**Comp200 - Structure and Interpretation of Computer Programs**

# Project 1 - Who you calling a Big Ape?

- **Due To**: 25.10.2015
- Files: Please retrieve your work file project1.scm from **DriveF@VOL/COURSES/UGRADS/COMP200/SHARE/USERNAME/project1/project1.scm** and put all your answers in there.
- Submission: Please name your homework file to project1.scm and upload it to **DriveF@VOL/COURSES/UGRADS/COMP200/HOMEWORK/USERNAME/project1/project1.scm** Projects that are uploaded incorrectly will not be graded.
    - Ex: **DriveF@VOL/COURSES/UGRADS/COMP200/HOMEWORK/myatbaz/project1/project1.scm**
- Reading: Through Section 1.2 in *Structure and Interpretation of Computer Programs*

## Purpose

The purpose of Project 1 is for you to gain experience with writing and testing relatively simple procedures. For each problem below, include your code (with identification of the problem number being solved), as well as comments and explanations of your code, **and** demonstrate your code's functionality against a set of test cases. On occasion, we may provide some example test cases, but you should always create and include your own additional, meaningful test cases to ensure that your code works not only on typical inputs, but also on "boundary" or difficult cases. Get in the habit of writing and running these test cases after **every** procedure you write - no matter how trivial the procedure may seem to you.

## Scenario

Back when the PC was young and frivolous, there was a game written in QBasic called Gorillas. Picture, if you will, a "vast" cityscape, with many tall buildings. Atop a building on the left and a building on the right stand two massive gorillas. Each gorilla has a bulging sack of bananas. In this arena, they duel by throwing these explosive bananas at the base of each other's building until only one gorilla remains standing. You will help bring this classic game back to life.

Figure 1. A screen shot from the original QBasic Gorillas game.

## Problem 1: Basic Physics

To get started in building our version of the game, we need a physical model for a flying explosive banana. For the moment, we'll model it as a particle that moves along a single dimension with some initial position *u*, some initial velocity *v*, and some initial acceleration *a*, as pictured in Figure 2 below. The equation for the position of the banana at time *t*, given *a*, *v*, and *u* is $u_t = \frac{1}{2} a t^2 + v t + u$. Note that this denotes a first order differential equation in time. Later, we can apply this equation to either the horizontal (*x*) component of banana motion, or the vertical (*y*) component of banana motion.

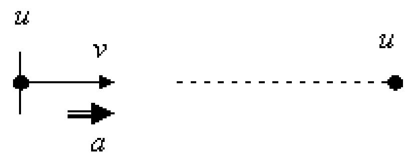

Figure 2: Motion of a banana in a generic direction.

Write a procedure that takes as input values for `a`, `v`, `u`, and `t` and returns as output the position of the banana at time `t`.

```
(define position
  (lambda (a v u t)
    your-answer-here))
```

Test your position code for at least the following cases:

```
(position 0 0 0 0)       ; -> 0
(position 0 0 20 0)      ; -> 20
(position 0 5 10 10)     ; -> 60
(position 2 2 2 2)       ; ->
(position 5 5 5 5)       ; ->
```

The template code file `project1.scm` will have these tests, and other test cases for other procedures, which you should run (you can add/show your output values). In addition, you should add some test cases of your own to these to cover other boundary and typical conditions.

## Problem 2: Basic Math

In order to compute when the banana hits the ground, we'll want to find when the *y* coordinate of the banana's position reaches zero. This can be discovered by finding the roots of the *y* position equation, and selecting the one that is larger (later in time). The proper tool for this is the quadratic formula. Given the coefficients of the quadratic equation $az^2 + bz + c = 0$, write a procedure to find one of the roots (call this `root1`), and another procedure to find the other root (call this `root2`).

```
(define root1
  (lambda (a b c)
    your-answer-here))

(define root2
  (lambda (a b c)
    your-answer-here))
```

You may notice that, depending on how you wrote your procedures, for some test cases you get an error. For example, try (root1 5 3 6). What happens? If you get an error, which is likely if you wrote your code the straightforward way, figure out how to change it so that your procedure returns a false value in those cases where there is not a valid solution.

