CSE 1729:Principles of Programming

Lecture 19 : Mutable Data in Scheme

Kaleel Mahmood

Department of Computer Science and Engineering

University of Connecticut

OUR STORY THUS FAR...

- We have focused, so far, on a programming paradigm called: *Functional Programming*.
- When you build a program in a functional style, computation proceeds as the result of function application.
- In particular: *binding a variable only happens when a function is called*.
- This means (the golden rule of a functional language):

> Once a variable is bound, its value (in a particular environment) never changes

THE VIRTUES OF FUNCTIONAL PROGRAMMING

- Functional programming is predictable [redacted]
  - Variable values "never change."
  - Function definitions never change: Once a function has been defined (in a particular environment), applying it multiple times to a particular value will *always result in the same value*--it behaves like a mathematical function.

Are there cases where changing variable values during run time are important?

Functional Scheme vs Python

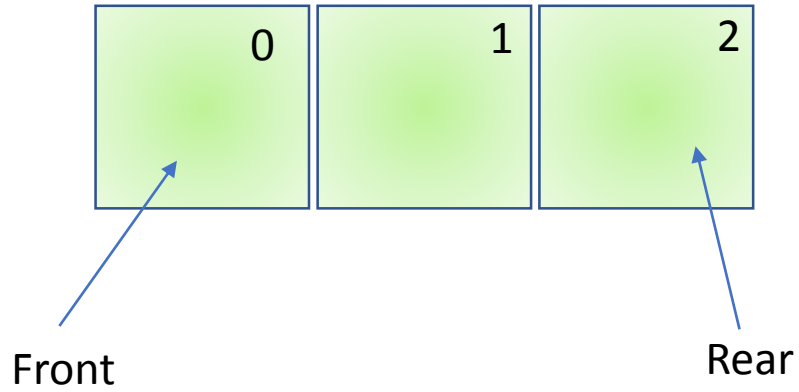
Queues

Queue Code in Python

```python
class Queue:
    def __init__(self, capacity):
        #Keep track of the front element and size of Queue
        self.front = self.size = 0
        self.rear = capacity - 1
        self.Q = [None] * capacity
        #Size of the queue is set on instantiation
        self.capacity = capacity

    def isFull(self):
        return self.size == self.capacity

    def isEmpty(self):
        return self.size == 0

    def EnQueue(self, item):
        #Check if the queue is full before adding an element
        if self.isFull():
            print("Full")
            return

        self.rear = (self.rear + 1) % (self.capacity)
        self.Q[self.rear] = item
        self.size = self.size + 1
        print("%s enqueue to queue" %str(item))

    def DeQueue(self):
        if self.isEmpty():
            return "Empty"
        print("%s dequeued from queue" %str(self.Q[self.front]))
        self.front = (self.front + 1) % (self.capacity)
        self.size = self.size - 1
```

Code Source: https://www.codegrepper.com/code-examples/python/queue+python+front+and+rear

Empty Queue with 3 nodes

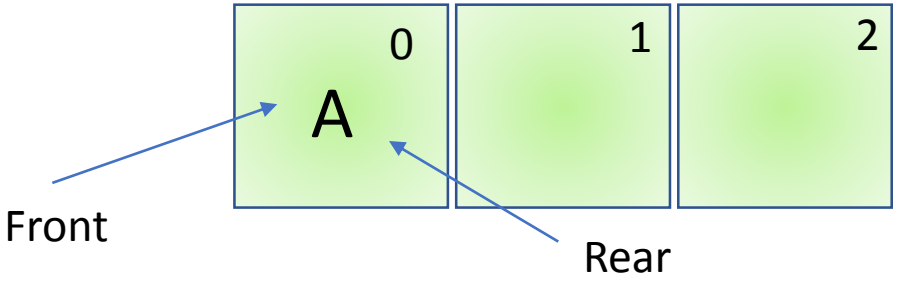


Now we want to add an element:

New Rear = (Old Rear + 1) % (Size of Queue)

New Rear = (2+ 1) % (3)

