からなるアルファベットでL-Systemが与えられたとします。

このシステムの文の初期状態は、単一の文字 **M** です。

このシステムの生成規則は次のように指定されています。

```
L -> L
R -> R
F -> FLMLFRMRFRMRFLMLF
M -> MRFRMLFLMLFLMRFRM
```

このL-Systemを描画してください。注意:最後の文のサイズは反復回数に従って指数関数的に増大するので、生成規則の繰り返しの回数を削減することが必要になります。

ここで、生成規則における **L** と **M** の間の対称性に注目してください。ふたつの「前進」命令は、次のようなアルファベット型を使用すると、**Boolean** 値を使って区別することができます。

```
data Alphabet = L | R | F Boolean
```

このアルファベットの表現を使用して、もう一度このL-Systemを実装してください。

7. (難しい) 翻訳関数で別のモナド **m** を使ってみましょう。**Trace** 作用を利用してコンソール上にL-Systemを出力したり、**Random** 作用を利用して状態の型に無作為の突然変異を適用したりしてみてください。

## 9.10 まとめ

この章では、**purescript-canvas** ライブラリを使用することにより、PureScriptからHTML5 Canvas APIを使う方法について学びました。マップや畳み込み、レコードと行多型、副作用を扱うための **Eff** モナドなど、これまで学んできた手法を利用した実用的な例について多く見ました。

この章の例では、高階関数の威力を示すとともに、実装からデータを分離も実演しました。



これは例えば、代数データ型を使用すると、これらの概念を次のように拡張し、描画関数からシーンの表現を完全に分離できるようになります。

```
data Scene = Rect Rectangle
          | Arc Arc
          | PiecewiseLinear [Point]
          | Transformed Transform Scene
          | Clipped Rectangle Scene
          | ...
```

この手法は `purescript-drawing` パッケージでも採用されており、描画前にさまざまな方法でデータとしてシーンを操作することができるという柔軟性をもたらせています。

次の章では、PureScriptの外部関数インタフェース(foreign function interface)を使って、既存のJavaScriptの関数をラップした `purescript-canvas` のようなライブラリを実装する方法について説明します。

## 10 外部関数インタフェース

### 10.1 この章の目標

この章では、PureScriptコードからJavaScriptコードへの呼び出し、およびその逆を可能にする、PureScriptの外部関数インタフェース(foreign function interface, FFI)を紹介します。これから扱うのは次のようなものです。

- PureScriptから純粋なJavaScript関数を呼び出す方法
- 既存のJavaScriptコードに基づいて、作用型と `Eff` モナドと一緒に使用する新しいアクションを作成する方法
- JavaScriptからPureScriptコードを呼び出す方法
- 実行時のPureScriptの値の表現を知る方法
- `purescript-foreign` パッケージを使用して型付けされていないデータを操作する方法

この章の終わりにかけて、再び住所録のコード例について検討します。この章の目的は、FFIを使ってアプリケーションに次のような新しい機能を追加することです。

- ポップアップ通知でユーザーに警告する
- フォームのデータを直列化してブラウザのローカルストレージに保存し、アプリケーションが再起動したときにそれを再読み込みする



## 10.2 プロジェクトの準備

---

このモジュールのソースコードは、第7章及び第8章の続きになります。今回もそれぞれのディレクトリから適切なソースファイルがGruntfileに含められています。

この章では型付けされていないデータを操作するためのデータ型と関数を提供する `purescript-foreign` ライブラリというBower依存関係がひとつ新しく追加されます。

新しいNPM依存関係もあります。この章のGruntfileは、`grunt-contrib-connect` パッケージを使用してコンパイル後に静的ファイルサーバを実行するようになっています。これは、ウェブページがローカルファイルから配信されているときに起こる、ローカルストレージとブラウザ固有の問題を避けるためです。この章の例を実行するには、まず `grunt` を実行して、それからブラウザで <http://localhost:8000/> を開いてください。

## 10.3 免責事項

---

JavaScriptとの共同作業をできる限り簡単にするため、PureScriptは単純な多言語関数インタフェースを提供します。しかしながら、FFIはPureScriptの高度な機能であることには留意していただきたいと思います。FFIを安全かつ効率的に使用するには、扱うつもりであるデータの実行時の表現についてよく理解していなければなりません。この章では、PureScriptの標準ライブラリのコードに関連する、そのような理解を与えることを目指しています。

PureScriptのFFIはとても柔軟に設計されています。実際には、外部関数に最低限の型だけを与えるか、それとも型システムを利用して外部のコードの誤った使い方を防ぐようにするか、開発者が選ぶことができるということを意味しています。標準ライブラリのコードは、後者の手法を好む傾向にあります。簡単な例としては、JavaScriptの関数で戻り値が `null` をされないことを保証することはできません。実のところ、既存のJavaScriptコードはかなり頻繁に `null` を返します！しかし、PureScriptの型は通常null値を持っていません。そのため、FFIを使ってJavaScriptコードのインターフェイスを設計するときは、これらの特殊な場合を適切に処理するのは開発者の責任です。

## 10.4 JavaScriptからPureScriptを呼び出す

---

少なくとも単純な型を持った関数については、JavaScriptからPureScript関数を呼び出すのはとても簡単です。

例として以下のような簡単なモジュールを見てみましょう。

```
module Test where
```

```
gcd :: Number -> Number -> Number
gcd 0 m = m
gcd n 0 = n
gcd n m | n > m = gcd (n - m) m
gcd n m = gcd (m - n) n
```

この関数は、減算を繰り返すことによって2つの数の最大公約数を見つけます。関数を定義するのにPureScriptを使いたくなるかもしれない良い例となっていますが、JavaScriptからそれ呼び出すためには条件があります。PureScriptでパターン照合と再帰を使用してこの関数を定義するのは簡単で、実装する開発者は型検証器の恩恵を受けることができます。

このモジュールを `psc` で次のようにコンパイルし、結果のJavaScriptを `Node` にロードしてみましょう。

```
$ psc Test.purs > Test.js
```

```
$ node Test.js
```

この関数をJavaScriptから呼び出す方法を理解するには、PureScriptの関数は常に引数がひとつのJavaScript関数へと変換され、引数へは次のようにひとつずつ適用していかなければならないことを理解するのが重要です。

```
> var test = PS.Test.gcd(15)(20);
```

`Test` モジュールはグローバルな `PS` オブジェクトのメンバ `Test` へとコンパイルされることに注意してください。これは `psc` コンパイラのデフォルトの動作ですが、グローバル名前空間は次のようにコマンドラインオプションを使用して変更することができます。

```
$ psc Test.purs --browser-namespace=MyNamespace > Test.js
```

`psc-make` を使用してCommonJSのモジュールにコードをコンパイルすると、コンパイルされたモジュールは、デフォルトでは `output` フォルダに配置されます。生成されたこれらのモジュールを `node_modules` ディレクトリにコピーすると、NodeJS(もしくはその他のCommonJS互換環境)の `require` 関数を使用して、モジュールを参照できるようになります。

```
var Test = require('Test');
```

このモジュールで定義された関数は、先ほどと同様に使うことができます。

```
Test.gcd(15)(20);
```

## 10.5 名前の生成を理解する

PureScriptはコード生成時にできるだけ名前を保存することを目的としています。具体的には、トップレベルでの宣言では、JavaScriptのキーワードでなければ任意の識別子が保存されます。

識別子としてJavaScriptの予約語を使う場合は、名前はダブルダラー記号でエスケープされます。たとえば、次のPureScriptコードを考えてみます。

```
null = []
```

これは以下のようなJavaScriptへコンパイルされます。

```
var $$null = [];
```

また、識別子に特殊文字を使用したい場合は、単一のドル記号を使用してエスケープされます。たとえば、このPureScriptコードを考えます。

```
example' = 100
```

これは以下のJavaScriptにコンパイルされます。

```
var example$prime = 100;
```

この方式は、ユーザー定義の中置演算子の名前を生成するためにも使用されます。

```
(%) a b = ...
```

これは次のようにコンパイルされます。

```
var $percent = ...
```

コンパイルされたPureScriptコードがJavaScriptから呼び出されることを意図している場合、識別子は英数字のみを使用し、JavaScriptの予約語を避けることをお勧めします。ユーザ定義演算子がPureScriptコードでの使用のために提供される場合でも、JavaScriptから使うための英数字の名前を持った代替関数を提供しておくことをお勧めします。

## 10.6 実行時のデータ表現

---

型はプログラムはある意味で「正しい」ことをコンパイル時に判断できるようにします。つまり、実行時には中断されません。しかし、これは何を意味するのでしょうか？PureScriptでは式の型は実行時の表現と互換性がなければならないことを意味します。

そのため、PureScriptとJavaScriptコードを一緒に効率的に使用できるように、実行時のデータ表現について理解することが重要です。これは、与えられた任意のPureScriptの式について、その値が実行時にどのように評価されるかという挙動を理解できるべきであることを意味しています。

PureScriptの式は、実行時に特に単純な表現を持っているということは朗報です。実際に標準ライブラリのコードについて、その型を考慮すれば式の実行時のデータ表現を把握することが可能です。

単純な型については、対応関係はほとんど自明です。たとえば、式が型 `Boolean` を持っていれば、実行時のその値 `v` は `typeof v === 'boolean'` を満たします。つまり、型 `Boolean` の式は `true` もしくは `false` のどちらか一方の(JavaScriptの)値へと評価されます。実のところ、`null` や `undefined` に評価される、型 `Boolean` のPureScriptの式はありません。

`Number` と `String` の型の式についても同様のことが成り立ちます。`Number` 型の式は `null` でないJavaScriptの数へと評価されますし、`String` 型の式は `null` でないJavaScriptの文字列へと評価されます。

もっと複雑な型についてはどうでしょうか？

すでに見てきたように、PureScriptの関数は引数がひとつのJavaScriptの関数に対応しています。厳密に言えば、任意の型 `a`、`b` について、式 `f` の型が `a -> b` で、式 `x` が型 `a` についての適切な実行時表現の値へと評価されるなら、`f` はJavaScriptの関数へと評価され、`x` を評価した結果に `f` を適用すると、それは型 `b` の適切な実行時表現を持ちます。簡単な例としては、`String -> String` 型の式は、`null` でないJavaScript文字列から `null` でないJavaScript文字列への関数へと評価されます。

ご想像のとおり、PureScriptの配列はJavaScriptの配列に対応しています。しかし、PureScriptの配列は均質であり、つまりすべての要素が同じ型を持っていることは覚えてお

いてください。具体的には、もしPureScriptの式 `e` が任意の型 `a` について型 `[a]` を持っているなら、`e` はすべての要素が型 `a` の適切な実行時表現を持った (`null` でない)JavaScript配列へと評価されます。

PureScriptのレコードがJavaScriptのオブジェクトへと評価されることはすでに見てきました。ちょうど関数と配列の場合のように、そのラベルに関連付けられている型を考慮すれば、レコードのフィールドのデータの実行時の表現についても推論することができます。もちろん、レコードのそれぞれのフィールドは、同じ型である必要はありません。

## 10.7 代数的データ型の実行時表現

PureScriptコンパイラは、代数的データ型のすべての構築子についてそれぞれ関数を定義し、新たなJavaScriptオブジェクト型を作成します。これらの構築子はこれらのプロトタイプに基づいて新しいJavaScriptオブジェクトを作成する関数に対応しています。

たとえば、次のような単純な代数的データ型を考えてみましょう。

```
data ZeroOrOne a = Zero | One a
```

PureScriptコンパイラは、次のようなコードを生成します。

```
function One(value0) {
    this.value0 = value0;
};

One.create = function (value0) {
    return new One(value0);
};

function Zero() {
};

Zero.value = new Zero();
```

ここで2つのJavaScriptオブジェクト型 `Zero` と `One` を見てください。JavaScriptの予約語 `new` を使用すると、それぞれの型の値を作成することができます。引数を持つ構築子については、コンパイラは `value0`、`value1` などと呼ばれるフィールドに対応するデータを格納します。

PureScriptコンパイラは補助関数も生成します。引数のない構築子については、コンパイラは構築子が使われるたびに `new` 演算子を使うのではなく、データを再利用できるよう

に `value` プロパティを生成します。ひとつ以上の引数を持つ構築子では、適切な表現を持つ引数を取り適切な構築子を適用する `create` 関数をコンパイラは生成します。

2引数以上の構築子についてはどうでしょうか？ その場合でも、PureScriptコンパイラは新しいオブジェクト型と補助関数を作成します。しかし今回は、補助関数は2引数のカーリー化された関数です。たとえば、次のような代数的データ型を考えます。

```
data Two a b = Two a b
```

このコードからは、次のようなJavaScriptコードを生成されます。

```
function Two(value0, value1) {
  this.value0 = value0;
  this.value1 = value1;
};

Two.create = function (value0) {
  return function (value1) {
    return new Two(value0, value1);
  };
};
```

ここで、オブジェクト型 `Two` の値は予約語 `new` または `Two.create` 関数を使用すると作成することができます。

newtypeの場合はまた少し異なります。newtypeは単一の引数を取る単一の構築子を持つよう制限された代数的データ型であることを思い出してください。この場合には、実際はnewtypeの実行時表現は、その引数の型と同じになります。

例えば、電話番号を表す次のようなnewtypeを考えます。

```
newtype PhoneNumber = PhoneNumber String
```

これは実行時にはJavaScriptの文字列として表されます。newtypeは型安全性の追加の層を提供しますが、実行時の関数呼び出しのオーバーヘッドがないので、ライブラリを設計するのに役に立ちます。

## 10.8 量化された型の実行時表現

量化された型(多相型)の式は、制限された表現を実行時に持っています。実際には、量化

された型の式が比較的少数与えられたとき、とても効率的に解決できることを意味しています。

例えば、次の多相型を考えてみます。

```
forall a. a -> a
```

この型を持っている関数にはどんなものがあるでしょうか。少なくともひとつはこの型を持つ関数が存在しています。すなわち、`Prelude` で定義されている恒等関数 `id` です。

```
id :: forall a. a -> a
id a = a
```

実際には、`id` の関数はこの型の唯一の(全)関数です！これは間違いなさそうに見えます(この型を持った `id` とは明らかに異なる式を書こうとしてみてください)が、しかし、これを確かめるにはどうしたらいいのでしょうか。これは型の実行時表現を考えることによって確認することができます。

量化された型 `forall a. t` の実行時表現とは何でしょうか。さて、この型の実行時表現を持つ任意の式は、型 `a` をどのように選んでも型 `t` の適切な実行時表現を持っていないけません。上の例では、型 `forall a. a -> a` の関数は、`String -> String`、`Number -> Number`、`[Boolean] -> [Boolean]` などといった型について、適切な実行時表現を持っていないけません。これらは、数から数、文字列から文字列の関数でなくてはなりません。

しかし、それだけでは十分ではありません。量化された型の実行時表現は、これよりも更に厳しくなります。任意の式がパラメトリックに多相的でなければなりません。つまり、その実装において、引数の型についてのどんな情報も使うことができないのです。この追加の条件は、考えられる多相型のうち、次のようなJavaScriptの関数として問題のある実装を禁止します。

```
function invalid(a) {
  if (typeof a === 'string') {
    return "Argument was a string.";
  } else {
    return a;
  }
}
```

確かにこの関数は文字列から文字列、数から数へというような関数ではありますが、追加の



条件を満たしていません。引数の実行時の型を調べているからです。したがって、この関数は型 `forall a. a -> a` の正しい実装だとはいえないのです。

関数の引数の実行時の型を検査することができなければ、唯一の選択肢は引数をそのまま返すことだけであり、したがって `id` は、`forall a. a -> a` のまったく唯一の実装なのです。

パラメトリック多相(parametric polymorphism)とパラメトリック性(parametricity)についての詳しい議論は本書の範囲を超えています。しかしながら、PureScriptの型は、実行時に消去されているので、PureScriptの多相関数は(FFIを使わない限り)引数の実行時表現を検査することができないし、この多相的なデータの表現は適切であることに注意してください。

## 10.9 制約された型の実行時表現

型クラス制約を持つ関数は、実行時に面白い表現を持っています。関数の振る舞いはコンパイラによって選ばれた型クラスのインスタンスに依存する可能性があるため、関数には選択したインスタンスから提供された型クラスの関数の実装が含まれた型クラス辞書(type class dictionary)と呼ばれる追加の引数を与えられています。

例えば、`Show` 型クラスを使用している制約された型を持つ、次のような単純なPureScript関数について考えます。

```
shout :: forall a. (Show a) => a -> String
shout a = show a ++ "!!!"
```

このコードから生成されるJavaScriptは次のようになります。

```
var shout = function (dict) {
  return function (a) {
    return show(dict)(a) + "!!!";
  };
};
```

`shout` は1引数ではなく、2引数の(カーリー化された)関数にコンパイルされていることに注意してください。最初の引数 `dict` は `Show` 制約の型クラス辞書です。`dict` には型 `a` の `show` 関数の実装が含まれています。

最初の引数として明示的にPreludeの型クラス辞書を渡すと、JavaScriptからこの関数を呼び出すことができます。



```
shout(Prelude.showNumber())(42);
```

## 演習

1. (簡単) これらの型の実行時の表現は何でしょうか。

```
forall a. a
  forall a. a -> a -> a
  forall a. (Ord a) => [a] -> Boolean
```

これらの型を持つ式についてわかることはなんでしょうか。

2. (やや難しい) `psc-make` を使ってコンパイルし、NodeJSの `require` 関数を使ってモジュールをインポートすることで、JavaScriptから `purescript-arrays` ライブラリの関数を使ってみてください。

## 10.10 PureScriptからのJavaScriptコードを使う

PureScriptからJavaScriptコードを使用する最も簡単な方法は、`foreign import` 宣言を使用し、既存のJavaScriptの値に型を与えることです。

たとえば、特殊文字をエスケープすることによりURIのコンポーネントを符号化するJavaScriptの `encodeURIComponent` 関数について考えてみます。

```
$ node
```

```
node> encodeURIComponent('Hello World')
'Hello%20World'
```

`null` でない文字列から `null` でない文字列への関数であり、副作用を持っていないので、この関数はその型 `String -> String` について適切な実行時表現を持っています。

次のような外部インポート宣言を使うと、この関数に型を割り当てることができます。

```
foreign import encodeURIComponent :: String -> String
```

また、PureScriptで記述された関数のように、この関数をPureScriptから使ってみます。たと

例えば、この宣言をモジュールとして保存して `psci` にロードすると、先ほどの計算を再現することができます。

```
> encodeURIComponent "Hello World"
"Hello%20World"
```

このアプローチは、簡単なJavaScriptの値には適していますが、もっと複雑な値に使うには限界があります。ほとんどの既存のJavaScriptコードは、基本的なPureScriptの型の実行時表現によって課せられた厳しい条件を満たしていないからです。このような場合のためには、適切な実行時表現に従うことを強制するようにJavaScriptコードをラップするという別の方法があります。

## 10.11 JavaScriptの値のラッピング

外部インポート宣言は、型注釈の直前に文字列リテラルを含めることで、JavaScriptコードのブロックと対にすることができます。そのJavaScriptコードは、コンパイル時に生成されたコードに直接挿入されます。

これはPureScriptの型を与えるためにJavaScriptコードの既存の部分をラップする場合に特に便利です。このようにしたくなる理由はいくつかあります。

- 任意のJavaScriptの副作用を追跡するために、`Eff` モナドを使うことができます。
- 関数の適切な実行時表現を与えるために、`null` や `undefined` のような特殊な場合を処理するために必要な場合があります。

外部インポート宣言を使用して、配列についての `head` 関数を作成したいとしましょう。JavaScriptでは次のような関数になるでしょう。

```
function head(arr) {
    return arr[0];
}
```

しかし、この関数には問題があります。型 `forall a. [a] -> a` を与えようとしても、空の配列に対してこの関数は `undefined` を返します。したがって、この特殊な場合を処理するために、ラッパー関数を使用する必要があります。

簡単な方法としては、空の配列の場合に例外を投げる方法があります。

```
foreign import head
    "function head(arr) {\
```

```
\ if (arr.length) {\
\   return arr[0];\
\ } else {\
\   throw new Error('Empty array!');\
\ }\
\}" :: forall a. [a] -> a
```

バックスラッシュを使用するとその行から次の1行まで継続することができ、JavaScriptの実装を複数行に分離できることに注意してください。

## 10.12 外部型の定義

失敗した場合に例外を投げるという方法は、あまり理想的とはいえません。PureScriptのコードでは、欠けた値のような副作用は型システムを使って扱うのが普通です。この手法としては `Maybe` 型構築子を使う方法もありますが、この節ではFFIを使用した別の解決策を扱います。

実行時には型 `a` のように表現されますが `undefined` の値も許容するような新しい型 `Undefined a` を定義したいとしましょう。

外部インポート宣言とFFIを使うと、外部型(foreign type)を定義することができます。構文は外部関数を定義するのと似ています。

```
foreign import data Undefined :: * -> *
```

この予約語 `data` は値ではなく定義している型を表していることに注意してください。型シグネチャの代わりに、新しい型の種を与えます。このとき、種 `Undefined` が `* -> *` であると宣言しています。つまり `Undefined` は型構築子です。

これで `head` の定義を簡素化することができます。

```
foreign import head
"function head(arr) {\
\   return arr[0];\
\}" :: forall a. [a] -> Undefined a
```

2点変更がある注意してください。 `head` 関数の本体ははるかに簡単で、もしその値が未定義であったとしても `arr[0]` を返し、型シグネチャはこの関数が未定義の値を返すことがあるという事実を反映するよう変更されています。

この関数はその型の適切な実行時表現を持っていますが、型 `Undefined a` の値を使用する方法がありませんので、まったく役に立ちません。しかし、FFIを使用して新しい関数を幾つか書くことによって、それを修正することができます！

