第3章 代数的データ型と型クラス

3.1 代数的データ型の定義

リストのようなデータ型をプログラマーがあらたに定義する方法も用意されている。このようなデータ型は複数の＿＿＿＿(constructor) を持つことができる。そして個々の構成子はいくつかのフィールド (field) を持つことができる。 このようなデータ型を＿＿＿＿＿＿＿＿(algebraic datatype) という。

データ型の宣言の一般的な形式は次のとおりである。

```
data 型構成子名 型引数1 型引数2 … 型引数k
   = 構成子名1 型1,1 … 型1,n1
   | 構成子名2 型2,1 … 型2,n2
   | …
   | 構成子名m 型m,1 … 型m,nm
```

型構成子名・構成子名ともに使える文字は変数名の場合と同じだが、変数名とは逆に＿＿＿＿から始まる必要がある。この型は 構成子名$_1$ ～ 構成子名$_m$ の m 種類のデータからなる。型$_{i,1}$, ..., 型$_{i,n_i}$ は 構成子名$_i$ が持つフィールドの型である。型引数$_1$, 型引数$_2$, ..., 型引数$_k$ はフィールドの型の中に現れうる型変数である。ここで「|」は “または” と読む。例えば

```haskell
data Foo x y = Alice x [y] | Bob String y | Charlie
```

という（特に意味のない）データ型の場合、`Foo Double Integer` 型になるのは次のような式である、 `Alice 3.14 [1,2]` あるいは `Bob "Hello" 3` あるいは `Charlie`, ...。

代数的データ型はすべての構成子にフィールドがない場合は、C や Java の列挙 (enum) 型と同じようなものである。次の例では、

```haskell
data Direction = North | South | West | East
               deriving (Eq,Ord,Show)
```

`North`, `South`, `West`, `East` の 4 つが `Direction` 型を構成している。
（**`deriving` ...** については後述する。）

Q 3.1.1 じゃんけんの 3 つの手（グー (`Rock`)・チョキ (`Scissors`)・パー (`Paper`)）を表すデータ型 `Janken` を定義せよ。(**`deriving` ...** はつけなくてよい。)

一般的には代数的データ型の構成子はフィールドを持つ。次の例は二分木 (binary tree) を表すデータ型を定義している。

```
data Tree a = Empty | Branch (Tree a) a (Tree a)
                            deriving (Eq,Ord,Show)
```

このデータ型は Branch と Empty の 2 つの構成子を持つ。 Empty はフィールドを持たず、それだけで二分木を構成する。 Branch は 3 つのフィールドを持つ。1 番目と 3 番目の Tree a は自分自身と同じ型の二分木であり、2 番目の a は要素である。つまり、Branch は次のような型を持っている。

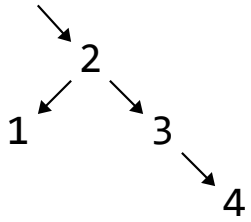
```
Branch ::
```

ここで、a は＿＿＿＿＿＿であり、ここには Integer や String などの具体的な型が入ることができる。例えば Tree Integer は要素が Integer 型であるような二分木の型である。Tree 自体は型ではなく型構成子 (type constructor)である。つまり、型パラメーターを伴ってはじめて型になる。

具体的な例として、次のような式の例がこの Tree の型を持つ。

```
tree1 :: Tree a
tree1 = Empty
tree2 :: Tree Integer
tree2 = Branch Empty 1 Empty
tree3 :: Tree String
tree3 = Branch (Branch Empty "a" Empty)
           "b" (Branch Empty "c" Empty)
tree4 :: Tree Integer
tree4 = Branch (Branch Empty 1 Empty)
             2 (Branch Empty 3 (Branch Empty 4 Empty))
```

例えば tree4 の構造は、図で表すと次のようになる。



Tree に対する関数は、パターンマッチングを用いて、次のように定義することができる。つまり、仮引数の位置に構成子と変数からなる式を書くことができる。

```
top :: Tree a -> a
top (Branch _ a _) = a

isEmpty :: Tree a -> Bool
isEmpty Empty = True
isEmpty _     = False

```

```haskell
-- 要素数、例えば treeSize tree4 は 4
treeSize :: Tree a -> Int
treeSize Empty          = 0
treeSize (Branch l a r) = treeSize l + 1 + treeSize r
```

つまり、top は二分木の先頭の要素を返す関数、isEmpty は空かどうかを判定する関数、treeSize は要素数を返す関数である。

Q 3.1.2 右、左、反対方向を表す関数 right, left, opposite :: Direction -> Direction を定義せよ。

Q 3.1.3 第 1 引数が第 2 引数に対して、Janken 型の勝ち・負け・引き分けのいずれであるかを判定する関数 judgeJanken :: Janken -> Janken -> Ordering を定義せよ。ただし、Janken に deriving (Eq, Show, Ord) はないものとして、つまり「==」や「<」などは Janken 型に適用できないとして考えよ。

