

Problem Set 4

Lunar Lander: more Higher-order Procedures and a bit of Data Abstraction

Issued: Section 201 (Dimock): Friday, 1-Oct-2004, Section 202 (Yanco): Thursday, 30-Sep-2004

Due: Section 201 (Dimock): Friday, 8-Oct-2004, Section 202 (Yanco): Thursday, 7-Oct-2004

Reading: Covers text (SICP 2nd Edition by Abelson & Sussman): Through 2.2.1

Overview

This problem set asks you to do some more with higher-order procedures.
Use the MzScheme language in DrScheme.

What to turn in

Keep your answers in a file named `ps4-ans.ss` (the file that saves your definitions buffer). Put the code for each problem sequentially in the definitions buffer. (If you work problems in a different sequence, put them in order before you submit.) Put the problem number in a comment before the code for that problem (use a semi-colon to make a line a comment).

Underneath the code for each problem, cut and paste the appropriate sample runs. You can put semi-colons in front of the sample runs, which will comment them out and allow you to load the entire buffer in another session.

Print the definitions buffer (or the file `ps4-ans.ss`, which should be equivalent) and turn it in during the class on the due date. Also submit the answers using the submit command:

Section 201: `submit dimock ps4 ps4-ans.ss`

Section 202: `submit holly 301-ps4 ps4-ans.ss`

Problems

Louis Reasoner, appalled by this semester's tuition bill, has decided to put his education to work and reap a fortune in the home video games business. His first best-selling game, he decides, is to be called "Lunar Lander." The object is to land a spaceship on the moon, firing the ship's rockets in order to reach the surface at a small enough velocity, and without running out of fuel.

In implementing the game, Louis assumes that the user will, at each instant, specify a *rate* at which fuel is to be burned. The rate is a number between 0 (no burn) and 1 (maximum burn). In his model of the rocket engine, burning fuel at a given rate will provide a force of magnitude `strength * rate` where `strength` is some constant that determines the strength of the rocket engine.

Louis creates a data structure called `ship-state` which consists of the parts of the ship's state relevant to his program: the `height` above the moon, the `velocity`, and the amount of `fuel` that ship has. He defines this data structure by a constructor `make-ship-state` and three selectors: `height`, `velocity`, and `fuel`.

You need to remember a few formulas from physics to understand this problem set. We are going to assume for this version of the game that there is only one dimension "height".

- If h is the height of the lunar lander from the surface of the moon, then the lander's velocity is given by $v = \frac{dh}{dt}$
In this problem set positive velocities are away from the moon, negative velocities are towards the moon. So a safe landing occurs if the lander reaches height 0 with a slight negative velocity.
- The lander's acceleration $a = \frac{dv}{dt}$
- Force = mass \times acceleration. We will assume that the lander has mass = 1, so that we don't have to do an extra multiplication everywhere. Recall that the units for mass are not the same as the units for acceleration, but that will not play a role in these problems. (We make the unwarranted assumption that burning the fuel does not make any significant difference in the mass of the lander.)
- The total force on the lander is then `(strength * rate) - (gravity * mass)` or just `(strength * rate) - gravity` since the `mass` is 1.

The heart of Louis' program is a procedure that updates the ship's position and velocity.

Louis embodies the equations in a procedure that takes as arguments a `ship-state` and the rate at which fuel should be burned and returns the new state of the ship after a time interval `dt`:

```
(define (update ship-state fuel-burn-rate)
  (make-ship-state
    (+ (height ship-state) (* (velocity ship-state) dt))
    (+ (velocity ship-state)
      (* (- (* engine-strength fuel-burn-rate) gravity)
         dt))
    (- (fuel ship-state) (* fuel-burn-rate dt))))
```

The first parameter to `make-ship-state` calculates a new height as an old height + dh ($= v \times dt$). The second parameter calculates the velocity as the old velocity + dv ($= a \times dt = f/m \times dt = f/1 \times dt$) using the equation for force given above. The third parameter calculates the amount of fuel left, based on the fact that the amount of fuel remaining will be the amount of fuel in the previous state diminished by the rate times `dt`.)