次の関数は、値が定義されているかどうかを教えてくれる最も基本的な関数です。

```
foreign import isUndefined
  "function isUndefined(value) {\
    \ return value === undefined;\
  }" :: forall a. Undefined a -> Boolean
```

PureScriptから `isUndefined` と `head` を一緒に使用すると、便利な関数を定義することができます。

```
isEmpty :: forall a. [a] -> Boolean
isEmpty = isUndefined <<< head
```

ここで、定義されたこの外部関数はとても簡単であり、PureScriptの型検査器を使うことによる利益をなるべく多く得るということを意味します。一般に外部関数は可能な限り小さく保ち、アプリケーションの処理はPureScriptコードへ移動しておくことをおすすめします。

## 10.13 多変数関数

PureScriptのPreludeには、興味深い外部型がいくつかも含まれています。すでに扱ってきたように、PureScriptの関数型は単一の引数だけを取りますが、カーリー化を使うと複数の引数の関数をシミュレートすることができます。これには明らかな利点があります。関数を部分適用することができ、関数型の型クラスインスタンスを与えることができます。ただし、効率上のペナルティが生じます。パフォーマンス重視するコードでは、複数の引数を受け入れる本物のJavaScript関数を定義することが必要な場合があります。Preludeではそのような関数を安全に扱うことができるようにする外部型が定義されています。

たとえば、Preludeの `Data.Function` モジュールには次の外部型宣言があります。

```
foreign import data Fn2 :: * -> * -> * -> *
```

これは3つの型引数を取る型構築子 `Fn2` を定義します。 `Fn2 a b c` は、型 `a` と `b` の2つの引数、戻り値の型 `c` をもつJavaScript関数の型を表現しています。

Preludeでは0引数から10引数までの関数について同様の型構築子が定義されています。

次のように `mkFn2` 関数を使うと、2引数の関数を作成することができます。

```
import Data.Function

divides :: Fn2 Number Number Boolean
divides = mkFn2 $ \n m -> m % n == 0
```

そして、`runFn2` 関数を使うと、2引数の関数を適用することができます。

```
> runFn2 divides 2 10
true

> runFn2 divides 3 10
false
```

ここで重要なのは、引数がすべて適用されるなら、コンパイラは `mkFn2` 関数や `runFn2` 関数をインライン化するということです。そのため、生成されるコードはとてもコンパクトになります。

```
var divides = function (n, m) {
    return m % n === 0;
};
```

## 10.14 均質なレコード

外部型のさらなる例として、均質なレコード(homogeneous records)の型を定義してみましょう。これは、どんなラベルでも持つことができますが、どのプロパティも同じ型をもっているレコードです。

PureScriptではレコードの各プロパティは異なる型を持つことができます。これは多くの場合に便利ですが、JavaScriptコードにおけるいくつかの典型的なパターンに意味のある型を与えるのがうまくいかないことがあります。

均質なレコードの型はそのプロパティの(統一された)型によってパラメータ化されることになるので、これは次のような `* -> *` という種を持つことになります。

```
foreign import data HRec :: * -> *
```

外部値(foreign value)を使用すると、簡単に空の均質なレコードを定義することができます。

```
foreign import empty
  "var empty = {}" :: forall a. HRec a
```

`forall a. HRec a` という型は、この空の均質なレコードは任意の型 `a` について型 `a` のプロパティを持っているのを表していることに注意してください。 `empty` はどのプロパティも持っていないので、これが正しいのはまったくの自明です！

また、均質なレコードに新しいフィールドを挿入する関数を定義することができます。PureScriptの値は不変なので、JavaScriptコードで既存のレコードをコピーする必要があります。

```
foreign import insert
  "function insert(key, value, rec) {\
    \ var copy = {};\
    \ for (var k in rec) {\
    \   if (rec.hasOwnProperty(k)) {\
    \     copy[k] = rec[k];\
    \   }\
    \ }\
    \ copy[key] = value;\
    \ return copy;\
  }" :: forall a. Fn3 String a (HRec a) (HRec a)
```

`insert` 関数は3引数の関数を表現するために型コンストラクタ `Fn3` を使っています。JavaScriptで手作業でカーリー化関数を書くことはとても面倒なので、`Fn3` を使うと便利です。この関数はレコードを複製し、複製へ新しいキーを追加します。

均質なレコードを使うと、通常のPureScriptレコードではできないような、いろいろな面白い操作を行うことができます。例えば、均質なレコードの値に対して関数をマッピングすることができます。

```
foreign import mapHRec
  "function mapHRec(f, rec) {\
    \ var mapped = {};\
    \ for (var k in rec) {\
    \   if (rec.hasOwnProperty(k)) {\
    \     mapped[k] = f(rec[k]);\
    \   }\
    \ }\
    \ return mapped;\
  }" :: forall a b. Fn2 (a -> b) (HRec a) (HRec b)
```

つまり、`HRec` は `Functor` なのです！

```
instance functorHRec :: Functor HRec where
  (<$>) f rec = runFn2 mapHRec f rec
```

また、`HRec` を `Foldable` 型クラスのインスタンスにすることもでき、レコードの値に対して畳み込みをすることもできます。さらに興味深いことに、レコードのプロパティの値だけでなくラベルも受け取る累積関数について畳み込みを実行することができます！

```
foreign import foldHRec
  "function foldHRec(f, r, rec) {\
    \ var acc = r;\
    \ for (var k in rec) {\
    \   if (rec.hasOwnProperty(k)) {\
    \     acc = f(acc, k, rec[k]);\
    \   }\
    \ }\
    \ return acc;\
  \}" :: forall a r. Fn3 (Fn3 r String a r) r (HRec a) r
```

この章のソースコードには、次の演習に対する解決策の基礎として使用することができる `HRec` モジュールの関数が含まれています。

## 演習

1. (簡単) `psci` でいくつかの簡単なレコードを構築し、`runFn3` 関数を使用して `insert` 関数を試してみてください。
2. (やや難しい) 2つの均質なレコードの和集合を計算する関数 `union` を書いてください。2つのレコードがラベルを共有している場合、2つめのレコードが優先させなければいけません。
3. (やや難しい) 通常の(カーリー化された)関数を使用する `foldHRec` ためのラッパー関数を書いてください。その関数は次のような型を持っていなければなりません。

```
forall a r. (r -> String -> a -> r) -> r -> HRec a -> r
```

この関数を定義するのにFFIは使用しないでください。

4. (難しい) 均質なレコードのキーを検索する関数 `lookup` を書いてください。その関数は次のような型を持っていなければなりません。

```
forall a. String -> HRec a -> Maybe a
```

この関数の次の2種類の実装を書いてください。最初のバージョンは `foldHRec` 関数を使用する必要があります。2つめのバージョンは、外部関数として定義しなければなりません。ヒント: 次のような関数の定義を探してみると参考になるかもしれません。

```
lookupHelper :: forall a r. Fn4 r (a -> r) String (HRec a) r
```

第1引数及び第2引数は、それぞれ `Nothing` と `Just` 関数に対応しています。

5. (難しい) マッピング関数が追加の引数としてプロパティのラベルを受け取る `mapHRec` 関数書いてください。その関数を使用して `HRec` の `Show` インスタンスを簡素化してください。

## 10.15 副作用の表現

`Eff` モナドも `Prelude` の外部型として定義されています。その実行時表現はとても簡単です。型 `Eff eff a` の式は、任意の副作用を実行し型 `a` の適切な実行時表現で値を返す、引数なしのJavaScript関数へと評価されます。

`Eff` 型の構築子の定義は、`Control.Monad.Eff` モジュールで次のように与えられています。

```
foreign import data Eff :: # ! -> * -> *
```

`Eff` 型の構築子は作用の行と返り値の型によってパラメータ化されおり、それが種に反映されることを思い出してください。

簡単な例として、`purescript-random` パッケージで定義される `random` 関数を考えてみてください。その型は次のようなものでした。

```
random :: forall eff. Eff (random :: Random) Number
```

`random` 関数の定義は次のように与えられます。

```
foreign import random
  "function random() {\
```



```
\ return Math.random();\
\}" :: forall eff. Eff (random :: Random | eff) Number
```

`random` 関数は実行時には引数なしの関数として表現されていることに注目してください。これは乱数生成という副作用を実行しそれを返しますが、返り値は `Number` 型の実行時表現と一致します。それは `null` でないJavaScriptの数です。

もう少し興味深い例として、Preludeの `Debug.Trace` モジュールで定義された `trace` 関数を考えてみましょう。 `trace` 関数は次の型を持っています。

```
forall eff. String -> Eff (trace :: Trace | eff) Unit
```

この定義は次のようになっています。

```
foreign import trace
  "function trace(s) {\
  \ return function() {\
  \   console.log(s);\
  \   return {};\
  \ };\
  \}" :: forall eff. String -> Eff (trace :: Trace | eff) Unit
```

実行時の `trace` の表現は、引数なしの関数を返す、単一の引数のJavaScript関数です。内側の関数はコンソールにメッセージを書き込むという副作用を実行し、空のレコードを返します。 `Unit` は空のレコード型のnewtypeとしてPreludeで定義されているので、内側の関数の戻り値の型は `Unit` 型の実行時表現と一致していることに注意してください。

作用 `Random` と `Trace` も外部型として定義されています。その種は `!`、つまり作用であると定義されています。例えば次のようになります。

```
foreign import data Random :: !
```

詳しくはあとで見えていきますが、このように新たな作用を定義することが可能なのです。

`Eff eff a` 型の式は、通常のJavaScriptのメソッドのようにJavaScriptから呼び出すことができます。例えば、この `main` 関数は作用の集合 `eff` と何らかの型 `a` について `Eff eff a` という型でなければならないので、次のように実行することができます。

```
PS.Main.main();
```

または、CommonJSの環境では次のようにします。

```
require('Main').main();
```

`psc` コンパイラを使用するときは、コマンドライン上で `--main` コンパイラオプションを使用すると、この `main` の呼び出しを自動的に生成することができます。

## 10.16 新しい作用の定義

この章のソースコードでは、2つの新しい作用が定義されています。最も簡単なのは `Control.Monad.Eff.Alert` モジュールで定義された `Alert` 作用です。これはその計算がポップアップウィンドウを使用してユーザに警告しうることを示すために使われます。

この作用は最初に外部型宣言を使用して定義されています。

```
foreign import data Alert :: !
```

`Alert` は種 `!` が与えられており、`Alert` が型ではなく作用であることを示しています。

次に、`alert` アクションが定義されています。`alert` アクションはポップアップを表示し、作用の行に `Alert` 作用を追加します。

```
foreign import alert
  "function alert(msg) {\
    \ return function() {\
      \ window.alert(msg);\
      \ return {};\
    };\
  }" :: forall eff. String -> Eff (alert :: Alert | eff) Unit
```

このアクションは `Debug.Trace` モジュールの `trace` アクションととてもよく似ています。唯一の違いは、`trace` アクションが `console.log` メソッドを使用しているのに対し、`alert` アクションは `window.alert` メソッドを使用していることです。このように、`alert` は `window.alert` が定義されているウェブブラウザのような環境で使うことができます。

`trace` の場合のように、`alert` 関数は型 `Eff (alert :: Alert | eff) Unit` の計算を表現するために引数なしの関数を使っていることに注意してください。

この章で定義される2つめの作用は、`Control.Monad.Eff.Storage` モジュールで定義され

ている `Storage` 作用です。これは計算がWeb Storage APIを使用して値を読み書きする可能性があることを示すために使われます。

この作用も同じように定義されています。

```
foreign import data Storage :: !
```

`Control.Monad.Eff.Storage` モジュールには、ローカルストレージから値を取得する `getItem` と、ローカルストレージに値を挿入したり値を更新する `setItem` という、2つのアクションが定義されています。この二つの関数は、次のような型を持っています。

```
getItem :: forall eff. String -> Eff (storage :: Storage | eff) Foreign
setItem :: forall eff. String -> String -> Eff (storage :: Storage | eff) Un
```

興味のある読者は、このモジュールのソースコードでこれらのアクションがどのように定義されているか調べてみてください。

`setItem` はキーと値(両方とも文字列)を受け取り、指定されたキーでローカルストレージに値を格納する計算を返します。

`getItem` の型はもっと興味深いものです。`getItem` はキーを引数に取り、キーに関連付けられた値をローカルストレージから取得しようとします。`window.localStorage` の `getItem` メソッドは `null` を返すことがあるので、戻り値は `String` ではなく、`purescript-foreign` パッケージの `Data.Foreign` モジュールで定義されている `Foreign` になっています。

`Data.Foreign` は、型付けされていないデータ、もっと一般的に言えば実行時表現が不明なデータを扱う方法を提供しています。

## 演習

1. (やや難しい) JavaScriptの `Window` オブジェクトの `confirm` メソッドのラップを書き、`Control.Monad.Eff.Alert` モジュールにその関数を追加してください。
2. (やや難しい) `localStorage` オブジェクトの `removeItem` メソッドのラップを書き、`Control.Monad.Eff.Storage` モジュールに追加してください

## 10.17 型付けされていないデータの操作

この節では、型付けされていないデータを、その型の適切な実行時表現を持った型付けさ

れたデータに変換する、`Data.Foreign` ライブラリの使い方について見て行きます。

この章のコードは、第8章の住所録の上にフォームの一番下に保存ボタンを追加することで作っていきます。保存ボタンがクリックされると、フォームの状態をJSONに直列化し、ローカルストレージに格納します。ページが再読み込みされると、JSON文書がローカルストレージから取得され、構文解析されます。

**Main** モジュールではフォームデータの型を定義します。

```
newtype FormData = FormData
  { firstName  :: String
  , lastName   :: String
  , street     :: String
  , city       :: String
  , state      :: String
  , homePhone  :: String
  , cellPhone  :: String
  }
```

問題は、このJSONが正しい形式を持っているという保証がないことです。別の言い方をすれば、JSONが実行時にデータの正しい型を表しているかはわかりません。この問題は `purescript-foreign` ライブラリによって解決することができます。他にも次のような使いかたがあります。

- WebサービスからJSONレスポンス
- JavaScriptコードから関数に渡された値

それでは、`psci` で `purescript-foreign` ライブラリを試してみましょう。二つのモジュールをインポートして起動します。

```
> :i Data.Foreign
> :i Data.Foreign.Class
```

`Foreign` な値を取得するためには、JSON文書を解析するのがいいでしょう。`purescript-foreign` はで次の2つの関数が定義されています。

```
parseJSON :: String -> F Foreign
readJSON  :: forall a. (IsForeign a) => String -> F a
```

型構築子 `F` は、実際は `Data.Foreign` で定義されている型同義語です。

```
type F = Either ForeignError
```

`purescript-foreign` ライブラリの関数のほとんどは、`F` モナドの値を返します。これは、型付けされた値を構築するのに、`do` 記法や `Applicative` 関手コンビネータを使うことができることを意味しています。

この `IsForeign` 型クラスは、それらの型が型付けされていないデータから得られることを表しています。プリミティブ型や配列について定義された型クラスインスタンスは存在しますが、独自のインスタンスを定義することもできます。

それでは `psci` で `readJSON` を使用していくつかの簡単なJSON文書を解析してみましょう。

```
> readJSON "\"Testing\"" :: F String
Right "Testing"

> readJSON "true" :: F Boolean
Right true

> readJSON "[1, 2, 3]" :: F [Number]
Right [1, 2, 3]
```

`Either` モナドでは `Right` データ構築子は成功を示していることを思い出してください。しかし、その不正なJSONや誤った型はエラーを引き起こすことに注意してください。

```
> readJSON "[1, 2, true]" :: F [Number]

Left (Error at array index 2: Type mismatch: expected Number, found Boolean)
```

`purescript-foreign` ライブラリはJSON文書で型エラーが発生した位置を教えてください。

## 10.18 nullとundefined値の取り扱い

実世界のJSON文書には`null`や`undefined`値が含まれているので、それらも扱えるようにしなければなりません。

`purescript-foreign` では、この問題を解決する3種類の構築子、`Null`、`Undefined`、`NullOrUndefined` が定義されています。先に定義した `Undefined` 型の構築子と似た目的を持っていますが、省略可能な値を表すために `Maybe` 型の構築子を内部的に使っています。

それぞれの型の構築子について、ラップされた値から内側の値を取り出す関数、`runNull`、`runUndefined`、`runNullOrUndefined` が提供されています。`null` 値を許容するJSON文書を解析するには、`readJSON` アクションまで対応する適切な関数を持ち上げます。

```
> runNull <$> readJSON "42" :: F (Null Number)
Right (Just 42)

> runNull <$> readJSON "null" :: F (Null Number)
Right Nothing
```

それぞれの場合で、型注釈が `<$>` 演算子の右辺に適用されています。たとえば、`readJSON "42"` は型 `F (Null Number)` を持っています。`runNull` 関数は最終的な型 `F (Maybe Number)` 与えるために `F` まで持ち上げられます。

型 `NULL Number` は数またはnullいずれかの値を表しています。各要素が `null` をかもしれない数値の配列のように、より興味深いの値を解析したい場合はどうでしょうか。その場合には、次のように `readJSON` アクションまで関数 `map runNull` を持ち上げます。

```
> :i Data.Array

> map runNull <$> readJSON "[1, 2, null]" :: F [Null Number]
Right [Just 1, Just 2, Nothing]
```

一般的には、同じ型に異なる直列化戦略を提供するには、`newtypes`を使って既存の型をラップするのがいいでしょう。`null`、`Undefined`、`NullOrUndefined` それぞれの型は、`Maybe` 型構築子に包まれた`newtype`として定義されています。

## 10.19 住所録の項目の直列化

フォームデータは `JSON.stringify` メソッドを使用して直列化されますが、これは `Data.JSON` モジュールで定義されている次の関数でラップされています。

```
foreign import stringify
  "function stringify(x) {\
  \   return JSON.stringify(x);\
  \}" :: Foreign -> String
```

保存ボタンをクリックすると、型 `FormData` の値が(`Foreign` の値に変換されたあと

で) `stringify` 関数に渡され、JSON文書として直列化されます。`FormData` 型はレコードの newtype で、`JSON.stringify` が渡された型 `FormData` の値はJSONオブジェクトとして扱われて直列化されます。newtypeはその基礎となるデータと同じ実行時表現を持っているためです。

生成されたJSONドキュメントを解析できるようにするためには、オブジェクトのプロパティを読み取れるようにしなければなりません。`purescript-foreign` ライブラリはその機能を `(!)` 演算子と `readProp` アクションによって提供しています。

```
(!) :: (Index i) => Foreign -> i -> F Foreign
readProp :: forall a i. (IsForeign a, Index i) => i -> Foreign -> F a
```

型クラス `Index` は外部値のプロパティをインデックスするために使われる型を表しています。`Index` のインスタンスは `String` (オブジェクトプロパティにアクセスするため) と `Number` (配列要素にアクセスするため) に対して提供されています。

`readProp` アクションを使うと、`FormData` 型の `IsForeign` のインスタンスを定義することができます。次のように `IsForeign` 型クラスで定義されている `read` 関数を実装する必要があります。

```
class IsForeign a where
  read :: Foreign -> F a
```

`read` 関数を実装するには、`F` の `Monad` 構造を使って小さな部分から `FormData` 構造体を次のように作っていきます。

```
instance formDataIsForeign :: IsForeign FormData where
  read value = do
    firstName <- readProp "firstName" value
    lastName  <- readProp "lastName"  value
    street    <- readProp "street"     value
    city      <- readProp "city"       value
    state     <- readProp "state"      value
    homePhone <- readProp "homePhone" value
    cellPhone <- readProp "cellPhone" value
    return $ FormData
      { firstName : firstName
      , lastName  : lastName
      , street    : street
      , city      : city
      , state     : state
      , homePhone : homePhone
```



```
    , cellPhone : cellPhone
  }
```

`FormData` の構築子関数を `F` 型構築子まで持ち上げると、このコードを `F` の `Applicative` 構造を使って書くこともできます。これは演習として残しておきます。

この型クラスのインスタンスは、ローカルストレージから取得したJSON文書を解析するために `readJSON` で次のように使われています。

```
loadSavedData = do
  item <- getItem "person"

  let
    savedData :: F (Maybe FormData)
    savedData = do
      jsonOrNull <- read item
      traverse readJSON (runNull jsonOrNull)
```

`savedData` アクションは2つのステップにわけて `FormData` 構造を読み取ります。まず、`getItem` から得た `Foreign` 値を解析します。`jsonOrNull` の型はコンパイラによって `Null String` だと推論されます(読者への演習: この型はどのように推論されているのでしょうか?)。 `traverse` 関数は `readJSON` を `Maybe.String` 型の結果の(不足しているかもしれない)要素へと適用するのに使われます。`readJSON` について推論される型クラスのインスタンスはちょうどさっき書いたもので、型 `F (Maybe FormData)` の値で結果を返します。

`traverse` の引数には `read` が最初の行で得た結果 `jsonOrNull` を使っているので、`F` のモナド構造を使う必要があります。

結果の `FormData` には3つの可能性があります。

- もし外側の構築子が `Left` なら、JSON文字列の解析中にエラーがあったか、それが間違った型の値を表しています。この場合、アプリケーションは先ほど書いた `alert` アクションを使用してエラーを表示します。
- もし外側の構築子が `Right` で内側の構築子が `Nothing` なら、`getItem` が `Nothing` を返しており、キーがローカルストレージに存在していなかったことを意味しています。この場合、アプリケーションは静かに実行を継続します。
- 最後に、`Right (Just _)` に適合した値はJSON文書としてただしく構文解析されたことを示しています。この場合、アプリケーションは適切な値でフォームフィールドを更新します。

`grunt` を実行し、それからブラウザで <http://localhost:8000> を開いて、これらのコードを試してみてください。保存ボタンをクリックするとフォームフィールドの内容をローカルストレ