New Rear = 0



Queue Code in Scheme (using functional programming)

- There's a problem:
  - placing an element at the back is expensive
- Can you think of a better implementation?

```
(define (enqueue x Q)
  (if (null? Q)
      (list x)
      (cons (car Q)
            (enqueue x (cdr Q)))))

(define (front Q) (car Q))

(define (empty? Q) (null? Q))

(define (dequeue Q) (cdr Q))
```

Functional Scheme vs Python

```
(define (enqueue x Q)
  (if (null? Q)
      (list x)
      (cons (car Q)
            (enqueue x (cdr Q)))))

(define (front Q) (car Q))

(define (empty? Q) (null? Q))

(define (dequeue Q) (cdr Q))
```

```python
class Queue:
    def __init__(self, capacity):
        #Keep track of the front element and size of Queue
        self.front = self.size = 0
        self.rear = capacity - 1
        self.Q = [None] * capacity
        #Size of the queue is set on instantiation
        self.capacity = capacity

    def isFull(self):
        return self.size == self.capacity

    def isEmpty(self):
        return self.size == 0

    def EnQueue(self, item):
        #Check if the queue is full before adding an element
        if self.isFull():
            print("Full")
            return

        self.rear = (self.rear + 1) % (self.capacity)
        self.Q[self.rear] = item
        self.size = self.size + 1
        print("%s enqueue to queue" %str(item))

    def DeQueue(self):
        if self.isEmpty():
            return "Empty"
        print("%s dequeued from queue" %str(self.Q[self.front]))
        self.front = (self.front + 1) % (self.capacity)
        self.size = self.size - 1
```

- With our functional implementation we can see that the runtime of enqueue is $O(n)$
- With our Python implementation with “mutable” variables we can see the runtime of enqueue is $O(1)$.

MOTIVATION

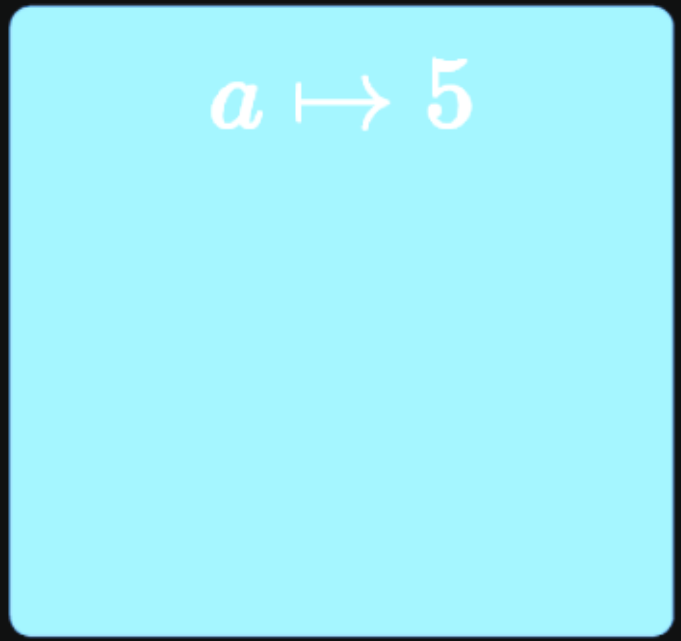
- Recall the queue datatype.
  - Problem: enqueue/dequeue in constant time.
- Idea: If we "knew" where the end of the queue was, we could avoid traversing the entire structure during an enqueue.
- Idea: represent a queue as a pair which maintains the front and back of the queue simultaneously.
- Then...

We need some more destructive tools in Scheme to accomplish this...

THE SET! FUNCTION

- The set! changes the value of a variable in the assignment in which it is called.
- A simple example:

```
1 > (define a 3)
2 > a
3 3
4 > (set! a 4)
5 > a
6 4
7 > (set! a 5)
8 > a
9 5
```



THE SEQUENCE/BEGIN FUNCTION

- The sequence function

```
(begin <expr1> <expr2> ... <exprk>)
```

carries out a sequence of function calls, returning the value of the last call.

