ML interpreter

http://www.fos.kuis.kyoto-u.ac.jp/~nishida/classes/isle4fp2014/text.pdf

How to test

```
opam install ocamlfind ounit re
make test
```

Solved exercises

- Ex3.1 [必修]
- Ex3.2 [**]
- Ex3.3 [*]
- Ex3.4 [必修]
- Ex3.6 [**]
- Ex3.8 [必修]
- Ex3.14 [必修]
- Ex4.1 [必修]
- Ex4.2 [必修]
- Ex4.3 [必修]

- Ex4.4 [必修]
- Ex4.5 [必修]
- Ex4.6 [必修]

Ex3.1

```
Exercise 3.1 [必修課題]
ML1 インタプリタのプログラムをコンパイル・実行し，インタプリタの動作を確かめよ．
大域環境として i，v，x の値のみが定義されているが，ii が 2，iii が 3，iv が 4 となるようにプログラムを変更して，動作を確かめよ．例えば，
iv + iii * ii
などを試してみよ．
```

大域変数を以下のように変更する.

```
let initial_env =
  Environment.extend "i" (IntV 1)
    (Environment.extend "ii" (IntV 2)
      (Environment.extend "iii" (IntV 3)
        (Environment.extend "iv" (IntV 4)
          (Environment.extend "v" (IntV 5)
             (Environment.extend "x" (IntV 10) Environment.empty)))))
```

`iv + iii * ii` を実行した結果、 `10` が返ってきた.

Ex3.2

```
Exercise 3.2 [**]
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と，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に改造せよ．
```

プログラムを実行した際にエラーを受け取ったら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に、 `try with` 構文で例外を受け取った際の処理を加えればよい.

`main.ml` の `read_eval_print` を以下のように変更する.

```
let rec read_eval_print env =
  print_string "# ";
  flush stdout;
  let showError str = Printf.printf "%s" str;
    print_newline();
    read_eval_print env in
  (try
```

```
      let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
      let (id, newenv, v) = eval_decl env decl in
        Printf.printf "val %s = " id;
        pp_val v;
        print_newline();
        read_eval_print newenv
      with Failure str -> showError str
      | Eval.Error str -> showError str
      | Parsing.Parse_error -> showError "parse error"
      | _ -> showError "Other Exception")
```

Ex3.3

```
Exercise 3.3 [*]
論理値演算のための二項演算子 &&, || を追加せよ.
```

`&&` と `||` が AND と OR としてparseされるよう、 `parser.mly` と `lexer.mll` を変更する.

```
--- a/interpreter/lexer.mll
+++ b/interpreter/lexer.mll
@@ -24,6 +22,8 @@ rule main = parse
 | "+" { Parser.PLUS }
 | "*" { Parser.MULT }
 | "<" { Parser.LT }
+| "&&" { Parser.AND }
+| "||" { Parser.OR }
```

```
--- a/interpreter/parser.mly
+++ b/interpreter/parser.mly
@@ -4,7 +4,7 @@ open Syntax

 %token LPAREN RPAREN SEMISEMI
 %token PLUS MULT LT
-%token IF THEN ELSE TRUE FALSE
+%token IF THEN ELSE TRUE FALSE AND OR
```

その後、And, Orをparseした場合に `eval.ml` のBinOpに渡され、BinOp内で正しく動作するようにする.

この際、式中の演算での優先順位は、ORが一番低く次にANDが来るようになるように注意する.

```
--- a/interpreter/parser.mly
+++ b/interpreter/parser.mly
@@ -18,6 +18,14 @@ toplevel :
```

```
 Expr :
     IfExpr { $1 }
+  | OrExpr { $1 }
+
+OrExpr :
+    AndExpr OR AndExpr { BinOp(Or, $1, $3) }
+  | AndExpr { $1 }
+
+AndExpr :
+    LTExpr AND LTExpr { BinOp(And, $1, $3) }
```

```
--- a/interpreter/eval.ml
+++ b/interpreter/eval.ml
@@ -23,21 +23,25 @@ let rec apply_prim op arg1 arg2 = match op, arg1, arg2 with
    | Mult, _, _ -> err ("Both arguments must be integer: *")
    | Lt, IntV i1, IntV i2 -> BoolV (i1 < i2)
    | Lt, _, _ -> err ("Both arguments must be integer: <")
+  | And, BoolV b1, BoolV b2 -> BoolV (b1 && b2)
+  | And, _, _ -> err ("Both arguments must be bool: &&")
+  | Or, BoolV b1, BoolV b2 -> BoolV (b1 || b2)
+  | Or, _, _ -> err ("Both arguments must be bool: ||")
```

```
--- a/interpreter/syntax.ml
+++ b/interpreter/syntax.ml
@@ -1,7 +1,7 @@
 (* ML interpreter / type reconstruction *)
 type id = string

-type binOp = Plus | Mult | Lt
+type binOp = Plus | Mult | Lt | And | Or
```

