MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Problem Set 7 Solutions

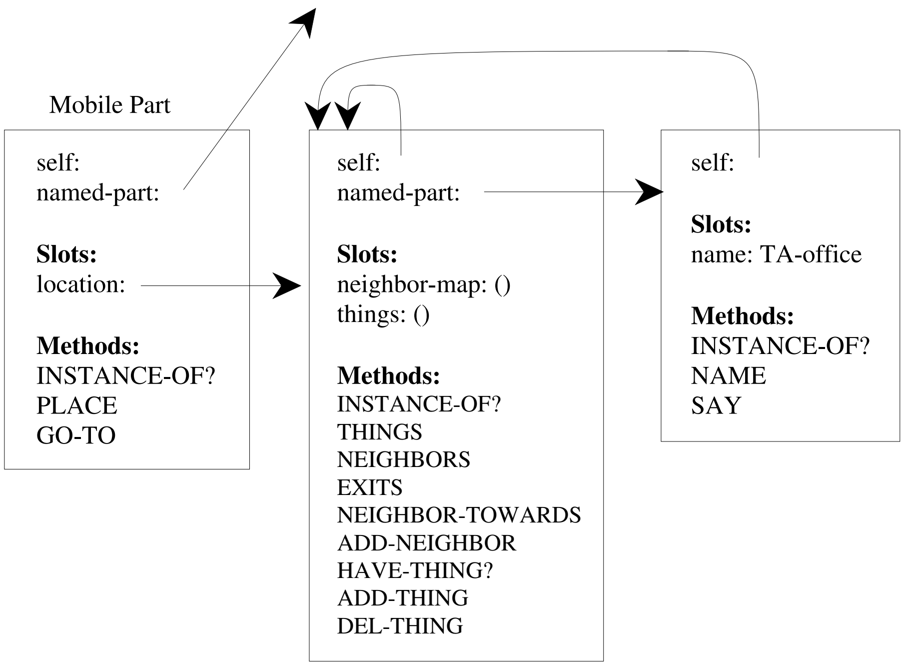
Lab Exercise 1: Here is what the print-state routine printed out when I loaded the file, ran it for 50 clock ticks, then ran print-state:

```
person grendel @ tech-square
person rod @ heaven person
albert @ building-36
place heaven : rod
place mars-landing-site
place rover-lab
place dormitory
place ta-office
place rod-office
place albert-office
place lounge
place tech-square : grendel
place computer-lab
place building-36 : albert
place dungeon
place whitehead
place egg-atrium
;Value: #[unspecified-return-value]
```

Lab Exercise 2: Here is the code I wrote to add Sojourner to the world:

```
(define sojourner (animate (new rover-class 'sojourner rover-lab 1 50)))
```

Lab Exercise 3: Here is the picture I drew:



Lab Exercise 4: The fact that Albert is in his office is stored in two places:

- The location slot in the mobile-part part of the albert object is set to the albert-office place object.
- The albert object is in the list of things in Albert's office, which is stored in the things slot of the albert-office object.

This information must satisfy 4 consistency properties:

1. The location slot in every mobile object must be a place.
2. Each person occurs exactly once in exactly one place's list of things.
3. If a person occurs in a place's list of things, the location slot in that person's mobile part must be that place object.
4. If the location slot in a person's mobile part is a given place, the person must occur in that place's list of things.

This information is kept consistent using the following protocol:

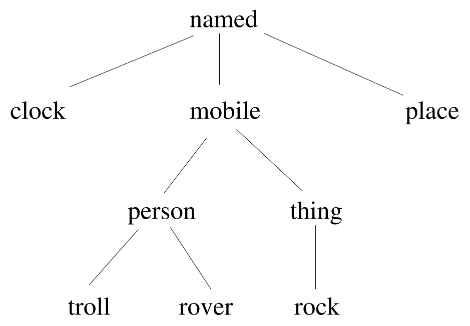
1. Person is sent a GO-TO message. The GO-TO method in the person object sends a GO-TO message to its mobile part object.
2. The GO-TO method in the mobile part object sends a DEL-THING message to the current place of the person.
3. The DEL-THING method in the current place of the person removes the person from the list of things at its location. At this point the system is inconsistent: the person is in no place's list of things.

4. The GO-TO method in the mobile part object sets its location to the new location. At this point the location slot in the person's mobile part is the new place, but the person is not in that place's list of objects.

5. The GO-TO method in the mobile part object sends an ADD THING message to the place where the person is moving.

6. The ADD-THING method in the place where the person is moving adds the person to its list of things. At this point the system is consistent again.

