計算機科学実験Ⅳ

1029-28-9483 勝田 峻太朗

2018 年 10 月 19 日

目 次

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

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

フィボナッチ数

```
(* calculate the fibonacci number *)
let fib n =
loop v = (n, (1, 0)) in
if v.1 > 1 then
let tmp1 = v.2.1 in
let tmp2 = v.2.1 + v.2.2 in
recur (v.1 - 1, (tmp2, tmp1))
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

```
let reservedWords = [
  (* Keywords *)
  ("else", Parser.ELSE);
  ("false", Parser.FALSE);
  ("fun", Parser.FUN);
  ("if", Parser.IF);
```

```
  ("in", Parser.IN);
  ("let", Parser.LET);
  ("rec", Parser.REC);
  ("then", Parser.THEN);
  ("true", Parser.TRUE);
  ("loop", Parser.LOOP);
  ("recur", Parser.RECUR);
]
}

rule main = parse
  (* ignore spacing and newline characters *)
  [' ' '\009' '\012' '\n']+     { main lexbuf }
...
| "(*" { comment 1 lexbuf }
...
| "(" { Parser.LPAREN }
| ")" { Parser.RPAREN }
| ";;" { Parser.SEMISEMI }
| "+" { Parser.PLUS }
| "*" { Parser.MULT }
| "<" { Parser.LT }
| "=" { Parser.EQ }
| "->" { Parser.RARROW }
| "," { Parser.COMMA }
| "." { Parser.DOT }
...
and comment i = parse
  | "*)" { if i = 1 then main lexbuf else comment (i-1) lexbuf }
  | "(*" { comment (i+1) lexbuf }
  | _ {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

```
open Syntax

 %}

-%token LPAREN RPAREN SEMISEMI RARROW
+%token LPAREN RPAREN SEMISEMI RARROW COMMA DOT
 %token PLUS MULT LT EQ
-%token IF THEN ELSE TRUE FALSE LET IN FUN REC
+%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=LTExpr     { e }
17 +   | e=LoopExpr   { e }
18
19  LTExpr :
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,

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

は, 新しい変数 f を用いて,

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

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 =
```

```
    let rec recur_subst_loop = function
      | FunExp(id, e) -> FunExp(id, recur_subst_loop e)
      | ProjExp(e, i) -> ProjExp(recur_subst_loop e, i)
      | BinOp(op, e1, e2) -> BinOp(op, recur_subst_loop e1, recur_subst_loop e2)
      | LetExp(id, e1, e2) -> LetExp(id, recur_subst_loop e1, recur_subst_loop e2)
      | AppExp(e1, e2) -> AppExp(recur_subst_loop e1, recur_subst_loop e2)
      | LetRecExp(i1, i2, e1, e2) -> LetRecExp(i1, i2, recur_subst_loop e1, recur_subst_loop e2)
      | LoopExp(id, e1, e2) -> LoopExp(id, recur_subst_loop e1, recur_subst_loop e2)
      | TupleExp(e1, e2) -> TupleExp(recur_subst_loop e1, recur_subst_loop e2)
      | IfExp(cond, e1, e2) -> IfExp(recur_subst_loop cond, recur_subst_loop e1, recur_subst_loop e2)
      | RecurExp(e) -> AppExp(newf, e)
      | _ as e -> e in
    recur_subst_loop e in
  let rec recexp_of_loop = function
    | LoopExp(v, e1, e2) ->
      let new_funct: id = fresh_loopvar () in
      let rece1 = recur_subst (Var new_funct) (recexp_of_loop e2) in
      let rece2 = AppExp(Var new_funct, e1) in
      LetRecExp(new_funct, v, rece1, rece2)
    | _ as e -> e in
  match p with
  | Exp e -> Exp (recexp_of_loop e)

let rec read_eval_print env tyenv =
  print_string "# ";
  flush stdout;
  try
    let decl = Exp(Parser.toplevel Lexer.main (Lexing.from_channel stdin)) in
    (* remove loop exp from program *)
    let decl' = recprog_of_loop decl in
    (match decl' with
     | Exp e -> string_of_exp e |> print_endline);
    let ty, new_tyenv = ty_decl tyenv decl' in
    let (id, newenv, v) = eval_decl env decl' in
    ...
```

