情報システム科学実習II第6回
レコード型/ヴァリアント型とその応用

担当: 山口 和紀・五十嵐 淳

2001年11月21日

レコード型やヴァリアント型は，組型，リスト型と同様に，そのような名前の一つの型が存在するわけではなく，構造のあるデータを導入するための，似たような構造を持った型の種類である．レコードは組と同様にいくつかの値に名前をつけて並べることで構成されるデータであるのに対し，ヴァリアントは異なる種類の値を同じ型の値として混ぜて扱うためのものである．また，ヴァリアント型はリストなどの再帰的な構造を持つデータ構造を導入するために使われる．どちらも，使用するためにはプログラマが具体的な型の構造を宣言をする必要がある．問題領域に応じて適切なデータ型をデザイン・定義することは (OCaml に限らず) プログラミングにおいて非常に重要な技術である．

1 レコード型

レコード型の値は，組と同様に，本質的には値をいくつか並べたものである．組との違いは，それぞれの要素に名前を与えて，その名前でレコードの要素にアクセスできることである．この名前と値の組を**フィールド**(*field*)，フィールドの名前をフィールド名，と呼ぶ．

例えば，学生のデータベースを作ることを考えよう．学生一人一人のデータは名前を表す name フィールド，学生証番号を表す id フィールドの並びとする．新しいレコード型は，type 宣言をつかって定義することができる．

```
# type student = {name : string; id : int};;
type student = { name : string; id : int; }
```

student が新しい型の識別子である．一般的には，各フィールドに格納される値の型をフィールド名とともに並べ，{} で囲むことでレコード型を示す．

type ⟨型名⟩ = {⟨フィールド名$_1$⟩ : ⟨型$_1$⟩; ...; ⟨フィールド名$_n$⟩ : ⟨型$_n$⟩}

定義されたレコードの値は，

{⟨フィールド名$_1$⟩ = ⟨式$_1$⟩; ...; ⟨フィールド名$_n$⟩ = ⟨式$_n$⟩}

で構成することができる．

```
# let st1 = {name = "Taro Yamada"; id = 123456};;
val st1 : student = {name="Taro Yamada"; id=123456}
```

レコードの要素は，組と同様にパターンマッチでアクセスする方法と，フィールドひとつの値のみをドット記法と呼ばれる方法でアクセスする方法がある．レコードパターンは，

{⟨ フィールド名$_1$⟩ = ⟨ パターン$_1$⟩; ...; ⟨ フィールド名$_n$⟩ = ⟨ パターン$_n$⟩}

という形で表される．

```
# let string_of_student {name = n; id = i} = n ^ "'s ID is " ^ string_of_int i;;
val string_of_student : student -> string = <fun>
# string_of_student st1;;
- : string = "Taro Yamada's ID is 123456"
```

または ⟨ レコード ⟩.⟨ フィールド名 ⟩ という形で ⟨ レコード ⟩ から ⟨ フィールド名 ⟩ に対応する要素を取り出すことができる．

```
# let string_of_student st = st.name ^ "'s ID is " ^ string_of_int st.id;;
val string_of_student : student -> string = <fun>
```

またレコードの一部のフィールドを変更しただけの新しいレコードを生成したい場合には，

{⟨ レコード式 ⟩ with
⟨ フィールド名$_1$⟩ = ⟨ パターン$_1$⟩; ...; ⟨ フィールド名$_n$⟩ = ⟨ パターン$_n$⟩}

という式でできる．長いレコードの一部だけ変更したいときに便利である．

```
# type teacher = {tname : string; room : string; ext : int};;
type teacher = { tname : string; room : string; ext : int; }
# let t1 = {tname = "Atsushi Igarashi"; room = "604B"; ext = 46808};;
val t1 : teacher = {tname="Atsushi Igarashi"; room="604B"; ext=46808}
# let t2 = {t1 with room = "605B"};;
val t2 : teacher = {tname="Atsushi Igarashi"; room="605B"; ext=46808}
```

