```
;;; OBJSYS.SCM
;;;
;;; MIT 6.001                                   Spring, 2005
;;;
;;; This file provides a basic object system, and
;;; a clock for objects in a simulation world.  Additional
;;; utility procedures are also provided.

;; Some terminology:
;;
;; "instance" of an object -- each individual object has
;; its own identity. The instance knows its type, and has
;; a message handler associated with it. One can "ask" an
;; object to do something, which will cause the object
;; to use the message handler to look for a method to
;; handle the request and then invoke the method on the
;; arguments.
;;
;; "make" an object message handler -- makes a new message
;; handler to inherit the state information and methods of the
;; specified class. The message handler is not a full "object
;; instance" in our system; the message handler needs to be
;; part of an instance object (or part of another message
;; handler that is part of an instance object). All
;; procedures that define class objects should take a self pointer (a
;; pointer to the enclosing instance) as the first argument.
;;
;; "create" an object -- this does three things: it makes
;; a new instance of the object, it makes and sets the
;; message handler for that instance, and finally it INSTALL's
;; that new object into the world.
;;
;; "install" an object -- this is a method in the object, by
;; which the object can initialize itself and insert itself
;; into the world by connecting itself up with other related
;; objects in the world.

;;-------------------------------------------------------------
;; Instance

; instance is a tagged data structure which holds the "self" of a normal
; object instance.  It passes all messages along to the handler
; procedure that it contains.
;

(define (make-instance)
  (list 'instance #f))

(define (instance? x)
  (and (pair? x) (eq? (car x) 'instance)))

(define (instance-handler instance) (cadr instance))

(define (set-instance-handler! instance handler)
  (set-car! (cdr instance) handler))

(define (create-instance maker . args)
  (let* ((instance (make-instance))
         (handler (apply maker instance args)))
    (set-instance-handler! instance handler)
    (if (method? (get-method 'INSTALL instance))
        (ask instance 'INSTALL))
    instance))

;;-------------------------------------------------------------
;; Handler
; handler is a procedure which responds to messages with methods
; it automatically implements the TYPE and METHODS methods.

(define (make-handler typename methods . super-parts)
  (cond ((not (symbol? typename))    ;check for possible programmer errors
         (error "bad typename" typename))
        ((not (method-list? methods))
         (error "bad method list" methods))
        ((and super-parts (not (filter handler? super-parts)))
         (error "bad part list" super-parts))
        (else
         (named-lambda (handler message)
           (case message
             ((TYPE)
              (lambda () (type-extend typename super-parts)))
             ((METHODS)
              (lambda ()
                (append (method-names methods)
                        (append-map (lambda (x) (ask x 'METHODS))
                                    super-parts))))
             (else
              (let ((entry (method-lookup message methods)))
                (if entry
                    (cadr entry)
                    (find-method-from-handler-list message super-parts)))))))))

(define (handler? x)
  (and (compound-procedure? x)
       (eq? 'handler (lambda-name (procedure-lambda x)))))

(define (->handler x)
  (cond ((instance? x)
         (instance-handler x))
        ((handler? x)
         x)
        (else
         (error "I don't know how to make a handler from" x))))

; builds a list of method (name,proc) pairs suitable as input to make-handler
; note that this puts a label on the methods, as a tagged list

(define (make-methods . args)
  (define (helper lst result)
    (cond ((null? lst) result)

          ; error catching
          ((null? (cdr lst))
           (error "unmatched method (name,proc) pair"))
          ((not (symbol? (car lst)))
           (if (procedure? (car lst))
```

```
              (pp (car lst)))
           (error "invalid method name" (car lst)))
          ((not (procedure? (cadr lst)))
           (error "invalid method procedure" (cadr lst)))

          (else
           (helper (cddr lst) (cons (list (car lst) (cadr lst)) result)))))
  (cons 'methods (reverse (helper args '()))))

(define (method-list? methods)
  (and (pair? methods) (eq? (car methods) 'methods)))

(define (empty-method-list? methods)
  (null? (cdr methods)))

(define (method-lookup message methods)
  (assq message (cdr methods)))

(define (method-names methods)
  (map car (cdr methods)))

;;------------------------------------------------------------
;; Root Object

; Root object.  It contains the IS-A method.
; All classes should inherit (directly or indirectly) from root.
;
(define (root-object self)
  (make-handler
   'root
   (make-methods
    'IS-A
    (lambda (type)
      (memq type (ask self 'TYPE))))))

;;------------------------------------------------------------
;; Object Interface

; ask
;
; We "ask" an object to invoke a named method on some arguments.
;
(define (ask object message . args)
  ;; See your Scheme manual to explain `. args' usage
  ;; which enables an arbitrary number of args to ask.
  (let ((method (get-method message object)))
    (cond ((method? method)
           (apply method args))
          (else
           (error "No method for" message 'in
                  (safe-ask 'UNNAMED-OBJECT
                            object 'NAME))))))

; Safe (doesn't generate errors) method of invoking methods
; on objects.  If the object doesn't have the method,
; simply returns the default-value.  safe-ask should only
; be used in extraordinary circumstances (like error handling).
;
(define (safe-ask default-value obj msg . args)
  (let ((method (get-method msg obj)))
    (if (method? method)
        (apply ask obj msg args)
        default-value)))

;;--------------------
;; Method Interface
;;
;; Objects have methods to handle messages.

