# プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 3: OCaml プログラミング、一階の関数型言語



## 目次





## OCaml における再帰関数の定義

「再帰関数」は多くの言語が持つ。
C, Java, Scheme, OCaml, Haskell etc.

例: 与えられた正の整数に対して、f(1)=1 で、$x > 1$ に対しては、「x が偶数なら 2 で割り、奇数なら 3 倍して 1 を足す」ことを繰返す関数 f.

```
let is_even n =
   n mod 2 = 0
let rec f n =
  if n = 1 then 1
  else if is_even n then f (n / 2)
  else f (n * 3 + 1)
```

Collatz Problem (未解決問題): どんな正の整数 $x$ から始めても f(x) の計算は止まる (いつか 1 になる)。



## OCaml における再帰関数の定義

f をそのまま計算するのではなく、「100 回まわったところで答えを返す」方法は？

```
let rec g n k =
  if k <= 0 then n
  else if n = 1 then 1
  else if is_even n then g (n / 2) (k - 1)
  else g (n * 3 + 1) (k - 1)
let f n = g n 100
```

上記の関数 g は以下のように書いてもよいが、別の型になる。(型の詳細は後日説明)

```
let rec g' (n,k) =
  if k <= 0 then n
  else if n = 1 then 1
  else if is_even n then g' (n / 2, k - 1)
  else g' (n * 3 + 1, k - 1)
let f n = g' (n,100)
```

## 再帰関数を書く際のポイント

基本的な考え方: f(m) を計算するために、f(n) の計算結果を利用する (再帰) が、このとき「n が m より小さい」ようにすれば、f の計算がいつか止まるのでよい。

しかし、「m が素数であるかどうかを返す関数」の場合、m-1 や m/2 が素数であるかどうかわかってもあまり意味がない。

m を固定して、2 から m-1 までのすべての整数で m が割り切れるかどうかテストしたい。

- m とは別に、「2 から m-1 までを動く変数 k」をパラメータにもつ関数を補助関数として作ってから、素数判定関数を書くとよい。
- ここから先は、ウェブ上の「再帰関数」の説明を読んでください。



## 割り切れるかの判定

m が k から m-1 までの整数のうち 1 つ以上で割りきれるかどうかを判定する関数 h:

```
let rec h m k =
  if k >= m then false
  else if m mod k = 0 then true
  else h m (k + 1)
```

この関数は h m 2 で起動する。

m が 2 から k までの整数のうち 1 つ以上で割りきれるかどうかを判定する関数 h:

```
let rec h' m k =
  if k <= 1 then false
  else if m mod k = 0 then true
  else h' m (k - 1)
```

この関数は h' m (m-1) で起動する。



## 目次





## この章の目的

- 対象言語を拡張; 変数と変数束縛
- 自由/束縛変数, スコープ、出現、代入
- スタック機械へのコンパイル

## 変数と変数スコープ

スコープ (scope):

```
let x = 10 in
  (let x = 20 in
    x + 10)      (* この x は 20  *)
  * (x + 3);;    (* この x は  3 *)

let x = 10 in
  let f y = y + x in
   f x ;;          (* この x は 10 *)
```

キーワード

- 静的スコープと動的スコープ (ここでは静的スコープのみ扱う)
- ブロック構造言語
- スコープの入れ子 (nest)



## 変数と変数スコープ

C 言語の静的スコープ:

```
int z;
int foo (int x, int a[]) {
  int i;
  ... /* z x,a,i が使える */
  { int y;
    float x;
    ...
    ... /* z,a,i,y,x が使える. x は 内側の x */
  }
  ... /* z,x,a,i が使える */
}
```



## 変数の出現と束縛

項 $e$ における変数 $x$ の出現は、$x$ をスコープに持つ束縛 (binding) があるとき、「$e$ において束縛されている」と言う。そうでないとき「$e$ において自由」と言う。

変数 $x$ の出現が、複数の束縛のスコープにあるとき (入れ子のとき)、一番内側の束縛を取る。

どの $x$ が、自由/束縛 出現か考えなさい。

```
(let x = 10 in
 (let f x = x + 3 in
 f x) + x)
+ x ;;
```



## 束縛のある言語の構文

以前の expr の定義をやめて、新たに定義しなおす。

```
type expr =
| CstI of int
| Prim of string * expr * expr
| Var  of string   (* 変数 *)
| Let  of string * expr * expr  (* 変数束縛 *)
```

例: `Let("x",CstI(10),Prim("+",Var("x"),CstI(20)))`
これは、OCaml の `let x=10 in x+20` という式に対応

## 自由変数

式の中に自由に出現する変数のリストを求める。

```
let rec freevars e =
 match e with
  | Cst I → []
  | Var x → [x]
  | Let(x,erhs, ebody) →
      union(freevars erhs, minus (freevars ebody, [x]))
  | Prim(ope, e1, e2) →
      union(freevars e1, freevars e2)
```

