第1章 講義の概要

1.1 プログラミング言語の意味とは

プログラミング言語の仕様を定める時には、その言語の“構文” — つまり、どのような記号の列が正当なプログラムとして受け入れられるのか — とともに、その言語の“意味”を定める必要がある。

プログラミング言語にはさまざまな構成要素がある。例えば、ほとんどのプログラミング言語には条件分岐・繰り返しなどの制御構造（C 言語では **`if`** 〜 **`else`** 文, **`while`** 文, **`for`** 文, **`do`** 〜 **`while`** 文など）がある。 C++ や Java には、例外処理のための **`try`** 〜 **`catch`** 文もあるし、 Prolog にはユニフィケーション（単一化）・バックトラッキング（後戻り）などの仕組みがある。関数型言語 Haskell には遅延評価、オブジェクト指向言語にはクラス・インタフェースなど、やはり特有の仕組みが存在する。

これらの構成要素の“意味”については、現在のところ、日本語などの自然言語によって、「“条件式”が成り立つ間、“文”の実行を繰り返す」のように記述するのが普通である。しかし、自然言語による記述では、曖昧な点が多く、例えば次のような疑問が生じた時に厳密に議論することができない。

- コンパイラーの正当性 — コンパイラーは、本当に仕様書に定められた通りの動作をする目的プログラムを生成しているか？
- プログラムの同値性 — プログラムのある部分を、（おそらく効率の良い）別の形に書き直した時に、書き直しの前後でプログラムの意味は本当に同値か？

そのため、プログラムの“意味”を形式的に記述するために、さまざまな方法がこれまでに考えられて来た。

1.2 さまざまな意味論

プログラミング言語の意味論は大きく分けて、操作的意味論・公理的意味論・表示的意味論の3つに分類される。（ただし、この分類はある程度便宜的なもので、それぞれの境界がはっきりしているわけではない。）

操作的意味論 (operational semantics) は、仮想的な抽象機械を定義し、プログラムの意味をその抽象機械の内部状態の変化として記述しようという方法である。

公理的意味論 (axiomatic semantics) は、プログラムの意味を公理と推論規則により与えようとする方法である。ホーア論理 (Hoare logic) などがその代表的なものである。

表示的意味論 (denotational semantics) は、集合や関数のような数学的概念を用いて、プログラムの意味を記述しようとする方法である。

この講義では、主に表示的意味論に関連する事柄を紹介する。

プログラミング言語の”構文”の記述については偉大な先人の努力により、 BNF のような記法や `yacc` などの LALR パーサジェネレーターのようなツールが考え出され、仕様の記述や処理系の作成に欠かせないものとして、完全に定着している。

しかし、現在のところ、プログラミング言語の”意味”の記述については、 ”構文”に比べて標準的な記法やツールが定着するには至っていない。

それでも、プログラミング言語の意味論について、これまで蓄積された知見は、上に挙げたようなさまざまなプログラミング言語の構成要素についての理解を深めるために必要不可欠なものとなっている。本講義ではこのような”意味論”の簡単なところを”つまみ食い”して行く予定である。

1.3 ラムダ計算

ラムダ計算（λ-calculus）は表示的意味論を展開するのに利用される計算体系である。 ラムダ計算は”関数”に関するもっとも単純な記法・体系であり、また、もっとも単純なプログラミング言語と考えることもできる。

本講義では、このラムダ計算を用いて、より複雑なプログラミング言語の構成要素の”意味”を記述していくことにする。

本来は表示的意味論では、ラムダ計算に加えて、D_∞ や $P\omega$ と呼ばれる領域（domain）に関する理論を展開する。 しかし、本講義では領域に関する理論については触れない。興味のある人は各自で参考文献を調べて欲しい。

ラムダ計算はシンプルでエレガントな体系ではあるが、これですべてを記述しようとすると、記述量がたいへん多くなり手に負えなくなる。そこで、ラムダ計算に基づくプログラミング言語である関数型言語 Haskell を紹介する。 Haskell は、基本的にはラムダ計算にいくつかの構文上の糖衣（syntax sugar — 本質的ではないが便利な記法）を追加したものである。そのため、その気になれば、Haskell のプログラムは簡単にラムダ計算の式に書き直すことができる。

1.4 接続

プログラミング言語のさまざまな構成要素の意味を記述する上でさまざまな概念を導入するが、なかでも接続（継続ともいう、 continuation）の概念は制御構造・例外処理・後戻りなど、さまざまな事柄を説明するために、重要な役割を果たす。また、接続はコンパイラーの設計の際にも用いられる。接続の概念を理解することは、プログラミング言語の意味論を理解する上で不可欠である。

また、オブジェクト指向言語の動的束縛などの諸概念についても触れる予定である。

1.5 さまざまなプログラミング言語

本講義では Haskell のほかに Scheme, Prolog, JavaScript など、様々なパラダイム (paradigm) のプログラミング言語を紹介する。また、プログラミングのさまざまなイディオムを紹介する。

下の図に示すように、第 2 次世界大戦後にコンピューターが生まれて 10 年後位に最初の高級言語が生まれ、その後、ワークステーション・インターネット・機械学習といった新しいコンピューターの応用分野が広まるにつれ、新しい言語が生まれてきた。

