プログラム言語論
制御構造と型システム

亀山幸義

筑波大学 情報科学類

No. 6

制御構造

- (先週) 末尾再帰
- (今週) 制御構造，例外，継続

制御構造 FORTRAN

```
10  IF (X .GT. 0.000001) GO TO 20
    X=-X
11  Y=X*X-SIN(Y)/(X+1)
    IF (X .LT. 0.000001) GO TO 50
20  IF (X*Y .LT. 0.000001) GO TO 30
    X=X-Y-Y
30  X=X+Y
    ...
50  CONTINUE
    X=A
    Y=B-A+C*C
    GO TO 11
```

スパゲッティ・コード (Mitchell, “Concepts in PL”, 2003 より)

構造化プログラミング

Dijkstra, “GO TO CONSIDERED HARMFUL” (1968)

- go to 文を多用したプログラムは理解しづらい.
- かわりに，より「構造的」な制御文を使うべき.
- if-then-else, while, for, case ...

リストの要素の積を返す関数

```
open List;;  (* おまじない *)
let rec mult x =
  if x=[] then 1
  else (hd x) * (mult (tl x))
in mult [2; -3; 5]  ;;
```

hd はリストの先頭要素を取る関数.
tl はリストの先頭要素を除いた残り (リスト) を取る関数.
`mult [2;-3;5] => -30`

例外的な状況

mult 関数で、リストの要素に負の数があったら，(積ではなく) その要素を返すことにしたい.

```
let rec mult2 x =
  if x=[] then 1
  else if (hd x) < 0 then (hd x)
  else (hd x) * (mult2 (tl x))
in mult2 [2; -3; 5]  ;;
```

これではうまくいかない.

例外的な状況

```
let rec mult3 x =
  if x=[] then 1
  else if (hd x) < 0 then (hd x)
  else
     let res = mult3 (tl x) in
     if res < 0 then res
     else (hd x) * res
in mult3 [2; -3; 5]  ;;
```

再帰呼出しをするごとに,「例外的な状況が起きたかどうか」をチェックしないといけない.
⇒プログラムが読みづらくなるし，効率も悪い.

例外機構の利用

```
exception Negative of int ;; (例外の宣言)

try
  let rec mult4 x =
    if x=[] then 1
    else if (hd x) <= 0 then
        raise (Negative (hd x))
    else (hd x) * (mult4 (tl x))
  in
    mult4 [2; -3; 5]
with
  Negative n -> n ;;
```

raise (例外の発生) を実行すると，対応する try-with (例外の処理) まで一気にジャンプする.

例外機構: 別の例

map 関数の利用:

```
open List;;
let inc x = x +. 1.0;;
map inc [1.0; 2.0; 3.0];;
map sqrt [1.0; 2.0; 3.0];;
map sqrt [1.0; -2.0; 3.0];;
```

負の数があったら，例外を出したい.

```
exception Neg of float;;
let f r =
   if r < 0.0 then raise (Neg r)
   else sqrt r ;;
try
    map f [1.0; -2.0; 3.0]
with
    Neg r -> [r] ;;
```


例外機構

- 深い関数呼出しから一気に抜ける.
- 「内から外」の方向のみ可能.
- 例外の種類に名前を付け，発生した例外と一致する名前の「受け手」まで飛ぶ.
- 例外の発生と受け手は，動的に対応付けられる.

例外機構は動的束縛 (やや高度な内容)

```
exception E3;;
try
  let f x = raise E3 in
  let g x =
    try
       f 10
    with E3 -> 20
  in
    g 0
with E3 -> 30;;
```

上記の答は「20」である．(「30」ではない)

いろいろな言語の例外機構

例外機構〜大域脱出機構

- C: setjmp(), longjmp()
- C++: try-catch, throw
- Java: try-catch-finally, throw
- ML: try-with (または handle), raise

現代のプログラム言語では，例外機構を持つことはほとんど必須と考えられる。

継続 (continuation); 発展的な内容

実行時の「残りの計算」を表す概念.

```
let rec fact n =
  if n=0 then 1
  else n * (fact (n-1))
in fact 10
;;
let rec fact2 n k =
 if n=0 then k 1
 else
   fact2 (n-1) (fun x -> k (n*x))
in fact2 10 (fun x -> x)
```

fact2 は fact と同じ計算

```
fact2 10 (fun x -> x)
fact2 9 (fun x -> 10*x)
fact2 8 (fun x -> 10*9*x)
...
```


