プログラミング言語処理系
MiniMLインタプリタ

末永 幸平

この資料について

- 京都大学工学部専門科目「プログラミング言語処理系」の講義資料
- 講義 Web ページ:
  - https://kuis-isle3sw.github.io/IoPLMaterials/
- 講義をする人: 末永幸平
  - https://researchmap.jp/ksuenaga/
  - https://twitter.com/ksuenaga

アウトライン

- MiniML1 インタプリタ
  - 文脈自由文法と BNF
  - MiniML1 インタプリタ
    - ソースコードとビルドの仕方
    - 各モジュールの説明
    - 字句解析と構文解析のためのツール
- MiniML2 インタプリタ: 変数定義の導入
- MiniML3 インタプリタ: 関数値の導入
- MiniML4 インタプリタ: 再帰関数の定義

インタプリタパートの講義計画

- OCaml のサブセットである MiniML 言語のインタプリタを作成

- シンプルな言語から始めてだんだん機能拡張
  - MiniML1: めっちゃサブセット（変数定義すらできない）
  - MiniML2: MiniML1 + 変数定義
  - MiniML3: MiniML2 + 再帰でない関数の定義・呼び出し
  - MiniML4: MiniML3 + 再帰関数定義・呼び出し

MiniML1

- 整数値，真偽値，条件分岐，加算乗算，変数の参照を備えた言語

- プログラム例
  - 3
  - true
  - x
  - if x then 3 else y
  - 3 + x
  - (3 + x1) * false

MiniML1を定義するためにやること

- シンタックスの定義
- セマンティクスの定義
  - （OCaml と大体同様なので省略）

文脈自由文法

- 語の集合を文法によって定義するための数学的ツール

$$(V, \Sigma, R, S)$$

- V: 非終端記号の集合
- Σ: 終端記号の集合
- R: 書き換え関係
  - V と $V \cup \Sigma$ の要素の間の関係
- S: 開始記号
  - V の元

例: a と b からなる回文を生成する文法

- 語の集合を文法によって定義するための数学的ツール

$$\mathbf{(V, \Sigma, R, S)}$$

- $V = \{S\}$
- $\Sigma = \{a, b\}$
- $R = \{ S \rightarrow a\ S\ a,\ S \rightarrow b\ S\ b,\ S \rightarrow a,\ S \rightarrow b,\ S \rightarrow \varepsilon \}$

シンタックスを定義するためのツール: BNF

- 文脈自由文法を簡便に記述するための記法
  - プログラミング言語のシンタックスの定義でよく出てくる

- 実際に教科書の例を見ながら説明

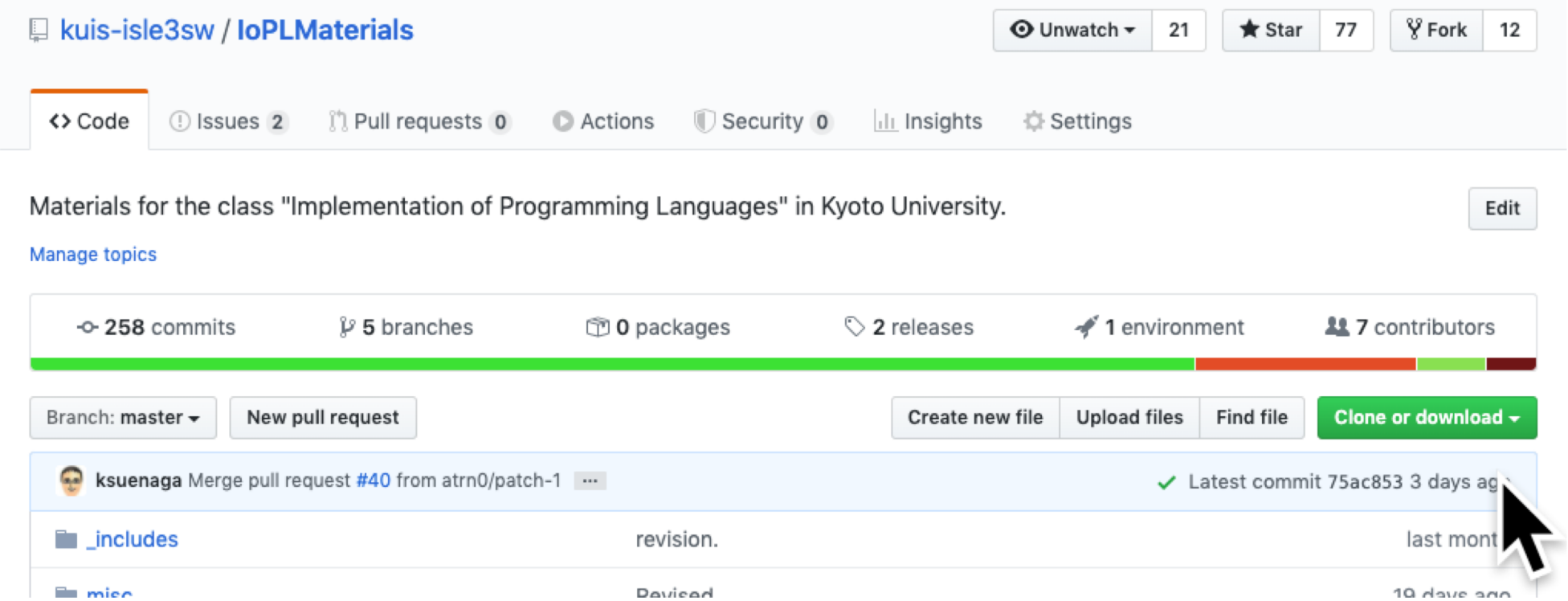
```
P ::= e ;;
b ::= true | false
e ::= <識別子> | <自然数リテラル> | b | e op e |  if e then e else e |  ( e )
op ::= + | * | <
```