Ex3.4

```
Exercise 3.4 [必修課題]
ML2 インタプリタを作成し，テストせよ．
```

資料のテキストの図7に従ってML2インタプリタを作成した.

テストに関しては `test/interpreter.ml` に `let x = 1;;` をテストするコードを作成し、通過することを確認した.

```
let test_let =
  let decl = Parser.toplevel Lexer.main (Lexing.from_string "let x = 1;;") in
  let (id, _, v) = Eval.eval_decl Environment.empty decl in
  let check_id _ = assert_equal id "x" in
  let check_val _ = assert_equal (Eval.string_of_exval v) "1" in
  "test let">:::
```

```
  ["check id">:: check_id;
  "check val">:: check_val;]
;;
```

Ex3.6

```
Exercise 3.6 [**]
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.（オプション: ;; で区切られたプログラムの列が読み込めるようにせよ. ）
```

引数に与えられたファイル名からファイルの内容を読み取り、そのプログラムを評価するように `main.ml` に変更を加えた.

```
--- a/interpreter/main.ml
+++ b/interpreter/main.ml
@@ -1,24 +1,34 @@
 open Syntax
 open Eval

-let rec read_eval_print env =
-  print_string "# ";
-  flush stdout;
-  let showError str = Printf.printf "%s" str;
-    print_newline();
-    read_eval_print env in
+let eval_print env lexer showError =
   (try
-    let decl = Parser.toplevel Lexer.main (Lexing.from_channel stdin) in
+    let decl = Parser.toplevel Lexer.main lexer in
     let (id, newenv, v) = eval_decl env decl in
       Printf.printf "val %s = " id;
       pp_val v;
       print_newline();
-      read_eval_print newenv
+      newenv
     with Failure str -> showError str
     | Eval.Error str -> showError str
     | Parsing.Parse_error -> showError "parse error"
     | _ -> showError "Other Exception")

+let rec read_eval_print env =
+  print_string "# ";
+  flush stdout;
+  let showError str = Printf.printf "%s" str;
+    print_newline();
+    read_eval_print env in
+  let newenv = eval_print env (Lexing.from_channel stdin) showError in
+  read_eval_print newenv
```

```
+
+let file_eval_print filename env =
+  let showError str = Printf.printf "%s" str;
+    print_newline();
+    env in
+  eval_print env (Lexing.from_channel (open_in filename)) showError
+
 let initial_env =
   Environment.extend "i" (IntV 1)
     (Environment.extend "ii" (IntV 2)
@@ -27,4 +37,6 @@ let initial_env =
            (Environment.extend "v" (IntV 5)
              (Environment.extend "x" (IntV 10) Environment.empty)))))
-let _ = read_eval_print initial_env
+let _ =
+  if (Array.length Sys.argv) > 1 then file_eval_print Sys.argv.(1) initial_env
+  else read_eval_print initial_env
```

このプログラムでは引数がある場合は `file_eval_print` を、 そうでない場合は従来通り `read_eval_print` 関数を呼び出すようになっている.

プログラムを評価し出力する部分に関しては `eval_print` 関数として用意してあり、 `read_eval_print` は標準入力から渡された文字列を、 `file_eval_print` はファイルの内容を `eval_print` 関数に渡すようにしてある.

また、プログラムの評価時にエラーが発生した場合、 `file_eval_print` ではプログラムが終了し、 `read_eval_print` では従来通りインタプリタプロンプトに戻るように設計してある.

