UNCOMPUTABILITY AND SELF-REFERENCING IN CPSC 421

JOEL FRIEDMAN

Contents

Disclaimer: The material may sketchy and/or contain errors, which I will elaborate upon and/or correct in class. For those not in CPSC 421/501: use this material at your own risk. . .

Research supported in part by an NSERC grant.

The most subtle questions in any introductory computation theory course—such as CPSC 421—is to determine which problems cannot be solved in various models of computation. An example of this is "P versus NP," the most famous (and fundamental) open problem in computer science, and worth $1,000,000.

Another well known example is the fact that the Halting Problem is *undecidable.* This proof typically takes a few lines once the proper terminology and definitions are given, although this proof is quite subtle. This proof is fairly "universal," in that it holds regardless of which classical notion of computation one uses (e.g., a Turning machine, a Random-Access Machine, a C++ program).

We shall begin CPSC 421 by describing the general ideas used to prove various uncomputability results, which are for a very general notion of computation. We shall then formally describe Turing machine computations, and revisit the principles described in this article. We will cover regular languages and context-free languages at the end of the course, as these topics are of much less interest to present day mainstream computer science.

A secondary goal of this article is to discuss how various "paradoxes" (real or apparent) in mathematics relate to the undecidability of the Halting problem.

Unfortunately, this article may well "ruin the surprises" in CPSC 421. Perhaps the most interesting surprises are:

(1) "self-referencing" leads to wonderful "paradoxes," which can give interesting theorems (e.g., the Halting Problem is undecidable);
(2) certain discrete math problems can "simulate" computers, so that if you could solve these problems you could solve any problem solvable by such computers (NP-completeness).

For the first few weeks of CPSC 421/501, we will study computability in an abstract setting, and describe why we usually get problems that are not "computable" in some sense. We will then try to make this precise but concrete, and we will see why Turning machines are popular for doing this. We shall also see many related problems and "paradoxes" in other fields. Sipser's text (or almost any introductory theory text) will explain the Halting Problem in more detail, and we shall later return to the Halting Problem and cover it rigorously.

1. Some "Paradoxes"

The two main results of this course are perhaps (1) the unsolvabilitiy of the halting problem, and (2) NP-completeness. Both results are linked with a number of other remarkable results in logic and computing, and appear as paradoxes:

(1) I am lying.
(2) This statement is a lie.
(3) This sentence refers to the smallest positive integer not defined by a sentence of thirty words or fewer.
(4) This statement has no proof that it is correct.
(5) I write about those who don't write about themselves.
(6) This is the set of sets that do not contain themselves.
(7) This is a C program that takes a string, s, as input and (i) loops infinitely if s is a valid C program that halts on input s, and (ii) halts otherwise.

One thing that these statements have in common is that they all either explicity "refer to themselves" or can be "applied to themselves." Another is that they involve fundamental ideas in logic or computing. Another is that on some naive level they lead to a "paradox."

Consider the first statement, "I am lying," famously used, of course, by Captain Kirk and Mr. Spock[1] to destroy the leader of an army of robots. This leads to a paradox: if the speaker is telling the truth, then he is lying ("I am lying"), but if he is lying, then he is lying that he is lying, meaning he is telling the truth. Either way we get a contradition.

All the other statements lead to "paradoxes" (of somewhat different types); this will be discussed in class and the exercises.

2. Dealing with Paradoxes

There are a number of approaches to dealing with paradoxes. They include:

(1) Ignore the paradox. Carry on regardless.
(2) Admit the paradox, but claim it doesn't matter in practice.
(3) State the paradox in precise terms.

In this course we take approach (3), which can lead to a number of results, such as:

(1) The paradox goes away when things are stated precisely.
(2) The paradox doesn't go away, and you have to change your theory if you want it free of contradictions.
(3) The paradox doesn't go away, but only if certain things are true. Therefore (assuming your setting is free of contradictions) you have a theorem that one of these certain things must be false.

As examples, the "smallest number" paradox gives result (1), i.e., the paradox goes away when things are stated precisely; the set theory paradox gives result (2), i.e., it does not go away, and set theory had to be rewritten so as not to allow the formation of concepts like, "the set of all sets that blah blah blah." The halting problem is an example of result (3), and gives us a theorem (!) about a problem that cannot be solved by an algorithm or computer.

In order to show the Halting Problem is undecidable, we need to make our setting precise.

3. Counting, The Pigeon Hole Principle, and Computability

A simple way to show that there exist problems that cannot be solved an algorithm—in computational complexity theory—is to show that there are more "problem instances" to be solved than there are algorithms. This idea—in a vastly more general sense—is often called the *pigeon hole* principle. This principle is easy to understand when it involves finite sets; the point is that one can make sense of this principle even when the sets are infinite.

In practice, most notions of a computer program (in C, in Java, Turing machines, RAM machines, etc.) describe each individual program by information which can be viewed as a finite string over some finite alphabet. This implies that there are only *countably* many computer programs. Since usually the programs recognize *languages* over some fixed alphabet, this implies that set of languages is *uncountable*. This implies that—in most notions of computer programs—there are languages which are not recognized by any computer program.

[1]Thanks to Benjamin Israel for pointing out an earlier inaccuracy.

3.1. **The Finite Pigeon Hole Principle.** The *pigeon hole* (or *drawers*) principle is the simple observation that if there are more pigeons than pigeon holes, then some pigeon hole has more than one pigeon. Here are some simple special cases:

(1) In any set of at least 8 people, at least two people were born on the same day of the week.
(2) In any set of at least 367 people, at least two people have the same birthday (in the Gregorian calendar).
(3) In any set of at least 1001 people, at least two people have the same last three digits in their phone numbers.
(4) If you are trying to sort 10 names in alphabetical order using comparisons (i.e., you are allowed to take any two names and ask which come first in alphabetical order), then you need at least 24 comparisons (since 2^{23} is strictly less than 10!).

3.2. **Finite Programs and Languages.** Imagine a computational problem which is supposed to determine some *property* of the numbers from one to ten. Formally, a *property* of $\{1,\ldots,10\}$ is just a subset of $\{1,\ldots,10\}$ such as

$$\mathrm{PRIMES} = \{2,3,5,7\},\quad \mathrm{EVEN} = \{2,4,6,8,10\},\quad \mathrm{SQUARES} = \{1,4,9\};$$

one can equivalently say that a property of $\{1,\ldots,10\}$ is a function

$$\{1,\ldots,10\} \to \{\texttt{yes},\texttt{no}\};$$

for example, PRIMES can be equivalently viewed as the function, Primes, with

$$\mathrm{Primes}(1) = \texttt{no},\quad \mathrm{Primes}(2) = \texttt{yes},\quad \mathrm{Primes}(3) = \texttt{yes},\quad \mathrm{Primes}(4) = \texttt{no},$$

$$\ldots,\quad \mathrm{Primes}(7) = \texttt{yes},\quad \mathrm{Primes}(8) = \texttt{no},\quad \mathrm{Primes}(9) = \texttt{no},\quad \mathrm{Primes}(10) = \texttt{no}.$$

One often sees $\{\texttt{true},\texttt{false}\}$ instead of $\{\texttt{yes},\texttt{no}\}$.

For a computer program to "recognize" the prime numbers from one to ten, we usually view the computer as being given an element of the set of inputs,

$$\mathcal{I}nputs = \{1,\ldots,10\};$$

the computer then does some computations, following the computer program's instructions, and returns an element of the set of outputs,

$$\mathcal{O}utputs = \{\texttt{yes},\texttt{no}\};$$

the computer program "recognizes" the prime numbers, i.e., the set PRIMES, i.e., the function Primes, provided that the function it computes from inputs to outputs is the function Primes. Let us introduce some standard definitions.

Definition 3.1. Let $\mathcal{I}$ be a set. By a *language* on *input set* $\mathcal{I}$ we mean a subset of $\mathcal{I}$. A *program* on $\mathcal{I}$ is a function $P\colon \mathcal{I} \to \{\texttt{yes},\texttt{no}\}$; furthermore we say that P *recognizes* the language

$$L = \{I \in \mathcal{I} \mid P(I) = \texttt{yes}\}.$$

It is common to use all capital letters for a language; for example

$$\mathrm{PRIMES} = \{2,3,5,7\} \subset \mathcal{I} = \{1,\ldots,10\},$$

so we can view PRIMES as a language over the input set $\mathcal{I} = \{1,\ldots,10\}$. The above definition doesn't care how the program works, just what it outputs on each of its possible inputs.

