関数・論理型プログラミング実験
第６回

江口 慎悟
酒寄 健
塚田 武志
松下祐介

講義のサポートページ

http://www.kb.is.s.u-tokyo.ac.jp/~tsukada/cgi-bin/m/

- 講義資料等が用意される
- レポートの提出先
- 利用にはアカウントが必要

- 名前/学籍番号/希望アカウント名をメールを
  tsukada@kb.is.s.u-tokyo.ac.jp
  までメールしてください。
  - 件名は「FL/LP実験アカウント申請」
  - アカウント名/パスワードを返信
  - PC からのメールを受け取れるように

インタプリタを作る（全５回）

第５回　基本的なインタプリタの作成
- 字句解析・構文解析、変数の扱い方

第６回　関数型言語への拡張
- 関数定義・呼び出し機構の作成

第７回　型システムと単純型推論
- 単純型検査器

第８回　単一化、let多相
- 単一化の定義とアルゴリズム、let多相

第９回　様々な拡張
- パターンマッチング

実行

型けんさで
やばいprogramを
はじく．

今日の内容

- 関数定義・呼び出し機構
  - 非再帰関数
  - 再帰関数

（非再帰）関数の扱い方

素朴なアプローチ（失敗例）

クロージャ

目標

o 関数抽象と適用を扱えるようにする

- 構文：

（fun 式）　　`fun x -> e`

（関数適用）　`e e'`


```
type expr = … | EFun of name * expr
              | EApp of expr * expr
```

fun式の「値」はなんだろう？

o 値の「使われ方」を考える

(fun x -> e) v の評価

x を v に束縛して e を評価

xがvという値を覚えて、
そのもとでeを評価
↑ これはうまくいかない
一工夫必要

（非再帰）関数の扱い方

素朴なアプローチ（失敗例）

クロージャ

素朴なアプローチ

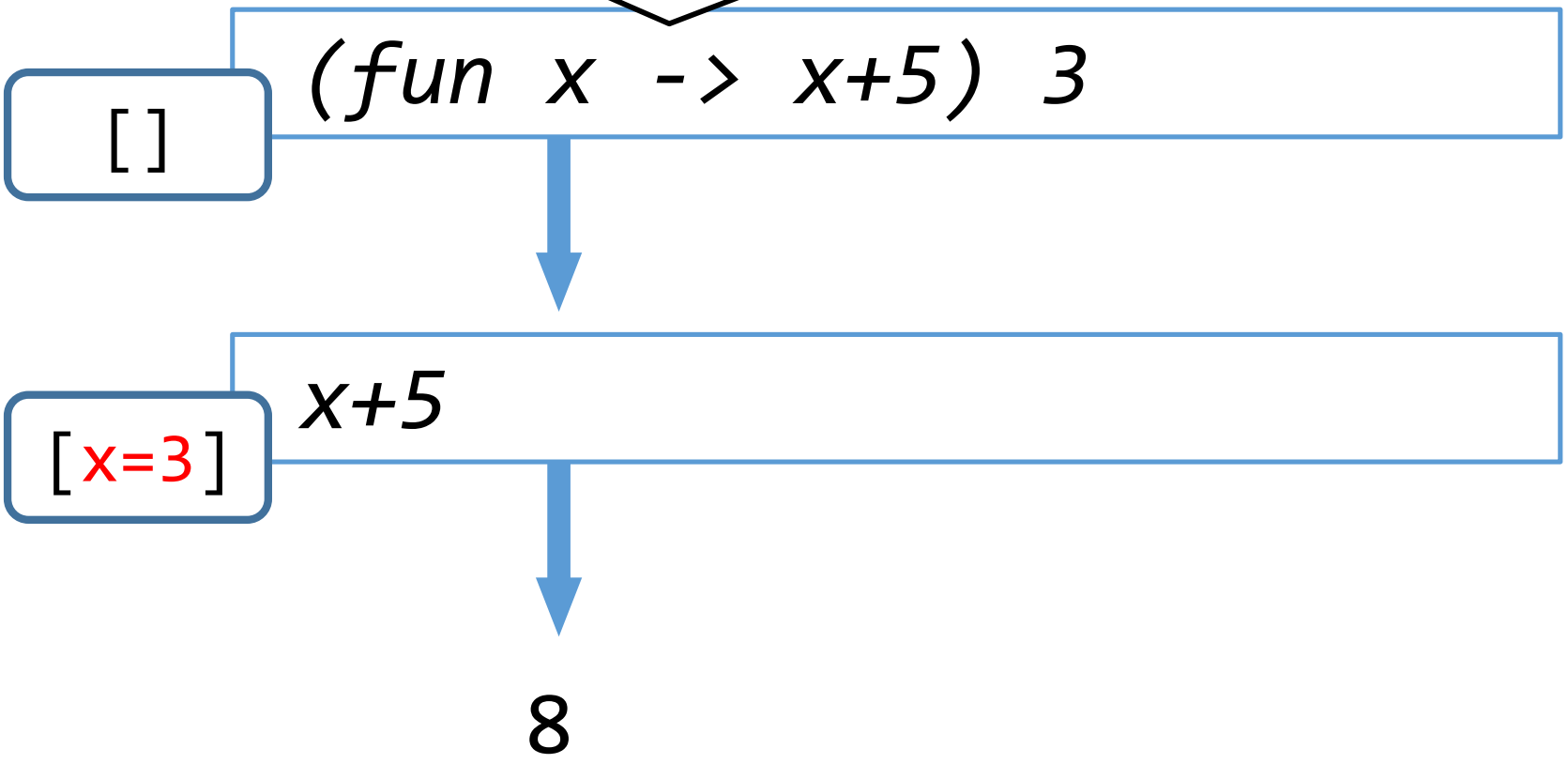
- fun 式そのものが値

```
type expr  = … | EFun of name * expr | …
type value = … | VFun of name * expr
```

```
let rec eval_expr env expr = match expr with
  | …
  | EFun (x,e)    -> VFun (x,e)
  | EApp (e1,e2) ->
      let VFun (x,e) = eval_expr env e1 in
      let v2 = eval_expr env e2 in
        eval_expr (extend x v2 env) e
```

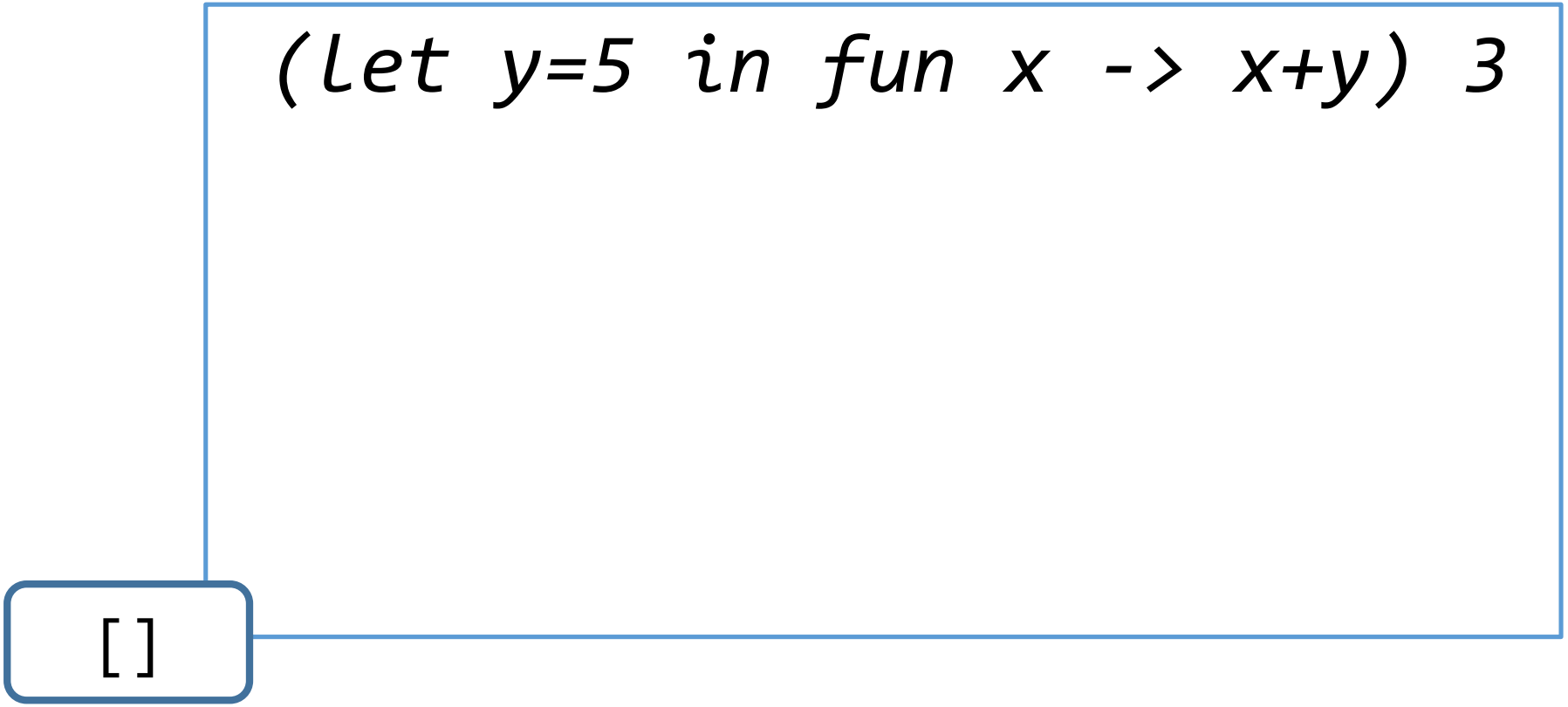
例：

```
EApp(
  EFun("x",EAdd(EVar "x", EConstInt 5)),
  EConstInt 3)
```

