プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 6： 高階の関数型言語

目次

先週の演習

- これまで講義した評価器を実際に使ってみて理解を深める。
- コンパイル (変数のインデックス)
- 静的束縛 vs 動的束縛
- 値呼び vs 名前呼び (vs 必要呼び)

de Bruijn index

```
let x = 10 in
x + (let y = 20 in
      x + y + (let z = 30 in
                x + y + z))
```

de Bruijn index

```
let x = 10 in
x + (let y = 20 in
      x + y + (let z = 30 in
                 x + y + z))
```

```
let * = 10 in
var0 + (let * = 20 in
         var1 + var0 + (let * = 30 in
                        var2+var1+var))
```

「内側から何番目の let で束縛されているか」を表す。

de Bruijn index

```
let x = 10 in
x + (let y = 20 in
      x + y + (let z = 30 in
                x + y + z))
```

それぞれの変数の値を環境から lookup するとき、de Bruijn index は、「環境の先頭から何番目にその変数の値があるか」を示す。
変数 x を表す index は変化する。

`[("x",10)]`	x は 0 番目
`[("y",20);("x",10)]`	x は 1 番目
`[("z",30);("y",20);("x",10)]`	x は 2 番目

de Bruijn index

```
let x = 10 in
x + (let y = 20 in
      x + y + (let z = 30 in
                x + y + z))
```

```
let * = 10 in
var0 + (let * = 20 in
         var1 + var0 + (let * = 30 in
                        var2+var1+var))
```

環境に、変数名をいれておく必要がなくなる。

```
[10]          x は 0 番目=var0
[20;10]       x は 1 番目=var1
[30;20;10]    x は 2 番目=var2
```

de Bruijn index

de Bruijn index は実装だけでなく、理論的にも有用：

- 表現の一意性: `let f x = x+10` と `let g y = y+10` とが、同じ形で格納される。
- 束縛変数の衝突 (による面倒) を避けられる
- 静的束縛の場合のみ (変数が、環境において何番目かが静的に決まらないと、インデックスの値も決まらない)

参考: de Bruijn level (束縛変数が、「外から数えて」何番目にいるかを表す)

目次

Sestoft 教科書の言語たち

- micro ML (4 章): 一階、関数型言語
- 名前なし (5 章): 高階、関数型言語 (今日)
- micro C (7 章): 命令型言語
- オブジェクト指向言語

目次

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

- 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, $f\ 10 = (\lambda x.\ x^2 + 5x)\ 10 = 10^2 + 5 \cdot 10$ が成立.

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，$f\ 10 = (\lambda x.\ x^2 + 5x)\ 10 = 10^2 + 5 \cdot 10$ 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:

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．

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)

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

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

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

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)

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

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.

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.

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, F# どがある.

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, F# どがある.
- 他のプログラミング言語にも影響 (Scala, JavaScript, ...)

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

単一代入: 変数に対する束縛は 1 回限り

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
```

単一代入でも,「異なる変数」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2'

C 言語でできること:

```
int x = 10;
if (...) {
    x = x + g(1);
}     /* else なし*/
```

この x が x=10 なのか、x=10+g(1) かわからない。
(これは単一代入の言語ではできない。)

関数型のプログラミング・スタイル2''

C 言語でできること:

```
int x = 10;
void foo () {
    x = x + h(5);
}
void goo () {
    printf ("%d\n", x);
}
int main () {
    foo();
    goo();
}
```

x=10 なのか、x=10+h(5) なのかわからない。(これは単一代入の言語ではできない。)

関数型のプログラミング・スタイル2''

ML(OCaml) の「参照」(C の破壊的変数 (mutable variable) に相当する)

```
let r = ref 0 in
  Printf.printf "%d\n" !r;
  r := !r + 3;
  Printf.printf "%d\n" !r;

let foo x  =
  let r = ref x in
    r
in let goo r  =
    r := r + 1;
in
  goo (foo 10)
```

C 言語で、メモリを確保 (malloc) してそのポインタを返したものと、ほぼ同じ。

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]
==>
  [30; 40; 50; 60]
```

map 関数は他のものにも使える

```
List.map
 (fun x -> x ^ ".ml")
 ["foo"; "goo"; "hoo"]
==>
["foo.ml"; "goo.ml"; "hoo.ml"]
```

関数型のプログラミング・スタイル4

高階関数を定義する

```
let foo1 f g x =
  g (f x)
let foo2 f g =
  fun x -> g (f x)
let foo3 =
  fun f -> fun g ->
    fun x -> g (f x)
```

高階関数を定義する

```
let rec goo n f x =
  if n=0 then x
  else goo (n-1) f (f x)
in
  goo 5 (fun x -> x + 10) 7
==>
  57
```

演習

問題: 以下の関数は何をするものか言葉で答えなさい。

```
let rec foo f n x =
  if n = 0 then x
  else f (foo f (n − 1) x)
```

ヒント: foo の第一引数 f として、let add1 x = x + 1 として定義された関数 add1 や let times2 x = x * 2 として定義された関数 times2 をいれたものを想定してみなさい。
オプション問題: 自然数 m,n に対して、foo (foo add1 m) n 0 を計算すると何が返るだろうか？

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 (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？

今日は 1、3 番目だけ。(2 番目の回答はあとで。)

目次

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: 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.
- ML, Haskell など: 整数などのほか，関数が first-class.
- Lisp/Scheme: 整数やシンボルのほか，S 式が first-class.

```
let x = fun y -> y * 5 ;;

let foo x = if x > 100 then
              fun y -> y - x
            else
              fun y -> 91 ;;
```

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.

