2021年度「プログラミング言語」配布資料 (13)

五十嵐 淳

2022年10月02日

1　モジュールと実装の隠蔽

モジュール (module) とは，工業製品などにおける規格化された構成単位を指すための用語である．要するに部品のことなのだが，この「規格化」というのは重要な点で，これによって例えば異なる会社が製造した部品を集めてひとつの大きな製品を作ることができるし．一部が故障しても簡単に交換することが可能になる．ソフトウェアにおいても，モジュールという用語はプログラム部品の構成単位を指していて，多くのプログラミング言語が，プログラムをモジュールの集まりとして構成するための支援機構を用意している．工業製品と違って**ソフトウェアはいくらでもコピーすることができる**ので，交換，といっても壊れた部品を新調するより，より高機能な別の部品にアップグレードすることが想定されるだろう．モジュール化の際に重要なのが，モジュールの**実装の隠蔽**である．部品は「規格」さえ満たしていれば部品の「つくり」(実装の方法) は何でもよいはずであるし，逆に，部品を使う側が，規格には現れない部品の機能に依存してしまうとうまく交換できなくなってしまうため，モジュールの実装の隠蔽はプログラミング言語でサポートされるべき重要な機能となる．別の言い方をすると，実装の隠蔽によって，変更の影響範囲が限定でき，結果としてシステムの変更に対する頑健性が高まるのである．

ここでは，Java と OCaml を中心に，モジュールプログラミングのための言語機構を学ぶ．機構の詳細は，言語毎にかなり異なるのが実際なのだが，多くの言語が以下のような機構の提供を通じて大規模プログラミングやモジュール化を支援している．

- (階層的) 名前空間
- 実装の隠蔽
- 分割コンパイル

1.1　階層的名前空間

プログラムが大規模になってくると，まず問題になることのひとつが「名前」である．つまり，クラス，メソッド，関数などの名前などで適当なものが尽きてきて，既に使われている名前と衝突してしまうなどの問題が発生する．例えば `add` などという名前は数値の足し算にも使うだろうし，(2分探索木のような) データ構造にデータを追加する際にも使いたくなるだろう．多くのプログラミング言語では，関連した定義の集まりをグループ化して，そのグループに名前をつける機能が提供されている．例えば数値の演算の集まりであれば `number`，データ構造ならば `collection` といった具合である．そして，定義を参照する際には，グループ内部では `add` などの単純な名前で，グループ外からは「`number` の `add`」「`collection` の `add`」といったグループ名と単純な名前の組といった複合的な名前を使って参照するようにすることで，名前の混同を解決することができる．

言語によっては「グループのグループ」が作れたり，別グループに属する定義を単純な名前で参照するための仕組みが用意されていることも多い．

1.1.1 Java の パッケージ (java/bstEncapsulated)

Java のクラスは「メソッドの集まりに名前をつけている」という意味では上のようなグループ化機能を提供しているともいえる (特に `Math` など `static` メソッドの集まりとしてのクラスは，グループ化用途に使われている) が，クラスは同時にオブジェクトの定義でもあるので，2 分探索木のような複数のクラスでひとつのまとまった機能を提供をしている場合には，複数のクラスをグループ化したくなる．Java では，**パッケージ (package)** という機能で，クラスのグループ化を行う[1]．

例として，2 分探索木のクラス・インターフェースである `BinarySearchTree`，`Leaf`，`Branch` をパッケージを使ってグループ化してみよう (それらを使う `Main` クラスをそのグループの外側に置く)．パッケージの名前は `bst` とする．クラスをパッケージに入れるのは非常に簡単で，クラスを定義した各ファイルの先頭で `package` 〈パッケージ名〉 ; という行 (パッケージ宣言という) を追加するだけである．

```
package bst;

public interface BinarySearchTree {
    ....
}
```

このようにすると，このインターフェースは二種類の名前を持つことになる．ひとつは `BinarySearchTree` (これを **単純名 (simple name)** という) で，同じパッケージに属する他のクラス，インターフェースは，このクラスを単純名で参照することができる．もうひとつは `bst.BinarySearchTree` (これを **完全限定名** または**完全修飾名 (fully qualified name)** という) で，これはパッケージ外から参照する時に使うことになる．

1.1.1.1 パッケージ名とファイルシステムについて Java ではパッケージ名は英小文字で始まる名前をつけることが習慣となっている．また，パッケージを整理する (パッケージ名の衝突を防ぐ) ために，ドット (.) で区切った長い，階層化された名前を使うことができる．例えば Java の標準ライブラリのパッケージは `java.lang` や `java.util` といった `java` で始まる名前がつけられている．さらに，全世界的に (!) パッケージの名前が衝突しないように，パッケージの名前は `jp.ac.kyoto-u` のような，組織のインターネット上のドメイン名 (京都大学なら `kyoto-u.ac.jp`) を逆にしたような名前を使うことが推奨されている．これまでの `package` 宣言を使わないクラスは，「無名パッケージ」という特殊なパッケージに含まれている，と考えることになっている．

