Haskellコーディング規約

藤永　直

Table of Contents

[1 用語の説明 2](#_Toc48557415)

[2 基本方針 6](#_Toc48557416)

[3 警告の一覧 7](#_Toc48557417)

[3.1 -Wallで有効になる警告の一覧 7](#_Toc48557418)

[3.2 -Wall外で推奨する警告の一覧 12](#_Toc48557419)

[4 ライブラリ 14](#_Toc48557420)

[5 言語拡張 14](#_Toc48557421)

[6 コメントのルール 15](#_Toc48557422)

[7 例外処理 15](#_Toc48557423)

[8 リソースの管理 16](#_Toc48557424)

[9 遅延評価によるスペース・リーク対策 16](#_Toc48557425)

[10 その他、一般的な推奨事項 16](#_Toc48557426)

[11 参考ガイド 21](#_Toc48557427)

このコーディング規約は日本信号においてソフトウェア開発においてHaskell言 語を使用する際に適用されます。

このドキュメントは、約５年間のHaskellによる商用のアプリケーション開発で 得られた知見をまとめたものです。

# 用語の説明

型シグニチャ

Haskellでは、スコープ内の関数や変数の型をその関数や変数の定義とは別に 宣言することができます。その型の宣言を型シグニチャと呼びます。e.g.

addInt :: Int -> Int -> Int -- 型シグニチャ  
addInt x y = x + y -- 関数の定義

副作用

関数を呼び出した時、関数の返り値以外で発生する全ての作用を副作用と言います。 e.g. 変数の書換え、ファイルの読み書き、ネットワークやマウス・キーボード などの入出力など。

純粋な関数

純粋な関数とは、副作用を伴わない関数のことです。Haskellでは純粋な関数と副作用 を伴う関数が型によって区別されます。

myPureFunction :: Int -> Int -- Intを受け取りIntを返す、純粋な関数  
myIOAction :: Int -> IO Int -- Intを受け取りIntを返す、副作用を伴う関数

GHC

本書が対象とするHaskellのデファクトスタンダードのコンパイラ。

Haskell2010

2010年に定められた最新のHaskellの言語仕様。

言語拡張

GHCでは、Haskell2010から逸脱する言語機能を言語拡張として提供します。各言 語拡張はプラグマとしてコンパイラに指示することで、有効／無効にできます。

ポイント・フリー・スタイル

関数を定義する際に、合成関数の演算子などを用いて、引数を明示的に書かな いコーディング・スタイル。

警告(Warning)

プログラムをコンパイル時に、エラーにはならないが、一般に非推奨とされて いる書き方などをコンパイラが見つけた場合は、コンパイラは「警告」としてユ ーザーにメッセージを表示します。

部分関数

全ての定義域の値に対して、返り値が定義されていない関数。

フォーマッタ

静的解析ツールの一つで、ソースコードを自動的にコーディングスタイルに合 致するように整形するツール。

Linter

静的解析ツールの一つで、ソースコードの意味を変えない範囲で冗長なコード の書き換えを提案するツール。

型クラス

Haskell言語において、関数の多重定義を行うためのしくみ。e.g. Int、 Float、DoubleなどをNumという型クラスのインスタンスにすることにより、(+)、 (-)などの演算子をInt、Float、Doubleなどの型によらず使用することができます。

自動導出(deriving)

Haskellでは、ユーザー定義の型に対して、自動的に特定の型クラスのメソッ ドを導出することができます。

モジュール

Haskellのソースコードはモジュール単位でファイル分割され、モジュール単 位で関数やデータ型をエクスポート・インポートすることができます。

ライブラリ

機能や目的の単位でデータ型や関数をひとまとめにしたもの。コードの再利用 を目的に一連のデータ型や関数がライブラリとして開発者に提供されます。 Haskellではライブラリは、複数のモジュールからなります。

パッケージ

ライブラリや実行体のソースコードをひとまとめにしたもの。Haskellのソー スコードはパッケージ単位でバージョン管理され、HackageやStackageなどのパ ッケージデータベースに登録されてWeb上に公開されています。

