Problem 1: Semantics

Exercise 1:

a. For conditional if e0 then e1 else e2, if e1 has type int, and if e1 and e2 both have the same type t (int), then the conditional has type t. If the then branch is taken, the resulting value has the same type as e1. The else branch produces a value of the same type as e2. Both branches must return the same type-int.

b. ‘a -> ‘a -> int

c. if env(e0) = e1
then env :: e0 -> e1
else env :: e0 -> e2

d. let is_match (e0 : int) : int =
```
match e0 with
0 -> e2
| _ -> e1
```

Exercise 2:

env :: {x=1}
env :: {x=1, f = <<fun y -> (let x = y+1 in fun z -> x+y+z, {x=1}>>}
env :: {x=1, f = <<fun y-> fun z -> x+y+z, {x=y+1}>> }
env :: {x=1, f=<<fun y -> <<fun z-> x+y+z ,{x=y+1}>> >>}
env ::{x=3, f=<<fun y -> <<fun z-> x+y+z, {x=y+1}>> >>}
env:: {x=3, f=<<fun y -> {g=f 4} <<fun z-> x+y+z, {x=y+1}>> >>}
env :: {x=3, y=5, f=<<fun y -> {g=f 4} <<fun z-> x+y+z, {x=y+1}>> >>}
env :: {x=3, y=5, f=<<fun y -> {g=f 4}, {z = g 6} <<fun z-> x+y+z, {x=y+1}>> >>}
{z = f 4 6} {x = y+1} <<fun z -> x+y=z>> fun z -> y+1+y+z
= 2(4) + 1 + 6 = 15

(Step by step):

- The first let binds x to 2
- The second let binds f to fun y, where x is still bounded in the closure. The original binding of x persists in the closure.
- The third let binds x to y+1, which is bounded in the function environment.
- Function z is binded to x+y+z
- The fourth let binds x to 3, which is not enclosed in the function environment
- The original x=y+1 is still bounded in the function environment
- The fifth let binds g to function f 4. This is enclosed in the function environment
- The sixth let binds y to 5, which remains outside the function environment

- The seventh let binds z to g 6, which is contained in the environment enclosure.
- Applying the x evaluation and value for y and z, the entire expression evaluates to 15.

Problem 4:

Overall, this project went well. We didn't encounter any problems, but we could not test the nats.ml file with very large values because we ran out of ram space on the virtual machine. All of our functions are tail-recursive, which allows for efficient and constant stack-space. Problem 1 was done by Nikita Gupta, and problems 2 and 3 were done collaboratively by both Suraj and Nikita. Suraj was able to compile and generate the code for most of the functions in the Quadtrees and Natural Numbers exercises.