第3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と前章で紹介した（型なし）ラムダ計算との主な違いは

1. 式は＿＿＿＿＿＿＿されている。
2. ＿＿＿＿＿＿＿＿＿＿を定義することができる。
3. ＿＿＿＿＿＿＿＿＿＿＿＿＿を使用することができる。
4. グラフ簡約によって実行される。
5. 中置記法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. ＿＿＿＿＿＿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、関数を値として使うことができる（＿＿＿＿＿＿）。
3. ＿＿＿＿＿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. ＿＿＿＿＿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。＿＿＿＿＿＿など、型システムが発達している。
5. 式に＿＿＿＿＿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. ＿＿＿＿＿＿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。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、ここでは主にプログラミング言語の意味を説明するための超言語として紹介することにする。

3.1　Haskell処理形の入手とインストール

Haskellの処理形としてここではGHC (Glasgow Haskell Compiler[1]) を利用することにする。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。GHCは、`http://www.haskell.`

[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。

org/ghc/ 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 RPM ファイル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。
:?		ヘルプを表示する。
: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
```

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 のプログラムファイルには通常 ____ 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、以下の例でも原則として省略する。省略の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。ただし、通常の変数名は＿＿＿＿（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
```

ラムダ式　Haskell では、関数を表すのに当然ラムダ式に似た記法を用いることができる。ただし、“*λ*” の代わりに “＿”（バックスラッシュまたは “¥”）、“.”（ピリオド） の代わりに “＿＿” を用いる。

ラムダ計算	Haskell
λx.x	＿＿＿＿＿＿
λxy.y	＿＿＿＿＿＿＿＿

関数の適用はラムダ計算と同様、関数と実引数を並べて書くだけである。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。

```
c0 = \ f x -> x
c1 = \ f x -> f x
c2 = \ f x -> f (f x)
true  = \ t f -> t
false = \ t f -> f
```

実は、このようにラムダ式に名前をつける時は、仮引数を “=” の左辺に並べて、

```
c0 f x = x
c1 f x = f x
c2 f x = f (f x)
true t f  = t
false t f = f
```

のように書くことができる。（むしろ、このように書く方が普通である。）

コメント　Haskell のコメントには 2 つの形がある。＿＿＿＿＿＿＿＿＿＿＿＿という形と＿＿＿＿＿＿＿＿＿＿＿＿＿＿という形である。

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, Integer は無限精度）、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＿＿＿＿と＿＿＿＿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。

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＿＿＿＿＿＿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、if〜then〜else 式は、Haskell では後で説明するように特殊な評価順を必要とする特殊形式ではない。

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＿＿＿＿＿＿＿＿することが可能である。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。

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 ___とコンス (cons) と呼ばれる演算子 "_" から構成される。

- 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 _____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、_____のことである。

リスト型はパラメトリックな型である。つまり、リストの要素の型をパラメータとする。要素の型が `Integer` 型の場合、そのリストの型は _____型、要素の型が `Double` 型の場合リストの型は _____型と書き表される。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、

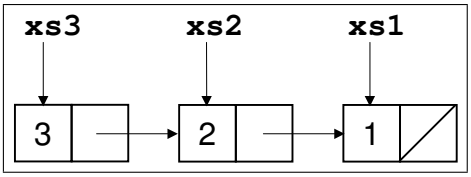
```
type 型名 = 型
```

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。

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

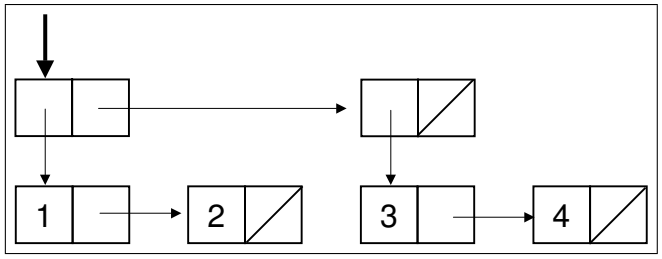
```
xs0 = []
xs1 = 1:xs0
xs2 = 2:xs1
xs3 = 3:xs2
```

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。

また、`[[1,2],[3,4]]` というリストのリストは



という箱矢印記法で表される。

問 3.4.2 次のリストを *box-pointer* 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。

```
struct _list {
  int car;
  struct _list* cdr;
};

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。(箱矢印記法で矢印になっているところが、C 言語ではポインタとして表される。)

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]。

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。多引数関数をカリー化して定義する場合、このように約束しておく方が便利である。

[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を紹介する。

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 fact の型を明示しておきたい場合は、

```
fact :: Integer -> Integer
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、構成子からのみ生成される式である。

```
-- Prelude の length とほぼ同じ
myLength :: [a] -> Integer
myLength []     = 0
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;
```

[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。

```
  ...
  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:xs) -> x
```

などである。（この関数は `head` という名前で標準ライブラリに用意されている。）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。

問 3.5.2 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 *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。

問 3.5.3 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。

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`:

```
-- append は Prelude の (++) 演算子と同じ
append            :: [a] -> [a] -> [a]
append [] ys       = ys
append (x:xs) ys   = x : (append xs ys)

-- reverse は Prelude に定義済み
reverse        :: [a] -> [a]
reverse []      = []
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。

```
-- shunt は rev の補助関数
shunt          :: [a] -> [a] -> [a]
shunt ys []     = ys
shunt ys (x:xs) = shunt (x:ys) xs

rev   :: [a] -> [a]
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、

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 有用なリストの高階関数

次のようなリストに対する高階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。

```
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 xs ys
zipWith f _       _     = []

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
```