Here is the main loop of Louis' program:

```
(define (lander-loop ship-state)
  (show-ship-state ship-state)
  (if (landed? ship-state)
      (end-game ship-state)
      (lander-loop (update ship-state (get-burn-rate)))))
```

The procedure first displays the ship's state, then checks to see if the ship has landed. If so, it ends the game. Otherwise, it continues with the updated state. The procedure `show-ship-state` simply prints the

state at the terminal:¹

```
(define (show-ship-state ship-state)
  (display (list
    "height" (height ship-state)
    "velocity" (velocity ship-state)
    "fuel" (fuel ship-state)))
  (newline))
```

Louis considers the ship to have landed if the height is less than or equal to 0:

```
(define (landed? ship-state)
  (<= (height ship-state) 0))
```

To end the game, Louis checks that the velocity is at least as large as some (downward) safe velocity. This determines whether or not the ship has crashed:

```
(define (end-game ship-state)
  (let ((final-velocity (velocity ship-state)))
    (display "final velocity ")
    (display final-velocity)
    (newline)
    (cond ((>= final-velocity safe-velocity)
      (display "good landing ")
        "game over")
      (else
        (display "you crashed! ")
        "game over")))))
```

The burn rate is determined by having the user to type a character at the keyboard. In this initial implementation, the only choices are maximum burn or zero burn. Typing a particular key will signal maximum burn, hitting any other key (except “Enter”) will signal zero burn:²

```
(define (get-burn-rate)
  (let ((c (read-char (current-input-port))))
    (cond ((char=? c #\newline) (get-burn-rate))
      ((char=? c burn-key) 1)
      (else 0))))
```

The final procedure simply sets the game in motion by calling `lander-loop` with some initial values:

```
(define (play) (lander-loop (initial-ship-state)))
(define (initial-ship-state)
  (make-ship-state 50 ;height
    0 ;velocity
    20) ;fuel
```

Now all that remains is to define some constants:

¹Louis is taking a graphics course this semester. At some point he will replace this step by something that draws a picture of the ship’s instrument panel, together with a view of the up-rushing moon.

²In later implementations, Louis will not make the procedure wait for the user, so that his game will be “real time.” In the meantime, there are a few things you should know:

The procedure uses the Scheme primitives `read-char`, and `current-input-port` to wait for the user to hit a key and then return the character for the key that was hit.

In DrScheme, `read-char` is not asynchronous: no character is read until you type a line terminator or click on the EOF button. You will have to press `Enter` to get `get-burn-rate` to return. If click on the EOF button you will probably need to click the `Execute` button before DrScheme will accept typed input again.

(If you want to extend the game to add other keys, read up on characters in the R5RS. If you want to use a key that produces a non-printing character, then you will need to use the `integer->char` and `char->integer` primitives.)

```
(define dt 1)

(define gravity .5)

(define safe-velocity -2)

(define engine-strength 1)

(define burn-key #\space)
```

(The final definition sets up the space bar as the key to hit to signal a burn. It does this by prescribing the character for space as the value against which to check the keyboard input. Hitting a space will signal a burn, and hitting any other key will signal not to burn. If you want to know more about characters, see the R5RS.)

Problem 1: Louis' program is not quite complete, because he has forgotten to write the constructor `make-ship-state` and the selectors `height`, `velocity`, and `fuel` that define the `ship-state` data structure. Define these procedures appropriately. Now you should be able to run the game by typing `play`. Try landing the ship a few times. For this exercise, you should turn in a listing of the four procedures that you defined.

Louis rushes to show his game to his friend Alyssa P. Hacker, who is not the least bit impressed.

“Oh, Louis! That game is older than Moses. I'm sure that half the kids in the country have programmed it for themselves on their home computers.³ Besides,” she adds as he slinks away, “your program has a bug.”

