

Solutions for Exercise Sheet 8

Exercise 1: Objects

Given below is a class for bank account objects (as a “new” procedure) and a sample piece of code using it.

The class is written in applied lambda calculus with assignments, similar to Scheme, but using structs to represent structures with multiple components (instead of tuples or message dispatch functions).

```
let newaccount =
  λ(initial).
    let balance = initial
    in {deposit =
        λ(amount). balance := balance + amount;
      wdraw =
        λ(amount). if balance > amount then
                      balance := balance - amount
                    else print(`insufficient funds`);
      getBalance =
        λ(). balance
    }
in let
  Mary = newaccount(500);
  Doug = newaccount(200);
  Lisa = Doug
in
  (Mary.wdraw(200);
   Lisa.deposit(200);
   Doug.wdraw(200);
   Mary.getBalance()
  )
```

- a. *Write equivalent code in Java with a class definition corresponding to newaccount and statements invoking the class in the fashion shown.*

```
class Account {
  int balance;

  Account (initial) {
    balance = initial;
  }

  void deposit (int amount) {
    balance = balance + amount;
  }

  void wdraw (int amount) {
    if (balance > amount)
      balance = balance - amount;
    else
      System.out.print "insufficient funds";
  }
}
```

```

    int getBalance () {
        return balance;
    }
}

public static void main (String[] args) {
    Account Mary = new Account(500);
    Account Doug = new Account(200);
    Account Lisa = Doug;

    Mary.wdraw(200);
    Lisa.deposit(200);
    Doug.wdraw(200);
    System.out.println(Mary.getBalance());
}

```

- b. Draw an environment diagram showing all the frames and variable values at the end of the inner `let` body.
- c. Add a method called `transfer` to the `account` class. It should take as arguments another account object to which money should be transferred and the amount to be transferred.

```

transfer =
  λ(toAccount, amount).
    if balance > amount then
      (balance := balance - amount;
       toAccount.deposit(amount))
    else print('insufficient funds')

```

- d. Write a procedure `payInterest`, which takes a list of account objects and deposits 5% interest in all of them.

```

payInterest =
  λ(accounts).
    if null(accounts) then skip
    else let a = car(accounts)
         in (a.deposit(a.getBalance() * 0.05);
            payInterest(cdr(accounts)))

```

Exercise 2: Higher order procedures

- a. Write a procedure called `mapdo` which takes as its arguments: a one-argument procedure `p`, and a list of items `items`, and applies `p` to all the items in `items` in the left-to-right order.

```

mapdo =
  λ(p, items).
    if null(items) then skip
    else (p(car(items)); mapdo(p, cdr(items)))

```

- b. Use the procedure `mapdo` to rewrite the `payInterest` procedure. You should not use recursion in this definition.

```

payInterest =
  λ(accounts).
    mapdo(λ(a). a.deposit(a.getBalance() * 0.05), accounts)

```

- c. Use the procedure `mapdo` to calculate the total balance of a list of accounts. Again, you should not use recursion in this definition.

```

totalBalance =
  λ(accounts).
    let total = 0
    in (mapdo(λ(a). total := total + a.getBalance(), accounts);
        total)

```

- d. Explain how the total balance procedure works, showing the environment frames that arise during its execution.

Assume some global environment frame F_0 . When the totalBalance procedure is called with a list of accounts as an argument, it first creates the following frames:

$$\begin{aligned}
 F_1 &= \{\text{accounts} \mapsto \dots\} \\
 F_2 &= \{\text{total} \mapsto 0\}
 \end{aligned}$$

and executes the body of the let block in the environment $F_0 \leftarrow F_1 \leftarrow F_2$. The first argument of mapdo evaluates to a closure:

```

cls(λ(a). total := total+a.getBalance(), F0 ← F1 ← F2)

```

The assignment to total in this closure will update the value of total in frame F_2 . The mapdo procedure will call this closure for every element of the account list. So, when mapdo returns, the sum of all the account balances will be found in the value of total in frame F_2 . This value is returned.

Exercise 3: More objects

- a. Write a class newbuffer2 for two-place buffers, which can hold at most two items. The objects should have: a put method, using which an item can be placed in the buffer, a get method, using which an item can be retrieved from the buffer, and a method present which tells whether there are any items in the buffer.

```

newbuffer2 =
  λ().
    let val1 = nil;
        val2 = nil;
        n = 0
    in struct {
      put = λ(x).
        if n = 0 then (val1 := x; n := 1)
        else if n = 1 then (val2 := x; n := 2)
        else println(`buffer isfull`);
      get = λ().
        let result = nil
        in if n > 0 then
          (result := val1; val1 := val2; n := n-1;
           result)
        else println(`buffer is empty`);
      present = λ(). n > 0
    }

```

- b. Next write a class newbuffer, for buffers that can hold an unbounded number of items. You should not use lists or any other data structure to write this class.

(Hint: Can you implement a two-place buffer using two one-place buffers?)

Yes. Imagine buffer1 sitting in the front row and handling get's and put's. When it gets a put-request but is already full, it passes it on to the buffer2 behind it. If it gets a get-request, it responds to it but also retrieves any extra item in buffer2 and stores it internally. That way, buffer2 gets freed up to hold any new items that arrive.

Similarly, an unbounded buffer can be implemented by having a buffer sitting at the front row and having another unbounded buffer behind it. So, this is going to be a recursive definition of the unbounded buffer class.

```
newbuffer =
  λ().
  let val = nil;
      more = nil;
      n = 0
  in struct {
    put = λ(x).
      if n = 0 then (val := x; n := 1)
      else (if null(more) then more := newbuffer() else skip;
            more.put(x))
    get = λ().
      let result = nil
      in if n > 0 then
          (result := val; val := more.get(); n := n-1;
           result)
      else println(`buffer is empty`);
    present = λ(). n > 0
  }
```