継続

例: `fact2 9 (fun x -> 10*x)` の第二引数
「残りの計算」を，関数で表すことにより，末尾呼び出しの形に変形できた．(「継続渡し方式」(Continuation Passing Style) のプログラム)
例外機構の表現:

```
let mult5 x =
  let rec f x k =
    if x=[] then (k 1)
    else if (hd x < 0) then
       (hd x)
    else f (tl x) (fun v -> k ((hd x)*v))
in f x (fun v -> v)
```

```
mult5 [1;5;-3]
=> f [1;5;-3] (fun v->v)
=> f [5;-3] (fun v->1*v)
=> f [-3] (fun v->5*1*v)
=> -3
```


継続

- 継続渡し方式: 継続を関数で表現したプログラミングスタイル.
- 一方，継続を直接扱うことにより、継続渡し方式と同等のプログラムを簡潔に書けるようにした言語もある.
  - call/cc (call-with-current-continuation; Scheme, SML/NJ, Ruby)
  - call/cc を使うと，例外機構以外に，バックトラックやコルーチンなど種々の制御を表現できる.

まとめ

- 構造化プログラミングとプログラムの制御構造
- 例外機構
- (継続)

問題:

- GO TO 文 (無制限のジャンプ命令) を乱用すると，なぜ，有害なのだろうか.
- 以下の C プログラムは、どういう点が意図通りでないか、説明せよ。

```
int mult2 (int* x, int len) {
  if (len==0) return 1;
  else if (*x < 0) { return *x;}
  else return *x * (mult2(x+1,len-1));
}
```

- (発展課題) Scheme の call/cc や Ruby の yield などの制御機構を調べ，特徴を述べなさい.

MiniML と OCaml の差

- MiniML/OCaml 言語では、int や bool の定数のほかに「関数」をあらわすデータがある。このような関数の型は、どうなるであろうか？
- ML 言語のプログラム中には、int や bool といった型名を書かないが、そんなことで大丈夫だろうか？
- OCaml では（100+"abc"）などの式は，コンパイル時にエラーとなる．(MiniML では実行時にエラーになる) この仕組みは？

型 (Type) とは?

「型」は「データの集合」の一種ではあるが、データの集合がすべて型になるとは限らない。

- コンピュータ (ハードウェア) で扱うことのできるデータの種類のこと。
- 同じ演算が適用できるデータの集まり。

型システム： どのようなプログラムにどのような型がつくか、定めるための体系。

具体的な型

基本型 (atomic type)

- 例: `int`, `bool`, `string`

複合的な型: 既にある型と型構成子 (type constructor) を使って構成。

- 例
  - C 言語: 構造体 (strucut)、共用 (union)、ポインタ、(関数) など。
  - ML 言語: 直積、レコード (record)、バリアント (variant), 参照、関数、リスト、再帰的な型など。

型の例 (関数の型)

```
(fun x→ x+1) :  int → int
(fun f→(fun x→ f(f(x+1)))) :  (int→ int)→(int→ int)
(fun x→ x) :  'a → 'a
(fun f → (fun x→ f (f x))) :  ('a → 'a) → ('a → 'a)
```

型推論の詳細は、3 学期「計算論理」の授業を参照。

実行時 vs コンパイル時

式 (1+"abc") が「いつ」エラーになるか。

- MiniC や MiniML では、実行時に (動的に) エラーになる。
- C や OCaml では、コンパイル時に (静的に) エラーになる。

エラーは静的に見つかる方がよい。

- 早い段階でエラーが見つかる。
- (実行に時間がかかる場合)、速く見つかる。

静的な型システム

式 (1+"abc") の整合性に関するエラーを、静的に発見したい。

- 式に「型」(type) を付け、その型を追うことによって整合性を検査する。
- 式の種類ごとに、どのような型が付くかを決めたものを「型システム」という。
- 型の整合性を検査するだけで、多くのバグを発見できる。
- 静的に検査ができていたら，実行時には検査は不要⇒実行時の効率が良くなる．

型システムの健全性 (Type Soundness):

- コンパイル時 (静的) に，型が整合したら，実行時の型の不整合 (実行時のエラー) は決して起きない。

型検査と型推論

型検査: 全ての変数 (や関数) の型が宣言されている言語で，型の整合性を検査すること．

- C 言語や Java 言語.
- 型検査は、変数や定数などのアトミックな式からはじめて、より大きな式の型が整合しているか検査する、という形式で行われる。

型推論: 変数 (や関数) の型が必ずしも宣言されていない言語で，その型を推論しつつ，型の整合性を検査すること．

- ML 言語や Haskell 言語.
- ML 言語では、「与えられた式に対して、最も一般的な型を推論する」という型推論アルゴリズムあり.