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.
- → 関数クロージャ

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環境として使われる。
- 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

```
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:

- 式 (fun y→ x+y) の値は関数クロージャ: Closure(fun y→ x+y, [x=1])

```
Call(f,3)  in  [x=2; f=Closure(..); x=1]
=> Call(Closure(fun y->x+y,[x=1]),3)
     in [x=2; f=Closure(..); x=1]
=> x+y  in [y=3; x=1]
```

関数クロージャはどこにあるか？

スタック機械で、処理系が動いているとき、関数クロージャがスタックに書きこまれているとすると。。。

```
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?

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。

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)
```

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。

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
- クロージャなどの構造をもったデータの置き場所⇒ヒープ

計算機演習

- Sestoft 教科書 5 章を、少しだけ変更したファイルが chap5.ml として置いてある。これをつかって、高階関数の処理がどうなるか、自習せよ。

目次

再帰と繰返し

関数型言語では、繰返しではなく再帰を多用する。

```
let rec sum n =
  if n = 0 then 0
  else n + sum (n - 1)

int sum (int n) {
  int i, res = 0;
  for (i = 0; i <= n; i++)
     res += i;
  return res;
}
```

再帰の効率は？

- 関数呼び出しをするごとに、「その呼び出しから戻ってきたあとの計算」の情報を覚えないといけない。
- 繰返し処理なら、100000 回ループしても問題ないが、再帰では、スタックがあふれて、処理が中断してしまう ???

末尾呼び出し、末尾再帰

Tail call: 関数呼び出しが、処理の「末尾」のみであるもの：

```
let rec sum n =
  if n = 0 then 0
  else n + sum (n - 1)  ;; NG
let rec sum2 n m =
  if n = 0 then m
  else sum2 (n - 1) (n + m) ;; OK
let rec power n x =
  if n = 0 then 1
  else if (even n) then square(power (n/2) x)
  else x * (power (n-1) x)  ;; NG
let rec power2 n x m =
  if n = 0 then m
  else if (even n) then
       power2 (n/2) (square x) m
  else power2 (n-1) x (n*m) ;; OK
```

末尾呼び出し、末尾再帰

末尾呼び出しの処理の最適化:

- 関数呼び出しのあと、「戻ってくる」必要がない。
- →繰返しと同様の処理が可能となる。(スタックを消費しない。)

演習ファイル: chap4.ml,chap5.ml における ex3 など。
現実のプログラム言語の処理系:

- Scheme: 処理系は必ず末尾再帰の最適化
- ML など多くの関数型言語: 末尾再帰の最適化をするのが普通
- C,JavaScript など： 末尾再帰の最適化は必ずしもしない

→ 自分の好きな言語の処理系で、「末尾再帰を 1 億回おこなう」コードを書いて試してください。(同じ言語でも、処理系ごとに違うことがあります。)

継続渡し方式

高階関数を用いた、ある種のプログラムの記述方法:

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)  ;; NG

let rec fact_cps n k =
  if n = 0 then k 1
  else fact_cps (n-1) (fun x -> k (n * x))  ;; OK

fact_cps 5 (fun x -> x)
=> fact_cps 4 (fun x -> 5 * x)
=> fact_cps 3 (fun x -> 5 * 4 * x)
=> ...
=> fact_cps 0 (fun x -> 5 * 4 * 3 * 2 * 1 * x)
=> 5 * 4 * 3 * 2 * 1 * 1
```

末尾再帰になっている。

継続渡し方式

前ページのプログラムへの疑問：

- k は何だろうか？
- 普通のプログラムを、継続渡し方式にするのはどうしたらいいだろうか？
- 何のメリットがあるのだろうか？

→ 教科書 11 章を参照。

継続

継続 (continuation): プログラム実行時の、「残りの計算」を表す概念。

```
let rec fact n =
  if n = 0 then 1
  else n * fact (n-1)  ;; NG
```

fact 5	この時点での継続 []
5 * (fact 4)	この時点での継続 5 * []
5 * 4 * (fact 3)	この時点での継続 5 * (4 * [])

継続渡し方式

継続渡し方式: Continuation-Passing Style

- 「継続」を関数として表現して、明示的に表す。
- もともとの関数は、「継続」を表す引数を取る高階関数となる。

```
let rec fact_cps n k =
  if n = 0 then k 1
  else fact_cps (n-1) (fun x -> k (n * x))  ;; OK
fact_cps 5 (fun x -> x)
=> fact_cps 4 (fun x -> 5 * x)
=> fact_cps 3 (fun x -> 5 * 4 * x)
=> ...
=> 5 * 4 * 3 * 2 * 1 * 1
```

CPS の利点:

- 末尾再帰、 全ての途中結果に名前が付いている
- →コンパイラの中間言語 (Appel, “Compiling with Continuations”)
- さまざまな制御を表現可能

継続渡し方式での制御

```
let rec sqrt_multiply lst =
  match lst with
  | [] -> 1
  | h :: t -> (sqrt h) * (sqrt_multiply t)

let rec sqrt_multiply_cps lst k =
  match lst with
  | [] -> k 1
  | h :: t ->
     if h >= 0 then
       sqrt_multiply_cps t
           (fun x -> k ((sqrt h) * x)
     else -1
```

リスト中に負の数があると、その途端に (残りの計算をやらずに) 抜けだす→ ML や Java の exception に相当することを、特別なしかけなしに実現。