プログラミング言語論2012 # 6 — 関数型言語

久野　靖*

2012.5.17

1　関数型言語

1.1　関数型言語とは

今回は最終回ということで、関数型言語を取り上げます。といっても Lisp ではありません。Lisp はあまり「関数型言語」とは呼ばれませんし…。

そもそも関数型言語とは何でしょうか。素直な定義としては、「関数とその適用に基盤を置く言語」ということになります。しかし今日的にはむしろ、「副作用の無い言語」という定義の方がぴったりする感じがします。

副作用が無い、とはどういうことでしょうか。たとえば「$f(x) = x^2 + 3x - 5$」のような数式で定義された関数を考えてみましょう。この関数の結果は、引数として与えた x の値が同じであれば何回評価しても同じであり、また、部分式の評価順序がどうであってもその結果が変わることはありません。このような性質のことを「参照透過 (referential transparency)」と呼びます。

参照透過性を失わせるものは何かというと、評価に際して環境に何らかの変更がなされ、その変更が引き続く評価の結果を最初とは違うものとさせるようなこと、つまり「副作用」なわけです。ですから、「参照透過性のある言語」と「副作用の無い言語」は同じ意味だということになります。

参照透過性を持つ言語には、次のような利点があります。

- 関数の返す結果は引数のみによって定まるので、テストや検証が容易である。
- 同じ記述は常に同じ結果になるので、記述や読解が容易である。
- 並列評価や遅延評価など、評価の順番を変更する手法が取り入れやすい。これによって性能や記述性の向上が見込まれる。

このような利点にもかかわらず、参照透過な言語が普及してこなかったのには一定の理由があります。それは、典型的な副作用は「変数への代入」であり、これまでの言語では代入があまりにも基本的な要素として言語に組み込まれてきたため、それを除外するということが真剣に検討されなかった、ということだと思います。

しかし一方で、既存の言語でもその記述の一部を参照透過であるように (つまり副作用が無いように) 書くことで、上のような利点が (一部) 得られますし、そのようなスタイルは推奨されています。ただ、既存の言語では副作用の有無を言語上で明確に区分しないので、どの部分が参照透過でどの部分が参照透過でないかは見ただけでは分からないという弱点がありました。

近年の関数型言語では、このような問題を克服するために、副作用の起こる場所を記述上で限定する方向に進んでいます。たとえば ML では書き換えが行える変数はすべて「`ref` T」という特別な型を持つ必要がありますし、オブジェクト指向と関数型のハイブリッドとして使用が増えている Scala では書き換え可能な変数は `var`、書き換えできない (副作用を持たない) 定数は `val` という宣言で記述するようにして、これらを明確に区分しています。

*経営システム科学専攻

1.2 関数型言語の歴史

関数型言語の一番大元は、既に学んだとおりラムダ計算とそこから派生した Lisp 言語だと言えるでしょう。ラムダ計算そのものは確かに副作用を持たない参照透過な計算モデルですが、Lisp 言語の方は既に述べたように、`setf` やベクタなどのオブジェクトの書き換えは基本的な操作として存在し続けており、このことから Lisp は関数型ではないとされるわけです (適用型 — applicative — という言い方がよくなされます)。

純粋な関数型言語へのステップとして大きなものは、John Backus (世界最初の高水準言語である FORTRAN や BNF を発明した人) が 1977 年の Turing 賞 (コンピュータサイエンス分野でのノーベル賞に相当) 受賞記念講演において提案さいた FP 言語です。この言語は「副作用がない」ことを大前提として設計されていましたが、副作用がないと入出力もできない、という問題に直面し、「トップレベルは特別」としてこの問題を回避していました。またもともと、この言語は実験的な提案であり、実用性はありませんでした。

これとは並行して、Robin Milner が Edinburgh LCF プロジェクト (自動証明系のプロジェクト) において当初証明の道筋を関数の形で記述するメタ言語として設計された ML 言語が 1975〜1976 年ごろに実装され、1980 年代になって広く知られるようになり、多くの処理系や、拡張言語の基盤となりました。この系列の代表的な言語仕様としては、SML (Standard ML)、Miranda、CAML、OCAML (Objective CAML)、F ♯などがあります。