; Gets the indicated method from the object or objects.
; This procedure can take one or more objects as
; arguments, and will return the first method it finds
; based on the order of the objects.
;
(define (get-method message . objects)
  (find-method-from-handler-list message (map ->handler objects)))

(define (find-method-from-handler-list message objects)
  (if (null? objects)
      (no-method)
      (let ((method ((car objects) message)))
        (if (not (eq? method (no-method)))
            method
            (find-method-from-handler-list message (cdr objects))))))

(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else (error "Object returned this non-message:" x))))

(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))

; used in make-handler to build the TYPE method for each handler
;
(define (type-extend type parents)
  (cons type
        (remove-duplicates
         (append-map (lambda (parent) (ask parent 'TYPE))
                     parents))))

;;------------------------------------------------------------
;; Utility procedures

(define (random-number n)
  ;; Generate a random number between 1 and n
  (+ 1 (random n)))

(define (pick-random lst)
  (if (null? lst)
      #F
      (list-ref lst (random (length lst)))))

(define (find-all source type)
  (filter (lambda (x) (ask x 'IS-A type))
```

```
         (ask source 'THINGS)))

(define (delq item lst)
  (cond ((null? lst) '())
        ((eq? item (car lst)) (delq item (cdr lst)))
        (else (cons (car lst) (delq item (cdr lst))))))

(define (filter predicate lst)
  (cond ((null? lst) '())
        ((predicate (car lst))
         (cons (car lst) (filter predicate (cdr lst))))
        (else (filter predicate (cdr lst)))))

(define (fold-right op init lst)
  (if (null? lst)
      init
      (op (car lst)
          (fold-right op init (cdr lst)))))

(define (remove-duplicates lst)
  (if (null? lst)
      '()
      (cons (car lst)
            (remove-duplicates (filter (lambda (x)
                                         (not (eq? x (car lst))))
                                       lst)))))

;; utility for finding all the people in the world
(define (all-people)
  (append-map (lambda (room) (find-all room 'person)) all-rooms))


;;-------------------------------------------------------------
;; Support for Objects in a Simulation World

;;--------------------
;; Clock
;;
;; A clock is an object with a notion of time, which it
;; imparts to all objects that have asked for it.  It does
;; this by invoking a list of CALLBACKs whenever the TICK
;; method is invoked on the clock.  A CALLBACK is an action to
;; invoke on each tick of the clock, by sending a message to an object

(define (clock self . args)
  (let ((root-part (root-object self))
        (name (if (not (null? args))
                  (car args)
                  'THE-CLOCK))
        (the-time 0)
        (callbacks '())
        (removed-callbacks '()))
    (make-handler
     'clock
     (make-methods
      'INSTALL
      (lambda ()
        ;; By default print out clock-ticks
        ;; -- note how we are adding a callback
        ;;    to a method of the clock object
        (ask self 'ADD-CALLBACK
             (create-clock-callback 'tick-printer self 'PRINT-TICK)))
      'NAME      (lambda () name)
      'THE-TIME  (lambda () the-time)
      'RESET     (lambda ()
                   (set! the-time 0)
                   (set! callbacks '()))
      'TICK
      (lambda ()
        (set! removed-callbacks '())
        (for-each (lambda (x)
                    (if (not (memq x removed-callbacks))
                        (ask x 'activate)))
                  (reverse callbacks))
        (set! the-time (+ the-time 1)))
      'ADD-CALLBACK
      (lambda (cb)
        (cond ((not (ask cb 'IS-A 'CLOCK-CALLBACK))
               (error "Non callback provided to ADD-CALLBACK"))
              ((null? (filter (lambda (x) (ask x 'SAME-AS? cb))
                              callbacks))
               (set! callbacks (cons cb callbacks))
               'added)
              (else
               'already-present)))
      'REMOVE-CALLBACK
      (lambda (obj cb-name)
        (set! callbacks
              (filter (lambda (x)
                        (cond ((and (eq? (ask x 'NAME) cb-name)
                                    (eq? (ask x 'OBJECT) obj))
                               (set! removed-callbacks
                                     (cons x removed-callbacks))
                               #f)
                              (else #t)))
                      callbacks))
        'removed)
      'PRINT-TICK
      ;; Method suitable for a callback that prints out the tick
      (lambda ()
        (ask screen 'TELL-WORLD
             (list "---" (ask self 'NAME) "Tick" (ask self 'THE-TIME) "---"))))
     root-part)))

(define (create-clock . args)
  (apply create-instance clock args))

;; Clock callbacks
;;
;; A callback is an object that stores a target object,
;; message, and arguments.  When activated, it sends the target
;; object the message.  It can be thought of as a button that executes an
;; action at every tick of the clock.

