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

$$A(\vec{x}) = \begin{cases} 0 & \text{if } \vec{x} \in A \\ 1 & \text{otherwise} \end{cases}$$

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

NOTE: The terms "recursive" and "computable" are interchangeable in this section, whether they are applied to functions or sets. This makes sense, since we proved that the computable functions are the same as the recursive functions (see the theorem on page 61 and the corollary to the Kleene Normal Form Theorem on page 69).

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). $\square$

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. $\square$

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.

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

Recall that $\{x\}_1$ is the unary function computed by the program (coded by) x (see page 68). Thus

$$K(x) = \begin{cases} 0 \text{ (true) if program } x \text{ halts on input } x \\ 1 \text{ (false) otherwise} \end{cases}$$

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 (page 69) 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. We can list all values of all these functions in an infinite table, whose n-th row is a list of the successive values $\phi_n(0), \phi_n(1), ...$ of the function ϕ_n. We now define a "diagonal function" $D(x)$ by making $D(n)$ defined iff $\phi_n(n)$ is undefined. That is,

$$D(x) = \begin{cases} 0 \text{ if } x \notin K \\ \infty \text{ if } x \in K \end{cases}$$

The list of values of $D(0), D(1), ...$ can be obtained by going down the main diagonal of the above table and changing each ∞ to 0 and changing each defined value to ∞. 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 D is not a computable function.

More formally, we can prove that D is not computable by contradiction. If D is computable, then $D = \{e\}_1$ for some $e \in \mathbb{N}$. But then

$$\{e\}_1(e) = D(e) = \begin{cases} 0 \text{ if } e \notin K & \text{i.e. } \{e\}_1(e) = \infty \\ \infty \text{ otherwise} & \text{i.e. } \{e\}_1(e) \neq \infty \end{cases}$$

i.e. $\{e\}_1(e)$ is defined iff $\{e\}_1(e)$ is not defined, a contradiction. Hence D is not computable.

Now comes the reduction: We can reduce the computation of D to the computation of K, so that if K is computable then D is computable. But we just showed that D is not computable, so K is not computable.

We can formalize this argument as follows: Assume K is computable. Then $D(x) = \mu y[K(x) = 1]$, so D is computable, contradiction. Therefore K is not computable. $\square$

Corollary: Let $f(x) = \mu y T(x, x, y)$. Then f is a (partial) computable function which has no extension to a total computable function. That is, there is no total computable function $g(x)$ such that $g(x) = f(x)$ for each x such that $f(x) \neq \infty$.

Exercise 1 *Prove the Corollary.*

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 $f : \mathbb{N} \to \mathbb{N}$, 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: $A(x) = B(f(x))$. $\square$

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

Example: Let $H = \{x \mid \{x\}_1(0) \neq \infty\}$ Thus $x \in H$ iff program $\{x\}$ halts when all registers are initially 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 $x = x_0$. The program $\{f(x_0)\}$ is simply

$$R_1 \leftarrow x_0, \{x_0\}'$$

where the first command is an abbreviation for the sequence of $x_0 + 1$ commands

$$R_1 \leftarrow 0, R_1 \leftarrow R_1 + 1, \cdots, R_1 \leftarrow R_1 + 1$$

Thus program $\{f(x_0)\}$ initializes R_1 to x_0, and then acts like program $\{x_0\}$. Here $\{x_0\}'$ is a modified version of program $\{x_0\}$, in which every jump instruction J_{ijk} is changed to $J_{i,j,k+x_0+1}$, because $x_0 + 1$ commands have been inserted in the front of program $\{x_0\}$.

Since there is an easy algorithm that transforms x_0 to the program $\{f(x_0)\}$ described above, it follows from Church's thesis that the function f is computable (i.e. recursive).

Exercise 3 *Show that f above is in fact primitive recursive. Do this by giving an explicit definition of $f(x)$, and use the techniques on pages 61-66.*

It turns out that reductions such as the one above can be made easier by using a special case of the so-called S-m-n theorem.

Theorem: (Special case of the S-m-n theorem)
Let $g(x, y)$ be computable. Then there exists a total computable function $f : \mathbb{N} \to \mathbb{N}$ such that $\{f(x)\}_1(y) = g(x, y)$, for all x, y.

