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)

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:

- ラムダ計算と関数型言語，意味論
- ヒープとメモリ管理
- 手続き型言語の意味論

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？

「対」のデータ型の利用 (1)

Fibonacci 関数の素朴な定義:

```
let rec fib n =
  if n < 2 then 1
  else (fib (n-2)) + (fib (n-1))
in fib 5
==> 8
```

Fibonacci 関数の改善した定義:

```
let rec fib n =
  if n = 0 then (1,1)
  else
    let p = fib (n - 1) in
      (snd p, fst p + snd p)
in fst (fib 5)
==> 8
```


「対」のデータ型の利用 (2)

最大公約数を求める関数:

```
let rec gcd m n =
  if m = n then m
  else if m > n then
    gcd (m-n) n
  else
    gcd (n-m) m
in gcd 100 35
==> 5
```


「対」のデータ型の処理 (1)

OCaml プログラム

```
let add p q =
   (fst p + fst q,
    snd p + snd q)
in let v = (10,20)
in add v v
==> (20,40)
```

C プログラム: NG!

```
int* add (int *p, int *q) {
  int r[2];
  *r = *p + *q;
  *(r+1) = *(p+1) + *(q+1);
  return r;
}
int main () {
  int v[2];
  int *q;
  v[0] = 10; v[1] = 20;
  q = add(v, v);
  printf ("(%d,%d)\n", *q, *(q+1));
}
```


「対」のデータ型の処理 (2)

OCaml プログラム

```
let add p q =
   (fst p + fst q,
    snd p + snd q)
in let v = (10,20)
in add v v
==> (20,40)
```

C プログラム

```
int* add (int *p, int *q) {
  int* r =
   (int*)malloc(sizeof(int)*2);
  *r = *p + *q;
  *(r+1) = *(p+1) + *(q+1);
  return r;
}
int main () {
  int v[2];
  int *q;
  v[0] = 10; v[1] = 20;
  q = add(v, v);
  printf ("(%d,%d)\n", *q, *(q+1));
}
```

malloc for memory allocation.

データ構造

MiniML 言語の「対 (ペア)」データ型を使えば、様々なデータ構造を表現できる。C 言語で同様なことをするためには、構造体 (struct 型) などを使う。

MiniML 言語では、一度作ったペア (のためのメモリ) は、どうなるだろうか？ ペアを大量に作り続ける関数を作成して呼びだして、実験せよ。
MiniML 言語処理系において「ペアなどのデータを覚えておくためのメモリ領域」は、スタック上に取られるかどうか考えなさい。

プログラム例

対 (ペア) を作る操作を多数回繰返すプログラム:

```
let limit=10000000 in
  let rec f x =
     if x =limit then "ok"
     else let _ = (x,x+1) in f (x+1)
  in f 0
```

ただし _ というのは、「無名の変数」のこと (後で使わない変数)。

これだけ繰返しても、エラーは起きず、計算が正常に終了する。

ペア $(x, x+1)$ のデータは、スタック上に取られるのではない。

ペア $(x, x+1)$ のデータを格納するためのメモリ領域は、このペアが不要になったら自動的に回収され、他のペアのために再利用される。

プログラム実行時のメモリ (再掲)

Register

Program Counter

Code

Environment Pointer (スタックを指す)

Data:

- Stack (スタック)
- Heap (ヒープ)

ヒープ (heap)

データを生成した関数が終了しても、そのデータが使われる可能性があるものを格納するためのメモリ。(永続的なデータ).

ヒープに格納されるデータ (例)

- 対 (ペア) など構造を持つデータ
- 文字列
- リスト
- 関数 (正確には関数クロージャ)
- オブジェクト指向言語でのオブジェクト
- その他、固定長のメモリ (通常は、32bit や 64bit) にはいりきらないデータ

x と y の対の生成の処理

- 対のためのメモリをヒープから取得する.
- その場所に (x, y) を書きこむ.
- それへのポインタ (x へのポインタ) を返す.

