Computability Theory

This section is partly inspired by the material in "A Course in Mathematical Logic" by Bell and Machover, Chap 6, sections 1-10.
Other references: "Introduction to the theory of computation" by Michael Sipser, and "Computability, Complexity, and Languages" by M. Davis and E. Weyuker.

Our first goal is to give a formal definition for what it means for a function on $\mathbb{N}$ to be computable by an algorithm. Historically the first convincing such definition was given by Alan Turing in 1936, in his paper which introduced what we now call Turing machines. Slightly before Turing, Alonzo Church gave a definition based on his lambda calculus. About the same time Gödel, Herbrand, and Kleene developed definitions based on recursion schemes. Fortunately all of these definitions are equivalent, and each of many other definitions proposed later are also equivalent to Turing's definition. This has lead to the general belief that these definitions have got it right, and this assertion is roughly what we now call "Church's Thesis".

A natural definition of computable function f on $\mathbb{N}$ allows for the possibility that $f(x)$ may not be defined for all $x \in \mathbb{N}$, because algorithms do not always halt. Thus we will use the symbol ∞ to mean "undefined".

Definition: A *partial function* is a function

$$f : (\mathbb{N} \cup \{\infty\})^n \to \mathbb{N} \cup \{\infty\}, n \geq 0$$

such that $f(c_1, ..., c_n) = \infty$ if some $c_i = \infty$.

In the context of computability theory, whenever we refer to a function on $\mathbb{N}$, we mean a partial function in the above sense.

Definitions:

$$\text{Domain}(f) = \{\vec{x} \in \mathbb{N}^n \mid f(\vec{x}) \neq \infty\}$$

where $\vec{x} = (x_1 \cdots x_n)$ We say f is *total* iff $\text{Domain}(f) = \mathbb{N}^n$ (i.e. if f is always defined when all its arguments are defined).

Turing machines

A Turing Machine is specified by a 7-tuple, $M = \{Q, \Sigma, \Gamma, \delta, q_1, B, \{q_2\}\}$, where $Q = \{q_1, q_2, \ldots, q_k\}$ is a finite set of states, Σ is a finite input alphabet, including the two elements 0 and 1; Γ is a finite tape alphabet, such that $\Sigma \subseteq \Gamma$; q_1 is the designated start state, q_2 is the designated halt state, and $B \in \Gamma$ is a special blank symbol not in Σ. Lastly, the transition function, δ, is a function from $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$.

Let Σ be a finite alphabet. Σ^* denotes the set of all finite length strings over Σ. Let $x \in \Sigma^*$. We visualize a Turing machine M over Σ on input x as consisting of an tape consisting of an infinite sequence of *cells*, $c_0, c_1, \ldots,$. There is a tape *head* which points to one cell of the tape, and at every point in time, the Turing machine is in one state $q_i \in Q$. Each cell of the tape contains an element from Γ. M on input x operates as follows. Initially M is in the start state q_1, and the infinite tape consists of x written on the first $|x|$ consecutive cells, followed by all blank symbols (B). At every time step, M makes one transition, according to δ. If M is in state q and the tape head is currently reading symbol s at time t, and $\delta(q, s) = (q', s', L/R)$, then at time $t + 1$, the new state is q', the symbol s is replaced by s', and the tape head moves left/right one cell. (If the tape head is already at the leftmost position, then on a L move, the head stays in place.) The computation terminates if the current state of M is the halt state, q_2. When M halts, define y to be the shortest string such that the contents of the tape is yB^* (y followed by all blank symbols). Then M on input x outputs y.