Intuitively this theorem tells us that if $g(x, y)$ is computable, then for each fixed value $x = x_0$ of the first argument, $g(x_0, y)$ becomes a computable function $g_{x_0}(y)$ of y alone. The function $f(x_0)$ tells us the code for the program which computes this function g_{x_0}.

Proof of special case of S-m-n theorem:
For each constant x_0, we want to find a program $\{f(x_0)\}$ such that $\{f(x_0)\}_1(y) = g(x_0, y)$, for all y.

Let $\mathcal{P}$ be a program which computes g. Then the program $\{f(x_0)\}$ expects its argument y in register R_1, so it first copies y into R_2, and then puts the constant x_0 in R_1, and executes P. That is, the program $\{f(x_0)\}$ is

$R_2 \leftarrow R_1$, $R_1 \leftarrow x_0$, $\mathcal{P}'$

where $\mathcal{P}'$ is $\mathcal{P}$ with all its jump instructions incremented by the number of commands we've placed in front of it (namely by $x_0 + 5$, since the copy macro requires 4 commands). (See the argument above.)

Again by Church's Thesis, f is computable. $\Box$

Exercise 4 *Show that f is in fact primitive recursive.*

We will now state the (general) so-called S-m-n (or index) theorem, even though in practice we will only use the above special case.

S-m-n Theorem: For any $m, n \geq 0$ there is a total computable $(m+1)$-ary function S^m_n such that

$$\{S^m_n(z, y_1, ..., y_m)\}_n(x_1, ..., x_n) = \{z\}_{m+n}(x_1, ..., x_n, y_1, ..., y_m)$$

Exercise 5 *Prove the S-m-n theorem for the case $m = 1$ using the special case above and the Kleene Normal Form Theorem. Now use this to prove it for arbitrary m.*

Application: We can use this form of S-m-n to give a slick argument that $K \leq_m H$, where $H = \{x \mid \{x\}_1(0) \neq \infty\}$ is as in the example above. Recall that $\Phi(x, y) = \{x\}_1(y)$ is the universal function for unary computable functions, and Φ is computable, by the Kleene Normal Form Theorem (KNFT). Let

$$g(x, y) = \{x\}_1(x) = \Phi(x, x)$$

By the S-m-n theorem, there is a total computable function f such that

$$\{f(x)\}_1(y) = g(x, y) = \{x\}_1(x), \qquad \text{for all } x, y$$

Hence $\{f(x)\}_1(0) = \{x\}_1(x)$, and in particular $\{f(x)\}_1(0) \neq \infty$ iff $\{x\}_1(x) \neq \infty$. Thus $x \in K$ iff $f(x) \in H$. $\Box$

Exercise 6 *Show that the following sets are not computable, using the S-m-n theorem as above. Note that it suffices to show that the complementary set is not computable, since a set A is computable iff A^c is computable.*

$A_1 = \{x \mid \{x\}_1(5) = \infty\}$
$A_2 = \{x \mid \mathrm{ran}(\{x\}_1) = \mathbb{N}\}$, where $\mathrm{ran}(f) = f(\mathbb{N})$ = range of f
$A_3 = \{x \mid \mathrm{dom}(\{x\}_1) \text{ is finite}\}$, where $\mathrm{dom}(f) = \{x \mid f(x) \neq \infty\}$ 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 7 *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 8 *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.*

(a) Define the relation $R(x, y)$ by the condition $R(x, y)$ holds iff at some time during the computation of program $\{x\}$ (where all registers are initialized to 0) register R_1 obtains the value y. Prove that $R(x, y)$ is not recursive.

(b) Define the relation $S(x, y)$ by the condition $S(x, y)$ holds iff at some time during the computation of program $\{x\}$ (where all registers are initialized to 0) some register obtains the value y. Prove that $S(x, y)$ is *recursive.*

Recursively Enumerable Sets

Definition: If $A \subseteq \mathbb{N}^n$ then A is r.e. (*recursively enumerable*, or *computably enumerable*, 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.

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

If $A = \mathrm{ran}(f)$, 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 $A = \mathrm{ran}(f)$. 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 $A = \mathrm{ran}(F)$. 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 (see Notes page 66). Thus f is a total computable unary function whose range is A.

