第3章　関数型言語 Haskell とは

Haskell は＿＿＿＿＿＿＿＿（空欄 3.0.1）と呼ばれ、ラムダ計算を基本としながらも実際の使用に便利な機能を追加したプログラミング言語である。Haskell と（型なし）ラムダ計算との主な違いは

1. 整数などの定数を導入している。
2. 式は＿＿＿＿＿＿＿＿（空欄 3.0.2）されている。つまり、コンパイル時に（実行する前に）型エラーが検出される。
3. 代数的データ型というユーザー定義のデータ型を定義することができる。
4. 代数的データ型に対してパターンマッチングによって場合分けすることができる。
5. グラフ簡約という手法によって実行される。
6. 中置記法の演算子を使用することができる。

などの点である。また、他のプログラミング言語と比べた時の特徴は、以下のようになる。

1. ＿＿＿＿＿＿＿＿（空欄 3.0.3）を持ち、プログラマがメモリの管理に煩わされることがない。
2. 関数を他の関数の引数としたり、関数を生成して他の関数の戻り値としたり、関数を一般の値として使うことができる（＿＿＿＿＿＿＿＿（空欄 3.0.4））。
3. 多相型を許すことにより、汎用性のある関数を定義することができる。
4. 型推論により、（ほとんどの場合）プログラム中に型を書く必要はない。また、型クラスなど、型システムが発達している。
5. 式に＿＿＿＿＿＿＿＿（空欄 3.0.5）はない。入出力や参照の書換えなどの効果も値として表現される。このため、プログラムの同値性などの性質に関する考察が、より容易になる。
6. 遅延評価を採用し、グラフ簡約によって実行される。

一般に関数型言語は記号処理に適している。もちろん、純関数型言語を C や Java などと同じように実世界のプログラミングに利用することも可能である。しかし、ここでは、主に他のプログラミング言語を実装するための超言語として紹介することにする。

3.1 Haskell 処理系の入手とインストール

Haskell の環境としてここでは Haskell Platform を利用することにする。Haskell Platform は Haskell 処理系の GHC (Glasgow Haskell Compiler[1]) に標準的なツール・ライブラリ・ドキュメントを付け加えたものである。GHC は Haskell の処理系の事実上の標準（デファクトスタンダード）である。Haskell Platform は、`http://hackage.haskell.org/platform/` からダウンロードすることができる。Windows の場合、インストールは入手したファイルをダブルクリックするだけである。

また Linux 用のインストーラーなども上記のホームページから入手することができる。

3.2 GHCi のコマンド

GHC は、多くのプログラムの集合体であり、`gcc` や `javac` のように実行可能形式を出力するバッチコンパイラ（`ghc`）もその中に含まれている。ここでは、その中で対話型の処理系である GHCi (`ghci`) の使用法を説明する。

GHCi を起動すると、

```
Prelude>
```

のようなプロンプトが表示される。このプロンプトからコマンドを入力することによって、ファイルからプログラムをロードし、式を評価することができる。

GHCi のコマンドには次のようなものがある。

コマンド	省略形	意味
`:load` *file*	`:l`	*file* をロードする。
`:also` *file*	`:a`	*file* を追加ロードする。
`:reload`	`:r`	以前にロードしたファイルをリロードする。
`:type` *expr*	`:t`	*expr* の型を表示する。
`:cd` *directory*	`:c`	ディレクトリを変更する。
`:!` *command*		*command* をコマンドプロンプトに渡して実行する。
		例: `:!cd`, `:!dir`, `:!start .` など
`:?`		ヘルプを表示する。
`:main`	`:ma`	`main` 関数を実行する
`:quit`	`:q`	GHCi を終了する。

また、単に式を入力すると、その式を評価してその結果を出力する。

```
Prelude> 1+2
3
```

[1] Guarded Horn Clauses という論理プログラミング言語と同じ頭文字だが、何の関係もない。

（このテキスト中の実行例では、*3* のように斜体になっているところがシステムの出力で、それ以外がユーザの入力である。）

Haskell のプログラムソースファイルには通常 ____ （空欄 3.2.1）という拡張子をつける。

3.3 Haskell のプログラムの基本

Haskell の仕様書は `http://www.haskell.org/` から入手することができる。以下では、Haskell の仕様の基本的なところを紹介する。

変数の宣言 Haskell では他のプログラミング言語同様、変数を宣言して良く使う式に名前をつけることができる。ただし、C 言語のような命令型言語と異なり、変数は一度宣言するとその値を変えることはできない（代入はできない）。

変数の宣言は次の形で行なう。

変数名 = 式

複数の変数をまとめて宣言する時は次の形式になる。

```
{
  変数名1 = 式1;
  変数名2 = 式2;
  ...;
  変数名n = 式n
}
```

つまり、前後を “{” と “}” （ブレース）で囲み、“;” （セミコロン）で区切る。ただし、ブレースとセミコロンは多くの場合省略でき、以下のプログラム例でも原則として省略する。省略の詳細な条件は付録で説明する。

変数名には C 言語と同じようにアルファベット、数字、“_” （アンダースコア）が使えるほか、“’” （アポストロフィ）も使うことができる。アルファベットの大文字と小文字は区別する。ただし、通常の変数名は小文字（またはアンダースコア）から始まる必要がある。大文字から始まる名前は、後で紹介する構成子名に用いる。

プログラム Haskell のプログラムはモジュールの集合で、1 つのモジュールは基本的には複数の変数の宣言の前に

```
module モジュール名 where
```

というヘッダ部分をつけた形式である。つまり、以下のようなかたちである。

```
module モジュール名 where {
  変数名1 = 式1;
  変数名2 = 式2;
  ...;
  変数名n = 式n
}
```

ただし変数の宣言の他に、import 宣言や型の宣言、型クラス関係の宣言などを書くことができる。これらについては後述する。通常、1 つのファイルに 1 つのモジュールを記述する。

「module モジュール名 where」の部分を省略すると Main という名前のモジュールの定義と解釈される。ブレースやセミコロンも多くの場合省略されるので、もっとも簡単な場合、Haskell のプログラムは単に次のような形式をしていることになる。

変数名$_1$ = 式$_1$
変数名$_2$ = 式$_2$
...
変数名$_n$ = 式$_n$

関数定義　関数を定義するときは、仮引数を “=” の左辺に並べて、

```
id x    = x
twice x = 2*x
foo x y = 2*x+y
```

のように書くことができる。関数 id や関数 twice の場合は、x が、関数 foo の場合は x と y が仮引数である。

四則演算の演算子は、C や Java と共通のものが多いが、割り算関係などは異なる。異なる点は必要になった時点で説明する。

関数の適用は “twice 2” や “foo 3 4” のように関数と実引数を並べて書くだけである。（その値はそれぞれ、4 と 10 になる） 通常の数学の記法や C 言語などのように引数に括弧をつける必要はない。もちろん演算の順番を明示するために “foo (twice 2) (id 3)” のように括弧を使用することはできる。（この値は 11 になる。）

Q 3.3.1 次のような関数を定義せよ。

1. 3 倍して 1 を引く関数 bar
2. 3 乗する関数 cube

ラムダ式　Haskell では、無名関数を表すのにラムダ式というラムダ計算に由来する記法を用いる。

“\”（空欄 3.3.1）”（バックスラッシュ）（ただし、日本語環境ではしばしば円マーク “¥” になってしまう）のあとに仮引数を空白で区切って並べて書き、そのあとに “->”（空欄 3.3.2）”、つづけて関数本体の式を書く。例えば、“\ x y -> 2*x+y” は、2 つの引数 x, y を受け取り、x の 2 倍と y の和を返す関数である。ラムダ式は無名関数（あるいは匿名関数）とも言う。

