プログラム言語論

亀山幸義

筑波大学 情報科学類

データ抽象

Short Quiz

TWINS システムを再設計したい。
以下の設計のうち、どれを選択すると良いか、簡単な理由をつけて答えよ。

- 学群・学類ごとの module (情報学群の学生 module, ...)
- 処理の種類ごとの module (履修登録、成績登録、成績閲覧、GUI, ...)
- データの種類ごとの module (履修している授業データ、成績データ、学生の連絡先等の個人情報, ...)
- その他 (具体的に)

抽象化 abstraction

適材適所

- 全てのプログラムを機械語で書く。
- 全てのプログラムを C 言語で書く。
- 記号処理プログラムは ML で、システムプログラムを C 言語で書く。
- 構文解析器は BNF で書き、評価器は‥。

Level of Abstraction

- メモリ管理までコントロールしたい⇒ C 言語。
- ごみ集めのみ言語処理系にやってもらう⇒ Lisp etc.
- XML のデータ型のチェックも言語処理系にやってもらう⇒ ML etc.
- 教科書の行列計算アルゴリズムをそのまま表現する⇒数式処理パッケージ

本日の話題: 型による抽象化

かな漢字変換ソフトウェア SKK

SKK の辞書ファイル:

```
おおk /大/多/夛/
きわm /極/
たいs /対/大/對/
おこなt /行/
くらb /比/
...
```

辞書を操作する関数:

- 「見出し語」から「その見出し語を含む行」を得る。
- 「見出し語」に対する「変換結果」を加える。
- 「見出し語」に対する「変換結果」を削る。
- 「見出し語」を加える。
- 「見出し語」を削る。

SKK 辞書のフォーマット

単純なテキストファイル、見出し語は順不同。

辞書が大きくなってきたので、毎回ファイルの頭から 1 文字ずつ検索するのは遅い。

「行」単位で、見出し語の 50 音順でソート。

「見出し語」自体が増えたり減ったりする。

「行」を 2 分木に格納。

非常に巨大な辞書が作成され、もっと高速に検索したい。

ハッシュを使って管理。

... というのは浅薄. 非常に巨大な辞書は (ハッシュする前の段階で既に) メモリにはいりきらない。

SKK 辞書の変遷

開発体制: SKK 本体と辞書管理部分とは、別々の人が開発。

なぜ、勝手に SKK 辞書フォーマットを変更してもうまく動いたか？

SKK 辞書を使うための関数群の仕様を変更しなかったから。

「見出し語」から「その見出し語を含む行」を得る。
「見出し語」に対する「変換結果」を加える。
「見出し語」に対する「変換結果」を削る。
「見出し語」を加える。
「見出し語」を削る。

フォーマットは変わっても、上記の 5 関数を使って得られる結果は常に同じ。

この話のポイント

辞書操作の関数群を、使う人 (正確には、プログラムのうちそれらを使っているパート) と提供する人の合意事項が保たれれば、関数群の実装をどう変更しようと、使う人には影響がない。

辞書関数を使う人は、辞書が特定のフォーマットであることを使ってはいけない。(情報隠蔽, Information Hiding, カプセル化, Encapsulation)

抽象データ型-1

今までのデータ型＝具体データ型 (Concrete Data Type)

新しく定義したいデータ型をどう構成したいかを、具体的に (型構成子を使って) 記述した。
そのデータ型が、具体的にどう実現されているかがわかっている。

抽象データ型 (Abstract Data Type)

データ型の具体的な構成方法 (実現方法) は定めない。
データ型がどう使われるかだけを定める。
つまり、データの実装ではなく、データの仕様。

抽象データ型-2

stack: スタック (その要素は整数) をあらわす抽象データ型

stack 型を操作する関数とその (具体) データ型。

emptystack: stack
push: Int* stack→ stack
pop: stack→ Int* stack
isempty: stack→ bool

これらの関数が満たすべき性質。

isempty(emptystack)=true
isempty(push(x,s))=false
pop(push(x,s))=(x,s)

抽象データ型-3

stack の実装: 前ページの型と性質を満たす限り、どんな実装でもよい。

stack を配列で実装。配列の第 0 要素が、スタックの底。

stack を配列で実装。配列の最終要素が、スタックの底。

stack をリストで実装。

stack を配列で実装。ただし、メモリが不足すれば malloc 関数でメモリを確保。

stack の利用: 前ページの関数を使う限り、どんな使用法でもよい。

前ページの関数以外を使って、stack にアクセスしてはいけない。

たとえば、stack の底のアドレスを得て、スタックの n 番目の要素にアクセスする (stack inspection) のは禁止。

モジュール (module)

ソフトウェアの構成単位 (部品)
プログラムにおける、「何らかの関心事についてのまとまり」
インターフェースと実装から構成される。

インタフェース (interface)

このモジュールを使うための仕様を定めたもの。
通常は、モジュールを使うための関数の名前と型、など。
stack の場合、push,pop,emptystack,isempty 関数とその型。

実装 (implementation)

インタフェースが定められた関数等を実現するプログラム。
インタフェースに従う限り、どのような実装でもよい。
実装のみに現れる関数は、外からは使えない。

データ抽象化とモジュールの歴史

CLU [1974-1975] by Barbara Liskov (2008 年の Turing 賞受賞)
抽象データ型/module 機能を使うことができる言語: ML, Ruby, Modula-2, Python, Perl, Fortran, COBOL, ...