ここで union は和集合、minus は差集合 (をリストで現したもの) を返す関数。

式 e に自由変数がないこと (`freevars e = [ ]`) が、e がまともな式 (コンパイルできる式) であることの必要条件である。



## 代入

```
subst e1 [("x", e2)]
```

$e1$ の中の変数 $x$ の自由な出現をすべて $e2$ にする変換。(ただし、$e2$ が別の変数の自由な出現をもっているときは、ややこしいので注意)
→ 次回の演習できちんとやる予定。



## 変数と変数束縛のある言語の意味

```
let rec eval e env =
 match e with
  | CstI i    → i
  | Var x     → lookup env x
  | Let(x, erhs, ebody)
      → let xval = eval erhs env in
          let env1 = (x, xval) :: env in
           eval ebody env1
  | Prim("+", e1, e2) → (eval e1 env) + (eval e2 env)
  | _ → failwith "unknown expression"
```

引数 env は 環境を表す。



## 実行時の環境 (environment)

変数の束縛を表す (変数とその値の対応): 例: `x=10, y=20, z=30`
ここでは、OCaml のリストを使って表現する。

空の環境 ... `[]`
`x=10, y=20` ... `[("x",10); ("y",20)]`

OCaml の組 (タプル) とリスト

- 組 `(a, b, c)`
- リスト: `[ a; b; c ]`

組とリストの違いは、型にある。

`(10,"abc",true)` と `[10;20]=[20;30;40]` は OK
`[10;"abc";true]` と `(10,20)=(20,30,40)` は駄目

## 環境の操作

初期値 (空の環境) `[ ]`
環境の延長 (extend) `a :: b`

```
 ("x",10) :: [("y",20); ("z",30)]
-> [("x",10); ("y",20); ("z",30)]
```

環境からの値の取り出し (lookup)

```
 lookup [("x",10); ("y",20); ("x",30)]  "x"
-> 10    (30 ではない)
```

後から入れたものが優先される.



## 評価の例

- let x=10 in x+20
- let x=10 in let x=30 in x+20
  - `x+20` を評価する時の環境は `[("x",30); ("x",10)]`
- let x=10 in (let x=30 in x+20) + x * 5
  - `x+20` を評価する時の環境は `[("x",30); ("x",10)]`
  - `x*5` を評価する時の環境は `[("x",10)]`



## 演習問題

今回の eval (変数と変数束縛のある言語に対する評価器) のもとで、以下の式を実行した過程 (1 ステップずつの実行) を手動でやってみなさい。

```
Let("x", CstI(10),
    Prim("+",Let("x",CstI(20),
                 Prim("+",Var("x"),Cst(20))),
        Var("x")))
```

(この式は `let x=10 in (let x=20 in x + 20) + x` を表す。)



## 目次

## これから現れる言語たち

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



## micro ML

Sestoft テキストでの micro ML:

- ML や F#の小さなサブセット
- 一階の言語 (not 高階言語)
- 関数型言語 (not 命令型言語)

一階 (first-order) vs 高階 (higher-order)

- 高階関数: 関数をもらったり、関数を返したりする関数
- 一階関数: 高階でない関数 (数をもらって、文字列を返す関数など)



## micro ML の構文

```
type expr =
  | CstI of int   | CstB of bool    | Var of string
  | Let of string * expr * expr
  | Prim of string * expr * expr
  | If of expr * expr * expr
  | Letfun of string * string * expr * expr
  | Call of expr * expr
```

ただし、`Call(e1,e2)` において `e1` は、`Var s` の形でなければいけない。
(「一階言語である」ための制限)

```
CstB true
If(Prim("=",Var "x",CstI 20),CstI 30,CstI 40)
   ... if x=20 then 30 else 40
Letfun("f", "x", Prim("+", Var "x", CstI 10),
       Call(Var "f", CstI 20))
   ... let rec f x = x + 10 in f 20
```



## micro ML

micro ML の特徴

- 「プログラム=(値を返す) 式」 である (「プログラム=命令」でない)
- (再帰) 関数の定義、関数呼出しがある
- 関数そのものは値とならない (変数に格納できない)
- 副作用はない (変数の値の書換え、入出力)

## micro ML プログラミング

OCaml:

```
z + 8   ;;
let rec f x = x + 7 in f 2 ;;
let rec f x = if x = 0 then 1
              else 2 + f(x−1)
in f 7 ;;
```

micro ML:

```
Prim("+", Var("z"), CstI(8))
Letfun("f","x",
  Prim("+",Var("x"),CstI(7)),
  Call(Var("f"),CstI(2)))
Letfun("f","x",
  If(Prim("=",Var("x"),CstI(0)),
     CstI(1),
     Plus("+",CstI(2),
          Call(Var("f"),
          Prim("−",Var("x"),CstI(1)))))
  Call(Var("f"),CstI(7)))
```



## 関数クロージャの必要性

例:

```
let x = 10 in
let f y = x + y in
let x = 20 in
  f 30
```

質問: f の中の x は 10 なのか 20 なのか？

- 静的束縛 (lexical, static binding)
- 動的束縛 (dynamic binding)

静的束縛を実現するには、`f` が `y -> x + y` であるという情報だけでは不足!



