Haskell 虎の巻

この虎の巻は、プログラミング言語 Haskell の基本要素を見やすく整えたものです: 基本要素には、構文やキーワード、その他の要素があります。虎の巻は、実行可能な Haskell ファイルと印刷可能なドキュメントの両方の形で提供されます。ソースをお気に入りのインタープリタにロードして、ここで示したコード例と遊んでください。

(訳者コメント: 本訳は著者の Bailey さんから許可を得て公開するものです。)

基本構文

コメント

一行コメントは '`--` ' で始まり、行の終わりまで続きます。複数行コメントは'`{-`' で始まり、'`-}`' まで続きます。コメントは入れ子にできます。

Haskell コードを文書化するためのシステム Haddock との互換性のためには、函数定義の上のコメントは'`{- |`' で始めるべきであり、パラメータタイプの隣のコメントは '`-- ^`' で始めるべきです。

予約語

Haskell では、以下の単語が予約されています。これらの名前を変数や函数に与えると構文エラーになります。

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

文字列

- `"abc"` – ユニコード文字列、`['a','b','c']` の糖衣。
- `'a'` – 単一文字。

複数行文字列　通常、もし文字列が実際の改行文字を含んでいたら、それは構文エラーです。すなわち、以下は構文エラーです:

```
string1 = "My long
string."
```

バックスラッシュ('\') は改行を「エスケープ」できます:

```
string1 = "My long \
\string."
```

バックスラッシュ間の領域は無視されます。文字列の**中の**改行は明示的に表されなければいけません:

```
string2 = "My long \n\
\string."
```

つまり、`string1` は以下のように評価されます:

```
My long string.
```

一方、`string2` は以下のように評価されます:

```
My long
string.
```

エスケープコード　以下のエスケープコードが文字や文字列の中で使うことができます:

- `\n`, `\r`, `\f`, etc. – 改行、キャリッジリターン、フォームフィードなどの標準コードがサポートされています。
- `\72`, `\x48`, `\o110` – それぞれ 10 進数、16 進数、8 進数で値 72 を持つ文字。
- `\&` – "null" エスケープ文字。数値エスケープコードが数リテラルの隣に現れることができるように使われます。例えば、`\x2C4` は (ユニコードで)∧で、一方、`\x2C\&4` は `,4` です。この順序は文字リテラルでは使うことはできません。

数字

- `1` – 整数値または浮動小数点数値。
- `1.0`, `1e10` – 浮動小数点数値。
- `0o1`, `0O1` – 8 進数値。
- `0x1`, `0X1` – 16 進数値。
- `-1` – 負の数; 負号 ("-") は数字から離すことはできません。

列挙

- `[1..10]` – 数– 1, 2, ..., 10 のリスト。
- `[100..]` – 数– 100, 101, 102, ... の無限リスト。

ichikawa.yuji@gmail.com

- `[110..100]` – 空のリスト。しかし、`[110, 109 .. 100]` は 110 から 100 までのリストを与えます。
- `[0, -1 ..]` – 負の整数。
- `[-110..-100]` – 構文エラー; 負号のためには `[-110.. -100]` である必要があります。
- `[1,3..99]`, `[-1,3..99]` – 1 から 99 までの 2 毎の、-1 から 99 までの 4 毎のリスト。

実際には、Enum クラスの任意の値を使うことができます:

- `['a' .. 'z']` – 文字– a, b, ..., z のリスト。
- `['z', 'y' .. 'a']` – z, y, x, ..., a。
- `[1.0, 1.5 .. 2]` – `[1.0,1.5,2.0]`。
- `[UppercaseLetter ..]` – (Data.Char からの)GeneralCategory 値のリスト。

リスト & タプル

- `[]` – 空のリスト。
- `[1,2,3]` – 3 つの数のリスト。
- `1 : 2 : 3 : []` – "cons" (:) と "nil" (`[]`) を使ってリストを書く代わりの方法。
- `"abc"` – 3 つの文字のリスト (文字列はリストです)。
- `'a' : 'b' : 'c' : []` – (`"abc"` と同じ) 文字のリスト。
- `(1,"a")` – 数と文字列の 2-要素タプル。
- `(head, tail, 3, 'a')` – 2 つの函数と数、文字の 4-要素タプル。

「レイアウト」ルールと中括弧、セミコロン

Haskell は、ちょうど C のように、中括弧とセミコロンを使って書くことができます。しかしながら、誰もそうはしません。代わりに、「レイアウト」ルールを使います。そのルールではスペースがスコープを表します。一般的なルールは次の通りです: いつもインデントしなさい。コンパイラが文句を言う時、更にインデントしなさい。

中括弧とセミコロン セミコロンは式を終了し、中括弧はスコープを表します。次に示すいくつかのキーワードの後でそれらを使うことができます: where, let, do, of。函数本体を定義する時にはそれらを使うことはできません。例えば、以下はコンパイルされません。

```
square2 x = { x * x; }
```

しかしながら、以下はうまくいきます:

```
square2 x = result
    where { result = x * x; }
```

函数定義 `where` 節が現れないなら、函数名から少なくとも 1 スペース分本体をインデントしなさい:

```
square x  =
  x * x
```

`where` 節が現れる場合、函数名から少なくとも 1 スペース分 where 節をインデントし、`where` キーワードから少なくとも 1 スペース分函数本体をインデントしなさい:

```
square x =
    x2
  where x2 =
    x * x
```

Let `let` の中の最初の定義から少なくとも 1 スペース分、let の本体をインデントしなさい。もし `let` が自身の行上に現れるなら、いかなる定義の本体も let の後の列に現れなければいけません:

```
square x =
  let x2 =
        x * x
  in x2
```

上で見られるように、`in` キーワードも let と同じ列になければいけません。最後に、多重定義が与えられた時、すべての識別子は同じ列に現れなければいけません。

