States and Imperative Programming

An object is said to ``have state" if its behavior is influenced by its history.

- A bank account has a state represented by the balance. It influences whether Rs. 5000/- can be withdrawn from it.

- A student in IIT has an academic state (CPI, Total Credits). Decides whether the student should be allowed to continue.

- The behavior of a lift can be modelled as a state modeled as a tuple: (Floor, req1, req2, ... reqn). The request are ordered as earlier first. Decides the next floor where the lift will stop.

  (2nd, 5, 3, 1, 6)

  (3rd, 5, 1, 6, 2)

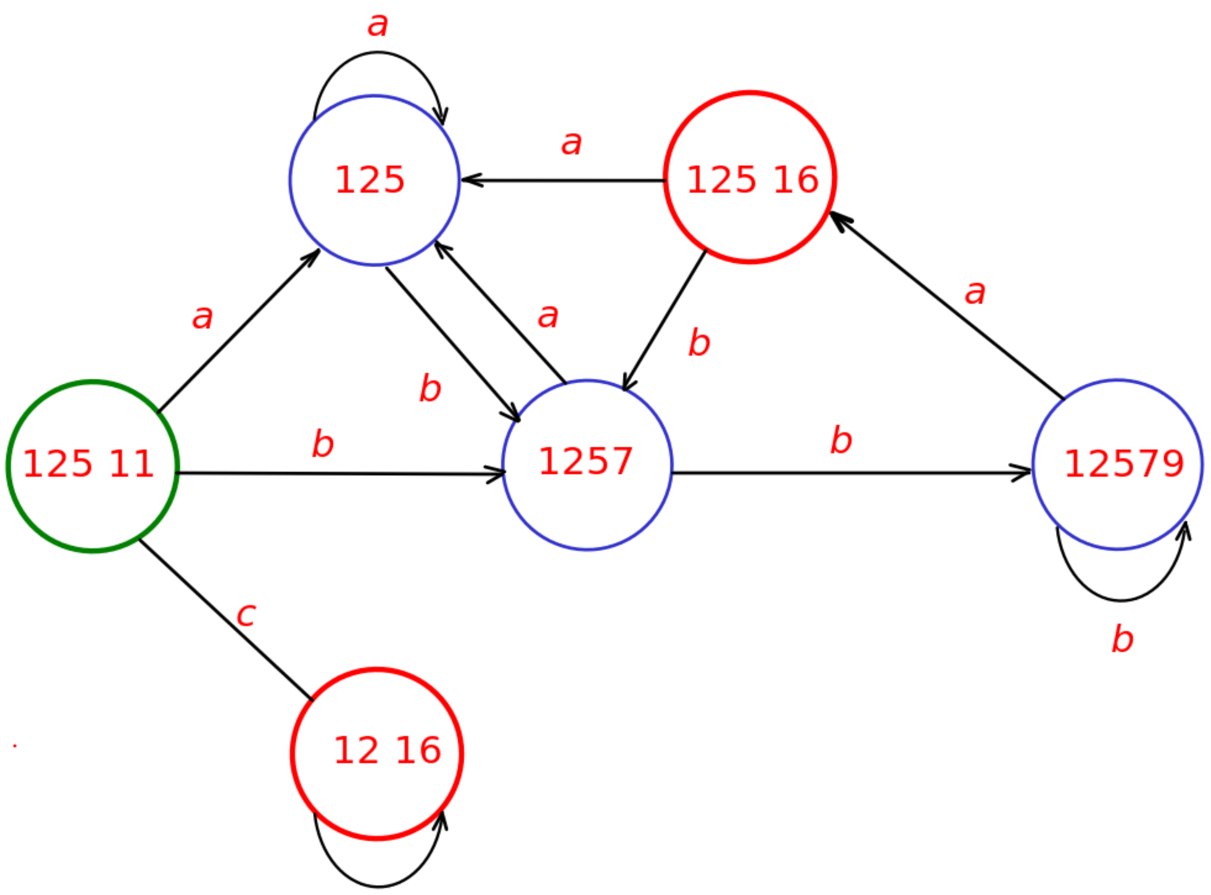
  (5th, 1, 6, 2, 3, 7)

- States should be represented in a compact manner. The entire academic record is not a good representation for our purpose.

- A state should contain just enough information required to decide the future course of computation.

- The state of an object (computational object) is represented by a state variable.