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

```
type exval =
  | IntV of int
  | BoolV of bool
  | TupleV of exval * exval
```

```
  | ProcV of id * exp * dnval Environment.t ref
and dnval = exval
```

また,eval_exp に対応する項目を追加した.

eval.ml

```
let rec eval_exp env = function
...
| TupleExp(e1, e2) ->
    let v1 = eval_exp env e1 in
    let v2 = eval_exp env e2 in
    TupleV(v1, v2)
| ProjExp(e, i) ->
    (match eval_exp env e with
    | TupleV(v1, v2) -> if i = 1 then v1
       else if i = 2 then v2
       else err "ProjExp: index not valid"
    | _ -> err "error: projection of non-tuple")
  | _ -> 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

```
  | TupleExp(e1, e2) ->
    let tyarg1, tysubst1 = ty_exp tyenv e1 in
    let tyarg2, tysubst2 = ty_exp tyenv e2 in
    let main_subst = unify(eqls_of_subst tysubst1 @ eqls_of_subst tysubst2) in
    let ty1 = subst_type main_subst tyarg1 in
    let ty2 = subst_type main_subst tyarg2 in
    (TyTuple(ty1, ty2), main_subst)
  | ProjExp(e, i) ->
    let tyarg, tysubst = ty_exp tyenv e in
    let t1 = TyVar(fresh_tyvar ()) in
    let t2 = TyVar(fresh_tyvar ()) in
    let main_subst = unify(eqls_of_subst tysubst @ [(tyarg, TyTuple(t1, t2))]) in
    if i = 1 then (subst_type main_subst t1, main_subst)
    else if i = 2 then (subst_type main_subst t2, main_subst)
    else err "fail"
```

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

```
(* ==== recur式が末尾位置にのみ書かれていることを検査 ==== *)
(* task4: S.exp -> unit *)
let rec recur_check e is_tail: unit =
  let recur_err () = err "illegal usage of recur" in
  S.(match e with
      | RecurExp _ ->
        if is_tail then ()
        else recur_err ()
      | LoopExp (x, e1, e2) ->
        recur_check e1 false;
        recur_check e2 true
      | IfExp(e1, e2, e3) ->
        recur_check e1 false;
        recur_check e2 is_tail;
        recur_check e3 is_tail
      | LetExp(x, e1, e2) ->
        recur_check e1 false;
        recur_check e2 is_tail
      | LetRecExp(f, x, e1, e2) ->
        recur_check e1 false;
        recur_check e2 is_tail
      | FunExp(_, e) | ProjExp(e, _) ->
        recur_check e false
      | BinOp(_, e1, e2) | AppExp(e1, e2) | TupleExp(e1, e2) ->
        recur_check e1 false;
        recur_check e2 false
      | _ -> () (* Var, ILit, BLit *)
    )

(* ==== entry point ==== *)
let rec convert prog =
  recur_check prog false;
  normalize prog
```

実行例

```
# loop v = (1, 0) in
if v.1 < 101 then
  (fun x -> recur x) (v.1 + 1, v.1 + v.2)
else
```

```
  v.2;;
Fatal error: exception Normal.Error("illegal usage of recur")
```

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

```
(* ==== 正規形への変換 ==== *)
let rec norm_exp (e: Syntax.exp) (f: cexp -> exp) (sigma: id Environment.t) =
  match e with
  | S.Var i ->
    let maybe_fail i =
      try f(ValExp(Var(Environment.lookup i sigma)))
      with Environment.Not_bound ->
        f (ValExp(Var ("_" ^ i ^ "temp")))
    in maybe_fail i
  | S.ILit i ->  f (ValExp (IntV i))
  | S.BLit b -> f (ValExp (IntV (int_of_bool b)))
  | BinOp(op, e1, e2) ->
    let x1 = fresh_id "bin" in
    let x2 = fresh_id "bin" in
    (norm_exp e1 (fun x ->
         (norm_exp e2 (fun y ->
              (LetExp(x2, y, LetExp(x1, x, f (BinOp(op, Var x1, Var x2))))))) sigma) sigma)
  | IfExp(cond, e1, e2) ->
    let x = fresh_id "if" in
    norm_exp cond (fun condy ->
        LetExp(x, condy, f (IfExp(Var x, norm_exp e1 (fun x -> CompExp x) sigma, norm_exp e2 (fun x ->
  | LetExp(id, e1, e2) ->
    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
  | 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)))
  | AppExp(e1, e2) ->
    let t1 = fresh_id "app" in
    let t2 = fresh_id "app" in
```

