第2章 関数型言語 Haskell とは

Haskell は＿＿＿＿＿＿と呼ばれ、ラムダ計算を基本としながら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。

他のプログラミング言語と比べたときの特徴は、以下のようになる。

1. （C 言語と比べると）＿＿＿＿(garbage collection) を持ち、プログラマーがメモリーの管理に煩わされることがない。
2. （JavaScript, Python などと比べると）＿＿＿＿＿＿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. 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（＿＿＿＿, higher-order function）。
4. 多相型 (polymorphic type) を許すことにより、汎用性の高い関数を定義することができる。
5. 型推論 (type inference) 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 (type class) など、型システムが発達している。
6. 代数的データ型 (algebraic data type) 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。
7. 代数的データ型に対してパターンマッチング (pattern matching) による場合分けで処理を定義することができる。
8. 式に＿＿＿＿(side effect) 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。
9. 遅延評価 (lazy evaluation) を採用し、グラフ簡約 (graph reduction) 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。

2.1 Haskell 処理系の入手とインストール

Haskell のインストーラーとして、ここでは GHCup を利用することにする。GHCup は Haskell 処理系の GHC (Glasgow Haskell Compiler, Haskell の処理系の事実上の標準（デ ファクトスタンダード）)、 Cabal（Haskell のパッケージマネージャー）などをインストールして管理することができる。 GHCup は `https://www.haskell.org/ghcup/`からインストールすることができる。

Windows の場合、インストールは PowerShell から指示されたコマンドを実行するだけである。

また Linux なども上記のホームページから、指示に従えばインストールすることができる。

2.2 GHCi のコマンド

GHC は、多くのプログラムの集合体であり、`gcc` のように実行可能形式を出力するバッチコンパイラー（`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。（プロンプトはバージョンによって異なる場合がある。）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`	ディレクトリーを変更する。
`:!` *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
```

（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。暴走したときは、Ctrl-c で中断できる。）

Haskell のプログラムソースファイルには通常 `.hs` という拡張子をつける。

2.3 Haskell のプログラムの基本

Haskell の仕様書は https://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（破壊的代入はできない）。

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ときは次の形式になる。

```
{
  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、「'」（アポストロフィー）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の名前に用いる。

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
```

というヘッダー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、

```
1 trivial x = x
2 twice x   = 2 * x
3 foo x y   = 2 * x + y
```

のように書くことができる。関数 trivial 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。

これらは、C ならば、

