

# 計算機科学実験 ソフトウェアレポート

2018 年 6 月 27 日

102928948 勝田 峻太郎

## 目 次

<b>3.2.1[]</b>	<b>3</b>
<b>3.2.2[**]</b>	<b>3</b>
<b>3.2.3[*]</b>	<b>3</b>
<b>3.2.4[**]</b>	<b>4</b>
<b>3.3.1[]</b>	<b>5</b>
<b>3.3.2[**]</b>	<b>5</b>
Parsing . . . . .	5
Evaluation . . . . .	5
<b>3.3.3[**]</b>	<b>6</b>
<b>3.3.4[**]</b>	<b>7</b>
構文解析 . . . . .	7
let 宣言の拡張 . . . . .	7
let 式の拡張 . . . . .	7
評価 . . . . .	8
let 宣言の評価 . . . . .	8
let 式の評価 . . . . .	9
<b>3.4.1</b>	<b>9</b>
<b>3.4.2[**]</b>	<b>9</b>
<b>3.4.3[*]</b>	<b>10</b>
fun 式 . . . . .	10
fun 宣言 . . . . .	10
<b>3.4.4[*]</b>	<b>11</b>
<b>3.4.5[*]</b>	<b>11</b>
構文解析 . . . . .	11
評価 . . . . .	11
<b>3.4.6[*]</b>	<b>12</b>
1 . . . . .	12

2 . . . . .	12
3 . . . . .	12
4 . . . . .	12
<b>3.5.1[]</b>	<b>13</b>
構文解析 . . . . .	13
評価 . . . . .	13
<b>3.5.2[**]</b>	<b>14</b>
<b>3.6.1[**]</b>	<b>14</b>
<b>3.6.2[*]</b>	<b>14</b>
<b>3.6.3[*]</b>	<b>14</b>
<b>3.6.4[***]</b>	<b>14</b>
<b>3.6.5[**]</b>	<b>14</b>

### 3.2.1[]

(略)

### 3.2.2[\*\*]

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, 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改造せよ.

関数 `read_eval_print` 内で, エラーが起きたときも, エラーを出力した後, 再び `read_eval_print` を呼び出せば良い.

このために, エラーの処理を, 以下のように変更し, 実装した.

**main.ml**

```
...
with
  e ->
    let msg = Printexc.to_string e in
    print_string ("there was an error: " ^ msg ^ "\n");
    read_eval_print env;;
```

### 3.2.3[\*]

論理値演算のための二項演算子 `&&`, `||` を追加せよ.

**lexer.mll**

まず, 論理演算子を `lexer` に追加した.

```
| "&&" { Parser.AND }
| "||" { Parser.OR }
```

**parser.mly**

構文木において, 演算子の結合力は, `|| && < + *` であり, 論理演算は (OCaml では左再帰だが,) 右再帰なので, 以下のように実装した.

```
Expr :
  ...
  | e=ORExpr { e } (* arithmetic / boolean expression *)
  ...

(* logical expressions *)
ORExpr : (* or *)
  l=ANDExpr OR r=ANDExpr { LogicOp (Or, l, r) }
```

```
| e=ANDExpr { e }
```

```
ANDExpr : (* and *)
```

```
  l=LTExpr AND r=ANDExpr { LogicOp (And, l, r) }
```

```
| e=LTExpr { e }
```

```
(* arithmetic expressions *)
```

```
LTExpr : (* less than expression *)
```

```
  l=PExpr LT r=PExpr { BinOp (Lt, l, r) }
```

```
| e=PExpr { e }
```

```
....
```

eval.ml

OCaml の論理演算は, 短絡評価をおこなうので, そのように実装した.

```
let rec eval_exp env = function
```

```
| LogicOp(op, e1, e2) ->
```

```
  ( match op with
```

```
    | And -> let arg1 = eval_exp env e1 in
```

```
      if arg1 = BoolV(false) then BoolV(false) else
```

```
        let arg2 = eval_exp env e2 in if (arg2 = BoolV(true)) || (arg2 = BoolV(false)) then arg2
```

```
        else err("non boolean values supplied: &&")
```

```
    | Or -> let arg1 = eval_exp env e1 in
```

```
      if arg1 = BoolV(true) then BoolV(true) else
```

```
        let arg2 = eval_exp env e2 in if (arg2 = BoolV(true)) || (arg2 = BoolV(false)) then arg2
```

```
        else err("non boolean values supplied: ||")
```

### 3.2.4[\*\*]

lexer.mll を改造し, (*\** と *\**) で囲まれたコメントを読み飛ばすようにせよ.