モジュラリティ (modularity)

モジュラリティの高いプログラム

- 関心事ごとのまとまり (モジュールなど) が、それぞれ独立性が高いこと。
- 独立性=インターフェースが実装と分離されていること。
- 情報の隠蔽; インタフェースの仕様を保つ限り実装をどのように変更してもよい。

モジュラリティの高いプログラムの利点

- 各モジュールごとに独立に実装しやすい。
- プログラムの保守性・再利用性がよくなる。

ここまでのまとめ

モジュラリティ＝大規模ソフトウェア作成における重要ポイントの 1 つ:

- 情報の隠蔽 or インターフェースと実装の分離。
- モジュール: モジュラープログラミングに対するプログラミング言語からのサポート (機能)。

Short Quiz, Revisited

TWINS システムを再設計したい。
以下の設計のうち、どれを選択すると modularity が高いか、
maintainability が高いか、簡単な理由をつけて答えよ。

- 学群・学類ごとの module (情報学群の学生 module, ...)
- 処理の種類ごとの module (履修登録、成績登録、成績閲覧、GUI, ...)
- データの種類ごとの module (履修している授業データ、成績データ、学生の連絡先等の個人情報, ...)
- その他 (具体的に)

モジュール (module)

モジュラープログラミングを実現するための、プログラム言語上の機能。
具体的には、Modula, Ada, ML などの言語が module 機能を持つ。

ML の module の例-スタック 1

スタック (要素は整数):

- push : int → スタック → スタック
- pop : スタック → スタック
- top : スタック → int
- emptystack : unit → スタック
- isempty : スタック → bool

ML の module の例-スタック 2

スタック (要素は整数):

- push : int → スタック → スタック
- pop : スタック → スタック
- top : スタック → int
- emptystack : unit → スタック
- isempty : スタック → bool

スタックのインタフェース:

```
module type STACK =
 sig
    type t
    exception EmptyStack
    val push : int -> t -> t
    val pop : t -> t
    val top : t -> int
    val emptystack : unit -> t
    val isempty : t -> bool
 end
```


ML の module の例-スタック 3

スタックの実装:

```
module Stack : STACK =
  struct
    type t = int list
    exception EmptyStack
    let push n st = n :: st
    let split (st : t)  = match st with
      | [] -> raise EmptyStack
      | n::st -> (n,st)
    let pop st = snd (split (n,st))
    let top st = fst (split (n,st))
    let emptystack () = ([] : t)
    let isempty st = (List.length st = 0)
  end
```


ML の module の例-図形 1

Point モジュールのインタフェース:

```
module type POINT =
sig
  type point
  val mk_point : float * float -> point
  val x_coord : point -> float
  val y_coord : point -> float
  val move_p : point * float * float -> point
end
```

ML の module の例-図形 2

Circle モジュールのインタフェース:

```
module type CIRCLE =
sig
  include POINT
  type circle
  val mk_circle : point * float -> circle
  val center : circle -> point
  val radius : circle -> float
  val move_c : circle * float * float -> circle
end
```


ML の module の例-図形 2

Point(平面上の点) モジュールの実装:

```
module Point =
struct
  type point = float * float
  let mk_point (x,y) = (x,y)
  let x_coord (x,y) = x
  let y_coord (x,y) = y
  let move_p (((x,y):point),dx,dy)
    = (x +. dx,y +. dy)
end
```


ML の module の例-図形 3

Circle(平面上の点) モジュールの実装:

```
module Circle =
struct
  include Point
  type circle = point * float
  let mk_circle (p,r) = (p,r)
  let center (p,_) = p
  let radius (_,r) = r
  let move_c (((p,r):circle),dx,dy) =
    mk_circle(move_p(p,dx,dy),r)
end
```

インタフェース、実装の両方の再利用が可能。

まとめ

大規模プログラミング、モジュラリティ
抽象データ型
抽象データ型の実現としてのモジュール
例: ML 言語のモジュール

Short Quiz

質問。結局、「モジュラーなプログラム」、「モジュラリティの高いプログラム」、とは何だろうか？あなたなりの解答を書いてください。
解答の一例。

> 駄目な答え: プログラムを「モジュール」から構成すること。
> 言葉の言い換えに過ぎない答え: プログラムを独立性の高い部分から構成すること。
> ある程度は正しい答え: プログラムをいくつかの部品から構成し、各部品の使いかたの情報を全て公開すること。
> もう少しまともな答え: プログラムの各構成部分において、インタフェース (仕様) と内部の実装を分離し、前者を外部に見せ、後者を隠蔽する (隠す) こと。

ただし、一番最後の答えでも「プログラムを、どのくらいの個数の部品から構成すればよいか」については、何も言っていない。(部品をどう設計するかは、解くべき問題ごと、あるいは、その問題における「関心事」ごとに決めるべきであり、一律に、プログラムを、何個のどのような部品から構成するとモジュラーである、と決めることは、もちろん、できない。)