非同期例外

他のスレッドやOSなど、自スレッド外から自スレッドに向けて発行される例外。 例外処理されない場合、基本的に該当スレッドはKillされます。

同期例外

自スレッドで発行され、自スレッドにより処理される例外。処理されない場合、 基本的に該当スレッドはKillされます。

Foreign Function Interface(FFI)

HaskellではFFIという仕組みによって、C言語などの他の言語で書かれたプロ グラムを呼び出す、または、他の言語からHaskellの関数を呼び出すことができ ます。

Haddock

Haskellソースコード上に特定の形式でコメントを記載することで、自動的に HTMLのAPIリファレンスを生成するためのツール。

正格・非正格な評価

正格な評価とは、関数の引数が常にその関数に引き渡される前に完全に評価 （aka 簡約）されることを意味します。一方、非正格な評価では、式の値が必要 になるまで式の評価を遅らせます。Haskellはデフォルトで非正格な評価を行う 言語です。

型推論

型推論によってコンパイラは、プログラム中の変数や関数の型に矛盾が無いこ とを確認し、また、型が明示されていない変数や関数の型を推論して、型が一意 に定まるかを確認します。矛盾がある、または、関数の型が一意に定まらない場 合はコンパイルエラーとなります。。

ホール

プログラム中の変数や関数の型がわからなくなった場合、該当の型もしくは変 数の箇所にアンダースコア（aka ホール）を記載することで、コンパイル時にコ ンパイラが型推論に基づいて、該当箇所に当てはまる型の候補を列挙してくれま す。変数にホールを指定した場合、その変数が評価されると例外が発生します。 e.g. 以下のコードをコンパイルするとコンパイラがホールに該当する型はInt だと教えてくれます。

addedNumber :: \_  
addedNumber = myAdd 1 2

シャドーイング

特定のスコープ内のみで有効なローカル変数の名称が、大域変数の名前と衝突 した場合、そのスコープ内ではローカル変数の定義で大域変数の定義が上書きさ れます。これをシャドーイングと呼びます。

BANG

Haskellでは変数やデータ型の識別子にエクスクラメーション記号（aka Bang） を付加することで、個別に評価戦略を正格評価に切り替えることができます。以 下の例では、xにBangを指定しているので、addIntStrict x yを評価する前 にxの評価が強制されます。

addIntStrict :: Int -> Int -> Int  
addIntStrict !x y = x + y

# 基本方針

HaskellのコンパイラであるGHCでは、オプション(-W..)を使ってコンパイル中 の「致命的ではないエラー」（i.e. 警告）に対するコンパイラの振る舞いを指 定できます。

一般にC言語などがコーディング規約によりカバーする内容は、Haskellにおいて は、コンパイラのエラーおよび警告という形で全て網羅されています。

従って、本コーディング規約では、Section 3.1に示す全ての警 告をプログラミング時に取り除くことを要請します。

コンパイラに-Wall、-Werrorのフラグを指定してコンパイルすることで、疑わし いコードであることを示しうるすべての警告オプションを有効にして、かつ、全 ての警告をエラーとして扱うことができます。

基本的に、プロダクションコードは-Wall、-Werrorのフラグを指定にしてコンパ イルすべきです。そうすることで、出荷される製品のコードが本章で指定する コーディング規約に合致していることが自動的に保証されます。