Lab Exercise 5: Here is the class hierarchy:



Lab Exercise 6

A) The original line was `(ask person-part 'ACT)`. The troll class inherits from person class, and trolls act like people except that they periodically eat other people they encounter. The original code for the ACT message in the troll class simply checks if there are any people in the same place as the troll, and if not, sends the ACT message to the trolls person part to make the troll act like a person.

The change sends the ACT message directly back to the troll object (not its person part), and if the troll is ever in a situation where there are no people around, the simulation will simply send the troll the ACT message forever.

B) The original line was `(new-part person-class self name birthplace threshold)`, which allocates a person object as part of the troll object. The important thing for this problem is that new-part sets the self slot in the allocated object to point to the troll object.

Changing the line to `(new person-class name birthplace threshold)` allocates a new person object and makes it part of the troll object, but the self slot in the person object points to the newly allocated person object, not the troll object. The upshot is that when the troll object is sent a CLOCK-TICK message, it is handled in the person part because the troll does not implement CLOCK-TICK. The CLOCK-TICK method in person sends ACT to self, which in this case is the person object and not the troll object. The observable behavior is that grendel never eats anyone, because he is acting just like a person, not a troll.

Lab Exercise 7: I added the following code:

```
(define barnacle-bill (new rock-class 'barnacle-bill mars-landing-site 9))
(define scooby-doo (new rock-class 'scooby-doo mars-landing-site 5))
(define yogi (new rock-class 'yogi mars-landing-site 3))
```

Lab Exercise 8: I added the following methods to the rover class. I decided to recompute the total weight of the rocks that the rover has every time it needed to know how much weight it was carrying. I also decided that rovers would only pick up rocks that nobody owned.

```
(CAN-TAKE
   (lambda (rock)
      (define (compute-weight)
        (define (helper l)
           (if (null? l) 0
               (+ (ask (car l) 'WEIGHT)
                  (helper (cdr l)))))
        (helper (filter (lambda (r) (ask r 'INSTANCE-OF? 'ROCK-CLASS))
                        (ask self 'POSSESSIONS))))
      (if (and (ask rock 'INSTANCE-OF? 'ROCK-CLASS)
               (not (ask rock 'OWNED?)))
          (let ((current-weight (compute-weight))
                (rock-weight (ask rock 'WEIGHT)))
             (if (<= (+ current-weight rock-weight) weight-limit)
                 true
                 false))
          false)))
(TAKE
   (lambda (rock)
      (if (ask self 'CAN-TAKE rock)
          (begin (ask person-part 'TAKE rock))
          (begin (ask self 'SAY)
                 (list "I can't take "
                 (ask rock 'NAME)))))))
(ACT
   (lambda ()
      (let ((rocks (filter
                     (lambda (rock) (ask self 'CAN-TAKE rock))
                     (ask (ask self 'PLACE) 'THINGS))))
        (if (not (null? rocks))
            (ask self 'TAKE (pick-random rocks)))
        (ask person-part 'ACT))))
```

Lab Exercise 9: I created a new class, rover maker, that is a subclass of person class. I overrode the ACT method so that when a rover maker acts, it creates a new rover instead of moving around. I made the rover maker take a bunch of parameters that determine the properties of the rovers that it creates.

```
(define-class
   '((class-name rover-maker-class)
     (parts (person-part))
     (slots ((rover-weight-limit 'not-initialized-yet)
             (rover-prefix 'not-initialized-yet)
             (rover-birthplace 'not-initialized-yet)
             (rover-threshold 'not-initialized-yet)
             (system-clock 'not-initialized-yet)
             (next-rover-number 0)))
     (constructor
        (lambda (name birthplace threshold initial-weight-limit
                 initial-rover-threshold initial-system-clock)
          (set! rover-prefix name)
          (set! rover-weight-limit initial-weight-limit)
          (set! rover-birthplace birthplace)
          (set! rover-threshold initial-rover-threshold)
          (set! system-clock initial-system-clock)
          (set! person-part
            (new-part person-class self name birthplace threshold)
          )
          'INITIAL-VALUES-SET))
    (methods
      ((INSTANCE-OF?
         (lambda (class-name) (if (eq? class-name 'ROVER-MAKER-CLASS)
                                  true
                                  (ask person-part
                                       'INSTANCE-OF? class-name))))
       (ACT
         (lambda ()
           (let ((rover-name
                   (attach-suffix rover-prefix next-rover-number)))
              (let ((new-rover (new rover-class rover-name
                                 rover-birthplace rover-threshold
                                 rover-weight-limit)))
                (set! next-rover-number (+ next-rover-number 1))
                (ask self 'SAY (list "Created rover " rover-name " in "
                                     (ask (ask self 'PLACE) 'NAME)))
                (ask system-clock 'add-to-list new-rover))))))))
```

Lab Exercise 10: I added the following method to the transporter class:

```
(ADD-THING
   (lambda (thing)
      (ask place-part 'ADD-THING thing)
      (if (ask thing 'INSTANCE-OF? 'ROVER-CLASS)
          (ask thing 'GO-TO destination))))
```

The basic idea is that when the teleporter gets the ADD-THING message to add a rover to the transporter room, it turns the message around and asks the rover to go to the destination.

