

# 計算機科学実験IV

1029-28-9483 勝田 峻太郎

2018 年 10 月 9 日

## 目 次

課題 1 (拡張構文: 任意)	1
階乗計算 . . . . .	1
フィボナッチ数 . . . . .	2
課題 2 (フロントエンド: 必須)	2
字句解析器 . . . . .	2
構文解析器 . . . . .	3
課題 3 (インタプリタ・型推論: 任意)	4
loop, recur 式への対応 . . . . .	4
インタプリタ . . . . .	6
型推論器 . . . . .	7
課題 4 (recur 式の検査: 必須)	7
課題 5 (正規形への変換: 必須)	7
課題 6 (クロージャ変換: 必須)	7

## 課題 1 (拡張構文: 任意)

階乗計算を行う MiniML プログラムを, 再帰を用いず loop 構文と組を使って書きなさい. 同じく, フィボナッチ数を求めるプログラムを, loop 構文と組を使って書きなさい.

### 階乗計算

```
1  (* calculate factorial *)
2  let fact n =
3  loop v = (n, 1) in
4  if v.1 > 1 then
5  recur (v.1 - 1, v.2 * v.1)
6  else v.2
```

## フィボナッチ数

```
1  (* calculate the fibonacci number *)
2  let fib n =
3    loop v = (n, (1, 0)) in
4    if v.1 > 1 then
5      let tmp1 = v.2.1 in
6      let tmp2 = v.2.1 + v.2.2 in
7      recur (v.1 - 1, (tmp2, tmp1))
8    else v.2.1 + v.2.2
```

## 課題2 (フロントエンド: 必須)

MiniML の文法規則に従う MiniML プログラムを入力とし, 以下の `syntax.ml` により定義される抽象構文木を返す字句解析器・構文解析器を作成しなさい.

字句解析器・構文解析器ともに実験Ⅲの実装を参考にした.

### 字句解析器

課題にない実装として, コメントアウト機能を実装した.

`lexer.mll`

```
1  let reservedWords = [
2    (* Keywords *)
3    ("else", Parser.ELSE);
4    ("false", Parser.FALSE);
5    ("fun", Parser.FUN);
6    ("if", Parser.IF);
7    ("in", Parser.IN);
8    ("let", Parser.LET);
9    ("rec", Parser.REC);
10   ("then", Parser.THEN);
11   ("true", Parser.TRUE);
12   ("loop", Parser.LOOP);
13   ("recur", Parser.RECUR);
14 ]
15 }
16
17 rule main = parse
18   (* ignore spacing and newline characters *)
19   [' ' '\009' '\012' '\n']+ { main lexbuf }
20   ...
21   | "(" { comment 1 lexbuf }
22   ...
23   | "(" { Parser.LPAREN }
```

```

24 | ")" { Parser.RPAREN }
25 | ";;" { Parser.SEMISEMI }
26 | "+" { Parser.PLUS }
27 | "*" { Parser.MULT }
28 | "<" { Parser.LT }
29 | "=" { Parser.EQ }
30 | "->" { Parser.RARROW }
31 | "," { Parser.COMMA }
32 | "." { Parser.DOT }
33 ...
34 and comment i = parse
35   | "*" { if i = 1 then main lexbuf else comment (i-1) lexbuf }
36   | "(" { comment (i+1) lexbuf }
37   | _ {comment i lexbuf}

```

## 構文解析器

以下のようにコードを追加し, 構文解析器を作成した. LetRecExp は, let rec f = fun x -> e1 in e2 以外にも let rec f x = e1 in e2 の表現にも対応した.

parser.mly

```

1  open Syntax
2
3  %}
4
5  -%token LPAREN RPAREN SEMISEMI RARROW
6  +%token LPAREN RPAREN SEMISEMI RARROW COMMA DOT
7  %token PLUS MULT LT EQ
8  -%token IF THEN ELSE TRUE FALSE LET IN FUN REC
9  +%token IF THEN ELSE TRUE FALSE LET IN FUN REC LOOP RECUR
10
11  %token <int> INTV
12  %token <Syntax.id> ID
13  Expr :
14    | e=LetExpr    { e }
15    | e=LetRecExpr { e }
16    | e=LTE Expr   { e }
17  + | e=LoopExpr   { e }
18
19  LTE Expr :
20    e1=PE Expr LT e2=PE Expr { BinOp (Lt, e1, e2) }
21  MExpr :
22
23  AppExpr :
24    e1=AppExpr e2=AExpr { AppExp (e1, e2) }
25  + | RECUR e1=AExpr { RecurExp (e1) }
26    | e=AExpr { e }