- In purely functional programming, *there would be no point to such an operation* (since the return value would always simply be the value of <exprk>), but now that we have *side-effects*, sequences can be a valuable tool.

Warning: Your textbook calls this command: sequence.

What is the big deal about changing variables?

- When we want to model objects that change over time, we have a choice to make:

1. Model them functionally, as a sequence of “different” objects.
2. Model them via mutation, as single object that is changing.

- Consider sorting: philosophically, are we creating a new list, that is sorted, or massaging the old list into sorted order?

What is the big deal about changing variables?

From the previous question there are two important things to further note:

1. There will not always be a right answer.
2. We can think of scenarios where each type of modeling/designing will be *more* applicable.

A SIMPLE EXAMPLE

- A simple example: maintaining balance information in a bank account:

```
 1 > (define balance 100)
 2 > (define (withdraw f)
 3     (if (> f balance)
 4         "Insufficient funds!"
 5         (begin (set! balance
 6                      (- balance f))
 7                balance)))
 8
 9 > (withdraw 40)
10 60
11 > (withdraw 40)
12 20
13 > (withdraw 40)
14 "Insufficient funds!"
```

A Scheme string

A PROBLEM WITH THIS IMPLEMENTATION

```
> (define balance 100)
> (define (withdraw f)
    (if (> f balance)
      "Insufficient funds!"
      (begin (set! balance (- balance f))
             balance)))
```

- Anyone can now change the bank balance... `(set! balance 1000000000)`

We'd like an abstraction barrier, so that the only way to change the balance is via the official withdraw function

ABSTRACTION BARRIERS

Expose only what must be exposed:

- Protect the internals from accidental (or malicious) tampering.
- Protect the user.

The User Interface

HIDING THE BALANCE: RECALL *ENVIRONMENT DEFINITION SEMANTICS*

- Recall lexical scope rules: *the value of a free variable in a function is determined by the environment in which it was defined.*
- To warm up:

The value of x in the body of add is determined by the environment in which add was defined.

```
1 > (define x 10)
2 > (define (add y) (+ x y))
3 > (add 1)
4 11
5 > (set! x 100)
6 > (add 1)
7 101
```

In which environment is x created?

Basic Idea: We can use the environments in Scheme like “walls” to protect variables from manipulation

A very basic adding code:

```
1 (define (methodA a x)
2   (define (adder x) (+ a x))
3   (adder x)
4   )
```

Is there any “protection” in the above code?

Basic Idea: We can use the environments in Scheme like “walls” to protect variables from manipulation

A very basic adding code:

```
1  (define (methodA a x)
2    (define (adder x) (+ a x))
3    (adder x)
4    )
```

Now we further define an add 5 function:

```
6  (define (add-5 x)
7    (methodA 5 x)
8    )
```

Now the first value “a” is protected from manipulation:

```
1  (define (methodA a x)
2    (define (adder x) (+ a x))
3    (adder x)
4    )
```

```
6  (define (add-5 x)
7    (methodA 5 x)
8    )
```

```
11  (add-5 3)
```

```
Welcome to DrRacket, version 8.3 [cs].
Language: R5RS; memory limit: 128 MB.
8
>
```

Example where we try to manipulate variable “a” outside of the environment:

```
11  (set! a 3)
12  (add-5 3)
```

If ALL variables were accessible all the time, what would this code produce?

a = 3

x = 3

So a+x should be 6 right?

Now the first value “a” is protected from manipulation:

Example where we try to manipulate variable “a” outside of the environment:

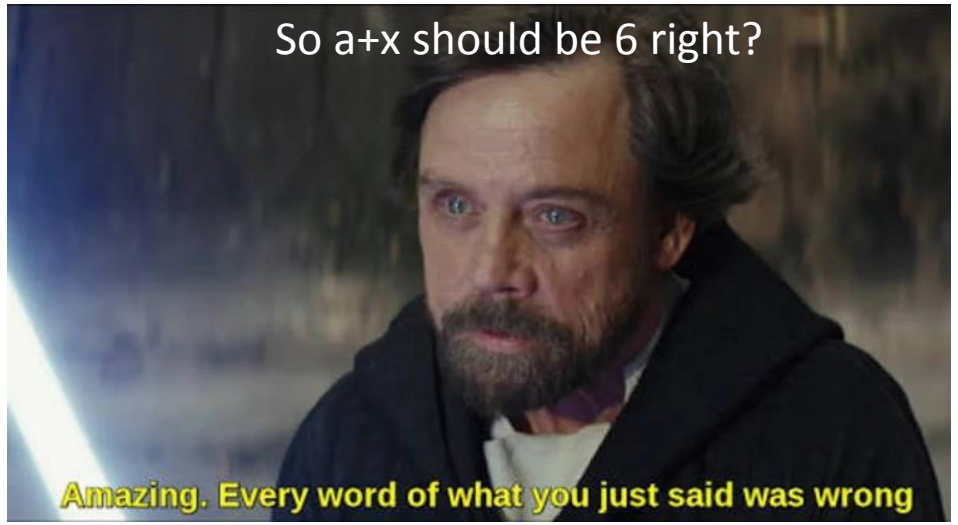
```
11 | (set! a 3)
12 | (add-5 3)
```

If ALL variables were accessible all the time, what would this code produce?

a = 3

x = 3

So a+x should be 6 right?



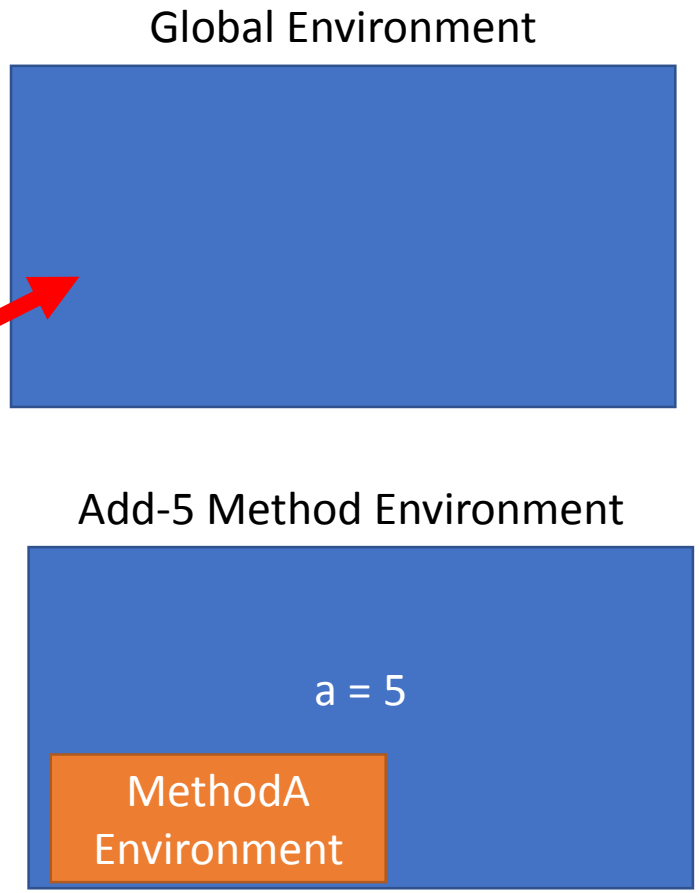

```
Welcome to DrRacket, version 8.3 [cs].
Language: R5RS; memory limit: 128 MB.
8
>
```

Why couldn’t we set “a”?

```
1  (define (methodA a x)
2    (define (adder x) (+ a x))
3    (adder x)
4    )
```

```
6  (define (add-5 x)
7    (methodA 5 x)
8    )
```

```
11 (set! a 3)
12 (add-5 3)
```



WE CAN USE THIS IDEA TO... HIDE (AND PROTECT) THE BALANCE

- The basic definition

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (lambda (withdrawal)
      (if (> withdrawal balance)
          "Insufficient funds"
          (begin
            (set! balance (- balance withdrawal))
          balance)))))
```

This function is returned; it has access to balance

- Usage

```
> (define my-acct-w (new-account 100))
> (my-acct-w 30)
70
> (my-acct-w 30)
40
```

Note nonfunctional behavior!

WHO SEES WHAT?

