

計算機科学実験Ⅳ

1029-28-9483 勝田 峻太郎

2018 年 10 月 15 日

目 次

課題 1 (拡張構文: 任意)	1
階乗計算	2
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	5
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)	8
LetExp の実装	10
LetRecExp の実装	10
課題 6 (クロージャ変換: 必須)	11
LetRec 式の実装	12
関数クロージャに必要な変数の発見	13
関数クロージャの生成と関数本体式の変換	13
LetRec 式の変換	14
AppExp の変換	14
課題 7 (平滑化: 必須)	14
LetRec 式の取り出しと Fun 変数型への対応	15
返り値	15
課題 8 (仮想機械コード生成: 必須)	16

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=LTEExpr      { e }

```

```

17 + | e=LoopExpr { e }
18
19 LExpr :
20     e1=PExpr LT e2=PExpr { 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 式は let rec 式の syntax sugar とみなせるため, 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,

$$\text{loop } v = e_1 \text{ in } e_2$$

は, 新しい変数 f を用いて,

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

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.

recur_check の内部に, Syntax.exp と, その expression が末尾位置であるかを示す is_tail を引数に取る再帰関数を定義し, recur 式が正しい位置にあるかどうか確認する.

```

normal.ml

163 (* ==== recur 式が末尾位置にのみ書かれていることを検査 ==== *)
164 (* task4: S.exp -> unit *)
165 let rec recur_check e is_tail: unit =
166   let recur_err () = err "illegal usage of recur" in
167   S.(match e with
168     | RecurExp _ ->
169       if is_tail then ()
170       else recur_err ()
171     | LoopExp (x, e1, e2) ->
172       recur_check e1 false;
173       recur_check e2 true
174     | IfExp(e1, e2, e3) ->
175       recur_check e1 false;
176       recur_check e2 is_tail;
177       recur_check e3 is_tail
178     | LetExp(x, e1, e2) ->
179       recur_check e1 false;
180       recur_check e2 is_tail
181     | LetRecExp(f, x, e1, e2) ->
182       recur_check e1 false;
183       recur_check e2 is_tail
184     | FunExp(_, e) | ProjExp(e, _) ->
185       recur_check e false
186     | BinOp(_, e1, e2) | AppExp(e1, e2) | TupleExp(e1, e2) ->
187       recur_check e1 false;
188       recur_check e2 false
189     | _ -> () (* Var, ILit, BLit *)
190   )
191
192 (* ==== entry point ==== *)
193 let rec convert prog =
194   recur_check prog false;
195   normalize prog

```

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.

正規形への変換は, norm_exp 関数に実装した. norm_exp の引数 sigma は, LetExp や LetRecExp を変換する過程で, 新しく生成した Normal.id と Syntax.id の対応関係を保持するためのものである.

実装は以下の通り.

```

normal.ml

1 (* ==== 正規形への変換 ==== *)
2 let rec norm_exp (e: Syntax.exp) (f: cexp -> exp) (sigma: id Environment.t) =