ここで Ordering は次のように標準ライブラリーで定義された型である。

```haskell
data Ordering = LT | EQ | GT
              deriving (Eq,Ord,Bounded,Enum,Read,Show)
```

LT が負け、GT が勝ち、EQ が引き分け、を表すものとする

問 3.1.4 Tree 型に対して、次のような関数を定義せよ。

```haskell
depth     :: Tree a -> Integer -- 深さ
preorder  :: Tree a -> [a]     -- 前順走査
inorder   :: Tree a -> [a]     -- 中順走査
postorder :: Tree a -> [a]     -- 後順走査
reflect   :: Tree a -> Tree a  -- 鏡像
```

例えば、① depth tree4, ② preorder tree4, ③ inorder tree4, ④ postorder tree4, ⑤ reflect tree4 の結果はそれぞれ、① 3, ② [2,1,3,4], ③ [1,2,3,4], ④ [1,4,3,2], ⑤ Branch (Branch (Branch Empty 4 Empty) 3 Empty) 2 (Branch Empty 1 Empty) となる。

問 3.1.5 一般の n 分木（任意の個数の部分木を持つことができる木、rose tree ともいう）を表すデータ型 RoseTree を定義せよ。また、n 分木の要素数を求める関数 roseSize :: RoseTree a -> Integer を定義せよ。

（発展） フィールドラベル

C の構造体のように代数的データ型の各フィールドに名前（フィールドラベル）をつけることも可能である。次のように、フィールド部分をブレース（{ ～ }）で囲み、「::」の前にフィールドラベルを書く。

```haskell
data C = F { field1, field2 :: Int, field3 :: Bool }
    deriving (Eq,Ord,Show)
```

この宣言は、次の宣言と同等のデータ型を定義する。

```haskell
data C = F Int Int Bool deriving (Eq,Ord,Show)
```

さらに、フィールドラベルは新しいデータを構築するときにも、パターンマッチングにも使用することが出来る。いずれもコンストラクターの後に、ブレース内に「フィールドラベル = 式」のコンマ区切りの並びを書く。

```haskell
v1 = F { field1 = 5, field2 = 12, field3 = False }

foo :: C -> Int
foo (F { field1 = x, field2 = y }) = x * x + y * y
```

このとき `foo v1` の値は $(5^2 + 12^2 =)$ `169` になる。

またフィールドラベルは、データからフィールドの値を取り出す関数として使用することが出来る。例えば、上で定義された `v1` に対して、 `field1 v1` の値は `5` になる。

式の後に、「フィールドラベル = 式」のコンマ区切りの並びをブレース内に書いて、指定したフィールドの値のみを変更した新しいデータを構成するときにも使用できる。`v1 { field2 = 3 }` の値は `F { field1 = 5, field2 = 3, field3 = False }` になる。

3.2 型クラスとは

型クラスの説明に入る前にまずいくつかの用語を紹介する。

ポリモルフィズム（多相, polymorphism）とは関数（メソッド）などが＿＿＿＿＿＿＿＿を許すことをいう。 さらに、関数などがいろいろな型の引数を許し、しかも＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿ことを、＿＿＿＿＿＿（ad-hoc ― その場限りの、という意味）多相という。《アドホックな多相に対して、 型によって処理が変わらない多相のことをパラメトリック (parametric) な多相と言う。》オブジェクト指向言語の動的束縛 (dynamic binding) はアドホック多相の一種である。

オブジェクト指向言語では、単にポリモルフィズムという言葉で動的束縛の意味まで含むことがある。

動的束縛と混同しがちな概念として＿＿＿＿＿＿＿＿＿＿（多重定義, overloading）がある。多重定義は一つの名前が複数の意味を持つことである。例えばＣの「+」「==」「<」などのオペレーターは `int`（整数）型にも `double`（倍精度浮動小数点数）型にも適用できる。しかし最終的には、適用される型によって全く異なる機械語に翻訳される。動的束縛と異なる点は、多重定義はコンパイル（型チェック）時に解決されてしまう、という点である（つ

まり静的な束縛である）。実行時にオペランドの型に応じて処理を振り分けるようなことは行なわない。

つまり、多重定義もアドホック多相の実現方法の一つであると言えるが、適用範囲はコンパイル時に型がわかる場合に限定されてしまう。

Haskell のように型推論とパラメトリック多相を許す言語では、このように他の言語では多重定義で済んでしまう演算で困った事態が生じる。例えば、次のような関数:

