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

Assignment 4, Issued: Tuesday, Feb. 26: Revised

Revised Version

Overview of this week's work

In software lab

- Work through the software lab.
- Submit whatever work you have finished at the end of the lab into the tutor.

Before the start of your design lab on Feb 28 or 29

- Read the class notes and review the lecture handout.
- Do the on-line tutor problems in section PS.5.2.
- Read the entire description of the design lab, so that you will be ready to work on it when you get to lab.

In design lab

- Take the nanoquiz in the first 15 minutes; don't be late.
- Work through the design lab with a partner, and take good notes on the results of your work.

At the beginning of your next software lab on Mar 4 or 5

- Submit online tutor problems in section PS.5.3.
- Submit written solutions to questions 1 through 14 as well as 17 and 18. All written work must be done individually, and conform to the homework guidelines on the web page.

- **You will need SoaR in this software lab, so if you don't have SoaR installed on your own machine, use a lab laptop or Athena station.**
- **Get the lab4 files via `athrun 6.01 update` or from the home page. We have given you several that have `Skeleton` in the name. You should make a copy of those files and rename them to remove the `Skeleton`. You will get failing `imports` otherwise.**

Software Lab: Combinations of state machines

In class we discussed three methods of making new state machines out of old ones: serial composition, parallel composition, and feedback composition. Below (and in the file `SimpleSMSkeleton.py`) is the skeleton of the `SerialSM` class. We've provided the constructor method, which take state machines as input and construct a new, composite state machine.

```
class SerialSM:
    def __init__(self, sm1, sm2):
        self.m1 = sm1
        self.m2 = sm2

    # Be careful to keep the timing consistent.  The input to m2 has
    # to be the current output of m1, not the output after it is
    # stepped.
    def step(self, input=None):
        # Your code here

    def currentOutput(self):
        # Your code here
```

Question 1: Write the `step` and `currentOutput` methods for a serial SM. Test it by combining two `incr` machines (the definition is in `SimpleSMSkeleton.py`). Draw a picture showing how the machines are connected. First, figure out what the result *ought* to be by filling out a table like this one (remember that, by definition, $output_1 = input_2$:

step	$input_1$	$state_1$	$output_1$	$state_2$	$output_2$
0					
1					
2					
3					

Then test to be sure your machine is doing the right thing.

Question 2: Compare the results of composing a `sum` machine (defined in `SimpleSMSkeleton.py`), which outputs the sum of all of its inputs so far with an `incr` machine. Predict what the result ought to be by filling out a table like this one:

step	$\mathtt{input}_1$	$\mathtt{state}_1$	$\mathtt{output}_1$	$\mathtt{state}_2$	$\mathtt{output}_2$
0					
1					
2					
3					

Then be sure you're getting the right result.

Feedback The following class defines a feedback state machine. Its `__init__` method takes a state machine, `sm`, and returns a state machine with no inputs, which consists of `sm` with its output fed back to its input. We'll define the output of the feedback machine to be the same as the output of `sm`.

```
class FeedbackSM:
    def __init__(self, sm):
        self.m = sm

    def step(self, input=None):
        return self.m.step(self.m.currentOutput())

    def currentOutput(self):
        return self.m.currentOutput()
```

Now, we can use this to couple two machines together, with the outputs of one machine serving as the inputs to the other, and vice versa.

```
def simulatorSM(m1, m2):
    return FeedbackSM(SerialSM(m1, m2))
```

Question 3: In the code file, you'll find the definition of `makeIncr`, which takes an initial state as an argument, and then thereafter updates its state to be its input plus 1. If we make the following coupled machine, we get a potentially surprising string of outputs:

```
>>> fizz = simulatorSM(makeIncr(10), makeIncr(100))
>>> run(fizz)
100
11
102
13
104
15
106
17
108
19
110
```

Explain why this happens. Draw a picture of the objects involved in this machine and say exactly what state variables they contain. Fill in a table describing the inputs, states, and outputs of each component machine at each step.

Question 4: Imagine a robot driving straight toward a wall. We would like to use a state machine to model this system, where the state is the distance from the robot to the wall in meters (pick an initial distance for your initial state) and the input is the robot's current forward velocity in meters per second. Assume that each state transition corresponds to the passage of 0.2 seconds. Write a primitive state machine to model this system. Hint: the distance should go down as the machine steps.

