Lab 1: Charging Station
15-424/15-624/15-824 Logical Foundations of Cyber-Physical Systems
TA: Katherine Cordwell (kcordwel@cs.cmu.edu)

Betabot Due Date: Friday, September 13th **BEFORE 12:00 noon with NO late days**, worth 20 points
Veribot Due Date: Thursday, September 19th, 11:59PM (2 late days, max 6 per semester), worth 70 points

Lab Resources: `https://lfcps.org/course/lfcps19/lab1.zip`

Reminder: Betabot and Veribot submissions

Every lab assignment from Lab 1 onwards is divided into two submissions: Betabot and Veribot. The material that you need to hand in for each submission will be clearly specified in each lab assignment. Read the submission instructions carefully, and ask if you have any doubts.

Typically, Betabot questions will ask you to provide a preliminary model conjecture or answer some starter questions, while Veribot questions will ask for completed safety proofs. For example, the Betabot for Sections 1 through 3 in this assignment asks you to fill out some model templates and safety conjectures, while Section 4 asks for a preliminary implementation of the interpreter. The Veribot submission then asks you to complete the safety proofs (and the interpreter).

The purpose of the Betabot submission is for us to give you preliminary feedback on your early model designs before you attempt to verify them. Thus you should strive, as much as possible, to give thorough and complete Betabot answers so that we can give you more informative feedback.

Reminder: Syntax

In this lab you will start writing more complicated hybrid programs, safety properties, and proofs by yourself. Pay careful attention that the models you write down mean what you think they mean, and use parentheses when you are not sure. For example, $A \wedge B \rightarrow C$ parses as $(A \wedge B) \rightarrow C$, which should not be confused for the very different proposition $A \wedge (B \rightarrow C)$. For a quick reminder on operator precedence, refer to the cheatsheet in Assignment 1, Question 2.

Additionally, KeYmaera X uses ASCII syntax that is easier to parse but might look slightly different from the syntax in lectures and assignments. For a detailed summary, look up this page:
`https://github.com/LS-Lab/KeYmaeraX-release/wiki/KeYmaera-X-Syntax-and-Informal-Semantics`

For example, in KeYmaera X's ASCII syntax, curly braces are used to group statements in a hybrid program, and `++` is ASCII syntax for the choice operator $\cup$ between two programs:

```
{x := 1;
 ?(x < 2 & x>0);
} ++
x := 1;
```

1 Autobots, Roll Out

As the most junior member in your robotics group, the senior members gave you the broken robot. Its steering is jammed, so it can only move in a straight line, and you can only control its acceleration.

1. (Betabot). Fill in the missing continuous dynamics in the hybrid program below that will model your robot accelerating in a straight line, with position `pos`, velocity `vel`, and acceleration `acc`. The template is provided in L1Q1.kyx.

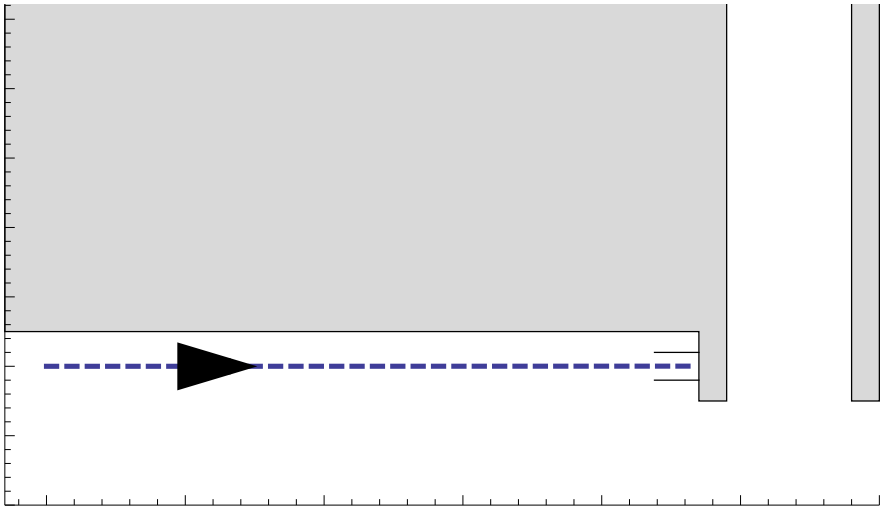

Figure 1: Charging station & wall.