この時，気をつけなければいけないのが，with はフィールドのへの代入を行うのではなく，新しいレコードを生成しているということである．t1 が束縛されたレコードの room フィールドの値は t2 の定義後も変っていない．つまり，

```
# t1;;
- : teacher = {tname="Atsushi Igarashi"; room="604B"; ext=46808}
```

t1 の値は t2 の定義前後で変化してないことに注意．

組型とレコード型の違いについて　レコード型が組型と違う点は，各要素に名前がつき，その名前でパターンマッチを介さず要素にアクセスできること，一部の要素だけを変更したレコードを容易に作ることができることの他に，型宣言をしないと使えないということがある．また，レコード型は常に名前で参照され，{...} という表現は型そのものとしては 使えない．つまり，引数の型として

```
let f (x : {name : string; id : int}) = ...
```

としたり，入れ子になったレコードを宣言するのに，

```
type student_teacher =
  {s : {name : string; id : int};
   t : {tname : string; room : string; ext : int}};;
```

と宣言することはできない．正しくは内側のレコード型を宣言しておいて，

```
# type student_teacher = {s : student; t : teacher};;
type student_teacher = { s : student; t : teacher; }
```

と行う．ただし，ネストしたレコードの値やパターンは直接構成可能である．

```
# let st = {s = {name = "Taro Yamada"; id = 123456}; t = t1};;
val st : student_teacher =
  {s={name="Taro Yamada"; id=123456};
   t={tname="Atsushi Igarashi"; room="604B"; ext=46808}}
```

型名/フィールド名についての注意・名前空間について　型名・フィールド名に使用できるのは変数名に使用できるのと同様な文字列である．フィールド名は type 宣言でレコード型を宣言することで，はじめて使用することができる．同じフィールド名を持つ別の型を宣言すると，変数の再宣言と同様，古いフィールド名の定義は隠されてしまうので注意が必要である．つまり，

```
# type foo = {name : bool};;
type foo = { name : bool; }
```

などと name フィールドを持つ別のレコード型を宣言してしまうと，新たに student の値を構成したり，name フィールドにアクセスなどができなくなってしまう．

```
# {name = "Ichiro Suzuki"; id = 51};;
Characters 9-24:
This expression has type string but is here used with type bool
# st1.name;;
Characters 0-3:
This expression has type student but is here used with type foo
```

ただし，変数名，フィールド名，型名は別の**名前空間**(*name space*) に属しているので，例えば，宣言済のフィールド名と同じ変数を用いても，フィールド名が隠されたりすることはない．

2 ヴァリアント型

ヴァリアント型が何かを説明する前に，動機付けの例として，様々な図形のデータを扱うことを考えてみよう．扱うのは以下の 4 種類の図形で，それぞれ形状を決定するためのデータをもつとする．

- 点はどれもみな同じ形・大きさをしている．
- 円は，形はどれも同じで，半径の長さで大きさを一意に決定できる．つまり整数 4, 9 などが 円を特徴づけるために必要なデータである．
- 長方形は長辺と短辺の長さで大きさ・形が決まる．
- 正方形は形はどれも同じで一辺の長さで大きさが決まる．

ここで，このような図形データを扱おうとすると，

- (2，4) のような長方形を示す値と，4 という円を示す値を混ぜて使いたい
- さらに悪いことに，7 という整数は，円も正方形も表しうる

という問題がある．ヴァリアント型はこのように，異なるデータ (表現が同じでも意味が異なる場合も含めて) を混ぜて，一つの型として扱いたい際に使うもので，ひとつひとつの値は，データの本体 ((2，4) や 8 など) に，そのデータはなにを表すためのものか (円，長方形，正方形) の情報を付加したもので表される．

では，図形を表すヴァリアント型を定義してみよう．

```
# type figure =
#   Point
# | Circle of int
# | Rectangle of int * int
# | Square of int;;
type figure = Point | Circle of int | Rectangle of int * int | Square of int
```

ヴァリアント型もレコード型と同様 type を使って宣言する．figure は新しい型の名前である．Point, Circle, Rectangle, Square は**コンストラクタ**(*constructor*) と呼ばれ，上で述べた「そのデータは何を表すためのものか」という情報に相当する．of の後には，コンストラクタが付加される値の型を記述する． of 以下は省略可能で，省略した場合コンストラクタ自体がその型の値となる．(この例では，点のように同じコンストラクタを持つ物同士は区別する必要がないので省略している．) ヴァリアント型の値は，コンストラクタを関数のように対応する型の値に「適用」することによって構成される．

```
# let c = Circle 3;;
val c : figure = Circle 3
# let figs = [Point; Square 5; Rectangle (4, 5)];;
val figs : figure list = [Point; Square 5; Rectangle (4, 5)]
```