(define (clock-callback self name object msg . data)
  (let ((root-part (root-object self)))
```

```
    (make-handler
     'clock-callback
     (make-methods
      'INSTALL  (lambda () 'INSTALLED)
      'NAME     (lambda () name)
      'OBJECT   (lambda () object)
      'MESSAGE  (lambda () msg)
      'ACTIVATE (lambda () (apply ask object msg data))
      'SAME-AS? (lambda (cb)
                  (and (ask cb 'IS-A 'CLOCK-CALLBACK)
                       (eq? (ask self 'NAME)
                            (ask cb 'NAME))
                       (eq? object (ask cb 'OBJECT)))))
     root-part)))

(define (create-clock-callback name object msg . data)
  (apply create-instance clock-callback name object msg data))

;; Setup global clock object
(define clock (create-clock))

;; Get the current time
(define (current-time)
  (ask clock 'THE-TIME))

;; Advance the clock some number of ticks
(define (run-clock n)
  (cond ((= n 0) 'DONE)
        (else (ask clock 'tick)
              ;; remember that this activates each item in callback list
              (run-clock (- n 1)))))

;; Using the clock:
;;
;; When you want the object to start being aware of the clock
;; (during initialization of autonomous-person, for example),
;; add a callback to the clock which activates a method on the
;; object:
;; (ask clock 'ADD-CALLBACK
;;      (create-clock-callback 'thingy self 'DO-THINGY))
;; The first argument is a name or descriptor of the callback.
;; The second argument is the object to which to send the message.
;; The third argument is the message (method-name) to send.
;; Additional arguments can be provided and they are sent to
;; the object with the message when the callback is activated.
;; In this case, the method do-thingy should be descriptive of
;; the behavior the object will exhibit when time passes.
;; When the object's lifetime expires (sometimes this is taken
;; literally!), it should remove its callback(s) from the clock.
;; This can be done with
;; (ask clock 'REMOVE-CALLBACK
;;      self 'thingy)
;;
;; An example of using callback names and additional arguments:
;; (ask clock 'ADD-CALLBACK
;;      (create-clock-callback 'whoopee me 'SAY '("Whoopee!")))
;; (ask clock 'ADD-CALLBACK
;;      (create-clock-callback 'fun me 'SAY '("I am having fun!")))
;; This causes the avatar to say two things every time the clock
;; ticks.

;;-----------
;; screen
;;
;; This is a singleton object (only one object of this type in
;; existence at any time), which deals with outputting text to
;; the user.
;;
;; If the screen is in deity-mode, the user will hear every message,
;; regardless of the location of the avatar.  If deity-mode is
;; false, only messages sent to the room which contains the avatar
;; will be heard.
;;
;; network-mode is something set only by the network code.

(define (screen self)
  (let ((deity-mode #t)
        (network-mode #f)
        (me #f)
        (root-part (root-object self)))
    (make-handler
     'screen
     (make-methods
      'TYPE   (lambda () (type-extend 'screen root-part))
      'NAME   (lambda () 'THE-SCREEN)
      'SET-ME (lambda (new-me) (set! me new-me))
      'TELL-ROOM    (lambda (room msg)
                      (if (or deity-mode
                              (eq? room (safe-ask #f me 'location)))
                          (if network-mode
                              (display-net-message msg)
                              (display-message msg))))
      'TELL-WORLD   (lambda (msg)
                      (if network-mode
                          (display-net-message msg)
                          (display-message msg)))
      'DEITY-MODE   (lambda (value) (set! deity-mode value))
      'NETWORK-MODE (lambda (value) (set! network-mode value))
      'DEITY-MODE?  (lambda () deity-mode))
     root-part)))

(define screen
  (create-instance screen))

;;--------------------
;; Utilities for our simulation world
;;

(define (display-message list-of-stuff)
  (if (not (null? list-of-stuff)) (newline))
  (for-each (lambda (s) (display s) (display " "))
            list-of-stuff)
  'MESSAGE-DISPLAYED)

(define (display-net-message list-of-stuff)
  (for-each (lambda (s) (display s server-port) (display " " server-port))
```

```
           list-of-stuff)
  (display #\newline server-port)
  (flush-output server-port)
  'MESSAGE-DISPLAYED)

; Grab any kind of thing from avatar's location,
; given its name.  The thing may be in the possession of
; the place, or in the possession of a person at the place.
; THING-NAMED SHOULD NEVER BE USED IN OBJTYPES OR ANY OBJECT
; YOU CREATE.
(define (thing-named name)
  (let* ((place (ask me 'LOCATION))
         (things (ask place 'THINGS))
         (peek-stuff (ask me 'PEEK-AROUND))
         (my-stuff (ask me 'THINGS))
         (all-things (append things (append my-stuff peek-stuff)))
         (things-named (filter (lambda (x) (eq? name (ask x 'NAME)))
                               all-things)))
    (cond ((null? things-named)
           (error "In here there is nothing named" name))
          ((null? (cdr things-named))   ; just one thing
           (car things-named))
          (else
           (display-message (list "There is more than one thing named"
                                  name "here. Picking one of them."))
           (pick-random things-named)))))



;;--------------------
;; show
;;
;; Some utilities.
;;
;; Treat these as gifts from the (Scheme) Gods.
;; Don't try to understand these procedures!