The pigeon hole principle gives the following easy theorem.

Theorem 3.2. *Let $\mathcal{I}$ be a set, $\mathcal{P}$ a set of programs on $\mathcal{I}$, and $\mathcal{L}$ a set of languages. If $|\mathcal{P}| < |\mathcal{L}|$, then some language is not recognized by any element of $\mathcal{P}$.*

Proof. (This proof is quite informal.) For the sake of contradiction, assume that each language $L \in \mathcal{L}$ is recognized by a program $P \in \mathcal{P}$. Then, by the pigeon hole principle, there are two languages that are recongnized by the same program; but each program recognizes only one language. □

For example, if you have a collection, $\mathcal{P}$, consisting of two programs on $\{1, \ldots, 10\}$, then one of the three languages

$$\{\mathrm{PRIMES}, \mathrm{EVEN}, \mathrm{SQUARES}\}$$

is not recognized by any program in $\mathcal{P}$.

Corollary 3.3. *Let $\mathcal{I}$ be a set, $\mathcal{P}$ a set of programs in $\mathcal{I}$. If*

$$|\mathcal{P}| < 2^{\mathcal{I}},$$

then there is some language, $L \subset I$, that is not recognized by any program in $\mathcal{P}$.

This discussion is quite easy when $\mathcal{P}$ and $\mathcal{I}$ are finite sets. The main point of this section is that the above corollary is also true when $\mathcal{P}$ and $\mathcal{I}$ are infinite sets, provided that we can make sense of the inequality

$$|\mathcal{P}| < 2^{|\mathcal{I}|}. \tag{3.1}$$

Notice that this corollary doesn't necessarily give you a quick way of finding a languages not recognized by $\mathcal{P}$; in principle you might have to examine each $P \in \mathcal{P}$, see which language of $\mathcal{I}$ each P recognizes, and then give some language missed by all the $P \in \mathcal{P}$.

4. Some Infinite Pigeon Hole Principles

We now want to interpret (3.1) when $\mathcal{P}$ and $\mathcal{I}$ are infinite sets. In the typical applications, the set of programs, $\mathcal{P}$, and the set of inputs, $\mathcal{I}$, are infinite, but only *countably* infinite (this is the "smallest" size of infinity). It turns out that (3.1) still holds.

To talk about different sizes of infinity, one would usually discuss some general aspects about set theory, as done in Sipser's textbook in Section 4.2 (on the Halting Problem); here we want to avoid this, and only talk about countable sets versus uncountable sets.

Definition 4.1. The *natural numbers*, denoted $\mathbb{N}$, is the set

$$\mathbb{N} = \{1, 2, 3, \ldots\}.$$

A set $\mathcal{S}$ is *countable* if any of the equivalent conditions hold:

(1) There is a sequence of elements of $\mathcal{S}$,
$$s_1, s_2, s_3, \ldots$$
such that each element of $\mathcal{S}$ appears at least once in this sequence.
(2) Either $\mathcal{S}$ is a finite set, or there is a bijection $f \colon \mathbb{N} \to \mathcal{S}$.
(3) There is an injective map $\mathcal{S} \to \mathbb{N}$.

A set is *uncountable* if it is not countable. A set is *countably infinite* if it is countable and infinite.

It is not hard to prove that all the conditions are actually equivalent; there are analogous statements for set in general (that may be uncountably infinite), but this requires more set theory.

Example 4.2. The set of integers is countable, the set of rational numbers is countable. The set of real numbers is uncountable (see Sipser, Section 4.2 for a proof). Any subset of a countable set is countable. Any superset of an uncountable set is uncountable.

Let's discuss the case of interest to us.

Definition 4.3. Let $\mathcal{A}$ be a set. A *word over (the alphabet)* $\mathcal{A}$ is a finite sequence of elements of $\mathcal{A}$; we understand that this includes a single sequece of length zero, the *empty string*. We often omit the phrase "the alphabet." We denote (1) the empty string by ϵ, (2) the set of all words over $\mathcal{A}$ by $\mathcal{A}^*$, and (3) the set of strings of length k (where k is a non-negative integer) by $\mathcal{A}^k$.

For example,

$$\{a,b\}^2 = \{aa, ab, ba, bb\}$$

(here ab denotes the sequence whose first element is a, and second element is b, and similarly for all words), and

$$\{c,d\}^* = \{\epsilon, c, d, cc, cd, dc, dd, ccc, ccd, \ldots\}.$$

Theorem 4.4. *Let $\mathcal{A}$ be a non-empty set that is either finite or countably infinite. Then $\mathcal{A}^*$ is countably infinite.*

Proof. Since $\mathcal{A}$ is non-empty, it contains some element, $a \in \mathcal{A}$. Then $\mathcal{A}^*$ contains the distinct words

$$\epsilon, a, aa, aaa, a^4, a^5, \ldots$$

(here a^4 is short for $aaaa$, and similarly for a^n for all integers n), and hence $\mathcal{A}^*$ is infinite. To show that $\mathcal{A}^*$ is countable, we use the following lemma.

Lemma 4.5. *The set $\mathbb{N}^*$, i.e., the set of all words over $\mathbb{N}$, i.e., of all finite sequences of elements of $\mathbb{N}$, is countable.*

Proof. For any finite sequence of natural numbers, $i_1, \ldots, i_k$, define

$$\text{weight}(i_1, \ldots, i_k) = i_1 + i_2 + \cdots + i_k, \tag{4.1}$$

and weight of the empty sequence to be 0 (this—in a sense—already follows from (4.1)). We claim that the number of elements of $\mathbb{N}^*$ with a given weight is finite (homework). Hence we can list all sequences of weight zero, then all of weight one, then all of weight two, etc., we get a countable sequence which contains all of $\mathbb{N}^*$. □

Returning to the proof of Theorem 4.4, since $\mathcal{A}$ is countable, there is a sequence

$$a_1, a_2, a_3, \ldots$$

such that each element of $\mathcal{A}$ is some element of this sequence. But then the map $\mathbb{N}^* \to \mathcal{A}^*$ given by

$$(i_1, \ldots, i_k) \mapsto a_{i_i} a_{i_2} \ldots a_{i_k}$$

is surjective. Since $\mathbb{N}^*$ is countable, this surjection shows that $\mathcal{A}^*$ is countable. □

Theorem 4.4 can also be proved from some more general lemmas below, whose proofs are homework.

Lemma 4.6. *Let $\mathcal{A}$ and $\mathcal{B}$ be countable sets. Then*

$$\mathcal{A} \times \mathcal{B} = \{(a, b) \mid a \in \mathcal{A},\ b \in \mathcal{B}\}$$

is countable.

Corollary 4.7. *If $\mathcal{A}$ is countable, then for any integer k, $\mathcal{A}^k$ is countable.*

Lemma 4.8. *If $\mathcal{A}_1, \mathcal{A}_2, \ldots$ are countable sets, then*

$$\bigcup_{i \in \mathbb{N}} \mathcal{A}_i$$

is a countable set.

Aside from the set of real numbers, there is a very important example of an uncountable set.

Notation 4.9. If $\mathcal{I}$ is any set, then we denote by $2^{\mathcal{I}}$ and $\mathcal{P}ower(I)$ the set of subsets of $\mathcal{I}$.

Theorem 4.10. *Let $\mathcal{I}$ be any set. Then there is no bijection (i.e., one-to-one correspondence) between $\mathcal{I}$ and $2^{\mathcal{I}}$, the set of subsets of $\mathcal{I}$. In particular, if $\mathcal{I}$ is countably infinite, then $2^{\mathcal{I}}$ is uncountable.*

We remark that if $\mathcal{I}$ is finite, this implies that $n < 2^n$ for any natural number, n.

Proof. For the first assertion, say that $f \colon \mathcal{I} \to 2^{\mathcal{I}}$ is a bijection. Let

$$J = \{j \in I \mid j \notin f(j)\}.$$

We claim that J there is no $i \in \mathcal{I}$ such that $f(i) = J$; indeed, if such an i exsited, then

(1) it is impossible that $i \in J$: otherwise, by definition of the set J, this would imply that $i \notin f(i) = J$, so $i \notin J$, which contradicts the fact that $i \in J$;
(2) it is impossible that $i \notin J$: otherwise, by definition of the set J, this would imply that $i \in f(i) = J$, so $i \in J$, which contradicts the fact that $i \notin J$.