次に，与えられた図形の面積を求める関数を考えてみよう．まずは，図形がどの種類のものであるかを調べて，それによって場合わけをする必要がある．この場合わけは match 式で行うことができる．ヴァリアント型の値に対するパターンは

⟨コンストラクタ⟩⟨パターン⟩

という形で，コンストラクタが適用された値が⟨パターン⟩にマッチすることになる．⟨パターン⟩が省略された場合には引数のないコンストラクタにマッチすることになる．

```
# let area = function
#     Point -> 0
#   | Circle r -> r * r * 3  (* elementary school approximation :-) *)
#   | Rectangle (lx, ly) -> lx * ly
#   | Square l -> l * l;;
val area : figure -> int = <fun>
```

引数に対しすぐにマッチングを行うので，function を使用している．(前回のプリント参照)

```
# area c;;
- : int = 27
# map area figs;;
- : int list = [0; 25; 20]
```

その他のパターン　ヴァリアント型に対するパターンマッチでは，基本的にコンストラクタの数だけ場合分けをかかなければならないが，いくつかの場合で処理が共通している場合には **or パターン** と呼ばれるパターンでまとめることができる．次の関数は，与えられた図形を囲むことができる正方形を返す関数である．(長方形は最初の辺が短辺と仮定する．)

```
# let enclosing_square = function
#     Point -> Square 1
#   | Circle r -> Square (r * 2)
#   | Rectangle (_, l) | Square l -> Square l;;
val enclosing_square : figure -> figure = <fun>
```

Rectangle (_, l) | Square l が or パターンと呼ばれるもので，一般的には

⟨パターン$_1$⟩ | ⟨パターン$_2$⟩

と書かれ，どちらかにパターンにマッチするものが全体にマッチする．各部分パターンで導入される変数 (群) は，(どちらのパターンにマッチしても -> 以降の処理がうまくいくように) 同じ名前/型でなければならない．

また，複数の値に対してマッチをとるときには組を利用してマッチをとるのがよい．次の関数は，二つの図形が相似であるかを判定する関数である．or パターンと，組の利用法に注意．

```
# let similar x y =
#   match (x, y) with
#     (Point, Point) | (Circle _, Circle _) | (Square _, Square _) -> true
#   | (Rectangle (l1, l2), Rectangle (l3, l4)) -> (l3 * l2 - l4 * l1) = 0
#   | _ -> false;;
val similar : figure -> figure -> bool = <fun>
# similar (Rectangle (2, 4)) (Rectangle (1, 2));;
- : bool = true
```

コンストラクタの名前について　コンストラクタの名前は，変数名，型名と違って，一文字目が英大文字でなければならない．より正確には

1. 一文字目が英小文字またはアンダースコア (_) で，
2. 二文字目以降は英数字 (A...Z, a...z, 0...9)，アンダースコアまたはプライム (')

であるような任意の長さの文字列である．

コンストラクタもレコード型のフィールド名と同じように，同じ名前のものを再宣言してしまうと，前に定義されたコンストラクタを伴う型は使い物にならなくなってしまうので注意すること．

3　ヴァリアント型の応用

列挙型　Pascal, C, C++ の列挙型 (enum 型) は引数を取らないコンストラクタだけからなるヴァリアント型で表現することができる．

```
# type color = Black | Blue | Red | Magenta | Green | Cyan | Yellow | White;;
type color = Black | Blue | Red | Magenta | Green | Cyan | Yellow | White
```

enum の値が実は整数でしかない C とは違い，コンストラクタ同士の足し算などは当然定義されない．(そういう関数を書かない限り．) この型上の関数は 8 つの場合わけを行わなければいけない．

```
# let reverse = function
#     Black -> White | Blue -> Yellow | Red -> Cyan | Magenta -> Green
#   | Green -> Magenta | Cyan -> Red | Yellow -> Blue | White -> Black;;
val reverse : color -> color = <fun>
```

bool 型も列挙型の一種と見なすことができる．

```
type bool = true | false
```

また，if 式は match 式の変形と見なすことができる．

```
if ⟨式1⟩ then ⟨式2⟩ else ⟨式3⟩
∼ match ⟨式1⟩ with true -> ⟨式2⟩ | false -> ⟨式3⟩
```