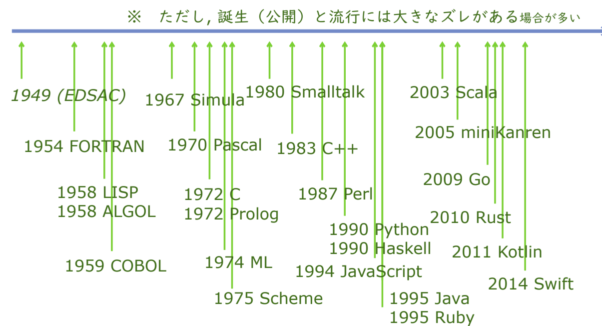


図: プログラミング言語の歴史

プログラミング言語の分類はいろいろな流儀があるが、主なものにパラダイムによる分類、型付けによる分類、実行方法による分類などがある。

- パラダイムによる分類
  命令型、オブジェクト指向、関数型、論理型
- 型付けによる分類
  静的型検査、動的型検査、他
- 実行方式による分類
  コンパイラー方式、インタプリター方式

言語によって得意分野が異なり、複数の言語を使い分ける必要がある。 また、複数の言語を知っていれば、ある言語 (A) では難しい処理を、別の言語 (B) では簡単に記述できる、という状況に出会うことがある。このときに、さまざまな都合により A 言語でプログラムを作成しなければならないとしても、 B 言語を知っていることによりプログラムの設計が容易になることがある。

例えば、オブジェクト指向言語と関数型言語は、どちらも命令型言語を拡張したものだが、方向性が異なる。オブジェクト指向言語は、GUI など用途に特化したデータ型を扱うような分野（例えばゲーム・シミュレーション）が得意である。一方、関数型言語は文字列・リストなど汎用のデータ型を扱うような分野（例えば言語処理系）が得意である（下図）。

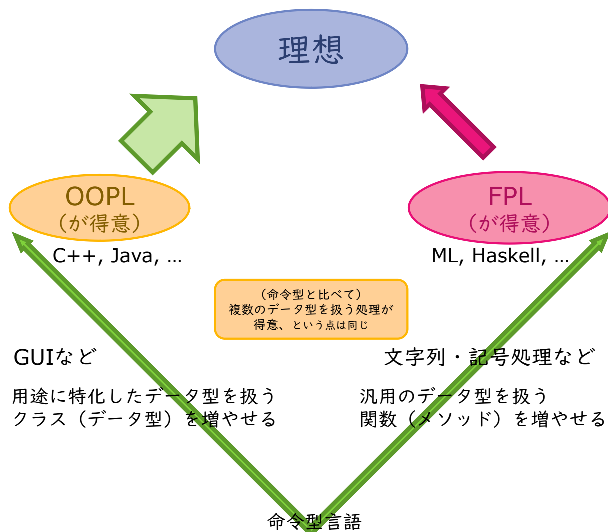


図: オブジェクト指向 vs. 関数型

一般的にスクリプト言語と呼ばれるプログラミング言語は動的型付けを採用していることが多い。動的型付けは実行前にチェックを行わないので、柔軟なプログラミングができて生産性が高い。一方でめったに使わない箇所を使ったときに、初めて不具合が見つかるということもありうる。そのため信頼性が必要とされる分野では使いづらい（下図）。

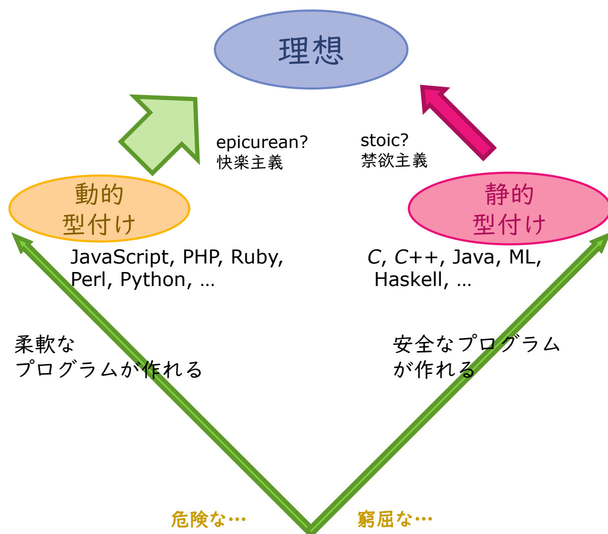


図: 動的型付け vs. 静的型付け

命令型やオブジェクト指向言語は“副作用”を多用する傾向がある。関数型言語は副作用を撤廃しているか、ひじょうにまれに使用する。副作用があると、プログラムの実行順序に実行結果が依存するので、プログラムの並列実行をしたときに予期しない結果となるときがある。副作用がないと並列実行しても結果が予期しやすいが、プログラムが書きにくいと言われている（下図）。

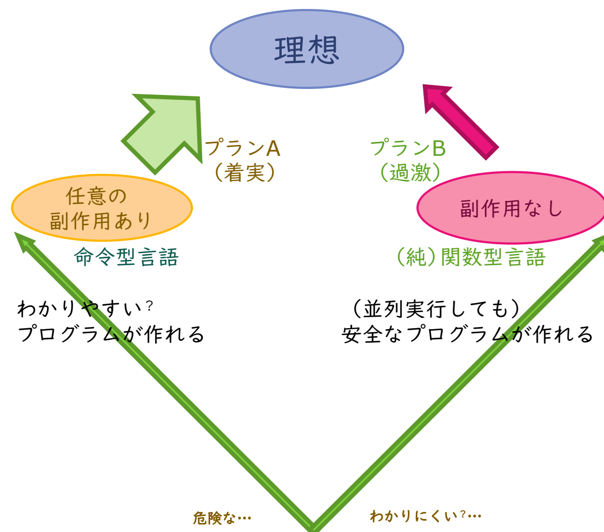


図: 副作用あり vs. 副作用なし

命令型言語やオブジェクト指向言語は共有の領域を書き換えていくことで計算が進むが、関数型言語は、新しいメモ用紙をどんどん使って、それを受け渡していくイメージである。