```
(define (new-account initial-balance)
  (let ((balance initial-balance))
    (lambda (withdrawal)
      (if (> withdrawal balance)
          "Insufficient funds"
          (begin
            (set! balance (- balance withdrawal))
            balance)))))
```

balance

This function is passed out of the environment; note that it refers to balance.

```
(define my-acct-w (new-account 100))
```

balance does not exist in this environment, but my-acct-w can change it!

THIS IS IMPORTANT...LET'S DO IT AGAIN

- Consider the code fragment:

```
(define (make-hidden-value initial-value)
  (let ((value initial-value))
    (define (mult-by x) (begin (set! value (* value x))
                               value))
    (define (add-to y)  (begin (set! value (+ value y))
                               value))
    (cons mult-by add-to)))
```

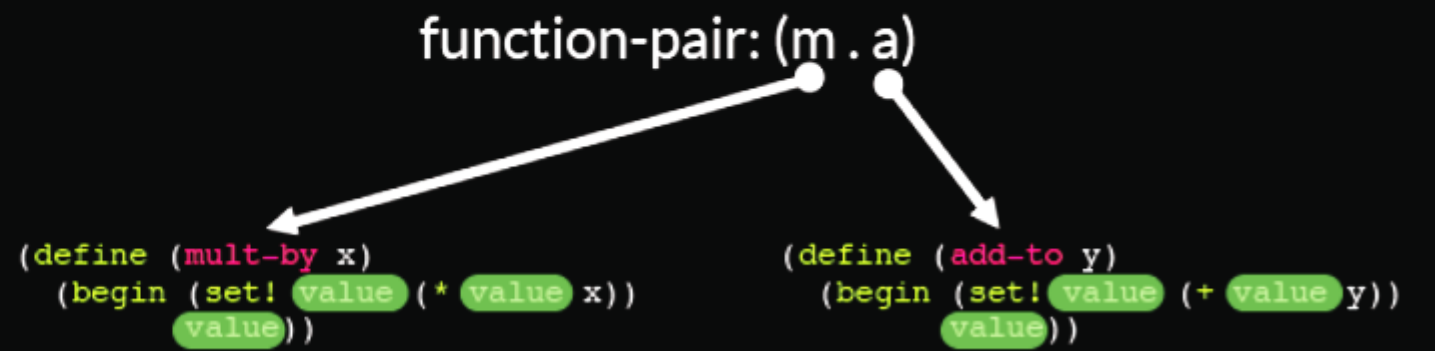
- What does it do?
  - It creates an environment with a variable value.
  - It passes back two functions (as a pair) which can operate on value.

One Example

```
(define (make-hidden-value initial-value)
  (let ((value initial-value))
    (define (mult-by x) (begin (set! value (* value x))
                               value))
    (define (add-to y)  (begin (set! value (+ value y))
                               value))
    (cons mult-by add-to)))
```

Let's follow it during an execution:

```
1 > (define function-pair (make-hidden-value 100))
2 > ((car function-pair) 2)
3 200
4 > ((car function-pair) 2)
5 400
6 > ((cdr function-pair) 2)
7 402
8 > ((cdr function-pair) 2)
9 404
```



OBJECTS

- In our bank account example, the only way to change the balance is through the withdrawal function. Thus, while there is destructive assignment, it is carried out behind an abstraction barrier.
- This fits into a paradigm called *object-oriented* programming:
  - Data objects are permitted private "state" information that can change over time, but
  - All interaction with these objects are carried out by specifically-crafted functions; no other functions have access to the state.

How would we do this in Python?

