課題2レポート

芦田聖太

提出日　17/07/06

Ex3.2

Ex3.2.1

設計方針

main.ml で定義されている、大域変数の部分にローマ数字を追加する。

実装

設計方針で述べたように、main.ml の大域変数に i〜v を初期値として定めた。以下に実際に実装したコード掲載しておく。

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

Environment の extend を使って、束縛変数と変数の情報を追加していった。変数を指定したので、実際に動きを確認する。i〜v の数字を用いる演算を入力した。

実行結果

```
#iv + v * ii + i + iii;;
val - = 18
```

以上の結果より、大域変数が正常な値を示していることが確認できた。

Ex3.2.2

設計方針

main.ml の read_eval_print の部分で error をキャッチできるようにする。

実装説明

lexer、Parser、eval で発生するエラーを補足できるように以下で示す変更を行った。

```
let rec read_eval_print env =e
  print_string "#␣";
  flush stdout;
  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 Error s -> print_string (s ^ "\n") ; read_eval_print env
      | Parser.Error -> print_string("Parser␣Error" ^ "\n") ;
          read_eval_print env
      | Failure s -> print_string( s^"\n") ; read_eval_print env
```

```
Parser.toplevel Lexer.main (Lexing.from_channel stdin)
```

の部分で発生するエラーを let decl 以降の部分を try with で囲うことで補足した。各エラーの原因は次のように対応している。

- Error は eval でのエラー
- Parser.Error は Parser でのエラー
- Failure は lexer でのエラー

各エラーの処理後、read_eval_print を再度行う。この処理を行うことで、エラーが起きても入力待ちの状態に戻すことができる。

Ex.3.2.3

設計方針

Syntax の Binop に And と Or を追加し、&&と||を認識させるために lexer にトークンとして“&&”と“||”を追加した。さらに、Parser では And と Or の木を追加して、eval では And と Or の評価部分を追加した。

実装説明

まず、Syntax と lexer の変更点を説明する。

Syntax

```
type id = string

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

~~~~~~~~
```

lexer の main の部分

```
~~~~

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

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

| "(" { Parser.LPAREN }
| ")" { Parser.RPAREN }
| ";;" { Parser.SEMISEMI }
| "+" { Parser.PLUS }
| "*" { Parser.MULT }
| "<" { Parser.LT }
| "&&" { Parser.AND }
| "||" { Parser.OR }

~~~~
```

Syntax では、binOp に And と Or が追加されているのがわかる。また、lexer では、main に“&&”と“||”が追加されている。
次に、Parser の変更点についてみる。

Parser 変更点の抜粋

```
~~~~~

%{
open Syntax
%}

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

~~~~~~

toplevel :
    e=Expr SEMISEMI { Exp e }

Expr :
    e=IfExpr { e }
  | e=ORExpr { e }

ORExpr :
    l=ORExpr OR r=ANDExpr {BinOp (Or, l, r)}
  | e=ANDExpr {e}

ANDExpr :
    l=ANDExpr AND r=LTExpr {BinOp (And, l, r)}
  | e=LTExpr {e}

LTExpr :
```

```
    l=PExpr LT r=PExpr { BinOp (Lt, l, r) }
  | e=PExpr { e }

PExpr :
    l=PExpr PLUS r=MExpr { BinOp (Plus, l, r) }
  | e=MExpr { e }

~~~~~~~
```

token に AND と OR が追加されている。また、ORExpr と ANDExpr という部分が追加されているのがわかる。これは、AND と OR の判定部分である。AND は OR よりも先に計算され、整数の大小評価は AND よりも先に計算されるものなので、ORExpr、ANDExpr、LTExpr という順番で定義してある。