ML 系列の言語は参照透過を基本とはしていましたが、上で述べたように書き換え可能な変数を含んでいるため、「純粋な (pure)」関数型ではない、とも言われます。これに対して純粋な関数型をめざした研究が行われ、Haskell、clean など多くの言語が提案されました。その中でも今日では Haskell が最も多くのユーザを持つ言語となってきています。Haskell 言語の特徴としては、次のものが挙げられます。

- 純粋な関数型 (pure functional)
- 強い型付け、ただし型推論により型宣言は大幅に省略可能
- 遅延評価をすべてに適用
- 副作用は IO モナドと呼ばれる形で統一して扱う
- 並行プログラムにも複数の機能を通じて対応

そういうわけで今回は、Haskell 98 仕様に基づく Haskell 言語を題材に、純粋関数型言語のもつ特徴や面白さを体験して頂こうと思います。

2 Haskell 言語入門

2.1 Hugs98、関数定義、ラムダ式、中置記法

ここでは Haskell 98 言語仕様を実装した Hugs98 と呼ばれるインタプリタ型処理系を使っています。ほかにより高速な GHC (Glasgow Haskell Compiler) と呼ばれる処理系も広く使われています。

```
% hugs
...
Hugs> :load test1.hs  ←定義ファイルをロード
Main>                 ←この状態で関数を実行してみられる
...
Main> :quit           ← Hugs 終了
%
```

このように、プログラムはファイルに記述しておき、それを読ませてから実行開始します。では、test1.hs というファイルに色々な例題を追加しながらロードして試して行くことにしましょう。まず最初の例はこれです。

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

これは、2 つの引数 x と y を受け取り、その和を返す関数 add を定義しています。実行してみましょう。

```
Main> :load test1.hs
Main> add 1 2
3
Main> :type add
add :: Num a => a -> a -> a
```

確かに動きますが、:type というのは? 上述のように、Haskell ではすべての式は強く型付けされているので。 「:type 式」によりその型を調べることができます。上の表示は「ある型 a は Num の一種であり、add は a -> a -> a 型である」と読みます。型の読み方はすぐ後で説明します。
今は関数を定義しましたが、定義しないで直接関数を書くこともできます (ラムダ式)。

```
Main> (\x -> x + x) 2
4
```

なお、Haskell では関数名を ‘...‘で囲むと中置記法の演算子として使えます。逆に中置演算子は (...) で囲むことで、普通の関数名と同じに扱えます。

```
Main> 1 ‘add‘ 2
3
Main> (+) 1 2
3
```

2.2 カリー化

では、型の読み方の説明です。「a -> a」は、「a 型を引数として受け取り、a 型を結果として返す関数」を意味しています。そして、すべての関数は引数は 1 つです。複数の値を組にした型タプルを使えば「2 つの値を取る」「3 つの値を取る」などのように書くこともできますが、普通はそうはしません。
では、先程の add や (+) はどうなのでしょう? 「a -> a -> a」は、「a を引数として受け取り、a -> a を返す」というふうに読みます。次の例を見てください。

```
add1 = add 1
add2 = (+) 2
```

つまり add に 1、(+) に 2 を与えた結果も 1 つの関数であり、その関数にさらに数を与えると、結果として数が返される、ということになるわけです。

```
Main> :type add1
add1 :: Integer -> Integer
Main> add1 2
3
```

このように、「N 引数の関数に 1 つの引数を与えて、残りの $N-1$ 引数を受け取って結果を返すような関数を得る」ことを「部分適用」と呼びます。そして、この考え方を適用してすべての関数を 1 引数として扱うことを「カリー化」と呼びます。この名前はコンビネータ理論で著名な数学者 Haskell Curry に由来しています (そして Haskell 言語も同じく)。カリー化についてはこの後もたびたび出て来ます。

なお、今度は `add1` の引数や返値の型は `Integer` になっていますね。これは「1 と足す」ことまで分かったから整数だと分かるためです。このように、演算や引数の値によって式の型を推論することを型推論 (type inference) と呼び、ML 以降の関数型言語の重要な特性となっています。型推論のおかげで、いちいち繁雑な型宣言を書かなくても、強い型検査によるエラー検出が行えるからです。しかし、あまりにも型を書かなすぎると、どのような型推論が行われているのか分からなくなって、エラーが出たとき理由が推測困難になります。そこで以下では、型検査と記述の手間を勘案して、「関数の型」は宣言し、中で使う変数は宣言しないことにします。