（ラムダ計算とは、“λ” の代わりに “\” を、“.”（ピリオド） の代わりに “->” を用いる点が異なる。）

ラムダ計算	Haskell
$\lambda x.x$	_____
$\lambda xy.y$	_____

当然ながら、仮引数の名前は（他の変数と衝突しない限り）自由に選ぶことができる。例えば、“`\ x y -> 2*x+y`” と “`\ dog cat -> 2*dog+cat`” は同じ関数を表す。このような変数名の付け替えを α 変換と言う。

上記の関数定義は、次の定義とまったく等価である。

```
id    = \ x -> x
twice = \ x -> 2*x
foo   = \ x y -> 2*x+y
```

Q 3.3.2 Q.3.3.1 の関数 bar, cube を、“変数 = ラムダ式” の形式で定義せよ。

コメント Haskell のコメントには 2 つのかたちがある。

- _____（空欄 3.3.3）というかたち（一行コメント）
- _____（空欄 3.3.4）というかたち（複数行コメント）

3.4 組み込みのデータ型と演算子

Haskell では真偽値（`Bool`）、整数（`Int`, `Integer`—`Int` は固定（有限）精度の整数, `Integer` は無限精度の整数）、浮動小数点数（`Float`, `Double`）、文字（`Char`）などは組み込みのデータ型として用意されている。`Bool` 型のリテラルは `True` と `False` であり、`Integer`, `Float`, `Double`, `Char` 型などのリテラルの記法は C 言語とほぼ同様である。（`0.2` の `0` は省略できないなど，細かい相違はある。）

これらのデータ型に対しては組み込みの関数や演算子がいくつか用意されている。Lisp の場合と異なり、算術演算子の “+”, “-”, “*” などは、`1+2*3` のように通常の中置記法で用いる。

`if`〜`then`〜`else` **式**も組み込みで用意されている。次の形で用いる。

`if` 式$_1$ `then` 式$_2$ `else` 式$_3$

式$_1$ は `Bool` 型でなければならず、式$_2$ と式$_3$ は同じ型でなければならない。式$_1$ が `True` の時は式$_2$、`False` の時は式$_3$ として評価される。なお、後で詳しく説明するが、`if`〜`then`〜`else` 式は、Haskell では特殊な評価順を必要とする特殊形式ではない。同様の働きをする通常の関数を定義することも可能である。

次の例は階乗（factorial）の関数の Haskell での宣言である。

```
fact n = if n==0 then 1 else n*fact(n-1)
```

関数適用は他のどんな中置記法の演算子よりも＿＿＿＿＿＿＿＿＿＿（空欄 3.4.1）。そのため、fact(n-1) は fact n-1 と書くことはできない。後者は＿＿＿＿＿＿＿＿（空欄 3.4.2）の意味になってしまう。逆に n*(fact(n-1)) の外側の括弧は必要ない。

なお、この例のように変数は＿＿＿＿定義することが可能である。つまり、定義の右辺に自分自身を使用することができる。再帰的定義に特別な文法を使用する必要はない。

問 3.4.1 fact 100 を計算してみよ。

Q 3.4.2 フィボナッチ数列を計算する関数 fib を（効率を気にせず単純に）定義せよ。

（発展）ガード　ガードといって、仮引数の並びの後に “|” を書き、そのあとに条件式を書くことが出来る。ガード節（「| 条件式 = 右辺」のかたち）は複数個並べることが出来る。その場合、上（左）のガードから条件式を評価し、真になった箇所の “=” の右辺を評価する。ガードを使うと、上記の fact の定義は次のように書くことが出来る。

```
1  fact n  | n==0       = 1
2          | otherwise  = n*fact(n-1)
```

リスト　リスト型も組み込みのデータ型として用意されている。リストとは簡単に言えばデータの並びである。リストは伝統的に Scheme などの Lisp 系の言語が得意とするデータ型であり、Haskell でも豊富なライブラリ関数が用意されている。

リストは、空リスト＿＿（空欄 3.4.3）とコンス (cons) と呼ばれる演算子 “＿（空欄 3.4.4）” から構成される。この 2 つをリストの構成子 (constructor) と呼ぶ。

- 空リスト（[]）は文字通り空のリストである。
- コンス（:）は右オペランドとして渡されるリストの先頭に左オペランドとして渡される要素を付け加えたリストを返す演算子である。

また、“:” は＿＿＿＿（空欄 3.4.5）である。つまり、1:2:[] は 1:(2:[]) のことを表す。

リストのリテラルの記法として、要素を “,”（コンマ）で区切って並べ、“[” と “]” で囲む記法も用意されている。例えば [1,2,3,4] は、1:2:3:4:[] のことである。

Q 3.4.3 [1,2,3] と [[1,2],[3,4]] を : と []（と括弧と整数リテラル）だけで定義せよ。

リスト型はパラメーターを持つ型である。つまり、リストの要素の型をパラメーターとする。要素の型が `Integer` 型の場合、そのリストの型は＿＿＿＿＿＿（空欄 3.4.6）型、要素の型が `Double` 型の場合リストの型は＿＿＿＿＿＿（空欄 3.4.7）型と書き表される。型の異なる要素が混在するリストは作成できない。つまり、`[2, 'a', [2]]` のような式は型エラーとなる。

Q 3.4.4 `[False, True]` の型と `["Kagawa", "University"]` の型を書け。

String 型と type 宣言　Haskell の文字列（`String`）型は実は文字のリスト型として表されている。つまり、

```
type String = [Char]
```

のように定義されている。ここで、

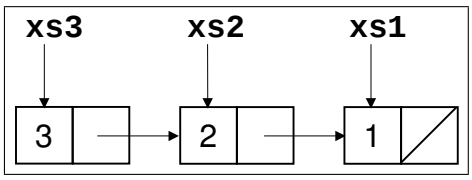
```
type 型名 = 型
```

は型の別名（type alias）を宣言する形式である。上の例の場合 `String` という型名が、`[Char]` という型の別名となる。型名は大文字から始まる識別子でなければならない。

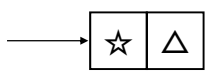
箱ポインター記法　リストを、箱と矢印を用いた図（箱ポインター記法、box-pointer 記法）で表すことがある。例えば、次のように構成されたリストの場合、

```
xs0 = []
xs1 = 1:xs0
xs2 = 2:xs1
xs3 = 3:xs2
```

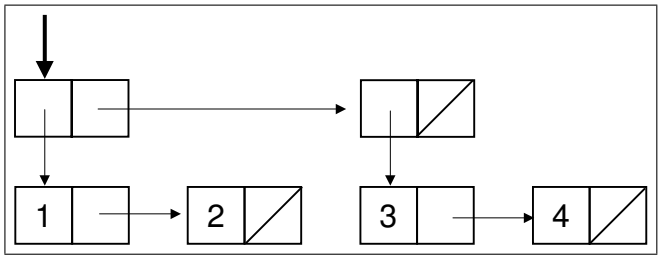
次の図のようになる。



1 つの “`:`” を 2 個の正方形の箱がつながった箱（コンスセル）**への矢印**で表す。箱の中身は、`1`, `2`, `3` などの数、他の箱への矢印などである。（箱の中に他の箱がまるごと入ることはない。）空リストは斜線を引いて表す。