また、コメントを無視するように `lexer.mll` に以下のような変更を加えた

```
--- a/interpreter/lexer.mll
+++ b/interpreter/lexer.mll
@@ -11,10 +11,14 @@ let reservedWords = [
 ]
 }

+let openComment = "(*"
+let closeComment = "*)"
+
 rule main = parse
   (* ignore spacing and newline characters *)
   [' ' '\009' '\012' '\n']+     { main lexbuf }

+| openComment { comment 1 lexbuf }
 | "-"? ['0'-'9']+
     { Parser.INTV (int_of_string (Lexing.lexeme lexbuf)) }

@@ -36,3 +40,8 @@ rule main = parse
       _ -> Parser.ID id
```

```
      }
 | eof { exit 0 }
+| _ { main lexbuf }
+and comment depth = parse
+    openComment { comment (depth + 1) lexbuf }
+  | closeComment { if depth > 1 then comment (depth - 1) lexbuf else main lexbuf
+  | _ { comment depth lexbuf }
```

入れ子になっているコメントにも対応するために `comment` 関数ではコメントの深さを引数に取り、 コメントの深さが1になるまで無視するようにしてある.

Ex3.8

```
Exercise 3.8 [必修課題]
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．
```

テキストの図9に従い変更を加え、ML3インタプリタを作成した.

また、いくつか変更点を加える事で正しく動作するようにした.

関数を文字列化するため、 `string_of_exval` 関数に `ProcV` が与えられた場合に `"<fun>"` が返るようにした.

```
--- a/interpreter/eval.ml
+++ b/interpreter/eval.ml
@@ -14,6 +14,7 @@ let err s = raise (Error s)
 let rec string_of_exval = function
     IntV i -> string_of_int i
   | BoolV b -> string_of_bool b
+  | ProcV (_, _, _) -> "<fun>"
```

`FunExpr` をparserに追加し、 `FunExp` 関数が呼ばれるようにした.

```
--- a/interpreter/parser.mly
+++ b/interpreter/parser.mly
@@ -25,6 +25,9 @@ Expr :
   | OrExpr { $1 }
   | FunExpr { $1 }

+FunExpr :
+    FUN ID RARROW Expr { FunExp($2, $4) }
+
```

また、高階関数のテストとして `test/interpreter.ml` に以下の様なテストを用意し、正しく動作することを確認した.

```
let test_fun =
  let program = "let x = fun y -> y + 1 in x 4;;" in
  let check_id _ = check_id_of_program program "-" in
  let check_val _ = check_value_of_program program "5" in
  let high_order_function = "let apply_one = fun f -> f 1 in let plus = fun x ->
  let check_val_for_high_order_function _ = check_value_of_program high_order_fur
  "test fun">:::
  ["check_id">:: check_id;
  "check_val">:: check_val;
  "check_val_for_high_order_function">:: check_val_for_high_order_function;]
;;
```

Ex3.14

Exercise 3.14 [必修課題]
図に示した syntax.ml にしたがって，parser.mly と lexer.mll
を完成させ，ML4 インタプリタを作成し，テストせよ．(let rec 宣言も実装すること．)

テキストに従い `parser.mly` と `lexer.mll` に変更を加えた。

```
--- a/interpreter/lexer.mll
+++ b/interpreter/lexer.mll
@@ -9,6 +9,7 @@ let reservedWords = [
   ("let", Parser.LET);
   ("in", Parser.IN);
   ("fun", Parser.FUN);
+  ("rec", Parser.REC);
 ]
 }
```

```
--- a/interpreter/parser.mly
+++ b/interpreter/parser.mly
@@ -6,7 +6,7 @@ open Syntax
 %token PLUS MULT LT
 %token IF THEN ELSE TRUE FALSE AND OR
 %token LET IN EQ
-%token RARROW FUN
+%token RARROW FUN REC

 %token <int> INTV
 %token <Syntax.id> ID
@@ -18,12 +18,17 @@ open Syntax
 toplevel :
     Expr SEMISEMI { Exp $1 }
   | LET ID EQ Expr SEMISEMI { Decl ($2, $4) }
+  | LET REC ID EQ FUN ID RARROW Expr SEMISEMI { RecDecl ($3, $6, $8) }

 Expr :
```

```
      IfExpr { $1 }
    | LetExpr { $1 }
    | OrExpr { $1 }
    | FunExpr { $1 }
+   | LetRecExpr { $1 }
+
+LetRecExpr :
+    LET REC ID EQ FUN ID RARROW Expr IN Expr { LetRecExp ($3, $6, $8, $10) }

 FunExpr :
     FUN ID RARROW Expr { FunExp($2, $4) }
```

