Notes on Recursion Theory

by Yurii Khomskii

This is a concise set of notes for the course Recursion Theory. It's not meant to replace any textbook, but rather as an additional guide for a better orientation in the material.

–Yurii

1. Models of Computation.

1.1. Introduction.

We are looking at the collection of natural numbers, denoted by $\mathbb{N} := \{0, 1, 2, 3, \dots\}$. Consider all $f : \mathbb{N} \longrightarrow \mathbb{N}$, or in general $f : \mathbb{N}^n \longrightarrow \mathbb{N}$, i.e., all functions that take some natural numbers as input and likewise output a natural number. Intuitively, some of these functions are **computable**, that is, for a given input $(x_0, \dots, x_n)$ there is a mechanical, algorithmic process for computing $f(x_0, \dots, x_n)$. For example, the function f defined by $f(x) := x \cdot 2 + 5$ is computable, because the following **algorithm** computes it: "take the input x, multiply it by 2 and then add 5, and output the result."

In general, there is a huge amount of functions from $\mathbb{N}^n$ to $\mathbb{N}$ (uncountably many) and not all of them can be computable. The essential point of Recursion Theory is to study this notion of computability, e.g. which functions are computable, "how" computable they are etc.

So far, the concept of "computability" was intuitive and, even though most of us have a good intuition about what qualifies as an algorithm, we still need to make the notion precise if we want to develop a mathematical theory about it. In other words, we need to give a formal definition capturing the intuitive notion of "algorithmically computable".

As it turns out, there are many equivalent definitions of this concept, the so-called **models of computation**, and we shall present the main candidates in the following sections. All of these define precisely the same class of computable functions, which we will denote by Comp.

$$\mathsf{Comp} := \{f : \mathbb{N}^n \longrightarrow \mathbb{N} \mid f \text{ is computable}\}$$

The postulation that the class defined by one of those mathematical formalisms is indeed the class of functions we would intuitively call "computable", is usually called the **Church-Turing Thesis**.

Note that the class Comp may also contains **partial** functions, i.e., functions whose value is not defined for every $x \in \mathbb{N}$. This is an essential feature of Recursion Theory, and it is not possible to give a satisfactory definition of computability which would avoid it (otherwise one could produce a contradiction by a diagonal argument.)

1.1. Definitions and Notation. Let f be a function from $\mathbb{N}^n$ to $\mathbb{N}$.

1. We denote a sequence $(x_0, \ldots, x_n)$ by $\vec{x}$.
2. If $f(\vec{x})$ is defined then we write $f(\vec{x}) \downarrow$. Otherwise we write $f(\vec{x}) \uparrow$.
3. The **domain** of f is $\mathrm{dom}(f) := \{\vec{x} \in \mathbb{N}^n \mid f(\vec{x}) \downarrow\}$.
4. The **range** of f is $\mathrm{ran}(f) := \{y \in \mathbb{N} \mid y = f(\vec{x}) \text{ for some } \vec{x} \in \mathbb{N}\}$.
5. If f and g are two functions then $f \circ g$ is the function defined by $(f \circ g)(x) := f(g(x))$.
6. A function f is **total** if $\mathrm{dom}(f) = \mathbb{N}$ and **partial** otherwise.
7. If $A \subseteq \mathbb{N}^n$ is a set then we use the notation $\overline{A}$ for the complement of A, i.e., $\mathbb{N}^n \setminus A$. $\lhd$

1.2. The Recursive Model.

Our first model of computability, namely the "recursive functions"-model, is based on the mathematical notion of **definitions by recursion**. We divide the definition into two steps: first, we define the simpler class of **primitive recursive** functions, denoted by PrimRec, and later we will extend it to the class of **recursive** functions, denoted by Rec.

1.2. Definition. The class PrimRec of primitive recursive functions is defined by induction as follows:

1. The **constant zero function** $f : \vec{x} \mapsto 0$ is primitive recursive. We will denote this function by $\mathbf{0}$.
2. The **successor function** $S : x \mapsto x + 1$ is primitive recursive.
3. The **i-th projection function** $e_i : (x_0, \ldots, x_n) \mapsto x_i$ is primitive recursive.
4. If $g, h_0, \ldots, h_n$ are primitive recursion then the **substitution** function f defined by $f(\vec{x}) := g(h_0(\vec{x}), \ldots, h_n(\vec{x}))$ is primitive recursive.

