2021年度「プログラミング言語」配布資料 (10)

五十嵐 淳

2022年10月02日

1 高階関数

$\sum_{i=n}^{m} x_i$ という記法は (実) 数列 $\{x_i\}$ の第 n 項から第 m 項までの和を表す．この $\sum$ をひとつの演算子だと捉えようとすると，これは自然数 n, m と数列 $\{x_i\}$ に作用する演算子，もしくは，関数と考えることができる．数列を自然数 i から実数への関数と考えれば，$\sum$ は**関数を引数とする関数**である．プログラミングにおいて，「関数を引数とする関数」が実現できたり，関数を変数に格納することができたりと，関数が (整数や文字列などの通常の) データと同様に[1]扱えることを指して「関数が**第一級の値 (first-class value)** である」といったりする．

また，概念的には「『関数を引数とする関数』を引数とする関数」などのより複雑な関数が考えられる．この「複雑さ」に特に制限が設けられていない時，プログラミング言語が**高階関数 (higher-order function)** をサポートしている，高階プログラミングができる，などという．

この「階 (order)」とは，関数の「複雑さ」を示す度合いで，* 一階の関数 (first-order functions) …整数，文字列などの基本的データ，基本的データのみで構成されるレコードなどを引数とする関数* 二階の関数 (second-order functions) …一階の関数を引数とする関数 ($\sum$ など) * n 階の関数 (n-th-order functions) … $(n-1)$ 階の関数を引数とする関数

と定義される．

関数が第一級の値であることは「データの操作」の受け渡しが許されている，ということとも考えられる．これはオブジェクトのような「データとその操作をいっしょにしたもの」が受け渡しできるオブジェクト指向言語では，ある意味当たり前にできていることである．そのため，明示的に関数の概念が現れないオブジェクト指向言語でも，実質的には高階関数が利用できると考えられる．

1.1 高階関数 in OCaml (ocaml/hofun)

OCaml も含め，一般に関数型言語は高階関数が書きやすく設計されている．

1.1.1 関数を引数とする関数

最初の例である $\sum$ を考えてみる．(ただし，煩雑にならないよう，最初の項は第 0 項で固定とする．) 関数 `sigma` は，整数上の関数 `f` と，自然数 `n` を引数として，`f(0) + f(1) + ... + f(n-1) + f(n)` を計算する．

[1]「同様に」の意味は厳密ではないが，通常，変数に格納したり，他の関数・手続・メソッドに受け渡しできる，実行時に生成できたりすることを指す．

以下が sigma の定義である．

```
let rec sigma(f, n) =
  if n < 1 then f 0
  else sigma(f, n-1) + f n
```

f は，n と同様に，特に特別な記法もなくふつうにパラメータとして書けばよい．関数本体では，f 0 のように，let で定義された関数と同様に使われている．sigma の型は例によって型推論機能が答えてくれる．

```
val sigma : (int -> int) * int -> int = <fun>
```

この型は，第 1 引数が int -> int 型の値，すなわち，整数から整数への関数であることが示されている．-> は * よりも結合が弱く，また右結合するので，関数型 int -> int を括る () が必要となる．(この括弧がないと，int -> ((int * int) -> int) と結合する．このような型の読み方・意味は後でふれることにする．)

この関数を呼ぶには，引数に定義済の関数の名前を与えてやればよい．

```
let square n = n * n
let cube n = n * n * n


let a = sigma(square, 20)
let b = sigma(cube, 20)
```

1.1.2 匿名関数

関数が第一級の値であるような言語の多くでは，名前をつけずに関数を表す記法が用意されている．このような名前のない関数を**匿名関数 (anonymous function)** と呼ぶ．OCaml では匿名関数は fun〈パラメータ〉->〈式〉という形で表す．以下が，上の a b の定義を匿名関数を使って書いたものである．

```
(* anonymous functions *)
let c = sigma ((fun n -> n*n), 20)
let d = sigma ((fun n -> n*n*n), 20)
```

fun n -> n*n で「n を受け取って n * n を返す関数」を意味する．

fun はできるだけ伸びるので，fun の範囲を示すためにしばしば括弧が必要となる．上の例で括弧を省略すると-> の後が n * n , 20 ということになって，n * n と 20 の組を返す関数になってしまう．(組のことはほとんど説明していない．あしからず．)

この記法を使うと，cube を以下のように定義することもできる．

```
let cube = fun n -> n * n * n
```

驚くなかれ，実は，let f x = ... という記法は let f = fun x -> ... の略記なのである．

複数の引数を受け取る匿名関数も fun (x, y, z) -> ... のような形式で定義することができる．

```
fun (x, y) -> (x +. y) /. 2.0;;
```