ージへ保存することができ、ページを再読込するとフィールドが再現されるはずです。

## 演習

1. (簡単) `readJSON` を使って、`[[1, 2, 3], [4, 5], [6]]` のようなJavaScriptの数の2次元配列を表現するJSON文書を解析してください。要素をnullにすることが許容されている場合はどうでしょうか。配列自体がnullにすることが許容されている場合はどうなりますか。
2. (やや難しい) Applicativeコンビネータ `<$>` と `<*>` を使って `formDataIsForeign` 型クラスを書きなおしてください。
3. (やや難しい) `savedData` の実装の型を検証し、計算のそれぞれの部分式の推論された型を書き出してみましょう。
4. (難しい) 次のnewtype型は、タグ付き共用体として直列化されなければならない `Either a b` の型の値を表しています。

```
newtype Tagged a b = Tagged (Either a b)
```

つまり、直列化されたJSON文書には `Left` 構築子と `Right` 構築子のどちらが値を構築するのに使われたかということを表すプロパティ `tag` を含まなければいけません。実際の値は、JSON文書の `value` のプロパティに格納される必要があります。

例えば、JSONデータ `{ tag: "Left", value: 0 }` は `Left 0` へと復元されなければいけません。

この `Tagged` 型構築子の `IsForeign` についての妥当なインスタンスを書いてください。

5. (難しい、拡張)次のデータ型は、葉で値を持つ二分木を表しています。

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

JSONドキュメントとしてこの型の適切な表現を選択してください。 `JSON.stringify` と中間のレコードのnewtypeを使って二分木をJSONへ直列化する関数を書き、関連する `IsForeign` のインスタンスを書いてください。

---

この章では、PureScriptから外部のJavaScriptコードを扱う方法、およびその逆の方法を学びました。また、FFIを使用して信頼できるコードを書く時に生じる問題について見てきました。

- データの実行時表現の重要性を見て、外部関数が正しい表現を持っていることを確かめました。
- 外部型、つまり `Foreign` データ型を使用することによって、null値のような特殊な場合やJavaScriptの他の型のデータに対処する方法を学びました。
- Preludeで定義されたいくつかの共通の外部型、既存のJavaScriptコードとどのように相互運用に使用するかを見てきました。特に、`Eff` モナドにおける副作用の表現を導入し、新たな副作用を追跡するために `Eff` モナドを使用する方法を説明しました。
- `IsForeign` 型クラスを使用して安全にJSONデータを復元する方法を説明しました。

その他の例については、Githubの `purescript` 組織および `purescript-contrib` 組織が、FFIを使用するライブラリの例を多数提供しています。残りの章では、型安全な方法で現実世界の問題を解決するために使うライブラリを幾つか見ていきます。

## 11 モナドの探求

---

### 11.1 この章の目標

---

この章の目標は、異なるモナドから提供された副作用を合成する方法を提供するモナド変換子(monad transformers)について学ぶことです。NodeJSのコンソール上で遊ぶことができる、テキストアドベンチャーゲームを題材として扱います。ゲームの様々な副作用(ログイン、状態、および設定)がすべてモナド変換子スタックによって提供されます。

### 11.2 プロジェクトの準備

---

このモジュールのプロジェクトでは以下のBower依存関係が新たに導入されます。

- `purescript-maps` - 不変のマップと集合のためのデータ型を提供します。
- `purescript-transformers` - 標準のモナド変換子の実装を提供します。
- `purescript-node-readline` - NodeJSが提供する `readline` インターフェイスへのFFIバインディングを提供します。
- `purescript-yargs` - `yargs` コマンドライン引数処理ライブラリにApplicativeなインターフェイスを提供します。

## 11.3 ゲームの遊びかた

プロジェクトを実行するには、`grunt` でソースコードをビルドしてから、NodeJSにコンパイルされたJavaScriptを渡します。

```
$ node dist/Main.js
```

デフォルトでは使い方が表示されます。

```
node ./dist/Main.js -p <player name>

Options:
  -p, --player  Player name [required]
  -d, --debug   Use debug mode

Missing required arguments: p
The player name is required
```

`-p` オプションを使ってプレイヤー名を提供してください。

```
node dist/Main.js -p Phil
>
```

プロンプトからは、`look`、`inventory`、`take`、`use`、`north`、`south`、`east`、`west` などのコマンドを入力することができます。`--debug` コマンドラインオプションが与えられたときには、ゲームの状態を出力するための `debug` コマンドも使えます。

ゲームは2次元の碁盤の目の上でプレイし、コマンド `north`、`south`、`east`、`west` を発行することによってプレイヤーが移動します。ゲームにはアイテムのコレクションがあり、プレイヤーの所持アイテム一覧を表したり、ゲーム盤上のその位置にあるアイテムの一覧を表すのに使われます。`take` コマンドを使うと、プレイヤーの位置にあるアイテムを拾い上げることができます。

参考までに、このゲームのひと通りの流れは次のようになります。

```
$ node dist/Main.js -p Phil

> look
You are at (0, 0)
You are in a dark forest. You see a path to the north.
```

```
You can see the Matches.
```

```
> take Matches
```

```
You now have the Matches
```

```
> north
```

```
> look
```

```
You are at (0, 1)
```

```
You are in a clearing.
```

```
You can see the Candle.
```

```
> take Candle
```

```
You now have the Candle
```

```
> inventory
```

```
You have the Candle.
```

```
You have the Matches.
```

```
> use Matches
```

```
You light the candle.
```

```
Congratulations, Phil!
```

```
You win!
```

このゲームはとても単純ですが、この章の目的は `purescript-transformers` パッケージを使用してこのようなゲームを素早く開発できるようにするライブラリを構築することです。

## 11.4 State モナド

`purescript-transformers` パッケージで提供されるモナドをいくつか見てみましょう。

最初の例は、純粋な変更可能状態を提供する `State` モナドです。すでに `Eff` モナド、すなわち `Ref` 作用と `ST` 作用によって提供された変更可能な状態という2つのアプローチについては見てきました。`State` は第3の選択肢を提供しますが、これは `Eff` モナドを使用して実装されているわけではありません。

`State` 型構築子は、状態の型 `s`、および戻り値の型 `a` という2種類の引数を取ります。「`State` モナド」というように説明はしていますが、実際には `Monad` 型クラスのインスタンスが用意されているのは `State` に対してではなく、任意の型 `s` についての `State s` 型構築子に対してです。

`Control.Monad.State` モジュールは以下のAPIを提供しています。

```
get    :: forall s.      State s s
put    :: forall s. s    -> State s Unit
```

```
modify :: forall s. (s -> s) -> State s Unit
```

これは `Ref` 作用や `ST` 作用が提供するAPIととてもよく似ています。しかし、これらのアクションに `RefVal` や `STRef` に渡しているような、可変領域への参照を引数に渡さないことに注意してください。`State` と `Eff` モナドが提供する解決策の違いは、`State` モナドは暗黙的な単一の状態だけを提供していることです。この状態は `State` モナドの型構築子によって隠された関数の引数として実装されており、参照は明示的には渡されないのです。

例を見てみましょう。`State` モナドの使いかたのひとつとしては、状態を数として、現在の状態に配列の値を加算していくようなものかもしれません。状態の型 `s` として `Number` を選択し、配列の走査に `traverse_` を使って、配列の要素それぞれについて `modify` を呼び出すと、これを実現することができます。

```
import Data.Foldable (traverse_)
import Control.Monad.State
import Control.Monad.State.Class

sumArray :: [Number] -> State Number Unit
sumArray = traverse_ $ \n -> modify (\sum -> sum + n)
```

`Control.Monad.State` モジュールは `State` モナドでの計算を実行するための次の3つの関数を提供します。

```
evalState :: forall s a. State s a -> s -> a
execState :: forall s a. State s a -> s -> s
runState  :: forall s a. State s a -> s -> Tuple a s
```

3つの関数はそれぞれ初期値の型 `s` と計算の型 `State s a` を引数にとります。`evalState` は返回值だけを返し、`execState` は最終的な状態だけを返し、`runState` は `Tuple a s` 型の値として表現された返回值と状態の両方を返します。

先ほどの `sumArray` 関数が与えられたとすると、`psci` で次のように `execState` を使うと複数の配列内の数字を合計することができます。

```
> execState (do
    sumArray [1, 2, 3]
    sumArray [4, 5]
    sumArray [6]
) 0
```

## 演習

1. (簡単) 上の例で、`execState` を `runState` や `evalState` で置き換えると結果はどうなるでしょうか。
2. (やや難しい) `State` モナドと `traverse_` 関数を使用して、次のような関数を書いてください。

```
testParens :: String -> Boolean
```

これは `String` が括弧の対応が正しく付けられているかどうかを調べる関数です。この関数は次のように動作しなくてはなりません。

```
> testParens ""
true

> testParens "(()(()))()"
true

> testParens ")"
false

> testParens "(()())"
false
```

ヒント: 入力の文字列を文字の配列に変換するのに、`Data.String` モジュールの `split` 関数を使うといいかもしれません。

## 11.5 Reader モナド

`purescript-transformers` パッケージでは `Reader` というモナドも提供されています。このモナドは大域的な設定を読み取る機能を提供します。`State` モナドがひとつの可変状態を読み書きする機能を提供するのに対し、`Reader` モナドはデータの読み取りの機能だけを提供します。

`Reader` 型構築子は、構成の型を表す型 `r`、および戻り値の型 `a` の2つの型引数を取ります。

`Contro.Monad.Reader` モジュールは以下のAPIを提供します。

```
ask    :: forall r. Reader r r
local  :: forall r a. (r -> r) -> Reader r a -> Reader r a
```

`ask` アクションは現在の設定を読み取るために使い、`local` アクションは局所的に設定を変更して計算を実行するために使います。

たとえば、権限で制御されたアプリケーションを開発しており、現在の利用者の権限オブジェクトを保持するのに `Reader` モナドを使いたいとしましょう。型 `r` を次のようなAPIを備えた型 `Permission` として選択します。

```
hasPermission :: String -> Permissions -> Boolean
addPermission :: String -> Permissions -> Permissions
```

利用者が特定の権限を持っているかどうかを確認したいときは、`ask` を使って現在の権限オブジェクトを取得すればいつでも調べることができます。たとえば、管理者だけが新しい利用者の作成を許可されているとしましょう。

```
createUser :: Reader Permissions (Maybe User)
createUser = do
  permissions <- ask
  if hasPermission "admin" permissions
  then Just <$> newUser
  else return Nothing
```

`local` アクションを使うと、計算の実行中に `Permissions` オブジェクトを局所的に変更し、ユーザーの権限を昇格させることもできます。

```
runAsAdmin :: forall a. Reader Permissions a -> Reader Permissions a
runAsAdmin = local (addPermission "admin")
```

こうすると、利用者が `admin` 権限を持っていなかった場合であっても、新しい利用者を作成する関数を書くことができます。

```
createUserAsAdmin :: Reader Permissions (Maybe User)
createUserAsAdmin = runAsAdmin createUser
```

`Reader` モナドの計算を実行するには、大域的な設定を与える `runReader` 関数を使います。

```
runReader :: forall r a. Reader r a -> r -> a
```

## 演習

以下の演習では、`Reader` モナドを使って、字下げのついた文書を出力するための小さなライブラリを作っていきます。「大域的な設定」は、現在の字下げの深さを示す数になります。

```
type Level = Number
```

```
type Doc = Reader Level String
```

1. (簡単) 現在の字下げの深さで文字列を出力する関数 `line` を書いてください。その関数は、以下の型を持っている必要があります。

```
line :: String -> Doc
```

ヒント: 現在の字下げの深さを読み取るためには `ask` 関数を使用します。

2. (やや難しい) `local` 関数を使用して、コードブロックの字下げの深さを大きくする次のような関数を書いてください。

```
indent :: Doc -> Doc
```

3. (やや難しい) `Data.Traversable` で定義された `sequence` 関数を使用して、文書のリストを改行で区切って連結する次のような関数を書いてください。

```
cat :: [Doc] -> Doc
```

4. (やや難しい) `runReader` 関数を使用して、文書を文字列として出力する次のような関数を書いてください。

```
render :: Doc -> String
```

これで、このライブラリを次のように使うと、簡単な文書を書くことができるはずです。



```
render $ cat
  [ line "Here is some indented text:"
  , indent $ cat
    [ line "I am indented"
    , line "So am I"
    , indent $ line "I am even more indented"
    ]
  ]
```

## 11.6 Writer モナド

**Writer** モナドは、計算の返り値に加えて、もうひとつの値を累積していく機能を提供します。

よくある使い方としては型 **String** もしくは **[String]** でログを累積していくというものなどがありますが、**Writer** モナドはこれよりもっと一般的なものです。これは累積するのに任意のモノイドの値を使うことができ、**Sum** モノイドを使って、合計を追跡し続けるのに使ったり、**Any** モノイドを使って途中の **Boolean** 値がすべて真であるかどうかを追跡するのに使うことができます。

**Writer** 型の構築子は、**Monoid** 型クラスのインスタンスである型 **w**、および返り値の型 **a** という2つの型引数を取ります。

**Writer** のAPIで重要なのは **tell** 関数です。

```
tell :: forall w a. (Monoid w) => w -> Writer w Unit
```

**tell** アクションは、与えられた値を現在の累積結果に加算します。

例として、**[String]** モノイドを使用して、既存の関数にログ機能を追加してみましょう。最大公約数関数の以前の実装を考えてみます。

```
gcd :: Number -> Number -> Number
gcd n 0 = n
gcd 0 m = m
gcd n m = if n > m
           then gcd (n - m) m
           else gcd n (m - n)
```

**Writer [String] Number** に返り値の型を変更することで、この関数にログ機能を追加することができます。

```
import Control.Monad.Writer
import Control.Monad.Writer.Class

gcdLog :: Number -> Number -> Writer [String] Number
```

各手順で二つの入力を記録するために、少し関数を変更する必要があります。

```
gcd n 0 = return n
gcd 0 m = return m
gcd n m = do
  tell ["gcd " ++ show n ++ " " ++ show m]
  if n > m
    then gcd (n - m) m
    else gcd n (m - n)
```

`Writer` モナドの計算を実行するには、`execWriter` 関数と `runWriter` 関数のいずれかを使います。

```
execWriter :: forall w a. Writer w a -> w
runWriter  :: forall w a. Writer w a -> Tuple a w
```

ちょうど `State` モナドの場合と同じように、`execWriter` が累積されたログだけを返すのに対して、`runWriter` は累積されたログと結果の両方を返します。

`psci` で修正された関数を試してみましょう。

```
> :i Data.Tuple
> :i Data.Monoid
> :i Control.Monad.Writer
> :i Control.Monad.Writer.Class

> runWriter (gcd 21 15)

Tuple 3 ["gcd 21 15", "gcd 6 15", "gcd 6 9", "gcd 6 3", "gcd 3 3"]
```

## 演習

1. (やや難しい) `Writer` モナドと `purescript-monoids` パッケージの `Sum` のモノイドを使うように、上の `sumArray` 関数を書き換えてください。
2. (やや難しい) コラッツ関数は、自然数 `n` が偶数なら `n / 2`、`n` が奇数なら `3 * n`

+ 1 であると定義されています。たとえば、10 で始まるコラッツ数列は次のようになります。

```
10, 5, 16, 8, 4, 2, 1, ...
```

コラッツ関数の有限回の適用を繰り返すと、コラッツ数列は必ず最終的に 1 になるということが予想できます。

数列が 1 に到達するまでに何回のコラッツ関数の適用が必要かを計算する再帰的な関数を書いてください。

コラッツ関数のそれぞれの適用のログを記録するために `Writer` モナドを使用するように関数を変更してください。

## 11.7 モナド変換子

上の3つのモナド、`State`、`Reader`、`Writer` は、いずれもいわゆるモナド変換子(monad transformers)の例となっています。対応するモナド変換子はそれぞれ

`StateT`、`ReaderT`、`WriterT` と呼ばれています。

モナド変換子とは何でしょうか。さて、これまで見てきたように、モナドはPureScriptで適切なハンドラ(`runState`、`runReader`、`runWriter` など)を使って解釈される、いろいろな種類の副作用でPureScriptコードを拡張します。使用する必要がある副作用がひとつだけなら、これで問題ありません。しかし、同時に複数の副作用を使用できると便利なのがよくあります。例えば、`Maybe` と `Reader` を一緒に使用すると、ある大域的な設定の文脈で省略可能な結果を表現することができます。もしくは、`Either` モナドの純粋なエラー追跡機能と、`State` モナドが提供する変更可能な状態が同時に欲しくなるかもしれません。この問題を解決するのがモナド変換子です。

拡張可能作用の手法を使うとネイティブな作用を混在させることができるので、`Eff` モナドはこの問題に対する部分的な解決策を提供していることをすでに見てきたことに注意してください。モナド変換子はまた異なった解決策を提供しますが、これらの手法にはそれぞれ利点と限界があります。

モナド変換子は型だけでなく別の型構築子によってもパラメータ化される型構築子です。モナド変換子はモナドをひとつ取り、独自のいろいろな副作用を追加した別のモナドへと変換します。

例を見てみましょう。`Control.Monad.State.Trans` で定義された `StateT` は `State` のモナド変換子版です。`psci` を使って `StateT` の種を見てみましょう。

```
> :i Control.Monad.State.Trans
> :k StateT
* -> (* -> *) -> * -> *
```

とても読みにくそうに思うかもしれませんが、使い方を理解するために、`StateT` にひとつ引数を与えてみましょう。

`State` の場合、最初の型引数は使いたい状態の型です。それでは型 `String` を与えてみましょう。

```
> :k StateT String
(* -> *) -> * -> *
```

次の引数は種 `* -> *` の型構築子です。これは `StateT` の機能を追加したい元のモナドを表します。例として、`Either String` モナドを選んでみます。

```
> :k StateT String (Either String)
* -> *
```

型構築子が残りました。最後の引数は戻り値の型を表しており、たとえばそれを `Number` にすることができます。

```
> :k StateT String (Either String) Number
*
```

最後に、種 `*` の何かが残りましたが、この型の値を探してみましょう。

構築したモナド `StateT String (Either String)` は、エラーで失敗する可能性があり、変更可能な状態を使える計算を表しています。

外側の `StateT String` モナドのアクション(`get`、`put`、`modify`)は直接使うことができますが、ラップされている内側のモナド(`Either String`)の作用を使うためには、これらの関数をモナド変換子まで「持ち上げ」なくてははいけません。`Control.MonadTrans` モジュールでは、モナド変換子であるような型構築子を捕捉する `MonadTrans` 型クラスを次のように定義しています。

```
class MonadTrans t where
  lift :: forall m a. (Monad m) => m a -> t m a
```

このクラスは、基礎となる任意のモナド `m` の計算をとり、それをラップされたモナド `t m` へと持ち上げる、`lift` というひとつの関数だけを持っています。今回の場合、型構築子 `t` は `StateT String` で、`m` は `Either String` モナドとなり、`lift` は型 `Either String a` の計算を、型 `State String (Either String) a` の計算へと持ち上げる方法を提供することになります。これは、型 `Either String a` の計算を使うときは、`lift` を使えばいつでも作用 `StateT String` と `Either String` を隣り合わせに使うことができることを意味します。

たとえば、次の計算は `StateT` モナド変換子で導入されている状態を読み込み、状態が空の文字列である場合はエラーを投げます。

```
import Data.String (drop, take)

split :: StateT String (Either String) String
split = do
  s <- get
  case s of
    "" -> lift $ Left "Empty string"
    _   -> do
      put (drop 1 s)
      return (take 1 s)
```

状態が空でなければ、この計算は `put` を使って状態を `drop 1 s` (最初の文字を取り除いた `s`)へと更新し、`take 1 s` (`s` の最初の文字)を返します。

それでは `psci` でこれを試してみましょう。

```
> runStateT split "test"
Right (Tuple "t" "est")

> runStateT split ""
Left "Empty string"
```

これは `StateT` を使わなくても実装できるので、さほど驚くようなことではありません。しかし、モナドとして扱っているので、`do` 記法や `Applicative` コンビネータを使って、小さな計算から大きな計算を構築していくことができます。例えば、2回 `split` を適用すると、文字列から最初の2文字を読むことができます。

```
> runStateT ((++) <$> split <*> split) "test"
Right (Tuple ("te") ("st"))
```

他にもアクションを幾つか用意すれば、`split` 関数を使って、基本的な構文解析ライブラリ

を構築することができます。これは実際に `purescript-parsing` ライブラリで採用されている手法です。これがモナド変換子の力なのです。必要な副作用を選択して、`do`記法と `Applicative`コンビネータで表現力を維持しながら、様々な問題のための特注のモナドを作成することができるのです。

## 11.8 ErrorTモナド変換子

`purescript-transformers` パッケージでは、`Either e` モナドに対応する変換子である `ErrorT e` モナド変換子も定義されています。これは次のAPIを提供します。

```
class Error a where
  noMsg :: a
  strMsg :: String -> a

  throwError :: forall m a. (Error e) =>
    e -> ErrorT e m a
  catchError :: forall m a. (Error e) =>
    ErrorT e m a ->
    (e -> ErrorT e m a) ->
    ErrorT e m a

  runErrorT :: forall e m a. ErrorT e m a -> m (Either e a)
```

ちょうど `Either e` モナドと同じように、`throwError` アクションは失敗を示すために使われます。

`catchError` アクションを使うと、`throwError` でエラーが投げられたあとでも処理を継続することができるようになります。

`runErrorT` ハンドラを使うと、型 `ErrorT e m a` の計算を実行することができます。

このAPIは `purescript-exceptions` パッケージの `Exception` 作用によって提供されているものと似ています。しかし、いくつかの重要な違いがあります。

- `ErrorT` モデルが代数的データ型を使っているのに対して、`Exception` は実際のJavaScriptの例外を使っています。
- `ErrorT` が `Error` 型クラスのどんな型のエラーでも扱うのに対して、`Exception` 作用はJavaScriptの `Error` 型というひとつ例外の型だけを扱います。つまり、`ErrorT` では新たなエラー型を自由に定義できます。