Hence it is impossible that $f(i) = J$ for any $i \in \mathcal{I}$, which means that J lies nowhere in the image of f, which was assumed to be a bijection. This proves the first assertion.

For the second assertion of the theorem, assume that $\mathcal{I}$ is countably infinite. Since $2^{\mathcal{I}}$ contains each singleton set, $\{i\}$, for $i \in \mathcal{I}$, it follows that $2^{\mathcal{I}}$ is infinite. If $2^{\mathcal{I}}$ were countable, then there would exist a bijection between $2^{\mathcal{I}}$ and $\mathbb{N}$; since $\mathcal{I}$ is countably infinite, there exists a bijection bewteen $\mathbb{N}$ and $\mathcal{I}$, which would then imply that there is a bijection between $2^{\mathcal{I}}$ and $\mathcal{I}$. But this contradicts the first assertion. □

We note that the above proof (1) is reminiscient of self-referrencing, and (2) is called *diagonalization* in Sipser's textbook. We shall explore these ideas in the exercises.

At this point we have a fundamental result regarding many notions of a "computer program" and the languages they recognize. Most computer programs—such as those in C, those in Java, Turing machines, RAM machines, etc.—are described by a finite string over a fixed alphabet. Hence the set of all such programs is countable. Also, we generally regard the inputs to these programs to be the set of all strings over a fixed, finite alphabet, $\mathcal{A}$.

Corollary 4.11. *Let $\mathcal{A}$ be a non-empty set. Let $\mathcal{P}$ be a countable set of programs on the input set consisting of the words over the alphabet $\mathcal{A}$. Then there is a language in the alphabet $\mathcal{A}$ that is not recognized by any element of $\mathcal{P}$.*

4.1. Russell's Paradox.

You should be aware that the discussion in this section used the language of set theory. Russell's Paradox will show you that there are some pitfalls in set theory, and that one needs to procede carefully in such matters. The paradox is

Let S be the set of all sets that do not contain themselves.

It is easy to see that both $S \in S$ and $S \notin S$ lead to contradictions (the details will be done in class). We had something similar in the proof of Theorem 4.10. However, this technique in the proof of Theorem 4.10 was used to prove a new theorem; in Russell's paradox, it serves to show us that one cannot have a set theory in which the collection of all sets is a set.

Indeed, consider the usual way of resolving the paradox. It seems bizarre that a set should be able to contain itself, so let's assume this is impossible. In this case, the above collection, S, consists of all sets. The question is whether the collection of all sets is itself a set: if so, then we get Russell's paradox. This means that the collection of all sets should not be a set, but something larger: it turns out that (1) a collection this large is called a *class*, and (2) one has to be careful that a set of axioms in set theory does not permit ourselves to speak of the collection all sets as a set. Point (2) is a bit subtle, because we do want the collection of all subsets of a set, T, (denoted $\mathcal{P}ower(T)$ or 2^T) to still be a set.

So in practice one can (1) ignore Russell's paradox, (2) have faith that set theorists have worked out axioms for set theory (such as Zermelo-Fraenkel set theory) which get around this paradox, or (3) embark upon further study of set theory. In any case, you should know that set theory allows for certain constructions to be sets, but that the collection of all sets is no allowed to be a set.

5. Assumptions Regarding Programs and Universal Programs

In the next section we describe explicit languages that are not recognized by any computer program. This is true for a number of notions of what is meant by a "computer programs." Here we describe intuitively our assumptions regarding the context of such programs.

5.1. Intuitive Description.

Consider a program in C (or BASIC, Fortran, Lisp or Scheme, Java, etc.). Each such program can be described by a set of ASCII characters (the ASCII alphabet is an essentially standard 256 letter alphabet which includes all English letters and punctuation marks, the digits 0–9, a "carriage return," etc.). It can take its input in a number of fashions, one of which is called the "standard input" stream (in modern language), and can give its output in a number of fashions, such as outputting to a file, to the "standard output" and the "standard error." One has analogous statements regarding theoretical computation models, such as Turing machines. We will following assumptions on our computing environment:

(1) Our programs can take an arbitrarily long ASCII string as input; typically this involves scanning an input array (or stream) during program execution,

where there is a special designated ASCII character which is the "end-of-stream" character, letting us know that this is the end of the input string.
(2) Our programs can output an arbitrarily sequence of ASCII characters on some device/stream/file/array/etc.
(3) Generally speaking, we simplify things by assuming there is one designated input array, and one designated output array; we also assume that the program has another infinite read/write array at its disposal to perform computations.
(4) For simplicity, for now we assume that all programs output one of two strings: a `yes` string (or "true"), and a `no` string (or "false"). Later we will use programs (Turing machines) that compute functions, that take an input string and output an arbitrary ASCII string.
(5) On a given input, a program either never halts (possibly—but not necessarily—due to some "infinite loop"), or halts. When a program halts, its output is determined by what it has written to the output array.
(6) There exist *universal programs* (see below).

A *universal program* is a computer program, U, that takes as input an encoding of a pair (p, i), where p is a program and i is an input; the program is given as an ASCII string, followed by the "end-of-stream" character, followed by the input (which ends in another "end-of-stream" character). The universal program then "simulates" what would happen if the program p is run on input i, usually "step-by-step"; so if p doesn't halt when given input i, then U shouldn't halt either; otherwise, when U is given (p, i) as input, it should halt and output exactly what p would have outputed on input i.

It is not hard to convince yourself that you could write a universal program (if you had a lot of spare time) for any of the standard languages and for Turning machines (once you learn how Turing machines work).

In the next section we write down axioms that are satified in any of the above mentioned notions of "computer programs." They are fairly abstract, and—as in Sections 3 and 4—we view a program as a function from the inputs to a fixed set of outputs: so we do not care how these programs are actually running.

5.2. **The Language 421Simple.**

The following programming language is very simple, akin to how a Turing machine works. We shall call it "421Simple."

It has a set of reserve words; any other word is interpreted as an integer variable, taking on the values $1, 2, 3, \ldots$ Each line of the program begins with a positive integer, giving its "line number." The reserved words are:

(1) `INPUT`,`WORKTAPE`, `OUTPUT`, which are arrays indexed on the positive integers whose compents take values in the 256 set of ASCII characters, so that "`INPUT[3]`" refers to the third character of the array `INPUT`;
(2) `GOTO` is an instruction to jump to a certain line number;
(3) `IF` and `THEN GOTO` instruct the computer to jump to a certain line number if the Boolean expression between `IF` and `THEN GOTO` evaluates to true.
(4) `AUG`, `DEC`, `RESET` are intructions to augment a variable by one (i.e., add 1 to it), decrement by one (i.e., subtract 1 from it), and reset its value to 1;
(5) `EOF` is the ASCII character "end of file" character;
(6) $=$ is used to test equality of ASCII characters or of integers;

(7) LET is the beginning of an assignment statement, such as “LET COUNTER = 3”;
(8) character constants are quoted ASCII characters, such as “LET OUTPUT[3] = ”C””;
(9) END means that the program halts at this statement;
(10) COMMENT means that anything following this word until the next “end of line” character is a comment, to be ignored.

Here is a sample program example:

```
 10  RESET COUNTER
 20  COMMENT      Here is a loop
 30  IF INPUT[COUNTER] = EOF  THEN GOTO 60
 40  AUG COUNTER
 50  GOTO 30
 60  COMMENT      The loop is over, and the counter is on the INPUT EOF
 70  DEC COUNTER
 80  IF INPUT[COUNTER] = "0" THEN GOTO 110
 90  LET OUTPUT[1] = "0"
100  END
110  LET OUTPUT[1] = "1"
120  END
```

This program outputs a “0” (as the first cell/component/character of OUTPUT) if the last character of INPUT is not equal to “0,” and otherwise outputs a “1”.

We claim that any C program can be written as a 421Simple program. This will take some time to see.

Notice how primitive this language is: for example, lines 10–50 form what would be called a “for” loop (one or two lines in the language C), although we have to write this in terms of an “if-then” loop.

The language 421Simple is very close to the classical model of a Turing machine.

5.3. **The Language 421TRAN.** In this subsection we describe another language which we will call “421TRAN,” which is more powerful than 421Simple, as an intermediary between 421Simple and programming languages in use, such as C, Java, awk, Python, etc.

