A Gentle Introduction to Haskell 98

Paul Hudak, Yale University
John Peterson, Yale University
Joseph Fasel, Los Alamos National Laboratory
September 28, 1999

翻訳：山下 伸夫 <nobsun@sampou.org>

訳者注：この翻訳は原文にある上記の記述を根拠に公開するものです。訳文の誤り、抜け、など訳文の質を向上させるための御指摘は歓迎いたします。

宗像注：直訳調の部分を書き直し普通の日本文に近づけた。

1 イントロダクション

このチュートリアルの目的は、プログラミングを教えることでも、関数プログラミングを教えることでもありません。高度に技術的な解説書である Haskell Report [4] を補うことを意図しています。少くとも 1 つの他の言語(関数型言語であることが望ましい)の経験のある人にやさしい Haskell の入門書を提供することを目指しました。 Haskell の言語設計原理、関数型プログラミング言語と技術を調査するには文献[3]を参照してください。

Haskell 言語は、1987 年に生れて以来、長足の進化をとげました。このチュートリアルでは Haskell 98 を扱います。これ以前のバージョンはすでに時代遅れになっています。皆さんには Haskell 98 の使用をお勧めします。 Haskell 98 の実装にはさらに多くの拡張がほどこされていますが、これらの拡張はまだ正式の Haskell 仕様には採用されていません。本書ではこうした拡張については触れません。

言語を紹介する方針は次の通りです。概念の背景を説明し、用語を定義し、例をあげ、詳細については *Haskell Report* の該当個所を示します。この「やさしい Haskell 入門」を読み終えるまでは、細いところは無視することを勧めます。 Haskell の Standard Prelude (*Haskell Report* の付録 A と *Library Report*[5]) には沢山の役に立つ Haskell コードがあるので、このチュートリアルを終えたら全部を通しで読むことをお勧めします。 Standard Prelude を読めば、実際の Haskell コードがどのようなものであるかが掴めるだけでなく、Haskell の定義済み標準関数や型に慣れ親しむことができます。http://www.haskell.org には Haskell 言語と実装に関する豊富な情報があります。

[構文は例のなかで必要になったときに、このように [ ] で括って示します。*Haskell Report* はすべての典拠です。参照はレポート(§2.1)のように示します。]

Luca Cardelli の表現でいえば、Haskell は**強く型付けされた** *(typeful)* プログラミング言語であると言えます。型は誤用されやすいので、初心者は最初から複雑で強力な Haskell の型システムに十分注意を払うべきです。Perl、Tcl、Scheme のような強い型付けのない(untypeful)言語の経験しかない人にとっては少し骨が折れるかもしれません。Java、C、Modula、ML に明るい人にとっても自明ではないでしょう。 Haskell の型システムは他の言語と違いより強力だからです。「強い型付けのあるプログラミング」は Haskell プログラミングの一部であって避けては通れません。

2 関数型言語 Haskell

Haskell は純関数型言語なので、計算は**式** *(expression)* を**値** *(value)* に簡約することで行われます。すべての値は**型** *(type)* をもちます。直観的には型は値の集合と考えることができます。式には、整数 5, 文字 'a', 関数 ¥x -> x+1 等のアトミックな値が含まれます。リスト [1,2,3] や 組 ('b',4) も式です。

式が値を表しているのと同様、**型式**(*type expression*)は型値あるいは単に**型**(*type*)を表します。型式には、Integer(多倍長整数型)、Char(文字型)、Integer -> Integer(Integer から Integer への関数型)等のアトミックな型と、Integer のリスト[Integer]、Char と Integer の組(Char,Integer)のような構造を持つ型があります。

値はすべて関数の引数と返り値になり得、データ構造に入れることもできます。一方、型はある意味では値について記述する役割のものです。値とその型を結びつけることを**型付け**(*typing*)といいます。例であげた値の型付けは以下のようになります。:: は「の型は」と読みます。

```
    5  :: Integer
  'a'  :: Char
  inc  :: Integer -> Integer
[1,2,3] :: [Integer]
('b',4) :: (Char,Integer)
```

関数は一連の等式で定義します。例えば、関数 inc は一つの等式で定義することができます。

```
inc n          = n+1
```

等式は**宣言**(*declaration*)の例です。**型シグネチャ**(*type signature*(§4.4.1))も宣言です。型シグネチャを用いて inc の型付けを宣言することができます。

```
inc            :: Integer -> Integer
```

式 e_1 が評価(簡約)されて別の式や値 e_2 になるのを e_1 => e_2 と表記します。例 inc (inc 3) => 5

Haskell の静的型システムは、型と値の関係を定義し Haskell プログラムの**型安全**(*type safe*)を保証します(§4.1.3)。プログラマが整合性のない型を使っていないことを保証します。 'a' + 'b' は適切に型付けできません。静的型付けを行う言語の利点は、型エラーはすべてコンパイル時に検出されることです。型システムはエラーをすべて検出するわけではありません。1/0 という式は型付け可能ですが、評価すると実行時エラーになります。型システムはコンパイル時に多くのプログラムエラーを見つけ、プログラムの正しさの確認に役立ち、実行時の型タグや型テストを省略する効率のよいコードを生成します。

型システムは型シグネチャが正しいことを確認します。 Haskell の型システムは強力なので、型シグネチャをまったく書かなくても良いくらいです。この場合、型システムが正しい**型推論**(*type infer*)を行ってくれます。しかし、inc でやったように型シグネチャを書くのは良い考えです。型シグネチャは最も有効なドキュメントであり、プログラミングエラーを白日の下に照らし出すのに役に立つからです。

[型を表わす識別子は Integer、Char のように大文字ではじまります。 inc のように値を表わす識別子は小文字ではじまります。また、foo、f0o、f00 はすべて違う識別子です。]

2.1 多相型

Haskell では**多相型**(*polymorphic type*)を取り扱うことができます。多相型はすべての型の上で全称修飾された型を表します。 [a] はすべての型 a に対して a のリスト型からなる型の族をあらわします。 [1,2,3], ['a','b','c'], [[1],[2,3]] は皆 [a] のメンバーです。

[上の a のような識別子を**型変数**(*type variable*)とよびます。型変数は大文字にはせず、 Int のような特定の型と区別します。型変数は暗黙に全称修飾されています。]

リストは関数型言語でよく使われるデータ構造であり、多相性の概念の説明に丁度いいデータ構造です。リスト [1,2,3] はリスト 1:(2:(3:[])) の簡略表記です。[] は空リスト、: は中置構成子で最初の引数を次の引数(リスト)の前に加えます。: は右結合性をもつので 1:2:3:[] と書くことができます。

ユーザ定義関数の例としてリストの要素数を数える問題を考えましょう。

```
length                  :: [a] -> Integer
length []               =  0
length (x:xs)           =  1 + length xs
```

この定義はこれだけで十分説明になります。この等式は「空リストの長さは 0、最初の要素が x で残りが xs であるようなリストの長さは 1 に xs の長さを加えたものである」と読むことができます。ここでは命名の習慣がつかわれています。 xs は x の複数形であり、読むときもそのように読みます。

この例は Haskell の**パターン照合**(*pattern matching*)を浮き彫りにしています。等式の左辺には [] や x:xs というパターンが含まれています。関数適用の際にパターンは実引数と照合します。 [] は空リストにのみマッチし、x:xs は 1 つ以上の要素をもつリストにマッチします。このとき、 x は最初の要素に束縛され、 xs はリストの残りの部分に束縛されます。照合が成功すると右辺が評価され、関数適用の結果として返されます。照合が失敗すると次の等式が試されます。すべての等式でパターン照合が失敗すればエラーとなります。パターン照合を使って関数を定義するのは Haskell の常套手段です。Haskell の種々のパターンに慣れておく必要があります。

関数 length は多相関数の例でもあります。この関数はあらゆる型の要素をもつリストに適用することができます。たとえば、[Integer]、[Char]、[[Integer]] などです。

リスト上の別の多相関数をあげておきましょう。関数 head はリストの最初の要素を返し、関数 tail は最初の要素をのぞく残りのリストを返します。関数は引数のすべての値に対して定義されているわけではありません。これらの関数が空リストに適用されると実行時エラーが起こります。

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

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

多相型を使うと或る型が別の型よりも一般的であり、定義されている値の集合がより大きいことに気付きます。型 [a] は型 [Char] よりも一般的であり、後者の型は前者の型から導出できます。型の一般性の順序付けに関して、 Haskell の型システムは 2 つの重要な特性を発揮します。正しく型付けされた式は唯一の**主型**(*principal type*)を持つことが保証され、主型は自動的に推論できます(§4.1.3)。型推論機構はプログラマが型の問題で頭を悩ますのを軽減してくれます。

主型は「式や関数のすべてのインスタンスを含む最も一般性のない型」ということができます。 head の主型は [a] -> a であり、[b] -> a, a -> a, a では一般的すぎ、[Integer]->Integer では限定的すぎます。唯一の主型が存在するのは **Hindley-Milner 型システム**の特徴です。この型システムは、Haskell、ML、Miranda (Research Software Ltd.の登録商標)の型システムの基盤になっています。

2.2 ユーザ定義型

Haskell では data 宣言を使って型を定義することができます(§4.2.1)。Haskell の定義済の型で重要なのは真理値の型です。

```
data Bool               = False | True
```

ここで定義した型は Bool 型であり 2 つの値 True と False を持っています。 Bool は無引数の**型構成子**(*type constructor*)です。 True や False は無引数の**データ構成子**(*data constructor*)(単に構成子ともいう)です。色の型を定義しましょう。

```
data Color              = Red | Green | Blue | Indigo | Violet
```

Bool 型と Color 型は列挙型の例であり、有限個の無引数のデータ構成子から構成されています。つぎは単一のデータ構成子からなる型の例です。

```
data Point a            = Pt a a
```

Point 型は本質的には型の直積(2 引数)であり、**タプル型**(*tuple type*)と呼ばれます。タプル型は別の言語ではレコード型と呼ばれています。 一方、 Bool や Color のように複数の構成子をもつ型は**直和型**(*(disjoint) union* or *sum type*)と呼ばれます。

重要なのは Point は多相型だということです。 Point はすべての型 t に対して t を座標とする点の型を定義するものです。 Point は 1 引数の型構成子です。型 t から新しい 型 Point t が作られます。同様に、リストの [] も型構成子です。すべての型 t に [] を適用すると、新しい型 [t] になります。Haskell の構文では [t] を [] t と書くことが許されています。同様に -> は中置の型構成子です。2 つの型 t と u に対して、t -> u は型 t の要素から型 u の要素へ写像する関数の型です。

データ構成子 Pt の型は a -> a -> Point a であることに注意します。式 Pt 'a' 1 は正しく型付けできません。'a' と 1 は別の型だからです。以下は正しい型付けです。

```
Pt  2.0  3.0          :: Point Float
Pt  'a'  'b'          :: Point Char
Pt True False         :: Point Bool
```

データ構成子を適用して**値**を得ることと、**型構成子**を適用して**型**を得ることを区別することが重要です。前者は実行時にどのように計算するかということであり、後者はコンパイル時の型安全を確かめる型システムのプロセスの一部です。

[型構成子 Point とデータ構成子 Pt は別の名前空間にあり、型構成子とデータ構成子に同じ名前を使うことができます。]

```
data Point a = Point a a
```

最初はややこしく見えますが、これで型構成子とデータ構成子の関連が明らかになります。

2.2.1 再帰型

型は再帰的に構成することができます。二分木の型を考えてみましょう。

```
data Tree a           = Leaf a | Branch (Tree a) (Tree a)
```

二分木の多相型を定義しました。木の要素は a 型の値を含む Leaf ノード、再帰的に二つのサブツリーを含む Branch ノードです。

データ宣言を読むときは、 Tree は型構成子、 Branch や Leaf はデータ構成子であることを思いだしてください。構成子間の関係を理解したら、上の宣言は本質的には以下のような構成子 Branch や Leaf の型の定義であることが分かるでしょう。

```
Branch                :: Tree a -> Tree a -> Tree a
Leaf                  :: a -> Tree a
```

Leaf の要素のリストを返す関数 fringe を定義したいとします。関数の型は Tree a -> [a] です。関数 fringe は、 length と同様、パターン照合を使って定義しています。違いはデータ構成子 Leaf と Branch があることです。 ++ は中置演算子で 2 つのリストを連結します。

```
fringe                    :: Tree a -> [a]
fringe (Leaf x)           =  [x]
fringe (Branch left right) =  fringe left ++ fringe right
```

2.3 型の同義名

Haskell には**型の同義名**(*type synonym*)を定義する方法があります。よく使う型に別名を付けるものです。型の同義名は type 宣言によって定義しますが、新しい型を定義するものではありません。(§4.2.2)。

```
type String           = [Char]
type Person           = (Name,Address)
type Name             = String
```

```
data Address            = None | Addr String
```

Person -> Name の型は (String,Address) -> String と同じ型です。型の同義名をニーモニックとすることでプログラムの可読性が改善されます。多相型に新しい名前を付けることもできます。型 a の値と型 b の値を結びつける「連想リスト(association list)」型を例示します。

```
type AssocList a b              = [(a,b)]
```

2.4 組み込み型は特別な型ではない

リスト、タプル、整数、文字などの組み込み型を紹介し、ユーザ定義型の定義方法を示しました。組み込み型はユーザ定義型と比べて特別なことがあるのでしょうか？答えはノーです。組み込み型は利便性と歴史的な慣習によるものであり、意味としてはユーザ定義型と何も違いがありません。

(以下、組み込み型が data 宣言でどのようなものになり得るかの例示を省略)

2.4.1 リストの内包表記と数列表記

他の関数型言語と同様、Haskell には**リストの内包表記**(*list comprehension*)として知られるリストを生成する構文があります。次の例で説明します。

```
[ f x | x <- xs ]
```

この式は直観的に「xs から順に引き出した x について、 f x をすべて集めたリスト」と読みます。x <- xs は**生成部**(*generator*)と呼ばれています。生成部を複数持つこともできます。

```
[ (x,y) | x <- xs, y <- ys ]
```

要素は左から右へ入れ子の生成部から選ばれていきます。右端にある生成部がもっとも速く変化します。xs を [1,2]、ys を [3,4] とすれば、結果は [(1,3),(1,4),(2,3),(2,4)] になります。

生成部のほかに**ガード**(*guard*)とよばれるブール式を書くことができます。ガードは生成される要素に制約を与えます。お馴染のソーティングアルゴリズムの定義をあげておきます。

```
quicksort  []           =  []
quicksort (x:xs)        =  quicksort [y | y <- xs, y<x] ++ x : quicksort [y | y <- xs, y>=x]
```

リストの**数列表記**という構文もサポートしています。

```
[1..10]   => [1,2,3,4,5,6,7,8,9,10]
[1,3..10] => [1,3,5,7,9]
[1,3..]   => [1,3,5,7,9, ...(infinite sequence)
```

2.4.2 文字列

文字列リテラルの "hello" は文字リスト ['h','e','l','l','o'] の簡略表記です。 "hello" の型 String は文字リストの型の同義名であり、リスト上の多相関数は文字列にも適用できます。

```
"hello" ++ " world" => "hello world"
```

