6.001 Recitation 16: Object Oriented Systems I

RI: Gerald Dalley, dalleyg@mit.edu, 11 Apr 2007
http://people.csail.mit.edu/dalleyg/6.001/SP2007/

Announcements

- Project 4 is out.
  - Project 3 was long. Project 4 is longer.
  - Project 4 uses a full-featured object system than we'll use today.
- Quiz 2: 18 April

Gerald's Top 3

1. *Higher-order procedures*
2. *The Environment Model*
3. PROJECT 4*:* making a text adventure game!

New Syntax and Special Forms

Most languages have some way of handling variable-length argument lists because they are extremely convenient. Scheme has a syntax too:

```
(define (mul . args)
```

Note that the (apply proc args) special form takes exactly two subexpressions: (1) a procedure and (2) a list of arguments and applies that procedure to the argument list.

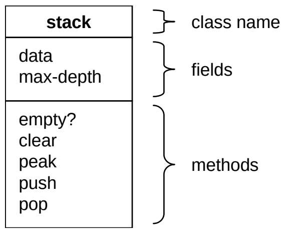
The cond special form is very power and useful, but it can be tedious when you want to do a lot of equality tests. As an alternative, case is available (it works much like a switch statement from C/C++/Java/*etc.*). The following two expressions are equivalent:

```
(cond ((eqv? x 'foo) 'got-foo)
      ((eqv? x 'bar) 'got-bar)
      ((eqv? x 'baz) 'got-baz)
      ((eqv? x   10) 'got-10)
      (else (error 'uh-oh)))

(case x
  ((foo) 'got-foo)
  ((bar) 'got-bar)
  ((baz) 'got-baz)
  ((10)  'got-10)
  (else (error 'uh-oh)))
```

Object-Oriented Stack

We'll build today's examples on an abstraction of a stack object. The graphical representation of our object is given below:



Just to make this class more interesting, we've included a max-depth value–the stack is not allowed to have more than max-depth items in it.

We'll now implement the stack class with the following methods:

- EMPTY? – returns #t if the stack is empty.
- CLEAR – empties the stack of all elements.
- PEEK – returns the top element of the stack, leaving the stack unchanged. If the stack is empty, signal a "stack underflow" error.
- PUSH – adds an element to the top of the stack, if there's room.
- POP – removes and returns the top element of the stack. Remember to program defensively.
- PUSH-ALL-ARGS – pushes each argument onto the stack, *e.g.* (my-stack 'PUSH-ALL-ARGS 1 2 3) is equivalent to pushing 1 then 2 then 3.
- PUSH-ALL-LIST – takes one argument, a list, and pushes each element onto the stack. Think: (my-stack 'PUSH-ALL-ARGS (list 1 2 3)).

Work to do:

1. Write the make-stack constructor.
2. Draw the environment diagram resulting from evaluating (define s (make-stack 10)).
3. Implement a *procedure* pop-all which takes a stack and pops elements off it until it becomes empty, adding each element to the output list.

Bonus: RPN Calculator

Some computing systems use postfix or "reverse polish notation:" values are pushed onto a system stack and standard operators always operate on the top elements of the stack. For example, the following are equivalent:

Type	Expression
infix (18.xxx / C / Java / Matlab)	(10 + 2) / 6
prefix (Scheme)	(/ (+ 10 2) 6)
postfix (Forth)	10 2 + 6 /

Create a 4-function RPN calculator with our stack.

```scheme
; constructor for stack objects with a maximum depth
(define (make-stack ________)
  ; private local state
  (let (                                )

    ; method implementations
    (define (empty?)

      )




















    ; create the message handler that we'll return
    (define (msg-handler msg . args)
      (case msg
        ((empty?)        (empty?))














        (else (error "stacks␣can't␣" msg))))

    ; any additional initialization, parameter checking, etc.




    ; return the message handler
    msg-handler))

(define my-stack (make-stack 10))
(my-stack 'empty?) ; --> #t
(s 'push 10)       ; --> pushed
(my-stack 'empty?) ; --> #f
(s 'push 20)       ; --> pushed
(my-stack 'peek)   ; --> 20
(my-stack 'pop)    ; --> 20
(my-stack 'peek)   ; --> 10
```

Solutions

Announcements

- Project 4 is out.
  - Project 3 was long. Project 4 is longer.
  - Project 4 uses a full-featured object system than we'll use today.
- Quiz 2: 18 April

Gerald's Top 3

1. *Higher-order procedures*
2. *The Environment Model*
3. PROJECT 4*:* making a text adventure game!