この宣言自身は OCaml ではコンストラクタの名前の制限などもあり実際には行えないが，**bool** 型もヴァリアントの一種と考えるのは理解の助けになるだろう．

再帰ヴァリアント型　ヴァリアント型は of 以下に今宣言しようとしている型名を参照することで再帰的なデータ型を定義することも可能である．ここでは，もっとも単純な再帰的データの例として，自然数を表す型を考えてみよう，自然数は以下のように再帰的に定義できる．

- ゼロは自然数である．
- 自然数より 1 大きい数は自然数である．

「ゼロ」「1 大きい」という部分をコンストラクタとして考えると次のような型定義が導かれる．

```
# type nat = Zero | OneMoreThan of nat;;
type nat = Zero | OneMoreThan of nat
# let zero = Zero and two = OneMoreThan (OneMoreThan Zero);;
val zero : nat = Zero
val two : nat = OneMoreThan (OneMoreThan Zero)
```

この自然数上での加算は，データ自体の再帰的定義に従って，

- ゼロに自然数 n を足したものは n である．
- m より 1 大きい数に n を足したものは，m と n を足したものに 1 大きい数である．

と定義できる．

```
# let rec add m n =
#   match m with Zero -> n | OneMoreThan m' -> OneMoreThan (add m' n);;
val add : nat -> nat -> nat = <fun>
# add two two;;
- : nat = OneMoreThan (OneMoreThan (OneMoreThan (OneMoreThan Zero)))
```

実は，この構造はよく見てみると，リストに良く似ていることに気づく．丁度 Zero は空リスト []，OneMoreThan は cons に対応している．本質的な違いは格納できる要素の有無だけである．nat の構造を拡張すれば，例えば，整数リストを表現する型を簡単に定義することができる．

```
# type intlist = INil | ICons of int * intlist;;
type intlist = INil | ICons of int * intlist
```

(多相的なリストの定義は下で行う．)

またヴァリアント型定義はふたつの型定義を and で結ぶことによって，相互再帰的にすることも可能である．下の例は，実数と文字列が交互に現れるリストを表現したものである．

```
# type fl_str_list = FNil | FCons of float * str_fl_list
# and str_fl_list = SNil | SCons of string * fl_str_list;;
type fl_str_list = FNil | FCons of float * str_fl_list
type str_fl_list = SNil | SCons of string * fl_str_list
# let fslist = FCons (3.14, SCons ("foo", FCons (2.7, SNil)));;
val fslist : fl_str_list = FCons (3.14, SCons ("foo", FCons (2.7, SNil)))
```

この型上の関数は自然に相互再帰の二つの関数で定義される．次の関数は，この 2 種類のリスト (実数から始まるものと，文字列から始まるもの) の長さを計算する関数である．

```
# let rec length_fs = function
#     FNil -> 0
#   | FCons (_, rest_sf) -> 1 + length_sf rest_sf
# and length_sf = function
#     SNil -> 0
#   | SCons (_, rest_fs) -> 1 + length_fs rest_fs;;
val length_fs : fl_str_list -> int = <fun>
val length_sf : str_fl_list -> int = <fun>
# length_fs fslist;;
- : int = 3
```

多相的ヴァリアント型　intlist と同様にして，stringlist などを定義してゆくと，定義が酷似していることがわかる．結局，

- 末端を表すコンストラクタ
- 要素を ...list に連結するためのコンストラクタ

という構造が共通しているためである．また違いは要素の型だけである． OCaml では，多相型関数が，(概念的に) 関数中の型情報をパラメータ化することで様々な型の値に適用できるのと同様，型定義の一部分をパラメータ化することも可能である．ただし，関数とは違い，型パラメータを明示的に型宣言中に示してやる必要がある．多相型リストの型をヴァリアント型で定義するとすれば，

```
# type 'a list = Nil | Cons of 'a * 'a list;;
type 'a list = Nil | Cons of 'a * 'a list
```

といった定義になる．'a が型パラメータを表している．

その他，頻繁に使われる多相的データ型としては，オプション型という以下で定義される型がある．

```
# type 'a option = None | Some of 'a;;
type 'a option = None | Some of 'a
```