3 関数

Haskell の関数の貌を見ていきます。二つの引数を足す関数の定義を考えましょう。

```
add                     :: Integer -> Integer -> Integer
add x y                 =  x + y
```

これは**カリー**(*curry*) 化された関数の例です。呼び名は、この概念を有名にした Haskell Curry から来ています。**アンカリー**(*uncurry*) 化された関数にはタプルを使います。

```
add (x,y)               = x + y
```

関数 add の適用は add e_1 e_2 という形式になります。これは (add e_1) e_2 と同等の式です。関数適用は左結合性をもちます。関数 add を最初の引数に適用すると新しい関数ができて、これを二つ目の引数に適用するということです。関数 add の型 Integer -> Integer -> Integer とも整合が取れています。この型は Integer -> (Integer -> Integer) と同等です。 -> は右結合性をもちます。 add を使うと違うやり方で inc を定義することができます。

```
inc                     = add 1
```

これはカリー化された関数の**部分適用**(*partial application*)の例であり、関数が返り値となる例です。今度は関数を引数としてわたす例を考えてみましょう。 map 関数が完璧な例です。

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

[関数適用は最も高い優先順位をもっており、等式の右辺は (f x) : (map f xs) と構文解析されます。]

map 関数は多相型であり、 map の型シグネチャは最初の引数が関数であることを明示しています。 2 つの a は同じ型にインスタンス化されなければなりません。 b についても同じです。 次の例はリストの要素をそれぞれ 1 だけ増加させます。

```
map (add 1) [1,2,3] => [2,3,4]
```

このような使いかたをする関数を**高階**(*higher-order*)関数と呼びます。

3.1 ラムダ抽象

ラムダ抽象(*lambda abstraction*)を使って無名関数を定義することができます。 inc 関数は ¥x -> x+1、add 関数は ¥x -> ¥y -> x+y と書くことができます。入れ子のラムダ抽象は ¥x y -> x+y と書いても同じです。inc x と add x y の等式定義は以下の等式を簡略化したものです。

```
inc                     = ¥x   -> x+1
add                     = ¥x y -> x+y
```

x の型が t_1 、 exp の型 が t_2 の場合、 ¥x->exp の型は t_1 -> t_2 になります。

3.2 中置演算子

中置演算子は関数なので等式を使って定義することができます。リストの連結演算子を定義すると次のようになります。

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

[一般の識別子が英数字で構成されるのに対して、演算子の字句はすべて記号で構成されます。 - は前置演算子にも中置演算子にもなります。 - 以外に前置演算子はありません。]

関数上の重要な中置演算子である**関数合成**(*function composition*)の例を見てみましょう。

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

3.2.1 セクション

中置演算子は関数ですから部分適用が可能です。 Haskell では中置演算子の部分適用のことを**セクション**(*section*)といいます。例を見ましょう。

```
(x+) =  ¥y    -> x+y
(+y) =  ¥x    -> x+y
 (+) =  ¥x y -> x+y
```

[この ( ) は必須です。]

セクション (+) は中置演算子を同等の関数へ変換します。セクションは中置演算子を関数の引数として渡すのに便利な方法です。map (+) [1,2,3] が関数のリストを返すことを確かめてください。すでに定義した add が (+)、inc が (+1) であることも分かります。

```
inc                     = (+1)
add                     = (+)
```

中置演算子を関数に変換しましたが、その逆は可能でしょうか。可能です。関数をバッククウォートで括るだけです。 x `add` y は add x y と同じです。リストの要素かどうかを調べる述語関数 elem は、x `elem` xs と中置演算子にした方が x is an element of xs と読みやすくなります。

関数の定義方法がいろいろあって混乱したかもしれません。これらの定義方法のサポートを決定するにあたっては、歴史的な慣習が反映されていますし、中置演算子と関数の取り扱いの一貫性を確保したいという要望も反映されています。

3.2.2 結合性宣言

結合性宣言(*fixity declaration*)は、中置演算子と中置構成子(`elem` のような一般の識別子も含む)の優先度と右結合(infixr)、左結合(infixl)、非結合(infix)を宣言します。優先度は 0 から 9 で指定します(9 が最強、関数適用の優先レベルは 10)。 ++ と . の結合性宣言は以下のようになります。

```
infixr 5 ++
infixr 9 .
```

両方とも右結合が指定されています。優先度は ++ が 5 、 . が 9 です。同じ結合性宣言で 1 つ以上の演算子を指定することができます。演算子の結合性宣言をしない場合、デフォルトで infixl 9 が指定されたことになります。結合性ルールの詳細は(§5.6)を参照してください。

3.3 関数は非正格

bot が次のように定義されているとします。 bot は停止しない式です。

```
bot                     = bot
```

停止しない式の**値**を⊥(ボトム)で表わします。1/0 のように実行時エラーになる式もこの値を持ちます。このようなエラーは回復不可能であり、プログラムをこれ以上継続できなくなります。I/O システムの EOF エラーなどは回復可能であり、例外の方法で取り扱います。

関数 f が**正格**(*strict*)であるとは、停止しない式に適用されれば適用式も停止しないことをいいます。f bot が ⊥ の時かつその時にかぎり f は正格であるといいます。ほとんどのプログラミング言語の関数は正格ですが、 Haskell では違います。常に 1 を返す関数 const1 を考えましょう。

```
const1 x                = 1
```

const1 bot の値は 1 です。操作的にいえば、const1 は引数の値を必要とせず、引数は評価されません。だから停止しない計算に陥いることがないのです。非正格関数は遅延関数(lazy function)とも呼ばれ、引数については遅延評価とか必要呼び評価といわれています。 const1 (1/0) も 1 になります。

非正格関数は多くの場面で役に立ちます。主要な利点は、プログラマが評価の順を気にしなくてもよいことです。必要もないのに余分にコストの掛かる値を計算するかもしれない、と心配せずに関数に渡すことができます。

別の方法で非正格関数を説明します。Haskell は、伝統的な言語の**代入**(*assignment*)ではなく、**定義**を用いて計算をすすめます。たとえば、

```
v                       = 1/0
```

の宣言は「 v を 1/0 と定義する」と読み、「 1/0 を計算し結果を v へ代入する」とは読みません。宣言だけでは計算は起こりません。 v の値(定義)が必要になったときにゼロ除算エラーが発生します。代入するプログラミングでは、プログラムの意味は代入する順序に依存するので、代入の順序に細心の注意が必要です。一方、定義は単純で、どのような順で出現してもかまわず、プログラムの意味には影響を与えません。

3.4 「無限」のデータ構造

非正格な性質の利点はデータ構成子も非正格であることです。構成子は特殊な関数にすぎません。関数との違いはパターン照合で使われることです。リスト構成子 : は非正格です。非正格の構成子で**無限**(*infinite*)のデータ構造を定義することができます。次は 1 の無限リストです。

```
ones                    = 1 : ones
```

面白いのは numsFrom 関数です。numsFrom n は n から始まる無限の整数リストです。

```
numsFrom n              = n : numsFrom (n+1)
```

これを使って平方数の無限リストを構成することができます。 (^) は中置の指数演算子です。

```
squares                 = map (^2) (numsFrom 0)
```

実際の計算では無限列から有限部分を取り出したいわけです。この用途で使える関数 take、takeWhile、filter 等が定義されています。多数の有用なリスト関数がモジュール PreludeList にあるので参照してください。 take n はリストから最初の n 個の要素を取り出します。

```
take 5 squares => [0,1,4,9,16]
```

ones は**循環リスト**(*circular list*)の例です。遅延性は効率の面も重要です。本当の循環リストとして実装されるので空間が節約されます。

フィボナッチ数列は循環数列を使って効率よく計算することができます。 fib が自分自身を使って定義されている点に注目してください。自分の尻尾をおいかける定義になっています。

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

zip は二つのリストの要素を互いにペアにしたリストを返します。

```
zip (x:xs) (y:ys)       = (x,y) : zip xs ys
zip  xs     ys          = []
```

3.5 エラー関数

Haskell には error という組み込み関数があります。返す値は多相型で実際の型は分かりません。この型の値を受けとることはないのです。

```
error                   :: String -> a
```

すべての型によって共有される値⊥があります。 error の返す値は⊥です。すべてのエラーの値は⊥であることを思い出してください。 error の実装は、チェックのために文字列引数を印字するというのが妥当なところでしょう。 error 関数は何か誤りがあったときにプログラムを停止させるのに使用します。head の定義は以下のようになっています。

```
head (x:xs)             =  x
head  []                =  error "head{PreludeList}: head []"
```

4 Case 式とパターン照合

関数はパターンを使って定義します。関数 length や fringe は、リストや Tree の**データ構成子**のパターンを使って定義されています。この節ではパターン照合を詳細に見ていきます(§3.17)。

パターン照合は、組み込み型(リスト、タプル、文字列、数、文字など)やユーザ定義型に関係なく、すべての型で使うことができます。定数のタプルのパターンを照合する contrived 関数の例を見ましょう。

```
contrived :: ([a], Char, (Int, Float), String, Bool) -> Bool
contrived    ([],  'b',  (1,   2.0),   "hi",   True) = True
```

関数の**仮引数**(*formal parameters*)または**変数**(*variable*)もパターンです。パターン照合が必ず成功し、副作用として仮引数(変数)は実引数に束縛されます。等式のパターンには 2 つ以上の同じ仮引数を含んではいけません。これは**線型性**(*linearity*)と呼ばれる性質です(§3.17, §3.3, §4.4.2)。

仮引数のように必ず照合が成功するパターンを**不可反駁**(*irrefutable*)パターンといいます。逆に**反駁**(*refutable*)パターンは照合が成功しない可能性があります。 contrived は反駁パターンの例です。不可反駁パターンには他に 3 種類があります。そのうちの 2 つを紹介します。残りの遅延パターンについては 4.4 節まで待ってください。

アズパターン

アズパターン(*as-pattern*)はパターンに名前を付けます。パターンに名前が付いていると等式の右辺で使えて便利なことがあります。リストの第一要素を複製する関数は以下の二通りで書けます。

```
f (x:xs)              = x:x:xs
f s@(x:xs)            = x:s
```

x:xs のパターンに名前 s を付け、右辺でパターンと同じ式 x:xs を使わずに済ますことができます。アズパターンを使う利点は x:xs を再構築しないことです。アズパターンは常に照合に成功しますが、サブパターン(x:xs)は照合に成功しないこともあります。

ワイルドカード

値を気にしないパターン照合があります。 head と tail 関数は次のように書き換えることができます。

```
head (x:_)            = x
tail (_:xs)           = xs
```

この定義ではパターンの一部は気にしないことが「広告」されています。各ワイルドカードはどんなものとも照合が成功し、何にも束縛されません。というわけで、ワイルドカードは等式のなかに 1 つ以上あってもかまいません。

4.1 パターン照合の意味論

パターン照合はどのような順序で行われ、どのパターン照合も成功しなかったらどうなるのでしょうか。

パターン照合は**失敗**(*fail*)、**成功**(*succeed*)、**発散**(*diverge*)することがあります。成功した照合はパターンの中の仮引数を束縛します。発散は値がエラー(⊥)を含んでいるときに起こります。照合は「上から下、左から右」に行われます。等式のどこでパターン照合が失敗しても等式のパターン照合が失敗したことになり、次の等式が試されます。すべての等式でパターン照合が失敗すると関数適用は実行時エラーとなり、値は⊥になります。

[1,2] と [0,bot] をパターン照合する場合、1 と 0 はマッチしないのでパターン照合は失敗します。[1,2] と [bot,0] をパターン照合する場合、1 と bot を照合すると発散がおこります。 bot は ⊥ に束縛されていることを思い出してください。

パターンが**ガード**部をもつことがあります。数の符号を調べる関数は次のようになります。

```
sign x |  x > 0       =  1
       |  x == 0      =  0
       |  x < 0       = -1
```

一連のガードが同じパターン x を使っていることに注意します。これらは上から下へ評価され、最初に True となったところでパターン照合が成功します。

4.2 例

パターン照合のルールが関数の意味に微妙に影響することがあります。次の take と take1 の定義を考えてみましょう。最初の 2 つの等式の順序が逆になっています。

```
take  0  _       =  []                    take1  _  []      =  []
take  _  []      =  []                    take1  0  _       =  []
take  n  (x:xs)  =  x : take (n-1) xs      take1  n  (x:xs)  =  x : take1 (n-1) xs
```

次の結果に注目してください。

```
take  0   bot  => []                      take1 0   bot  => ⊥
take  bot []   => ⊥                       take1 bot []   => []
```

どちらがよいかは難しいところです。take は 第一引数が 0 のときに第二引数で発散しないことに重きをおき、 take1 は第二引数が [] のとき第一引数で発散しないことにに重きをおいて定義されていることがわかります。標準プレリュードは take の定義を採用しています。

4.3 Case 式

関数定義のパターン照合を示してきました。**case 式**(*case expression*)はパターン照合を使い値の構造的な性質に基づいた「ディスパッチ制御」を可能にします。関数定義のパターン照合の意味は、case 式より単純なものと考えられることが Haskell レポートに示されています。次のようなパターン p_{ij} をもつ関数定義は、意味論的には右側の case 式と同等です。ここで xi は新しい識別子です。

```
                                f x1 x2 ... xk = case (x1, ..., xk) of
f p11 ... p1k = e1                                   (p11, ..., p1k) -> e1
...                        ==>                       ...
f pn1 ... pnk = en                                   (pn1, ..., pnk) -> en
```

take の関数定義は以下の case 式と同等になります。一般的な解釈については(§4.4.2)を参照。

```
take m ys              = case (m,ys) of
                              (0, _)    ->  []
                              (_, [])   ->  []
                              (n, x:xs) ->  x : take (n-1) xs
```

case 式のパターン照合ルールは関数定義の場合と同じです。型については言及しませんでしたが、case 式の右辺や関数定義の一連の等式の右辺は同じ型でなければなりません。正確には同じ主型を共有しなければなりません。

Haskell の条件式 if e_1 then e_2 else e_3 は下の構文の簡略形です。

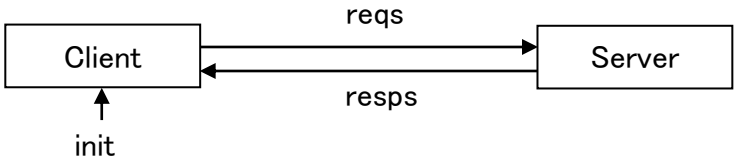
```
case e1 of True  -> e2
           False -> e3
```

この展開を見ると e_1 が Bool 型、e_2 と e_3 は同じ型でなければならないことがはっきりと分かります。if_then_else_ を関数と見なせば Bool->a->a->a の型をもちます。

4.4 遅延パターン

Haskell では**遅延パター ン**(*lazy pattern*)とよばれるパターンがあり、~pat という形式をもちます。遅延パターンは**不可反駁**であり、値 v と ~pat の照合は pat が何であっても成功します。 pat のなかの識別子が右辺で利用されるときは v が pat にマッチすると仮定した場合の値に束縛されるか、さもなければ ⊥ になります。

