ソフトウェアサイエンス実験 S8 最終レポート

200911434 青木大祐

2012 年 12 月 21 日

1 概要

本実験では、関数プログラミング言語 Ocaml を用いて、Ocaml 自身のサブセットである「ミニ Ocaml 言語 (以下「ミニ Ocaml」)」のインタプリタおよびコンパイラを作成した。

2 Ocaml について

まず最初に、実験に用いる Ocaml について理解を深めるため、関数プログラミングの技法を学んだ。具体的には、再帰やパターンマッチを用いて Euclid の互除法から最大公約数を求める関数や、Fibonacci 数を求める関数を記述した。例を以下に示す。

```
let rec gcd (x, y) =
  if x <= 0 || y <= 0 then gcd (abs x, abs y)
  else if x = y then x
  else if x > y then gcd(x-y, y)
  else gcd(x, y-x);;

let rec fib n =
  match n with
      1 -> 1
    | 2 -> 1
    | n -> fib(n - 2) + fib(n - 1);;
```

3 ミニ Ocaml 言語

前章の内容を終えた後、本題である「ミニ Ocaml インタプリタの実装」に着手した。本実験で扱うミニ Ocaml の構文は以下のとおり[*1]である。

```
e ::= x                    変数
  | let x = e in e         let式
  | let rec f x = e in e   let-rec式
  | fun x -> e             関数
  | e e                    関数適用
  | true                   真理値リテラル(定数)
  | false                  真理値リテラル(定数)
  | if e then e else e     if式
  | n                      自然数リテラル (n は自然数 0, 1, 2, ...)
  | - e                    整数演算(符号の反転)
  | e + e                  整数演算(足し算)
  | e * e                  整数演算(かけ算)
  | e / e                  整数演算(割り算)
  | e = e                  等しさ(整数と真理値)
  | e > e                  大小比較(整数)
  | e < e                  大小比較(整数)
```

このミニ Ocaml を実装するにあたって、入力されるプログラムを扱う内部的な表現として**式** (*exp*) という構造を考える。プログラムは式からなり、また式は一つないしは複数の部分的な式から成り立っている。この再帰的な構造を表現するため、Ocaml のヴァリアント型を用いて代数的データ構造を実現することにする。

上記のミニ Ocaml の仕様を満たすために、次のような式の種類を *Syntax.ml* として用いる。なお、これは講義ページで配布されたものをそのまま利用するため、上記の仕様を含む、より多種の式の定義となっている。

```
type exp =
  | Var of string          (* variable e.g. x *)
  | IntLit of int          (* integer literal e.g. 17 *)
  | BoolLit of bool        (* boolean literal e.g. true *)
  | If of exp * exp * exp (* if e then e else e *)
  | Let of string * exp * exp   (* let x=e in e *)
  | LetRec of string * string * exp * exp   (* letrec f x=e in e *)
  | Fun of string * exp    (* fun x -> e *)
  | App of exp * exp       (* function application i.e. e e *)
  | Eq of exp * exp        (* e = e *)
  | Noteq of exp * exp (* e <> e *)
  | Greater of exp * exp   (* e > e *)
  | Less of exp * exp      (* e < e *)
  | Plus of exp * exp      (* e + e *)
  | Minus of exp * exp     (* e - e *)
  | Times of exp * exp     (* e * e *)
  | Div of exp * exp       (* e / e *)
  | Empty                  (* [ ] *)
  | Match of exp * ((exp * exp) list)     (* match e with e->e | ... *)
  | Cons of exp * exp      (* e :: e *)
```

[*1] 実験テキスト (http://logic.cs.tsukuba.ac.jp/ kam/jikken/mini.html) より引用

```
  | Head of exp           (* List.hd e *)
  | Tail of exp           (* List.tl e *)
```

これらを踏まえて、与えられた式を評価して、最終的な値を計算するインタプリタを実装する。

3.1 eval 関数

式を受け取り、式の種類によって違う処理を行うため、Ocaml のパターンマッチングを用いて *eval* 関数を実装する。整数値の足し算と掛け算を評価する *eval*1 を示す。

```
let rec eval1 e =
  match e with
  | IntLit(n)    -> n
  | Plus(e1,e2)  -> (eval1 e1) + (eval1 e2)
  | Times(e1,e2) -> (eval1 e1) * (eval1 e2)
  | _ -> failwith "unknown expression"
```

もし与えられた式が**整数リテラル** (*IntLit*) だった場合は、その数値が式を評価した値となる。また、与えられた式が足し算 (*Plus*) だった場合は、2 つの引数として与えられた式を再帰的に評価し、その 2 値を合計し、評価した値として返す。掛け算についても同様である。
もし exp 型の値であるが $IntLit, Plus, Times$ でないものが渡された場合、最後のパターン (_) にマッチし、*unknown expression* という例外が投げられる。

この *eval*1 に、入力として $2*2+(3+(-4))$ という式を与えてみる。*exp* 型の表現では $Plus(Times(IntLit\ 2, IntLit\ 2), Plus(IntLit\ 3, IntLit\ (-$
である。実行結果を以下に示す。

```
val easy : exp =
  Plus (Times (IntLit 2, IntLit 2), Plus (IntLit 3, IntLit (-4)))
- : int = 3
```