To describe 421TRAN we will simply add some more reserve words and more variable types. Specifically we wish to add:

(1) ADD is a function that takes two integers and adds them, and this requires a (and) to be matching parenthesis to describe the arguments of the ADD function, and requires LET in the beginning, such as

```
10 LET C = ADD(A,B)  .
```

(2) ARRAY is a word that declares the word the follows to be an infinite array of 64-bit positive integers, such as

```
10 ARRAY A
```

which declares means that we can use A[1], A[2], . . .
(3) We could add some logical functions, and/or more integer functions such as subtraction, multiplication.
(4) We could add some more loops, such as some form of “while” loops, “for” loops, etc.

(5) We could add subroutines or subfunctions.

It should be easy to see that all the above can be expressed in the language 421Simple (the `WORKTAPE` array would be used for all the `ARRAY` variables, so this would be quite tedious). Yet 421TRAN looks a lot closer to modern programming languages, so we may feel more comfortable writing such programs.

6. Deciding and Recognizing

In this section we want to give an explicit description of a language that cannot be recognized by any C program (or Java program, or Turing machine). In the last section we used counting and an infinite pigeon hole principle to show that such a language exists, although we did not have a simple description of such a language.

The trade-off is that we have to make some stronger assumptions about our computer programs.

6.1. **The Axioms and Main Theorems.**

In this section we will list all the axioms we need to make about our notion of computing; with these axioms we will be able to describe certain specific languages cannot be solved with a computer program; we will also need two refinements of this notion of "solved," namely *deciding* and *recognizing* (sometimes recognizing is called *accepting*). We will also give illustrative examples of the main theorems that we prove, leaving the proofs and a fuller discussion for later subsections.

These axioms will hold for programs in most standard computer languages (C, Java, etc.) and for ideal models of computers (Turning machines, RAMs, etc.)

We will explain these axioms in the subsections that follow; however, we feel it best to put all axioms on the table.

We remind the reader that our notion of a program in the previous sections does not refer to the program description or how it works: we only care what results it returns on certain inputs. For this reason, our axioms will seem fairly abstract.

We will also give some definitions and our main theorems.

Axiom 1. *We are given sets, $\mathcal{P}$ (the "programs"), $\mathcal{I}$ (the "inputs"), and a map:*

$$\text{Result}\colon \mathcal{P} \times \mathcal{I} \to \mathcal{R},$$

where $\mathcal{R} = \{\texttt{yes}, \texttt{no}, \texttt{NoHalt}\}$ (the "results").

Intuitively, this means that our programs may never stop running, or can halt and output "yes" or "no." Computer programs that never stop running may have some sort of "infinite loop" or they may not; we don't really care which, and the term `NoHalt` is used for any computation that never stops.

The above axiom permits two distinct programs in $\mathcal{P}$ to yield the same result on any input; it would be extremely unrealistic to assume otherwise (see the homework).

One can also consider programs as taking an input and—if they halt—giving an output that is larger than the set $\{\texttt{yes}, \texttt{no}\}$. This has advantages and disadvantages (see the homework).

Definition 6.1. By a *language* we mean a subset of $\mathcal{I}$.

Definition 6.2. We say that a program, $P \in \mathcal{P}$, is a *decider* if the result of P is never `NoHalt`, i.e.,

$$\forall i \in \mathcal{I}, \quad \text{Result}(P, i) \in \{\texttt{yes}, \texttt{no}\}.$$

A program $P \in \mathcal{P}$ is said to *recognize* the language

$$L_P = \{i \in \mathcal{I} \mid \mathrm{Result}(P, i) = \texttt{yes}\}.$$

We say that P *decides* the language L if (1) L is recognized by P and (2) P is a decider. We say that a language, L, is *decidable* if it is decided by some $P \in \mathcal{P}$; othewise we say that L is *undecidable.* We say that a language, L, is *recognizable* if it is recognized by some $P \in \mathcal{P}$; othewise we say that L is *unrecognizable.*

So Axiom 1 allows us to define what is meant by a language, and the most fundamental properties of a language. The next axiom says that there is a "universal machine" in our setup.

Axiom 2. *There is a fixed injection* EncodeBoth: $\mathcal{P} \times \mathcal{I} \to \mathcal{I}$*, and a* $U \in \mathcal{P}$ *such that for all* $P \in \mathcal{P}$ *and* $x \in \mathcal{I}$*,*

$$\mathrm{Result}(U, \mathrm{EncodeBoth}(P, x)) = \mathrm{Result}(P, x)$$

The injection EncodeBoth *is fixed as part of the axioms; there is typically more than one program* $U \in \mathcal{P}$ *as above, and any such* $U \in \mathcal{P}$ *is called a* universal program*.*

Intuitively, the function EncodeBoth takes an arbitrary program and an input, and encodes both of them as a single element of $\mathcal{I}$ (i.e., a single input).

Notation 6.3. We use $\langle P, i\rangle$ to denote EncodeBoth(P, i). We use $P[i]$ to denote Result(P, i).

The notation $\langle P, i\rangle$ is the notation used in Sipser's textbook; it emphasizes that $\langle P, i\rangle$ is the description (as a string) of the pair (P, i); this is less cumbersome that writing EncodeBoth everywhere. We introduce the second notation to make things less cumbersome. We emphasize that in making things less cumbersome, we must be careful to remember the precise meaning of the notation; otherwise our proofs may seem deceivingly short.

For example, a universal program, U, as defined in Axiom 2, can be described via the condition

$$\forall P \in \mathcal{P}, \quad \forall i \in \mathcal{I}, \qquad U[\langle P, i\rangle] = P[i].$$

Definition 6.4. The *Accepting Problem* is the language

$$\mathrm{A}_{\mathcal{P}} = \{\langle P, x\rangle \mid P \in \mathcal{P},\ x \in \mathcal{I}, P[x] = \texttt{yes}\}$$

(this conforms to Sipser's textbook's notation, except that Sipser TM is used because he only discusses this for Turing machines). The *Halting Problem* is the language

$$\mathrm{HALT}_{\mathcal{P}} = \{\langle P, x\rangle \mid P \in \mathcal{P},\ x \in \mathcal{I}, P[x] \neq \texttt{NoHalt}\}.$$

Now we know some langauges that we suspect are not decidable. Sipser's textbook uses $\mathrm{A}_{\texttt{TM}}$ and $\mathrm{HALT}_{\texttt{TM}}$ in Section 4.2 since his model of a computer program is a *Turing machine.*

Recall that $\mathcal{R}$ denotes the set $\{\texttt{yes}, \texttt{no}, \texttt{NoHalt}\}$, and intuitively speaking describes the possible results of a program running on a given input. We shall make use of the negation function defined as follows.

Definition 6.5. By the *negation function*, denoted $\neg$, we mean the function from $\mathcal{R}$ to $\mathcal{R}$ defined by

$$\neg(\texttt{yes}) = \texttt{no}, \quad \neg(\texttt{no}) = \texttt{yes}, \quad \neg(\texttt{NoHalt}) = \texttt{NoHalt}.$$

Axiom 3. *Let $f\colon \mathcal{R} \to \mathcal{R}$ be any function with $f(\texttt{NoHalt}) = \texttt{NoHalt}$. Given any program, $P \in \mathcal{P}$, there is a program, P', such that*

$$\forall i \in \mathcal{I}, \quad P'[i] = f\bigl(P[i]\bigr).$$

The negation is an example of a function $f\colon \mathcal{R} \to \mathcal{R}$ that satisfies $f(\texttt{NoHalt}) = \texttt{NoHalt}$. This axiom intuitively says that any program P can be modified so that when it halts and gives `yes` or `no`, we can change its result on either of these to any of the three possible results in $\mathcal{R}$. This axiom seems simple, but this will be crucial in constructing a sort of "paradoxical program."

Axiom 4. *There is a fixed injection* EncodeProg: $\mathcal{P} \to \mathcal{I}$. *For any $Q \in \mathcal{P}$, we use $\langle Q \rangle$ to denote* EncodeProg(Q). *Given any program, P, there is a program, P', such that*

$$\forall Q \in \mathcal{P}, \qquad P'[\langle Q \rangle] = P[\langle Q, \langle Q \rangle \rangle].$$

This means that part of the axioms include a fixed way to encode any program, Q, as an input string $\langle Q \rangle \in \mathcal{I}$; furthermore, any program, P, can be modified so that when it reads $\langle Q \rangle$ it runs as if it saw the input representing $(Q, \langle Q \rangle)$. In most applications P' works by (1) converting $\langle Q \rangle$ to $(Q, \langle Q \rangle)$ by a preliminary calculation, and then (2) running P on $(Q, \langle Q \rangle)$.