また、 `RecDecl` を `eval.ml` に実装する必要があるため、 `eval.ml` にも変更を加えた。

```
--- a/interpreter/eval.ml
+++ b/interpreter/eval.ml
@@ -3,7 +3,7 @@ open Syntax
 type exval =
     IntV of int
   | BoolV of bool
-  | ProcV of id * exp * dnval Environment.t
+  | ProcV of id * exp * dnval Environment.t ref
 and dnval = exval

 exception Error of string
@@ -49,17 +49,28 @@ let rec eval_exp env = function
   | LetExp (id, exp1, exp2) ->
       let value = eval_exp env exp1 in
           eval_exp (Environment.extend id value env) exp2
-  | FunExp (id, exp) -> ProcV (id, exp, env)
+  | FunExp (id, exp) -> ProcV (id, exp, ref env)
   | AppExp (exp1, exp2) ->
     let funval = eval_exp env exp1 in
     let arg = eval_exp env exp2 in
       (match funval with
           ProcV (id, body, env') ->
-            let newenv = Environment.extend id arg env' in
+            let newenv = Environment.extend id arg env'.contents in
               eval_exp newenv body
         | _ -> err ("Non-function value is applied"))
+  | 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

 let 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)
+  | RecDecl (id, para, exp) ->
+    let dummyenv = ref Environment.empty in
```

```
+    let v = ProcV (para, exp, dummyenv) in
+    let newenv = Environment.extend id v env in
+      dummyenv := newenv;
+      ("-", newenv, v)
```

また、テストには以下の様なものを書いた。

```
let test_rec_fun =
  let let_rec_exp = "let rec x = fun y -> if y < 1 then 1 else (x (y  + (-1))) *
  let check_id_for_let_rec_exp _ = check_id_of_program let_rec_exp "-" in
  let check_val_for_let_rec_exp _ = check_value_of_program let_rec_exp "24" in
  let rec_decl = "let rec x = fun y -> x 1;;" in
  let check_id_for_rec_decl _ = check_id_of_program rec_decl "x" in
  "test rec fun">:::
  ["check_id_for_let_rec_exp">:: check_id_for_let_rec_exp;
  "check_val_for_let_rec_exp">:: check_val_for_let_rec_exp;
  "check_id_for_rec_decl">:: check_id_for_rec_decl;]
;;
```

Ex4.1

```
Exercise 4.1 [必修課題]
図 11，図 12 に示すコードを参考にして，型推論アルゴリズムを完成させよ．
（ソースファイルとして typing.ml を追加するので，make depend の実行を一度行うこと．）
```

資料にあるコードを元に `typing.ml` を追加した後、 ConsやMultの型推論などが不十分なため修正した。

```
--- a/interpreter/typing.ml
+++ b/interpreter/typing.ml
@@ -11,8 +11,18 @@ let ty_prim op ty1 ty2 = match op with
     Plus -> (match ty1, ty2 with
         TyInt, TyInt -> TyInt
       | _ -> err ("Argument must be of integer: +"))
-  ...
-  | Cons -> err "Not Implemented!"
+  | Mult -> (match ty1, ty2 with
+        TyInt, TyInt -> TyInt
+      | _ -> err ("Argument must be of integer: *"))
+  | Lt -> (match ty1, ty2 with
+        TyInt, TyInt -> TyBool
+      | _ -> err ("Argument must be of integer: <"))
+  | And -> (match ty1, ty2 with
+        TyBool, TyBool -> TyBool
+      | _ -> err ("Argument must be of bool: &&"))
+  | Or -> (match ty1, ty2 with
+        TyBool, TyBool -> TyBool
```

```
+      | _ -> err ("Argument must be of bool: ||"))

 let rec ty_exp tyenv = function
     Var x ->
@@ -25,11 +35,18 @@ let rec ty_exp tyenv = function
     let tyarg2 = ty_exp tyenv exp2 in
       ty_prim op tyarg1 tyarg2
   | IfExp (exp1, exp2, exp3) ->
-    ...
+    let tycond = ty_exp tyenv exp1 in
+    let tythen = ty_exp tyenv exp2 in
+    let tyelse = ty_exp tyenv exp3 in
+      (match tycond with
+          TyBool -> if tythen = tyelse then tythen else err ("Type of then expre
+        | _ -> err ("Condition must be of bool"))
   | LetExp (id, exp1, exp2) ->
-    ...
+    let tyvalue = ty_exp tyenv exp1 in
+      ty_exp (Environment.extend id tyvalue tyenv) exp2
   | _ -> err ("Not Implemented!")

 let ty_decl tyenv = function
     Exp e -> ty_exp tyenv e
+  | Decl (id, e) -> ty_exp tyenv e
   | _ -> err ("Not Implemented!")
```