```
    norm_exp e1 (fun y1 ->
        (norm_exp e2 (fun y2 ->
            LetExp(t1, y1, LetExp(t2, y2, f (AppExp(Var t1, Var t2)))))) sigma) sigma
  | 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')
  | LoopExp(id, e1, e2) ->
    let loopvar = fresh_id "loopval" in
    let loopinit = fresh_id "loopinit" in
    let sigma' = Environment.extend id loopvar sigma in
    norm_exp e1 (fun y1 ->
        LetExp(loopinit, y1, LoopExp(loopvar, ValExp(Var loopinit), norm_exp e2 f sigma'))) sigma'
  | RecurExp(e) ->
    let t = fresh_id "recur" in
    norm_exp e (fun y1 ->
        LetExp(t, y1, RecurExp(Var t))) sigma
  | TupleExp(e1, e2) ->
    let t1 = fresh_id "tuple" in
    let t2 = fresh_id "tuple" in
    norm_exp e1 (fun y1 ->
        norm_exp e2 (fun y2 ->
            LetExp(t1, y1, LetExp(t2, y2, f (TupleExp(Var t1, Var t2))))) sigma) sigma
  | ProjExp(e, i) ->
    let t = fresh_id "proj" in
    norm_exp e (fun y ->
        LetExp(t, y, f (ProjExp(Var t, i)))) sigma
```

LetExp の実装

LetExp の正規化においては,let $x = e_1$ in e_2 は, 以下のように書き換えできる. ただし,$[\cdot]$ は, 正規変換を表すものとする.

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

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.

`norm_exp e2 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為である.

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 `fun x -> e` は, LetRec 式に以下の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 `let rec f x = e1 in e2` の正規形では,

- `e1` 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 `fun ce -> CompExp ce` を用いて, `norm_exp e1 (fun ce -> CompExp ce) sigma'` と変換している. また `e1` 内で `x` を含む可能性があるので, 既存の変換に,`x` と `funx` の対応を追加した `sigma'` を渡している.
- `e2` 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 `funf` の対応を追加した `sigma'` を渡している.

また,`e1`, `e2` 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, 他の実装とは異なり,`norm_exp` の呼び出しが `LetRec` 式内部に来るようになっている.

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

```
(* === conversion to closed normal form === *)
let rec closure_exp (e: N.exp) (f: cexp -> exp) (sigma: cexp Environment.t): exp =
  match e with
  | N.CompExp(N.ValExp(Var v)) ->
    let may_fail v =
      try
        f (Environment.lookup v sigma)
      with _ -> f (ValExp(Var ("_" ^ v)))
    in may_fail v
  | N.CompExp(N.ValExp(IntV i)) -> f (ValExp(IntV i))
  | N.CompExp(N.BinOp(op, v1, v2)) -> f (BinOp(op, convert_val v1, convert_val v2))
```