最後に eval の変更点を確認する。
eval の変更点

```
~~~~~~~

let rec apply_prim op arg1 arg2 = match op, arg1, arg2 with
    Plus, IntV i1, IntV i2 -> IntV (i1 + i2)
  | Plus, _, _ -> err ("Both␣arguments␣must␣be␣integer:␣+")
  | Mult, IntV i1, IntV i2 -> IntV (i1 * i2)
  | Mult, _, _ -> err ("Both␣arguments␣must␣be␣integer:␣*")
  | Lt, IntV i1, IntV i2 -> BoolV (i1 < i2)
  | Lt, _, _ -> err ("Both␣arguments␣must␣be␣integer:␣<")
  | And, BoolV i1, BoolV i2 -> BoolV (i1 && i2)
  | And, _, _ -> err ("Both␣arguments␣must␣be␣boolean:␣&&")
  | Or, BoolV i1, BoolV i2 -> BoolV (i1 || i2)
  | Or, _, _ -> err ("Both␣arguments␣must␣be␣boolean:␣||")

~~~~~~~
let rec eval_exp env = function
.......
  | BinOp (op, exp1, exp2) ->
    let arg1 = eval_exp env exp1 in
    (match op with
        And -> (match arg1 with
                    BoolV false -> BoolV false
                  | _ -> let arg2 = eval_exp env exp2 in
                    apply_prim op arg1 arg2)
      | Or -> (match arg1 with
                    BoolV true -> BoolV true
                  | _ -> let arg2 = eval_exp env exp2 in
                    apply_prim op arg1 arg2)
      | _ -> let arg2 = eval_exp env exp2 in apply_prim op arg1
        arg2)
 ...........
```

変更しているのは apply_prim の部分である。AND と OR はそれぞれ、2 つの Bool V に対して、AND と OR の計算を行っている。また、BinOp の部分で exp2 の値に関係なく true になる場合と false になる場合の対処を行なっている。

以上の変更を行うことで、AND と OR 演算を実現した。

Ex.3.2.4

設計方針

コメントを無視できるように、lexer の main に変更を加える。さらに入れ子のコメントにも対応できるように新たなルールを追加する。

実装説明

lexer.mll の変更部分

```
~~~~~~~

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

| "(*"  { comment lexbuf; main lexbuf }

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

| "(" { Parser.LPAREN }

~~~~~~~

and comment = parse
  _ { comment lexbuf }
| "*)" { () }
| "(*" { comment lexbuf; comment lexbuf }

~~~~~~~
```

まず、main の部分では (*の入力に対して、comment lexbuf というアクションを行ったあと、main lexbuf を行うようにしてある。comment lexbuf 起こすことで、新しく追加したルール comment を適用し操作完了後、main lexbuf でルール main に戻るようになっている。ルール comment の内容は以下のようなものである。

- “(*”と“*)”以外の入力に対して、トークンを返さずルール comment に戻る。
- “(*”の入力に対して、ルール comment を行い、終了後もう一度ルール comment を行う。
- “*)”の入力に対して、何も返さない。

3 つ目のルールでルール適用が完了すると、呼び出し元の comment lexbuf が完了し次のアクションを行う。一番外側のコメントであれば次のアクションは main lexbuf となり、コメントが閉じられて再びプログラムを読み込むようになる。入れ子となっているコメントであれば、次のアクションは commnet lexbuf となり依然としてコメントのルールが適用されることとなる。

Ex3.3

Ex3.3.1

設計方針

以下に変更を加えるプログラムと変更について記す。

- syntax.ml の exp に Letexp、program に Decl をそれぞれ追加。
- lexer.mll の reservedWords に let と in を追加。さらに main にも“=”を token として追加。
- Parser.mly は token,toplevel,Expr に変更を加え、LetExpr を追加。
- eval には let 宣言の評価部分と let 式の評価部分を加える。

これらの変更によって、let 宣言と let 式を実現する。詳しい動作の説明は実装説明で行う。

実装説明

まず、Syntax の変更について見てみる。追加とコメントが入れられている部分が追加部分となる。syntax.ml

```
~~~~~~~

type exp =
    Var of id
  | ILit of int
  | BLit of bool
  | BinOp of binOp * exp * exp
  | IfExp of exp * exp * exp
  | Letexp of id * exp * exp (* 追加 *)

~~~~~~~

type program =
    Exp of exp
    | Decl of id * exp (*追加*)

~~~~~~~
```

Letexp は id と 2 つの exp を保持する。id は束縛したい変数、1 つ目の exp が id に束縛したい式で、2 つ目の exp はデータが束縛された id を用いて評価したい式となる。
Decl は id と exp を保持する。id が束縛したい変数で exp が束縛したい式となる。