下記にGHCの警告の一覧を示します。

# 警告の一覧

## -Wallで有効になる警告の一覧

* -Wtyped-holes
  + 変数や関数の定義中にホール(\_)が使用されている場合に警告します。
* -Wpartial-type-signatures
  + 型シグネチャ中にホール(\_)が使用されている場合に警告します。
* -Wdeprecations
  + WARNING あるいはDEPRECATEDプラグマの付いたモジュール、関数、型を使ったときに警告します。
* -Wnoncanonical-monad-instances
  + MonadあるいはMonadFailのインスタンス宣言が標準的ではない場合に警告します。
* -Wmissing-monadfail-instances
  + do ブロックで失敗する可能性のあるパターンが用いられていてMonadFailのインスタンスではないときに警告します。
* -Wdeprecated-flags
  + 非推奨になっているコンパイラ・オプションを使用している場合に警告します。
* -Wdodgy-exports
  + 型シノニムにすぎないデータ型Tがすべての構成子をT(..)のようにエクスポートしているときに警告します。 また、何もエクスポートしていないモジュールを再エクスポートしているときも警告します。
* -Wdodgy-imports
  + 以下の場合に警告します。
    - データ型TをT(..)のようにすべての構成子付きでインポートしているのに、その型がTのように 抽象的にしかエクスポートされていない場合。
    - import 宣言がエクスポートされていない実体を隠蔽している場合。
* -Woverflowed-literals
  + リテラルがオーバーフローを起こすような場合、たとえば300 :: Word8のような場合に警告します。
* -Wempty-enumerations
  + たとえば[5 .. 3]のように列挙が空になる場合に警告します。
* -Wduplicate-constraints
  + 型シグネチャで冗長な制約がある場合にコンパイラが警告します。
* -Wduplicate-exports
  + エクスポートリストに重複したエントリがある場合にコンパイラが警告します。
* -Whi-shadowing
  + 現在のディレクトリにあるモジュールやインターフェイスファイルが、ライブラリや別のディレクトリにある 同名のモジュールをシャドーイングしてしまっている場合、コンパイラが警告を発行するようになります。
* -Wincomplete-patterns
  + パターン照合が実行時に失敗する可能性がある場合に警告します。
* -Wmissing-fields
  + ラベル付きフィールド構成子を構成するときに、1つ以上のフィールドについて初期化子が欠如している場合に警告します。
* -Wmissing-methods
  + インスタンス宣言が1つ以上のメソッドを欠き、そのメソッドのデフォルト定義が対応するクラス宣言にないときに警告します。
* -Wmissing-signatures
  + どのトップレベルの関数や値にも型シグネチャがあることを GHC に確認させたければ、-Wmissing-signaturesオプションを使うのがよいでしょう。
* -Wmissing-pattern-synonym-signatures
  + GHC はあらゆる多相的な局所束縛について警告を発行するようになります。
* -Wname-shadowing
  + このオプションを有効にすると、内側のスコープの値と同じ名前の値が外側のスコープにあるとき、 すなわち、内側の名前が外側の名前を隠す(シャドウする)ときに警告を発行する。
* -Worphans
  + モジュールに「孤立した」インスタンス宣言あるいは書き換え規則があるときに警告します。
  + インスタンス宣言が孤立しているとは、インスタンス宣言があるモジュールに、対応するクラスも具体化された型も定義されていないということです。
* -Woverlapping-patterns
  + コンパイラはパターンが重複しているときに警告します。
* -Wsimplifiable-class-constraints
  + トップレベルのインスタンス宣言を使用して簡略化できる型シグネチャのクラス制約について警告します
* -Wtabs
  + コンパイラはソースファイルにタブ文字が含まれていると警告をだします。
* -Wtype-defaults
  + 数値型に対するデフォルト化機構が発動したときに警告/通知します。
* -Wmissed-extra-shared-lib
  + GHCiがターゲットパッケージのextra-librariesの項に記載された共有ライブラリをロードできない場合に警告します。
* -Wunused-binds
  + 未使用の関数定義(および局所的な束縛)をすべて報告します
* -Wunused-imports
  + 明示的にインポートされながら、使われていないモジュールをすべて報告します。
* -Wunused-matches
  + 項レベルパターン照合で発生する未使用変数をすべて報告します。この警告は変数がアンダースコアで始まる変数の場合、すなわち以下のような場合には抑制されます。
* -Wunused-do-bind
  + doブロックで情報を黙って捨てているように見える式を報告します。 たとえばdo { mapM popInt xs ; return 10 }の場合、doブロックの最初の文があやしいと報告されます。
* -Wunused-foralls
  + ユーザが明示的に書いたforall文にある未使用型変数をすべて報告する。
* -Wunused-record-wildcards
  + レコード・ワイルドカードで束縛した変数が一つも使用されていない場合に警告する。
* -Wredundant-record-wildcards
  + レコード・ワイルドカードで一つも変数が束縛されていない場合に警告する。
* -Wwrong-do-bind
  + 次に示すコードのようにアクションの結果が束縛されていない場合に報告します: do { return (popInt 10) ; return 10 }
  + この場合、最初のアクションの型はStackM (StackM Int)にも関わらず、束縛をしていないため、中身が取り出され無いままになっています。
  + 大抵の場合、このような場合はコーディング・ミスであり、実際には次のコードを意図していると思われます: do { popInt 10 ; return 10 }
* -Wunused-packages
  + -package pkgや-package-id unit-idでコマンドラインより指定されたパッケージがコンパイル時にロードされていない場合に警告します。
* -Wderiving-defaults
  + DeriveAnyClassとGeneralizedNewtypeDerivingが有効な場合に明示的に導出戦略(i.e.、stock、newtype、anyclass)が指定されていない場合に警告します。
* -Wunsupported-llvm-version
  + コンパイラに-fllvmフラグが指定されたときに、サポートされていないバージョンのLLVMをターゲットとした場合に警告します。
* -Wdeferred-type-errors
  + 型チェックを実行時まで行わないようにコンパイラに指示されている場合に警告します。
* -Wunrecognised-warning-flags
  + コンパイラが認識できない(-W..)フラグに遭遇したときの警告を有効にします。