2. (Betabot). Fill in a safety condition expressing that your robot never travels backwards (i.e., its velocity is never negative). Ensure that your model parses in KeYmaera X. Save this file as L1Q1.kyx.

3. (Veribot). Complete the proof with KeYmaera X. Export the resulting proof and save it as L1Q1.kyx. All of the tasks in this lab should prove automatically, but keep in mind that manual proofs can help KeYmaera X scale to more complex challenges in later labs and it helps if you start learning how to do these early!

```
(vel = 0 & acc >= 0)    /* Requires */
->
[{________}]            /* Continuous dynamics */
(________)              /* Ensures (safety condition) */
```

2 Charging Station, Single Control

Oh no! Your robot's battery is almost dead! Luckily, the robot is already on a straight trajectory to a wall charging station and has positive velocity. With its remaining power, your robot has just *one chance* to engage the brakes properly to bring the robot to a stop at the right place:

- If the robot brakes too hard, it will not make it to the charging station and will have to wait for a human to plug it in. You never know how long it takes for the next human to come by. Running out of power is *inefficient*.
- If the robot doesn't brake hard enough to stop at the charging station, it will dent the wall behind the station. When building manager Jim Skees finds out, he'll have your robot banned from the building forever! Running into walls is *unsafe*.

Hint: Think carefully about all of the variables that the acceleration/braking should depend on. If you find you can't prove the property, it could be that you missed something and your property is falsifiable. Don't forget that you learned how to find counterexamples in Lab 0!

1. (Betabot). Specify the missing safety and efficiency requirements for the hybrid program below, then fill in the missing control and continuous dynamics. Save the file as L1Q2.kyx (template provided).

2. (Veribot). Use KeYmaera X to prove that the hybrid program is safe and efficient. Export the resulting proof as L1Q2.kyx.

3. (Veribot). **Question:** What is the evolution domain for the continuous dynamics in this hybrid program? Why is it necessary? Submit your answer in lab1.txt.

```
(pos < station & vel > 0) /* Requires */
->
[
    acc := _________;      /* Assign a safe acceleration. */
    {____ & vel >= 0}      /* Use continuous dynamics from previous part. */
]
(____________              /* Ensures (safety condition) */
 &____________)            /* Ensures (efficiency condition) */
```

3 Charging Station, Double Control

In Part 2, you are always able to coast the robot all the way to the charging station, since it is already moving. But what if the robot is stopped? In this question, the goals are the same as in Part 2; however, the robot starts with zero velocity. You have **two** chances to control your robot, first to get it moving by accelerating, and then to bring it to a stop at the right point by braking.

The robot also has a time limit `T` on how long it can accelerate before exhausting the remaining battery life. Once the brakes are engaged, they will stay engaged until the robot comes to a complete stop.

1. (Betabot). Fill in the formula below and save the file as L1Q3.kyx (template provided).

2. (Veribot). Prove the formula and export the proof as L1Q3.kyx.

