平成 30 年度 3 回生前期

計算機科学実験及演習 3

レポート 2

提出

2018/7/5

28 年度入学 1029288922 松島優太

3.2.1

main.ml に対し、initial_env として以下を定義する。

main.ml

```
let initial_env =
   Environment.extend "i" (IntV 1)
     (Environment.extend "v" (IntV 5)
(*  中略  *)
                    (Environment.extend "iv" (IntV 4) Environment.empty)))))
```

これは、ID ”i”に IntV 1 が代入されるよう予め定義して、環境を拡張する。

結果として

端末

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

3.2.2

main.ml において、その端末上でのインタフェースを担う関数 read_eval_print に対し、以下の変更を行なう。

まず try-with を用いてエラーがあれば捕捉。

main.ml

```
let rec read_eval_print env =
   print_string "# ";
   flush stdout;
   try
(*  中略  *)
```

with 内は、パターンマッチによりエラーに応じたメッセージを表示し、環境は拡張せず元の通り。

main.ml

```
with
      Error message -> (print_string message;
                                print_newline();
                                read_eval_print env)
   | Parser.Error -> (print_string "parser error";
                                  print_newline();
                                  read_eval_print env)
   | _ -> (print_string "error: other error";
              print_newline();
```

```
            read_eval_print env)
```

ここで、エラーError は eval.ml による評価エラー。

eval.ml

```
exception Error of string
let err s = raise (Error s)
```

例えば

eval.ml

```
let rec apply_prim op arg1 arg2 = match op, arg1, arg2 with
      Plus, IntV i1, IntV i2 -> IntV (i1 + i2)
  | Plus, _, _ -> err ("Both arguments must be integer: +")
```

これにより、false+3 を行なおうとすると eval.ml の apply_prim 関数において Error “Both arguments must be integer: +”が吐かれる。これが main.ml において捕捉されこの”Both arguments must be integer: +”をエラーメッセージとして表示しつつこれを環境に含まず対話が再開される。

また、Parser.Error は paser.mly によって投げられたエラーをキャッチしている。

3.2.3

記号として”&&”,”||”を定義。

lexer.mll

```
rule main = parse
(*  中略  *)
| "&&" { Parser.AND }
| "||" { Parser.OR }
```

文法規則として

parser.mly

```
OrExpr :
l=OrExpr OR r=AndExpr { BinOp (Or, l, r) } (*  左結合  *)
   | e=AndExpr { e }

AndExpr :
l=AndExpr AND r=LTExpr { BinOp (And, l, r) } (* &&のほうが||より強い  *)
   | e=LTExpr { e }
```

として定義。ここで、&&は||より強いため、&&される要素として||を許さないが、||さ

れる要素として&&を考える。また、||、&&は左結合であるため 1 はそれぞれ OrExpr、AndExpr として「a&&b&&c」を「(a&&b)&&c」と読む実装としている。

評価の際、型制約としては

eval.ml

```
let rec apply_prim op arg1 arg2 = match op, arg1, arg2 with
  | And, BoolV b1, BoolV b2 -> BoolV (b1 && b2)
  | And, _, _ -> err ("Both arguments must be boolean: &&")
  | Or, BoolV b1, BoolV b2 -> BoolV (b1 || b2)
  | Or, _, _ -> err ("Both arguments must be boolean: ||")
```

&&、||の両辺は BoolV で結果は BoolV である。

eval.ml

```
let rec eval_exp env = function
  | BinOp (op, exp1, exp2) ->
      let arg1 = eval_exp env exp1 in
      if (op=And && arg1=BoolV false) then BoolV false
      else if (op=Or && arg1=BoolV true) then BoolV true
      else
      let arg2 = eval_exp env exp2 in
      apply_prim op arg1 arg2
```

eval_exp はこうすることにより、&&の第一引数が false ならば第二引数の評価に関係なく false となる、短絡評価を実現している。||についても同様。

3.2.4

以下を追記。

lexer.mll

```
rule main = parse
(*  中略  *)
| "(*" { comments 0 lexbuf }
```

これにより、”(*”があった場合にルール comments 0 へと移行。

lexer.mll

```
and comments level = parse
| "*)" { if level = 0 then main lexbuf
          else comments (level-1) lexbuf
        }
| "(*" { comments (level+1) lexbuf }
```

```
| _  { comments level lexbuf }
| eof  { print_endline "comments are not closed";
          raise End_of_file
        }
```

comments は、”(*”がある度に level が+1 され、”*)”の度に level が-1 されて再帰的に読み出されるルールである。これによりコメント内にコメントを書くことが可能となる。
level が 0 時に”*)”が来ると、コメントを抜けなければならない。これが上記 2 行目で main ルールに移行している。

3.3.1

教科書の通り追記した。

3.4.1

教科書の通り追記した。

また、

eval.ml

```
(* pretty printing *)
let rec string_of_exval = function
      IntV i -> string_of_int i
  | BoolV b -> string_of_bool b
  | ProcV (p, e, t) -> "<function>"
```

により、関数を定義した際のメッセージを表示できる。例えば

端末

```
# let f = fun x -> x + 1;;
val f = <function>
```

テストは、高階関数を定義し、計算させることで行なった。

端末

```
# let three = fun f -> fun x -> f (f x x) (f x x);;
val three = <function>
# let plus = fun x -> fun y -> x + y;;
val plus = <function>
# three plus 5;;
val - = 20
```

3.5.1

教科書の通り追記した。

また、

parser.mly

```
LetRecExpr：
        LET REC id=ID EQ FUN para=ID RARROW e1=Expr IN e2=Expr { LetRecExp
(id, para, e1, e2) }
```

により文法規則を追記。

また、eval.ml で

eval.ml

```
type exval =
(*  中略  *)
    | ProcV of id * exp * dnval Environment.t ref
```

として、環境を ref としたため、

eval.ml

```
let rec eval_exp env = function
(*  中略  *)
    | FunExp (id, exp) -> ProcV (id, exp, ref env) (* env をクロージャー内に保存  *)
    | AppExp (exp1, exp2) ->
        let funval = eval_exp env exp1 in
        let arg = eval_exp env exp2 in
          (match funval with
              ProcV (id, body, env') ->
                    (*  クロージャー内の環境を取り出し仮引数に対する束縛で拡張  *)
                  let newenv = Environment.extend id arg !env' in
                    eval_exp newenv body
              | _ -> err ("Non-function value is applied"))
```

なるよう、上記 3 行目を ref env に、10 行目を!env'に変更する必要があった。