Question 5: Now, let's think of the controller that this robot might be executing. Assume the robot would like to stop at some distance $\mathtt{d}_{Desired}$ from the wall. The controller can be modeled as a state machine that receives, as input, the current distance to the wall and generates, as output, a velocity. For now, assume there's no inertia involved, and so the controller can choose any new velocity it wants, independent of its previous velocity. In particular, let's assume it selects a new velocity that is equal to $\mathtt{k}(\mathtt{d} - \mathtt{d}_{Desired})$, where $\mathtt{d}$ is the current distance to the wall, $\mathtt{d}_{Desired}$ is the desired distance to the wall, and $\mathtt{k}$ is a "gain" constant. Write a primitive state machine to model this controller.

Question 6: Now, use the `simulatorSM` function to put these two machines together. Run the coupled machine. Try a large value of `k` and a small value of `k`. What happens? (By setting `SimpleSM.verbose` to `True`, you can get `PrimitiveSM.step` to print out information about each step of each primitive machine.)

Question 7: You can plot data from inside soar. If all you want to do is plot, the best thing is to make a brain with only a setup method, and put your plotting commands in there. Then, when you run the brain, you'll see your data plotted in a window. The `SimpleSMBrainSkeleton.py` file defines a `setup` function that will plot some data.

We have modified `run`, in the `SimpleSMSkeleton.py` file, so that it returns a list of all the outputs generated by the machine. So, you can just copy the `setup` and `step` definitions from `SimpleSMBrainSkeleton.py` file to the file with your state-machine definitions, call `run` inside of `setup` to get some data, and then plot it using `graphDiscrete`. If you select that file as a brain in soar, it will run your `setup` method.

Produce plots of the distance between the robot and the wall, over time, with two interestingly different values of `k`. Note that the graphing windows have a Save button that lets you save the graphs to a Postscript (.ps) file.

Now, check your understanding with the following question:

Question 8: Ralph Reticulatus correctly implements `SerialSM.step`, and then makes a common mistake, and tries to run the following code:

```
rm = makeSumSM()
ralph = SerialSM(rm, rm)
>>> transduce(ralph, [1]*10)
[0, 0, 2, 6, 14, 30, 62, 126, 254, 510, 1022]
```

(Note that this result depends on how, precisely, Ralph wrote his `SerialSM.step` method. There are multiple ways to write that method correctly that will result in different outputs in this case; but none will generate the desired sequence.)