There are several glitches. First, the messages that say where the rover is going come out in reverse order. So if a rover moves from the rover lab to transporter room 1, which transports the rover to the mars landing site, the messages would come out in the order that first says the rover moves from transporter room 1 to the mars landing site, then says that the rover moves from the rover lab to transporter room 1. A second glitch is that any rocks the rover is carrying are asked to follow the rover twice to the mars landing site, and are never in the transporter room at all. There appears to be no easy fix to these glitches that preserves the structure of the system.

Lab Exercise 11: Here is the pretty-printed definition of my rover-maker class:

```
(define
 (rover-maker-class)
 (let
  ((self 'uninitialized-self)
   (person-part 'uninitialized-part)
   (rover-weight-limit 'not-initialized-yet)
   (rover-prefix 'not-initialized-yet)
   (rover-birthplace 'not-initialized-yet)
   (rover-threshold 'not-initialized-yet)
   (system-clock 'not-initialized-yet)
   (next-rover-number 0))
  (let*
   ((instance-object
     (let
      ((instance-of?
        (lambda
         (class-name)
         (if   (eq? class-name 'rover-maker-class)
          true (ask person-part 'instance-of? class-name))))
       (act
        (lambda
         ()
         (let
          ((rover-name (attach-suffix rover-prefix next-rover-number)))
          (let
           ((new-rover
             (new               rover-class     rover-name
              rover-birthplace rover-threshold rover-weight-limit)))
           (set! next-rover-number (+ next-rover-number 1))
           (ask
            self
            'say
            (list   "Created rover "              rover-name
             " in " (ask (ask self 'place) 'name)))
           (ask system-clock 'add-to-list new-rover))))))
      (lambda
       (message)
       (cond                     ((eq? message 'instance-of?) instance-of?)
        ((eq? message 'act) act) (else (get-method message person-part))))))
    (constructor
     (lambda
      (name                 birthplace              threshold
       initial-weight-limit initial-rover-threshold initial-system-clock)
      (set! rover-prefix name)
      (set! rover-weight-limit initial-weight-limit)
```

```
      (set! rover-birthplace birthplace)
      (set! rover-threshold initial-rover-threshold)
      (set! system-clock initial-system-clock)
      (set! person-part (new-part person-class self name birthplace threshold))
      'initial-values-set)))
   (define
    (make-object class-message)
    (cond
     ((eq? class-message 'new-part)
      (lambda
       (new-self . args)
       (begin (set! self new-self) (apply constructor args) instance-object)))
     ((eq? class-message 'new)
      (lambda
       args
       (begin                      (set! self instance-object)
        (apply constructor args) instance-object)))
     (else (fail))))
   make-object)))
```

Optional Contest: I defined a thief class whose instances steal rocks from rovers. Before it steals the rock, it drops all the rocks that it currently has. Here is the code:

```
(define-class
  '((class-name thief-class)
    (parts (person-part))
    (slots ())
    (constructor
      (lambda (name birthplace threshold)
        (set! person-part
              (new-part person-class self name birthplace threshold)
        )
        'INITIAL-VALUES-SET))
    (methods
      ((INSTANCE-OF?
         (lambda (class-name) (if (eq? class-name 'THIEF-CLASS)
                                  true
                                  (ask person-part
                                       'INSTANCE-OF? class-name))))
      (ACT
       (lambda ()
         (let ((rovers (filter (lambda (p)
                                 (ask p 'INSTANCE-OF? 'ROVER-CLASS))
                               (ask self 'OTHER-PEOPLE))))
           (let ((rocks
                  (filter (lambda (p) (ask p 'INSTANCE-OF? 'ROCK-CLASS))
                          (apply append (map (lambda (p)
                                                (ask p 'POSSESSIONS))
                                              rovers)))))
             (if (not (null? rocks))
                 (begin
                   (map (lambda (t)
                          (if (ask t 'INSTANCE-OF? 'ROCK-CLASS)
                              (ask self 'LOSE t)))
                        (ask self 'POSSESSIONS))
                   (ask self 'TAKE (pick-random rocks))))))
          (ask person-part 'ACT)))
   ))))
```