1.1.3 カリー化による複数引数関数の表現

これまで複数の引数をとる関数は `let f(x,y) = ...` のように，複数の引数を組にしてとるように書いてきた．実は，高階関数，特に，関数を返す関数を使うと，複数引数関数を，1 引数関数の組み合わせで表現することができる．このテクニックを**カリー化 (Currying)** (アメリカの数学者・論理学者ハスケル・カリーにちなんでいる[2][3])，またそのような方法を使って表現された複数引数関数を**カリー化関数 (Curried function)** と呼ぶ．

例を見てみよう，以下の `greeting` は，性別 `gen` と名前を表す文字列 `name` を受け取って，挨拶文 (敬称が性別で違う) を返す簡単な関数である．[4]

```
type gender = Male | Female

let greeting(gen, name) =
  match gen with
    Male ->   "Hello, Mr. " ^ name
  | Female -> "Hello, Ms. " ^ name

let g1 = greeting(Male,   "Poirot")
let g2 = greeting(Female, "Marple")
```

カリー化のアイデアは 2 引数関数を，「最初の引数を受け取って，『第 2 の引数を受け取って結果を返す関数』を返す関数」として表現する，ということである．一般に，n 引数関数なら「最初の引数を受け取って『残りの引数を受け取って結果を返す $n-1$ 引数関数』を返す関数」と表現される．以下はカリー化された `greeting` である．

```
let curried_greeting gen = fun name ->
  match gen with
    Male ->   "Hello, Mr. " ^ name
  | Female -> "Hello, Ms. " ^ name
```

この関数の型は以下のようになる:

```
val : curried_greeting : gender -> string -> string = <fun>
```

この型は `gender -> (string -> string)` と同じ (既に述べたように`->`は右結合) で，`gender` を受け取ると，`string -> string` を返す関数であることが型から読み取れる．

カリー化された関数を呼ぶ時は，まず，性別を与えると，(名前を受け取って挨拶文を返す) 関数が返ってくるので，それを呼ぶ，という手順になる．

```
let greeting_for_men = curried_greeting Male
let greeting_for_women = curried_greeting Female

let g1 = greeting_for_men "Poirot"
```

[2] ファーストネームもラストネームもプログラミング言語の名前になっている

[3] カリー化はその名前に反して，どうも組合せ論理という計算体系を発明したロシアの論理学者・数学者のモーゼス・シェーンフィンケルや現代論理学の始祖であるドイツの哲学者・論理学者ゴットロープ・フレーゲに帰すべきものらしい．

[4] この例だと“Mr.”よりも“Monsieur”の方がよかったですかね…

```
let g2 = greeting_for_women "Marple"
```

curried_greeting x は，関数 fun name -> ... を返すわけだが，この時 gen の値が，渡した値 (Male もしくは Female) に固定されたような関数が返ってきている，ということに注目してもらいたい．だからこそ，次の引数である文字列が渡された時に，前に渡した引数によって異なる挙動を示している．

一般に，(匿名) 関数本体では，パラメータ以外の変数が現れるわけだが，その値は，fun が評価された時 (fun に実行が到達した時) に固定される．関数が呼び出された時には，既に固定された値を使って，本体の計算が行われる．それゆえ，一般に関数を (コンパイラなどの言語処理系のレベルで) 表現するためには，関数のコード (fun 以下の情報) に加え，パラメータ以外の (固定された) 変数の値を記録しておく必要がある．それらをまとめたものを**関数閉包 (function closure)** と呼ぶ．これは言語処理系レベルで第一級関数を実現するための技法である．例えば，関数 greeting_for_men は，1. fun name -> match gen with ... と 1. gen の値は Male であるという情報の組で表される．そして，greeting_for_men "Poirot" と呼ばれた時点で，name を "Poirot" として match gen with 以下の式の値が計算されて"Hello, Mr. Poirot" が返されることになる．

1.1.3.1 curried_greeting や g1, g2 の別定義

cube の別定義でみたように，定義のための = の直後の fun x -> は = の左側に移せるので，curried_greeting も fun キーワードを使わずに以下のように定義することができる．

```
let curried_greeting gen name =
  match gen with
    Male ->   "Hello, Mr. " ^ name
  | Female -> "Hello, Ms. " ^ name
```