なお、ちょっとだけ部分適用の話題に戻りますが、中置記法を使えば第 2 引数を部分適用することもできます。

```
sub x y = x - y
sub1 = (`sub` 1)  ← (? `sub` 1) の意味
```

2.3 再帰関数

さて、段々関数型らしい話題に進みましょう。関数型言語では、変数の書き換えがないため、繰り返しが必要な場合にはすべて再帰を使います。これは、変数を書き換える変わりに新たな呼び出しによって別のインスタンスを作るためです。階乗の例を見てみましょう。

```
fact1 :: Integer -> Integer
fact1 n
  | n > 0     = n * fact1 (n-1)
  | otherwise = 1
```

このように、関数定義の中で条件による枝分かれをする場合は、「`|` に続いてガード条件を記述する行を連ねることで本体の振り分けができます。ちなみに、Haskell ではソースコードの字下げに意味があり (他の関数型も多くはそう、手続き型では Python が著名)、字下げのより大きい行は上の行の内側になることを意味します。

ところで、上記の素朴な階乗の定義だと、下請けの `fact` を呼んだ後戻って来てから `n` を掛けるので、再帰 1 回ごとに中間結果を保存する必要があります。そこで、次のような別バージョンを見てみます。こちらは補助関数を `where` を使って定義し、それを呼び出しています。

```
fact2 :: Integer -> Integer
fact2 n = f2sub 1 n
  where
    f2sub r n
      | n > 0     = f2sub (n*r) (n-1)
      | otherwise = r
```

このように「最後に自分自身を呼びその結果を自分の結果とする」場合、実装では新しい引数を持って関数の先頭にジャンプするだけで済みます。このような最適化を TCO(tail call optimization) と呼びます。TCO を意識して記述する必要があるというのは、ちょっとワザを強制されている感じもしますが、効率のためには重要なことです。

2.4 リストデータ構造

次は、Lisp 以来、関数型言語で広く使われるデータ構造であるリストについて見てみます。Haskell では「:」がリストの「先頭」と「残り」をつなぐ構成子 (つまり Lisp の cons) に相当します。だから、[1,2,3] == 1:2:3:[] ということになります。

そこで素朴には、リストを扱う関数は次のように再帰を使ってリストを分解していくことで書けるわけです。

```
addlist1 :: [Integer] -> [Integer]
addlist1 (x:xs) = x+1:addlist1 xs
addlist1 []     = []
```

なお、このコードでは引数部分にパターンを記述しており、パターンマッチによる枝分かれが関数本体の枝分かれということになります。

```
Main> addlist1 [1,2,3]
[2,3,4]
```

しかし実は上の例のような記述は Haskell のコードとしてはいまいちです。というのは、「1 足す」という仕事と「リストにそれぞれ適用」という仕事は別の概念であるので、それぞれを別に作成してから組み合わせた方が汎用性があるからです。

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

addlist2 :: [Integer] -> [Integer]
addlist2 = mymap (+ 2)
```

ここでは mymap の型は「ある型 a を受け取り a を返す関数と、a の並んだリストとを受け取り、a の並んだリストを返す」と読みます。そしてこの a のところに任意の特定の型が入るわけです。これはちょうど、前にやった Generics の型パラメタ機構と同様ですね。そしてこれに「2 足す」関数を与えるので、できあがる関数は「整数のリストを受け取り、整数のリストを返す」型になります。

```
Main> addlist2 [1,2,3]
[3,4,5]
```

このように書いて来ましたが、実は Haskell にはこれと同じことをする標準関数 map が用意されています。ですから、プログラマが実際に書くのは「addlist2 = map (+ 2)」だけでよいのです。

さて、同様のものとして「リストから特定の条件のものを選別」することを考えてみます。

```
myfilter fn (x:xs) =
  | fn x      = x:myfilter fn xs
  | otherwise = myfilter fn xs
myfilter fn []     = []

larger1 = myfilter (> 1)
```

こんどはガード振り分けとパターンマッチの両方が使われていますが、あとは既に学んだことと同様ですね。

```
Main> larger1 [0,1,2,3]
[2,3]
```

そして実は、これも標準関数 filter が用意されているので「larger1 = filter (> 1)」でよいわけです。

では map(変換)、filter(選別) ときたので、次は「合計」のようなものはどうしたらよいか考えてみましょう。これには、「2 項演算」「初期値」「リスト」の 3 つを受け取る関数 foldr と foldl が使えます。これらは次のように働きます。