Java のほとんどの (筆者が知る限り全ての) 処理系は，パッケージ名の (. を使って作られる) 階層と，ファイルシステムのディレクトリ (フォルダ) の階層を対応させてプログラムが書かれたファイルを整理している．つまり，プロジェクトのディレクトリが「名前なしパッケージ」に属する定義のファイル (例えば `Main.java`) が置かれ，`bst` というディレクトリの `BinarySearchTree.java` というファイルに `bst.BinarySearchTree` の定義が書かれている，といった具合である[2]．Java の言語仕様を読む限りこのようなファイルの配置は要請さ

[1] これから見るように Java で当初から提供されているパッケージ機能は，パッケージをグループ化できないなど，かなり貧弱である．Java 9 では，もう少し本格的なモジュール機能が追加されているが，これについては扱わない．また，実は Java ではクラスの中にクラスが定義できるので，クラスを使ってクラスをグループ化することも可能なのでが，これについてもここでは扱わない．

[2] BlueJ では，パッケージはフォルダ表示されてフォルダの中や親フォルダに「移動」することができる．また，ファイル編集中に `package` 宣言を追加したりパッケージ名を変更したりすると，ファイルの保存時にファイルを移動してもよいかの確認が行われる．

れていないのだが，最初に開発された Java コンパイラがこういう方式を取ったため他の処理系も追随しているのだと思われる．

このように Java のパッケージの名前は階層化されていて，ファイルの配置もファイルシステムに合わせて階層化されているのだが，パッケージ自体には階層化機能がなく，aaa.bbb というパッケージが aaa パッケージの**内部にあるわけではない**．Java のパッケージはフラットで aaa と aaa.bbb は関係のない単なる別々のパッケージとなり，aaa.bbb から aaa 内のクラスを参照するためには完全限定名を使う必要がある．また無名パッケージは完全限定名を持たないため，無名パッケージの外側からは使うことができない．

1.1.1.2 import 宣言による完全限定名の省略

パッケージによる名前の整理は便利であるものの，完全限定名を常に使わなければいけないとするとプログラムが読み辛くなる一因となる．そこで，Java では import 宣言を使うことで，そのファイル内部では単純名で参照することができるようになる．

```
import〈パッケージ名〉.〈クラス/インターフェース名〉;
import〈パッケージ名〉.*;    // パッケージ内全てのクラス・インターフェースをまとめて import
```

String などの基本的なクラスは java.lang というパッケージに属しているのだが，このパッケージ内のクラスはプログラムに何も書かなくても，自動的に全て import されている．

また，static メソッド・変数は基本的に〈クラス/インターフェースの完全限定名〉.〈メソッド名〉で参照するが，これをクラス/インターフェース名すら省いた名前で使うための import static 宣言というのも用意されている．

```
import static java.lang.Math.sin;  // java.lang.Math クラス の sin メソッド を sin で使う
import static java.lang.Math.PI;   // import static 宣言のクラス/インターフェース名は完全限定名でなければな

... double x = sin(PI);
```

import や import static 宣言は，package 宣言 (あれば) とクラスやインターフェース定義の間に書かなければならない．

1.1.2 OCaml のモジュール (ml/bstModule)

OCaml では，定義をグループ化したものをモジュールと呼ぶ．モジュールは，以下のようなモジュール宣言で定義することができる．

```
module〈モジュール名〉=
  struct
    〈let や type による定義の列〉
  end
```

以下は非常に簡単な，1 を足す関数 inc のみを含むモジュール Sample の定義である．

```
# module Sample =
    struct
      let inc x = x + 1
    end;;
module Sample : sig val inc : int -> int end
```

対話的処理系からの応答，特にコロン以下は，**シグネチャ (signature)** と呼ばれ[3]，モジュール内にどのような定義が含まれるかを示している．この例の場合，`sig end` にはさまれて，`int->int` 型の関数 `inc` が定義されている，ということが示されている．OCaml のモジュール名は英大文字で始める必要がある．

モジュールの中の定義は (Java と同様に)．を使って〈モジュール名〉.〈名前〉という形式で参照できる．

```
# Sample.inc 100;;
- : int = 101
```

また，Java のパッケージと違ってモジュール内にモジュールを定義することもできる．内側のモジュールの定義は〈外側のモジュール名〉.〈内側のモジュール名〉.〈名前〉として参照するが，これは単に〈外側のモジュール名〉.〈内側のモジュール名〉が内側のモジュールを指すための記法であることからでてくることである．内側のモジュールから外側のモジュール内の定義を参照する際には，単純な名前で参照することができる．

```
# module Sample =
    struct
      let inc x = x + 1
      module Inner =
        struct
          let z = inc 100  (* inc doesn't have to be qualified *)
        end
    end;;
module Sample :
  sig val inc : int -> int module Inner : sig val z : int end end
# Sample.Inner.z;;
- : int = 101
```

シグネチャが読み辛いが，整形すると，

```
module Sample :
  sig
    val inc : int -> int
    module Inner :
      sig
        val z : int
      end
  end
```

となって，モジュール内にモジュールが定義されていることが読み取れるだろう．

2 分探索木のサンプルコード (`src/ocaml/bstModule`) にあるように，`struct end` の間には `type` 宣言を並べることができる．

