

# 関数・論理型プログラミング実験 ML演習第5回

松田 一孝  
TA: 武田広太郎 寺尾拓

# 今日の話

- 5/01：簡単な評価器
  - ◆ 字句解析・構文解析. 簡単な評価器
- 5/13：関数型言語の評価器
  - ◆ (高階) 関数定義・呼出機構の作成
- 5/20：型システム
  - ◆ ML風の型推論の実装
- 5/27：その他拡張
  - ◆ 評価規則等

# 話の流れ

- 関数定義・呼出機構の実現
  - ◆ 非再帰関数
  - ◆ 再帰関数
- パターンマッチ

# 復習：前回のレポート課題

## ○ 環境

- ◆ 変数の実現に利用
- ◆ 変数から値へのマッピング

`eval : env -> expr -> value`

# 目標その1

- 前回の言語に  
さらに関数抽象・適用を加えよう
  - ◆ 関数抽象 (fun式)
  - ◆ 関数適用

# fun式の「値」？

- fun  $x \rightarrow e$  を評価すると何に？
  - ◆ 値の「つかわれ方」を考える

(fun  $x \rightarrow e$ )  $e'$  の評価

$e'$  の評価結果を  $v'$  とすると  
 $x$  を  $v'$  として  $e$  を評価

# 試案

## ○ fun式そのものが値

```
type expr  = ... | EFun of name * expr
              | EApp of expr * expr
and value = ... | VFun of name * expr
...
let rec eval env expr = match expr with
| EFun (x, e) -> VFun (x, e)
| EApp (e1, e2) ->
    let VFun (x, e) = eval env e1 in
    let v2 = eval env e2 in
    eval ((x, v2)::env) e
```

例

$(\text{fun } x \rightarrow x + 5) \ 3$

環境  $\{x=3\}$  で  
 $x+5$  を評価

8

うまく行きそう？



# うまくいかない例

`(let y=5 in fun x -> x + y)` 3

環境  $\{y=5\}$  で  
枠内の式を評価

`(fun x -> x+y)`

環境  $\{x=3\}$  で  
 $x+y$  を評価

エラー:  $y$  は未定義

# 観察

$\text{fun } x \rightarrow e$

- 式 $e$ はこの式の外側で定義される変数（自由変数）を含む
- $\text{fun}$ 式そのものを評価結果とすると自由変数の値の情報が失われる
  - ◆ 例:  $\text{let } y = 5 \text{ in fun } x \rightarrow x+y$

# 解決案：クロージャ

- fun式の評価結果をクロージャにする
  - ◆ クロージャ：関数と環境の組
    - \* スライドでは〈関数, 環境〉と書く

```
and value = ... | VFun of name * expr * env
...
let rec eval env expr = match expr with
...
| EFun (x, e) -> VFun (x, e, env)
...
```

# クロージャの適用

- $\langle \text{fun } x \rightarrow e, \text{oenv} \rangle$   $v$  の評価
  - ◆ 環境  $\text{oenv}$  に  $x$  と  $v$  の対応を追加し  $e$  を評価

```
let rec eval env expr = match expr with
...
| EApp (e1, e2) ->
    let VFun (x, e) oenv = eval env e1 in
    let v2 = eval env e2 in
    eval ((x, v2) :: oenv) e
```

# 例

$(\text{let } y=5 \text{ in fun } x \rightarrow x + y)$  3

環境  $\{y=5\}$  で  
枠内の式を評価

$\langle \text{fun } x \rightarrow x+y, \{y=5\} \rangle$

環境  $\{x=3, y=5\}$  で  
 $x+y$  を評価

8

# まとめ

- 関数抽象・関数適用の評価
  - ◆ クロージャ
    - \* 関数抽象がどの環境で行われたかが大事

# 余談：動的スコープ

- 「うまく行かなかった」とした方法は動的スコープに対応
  - ◆ 以下の式の評価結果は？

```
let y = 1 in  
(let y=5 in fun x -> x + y) 3
```

\* 静的スコープ : 8  
動的スコープ : 4

# 余談：letと関数抽象・適用

- $\text{let } x = e_1 \text{ in } e_2$  と  $(\text{fun } x \rightarrow e_2) e_1$  はとても似ている
  - ◆ 今回の範囲では同一視可
  - ◆ しかし，OCamlでは（Haskellでも）両者は異なる
    - \* どのようなとき異なるか？
- 答えは来週



# 目標その2

- さらに再帰関数を追加せよ

# 問題

```
let rec loop n = loop n in  
  loop 0
```

loopの「値」は？

$\langle \text{fun } n \rightarrow \text{loop } n, \{\} \rangle$ では×

# 復習 : fix

o  $\text{fix}: ((\text{'a} \rightarrow \text{'b}) \rightarrow (\text{'a} \rightarrow \text{'b})) \rightarrow (\text{'a} \rightarrow \text{'b})$

$$\begin{aligned} &\text{fix } (\text{fun } f \ x \rightarrow e) \\ &\equiv \\ &\text{let rec } f \ x = e \text{ in } f \end{aligned}$$