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

を定義したとき、コンパイル時に得られる型情報では演算子の実装を決定することはできない。つまり、 `x` の型が整数か浮動小数点数か決定できないので、「`*`」の実装も決定できない。つまり、C や Java（の演算子）で使われているような多重定義は、Haskell では使えない。 すぐに考えられる処方は、`squareInt :: Int -> Int`, `squareDouble :: Double -> Double` のように型ごとに関数名を変えて、関数を定義することだが、同じような関数を何個も定義することになり受け入れがたい。もう一つの処方は、`square x = x * x` という定義をそのまま許し、実行時の `x` の型に応じて `*` の実装を選択することである。これは、実行時に型情報を受け渡さなければいけなくなるし、`square "abc"` のような式をエラーとして検知できなくなる、という大きな欠点がある。

一方、Haskell では一つの代数的データ型の異なるコンストラクターのなかでは、パターンマッチングによりアドホック多相を実現している。例えば、`Tree` 型の `Branch` と `Empty` では関数 `isEmpty` や `treeSize` の実装が異なる。上記の `square` のような関数の定義を可能にするためには複数の型にまたがるアドホック多相を、できれば効率よく実現する必要がある。そこで、Haskell では型クラスという仕組みが導入されている。型クラスは概念的には、ある条件を満たす型の集まりである。

3.3 Haskell の型クラス

以下では Haskell の型クラス (type class) を説明する。例として、`Eq` という型クラスを取り上げる。

Haskell でも C と同じように「`==`」（等号）オペレーターは `Integer`（整数）型にも `Double`（倍精度浮動小数点数）型にも、さらに他のいくつかの型にも適用できる。動作はもちろん型ごとに異なる。一方、関数型は「`==`」で比較できない。例えば `(\ x -> x + x) == (\ x -> 2 * x)` はエラーである。《関数の同値性は計算不能問題である。》Haskell は多相を許す言語であるので、例えば、

```
1 member x []     = False
2 member x (y:ys) = x == y || member x ys
3
```

```
4 subset xs ys    = all (\ x -> member x ys) xs
```

という関数 member や subset を「==」を使って定義すると、 member 5 [1, 4, 7] のように [Integer]（整数のリスト）型の引数にも、 member "Kagawa" ["Tokushima", "Ehime", "Kochi"] のように [String]（文字列のリスト）型の引数にも適用できる。しかし、 member (\ x -> x + x) [\ x -> x + 1, \ x -> 2 * x] のように関数のリストに適用することはできない。

ここで member や subset の型は、単なる a -> [a] -> Bool や [a] -> [a] -> Bool ではない。それでは、member や subset はどのような型を持ち、どのように実装されているのであろうか? Scheme や Python のように動的に型付けされる言語では、各データオブジェクトが型の情報をつねに保持しているので、実行時に型に応じて適切な関数を選択することができる。しかし、Haskell のようにコンパイル時に型チェックを行なう言語では、実行時にはデータは型の情報を保持していないのが普通である。

Haskell では、これらの関数の型は自動的に推論される（型推論 — ただし、その方法の詳細については本稿で取り扱う範囲を超える）。型推論の結果だけを示すと、memberとsubsetは次のような型を持っている。

```
member ::

subset ::
```

ここで "Eq a =>"という部分は、 a という型変数が Eq という型の集まり （　　） に属していなければいけない、という型に関する制約 (type constraint) を表す。Eq という型クラスは、「==」（等号）が定義されているような型の集まりのことである。一般に型クラスとは、　　のことである。

型クラスは通常のオブジェクト指向言語でのクラス・インスタンスという言葉とは意味が異なるので注意する。通常のオブジェクト指向言語ではクラスは　　（つまり型）であるのに対し、Haskell の型クラスは　　である。（Java のインタフェースの概念に似ている（というよりも、順序から言えば Java のインタフェースが Haskell の型クラスに似ている、というほうが適切である。）。）

3.4 クラス宣言とインスタンス宣言

型クラスの定義には **class** というキーワードを用いる。例えば、 Eq クラスの定義は Haskell では次のように書く。（Prelude に定義済み。）

```
1 class Eq a where
2   (==), (/=) :: a -> a -> Bool    -- != ではないので注意
3   a /= b = not (a == b)           -- デフォルトの定義
```

これが、「型 a が型クラス Eq に属するためには、 a -> a -> Bool という型を持つ 2 つの関数 (==), (/=) を持たなければいけない」という意味になる。
一般的には、

```
class クラス0 型変数0 where
  関数1 :: 型1
  関数2 :: 型2
  …
```