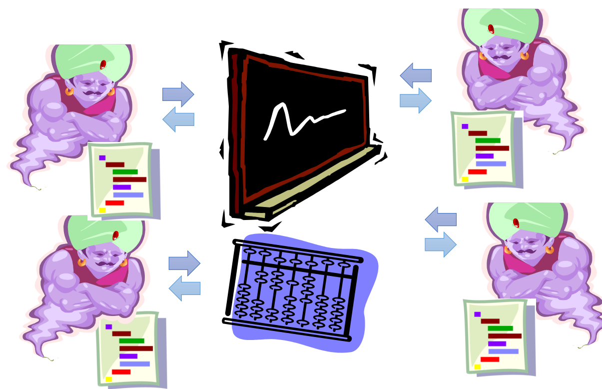


図: 命令型言語の実行のイメージ図

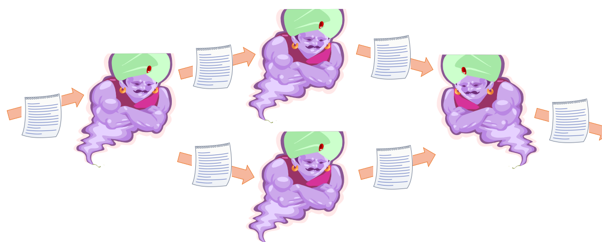


図: 関数型言語の実行のイメージ図

さらに、今後も当面は"万能の"言語が生まれるとは考えにくく、新しい言語が生まれ続けるだろう。あるいは、既存のプログラミング言語にも新しい概念が導入されていくだろう。しかし、新しいプログラミング言語といっても、まったく新しい概念ばかりからなるということはありえず、既存の言語から概念を借りてくることがほとんどである。そのため、複数の言語を知っていれば、新しい言語に対応するときにも有利である。

1.6 Haskell

Haskell は、代表的な関数型プログラミング言語であり、本質的に単純な構造を持っている。その代わりに型推論などを持つ高度な型システムを有している。副作用を撤廃し、代数的データ型 という OOPL のクラスとは異なる方向の拡張性を持つデータ型を提供している。多くの面で、メジャーな言語とは違う方向から理想に近づく言語ということができる。

この講義では、メジャーな言語と異なる特徴を多く持ち、かつ他の言語に大きな影響を与えている言語として Haskell を取り上げる。

Haskell はパズルや言語処理系など記号処理を得意とする。また、遅延評価は他の言語では難しいプログラムの組み立て方を可能にする。 Haskell を学習することは、既知の言語でのプログラミング能力の向上にも役立つと考えられ、また、新しい言語に Haskell の特徴が取り入れられることも多い。

1.7 さらに詳しく知りたい人のために...

この講義を構成するのに参考にした教科書を 2 つ挙げる。残念ながら、いずれも現在は入手困難である。

（中島 1982） 中島 玲二 「数理情報学入門 — スコット・プログラム理論」 朝倉書店, 1982, ISBN4-254-11413-3

（武市 1994） 武市 正人 「プログラミング言語 — 岩波講座ソフトウェア科学4」 岩波書店, 1994, ISBN4-00-010344-X

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. 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）。
3. （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。
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 ((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。)) に標準的なツール・ライブラリー・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、 `https://www.haskell.org/platform/` からインストールすることができる。

また Linux なども上記のホームページから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＿＿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

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**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。

```
1 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。

問 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。

```
1 fact n   | n == 0      = 1
2          | 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。

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 ______ 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、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 ＿＿＿＿＿ 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。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。

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

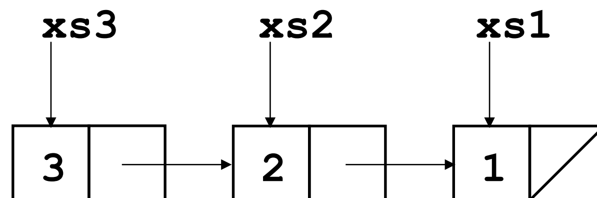
＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

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、

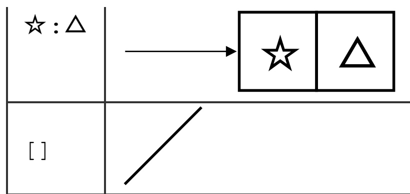
```
1 xs0 = []
2 xs1 = 1:xs0
3 xs2 = 2:xs1
4 xs3 = 3:xs2
```

次の図のようになる。

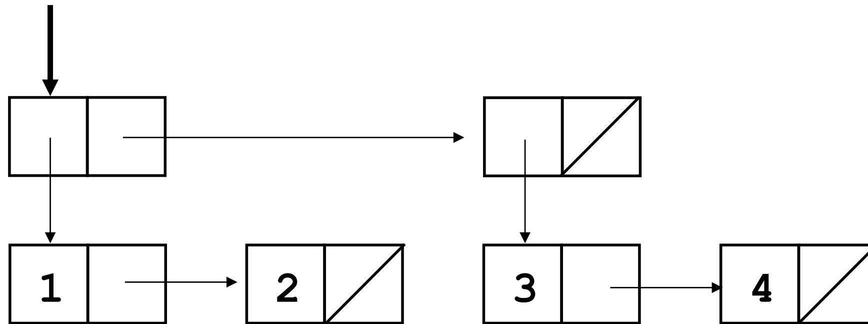


1つの 「:」を 2個の正方形の箱がつながった箱（コンスセル） **への矢印**で表す。箱の中身は、1, 2, 3 などの数、他の箱への矢印などである。（箱の中に他の箱がまるごと入ることはない。箱に何も入らないということもない。）空リストは斜線を引いて表す。