```
1 int trivial(int x) { return x; }
2 int twice(int x) { return 2 * x; }
3 int foo(int x, int y) { return 2 * x + y; }
```

という関数に相当する。(Haskell では x, y が int 型という制限はないが、C ではこのように型の制限のない関数を定義することはできないので、便宜上 int 型にしている。)

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

関数の適用（呼出し）は "twice 2" や "foo 3 4" のように関数と実引数を、（必要なら空白で区切って）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丸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) (trivial 3)"のように丸括弧を使用することはできる。（この値は 11 になる。）

Q 2.3.1 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、名前のない関数を表すのにラムダ式というラムダ計算 (λ-calculus) に由来する記法を用いる。

「`\`」（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「`->`」、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「`->`」（「アロー」と読む）を用いる点が異なる。）

当然ながら、仮引数の名前は（Haskell の識別子名の規則に従い、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 (alpha conversion) と言う。

上記の関数定義は、次のラムダ式を使った定義とまったく等価である。

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

ラムダ式の例

1. `\ x -> x` — これは `x` という引数を受け取って、`x` をそのまま返すので、恒等関数を表している。

   C ならば、

   ```
   int trivial(int x) { return x; }
   ```

   という関数 `trivial` に相当する。

2. `\ x y -> x` — これは、`x` と `y` という引数を受け取り、`x` を返す関数である。C の記法では

   ```
   int baz(int x, int y) { return x; }
   ```

   と表される 2 引数の関数 `baz` に相当する。

Haskell は多引数関数と"関数を返す関数"とを同一視する。つまり、`\ x y -> x` は、`\ x -> (\ y -> x)` と同等である。このように、多引数関数を"関数を返す関数"として表現することを、＿＿＿＿と言う。カリー (Curry) は、著名な数理論理学者 Haskell B. Curry (1900–1982) の名にちなんでいる。

`\ x -> (\ y -> x)` として見る場合、これは、`x` という引数を受け取り、`\ y -> x` という関数を返す関数である。そして、`\ y -> x` という関数は、`y` という引数を受け取り、（これを無視して）`x` を返す関数である。つまり、式全体は高階関数である。

3. `\ f x -> f (f x)` — 関数 `f` とデータ `x` を受け取って、`f` を `x` に 2 回適用する関数である。

> **Q 2.3.2** 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。

- `--` というかたち（一行コメント）
- `{- -}` というかたち（複数行コメント）

2.4 組み込みのデータ型と演算子

2.4.1 基本的なデータ型と演算子

Haskell では真偽値（`Bool`）、整数（`Int`, `Integer`—`Int` は固定（有限）精度の整数, `Integer` は任意精度（メモリーが許す限り、いくらでも桁数を大きくとることができる。）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（`12`, `3.14`, `'x'` など）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。（Lisp は `(+ 1 (* 2 3))` のような前置記法で書く。）

`if` ～ `then` ～ `else` 式

`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` のときは式$_2$、`False` のときは式$_3$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。

比較演算子「`==`」，「`<`」，「`<=`」など、論理演算子「`&&`」，「`||`」は C や Java と同じである。

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＿＿＿＿＿＿。そのため、`fact (n - 1)` は `fact n - 1` と書くことはできない。後者は＿＿＿＿＿＿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。

Q 2.4.1 `fact 100` を計算してみよ。

（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。

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

なお、`otherwise` は「それ以外は」という意味だが、標準ライブラリーで単に `True` と定義されている変数である。

Q 2.4.2 フィボナッチ数列

$$\begin{cases} a_n &= 1 & (n = 0, 1) \\ a_n &= a_{n-2} + a_{n-1} & (n \geq 2) \end{cases}$$

を計算する関数 `fib` を（効率を気にせず単純な再帰で）定義せよ。ただし、`fib 0 = 1`, `fib 1 = 1`, `fib 2 = 2`, `fib 3 = 3`, `fib 4 = 5`, `fib 5 = 8`, ... である。（あとで同じ計算を繰り返すことのない、効率の良い定義の仕方も紹介する。）

注意: 定義を誤って処理系が暴走したら、通常 Ctrl-c で止めることができる。

2.4.2 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（スキーム）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ー関数が用意されている。

リストは、空（くう）リスト`[]`とコンス (cons) と呼ばれる演算子「`:`」から構成される。この 2 つをリストの構成子 (constructor) と呼ぶ。構成子は、与えられた引数をグループ化して、区別するためのタグを付ける。

- 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`右結合`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。（Python や JavaScript もほぼ同じ記法を使っている。）例えば `[1,2,3,4]` は、先頭の要素が `1`、次の要素が `2`、…というリストで、 `1:2:3:4:[]` のことである。

> **Q 2.4.3** ① `[1,2,3]` と ② `[[1,2],[3,4]]` を「`:`」と 「`[]`」（と丸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`[Integer]`型、要素の型が `Double` 型の場合リストの型は `[Double]`型と書き表される。それぞれ list of integer, list of double と読む。 型の異なる要素が混在するリスト（ヘテロジニアス・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。

`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。

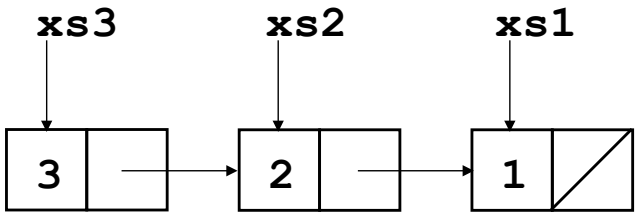
> **Q 2.4.4** ① `[False,True]` の型と② `["Kagawa","University"]`の型を書け。

箱ポインター記法

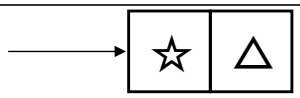
リストを、箱と矢印を用いた図（箱ポインター記法、box-pointer 記法）で表すことがある。例えば、次のように構成されたリストの場合、

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

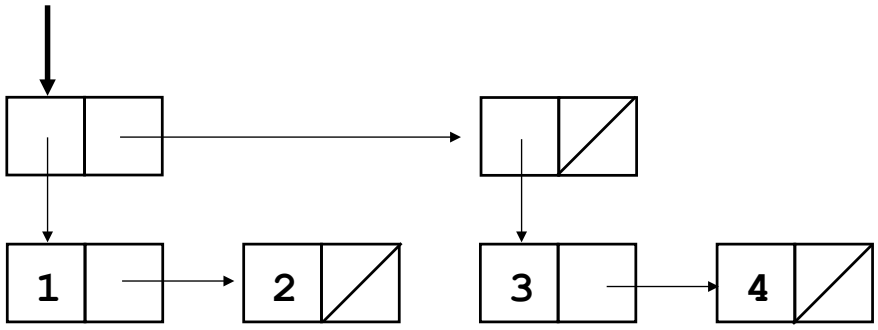
次の図のようになる。



1 つの「:」を 2 個の正方形の箱がつながった箱のペア（コンスセル）**への矢印**で表す。箱の中身は、1, 2, 3 などの数、他の箱のペアへの矢印などである。（箱の中に他の箱のペアがまるごと入ることはない。箱に何も入らないということもない。リストの場合、箱をペアではない単独で使用することはない。）
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 2.4.5 次のリストを箱ポインター記法で書け。

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

（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。（Haskell には遅延評価があるので、実際はもう少し複雑なデータ構造が必要になる。）