Turing machines compute n-ary partial (or total) functions from $\mathbb{N}^n$ to $\mathbb{N}$ by encoding the input tuples and outputs as strings over Σ. First, we will assume that each number in $\mathbb{N}$ is written in binary. We will encode an n-tuple $a_1, \ldots, a_n$ by a string x where x consists of the concatenation of $a_1, \ldots, a_n$ (written in binary), with each a_i separated by the symbol "2". Let $< a_1, \ldots, a_n >$ denote the encoding of $a_1, \ldots, a_n$. Let f be an n-ary total function from $\mathbb{N}^n$ to $\mathbb{N}$. M computes the function f if for every n-tuple $a_1, \ldots, a_n$, M on input $< a_1, \ldots, a_n >$ outputs $y = f(a_1..a_n)$ (again written in binary). In such a case, f is called a *total computable* function.

If f is a partial function from $\mathbb{N}^n$ to $\mathbb{N}$, then M computes f if for all tuples $a_1, \ldots, a_n$ in the domain of f, M on input $< a_1, \ldots, a_n >$ outputs $f(a_1, \ldots, a_n)$. Note that on inputs not in the domain of f, M may not halt. In such a case, f is called a *computable partial* function.

Our form of **Church's Thesis:**

Every algorithmically computable function is TM-computable.

Here the notion "algorithmically computable" is not a precise mathematical notion, but rather an intuitive notion. It is understood that the algorithms in question have unlimited memory. In the case of register machines, this means that each register can hold an arbitrarily large natural number. It is a strong claim about the robustness of our formal notion of computable function. In general, if we give an informal algorithm to compute a function, then we can claim that it is computable, by Church's Thesis.

Alonzo Church proclaimed this famous "thesis" in a footnote to a paper in 1936. Actually, he did not talk about TM's, but rather claimed that every algorithmically computable function is definable using the λ-calculus which he had invented. A little later Alan Turing published his famous paper defining what are now called Turing machines, and argued, more convincingly than Church, that every algorithmically computable function is computable on a Turing machine. (Hence "Church's Thesis" is sometimes called the "Church-Turing thesis".) Turing proved that Church's λ-definable functions coincide with the Turing computable functions. In fact, many other formalisms for defining algorithmically computable functions have been given, and all of them turn out to be equivalent. This robustness is a powerful argument in favour of Church's thesis.

Exercise 1 *Write Turing machine programs to compute each of the following functions:*

$f_1(x) = x + 1$
$f_2(x, y) = x \cdot y$

Be sure to respect our input/output conventions for TM's.

Let $R \subseteq \mathbb{N}^n$. Thus R is an n-ary relation (predicate). We will think of R as a total 0-1 valued function as follows: $R(\vec{x}) = 0$ if and only if $\vec{x} \in R$, and $R(\vec{x}) = 1$ otherwise.

Configurations

We can describe the computation of M on x at time t by a *configuration.* Let m be the leftmost tape cell such that all cells to the right of q contain the blank symbol, B. Then the configuration consists of the contents of the tape up to cell m, plus the current state, q, plus the location, i, of the tape head. We will represent this information by the string $s_1, s_2, ..., (q, s_i), s_{i+1}, \ldots, s_m$, where $s_1, \ldots, s_q$ are the contents of the tape cells up to cell m, and where q is the current state. (The location of q within the string tells us the location of the head.) The initial configuation of M on x is therefore the string $(q_1, x_1), x_2, \ldots, x_{n-1}, x_n$.

The computation of M on x is described by a sequence of configurations, $c_1, c_2, \ldots$, where c_1 is the initial configuration of M on x, and such that each c_i follows from the previous configuration c_{i-1} by applying the transition function δ. If M halts on x, then this sequence of configurations is finite. If M halts on x in m steps, we will visualize the corresponding sequence of configurations as a *tableaux*, or $m - by - m$ matrix, where the first row of the matrix is the start configuration, and row i is the configuration at time t.

Encoding Turing Machines

We want to associate a number with each Turing machine over the input alphabet $\Sigma = \{0, 1, 2\}$. Here is one way to do this.

