ML 演習 第 5 回

新井淳也、中村宇佑、前田俊行

2011/05/17

言語処理系の作成

- 第 4 回～第 7 回の予定
  - 第 4 回: 基本的なインタプリタの作成
    - 字句解析・構文解析・簡単な言語の処理系の作成
  - 第 5 回: 関数型言語への拡張
    - 関数を作成し呼び出せるように
  - 第 6 回: 言語処理系と型システム
    - ML 風の型推論の実装
  - 第 7 回: インタプリタの様々な拡張
    - 式の評価順序に関する考察

今回の内容

- 関数 (クロージャ) を扱えるようにする
  - 普通の関数
  - 再帰関数
- パターンマッチを扱えるようにする

Functions (Closures)

関数 (クロージャ)

クロージャとは

- 関数を表す「値」としてよく用いられる表現方法
  - 関数を作成する「式」 (λ 抽象と呼んだりする)
    の評価結果の「値」として用いる

- 大抵は以下の三つの要素の組で実装する
  - 関数の仮引数を表す「変数」
  - 関数の本体を表す「式」
  - 関数の自由変数の値を表す「環境」
    - 変数から値への写像 (第 4 回の課題 3 参照)

なぜクロージャに環境が必要か

- 仮にクロージャに環境を含めなかったとして以下の式を評価することを考える

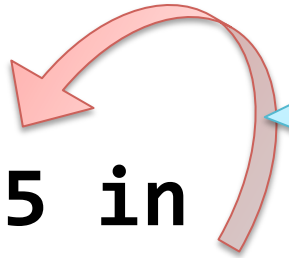
```
let f =
  let x = 5 in
  fun y -> x + y
in
f 37
```

次ページに続く

なぜクロージャに環境が必要か

- 仮にクロージャに環境を含めなかったとして以下の式を評価することを考える

```
let f =
  let x = 5 in
  fun y -> x + y
in
f 37
```



この自由変数 x は x = 5 であることが期待される

次ページに続く

なぜクロージャに環境が必要か

- 仮にクロージャに環境を含めなかったとして以下の式を評価することを考える

```
let f =
  let x = 5 in
  fun y -> x + y
in
f 37
```

この fun 式の評価結果はクロージャになる

次ページに続く

なぜクロージャに環境が必要か

- 仮にクロージャに環境を含めなかったとして以下の式を評価することを考える

```
let f =
    let x = 5 in
    fun y -> x + y
in
f 37
```

fun 式の評価結果を変数 f が束縛する

この fun 式の評価結果はクロージャになる

次ページに続く

なぜクロージャに環境が必要か

- 仮にクロージャに環境を含めなかったとして以下の式を評価することを考える

```
let f =
    let x = 5 in
    fun y -> x + y
in
f 37
```

fun 式の評価結果を変数 f が束縛する

この fun 式の評価結果はクロージャになる

環境

```
f = (y, x + y)
```

関数の引数

関数の本体

つまりこの関数適用の評価時の環境は右のようになる

次ページに続く

なぜクロージャに環境が必要か

- 仮にクロージャに環境を含めなかったとして以下の式を評価することを考える

```
let f =
    let x = 5 in
    fun y -> x + y
in
f 37
```

次に
この関数適用式の
評価を考える

環境

f = (y, x + y)