式	箱ポインター記法
☆`:`△	
`[]`	

また、`[[1,2],[3,4]]` というリストのリストは



という箱ポインター記法（の左上の太い矢印）で表される。

Q 3.4.5 次のリストを箱ポインター記法で書け。

1. `[2,3,5,7,11]`
2. `[0]`
3. `[[1],[2,3,4],[]]`

（参考） リストを C 言語の構造体として定義すると次のようになる。

```
1  struct _list {
2    int car;
3    struct _list* cdr;
4  }
5
6  typedef struct _list* list;
```

`_list` という構造体は、`car` と `cdr` という 2 つのメンバを持つ。このうち `cdr` は現在定義されている型へのポインタ型である。`list` という型は、**`typedef`** 宣言によって、**`struct`** `_list*`という型の別名として定義される。（箱ポインター記法で矢印になっているところが、C 言語ではポインターとして表される。）

また、空リストは C 言語では通常、定数 `NULL` で表される。

`malloc` 関数が使われていることからわかるように、C 言語でリストを扱う場合は、いつ `free` をするかを気を付けなければいけない。Haskell や他の関数型言語ではゴミ集めという仕組みのおかげで明示的に `free` しなくても、いらなくなったメモリ領域は自動的に回収されることになっている。

組 組 (tuple) も組み込みのデータ型として用意されている。組は要素を “,” （コンマ）で区切って並べ、 “(” と “)” で囲んで表す。組はリストの場合と異なり、要素の型が同一である必要はない。`(1, 'a')` という式の型は _____ と表記される。また、`(2, 'b', [3])` という式の型は _____ と表記される[2]。

Q 3.4.6 次の式の型は何か？

1. `(False, "Hello")`　　2. `('X', ("Aloha", False))`

[2]実際の Haskell では、型クラス（type class）というものが関係するため、これらの式はもう少し複雑な型を持つ。ここでは型クラスの説明は避けて、実際よりも単純化した型（整数リテラルは `Integer`, 浮動小数点数リテラルは `Double`）を紹介している。

() という要素がゼロ個の組もある。ユニット (unit) と呼ばれる。() の型も () と表記し、ユニット型と呼ばれる。C 言語の void 型のような使い方をする。

関数型　関数の型は “->” という記号を使って、Integer -> Char のように表記される。これは、引数の型が Integer で戻り値の型が Char の関数の型である。“->” は＿＿＿＿＿である。つまり、Bool -> Bool -> Bool という型は＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿と解釈される。Haskell では多引数関数を “関数を返す関数” として表現する（このことを “カリー化” という）ので、このように約束しておくほうが便利である。

多相型　Integer や Char のように型定数の名前は必ず大文字から始まる。一方、a や b のように小文字で始まる識別子が型の中に現れる場合、これらは＿＿＿＿＿である。これらの型変数は使用する時に、より具体的な型に置き換えることができる。例えば、[a] -> [b] -> [(a, b)] という型を持つ関数は、[Char] -> [Integer] -> [(Char, Integer)] という型を持つ関数として使用しても良いし、[String] -> [Integer -> Integer] -> [(String, Integer -> Integer)] として使用しても良い。

このように型変数を含む型を＿＿＿＿＿という。Haskell は＿＿＿とともに多相型を許す代表的なプログラミング言語である。

なお、変数の型をプログラム中に明示したい場合 “::” という記号を使って、

変数名 :: 型

と書く。例えば id や fact の型を明示しておきたい場合は、

```
1  id :: a -> a
2  id x = x
3
4  fact :: Integer -> Integer
5  fact n = if n==0 then 1 else n*fact(n-1)
```

のように書く。ただし通常は、変数の型はプログラマが明示しなくても、Haskell 処理系が推論してくれる。この仕組みを＿＿＿＿＿という。

3.5　パターンマッチング

Haskell では、関数の仮引数の部分に＿＿＿＿＿＿というものを書いて、引数の形に応じて場合分けを行なう事が可能である[3]。パターンとは、大雑把に言って変数と定数、構成子（:や [] など）からのみ生成される式である。

```
1  fact :: Integer -> Integer
2  fact 0 = 1
3  fact n = n * fact (n-1)
```

[3]一般に関数型言語はデータの種類は増えずに処理（関数）が増えるような場合が得意、オブジェクト指向言語は、データ（クラス）の種類が増えて、処理（メソッド）が増えないような場合が得意である。

```
4
5   -- Prelude の lengthとほぼ同じ
6   myLength :: [a] -> Integer
7   myLength []     = 0
8   myLength (x:xs) = 1 + myLength xs
```

この例の場合、myLengthの引数が空リスト（[]）ならば1行目が選択される。一方、1つ以上の要素を持つリストならば2行目が選択され、リストの先頭要素が変数xに束縛され、残りのリストが変数xsに束縛される。（なお、myLengthとほとんど同じ関数length :: [a] -> Intが標準ライブラリに用意されている。）

パターンマッチングで、右辺で使わない部分には“_”（アンダースコア）を使って変数に束縛せず、無視することができる。例えばmyLengthの場合、xは右辺で使用していないので、

```
myLength (_:xs) = 1 + myLength xs
```

と書くことができる。

case〜of式もパターンマッチングを行なう。case〜of式は次の形で用いる。（やはり、ブレースとセミコロンは通常省略する。）

```
case 式0 of {
  パターン1 -> 式1;
  パターン2 -> 式2;
  ...
  パターンn -> 式n
}
```

式$_0$を評価して、上から順にパターンと照合し、パターン$_1$にマッチするならば式$_1$が、パターン$_m$にマッチするならば式$_m$がそれぞれ評価される。

if 式$_1$ then 式$_2$ else 式$_3$という式も次のcase〜of式の略記法と解釈することが可能である。

```
case 式1 of { True -> 式2; False -> 式3 }
```

この他に、let式（後述）やλ式など変数を束縛するところでもパターンを書くこともできる。例えば、

```
\ (x,y) -> x
let (xs,ys) = unzip zs in xs++ys
```

などである。この場合は、パターンは一種類のみで場合分けはできず、パターンにマッチしない引数が与えられればエラーとなる。

問 3.5.1 リスト中の数の和を求める関数mySumを定義せよ。（同等の関数sumは標準ライブラリに用意されている。）

問 3.5.2 リスト中の数の積を求める関数myProdを定義せよ。（同等の関数productは標準ライブラリに用意されている。）

問 3.5.3 真偽値のリスト [Bool] を 2 進数と見なして、対応する整数を計算する関数 fromBin :: [Bool] -> Integer を定義せよ。例えば、fromBin [True, True] は 3, fromBin [True, False, True, False] は 10 になる。
ヒント : 引数の数を一つ増やした補助関数が必要になる。
ヒント : ホーナーの方法を使う。例えば fromBin [True, False, True, False] は $((1\times2+0)\times2+1)\times2+0$ fromBin [True, True, False, True] は $((1\times2+1)\times2+0)\times2+1$ となるように計算する。