2 つ目は lexer の変更点について確認する。

```
let reservedWords = [
  (* Keywords *)
.........
  ("in", Parser.IN);
  ("let", Parser.LET)
]
}

rule main = parse
  (* ignore spacing and newline characters *)
............

| "=" { Parser.EQ }
............
```

Keyword として in と let を追加し、rule main に=を追加。この処理を行うことで let を token LET、in を token IN、=を token EQ として出力するようになる。

3 つ目に Parser.mly の変更点について確認する。
Parser.mly

```
toplevel :
    e=Expr SEMISEMI { Exp e }
  | LET x = ID EQ e = Expr SEMISEMI { Decl (x, e) }

Expr :
    e=IfExpr { e }
  | e=LetExpr { e }
  | e=ORExpr { e }

~~~~~~~~~

LetExpr :
    LET x=ID EQ e1=Expr IN e2=Expr { Letexp (x, e1, e2) }
```

toplevel,Expr にそれぞれ変更を加え。LetExpr を追加。
toplevel の変更により let 宣言の構文木が、LetExpr を追加することで let 式の構文木がそれぞれ認識可能となる。返す値はそれぞれ、let 宣言は Decl、let 式は Exp LetExp となる。

最後に、eval.ml についての変更点を確認する。
eval.ml

```
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

let eval_decl env = function
    Exp e -> let v = eval_exp env e in ("-", env, v, Nothing)
  | Decl (id, e) ->
          let v = eval_exp env e in (id, Environment.extend id v
               env, v, Nothing)
```

Letexp では、eval_env exp1 で exp1 を評価し、value に束縛。id と value の関係を environment に追加した環境で exp2 を評価する。eval_decl では Decl(id, e) の場合の操作を追加する。e を評価し、id と e の評価値の関係を環境に追加。id と新たな環境、e の評価値の組を返す。Nothing は Ex3.3.2 のための拡張なので、今は説明しない。

Ex3.3.2

設計方針

- syntax.ml の program に DeclDecl と Nothing を追加。
- Parser.mly は toplevel への追加と LetLetExpr を追加する。
- eval.ml には eval_Decl の部分に DeclDecl の評価を追加。
- main.ml には read_eval_print に DeclDecl の処理を追加する。

実装説明

syntax.ml

```
~~~~~~~
type program =
    Exp of exp
    | Decl of id * exp
    | DeclDecl of id * exp * program
    | Nothing
```

DeclDecl は id と exp と program を保持する。id に exp を束縛する。program は次以降の let 宣言の情報が含まれている program である。Nothing は評価の時に必要となる。最右端の Let 宣言は、最 DeclDecl の program の部分に Nothing が格納される。

Parser.mly

```
~~~~~~~
toplevel :
    e=Expr SEMISEMI { Exp e }
  | LET x = ID EQ e = Expr SEMISEMI { Decl (x, e) }
  | LET x = ID EQ e1 = Expr e2 = LetLetExpr SEMISEMI { DeclDecl
      (x, e1, e2) }

LetLetExpr :
    LET x = ID EQ e = Expr { Decl (x, e) }
  | LET x = ID EQ e1 = Expr e2 = LetLetExpr { DeclDecl (x, e1,
      e2) }
~~~~~~~
```

連続する let 宣言に対応するために、toplevel に 3 つ目の式を追加した。e2 は LetLetExpr になり、program を返すようにしてある。LetLetExpr では、2 つの式が定義されている。1 つ目の式は連なる let 宣言の最右端の let 宣言を把握するようになっており、2 つ目の式は 3 つ以上 Let 宣言が連なる時の間の Let 宣言を把握するようになっている。

eval.ml

```
~~~~~~~~
let eval_decl env = function
    Exp e -> let v = eval_exp env e in ("-", env, v, Nothing)
  | Decl (id, e) ->
          let v = eval_exp env e in (id, Environment.extend id v
               env, v, Nothing)
  | DeclDecl (id, e1, e2) ->
          let v = eval_exp env e1 in (id, Environment.extend id
              v env, v, e2)
```

返り値を 4 種類のデータの組に変えた。これは let 宣言が続いた場合、DeclDecl の e2 を取り出すために必要であったからである。取り出した e2 に関する処理は main の変更で詳しく触れる。したがって、4 つ目のデータは DeclDecl 以外の program には関係ないので Nothing としておく。DeclDecl の場合は e1 を評価して、id と評価値を拡張した新しい環境、評価値、そして e2 にある program を返す。

main.ml

