Haskell 入門*

橋本健二

初版：2020 年 4 月 22 日
改訂：2021 年 4 月 22 日

1 概要

この資料は，酒井研 B4 トレーニングゼミである『Haskell を利用した Knuth-Bendix 完備化手続きの実現』の資料として作成した．酒井研における Haskell の処理系の利用方法と，ゼミで必要となる基本概念について簡単に説明する．

2 起動・終了・中断の方法

本資料では Haskell を，Unix 系 OS 環境 (FreeBSD，Linux 等) で利用するための簡単な解説を行う．各自の計算機に Unix 系 OS がインストールされてない場合は，酒井研環境での共用計算機である `atlas` 等を利用されたし．なお，Windows や Mac 環境で各種作業を行いたい方は各自で戦われたし．

さて，先ずは起動してみよう．コマンドライン上で ghci とタイプすることで Haskell のインタプリタ ghci は起動できる．

```
$ ghci
GHCi, version 8.0.2: http://www.haskell.org/ghc/  :? for help
Prelude>
```

1 行目の先頭の `$` はシェルのプロンプトである．3 行目の `Prelude>` は ghci のプロンプトであり，ここで何か入力をうけつける状態であることを表している．ghci での作業を終了してシェルに戻りたいときには，プロンプトがでているときに `:quit` を入力するか `C-d` (control-d) を入力すればよい．止まらなくなってしまったと思われる (あるいは単に長い) 処理を中断したいときには `C-c` (control-c) を入力する．

また，Emacs エディタで Haskell プログラムを実行するために，`haskll-mode` というプラグインがある．Emacs のバッファ内で ghci を起動して Haskell プログラムを実行することができる．Emacs を利用する場合 `haskell-mode` を導入することで，入力した式を編集したり，編集して入れ直したりするのが非常に楽になる．なお，atlas 上で Emacs の Haskell モードを利用する場合には，`~/.emacs` に以下を追加する必要がある．

```
;; for haskell
(add-to-list 'load-path "/usr/local/share/emacs/site-lisp/haskell-mode/")
(require 'haskell-mode-autoloads)
(setq haskell-program-name "/usr/local/bin/ghci")
(add-hook 'haskell-mode-hook 'inf-haskell-mode) ;; enable C-c C-l for loading
```

*この資料は，KB ゼミのために用意されていた SML/NJ 入門（草刈先生著，初版 2003 年 4 月 25 日）という資料を Haskell 用に改訂したものである．

Emacs を起動して `M-x run-haskell` とタイプしてみよう．こうすると `*haskell*` という名前のバッファが作られその中で ghci が起動する．`hs` の拡張子がついた`*.hs` なファイルを読みこんだり新規に作成したときは，自動的にそのファイルのバッファは Haskell モードになる．現在のバッファのファイルを ghci にロードする場合には `C-c C-l` とタイプする．また，Emacs で実行している Haskell の計算を中断する場合には `C-c` を 2 回タイプする．

3 処理系との対話

ghci に慣れるために処理系と対話してみよう．式 (expression) を入力すると処理系はその式を評価した (evaluating) 結果得られた値 (value) を出力してくれる．先ずは簡単な算術式を試してみよう．

```
Prelude> 3 + 4 * 5
23
```

この例において ghci は `3 + 4 * 5` なる算術式を評価して `23` という値を求めている．

上記の実行例のように，ユーザーが式を入力し，その式を評価して再び入力受付状態に戻る，というような作業の繰り返しを対話的（interactive）な作業と呼ぶ．他の言語と同様，Haskell でも値に名前を与えて後で利用できるような仕組みがある．

```
Prelude> p = 3.1415926535
Prelude> p
3.1415926535
Prelude> p * 3.0 * 3.0
28.2743338815
```