以下の警告も、-Wallで有効になりますが、本コーディング規約を守る限り出く わすことはありませんので、ここでは説明を省略します。

* 型システム拡張関連
  + -Winaccessible-code
  + -Wstar-binder
  + -Wunused-foralls
* FFI関連
  + -Wunsupported-calling-conventions
  + -Wdodgy-foreign-imports
* プラグマ・正格性関連
  + -Winline-rule-shadowing
  + -Wunrecognised-pragmas
  + -Wunbanged-strict-patterns

## -Wall外で推奨する警告の一覧

* -Wincomplete-uni-patterns
  + -Wincomplete-patterns と似ていますが、ラムダ式とパターンバインディングに適用されます。(例えば、h = \[] -> 2やlet Just k = f y)
* -Widentities
  + Preludeの数値変換がT型の値を同じT型の値に変換しようとしている場合に、コンパイラが警告を発行するようになります。
  + このような呼び出しは何もしないのと同じで省略できます。 検査される関数は toInteger、toRational、fromIntegral、realToFrac です。
* -Wredundant-constraints
  + 型シグネチャで冗長な制約がある場合にコンパイラが警告します。 具体的には、
    - 型シグネチャ内に冗長な制約がある場合。e.g. f :: (Eq a、Ord a) => a -> a
    - 型シグネチャ内の制約がカバーする範囲のコードで、その制約が使われていない場合。e.g. f :: Eq a => a -> a -> Bool; f x y = True
* -Wmissing-export-lists
  + 明示的にエクスポート・リストが宣言されていない場合に警告します。
* -Wmissing-import-lists
  + 明示的にインポート・リストを記載せず、かつ、修飾されていないインポート宣言を使用した場合に警告します。
* -Wpartial-fields
  + 次に示すコードのように列挙型のデータ型の一部にレコードラベルを定義した場合、部分関数を生成するため、警告します。e.g. data Foo = Foo { x :: Int } | Bar
* -Wincomplete-record-updates
  + 次に示す関数 f :: Foo -> FooはBarに適用すると失敗します。-Wincomplete-record-updatesを有効にしておくと、この場合に警告が発行されます。e.g. f foo = foo { x = 6 }
* -Wcompat
  + 将来のGHCのリリースでデフォルトで有効になる予定ですが、当面、デフォルトで無効となっている以下の警告を有効にします。これにより、コードを将来的に互換性のあるものにしたいと考えているライブラリの作者は、警告を生成する前に新しい機能に適応することができます。
    - -Wsemigroup
      * 将来、標準の型クラスにSemigroupが含まれるようになったときに、衝突を引き起す定義に対して警告を出します。
    - -Wnoncanonical-monoid-instances
      * SemigroupあるいはMonoidのインスタンス宣言が標準的でないときに警告します。
    - -Wcompat-unqualified-imports
      * 将来のGHCリリースで変更される可能性のあるコアライブラリモジュール(Data.List)が修飾無しで一括インポートされている場合に警告します。