正しく計算できていることが分かる。

3.2 値と型

*eval*1 では評価した値は Ocaml の *integer* 型であるが、ミニ Ocaml のデータ型としてはまず *Int* 型と *Bool* 型を扱いたいため、このままでは不十分である。複数のデータ型を扱えるようにするため、**値** (*value*) というデータ構造を導入する。

```
type value =
  | IntVal  of int        (* integer value e.g. 17 *)
  | BoolVal of bool       (* booleanvalue e.g. true *)
```

式を評価すると値が返る。整数型が求められる式上の位置に真偽値型がある場合は型エラーであり、またその逆も同じくエラーである。例えば、条件節の真偽によって実行する処理を分岐する If 文では、条件節は真偽値型でなければならない。
以下に、*eval*1 の式に加えて真偽値によって分岐する If、式 1 と式 2 が同じ値なら真を返す Eq、式 1 が式 2 よりも大きい整数であれば真を返す *Greater* を実装した *eval*2 を以下に示す。

```
let rec eval2 e =
  match e with
  | IntLit(n)  -> IntVal(n)
  | Plus(e1,e2) ->
      begin
        match (eval2 e1, eval2 e2) with
    | (IntVal(n1),IntVal(n2)) -> IntVal(n1+n2)
          | _ -> failwith "integer values expected"
      end
  | Times(e1,e2) ->
      begin
        match (eval2 e1, eval2 e2) with
    | (IntVal(n1),IntVal(n2)) -> IntVal(n1*n2)
          | _ -> failwith "integer values expected"
      end
  | Eq(e1,e2) ->
      begin
  match (eval2 e1, eval2 e2) with
    | (IntVal(n1),IntVal(n2)) -> BoolVal(n1=n2)
    | (BoolVal(b1),BoolVal(b2)) -> BoolVal(b1=b2)
    | _ -> failwith "wrong value"
      end
  | BoolLit(b) -> BoolVal(b)
  | If(e1,e2,e3) ->
      begin
  match (eval2 e1) with
    | BoolVal(true) -> eval2 e2
```

```
    | BoolVal(false) -> eval2 e3
    | _ -> failwith "wrong value"
      end
  | Greater (e1, e2) ->
      begin
        match (eval2 e1, eval2 e2) with
          | (IntVal(n1), IntVal(n2)) -> BoolVal(n1 > n2)
          | _ -> failwith "wrong value"
      end
  | _ -> failwith "unknown expression e";;
```

このままでは $Plus$ や $Times$ の定義の大部分が重複しており、また後から二項の整数演算を追加する際に煩雑になってしまう。これを回避するため、実際の計算を $binop$ 関数に切り分けることにする。$binop$ 関数の定義は以下のとおり。

```
let binop f e1 e2 =
  match (eval2b e1, eval2b e2) with
  | (IntVal(n1),IntVal(n2)) -> IntVal(f n1 n2)
  | _ -> failwith "integer values expected"
```

これに伴って、$Plus, Times$ の定義を以下のように変更する。

```
...
    | Plus(e1,e2)  -> binop (+) e1 e2
    | Times(e1,e2) -> binop ( * ) e1 e2
...
```

3.3 変数と環境

ミニ Ocaml において**変数**を実現するため、let 文の導入を試みる。変数を束ねておく**環境**として、名前と値のタプルをリストにした $(string * value)\ list$ を考える。変数を宣言した際にはこの環境に名前と値を追加し、式の中に変数が出現した際にはこの環境から名前を探しだして値を取り出すことが出来る。

3.3.1 変数の定義・参照

環境に変数を追加する操作として、環境の拡張 ext という関数を実装する。この関数は、引数に変数名と値、そして現在の環境を受け取り、先頭に追加したものを返す。

```
let ext env x v = (x,v) :: env
```

また、現在の環境から変数を参照する関数として $lookup$ を実装する。これは変数名と現在の環境を受け取り、対応する値を返す。

```
let rec lookup x env =
   match env with
   | [] -> failwith ("unbound variable: " ^ x)
   | (y,v)::tl -> if x=y then v
                  else lookup x tl
```

想定どおりの環境が作られているか、以下のようなテストプログラムを実行して確認してみる。なお、$emptyenv$ は空の環境を返す関数である。

```
let env = emptyenv();;
let env = ext env "x" (IntVal 1);;
let env = ext env "y" (BoolVal true);;
let env = ext env "z" (IntVal 5);;

(* 実行結果 *)
val env : 'a list = []
val env : (string * value) list = [("x", IntVal 1)]
val env : (string * value) list = [("y", BoolVal true); ("x", IntVal 1)]
val env : (string * value) list =
  [("z", IntVal 5); ("y", BoolVal true); ("x", IntVal 1)]
```

正しく環境が作られていることが分かったので、変数名を参照してみる。

```
let y = lookup "y" env;; (*存在する変数*)
let w = lookup "w" env;; (*存在しない変数*)

(*実行結果*)
val y : value = BoolVal true
Exception: Failure "unbound variable: w".
```

存在する変数は正しく参照できており、また未定義の変数は例外になっている。

なお、この環境は定義される度に先頭に追加されていく。$lookup$ は先頭から順に変数名の探索を行うため、同名の変数が存在する場合は最新の物が参照されるため、変数名の上書きが可能になる。