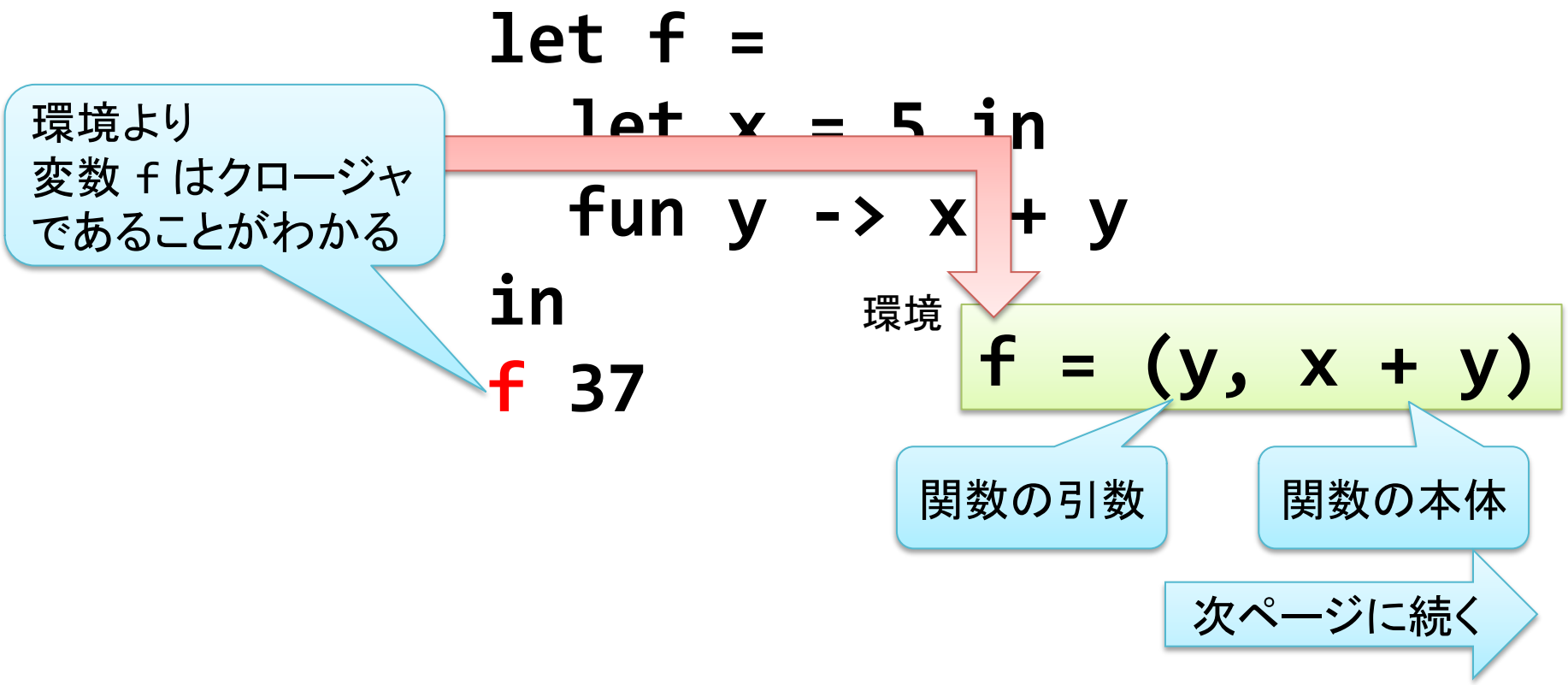
関数の引数

関数の本体

次ページに続く

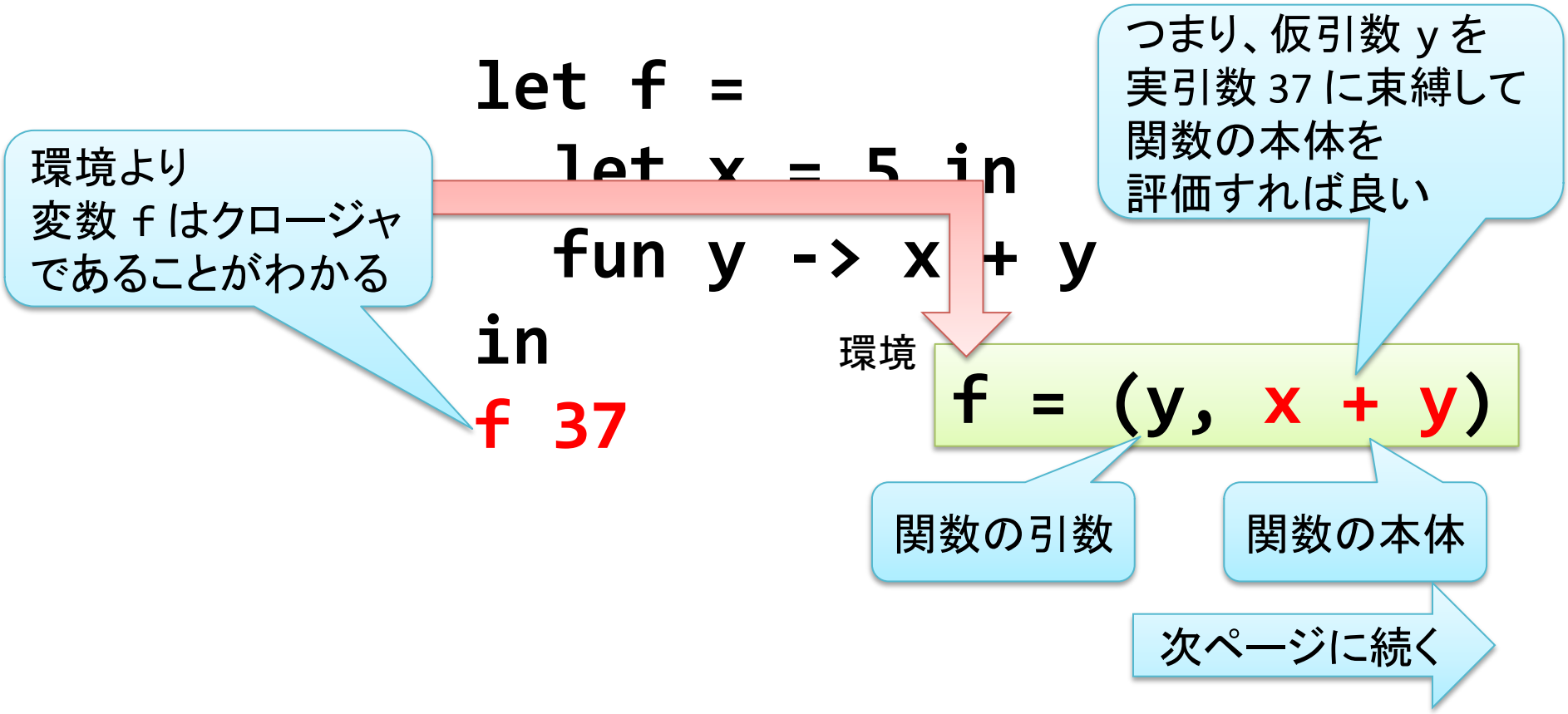
なぜクロージャに環境が必要か

- 仮にクロージャに環境を含めなかったとして以下の式を評価することを考える



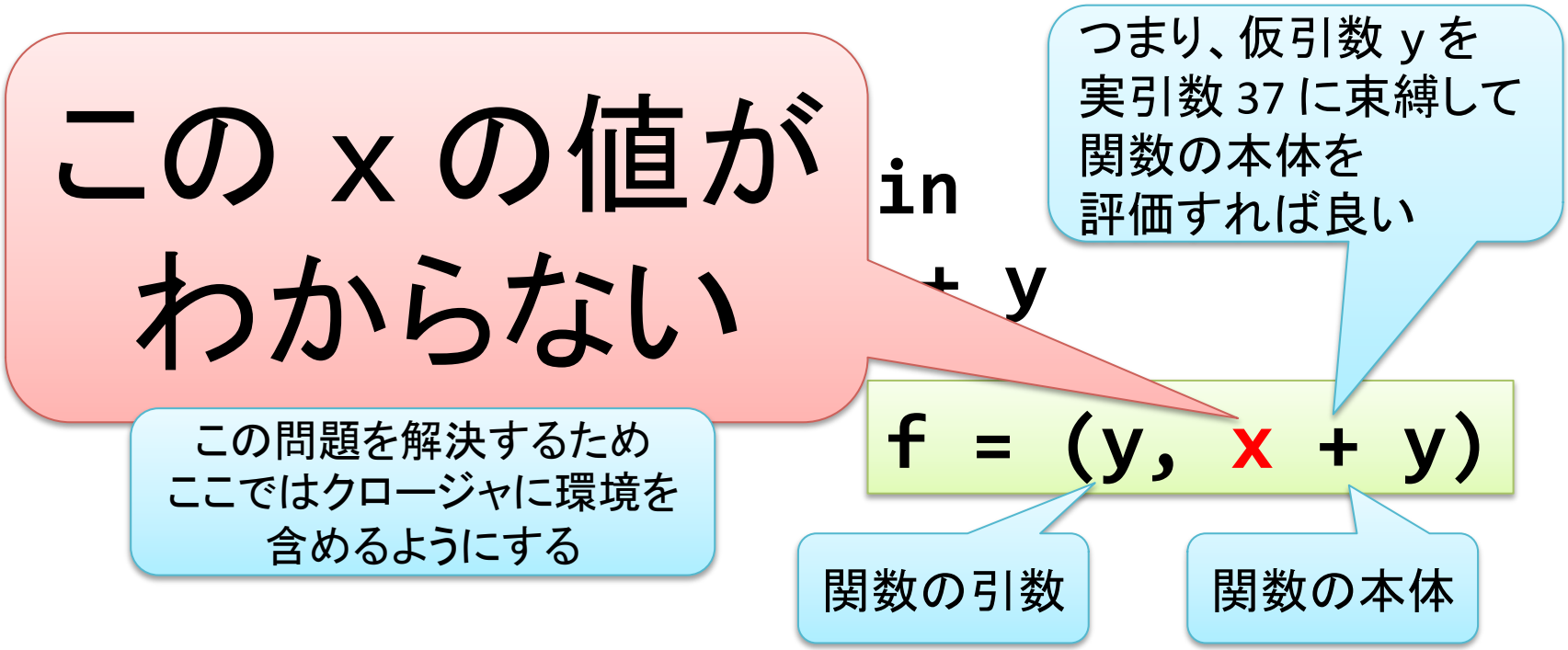
なぜクロージャに環境が必要か

- 仮にクロージャに環境を含めなかったとして以下の式を評価することを考える



なぜクロージャに環境が必要か

- 仮にクロージャに環境を含めなかったとして以下の式を評価することを考える



λ抽象式の評価

- 式「fun *x* -> *e*」は次のように評価される
  1. クロージャを作成する (これだけ)
     - 関数の仮引数 *x* と
     - 関数の本体 *e* と
     - この式 (fun *x* -> *e*) を評価するときの環境

       の 3 つを用いて作成する

λ抽象式の評価の例 (1/2)

- 以下の式を評価することを考える
  ```
  let x = 5 in fun y -> x + y
  ```

1. 部分式 5 を評価する
2. この式を評価する時の環境を拡張する
   - 変数 x を 1. の評価結果 (整数値 5) に束縛する (第 4 回の課題 3 参照)

