COMP 314 Class 6: Rice's Theorem, and many more undecidable problems

1 Programs

From now on, we will use the word *program* to mean any kind of computer program, including Python programs and Turing machines.

2 Problems again

Recall that in class 2, we defined a *problem* to be a function that maps ASCII strings to ASCII strings, and a *decision problem* to be a problem whose output is always "`yes`" or "`no`". Given a decision problem, we say a program *decides* the problem if it always terminates with the correct output ("`yes`" or "`no`").

More precisely, denote the decision problem by d, let I be an ASCII string input for d, and write the output for this problem as $d(I)$. Let P be a program, and write $P(I)$ for the output of P on input I. Then P decides d if $P(I) = d(I)$ for all I.

A decision problem is *decidable* if there exists a program that decides it; otherwise, the problem is *undecidable*. Figure 1 shows three examples of Python programs that decide problems. From these examples, it is clear that the problems IsBinaryString, IsPrime, and Has3Gs are decidable. On the other hand, we have seen in previous classes that various problems are undecidable, including HaltsOnInput, HaltsOnEmpty, HaltsOnSome, HaltsOnAll, YesOnInput and AlwaysYes.

Of course, many interesting problems are not decision problems, but we would still like to know whether we can solve these problems using a computer. Given a problem, we say a program *computes* or *solves* the problem if it always terminates with the correct output. The formal definition is exactly as for decision problems, but we will denote the problem by f (for $f$unction) instead of d (for $d$ecision). Note that we will reserve the letter P for Program; we will never use p or P for $p$roblem. Formally then, a program P computes a problem f if $P(I) = d(I)$ for all ASCII input strings I.

There are no surprises with the definitions of computable and uncomputable: a problem is *computable* if there exists a program that computes it; otherwise, the problem is *uncomputable*.

It's important to realize that $P(I)$ can be the special value `Loop`, which means that P doesn't terminate on input I—instead, it goes into an infinite loop or some other kind of never-ending computation. However, if P computes or decides some problem, then this behavior is not permitted: P is required to terminate for every input I.

```
# Python program binaryStrings.py
isBinary = True
for c in input:
    if c != '0' and c != '1':
        isBinary = False
if isBinary:
    print "yes"
else:
    print "no"
```

```
# Python program isPrime.py
inputNum = int(input)
isPrimeNum = True
for i in range(2,inputNum): # inefficient, but it works
    if inputNum % i == 0:
        isPrimeNum = False
if isPrimeNum:
    print "yes"
else:
    print "no"
```

```
# Python program has3Gs.py
if input.count("g") >= 3:
    print "yes"
else:
    print "no"
```

Figure 1: **Three examples of Python programs that decide problems.** Top: Decides whether the input is a binary string. Middle: Decides whether the input is a string representing a prime number. Bottom: Decides whether the input contains 3 or more `g`'s.

3 Semi-deciding

Undecidable problems are bad news for computers, but it turns out that some are not quite as bad as others: these are the semi-decidable problems. Informally, a decision problem is semi-decidable if you can compute all of the "`yes`" answers correctly, but not necessarily all of the "`no`" answers.

Formally, a program P *semi-decides* a decision problem d if:

- $P(I)$ = "`yes`", for all I with $d(I)$ = "`yes`"; and
- $P(I)$ = "`no`" or $P(I)$ = `Loop`, for all I with $d(I)$ = "`no`".

The key point is that $P(I)$ always give the right answer if it terminates, but on "`no`" instances, P is allowed to run forever.

Unsurprisingly, a decision problem is *semi-decidable* if some program semi-decides it.

Exercise. By definition, all decidable problems are also semi-decidable. But which of the undecidable problems listed above are semi-decidable?

Solution (Try to come up with this on your own): HaltsOnInput, HaltsOnEmpty, YesOnInput. (In contrast, it can be shown that HaltsOnAll and AlwaysYes are not even semi-decidable—but we won't prove it in this course. Also, it's worth noting that HaltsOnSome is, rather surprisingly, semi-decidable. but again, we won't prove it here.) It's easy to write Python programs that semi-decide each of these problems. For example, Figure 2 shows how to do this for HaltsOnInput: you just execute the given program with the given input (this is easily done using our universal program for executing other programs). If that terminates, we print "`yes`"; otherwise, our program runs forever, but that doesn't matter since it isn't required to terminate on "`no`" instances.

```
# Python program HaltsOnInput.py
from universalFunction import universal
universal(input)
print "yes"
```

Figure 2: The Python program `haltsOnInput.py`, which semi-decides the problem HaltsOnInput.

4 Our version of Rice's theorem

4.1 Problems about problems

Just as we can write programs that analyze other programs, we can consider problems about other problems. Consider, for example, the relatively simple (and certainly decidable)

problem IsPrime, which asks whether a string represents a prime number. Now suppose you are given a program, and asked to determine whether that program decides IsPrime. This is, in itself, a problem that we call DecidesIsPrime. The input is a program P, and the output is "`yes`" if P decides IsPrime, and "`no`" otherwise.

We can repeat this process with any decision problem d. The particular case of d being IsPrime yields DecidesIsPrime; a general decision problem d yields a new problem that we write as Decides(d). So, the input to Decides(d) is a program P, and the output is "`yes`" if P decides d, and "`no`" otherwise.

We can define the same kind of "problems about problems" for non-decision problems f: the problem Computes(f) asks whether the input P computes the problem f.

As we will see in the next section, these "problems about problems" are almost always undecidable.

4.2 Finally, Rice's theorem

Rice's theorem is usually stated in a fairly general setting, about sets of semi-decidable problems. But it is still very useful, and easier to understand, if we consider the following simpler variant.

Our simplified version of Rice's Theorem: Let d be a decidable problem. Then Decides(d) is undecidable.

Proof: The proof will be sketched in class, but will not be part of any homework or exam. As you might expect, it involves a reductin from the halting problem.

Examples: Many of the undecidable problems we have seen earlier in the course can be proved undecidable by a simple application of Rice's theorem. This includes:

- YesOnRedDevils: take d to be the problem of determining whether the input I is "`Red Devils`".
- AlwaysYes: take d to be the trivial problem whose answer is "`yes`" for all inputs.
- YesOnEvenLength: take d to be the problem of determining whether the input I is of even length.

4.3 Stronger versions of Rice's theorem

Rice's theorem still holds when d is:

- a semi-decidable problem. [Example, using ideas that haven't been covered yet: Take d to be determining whether a string is in the language generated by a given grammar. Then Rice's theorem tells us that, given a program, it's undecidable whether the program decides membership in the language.]

- a set of semi-decidable problems (with a minor technical caveat). [Examples, using ideas that haven't been covered yet: Determining whether a program accepts a regular language, or a context-free language – we will come back to these ideas later in the course.]
- a computable problem. Example: determining whether a given program adds its inputs.

4.4 But not everything is undecidable

Rice's theorem gives us a huge new supply of undecidable problems about programs. You might be tempted to think, at this point, that any problem that takes a program as input is undecidable. However, this is not true. Here are three examples of decidable problems that take programs as inputs:

- Given a Python program, determine whether the Python program contains more than 50 lines.
- Given a Python program, determine whether the Python program contains any variables called `numWombats`.
- Given a Turing machine, determine the number of halting states in the machine.

Exercise. Prove, at least informally, that each of these problems is decidable.

5 Undecidable problems aren't always about programs

A vast array of problems are undecidable. Many of them appear to have nothing to do with computers or computer programs. See the Wikipedia page "List of undecidable problems" for a good selection.