```c
struct _list {
  int head;
  struct _list* tail;
};

typedef struct _list* list;
```

この `_list` という構造体は、`head` と `tail` という 2 つのメンバーを持つ。このうち `tail` は現在定義されている型へのポインター型である。また `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。

C 言語でリストを扱う場合は、`malloc` で割り当てたメモリーをいつ `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、不要になったメモリー領域は自動的に回収されることになっている。

2.4.3 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 `(Num t, Char)` と表記される。また、`(2,'b',[3])` という式の型は `(Num t, Char, [Num t])` と表記される（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。 例えば、`(1,'a')` は、本当は `Num t => (t, Char)` という型を持つ。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。 なお、型クラスを学習するまでは型をむやみに明示しないことを推奨する。（型を限定することになりかねない。））。

> **Q 2.4.6** 次の式の型は何か？
>
> ① `(False,"Hello")` ② `('X',("Aloha",False))`
>
> ---
>
> ---

`()` 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。

2.4.4 関数型

関数の型は「`->`」という記号（「アロー」と読む）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、このように右結合として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＿＿＿＿(type variable) である。これらの型変数は使用するときに、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＿＿＿＿(polymorphic type) 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。今でこそ、多相型は多くのプログラミング言語に取り入れられているが、ML が多相型の先駆けとなった言語であった。

なお、変数の型をプログラム中に明示したい場合「`::`」を使って、

```
変数名 :: 型
```

と書く。例えば `trivial` や `fact` の型を明示しておきたい場合は、

```
1 trivial :: a -> a
2 trivial 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、変数や関数の型はプログラマーが明示しなくても、 Haskell 処理系が推論してくれる。この仕組みを＿＿＿＿(type inference) という。

2.5 パターンマッチング

Haskell では、関数定義の仮引数の部分に＿＿＿＿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ことが可能である（一般的に 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であり、 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。）。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。

```haskell
fact 0 = 1
fact n = n * fact (n - 1)

-- Prelude の length とほぼ同じ
myLength []     = 0
myLength (x:xs) = 1 + myLength xs

-- ネストしたパターンの例
foo []                      = -1
foo ((x,y):xys) | odd x     = y
                | otherwise = foo xys
```

関数名 パターン$_{11}$ … パターン$_{1m}$ = 式$_1$
関数名 パターン$_{21}$ … パターン$_{2m}$ = 式$_2$
…
関数名 パターン$_{n1}$ … パターン$_{nm}$ = 式$_n$

呼出し時には、関数定義の上から下の順に、関数の実引数をパターンと照合し、マッチした定義の右辺が評価される。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** が標準ライブラリーに用意されている。）

前述したように if ～ then ～ else **式** は Haskell では特殊形式として扱う必要はなく、以下のように同等の働きをする関数 ifte を定義することができる。

```haskell
ifte True t e  = t
ifte False t e = e
```

なお、GHC はパターンマッチングがすべての場合を尽くしていなくても、通常警告を出力しないが、コマンドラインオプション -fwarn-incomplete-patterns を指定することで警告を出すようにすることができる。

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、

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

と書くことができる。

次の case ～ of 式もパターンマッチングを行なう。（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。

また `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）やラムダ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 2.5.1 リスト中の数の和、積を求める関数 `mySum`, `myProd` をパターンマッチングを使って定義せよ。（同等の関数 `sum`, `product` は標準ライブラリーに用意されている。）

問 2.5.2 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`
3. 最後に現れる一つの要素だけ取り除いたリストを返す関数 `deleteLast`（`reverse` は用いない）

をそれぞれ定義せよ。等しい要素が一つもないときは、元と同じリストを返すようにせよ。

問 2.5.3

1. 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 (Horner's rule) を使う。つまり、多項式 $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$ を

$(\ldots(a_nx+a_{n-1})x+\ldots+a_1)x+a_0$ の順で計算する。例えば `fromBin [True,False,True,False]` は $((1\times 2+0)\times 2+1)\times 2+0$、 `fromBin [True,True,False,True]` は $((1\times 2+1)\times 2+0)\times 2+1$ となるように計算する。

2. 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、`fromBinRev [True,True,False,True]` は $1+2\times(1+2\times(0+2\times 1))=11_{(10)}$となる。

3. 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 $1+2\times 10+3\times 10^2+4\times 10^3=4321$になる。

問 2.5.4 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。

2.6 帰納法による証明

関数型言語の利点は、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、数学の等号「$=$」と C 言語の「=」は意味が異なる。つまり、`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。例えば、「`c = getchar(); putchar(c); putchar(c);`」と「`putchar(getchar()); putchar(getchar());`」は意味が異なる。）

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。

問 2.6.1 （復習）漸化式 $a_0=1, a_n=2a_{n-1}+1\ \ (n\geq 1)$ で定義される数列の一般項が $a_n=2^{n+1}-1$ で表されることを数学的帰納法を用いて証明せよ。