We remark that we are following the notation $\langle Q \rangle$ in Sipser's textbook to denote EncodeProg(Q); in this textbook, angle brackets always mean "the description as a string of whatever is inside the angle brackets." However, this leads to nested angle brackets, which is (1) could be confusing, and (2) tempts the reader to gloss too quickly over what is really going on (at least when I am the reader...). So you should always en garde to the fact that

$$\langle Q, \langle Q \rangle \rangle = \text{EncodeBoth}(Q, \text{EncodeProg}(Q)).$$

You should also remember that there are many choices involved as to what conventions one uses to define EncodeProg, and to define EncodeBoth, and that these conventions must (almost always) be viewed as fixed.

Axiom 4, which intuitively gives us the power to take a string of the form $\langle Q \rangle$ and convert it to the string representing $(Q, \langle Q \rangle)$, is crucial—along with Axioms 2 and 3—to constructing a sort of "paradoxical program."

Theorem 6.6. *Under Axioms 1–4,* $\mathrm{A}_{\mathcal{P}}$ *is undecidable.*

We remark that, as best we can tell, Axioms 1–4 do not enable us to prove that the Halting Problem, $\mathrm{HALT}_{\mathcal{P}}$, is undecidable (see Axiom 6 below).

Axiom 5. *For any programs, P, Q, there is a program, C, such that for all $i \in \mathcal{I}$ we have*

$$P[i] = \texttt{yes} \quad \text{and} \quad Q[i] \neq \texttt{yes} \quad \Rightarrow \quad C[i] = \texttt{yes},$$

and

$$Q[i] = \texttt{yes} \quad \text{and} \quad P[i] \neq \texttt{yes} \quad \Rightarrow \quad C[i] = \texttt{no}.$$

Intuitively this means we can run P and Q together, say alternating one step between each machine; provided that either P or Q says "yes" and the other doesn't (i.e., says "no" or loops infinitely), then the result of C is "yes" or "no" according to which of P or Q says "yes." Note that we do not care what $C[i]$ is if neither of $P[i]$ or $Q[i]$ equals `yes`.

The following theorems are easy to prove.

Theorem 6.7. *If a language, L, and its complement ($\overline{L} = \mathcal{I} \setminus L$) are recognizable, then L (and its complement) are decidable. In particular, if L is recognizable but not decidable, then $\overline{L}$ is not recongnizable.*

Proof. The first statement follows immediately from Axiom 5. The second statement is a consequence of the first. □

Theorem 6.8. *The languages $\mathrm{A}_\mathcal{P}$ and $\mathrm{HALT}_\mathcal{P}$ are recognizable.*

Proof. Let U be any universal program, and let $f : \mathcal{R} \to \mathcal{R}$ be given by

$$f(\texttt{yes}) = \texttt{yes}, \quad f(\texttt{no}) = \texttt{NoHalt}, \quad f(\texttt{NoHalt}) = \texttt{NoHalt}.$$

Applying Axiom 3 with the above choice of f and with $P = U$, there is a program $U' \in \mathcal{P}$ such that

$$\forall i \in \mathcal{I}, \quad U'[i] = f\bigl(U[i]\bigr).$$

Then $\mathrm{A}_\mathcal{P}$ is recognized by U'.

Similarly we show that $\mathrm{HALT}_\mathcal{P}$ is recognizable, with the same argument except that we replace the above f by the function, f, given by

$$f(\texttt{yes}) = \texttt{yes}, \quad f(\texttt{no}) = \texttt{yes}, \quad f(\texttt{NoHalt}) = \texttt{NoHalt}.$$

□

Corollary 6.9. *The complement of $\mathrm{A}_\mathcal{P}$ is unrecognizable.*

The point of these axioms and theorems is to show that any "sufficiently rich" programming or computing environment (and we don't need that much richness...) has a type of Acceptance Problem that cannot be "solved" by a computer program.

It is not clear to us that Axioms 1–4 allow us to conclude that the Halting Problem, i.e., $\mathrm{HALT}_\mathcal{P}$ of Definition 6.4, is undecidable. To do this we will add an axiom that is a stronger form of Axiom 3.

Axiom 6. *Let $f : \mathcal{R} \to \mathcal{R}$ be any function with $f(\texttt{NoHalt}) = \texttt{NoHalt}$. There is a map $\mathrm{Compose}_f : \mathcal{P} \to \mathcal{P}$, such that*

(1) using $f \circ P$ to denote $\mathrm{Compose}_f(P)$, we have that

$$\forall i \in \mathcal{I}, \quad (f \circ P)[i] = f\bigl(P[i]\bigr);$$

and

(2) given any program, $R \in \mathcal{P}$, there is a program, R', such that

$$\forall Q \in \mathcal{P},\ \forall i \in \mathcal{I}, \qquad R'[\langle Q, i\rangle] = R[\langle f \circ Q, i\rangle].$$

The first assertion in this axiom is just Axiom 3. The second assertion will hold in all our applications because (1) there is a way to take modify any program, P, by "tacking on" some "extra lines of code" (or some "extra states" if we are working with Turning machines) to P to get $f \circ P$, and (2) this "tacking on" process can be done on the level of the description of (P, i); hence we can implement R' by converting $\langle P, i\rangle$ to $\langle f \circ P, i\rangle$, and feeding the latter into R.

Theorem 6.10. *Under Axioms 1,2 4, and 6, $\mathrm{HALT}_\mathcal{P}$ is undecidable.*

Corollary 6.11. *Under Axioms 1,2 and 4—6, the complement of $\mathrm{HALT}_\mathcal{P}$ is unrecognizable.*

6.2. **Settings Where the Axioms Hold.** You should be able to take almost any standard programming language (C, C++, Fortran, Basic, Jave, awk, Python, etc.), that can be written as a finite ASCII string, and solve the following problems:

(1) What are the sets $\mathcal{P}$ and $\mathcal{I}$, and what is the function Result?
(2) What are the maps EncodeProg and EncodeBoth?
(3) Show that Axioms 1–6 hold.

6.3. **Remarks on the Axioms.** In this subsection we make some remarks about the above axioms and definitions.

Terms such as "recognizable" have a lot of variations in the literature. In particular, the term "recognizable" is sometimes called "acceptable," "Turing-acceptable" (in the context of Turing machines), and recursively enumerable. The term "decidable" is sometimes called "recursive."

In the literature (but not in this course), notions such as "recognizable" and "decidable" may depend on $\mathcal{P}$, in which case we may write $\mathcal{P}$-recognizable and $\mathcal{P}$-decidable. Generally speaking, these are usually discussed in the model of Turing machine computations; however, these notions are essentially the same when a Turing machine is replaced by C programs, Fortran programs, etc.

Our choice of the terms "recognize" and "decide" are the ones that are used in Sipser's textbook; he also uses the terminology "P accepts i" to mean

$$P[i] = \texttt{yes}.$$

For this reason it would be confusing to use the term "accepts" or "Turing-accepts" when speaking of languages.

Our axioms intuitively speak of machine computations, but you could write the same definitions and obtain the same theorems for any situations where the axioms hold.

The fact that there is an injection EncodeProg: $\mathcal{P} \to \mathcal{I}$ means that $\mathcal{P}$ is of *cardinality* (intuitively the number of elements of $\mathcal{P}$) at most that of $\mathcal{I}$. This also holds if $\mathcal{I}$ is infinite (assuming that you have a notion of set theory where the "cardinality of a set" behaves well enough). For example, if $\mathcal{I}$ is countably infinite (which will hold for our applications), then $\mathcal{P}$ is automatically at most countable, and hence there are automatically languages that are not recognized by any element of $\mathcal{P}$.

You should be able to convince yourself that in the settings described in Section 5, all the axioms given here hold, and all the terminology is the usual terminology.

Once we know that $\mathrm{A}_{\mathcal{P}}$ is undecidable, and that its complement is unrecognizable, we can say the same about a number of related problems. Some of these problems depend on the specific programming context. For example, a line of "dead code" in a computer program is a line in the program (in languages such as C, Fortran, Java, Python, etc.) that is never used on any input. So, for example, consider the pairs (F, ℓ) where F is a Fortran program and ℓ is an integer between 1 and the number of lines in the program F; from such pairs we can form two languages:

(1) F-DEAD-CODE, consisting of those pairs (F, ℓ) such that the ℓ-th line of F is "dead code," i.e., never reached when F is run on any input, and
(2) F-LIVE-CODE, consisting of those pairs (F, ℓ) such that the ℓ-th line of F is reached at least once when F is run on some input.