上の例はインタラクティブな処理系でのモジュールの定義の方法だが，OCaml ではモジュールとプログラムファイルとが対応していて，ひとつのプログラムを複数のファイル (モジュール) に分割して記述することができる．これについては後で詳しくみていく．

[3]「シグネチャ」はいわゆる「サイン」ではなく，論理学や代数学で使われる用語で，ある体系で使われる演算子をその引数の数や型とともに列挙したものを指す．

1.1.2.1 open 宣言

Java の import と同様なことは OCaml では open 宣言というもので行なう．open〈モジュール名〉で，指定されたモジュール内で定義された名前が単純名で参照できるようになる．

```
# open Sample;;
# inc Inner.z;;
- : int = 102
```

このように inc と Inner が Sample．なしで使えるようになる．(Java の import と import static のような違いも特にない．)

open は，モジュール定義の struct end 間に配置することもできて，当該モジュールの定義の範囲でのみ単純名を使いたい場合に有用である．

```
# module Sample2 =
    struct
      open Sample.Inner
      let a = z + 1
    end;;
module Sample2 : sig val a : int end
# Sample2.a;;
- : int = 102
# z;;
Characters 0-1:
  z;;
  ^
Error: Unbound value z
```

また，let open〈モジュール名〉in〈式〉という形式で〈式〉の計算中に一時的にモジュールを open することもできる．

```
# let open Sample.Inner in z + 100;;
- : int = 201
```

1.2 実装の隠蔽

モジュールの大事な機能が実装の詳細の隠蔽であることは既に述べた．これまでに本講義で扱った 2 分探索木について，隠蔽したい実装の詳細とはなんだろうか．例えば，Java 版ならば部分木を表すインスタンス変数に left，right といった名前が与えられているということは使う側にとってはどうでもよいことである．実際，Java 版では，インスタンス変数は private 修飾子をつけて宣言されているおかげで，2 分探索木を使う側は直接アクセスすることはできず，名前についてはうまく隠蔽が行われている．一方，OCaml 版では，tree 型の値にパターンマッチをすることで簡単に右の部分木や左の部分木の値を取り出すことができてしまう．Java 版に問題がないわけではなく，例えば，2 分探索木の構造をとっていない 2 分木を簡単に作ることができてしまう (もちろん OCaml 版も同じ問題を抱えている)．そもそも，2 分探索木の目的が (順序づけられた) データ (ここでは整数) の集まりを表すことであると考えると，データの追加，削除，検索さえ行えれば，内部的には，配列，連結リスト，赤黒木など，どんなデータ構造を使っていてもよいはずである．

以下，Java, OCaml それぞれでの実装の隠蔽を行うための仕組みについて紹介し，勝手な 2 分探索木の操作を許さないようにする．

1.2.1 Java (java/bstEncapsulated)

Java での実装の隠蔽の基本手段は public, private などの**アクセス修飾子 (access modifier)** による．アクセス修飾子は，クラス，メソッドなどの定義の単位につけられて，それが「プログラムのどの範囲から見えるか」を規定する．例えば，public は，どこからでも見えることを許す．private の一般的な意味は「その定義を囲む単位の中だけで見える」ということで，インスタンス変数やメソッドにつければ，そのクラス内でのみ見えることになる．(クラスには private をつけてはいけない．ただし，本講義では扱わないクラス内クラスを除く．) これまでのプログラミングでは，クラス，コンストラクタ，メソッドは public，インスタンス変数には private をつけてきたが，これは，オブジェクトの持つ状態の詳細は private で隠蔽し，メソッドによってのみオブジェクトを操作できるようにするという意味あいがあったわけである．

Java には public, private の他に，二種類のアクセス範囲が設けられている．そのひとつが「同じパッケージ内からのアクセスを許す」というパッケージ・アクセスで，これは「アクセス修飾子を省略する」ことで設定する．もうひとつ，継承というクラスを拡張する機構と関連がある protected アクセスというものがあるのだが，ここでは触れない．

既に述べたように，Java 版の 2 分探索木実装では，インスタンス変数に関する隠蔽は達成できている．以下では，主にクラスやコンストラクタのアクセス修飾子を変更することによって，まず，2 分探索木ではない木構造の生成を禁止する．これは，Branch クラスのインスタンスを好き勝手に作れてしまうのが問題なので，コンストラクタのアクセス修飾子を変更することによって，これを制限する．実はパッケージアクセスがその目的にうってつけである．以下が，コンストラクタをパッケージアクセスに制限したクラス定義である．

```
package bst;

public class Branch implements BinarySearchTree {
  ...
  Branch(BinarySearchTree left, int v, BinarySearchTree right) {
     ...
  }
}
```

コンストラクタのヘッダに public も private もついていないことに注意してもらいたい．このようにすることで，new Branch(...) という式は bst パッケージ内のクラスでしか書けないことになる．一番制限のきつい private を指定すると，new Branch(...) が書けるのが Branch クラス内部だけに制限されることになるが，Leaf クラス内で Branch オブジェクトを作ることがあるため，これでは制限がきつすぎる．コンストラクタの本体は全く変更する必要はない．このように定義をして，(bst に属さない)Main クラスで new BinarySearchTree(...) と書くと (きちんと (？)) コンパイル・エラーが発生する．