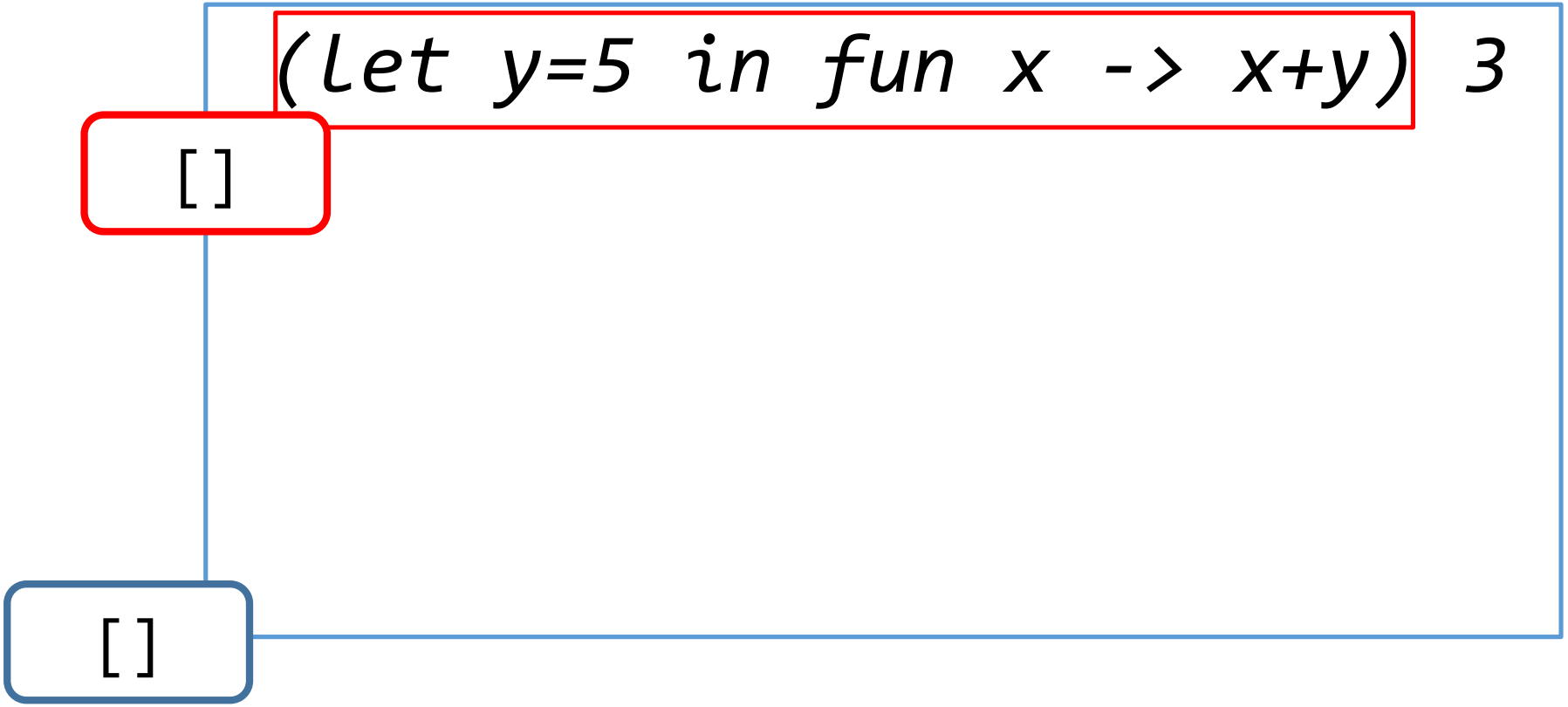


上手く行きそう？

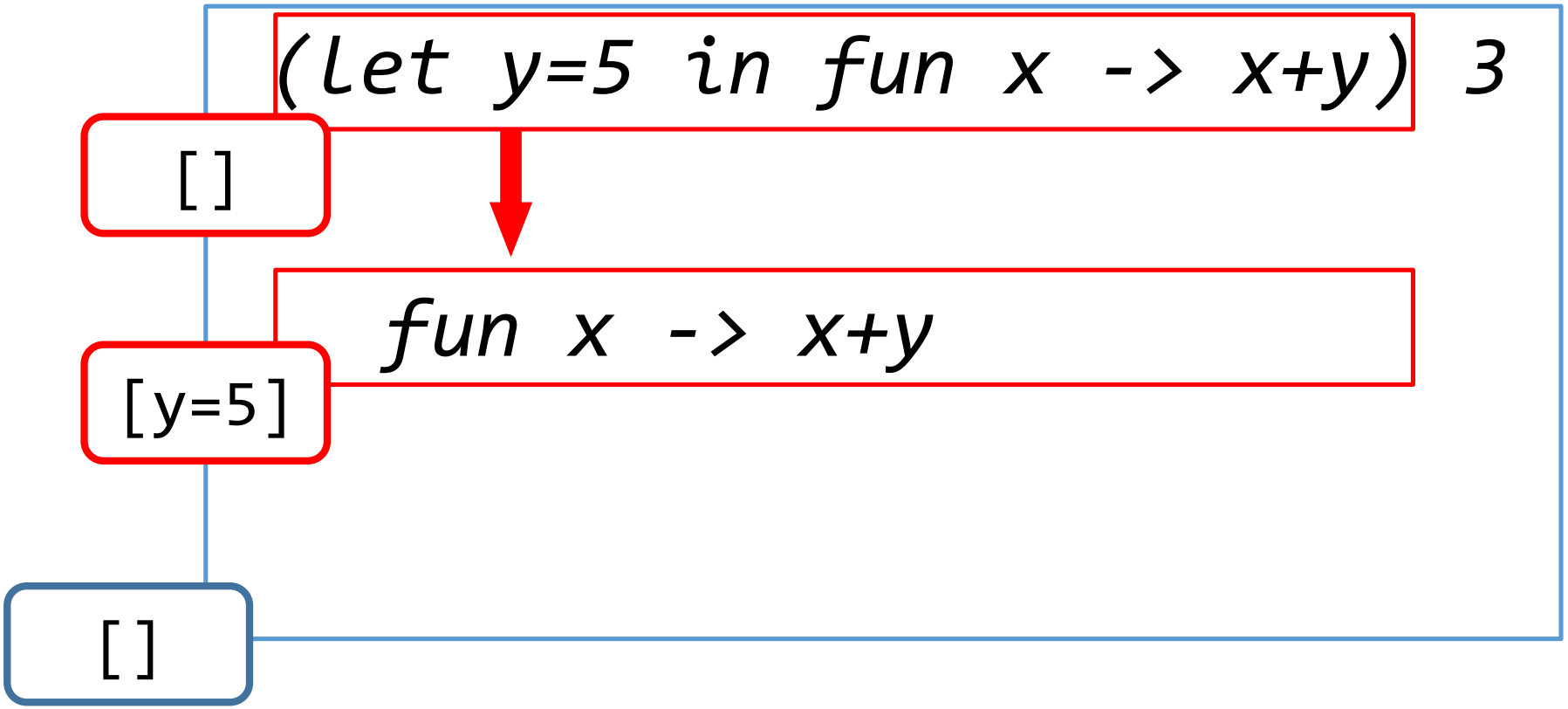
うまく行かない例



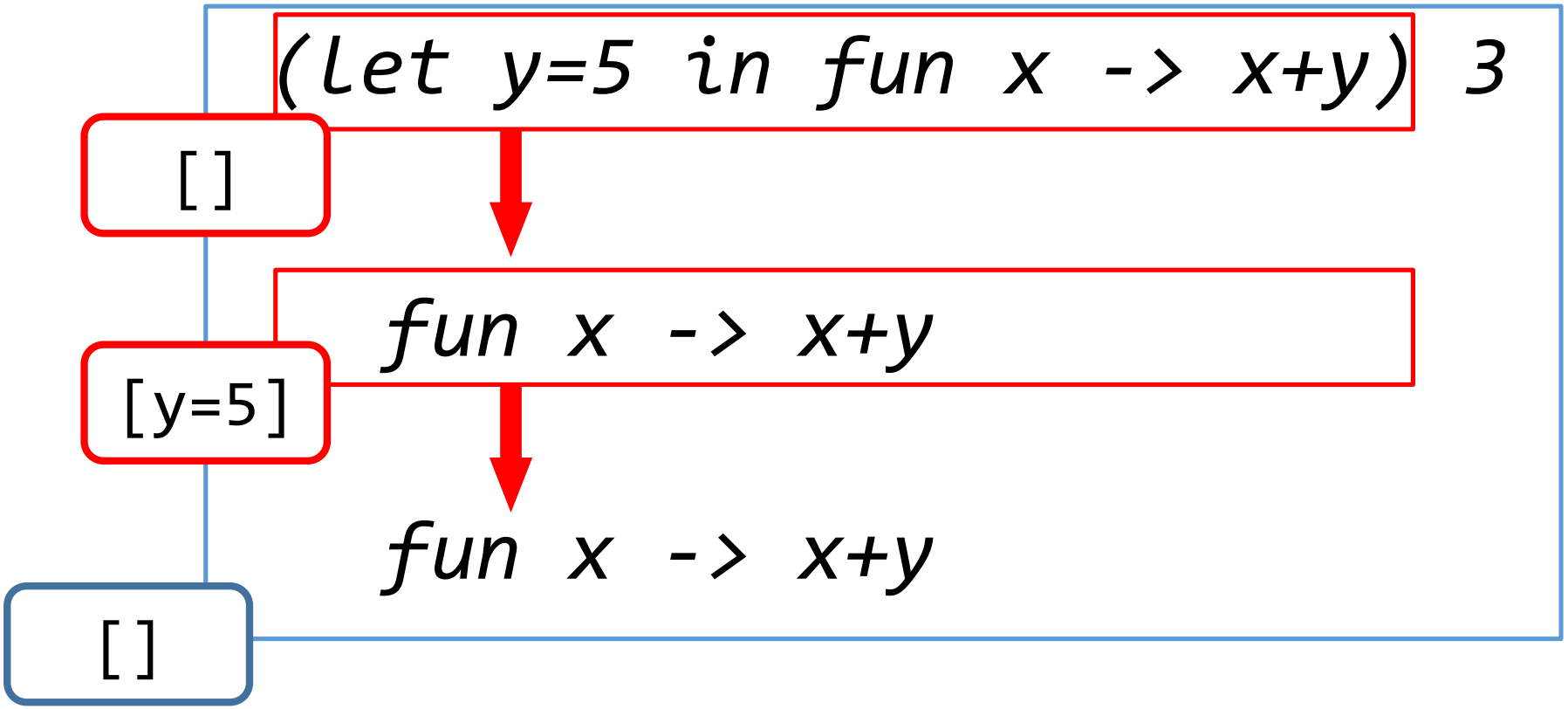
うまく行かない例



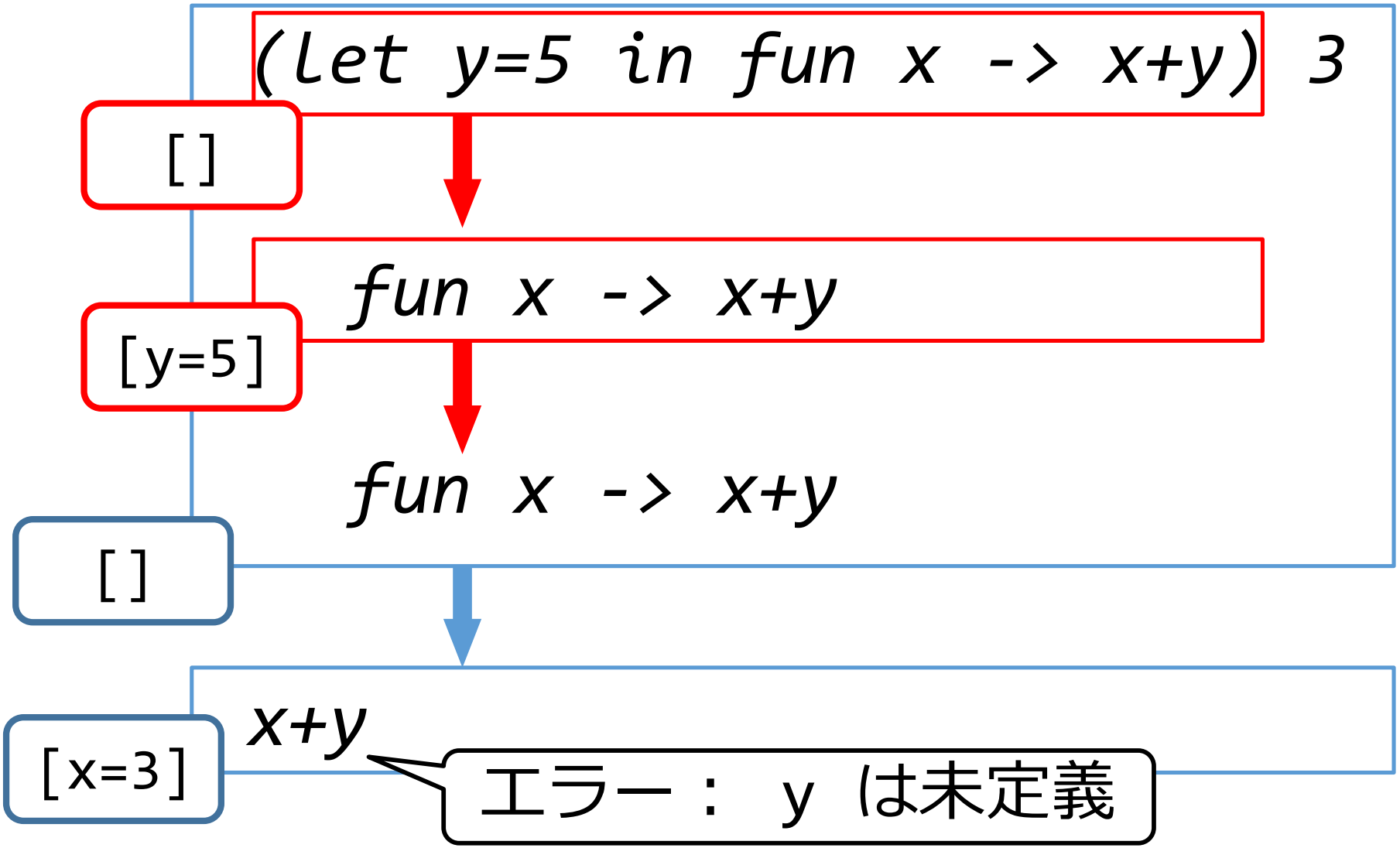
うまく行かない例



うまく行かない例

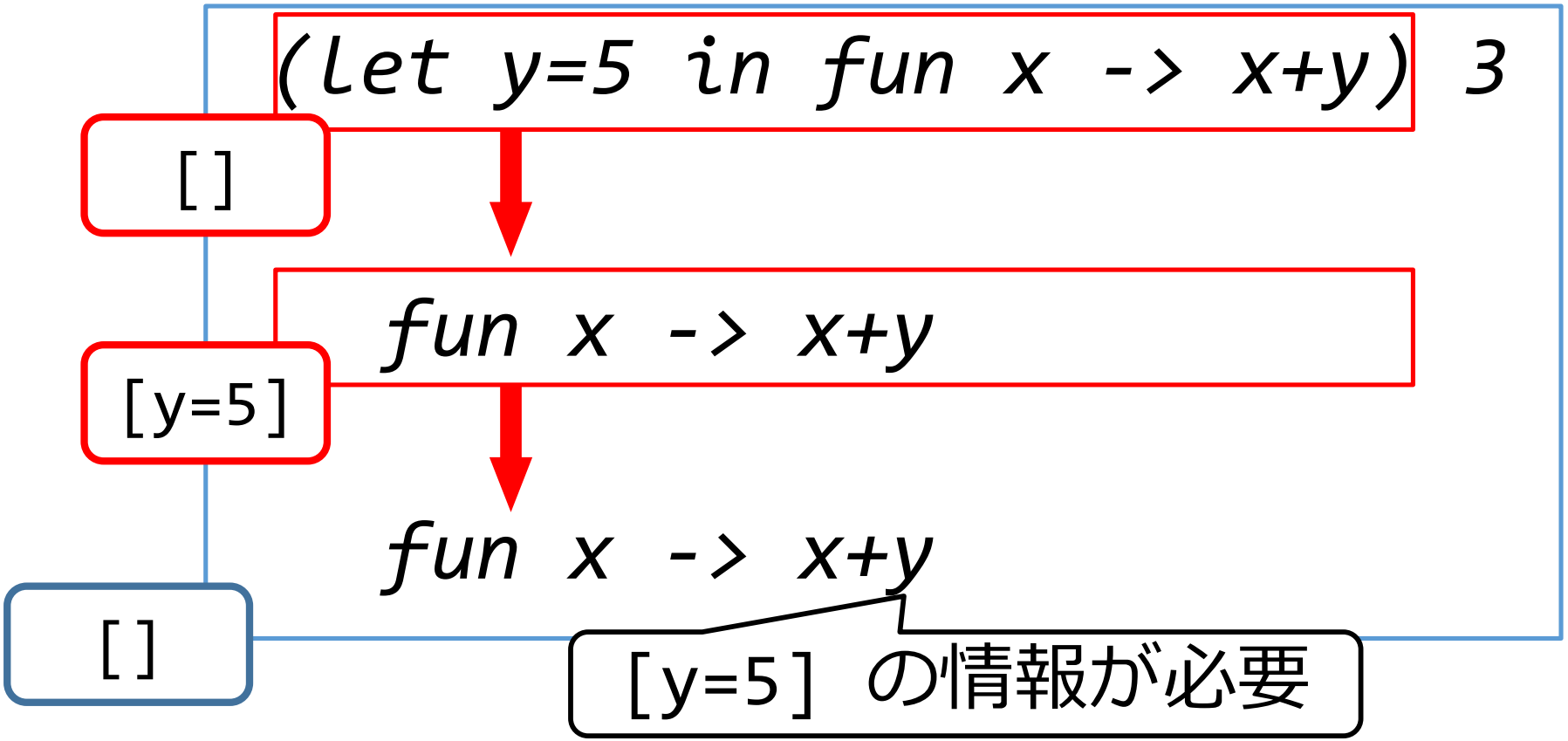


うまく行かない例



観察

o 関数値を作ったときの環境が必要



観察

```
fun x -> e
```

- e はこの式の外側で定義された変数を含む
  自由変数

- fun 式の評価時に自由変数の値の情報を
  忘れてしまったことがエラーの原因

（非再帰）関数の扱い方

素朴なアプローチ（失敗例）

クロージャ

クロージャ

- fun 式の評価結果の値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

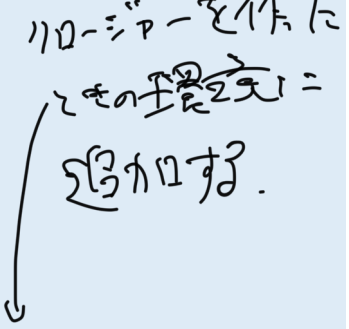
```
type expr  = … | EFun of name * expr | …
type value = … | VFun of name * expr * env
```

```
let rec eval_expr env expr = match expr with
   | …
   | EFun (x,e) -> VFun (x,e,env)
   | …
```

クロージャの適用

o クロージャの持つ環境に束縛を追加して
　関数本体の式を評価する

```
let rec eval_expr env expr = match expr with
  | …
  | EApp (e1,e2) ->
      let v1 = eval_expr env e1 in
      let v2 = eval_expr env e2 in
      (match v1 with
        | VFun(x,e,oenv) ->
            eval_expr (extend x v2 oenv) e
        | _ -> raise Eval_Error)
```