試しに `ErrorT` を使って `Writer` モナドを包んでみましょう。ここでもモナド変換子 `ErrorT e` のアクションは自由に使えますが、`Writer` モナドの計算は `lift` を使って持ちあげなけ

ればなりません。

```
import Control.Monad.Trans
import Control.Monad.Writer
import Control.Monad.Writer.Class
import Control.Monad.Error
import Control.Monad.Error.Class

writerAndErrorT :: ErrorT String (Writer [String]) String
writerAndErrorT = do
  tell ["Before the error"]
  throwError "Error!"
  tell ["After the error"]
  return "Return value"
```

`psci` でこの関数を試すと、ログの蓄積とエラーの送出という2つの作用がどのように相互作用しているのかを見ることができます。まず、`runErrorT` を使って外側の `ErrorT` 計算を実行し、型 `Write String (Either String String)` の結果を残します。それから、`runWriter` で内側の `Writer` 計算を実行します。

```
> runWriter $ runErrorT writerAndErrorT
Tuple (Left "Error!") ["Before the error"]
```

実際に追加されるログは、エラーが投げられる前に書かれたログメッセージだけであることにも注目してください。

## 11.9 モナド変換子スタック

これまで見てきたように、モナド変換子を使うと既存のモナドの上に新しいモナドを構築することができます。任意のモナド変換子 `t1` と任意のモナド `m` について、その適用 `t1 m` もまたモナドになります。これはふたつめのモナド変換子 `t2` を先ほどの結果 `t1 m` に適用すると、第3のモナド `t2 (t1 m)` を作れることを意味しています。このように、構成するモナドによって提供された副作用を組み合わせる、モナド変換子のスタックを構築することができます。

実際には、基本となるモナド `m` は、ネイティブの副作用が必要なら `Eff` モナド、さもなければ `Control.Monad.Identity` モジュールで定義されている `Identity` モナドになります。`Identity` モナドは何の新しい副作用も追加しませんから、`Identity` モナドの変換は、モナド変換子の作用だけを提供します。実際に、`State` モナド、`Reader` モナド、`Writer` モナドは、`Identity` モナドをそれぞれ `StateT`、`ReaderT`、`WriterT` で変換

することによって実装されています。

それでは3つの副作用が組み合わされている例を見てみましょう。`Identity` モナドをスタックの底にして、`StateT` 作用、`WriterT` 作用、`ErrorT` 作用を使います。このモナド変換子スタックは、ログの蓄積し、純粋なエラー、可変状態の副作用を提供します。

このモナド変圧器スタックを使うと、ロギングの機能が追加された `split` アクションを作ることができます。

```
type Parser = StateT String (WriterT [String] (ErrorT String Identity))

split :: Parser String
split = do
  s <- get
  lift $ tell ["The state is " ++ show s]
  case s of
    "" -> lift $ lift $ throwError "Empty string"
    _   -> do
      put (drop 1 s)
      return (take 1 s)
```

この計算を `psci` で試してみると、`split` が実行されるたびに状態がログに追加されることがわかります。

モナド変換子スタックに現れる順序に従って、副作用を取り除いていかなければならないことに注意してください。最初に `StateT` 型構築子を取り除くために `runStateT` を使い、それから `runWriterT` を使い、その後 `runErrorT` を使います。最後に `runIdentity` を使用して `Identity` モナドの演算を実行します。

```
> let runParser p s = runIdentity $ runErrorT $ runWriterT $ runStateT p s

> runParser split "test"

Right (Tuple (Tuple "t" "est") ["The state is test"])

> runParser ((++) <$> split <*> split) "test"

Right (Tuple (Tuple "te" "st") ["The state is test", "The state is est"])
```

しかしながら解析が失敗した場合は、状態が空であるためログはまったく出力されません。

```
runParser split ""
> Left "Empty string"
```



これは、`ErrorT` モナド変換子が提供する副作用が、`WriterT` モナド変換子が提供する副作用に影響を受けるためです。これはモナド変換子スタックが構成されている順序を変更することで解決することができます。スタックの最上部に `ErrorT` 変換子を移動すると、先ほど `Writer` を `ErrorT` に変換したときと同じように、最初のエラーまでに書かれたすべてのメッセージが含まれるようになります。

このコードの問題のひとつは、複数のモナド変換子の上まで計算を持ち上げるために、`lift` 関数を複数回使わなければならないということです。たとえば、`throwError` の呼び出しは、1回めは `WriterT` へ、2回めは `StateT` へと、2回持ちあげなければなりません。小さなモナド変換子スタックならなんとかありますが、そのうち不便だと感じるようになるでしょう。

幸いなことに、これから見るような型クラス推論によって提供されるコードの自動生成を使うと、ほとんどの「多段持ち上げ」を行うことができます。

## 演習

1. (簡単) `Identity` 関手の上の `ErrorT` モナド変換子を使って、分母がゼロの場合はエラーを投げる、2つの数の商を求める関数 `safeDivide` を書いてください。
2. (やや難しい) 現在の状態が接頭辞に適合するか、エラーメッセージとともに失敗する、次のような構文解析関数を書いてください。

```
string :: String -> Parser String
```

この構文解析器は次のように動作しなくてはなりません。

```
> runParser (string "abc") "abcdef"

Right (Tuple (Tuple "abc" "def") ["The state is abcdef"])
```

ヒント: 出発点として `split` の実装を使うといいでしょう。

3. (難しい) 以前 `Reader` モナドを使用して書いた文書出力ライブラリを、`ReaderT` と `WriterT` モナド変圧器を使用して再実装してください。

文字列を出力する `line` や文字列を連結する `cat` を使うのではなく、`WriterT` モナド変換子と一緒に `[String]` モノイドを使い、結果へ行を追加するのに `tell` を

使ってください。

## 11.10 救済のための型クラス

章の最初で扱った `State` モナドを見てみると、`State` モナドのアクションには次のような型が与えられていました。

```
get    :: forall s.          State s s
put    :: forall s. s        -> State s Unit
modify :: forall s. (s -> s) -> State s Unit
```

`Control.Monad.State.Class` モジュールで与えられている型は、実際には次のようにもっと一般的です。

```
get    :: forall m s. (MonadState s m) =>          m s
put    :: forall m s. (MonadState s m) => s        -> m Unit
modify :: forall m s. (MonadState s m) => (s -> s) -> m Unit
```

`Control.Monad.State.Class` モジュールには「純粋な変更可能な状態を提供するモナド」への抽象化を可能にする `MonadState` (多変数)型クラスが定義されています。予想できると思いますが、`State s` 型構築子は `MonadState s` 型クラスのインスタンスになっており、このクラスには他にも興味深いインスタンスが数多くあります。

特に、`purescript-transformers` パッケージではモナド変換子 `WriterT`、`ReaderT`、`ErrorT` についての `MonadState` のインスタンスが提供されています。実際に、`StateT` がモナド変換子スタックのどこかに現れ、`StateT` より上のすべてが `MonadState` のインスタンスであれば、`get`、`put`、`modify` を直接自由に使用することができます。

実は、これまで扱ってきた `ReaderT`、`WriterT`、`ErrorT` 変換子についても、同じことが成り立っています。`purescript-transformers` では、それらの操作をサポートするモナドの上に抽象化することを可能にする、主な変換子それぞれについての型クラスが定義されています。

上の `split` 関数の場合、構築されたこのモナドスタックは型クラス `MonadState`、`MonadWriter`、`MonadError` それぞれのインスタンスです。これはつまり、`lift` をまったく呼び出す必要がないことを意味します！まるでモナドスタック自体に定義されていたかのように、アクション `get`、`put`、`tell`、`throwError` をそのまま使用することができます。

```

split :: Parser String
split = do
  s <- get
  tell ["The state is " ++ show s]
  case s of
    "" -> throwError "Empty string"
    _  -> do
      put (drop 1 s)
      return (take 1 s)

```

この計算はまるで、可変状態、ロギング、エラー処理という3つの副作用に対応した、独自のプログラミング言語を拡張したかのようにみえます。しかしながら、内部的にはすべてはあくまで純粋な関数と普通のデータを使って実装されているのです。

## 11.11 Alternative 型クラス

`purescript-control` パッケージでは失敗しうる計算を操作するための抽象化がいくつか定義されています。そのひとつは `Alternative` 型クラスです。

```

class (Functor f) <= Alt f where
  (<|>) :: forall a. f a -> f a -> f a

class (Alt f) <= Plus f where
  empty :: forall a. f a

class (Applicative f, Plus f) <= Alternative f where

```

`Alternative` は、失敗しうる計算のプロトタイプを提供する `empty` 値、エラーが起きたときに代替(Alternative)計算へ戻ってやり直す機能を提供する `<|>` 演算子 という、2つの新しいコンビネータを提供しています。

`Control.Alternative` モジュールでは `Alternative` 型クラスで型構築子を操作する2つの便利な関数を提供します。

```

many :: forall f a. (Alternative f, Lazy1 f) => f a -> f [a]
some :: forall f a. (Alternative f, Lazy1 f) => f a -> f [a]

```

`many` コンビネータは計算をゼロ回以上繰り返し実行するために `Alternative` 型クラスを使用しています。`some` コンビネータも似ていますが、成功するために少なくとも1回の計算を必要とします。

今回の `Parser` モナド変換子スタックの場合は、`ErrorT` コンポーネントから導かれた、明らかな方法で失敗をサポートする、`Alternative` のインスタンスが存在します。これは、構文解析器を複数回実行するために `many` 関数と `some` 関数を使うことができることを意味します。

```
> :i Split
> :i Control.Alternative

> runParser (many split) "test"

Right (Tuple (Tuple ["t", "e", "s", "t"] "")
  [ "The state is \"test\""
  , "The state is \"est\""
  , "The state is \"st\""
  , "The state is \"t\""
  ])
```

ここで、入力文字列 `"test"` は、1文字の文字列4つの配列を返すように、繰り返し分割されています。残った状態は空文字列で、ログは `split` コンビネータが4回適用されたことを示しています。

`Alternative` 型構築子の他の例としては、`Maybe` や、`[]` つまり配列の型構築子があります。

## 11.12 モナド内包表記

`Control.MonadPlus` モジュールには `MonadPlus` と呼ばれる `Alternative` 型クラスの若干の変形が定義されています。`MonadPlus` はモナドと `Alternative` のインスタンスの両方である型構築子を補足します。

```
class (Monad m, Alternative m) <= MonadPlus m
```

実際、`Parser` モナドは `MonadPlus` のインスタンスです。

以前に本書中で配列の内包表記を扱ったとき、不要な結果をフィルタリングするため使われる `guard` 関数を導入しました。実際は `guard` 関数はもっと一般的で、`MonadPlus` のインスタンスであるすべてのモナドに対して使うことができます。

```
guard :: forall m. (MonadPlus m) => Boolean -> m Unit
```

<|> 演算子は失敗時のバックトラッキングをできるようにします。これがどのように役立つかを見るために、大文字だけに適合する `split` コンビネータの亜種を定義してみましょう。

```
upper :: Parser String
upper = do
  s <- split
  guard $ toUpper s == s
  return s
```

ここで、文字列が大文字でない場合に失敗するよう `guard` を使用しています。このコードは前に見た配列内包表記とよく似ていることに注目してください。このように `MonadPlus` が使われておりモナド内包表記(monad comprehensions)を構築するために参照することがあります。

## 11.13 バックトラッキング

<|> 演算子を使うと、失敗したときに別の代替計算へとバックトラックすることができます。これを確かめるために、小文字に一致するもう一つの構文解析器を定義してみましょう。

```
lower :: Parser String
lower = do
  s <- split
  guard $ toLower s == s
  return s
```

これにより、まずもし最初の文字が大文字なら複数の大文字に適合し、さもなくばもし最初の文字が小文字なら複数の小文字に適合する、という構文解析器を定義することができます。

```
> let upperOrLower = some upper <|> some lower
```

この構文解析器は、大文字と小文字が切り替わるまで、文字に適合し続けます。

```
> runParser upperOrLower "abcDEF"

Right (Tuple (Tuple ["a","b","c"] ("DEF"))
  [ "The state is \"abcDEF\"",
    , "The state is \"bcDEF\"",
    , "The state is \"cDEF\"",
  ])
```

`many` を使うと、文字列を小文字と大文字の要素に完全に分割することもできます。

```
> let components = many upperOrLower

> runParser components "abCDeFgh"

Right (Tuple (Tuple [["a","b"],["C","D"],["e"],["F"],["g","h"]], ""))
  [ "The state is \"abCDeFgh\""
  , "The state is \"bCDeFgh\""
  , "The state is \"CDeFgh\""
  , "The state is \"DeFgh\""
  , "The state is \"eFgh\""
  , "The state is \"Fgh\""
  , "The state is \"gh\""
  , "The state is \"h\""
  ])
```

繰り返しになりますが、これはモナド変換子がもたらす再利用性の威力を示しています。標準的な抽象化を再利用することで、バックトラック構文解析器を宣言型のスタイルでわずか数行のコードで書くことができました！

## 演習

1. (簡単) `string` 構文解析器の実装から `lift` 関数の呼び出しを取り除いてください。新しい実装の型が整合していることを確認し、そうでなることをよく納得しておきましょう。
2. (やや難しい) `string` 構文解析器と `many` コンビネータを使って、文字列 `"a"` の連続と、それに続く文字列 `"b"` の連続からなる文字列を認識する構文解析器を書いてください。
3. (やや難しい) `<|>` 演算子を使って、文字 `a` と文字 `b` が任意の順序で現れるような文字列を認識する構文解析器を書いてください。
4. (難しい) `Parser` モナドは次のように定義されるかもしれません。

```
type Parser = ErrorT String (StateT String (WriterT [String] Identity))
```

このように変更すると、構文解析関数にどのような影響を与えるでしょうか。

## 11.14 RWSモナド

モナド変換子のある特定の組み合わせは、`purescript-transformers` パッケージ内の単一のモナド変換子として提供されるのが一般的です。`Reader`、`Writer`、`State` のモナドは、`Reader-Writer-State`モナド(`RWS` モナド)へと結合されます。このモナドは `RWST` モナド変換子と呼ばれる、対応するモナド変換子を持っています。

ここでは `RWS` モナドを使ってテキストアドベンチャーゲームの処理を設計していきます。

`RWS` モナドは(戻り値の型に加えて)3つの型変数で定義されています。

```
type RWS r w s = RWST r w s Identity
```

副作用を提供しない `Identity` にベースモナドを設定することで、`RWS` モナドが独自のモナド変換子の観点から定義されていることに注意してください。

第1型引数 `r` は大域的な設定の型を表します。第2型引数 `w` はログを蓄積するために使用するモノイド、第3型引数 `s` は可変状態の型を表しています。

このゲームの場合には、大域的な設定は `Data.GameEnvironment` モジュールの `GameEnvironment` と呼ばれる型で定義されています。

```
type PlayerName = String

newtype GameEnvironment = GameEnvironment
  { playerName    :: PlayerName
  , debugMode     :: Boolean
  }
```

`GameEnvironment` では、プレイヤー名と、ゲームがデバッグモードで動作しているか否かを示すフラグが定義されています。これらのオプションは、モナド変換子を実行するときにコマンドラインから設定されます。

可変状態は `Data.GameState` モジュールの `GameState` と呼ばれる型で定義されています。

```
import qualified Data.Map as M
import qualified Data.Set as S

newtype GameState = GameState
  { items      :: M.Map Coords (S.Set GameItem)
  , player     :: Coords
  , inventory  :: S.Set GameItem
  }
```

```
}
```

`Coords` データ型は2次元平面の点を表し、`GameItem` データ型はゲーム内のアイテムです。

```
data GameItem = Candle | Matches
```

`GameState` 型はソートされたマップを表す `Map` とソートされた集合を表す `Set` という2つの新しいデータ構造を使っています。`items` プロパティは、そのゲーム平面上の座標と、ゲームアイテムの集合へのマッピングになっています。`player` プロパティはプレイヤーの現在の座標を格納しており、`inventory` プロパティは現在プレイヤーが保有するゲームアイテムの集合です。

`Map` と `Set` のデータ構造は平衡2-3木を使って実装されており、`Ord` 型クラス内の任意の型をキーとして使用することができます。これは今回のデータ構造のキーが完全に順序付けできることを意味します。

ゲームのアクションを書くために、`Map` と `Set` 構造がどのように使っていくのかを見ていきましょう。

ログとしては `[String]` モノイドを使います。`RWS` を使って `Game` モナドのための型同義語を定義しておきます。

```
type Log = [String]
```

```
type Game = RWS GameEnvironment Log GameState
```

## 11.15 ゲームロジックの実装

今回は、`Reader` モナド、`Writer` モナド、`State` モナドのアクションを再利用し、`Game` モナドで定義されている単純なアクションを組み合わせるゲームを構築していきます。このアプリケーションの最上位では、`Game` モナドで純粋な計算を実行しており、`Eff` モナドはコンソールにテキストを出力するような追跡可能な副作用へと結果を変換するために使っています。

このゲームで最も簡単なアクションのひとつは `has` アクションです。このアクションはプレイヤーの持ち物に特定のゲームアイテムが含まれているかどうかを調べます。これは次のように定義されます。



```
has :: GameItem -> Game Boolean
has item = do
    GameState state <- get
    return $ item `S.member` state.inventory
```

この関数は、現在のゲームの状態を読み取るために `Monad.State` 型クラスで定義されている `get` アクションを使っており、指定した `GameItem` が持ち物の `Set` のなかに出現するかどうかを調べるために `Data.Set` で定義されている `member` 関数を使っています。

他にも `pickUp` アクションがあります。現在の位置にゲームアイテムがある場合、プレイヤーの持ち物にそのアイテムを追加します。これには `MonadWriter` と `MonadState` 型クラスのアクションを使っています。まず、現在のゲームの状態を読み取ります。

```
pickUp :: GameItem -> Game Unit
pickUp item = do
    GameState state <- get
```

次に `pickUp` は現在の位置にあるアイテムの集合を検索します。これは `Data.Map` で定義された `lookup` 関数を使って行います。

```
case state.player `M.lookup` state.items of
```

`lookup` 関数は `Maybe` 型構築子で示されたオプションな結果を返します。`lookup` 関数は、キーがマップにない場合は `Nothing` を返し、それ以外の場合は `Just` 構築子で対応する値を返します。

関心があるのは、指定されたゲームアイテムが対応するアイテムの集合に含まれている場合です。`member` 関数を使うとこれを調べることができます。

```
Just items | item `S.member` items -> do
```

この場合、`put` を使ってゲームの状態を更新し、`tell` を使ってログにメッセージを追加します。

```
let newItems = M.update (Just <<< S.delete item) state.player state.items
    newInventory = S.insert item state.inventory
    put $ GameState state { items      = newItems
                           , inventory = newInventory
                           }
    tell ["You now have the " ++ show item]
```

ここで、`MonadState` と `MonadWriter` の両方について `Game` モナド変換子スタックについての適切なインスタンスが存在するので、2つの計算はどちらも `lift` は必要ないことに注意してください。

`put` の引数では、レコード更新を使ってゲームの状態の `items` と `inventory` フィールドを変更しています。特定のキーの値を変更するには `Data.Map` の `update` 関数を使います。このとき、`delete` 関数を使い指定したアイテムを集合から取り除くことで、プレイヤーの現在の位置にあるアイテムの集合を変更します。

最後に、`pickUp` 関数は `tell` を使ってユーザに次のように通知することにより、残りの場合を処理します。

```
_ -> tell ["I don't see that item here."]
```

`Reader` モナドを使う例として、`debug` コマンドのコードを見てみましょう。ゲームがデバッグモードで実行されている場合、このコマンドを使うとユーザは実行時にゲームの状態を調べることができます。

```
GameEnvironment env <- ask
if env.debugMode
  then do
    state <- get
    tell [show state]
  else tell ["Not running in debug mode."]
```

ここでは、ゲームの設定を読み込むために `ask` アクションを使用しています。繰り返しますが、どんな計算の `lift` も必要なく、同じ `do` 記法ブロック内で `MonadState`、`MonadReader`、`MonadWriter` 型クラスで定義されているアクションを使うことができますことに注意してください。

`debugMode` フラグが設定されている場合、`tell` アクションを使ってログに状態が追加されます。そうでなければ、エラーメッセージが追加されます。

`Game.purs` モジュールでは、`MonadState` 型クラス、`MonadReader` 型クラス、`MonadWriter` 型クラスでそれぞれ定義されたアクションだけを使って、同様のアクションが定義されています。

## 11.16 計算の実行

このゲームロジックは `RWS` モナドで動くため、ユーザのコマンドに応答するためには計算を実行する必要があります。

このゲームのフロントエンドは、`yargs` コマンドライン構文解析ライブラリへのApplicativeなインターフェイスを提供する `purescript-yargs` パッケージと、対話的なコンソールベースのアプリケーションを書くことを可能にするNodeJSの `readline` モジュールをラップする `purescript-node-readline` パッケージという2つのパッケージで構成されています。

このゲームロジックへのインタフェースは `Game` モジュール内の関数 `game` によって提供されます。

```
game :: [String] -> Game Unit
```

この計算を実行するには、ユーザが入力した単語のリストを文字列の配列として渡してから、`runRWS` を使って `RWS` の計算を実行します。

```
type See s a w = { log :: w, result :: a, state :: s }

runRWS :: forall r w s a. RWS r w s a -> r -> s -> See s a w
```

`runRWS` は `runReader`、`runWriter`、`runState` を組み合わせたように見えます。これは、引数として大域的な設定および初期状態をとり、ログ、結果、最的な終状態を含むレコードを返します。

このアプリケーションのフロントエンドは、次の型シグネチャを持つ関数 `runGame` によって定義されます。

```
runGame :: GameEnvironment -> Eff (console :: Console, trace :: Trace) Unit
```

`Console` 作用は、この関数が `purescript-node-readline` パッケージを使ってコンソールを介してユーザと対話することを示しています。`runGame` は関数の引数としてのゲームの設定とります。

