

レポート2（インタプリタ）

Name: Pyii Phyo Maung

Student ID: 1029322149

July 23, 2023

1 Exercises Done

ex3_2_1
ex3_2_2
ex3_2_3
ex3_2_4
ex3_3_1
ex3_3_4
ex3_4_1
ex3_4_2
ex3_4_3
ex3_4_4
ex3_4_5
ex3_4_6
ex3_5_1

2 Details of Exercises

Exercise 3.2.1 [必修]

MiniML1 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`などが正しく評価されるかを試してみよ．

Exercise 3.2.2 [**]

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，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改造せよ．

Exercise 3.2.3 [*]

論理値演算のための二項演算子 `&&`, `||` を追加せよ．

Exercise 3.2.4 [**]

`lexer.mll`を改造し，`(*と*)`で囲まれたコメントを読み飛ばすようにせよ．なお，OCamlのコメントは入れ子にできることに注意せよ．`ocamllex` のド

コメントを読む必要があるかもしれない。(ヒント1: `(*と*)`が正しく入れ子になっている語の集合は正則言語ではないので, 正則表現を工夫するだけで頑張るのは無理.) (ヒント2: `comment`という再帰的なルールを`lexer.mll`に新しく定義するとよい.)

Exercise 3.3.1 [必修]

MiniML1 インタプリタを拡張して, MiniML2 インタプリタを作成し, テストせよ.

Exercise 3.3.4 [**]

`and`を使って変数を同時にふたつ以上宣言できるように `let`式・宣言を拡張せよ. 例えば以下のプログラム

```
let x = 100
and y = x in x+y
```

の実行結果は 200 ではなく, (`x`が大域環境で 10に束縛されているので) 110 である.

Exercise 3.4.1 [必修]

MiniML3 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.

Exercise 3.4.2 [**]

OCaml での「(中置演算子)」記法をサポートし, プリミティブ演算を通常関数と同様に扱えるようにせよ. 例えば

```
let threetimes = fun f -> fun x -> f (f x x) (f x x) in
  threetimes (+) 5
```

は, 20を出力する.

Exercise 3.4.3 [*]

OCaml の

```
fun x1 ... xn -> ...
let f x1 ... xn = ...
```

といった簡略記法をサポートせよ.

Exercise 3.4.4 [*]

以下は、加算を繰り返して 4 による掛け算を実現している MiniML3 プログラムである。これを改造して、階乗を計算するプログラムを書け。

```
let makemult = fun maker -> fun x ->
    if x < 1 then 0 else 4 + maker maker (x + -1) in
let times4 = fun x -> makemult makemult x in
    times4 3
```

Exercise 3.4.5 [*]

静的束縛とは対照的な概念として 動的束縛 (dynamic binding) がある。動的束縛の下では、関数本体は、関数式を評価した時点ではなく、関数呼び出しがあった時点での環境をパラメータ・実引数で拡張した環境下で評価される。インタプリタを改造し、fun の代わりに dfun を使った関数は動的束縛を行うようにせよ。例えば、

```
let a = 3 in
let p = dfun x -> x + a in
let a = 5 in
    a * p 2
```

というプログラムでは、関数 p 本体中の a は 3 ではなく 5 に束縛され、結果は、35になる。(fun を使った場合は 25 になる。)

Exercise 3.4.6 [*]

動的束縛の下では、MiniML4 で導入するような再帰定義を実現するための特別な仕組みや、このExerciseのようなトリックを使うことなく、再帰関数を定義できる。以下のプログラムで、二箇所の fun を dfun (このExerciseを参照)に置き換えて(4通りのプログラムを)実行し、その結果について説明せよ。

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

Exercise 3.5.1 [必修]

図に示した syntax.ml にしたがって、parser.mly と lexer.mll を完成させ、MiniML4 インタプリタを作成し、テストせよ。(let rec式だけでなくlet rec宣言も実装すること。)