(load-option 'format)

(define (show obj)
  (define (show-guts obj)
    (format #t "INSTANCE ~A~% TYPE: ~A~%" obj (ask obj 'TYPE))
    (show-handler (->handler obj))
    'instance)
  (if (instance? obj)
      (show-guts obj)
      (show-handler obj)))

(define (show-handler proc)
  (define (show-frame frame depth)
    (define *max-frame-depth* 1)
    (if (global-environment? frame)
        (display (env-name frame))
        (let* ((bindings (environment-bindings frame))
               (parent   (environment-parent frame))
               (names    (cons "Parent frame"
                               (map symbol->string (map car bindings))))
               (values   (cons (env-name parent)
                               (map cadr bindings)))
               (width    (reduce max 0 (map string-length names))))
          (for-each (lambda (n v) (pp-binding n v width depth))
                    names values)
          (if (and (not (global-environment? parent))
                   (< depth *max-frame-depth*))
              (show-frame parent (+ depth 1))))))
  (define (global-environment? frame)
    (environment->package frame))
  (define (env-name env)
    (if (global-environment? env) 'GLOBAL-ENVIRONMENT env))
  (define (pp-binding name value width depth)
    (let ((value* (with-string-output-port
                    (lambda (port)
                      (if (pair? value)
                          (pretty-print value port #F (+ width 2))
                          (display value port))))))
      (display-spaces (* 2 (+ depth 1)))
      (display name) (display ": ")
      (display (make-string (- width (string-length name)) #\Space))
      (if (pair? value)
          (display (substring value* (+ width 2) (string-length value*)))
          (display value*))
      (newline)))
  (define (display-spaces num)
    (if (> num 0) (begin (display " ") (display-spaces (- num 1)))))
  (if (handler? proc)
      (fluid-let ((*unparser-list-depth-limit* 5)
                  (*unparser-list-breadth-limit* 6))
        (let ((methods (environment-lookup (procedure-environment proc)
                                           'methods))
              (parts   (environment-lookup (procedure-environment proc)
                                           'super-parts))
              (type    (environment-lookup (procedure-environment proc)
                                           'typename)))
          (format #t " HANDLER: ~A~%" proc)
          (format #t " TYPE: ~A~%" type)
          (format #t "~A~%" (with-output-to-string
                              (lambda () (pretty-print methods))))
          (if (cdr methods)
              (show-frame (procedure-environment (cadadr methods)) 0)
              (format #t " PARENTS: ~A~%" parts))
          ;(display " HANDLER PROCEDURE:\n")
          ;(pretty-print (procedure-lambda proc) (current-output-port) #T 2)
          'handler))
      'not-a-handler))
```

```
;;; OBJTYPES.SCM
;;;
;;; MIT 6.001                                   Spring, 2005
;;;
;;; This file defines object types for use in our simulation
;;; world.  The full world is created in setup.scm.

;;--------------------
;; named-object
;;
;; Named objects are the basic underlying object type in our
;; system. For example, persons, places, and things will all
;; be kinds of (inherit from) named objects.
;;
;; Behavior (messages) supported by all named objects:
;;  - Has a NAME that it can return
;;  - Handles an INSTALL message
;;  - Handles a DESTROY message

(define (create-named-object name)      ; symbol -> named-object
  (create-instance named-object name))

(define (named-object self name)
  (let ((root-part (root-object self)))
    (make-handler
     'named-object
     (make-methods
      'NAME    (lambda () name)
      'INSTALL (lambda () 'installed)
      'DESTROY (lambda () 'destroyed))
     root-part)))

(define (names-of objects)
  ; Given a list of objects, returns a list of their names.
  (map (lambda (x) (ask x 'NAME)) objects))


;;--------------------
;; container
;;
;; A container holds THINGS.
;;
;; This class is not meant for "stand-alone" objects; rather,
;; it is expected that other classes will inherit from the
;; container class in order to be able to contain things.
;; For this reason, there is no create-container procedure.

(define (container self)
  (let ((root-part (root-object self))
        (things '()))
    (make-handler
     'container
     (make-methods
      'THINGS      (lambda () things)
      'HAVE-THING? (lambda (thing)
                     (not (null? (memq thing things))))
      'ADD-THING   (lambda (thing)
                     (if (not (ask self 'HAVE-THING? thing))
                         (set! things (cons thing things)))
                     'DONE)
      'DEL-THING   (lambda (thing)
                     (set! things (delq thing things))
                     'DONE))
     root-part)))


;;--------------------
;; thing
;;
;; A thing is a named-object that has a LOCATION
;;
;; Note that there is a non-trivial INSTALL here.  What does it do?

(define (create-thing name location)    ; symbol, location -> thing
  (create-instance thing name location))

(define (thing self name location)
  (let ((named-part (named-object self name)))
    (make-handler
     'thing
     (make-methods
      'INSTALL  (lambda ()
                  (ask named-part 'INSTALL)
                  (ask (ask self 'LOCATION) 'ADD-THING self))
      'LOCATION (lambda () location)
      'DESTROY  (lambda ()
                  (ask (ask self 'LOCATION) 'DEL-THING self))
      'EMIT     (lambda (text)
                  (ask screen 'TELL-ROOM (ask self 'LOCATION)
                       (append (list "At" (ask (ask self 'LOCATION) 'NAME))
                               text))))
     named-part)))

;;--------------------
;; mobile-thing
;;
;; A mobile thing is a thing that has a LOCATION that can change.

(define (create-mobile-thing name location)
  ; symbol, location -> mobile-thing
  (create-instance mobile-thing name location))

(define (mobile-thing self name location)
  (let ((thing-part (thing self name location)))
    (make-handler
     'mobile-thing
     (make-methods
      'LOCATION  (lambda () location) ; This shadows message to thing-part!