Rita Rocksnake recognized Ralph's problem, and reminded him that if we were working with two bank accounts, he'd make two different instances of the `Account` class. Rita wrote this code, instead:

```
rita = SerialSM(makeSumSM(),makeSumSM())
transduce(rita, [1]*10)
[0, 0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

She got the intended result. Explain why. Draw a picture of all of the objects in both Ralph and Rita's attempts, and say exactly what say what state variables they contain.

Software Exploration: New primitive state machines

This is worth 2 out of 10 points for the Week 4 exploration.

In the basic work of this lab, we approached the serial, parallel, and feedback composition of machines by defining new classes of state machines, in such a way that, for example, an instance of a serial composition state machine contained instances of the two original state machines, asked them to store their state internally, and simply asked them to step when necessary.

That approach works well when the goal is only to be able to run the composite state machines. However, later in the course, we will find that state machines can be used as models of an environment, for example, to plan a sequence of actions by trying out different action sequences in the model and seeing which ones perform the best. In order to do that, we need to know the transition function for the composite state machine as a single function.

So, in this exploration, we will pursue another strategy for composing state machines, in which we provide a function `makeSerialSM` that takes as input two *primitive* state machines and returns another *primitive* state machine. That means, it must construct new transition, output, and initialization functions that describe the operation of the entire serial state machine, and use them as arguments to the constructor of the `primitiveSM` class. To write such a function, you have to start by thinking carefully about what the state of the composite machine will be.

Exploration 1: Write the function `makeSerialSM` of type

$$(\mathit{primitiveSM}, \mathit{primitiveSM}) \rightarrow \mathit{primitiveSM}$$

Exploration 2: Write the function `makeParallelSM` of type

$$(\mathit{primitiveSM}, \mathit{primitiveSM}) \rightarrow \mathit{primitiveSM}$$

Exploration 3: Write the function `makeFeedbackSM` of type

$$\mathit{primitiveSM} \rightarrow \mathit{primitiveSM}$$

Exploration 4: Demonstrate these composition methods using various primitive machines that you constructed in the earlier parts of the lab.

Design Lab: Combinations of terminating behaviors

In lab 2, we explored a system for making primitive behaviors and combining them. In that case, our behaviors were all operating at once, and their outputs were expressed as sets of actions. It's possible to do a lot of different things this way, but many people are frustrated by the inability to ask their robot to do things sequentially, like drive up to the wall and then turn left. In this lab, we'll develop a new system of behavior definition and combination, which is yet another instance of the PCAP idea, that allows sequential combination of behaviors.

Terminating state machines

So far, the state machines that we have looked at all run forever, performing a transduction from a potentially infinite stream of inputs to a potentially infinite stream of outputs. Sometimes our work feels like that too! But sometimes, jobs actually get done, and we move on to new tasks.

Our first step will be to build on our original state-machine abstract data type, to construct a *terminating state machine* ADT. In addition to providing the `step` and `currentOutput` methods that regular SM's provided, terminating state machines have to provide two more methods:

- `done`: $() \rightarrow \mathtt{Boolean}$, which is `True` when the machine has terminated; and
- `reset`: $() \rightarrow ()$, which resets the machine back to its original initial state.

In other words, a terminating state machine is one where we can ask if it is "done." A machine is said to have *terminated* when its `done` method returns `True`. The way the machine decides if it has terminated is by computing a function of the machine state, called `terminationFunction`.

We can take advantage of the object-oriented programming facility of *inheritance* to build terminating state machines. Rather than redefining a whole new class of primitive terminating state machines, we can make it be a subclass of the `PrimitiveSM` class. Here's how it goes:

```
class PrimitiveTSM(PrimitiveSM):
    def __init__(self, transitionfn, outputfn, initStatefn,
                 terminationfn = lambda x: False):
        self.terminationFunction = terminationfn
        self.initialStateFunction = initStatefn
        PrimitiveSM.__init__(self, transitionfn, outputfn, initStatefn)

    def done(self):
        return self.terminationFunction(self.currentState)

    def reset(self):
        self.currentState = self.initialStateFunction()
```

Here, we had to redefine the `__init__` method to do some additional work. It has a new argument, `terminationfn`, which defaults to be a function that always returns `False`; so, the first thing it does is to store that function. Then, it also stores the `initStatefn` argument, because it will need it to implement the `reset` method. Finally, it calls the `__init__` method of the `PrimitiveSM` class to do the rest of the work.

We don't need to change the `step` or `currentOutput` methods from the way they work on `PrimitiveSM`s, so we inherit them from `PrimitiveSM` and don't need to say anything about them.

Then, we define the `done` method, which simply calls the `terminationFunction` function on `self.currentState`. Who defined `self.currentState`? The `__init__` method of `PrimitiveSM`.

And we define the `reset` method to reset the current state to whatever value is returned by `self.initialStateFunction`.

Sequential combination

Now, we'd like to implement some means of combining these machines so we can make more complex overall sequential structures. How can we make this happen?

We'd like to define a new class of terminating state machines, called `SequentialTSM`, that takes as an initialization argument a list of terminating state machines. Running the sequence amounts to running the first machine until it terminates, then the second machine until it terminates, and so on.

To do this the sequence machine keeps track of which component machine it is currently running, and keeps calling that machine's associated `step` function until it has terminated; then starts running the next machine in the sequence, and so on. The overall sequence machine terminates on the step after the last component machine terminates.

The code below contains all but the `step` and `done` methods for the class, which you should complete. Our `reset` method actually sets the initial state, so be sure to reset any component SM before beginning to step it.

```
class SequentialTSM():
    def __init__(self, smList):
        self.smList = smList
        self.reset()

    def reset(self):
        self.counter = 0
        self.smList[self.counter].reset()

    def currentOutput(self):
        if not self.done():
            return self.smList[self.counter].currentOutput()
```

Here are some simple state machines and combinations to use when you're testing your work, below.