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。）

```
1 struct _list {
2   int head;
3   struct _list* tail;
4 };
5
6 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。

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 (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。

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。

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 `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。

```
1 fact 0 = 1
2 fact n = n * fact (n - 1)
3
4 -- Prelude の length とほぼ同じ
5 myLength []     = 0
6 myLength (x:xs) = 1 + myLength xs
```

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。）

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、

```
1 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。1, 3 では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 $(\ldots(a_n x + a_{n-1})x + \ldots + a_1)x + a_0$ 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. リストを昇ベ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

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。例えば、「`c = getchar(); putchar(c); putchar(c);`」と「`putchar(getchar()); putchar(getchar());`」は意味が異なる。つまり、数学の等号「$=$」と C 言語の「`=`」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 `reverse`:

```
-- (++) 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計算に時間がかかる、もう片方は正しいかどうか明らかでないが、高速である、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。

```
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 _      _      = []

```

```
 9 take :: Int -> [a] -> [a]
10 take 0 _       = []
11 take _ []      = []
12 take n (x:xs) = x : take (n - 1) xs
13
14 filter :: (a -> Bool) -> [a] -> [a]
15 filter p []      = []
16 filter p (x:xs) = if p x then x : filter p xs
17                          else filter p xs
18
19 iterate :: (a -> a) -> a -> [a]
20 iterate f x = x : iterate f (f x)
21
22 foldr :: (a -> b -> b) -> b -> [a] -> b
23 foldr f x []      = x
24 foldr f x (y:ys) = f y (foldr f x ys)
25
26 foldl :: (a -> b -> a) -> a -> [b] -> a
27 foldl f x []      = x
28 foldl f x (y:ys) = foldl (f x y) ys
29
30 concat :: [[a]] -> [a]
31 concat []         = []
32 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 $[x_1, x_2, \ldots] \Rightarrow [f\ x_1,\ f\ x_2, \ldots]$

その 2 引数版が zipWith である。

zipWith f $[x_1, x_2, \ldots]\ [y_1, y_2, \ldots] \Rightarrow [f\ x_1\ y_1,\ f\ x_2\ y_2, \ldots]$

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 $\circledast$ y) x $[y_1, y_2, y_3, \ldots\ y_n]$
$\Rightarrow y_1 \circledast (y_2 \circledast (y_3 \circledast (\ldots \circledast (y_n \circledast x)\ldots)))$
foldl (\ x y -> x $\oslash$ y) x $[y_1, y_2, y_3, \ldots\ y_n]$
$\Rightarrow (\ldots(((x \oslash y_1) \oslash y_2) \oslash y_3) \oslash \ldots) \oslash y_n$

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 $[[x_{1,1}, x_{1,2}, \ldots], [x_{2,1}, x_{2,2}, \ldots], [x_{3,1}, x_{3,2}, \ldots], \ldots]$
$\Rightarrow [x_{1,1}, x_{1,2}, \ldots, x_{2,1}, x_{2,2}, \ldots, x_{3,1}, x_{3,2}, \ldots]$

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`

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。

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

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

```
1 pow4 x   = let y = x * x in y * y
2 myHead ys = let (z:zs) = ys in z
3 -- 同等の関数 head は Prelude に定義済み
4 -- 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（「^」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。

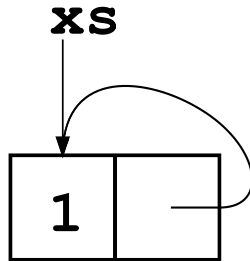
```
1 -- repeat は Prelude に定義済み
2 repeat  :: a -> [a]
```

```
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。

$\longrightarrow$

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 -- 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 （難）

$2^i \cdot 3^j \cdot 5^k$（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）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]]
2 [(1,5),(1,6),(1,7),(2,5),(2,6),(2,7),(3,5),(3,6),
  (3,7),↙
3 (4,5),(4,6),(4,7)]
4 Prelude> [x * x | x <- [1..10], odd x]
5 [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、$0 \le x \le y \le n$ 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 2.12.3 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 = (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。）

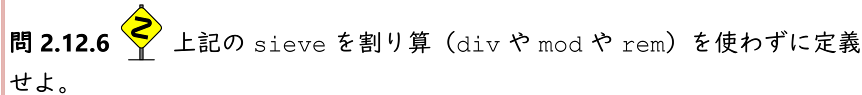
考え方:

“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。

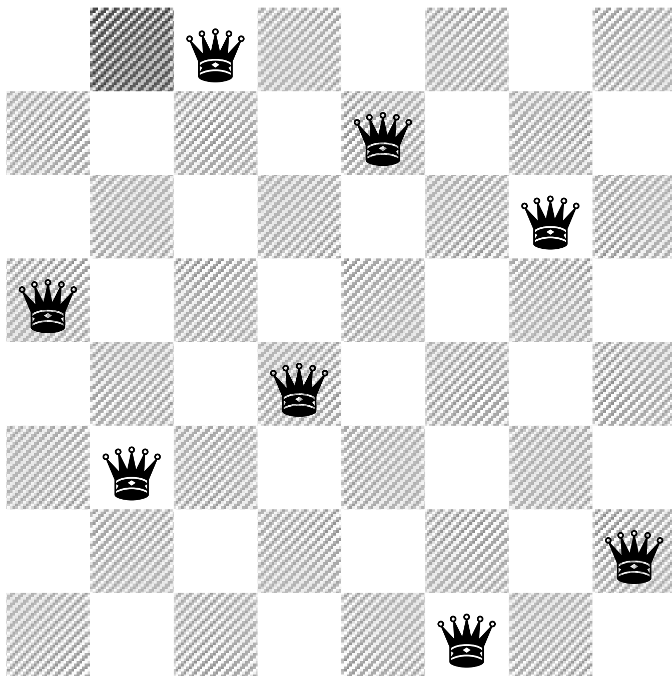
問 2.12.7 Haskell 以外の言語で、無限リストをシミュレートする方法を調査または考察せよ。またその方法を用いて、その言語で素数列を生成せよ。

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。

```
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
6                     || (i + j == m + n)
7                     || (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
```

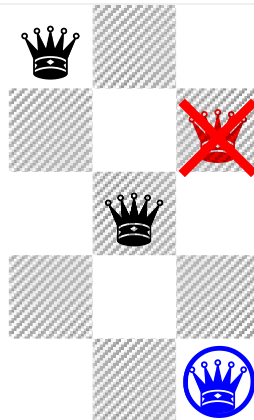
と定義された標準ライブラリーの関数である。また、[1..] は [1,2,3,…] という無限リストの略記法である。つまり、 check は横または斜めで同じ線上に乗っていないことをチェックしている。

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?

```
1 > safe [1,3] 5
2 True
3 > safe [1,3] 2
4 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。

```
1 queens 0   = [[]]
2 queens m   = [ p ++ [n]
3              | p <- queens (m - 1), n <- [1..8],
4                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

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

```
1 > head (queens 8)
2 [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、

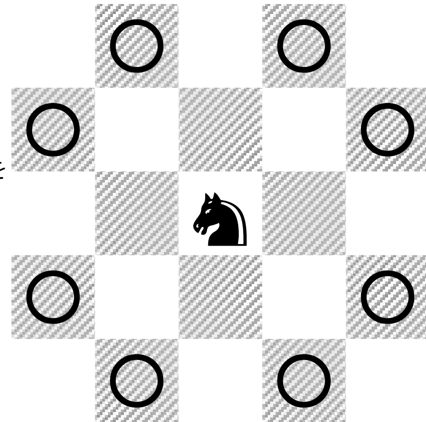
```
1 > queens 8
2 [[1,5,8,6,3,7,2,4],[1,6,8,3,7,4,2,5],
3 （略）
4 ,[8,3,1,6,2,5,7,4],[8,4,1,3,6,2,7,5]]
```

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。

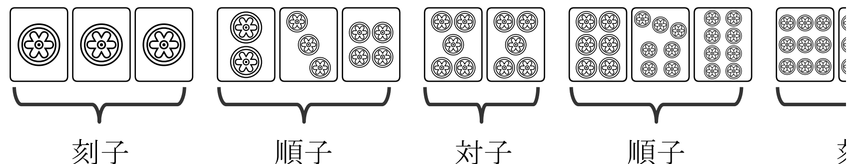


問 2.13.5 （アガリ判定）

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。

```
1 Prelude> shuntsu [1,3,3,4,5,7,8,9]
2 [[3,4,5],[7,8,9]]
3 Prelude> toitsu [1,3,3,4,5,7,7,9]
4 [[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（**和田 2005**）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（**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” http://www.haskell.org/

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

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

（Jones et al. 1999） Mark P Jones, Alastair Reid他, “The Hugs 98 User Manual” http://cvs.haskell.org/Hugs/pages/hugsman/

（Peyton Jones & Lester 1992） Simon Peyton Jones, and David Lester, “Implementing Functional Languages” http://research.microsoft.com/Users/simonpj/Papers/pj-lester-book/, Prentice Hall, 1992 年

（Hughes 1989） John Hughes, “Why Functional Programming Matters” 1989 年,
http://www.md.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」
http://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
http://research.microsoft.com/users/simonpj/Papers/slpj-book-1987/ のなかの第 7 章）

（和田 2005） 和田 英一 他 「Haskell プログラミング」
http://www.ipsj.or.jp/07editj/promenade/

（Peyton Jones 2003） Simon Peyton Jones, “Wearing the hair shirt — A retrospective on Haskell”
http://research.microsoft.com/~simonpj/papers/haskell%2Dret
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

第A章 Haskell のレイアウトルール

これまで Haskell のレイアウトルールについては明確に述べていなかった。Haskell は字下げ（インデント）の仕方により、字句解析・構文解析が影響を 受ける言語である。（ブレースやセミコロンを明示的に使って影響を受けない ようにすることも可能である。）

例えば、

```
1 f x = let a = 1; b = 2
2           g y = exp2
3         in exp1
```

という式は、

```
1 f x = let {a = 1; b = 2
2           ;g y = exp2
3       } in exp1
```

と解釈される。レイアウトルールはこのようなブレースやセミコロンがどのような場合に挿入されるかを定める。

以下の字下げ部分は Haskell の仕様書である“Haskell 2010 Language Report”の§2.7 からの抜粋である。（なお、同書の § 10.3 に、より詳細な仕様が掲載されている。）