例をあげましょう。無限リストは**シミュレーション**プログラムを書くのに使いよいデータ構造です。こうした使い方をする無限リストを**ストリーム**といいます。サーバプロセス server とクライアントプロセス client の間のやりとりの簡単なケースを考えてみましょう。 client は**要求**を server に送り、server は要求に対して**応答**を返します。 client は引数として初期メッセージ init を取ります。



メッセージシーケンスをシミュレートするコードは図式と同じになります。

```
reqs                    = client init resps
resps                   = server reqs
```

サーバとクライアントの構造は以下のように仮定します。 next 関数はサーバの応答を得て次の要求を決定し、process 関数はクライアントの要求に対する応答を返すとします。

```
client init (resp:resps) = init : client (next resp) resps
server      (req:reqs)   = process req : server reqs
```

このプログラムは何も出力しないという問題があります。問題は client が最初の要求 init を送信する前に resp とパターン照合を試みることです。パターン照合が「早すぎる」のです。これを是正するには client の再定義が必要です。

```
client init resps         = init : client (next (head resps)) (tail resps)
```

これで動作するようになりますが、遅延パターンを使う方がもっとよい解決法です。

```
client init ~(resp:resps) = init : client (next resp) resps
```

遅延パターンは不可反駁なので直ちに照合は成功し、最初の要求が発信され、最初の応答が生成されます。残りは reqs と resps の再帰の部分が面倒をみてくれます。この例は以下の定義を加えると実際に動作します。

```
init                    = 0
next resp               = resp
process req             = req+1
```

動作結果を例示します。

```
take 10 reqs => [0,1,2,3,4,5,6,7,8,9]
```

4.5 字句の有効範囲と入れ子形式

式のなかで入れ子の有効範囲を作る「ブロック構造」が欲しくなることがあります。ある部分だけの局所的な束縛を作りたいときです。

let 式

let 式(*let expression*)は局所的な束縛が必要なときに役に立ちます。簡単な例を考えてみましょう。

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

let 式によって作られる局所的な束縛は**相互再帰的**(*mutually recursive*)です。パターン束縛は遅延パターンとして扱われます。 let 式内で許される宣言は、関数束縛、パターン束縛、型シグネチャーです。

where 節

where 節(*where clause*)はガードをもつ等式にまたがる有効範囲の束縛に便利な構文です。

```
f x y  |  y>z          =  ...
       |  y==z         =  ...
       |  y<z          =  ...
     where z = x*x
```

where 節は等式の集まりと case 式のトップレベルでしか使えません。where 節は関数宣言と case 式の構文の一部です。where 節の有効範囲はそれを覆う式の上だけです。let 式の束縛と同じ制約が where 節にも適用されます。

4.6 レイアウト

Haskell で等式や宣言の終端を示す終端記号(; 等)がなぜ必要ないのでしょうか。前節の let 式はなぜ右側のように構文解析されないのでしょうか。

```
let y   = a*b                let y   = a*b f
    f x = (x+y)/y    ==>         x   = (x+y)/y
in f c + f d                 in f c + f d
```

レイアウト(*layout*)と呼ばれる二次元構文を用いているからです。レイアウトは「カラム位置に揃えられた」宣言に依存しています。上の例で y と f が同一のカラムから始まっていることに注目してください。レイアウトの詳細は Haskell レポート(§1.5, §B.3)にまかせますが、レイアウトの使い方は直観的にわかります。ただ、覚えておいて欲しいことがあります。

キーワード where, let, of に続く次の文字は、where 節、let 式、case 式の開始カラム位置を決定します。開始カラムは、where 等を含むレイアウトの開始カラムより右になります。そうしなければ曖昧になってしまいます。レイアウトは開始カラム位置よりも左側で何かが開始されたときに終わります。

このルールはクラス宣言やインスタンス宣言の where にも適用されます。5 節を参照してください。後で議論する do キーワードもレイアウトを使います。宣言部はキーワードと同じ行でも次の行でも始めることができます。

レイアウトはグルーピング機構の簡略表記です。上の let は以下のものと同等です。

```
let { y = a*b ; f x = (x+y)/y } in f c + f d
```

{ } と ; に注目してください。この表記法は二つ以上の宣言を 1 行に収めたいときに便利です。以下も正しい式です。

```
let y = a*b;  z = a/b; f x = (x+y)/z
in f c + f d
```

5　型クラスと多重定義

Haskell の型システムについて説明します。既に述べた多相は**パラメータ**(*parametric*)多相とよばれるものです。この他に**アドホック**多相あるいは**多重定義**(*overloading*)と呼ばれる種類のものがあります。アドホック多相の例を幾つかあげると、

- 1、2 などのリテラルは固定長整数と任意長整数の両方で使われることが多い。
- + のような数値演算子は何種類もの数値上で定義されることが多い。
- 同値演算子 == は数値やその他多くの型の上で動作することが多い。

Haskell は型クラスによってアドホック多相(多重定義)を制御する方法を提供しています。多重定義されたものの振舞いは、型によって違いがあることに注意してください。振舞いが定義されていないか、エラーになる場合もあります。一方、パラメータ多相は、型そのものは重要ではないことに注意してください。 fringe 関数は木の葉の要素が何であるかには関係ありません。

同値性の例からはじめましょう。同値性を定義したい多くの型があります。同値性を定義したくない型もあります。関数が同値であるかどうかの判定は一般に困難です。同値性といっているのは値同値性のことです。対照的な概念としてポインタ同値性があります。Java 言語の == です。ポインタ同値性は参照透明性を持たないので関数型言語とは相性がよくありません。この点を明かにするために、リストの要素かどうかを検証する elem 関数の定義をみてみます。

```
x `elem`  []            = False
x `elem` (y:ys)         = x == y || (x `elem` ys)
```

[`elem` は関数 elem の中置形式です。== と || は同値演算子と論理和演算子です。]

elem の型は a->[a]->Bool です。そのためには == の型は a->a->Bool でなければなりません。同値性 == を定義したくない型もあると言ったにもかかわらずです。こんなことにも注目しています。== を全ての型の上で定義できたとしても、二つのリストを比較するのと二つの数を比較するのは全然違うことです。これらの仕事をさせるために == が多重定義できることを期待するわけです。

型クラス(*type class*)はこの問題を解決するのに都合のよい仕組です。型クラスを使うとクラスのインスタンスとして型を定義でき、クラスに付随する操作を多重定義することができます。同値演算子のクラスを定義してみましょう。

```
class Eq a where
  (==)                  :: a -> a -> Bool
```

Eq は定義するクラスの名前、== はクラスの操作です。この宣言は「型 a がクラス Eq のインスタンスであるのは、 == が存在しクラスで定義された型をもつ場合である」と読みます。

型 a がクラス Eq のインスタンスであるという制約を Eq a => と書きます。 Eq a => は型式ではなく型に対する制約を表現しているので**文脈**(*context*)と呼ばれます。文脈は型式の前に置きます。このクラス宣言の効果は次の型を == に割当ることになります。

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

これは「== はクラス Eq のインスタンスである型 a に対して a->a->Bool の型をもつ」と読みます。このシグネチャは elem の例で使われる == の型です。文脈による制約は elem の主型に波及します。

```
elem                    :: (Eq a) => a -> [a] -> Bool
```

これは「elem はクラス Eq のインスタンスである型 a に対して a->[a]->Bool の型をもつ」と読みます。 elem がすべての型の上で定義されているわけではなく、要素の同値性の比較方法がわかっている型の上で定義されていることを示しています。

インスタンス宣言(*instance declaration*)は Eq クラスのインスタンスになる型と == の実際の振舞いを指定します。たとえば次のようにします。

```
instance Eq Integer where
  x == y                =  x `integerEq` y
```

この宣言は「 Integer はクラス Eq のインスタンス型であり、== のメソッドを定義する」と読みます。== の定義は**メソッド**(*method*)と呼ばれます。 integerEq は整数の同値性を比較するプリミティブ関数です。関数定義と同様、メソッドの右辺はどのような式でもよいことになっています。この宣言により任意倍長整数の同値性を == を用いて比較することができます。同様に、浮動小数の比較を == を用いて行うことができます。

```
instance Eq Float where
  x == y                =  x `floatEq` y
```

Tree a のような再帰的な型でも同様に扱うことができます。この場合は Eq a => の文脈が必須です。

```
instance    Eq a => Eq (Tree a) where
  Leaf x         == Leaf y          =  x == y
  (Branch l1 r1) == (Branch l2 r2)  =  (l1==l2) && (r1==r2)
  _              == _               =  False
```

葉の同値性は要素の同値性を用いて比較されます。付加される制約は、 Tree a の同値性は既に知られている a の同値性を用いて実現するということです。文脈をインスタンス宣言から取り除くと静的型エラーが発生します。

プレリュードには型クラスが豊富にあります。実際の Eq クラスの定義はもう少し複雑です。

```
class  Eq a  where
  (==), (/=)            :: a -> a -> Bool
  x /= y                =  not (x == y)
```

Eq は同値性と非同値性の操作をもつクラスであり、非同値性操作 /= の**デフォルトメソッド**の使い方を例示しています。クラス宣言にデフォルトメソッドが定義されていると、インスタンス宣言でメソッドを定義しなくてもデフォルトメソッドが使われます。前に定義した 3 つの Eq クラスのインスタンスも完全に動作し、非同値性の定義(同値性の論理否定)が作られます。

クラス拡張(*class extension*)の記法もサポートしています。Eq クラスのすべての操作を**継承**(*inherit*)した Ord クラスを定義しましょう。Ord クラスは Eq クラスから継承した操作の他に、比較操作と min と max 関数をもつものとします。

```
class  (Eq a) => Ord a  where
  (<), (<=), (>=), (>)  :: a -> a -> Bool
  max, min              :: a -> a -> a
```

クラス宣言の文脈に注目してください。 Eq は Ord の**スーパークラス**(*super class*)、Ord は Eq の**サブクラス**(*subclass*)といいます。 Ord のインスタンスの型は Eq のインスタンスでなければなりません。

クラス包含の利点は文脈の簡略化にあります。 Eq と Ord のクラス操作を使用する関数の型式は、文脈 (Eq a, Ord a) ではなく (Ord a) を利用することができます。 Ord が Eq を包含しているからです。さらに、サブクラスのメソッドはスーパークラスのメソッドの存在を仮定することができます。例えば Ord の実際の宣言は (<) のデフォルトメソッドを非同値操作で定義しています。

```
x < y                   =  x <= y && x /= y
```

2.4.1 節で定義した quicksort の主型は次のようになります。

```
quicksort               ::  (Ord a) => [a] -> [a]
```

quicksort は順序付き型の値のリストでのみ操作できるということです。この文脈は quicksort の定義のなかの比較操作 < と >= から導かれます。

多重継承(*multiple inheritance*)も認められています。クラスが 2 つ以上のスーパークラスを持つこともあるからです。次の例のクラス C は Eq と Show の両方から操作を継承しています。

```
class (Eq a, Show a) => C a where ...
```

[クラスメソッドはトップレベルの宣言として扱われ、変数と同じ名前空間を共有します。クラスメソッド名は、変数名あるいは別のクラスのメソッドの両方を表示するのには使えません。]

これまで「1 階」の型を使ってきました。型構成子 Tree は a を引数にとり Tree a の型を返します。このような高階の型をクラス宣言で使用することができます。プレリュードにある Functor クラスを考えましょう。

```
class Functor f where
  fmap                  :: (a -> b) -> f a -> f b
```

fmap は map 関数を一般化したものです。型変数 f は f a と他の型に適用されています。型変数 f が Tree のように引数をとる型構成子に束縛されることを期待しています。 Functor の Tree インスタンスは次のようになります。

```
instance Functor Tree where
  fmap f (Leaf x)       = Leaf (f x)
  fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

インスタンス宣言は型構成子 Tree が Functor のインスタンスであると宣言しています。これは総称的「コンテナ」型を記述する能力のあることを示し、任意の木、リスト等の型の上で統一的に動作する関数を可能にします。

[型適用は関数適用と同じ書き方をします。型 T a b は (T a) b と構文解析されます。関数は (->) が型構成子です。 (->) f g と f -> g は同じです。同様に、[] a と [a] は同じです。タプルの型構成子(データ構成子でもあります)は (,)、(,,) などとなります。]

型システムは型付けできない不正な式を検出しますが、不正な型式に起因するエラーについてはどうでしょうか。 Tree Int Int はある種のエラーを起こすはずです。 Tree は一つの引数しかとらないからです。どのように不正な型式を検出するのでしょうか。答は型の正当性を確認する第 2 の型システムにあります。型は**類**(*kind*)と結びついています。類は型が正しく使用されることを確認するものです。

型式は 2 つの形式をとる**類**に分類されます。

- 記号 * は具体的なデータ対象に結びついた型の類を表す。値 v の型が t なら t の類は * です。
- $\cdot_1$ と $\cdot_2$ が類ならば $\cdot_1 \to \cdot_2$ は類 $\cdot_1$ の型を取り類 $\cdot_2$ の型を返す型の類である。

型構成子 Tree の類は *->* 、Tree Int の類は * です。Functor クラスのメンバーは類 *->* でなければならず、次のような宣言は類付けエラーを引き起こします。Integer の類は * だからです。

```
instance Functor Integer where ...
```

類はプログラムに直接あらわれることはありません。コンパイラは型検査の前に類を推論します。型シグネチャが類エラーを起こすとき以外は、類はプログラムの後ろに隠れています。類は非常にシンプルであり、類衝突が起こればコンパイラは類エラーメッセージを表示できなければなりません。類に関する詳細は(§4.1.1, §4.6)を参照してください。(以下、OOP との対比を省略)

6 再び、型について

ここでは、型宣言のより進んだ側面を詳しくみていくことにします。

6.1 Newtype 宣言

プログラミングの際によくやるのが, 表現は既存の型と同じだが、型システムでは別の型として識別されるような型を定義することです。 newtype 宣言は既存の型から新しい型をつくりだします。たとえば、次の宣言で Integer から自然数を表現することができます。

```
newtype Natural = MakeNatural Integer
```

この宣言は新しい型 Natural を生成し、構成子 MakeNatural は型変換 Integer -> Natural を行います。

```
toNatural               :: Integer -> Natural
toNatural x | x < 0     = error "Can't create negative naturals!"
            | otherwise = MakeNatural x

fromNatural             :: Natural -> Integer
fromNatural (MakeNatural i) = i
```

同じことが data 宣言でも可能です。しかし data 宣言では Natural の値の表現にオーバヘッドが生じます。宣言によって間接参照レベルの深さが遅延性の故に増すのです。 newtype, data, type 宣言の関係の詳しい議論はレポート(4.2.3)節を参照してください。

Natural を Num クラスに所属させるにはインスタンス宣言が必要です。元の型のインスタンスが新しい型に持ち越されることはありません。この宣言の目的は Natural を Num のインスタンスとすることなのです。 Natural が Integer の型シノニムとして宣言されていればインスタンスの導入はできません。

```
instance Num Natural where
    fromInteger         = toNatural
    x + y               = toNatural (fromNatural x + fromNatural y)
    x - y               = let r = fromNatural x - fromNatural y in
                              if r < 0 then error "Unnatural subtraction"
                                       else toNatural r
    x * y               = toNatural (fromNatural x * fromNatural y)
```

[newtype 宣言は data 宣言と同じ構文を用います。 newtype で定義された型は data 宣言で生成される型とほとんど同じなのです。]

6.2 フィールドラベル

データ型のフィールドは、位置またはフィールドラベルでアクセスすることができます。2 次元の点の型について考えましょう。

