2021年度「プログラミング言語」配布資料 (11)

五十嵐 淳

2022年10月02日

1　多相的2分探索木

本章では配布資料 (9) と配布資料 (10) で学んだことを総合して，多相的な 2 分探索木の実装を示す．さらに，map や fold といった操作を多相的な 2 分木に加える際の新しい課題をみる．

1.1　要素の比較操作の表現

既に配布資料 (9) 冒頭でも述べたように，2 分探索木に格納できるデータには全順序が与えられていて大小比較ができることが必要である．実際，整数の 2 分探索木においては随所で比較演算子== (OCaml なら =)，や比較演算子 < が使われている．こういった比較演算子がどんな種類の値の比較に適用できるかは言語によって様々であるが，比較できるものが限られているか，言語がデフォルトで用意した比較方式に従うため (特に，プログラマが定義したデータについては) 必ずしもプログラマの思う順序と合致しないこともある．そのため，比較演算はプログラマが何らかの形で 2 分探索木に与えることが必要となる．これは，高階関数が使える言語であれば比較演算をパラメータ化することで問題なく実現できるだろう．

必要なのは「等しさ」と「小さいかどうか」を判定する操作だが，このような「比較演算でパラメータ化されている処理」は，「以下」「未満」などの比較演算を全て用意するのではなく，以下のような関数 (仮に名前を $compare$ とする) をひとつだけ考えることが多い．

- $compare(x, y)$ は整数を返す．
- 「x が y より小さい」ならば $compare(x, y)$ は何らかの負整数を返す (つまり $compare(x, y) < 0$)．
- 「x が y と等しい」ならば $compare(x, y)$ は 0 を返す (つまり $compare(x, y) = 0$)．
- 「x が y より大きい」ならば $compare(x, y)$ は何らかの正整数を返す (つまり $compare(x, y) > 0$)．

括弧書き内にあるように $compare$ の結果は通常 0 と比較 (ここで使う不等号の向きは x, y の大小関係と一致している) することになる．

比較をこのように整数を返す関数ひとつで表現する技法は，言語を問わず，幅広く使われており，以下で見るように，データ型によっては，この仕様に沿った比較関数が最初から提供されていることがある．

1.2　OCaml の場合 (ocaml/polyBST)

まずは OCaml で多相的 2 分探索木を記述してみよう．木構造自体は，単なる 2 分木であるから既に見た通りである．

```
type 'elm tree =
    Lf       (* Leaf *)
  | Br of {  (* Branch *)
      left:  'elm tree;
      value: 'elm;
      right: 'elm tree;
    }
```

find などの関数は木や探索・追加・削除対象のデータだけでなく，比較関数もパラメータ (名前を cmp とする) として取ることになる．代表例として find を見てみよう．

```
(* (Recursive) function find, which returns whether given integer n exists in BST t *)
let rec find cmp t n =
  match t with
    Lf -> false
  | Br {left=l; value=v; right=r} ->
     if cmp n v = 0 then true
     else if cmp n v < 0 then find cmp l n
     else (* n > v *) find cmp r n
```

整数の場合と比べると，n = v が cmp n v = 0 に，n < v が cmp n v < 0 に置き換わっただけである．insert や delete も比較部分が変わるだけで特に説明するところはない．これらの操作が，比較以外，格納されるデータの種類に依存していないことの現れである．

ただし，min だけがもう少し説明が必要である．それは，引数 t が (想定していない入力である) leaf であった場合にダミーの値 min_int を返していた処理についてである．min は 2 分探索木中の最小の **要素** を返すわけで，これが min_int でよかったのは，整数が要素だったからである．今や，t に格納されるデータの型は抽象的な型パラメータになってしまって正体がわからないので，min_int はもちろんのこと，何も返せる値がない．

このような場合に対処するのが **例外 (exception)** の機構である．例外は，その名の通りプログラム実行中に発生する例外的な状況である．プログラマは例外機構を使って，例外的な状況でのプログラム実行の中断や，中断からの回復処理を記述することができる．例外機構の本格的な紹介はここでは行わないが，OCaml では，いくつか例外によってプログラムの実行を中断 (回復処理をしないので，実質的には強制終了) させるための関数が用意されている．invalid_arg〈文字列〉という，引数が不正であることを示す Invalid_argument 例外を発生させる関数があるのでそれを使うことにしよう．

```
(* Function min, which, given BST t, returns the minimum value stored in t.
   If t is empty, it fails. *)
let rec min t =
  match t with
    Lf -> invalid_arg "Input can't be a leaf!"
  | Br {left=Lf; value=v; right=_} -> v
  | Br {left=l;  value=_; right=_} -> min l
```

invalid_arg 関数の型は string -> 'a となっていて，文字列を与えると任意の型の (!) 式として使えることを表している．これは不思議に思えるかもしれないが，ここが実行されるとプログラムが強制終了してしまい

