Brandeis University

COMPUTER SCIENCE (1)21B (SPRING TERM, 2019)

Structure and Interpretation of Computer Programs

Problem Set 1: Lunar Lander

Due Monday, February 4

Reading Assignment: Chapter 1.

1 Homework Exercises

Exercise 1.17: Fast product based on fast exponentiation

Exercise 1.34: Weird self-application

Exercise 1.43: Repeated composition of a function

Exercise 1.44: Smoothing a function

Function doubling: (a little difficult) Suppose we specify the *doubling function* as:

```
(define (double fn) (lambda (x) (fn (fn x))))
```

Evaluate the following, and explain what is going on using the substitution model:

```
((double 1+) 0)
(((double double) 1+) 0)
((((double double) double) 1+) 0)
(((((double double) double) double) 1+) 0)
```

Try to estimate the value of `((((((double double) double) double) double) 1+) 0)`. Why can this value not be computed using the Scheme interpreter?

2 Laboratory Assignment: Lunar Lander

Louis Reasoner, appalled by this term's tuition bill, has decided to put his education to work and reap a fortune in the video game business. His first best-selling game, he decides, is to be called "Lunar Lander." The object is to land a spaceship on a planet, 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 motor, 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 planet, the `velocity`, and the amount of `fuel` that the ship has. He defines this data structure by a constructor, `make-ship-state` and three selectors: `height`, `velocity`, and `fuel`.

The heart of Louis' program is a procedure that updates the ship's position and velocity. Louis knows that if `h` is the height of the ship and `v` is the velocity, then

$$\frac{dh}{dt} = v \quad \text{and} \quad \frac{dv}{dt} = \text{total force} = strength * rate - gravity$$

where `gravity` measures the gravitational attraction of the planet. (The above equations assume that the ship has unit mass.) Louis embodies these equations in a procedure that takes as arguments a `ship-state`, the rate at which fuel is to be burned over the next time interval, and returns the new state of the ship after 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))))
```

(Besides the two equations above, the procedure also reflects the fact that the amount of fuel remaining will be the original amount of fuel diminished by the `fuel-burn-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 of the ship has landed. If so, it ends the game. Otherwise, it continues with the update state. The procedure `show-ship-state` simply prints the state at the terminal.[1]

```
(define (show-ship-state ship-state)
  (write-line
    (list 'height (height ship-state)
          'velocity (velocity ship-state)
          'fuel (fuel ship-state))))
```

It's worth adding that this time-honored—or time-worn?—primitive video game has quite a history. It was originally conceived and implemented by Jim Storer in 1969, who was a student at Lexington High School, and now a computer science faculty member at Brandeis. We're using it to give some nice examples of higher-order procedures. For the history, see

`http://technlogizer.com/2009/07/19/lunar-lander/`
`http://en.wikipedia.org/wiki/Lunar_Lander_(video_game)`

[1] Louis is planning to buy a graphics terminal, at which point he will replace this step by something that draws a picture of the ship's instrument panel, together with a view of the uprushing planet. If you are a venture capitalist with money, rather than fuel, to burn, it's possible that Louis would cooperate in a joint business venture.

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)))
     (write-line final-velocity)
     (cond ((>= final-velocity safe-velocity)
            (write-line "good landing")
            'game-over)
           (else
            (write-line "you crashed!")
            'game-over))))
```

The burn rate is determined by asking the user to type a character at the keyboard. In this initial implementation, the only choices are maximum burn or zero burn. We "normalize" this burn rate to be 0 or 1: in full generality, the burn rate can be any real number between 0 and 1, and a little later on we'll look at some clever ways of computing such burn rates. For the moment, typing a particular key will signal maximum burn, and hitting any other key will signal zero burn:[2]

```
(define (get-burn-rate)
  (if (= (player-input) burn-key)
      1
      0))

(define (player-input)
  (char->integer (prompt-for-command-char " action: ")))
```

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
                   0
                  20))
```

Now all that remains is to define some constants:

```
(define dt 1)

(define gravity 0.5)

(define safe-velocity -0.5)

(define engine-strength 1)