Proof of direction $\Leftarrow$: Suppose $A = \mathrm{ran}(f)$, 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 because it results from substituting a total computable function $f(y)$ for z in the recursive relation $(x = z)$. (See the second Proposition on page 71.) Now it is clear that

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

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. By the KNFT (page 68), $\{x\}_1(x) = U(\mu y T(x,x,y))$, where $T \equiv T_1$ is the Kleene T predicate. Therefore

$$x \in K \Leftrightarrow \exists y T(x,x,y)$$

Exercise 9 *We say that a function $f : \mathbb{N} \to \mathbb{N}$ is* nondecreasing *if*

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

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.*

Pairing Functions

A pairing function is a one-one function $p : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. A simple example is: $p(x,y) = 2^x 3^y$. However a nicer example is the function J, which provides a bijection from $\mathbb{N} \times \mathbb{N}$ onto $\mathbb{N}$. Define

$$J(x,y) = \frac{1}{2}(x+y)(x+y+1) + x$$

Using the fact that $J(x,y) = (\sum_{i=0}^{x+y} i) + x$, it is not hard to see that J maps pairs of natural numbers to natural numbers according to the following scheme:

$$\begin{array}{cc|cccccc}
y & 3 & 6 & & & & \\
 & 2 & 3 & 7 & & & \\
 & 1 & 1 & 4 & 8 & & \\
 & 0 & 0 & 2 & 5 & 9 & \\
\hline
 & & 0 & 1 & 2 & 3 & x
\end{array}$$

We define the inverses K, L of J to satisfy

$$K(J(x,y)) = x, \ L(J(x,y)) = y, \ J(K(z), L(z)) = z$$

Exercise 10 *Show that J, K, L are each primitive recursive. (In fact, they are polynomial time computable.)*

Characterizing r.e. sets

Recall that by definition, if $A \subseteq \mathbb{N}$, then A is r.e. iff there is a recursive relation $R(x,y)$ such that $x \in A \Leftrightarrow \exists y R(x,y)$.

Theorem: Suppose $A \subseteq \mathbb{N}$. Then the following are equivalent (where dom stands for domain and ran stands for range):

i) A is r.e.
ii) $A = \text{dom}(f)$ for some recursive f.
iii) $A = \{x \mid \exists y R(x,y)\}$ for some primitive recursive relation R.

iv) $A = \varnothing$ or $A = \mathrm{ran}(f)$ for some primitive recursive unary function f.
v) $A = \mathrm{ran}(f)$ for some recursive unary function f (f not necessarily total).

Proof: We show i) $\Rightarrow$ ii) $\Rightarrow$ iii) $\Rightarrow$ iv) $\Rightarrow$ v) $\Rightarrow$ i)

i) $\Rightarrow$ ii): Assume A is r.e., so $x \in A \Leftrightarrow \exists y R(x,y)$, where R is recursive. We want a recursive f so $A = \mathrm{dom}(f)$, i.e. $A = \{x \mid f(x) \text{ is defined}\}$. Simply define $f(x) = \mu y R(x,y)$.

ii) $\rightarrow$ iii): Assume $A = \mathrm{dom}(f)$, where f is computable. We find a primitive recursive relation R such that $A = \{x \mid \exists y R(x,y)\}$. The idea is to use the KNFT. Since f is computable, some program $\{e\}$ computes f. Thus

$$f(x) = \{e\}_1(x) = U(\mu y T(e,x,y))$$

Then $f(x)$ is defined iff $\exists y T(e,x,y)$, so $x \in A \Leftrightarrow \exists y T(e,x,y)$. Let $R(x,y) = T(e,x,y)$.

iii) $\Rightarrow$ iv): Assume $x \in A \Leftrightarrow \exists y R(x,y)$, where R is prim rec. Assume $A \neq \varnothing$ so $a \in A$. We want to find a primitive recursive function f whose range is A. This is easy if we allow F to be a function of two variables:

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

To turn F into a function of one variable, we could use the technique used in the proof of the Lemma two pages ago, but we choose to use the inverse pairing functions K and L (defined above).

$$f(z) = \begin{cases} K(z) & \text{if } R(K(z), L(z)) \\ a & \text{otherwise} \end{cases}$$

Then f is a unary primitive recursive function whose range is A.