アウトライン

- MiniML1 インタプリタ
  - 文脈自由文法と BNF
  - MiniML1 インタプリタ
    - ソースコードとビルドの仕方
    - 各モジュールの説明
    - 字句解析と構文解析のためのツール
- MiniML2 インタプリタ: 変数定義の導入
- MiniML3 インタプリタ: 関数値の導入
- MiniML4 インタプリタ: 再帰関数の定義

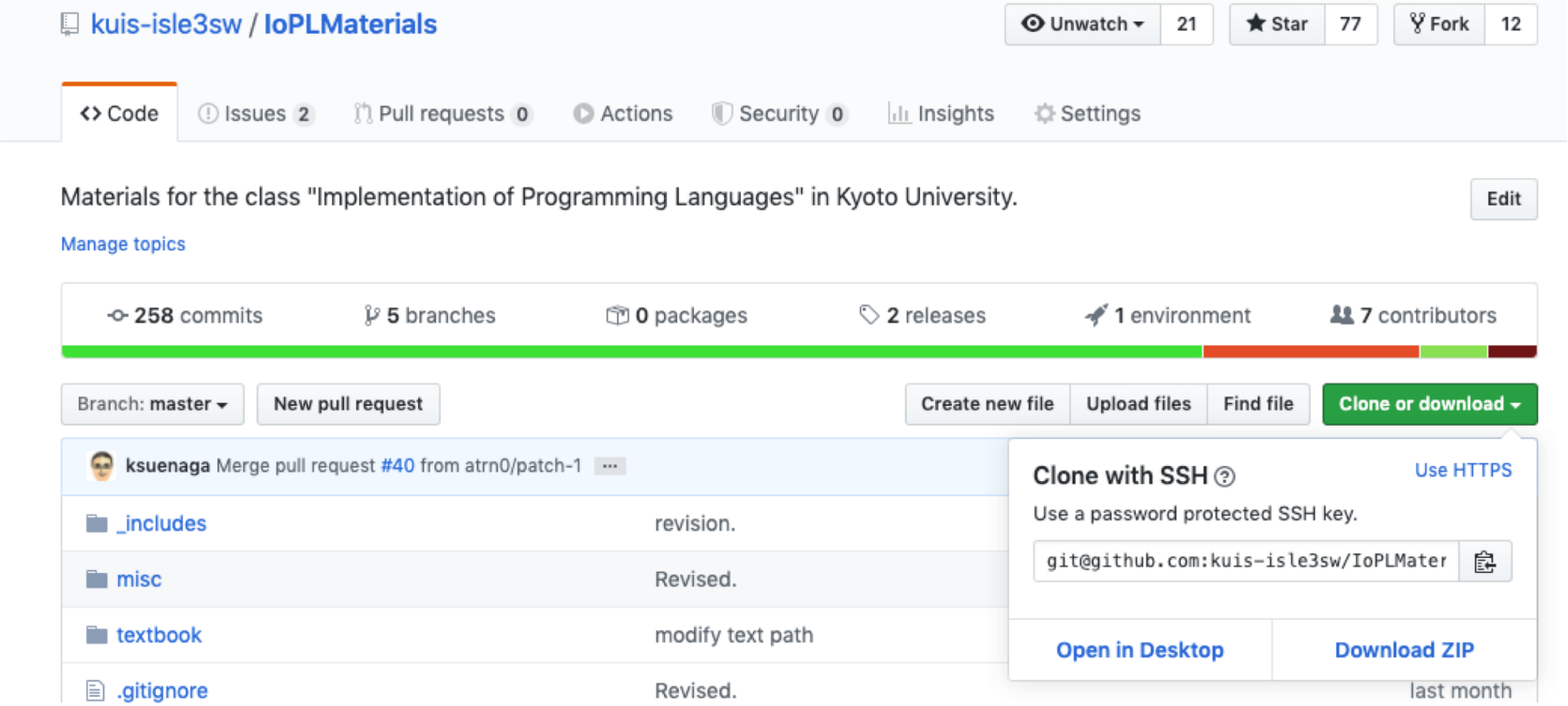
ソースコードのありか

- 講義リポジトリ中の textbook/interpreter の中
  - 講義リポジトリ: https://github.com/kuis-isle3sw/IoPLMaterials
  - 手元に clone して動かすのがおすすめ



ソースコードのありか

- 講義リポジトリ中の textbook/interpreter の中
  - 講義リポジトリ: https://github.com/kuis-isle3sw/IoPLMaterials
  - 手元に clone して動かすのがおすすめ