典型的には None が例外的な「答えがない」値を表し，正常に計算が行われた場合に Some v という形で v という値が返ってくる．(Java, C などでは，たいてい null もしくは NULL を用いて例外的な値を示しているのと似ている．) オプション型は ocaml 起動時に定義済の型である．

ヴァリアント型宣言のまとめ　ヴァリアント型は，一般的には以下のような文法で宣言される．[] で囲まれた部分は省略可能である．

```
type [⟨型変数1⟩] ⟨型名1⟩ =
    ⟨コンストラクタ11⟩ [of ⟨型11⟩] | ··· | ⟨コンストラクタ1n⟩ [of ⟨型1n⟩]
and [⟨型変数2⟩] ⟨型名2⟩ =
    ⟨コンストラクタ21⟩ [of ⟨型21⟩] | ··· | ⟨コンストラクタ2m⟩ [of ⟨型2m⟩]
and [⟨型変数3⟩] ⟨型名3⟩ = ···
 ⋮
```

4 Case Study: 二分木

木(*tree*) 構造はリスト，配列などと並ぶ代表的なデータ構造で様々なところに応用が見られる．木は，ラベルと呼ばれるデータを格納するための**ノード**(*node*) の集まりから構成され，各ノードは，0 個以上の**子ノード**(*child node*) を持つような階層構造をなしている．階層構造の一番「上」のノードを**根**(*root*) と呼ぶ．木は様々なところに応用されていて，UNIX のファイルシステムは木構造の一例である．木構造のうち各ノードの子供の (最大) 数 n が決まっているものを n 分木と呼ぶ．$n = 1$ のものはリストと同様な構造になる．ここでは最も単純な二分木を扱っていく．

二分木構造は再帰的に次のように定義することができる．

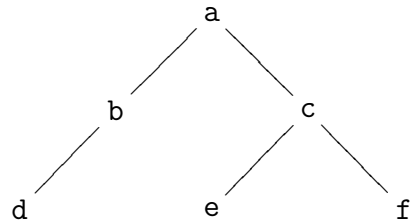
- (ラベルをもたない) 空の木 (ここでは，葉または leaf と呼ぶ) は二分木である．
- ふたつの二分木 left と right を子ノードとするノードは二分木を構成する．

これを OCaml の型定義に直したものが以下の定義である．

```
# type 'a tree = Lf | Br of 'a * 'a tree * 'a tree;;
type 'a tree = Lf | Br of 'a * 'a tree * 'a tree
```

`Lf` は葉，`Br` (枝/branch) はノードのことで，左右の部分木とそのラベルから `tree` を構成する．また，リストと同様，データ構造を記述するデータ型であるのでラベルの型は特定していない (多相的である)．

実際の木の例を見てみよう．例えば，



のような文字からなる木は，

```
# let chartree = Br ('a', Br ('b', Br ('d', Lf, Lf), Lf),
#                         Br ('c', Br ('e', Lf, Lf), Br ('f', Lf, Lf)));;
val chartree : char tree =
  Br
   ('a', Br ('b', Br ('d', Lf, Lf), Lf),
    Br ('c', Br ('e', Lf, Lf), Br ('f', Lf, Lf)))
```

と表現できる．子ノードのないノードは Br（.., Lf, Lf）で表している．

木の大きさを計る指標としては，(ラベルつき) ノードの数や，根から一番「深い」ノードまでの深さを用いる．以下の関数はそれらを求めるものである．

```
# let rec size = function
#     Lf -> 0
#   | Br (_, left, right) -> 1 + size left + size right;;
val size : 'a tree -> int = <fun>
# let rec depth = function
#      Lf -> 0
#    | Br (_, left, right) -> 1 + max (depth left) (depth right);;
val depth : 'a tree -> int = <fun>
```

明らかに，二分木 t に対しては常に，$size(t) \leq 2^{depth(t)} - 1$ が成立する．また，$size(t) = 2^{depth(t)} - 1$ であるような二分木を **完全二分木**($complete\ binary\ tree$) と呼ぶ．

```
# let comptree = Br(1, Br(2, Br(4, Lf, Lf),
#                            Br(5, Lf, Lf)),
#                      Br(3, Br(6, Lf, Lf),
#                            Br(7, Lf, Lf)));;
val comptree : int tree =
  Br
   (1, Br (2, Br (4, Lf, Lf), Br (5, Lf, Lf)),
    Br (3, Br (6, Lf, Lf), Br (7, Lf, Lf)))
```