拡張された環境

```
x = 5
```

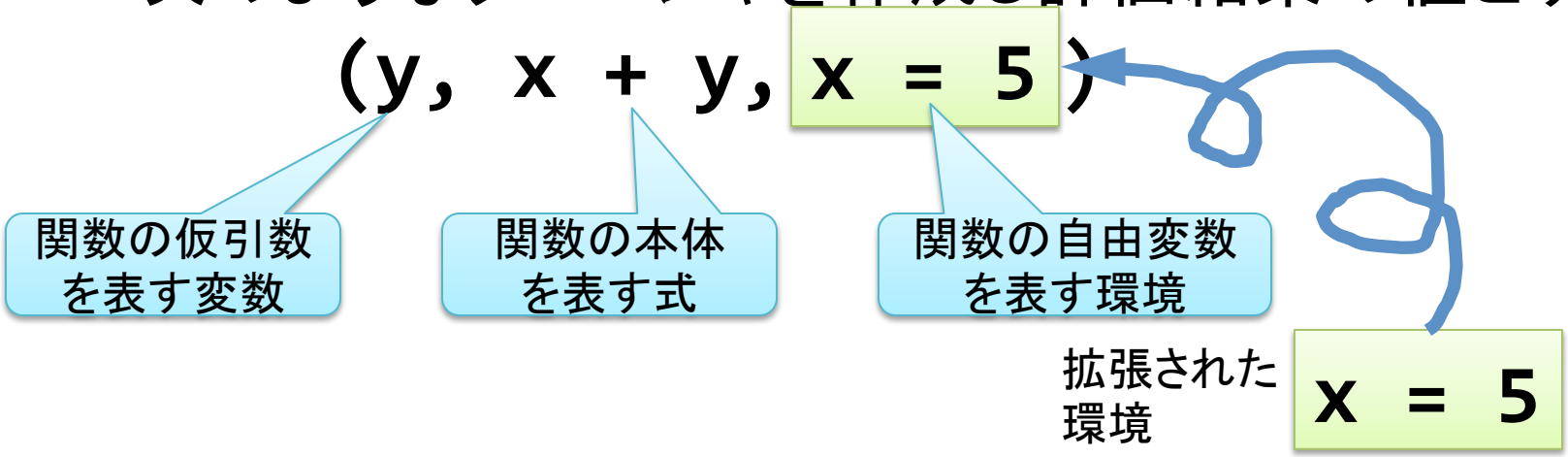
(ここでは元々環境が空だったとする)

次ページに続く

λ抽象式の評価の例 (2/2)

- 以下の式を評価することを考える
  `let x = 5 in fun y -> x + y`

3. 拡張された環境でλ抽象式を評価する
   - 次のようなクロージャを作成し評価結果の値とする



関数適用式の評価

- 関数適用式「$e_1\ e_2$」は次のように評価される
  1. e_1 と e_2 を現在の環境でそれぞれ評価する
     - e_1 の評価結果がクロージャでなかったらエラー
  2. クロージャに保存された環境を取り出し拡張する
     - 引数を表す変数を値 (e_2 の評価結果) に束縛する
  3. 2. で拡張された環境で
     クロージャに保存された関数本体を評価する
     - 即ち、関数本体を評価するときの環境は、
       「クロージャに保存されていた環境」+「引数の束縛」

関数適用式の評価の例 (1/2)

- 以下の式を評価することを考える

  f (3 + 4)

  – ただし変数 f は
  以下のクロージャに束縛されているとする

  (y, x + y, x = 5)

1. まず変数 f を評価する
   – クロージャが得られる
2. 式 (3 + 4) を評価する
   – 整数値 7 が得られる

次ページに続く

関数適用式の評価の例 (2/2)

- 以下の式を評価することを考える

  f (3 + 4)

  – ただし変数 f は
  以下のクロージャに束縛されているとする

  (y, x + y, x = 5)

3. 1. で得られたクロージャに保存された環境を取り出し拡張する
   – 変数 y を 2. で得られた値 7 に束縛する
4. 拡張された環境でクロージャ内の関数本体 x + y を評価する

拡張された環境

y = 7
x = 5

再帰関数を扱う上での問題 (1/3)

- これまで説明した方法だけでは上手くいかない
- 例えば以下の式を評価することを考えてみる

```
let rec loop x = loop x in loop 0
```

関数の中から関数自身を参照している

次ページに続く

再帰関数を扱う上での問題 (2/3)

- これまで説明した方法だけでは上手くいかない
- 例えば以下の式を評価することを考えてみる

```
let rec loop x = loop x in loop 0
```

- まず環境が拡張される
  - 変数 loop がクロージャに束縛される

拡張された
環境

```
loop = (x, loop x, )
```

クロージャの
中の環境は空

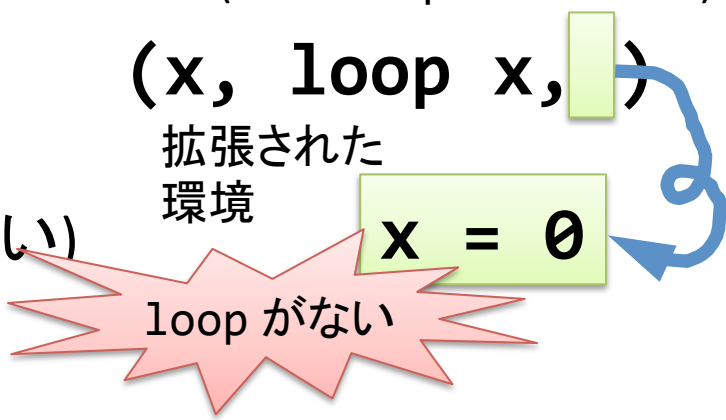
次ページに続く

再帰関数を扱う上での問題 (3/3)

- これまで説明した方法だけでは上手くいかない
- 例えば以下の式を評価することを考えてみる

```
let rec loop x = loop x in loop 0
```

