計算機科学実験及演習 3 ソフトウェア実験レポート 2

- 学籍番号: 1029-28-8969
- 名前: 渡邊綾仁

各課題の実装内容

以下、ソースコードについては、GitHub の表記にならい、先頭に-がついている行は削除した行、+がついている行は新たに追加、変更した行を表すこととする。

Exercise 3.2.1 [必修]

ML1 インタプリタのプログラムをコンパイル・実行し，インタプリタの動
作を確かめよ．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，動作を確かめよ．例えば，
iv + iii * ii
などを試してみよ．

main.ml の大域環境に新たに ii、iii、iv を追加した。

```
let initial_env =
   Environment.extend "i" (IntV 1)
     (Environment.extend "v" (IntV 5)
-       (Environment.extend "x" (IntV 10) Environment.empty))
+      (Environment.extend "x" (IntV 10)
+      (* Exercise 3.2.1 *)
+        (Environment.extend "ii" (IntV 2)
+          (Environment.extend "iii" (IntV 3)
+            (Environment.extend "iv" (IntV 4) Environment.empty)))))
 ```
 インタプリタに、以下の入力を与えて、動作が正しいことを検証した（注: Exercise 4.2.1 の実装後に入力したものを使用している)。
 ```
 # iv + iii * ii;;
val - : int = 10
# iii * x * x + ii * (x + iv + iii);;
val - : int = 334
 ```

 ## Exercise 3.2.2 [★★]
 `main.ml` において、例外を受け取った場合、例外処理を行った上で、再び入力を受け付けるように変更した。

 ```
+open Typing
```

```
-let rec read_eval_print env =
+let rec read_eval_print env tyenv =
   print_string "# ";
   flush stdout;
   (* Exercise 3.2.2 *)
   let err_process message =
     Printf.printf "Error: %s " message;
     print_newline();
-    read_eval_print env in
+    read_eval_print env tyenv in
   (try
     let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
+    let ty = ty_decl tyenv decl in
     let (id, newenv, v) = eval_decl env decl in
-    Printf.printf "val %s = " id;
+    Printf.printf "val %s : " id;
+    pp_ty ty;
+    print_string " = ";
     pp_val v;
     print_newline();
-    read_eval_print newenv
-    with _ -> err_process "Error")
+    read_eval_print newenv tyenv
+    with Eval.Error err_message -> err_process err_message
+        | Parsing.Parse_error -> err_process "Parsing error"
+        | Typing.Error err_message -> err_process err_message
+        | _ -> err_process "(´・ω・`)"
+    )
```

インタプリタに以下のような入力を与えてみたところ、インタプリタが終了することなく、入力を受けつけることが確認できた (注: 型推論の実装後にこの課題の実装を行なっている。)

```
# hoge;;
Error: variable not bound: hoge
# iii + true;;
Error: Argument must be of integer: +
```

Exercise 3.2.4 [★★]

lexer.mll のルールに新たに comment というルールを追加した。このルールでは、トークン (*が出てくるたびに nest_count の値を 1 増やし、*) が出てくるたびに nest_count の値を 1 減らすように設定してある。このように実装することによって、入れ子になったコメントについても正しくコメントとして認識して、読み飛ばしてくれるようになる。

```
 | "<" { Parser.LT }
+| "(*" { comment 0 lexbuf }
```

```
 | ['a'-'z'] ['a'-'z' '0'-'9' '_' '\'']*
     { let id = Lexing.lexeme lexbuf in
@@ -32,4 +33,9 @@ rule main = parse
      }
 | eof { exit 0 }

+(* Exercise 3.2.4 *)

+and comment nest_times = parse
+  "(*" {comment (nest_times+1) lexbuf}
+| "*)" {if nest_times>0 then (comment (nest_times-1) lexbuf) else (main lexbuf)}
+| _ {comment nest_times lexbuf}
```