Ex4.2

```
Exercise 4.2 [必修課題]
図 13 中の pp_ty，freevar_ty を完成させよ．
freevar_ty は，与えられた型中の型変数の集合を返す関数で，型は
  val freevar_ty : ty -> tyvar MySet.t
とする．
型 'a MySet.t は（実験 WWW ページにある）mySet.mli で定義されている 'a を要素とす
る集合を表す型である．
```

pp_ty では、TyVarとTyFunのパターンを追加した。(TyVarでは 'a と、 TyFunでは ty1 -> ty2 と返す)

freevar_ty では TyVar var のパターンでは Myset.singleton var を、 TyFun (ty1, ty2) のパターンではty1とty2の型変数の集合を合成して返すようにした。

```
--- a/interpreter/syntax.ml
+++ b/interpreter/syntax.ml
@@ -19,10 +19,31 @@ type program =
   | Decl of id * exp
   | RecDecl of id * id * exp

-let pp_ty = function
```

```
+let rec pp_ty = function
      TyInt -> print_string "int"
    | TyBool -> print_string "bool"
+   | TyVar _ -> print_string "'a"
+   | TyFun (ty1, ty2) ->
+     pp_ty ty1;
+     print_string " -> ";
+     pp_ty ty2
+
+let rec freevar_ty ty = (match ty with
+     TyVar var -> MySet.singleton var
+   | TyFun (ty1, ty2) -> MySet.union (freevar_ty ty1) (freevar_ty ty2)
+   | _ -> MySet.empty)
```

Ex4.3

```
Exercise 4.3 [必修課題]
型代入に関する以下の型，関数を typing.ml 中に実装せよ．
  type subst = (tyvar * ty) list
  val subst_type : subst -> ty -> ty
型代入を表す型 subst は型変数と型のペアのリスト [(id1,ty1); ...; (idn,tyn)] で表現する．
このリストは [idn 7→ tyn] ◦ ・ ・ ・ ◦ [id1 7→ ty1] という型代入を表す．
順番が反転していること，また，代入の合成を表すので，ty1 に現れる型変数は後続のリストの表す型代入の影響を受けることに注意すること．
例えば，
  let alpha = fresh_tyvar () in
  subst_type [(alpha, TyInt)] (TyFun (TyVar alpha, TyBool))
の値は TyFun (TyInt, TyBool) になり，
  let alpha = fresh_tyvar () in
  let beta = fresh_tyvar () in
  subst_type [(beta, (TyFun (TyVar alpha, TyInt))); (alpha, TyBool)] (TyVar beta)
の値は TyFun (TyBool, TyInt) になる．
```

subst_typeでは末尾の型代入から、自身より前の型代入に適用していき、それにより新しく出来た型代入を型変数に適用すればよい。

```
type subst = (tyvar * ty) list

let rec subst_type s typ =
  let rec resolve_type s = function
      TyVar v -> (try List.assoc v s with Not_found -> TyVar v)
    | TyFun (ty1, ty2) -> TyFun (resolve_type s ty1, resolve_type s ty2)
    | a -> a in
  let rec resolve_subst = function
      [] -> []
    | (id, typ) :: rest -> let new_subst = resolve_subst rest in
        ((id, resolve_type new_subst typ) :: new_subst) in
  resolve_type (resolve_subst s) typ
```

Ex4.4

```
Exercise 4.4 [必修課題]
上の単一化アルゴリズムを
  val unify : (ty * ty) list -> subst
として実装せよ.
```

資料の単一化アルゴリズムを `(ty * ty)` の形式に当てはめていく

`U({(τ, τ )} ⊎ X)` は `TyInt, TyInt` や `TyBool, TyBool` のパターン、また同一変数のペアが当てはまる。

`U({(τ11 → τ12, τ21 → τ22)} ⊎ X)` は `TyFun (ty11, ty12), TyFun (ty21, ty22)` が当て

886 lines (745 sloc) | 30.3 KB

`U({(α, τ )} ⊎ X) (if τ ≠ α)` は `TyVar var1, TyVar var2` で、var1とvar2が異なる場合と、TyVarとそれ以外のペアが当てはまる。