`purescript-node-readline` パッケージでは、端末からのユーザ入力を扱う `Eff` モナドのアクションを表す `LineHandler` 型が提供されています。対応するAPIは次のとおりです。

```
type LineHandler eff = String -> Eff eff Unit

setLineHandler :: forall eff. LineHandler eff ->
  Interface ->
```

```
Eff (console :: Console | eff) Interface
```

`Interface` 型はコンソールのハンドルを表しており、コンソールと対話する関数への引数として渡されます。`createInterface` 関数を使用すると `Interface` を作成することができます。

```
runGame env = do
    interface <- createInterface process.stdin process.stdout noCompletion
```

最初の手順はコンソールにプロンプトを設定することです。`interface` ハンドルを渡し、プロンプト文字列とインデントレベルを提供します。

```
setPrompt "> " 2 interface
```

今回の場合、ラインハンドラ関数を実装することに関心があります。ラインハンドラは `let` 宣言内の補助関数を使って次のように定義されています。

```
lineHandler :: GameState -> String -> Eff (console :: Console, trace :: Trace) Unit
lineHandler currentState input = do
    let result = runRWS (game (split " " input)) env currentState
    foreachE result.log trace
    setLineHandler (lineHandler result.state) interface
    prompt interface
    return unit
```

`lineHandler` では `env` という名前のゲーム構成や、`interface` という名前のコンソールハンドルを参照しています。

このハンドラは追加の最初の引数としてゲームの状態を取ります。ゲームのロジックを実行するために `runRWS` にゲームの状態を渡さなければならないので、これは必要となっています。

このアクションが最初に行うことは、`Data.String` モジュールの `split` 関数を使用して、ユーザーの入力を単語に分割することです。それから、ゲーム環境と現在のゲームの状態を渡し、`runRWS` を使用して(`RWS` モナドで) `game` アクションを実行しています。

純粋な計算であるゲームロジックを実行し、画面にすべてのログメッセージを出力して、ユーザに次のコマンドのプロンプトを表示する必要があります。`foreachE` アクションは (`[String]` 型の)ログを走査し、コンソールにその内容を出力するために使われています。そして `setLineHandler` を使ってラインハンドラ関数を更新することで、ゲームの状態を更

新します。最後に `prompt` アクションを使ってプロンプトが再び表示しています。

`runGame` 関数ではコンソールインターフェイスに最初のラインハンドラを設定して、最初のプロンプトを表示します。

```
setLineHandler (lineHandler initialState) interface
  prompt interface
```

## 演習

1. (やや難しい) ゲームフィールド上にあるすべてのゲームアイテムをユーザの持ちものに移動する新しいコマンド `cheat` を実装してください。
2. (難しい) 今のところ `WriteT` モナド変換子は、エラーメッセージと情報メッセージの2つの種類のメッセージのために使われています。このため、コードのいくつかの箇所では、エラーの場合を扱うためにcase式を使用しています。

エラーメッセージを扱うのに `ErrorT` モナド変換子を使い、情報メッセージを扱うのに `WriteT` を使うように、コードをリファクタリングしてください。

## 11.17 コマンドラインオプションの扱い

このアプリケーションの最後の部品は、コマンドラインオプションの解析と `GameEnvironment` レコードを作成する役目にあります。このためには `purescript-yargs` パッケージを使用します。

`purescript-yargs` はApplicativeなコマンドラインオプション構文解析器の例です。Applicative関手を使うと、いろいろな副作用の型を表す型構築子まで任意個数の引数の関数をを持ち上げられることを思い出してください。`purescript-yargs` パッケージの場合には、コマンドラインオプションからの読み取りの副作用を追加する `Y` 関手が興味深い関手になっています。これは次のようなハンドラを提供しています。

```
runY :: forall a eff. YargsSetup ->
      Y (Eff eff a) ->
      Eff (console :: Console, err :: Exception | eff) a
```

この関数の使いかたは、例で示すのが最も適しているでしょう。このアプリケーションの `main` 関数は `runY` を使って次のように定義されています。

```
main = runY (usage "$0 -p <player name>") $ runGame <$> env
```

最初の引数は `yargs` ライブラリを設定するために使用されます。今回の場合、使用方法のメッセージだけを提供していますが、`Node.Yargs.Setup` モジュールには他にもいくつかのオプションを提供しています。

2番目の引数では、`Y` 型構築子まで `runGame` 関数を持ち上げるために `<$>` コンビネータを使用しています。引数 `env` は `where` 節で `Applicative` 演算子 `<$>`、`<*>` を使って構築されています。

```
where
  env :: Y GameEnvironment
  env = gameEnvironment
        <$> yarg "p" ["player"]
            (Just "Player name")
            (Right "The player name is required")
            false
        <*> flag "d" ["debug"]
            (Just "Use debug mode")
```

`PlayerName -> Boolean -> GameEnvironment` という型を持つこの `gameEnvironment` 関数は、`Y` まで持ち上げられています。このふたつの引数は、コマンドラインオプションからプレイヤー名とデバッグフラグを読み取る方法を指定しています。最初の引数は `-p` もしくは `--player` オプションで指定されるプレイヤー名オプションについて記述しており、2つ目の引数は `-d` もしくは `--debug` オプションで指定されるデバッグモードフラグについて記述しています。

これは `Node.Yargs.Applicative` モジュールで定義されているふたつの基本的な関数について示しています。`yarg` は(型 `String`、`Number`、`Boolean` の)オプションな引数を取りコマンドラインオプションを定義し、`flag` は型 `Boolean` のコマンドラインフラグを定義しています。

`Applicative` 演算子によるこの記法を使うことで、コマンドラインインターフェイスに対してコンパクトで宣言的な仕様を与えることが可能になったことに注意してください。また、`env` の定義で `runGame` 関数に新しい引数を追加し、`<*>` を使って追加の引数まで `runGame` を持ち上げるだけで、簡単に新しいコマンドライン引数を追加することができます。

## 演習

1. (やや難しい) `GameEnvironment` レコードに新しい真偽値のプロパティ

イ `cheatMode` を追加してください。また、`yargs` 設定に、チートモードを有効にする新しいコマンドラインフラグ `-c` を追加してください。チートモードが有効になっていない場合、`cheat` コマンドは禁止されなければなりません。

## 11.18 まとめ

---

モナド変換子を使用したゲームの純粋な定義、コンソールを使用したフロントエンドを構築するための `Eff` モナドなど、この章ではこれまで学んできた手法を実用的に使いました。

ユーザインターフェースからの実装を分離したので、ゲームの別のフロントエンドを作成することも可能でしょう。例えば、`Eff` モナドでCanvas APIやDOMを使用して、ブラウザでゲームを描画するようなことができます。

モナド変換子によって、型システムによって作用が追跡される命令型のスタイルで、安全なコードを書くことができることを見してきました。また、型クラスは、コードの再利用を可能にするモナドによって提供される、アクション上の抽象化の強力な方法を提供します。標準的なモナド変換子を組み合わせることにより、`Alternative` や `MonadPlus` のような標準的な抽象化を使用して、役に立つモナドを構築することができました。

モナド変換子は、高階多相や多変数型クラスなどの高度な型システムの機能を利用することによって記述することができ、表現力の高いコードの優れた実演となっています。

次の章では、非同期なJavaScriptのコードにありがちな不満、コールバック地獄の問題に対して、モナド変換子がどのような洗練された解決策を与えるのかを見ていきます。

# 12 コールバック地獄

---

## 12.1 この章の目標

---

この章では、これまでに見てきたモナド変換子やApplicative関手といった道具が、現実世界の問題解決にどのように役立つかを見ていきましょう。ここでは特に、コールバック地獄(callback hell)の問題を解決について見ていきます。

## 12.2 プロジェクトの準備

---

この章のソースコードは、`grunt` でコンパイルし、NodeJSを使って実行することができます。

## 12.3 問題

通常、JavaScriptの非同期処理コードでは、プログラムの流れを構造化するためにコールバック(callbacks)を使用します。たとえば、ファイルからテキストを読み取るのに好ましいアプローチとしては、`readFile` 関数を使用し、コールバック、つまりテキストが利用可能になったときに呼び出される関数を渡すことです。

```
function readText(onSuccess, onFailure) {
    var fs = require('fs');
    fs.readFile('file1.txt', { encoding: 'utf-8' }, function (error, data) {
        if (error) {
            onFailure(error.code);
        } else {
            onSuccess(data);
        }
    });
}
```

しかしながら、複数の非同期操作が関与している場合には入れ子になったコールバックを生じることになり、すぐに読めないコードになってしまいます。

```
function copyFile(onSuccess, onFailure) {
    var fs = require('fs');
    fs.readFile('file1.txt', { encoding: 'utf-8' }, function (error, data1) {
        if (error) {
            onFailure(error.code);
        } else {
            fs.writeFile('file2.txt', data, { encoding: 'utf-8' }, function (error) {
                if (error) {
                    onFailure(error.code);
                } else {
                    onSuccess();
                }
            });
        }
    });
}
```

この問題に対する解決策のひとつとしては、独自の関数に個々の非同期呼び出しを分割することです。

```
function writeCopy(data, onSuccess, onFailure) {
    var fs = require('fs');
```



```

    fs.writeFile('file2.txt', data, { encoding: 'utf-8' }, function (error) {
      if (error) {
        onFailure(error.code);
      } else {
        onSuccess();
      }
    });
  }

function copyFile(onSuccess, onFailure) {
  var fs = require('fs');
  fs.readFile('file1.txt', { encoding: 'utf-8' }, function (error, data) {
    if (error) {
      onFailure(error.code);
    } else {
      writeCopy(data, onSuccess, onFailure);
    }
  });
}

```

この解決策は一応は機能しますが、いくつか問題があります。

- 上で `writeCopy` へ `data` を渡したのと同じ方法で、非同期関数に関数の引数として途中の結果を渡さなければなりません。これは小さな関数についてはうまくいきますが、多くのコールバック関係する場合はデータの依存関係は複雑になることがあり、関数の引数が大量に追加される結果になります。
- どんな非同期関数でもコールバック `onSuccess` と `onFailure` が引数として定義されるという共通のパターンがありますが、このパターンはソースコードに付随したモジュールのドキュメントに記述することで実施しなければなりません。このパターンを管理するには型システムのほうがよいですし、型システムで使い方を強制しておくほうがいいでしょう。

次に、これらの問題を解決するために、これまでに学んだ手法を使用する方法について説明していきます。

## 12.4 継続モナド

`copyFile` の例をFFIを使ってPureScriptへと翻訳していきましょう。PureScriptで書いていくにつれ、計算の構造はわかりやすくなり、`purescript-transformers` パッケージで定義されている継続モナド変換子 `ContT` が自然に導入されることになるでしょう。

まず、FFIを使って `readFile` と `writeFile` に型を与えなくてはなりません。型同義語をいくつかと、ファイル入出力のための作用を定義することから始めましょう。

```
foreign import data FS :: !

type ErrorCode = String
type FilePath = String
```

`readFile` はファイル名と2引数のコールバックを引数に取ります。ファイルが正常に読み込まれた場合は、2番目の引数にはファイルの内容が含まれますが、そうでない場合は、最初の引数がエラーを示すために使われます。

今回は `readFile` を2つのコールバックを引数としてとる関数としてラップすることになります。先ほどの `copyFile` や `writeCopy` とまったく同じように、エラーコールバック(`onFailure`)と結果コールバック(`onSuccess`)の2つです。簡単のために `Data.Function` の多引数関数の機能を使うと、このラップされた関数 `readFileImpl` は次のようになります。

```
foreign import readFileImpl
  "function readFileImpl(path, onSuccess, onFailure) {\
    \ return function() {\
    \   require('fs').readFile(path, \
    \     { encoding: 'utf-8' }, \
    \     function(error, data) {\
    \       if (error) {\
    \         onFailure(error.code());\
    \       } else {\
    \         onSuccess(data());\
    \       }\
    \     }\
    \   );\
    \ };\
  \}" :: forall eff. Fn3 FilePath
      (String -> Eff (fs :: FS | eff) Unit)
      (ErrorCode -> Eff (fs :: FS | eff) Unit)
      (Eff (fs :: FS | eff) Unit)
```

`readFileImpl` はファイルパス、成功時のコールバック、失敗時のコールバックという3つの引数を取り、空(`Unit`)の結果を返す副作用のある計算を返す、ということをこの型は言っています。コールバック自身にも、その作用を追跡するために `Eff` モナドを使うような型が与えられていることに注意してください。

この `readFileImpl` の実装がその型の正しい実行時表現を持っている理由を、よく理解しておくようにしてください。

`writeFileImpl` もよく似ています。違いはファイルがコールバックではなく関数自身に渡されるということだけです。実装は次のようになります。

```
foreign import writeFileImpl
  "function writeFileImpl(path, data, onSuccess, onFailure) {\
    \ return function() {\
    \   require('fs').writeFile(path, data, \
    \     { encoding: 'utf-8' }, \
    \     function(error) {\
    \       if (error) {\
    \         onFailure(error.code());\
    \       } else {\
    \         onSuccess();\
    \       }\
    \     }\
    \   );\
    \ }; \
  \}" :: forall eff. Fn4 FilePath
    String
    (Eff (fs :: FS | eff) Unit)
    (ErrorCode -> Eff (fs :: FS | eff) Unit)
    (Eff (fs :: FS | eff) Unit)
```

これらのFFIの宣言が与えられれば、`readFile` と `writeFile` の実装を書くことができます。`Data.Function` ライブラリを使って、多引数のFFIバインディングを通常の(カーリー化された)PureScript関数へと変換するので、もう少し読みやすい型になるでしょう。

さらに、成功時と失敗時の2つの必須のコールバックに代わって、成功か失敗のどちらか(Either) に対応した単一のコールバックを要求するようにします。つまり、新しいコールバックは引数として `Either ErrorCode` モナドの値をとります。

```
readFile :: forall eff.
  FilePath ->
  (Either ErrorCode String -> Eff (fs :: FS | eff) Unit) ->
  Eff (fs :: FS | eff) Unit
readFile path k =
  runFn3 readFileImpl
    path
    (k <<< Right)
    (k <<< Left)

writeFile :: forall eff.
  FilePath ->
  String ->
  (Either ErrorCode Unit -> Eff (fs :: FS | eff) Unit) ->
  Eff (fs :: FS | eff) Unit
writeFile path text k =
  runFn4 writeFileImpl
```

```
path
text
(k $ Right unit)
(k <<< Left)
```

**Eff** モナドはこれらの型シグネチャの両方に現れます。次のような新しい型同義語を導入すると、型を簡素化できます。

```
type M eff = Eff (fs :: FS | eff)

readFile :: forall eff.
  FilePath ->
  (Either ErrorCode String -> M eff Unit) ->
  M eff Unit

writeFile :: forall eff.
  FilePath ->
  String
  (Either ErrorCode Unit -> M eff Unit) ->
  M eff Unit
```

ここで、重要なパターンを見つけることができます。これらの関数は何らかのモナド(この場合は **M eff**)で値を返すコールバックをとり、同一のモナドで値を返します。これは、最初のコールバックが結果を返したときに、そのモナドは次の非同期関数の入力に結合するためにその結果を使用することができることを意味しています。実際、**copyFile** の例で手作業でやったことがまさにそれです。

これは **purescript-transformers** の **Control.Monad.Cont.Trans** モジュールで定義されている継続モナド変換子(continuation monad transformer)の基礎となっています。

**ContT** は次のようなnewtypeとして定義されます。

```
newtype ContT r m a = ContT ((a -> m r) -> m r)
```

継続(continuation)はコールバックの別名です。継続は計算の残余(remainder)を捕捉します。ここで「残余」とは、非同期呼び出しが行われ、結果が提供された後に起こることを指しています。

**ContT** データ構築子の引数は **readFile** と **writeFile** の型ととてもよく似ています。実際、もし型 **a** を型 **Either ErrorCode String**、**r** を **Unit**、**m** をモナド **M eff** というように選ぶと、**readFile** の型の右辺を復元することができます。

今回の目的では `Eff` モナドを変換するために常に `ContT` を使い、型 `r` は常に `Unit` になりますが、このことは必須ではありません。

`ContT` 構築子を適用するだけで、`readFile` と `writeFile` を `ContT Unit (M eff)` モナドの計算として扱うことができます。

```
type C eff = ContT Unit (M eff)

readFileCont :: forall eff.
  FilePath ->
  C eff (Either ErrorCode String)
readFileCont path = ContT $ readFile path

writeFileCont :: forall eff.
  FilePath ->
  String ->
  C eff (Either ErrorCode Unit)
writeFileCont path text = ContT $ writeFile path text
```

ここで `ContT` モナド変換子に対して `do` 記法を使うだけで、ファイル複製処理を書くことができます。

```
copyFileCont :: forall eff. FilePath -> FilePath -> C eff (Either ErrorCode Unit)
copyFileCont src dest = do
  e <- readFileCont src
  case e of
    Left err -> return $ Left err
    Right content -> writeFileCont dest content
```

`readFileCont` の非同期性が `do` 記法によってモナドの束縛に隠されていることに注目してください。これはまさに同期的なコードのように見えますが、`ContT` モナドは非同期関数を書くのを手助けしているのです。

継続を与えて `runContT` ハンドラを使うと、この計算を実行することができます。この継続は次に何をするか、例えば非同期なファイル複製処理が完了した時に何をするか、を表しています。この簡単な例では、型 `Either ErrorCode Unit` の結果をコンソールに出力する `print` 関数を単に継続として選んでいます。

```
import Debug.Trace

import Control.Monad.Eff
import Control.Monad.Cont.Trans
```

```
main = runContT
  (copyFileCont "/tmp/1.txt" "/tmp/2.txt")
  print
```

## 演習

1. (簡単) `readFileCont` と `writeFileCont` を使って、2つのテキストファイルを連結する関数を書いてください。
2. (やや難しい) FFIを使って、`setTimeout` 関数に適切な型を与えてください。また、`ContT` モナド変換子を使った次のようなラッパー関数を書いてください。

```
type Milliseconds = Number

foreign import data Timeout :: !

setTimeoutCont :: forall eff.
  Milliseconds ->
  ContT Unit (Eff (timeout :: Timeout | eff)) Unit
```

## 12.5 ErrorTを機能させる

この方法はうまく動きますが、まだ改良の余地があります。

`copyFileCont` の実装において、次に何をするかを決定するためには、パターン照合を使って(型 `Either ErrorCode String` の) `readFileCont` 計算の結果を解析しなければなりません。しかしながら、`Either` モナドは対応するモナド変換子 `ErrorT` を持っていることがわかっているので、`ErrorT` を使って非同期計算とエラー処理の2つの作用を結合できると期待するのは理にかなっています。

実際にそれは可能で、`ErrorT` の定義を見ればそれがなぜかがわかります。

```
newtype ErrorT e m a = ErrorT (m (Either e a))
```

`ErrorT` は基礎のモナドの結果を単純に `a` から `Either e a` に変更します。現在のモナドスタックを `ErrorT ErrorCode` 変換子で変換するように、`copyFileCont` を書き換えることができることを意味します。それは現在の方法に `ErrorT` データ構築子を適用するだけなので簡単です。型同義語を与えると、ここでも型シグネチャを整理することができます。

```

type EC eff = ErrorT ErrorCode (C eff)

readFileContErr :: forall eff. FilePath -> EC eff String
readFileContErr path = ErrorT $ readFileCont path

writeFileContErr :: forall eff. FilePath -> String -> EC eff Unit
writeFileContErr path text = ErrorT $ writeFileCont path text

```

非同期エラー処理が `ErrorT` モナド変換子の内部に隠されているので、このファイル複製処理ははるかに単純になります。

```

copyFileContErr :: forall eff. FilePath -> FilePath -> EC eff Unit
copyFileContErr src dest = do
  content <- readFileContErr src
  writeFileContErr dest content

```

## 演習

1. (やや難しい) 任意のエラーを処理するために、`ErrorT` を使用して2つのファイルを連結しする先ほどの解決策を書きなおしてください。

## 12.6 HTTPクライアント

`ContT` を使って非同期機能进行处理する例として、この章のソースコードの `Network.HTTP.Client` モジュールについても見ていきましょう。このモジュールでは、NodeJSの非同期HTTPリクエストをラップするために継続を使っています。

`http` モジュールを使った典型的な `GET` リクエストは次のようになります。

```

function getRequest(onChunk, onComplete) {
  return function() {
    require('http').request({
      host: 'www.purescript.org',
      path: '/'
    }, function(res) {
      res.setEncoding('utf8');
      res.on('data', function (chunk) {
        onChunk(chunk);
      });
      res.on('end', function () {
        onComplete();
      });
    });
  };
}

```

```

    });
  }).end();
};
}

```

`http` モジュールの `request` メソッドは、ホストとパスを指定するオブジェクトをとり、レスポンスオブジェクトを返します。レスポンスオブジェクトは今回扱う2種類のイベントを發します。

- レスポンスの新しいチャンクが使用可能であることを示す `data` イベント
- レスポンスが完了したことを示す `end` イベント

上の例では、`data` と `end` イベントが發生した時に呼び出される2つのコールバック `onChunk` と `onComplete` を渡しています。

`Network.HTTP.Client` モジュールでは、`request` メソッドは以下のようなAPIを持つ関数 `getImpl` としてラップされています。

```

foreign import data HTTP :: !

type WithHTTP eff = Eff (http :: HTTP | eff)

newtype Request = Request
  { host :: String
  , path :: String
  }

newtype Chunk = Chunk String

getImpl :: forall eff.
  Fn3 Request
    (Chunk -> WithHTTP eff Unit)
    (WithHTTP eff Unit)
    (WithHTTP eff Unit)

