MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Assignment for Week 2, Issued: Tuesday, Feb. 12

This week's work gives you practice with *higher-order procedures*, i.e., procedures that manipulate other procedures. Tuesday's lecture, the software lab, and the on-line tutor problems all explore Python's support for *functional programming* as a *means of capturing commons patterns*. The design lab extends your work from last week, by applying higher-order procedures to the design of robot *behaviors*.

Overview of this week's work

. . . to do in software lab You've seen in homework how to represent sets as lists. In software lab, you'll look at a different representation of sets that uses higher-order procedures, and also practice the material on higher-order procedures that was presented in lecture.

. . . homework to do before the start of your design lab on Feb 14 or 15

- Read the class notes and review the lecture handout.
- Do the first half of the on-line tutor problems for week 1 (the part that is due before design lab).
- Read the entire description of the design lab, so that you will be ready to work on it when you get to lab.

. . . to do in design lab Just as last week, the design lab will begin with a *nanoquiz*. The quiz may include material from the class notes, from the on-line tutor problems due that day, or from the design lab description. The quiz will be given in the first 15 minutes of lab, and if you miss it, we will not give you time to make it up before starting the lab, so please don't be late.

After the quiz, you should work with your same partner from last week. (We'll change partners in about two weeks.) This week's lab will extend the work on behaviors that you did last week to cover *non-deterministic behaviors* and using the PCAP framework to design behaviors.

. . . homework due at the beginning of your next design lab on Feb 21 or 22

- Additional problems to do using the on-line tutor.
- Lab writeup to hand in.

In general, written homework should be handed in at the beginning of the following week's software lab. But next Tuesday, February 19, is an MIT virtual Monday, so there will be no lecture. So for next week, homework should be handed in at the beginning of the design lab.

Software Lab for Week 2

Reminder:

On lab laptops or Athena terminals, you should start by executing
`athrun 6.01 update`
in a Unix terminal window. This creates the `Desktop/6.01/lab2` directory that will contain the code files mentioned in this handout.

If you are running from an Athena terminal, you should also execute
`add -f 6.01`
to get the appropriate versions of Idle and Python.

If you are using your own laptop, you should download the lab code files from the course Web site, linked from the Calendar page. Today's code file is `ps2-software-lab.py`

Lecture review on higher-order procedures

The code file for today's lab contains several of the higher-order procedures shown in lecture, including `summation`, `deriv`, and `newtonsMethod`. Here's what's in the file, so you can have it in front of you as you work on these exercises today.

```
def summation(low,high,f,next):
    s=0
    x=low
    while x <= high:
        s = s + f(x)
        x = next(x)
    return s

def fixedPoint(f,firstGuess):
    def close(g1,g2):
        return abs(g1-g2)<.0001
    guess = firstGuess
    next = f(guess)
    while not close(guess, next):
        guess = next
        next = f(guess)
    return next

def deriv(f):
    dx=0.0001
    def df(x):
        return (f(x+dx)-f(x))/dx
    return df

def newtonTransform(f):
    def g(x):
        return x - f(x)/deriv(f)(x)
    return g

def newtonTransform(f):
    return lambda x: x - f(x)/deriv(f)(x)
```

```
def newtonsMethod(f,firstGuess):
    return fixedPoint(newtonTransform(f),firstGuess)

## computing square roots by Newton's method
def newtonSqrt(x):
    return newtonsMethod(
        lambda y: y**2 - x,
        1.0)
```

Question 1: Define a simple procedure called `inc` that takes a number as input and returns that number plus 1, and a simple procedure `square` that takes a numeric input and returns the square. What will Python print at each step when you evaluate the following sequence:

```
>> inc(7)
```

How about if you evaluate

```
>> inc
```

What if you evaluate

```
def compose(f, g):
    return lambda x: f(g(x))
>> compose(inc, inc)
```

Then

```
>> compose(inc,inc)(3)
```

And finally

```
>> a = compose(inc,inc)
>> a(4)
>> b = compose(inc,square)
>> b(5)
>> c = compose(square,inc)
>> c(5)
```

Predict what the result will be at the end of each line, and then try it and see if you were correct. Check with a staff member if the results are puzzling.