値が返ってこないことと対応しているのである．実際，min に Lf を与えると

```
# min Lf;;
Exception: Invalid_argument "Input can't be a leaf!".
#
```

と，値が返らずに，発生した例外がメッセージとして表示されて，入力プロンプトに戻ってくる．

さて，このように定義した関数の型を見てみよう．find，insert，delete の型は以下のように推論される．

```
val find : ('a -> 'b -> int) -> 'b tree -> 'a -> bool = <fun>
val insert : ('a -> 'a -> int) -> 'a tree -> 'a -> 'a tree = <fun>
val delete : ('a -> 'a -> int) -> 'a tree -> 'a -> 'a tree = <fun>
```

insert と delete の ('a -> 'a -> int) が cmp の型，'a tree が t の型，'a が n の型に対応している．返値は新しい ('a を要素とした) 木なので 'a tree である．一方，find は，ちょっと違う型が推論されている．ふたつの型変数'a と 'b が現れるので，cmp は，int を返すならどんな (カリー化された)2 引数関数でもよいことになっている．よくよく find の定義を見てみると，(cmp に与えられる実際の関数は int -> int -> int など 2 引数の型が一致しているものではあるものの，) 確かに n と t の要素型が一致している **必要はない** ことがわかる．前にも述べたように，OCaml の型推論は最も多相的な型 (主要型) を推論するので，この場合のように，想定していたよりも多相的な型が推論される場合もある．

ちなみに，insert については，n が Br {...; value = n; ...} と使われているので，n の型と t の要素型が一致している必要がある．また delete については，min で得られた m を使って delete を呼び出すことから，芋蔓式に n も m と型が一致していることが要請されて，上のような型が推論される．

さて，最後に，これらの使用例として，整数と文字列の 2 分探索木を示す．整数の比較は単に差を取ればよいので，cmp として fun x y -> x - y という匿名関数を与えている．(何度も使うので，cmp を fun x y -> x - y に固定した関数を定義している．) 文字列については，OCaml のライブラリが String.compare という上の *compare* の仕様を満たした関数を提供してくれている．

```
(* Constructing a sample tree *)
let t1 = Br {left = Lf; value = 10; right = Lf}
let t2 = Br {left = Lf; value = 25; right = Lf}
let t3 = Br {left = t1; value = 15; right = t2}
let t4 = Br {left = Lf; value = 60; right = Lf}
let t5 = Br {left = Lf; value = 48; right = t4}
let t6 = Br {left = t3; value = 30; right = t5}

let find_int t i = find (fun x y -> x - y) t i
let insert_int t i = insert (fun x y -> x - y) t i
let delete_int t i = delete (fun x y -> x - y) t i

(* Testing find *)
let test1 = find_int t6 30  (* should be true *)
let test2 = find_int t6 13  (* should be false *)
```

```
(* Testing insert *)
let t7 = insert_int t6 23
let t8 = insert_int t6 0
let test3 = find_int t7 23  (* should return true *)
let test4 = find_int t8 30  (* should return false *)
let test5 = find_int t8 23  (* should return false *)

(* Testing delete *)
let t9 = delete_int t8 30
let test6 = find_int t9 30
let test7 = find_int t9 48

(* Constructing another sample tree *)
let t11 = Br {left = Lf; value = "I"; right = Lf}
let t12 = Br {left = Lf; value = "love"; right = Lf}
let t13 = Br {left = t11; value = "OCaml"; right = t12}
let t14 = Br {left = Lf; value = "you"; right = Lf}
let t15 = Br {left = Lf; value = "think"; right = t14}
let t16 = Br {left = t13; value = "so?"; right = t15}

let find_str t i = find String.compare t i
let insert_str t i = insert String.compare t i
let delete_str t i = delete String.compare t i

(* Testing find *)
let test11 = find_str t16 "so?"  (* should be true *)
let test12 = find_str t16 "Ocaml"  (* should be false *)

(* Testing insert *)
let t17 = insert_str t16 "Me"
let t18 = insert_str t16 "too"
let test13 = find_str t17 "Me"  (* should return true *)
let test14 = find_str t18 "Why"  (* should return false *)
let test15 = find_str t18 "Me"  (* should return false *)

(* Testing delete *)
let t19 = delete_str t18 "Why"
let test16 = find_str t19 "Why" (* should return false *)
let test17 = find_str t19 "She" (* should return false *)
```