```

再び `Data.Function` モジュールを使って、これを通常のカリー化されたPureScript関数に変換します。先ほどと同じように、2つのコールバックを型 `Maybe Chunk` の値を受け入れるひとつのコールバックに変換しています。コールバックに渡された `Nothing` の値は `end` イベントに対応しており、`Just chunk` の値は `data` イベントに対応しています。

```

getChunk :: forall eff.
  Request ->
  (Maybe Chunk -> WithHTTP eff Unit) ->
  WithHTTP eff Unit
getChunk req k =

```



```
runFn3 getImpl
      req
      (k <<< Just)
      (k Nothing)
```

ここでも `ContT` データ構築子を適用することにより、この非同期関数をこの継続モナドの演算に変換しています。

```
getCont :: forall eff.
  Request ->
  ContT Unit (WithHTTP eff) (Maybe Chunk)
getCont req = ContT $ getChunk req
```

`readFile` の例では、ファイルの内容が利用可能になったとき(または、エラーが発生したとき)、コールバックは一度だけ呼ばれていました。しかし今度は、レスポンスのそれぞれのチャンクについて1回ずつ、複数回コールバックが呼び出されることが期待されます。

## 演習

1. (やや難しい) `runContT` を使ってHTTP応答の各チャンクをコンソールへ出力することで、`getCont` を試してみてください。
2. (難しい) `getImpl` と `getCont` 関数は非同期エラーを処理しません。`getImpl` を `error` イベントに対応するよう変更し、`ErrorT` を使って非同期エラーを表現する `getCont` の亜種を書いてください。

ヒント: `readFile` の例で取ったのと同じアプローチに従うことができます。

## 12.7 チャンク応答の畳み込み

これでHTTP応答の個々のチャンクを集めることができるようになりましたが、すべての応答が利用可能になったときだけ継続が呼び出される非同期関数を作ると便利な時があるかもしれません。このような関数を実装する方法のひとつは、HTTP応答のチャンクに対する畳み込みを書くことです。

継続に渡された複数の結果を畳み込む関数 `foldC` を書きましょう。`foldC` 関数はこの章のソースコードの `Control.Monad.Cont.Extras` モジュールで定義されています。

累積値を追跡するために、`Eff` モナドで `Ref` 作用を使います。次の型同義語を使って型シグネチャを整理します。

```
type WithRef eff = Eff (ref :: Ref | eff)

type ContRef eff = ContT Unit (WithRef eff)
```

これらの同義語を使うと、`foldC` には次のような型を与えることができます。

```
foldC :: forall eff a b r.
  (b -> a -> Either b r) ->
  b -> ContRef eff a -> ContRef eff r
```

`foldC` に渡された関数は、現在の累積値と継続に渡された値を受け取り、新しい累積値か新しい継続に渡される結果のどちらかを返します。

`foldC` が実装されれば、応答のデータ本体の様々なチャンクの収集を可能にする簡単な関数 `collect` を書くことができます。

```
collect :: forall eff a.
  ContRef eff (Maybe a) ->
  ContRef eff [a]
collect = foldC f []
  where
    f xs Nothing = Right xs
    f xs (Just x) = Left (xs ++ [x])
```

`foldC` の実装では、累積値の初期値を持つ新しい参照を作成して開始します。この参照は、コールバックの本体でそれが変更されるときに、累積器を追跡し続けるために使用されます。

```
foldC f b0 c = do
  current <- lift $ newRef b0
```

また、`foldC` は現在の継続とともに呼び出す(call with current continuation)を略して `callCC` と呼ばれる関数を使っています。`callCC` は引数として関数をひとつ取り、この関数は現在の継続、つまり現在の`do`記法ブロックの `callCC` のあとのコードを表しています。現在の継続に返り値を渡すと、`callCC` 内のコードのブロックからいつでも早期に返ることができます。

```
callCC $ \k -> quietly $ do
```

ここで `k` は現在の継続です。これは `foldC` を定義する `do` 記法ブロックの最後の式であるため、現在の継続は実際には `foldC` に渡されたちょうどその継続です。畳み込み関数の結果が累積の結果を表しているとき、`foldC` の最終的な値にこれを使います。

`quietly` コンビネータは `where` 宣言で定義されており、あとでその定義について見ていきます。`quietly` コンビネータの役目は、ここで明示的に `k` を呼び出さない場合に、`callCC` の内側のコードがその継続へ値を返すのを妨げることです。これが必要な理由はずぐに明らかになるはずです。

次に、`foldC` は非同期関数 `c` の結果を名前 `a` に束縛します。

```
a <- c
```

もとの計算によって新しい値が非同期に生成されたとき(この場合は、HTTP応答の新しいチャンクが利用可能になったとき)、この行の後ろのコードが実行されるでしょう。それが起こるとき、畳み込み関数を適用したいので、次のように累積器の現在の値を読み取る必要があります。

```
r <- lift $ readRef current
```

最後に、畳み込み関数を評価し、その結果に応じて2つの場合に場合分けします。畳み込み関数が新しい累積値を返すなら、参照を新しい値で更新します。畳み込み関数が結果を返すなら、これを継続 `k` に渡します。

```
case f b a of
  Left next -> lift $ writeRef current next
  Right r   -> k r
```

ここで `quietly` 関数が必要だった理由が明らかになったと思います。`callCC` 内部のコードの結果を `quietly` 関数で黙らせなかったら、畳み込み関数が `Left` 構築子で包んだ値を返すとき、型 `r` の結果を生成しなければならなくなっただでしょう。しかし、そのような結果を生成する方法は一切ありません！

`quietly` 関数の定義は次のようになっています。`quietly` は非同期関数の結果の型を変更できるようにします。これは継続関数を変換することを可能にする `withContT` 関数を使って書かれています。

```
where
  quietly :: forall m a b. (Monad m) => ContT Unit m a -> ContT Unit m b
```

```
quietly = withContT (\_ _ -> return unit)
```

`foldC` 関数とその亜種 `collect` は特に、チャンクが利用可能になった時に連結することで、完全なHTTP応答本体を累積することを可能にします。

```
newtype Response = Response [Chunk]

getAll :: forall eff.
  Request ->
  ContT Unit (WithHTTP (ref :: Ref | eff)) Response
getAll req = Response <$> collect (getCont req)
```

これで、`String` として応答本体を次のように取得することができます。

```
getResponseText :: forall eff.
  Request ->
  ContT Unit (WithHTTP (ref :: Ref | eff)) String
getResponseText req = responseToString <$> getAll req
where
  responseToString :: Response -> String
  responseToString (Response chunks) = joinWith "" $ map runChunk chunks
```

例えば、次のように継続の中で `getResponseText` と `trace` アクションを使えば、HTTP応答本体の長さをコンソールに出力することができます。

```
main = runContT (getResponseText request) $ \response -> do
  let responseLength = length response
  trace responseLength

  where
    request :: Request
    request = Request
      { host: "www.purescript.org"
      , path: "/"
      }
```

これはうまく動作しますが、次の演習で見るように、もっと慎重に `foldC` を使うとこの方法を改良することができます。

## 演習

1. (やや難しい) `writeFileCont` を使用して、ディスク上のファイルにそのHTTP要

求の応答本体を保存する関数を書いてください。

2. (難しい) 長さを決定するのに、メモリ内のHTTP応答本体全体を連結する必要はありません。チャンクが利用可能になるたびにそのバイトサイズを調べるようにすれば、応答全体のサイズから単一のチャンクのサイズへと、この関数のメモリ使用量を低減することができます。

`collect` の代わりに `foldC` を直接使って、このコード例を書きなおしてください。

## 12.8 並列計算

`ContT` モナドと`do`記法を使って、非同期計算を順番に実行されるように合成する方法を見ました。非同期計算を並列に合成することもできたら便利でしょう。

`Eff` モナドを変換するために `ContT` を使用している場合、単に2つの計算のうち一方を開始した後に他方の計算を開始すれば、並列に計算することができます。

次のような型シグネチャを持つ関数を書きましょう。

```
par :: forall a b r eff.  
    (a -> b -> r) ->  
    ContRef eff a -> ContRef eff b -> ContRef eff r
```

`par` は、2つの非同期計算とその結果を合成する関数を取り、並列に計算を実行し結果を合成するような単一の計算を返します。

(`Ref` 作用で)変更可能な参照を使い、呼び出された2つの継続を追跡します。両方の結果が返ってきたとき、最終的な結果を計算し、メインの継続に渡すことができます。

直接 `ContT` データ構築子で値を構築すると、`par` を最も簡単に実装できます。

```
par f ca cb = ContT $ \k -> do
```

ここで `f` は合成を行う関数で、`ca` と `cb` はそれぞれ型 `a` と `b` の値を返す非同期的な計算です。 `k` は `ca` と `cb` の両方が完了した時に型 `r` の値を返すのに使う継続です。

利用可能になったときに `ca` と `cb` の結果を保持するために、2つの新しい参照を作成することから始めます。

```
ra <- newRef Nothing  
rb <- newRef Nothing
```

これらの参照 `ra` と `rb` は、それぞれ型 `Maybe a` と `Maybe b` の値を保持します。どちらも最初は `Nothing` の値が格納されていますが、計算が完了したとき値が更新されます。

次に、`runContT` を使用して最初の非同期計算を開始します。

```
runContT ca $ \a -> do
  mb <- readRef rb
  case mb of
    Nothing -> writeRef ra $ Just a
    Just b -> k (f a b)
```

第二の値が利用可能であるかどうかを調べるする継続を提供します。そうである場合は、継続 `k` に最終結果を渡すために結合関数を使用します。そうでなければ、単に最初の値を含むように参照 `ra` を更新します。

ふたつめの計算についても同様です。

```
runContT cb $ \b -> do
  ma <- readRef ra
  case ma of
    Nothing -> writeRef rb $ Just b
    Just a -> k (f a b)
```

`par` コンビネータを使うと、ふたつのファイルを並列に読んだり、2つのHTTP要求を平行して発行し、並列に結果を待つことができます。

2つのテキストファイルを並列に読み取り、連結してその結果を出力する簡単な例は次のようになります。

```
import Control.Apply (lift2)

main = flip runContT print $
  par (lift2 (++)) (readFileCont "/tmp/1.txt")
    (readFileCont "/tmp/2.txt")
```

`readFileCont` は型 `Either Errorcode String` の値を返すので、結合関数を作るには `lift2` を使って演算子 `(++)` を `Either` 型構築子まで持ち上げなければいけません。

## 演習

1. (簡単) `par` を使用して、2つのHTTP要求を作成し、並列に応答本体を集めてください。結合関数は2つの応答本体を連結する必要があり、継続は `trace` を使用してコンソールに結果を出力しなくてはなりません。
2. (やや難しい) 2つの計算を並列に実行し、先に完了したほうの計算の結果を返す次のような関数を書いてください。

```
race :: forall a eff.  
  ContRef eff a ->  
  ContRef eff a ->  
  ContRef eff a
```

ヒント: 結果が返されたかどうかを示す `Boolean` を格納する参照を使ってみましょう。

3. (やや難しい) `race` 関数を `setTimeoutCont` 関数と一緒に使って、次のような関数を定義してください。

```
timeout :: forall a eff.  
  Milliseconds ->  
  ContRef eff a ->  
  ContRef eff (Maybe a)
```

この関数は指定された計算が与えられたミリ秒以内に結果を返さないなら `Nothing` を返します。

## 12.9 並列処理のための `Applicative` 関手

`par` コンビネータの型は `ContRef eff` モナドについての `lift2` の型にとっても良く似ています。実際に、`par` が厳密に `lift2` であるような新しい `Applicative` 関手を定義することは可能で、`par` と `ContRef eff` に関してこれを簡単に定義することができます。

`par` に関する `ContRef eff` の `Applicative` インスタンスを定義していないのはなぜかと不思議に思われるかもしれません。これには2つの理由があります。

- 型コンストラクタが `Monad` インスタンスも持っている場合、それは通常 `(<*>)` が以下の関数と同値であるという意味で、`Monad` と `Applicative` インスタンスは一致しています。

```

ap :: forall m a b. (Monad m) => m (a -> b) -> m a -> m b
ap mf ma = do
  f <- mf
  a <- ma
  return (f a)

```

しかしながら、この仮定的な `Applicative` インスタンスは、並列性に関して `Monad` インスタンスとは異なるでしょう。`(<*>)` は引数を並列に評価するのに対して、`ap` は2番目の計算を実行する前に、最初の計算の完了を待つからです。

- PureScriptは行型を含む型の型クラスのインスタンスを許可していません。`Eff` モナドは作用の行によってパラメータ化され、その場合、行は `Ref` 作用を含まなければならぬので、`ContRef` についての `Applicative` インスタンスを定義することは不可能なのです。

その代わりに、`Parallel eff` と呼ばれる `ContRef eff` のnewtypeラッパーを次のように作成します。

```

newtype Parallel eff a = Parallel (ContRef eff a)

```

単に外側のデータ構築子を除去することで、`Parallel` 計算を `ContRef eff` モナドにおける演算に変換する関数を書くことができます。

```

runParallel :: forall eff a. Parallel eff a -> ContRef eff a
runParallel (Parallel c) = c

```

型クラスのインスタンスは、大部分は `ContT` の対応するインスタンスから複製することができます。しかし、`Apply` 型クラスの場合には、`(<*>)` を再定義するために `par` を使用してください。

```

instance functorParallel :: Functor (Parallel eff) where
  (<$>) f (Parallel c) = Parallel (f <$> c)

instance applyParallel :: Apply (Parallel eff) where
  (<*>) (Parallel f) (Parallel x) = Parallel (par ($) f x)

instance applicativeParallel :: Applicative (Parallel eff) where
  pure a = Parallel $ pure a

```

`Apply` インスタンスの定義では、結合関数として関数適用 `($)` を使って、関数をその引数と結合するために `par` を使っています。



`Parallel` 型構築子を使用して並列に二つのファイルを読むように上の例を書き直すことができますようになりました。

```
import Control.Apply (lift2)

main = flip runContT print $ runParallel $
  lift2 (++) <$> Parallel (readFileCont "/tmp/1.txt")
  <*> Parallel (readFileCont "/tmp/2.txt")
```

Applicative関手では任意個引数の関数の持ち上げができるので、このApplicativeコンビネータを使ってより多くの計算を並列に実行することができます。`traverse` と `sequence` のようなApplicative関手を扱うすべての標準ライブラリ関数から恩恵を受けることもできます。

必要に応じて `Parallel` と `runParallel` を使って型構築子を変更することで、`do` 記法ブロックのApplicativeコンビネータを使って、直列的なコードの一部で並列計算を結合したり、またはその逆を行ったりすることができます。

## 演習

1. (簡単) `traverse` 関数を使って、ファイルの名前の配列を与えるとその内容を並列に読み取り、内容の文字列表現の配列を返す関数 `readMany` を書いてください。
2. (簡単) `race` コンビネータを使って、`Parallel eff` の `Alt` インスタンスを書いてください。`Alternative` のインスタンスは作れるでしょうか。
3. (やや難しい) `lift2` で `(++)` を持ち上げる代わりに、`ErrorT` を使ってエラー処理を行うように、並列ファイル入出力の例を書きなおしてください。解決策は `Parallel` 関手を変換するために `ErrorT` 変換子を使用しなければいけません。

同様の手法で `readMany` 関数を書き換えてください。

4. (難しい、拡張) ディスク上のJSON文書のコレクションが与えられ、それぞれの文書はディスク上の他のファイルへの参照の配列を含んでいるとします。

```
{ references: ['/tmp/1.json', '/tmp/2.json'] }
```

入力として単一のファイル名をとり、そのファイルから参照されているディスク上のすべてのJSONファイルをたどって、参照されたすべてのファイルの一覧を収集するユーティリティを書いてください。

そのユーティリティは、JSON文書を解析するために `purescript-foreign` ライブラリを使用する必要があり、単一のファイルが参照するファイルは並列に取得しなければなりません！

## 12.10 まとめ

---

この章ではモナド変換子の実用的なデモンストレーションを見てきました。

- コールバック渡しの一般的なJavaScriptのイディオムを `ContT` モナド変換子によって捉えることができる方法を説明しました。
- どのようにコールバック地獄の問題を解決するかを説明しました。 直列の非同期計算を表現するdo記法を使用して、かつ並列性を表現するためにApplicative関手によって解決することができる方法を説明しました。
- 非同期エラーを表現するために `ErrorT` を使いました。

## 13 テストの自動生成

---

### 13.1 この章の目標

---

この章では、テストングの問題に対する、型クラスの特に洗練された応用について示します。どのようにテストするかをコンパイラに教えるのではなく、コードがどのような性質を持っているべきかを教えることでテストします。型クラスを使って無作為データ生成のための定型コードを隠し、テストケースを仕様から無作為に生成することができます。これは生成的テストング(generative testing、またはproperty-based testing)と呼ばれ、Haskellの[QuickCheck](#)ライブラリによって知られるようになった手法です。

`purescript-quickcheck` パッケージはHaskellのQuickCheckをライブラリをPureScriptにポーティングしたもので、型や構文はもとのライブラリとほとんど同じようになっています。`purescript-quickcheck` を使って簡単なライブラリをテストし、Gruntでテストスイートを自動化されたビルドに統合する方法を見ていきます。

### 13.2 プロジェクトの準備

---

この章のプロジェクトにはBower依存関係として `purescript-quickcheck` が追加されます。実際には、`purescript-quickcheck` は `bower.json` の `devDependencies` セクションに追加されます。

```
"devDependencies": {
  "purescript-quickcheck": "~0.1.3"
}
```

これは `purescript-quickcheck` は開発時のみ必要であることを示しています。製品ビルドのときは QuickCheck ライブラリコードを出力に含むのを避けるため、`bower` コマンドで `--production` フラグを使ってください。

```
$ bower update --production
```

```
$ grunt build
```

## 13.3 テストの自動化

このプロジェクトの `Gruntfile.js` ファイルは、テストスイートをサポートするために少し変更されています。

まず、`psc` タスクに新しいセクションが追加されています。これはソースコードをテストスイートのコードと一緒にビルドし、副次的なJavaScriptファイルを出力するようにします。

```
psc: {
  lib: {
    src: ["<%=srcFiles%>"],
    dest: "dist/Main.js"
  },
  tests: {
    options: {
      module: ["Main"],
      main: true
    },
    src: ["tests/Main.purs", "<%=srcFiles%>"],
    dest: "dist/tests.js"
  }
}
```

`psc:tests` タスクはテストスイートを実行するために追加の `dist/tests.js` ファイルを生成するようになっています。次の手順は、`grunt-execute` パッケージを使って、このプロセスを自動化することです。

```
execute: {
```

```
    tests: {
      src: "dist/tests.js"
    }
  }
}
```

`grunt-execute` パッケージもNPMの依存関係として追加されます。最後に、Gruntタスクリストにタスクとしてこれを追加する必要があります。

```
grunt.loadNpmTasks("grunt-execute");

grunt.registerTask("build",
  ["psc:lib", "dotPsci"]);
grunt.registerTask("test",
  ["build", "psc:tests", "execute:tests"]);
```

これで、ライブラリのソースコードだけをビルドする `build`、ライブラリとテストスイートをビルドしテストも実行する `test` の、2つのタスクが新しく利用できるようになります。

## 13.4 プロパティの書き込み

`Merge.purs` ファイルでは `purescript-quickcheck` ライブラリの機能を実演するために使う簡単な関数 `merge` が実装されています。

```
merge :: [Number] -> [Number] -> [Number]
```

`merge` は2つのソートされた数の配列をとって、その要素を統合し、ソートされた結果を返します。例えば次のようになります。

```
> :i Merge
> merge [1, 3, 5] [2, 4, 6]

[1, 2, 3, 4, 5, 6]
```

典型的なテストスイートでは、手作業でこのような小さなテストケースをいくつも作成し、結果が正しい値と等しいことを確認することでテスト実施します。しかし、`merge` 関数について知る必要があるものはすべて、2つの性質に要約することができます。

- （既ソート性） `xs` と `ys` がソート済みなら、`merge xs ys` もソート済みになります。
- （部分配列） `xs` と `ys` はどちらも `merge xs ys` の部分配列で、要素は元の配列と同じ順序で現れます。

`purescript-quickcheck` では、無作為なテストケースを生成することで、直接これらの性質をテストすることができます。コードが持つべき性質を、次のような関数として述べるだけです。

```
main = do
  quickCheck $ \xs ys ->
    isSorted $ merge (sort xs) (sort ys)
  quickCheck $ \xs ys ->
    xs `isSubarrayOf` merge xs ys
```

ここで、`isSorted` と `isSubarrayOf` は次のような型を持つ補助関数として実装されています。

```
isSorted :: forall a. (Ord a) => [a] -> Boolean
isSubarrayOf :: forall a. (Eq a) => [a] -> [a] -> Boolean
```

このコードを実行すると、`purescript-quickcheck` は無作為な入力 `xs` と `ys` を生成してこの関数に渡すことで、主張しようとしている性質を反証しようとします。何らかの入力に対して関数が `false` を返した場合、性質は正しくないことが示され、ライブラリはエラーを発生させます。幸いなことに、次のように100個の無作為なテストケースを生成しても、ライブラリはこの性質を反証することができません。

```
$ grunt

Running "execute:tests" (execute) task
-> executing dist/tests.js
100/100 test(s) passed.
100/100 test(s) passed.
-> completed dist/tests.js

>> 1 file and 0 calls executed

Done, without errors.
```

もし `merge` 関数に意図的にバグを混入した場合（例えば、大なりのチェックを小なりのチェックへと変更するなど）、最初に失敗したテストケースの後で例外が実行時に投げられます。

```
Error: Test 1 failed:
  Test returned false
```

このエラーメッセージではあまり役に立ちませんが、これから見ていくように、少しの作業で改良することができます。

## 13.5 エラーメッセージの改善

テストケースが失敗した時に同時にエラーメッセージを提供するには、`purescript-quickcheck` の `<?>` 演算子を使います。次のように性質の定義に続けて `<?>` で区切ってエラーメッセージを書くだけです。

```
quickCheck $ \xs ys ->
  let
    result = merge xs ys
  in
    xs `isSubarrayOf` result <?> show xs ++ " not a subarray of " ++ show re
```