コメントを読み飛ばすように,lexer.mll 内に, コメントを受理し, 読み飛ばすルールを追加した. コメントの開始記号で,rule main から rule comment に移動する. comment ルールは, 引数 *i* を 1 つとり, コメント開始記号で *i* をインクリメントし, コメント終了記号で *i* をデクリメントする.

lexer.mll

```
rule main = parse
```

```
...
```

```
| "(" { comment 1 lexbuf }
```

```
...
```

```
and comment i = parse
```

```
| ")" { if i = 1 then main lexbuf else comment (i-1) lexbuf }
```

```
| "(" { comment (i+1) lexbuf }
```

```
| _ {comment i lexbuf}
```

### 3.3.1[]

ML2 インタプリタを作成し、テストせよ。

### 3.3.2[\*\*]

OCaml では、let 宣言の列を一度に入力することができる。この機能を実装せよ。

## Parsing

To parse multiple let declarations, I added the following to `parser.mly`. By this rule, multiple declarations, for example `let x1 = e1 let x2 = e2 let x3 = e3` is parsed `DeclList((x1, e1) :: (x2, e2) :: (x3, e3))`.

### parser.mly

```
toplevel :
    ...
    | LET x=ID EQ e1=Expr l2=DECLLISTBOTTOExpr { DeclList((x, e1):: l2) }
    ...

(* continuous declarations *)
DECLLISTBOTTOExpr :
    | LET x=ID EQ e=Expr l2=DECLLISTBOTTOExpr { (x, e) :: l2 }
    | LET x=ID EQ e=Expr SEMISEMI { (x, e) :: [] }
```

## Evaluation

Initially on `main.ml`, the function `eval_decl` in `eval.ml` were called to evaluate the expression. However, to evaluate multiple declarations, I changed this to `eval_decls`. When the expression contains multiple declarations, `eval_decls` calls `eval_decl` for each of the declarations in order.

When evaluating multiple declarations, evaluating the n-th declaration must be done in an environment containing the past (n-1) declarations. To do this, `loop` passes the expanded environment to the next `loop`.

### eval.ml

```
let rec 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)
  ...
  | _ -> err("eval_decl failed")

let rec eval_decls env = function
  Exp e -> eval_decl env (Exp e) :: []
  | Decl (id, e) -> eval_decl env (Decl(id, e)) :: []
```

```

| DeclList(lst) ->
  let rec loop env l result =
    (match l with
    | (x, e) :: rest ->
      let (_, new_env, _) as top_eval = eval_decl env (Decl(x, e)) in (* evaluate top *)
      loop new_env rest (top_eval :: result)
    | [] -> result) in
  loop env lst []
  ...
| _ -> err("eval_decls failed")

```

### 3.3.3[\*\*]

バッチインタプリタを作成せよ. 具体的には miniml コマンドの引数としてファイル名をとり, そのファイルに書かれたプログラムを評価し, 結果をディスプレイに出力するように変更せよ. また, コメントを無視するよう実装せよ.

まず, 関数\_において, 引数が2つある場合は, 関数 batch\_interpreter を呼び出す. batch\_interpreter は, ファイルの各行を読んで, 結果を出力した後, 通常のインタプリタを呼び出す.

main.ml

```

let read_file filename =
  let lines = ref [] in
  let chan = open_in filename in
  try
    while true; do
      lines := input_line chan :: !lines
    done; !lines
  with End_of_file ->
    close_in chan;
    List.rev !lines ;;

let rec batch_interpreter env l =
  match l with
  | top :: rest -> print_string top; print_newline();
    let decl = Parser.toplevel Lexer.main (Lexing.from_string top) in
    let (id, newenv, v) = eval_decl env decl in
    Printf.printf "val %s = " id;
    pp_val v;
    print_newline();
    batch_interpreter newenv rest
  | [] -> read_eval_print env

let _ =
  if Array.length Sys.argv = 1
  then read_eval_print initial_env;
  if Array.length Sys.argv = 2

```

```

then
  (print_string ("reading : " ^ Sys.argv.(1));
   print_newline();
   batch_interpreter initial_env ( read_file Sys.argv.(1) );)

```

### 3.3.4[\*\*]

and を使って変数を同時にふたつ以上宣言できるように let 式・宣言を拡張せよ.