さらに，Branch については，パッケージ外で Branch 型の変数すら使うことがないので，クラスの存在すらパッケージ外に見せる必要はない．クラスの存在を隠蔽するには class キーワードの前の public を取ればよい．結局 Branch クラスの定義は以下のようになる．

```
package bst;

class Branch implements BinarySearchTree {
  ...
  Branch(BinarySearchTree left, int v, BinarySearchTree right) {
     ...
  }
}
```

クラスそのものを隠蔽すると，コンストラクタも (例え public をつけていたとしても) 隠蔽されるので，クラスの存在さえ隠蔽できれば十分に思えるかもしれない．実際，今回の用途では大きな違いはない．しかし，Java ではクラス内にコンストラクタを複数定義することができ，それぞれ違うアクセス修飾子を与えることができるので，クラスは公開しつつ，いくつかのコンストラクタだけ隠蔽する，といった細かい制御が可能である．(ちなみに，コンストラクタに private をつけると，パッケージ内部の他のクラスからも隠蔽することができる．) また，クラスを隠蔽すると，その型の変数すら宣言できなくなるし，そのクラスを返り値型とするメソッドも全く使えなくなってしまう．

Leaf も同じ要領で隠蔽できるが，ここは注意が必要である．なぜなら，2 分探索木を使うにあたって，最初は空の木がどうしても必要である，ということである．素朴に Leaf クラスを隠してしまうと，2 分探索木でない構造が作れないどころか，そもそも 2 分探索木を手に入れる手段がなくなってしまう．そこで，予め空の木のインスタンスをひとつ用意して public に使えるようにしておく必要がある．実はインターフェースには，static 変数を持たせることができるので，ここに追加しよう．

```
public interface BinarySearchTree {
    ....
    BinarySearchTree delete(int n);

    BinarySearchTree EMPTY = new Leaf();
}
```

インターフェースでは，アクセス修飾子を省略しても public となり，変数は自動的に static となるので，この変数 EMPTY は BinarySearchTree.EMPTY という名前でどこからでも使うことができる．

こうした 2 分探索木パッケージは以下のようにして使うことができる．

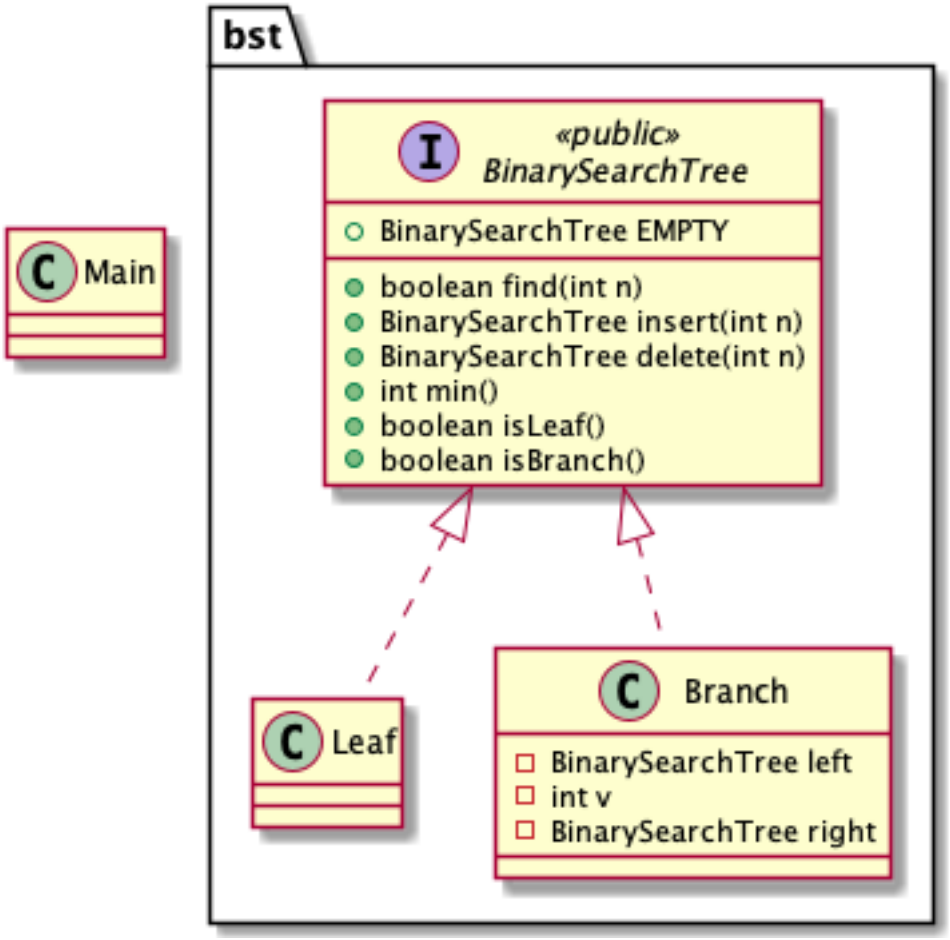
```
import bst.*;

public class Main {
    public static void main(String[] args) {
        BinarySearchTree t0 = BinarySearchTree.EMPTY;
        BinarySearchTree t1 = t0.insert(30);
        ...
```

