

3.4 ML³ — 関数の導入

ここまでのところ、この言語には、いくつかのプリミティブ関数（二項演算子）しか提供されておらず、ML プログラマが（プリミティブを組み合わせて）新しい関数を定義することはできなかった。ML³ では、fun 式による関数抽象と、関数適用を提供する。

3.4.1 関数式と適用式

まずは、ML³ の式の文法を示す。

```
〈式〉 ::= ...
        | fun 〈識別子〉 → 〈式〉
        | 〈式1〉 〈式2〉
```

構文に関するインタプリタ・プログラムは、図 3.1 に示す。適用式は左結合で、他の全ての演算子よりも結合が強いとする。

3.4.2 関数閉包と適用式の評価

さて、OCaml と同様、ML³ においても、関数は式を評価した結果となるだけでなく、変数の束縛対象にもなる第一級の値 (*first-class value*) として扱う。そのため、expressed value, denoted value ともに関数値を加えて拡張する。

```
Expressed Value = 整数 (... , -2, -1, 0, 1, 2, 3, ...) ⊕ 真偽値 ⊕ 関数値
Denoted Value  = Expressed Value
```

さて、関数値をどのようなデータで表現すればよいか、すなわち `fun x -> e` という関数式を評価した結果をどう表現すればよいかを考えよう。この式を評価して得られる関数値は、何らかの値に適用されると、仮引数 `x` を受け取った値に束縛し、関数本体の式 `e` を評価する。したがって、関数値は少なくともパラメータの名前と、本体の式の情報を含んでいなければならない。したがって、以下のように `exval` を拡張して関数値のためのコンストラクタ `ProcV` を定義することが考えられる。

```
(* 注：これはうまくいかない *)
type exval =
  ...
  | ProcV of id * exp
and dnval = exval
```

しかし、実際はこれだけではうまくいかない。以下の ML³ のプログラム例を見てみよう。

syntax.ml:

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

parser.mly:

```
%token RARROW FUN  
  
Expr :  
  ...  
  | e=FunExpr { e }  
  
MExpr :  
  e1=MExpr MULT e2=AppExpr { BinOp (Mult, e1, e2) }  
  | e=AppExpr { e }  
  
AppExpr :  
  e1=AppExpr e2=AExpr { AppExp (e1, e2) }  
  | e=AExpr { e }
```

lexer.mll:

```
let reservedWords = [  
  ...  
  ("fun", Parser.FUN);  
  ...  
]  
...  
| "=" { Parser.EQ }  
| "->" { Parser.RARROW }
```

図 3.1: 関数と適用 (1)

```

let f =
  let x = 2 in (* (A) *)
  let addx = fun y → x + y in
  addx
in
f 4

```

この例で定義している関数 `addx` は受け取った値に `x` の値を加えて返す関数である。前述のように関数値を表現するとこの `addx` は `ProcV("y", BinOp(Plus, Var "x", Var "y"))` に束縛されるはずである。このプログラムは `f` を `addx` の評価結果に束縛するので、`f` を

```
ProcV("y", BinOp(Plus, Var "x", Var "y"))
```

に束縛した環境で関数適用式 `f 4` を評価しようとする。したがって、上述した関数適用の直観的なセマンティクスによれば、変数 `y` を 4 に束縛して `BinOp(Plus, Var "x", Var "y")` を評価することになるのだが、この時点では環境中に `x` がいないために `((* (A) *))`¹ の `x` の束縛はこの時点ではスコープを外れていることに注意) このままだと正しくプログラムを評価することができない。

問題は、`addx` が束縛される関数式 `fun y → x + y` に自由変数 `x` が含まれていることである。この `x` は、OCaml と同様に `addx` が定義された時点の `x` の値（すなわち、`(A)` の行で導入される束縛が有効）なのだが、関数値に仮引数と関数本体の式のみを含める現在の定義では、このような関数式の自由変数が扱えない。このような自由変数を扱うためには、関数値に仮引数、関数本体の式に加えて、関数値が作られたときに自由変数が何に束縛されているか、すなわち現在の例では `x` が 2 に束縛されているという情報を記録しておかなければならない。というわけで、一般的に関数が適用される時には、

1. パラメータ名
2. 関数本体の式、に加え
3. 本体中のパラメータで束縛されていない変数 (自由変数) の束縛情報 (名前と値)

が必要になる。この3つを組にしたデータを関数閉包・クロージャ (*function closure*) と呼び、これを関数値として用いる。ここで作成するインタプリタでは、本体中の自由変数の束縛情報として、`fun` 式が評価された時点での環境全体を使用する。これは、本体中に現れない変数に関する余計な束縛情報を含んでいるが、もっとも単純な関数閉包の実現方法である。

以上を踏まえた `eval.ml` への主な変更点は図 3.2 のようになる。式の値には、環境を含むデータである関数閉包が含まれるため、`exval` と `dnval` の定義が (相互) 再帰的になる。関数値は `ProcV` コンストラクタで表され、上で述べたように、パラメータ名のリスト、本体の式と環境の組を保持する。`eval_exp` で `FunExp` を処理する時には、その時点での環境、つまり `env` を使って関数閉包を作っている。適用式の処理は、適用される関数の評価、実引数の

¹`(* (A) *)`ってクマみたいに見えますね。

eval.ml:

```
type exval =
  IntV of int
  | BoolV of bool
  | ProcV of id * exp * dnval Environment.t
and dnval = exval

let rec eval_exp env = function
  ...
  (* 現在の環境 env をクロージャ内に保存 *)
  | 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') ->
       (* クロージャ内の環境を取り出して仮引数に対する束縛で拡張 *)
       let newenv = Environment.extend id arg env' in
       eval_exp newenv body
     | _ -> err ("Non-function value is applied"))
```

図 3.2: 関数と適用 (3)

評価を行った後、本当に適用されている式が関数かどうかのチェックをして、本体の評価を行っている。本体の評価を行う際の環境 `newenv` は、関数閉包に格納されている環境を、パラメータ・実引数で拡張して得ている。

Exercise 3.4.1 ML^3 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) (f 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 ML³ 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、ML⁴ 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、Exercise 3.4.4 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` (Exercise 3.4.5 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
```