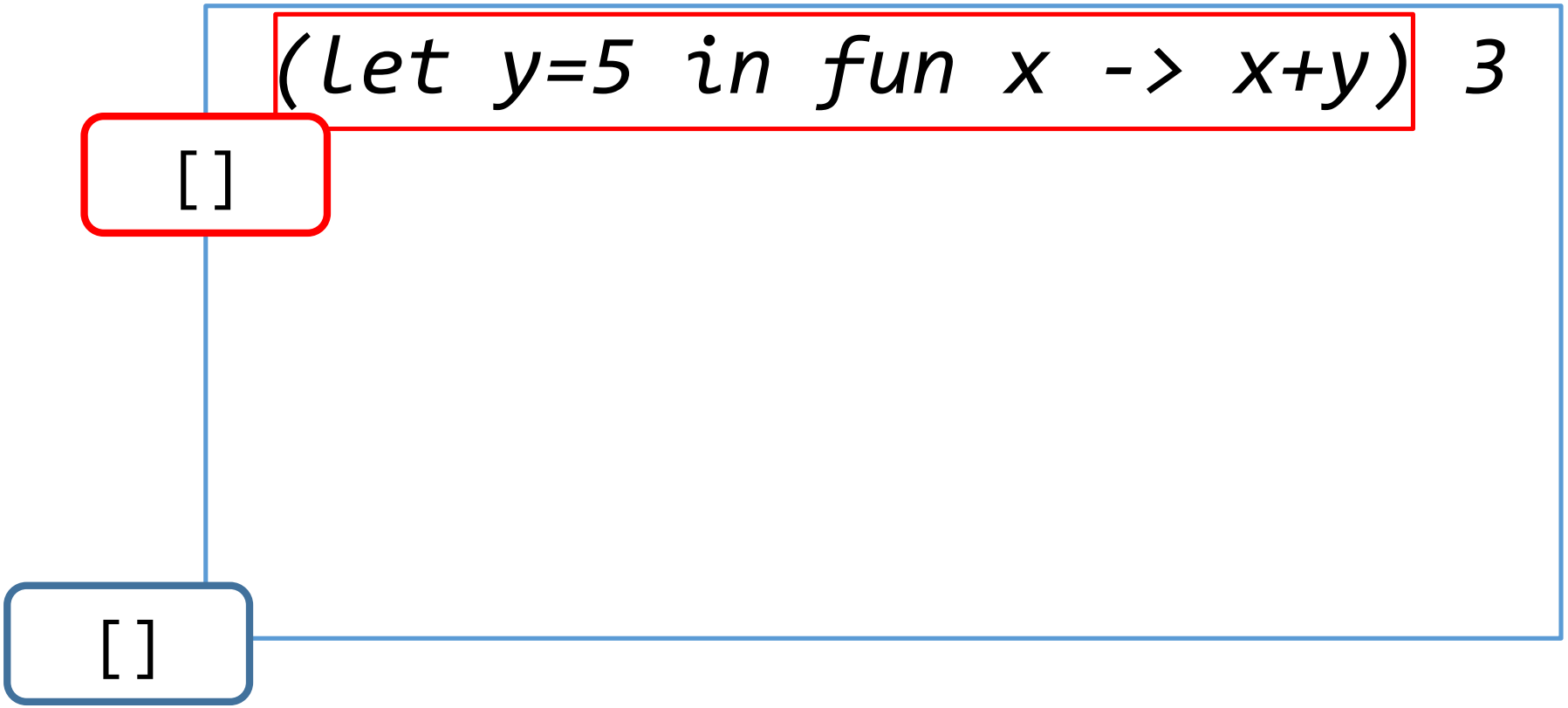


例

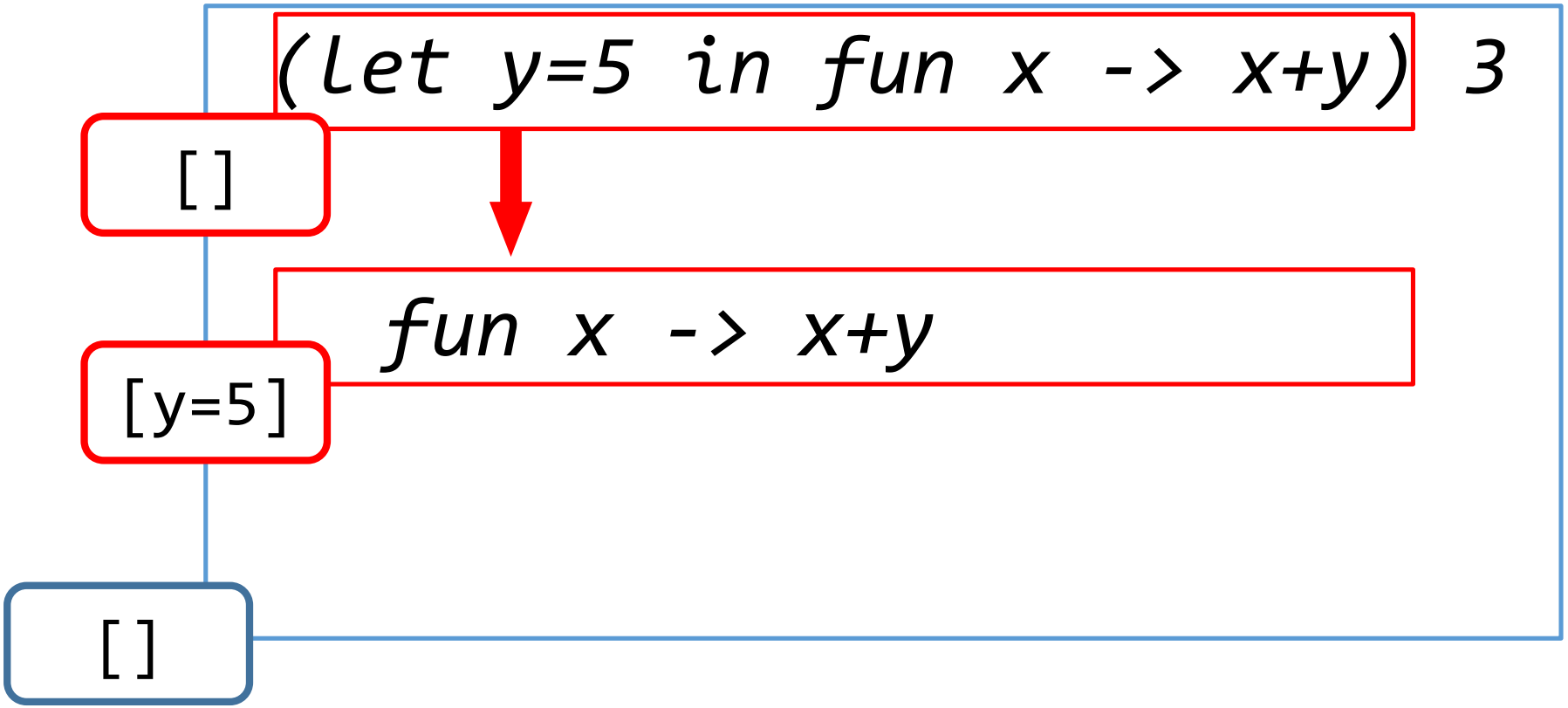
```
(let y=5 in fun x -> x+y) 3
```

[]