ソースコードのありか

- 講義リポジトリ中の textbook/interpreter の中
  - 講義リポジトリ: https://github.com/kuis-isle3sw/IoPLMaterials
  - 手元に clone して動かすのがおすすめ

```
(base) KoheinoMacBook-Pro:tmp ksuenaga$ git clone git@github.com:kuis-isle3sw/IoPLMaterials.git
Cloning into 'IoPLMaterials'...
remote: Enumerating objects: 145, done.
remote: Counting objects: 100% (145/145), done.
remote: Compressing objects: 100% (99/99), done.
remote: Total 1035 (delta 85), reused 86 (delta 45), pack-reused 890
Receiving objects: 100% (1035/1035), 11.78 MiB | 6.38 MiB/s, done.
Resolving deltas: 100% (580/580), done.
(base) KoheinoMacBook-Pro:tmp ksuenaga$ ls IoPLMaterials/
Gemfile         README.md       _config.yml     _includes       build.sh        misc            textbook
(base) KoheinoMacBook-Pro:tmp ksuenaga$
```

ビルドの仕方

- textbook/interpreter ディレクトリで dune exec miniml と実行
  - コンパイルが必要なファイルがコンパイルされてインタプリタが起動される
  - opam で dune をインストールしていることが必要

ソースコードの構成

- インタプリタ全体がいくつかのファイルに分割されている

```
(base) KoheinoMacBook-Pro:tmp ksuenaga$ ls IoPLMaterials/textbook/interpreter/src/
cui.ml          environment.ml  eval.ml         mySet.ml        parser.mly      typing.ml
dune            environment.mli lexer.mll       mySet.mli       syntax.ml
(base) KoheinoMacBook-Pro:tmp ksuenaga$
```

アウトライン

- MiniML1 インタプリタ
  - 文脈自由文法と BNF
  - MiniML1 インタプリタ
    - ソースコードとビルドの仕方
    - 各モジュールの説明
    - 字句解析と構文解析のためのツール
- MiniML2 インタプリタ: 変数定義の導入
- MiniML3 インタプリタ: 関数値の導入
- MiniML4 インタプリタ: 再帰関数の定義

困難は分割せよ

- 大きなものは分割しないと作れない
  - 家: 床と壁と窓から部屋を作り，部屋を組み合わせて作る
  - コンピュータ: CPU とメモリと外部記憶等を組み合わせて作る
  - 人体: 五臓六腑を組み合わせて作る
  - （今週のクイズにしておくので，各自大喜利をすること）
  - **ソフトウエア: モジュールを組み合わせて作る**

プログラミング言語におけるモジュール

- ソフトウェアの機能を分割したときの単位
  - C や OCaml においては: 値や関数の集まり
  - Java においては: クラス
  - C++ においては: クラスや値や関数の集まり

- モジュールを作る際には，別のモジュールの実装への依存を減らすことが一般には重要
  - 依存関係が大きいと: 一つのモジュールの実装を変更すると別のモジュールの実装を大量に書き換える必要

インターフェイス

- モジュールの挙動についてモジュール外部に公開する情報
  - 変数や関数の名前
  - 型
  - 事前条件や事後条件

- 実装そのものではなく，実装に関する一部の情報を公開して
  その情報には依存して良いことにすることで
  モジュールの実装間の依存を減らすことができる

OCaml におけるモジュール

- .mlファイルと.mliファイルで一つのモジュールを作る
  - .mlファイルにモジュールの実装（変数や関数の定義）を，
  - .mliファイルにモジュールのインターフェイス（型）を記述
  - ファイル名がモジュール名に対応
    - ahoge.(ml|mli) から，Ahoge というモジュールが作られる

- モジュール外からは .mli ファイルで宣言されている名前を宣言されている型としてしか使用できない
  - ahoge.mli ファイルで宣言されている x という名前は，外部からは Ahoge.x として参照

- 例: textbook/camltutorial/module に入っている自然数を表すモジュールの例

ここからやること

- textbook/interpreter ディレクトリに入っている MiniML1 インタプリタを構成するモジュールを説明

Syntax モジュール (syntax.ml)

- 抽象構文木を表す型の定義
  - id: 識別子の型
  - binOp: 二項演算の型
  - exp: 式を表す抽象構文木の型
  - program: プログラムを表す型
- 後で型推論の際に必要になる型も定義されている
  - tyvar: 型変数の型
  - ty: MiniML における型を表す型

Eval モジュール (eval.ml)

- 解釈部の定義
  - exval: 式の評価結果の値の型
  - dnval: 変数が束縛される値の型
  - apply_prim: 二項演算の定義
    - binOp 型の値 op と，exval 型の値 arg1, arg2 を受け取って，arg1 と arg2 に op を適用した結果を返す
  - eval_exp: 解釈部本体
    - 環境と呼ばれるデータ構造 env と式 e を受け取って e を env の下で評価した結果を返す

Environment モジュール (environment.(ml|mli))

- 変数が紐付けられている値を管理する**環境**と呼ばれるデータ構造
  - x が v に紐付けられていることを「x が v に**束縛されている**」と言う

- 定義されている値
  - ‘a environment: 変数の ‘a 型への束縛を表現する環境の型
  - empty: 空の環境
  - extend: 環境を新しい束縛で拡張する関数
  - lookup: 環境から変数 x が束縛されている値を取り出す関数
  - map: 変数の束縛先の値に一括して関数を適用する関数
  - fold_right: 環境から何らかの値を計算する関数

Cui モジュール (cui.ml)