πÖ‡ÎÌ§¶‡Î|¶[ú›[∞Îr[»Ì¯›ÖÎÌ§ÖÎ≥πÌ¶π≥ÎπîÎ[Î‡Ì[ÌÖ”



This graph is for ((a|b)*bba | c+)

12 16 -- is a state of having seen one or more c's. What does it decide?
1257 -- is a state of having seen any string of a's and b's but ending with a single b.
125 -- is a state of having seen any string of a's and b's but ending with a a.

A fundamental feature of imperative programs is states changing with time. This is done through an assignment statement (set!) that changes the state variable.

Here is an example of simple account making procedure:

```
(define (make-account balance)
    (lambda (amount)
      (if (>= balance amount)
          (begin (set! balance (- balance amount))
                 balance)
          "Insufficient funds")))
```

Makes:
1. An account per invocation.
2. Maintains separate balances for each invocation.

```
(define W1 (make-account 100))
(define W2 (make-account 100))
(W1 50)
50
(W2 70)
30
(W2 40)
"Insufficient funds"
(W1 40)
10
```

W1 and W2 are independent objects with their local states

We augment accounts with the ability to do more things:

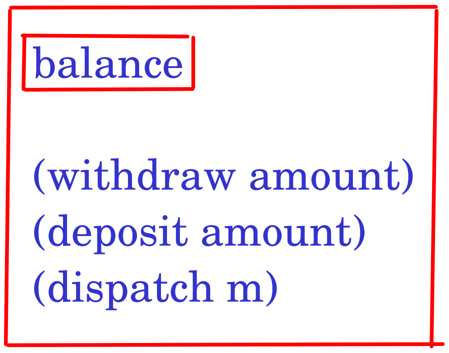
```scheme
(define (make-account balance)

  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))

  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)

  (define (dispatch m)
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else (error "Unknown request -- MAKE-ACCOUNT"
                       m))))

dispatch)
```



A bird's eye view
of the account object

Exercise 3.1. An accumulator is a procedure that is called repeatedly with a single numeric argument and accumulates its arguments into a sum. Each time it is called, it returns the currently accumulated sum. Write a procedure make-accumulator that generates accumulators, each maintaining an independent sum. The input to make-accumulator should specify the initial value of the sum; for example:

```
(define A (make-accumulator 5))
(A 10)
15
(A 10)
25
```

```
(define (make-accumulator a)
  (lambda (m)
    (begin
      (set! a (+ m a))
      a)))
```

Exercise 3.2. In software-testing applications, it is useful to be able to count the number of times a given procedure is called during the course of a computation. Write a procedure make-monitored that takes as input a procedure, f, that itself takes one input. The result returned by make-monitored is a third procedure, say mf, that keeps track of the number of times it has been called by maintaining an internal counter. If the input to mf is the special symbol how-many-calls?, then mf returns the value of the counter. If the input is the special symbol reset-count, then mf resets the counter to zero. For any other input, mf returns the result of calling f on that input and increments the counter. For instance, we could make a monitored version of the sqrt procedure:

```
(define ms (make-monitored sqrt))
(ms 100)
10
(ms 'how-many-calls?)
1
```

```
(define (make-monitored f)
  (define count 0)
  (lambda (n)
    (cond [(eq? n 'how-many-calls) count]
          [else (begin
                  (set! count (+ count 1))
                  (f n))])))
```

Does not work for recursive procedures. Why?

Benefits of introducing states

Consider the function for displaying a general tree:

```
(struct gnode (mass posn subtrees) #:transparent)

(define (display-tree t)
  (display-tree-helper t 0))

(define (display-tree-helper t indent)
  (cond [(null? (gnode-subtrees t))
         (begin
           (display "(leaf ")
           ...]
        [else (display "(gnode ")
              ...
              (newline)
              (printblanks (+ indent 2))
              (display "[")
              (display-list (gnode-subtrees t) (+ indent 3))
              (display "])")]))

(define (display-list lt indent)
  (begin
    (cond [(not (null? lt))
           (begin
             (display-tree-helper (car lt) indent)
             (cond [(not (null? (cdr lt)))
                    (begin
                      (newline)
                      (printblanks indent)
                      (display-list (cdr lt) indent))]))])))
```

There is a cumbersome passing of the indentation level from one procedure to another.