> Haskell permits the omission of the braces and semicolons used in several grammar productions, by using layout to convey the same information.
> Haskellは、 同等の情報を持つレイアウトを使うことによって、いくつかの文法規則で使われているブレースとセミコロンを省略することを許している。
>
> （中略）
>
> The effect of layout on the meaning of a Haskell program can be completely specified by adding braces and semicolons in places determined by the layout. The meaning of this augmented program is now layout insensitive.
> レイアウトの Haskell プログラムに対する効果は、レイアウトによって決定される場所にブレースとセミコロンを追加することにより完全に記述することができる。すると、その追加の結果のプログラムの意味はレイアウトに依存しなくなる。
>
> Informally stated, the braces and semicolons are inserted as follows.
> ブレースとセミコロンはおおむね次のように挿入される。
>
> The layout (or “off-side”) rule takes effect whenever the open brace is omitted after the keyword **`where`**, **`let`**, **`do`**, or **`of`**.

> レイアウト（あるいは"オフサイド"）ルールは **where**, **let**, **do**, **of** というキーワードの後で開ブレースが省略されたときに、有効になる。

```
let
  f x y =
    case x of
       0 -> foo x 2
       1 -> bar 1 x 999999
              3 4 5
       _ -> baz 6 x
   +
    case y of 2 -> qux 9 y 1
              _ -> quux y  in f 0 1
```

> When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted (the whitespace preceding the lexeme may include comments).
> このときは、（新しい行にあるかどうかに係わらず）次の字句の字下げ数が記憶され、省略された開ブレースが挿入される（タブ文字は次の **8** の倍数文字目までの空白文字と解釈される。 Windows のエディタはタブ文字が次の **4** の倍数文字目までの空白文字と解釈されるものが多いので注意すること）。（字句の前の空白はコメントの場合もある（漢字・かななどの全角文字も Haskell の処理系にとっては英数字と同じ一文字なので、コメントがレイアウトに関係する場合は注意すること。）。）

```
-- 開ブレース挿入後
let
 {f x y =
    case x of
      {0 -> foo x 2
       1 -> bar 1 x 999999
              3 4 5
       _ -> baz 6 x
   +
    case y of{2 -> qux 9 y 1
              _ -> quux y  in f 0 1
```

> For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted); if it is indented the same amount, then a new item begins (a semicolon is inserted); and if it is indented less, then the layout list ends (a close brace is inserted).
> それに続く各行について、もし、それが空白のみを含むか、より多くの字下げがされているならば、直前の項目が継続される（つまりなにも挿入されない）。もし、同じだけ字下げされているなら、新しい項目が始まるとみなす（セミコロンが挿入される）。もし字下げ数が少ないならばレイアウトリストは終わりとみなされる（閉ブレースが挿入される）。

```
-- セミコロン挿入後
let
 {f x y =
```

```
    case x of
      {0 -> foo x 2
      ;1 -> bar 1 x 999999
              3 4 5 -- この行は挿入なし
      ;_ -> baz 6 x
    }+
    case y of{2 -> qux 9 y 1
             ;_ -> quux y  in f 0 1
```

> A close brace is also inserted whenever the syntactic category containing the layout list ends; that is, if an illegal lexeme is encountered at a point where a close brace would be legal, a close brace is inserted.
> 閉ブレースはレイアウトリストを含む構文カテゴリが終了するときにも挿入される。つまり、閉ブレースが現れても良いときに不正な字句が現れたとき、閉ブレースが挿入される。

```
-- 閉ブレース挿入後
let
 {f x y =
    case x of
      {0 -> foo x 2
      ;1 -> bar 1 x 999999
              3 4 5 -- この行は挿入なし
      ;_ -> baz 6 x
    }+
    case y of{2 -> qux 9 y 1
             ;_ -> quux y}}in f 0 1
```

> The layout rule matches only those open braces that it has inserted; an explicit open brace must be matched by an explicit close brace.
> レイアウト規則は開ブレースが挿入されたときにのみ発動される。明示的な開ブレースがあったときは、明示的な閉ブレースで終らなければならない。
>
> Within these explicit open braces, *no* layout processing is performed for constructs outside the braces, even if a line is indented to the left of an earlier implicit open brace.
> このような明示的な開ブレースのなかでは、そのブレースの外側の構成要素に対しては、たとえある行が以前にあった暗黙の開ブレースよりも左から始まったとしても、レイアウトルールは適用されない。

```
let{foo = let {
  x = 0; y = 1;
  z = 2 }
     in x * y
   ;bar = 99}in foo * bar
```

第3章 代数的データ型と型クラス

3.1 代数的データ型の定義

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

データ型の宣言の一般的な形式は次のとおりである。

$$
\begin{array}{ll}
\textbf{data} & \text{型構成子名}\ \ \text{型引数}_1\ \ \text{型引数}_2\ \ldots\ \text{型引数}_k \\
\quad = & \text{構成子名}_1\ \ \text{型}_{1,1}\ \ldots\ \text{型}_{1,n_1} \\
\quad \mid & \text{構成子名}_2\ \ \text{型}_{2,1}\ \ldots\ \text{型}_{2,n_2} \\
\quad \mid & \ldots \\
\quad \mid & \text{構成子名}_m\ \ \text{型}_{m,1}\ \ldots\ \text{型}_{m,n_m}
\end{array}
$$

