

Lab 7: Functional Data Abstraction and Memory Management

Learning objectives:

- Understand how data abstractions can be built using functions
- Gain experience implementing functional data abstractions in multiple languages
- Understand how the reference-counting memory management scheme works

You will make heavy use of higher-order functions and functional data abstractions in project 3, and this lab will prepare you for the work you will do for that project.

Because the Scheme questions in this lab require mutation, you are **allowed** to use procedures that mutate data such as `set!`, `set-car!`, and `set-cdr!`. However, you may only use `define` at the top (global) level.

Use the following commands to download and unpack the distribution code:

```
$ wget https://eecs390.github.io/lab/lab07/starter-files.tar.gz
$ tar xzf starter-files.tar.gz
```

1. *Functional pairs.* Implement a mutable-pair functional data abstraction in Scheme. The `mutable-pair` procedure should return a new instance of a mutable-pair data structure. A mutable pair is a pair of values that can be mutated after creation.

The mutable-pair data structure should support four operations via message passing:

- `'first` -- get the first element of the pair
- `'second` -- get the second element of the pair
- `'set-first!` -- set the first element of the pair
- `'set-second!` -- set the second element of the pair

An example of using this abstraction:

```
> (define p (mutable-pair 3 -1))
> (p 'first)
3
> (p 'second)
-1
> (p 'set-first! 5)
> (p 'set-second! 7)
> (p 'first)
5
> (p 'second)
7
```

Your implementation does **not** have to do any error checking (e.g. for unknown messages or a wrong number of arguments).

Write your implementation in `pair.scm`. Simple tests are included in the same file. To run the tests:

```
$ plt-r5rs pair.scm
```

2. *Functional classes.* Implement a bank-account functional data abstraction in Scheme. The `account` procedure takes in a number and should return a new instance of a bank-account data structure whose balance is the given number.

The bank account data structure should model a real bank account, supporting the following operations via message passing:

- 'balance -- get the current balance of the account
- 'deposit -- add the given amount to the balance
- 'withdraw -- remove the given amount from the balance and return the new balance. If the user tries to withdraw more than the account balance, return "Insufficient funds".

An example of using this abstraction:

```
> (define a (account 10))
> (a 'balance)
10
> (a 'deposit 15)
25
> (a 'balance)
25
> (a 'withdraw 10)
15
> (a 'withdraw 100)
"Insufficient funds"
```

Your implementation does **not** have to do any error checking (e.g. for unknown messages or a wrong number of arguments).

Write your implementation in `account.scm`. Simple tests are included in the same file. To run the tests:

```
$ plt-r5rs account.scm
```

3. *Functional inheritance*. In the course notes, we saw a definition of a bank account ADT using functions and dispatch dictionaries. The following is a version of this ADT using built-in Python dictionaries:

```
def account(initial_balance):
    def deposit(amount):
        new_balance = dispatch['balance'] + amount
        dispatch['balance'] = new_balance
        return new_balance

    def withdraw(amount):
        balance = dispatch['balance']
        if amount > balance:
            return 'Insufficient funds'
        balance -= amount
        dispatch['balance'] = balance
        return balance

    def get_balance():
        return dispatch['balance']

    dispatch = {}
    dispatch['balance'] = initial_balance
    dispatch['deposit'] = deposit
    dispatch['withdraw'] = withdraw
    dispatch['get_balance'] = get_balance

    def dispatch_message(message):
        return dispatch[message]

    return dispatch_message
```

Implement an ADT for a checking account that is a derived version of a bank account but charges a \$1 fee for withdrawal. Fill in the ADT definition for `checking_account()` in the `accounts.py` file.

Do **not** repeat code from `account()`. Instead, implement a scheme for deferring to `account()` where possible.

Your implementation does **not** have to do any error checking (e.g. for unknown messages or a wrong number of arguments).

To test your implementation, run the included doctests:

```
$ python3 -m doctest accounts.py
```

4. *Reference counting*. Consider the following Python code. Assume that Alpha and Beta are classes defined in the library module.

```
from library import Alpha, Beta

a = Alpha() # Alpha object created

def outer():
    b = Beta() # Beta object created
    c = a

    def inner():
        return b is c

    return inner

# Position 1

f = outer()
x = f()

# Position 2

f = a

# Position 3
```

Suppose the Python interpreter uses reference counting (with cycle detection) to manage heap objects.

- What is the reference count for the Alpha object when execution reaches *Position 1*?
- What is the reference count for the Alpha object when execution reaches *Position 2*?
- What is the reference count for the Beta object when execution reaches *Position 2*?
- What is the reference count for the Alpha object when execution reaches *Position 3*?
- What is the reference count for the Beta object when execution reaches *Position 3*?