プログラム言語論

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

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
- 特徴
  - 「文」ではなく「式」が中心.
  - 単一代入 (代入文はないか，あっても制限がかかる)
  - 再帰呼出しが基本 (「繰返し」構文はあまり使わない)
  - 高階関数が使える.
  - 強力なデータ型，静的型システムを持つ (ものが多い).

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？

関数をデータとして扱う-1

第一級 (first-class) のオブジェクト (もの，要素)

- 通常のデータと同様に扱われるもののこと．
- C 言語では，配列は first-class.
- C 言語や Java 言語では，関数は first-class ではない．
- Java 言語では，(オブジェクト指向の意味での) オブジェクトは first-class.
- 関数型言語では，関数は 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, 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):

- この関数が `f 3` において使われるとき，(fun y→ x+y) に対応するスタックフレームがスタックに積まれる．

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

- ここで frame4 における access link のリンク先は，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)

- 関数が第一級であって，静的束縛であるプログラム言語における「計算結果 (値) としての関数」のこと。
- 具体的には、関数の定義そのものと環境 (あるいは状態) をセットにしたもの: (`fun` x `->` e, σ). ここで，σ は，環境 (へのポインタ) で，将来この関数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

```
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 (fun y→ x+y, x=1 のフレームへのリンク)
- (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
frame3: x=2,
frame2: f=(fun y->x+y, frame1 へのリンク)
frame1: x=1
global:

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。

「対」のデータ型

C プログラム

```
int* mk_pair (int x, int y) {
 int *p;
 p=(int*)malloc(sizeof(int)*2);
 *p = x;
 *(p+1) = y;
 return p;
}
...
  q = mk_pair (10, 20);
...
  free(q);
```

OCaml プログラム

```
let q = (10,20) in
 ...
```

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.

データ型とヒープ

プログラム言語におけるデータ型は、(通常) ヒープを用いて実現される。ヒープについては、前回の授業資料 (講義では説明しきれなかった部分) を参照。

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。

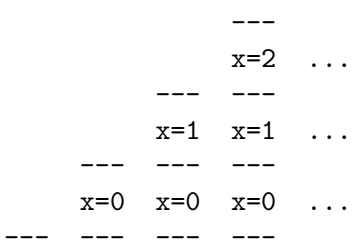
- スタックからペア $(x, x+1)$ へのポインタがあるなら、(スタックがまきもどされない以上) このペアも、「ごみ集め」の対象にならないのでは？
- 確かにそうだが、_ という特殊な変数は値を記憶しない (スタック上に変数の領域が取られない)。

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

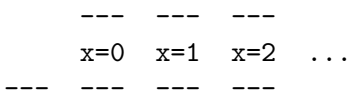
```
let rec f x =
 if ... then ...
 else f (x+1)
in f 0
```



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
```



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．)

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

来週: 制御構造と型システム