ヒープ

C 言語ではプログラマがメモリ管理を行う。(ヒープ上のメモリ確保 malloc(), メモリ解放 free())

OCaml などの言語では、処理系が自動的にやってくれる。(対を実行すればメモリが確保され，対を利用することがなくなれば，いつの間にかメモリが解放される．)

論理式、数式、プログラム、XML データなど、構造のあるデータ (特に、可変長のデータ) を扱う際には、上記の 2 機能があるプログラム言語を使うことは、ほぼ必須。

ヒープは、後で回収することも考えると、単なる「配列」状ではなく、「リンクをもつリスト」状の構造を持つ。(長く使い続けていると、ヒープ全体が「使用領域」と「未使用領域」による「まだら模様」になっていく。)

ごみ集め (Garbage Collection)

ヒープ領域において、使われなくなったデータに対応する領域を回収する (空き領域に連結する) 操作。

通常は、ヒープが足りなくなったときに、一斉に回収する。

処理速度が大事; 良いごみ集めアルゴリズム採用することが必須。

古くから研究されているが、現在でも更なる改善のための研究がなされている。(高速性とともに、「絶対に間違いがあってはいけない」という意味で、正当性も非常に大事。)

データ型の意義

C 言語でのデータ型

- 比較的少数 (配列，ポインタ，struct 型など).
- メモリ確保と解放はプログラマの責任.
- 低レベルの詳細なコントロールが可能.

OCaml 等でのデータ型

- 非常に豊富なデータ型を用意 (関数，直積，レコード，バリアント，帰納的データ型など)
- メモリ確保と解放は自動.
- 「ごみ集め」の仕組みが必要.

※ C 言語は自前でメモリ管理をしたいプログラマ向け，OCaml 等はメモリ管理をシステム (処理系) に委ねたい人向け.

まとめ

データ構造，永続的なデータ，ヒープ

メモリ管理

関数をデータとして扱う-1

プログラム言語における「第一級のもの (first-class citizen)」

通常のデータと同様に扱われるもの．変数の値になったり，関数の引数や返り値になれるもの．

C 言語: 整数などのほか，ポインタが first-class.

Java: 整数などのほか，オブジェクトが first-class.

OCaml, Haskell など: 整数などのほか，関数が first-class.

Scheme: 整数やシンボルのほか，S 式が first-class.

関数をデータとして扱う-2

処理系内部では,「関数を表す式を計算した結果の値」が必要.

動的束縛の場合 (昔の Lisp, 今の emacs lisp)

$\lambda x.e$ を計算した結果は，$\lambda x.e$ そのものでよい．

静的束縛の場合 (ほとんどの関数型言語)

$\lambda x.e$ を計算した結果は，$\lambda x.e$ そのものではいけない．

C 言語の処理で，静的束縛では，access link が必要だった．

関数をデータとして扱う-3

例題:

```
let x=1 in
let f=(fun y-> x+y) in
let x=2 in
  f 3
```

上記プログラムの処理 (誤ったバージョン):

frame4: y=3
frame3: x=2
frame2: f=fun y->x+y
frame1: x=1
global:

返り値: 5

ここで frame4 における access link が指すのは，frame3 でなく，frame1 でなければならない．

関数クロージャ(関数閉包，function closure)

静的束縛の関数型言語で，実行時に用いられる.「関数を計算した結果 (値)」を表す.

関数の定義と，環境 (スタックフレームへのポインタ，あるいは，変数ごとにその値を決めるもの) をセットにしたもの: $(\lambda x.e,\ \sigma)$.
ここで，σ は，環境 (へのポインタ) で，将来この関数の本体 e が実行されるときの access link となる．

要するに，この関数を作ったときの環境を保存しておく，ということ．

参考: closure とは，閉じたもののこと．関数 `fun x->x+y` は，変数 y が自由変数になっているので，その値とセットにして，はじめて「閉じる」ことができる．(自由変数が 1 つもない式のことをラムダ計算では，closed term と言う．)

関数クロージャを用いた処理 (2)

```
let x=1 in
let f=(fun y-> x+y) in
let x=2 in
  f 3
```

上記プログラムの処理 (正しいバージョン):

式 (fun y→ x+y) の値は関数クロージャ: clo(fun y→ x+y, 環境へのリンク)

(f 3) の計算では、関数クロージャに保存されていたリンクが, access link となる.

frame4: y=3, access link ⇒ frame1	返り値: 4
frame3: x=2	返り値: 4
frame2: f=clo(fun y->x+y,frame1)	返り値: 4
frame1: x=1	返り値: 4
global:	

関数クロージャを用いた処理 (3)

もう一工夫しないと、うまく動作しない。

```
(let x=1 in
 let f=(fun y-> x+y) in
 let x=2 in
  f) 3
```

frame4: y=3, access link ⇒ frame1	返り値: clo(...)
frame3: x=2	返り値: clo(...)
frame2: f=clo(fun y->x+y,frame1)	返り値: clo(...)
frame1: x=1frame1': y=3, access link ⇒ frame1	返り値: clo(...)
global:	

NG! frame1 はもはや存在しない。

関数クロージャはどこにあるか？

```
let f =
  let foo x =
    fun y-> x+y
  in
    foo 10
in
  f 20
```

関数クロージャは，C 言語の関数と違い，プログラム実行時に (動的に) 生成される.