は，深さ 3 の完全二分木の例である．

```
# size comptree;;
- : int = 7
# depth comptree;;
- : int = 3
```

さて，木構造から要素を列挙する方法を考えてみよう．列挙するためには，ラベルデータに適当な順序をつける必要がある．この順序の付け方の代表的なもの 3 つ，**行きがけ順**($preorder$)，**通りがけ順**($inorder$)，**帰りがけ順**($postorder$) を紹介する．列挙する関数はいずれもリストを返す関数として定義されるが，説明としては木のノードを訪れる順番のつけかたとして説明する．

行きがけ順は，訪れたラベルをまず取り上げてから，左の部分木，右の部分木の順でノードを列挙する．

```
# let rec preorder = function
#     Lf -> []
#   | Br (x, left, right) -> x :: (preorder left) @ (preorder right);;
val preorder : 'a tree -> 'a list = <fun>
# preorder comptree;;
- : int list = [1; 2; 4; 5; 3; 6; 7]
```

通りがけ順は，まず左の部分木から始めて，右の木に行く前に，ラベルを取り上げる．

```
# let rec inorder = function
#     Lf -> []
#   | Br (x, left, right) -> (inorder left) @ (x :: inorder right);;
val inorder : 'a tree -> 'a list = <fun>
# inorder comptree;;
- : int list = [4; 2; 5; 1; 6; 3; 7]
```

帰りがけ順は，まず，部分木を列挙してから，最後にノードラベルを取り上げる．

```
# let rec postorder = function
#     Lf -> []
#   | Br (x, left, right) -> (postorder left) @ (postorder right) @ [x];;
val postorder : 'a tree -> 'a list = <fun>
# postorder comptree;;
- : int list = [4; 5; 2; 6; 7; 3; 1]
```

これらの定義は，@ を使っているせいで，サイズに比べて深さが大きいような (アンバランスな) 木に対して効率が良くない．これを改善したのが次の定義である．列挙済要素を引数として追加し，各ノードではそのリストに要素を追加 (cons) することだけで，計算を行っている．(この類いの定義は，効率を良くするためにわりとよくとられる手であるので，上の素直な定義を理解したうえで，マスターする価値はある．)

```
# let rec preord t l =
#   match t with
#     Lf -> l
#   | Br(x, left, right) -> x :: (preord left (preord right l));;
val preord : 'a tree -> 'a list -> 'a list = <fun>
# preord comptree [];;
- : int list = [1; 2; 4; 5; 3; 6; 7]
```

二分探索木　二分探索木(*binary search tree*) は，順序のつけられるデータを格納するときに，あとで検索がしやすいように，一定の規則にしたがってデータを配置した木である．具体的には，あるノードの左の部分木にはそのノード上のデータより小さいデータだけが，右の部分木には大きいデータだけが並んでいるような木である．例えば

```
Br (4, Br (2, Lf, Br (3, Lf, Lf)), Br (5, Lf, Lf))
```

の表す木は二分探索木であるが，

```
Br (3, Br (2, Br (4, Lf, Lf), Lf), Br (5, Lf, Lf))
```

はそうではない．

二分探索木上で，ある要素が木の中にあるかを問い合わせる mem，要素を木に追加する add 関数はそれぞれ以下のようになる．

```
# let rec mem t x =
#   match t with
#     Lf -> false
```

```
#   | Br (y, left, right) ->
#       if x = y then true
#       else if x < y then mem left x else mem right x
# let rec add t x =
#   match t with
#     Lf -> Br (x, Lf, Lf)
#   | (Br (y, left, right) as whole) ->
#       if x = y then whole
#       else if x < y then Br(y, add left x, right) else Br(y, left, add right x);;
val mem : 'a tree -> 'a -> bool = <fun>
val add : 'a tree -> 'a -> 'a tree = <fun>
```

定義からわかるようにどちらも計算の構造は同じで,

- 空の木であれば，見つからないという false，もしくは加えようとする要素だけからなる木を返す.
- ノードであれば，ノード上のデータと比較を行う．等しければ，true を返すか，そのままの木 whole を返すし，等しくない場合は大小関係に応じて，左か右の部分木にむかって探索，追加を続行する.