3. (Veribot). **Question:** What is your efficiency condition? Is it different from Part 2, why or why not? Submit your answer in lab1.txt

```
(pos < station & vel = 0 & 0 < T)       /* Requires */
->
[
    t := 0;
    acc := _________;                   /* Assign a safe acceleration. */
    {____, t' = 1 & vel >= 0 & t <= T};
    ?(t > 0);
    acc := _________;                   /* Assign a safe deceleration. */
    {____, t' = 1 & vel >= 0}
]
(____________                           /* Ensures (safety condition) */
 &____________)                         /* Ensures (efficiency condition)*/
```

4 Implementing an Interpreter

To get some practice with semantics, you are going to program the semantics of first-order logic! That is, for terms you will write a function that computes their value in a given state. For formulas (which might have quantifiers), you will write a function that computes whether they are true in a given state. We shall use the same definition of first-order logic as given in Definitions 2.2 and 2.3 of Chapter 2 of the textbook, and also in the lecture slides: `http://lfcps.org/course/lfcps19/02-diffeq-slides.pdf`

The syntax of terms is given by the grammar (where x is a variable and c is a rational constant):

$$e, \widetilde{e} ::= x \mid c \mid e + \widetilde{e} \mid e \cdot \widetilde{e}$$

The syntax of formulas is given by the grammar:

$$P, Q ::= e \geq \widetilde{e} \mid e = \widetilde{e} \mid \neg P \mid P \wedge Q \mid P \vee Q \mid P \to Q \mid P \leftrightarrow Q \mid \forall x\, P \mid \exists x\, P$$

A few parts of this are tricky and hard to do exactly in a computer program. Most real numbers require infinite storage space, making them hard to implement on a computer. Instead you should use an approximation of the real numbers, such as floats or rationals. Floats will most likely be easiest. Quantifiers are also problematic because there are so many reals we could never try them all. When instantiating quantifiers, pretend the only real numbers are $0, 0.5, 1, \ldots, 10$. [1]

Your interpreter also needs to manage the *state* telling you which variables have which values. This is abstracted out into a state data structure which maps variables to their current values. This structure is provided and should have all the operations you need for the assignment, but you will need to use it correctly.

Because you might not all know the same programming languages, we have provided starter code in 3 languages: Standard ML, Java, and C++. The size of each file is indicative: if you know Standard ML, you should probably use that because functional languages make it very easy to write an interpreter.

1. Implement the two `interpret` functions, one for terms and one for formulas. Submit one file: src/L1Q4.sml, src/L1Q4.java, or src/L1Q4.cpp

   For Betabot, submit your progress so far (which should include, at the very least, writing a few tests). For Veribot, submit the completed interpreter. For Veribot, you will receive all of the testing points if your tests correctly cover each operator at least once.

2. (Veribot). **Question:** Write a formula and a state for which your interpreter says it's true, but it should be false. Why is that? Write your answer in lab1.txt

3. (Veribot). **Question:** Write a formula and a state for which your interpreter says it's false, but it should be true. Why is that? Write your answer in lab1.txt

5 Interacting with KeYmaera X

This section is mainly for you to gain familiarity with KeYmaera X. *You are not required to submit anything for this section.*

Recall the formula $\forall x(x > 0 \wedge x < 1)$ from Lab 0. You will manually interact with this formula now, by trying to prove it. Since the formula is false you will not be able to, but you will see how doing so affects the proof tree.

1. Start a proof and then highlight the entire formula by hovering over the $\forall$ symbol and **right**-click. You will see a list of applicable proof rules. Use the "$\forall$R" rule by clicking on the name of the rule in the top-left portion of the box describing the rule (see Figure 2).

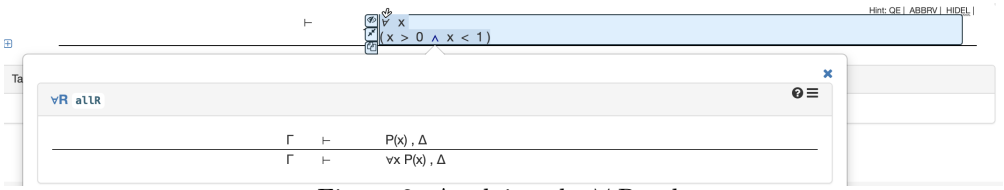

Figure 2: Applying the $\forall$ R rule.