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 `rev0`:

```haskell
-- (++) は Prelude に定義済み
(++)          :: [a] -> [a] -> [a]
[] ++ ys       = ys                        -- (++)-(1)
(x:xs) ++ ys   = x : (xs ++ ys)            -- (++)-(2)

rev0        :: [a] -> [a]
rev0 []      = []                          -- rev0-(1)
rev0 (x:xs) = (rev0 xs) ++ [x]             -- rev0-(2)
```

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。（注: `(++)` の計算量は左オペランドのリストの長さに比例する。）

そこで次のような定義を考える。

```
-- shunt は reverse の補助関数
shunt          :: [a] -> [a] -> [a]
shunt ys []      = ys                      -- shunt-(1)
shunt ys (z:zs)  = shunt (z:ys) zs         -- shunt-(2)

-- reverse は Prelude に定義済み
reverse    :: [a] -> [a]                   -- reverse-(1)
reverse xs = shunt [] xs                   -- reverse-(2)
```

この `reverse` 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。

この `reverse` と `rev0` 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

```
reverse xs = rev0 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 = (rev0 xs) ++ ys     -- ☆
```

これと、あとで証明する `(++)` に関する定理を合わせれば、 `ys` に `[]` を代入すれば `reverse xs = rev0 xs` が証明できる。

この補助定理は、 ______ に関する帰納法で証明することができる。

証明:

xs = [] のとき:

______

______

______

______

xs = z:zs のとき:

______

______

______

______

______

______

______

問 2.6.2 すべての有限リスト `xs` について、

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

が成り立つことを、`xs` に関する帰納法で証明せよ。（どこで `(++)-(1)`, `(++)-(2)`, 帰納法の仮定を使用するか明示せよ。）

例えば、（数独のような）パズルを解くプログラムで、片方は明らかに正しいが、すべての場合をしらみつぶしに調べるため、計算に天文学的な時間がかかり過ぎて実用的ではない、もう片方はトリッキーなところがあって正しいかどうか明らかでないが、高速で実用的な時間で解ける、という場合に、両者が同値であることを証明できるかもしれない。

2.7 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（標準ライブラリー）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]
```

```
map f []     = []
map f (x:xs) = f x : map f xs

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWith f _       _     = []

take :: Int -> [a] -> [a]
take 0 _       = []
take _ []      = []
take n (x:xs) = x : take (n - 1) xs

filter :: (a -> Bool) -> [a] -> [a]
filter p []     = []
filter p (x:xs) = if p x then x : filter p xs
                         else filter p xs

iterate :: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f x []     = x
foldr f x (y:ys) = f y (foldr f x ys)

foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f x []     = x
foldl f x (y:ys) = foldl (f x y) ys

concat :: [[a]] -> [a]
concat []       = []
concat (xs:xss) = xs ++ concat xss
```

例えば map はリストの各要素に同じ関数を適用する。

map f [x_1,x_2,…] ⇨ [f x_1, f x_2,…]

その 2 引数版が zipWith である。

zipWith f [x_1,x_2,…] [y_1,y_2,…] ⇨ [f x_1 y_1, f x_2 y_2,…]

さらに、take はリストのいくつかの先頭の要素を取り出す、filter はリストの中から条件を満たすものだけを列挙する、iterate は初項と漸化式から数列（数とは限らないが…）を生成する、 foldr と foldl はそれぞれ右と左から畳み込む関数である。

foldr (\ x y -> x ⊛ y) x [y_1,y_2,y_3,… y_n]
⇨ y_1⊛(y_2⊛(y_3⊛(…⊛(y_n⊛x)…)))
foldl (\ x y -> x ⊘ y) x [y_1,y_2,y_3,… y_n]
⇨ (…(((x⊘y_1)⊘y_2)⊘y_3)⊘…)⊘y_n

また、concat はリストのリストの各要素を連接することでフラットなリストにする。

concat [[$x_{1,1}$,$x_{1,2}$,…],[$x_{2,1}$,$x_{2,2}$,…],[$x_{3,1}$,$x_{3,2}$,…],…]
⇨[$x_{1,1}$,$x_{1,2}$,…,$x_{2,1}$,$x_{2,2}$,…,$x_{3,1}$,$x_{3,2}$,…]

実行例をまとめると以下のようになる。

`map (\ x -> x * x) [1,3,2]`	⇨ _____
`zipWith (\ x y -> x * y) [3,2,4] [2,7,5]`	⇨ _____
`filter odd [1,3,2]`	⇨ _____
`take 4 (iterate (\ x -> x * x) 2)`	⇨ _____
`foldr (\ x y -> x + 10 * y) 0 [2,3,5]`	⇨ _____
`foldl (\ x y -> 10 * x + y) 0 [2,3,5]`	⇨ _____
`concat [[2,3,5],[0],[1,3]]`	⇨ _____

問 2.7.1 上記の `take` の反対に、リストの最初の n 個の要素を取り除く `drop :: Int -> [a] -> [a]` というライブラリー関数があるが、これと同じ動作をする関数 `myDrop` を定義せよ。例えば、 `myDrop 2 [7,9,6,5,3]` は `[6,5,3]` である。

2.8 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、

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

通常は `zip [1,2] [3,4]` のように前に関数名を書くが、これを `[1,2] `zip` [3,4]` と引数の間に書くことができる。

中置記法で用いる演算子に対して、`infixl`, `infixr`, `infix` という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ー）では次のように宣言されている。

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

ここで `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「<」は非結合なので `1 < x < 2` は構文エラー

になる）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 7 なので、6 の「+」よりも結合力が強い。

Q 2.8.1 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`

一方「!#$%&*+./<=>?@\^|-~:」の中の文字を用いた識別子は最初から中置記法の演算子として扱われる。例えば、「^」演算子は次のように定義できる。（実際には Prelude の中で定義されている。）