関数クロージャは，スタックに積まれるのではない．(それを生成した関数呼出しが終わった後も行き残る．下記プログラムを参照)

関数クロージャは，ヒープに置かれる.

C 言語の「高階関数」?

質問. C 言語でも、「関数へのポインタを引数とした関数」や「関数へのポインタを返す関数」は書ける。では C 言語も関数型言語か?

No. C 言語では、「関数を生成する」ことは (通常は) できない。

関数型言語では、関数を動的に生成して、(計算結果として) 返すことができる。

```
let fun f x = (fun y -> x + y)
```

(参考) オブジェクト指向言語では、オブジェクトを動的に生成して、(計算結果として) 返すことができる。

今日のまとめ

関数プログラミング: 単一代入，再帰呼出し，高階関数，データ型の活用

単一代入⇒副作用の分離・明示

高階関数と静的束縛⇒関数クロージャ

データ型⇒ヒープ

再帰呼出し⇒末尾再帰

例

```
let limit=10000000 in
 let rec f x =
    if x=limit then "ok"
    else let _ = (x,x+1) in f (x+1)
  in f 0
```

このプログラムを OCaml で実行しても、overflow のエラーを起こさない
理由: 「ペア $(x, x+1)$ は heap に格納される」
その理由だけでは説明がつかない。

> スタックからペア $(x, x+1)$ へのポインタがあるなら、(スタックがまきもどされない以上) このペアも、「ごみ集め」の対象にならないのでは？
>
> 確かにそうだが、_ という特殊な変数は値を記憶しない (スタック上に変数の領域が取られない)。

以前の例 (続き)

「対」が heap に取られてごみ集めの対象となってメモリ領域が回収されるとしても、stack に積まれる stack frame が、関数呼び出しごとに 1 つずつ多くなり、いつか stack overflow になるのでは？

その「からくり」を考えるのが、この章の目的。

関数呼出し時のスタック-1

```
let rec f x =
 if ... then ...
 else f (x+1)
in f 0
```

```
                 ---
                 x=2   ...
           ---   ---
           x=1   x=1   ...
     ---   ---   ---
     x=0   x=0   x=0   ...
---   ---   ---   ---
```

これでは、いつか、stack overflow になる。

関数呼出し時のスタック-2

```
let rec f x =
 if ... then ...
 else f (x+1)
in f 0
```

```
     ---   ---   ---
     x=0   x=1   x=2   ...
---   ---   ---   ---   ---
```


末尾呼び出し

関数 f の本体で、関数呼出し (g e) を行なうとき、(g e) の結果が、そのまま関数 f の結果となるとき、この関数呼出しを末尾呼出し (tail call) と言う。

末尾呼出しは、「それより後で関数 f の計算はない」ので、関数 f (の現在の呼出し) に対する stack frame は消してしまってよい。(while ループ等と同じ処理)

末尾呼出しでない例:

```
let rec f x = if x=0 then 1 else x * f (x-1)
let rec f x = if x=0 then 0 else f (x-1) + 0
```

末尾呼出しの例:

```
let rec f x = if x=0 then 1 else f x
let rec f x y = if x=0 then y else f (x-1) (x*y)
```


関数型言語と末尾呼び出し

C 言語等における「ループ」を、関数型言語では、通常、「関数の再帰呼出し」で実現する。

再帰呼出しは、ループよりも表現力が高いが、その反面、実行効率がわるい (stack に stack frame をどんどん積む必要があるため。)

しかし、再帰呼出しが末尾再帰であれば、(また、処理系が末尾再帰最適化を組みこんでいれば)、コンパイラが、「ループ」として実現するので、実行効率はループと同等になる。

多くの関数型言語は、末尾再帰の最適化を組みこんでいる。
(Scheme, SML/NJ, OCaml ..... ただし C 言語処理系は通常，末尾再帰の最適化はしない．)

まとめ

関数プログラミング: 単一代入，再帰呼出し，高階関数，データ型の活用

単一代入⇒副作用の分離・明示

高階関数と静的束縛⇒関数クロージャ

データ型⇒ヒープ

再帰呼出し⇒末尾再帰