- 拡張された環境で関数適用式を評価する
  - まずクロージャから環境を取り出して拡張する
    - 変数 x を値 0 に束縛する
  - 拡張された環境で関数本体を評価する
    - エラー! (変数 loop が環境にない)



再帰関数を実現するには

- クロージャ中の環境に loop があれば良い
- ここでは 2 種類の解決法を紹介する
  1. クロージャを工夫する
     - 自分自身の変数 (loop) を覚えておき、
       関数適用式を評価するときに
       「自分自身の変数」を「自分自身のクロージャ」
       に束縛するように環境を拡張する
  2. 環境を工夫する
     - クロージャを作成するときに
       参照や再帰などを使って
       自分自身を指し示すように環境を拡張する

再帰関数の実現方法その 1: クロージャを工夫する

- 前準備
  - クロージャの実装を変更して
    自分自身に束縛される変数も組に加えておく
- λ 抽象式の評価
  - 元の処理に加えて自身を表す変数を
    クロージャに保存する
- 関数適用式の評価
  - クロージャに保存された環境の拡張時に
    引数に加えて自身の変数も追加する

その 1 の方法で 再帰関数を評価する例 (1/3)

- 以下の式を評価することを考えてみる

```
let rec loop x = loop x in loop 0
```

- まず環境が拡張される
  - 変数 loop がクロージャに束縛される
    - 自身を表す変数がクロージャに保存される

拡張された
環境

```
loop = (loop, x, loop x, )
```

自身を表す変数も
保存しておく

次ページに続く

その 1 の方法で 再帰関数を評価する例 (2/3)

- 以下の式を評価することを考えてみる

```
let rec loop x = loop x in loop 0
```

- 拡張された環境で関数適用式を評価する
  - まずクロージャから環境を取り出して拡張する
    - 変数 x を値 0 に束縛する

ここまでは普通のクロージャと同じ

クロージャ

```
(loop, x, loop x, )
```

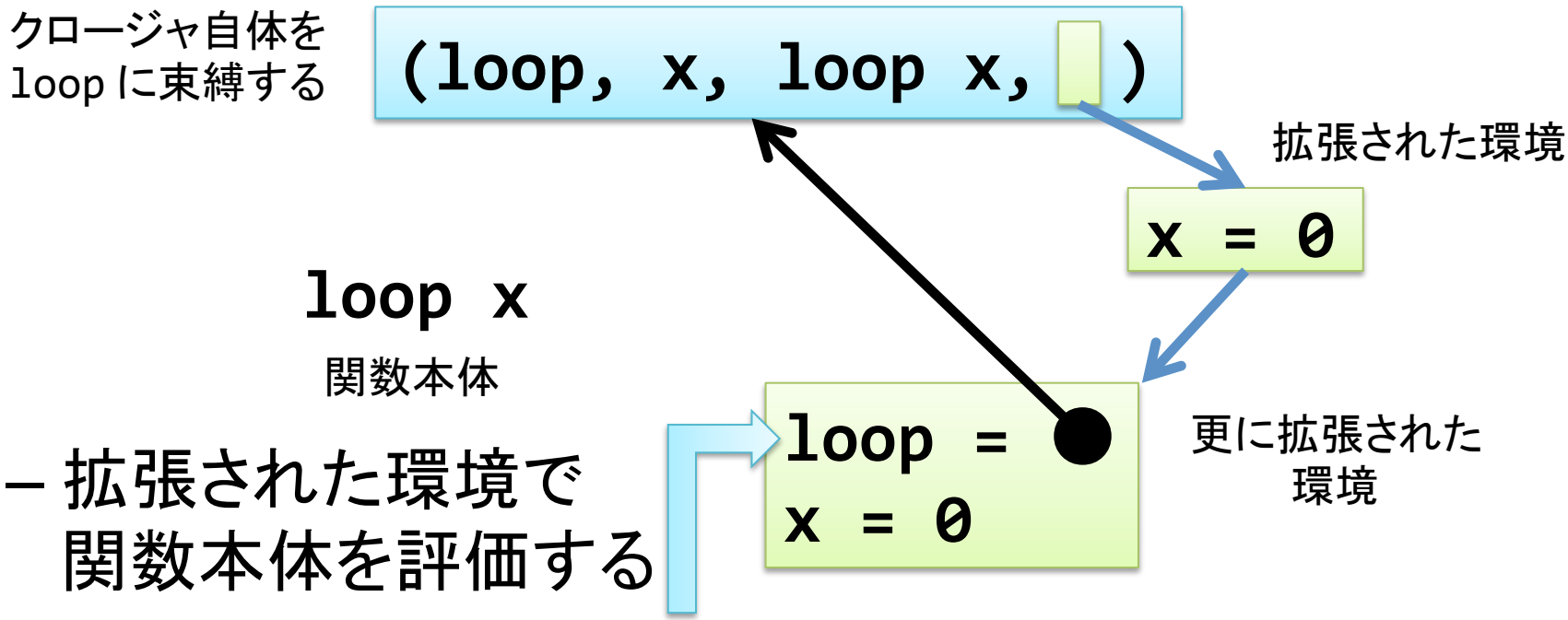
拡張された環境

```
x = 0
```

次ページに続く

その 1 の方法で 再帰関数を評価する例 (3/3)