問 3.5.4 真偽値のリスト [Bool] を 2 進数と見なして、対応する整数を計算する関数 fromBinRev :: [Bool] -> Integer を定義せよ。ただし、先の問とは逆順に真偽値がならんでいると仮定せよ。例えば、fronBinRev [True, True, False, True] は $1+2\times(1+2\times(0+2\times1))=11_{(10)}$ となる。

問 3.5.5 実数のリスト xs と実数から実数への関数 f を受け取り、リストの各要素に f を適用した結果の和を計算する関数 sumf :: [Double] -> (Double -> Double) -> Double を定義せよ。

問 3.5.6 2 つの引数 x, xs を受け取り、リスト xs から x と等しい要素を

1. もっとも先頭に現れる一つの要素だけ取り除いたリストを返す関数 deleteOne
2. すべて取り除いたリストを返す関数 deleteAll

をそれぞれ定義せよ。

問 3.5.7 リストを昇べキの順に表された多項式と見なし、多項式の値を計算する関数
evalPoly :: [Double] -> Double -> Double を定義せよ。例えば、[1, 2, 3, 4] というリストは $1 + 2x + 3x^2 + 4x^3$ という多項式と見なし、evalPoly [1, 2, 3, 4] 10 の値は 4321 になる。

3.6 帰納法による証明

プログラミング言語の意味をラムダ計算や関数型言語で表現することの利点は、プログラムの同値性（等価性）などの議論が容易になるというところにある。（これに対して例えば C 言語では、c = getchar(); という代入文があったとしても、c の出現を getchar() で置き換えることはできない。）ここでは、そのような議論の例として、2 つのリストに関する関数が等価であることの証明を取り上げる。このような証明は帰納法と併用することが多い。以下の例も帰納法を使用している。

リストを反転する関数 reverse:

```
1  -- append は Prelude の (++) 演算子と同じ
2  append           :: [a] -> [a] -> [a]
3  append [] ys      = ys
4  append (x:xs) ys  = x : (append xs ys)
5
6  -- reverse は Prelude に定義済み
7  reverse       :: [a] -> [a]
8  reverse []     = []
9  reverse (x:xs) = append (reverse xs) [x]
```

は上の定義では、引数の＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿に比例する時間がかかるために効率が悪い。

そこで次のような定義を考える。

```
1  -- shunt は rev の補助関数
2  shunt          :: [a] -> [a] -> [a]
3  shunt ys []     = ys
4  shunt ys (x:xs) = shunt (x:ys) xs
5
6  rev   :: [a] -> [a]
7  rev xs = shunt [] xs
```

この rev という関数は、引数の＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿に比例する時間でリストを反転できるので効率が良い。

rev と reverse が等価であること – 正確に言うと、すべての有限リスト *xs* に対して

rev xs = reverse xs

が成り立つことを証明できる。

そのためには、次のような補助定理を証明すれば良い。

shunt ys xs = append (reverse xs) ys

これは、＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿で証明することができる。

証明:

xs = [] のとき:

xs = z:zs のとき:

問 3.6.1 すべての有限リスト `xs` について、

1. `append xs [] = xs`
2. `append xs (append ys zs) = append (append xs ys) zs`

が成り立つことを、`xs` に関する帰納法で証明せよ。

3.7 有用なリスト処理関数

次のようなリストに対する関数がよく利用される。これらは Prelude（標準ライブラリ）に定義済みである。

これらのリスト処理関数の多くは高階関数である。

高階関数とは、関数を引数としたり、関数を生成して戻り値とするような関数のことである。

```
1   map :: (a -> b) -> [a] -> [b]
2   map f []     = []
3   map f (x:xs) = f x : map f xs
4
5   zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
6   zipWith f (x:xs) (y:ys) = f x y : zipWith xs ys
7   zipWith f _       _     = []
8
9   filter :: (a -> Bool) -> [a] -> [a]
10  filter p []     = []
11  filter p (x:xs) = if p x then x : filter p xs else filter p xs
12
13  iterate :: (a -> a) -> a -> [a]
14  iterate f x = x : iterate f (f x)
```

```
15
16  foldr :: (a -> b -> b) -> b -> [a] -> b
17  foldr f x []     = x
18  foldr f x (y:ys) = f y (foldr f x ys)
19
20  foldl :: (a -> b -> a) -> a -> [b] -> a
21  foldl f x []     = x
22  foldl f x (y:ys) = foldl (f x y) ys
23
24  concat :: [[a]] -> [a]
25  concat []       = []
26  concat (xs:xss) = xs ++ concat xss
```

問 3.7.1 これらの関数（や以前に紹介した関数）を使って、次のような関数を定義せよ。

1. 2 つのリストの 0 番目の要素同士、1 番目の要素同士、... を比較し、等しい要素の個数を返す関数 `countEq`
   例えば `countEq [1,2,3,5] [2,2,6,5,3]` は 2 になる。

2. 文字列のリストを受け取り、各文字列の最後に “;” をつけて連接した文字列を返す `addSemicolon`
   例えば `addSemicolon ["abc", "xyz", "123"]` は `"abc;xyz;123;"`になる。

3.8 関数の中置記法化

Haskell の関数は通常は前置記法で用いるが、識別子を “`` ` ``”（バッククォート、クォート（“’”）ではないことに注意）で囲むことによって、中置記法で書くことができる。これは小さなことに見えるが実は意外に便利である。例えば、次のように Prelude で定義された `zip` の場合、

```
1  zip :: [a] -> [b] -> [(a,b)]
2  zip (a:as) (b:bs) = (a,b) : zip as bs
3  zip _      _      = []
```

通常は zip [1,2] [3,4] のように書くが、これを [1,2] `zip` [3,4] と書いても良い。

Q 3.8.1 append [1,2] [3,4,5] を `〜` を使って中置記法で書け。

中置記法で用いる演算子に対して、infixl, infixr, infix などのキーワードを使って、優先順位と結合性を定めることができる。たとえば、Prelude（Haskell にはじめから読み込まれる標準ライブラリ）では次のように宣言されている。

```
1   infixr 9  .
2   infixl 9  !!
3   infixr 8  ^, ^^, **
4   infixl 7  *, /, `quot`, `rem`, `div`, `mod`, :%, %
5   infixl 6  +, -
6   infixr 5  :
7   infixr 5  ++
8   infix  4  ==, /=, <, <=, >=, >, `elem`, `notElem`
9   infixr 3  &&
10  infixr 2  ||
11  infixl 1  >>, >>=
12  infixr 1  =<<
13  infixr 0  $, $!, `seq`
```

infixl は＿＿＿＿＿、infixr は＿＿＿＿＿を表す。ただの infix はどちらでもないこと（非結合—例えば 1 < x < 2 は構文エラー！）を表す。また、2 列目の数字が大きいほど、優先順位が高い。例えば*は +よりも結合力が強い。

Q 3.8.2 次の式の解釈はどうなるか？括弧を挿入して演算順を明示的にせよ。

1. x `rem` 3 /= 1　　2. xs !! 9 : ys ++ zs

バッククォートと逆に本来中置記法で使用される演算子を “(” と “)” でくくって、ふつうの前置記法で用いることができる。例えば1+2 を(+) 1 2 と書くことができる。あるいは関数の引数として渡すこともできる。

Q 3.8.3 次の式を通常の中置記法で書け。

1. (+) ((*) 2 x) 1　　2. (++) xs ((++) ys zs)