## 関数クロージャの必要性

関数クロージャ(関数閉包, function closure): 関数定義と環境をセットにしたもの。

```
let x = 10 in
let f y = x + y in
let x = 20 in
  f 30
```

上記の f に対応する関数クロージャ

```
(f y = x + y, [(x,10)])
```

評価器での表現

```
type value =
  | Int of int
  | Closure of string * string * expr * value env
```

上記の f に対応する関数クロージャ

```
Closure("f", "y", Prim("+",...), [(x,Int(10))])
```

関数適用後の「関数本体の計算」で、現在の環境ではなく、関数クロージャに格納していた環境を拡張したもので計算。



## 関数クロージャを使った計算

```
let x = 10 in
let f y = x + y in
let x = 20 in
  f 30
```

環境に格納するのは、x=10 の形、もしくは、関数クロージャ：

```
env0 = [("x",Int(10))]
env1 = [("x",Int(20));("f",Closure("f","y",Prim(..),env0)
   );("x",Int(10))]
```

env1 のもとでの (f 30) の実行：

1. `f` の値を `env1` から拾う。
2. 引数 30 を評価する。
3. `f` の仮引数 `y` を 30 に束縛する。
4. それを `env0` に追加する (それを `env2` とする)
5. `f` の本体を `env2` で評価する。

## 関数クロージャを使った計算

課題: 以下を手で計算せよ。

```
let x = 10 in let f y = x + y in
let x = 30 in let g z = x + z + (f z) in
let x = 40 in f (g 100)
```

```
env0 = (x=10)
env1 = (x=30; f=Closure(f,y,x+y,env0); env0)
env2 = (x=40; g=Closure(g,z,x+z+f(z),env1); env1)
```

計算過程:

- 100 の計算では env2 を使う。
- g の本体 x+z+f(z) の計算を環境 (z=100;env1) で行う。
- その中の x を計算して 30 を、z を計算して 100 を得る。
- f の本体で x+y の計算を環境 (y=100;env0) で行う。(結果は 110)
- x+z+f(z) の計算結果は 240 である。
- f(240) の計算を env2 で行なう。
- f の本体 x+y の計算を環境 (y=240;env0) で行う。(結果は 250)



## 関数クロージャの処理

関数クロージャへの対応 1. (関数定義への対応)

```
let rec eval e env =
 match e with
  | CstI i    → i
  ...
  | Letfun(f, x, fBody, letBody) →
    let bodyEnv = (f, Closure(f, x, fBody, env)) :: env
    in eval letBody bodyEnv
  ...
```

関数クロージャを作って、それを環境に格納するだけ。



## 関数クロージャの処理

関数クロージャへの対応 2. (関数適用への対応)

```
...
| Call(Var f, eArg) →
  let fClosure = lookup env f in
  match fClosure with
  | Closure (f,x,fBody,fDeclEnv) →
    let xVal = Int(eval eArg env) in
    let fBodyEnv = (x,xVal)::(f,fClosure)::fDeclEnv in
       eval fBody fBodyEnv
```

- f に対応する値を、環境 env から取得
- それを Closure(...) とパターンマッチ
- f の実引数 eArg を「現在の」環境 env で評価 (計算)
- f の仮引数 x を実引数の評価結果に束縛 (環境に格納)
- (microML の関数は再帰関数なので) f 自身の値を再度環境に格納
- f の本体 fBody を新しい環境のもとで評価



## 疑問

いろいろな疑問：

- これで本当に静的束縛になるのか？
- 動的束縛にするのにはどうしたらよいか？
- 関数クロージャは，動的に (実行時に) 作られるが，メモリを消費するのではないか？

## 動的束縛

```
let x = 10 in
let f y = x + y in
let x = 20 in
  f 30
```

[以下の部分は、2017/5/11 に修正しました。]
動的束縛では、評価結果が 50 になる。



## 動的束縛への対応

eval の定義を 1 か所変更する：関数クロージャにおける「環境」を無視する。
静的束縛に対応する eval の定義の一部 (再掲)

```
| Call(Var f, eArg) →
  let fClosure = lookup env f in
  match fClosure with
  | Closure (f,x,fBody,fDeclEnv) →
    let xVal = Int(eval eArg env) in
    let fBodyEnv = (x,xVal)::(f,fClosure)::fDeclEnv in
       eval fBody fBodyEnv
```

動的束縛: 上記の下から 2 番目の行を以下に置きかえればよい。

```
    let fBodyEnv = (x,xVal)::env in
```

あるいは、最初から、関数クロージャを作らずに、関数の本体の定義だけを覚えるという実装でもよい。



## まとめ

- 拡張: 関数定義と関数適用を含むプログラム言語
- 静的束縛: 関数クロージャが必要
- 動的束縛: 関数クロージャは不要 (前回のインタプリタと同様の仕組みで単純に実装できる)

来週の演習にそなえて，スライドの復習 (と、できれば、Sestoft 教科書の第 4 章を参照) をしておいてください。テキストのプログラムは、coins マシンの~kam/plm/ の下に**既に**置いてあります。