- 標準入力から文字列を受け取り，字句解析・構文解析をして，評価して，結果を表示するというループを繰り返す
  - このループは read-eval-print loop (REPL) と呼ばれる

Parser モジュール

- 構文解析を担うモジュール
  - 文法を定義する parser.mly ファイルから menhir というツールを用いて parser.ml と parser.mli が自動生成される
  - 詳しくは後で

Lexer モジュール

- 字句解析を担うモジュール
  - トークン定義用の lexer.mll ファイルから ocamllex というツールを用いて lexer.ml が自動生成される
  - 詳しくは後で

アウトライン

- MiniML1 インタプリタ
  - 文脈自由文法と BNF
  - MiniML1 インタプリタ
    - ソースコードとビルドの仕方
    - 各モジュールの説明
    - 字句解析と構文解析のためのツール
- MiniML2 インタプリタ: 変数定義の導入
- MiniML3 インタプリタ: 関数値の導入
- MiniML4 インタプリタ: 再帰関数の定義

Menhir

- 構文解析を行うプログラムを生成するツール（**パーザジェネレータ**）
  - 入力: 文法定義ファイル（拡張子 .mly）
  - 出力: 構文解析を行う関数を持つモジュール（.ml と .mli）

- 世の中にはいろいろなパーザジェネレータがある
  - Yacc (C)
  - Bison (C)
  - ANTLR (Java)
  - Parsec (Haskell)

- いろいろな構文解析アルゴリズムもある
  - 講義後半で解説

文法定義ファイル parser.mly

parser.mly の宣言部

```
%token LPAREN RPAREN SEMISEMI
%token PLUS MULT LT
%token IF THEN ELSE TRUE FALSE
%token <int> INTV
%token <Syntax.id> ID

%start toplevel

%type <Syntax.program> toplevel
```

文法定義中で使用する
トークンの宣言

トークンに属性を持たせ
るときは属性の型を書く

開始記号の宣言

非終端記号に対応する構文木の
型を宣言（開始記号には必ず必要）

parser.mly の文法規則部

```
toplevel :
  e=Expr SEMISEMI { Exp e }

Expr :
  e=IfExpr { e }
| e=LTExpr { e }

LTExpr :
  l=PExpr LT r=PExpr { BinOp (Lt, l, r) }
| e=PExpr { e }

PExpr :
  l=PExpr PLUS r=MExpr { BinOp (Plus, l, r) }
| e=MExpr { e }

MExpr :
  l=MExpr MULT r=AExpr { BinOp (Mult, l, r) }
| e=AExpr { e } AExpr : i=INTV { ILit i }
| TRUE { BLit true }
| FALSE { BLit false }
| i=ID { Var i }
| LPAREN e=Expr RPAREN { e }

IfExpr : IF c=Expr THEN t=Expr ELSE e=Expr { IfExp (c, t, e) }
```

true ;; が構文解析される様子

教科書に書き込みながら説明
https://kuis-isle3sw.github.io/IoPLMaterials/textbook/chap03-3.html