宣言など

以下の節は、函数宣言やリスト内包表現、言語の他の領域に関するルールを詳しく述べます。(訳者コメント: 函数型言語らしさを出す気持ちで、argument を引数と訳さずに独立変数と訳してみました。プログラマの方には気持ち悪いかもしれません。評判悪ければ直します。しばらくおつきあいください。)

函数定義

函数は、その名前とすべての独立変数、等号を宣言することで定義されます:

```
square x = x * x
```

すべての函数名は小文字か "_" で始めなければいけません。そうでなければ、構文エラーになります。

パターンマッチング　函数の複数「条項」が、独立変数の値の「パターンマッチング」で定義されることがあります。以下、agree 函数は 4 つの別個の場合を持ちます:

```
-- Matches when the string "y" is given.
agree1 "y" = "Great!"
-- Matches when the string "n" is given.
agree1 "n" = "Too bad."
-- Matches when string beginning
agree1 ('y':_) = "YAHOO!"
-- Matches for any other value given.
agree1 _ = "SO SAD."
```

'_' 文字はワイルドカードであり、任意の値とマッチすることに注意してください。

パターンマッチングは入れ子の値に拡張することができます。以下のデータ宣言を仮定して:

```
data Bar = Bil (Maybe Int) | Baz
```

8ページからMaybe の定義を思い出すと、Bil が存在している時、入れ子の Maybe の値に関してマッチできます:

```
f (Bil (Just _)) = ...
f (Bil Nothing) = ...
f Baz = ...
```

パターンマッチングは、値を変数に割り当てることも許します。例えば、以下の函数は、与えられた文字列が空かそうでないか決定します。もし空でないなら、str にバインドされた値が小文字に変換されます:

```
toLowerStr [] = []
toLowerStr str = map toLower str
```

上の str は、なんでもマッチする _ に似ていることに注意してください; 唯一の違いは、マッチした値も名前を与えられることです。

$n+k$ パターン　この (時々異論のある) パターンマッチング機能は、ある種の数値表現にマッチすることを容易にします。アイデアは、マッチングのための定数を伴う基本ケース ("n" 部分) を定義し、基本ケースへの付加として他のマッチ ("k" 部分) を定義することです。以下は、数字が偶数か否かをテストするかなり非効率な方法です:

```
isEven 0 = True
isEven 1 = False
isEven (n + 2) = isEven n
```

独立変数捕捉　余分な変数を宣言することなしに、値にパターンマッチし、**かつ**、それを使うには、独立変数捕捉が役に立ちます。マッチするパターンと値をバインドする変数の間に '@' シンボルを使ってください。以下では、長さを計算するためにリスト全体を ls にバインドする一方、表示のためにリストの先頭を l にバインドするのにこの機能が使われます:

```
len ls@(l:_) = "List starts with " ++
  show l ++ " and is " ++
  show (length ls) ++ " items long."
len [] = "List is empty!"
```

ガード　パターンマッチングと一緒に、函数定義の「ガード」としてブーリアン函数を使うことができます。以下、パターンマッチングを伴わない例:

```
which n
  | n == 0 = "zero!"
  | even n = "even!"
  | otherwise = "odd!"
```

otherwise に注意してください。 – それはいつも True に評価され、「デフォルトの」分岐を指定するのに使うことができます。

ガードはパターンと一緒に使うことができます。以下は、文字列の中の最初の文字が大文字か小文字か決定する函数です:

```
what [] = "empty string!"
what (c:_)
  | isUpper c = "upper case!"
  | isLower c = "lower case"
  | otherwise = "not a letter!"
```

マッチング & ガードの順序　パターンマッチングは上から下の順に処理されます。同様に、ガード式は上から下にテストされます。例えば、以下の函数のどちらもほとんど益のないものです。

```
allEmpty _ = False
allEmpty [] = True

alwaysEven n
  | otherwise = False
  | n `div` 2 == 0 = True
```

レコード構文　通常、マッチされる値の中の独立変数の位置に基づいてパターンマッチングが起こります。しかしながら、レコード構文で宣言されたタイプは、それら

のレコード名に基づいてマッチすることができます。以下のデータタイプが与えられたとして:

```
data Color = C { red
  , green
  , blue :: Int }
```

以下のように、green に関してだけマッチできます:

```
isGreenZero (C { green = 0 }) = True
isGreenZero _ = False
```

不細工になるけれども、独立変数捕捉はこの構文と一緒に可能です。さて、上に続いて、Pixel タイプと、ゼロでない green 成分を持つ値をすべて黒で置き換える函数を定義します:

```
data Pixel = P Color

-- Color value untouched if green is 0
setGreen (P col@(C { green = 0 })) = P col
setGreen _ = P (C 0 0 0)
```

遅延パターン　反駁できないパターンとしても知られているこの構文は、常に成功するパターンマッチを許します。それは、パターンを使う任意の条項が成功することを意味しますが、もし実際にマッチした値を使おうとするなら、エラーが起こるかもしれません。一般に、たとえ値が存在しなくても特定の値の**型**に関して動作する必要がある時、これが役に立ちます。

例えば、デフォルト値に関してクラスを定義する:

```
class Def a where
  defValue :: a -> a
```

アイデアは、defValue に右の型の値を与え、その型のデフォルト値を戻すところです。基本的な型に関するインスタンスを定義することは簡単です:

```
instance Def Bool where
  defValue _ = False


instance Def Char where
  defValue _ = ' '
```

型に関するデフォルト値を得たいが、コンストラクタは Nothing かもしれないので、Maybe はさらに少し手の込んだものです。以下の定義は機能しますが、Nothing が渡される時 Nothing を得るので、最も望ましいものではありません。

```
instance Def a => Def (Maybe a) where
  defValue (Just x) = Just (defValue x)
  defValue Nothing = Nothing
```

それよりむしろ Just (*default value*) を戻してもらいたい。以下は遅延パターンが私たちを救う例です – たとえ Nothing が与えられても、Just x をマッチしたように装うことができ、それを使ってデフォルト値を得ます:

```
instance Def a => Def (Maybe a) where
  defValue ~(Just x) = Just (defValue x)
```