```

```

27
28   AExpr :
29   AExpr :
30       | FALSE { BLit false }
31       | i=ID { Var i }
32       | LPAREN e=Expr RPAREN { e }
33 +   | LPAREN e1=Expr COMMA e2=Expr RPAREN { TupleExp(e1, e2) }
34 +   | e1=AExpr DOT i=INTV { ProjExp(e1, i) }
35
36   IfExpr :
37       IF e1=Expr THEN e2=Expr ELSE e3=Expr { IfExp (e1, e2, e3) }
38   FunExpr :
39       FUN i=ID RARROW e=Expr { FunExp (i, e) }
40
41   LetRecExpr :
42 -   LET REC i=ID EQ FUN p=ID RARROW e1=Expr IN e2=Expr
43 +   | LET REC i=ID EQ FUN p=ID RARROW e1=Expr IN e2=Expr
44 +   | LET REC i=ID p=ID EQ e1=Expr IN e2=Expr
45       { if i = p then
46           err "Name conflict"
47       else if i = "main" then
48           err "main must not be declared"
49       else
50           LetRecExp (i, p, e1, e2) }
51 +
52 +LoopExpr :
53 +   LOOP id=ID EQ e1=Expr IN e2=Expr { LoopExp (id, e1, e2) }

```

### 課題3 (インタプリタ・型推論: 任意)

実験3で作成した  $ML^4$  言語のインタプリタと型推論器を基に, MiniML 言語のインタプリタと型推論器を作成しなさい.

インタプリタ・型推論器ともに実験Ⅲを参考に作成した.

#### loop, recur 式への対応

loop 式, recur 式のインタプリタと型推論器を作成するのは困難であるため, これらはすべてインタプリタ・型推論器に通す前に let rec 式に変換した.

一般的に,

$$loop\ v = e_1\ in\ e_2$$

は, 新しい変数  $f$  を用いて,

$$let\ rec\ f\ v = e_{2[recur\ e \rightarrow f\ e]} in\ f\ e_1$$

と表現できる.

例えば,

```

loop v = (1, 0) in
  if v.1 < 101
  then recur (v.1 + 1, v.1 + v.2)
  else v.2;;

```

は,

```

let rec f = fun v ->
  if v.1 < 101 then f (v.1 + 1, v.1 + v.2)
  else v.2

```

に変換される.

この操作を構文解析後の段階で, インタプリタ・型推論器に入る前に実装した.

main.ml

```

5  (* create fresh variable *)
6  let fresh_loopvar =
7    let counter = ref 0 in
8    let body () =
9      let v = !counter in
10     counter := v + 1;
11     "f_" ^ string_of_int v
12   in body
13
14  (* replace loop expressions with letrec *)
15  let recprog_of_loop p =
16    (* replace recur e with f e *)
17    let recur_subst newf e =
18      let rec recur_subst_loop = function
19        | FunExp(id, e) -> FunExp(id, recur_subst_loop e)
20        | ProjExp(e, i) -> ProjExp(recur_subst_loop e, i)
21        | BinOp(op, e1, e2) -> BinOp(op, recur_subst_loop e1, recur_subst_loop e2)
22        | LetExp(id, e1, e2) -> LetExp(id, recur_subst_loop e1, recur_subst_loop e2)
23        | AppExp(e1, e2) -> AppExp(recur_subst_loop e1, recur_subst_loop e2)
24        | LetRecExp(i1, i2, e1, e2) -> LetRecExp(i1, i2, recur_subst_loop e1, recur_subst_loop e2)
25        | LoopExp(id, e1, e2) -> LoopExp(id, recur_subst_loop e1, recur_subst_loop e2)
26        | TupleExp(e1, e2) -> TupleExp(recur_subst_loop e1, recur_subst_loop e2)
27        | IfExp(cond, e1, e2) -> IfExp(recur_subst_loop cond, recur_subst_loop e1, recur_subst_loop e2)
28        | RecurExp(e) -> AppExp(newf, e)
29        | _ as e -> e in
30      recur_subst_loop e in
31    let rec recexp_of_loop = function
32      | LoopExp(v, e1, e2) ->
33        let new_funct: id = fresh_loopvar () in
34        let rece1 = recur_subst (Var new_funct) (recexp_of_loop e2) in
35        let rece2 = AppExp(Var new_funct, e1) in
36        LetRecExp(new_funct, v, rece1, rece2)
37      | _ as e -> e in
38    match p with