3.9 部分適用とセクション

Haskell の関数はカリー化されている。すなわち、多引数の関数は「関数を返す関数」として表現されている。引数の一部だけを適用して、結果の関数を map のような関数の引数に使うことは良くある。

```
Prelude> map (take 2) [[1,2,3],[4,5,6,7],[8,9,10]]
[[1,2],[4,5],[8,9]]
Prelude> map (zip [8,7,6]) [[1,2,3],[4,5,6,7],[8,9]]
[[(8,1),(7,2),(6,3)],[(8,4),(7,5),(6,6)],[(8,8),(7,9)]]}
```

Haskell では演算子にも部分適用ができるようになっている。例えば (2*) は 2 倍する関数。(/2) は 2 で割る関数である。このような二項演算子の部分適用をセクションという。

```
Prelude> map (2*) [1,2,3]
[2,4,6]
Prelude> map (/2) [1,2,3]
[0.5,1.0,1.5]
Prelude> map (1/) [1,2,3]
[1.0,0.5,0.3333333333333333]}
```

Q 3.9.1 次のセクションをラムダ式で書け。

1. (xs++)　　2. (!!9)

ただし "-" 演算子は、単項演算子の "-" も存在するので、(-2) はセクションにはならない。その代わり subtract という関数（subtract x y = y - x）が存在するので、subtract 2 と書くことができる。

3.10 局所的定義

let というキーワードを用いて、局所的な変数を定義することができる。

let （複数の）変数の定義 **in** 式

という形で用いる。

```
1  pow4 x  = let y = x*x in y*y
2  -- head は Prelude に定義済み
3  head ys = let (x:xs) = ys in x
4  -- head (x:xs) = x と定義することもできる
```

などである。

Q 3.10.1 次のような関数を let を用いて定義せよ。

1. 27 乗する関数 pow27
2. リストの尾部（頭部を除いた残り）を求める関数 tail

この例では 4 乗する関数（pow4）を定義しているが、y*y の部分が変数 y の有効範囲（＿＿＿＿＿＿）に属している。実は、さらに "変数の定義" の右辺の部分（この例では、＿＿＿の部分）もスコープに属している。これは次の例でわかる。

```
1  -- repeat は Prelude に定義済み
2  repeat  :: a -> [a]
3  repeat x = let xs = x:xs in xs
```

この関数は要素 x の無限リストを生成する。

```
Prelude> repeat 1
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, ... [4]
```

（このような定義は、Haskell では “止まらない・役に立たない式” ではなく、意味のある式となる。このことは、あとで Haskell の評価戦略を紹介する時に説明する。）

Q 3.10.2 `repeatList [1,2,3]` が `[1,2,3,1,2,3,1,2,3, ... ]` という無限リストになるような関数 `repeatList :: [a] -> [a]` を定義せよ。

問 3.10.3 リストを集合だと見なして、そのベキ集合（部分集合の集合）を返す関数 `powerset` を定義せよ。

例えば `powerset [1, 2, 3]` は `[[], [1], [2], [3], [1, 2], [2, 3], [1, 3], [1, 2, 3]]` になる。（ただし、順番はこの通りでなくても良い。）

ヒント： セクションを使うと簡潔に定義できる。

（発展）where 節 **where** というキーワードを使うと、関数定義の右辺で使用される変数・関数を後ろに局所的に定義することが出来る。

関数名 *パターンの並び* = 式
  where （複数の）変数の定義

という形で用いる。（この形式では示されていないが、局所的定義された変数が複数のガード節にまたがって使用されていてもよい）上記の pow4, head は次のように定義することも出来る。

```
1  pow4 x  = y*y
2    where y = x*x
3  -- head は Prelude に定義済み
4  head ys = x
5    where (x:xs) = ys
```

3.11 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＿＿＿＿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$_{1,n_1}$
  | 構成子名$_2$ 型$_{2,1}$ ... 型$_{2,n_2}$
  | ...
  | 構成子名$_m$ 型$_{m,1}$ ... 型$_{m,n_m}$

[4] Ctrl-c で中断する。

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。

代数的データ型は構成子にパラメータがない場合は、C や Java の列挙（enum）型と同じようなものである。次の例では、

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

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

Q 3.11.1 じゃんけんの 3 つの手（グー・チョキ・パー）を表すデータ型 RPS を定義せよ。

一般的には代数的データ型の構成子はパラメータを持つ。次の例は二分木を表すデータ型を定義している。

```
1  data Tree a = Empty | Branch (Tree a) a (Tree a)
2                               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 は引数を取らず、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 番目は自分自身と同じ型の二分木であり、2 番目は要素である。つまり、Branch は次のような型を持っている。

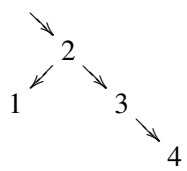
```
Branch :: _____________________________________________
```

ここで、a は_____であり、ここには Integer や String などの具体化な型がはいることができる。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、型パラメータを伴ってはじめて型になる。

具体的には次のような式がこの Tree 型を持つ。

```
1  tree1 :: Tree a
2  tree1 = Empty
3  tree2 :: Tree Integer
4  tree2 = Branch Empty 1 Empty
5  tree3 :: Tree String
6  tree3 = Branch (Branch Empty "a" Empty)
7             "b" (Branch Empty "c" Empty)
8  tree4 :: Tree Integer
9  tree4 = Branch (Branch Empty 1 Empty)
10               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 型に対する関数は、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。

```
1  top :: Tree a -> a
2  top (Branch _ a _) = a
3
4  isEmpty :: Tree a -> Bool
5  isEmpty Empty = True
6  isEmpty _     = False
```

問 3.11.2 Tree 型に対して、次のような関数を定義せよ。

```
size      :: Tree a -> Integer -- 要素数
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 size tree4, ② depth tree4, ③ preorder tree4, ④ inorder tree4, ⑤ postorder tree4, ⑥ reflect tree4 の結果はそれぞれ、① 4, ② 3, ③ [2,1,3,4], ④ [1,2,3,4], ⑤ [1,4,3,2], ⑥ Branch (Branch (Branch Empty 4 Empty) 3 Empty) 2 (Branch Empty 1 Empty) となる。

問 3.11.3 一般の n 分木（任意の個数の子を持つことができる木）を表すデータ型を定義せよ。

問 3.11.4 PL/0 (http://www.inf.ethz.ch/personal/wirth/books/AlgorighmE0) の構文木を表すデータ型を定義せよ。

（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。

```
data C = F { f1, f2 :: Int, f3 :: 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。

```
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。