## Problem 3: Flight Time

Given an initial upward velocity (in meters per second, or m/s) and initial elevation or height (in meters, or m), write a procedure that computes how long the banana will be in flight. Remember that gravity is a downward acceleration of $9.8\text{m/s}^2$. Note that to solve this you will need a root of a quadratic equation. Try using `root1`, and using `root2`. Only one of these solutions makes sense. Which one? And why? Use this to create a correct version of the procedure below.

```
(define flight-time
  (lambda (vertical-velocity elevation)
    your-answer-here))
```

## Problem 4: Flight Distance

Suppose the banana is thrown with some velocity *v*, at a starting angle *alpha* relative to the horizontal (in degrees), and from an initial elevation (in meters). We wish to compute the distance in the horizontal direction the banana will travel by the time it lands. Remember that some of the velocity vector goes into the *x* direction, and some into the *y*, as pictured in Figure 3 below.

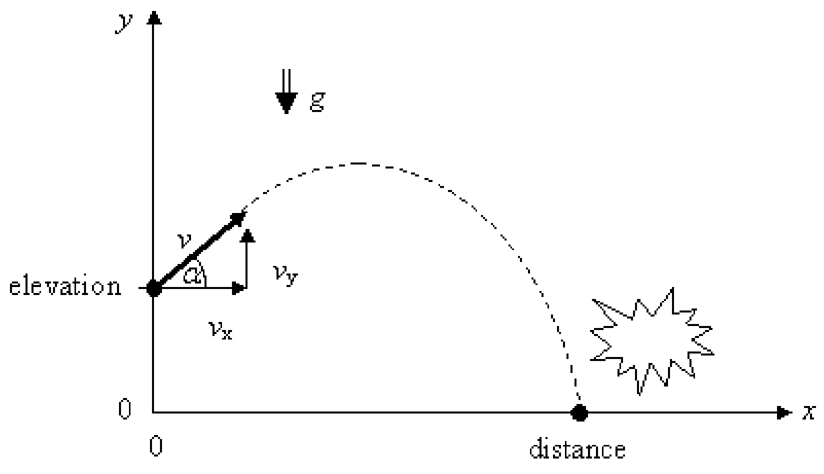

Figure 3: Motion of a banana in two dimensions, acting under gravitational acceleration *a*.

Checking the Scheme manual, you will find procedures `sin` and `cos`. To use these (which require angles in radians rather than degrees), you may also find the procedure `degree2radian` useful. It is provided in `project1.scm`, and is given below:

```
(define degree2radian
  (lambda (deg)
    (/ (* deg pi) 180.)))
```

Write a procedure `distance` that returns the lateral distance the banana thrown with given velocity, angle, and initial elevation will travel before hitting the ground.

```
(define distance
  (lambda (angle velocity elevation)
   your-answer-here))
```

## Problem 5: What's a Hit?

In order for a thrown banana to hit the target, it must land within some proximity of the target. Write a procedure `hit?` that returns `#t` when a throw comes within `burst-radius` of the target and `#f` otherwise.

```
(define hit?
  (lambda (angle velocity elevation burst-radius target-distance)
   your-answer-here))
```

## Problem 6: Acting like a big ape.

With the procedures from problems 1 through 5, we now have enough to resurrect the game. A procedure called `play` has been provided for you in the `project1.scm` file. Try it out by evaluating the play procedure definition, and then evaluating `(play 10 3 1 2)`, for example. When running, you enter values for velocity and range (in the `*scheme*` buffer) to try to hit the other target.

## Problem 7: Distance Targeting

You are merrily playing the game, when the gorilla on the left reaches out and captures you. He wants you to write some procedures to help him in targeting his throws.

The gorilla will tell you his elevation, the angle relative to the horizontal (in degrees) at which he will throw the banana, and the target-distance. The gorilla wants a `find-velocity` procedure in order to know at what velocity to throw the banana to hit the target. The gorilla will also tell you how close the banana needs to be to the target in order for the burst to destroy the target (i.e. the `burst-radius`).