このとき、もしバグを混入するようにコードを変更すると、最初のテストケースが失敗したときに改良されたエラーメッセージが表示されます。

```
Error: Test 6 failed:
  [0.85] not a subarray of [0.89,0.82,0.44,0.01]
```

入力 `xs` が無作為に選ばれた数の配列として生成されていることに注目してください。

### 演習

1. (簡単) 空の配列を持つ配列を統合しても元の配列は変更されない、と主張する性質を書いてください。
2. (簡単) `merge` の残りの性質に対して、適切なエラーメッセージを追加してください。

## 13.6 多相的なコードのテスト

`Merge` モジュールでは、数の配列だけでなく、`Ord` 型クラスに属するどんな型の配列に対しても動作する、`merge` 関数を一般化した `mergePoly` という関数が定義されています。

```
mergePoly :: forall a. (Ord a) => [a] -> [a] -> [a]
```

`merge` の代わりに `mergePoly` を使うように元のテストを変更すると、次のようなエラーメッセージが表示されます。

```
Error in declaration main
  No instance found for Testable ([u1] -> [u1] -> Boolean)
```

このエラーメッセージは、配列に持たせたい要素の型が何なのか分からないので、コンパイラが無作為なテストケースを生成できなかったことを示しています。このような場合、補助関数を使えば、コンパイラが特定の型を推論すること強制できます。例えば、恒等関数の同義語として `numbers` という関数を定義します。

```
numbers :: [Number] -> [Number]
numbers = id
```

それから、コンパイラが引数の2つの配列の型 `[Number]` を推論するように、テストを変更します。

```
quickCheck $ \xs ys ->
  isSorted $ numbers $ mergePoly (sort xs) (sort ys)
quickCheck $ \xs ys ->
  numbers xs `isSubarrayOf` mergePoly xs ys
```

ここで、`numbers` 関数が不明な型を解消するために使われるので、`xs` と `ys` はどちらも型 `[Number]` を持っています。

## 演習

1. (簡単) `xs` と `ys` の型を `[Boolean]` に強制する関数 `bools` を書き、`mergePoly` をその型でテストする性質を追加してください。
2. (やや難しい) 標準関数から(例えば `purescript-arrays` パッケージから)ひとつ関数を選び、適切なエラーメッセージを含めて `QuickCheck` の性質を書いてください。その性質は、補助関数を使って多相型引数を `Number` か `Boolean` のどちらかに固定しなければいけません。

## 13.7 任意のデータの生成

`purescript-quickcheck` ライブラリを使って性質についてのテストケースを無作為に生成

する方法について説明します。

無作為に値を生成することができるような型は、次のような型クラス `Arbitrary` のインスタンスを持っています。

```
class Arbitrary t where
  arbitrary :: Gen t
```

`Gen` 型構築子は決定的無作為データ生成の副作用を表しています。決定的無作為データ生成は、擬似乱数生成器を使って、シード値から決定的無作為関数の引数を生成します。`Gen` はモナドでも `Applicative` 関手でもあるので、`Arbitrary` 型クラスの新しいインスタンスを作成するのに、いつも使っているようなコンビネータを自由に使うことができます。

例えば、`purescript-quickcheck` ライブラリで提供されている `Number` 型の `Arbitrary` インスタンスは、0と1の間に均一に分布した値を生成します。もし異なる分布を持った数生成する `Arbitrary` インスタンスを持った型を定義したい場合は、`Applicative` インスタンスを使うと、関数適用によって、均一な無作為な変数を、次のような無作為な値へと変換することができます。

```
newtype Byte = Byte Number

instance arbitraryByte :: Arbitrary Byte where
  arbitrary = uniformToByte <$> arbitrary
  where
    uniformToByte n = Math.floor (n * 256)
```

ここでは、0から255までの間の整数値であるような型 `Byte` を定義しています。`Arbitrary` インスタンスの `<$>` 演算子を使って、`uniformToByte` 関数を `arbitrary` アクションまで持ち上げています。この型の `arbitrary` アクションの型は `Gen Number` だと推論されますが、これは0から1の間に均一に分布する数生成することを意味しています。

この考え方を `merge` に対しての既ソート性テストを改良するのに使うこともできます。

```
quickCheck $ \xs ys ->
  isSorted $ numbers $ mergePoly (sort xs) (sort ys)
```

このテストでは、任意の配列 `xs` と `ys` を生成しますが、`merge` はソート済みの入力を期待しているので、`xs` と `ys` をソートしておかなければなりません。一方で、ソートされた配列を表す `newtype` を作成し、ソートされたデータを生成する `Arbitrary` インスタンスを書くこともできます。



```

newtype Sorted a = Sorted [a]

sorted :: forall a. Sorted a -> [a]
sorted (Sorted xs) = xs

instance arbSorted :: (Arbitrary a, Ord a) => Arbitrary (Sorted a) where
  arbitrary = Sorted <<< sort <$> arbitrary

```

この型構築子を使うと、テストを次のように変更することができます。

```

quickCheck $ \xs ys ->
  isSorted $ numbers $ mergePoly (sorted xs) (sorted ys)

```

これは些細な変更に見えるかもしれませんが、`xs` と `ys` の型はただの `[Number]` から `Sorted Number` へと変更されています。これにより、`mergePoly` 関数はソート済みの入力を取る、という意図を、わかりやすく示すことができます。理想的には、`mergePoly` 関数自体の型が `Sorted` 型構築子を使うようにするといいいでしょう。

より興味深い例として、`Tree.purs` ファイルでは枝の値でソートされた二分木の型が定義されています。

```

data Tree a
  = Leaf
  | Branch (Tree a) a (Tree a)

```

`Tree` モジュールでは次のAPIが定義されています。

```

insert :: forall a. (Ord a) => a -> Tree a -> Tree a
member :: forall a. (Ord a) => a -> Tree a -> Boolean

toArray :: forall a. Tree a -> [a]
fromArray :: forall a. (Ord a) => [a] -> Tree a

```

`insert` 関数は新しい要素をソート済みの二分木に挿入するのに使われ、`member` 関数は特定の値の有無を木に問い合わせるのに使われます。例えば次のようになります。

```

> :i Tree

> member 2 $ insert 1 $ insert 2 Leaf
true

```

```
> member 1 Leaf
false
```

`toArray` 関数と `fromArray` 関数は、ソートされた木とソートされた配列を相互に変換するために使われます。`fromArray` を使うと、木についての `Arbitrary` インスタンスを書くことができます。

```
instance arbTree :: (Arbitrary a, Ord a) => Arbitrary (Tree a) where
  arbitrary = fromArray <<< sorted <$> arbitrary
```

`fromArray` への入力がソートされた配列であることを保証するために、ここで `sort` 関数を使っていることに注意してください。

型 `a` についての有効な `Arbitrary` インスタンスが存在していれば、テストする性質の引数の型として `Tree a` を使うことができます。例えば、`member` テストは値を挿入した後は常に `true` を返すことをテストできます。

```
quickCheck $ \t a ->
  member a $ insert a (t :: Tree Number)
```

ここでは、引数 `t` は `Tree Number` 型の無作為に生成された木です。

## 演習

1. (やや難しい) `a-z` の範囲から無作為に選ばれた文字の集まりを生成する `Arbitrary` インスタンスを持った、`String` の newtype を作ってください。ヒント: `[Number]` の `Arbitrary` インスタンスと、型 `[Number] -> String` を持つ関数を使ってください。
2. (難しい) 木に挿入された値は、任意に多くの挿入があった後も、その木の構成要素であることを主張する性質を書いてください。

## 13.8 高階関数のテスト

`Merge` モジュールは `merge` 関数についての他の生成も定義します。`mergeAith` 関数は、統合される要素の順序を決定するのに使われる、追加の関数を引数としてとります。つまり `mergeWith` は高階関数です。

例えば、すでに長さの昇順になっている2つの配列を統合するのに、`length` 関数を最初

の引数として渡します。このとき、結果も長さの昇順になっていなければなりません。

```
> :i Data.String
> mergeWith length
  ["", "ab", "abcd"]
  ["x", "xyz"]

["", "x", "ab", "xyz", "abcd"]
```

このような関数をテストするにはどうしたらいいでしょうか。理想的には、関数であるような最初の引数を含めた、3つの引数すべてについて、値を生成したいと思うでしょう。

関数を生産的に生成せざるようにする、もうひとつの型クラスがあります。この型クラスは `CoArbitrary` と呼ばれており、次のように定義されています。

```
class CoArbitrary t where
  coarbitrary :: forall r. t -> Gen r -> Gen r
```

`coarbitrary` 関数は、型 `t` と、関数の結果の型 `r` についての無作為な生成器を関数の引数としてとり、無作為な生成器をかき乱すのにこの引数を使います。つまり、この引数を使って、乱数生成器の無作為な出力を変更しているのです。

また、もし関数の定義域が `CoArbitrary` で、値域が `Arbitrary` なら、`Arbitrary` の関数を与える型クラスインスタンスが存在しています。

```
instance arbFunction :: (CoArbitrary a, Arbitrary b) => Arbitrary (a -> b)
```

実は、これが意味しているのは、引数として関数を取るような性質を記述できるということです。`mergeWith` 関数の場合では、新しい引数を考慮するようにテストを修正すると、最初の引数を生産的に生成することができます。

既ソート性の性質については、必ずしも `Ord` インスタンスを持っているとは限らないので、結果がソートされているということを保証することができませんが、引数として渡す関数 `f` にしたがって結果がソートされている期待することはできます。さらに、2つの入力配列が `f` に従ってソートされている必要がありますので、`sortBy` 関数を使って関数 `f` が適用されたあとの比較に基づいて `xs` と `ys` をソートします。

```
quickCheck $ \xs ys f ->
  isSorted $
    map f $
```

```
mergeWith (numberToBool f)
  (sortBy (compare `on` f) xs)
  (sortBy (compare `on` f) ys)
```

ここでは、関数 `f` の型を明確にするために、関数 `numberToBool` を使用しています。

```
numberToBool :: (Number -> Boolean) -> Number -> Boolean
numberToBool = id
```

部分配列性については、単に関数の名前を `mergeWith` に変えるだけです。引き続き入力配列は結果の部分配列になっていると期待できます。

```
quickCheck $ \xs ys f ->
  xs `isSubarrayOf` mergeWith (numberToBool f) xs ys
```

関数は `Arbitrary` であるだけでなく `CoArbitrary` でもあります。

```
instance coarbFunction :: (Arbitrary a, CoArbitrary b) => CoArbitrary (a -> b)
```

これは値の生成が単純な関数だけに限定されるものではないことを意味しています。つまり、高階関数や、引数が高階関数であるような関数すら無作為に生成することができるのです。

## 13.9 CoArbitrary のインスタンスを書く

`Gen` の `Monad` や `Applicative` インスタンスを使って独自のデータ型に対して `Arbitrary` インスタンスを書くことができるのと同じように、独自の `CoArbitrary` インスタンスを書くこともできます。これにより、無作為に生成される関数の定義域として、独自のデータ型を使うことができるようになります。

`Tree` 型の `CoArbitrary` インスタンスを書いてみましょう。枝に格納されている要素の型に `CoArbitrary` インスタンスが必要になります。

```
instance coarbTree :: (CoArbitrary a) => CoArbitrary (Tree a) where
```

型 `Tree a` の値を与えられた乱数発生器をかき乱す関数を記述する必要があります。入力値が `Leaf` であれば、そのままの生成器を返します。

```
coarbitrary Leaf = id
```

もし木が `Branch` なら、関数合成で独自のかき乱し関数を作ることにより、左の部分木、値、右の部分木を使って生成器をかき乱します。

```
coarbitrary (Branch l a r) =  
  coarbitrary l <<<  
  coarbitrary a <<<  
  coarbitrary r
```

これで、木を引数にとるような関数を含む性質を自由に書くことができるようになりました。たとえば、`Tree` モジュールでは述語が引数のどんな部分木についても成り立っているかを調べる関数 `anywhere` が定義されています。

```
anywhere :: forall a. (Tree a -> Boolean) -> Tree a -> Boolean
```

これで、無作為にこの述語関数 `anywhere` を生成することができるようになりました。例えば、`anywhere` 関数が次のようなある命題のもとで不変であることを期待します。

```
quickCheck $ \f g t ->  
  anywhere (\s -> f s || g s) t ==  
  anywhere f (treeOfNumber t) || anywhere g t
```

ここで、`treeOfNumber` 関数は木に含まれる値の型を型 `Number` に固定するために使われています。

```
treeOfNumber :: Tree Number -> Tree Number  
treeOfNumber = id
```

## 13.10 副作用のないテスト

テストの目的では通常、テストスイートの `main` アクションには `quickCheck` 関数の呼び出しが含まれています。しかし、副作用を使わない `quickCheckPure` と呼ばれる `quickCheck` 関数の亜種もあります。`quickCheckPure` は、入力として乱数の種をとり、テスト結果の配列を返す純粋な関数です。

`psci` を使用して `quickCheckPure` を使ってみましょう。ここでは `merge` 操作が結合法則を

満たすことをテストしてみます。

```
> :i Test.QuickCheck
> :i Merge

> quickCheckPure 12345 10 $ \xs ys zs ->
  ((xs `merge` ys) `merge` zs) ==
  (xs `merge` (ys `merge` zs))

[Success, Success, ..., Success]
```

`quickCheckPure` は乱数の種、生成するテストケースの数、テストする性質の3つの引数をとります。もしすべてのテストケースに成功したら、`Success` データ構築子の配列がコンソールに出力されます。

`quickCheckPure` は、性能ベンチマークの入力データ生成や、ウェブアプリケーションのフォームデータ例を無作為に生成するというような状況で便利かもしれません。

## 演習

1. (簡単) `Byte` と `Sorted` 型構築子についての `CoArbitrary` インスタンスを書いてください。
2. (やや難しい) 任意の関数 `f` について、`mergeWith f` 関数の結合性を主張する(高階)性質を書いてください。`quickCheckPure` を使って `psci` でその性質をテストしてください。
3. (やや難しい) 次のデータ型の `CoArbitrary` インスタンスを書いてください。

```
data OneTwoThree a = One a | Two a a | Three a a a
```

4. (やや難しい) `all` 関数を使って `quickCheckPure` 関数の結果を単純化してください。その関数はもしどんなテストもパスするなら `true` を返し、そうでなければ `false` を返さなくてはなりません。`purescript-monoids` で定義されている `First` モノイドを、失敗時の最初のエラーを保存するために `foldMap` 関数と一緒に使ってみてください。

## 13.11 まとめ

この章では、生成的テストのパラダイムを使って宣言的な方法でテストを書くため

の、`purescript-quickcheck` パッケージを導入しました。

- Grunt使ってQuickCheckをテストを自動化する方法を説明しました。
- エラーメッセージを改良する `<?>` 演算子の使い方と、性質を関数として書く方法を説明しました。
- `Arbitrary` と `CoArbitrary` 型クラスは、定型的なテストコードの自動生成を可能にし、高階性質関数を可能にすることも説明しました。
- 独自のデータ型に対して `Arbitrary` と `CoArbitrary` インスタンスを実装する方法を説明しました。

## 14 領域特化言語

---

### 14.1 この章の目標

---

この章では、多数の標準的な手法を使ったPureScriptにおける領域特化言語(domain-specific language, DSL) の実装について探求していきます。

領域特化言語とは、特定の問題領域での開発に適した言語のことです。領域特化言語の構文および機能は、その領域内の考え方を表現するコードの読みやすさを最大限に発揮すべく選択されます。本書の中では、すでに領域特化言語の例を幾つか見てきています。

- 第11章で開発された `Game` モナドと関連するアクションは、テキストアドベンチャーゲーム開発という領域に対しての領域特化言語を構成しています。
- 第12章で `ContT` と `Parallel` 関手のために書いたコンビネータのライブラリは、非同期プログラミングの領域に対する領域特化言語の例と考えることができます。
- 第13章で扱った `purescript-quickcheck` パッケージは、生成的テストの領域の領域特化言語です。このコンビネータはテストの性質に対して特に表現力の高い記法を可能にします。

この章では、領域特化言語の実装において、いくつかの標準的な手法による構造的なアプローチを取ります。これがこの話題の完全な説明だということでは決してありませんが、独自の目的に対する具体的なDSLを構築するには十分な知識を与えてくれるでしょう。

この章で実行している例は、HTML文書を作成するための領域特化言語になります。正しいHTML文書を記述するための型安全な言語を開発することが目的で、少しずつ実装を改善することによって作業していきます。

### 14.2 プロジェクトの準備

---

この章で使うプロジェクトには新しいBower依存性が追加されます。これから使う道具のひとつであるFreeモナドが定義されている `purescript-free` ライブラリです。

このプロジェクトのソースコードは、Gruntを使ってビルドすることができます。

## 14.3 HTMLデータ型

このHTMLライブラリの最も基本的なバージョンは `Data.DOM.Simple` モジュールで定義されています。このモジュールには次の型定義が含まれています。

```
newtype Element = Element
  { name      :: String
  , attribs   :: [Attribute]
  , content   :: Maybe [Content]
  }

data Content
  = TextContent String
  | ElementContent Element

newtype Attribute = Attribute
  { key      :: String
  , value    :: String
  }
```

`Element` 型はHTMLの要素を表しており、各要素は要素名、属性のペアの配列と、要素の内容で構成されています。`content` プロパティでは、`Maybe` タイプを使って要素が開いている(他の要素やテキストを含む)か閉じているかを示しています。

このライブラリの鍵となる機能は次の関数です。

```
render :: Element -> String
```

この関数はHTML要素をHTML文字列として出力します。`psci` で明示的に適当な型の値を構築し、ライブラリのこのバージョンを試してみましょう。

```
> :i Data.DOM.Simple
> :i Data.Maybe

> render $ Element
  { name: "p"
  , attribs: [
```



```

        Attribute
        { key: "class"
        , value: "main"
        }
    ]
    , content: Just [
        TextContent "Hello World!"
    ]
    }

"<p class=\"main\">Hello World!</p>"

```

現状のライブラリにはいくつかの問題があります。

- HTML文書の作成に手がかかります。すべての新しい要素が少なくとも1つのレコードと1つのデータ構築子が必要です。
- 無効な文書を表現できてしまいます。
  - 要素名の入力を間違えるかもしれません
  - 要素に間違った型の属性を関連付けることができてしまいます
  - 開いた要素が正しい場合でも、閉じた要素を使用することができてしまいます

この章では、さまざまな手法を用いてこれらの問題を解決し、このライブラリーをHTML文書を作成するために使える領域特化言語にしていきます。

## 14.4 スマート構築子

最初に導入する手法は方法は単純なものですが、とても効果的です。モジュールの使用にデータの表現を露出する代わりに、モジュールエクスポートリスト(module exports list)を使ってデータ構築子 `Element`、`Content`、`Attribute` を隠蔽し、正しいことが明らかなデータだけ構築する、いわゆるスマート構築子(smart constructors)だけをエクスポートします。

例を示しましょう。まず、HTML要素を作成するための便利な関数を提供します。

```

element :: String -> [Attribute] -> Maybe [Content] -> Element
element name attribs content = Element
    { name:      name
    , attribs:   attribs
    , content:   content
    }

```

次に、`element` 関数を適用することによってHTML要素を作成する、スマート構築子を作

成します。

```
a :: [Attribute] -> [Content] -> Element
a attrs content = element "a" attrs (Just content)

div :: [Attribute] -> [Content] -> Element
div attrs content = element "div" attrs (Just content)

p :: [Attribute] -> [Content] -> Element
p attrs content = element "p" attrs (Just content)

img :: [Attribute] -> Element
img attrs = element "img" attrs Nothing
```

最後に、正しいデータ構造だけを構築することがわかっているこれらの関数をエクスポートするように、モジュールエクスポートリストを更新します。

```
module Data.DOM.Smart
  ( Element()
  , Attribute(..)
  , Content(..)

  , a
  , div
  , p
  , img

  , render
  ) where
```

モジュールエクスポートリストはモジュール名の直後の括弧内に書きます。各モジュールのエクスポートは次の3種類のいずれかです。

- 値の名前で示された、値(または関数)
- クラスの名で示された、型クラス
- 型の名前で示された型構築子、およびそれに続けて括弧で囲まれた関連するデータ構築子のリスト

ここでは、`Element` の型をエクスポートしていますが、データ構築子はエクスポートしていません。もしデータ構築子をエクスポートすると、モジュールの使用者が不正なHTML要素を構築できてしまいます。

`Attribute` と `Content` 型についてはデータ構築子をすべてエクスポートしています(エクス

ポートリストの記号 `..` で示されています)。これから、これらの型にスマート構築子の手法を適用していきます。

すでにライブラリにいくつかの大きな改良を加わっていることに注意してください。

- 不正な名前を持つHTML要素を表現することは不可能です(もちろん、ライブラリが提供する要素名に制限されています)。
- 閉じた要素は、構築するときに内容を含めることはできません。

`Content` 型にもとても簡単にこの手法を適用することができます。単にエクスポートリストから `Content` 型のデータ構築子を取り除き、次のスマート構築子を提供します。

```
text :: String -> Content
text = TextContent

elem :: Element -> Content
elem = ElementContent
```

`Attribute` 型にも同じ手法を適用してみましょう。まず、属性のための汎用のスマート構築子を用意します。最初の試みとしては、次のようなものになるかもしれません。

```
(:=) :: String -> String -> Attribute
(:=) key value = Attribute
  { key: key
  , value: value
  }
```

この定義では元の `Element` 型と同じ問題に悩まされています。存在しなかったり、名前が間違っているような属性を表現することが可能です。この問題を解決するために、属性名を表すnewtypeを作成します。

```
newtype AttributeKey = AttributeKey String
```