Problem 2: Alyssa has noticed that Louis has forgotten to take account of the fact that the ship might run out of fuel. If there is x amount of fuel left, then, no matter what rate is specified, the maximum rate at which fuel can be burned during the next time interval is x/dt . Show how to install this constraint as a simple modification to the `update` procedure. (`Update` is the only procedure that should be changed.) As a check, run the program and respond by typing space each time the program asks how much fuel to burn. Describe the behavior of the ship in response to this strategy. Put the definition of your modified `update` procedure in your answer file here. You do not need to update the old procedures in place, you can leave the original versions unchanged and redefine them here.

Louis is dejected, but not defeated, for he has a new idea. In his new game, the object will be to come up with a *general strategy* for landing the ship. Since Louis' game will be played using Scheme, a strategy can be represented as a *procedure* that takes a ship state and returns a burn rate between 0 and 1. Two very simple strategies are

```
(define (full-burn ship-state) 1)
(define (no-burn ship-state) 0)
```

The new game reduces to the original one if we use a strategy that says in effect to “ask the user”:

```
(define (ask-user ship-state) (get-burn-rate))
```

where `get-burn-rate` is the procedure used in the original game above.

Problem 3: Modify Louis' `play` and `lander-loop` procedures so that `play` now takes an input – the strategy procedure to use to get the new burn rate. To test your modification, define the three simple procedures above, and check that

³One professor at UMass Lowell recalls playing Lunar Lander circa 1973 on a Tektronix terminal hooked to a PDP-10 computer. A feature of that version of the game was that if the player landed the ship within a certain distance of the lunar McDonalds franchise, a figure of an astronaut would be seen to enter the franchise and order two burgers to go (the text being put on the screen since the game did not require speakers). The player recalls with some embarrassment crashing the lunar lander into the only McDonalds on the moon.

```
(play ask-user)
```

has the same behavior as before, and that

```
(play full-burn)
```

makes the ship crash. Put your modified procedures in your answer file. You do not need to update the old procedures in place, you can leave the original versions unchanged and redefine them here.

Alyssa likes this new idea much better, and comes up with a twist of her own by suggesting that one can create new strategies by combining old ones. For example, we could have make a new strategy by, at each instant, choosing between two given strategies. If the two strategies were, say, `full-burn` and `no-burn`, we could express this new strategy as

```
(lambda (ship-state)
  (if (= (random 2) 0)
      (full-burn ship-state)
      (no-burn ship-state)))
```

The MzScheme or MIT Scheme built-in procedure `random` is used to return either 0 or 1 with equal odds. Testing whether the result is zero determines whether to apply `full-burn` or `no-burn`.

Note the important point that since the combined strategy is a *strategy*, it must itself be a procedure that takes a ship-state as argument, hence the use of `lambda`.

Problem 4: Generalize this idea further by defining a procedure for combining strategies called `random-choice`. This takes as arguments two *strategies* and returns the new *strategy* whose behavior is, at each instant, to apply at random one or the other of the two component strategies. In other words, running

```
(random-choice full-burn no-burn)
```

should generate the strategy shown above.

Note that we can think of a *language of strategies*. The language of strategies has primitive strategies like `ask-user`, `full-burn`, or `no-burn`. We write strategies as Scheme procedures so, in the language of strategies, the means of combination is procedure application (just as in Scheme) and the means of abstraction of strategies is procedure definition (just as in Scheme).

Test your procedure by running

```
(play (random-choice full-burn no-burn))
```

Turn in a listing of your procedure.

Problem 5: Define a procedure to combine strategies called `height-choice` that chooses between two strategies depending on the height of the ship. `Height-choice` itself should be implemented as a procedure that takes as arguments two strategies and a height at which to change from one strategy to the other. For example, running

```
(play (height-choice no-burn full-burn 30))
```

should result in a strategy that does not burn the rockets when the ship's height is at least 30 and does a full-burn when the height is below 30. Try this. You should find that, with the initial values provided in Louis' program, this strategy actually lands the ship safely. Turn in a listing of `height-choice`.