Question 2: Show how to use the `summation` procedure from lecture to compute

- $1^3 + 2^3 + 3^3 + \cdots + 100^3$
- $1^4 + 3^4 + 5^4 + \cdots + 99^4$

Question 3: Show how to use the `newtonsMethod` procedure from lecture to solve the equation

$$y^4 + 2y^2 = x$$

for y, with $x = 1$. Try it for other positive values of x. (Hint: Compare the use of `newtonsMethod` in lecture for computing square roots.)

Representing sets as procedures

In homework last week, you explored representing sets using Python lists. Another way to represent a set is as a procedure that takes a value as input and returns `True` if the value is in the set, and `False` otherwise. For example,

```
def s1(x):
    return x in ['this', 'is', 'my', 'set', 'of', 'strings']
```

would represent the set of six strings. That may seem hardly different from the list representation you worked with for homework, but notice that we could also have something like:

```
def s2(x):
    return x > 99
```

and this would represent the set of all numbers greater than 99 (an infinite set).

Question 4: Define some sets (both finite and infinite). How can you test your definitions?

Question 5: For homework last week, you implemented the `setUnion` and `setIntersection` operations in terms of lists. In terms of the PCAP framework, these are *means of combination* for sets.

Define `setUnion` and `setIntersection` for sets represented as procedures and create some tests to show that they work. They take two sets as input and return a set.

Hint 1 (caveat): Remember that the results of these operations must be *sets*, in the same representation as the input sets. That is, the results produced by the means of combination should be things that themselves can be combined. For example, if `s1`, `s2`, and `s3` are sets, then you ought to be able to compute

```
setUnion(s1, setIntersection(s3,s4))
```

Hint 2: You can use Python's `and` and `or` operations to combine logical values.

If you've defined these procedures correctly, and you have defined sets `a` and `b`, then you ought to be able to evaluate

```
c = setUnion(s1, setIntersection(s3,s4))
```

What could you evaluate now, to demonstrate that your definition produces the correct set?

Question 6: Define `setComplement`. The complement of a set is all the elements in the universe that are not in the set. The procedure takes a set as input and returns a set as a result.

Question 7: Define `setDifference`. The difference between two sets, $s_1 - s_2$, is the set of elements that are in s_1 but not s_2. Try to define this using operations on sets that you have already built, rather than looking at individual elements. The procedure takes two sets as input and returns a set as a result.

Question 8: In this representation of sets, how would you define the empty set? The universal set (that contains all elements)?

In homework and design lab this week, you'll look at yet another representation for sets.

A serendipitous excursion into the complex plane

In the first part of this lab, you wrote a procedure that applied Newton's method to solve

$$y^4 + 2y^2 = x$$

for various positive values of x. There are no real-valued solutions for negative x, because the left-hand-side is always positive.

So you wouldn't expect your procedure to work when x is negative—and it doesn't. If you try it (go ahead and try) you'll see that it runs forever, never converging to an answer.[1] The same thing happens if you use the lecture code `newtonSqrt` procedure to attempt to compute the square root of a negative number.

But here's the surprise: Newton's method and all the algebra behind it works even if the numbers involved are *complex numbers*.

Not only that, but Python's addition, subtraction, multiplication, and division operations work automatically with complex numbers. Python represents complex numbers as $a + bj$, for example:

```
>>z = 3 + 2j
>>w = 7 - 10j
>> z+w
(10-8j)

>> z*w
(41-16j)

>> z/w
(0.006711409395973130 8+0.29530201342281887j)
```

Consequently, you can make Newton's method work to find complex roots of equations, *without changing the code at all*, except to pick an initial value that's a non-real number, e.g., instead of starting at 1.0, start at `1j`.

Question 9: Modify your program for solving

$$y^4 + 2y^2 = x$$

and find solutions to the equation for $x = -10$ and $x = -100$.

Question 10: Show that the same idea works with the Newton's method square root procedure, for computing square roots of negative numbers.

Question 11: What happens if you ask your modified program for the square root of a positive number? Make another change to your program so that if the result is nearly real (the complex part is very small), then you return a float instead of a complex. Note that if `c` is a complex number, then `c.real` will return the real part and `c.imag` will return the imaginary part.

[1] If IDLE is running, apparently forever, or at least longer than you want to wait, you can type CONTROL-C to interrupt it. If you don't see a shell prompt, you may need to select *Restart Shell* from the *Shell* menu.