```



```

3  match e with
4  | S.Var i ->
5      let maybe_fail i =
6          try f (ValExp (Var (Environment.lookup i sigma)))
7          with Environment.Not_bound ->
8              f (ValExp (Var ("_" ^ i ^ "temp")))
9      in maybe_fail i
10 | S.ILit i -> f (ValExp (IntV i))
11 | S.BLit b -> f (ValExp (IntV (int_of_bool b)))
12 | BinOp(op, e1, e2) ->
13     let x1 = fresh_id "bin" in
14     let x2 = fresh_id "bin" in
15     (norm_exp e1 (fun x ->
16         (norm_exp e2 (fun y ->
17             (LetExp(x2, y, LetExp(x1, x, f (BinOp(op, Var x1, Var x2)))))) sigma) sigma)
18 | IfExp(cond, e1, e2) ->
19     let x = fresh_id "if" in
20     norm_exp cond (fun condy ->
21         LetExp(x, condy, f (IfExp(Var x, norm_exp e1 (fun x -> CompExp x) sigma, norm_exp e2 (fun x ->
22 | LetExp(id, e1, e2) ->
23     let t1 = fresh_id "let" in
24     let sigma' = Environment.extend id t1 sigma in
25     norm_exp e1 (fun y1 ->
26         LetExp(t1, y1, norm_exp e2 f sigma'))) sigma
27 | FunExp(id, e) ->
28     let funf = fresh_id "funf" in
29     let funx = fresh_id "funx" in
30     (* let rec funf funx = e[id-> funx] in f *)
31     let sigma' = Environment.extend id funx sigma in
32     LetRecExp(funf, funx, norm_exp e (fun ce -> CompExp ce) sigma', f (ValExp (Var funf)))
33 | AppExp(e1, e2) ->
34     let t1 = fresh_id "app" in
35     let t2 = fresh_id "app" in
36     norm_exp e1 (fun y1 ->
37         (norm_exp e2 (fun y2 ->
38             LetExp(t1, y1, LetExp(t2, y2, f (AppExp(Var t1, Var t2)))) sigma) sigma)
39 | LetRecExp(func, id, e1, e2) ->
40     let recf = fresh_id "recf" in
41     let recx = fresh_id "recx" in
42     let sigma' = Environment.extend func recf (Environment.extend id recx sigma) in
43     LetRecExp(recf, recx, norm_exp e1 (fun ce -> CompExp ce) sigma', norm_exp e2 f sigma')
44 | LoopExp(id, e1, e2) ->
45     let loopvar = fresh_id "loopval" in
46     let loopinit = fresh_id "loopinit" in
47     let sigma' = Environment.extend id loopvar sigma in
48     norm_exp e1 (fun y1 ->
49         LetExp(loopinit, y1, LoopExp(loopvar, ValExp (Var loopinit), norm_exp e2 f sigma'))) sigma'
50 | RecurExp(e) ->
51     let t = fresh_id "recur" in

```

```

52     norm_exp e (fun y1 ->
53         LetExp(t, y1, RecurExp(Var t))) sigma
54 | TupleExp(e1, e2) ->
55     let t1 = fresh_id "tuple" in
56     let t2 = fresh_id "tuple" in
57     norm_exp e1 (fun y1 ->
58         norm_exp e2 (fun y2 ->
59             LetExp(t1, y1, LetExp(t2, y2, f (TupleExp(Var t1, Var t2))))) sigma) sigma
60 | ProjExp(e, i) ->
61     let t = fresh_id "proj" in
62     norm_exp e (fun y ->
63         LetExp(t, y, f (ProjExp(Var t, i)))) sigma

```

LetExp の実装

LetExp の正規化においては, $\text{let } x = e_1 \text{ in } e_2$ は, 以下のように表現できる. ただし, $[\cdot]$ は, 正規変換を表すものとする.

$$[\text{let } x = e_1 \text{ in } e_2] =$$

$$\text{let } t_1 = [e_1] \text{ in } [e_2]_{(x \rightarrow t_1)}$$

また, 実装は,

```

let t1 = fresh_id "let" in
let sigma' = Environment.extend id t1 sigma in
norm_exp e1 (fun y1 ->
    LetExp(t1, y1, norm_exp e2 f sigma')) sigma

```

となっている.

$\text{norm_exp } e2 \text{ f sigma'}$ が $e1$ のように LetExp の外部ではなく, 内部に記述されているのは, $e2$ は, x が $e1$ に束縛された環境のもとで評価される必要があるため, $e2$ の正規形はすべて Let 式の内部に存在する必要があるためである.

LetRecExp の実装

```

| FunExp(id, e) ->
    let funf = fresh_id "funf" in
    let funx = fresh_id "funx" in
    (* let rec funf funx = e[id-> funx] in f *)
    let sigma' = Environment.extend id funx sigma in
    LetRecExp(funf, funx, norm_exp e (fun ce -> CompExp ce) sigma', f (ValExp(Var funf)))
| ...
| LetRecExp(funct, id, e1, e2) ->
    let recf = fresh_id "recf" in
    let recx = fresh_id "recx" in

```

```
let sigma' = Environment.extend funct recf (Environment.extend id recx sigma) in
  LetRecExp(recf, recx, norm_exp e1 (fun ce -> CompExp ce) sigma, norm_exp e2 f sigma')
```

Fun 式 $\text{fun } x \rightarrow e$ は, LetRec 式に以下のように変換できるので, Fun 式の実装は LetRec 式と同じ様になっている.

```
let rec f x = e in f
```

