プログラム言語論-第 9 週-

亀山幸義

筑波大学コンピュータサイエンス専攻

筑波大学 情報科学類 講義, 2009 年 6 月 15 日

1 はじめに

2 プログラムの構造化と抽象化

3 論理型プログラム言語

概要

前々回 3 限　前回 2 限: modularity からオブジェクト指向までを駆け足で。
今回:

- プログラムの構造化と抽象化
  - データ抽象化とモジュールとオブジェクト
- 論理型プログラム言語
  - プログラムの仕様と実装
  - Prolog 言語

構造化プログラミング

E. W. Dijkstra, “Structured Programming”, 1969.

- プログラムは、トップダウン (top down) に構築するべき。
  - 最初は (データ構造のほかに) 大まかに何をやるかを決める。
  - 次に、その仕事をいくつかの subtask に分割する。
  - この詳細化 (refinement) のステップを繰返す。

ちなみに、以下の論文も興味深い。
E. W. Dijkstra, “GO TO Statement Considered Harmful.”, 1969.

具体例-1

「1000 番目までの素数を印刷するプログラム」

```
begin
  print first thousand prime numbers
end
```


具体例-2

「1000 番目までの素数を印刷するプログラム」

```
begin variable table p
  fill table p with first thousand primes
  print table p
end
```


具体例-3

「1000 番目までの素数を印刷するプログラム」

```
begin integer array p[1:1000]
  make for k from 1 through 1000
     p[k] equal to the k-th prime number
  print p[k] for k from 1 through 1000
end
```

In C:

```
void f () {
  int p[1000], k;
  for (k=1; k<1000; k++)
     p[k] = the k-th prime number
  for (k=1; k<1000; k++)
     print p[k]
}
```


モジュラリティ (再掲)

現実には、top-down の構造化のみではうまく行かない。

- データ構造の変更
- 部品間のインタフェースの変更

モジュラープログラミング (modular programming)

- インタフェース (interface) と実装の分離
- 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を指す (ことが多い)。
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
```


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。

ML Module vs Object

module と object の比較:

- 基本的な違い: module は内部状態 (OO 言語のインスタンス変数) を持たない。
- 抽象化: 同じ。
- 関数のルックアップ: module は静的, object は動的。
- 継承: module に継承はないが、実装の再利用は可能。
- サブタイピング: module にはサブタイピング機能はない。

サブタイピング

サブタイピングとは?

- B が A のサブタイプであるとき、
- A 型のオブジェクトを書ける (プログラム上の) 場所に、
- いつでも、B 型のオブジェクトを書いてよい。

Point, Circle を、OO 言語で書いていたら..

```
Point pnt = ...;
y = pnt.move(dx,dy);
```

というプログラムが書ければ、

```
Circle cle = ...;
y = cle.move(dx,dy);
```

というプログラムも書ける。
(ここで `move_p` と `move_c` は、`move` という同じ名前のメソッドとなっていると仮定。)

サブタイピングと多相型

OO 言語では:

- `move` メソッドが Point オブジェクトにも Circle オブジェクトにも適用可能。
- `move` メソッドは、Point クラスを継承した任意のクラスのオブジェクトに対して適用可能。
- 一種の多相型 (polymorphic type)。

ML の多相型

モジュールやオブジェクト指向とは関係ない。

```
# let rec append l1 l2 =
  match l1 with
    | [] -> l2
    | x :: l11 -> x :: (append l11 l2) ;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2;3] [4;5] ;;
- : int list = [1; 2; 3; 4; 5]
# append ["abc";"def"] ["gh";"ijkl"] ;;
- : string list = ["abc"; "def"; "gh"; "ijkl"]
```

どんな型の要素に対しても適用できる関数 (多相型をもつ関数) を定義することができる。(⇒異なる型ごとに関数を再定義する必要がなくなる。)

まとめ

- 大規模プログラミングへの対処としてのモジュラープログラミング
- その実現としてのモジュール
- オブジェクト指向との共通点、相異点

具体例として、ML 言語 (静的型システム+モジュール), Java などの OO 言語 (動的ルックアップを始めとする 4 つの特徴) を取りあげた。

Short Quiz (2 限)

以下の意見について、どう考えるか。
「プログラムの再利用」のためには、(emacs などの) エディタで、プログラムの一部をコピーして使えばいいのではないか？いまどきのエディタは、かなり高度な編集機能を持つので、コピーや改変は容易である。そうすれば、継承などのプログラム言語の機能を何も覚えずに、プログラム再利用ができて便利である。

論理型プログラム言語とは

- 「推論＝計算」という考えで設計された言語.
- プログラムは，事実，あるいは，推論規則 (事実から事実を導く) として記述.
- どのように推論するかの手順は，記述しない.
- 宣言型プログラム言語 (Declarative Programming Language) の一種.

Prolog プログラミング-1

プログラムは別紙参照.
Alice, Bob, Charlie, David, Eliza, Fritz, George, Hillary の 8 人の関係.

Prolog プログラミング-2

```
?- is_mother(X, charlie).
X = alice

?- is_mother(alice, X).
X = charlie n
X = eliza.