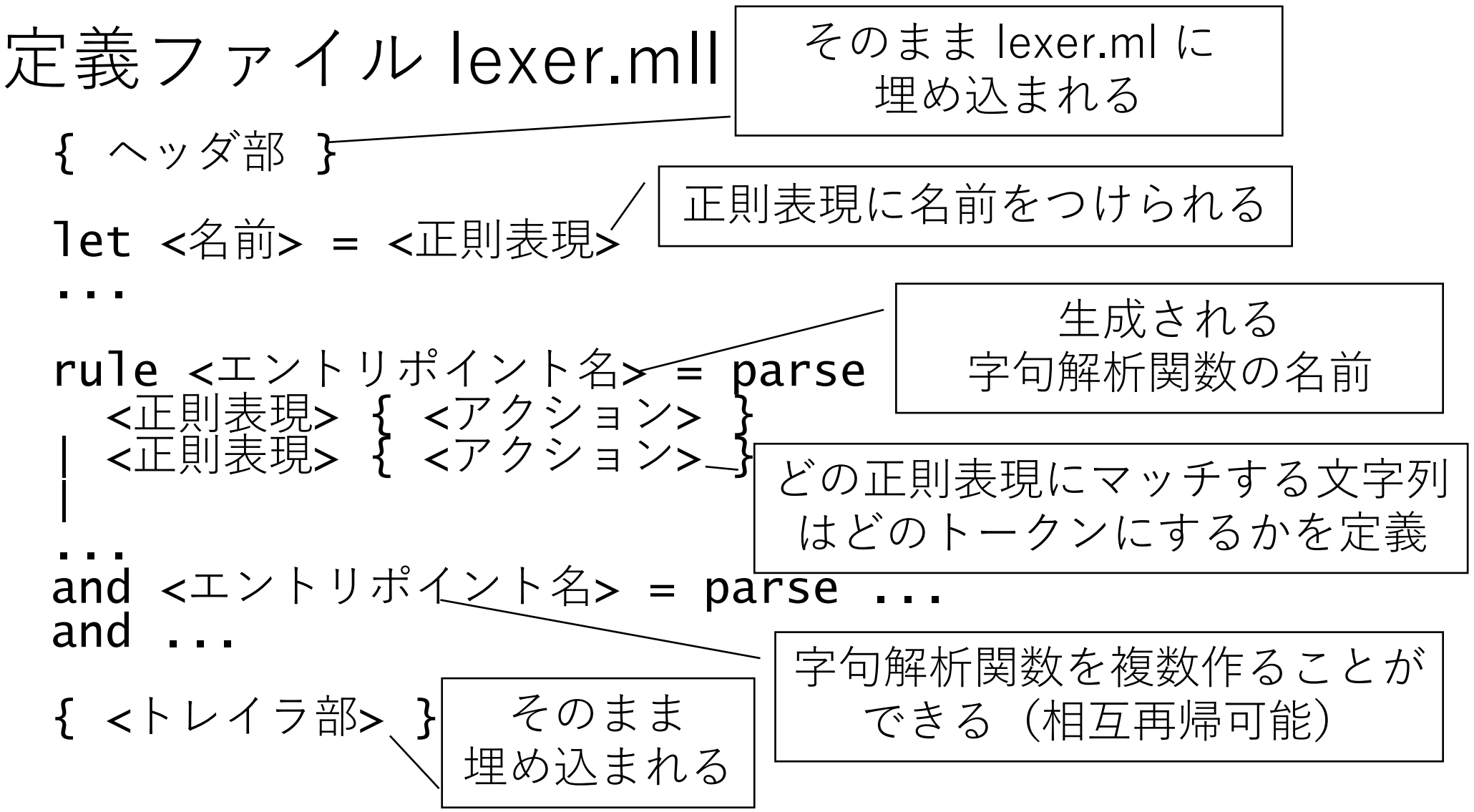
ocamllex

- 字句解析を行うプログラムを生成するツール
  - 入力: トークン定義ファイル（拡張子 .mll）
  - 出力: 字句解析を行う関数を持つモジュール（.ml）

- 世の中にはいろいろな字句解析器生成器がある
  - Lex (C)
  - Flex (C)

- 字句解析アルゴリズムは講義後半で解説

定義ファイル lexer.mll



lexer.mll

ソースコードを見ながら説明
https://kuis-isle3sw.github.io/IoPLMaterials/textbook/chap03-3.html

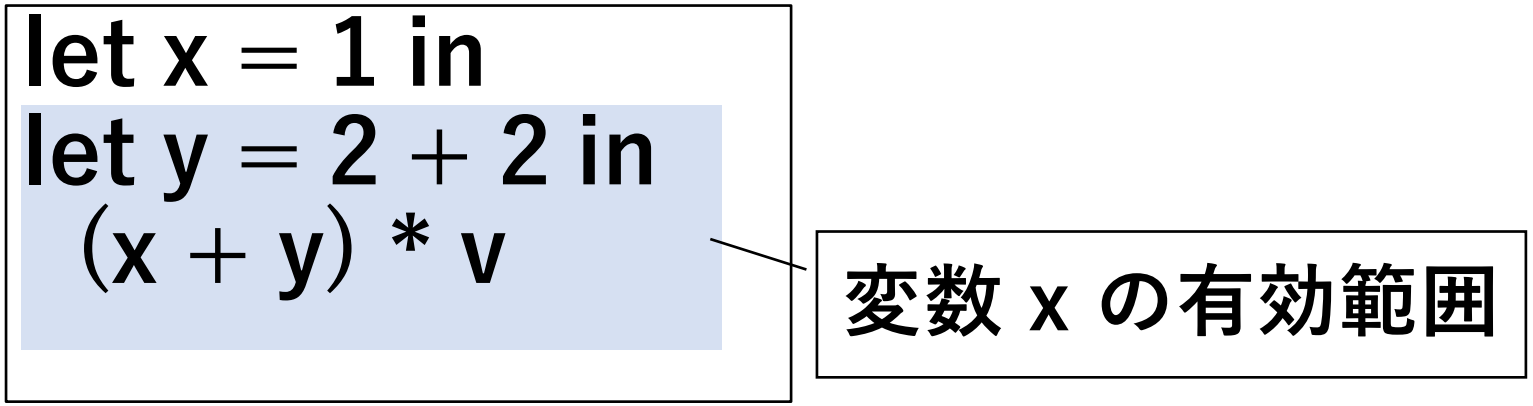
アウトライン

- MiniML1 インタプリタ
  - 文脈自由文法と BNF
  - MiniML1 インタプリタ
    - ソースコードとビルドの仕方
    - 各モジュールの説明
    - 字句解析と構文解析のためのツール
- MiniML2 インタプリタ: 変数定義の導入
- MiniML3 インタプリタ: 関数値の導入
- MiniML4 インタプリタ: 再帰関数の定義