We didn't deserve to be that lucky, did we?[2]

At the end of this lab, go to the on-line Tutor at `http://sicp.csail.mit.edu/6.01`, choose the problem set for this week, and paste all your code, including your test cases, into the box provided in the "Software Lab" problem. Do this even if you have not finished everything.

Software Lab Explorations I: Sets of prime numbers

This exploration is worth 2 points. Together with the other explorations from the software and design labs, this constitutes one 10-point exploration assignment.

These Explorations sections are not required. However you can get additional credit in 6.01, by doing explorations. (See the grading policy).

Continue working with the procedural representation of sets by completing the following problems:

Exploration 1: Define a procedure `divisibleBy` that takes an integer argument `n`, and returns the set of integers that are divisible by `n`.

Exploration 2: Define a procedure `singleton` that takes an integer argument `n`, and returns the set containing the single integer `n`.

Now let's put these together to do something more interesting:

There are all sorts of ways to generate prime numbers. One very beautiful (but not very efficient) way is by a *sieve*: You start with the set `P` of potential primes, which is initially all positive integers between 2 and `n`. Then for `i` between 2 and `n`:

- If `i` is in `P`, then it's prime. Remove from `P` all elements that are divisible by `i` but not equal to `i`.

At the end of this process, the set `P` will contain only the primes between 2 and `n`.

- Implement a procedure `primesUpTo` that takes an integer argument `n`, and returns the set of prime numbers `p` where $2 \leq p \leq n$ and `n`. Use your set operations. It shouldn't require more than 6 lines of code.

Software Lab Explorations II: Numerical integration

This exploration is worth 2 points. Together with the other explorations from the software and design labs, this constitutes one 10-point exploration assignment.

As you know from calculus, the definite integral of a function `f` can be approximated by:

[2] Not that we'll be looking at this in 6.01, but the mathematical phenomena around Newton's method and complex numbers are fascinating. Polynomials have multiple roots, and choosing different starting values for the iteration changes which root you end up at. The set of starting values that produce a given root is in general an extremely complicated shape—an example of a mathematical object called a *Julia set*, which arises in the study of *fractals* and *chaos*. See `http://aleph0.clarku.edu/~djoyce/newton/newton.html` for a discussion and for some pretty pictures.

$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \cdots \right] dx$$

You've seen how to use higher-order procedures to express the mathematical idea of summation in computing sums. We can use the same basic idea to do numerical integration.

Exploration 3: Write a procedure `integral` that takes as input a function `f` and lower and upper bounds `a` and `b` and computes an approximation to the integral of `f` over the interval $[a, b]$. For example,

```
integral(square, 0.0, 1.0)
```

should be close to 1/3—just how close depends on how you choose `dx`.

Exploration 4: Now implement two other methods of integration: the trapezoidal rule and Simpson's Rule. Compare the accuracy of these three methods for several small values of `dx`. Can you say anything systematic about how the error depends on `dx`?

Homework due before design lab

The following homework is due before your section's design lab:

1. Read the course notes for week 2 and review the lecture notes.
2. Read the section on Python sets in van Rossum's tutorial (section 5.4), linked from the class site resources page.
3. Log in to the online tutor and do problems that are assigned. This is the first half of this week's tutor problems.
4. Finally, read through the entire description of the design lab so that you'll be prepared to work on it. Note the questions you'll need to answer in class. The design lab *nanoquiz* will cover some of these items.

Design Lab for Week 2: Higher-order procedures and nondeterministic behaviors

On a lab laptop, you should start by executing
`athrun 6.01 update`
in a Unix terminal window. This creates the `Desktop/6.01/lab2` directory that will contain the code files mentioned in this handout. Today's code file is `behaviorCombinationSkeleton.py`

Lab background: Behaviors and behavior combination

How does a robot faced with a choice of several actions decide which one to do next? This week's topic deals with *non-deterministic behaviors* as a framework for choosing. Our implementation of non-deterministic behaviors is based on *higher-order procedures*, and it illustrates the general PCAP perspective this course takes on engineering complex systems:

> Start with *primitive elements*, build more complex elements through *means of combination*, and use *means of abstraction* to capture common patterns of use.