- 更に変数 loop をクロージャに束縛する



  - 拡張された環境で 関数本体を評価する
    - 環境の中に loop が存在する

再帰関数の実現方法その 2: 環境を工夫する

- 前準備
  - 環境の実装を変更して
    クロージャ中の環境の値が
    そのクロージャ自身を参照できる構造にする
- λ 抽象式の評価
  - 関数を表す変数を
    再帰的にそのクロージャ自身に束縛して
    環境を拡張する
- 関数適用式の評価
  - そのまま

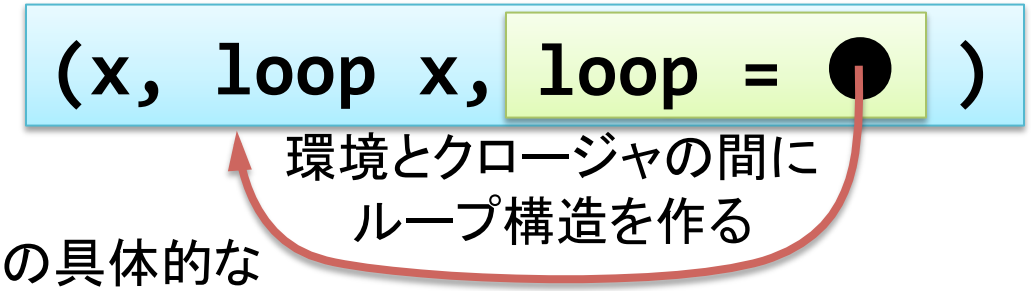
その 2 の方法で 再帰関数を評価する例 (1/3)

- 以下の式を評価する

```
let rec loop x = loop x in loop 0
```

- まず以下のような再帰的なクロージャを作る
  - クロージャ中の環境で、変数 loop がそのクロージャ自身に束縛されている

(ここでは元々環境が空だったとする)



ループ構造の具体的な
作り方は後述

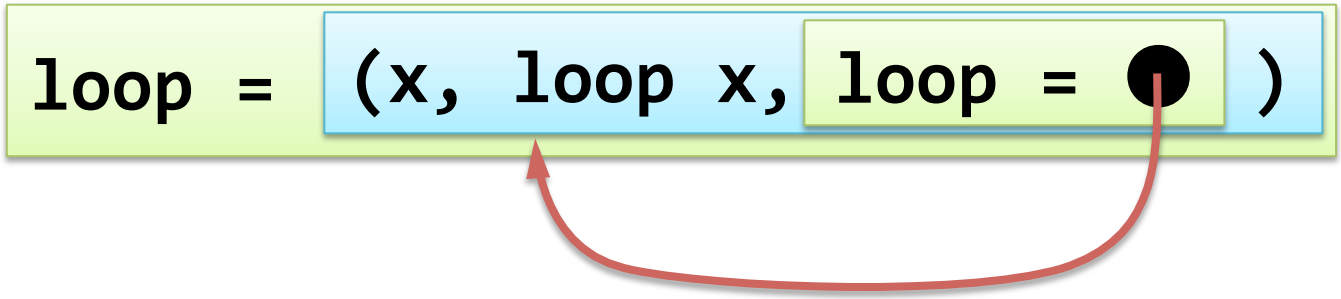
次ページに続く

その 2 の方法で
再帰関数を評価する例 (2/3)

- 以下の式を評価する

```
let rec loop x = loop x in loop 0
```

- 変数 loop を作成したクロージャに束縛する



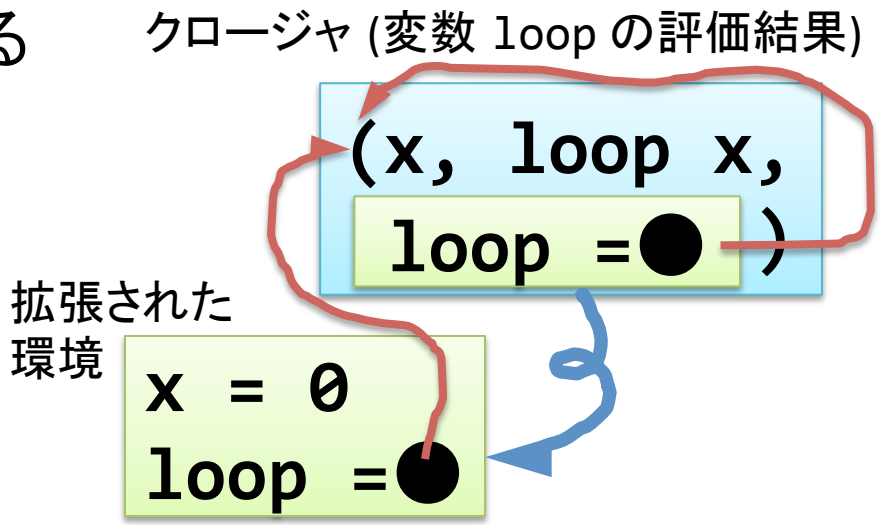
次ページに続く

その 2 の方法で 再帰関数を評価する例 (3/3)

- 以下の式を評価する

```
let rec loop x = loop x in loop 0
```

- 関数適用式を評価する
  - まずクロージャから環境を取り出して拡張する
    - 変数 x を値 0 に束縛する
  - 拡張された環境で 関数本体を評価する
    - 環境の中に loop が 存在する