# ライブラリ

外部のパッケージを使用する際は、なるべくStackageに登録されているパッケー ジを使用すること。Haskellのスタンダード・モジュールであるPreludeは危険な 部分関数を含んでいますので、必要に応じてrioなどのPreludeの代替ライブラリ を使用することを推奨します。

* [Stackage](https://www.stackage.org/)
* [rio](https://hackage.haskell.org/package/rio)

# 言語拡張

パッケージ内の全てのソースコードに適用する言語拡張を.cabalファイルの default-extensionsにて指定することができます。下記の言語拡張をデフォル トで有効にすることを推奨します。

* StrictData
* TupleSections
* LambdaCase
* MultiWayIf
* RecordWildCards
* DerivingStrategies
* DeriveGeneric
* DeriveAnyClass
* GeneralizedNewtypeDeriving
* BinaryLiterals

その他の言語拡張は必要に応じて、ソースコードファイルの先頭行に明示的にリ ストアップしてください。

※GHCで使用可能な言語拡張の一覧と各言語拡張の詳細に関しては、本書では説 明しないので、下記のガイドを参照してください：

* [GHC言語拡張の一覧](https://limperg.de/ghc-extensions/)

# コメントのルール

なるべく[Haddockコメント形式](https://haskell.e-bigmoon.com/stack/doc/haddock-comment.html)でコメン トを記載してください。また、なるべくhaddockコマンドによりAPIリファレンスをソー スコードより自動生成するようにしてください。

stackを使用している場合はstack haddockのコマンドでAPIリファレンスを自 動生成できます。

# 例外処理

非同期例外を自らthrowするプログラムを書かないようにしてください(i.e.  throwToを使わない)。

同期例外であっても、（main以外で）トップレベルで定義するIO関数はその関数 の外部に例外を漏らさないようにしてください。該当IO関数のプログラム内部で 例外を使用するのは構いませんが、必ずそのプログラム内で例外をcatchするよ うにしてください。

下記のようなExceptT IOのパターンは使わずに、単純にIOの例外を使う、もしく はIO Eitherを使うようにしてください。

-- - Bad  
myFailableAction :: String -> ExceptT MyException IO Int  
  
-- + Good  
myFailableAction :: String -> IO Int  
  
-- + Also Good  
myFailableAction :: String -> IO (Either MyException Int)

※純粋な関数の場合はExceptTを使うことに何の問題もありません。

# リソースの管理

リソースの開放を忘れないようになるべく、openFile、close等のリソースを単 独で取得・開放する関数の使用を避けてください。

withFile、withSocket等のwithXXX系の関数もしくは、bracketのようにリソース の取得と開放を強制する関数を使用するように心がけてください。

# 遅延評価によるスペース・リーク対策

* default-extensionsにてStrictDataを指定するようにしてください。
* 変更可能な変数(i.e.、IORef、MVar)を使用する場合は、正格な版(Data.IORef.Strict、Control.Concurrent.MVar.Strict)を使用してください。