Our convention is that the states of a TM are always called $Q = \{q_1, q_2, \ldots, q_n\}$, where q_1 is always the start state, and q_2 is always the halt state. Similarly, assume that the tape symbols are $\Gamma = \{x_1, x_2, \ldots, x_k\}$ where $x_1 = 0$, $x_2 = 1$, $x_3 = 2$ and $x_4 = B$. Let "left" be denoted by D_1 and let "right" be denoted by D_2. Then we represent the transition $\delta(q_i, x_j) = (q_k, x_l, D_m)$ by $0^i 1 0^j 1 0^k 1 0^l 1 0^m$. The code for M is: $111 code_1 11 code_2 11 \ldots 11 code_r 111$, where $code_i$ is the code for one of the possible transitions.

Example Let $M = (Q = \{q_1, q_2, q_3\}, \Sigma = \{0, 1\}, \Gamma = \{0, 1, B\}, \delta, q_1, B, \{q_2\})$. Below we describe the transitions and their corresponding codes:

- $\delta(q_1, 1) = (q_3, 0, R)$, $c_1 = 0^1 1 0^2 1 0^3 1 0 1 0^2$

- $\delta(q_3, 0) = (q_1, 1, R)$, $c_2 = 0^3101010^210^2$
- $\delta(q_3, 1) = (q_2, 0, R)$, $c_3 = 0^310^210^21010^2$
- $\delta(q_3, B) = (q_3, 1, L)$, $c_4 = 0^310^410^310^210$

Thus M is encoded by the string $111c_111c_211c_311c_4111$, and the pair $(M, 1011)$ is encoded by the string $111c_111c_211c_311c_41111011$. Note that given an encoding $\#(M, w)$, we can recover the associated Turing machine M and string w.

A Universal Turing Machine

A Universal Turing function U takes as input a number $\#(M, x)$ where M is a Turing machine. U on input $\#(M, x)$ outputs y if M on input x outputs y. Note that if M does not halt on x, then U is undefined on x.

U is a computable function.

We will describe a 3-tape TM, M_U. It is a well-known result that a k-tape Turing machine can be simulated by a 1-tape Turing machine, and thus U is a computable relation. First M_U checks to see if $\#(M, x)$ is a valid encoding of a Turing machine, M, and if x is of the proper format. If not, then M_U does not halt. Otherwise, on input $\#(M, x)$, M_U will simulate the computation of M on x. The first tape of M_U contains the code of M. The second tape of M_U will contain the contents of the tape of TM M as it is being simulated on input x. The third tape of M_U will contain state information.

- Initially, $\#(M, x)$ is on tape 1.
- Check tape 1 to make sure it is a valid input. That is, valid codes for M begin and end with "111" and each code $code_i$ is separated by 11, and the transitions are of the form $0^i10^j10^k10^l10^m$, where $m = 1$ or 2.
- Initialize tape 2 to contain $\$x$.
- Initialize tape 3 to contain \$0 (this is the start state, q_1, in unary).
- Initialize tape 1 to hold $11\$code_111code_211\ldots11code_r111$. (a) If tape 3 holds \$000 (halt state q_2), halt and output y, where y is the output of M when it halts; (b) otherwise simulate the next step as follows. Let x_j be the symbol scanned by head 2 and let 0^i be the contents of tape 3. Scan tape 1 from \$ to 111, looking for a string beginning with 110^i10^j1. If no such string is found, halt and output 0 (reject). If such a string is found, say it is $110^i10^j10^k10^l10^m$. Put 0^k on tape 3, and write x_l on the tape cell scanned by head 2, and then move that head in direction D_m one cell.

Notation: $\{z\}$ = the program $\mathcal{P}$ s.t. $\#(\mathcal{P}) = \hat{z}$

$$\text{Thus } \{z\} = \begin{cases} \text{the program } P \text{ such that } \#(P) = z \text{ if } P \text{ exists} \\ \Lambda \text{ (empty program) otherwise} \end{cases}$$

$\{z\}_n$ is the n-ary function computed by the program $\{z\}$.