      'CHANGE-LOCATION
      (lambda (new-location)
        (ask location 'DEL-THING self)
        (ask new-location 'ADD-THING self)
        (set! location new-location))
      'ENTER-ROOM    (lambda () #t)
      'LEAVE-ROOM    (lambda () #t)
      'CREATION-SITE (lambda () (ask thing-part 'location)))
```

```
     thing-part)))

;;--------------------
;; place
;;
;; A place is a container (so things may be in the place).
;;
;; A place has EXITS, which are passages from one place
;; to another.  One can retrieve all of the exits of a
;; place, or an exit in a given direction from place.

(define (create-place name)     ; symbol -> place
  (create-instance place name))

(define (place self name)
  (let ((named-part (named-object self name))
        (container-part (container self))
        (exits '()))
    (make-handler
     'place
     (make-methods
      'EXITS (lambda () exits)
      'EXIT-TOWARDS
      (lambda (direction)
        (find-exit-in-direction exits direction))
      'ADD-EXIT
      (lambda (exit)
        (let ((direction (ask exit 'DIRECTION)))
          (if (ask self 'EXIT-TOWARDS direction)
              (error (list name "already has exit" direction))
              (set! exits (cons exit exits)))
          'DONE)))
     container-part named-part)))

;;------------------------------------------------------------
;; exit
;;
;; An exit leads FROM one place TO another in some DIRECTION.

(define (create-exit from direction to)
  ; place, symbol, place -> exit
  (create-instance exit from direction to))

(define (exit self from direction to)
  (let ((named-object-part (named-object self direction)))
    (make-handler
     'exit
     (make-methods
      'INSTALL
      (lambda ()
        (ask named-object-part 'INSTALL)
        (if (not (null? (ask self 'FROM)))
            (ask (ask self 'FROM) 'ADD-EXIT self)))
      'FROM         (lambda () from)
      'TO           (lambda () to)
      'DIRECTION    (lambda () direction)
      'USE
      (lambda (whom)
        (ask whom 'LEAVE-ROOM)
        (ask screen 'TELL-ROOM (ask whom 'location)
             (list (ask whom 'NAME)
                   "moves from"
                   (ask (ask whom 'LOCATION) 'NAME)
                   "to"
                   (ask to 'NAME)))
        (ask whom 'CHANGE-LOCATION to)
        (ask whom 'ENTER-ROOM)))
     named-object-part)))

(define (find-exit-in-direction exits dir)
  ; Given a list of exits, find one in the desired direction.
  (cond ((null? exits) #f)
        ((eq? dir (ask (car exits) 'DIRECTION))
         (car exits))
        (else (find-exit-in-direction (cdr exits) dir))))

(define (random-exit place)
  (pick-random (ask place 'EXITS)))

;;--------------------
;; person
;;
;; There are several kinds of person:
;;   There are autonomous persons, including vampires, and there
;;   is the avatar of the user.  The foundation is here.
;;
;; A person can move around (is a mobile-thing),
;; and can hold things (is a container). A person responds to
;; a plethora of messages, including 'SAY to say something.
;;

(define (create-person name birthplace) ; symbol, place -> person
  (create-instance person name birthplace))

(define (person self name birthplace)
  (let ((mobile-thing-part (mobile-thing self name birthplace))
        (container-part    (container self))
        (health            3)
        (strength          1))
    (make-handler
     'person
     (make-methods
      'STRENGTH (lambda () strength)
      'HEALTH (lambda () health)
      'SAY
      (lambda (list-of-stuff)
        (ask screen 'TELL-ROOM (ask self 'location)
             (append (list "At" (ask (ask self 'LOCATION) 'NAME)
                           (ask self 'NAME) "says --")
                     list-of-stuff))
        'SAID-AND-HEARD)
      'HAVE-FIT
      (lambda ()
        (ask self 'SAY '("Yaaaah! I am upset!"))
        'I-feel-better-now)
```

```
      'PEOPLE-AROUND        ; other people in room...
      (lambda ()
        (delq self (find-all (ask self 'LOCATION) 'PERSON)))

      'STUFF-AROUND         ; stuff (non people) in room...
      (lambda ()
        (let* ((in-room (ask (ask self 'LOCATION) 'THINGS))
               (stuff (filter (lambda (x) (not (ask x 'IS-A 'PERSON)))
                              in-room)))
          stuff))

      'PEEK-AROUND          ; other people's stuff...
