2022 年度 3 回生前期学生実験 SW

レポート 2（インタプリタ作成実験）

作成者: 伊藤舜一郎 (学籍番号:1029-32-7548)

提出期限：7 月 1 日 24 時 00 分　再提出日: 2022 年 7 月 8 日

1 Exercise 3.2.1

大域環境として i, v, x の値のみが定義されているが，ii が 2，iii が 3，iv が 4 となるようにプログラムを変更して，動作を確かめる。

1.1 変更した箇所

cui.ml に以下の変更点を加えることで実装した。

ソースコード 1: cui.ml

```
let initial_env =
  Environment.extend "i" (IntV 1)
    (Environment.extend "ii" (IntV 2) (∗3.2.1∗)
      (Environment.extend "iii" (IntV 3)
        (Environment.extend "iv" (IntV 4)
          (Environment.extend "v" (IntV 5)
            (Environment.extend "x" (IntV 10) Environment.empty)))))
```

1.2 実行例

iv + iii * ii が正しく動作することを確認すると、以下のようになった。

```
# iv + iii * ii;;
val - = 10
```

このように ii が 2，iii が 3，iv が 4 となるようにプログラムを変更することができた。

2 Exercise 3.2.2(7/8 変更)

このインタプリタは文法にあわない入力を与えたり，束縛されていない変数を参照しようとすると，プログラムの実行が終了してしまう．このような入力を与えた場合，適宜メッセージを出力して，インタプリタプロンプトに戻るように改造する．

2.1 変更した箇所

cui.ml に以下の変更点を加えることで実装した。

ソースコード 2: cui.ml

```
let rec read_eval_print env =
  print_string "#␣";
  flush stdout;
  try(let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
  let (id, newenv, v) = eval_decl env decl in
  Printf.printf "val␣%s␣=␣" id;
  pp_val v;
  print_newline();
  read_eval_print newenv) (∗3.2.2 Eval で発生したエラーを try でキャッチする∗) with
  Eval.Error e −> print_string e;print_newline();read_eval_print env (∗3.2.2
      Eval でエラーが発生したら対応するエラーメッセージを出力してインタプリタプロンプトに戻るようにする。
      ∗)
```

```
11  |Parser.Error −> print_string "Parser.error";print_newline();read_eval_print env (∗3.2.2
        Parser でエラーが発生したら Parser.
        error と出力してインタプリタプロンプトに戻るようにする。∗)
12  |_ −> print_string "Eval、Parser 以外でのエラー";print_newline();read_eval_print env (∗3.2.2
        lexer などでエラーが発生したら Eval、Parser 以外でのエラーと出力してインタプリタプロンプトに戻るようにする。
        ∗)
```

read_eval_print の try 内で Eval.Error が起きた場合に、print_string ”Eval.error”でエラーメッセージを出力して、print_newline() で改行した後、read_eval_print env を実行することでインタプリタプロンプトに戻るようにしている。Parser.Error やその他のエラーが起きた場合でも同様の処理を行っている。

2.2 実行例

実際にこの実装が正しく動作することを確認する。

```
# 1 + true;;
Both arguments must be integer: +
# 1 + a;;
Variable not bound: a
# let y = x -> x;;
Parser.error
```

文法にあわない入力を与えたり，束縛されていない変数を参照しようとしてもプログラムが終了しないようにすることができた。

3 Exercise 3.2.3

論理値演算のための二項演算子 &&, || を追加する. そのために、+や*と同様な実装に加え、true || undef のように前半だけで評価値が決まる場合はそれを出力するような実装を行う。

3.1 変更した箇所

lexer.mll,syntax.ml,parser.mly,eval.ml に以下の変更点を加えることで実装した。

ソースコード 3: lexer.mll

```
1 rule main = parse
2 ...
3 | "&&" { Parser.AND } (∗3.2.3 &&の実装∗)
4 | "||" { Parser.OR } (∗3.2.3 ||の実装∗)
```