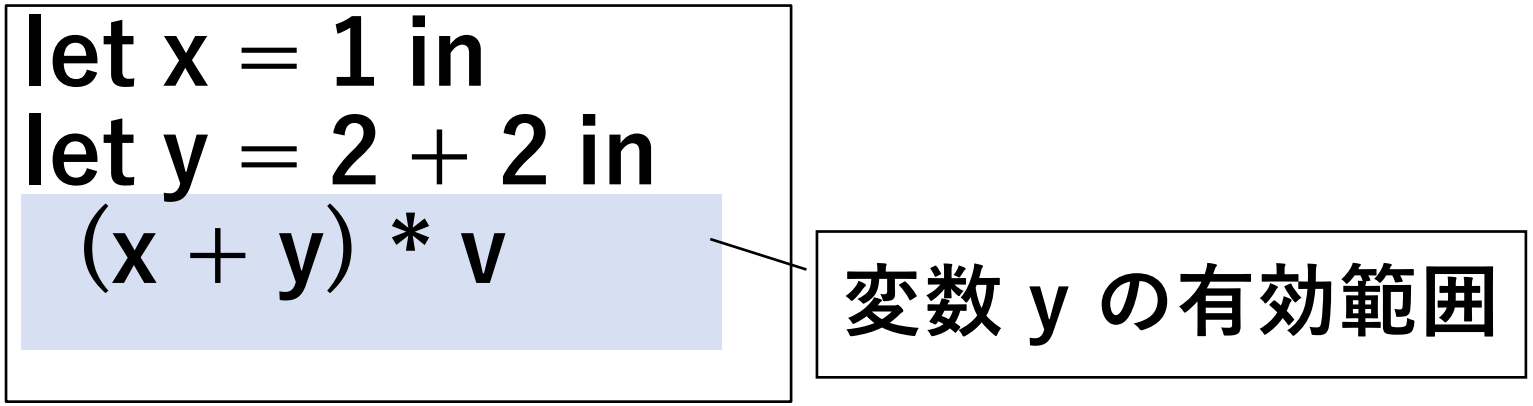
ここでやること

- let 式の追加
  - let x = e1 in e2
- let 文の追加
  - let x = e;;

変数の宣言や定義には有効範囲（**スコープ**）がある



変数の宣言や定義には有効範囲（**スコープ**）がある

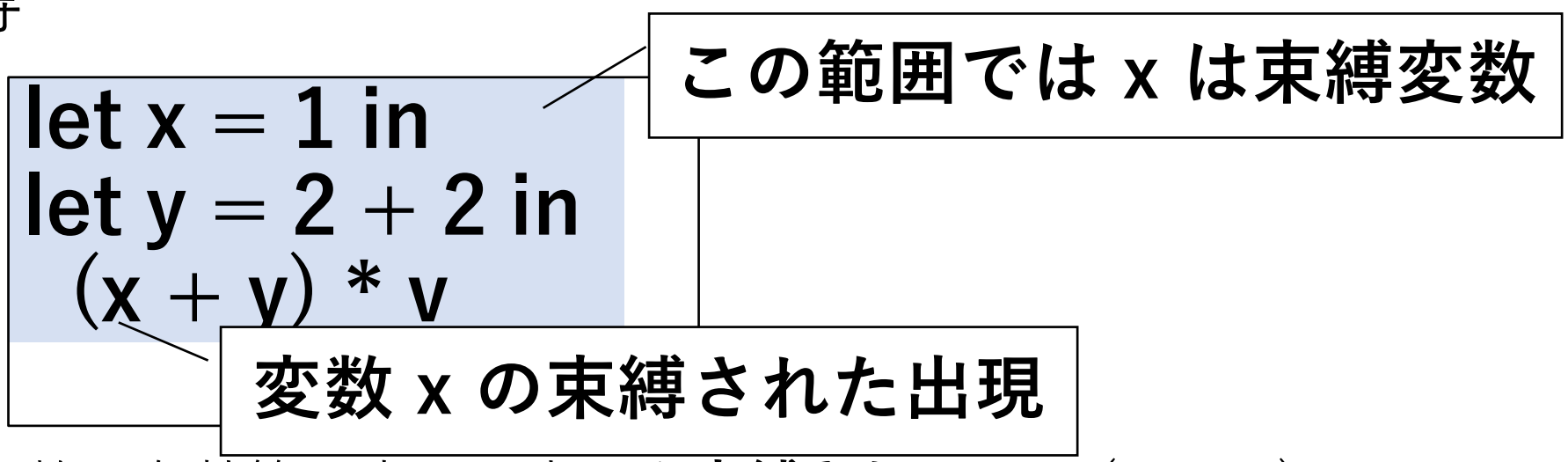


変数の宣言や定義には有効範囲（**スコープ**）がある

```
let x = 1 in
let y = 2 + 2 in
  (x + y) * v
```

変数 v の有効範囲は
プログラム全体

束縛



- 変数の有効範囲中での出現を**束縛されている** (bound) という
- 束縛された出現をしている変数自身を**束縛変数** (bound variable) という
- 束縛変数が使われている箇所を**変数の束縛された出現** (bound occurrence of a variable) という

自由 = 束縛されていないこと

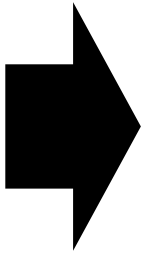
```
let x = 1 in
let y = 2 + 2 in
  (x + y) * v
```

この範囲では x は自由変数

- ある式において束縛されていない変数を**自由変数** (free variable) と呼ぶ

束縛変数を名前変えしても意味は変わらない

```
let x = 1 in
let y = 2 + 2 in
  (x + y) * v
```



```
let z = 1 in
let y = 2 + 2 in
  (z + y) * v
```