```
  | 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])))
  | N.CompExp(N.IfExp(v, e1, e2)) ->
    closure_exp e1 (fun y1 ->
        closure_exp e2 (fun y2 ->
            CompExp(IfExp(convert_val v, f y1, f y2))) sigma) sigma
  | N.CompExp(N.TupleExp (v1, v2)) -> f(TupleExp([convert_val v1; convert_val v2]))
  | N.CompExp(N.ProjExp (v, i)) -> f(ProjExp(convert_val v, i))
  | N.LetExp(id, ce1, e2) ->
    closure_exp (CompExp ce1) (fun y1 ->
        LetExp(convert_id id, y1, closure_exp e2 f sigma)) sigma
  | N.LetRecExp(funct, id, e1, e2) ->
    let recpointer = fresh_id ("b_" ^ funct) in
    let funct_tuple_list = (Var recpointer:: get_out_of_scope_variables e1 [id]) in
    let rec make_tuple_env l i env =
      match l with
      | Var hd:: tl ->
        let env' =  Environment.extend hd (ProjExp(convert_val (Var funct), i)) env in
        make_tuple_env tl (i+1) env'
      | [] -> env
      | _ -> (match l with
          | hd:: tl -> err ("unknown input in make_tuple_env" ^ string_of_closure(CompExp(ValExp(hd))))
          | _ -> err "none valid match") in
    let sigma' = make_tuple_env funct_tuple_list 0 Environment.empty in
    let closure_contents = TupleExp(funct_tuple_list) in
    let e2' = LetExp(convert_id funct, closure_contents, closure_exp e2 f sigma') in
    LetRecExp(recpointer, [convert_id funct; convert_id id], closure_exp e1 f sigma', e2')
  | N.LoopExp(id, ce1, e2) ->
    closure_exp (CompExp ce1) (fun y1 ->
        LoopExp(convert_id id, y1, closure_exp e2 f sigma)) sigma
  | N.RecurExp(v) -> RecurExp(convert_val v)

(* entry point *)
let convert e = closure_exp e (fun ce -> CompExp ce) Environment.empty
```

LetRec 式の実装

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.

```
let get_out_of_scope_variables (e: N.exp) (included: N.id list): value list =
  let rec loop_e ex accum incl =
    let rec loop_ce cex caccum incl =
```

```
      N.(match cex with
          | ValExp v | ProjExp(v, _) ->
            (match (List.find_opt (fun x -> Var x = v) incl), v with
             | Some x, _ -> caccum
             | None, Var _ -> MySet.insert v caccum
             | None, _ -> caccum)
          | BinOp(_, v1, v2) | AppExp(v1, v2) | TupleExp(v1, v2) ->
            MySet.union (loop_ce (ValExp v1) caccum incl) (loop_ce (ValExp v2) caccum incl)
          | IfExp(v, e1, e2) ->
            MySet.union (loop_ce (ValExp v) caccum incl) (MySet.union (loop_e e1 accum incl) (loop_e e2
    in
    N.(match ex with
        | LetExp(i, cex, ex) ->
          MySet.union (loop_ce cex accum incl) (loop_e ex accum (i::incl))
        | LoopExp(i, cex, ex) ->
          MySet.union (loop_ce cex accum incl) (loop_e ex accum incl)
        | LetRecExp(i1, i2, e1, e2) ->
          MySet.union (loop_e e1 accum (i2::incl)) (loop_e e2 accum (i2::incl))
        | _ -> accum
      )
  in
  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.

全体の流れ

LetRec 式においては, 関数の id だけだったものが, 関数クロージャと関数ポインターの 2 つが必要になる.

そこで, 以下のような操作をしている.

1. 関数ポインターを `recpointer` に,`fresh_id ("b_" ^ funct)` で生成する.(関数クロージャは関数名を引き継ぐ.)

2. funct_tuple_list に,get_out_of_scope_variables を用いて, クロージャに必要な値を代入する.
3. 関数本体式内でクロージャのインデックスを用いてスコープ外変数を参照しなければいけないので, 自由変数とクロージャからインデックスで値を取り出す表現の対応を make_tuple_env を用いて生成し,sigma' 代入する.

```
| N.LetRecExp(funct, id, e1, e2) ->
    let recpointer = fresh_id ("b_" ^ funct) in
    let funct_tuple_list = (Var recpointer:: get_out_of_scope_variables e1 [id]) in
    let rec make_tuple_env l i env =
      match l with
      | Var hd:: tl ->
        let env' =  Environment.extend hd (ProjExp(convert_val (Var funct), i)) env in
        make_tuple_env tl (i+1) env'
      | [] -> env
      | _ -> (match l with
          | hd:: tl -> err ("unknown input in make_tuple_env" ^ "\n" ^ string_of_closure(CompExp(ValExp
          | _ -> err "none valid match") in
    let sigma' = make_tuple_env funct_tuple_list 0 Environment.empty in
    let closure_contents = TupleExp(funct_tuple_list) in
    let e2' = LetExp(convert_id funct, closure_contents, closure_exp e2 f sigma') in
    LetRecExp(recpointer, [convert_id funct; convert_id id], closure_exp e1 f sigma', e2')
```

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. 関数クロージャから関数ポインタを得るには, 関数クロージャの先頭要素を取れば良い.

```
| 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.