```

```

39   | Exp e -> Exp (recexp_of_loop e)
40
41 let rec read_eval_print env tyenv =
42   print_string "# ";
43   flush stdout;
44   try
45     let decl = Exp(Parser.toplevel Lexer.main (Lexing.from_channel stdin)) in
46     (* remove loop exp from program *)
47     let decl' = recprog_of_loop decl in
48     (match decl' with
49      | Exp e -> string_of_exp e |> print_endline);
50     let ty, new_tyenv = ty_decl tyenv decl' in
51     let (id, newenv, v) = eval_decl env decl' in
52     ...

```

## インタプリタ

インタプリタでは, 新しい表現である tuple と proj に対応した.

まず, 新しい tuple データ型を追加した.

eval.ml

```

1  type exval =
2    | IntV of int
3    | BoolV of bool
4    | TupleV of exval * exval
5    | ProcV of id * exp * dval Environment.t ref
6  and dval = exval

```

また, eval\_exp に対応する項目を追加した.

eval.ml

```

1  let rec eval_exp env = function
2    ...
3  | TupleExp(e1, e2) ->
4    let v1 = eval_exp env e1 in
5    let v2 = eval_exp env e2 in
6    TupleV(v1, v2)
7  | ProjExp(e, i) ->
8    (match eval_exp env e with
9     | TupleV(v1, v2) -> if i = 1 then v1
10      else if i = 2 then v2
11      else err "ProjExp: index not valid"
12     | _ -> err "error: projection of non-tuple")
13  | _ -> err "eval_exp: should not enter this match"

```

## 型推論器

型推論器にも, tuple と proj の型推論を追加した.

typing.ml

```
1 | TupleExp(e1, e2) ->
2   let tyarg1, tysubst1 = ty_exp tyenv e1 in
3   let tyarg2, tysubst2 = ty_exp tyenv e2 in
4   let main_subst = unify(eqls_of_subst tysubst1 @ eqls_of_subst tysubst2) in
5   let ty1 = subst_type main_subst tyarg1 in
6   let ty2 = subst_type main_subst tyarg2 in
7   (TyTuple(ty1, ty2), main_subst)
8 | ProjExp(e, i) ->
9   (let tyarg, tysubst = ty_exp tyenv e in
10    let t1 = TyVar(fresh_tyvar()) in
11    let t2 = TyVar(fresh_tyvar()) in
12    let main_subst = unify(eqls_of_subst tysubst @ [(tyarg, TyTuple(t1, t2))]) in
13    let ty1 = subst_type main_subst t1 in
14    let ty2 = subst_type main_subst t2 in
15    if i = 1 then (subst_type tysubst ty1, tysubst)
16    else if i = 2 then (subst_type tysubst ty2, tysubst)
17    else err "non valid projection target")
```

## 課題 4 (recur 式の検査: 必須)

syntax.ml 中の recur\_check 関数を完成させることにより, recur 式の検査を実装しなさい. parser.mly 中の呼び出している箇所を見ると分かるとおり, recur\_check 関数は unit 型の値を返す. 末尾位置ではないところに書かれた recur 式を発見したら, 即座に例外を投げコンパイル処理を中断すること.

## 課題 5 (正規形への変換: 必須)

言語 C への変換と, 正規形への変換を同時に行う, normal.ml 中の norm\_exp 関数を完成させよ. 関数は次に示す形で実装すること. 引数 f を適切に用いれば各場合分けで数行書くだけで完成する.

## 課題 6 (クロージャ変換: 必須)

closure.ml の convert 関数を完成させることにより, クロージャ変換を実装しなさい.