Brandeis University

Computer Science (1)21b (Spring Term, 2020)

Structure and Interpretation of Computer Programs

Problem Set 1: Lunar Lander

Due Monday, January 27

Reading Assignment: Chapter 1.

1 Homework Exercises

Not to be handed in, but sure to be helpful.

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

The goal of the game is to safely land a spaceship on a planet by choosing how much fuel to burn. Actually, the goal is to *write code* that controls how much fuel to burn. You are provided with `lunar-defns.rkt`, a file containing the building blocks of the game; `lunar.rkt`, a starting point for your code, and `lunar-test.rkt`, a which contains unit tests for you to run. You hand in only `lunar.rkt`, which we will test using **unmodified** versions of `lunar-defns.rkt` and `lunar-test.rkt`. So if you modify those files in the course of debugging, or out of curiosity, make sure your code runs and passes tests with the original versions.

`lunar-defns.rkt` defines a data structure called `ship` which consists of the parts of the ship's state relevant to our program: the `height` above the planet, the `velocity`, and the amount of `fuel` that the ship has. The procedure to construct a `ship` is simply called `ship`, and the procedures to access its fields are `ship-height`, `ship-velocity`, `ship-fuel`.

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

`lunar-defns.rkt` contains a procedure to simulate these equations over one `dt`-long interval of time:

```
(define (update state fuel-burn-rate)
  (ship ; create a new ship state
   ; the change in height is proportional to the velocity
   (+ (ship-height state) (* (ship-velocity state) dt))
   ; the change in velocity is proportional to the burn rate
   (+ (ship-velocity state)
      (* (- (* engine-strength fuel-burn-rate) gravity)
         dt))
   ; and the so is the change in fuel reserves
   (- (ship-fuel 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`.)

The main loop is the procedure `lander-loop`. It takes both a state and a `burn-strategy`. A burn strategy is itself a procedure– one that takes a state and returns a burn rate.

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

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 calls the `burn-strategy` to determine how much fuel to burn, updates the state accordingly, and loops again. The procedure `show-ship` simply prints the state at the terminal.

Have a look in `lunar.rkt`, which contains three very simple burn strategies. `full-burn` always burns the maximum amount of fuel. `no-burn` never burns any fuel. `ask-user` asks the user whether to burn or not for that time step, and acts accordingly. Note that even though these are all passed a `ship` as an argument, they don't bother to inspect it (yours will).

We consider the ship to have landed if the height is less than or equal to 0:

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

and to have landed *safely* if it didn't hit the ground too fast:

```
(define (landed-safely? state)
  (and (landed? state) (>= (ship-velocity state) safe-velocity))
```

Note that a positive velocity denotes upward movement, which is why we are using `>=`.

The `play` procedure in `lunar.rkt` simply sets the game in motion by calling `lander-loop` with some initial values:

```
(define (play strat) (lander-loop (initial-ship) strat))
```

Now all that remains is to define some constants, used in `update`:

```
(define dt 1)

(define gravity 0.5)

(define safe-velocity -0.5)

(define engine-strength 1)
```

Problem 1. Our `update` procedure doesn't 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!...)

Additionally, no matter how much fuel we have, we can only burn it so fast; the maximum burn rate is 1. And negative burn rates make no sense either; our rocket engine isn't a vacuum cleaner, and even if it were, vacuum cleaners don't work well in space.

Our `update` procedure is a beautiful elegant physics simulation, and shouldn't have to worry about such crass material concerns. We should just make sure that our burn strategies only give sensible outputs.

Implement a procedure `clamp` which takes as its argument a burn strategy and returns a new burn strategy, whose outputs are the same as the old, but constrained to make physical sense. That is, if the old burn strategy wants to burn negative fuel, the new strategy should burn nothing. And if the old strategy wants to burn more fuel than is possible, we should simply do our best and burn all we can.

Since burn strategies are procedures, `clamp` must accept a procedure as an argument, and return another procedure.

Problem 2. Suppose you have two strategies, and you can't decide which one you like best.

Implement a proceure `average-strat` which takes *two* burn strategies, and returns a new strategy which always returns the average of the two strategies.

Problem 3. Blindly taking the average of two strategies is a bit indecisive. Maybe some strategies perform better when we're high up, and some perform better close to the ground?

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. You can check for yourself that `(play (clamp (height-choice no-burn full-burn 30)))` results in a safe landing.

Problem 4. Both `ask-user` and `height-choice` choose between two strategies based on some boolean condition.

Define a generalization of these two procedures, called `choice`, which takes two strategies and a predicate. The predicate should take a `ship` as an input and return a boolean. The return value should be a new strategy which executes the first strategy if the predicate is true, otherwise the second.

For example, the following should be equivalent to `ask-user`:

```
(choice full-burn no-burn (lambda (state) (ask-to-burn)))
```

and the following should be equivalent to `height-choice`:

```
(lambda (strat1 strat2 height)
  (choice strat1 strat2
          (lambda (state)
            (>= (ship-height state) height))))
```

Problem 5. Using your previously-defined procedures, implement a strategy `ask-after-40` that represents the compound strategy: "Don't bother the user as long as the height is above 40. After that, ask them."

Problem 6.

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.

Implement this 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 to check that it lands the ship safely.

One minor problem with this 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. Check for yourself that

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

gives good results.

If you experiment with the cutoff height in the above expression, you might observe a curious phenomenon: the longer you allow the ship to fall before turning on the rockets, the less fuel is consumed during the landing.

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.

So the best you can do is allow the ship to fall, picking up speed, until just before the point where your engine would not be strong enough to land safely.

Problem 7. Implement this strategy as a procedure, called `optimal-constant-acc`. It must go from zero to constant output, only produce physically plausible burn rates, and land the ship safely using as little fuel as possible. **Hint:** how hard must you fire the rockets after delaying as long as possible? You don't need to actually use the **constant-acc** strategy.

Problem 8. To test whether this `optimal-constant-acc` strategy is really so good, implement a procedure `best-strategy`, that takes two strategies as arguments and returns the best one, by evaluating their performance on the `initial-state`. By "best" we mean:

- Landing safely is better than crashing.
- If both land safely, using less fuel is better than using more fuel.

You don't need to worry about both strategies crashing, or both using the same amount of fuel.