どっちも計算される値は 25

- 束縛変数は着目している範囲において局所的な名前
- 局所的な名前を一貫して新規な名前に変えても，
  計算される値は変わらない

静的束縛・動的束縛

- 変数のスコープや束縛がプログラムの字面から実行前に決まることを変数が**静的有効範囲** (static scope) や**静的束縛** (static binding) を持つという
- 実行時までスコープや有効範囲がわからないことを**動的有効範囲** (dynamic scope)，**動的束縛** (dynamic binding) という

MiniML2 での構文と実装の拡張

- 教科書をみながら
  https://kuis-isle3sw.github.io/IoPLMaterials/textbook/chap03-4.html

アウトライン

- MiniML1 インタプリタ
  - 文脈自由文法と BNF
  - MiniML1 インタプリタ
    - ソースコードとビルドの仕方
    - 各モジュールの説明
    - 字句解析と構文解析のためのツール
- MiniML2 インタプリタ: 変数定義の導入
- MiniML3 インタプリタ: 関数値の導入
- MiniML4 インタプリタ: 再帰関数の定義

ここでやること

- 関数抽象 (fun 式) の追加
  - fun x -> e
- 関数適用の追加
  - e1 e2

Syntax, Parser, Lexer の拡張

- 教科書をみながら
  https://kuis-isle3sw.github.io/IoPLMaterials/textbook/chap03-5.html

関数を表す値

- Eval モジュールの exval 型に関数を表す値を追加する必要

```
type exval =
    IntV of int
  | BoolV of bool
and dnval = exval
```

fun x -> e の評価結果として何を追加したらよい?

第一案: 仮引数の名前と本体の式を保持

- fun x -> e は仮引数 x を受け取った値に束縛し，式 e を評価して返す関数
- したがって，仮引数名 x と，本体の式 e さえ保持すればよいように思える

```
type exval =
  ...
  | ProcV of id * exp (* 仮引数と関数本体の情報だけで良いだろうか? *)
and dnval = exval
```

これでよい?

実はダメ: 関数は自由変数を含みうる

```
let f =
  let x = 2 in
  let addx = fun y -> x + y in
  addx
in
f 4
```

addx は受け取った値に x を加えて返す関数

x の値は 2 なので addx は受け取った値に 2 を加えて返す関数

f は addx に束縛される

その f を x のスコープの **外で** 呼び出す

関数呼び出し時に x が環境にない!

```
let f =
  let x = 2 in
  let addx = fun y -> x + y in
  addx
in
f 4
```

この f 4 を評価するときに f は
Proc(y, *x+y を表す exp 型の値*) に
束縛されている

f 4 の結果を得るには
y を 4 に束縛して
x + y を評価

x が環境にないので
評価できない!!

解決策: 関数値が自由変数の値も保持

```
let f =
  let x = 2 in
  let addx = fun y -> x + y in
  addx
in
f 4
```

この関数を表す値が
変数名 y，関数本体式 x+y に加えて
自由変数 x が何に束縛されているかの
情報も保持すればよい

正しい関数値の表現:

```
type exval =
    IntV of int
  | BoolV of bool
  | ProcV of id * exp * dnval Environment.t
and dnval = exval
```

自由変数の情報が全部入っている
はずの関数作成時の環境を保持

- 仮引数名，関数本体式，自由変数の束縛の情報をひとまとめにした関数値を表すデータを**関数閉包**とか**クロージャ**(function closure) とか呼ぶ

Eval の拡張

- 教科書をみながら
  https://kuis-isle3sw.github.io/IoPLMaterials/textbook/chap03-5.html

アウトライン

- MiniML1 インタプリタ
  - 文脈自由文法と BNF
  - MiniML1 インタプリタ
    - ソースコードとビルドの仕方
    - 各モジュールの説明
    - 字句解析と構文解析のためのツール
- MiniML2 インタプリタ: 変数定義の導入
- MiniML3 インタプリタ: 関数値の導入
- MiniML4 インタプリタ: 再帰関数の定義

ここでやること

- 再帰関数定義(let rec 式，let rec 文) の追加
  - let rec f x = e1 in e2
  - let rec f x = e;; f

簡単のため，1引数の再帰関数定義に限定

- BNFの拡張

```
P ::= ...
    |  let rec <識別子> = fun <識別子> -> e ;;
e ::= ...
    | let rec <識別子> = fun <識別子> -> e in e
```

再帰関数を特別に扱う必要がある理由

```
let rec fact = fun n ->
  if n = 0 then 1
  else n * fact (n-1)
in
fact 5
```

ここを e と略記

再帰関数を特別に扱う必要がある理由

```
let rec fact = fun n ->
  e
in
fact 5
```

fact はクロージャ
Proc(“n”, e, initial_env) に束縛される

fact 5 の呼び出し時には
initial_env∪{n→5} の下で
e を評価することになる

e 中では fact を使用しているが initial_env∪{n→5} 中では fact が束縛されていないのでエラー

どうしたらよいか

```
let rec fact = fun n ->
  e
in
fact 5
```

fact はクロージャ
Proc(“n”, e, initial_env) に束縛される

環境中で fact が自分自身を指すように環境を作ればよい

作りたいクロージャ

- こういう circular な構造をどのように作ればよい?

