COMP 314 Class 2: Proving that some programs are impossible

Our objective for today is to prove that there are some programs which are impossible to write. Our first example will be the program `alwaysYes.py`. This program takes another program as input, and determines whether or not the input always decides Yes. As we'll soon see, it is provably impossible to write `alwaysYes.py`. Figure 1 summarizes the argument, which is described in more detail in class.

Important exercise: be able to describe the output of each of the programs in Figure 1, when given various programs as input (including themselves).

Program name	**Program behavior**
`alwaysYes.py`	– decide Yes, if input always decides Yes – decide No, otherwise (e.g. input can produce No, or crash, or enter an infinite loop)
↓	
`yesOnSelf.py`	– decide Yes, if input decides Yes when given itself as input – decide No, otherwise
↓	
`antiYesOnSelf.py`	– decide No, if input decides Yes when given itself as input – decide Yes, otherwise

Figure 1: **A sequence of three decision programs that cannot exist.** The last program, `antiYesOnSelf.py`, is obviously impossible, since it produces a contradiction when run on itself. However, each of the programs can be produced easily by a small change to the one above it (shown by the arrows). Therefore, none of the three programs can exist.

The impossibility of writing the program `alwaysYes.py` is interesting, but perhaps it's not immediately clear why this program might be useful. In contrast, our next example would be incredibly useful if it were actually possible: we will discuss the program `canCrash.py`, and prove that it too cannot exist. `canCrash.py` takes another program as input, and decides whether that input can crash. (For our purposes, the word "crash" here means the same thing as "throws an exception.") Figure 2 summarizes the argument proving that `canCrash.py` is impossible.

Program name	**Program behavior**
`canCrash.py`	– output Yes, if input can crash – output No, if input never crashes
↓	
`canCrashWeird.py`	– Crash, if input can crash – output No, if input never crashes
↓	
`crashOnSelf.py`	– Crash, if input crashes when run on itself – output No, if input doesn't crash when run on itself
↓	
`antiCrashOnSelf.py`	– output Yes, if input crashes when run on itself – Crash, if input doesn't crash when run on itself

Figure 2: **A sequence of four crash-detecting programs that cannot exist.** The last program, `antiCrashOnSelf.py`, is obviously impossible, since it produces a contradiction when run on itself. However, each of the programs can be produced easily by a small change to the one above it (shown by the arrows). Therefore, none of the four programs can exist.

Our third example is related to (but not quite the same as) the famous *halting problem*. We can show that the program `alwaysHalts.py`, which determines whether or not its input alway halts, is impossible. Figure 3 summarizes the argument, which should now look familiar.

Program name	**Program behavior**
`alwaysHalts.py`	– decide Yes, if input always halts – decide No, otherwise (i.e. input can enter an infinite loop)
↓	
`alwaysHaltsWeird.py`	– decide Yes, if input always halts – enter infinite loop, otherwise
↓	
`haltsOnSelf.py`	– decide Yes, if input halts when given itself as input – enter infinite loop, otherwise
↓	
`antiHaltsOnSelf.py`	– enter infinite loop, if input halts when given itself as input – decide No, otherwise

Figure 3: **A sequence of four halting-detection programs that cannot exist.** The last program, `antiHaltsOnSelf.py`, is obviously impossible, since it produces a contradiction when run on itself. However, each of the programs can be produced easily by a small change to the one above it (shown by the arrows). Therefore, none of the four programs can exist.

A more formal approach

Let's now discuss these issues using a more conventional, mathematical approach. Formally, "impossible programs" are called *undecidable problems*. For our purposes, a *problem* is just a mathematical function, mapping ASCII strings to other ASCII strings. For example, the problem MULTIPLY takes as input strings like "`2*45`" (and the correct output in this case would be "`90`"). A *decision problem* is a problem whose output is always "`yes`" or "`no`". For example, the problem ISPRIME takes as input a string like "`18`" (and the correct output in this case would be "`no`").

Obviously, it's possible to write Python programs that implement both MULTIPLY and ISPRIME. But we proved above that some problems can't be solved with a Python program (and we'll find out next week that the restriction to Python is irrelevant—these problems cannot be solved with any other known programming language or computatational device, either). A decision problem that cannot be solved by a computer program is called an *undecidable problem*.

Specifically, we showed above that the problems ALWAYSYES, CANCRASH, and ALWAYSHALTS are undecidable. In Topic 3 of the course, we will find out that a vast array of other problems are undecidable.

A more conventional proof of the undecidability of the halting problem

Let's finish our discussion with a more conventional treatment of the famous halting problem. The most usual definition of the halting problem asks whether a given program halts on a given input. For our purposes, then, the problem HALTS takes as input a single string that is the concatenation of (i) some Python program P, and (ii) an input string I. The output is "`yes`" if P halts on input I, and "`no`" otherwise. We write the input string as $P + I$.

We claim that HALTS is undecidable. Suppose not, and argue for a contradiction. By our assumption, we can write a Python program H which outputs "yes" on input $P + I$ if P is a Python program that halts on input I (and otherwise H outputs "no"). We can make a small change to H, so that instead of outputting "yes", it enters an infinite loop. Call this new program W (for Weird). Note that, if P halts on input I, then W loops on input $P + I$. (And if P doesn't halt, W does halt—and outputs "`no`".) Now make a small change to W, producing a new program S (for Self). The program S takes as input just a Python program P, then invokes W with input $P + P$. So, S loops forever if P halts when given itself as input. And S halts (outputting "`no`") if P doesn't halt when given itself as input. Consider what happens when S is given the input S. Either it halts, or it doesn't. But each possibility produces a contradiction: if S halts on input S, its definition says it loops forever; if S doesn't halt on input S, its definition says it halts. Hence, we have produced a contradiction, and our claim is proved.