1.3 Java の場合 (java/polyBST)

1.3.1 関数オブジェクトを用いた手法

Java でも OCaml と同様なアイデアを用いれば多相的な 2 分探索木のインターフェース・クラスを記述することができる．比較関数の返り値型を (Integer ではなく) プリミティブ型の int にする場合，ライブラリに「T 型と U 型から int 型への関数」を表すインターフェース ToIntBiFunction があるので，これを使うと，ジェネリック・インターフェース BinarySearchTree<Elm> は以下のように記述できるだろう．

```
import java.util.function.*;  // for ToIntBiFunction

public interface BinarySearchTree<Elm> {
    boolean find(Elm e, ToIntBiFunction<Elm,Elm> compare);
    ...
}
```

find の型のみ示しているが，検索する要素だけでなく，比較関数を表す引数を追加している．ToIntBiFunction は java.util.function というパッケージ (ライブラリ) に属しているので，上のような import 宣言をしておくとよい[1]．ToIntBiFunction がどのようなインターフェースかは以下に示す．関数を呼ぶためのメソッド名が，返値型を含んだ applyAsInt になっていることに注意してほしい．

```
// ライブラリ java.util.function にあるインターフェース
public interface ToIntBiFunction<T,U> {
    int applyAsInt(T t, U u);
}
```

実は，Java には，比較関数を表すための特別なジェネリック・インターフェース Comparator<T> が用意されている[2]ので，これを使う方がより Java らしい．

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    ...
}

// import は特に要らない
public interface BinarySearchTree<Elm> {
    boolean find(Elm e, Comparator<Elm> c);
    ...
}
```

Comparator を使った場合，find などのメソッド定義はほとんど OCaml と同様になることもあり，ここでは，この方法にはこれ以上踏み込まない．

[1] この宣言は必須ではないが，その場合，java.util.function.ToIntBiFunction という名前で参照する必要がでてくる．

[2] このインターフェースもメソッドは compare ひとつ (正確にはオブジェクトが持つべきインスタンス・メソッドがひとつ．他に static メソッドと呼ばれるものが定義されている) なので，ラムダ式を使うことができる．

1.3.2 メソッドとしての比較処理と制限付型パラメータ

ここで詳しく紹介するのは，比較の処理を Comparator を implements した関数オブジェクトではなく，2 分探索木に格納されるオブジェクトのメソッドとして定義する方法である．実は，Java ライブラリにあるクラスの多くには compareTo という int を返すメソッドが実装されていて，これによって比較を行うことができる．

```
Integer i1 = new Integer(10);
Integer i2 = new Integer(20);
System.out.println(i1.compareTo(i2));  // displays some negative int

String s1 = "hoge";
String s2 = "fuga";
System.out.print(s1.compareTo(s2));  // displays some (perhaps positive) int
```

比較には，このメソッドを使うことにすれば，find などの 2 分探索木を操作するメソッドに比較関数オブジェクトの引数を追加することなくクラスを記述することができる．

1.3.2.1 バージョン 1 この方針に従って書いたインターフェース・クラスが以下である．ここでは find のみ定義を示している．

```
public interface BinarySearchTree<Elm> {
    ...
    boolean find(Elm e);
    ...
}

public class Leaf<Elm> implements BinarySearchTree<Elm> {

    ...

    public boolean find(Elm e) {
       return false;
    }

    ...
}

public class Branch<Elm> implements BinarySearchTree<Elm> {
    ...

    public boolean find(Elm e) {
        if (e.compareTo(v) == 0) { return true; }
        else if (e.compareTo(v) < 0) { return left.find(e); }
        else /* e > v */ { return right.find(e); }
    }
```

```
    ...
}
```

比較のために e の compareTo メソッドが呼ばれていることに注意せよ．しかし，残念ながらこのクラス定義は (Branch クラスの) コンパイル時に以下のようなエラーが発生する．