Exercise 3.3.1 [必修]

eval.ml を以下のように編集した。

```
           | BoolV false -> eval_exp env exp3
           | _ -> err ("Test expression must be boolean: if"))
+  (* ML2 interpreter *)
+  | 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)
```

lexer.mll には新たにトークン let,in,=を追加した。

```
   ("true", Parser.TRUE);
+  (* ML2 interpreter *)
+  ("let", Parser.LET);
+  ("in", Parser.IN);
...
 | "<" { Parser.LT }
 | "(*" { comment 0 lexbuf }
+(* ML2 interpreter *)
+| "=" { Parser.EQ }
```

syntax.ml には新たに LetExp と、Decl を追加した。

```
+    (* ML2 interpreter *)
+  | LetExp of id * exp * exp
```

```
 type program =
     Exp of exp
+  | Decl of id * exp
```

parser.mly には、新たに追加されたトークンと、構文規則を追加した。

```
 %token IF THEN ELSE TRUE FALSE
+(* ML2 interpreter *)
+%token LET IN EQ
...
 toplevel :
     e=Expr SEMISEMI { Exp e }
+  | LET x=ID EQ e=Expr SEMISEMI { Decl (x, e) }

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

+(* ML2 interpreter "Let" expression *)
+LetExpr :
+    LET x=ID EQ e1=Expr IN e2=Expr { LetExp (x, e1, e2) }
+
```

これらの変更を加えた上で、実際に miniml 上でテストをして見たところ、以下のようになり正しく動いていることがわかった。

```
# let hanshin = 4 in let lotte = 33 in lotte * hanshin;;
val - : int = 132
```

Exercise 3.4.1 [必修]

まず、テキストに従って eval.ml に変更を加えた。

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

......

     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
+  (* ML3 interpreter *)
+  (* 現在の環境 env をクロージャ内に保存 *)
+  | FunExp (id, exp) -> ProcV (id, exp, env)
```

```
+  | AppExp (exp1, exp2) ->
+      let funval = eval_exp env exp1 in
+      let arg = eval_exp env exp2 in
+        (match funval with
+          ProcV (id, body, env') ->
+            (* クロージャ内の環境を取り出して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")
+        )
+
```

テキストの記述だけでは不足しているため、eval.ml の string_of_exval の部分に先ほど exval に追加した ProcV について、出力に関する記述を加えた。

```
 let rec string_of_exval = function
     IntV i -> string_of_int i
   | BoolV b -> string_of_bool b
+  | ProcV (_, _, _) -> "< (`･ ω･ ´) つ<fun> >"
```

また、テキストに沿って、lexer.mll に fun と->をトークンとして認識するように記述を加えた。

```
   (* ML2 interpreter *)
   ("let", Parser.LET);
   ("in", Parser.IN);
+  (* ML3 interpreter *)
+  ("fun", Parser.FUN);
 ]
 }

 ......

  | "(*" { comment 0 lexbuf }
 (* ML2 interpreter *)
 | "=" { Parser.EQ }
+(* ML3 interpreter *)
+| "->" { Parser.RARROW }

 | ['a'-'z'] ['a'-'z' '0'-'9' '_' '\'']*
     { let id = Lexing.lexeme lexbuf in
```

次に、parser.mly には、トークンと構文規則を以下のように追加した。

```
 %token IF THEN ELSE TRUE FALSE
 (* ML2 interpreter *)
 %token LET IN EQ
+(* ML3 interpreter *)
+%token RARROW FUN

......
```

```
   | e=LetExpr { e }
   | e=LTExpr { e }
+  | e=FunExpr { e }

......

-MExpr :
-    l=MExpr MULT r=AExpr { BinOp (Mult, l, r) }
+MExpr :
+    e1=MExpr MULT e2=AppExpr { BinOp (Mult, e1, e2) }
+  | e=AppExpr { e }
+
+(* ML3 interpreter *)
+AppExpr :
+    e1=AppExpr e2=AExpr { AppExp (e1, e2) }
   | e=AExpr { e }

+FunExpr :
+    FUN x=ID RARROW e=Expr { FunExp (x, e) }
+
 AExpr :
```

さらに、syntax.ml には FunExp と AppExp の定義を追加した。

```
     (* ML2 interpreter *)
   | LetExp of id * exp * exp
+    (* ML3 interpreter *)
+  | FunExp of id * exp
+  | AppExp of exp * exp
```

これらの変更を加えた上で、関数の動作を確かめた。 cube は与えられた値を 3 乗する関数、araiguma は 2 つの引数をとってその和を計算する関数である。