LetRec 式 $\text{let rec } f \ x = e_1 \text{ in } e_2$ の正規形では,

- e_1 はそれまでの文脈とは独立に変換されるべきであり, 第 2 引数に f を用いず $\text{fun } ce \rightarrow \text{CompExp } ce$ を用いて, $\text{norm_exp } e_1 (\text{fun } ce \rightarrow \text{CompExp } ce) \text{ sigma}'$ と変換している. また e_1 内で x を含む可能性があるので, 既存の変換に, x と $\text{fun } x$ の対応を追加した sigma' を渡している.
- e_2 は, 内部で f を参照する可能性があるので, 既存の変換に, f と $\text{fun } f$ の対応を追加した sigma' を渡している.

また, e_1, e_2 ともに正規形のものが, LetRec 式の内部に存在する必要があるので, 上記のような実装になっている.

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.

クロージャ変換は, 以下のように実装した.

`closure.ml`

```
1  (* === conversion to closed normal form === *)
2  let rec closure_exp (e: N.exp) (f: cexp -> exp) (sigma: cexp Environment.t): exp =
3    match e with
4    | N.CompExp(N.ValExp(Var v)) ->
5      let may_fail v =
6        try
7          f (Environment.lookup v sigma)
8        with _ -> f (ValExp(Var ("_" ^ v)))
9      in may_fail v
10   | N.CompExp(N.ValExp(IntV i)) -> f (ValExp(IntV i))
11   | N.CompExp(N.BinOp(op, v1, v2)) -> f (BinOp(op, convert_val v1, convert_val v2))
12   | N.CompExp(N.AppExp(v1, v2)) ->
13     let new_app0 = fresh_id "closure_app" in
14     LetExp(new_app0, ProjExp(convert_val v1, 0),
15           f (AppExp(Var new_app0, [convert_val v1; convert_val v2])))
16   | N.CompExp(N.IfExp(v, e1, e2)) ->
17     closure_exp e1 (fun y1 ->
18       closure_exp e2 (fun y2 ->
19         CompExp(IfExp(convert_val v, f y1, f y2))) sigma) sigma
20   | N.CompExp(N.TupleExp (v1, v2)) -> f (TupleExp([convert_val v1; convert_val v2]))
21   | N.CompExp(N.ProjExp (v, i)) -> f (ProjExp(convert_val v, i))
22   | N.LetExp(id, ce1, e2) ->
23     closure_exp (CompExp ce1) (fun y1 ->
24       LetExp(convert_id id, y1, closure_exp e2 f sigma)) sigma
```

```

25 | N.LetRecExp(funcnt, id, e1, e2) ->
26   let recpointer = fresh_id ("b_" ^ funcnt) in
27   let funcnt_tuple_list = (Var recpointer:: get_out_of_scope_variables e1 [id]) in
28   let rec make_tuple_env l i env =
29     match l with
30     | Var hd:: tl ->
31       let env' = Environment.extend hd (ProjExp(convert_val (Var funcnt), i)) env in
32       make_tuple_env tl (i+1) env'
33     | [] -> env
34     | _ -> (match l with
35       | hd:: tl -> err ("unknown input in make_tuple_env" ^ string_of_closure(CompExp(ValExp(hd))))
36       | _ -> err "none valid match") in
37   let sigma' = make_tuple_env funcnt_tuple_list 0 Environment.empty in
38   let closure_contents = TupleExp(funcnt_tuple_list) in
39   let e2' = LetExp(convert_id funcnt, closure_contents, closure_exp e2 f sigma') in
40   LetRecExp(recpointer, [convert_id funcnt; convert_id id], closure_exp e1 f sigma', e2')
41 | N.LoopExp(id, ce1, e2) ->
42   closure_exp (CompExp ce1) (fun y1 ->
43     LoopExp(convert_id id, y1, closure_exp e2 f sigma)) sigma
44 | N.RecurExp(v) -> RecurExp(convert_val v)
45
46 (* entry point *)
47 let convert e = closure_exp e (fun ce -> CompExp ce) Environment.empty