3.3.2 ミニ Ocaml への組み込み

実装した $ext, lookup$ をミニ Ocaml インタプリタに組みこむことを考える。
まず、動作の理解を深めるため、環境の変化を順に表示するデバッグ関数 $string\_of\_env, print\_env$ を実装した。

```
let string_of_env env =
  let string_of_val (e:value) : string =
    match e with
      | IntVal n -> string_of_int n
      | BoolVal true ->  "true"
      | BoolVal false ->  "false"
  in
  let rec internal (env:(string * value) list) =
    match env with
      | [] -> ""
      | (name, v)::rest -> name ^ " = " ^ (string_of_val v) ^ ";" ^ (internal rest)
  in
  "[" ^ (internal env) ^ "]";;

let print_env (env: (string * value) list) =
  print_string( string_of_env env);;
```

これを踏まえて、変数を定義する式 Let と変数を参照する式 Var を追加した $eval3$ を以下に示す。デバッグモードを有効にするには、オプショナル引数として $\sim mode : 1$ を指定すると、環境の変化が逐一出力される。

```
let rec eval3 ?(mode=0) e env =           (* env を引数に追加 *)
  if mode != 0 then print_env env;

  let binop f e1 e2 env =       (* binop の中でも eval3 を呼ぶので env を追加 *)
    match (eval3 e1 env ~mode, eval3 e2 env ~mode) with
    | (IntVal(n1),IntVal(n2)) -> IntVal(f n1 n2)
    | _ -> failwith "integer value expected"
  in
  match e with
  | Var(x)       -> lookup x env
  | IntLit(n)    -> IntVal(n)
  | BoolLit(b) -> BoolVal(b)
  | Plus(e1,e2)  -> binop (+) e1 e2 env     (* env を追加 *)
  | Times(e1,e2) -> binop ( * ) e1 e2 env   (* env を追加 *)
  | Eq(e1,e2) ->
      begin
  match (eval3 e1 env ~mode, eval3 e2 env ~mode) with
    | (IntVal(n1),IntVal(n2)) -> BoolVal(n1=n2)
    | (BoolVal(b1),BoolVal(b2)) -> BoolVal(b1=b2)
    | _ -> failwith "wrong value"
      end
  | If(e1,e2,e3) ->
      begin
        match (eval3 e1 env ~mode) with          (* env を追加 *)
          | BoolVal(true)  -> eval3 e2 env   (* env を追加 *)
          | BoolVal(false) -> eval3 e3 env   (* env を追加 *)
          | _ -> failwith "wrong value"
      end
  | Let(x,e1,e2) ->
      let env1 = ext env x (eval3 e1 env ~mode)
      in eval3 e2 env1 ~mode
  | _ -> failwith "unknown expression";;
```

テストとして、幾つかの入力を与えてみる。

```
(* let x = 1 in 2 + x *)
eval3 (Let ("x", IntLit 1, (Plus (IntLit 2, Var "x")))) [];;

(* let x = 1 in let y = 3 in x + y *)
eval3 (Let ("x", IntLit 1, Let("y", IntLit 3, (Plus (Var "y", Var "x"))))) [];;

(* let x = true in if x = true then 1 else 2 *)
eval3 (Let ("x", BoolLit true, If(Eq(Var "x", BoolLit true), IntLit 1, IntLit 2))) [];;

(* 実行結果 *)
- : value = IntVal 3
- : value = IntVal 4
- : value = IntVal 1
```

また、デバッグモードを用いて環境の遷移を表示してみる。

```
(* let x = 1 in let y = x + 1 in x + y *)
eval3 ~mode:1 (Let ("x", IntLit 1, (Let ("y", Plus(Var "x", IntLit 1), Plus(Var "x", Var "y"))))) [];;

[][][x = 1;][x = 1;][x = 1;][x = 1;][y = 2;x = 1;][y = 2;x = 1;][y = 2;x = 1;]
- : value = IntVal 3
```

まず最初に x が定義され、次に y が定義されていく過程がわかる。

4 構文解析

今までは *exp* 型の式を自分で作成していたが、*ocamllex* と *ocamlyacc* を用いて、ミニ Ocaml 言語の構文を解析して *exp* 型の式を作り出すパーザを生成する。parser と lexer は実験にあたって与えられた *parser.mly* と *lexer.mll* を用いる。また、実行ファイルの生成にあたって必要となる他のファイル (*main.ml*, *syntax.ml*) も、与えられたものを用いることにする。

4.1 構文の拡張

lexer と parser を拡張して、新たに演算子を定義してみる。例として、二数が等しくないと真を返す演算子 <> を定義する。

lexer.mll に新しいトークンとして <> を *NOTEQUAL* という名前で追加する。

```
 ...

  (* 演算子 *)
  | '+'       { PLUS }
  | '-'       { MINUS }
  | '*'       { ASTERISK }
  | '/'       { SLASH }
  | '='       { EQUAL }
  | '<'       { LESS }
  | '>'       { GREATER }
  | "::"      { COLCOL }
  | "<>"      { NOTEQUAL }

...
```

また、*parser.mly* を以下のように変更し、*NOTEQUAL* が出現した場合の処理として *Noteq* という *exp* を出力するように定義する。

```
...

// 演算子
%token PLUS     // '+'
%token MINUS    // '-'
%token ASTERISK // '*'
%token SLASH    // '/'
%token EQUAL    // '='
%token LESS     // '<'
%token GREATER  // '>'
%token COLCOL   // "::"
%token NOTEQUAL // "<>"

...

%left EQUAL GREATER LESS NOTEQUAL

...

  // e1 <> e2
  | exp NOTEQUAL exp
    { Noteq ($1, $3) }

...
```