ソースコード 4: syntax.ml

```
1 type binOp = Plus | Mult | Lt | And | Or
```

ソースコード 5: parser.mly

```
1 %token PLUS MULT LT AND OR
2
```

```
3  Expr :
4  ...
5    | e=ANDExpr{ e }
6    | e=OREpr{ e }
7
8  OREpr : (*3.2.3 ||の生成規則*)
9      l=OREpr OR r=ANDExpr { BinOp (Or, l, r) }
10   | e=ANDExpr { e }
11
12 ANDExpr : (*3.2.3 &&の生成規則*)
13     l=ANDExpr AND r=LTExpr { BinOp (And, l, r) }
14   | e=LTExpr { e }
```

生成規則は *,+,¡,&&,||の順に結合が強くなるように生成規則を実装した。

ソースコード 6: eval.ml

```
1  let rec apply_prim op arg1 arg2 = match op, arg1, arg2 with
2  ...
3    | And, BoolV b1, BoolV b2 -> BoolV (b1 && b2) (*3.2.3 &&の計算を行う*)
4    | And, _, _ -> err ("Both␣arguments␣must␣be␣boolean:␣&&")
5    | Or, BoolV b1, BoolV b2 -> BoolV (b1 || b2) (*3.2.3 ||の計算を行う*)
6    | Or, _, _ -> err ("Both␣arguments␣must␣be␣boolean:␣||")
7
8  let rec eval_exp env = function
9  ...
10   | BinOp (op, exp1, exp2) ->
11     let arg1 = eval_exp env exp1 in(
12       if(op = And && arg1 = BoolV false) then BoolV false (*3.2.3 前半だけで評価値が決まる
             場合はそれを出力する*)
13       else if(op = Or && arg1 = BoolV true) then BoolV true
14       else let arg2 = eval_exp env exp2 in
15       apply_prim op arg1 arg2
16     )
```

3.2 実行例

&&と||が正しく動作するか確認するために以下のような入力を与えると、次のようになった。

```
# true && false;;
val - = false
# true || false;;
val - = true
# false && a < 2;;
val - = false
# true || a < 2;;
val - = true
```

この結果から&&と||は正しく計算ができていて、かつ前半だけで評価値が決まる場合はそれを出力するように実装できていることがわかる。

4 Exercise 3.3.1

MiniML1 インタプリタを拡張して，MiniML2 インタプリタを作成し，テストする。

4.1 変更した箇所

lexer.mll,syntax.ml,parser.mly,eval.ml に以下の変更点を加えることで実装した。

ソースコード 7: lexer.mll

```
let reservedWords = [
...
  ("in", Parser.IN); (* New! *)
  ("let", Parser.LET); (* New! *)
]
...
| "<" { Parser.LT }
| "=" { Parser.EQ } (* New! *)
```

ソースコード 8: syntax.ml

```
type exp =
    ...
| LetExp of id * exp * exp (* <-- New! *)

type program =
  Exp of exp
| Decl of id * exp (* <-- New! *)
```

ソースコード 9: parser.mly

```
%token LET IN EQ (* <-- New! *)
toplevel :
      e=Expr SEMISEMI { Exp e }
    | LET x=ID EQ e=Expr SEMISEMI { Decl (x, e) } (* <-- New! *)

Expr :
      e=IfExpr { e }
    | e=LetExpr { e } (* <-- New! *)
    | e=LTExpr { e }

LetExpr :
      LET x=ID EQ e1=Expr IN e2=Expr { LetExp (x, e1, e2) } (* <-- New! *)
```

ソースコード 10: eval.ml

```
let rec eval_exp env = function
    ...
    | LetExp (id, exp1, exp2) ->
      (* 現在の環境で exp1 を評価 *)
      let value = eval_exp env exp1 in
      (* exp1 の評価結果を id の値として環境に追加して exp2 を評価 *)
      eval_exp (Environment.extend id value env) exp2

let eval_decl env = function
      Exp e -> let v = eval_exp env e in ("-", env, v)
    | Decl (id, e) ->
        let v = eval_exp env e in (id, Environment.extend id v env, v)
```