We want you to write the procedure `find-velocity` that returns the desired velocity to hit the target.

**Part a:** Intuition suggests that the distance traveled will vary monotonically with the magnitude of the throw. That is, the higher the initial velocity, the further the distance. Write a procedure to verify this, that is, given an angle, an elevation and a minimum and maximum velocity, the procedure will show you which velocity results in the furthest distance. Use this to confirm the intuition stated above.

**Part b:** One way to find the best velocity is just to search sequentially over a range of possible velocities until we find one that works. However, we can be a bit more clever. We can use a binary search, in which we reduce the range of choices by half at each step. To do this, we first need two procedures `short?` and `long?` These will respectively tell you if the distance traveled by a throw with the given angle, velocity, and elevation is short of the target distance or beyond the target distance.

**Part c:** Using these tests, we can implement a procedure to find the best velocity, using the following idea. We are given a minimum and maximum possible velocity. If the maximum velocity is short of the target distance, then we simply need to see if it is close enough to hit the target. If the minimum velocity is too long, we also just need to see if it is close enough. Otherwise we can try an average of the minimum and maximum velocities. If that is close enough, we can just return the velocity. On the other hand, if that average velocity is short of the target distance, then we can try again using that average velocity as the minimum velocity, whereas if it is too long, we can try again using that average velocity as the maximum velocity. Use this idea to implement `find-velocity`.

`**Part d:** What is the order of growth of your procedure in time?`

Here is the set of procedures that you are to complete:

```
(define short?
  (lambda (angle velocity elevation target-distance)
    your-answer-here
))

(define long?
  (lambda (angle velocity elevation target-distance)
    your-answer-here
))

(define find-velocity
  (lambda (angle elevation burst-radius target-distance)
    (find-velocity-helper angle min-velocity max-velocity elevation burst-radius target-
distance)))

(define find-velocity-helper
  (lambda (angle min-vel max-vel elevation burst-radius target-distance)
     (cond ((short? angle max-vel elevation target-distance)
            your-answer-here)
           ((long? angle min-vel elevation target-distance)
            your-answer-here)
           ((hit? Angle (* .5 (+ min-vel max-vel)) elevation burst-radius target-distance)
            your-answer-here)
          your-answer-here)))
```

## Problem 8: Angle and Velocity Targeting

Write a procedure `find-throw` that determines both an angle and a velocity that hits the target. You already have code to find the velocity, now you just need to find the angle. For this, you can simply search over the range of angles from 0 to 90 degrees, stopping when one of those angles has a velocity that works.

In this case, because you have to find and communicate two values to the gorilla (which is a little tricky given the Scheme we have so far covered), you should use a procedure `(tell-gorilla angle velocity)` from inside your procedure to finally communicate your answer to the gorilla. If no angle and velocity combination works, you can return `(tell-gorilla #f #f)` . The `tell-gorilla` procedure is already defined for you in the `project1.scm` file, and an example use of `tell-gorilla` is shown below.

```
; A bogus find-throw procedure that
; simply refuses to find a correct solution, but
; instead just tells the gorilla anything (not a
; smart thing for a pet programmer to do!)
;
(define find-throw
  (lambda (elevation burst-radius target-distance)
    (tell-gorilla 45 10)))
```

You want to survive longer than the rather unwise programmer above. Implement a working `find-throw` procedure.

Hint: your procedure might try all angles (in range 0-90) in an attempt to find an angle that works, and then `tell-gorilla` a throw that comes within burst-radius of target-distance.

What is the order of growth in time of your procedure?

## Problem 9: Bouncing Bundles of Bananas

Your gorilla has an innovation: he has just acquired a bouncing banana bundle that he is very excited about. The banana bundle consists of some number (`num`) of the usual explosive bananas. The gorilla throws the bundle in the usual way, but when a bundle hits the ground, half of the bananas explode and thus propel the remaining smaller bundle (consisting of the other half of the bananas) at twice the velocity, as illustrated in Figure 4. At each bounce, the smaller bundle leaves the ground at the same angle to the horizontal as the original bundle was thrown at.