Here is the same program with the indentation level modeled as a state.

```
(define indent 0)

(define (display-tree t)
  (display-tree-helper t))

(define (display-tree-helper t)
  (cond [(null? (gnode-subtrees t))
         (begin
           (display "(leaf ")
           ...]
        [else (display "(gnode ")
              ...
              (newline)
              (printblanks (+ indent 2))
              (display "[")
              (set! indent (+ indent 3))
              (display-list (gnode-subtrees t))
              (set! indent (- indent 3))
              (display "])")]))

(define (display-list lt)
  (begin
    (cond [(not (null? lt))
           (begin
             (display-tree-helper (car lt))
             (cond [(not (null? (cdr lt)))
                    (begin
                      (newline)
                      (printblanks indent)
                      (display-list (cdr lt)))]))])))
```

A similar example is given in the book in Section 3.1.2

The costs of introducing assignments.

1. The substitution model breaks down:

Consider a simplified version of the make-account procedure:

```
(define (make-simplified-account balance)
    (lambda (amount)
      (set! balance (- balance amount))
            balance))

(define W (make-simplified-account 25))
(W 20)
5
(W 10)
- 5
```

Compare this procedure with the following make-decrementer procedure, which does not use set!:

```
(define (make-decrementer balance)
  (lambda (amount)
     (- balance amount)))

(define D (make-decrementer 25))
(D 20)
5
(D 10)
15
```

We can explain the behavior of make-decrementer using the substitution model:

```
((make-decrementer 25) 20)
((lambda (amount) (- 25 amount)) 20)
(- 25 20)
5
```

Try doing the same thing for make-simplified-account

```
((make-simplied-account 25) 20)
((lambda (amount)
    (set! balance (- 25 amount))
     25) 20)
((set! balance (- 25 20)) 25)                                       )
25
```

Which is not the correct answer.

```
(define (make-simplified-account balance)
    (lambda (amount)
      (set! balance (- balance amount))
            balance))
```

The problem is that the substitution model assumes that all occurrences of a variable have the same value: In particular the occurrence of balance in red and green have the same value. They do not.

2. Equality of expressions cannot be detected syntactically.

```
  (define D1 (make-decrementer 25))
  (define D2 (make-decrementer 25))
```

Are D1 and D2 the same? Yes, because D1 and D2 have the same computational behavior.

```
  (define W1 (make-simplified-account 25))
  (define W2 (make-simplified-account 25))
```

Are W1 and W2 the same? No, because calls to W1 and W2 have distinct effects,

Referential transparency (side-effect free): Syntactically identical expressions in the same scope should evaluate to the same value.

Programs with states are not side-effect free.

Another example:

```
 (define my-account (make-account 100))
 (define your-account (make-account 100))
```

Is not the same as:

```
 (define my-account (make-account 100))
 (define your-account my-account))
```

All this makes understanding programs hard:

```
(define (factorial n)
  (define (iter counter product)
   (if (> counter n)
       product
       (iter (+ counter 1)
             (* counter product)
             )))
  (iter 1 1))
```

```
(define (factorial n)
  (define (iter counter product)
   (if (> counter n)
       product
       (iter (+ counter 1)
             (* counter product)
             )))
  (iter 1 1))
```

Convince yourself that each time iter is executed, the following relations hold:

```
product = (counter-1)!
counter <= n+1
```

```
(define (factorial n)
   (let((product 1)
        (counter 1))
     (define (iter)
        (if (> counter n)
            product
            (begin (set! counter (+ counter 1))
                   (set! product (* counter product))
                   (iter))))
     (iter)))
```

Can you convince yourself of similar relations holding in this case?

```
(define (factorial n)
  (define (iter counter product)
   (if (> counter n)
       product
       (iter (+ counter 1)
             (* counter product)
             )))
  (iter 1 1))
```

Summary:

A state is a summary of the history of the past that decides the course of the computation in the future. It is concretely represented as a mapping from state variables to values.

In the functional style of programming, the only place where a state can change is at the declaration boundaries - a function call, a let declaration.

In the imperative style (C, C++ etc.) the state can be changed anywhere by an assignment.

A state can be global (seen or changed anywhere) or local (seen and changed within a scope).

A lambda with a state "stuck" to it looks like an object in the C++ sense.

Advantage of having states:

1. Some situations are naturally modelled as stateful computations -- e.g. bank-account, academic records.
2. Sometimes the stateless program can be more cumbersome that a corresponding stateful program (eg indentation level, rand-update from the book)

Disadvantages of having states:

1. The easy-to-understand "substitution model" of execution breaks down.
2. The notion of equality (when are two things equal?) becomes harder to capture.
3. The ordering of expressions can change the result of the program.

In general, programs can become hard to understand.