New Syntax and Special Forms

Most languages have some way of handling variable-length argument lists because they are extremely convenient. Scheme has a syntax too:

```
(define (mul . args)
  (if (null? args)
      1
      (* (car args) (apply mul (cdr args)))))
(mul 2 3) ; -> 6
(mul (list 2 3)) ; -> error
(mul 2 3 4 5) ; -> 120
(mul) ; -> 1
```

Note that the `(apply proc args)` special form takes exactly two subexpressions: (1) a procedure and (2) a list of arguments and applies that procedure to the argument list.

The `cond` special form is very power and useful, but it can be tedious when you want to do a lot of equality tests. As an alternative, `case` is available (it works much like a `switch` statement from C/C++/Java/*etc.*). The following two expressions are equivalent:

```
(cond ((eqv? x 'foo) 'got-foo)
      ((eqv? x 'bar) 'got-bar)
      ((eqv? x 'baz) 'got-baz)
      ((eqv? x   10) 'got-10)
      (else (error 'uh-oh)))

(case x
  ((foo) 'got-foo)
  ((bar) 'got-bar)
  ((baz) 'got-baz)
  ((10)  'got-10)
  (else (error 'uh-oh)))
```

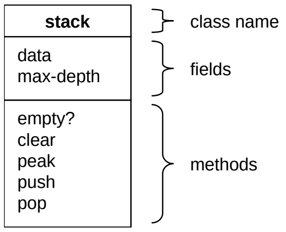
Object-Oriented Stack

In this class, we've been building up higher and higher levels of abstraction. We started with procedures as ways of abstracting computations that we want to repeat. We then used higher-order procedures like `map` to abstract common operations further. We then began building abstract data types. Using tagged data structures, we could build procedures that could operate on many different data types (*e.g.* the addition procedure that operated on numbers, variables, and polynomials).

Often, the set of operations to be performed is very tied to the data type. In such cases, it can make sense to organize our software in a way that collects data and operations on that data together into a single entity.

This is the essence of the object-oriented style of programming.

We'll build today's examples on an abstraction of a stack object. The graphical representation of our object is given below:



Just to make this class more interesting, we've included a `max-depth` value–the stack is not allowed to have more than `max-depth` items in it.

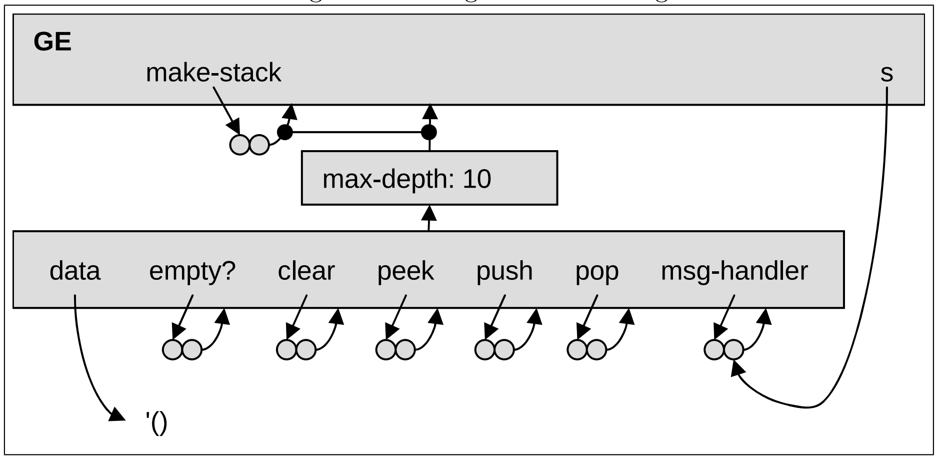
We'll now implement the stack class with the following methods:

- `EMPTY?` – returns `#t` if the stack is empty.
- `CLEAR` – empties the stack of all elements.
- `PEEK` – returns the top element of the stack, leaving the stack unchanged. If the stack is empty, signal a "stack underflow" error.
- `PUSH` – adds an element to the top of the stack, if there's room.
- `POP` – removes and returns the top element of the stack. Remember to program defensively.
- `PUSH-ALL-ARGS` – pushes each argument onto the stack, *e.g.* `(my-stack 'PUSH-ALL-ARGS 1 2 3)` is equivalent to pushing 1 then 2 then 3.
- `PUSH-ALL-LIST` – takes one argument, a list, and pushes each element onto the stack. Think: `(my-stack 'PUSH-ALL-ARGS (list 1 2 3))`.

To do so, we'll need to create a constructor that has some initialization arguments, creates additional internal state, defines method implementations, creates a message dispatcher, performs any additional initialization, and returns the dispatcher as our object representation.