また，呼び出す際にも，最初の引数を渡したものに，いちいち名前をつけずに，以下のように呼び出すこともできる．

```
let g1 = (curried_greeting Male) "Poirot"
let g2 = curried_greeting Female "Marple"
```

しかも，g2 のように左側の括弧は省略できる (関数適用は左結合) ので，こう書くと，() を使わずに 2 引数関数が書けているように見えるだろう．

カリー化された複数引数関数は，全ての引数を一度に与えて呼び出すことが多いが，greeting_for_men のように，最初の方の一部の引数だけを与えて，引数の一部の値を固定した関数を作ることもしばしばある．このような一部の引数だけを渡すことを**部分適用 (partial application)** という．後の方の引数を固定するのは OCaml では少し面倒なので，カリー化を行う際には，固定されやすい引数を最初にもってくることが多い．

1.1.4 2 分木のための高階関数

いつもの (整数を格納した)2 分木を考える．

```
type tree =
  Lf
| Br of {
    left:  tree;
    value: int;
```

```
    right: 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}
```

データ構造に典型的ないくつかの高階関数を見ていこう．

1.1.4.1 map `map` は，データ構造に格納された各データ (この 2 分木の場合は branch の整数) に関数 (パラメータ) `f` を適用して得られる別の 2 分木を返す操作である．

```
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's double the values in t6 keeping its shape *)
let t7 = treemap (fun n -> n * 2) t6
```

1.1.4.2 fold `fold` は畳み込みとも呼ばれる．データ構造の各コンストラクタ C_i (n_i 引数) を n_i 引数の関数 f_i に置き換えて「解釈」し，値を得る関数である．(0 引数コンストラクタには定数を割り当てる．)

```
(* Replace Lf with e and Br with f *)
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)

(* Interpret Lf as 0 and Br as addition *)
let s6 = treefold 0 (fun l v r -> l + v + r) t6
let s7 = treefold 0 (fun l v r -> l + v + r) t7

(* Conversion to a string can be written in terms of fold *)
let str6 = treefold "leaf" (fun l v r -> "branch(" ^ l ^ ", " ^ (string_of_int v) ^ ", " ^ r ^ ")") t6
let str7 = treefold "leaf" (fun l v r -> "branch(" ^ l ^ ", " ^ (string_of_int v) ^ ", " ^ r ^ ")") t7
```

1.2 オブジェクトを使った高階関数プログラミング in Java (java/hofun)

1.2.1 オブジェクト≒関数のレコード

メソッドを関数と考えると，オブジェクトはメソッドを集めたレコードだと考えることができる．逆に，メソッドをひとつだけ持つオブジェクトを考えれば，第一級関数を表現することができる．

まず，関数の型毎に (メソッドがひとつだけの) インターフェースを用意する．インターフェースの名前は引数・返値の型を表す名前にしておく．メソッドの名前はなんでもよいが，ここでは Java のライブラリにある `Function` という (ジェネリック) インターフェースにあわせ `apply` とする．(`Function` などのライブラリのインターフェースについては後述する．)

```
public interface IntToInt {
    int apply(int x);
}
```

そして，このインターフェースを `implements` したクラスを作ることで関数を表現する．整数を 2 倍する関数であれば，以下のようになる．インスタンス変数は特に必要ない．

```
public class Dbl implements IntToInt {
    // no instance variables

    public Dbl() {
        // nothing to initialize
    }

    public int apply(int n) { return n * 2; }
}
```

例えば，この「関数」を作って，呼び出すには以下のようにすることになる．

```
IntToInt f = new Dbl();
int x = f.apply(3);   // x = 6
```

さて，この手法を使って，2 分木の `map` や `fold` をプログラムしてみよう．

1.2.1.1 map 今までの話を総合すれば特筆すべきことはない．まず，`Tree` インターフェースには `map` の宣言を追加する．引数が上で定義した整数上の関数を表すインターフェースになる．

```
public interface Tree {
    Tree map(IntToInt f);
    ...
}
```

そして，`Leaf` と `Branch` の各クラスに，処理を追加する．`Branch` では関数オブジェクト `f` を呼び出して新しい木に格納する整数を計算する．

```
    // Leaf クラスに追加
    public Tree map(IntToInt f) {
```

```
        return new Leaf();
    }

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

以下が，この map を使った例となる．

```
        // Main クラスより
        Tree t1 = new Branch(new Leaf(), 10, new Leaf());
        Tree t2 = new Branch(new Leaf(), 25, new Leaf());
        Tree t3 = new Branch(t1, 15, t2);
        Tree t4 = new Branch(new Leaf(), 60, new Leaf());
        Tree t5 = new Branch(new Leaf(), 48, t4);
        Tree t6 = new Branch(t3, 30, t5);