同じデータの集合でも，様々な形の木が考えられる．探索の効率は，木の深さによるので，深さがなるべく浅い木を維持するのが二分探索木を扱う際の目標のひとつとなる.

5 Case Study: 無限リスト

もうひとつ，応用例として，無限のデータ列を表すデータ型をみてみよう．無限列を有限のメモリ上にそのまま表現することはできないので，工夫が必要である．ここでは無限列は，先頭要素と,「以降の (無限) 列を計算するための関数」の組で表現する．この「関数」は特に引数となるべき情報を必要としないので，unit からの関数として表現しよう．これを書いたのが以下の type 宣言である.

```
# type 'a seq = Cons of 'a * (unit -> 'a seq);;
type 'a seq = Cons of 'a * (unit -> 'a seq)
```

seq は list や tree のように要素型に関して多相的である． Cons は要素を先頭に追加するためのコンストラクタであり， of 以下の型が示すように，tail 部分は関数型になっている．このヴァリアント型はコンストラクタが一つしかないため，ヴァリアント型をわざわざ使わずに，要素と関数の組で表現することも原理的に可能である．このような型を宣言するのは，データ抽象の手段のひとつであると考えられる．つまり，プログラマがデータにこめた意味をプログラム上で読み取ることができ，たまたま同じ表現をもつ違う意味のデータと混乱する可能性が少なくなる，という意義がある.

次の関数 from は，整数 n から 1 ずつ増える無限列を生成する.

```
# let rec from n = Cons (n, fun () -> from (n + 1));;
val from : int -> int seq = <fun>
```

fun () -> に注目したい．無限列は，fun () -> ... 使って関数を構成し，tail 部の評価を，必要なときまで遅らせているから実現できているのである．同様な関数をリストを使って定義しても，

```
let rec list_from n = n :: list_from (n + 1)
```

ひとたび呼び出されると，list_from の呼び出しを無限に行ってしまうのでうまくいかない．

from の定義にみられるように，式の評価を遅らせるために導入される関数を**サンク**(*thunk*)と呼ぶ．余談であるが，サンクの明示的な導入は lazy な言語 (第 3 週のプリント参照) であれば必要のないところである (もともと部分式の評価は必要になるまで遅らされる) ため，このような無限の構造は lazy な言語の得意とするところである．

列から先頭要素，後続の列を返す関数，先頭から n 要素をリストとして取出す関数は，

```
# let head (Cons (x, _)) = x;;
val head : 'a seq -> 'a = <fun>
# let tail (Cons (_, f)) = f ();;
val tail : 'a seq -> 'a seq = <fun>
# let rec take n s =
#   if n = 0 then [] else head s :: take (n - 1) (tail s);;
val take : int -> 'a seq -> 'a list = <fun>
# take 10 (from 4);;
- : int list = [4; 5; 6; 7; 8; 9; 10; 11; 12; 13]
```

と定義される．tail 関数の本体ではサンクに () を適用して，後続列を計算している． コンストラクタがひとつのヴァリアント型に対しては，場合わけをする必要がない ので，match を使わずに引数にパターンを直接書いても支障がない． take はリストに対してのものと同様に定義できる．

さて，無限列に対する map を定義してみよう．

```
# let rec mapseq f (Cons (x, tail)) =
#   Cons (f x, fun () -> mapseq f (tail ()));;
val mapseq : ('a -> 'b) -> 'a seq -> 'b seq = <fun>
# let reciprocals = mapseq (fun x -> 1.0 /. float_of_int x) (from 2);;
val reciprocals : float seq = Cons (0.5, <fun>)
# take 5 reciprocals;;
- : float list = [0.5; 0.333333333333; 0.25; 0.2; 0.166666666667]
```

注目すべき点は，reciprocals の値を見るとわかるように，0.5 以降の要素は take をするまで実際には計算されていないことである．take の中で，tail を呼び出す度に逆数が計算されていく．

余談であるが，複雑なデータ型にたいする関数を書いていればいるほど型情報がデバグに役立つことが実感できる．上の関数定義で，つい，サンク導入のための fun () -> を書き忘れてしまったり，サンクの評価 (tail ()) をし忘れたりするのだが，型のおかげで (エラーメッセージになれれば) すぐに間違いを発見することができる．