# その他、一般的な推奨事項

その他、一般的な推奨事項を下記に列挙します：

* Haskellのソースコードファイル(.hs)はUTF8のエンコーディングを使用すること。
* IOは極力使わない
  + 副作用を必要としない計算にはIOを用いないでください。
  + 純粋な関数を定義する際に、書き換え可能な変数を使ったほうが見通しが良くなる場合も極稀にありますが、そのような場合も、IOモナドではなく、STモナドなどを使うようにしてください。
* 二項演算子はなるべく自分で定義しない
  + Haskellでは、二項演算子をユーザーが独自に定義できますが、一般的な二項演算子はすでに標準ライブラリやStackageに登録されたパッケージなどでカバーされています。独自に定義した二項演算子は、関数名から機能を類推できないので、プログラムの理解を困難します。
* 型クラスはなるべく自分で定義しない
  + アプリケーション作成において、一般的に必要になる型クラスは、標準ライブラリやStackageに登録されたパッケージなどですでにカバーされているので、自前の型クラスが必要になることはまずありません。自分で定義する前に探してください。
* 型システムを拡張する類の言語拡張はなるべく使用しない
  + 型システム関連の言語拡張は、ライブラリの開発者にとっては有効な場合もありますが、アプリケーションの開発者には不要です。
  + 複雑な型システムは、コードの読者の理解を困難にするので、使用するライブラリの要請で必要となる場合を除いて、なるべく使用しないでください。
* 言語拡張以外のプラグマは使用しない
  + INLINEやSPECILIZE、UNPACKなどのプラグマは、パフォーマンスの最適化には有効ですが、本書が想定するアプリケーションにおいては、そこまでのパフォーマンスのチューニングが 必要になることはまず無いので、使用しないでください。
* マジックハッシュ(e.g. Int#)、UNPACKプラグマ、BANGなどは使用しない
  + これらはパフォーマンスの最適化には有効ですが、本書が想定するアプリケーションにおいては、そこまでのパフォーマンスのチューニングが 必要になることはまず無いので、使用しないでください。
  + データ型のコンストラクタのフィールドはデフォルトで正格とすべきなので、コンパイラのdefault-extensionsにStrictDataの言語拡張を指定するようにして、個別にBANGを使用することは避けてください。
* なるべくForeign Function Interface(FFI)を使わない
  + HaskellではFFIという仕組みによって、C言語などの他の言語で書かれたプログラムを呼び出す、または、他の言語からHaskellの関数を呼び出すことができますが、 外部の言語で書かれたプログラムに対しては、Haskellによる型システム等による保証が及ばないので、なるべく避けるべきです。
* なるべくForeign Function Interfaceを使ったライブラリの使用は避ける
  + 同様の理由でFFIを使ったHaskellのライブラリはHaskellの型システムによる品質の保証がされないので、なるべく避けるべきです。
* 型クラスのインスタンス宣言はなるべく自分で書かずderivingを使って自動導出する
  + アプリケーション作成においてユーザー定義のデータ型を既存の型クラスのインスタンスにする際、多くの場合はderivingを使ってインスタンス宣言を自動導出できます。
* 部分関数は定義しない(-Wincomplete-patterns)
  + 純粋な関数を定義する場合は、引数の型が取りうる全ての入力に対して、返り値を定義するように心がけてください。
  + -Wincomplete-patternsを使用することで、コンパイラが全ての場合分けについて、返り値が定義されているか確認してくれます。
  + 部分関数を生成する下記のようなデータ型は定義しないでください。(-Wpartial-fields、-Wincomplete-record-updatesでこのような定義に対して警告を有効にできます)

data Foo = Foo { x :: Int }  
 | Bar  
f :: Foo -> Foo  
f foo = foo { x = 6 }

* 部分関数を使わないこと
  + Haskellの標準ライブラリ(Prelude)にも部分関数が含まれていますが、これらの関数は危険なので使用しないでください。
    - [Preludeの部分関数の一覧](https://wiki.haskell.org/List_of_partial_functions)
  + 必要に応じて[rio](https://hackage.haskell.org/package/rio)などのPreludeの代替ライブラリの使用を推奨します。
* unsafeの接頭辞がついている関数はなるべく使わない
  + 型システムによる安全性を破壊するような特殊な関数の名前には慣習的にunsafeの接頭辞がつけられています(e.g. unsafePerformIO :: IO a -> a)。 そのような関数は使用しないでください。
* newtypeには-XDerivingStrategies拡張機能を使用して、型クラスの導出方法を明示的に指定してください。e.g.