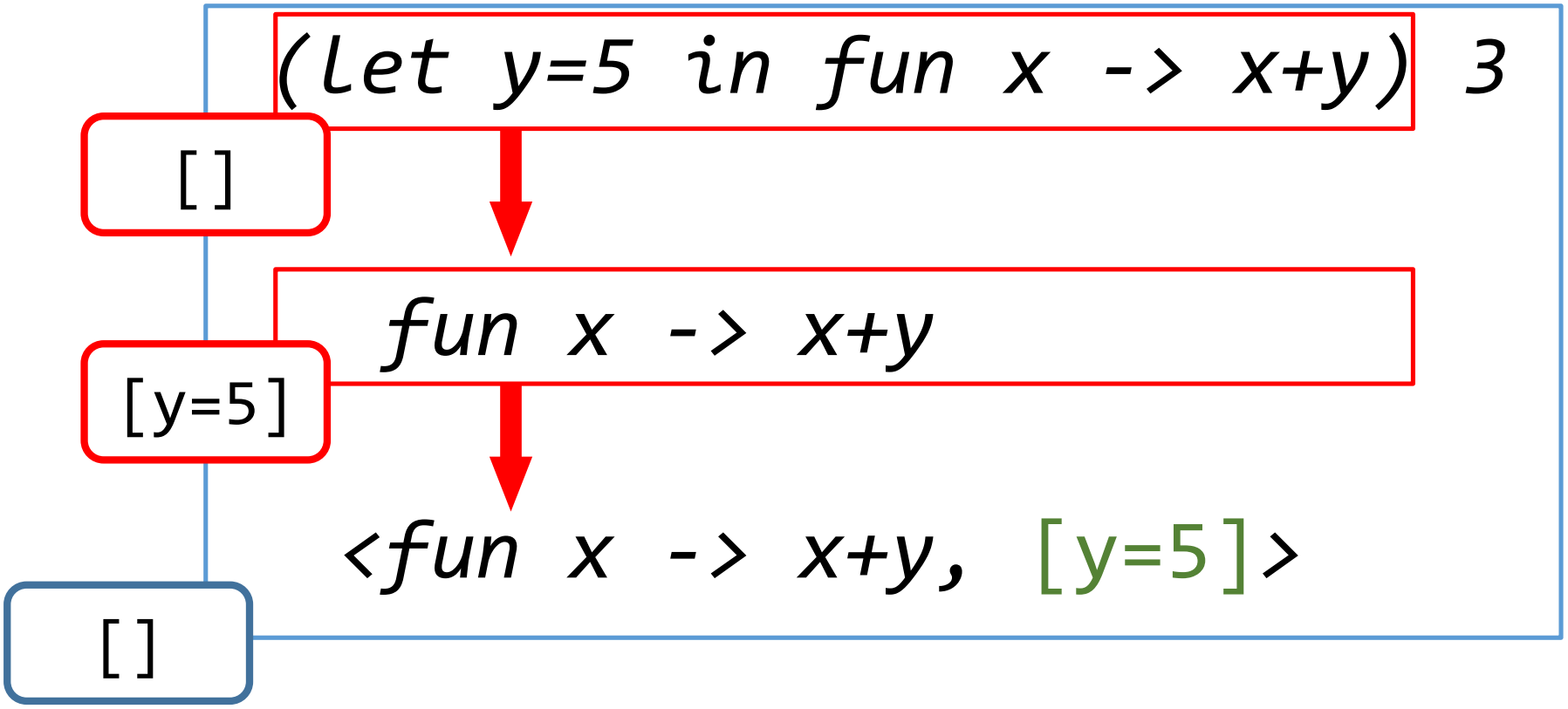
例



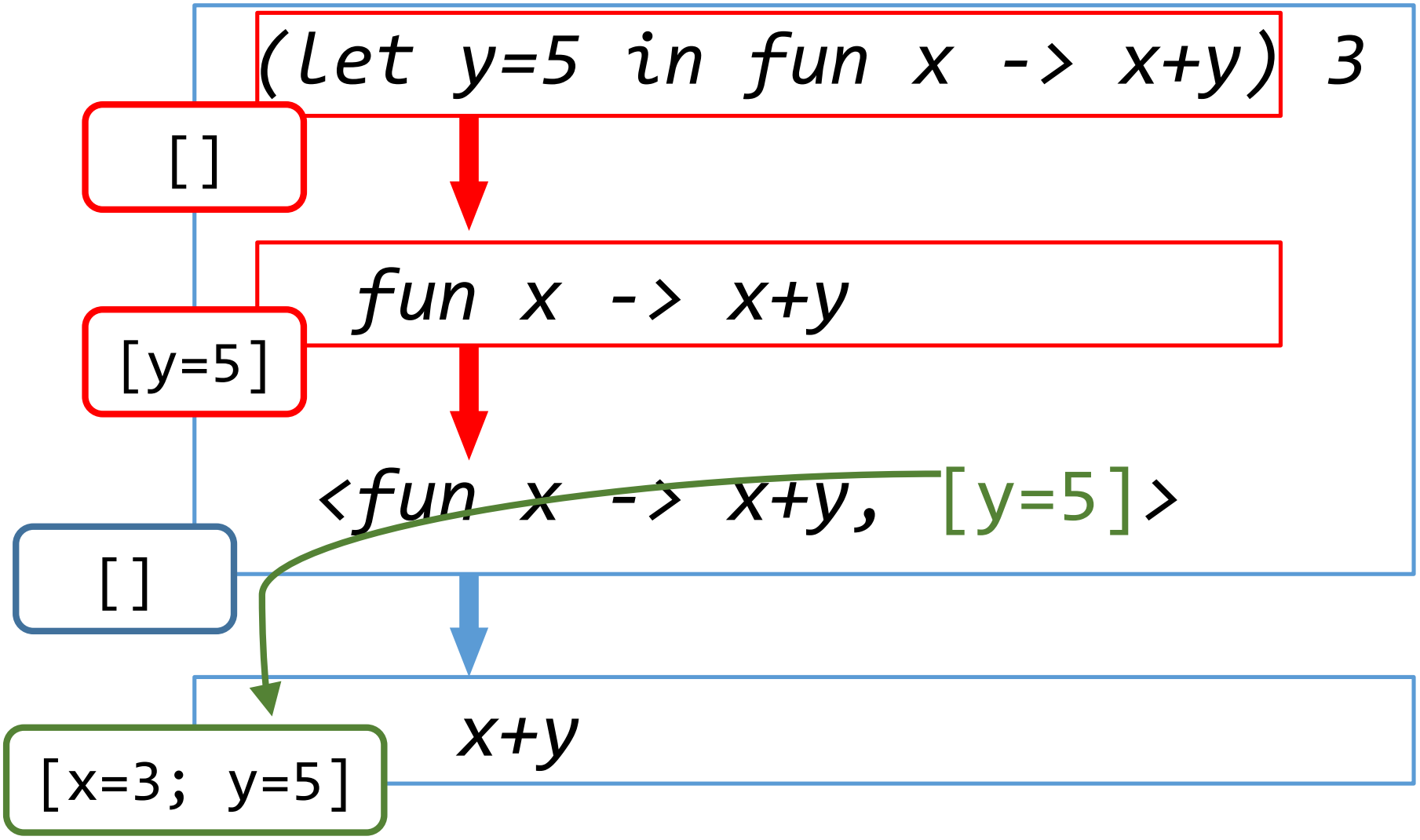
例



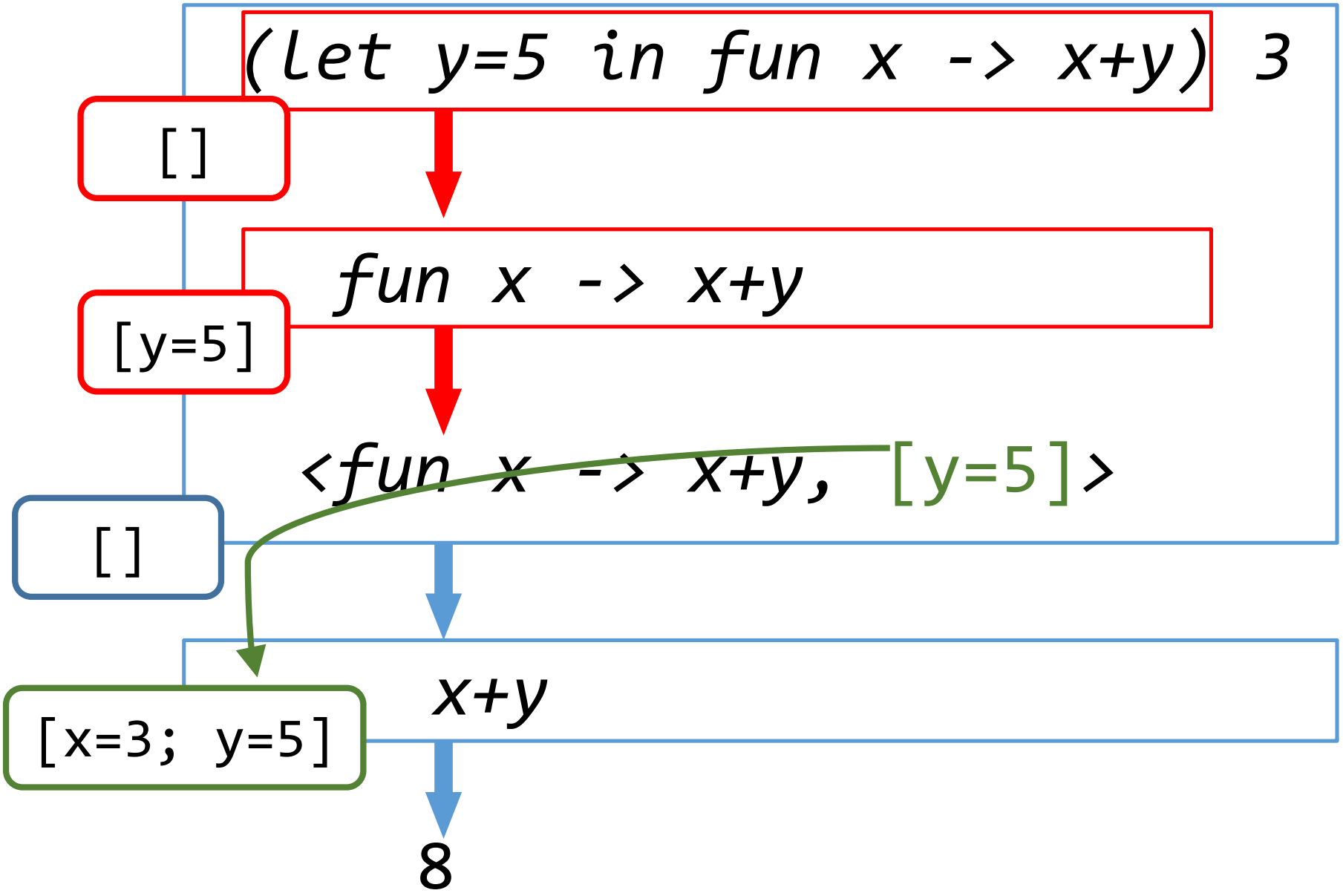
例



例



例



余談：動的束縛

o 「素朴な方法」は動的スコープに対応

```
# let a = 10;;
val a = 10
# let f x = x + a;;
val f = <fun>
# let a = 20;;
val a = 20
```

- 静的スコープ

```
# f 5;;
- = 15
```

動的スコープ

```
# f 5;;
- = 25
```

再帰関数の扱い方

クロージャと関数適用を工夫する
環境を工夫する

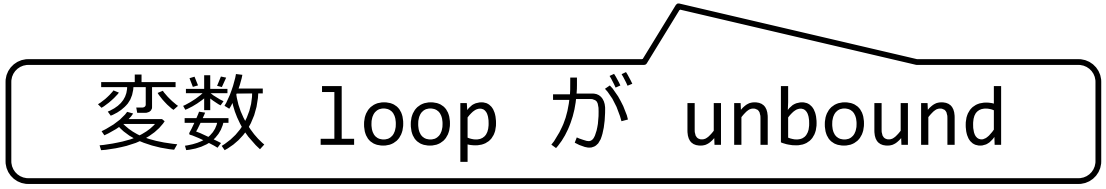
再帰関数の困難

o loop の評価結果の値は？

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

- 普通のクロージャではダメ

<fun n -> loop n, []>



再帰関数の実装方法

o ２種類の方法を紹介する

- クロージャと関数適用を工夫する
  - 再帰関数クロージャと普通のクロージャを区別する
  - 再帰関数クロージャへの関数適用のときに
    引数への束縛と共に自分自身への束縛も環境に追加する

- 環境を工夫する
  - クロージャは環境への参照を持つようにする
  - クロージャの作成のときに
    自分自身への束縛を環境に追加する

再帰関数の扱い方

クロージャと関数適用を工夫する

環境を工夫する

再帰関数用クロージャ

```
type expr  = …
   | EFun    of name * expr
   | ELetRec of name * name * expr * expr
   | …
type value = …
   | VFun    of name * expr * env
   | VRecFun of name * name * expr * env
```

自分の名前

引数の名前

定義式

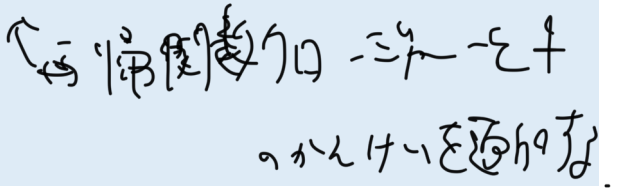
環境

再帰関数用クロージャの生成

```
type expr  = …
  | ELetRec of name * name * expr * expr | …

type value = …
  | VRecFun of name * name * expr * env
```

```
let rec eval_expr env expr = match expr with
  | …
  | ELetRec (f,x,e1,e2) ->
      let env' =
        extend f (VRecFun(f,x,e1,env)) env
      in
        eval_expr env' e2
```