iv) $\Rightarrow$ v): Assume A is $\varnothing$ or $A = \mathrm{ran}(f)$ where f is prim rec. Observe that the empty function is recursive, and its range is the empty set.

v) $\Rightarrow$ i): Assume $A = \mathrm{ran}(f)$, where f is recursive. We want a recursive relation $R(x,y)$ so $x \in A \Leftrightarrow \exists y R(x,y)$, that is $x \in \mathrm{ran}(f) \Leftrightarrow \exists y R(x,y)$. Note that $x \in \mathrm{ran}(f) \Leftrightarrow \exists y (f(y) = x)$, so can we take $R(x,y)$ to be the relation $(f(y) = x)$? Unfortunately this relation may not be recursive, because $f(y)$ may be undefined. Kleene comes to the rescue: For some $e \in \mathbb{N}$

$$f(x) = \{e\}_1(x) = U(\mu y T(e,x,y)) \tag{1}$$

Remember that by definition of the T-predicate, $T(e,x,y)$ holds iff y codes a halting computation of program $\{e\}$ on input x. Note that by definition of the T predicate (page 68), for each fixed e and x, if there is a computation code y satisfying $T(e,x,y)$ then y is unique. Thus

$$z \in \mathrm{ran}(f) \Leftrightarrow \exists x (f(x) = z) \Leftrightarrow \exists x \exists y (T(e,x,y) \wedge U(y) = z)$$

We are not quite done because there are two existential quantifiers in front of the relation. We could combine the two variable x and y using a pairing function. Alternatively, we can simply note that if $T(e,x,y)$ holds then $x \leq y$ (since the computation coded by y includes the initial state, which has x in register R_1). Thus

$$z \in \text{ran}(f) \Leftrightarrow \exists y \underbrace{[\exists x \leq y(T(e,x,y) \wedge U(y) = z)]}_{R(z,y)}$$

Then R is a recursive relation, and $z \in \text{ran}(f)$ iff $\exists y R(z,y)$. $\square$

Remark: The above theorem remains true if in (iii), "primitive recursive" is replaced by "polynomial time computable". The same holds for (iv). However we will not need these facts.

Application of the Theorem: We can use the above theorem to give easy proofs that various sets are r.e. For example, recall $K = \{x \mid \{x\}_1(x) \neq \infty\}$. Let $f(x) = \Phi(x,x) = \{x\}_1(x)$. Then f is computable, and $K = \text{dom}(f)$. Thus K is r.e. by part ii) of the above theorem.

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 71). However, if f is computable but not total, then graph(f) is not necessarily recursive. As an example, let

$$f(x) = 0 \cdot (\mu y T(x,x,y))$$

Then $f(x) = 0$ 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 11 *Prove the above theorem. To show the if direction, first give an informal algorithm for computing f from an enumeration of the tuples in graph(f). Then formalize the argument by showing that f is recursive, using the least number operator μ.*

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.

Slick Proof: $A(x) = R(x, \mu y[R(x, y) \vee S(x, y)])$ $\square$

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.

Proof: Use ii) in the theorem on page 77 characterizing r.e. sets. Since B is r.e., $B = \mathrm{dom}(f)$ for some computable f. Since $A \leq_m B$, there is a total computable g such that

$$x \in A \Leftrightarrow g(x) \in B \Leftrightarrow f(g(x)) \neq \infty$$

then $A = \mathrm{dom}(f \circ g)$. Thus A is r.e. $\square$

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

Exercise 12 *Show that the following sets are not r.e. Note that*

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

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

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), but use the KNFT in an interesting way.

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 13 *Show that K is r.e. complete.*

Effective enumeration of functions

Recall that $\phi_e(x) = \{e\}_1(x)$, so ϕ_e is the unary function computed by program $\{e\}$. Thus $\phi_0, \phi_1, \phi_2, ...$ is an effective enumeration of all unary computable functions. However, it follows from the theorem on page 69 that there is no such effective enumeration of all *total* unary computable functions. Since the unary primitive recursive functions can be effectively enumerated, this is another way of showing that not all total computable functions are primitive recursive.

Exercise 14 *What goes wrong with the proof of the theorem on page 69 if we replace the list $f_0, f_1, ...$ of total computable functions by the list $\phi_0, \phi_1, ...$ of computable functions?*

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.