#### 構文解析

lexer.mll に, and に関する規則を付け加えた.

lexer.mll

```

{
let reservedWords = [
  (* Keywords *)
  ...
  ("and", Parser.LETAND);
  ...
]
}

```

また, parser.mly で, 拡張した let 式・宣言から構文木を生成する.

let 宣言の拡張

parser.mly

```

toplevel :
  e=Expr SEMISEMI { Exp e } (* expressions *)
  ...
  | LET x=ID EQ e1=Expr LETAND l2=CLOSEDDECLBOTTOMExpr { ClosedDeclList(ClosedDecl(x, e1):: l2) }

  ...
  (* closed declarations *)
CLOSEDDECLBOTTOMExpr :
  | x=ID EQ e1=Expr LETAND l2=CLOSEDDECLBOTTOMExpr { ClosedDecl(x, e1):: l2 }
  | x=ID EQ e1=Expr SEMISEMI { ClosedDecl(x, e1)::[] }
  ...

```

let 式の拡張

parser.mly

```

LETFUNPARAExpr :
  | x=ID l=LETFUNPARAExpr { x :: l }
  | x=ID EQ { x :: [] }

...
(* let expression *)
LETEExpr :
  | LET e1=MULTILETEExpr IN e2=Expr { MultiLetExp(e1, e2) } (* simple value declarations *)

...
(* multiple declarations for let expression *)
MULTILETEExpr :
  | x=ID EQ e=Expr LETAND l=MULTILETEExpr { (x, e) :: l }
  | f=ID params=LETFUNPARAExpr e=Expr LETAND l=MULTILETEExpr { (f, FunExp(params, e)) :: l }
  | x=ID EQ e=Expr { (x, e) :: [] }
  | f=ID params=LETFUNPARAExpr e=Expr { (f, FunExp(params, e)) :: [] }

```

例えば, `let x = 10 and y = 100 in e;;` は, `MultiLetExp([(x, 10);(y,100)],e)` となる.

## 評価

### let 宣言の評価

まず, 同じ変数名が複数宣言された場合には, エラーを返さなければいけない. `multiple_closed_decl_sanity` で, 同じ変数名もものがないかチェックする.

また, 複数 `let` 宣言の評価においては, 各宣言の値は, すべて最初の環境において評価しなければならない. よって, `loop` 関数内で, 一定の評価用環境 (開始時の環境) と, 出力環境を引数とし, 評価用環境での評価の結果を出力環境に追加していく.

### eval.ml

```

let multiple_decls_sanity lst =
  let rec loop l defined =
    match l with
    | (id, _) :: rest -> if find defined id then false else loop rest (id :: defined)
    | [] -> true
  in loop lst []
...
let rec eval_decls env = function
  Exp e -> eval_decl env (Exp e) :: []
...
| ClosedDeclList(lst) ->
  if multiple_closed_decl_sanity lst then
    let rec loop const_env current_env l result =
      (match l with
       | top :: rest -> let (_, new_env, _) as top_eval = eval_closed_decl const_env current_env top
                        loop const_env new_env rest (top_eval :: result)
       | [] -> result) in

```



```

    loop env env lst []
  ...

```

## let 式の評価

let 式においても, 同様なエラーを, `multiple_decls_sanity` でチェックし, let 宣言の場合と同じように, 前半の let 宣言から後半の評価用の環境を生成し, 評価用の結果を返す.

### eval.ml

```

let multiple_decls_sanity lst =
  let rec loop l defined =
    match l with
    | (id, _) :: rest -> if find defined id then false else loop rest (id :: defined)
    | [] -> true
  in loop lst []
...
let rec eval_exp env = function
...
| MultiLetExp(decls, e) ->
  if multiple_decls_sanity decls then
    let rec make_env current_env d =
      (match d with
       | top :: rest -> let (id, e) = top in
         let v = eval_exp env e in
         let update_env = Environment.extend id v current_env in
         make_env update_env rest
       | [] -> current_env )
    in let eval_env = make_env env decls in
      eval_exp eval_env e
  else err("variable is bound several times")
...

```

## 3.4.1

ML3 インタプリタを作成し, 高階関数が正しく動作するかなどを含めてテストせよ.

## 3.4.2[\*\*]

OCaml での「(中置演算子)」記法をサポートし, プリミティブ演算を通常の関数と同様に扱えるようにせよ.