      (lambda ()
        (let ((people (ask self 'PEOPLE-AROUND)))
          (fold-right append '() (map (lambda (p) (ask p 'THINGS)) people))))

      'TAKE
      (lambda (thing)
        (cond ((ask self 'HAVE-THING? thing)  ; already have it
               (ask self 'SAY (list "I am already carrying"
                                    (ask thing 'NAME)))
               #f)
              ((or (ask thing 'IS-A 'PERSON)
                   (not (ask thing 'IS-A 'MOBILE-THING)))
               (ask self 'SAY (list "I try but cannot take"
                                    (ask thing 'NAME)))
               #F)
              (else
               (let ((owner (ask thing 'LOCATION)))
                 (ask self 'SAY (list "I take" (ask thing 'NAME)
                                      "from" (ask owner 'NAME)))
                 (if (ask owner 'IS-A 'PERSON)
                     (ask owner 'LOSE thing self)
                     (ask thing 'CHANGE-LOCATION self))
                 thing))))

      'LOSE
      (lambda (thing lose-to)
        (ask self 'SAY (list "I lose" (ask thing 'NAME)))
        (ask self 'HAVE-FIT)
        (ask thing 'CHANGE-LOCATION lose-to))

      'DROP
      (lambda (thing)
        (ask self 'SAY (list "I drop" (ask thing 'NAME)
                             "at" (ask (ask self 'LOCATION) 'NAME)))
        (ask thing 'CHANGE-LOCATION (ask self 'LOCATION)))

      'GO-EXIT
      (lambda (exit)
        (ask exit 'USE self))

      'GO
      (lambda (direction) ; symbol -> boolean
        (let ((exit (ask (ask self 'LOCATION) 'EXIT-TOWARDS direction)))
          (if (and exit (ask exit 'IS-A 'EXIT))
              (ask self 'GO-EXIT exit)
              (begin (ask screen 'TELL-ROOM (ask self 'LOCATION)
                          (list "No exit in" direction "direction"))
                     #F))))
      'SUFFER
      (lambda (hits perp)
        (ask self 'SAY (list "Ouch!" hits "hits is more than I want!"))
        (set! health (- health hits))
        (if (<= health 0) (ask self 'DIE perp))
        health)

      'DIE          ; depends on global variable "death-exit"
      (lambda (perp)
        (for-each (lambda (item) (ask self 'LOSE item (ask self 'LOCATION)))
                  (ask self 'THINGS))
        (ask screen 'TELL-WORLD
             '("An earth-shattering, soul-piercing scream is heard..."))
        (ask self 'DESTROY))

      'ENTER-ROOM
      (lambda ()
        (let ((others (ask self 'PEOPLE-AROUND)))
          (if (not (null? others))
              (ask self 'SAY (cons "Hi" (names-of others)))))
        #T))
     mobile-thing-part container-part)))

;;--------------------
;; autonomous-person
;;
;; activity determines maximum movement
;; miserly determines chance of picking stuff up

(define (create-autonomous-person name birthplace activity miserly)
  (create-instance autonomous-person name birthplace activity miserly))

(define (autonomous-person self name birthplace activity miserly)
  (let ((person-part (person self name birthplace)))
    (make-handler
     'autonomous-person
     (make-methods
      'INSTALL
      (lambda ()
        (ask person-part 'INSTALL)
        (ask clock 'ADD-CALLBACK
             (create-clock-callback 'move-and-take-stuff self
                                    'MOVE-AND-TAKE-STUFF)))
      'MOVE-AND-TAKE-STUFF
      (lambda ()
        ;; first move
        (let loop ((moves (random-number activity)))
          (if (= moves 0)
              'done-moving
              (begin
                (ask self 'MOVE-SOMEWHERE)
                (loop (- moves 1)))))
        ;; then take stuff
        (if (= (random miserly) 0)
            (ask self 'TAKE-SOMETHING))
        'done-for-this-tick)
```

```
      'DIE
      (lambda (perp)
        (ask clock 'REMOVE-CALLBACK self 'move-and-take-stuff)
        (ask self 'SAY '("SHREEEEK!  I, uh, suddenly feel very faint..."))
        (ask person-part 'DIE perp))
      'MOVE-SOMEWHERE
      (lambda ()
        (let ((exit (random-exit (ask self 'LOCATION))))
          (if (not (null? exit)) (ask self 'GO-EXIT exit))))
      'TAKE-SOMETHING
      (lambda ()
        (let* ((stuff-in-room (ask self 'STUFF-AROUND))
               (other-peoples-stuff (ask self 'PEEK-AROUND))
               (pick-from (append stuff-in-room other-peoples-stuff)))
          (if (not (null? pick-from))
              (ask self 'TAKE (pick-random pick-from))
              #F))))
     person-part)))

;;
;; hall-monitor
;;
(define (create-hall-monitor name birthplace speed irritability)
  (create-instance hall-monitor name birthplace speed irritability))

(define (hall-monitor self name birthplace speed irritability)
  (let ((auto-part (autonomous-person self name birthplace speed 10)))
    (make-handler
     'hall-monitor
     (make-methods
      'INSTALL
      (lambda ()
        (ask auto-part 'INSTALL)
        (ask clock 'ADD-CALLBACK
             (create-clock-callback 'irritate-students self
                                    'IRRITATE-STUDENTS)))
      'IRRITATE-STUDENTS
      (lambda ()
        (if (= (random irritability) 0)
            (let ((people (ask self 'PEOPLE-AROUND)))
              (if people
                  (begin
                    (ask self 'SAY '("What are you doing still up?"
                                     "Everyone back to their rooms!"))
                    (for-each (lambda (person)
                                (ask person 'EMIT
                                     (list (ask person 'NAME) "goes home to"
                                           (ask (ask person 'CREATION-SITE)
                                                'NAME)))
                                (ask person 'CHANGE-LOCATION
                                     (ask person 'CREATION-SITE)))
                              people)
                    'grumped)
                  (ask self 'SAY '("Grrr... When I catch those students..."))))
            (if (ask self 'PEOPLE-AROUND)
                (ask self 'SAY '("I'll let you off this once..."))))))