5. If g and h are primitive recursive, then f defined as follows is primitive recursive:

$$f(\vec{x}, 0) := g(\vec{x})$$
$$f(\vec{x}, y+1) := h(y, f(\vec{x}, y), \vec{x})^1$$

In this case, f is a definition **by primitive recursion based on initial function g and recursion function h.** $\triangleleft$

Examples:

- Suppose we want to prove that the identity function id defined by $id(x) := x$ is primitive recursive. Simply write $id = e_1$ and we are done.
- Suppose we want to prove that the function f defined by $f(x) := 3$ is primitive recursive. Then we can write

$$f = S \circ S \circ S \circ \mathbf{0}$$

The equality clearly holds because for all x we have $S(S(S(\mathbf{0})))(x) = S(S(S(0))) = S(S(1)) = S(2) = 3$. And, by the inductive definition, this function is primitive recursive.

- Suppose we want to prove that addition, i.e. the function mapping (x, y) to $x + y$, is primitive recursive. First, note that the intuitive recursion involved is the following:

$$x + 0 = x$$
$$x + (y + 1) = (x + y) + 1$$

Formally, then, we need the initial function $g := id$, and the recursion function $h = S \circ e_2$. Clearly, g and h are primitive recursive. But then, so is f defined by

$$f(x, 0) := g(x) = x$$
$$f(x, y+1) := h(y, f(x, y), x) = S(f(x, y)) = f(x, y) + 1$$

Clearly, f is the addition function, since the formal definition corresponds precisely to the intuitive one.

Note that all functions in PrimRec are total. Although many simple arithmetical functions are primitive recursive, this notion is not sufficient to formalize the intuitive concept of computability. Therefore, we expand the class of primitive recursive functions to the class of **recursive** (sometimes called μ-recursive) functions, denoted by Rec, by adding the μ-operator.

1.3. Definition. If $P(x)$ is any predicate (relation), then $\mu i\ (P(i))$ is defined to be **the least** $i \in \mathbb{N}$ such that $P(i)$ holds.

Now the class Rec is defined by induction:

[1] To avoid confusion, note that different textbooks use a different order here. This will be important when using projection functions: e_1, e_2 or e_3.

1. If g is primitive recursive then it is recursive.

2. If g is recursive then so is f defined by

$$f(\vec{x}) := \mu i[g(\vec{x}, i) = 0 \ \wedge \ \forall j < i \ [g(\vec{x}, j) \downarrow \neq 0]$$

$\lhd$

Essentially, the idea of point 2. above is the mathematical simulation of the following algorithm:

> "If g is a recursive function, compute the value of $g(\vec{x}, i)$ for incrementing i (i.e. $i = 0$, then $i = 1$, then $i = 2$ etc.) As soon as you find $g(\vec{x}, i) = 0$, output i."

N.B. $f(x)$ is only defined if there exists a i such that $g(\vec{x}, i) = 0$ **and** for all $j < i$, $g(\vec{x}, j)$ is defined and $\neq 0$. This corresponds to the intuition of μ being an algorithmic process computing the value of $g(\vec{x}, i)$ for incrementing i, which may get stuck while trying to compute an undefined value of $g(\vec{x}, j)$.

1.3. The Turing Machine Model.

Another model of computation is the Turing Machine. This is a formalism based not on an arithmetical notion like the recursive functions, but rather on the notion of an actual algorithmic process. Abstracting itself from a real computer on one hand, and a real mathematician with pen and paper on the other, it is arguably the strongest model of computation.

1.4. Definition. A **Turing Machine** is a device with a two-way **infinite tape** divided into cells, a **reading** and **writing head** and an internal finite set of **states** $Q := \{q_0, q_1, \ldots, q_n\}$. On the tape can be written two symbols: 0 (for "blank") or 1. The head can read one symbol at a time, write a new symbol on its place, and move left, right, or stay where it is. The actions of this machine are controlled by a **Turing program**, which is a finite sequence of **quintuples** of the following kind:

$$(q, s, q', s', X) \in Q \times \{0, 1\} \times Q \times \{0, 1\} \times \{L, R, C\}$$

Such a quintuple is interpreted as the following instruction to the Turing machine:

> "If you are in state q and you read an s, go to state q', write an s', and move the head left, right, or nowhere, depending on whether $X = L, R$ or C respectively."[2]

[2]Note that many texts give slightly different versions of the Turing machine. For example, in some cases the alphabet for the tape can consist not only of $\{0, 1\}$ but all natural numbers etc. In some cases (Soares) $X = C$ is prohibited, while in yet other presentations (Cooper) the machine can **either** print a new symbol **or** move to the right or left, in each step. These details, however, are irrelevant and it is easy to translate a program from one style to another.