Recursive and Recursively Enumerable Sets

Recursive Sets

For this section, a *set* means a subset of $\mathbb{N}^n$, where usually $n = 1$. Thus formally a set is the same thing as a relation, which is the same as a total 0-1 valued function. Thus if $A \subseteq \mathbb{N}^n$, then we write

$$A(\vec{x}) = \begin{cases} \textit{true} & \text{if } \vec{x} \in A \\ \textit{false} & \text{otherwise} \end{cases}$$

Our convention will usually be that true=1 and false=0.

Definition: A set (or relation) is *recursive* (or *computable* or *decidable*) if it is computable as a total 0-1 valued function.

By Church's thesis, a set A is recursive iff there is an algorithm which, given $\vec{x}$, determines whether $\vec{x} \in A$. (The algorithm must halt on all inputs.)

Proposition: The class of recursive subsets of $\mathbb{N}^n$ is closed under the operations $\cup, \cap$, complement.

Proof: This is the same as saying that the class of recursive n-ary relations is closed under the Boolean operations $\wedge, \vee, \neg$ (see Lemma, page 62). □

Proposition: If $R(\vec{x}, y)$ is a recursive relation, and $f(\vec{x})$ is a total computable function, then the relation $S(\vec{x}) = R(\vec{x}, f(\vec{x}))$ is a recursive relation.

Proof:. The class of total computable functions is closed under composition. □

Note that the assumption that f is total is necessary in the above proposition, since by definition a recursive relation must be a **total** 0-1 valued function.

We are interested in proving that certain sets are *not* recursive. The standard example is the diagonal halting set K. K takes as input the encoding of a Turing Machine, and accepts if and only if that TM, on its own encoding as input, halts.

Notation: $K = \{x \mid \{x\}_1(x) \neq \infty\}$

Recall that $\{x\}_1$ is the unary function computed by the program (coded by) x . Thus

$$K(x) = \begin{cases} \text{true if program } x \text{ halts on input } x \\ \text{false otherwise} \end{cases}$$

Note that K is a version of the famous "halting problem", originally formulated by Alan Turing in the context of Turing machines.

Theorem: K is not recursive.

Proof: The proof is a combination of a "diagonal argument" and a reduction. First the diagonal argument.

Recall that $\phi_n(x) = \{n\}_1(x)$ for $n = 0, 1, 2, ...$ That is, ϕ_n is the (partial) function of one variable computed by program $\{n\}$. Thus $\phi_0, \phi_1, ...$ is an enumeration of all computable functions of one variable...so we are enumerating just those TMs that on every input, either halt and output 0 or 1, or do not halt. This effectively is an enumeration of all univariate relations that are computable by TMs. We can list all values of all these functions in an infinite table. The n-th row will correspond to a list of the successive values $\phi_n(0), \phi_n(1), ...$ of the function ϕ_n. We will label the entry (n, i) by the value output if ϕ_n on input i halts, and by ∞ if ϕ_n on input i does not halt.

Define the function D, called the "diagonal function" as follows. $D(x)$ will be 1 (accept) if program x halts and rejects x, or does not halt on x; and $D(x)$ will be 0 (reject) if program x halts and accepts x.

The list of values of $D(0), D(1), ...$ can be obtained by going down the main diagonal of the above table and changing any value that is not 1 to 1 (in particular all ∞'s will be changed to 1 and all 0's will be changed to 1.) and changing all values that are 1 to 0. Thus it is clear that this list of values cannot coincide completely with any row in the table, because the n-th value in the list disagrees with the n-th row at position n. It follows that K is not a computable function.

Now comes the reduction: We can reduce the computation of D to the computation of K, so that if K is recursive then D is also recursive. That is, assume (for sake of contradiction) that K is recursive. We will use the program for K to solve D as follows. On input x, we will first simulate K on x to determine if TM x run on x halts or not. If it does not halt, then we halt and accept (output 1), and otherwise if it does halt, then we simulate TM x and output the opposite answer. Thus, we have shown that if K is computable, then D is computable. But this is a contradiction since we already have proven that D is not computable. Thus K is also not computable.