```
Branch.java:45: エラー: シンボルを見つけられません
        if (e.compareTo(v) == 0) { return true; }
             ^
  シンボル:   メソッド compareTo(Elm)
  場所: タイプ Elm の変数 e
  Elm が型変数の場合:
    クラス Branch で宣言されている Elm は Object を拡張します
```

他のエラーも compareTo を呼出すところで発生しているようだ．これは一体どういうことだろうか．このエラーは，e の型が (型変数) Elm なのだけれども，Elm 型のオブジェクトに compareTo メソッドがない (かもよ)，といっているのである．今，Elm は任意のオブジェクト型 (これがメッセージ中の「Object を拡張します」のだいたいの意味である) を表しうる—使う側は任意のオブジェクト型を Elm に割り当てて使う可能性がある—のだが，compareTo メソッドは全てのオブジェクトが持っているわけではないのでまずいのである．配布資料 (9) で 2 分木の toString メソッドを定義した時，Elm 型の変数に対して toString メソッドを (暗黙に) 呼んでいたが，あれは，**型に関わらず全てのオブジェクトは toString を持っている**おかげで呼んでよかったのである．

1.3.3 バージョン 2

このような状況に対処するために，Java では型変数が表す型の範囲を，インターフェースを使って，特定のメソッドを持つクラスに制限することができる．この時に鍵になるのは，インターフェース I を implements したクラスのオブジェクトは全て I に書かれたメソッドを持つ，という性質である．具体的には，型変数 X に extends I という修飾をつけ public class C<X extends I> {...} のように書いて宣言する[3]．すると，* C を使う箇所では X を I を implements したクラスでしか具体化できなくなる，が，同時に * X 型の変数に対して，I に書かれたメソッドが C 内で呼べるようになる．

具体例を見てみよう．まず，Java では Integer や String などの比較メソッド compareTo を持つクラスは Comparable<T> というジェネリック・インターフェースを implements している．Comparable<T> は以下のように定義されている．

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

ここで compareTo の引数の型が，具体的な型ではなく型変数になっているのは，オブジェクト毎に何と比較できるかが異なるためである．実際，Integer の compareTo メソッドは Integer を引数にとるし，String の compareTo メソッドは String を引数にとるので，Integer の場合 T を Integer にしなければいけないし，

[3]意味的には「implements していること」という条件なのに extends というキーワードを使うのには，いくつか事情があるがここでは説明しない．

String の場合 T を String にしなければならない．以下は，いくらか単純化した，Integer と String クラスの定義である．

```
public class Integer implements Comparable<Integer> {
    ...
    int compareTo(Integer i) {
        ....
    };
    ...
}

public class String implements Comparable<String> {
    ...
    int compareTo(String i) {
        ....
    };
}
```

つまり，比較メソッドを提供するクラス C は

```
public class C implements Comparable<C> {
```

というヘッダで定義がされるのである．

これに対応して「型変数 Elm は compareTo メソッドを持つ」ということを表現するためには，

```
public interface BinarySearchTree<Elm extends Comparable<Elm>> {
    ...
}

public class Leaf<Elm extends Comparable<Elm>> implements BinarySearchTree<Elm> {
    ...
}

public class Branch<Elm extends Comparable<Elm>> implements BinarySearchTree<Elm> {
    ...
}
```

という定義をする．extends Comparable<Elm> の部分が追加された部分で，いわば int compareTo(Elm o); というメソッドを持つ，ということを表現している[4]．このおかげで，e.compareTo(v) というメソッド呼出しが適正なものと認識されるようになる (e と v の型はともに Elm であることに注意せよ．)

このような「型変数の動く範囲」の指定は，しばしば，**境界 (bound)** と呼ばれる．また，bound を指定することができるような多相性の概念の拡張を bounded polymorphism (もしくは constrained polymorphism) と

[4] これは実は少し不正確である．実際には，Comparable インターフェースを implements しなくても compareTo メソッドを持つクラスを定義することができるので，Comparable を implements しているならば，compareTo を持つ，は正しいが，その逆 (compareTo があるクラスは Comparable を implements している) は必ずしも成り立たない．ただし，Java ライブラリで提供しているクラスについては，比較メソッドを持つならば Comparable を implements するように設計されている．

呼ぶ.

1.3.4 min の実装 (例外を投げる)

OCaml の min の実装を変更する必要があったのと同様，Java でも Elm を返り値とするメソッドで return Integer.MIN_VALUE; ができるわけではないので，例外を発生させて実行中断をする.