```
data Point = Pt Float Float
```

Point の構成要素は構成子 Pt の二つの引数です。次の pointx 関数は Pt の第一構成要素を参照しますが、このような関数を手でつくるのは面倒です。

```
pointx                  :: Point -> Float
pointx (Pt x _)         =  x
```

data 宣言の構成要素はフィールドラベル付で宣言することもできます。フィールドラベルは構成要素を名前で同定します。構成要素のフィールドラベルと型は { } で括って指定します。

```
data Point = Pt { pointx, pointy :: Float }
```

この宣言はフィールドラベル pointx, pointy を定義しています。データ型は前の Point と同じ型です。フィー ルドラベルは構成要素を取り出す**選択関数**(*selector function*) として使うことができます。

```
pointx                  ::   Point -> Float
pointy                  ::   Point -> Float
```

次はこれらの選択関数をつかった関数定義です。

```
absPoint                :: Point -> Float
absPoint p              =  sqrt (pointx p * pointx p + pointy p * pointy p)
```

フィールドラベルを使ったパターン照合も可能です。

```
absPoint (Pt { pointx = x, pointy = y }) = sqrt (x*x + y*y)
```

フィールドラベルは新しい値を構築するのにも使えます。 Pt {pointx=1, pointy=2} は (Pt 1 2) と同じです。フィールドラベル付の宣言は、位置によるフィールドアクセスを排除するものではありません。共に可能です。フィールドラベルを使って値を構築する場合、フィールドをいくつか省略することができます。省略されたフィールドは未定義です。

フィールド更新はフィールドを新しい値で埋めます。 p が Point のとき p {pointx=2} は p と同じ pointy をもちますが、pointx は 2 に置き換ります。これは破壊的更新ではありません。この更新は、指定されたフィールドを新しい値で埋めた新しいオブジェクトを生成します。(複数の構成子が夫々フィールドラベル付の構成要素をもつ場合の話を省略)

フィールドラベルが代数的データ型の基本性質を変えることはありません。フィールドラベルはデータ構造の構成要素をアクセスするのに名前を用いるために用意された便宜的な構文にすぎません。このおかげで、多くの構成要素をもつ構成子の扱いが簡単になります。フィー ルドラベルはフィールドの追加、削除を最小限の変更で可能にします。フィールドラベルの詳細と意味論については(レポート§4.2.1)を参照してください。

6.3 正格データ構造

一般のプログラミング言語は正格性をもっており、フィールドの要素はデータ構造に入れる前に値にまで簡約されます。 Haskell のデータ構造は**遅延性**をもちます。フィールドは必要になるまで評価されません。このおかげで、評価されるとエラーになったり停止しない要素をデータ構造に含めることができます。 遅延データ構造は Haskell の表現力を強化し、プログラミングスタイルの要になります。

遅延データの各フィールドは**サンク**(*thunk*)といわれる構造に包みこまれます。サンクはフィールド値をカプセル化したものです。値が必要にならない限り、サンクの中へ入ることはありません。エラー(⊥)を含むサンクがあっても、データ構造の他の要素には影響しません。たとえば、タプル ('a',⊥) は完全に正しい値です。'a' はタプルのもう一方の要素に関係なく使うことができます。

サンクにはオーバヘッドがついてまわります。サンクを構成し評価するのにも時間がかかり、ヒープの領域を必要とします。サンクの評価に必要な別の構造を確保するため、ガーベッジコレクタを起動することもあります。こうしたオーバヘッドを回避するためには data 宣言で**正格性フラグ**を使います。

複素数のライブラリが定義している Complex 型は次のようになっています。

```
data RealFloat a => Complex a = !a :+ !a
```

! が正格性フラグです。複素数の実数部と虚数部は正格性をもつようにマークされています。 ! でマークされたフィールドは構造を生成するときに直ちに評価され、サンクには入れません。複素数のコンパクトな表現ですが、 1 :+ ⊥ のように未定義の構成要素があった場合、全体が未定義(⊥)になってしま

う代償を払った上でのことです。実際には、部分的にしか定義されていない複素数が必要になることはないので、正格性フラグを使用し、効率よい表現を達成するのは意味のあることです。

正格性フラグを用いるほうがよい状況がいくつかあります。

- プログラム実行中に必ず評価されることが確定している構成要素
- 簡単に評価でき、絶対にエラーを起こさないことが確定している構成要素
- 部分的に未定義の値であると意味のない型

正格性フラグ ！ は data 宣言でのみ使われます。型シグネチャー、関数引数に正格性フラグをつける方法はありません。しかし、 seq あるいは !$ という関数を使用すると同じ効果が得られます。詳細についてはレポート(§4.2.1)を参照してください。

遅延性は Haskell の基本的な特徴です。正格性フラグの正確なガイドラインを示すのは難しく、注意して使用しなければなりません。正格性フラグをつけることは無限ループの発見を困難にし、予期せぬ結果をまねきかねません。

7　入出力

Haskell の I/O システムは純粋に関数的でありながら、伝統的なプログラミング言語がもつ表現力をすべて兼備えています。命令型言語では、プログラムは世界の状態を確認、変更する**アクション**を通じて進行します。アクションには、変数の読み書き、ファイルの読み書き、ウィンドウのオープンなどがあります。アクションは Haskell の一部ですが、言語のコアの部分からすっきりと切離されています。

Haskell の I/O システムは**モナド**(*monad*)を基礎にして築かれています。しかし、I/O システムを使うのにモナド理論を必要とすることはありません。モナド理論はたまたま I/O に適合した概念上の構造です。算術演算を実行するのに群論を理解する必要がないのと同じで、Haskell の I/O を実行するのにモナド理論が必要になることはありません。モナドの用語を避け、I/O システムの使い方に集中しましょう。I/O モナドを単に抽象データ型だと考えておくのが良いでしょう。

アクションは呼出すのではなく定義するものです。定義を評価しても実際のアクションは起こりません。アクションは式の評価の外側で起こることです。

アクションはプリミティブとして定義されるアトミックなものであり、一連のアクションの合成でもあります。他の言語では ； を使って文を一列にならべるのと同様の複合アクションを構成します。モナドには複合アクションを構成するためのプリミティブがあり、アクションを貼り付ける糊のような役目をします。

7.1　I/O の基本演算

I/O アクションは値を返し、返り値には IO というタグが付けられます。アクションの値を他の値と区別するためです。関数 getChar と putChar の型を示します。

```
getChar                 ::  IO Char
putChar                 ::  Char -> IO ()
```

型 IO Char は getChar が文字を返す動作をすることを示しています。putChar は文字を引数に取りますが、意味のある値は返しません。意味のある値を返さないアクションにはユニット型 () を用います。

アクションは getChar >>= putChar >> putChar '¥n' のようにモナド演算子 >>= と >> を使って順序付けをします。 >>= はアクションの「タグ」無しの値を次のアクションに渡し、 >> はアクションの値を無視します。

モナド演算子の代わりに do 記法を使うことができます。 do 記法は順序制御の演算子を構文の後に隠し、伝統的な言語に近くなります。キーワード do は順に実行する文のならびを導入します。文はアクションであり、<- によりアクションの結果に束縛されるパターンです。 do 記法は let や where と同様、レイアウトを使います。次は文字を読み込み印字するプログラムです。

```
main                    :: IO ()
main                    =  do c <- getChar
                              putChar c
```

main は C の main 関数と同様、 Haskell プログラムのエントリポイントになります。 main は IO 型でなければなりません。普通は IO () です。 main は Main モジュールの中でだけ特別な意味をもちます。上のプログラムは順に 2 つのアクションを実行します。最初に文字を読み込み結果を変数 c へ束縛します。次にその文字を印字します。 <- で定義した変数は後に続く文のなかでのみ有効です。

do を用いてアクションを起動し結果を知ることができますが、一連のアクションから結果を返すにはどうすれば良いのでしょうか。読み込んだ文字が 'y' であれば True を返す ready 関数を考えます。

```
ready                   :: IO Bool
ready                   =  do c <- getChar
                              c == 'y'  -- ダメ!!!
```

この関数は do の最後の式がアクションではなく真理値なので動作しません。何もしないが結果としてこの真理値を返すアクションを生成しなければなりません。 return 関数がこれを行います。

```
return                  ::   a -> IO a
```

ready の最後の行は return (c == 'y') でなくてはなりません。

```
ready                   =  do c <- getChar
                              return (c == 'y')
```

もっと複雑な getLine 関数を見てみましょう。

```
getLine     :: IO String
getLine     =  do c <- getChar
                  if c == '¥n'
                       then return ""
                       else do l <- getLine
                               return (c:l)
```

else 節にある do に注目してください。 2 つの do は、それぞれ文の列を構成します。if のような言語要素がアクション列を導入するには新たな do を使います。

return 関数は一般の値を I/O アクションの領域に入れます。逆はどうでしょう。I/O アクションを式のなかで起動できるでしょうか。 x + print y と式の中に入れ式の評価の結果として印字することができるでしょうか。答は「できない」です。純粋関数型のコードから命令型の世界にしのび込むことはできません。命令型の世界の値には、それと分かるタグがついている必要があります。

f :: Int -> Int -> Int のような関数は絶対に I/O を行うことはできません。タグ IO が返り値に現れていないからです。デバッグのときにコード全体に print 文を埋めこむことの多いプログラマにとっては悩みの種でしょう。この問題に対処するために安全ではない関数が用意されていますが、これらは上級プログラマにとっておいた方がよいものです。Trace などデバッグ用のパッケージは安全な手法では禁じ手になっている関数を気前よく使っています。

7.2 アクションを使ったプログラミング

I/O アクションは普通の Haskell の値です。他の値のように関数に渡されることもあり、構造のなかに入れることもできます。アクションのリストを考えてみましょう。

```
todoList :: [IO ()]
todoList = [putChar 'a',
            do putChar 'b'
               putChar 'c',
            do c <- getChar
               putChar c]
```

このリストはアクションを保持しているだけで起動することはありません。これらのアクションを起動するには次の sequence_ 関数が必要です。

```
sequence_        :: [IO ()] -> IO ()
sequence_ []     =  return ()
sequence_ (a:as) =  do a
                       sequence_ as
```

do 記法はモナド演算子 >>=, >> に展開できます。 do x ; y は x >> y に展開します(9.1 節を参照)。sequence_ 関数の右再帰は foldr 関数で捉えることができます。 foldr は右結合の中置演算子 #, 初期値 z, リストを引数に取り、次のようにリストを展開します。

```
foldr # z [x1, x2, ..., xn]  =>  x1 # x2 # ... xn # z  =  x1 # (x2 # ( ... (xn # z) ... ) )
```

sequence_ のさらに良い定義は次のものです。

```
sequence_        :: [IO ()] -> IO ()
sequence_        =  foldr (>>) (return ())
```

sequence_ 関数は putChar から putStr を構築するのにも使えます。

```
putStr                  :: String -> IO ()
putStr s                =  sequence_ (map putChar s)
```

Haskell と命令型言語の違いがこの putStr に見てとれます。命令型言語では putChar を文字列上にマッピングするだけで印字が可能です。しかし、Haskell では文字列の各文字に対応する putChar アクションのリストを生成するだけです。 sequence_ の foldr は >> 演算を使って個別のアクションを起動します。一連のアクションの最後には、何もしないアクション return () が必要です。

プレリュードやライブラリには、アクションの順序付けに便利な関数がたくさんあります。関数の多くは任意のモナド用に汎用化されています。Monad m => という文脈を含む関数はどれも IO 型で使えます。

7.3 例外処理

getChar がファイルの最後に当たったらどうなるでしょう。end of file の例外は I/O モナド内で捕捉可能、処理可能なものです。例外は standard ML の入出力ハンドリング機構と同様に取り扱います。特別な構文や意味論を用いることはありません。

I/O エラーは型 IOError としてコード化されています。 IOError は一種の抽象型であり、ユーザが利用できる構成子はありません。抽象型とすることで新しいエラーをシステムに導入することができます。IOError 型の値にたいして幾つかの述語が定義されています。

```
isEOFError      :: IOError -> Bool
```

isEOFError 関数はエラーがファイル終端で引き起こされたかどうかを判定します。関数はプレリュードとは別の IO モジュールのなかで定義されており、インポートする必要があります。

例外ハンドラは IOError -> IO a の型になり、catch 関数を使ってアクションと関連づけます。

```
catch                   :: IO a -> (IOError -> IO a) -> IO a
```

catch 関数はアクションと例外ハンドラを引数に取ります。アクションが成功すれば例外ハンドラを起動せずに結果だけを返します。エラーが起これば IOError の値を例外ハンドラに渡し、例外ハンドラのアクションを起動します。エラーがあると改行を返す getChar' を例示します。

```
getChar'                :: IO Char
getChar'                =  getChar `catch` (¥_ -> return '¥n')
```

ファイル終端になった場合を扱うにはエラーの種類を確かめなければなりません。

```
getChar'                :: IO Char
getChar'                =  getChar `catch` eofHandler where
          eofHandler e = if isEofError e then return '¥n' else ioError e
```

ioError 関数は次の例外ハンドラに例外を投げます。 ioError は、制御を例外ハンドラに移すという点を除けば、return と類似しています。

```
ioError                 :: IOError -> IO a
```

catch の入れ子の呼び出しも可能です。入れ子の catch 呼び出しは入れ子の例外ハンドラを生成します。この例を getChar' を使って getLine を再定義することで示しましょう。

```
getLine'        :: IO String
getLine'        = catch getLine'' (¥err -> return ("Error: " ++ show err)) where
      getLine'' = do c <- getChar'
                     if c == '¥n' then return ""
                                  else do l <- getLine'
                                          return (c:l)
```

getChar' の例外ハンドラがファイル終端を捕捉し、getLine' の例外ハンドラがエラー文字列を返します。プログラムの最上位には例外を印字して停止する例外ハンドラが用意されています。

7.4 ファイル、チャネル、ハンドル

他の言語の I/O と同じ機構が Haskell にも備わっています。ファイルをオープンすると**ハンドル**が生成され、ハンドルを使って I/O のやりとりをします。ハンドルをクローズすると関連するファイルがクローズされます。ハンドルの型は Handle です。

```
type FilePath         =  String  -- path names in the file system
openFile              :: FilePath -> IOMode -> IO Handle
hClose                :: Handle -> IO ()
data IOMode           =  ReadMode | WriteMode | AppendMode | ReadWriteMode
```

ハンドルはチャネルにも関連付けられます。チャネルはファイルとは結びつかないポートです。 stdin (標準入力)、 stdout (標準出力)、stderr (標準エラー)のチャネルハンドルが定義済になっています。hGetChar や hPutChar の文字単位の I/O は、ハンドルを引数にとります。getChar は次のように定義されています。

```
getChar               = hGetChar stdin
```

さらに、ファイルやチャネルの内容全体の文字列を返す関数があります。