Reducibility

Definition: Suppose $A, B \subseteq \mathbb{N}$. Then $A \leq_m B$ (A is many-one reducible to B) iff there is a total recursive function $f : \mathbb{N} \rightarrow \mathbb{N}$, such that $x \in A \Leftrightarrow f(x) \in B$, for all $x \in \mathbb{N}$.

Note that $\leq_m$ is similar to the notion of $\leq_p$ of polynomial time reducibility. The difference is that for the latter we require that the function f be polynomial time computable.

Proposition: The relation $\leq_m$ is transitive. That is, if $A \leq_m B$ and $B \leq_m C$ then $A \leq_m C$.

Exercise 2 *Prove the above proposition.*

Proposition: If $A \leq_m B$ and B is recursive then A is recursive

Proof: $A(x) = B(f(x))$. □

Application: To show that B is not recursive, it suffices to show that $K \leq_m B$.

Example: the famous Halting Problem, HALT. The input is the encoding of a pair $< x, y >$ which should be accepted if and only if TM x halts on input y. HALT is not recursive. To see this, we will show that K reduces to HALT. Suppose for sake of contradiction that HALT is recursive. Given an input x for K, we simulate HALT on the pair $< x, x >$ and accept if and only if HALT accepts.

Example: Let $H = \{x \mid \{x\}_1(0) \neq \infty\}$ Thus $x \in H$ iff program $\{x\}$ halts on input 0.

Claim: H is not recursive

Proof: It suffices to show $K \leq_m H$. Thus we want a total computable f so $x \in K$ iff $f(x) \in H$. That is, $\{x\}_1(x) \neq \infty$ iff $\{f(x)\}_1(0) \neq \infty$

What is the program $\{f(x)\}$? Program $\{f(x)\}$ on any input y simulates program $\{x\}$ on input x.

From the point of view of the program $\{f(x)\}$, x is a constant; say $x = x_0$. Since there is an easy algorithm that transforms x_0 to the program $\{f(x_0)\}$, it follows from Church's thesis that the function f is computable (i.e. recursive).

NOTE: In order to prove that a set is not recursive, one can relax the notion of a reduction to a Turing reduction. To show that A is Turing reducible to B, written $A \leq_T B$, we assume that we have a Turing Machine M_B that always halts and accepts exactly B. Then we design another Turing machine, T_A that should always halt and accept exactly A¿ T_A may use T_B as a subroutine. Note that our reduction above, showing that D reduces to K was a Turing reduction. A many-one reduction, $A \leq_m B$ is a special kind of reduction where T_A can only call T_B once, on $f(x)$, and T_A must accept x if and only if T_B accepts $f(x)$. We will see later that many-one reductions are necessary in order to prove that certain sets are not r.e. But to prove that a set is not recursive, it is ok to use the more general notion of a Turing reduction.

Exercise 3 *Show that the following sets are not computable. Note that it suffices to show that the complementary set is not computable, since a set A is computable iff A^c is computable.*

$A_1 = \{x \mid \{x\}_1(5) = \infty\}$
$A_2 = \{x \mid \text{ran}(\{x\}_1) = \mathbb{N}\}$, where $\text{ran}(f) = f(\mathbb{N}) =$ range of f
$A_3 = \{x \mid \text{dom}(\{x\}_1) \text{ is finite}\}$, where $\text{dom}(f) = \{x \mid f(x) \neq \infty\}$ is the domain of f.

Rice's Theorem

It turns out that the noncomputability of all of the above examples, and many more, follow from a single result, known as Rice's Theorem. We say that $A \subseteq \mathbb{N}$ is a *function index set* if for all $e \in A$, if $\{e\}_1 = \{e'\}_1$ then $e' \in A$. Thus if A contains a code for a program that computes a unary function ϕ, then A must contain codes for all programs that compute ϕ. We can think of a function index set as a set of computable functions rather than a set of numbers.