```
(* ==== フラット化：変数参照と関数参照の区別も同時に行う ==== *)
let convert_id (i: C.id): id = i
let convert_id_list (il: C.id list): id list = il

let get_flat_exp ex =
  (* === helper functions === *)
  let fun_list = ref (MySet.empty: C.id MySet.t) in
  let append_fun v = fun_list := MySet.insert v !fun_list in
```

```
  let search_fun v = MySet.member v !fun_list in
  let decl_list = ref ([]: decl list) in
  let append_decl d = decl_list := (d :: !decl_list) in
  let convert_val (v: C.value): value =
    match v with
    | C.Var id -> if search_fun id
      then Fun(id)
      else Var(convert_id id)
    | C.IntV i -> IntV(i) in
  let convert_val_list (vl: C.value list): value list = List.map convert_val vl in
  let rec flat_exp (e: C.exp) (f: cexp -> exp): exp =
    match e with
    | C.CompExp(C.ValExp v) -> f (ValExp(convert_val v))
    | C.CompExp(C.BinOp(op, v1, v2)) ->
      let v1' = convert_val v1 in
      let v2' = convert_val v2 in
      f (BinOp(op, v1', v2'))
    | C.CompExp(C.AppExp(v, vl)) ->
      let v' = convert_val v in
      let vl' = convert_val_list vl in
      f (AppExp(v', vl'))
    | C.CompExp(C.IfExp(v, e1, e2)) ->
      let v' = convert_val v in
      flat_exp e1 (fun y1 ->
          flat_exp e2 (fun y2 ->
              f (IfExp(v', f y1, f y2))))
    | C.CompExp(C.TupleExp(vl)) -> f (TupleExp(convert_val_list vl))
    | C.CompExp(C.ProjExp(v, i)) -> f (ProjExp(convert_val v, i))
    | C.LetExp(id, ce, e) ->
      flat_exp (CompExp ce) (fun cy1 ->
          LetExp(convert_id id, cy1, flat_exp e f))
    | C.LetRecExp(funct, idl, e1, e2) ->
      append_fun funct;
      let letrec' = RecDecl(convert_id funct, convert_id_list idl, flat_exp e1 (fun x -> CompExp x)) in
      append_decl letrec';
      flat_exp e2 f
    | C.LoopExp(id, ce, e) ->
      let id' = convert_id id in
      flat_exp (CompExp ce) (fun cy1 ->
          LoopExp(id', cy1, flat_exp e f))
    | C.RecurExp(v) -> RecurExp(convert_val v)
  in let converted = flat_exp ex (fun x -> CompExp x) in
  (converted, !decl_list)

let flatten exp =
  let toplevel_content, decl_list = get_flat_exp exp in
  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.