      'DIE
      (lambda (perp)
        (ask clock 'REMOVE-CALLBACK self 'irritate-students)
        (ask auto-part 'DIE perp)))
     auto-part)))

;;
;; troll
;;
(define (create-troll name birthplace speed hunger)
  (create-instance troll name birthplace speed hunger))

(define (troll self name birthplace speed hunger)
  (let ((auto-part (autonomous-person self name birthplace speed 10)))
    (make-handler
     'troll
     (make-methods
      'INSTALL
      (lambda ()
        (ask auto-part 'INSTALL)
        (ask clock 'ADD-CALLBACK
             (create-clock-callback 'eat-people self
                                    'EAT-PEOPLE)))
      'EAT-PEOPLE
      (lambda ()
        (if (= (random hunger) 0)
            (let ((people (ask self 'PEOPLE-AROUND)))
              (if people
                  (let ((victim (pick-random people)))
                    (ask self 'EMIT
                         (list (ask self 'NAME) "takes a bite out of"
                               (ask victim 'NAME)))
                    (ask victim 'SUFFER (random-number 3) self)
                    'tasty)
                  (ask self 'EMIT
                       (list (ask self 'NAME) "'s belly rumbles"))))
            'not-hungry-now))
      'DIE
      (lambda (perp)
        (ask clock 'REMOVE-CALLBACK self 'eat-people)
        (ask auto-part 'DIE perp)))
     auto-part)))

;;
;; spell
;;
(define (create-spell name location incant action)
  (create-instance spell name location incant action))

(define (spell self name location incant action)
  (let ((mobile-part (mobile-thing self name location)))
    (make-handler
     'spell
     (make-methods
      'INCANT
      (lambda () incant)
      'ACTION
      (lambda () action)
      'USE
      (lambda (caster target)
```

```
       (action caster target)))
    mobile-part)))

(define (clone-spell spell newloc)
  (create-spell (ask spell 'NAME)
                newloc
                (ask spell 'INCANT)
                (ask spell 'ACTION)))

;;--------------------
;; avatar
;;
;; The avatar of the user is also a person.

(define (create-avatar name birthplace)
  ; symbol, place -> avatar
  (create-instance avatar name birthplace))

(define (avatar self name birthplace)
  (let ((person-part (person self name birthplace)))
    (make-handler
     'avatar
     (make-methods
      'LOOK-AROUND          ; report on world around you
      (lambda ()
        (let* ((place (ask self 'LOCATION))
               (exits (ask place 'EXITS))
               (other-people (ask self 'PEOPLE-AROUND))
               (my-stuff (ask self 'THINGS))
               (stuff (ask self 'STUFF-AROUND)))
          (ask screen 'TELL-WORLD (list "You are in" (ask place 'NAME)))
          (ask screen 'TELL-WORLD
               (if (null? my-stuff)
                   '("You are not holding anything.")
                   (append '("You are holding:") (names-of my-stuff))))
          (ask screen 'TELL-WORLD
               (if (null? stuff)
                   '("There is no stuff in the room.")
                   (append '("You see stuff in the room:") (names-of stuff))))
          (ask screen 'TELL-WORLD
               (if (null? other-people)
                   '("There are no other people around you.")
                   (append '("You see other people:") (names-of other-people))))
          (ask screen 'TELL-WORLD
               (if (not (null? exits))
                   (append '("The exits are in directions:") (names-of exits))
                   ;; heaven is only place with no exits
                   '("There are no exits... you are dead and gone to heaven!")))
          'OK))

      'GO
      (lambda (direction)  ; Shadows person's GO
        (let ((success? (ask person-part 'GO direction)))
          (if success? (ask clock 'TICK))
          success?))

      'DIE
      (lambda (perp)
        (ask self 'SAY (list "I am slain!"))
        (ask person-part 'DIE perp)))

     person-part)))
```

```
;;; SETUP.SCM
;;;
;;; MIT 6.001                                   Spring, 2005
;;; PROJECT 4

;;;============================================================================
;;; You can extend this file to extend your world.
;;;============================================================================

;;----------------------------------------------------------------
;; Utils to connect places by way of exits

(define (can-go-both-ways from direction reverse-direction to)
  (create-exit from direction to)
  (create-exit to reverse-direction from))

;;----------------------------------------------------------------
;; Create our world...