```
1  v1 = F { f1 = 5, f2 = 12, f3 = False }
2
3  foo :: C -> Int
4  foo (F { f1 = x, f2 = y }) = x*x + y*y
```

このとき `foo v1` の値は 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、`f1 v1` の値は `5` になる。

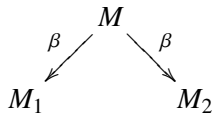
式の後に、“フィールドラベル = 式” のコンマ区切りの並びをブレース内に書いて、指定したフィールドの値のみを変更した新しいデータを構成するときにも使用できる。`v1 { f2 = 3 }` の値は `F { f1 = 5, f2 = 3, f3 = False }`になる。

3.12 Haskellの評価戦略

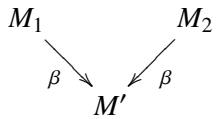
Haskellの式の評価は、ラムダ計算と同じく β 変換に基づいている。つまり、`(\ x -> M) N` という部分式を、関数の戻り値 `M` のなかの 仮引数 `x` の自由な出現を実引数 `N` に置き換えた式に書き換える。例えば、“`(\ x -> x+2) 3`” を `3+2` に置き換える。

これ以上 β 変換ができない式を正規形という。

一つの式に幾通りもの β 変換が可能なことがある。このとき、異なる β 変換を選ぶと、評価の結果が別の形に枝分かれしてしまう。



しかし、うまく何回か β 変換を行なうと、この枝分かれしたものを再び合流させられることが知られている。（チャーチ・ロッサーの定理）



これは同時に、ある式に正規形が存在するならば、それは一つしかない（α 変換による違いを除く）ということを保証している。

ラムダ計算のところでも紹介したが、最も左からはじまる β 基を選んでいけば、正規形の存在する式ならば、必ず正規形に到達することが可能であるということが知られている。この評価戦略を＿＿＿＿＿＿という

Haskellの評価戦略は、基本的にこの最左戦略による評価方法である。つまり正規形を持つ式の評価は必ず止まる。ただし、内部的には＿＿＿＿＿＿＿を用いる点がラムダ計算の場合と異なる。

例えば、次のように定義された `twice` という関数を考える。

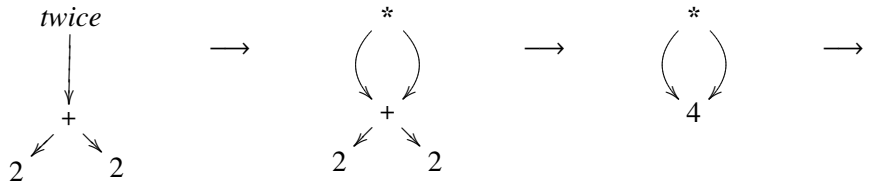
```
twice x = x*x
```

最左戦略では `twice (2+2)` という式は次のように計算することになる。

$$\begin{aligned} twice\ (2+2) &= (2+2)*(2+2) \\ &= 4*4 \\ &= 16 \end{aligned}$$

つまり、twice の引数である (2+2) は計算されないまま、まず twice の定義にしたがって式が展開される。そして、本当に必要になって (この場合は*の引数だから必要) はじめて 2+2 が計算される。この方式をナイーブに実行すると、2+2 が二度計算されてしまう。

グラフ簡約では、このような計算をグラフの形で表して、2+2 を一度しか計算しないようにしている。



16

最左戦略は必要になるまで評価を遅らせるので＿＿＿＿＿＿（lazy evaluation）とも言われる。ただし、プログラミング言語で遅延評価を用いる場合は、このようにグラフ簡約と組み合わせて、同じ計算の繰り返しを避けるようにするのが一般的である。

遅延評価の良いところは概念的に無限の大きさのデータ構造を扱えることである。例えば次のような関数を考える。

```
1  from :: Integer -> [Integer]
2  from n = n : from (n+1)
3
4  -- take は Prelude に定義済み
5  take :: Integer -> [a] -> [a]
6  take 0 _      = []
7  take _ []     = []
8  take n (x:xs) = x : take (n-1) xs
```

すると from 1 は [1, 2, 3, ...] という無限リストだが、この無限リストを途中で用いている take 3 (from 1)という式は

```
take 3 (from 1) → take 3 (1:from (1+1))
→ 1:(take 2 (from (1+1))) → 1:(take 2 ((1+1):from (1+1+1)))
→ 1:(1+1):(take 1 (from (1+1+1)))
→ ··· → 1:(1+1):(1+1+1):(take 0 (from (1+1+1+1)))
→ 1:(1+1):(1+1+1):[] (= [1,2,3])
```

のように有限時間で計算できる。

なお、from で生成されるような等差数列については “..” を使った略記法（糖衣構文）がいくつか用意されている。

```
Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,...
Prelude> [2,4..]
[2,4,6,8,10,12,14,16,18,20,22,...
Prelude> [1..10]
[1,2,3,4,5,6,7,8,9,10]
Prelude> [1,4..20]
```

```
[1,4,7,10,13,16,19]
```

（発展）“..” の翻訳

これらは単に Prelude で定義された関数の呼出しに翻訳される。

```
[ e1 .. ]          = enumFrom e1
[ e1, e2.. ]       = enumFromThen e1 e2
[ e1 .. e2 ]       = enumFromTo e1 e2
[ e1, e2 .. e3 ] = enumFromThenTo e1 e2 e3
```

遅延評価を用いるといろいろと興味深いプログラミングが可能になる。参考文献 [6] は、遅延評価を利用したプログラムの部品化の手法を詳しく説明している。

一方、遅延評価を用いるとプログラムの各部分がどのような順序で実行されるのか、事前に推測することが難しい。遅延評価を用いるためには、副作用が存在しないことが大前提である。

問 3.12.1 take の反対に、リストの最初の n 個の要素を取り除く関数 `myDrop :: Integer -> [a] -> [a]` を定義せよ。

問 3.12.2 fib をフィボナッチ数列の無限リストとして定義せよ。
ヒント: フィボナッチ数列同士をずらして足し算すると ...

```
      1  1  2  3  5  8 ...
         1  1  2  3  5 ...
+)    1  2  3  5  8  13 ...
```

ヒント: zipWith を使う。

問 3.12.3 要素に重複のない昇順に並んだ 2 つのリストをマージして、やはり重複なしの昇順のリストを生成する関数 merge を定義せよ。

問 3.12.4 $2^i \cdot 3^j \cdot 5^k$（i, j, k は 0 以上の整数）の形で表せる整数のみを重複なしに昇順に並べたリスト hamming を定義せよ。このリストの 699 番目の要素（ただし最初を 0 番目と数えた場合）が 5898240 であることを確認せよ。
ヒント: map と 問 3.12.3 の merge を使う。

問 3.12.5 Haskell 以外の言語で、無限リストをシミュレートする方法を考察せよ。またそれを用いて、素数列を生成せよ。

3.13 リストの内包表記（List Comprehension）

Haskell は、リストの＿＿＿＿＿＿（list comprehension）という糖衣構文（syntax sugar）を持つ。これは数学で使われる集合の内包表記に似た記法である。

例

```
Prelude> [(x, y) | x <- [1, 2, 3, 4], y <- [5, 6, 7]]
[(1,5),(1,6),(1,7),(2,5),(2,6),(2,7),(3,5),(3,6),(3,7),↙
(4,5),(4,6),(4,7)]
Prelude> [x*x | x <- [1..10], odd x]
[1,9,25,49,81]
```

ただし `[1..10]` は `[1,2,3,4,5,6,7,8,9,10]` の略記法である。また、↙は、本当は改行しない（プリントの幅の都合で改行している）ことを示している。

リストの内包記法は次のような形のものである。

`[` 式 `|` 限定式, ... , 限定式 `]`

ここで限定式は、`Bool` 型の式（ガード）か、次の形（生成式）:

変数（一般にはパターン） `<-` 式

のいずれかである。生成式の左辺の変数のスコープは、それより後の限定式である。生成式で変数に右辺の式を評価して得られるリストの要素を順に代入し、ガードで真となるもののみ抽出して、すべての組み合わせを列挙する。

Q 3.13.1 次の内容表記の値は何か?

1. `[ x*y | x <- [1, 2], y <- [3, 5, 7] ]`
2. `[ (x, y) | x <- [1, 4, 7], y <- [2, 5, 8] , x < y ]`

問 3.13.2 非負の整数 n を受け取り、$0 \le x \le y \le n$ となるすべての x, y の組を生成する関数 `foo :: Integer -> [(Integer, Integer)]` を内包表記を用いて定義せよ。
ヒント: `..` を使う。

問 3.13.3 非負の整数 n を受け取り、$1 \le x < y < z \le n$ の範囲で $x^2 + y^2 = z^2$ となるすべての x, y, z の組を生成する関数 `chokkaku :: Integer -> [(Integer, Integer, Integer)]` を内包表記を用いて定義せよ。例えば、`chokkaku 20` の値は、`[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17),(12,16,20)]` となる。（順番はこの例と異なっていても良い。）

内包表記の翻訳 リストの内包表記は次のような高階関数を用いて翻訳することができる。