例

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

```
o loop =
    VRecFun("loop", "n", Loop n, [])
```

ここにはループの情報がまだ入ってない

2引数のときにがんばる

再帰関数用クロージャの適用

o 引数と自分自身を環境に加えて本体を評価

```
let rec eval_expr env expr = match expr with
  | …
  | EApp(e1,e2) ->
      let v1 = eval_expr env e1 in
      let v2 = eval_expr env e2 in
      (match v1 with …
        | VFun    (x,e,oenv) -> …
        | VRecFun (f,x,e,oenv) ->
            let env' =
              extend x v2
                     (extend f (VRecFun(f,x,e,oenv)) oenv)
            in
              eval_expr env' e
      )
```

引数の情報

自分自身の情報

再帰関数の扱い方

クロージャと関数適用を工夫する

環境を工夫する

環境を工夫する

o クロージャが環境の参照を取るようにする

```
type expr  = …
  | EFun    of name * expr
  | ELetRec of name * name * expr * expr

type value = …
  | VFun    of name * expr * env ref
```

クロージャの生成

```
type expr  = …
  | EFun    of name * expr
  | ELetRec of name * name * expr * expr

type value = …
  | VFun    of name * expr * env ref
```

```
let rec eval_expr env expr = match expr with …
  | EFun    (x,e)          ->
      VFun (x, e, ref env)
  | ELetRec (f,x,e1,e2) ->
      let oenv = ref [] in
      let v = VFun(x, e1, oenv) in
        (oenv := extend f v env;
         eval_expr (extend f v env) e2)
```