```

LetRec 式の実装

```

1 | N.LetRecExp(funcnt, id, e1, e2) ->
2   let recpointer = fresh_id ("b_" ^ funcnt) in
3   let funcnt_tuple_list = (Var recpointer:: get_out_of_scope_variables e1 [id]) in
4   let rec make_tuple_env l i env =
5     match l with
6     | Var hd:: tl ->
7       let env' = Environment.extend hd (ProjExp(convert_val (Var funcnt), i)) env in
8       make_tuple_env tl (i+1) env'
9     | [] -> env
10    | _ -> (match l with
11      | hd:: tl -> err ("unknown input in make_tuple_env" ^ "\n" ^ string_of_closure(CompExp(ValExp
12      | _ -> err "none valid match") in
13    let sigma' = make_tuple_env funcnt_tuple_list 0 Environment.empty in
14    let closure_contents = TupleExp(funcnt_tuple_list) in
15    let e2' = LetExp(convert_id funcnt, closure_contents, closure_exp e2 f sigma') in
16    LetRecExp(recpointer, [convert_id funcnt; convert_id id], closure_exp e1 f sigma', e2')

```

LetRec 式においては、関数の id だけだったものが、関数クロージャ(関数名を転用) と関数ポインター (fresh_id ("b_" ^ funcnt) で生成¹) の 2 つが必要になる。また、関数本体式内の変数をクロージャに含めなければいけない。

¹funcnt は関数名とする。

関数クロージャに必要な変数の発見

関数クロージャに必要な参照範囲外の変数を発見するため、`get_out_of_scope_variables` 関数を定義した。この関数は、探索対象の `N.exp` を `e` として受け取り、すでにスコープに入っている変数名を `included` で受け取り、スコープ外の変数を `list` で返す。

```
1 let get_out_of_scope_variables (e: N.exp) (included: N.id list): value list =
2   let rec loop_e ex accum incl =
3     let rec loop_ce cex caccum incl =
4       N.(match cex with
5         | ValExp v | ProjExp(v, _) ->
6           (match (List.find_opt (fun x -> Var x = v) incl), v with
7             | Some x, _ -> caccum
8             | None, Var _ -> MySet.insert v caccum
9             | None, _ -> caccum)
10        | BinOp(_, v1, v2) | AppExp(v1, v2) | TupleExp(v1, v2) ->
11          MySet.union (loop_ce (ValExp v1) caccum incl) (loop_ce (ValExp v2) caccum incl)
12        | IfExp(v, e1, e2) ->
13          MySet.union (loop_ce (ValExp v) caccum incl) (MySet.union (loop_e e1 accum incl) (loop_e e2
14            in
15            N.(match ex with
16              | LetExp(i, cex, ex) ->
17                MySet.union (loop_ce cex accum incl) (loop_e ex accum (i::incl))
18              | LoopExp(i, cex, ex) ->
19                MySet.union (loop_ce cex accum incl) (loop_e ex accum incl)
20              | LetRecExp(i1, i2, e1, e2) ->
21                MySet.union (loop_e e1 accum (i2::incl)) (loop_e e2 accum (i2::incl))
22              | _ -> accum
23            )
24          in
25      List.map convert_val (MySet.to_list (loop_e e MySet.empty included))
```

関数クロージャの生成と関数本体式の変換

得られたスコープ外変数が、関数本体式内で正しくクロージャの要素として参照されるような変換を施すため、`make_tuple_env` 関数を用いて、変数と `ProjExp` を対応付ける環境を生成する。

例えば、`get_out_of_scope_variables` で得られたスコープ外変数の列が、

```
[Var "x"; Var "y"; Var "z"]
```

であった場合、`make_tuple_env` は、

環境 `sigma'`

```
[(Var "x", funct.1); (Var "y", funct.2); (Var "z", funct.3)]
```

を生成する。

これを用いて関数雨本体式を `closure_exp` で変換することで、関数本体式の変数はクロージャを参照するようになる。

LetRec 式の変換

最後に, LetRec 式は, `let rec f = <closure of f> in <closed normal form of e2>`

という形を取ればよい.

AppExp の変換

AppExp では, 関数のクロージャではなく, クロージャの第一要素である関数ポインタに対して関数適用することになる.

よって, 呼び出す関数のポインタを新しい変数に代入し, それに対して関数適用することとなる.

```
| N.CompExp(N.AppExp(v1, v2)) ->
  let new_app0 = fresh_id "closure_app" in
  LetExp(new_app0, ProjExp(convert_val v1, 0),
    f (AppExp(Var new_app0, [convert_val v1; convert_val v2])))
```

それを実装したのが, 以上のコードである.