Using Theorems 6.6 and 6.7 it is easy to see that F-LIVE-CODE is undecidable, and that F-DEAD-CODE is unrecognizable. Hence no Fortran compiler will be able to remove all lines of "dead code" (keeping its lines of "live code") from an arbitrary Fortran program. Of course, in our axioms we do not look at how the program runs, and there is no simple way to say that a $P \in \mathcal{P}$ has a "line of dead code."

We remark that one can give a different presentation of all these axioms by allowing programs to compute functions in some fashion. For example, one could add to $\mathcal{P}$ and $\mathcal{I}$ another set, one of "functions," denoted $\mathcal{F}$, where the result of a function and an input is $\mathcal{I} \cup \{\texttt{NoHalt}\}$; this gives us functions that either return a value that is an arbitrary element of $\mathcal{I}$, rather than just a value in the set $\{\texttt{yes}, \texttt{no}\}$ (in addition to the possibility of the result being `NoHalt`). This may be a more realistic setting to work in; this probably makes some of our axioms simpler, at the cost of complicating some of the notation (and blurring some of the issues, such as the need for an additional axiom to show that $\mathrm{HALT}_{\mathcal{P}}$ is undecidable).

6.4. **The Proof of Theorem 6.6.**

In this subsection we give the proof of Theorem 6.6. It is subtle, and we will proceed very carefully (and slowly).

Proof of Theorem 6.6. We will assume that $\mathrm{A}_{\mathcal{P}}$ is decidable and get a contradiction; hence we will conclude that $\mathrm{A}_{\mathcal{P}}$ is not decidable.

Assume for the sake of contradiction, that $\mathrm{A}_{\mathcal{P}}$ is decidable, and let $H \in \mathcal{P}$ be a program deciding $\mathrm{A}_{\mathcal{P}}$; hence

$$\forall Q \in \mathcal{P}, \quad H[\langle Q, i\rangle] = \begin{cases} \texttt{yes} & \text{if } Q[i] = \texttt{yes}, \text{ and} \\ \texttt{no} & \text{otherwise.} \end{cases} \tag{6.1}$$

We shall obtain a contradiction, using Axioms 1–4.

According to Axiom 3, where f is the negation function, there exists a program, $E \in \mathcal{P}$, such that

$$\forall Q \in \mathcal{P}, \quad E[\langle Q, i\rangle] = \begin{cases} \texttt{no} & \text{if } Q[i] = \texttt{yes}, \text{ and} \\ \texttt{yes} & \text{otherwise.} \end{cases} \tag{6.2}$$

Since E takes on only the values `yes` and `no`, it follows that E is a decider. Then using Axiom 4, it follows that there is a program, D, such that

$$\forall Q \in \mathcal{P}, \qquad D[\langle Q\rangle] = E[\langle Q, \langle Q\rangle\rangle].$$

In view of (6.2), one can describe D by

$$\forall Q \in \mathcal{P}, \quad D[\langle Q\rangle] = \neg P[\langle Q, \langle Q\rangle\rangle]; \tag{6.3}$$

Combining (6.1) and (6.3) gives us that

$$D[\langle Q\rangle] = \begin{cases} \texttt{no} & \text{if } Q[\langle Q\rangle] = \texttt{yes}, \text{ and} \\ \texttt{yes} & \text{otherwise.} \end{cases} \tag{6.4}$$

Furthermore, since E is a decider, then if D is given any input of the form $i = \langle Q\rangle$, then $D[i] = D[\langle Q\rangle]$ is either `yes` or `no`.

Now consider what D does on input $\langle D\rangle$; by the previous sentence (previous paragraph), we have that either

(1) $D[\langle D\rangle] = \texttt{no}$, or
(2) $D[\langle D\rangle] = \texttt{yes}$.

We shall show that both of these are impossibe; we then conclude that our assumption that $\mathrm{A}_{\mathcal{P}}$ is decidable leads to a contradiction.

Observe that (6.4) implies that

$$D[\langle D\rangle] = \begin{cases} \texttt{no} & \text{if } D[\langle D\rangle] = \texttt{yes}, \text{ and} \\ \texttt{yes} & \text{otherwise.} \end{cases} \tag{6.5}$$

Hence in the first case, where $D[\langle D\rangle] = \texttt{no}$, then (6.4) gives us

$$\texttt{no} = \begin{cases} \texttt{no} & \text{if } D[\langle D\rangle] = \texttt{yes}, \text{ and} \\ \texttt{yes} & \text{otherwise,} \end{cases}$$

which implies that $D[\langle D\rangle] = \texttt{yes}$, which is impossible. Similarly, in the second case, where $D[\langle D\rangle] = \texttt{yes}$, then (6.4) gives us

$$\texttt{yes} = \begin{cases} \texttt{no} & \text{if } D[\langle D\rangle] = \texttt{yes}, \text{ and} \\ \texttt{yes} & \text{otherwise,} \end{cases}$$

which implies that $D[\langle D\rangle] \neq \texttt{yes}$, which is impossible.

It follows that our assumption that $\mathrm{A}_{\mathcal{P}}$ is decidable leads to a contradiction, and hence $\mathrm{A}_{\mathcal{P}}$ is undecidable. □

6.5. **The Proof of Theorem 6.10.**

In this subsection we prove Theorem 6.10 using Theorem 6.6.

Proof of Theorem 6.10. Let $f\colon \mathcal{R} \to \mathcal{R}$ be given by

$$f(\texttt{yes}) = \texttt{yes}, \quad f(\texttt{no}) = \texttt{NoHalt}, \quad f(\texttt{NoHalt}) = \texttt{NoHalt}.$$

Assume (for the sake of contradiction) that $\mathrm{HALT}_{\mathcal{P}}$ is recognized by a decider $R \in \mathcal{P}$. Then it follows that there exists a decider $R' \in \mathcal{P}$ such that

$$\forall Q \in \mathcal{P},\ \forall i \in \mathcal{I}, \qquad R'[\langle Q, i\rangle] = R[\langle f \circ Q, i\rangle] = \begin{cases} \texttt{yes} & \text{if } Q[i] = \texttt{yes}, \text{ and} \\ \texttt{no} & \text{otherwise.} \end{cases}$$

It follows that R' is a decider for $\mathrm{A}_{\mathcal{P}}$, which, by Theorem 6.10 is impossible.

Hence our assumption that $\mathrm{HALT}_{\mathcal{P}}$ is decidable leads to a contradiction; hence $\mathrm{HALT}_{\mathcal{P}}$ is undecidable. □

7. Learning Goals and Sample Exam Problems

It seems best to make general learning goals concrete by connecting them with sample exam problems.

7.1. **Paradoxes (Sections 1 and 2).**

Learning Goals:

(1) You should be familiar with the "paradoxes" of Sections 1 and you should know the standard way of dealing with
   (a) Paradox 3, the "Berry Paradox" (first discussed by Russell, who attributed the paradox to Berry); (see Exercises 7.1.1, 7.1.2, 7.1.3);
   (b) Paradox 4, related to Gödel's Incompleteness Theorems (see Exercises 7.1.4, 7.1.5);
   (c) Paradoxes 5 and 6, relating to Russell's Paradox (see Exercises 7.1.6, 7.1.7).
   (d) Paradox 7, relating to the Acceptance Problem and the Halting Problem (see the exercises in Subsection **??**).

(2) Given a paradox that is similar to the paradoxes in Section 1, you should be able to identify to which paradox it is similar, and in which of the following categories it belongs:
   (a) The paradox goes away when things are stated precisely.
   (b) The paradox doesn't go away, and you have to change your theory if you want it free of contradictions.
   (c) The paradox doesn't go away, but only if certain things are true. Therefore (assuming your setting is free of contradictions) you have a theorem that one of these certain things must be false.

   See Exercises 7.1.8 and 7.1.9. See also the Wikipedia entry "List of Paradoxes," `http://en.wikipedia.org/wiki/List_of_paradoxes`; some (but not all) examples there fall into one of the above categories.

