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 11 月 12 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 *)
```

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

```
  | 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、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 $Plus(Times(IntLit\,2, IntLit\,2), Plus(IntLit\,3, IntLit\,(-4)))$ 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 ~*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;;

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

(* eval3 : exp -> (string * value) list -> value *)
(* let と変数、環境の導入 *)
let print_env env =
  print_string(string_of_env env);;

let rec eval6 e env =           (* env を引数に追加 *)
  let binop f e1 e2 env =       (* binop の中でも eval6 を呼ぶので env を追加 *)
    match (eval6 e1 env , eval6 e2 env ) 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 *)
  | Minus(e1, e2) -> binop (-) e1 e2 env
  | 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)
    | _ -> failwith "wrong value"
      end
  | If(e1,e2,e3) ->
      begin
        match (eval6 e1 env ) with          (* env を追加 *)
          | BoolVal(true)  -> eval6 e2 env   (* env を追加 *)
          | BoolVal(false) -> eval6 e3 env   (* env を追加 *)
          | _ -> failwith "wrong value"
      end
  | Let(x,e1,e2) ->
      let env1 = ext env x (eval6 e1 env )
      in eval6 e2 env1
 | LetRec(f,x,e1,e2) ->
      let env1 = ext env f (RecFunVal (f, x, e1, env))
      in eval6 e2 env1
  | Noteq(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)
    | _ -> failwith "wrong value"
      end
  | Fun(x,e1) -> FunVal(x, e1, env)
  | App(e1,e2) ->
      let funpart = (eval6 e1 env) in
      let arg = (eval6 e2 env) in
        begin
         match funpart with
          | 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";;
```