環境とクロージャの間にループ構造を作る方法の１つ: 参照を利用する

- 前準備
  - 環境の実装を変更して、環境中の値を参照型にし、後から上書きできるようにする
- クロージャの作成
  1. まず環境を拡張する
     - 関数自身を表す変数をとりあえずダミーの値に束縛
  2. 拡張された環境を含むクロージャを作成する
  3. 1. でとりあえず作ったダミーの値を 2. で作ったクロージャで上書きする

環境とクロージャの間にループ構造を作る方法の 1 つ: 参照を利用する例 (1/3)

- 以下の式を評価する

```
let rec loop x = loop x in loop 0
```

- まず環境が拡張される
  - 変数 loop がダミーの値に束縛される

拡張された環境

```
loop = Dummy
```

(ここでは元々環境が空だったとする)

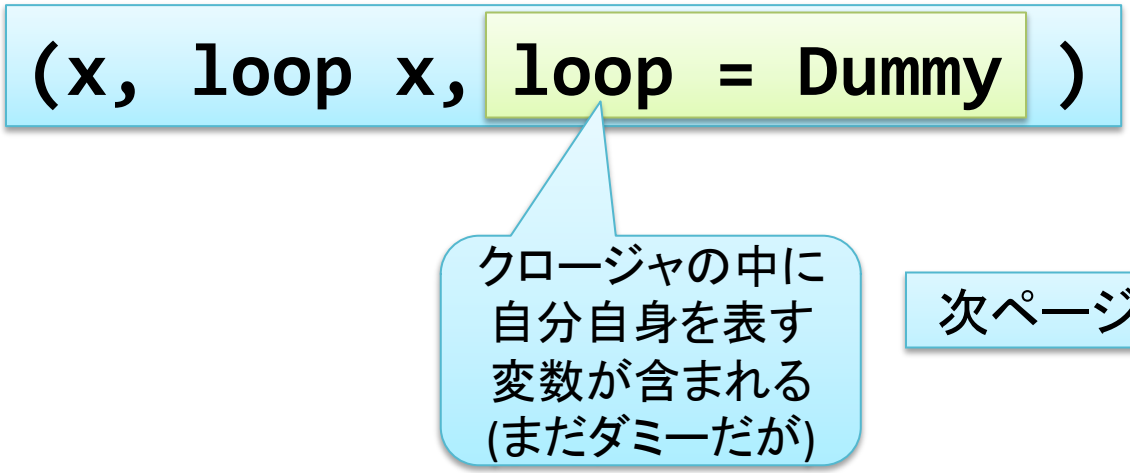
次ページに続く

環境とクロージャの間にループ構造を作る方法の 1 つ: 参照を利用する例 (2/3)

- 以下の式を評価する

```
let rec loop x = loop x in loop 0
```

- 次に λ 抽象式を評価する
  - 以下のクロージャが作成される

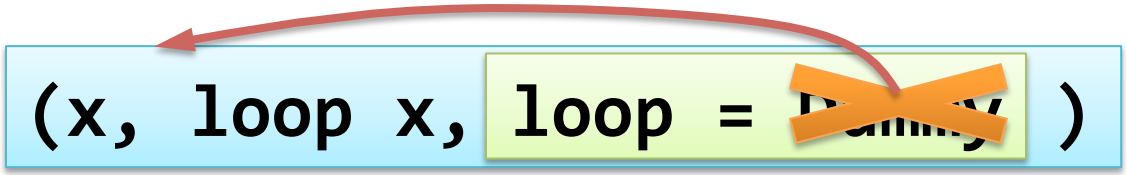


環境とクロージャの間にループ構造を作る方法の 1 つ: 参照を利用する例 (3/3)

- 以下の式を評価する

```
let rec loop x = loop x in loop 0
```

- ダミーの値をクロージャで上書きする
  - 環境とクロージャの間にループができる



ここで紹介した
再帰関数を実現する方法の比較

	クロージャの実装の修正	関数適用の実装の修正	環境・値の実装の修正	参照の利用
方法 1	必要あり	必要あり	必要なし	必要なし
方法 2	必要なし	必要なし	必要あり	必要あり

- 他にも色々実装の方法あり

Pattern Matching

パターンマッチング

パターンマッチング式の評価

- 式「match e with p_1 -> e_1 | ... | p_{n-1} -> e_{n-1}」は次のように評価される
  1. 式 e を評価する
  2. パターン p_1 と 1. の評価結果を比べる
     - マッチしたら式 e_1 を評価して全体の評価結果とする
       - マッチしなければ次のステップへ

  ...

  n. パターン p_{n-1} と 1. の評価結果を比べる
     - マッチしたら式 e_{n-1} を評価して全体の評価結果とする
       - マッチしなければエラー

変数束縛パターンの処理

  - 基本的に let 式の評価と同じ
- 式「match *e* with *x* -> *e′*」は次のように評価される
  1. 式 *e* を評価する
  2. 環境を拡張する
     - 変数 *x* を 1. の評価結果の値に束縛する
  3. 拡張した環境で式 *e′* を評価する