Note that each of the three sets A_1, A_2, A_3 in the above exercise is a function index set.

Theorem: (Rice) If A is a function index set and $A \neq \varnothing$ and $A \neq \mathbb{N}$ then A is not computable.

Exercise 4 *Prove Rice's Theorem. Use the same techniques that you used to prove A_1, A_2, A_3 are not computable. (Hint: First consider the case in which no code for the empty function Empt (which has empty domain) is in A.)*

Exercise 5 *You may use Church's Thesis in answering the following questions. That is, to justify that a particular function is computable is suffices to give an algorithm for computing it.*

Define the relation $R(x, y)$ by the condition $R(x, y)$ holds iff at some time during the computation of program $\{x\}$ on input 0, the first $|y|$ symbols of the tape contain y. Prove that $R(x, y)$ is not recursive.

Recursively Enumerable Sets

Definition: If $A \subseteq \mathbb{N}^n$ then A is r.e. (*recursively enumerable (r.e.)*, or *semidecidable* or *semidecidable*) if there exists a recursive relation $R \subseteq \mathbb{N}^{n+1}$ such that

$$\vec{x} \in A \Leftrightarrow \exists y R(\vec{x}, y), \quad \text{for all } \vec{x} \in \mathbb{N}^n$$

Intuition: Let $n = 1$. A is r.e. iff there is an algorithm for enumerating members of A in some order. The following Lemma justifies this intuition.

Intuition: Recall that A is recursive (decidable) if there is a TM M that always halts and such that for all $x \in A$, M halts and outputs 1 (accepts), and for all $x \notin A$, M halts and outputs 0 (rejects). In contrast, A is r.e. if there is a TM M such that for all $x \in M$, M halts and outputs 1 (accepts), and for all $x \notin M$, M either does not halt, or halts and does not output 1 (halts and does not accept). Thus any A that is recursive is also r.e., but the converse does not necessarily hold.

Recall the diagonal language D that we defined previously. (x is accepted if and only if TM x when run on x does not halt, or halts and rejects x. Our "diagonal" argument from the previous section actually proves the stronger result that D is not r.e. Do you see why? Since we enumerated all Turing Machines (and not just those that halted on all inputs), we constructed a language that was different from *every* language accepted by a TM.

Lemma: If $A \subseteq \mathbb{N}$ then A is r.e. iff $A = \varnothing$ or $A = \text{ran}(f)$ for some total computable $f : \mathbb{N} \to \mathbb{N}$.

If $A = \text{ran}(f)$, then $A = \{f(0), f(1), f(2) \cdots\}$. Hence there is an algorithm for enumerating A, namely compute $f(0), f(1), f(2) \cdots$. It is important that f be total in order for this algorithm to work. Notice that this does not necessarily enumerate A in order, and there may be repetitions.

Proof of Lemma: $\Rightarrow$: Suppose $x \in A \Leftrightarrow \exists y R(x, y)$. We want a total computable f so $A = \text{ran}(f)$. Idea: Enumerate all pairs $\langle x, y\rangle$. We may assume $A \neq \varnothing$, so let $a \in A$. First we define a total computable function $F(x, y)$ of two variables whose range is A:

$$F(x,y) = \begin{cases} x & \text{if } R(x,y) \text{ holds} \\ a & \text{otherwise} \end{cases}$$

Then $A = \text{ran}(F)$. To convert F to a unary function f with the same range, we fix things so that $f(2^x 3^y) = F(x, y)$. Explicitly, define $f(z) = F((z)_0, (z)_1)$, where $(z)_x$ is the exponent of prime p_x in the prime decomposition of z Thus f is a total computable unary function whose range is A.

