レポート**2**（インタプリタ）

計算機科学実験及演習3（ソフトウェア）

Dang Viet Trung (1029272735)

締切り：7月6日（金）

京都大学・工学部
情報学科

1 ML[1]

1.1 課題3.2.1

main.mlの初期環境にデフォルト値を追加すればいい。

```
let initial_env = Environment.append Environment.empty
  [("i", IntV 1); ("ii", IntV 2); ("iii", IntV 3); ("iv", IntV 4); ("v", IntV
      5); ("x", IntV 10)]
```

1.2 課題3.2.2 (⋆⋆)

```
(* main.ml *)
let err msg = Printf.printf "Error: %s\n" msg

let rec read_eval_print env =
  ...
  try ...
  with
  | Error msg -> err msg; read_eval_print env
    ...
```

main.mlは上のように調整すれば、エラが発生しても終わらずにエラを通信して続ける。

1.3 課題3.2.3 (⋆)

まずlexerとsyntaxを定義する。

```
(* lexer.mli *)
rule main = parse
...
| "||" { Parser.OR }
| "&&" { Parser.AND }]
```

```
(* parser.mly *)
AndExpr:
  l=AExpr AND r=AExpr { BinOp(And, l, r) }
| AND { OpFunExp (And) }

OrExpr:
  l=AExpr OR r=AExpr { BinOp(Or, l, r) }
| OR { OpFunExp (Or) }
```

ANDとORは二項演算子なので、結果を得るためapply_primに項目を追加すれば十分。

```
(* eval.ml *)
let rec apply_prim op arg1 arg2 = match op, arg1, arg2 with
...
| And, BoolV b1, BoolV b2 -> BoolV (b1 && b2)
| And, _, _ -> err ("Both arguments must be boolean: &&")
| Or, BoolV b1, BoolV b2 -> BoolV (b1 || b2)
| Or, _, _ -> err ("Both arguments must be boolean: ||")
```

1.4 課題3.2.4 (⋆⋆)

```
(* lexer.mli *)
rule main = parse
...
| '(' '*' [^'*']* '*' ')' { main lexbuf }
```

個の課題において (**) のなか*がある場合にまだ対応していない。

2 ML^2

2.1 課題3.3.1

資料のように作成した。

2.2 課題3.3.2 (⋆⋆)

Decl代わりにDeclsを定義する。これは複数の(id * exp)を受理し、一つ以上のletを可能にする。

```
(* syntax.ml *)
type program =
...
| Decls of (id * exp) list
```

toplevelの構文で複数のlet定義を有効する。

```
(* parser.mly *)
TopLetExpr :
| LET x=ID EQ e=Expr ls=TopLetExpr { (x, e)::ls }
| LET x=ID EQ e=Expr { [(x, e)] }

toplevel :
...
| ls=TopLetExpr SEMISEMI { Decls ls }
```

計算するevalのコードが変わらないが、eval_declにはDeclsで格納されたlet定義を一個ずつ読んで計算する。結果もそれに相当にに出す。毎回計算するとき、今まで得た環境で行う。

```
(* eval.ml *)
let eval_decl env = function
...
| Decls id_exp_ls ->
    let newenv = ref env in
    (* Add id and value to the extended newenv *)
    let rec decl_ls = function
    | [] -> []
    | (id, exp)::rest ->
        (* All expressions are calculated under new env *)
        let value = eval_exp !newenv exp in
        newenv := Environment.extend id value !newenv;
```

```
        (id, value)::decl_ls rest
    in
    (decl_ls id_exp_ls, !newenv)
...
```

最後に結果を出力するコードも調整する必要がある。

```
(* main.ml *)
let rec print_result ls = List.iter
  (function (id, v) -> printf "val %s = " id; pp_val v; print_newline ()) ls
```

2.3 課題3.3.3 (⋆⋆)

main.mlに

1. contains_semisemi s: sが ';;' を含むかチェックする
2. get_instr instr: ';;' が含まれる次の行まで読み込む

を定義しておいた。コードを読むとき使う。

```
let rec read_eval_print env =
  print_string("# ");
  flush stdout;
  let instr = get_instr "" in ...
```

2.4 課題3.3.4 (⋆⋆)