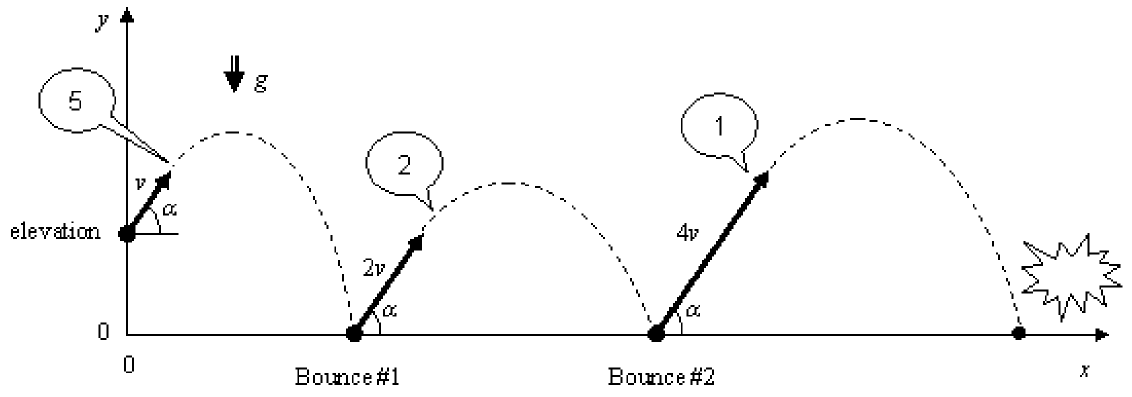

Figure 4. How a bundle of five bananas bounces. A smaller bundle of two bananas survives the first bounce and launches with twice the velocity. One banana survives the next bounce and launches with another doubling in velocity.

The utility procedure `num-survive` can be used to calculate the number of bananas which survive any given bounce (when there are `num` bananas in the bundle at the time it contacts the ground).

```
; How many bananas in a bundle survive a bounce
;
(define num-survive
  (lambda (num)
    (if (even? num)
      (/ num 2)
      (/ (- num 1) 2))))
```

Write a procedure `bundle-bounces` that returns the number of times the bundle bounces before the last banana explodes. Note that the last banana hitting the ground does not count as a bounce, as illustrated in Figure 4.

```
(define bundle-bounces
  (lambda (num)
    your-answer-here))
```

## Problem 10: If a Bundle of Bananas Could Bounce, How Far Would It Go?

Write a procedure `bundle-distance` that returns the total distance the last banana in a bundle with `num` bananas, thrown at some `velocity`, some `angle`, and from initial `elevation`, will have traveled. For example, in Figure 4 the `bundle-distance` of the original bundle thrown by the gorilla would include the distance traveled before the first bounce, the distance from the first to the second bounce, and the distance from the second bounce to the final explosion.

```
(define bundle-distance
  (lambda (num angle velocity elevation)
    your-answer-here))
```

Comment on the similarity and differences between your implementation of this `bundle-distance` procedure and the `bundle-bounces` procedure.

## Problem 11: Banana Bundle Targeting

Now that you can calculate the total distance a banana bundle will go, your gorilla agrees to set you free if you can provide a procedure `find-bundle-throw` that tells him the angle and velocity at which to throw the bundle such that the final banana hits the target. He will specify the number of bananas in the bundle, as well as the initial elevation and the distance to the target, and the burst radius as before.

You are now an experienced programmer in all things related to bananas. Based on your experience, implement this new capability. Note that you may need to implement several additional utility procedures, or variants of other existing procedures, in order to solve this problem - and thus ultimately be freed from gorilla banana land to find a warm cozy bed and get some sleep.

```
(define find-bundle-throw
  (lambda (num elevation burst-radius target-distance)
    your-answer-here))
```

## Congratulations! You have reached the end of Project 1!

We encourage you to work with others on problem sets as long as you acknowledge it (**nothing written gets exchanged!**). If you cooperated with other students, TA's, or others, please indicate your consultants' names. Otherwise, write "I worked alone using only the course reference materials."