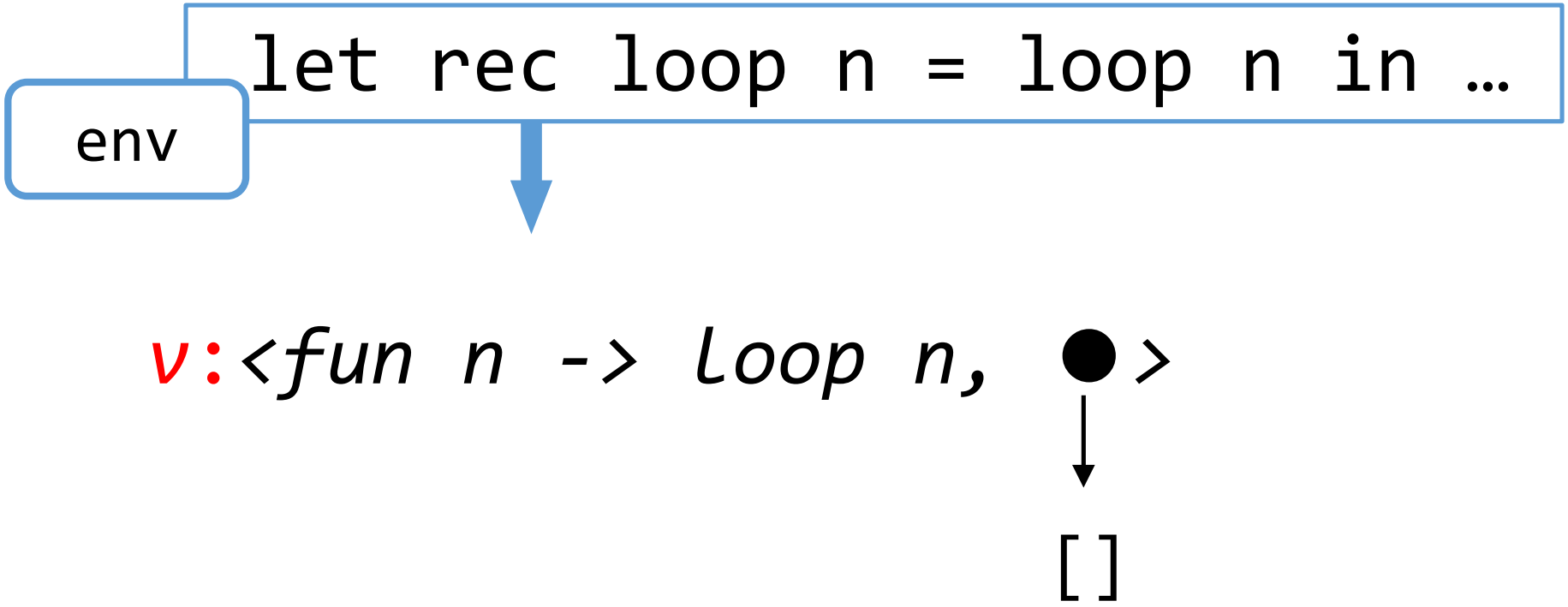
ダミーの環境

環境を更新

e2を拡張するかんきょうにしてる

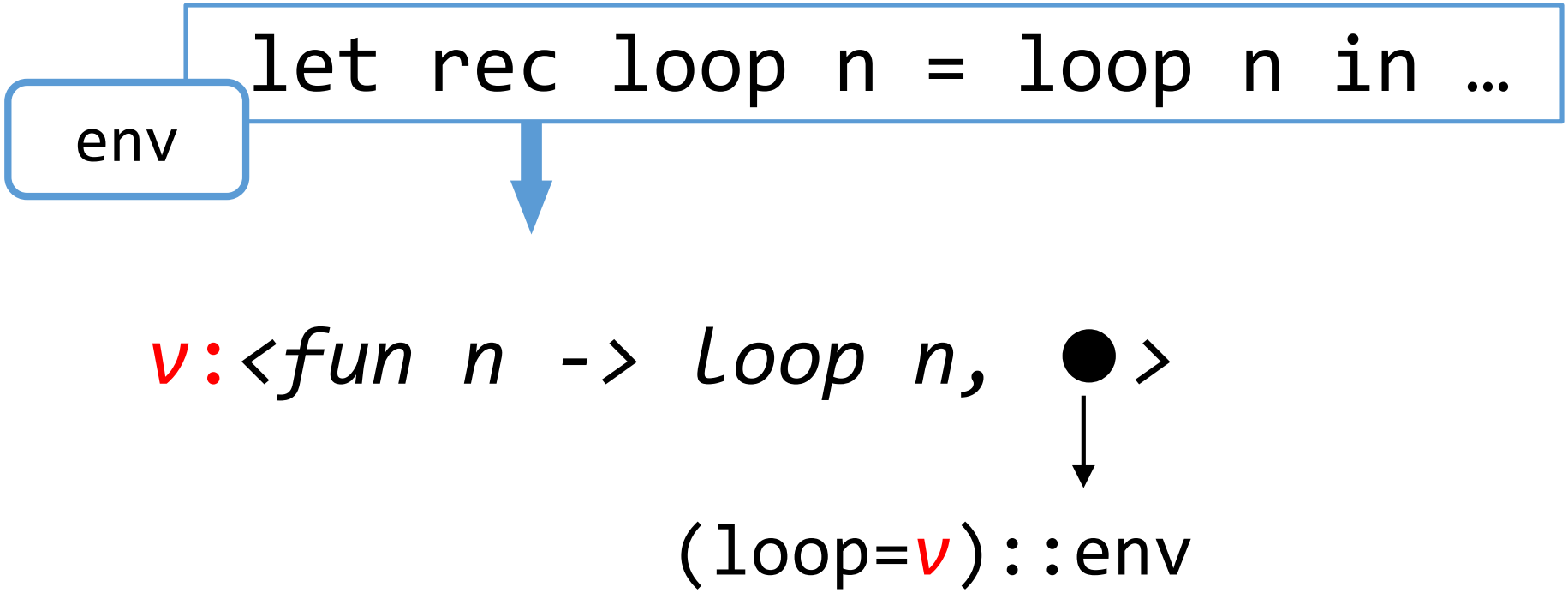
循環的なクロージャ

o 再帰関数のクロージャは循環的になっている



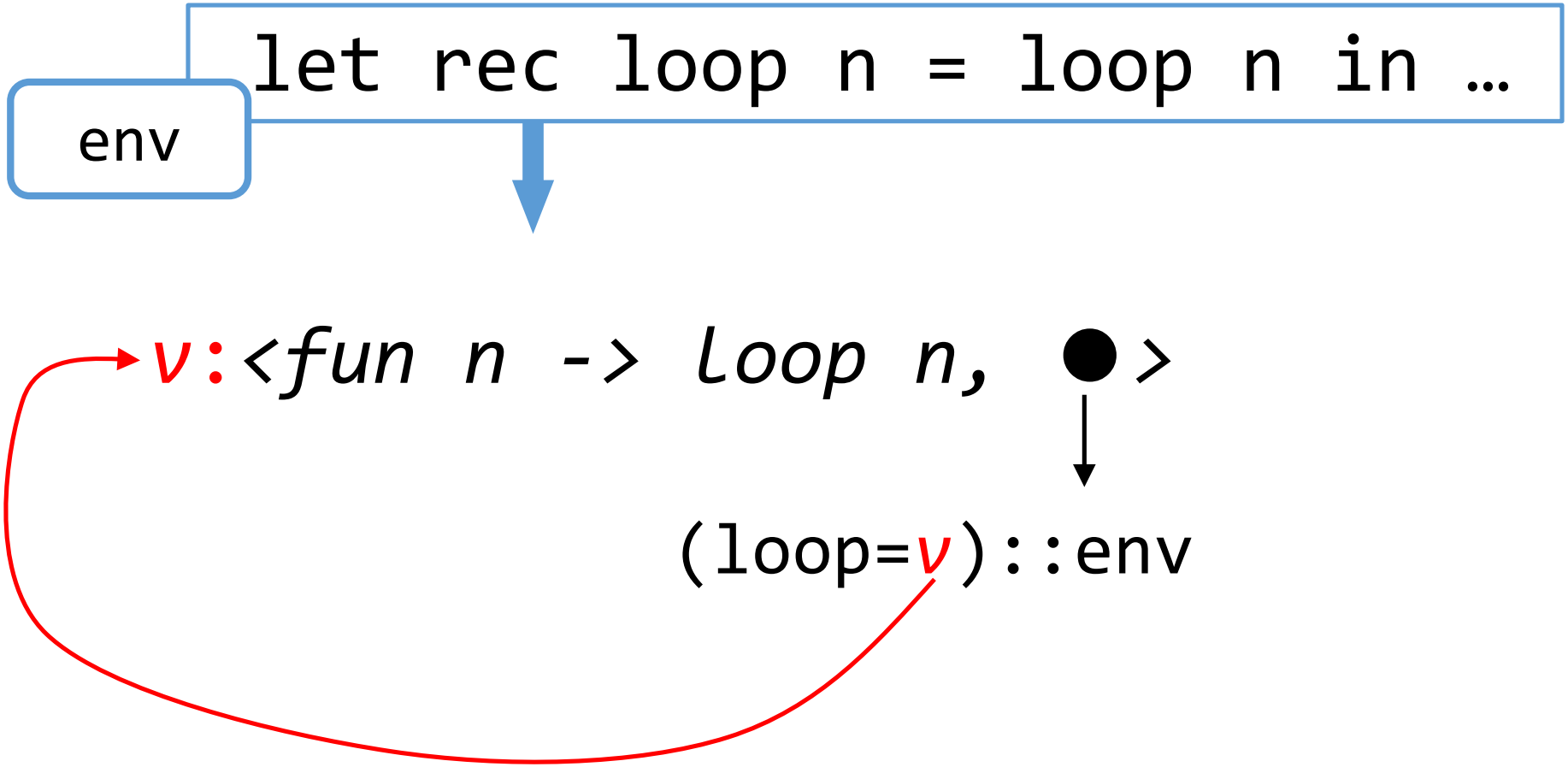
循環的な環境

o 再帰関数のクロージャは循環的になっている



循環的な環境

o 再帰関数のクロージャは循環的になっている



関数適用の評価

- 非再帰関数とほぼ同じ
  - 参照の dereference が必要になるのだけが違い

```
let rec eval_expr env expr = match expr with
  | …
  | EApp (e1,e2) ->
      let VFun (x,e,oenv) = eval_expr env e1 in
      let v2 = eval_expr env e2 in
        eval_expr (extend x v2 (!oenv)) e
```

ポインタなぞ出いなかったから

例題

理解の確認をするための課題です
課題提出システム上での提出の必要はありません
例題を解きTAに見せることで出席とします
分からないことがあったら、積極的に質問しましょう

例題

- （非再帰）関数を扱うことができるように value 型を拡張せよ
  - 前回の課題の value型にクロージャを追加する
  - 環境の型と相互再帰で定義する必要がある

TAチェック

- value型の定義
  - 例題だけでなく問１まで解くのがお勧め
- **OCaml のバージョン**

レポート課題６

締切：**2019/6/11** 13:00(JST)

問 1

o 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

- 構文は以下をサポートすること
  - fun 変数 -> 式
  - 式 式

- 参考資料の ex1 に適当な eval.ml を与えればよい

問２

o 問１のインタプリタを拡張し、
再帰関数を扱えるようにせよ

- 構文は以下をサポート
  - `let rec 変数 変数 = 式`
  - `let rec 変数 変数 = 式 in 式`

- 参考資料の ex2 に適当な eval.ml を与えればよい
- 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

- 問２のインタプリタを改良し、相互再帰を扱えるようにせよ
  - 構文は以下をサポート
    - let rec f_1 x_1 = e_1
      and f_2 x_2 = e_2
      …
      and f_n x_n = e_n in e
    - let rec f_1 x_1 = e_1
      and f_2 x_2 = e_2
      …
      and f_n x_n = e_n

構文解析器は、参考資料のex3 のものを用いてよい

方法１

- 再帰関数用のクロージャの定義を変え、
  f_i を次の値に束縛すればよい

  $<i, [(f_1,x_1,e_1);\ldots;(f_n,x_n,e_n)], env>$

- f_i の v への適用は、次のようにできる

  環境 env に、
  - f_1 を $<1,[(f_1,x_1,e_1);\ldots;(f_n,x_n,e_n)],env>$に束縛