```
let rec checker x y = match y with
                    DeclDecl(x1, e1, e2) -> if x = x1 then true
                        else checker x e2
                  | Decl(x1, e) -> if x = x1 then true else
                      false

let rec read_eval_print env =
..........
  let decl = Parser.toplevel Lexer.main (Lexing.from_channel
      stdin) in
  let rec repeat env x =
    let (id, newenv, v, expr) = eval_decl env x in
```

```
    match expr with
            Nothing -> Printf.printf "val␣%s␣=␣" id;
                       pp_val v;
                       print_newline(); read_eval_print newenv
          | Decl( x, e ) -> if checker id expr
                            then
                            repeat newenv expr
                            else
                            Printf.printf "val␣%s␣=␣" id; pp_val
                                 v;
                            print_newline();
                            repeat newenv expr
          | DeclDecl( x, e1, e2 ) -> if checker id expr
                                     then
                                     repeat newenv expr
                                     else
                                     Printf.printf "val␣%s␣=␣" id
                                         ; pp_val v;
                                     print_newline();
                                     repeat newenv expr
  in repeat env decl
~~~~~~~
```

repeat は環境と program を引数にとる。program が Exp か単体の let 宣言、もしくは連続する let 宣言の最右端の場合 expr が Nothing になる。その場合は今までの処理と同じ処理を行う。連続する let 宣言の途中の部分がくるとき expr は Decl もしくは DeclDecl となる。その時は、新しい環境と expr を引数にして、repeat を行う。このようにすることで、連続する let 宣言を次々に評価していくことが可能となった。連なった let 宣言の途中で、同じ変数を何度も束縛する際に最後の束縛のみを表記するために、checker を定義した。checker では、DeclDecl に含まれている program を再帰的に調べていき、同じ変数を発見すると true を返す。checker が true のときは関数の束縛を表示しない。

Ex3.4

Ex3.4.1

設計方針

- syntax.ml の exp に関数の定義部分と 関数適用の部分を追加する。
- Parser.mly も同様に関数の定義部分と 関数適用の部分を追加する。また、fun と矢印を定義する。
- lexer.mll は fun と矢印を定義する。
- eval.ml には exval に関数のデータ型を追加。また、eval_exp に関数の定義の処理と関数適用の処理を追加。

実装

syntax.ml

```
~~~~
type exp =
    ........
  | FunExp of id * exp
  | AppExp of exp * exp

~~~~
```

syntax の exp に関数 FunExp と関数適用 AppExp を追加。FunExp は引数と関数 exp を保持する。また、AppExp は関数 exp と適用される exp を保持する。

Parser.mly

```
~~~~~~~
Expr :
    e=IfExpr { e }
  | e=LetExpr { e }
  | e=LTExpr { e }
  | e=FunExpr { e }

............

MExpr :
    e1=MExpr MULT e2=AExpr { BinOp (Mult, e1, e2) }
  | e=AppExpr { e }

 AppExpr :
    e1=AppExpr e2=AExpr { AppExp (e1, e2) }
  | e=AExpr { e }

AExpr :
    i=INTV { ILit i }
  | TRUE   { BLit true }
  | FALSE  { BLit false }
  | i=ID   { Var i }
  | LPAREN e=Expr RPAREN { e }

.........