?- is_husband(alice, X).
false.

?- is_father(bob, X).
X = charlie n
X = eliza.
```


Prolog プログラミング-3

```
?- is_parent(X,eliza).
X = alice n
X = bob n
false.

?- is_grandparent(X,Y).
X = alice,
Y = george n
X = bob,
Y = george n
false.
```


Prolog プログラミング-4

```
?- is_relative(alice,X).
X = alice n
X = bob n
X = charlie n
X = eliza n
X = george n
false.

?- is_relative(eliza, X).
X = eliza n
X = fritz n
X = george n
X = alice n
X = bob n
false.
```


Prolog プログラミング-5

```
?- is_superrelative(alice, X).
X = alice n
X = bob n
X = charlie n
X = eliza n
X = george n
X = alice n
X = bob n
X = charlie n
X = eliza n
X = george n
X = alice n
X = bob n
X = charlie n
X = eliza n
X = george n
```


Prolog プログラミング-6

```
?- fact(10, X).
X = 3628800.

?-  append([a, b, c], [], X ).
X = [a, b, c].
?-  append([a, b, c], [d, e, f], X ).
X = [a, b, c, d, e].
?- append([a, b, c], X, [a, b, c, d, e, f]).
X = [d, e, f].
```


Prolog プログラミング-7

`append(X, Y, [a, b, c, d, e, f])` とやると何が返ってくるだろうか？

Prolog プログラミング-8

```
?- append(X, Y, [a, b, c, d, e, f]).
X = [],
Y = [a, b, c, d, e, f] n
X = [a],
Y = [b, c, d, e, f] n
X = [a, b],
Y = [c, d, e, f] n
X = [a, b, c],
Y = [d, e, f] n
X = [a, b, c, d],
Y = [e, f] n
X = [a, b, c, d, e],
Y = [f] n
X = [a, b, c, d, e, f],
Y = [] n
false.
```


Prolog プログラミング

宣言型プログラム言語 (Declarative Programming Language) の一種.

- 推論＝計算.
- プログラム＝事実，あるいは，推論規則.
- ゴール＝その事実が成立するかどうかを質問.
- 推論をする手順は，記述しない.

特徴.

- プログラムの目的=ゴールが成立するかどうか，を求める.
- 求解＝ゴール中の変数のある値に対して，ゴールが成立するか？
- 複数の解があることもある.
- あらゆる可能性を網羅的に探索する．双方向計算が可能.
- 一階述語論理のサブセット (Horn 節論理) に対応.

論理型プログラム言語

- Kowalski 1974 が提唱.
- Colmerauer 1973 が Prolog の最初の処理系.
- 日本の ICOT(第 5 世代コンピュータ・プロジェクト) が大きな貢献.
- 現在でも，知識表現，帰納論理プログラミングなどの分野で活躍.

Prolog = Programming in Logic

Short Quiz (3 限)

「事実やルール (推論規則) を記述するだけで，プログラムとして動く」という言語は，メリットもデメリットもある.

- メリット: どのルールをどう使ったら，ゴールとなる事実を導けるか (効率的に導くにはどうしたらよいか) を，プログラマは考えなくてよい．解決したい問題を正確に記述するだけでよい．(プログラミングが楽になる．)
- デメリット: 常に Prolog 処理系が決めた順序で解を探索するので，効率が必ずしも良くない．(効率を良くしようと思うと，処理系の動きを把握していないといけない．)

このような言語が，現実に有用な場面としてどんなものがあるか、想像して書きなさい.
参考: 「プログラムの仕様」がそのまま動くものを、executable specification (実行可能な仕様) という。

まとめ

- 「推論」を計算過程と見なしたプログラミング言語.
- その他にも様々な (驚くような) 仕組みを計算の原理に使ったものがあり得る。
  - 量子, DNA, ...