        System.out.println(t6);

        Tree t7 = t6.map(new Dbl());
        System.out.println(t7);
```

1.2.1.2 fold その 1 Br を置き換える 3 引数関数のためにインターフェースを新しく用意する必要があるが，その他は特筆すべきところはない．インターフェース ThreeIntsToInt が 3 つの整数から整数への関数を表すインターフェース，Sum3 が与えられた 3 整数の和を返す関数オブジェクトのクラスである．

```
public interface Tree {
    ...
    int fold(int e, ThreeIntsToInt f);
}

public interface ThreeIntsToInt {
    int apply(int l, int v, int r);
}

public class Sum3 implements ThreeIntsToInt {
    // no instance variables

    public Sum3() {
        // nothing to initialize
    }
```

```
    public int apply(int n, int m, int p) { return n + m + p; }
}

    // Leaf クラスに追加
    public int fold(int e, ThreeIntsToInt f) {
        return e;
    }

    // Branch クラスに追加
    public int fold(int e, ThreeIntsToInt f) {
        int l = left.fold(e, f);
        int r = right.fold(e, f);
        return f.apply(l, v, r);
    }

    System.out.println(t6.fold(0, new Sum3()));
       // displays the sum of 10, 25, ..., and 30.
```

1.2.1.3　fold その 2　その 1 の実装では，leaf と branch のために別の引数を用意したが，これらをひとつのオブジェクトにまとめることも可能である．実装その 2 として，* 葉の時に呼び出すメソッド * 枝の時に呼び出すメソッド

を持つオブジェクトを受け取る形で書いてみよう．このオブジェクトは，木の場合分け処理を表現していることから，インターフェース名を CaseForTree としている．

```
public interface Tree {
    ...
    int fold(int e, ThreeIntsToInt f);
    int fold(CaseForTree c);  // Overloading!
}

public interface CaseForTree {
    int caseLeaf();
    int caseBranch(int l, int v, int r);
}

public class SumTree implements CaseForTree {
    // no instance variables

    public SumTree() {
        // nothing to initialize
    }

    public int caseLeaf() { return 0; }
    public int caseBranch(int n, int m, int p) { return n + m + p; }
}
```

```
// Leaf クラスに追加
public int fold(CaseForTree c) {
    return c.caseLeaf();
}

// Branch クラスに追加
public int fold(CaseForTree c) {
    int l = left.fold(c);
    int r = right.fold(c);
    return c.caseBranch(l, v, r);
}

    CaseForTree c = new SumTree();
    System.out.println(t6.fold(c));
```

1.2.2 関数インターフェースとラムダ式

さて，ここまで関数を表現するためにいちいちクラスを書いてきたが，これらの実装の問題点は，* 関数の引数型・返値型毎にインターフェースを用意する * 関数毎に別のクラスを用意する

のがかなり煩雑である，という点である．ひとつめの問題は，ジェネリクスを用いることである程度解決する．例えば，ライブラリ `java.util.function` にある[5]`Function<T,R>` というジェネリックインターフェースには `R apply(T x);` というメソッドが宣言されている．よって，上の `IntToInt` のようなインターフェースは `Function<Integer, Integer>` と書けばよい[6]．(`int` を `Integer` に変える必要はある．) また，ふたつめの問題は，現在の Java では「ラムダ式」で解決することができる．「ラムダ式」は OCaml の `fun` 記法に対応する，Java で匿名関数を表すための記法である[7]．

Java でのラムダ式の一番基本的な記法は，

`(T1 x1,...,Tn xn) -> { 〈文の並び〉 }`

という形を取る．これで n 引数を受け取って〈文の並び〉を実行する匿名関数として使うことができる．`return` された値が返り値となる．例えば，上の `Sum3` は，クラスを書かずとも，ラムダ式を使って以下のようにして作ることができる．