FunExpr :
    FUN x = ID RARROW e = Expr { FunExp (x, e) }
```

Expr に FunExpr を追加。ME の下で AE の上に AppExpr を配置する。関数は Let 宣言や他の計算式と同レベルに配置される。また、関数適用は整数、変数や Bool 値などの値の次に計算される。

lexer.mll

```
~~~~
let reservedWords = [
  ......
  ("fun", Parser.FUN)
```

```
]

rule main = parse
 ............
| "->" { Parser.RARROW }

~~~~
```

reservedWords に“=”を追加。main に矢印マークを追加。以上の追加によって Parser が=と矢印をマッチングできるようになる。

eval.ml

```
~~~~
type exval =
    ..........
  | ProcV of id * exp * dnval Environment.t

~~~~
let rec eval_exp env = function
    ..............
  | FunExp (id, exp) -> ProcV (id, exp, 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
        eval_exp newenv body
       | _ -> err ("Non-function␣value␣is␣applied"))
~~~~
```

exval で ProcV という関数のためのデータを定義。ProcV は引数と関数の内容、そして関数定義のための変数情報を含んだ環境を保持する。評価する Expr が FunExp だったとき、id と exp、さらに環境を組にして ProcV を返す。また、関数適用のときは、まず exp1 が関数かどうかをチェックする。関数であれば exp2 の評価結果を引数に束縛し環境を拡張し。新たな環境で、関数の内容を評価する。

テスト

高階関数を定義できるかを確認する。以下は確認した結果である。

```
# let f = fun x -> x + 1;;
val f = function
# let c = fun g -> g 4;;
val c = function
# c f;;
val - = 5
```

c は引数に関数をとることができている。したがって高階関数が定義できることが確認できた。

Ex3.5

Ex3.5.1

設計方針

- syntax.ml の exp に LetRecExp を追加。さらに program に RecDecl も追加する。
- lexer.mll の reservedWords に rec を追加。
- parser.mly の toplevel に let rec 宣言を追加。Expr に let rec 式を追加。さらに tokenREC を追加。
- eval.ml の

実装

syntax.ml

```
~~~~
type exp =
    ........
  | LetRecExp of id * id * exp * exp (*追加*)

type program =
    .........
    | RecDecl of id * id * exp (*追加*)
```

exp に LetRexExp、program に RecDecl をそれぞれ追加。RecDecl は let rec 宣言のときに必要なもので、束縛される変数と引数、関数の本体を保持する。LetRecDecl は let rec 式のときに必要となる。変数と再帰の引数、さらに関数本体と最後の評価式を保持する。

lexer.mll

```
let reservedWords = [
  ........
  ("rec", Parser.REC);
]
```

reservedWords に rec を追加する。追加によって Parser が rec をマッチングできるようになる。

parser.mly

```
~~~~
%token REC

~~~~
toplevel :
```

```
 ............
  | LET REC x=ID n=ID EQ e=Expr SEMISEMI { RecDecl (x, n, e) }
 .........

Expr :
  ............
  | e=LetRecExpr { e }
  ............

~~~~~~~
LetRecExpr :
    LET REC x=ID n=ID EQ e1=Expr IN e2=Expr { LetRecExp (x, n,
        e1, e2) }
```

token に REC を追加。toplevel に let rec 宣言を追加。返り値は RecDecl とする。Expr に LetRecExpr を追加。返り値は LetRecExp とする。保持する値は syntax の部分で表記した通りである。

eval.ml

```
~~~~
let rec eval_exp env = function
   ............
  | 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
    ............
  | RecDecl (id1, id2, e) -> (*追加*)
        let dummyenv = ref Environment.empty in
        let y = ProcV (id2, e, dummyenv) in
        let newenv =
          Environment.extend id1 y env in
          dummyenv := newenv ; print_newline() ; (id1, newenv, y
              )
```

eval_exp に LetRecExp の評価を追加する。ダミー環境への参照を作り、その参照を含めた新たな環境を構築し、最後に参照を新たな環境に付け替える。RecDecl に関する評価も付け加える。方法は LetRecExp と同じ。

テスト

fact を実行した。

```
# let rec fact = fun n -> if n = 0 then 1 else n * fact (n - 1)
    ;;
val fact = <fun>
# fact 5;;
val - = 120
```

以上より正常に動いていることがわかった。

感想

全体として難しい印象を感じた。comment の課題と let をいくつも定義する部分ではかなり苦戦した。それ以外の部分では、ほとんどテキストを読めば理解ができた。ただ、Parser と lexer の構文を理解するのに少し時間がかかった。任意課題は余力があれば取り組みたい。