最初のステップで値 `3.1415926535` を名前 `p` を持つ変数に覚えさせ，以降ではこの変数を利用できるようになっている．もちろん，関数定義も可能である．実際に階乗関数とべき乗関数を定義してみる．ここで，`if _ then _ else _` は条件演算子（conditional operator）であり，`{- … -}` はコメント文である．

```
Prelude> {- 階乗関数 fact1 の定義 -}
Prelude> fact1 n = if n==0 then 1 else n * (fact1 (n-1))
Prelude> fact1 3
6
Prelude> {- べき乗関数 power の定義; ただし，m >= 0 を仮定 -}
Prelude> power n m = if m==0 then 1 else n * (power n (m-1))
Prelude> power 2 3
8
```

また次のようなパターン（pattern）を利用した関数定義も可能である．なお，複数行にわたる定義を入力する場合には`:{` と `:}` で囲むことで定義できる．なお，ファイルに関数定義を書く場合は`:{` と `:}` は不要である．

```
Prelude> :{
Prelude| fact2 0 = 1
Prelude| fact2 n = n * fact2 (n-1)
Prelude| :}
Prelude> fact2 3
6
Prelude> :{
Prelude| power x 0 = 1
Prelude| power x n = x * (power x (n-1))
Prelude| :}
Prelude> power 2 3
8
```

プログラムを一々対話的に打ち込むのは非常に困難である (って言うか無理)．通常はプログラムをファイルに書き込み，そのファイルを処理系で読み込む．Emacs の Haskell モードを設定してるならばコマンド (C-c C-l) で処理系にファイルの内容を送ることができる．また，ghci 上でファイルを読み込むには :load なるコマンドを用いる．例えば，sample.hs というファイルにプログラムを書き，それを読み込むには次のようにすれば良い．

```
Prelude> :load sample.hs
```

以降では，プログラムはファイルに書き，それをロードして実行する前提で説明を記述する．また，ghci のプロンプト Prelude> を > と省略する．

実行形式を作成する（コンパイルする）方法もある．コマンドライン上で ghc を利用してコンパイルを行う．ただし，コンパイル対象のファイルには main 関数の記述が必要である．例えば，以下のような cat.hs を用意する．

```
main :: IO ()
main = do {ln <- getContents; putStr ln}
```

cat.hs をコンパイルして実行する．

```
$ ghc cat
[1 of 1] Compiling Main ( cat.hs, cat.o )
Linking cat ...
$ ./cat
fghj
fghj
ssss (最後に^D とタイプ)
ssss
```

4 基本型

Haskell では基本型として，ブール型，整数型，実数型，文字型，文字列型，が用意されている．これらは順に Bool，Int，Float ，Char，String で表現される．本節ではこれらの基本型について簡単に解説する．

4.1 ブール型

ブール型の値は True と False のみである．通常のブール値上の *or* と *and* に相当する関数はそれぞれ||および&&となっている．また，通常のブール値上の *not* に相当する関数として関数 not が用意されている．

```
> True && False
False
> True || False
True
> not True
False
> if True then 0 else 1
0
> if False then 0 else 1
1
```

問 1 条件演算子（*if* 文），変数，値 True,False のみを用いて，通常のブール値上の *not*，*and*，*or* に相当する関数 my_not, my_and, my_or をそれぞれ一行で定義せよ．

4.2 整数・実数型

算術式は他の多くの言語と同様に表現される．ただし，数 3 は整数型 Int においては 3，実数型 Float においては 3.0 と表現しなければならない．算術演算子としては+,-,*, /（実数除算），div（整数除算）mod（整数除算の剰余）などが用意されている．なお，div や mod は 2 引数関数として与えられているため div 5 2 のように書いて利用するが，バッククォートを利用して 5 `div` 2　のように中値形式で書くこともできる．整数・実数型の値の比較演算子として==, <, >, <=, >=, /= が用意されている．これらはそれぞれ通常の意味での $=, <, >, \leq, \geq, \neq$ に対応している．なお，比較演算子==と/=は文字列型などの比較にも利用できる．実際に例で見てみよう．

```
> 9 - 2 * 3
3
> 9 `div` (8 `mod` 3) * 2
8
```