最後に、*syntax.ml* に *Noteq* という *exp* を追加する。

```
...

(* 式の型 *)
type exp =
  | Var of string          (* variable e.g. x *)
  | IntLit of int          (* integer literal e.g. 17 *)
  | BoolLit of bool        (* boolean literal e.g. true *)
  | If of exp * exp * exp (* if e then e else e *)
  | Let of string * exp * exp   (* let x=e in e *)
  | LetRec of string * string * exp * exp   (* letrec f x=e in e *)
  | Fun of string * exp    (* fun x -> e *)
  | App of exp * exp       (* function application i.e. e e *)
  | Eq of exp * exp        (* e = e *)
  | Noteq of exp * exp (* e <> e *)
  | Greater of exp * exp   (* e > e *)
  | Less of exp * exp      (* e < e *)
  | Plus of exp * exp      (* e + e *)
  | Minus of exp * exp     (* e - e *)
  | Times of exp * exp     (* e * e *)
  | Div of exp * exp       (* e / e *)
  | Empty                  (* [ ] *)
  | Match of exp * ((exp * exp) list)    (* match e with e->e | ... *)
  | Cons of exp * exp      (* e :: e *)
  | Head of exp            (* List.hd e *)
```

```
  | Tail of exp           (* List.tl e *)

...
```

これらを踏まえて、*eval* に *Noteq* の処理を記述する。

```
  | Noteq(e1, e2) ->
      begin
  match (eval3 e1 env ~mode, eval3 e2 env ~mode) with
    | (IntVal(n1),IntVal(n2)) -> BoolVal(n1!=n2)
    | (BoolVal(b1),BoolVal(b2)) -> BoolVal(b1!=b2)
    | _ -> failwith "wrong value"
      end
```

以下のような入力を与えて動作を確認してみる。

```
run "true <> false";;
run "(1+1) <> 2";;
run "let x = 1 in let y = 2 in x <> y";;

- : Syntax.value = Syntax.BoolVal true
- : Syntax.value = Syntax.BoolVal false
- : Syntax.value = Syntax.BoolVal true
```

正しく動いていることが分かる。

4.2 関数と束縛

ミニ Ocaml において関数を実現するために、関数型という値を導入する。関数は静的束縛を実現するため、名前と関数本体に加えて関数抽象が行われた時点での環境を同時に保持する。

これに伴って、*value* 型の定義を以下のように拡張する。

```
type value =
  | IntVal  of int
  | BoolVal of bool
  | FunVal  of string * exp * ((string * value) list)
```

また、関数抽象を行う式 *Fun* と関数適用を行う式 *App* の処理を以下のように定義する。

```
...

  | Fun(x,e1) -> FunVal(x, e1, env)
  | App(e1,e2) ->
     begin
      match (eval4 e1 env) with
        | FunVal(x,body,env1) ->
            let arg = (eval4 e2 env)
      in eval4 body (ext env1 x arg)
        | _ -> failwith "function value expected"
     end

...
```

4.3 再帰関数

関数の再帰を実現するため、「再帰しない関数」とは別に再帰関数という型を用意する。型の定義は以下のとおり。

```
 | RecFunVal  of string * string * exp * ((string * value) list)
```

再帰関数を定義する式 *LetRec* と、関数適用 *App* に再帰関数を受け取った場合の処理を追加し、*eval*6 を作成する。これで、ミニ Ocaml インタプリタの基本的な機能の実装は終了とする。