4.2 実行例

MiniML2 インタプリタが正しく動作するか確認するために以下のような入力を与えると、次のようになった。

```
#let x = 1 in let y = 2 + 2 in x + y * v;;
val - = 21
```

このことから let 式を用いたプログラムの実行ができていることがわかる。

5 Exercise 3.3.2

let 宣言の列を一度に入力する機能を実装し、テストする。

5.1 変更した箇所

lexer.mll,syntax.ml,parser.mly,eval.ml に以下の変更点を加えることで実装した。

ソースコード 11: lexer.mll

```
1 let reservedWords = [
2 ...
3   ("in", Parser.IN); (* New! *)
4   ("let", Parser.LET); (* New! *)
5 ]
```

ソースコード 12: syntax.ml

```
1 type program =
2 ...
3   | MultDecl of (id*exp) list
```

ソースコード 13: parser.mly

```
1 %token LET IN EQ (* <-- New! *)
2 toplevel :
3 ...
4   | LET x=ID EQ e1=Expr p2=LetMultExpr SEMISEMI { MultDecl ((x, e1) :: p2) }
5
6 LetMultExpr :
7       LET x=ID EQ e1=Expr p2=LetMultExpr { (x, e1) :: p2}
8   | LET x=ID EQ e=Expr { [(x, e)] }
```

ソースコード 14: eval.ml

```
1 let rec eval_decl env = function
2 ...
3   | MultDecl (p) ->
4     (match p with
5       [(x,e)] -> eval_decl env (Decl (x,e))
6       |((x,e1)::p2) -> let v = eval_exp env e1 in eval_decl (Environment.extend x v env) (
            MultDecl (p2))
7     )
```

let 宣言の列を受け取った場合、その let 宣言が 1 つのみだった場合 eval_decl env (Decl (x,e)) とすることで 3.3.1 と同様の処理ができるようにし、宣言が複数あった場合、先頭の宣言について環境に宣言した内容を保存して以降の宣言も再帰的に処理するように実装した。

5.2 実行例

let 宣言の列を一度に入力する機能が正しく動作するか確認するために以下のような入力を与えると、次のようになった。

```
# let x = 1
let y = x + 1;;
val y = 2
# x;;
val - = 1
```

教科書の例のように x の値を実行した段階で出力することはできなかったが、x;; で x の値が宣言通りになっていることが確認できたのでこの実装は正しく動作することがわかる。

6 Exercise 3.4.1

MiniML3 インタプリタを作成し，テストする。

6.1 変更した箇所

lexer.mll,syntax.ml,parser.mly,eval.ml に以下の変更点を加えることで実装した。

ソースコード 15: lexer.mll

```
let reservedWords = [
    ...
  ("fun", Parser.FUN);
    ...
]
...
| "=" { Parser.EQ }
| "->" { Parser.RARROW } (* New! *)
```

ソースコード 16: syntax.ml

```
type exp =
    ...
  | FunExp of id * exp (* New! *)
  | AppExp of exp * exp (* New! *)
```

ソースコード 17: parser.mly

```
%token RARROW FUN (* New! *)

Expr :
    ...
    | e=FunExpr { e }

FunExpr: (*3.4.1 fun 式の生成規則*)
      FUN e1=ID RARROW e2=Expr { FunExp (e1, e2) }

MExpr :
      e1=MExpr MULT e2=AppExpr { BinOp (Mult, e1, e2) }
  | e=AppExpr { e } (* New! *)

(* New! *)
```

```
15 AppExpr :
16     e1=AppExpr e2=AExpr { AppExp (e1, e2) }
17   | e=AExpr { e }
```