[1] Writing this interpreter with exact reals and quantifiers is the same as implementing *quantifier elimination*, an important tool used to solve arithmetic questions generated by KeYmaera X.

2. This gets rid of the $\forall$ symbol, leaving you only with the conjunction $x > 0 \land x < 1$. Click on the $\boxed{+}$ to the left of the proof tree. Notice how the previous formula moved below a horizontal line and you now have a new formula that was produced by applying the $\forall$R rule (Figure 3). The proof tree keeps track of everything you have done to the original formula.

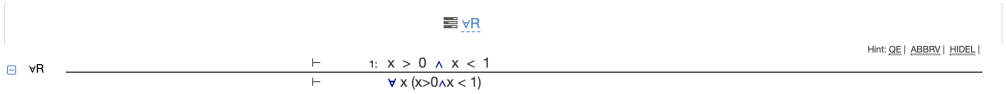

Figure 3: Expanding the proof tree.

3. To prove a conjunction, both conjuncts need to be true. In this case, you would need to prove both $x > 0$ as well as $x < 1$. As before, hover your mouse over the $\land$ symbol, right-click, and select "$\land$R". This rule applies the reasoning we just made, where to prove $x > 0 \land x < 1$, you must prove each conjunct separately.

Figure 4: Applying the $\land$ R rule results in two tabs.

Look at the tabs (Figure 4)! Two new subgoals were added, one for each conjunct. The proof tree now actually has branches rather than just being a sequence of nodes. Now click on the $\boxed{+}$ button on the left side of the proof tree to see the sequence of steps that produced each subgoal (see Figure 5).

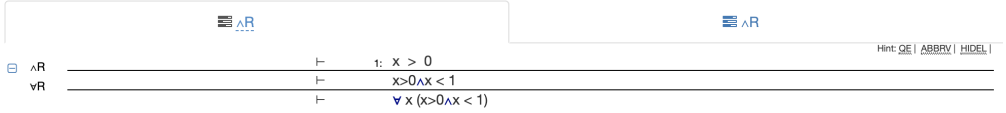

Figure 5: The expanded proof steps.

4. You can also re-label proof tree nodes by clicking on their names (such as "Goal: $\land$R (13,0)" in the figures above). This can be helpful in large proofs (just like giving your variables descriptive names like "lander_acceleration" instead of "la11" is useful when programming in the large). Try changing the node labels yourself (see Figure 6).

Figure 6: Naming open proof goals.

5. Since you haven't proven either of the cases, they are called *open goals*. Select each open goal, and notice how KeYmaera X updates your screen to show the selected goal. Each goal should represent one of the conjuncts. In a bigger proof, you would now continue to try to prove each conjunct separately, growing that part of the tree.

6 Submission Checklist

Use the provided templates, and *do not forget to fill in the section at the top.* It gives us important information when grading your submission!

1. **Initial submission (Betabot)**. Submit a .zip file on Autolab containing your preliminary .kyx files for each of the tasks and your preliminary interpreter. This will enable us to give you feedback halfway through the assignment, so that you don't get stuck! If you want, you can include some *small* comments about your approach and questions you might have.

   The Betabot zip file should contain:

   - `L1Q1.kyx`
   - `L1Q2.kyx`
   - `L1Q3.kyx`
   - `src/L1Q4.sml` or `src/L1Q4.java` or `src/L1Q4.cpp`

2. **Final submission (Veribot)**. The final submission works the same way, but now your .kyx files should include both the model and your proof tactic. To receive full credit, proofs must be complete (i.e., a successful "Proof Result" window appears after running the tactic you have submitted).

   The Veribot zip file should contain:

   - `L1Q1.kyx`
   - `L1Q2.kyx`
   - `L1Q3.kyx`
   - `src/L1Q4.sml` or `src/L1Q4.java` or `src/L1Q4.cpp`
   - `lab1.txt`