のようにキーワード class のあとにクラス名と型変数を書き、キーワード where の後に関数の型宣言を並べる。これは、型変数$_0$ がクラス$_0$に属するためには、型$_1$ という型をもつ関数$_1$、 型$_2$ という型をもつ関数$_2$ … が必要である（関数でなくても良いのだが、説明を簡単にするため、すべて関数ということにした。）、という意味である。

そして例えば以前紹介した Direction 型が（deriving 節がなかったとして） Eq クラスに属する（Direction が Eq の＿＿＿＿＿＿である）ことを宣言するためには **instance** というキーワードを用いる。

```
instance Eq Direction where
  North == North = True
  South == South = True
  West  == West  = True
  East  == East  = True
  _     == _     = False
```

ここで「(/=)」演算子については、クラス宣言の中でデフォルトの定義が用意されているので、インスタンス宣言では定義を省略することができる。

また、Tree 型の場合（deriving 節がなかったとして）、次のように宣言する。

```
instance Eq a => Eq (Tree a) where
  Empty           == Empty           = True
  Branch l1 n1 r1 == Branch l2 n2 r2
                   = l1 == l2 && n1 == n2 && r1 == r2
  _               == _               = False
```

"Eq a =>" の部分は Tree a に等号を定義するためには、要素の型である a に等号が定義されていなければいけないという制約を表す。Tree のようにパラメーターを持つ型の場合、こういう制約が必要な場合が多い。

一般的には、

```
instance 制約0 => クラス0 型0 where
  関数1 = 式1
  関数2 = 式2
  …
```

のようにキーワード instance のあとに制約とクラス名、型を書き、where というキーワードのあとに関数の定義を並べる。これは、制約$_0$ のもとで、型$_0$

は、クラス$_0$ のインスタンスであり、関数$_1$、関数$_2$ の実装はそれぞれ式$_1$、式$_2$ である、という意味である。制約は、一般に（クラス$_1$ 型変数$_1$，クラス$_2$ 型変数$_2$，…）という形式である。

ほとんどの型（例えばリストや組）は `Eq` クラスに属するが、関数型については、一般に 2 つの関数が等価であるかどうかを判定することは原理的に不可能なので、`Eq` クラスに属さない。

Q 3.4.1 組込みの `Bool` 型と等価なデータ型:

```
data MyBool = MyTrue | MyFalse
```

を `Eq` クラスのインスタンスとして宣言せよ。（なお `Bool` 型に対する `Eq` クラスのインスタンスは標準ライブラリー中で宣言されているので、宣言をプログラム中に書く必要はない。）

Q 3.4.2 以前の Quiz で定義したじゃんけん（`Janken`）型を `Eq` クラスのインスタンスとして宣言せよ。例えば `Rock == Rock` は `True`、 `Scissors == Paper` は `False` である。

Q 3.4.3 "`size :: a -> Int`" というメソッドを持つクラス `Sizable` クラスを定義せよ。例えば、

```
instance Sizable [a] where
    size xs = length xs

instance Sizable (Tree a) where
    size t = treeSize t
```

のようにインスタンスを宣言すると、 `size [1,5,2]` は `3` に、 `size tree4` は `4` になる。

3.5 その他の標準的な型クラス

Eq の他に実用上重要な型クラスとして、Ord, Show, Num などがある。（以下の説明用コードでは主要でないメソッドは省略している。）

```
class Eq a => Ord a where    -- Ord は order （順序）のこと
   (<), (<=), (>=), (>) :: a -> a -> Bool
   max, min             :: a -> a -> a
   -- …

class Show a where
   show                 :: a -> String
   -- …

class (Eq a, Show a) => Num a where
   (+), (-), (*)        :: a -> a -> a
   fromInteger          :: Integer -> a
   -- …

class Num a => Fractional a where
   (/)                  :: a -> a -> a
   -- …
```

Ord は不等号、Show は文字列への変換、Num と Fractional は四則演算のメソッドを定義している型クラスである。

クラス宣言の「=>」の左側にあるクラスは、スーパークラスと呼ばれる。

例えば、Eq は Ord のスーパークラスである。これは、例えば Ord クラスのインスタンスになる型は、必ず Eq クラスのインスタンスでもなければならない、ということを意味する。

> 少し余談になるが、型クラス宣言の文法を決めるときにスーパークラスを示す矢印は逆向きに（つまり `class Eq a <= Ord a where …` のように）するべきだった、という議論がある。これは `Ord τ` という制約が、`Eq τ` という制約を含意しているためである。恐らく、そのように定めるほうがすっきりしていたのだろうが、今さら一度決まった文法は変えられない。

実は、これまで触れていなかったが、1 や 3.14 などの数値リテラルは、Haskell ではそれぞれ、

```
1    :: Num t => t
3.14 :: Fractional t => t
```