以下に、完成した *eval*6 を含むミニ Ocaml インタプリタの全体を示す。

```
open Syntax;;

let emptyenv () = [];;

let ext env x v = (x,v) :: env;;

let rec lookup x env =
   match env with
   | [] -> failwith ("unbound variable: " ^ x)
   | (y,v)::tl -> if x=y then v
                  else lookup x tl;;
```

```
12
13 let string_of_env env =
14   let string_of_val e =
15     match e with
16       | IntVal n -> string_of_int n
17       | BoolVal true ->  "true"
18       | BoolVal false ->  "false"
19   in
20   let rec internal env =
21     match env with
22       | [] -> ""
23       | (name, v)::rest -> name ^ " = " ^ (string_of_val v) ^ ";" ^ (internal rest)
24   in
25   "[" ^ (internal env) ^ "]";;
26
27 (* eval3 : exp -> (string * value) list -> value *)
28 (* let と変数、環境の導入 *)
29 let print_env env =
30   print_string(string_of_env env);;
31
32 let rec eval6 e env =           (* env を引数に追加 *)
33   let binop f e1 e2 env =       (* binop の中でも eval6 を呼ぶので env を追加 *)
34     match (eval6 e1 env , eval6 e2 env ) with
35     | (IntVal(n1),IntVal(n2)) -> IntVal(f n1 n2)
36     | _ -> failwith "integer value expected"
37   in
38   match e with
39   | Var(x)       -> lookup x env
40   | IntLit(n)    -> IntVal(n)
41   | BoolLit(b) -> BoolVal(b)
42   | Plus(e1,e2)  -> binop (+) e1 e2 env      (* env を追加 *)
43   | Times(e1,e2) -> binop ( * ) e1 e2 env    (* env を追加 *)
44   | Minus(e1, e2) -> binop (-) e1 e2 env
45   | Eq(e1,e2) ->
46       begin
47   match (eval6 e1 env , eval6 e2 env ) with
48     | (IntVal(n1),IntVal(n2)) -> BoolVal(n1=n2)
49     | (BoolVal(b1),BoolVal(b2)) -> BoolVal(b1=b2)
50     | _ -> failwith "wrong value"
51       end
52   | If(e1,e2,e3) ->
53       begin
54         match (eval6 e1 env ) with          (* env を追加 *)
55           | BoolVal(true)  -> eval6 e2 env   (* env を追加 *)
56           | BoolVal(false) -> eval6 e3 env   (* env を追加 *)
57           | _ -> failwith "wrong value"
58       end
59   | Let(x,e1,e2) ->
60       let env1 = ext env x (eval6 e1 env )
61       in eval6 e2 env1
62  | LetRec(f,x,e1,e2) ->
63       let env1 = ext env f (RecFunVal (f, x, e1, env))
64       in eval6 e2 env1
65   | Noteq(e1, e2) ->
66       begin
67   match (eval6 e1 env , eval6 e2 env ) with
68     | (IntVal(n1),IntVal(n2)) -> BoolVal(n1!=n2)
69     | (BoolVal(b1),BoolVal(b2)) -> BoolVal(b1!=b2)
70     | _ -> failwith "wrong value"
71       end
72   | Fun(x,e1) -> FunVal(x, e1, env)
73   | App(e1,e2) ->
74       let funpart = (eval6 e1 env) in
75       let arg = (eval6 e2 env) in
76         begin
77          match funpart with
78          | FunVal(x,body,env1) ->
79             let env2 = (ext env1 x arg) in
80             eval6 body env2
81          | RecFunVal(f,x,body,env1) ->
82             let env2 = (ext (ext env1 x arg) f funpart) in
83             eval6 body env2
84          | _ -> failwith "wrong value in App"
85         end
86   | Greater(e1, e2) ->
87       begin
88         match (eval6 e1 env , eval6 e2 env ) with
89           | (IntVal(n1), IntVal(n2)) -> BoolVal(n1 > n2)
90           | _ -> failwith "wrong value"
91       end
92   | _ -> failwith "unknown expression";;
```

動作確認として、自然数 x について階乗を求める再帰関数 $fact$ をミニ Ocaml で実装し、実行した。結果は以下のとおり。

```
1 let run src =
2   Eval.eval6 (Main.parse(src)) (Eval.emptyenv())
3 ;;
4
5 run "let rec fact x = if x = 0 then 1 else x * fact (x-1) in fact 10";;
```

```
1 # val run : string -> Syntax.value = <fun>
```

```
- : Syntax.value = Syntax.IntVal 3628800
```

また別の例として、正しくない式の評価を考える。定義されていない演算子や、式中の位置に相応しくない型のデータが出現した場合は、エラーとして失敗する必要がある。

```
run "1++1";;
Exception: Failure "parse error near characters 2-3".

run "1+true";;
Exception: Failure "integer value expected".

run "if 1 then 1 else 0";;
Exception: Failure "wrong value".
```

不正な式は正しく弾くことができている。

4.4 機能拡張

作成したインタプリタの機能拡張として、リスト型の実装を行った。具体的な操作としては「リストを生成する ($Cons$)」、「リストの先頭の要素を返す ($Head$)」、「リストの先頭を除いた残りを返す ($Tail$)」の 3 つである。また、リスト同士の同値性を比較するため、Eq に関しても比較の操作を追加した。

以下に $eval$ 関数に追加したパターンを示す。

```
  | Eq(e1,e2) ->
      begin
  match (eval6 e1 env , eval6 e2 env ) with
    | (IntVal(n1),IntVal(n2)) -> BoolVal(n1=n2)
    | (BoolVal(b1),BoolVal(b2)) -> BoolVal(b1=b2)
    | (ListVal(l1),ListVal(l2)) -> BoolVal(l1=l2)
    | _ -> failwith "wrong value"
      end
...
  | Cons(e1,e2) ->
     begin
      match (eval6 e1 env, eval6 e2 env) with
      | (v1,ListVal(v2)) -> ListVal(v1 :: v2)
     end
  | Head e1 ->
      begin
        match (eval6 e1 env) with
          |  ListVal(v1::_) -> v1
          | _ -> failwith "argument is not a list."
      end
  | Tail e1 ->
      begin
        match (eval6 e1 env) with
          | (ListVal (_::v1)) -> ListVal v1
          | _ -> failwith "argument is not a list."
      end
```

例として、以下の様な入力を与え、実行結果を確認する。

```
run "1::2::[]";;
run "true::false::[]";;
run "List.hd (1::2::3::[])";;
run "List.tl (1::2::3::[])";;
run "((3::true::1::[])::((2::1::[])::(1::2::3::[])::[]))";;
```

```
- : Syntax.value = ListVal [IntVal 1; IntVal 2]
- : Syntax.value = ListVal [BoolVal true; BoolVal false]
- : Syntax.value = IntVal 1
- : Syntax.value = ListVal [IntVal 2; IntVal 3]
- : Syntax.value =
ListVal
 [ListVal [IntVal 3; BoolVal true; IntVal 1]; ListVal [IntVal 2; IntVal 1];
  ListVal [IntVal 1; IntVal 2; IntVal 3]]
```