Problem 6: `Random-choice` and `height-choice` are special cases of a more general strategy combiner called `choice`, which takes as arguments two strategies together with a *predicate* used to select between them. The predicate should take a ship-state as argument. For example, `random-choice` could alternatively be defined as

```
(define (random-choice strategy-1 strategy-2)
  (choice strategy-1
          strategy-2
          (lambda (ship-state)
            (= (random 2) 0))))
```

Define `choice` and show how to define `height-choice` in terms of `choice`. Turn in listings of `choice` and the new definition of `height-choice` named `height-choice2`.

Problem 7: Using your procedures, give an expression that represents the strategy: “If the height is above 40 then do nothing. Otherwise randomly choose between doing a full burn and asking the user.”

Call your new strategy `height-choice3`. Turn in a listing of its code.

Louis and Alyssa explain their idea to Eva Lu Ator, who says that the game would be more interesting if they provided some way to specify burn rates other than 0 or 1.

“In fact,” says Eva, who, despite the fact that she is a sophomore, still has some dim recollection of freshman physics, “you can compute a burn rate that will bring the ship to a safe landing.” Eva asserts that, if a body at height h is moving downward with velocity $-v$, then applying a constant acceleration $a = \frac{v^2}{2h}$ will bring the body to rest at the surface.

Eva’s observation translates into a strategy: At each instant, burn enough fuel so that the acceleration on the ship will be as given by the formula above. In other words, *force* the ship to fall at the right constant acceleration. (Observe that this reasoning implies that even though v and h change as the ship falls, a , as computed by the formula, will remain approximately constant.)

Problem 8: Implement Eva’s idea as a strategy, called `constant-acc`. (You must compute what burn rate to use in order to apply the correct acceleration. Don’t forget about gravity!) Try your procedure and show that it lands the ship safely. Turn in a listing.

One minor problem with Eva’s strategy is that it only works if the ship is moving, while the game starts with the ship at zero velocity. This is easily fixed by letting the ship fall for a bit before using the strategy. Louis, Eva, and Alyssa experiment and find that

```
(play (height-choice no-burn constant-acc 40))
```

gives good results.

Continuing to experiment, they observe a curious phenomenon: the longer they allow the ship to fall before turning on the rockets, the less fuel is consumed during the landing.

Problem 9: By running your program, show that
`(height-choice no-burn constant-acc 30)`
lands the ship using less fuel than
`(height-choice no-burn constant-acc 40)`

Turn in (as comments) the amount of fuel left on landing for each choice of height.

This suggests that one can land the ship using the least amount of fuel by waiting until the very end, when the ship has almost hit the surface, before turning on the rockets. But this is unrealistic because it ignores the fact that the ship cannot burn fuel at an arbitrarily high rate.

Problem 10: This uncovers another bug in the `update` procedure. Fix the bug in a new procedure called `update2`. In `update2`, no matter what rate is specified, the actual burn rate will never be greater than 1. Also define `play2`, which calls `lander-loop2`, which in turn calls `update2`.

Turn in `update2`, `play2`, and `lander-loop2`. Try

```
(play2 (height-choice no-burn constant-acc X))
```

for a couple of values of `X`.

A realistic modification to the “wait until the very end” strategy is to let the ship fall as long as the desired burn rate, as computed by Eva’s formula, is sufficiently less than 1, and then follow the `constant-acc` strategy.

Problem 11: Implement this strategy as a procedure, called `optimal-constant-acc`. (You may have to experiment to determine an operational definition of “sufficiently less than 1.”) Try your procedure. How much fuel does it use? Turn in a listing.

Provided code for Lunar Lander

You should find initial code that will get you started (once you write `make-ship-state`, `height`, `velocity`, and `fuel`) in file `ps4.ss`