{-# LANGUAGE DeriveAnyClass #-}  
{-# LANGUAGE DerivingStrategies #-}  
{-# LANGUAGE GeneralizedNewtypeDeriving #-}  
  
newtype Id a = Id { unId :: Int }  
 deriving stock (Generic)  
 deriving newtype (Eq、Ord、Show、Hashable)  
 deriving anyclass (FromJSON、ToJSON)

* 型宣言はなるべく書く(-Wmissing-signatures)
* ポイント・フリースタイルの乱用は避けること。e.g.

-- + Good  
foo :: Int -> a -> Int  
foo n x = length $ replicate n x  
  
-- - Bad  
foo :: Int -> a -> Int  
foo = (length . ) . replicate

* 無名関数の乱用は避ける
  + 純粋関数を定義する場合に、無名関数は乱用すると読みにくいので、なるべく明示的に関数を定義してから使用するようにしてください。e.g.

-- - Bad  
makeAsocList :: [Key] -> [(Key、Value)]  
makeAsocList keys = mapMaybe (\ k -> liftM (k ,) (if p k then Just (f k) else Nothing)) keys  
 where p :: Key -> Bool  
 p = ...  
 f :: Key -> Value  
 f = ...  
  
-- + Good  
makeAsocList :: [Key] -> [(Key、Value)]  
makeAsocList keys = mapMaybe keyToPair keys  
 where keyToPair :: Key -> Maybe (Key、Value)  
 keyToPair k  
 | p k = Just (k、f k)  
 | otherwise = Nothing  
 p :: Key -> Bool  
 p = ...  
 f :: Key -> Value  
 f = ...  
  
-- Acceptable  
print1To10 :: IO ()  
print1To10 = forM\_ [1..10] $ \ n -> do  
 putStrLn "Print Number"  
 putStrLn $ show n

* 適切な型を使用して、関数の引数や変数が「ありえない値」をそもそも取り得ないようにしてください。

例：headは空でないリストを受け取って、先頭の要素を返す関数である。

例１

-- Return the first value wrapped in a `Just` if present, `Nothing` otherwise  
head :: [a] -> a

例２

-- Return the first value wrapped in a `Just` if present, `Nothing` otherwise  
head :: [a] -> Maybe a

例３

-- Return the first value of a list, which never fails if the list is `NonEmpty`  
head :: NonEmpty a -> a

* 例１では空のリストを渡すとエラーとなる部分関数であり、危険です。
* 例２では、空のリストを渡すとNothingを返し、エラーをあげることは無いので、1.よりは改善されています。
* 例３では、そもそも引数の型をNonEmpty a（非空のリスト）にすることで空のリストを引数として受け付け無いので、 これらの例の中では最も安全です。（ただし、入力に空のリストがありえる場合など例２のように書かざるを得ない場合もあります。）

# 参考ガイド

* [GHCユーザーガイド](https://ghcguide.haskell.jp/8.2.2/users_guide/using-warnings.html)
* [Haddock User Guide](https://www.haskell.org/haddock/doc/html/index.html)
* [An opinionated guide to Haskell in 2018](https://lexi-lambda.github.io/blog/2018/02/10/an-opinionated-guide-to-haskell-in-2018/)
* [A Guide to GHC’s Extensions](https://limperg.de/ghc-extensions/)
* [The golden rule of software quality](http://www.haskellforall.com/2020/07/the-golden-rule-of-software-quality.html)
* [EXCEPTIONS BEST PRACTICES IN HASKELL](https://www.fpcomplete.com/blog/2016/11/exceptions-best-practices-haskell/)