動的な型付け

- Lisp, Scheme, Ruby など。
- 実行時に型検査を行う。(`((lambda (x) (+ "abc" 100))` はエラーでない)
- 実行効率と，プログラムの理解のしやすさの観点からは、静的型システムに比べて不利。
- 静的な型システムで記述できないような、柔軟なプログラミングができる可能性がある。

多相型 (Polymorphism)

```
void swap (int *p, int *q) {
  int r;
  r = *p; *p = *q; *q = r;
}
```

swap 関数は (int *) 型だけでなく、どんな型でも使える。

```
void swap (T *p, T *q) {
  T r;
  r = *p; *p = *q; *q = r;
}
for any T.
```

```
# let swap (x,y) = (y,x) ;;
  - : 'a * 'b -> 'b * 'a = <fun>
```

多相型＝「任意の型」を含む型。

map 関数は多相的

```
open List;;

let inc x = x + 1;;
map inc [1; 2; 3];;
=> [2; 3; 4]

let add1 x = x ^ "1";;
map add1 ["kameyama1"; "yukiyoshi1"];;
=> ["kameyama1"; "yukiyoshi1"]
```

map の型: `('a -> 'b) -> ('a list -> 'b list)`

まとめ

- 制御構造: 例外、継続
- 型システム: 静的 vs 動的、型検査と型推論

プログラム言語の型システム

	C/C++	Lisp	ML,Haskell	Java	Ruby,JavaScript
静的/動的	静的	動的	静的	静的	動的
検査/推論	型検査	—	型推論	型検査	—

- 静的型システムと動的型システム
  - 静的:実行前に型の整合性を検査/推論。
  - 動的:実行時に行う。
- 型検査/型推論
  - 型検査: 変数の型は宣言済み⇒プログラムの型の整合性を検査。
  - 型推論: 変数の型が未知⇒推論しつつプログラムの型の整合性を検査。

ML 言語の多相型 (polymorphic type)

map の型: ('a -> 'b) -> ('a list -> 'b list)

```
let inc x = x + 1;;
map inc [1; 2; 3];;
=> [2; 3; 4]
```

(map の型は、(int -> int) -> (int list -> int list))

```
let add1 x = x ^ "1";;
map add1 ["kameyama1"; "yukiyoshi1"];;
=> ["kameyama1"; "yukiyoshi1"]
```

(map の型は、
(string -> string) -> (string list -> string list))

ML 言語の多相型 (polymorphic type)

ユーザ定義関数における多相型; let で導入される。

```
let f1 (x,y) = (y,x);;
==> f1 : 'a * 'b -> 'b * 'a
let f4 x = x in ((f4 10), (f4 "abc"))
==> f4 : 'a -> 'a
```

ちなみに、以下の式は ML では、多相型と見なされない。

```
(fun f4 ->  ((f4 10), (f4 "abc"))) (fun x -> x)
==> type error
```


多相型の利点

もし、map 関数を C 言語で書くとしたら‥。

- 方法 1. int 型に対する map, string 型に対する map などを別々に定義する。
- 方法 2. 「void 型に対する map」を定義して、使うときに各型に cast する。
- 方法 3. C++ の template を使う。「T 型に対する map」を定義して、この関数を使うときに T を具体化する。

方法 1 は、コード量が多くなる、同じコードを何度も書くため保守性が悪い、等のデメリットがある。
方法 2 は、型の検査を素通りするため、型に関する間違いのチェックができなくなる等のデメリットがある。
方法 3 は、多相型と基本的に同じ効用がある。ただし、C/C++言語自体に組み込まれた機能ではないので、型エラーが起きたときに原因となるコードを発見しにくい等のデメリットがある。

多相型の利点

- 前のスライドで挙げた問題点がない。

今回学んだ多相型は、parametric polymorphism と呼ばれるもの。ML 言語のほか、Haskell などの関数型言語で利用可能。

まとめ

- 型システム
- 静的型付け vs 動的型付け
-

Short Quiz の問題: 以下の関数 f と g を型付けしてみよ。

```
# let f x y = x (x (y+10));
# let g x y z = (x z) (y z) ;;
```