Work to do:

1. Write the `make-stack` constructor.
   see solution at the end
2. Draw the environment diagram resulting from evaluating `(define s (make-stack 10))`.



3. Implement a *procedure* `pop-all` which takes a stack and pops elements off it until it becomes empty, adding each element to the output list.
   see solution at the end

Bonus: RPN Calculator

Some computing systems use postfix or "reverse polish notation:" values are pushed onto a system stack and standard operators always operate on the top elements of the stack. For example, the following are equivalent:

Type	Expression
infix (18.xxx / C / Java / Matlab)	`(10 + 2) / 6`
prefix (Scheme)	`(/ (+ 10 2) 6)`
postfix (Forth)	`10 2 + 6 /`

This notation is compact, lends itself to some very simple and efficient hardware implementations, doesn't require (or even allow) any parentheses, and is even more obtuse than prefix notation ☺.

Create a 4-function RPN calculator with our stack.

Solution Code

```
(define (make-stack max-depth)
  ; This constructor creates stack instances.  The stack can have at most max-depth
  ; elements in it.

  ; private local state
  (let ((data '()))

    ; method implementations
    (define (empty?)
      (null? data))
    (define (clear)
      (set! data '())
      'cleared)
    (define (peek)
      (if (empty?)
          (error "stack␣underflow")
          (car data)))
    (define (push elm)
      (if (> (+ (length data) 1) max-depth)
          (error "The␣maximum␣number␣of␣elements␣this␣stack␣can␣hold␣is" max-depth)
          (set! data (cons elm data)))
      'pushed)
    (define (pop)
      (let ((return-value (peek)))
        (set! data (cdr data))
        return-value))

    ; create the message handler that we'll return
    (define (msg-handler msg . args)
      (case msg
        ((empty?)        (empty?))
        ((clear)         (clear))
        ((peek)          (peek))
        ((push)          (push (car args)))
        ((pop)           (pop))
        ((push-all-args) (for-each push args))
        ((push-all-list) (for-each push (car args)))
        (else (error "stacks␣can't␣" msg))))

    ; any additional initialization, parameter checking, etc.
    (if (< max-depth 0)
        (error "max-depth␣must␣be␣non-negative,␣not:" max-depth))

    ; return the message handler
    msg-handler))

(define (pop-all stack)
  (if (stack 'empty?)
      '()
      (let ((val (stack 'POP)))
        ; Tricky question: What's (potentially) wrong with getting rid
        ; of the entire let expression and replacing it with the following?
        ;   (cons (stack 'POP) (pop-all stack))
        ; Ask your TA or recitation instructor if you're not sure.
        (cons val
              (pop-all stack)))))

(define (reverse-using-stack lst)
  (let ((stack (make-stack (length lst))))
    (stack 'PUSH-ALL-LIST lst)
    (pop-all stack)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Tests your recitation instructor used to make sure the code actually
;; worked correctly.

; Complains if val and expected are not equal?
; See notes at the end if you want to understand the implementation.
(define (assert-equal val expected)
  (if (not (equal? val expected))
      (raise (list "expected␣" expected ",␣but␣got␣" val))))

; Complains if evaluating expr does not result in a complaint.
; See notes at the end if you want to understand the implementation.
(define (assert-throws expr)
  ; This assert evaluates the given expression and then makes sure that
  ; it raises an exception.
  ; ...black magic lies herein
  (let ((exception-thrown #f))
    (with-handlers (((lambda (exn) #t) ; filter all exceptions
                     (lambda (exn)
                       ; Uncomment the next line to have suppressed exceptions
                       ; get printed anyway.
                       ;(display (vector-ref (struct->vector exn) 1)) (newline)
                       (set! exception-thrown #t))))
      (eval expr)) ; try evaluating
    (if (not exception-thrown)
        (error "The␣following␣expression␣should␣have␣raised␣an␣exception:" expr))))

;(assert-throws '(+ 1 2)) ; uncomment this line to test assert-throws: (+ 1 2) is
;                           a valid expression, so it will fail to throw an exception,
;                           and thus assert-throws will complain.

; Test stack methods
(define s (make-stack 10))
(assert-equal (s 'empty?) #t)
(s 'push 10)
(assert-equal (s 'empty?) #f)
(s 'push 20)
(assert-equal (s 'empty?) #f)
(assert-equal (s 'peek) 20)
(assert-equal (s 'peek) 20)
(assert-equal (s 'pop) 20)