問 2 次の式を入力すると処理系は如何なる応答を示すか予測し，その予測の正当性を処理系で確認せよ．

(i) 1 + 2 * 3

(ii) 2.0 + 3

(iii) 7 / 2

(iv) 7 - 5 `div` 3

(v) if 1 >= 2 then 3 else 4

4.3 文字・文字列型

文字 A は文字型 Char において'A' のように，文字列 ABC は文字列型 String において"ABC"のように表現される．また，文字列に限らずリスト一般の連接用に関数 ++ が用意されている．実際の使用例を見てみよう．

```
> 'A'
'A'
> :type it
it :: Char
> "ABC"
"ABC"
> :type it
it :: [Char]
> "ABC" ++ "DEF"
"ABCDEF"
```

文字列の出力は関数 print や putStr を用いる．

```
> print "Hello World!"
"Hellow World!"
> print 3.14
3.14
> putStr "\nHello World!\n"

Hello World!!
> putStr "Hello World!"
Hello World!!>
```

ここで，\n は改行文字である．デバッグ用の出力には Debug.Trace モジュールの trace が便利である．

```
import Debug.Trace

fact3 0 = 1
fact3 n = trace ("n is " ++ show n) (n * fact3 (n-1))
```

以上の関数の実行例は以下の通りである．

```
> fact3 3
n is 3
n is 2
n is 1
6
```

trace は第 2 引数をそのまま返り値としますが，その際に第 1 引数の String が副作用的に出力される．

5 型の構成

Haskell でも他のほとんどの言語と同様に，単純な型から新しい型を定義する演算子や記法を使って，より複雑な型を構成することができる．この構成に置いて Haskell では型変数 (type variable) を利用することができる．

5.1 タプル型

タプルとは任意の型の 2 個以上の式のリストを取り，それらをコンマで区切って丸括弧で囲む事によって形成される．

```
> (7, 7.0, '7', "seven") :: (Int, Float, Char, String)
(7, 7.0, '7', "seven")
```

上記の例では入力の型は (Int, Float, Char, String) である．このような組を表現する型をタプル型と呼ぶ．

5.2 リスト型

リスト型とは，ある唯一の型を要素として持つ型であり要素の並びをコンマで分けて角括弧で囲む．

```
> :type []
[] :: [t]
> :type [1,2,3]
[1,2,3] :: Num t => [t]
> :type ['a', '2']
['a', '2'] :: [Char]
> :type ["One", "Two"]
["One", "Two"] :: [[Char]]
> :type [[1], [1,2], []]
[[1], [1,2], []] :: Num t => [[t]]
```

リスト [1,2,3] に対し，その最初の要素 1 を頭部 (head) と呼び，それ以外の全ての要素のリストである [2,3] を尾部 (tail) と呼ぶ．ここで頭部と尾部の型が異なる事に注意する．Haskell ではリストを構成するコンス演算子:やリストを連結 (concatenation) する演算子++が用意されている．

```
> 1 : [2, 3]
[1,2,3]
> 1 : 2 : [3]
[1,2,3]
> 1 : 2 : 3 : []
[1,2,3]
> [1,2] ++ [3]
[1,2,3]
```

上記のようにコロンで表現されるコンス演算子は，要素を一つ (頭部) と頭部と同じ型の要素からなるリストを受取り，最初の要素が頭部で残りが尾部であるような 1 個のリストを生じる．ちなみに，厳密に言うと `[1,2,3]` や `1:[2,3]` などは `1:(2:(3:[]))` の略記である．以下に，リスト反転関数 `rev` を 3 通りの方法で定義する，参考にされたし．なお，関数 `null` は与えられたリストが空リストかどうかを判定する述語 (返し値の型がブール型の関数) であり，関数 `head` はリストの頭部を，関数 `tail` はリストの尾部を取り出す組み込みの関数である．

```
rev1 [] = []
rev1 (x:xs) = (rev1 xs) ++ [x]

rev2 xs = if null xs then [] else (rev2 (tail xs)) ++ [head xs]

rev3_sub [] ys = ys
rev3_sub (x:xs) ys = rev3_sub xs (x:ys)
rev3 xs = rev3_sub xs []
```