Just as we did last week, we'll be writing programs to give our robots *behaviors*, which are purposeful ways of interacting with the world: for example, avoiding obstacles or just wandering around. As you saw last week, a behavior is exhibited by a sequence of actions, where an *action* is some simple thing a robot might do, like go forward or turn. The essence of the behavior lies in the choice, based on the current circumstances, of which action to do next.

If a behavior is to be executed in isolation, as the behaviors we wrote last week were, then it's appropriate for a behavior to be a function from the current sensor values to a single action, which will then be executed. But sometimes we would like to regard behaviors as not specifying actions explicitly, but rather as providing constraints on what actions are possible. So, for instance, we might have a behavior that constrains the robot to moving forward or stopping (but not moving backward). Then we might have a second behavior that constrains the robot to moving fast. We might then *combine* the two behaviors so that the robot obeys both constraints, which in this case would constrain the robot to moving forward fast.

In order to implement this view of behaviors as providing constraints on actions, we'll think about *nondeterministic* behaviors, formally, as functions that map sensor data to *sets* of actions, rather than to individual actions as we did last week. If the output of a behavior is a set of three actions, it means that, from the perspective of that behavior, any of those actions is satisfactory.

Implementing non-deterministic behaviors

In this week's assignment, we'll create some primitive non-deterministic behaviors and combine them to produce compound behaviors. In our Python implementation, we'll represent each behavior as a procedure that takes sense data as input and returns a set of actions. As a consequence of this representation choice, Python's mechanisms for manipulating procedures as first-class objects (e.g., naming procedures, passing procedures as arguments to procedures, returning procedures as values from procedures) will be available to us as means of combination and abstraction for designing behaviors.

Sets of actions

You've now seen three different Python implementations of sets: as lists, as procedures, and as a built-in data structure. For this lab, we'll be using Python's built-in sets.

Remember that our actions from last week were lists like

```
go = ["go", [speed, 0]]
```

Unfortunately, Python doesn't allow you to include lists as elements of sets.[3] So we'll change our implementation of actions to use *tuples* instead of lists, because it's fine to include tuples as elements of Python sets:

```
go = ("go", (speed, 0))
```

We'll also slightly expand our repertoire of actions to include both moving fast and moving slowly:

```
speed = 0.1
stop = ("stop", (0, 0))
go = ("go", (speed, 0))
goFast = ("goFast", (speed*2, 0))
left = ("left", (0, speed))
right = ("right", (0, -speed))
leftFast = ("left", (0, speed*2))
rightFast = ("right", (0, -speed*2))
```

Then, analogous to last week, we'll have brains of the form:

```
def step():
    doAction(selectAction(behavior(collectSensors()), stop))

def collectSensors():
    return [sonarDistances(), pose()]

def doAction(action):
    (name, (transVel, rotVel)) = action
    print name
    motorOutput(transVel, rotVel)
```

The only new thing here besides the change from lists to tuples is `selectAction`, which picks an action from the set of actions returned by a behavior.

```
def selectAction(actionSet, default):
    actionList = [a for a in actionSet]
    if len(actionList) > 0:
        return a[0]
    else:
        print "No legal action!!"
```

This procedure returns some arbitrary action in the set. In `behaviorCombinationSkeleton.py` we've also provided a procedure called `selectActionRandomly`, which returns a action from the set, chosen uniformly at random. You can experiment in this lab with both methods of selecting actions.

[3] This restriction arises from the fact that Python's designers want to enforce the condition that elements of sets are *immutable*, i.e., cannot be modified (as in modifying a list by removing an item). This issue goes deeper into Python implementation topics than we want to go in 6.01, but you can consult books on Python if you're interested.

Primitive non-deterministic behaviors

Here is a very simple non-deterministic behavior:

```
def noCrash(sensors):
    # Assume turning never crashes (in fact, it occasionally does)
    actions = set([left, right, stop])
    # Moving forward can crash even if only sensor 0 or 7 is blocked
    if frontClear(sensors) and leftClear(sensors) and rightClear(sensors):
        actions.add(go)
        actions.add(goFast)
    return actions
```

To run this in SoaR as a brain, we could use:

```
behavior = noCrash
def step():
    doAction(selectAction(behavior(collectSensors()), stop))
```

Question 12: What is the type of `noCrash`? That is, what kind of thing does it take as input and what kind of thing does it return as a result?