# 解決案

- `fix (fun f x -> e)` に  
対応する「値」を作る
  - ◆ スライドでは `<fix (fun f x -> e), env>`

```
type expr = ...  
          | ERLet of name * name * expr * expr  
and value = ...  
          | VRFun of name * name * expr * env  
...  
let rec eval env expr = match expr with  
  | ERLet (f, x, e1, e2) ->  
    let env' = (f, VRFun (f, x, e1, env)) :: env in  
    eval env' e2
```

# 適用のアイデア

○  $\text{fix } h \ x = h \ (\text{fix } h) \ x$   
に基づく

$$\begin{aligned} & \text{fix } (\text{fun } f \ x \rightarrow e) \ v \\ &= (\text{fun } f \ x \rightarrow e) \\ & \quad (\text{fix } (\text{fun } f \ x \rightarrow e)) \ v \end{aligned}$$

# 再帰関数の適用

- $\langle \text{fix } (\text{fun } f \ x \rightarrow e), \text{ env} \rangle \ v$ 
  - ◆ 環境  $\text{env}$  に,  
     $f$  と  $\langle \text{fix } (\text{fun } f \ x \rightarrow e), \text{ env} \rangle$ ,  $x$  と  $v$   
    の対応を追加し,  $e$  を評価

```
let rec eval env expr = match expr with
| EApp (e1, e2) ->
  let v1 = eval env e1 in let v2 = eval env e2 in
  (match v1 with ->
  | VFun (x, e, oenv) -> ...
  | VRFun (f, x, e, oenv) ->
    let env' =
      (x, v) :: (f, VRFun (f, x, e, oenv)) :: oenv in
    eval env' e)
```

# 議論 : `let rec f = e`

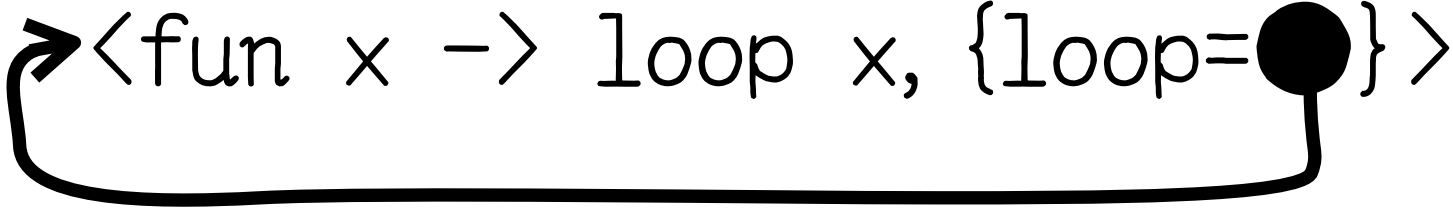
Q. `let rec f = e`の形をどうして禁じる？

A. 簡便のため

- ◆ `let rec f = e`だと、  
eがこの時点で評価されることを期待
  - \* `let rec f = print_string "hey";  
          fun x -> ...`
- ◆ しかし、eはfを含むかかもしれず  
一般には評価できない
- ◆ 評価してよいeを適切に定めるのは大変
  - \* 事実、ocamlの許容するeの形は複雑

# 他の解決法

- 循環的なクロージャを用いる

`<fun x -> loop x, {loop=●}>`

- ◆ 循環構造は副作用を用いて作成可



# さらに別の解決方法

- 再帰関数がトップレベルのみに出現するのなら楽
  - ◆ 再帰関数名  $\rightarrow$  式のマッピングを予め与えられる
- lambda-liftingを利用すれば式を上記の形に変換可

```
let y = 1 in  
let rec f x = f x + y in f 2
```



```
let rec f x y = f x + y in  
let y = 1 in f 2 y
```

# 目標その3

- パターンマッチング  
(による場合分け処理) を実装しよう
  - ◆ 要はmatch式の実装

# match式の評価

- $\text{match } e \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_n \rightarrow e_n$   
は以下のように評価
  - ◆ 式 $e$ を評価し値 $v$ を得る
  - ◆ パターン $p_1$ と $v$ を照合：
    - \* 照合すれば環境に結果を追加し $e_1$ を評価
    - \* しなければ次ステップへ
  - ◆ ...
  - ◆ パターン $p_n$ と $v$ を照合：
    - \* 照合すれば環境に結果を追加し $e_n$ を評価
    - \* しなければエラー