パターンマッチング式を評価する例

- 以下の式を評価する
  ```
  match 1 + 3 with 0 -> 0 | x -> 42 / x
  ```

1. まず式「1 + 3」を評価する
2. 次に定数パターン「0」と 1. の評価結果を比べる
   - マッチしないので次のパターンへ
3. 次に変数パターン「x」より、環境を拡張する
   - 変数 x を 1. の評価結果に束縛する
4. 拡張した環境で
   式「42 / x」を評価する

拡張された
環境

x = 4

(ここでは元々環境が空だったとする)

第 5 回 課題

締切: 5/31 13:00 (日本標準時)

課題 1 (10点)

- 第 4 回の課題のインタプリタを拡張して関数を扱えるようにせよ
  - 関数内で自由変数を使えるようにすること
  - 構文は自由
    - 例: $E \rightarrow$ fun I -> E
      $E \rightarrow E\ E$
  - 変数束縛式で関数を定義できるようにしてもよい
    - 例: $E \rightarrow$ let $I\ I = E$ in E

課題 2 (20点)

- インタプリタを拡張し
  再帰関数を扱えるようにせよ
  - 構文は自由
    - 例: $E \rightarrow$ let rec $I\ I = E$ in E
  - クロージャや環境の表現・実装を変更したり
    参照を用いなければ + 10 点 (つまり合計 30 点)

課題 3 (10点)

- インタプリタを拡張し
  パターンマッチングを行えるようにせよ

  – 構文は自由

    - 例:

      $E \rightarrow$ `match` E `with` P_s

      $P_s \rightarrow P_i$　　$P_s \rightarrow P_i \mid P_s$

      $P_i \rightarrow P$ `->` E

      $P \rightarrow V$　　$P \rightarrow I$

      …

課題 4 (15点)

- インタプリタを拡張し
  タプル・リストを扱えるようにせよ
  - パターンマッチングも行えるようにすること

  – 構文は自由
    - 例:
      $E \rightarrow (E, E)$
      $E \rightarrow []$　　$E \rightarrow E :: E$
      $P \rightarrow (P, P)$
      $P \rightarrow []$　　$P \rightarrow P :: P$
      …

課題 5 (15点)

- 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

  – 構文は自由

    - 例:
      $E \rightarrow$ let rec F_s in E
      $F_s \rightarrow F$　　$F_s \rightarrow F$ and F_s
      $F \rightarrow I\ I = E$

課題 6 (15点)

- インタプリタを拡張し
  再帰的な値の定義を扱えるようにせよ

  – 構文は自由
    - 例: $E \rightarrow$ let rec I = E in E

再帰的な値の定義とは

- O'Caml では以下のような
  再帰的な値の定義ができる (場合がある)

```
# let rec x = 1 :: x;;
val x : int list = [1; 1; 1; 1; 1; 1; 1; ...]
# type b = B of b;;
type b = B of b
# let rec y = B y;;
val y : b = B (B (B ...))
```

- なんでもできるわけではない

```
# let rec z = [1] @ z;;
Error: This kind of expression is not allowed as right-hand
side of `let rec'
```

  - 詳細は以下を参照
    - http://caml.inria.fr/pub/docs/manual-ocaml/manual021.html#toc70

課題 7 (20点)

- 第 3 回の課題 9 を修正して
  λ 抽象と関数適用を実装せよ

課題 8 (15 点)

- 以下のような自然数を表す型

```
type nat =
  | Zero
  | Succ of nat
```

と同様の型を再帰型を用いずに定義せよ

  - またその型の値や値に対する操作も定義せよ

課題 8 のヒント

- まず、以下のような signature を持つ functor LFIX を再帰型や再帰関数、参照を用いずに定義する

```
module LFIX :
functor (T : sig type 'a t
  val conv : ('a -> 'b) -> 'a t -> 'b t end) ->
sig
  type fix_t = { f : 'a. ('a T.t -> 'a) -> 'a; }
  val fold : ('a T.t -> 'a) -> fix_t -> 'a
  val in_f : fix_t T.t -> fix_t
end
```

課題 8 のヒント

- 次に、以下のような signature を持つ module Nat_t を再帰を使わず定義する

```
type 'a nat_t =
 | Zero_t
 | Succ_t of 'a

module Nat_t :
sig
 type 'a t = 'a nat_t
 val conv : ('a -> 'b) -> 'a t -> 'b t
end
```

課題 8 のヒント

- さらに functor LFIX に Nat_t を適用する
  - Nat_t にLFIXを適用した module を Nat とする

課題 8 のヒント

- Nat を用いると以下のような関数を
  再帰型・再帰関数を用いずに定義できるはず

```
val zero : Nat.fix_t
val succ : Nat.fix_t -> Nat.fix_t
val is_zero : Nat.fix_t -> bool
val pred : Nat.fix_t -> Nat.fix_t
val add : Nat.fix_t -> Nat.fix_t -> Nat.fix_t
val sub : Nat.fix_t -> Nat.fix_t -> Nat.fix_t
val mul : Nat.fix_t -> Nat.fix_t -> Nat.fix_t
```

課題 9 (10点)

- 整数のリスト

```
type int_list =
  | Nil
  | Cons of int * int_list
```

と同様の型を再帰型を用いずに定義せよ

  - またその型の値や値に対する操作も定義せよ