値 x が実際に評価されない限り、安全です。ベース型の何も x を見る必要はありません (see the "_" matches they use)。だからことはちょうどうまく機能します。

上にともなう 1 つの難点は、Nothing コンストラクタを使う時インタープリタやコードに型注釈を供給しなければいけないことです。Nothing は型 Maybe a を持ちますが、もし他の十分な情報が利用可能でないなら、a が何か Haskell に教えなければいけません。いつかの例のデフォルト値:

```
-- Return "Just False"
defMB = defValue (Nothing :: Maybe Bool)
-- Return "Just ' '"
defMC = defValue (Nothing :: Maybe Char)
```

リスト内包

リスト内包は以下の 4 つの型の要素から成ります: *generators*, *guards*, *local bindings*, *targets*。リスト内包は与えられた生成子とガードに基づくターゲット値のリストを生成します。以下の内包はすべての平方を生成します:

```
squares = [x * x | x <- [1..]]
```

x <- [1..] は Integer 値すべてを生成し、x にそれらを 1 つずつ格納します。x * x は、それ自身に x を掛けることでリストのそれぞれの要素を生成します。

ガードはある要素を除外することを可能にします。以下は与えられた数の (それ自身を除く) 約数が如何に計算できるかを示します。d がガードとターゲット式の両方で如何に使われるかに注意してください。

```
divisors n =
  [d | d <- [1..(n `div` 2)]
     , n `mod` d == 0]
```

ローカルバインドは、生成される式や続く生成子とガードの中で使うために新しい定義を提供します。以下では、z は a と b のうち小さい方を表すのに使われています:

```
strange = [(a,z) | a <-[1..3]
                 , b <-[1..3]
```

```
                , c <- [1..3]
                , let z = min a b
                , z < c ]
```

内包は数に限りません。任意のリストが可能です。すべての大文字が生成できます:

```
ups =
  [c | c <- [minBound .. maxBound]
     , isUpper c]
```

また、(0からインデックスされる) リスト word の中で特殊なブレーク値 br のすべての出現を見つけるために:

```
idxs word br =
  [i | (i, c) <- zip [0..] word
     , c == br]
```

リスト内包独特の特徴は、パターンマッチングの失敗がエラーを起こさないことです; それらは結果のリストから除外されるだけです。

演算子

Haskell にはあらかじめ定義された「演算子」がほとんどありません– あらかじめ定義されたように現れるほとんどは実際には構文 (例えば、"=") です。むしろ、演算子は単に、2 つの独立変数を取り、特別な構文サポートを持つ函数です。いわゆる演算子はいずれも括弧を使った前置函数として適用できます:

```
3 + 4 == (+) 3 4
```

新しい演算子を定義するには、演算子が 2 つの独立変数の間に現れることを除いて、単にそれを通常の函数として定義してください。以下は、2 つの文字列の間にコンマを挿入し、余分なスペースが現れないことを保証するものです:

```
first ## last =
  let trim s = dropWhile isSpace
         (reverse (dropWhile isSpace
           (reverse s)))
  in trim last ++ ", " ++ trim first

> "  Haskell " ## " Curry "
Curry, Haskell
```

もちろん、完全なパターンマッチング、ガードなどがこの形式で利用可能です。型シグニチャは若干異なるけれども。演算子の「名前」は括弧の中に現れなければいけません:

```
(##) :: String -> String -> String
```

演算子を定義するのに使うことができるシンボルは以下の通りです:

```
# $ % & * + . / < = > ? @ \ ^ | - ~
```

しかしながら、再定義できないいくつかの「演算子」があります。それらは以下のものです: <-, ->, =。最後の =はそれ自身では再定義できませんが、複数文字演算子の一部として使うことはできます。"bind" 函数>>=は 1 つの例です。

優先順位 & 結合性　両方合わせて**固定性**と呼ばれる任意の演算子の優先順位と結合性は、infix、infixr、infixl キーワードを通じて設定することができます。これらはトップレベルの函数とローカル定義の両方に適用することができます。構文は以下の通りです:

{infix | infixr | infixl} *precedence op*

ここで *precedence* は 0 から 9 まで変わります。*Op* は実際には 2 つの独立変数を取る任意の函数 (すなわち任意の 2 変数演算) であり得ます。演算子が左結合か右結合かどうかは、それぞれ、infixl か infixr によって指定されます。そんな infix 宣言は結合性を持ちません。

優先順位と結合性は算術計算の規則の多くを「期待通りに」します。例えば、以下の足し算と掛け算の優先順位のマイナーな更新を考えてください:

```
infixl 8 `plus1`
plus1 a b = a + b
infixl 7 `mult1`
mult1 a b = a * b
```

結果は驚くべきものです:

```
> 2 + 3 * 5
17
> 2 `plus1` 3 `mult1` 5
25
```

結合性を逆にすることも興味ある結果を持ちます。割り算の右結合として再定義:

```
infixr 7 `div1`
div1 a b = a / b
```

興味ある結果を得ます:

```
> 20 / 2 / 2
5.0
```

```
> 20 `div1` 2 `div1` 2
20.0
```

カリー化

Haskell では、函数は一度に独立変数すべてを得る必要はありません。例えば、convertOnly 函数を考えてください。それは、文字列の中の test によって決まる特定の要素だけを変換します:

```
convertOnly test change str =
    map (\c -> if test c
               then change c
               else c) str
```

convertOnly を使って、特定の文字を数に変換する l33t 函数を書くことができます:

```
l33t = convertOnly isL33t toL33t
  where
    isL33t 'o' = True
    isL33t 'a' = True
    -- etc.
    isL33t _ = False
    toL33t 'o' = '0'
    toL33t 'a' = '4'
    -- etc.
    toL33t c = c
```

l33t は指定された独立変数を持たないことに注意してください。また、convertOnly の最後の独立変数は与えられません。しかしながら、l33t の型シグニチャは全体の構想を告げます:

```
l33t :: String -> String
```