これを元に実装した場合以下のようになった。

```
let rec unify = function
    [] -> []
  | (ty1, ty2) :: rest -> (match ty1, ty2 with
      TyInt, TyInt | TyBool, TyBool -> unify rest
    | TyFun (ty11, ty12), TyFun (ty21, ty22) -> unify ((ty12, ty22) :: (ty11, ty2
    | TyVar var1, TyVar var2 ->
      if var1 = var2 then unify rest
      else let eqs = [(var1, ty2)] in
        eqs @ (unify (subst_eqs eqs rest))
    | TyVar var, ty | ty, TyVar var ->
      if MySet.member var (Syntax.freevar_ty ty) then err ("type err")
      else let eqs = [(var, ty)] in
        eqs @ (unify (subst_eqs eqs rest))
    | _, _ -> err ("type err"))
```

Ex4.5

```
Exercise 4.5 [必修課題]
単一化アルゴリズムにおいて，α ∉ FTV(τ) という条件はなぜ必要か考察せよ.
```

aとτが異なり、FTV(τ)にαが含まれている場合としては例えば以下のような場合が考えられる

```
(TyVar a, TyFun (TyVar a, TyInt))
```

もし単一化アルゴリズムにおいて上記の条件がない場合、aの型が `TyFun (TyVar a, TyInt)` となるが、 aが自身の型を内包するため型が無限に展開される

そのため、上記の条件をつけることにより、自身の型を含み無限展開される再帰的な型を生成しないようにしていると考えられる

Ex4.6

```
Exercise 4.6 [必修課題]
他の型付け規則に関しても同様に型推論の手続きを与えよ(レポートの一部としてまとめよ).
そして，図 14 を参考にして，型推論アルゴリズムの実装を完成させよ.
```

Plus以外の型付け規則の型推論の手続きに関しては以下に述べる。

id (変数)

1. Γからidの型変数を探して返す

数値("-"? ['0'-'9']+)

1. 空の型代入 と int を出力として返す.

真偽値(true, false)

1. 空の型代入 と bool を出力として返す.

e1 * e2

1. Γ, e1 を入力として型推論を行い，S1, τ1 を得る.
2. Γ, e2 を入力として型推論を行い，S2, τ2 を得る.
3. 型代入 S1, S2 を α = τ という形の方程式の集まりとみなして， `S1 U S2 U {(τ1, int),(τ2, int)}` を単一化し，型代入 S3 を得る.
4. S3 と int を出力として返す.

e1 < e2

1. Γ, e1 を入力として型推論を行い，S1, τ1 を得る.
2. Γ, e2 を入力として型推論を行い，S2, τ2 を得る.
3. 型代入 S1, S2 を α = τ という形の方程式の集まりとみなして， `S1 U S2 U {(τ1, int),(τ2, int)}` を単一化し，型代入 S3 を得る.
4. S3 と bool を出力として返す.

e1 and e2

1. Γ, e1 を入力として型推論を行い，S1, τ1 を得る.

2. Γ, e2 を入力として型推論を行い，S2, τ2 を得る．
3. 型代入 S1, S2 を α = τ という形の方程式の集まりとみなして， `S1 U S2 U {(τ1, bool),(τ2, bool)}` を単一化し，型代入 S3 を得る．
4. S3 と bool を出力として返す．

e1 or e2

1. Γ, e1 を入力として型推論を行い，S1, τ1 を得る．
2. Γ, e2 を入力として型推論を行い，S2, τ2 を得る．
3. 型代入 S1, S2 を α = τ という形の方程式の集まりとみなして， `S1 U S2 U {(τ1, bool),(τ2, bool)}` を単一化し，型代入 S3 を得る．
4. S3 と bool を出力として返す．

if e1 then e2 else e3

1. Γ, e1 を入力として型推論を行い，S1, τ1 を得る．
2. Γ, e2 を入力として型推論を行い，S2, τ2 を得る．
3. Γ, e3 を入力として型推論を行い，S3, τ3 を得る．
4. 型代入 S1, S2, S3 を α = τ という形の方程式の集まりとみなして， `{(τ1, bool)} U S1 U S2 U S3 U {(τ2, τ3)}` を単一化し，型代入 S4 を得る．
5. S4 と τ2 を出力として返す．

let id = e1 in e2

1. Γ, e1 を入力として型推論を行い，S1, τ1 を得る．
2. Γをid, τ1で拡張しΓ2を得る
3. Γ2, e2 を入力として型推論を行い，S2, τ2 を得る．
4. 新しく型変数 a1 を確保する
5. 型代入 S1, S2 を α = τ という形の方程式の集まりとみなして， `S1 U {(a1, τ1)} U S2` を単一化し，型代入 S3 を得る．
6. S3 と τ2 を出力して返す.

