

主専攻実験[S-8] 関数プログラミング

課題 5-1,5-2,5-3

情報科学類 202113564 三村潤之介

## 課題 5-1

- 示されたプログラムを静的束縛および動的束縛のもとで、手で実行し、計算結果を導け。  
最後の行の関数適用部から等号でつないで計算する。

<静的束縛>

```
f(x × y)      ← x = 20
= f(20 × y)   ← y = f(x)
= f(20 × f(x))      ← x = 10
= f(20 × f(10)) ← f(y) = x + y
= f(20 × (x + 10)) ← x = 10
= f(20 × 20)
= f(400)      ← f(y) = x + y
= x + 400      ← x = 10
= 410
```

<動的束縛>

```
f(x × y)      ← x = 20
= f(20 × y)   ← y = f(x)
= f(20 × f(x))      ← x = 20
= f(20 × f(20)) ← f(y) = x + y
= f(20 × (x + 20)) ← x = 20
= f(20 × 40)
= f(800)      ← f(y) = x + y
= x + 800      ← x = 20
= 820
```

## 課題 5-2

- 示されたプログラムをミニ OCaml 言語のプログラムとして表現し、eval4 で計算せよ。  
示されたプログラムをミニ OCaml 言語のプログラムとして表現し、実行すると以下のようになる。ここで、e は何も変数が宣言されていない環境である。

```
# eval4
(
  Let("x", IntLit(1),
    Let("f", Fun("y", Plus(Var("x"), Var("y"))),
      Let("x", IntLit(2),
        App(Var("f"), Plus(Var("x"), IntLit(3)))
      )
    )
  )
)
```

```

    )
  )
)
(e)

```

```
;;
```

```
- : value = IntVal 6
```

計算結果として、IntVal 6 が返ってくることが確かめられた。

・eval4 の APP ケースで、env1 を env に変更したとすると、プログラム例がどう計算されるか確かめよ。

以下に計算結果を示す。

```
# eval4
```

```

(
  Let("x", IntLit(1),
    Let ("f", Fun("y", Plus(Var("x"),Var("y"))),
      Let("x", IntLit(2),
        App(Var("f") , Plus(Var("x"), IntLit(3)))
      )
    )
  )
)
(e)

```

```
;;
```

```
- : value = IntVal 7
```

計算結果は IntVal 7 になる。App にて、本来 env1 を用いて評価していたのは、Fun で定義された時点での環境を呼び出すためである。上の例で言えば、もともとは、env1 を用いて、let f fun y -> x+y という定義がされた時点での環境、すなわち x=1 とされている環境が呼び出されていた。App にて、これを今回のように env を用いて評価すると、Fun で定義された時点での環境を参照することなく、現時点(App が呼び出された時点)での環境、すなわち x=2 とされている環境が呼び出され、評価される。

この動作は動的束縛であるといえる考える。

・ここまでに出来たミニ OCaml の構文だけを使って、階乗を求める関数を定義できるか、考察せよ。

階乗を求める関数を定義する際、一般的には再帰関数の形を使うため、定義できるかどうかは、この構文に再帰関数の機能があるかどうかであると考えた。そこで、以下のようなミニ Ocaml 用プログラムを書いて eval4 で評価させた。

このプログラムでは、y が 5 であれば 1 を返し、そうでなければ、 $1+f(y+1)$  を返すと関数  $f(y)$  を定義し、これを用いて  $f(3)$  を計算させようとしている。

```
# eval4
(
  Let(
    "f",
    Fun(
      "y",
      If(
        Eq (Var("y"), IntLit(5)) ,
        IntLit(1),
        Plus(IntLit(1), App(Var("f"),Plus(IntLit(1), Var("y"))))
      )
    ),
    App(Var("f"),IntLit(3))
  )
)
(e)
;;
Exception: Failure "unbound variable: f".
```

$f$  が定義されていないとするエラー文が出るという結果となった。この文が出る可能性があるのは 2 回出現している  $\text{Var}("f")$  であるが、そのうち 2 つめの出現に原因があるとする、 $f$  を定義している  $\text{Let}$  文が正常に働かなかったことになる。とするならば、 $\text{Let}$  文内に関するエラー文が出るはずであるので、結果のエラー文は  $\text{Let}$  文内の  $\text{Var}("f")$  を評価している際のものであると考える。

これは、 $f$  を定義する  $\text{Let}$  文中において  $f$  を用いて定義すること、すなわち再帰的に関数を定義することができないことを示している。