```python
# Python program to
# demonstrate private/public methods and variables

# Creating a class
class PrivateCompute:
    def __init__(self, privateValue):
        #private variable denoted with "__"
        self.__x = privateValue
        #public variable
        self.y = 3

    # Declaring public method
    def AddFive(self, input):
       return self.__x + input

    def revealPrivateValue(self):
        return self.__x

    # Declaring private method
    def __fun(self):
        print("Private Method")
```

Original Code Source: https://www.geeksforgeeks.org/private-methods-in-python/

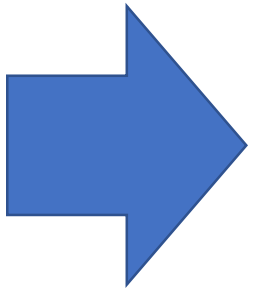
Preventing Manipulation in Python (1)

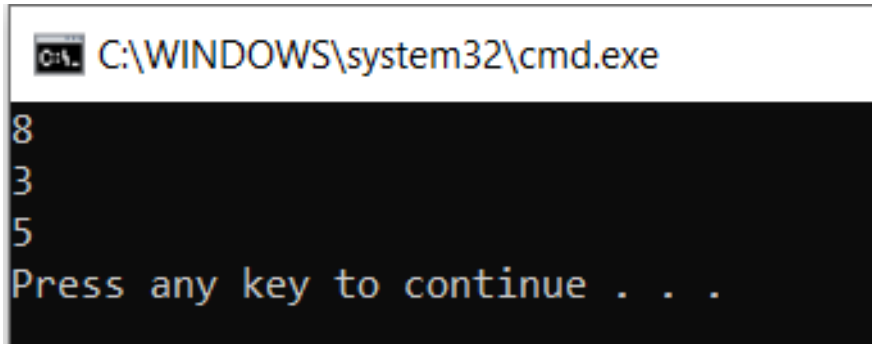
```python
# Python program to
# demonstrate private/public methods and variables

# Creating a class
class PrivateCompute:
    def __init__(self, privateValue):
        #private variable denoted with "__"
        self.__x = privateValue
        #public variable
        self.y = 3

    # Declaring public method
    def AddFive(self, input):
       return self.__x + input

    def revealPrivateValue(self):
        return self.__x

    # Declaring private method
    def __fun(self):
        print("Private Method")
```

- Using the “__” lets us denote variables as private or not able to be accessed outside the class in which they are defined.
- Essentially this functionality although not commonly used in Python, is built in.
- These features are something we should expect from a widely adopted object oriented programming language.

Preventing Manipulation in Python (2)

```
23  #Create an object
24  obj = PrivateCompute(5)
25  #Call the object to do computation
26  print(obj.AddFive(3))
27  #Try and access public variable
28  print(obj.y)
29  #Try and access private variable
30  #print(obj.__x) #The wrong way
31  print(obj.revealPrivateValue()) #The right way
```





- We can instantiate our object and then run the computing method.

Preventing Manipulation in Python (3)

```python
# Python program to
# demonstrate private/public methods and variables

# Creating a class
class PrivateCompute:
    def __init__(self, privateValue):
        #private variable denoted with "__"
        self.__x = privateValue
        #public variable
        self.y = 3

    # Declaring public method
    def AddFive(self, input):
       return self.__x + input

    def revealPrivateValue(self):
        return self.__x

    # Declaring private method
    def __fun(self):
        print("Private Method")
```

```python
#Create an object
obj = PrivateCompute(5)
#Call the object to do computation
print(obj.AddFive(3))
#Try and access public variable
print(obj.y)
#Try and access private variable
#print(obj.__x) #The wrong way
print(obj.revealPrivateValue()) #The right way
```

Note how the way to access private variables is by calling PUBLIC methods that are associated with the object.

Why is the Professor forcing you to look at Python and Scheme code?

Figure Sources

- https://i.chzbgr.com/thumb800/14732037/hDE6A9508/software-developer-nerdy-memes-geeky-memes-relatable-memes-funny-memes-memes-lol-programming-memes
- https://static.wikia.nocookie.net/e68bdafd-1f83-4d87-afdd-b49fdfb27f87/scale-to-width/755
- https://cdn.searchenginejournal.com/wp-content/uploads/2017/06/shutterstock_268688447.jpg