Proof of direction $\Leftarrow$: Suppose $A = \text{ran}(f)$, where $f : \mathbb{N} \to \mathbb{N}$ is a total computable function. Define the relation $R(x, y)$ by

$$R(x, y) = (x = f(y))$$

Then $R(x, y)$ is recursive. Now it is clear that

$$x \in A \Leftrightarrow \exists y(x = f(y)) \Leftrightarrow \exists y R(x, y) \qquad \Box$$

The technique used in the first half of the above proof of enumerating A by, in effect, enumerating all pairs (x, y) is called *dovetailing*.

Remark: Every recursive set is r.e. Given a recursive set A, simply define the relation R by $R(x, y) \Leftrightarrow x \in A$. Then $x \in A \Leftrightarrow \exists y R(x, y)$, so A is r.e.

The converse is false, as we shall soon see.

Analogy: P is to NP as the recursive sets are to the r.e. sets. In fact, one way to define NP is to modify our definition of r.e. by requiring the relation $R(x, y)$ be polynomial time computable (instead of just recursive), and by putting a suitable bound on the quantifier $\exists y R(x, y)$. Then P is a subset of NP just as every recursive set is r.e. However, unlike P vs NP we can prove that not all r.e. sets are recursive.

Theorem: K is r.e but not recursive.

Proof: Recall that $K = \{x \mid \{x\}_1(x) \neq \infty\}$. We have already shown that K is not recursive, so it suffices to show that K is r.e. We will modify our Universal Turing machine, M_U, as follows. On input x, we construct $\{x\}$, the program encoded by x, and simulate $\{x\}$ on input x. If the simulation halts and outputs 1, then we halt and output 1; if the simulation halts and does not output 1, then we also halt and output 0, otherwise, if the simulation never halts, then our program also never halts.

Exercise 6 *We say that a function $f : \mathbb{N} \to \mathbb{N}$ is* nondecreasing *if*

$$x \le y \Rightarrow f(x) \le f(y), \text{ for all } x, y \in \mathbb{N}$$

Prove that a set $A \subseteq \mathbb{N}$ is recursive iff $A = \varnothing$ or A is the range of some total computable unary nondecreasing function f. Give a careful proof, without using Church's thesis. **Hint:** *For the $\Longleftarrow$ direction, consider separately the case in which A is finite.*

Definition: If $f(\vec{x})$ is a (partial) function, then graph(f) is the relation

$$R_f(\vec{x}, y) = (y = f(\vec{x}))$$

If f is a total computable function, then graph(f) is a recursive relation, since in general the substitution of a total computable function into a recursive relation (in this case the relation $(y = z)$ is always a recursive relation (by the second Proposition, page 58). However, if f is computable but not total, then graph(f) is not necessarily recursive. As an example, let $f(x) = 0$ if program $\{x\}$ halts on input x, and otherwise $f(x)$ is undefined. Thus

$$x \in K \Leftrightarrow (x, 0) \in \text{graph}(f)$$

Thus graph(f) is not recursive, since otherwise K would be recursive.

Although graph(f) is not always recursive for computable functions f, it is r.e. In fact, there is a converse:

Theorem: Suppose f is a (partial) n-ary function. Then f is computable iff graph(f) is recursively enumerable.

Exercise 7 *Prove the above theorem.*

Theorem A is recursive iff both A and A^c are r.e.

Proof: $\Rightarrow$: Recursive sets are r.e., and complements of recursive sets are recursive.

$\Leftarrow$: Assume A and A^c are both r.e. Then there are recursive relations R and S such that $x \in A$ iff $\exists y R(x, y)$ and $x \in A^c$ iff $\exists y S(x, y)$.

Here is a decision procedure to determine whether $x \in A$:

```
for $y : 0 \cdots \infty$
    If $R(x, y)$ then output yes $(x \in A)$ exit
    end if
    If $S(x, y)$ then output no $(x \notin A)$ exit
    end if
end for
```

We know this terminates.