vm.ml

```
(* ==== 仮想機械コードへの変換 ==== *)

let label_of_id (i: F.id): label = i

let trans_decl (F.RecDecl (proc_name, params, body)): decl =
  (* convert function names to label *)
  let proc_name' = label_of_id proc_name in
  (* generate new id *)
  let fresh_id_count = ref 0 in
  let fresh_id () =
    let ret = !fresh_id_count in
    fresh_id_count := ret + 1;
    ret in
  (* >>> association between F.Var and local(id)s >>> *)
  let var_alloc = ref (MyMap.empty: (F.id, id) MyMap.t) in
  let append_local_var (id: F.id) (op: id) = var_alloc := MyMap.append id op !var_alloc in
  let convert_id i =
    match MyMap.search i !var_alloc with
    | Some x -> x
```

```
20      | None -> let new_id: id = fresh_id () in
21        append_local_var i new_id;
22        new_id in
23    let operand_of_val v =
24      match v with
25      | F.Var id -> Local(convert_id id)
26      | F.Fun id -> Proc(id)
27      | F.IntV i -> IntV i in
28    (* get number of local var (that need to be allocated) *)
29    let n_local_var () = List.length(MyMap.to_list !var_alloc) in
30    (* <<< association between F.Var and local(id)s <<< *)
31    (* >>> remember loop >>> *)
32    let loop_stack = ref ([]: (id * label) list) in
33    let push_loop_stack (i, l) = loop_stack := (i, l) :: !loop_stack in
34    let pop_loop_stack () =
35      match !loop_stack with
36      | hd :: tl -> hd
37      | [] -> (114514, "temp_label") in
38    (* <<< remember loop <<< *)
39    let rec trans_cexp id ce: instr list =
40      match ce with
41      | F.ValExp(v) -> [Move(convert_id id, operand_of_val v)]
42      | F.BinOp(op, v1, v2) -> [BinOp(convert_id id, op, operand_of_val v1, operand_of_val v2)]
43      | F.AppExp(v, vl) -> [Call(convert_id id, operand_of_val v, List.map operand_of_val vl)]
44      | F.IfExp(v, e1, e2) ->
45        let new_label1 = "lab" ^ string_of_int(fresh_id ()) in
46        let new_label2 = "lab" ^ string_of_int(fresh_id ()) in
47        let e2' = trans_exp e2 [] ~ret:id in
48        let e1' = trans_exp e1 [] ~ret:id in
49        [BranchIf(operand_of_val v, new_label1)] @ e2' @ [Goto(new_label2); Label(new_label1)] @ e1' @ [L
50      | F.TupleExp(vl) -> [Malloc(convert_id id, List.map operand_of_val vl)]
51      | F.ProjExp(v, i) -> [Read(convert_id id, operand_of_val v, i)]
52    and trans_exp (e: F.exp) (accum_instr: instr list) ?(ret="default"): instr list =
53      match e with
54      | F.CompExp(ce) ->
55        if ret = "default" then
56          let return_id: F.id = "ret" ^ (string_of_int (fresh_id())) in
57          (match ce with
58           | F.ValExp(Var id) -> accum_instr @ [Return(operand_of_val (F.Var id))]
59           | _ -> let ret_assign_instr = trans_cexp return_id ce in
60             accum_instr @ ret_assign_instr @ [Return(operand_of_val (F.Var return_id))])
61        else let ret_assign_instr = trans_cexp ret ce in
62          accum_instr @ ret_assign_instr
63      | F.LetExp(id, ce, e) ->
64        let instr' = accum_instr @ trans_cexp id ce in
65        instr' @ trans_exp e [] ~ret
66      | F.LoopExp(id, ce, e) ->
67        let loop_label = "loop" ^ (string_of_int (fresh_id ())) in
68        push_loop_stack (convert_id id, loop_label);
```

```
69        trans_cexp id ce @ [Label (loop_label)] @ trans_exp e [] ~ret:"default"
70      | F.RecurExp(v) ->
71        let (id, loop_lab) = pop_loop_stack () in
72        [Move(id, operand_of_val v); Goto(loop_lab)]
73    in ProcDecl(proc_name', n_local_var (), trans_exp body [] ~ret:"default")
74
75  (* entry point *)
76  let trans = List.map trans_decl
```

補助的な値, 関数

var_alloc: (F.id, id) MyMap.t 平滑化後の id と,Vm.id の関係を保持する. 変換後のこの Map の長さが, 必要な local 変数の数となる.

convert_id: F.id -> id id の変換を行う. var_alloc にすでに変換が存在すればそれを返し, なければ, 新しい id を生成し,var_alloc に記録する.

operand_of_val: F.value -> operand 値の変換を行う.

loop_stack 現在どの loop 文にいるかを保持する.

trans_cexp: F.id -> F.cexp -> instr list cexp の変換を行う. F.cexp を F.id に代入するような命令列を生成する.

Vm.exp への変換

変換は,trans_exp で行う.

引数

e: F.exp 変換対象の F.exp

accum_instr: instr list toplevel に来る表現を持ち回るための引数.

ret: F.id 変換後の表現が返り値となる場合,"default"が入れられ, 変換後の表現がある id'が代入される場合はその id が入力される.

F.CompExp の変換

trans_exp に F.CompExp(cexp) が入力されたとき

1. ret="default"であったとき

   Return を通じて,cexp が返り値となる.

2. ret="some_id"であったとき,

   trans_cexp を用いて,cexp を some_id に代入する命令列を生成する.