この例では正しく動作している。しかし、このリストは Ocaml のリストとは異なり、複数の型の値を要素として含むことが出来る。以下は、Ocaml のリストとして正しくない例であるが、現在のミニ Ocaml の実装では正しいリストとして解釈してしまう。

```
run "1::true::[]";;
run "((3::true::1::[])::((true::false::(1::[])::[])::(1::2::3::[])::[]))";;
```

```
- : Syntax.value = ListVal [IntVal 1; BoolVal true]
- : Syntax.value =
ListVal
 [ListVal [IntVal 3; BoolVal true; IntVal 1];
  ListVal [BoolVal true; BoolVal false; ListVal [IntVal 1]];
```

```
  ListVal [IntVal 1; IntVal 2; IntVal 3]]
#
```

これを防ぐため、$Cons$ に以下のような修正を行った。このパターンは、リストの生成時に、連結されるリストの先頭の要素が、連結する要素と同じ型であることを強制している。ただし、連結されるリストが空リストである場合は検査の必要がないため、そのまま連結している。

```
  | Cons(e1,e2) ->
     begin
      match (eval6 e1 env, eval6 e2 env) with
        | (v1, ListVal([])) -> ListVal([v1])
        | (IntVal(v1),ListVal(IntVal(v2)::v3)) -> ListVal(IntVal(v1)::(IntVal(v2)::v3))
        | (BoolVal(v1),ListVal(BoolVal(v2)::v3)) -> ListVal(BoolVal(v1)::(BoolVal(v2)::v3))
        | (ListVal(v1),ListVal(ListVal(v2)::v3)) -> ListVal(ListVal(v1)::(ListVal(v2)::v3))
        | _ -> failwith "mismatch type of elements";;
     end
```

これにより、$[IntVal; BooVal]$ や $[IntVal; ListVal]$ のようなリストは生成できなくなった。しかしリストをネストした場合、リストの型を取得する方法がわからなかったため、この型検査が有効なのは単一の値を要素とするリストについてのみである。

以下に、修正後の実行結果を示す。

```
run "((3::2::1::[])::((true::false::[])::(1::2::3::[])::[]))";;
run "1::true::[]";;
```

```
- : Syntax.value =
ListVal
 [ListVal [IntVal 3; IntVal 2; IntVal 1];
  ListVal [BoolVal true; BoolVal false];
  ListVal [IntVal 1; IntVal 2; IntVal 3]]
Exception: Failure "mismatch type of elements".
```

これを踏まえて、整数リストの要素の総和を求める sum 関数を実装する。

```
run "let rec sum x = if x = [] then 0 else (List.hd x) + sum(List.tl x) in sum (1::2::3::[])";;
```

実行結果は以下のとおり。

```
- : Syntax.value = IntVal 6
```

正しく計算できていることが分かる。

5 ミニ Ocaml コンパイラの作成

ここまで作成したミニ Ocaml インタプリタをもとに、仮想機械 ASEC の命令列を出力するコンパイラを作成する。

5.1 ASEC 機械

ASEC 機械とは、$Accumulator, Stack, Environment, Code$ の 4 部分で構成される仮想機械である。資料として提示された ASEC 仮想機械の遷移表と命令列の変換規則にしたがって、今まで製作したインタプリタをもとにコンパイラを作成した。

以下に、コンパイラとして製作した $eval$ 関数を含む $eval.ml$ と、課題として ASEC 機械に機能追加を行った $am.ml$ を示す。

ソースコード 1　eval.ml

```
open Syntax;;
open Am;;

let emptyenv () = [];;

let rec position (x : string) (venv : string list) : int =
  match venv with
    | [] -> failwith "no matching variable in environment"
    | y::venv2 -> if x=y then 0 else (position x venv2) + 1;;

let rec eval e env =           (* env を引数に追加 *)
  let binop f e1 e2 env =       (* binop の中でも eval を呼ぶので env を追加 *)
    (eval e2 env) @ [I_Push] @ (eval e1 env) @ [f]
  in
  match e with
  | Var(x)       -> [I_Search (position x env)]
  | IntLit(n)    -> [I_Ldi n]
```

```
  | BoolLit(b) -> [I_Ldb b]
  | Plus(e1,e2)  -> binop I_Add e1 e2 env
  | Minus(e1,e2) -> binop I_Sub e1 e2 env
  | Times(e1,e2) -> binop I_Mult e1 e2 env
  | Div(e1,e2) -> binop I_Div e1 e2 env
  | Eq(e1,e2) -> binop I_Eq e1 e2 env
  | Noteq(e1,e2) -> binop I_Noteq e1 e2 env
  | If(e1,e2,e3) -> (eval e1 env) @ [I_Test ((eval e2 env), (eval e3 env))]
  | Let(x,e1,e2) ->
      let env1 = x::env in
      [I_Pushenv] @ (eval e1 env) @ [I_Extend] @ (eval e2 env1) @ [I_Popenv]
  | LetRec(f,x,e1,e2) ->
     let env1 = f::env in
      [I_Pushenv] @ [I_Mkclos (eval e1 (x::env1))] @ [I_Extend] @ (eval e2 env1) @ [I_Popenv]
  | Fun(x,e1) -> [I_Mkclos (eval e1 (x::("_"::env)))]
  | App(e1,e2) -> [I_Pushenv] @ (eval e2 env) @ [I_Push] @ (eval e1 env) @ [I_Apply; I_Popenv]
  | Greater(e1, e2) -> binop I_Greater e1 e2 env
  | _ -> failwith "unknown expression";;
```