エラトステネスのふるい　さて，もうすこし面白い無限列の応用例をみてみたい．エラトステネスのふるいは素数を求めるための計算方法で，2 以上の整数の列から，

- 先頭の数字 2 は素数である．
- 残りの列 (3, 4, 5, ...) から 2 の倍数 (4, 6, 8, ...) を消す．
- 残った数字列の先頭 3 は素数である．
- 残りの列 (5, 7, 9, ...) から 3 の倍数 (9, 15, ...) を消す．
- 残った数字列の先頭 5 は素数である．
- 以下同様

として素数の列を求める方法である．ここでは，seq 型を用いてエラトステネスのふるいを実現してみよう．まず必要なのは，整数列から n の倍数を取り去った列を返す関数 sift である．

```
let rec sift n = ...;;
```

これを使うとエラトステネスのふるいは，次のように定義できる．

```
# let rec sieve (Cons (x, f)) = Cons (x, fun () -> sieve (sift x (f())));;
val sieve : int seq -> int seq = <fun>

# let primes = sieve (from 2);;
val primes : int seq = Cons (2, <fun>)
# take 20 primes;;
- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71]
#
# let rec nthseq n (Cons (x, f)) =
#   if n = 1 then x else nthseq (n - 1) (f());;
val nthseq : int -> 'a seq -> 'a = <fun>
# nthseq 1000 primes;;
- : int = 7919
```

6　練習問題

Exercise 6.1 以下のレコード型 loc_fig は，図形に xy 平面上での位置情報をもたせたものである．正方形，長方形は，各辺が軸に並行であるように配置されていると仮定 (長方形に関しては，Rectangle (x, y) の x の表す辺が x 軸に並行，とする．) し，二つの図形が重なりを持つか判定する関数 overlap を定義せよ．

```
type loc_fig = {x : int; y : int; fig : figure};;
```

Exercise 6.2 nat 型の値をそれが表現する int に変換する関数 int_of_nat, nat 上の掛け算を行う関数 mul，nat 上の引き算を行う関数 (ただし $0 - n = 0$) monus (モーナス) を定義せよ.

Exercise 6.3 上の monus 関数を変更して, $0-n$ $(n > 0)$ は None を返す nat -> nat -> nat option 型の関数 minus を定義せよ.

Exercise 6.4 x を各ノードのラベルとし, 深さ n の完全二分木を生成する関数 comptree x n を定義せよ.

Exercise 6.5 preord と同様な方法で, 通りがけ順, 帰りがけ順に列挙する関数 inord, postord を定義せよ.

Exercise 6.6 二分木の左右を反転させた木を返す関数 reflect を定義せよ.

```
# reflect comptree;;
- : int tree =
Br
 (1, Br (3, Br (7, Lf, Lf), Br (6, Lf, Lf)),
  Br (2, Br (5, Lf, Lf), Br (4, Lf, Lf)))
```

また，二分木 t に対して，以下の方程式を完成させよ.

$$\texttt{preorder(reflect(t))} \;=\; ?$$

$$\texttt{inorder(reflect(t))} \;=\; ?$$

$$\texttt{postorder(reflect(t))} \;=\; ?$$

Exercise 6.7 以下は，足し算と掛け算からなる数式の構文を表した型定義である.

```
# type arith =
#   Const of int | Add of arith * arith | Mul of arith * arith;;
type arith = Const of int | Add of arith * arith | Mul of arith * arith
# (* exp stands for (3+4) * (2+5) *)
# let exp = Mul (Add (Const 3, Const 4), Add (Const 2, Const 5));;
val exp : arith = Mul (Add (Const 3, Const 4), Add (Const 2, Const 5))
```

数式の文字列表現を求める関数 string_of_arith, 分配則を用いて数式を $(i_{11} \times \cdots \times i_{1n_1}) + \cdots + (i_{m1} \times \cdots \times i_{mn_m})$ の形に変形する関数 expand を定義せよ.

```
# string_of_arith exp;;
- : string = "((3+4)*(2+5))"
# string_of_arith (expand exp);;
- : string = "(((3*2)+(3*5))+((4*2)+(4*5)))"
```

Exercise 6.8 1, 2, 3, 4 からなる可能な二分探索木の形を列挙し，それぞれの形を作るためには空の木から始めて，どの順番で要素を add していけばよいか示せ.

Exercise 6.9 関数 sift を定義し，⟨ 自分の学生証番号 ⟩ 番目の素数を求めよ.