主専攻実験[S-8] 関数プログラミング

課題 3-1,3-2

情報科学類 202113564 三村潤之介

課題 3-1

・環境を操作する関数を使って[("x",IntVal 1); ("y",BoolVal true); ("z", IntVal 5)] という環境を作りなさい。

環境に変数を追加する ext 式は、受け取ったペアを環境の先頭に追加するため、最初に追加したものがリストの最後部となる。よって、この場合、変数”z”から環境に追加していく必要がある。以下のように操作した。

```
# let env = emptyenv();;
val env : 'a list = []
# let env = ext env "z" (IntVal 5);;
val env : (string * value) list = [("z", IntVal 5)]
# let env = ext env "y" (BoolVal true);;
val env : (string * value) list = [("y", BoolVal true); ("z",
IntVal 5)]
# let env = ext env "x" (IntVal 1);;
val env : (string * value) list =
  [("x", IntVal 1); ("y", BoolVal true); ("z", IntVal 5)]
```

最初に環境 env を初期化して定義し、以降その env について z→y→x の順に環境に情報を追加していった。最後の行から、適切に環境を作れていることがわかる。

・この環境に y 及び w という変数が登録されているかどうか、lookup 関数で調べなさい。以下のように操作した。

```
# lookup "y" env;;
- : value = BoolVal true
# lookup "w" env;;
Exception: Failure "unbound variable: w".
```

上部は、y が env 内に存在し、値は BoolVal true であることを示している。下部は、w が env 内に存在しない変数であることを示している(エラー)。これらは正常な動作である。

・1 つの環境の中に同じ変数が 2 回現れると、lookup のときにどちらの値が使われるか。

例として、 [("z", BoolVal false); ("x", IntVal 1); ("y", BoolVal true); ("z", IntVal 5)] である環境を用意した。これは先程までに作っていた環境の先頭に、すでに登録されている z をもう一度、別の値で登録させたものである。ここで z に対して lookup を行うと、以下のようになった。

```
# lookup "z" env;;
- : value = BoolVal false
```

これは、z が BoolVal false として登録されていることを示す。つまり、リストの先頭側にあるもののほうが、選ばれている。lookup が先頭からリストを見ていって、該当する変数が見つかったら、それに対応する値を返し、以降探索を行わないため、こうなると予想する。…(ref.1)

・関数 ext より、「新しく環境に定義された変数と値のペア」が環境の先頭と末尾のどちらに追加されるか考えなさい。また、ある変数に対して異なる値を、続けて環境に登録すると、古い値と新しい値のどちらが生き残るか考えなさい。

ext において、「::」はリストの結合を表しており、これにより、与えられた env の前に与えられたデータペアを結合するため、環境の先頭に追加されることになると考える。

また、同じ変数が同環境内に存在したとき、lookup は、前述した理由(ref.1)によって、先頭側の方を返すので、「新しい値が生き残る」と表現できると考える。

課題 3-2.

・eval3 に種々の例題を与えてテストしなさい。

環境 env で、(“x”,3)が登録されているとき、以下のように例を与えた。

```
# eval3
  (If(Eq(Var("x"),Plus(IntLit(1),   IntLit(2))),   IntLit(1),
IntLit(0)))
  (env)
  ;;
- : value = IntVal 1
```

これは、 もし x＝(1+2) であれば 1、そうでなければ 0 を返す式である。x には予め 3 が与えられているので、1 が返されている。

・Ocaml と同様に eval3 でも内側の let が優先されていることを確かめよ。

以下は let x=1 in let x=2 in x を表した式を与えた結果である。

```
# eval3
  (Let("x", IntLit(1), Let("x", IntLit(2), Var("x"))))
  (env)
  ;;
- : value = IntVal 2
```

2、すなわち内側の let 文が反映された状態での x の値が返された。このことから、内側の let が優先されていることがわかる。

・Ocaml と同様に、eval3 でも外側の let で定義された変数を内側の let 式の中で使うことができることを確かめよ。また、処理の過程で、環境がどのように変化していくか考えなさい。

実行した eval3 式とその結果を示す。

```
# eval3
  (Let("x", IntLit(1), Let("y", Plus(Var("x"), IntLit(1)),
Plus(Var("x"), Var("y")))))
  (env)
  ;;
- : value = IntVal 3
```

よって、内側の Let 文内の x について、外側の Let の内容が反映されていることがわかる。値は以下のように計算できることから、正しい。

x + y　　y:=x+1
=x+x+1　x:=1
=1+1+1
=3

環境は、Let 文によって、その内側を評価するタイミングで、変化すると考える。この例で言えば、外側の Let によって、その第三引数である Let("y", Plus(Var("x"), IntLit(1)), Plus(Var("x"), Var("y")))内で x の値を参照することができるようになっている。

・eval3 では「環境から、束縛を除去する」という操作は一切使っていないのに、どうして「解除」の操作が実現できているのか考えなさい。

例(let x=1 in (let x=2 in x+1)+(x*2))の式を eval3 用に書き換えると、

```
(Let("x", IntLit(1), Plus(Let("x", IntLit(2), Plus(Var("x"),
IntLit(1))), Times(Var("x"), IntLit(2)))))
```

となる。

この例で言えば、「解除」の動作は、1 つめの Var(“x”)の出現は IntLit(2)であるのに、2 つめの出現は IntLit(1)であることに当たる。このような Let 文の入れ子では、env をスタックを積み上げるように複数保持しながら、適切なところでポップするという、再帰処理の特性がこの「解除」の操作に見えると考える。