中置プリミティブ演算のために, `parser.mly` に, 以下のようなような表現を追加した. `BinExpr` における解析の結果は, それぞれ `fun a b -> a + b` などの構文解析結果である.

parser.mly

```
AppExpr : (* function application *)
```

```
  e1=AppExpr e2=AExpr { AppExp(e1, e2) }  
| e1=AppExpr e2=BinExpr { AppExp(e1, e2) }  
| e=BinExpr { e }  
| e=AExpr { e }
```

```
BinExpr : (* binary expression *)
```

```
  LPAREN PLUS RPAREN { FunExp(["a" ; "b"], BinOp (Plus,  Var "a", Var "b")) }  
| LPAREN MULT RPAREN { FunExp(["a" ; "b"], BinOp (Mult,  Var "a", Var "b")) }  
| LPAREN LT RPAREN   { FunExp(["a" ; "b"], BinOp (Lt,    Var "a", Var "b")) }  
| LPAREN AND RPAREN  { FunExp(["a" ; "b"], LogicOp (And,  Var "a", Var "b")) }  
| LPAREN OR RPAREN   { FunExp(["a" ; "b"], LogicOp (Or,   Var "a", Var "b")) }
```

### 3.4.3[\*]

OCaml の `fun x1 ... xn → ... let f x1 ... xn = ...` といった簡略記法をサポートせよ.

#### fun 式

parser.mly が `fun x y z -> e` を `FunExp([x;y;z], e)` と解釈するように変更した.

parser.mly

```
FUNExpr : (* store ids as list *)
```

```
  FUN params=FUNPARAExpr e=Expr { FunExp(params, e) }
```

```
FUNPARAExpr :
```

```
  | x=ID l=FUNPARAExpr { x :: l }  
  | x=ID RARROW { x :: [] }
```

#### eval.ml

関数閉包を作成するために, 新しい環境への ref を作成し, そこに現在の環境をは破壊的代入する. ListProcV は, 関数への

```
let rec eval_exp env = function
```

```
  | FunExp (params, exp) -> let dummyenv = ref Environment.empty in dummyenv := env;  
    ListProcV (params, exp, dummyenv) (* save reference to current environment "env" inside closure *)
```

#### fun 宣言

parser.mly が `let f x y z -> e` を `Decl(f, FunExp([x;y;z], e))` と解釈するように変更した.

parser.mly

```
toplevel :
    e=Expr SEMISEMI { Exp e } (* expressions *)
    ...
    | LET f=ID b=LETFUNExpr { Decl(f, b) } (* declaration *)
    ...

...
(* let function declarations *)
LETFUNExpr :
    | para=LETFUNPARAExpr e=Expr SEMISEMI { FunExp(para, e) }

LETFUNPARAExpr :
    | x=ID l=LETFUNPARAExpr { x :: l }
    | x=ID EQ { x :: [] }
```

eval.ml

上記の fun 式の評価結果を, 環境に加えるだけである.

### 3.4.4[\*]

write program

```
(* example *)
let makemult = fun maker -> fun x ->
if x < 1 then 0 else 4 + maker maker (x + -1) in let times4 = fun x -> makemult makemult x in times4 3;
let makefact = fun maker -> fun x ->
if x < 1 then 0 else maker maker (x + -1) in let fact4 = fun x -> makefact makefact x in fact4 3;;
```

### 3.4.5[\*]

インタプリタを改造し, fun の代わりに dfun を使った関数は動的束縛を行うようにせよ.

## 構文解析

fun と同じように parser.mly に dfun を追加したので, 省略する.

## 評価

eval.ml

dfun は動的束縛なので, 関数閉包は必要なく, 宣言時の環境を保持する必要がある. 関数閉包 ProcV から, 環境を除いたものを, DProcV とした.