```
(^) :: Integer -> Integer -> Integer
x ^ 0  = 1
x ^ n  = x * (x ^ (n - 1))
```

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。あるいは演算子を関数の引数として渡すこともできる。例えば、`zipWith (+) [3,5,1] [6,2,7]` は `[9,7,8]` である。

Q 2.8.2 次の式を通常の中置記法で書け。括弧はできるだけ省略せよ。

1. `(-) ((/) 2 x) 1`
2. `(++) xs ((:) y ys)`

2.9 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。演算子の片方のオペランドだけを書いて丸括弧で囲む。例えば `(2 *)` は 2 倍する関数 `\ x -> 2 * x` であり、 `(/ 2)` は 2 で割る関数 `\ x -> x / 2`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 2.9.1 次のセクションをラムダ式で書け。

1. `([1,2] ++)`
2. `(!! 2)`

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（負の数である）。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。

問 2.9.2 「有用なリスト処理関数」で紹介した `map` などの関数（とそれ以前に紹介した `length`, `id` などの関数）を使って、次のような関数を再帰呼出しを使わずに定義せよ。

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`（リストの要素数が異なる場合は、短いほうにあわせる）

   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。

2.10 局所的定義

次のように `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。

```
pow4 x  = let y = x * x in y * y
myHead ys = let (z:zs) = ys in z
-- 同等の関数 head は Prelude に定義済み
-- myHead (x:xs) = x と定義することもできる
```

などである。このとき `y` や `z` や `zs` はスコープが限られるので、関数の定義の外では未定義の変数のままである。

Q 2.10.1 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`（「^」や「**」演算子を使わずに）

2. リストの尾部（頭部を除いた残り）を求める関数 `myTail`

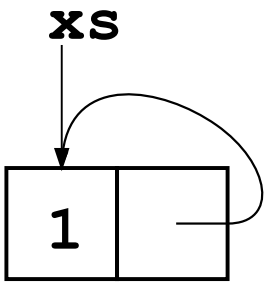
この例では 4 乗する関数（`pow4`）を定義しているが、 `y * y` の部分が変数 `y` の有効範囲（　　）に属している。実は、さらに"変数の定義"の右辺の部分（この例では、　　の部分）もスコープに属している。これは次の例でわかる。

```
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。（この例のように出力が止まらなくなったときは Ctrl-c で中断する。）

```
1 Prelude> repeat 1
2 [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,…
```

このリストは、次の箱ポインター記法で表される。



（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 2.10.2 `repeatList [2,5]` が `[2,5,2,5,2,5,2,5,2,5,…]`、`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。なお、引数のリストの要素数は任意である。

ヒント: `(++)` を使用せよ。

問 2.10.3 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。

注: あまり大きなリストで試すと、非常にメモリーを消費して時間がかかる。要素数 10 くらいまでにとどめておくこと。

（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できる。

```
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、where で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できる。

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

ただし、`where` による局所的定義は複数のパターンマッチにまたがることはできない