Application: K is r.e. but not recursive. Therefore by the above theorem, K^c is *not* r.e.

Proposition: Suppose $A, B \subseteq \mathbb{N}$. If $A \leq_m B$ and B is r.e. then A is r.e.

Exercise 8 *Prove the above proposition.*

Application: To show A is not r.e., it suffices to show $K^c \leq_m A$.

Exercise 9 *Show that the following sets are not r.e. Note that*

$$A \leq_m B \Leftrightarrow A^c \leq_m B^c$$

$A_1 = \{x \mid \{x\}_1(5) = \infty\}$
$A_2 = \{x \mid ran(\{x\}_1) = \mathbb{N}\}$
$A_3 = \{x \mid dom(\{x\}_1) \textit{ is finite}\}$

Also show that A_2^c and A_3^c are not r.e. In fact, it is easier to show A_2^c is not r.e. than to show A_2 is not r.e. To show A_2 and A_3^c are not r.e. use the method suggested above (reduce K^c to them).

r.e. completeness: We say that a set $A \subseteq \mathbb{N}$ is *r.e. complete* iff

(i) A is r.e., and
(ii) for every set $B \subseteq \mathbb{N}$, if B is r.e. then $B \leq_m A$.

The notion of NP-completeness was taken from the above definition.

It turns out that every "natural" r.e. set $A \subseteq \mathbb{N}$ that has been shown to be not recursive is in fact r.e. complete.

Exercise 10 *Show that K is r.e. complete.*

Undecidable combinatorial problems

So far all of our examples of nonrecursive sets have referred directly or indirectly to programs, as for example the set K. However there are many known nonrecursive sets which arrive from combinatorial problems which on the surface appear to have nothing to do with computation. An example is the set TG of all context-free grammars G over some alphabet Σ such that $L(G) = \Sigma^*$. (Technically TG consists of all numerical codes for such grammars G, where we assign a numerical code to a grammar in the same way as we assigned codes to RM programs.) The method for proving that TG is nonrecursive is the same as for examples above; namely reduce K^c to TG. See for example "Elements of the Theory of Computation" by H. R. Lewis and C. H. Papadimitriou or "Formal Languages and their Relation to Automata" by J. E. Hopcroft and J. D. Ullman for this and other examples.

The crowning achievement for showing sets are not recursive is the following.

Hilbert's 10th Problem (posed 1900, solved 1970)

Hilbert's problem: Find a procedure to determine whether a Diophantine equation $p(\vec{x}) = q(\vec{x})$ has a solution in $\mathbb{N}$.

Definition: A *Diophantine equation* is one of the form $p(\vec{x}) = q(\vec{x})$, where p and q are multivariate polynomials with natural number coefficients.

Examples are $3x^3yz^5 + 2y^4 + 5 = 0$, and $(x+1)^n + (y+1)^n = z^n$, for any fixed positive integer n.

Definition: A *Diophantine* relation $R(\vec{x})$ is one of the form

$$\exists y_1 \cdots \exists y_m \ (p(\vec{x}, y, \ldots, y_m) = q(\vec{x}, y_1, \ldots, y_m))$$

where p and q are polynomials as above.

MRDP **Theorem** (1970) Every r.e. set is Diophantine.

Corollary: There is no algorithm for Hilbert's 10th problem.

Proof of Corollary: Choose any set, say K, which is r.e. but not recursive. Since K is r.e., it follows from the MRDP Theorem that K has a representation of the form

$$a \in K \Leftrightarrow \exists y_1 \cdots \exists y_m (p(a, y_1 \cdots y_m) = q(a, y_1 \cdots y_m))$$

If there were an algorithm for Hilbert's 10th, then we could determine membership in K. $\square$

The proof of the MRDP Theorem is beyond the scope of this course. For a readable proof, see "Proof of recursive unsolvability of Hilbert's Tenth Problem" by Jones and Matiyasevich, Amer. Math. Monthly vol. 98 (1991) 689-709.