5 Modularity, Objects and State

What's in This Set of Notes ?

5.1 Introduction

- Saw basic elements from which programs are made
- Saw how primitive procedures and primitive data are combined to construct compound entities
- Saw how abstraction is vital to helping us cope with complexity of large systems
- Need strategies to help structure large systems so that they are ***modular*** - they can be divided naturally into parts that can be separately developed and maintained
- We will examine two organization strategies
  - Objects
    - structure application on structure of system being modeled
    - construct corresponding computational objects
    - a large system is a collection of distinct objects whose behaviors may change over time
    - force us to abandon our ***substitutional*** model
    - use a less theoretically tractable ***environment*** model
  - Streams
    - model stream of information that flows into a system
    - decouple simulated time in our model from the order of the events that take place in the computer during evaluation
- Difficulties of dealing with time in computational models
  1. dealing with objects
  2. change
  3. identity

5.2 Assignment and Local State

- View world as populated by independent object each of which has a state that changes over time
- For model to be modular, it should be composed into computation objects that model the actual objects in the system
- Each computation object must have its own ***local state variables*** describing the object's state
- State can change over time

- If we choose to model the flow of time in the system by the

elapsed time in the computer, then we must have a away to construct computation objects whose behaviors change as the program runs

- If we want to model state variables by ordinary symbolic names the we must provide an ***assignment operator*** to enable use to change the value associated with a name

```
(define balance 100)

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

Special Forms

```
(set! <name> <new-value>)
(begin <exp1> <exp2>  ... <expn>)
```

Problem: balance is defined in the global environment

```
(define new-withdraw
  (let (( balance 100))
    (lambda (amount)
      (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
          "Insufficient funds"))))
```

- Let establishes local environment
- Causes problems for our substitution model (see later)
- Maybe we can try to write it again

```
(define (make-withdraw balance)
    (lambda (amount)
      (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
          "Insufficient funds")))
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))
(W1 50) => 50
(W2 70) => 30
(W2 40) => "Insufficient funds"
(W1 40) => 10
```

BankAccount Object

```
(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 method)
     (cond     ((eq? method 'withdraw) withdraw)
               ((eq? method 'deposit) deposit)
               (else (error "Unknown Request -- MAKE
ACCOUNT" method))))
   dispatch)
(define account (make-account 100))

((account 'withdraw) 60) => 40
((account 'withdraw) 60) =>  "Insufficient funds"
((account 'deposit) 50) =>  90
((account 'clear)) =>  "Unknown Request -- MAKE ACCOUNT"
clear
```

Introducing BankAccount Class

```
(define (make-account-class)
        (let ((balance 0))

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

         (define (deposit amount)
             (set-balance (+ (get-balance) amount))
             (get-balance))

         (define (set-balance amount)
                (set! balance amount))

         (define (get-balance)
                balance)

         (define (dispatch method)
             (cond      ((eq? method 'withdraw) withdraw)
                                ((eq? method 'deposit) deposit)
                                ((eq? method 'set-balance) set-balance)
                                ((eq? method 'get-balance) get-balance)
                                (else (error "Unknown Request -- MAKE ACCOUNT" method
         dispatch))

(define (make-account-meta-class)

        (let ((interest-rate 0))

        (define (make-account initial-balance)
                (let ((account (make-account-class)))
                        ((account 'set-balance) initial-balance)
                        account))

        (define (set-interest-rate amount)
```

```
                (set! interest-rate  amount))

        (define (get-interest-rate)
                interest-rate)

        (define (dispatch method)
                (cond ((eq? method 'set-interest-rate) set-interest-rate)
                        ((eq? method 'get-interest-rate) get-interest-rate)
                        ((eq? method 'make-account) make-account)
                        (else (error "Unknown Request -- Account Meta Class" method))
         dispatch))

(define account-meta-class (make-account-meta-class))

(define account1 ((account-meta-class 'make-account) 100))

(define account2 ((account-meta-class 'make-account) 200))
```

Benefits of Introducing Assignment

- Allows us to view a system as collection of objects with local state is a powerful technique for maintaining a modular design
- Allows us to "Remember" things
- Entities don't "leak-out" into other parts of program
- Allows encapsulation (hidden time varying local state)

Cost of Introducing Assignment

- Our program can no longer be interpreted in terms of the substitution model of procedural application
- No simple model with "nice" mathematical properties can be an adequate framework for dealing with objects and assignment in programming languages
- Compare the following procedure using set!

  (define (make-simplified-withdraw balance)

  (lambda (amount)
  (set! balance (- balance amount))
  balance))

  (define W (make-simplified-withdraw 25))
  (W 20) => 5
  (W 10) => -5

- With one that does not use it

  (define (make-decrementer balance)
  (lambda (amount)
  (- balance amount)))
  (define D (make-decrementer 25))
  (D 20) => 5
  (D 10) => 15

- Applying Substitution Model to make-decrementer

  ((make-decrementer 25) 20)

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

- Applying Substitution Model to make-simplified-withdraw