Question 13: Write a non-deterministic behavior that only allows the robot to move forward or stop.

Question 14: Run the `noCrash` behavior. Try it both with using `selectAction` and with `selectActionRandomly` and compare the results.

Checkpoint 1: To be completed by 30 minutes after the beginning of lab

Show your results for questions 12–14 to your LA and be ready to explain them.

Behavior combinations

Now, we can create means of combination for behaviors. One behavior combiner is *or*: if `b1` is a behavior and `b2` is a behavior, then an action is allowed by the combination, in some situation, if it would have been allowed by `b1` or by `b2`. Here is the code for a procedure `orNDB`, which takes two behaviors as arguments, and returns a new behavior, which is the *or* of those behaviors:

```
def orNDB(ndb1, ndb2):
    def resultNDB(sensors):
        return ndb1(sensors) | ndb2(sensors)
    return resultNDB
```

Question 15: Why would it not work to define `orNDB` this way?

```
def orNDB(ndb1, ndb2):
    return ndb1(sensors) | ndb2(sensors)
```

Question 16: Define the procedure `andNDB`, which takes two behaviors as arguments and returns a new procedure that allows an action only if it would have been allowed by both of the input behaviors.

Question 17: Define some dummy behaviors that ignore the sensor conditions and test your `andNDB` to be sure it's working correctly.

Question 18: Use `andNDB` to make a new behavior that combines `noCrash` and your behavior from question 13 that only moves forward or stops. Run it in the simulator. How does it behave?

Question 19: Define a procedure `ifNDB(c, ndb1, ndb2)` that takes a condition `c` and two NDBs as arguments. The condition is a function of type *sensors* $\rightarrow$ *boolean*. Your procedure should return a new behavior that, if the condition is true, allows the actions that would have been allowed by the first behavior; otherwise it allows the actions that would have been allowed by the second behavior.

Checkpoint 2: To be completed by 70 minutes after the beginning of lab

A more complex combination

A very useful type of behavior combination is *prioritized and*. It takes a list of behaviors $b_1 \ldots b_k$. If there are any actions that are allowed by all of b_1 through b_k, then the set of those actions is the output of this behavior. If there are no such actions, then if there are any actions allowed by b_1 through b_{k-1}, those are the output of the combined behavior. This process of backing off is repeated until an allowable action is found. As an illustration, imagine that there are four behaviors, and in the current situation, they allow the following action sets:

$$\begin{aligned} b_1: &\quad \{a_1, a_2, a_4\} \\ b_2: &\quad \{a_1, a_3, a_4\} \\ b_3: &\quad \{a_3, a_2\} \\ b_4: &\quad \{a_3, a_4\} \end{aligned}$$

There are no actions that are in the intersection of all four sets. So, we drop b_4, and find that there are still no actions in the intersections of the first three sets. So, we drop b_3 as well, and find that both a_1 and a_4 satisfy the first two behaviors, and so those actions are returned.

If you know about Asimov's Three Laws of Robotics, you can think of them as having the structure of a *prioritized and*:

1. A robot may not injure a human being or, through inaction, allow a human being to come to harm.
2. A robot must obey orders given to it by human beings, except where such orders would conflict with the First Law.

3. A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

A robot is supposed to satisfy all three of these requirements if possible. If not, it should try to satisfy the first two; and, failing that, it should satisfy the first one.

Question 20: What is `prioritizedAnd` of the following list of sets?

$$b_1: \quad \{a_1, a_2, a_4\}$$
$$b_2: \quad \{a_2, a_3, a_4\}$$
$$b_3: \quad \{a_1, a_2\}$$
$$b_4: \quad \{a_1, a_4\}$$

Question 21: Under what circumstances would the `prioritizedAnd` of a list of sets be empty?

Question 22: Write `prioritizedAndNDB`. Test it outside of soar, on the example above, and other examples of your construction. Here's how you might go about testing:

```
def b1():
    return Set([1, 2, 3])
def b2():
    return Set([2, 3, 4])
def b3():
    return Set([1, 2])
def b4():
    return Set([1, 4])
prioritizedAndNDB([b1, b2, b3, b4])
```

Hint: In thinking about how to write `prioritizedAndNDB` remember that in last week's tutor problems you wrote a procedure `fullintersection`. That doesn't solve the problem, but it could be a useful building block. Note that you can't just copy that code wholesale because it used a different representation of sets.