By convention, every Turing machine starts in state q_1 (the **initial state**) and halts if and only if it reaches state q_0 (the **halting state**). $\lhd$

By convention, an **input** $x \in \mathbb{N}$ is represented by $x+1$ consecutive 1's and the **output** is the total number of 1's on the tape.

1.5. Definition. If T is a Turing machine then the corresponding function f_T is defined as follows:

$$f_T : \begin{array}{ccc} \mathbb{N} & \longrightarrow & \mathbb{N} \\ x & \longmapsto & \begin{cases} y & \text{if on input } x,\ T \text{ halts with output } y \\ \uparrow & \text{if } T \text{ never halts on input } x \end{cases} \end{array}$$

A function f is **Turing computable** if $f = f_T$ for a Turing machine T. The class of all Turing computable functions is denoted by $\mathsf{TuringComp}$. $\lhd$

1.5. Equivalence of Models.

As we already mentioned, all models of computability are equivalent. In our case, that means $\mathsf{Rec} = \mathsf{TuringComp}$. The **Church-Turing thesis** is the statement that this class is indeed the class of functions that we want to call "computable".

Henceforth, we will denote the class of computable functions simply as Comp, and refer to them as "computable". When necessary, we can use whichever formalism is the most convenient (usually Turing machines). In practice, however, it is rarely necessary to produce a formal Turing program, and when arguing that a given function is computable, it will be sufficient to give an intuitive description of the algorithm computing it (such an argument is sometimes called "proof by the Church-Turing thesis").

Recall that Comp also contains partial functions. Let $\mathsf{TotComp}$ denote the collection of computable functions which are also total (i.e. $\forall x\ f(x)\downarrow$).[3]

Note also that although we have only analyzed the computability of functions so far, we can naturally extend the definition of computability to cope with sets.

1.9. Definition. Let $X \subseteq \mathbb{N}$. Then

- X is **computable** if its characteristic function χ_X is **total** computable, where
$$\chi_X(x) = \begin{cases} 1 & \text{if } x \in X \\ 0 & \text{if } x \notin X \end{cases}$$

$\lhd$

[3]Sometimes the $f \in \mathsf{Comp}$ are explicitly called **partial computable**, or **p.c.** functions (Soares) whereas total computable functions are just called "computable". To avoid confusion, you should always check which concept we are talking about.

Intuitively, a set $X \subseteq \mathbb{N}$ is computable precisely if there is an algorithm which, given input x, determines whether $x \in A$ or $x \notin A$. The same thing holds for relations $R \subseteq \mathbb{N}^n$ using the same definition but with $\mathbb{N}$ replaced by $\mathbb{N}^n$.

2. Coding and Enumerating Computable Functions

2.1. Enumerating Turing Machines

Since our domain of discourse is the natural numbers $\mathbb{N}$, everything we do, including Turing machines, their programs, computations etc. needs to be coded as natural numbers. To do this, we first need to code sequences of numbers as numbers.

We know that in set theory there is a bijection $\mathbb{N}^n \cong \mathbb{N}$ for every n. But here, we would like this bijection to be computable. This can be achieved in many ways, and we present two:

1. **Gödel Numbering.** Let $(x_0, \dots, x_n)$ be a sequence of natural numbers. Let $\{p_0, p_1, p_2, \dots\}$ denote the prime numbers in order. Then we define the function gn by $gn(x_0, \dots, x_n) := p_0^{x_0} \cdot \ \dots \ \cdot p_n^{x_n}$. It can be checked that this function is injective, so every sequence of n natural numbers, regardless of n, has a unique such coding. It is also easy to see that this function is computable.