ソースコード 2　am.ml(抜粋)

```
...
  instr =
  | I_Ldi of int            (* I_Ldi(n) は、整数nを Accumulator に置く (loadする) *)
  | I_Ldb of bool           (* I_Ldb(b) は、真理値bを Accumulator に置く (loadする) *)
  | I_Push                  (* Accumulatorにある値をスタックに積む *)
  | I_Extend                (* Accumulatorにある値を環境に積む (環境を拡張する) *)
  | I_Search of int         (* I_Search(i) は、環境の i 番目の値を取ってきて Accumulatorに置く *)
  | I_Pushenv               (* 現在の環境を、スタックに積む *)
  | I_Popenv                (* スタックのトップにある環境を、現在の環境とする *)
  | I_Mkclos of code        (* I_Mkclos(c) は、関数本体のコードが c で
           * ある関数クロージャを生成し、Accumulatorに置く。
           * なお、関数の引数は、変数名を除去しているので、
           * このクロージャに含まれない。関数本体で
           * は「環境の中の0番目の変数として指定される。
           * ここでは、すべての関数を再帰関数として処理している。
           *)
  | I_Apply                 (* Accumulator の値が関数クロージャである
           * とき、その関数を、スタックトップにある
           * 値に適用した計算を行なう。
           *)
  | I_Test of code * code   (* I_Test(c1,c2)は、Accumulatorにある値が
           * true ならば、コードc1 を実行し、false
           * ならばコード c2 を実行する。
           *)
  | I_Add                   (* Accumulatorにある値にスタックトップの値
           * を加えて結果をAccumulator にいれる
           *)
  | I_Sub                   (* Accumulatorにある値にスタックトップの値
                               を引いて結果をAccumulatorにいれる *)
  | I_Mult                  (* Accumulatorにある値にスタックトップの値
                               を掛けて結果をAccumulatorにいれる *)
  | I_Div                   (* Accumulatorにある値にスタックトップの値
                               で割って結果をAccumulatorにいれる *)
  | I_Greater               (* Accumulatorにある値がスタックトップの値
                               より大きいかどうかを比較して、結果をAccumulatorにいれる *)
  | I_Eq                    (* Accumulatorにある値とスタックトップの値
           * が同じ整数であるかどうかをテストして、結
           * 果をAccumulator にいれる
           *)
  | I_Noteq                  (* Accumulatorにある値とスタックトップの値
           * が違う整数であるかどうかをテストして、結
           * 果をAccumulator にいれる
           *)
...

(* ASEC machine の状態遷移(計算)*)
let rec trans (a:am_value) (s:am_stack) (e:am_env) (c:code) : am_value =
  let binop (a:am_value) (s:am_stack) (f:instr) : (am_value * am_stack) =
    begin
      match (f,      a,            s               ) with
  |   (I_Add,  AM_IntVal(n), AM_IntVal(m)::s1) -> (AM_IntVal (n+m), s1)
  |   (I_Sub,  AM_IntVal(n), AM_IntVal(m)::s1) -> (AM_IntVal (n-m), s1)
  |   (I_Mult,  AM_IntVal(n), AM_IntVal(m)::s1) -> (AM_IntVal (n*m), s1)
  |   (I_Div,  AM_IntVal(n), AM_IntVal(m)::s1) -> (AM_IntVal (n/m), s1)
  |   (I_Greater,  AM_IntVal(n), AM_IntVal(m)::s1) -> (AM_BoolVal (n>m), s1)
  |   (I_Eq ,  AM_IntVal(n), AM_IntVal(m)::s1) -> (AM_BoolVal(n=m), s1)
  |   (I_Noteq ,  AM_IntVal(n), AM_IntVal(m)::s1) -> (AM_BoolVal(n!=m), s1)
  | _ -> failwith "unexpected type of argument(s) for binary operation"
    end
  in
  match (a,s,e,c) with
    | (_,[],[],[]) -> a   (* コードが空の時、Accumulatorの値が最終結果 *)
    | (_,_,_,[]) -> failwith "non-empty stack or environment"
  (* 計算が終わった時、スタックと環境は空であるべき。 *)
    | (_,_,_,I_Mkclos(c2)::c1) -> trans (AM_Closure(c2,e)) s e c1
    | (_,_,_,I_Push      ::c1) -> trans a (a::s) e c1
    | (_,_,_,I_Extend    ::c1) -> trans a s (a::e) c1
    | (_,_,_,I_Search(n) ::c1) -> trans (List.nth e n) s e c1
    | (_,_,_,I_Pushenv   ::c1) -> trans a (AM_Env(e)::s) e c1
    | (_,AM_Env(e1)::s1,_,I_Popenv::c1) -> trans a s1 e1 c1
    | (_,_::_          ,_,I_Popenv::c1) -> failwith "non-environment on the stack top"
    | (_,_             ,_,I_Popenv::c1) -> failwith "empty stack for Pop"
```

```
    | (AM_Closure(c2,e1),v::s1,_,I_Apply::c1) -> trans a s1 (v::a::e1) (c2@c1)
    | (_                ,_::_ ,_,I_Apply::c1) -> failwith "closure expected in accumulator for Apply"
    | (_                ,_    ,_,I_Apply::c1) -> failwith "empty stack for Apply"
    | (_,_,_,I_Ldi(n)::c1) -> trans (AM_IntVal(n))  s e c1
    | (_,_,_,I_Ldb(b)::c1) -> trans (AM_BoolVal(b)) s e c1
    | (AM_BoolVal(true), _,_,I_Test(c2,_ )::c1) -> trans a s e (c2 @ c1)
    | (AM_BoolVal(false),_,_,I_Test(_ ,c3)::c1) -> trans a s e (c3 @ c1)
    | (_,                _,_,I_Test(_ ,_ )::_ ) -> failwith "boolean expected for if-test"
    | (_,_,_,i       ::c1)
  when List.mem i [I_Add; I_Sub; I_Mult; I_Div; I_Greater; I_Eq; I_Noteq]
  ->
  let (v,s1) = binop a s i
  in trans v s1 e c1
    | _ -> failwith "unknown instruction"
...
```