3.8 その他の便利な記法

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

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。

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

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。

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。

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

などである。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。

```
-- repeat は Prelude に定義済み
repeat  :: a -> [a]
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
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1, ...
```
[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。）

問 3.8.1 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。

[4] Ctrl-c で中断する。

問 3.8.2 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。）

3.9 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$_{m,n_m}$

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。

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

```
data Direction = Up | Down | Left | Right
```

Up, Down, Left, Right の 4 つがが Direction 型を構成している。

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。

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

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 つのパラメータを取る。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。

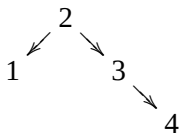
```
Branch :: _____________________________________
```

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。

```
Empty :: Tree a
Branch Empty 1 Empty :: Tree Integer
Branch (Branch Empty "a" Empty)  "b" (Branch Empty "c" Empty) :: Tree String
Branch (Branch Empty 1 Empty) 2 (Branch Empty 3 (Branch Empty 4 Empty))
  :: Tree Integer
```

例えば最後の式の構造は、図で表すと次のようになる。



問 3.9.1 *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
postorder :: Tree a -> [a]     -- 後順走査
reflect   :: Tree a -> Tree a  -- 鏡像
```

3.10 Haskellの評価戦略

Haskell の評価戦略は基本的にラムダ計算のところで紹介した _____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。

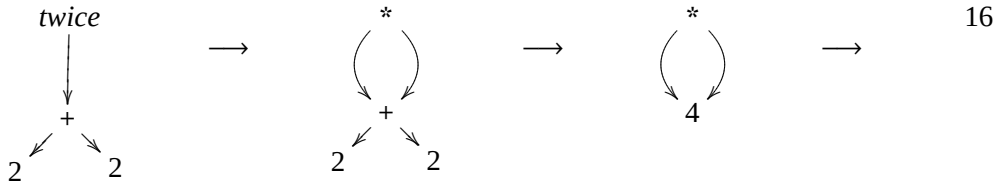
```
twice x = x*x
```

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。

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

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。



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。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)

-- take は Prelude に定義済み
take :: Integer -> [a] -> [a]
take 0 _      = []
take _ []     = []
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]
```

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。

問 3.10.1 *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 3.10.2 *fib* をフィボナッチ数列の無限リストとして定義せよ。

問 3.10.3 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 3.10.4 $2^i \cdot 3^j \cdot 5^k$（*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。

問 3.10.5 *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。

3.11 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。）

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。

問 3.11.1 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。

問 3.11.2 非負の整数 n を受け取り、$0 < x < y < z \le n$ 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。

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。

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

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

翻訳規則は次のとおりである。

$$[t] \Rightarrow unit\ t$$

$$[t \mid x \leftarrow u, P] \Rightarrow bind\ u\ (\backslash\ x \rightarrow [t \mid P])$$

$$[t \mid b, P] \Rightarrow \textbf{if}\ b\ \textbf{then}\ [t \mid P]\ \textbf{else}\ []$$

ただし、b は Bool 値の式、P は限定式のならびである。

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。

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

問 3.11.3 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 3.11.4 *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。
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。

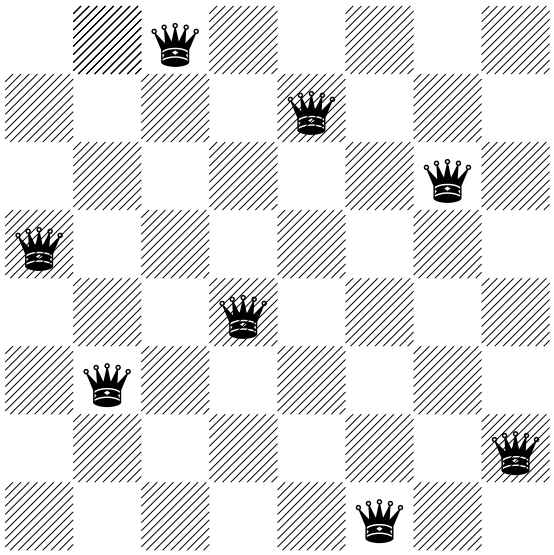
このようにして、素数を無限リストとして表現することで、さまざまな “境界条件” に対応することができる。*C* などで実装しようとすると、*1000* までの素数というのは配列を用いて簡単に求めることができるが、最初の *100* 個の素数を求めるのは急に難しくなる。

3.12 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。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。

```
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)
```

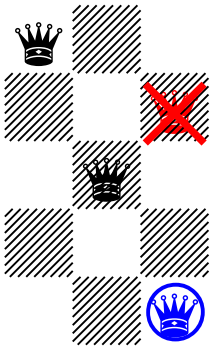
ここで、all は

```
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の関数である。[1..] は [1, 2, 3, ...] という無限リストである。

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

となる。



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  = [ append p [n] | 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 (backtrack)** と同じような効果を実現する。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。

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/
Haskell に関するもっとも主要な情報源である。

[2] 「Programming in Haskell」http://www.sampou.org/cgi-bin/haskell.cgi
日本語での Haskell の主要な情報源である。

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

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

[5] Simon Peyton Jones, David Lester 「Implementing Functional Languages」
Prentice Hall, 1992 年
http://research.microsoft.com/Users/simonpj/Papers/pj-lester-book/
Haskell の実装について知りたい人にお勧めする。

[6] John Hughes 「Why Functional Programming Matters」
1989 年, http://www.md.chalmers.se/~rjmh/Papers/whyfp.html
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。
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）
リストの内包表記を解説している。

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

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

[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
Haskell でなく、Miranda を使っているがとても有名な良書である。

[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