```
((make-simplified-withdraw 25) 20)
    ((lambda (amount) (set! balance (-25 amount)) 25) 20)
```

- Now we apply the operator by substituting 20 for amount in the **body** of the lambda expression

```
(set! balance (-25 20)) 25
(set! balance 5 ) 25
```

- If we adhere to the substitution model, we would have to say that the meaning of the procedure is to first set balance to 5 and then return 25.
- We would have to distinguish between to values for balance which substitution model can't handle
- ***Substitution models assumes symbols are names for values in environment and that they don't change***

Sameness and Change

```
(define w1 (make-simplified-withdraw 25))
(define w2 (make-simplified-withdraw 25))
```

- Are w1 and w2 the same? NO

```
(w1 20) => 5
(w1 20) => -15
(w2 20) => 5
```

- A language that supports the notion that equal things can be substituted for equal things in an expression without changing the value of the expression, that language is said to be ***referentially transparent***.
- Referentially transparent is violated with we include set!

Cost of Assignment

- Can't use substitution model
- Two evaluations of the same function with same arguments can produce different values.
- Procedures are not like math functions anymore
- Programming that makes extensive use of assignment is known as imperative programming
- Questions about computational models
- Programs are susceptible to bugs not found in functional programming because order of assignment is very important

Example of Imperative Style

```
(define (factorial n)
  (let ((product 1)
        (counter 1))
```

```
(define (iterate)
   (if (> counter n)
         product
         (begin
               (set! product (* counter product))
                  // note that the order of these set!
statements is important
               (set! counter (+ counter 1))
               (iterate))))
   (iterate)))
```

Example of Functional Style

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

5.3 Streams

- Assignment causes problems
- Is there another way to model state?
- Where does complexity come from?

Real World	Computer World
objects with local state	computational objects with local variables
time variation	time variation in computer
time variation of state	assignment

- Must we model time with change in state? NO!
- Let x be an object, model change in *x* as function x(t)
- If time measured in discrete steps, then model a time function as a sequence
- Therefore, model change in terms of sequence that represents time history of system being modeled
- Need new data structure called a stream
- Abstract view: Stream is a sequence
- Implementation view: Stream is a list (but we must have list to begin with)
- Need Delayed evaluation
  - Let's us represent very large (even infinite) sequences as streams
- Need more than just simple lists to represent sequences, For example

---

Compute sum of primes in an interval accumulator style

- Store only sum being accumulate

```
(define (sum-primes a b)
 (define (iteration count accumulator)
  (cond ((> count b) accumulator)
```

```
    ((prime? count) (iteration (+ count 1) (+ count accumulator)))
    (else (iteration (+ count 1) accumulator))))
  (iteration a 0))
```

Compute sum of primes in an interval List style

- use list of numbers
- construct interval between a and b
- then filter primes

```
(define (sum-primes a b)
  (accumulate + 0 (filter-prime? (enumerator-iterval a b))))
```

What about (car (cdr (filter-prime? (enumerate-interval 10000 1000000))))?

This expression finds second prime but at what cost?

5.4 Stream Approach

- Similar to lists but without incurring the costs of manipulating sequences as list
- Idea is to construct stream only partially
- Pass the partial construction to the program that consumes the stream
- If consumer access part of stream not constructed, the stream produces just enough to continue
- On surface like lists with different names for procedures to manipulate them

Empty Stream

```
(define the-empty-stream '())
```

Building and accessing parts of a Stream

```
(stream-car (cons-stream x y)) => x
(stream-cdr (cons-stream x y)) => y
```

Find element *n* of stream

```
(define (stream-ref stream n)
    (if (= n 0)
        (stream-car stream )
        (stream-ref (stream-cdr stream ) (- n 1))))
```

Apply procedure to values in stream

```
(define (stream-map procedure stream )
    (if (stream-null? stream )
        the-empty-stream
        (cons-stream (procedure (stream-car stream ))
                     (stream-map procedure (stream-cdr stream
)))))
```