Sample Exam Problems

Exercise 7.1.1. Consider Paradox 3 of Section 1. [This is usually called the "Berry Paradox;" feel free to look it up somewhere.] Let W be the four element set

$$W = \{\texttt{one}, \texttt{two}, \texttt{plus}, \texttt{times}\}.$$

Ascribe a "meaning" to each sentence with words from W (i.e., each string over the alphabet W) in the usual way of evaluating expressions, so that

$$\texttt{one plus two times two} \quad \text{means} \quad 1 + 2 \times 2 = 5,$$

$$\texttt{plus times two plus} \quad \text{is meaningless,}$$

and each sentence either "means" some positive integer or is "meaningless."

(1) Show that every positive integer is the "meaning" of some sentence with words from W.
(2) Can one form a paradox akin to Paradox 3 in the framework of sentences in W^* and the above notion of the meaning of a sentence? Explain.

Exercise 7.1.2. Consider the five element set

$$U = W \cup \{\texttt{moo}\},$$

where W is the set in Exercise 7.1.1 with the ascribed meanings there, and where `moo` has the following meaning:

(1) if it appears anywhere after the first word of a sentence, then the sentence is meaningless,
(2) if it appears only once and at the beginning of a sentence, then we evaluate the rest of the sentence (as usual), and
   (a) if the rest evaluates to the integer k, then the sentence means "the smallest positive integer not described by a sentence of k words or fewer," and
   (b) if the rest evaluates to meaningless, then the sentence is meaningless.

For example, "`moo moo`" and "`moo plus times two`" are meaningless, and "`moo two times two`" means "the smallest positive integer not described by a sentence of four words or fewer."

(1) What is the meaning of "`moo one`"?
(2) What seems paradoxical in trying to ascribe a meaning to "`moo two`"?
(3) Explain why this "paradox" is similar to Paradox 3 of Section 1.
(4) Give one way of resolving this paradox.

Exercise 7.1.3. Explain the standard way of resolving Paradox 3 (the "Berry Paradox"). Do we get a new theorem from this paradox?

Exercise 7.1.4. Consider Paradox 4 of Section 1. [This is the basis for Gödel's Incompleteness Theorems.] Assume that there is some setting such that you have a statement, S, which asserts that

"There is no proof that S is correct."

The word "setting" is imprecise and a bit of a loaded; see Appendix A; informally, this means that you have an alphabet, including $\exists, \neg, (,), \Rightarrow$, that allows you to make "statements," which are certain strings over the alphabet, which are true/false assertions in the "usual sense." Argue informally that if (1) S must be true or false, (2) S cannot be both true and false, and (3) any statement that has a proof that it is correct is a true statement, then S must be true but not provably true.

Exercise 7.1.5. Summarize the signficance Paradox 4 and its role in Gödel's Incompleteness Theorems.

Exercise 7.1.6. Assume that you have a form of set theory in which no set can contain itself. How does this lead to the standard way of resolving Russell's paradox (Paradox 6 of Section 1).

Exercise 7.1.7. Discuss how one can resolve the paradoxical statement "I write about those who don't write about themselves." Does this paradox give a new theorem?

Exercise 7.1.8. Consider the three general categories of "paradoxes:"

(1) The paradox goes away when things are stated precisely.
(2) The paradox doesn't go away, and you have to change your theory if you want it free of contradictions.
(3) The paradox doesn't go away, but only if certain things are true. Therefore (assuming your setting is free of contradictions) you have a theorem that one of these certain things must be false.

For Paradoxes 1–7 of Section 1, say in which category (or categories) it lies, and give a brief justification.

Exercise 7.1.9. Identify the "paradox" in the following statements and resolve the following paradoxes (in the sense of Exercise 7.1.8):

(1) "I know that I know nothing."
(2) "A hair dresser in town dresses the hair of anyone who does not dress their own hair."
(3) "If you remove a grain of sand from a heap of sand, you still have a heap of sand." By removing grains of sand, one by one, you conclude that a collection of zero grains of sand is a heap.
(4) "All things in moderation."

7.2. **Counting (Sections 3 and 4).**

Learning Goals:

(1) You should be familiar with the standard methods of showing that a set is countable (Exercises 7.2.1, 7.2.2, 7.2.3, 7.2.4).

(2) You should be able to explain what technique in Sipser's textbook (in Section 4.2) is called "diagonalization" and how to use this idea in a number of different contexts to show that a set is uncountable (Exercises 7.2.5, 7.2.6, 7.2.7).

Sample Exam Problems

Exercise 7.2.1. Show that a countable union of countable sets is countable; i.e., if $C_1, C_2, \ldots$ are countable sets, show that $C_1 \cup C_2 \cup \cdots$ is countable.

Exercise 7.2.2. Let $\mathcal{A}$ and $\mathcal{B}$ be countable sets. Show that

$$\mathcal{A} \times \mathcal{B} = \{(a, b) \mid a \in \mathcal{A},\ b \in \mathcal{B}\}$$

is countable.

Exercise 7.2.3. Use the statements in Exercises 7.2.1 and 7.2.2 to show that if Σ is countable, then Σ^* is countable. [That is, do not simply quote Theorem 4.4.]

Exercise 7.2.4. Show that the set of rational numbers (including 0 and negative rational numbers) is countable.

Exercise 7.2.5. Summarize the proof in Sipser's textbook that the real numbers are uncountable. Explain why this is proof called "diagonalization."

Exercise 7.2.6. Let F be the set of functions from the integers to the integers. Show from scratch, by "diagonlization," that F is uncountable.

Exercise 7.2.7. Let C be any countable set of real numbers, and let $\overline{C} = \mathbb{R} \setminus C$ be the complement of C (i.e., the set of reals numbers that do not lie in C). Show that $\overline{C}$ is uncountable.

Exercise 7.2.8. Consider the set of all functions from $\mathbb{N} = \{1, 2, \ldots\}$ to $\{1, 2, 3\}$. Is this set countable or uncountable?

7.3. **Decidability and Recognizability (Sections 5 and 6).**

Learning Goals:

(1) You should be able to show that for most common programming languages (C, awk, Fortran, Python, etc.) and Turing Machines, Axioms 1–6 hold under some reasonable conditions. See Exercises 7.3.1 and 7.3.2.
(2) You should be able to show that some problems related to "the acceptance problem" and "the halting problem" are undecidable or even unrecognizable; generally, you do this by assuming—for the sake of contradiction—that the related problem is recognizable or decidable, and then show that this implies that the acceptance problem or the halting problem is decidable. See Exercise 7.3.3.
(3) You should be able to write some 421Simple code and—once we define Turing machines (Section 3.1 of Sipser's textbook)—some complete Turing machines descriptions for relatively simple problems. See Exercise 7.3.4.

Exercise 7.3.1. Pick a standard programming language (C, Python, Java, awk, perl, etc.), and outline in a few paragraphs that they satisfy Axioms 1–6. (Axioms 2 and 4 are the only serious issues.)

Exercise 7.3.2. (As soon as we define Turing machines, Chapter 3, Section 1:) Show that if we represent Turing machines in one of the "standard ways" as binary strings, and allow only $\Sigma = \{0, 1\}$ to be the alphabet of the inputs to our Turing machines, then we can define a suitable "EncodeBoth" function so that Axioms 1–6 hold. Describe your answer in rough terms in a few paragraphs. (Axioms 2 and 4 are the only serious issues.)

Exercise 7.3.3.
Consider the languages:

(1) 421Simple programs and inputs such that running that program on that input leads to the assignment of at least one value of OUTPUT; i.e., reach a statement that begins with "LET OUTPUT";
(2) 421Simple programs that reach line number 40 on at least one input;
(3) 421Simple programs that give identical outputs on every input.

Explain why each of these problems is recognizable by a 421Simple program (or a C program, or a Python program, etc.); argue that each of these problems is not decidable. [Argue, for any of the above problems, that if this problem were decidable, then the Halting problem or Acceptance problem would have to be decidable.]

Exercise 7.3.4. Write out a complete 421Simple program and a complete Turing machine description (on the level of listing the "δ-function") for the following problems:

(1) the language of strings that are palindromes over $\{0, 1\}$ (i.e., the string equal to its own reversal, such as 110010011);
(2) the language of strings over $\{0, 1\}$ which have twice as many 0's as 1's (see Sipser's text).

Appendix A. Logic in Complexity Theory and Gödel's Incompleteness Theorems

In this course we will prove that certain techniques will not be able to resolve the problem of P versus NP. This will be done by using a standard set of ideas from logic; this idea is also used to study notions in logic such as Gödel's incompleteness theorems. Let us explain these ideas.