ここで，リスト型の変数に与えた名前に注意しよう．リストの要素を表す変数には名前 `x` や `y` を与え，リストを表す変数には `xs` や `ys` を与えている．さらに，この慣例を拡張してリストの要素がリストであるようなリストを表すには `xss` や `yss` を用いる．この慣例は広く関数型言語によるプログラミングで用いられている．本質的ではないかも知れないが，プログラマーの不必要な混乱を避けると言う点で有用である．

問 3 以下の関数を定義せよ．

(i) リストを入力としその長さを返す関数 `mylength`，すなわち，`mylength` $[a_1, a_2, \ldots, a_n] = n$.

(ii) 整数型のリストを入力としその総和を返す関数 `mysum`，すなわち，`mysum` $[a_1, a_2, \ldots, a_n] = \Sigma_{i=1}^{n} a_n$.

(iii) メンバー関数 `myelem`，すなわち，`myelem x xs` は，`x` がリスト `xs` の要素のとき `True`，そうでないとき `False` を返す．

(iv) 長さ 2 のリストを反転し，それ以外のリストは変更しない関数 `rev_len2`.

5.3 Maybe 型

ある型 α の上の Maybe 型の値は定数構成子 `Nothing` と 1 引数構成子 `Just` を用いて定義される．このとき `Just` の引数は α 型でなければならない．

```
> :type Nothing
Nothing :: Maybe a
> :type Just 'A'
Just 'A' :: Maybe Char
> :type Just "A"
Just "A" :: Maybe [Char]
```

空リストの場合と同様に，Nothing の型が Maybe a と型変数 a を用いて表されているのは何型のオプション型かが型推論できないためである．以下に，Maybe 型上の関数定義の例を記す．参考にされたし．

```
find _ [] = Nothing
find x ((y, v):zs) = if x==y then Just v else find x zs
insert (l,v) zs =
    case find l zs of
        Nothing -> Just ((l,v):zs)
        Just v1 -> if v==v1 then Just zs else Nothing
```

関数 find の定義に用いた “_” は匿名変数（anonymous variable）またはワイルドカード変数（wildcard variable）と呼ばれる表現である．名前を知る必要のない変数を表現するのに用いる．最後に，関数 insert の実行例を記しておく．

```
> insert (2, "Noodle") [(0, "Rice"), (1, "Bread")]
Just [(2,"Noodle"),(0,"Rice"),(1,"Bread")]
> insert (0, "Noodle") [(0, "Rice"), (1, "Bread")]
Nothing
```

問 4 上で定義した関数 insert がどのような関数か説明せよ．

5.4 関数型

関数型は -> を用いて定義される．例えばリスト型の節で紹介したリスト反転関数 (rev1, rev2, rev3) の型は [a] -> [a] となる．また，複数の引数をとる関数の型は a -> b -> c のように -> を並べる事によって表現される．なお，ここでは a -> (b -> c) の括弧が省略されている．

6 新しい型

当然のことながら Haskell でもユーザが新しい型を定義する事が出来る．本節では新しい型の導入方を簡単に説明する．

6.1 型の別名

キーワード type を用いると型に別名を与えることにより新しい型を導入することができる．例で見てみよう．

```
> type Symbol = String
```

上記のように新しい型 Symbol が定義できた．これは文字列の型 String に別名 Symbol を割り当てた事に相当する．さらに，より一般化して一つ以上の型変数をパラメーターとして含む多相型と呼ばれる型の集合を定義する事も出来る．

```
> type Set a = [a]
```