```
public Elm min() {
    throw new UnsupportedOperationException();
}
```

(今回の講義では例外について説明する時間が全く取れなかったので，上はおまじないだと思ってください．)

1.3.5 使用例

ここまでできてしまえば，使用例は，2 分木の時と大して変わらない.

```
// Main クラスより
BinarySearchTree<Integer> t1 =
    new Branch<Integer>(new Leaf<Integer>(), 10, new Leaf<Integer>());
 BinarySearchTree<Integer> t2 =
    new Branch<Integer>(new Leaf<Integer>(), 25, new Leaf<Integer>());
 BinarySearchTree<Integer> t3 = new Branch<Integer>(t1, 15, t2);
 BinarySearchTree<Integer> t4 =
    new Branch<Integer>(new Leaf<Integer>(), 60, new Leaf<Integer>());
 BinarySearchTree<Integer> t5 =
    new Branch<Integer>(new Leaf<Integer>(), 48, t4);
 BinarySearchTree<Integer> t6 = new Branch<Integer>(t3, 30, t5);
 boolean test1 = t6.find(30);  // should be true
 boolean test2 = t6.find(13);  // should be false
 BinarySearchTree<Integer> t7 = t6.insert(23);
 BinarySearchTree<Integer> t8 = t6.insert(0);
 boolean test3 = t7.find(23);  // should be true
 boolean test4 = t8.find(30);  // should be true
 boolean test5 = t8.find(23);  // should be false
 BinarySearchTree<Integer> t9 = t8.delete(30);
 boolean test6 = t9.find(30);  // should be false
 boolean test7 = t9.find(48);  // should be true

 BinarySearchTree<String> t11 =
    new Branch<String>(new Leaf<String>(), "I", new Leaf<String>());
 BinarySearchTree<String> t12 =
    new Branch<String>(new Leaf<String>(), "love", new Leaf<String>());
 BinarySearchTree<String> t13 = new Branch<String>(t11, "OCaml", t12);
```

```
BinarySearchTree<String> t14 =
    new Branch<String>(new Leaf<String>(), "you", new Leaf<String>());
BinarySearchTree<String> t15 =
    new Branch<String>(new Leaf<String>(), "think", t14);
BinarySearchTree<String> t16 = new Branch<String>(t13, "so?", t15);
boolean test11 = t16.find("so?");  // should be true
boolean test12 = t16.find("Ocaml");  // should be false
BinarySearchTree<String> t17 = t16.insert("Me");
BinarySearchTree<String> t18 = t16.insert("too");
boolean test13 = t17.find("Me");  // should be true
boolean test14 = t18.find("so?");  // should be true
boolean test15 = t18.find("Why");  // should be false
BinarySearchTree<String> t19 = t18.delete("Why");
boolean test16 = t19.find("Why");  // should be false
boolean test17 = t19.find("you");  // should be true
```

上で述べたように，Integer, String ともに Comparable インターフェースを実装 (implements) しているため，BinarySearchTree, Leaf, Branch の型引数として適当だが，Comparable を実装していないクラスを書くことはできない．例えば，

```
BinarySearchTree<Object> t;
```

という変数宣言をすると，以下のような「型引数が境界内にない」というエラーになる．

```
Main.java:9: エラー: 型引数 Object は型変数 Elm の境界内にありません
    BinarySearchTree<Object> t;
                     ^
  Elm が型変数の場合:
    インタフェース BinarySearchTree で宣言されている Elm は Comparable<Elm>を拡張します
エラー 1 個
```

1.3.6 Comparable vs Comparator


いつか書きたい

1.4 C の場合 (C/polyBST)

いつか書きたい

1.5 多相性 + 高階関数

多相的 2 分木上で map や fold を定義してみよう．基本的な計算の方法は多相的であろうとなかろうと変わらない．主な興味の対象は，これらの型がどのように表現されるか，というところである．

1.5.1 OCaml の場合 (polyTree/tree.ml)

まず，OCaml の場合，treemap と treefold の定義は配布資料 (10) で示したものと 全く変わらない．