```
def makeCharTSM(c):
    return PrimitiveTSM(lambda s, i: True,      # transition
                        lambda s: c,            # output
                        lambda: False,          # initial state
                        lambda s: s)            # done
def makeTextSequenceTSM(str):
    return SequentialTSM([makeCharTSM(c) for c in str])
```

The `makeCharTSM` procedure takes a character `c` and returns a terminating state machine whose state is either `True` or `False`. If the state is `True`, then the machine is done. The initial state is `False`, and one step will change the state to `True`. The output is the character `c`. So, this machine runs for one step and generates the output sequence [`c`].

The `makeTextSequenceTSM` procedure takes a string as an argument and returns a `SequentialTSM` that is made up of individual primitive TSMs that output a single character. So, if you `run` the machine `makeTextSequenceTSM('abcd')`, the output sequence should be `['a', 'b', 'c', 'd']`. (Note that we have provided a new version of `run`, that runs a machine until it terminates (or until a limit is reached, whichever comes first).

Use these procedures to test your `SequentialTSM` class, below.

Question 9: Write the `step` and `done` methods of the `SequentialTSM` class. Be sure that each call to `step` of the sequential machine results in exactly one call to `step` of some component machine. The sequential machine should terminate one step after the last of its component machines does (that is, if the `done` method of the last component machine first returns `True` on step n, then the sequential machine should first return `True` on step $n + 1$).

Checkpoint 1: To be completed by 45 minutes after the beginning of lab

Terminating behaviors

We can use our new sequencing abilities to get the robot to perform behaviors in sequence. We'll concentrate, this week, on building deterministic behaviors that specify completely how the robot is to be controlled; but we'll combine them by building sequential combinations of them.

A *terminating behavior* is a TSM that transduces a sequence of sensor inputs (each of which is our usual list of sonar readings and a pose) into a sequence of actions. Here's the basic brain structure for using a terminating behavior.

```
def setup():
    robot.behavior = yourTSMHere
    robot.behavior.reset()

def step():
    if robot.behavior.done():
        doAction(stop)
    else:
        doAction(robot.behavior.step(collectSensors()))
```

If you defined a new terminating behavior class, like `DoTheMacarena`, then you'd have `DoTheMacarena()` in the code above in place of `yourTSMHere`.

In the `setup` function, we make whatever calls are necessary to construct the terminating state machine that will generate the robot's behavior. We store that machine in the variable `robot.behavior`. You can think of `robot` as an object that will persist across invocations of `setup` and `step` in the robot brain, and will let us store the state of our behavior (which is stored in the state of a TSM).

In the `step` function, we check to see if the behavior is done, and, if so, we stop. Otherwise, we collect a fresh set of sensor values, feed them as input into the behavior machine's `step` method, get an action as an output from that machine, and issue the associated command.

Notice that the brain and the behavior each have their own `step` method, and that these are different. This is an example of generic functions, implemented with the aid of object-oriented programming. In this implementation, the brain's `step` method calls the behavior's `step` method as long as the behavior is not `done`.[1]

[1]Are you wondering why we refer to the brain's `step` as a *method*, when you don't see any class definition here? There really is a `Brain` class with `step` and `setup` methods, but the class definition is buried inside the implementation of SoaR. There's also a `robot` object that the brain uses in order to store attributes. That's what lets the `setup` procedure store something as `robot.behavior`, which can be referenced by the brain's `step`. You can use `robot` for storing anything that you like, and it will be local to the particular instance of the brain that SoaR is running.

Let's start by building some basic terminating behaviors for the robot. Here's a somewhat stupid[2] behavior, to illustrate how it might work. Imagine that we wanted to define a new primitive terminating behavior object, called `GoForwardUntilDeltaX`, which moves the robot forward until its x coordinate has increased by some value `deltaX`. We might write code like this:

```
class GoForwardUntilDeltaX:
    def __init__ (self, deltaX):
        self.deltaX = deltaX

    def reset(self):
        (sonars, pose) = collectSensors()
        self.initialX = pose[0]
        self.currentX = pose[0]

    def step(self, sensors):
        (sonars, pose) = sensors
        self.currentX = pose[0]
        return self.currentOutput()

    def self.currentOutput():
        if self.done():
            return stop
        else:
            return go

    def done(self):
        return self.currentX > self.initialX + self.deltaX
```

The initializer for the class just remembers the delta value. When the behavior is reset (which must be done before it is run for the first time), it looks at the robot's current pose, and the initial x coordinate, both as `self.initialX`, which will remain the same until the behavior is reset again, and as `self.currentX`, which will be updated on each step. On each step, the behavior selects the x coordinate of the pose from the sensor readings and remembers it. Then it generates an appropriate output, by calling its own `currentOutput` method. This is a behavior, so the output should be an action. The `currentOutput` method checks to see if the behavior is done; if so, it returns `stop` and otherwise it returns `go`. The `done` method returns the value `True` when the x coordinate of the current pose is greater than the x coordinate of the initial pose by at least `self.deltaX`.

Implement the following primitive terminating behaviors and test them by themselves using the brain code found in `TSMBrainSkeleton.py`. That file also contains some helper functions that you might find useful.

When you're testing terminating behaviors, you'll need to reload the brain inside SoaR once a behavior has terminated. Why? Because the program has to be run again from the beginning; if it just continues from where it was, the behavior might still think it's done.

[2]Why stupid? Because it's hard to imagine why it would be useful to move in whatever direction the robot is pointed until its global x coordinate has increased by some amount. It's entirely possible that, given the heading of the robot, x will decrease or remain unchanged as the robot moves forward.

Question 10: Define a class `TBDrive`, with an `__init__` method that takes as an argument a distance `d` to move. It should move forward a fixed distance, `d`, from the robot's position at the time the `reset` method is called, along the current heading, and then terminate. The class should be a terminating behavior in the sense that it provides `step`, `currentOutput`, `reset`, and `done` methods. *Caveat:* Remember that the robot might be moving at an arbitrary angle with respect to its global coordinate system.

Question 11: Define a class `TBTurn`, with an `__init__` method that takes as an argument an angle `α` to rotate through. It should be a terminating behavior, as above. It should rotate `α` radians from the robot's heading at the time the `reset` method is called, and then terminate. *Caveat:* Remember that `pose()` returns an angle between 0 and 2π, so finding the difference between two angles is not trivial. We have provided a helper in `TSMBrainSkeleton.py` function you may find useful.

Question 12: Test these behaviors in the simulator, and take note of their performance.

Think carefully about an appropriate termination condition to use in each case. **Don't start implementing these until you've talked with your LA about your plan!**

Checkpoint 2: To be completed by 1.5 hours after the beginning of lab

Hip to be Square

Question 13: Now, use sequencing and your new primitives to make the actual robot drive in a one meter square. Explain how your program works. Draw a diagram of the different class instances involved.

Question 14: Execute your program for driving in a square repeatedly. Describe your results. Think about a simple way of measuring and describing the accuracy of your program running on the robot.

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

At this point, if you have not completed questions 4 through 7, switch to doing them.

If you have completed them, you may continue on to do questions 15 and 16, which will count for 4 exploration points. You may do this **or** the "Bug Algorithm" exploration at the end of the lab (which is considerably more difficult), but may not get credit for both.

Safety