集合 $\{1, 2, 3\}$ をリスト [1,2,3] で表現し，集合を要素が重複しないリストとして Haskell で実現するとしよう．この仕様では，[1,2,3] と [2,3,1] などは等価であると判断されるべきであろうし，[1,2,2,3] などは予期せぬ値と判断されるべきであろう．Set a 型を明示的に与えることによって，取り扱ってるリストが集合の表現としてのリストか通常のリストかを明示的に宣言できるため，プログラムの保守性などの観点から重要である（7.5 節参照）．

6.2 新しい型の導入

type 宣言は略記の定義に制限されているので表現に限界がある．しばしば新しい構造を持つデータ型を生成したい事がある．例えばこれまで学んできた型では木の概念を表現する事が出来ない．そのためには，新しいデータ型に属する値を構築するのに識別子を演算子として用いるデータ構成子 (data constructor) が必要である．まずは簡単な例で見てみよう．

```
> data Fruit = Apple | Grape | Orange deriving Show
```

この宣言により，3 個の値 Apple, Grape, Orange を構成要素とするデータ型 Fruit が定義できた．この様な単純な場合でも data の定義は他のいかなる型の略記でもない新しい型を定義している点に注意する．この点で type による宣言とは異なる．なお，deriving Show の部分は定義した型のデータを印字するためのものである．この部分を省略してもデータ型の定義は可能であるが，定義した型のデータはそのままでは印字できない（例えば，print Apple の結果はエラーになる）．

さらに，data は再帰的な型も定義できる．

```
> data Term = Node (String, [Term])
```

Term 型のデータを用いると，項 F(x, A) を Node ("F", [Node ("x", []), Node ("A", [])]) と表現することができる．

7 その他

本節では前節までに紹介できなかったがゼミで必要になると思われる各種概念を簡単に解説する．

7.1 高階関数

Haskell では，関数も一人前のデータ として扱われる．つまり，関数に任意の名前をつけたり，関数を引数や結果とするような関数 (高階関数) を定義することもできる．高階関数の例として関数と値を受け取り，その関数を与えられた値に対して 2 度適用して得られる値を計算する関数 twice を挙げておこう．

```
> twice f x = f (f x)
> add x y = x + y
> twice (add 2) 5
9
```

ここで，関数 add は所謂 2 引数関数[1]であるにも関わらず，(add 2) のような式も許されていることに注意．

7.2 相互再帰

相互に再帰（mutually recursive）であるような複数の関数を定義したい時がある．つまり，呼び出しの度にグループ内の他の関数を少なくとも一つ呼び出すのである．例えば，入力したリストの第一要素から一つおきに中身を取り出す関数 take と，第二要素から一つおきに中身を取り出す関数 skip を相互再帰で定義すると以下のようになる．

```
mytake [] = []
mytake (x:xs) = x:(skip xs)
skip [] = []
skip (x:xs) = mytake xs
```

[1] 本当は 1 引数関数．興味がある方は文献 [1] などで “カリー化” をキーワードに調べるべし．

実行例を記す.

```
> mytake [1,2,3,4,5]
[1,3,5]
> skip [1,2,3,4,5]
[2,4]
```

問 5 偶数長のリストを入力すると True，それ以外の入力には False を返す関数 evenlist と，奇数長のリストを入力すると True，それ以外の入力には False を返す関数 oddlist を相互再帰で定義せよ.

7.3 λ記法

Haskell では λ 記法も自然に記述できる．例えば λ 式 $\lambda x.x$ や $\lambda x.(\lambda y.x)$ はそれぞれ Haskell で\x -> x, \x -> (\y -> x) のように記述できる．例として，λ 記法を用いて階乗関数を定義してみよう．なお，この関数定義が何をやってるかは理解しなくても良い[2].

```
> fix f x = f (fix f) x
> fact4 = fix (\f -> (\n -> if n==0 then 1 else n * f (n-1)))
> fact4 3
6
```

7.4 let による局所環境と局所定義

ときとして一時的な値，すなわち局所変数を関数内部に生成したい時や，局所的に関数を定義したい時がある．このような場合は構文 let _ in _ を用いる．実際に例で見てみよう.

```
fact5 0 = 1
fact5 n =
    let
        multi x y = x * y
        z = fact5 (n-1)
     in
        multi n z
```

実行例を記す.

```
> fact5 3
6
> multi

<interactive>:2:1: error: Variable not in scope: multi
> z

<interactive>:3:1: error: Variable not in scope: z
```

上記の例のように，let 文で局所的に定義された変数 z や関数 multi は in _ の内側にのみ影響を及ぼす.