fun id -> e1

1. 新しく型変数 domty を確保する
2. Γを id, domty で拡張し Γ2 を得る
3. Γ2, e1 を入力として型推論を行い，S1, ranty を得る．
4. S1 と domty から 型 a2 を得る
5. S1 と TyFun(a2, ranty) を出力して返す.

e1 e2 (関数適用)

1. Γ, e1 を入力として型推論を行い，S1, τ1 を得る．

2. Γ, e2 を入力として型推論を行い，S2, τ2 を得る.
3. 新しく型変数 domty を確保する
4. 型代入 S1, S2 を α = τ という形の方程式の集まりとみなして， `{(τ1, TyFun(τ2, domty))} U S1 U S2` を単一化し，型代入 S3 を得る.
5. S3 と domty を出力して返す.

型推論アルゴリズムの実装

上記の片付け規則を元に、図14から以下のような変更を加えた

```
--- a/interpreter/typing.ml
+++ b/interpreter/typing.ml
@@ -22,36 +22,37 @@ let rec subst_type s typ =

 (* eqs_of_subst : subst -> (ty * ty) list
 型代入を型の等式集合に変換 *)
-let eqs_of_subst s = (* XXX *)[]
+let rec eqs_of_subst s = match s with
+    [] -> []
+  | (tyvar, ty) :: rest -> (TyVar tyvar, ty) :: eqs_of_subst rest
+
+(* subst_eqs: subst -> (ty * ty) list -> (ty * ty) list
+型の等式集合に型代入を適用 *)
+let rec subst_eqs s eqs = match eqs with
+    [] -> []
+  | (ty1, ty2) :: rest -> (subst_type s ty1, subst_type s ty2) :: (subst_eqs s r

 let rec unify = function
     [] -> []
   | (ty1, ty2) :: rest -> (match ty1, ty2 with
-      TyInt, TyInt -> unify rest
-    | TyBool, TyBool -> unify rest
-    | TyFun (ty11, ty12), TyFun (ty21, ty22) -> unify ((ty11, ty12) :: (ty21, ty
-    | TyVar var1, TyVar var2 -> if var1 = var2 then unify rest else (unify rest)
-    | TyVar var, _ -> if MySet.member var (Syntax.freevar_ty ty2) then err ("typ
-        else (unify rest) @ [(var, ty2)]
-    | _, TyVar var -> if MySet.member var (Syntax.freevar_ty ty1) then err ("typ
-        else (unify rest) @ [(var, ty1)]
-    | _, _ -> unify rest)
+      TyInt, TyInt | TyBool, TyBool -> unify rest
+    | TyFun (ty11, ty12), TyFun (ty21, ty22) -> unify ((ty12, ty22) :: (ty11, ty
+    | TyVar var1, TyVar var2 ->
+      if var1 = var2 then unify rest
+      else let eqs = [(var1, ty2)] in
+        eqs @ (unify (subst_eqs eqs rest))
+    | TyVar var, ty | ty, TyVar var ->
+      if MySet.member var (Syntax.freevar_ty ty) then err ("type err")
+      else let eqs = [(var, ty)] in
+        eqs @ (unify (subst_eqs eqs rest))
+    | _, _ -> err ("type err"))
```

```
 let ty_prim op ty1 ty2 = match op with
     Plus -> ([(ty1, TyInt); (ty2, TyInt)], TyInt)
-  | Mult -> (match ty1, ty2 with
-        TyInt, TyInt -> TyInt
-      | _ -> err ("Argument must be of integer: *"))
-  | Lt -> (match ty1, ty2 with
-        TyInt, TyInt -> TyBool
-      | _ -> err ("Argument must be of integer: <"))
-  | And -> (match ty1, ty2 with
-        TyBool, TyBool -> TyBool
-      | _ -> err ("Argument must be of bool: &&"))
-  | Or -> (match ty1, ty2 with
-        TyBool, TyBool -> TyBool
-      | _ -> err ("Argument must be of bool: ||"))
-
+  | Mult -> ([(ty1, TyInt); (ty2, TyInt)], TyInt)
+  | Lt -> ([(ty1, TyInt); (ty2, TyInt)], TyBool)
+  | And -> ([(ty1, TyBool); (ty2, TyBool)], TyBool)
+  | Or -> ([(ty1, TyBool); (ty2, TyBool)], TyBool)

 let rec ty_exp tyenv = function
     Var x ->
@@ -66,21 +67,30 @@ let rec ty_exp tyenv = function
     let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) @ eqs3 in
     let s3 = unify eqs in (s3, subst_type s3 ty)
   | IfExp (exp1, exp2, exp3) ->
-    let tycond = ty_exp tyenv exp1 in
-    let tythen = ty_exp tyenv exp2 in
-    let tyelse = ty_exp tyenv exp3 in
-      (match tycond with
-          TyBool -> if tythen = tyelse then tythen else err ("Type of then expre
-        | _ -> err ("Condition must be of bool"))
+    let (scond, tycond) = ty_exp tyenv exp1 in
+    let (sthen, tythen) = ty_exp tyenv exp2 in
+    let (selse, tyelse) = ty_exp tyenv exp3 in
+    let eqs = [(tycond, TyBool)] @ (eqs_of_subst scond) @ (eqs_of_subst sthen) @
+      (eqs_of_subst selse) @ [(tythen, tyelse)] in
+    let s3 = unify eqs in (s3, subst_type s3 tythen)
   | LetExp (id, exp1, exp2) ->
-    let tyvalue = ty_exp tyenv exp1 in
-      ty_exp (Environment.extend id tyvalue tyenv) exp2
+    let (s1, ty1) = ty_exp tyenv exp1 in
+    let (s2, ty2) = ty_exp (Environment.extend id ty1 tyenv) exp2 in
+    let domty = TyVar (fresh_tyvar ()) in
+    let eqs3 = [(domty, ty1)] in
+    let eqs = (eqs_of_subst s1) @ eqs3 @ (eqs_of_subst s2) in
+    let s3 = unify eqs in (s3, subst_type s3 ty2)
   | FunExp (id, exp) ->
-    let domty = TryVar (fresh_tyvar ()) in
+    let domty = TyVar (fresh_tyvar ()) in
     let s, ranty =
       ty_exp (Environment.extend id domty tyenv) exp in
       (s, TyFun (subst_type s domty, ranty))
-  | AppExp (exp1, exp2) -> (* XXX *) ([], TyInt)
+  | AppExp (exp1, exp2) ->
```

```
+    let (s1, ty1) = ty_exp tyenv exp1 in
+    let (s2, ty2) = ty_exp tyenv exp2 in
+    let domty = TyVar (fresh_tyvar ()) in
+    let eqs = (ty1, TyFun(ty2, domty)) :: (eqs_of_subst s1) @ (eqs_of_subst s2)
+    let s3 = unify eqs in (s3, subst_type s3 domty)
   | _ -> err ("Not Implemented!")
```