まず、LetExpを多数の定義ができるように変える。

```
(* syntax.ml *)
type exp =
...
| LetExp of (id * exp) list * exp
```

```
(* parser.mly *)
LetAndExpr :
| x=ID EQ e=Expr ANDKW ls=LetAndExpr { (x,e)::ls }
| x=ID EQ e=Expr { [(x ,e)] }

LetExpr :
| LET ls=LetAndExpr IN e=Expr { LetExp(ls, e) }
```

計算するとき、まずletで定義された変数を元々の環境で値を得る。途中でidと結果を新しい環境に入れ、最後にその環境で最後の結果を得る。

```
(* eval.ml *)
let rec eval_exp env = function
| LetExp(ls, ret_exp) ->
    (* Allow multiple 'let' with 'and' connected *)
    (* Extend env with list of (id, exp) with calculation under original env
       *)
```

```
    let _env_ = let f = fun newenv id_exp ->
      match id_exp with (id, exp) -> Environment.extend id (eval_exp env exp)
          newenv in
    List.fold_left f env ls in
    eval_exp _env_ ret_exp
```

また

```
let a = 1 and b = 2;
```

を可能にするため、eval.mlのeval_declのなかAndDeclsというパターンを追加しておいた。同じ操作を行うこと。

3 ML^3

3.1 課題3.4.1

資料のように作成した。

3.2 課題3.4.2 (⋆⋆)

中置演算子は普通の関数のように見える。それに対して、関数系を返す。

```
(* syntax.ml *)
type exp = ...
| OpFunExp of binOp
...
```

```
(* eval.ml *)
let rec eval_exp env = function ...
| OpFunExp (op) ->
    ProcV (["x"; "y"], BinOp(op, Var("x"), Var("y")), ref env)
...
```

3.3 課題3.4.3 (⋆)

関数は多変数を受理できることにする。

```
(* syntax.ml *)
type exp = ...
| FunExp of id list * exp
...
```

二つの形を受理する。 まずfun x1 ... xn -> ...

```
(* parser.mly *)
FunExpr :
| FUN ls_param=FunParamListExpr RARROW e=Expr { FunExp(ls_param, e) }

FunParamListExpr :
| id=ID ls_param=FunParamListExpr { id :: ls_param }
| id=ID { [id] }
```

let f x1 ... xn = ...

```
(* parser.mly *)
TopLetExpr :
...
| LET x=ID ls_param=FunParamListExpr EQ e=Expr { [(x, FunExp(ls_param, e))] }
```

f x1 x2 ... xnを適用するとき：

1. fからProcV (ls_id, body, env')を得る。ls_idは関数の引数のidのリスト、bodyは関数の内容。
2. 適合公式の中の全部の適合値(ls_exp)を計算して、相応的なidと一緒にenvに入れる。
3. ここで三つの場合にわける。
   - 適合値数＝関数パラメータ数：適合して値を返す
   - 適合値数＞関数パラメータ数：残りのパラメータを新しいパラメータとして関数を作って返す
   - 適合値数＜関数パラメータ数：残りの適合値を適合下結果で適合して続ける。

```
let rec eval_exp env = function
...
| AppExp (expfun, ls_exp) -> (match (eval_exp env expfun) with
    | ProcV (ls_id, body, env') ->
        let _env_ = ref Environment.empty in _env_ := !env';
        let (ids, exps) = extend_env _env_ env ls_id ls_exp in
        (* Apply *)
        if (List.length ids) > 0 then
          (* More parameters than applied values -> return new function *)
          ProcV(ids, body, _env_)
        else if (List.length exps) > 0 then
          (* More applied values than parameters -> continue applying *)
          let ret_exval = eval_exp !_env_ body in
          match ret_exval with ProcV(ls_id2, body2, env2') ->
            eval_exp !_env_ (AppExp (FunExp (ls_id2, body2), exps))
        else eval_exp !_env_ body
    | _ -> err ("None-function value is applied: " ^ string_of_exp expfun))
```

結果

1. 適用パラメータの方が多い

```
let add = fun x y -> x + y in
let add4 = add 4 in
add4 5;; (* Result: 9 *)
```

2. 関数パラメータの方が多い

```
let add a b = fun c -> a + b + c in
add 1 2 3;; (* Result: 6 *)
```

3.4 課題3.4.5 (⋆)