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

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

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

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

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

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

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

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

```
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 はフィールドを持たず、それだけで二分木を構成する。 Branch は 3 つのフィールドを持つ。1 番目と 3 番目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。

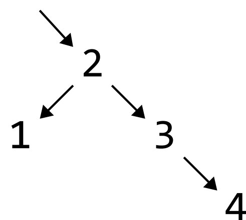
```
Branch :: ______________________________
```

ここで、a は____________であり、ここには Integer や String などの具体的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、型パラメーター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 の型を持つ。

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

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

```
 6 isEmpty _      = False
 7
 8 -- 要素数、例えば treeSize tree4 は 4
 9 treeSize :: Tree a -> Int
10 treeSize Empty            = 0
11 treeSize (Branch l a r) = treeSize l + 1 + treeSize r
```

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

Q 3.1.2 第 1 引数が第 2 引数に対して、Janken 型の勝ち・負け・引き分けのいずれであるかを判定する関数 judgeJanken :: Janken -> Janken -> Ordering を定義せよ。

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

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

LT が負け、GT が勝ち、EQ が引き分け、を表すものとする

問 3.1.3 Tree 型に対して、次のような関数を定義せよ。

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

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

問 3.1.4 一般の n 分木（任意の個数の部分木を持つことができる木、rose tree ともいう）を表すデータ型 RoseTree を定義せよ。また、n 分木の要素数を求める関数 roseSize :: RoseTree a -> Integer を定義せよ。

（発展） フィールドラベル

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

```
1 data C = F { field1, field2 :: Int, field3 :: Bool }
2     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。

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

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

```
1 v1 = F { field1 = 5, field2 = 12, field3 = False }
2
3 foo :: C -> Int
4 foo (F { field1 = x, field2 = y }) = x * x + y * y
```

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

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

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

3.2 型クラスとは

型クラスの説明に入る前にまずいくつかの用語を紹介する。

ポリモルフィズム（多相, polymorphism）とは関数（メソッド）などが________ ________を許すことをいう。 さらに、関数などがいろいろな型の引数を許し、しかも ________________ ことを、 ________ （ad-hoc — その場限りの、という意味）多相という。オブジェクト指向言語の動的束縛（dynamic binding）はアドホック多相の一種である。

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

動的束縛と混同しがちな概念として ____________ （多重定義, overloading）がある。多重定義は一つの名前が複数の意味を持つことである。例えば C の「+」オペレーターは `int`（整数）型にも `double`（倍精度浮動小数点数）型にも適用できる。しかし最終的には、適用される型によって全く異なる機械語に翻訳される。動的束縛と異なる点は、多重定義はコンパイル（型チェック）時に解決されてしまう、という点である（つまり静的な束縛である）。実行時にオペランドの型に応じて処理を振り分けるようなことは行なわない。

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

Haskell のように型推論と多相型 (アドホックな多相に対して、型によって処理が変わらない多相のことをパラメトリック（parametric）な多相と言う。) を許す言語では、コンパイル時に得られる型情報では演算子の実装を決定することはできない。例えば、次のような関数:

```
twice x = x + x
```

を定義した時、x の型が整数か浮動小数点数か決定できないので、「+」の実装も決定できない。つまり、C や Java（の演算子）で使われているような多重定義は、Haskell では使えない。では、twice はどのように実装され、どのような型を持つべきか？

一方、Haskell では一つの代数的データ型の異なるコンストラクターのなかでは、パターンマッチングによりアドホック多相を実現している。例えば、Tree 型の Branch と Empty では関数 isEmpty や size の実装が異なる。上記の twice のような関数の定義を可能にするためには複数の型にまたがるアドホック多相を実現する必要がある。そこで、Haskell では＿＿＿＿＿という仕組みが導入されている。

3.3 Haskell の型クラス

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

Haskell でも C と同じように「==」（等号）オペレーターは Integer（整数）型にも Double（倍精度浮動小数点数）型にも、さらに他のいくつかの型にも適用できる。一方、関数型は比較できない。例えば (\ x -> x + x) == (\ x -> 2 * x) はエラーである。Haskell は多相を許す言語であるので、例えば、

```
1 member x []      = False
2 member x (y:ys)  = x == y || member x ys
3
4 subset xs ys     = all (\ x -> member x ys) xs
```

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

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

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

```
member :: ____________________

subset :: ______________________
```

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

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

3.4 クラス宣言とインスタンス宣言

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

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

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

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

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

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

```
1 instance Eq Direction where
2   North == North = True
3   South == South = True
4   West  == West  = True
5   East  == East  = True
6   _     == _     = False
```

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

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

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

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

一般的には、

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

のようにキーワード instance のあとに制約とクラス名、型を書き、where というキーワードのあとに関数の定義を並べる。これは、制約$_0$のもとで、型$_0$は、クラス$_0$のインスタンスであり、関数$_1$、関数$_2$の実装はそれぞれ式$_1$、式$_2$である、という意味である。制約は、一般に (クラス$_1$ 型変数$_1$, クラス$_2$ 型変数$_2$, …) という形式である。

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

Q 3.4.1 組込みの Bool 型と等価なデータ型:

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

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

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

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

```
1 instance Sizable [a] where
2     size xs = length xs
3
4 instance Sizable (Tree a) where
5     size t = treeSize t
```

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

3.5 その他の標準的な型クラス

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

```
 1 class Eq a => Ord a where     -- Ord は order （順序）の
   こと
 2    (<), (<=), (>=), (>) :: a -> a -> Bool
 3    max, min             :: a -> a -> a
 4    -- …
 5
 6 class Show a where
 7    show                 :: a -> String
 8    -- …
 9
10 class (Eq a, Show a) => Num a where
11    (+), (-), (*)        :: a -> a -> a
12    fromInteger          :: Integer -> a
13    -- …
14
15 class Num a => Fractional a where
16    (/)                  :: a -> a -> a
17    -- …
```