```
ThreeIntsToInt f = (int n, int m, int p) -> { return n + m + p; }
```

ラムダ式において返り値の型は宣言しない (できない)．また，ラムダ式はメソッドをひとつしか持たない (名前は `apply` でなくてもよい) インターフェースが期待される文脈でしか使うことができない．ここで「文脈」といっているのは，`ThreeIntsToInt` 型の `f` に代入しようとしている，というプログラム上の文脈のことである．`f` の型であるインターフェース `ThreeIntsToInt` が `apply` メソッドしか持たないことから，この関数は `apply` メソッドの実装であると解釈してもらえるわけである．

---

[5] 使うには 'import java.util.function.*;' を各ファイル冒頭に置くこと

[6] 他に 2 引数関数のための `BiFunction<T,U,R>` というインターフェース (`T` と `U` が引数の型を表す) もあるが，3 引数以上の場合は用意されていないようだ．

[7] 「ラムダ」というのはギリシャ文字の λ のことである．**ラムダ計算 (λ-calculus)** という計算のモデルの理論で，x を引数とし (x を含む)〈式〉を計算して返す関数を $\lambda x.$〈式〉という形式で表すことに由来している．例えば，「1 を足す関数」は $\lambda x.x+1$ と書く．プログラミング言語によっては `lambda` が匿名関数のキーワードになっている場合もある (Lisp など)．

上の記法以外にも，以下の表のような様々な短縮形が用意されている．

記法	備考
`(T1 x1,...,Tn xn) -> { ... }`	n 引数で，… の文 (の列) を実行する．return された値が返る
`(x1,...,xn) -> { ... }`	パラメータの型宣言は省略できる
`x -> { ... }`	1 引数の場合，括弧すら省略できる
`(T1 x1,...,Tn xn) ->`〈式〉	中括弧の中が単一の `return`〈式〉`;` の場合，〈式〉だけでよい
`(x1,...,xn) ->`〈式〉	型宣言の省略
`x ->`〈式〉	1 引数の場合の括弧の省略

(ふつうは) 文脈側にあるインターフェースから型が決まるので，パラメータの型宣言は特にしなくても型推論が行われる．(ちなみに，型宣言のないパラメータと型宣言のあるパラメータを混ぜることはできない．`(n, int m, p) -> ...` のようなものはエラーになる[8]．)

上の `map` や `fold`(その 1) を使う例をラムダ式 (の省略形) を使って書き直すと以下のようになる．

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

Tree t8 = t6.map(n -> n * 2);
System.out.println(t7);
System.out.println(t7.fold(0, f));
```

Java のラムダ式は，結局のところオブジェクトである．よって，ラムダ式を呼び出す特殊な構文はなく，呼び出すためには必ずメソッド呼び出しを経由する必要がある．つまり，クラスを定義せずにオブジェクトを作る特殊な記法，程度に理解しておくべきである．コンパイルした結果を見ると，実はラムダ式に対応するクラスが自動で生成されている．また，(本講義ではこれ以上深くはふれないが) ラムダ式の使用に関するいくつかの規則も，ラムダ式がクラス定義+`new` の略記であることを知っておくて理解しやすいだろう．

1.3 関数ポインタを使ったプログラミング in C (C/hofun)

C 言語には OCaml のような一般的な高階関数を扱う機能はない．そもそも (GCC の独自拡張を除くと) 関数の中で関数を定義できないので，カリー化関数を表現したかったら，自分で関数閉包を作ったり，関数閉包を呼び出す仕組みを作ることになる．また，匿名関数のための仕組みはない．それでも，上で紹介したような高階関数を使ったプログラミングが「ある程度」可能であることを紹介する．

関数ポインタは，概念としては関数を指すためのポインタで，`*` を使って関数を参照して呼び出すことができる．まず，$\sum$ の例を見てみよう．

```
int sigma(int (*f)(int), int n) {
  if (n < 1) {
    return (*f)(0);
  } else {
```

[8] Java 言語仕様 Java SE8 版 15.27 節

```
    return (*f)(n) + sigma(f, n - 1);
  }
}

int square(int n) {
  return n * n;
}

int cube(int n) {
  return n * n * n;
}

int main(void) {
  printf("1^2 + ... + 20^2 = %d\n", sigma(square, 20));
  printf("1^3 + ... + 20^3 = %d\n", sigma(cube, 20));

  return 0;
}
```

C 言語の関数ポインタまわりでややこしいのは変数の型宣言の構文である．後置された `(int)` が整数を 1 つ引数に取る関数であること，`f` の前の`*` が `f` がポインタであることを示している．

`main` 関数にあるように，関数へのポインタを他の関数に渡す時には関数名だけを書けば，`&`をつけなくても暗黙のうちにポインタとして扱われる．また，呼び出しの際にはポインタの参照`*f` をしているが，これには括弧が必要である．この括弧がないと，「関数 `f` を呼んだ結果 (`f(n)`) であるところのポインタが指す先を`*` 参照する」という意味になってしまう．(C 言語では関数呼び出しも後置演算子扱いで，後置演算子は常に前置演算子よりも優先される．)