むしろ大事なのは，Main クラスの中で new Leaf() や new Branch(...) と書いた時にどのようなエラーが報告されるかを見ておくことだろう．

1.2.1.1 クラス図

クラス図を示す．パッケージ `bst` の中では `BinarySearchTeee` のみが public であるため (この表記は特に UML 公式のものではない)，`bst` の外側に置かれた `Main` からは `BinarySearchTree` (と，それが含むメソッドと static 変数) しかアクセスできない．



1.2.1.2 まとめと限界

- Java では定義ごとにアクセス修飾子を使って参照する範囲を指定することができる．
- `private` は当該クラス内のみ
- `public` はどこからでも．インターフェースに宣言されたメソッドは (修飾子を省略しても) `public` メソッドとして扱われる．
- 何もつけない場合は，パッケージ内のみで，アクセス可能

ただし，公開範囲の制限の粒度が粗いので，例えば「パッケージ P のみにこのクラスを使えるようにしたい」のような制御までは難しい．また，`min()` のような，必ずしも 2 分探索木を使う側に公開しなくてもよいメソッドも `public` になってしまっている．Java 9 に追加されたモジュールシステムはある程度これを解決することができる．

1.2.2 OCaml (ocaml/bstModule)

OCaml のモジュールシステムの場合，シグネチャを使ってモジュールの詳細の隠蔽を行う．隠蔽できるものは，基本的には

- 関数や型の定義の存在
- 型の定義内容

である．前者は名前がそもそも見えなくなるという意味で Java のアクセス修飾子と似たような効果が得られる．後者は，型が定義されていることは公開されているが，その型がどのようなコンストラクタから構成されているかといった情報は隠蔽される，といったものである．

1.2.2.1 シグネチャ モジュールのシグネチャは，いわばモジュールの型で，既に見たようにモジュールを定義すると，そのシグネチャが推論される．ここで推論されたシグネチャは，モジュール内の定義が丸見えになっているものである．例えば，2 分探索木の型・関数群をモジュール化してみよう．名前を `Bst` とする．これまで 2 分探索木のデータ型には `tree` という名前をつけてきたが，モジュールの名前とあわせて `Bst.tree` となると `Bst` の `t` と `tree` の意味が重複するので，「`Bst` モジュールで定義されている型」という意味で type の頭文字で単に `t` という名前をつける．(このモジュールで定義される一番重要な型に `t` という名前をつけるのは OCaml のライブラリでもよく使われる慣習である．)

```
# module Bst =
  struct
    type t =
      Lf       (* Leaf *)
    | Br of {  (* Branch *)
        left:  t;
        value: int;
        right: t;
      }

    let rec find(t, n) = ...
    let rec insert(t, n) = ...
    let rec min t = ...
    let rec delete(t, n) = ...
  end;;
module Bst :
  sig
    type t = Lf | Br of { left : t; value : int; right : t; }
    val find : t * int -> bool
    val insert : t * int -> t
    val min : t -> int
    val delete : t * int -> t
  end
#
```

となる．前述したように，この sig end で挟まれた部分がシグネチャで，

- 関数・値の定義を示す val〈名前〉:〈型〉
- 型の定義を示す type〈名前〉=〈型定義〉

が並んでいる．特に，上のシグネチャでは t の定義が全て繰り返して書かれているので，この Bst モジュールを使う側としては，「t 型の値は，Lf と Br で作ったり，逆にこれらを使ってパターンマッチできるんだな」ということがわかる．この「値の作り方」「値の壊し方」がわかる，という意味で Bst.t 型は実装が公開されているといえる．

1.2.2.2 シグネチャの定義と signature ascription

OCaml では，モジュールを定義する際に，いっしょにそのシグネチャを指定することで，モジュールの外部に公開する情報を制限することができる．構文としては，

```
module〈モジュール名〉:〈シグネチャ〉=
  struct
    ...
  end
```

という形でモジュール名の後にコロン付きで指定する．このシグネチャの指定を signature ascription[4]ともいう．〈シグネチャ〉部分には上記のような sig 〜end を直接書いてもよいが，煩雑になりがちなので，別途 module type という構文

```
module type〈シグネチャ名〉=
  sig
    ...
  end
```

を使ってシグネチャに名前をつけて (定義して) おいて，その名前を使うことができる．以下，いくつかの隠蔽パターンを見てみよう．

1.2.2.2.1 min の隠蔽

まず，2 分探索木の最小値を求める関数 min を外側から使えないようにしてみよう．min を隠蔽するためには該当する val 行を除いたようなシグネチャを考えればよい．これに BST_WITHOUT_MIN という名前をつけてみよう[5]．

```
# module type BST_WITHOUT_MIN =
  sig
    type t = Lf | Br of { left : t; value : int; right : t; }
    val find : t * int -> bool
    val insert : t * int -> t
    val delete : t * int -> t
  end;;
```

