プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 4 (関数型言語その 1)

授業の流れ

ここまで:

- コンパイラ、インタープリタ
- プログラム言語の構文論と意味論、抽象機械
- ブロック構造言語の基本, 束縛と環境、評価順序, miniC 演習

これから (嘘です):

- miniC 言語の意味論、抽象機械 ???

本当のこれから:

- ラムダ計算と関数型言語
- 関数型言語の意味論、抽象機械
- ヒープとメモリ管理
- (ほんの少しだけ)miniC 言語の意味論、抽象機械

理由: C 言語の意味論の方が、関数型言語の意味論より難しい。

関数型言語

- ラムダ計算 (=「関数」概念を追究した体系) に基づくプログラム言語たちのこと
- 例. Scheme, Lisp (Common Lisp etc.), ML (SML, OCaml), Haskell
- 関数型言語の機能は Ruby など、他の言語が取りいれている。
- 例. 関数クロージャ(C++など), Java generics, map/reduce,...

ラムダ計算 (λ-calculus)

- 関数の入力と出力を明記する記法
- 「$f(x) = x^2 + 5x$ となる関数 f」 を,「$\lambda x.\ x^2 + 5x$」と表す. (無名関数, 匿名関数)
- 「上記の f に引数として 10 を与えた結果 (値)」を「$f\ 10$」あるいは「$(\lambda x.\ x^2 + 5x)\ 10$」と書く.
- つまり, $f\ 10 = (\lambda x.\ x^2 + 5x)\ 10 = (10^2 + 5)10$ が成立.
- 高階関数 (higher-order function): 関数を引数としてもらったり, 返す値にしたりする (高いレベルの) 関数, (数学では「汎関数」と言うこともある.)

関数型言語

C 言語などの手続き型 (あるいは命令型) 言語と比較したときの関数型言語の特徴:

- ラムダ計算に基づく．つまり,「関数」概念に基づく．
- 単一代入が基本． 参照透明性 (referential transparency)
- 意味論が明快・簡潔で検証しやすい
- 簡単な割に実は強力; 高階関数，データ型
- 得意な分野: 種々のアルゴリズムの記述，プログラム言語処理系，記号処理システム (不定長データの複雑な処理)
- 不得意な分野: 固定長データの数値計算，高性能計算

関数型言語

- Lisp: 古くからある関数型言語，人工知能システムや数式処理システムなどの記述言語．
- Scheme: Lisp の意味論を洗練したもの．
- ML (Meta-Language): 関数型言語の一族の名前，SML, OCaml などがある．最も成功した関数型言語．
- ほかには，ErLang (企業が実際に利用), Haskell (研究者が作った言語)，F#(MicroSoft の ML-like な言語) など．

miniML は OCaml のサブセットとして設計．Scheme や Haskell などとは構文は異なるが，それらのサブセットと思うことも可能．

関数型のプログラミング・スタイル1

手続き的スタイル: 繰返し (for, while,...)

```
int fib (int n) {
  int i, tmp;
  int x=1, y=1;
  for (i=2; i<n; i++) {
    tmp = x;
    x = y;
    y += tmp;
  }
  return y;
}
```

関数的スタイル: 再帰呼出し

```
let rec fib n =
  if n<=2 then 1
  else fib(n-1) + fib(n-2)
```


関数型のプログラミング・スタイル2

手続き的スタイル: 変数への値の代入

```
int foo (int x) {
  int y;
  y = x + goo(x+1);
  y += hoo(y*y);
  y = goo(y+2);
  ...
  return y;
}
```

関数的スタイル: 局所的な変数束縛

```
let foo x =
  let y = x + goo(x+1) in
  let y = y + hoo(y*y) in
  let y = goo(y+2) in
  ...
    y
```

関数型言語は単一代入だが,「異なる変数宣言」に対しては，それぞれ代入できる．

関数型のプログラミング・スタイル3

高階関数の例:

map 関数の利用

```
let foo a b lst =
  List.map
   (fun x -> x*a+b) lst
in
  foo 10 20 [1; 2; 3; 4; 5]
==>
  [30; 40; 50; 60; 70]
```

自前で定義する

```
let rec goo f n x =
  if n=0 then x
  else f (goo f (n-1) x)
in
  goo (fun x -> x + 10) 5 2
==>
  52
```


関数型のプログラミング・スタイル4

複数の値を返す関数

```
let rec fib n =
  if n<=1 then (1,1)
  else
    let (x,y) = fib(n-1)
    in (y,x+y)
in fib 5
==>
  (5,8)
```

(a1,a2,..,an) は，n 個組 (tuple, タプル) のデータ型.
`let (x,y) = e1 in e2` は，e1 の値が 2 個組 (対) で，その第 1 要素を変数 x にとり，第 2 要素を変数 y にとって e2 の計算をおこなう.

副作用 (side effect)

- 「主たる作用」以外の全て.
- 関数の場合、その主たる仕事は「値を返す」こと.
  - 例 1: 変数の値を変更する (状態の変更)
  - 例 2: ファイルに対して読み書きする (IO)
  - 例 3: プログラムの制御を変更する (ジャンプする)
- 手続き型言語のプログラムは，副作用にあふれている.
- 関数型言語のプログラムは，どこで副作用を使うかが明示される.
- 「副作用」は悪いイメージ; 効果 (effect) ともいう.

副作用がなければ，プログラムの理解・解析・変換は簡単.

- f(e1,e2) で，e1 と e2 のどちらから計算しようと同じ.
- e1+e1 = e1*2 が成立.

関数型言語の処理

3 つの大きな疑問.

- 関数をデータとして扱っているが，その処理の仕組みは？
- データ型を多用することになるが，その処理の仕組みは？
- 繰返し構文に比べて，再帰呼出しは効率が悪いのでは？