We have now constructed a useful framework for building sequential behaviors on top of a substrate that still affords frequent reading of and reaction to sensor readings. This means that, while carrying out a sequence of behaviors, the robot can react to surprises. For now, we'll show how it can change its low-level behavior in reaction to sensor values, without changing the actual sequence of high-level steps it is taking. Later in the class, we'll develop a stream-based method of sequential programming that affords a great deal of flexibility in the high-level sequencing, as well.

Question 15: Write a class `TBDriveSafely` that has the basic `TBDrive` terminating behavior as a superclass, and that overrides one of the superclass methods so that the robot will sit and wait rather than run into an obstacle. Note that it shouldn't terminate when it's blocked. Test this on the robot and describe your experiments.

More means of combination

Implement two more means of combination for terminating behaviors. Here are some that we have thought of:

- **RepeatTSM**: take a terminating state machine and a positive integer `k`. Executes that state machine until done, `k` times.
- **IfTSM**: take a condition and two terminating behaviors. Test the condition when this behavior is reset (not created!), and then execute the first behavior if the condition was true and the second if it was false.
- **WhileTSM**: take a condition and a terminating behavior. When the behavior is reset, evaluate the condition. If it's false, terminate. If not, execute the terminating behavior, then test the condition again, etc.
- **ParallelFunTSM**: take a terminating behavior and a function, and call the function on every step of the terminating behavior.

And here's another possibly interesting behavior

- **ForwardUntilBlockedTB** that is a terminating behavior that moves forward until it is blocked in front, and then terminates.

Question 16: Use all of your means of combination to make an interesting robot behavior (the robot macarena, for example). Run this on the robot and describe the results.

Concepts covered in this lab

- Encapsulation of state into objects can provide useful abstraction.
- There are many different frameworks for abstraction and combination, and it's important to choose or design one suited to your problem.
- It can be hard to get repeatable behavior from a physical device.

Homework due in your lab on March 4 or 5

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 questions 1 through 14, as well as the following thought questions.

Question 17: How reliable was your program to drive in a square? What do you think was the main contributing source to the errors? What are some strategies for reducing the error?

Question 18: We have now seen two different frameworks for making primitive robot behaviors and combining them: the constraint (set-based) and sequential (terminating behaviors) approaches. Consider a robot for operating in a household, doing chores. Give examples of situations in which each type of behavior combination would be appropriate.

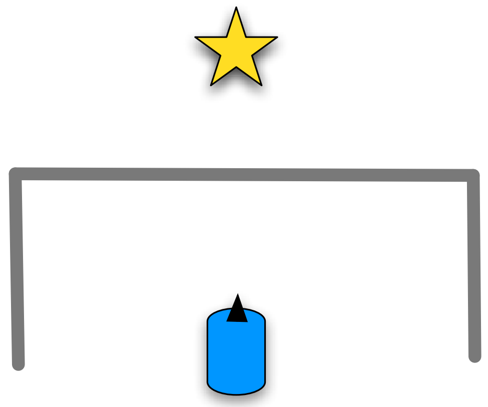

Figure 1: Robot in a cul-de-sac.

Exploration: Bug Algorithm

This is worth 8 out of 10 points for the Week 4 exploration.

You should have found that your program for moving forward is of of limited utility in the presence of obstacles. That is, that your robot could find an obstacle directly between itself and the goal that it is trying to reach. Some kinds of blockages can be escaped by simple behaviors (consider the goal as an attractive force and the sonar measurements as repulsive forces, add up all the forces and move along the resultant force), but a robot trapped in a cul-de-sac, as shown in figure 1, will have to be smarter to reach its goal.

One effective method for getting out of such jams is the "Bug 2" algorithm[3]. The idea is illustrated in figure 2. Here's the algorithm:

- Compute the line L from start to goal
- Follow that line until an obstacle is encountered
- Follow the obstacle (in either direction) until you encounter the L line again, *closer to the goal*
- Leave the obstacle and continue toward the goal

Note that in the Bug2 algorithm the robot's motion is not just a function of the sensors, it depends on the line L which was defined when the robot started. So, it requires remembering values between invocations of the `step` procedure. If you want to use a variable to store a value between invocations of the `step` procedure, you can extend your brain to something like this:

```
#code that is called once at startup
def setup():
    print "start"
    robot.x = 0

#code that is called about 10 times a second
def step():
    robot.x = robot.x + 1
    print "Step ", robot.x
```

[3]It is often better, but sometimes much worse than another algorithm called "Bug 1". For more info see `http://www.cs.jhu.edu/~hager/Teaching/cs336/Notes/Chap2-Bug-Alg.pdf`

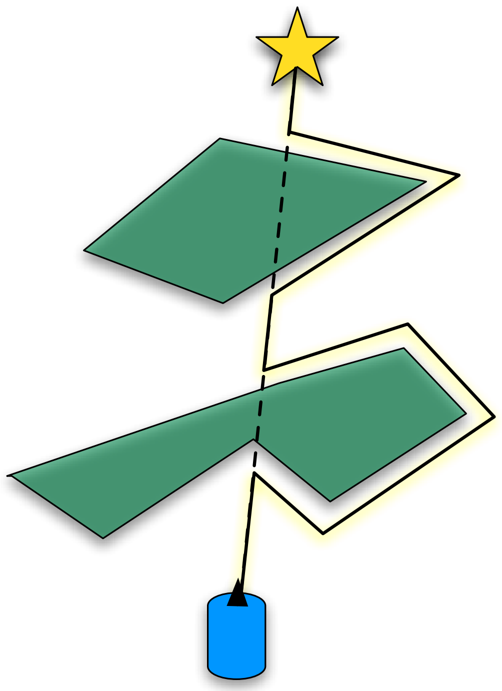

Figure 2: The Bug2 algorithm.

When implementing Bug2, you can define the line L in the `setup` procedure and then the basic behavior in the `step` procedure. In later weeks, we will develop more elaborate algorithms that depend on *state*, which encapsulates the robot's history.

A comment on representing lines. In high school, they probably taught you to represent the line between two points (x_1, y_1) and (x_2, y_2) in the form $y = mx + b$ where m is the slope $y_2 - y_1 / x_2 - x_1$ and b is the y-intercept (what's the expression for that?). You can use m and b to represent the line L, you can test whether an (x, y) position is on the line by checking whether it (nearly) satisfies[4] the equation for the line. You should note that the $y = mx + b$ is not really a very good representation for lines; it cannot represent vertical lines, since their slope is infinite. A better representation is $ax + by + c = 0$. You can stick with $y = mx + b$ for now, but you should check for the vertical case and possibly change one of the endpoints a little bit.

The bug algorithm has some state in it. Think about how you could use the terminating sequential behavior framework (or state machines, or some other systematic way of using state) to structure your algorithm.

To simplify testing and debugging in the simulator, we recommend adding this definition to your file

```
def cheatPose():
    return app.soar.output.abspose.get()
```

and using `cheatPose()` instead of `pose()` to determine where the robot is. If you do this, then when you drag the robot around the simulator window with the mouse, it will magically know the

[4]Even in the simulator you cannot expect that the position of the robot will fall exactly on the line.

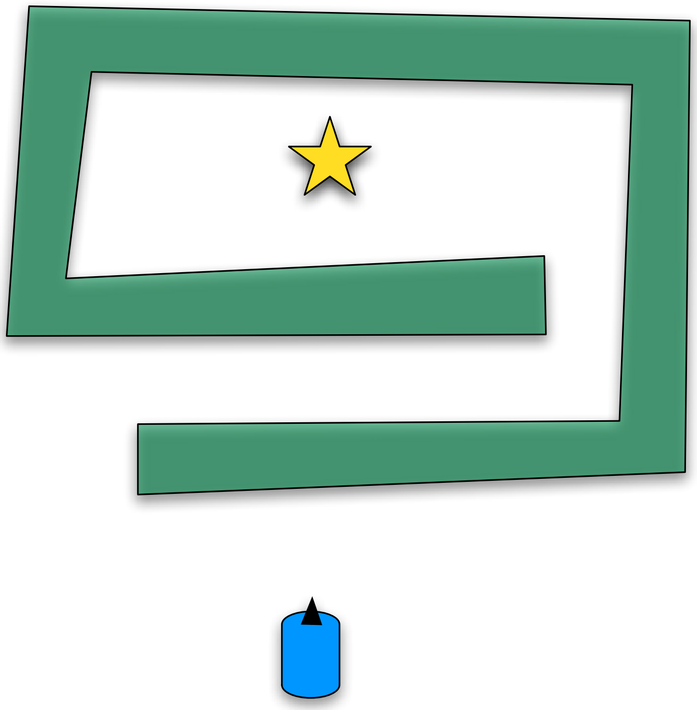

Figure 3: What would Bug2 do in this domain without the *closer to goal* test?

true pose. (On the real robot, of course, if you were to pick it up and put it down, it would have no idea that it had moved.)

Exploration 5: Implement Bug2 in the SoaR simulator. Test it on the `BugTest` worlds in the `ps4` distribution.

Exploration 6: Explain why Bug2 would be hard to implement on the real robots.

Exploration 7: If we hadn't added the *closer to the goal* requirement in the Bug 2 algorithm, what would it have done on the environment in figure 3? Try it with the robot turning to the right when it hits the obstacle, and with it turning left.

Exploration 8: Integrate your implementation of Bug2 into your program to drive in a square.