  - …
  - f_n を $<n,[(f_1,x_1,e_1);\ldots;(f_n,x_n,e_n)],env>$に束縛
  - x_i を v に束縛

  を追加してから、e_i を評価

方法２

o 参照を利用し循環的クロージャを作ればよい

o 循環的クロージャは次のように作れる

- はじめにダミー環境への参照 oenv を作る
- 各 i について、次のクロージャを作り、vi とする

  $\mathtt{VFun}(x_i,\ e_i,\ \mathtt{oenv})$
- oenv を次の環境に書き換える

  $(f_1,v_1)::(f_2,v_2)::\ \ldots\ ::(f_n,v_n)::\mathtt{env}$

問４

o 問２とは異なる方法で、
　再帰関数を扱える評価器を書け

- 参考資料の ex2 に適当な eval.ml を与えればよい
- 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

発展 1

o 関数の定義のための略記法をサポートせよ

- 例：

```
fun x y z -> e
let f x y = e in e'
```

ref に近い. loop こうぞう. ← これを ref を使わないで, がんばる.

発展２

o 拡張機能の「値の再帰的定義」を使うことで、
問１からほとんど変更することなく、
再帰関数をサポートすることができる

- value 型の定義は変更する必要がない
- eval 関数の定義も、既存の部分を変更せず、
新しい構文（ELetRec）のケースを
追加するだけでよい

o このような方法で再帰関数をサポートせよ

補足：値の再帰定義 (Recursive definitions of values)

- 関数以外の値も再帰的定義ができる
  - 定義式に許される形の詳細はマニュアルを参照
  - 変数、定数とコンストラクタから成る式はOK

```
# let rec x = 1 :: x;;
val x : int list =
[1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; …]

# let rec y = 2 :: z
  and     z = 3 :: y;;
val y : int list =
[2; 3; 2; 3; 2; 3; 2; 3; 2; 3; 2; 3; 2; 3; 2; …]
val z : int list =
[3; 2; 3; 2; 3; 2; 3; 2; 3; 2; 3; 2; 3; 2; 3; …]
```

発展３

o インタプリタを拡張し、
　動的束縛の関数定義も扱えるようにせよ
- 構文例： `dfun x -> e`

o 動的束縛があれば、特別な構文がなくとも
　再帰関数を定義することができる
- 動的束縛を用いて階乗関数 fact を定義せよ
- 一般的な再帰の定義方法について議論せよ

発展 3

o 動作例

```
# let a = 10;;
val a = 10

# let f = (dfun x -> x + a);;
val f = <fun>

# let a = 20;;
val a = 20

# f 10;;
- = 30
```

注意

- 問１・問２・問３はまとめて提出してもよい
  - どの問に答えているかは明らかにすること
  - 考察はそれぞれ行うこと
- OCaml の標準ライブラリは自由に用いてよい
- ビルド方法の記述は忘れないように
- Conflict は可能な限り消すこと
- 参考資料のコードを使う必要はない