[2]興味がある方は λ 計算かプログラム理論の教科書で “不動点演算子” をキーワードに調べるべし.

7.5 モジュール

新しいプログラミング言語設計において，情報のカプセル化 (encapsulation) をいかに実現するかは重要な課題である．情報のカプセル化とは，型やその型上での関数等の概念をグループ化して限定された方法でのみ使えるようにする事である．使用の限定とはプログラミングを困難にするものではなく，予期に反した使用からのデータの保護やプログラム・コードの再利用の促進などに関して大変有用である．

モジュールを使用して，6.1 節で紹介した集合を表現する `Set a` 型と，和集合を求める演算子 $\cup$ に対応する `Set a` 型上の関数 `union` を定義してみよう．このとき，リストの単純な連結ではなく，二つの入力リストに要素の重複が無ければ（すなわち，集合の表現であれば），出力にも要素の重複が無いように定義されてることに気をつけたし．なお，以下の定義に現れる関数 `elem` は，入力の要素 `x` リスト `ys` に対して，`x` が `ys` に含まれるか否かを返す．`union` の型宣言部において `Eq a => ...` と記述している部分は，型変数 `a` が同等クラス `Eq` に属するという制約を表している．これは，関数 `elem` はその定義において要素間の等価判定を扱うため，`union` の引数として与えられる集合の要素は等価判定が可能であることが要求される．`Eq` は等価判定が可能なデータの型のクラスを表す．

```
module Set where

type Set a = [a]

union :: Eq a => Set a -> Set a -> Set a
union [] ys = ys
union (x:xs) ys =
    if elem x ys then union xs ys else x:(union xs ys)
```

上述の内容をファイル `Set.hs` に保存する．同じディレクトリ内にある他のプログラムファイル内で `Set` 型や関数 `union` を利用する際には，そのファイルで `import Set` と書くことによって `Set` モジュールをインポートできる．

問 6 上で定義した関数 `union` と同様に，積集合を求める演算子 $\cap$ に対応する `Set a` 上の関数 `intersection` を定義せよ．

8 練習問題

問 7 以下の関数を定義せよ．ただし指定されていない入力に対する評価は各自で考えられたし．

(i) 整数 n の入力に対して，その平方 n^2 を返す関数 `square`.

(ii) 整数 n の入力に対して，n が偶数の時は `True`，そうでないときは `False` を返す関数 `myeven`.

(iii) 次の性質を持つ関数 `myzip`.

$$\texttt{myzip}\ [a_1, a_2, \ldots, a_k]\ [b_1, b_2, \ldots, b_m] = [(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)]$$

ただし，n は k と m の小さい方の値とする．

(iv) 次の性質を持つ関数 `myconcat`.

$$\texttt{myconcat}\ [[a_{11}, a_{12}, \ldots, a_{1n_1}], \ldots, [a_{m1}, a_{m2}, \ldots, a_{mn_m}]] = [a_{11}, a_{12}, \ldots, a_{1n_1}, \ldots, a_{m1}, a_{m2}, \ldots, a_{mn_m}]$$

(v) 次の性質を持つ関数 `myand`.

$$\texttt{myand}\ [b_1, b_2, \ldots, b_n] = b_1 \wedge b_2 \wedge \cdots \wedge b_n$$

(vi) 次の性質を持つ関数 myor.

$$\texttt{myor}\ [b_1, b_2, \ldots, b_n] = b_1 \vee b_2 \vee \cdots \vee b_n$$

(vii) 述語 (ブール型の値を返す関数)p とリスト xs を引数にとり，要素が p を満たすような xs の部分リストを返す関数 myfilter. 例えば，(ii) で定義した述語 myeven に対して，myfilter myeven [1,2,3,4,5] = [2,4] となる.