```
1  unit :: a -> [a]   -- 要素数 1 のリストを返す
2  unit a = a : []
3
4  bind :: [a] -> (a -> [b]) -> [b]
5  bind []     _ = []
6  bind (x:xs) f = append (f x) (bind xs f)
```

翻訳規則は次のとおりである。(下線部に翻訳規則を再帰的に適用していく。)

$$
\begin{aligned}
\texttt{[t |]} &\Rightarrow \texttt{unit t} \\
\texttt{[t | x <- u, P]} &\Rightarrow \texttt{bind u (\ x -> } \underline{\texttt{[t | P]}}\texttt{)} \\
\texttt{[t | b, P]} &\Rightarrow \texttt{if b then } \underline{\texttt{[t | P]}}\texttt{ else []} \\
\texttt{[t | let decls, Q]} &\Rightarrow \texttt{let decls in } \underline{\texttt{[t | Q]}}
\end{aligned}
$$

ただし、*b* は Bool 値の式、*P* は限定式の並びである。例えば、

```
[(x,y) | x <- [1..3], y <- [2..4], odd (x+y) ]
```

は次のように翻訳される。

```
⇒ bind [1..3] (\ x -> [(x,y) | y <- [2..4], odd (x+y) ])
⇒ bind [1..3] (\ x ->
     bind [2..4] (\ y -> [(x,y) | odd (x+y) ]))
⇒ bind [1..3] (\ x ->
     bind [2..4] (\ y -> if odd (x+y) then [(x,y) |] else []))
⇒ bind [1..3] (\ x ->
    bind [2..4] (\ y -> if odd (x+y) then unit (x,y) else []))
```

内包表記を用いると、クイックソートは次のように簡潔に表される。

```
1  qsort []     = []
2  qsort (x:xs) =
3        qsort [ y | y <- xs, y < x]
4     ++ x : qsort [ y | y <- xs, y >= x]
```

問 3.13.4 次の内包記法を上の翻訳規則を用いて、unit, bind を用いた形にせよ。

1. `[ (x, y) | x <- [1, 2, 3, 4], y <- [5, 6, 7] ]`
2. `[ x*x | x <- [1..10], odd x ]`

問 3.13.5 primes を素数列 `[2, 3, 5, 7, 11, ... ]` の無限リストとして定義せよ。(内包表記を用いても、用いなくても良い。)

考え方: “エラトステネス (Eratosthenes) のふるい” というアルゴリズムを実装する。このおなじみのアルゴリズムを言葉で表現すると次のようになる。

1. 2 以上の自然数を並べる。
2. 先頭の数を取り除き、その倍数を同時にとり除く。(この処理を “ふるい” (sieve) と呼んでいる。)
3. 2 を繰り返す。

この時に先頭に現れた数を順番に並べたものが素数の列である。途中で無限リストの無限リストが現われるが、問題ない。

このようにして、素数を無限リストとして表現することで、さまざまな “境界条件” に対応することができる。C などで実装しようとすると、1000 までの素数というのは配列を用いて簡単に求めることができるが、最初の 100 個の素数を求めるのは急に難しくなる。

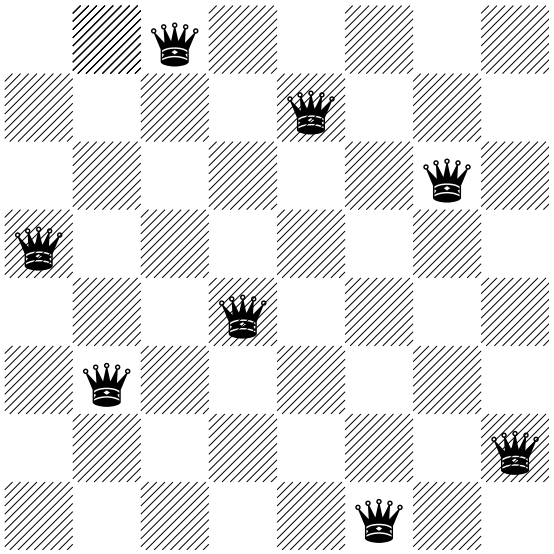
問 3.13.6 他の言語で “エラトステネスのふるい” を実装してみよ。

3.14 8 クイーンの問題

リストの内包表記を用いて、有名なパズルを解いてみることにする。

8 クイーンの問題は 8 個のクイーンを、お互いにとることができないように、チェス盤の上に置くという問題である。クイーンは縦・横・斜めのすべての方向に、何マスでも移動できる。この問題の可能な解はいくつか存在する。この可能な解の集まりをリストとして表現する。ここでは無限リストは本質的には使用しないが、遅延評価は効率の点で決定的な役割を果たす。

クイーンの配置は、ここでは数のリストで表す。`[4, 6, 1, 5, 2, 8, 3, 7]` は次のような配置を表す。



safe p n は、length p 列までのクイーンの配置が p というリストで与えられた時、第 length p + 1 列の第 n 行にクイーンを置くことができるかどうかを示す関数である。

```
1  safe p n  =
2    all not [ check (i, j) (1+length p, n)
3            | (i, j) <- zip [1..] p ]
4
5  check (i,j) (m,n) = j==n || (i+j==m+n) || (i-j==m-n)
```

ここで、all は

```
1  all               :: (a -> Bool) -> [a] -> Bool
2  all p []           = True
3  all p (x:xs)       = if p x then all p xs else False
```

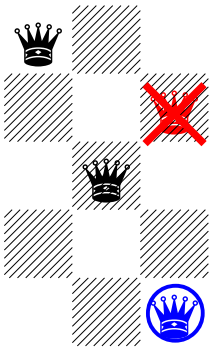
と定義された標準ライブラリの関数である。[1..] は [1, 2, 3, ...] という無限リストの略記法である。

Q 3.14.1

1. all odd [1,3,5] の値は何か?
2. all (>1) [2,0,3] の値は何か?