まずdfunを導入する（funとほぼ同じ）

```
type exval =
...
| DProcV of id * exp * dnval Environment.t ref

let rec eval_exp env = function
...
| DFunExp (id, exp) -> DProcV(id, exp, ref env)
...
```

update_envという関数を定義しておいた。それはenv1とenv2を引数として受け、env2からenv1に値をアップデートして、新しい環境を戻す。この関数を用いてdfunの計算は下のようなコードで実行する。

```
(* eval.ml *)
let rec eval_exp env = function
| AppExp (expfun, ls_exp) -> (match (eval_exp env expfun) with
    | ProcV (ls_id, body, env') -> ...
    | DProcV (id, body, env') ->
        let _env_ = update_env !env' env in
        let exp = List.hd ls_exp in
        _env_ := Environment.extend id (eval_exp env exp) !_env_;
        eval_exp !_env_ body
```

3.5 課題3.4.6 (⋆)

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

を実行すると、120を得る。この理由は二目のfactを呼び出すとき毎回に新しい環境を作って前の値を指定しておく。

fact 5 -> fact_2 5 -> 5 * fact_1 4 -> 5 * fact_2 4 -> ...

という手順で進んでいた。

4 ML[4]

4.1 課題3.5.1

```
(* parser.mly *)
LetRecExpr :
| LET REC ls=LetRecAndExpr IN e=Expr { LetRecExp(ls, e) }
```

資料のように作成した。課題3.5.2にはand機能も含めてコードを示し、説明しておいた。

4.2 課題3.5.2 (⋆⋆)

```
(* syntax.ml *)
type exp =
...
| LetRecExp of (id * id * exp) list * exp
```

```
(* parser.mly *)
LetRecAndExpr :
| x=ID param=ID EQ e=Expr ANDKW ls=LetRecAndExpr { (x, param, e)::ls }
| x=ID EQ FUN param=ID RARROW e=Expr ANDKW ls=LetRecAndExpr { (x, param, e)::
    ls }
| x=ID param=ID EQ e=Expr { [(x, param, e)] }
| x=ID EQ FUN param=ID RARROW e=Expr { [(x, param, e)] }

LetRecExpr :
| LET REC ls=LetRecAndExpr IN e=Expr { LetRecExp(ls, e) }
```

やるときに、再帰関数の計算のように、各定義に対してdummyenvを作って埋める。終わったらすべての関数の環境を更新して、目標の公式を計算する。

```
(* eval.ml *)
let rec eval_exp env = function
...
| LetRecExp (ls, ret_exp) ->
    (* Mutual recursion *)
    let rec extend_env ls newenv dummyenv_ls = match ls with
      | [] ->
          (* Update environment of all Proc *)
          List.map (fun dummyenv -> (dummyenv := newenv;)) dummyenv_ls;
          newenv
      | it::rest -> (match it with (id, param, exp) ->
        let dummyenv = ref Environment.empty in
          let newenv' = Environment.extend id (ProcV ([param], exp, dummyenv)
              ) newenv in
            dummyenv := newenv';
            extend_env rest newenv' (dummyenv::dummyenv_ls)
      )
    in eval_exp (extend_env ls env []) ret_exp
```

以下のような例で成功だった。

```
let rec is_even n = if n = 0 then true else is_odd (n - 1)
and is_odd n = if n = 0 then false else is_even (n - 1) in
is_even(99);; (* Result: false *)
```

プログラム型 (in がない) も対応できるように、eval_declはこうようなコードに調整する。

```
let eval_decl env = function
...
| RecDecls id_exp_ls ->
  let id_val_ls = ref [] in
```

```
  (* Mutual recursion *)
  let rec extend_env ls newenv dummyenv_ls = match ls with
  | [] ->
    (* Update environment of all Proc *)
    List.map (fun dummyenv -> (dummyenv := newenv;)) dummyenv_ls;
    newenv
  | it::rest -> (match it with (id, param, exp) ->
    let dummyenv = ref Environment.empty in
    let proc = ProcV ([param], exp, dummyenv) in
      let newenv' = Environment.extend id proc newenv in
        dummyenv := newenv';
        id_val_ls := (id, proc) :: !id_val_ls;
        extend_env rest newenv' (dummyenv::dummyenv_ls)
    ) in
    let newenv = extend_env id_exp_ls env [] in
    (!id_val_ls, newenv)
```

5 ML[5]

5.1 課題3.6.1 (⋆⋆)