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

クラス宣言の「=>」の左側にあるクラスは、スーパークラスと呼ばれる。

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

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

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

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

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

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

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

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

Q 3.5.1 組込みのリスト型と等価なデータ型

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

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

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

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

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

いくつかのケースをテストせよ。

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

3.6 オブジェクト指向との関係

オブジェクト指向言語で使用される概念との関連について触れる。

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

Q 3.6.1 左の語句の意味と対応する右の説明を線で結べ。

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

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

3.7 オブジェクト指向のクラスと代数的データ型

Haskell や ML といった関数型言語では、複数の構成子 (constructor)からなる代数的データ型 (algebraic data type) を定義できる。代数的データ型を構成する各構成子を、オブジェクト指向言語のクラスに相当すると見なせば、関数型言語もアドホック多相や動的束縛に相当する機構を持っている。関数は、さまざ

まな構成子の引数を受け取り、また呼び出されるコードがパターンマッチングにより、オブジェクトの実行時の構成子で定まる。例えば、Tree 型の isEmpty という関数は Branch と Empty で実行されるコードが変わる。

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

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

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

3.8 型クラスとサブタイピング

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

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

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

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

もう少し一般的な例として、次のような型クラスを考える。

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

これに対して、次のような型と変換関数を考えることができる。

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

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

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

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

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

一般的に型クラス

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

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

```
data T = T σ1 σ2 σ3
```

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

3.9 （参考） 型クラスの問題点

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

わかりにくいエラーメッセージ

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

曖昧さ (ambiguity)

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

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

class Read a where
  read :: String -> a
```

この定義の元で、次のような関数を定義する。

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

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

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

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

3.10 （参考） Dictionary-Passing Style 変換

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

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

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

```
type Eq' a = (a -> a -> Bool, a -> a -> Bool)

eq' :: Eq' a -> (a -> a -> Bool)
eq' = \ (e, _) -> e          -- (==) に対応する

```

```
6 ne' :: Eq' a -> (a -> a -> Bool)
7 ne' = \ (_, n) -> n          -- (/=) に対応する
```

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

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

```
 1 eqDirectionDic :: Eq' Direction
 2 eqDirectionDic = (eqDirection,
 3                   \ a b -> not (a `eqDirection` b))
 4   where North `eqDirection` North = True
 5         South `eqDirection` South = True
 6         West  `eqDirection` West  = True
 7         East  `eqDirection` East  = True
 8         _     `eqDirection` _     = False
 9
10 eqTreeDic :: Eq' a -> Eq' (Tree a)
11 eqTreeDic (eqA, _) = (eqTree, \ a b -> not (eqTree a
   b))
12   where
13     Empty `eqTree` Empty  = True
14     Branch l1 n1 r1 `eqTree` Branch l2 n2 r2
15       = l1 `eqTree` l2 && n1 `eqA` n2 && r1 `eqTree`
   r2
16     _      `eqTree` _      = False
```

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

```
1 member' :: Eq' a -> a -> [a] -> Bool
2 member' d x []     = False
3 member' d x (y:ys) = eq' d x y || member' d x ys
4
5 subset' :: Eq' a -> [a] -> [a] -> Bool
6 subset' d xs ys = all (\ x -> member' d x ys) xs
```

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

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

このような書き換え（Dictionary-Passing Style 変換）は Haskell ではコンパイル中の型推論時に自動的に行なわれる。つまり、アドホック多相の実行時のコストは、辞書オブジェクトの中から関数を取り出し、それを起動するだけにな

る。要するに、アドホック多相は高階関数で解釈できる (ただし高階関数になるので、 最適化が難しくなってしまう可能性はある。)。動的に（つまり実行時に）メソッドを探しているように見えて、実際には静的に（コンパイル時に）ほとんどの必要な処理が済んでいる。

問 3.10.1 次のように定義されている関数 `lookup`:

```
1 data Maybe a = Just a | Nothing
2 
3 lookup :: Eq a => a -> [(a, b)] -> Maybe b
4 lookup x ((n,v):rest)
5    = if n == x then Just v else lookup x rest
6 lookup x [] = Nothing
```

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

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

を示せ。例えば、`lookup 1 [(1,2),(4,7)]` と `lookup’ eqIntDic 1 [(1,2),(4,7)]` の値が、あるいは `lookup 1.1 [(1.4,0.1),(4.2,6.9)]` と `lookup’ eqDoubleDic 1.1 [(1.4,0.1),(4.2,6.9)]` の値が同じ値になる。（`eqIntDic`, `eqDoubleDic` は各自で適宜定義する。）

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

3.11 （参考）他のオブジェクト指向言語について

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

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

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

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

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

3.12 まとめ

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

3.13 さらに詳しく知りたい人のために…

文献（Wadler & Blott 1988）は、型クラスのアイデアを最初に紹介した論文である。文献（Hall et al. 1996）は、現在の Haskell の型クラスを詳しく説明している。文献（Jones 2000）は、Haskell の（型クラスに関する部分を含む）型推論を、具体的に Haskell のプログラムを用いて説明している。文献（Odersky 1991）は、オブジェクト指向言語のサブタイプと同様の効果を、関数型言語の枠組みで得るための手法を提案している。文献（Heeren & Hage 2005）は、型クラスのエラーメッセージの問題点と改良法について詳しく述べている。

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

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

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

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

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

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

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

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