```
type 'elm tree =
    Lf       (* Leaf *)
  | Br of {  (* Branch *)
      left:  'elm tree;
      value: 'elm;
      right: 'elm tree;
    }

let rec treemap f t =
  match t with
    Lf -> Lf
  | Br {left=l; value=v; right=r} ->
     Br {left = treemap f l;
         value = f v;
         right = treemap f r}

let rec treefold e f t =
  match t with
    Lf -> e
  | Br {left=l; value=v; right=r} ->
     f (treefold e f l)
       v
       (treefold e f r)
```

このふたつの関数に対して，型推論によって，以下のような型が推論される．

```
val treemap : ('a -> 'b) -> 'a tree -> 'b tree
val treefold : 'a -> ('a -> 'b -> 'a -> 'a) -> 'b tree -> 'a
```

treemap の型が我々に教えてくれるのは，木に格納されたデータを変換するための関数の型は'a -> 'b となっていて，引数と返値の型は同じでなくてもよい，ということである．渡した関数の型に応じて入力と出力の木の型が決まる．例えば，以下のように，整数の 2 分木から文字列の 2 分木へ，または，その逆の変換に使うことができる．

```
let t26 = treemap (fun i -> string_of_int i ^ " yen") t6
let t36 = treemap String.length t16
```

treefold の 'a は畳み込みの最終結果の型，'b は 2 分木に格納されたデータの型を表す．関数引数 f の型 ('a -> 'b -> 'a -> 'a) は，f が左の部分木を再帰的に畳み込んだ結果 ('a) と branch のデータ ('b) と右の部分木を再帰的に畳み込んだ結果 ('a) に適用されることを表している．例えば，以下のような呼出しで，文字列の 2 分木から，全文字列の長さの合計を計算することができる．(ここで'a と 'b は何にあたるか考えてみよ．)

```
let s = treefold 0 (fun l v r -> l + String.length v + r) t16
```

1.5.2 Java の場合 (java/polyTree)

Java の場合もメソッドの定義の本体は，配布資料 (10) で示したものと (型宣言を除いて) ほとんど変わらないが，実は，OCaml と同レベルの柔軟性 (多相性) を得るには新しい仕組みが必要である．まずは map について紹介する．

1.5.2.1 map メソッド，バージョン 1 まず，以下は素朴に多相的 2 分木のインターフェースに map を追加したものである．f の型で少し悩むが，使えそうな型が Elm くらいしかないので，Function<Elm,Elm> とした．(Function<T,R> は java.util.function で定義されているジェネリック・インターフェースで T から R への関数 (オブジェクト) を表す．)

```
public interface Tree<Elm> {
    ...
    Tree<Elm> map(Function<Elm,Elm> f);
    ...
}
```

しかし，これでは，同じ型の木への変換しかできず，上の OCaml の例でみたような，整数の 2 分木から文字列の 2 分木へといった変換ができない．

1.5.2.2 map メソッド，バージョン 2 バージョン 1 の欠点を修正しようとしたのが次のバージョン 2 である．OCaml の treemap の型スキームに 'a (変換元の 2 分木の要素型) と 'b (変換先の 2 分木の要素型) が現れていたのと同様に，クラスをふたつの型変数でパラメータ化してみた．

```
public interface Tree<Elm,Elm2> {
    ...
    Tree<Elm2> map(Function<Elm,Elm2> f);
    ...
}
```

これで，うまく行きそうに思えるが，実は不十分である．まず，map の返り値型の Tree の型引数の数が足りない．これを適当に決めたとして (Elm として，返り値型は Tree<Elm2,Elm> としておこう) も，まだ問題がある．このインターフェース (と対応する Leaf と Branch クラス) は，

```
Tree<Integer,String> t = new Branch<Integer,String>(new Leaf<Integer,String>(), 11, new Leaf<Integer,S
```

のように使うことになるが，オブジェクトを作った時点で map の変換先の要素型も指定しなければいけないので，t から map を使って得られる 2 分木は常に文字列の 2 分木になってしまう．しかも，map の返り値型を (いい加減に) Tree<Elm2,Elm> と定義したので，t.map(...) の型は Tree<String,Integer> つまり，もう一度 map する場合の変換先が今度は整数の 2 分木に固定されてしまう．

我々は，要素変換関数に応じて，変換後の 2 分木の要素型を決められるようにしたい．クラス定義に型パラメータ Elm2 を追加するのでは，2 分木オブジェクトを作る時に map 時の変換先の型をひとつに定めなければならないわけで，これでは早すぎるのである．

1.5.2.3 map メソッド，バージョン 3(完成版) Java には，ジェネリック・クラス，ジェネリック・インターフェースだけではなく，**ジェネリック・メソッド (generic methods)** といって，メソッドに型引数を指

定することができる．これを map の定義に使えば，map の呼出し毎に変換後の 2 分木の要素型を指定することができる．ジェネリック・メソッドを使った Tree の定義が以下である．