課題3.6.1で拡張されたパターンについて説明する。

5.2 課題3.6.2 (⋆)

```
(* lexer.mli *)
rule main = parse
...
| "[]" { Parser.EMPTYLIST }
| "[" { Parser.LSQBRACKET }
| "]" { Parser.RSQBRACKET }
```

```
(* syntex.ml *)
type exp =
...
| ListExp of exp list
```

```
(* parser.mly *)
ListItemExpr :
| exp=Expr SEMI ls=ListItemExpr { exp::ls }
| exp=Expr { [exp] }

ListExpr :
| LSQBRACKET ls=ListItemExpr RSQBRACKET { ListExp(ls) }
| EMPTYLIST { ListExp ([]) }
```

`string_of_eval`という関数を定義し、リストから文字列に変える関数である。例えば1, 2, 3要素からなるリストに大して"[1; 2; 3]"が返される。

```
(* eval.ml *)
let rec string_of_exval = function
...
| ListV (ls) -> let rec string_of_list ls ret = match ls with
  | [] -> ret ^ "]"
  | [x] -> ret ^ (string_of_exval x) ^ "]"
  | item::rest -> string_of_list rest (ret ^ (string_of_exval item) ^ "; ")
  in string_of_list ls "["
...

let rec eval_exp env = function
...
| ListExp (exps) ->
    let rec eval_list exps values = match exps with
      | [] -> values
      | exp :: rest -> eval_list rest (values  [eval_exp env exp])
    in ListV (eval_list exps [])
```

5.3 課題3.6.3 (⋆)

ListHeadTailをマッチン グするとき、簡単に条件確認コードを追加すればいい。

```
if head_id = tail_id then err("Variable " ^ head_id ^ " is bound several
    times in this matching") else ...
```

5.4 課題3.6.4 (⋆⋆⋆)

今三種類のmatching patternが対応される。

- []: EmptyList
- [x]: SingleElementList
- hd::tail: ListHeadTail

命令で全部じゃなくて、必要なパターンだけ指定してもいい。match命令の一般のパターンがない。

```
(* syntax.ml *)
type matchPattern =
| EmptyList
| SingleElementList of id
| ListHeadTail of id * id

type exp =
...
| MatchExp of exp * (matchPattern * exp) list
```

- MatchPatternExpr: [], [id], head::tailなどにマッチする。

- MatchListExprs: [] -> exp1 | [id] -> exp2などにマッチする。
- MatchExpr: matchの完全な構文。

と言う意味でparser.mlyで定義される。

マッチン グするとき、eval.mlでmatch_expという関数より処理されマッチン グ場合の結果値を返す。例えばEmptyListとSingleElementListのコードを下に表せる。

```
match_exp env value = function
| [] -> err "No matching pattern found."
| mp::mp_rest -> match mp with (pattern, ret_exp) ->
    (match pattern with
    (* [] *)
    | EmptyList -> (match value with ListV ls ->
        match ls with
        | [] -> eval_exp env ret_exp
        | _ -> match_exp env value mp_rest)
    (* [x] *)
    | SingleElementList(id) -> (match value with ListV ls ->
        match ls with
        | [x] -> let newenv = Environment.extend id x env in
            eval_exp newenv ret_exp
        | _ -> match_exp env value mp_rest)
...
```

5.5 課題3.6.5 (⋆⋆)

AExprExを定義してAExprにlet, if などに追加して作成される公式とする。次に適当なところでAExprかわりにAExprExを使う。

```
(* parser.mly *)
AExprEx :
| i=INTV { ILit i }
| TRUE   { BLit true }
| FALSE  { BLit false }
| i=ID   { Var i }
| LPAREN e=Expr RPAREN { e }
| e=IfExpr { e }
| e=LetExpr{ e }
| e=LetRecExpr { e }
| e=FunExpr { e }
| e=DFunExpr { e }
| e=ListExpr { e }
| e=MatchExpr { e }

AppExpr :
| e=AppExpr ls=AppParamListExpr { AppExp(e, ls) }
| e=AExpr { e }
| e=AExprEx { e }

...
```

6 感想

各課題を一個ずつ解けました。3.4.4を残して必須課題と自由課題を全部解けそうです。簡単な例プログラムでテストしてきたが、ミスがまだあるかもしれません。課題3.4.4はあまり理解できないので残しました。

結構解けましたが、途中で困難なことも様々にあります。一番困っていたのはデバッグだともいます。デバッグ補助関数をいくつかを書くことになりました。ですが、構文の定義のところとかでミスがあれば、具体劇に分からないです。直すのは時間がかかりました。

最後に提出したら、バーグが結構出ました。できたと思った課題が間違ってしまったのです。全部直すことができなかったんです。