すなわち、l33t は文字列を取り、文字列を生成します。それは、文字列を取り文字列を生成する函数という値をいつも返すという意味で、「一定不変のもの」です。l33t は、3 つの独立変数の 2 つだけが供給された「カリー化された」形式の convertOnly を返します。

これはさらに続けることができます。大文字だけを変える函数を書きたいとしましょう。適用するテスト isUpper を知っていますが、変換を指定したくありません。その函数は以下のように書くことができます:

```
convertUpper = convertOnly isUpper
```

これは以下の型シグニチャを持ちます:

```
convertUpper :: (Char -> Char)
  -> String -> String
```

すなわち、convertUpper は 2 つの独立変数を取ることができます。一番目は個々の文字を変換する変換函数で、二番目は変換される文字列です。

複数の独立変数を取る任意の函数のカリー化された形式が生成できます。これを考える 1 つの方法は、函数のシグニチャの中のそれぞれの「矢」がもう 1 つの独立変数を供給することで生成できる新しい函数を表すということです。

セクション　演算子は函数であり、それらは他のもの同様カリー化できます。例えば、"+" のカリー化バージョンは以下のように書くことができます:

```
add10 = (+) 10
```

しかしながら、これは扱いにくく読みにくいかもしれません。「セクション」は、括弧を使ってカリー化された演算子です。以下はセクションを使った add10 です:

```
add10 = (10 +)
```

供給される独立変数は右か左に置くことができます。置き場所は、独立変数がどの位置を取るべきか示しています。これは結合のような演算子には重要です:

```
onLeft str = (++ str)
onRight str = (str ++)
```

上の 2 つは全く違った結果を生成します:

```
> onLeft "foo" "bar"
"barfoo"
> onRight "foo" "bar"
"foobar"
```

値の「更新」とレコード構文

Haskell は純粋言語であり、それ自体値の変わる状態を持ちません。すなわち、一旦値が設定されると、決して変わりません。「更新」は実際には「変化した」フィールドの中の新しい値を持つコピー演算です。例えば、以前定義された Color タイプを使って、green フィールドをゼロに設定する函数を書くことが容易にできます:

```
noGreen1 (C r _ b) = C r 0 b
```

上は少し冗長であり、レコード構文を使って書き直すことができます。この種の「更新」は指定されたフィールドの値を設定するだけであり、残りをコピーします:

```
noGreen2 c = c { green = 0 }
```

ここで、c に関する Color 値を捕捉し、新しい Color 値を返します。その値はたまたま c として red や blue に関して同じ値を持ち、その green 成分は 0 です。green と blue フィールドを red フィールドに等しくするよう設定するためにこれをパターンマッチングと組み合わせることができます:

```
makeGrey c@(C { red = r }) =
  c { green = r, blue = r }
```

Color 値を得るために独立変数の捕捉 ("c@") を、内部 red フィールドを得るためにレコード構文 ("C { red = r}") とのパターンマッチングを使わなければいけないことに注意してください。

匿名函数

匿名函数 (すなわち、 **ラムダ式**、または単に**ラムダ**) は、名前を持たない函数です。それらは以下のようにいつでも定義できます:

```
\c -> (c, c)
```

これは独立変数を 1 つ取り、両方の位置にその独立変数を含むタプルを返す函数を定義します。それらは名前を必要としない簡単な函数には役に立ちます。以下は文字列が大文字小文字と空白文字だけからなるか明らかにします。

```
mixedCase str =
  all (\c -> isSpace c ||
             isLower c ||
             isUpper c) str
```

もちろん、ラムダは函数の戻り値になることもできます。以下の典型は独立変数を元々与えられたものに掛ける函数を返します:

```
multBy n = \m -> n * m
```

例えば:

```
> let mult10 = multBy 10
> mult10 10
100
```

型シグニチャ

Haskell は最高の型推論をサポートします。それはほとんどの場合に型を書き下す必要がないことを意味します。それでも型シグニチャは少なくとも 2 つの理由で役に立ちます。

ドキュメンテーション—たとえコンパイラがあなたの函数の型をわかったとしても、他のプログラマやあなた自身でさえ後にはりかいできないかもしれません。すべてのトップレベル函数に関する型シグニチャはとてもよい形式と考えられます。

特殊化—型クラスはオーバーロードを伴う函数を許します。例えば、数の任意のリストを負にする函数は以下のシグニチャを持ちます:

```
negateAll :: Num a => [a] -> [a]
```

しかしながら、効率や他の理由で Int 型を許すことだけを欲っするかもしれません。型シグニチャでそれを達成します:

```
negateAll :: [Int] -> [Int]
```

型シグニチャはトップレベル函数と入れ子の let または where 定義で現れることができます。いくつかの場合には、多相を妨げるために必要とされますが、一般にこれはドキュメンテーションのために役立ちます。型シグニチャは最初に型付けされるアイテムの名前があり、:: が続き、型が続きます。この例は既に上で見ています。

型シグニチャは実装の上に直接現れる必要はありません。それらは、含んでいるモジュールの中のどこでも (そう、後でも) 指定することができます。同じシグニチャを持つ複数のアイテムも一緒に定義することができます:

```
pos, neg :: Int -> Int

...

pos x | x < 0 = negate x
      | otherwise = x

neg y | y > 0 = negate y
      | otherwise = y
```

型注釈　時々、Haskell は型が何を意味するか明らかにすることができません。これの典型的な実物例はいわゆる "show . read" 問題です:

```
canParseInt x = show (read x)
```

read x の型を知らないので、Haskell はこの函数をコンパイルできません。注釈を通して型を限定しなければいけません:

```
canParseInt x = show (read x :: Int)
```

注釈は型シグニチャと同じ構文を持ちますが、任意の式で装飾されます。上の注釈は 変数 x 上ではなく式 read x 上であることに注意してください。函数適用 (例えば、read x) だけが注釈より強くバインドされます。もしそれがその場合に当てはまらないなら、上は (read x) :: Int と書く必要があるでしょう。