3 Explanations of Exercises

3.1 3.2

1. Additions made

```
with
err ->
  let e = Printexc.to_string err in (*get error message*)
  print_string ("Error: " ^ e ^ "\n");
  read_eval_print env

  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))))))
```

2. Thought of trying to add GT > but after checking the test files, it wasn't necessary. This is what it would look like.

```
eval.ml
let rec apply_prim op arg1 arg2 = match op, arg1, arg2 with
...
| Gt, IntV i1, IntV i2 -> BoolV (i1 > i2)
| Gt, _, _ -> err ("Both arguments must be integer: >")
...
```

```
lexer.mll
| ">" { Parser.GT }
```

```
parser.mly
%token GT
...
CompExpr :
```

```

    l=AddExpr LT r=AddExpr { BinOp (Lt, l, r) }
  | l=AddExpr GT r=AddExpr { BinOp (Gt, l, r) }
  | e=AddExpr { e }

```

```

syntax.ml
type binOp = Plus | Mult | Lt | Gt

```

3. I also added the necessary implementations for logical AND and OR for the program.

```

syntax.ml

type binOp = Plus | Mult | Lt | And | Or

let var_of_binop = function
  Plus -> Var "+"
  | Mult -> Var "*"
  | Lt -> Var "<"
  | And -> Var "&&"
  | Or -> Var "||"

let id_of_binop = function
  Var "+" -> "+"
  | Var "*" -> "*"
  | Var "<" -> "<"
  | Var "&&" -> "&&"
  | Var "||" -> "||"
  | _ -> "Not Implemented!"

parser.mly

%token PLUS MULT LT AND OR
ORExpr :
  l=ANDExpr OR r=ORExpr { BinOp (Or, l, r) }
  | e=ANDExpr { e }

```

```

ANDEExpr :
  l=LTEExpr AND r=ANDEExpr { BinOp (And, l, r) }
| e=LTEExpr { e }

eval.ml

| Or, BoolV i1, BoolV i2 -> BoolV (i1 || i2) (* Logical or *)
| Or, _, _ -> err ("Both arguments must be boolean: ||")
| And, BoolV i1, BoolV i2 -> BoolV (i1 && i2) (* Logical and *)
| And, _, _ -> err ("Both arguments must be boolean: &&")

```

3.2 Explanation of the rest of the files

I have added comments to all of the files that are changed in src on GitHub so please refer to the files as well.

Exercise 3.3.1 [必修]

MiniML1 インタプリタを拡張して, MiniML2 インタプリタを作成し, テストせよ.

The necessary changes have been added according to the instructions and it has been implemented.

Exercise 3.3.4 [**]

andを使って変数を同時にふたつ以上宣言できるように let式・宣言を拡張せよ. 例えば以下のプログラム

```

let x = 100
and y = x in x+y

```

の実行結果は 200 ではなく, (xが大域環境で 10に束縛されているので) 110 である.

The necessary changes with implemented but from changing this, 3.3.2 stopped working as well.

Exercise 3.4.1 [必修]

MiniML3 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.

The necessary changes have been added according to the instructions and it has been implemented.

Exercise 3.4.2 [**]

OCaml での「(中置演算子)」記法をサポートし、プリミティブ演算を通常の関数と同様に扱えるようにせよ。例えば

```
let threetimes = fun f -> fun x -> f (f x x) (f x x) in
  threetimes (+) 5
```

は、20を出力する。

This has been implemented. Please check the comments of the code on GitHub for the explanation.

Exercise 3.4.3 [*]

OCaml の

```
fun x1 ... xn -> ...
let f x1 ... xn = ...
```

といった簡略記法をサポートせよ。

This has been implemented. Please check the comments of the code on GitHub for the explanation.