```
getContents           :: Handle -> IO String
```

getContents はチャネルやファイルの全内容をすべて読み込むのでメモリが足りなくなりそうに見えますが、それは違います。鍵となる要点は getContents は文字の「遅延」リストを返すことです。リストの要素は「必要になってはじめて読み込まれる」のです。

次の例はファイルの copy コマンドのプログラムです。

```
main = do arg1:arg2:[] <- getArgs
          fromHandle   <- catchAndOpenFile arg1 ReadMode
          toHandle     <- catchAndOpenFile arg2 WriteMode
          contents     <- hGetContents fromHandle
          hPutStr toHandle contents
          hClose  toHandle
          putStrLn "Done."

catchAndOpenFile :: FilePath -> IOMode -> IO Handle
catchAndOpenFile name mode = catch (openFile name mode) (¥e -> do putStr ("Cannot open " ++ name)
                                                                   ioError e)
```

7.5 Haskell と命令型プログラミング

I/O のプログラミングスタイルは、命令型のプログラミングとさして変わらないと思えます。

```
getLine = do c <- getChar
             if c == '¥n' then return ""
                          else do l <- getLine
                                  return (c:l)
```

```
function getLine() {
    c := getChar();
    if c == `¥n` then return ""
                 else {l := getLine();
                       return c:l}
}
```

Haskell は単に命令型の車輪を再発明しただけなのでしょうか。ある意味ではそのとおりです。I/O モナドは Haskell のなかに小さな命令型のサブ言語を構成しています。それゆえ、プログラムの I/O の構成要素は、従来の命令型のコードにそっくりになるのです。

しかし、一つ大きな違いがあります。 I/O を扱うのに特別な意味論を必要としないことです。 Haskell の等式論証の仕組はなんら損われていません。コードの命令型のフィーリングは、 Haskell の関数的な側面を損なうものではありません。経験を積んだ関数プログラマは、プログラムの命令型の構成要素を最小限にし、トップレベルでの順序付けを最小にし、そこでだけ I/O モナドを使用するようにできるにちがいありません。モナドはプログラムの関数的な構成要素と命令的な構成要素を綺麗に分けます。一方、関数的なサブセットをもつ命令型言語では、関数的な世界と命令的な世界をへだてるものがハッキリとは定義されていません。

8 標準クラス

Haskell の定義済みの標準型クラスを紹介します。一部のクラスは簡単に紹介しました。これらのクラスは Haskell の標準ライブラリになっており、Haskell ライブラリレポートに記述されています。

8.1 順序クラス

Ord クラスは既に議論しました。 プレリュードにある Ord クラスの定義は前に定義したものよりも少し複雑です。特に、compare メソッドは以下のようになっています。

```
data Ordering           =  EQ | LT | GT
compare                 :: Ord a => a -> a -> Ordering
```

compare メソッドがあれば、このクラスの他のメソッドはすべてデフォルトメソッドで定義できます。Ord のインスタンスを生成するには、この方法が一番すぐれています。

8.2 Enum クラス

Enum クラスには数列表現の底流にある一連の演算があります。数列式 [1,3..] は enumFromThen 1 3 を表わしています。Enum のインスタンス型であれば、数列式を用いてリストを生成できることがわかります。 Enum のインスタンスには Char 型も含まれており、['a'..'z'] はアルファベット順の小文字のリストを表わします。 Color のようなユーザ定義の列挙型を Enum のインスタンスとして宣言し、数列式を使用することができます。

```
[Red .. Violet] => [Red, Green, Blue, Indigo, Violet]
```

このような列は算術的である点を注意します。各値の増分は数値であろうとなかろうと一定です。Enum のほとんどのインスタンスは、固定長整数(Int)に写像することが可能です。fromEnum や toEnum 関数が用意されており、Int と Enum インスタンスの間での変換をおこないます。

8.3 Show クラスと Read クラス

Show は文字列への変換が可能な型のクラスです。 Read は文字列を値に変換できる型のクラスです。二つのクラスは PreludeText モジュールにあります。Show クラスの基本関数は show です。

```
show                    :: (Show a) => a -> String
```

show は対応する型の値をとり、その表現文字列を返します。たとえば show (2+2) は "4" を返します。

しかし、もっと複雑な文字列表現が必要になる場合があります。次のような文字列を連結するのは能率的ではありません。(++) は左側の引数の長さの線形で効く計算量をもつからです。

```
"The sum of " ++ show x ++ " and " ++ show y ++ " is " ++ show (x+y) ++ "."
```

二分木の文字列表現を考えてみましょう。部分木を表示する印 < ... > を付けたり、左の枝と右の枝を分ける印 | を付けたりします。

```
showTree               :: (Show a) => Tree a -> String
showTree (Leaf x)      =  show x
showTree (Branch l r)  =  "<" ++ showTree l ++ "|" ++ showTree r ++ ">"
```

線形の複雑さを持つ shows 関数を定義します。 shows は印字可能な値と文字列積算子を引数にとり、値の文字列と積算子を連結した文字列を返します。

```
shows                  :: (Show a) => a -> String -> String
```

show は shows に空積算子を与えたものであり、 show のデフォルト定義になっています。

```
show x                 =  shows x ""
```

shows と文字列積算子を使い、線形のオーダになる showsTree を定義をすることができます。

```
showsTree              :: (Show a) => Tree a -> String -> String
showsTree (Leaf x) s   =  shows x s
showsTree (Branch l r) s=  '<' : showsTree l ('|' : showsTree r ('>' : s))
```

showsTree は Branch のような長い文字列を構築すると右端に括弧が蓄積する欠点があります。これを関数合成により改良するために、文字列積算子を showsTree から表示関数の引数に移した型シノニム ShowS を定義します。

```
type ShowS             =  String -> String
```

ShowS は表示関数の型です。表示関数は文字列積算子を引数に取り、表示関数を文字列積算子に積算した文字列を返します。 shows x 、('<':) などが ShowS のプリミティブな表示関数です。

```
showsTree              :: (Show a) => Tree a -> ShowS
showsTree (Leaf x)     =  shows x
showsTree (Branch l r) =  ('<':) . showsTree l . ('|':) . showsTree r . ('>':)
```

このプログラム変換は、オブジェクトレベルの表現から関数レベルの表現への変換です。 showsTree の型付けは Tree から表示関数への写像ということができます。

逆の問題を考えます。基本のアイディアは構文解析子です。型 a の構文解析子は、入力文字列から読み込んだ型 a の値と残りの文字列の組のリスト [(a, String)] を返す関数です[9]。プレリュードにはこの型シノニムが備えられています。

```
type ReadS a           =  String -> [(a,String)]
```

構文解析子は普通は単一要素のリストを返します。構文解析が失敗すれば空リスト、 2 つ以上の構文解析が可能(曖昧な構文)ならば 2 つ以上の組を含むリストになります。標準関数の reads は Read のあ

らゆるインスタンスに対応する構文解析関数です。

```
reads                   :: (Read a) => ReadS a
```

reads を使って showsTree で作った二分木の文字列を構文解析する関数を定義することができます。リストの内包表記は、構文解析関数を構築するのに便利なイディオムです。モナドや構文解析結合子を使ったもっとエレガントな方法もあり、標準の構文解析ライブラリの一部になっています。

```
readsTree          :: (Read a) => ReadS (Tree a)
readsTree ('<':s) =  [ (Branch l r,u) | (l,'|':t) <- readsTree s,
                                        (r,'>':u) <- readsTree t ]
readsTree s       =  [ (Leaf x,t)     | (x,t)     <- reads s ]
```

関数定義を詳細に見ていきましょう。考慮すべき状況が二つあります。構文解析の対象となる文字列の最初の文字が '<' の場合は枝の表現にちがいありません。そうでない場合は葉の表現です。枝の場合、'<' に続く入力文字列を s とすると、可能な構文解析は Branch l r であり、残りの文字列 u は以下の条件に従います。

1. l は文字列 s から構文解析され、 l に続く文字列を '|':t とする。
2. r は文字列 t から構文解析され、 r に続く文字列を '>':u とする。

リスト内包表記のパターン照合に注目してください。生成部 (l,'|':t) は s の構文解析結果のリストから引出され、条件 1 を表現しています。条件 2 は次の生成部によって表現されています。葉の入力文字列に対する構文解析は、要素 x を構文解析し結果に構成子 Leaf を適用します。

Integer の Read と Show インスタンスがあって、期待通りの振舞いをする reads が備わっているとしましょう。以下のように振舞うはずです。

```
reads "5 golden rings" => [(5, " golden rings")] :: [(Integer,String)]
```

これが理解できれば次の評価を確認できるはずです。

```
readsTree "<1|<2|3>>" => [(Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)), "")]
readsTree "<1|2"      => []
```

readsTree には欠点があります。木の表現文字列の前や間に空白を許さないこと、区切の記号の構文解析法が葉の場合と部分木の場合でまったく違うことです。この統一性の無さが関数の定義を読みにくくしています。問題はプレリュードにある字句解析器 lex を使って解決することができます。

```
lex                     :: ReadS String
```

lex の型 String -> [(String,String)] は、入力文字列から [(入力字句, 残りの入力文字列)] を返す関数であることを示します。例えば、lex は以下のように振舞います。

```
lex "  <22|23> "  =>  [("<","22|23> ")]
lex "22|23> "     =>  [("22","|23> ")]
lex "|23> "       =>  [("|","23> ")]
```

lex は Haskell の字句規則に従います。ただし、空白はスキップしますがコメントはスキップしません。空白だけの入力文字列には [("","")] を返し、不正な字句で始まっている場合は [] を返します。

lex を使うと木の構文解析はこんなふうになります。

```
readsTree               :: (Read a) => ReadS (Tree a)
readsTree s             =  [(Branch l r, x) | ("<", t) <- lex s,
                                              (l,   u) <- readsTree t,
                                              ("|", v) <- lex u,
                                              (r,   w) <- readsTree v,
                                              (">", x) <- lex w         ]
                           ++
                           [(Leaf x, t)     | (x,   t) <- reads s       ]
```

readsTree と showsTree は Read と Show のメソッドをほぼ満しています。readsTree と showsTree を使って、 Read と Show のインスタンス (Read a) => Tree a と (Show a) => Tree a を宣言してみます。プレリュードにある総称的多重定義関数を利用できるようになり、木を含む多くの型の構文解析や表示ができるようになります。Show と Read の Tree のインスタンスは以下のようになります。

```
instance Show a => Show (Tree a) where
    showsPrec _ x = showsTree x

instance Read a => Read (Tree a) where
    readsPrec _ s = readsTree s
```

showsPrec と readsPrec は shows と reads に優先度パラメータを追加したものです。優先度は式に中置演算子が含まれているとき、正しく括弧付けするのに使用されます。 Tree 型は優先度を無視します。

Show のインスタンスは showTree を使って次のようにも定義できますが、これは ShowS バージョンよりも効率の面で劣ります。

```
instance Show a => Show (Tree a) where
   show t = showTree t
```

Show クラスは showsPrec と show の両方にデフォルトメソッドを定義しており、インスタンス宣言ではどちらかを定義すればよくなっています。デフォルトメソッドは相互再帰的になっているので、インスタンス宣言でどちらも定義しないと、呼び出しはループを引き起こします。 Num クラスにも「相互ロック性のデフォルトメソッド」があります。

関数合成 read . show (恒等関数になるはず)を木に適用して Read と Show のインスタンスをテストすることができます。 read は reads を特殊化したものです。

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

ただし、入力の構文解析結果が一通りでない場合、あるいは、入力が a 型の値に対して 2 つ以上の表現を含むような場合、あるいは、コメントや空白を含む場合はこのテストは失敗します。

8.4 インスタンスの導出

木の Eq インスタンスについて思いだしてください。葉の要素の型が同値型であることを要求しています。 2 つの葉は同値の要素をもつ場合且つその場合にかぎり同値であり、2 つの枝は左の部分木と右の部分木が等しい場合且つその場合にかぎり等しいわけです。

```
instance  (Eq a) => Eq (Tree a)  where
    (Leaf x)     == (Leaf y)        =  x == y
    (Branch l r) == (Branch l' r')  =  l == l' && r == r'
    _            == _               =  False
```

新しい型にインスタンス宣言を書くのは単純ですが面倒です。面倒な作業を繰り返す必要はありません。data 宣言で Eq インスタンスを自動的に導出することができます。

```
data  Tree a            =  Leaf a | Branch (Tree a) (Tree a)  deriving Eq
```

deriving 節は自動的に Eq インスタンスを導出します。同様に Ord、Enum、Ix、 Read、Show の各インスタンスを導出することができます。 deriving (Read, Show) のように 2 つ以上のクラスから導出することもできます。ほとんどの Eq と Ord のインスタンスは deriving 節を使って導出されています。

Tree に対して自動的に導出される Ord インスタンスは**辞書順**になっています。 data 宣言の構成子は出現順にならべられ、構成子の引数は左から右へと比較されていきます。

```
instance  (Ord a) => Ord (Tree a)  where
    (Leaf x)     <= (Leaf y)        =  x <= y
    (Leaf _)     <= (Branch _ _)    =  True
    (Branch _ _) <= (Leaf _)        =  False
    (Branch l r) <= (Branch l' r')  =  l == l' && r <= r' || l <= l'
```

リスト型が 2 つの構成子をもつ型であることを思いだしてください。この定義は次のようになります。

```
data [a]        = [] | a : [a] deriving (Eq, Ord)     -- pseudo-code
```

リストに対して導出された Eq と Ord のインスタンスは普通のものです。特に文字リストとしての文字列は Char 型の順序にならべられます。"cat" < "catalog" のように文字列の先頭部分はその文字列よりも小さいと判定されます。リストの Text インスタンスは導出できません。

列挙型は導出された Enum インスタンスをもちます。順序は data 宣言での構成子の出現順になります。Day 型の導出されたインスタンスを用いた例を示します。

```
data Day = Sunday | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday deriving Enum

[Wednesday .. Friday] =>[Wednesday, Thursday, Friday]
[Monday, Wednesday ..] =>[Monday, Wednesday, Friday]
```

構成要素が Read, Show のインスタンス型なら、その型に対して Read, Show のインスタンスを導出することが可能です。標準の型はほとんど Read, Show のインスタンスです。関数型 (->) は Show インスタンスですが、Read インスタンスではありません。導出された Show インスタンスのテキストは定数式と整合しています。

型 Day に Show の deriving 節に加えると次の結果を得ることができます。