という型を持っている。だから、例えば 1 という数値リテラルは、 Int としても、Double としても使用できる。

Show, Eq, Ord クラスなどに対するインスタンス宣言は、ほとんどのデータ型で必要になり、しかも同じような定義になるので、データ型の宣言が deriving /dɪ'raɪvɪŋ/ というキーワードを持っていれば、Haskell の処理系がこれらのインスタンス宣言を自動的に生成してくれることになっている。例えば Tree については次のように書く。

```haskell
data Tree a = Empty | Branch (Tree a) a (Tree a)
                            deriving (Eq,Ord,Show)
```

これで、Tree に対して期待どおりの (==), (>), show などのメソッドが定義される。

Q 3.5.1 組込みのリスト型と等価なデータ型

```haskell
data MyList a = MyNil | MyCons a (MyList a)
```

を、deriving を用いずに、Eq クラスと Ord クラスのインスタンスとして宣言せよ。Ord クラスのメソッドにはいわゆる辞書式の順序を用いよ。

> 辞書式順序とは、何かの列 $\mathbf{a} = a_1a_2a_3\cdots a_m$ と $\mathbf{b} = b_1b_2b_3\cdots b_n$ の順序を同じ位置の要素（例えば、a_1と b_1、a_2 と b_2、...）同士で、前から順番に比べていき、
>
> - ある j があって $i < j$ となるすべての i に対して $a_i = b_i$ かつ $a_j < b_j$ ならば $\mathbf{a} < \mathbf{b}$
> - $m < n$ で $i \leq m$ となるすべての i に対して $a_i = b_i$ ならば $\mathbf{a} < \mathbf{b}$
>
> となるようなものである。

（クラスの定義中にデフォルトの実装が定義されているので、Eq クラスの == メソッドと Ord クラスの <= メソッドだけを定義すれば、他のメソッドの定義は自動的に生成される。）

ヒント: 次のような、リストから MyList への変換を行う関数を定義して、

```haskell
toMyList :: [a] -> MyList a
toMyList []     = MyNil
toMyList (x:xs) = MyCons x (toMyList xs)
```

いくつかのケースをテストせよ。

```haskell
toMyList "abc" <= toMyList "abd"  -- True
toMyList "ab"  <= toMyList "abc"  -- True
toMyList "ab"  <= toMyList "a"    -- False
toMyList "ab"  <= toMyList "ba"   -- True
```

3.6 オブジェクト指向との関係

オブジェクト指向言語で使用される概念との関連について触れる。

オブジェクト指向を特徴づけるキーワードとして動的束縛 (dynamic binding)・　　(inheritance)・　　(encapsulation) の3つがよく挙げられる。

Q 3.6.1 左の語句の意味と対応する右の説明を線で結べ。

動的束縛	•	• クラスの実装の詳細を他のクラスから隠すことができること
継承	•	• 呼び出されるメソッドの実装がオブジェクトの実行時の型で決まること
カプセル化	•	• 上位クラスのメソッドの実装を下位クラスで再利用できること

カプセル化と継承は、ポリモルフィズムと動的束縛があってこそ意味がある概念である。ポリモルフィズムがあるから、実装の詳細を入れ替えても再コンパイルせずにコードを実行することができる。また、動的束縛があるから、継承したメソッドの意味がクラスに応じて自動的に変わり、同じようなメソッドを何度も定義する必要がなくなる。そのため、プログラミング言語の実装という観点から見れば、ポリモルフィズム、特に動的束縛こそがオブジェクト指向の本質であると言っても良い。

3.7 オブジェクト指向のクラスと代数的データ型

Haskell や ML といった関数型言語では、複数の構成子 (constructor)からなる代数的データ型 (algebraic data type) を定義できる。代数的データ型を構成する各構成子を、オブジェクト指向型言語のクラスに相当すると見なせば、関数型言語もアドホック多相や動的束縛に相当する機構を持っている。関数は、さまざまな構成子の引数を受け取り、また呼び出されるコードがパターンマッチングにより、オブジェクトの実行時の構成子で定まる。例えば、`Tree` 型の `isEmpty` という関数は `Branch` と `Empty` で実行されるコードが変わる。

オブジェクト指向言語のクラスと関数型言語の代数的データ型の構成子の違いは、拡張性の方向である。代数的データ型は、既存の構成子にそれを引数とする新しい関数を追加していくのは可能だが、既存の関数に新しい構成子を追加することはできない。（例えば組み込みの代数的データ型であるリスト型に `[]` と（`:`）以外の新しい構成子を後から追加することはできない。）逆にクラスは既存の関数（メソッド）を新しいデータ型（クラス）に定義することは可能だが、既存のデータ型（クラス）に新しい関数（メソッド）を追加することはできない。（例えば、Java の組み込みのクラスである `String` クラスに、新しいメソッドを後から追加することはできない。静的 (static) なメソッドなら追加できるが、動的束縛を伴わせることはできない。 Kotlin の拡張関数も動的束縛ではない。）