これを使って，(struct〜end の部分は Bst と同一の) Bst2 を定義する[6]．

---

[4]ascription は「A の原因を B に帰する」「A を B に属するものとみなす」といった意味の動詞 ascribe A to B からきている．なぜこの文脈で ascription と呼ぶのか正直あまりよくわからないのだが「モジュールを与えられたシグネチャに属するものとみなす」ということかなと思う．

[5]OCaml ではシグネチャの名前は全て大文字で (単語間はアンダースコアで繋いで) 定義する慣習がある．

[6]上の定義では struct〜end 部分を繰り返したが，以下のようにすると既存のモジュールを使って別シグネチャを与えることもできる．

```
module Bst2 : BST_WITHOUT_MIN = (* signature ascription! *)
  struct
    type t =
      Lf       (* Leaf *)
    | Br of {  (* Branch *)
        left:  t;
        value: int;
        right: t;
      }

    let rec find(t, n) = ...
    let rec insert(t, n) = ...
    let rec min t = ...
    let rec delete(t, n) = ...
  end;;
module Bst2 :
  sig
    type t = Lf | Br of { left : t; value : int; right : t; }
    val find : t * int -> bool
    val insert : t * int -> t
    val delete : t * int -> t
  end
```

と，ascribe した通りの min のないシグネチャとなり，min 関数は Bst2. をつけても使えなくなってしまう．

```
# let open Bst2 in min (Br{left=Lf; value=20; right=Lf});;
Characters ....:
  let open Bst2 in min (Br{left=Lf; value=20; right=Lf});;
                  ^^^
Error: Unbound value min
```

もちろん，モジュールで定義されてもいないものをシグネチャに書くとエラーになる．

```
# module type ERRONEOUS_BST =
   sig
      type t = Lf | Br of { left : t; value : int; right : t; }
      val find : t * int -> bool
      val insert : t * int -> t
      val delete : t * int -> t
      val max : t -> int    (* doesn't exist! *)
    end;;
...
# module ErroneousBst : ERRONEOUS_BST =
```

---

module Bst3 : BST_WITHOUT_MIN = Bst;;
このようにすると，Bst と Bst3 で定義を共有しながらも，公開バージョンと隠蔽バージョンを名前で使いわけることもできる．

```
  struct
    type t =
      Lf       (* Leaf *)
    | Br of {  (* Branch *)
        left:  t;
        value: int;
        right: t;
      }

    let rec find(t, n) = ...
    let rec insert(t, n) = ...
    let rec min t = ...
    let rec delete(t, n) = ...
  end;;
```

とすると，

```
Error: Signature mismatch:
       ...
       The value `max' is required but not provided
```

と，max がないぞ (正確にはシグネチャで要求されているのに実装モジュール側が提供してないぞ)，というエラーになる．

1.2.2.2.2　t の隠蔽　次に，勝手に (2 分探索木になっていない) 木構造を作るのを防ぐために，型 t を隠蔽してみよう．しかし，t を隠蔽するといっても，min の場合と違って，type t = ... の行を丸ごと除いてしまうと，困ったことになってしまう．

```
module type BST_WITHOUT_T =
  sig
    val find : t * int -> bool
    val insert : t * int -> t
    val delete : t * int -> t
  end;;
Characters 49-50:
      val find : t * int -> bool
                 ^
Error: Unbound type constructor t
```

これは，find の引数の t って何？ というエラーである．型の存在 (名前) は残しながら，それがどのような定義になってるかは隠蔽したいのだが，このためにシグネチャ内では (= 以降の右辺のない) type t という宣言が許されている．

```
module type ABSTRACT_BST_VER1 =
  sig
    type t
```

```
    val find : t * int -> bool
    val insert : t * int -> t
    val delete : t * int -> t
  end;;
```

こうすることで，このシグネチャが与えられたモジュールは「中身はよくわからないが，とにかく t という名前の型を定義していて，find が「その型」の値と整数を引数として「その型」の値を返す，うんぬん」ということがわかる．一般に，このような「関連する操作と合わせて定義された，詳細が隠蔽された型」を**抽象データ型 (abstract data type, ADT)** と呼ぶ．

先程と同じように signature ascription を使って，抽象データ型 Bst モジュールを定義してみよう．

```
module Bst : ABSTRACT_BST_VER1 =
  struct
    ...
  end;;
```

こうすると，コンストラクタを使って木を作ろうとしても (意図通り) エラーになってしまう．

```
# let open Bst in Br {left=Lf; value=20; right=Lf};;
Error: Unbound constructor Br
```

しかし，これでは Java 版で Leaf クラスを素朴に隠蔽してしまうと 2 分探索木が全く使えなくなってしまう，という問題があったのと同じで，隠しすぎである．せめて空の木だけは公開しておかなければならない．空の木は empty という値で公開することにしよう[7]．

```
# module type ABSTRACT_BST_VER2 =
    sig
      type t
      val empty : t
      val find : t * int -> bool
      val insert : t * int -> t
      val delete : t * int -> t
    end;;
...
# module Bst : ABSTRACT_BST_VER2 =
    struct
      type t = Lf | Br of ...
      let empty = Lf
      ...
```