それから、この演算子を次のように変更します。

```
(:=) :: AttributeKey -> String -> Attribute
(:=) (AttributeKey key) value = Attribute
  { key: key
  , value: value
  }
```

`AttributeKey` データ構築子をエクスポートしなければ、明示的にエクスポートされた次のような関数を使う以外に、使用者が型 `AttributeKey` の値を構築する方法はありません。いくつかの例を示します。

```
href :: AttributeKey
href = AttributeKey "href"

_class :: AttributeKey
_class = AttributeKey "class"

src :: AttributeKey
src = AttributeKey "src"

width :: AttributeKey
width = AttributeKey "width"

height :: AttributeKey
height = AttributeKey "height"
```

新しいモジュールの最終的なエクスポートリストは次のようになります。もうどんなデータ構築子も直接エクスポートしていないことに注意してください。

```
module Data.DOM.Smart
  ( Element()
  , Attribute()
  , Content()
  , AttributeKey()

  , a
  , div
  , p
  , img

  , href
  , _class
  , src
  , width
  , height

  , (:=)
  , text
  , elem

  , render
  ) where
```

`psci` でこの新しいモジュールを試してみると、コードが大幅に簡潔になり、改良されていることがわかります。

```
> :i Data.DOM.Smart
  > render $ p [ _class := "main" ] [ text "Hello World!" ]

"<p class=\"main\">Hello World!</p>"
```

しかし、基礎のデータ表現が変更されていないので、`render` 関数を変更する必要はなかったことにも注目してください。これはスマート構築子による手法の利点のひとつです。外部APIの利用者によって認識される表現から、モジュールの内部データ表現を分離することができます。

## 演習

1. (簡単) `Data.DOM.Smart` モジュールで `render` を使った新しいHTML文書の作成を試してみましょう。
2. (やや難しい) `checked` と `disabled` など、値を要求しないHTML属性がありますが、これらは次のような空の属性として表示されるかもしれません。

```
<input disabled>
```

空の属性を扱えるように `Attribute` の表現を変更してください。要素に空の属性を追加するために、`: (=)` の代わりに使える関数を記述してください。

## 14.5 幻影型

次に適用する手法についての動機を与えるために、次のコードを考えてみます。

```
> :i Data.DOM.Phantom
  > render $ img [ src    := "cat.jpg"
                  , width := "foo"
                  , height := "bar"
                  ]

"<img src=\"cat.jpg\" width=\"foo\" height=\"bar\" />"
```

ここでの問題は、`width` と `height` についての文字列値を提供しているということで、ここで

与えることができるのはピクセルやパーセントの単位の数値だけであるべきです。

`AttributeKey` 型にいわゆる幻影型(phantom type)引数を導入すると、この問題を解決できます。

```
newtype AttributeKey a = AttributeKey String
```

定義の右辺に対応する型 `a` の値が存在しないので、この型変数 `a` は幻影型と呼ばれています。この型 `a` はコンパイル時により多くの情報を提供するためだけに存在しています。任意の型 `AttributeKey a` の値は実行時には単なる文字列ですが、そのキーに関連付けられた値に期待されている型を教えてください。

`AttributeKey` の新しい形式で受け取るように、`(:=)` 演算子の型を次のように変更します。

```
(:=) :: forall a. (IsValue a) => AttributeKey a -> a -> Attribute
(:=) (AttributeKey key) value = Attribute
    { key: key
    , value: toValue value
    }
```

ここで、ファントム型引数 `a` は、属性キーと属性値が互換性のある型を持っていることを確認するために使われます。使用者は `AttributeKey a` を型の値を直接作成できないので(ライブラリで提供されている定数を介してのみ得ることができます)、すべての属性が正しくなります。

`IsValue` 制約は、キーに関連付けられた値がなんであれ、その値を文字列に変換し、生成したHTML内に出力できることを保証します。`IsValue` 型クラスは次のように定義されています。

```
class IsValue a where
    toValue :: a -> String
```

`String` と `Number` 型についての型クラスインスタンスも提供しておきます。

```
instance stringIsValue :: IsValue String where
    toValue = id

instance numberIsValue :: IsValue Number where
    toValue = show
```

また、これらの型が新しい型変数を反映するように、`AttributeKey` 定数を更新しなければいけません。

```
href :: AttributeKey String
href = AttributeKey "href"

_class :: AttributeKey String
_class = AttributeKey "class"

src :: AttributeKey String
src = AttributeKey "src"

width :: AttributeKey Number
width = AttributeKey "width"

height :: AttributeKey Number
height = AttributeKey "height"
```

これで、不正なHTML文書を表現することが不可能で、`width` と `height` 属性を表現するのに数を使うことが強制されていることがわかります。

```
> :i Data.DOM.Phantom
> render $ img [ src    := "cat.jpg"
                  , width := 100
                  , height := 200
                  ]

"<img src=\"cat.jpg\" width=\"100\" height=\"200\" />"
```

## 演習

1. (簡単) ピクセルまたはパーセントの長さのいずれかを表すデータ型を作成してください。その型について `IsValue` のインスタンスを書いてください。この型を使うように `width` と `height` 属性を変更してください。
2. (難しい) ファントム型を使って真偽値 `true`、`false` についての表現を最上位で定義することで、`AttributeKey` が `disabled` や `checked` のような空の属性を表現しているかどうかを符号化することができます。

```
data True
data False
```

ファントム型を使って、使用者が `(:=)` 演算子を空の属性に対して使うことを防ぐように、前の演習の解答を変更してください。

## 14.6 Freeモナド

APIに施す最後の変更は、`Content` 型をモナドにして`do`記法を使えるようにするために、Freeモナドと呼ばれる構造を使うことです。Freeモナドは、入れ子になった要素をわかりやすくなるよう、HTML文書の構造化を可能にします。次のようなコードを考えます。

```
p [ _class := "main" ]
  [ elem $ img
    [ src    := "cat.jpg"
    , width  := 100
    , height := 200
    ]
  , text "A cat"
  ]
```

これを次のように書くことができるようになります。

```
p [ _class := "main" ] $ do
  elem $ img
    [ src    := "cat.jpg"
    , width  := 100
    , height := 200
    ]
  text "A cat"
```

しかし、`do`記法だけがFreeモナドの恩恵だというわけではありません。モナドのアクションの表現をその解釈から分離し、同じアクションに複数の解釈を持たせることをFreeモナドは可能にします。

Free モナドは `purescript-free` ライブラリの `Control.Monad.Free` モジュールで定義されています。 `psci` を使うと、次のようにFreeモナドについての基本的な情報を見ることができます。

```
> :i Control.Monad.Free
> :k Free
(* -> *) -> * -> *
```



`Free` の種は、引数として型構築子を取り、別の型構築子を返すことを示しています。実は、`Free` モナドは任意の `Functor` を `Monad` にするために使うことができます！

モナドのアクションの表現を定義することから始めます。これを行うには、サポートする各モナドアクションそれぞれについて、ひとつのデータ構築子を持つ `Functor` を作成する必要があります。今回の場合、2つのモナドのアクションは `elem` と `text` になります。実際には、`Content` 型を次のように変更するだけです。

```
data ContentF a
    = TextContent String a
    | ElementContent Element a

instance functorContentF :: Functor ContentF where
    (<$>) f (TextContent s a) = TextContent s (f a)
    (<$>) f (ElementContent e a) = ElementContent e (f a)
```

ここで、この `ContentF` 型構築子は以前の `Content` データ型とよく似ています。 `Functor` インスタンスでは、単に各データ構築子で型 `a` の構成要素に関数 `f` を適用します。

これにより、最初の型引数として `ContentF` 型構築子を使うことで構築された、新しい `Content` 型構築子を `Free` モナドを包む newtype として定義することができます。

```
newtype Content a = Content (Free ContentF a)
```

ここで newtype を使っているのは、使用者に対してライブラリの内部表現を露出することを避けるためです。 `Content` データ構築子を隠すことで、提供しているモナドのアクションだけを使うことを仕様者に制限しています。

`ContentF` は `Functor` なので、`Free ContentF` に対する `Monad` インスタンスが自動的に手に入り、このインスタンスを `Content` 上の `Monad` インスタンスへと持ち上げることができます。

```
runContent :: forall a. Content a -> Free ContentF a
runContent (Content x) = x

instance functorContent :: Functor Content where
    (<$>) f (Content x) = Content (f <$> x)

instance applyContent :: Apply Content where
    (<*>) (Content f) (Content x) = Content (f <*> x)

instance applicativeContent :: Applicative Content where
```

```

pure = Content <<< pure

instance bindContent :: Bind Content where
  (>>=) (Content x) f = Content (x >>= (runContent <<< f))

instance monadContent :: Monad Content

```

`Content` の新しい型引数を考慮するように、少し `Element` データ型を変更する必要があります。モナドの計算の戻り値の型が `Unit` であることだけが要求されます。

```

newtype Element = Element
  { name      :: String
  , attribs   :: [Attribute]
  , content   :: Maybe (Content Unit)
  }

```

また、`Content` モナドについての新しいモナドのアクションになる `elem` と `text` 関数を変更する必要があります。これを行うには、`Control.Monad.Free` モジュールで提供されている `liftF` 関数を使います。この関数の(簡略化された)型は次のようになっています。

```

liftF :: forall f a. (Functor f) => f a -> Free f a

```

`liftF` は、何らかの型 `a` について、型 `f a` の値からFreeモナドのアクションを構築できるようにします。今回の場合、`ContentF` 型構築子のデータ構築子を次のようにそのまま使うだけです。

```

text :: String -> Content Unit
text s = Content $ liftF $ TextContent s unit

elem :: Element -> Content Unit
elem e = Content $ liftF $ ElementContent e unit

```

他にもコードの変更はありますが、興味深い変更は `render` 関数に対してのものです。ここでは、このFreeモナドを解釈しなければいけません。

## 14.7 モナドの解釈

`Control.Monad.Free` モジュールでは、Freeモナドで計算を解釈するための多数の関数が提供されています。

```

go :: forall f a. (Functor f) =>
  (f (Free f a) -> Free f a) ->
  Free f a ->
  a

goM :: forall f m a. (Functor f, Monad m) =>
  (f (Free f a) -> m (Free f a)) ->
  Free f a ->
  m a

iterM :: forall f m a. (Functor f, Monad m) =>
  (forall a. f (m a) -> m a) ->
  Free f a ->
  m a

```

純粋な結果を計算するためにFreeモナドを使いたいなら、`go` 関数が便利です。`goM` 関数と `iterM` 関数は、モナドを使用してFreeモナドのアクションを解釈することができます。この2つの関数は解釈関数の型が若干異なりますが、ここでは `iterM` 関数を使います。興味のある読者は、代わりに `goM` 関数を使用してこのコードを再実装してみるといいでしょう。

まず、アクションを解釈することができるモナドを選ばなければなりません。`Writer String` モナドを使って、結果のHTML文字列を累積することになります。

新しい `render` メソッドは補助関数 `renderElement` に移譲して開始し、`Writer` モナドで計算を実行するため `execWriter` を使用します。

```

render :: Element -> String
render e = execWriter $ renderElement e

```

`renderElement` はwhereブロックで定義されています。

```

where
  renderElement :: Element -> Writer String Unit
  renderElement (Element e) = do

```

`renderElement` の定義は簡単で、いくつかの小さな文字列を累積するために `Writer` モナドの `tell` アクションを使っています。

```

tell "<"
  tell e.name
  for_ e.attrs $ \a -> do
    tell " "

```

```
renderAttribute a
renderContent e.content
```

次に、同じように簡単な `renderAttribute` 関数を定義します。

```
where
  renderAttribute :: Attribute -> Writer String Unit
  renderAttribute (Attribute a) = do
    tell a.key
    tell "=\\"
    tell a.value
    tell "\\\""
```

`renderContent` 関数は、もっと興味深いものです。ここでは、`iterM` 関数を使って、Freeモナドの内部で補助関数 `renderContentItem` に移譲する計算を解釈しています。

```
renderContent :: Maybe (Content Unit) -> Writer String Unit
renderContent Nothing = tell " />"
renderContent (Just (Content content)) = do
  tell ">"
  iterM renderContentItem content
  tell "</"
  tell e.name
  tell ">"
```

`renderContentItem` の型は `iterM` の型シグネチャから推測することができます。関数 `f` は型構築子 `ContentF` で、モナド `m` は解釈している計算のモナド、つまり `Writer String` です。これにより `renderContentItem` について次の型シグネチャがわかります。

```
renderContentItem :: forall a. ContentF (Writer String a) -> Writer String a
```

`ContentF` の二つのデータ構築子でパターン照合するだけで、この関数を実装することができます。

```
renderContentItem (TextContent s rest) = do
  tell s
  rest
renderContentItem (ElementContent e rest) = do
  renderElement e
  rest
```

それぞれの場合において、式 `rest` は型 `Writer String` を持っており、解釈計算の残りを表しています。`rest` アクションを呼び出すことによって、それぞれの場合を完了することができます。

これで完了です！ `psci` で、次のように新しいモナドのAPIを試してみましょう。

```
> :i Data.DOM.Free
> render $ p [] $ do
  elem $ img [ src := "cat.jpg" ]
  text "A cat"

"<p><img src=\"cat.jpg\" />A cat</p>"
```

## 演習

1. (やや難しい) `ContentF` 型に新しいデータ構築子を追加して、生成されたHTMLにコメントを出力する新しいアクション `comment` に対応してください。`liftF` を使ってこの新しいアクションを実装してください。新しい構築子を適切に解釈するように、解釈 `renderContentItem` を更新してください。
2. (難しい) `goM` と `iterM` 関数の問題のひとつに、スタック安全でないというものがあります。大きいモナドのアクションは、解釈したときにスタックオーバーフローを引き起こす可能性があるのです。しかしながら、`Control.Monad.Free` ライブラリは、スタック安全な `go` と `goEff` 関数を提供しています。`Writer` モナドの代わりに `ST` 作用を利用して、`goEff` 関数を使って `Content` モナドを解釈してください。

## 14.8 言語の拡張

すべてのアクションが型 `Unit` の何かを返すようなモナドは、さほど興味深いものではありません。実際のところ、概ね良くなったと思われる構文は別として、このモナドは `Monoid` 以上の機能は何の追加していません。

意味のある結果を返す新しいモナドアクションでこの言語を拡張することで、Freeモナド構造の威力を説明しましょう。

アンカーを使用して文書のさまざまな節へのハイパーリンクが含まれているHTML文書を作成するとします。手作業でアンカーの名前を生成すればいいので、これは既に実現できています。文書中で少なくとも2回、ひとつはアンカーの定義自身に、もうひとつはハイパーリンクに、アンカーが含まれています。しかし、この方法には根本的な問題がいくつかありま

す。

- 開発者は一意なアンカー名を生成するために失敗することがあります。
- 開発者は、アンカー名のひとつまたは複数のインスタンスを誤って入力するかもしれません。

自分の間違いから開発者を保護するために、アンカー名を表す新しい型を導入し、新しい一意な名前を生成するためのモナドアクションを提供することができます。

最初の手順は、名前の型を新しく追加することです。

```
newtype Name = Name String

runName :: Name -> String
runName (Name n) = n
```

繰り返しになりますが、`Name` は `String` の newtype として定義しており、モジュールのエクスポートリスト内でデータ構築子をエクスポートしないように注意する必要があります。

次に、属性値として `Name` を使うことができるように、新しい型 `IsValue` 型クラスのインスタンスを定義します。

```
instance nameIsValue :: IsValue Name where
  toValue (Name n) = n
```

また、次のように `a` 要素に現れるハイパーリンクの新しいデータ型を定義します。

```
data Href
  = URLHref String
  | AnchorHref Name

instance hrefIsValue :: IsValue Href where
  toValue (URLHref url) = url
  toValue (AnchorHref (Name nm)) = "#" ++ nm
```

`href` 属性の型の値を変更して、この新しい `Href` 型の使用を強制します。また、要素をアンカーに変換するのに使う新しい `name` 属性を作成します。

```
href :: AttributeKey Href
href = AttributeKey "href"

name :: AttributeKey Name
```

```
name = AttributeKey "name"
```

残りの問題は、現在モジュールの使用者が新しい名前を生成する方法がないということです。`Content` モナドでこの機能を提供することができます。まず、`ContentF` 型構築子に新しいデータ構築子を追加する必要があります。

```
data ContentF a
  = TextContent String a
  | ElementContent Element a
  | NewName (Name -> a)
```

`NewName` データ構築子は型 `Name` の値を返すアクションに対応しています。データ構築子の引数として `Name` を要求するのではなく、型 `Name -> a` の関数を提供するように使用者に要求していることに注意してください。型 `a` は計算の残りを表していることを思い出すと、この関数は、型 `Name` の値が返されたあとで、計算を継続する方法を提供するというように直感的に理解することができます。

新しいデータ構築子を考慮するように、`ContentF` についての `Functor` インスタンスを更新する必要があります。

```
instance functorContentF :: Functor ContentF where
  (<$>) f (TextContent s a) = TextContent s (f a)
  (<$>) f (ElementContent e a) = ElementContent e (f a)
  (<$>) f (NewName k) = NewName (f <<< k)
```

そして、先ほど述べたように、`liftF` 関数を使うと新しいアクションを構築することができます。

```
newName :: Content Name
newName = Content $ liftF $ NewName id
```

`id` 関数を継続として提供していることに注意してください。型 `Name` の結果を変更せずに返すということを意味しています。

最後に、新しいアクションを解釈するために、解釈関数を更新する必要があります。以前は計算を解釈するために `Writer String` モナドを使っていましたが、このモナドは新しい名前を生成する能力を持っていないので、何か他のものに切り替えなければなりません。`RWS` モナドなら、`Writer` の機能を提供するだけでなく、純粋な状態を扱うことができます。型シグネチャを短く保てるように、この解釈モナドを型同義語としての定義しておきます。

```
type Interp = RWS Unit String Number
```

`RWS` モナドは3つの型引数を取ることを思い出してください。最初は大域的な設定で、今回は単なる `Unit` です。2つめは「ログ」型で、累積するHTML文字列です。最後の引数は状態の型で、この場合は増加していくカウンタとして振る舞う数で、一意な名前を生成するのに使われます。

`Writer` と `RWS` モナドはそれらのアクションを抽象化するのに同じ型クラスメンバを使うので、どのアクションも変更する必要がありません。必要なのは、`Writer String` への参照すべてを `Interp` で置き換えることだけです。しかし、この計算を実行するために使われるハンドラを変更しなければいけません。`execWriter` の代わりに、`evalRWS` を使います。

```
render :: Element -> String
render e = snd $ evalRWS (renderElement e) unit 0
```

`snd` の呼び出しは `evalRWS` から返された `Tuple` の2番目の要素だけを返すようにします。この場合は、累積されたHTML文字列を表しています。

新しい `NewName` データ構築子を解釈するために、`renderContentItem` に新しい場合分けを追加しなければいけません。

```
renderContentItem (NewName k) = do
  n <- get
  let name = Name $ "name" ++ show n
  put $ n + 1
  k name
```

ここで、型 `Name -> Interp a` の継続 `k` が与えられているので、型 `Interp a` の解釈を構築しなければいけません。この解釈は単純です。`get` を使って状態を読み、その状態を使って一意な名前を生成し、それから `put` で状態をインクリメントしています。最後に、継続にこの新しい名前を渡して、計算を完了します。

これにより、`psci` で、`Content` モナドの内部で一意な名前を生成し、要素の名前とハイパーリンクのリンク先の両方を使って、この新しい機能を試してみましょう。

```
> :i Data.DOM.Name
> render $ p [ ] $ do
  top <- newName
  elem $ a [ name := top ] $
    text "Top"
```



```
elem $ a [ href := AnchorHref top1 ] $
  text "Back to top"
```

```
"<p><a name=\"name0\">Top</a><a href=\"#name0\">Back to top</a></p>"
```

複数回の `newName` 呼び出しの結果が、実際に一意な名前になっていることを確かめてみてください。

## 演習

1. (やや難しい) 使用者から `Element` 型を隠蔽すると、さらにAPIを簡素化することができます。次の手順に従って、これらの変更を行ってください。
  - `p` や `img` のような(返り値が `Element` の)関数を `elem` アクションと結合して、型 `Content Unit` を返す新しいアクションを作ってください。
  - 型 `Content a` の引数を許容し、結果の型 `Tuple String` を返すように、`render` 関数を変更してください。
2. (難しい) 次の新しいアクションをサポートするように、`ContentF` タイプを変更してください。

```
isMobile :: Content Boolean
```

このアクションは、この文書がモバイルデバイス上での表示のためにレンダリングされているかどうかを示す真偽値を返します。

ヒント: `RWS` モナドの `ask` アクションと `Reader` コンポーネントを使って、このアクションを解釈してください。

## 14.9 まとめ

この章では、いくつかの標準的な技術を使って、単純な実装を段階的に改善することにより、HTML文書を作成するための領域特化言語を開発しました。

- データ表現の詳細を隠蔽し、構築方法により正しい文書を作ることだけを許可するために、スマート構築子を使用しました。
- 言語の構文を改善するために、ユーザ定義の中置2項演算子を使用しました。
- 使用者が間違った型の属性値を提供するのを防ぐために、データの型に追加の情報を符号化する幻影型を使用しました。
- `Free` モナドを使って、内容の集まりの配列内包表記を `do` 表記を提供するモナド表現に

変換しました。モナドの新しいアクションをサポートするためにこの表現を拡張し、`Writer` と `RWS` モナドでモナド計算を解釈しました。

使用者が間違いを犯すのを防ぎ、領域特化言語の構文を改良するために、これらの手法はすべてPureScriptのモジュールと型システムを活用しています。

関数型プログラミング言語による領域特化言語の実装は活発に研究されている分野ですが、いくつかの簡単なテクニックに対して役に立つ導入を提供し、表現力豊かな型を持つ言語で作業すること威力を示すことができているのであれば幸いです。

[目次](#)