(define (create-world)
  ; Create some places
  (let ((10-250 (create-place '10-250))
        (lobby-10 (create-place 'lobby-10))
        (grendels-den (create-place 'grendels-den))
        (barker-library (create-place 'barker-library))
        (lobby-7 (create-place 'lobby-7))
        (eecs-hq (create-place 'eecs-hq))
        (eecs-ug-office (create-place 'eecs-ug-office))
        (edgerton-hall (create-place 'edgerton-hall))
        (34-301 (create-place '34-301))
        (stata-center (create-place 'stata-center))
        (6001-lab (create-place '6001-lab))
        (building-13 (create-place 'building-13))
        (great-court (create-place 'great-court))
        (student-center (create-place 'student-center))
        (bexley (create-place 'bexley))
        (baker (create-place 'baker))
        (legal-seafood (create-place 'legal-seafood))
        (graduation-stage (create-place 'graduation-stage)))

    ; Connect up places
    (can-go-both-ways lobby-10 'up 'down 10-250)
    (can-go-both-ways grendels-den 'up 'down lobby-10)
    (can-go-both-ways 10-250 'up 'down barker-library)
    (can-go-both-ways lobby-10 'west 'east lobby-7)
    (can-go-both-ways lobby-7 'west 'east student-center)
    (can-go-both-ways student-center 'south 'north bexley)
    (can-go-both-ways bexley 'west 'east baker)
    (can-go-both-ways lobby-10 'north 'south building-13)
    (can-go-both-ways lobby-10 'south 'north great-court)
    (can-go-both-ways building-13 'north 'south edgerton-hall)
    (can-go-both-ways edgerton-hall 'up 'down 34-301)
    (can-go-both-ways 34-301 'up 'down eecs-hq)
    (can-go-both-ways 34-301 'east 'west stata-center)
    (can-go-both-ways stata-center 'north 'south stata-center)
    (can-go-both-ways stata-center 'up 'down stata-center)
    (can-go-both-ways eecs-hq 'west 'east eecs-ug-office)
    (can-go-both-ways edgerton-hall 'north 'south legal-seafood)
    (can-go-both-ways eecs-hq 'up 'down 6001-lab)
    (can-go-both-ways legal-seafood 'east 'west great-court)
    (can-go-both-ways great-court 'up 'down graduation-stage)

    ; Create some things
    (create-thing 'blackboard 10-250)
    (create-thing 'lovely-trees great-court)
    (create-thing 'flag-pole great-court)
    (create-mobile-thing 'tons-of-code baker)
    (create-mobile-thing 'problem-set 10-250)
    (create-mobile-thing 'recitation-problem 10-250)
    (create-mobile-thing 'sicp stata-center)
    (create-mobile-thing 'engineering-book barker-library)
    (create-mobile-thing 'diploma graduation-stage)

    (list 10-250 lobby-10 grendels-den barker-library lobby-7
          eecs-hq eecs-ug-office edgerton-hall 34-301 6001-lab
          building-13 great-court stata-center
          student-center bexley baker legal-seafood
          graduation-stage)))

; all spells exist in the chamber-of-stata.  When placing a spell
; in the outside world, the original spell from the chamber-of stata
; is cloned (using clone-spell; see objtypes.scm).
; There are no entrances, exits, or people in the chamber, preventing
;  the spells there from being stolen.
(define (instantiate-spells)
  (let ((chamber (create-place 'chamber-of-stata)))
    (create-spell
     'boil-spell
     chamber
     "habooic katarnum"
     (lambda (caster target)
       (ask target 'EMIT
            (list (ask target 'NAME) "grows boils on their nose"))))
    (create-spell
     'slug-spell
     chamber
     "dagnabbit ekaterin"
     (lambda (caster target)
       (ask target 'EMIT (list "A slug comes out of" (ask target 'NAME)
                               "'s mouth."))
       (create-mobile-thing 'slug (ask target 'LOCATION))))
    chamber))

(define (populate-spells rooms)
  (for-each (lambda (room)
              (clone-spell (pick-random (ask chamber-of-stata 'THINGS)) room))
            rooms))

(define (populate-players rooms)
  (let* ((students (map (lambda (name)
                          (create-autonomous-person name
                                                    (pick-random rooms)
                                                    (random-number 3)
                                                    (random-number 3)))
                        '(ben-bitdiddle alyssa-hacker
                          course-6-frosh lambda-man)))
```

```
;uncomment after writing professors
;        (profs (map (lambda (name)
;                      (create-wit-professor name
;                                            (pick-random rooms)
;                                            (random-number 3)
;                                            (random-number 3)))
;                    '(susan-hockfield eric-grimson)))
         (monitors (map (lambda (name)
                          (create-hall-monitor name
                                               (pick-random rooms)
                                               (random-number 3)
                                               (random-number 3)))
                        '(dr-evil mr-bigglesworth)))
         (trolls (map (lambda (name)
                        (create-troll name
                                      (pick-random rooms)
                                      (random-number 3)
                                      (random-number 3)))
                      '(grendel registrar))))

    (append students
;           profs        ;uncomment after writing wit-professor
            monitors trolls)))

(define me 'will-be-set-by-setup)
(define all-rooms 'will-be-set-by-setup)
(define chamber-of-stata 'will-be-set-by-setup)

(define (setup name)
  (ask clock 'RESET)
  (ask clock 'ADD-CALLBACK
       (create-clock-callback 'tick-printer clock 'PRINT-TICK))
  (let ((rooms (create-world)))
    (set! chamber-of-stata (instantiate-spells))

    (populate-spells rooms)

    (populate-players rooms)

    ;uncomment after writing chosen one
;    (create-chosen-one 'hairy-cdr (pick-random rooms)
;                       (random-number 3) (random-number 3))

    (set! me (create-avatar name (pick-random rooms)))
    (ask screen 'SET-ME me)
    (set! all-rooms rooms)
    'ready))

;; Some useful example expressions...

; (setup 'ben-bitdiddle)
; (run-clock 5)
; (ask screen 'DEITY-MODE #f)
; (ask screen 'DEITY-MODE #t)
; (ask me 'look-around)
; (ask me 'take (thing-named 'engineering-book))
; (ask me 'go 'up)
; (ask me 'go 'down)
; (ask me 'go 'north)
;
; (show me)
; (show screen)
; (show clock)
; (pp me)
```