A.1. **Gödel's Incompleteness Theorems.** Many things we do in logic can be described in intuitive terms, although it can be equally valuable to describe things precisely. We shall give the precise setting of one of Gödel's incompleteness theorems, although we refer to [End01] for details.

To summarize this subsection, the setting consists of

(1) an *alphabet*, A, which is capable of expressing basic logical statements (involving "not," "and," "implies," etc.), the quantifiers $\forall$ and $\exists$ (one of which is unnecessary, if we have a letter for "not"), some arithmetic functions, parenthesis, and a sequence of variables;
(2) a subset $\mathcal{S} \subset A^*$, called *statements*, which is a subset of the words over the alphabet which can be viewed as true/false assertions;
(3) a *theory*, which is a subset of the statements, which might be
    (a) *consistent*, meaning that for each statement $\phi \in \mathcal{S}$ we have that ϕ and its negation do not both lie in $\mathcal{S}$; and

   (b) *complete*, meaning that for each statement $\phi \in \mathcal{S}$ we have that either ϕ or its negation lies in $\mathcal{S}$;

(4) a set of *axioms*, $\Sigma \subset \mathcal{S}$, and a method of deriving a set of *consequences*, $\mathrm{Cn}(\Sigma)$, which are intuitively the "theorems" that result when Σ is taken to be the set of axioms.

One form of Gödel's Incompleteness Theorems say that if the alphabet A is "sufficiently expressive" to express the usual logic and number theory, and $\mathcal{S}$ is defined in the "usual way," then for any *finite* set of axioms, $\Sigma \subset \mathcal{S}$, the theory consisting of $\mathrm{Cn}(\Sigma)$, the consequences of Σ, must be either (1) incomplete, or (2) inconsistent. The basic approach of Gödel here is to build a statement, S, which expresses the fact that "S is a statement that is not provably true." Furthermore, this is also true if the set of axioms is infinite but can be listed as an infinite sequence by a Turing machine.

We refer to [End01] for the details of the above setting and theorems, although below we will roughly describe what this looks like.

First we begin with an alphabet. Here we could take the alphabet to be

$$A_{\mathrm{NumThy}} = \{(,),\neg,\Rightarrow,=,\forall,\exists,\wedge,<,0,1,+,\times\} \cup \{v_1, v_2, \ldots\}$$

This alphabet is rather limited, but will suffice for our purposes. The rough idea is to express statements about the natural numbers

$$\mathcal{N} = \{0, 1, 2, \ldots\}$$

as strings in A_{NumThy}, where A_{NumThy} strings are interpreted in their usual sense ($\vee$ is the logical "or," $\wedge$ the "and," and $v_1, v_2, \ldots$ are variables at our disposal).

For example, one can assert that for every $v_1 \in \mathcal{N}$, there is a prime number greater than v_1 (this is one way of saying that there are infinitely many primes, and this avoids the need to use concepts in set theory such as "the set of prime numbers"); one can write this as:

> For any $v_1 \in \mathcal{N}$ there is a $v_2 \in \mathcal{N}$ such that $v_2 > v_1$ and for any $v_3, v_4 \in \mathcal{N}$, if $v_3 \times v_4 = v_2$ then either $v_3 = 1$ or $v_4 = 1$.

Now let us express this statement in the above alphabet; we may write

$$\text{For any } v_1 \in \mathcal{N} \qquad \text{as} \qquad \forall v_1$$

understanding that $\forall$ is an abbreviation for "for all natural numbers;" as another abbreviation, we never write the phrase "such that," as it can be inferred in context; we will write

$$\text{if } v_3 \times v_4 = v_2 \text{ then either } v_3 = 1 \text{ or } v_4 = 1$$

in its logically equivalent form

$$(v_3 \times v_4 = v_2) \Rightarrow (v_3 = 1) \wedge (v_4 = 1).$$

Hence we can write the above quoted assertion as

$$\forall v_1 \exists v_2 ((v_2 > v_1) \wedge (\forall v_3 \forall v_4 (v_3 \times v_4 = v_2) \Rightarrow (v_3 = 1) \wedge (v_4 = 1)))$$

Of course, many strings in A_{NumThy} aren't statements, for various reasons: for example $\forall + \exists$ is not meaningful, and $\neg(v_1 = 1+1)$ is an assertion that depends on v_1 (which is a "free" or "unbound" variable). However, one can give rules to describe a distinguished subset, $\mathcal{S}$, of A^*_{NumThy} that represents statements to be true or false. This requires some technical details (see [End01]), although it would yield what you expect. We call $\mathcal{S}$ the set of *statements*, and in the literature $\mathcal{S}$ has other names, such as the set of "well-formed formulas without unbounded variables." The set of statements, $\mathcal{S}$, has various properties that you would expect, such as if $\phi \in \mathcal{S}$, then the string $\neg(\phi)$ (or simply $\neg\phi$) also lies in $\mathcal{S}$; i.e., if ϕ is a statement, then so is its negation.

There are a number of variants of the above alphabet; for example in [End01] one uses S to mean "the successor," so that $S0$ refers to 1, $SS0$ refers to 2, etc. One could, of course, eliminate one of $\vee$ or $\wedge$ by De Morgan's laws.

The basic idea of Gödel is that in A_{NumThy}, if we interpret statements in $\mathcal{S}$ as referring to statements about the natural numbers (where the letters of A_{NumThy} have their usual meaning), then for any finite set $\Sigma \subset \mathcal{S}$, one can build a statement, $S = S(\Sigma)$, which says that

> "We have $S \notin \mathrm{Cn}(\Sigma)$, i.e., the statement S has no proof that it is true based on taking the set Σ as our axioms."

So let $\mathcal{T} \subset \mathcal{S}$ be the statements that are true when interpreted as statements in number theory. It is not hard to see that all statements of $\mathcal{S}$ (when build in the "usual" way) are either true or false. It follows that S is either true or false, and it is easy to see that this implies that $\mathrm{Cn}(\Sigma)$ is either incomplete or inconsistent. (See Exercise 7.1.4.)

Remark A.1. Since $\mathcal{T} \subset \mathcal{S}$ is the set of statements that are true in number theory, it is not hard to see that $\mathcal{T}$ is consistent and complete. Hence we can set $\Sigma = \mathcal{T}$ to be our set of axioms, and then $\mathrm{Cn}(\mathcal{T})$, the consequences of $\mathcal{T}$, is just $\mathcal{T}$ itself. So if we take $\mathcal{T}$ to be the "axioms" of our theory, we get a complete and consistent theory. The problem is that there is not easy way to generate all of the axioms $\mathcal{T}$ with a Turing machine; so not only is $\mathcal{T}$ an infinite set (it is, however, countable, assuming that the alphabet is countable), but it is hard to describe. For example, any way of generating $\mathcal{T}$ would have to figure out some difficult statements in number theory, such as

$$\forall v_1 \exists v_2 ((v_2 > v_1) \wedge (\forall v_3 \forall v_4 ((v_3 \times v_4 = v_2) \vee (v_3 \times v_4 = v_2 + 1 + 1)) \Rightarrow (v_3 = 1) \wedge (v_4 = 1)))$$

(i.e., the twin prime conjecture).

A.2. **False Proofs of Fermat's Last Theorem.** Imagine that you or someone else claims to have a proof of that for any positive integers x, y, z, n with $n \geq 3$ one cannot have

$$x^n + y^n = z^n.$$

Say that you suspect that the claimed proof is not correct, but this claimed proof involves some tedious calculations.

Here is one way to refute some of the many false proofs of the above "Fermat's Last Theorem." If the proof does not make use of the fact that x, y, z are integers, and would work even for x, y, z being any positive real numbers, then the proof must be false: indeed, there are many solutions to the above equation for integers $n \geq 3$ and positive real numbers x, y, z.

The above is a commonly used idea in logic. Let's explore this a bit more.

A.3. **The Baker-Gill-Soloway Theorem.** Will be discussed in our coverage of Chapter 9 of Sipser's textbook.

References

[End01] Herbert B. Enderton, *A mathematical introduction to logic*, second ed., Harcourt/Academic Press, Burlington, MA, 2001. MR 1801397 (2001h:03001)

Department of Computer Science, University of British Columbia, Vancouver, BC V6T 1Z4, CANADA, and Department of Mathematics, University of British Columbia, Vancouver, BC V6T 1Z2, CANADA.

E-mail address: `jf@cs.ubc.ca or jf@math.ubc.ca`

URL: `http://www.math.ubc.ca/~jf`