型クラスは、関数型言語にオブジェクト指向言語と同じ方向の拡張性（つまり既存の関数に新しい型を引数として追加する）を付与する仕組み、あるいはオブジェクト指向言語の動的束縛を関数型言語で解釈する仕組みと考えることもできる。

ただし、オブジェクト指向言語は新しいデータ型（クラス）を定義するときにしか、新しい関数（メソッド）を追加することはできない。これに対して型クラスは既存の型に新しい関数（メソッド）を定義することができる。オブジェクト指向言語でこれに同等なことを考えれば、既存のクラスに新しい（Javaの）インターフェースの実装を追加することに相当する。

3.8 型クラスとサブタイピング

オブジェクト指向言語では、あるスーパークラスを継承するクラスに属するオブジェクトは、一つの変数に代入したり、一つのコンテナーにまとめたりすることができる。しかし、Haskell では、例えば `Integer` と `Char` と `[Integer]` はすべて `Show` クラスに属するが、それらを一つのリストにまとめることはできない。つまり、次のような複数の型を持つ要素を含むリストは型付けできない。

```
-- 実際には部分式の [1,'a',[1,4,7]] が型付け不可なので
-- 全体の impossible も型付け不可
impossible :: [String]   -- とはいかない
impossible = map show [1,'a',[1,4,7]]
```

一般的には、同じクラスに属していても、同一のコンテナーに収納することはできない。

しかし、`Show` の場合、`String` に変換した結果がわかれば良いのだから、次のようなリストなら構成することができる。

```
possible :: [String]
possible = [show 1, show 'a', show [1,4,7]]
```

つまり、クラスに応じた適切な変換を適用すれば、一つの型に変換して、同一のコンテナーに収めることができる場合がある。

もう少し一般的な例として、次のような型クラスを考える。

```
class C a where
  foo :: a -> Int -> Bool
  bar :: a -> a
  baz :: a -> Char -> a
```

これに対して、次のような型と変換関数を考えることができる。

```
data T = T (Int -> Bool)  -- foo の型に対応
           T              -- bar の型に対応
           (Char -> T)    -- baz の型に対応

instance C T where
  foo (T f g h) n = f n
  bar (T f g h)   = g
  baz (T f g h) c = h c

toT :: C a => a -> T
```

```
11 toT a = T (foo a) (toT (bar a)) (\ c -> toT (baz a c))
```

構成子 T の 3 つの構成要素の型、 Int -> Bool と T と Char -> T は、それぞれ foo :: C a => a -> Int -> Bool と bar :: C a => a -> a と baz :: C a => a -> Char -> a に対応している。下線部の型の中の型変数 a を自身の型 T に置き換える。

このとき、a が C のインスタンスの型を持つならば、 a と toT a :: T は、foo, bar, baz に対して、どれも同じように振る舞う。さらに、 a, b, c がすべて C のインスタンスだが、異なる型に属していたとしても、toT a, toT b, toT c はどれも T 型であり、一つのリストにまとめることができる。

一般的に型クラス

```
class C α where
    m1 :: α -> τ1
    m2 :: α -> τ2
    m3 :: α -> τ3
```

の各メソッドの戻り値の型 τ_1, τ_2, τ_3 に、対象の型変数 α が引数の型の位置（つまり -> の左側に）に現れないのならば—戻り値の型の位置に現れるのは構わない—変換先の型の T 型:

```
data T = T σ1 σ2 σ3
```

（ただし σ_i は τ_i 中の α の出現を T で置き換えた型である。） と $toT :: C\ \alpha \Rightarrow \alpha \to T$ のような変換関数を定義することができる。変換したオブジェクトは元のオブジェクトと同じ振る舞いをし、単一の型なので一つのコンテナーにまとめることができる。この手法については、参考文献 (Odersky 1991) に詳しい説明がある。

3.9 （参考） 型クラスの問題点

型推論とアドホック多相をうまく両立した型クラスだが、いくつかの問題点も残っている。

わかりにくいエラーメッセージ

型クラスはエラーメッセージがわかりにくい、またはエラーになってほしいものがそもそもエラーにならない、という欠点がある。参考文献 (Heeren & Hage 2005) にいくつか例が挙げられている。

曖昧さ (ambiguity)