```
> safe [1,3] 5
True
> safe [1,3] 2
False
```

となる。



Q 3.14.2

1. `safe [1,4] 2` の値は何か?
2. `safe [1,5] 3` の値は何か?

順に、最初の `m` 列のすべての安全な配置を調べていく、そのために、そのようなすべての配置 (のリスト) を返す queens という関数を定義する。

```
1  queens 0  = [[]]
2  queens m  = [ append p [n]
3              | p<-queens (m-1), n<-[1..8], safe p n ]
```

例えば

```
> queens 1
[[1],[2],[3],[4],[5],[6],[7],[8]]}
> queens 2
[[1,3],[1,4],[1,5],[1,6],[1,7],[1,8],[2,4],[2,5],[2,6],[2,7],↙
 [2,8],[3,1],[3,5],[3,6],[3,7],[3,8],[4,1],[4,2],[4,6],[4,7],↙
 [4,8],[5,1],[5,2],[5,3],[5,7],[5,8],[6,1],[6,2],[6,3],[6,4],↙
 [6,8],[7,1],[7,2],[7,3],[7,4],[7,5],[8,1],[8,2],[8,3],[8,4],↙
 [8,5],[8,6]]
```

となる。そうすると `head (queens 8)` で最初の解を求めることができる。

```
> head (queens 8)
[1,5,8,6,3,7,2,4]
```

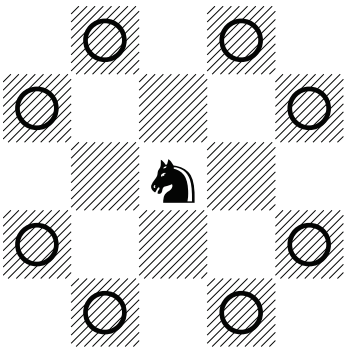
遅延評価を用いているので、最初の解を求めるためには本当に必要な部分の簡約しか行なわない。つまり、上の `[1,5,8,6,3,7,2,4]` を求めるのに、`queens 7` の計算をすべて行なっているわけではなく、この最初の解を求めるのに必要なだけの部分の計算をしている。これは、`queens 7` を完全に計算させると `head (queens 8)` よりも計算に時間がかかることからもわかる。ここでは、遅延評価は Prolog でいうところの**後戻り (バックトラック、 backtrack)** と同じような効果を実現する。つまり、1 つだけ解を求める計算とすべての解を求める計算を別々に記述する必要はなく、しかも、1 つだけ解を求めるためには、それに必要なだけの計算しか行なわない

ちなみに `queens 8` を計算させると、

```
> queens 8
[[1,5,8,6,3,7,2,4],[1,6,8,3,7,4,2,5],
(略)
,[8,3,1,6,2,5,7,4],[8,4,1,3,6,2,7,5]]
```

解はすべてで 92 個あることがわかる。

問 3.14.3 他の言語で、8 クイーンを実装せよ。可能ならば、後戻りをシミュレートして、一つの解、すべての解をおなじプログラムで効率良く求められるようにせよ。



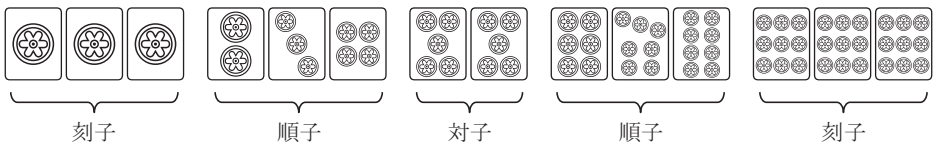
問 3.14.4 （ナイト巡回）ナイトは、右の図の中央のマスから○印のマスへ移動できるチェスのコマである。5×5 マスのチェス盤で、あるマス（例えば、左上隅）から始めてすべてのマスを 1 回ずつ訪れる経路を求めるプログラムを作成せよ。

問 3.14.5 （アガリ判定）

1. ソートされた整数のリストのなかで、3 つ並びの数、2 つの同じ数を見つけてリストアップする関数 shuntsu, toitsu をそれぞれ定義せよ。

```
Prelude> shuntsu [1,3,3,4,5,7,8,9]
[[3,4,5],[7,8,9]]
Prelude> toitsu [1,3,3,4,5,7,7,9]
[[3,3],[7,7]]
```

2. 14 個の要素を持つ Int のソートされたリストがマージャンの清一色（チンイーソー）のアガリ形になっているかを判定する関数 chinitsu を定義せよ。ただし、アガリ形とは、順子（3 つ並びの数）または刻子（3 つの同じ数）が 4 つ、対子（2 つの同じ数）が 1 つ、そろった形である。



3.15 さらに詳しく知りたい人のために ...

文献 [1] は、Haskell に関するもっとも主要な情報源である。文献 [2] は日本語での Haskell の主要な情報源である。文献 [3] は Haskell の仕様書、Haskell を利用す

る上での基本ドキュメントである。文献 [4] は、もうひとつの Haskell のメジャーな処理系 Hugs のユーザーマニュアルである。文献 [5] は、Haskell の実装について知りたい人にお勧めする。文献 [6] は、遅延評価を実際のプログラムでどのように用いるかを解説している。文献 [7] はリストの内包表記を解説している。文献 [8] は、情報処理学会の会誌「情報処理」に 2005 年 4 月から 2006 年 3 月まで連載された Haskell に関する記事である。文献 [9] は Haskell を設計した中心人物の一人による Haskell の設計の “回顧”（と “展望”）である。文献 [10] は Haskell でなく、Miranda という言語を使っているが関数プログラミングに関する、とても有名な良書である。文献 [11, 12, 13] は日本語の初級者向けの解説書である。

この章の参考文献

[1] 「Haskell – A Purely Functional Language featuring static typing, higher-order functions, polymorphism, type classes and monadic effects」http://www.haskell.org/

[2] 「Programming in Haskell」http://www.sampou.org/cgi-bin/haskell.cgi

[3] Simon Peyton Jones, John Hughes 他「Haskell 98: A Non-strict, Purely Functional Language」
1999 年 2 月, http://www.haskell.org/onlinereport/

[4] Mark P Jones, Alastair Reid 他「The Hugs 98 User Manual」
http://cvs.haskell.org/Hugs/pages/hugsman/

[5] Simon Peyton Jones, David Lester 「Implementing Functional Languages」
Prentice Hall, 1992 年
http://research.microsoft.com/Users/simonpj/Papers/pj-lester-book/

[6] John Hughes 「Why Functional Programming Matters」
1989 年, http://www.md.chalmers.se/~rjmh/Papers/whyfp.html
日本語訳 – 山下 伸夫 訳 「なぜ関数プログラミングは重要か」
http://www.sampou.org/haskell/article/whyfp.html

[7] Philip Wadler 「List Comprehensions」
（Simon Peyton Jones 「The Implementation of Functional Programming Languages」
Prentice Hall, 1987 年
http://research.microsoft.com/users/simonpj/Papers/slpj-book-1987/
のなかの第 7 章）

[8] 和田 英一 他 「Haskell プログラミング」
http://www.ipsj.or.jp/07editj/promenade/

[9] Simon Peyton Jones 「Wearing the hair shirt – A retrospective on Haskell」
2003 年, http://research.microsoft.com/~simonpj/papers/haskell%2Dretrospective/

[10] Richard Bird, Philip Wadler 著 武市 正人 訳 「関数プログラミング」
近代科学社, 1991 年 4 月, ISBN4-7649-0181-1

[11] 向井 淳 「入門 Haskell はじめて学ぶ関数型言語」
毎日コミュニケーションズ, 2006 年 3 月, ISBN4-8399-1962-3

[12] 山下 伸夫（監） 青木 峰郎（著）
「ふつうの Haskell プログラミング ふつうのプログラマのための関数型言語入門」
ソフトバンク クリエイティブ, 2006 年 6 月, ISBN4-7973-3602-1

[13] Graham Hutton 著 山本 和彦 訳 「プログラミング Haskell」
オーム社, 2009 年 11 月, ISBN978-4-274-66781-5