(assert-equal (s 'peek) 10)
(s 'clear)
(assert-equal (s 'empty?) #t)
(s 'push-all-list '(1 2 3 4 5))
(assert-equal (s 'peek) 5)
(s 'clear)
(assert-equal (s 'empty?) #t)
(s 'push-all-args '(1 2 3 4 5))
(assert-equal (s 'peek) '(1 2 3 4 5))

; Test pop-all procedure
(s 'clear)
```

```
(s 'push-all-args 1 2 3 4 5)
(assert-equal (pop-all s) '(5 4 3 2 1))
(assert-equal (s 'empty?) #t)

; Test reverse-using-stack
(define lst '(1 2 3 4 5 6 7 8 9 10))
(assert-equal (reverse-using-stack lst) (reverse lst))

; Make sure errors are generated appropriately.
(define s (make-stack 10))
(assert-throws '(s 'push-all-list '(1 2 3 4 5 6 7 8 9 10 11)))
(assert-throws '(define s (make-stack -10))) ; --> error



; You do *not* need to understand the following for this course, but it's
; provided here for the curious...
;
; Explanation of what all this (raise ...) and (with-handlers ...) stuff is:
;   When you use the (error ...) procedure, an error is "thrown" or "raised."
;   This is an object.  By default, this object is caught by a global error
;   handler that prints out the error message with the little ladybug and so
;   forth.  It's possible to catch an error in your own code as well.  This is
;   done with the with-handlers special form in DrScheme.  Its first parameter
;   is a list of two procedures.  The first is a predicate that tells DrScheme
;   which errors we want catch.  The second element in the list is a procedure
;   that's called if an error is thrown and the predicate returned #t (or
;   anything other than #f probably).  The remaining arguments to with-handlers
;   are its body.  The body is evaluated.  If any errors are raised, execution
;   stops and the handler predicate is used.  If #f is returned, the next
;   enclosing with-handlers is looked at.  In this case, we just have our
;   with-handlers and the default one that's implicitly used for all evaluations.
;   For generating errors, I happened to use (raise ...) instead of (error ...)
;   because I had trouble finding documentation on the data type of an error
;   object.  (raise ...) is a more flexible way of generating the objects.
;
;   assert-throws is a little procedure that makes sure that evaluating its
;   argument will throw an error.  If an error is thrown by evaluating its
;   argument, it returns silently.  If an error is not thrown, it complains.
;   Since it is supposed make sure errors are generated and suppress them, quote
;   the expression in question instead of evaluating it in the global environment.
;   (eval...) is a special form that takes an object and evaluates it using the
;   interpreter.  We'll be doing stuff like this this week in class.
```

```
;;; This reverse polish notation (postfix) calculator is an interesting
;;; use of our stack object.

(load "stack.scm")

(define (make-calculator)
  ; an RPN calculator
  (let ((stack (make-stack 100)))

    (define (eval-binary-op op)
      ; This helper handles the stack manipulation and evaluation of binary
      ; RPN operations
      (let ((second-arg (stack 'POP))
            (first-arg  (stack 'POP)))
        ; For non-commutative operations, we need to reverse the arguments so
        ; that, for example 5, then 2, then - gives (- 5 2)
      (stack 'push (op first-arg second-arg))))

    (lambda (message . args)
      ; The procedure created by this lambda is returned by make-calculator.
      ; It's a message handler with a variable-length argument list.  Note:
      ; In order to focus on the basic concepts, we haven't bothered checking
      ; the length of args (e.g. for a CLEAR message args should be nil, for
      ; NUMBER-INPUT args should be a list of exactly one number.
      (case message
        ((ANSWER)
         (if (stack 'EMPTY?)
             'empty-stack
             (stack 'peek)))
        ((CLEAR)
         (stack 'clear)
         'cleared)
        ((NUMBER-INPUT) (stack 'PUSH (car args)))
        ((ADD) (eval-binary-op +))
        ((SUB) (eval-binary-op -))
        ((MUL) (eval-binary-op *))
        ((DIV) (eval-binary-op /))
        (else (error "calculator␣doesn t" message))))))

(define c (make-calculator))

(c 'ANSWER)              ; empty-stack
(c 'NUMBER-INPUT 4)      ; pushed
(c 'ANSWER)              ; 4
(c 'NUMBER-INPUT 5)      ; pushed
(c 'ANSWER)              ; 5
(c 'ADD)                 ; pushed
(c 'ANSWER)              ; 9
(c 'NUMBER-INPUT 7)      ; pushed
(c 'SUB)                 ; pushed
(c 'ANSWER)              ; 2
(c 'CLEAR)               ; cleared
(c 'ANSWER)              ; empty-stack
(c 'ANSWER-MY-HOMEWORK) ; error
```