```
public interface Tree<Elm> {
    ...
    <Elm2> Tree<Elm2> map(Function<Elm,Elm2> f);
    ...
}
```

このように，ジェネリック・メソッドはメソッドの先頭 (public の後) に型変数の宣言 (ここでは<Elm2>) をつける．これをつけると，そのメソッドの返り値型，引数型，本体の中では，宣言された型変数 (Elm2) をふつうの型と同じように使うことができる[5]．以下が Leaf, Branch クラスの map の定義である．

```
    // Leaf クラスに追加
    public <Elm2> Tree<Elm2> map(Function<Elm,Elm2> f) {
        return new Leaf<Elm2>();
    }

    // Branch クラスに追加
    public <Elm2> Tree<Elm2> map(Function<Elm,Elm2> f) {
        Tree<Elm2> newLeft = left.map(f);
        Tree<Elm2> newRight = right.map(f);
        Elm2 newVal = f.apply(v);
        return new Branch<Elm2>(newLeft, newVal, newRight);
    }
```

両メソッド内で 2 分木を作る時には Leaf<Elm2>, Branch<Elm2> と，ジェネリック・メソッドの型変数 Elm2 を使って作っていることに注目しよう．

ジェネリック・メソッドを呼出す時は，以下のように，ドットとメソッド名の間に型引数を指定することができるが，多くの場合，省略することができる．これも一種の型推論である．ここでは，文字列の 2 分木から，各文字列の長さを格納した整数の 2 分木を得ている．Elm2 として結果の 2 分木の要素型 Integer を指定し，Function<String,Integer> 型のラムダ式を渡している．

```
        Tree<Integer> t9 = t18.<Integer>map(s -> s.length());
        // "<Integer>" before map specifies what Elm2 is in this invocation
        // It can be omitted and written t18.map(s -> s.length())
        // This is another form of type inference in Java!
```

1.5.2.4 fold メソッド fold も基本的なアイデアは同じである．fold の結果の型は，木が格納している要素の型 Elm とは独立に呼出し毎に指定したいので，(型パラメータ Result をとる) ジェネリック・メソッドとして定義する．この時，fold に与える引数を考えると，まず Leaf の場合を表す引数は Result 型になる．そして，Branch の場合に使われる関数引数は Result と Elm と Result から Result を返すものになる．3 引数関数オブジェクトのためのインターフェースはライブラリに用意されていないので，以下のように自分で定義をする．

[5] ジェネリック・クラスがそうであったように，なぜメソッドの名前の直後に型変数宣言が来ないのか不思議に思うかもしれない．これは返り値型に型変数が含まれる場合に，型変数の宣言に先行して型変数の使用箇所が現れるとおかしく見えるためだと思われる．

```
// 3 引数関数オブジェクトのためのインターフェース
public interface TriFunction<T1,T2,T3,R> { // T1 * T2 * T3 -> R 相当
    R apply(T1 x, T2 y, T3 z);
}
```

この上で，fold メソッドは以下のように定義できる．

```
    // Leaf クラスに追加
    public <Result> Result fold(Result e, TriFunction<Result,Elm,Result,Result> f) {
        return e;
    }

    // Branch クラスに追加
    public <Result> Result fold(Result e, TriFunction<Result,Elm,Result,Result> f) {
        Result resl = left.fold(e,f);
        Result resr = right.fold(e,f);
        return f.apply(resl,v,resr);
    }
```

例えば，Tree<Integer> に対して，格納されている整数の和を計算するには，

```
TriFunction<Integer,Integer,Integer,Integer> f = (n, m, p) -> n + m + p;
System.out.println(t6.fold(0, f));
```

とする．fold の型パラメータ Result に対して渡される型は fold した結果の Integer であるが，Java が推論してくれるので書く必要はない．また，Tree<String> に対して，格納されている文字列の長さの和を計算したければ，

```
TriFunction<Integer,String,Integer,Integer> f = (l, v, r) -> l + v.length() + r;
// Computes the sum of the lengths of the strings in t18
Integer i = t18.fold(0, f);
```

とする．