   Moreover, given an $n \in \mathbb{N}$, there are computable functions π_i which "decode" the number n and return the i-th decoded coordinate. That is, if $n = gn(x_0, \dots, x_n)$ then $\pi_i(n) = x_i$. Thus we can code and decode sequences of numbers as numbers, using a computable algorithm (e.g. a Turing machine).

2. **Standard Pairing Function.** The function gn is not surjective, i.e. there are n which do not code any sequence. Another, injective **and** surjective coding is the standard pairing function $\langle ., . \rangle : \mathbb{N}^2 \longrightarrow \mathbb{N}$ defined by $\langle x, y \rangle := \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$. If we want to code three numbers, we simply use $\langle x, y, z \rangle := \langle \langle x, y \rangle, z \rangle$, and similarly for $\langle x_0, \dots, x_n \rangle$. Here also, we can define a computable "decoding function", i.e. a π_i such that $\pi_i(\langle x_0, \dots, x_n \rangle) = x_i$.

It is not important which coding we choose (there are many others available as well), but the important thing is that it can be done.

Using this method, we can easily also code more complicated things, like Turing programs, Turing computations, primitive recursive functions etc. In particular, we fix an enumeration of **all Turing machines**:

$$T_0, T_1, T_2, \dots$$

using such a coding function. Intuitively, this enumeration is computable, in the sense that

$$e \longmapsto T_e$$

$$T_e \longmapsto e$$

are computable functions. We will see later on what this means formally.

Because computable functions correspond to Turing machines, we can also fix an enumeration of all computable functions. We denote these by

$$\varphi_0, \varphi_1, \varphi_2, \dots$$

For a φ_e, we call e the **index** of φ_e. Also, we use the notation $W_e := \mathrm{dom}(\varphi_e) = \{\vec{x} \mid \varphi_e(\vec{x}) \downarrow\}$.

2.1. Padding Lemma. For any $f \in \mathsf{Comp}$ there are infinitely many indices e such that $f = \varphi_e$.

Proof. Take any Turing machine T computing f. There are infinitely many ways of adding redundant instructions at the end of the program (for example those starting with a state which the Turing machine cannot enter.) All these will yield a different index. □

2.2. The Universal Turing Machine, Enumeration and S-m-n Theorem

As we mentioned, given an $e \in \mathbb{N}$, there is an effective algorithm for recovering T_e. Since it's effective, a Turing machine should be able to do it. But rather than outputting T_e (what should that mean anyway?), what it does instead is **running a simulation** of T_e.

2.2. Definition. A **Universal Turing Machine (UTM)** is a Turing machine which works as follows: taking (e, x) as input, it simulates the action of T_e on input x. ◁

2.3. Enumeration Theorem. *There is a computable function $\varphi_U : \mathbb{N}^2 \longrightarrow \mathbb{N}$ such that $\varphi_U(e, x) = \varphi_e(x)$.*

Proof. Any UTM computes φ_U. □

A more complex version of the Enumeration Theorem, where x is replaced by $\vec{x}$, is of course also possible.

As a converse to the Enumeration Theorem we have the so-called S-m-n Theorem, which we will again present in the simplest possible case where $n = m = 1$.

2.4. S-m-n Theorem. *There is a computable function* $S_1^1 : \mathbb{N}^2 \longrightarrow \mathbb{N}$ *such that for every computable* $\varphi_e : \mathbb{N}^2 \longrightarrow \mathbb{N}$ *we have*

$$\varphi_e(x, y) = \varphi_{S_1^1(e,x)}(y)$$

Proof. This involves a kind of "automated programmer", i.e., an algorithm that alters one Turing program into another.

Suppose we are given an index e and an input x. Then an algorithm for computing $S_1^1(e, x)$ can be described as follows:

- Recover the Turing machine T_e from e.
- Alter the machine T_e into another one $T_{e'}$, in such a way that
  - $T_{e'}$ first writes $x + 1$ many "1"s to the **left** of the input, leaving a blank between them and the other input.
  - Then it positions the head to the left of that new input, and proceeds by running T_e.
- Now output e'.