```
foo []     = … -- foo-(1)
foo (x:xs) = … -- foo-(2)
  where bar = …
```

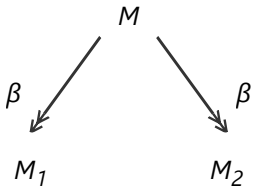
この例で `bar` が有効なのは `foo-(2)` のほうだけである。

2.11 Haskell の評価戦略

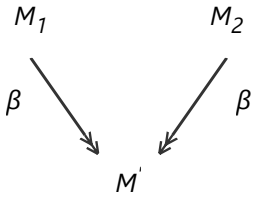
Haskell の式の評価は、`(\ x -> M) N` という部分式を、関数の戻り値 M のなかの 仮引数 x の"自由な"出現を実引数 N に置き換えた式に書き換える。これを β 簡約 (beta reduction) という。例えば、"`(\ x -> x + 2) 3`" を `3 + 2` に置き換える。

これ以上 β 簡約ができない式を正規形という。

評価順序を定めなければ、一つの式に幾通りもの β 簡約が可能なことがある。（例: `(\ x -> x * x) ((\ x -> x + 1) 2)`）このとき、異なる β 簡約を選ぶと、評価の結果が別の形に枝分かれしてしまう。（例: `(\ x -> x * x) 3)` と `((\ x -> x + 1) 2) * ((\ x -> x + 1) 2)`）



しかし、うまく何回かβ簡約を行なうと、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（仮引数の名前付けによる違いを除く）ということを保証している。

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 _____(leftmost strategy) 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、内部的には次に説明するように __ _____(graph reduction) を用いる 。

例えば、次のように定義された `square` という関数を考える。

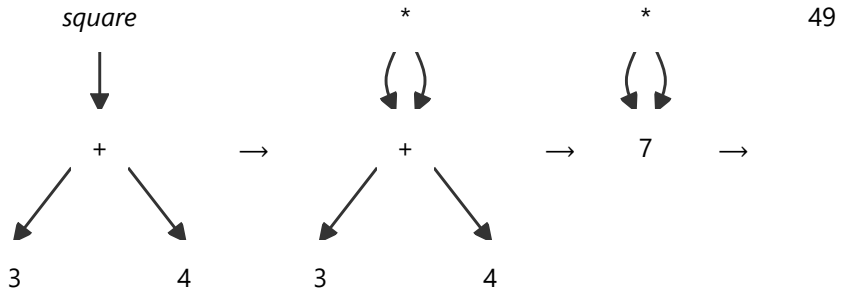
```
1 square x = x * x
```

最左戦略では`square (3 + 4)`という式は次のように計算することになる。

$$\begin{aligned} square\ (3+4) &= (3+4) * (3+4) \\ &= 7 * 7 \\ &= 49 \end{aligned}$$

つまり、`square` の引数である `(3 + 4)` は計算されないまま、まず `square` 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 `*` の引数だから必要とわかる）はじめて `3 + 4` が計算される。この方式をナイーブに実行すると、 `3 + 4` 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、`3 + 4` を一度しか計算しないようにしている。



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。

```
from :: Integer -> [Integer]
from n = n : from (n + 1)
-- from n = iterate (\ x -> x + 1) n でも同じ
```

すると `from 1` は `[1,2,3,…]` という無限リストだが、この無限リストを部分式として用いている `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)))
→ 1:2:(take 1 (from (1 + 1 + 1)))
→ …
→ 1:2:3:(take 0 (from (1 + 1 + 1 + 1)))
→ 1:2:3:[] (= [1,2,3])
```

のように有限時間で計算できる。例えば `take` の計算をするために、第１引数が０でないときは、第２引数が空リストかそうでないかを知る必要がある。また、最終的に画面に結果を表示するために、`(1 + 1)`, `(1 + 1 + 1)` などを評価する必要がある。

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 (Hughes 1989) 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。副作用は実行される順序によって結果が異なるため、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。また、遅延評価では、メモリーの使用量について見積もることも難しくなる場合が多い。

問 2.11.1 フィボナッチ数列 1, 1, 2, 3, 5, ... の無限リストとして `fib :: [Integer]` を定義せよ。

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: 関数 `zipWith` を使う。

問 2.11.2 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。例えば `merge [2,4,6,8] [3,6,9]` は `[2,3,4,6,8,9]` になる。

問 2.11.3  $2^i \cdot 3^j \cdot 5^k$（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。例えば `take 14 hamming` は `[1,2,3,4,5,6,8,9,10,12,15,16,18,20]` である。 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）、つまり `hamming !! 699`が 5898240 であることを確認せよ。

ヒント: 関数 `map` と、以前の問の `merge` を使う。

2.12 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（とういこうぶん）(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

```
1 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。

リストの内包表記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 <- 式
```

のいずれかである。生成式の「`<-`」の左辺にはパターンを書くことも可能である。生成式の左辺に現れる変数のスコープは、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 2.12.1 次の内包表記の値は何か?

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

Q 2.12.2 非負の整数 n を受け取り、$0 \leq x \leq y \leq n$ 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 2.12.3 非負の整数 n を受け取り、$1 \leq x < y < z \leq n$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関数を用いて翻訳することができる。

```haskell
unit :: a -> [a]    -- 要素数 1 のリストを返す
unit a = a : []

bind :: [a] -> (a -> [b]) -> [b]
bind []     _ = []
bind (x:xs) f = (f x) ++ (bind xs f)
```

例えば `bind [1,7,5] (\ x -> [x, x - 1])` は `[1,0,7,6,5,4]` になる。

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。）

```
[t | ]              ⇒ unit t
[t | x <- u, P]     ⇒ bind u (\ x -> [t | P])
[t | b, P]          ⇒ if b then [t | P] else []
[t | let decls, P] ⇒ let decls in [t | P]
```