# パターン照合

- 入力：パターンと値  
出力：成否と束縛
  - ◆ 束縛：変数から値へのマッピング
- 例
  - ◆ 1と0→失敗
  - ◆ 1と1→成功
  - ◆  $x$ と1→成功  $\{x=1\}$
  - ◆  $(x, y)$ と $(1, (2, 3))$ →成功  $\{x=1, y=(2, 3)\}$
  - ◆  $(x, 1)$ と $(2, 3)$ →失敗

第5回レポート課題  
締切: 2週間後の13:00 (JST)

# 問1

- 前回の課題のインタプリタを拡張し関数抽象・適用を扱えるようにせよ
  - ◆ 構文は最低以下のものをサポート
    - \* fun 変数  $\rightarrow$  式
- 関数適用の優先度は最強, 左結合とすること
  - ◆  $f\ x\ *\ 1$  は  $(f\ x)\ *\ 1$
  - ◆  $f\ x\ y$  は  $(f\ x)\ y$

# 問2

- インタプリタを拡張し再帰関数を扱えるようにせよ
  - ◆ 構文は最低以下をサポート
    - \* `let rec 変数 変数 = 式`
      - 変数が二つ（以上）なのは簡便のため
- 実装はどんな方法を用いてもよいが、きちんと自分の言葉で説明すること

# 問3

- インタプリタを拡張しパターンマッチを行えるようにせよ
  - ◆ 構文は最低以下はサポート
    - \*  $\text{match 式 with } \begin{array}{l} \text{パターン}_1 \rightarrow \text{式}_1 \\ \vdots \\ \text{パターン}_n \rightarrow \text{式}_n \end{array}$
- パターンは整数, 真理値, 変数でよい



# ヒント

- 以下の型を持つfind\_match関数を作る
  - ◆ find\_match :  
    pattern \* value -> binding option
    - \* patternはパターンの型
    - \* bindingは束縛の型
      - 環境の型と同じでよい
- あとはfind\_matchを利用しパターンマッチング式の評価規則を実装する

# 注意

- ネストしたときの挙動に気をつけよ
  - ◆ `match x with`  
    `p -> match y with`  
        `q -> e`  
        `| r -> ...`
  - ◆ 解決法は  
    「dangling else」等のキーワードで  
    調べる

# 問4

- インタプリタを拡張し，組とリストを扱えるようにせよ
  - ◆ 構文は最低以下をサポート
    - \* 式 :: 式
      - 右結合
    - \* []
- パターンマッチも行えるようにせよ

# 問5

- インタプリタを拡張し,  
相互再帰関数を扱えるようにせよ
- ◆ 構文は最低以下をサポート
  - \*  $\text{let rec } f_1 \ x_1 = e_1$   
     $\text{and } f_2 \ x_2 = e_2$   
     $\dots$   
     $\text{and } f_n \ x_n = e_n \text{ in } e$

# 補足1

- fixのアイデアを利用する方法だと：
  - ◆ 関数 $f_i$ を  
 $\langle i; (f_1, x_1, e_1), \dots, (f_n, x_n, e_n); env \rangle$   
という形の値で表現すればよい
  - ◆ 上の値を $v$ に適用
    - \* 環境 $env$ に  
 $f_1$ と $\langle 1; (f_1, x_1, e_1), \dots, (f_n, x_n, e_n); env \rangle$   
...  
 $f_n$ と $\langle n; (f_1, x_1, e_1), \dots, (f_n, x_n, e_n); env \rangle$   
と  
 $x_i$ と $v$ の対応を追加した環境で  
 $e_i$ を評価

# 補足2

- 循環的なクロージャを用いる方法だと
  - 以下のような循環的な環境を作ればよい

env:

$\begin{aligned} &\{f1 = \langle \text{fun } x1 \rightarrow e1, \text{ env} \rangle, \\ &\quad f2 = \langle \text{fun } x2 \rightarrow e2, \text{ env} \rangle, \\ &\quad \dots \\ &\quad fn = \langle \text{fun } xn \rightarrow en, \text{ env} \rangle\} \end{aligned}$
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

# 補足3

- lambda-liftingを用いると  
プログラムを以下の形で  
 $e$ が`let rec`や`fun`式を含まないもの  
に書き換えられる

```
let rec f1 x1 = e1  
and      f2 x2 = e2  
...  
and      fn xn = en in e
```

# 課題全体の補足

- 問1～5の実装はまとめて一つでよい
  - ◆ レポートはだめ
- 構文解析器の作成に時間を掛けるのは問題の意図から外れるので、サポートサイトに置いてあるものを利用可
  - ◆ ただし、参考にした旨は書くこと



# 発展1

- CEK抽象機械について調べ、  
今回の演習で扱う言語の  
CEKに基づくインタプリタを作成せよ

# 発展2

- インタプリタにcall/ccを追加せよ
  - ◆ 発展1のインタプリタに追加すると楽
  - ◆ 評価器を変更せずとも、  
式を別の形の式に変更することで実装が可能となる？