よって、これまでの構文で階乗を求める関数は定義できないと考える。

以下は実験 5-3 を終えての追記であるが、5-3 の実験資料には、「ラムダ計算の授業などを受講している人は、『ラムダ式だけを使って再帰関数を定義する方法』として習ったことがあるだろう。」とある。これによれば再帰関数はラムダ式のみで定義できるらしいの

である。eval4 自体にはラムダ式の実装の機能があると思われるため、階乗を求める関数は定義できる可能性が高いと考える。

・eval4 の関数適用の計算の順序は、OCaml 処理系の評価順序とは一致していない。例に示す Ocaml プログラムを実行してみよ。

示される 2 つのプログラムを実行した結果を以下に示す。

```
# (print_string "1"; 10) + (print_string "2"; 20);;
21- : int = 30
# (print_string "1"; (fun x -> x))(print_string "2"; 20);;
21- : int = 20
```

ここから、和の演算および関数適用において、後ろの引数から評価が行われていることが分かる。

・eval4 が  $(e1+e2)$  や  $(e1\ e2)$  という式の計算をどういう順序で行うか答えなさい。また、OCaml と一致させるように eval4 を変更しなさい。

eval4 内では、基本的に eval4 e1 の方が eval4 e2 よりも先に出現する。よって  $e1 \rightarrow e2$  の順序で計算される。これを逆転させるためには eval4 e2 の方が先に出現させる必要がある。

Plus(e1, e2)においては、関数 binop を用いるので、binop を変更した。

```
let binop f e1 e2 env =
  match (eval4 e2 env, eval4 e1 env) with
  | (IntVal(n2), IntVal(n1)) -> IntVal(f n1 n2)
  | _ -> failwith "integer value expected"
in
```

Fun(e1 e2)においては、e2 を評価したものを arg として先に宣言することで、e2 の方が先に評価されることになる。

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

### 課題 5-3

・再帰関数の動きを見るため、例に示すテストプログラムを空の環境のもとで評価し、その実行過程を考えよ。

```
# eval6
(
  LetRec("f", "x", Var("x"), IntLit(0))
)
(emptyenv())
```

::

- : value = IntVal 0

これは `f` を定義しているが `App` などにより呼び出されることはない。この場合でも `f` は `RecFunVal` として新たな環境 `env1` に登録され、この `env1` を用いて `IntLit(0)` を評価している。

```
# eval6
(
  LetRec("f", "x", Var("x"), App(Var("f"), IntLit(0)))
)
(emptyenv())
```

::

これは実際に定義された `f` を再帰呼び出ししている。

- : value = IntVal 0

```
# eval6
(
  LetRec("f", "x", If(Eq(Var("x"), IntLit(0)), IntLit(1), Plus(IntLit(2), App(Var("f"),
Plus(Var("x"), IntLit(-1))))), App(Var("f"), IntLit(0)))
)
(emptyenv())
```

::

- : value = IntVal 1

```
# eval6
(
  LetRec("f", "x", If(Eq(Var("x"), IntLit(0)), IntLit(1), Times(Var("x"), App(Var("f"),
Plus(Var("x"), IntLit(-1))))), App(Var("f"), IntLit(3)))
```

```

    )
    (emptyenv())
;;
- : value = IntVal 6
# eval6
(
    LetRec("f", "x", If(Eq(Var("x"),IntLit(0)), IntLit(1), Times(Var("x"), App(Var("f"),
Plus(Var("x"), IntLit(-1))))), App(Var("f"), IntLit(5)))
    )
    (emptyenv())
;;
- : value = IntVal 120

```

2 つめ以降は基本的な処理の流れは似ていて、違うところはいわゆる body 部である。body 部は App が呼び出されたときに、評価される。

- ・上記以外に再帰関数を定義し、eval6 を使って実行せよ。

例に示されている fib を定義し、実行した。

```

# eval6
(
    LetRec("fib", "x",
        If (Greater(IntLit(3), Var("x")), IntLit(1), Plus(App(Var("fib"),
Plus(Var("x"), IntLit(-1))), App(Var("fib"), Plus(Var("x"), IntLit(-2))))),
        App(Var("fib"), IntLit(6))
    )
    )
    (emptyenv())
;;
- : value = IntVal 8

```

課題 1 での実装とは異なり、定義通りの実装である。正しく計算されている。

もう一つ例として示されている gcd を定義しようと試みたが、2 つ引数をもつ再帰関数をこの Ocaml でどう実装するかが分からなかった。value として value のリストである ListVal が定義されているため、これを用いようとしたが、LetRec 自体が引数を 1 つしかもたないよう設計されている。よって 2 引数をもつ gcd は実装できないと考えた。