ソースコード 18: eval.ml

```
1  type exval =
2      IntV of int
3    | BoolV of bool
4    | ProcV of id * exp * dnval Environment.t (* New! クロージャが作成された時点の環境をデータ
         構造に含めている. *)
5  and dnval = exval
6
7  let rec eval_exp env = function
8    ...
9    (* 関数定義式: 現在の環境 env をクロージャ内に保存 *)
10   | FunExp (id, exp) -> ProcV (id, exp, env)
11   (* 関数適用式 *)
12   | AppExp (exp1, exp2) ->
13       (* 関数 exp1 を現在の環境で評価 *)
14       let funval = eval_exp env exp1 in
15       (* 実引数 exp2 を現在の環境で評価 *)
16       let arg = eval_exp env exp2 in
17       (* 関数 exp1 の評価結果をパターンマッチで取り出す *)
18       (match funval with
19           ProcV (id, body, env') -> (* 評価結果が実際にクロージャであれば *)
20 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣(*クロージャ内の環境を取り出して仮引数に対する束縛で拡張*)
21 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣let newenv = Environment.extend id arg !env' in
22                   eval_exp newenv body
23         | _ ->
24           (* 評価結果がクロージャでなければ,実行時型エラー *)
25           err ("Non-function value is applied"))
```

6.2 実行例

MiniML3 インタプリタが正しく動作するか確認するために以下のような入力を与えると、次のようになった。

```
let f = fun x -> x + 2 in f 5;;
val - = 7
```

上記の結果から fun 式を用いた入力が正しく動作することがわかる。

7 Exercise 3.4.4

階乗を計算するプログラムを書く。

7.1 作成したプログラム

ソースコード 19: 作成したプログラム

```
1  let makemult = fun maker -> fun x ->
2                    if x < 1 then 1 else x * maker maker (x + -1) in
3  let kaijyou = fun x -> makemult makemult x in
4    kaijyou 7
```

このプログラムは times4 の 4 を加算するところを x の乗算を行うようにして、x が 1 より小さい場合に 0 でなく 1 を出力するように変更したものだ。上記の実行例では 7 の階乗を計算していて、実行すると 5040 が出力される。

8 Exercise 3.5.1

MiniML4 インタプリタを作成し，テストする。

8.1 変更した箇所

lexer.mll,syntax.ml,parser.mly,eval.ml に以下の変更点を加えることで実装した。

ソースコード 20: lexer.mll

```
let reservedWords = [
...
  ("rec",Parser.REC)
```

ソースコード 21: syntax.ml

```
type exp =
    ...
  | LetRecExp of id * id * exp * exp

type program =
    ...
  | RecDecl of id * id * exp
```

ソースコード 22: parser.mly

```
%token REC

toplevel :
...
  | LET REC x=ID EQ FUN y=ID RARROW e=Expr SEMISEMI { RecDecl (x, y, e) }

Expr :
    ...
  | e=LetRecExpr { e }

LetRecExpr :(*3.5.1 再帰的関数の生成規則*)
    LET REC x=ID EQ FUN y=ID RARROW e1=Expr IN e2=Expr { LetRecExp ( x, y, e1,
        e2) }
```

ソースコード 23: eval.ml

```
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

let rec eval_exp env = function
  ...
  | 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
```

8.2 実行例

MiniML4 インタプリタが正しく動作するか確認するために以下のような入力を与えると、次のようになった。

```
let rec fact = fun n -> if n ¡ 1 then 1 else n * fact (n + -1) in fact 5;;  val - = 120
```

上記の実行例から再帰的関数定義が実装できていることがわかる。

9 工夫した点、実験の感想など

自分はこのインタプリタの作成に当たり、わからないことがあったら過去の Slack の質問とそれに対する回答や TA の方からのアドバイスをもらったりするなどして進めた。実験の感想としては、星の数と難易度が釣り合っていない課題があるように自分には思え、結局実装することができなかった課題もあったため悔しく感じた。前期終了までにそれらの問題が解けたらいいと思う。