[7] empty を定義する代わりに，Lf だけ公開したようなシグネチャ

```
sig
  type t = Lf
  ...
end
```

にしてはいけないのかと思うかもしれない．しかし，この t は Lf というコンストラクタひとつだけ持つ，という意味である．このことからモジュールの外側で match 〈t 型の式〉 with Lf -> ... (場合分けがひとつ) のような式が書けてしまいまずいことになる．(なぜまずいのかわかりますか？)

```
    end;;
...
```

これでようやく empty を種にして，insert で大きな 2 分探索木を作っていくことができる．

```
# let t1 = Bst.insert(Bst.empty, 10);;
val t1 : Bst.t = <abstr>
```

こうしてできる 2 分探索木の型は (モジュールの外側なので)Bst.t となる．面白いのは= の右側である．<abstr> となっていて，できた値の中身がわからないようになっている．

抽象データ型を使うことで，モジュールを実装する側は，使用側が実装に依存したプログラムになっていないことを保証することができるので，(シグネチャに書かれた操作を提供する限りにおいては) 自由に，その実装を (今回の場合) 配列だったり，赤黒木だったりに変更することができる．

1.2.2.3 まとめ

- signature ascription
- 抽象データ型

1.2.3 クラス vs モジュール

Java のインターフェースと OCaml モジュールのシグネチャ，Java のクラスと OCaml モジュールは，型宣言を集めたもの，関数・メソッド定義を集めたもの，ということで，よく似たものに見えるかもしれない．確かに後述するように，OCaml のモジュールはコンパイルの単位ともなるので，似ている部分もあるが，インターフェースやクラスは基本的にはオブジェクトのための機構である．オブジェクトはひとつのクラスから好きなだけ作り出すことができるが，一方モジュールには「インスタンス」のような概念はないし，そもそも型検査が無事に終わればお役御免のものである (OCaml では，実行時に，この関数はどのモジュールに属するかといった情報は必要ないし利用できない)．また，Java のアクセス修飾子は実装側に付加していくものなので，ひとつのプログラム部品に様々な公開レベルを設けるといったことは非常にしづらい．

1.3 分割コンパイル (ocaml/bstModule2)

分割コンパイル (separate compilation) とは，複数のファイルにわかれたプログラムからひとつの実行可能ファイルを生成するための仕組みである．分割コンパイルでは，まず，プログラムの一部を構成するソース言語のファイル (**コンパイル単位 (compilation unit)** とも呼ぶ) をオブジェクトファイルと呼ばれる中間表現にコンパイルする．(この「オブジェクト」は Java などの「オブジェクト」とは関係がない．) このオブジェクトファイルを**リンク (結合，linking)** と呼ばれるプロセスでオブジェクトファイルを結合し，実行可能ファイルを生成する．分割コンパイルは，大規模なプログラムの一部のみを修正した時に，修正に依存しない部分のコンパイル (オブジェクトファイルの生成) を省略できるので，全体のコンパイル時間が大幅に削減できる可能性がある．

分割コンパイルを Java や OCaml のような静的に型付けされる言語で実現するためには，コンパイル単位外部で定義された名前についてその型情報を取得しないとコンパイル (特に型検査) ができない．

1.3.1 OCaml における分割コンパイル (ocaml/bstModule2)

OCaml では分割コンパイルの単位はモジュールである．基本的には，ファイルがモジュールと 1 対 1 に対応していて，abc.ml というファイルは Abc という (先頭の小文字を大文字にした) 名前のモジュールとして他のファイルから参照できる．ファイルの中身には (module うんぬんは書かずに) struct〜end の間に書く部分だけ書けばよい．また，モジュール Abc のシグネチャは abc.mli ファイルに (sig〜end の間に書くものだけ) 記述する．この.mli という拡張子を持つファイルはインターフェースとも呼ぶ．.mli がない場合には，対話的環境でシグネチャを指定しない時と同様，コンパイラが.ml から，中身を全て公開するようなシグネチャを推論することになる．

バッチコンパイラ ocamlopt は (-c オプションとつけて呼び出すと) .mli ファイルから .cmi ファイルを，.ml ファイルからオブジェクトファイル .cmx を生成する．.mli ファイルをコンパイルする際には，シグネチャに登場するモジュールの.cmi ファイルが必要となる．また，.ml ファイルをコンパイルする際には，このモジュールから参照しているモジュールと自分自身の.cmi ファイルが必要となる．最終的には，.cmx ファイルを結合して実行可能ファイルを生成する．

例えば，モジュール Abc と Def から成るプログラム (ただし，Def から Abc 内の定義を参照している) は，以下のようにしてコンパイル・リンクすることができる．