```
show [Monday .. Wednesday] => "[Monday,Tuesday,Wednesday]"
```

9 モナド

はじめて Haskell にふれる人にとって**モナド**(*monad*)の概念は謎です。Haskell ではモナドに頻繁に出会います。 I/O システムはモナドによって構成されており、モナドのための構文(do 式)も用意されています。

このセクションは他に比べると「易しい」というわけにいかないでしょう。ここでは、モナドを含む言語の特徴を示すだけではなく、モナドがなぜ重要なツールであり、どのように使うかを明かにしようと思います。誰もが分かるモナドの説明というものはありません。モナドを使ったプログラミング入門の良い文献として Wadler の Monads for Functional Programming [10] をあげておきます。

9.1 モナドのクラス

プレリュードには Functor と Monad というモナドを定義したクラスがあります。モナドクラスは圏論(category theory)におけるモナドの構成を基礎としています。圏論の用語はモナドのクラスや演算に名前を残していますが、モナドクラスの使い方を直観的に理解するのに抽象数学に通じている必要はありません。

モナドは多相型の上に構成され、モナドクラスの型はインスタンス宣言を用いて導入します。 deriving 節によって導出することはできません。プレリュードには IO, リスト [], Maybe というモナドクラスの型が定義されています。

モナドはモナド演算に対する一連の法則に支配されています。法則の概念は何もモナドに限られてはいません。法則に支配されている演算は他にもあります。例えば x /= y と not (x == y) はあらゆる型の値に対して同じでなければなりません。モナドの法則は、モナドクラスのインスタンスであればどれも満すべき法則です。モナドの法則を検証しながらどのようにモナドを使うかの感覚をおぼえましょう。

Functor クラスは 5 章で議論しており、演算 fmap のみを定義しています。 fmap は型構成子 f を f a と他の型に適用し、関数 a -> b をモナド関数 f a -> f b に持ち上げます。

```
class Functor f where
  fmap                  :: (a -> b) -> f a -> f b
```

しかし、 fmap id は id 自身です。合成された関数の持ち上げは、持ち上げられた関数の合成です。次の法則が Functor クラスの fmap に適用されます。これらの法則は、fmap によってコンテナの形は変更されないこと、コンテナの中身がならび変えられることはないことを保証します。

```
fmap id      = id
fmap (f . g) = fmap f . fmap g
```

Monad クラスは 2 つの基本演算 >>= と return を定義しています。

```
infixl 1  >>, >>=
class  Monad m  where
    (>>=)            :: m a -> (a -> m b) -> m b
    (>>)             :: m a -> m b -> m b
    return           :: a -> m a
    fail             :: String -> m a
    m >> k           =  m >>= ¥_ -> k  -- default method
```

演算 >> と >>= は二つのモナド値を合成し、return 演算は値をモナド(コンテナ)の中へ投入します。>>= の演算 ma >>= ¥v -> mb を考えます。 ma は型 a の値を含むモナド値、ラムダ式は型 a の値 v をモナド値 mb に渡す関数です。>>= は ma と mb を合成して型 b の値を含むモナド値 mb にします。>> 演算は第二引数のモナド値が第一引数のモナドがもつ値を必要としない場合に用いられます。

>>= の意味はモナドに依存します。IO モナドの x >>= y は二つのアクションを順に実行し、 x のアクションの結果が y のアクションに渡されます。リストモナドの >>= は、ゼロ個以上の値のリストを一つの計算から次の計算へ渡すということで理解できます。

do 構文を使うとモナド演算の鎖を簡単に表記できます。 do 構文からモナド演算への変換は、以下のルールで捉えることができます。

```
do e1 ; e2       = e1 >> e2
do p <- e1; e2   = e1 >>= ¥p -> e2
```

do の p<-e のパターンは反駁可能です。パターン照合が失敗すると fail 関数が呼ばれます。 fail は error を呼ぶか(IO モナド)、[] を返します(リストモナド)。次はもっと詳細な変換です。"s" は do 文の位置を同定するエラーメッセージで使われる文字列です。

```
do p <- e1; e2   = e1 >>= (¥v -> case v of p -> e2
                                           _ -> fail "s")
```

>>= と return を支配している法則は以下の通りです。

```
return a >>= k  = k a                          xs >>= return . f        = fmap f xs
m >>= return    = m                            m >>= (¥x -> k x >>= h)  = (m >>= k) >>= h
```

クラス MonadZero は zero 要素と plus 演算をもつモナドに対して定義されています。

```
class  (Monad m) => MonadZero m  where
    mzero             :: m a
    mplus             :: m a -> m a -> m a
```

IO モナドは zero 要素を持たないので、このクラスには属しません。リストモナドの zero 値は [], mplus 演算はリストの連結演算です。 zero 要素と mplus 演算は次の法則に従います。

```
m >>= ¥x -> mzero = mzero                      m `mplus` mzero   = m
mzero >>= m       = mzero                      mzero `mplus` m   = m
```

9.2 組み込みモナド

モナド演算と演算を支配する法則があると何を構築できるのでしょうか。リストモナドからはじめます。リストのモナド演算のシグネチャと定義は以下のようになります。

```
(>>=)                 :: [a] -> (a -> [b]) -> [b]
return                ::  a  -> [a]
m >>= f               = concat (map f m)
return x              = [x]
```

>>= 演算は、リスト [a] と a -> [b] の関数 f が与えられたとき、リストの各要素に f を適用して生成されたすべての [b] を一つのリストに連結します。 return は単一要素のリストを生成します。

リストの内包表記はモナド演算を使って表すことができます。以下の 3 つの式はすべて同じ式を表現したものです。最後のモナド式は do 式に前記の変換を施したものです。

```
[(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]

do x <- [1,2,3]
   y <- [1,2,3]
   True <- return (x /= y)
   return (x,y)

[1,2,3] >>= (¥x->[1,2,3] >>= (¥y->return (x/=y) >>= (¥r->case r of True->return (x,y)
                                                                   _   ->fail """)))
```

x <- [1,2,3] は残りのモナド計算に渡される値の集合を生成し、残りのモナド計算を 3 回起動します。リストの各要素に 1 回づつです。return (x,y) は、取り巻く束縛の可能な全ての組み合わせについて評価されます。リストモナドは多重値の引数(リスト)をとる関数を記述しているものと考えることができます。

次の関数を考えてみましょう。mvLift2 は、2 引数の関数を多重値(リストモナド)関数に持ち上げます。持ち上げられた関数は 2 つのリストのすべての要素の組み合わせに対して一つの値を返します。

```
mvLift2               :: (a -> b -> c) -> [a] -> [b] -> [c]
mvLift2 f x y         =  do x' <- x
                            y' <- y
                            return (f x' y')
```

この関数はモナドライブラリの LiftM2 の特別版です。 mvLift2 は、関数をリストモナドの外からリストモナドの世界へ持ち上げ、多重値計算を行わせるものと考えることができます。

```
mvLift2 (+) [1,3] [10,20,30]    => [11,21,31,13,23,33]
mvLift2 (¥a b->[a,b]) "ab" "cd" => ["ac","ad","bc","bd"]
mvLift2 (*) [1,2,4] []          => []
```

Maybe モナドはリストモナドに類似してり、 Nothing は []、Just x は [x] のように機能します。

9.3 モナドを使う

モナド演算子とその法則を説明するだけでは、モナドが何の役に立つのか分かりません。モナドがもたらす効用は**モジュール化**(*modularity*)です。演算をモナド的に定義して計算機を隠蔽し、新しい特徴をモナドに結びつけることができます。 Wadler の論文 [10] は、モナドを使ってモジュール化プログラムを構築する秀逸な例です。この論文から例をとるところから始めます。はじめに状態モナドを定義します。次の data 宣言は、状態モナドの型 SM が状態の型 S を暗黙に運ぶ計算であることを表します。

```
data SM a = SM (S -> (a,S))  -- The monadic type
```

SM の型は、状態 S をとり(型 a の値, 更新後の状態 S)の組を返す関数で構成されています。 SM t は型 t の値の計算を定義する一方、型 S の状態と読み／書きの相互作用をします。

インスタンス宣言は直列化演算子 >>= と return を定義します。

```
instance Monad SM where
  -- defines state propagation
  SM c1 >>= fc2         =  SM (¥s0 -> let { (r,s1) = c1 s0 ; SM c2 = fc2 r } in c2 s1)
  return k              =  SM (¥s -> (k,s))
```

SM で表示される計算は、初期状態 s0 を c1 へ渡し、c1 s0 で得られた値 r を次の計算 c2 を返す関数に渡します。 c1 後の状態 s1 を c2 に渡し、計算 c2 s1 の結果を全体の結果とします。return は何もしない空計算です

モナドプリミティブ(*monadic primitives*)も必要になります。プリミティブはモナドの内部的な仕事のみを取り扱います。readSM と updateSM が状態モナドの内部構造に依存していることに注目してください。SM 型を変更するとプリミティブも変更しなくてはなりません。

```
 -- monadic primitives
readSM                  :: SM S
readSM                  =  SM (¥s -> (s,s))

updateSM                :: (S -> S) -> SM ()  -- alters the state
updateSM f              =  SM (¥s -> ((), f s))
```

readSM は状態を観察できるようにモナドの外へ持ち出します。updateSM はモナドの状態を変更するのを許します。writeSM をプリミティブとすることも可能ですが、状態を扱う場合には update の方がより自然な場合が多いようです。

最後に必要な関数は、状態モナドの計算を駆動する関数 runSM です。この関数は初期状態と計算をとり、返り値と最終状態の両方を返す関数です。

```
-- run a computation in the SM monad
runSM                   :: S -> SM a -> (a,S)
runSM s0 (SM c)         =  c s0
```

状態モナドの意図するところは、計算を一連のステップ(SM a 型の関数)として定義し、>>= と return を使って直列化することです。各ステップは状態と readSM, updateSM を通じてやりとりするか、状態を無視します。状態の使用／不使用は隠蔽されています。

状態モナドと類似のもっと複雑な例に進みましょう。リソースを利用する計算のための**埋め込み言語**(*embedded language*)を定義します。特別な目的のための言語を Haskell の型と関数の集合として構築し、対象分野に特化した型と関数のライブラリを構築します。

何らかのリソースを必要とする計算を考えます。リソースが利用可能な場合は計算を先へ進め、リソースが利用できない場合は計算を保留します。リソースを使う計算の型 R を次のように定義します。

```
data R a = R (Resource -> (Resource, Either a (R a)))
```

計算は、リソースを与えると(残りのリソース, 型 a の結果か保留された計算)の組を返す関数です。計算はリソースが残っている間は続けます。

Either 型は次の通りです。

```
data Either a b = Left a | Right b
```

リソースを制御する R の Monad インスタンスは以下のようになります。

```
instance Monad R where
  R c1 >>= fc2          = R ¥r -> case c1 r of
                                    (r', Left v)    -> let R c2 = fc2 v in c2 r'
                                    (r', Right pc1) -> (r', Right (pc1 >>= fc2))
  return v              = R ¥r -> (r,  Left v)
```

定義はこんな風に読みます。リソースを消費する計算 c1 と fc2(c2 を生成する)を合成するには、初期リソースを c1 へ渡し、残りのリソース r'と次の結果の組を返します。

- 値 v : v は次の計算(c2 = fc2 v)の決定に使われ、次の計算を起動します(c2 r')。
- 保留された計算 pc1 : pc1 と次の計算の合成(pc1 >>= fc2)を保留します。

計算を保留する際は次の計算を考慮に入れなくてはなりません。 pc1 は最初の計算 c1 だけを保留するので、計算全体を保留するには pc1 を c2 と合成しておく必要があります。 return は値 v をモナドへ投入するだけでリソースは変更しません。

インスタンス宣言はリソースモナドの基本構造を定義しているだけで、リソースがどのように使われるかは何も決めていません。リソースモナドは多くのリソースを制御するのに使うことができ、種々のリソースタイプの使用ポリシーを実装することができます。利用可能な計算ステップ数をリソースとしてみます。step 関数は利用可能なステップ数がなくなるまで 1 ステップを消費します。

```
type Resource           =  Integer

step                    :: a -> R a
step v                  =  c where
                               c = R ¥r -> if r /= 0 then (r-1, Left v)
                                                     else (r, Right c)
```

step は利用可能なステップがあれば v を返して R 内の計算を継続します。利用可能なステップが無ければ現在の計算を保留し(これは c で捕捉されます)、保留された計算をモナドへ戻します。

次にリソースモナドの世界のインクリメント関数を考えてみましょう。 step を用いてリソースを隠蔽することができます。

```
inc                     :: R Integer -> R Integer
inc i                   =  do iValue <- i
                              step (iValue+1)
```

inc はリソースを消費しながらモナド値をインクリメントします。 <- はモナド R i から値 i を引出します。 iValue の型は R Integer ではなく Integer です。

この inc は一般のインクリメント関数に比べると満足できるものではありません。既存の + 演算子をモナドの世界で機能するようにできないでしょうか。**持ち上げ**(*lifting*)関数は、既存の機能をモナド内へ持ち上げます。

二つの lift 関数の定義を考察してみます。

```
lift1                   :: (a -> b) -> (R a -> R b)
lift1 f                 =  ¥ra1 -> do a1 <- ra1
                                      step (f a1)

lift2                   :: (a -> b -> c) -> (R a -> R b -> R c)
lift2 f                 =  ¥ra1 ra2 -> do a1 <- ra1
                                          a2 <- ra2
                                          step (f a1 a2)
```

lift 関数は f を R 内でリソースを消費する関数に持ち上げます。 inc を lift1 を用いて定義します。

```
inc                     :: R Integer -> R Integer
inc                     =  lift1 (+1)
```

lift2 を使うと R モナド内の新しい ==* を定義することができます。 ==* のシグネチャは == の多重定義で何とかなるようなものではありません。 ==* の結果が Bool ではなく R Bool だからです。

```
(==*)                   :: Ord a => R a -> R a -> R Bool
(==*)                   =  lift2 (==)
```

多重定義できるものもあります。 以下のインスタンス宣言は Num の演算子を R 内に持ち上げます。

```
instance Num a => Num (R a) where
  (+)                   =  lift2 (+)
  (-)                   =  lift2 (-)
  (*)                   =  lift2 (*)
  negate                =  lift1 negate
  abs                   =  lift1 abs
  fromInteger           =  return . fromInteger
```

Haskell では、整数定数に fromInteger 関数が暗黙のうちに適用されます(10.3 を参照)。この定義により、整数定数の型は R Integer になります。これで自然で完全な inc を書くことができます。

```
inc                     :: R Integer -> R Integer
inc x                   =  x + 1
```

R 内で興味のある計算を表現するためには条件式が必要です。if は使えませんから、ifR という関数を考えます。

```
ifR                     :: R Bool -> R a -> R a -> R a
ifR tst thn els         =  do t <- tst
                              if t then thn else els
```

これで R モナドでプログラムを書く準備が整いました。モナドは埋め込み言語の意味論をクリーンでモジュール性のある方法でカプセル化します。 fact 関数は十分に読みやすくなっています。

```
fact                    :: R Integer -> R Integer
fact x                  =  ifR (x ==* 0) 1 (x * fact (x-1))
```

プログラムを走らせる準備ができました。 run はプログラムを最大ステップ数内で動作させます。与えられたステップ数内で計算が終了したかどうかを Maybe 型で表示します。

```
run                     :: Resource -> R a -> Maybe a
run s (R p)             =  case (p s) of (_, Left v) -> Just v
                                         _          -> Nothing
```

以下のような計算が可能になります。

```
run 10 (fact 2)  => Just 2
run 10 (fact 20) => Nothing
```

リソースモナドに興味深い機能を加えることができます。2 つの計算を並列に実行し、最初に終った方の結果を返す関数です。次の関数を考えてみましょう。

