PS9: Uncomputability

Collaboration: You should work on the problems yourself, before discussing with others, including your cohorts at your cohort meeting. 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. Sharing and subsequently submitting any text, code, images, figures, etc. constitutes plagiarism, so make sure all submitted materials are created exclusively by members of your cohort.

Problem 1: Self Reject

In lecture, 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 may 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_outputfile()
```

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"))
```

Explain what happens if we run the above (failed) attempt of implementing `self_rejecting_py` on its own source. Next, prove that `self_rejecting_py` cannot be implemented as defined.

Problem 2: Entranced By Re-Entrances

Use a proof by reduction to show that the function $REENTERS$ defined below is uncomputable.

> **Input:** A string w that describes a Turing Machine.
>
> **Output:** **1** if the machine described by w would re-enter its start state when executed on the input ε. Otherwise, **0**.

That is, a machine which computes $REENTERS$ outputs **1** when the input describes a Turing Machine which, when run with the input ε, enters the start state as a result of some transition.

(Note: For this problem, we want to see that you understand how to do a reduction proof, so even if you can prove it using some other method, you should also be able to explain how to prove it using the proof by reduction method.)

Problem 3: Maximum Recursion Depth

Python has a default recursion depth limit of 1000 calls.[1] This means that if ever we have a function which calls itself over 1000 times (nested), Python will throw a `RecursionError` exception.

For example, calling the `rec(x)` function defined below with input `x = 500` would terminate "happily", but on input `x = 1001` would result in a `RecursionError` exception.

```
def rec(x):
   if x > 0:
      return rec(x-1)
   else:
      return "Done"
```

Using a reduction proof, show that, in general, the problem of determining whether a given Python program will exceed the maximum recursion depth is not computable. That is, show that there does not exist an always-halting Turing Machine which, when given a python program, returns 0 if the program does not throw a `RecursionError` exception, and will return 1 if it does. For example, if we followed the above code with the line `rec(1001)` then the Turing machine should return 1 when given that program as input. If we followed the above code with the line `rec(500)` then the Turing machine should return 0 when given that program as input.

For this problem, you should assume an "idealized" version of Python with no other implementation limits, other than the setting for recursion depth.

[1] This has lead many dysfunctional programmers to conclude that "recursion is bad" and should be avoided, even though recursive definitions are often the most elegant and clear way to express many functions. The real problem is that most versions of Python have implementations of recursive function calls that do not do tail call optimizations that are done by more functional programming languages (this means that a recursive function execution requires $\Theta(n)$ stack space in Python where n is the number of recursive calls, whereas a "correct" implementation would only use $O(1)$ stack space). This has lead to the recursively bad situation where Python programmers are trained to think recursion is bad, so Python language implementers don't think it is important to make tail recursive functions perform well, which means Python programmers continue to correctly learn that in Python recursive function definitions are "bad".

Problem 4: Can we avoid the Halting Problem?

We showed in lecture that Turing machines suffer from a "Halting Problem". I.e., it is impossible to design a Turing Machine which can determine whether the Turing machine described by a given string will halt when running on a particular input string. Since the idealized python computing model is equivalent to the Turing machine model, python must have this "Halting Problem" as well (i.e. it is impossible to write a python program that can tell whether an arbitrary python program will halt on some input). One might wonder, however, if the presence of a "Halting Problem" for a programming language is absolutely necessary, or if it is only ever an avoidable flaw in a language's design. Consider these restrictions we could add to python so that infinite loops are always detectable (we consider programs which throw errors as halting):

– All `while` loops must be `while(True)` loops

– All other loops must be `for` loops

– `break` and `return` cannot appear in the body of a `while(True)` loop

– All recursive calls (even indirect) result in a runtime error

The idea here is that we add rules to Python so that a program will run forever if and only if it has a `while(True)` loop[2]. Importantly, programs written in our modified python *can* run forever, but it will be easy for us to tell by just looking for `while(True)`. We would, next, want to consider whether this restricted python language was weaker than the original python. Towards this end, answer the following questions:

a) Consider Problem 1 above. If we were to implement `self_rejecting_py` using our restricted Python, what happens when we try to invoke `self_rejecting_py` on its own source?

b) Next show that any programming language which does not have a "Halting Problem" is a weaker computing model than a Turing Machine. To do this, we'll say that the function H_ℓ represents the "Halts Function" for programming language ℓ (e.g. our restricted Python), i.e. $H_\ell(p, x) = 1$ if program p written in language ℓ halts on input x, and $H_\ell(p, x) = 0$ otherwise. Show that if there exists a Turing Machine which solves H_ℓ then programming language ℓ cannot be used to implement a Universal Turing Machine. (*Hint:* We recommend you use a proof by contradiction.)

[2] I have not formally shown that there are no other ways to get infinite behavior with Python, but if there are, suppose we have additional rules above to disallow those as well.