Exercise 3.4.4 [*]

以下は、加算を繰り返して 4 による掛け算を実現している MiniML3 プログラムである。これを改造して、階乗を計算するプログラムを書け。

```
let makemult = fun maker -> fun x ->
  if x < 1 then 0 else 4 + maker maker (x + -1) in
let times4 = fun x -> makemult makemult x in
  times4 3
```

This has been implemented. Please check the comments of the code on GitHub for the explanation.

Exercise 3.4.5 [*]

静的束縛とは対照的な概念として 動的束縛 (dynamic binding) がある。動的束縛の下では、関数本体は、関数式を評価した時点ではなく、関数呼び出しがあった時点での環境をパラメータ・実引数で拡張した環境下で評価される。インタプリタを改造し、fun の代わりに dfun を使った関数は動的束縛を行うようにせよ。例えば、

```
let a = 3 in
let p = dfun x -> x + a in
```



```
let a = 5 in
  a * p 2
```

というプログラムでは、関数 `p` 本体中の `a` は 3 ではなく 5 に束縛され、結果は、35になる。(fun を使った場合は 25 になる。)

This has been implemented. Please check the comments of the code on GitHub for the explanation.

Exercise 3.4.6 [*]

動的束縛の下では、MiniML4 で導入するような再帰定義を実現するための特別な仕組みや、このExerciseのようなトリックを使うことなく、再帰関数を定義できる。以下のプログラムで、二箇所の `fun` を `dfun` (このExerciseを参照)に置き換えて(4通りのプログラムを)実行し、その結果について説明せよ。

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

This has been implemented. Please check the comments of the code on GitHub for the explanation.

Exercise 3.5.1 [必修]

図に示した `syntax.ml` にしたがって、`parser.mly` と `lexer.mll` を完成させ、MiniML4 インタプリタを作成し、テストせよ。(let rec式だけでなくlet rec宣言も実装すること。)

The necessary changes have been added according to the instructions and it has been implemented.

4 Sample Test case for all the files

```
open OUnit
open EvalTestGenerator
open Miniml.Eval

let dataset_for_eval = [
  (* Test for Exercise 3.2.1 *)
  { input = "let ii = 2 and iii = 3 and iv = 4 in iv + iii * ii;;"; expected = IntV 10 };

  (* Test for Exercise 3.2.3 *)
  { input = "true && false;;"; expected = BoolV false };
```

```

{ input = "true || false;;"; expected = BoolV true };

(* Test for Exercise 3.3.4 *)
{ input = "let x = 100 and y = x in x + y;;"; expected = IntV 110 };

(* Test for Exercise 3.4.1 *)
{ input = "let f = fun x -> x * x in f 5;;"; expected = IntV 25 };

(* Test for Exercise 3.4.2 *)
{ input = "let threetimes = fun f -> fun x -> f (f x x) (f x x) in threetimes (fun x y ->

(* Test for Exercise 3.4.3 *)
{ input = "let f x y = x + y in f 3 4;;"; expected = IntV 7 };

(* Test for Exercise 3.4.4 *)
{ input = "let makemult = fun maker -> fun x -> if x < 1 then 0 else 4 + maker maker (x +

(* Test for Exercise 3.4.5 *)
{ input = "let a = 3 in let p = dfun x -> x + a in let a = 5 in a * p 2;;"; expected = Int

(* Test for Exercise 3.4.6 *)
{ input = "let fact = dfun n -> n + 1 in let fact = dfun n -> if n < 1 then 1 else n * fa

(* Test for Exercise 3.5.1 *)
{ input = "let rec f x = if x < 1 then 1 else x * f (x + -1) in f 5;;"; expected = IntV 12
];;

let dataset_for_evalerror = [
  (* Test for Exercise 3.2.2 *)
  { input = "let f x y z -> 10 in f 0 0 0;;" };
  { input = "fun x y z = 10;;" };
  { input = "let f x y z -> 10;;" };
];;

let () = ignore(run_test_tt_main (
  "ex3.4.3" >:::
  gen_eval_tests dataset_for_eval

```

```
@ gen_evalerror_tests dataset_for_evalerror
))
```