1. ocamlopt -c abc.mli や ocamlopt -c def.mli で abc.cmi と def.cmi が生成される．
2. ocamlopt -c abc.ml で abc.cmx を生成する．
3. ocamlopt -c def.ml で def.cmx を生成する．(これは abc.ml のコンパイルをしなくても行える．)
4. ocamlopt -o〈実行可能ファイル名〉abc.cmx def.cmx で，リンクを行い実行可能ファイルを生成する．この時，依存されているものから先に並べる．abc.cmx と def.cmx の順を間違えるとリンクエラーになる．このようにして得られた実行可能ファイルは，以下のプログラムを対話的処理系で実行した時と同じ結果が得られる．

```
module Abc : sig〈abc.mli の中身〉end =
  struct
    〈abc.ml の中身〉
  end

module Def : sig〈def.mli の中身〉end =
  struct
    〈def.ml の中身〉
  end
```

ただし，対話的処理系ならば表示してくれる定義されたモジュールのシグネチャなどの応答は全く表示されず，モジュールを全て定義するとプログラムの実行が終了してしまうので，計算効果を使ってプログラムの実行結果を表示させる必要がある．例えば，最後のモジュールに，何らかの関数を呼び出してその結果を表示するような処理を書く．例えば 2 分探索木のサンプルコードでは，テスト結果の定義群の後に

```
let () =
  print_bool test1;
  print_newline ();
  ...
```

のようなコードを追加して，test1 などの値を表示するようにしている．

OCaml の分割コンパイルにおいて重要なポイントは，.ml をコンパイルする際には，他のモジュールの .mli ファイルさえあればよく，.ml ファイルは必ずしも必要ない，ということである．つまり，.mli さえ揃えておけば，各モジュールで定義されている型・関数の情報がわかるので独立して型検査・コンパイルできるのである．もちろん，ある .ml を実行させるには，結局，それが依存するモジュールの .cmx ファイルが必要になるのであまり意味がないように思えるかもしれないが，OCaml では型検査を通す過程でかなりのバグを取ることになるので，経験上はかなり有用である．

サンプルプログラムでは，bst.mli，bst.ml が 2 分探索木モジュールを，main.ml がそれを呼び出すテストコードになっている．Makefile にコンパイルする依存関係が記述されていてコマンドラインから make を実行すると，main.out という実行可能ファイルが生成されるようになっている．

1.3.2 Java における分割コンパイル

Java では，ひとつのファイル (例えば Foo.java) に対し，そのファイル名 (から .java を除いたもの) と同名のクラスまたはインターフェースを public 修飾子をつけて定義しなくてはならない．また，他のクラス・インターフェースは public をつけずに定義しなくてはならない．既に述べたように，ひとつのパッケージはひとつのフォルダに対応するので，.java ファイルにどんな名前があるかを見れば，どんな public クラス・インターフェースがそのパッケージに定義されているかがわかるようになっている．

プログラムをコンパイルした結果はクラス毎に .class という拡張子を持つファイル (クラスファイルと呼ぶ) に格納される．インターフェースのコンパイル結果も .class になる．ひとつのファイルに複数のクラス・インターフェースが定義されている場合，複数のクラスファイルが生成される．クラスファイルは OCaml の .cmi ファイルの機能も兼ねていて，あるクラスに依存する (を参照する) 別のクラスをコンパイルする時には，依存されているクラスの .class ファイルから，メソッドなどの型情報を取得する．(.class が存在しない時には，対応する .java のコンパイルも同時に行ってくれるようだ．)

Java プログラムは，Java 仮想機械にクラスファイル (例えば Foo.class としよう) を与えると，Foo 内に定義された public static void main(String[]) メソッドを呼び出すことでプログラムの実行を開始する．この際，実行に必要なクラスの定義が順次仮想機械に読み込まれ (**ロード (load)** という)，既に読み込まれたクラスファイルとリンクされて実行が進む．このように，実行時にプログラムを読み込み・結合することを**動的ローディング (dynamic loading)**，**動的リンク (dynamic linking)** という．動的ロード・リンクは失敗することもある．例えばクラス Foo が Bar を使っているとしよう．コンパイル時には存在していた Bar.class が，Foo の実行時には何かの事情で見つからないと，動的ロードに失敗してしまう．また，Bar.class が変更されている場合，Foo が要求している機能を提供していないかもしれないので，Java 仮想機械はロード後に，クラスファイルの検証 (verification)—コンパイルされたコードの型検査と考えられる—を行って，Foo と Bar のつじつまがあっているかを検査する．この時，つじつまが合わない場合にも例外が発生してプログラムの実行が異常終了する．

1.4 講義範囲を越えて

OCaml のモジュールシステムには，モジュールを受け取ってモジュールを返す，というモジュール上の関数である**ファンクター (functor)** など，さらなる (複雑な，実際にプログラムが大規模にならないとなかなか出てこないような状況に対応するための) 機能が沢山あり紹介しきれない．

Java のクラスをロードする仕組みはかなり複雑である．しかも，このクラスファイルをロード・リンクする手続きの詳細すらプログラムによって変更することができる．これを使って，クラスファイルの検索方法をシステム毎に変更したりもできるし，極端な例では，暗号化されたクラスファイルをロードしたり，ロード時にプログラムを書き換えたりすることができる．すごい!