(viii) 述語 p とリスト xs を引数にとり，xs の全ての要素が p を満たせば True，そうでないときは False を返す述語 myall. 例えば，(ii) で定義した述語 myeven に対して，myall myeven [1,2,3,4,5] = False，myall myeven [2,4] = True となる.

(ix) 述語 p とリスト xs を引数にとり，xs の要素の少なくとも一つが p を満たせば True，そうでないときは False を返す述語 myany. 例えば，(ii) で定義した述語 myeven に対して，
myany myeven [1,2,3,4,5] = True，myany myeven [1,3,5] = False となる.

問 8 *Haskell* では以下で定義される関数 map が組み込みで定義されている.

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

整数型のリスト $[a_1, \ldots, a_n]$ の入力にたいし，各要素の平方の和である $a_1^2 + \cdots + a_n^2$ を計算する関数 squaresum を map 関数と mysum 関数を用いて定義せよ.

問 9 以下の順に各関数を定義し，整数 n が素数であるかどうかを判定する述語 isPrime を定義せよ.

(i) リスト xs と整数 n を入力すると，全てのリストの要素が n の約数でないとき True，そうでないとき False を返す述語 divides.

(ii) 整数 n を入力とし，n が素数のとき True，そうでないとき False を返す述語 isPrime を関数 divides を用いて定義せよ.

問 10 連想リスト *(association list)* とはラベルと値の対のからなる以下の条件を満たすリスト A である.

$$\forall (l1, v1), (l2, v2) \in A.\ l1 = l2 \Rightarrow v1 = v2$$

連想リストを以下で定義される連想型 Assoc で表現する.

```
type Assoc l v = [(l, v)]
```

以下の関数を定義せよ.

(i) ラベル l と連想リスト zs を入力とすると，(l,v) の形の要素が zs に存在するときは Just v を，そうでないときは Nothing を返す関数 find. ここで，Just や Nothing は Maybe 型の値を表現する組み込みの構成子であったことに注意.

(ii) 連想リスト xs と ys を入力すると，集合 xs ∪ ys が連想リストになってるときは Just zs を，そうでないときは Nothing を返す関数 append. ただし，リスト zs は集合 xs ∪ ys に対応する連想リストであるとする.

問 11 自然数に対応するデータ型 Nat とデータ型 Nat 上の足し算 add は以下で定義される．

```
data Nat = Zero | Succ Nat deriving Show
add Zero y = y
add (Succ x) y = Succ (add x y)
```

なお，自然数 $0, 1, 2, \ldots$ はデータ型 Nat においては Zero，Succ Zero，Succ (Succ Zero), ... と表現される．

(i) データ型 Nat 上の掛け算 mul を定義せよ．

(ii) データ型 Nat 上の引き算 sub を定義せよ．なお，計算可能性の分野において自然数上の引き算は，$2-3$ のような第二引数が第一引数より大きい値の場合，返し値を 0 で定義するのが通例である．

(iii) Nat 型の式 2 つを入力とし，自然数として大きい方の値を返す関数 maxNat を関数 add, mul, sub のみを用いて定義せよ．

(iv) Nat 型の式 2 つを入力とし，自然数として小さい方の値を返す関数 minNat を関数 add, mul, sub のみを用いて定義せよ．

参考文献

[1] R.Bird and P.Wadler 著, 武市正人訳, 関数プログラミング, 近代科学社, ISBN 4-7649-0181-1, 1991.

[2] Graham Hutton 著，山本 和彦 訳，プログラミング Haskell 第 2 版，技術書出版ラムダノート，ISBN: 978-4908686078，2019.