例えば、次のようなクラス定義があるとする。これは実際の Show, Read クラスを少し単純化している。

```
1 class Show a where
2   show :: a -> String
```

```
3
4 class Read a where
5   read :: String -> a
```

この定義の元で、次のような関数を定義する。

```
1 foo x = show (read x)
```

この関数の型は、`foo :: (Show a, Read a) => String -> String` となる。型変数 a の型は、=> の左辺にしか現れないので、型推論で決定できない。（すなわち曖昧である。） すると次のプログラムのように、

```
1 foo x = show (read x :: Integer)
```

a の具体的な型をユーザが明示しなければ意味が定まらなくなってしまう。（プログラムの意味が型宣言に依存することになり、気持ちが悪い。）

3.10 （参考） Dictionary-Passing Style 変換

ここからは、Haskell が型クラスをどのように実装しているかを説明する。（ただし、このような実装方法が Haskell の仕様に定められているわけではない。あくまでも良く使われる実装方法の一例である。）

クラス宣言・インスタンス宣言や制約された型（... => ...）を持つ関数は、コンパイル時にそれらを用いない普通の関数やデータの定義に書き換えられる。

まず、クラス宣言はメソッドを要素に持つような型の宣言とアクセサーの定義に翻訳される。例えば `Eq` クラスの場合、

```
1 type Eq' a =
2
3 eq' :: Eq' a -> (a -> a -> Bool)
4 eq' = \ (e, _) -> e          -- (==) に対応する
5
6 ne' :: Eq' a -> (a -> a -> Bool)
7 ne' = \ (_, n) -> n          -- (/=) に対応する
```

この `Eq' a` 型のようなオブジェクトは一般に_____(method dictionary)と呼ばれる。

インスタンス宣言は具体的な型を持つメソッド辞書の定義に翻訳される。例えば、`instance Eq Direction` は次のように `Eq' Direction` 型のオブジェクトの定義になる。

```
1 eqDirectionDic :: Eq' Direction
2 eqDirectionDic = (eqDirection,
3                   \ a b -> not (a `eqDirection` b))
4   where North `eqDirection` North = True
5         South `eqDirection` South = True
6         West  `eqDirection` West  = True
7         East  `eqDirection` East  = True
8         _     `eqDirection` _     = False
9
```

```haskell
eqTreeDic :: Eq' a -> Eq' (Tree a)
eqTreeDic (eqA, _) = (eqT, \ a b -> not (eqT a b))
  where
    Empty `eqT` Empty  = True
    Branch l1 n1 r1 `eqT` Branch l2 n2 r2
      = l1 `eqT` l2 && n1 `eqA` n2 && r1 `eqT` r2
    _      `eqT` _      = False
```

そして型クラスを使っている（… => … という型を持つ）関数の定義は、コンパイル時に次のようにメソッド辞書（`Eq' a` 型）を追加の引数とする関数の定義に書き換えられている。メソッド辞書には、通常は関数が含まれるので、これらの関数は高階関数になる。

```haskell
member' ::        -> a -> [a] -> Bool
member' d x []      = False
member' d x (y:ys) = eq' d x y || member' d x ys

subset' ::        -> [a] -> [a] -> Bool
subset' d xs ys = all (\ x -> member' d x ys) xs
```

これらの関数の呼出しは次のように型に応じて具体的なメソッド辞書を渡される形に書き換えられる。

```
member North [North, South, West]
  ↦ member'                  North [North, South, West]
subset [North, South] [North, South, West]
  ↦ subset'                  [North, South]
                             [North, South, West]
```

このような書き換え（Dictionary-Passing Style 変換）は Haskell ではコンパイル中の型推論時に自動的に行なわれる。つまり、アドホック多相の実行時のコストは、辞書オブジェクトの中から関数を取り出し、それを起動するだけになる。要するに、アドホック多相は高階関数で解釈できる（ただし高階関数になるので、最適化が難しくなってしまう可能性はある。）。動的に（つまり実行時に）メソッドを探しているように見えて、実際には静的に（コンパイル時に）ほとんどの必要な処理が済んでいる。

問 3.10.1 次のように定義されている関数 `lookup`:

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

lookup :: Eq a => a -> [(a, b)] -> Maybe b
lookup x ((n,v):rest)
   = if n == x then Just v else lookup x rest
lookup x [] = Nothing
```

（`lookup` は標準ライブラリーに定義済みの関数である。）の Dictionary-Passing Style 変換後の形 `lookup'`:

```haskell
lookup' :: Eq' a -> a -> [(a, b)] -> Maybe b
```

を示せ。例えば、`lookup 1 [(1,2),(4,7)]` と `lookup' eqIntDic 1 [(1,2),(4,7)]` の値が、あるいは `lookup 1.1 [(1.4,0.1),(4.2,6.9)]`

と `lookup' eqDoubleDic 1.1 [(1.4,0.1),(4.2,6.9)]` の値が同じ値になる。（`eqIntDic`, `eqDoubleDic` は各自で適宜定義する。）

```
lookup' :: Eq' a -> a -> [(a, b)] -> Maybe b
lookup' d x ((n,v):rest)
    = if           then Just v else
lookup' d x [] = Nothing
```

3.11 （参考）他のオブジェクト指向言語について

一般的なオブジェクト指向言語でも、たいていは"メソッド辞書"に対応するデータを扱うことで、動的束縛を実現していると考えられる。つまり、"隠れた"高階関数である。しかし、関数の独立した引数としてではなく、オブジェクトに付随する形になっていることが多いと思われる。つまり、各オブジェクトがクラスに対応するデータ構造へのポインターを持っていて、クラスに対応するデータ構造がメソッドの辞書を含んでいるという場合が多い。 Smalltalk のメソッド呼出しの実装の方法は、例えば参考文献 (Guzdial & Rose 2003) の第 6 章に説明されている。

一方、代数的データ型の場合は、メソッド辞書に相当するものは各関数に付随しているかたちになる。

JavaScript は、クラスではなく_____(prototype) という概念に基づくオブジェクト指向を採用している。（ちなみにプロトタイプ方式を最初に広めた言語は Self という言語である。）JavaScript のメソッド呼出 しの仕組みは参考文献（久野 2001）の第 6 章に解説がある。ただし、最近の JavaScript はクラスも採り入れている。

Common Lispのオブジェクト指向拡張 (CLOS) は多重メソッド (multi-method) と言って、他の多くのオブジェクト指向言語と異なり、2 つ以上のパラメーターの型（クラス）によって実際に呼出すメソッドの実装を決定する仕組みを持っている。

問 3.11.1 実際のオブジェクト指向言語（Smalltalk（Guzdial & Rose 2003）, CLOS, JavaScript（久野 2001）, C++, Java（Lindholm & Yellin 2001）, Python, Ruby など）で動的束縛や継承がどのように実装されているか調べよ。

3.12 まとめ

型クラスは、Haskell でアドホック多相を可能にするための仕組みである。既存の関数を新しいデータ型に拡張するための仕組みでもある。コンパイル時に高階関数への変換が行なわれるので、実行時のコストはほとんどかからない。しかし、わかりにくいエラーメッセージなど、いくつかの問題は残っている。

3.13 さらに詳しく知りたい人のために...

文献（Wadler & Blott 1988）は、型クラスのアイデアを最初に紹介した論文である。文献（Hall et al. 1996）は、現在の Haskell の型クラスを詳しく説明している。文献（Jones 2000）は、Haskell の（型クラスに関する部分を含む）型推

論を、具体的に Haskell のプログラムを用いて説明している。文献（Odersky 1991）は、オブジェクト指向言語のサブタイプと同様の効果を、関数型言語の枠組みで得るための手法を提案している。文献（Heeren & Hage 2005）は、型クラスのエラーメッセージの問題点と改良法について詳しく述べている。

（Wadler & Blott 1988） Philip Wadler and Stephen Blott, “How to make *ad-hoc* polymorphism less *ad-hoc*” Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, pp. 60–73, 1988 年 10 月

（Hall et al. 1996） Cordelia Hall, Kevin Hammond, Simon Peyton Jones and Philip Wadler, “Type Classes in Haskell” ACM Transactions on Programming Languages and Systems 18 巻 2 号, pp. 109–138, 1996 年

（Jones 2000） Mark P. Jones, “Typing Haskell in Haskell” https://web.cecs.pdx.edu/~mpj/thih/, 2000 年 11 月

（Odersky 1991） Martin Ordersky, “Objects and Subtyping in a Functional Perspective” IBM Research Report RC 16423, 1991 年 1 月

（Heeren & Hage 2005） Bastiaan Heeren and Jurriaan Hage, “Type Class Directives” Seventh International Symposium on Practical Aspects of Declarative Languages (LNCS 3350), pp.253–267, 2005 年 1 月

（Guzdial & Rose 2003） Mark Guzdial and Kim Rose 編、 軋音組 訳 「Squeak 入門 過去から来た未来のプログラミング環境」 2003 年 3 月 星雲社

（久野 2001） 久野 靖 「入門 JavaScript」 2001 年 8 月 ASCII

（Lindholm & Yellin 2001） Tim Lindholm and Frank Yellin 著、村上 雅章 訳「Java 仮想マシン仕様 第2版」 2001 年 5 月 ピアソン・エデュケーション