単位

() – 「単位」型と「単位」値。役立つ情報を表さない値と型。

キーワード

Haskell キーワードが以下でアルファベット順にリストされます。

Case

case は、C#や Java の switch 文と似ていますが、パターンをマッチできます: 検査される値の形。簡単なデータ型を考えてください:

```
data Choices = First String | Second |
  Third | Fourth
```

case はどの選択が与えられたか明らかにするために使うことができます:

```
whichChoice ch =
  case ch of
    First _ -> "1st!"
    Second -> "2nd!"
    _ -> "Something else."
```

函数定義でのパターンマッチングと同様、'_' トークンは任意の値にマッチする「ワイルドカード」です。

入れ子 & 捕捉　入れ子になったマッチングとバインドも許されます。例えば、以下は Maybe 型の定義です:

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

Maybe を使うと、入れ子になったマッチを使って任意の選択が与えられたか明らかにできます:

```
anyChoice1 ch =
  case ch of
    Nothing -> "No choice!"
    Just (First _) -> "First!"
    Just Second -> "Second!"
    _ -> "Something else."
```

バインドはマッチした値を操作するのに使うことができます:

```
anyChoice2 ch =
  case ch of
    Nothing -> "No choice!"
    Just score@(First "gold") ->
      "First with gold!"
    Just score@(First _) ->
      "First with something else: "
        ++ show score
    _ -> "Not first."
```

マッチング順序　マッチングは上から下に順に続きます。もし anyChoice1 が以下のように再整理されるなら、最初のパターンはいつも成功します:

```
anyChoice3 ch =
  case ch of
    _ -> "Something else."
    Nothing -> "No choice!"
    Just (First _) -> "First!"
    Just Second -> "Second!"
```

ガード　ガード、すなわち、条件付きマッチはちょうど函数定義のような場合に使うことができます。唯一の違いは=の代わりに->を使うことです。以下は大文字小文字を区別しない文字列マッチを実行する簡単な函数です:

```
strcmp s1 s2 = case (s1, s2) of
  ([], []) -> True
  (s1:ss1, s2:ss2)
    | toUpper s1 == toUpper s2 ->
        strcmp ss1 ss2
    | otherwise -> False
  _ -> False
```

Class

Haskell 函数は特定の型や一群の型に対して機能するように定義されます。複数回定義することはできません。ほとんどの言語は、函数が独立変数の型に依存して異なる振る舞いを持つことができる「オーバーロード」の概念をサポートします。Haskell は、クラスとインスタンス宣言を通してオーバーロードを達成します。クラスは、そのクラスのメンバー (すなわち、インスタンス) である任意の型に適用できる 1 つ以上の函数を定義します。クラスは Java や C#におけるインタフェースに似ていて、インスタンスはインタフェースの具体的な実装です。

クラスは1つ以上の型変数を使って宣言されなければいけません。技術的には、Haskell 98は1型変数だけを許しますが、Haskellのほとんどの実装は、複数型変数を許す、いわゆる**多重パラメータ型クラス**をサポートします。

与えられた型に関してフレーバを供給するクラスを定義できます:

```
class Flavor a where
  flavor :: a -> String
```

宣言は函数の型シグニチャだけを与えることに注意してください– ここでは実装は与えられません (いくつかの例外があります。9ページの「デフォルト」を参照してください。) 続けて、いくつかのインスタンスを定義できます:

```
instance Flavor Bool where
  flavor _ = "sweet"

instance Flavor Char where
  flavor _ = "sour"
```

`flavor True`の評価は以下をあたえます:

```
> flavor True
"sweet"
```

一方、`flavor 'x'`は以下を与えます:

```
> flavor 'x'
"sour"
```

デフォルト　デフォルト実装はクラスの中の函数に関して与えることができます。これらは、特定の函数がクラスの他を使って定義できる時、役に立ちます。デフォルトは、本体をメンバー函数の1つに与えることで定義されます。標準例は`Eq`です。それは`/=`(不等号) を`==`を使って定義します:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  (/=) a b = not (a == b)
```

再帰定義が生成できます。`Eq`例を続けると、`==`は`/=`を使って定義できます:

```
  (==) a b = not (a /= b)
```

しかしながら、もしインスタンスがメンバー函数の具体的な実装を十分供給しないなら、それらのインスタンスを使ったいかなるプログラムもループします。

Data

いわゆる**代数データ型**は以下のように宣言できます:

```
data MyType = MyValue1 | MyValue2
```

`MyType`は型の**名前**です。`MyValue1`と`MyValue`は型の**値**であり、いわゆる**コンストラクタ**です。多重コンストラクタは'|'文字で分離されます。型とコンストラクタ名は大文字で始まらなければいけないことに注意してください。そうでなければ構文エラーとなります。

独立変数付きコンストラクタ　上の型は列挙として以外にはあまり面白くありません。更に情報を保持すること許すよう、独立変数を取るコンストラクタが定義できます:

```
data Point = TwoD Int Int
  | ThreeD Int Int Int
```

それぞれのコンストラクタの独立変数は**型**名であって、コンストラクタでないことに注意してください。それは以下の種類の宣言が許容されていないことを意味します:

```
data Poly = Triangle TwoD TwoD TwoD
```

代わりに、`Point`型を使わなければいけません:

```
data Poly = Triangle Point Point Point
```

型とコンストラクタの名前　型とコンストラクタの名前は、混乱をもたらす場所で使われることは決してないので、同じにできます。例えば:

```
data User = User String | Admin String
```

これは、2つのコンストラクタ`User`と`Admin`を持つ、`User`と名付けられた型を宣言します。函数の中でこの型を使うことで違いがはっきりします:

```
whatUser (User _) = "normal user."
whatUser (Admin _) = "admin user."
```

ある文献はこの実践を**型もじり**と呼びます。

型変数　いわゆる**多相**データ型の宣言は、宣言の中に型変数を区割ることと同じぐらい簡単です:

```
data Slot1 a = Slot1 a | Empty1
```

これは2つのコンストラクタ Slot1 と Empty1 を持つ型 Slot1 を宣言します。Slot1 コンストラクタは、上で型変数 a で表される**任意の**型の独立変数を取ることができます。

コンストラクタで型変数と特定の型を混ぜることができます:

```
data Slot2 a = Slot2 a Int | Empty2
```

以上、Slot2 コンストラクタは任意の型の値と Int 値を取ることができます。

レコード構文　コンストラクタの独立変数は、上のように位置的にも、名前をそれぞれの独立変数に与えるレコード構文を使っても宣言できます。例えば、以下では、適切な独立変数に名前を持つ Contact 型を宣言します:

```
data Contact = Contact { ctName :: String
       , ctEmail :: String
       , ctPhone :: String }
```

これらの名前は**セレクタ**または**アクセサ**函数と呼ばれ、函数となります。それらは小文字かアンダースコアで始まらなければいけません。そして、同じスコープで別の函数として同じ名前を持つことはできません。例えば、上ではそれぞれ“ct”前置。(同じ型の) 多重コンストラクタは、同じ型の値に関して同じアクセサ函数を使うことができますが、もしアクセサがコンストラクタすべてで使われないならそれは危険です。幾分不自然な例でこれを考えてください:

```
data Con = Con { conValue :: String }
  | Uncon { conValue :: String }
  | Noncon

whichCon con = "convalue is " ++
  conValue con
```

もし whichCon が Noncon 値でコールされるなら、実行時エラーが起こります。

最後に、他で説明されたように、これらの名前は、パターンマッチングや独立変数捕捉、「更新」で使うことができます。

派生　多くの型は、文字列へ、文字列からの変換、等式や列の順序のための比較の機能のように、定義するのは退屈ですが必要とされる共通の演算を持ちます。Haskell では、これらの機能は型クラスとして定義されます。

これらの演算の7つは非常によくあるので、Haskell は関連した型上に型クラスを自動的に実装する deriving キーワードを提供します。サポートされる 7 つの型クラスは以下の通りです: Eq, Read, Show, Ord, Enum, Ix, Bounded。

2つの形式の deriving が可能です。1番目は型が1つのクラスだけに派生する時使われます:

```
data Priority = Low | Medium | High
  deriving Show
```

二番目は複数のクラスが派生される時使われます:

```
data Alarm = Soft | Loud | Deafening
  deriving (Read, Show)
```

上で与えられた7つを除いて他のいかなるクラスに関して deriving を指定することは構文エラーです。

クラス制約　データ型は型変数に関するクラス制約で定義することができますが、この実践は奨励されません。モジュールシステムを使って「生の」データコンストラクタを隠し、代わりに適切な制約を適用する「賢い」コンストラクタをエクスポートすることがより良いです。どんな場合でも、使われる構文は以下の通りです:

```
data (Num a) => SomeNumber a = Two a a
  | Three a a a
```

これは、型変数独立変数を 1 つ持つ型 SomeNumber を宣言します。有効な型は Num クラスの中のものです。

Deriving

10ページの data キーワード下のderivingに関する節を参照してください。

Do

do キーワードは続くコードが**モナドな**文脈の中にあることを示します。文は改行で分離され、割り当ては<-で示され、let 形式は in キーワードを要求しない挿入です。

If と IO　IO と一緒に使う時、if は扱いにくいものになる可能性があります。概念的には、他のいかなる武運脈での if と違いはありませんが、直感的に展開するのが困難です。System.Directory からの函数 doesFileExists を考えてください:

```
doesFileExist :: FilePath -> IO Bool
```

if 文は以下の「シグニチャ」を持ちます:

```
if-then-else :: Bool -> a -> a -> a
```

すなわち、Bool 値を取り、条件に基づいたある他の値に評価されます。型シグニチャから、if は doesFileExist を直接使えないことは明らかです:

```
wrong fileName =
  if doesFileExist fileName
    then ...
    else ...
```

すなわち、if は Bool 値を望む一方、doesFileExist は IO Bool 値に帰着します。代わりに、正しい値は IO アクションを走らせることで「抽出」されなければいけません:

```
right1 fileName = do
  exists <- doesFileExist fileName
  if exists
    then return 1
    else return 0
```

return の使用に注意してください。do は私たちを IO モナドの「内部に」置くので、return を通すことを除いて「脱出」できません。ここで if インラインを使う必要がないことに注意してください。—let を使って、条件を評価して先に値を得ることもできます:

```
right2 fileName = do
  exists <- doesFileExist fileName
  let result =
        if exists
          then 1
          else 0
  return result
```

再度、return の場所に注意してください。let 文には入れません。代わりに、函数の終わりに一度だけそれを使います。

多重 do　if や case と一緒に do を使う時、もしいずれかの分岐が複数の文を持つなら、別の do が要求されます。if を持つ例:

```
countBytes1 f =
  do
    putStrLn "Enter a filename."
    args <- getLine
    if length args == 0
      -- no 'do'.
      then putStrLn "No filename given."
      else
        -- multiple statements require
        -- a new 'do'.
        do
          f <- readFile args
          putStrLn ("The file is " ++
            show (length f)
            ++ " bytes long.")
```

case を持つ例:

```
countBytes2 =
  do
    putStrLn "Enter a filename."
    args <- getLine
    case args of
      [] -> putStrLn "No args given."
      file -> do
       f <- readFile file
       putStrLn ("The file is " ++
         show (length f)
         ++ " bytes long.")
```

代わりの構文はセミコロンと中括弧を使います。do は依然要求されますが、インデントは不要です。以下のコードは case 例を示しますが、原理は if にも適用されます:

```
countBytes3 =
  do
    putStrLn "Enter a filename."
    args <- getLine
    case args of
      [] -> putStrLn "No args given."
      file -> do { f <- readFile file;
       putStrLn ("The file is " ++
         show (length f)
         ++ " bytes long."); }
```

Export

12ページの moduleに関する節を参照してください。

If, Then, Else

if はいつも値を「返す」ことを思い出してください。それは式であり、単になる制御フロー文ではありません。以下の函数は、与えられた文字列が小文字で始まるかどうかテストし、もしそうなら大文字に変換します:

```
-- Use pattern-matching to
-- get first character
sentenceCase (s:rest) =
 if isLower s
   then toUpper s : rest
   else s : rest
-- Anything else is empty string
sentenceCase _ = []
```

Import

12ページの module に関する節を参照してください。

In

12ページの let を参照してください。

Infix, infixl と infixr

5ページの operators に関する節を参照してください。

Instance

8ページの class に関する節を参照してください。

Let

let を使って函数の中でローカル函数が定義できます。let キーワードにはいつも in が続かなければいけません。in は let キーワードと同じ列に現れなければいけません。定義された函数は、同じスコープ内の (let で定義されたものを含む) 他のすべての函数と変数にアクセス可能です。以下の例では、mult は独立変数 n を元の multiples に渡された x に掛けます。mult は、map が 10 までの数と x の積を与えるために使われます:

```
multiples x =
  let mult n = n * x
  in map mult [1..10]
```

独立変数を持たない let「函数」は実際には定数であり、一度評価され、二度は評価されません。これは函数の共通の部分を捕捉し、それらを再利用するのに役立ちます。以下は、数のリストの和、平均、メディアンを与えるばかげた例です:

```
listStats m =
  let numbers = [1,3 .. m]
      total = sum numbers
      mid = head (drop (m `div` 2)
                       numbers)
  in "total: " ++ show total ++
     ", mid: " ++ show mid
```

脱構築 let 定義の左辺は、部分成分をアクセスする必要がある場合、独立変数を分解することもできます。以下の定義は文字列から最初の 3 つの文字を抽出します:

```
firstThree str =
  let (a:b:c:_) = str
  in "Initial three characters are: " ++
      show a ++ ", " ++
      show b ++ ", and " ++
      show c
```

これは以下のものと違うことに注意してください。以下は文字列が正確に 3 文字を持つ場合だけ機能します:

```
onlyThree str =
  let (a:b:c:[]) = str
  in "The characters given are: " ++
      show a ++ ", " ++
      show b ++ ", and " ++
      show c
```

Of

8ページの case に関する節を参照してください。

Module

モジュールは、函数や型、クラス、インスタンス、他のモジュールをエクスポートするコンパイル単位です。モジュールのエクスポートは複数のソースから来るかもしれませんが、モジュールは 1 つのファイルでだけ定義することができます。Haskell ファイルをモジュールにするには、トップで以下のモジュール宣言を追加だけしてください:

```
module MyModule where
```

モジュール名は大文字で始まらなければいけませんが、他ではピリオドや数、アンダースコアを含むことができます。ピリオドは構造の意味を与えるのに使われ、Haskellコンパイラは、特定のソースファイルがあるディレクトリの印として使いますが、他の点ではそれらは意味を持ちません。

Haskellコミュニティは、Data, System, Networkなどのように、トップレベルモジュール名を標準化しています。もしパブリックへのリリースを計画するなら、あなた自身のモジュールの適切な場所に関してそれらを必ず参考にしてください。

インポート　Haskell標準ライブラリはたくさんのモジュールに分割されています。ソースファイルにインポートすることでそれらのライブラリが供給する機能にアクセスします。ライブラリによってエクスポートされたすべてをインポートするには、モジュール名を使うだけです:

```
import Text.Read
```

すべては**以下のすべて**を意味します: 函数、データ型、コンストラクタ、クラス宣言、そのモジュールによってインポートされ、エクスポートされた他のモジュールでさえ。インポートする名前のリストを与えると、選択的にインポートします。例えば、ここでText.Readからいくるかの函数をインポートします:

```
import Text.Read (readParen, lex)
```

データ型はたくさんの方法でインポートできます。コンストラクタはインポートせず型だけインポートすることができます:

```
import Text.Read (Lexeme)
```

もちろん、これはモジュールが型Lexemeの値に関してパターンマッチングすることを妨げます。1つ以上のコンストラクタを明示的にインポートできます:

```
import Text.Read (Lexeme(Ident, Symbol))
```

与えられた型のコンストラクタすべてをインポートすることもできます:

```
import Text.Read (Lexeme(..))
```

モジュールの中で定義された型とクラスをインポートすることもできます:

```
import Text.Read (Read, ReadS)
```

クラスの場合、データ型のコンストラクタをインポートすることと同様の構文を使って、クラスをインポートすることができます:

```
import Text.Read (Read(readsPrec
                     , readList))
```

データ型と違って、明示的に除外しない限りクラス函数すべてがインポートされます。クラス**だけ**をインポートするには、以下の構文を使います:

```
import Text.Read (Read())
```

除外　もしほとんどだがすべてではない名前をモジュールからインポートするなら、それらすべてをリストすることは退屈なものです。この理由からインポートはhidingキーワードを介して特定することもできます:

```
import Data.Char hiding (isControl
                       , isMark)
```

インスタンス宣言を除いて、任意の型、函数、コンストラクタ、クラスを隠すことができます。

インスタンス宣言　インスタンス宣言はインポートから除外**できない**ことに注意しなければいけません: モジュールがインポートされる時、モジュールの中のすべてのインスタンス宣言がインポートされます。

限定インポート　モジュールがエクスポートする名前(すなわち、函数、型、演算子など)は、限定インポートを通じて付属された前置を持ちます。これは特にPrelude函数と同じ名前を持つたくさんの函数を持つモジュールには役に立ちます。Data.Setはよい例です:

```
import qualified Data.Set as Set
```

この形式は、Data.Setがエクスポートした任意の函数、型、コンストラクタ、または他の名前が与えられた**別名**(すなわち、Set)で前置されるよう要求します。以下は、リストから重複すべてを取り除く1つの方法です:

```
removeDups a =
  Set.toList (Set.fromList a)
```

二番目の形式は別名を生成しません。代わりに、前置はモジュール名になります。文字列がすべて大文字かチェックする簡単な函数を書くことができます:

```
import qualified Char

allUpper str =
  all Char.isUpper str
```

指定した前置を除いて、通常のインポートと限定インポートは同じ構文をサポートします。インポートされた名前は上で記述されたのと同じ方法で限定することができます。

エクスポート　もしエクスポートリストが供給されないなら、函数、型、コンストラクタなどすべてはモジュールをインポートした誰にも利用可能になります。インポートされた任意のモジュールはこの場合、エクスポート**されない**ことに注意してください。where キーワードの前に括弧でくくられた名前のリストを加えることで、エクスポートされる名前を限定することができます:

```
module MyModule (MyType
  , MyClass
  , myFunc1
  ...)
where
```

どの函数、型、コンストラクタ、クラスがエクスポートされるか指定するために、2,3 の違いはありますが、インポートのために使われるのと同じ構文が使えます。もしモジュールが別のモジュールをインポートするなら、それはそのモジュールもエクスポートすることができます:

```
module MyBigModule (module Data.Set
  , module Data.Char)
where

import Data.Set
import Data.Char
```

モジュールはそれ自身を再エクスポートさえできます。それは、ローカル定義すべてと与えられたインポートされたモジュールをエクスポートする時、役に立つかもしれません。以下では、自身と Data.Set をエクスポートしますが、Data.Char はエクスポートしません:

```
module AnotherBigModule (module Data.Set
  , module AnotherBigModule)
where

import Data.Set
import Data.Char
```

Newtype

data が新しい値を導入し、type がただ同義語を生成する一方、newtype は間の何かに収まります。newtype の構文は全く正弦されています— ただ 1 つのコンストラクタが定義できて、そのコンストラクタはただ 1 つの独立変数を取ることができます。上の例に続いて、以下のように Phone 型を定義できます:

```
newtype Home = H String
newtype Work = W String
data Phone = Phone Home Work
```

type と対照的に、Phone に関する H と W の「値」はただの文字列値では**ありません**。型チェッカーはそれらを完全に新しい型として扱います。それは、上からの lowerName 函数はコンパイルされないことを意味します。以下は型エラーを生じます:

```
lPhone (Phone hm wk) =
  Phone (lower hm) (lower wk)
```

代わりに、lower を適用する「値」を得るためにパターンマッチングを使わなければいけません:

```
lPhone (Phone (H hm) (W wk)) =
  Phone (H (lower hm)) (W (lower wk))
```

キーとなる観察は、このキーワードは新しい値を導入しないということです; 代わりにそれは新しい型を導入します。これは 2 つのとても役立つ性質を与えてくれます:

- 実際には新しい値を生成しないので、新しい型に実行時コストが結びつきません。言い換えると、newtype は絶対にただです!
- 型チェッカーは、Int や String のような共通の型を、プログラマが指定した制限された方法で使うことを強制することができます。

最後に、data 宣言に付随されることができるいかなる deriving 節も newtype を宣言する時使うことができることに注意してください。

Return

10ページの doを参照してください。

Type

このキーワードは**型同義語** (すなわち別名) を定義します。このキーワードは、data や newtype のように新しい型を定義したりはしません。コードをドキュメントするために役に立ちます。他の点で、与えられた函数や値の実際の型に影響を持ちません。例えば、Person データ型は以下のように定義されます:

```
data Person = Person String String
```

ここで、最初のコンストラクタの独立変数はファーストネームを、二番目はラストネームを表します。しかしながら、2 つの独立変数の順序や意味はあまりはっきりしていません。type 宣言が助けになります。

```
type FirstName = String
type LastName = String
data Person = Person FirstName LastName
```

type は同義語を導入するので、型チェックはいかなる意味でも影響はありません。以下のように定義された函数 lower:

```
lower s = map toLower s
```

それは以下の型を持ちます。

```
lower :: String -> String
```

これは、型 FirstName や LastName を持つ値に関して、ただ簡単に使うことができます:

```
lName (Person f l) =
  Person (lower f) (lower l)
```

Where

let と同様、where はローカル函数、定数を定義します。where 定義のスコープは現在の函数です。もし函数がパターンマッチングを通して複数尾定義に分解されるなら、特定の where 節のスコープはその定義にだけ適用されます。例えば、以下の函数 result は、函数 strlen に与えられた独立変数に依存して異なる意味を持ちます:

```
strlen [] = result
  where result = "No string given!"
strlen f = result ++ " characters long!"
  where result = show (length f)
```

Where vs. Let where 節は函数定義のレベルでだけ定義することができます。普通、それは let 定義のスコープと同一です。唯一の違いはガードが使われた時です。where 節のスコープはガードすべてに及びます。対照的に、let 式のスコープは、現在の函数節**かつ**(もしあるなら) ガードだけです。

貢献者

パッチや役立つ示唆を提供してくれた以下の方々に感謝致します: Dave Bayer, Paul Butler, Elisa Firth, Marc Fontaine, Brian Gianforcaro, Cale Gibbard, Andrew Harris, Stephen Hicks, Kurt Hutchinson, Johan Kiviniemi, Patrik Jansson, Adrian Neumann, Barak Pearlmutter, Lanny Ripple, Markus Roberts, Holger Siegel, Falko Spiller, Adam Vogt, Leif Warner, そして Jeff Zaroyko.

バージョン

これはバージョン 2.8 です。(原文の) ソースは GitHub(http://github.com/m4dc4p/cheatsheet) で見つけられます。(原文の) 最新リリースバージョンの PDF は http://cheatsheet.codeslower.com からダウンロードできます。他のプロジェクトや書き物に関しては、CodeSlower.com(http://blog.codeslower.com/) に寄ってください。

日本語訳の問題に関しては、市川雄二 (mailto:ichikawa.yuji@gmail.com) まで連絡ください。