1.3.1 型宣言の読み方

(参考: http://www.unixwiz.net/techtips/reading-cdecl.html など，解説記事多数あり．)

C 言語の型宣言は，識別子の前後にいろいろくっついて型を表すこともあり，読み方がややこしい．

1. 識別子から読み始める．`f` なら“`f` is …”と読み始める
2. 各記号の読み方は以下の表の通り．
3. (括弧がない限り) 後置の `()` (や `[]`) を優先して読む
4. 最後に左の型を読むと `f` が何かがわかる．

記法	読み方	備考
`(...)` (後置)	a function returning	括弧内は引数の型をカンマで並べる
`[...]` (後置)	an array of	…には配列の要素数を書いてもよい

記法	読み方	備考
* (前置)	a pointer to	

例えば `int (*f)(int)` は `f` is a pointer to a function (taking an `int`) returning `int` と読むことができる．`*f` のまわりの括弧がないと，`f` is a function returning a pointer to `int` となって，違う意味になってしまう．

また，関数は「関数へのポインタ」を返すことはできるが「関数そのもの」を返すことはできないため，OCaml での `f: T1 -> T2 -> T3` のような (`T1` 型の引数を受け取ったら，「`T2` 型の引数を受け取り `T3` 型の値を返す関数」を返す) 関数を定義したい場合は，「`T1` 型の引数 (仮に `x` としよう) を受け取ったら『`T2` を受け取り `T3` を返す関数』へのポインタを返す関数」として定義する．この場合，`f` の定義は `T3 (*f(T1))(T2) { ...}` のように書く．(上の読み方との対応を取ってみよう．)

1.3.2 表記ゆれいろいろ

関数の受け渡しは必ずポインタでされることから，いくつかそれを利用した表記揺れが存在する．

- 実は，`f` の宣言につけた `*` は省略して，

```
int sigma(int f(int), int n) { ... }
```

と書いてもよい．が，この省略ができるのは関数パラメータに限られ，`{}` 内の局所変数の宣言の場合は省略できないので，この講義では常に `*` をつける．

- 関数ポインタを使った呼び出し `(*f)(n)` も，`*` を省略して `f(n)` と書いてよい．`f` がポインタであることを意識・明示するために，この講義では常に`*` をつける．
- 関数のポインタを取得する際に `&` をつけて (`sigma(&cube,20)` のように書いて) もよい．上の点とは一貫していないようにも思えるが，この講義では`&` はつけない．配列 (これも関数と同様受け渡しできるのはポインタだけである) の場合，配列名が先頭要素へのポインタに自動変換されるのと揃えている．

1.3.3 2 分木の map と fold

記法さえ飲み込めば `map` や `fold` を書くのは難しくない．

```
struct tree *map(int (*f)(int), struct tree *t) {
  if (t->tag == LEAF) {
    return newleaf();
  } else /* t->tag == BRANCH */ {
    struct branch *b = &t->dat.br;
    struct tree *newleft = map(f, b->left);
    struct tree *newright = map(f, b->right);
    int newvalue = (*f)(b->value);
    return newbranch(newleft, newvalue, newright);
  }
}
```

```
int fold(int e, int (*f)(int, int, int), struct tree *t) {
  if (t->tag == LEAF) {
    return e;
  } else /* t->tag == BRANCH */ {
    struct branch *b = &t->dat.br;
    int l = fold(e, f, b->left);
    int r = fold(e, f, b->right);
    return f(l, b->value, r);
  }
}

int add3(int a, int b, int c) {
  return a + b + c;
}

int dbl(int a) {
  return a * 2;
}

int main(void) {
  // Construct a tree
  struct tree *t1 = newbranch(newleaf(), 10, newleaf());
  struct tree *t2 = newbranch(newleaf(), 25, newleaf());
  struct tree *t3 = newbranch(t1, 15, t2);
  struct tree *t4 = newbranch(newleaf(), 60, newleaf());
  struct tree *t5 = newbranch(newleaf(), 48, t4);
  struct tree *t6 = newbranch(t3, 30, t5);

  print_tree(t6); printf("\n");
  int s6 = fold(0, add3, t6);
  printf("the sum of integers in t6 is %d\n", s6);

  struct tree *t7 = map(dbl, t6);
  print_tree(t7); printf("\n");
  int s7 = fold(0, add3, t7);
  printf("the sum of integers in t7 is %d\n", s7);

  return 0;
}
```