```
j2 foldl (+) 0 [w,x,y,z] = ((((0+w)+x)+y)+z)
j2 foldr (+) 0 [w,x,y,z] = (w+(x+(y+(z+0))))
```

足し算の場合はどちらでも結果は同じですが、引き算の場合は結果は当然変わってきます。そして型も少しだけ違います。

```
:type foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
:type foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

合計の場合は a も b も Integer ですから、どちらでも型は同じです。

```
mysum :: [Integer] -> Integer
mysum = foldr (+) 0
```

```
Main> mysum [1,2,3,4]
10
```

このようにどちらでもよさそうに見えますが、リストの演算子: が右結合なので、それと対応した foldr の方が効率が良いとされます。

2.5 リストの応用編

では今度は、「1 から 10 までの奇数の自乗和を求めよ」という例題を考えてみます。まず。「1 から 10 までのリスト」は次の記法で作ることができます。

```
Main> [1..10]
[1,2,3,4,5,6,7,8,9,10]
```

次に、関数の合成を表す演算子「.」を活用します。一般に「(f (g x)) = (f.g) x」により関数が合成できます。そして、「奇数のものを抜き出し、自乗して、合計」により求めるものが得られるはずです。

```
oddsqsum :: [Integer] -> Integer
oddsqsum = (foldr (+) 0).(map sq).(filter evenp)
  where
    evenp x = mod x 2 /= 0
    sq x = x * x
```

```
Main> oddsqsum [1..10]
165
```

このように「引数に言及せず関数だけ合成して行く」やり方を「pointless な記法」と呼びます。このように関数を加工していくことで必要な関数を得るのが「関数型らしい」スタイルだと言えます。

さて次に、他の言語にもある機能ですが、「リスト内包表記 (list comprehension)」について説明しておきます。これは次の形をしたもので、指定したリストの要素を組み合わせ、条件で filter して新しいリストを作れます。

[式|変数<-リスト，変数<-リスト，...，条件]

たとえば、「前者が後者で他方で割り切れるような 1〜5 の整数の組」は次のようになります。

```
Main> [(a,b)|a<-[1..5],b<-[1..5],mod a b == 0]
[(1,1),(2,1),(2,2),(3,1),(3,3),(4,1),(4,2),(4,4),
(5,1),(5,5)]
```

これを使うことで、クイックソートは次のように非常に短く書けます。ただし「++」はリストどうしの連結演算です。

```
qsort :: [Integer] -> [Integer]
qsort (x:xs) = (qsort [a|a<-xs,a<x])++[x]++(qsort [a|a<-xs,a>x])
qsort [] = []
```

ところで、上では qsort の対象を「整数のリスト」としていましたが、整列は「順序があるもの」のリストなら何でもいいはずです。つまり「演算子 `> :: a -> a -> Bool` を持つような型 a」であれば (かつ、この演算子が全順序を構成していれば)、いいわけです。

Haskell では、このように「ある型についてこういう関数を持つ」ということを表現するのには「型クラス」と呼ばれる機構を使用します (これはオブジェクト指向のクラスとはまったく別のもので、どちらかというと Java のインタフェースに近いです)。そして、自分で定義することもできますが、既にある型クラスを使うことの方が多いです。ここの例では、「順序を持つ値」は型クラス Ord で表すので、これを用いて型宣言を次のように変更します。

```
qsort :: Ord a => [a] -> [a]
```

```
Main> qsort "How are you?"
" ?Haeoruwy"
```

なぜこうなるかというと、実は「文字列」は「文字のリスト」なので、文字列が文字コード順に並べ直され、重複する文字は削除された結果、このようになった訳です。

Ord の他によく使われる型クラスとしては、「等しい演算==を持つ」Eq、「文字列に変換できる関数 show を持つ」Show などがあります。

2.6 遅延評価

さて、上で「[1..n]」のような記法を紹介しましたが、Haskell ではさらに、「[1..]」などで「無限のリスト」を作ることもできます。もっとも、これを直接打つと無限に打ち出し始めて止まらなくなりますが…でも、「先頭から N 個取る」関数 take と組み合わせれば問題なく扱えます。

```
Main> take 10 [1..]
[1,2,3,4,5,6,7,8,9,10]
```

さらに、無限リストに対して演算することも問題なくおこなえます。

```
Main> take 10 (map (* 2) [1..])
[2,4,6,8,10,12,14,16,18,20]
```