Question 23: Use `prioritizedAndNDB` to combine `noCrash` with a behavior that insists on moving forward. Test it in the simulator. What happens?

Checkpoint 3: To be completed by 2 hours after the beginning of lab

New behaviors

You can define a set of new behaviors and see how they interact using the different methods of behavior combination that you implemented above. For example, in our own playing with this, we defined a set of behaviors, including

- `noCrash` (as above)
- `noTurnTowardWall`, which disallows turning left when there's an obstacle immediately to the left and turning right when there's an obstacle immediately to the right
- `move`, which allows all actions but `stop`
- `forward`, which allows `go` and `goFast`
- `fast`, which allows `goFast`, `leftFast`, and `rightFast`

We found that putting them all together in a *prioritized and* generated a nice wandering behavior.

Question 24: Define three new behaviors.

Question 25: Use the combination methods you've implemented to put these new behaviors together with each other and/or our existing behaviors to generate some interesting new robot behavior.

Question 26: Test your behaviors in simulation and on a robot. Explain what they do and why.

Checkpoint 4: To be completed by the end of lab

Homework due in your lab on February 21 or 22

1. Log in to the online tutor and complete the second half of this week's tutor problems.
2. Write up and hand in the answers to exercises 19–26.
3. Hand in your code and test cases from the software lab.

Concepts covered in this assignment

Here are the important points covered in this assignment:

- One general perspective on engineering complex systems is to start with *primitive elements*, build more complex elements through *means of combination*, and use *means of abstraction* to capture common patterns of use.
- As a programmer, you have a lot of powerful tools at hand for *modeling and representation*. To make effective use of these, it's important to use *abstractions*, so that you can avoid thinking about all details of a system at the same time.
- *Higher-order procedures* can be powerful tools in computer modeling because the computer language's capabilities for manipulating procedures—naming, functional composition, parameter passing, and so on—can be used directly to support means of combination and abstraction.

Week 2 Design Lab Explorations: Numerical priorities

This exploration is worth 6 points. Together with the explorations from the software lab, this constitutes one 10-point exploration assignment.

In the design lab we looked at behaviors that computed sets of actions. The idea is that any of these actions are consistent with achieving the goal of the behavior. The *prioritized and* combination allows us to express some priorities among different primitive behaviors but it still treats all the actions returned by a single behavior as equivalent. And, for example, the fact that a group of behaviors all return one particular action does not make that action more likely to be chosen. In this exploration, we will extend our framework to include numeric priorities on individual actions. The numbers will be used to encode a behavior's preference for the different actions.

There are two broad categories of numerical preferences that we can imagine using:

- *Ordinal* – In ordinal preferences, the numbers are interpreted as rankings, so a 5 might be most preferable and 1 least preferable.
- *Cardinal* – In cardinal preferences, the magnitude of the number is meaningful, so a preference of 4 is twice as strong as a preference of 2.

We will assume that behaviors are defined so that their input is a list of sensor values (as before) and that their output is a **function** which when called with an action will return the numerical preference for that action. For example:

```
>>> pref = behavior(collectSensors())
>>> pref(go)
2
>>> pref(left)
0
```

One key question is how to combine the preferences of individual behaviors for particular actions so as to select the best action. If one is using cardinal preferences, one can sum the preferences of the behaviors for each action and choosing the action with the largest sum. Combining ordinal preferences is more subtle and an important topic of research in economics and, more recently, in search engines (search for "rank aggregation").

Exploration 5: Decide on an approach to ordinal preferences and how to combine them. Explain your reasoning. Implement your approach on a simple example.

Exploration 6: Decide on an approach to cardinal preferences and how to combine them. Explain your reasoning. Implement your approach on a simple example. Optionally, you can try generalizing the notion of an action to be some rectangle in the translational velocity vs. rotational velocity space and then having the preference be some continuous function on that space.

Exploration 7: Demonstrate either your ordinal or cardinal preference approach on some more test cases, for example, extend some of the examples from this week's design lab. Can you combine the *seek* behavior from last week with some form of obstacle avoidance behavior? Describe your experiences.

Exploration 8: Reflect on the modularity of your system. Can you design the primitive behaviors without understanding how they are going to be combined?