解決策: クロージャの環境を**環境への参照**に

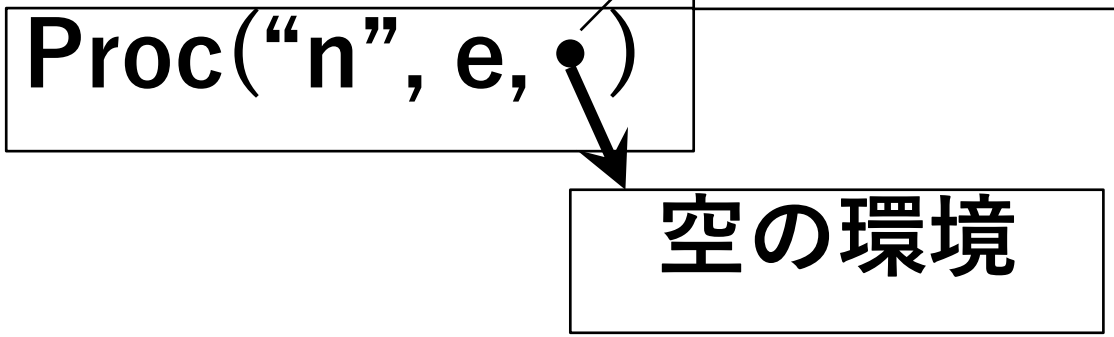
```
type exval =
    ...
     (* Changed! 関数閉包内の環境を参照型で保持するように変更 *)
  | ProcV of id * exp * dnval Environment.t ref
```

解決策: **バックパッチ**で循環構造を作る

```
| LetRecExp (id, para, exp1, exp2) ->
    (* ダミーの環境への参照を作る *)
    let dummyenv = ref Environment.empty in
    (* 関数閉包を作り,idをこの関数閉包に写像するように現在の環境envを拡張 *)
    let newenv = Environment.extend id (ProcV (para, exp1, dummyenv)) env in
    (* ダミーの環境への参照に,拡張された環境を破壊的代入してバックパッチ *)
        dummyenv := newenv;
        eval_exp newenv exp2
```

まずダミーの環境で
クロージャを作る

この参照が dummyenv
という名前



空の環境

解決策: **バックパッチ**で循環構造を作る

```
| Let                                                          ->
    (*
    le                                                    empty in
    (* 関数閉包を作り,idをこの関数閉包に写像するように現在の環境envを拡張 *)
    let newenv = Environment.extend id (ProcV (para, exp1, dummyenv)) env in
    (* ダミーの環境への参照に,拡張された環境を破壊的代入してバックパッチ *)
        dummyenv := newenv;
```

作ったクロージャを含む
新しい環境を作る

この環境が newenv
という名前

この参照が dummyenv
という名前

initial_env ∪ {fact → Proc(“n”, e, ●)　　}

空の環境

解決策: **バックパッチ**で循環構造を作る

```
| LetRecExp (id, para, exp1, exp2) ->
    (* ダミーの環境への参照を作る *)
    let dummyenv = ref Environment.empty in
    (* 関数閉包を作り,idをこの関数閉包に拡張 *)
    let newenv = Environment.extend id (Closure(para, exp1, dummyenv)) env in
    (* ダミーの環境への参照に,拡張された環境を破壊的代入してバックパッチ *)
        dummyenv := newenv;
```

dummyenv が newenv を指すようにする

この環境が newenv という名前

この参照が dummyenv という名前

initial_env ∪ {fact → Proc(“n”, e, •)　　}

空の環境

解決策: **バックパッチ**で循環構造を作る

```
| LetRecExp (id, para, exp1, exp2) ->
    (* ダミーの環境への参照を作る *)
    let dummyenv = ref Environment.empty in
    (* 関数閉包を作り,idをこの関数閉包に写像するように現在の環境envを拡張 *)
    let newenv = Environment.extend id (ProcV (para, exp1, dummyenv)) env in
    (* ダミーの環境への参照に,拡張された環境を破壊的代入してバックパッチ *)
        dummyenv := newenv;
        eval_exp newenv exp2
```

newenv で exp2 を評価

initial_env ∪ {fact → **Proc(“n”, e,)** **}**

この環境が newenv という名前

空の環境

（多分動く）環境への参照を使わない解決策

- 再帰関数用のクロージャと普通の関数用のクロージャを区別

- exval に ProcV に加えて RecProcV を加える
  - type exval =
    …
    | ProcV of id * exp * dnval Environment.t
    | **RecProcV of id * id * exp * dnval Environment.t**
  - RecProcV(f, x, e, env) は x を受け取って e を評価して返す関数で,
    e 中で f という名前で自分自身が参照されうることを表す
- クロージャ作成時: 再帰関数の名前を RecProcV 中に保持する以外は
  非再帰の関数と同様
  - 例: RecProcV(fact, “n”, e, initial_env)
- 関数適用時: クロージャ中の環境 (env) と再帰関数名 (f) を取り出し,
  f の自分自身への束縛で env を拡張子 (env∪{f→RecProcV(f,x,e,env)})

（多分動く）環境への参照を使わない解決策

- 関数適用時:
  - クロージャ中の環境 (env) と再帰関数名 (f) を取り出す
  - f の自分自身への束縛と仮引数の実引数への束縛で env を拡張 (env∪{f→RecProcV(f,x,e,env), x→arg})
  - 関数本体 (e) を評価

  - 例 (RecProcV(fact, n, e, env) を 5 に適用するとき): env に {n→5, fact→RecProcV(fact,n,e,env)} を追加した環境 newenv で e を評価