一般に、通常の言語では、関数 (メソッド) 呼び出し時にパラメタを値に評価します。これを「正格評価 (eger evaluation)」と呼びます。しかし Haskell では、関数適用は「そのままの形」で覚えられていて、実際にその値が必要になった時にはじめて評価 (実行) されます。これを「遅延評価 (lazy evaluation)」と呼びます。このため、無限リストに対する演算が書かれていても、その各要素が取り出される時まで演算は起こらないわけです。

たとえば、「フィボナッチ数から成る無限リスト」を計算する関数 fib は次のように書けます (引数がないので、->がなく結果の型だけになることに注意)。

```
fib :: [Integer]
fib = fibx [] 1 1
  where
    fibx l a b = l ++ (fibx [a] b (a + b))
```

ここで、連結演算 (++) の右側の fibx の呼び出しは、必要になった時に行われることに注意。なお、一度呼び出されて計算されたら、その値は覚えて置かれますから、何回も同じ値を計算することはありません。

```
Main> take 10 fib
[1,1,2,3,5,8,13,21,34,55]
```

遅延評価の利点としては、次のものが挙げられます。

- 再帰を書くのに枝分かれが要らない (先の fib の例) ので簡潔である。
- 「いくつまで用意するか」を考えないでよい (無限に用意して使う時に必要なだけ取るようにすればよい)。
- きっちり必要なだけしか計算しないで済む。

一方で、次のような弱点もあります。

- 普通の人は慣れていない。
- いつ実行されるか予測しづらい。

しかし、純粋な関数型言語であれば副作用がないので、いつ実行されても結果は同じであり、このことは問題にならない「はず」です。

2.7 データ型の定義

ここまでは既にある型だけを使って来ましたが、既存の型をもとに新しい型を定義する方法である「data 定義」を紹介します。これは代数的データ型の定義と呼ばれます (他にもいくつか型を定義する方法がありますがここでは省略。以下は、2 分木データを定義しています。

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
  deriving (Show, Eq)
```

すなわち、保持するデータは型パラメタ a で表し、型クラス Show と Eq に従うことで標準的な show と (==) を自動的に生成してもらいます。
ここで、データを 1 つだけ持った木を生成してみましょう。

```
newtree :: a -> Tree a
newtree x = Node (Leaf) x (Leaf)
```

では次に、木への新たなデータの挿入も行ってみます。ここで副作用がないため、instree はデータを挿入した後の新しい木を返すことに注意。

```
instree :: Ord a => a -> Tree a -> Tree a
instree x (Leaf) = newtree x
instree x (Node l y r)
  | x < y = Node (instree x l) y r
  | x > y = Node l y (instree x r)
  | otherwise = Node l y r
```

実際に動かしてみます。

```
Main> instree 2 (instree 5 (newtree 3))
Node (Node Leaf 2 Leaf) 3 (Node Leaf 5 Leaf)
```

ところで、これをやっていると「(...)」が沢山必要になって読みにくくなります。ここで、「$」を使うとそこから行末まですべてを「(...)」で囲んだのと同じ効果があるので、平らになって読みやすくできます。

```
Main> instree 2 $ instree 5 $ newtree 3
Node (Node Leaf 2 Leaf) 3 (Node Leaf 5 Leaf)
```

では次に、木の中にめざすデータがあるかどうか検索する関数も書いてみましょう。

```
searchtree :: Ord a => a -> Tree a -> Bool
searchtree x (Leaf) = False
searchtree x (Node l y r)
  | x < y     = searchtree x l
  | x > y     = searchtree x r
  | otherwise = True
```

これも動かしてみます。

```
Main> searchtree 3 $ instree 2 $ instree 5 $ newtree 3
True
Main> searchtree 4 $ instree 2 $ instree 5 $ newtree 3
False
```

2.8 IO モナド

それではいよいよ、純粋な関数型言語最大の問題「副作用がないのに、どうやって入出力とかするの?」を Haskell でどのように解消しているかについて説明します。入出力が副作用だという話はよいですね? (C 言語で getchar() を 2 回呼んだら別の文字が取れて来るということは、副作用を起こしていることに他なりません。同様に、同じ文字を 1 回出力するのと 2 回出力するのは人間にとって明らかに違います)。文字は取り

この問題に対する Haskell の解答は「世界を関数の引数の一部として渡し、世界を返してもらう」というものです。このために「IO a」という型を使います。これが「世界の状態ともに型 a の値を持っている」ことを表しています。

実は「IO a」は Monad という型クラスに属していて、この型クラスは次の 2 種類の演算 (関数) を定義しています。

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

とりわけ重要なのは (>>=) でして、これは「ある世界の状態での a の値と、a の値を処理してある世界の状態での b の値を返す関数とを組み合わせることで、処理を実行した後の世界の状態での b の値を返す」演算子です。return については後で説明しましょう。

そういうわけで、Haskell では入出力を行う関数などはすべて、結果が IO a 型になっています。