```
# let cube = fun x -> x * x * x;;
val cube = < (`･ ω･ ´) つ<fun> >
# cube 4;;
val - = 64
# let araiguma = fun arai -> fun omakase -> arai + omakase;;
val araiguma = < (`･ ω･ ´) つ<fun> >
# araiguma 4009 640;;
val - = 4649
```

これより、関数式が高階関数も含めて正しく動いていることがわかる。

Exercise 3.4.4 [★]

階乗を計算するプログラムは以下のように書ける。このコードでは timesx 6 と書いているので 6!を計算している。

```
# let makefact = fun maker -> fun x ->
if x < 1 then 1
else x * maker maker (x + -1) in
let timesx = fun x -> makefact makefact x in
timesx 6;;
val - = 720
```

Exercise 3.5.1 [必修]

eval.ml についての変更は ProcV が以下のように変わっている。

```
-  | ProcV of id * exp * dnval Environment.t
+  | ProcV of id * exp * dnval Environment.t ref
```

それに伴い、FunExp について記述を変更している。

```
     eval_exp (Environment.extend id value env) exp2
   (* ML3 interpreter *)
   (* 現在の環境 env をクロージャ内に保存 *)
-  | FunExp (id, exp) -> ProcV (id, exp, env)
+  | FunExp (id, exp) -> ProcV (id, exp, ref env)
   | AppExp (exp1, exp2) ->
       let funval = eval_exp env exp1 in
       let arg = eval_exp env exp2 in
         (match funval with
           ProcV (id, body, env') ->
             (* クロージャ内の環境を取り出して仮引数に対する束縛で拡張 *)
-            let newenv = Environment.extend id arg env' in
+            let newenv = Environment.extend id arg !env' in
               eval_exp newenv body
         | _ -> err ("Non-function value is applied")
         )
```

さらに、LetRecExp についてテキストを参考に記述を加えた後、eval_decl に RecDecl についても、ダミーの環境への参照を作り、環境を拡張した後バックパッチを行なった。

```
+  | 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
+      (* ダミーへの環境への参照に、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


 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)
+      let v = eval_exp env e in (id, Environment.extend id v env, v)
+  | RecDecl (id, para, e) ->
+      let dummy = ref Environment.empty in
+        let new_env = Environment.extend id (ProcV (para, e, dummy)) env in
+        dummy := new_env;
+        (id, new_env, ProcV(para, e, dummy))
```

lexer.mll には rec を新たにトークンとして認識するように追加した。

```
   ("fun", Parser.FUN);
+  (* ML4 interpreter *)
+  ("rec", Parser.REC);
```

parser.mly は、rec をトークンとして加え、再帰についての文法定義を加えた。

```
 %token RARROW FUN
+(* ML4 interpreter *)
+%token REC

......

 Expr :
     e=IfExpr { e }
+  | e=LetRecExpr { e }
   | e=LetExpr { e }

......

 LetExpr :
     LET x=ID EQ e1=Expr IN e2=Expr { LetExp (x, e1, e2) }

+(* ML4 interpreter "Let rec" expression *)
+LetRecExpr :
+    LET REC x1=ID EQ FUN x2=ID RARROW e1=Expr IN e2=Expr { LetRecExp (x1, x2, e1, e2) }
+
+OrExpr :
+    l=AndExpr OR r=AndExpr { BinOp (Or, l, r) }
+  | e=AndExpr { e }
+
+AndExpr :
+    l=LTExpr AND r=LTExpr { BinOp (And, l, r) }
+  | e=LTExpr { e }
```

```
+
```

最後に、syntax.ml に、LetRecExp についての定義と、RecDecl についての定義を記述した。

```
     (* ML3 interpreter *)
   | FunExp of id * exp
   | AppExp of exp * exp
+    (* ML4 interpreter *)
+  | LetRecExp of id * id * exp * exp

 type program =
     Exp of exp
   | Decl of id * exp
+  | RecDecl of id * id * exp
```

これらの実装を行なった後、let rec 宣言を実際に行った。 fact は階乗を計算する再帰関数であり、以下のプログラムでは 6!を計算している。

```
# let rec fact =
fun n -> if n < 1 then 1 else n * (fact (n + -1)) in fact 6;;
val - = 720
```

6! = 720 であるので正しくプログラムが動作していることが確認できる。

インタプリタの実装についての感想

バックパッチの考え方を理解することと実際に実装するのが一番手間取った。また、&&と||の短絡評価の実装が意外と難しかった。