課題 7 (平滑化: 必須)

flat.ml の flatten 関数を完成させることにより, 正規形コードを平滑化しなさい.

```
1  (* ==== フラット化 : 変数参照と関数参照の区別も同時に行う ==== *)
2  let convert_id (i: C.id): id = i
3  let convert_id_list (il: C.id list): id list = il
4
5  let get_flat_exp ex =
6    (* === helper functions === *)
7    let fun_list = ref (MySet.empty: C.id MySet.t) in
8    let append_fun v = fun_list := MySet.insert v !fun_list in
9    let search_fun v = MySet.member v !fun_list in
10   let decl_list = ref ([]: decl list) in
11   let append_decl d = decl_list := (d :: !decl_list) in
12   let convert_val (v: C.value): value =
13     match v with
14     | C.Var id -> if search_fun id
15       then Fun(id)
16       else Var(convert_id id)
17     | C.IntV i -> IntV(i) in
18   let convert_val_list (vl: C.value list): value list = List.map convert_val vl in
19   let rec flat_exp (e: C.exp) (f: cexp -> exp): exp =
20     match e with
21     | C.CompExp(C.ValExp v) -> f (ValExp(convert_val v))
22     | C.CompExp(C.BinOp(op, v1, v2)) ->
23       let v1' = convert_val v1 in
24       let v2' = convert_val v2 in
25       f (BinOp(op, v1', v2'))
```

```

26 | C.CompExp(C.AppExp(v, vl)) ->
27   let v' = convert_val v in
28   let vl' = convert_val_list vl in
29   f (AppExp(v', vl'))
30 | C.CompExp(C.IfExp(v, e1, e2)) ->
31   let v' = convert_val v in
32   flat_exp e1 (fun y1 ->
33     flat_exp e2 (fun y2 ->
34       f (IfExp(v', f y1, f y2))))
35 | C.CompExp(C.TupleExp(vl)) -> f (TupleExp(convert_val_list vl))
36 | C.CompExp(C.ProjExp(v, i)) -> f (ProjExp(convert_val v, i))
37 | C.LetExp(id, ce, e) ->
38   flat_exp (CompExp ce) (fun cy1 ->
39     LetExp(convert_id id, cy1, flat_exp e f))
40 | C.LetRecExp(func, idl, e1, e2) ->
41   append_fun (List.hd idl);
42   let letrec' = RecDecl(convert_id func, convert_id_list idl, flat_exp e1 (fun x -> CompExp x)) in
43   append_decl letrec';
44   flat_exp e2 f
45 | C.LoopExp(id, ce, e) ->
46   let id' = convert_id id in
47   flat_exp (CompExp ce) (fun cy1 ->
48     LoopExp(id', cy1, flat_exp e f))
49 | C.RecurExp(v) -> RecurExp(convert_val v)
50 in let converted = flat_exp ex (fun x -> CompExp x) in
51 (converted, !decl_list)
52
53 let flatten exp =
54   let toplevel_content, decl_list = get_flat_exp exp in
55   decl_list @ [RecDecl("_toplevel", [], toplevel_content)]

```

LetRec 式の取り出しと Fun 変数型への対応

以下のポインタを定義し, 変換をする.

- **fun_list** Fun 変数型へ対応するために, 関数へのポインタの集合を追加していく. `convert_val` では, `id` が `fun_list` に存在すれば Fun 変数型に変換され, そうでなければ Var 型に変換される.
- **decl_list** LetRec 式を入れ子から取り出し, 並べるため, `decl_list` に追加していく.

LetRec 式の平滑化処理は次のように行う.

1. 関数ポインタの `id` を `fun_list` に追加する.
2. 平滑化した LetRec 関数を取り出し, `decl_list` に追加する.
3. 続きの `exp` を `flat_exp` に適応する.

返り値

`decl_list` と, 残った式を, 以下のように `decl list` にして返す.

```
let flatten exp =  
  let toplevel_content, decl_list = get_flat_exp exp in  
  decl_list @ [RecDecl("_toplevel", [], toplevel_content)]
```

課題 8 (仮想機械コード生成: 必須)

vm.ml の trans 関数を完成させることにより, フラット表現から仮想機械コードへの変換を実現しなさい.