Now it is clear that $T_{e'}$ on input y computes precisely what T_e would compute on input (x, y). In other words, $\varphi_{e'}(y) = \varphi_e(x, y)$. But $e' = S_1^1(e, x)$, so this completes the proof. □

If we wish, we can be more exact as to the nature of S_1^1. An algorithm for altering T_e to $T_{e'}$ can be done as follows:

- Take T_e. Let T_e^* be T_e but with all states q_i renamed by q_{i+x+3} except q_0 which stays q_0.
- Then $T_{e'}$ is the following Turing program:

$$T_{e'} \quad = \quad \begin{bmatrix} (q_1, 1, q_2, 1, L) \\ (q_2, 0, q_3, 0, L) \\ (q_3, 0, q_4, 1, L) \\ (q_4, 0, q_5, 1, L) \\ \cdots \\ (q_{x+3}, 0, q_{x+4}, 1, C) \end{bmatrix}$$

$$\begin{bmatrix} \quad T_e^* \quad \end{bmatrix}$$

All that the function S_1^1 then does, is cleverly mapping (e, x) to e' by doing what we just did directly in terms of the codes of programs (for example, renaming q_i by q_{i+x+3} corresponds to some computable arithmetic operation like multiplication, exponentiation etc.)

3. Uncomputable Problems

3.1. Halting Sets

Recall that a set $X \subseteq \mathbb{N}$ is computable if, given x, there is an algorithm for deciding whether $x \in X$ or $x \notin X$. Most sets from "ordinary mathematics" are computable, e.g. $X = 2\mathbb{N}$, $X = \{\text{prime numbers}\}$ etc. But when the sets in question are connected with computable functions themselves, then we can use diagonalization arguments to show uncomputability.

3.1. Definition.

1. The **halting set** is the set $K_0 := \{\langle e, x\rangle \mid \varphi_e(x) \downarrow\}$.
2. The **diagonal halting set** is the set $K := \{x \mid \varphi_x(x) \downarrow\}$.

3.2. Theorem. *K is not computable.*

Proof. Suppose, towards contradiction, that K is computable. Then the following function is computable:

$$f(x) := \begin{cases} \uparrow & \text{if} \quad x \in K \\ 0 & \text{if} \quad x \notin K \end{cases}$$

Let e be its index, i.e., $f = \varphi_e$. But then

$$f(e) \downarrow \iff \varphi_e(e) \downarrow \iff e \in K \iff f(e) \uparrow$$

which is a contradiction. □

3.3. Corollary. *K_0 is not computable (and hence, the "halting problem" is not decidable).*

Proof. If K_0 is computable then K is computable, because we could decide whether $x \in K$ or not by asking whether $\langle x, x\rangle \in K_0$ or not. □

The last corollary is an example of reducing one set to another, in such a way that the computability of the latter implies the computability of the former.

3.4. Definition. Suppose $A, B \subseteq \mathbb{N}$. Then A is **many-one reducible** to B, notation $A \leq_m B$, if there is a total computable function $f : \mathbb{N} \longrightarrow \mathbb{N}$ such that for all x we have

$$x \in A \iff f(x) \in B$$

◁

3.5. Theorem. *If $A \leq_m B$ and B is computable, then A is computable.*

Proof. If $A \leq_m B$ via f then $\chi_A = \chi_B \circ f$. So if χ_B is computable, so is χ_A.

3.2. Index Sets

Index sets are special kinds of subsets of $\mathbb{N}$, namely those that correspond to properties of **functions** rather than numbers.

3.6. Definition. A set $X \subseteq \mathbb{N}$ is an **index set** if

$$\varphi_x = \varphi_y \to (x \in X \leftrightarrow y \in X)$$

Or equivalently: if $X = \{x \mid \varphi_x \in C\}$ for some $C \subseteq \mathsf{Comp}$. $\lhd$

It is immediately clear that if A is an index set then $\overline{A}$ is also an index set. Examples of such index sets are:

1. $K_1 := \{x \mid W_x \neq \varnothing\}$
2. $\mathrm{Fin} := \{x \mid W_x \text{ is finite}\}$
3. $\mathrm{Inf} := \overline{\mathrm{Fin}} = \{x \mid W_x \text{ is infinite}\}$
4. $\mathrm{Cof} := \{x \mid W_x \text{ is cofinite}\}$
5. $\mathrm{Tot} := \{x \mid W_x = \mathbb{N}\} = \{x \mid \varphi_x \text{ is total }\}$