```
(define (stream-null? stream )
    (null? stream))
```

5.5 Stream Implementation

- Need special form (delay <exp>) that returns a delayed object – promise to evaluate <exp> in the future
- Is syntactic sugar for (lambda () <exp>)
- Need (force delayed-object) which perform the evaluation of the expression in the delayed object which is equivalent to (delayed-object)
- Therefore:

```
(define (force delayed-object)
            (delayed-object))
```

- Need new special form (cons-stream <a> <b>) which is equivalent to (cons <a> (delay <b>))
- Therefore stream is implemented as a pair

```
(define (stream-car stream ) (car stream))
(define (stream-cdr stream) (force (cdr stream)))
```

---

prime?

```
(define (has-divisor-in-range? n lower-bound upper-bound)
   (if (< upper-bound lower-bound)
      #f
      (if (divisible? n lower-bound)
        #t
        (has-divisor-in-range? n (+ 1 lower-bound) upper-
bound))))

(define (prime? n)
   (not (has-divisor-in-range? n 2 (- n 1))))

(define (divisible? a b)
   (= (modulo a b) 0))
```

Implementation in action

```
(stream-car
    (stream-cdr
       (stream-filter prime? (stream-enumerate-interval 10000
1000000))))

(define (stream-enumerate-interval low high)
    (if (> low high)
       the-empty-stream
          (cons-stream low (stream-enumerate-interval (+ low 1)
high))))
```

Therefore (stream-enumerate-interval 10000 1000000) =>

```
            (cons 10000 (delay (stream-enumerate-interval 10001
1000000)))
```

If

```
(define (stream-filter predicate stream)
    (cond  ((stream-null? stream) the-empty-stream)
               ((predicate (stream-car stream))
                     (cons-stream (stream-car stream)
                                  (stream-filter predicate (stream-cdr
stream))))
               (else  (stream-filter predicate (stream-cdr stream)))))
```

Then the call to stream-cdr on the delayed object (cons 10000 (delay (stream-enumerate-interval 10001 1000000))) returns the delayed object

(cons 10001 (delay (stream-enumerate-interval 10002 1000000)))

Problem, in some situation you may repeatedly evaluate the same delayed object which is inefficient since you will always return the same result

Solution is to cache result after first evaluation and return it

```
(define (memory-procedure procedure)
    (let  ((already-run? #f)
           (result #f))
        (lambda()
              (if (not already-run?)
                  (begin
                      (set! result (procedure))
                      (set! already-run? #t)
                      result)
                  result))))
```

and then Special Form (delay <exp>) is made equivalent to

(memory-procedure (lambda () <exp>))

5.6 Infinite Streams

Using delay/force to build infinite streams

```
(define (integers-starting-from n)
 (cons-stream n (integers-starting-from (+ n 1))))

(define integers (integers-starting-from 1))
```

- Note, we will only ever be able to work with a portion of the stream

---

Let's build infinite stream with integers not divisible by 7

```
(define (divisible? x y) (= (remainder x y) 0))

(define no-sevens (stream-filter
          (lambda (x) (not (divisible? x 7))) integers))

(stream-ref no-sevens 100) => 117
```

Let's build an infinite stream of Fibonacci numbers

```
(define (fib-generator a b)
        (cons-stream a (fib-generator  b (+ a b))))

(define fibs (fib-generator 0 1))
```

Formulating iterations as stream processes – revisit Newton's method

```
(define (sqrt-improve guess x)
    (average guess (/ x guess)))

(define (average x y ) (/ (+ x y) 2))


(define (sqrt-stream x)
    (define guesses (cons-stream 1.0
                                 (stream-map (lambda (guess) (sqrt-
improve guess x)) guesses)))
    guesses)

(display-stream (sqrt-stream 2))
```

1.0
1.5
1.4166666
….

Using Streams with objects

- Recall

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

- Instead model withdrawal processor as a procedure that takes as input a balance an a stream of amounts to withdraw and produces a stream of successive balances in the account

```
(define (stream-withdraw balance amount-stream)
        (cons-stream balance
                     (stream-withdraw
                                 (- balance (stream-car amount-
stream))
                                 (stream-cdr amount-stream))))


(define amount-stream (cons 10 (delay (cons 10 (delay
'()))))))
(define account (stream-withdraw 100 amount-stream))
(stream-ref account 2) ==> 80
```

  - no assignments
  - no local state
  - no problems – merging multiple amount-streams is difficult