Week 10

Collaboration: You should work on the problems yourself, before discussing with others, including your cohorts at your cohort meeting. By the Assessed Cohort Meeting, you and all of your cohortmates, should be prepared to present and discuss solutions to all of the assigned problems. In addition to discussing with your cohortmates, you may discuss the problems with any other current CS3102 students you want, and use any resources you want except for any materials from previous offerings of this course or complete solutions that might be available on the web, which are not permitted. **At the end of your assessed cohort meeting, your Cohort Coach will assign one of these problems as a writeup. You may not collaborate on this writeup with your cohort-mates, but you may use notes taken before or during your assessed cohort meeting.**

Problem 1: Laptop vs. Turing Machine

Are there any things that your laptop can do that a Turing Machine cannot? Are there any things that a Turing Machine can do that your laptop cannot? What if we consider things that they can/cannot "do" as things other than functions/languages?

Problem 2: Self Reject

In the *Self-Rejection (An Uncomputable Function)* video, we gave a specific uncomputable function. Below, we begin a similar proof, but in the context of Python code. Help us complete this proof below of a function not computable by Python. (Note: you will find Section 9.3.2 of the TCS book helpful for this also.)

Definition 1 (`self_rejecting_py`) The Python function `self_rejecting_py(w)` should behave as follows for input string `w`:

1. if `w` is anything other than syntactically valid Python source code that defines a function which takes a single input parameter, return `True`.
2. Otherwise (meaning `w` is Python code for a function that takes a single string as input), then return the negation of the output that invoking the program `w` on the input string `w` returns.

(Note that "negation" means what we expect (`NOT`) if the output is a Python Boolean, but is also defined for other outputs, which Python can interpret as Boolean values. It is fine to ignore these typing issues and assume you only need to deal with normal Boolean values.)

With this in mind, we might make the following attempt at implementing `self_rejecting_py`. This function will take the source code, check that it is a python function with one input parameter (using `one_input(w)`), then modifies the source code to execute the function on itself and save the answer to a file (using `add_self_invoke`), runs the modified code (using `exec`, which is essentially Python's universal Turing Machine), then answers the opposite of the file's contents. (Note that all subroutines mentioned for this function can be computed, see us in office hours to discuss how.)

```
    def self_rejecting_py(w):
        if not one_input(w):
            return True
        modified_w = add_self_invoke(w)
        exec(modified_w)
        return not read_result()
```

So for example, running `self_rejecting_py` on the string:

```
def f(m):
  return len(m) % 2 == 0
```

would generate and run the modified code:

```
def f(m):
  return len(m) % 2 == 0
x = '''
def f(m):
  return len(m) % 2 == 0
'''
if f(x):
  print("True", file=open('outputfile'))
else:
  print("False", file=open('outputfile'))
```

What happens if we run the `self_rejecting_py` program on its own source? Show that this function cannot be implemented as described.

Problem 3: ACCEPTS

Review the *ACCEPTS, Practical but Uncomputable* video.

Be able to answer questions about this proof such as: *What is the ACCEPTS language?, What is the input to the machine M_A?, Can you determine whether or not some machines accept on empty input without running them?, If we had a procedure that implements ACCEPTS, what would it be useful for?.*

Problem 4: ACCEPTS in k Steps

Consider the Language:

$$A_k = \{M|M \text{ is the description of a no-input Turing Machine which accepts in } k \text{ or fewer steps }\}.$$

Show that A_k is computable for every choice of $k \in \mathbb{N}$.

Problem 5: A Lot of Unions

If two languages L_1 and L_2 are computable, then $L_1 \cup L_2$ is also computable. If you don't understand why this is the case, then first convince yourself.

You should be able to explain why this means that the union of finitely many computable languages will be computable. Next, you will prove that this is not the case for an infinite union using the A_* language below.

Definition 2 (A_*) *Define the language* $A_* = \bigcup_{k\in\mathbb{N}} A_k$.

Show that, even though it is constructed by unioning only computable languages, A_* is not computable. (Hint: you should find it to be very similar to an uncomputable language you have seen.)

Problem 6: A Flawed Proof

The "proof" below seems to prove that A_* is computable. Identify and explain a flaw in the proof.

Consider the following bogus "proof" by induction that A_* is computable:

Our inductive hypothesis is:

$$P(n) := A_0 \cup A_1 \cup ... \cup A_n \text{ is computable.}$$

Base case: A_0 is computable.

Proof: We showed (in your correct answer to Problem 4 above, which we will assume exists) that $\forall k \in \mathbb{N}$, A_k is computable.

Inductive case: We show $P(n) \implies P(n+1)$. That is, if $A_0 \cup A_1 \cup ... \cup A_n$ is computable, so is $A_0 \cup A_1 \cup ... \cup A_n \cup A_{n+1}$.

Proof: By the inductive hypothesis, $A_0 \cup A_1 \cup ... \cup A_n$ is computable.

By your answer above A_{n+1} is computable.

The union of two computable languages is computable.

Thus, it must be that $A_0 \cup A_1 \cup ... \cup A_n \cup A_{n+1}$ is computable as well.

So, by induction, we have shown that $P(n)$ holds for all $\mathbb{N}$, and A_* is computable.