DProcV への関数適用のときには、関数呼び出し時の環境で、評価を行う。

```
let rec eval_exp env = function
  ...
  | AppExp (exp1, exp2) -> (
    let funval = eval_exp env exp1 in
    let arg = eval_exp env exp2 in
    (match funval with
     ...
     | DProcV(id, body) -> let newenv = Environment.extend id arg env in
        eval_exp newenv body
     | e -> err ("Non function value is applied")))
  | DFunExp (id, exp) -> DProcV(id, exp)
```

### 3.4.6[\*]

1

```
let fact = fun n -> n + 1 in let fact = fun n -> if n < 1 then 1 else n * fact (n + -1) in fact 5;;
val - = 25
```

let fact = fun n -> if n < 1 then 1 else n \* fact (n + -1) は、let fact = fun n -> n + 1 を環境に加えた状態で評価されるので、2 つ目の fact の左辺は、if n < 1 then 1 else n \* n となる。fact 5 は、環境の最上位に、2 つ目の fact が乗った状態で評価され、5 \* 5 = 25 を得る。

2

```
let fact = dfun n -> n + 1 in let fact = fun n -> if n < 1 then 1 else n * fact (n + -1) in fact 5;;
val - = 25
```

let fact = dfun n -> n + 1 では、関数式は引数以外の変数を含まないの、1 と同じである。よって、結果も 1 と同じである。

3

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

let fact = dfun n -> if n < 1 then 1 else n \* fact (n + -1) では、評価するときの環境における fact を用いるので、fact 5 においては、2 つめに宣言した (自分自身の) fact を参照する。よって、fact 5 = 5 \* fact 4 = 5 \* 4 \* fact 3 = ... = 5 \* 4 \* ... 2 \* 1 = 120 と評価される。

4

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

3 と同じ理由で 120 と評価される.

### 3.5.1[]

#### 構文解析

##### parser.mly

再帰関数の let 式・宣言は, 以下を受理するように定義した.

- `let rec f = fun x -> e`
- `let rec f x = e`
- `let rec f = fun x -> e in f t`
- `let rec f x = e in f t`

それぞれ, 解析結果は以下のようになる.

- `RecDecl(f, x, e)`
- `LetRecExp(f, x, e, AppExp(f t))`

`toplevel :`

```
e=Expr SEMISEMI { Exp e } (* expressions *)
```

```
...
```

```
| LET REC f=ID EQ FUN para=ID RARROW e=Expr SEMISEMI { RecDecl(f, para, e) }
```

```
| LET REC f=ID para=ID EQ e=Expr SEMISEMI { RecDecl(f, para, e) } (* recursive declaration 2 *)
```

```
...
```

```
...
```

`LETRECEXpr :`

```
| LET REC f=ID EQ FUN para=ID RARROW e1=Expr IN e2=Expr { LetRecExp(f, para, e1, e2) }
```

```
| LET REC f=ID para=ID EQ e1=Expr IN e2=Expr { LetRecExp(f, para, e1 ,e2) }
```

#### 評価

##### eval.ml

再帰関数宣言の評価は, 空の新しい環境への ref を作成, 現在の環境を破壊的代入し, その ref を保持する.

また, 再帰関数への関数適用の際には, 環境の ref が指す中身を適用する値で拡張し, そのもとで関数式を評価する.

```
type exval =
```

```
| IntV of int
```

```
| BoolV of bool
```

```
| ProcV of id * exp * dnal Environment.t ref
```

```
| ListProcV of id list * exp * dnal Environment.t ref
```

```
| DProcV of id * exp
```

```
let rec eval_exp env = function
```

```
  Var x ->
```

```
    (try Environment.lookup x env with
```

```
      Environment.Not_bound -> err ("Variable not bound: " ^ x))
```

```

...
| LetRecExp (id, para, exp1, exp2) ->
  let dummyenv = ref Environment.empty in
  let newenv = Environment.extend id (ProcV(para, exp1, dummyenv)) env in
  dummyenv := newenv;
  eval_exp newenv exp2
...
| AppExp (exp1, exp2) -> (
  let funval = eval_exp env exp1 in
  let arg = eval_exp env exp2 in
  (match funval with
    (* recursive function *)
    | ProcV(id, body, env_ref) -> let eval_env = Environment.extend id arg !env_ref in
      eval_exp eval_env body
    ...
  )
...
let rec eval_decl env = function
...
| RecDecl(id, para, e) -> (
  let dummyenv = ref Environment.empty in
  let newenv = Environment.extend id (ProcV(para, e, dummyenv)) env in
  dummyenv := newenv;
  (id, newenv, ProcV(para, e, dummyenv))
)
| _ -> err("eval_decl failed")

```

### 3.5.2[\*\*]

### 3.6.1[\*\*]

### 3.6.2[\*]

### 3.6.3[\*]

### 3.6.4[\*\*\*]

### 3.6.5[\*\*]