(define burn-key 32)
```

[2]The procedure uses a procedure we have defined called `(player-input)`, which waits for the user to hit a key and then returns the ASCII code of the key that was hit. The code for `(player-input)` is given in the code listing at the end of this handout. (If you want to extend the game to add other keys, you can find the right code by simply evaluating `(player-input)` in Scheme followed by RETURN, hitting a key, and seeing what number was returned). In later implementations, Louis will not make the procedure wait for the user, so that his game will be "real time."

(The final definition sets up the space bar as the key to hit to signal a burn. It does this by prescribing the ASCII code 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.)

The code needed for this problem set can be found at `http://www.cs.brandeis.edu/~cs21b`.

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

"Oh, Louis, you make me ashamed to be a nerd! 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. Besides," she adds as he slinks away like a whipped dog with his tail between his legs, "your program has a bug."

Problem 1. 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 (average) rate at which fuel can be burned during the next time interval is `(/ x dt)`. (Why? Imagine what happens if you burn f gallons of fuel per second for Δt seconds when you have less than $f\Delta t$ gallons... you run out of gas a little earlier than expected!...)

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 the space key each time the program asks how much fuel to burn; include, in what you submit, your modified `update` procedure.

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*. We could specify a strategy by defining a procedure that takes a ship state as input, 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 2. Modify Louis' `play` and `lander-loop` procedures so that `play` now takes an input—the strategy procedure to use to get the new state. To test your modification, define the three simple procedures above, and check that

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

has the same behavior as before, and that

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

makes the rocket behave as you saw in Problem 1. Turn in your modified procedures.

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 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 Scheme 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 *also* a strategy, it must itself be a procedure that takes a `ship-state` as an argument, hence the use of `lambda`.

Problem 3. Generalize this idea further by defining a "higher level" strategy called `random-choice`. This procedure takes as arguments two *strategies* and returns the compound *strategy* whose behaviour 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 compound strategy shown above. Test your procedure by running

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

Problem 4. Define a new compound strategy called `height-choice` that chooses between two strategies depending on the height of the rocket. `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 above 30 and does a full-burn when the height is below 30. Try this. You should find that, with the intial values provided in Louis' program, this strategy actually lands the ship safely. Turn in your code for `height-choice`.

Problem 5. `Random-choice` and `height-choice` are special cases of a more general compound strategy 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 your code for `choice` and the new definition of `height-choice`.

Problem 6. Using your procedures, give an expression that represents the compound strategy: "If the height is above 40 then do nothing. Otherwise randomly choose between doing a fullburn and asking the user."

Louis and Alyssa explain their idea to Eva Lu Ator, who says that the game would be more interesting if the 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.[3]

Problem 7. Show that Eva is correct.[4] More specifically, show that if the ship is at initial height h_0 and initial velocity $-v_0$ (i.e., *towards* the ground), and is subjected to an *upwards* acceleration of $v_0^2/2h_0$, then the ship lands with final velocity $v_f = 0$.

[3] Set `dt` to a smaller value than 1—otherwise this strategy may crash since it's a bad numerical approximation.

[4] Don't gripe about a physics problem in this course. Computer scientists should know some physics.

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 rocket 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. Let's see your code...

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 20)
```

lands the ship using less fuel than

```
(height-choice no-burn constant-acc 30)
```

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 strategy 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. Change the procedure so that, no matter what rate is specified, the actual burn rate will never be greater than 1.

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?

Problem 12. *Optional:* Suppose the ship begins at height `h` with 0 velocity, falls under gravity through a height Δh and then uses the `constant-acc` strategy. Define a formula that tells how much fuel will be used during the landing. The formula should be in terms of `h`, Δh, gravity, and engine strangth. Check that your formula predicts that the amount of fuel used decreases as Δh increases. In doing this exercise, assume that time runs continously. That is, ignore the effect in the simulation caused by `dt`. Thus your formula will not give exactly the same results as your simulation unless you make `dt` very small. Also ignore any limits on burn rate such as the one you implemented in Problem 10.

Problem 13. *Optional:* What amount of fuel does your formula of Problem 12 predict will be used when you "wait until the last minute," i.e., when $\Delta h = h$? Can you think of a way to show that this is the *minimum* amount of fuel that *any* method of landing the ship (i.e., making it reach the ground with zero velocity) could use? (Hint: prove that the fuel consumed by a successful landing strategy grows as a constant multiple of the amount of time spent carrying out the landing strategy.)

```
;; this is the code for problem set -- Lunar Lander

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

(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)))))

(define (show-ship-state ship-state)
  (write-line
    (list 'height (height ship-state)
          'velocity (velocity ship-state)
          'fuel (fuel ship-state))))

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

(define (end-game ship-state)
  (let ((final-velocity (velocity ship-state)))
       (write-line final-velocity)
       (cond ((>= final-velocity safe-velocity)
              (write-line "good landing")
              'game-over)
             (else
              (write-line "you crashed!")
              'game-over))))

(define (get-burn-rate)
  (if (= (player-input) burn-key)
      1
      0))

(define (play) (lander-loop (initial-ship-state)))

(define (initial-ship-state)
  (make-ship-state 50       ; 50 km high
                   0        ; not moving (0 km/sec)
                   20))     ; 20 kg of fuel left

(define dt 1)               ; 1 second interval of simulation
(define gravity 0.5)        ; 0.5 km/sec/sec
(define safe-velocity -0.5) ; 0.5 km/sec or faster is a crash
(define engine-strength 1)  ; 1 kilonewton-second
(define burn-key 32)        ;space key

(define (player-input)
  (char->integer (prompt-for-command-char " action: ")))
```

```
; You'll learn about the stuff below here in Chapter 2.  For now, think of make-ship-state,
; height, velocity, and fuel as primitive procedures built in to Scheme.

(define (make-ship-state height velocity fuel)
  (list 'HEIGHT   height
        'VELOCITY velocity
        'FUEL     fuel))

(define (height state) (second state))
(define (velocity state) (fourth state))
(define (fuel state) (sixth state))

(define (second l) (cadr l))
(define (fourth l) (cadr (cddr l)))
(define (sixth l) (cadr (cddr (cddr l))))


; Users of DrScheme or DrRacket: add these for compatibility with MIT Scheme...
; for input and output


(define (write-line x)
  (display x)
  (newline))

(define (prompt-for-command-char prompt)
  (display prompt)
  (read-char))

; for random number generation

(#%require (only racket/base random))

; a ridiculous addendum  (you'll need this for the exercises)

(define (1+ x) (+ 1 x))
```