また、型推論の結果が正しく出力されるように `syntax.ml` にも変更を加えた

```
--- a/interpreter/syntax.ml
+++ b/interpreter/syntax.ml
@@ -27,14 +27,28 @@ type ty =
   | TyVar of tyvar
   | TyFun of ty * ty

-let rec pp_ty = function
-    TyInt -> print_string "int"
-  | TyBool -> print_string "bool"
-  | TyVar _ -> print_string "'a"
-  | TyFun (ty1, ty2) ->
-    pp_ty ty1;
-    print_string " -> ";
-    pp_ty ty2
+let string_of_ty ty =
+  let var_list = ref [] in
+  let var_id =
+    let body tyvar =
+      let rec index_of counter = function
+          [] -> var_list := !var_list @ [tyvar]; counter
+        | x :: rest -> if x = tyvar then counter else index_of (counter + 1) res
+      in index_of 0 !var_list
+    in body in
+  let rec to_string = function
+      TyInt -> "int"
+    | TyBool -> "bool"
+    | TyVar tyvar -> Printf.sprintf "'%c" (char_of_int ((int_of_char 'a') + (var
+    | TyFun (ty1, ty2) ->
+      let string_ty1 = (match ty1 with
+          TyFun (_, _) -> "(" ^ to_string ty1 ^ ")"
+        | _ -> to_string ty1) in
+      let string_ty2 = to_string ty2 in
+        string_ty1 ^ " -> " ^ string_ty2
+  in to_string ty
+
+let pp_ty ty = print_string (string_of_ty ty)
```

これにより、型推論のテストケースは全て通るようになっている.

また、型推論のテストケースは `test/interpreter.ml` に全て載せてある.