インタプリタと同様に、整数の階乗を求める $fact$ 関数を実行してみる。

```
let run src =
  Eval.eval (Main.parse(src)) (Eval.emptyenv())
;;


let instr = run "let rec fact x = if x = 0 then 1 else x * fact (x-1) in fact 10";;
Am.am_eval instr;;
```

```
# val run : string -> Am.code = <fun>
val instr : Am.code =
  [Am.I_Pushenv;
   Am.I_Mkclos
    [Am.I_Ldi 0; Am.I_Push; Am.I_Search 0; Am.I_Eq;
     Am.I_Test ([Am.I_Ldi 1],
      [Am.I_Pushenv; Am.I_Ldi 1; Am.I_Push; Am.I_Search 0; Am.I_Sub;
       Am.I_Push; Am.I_Search 1; Am.I_Apply; Am.I_Popenv; Am.I_Push;
       Am.I_Search 0; Am.I_Mult])];
   Am.I_Extend; Am.I_Pushenv; Am.I_Ldi 10; Am.I_Push; Am.I_Search 0;
   Am.I_Apply; Am.I_Popenv; Am.I_Popenv]
- : Am.am_value = Am.AM_IntVal 3628800
```

正しく計算できていることが分かる。

また、インタプリタと同様に正しくない式についても実行してみる。

```
run "1++1";;
Exception: Failure "parse error near characters 2-3".

run "1+true";;
val instr : Am.code = [Am.I_Ldb true; Am.I_Push; Am.I_Ldi 1; Am.I_Add]
Exception: Failure "unexpected type of argument(s) for binary operation".

run "if 1 then 1 else 0";;
val instr : Am.code = [Am.I_Ldi 1; Am.I_Test ([Am.I_Ldi 1], [Am.I_Ldi 0])]
Exception: Failure "boolean expected for if-test".
```

正しくない構文についてはインタプリタ同様パース時にエラーが出ているが、型エラーについてはコンパイルは成功し、実行時にエラーになっている。

5.2 ベンチマーク

fibonacci 数を求めるプログラムを作成し、「標準の Ocaml」「ミニ Ocaml インタプリタ」「ミニ Ocaml コンパイラ」のそれぞれで 100 回ずつ実行し、結果を比較した。

実行したソースコードは以下のとおり。末尾の N の数を変えて、幾つかの計測を行う[*2]。

```
let rec fib n = if n > 1 then fib(n-1) + fib(n-2) else 1 in fib N;;
```

■10 番目の fibonacci 数

```
               Rate mini_compile  mini_interp        ocaml
mini_compile 40.7/s           --          -8%         -18%
mini_interp  44.2/s           9%           --         -11%
ocaml        49.5/s          22%          12%           --
perl benchmark.pl 100  5.77s user 1.06s system 95% cpu 7.173 total
```

[*2] 計測を行ったプロセッサは Intel Core i7 2.3GHz

■20 番目の fibonacci 数

```
               Rate mini_compile  mini_interp        ocaml
mini_compile 2.88/s           --         -67%         -93%
mini_interp  8.68/s         202%           --         -80%
ocaml        44.1/s        1431%         407%           --
perl benchmark.pl 100  47.43s user 1.22s system 99% cpu 49.041 total
```

■50 番目の fibonacci 数

一時間半以上たっても計算が終わらなかったため省略。

5.3 考察

結果は常に「コンパイラ」「インタプリタ」「標準」の順に時間が掛かっている。また、計算回数が多ければ多いほど、大きく差が広がるのがわかる。

インタプリタは式をパースして評価するだけであるが、コンパイラは表現を変換してから評価するという手順を踏むため、より長い時間が掛かっていると考えられる。

6 実験の感想

私が Ocaml を触ったのは本実験が初めてであるが、この実験を通して、Ocaml の強力さの一端に触れられたと思う。

普段は Ruby のような動的型付けの言語や Perl のように型システムを殆ど持っていない言語を触っていたため、最初は強い型付けに戸惑うことも多かった。しかし、パターンマッチングとヴァリアント型を使った時に、型があることの優位性に気付かされた。型のゆるい言語では、データ構造からパターンに値を束縛することや、パターンの網羅性チェックなどは難しい。

また、関数プログラミングの利点としては、再帰的なデータ構造に対するアプローチが自然な方法で記述できることがあげられる。今までは関数の再帰呼び出しは難解なものというイメージを持っていたが、この実験でそのイメージは払拭されたように思う。