ただし、*b* は `Bool` 値の式、 *P* や *Q* 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 []))
```

内包表記の使用例

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

（ただし、ちゃんと動作する定義ではあるが、効率的には改善の余地はある。）

問 2.12.4 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 2.12.5 素数列 `[2,3,5,7,11,…]` の無限リストとなるように `primes :: [Integer]` を定義せよ。（内包表記を用いても、用いなくても良い。）

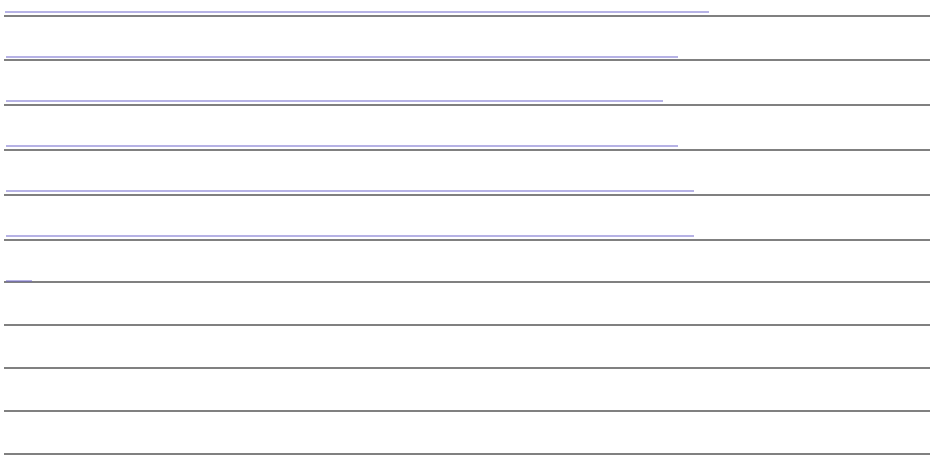
考え方:

“エラトステネス (Eratosthenes, 275 BC – 194 BC) 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。）なお、割り算の余りを求めるには `` `mod` `` 演算子を用いる。例えば、 `11 `mod` 3` は `2` である。
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、遅延評価のおかげで問題はない。

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いくつの配列を用意すれば良いのかわからないので急に難しくなる。



問 2.12.6  上記の `sieve` を割り算（`div` や `mod` や `rem`）を使わずに定義せよ。（**ヒント:** 補助関数として、 2 つの昇順のリスト `xs`, `ys` を受け取り、 `xs` に含まれるが、`ys` に含まれない要素を昇順のリストとして返す関数を定義する。）

問 2.12.7 Haskell 以外の言語で、無限リストをシミュレートする方法を調査または考察せよ。またその方法を用いて、その言語で素数列を生成せよ。

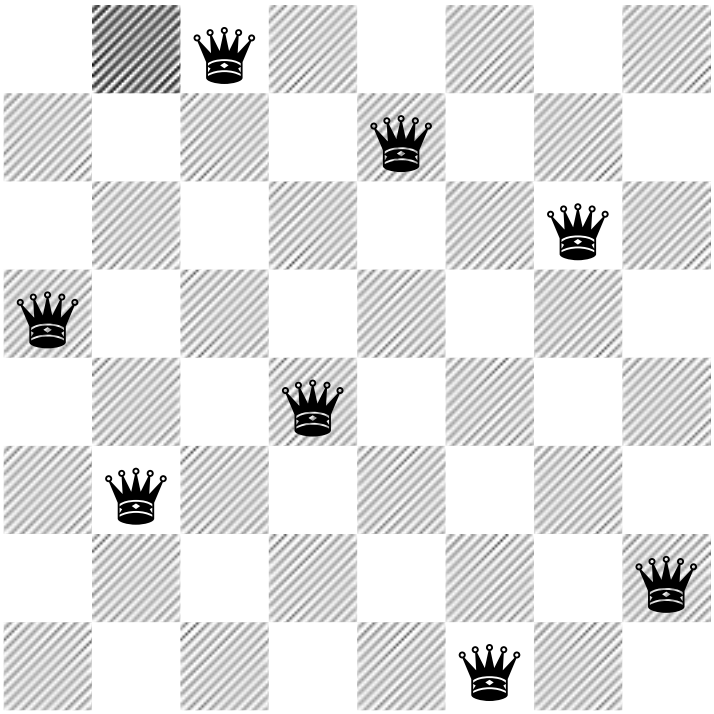
2.13 エイト (8) クイーンの問題

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。

エイト (8) クイーンの問題は 8 個のクイーンを、お互いに取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。将棋の飛車と角を兼ねたような動きであ

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大きな役割を果たす。

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。つまり、 `[(1,4),(2,6),(3,1),(4,5),(5,2),(6,8),(7,3),(8,7)]` という座標を略記しているということである。



まず左の k 列までクイーンをお互いに取れないように配置して、第 k + 1 列めにさらにクイーンを配置できるかどうか確認していく。次に示す `safe` は `safe p n` が `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。

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

check (i,j) (m,n) = j == n
                    || (i + j == m + n)
                    || (i - j == m - n)
```