```
(|||)           :: R a -> R a -> R a
c1 ||| c2       = oneStep c1 (¥c1' -> c2 ||| c1') where
       oneStep          :: R a -> (R a -> R a) -> R a
       oneStep (R c1) f = R ¥r -> case c1 1 of
                                  (r', Left v)    -> (r+r'-1, Left v)
                                  (r', Right c1') -> let R next = f c1' in next (r+r'-1)
```

oneStep は c1 にリソース 1 を与え、それで最終値に達したらその値を返します。 c1 が保留された計算(c1')を返したら次に c2 ||| c1' を評価します。リソースを消費せずに戻ることもあるので残りのリソースが 0 であるとは限りません。 oneStep 関数は 1 ステップを消費して評価された値を返すか、保留された計算 c2 ||| c1' を評価するかのどちらかです。いずれの場合もリソースが調整されます。

これで run 100 (fact (-1) ||| (fact 3)) のような式をループに陥らずに評価することができます。先に定義した fact は -1 を与えるとループに陥ります。この基本構造の上に色々なバリエーションを考えることができます。計算ステップのトレースをとることも可能になります。リソースモナドを IO モナドに埋め込むと、モナド内の計算が外界とやり取りできるようになります。

この節はシステムの基本的な意味論を定義する道具としてのモナドの力を説明しています。小規模の**領域限定言語**(*Domain Specific Language*)のモデルを説明し、Haskell がこうした言語を定義するのに適していることを示しました。 Haskell で開発された多くの DSL が haskell.org にありますので参照してください。リアクションのあるアニメーション言語 Fran とコンピュータミュージック言語 Haskore は特に興味を引くと思います。

10 数

Haskell には種々の数値型があります。これらは Scheme [7] と Common Lisp [8] の数値型をもとにしています。標準の数値型には、固定長と任意倍長の整数(Int, Integer)、整数で構成される比(Rational)、単精度と倍精度の実数(Float, Double)、浮動小数の複素数(Complex)があります。ここでは数値クラスの構造の概略を説明します。詳細についてはレポートの(§6.3)を参照してください。

10.1 数値クラスの構造

数値クラス Num とその下にあるクラスは種々の標準クラスを考慮しています。Num は Eq のサブクラスですが、Ord のサブクラスではありません。順序述語が複素数に適用できないからです。しかし、 Num

のサブクラス Real は Ord のサブクラスです。以下は数値クラスの一覧です。

```
class (Eq a, Show a)           => Num a         where ... +, -, *, negate, abs, signum, ...
class (Num a, Ord a)           => Real a        where ...
class (Real a, Enum a)         => Integral a    where ... quot, rem, div, mod, ...
class (Num a)                  => Fractional a  where ... /, recip, ...
class (Fractional a)           => Floating a    where ... pi, exp, log, sqrt, sin, cos, ...
class (Real a, Fractional a)   => RealFrac a    where ... truncate, round, ceiling, floor, ..
class (RealFrac a, Floating a) => RealFloat a   where ... exponent, significand, ...
```

Num クラスはすべての数値型に共通の基本演算を備えています。加算、減算、乗算、符号反転、絶対値などです。除算がないことに注意してください。

```
(+), (-), (*)           :: (Num a) => a -> a -> a
negate, abs, signum     :: (Num a) => a -> a
```

[negate は前置演算子 - (マイナス)を適用する関数です。]

Integral クラスは 2 種類の除算と剰余演算 (quote, rem と div, mod) を提供しています。Integral の標準インスタンスは Integer と Int です。 Integral は Num のサブクラスではなく、Real のサブクラスであることに注意してください。その意味するところは、ガウス整数を提供するつもりはないということです。

他の数値型は Fractional クラスとその下のクラスに属します。 Fractional は普通の除算 (/) を備えています。そのサブクラスに Floating があり、三角関数、対数関数、指数関数を備えています。Fractionnal と Real のサブクラスである RealFrac は properFraction 関数を備えています。この関数は数を整数部と小数部に分けます。整数に丸める 2 種類の関数群も備えています。

```
properFraction                  :: (Fractional a, Integral b) => a -> (b,a)
truncate, round, floor, ceiling :: (Fractional a, Integral b) => a -> b
```

標準の型 Float と Double は RealFloat クラスになります。 RealFloat は Floating と RealFrac のサブクラスで浮動小数部をアクセスするめの関数 exponent と significand を備えています。

10.2 構造のある数値型

標準の数値型 Int、Integer、Float、Double はプリミティブです。他の数値型はこれらの型から構築されます。

複素数型は型構成子 Complex とデータ構成子 (:+) を用いて構築します。複素数は Complex ライブラリにあります。

```
data (RealFloat a) => Complex a = !a :+ !a  deriving (Eq, Text)
```

文脈 (RealFloat a) => に注目してください。複素数型は Complex Float と Complex Double だということです。複素数は x :+ y と書きます。 x と y は実数部と虚数部です。記号 ! は正格性フラグです。 :+ はデータ構成子なのでパターン照合に使用することができます。

```
conjugate               :: (RealFloat a) => Complex a -> Complex a
conjugate (x:+y)        =  x :+ (-y)
```

分数は型構成子 Ratio を使って Integral のインスタンスから構成します。 Ratio は抽象型構成子です。中置の演算 (%) を使って二つの整数から分数を形成します。データ構成子はなく、パターン照合の代わりに構成要素を引出す関数が用意されています。

```
(%)                   :: (Integral a) => a -> a -> Ratio a
numerator, denominator  :: (Integral a) => Ratio a -> a
```

複素数との差の理由は何でしょう。複素数 x:+y の同一性は明白でありユニークに決定できます。分数 x%y は既約形式を持ちますがユニークには決定できません。既約形式は抽象型の実装が保持すべきものです。 x:+y の実数部は常に x ですが (x%y) の分子は x に等しいとはかぎりません。型 Rational は Ratio Integer の型シノニムです。

```
type Rational = Ratio Integer
```

10.3 数値型変換と多重定義されたリテラル

標準プレリュードと標準ライブラリには明示的な型変換を行う多重定義関数が備わっています。

```
fromInteger           :: (Num a)                    => Integer -> a
fromRational          :: (Fractional a)             => Rational -> a
toInteger             :: (Integral a)               => a -> Integer
toRational            :: (RealFrac a)               => a -> Rational
fromIntegral          :: (Integral a, Num b)        => a -> b
fromRealFrac          :: (RealFrac a, Fractional b) => a -> b
fromIntegral          =  fromInteger . toInteger
fromRealFrac          =  fromRational . toRational
```

fromInteger と fromRational は、 Integer と Rational を多相型に変換し、多重定義数値リテラルを実現するために暗黙に使用されています。 7 の型は (Num a) => a 、 7.3 の型は (Fractional a) => a です。数値リテラルは総称的な数値関数で用いることができます。

```
halve                 :: (Fractional a) => a -> a
halve x               =  x * 0.5
```

fromInteger と fromRational による数値の多重定義法には、与えられた型の数値として解釈するためのメソッドを Integral あるいは Fractional のインスタンス宣言で指定できる利点があります。これが可能なのは、 fromInteger と fromRational が対応するクラスの演算子であるからです。たとえば、 (RealFloat a) => Complex a は次のメソッドを含んでいます。

```
fromInteger x         =  fromInteger x :+ 0
```

fromInteger の Complex 版インスタンスは、実数部が RealFloat 版インスタンスの fromInteger により生成される複素数のためのものである、ということです。

他の例をあげましょう。2 節の inc の最初の定義を思い出してください。

```
inc                   :: Integer -> Integer
inc n                 =  n+1
```

この型シグネチャには inc の型を限定する効果があります。型シグネチャを無視すれば、一般的な inc

の型は (Num a) => a->a です。 inc の型シグネチャが正当なのは、これが主型より**限定的**であるからです。主型より一般的な型では静的エラーが発生します。

10.4 デフォルトの数値型

以下のような関数の定義を考えてみてください。

```
rms             :: (Floating a) => a -> a -> a
rms x y         =  sqrt ((x^2 + y^2) * 0.5)
```

指数関数 (^) の型は (Num a, Integral b) => a -> b -> a です。これは別の型付けがされた 3 つの指数関数のうちの一つです。これについてはレポートの 6.8.5 節を参照。 2 の型は (Num a) => a なので、 x^2 の型は (Num a, Integral b) => a になります。型変数 b は文脈にありますが型式には無い問題があります。 b の多重定義を解決する方法がありません。この問題は解決することはできます。

```
rms x y         =  sqrt ((x ^ (2::Integer) + y ^ (2::Integer)) * 0.5)
```

しかし、面倒になるのは明白です。この手の多重定義の曖昧性は数値に限ったことではありません。

```
show (read "xyz")
```

この文字列は Text インスタンスのどのような型だと仮定すればよいのでしょうか。これは指数関数の曖昧性よりも深刻な問題です。なぜなら、先程の場合は Integral のインスタンスであればよいのですが、今度の場合は Text のどのインスタンスがこの曖昧性の解決に用いられるかによって期待できる振舞いが変わってしまうからです。

Haskell は数値に限って、多重定義の曖昧性の問題の解決法を提供しています。モジュールには**デフォルト宣言**(*default declaration*)を置くことができます。例えば次のように宣言します。( ) の中は数値のモノタイプ(型変数を含まない型)です。

```
default (Int, Float)
```

曖昧な型変数がみつかったとき(上例の b)、少なくともクラスの一つが数値クラスであり、かつ、クラスがすべてが標準のものであれば、デフォルトリストが参照され、型変数の文脈を満足する最初のものが使用されます。例のデフォルト宣言があれば、上の曖昧な指数は Int と解決されるでしょう。詳細についてはレポートの §4.3.4 を参照してください。

デフォルトのデフォルトは(Integer, Double)です。 (Integer, Rational, Double)でもかまわないでしょう。 注意深いプログラマなら default () とするかもしれません。デフォルトが無いということです。

11 モジュール

プログラムは**モジュール**(*module*)の集合です。モジュールは名前空間の制御と抽象データ型のためにあります。トップレベルのモジュールは、これまで議論した結合性宣言、データ宣言、type 宣言、クラス宣言、インスタンス宣言、型シグネチャ、 関数定義、パターン束縛等をすべて含みます。宣言はどのような順序で出現してもかまいません。ただし、インポート宣言はモジュールの最初になければなりません。トップレベルのスコープは相互再帰的です。

モジュールは比較的保守的にできており、モジュールの名前空間は平坦です。モジュール名は英数字で最初は大文字です。実装ではモジュール名とファイル名を関連させるのが普通ですが、正式にはモジュ

ール名とファイル名とは結びつきません。二つ以上のモジュールを一つのファイルに含むことができ、一つのモジュールを複数のファイルに分けることもできます。

モジュールは module キーワードで始まる一つの大きな宣言とみなすことができます。しかし、モジュールは「第一級」の対象ではありません。次は Tree モジュールの例です。

```
module Tree ( Tree(Leaf,Branch), fringe ) where

data Tree a                = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

[where キーワードがあり、モジュールでレイアウトが有効になります。宣言は開始カラム位置(普通は第一カラム)に配置されます。モジュール名は型の名前と同じですが、これは許されています。]

このモジュールは、Tree, Leaf, Branch, fringe を**エクスポート**しています。モジュール名の後にエクスポートリストがなければ、モジュールのトップレベルにある名前がすべてエクスポートされます。上の例では、効果としてはエクスポートリストがない場合と同じです。型名と構成子は Tree(Leaf,Branch) のようにグループにまとめます。簡略して Tree(..) のように書くこともできます。構成子の一部だけをエクスポートするのも可能です。エクスポートリスト内の名前は、モジュールで宣言した名前である必要はありません。スコープ内の名前であればエクスポートリストに入れることができます。

モジュールは別のモジュールに**インポート**することができます。インポート、エクスポートされる項目のことを**エンティティ**と呼びます。 import 宣言でインポートリストを指定することが出来ます。インポートリストを省略すると、エクスポートされたすべてのエンティティがインポートされます。

```
module Main (main) where
import Tree (Tree(Leaf,Branch), fringe)

main = print (fringe (Branch (Leaf 1) (Leaf 2)))
```

11.1 修飾子

インポートされた名前は問題をはらんでいます。インポートモジュールが同じ名前の別のエンティティを含んでいたらどうなるでしょう。**修飾子**(*qualified name*)を使うことで解決します。インポート宣言に qualified キーワードを指定し、インポートモジュール名をインポートされたエンティティの修飾子とします。エンティティは、空白を入れずに修飾子.エンティティ名で使用します。

[修飾子は字句構文の一部です。 A.x は修飾された名前、 A . x は関数合成です。]

以下の Fringe モジュールがあるとします。Fringe モジュールと Tree モジュールを使ってみます。

```
module Fringe(fringe) where
import Tree(Tree(..))

fringe :: Tree a -> [a]   -- A different definition of fringe
fringe (Leaf x) = [x]
fringe (Branch x y) = fringe x
```

修飾子を使い、インポートされた名前の由来が分かるのを好む人もいるでしょう。一方、短い名前を好み、修飾子は必要最小限にしようとする人もいるでしょう。

```
module Main where
import Tree (Tree(Leaf,Branch), fringe)
import qualified Fringe ( fringe )

main = do print (fringe (Branch (Leaf 1) (Leaf 2)))
          print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)))
```

修飾子は同じ名前の別のエンティティ間の衝突を解決します。二つ以上のモジュールから同じエンティティがインポートされた場合どうなるでしょう。このような名前の衝突は許されています。エンティティが別の経路でインポートされても衝突は越しません。コンパイラが別のモジュールからのエンティティが同じものかどうかを判別します。

11.2 抽象データ型

モジュールは Haskell で抽象データ型(ADT)を構築する唯一の方法です。抽象データ型の特徴は**表現型**(*representation type*)が**隠蔽**されることです。 ADT に対する演算は、表現型に依存しない抽象レベルで実行されます。 Tree 型は普通は抽象型にはしませんが、TreeADT は次の演算をもちます。

```
data Tree a             -- just the type name
leaf                    :: a -> Tree a
branch                  :: Tree a -> Tree a -> Tree a
cell                    :: Tree a -> a
left, right             :: Tree a -> Tree a
isLeaf                  :: Tree a -> Bool
```

モジュールはこの ADT を以下のように支援します。エクスポートリストには Tree のみがあり、型構成子 Leaf, Branch はエクスポートされません。木を構成したり、要素を取り出す唯一の方法は、抽象演算を使うことです。

```
module TreeADT (Tree, leaf, branch, cell, left, right, isLeaf) where
data Tree a             = Leaf a | Branch (Tree a) (Tree a)
leaf                    = Leaf
branch                  = Branch
cell   (Leaf a)         = a
left   (Branch l _)     = l
right  (Branch _ r)     = r
isLeaf (Leaf _)         = True
isLeaf   _              = False
```

11.3 その他の特徴

モジュールの他の側面について手短に説明しましょう。詳細はレポートを参照してください。

- import 宣言では hiding 節を用いてエンティティを隠蔽できます。これは他の目的で使いたい名前を排除するのに便利です。

- import 宣言に as 節を置くことより、モジュール名とは別の修飾子を指定することができます。修飾子を短くし、修飾子を変更せずにモジュール名を変更することができます。
- Prelude モジュールは暗黙のうちにインポートされます。プレリュードのインポート宣言は、暗黙のインポートのすべてを上書きします。例えば、import Prelude hiding length は length をインポートせず、 length を別の方法で定義することができます。
- インスタンス宣言は import, export 宣言と無関係にインポート、エクスポートされます。モジュールはすべてのインスタンス宣言をエクスポートし、インポート宣言はモジュールのすべてのインスタンス宣言をインポートします。