```
Main> :type getLine
getLine :: IO String
Main> :type putStr
putStr :: String -> IO ()
```

なお、型「()」は unit と読み、「何か値を渡す必要があるが値の内容は何もなくていい」時に使います。その値もやはり「()」です。そして、Haskell のメイン関数は「IO ()」を返す関数を実行できる (=副作用を持つ動作を実行できる) 能力があります。例題として、「2 つの文字列を読み込み、逆順に連結して出力」するものを書いてみます。

```
test1 :: IO ()
test1 = putStr "str1? "
        >>=
        \x1 -> getLine
        >>=
        \x2 -> putStr "str2? "
        >>=
        \x3 -> getLine
        >>=
        \x4 -> putStr (x4 ++ x2)
        >>=
        \x5 -> return ()
```

このコードは、次のように読んでください。

- まず、「str1?」と表示した後の IO 型ができます。結果は IO() 型です。
- ここに、() を受け取って次の動作として 1 行読み込む関数を「>==」で連結動作させます。その結果は IO String」型であり、読み込んだ文字列が包まれています。
- ここに対して、文字列を引数とし、残りの処理を行う関数 (パラメタ名は x2) を「>==」で連結動作させます。そうすると、この演算子によって読み込んだ文字列が x2 に渡されて動作本体が評価されます。
- すると、次はまた「str2?」と表示した後の IO 型ができ、結果は IO () になります。
- ここに、() を受け取って次の動作として 1 行読み込む関数を「>==」で連結動作させます。その結果は IO String」型であり、読み込んだ文字列が包まれています。
- ここに対して、文字列を引数とし、残りの処理を行う関数 (パラメタ名は x4) を「>==」で連結動作させます。そうすると、この演算子によって読み込んだ文字列が x4 に渡されて動作本体が評価されます。
- その中では、x4 ++ x2 を出力する動作を行い、IO () を返します。

このように、次々に「前の結果を受け取って次の結果を生成する」連鎖としていくことで、順番に実行することが可能になり、なおかつ途中で受け取った結果を後の方で参照することもできます。このプログラムを動作させると当然、次のようになります。

```
Main> test1
str1? aaa
str2? bbb
bbbaaa
```

さて、この「>>=」でつなげて書くのは書きづらくよみづらいので、これと同じことをもっと短く書ける do という構文が用意されています。Haskell の入門書ではこちらをまず習うことが多いと思います。

```
test2:: IO ()
test2 = do
        putStr "str1? "
```

```
    s1 <- getLine
    putStr "str2? "
    s2 <- getLine
    putStr (s2 ++ s1)
    return ()
```

さて、ここまで読んで来て「Haskell は副作用のない言語である」というのは本当なのかどうか? という気持ちになったかと思います。より正確には、「`IO a` を使うことで副作用のある箇所を明示/分離している」という方がよさそうです。

つまり、ほとんどの場所は IO a を使わずに書ける副作用のない世界であり、「少量の副作用のある世界から、大量の副作用のない世界を呼び出してやりたい仕事をこなす」と考えるのがいいでしょう。

2.9 Haskell のまとめ

関数型プログラミング言語 Haskell について、その特徴を改めて整理すると次のようになるかと思います。

- 高階関数を駆使するなどにより記述がコンパクト
- 強い型検査、ただし型宣言はしなくても型推論が補ってくれる
- 参照透過性 — ある式を評価した結果は常に同じ
- 遅延評価 — 無限データ構造なども作れる。実際に値を計算するのは「使われた時」でよい
- 副作用のある部分は明確に分離して記述 (型が違うので分かる)

普通の言語とはかなり「違う世界」でしたが、どうだったでしょうか。

演習 X Haskell ならではの特徴を活かしたプログラムを何か作ってみなさい。