ここで、`all` は

```
all           :: (a -> Bool) -> [a] -> Bool
all p []       = True
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ーの関数である。また、`[1..]` は `[1,2,3,…]` という無限リストの略記法である。つまり、 `check` は横または斜めで同じ線上に乗っていないことをチェックしている。

Q 2.13.1

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 2.13.2

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。

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

つまり、最初の 0 列の安全な配置は 1 つあって、それは空リスト「`[]`」で表される。 `m - 1` 列目までの安全な配置を求められれば、`safe` 関数を使って、第 `m` 列目の安全な配置を求めることができる。

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。（GHCi で `:set +s` というコマンドを実行すると、以降、評価に書かた時間やメモリー量が表示されるようになる。）

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、

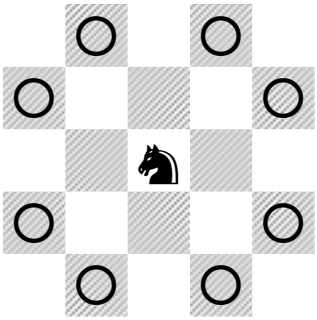
```
> queens 8
[[1,5,8,6,3,7,2,4],[1,6,8,3,7,4,2,5],
(略)
,[8,3,1,6,2,5,7,4],[8,4,1,3,6,2,7,5]]
```

解はすべてで 92 個あることがわかる。

問 2.13.3 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 2.13.4

（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。

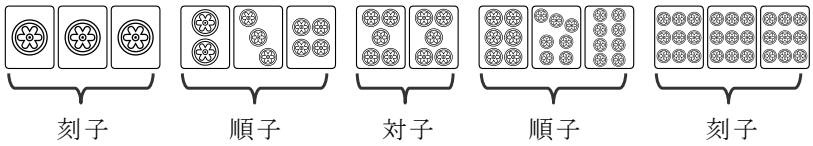


問 2.13.5 （アガリ判定）

1. ソートされた整数のリストのなかで、3 つ並びの数、3 つの同じ数、2 つの同じ数を見つけてリストアップする関数 `shuntsu`（順子）、 `kotsu`（刻子）、 `toitsu`（対子）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 個の要素を持つ整数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。



2.14 さらに詳しく知りたい人のために...

文献（**Haskell**）は、Haskell に関するもっとも主要な情報源である。文献（**山下**）は日本語での Haskell の主要な情報源である。文献（**Peyton Jones et al. 1999**）は Haskell の仕様書で、Haskell を利用する上での基本ドキュメントである。文献（**Jones et al. 1999**）は、Haskell のかつて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（**Peyton Jones & Lester 1992**）は、Haskell の実装について知りたい人にお勧めする。文献（**Hughes 1989**）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（**Wadler 1987**）はリストの内包表記を解説している。 文献（**Peyton Jones 2003**） 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（**Bird 2014**）は関数プログラミングに関する、著名な研究者による良書である。文献（**向井 2006**）は日本語の初級者向けの解説書である。

（**Haskell**） haskell.org, “Haskell — A Purely Functional Language featuring static typing, higher-order functions, polymorphism, type classes and monadic effects” https://www.haskell.org/

（**山下**） 山下 伸夫 「{算法|算譜}.ORG」 https://www.sampou.org/

（**Peyton Jones et al. 1999**） Simon Peyton Jones, John Hughes他, “Haskell 98: A Non-strict, Purely Functional Language” https://www.haskell.org/onlinereport/, 1999 年 2 月

（**Jones et al. 1999**） Mark P Jones, Alastair Reid他, “The Hugs 98 User Manual” https://www.haskell.org/hugs/pages/hugsman/index.html

（**Peyton Jones & Lester 1992**） Simon Peyton Jones, and David Lester, “Implementing Functional Languages” https://www.microsoft.com/en-

us/research/publication/implementing-functional-languages-a-tutorial/, Prentice Hall, 1992 年

（Hughes 1989） John Hughes, “Why Functional Programming Matters” 1989 年, https://www.cse.chalmers.se/~rjmh/Papers/whyfp.html
山下 伸夫 訳 「なぜ関数プログラミングは重要か」 https://www.sampou.org/haskell/article/whyfp.html

（Wadler 1987） Philip Wadler, “List Comprehensions”（Simon Peyton Jones “The Implementation of Functional Programming Languages” Prentice Hall, 1987 年 https://www.microsoft.com/en-us/research/uploads/prod/1987/01/slpj-book-1987.pdf のなかの第 7 章）

（Peyton Jones 2003） Simon Peyton Jones, “Wearing the hair shirt — A retrospective on Haskell” https://www.microsoft.com/en-us/research/publication/wearing-hair-shirt-retrospective-haskell-2003/, 2003 年

（Bird 2014） Richard Bird, “Thinking Functionally with Haskell” Cambridge University Press, 2014 年,
山下 伸夫 訳 「Haskell による関数プログラミングの思考法」 KADOKAWA, 2017 年 2 月, ISBN978-4048930536

（向井 2006） 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

（山下 & 青木 2006） 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

（Hutton 2007） Graham Hutton, “Programming in Haskell” Cambridge University Press, 2007 年,
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