インポートとエクスポートには多くの明白なルールがあります。同じ名前をもつ別のエンティティをインポートするのは違反です。これほど明白ではないルールもあります。詳細はレポート(§5)を参照。

12 型付けの落し穴

Haskell を使う際、初心者に共通する型システムの問題について直観的な説明をします。

12.1 let-束縛の多相性

Hidler-Milner の型システムを使う言語には**let-束縛の多相性**(*let-bound polymorphism*)と呼ばれるものがあります。 let あるいは where 節で束縛のない識別子は、多相性に関して制限があります。特に、**ラムダ束縛された**(*lambda-bound*)関数、すなわち、別の関数の引数になった関数は、 2 種類の方法ではインスタンス化できません。以下のプログラムは不正なプログラムです。

```
let f g  =  (g [], g 'a')                    -- ill-typed expression
in f (¥x->x)
```

主型が a->a のラムダ抽象に束縛された関数 g が let 内で別々の方法で使われているからです。 一度は [a]->[a] 型、もう一度は Char->Char 型で使われています。

12.2 数値の多重定義

数が多重定義されること、数の型変換が暗黙のうちに起こることはないことをよく忘れてしまいます。数値式が総称的になれないこともあります。数値の型付けエラーは次のようなものです。

```
average xs              =  sum xs / length xs          -- Wrong!
```

(/) は Fractional a => a の引数を必要とします。 sum は多相型ですが length は Int です。型の不一致は、明示的な型変換によって訂正しなければなりません。

```
average                 :: (Fractional a) => [a] -> a
average xs              =  sum xs / fromIntegral (length xs)
```

12.3 単相性限定

型システムには Hidley-Milner の型システムには無い**単相性限定**(*monomorphism restriction*)の制限があります。制限は微妙な型の曖昧性に関連しています。詳細はレポート(§4.5.5)を参照してください。

単相性限定とは、パターン束縛を受け、型シグネチャのない識別子は、**単相的**(*monomorphic*)でなければならない、というものです。識別子が単相的とは、多重定義されていないか、多重定義されていても高々一つの多重定義で用いられ、エクスポートされていないことです。

この制限に違反すると静的型エラーが起こります。問題を回避するには、識別子に型シグネチャをつけることです。型シグネチャが正しければ問題を回避できます。

この制限違反は高階関数の手法で関数を定義する際におこります。標準プレリュードにある sum の定義の例をあげましょう。

```
sum                     =  foldl (+) 0
```

引数の無い sum はパターン束縛を受けます。また、 sum は多重定義されており単相的ではないので、静的型エラーを起こしますが、型シグネチャを付けることで解決します。

```
sum                     :: (Num a) => [a] -> a
```

この問題は以下のように書けば起こりませんでした。 sum がパターン束縛を受けず、この制限はパターン束縛にのみ適用されるからです。

```
sum xs                  =  foldl (+) 0 xs
```

13　配列

一般に関数型言語の配列は、インデックスから値への関数とみなします。しかし、配列要素への効率的なアクセスを確保するために、アクセス関数の領域の特別な性質の利点を十分活用できることを確認しておく必要があります。連続した整数からなる整数の有限部分集合の同型性という性質です。 Haskell は配列を関数としては扱わず、サブスクリプト演算を行う抽象データ型として扱います。

配列へのアプローチには、インクリメンタル配列とモノリシック配列があります。インクリメンタル配列は、与えられたサイズの空配列を作る生成関数と、配列，インデックス，値を引数に取りインデックスの要素だけが元の配列と違う新しい配列を生成する更新関数からなります。インクリメンタル配列は、更新のたびに配列の新しい複製が必要になり非効率になります。このアプローチを使いたい場合は、過度の複製を回避するための分析と巧妙な実行時の仕組を必要とします。

一方、モノリシック配列は、配列全体を一度に生成します。生成中に配列の値を参照することはありません。Haskell の配列はモノリシック配列ですが、インクリメンタルな配列更新の演算子もあります。

配列を使用するときは Array モジュー ルをインポートします。配列はプレリュードにはありません。

13.1　インデックス型

Ix ライブラリは配列インデックスの型クラスを定義しています。Int、Integer、Char、Bool のプリミティブな型を 1 次元インデックス、その組を多次元インデックスとします。 5 次元までの Ix の型 (a,a,a,a,a) が可能です。列挙型と組型は Ix インスタンスを自動的に導出することができます。

```
class  (Ord a) => Ix a  where
    range       :: (a,a) -> [a]
    index       :: (a,a) -> a -> Int
    inRange     :: (a,a) -> a -> Bool
```

クラス Ix の各演算の第一引数がインデックスの組である点に注目して下さい。第一引数は配列の境界です。境界は配列の(最初インデックス, 最後のインデックス)の組です。 Int インデックスの 0-オリジンの要素数 10 の境界は (0,9) です。 1-オリジンの 100 × 100 行列の境界は ((1,1),(100,100)) です。多くの言語では、境界は 1:100, 1:100 のように書きますが、ここに示した形式の方が型システムにピッタリします。境界がインデックスと同じ型の組になるからです。

range 関数は境界の間にあるインデックスの順序リストを生成し、inRange 述語はインデックスが境界に入っているかどうかを判定します。

```
range (0,4)          => [0,1,2,3,4]
range ((0,0),(1,2)) => [(0,0),(0,1),(0,2),(1,0),(1,1),(1,2)]
```

index 関数は配列の要素を特定するのに必要です。境界とインデックスを与えるとインデックスが 0-オリジンで何番目になるかを返します。たとえば、次のようになります。

```
index (1,9) 2              => 1
index ((0,0),(1,2)) (1,1) => 4
```

13.2 配列の生成

モノリシック配列の生成関数 array は、境界と連想リスト[(インデックス,値)]から配列を生成します。

```
array                   :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

1 から 100 までの自乗の配列を生成してみます。配列の境界を引数にとる場合も例示します。配列は連想リストの内包表記の典型的な使用例です。

```
squares                 =  array (1,100) [(i, i*i) | i <- [1..100]  ]
squares bnds            =  array  bnds   [(i, i*i) | i <- range bnds]
```

配列からインデックスの要素を取り出すには中置演算子 ! を使います。境界は bounds 関数を使って取り出すことができます。

```
squares!7      => 49
bounds squares => (1,100)
```

array 関数を一般化することができます。 mkArray 関数は、インデックスの配列要素を決める関数 f と境界 bnds をパラメータ化します。 squares は mkArray (¥i -> i*i) (1,100) と定義できます。

```
mkArray                 :: (Ix a) => (a -> b) -> (a,a) -> Array a b
mkArray f bnds          =  array bnds [(i, f i) | i <- range bnds]
```

多くの配列は再帰的に定義します。フィボナッチ数の配列を返す関数をみましょう。

```
fibs    :: Int -> Array Int Int
fibs n  =  a where a = array (0,n) ([(0,1),(1,1)] ++ [(i,a!(i-2)+a!(i-1)) | i <- [2..n]])
```

2 次元配列の再帰例として、$n \times n$ のウェーブフロント行列を取り上げます。最初の行と最初の列の要素がすべて 1 、その他の要素はその左側と左上側と上側の和であるような行列です。

```
wavefront   :: Int -> Array (Int,Int) Int
wavefront n  = a  where
               a = array ((1,1), (n,n))
                    ([((1,j),1) | j <- [1..n]] ++ [((i,1),1) | i <- [2..n]] ++
                     [((i,j), a!(i,j-1)+a!(i-1,j-1)+a!(i-1,j)) | i <- [2..n], j <- [2..n]])
```

ウェーブフロント行列の生成は、最初の行と最初の列から並列に始まり、左上から右下へ進み、並列実装ゆえに反復命令とよばれます。しかし、生成の順序が連想リストによって指定されているわけではありません。

これまでは、境界内のインデックスにのみ連想リストを与えました。配列を定義するには、このようにしなければなりません。境界を越えるインデックスの連想リストはエラーを起こします。インデックスが欠けたり、同じインデックスが 2 回以上出現しても即座にエラーにはなりませんが、当該インデックスの値を参照するときにはエラーとなります。

13.3 累積配列

累積配列は、同じインデックスが連想リストに 1 度しか出現してはいけないという制限をなくします。**累積配列**(*accumulated array*)には、配列要素の値の**累積関数**(*accumulating function*)と初期値を与えます。初期値は各要素に対して同じ値です。典型的な累積関数は(+)、初期値は 0 です。

```
accumArray :: (Ix a) => (b -> c -> b) -> b -> (a,a) -> [Assoc a c] -> Array a b
```

accumArray の引数は、累積関数、初期値、境界、連想リストです。次の関数 hist は、ヒストグラム、すなわち、インデックスの出現頻度表を生成します。関数 hist は境界とインデックスリストを引数にとります。

```
hist            :: (Ix a, Integral b) => (a,a) -> [a] -> Array a b
hist bnds is    =  accumArray (+) 0 bnds [(i, 1) | i <- is, inRange bnds i]
```

インデックスの区間 *[a,b)* を 10 区間 (0,9) に等分したヒストグラムは次のようになります。

```
decades         :: (RealFrac a) => a -> a -> [a] -> Array Int Int
decades a b     =  hist (0,9) . map decade where
                       decade x = floor ((x - a) * s)
                       s        = 10 / (b - a)
```

13.4 配列更新

配列の更新関数は中置演算子 // で実現します。

```
(//)            :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

a // [(i, v)] は、配列 a のインデックス i の要素を値 v に更新します。 [(i, v)] は連想リストです。 i は連想リストで重複することはできません。

次の例は行列の二つの行を入れ替えます。

```
swapRows :: (Ix a, Ix b, Enum b) => a -> a -> Array (a,b) c -> Array (a,b) c
swapRows i i' a = a // ([((i ,j), a!(i',j)) | j <- [jLo..jHi]] ++
                        [((i',j), a!(i ,j)) | j <- [jLo..jHi]])
                  where ((iLo,jLo),(iHi,jHi)) = bounds a
```

同じリストの二つのリスト内包表記の連結は効率の悪いもので、命令型言語のループを二つ書くようなものです。ループの融合最適化と同じことが可能です。

```
swapRows i i' a = a // [ assoc | j <- [jLo..jHi],
                          assoc <- [((i ,j), a!(i',j)), ((i',j), a!(i ,j))] ]
                  where ((iLo,jLo),(iHi,jHi)) = bounds a
```

13.5 行列の積

配列の紹介を行列の積で締めくくりましょう。多重定義の長所を利用します。乗法と加法しか必要ないので、全ての数値型の行列の積の関数を手に入れることが出来ます。インデックス型に対しても一般性を得ることが出来ます。行と列の 4 つのインデックスの型は同じである必要はありません。しかし、左側の行列の列インデックスと右側の行列の行インデックスは同じ型、同じ境界とします。

```
matMult         :: (Ix a, Ix b, Ix c, Num d) =>
                   Array (a,b) d -> Array (b,c) d -> Array (a,c) d
matMult x y     =
        array resultBounds [ ((i,j), sum [ x!(i,k) * y!(k,j) | k <- range (lj,uj) ])
                                | i <- range (li, ui ), j <- range (lj',uj') ]
        where ( (li ,lj ), (ui ,uj ) ) =  bounds x
              ( (li',lj'), (ui',uj') ) =  bounds y
              resultBounds | (lj,uj) == (li',ui') =  ( (li,lj'), (ui,uj') )
                           | otherwise            = error "matMult: incompatible bounds"
```

accumArray を用いて matMult を定義することも可能です。

```
matMult x y     =
        accumArray (+) 0 resultBounds [((i,j), x!(i,k) * y!(k,j)) | i <- range (li, ui ),
                                                                    j <- range (lj',uj'),
                                                                    k <- range (lj, uj )]
        where -- 同上
```

sum や (*) を関数パラメータにしてさらに一般化をはかることができます。以下のような便利な関数を使うことができます。

```
genMatMult      :: (Ix a, Ix b, Ix c) => ([f] -> g) -> (d -> e -> f) ->
                   Array (a,b) d -> Array (b,c) e -> Array (a,c) g
genMatMult sum' star x y  =
        array resultBounds [((i,j), sum' [x!(i,k) `star` y!(k,j) | k <- range (lj,uj)])
                                | i <- range (li, ui ), j <- range (lj',uj')]
        where -- 同上

genMatMult maximum (-) -- 行列の積の(i,j)要素は i 行と j 列の要素の最大差
genMatMult and (==)    -- 行列の積の(i,j)要素は i 行と j 列がベクトルとして同値のときに True
```

genMatMult の要素の型は単に引数 star の適切な型であればよいだけです。 x の列と y の行のインデックスの型が同じという要請を落とすことができます。 x の列数と y の行数が同じであれば行列の積が定義できます。この版を作りたいと思いませんか。長さの決定には index 演算を使用します。

14 次の段階

Haskell の膨大なコレクションが haskell.org のウェブサイトで利用できます。コンパイラ、デモ、論文、そして Haskell と関数プログラミングに関する重要な情報を見つけることができます。コンパイラ／インタプリタは、ほとんどのハードウェアと OS で動作します。 Hugs システムは小さく可搬性があり、Haskell を学習するのに大変秀れたシステムです。リアクションのあるアニメーションのための言語 Fran とコンピュータミュージックのための言語 Haskore は、特に興味を引くと思います。

15 謝辞

Los Alamos の Patricia Fasel, Mark Mundt、Yale 大学の Nick Carriero, Charles Consel, Amir Kishon, Sandra Loosemore, Martin Odersky, David Rochberg に感謝します。この原稿の初期の草稿をすぐに読んでくれました。 Erik Meijer はこのチュートリアルの 1.4 版への追加した題材について詳細にわたるコメントをくれました。

参考文献

[1] R. Bird. *Introduction to Functional Programming using Haskell.* Prentice Hall, New York, 1988.

[2] A. Davie. *Introduction to Functional Programming System using Haskell.* Cambridge University Press, 1992

[3] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359-411,1989.

[4] Simon Peyton Jones (editor). Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language. *Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1106*, Feb 1999.

[5] Simon Peyton Jones (editor) The Haskell 98 Library Report. *Yale University, Department of Cmputer Science Tech Report YALEU/DCS/RR-1105* Feb 1999.

[6] R.S. Nikhil. Id (version 90.0) reference manual. Technical Report, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1990.

[7] J. Rees and W. Clinger (eds.). The revised^3 report on the algorithmic language Scheme. *SIG- PLAN Notices*, 21(12):37-79, December 1986.

[8] G.L. Steele Jr. *Common Lisp: The Language*. Digital Press, Burlington, Mass., 1984.

[9] P. Wadler. How to replace failure by a list of successes. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture, LNCS Vol. 201*, pages 113-128. Springer Verlag, 1985.

[10] P. Wadler. Monads for Functional Programming In *Advanced Functional Programming* , Springer Verlag, LNCS 925, 1995.