3.7. Index Set Theorem. *Let A be an index set such that $A \neq \mathbb{N}$ and $A \neq \varnothing$. Then either $K \leq_m A$ or $K \leq_m \overline{A}$.*

Proof. Let e_0 be such that $\varphi_{e_0}(x) \uparrow$ for all x. Suppose $e_0 \in \overline{A}$. Pick any e_1 such that $\varphi_{e_1} \in A$. Define the function

$$\psi(x,y) := \begin{cases} \varphi_{e_1}(y) & \text{if } x \in K \\ \uparrow & \text{if } x \notin K \end{cases}$$

which is computable because an algorithm computing it is "run φ_x on input x; if it halts, run φ_{e_1} on input y; if this halts, return the output". By the S-m-n theorem, there is a computable function f such that for every x we have $\varphi_{f(x)}(y) = \psi(x,y)$. To be precise, $f(y) = S^1_1(e,y)$ where e is the index of ψ. But now

$$x \in K \implies \forall y\, [\varphi_{f(x)}(y) = \varphi_{e_1}(y)] \implies \varphi_{f(x)} = \varphi_{e_1} \overset{(*)}{\implies} f(x) \in A$$

$$x \notin K \implies \forall y\, [\varphi_{f(x)}(y) \uparrow] \implies \varphi_{f(x)} = \varphi_{e_0} \overset{(*)}{\implies} f(x) \in \overline{A}$$

where the $(*)$ implication holds because A and $\overline{A}$ are both index sets.

So K is many-one reducible to A via f. Now, if $e_0 \in A$, we get $K \leq_m \overline{A}$ by a similar argument but with the roles of A and $\overline{A}$ reversed. □

3.8. Corollary (Rice's Theorem). *Let A be any index set such that $A \neq \mathbb{N}$ and $A \neq \varnothing$. Then A is not computable.*

Proof. This follows from the Index Set Theorem and Theorem 3.5. □

Here is a quote about Rice's theorem from Cooper (p 105):

> "Magical as the above result may seem, its proof is firmly rooted in the unsolvability of the halting problem. Intuitively, it says that because we cannot computably decide whether a given computation ever halts, we cannot distinguish between machines which halt on *some* input and those which halt on *no* input — and if we cannot even do that, then there cannot be *any* computable dichotomy of the (indices of) machines."

4. Computably Enumerable Sets

4.1. Basic Things

The presentation of computably enumerable sets is slightly different from that in Soares.

4.1. Definition. A set $A \subseteq \mathbb{N}$ is **computably enumerable (c.e.)** if it is either $\varnothing$ or the range of a total computable function. $\lhd$

In other words, $A \neq \varnothing$ is c.e. iff there is a total computable f which "enumerates" all the elements of A:

$$A = \{f(0), f(1), f(2), f(3), \dots\}$$

There are two other useful and equivalent characterizations of c.e. sets.

4.2. Definition. A set $A \subseteq \mathbb{N}$ is called "**in Σ_1 form**", or simply Σ_1, if it is of the form $A = \{x \mid \exists y R(x, y)\}$ for some computable relation R. $\lhd$

This is the main characterization theorem for c.e. sets:

4.3. Theorem. *Let $A \subseteq \mathbb{N}$ be a set. The following are equivalent:*

1. A is c.e.
2. A is Σ_1.
3. $A = W_e = \mathrm{dom}(\varphi_e)$ for some e.

Proof.

- **1 $\Rightarrow$ 3.** If $A = \varnothing$ then $A = W_e$ for every nowhere-defined function φ_e. Otherwise, consider the following algorithm: "given input x, compute $f(i)$ for incrementing i. If $f(i) = x$, output 1." It is clear that this algorithm computes the following function

$$f(x) := \begin{cases} 1 & \text{if } x \in A \\ \uparrow & \text{if } x \notin A \end{cases}$$

  Letting e be the index of f, we get $A = W_e$. (Equivalently, consider $f(x) := \mu i\, (f(i) = x)$).

- **3 ⇒ 2.** For all x we have:

$$x \in A \iff \varphi_e(A)\downarrow \iff \exists s\,[\varphi_{e,s}(x)\downarrow]$$

and we know that $\varphi_{e,s}(x)$ is a computable relation.

- **2 ⇒ 1.** Suppose $A = \{x \mid \exists y R(x,y)\}$. If $A = \varnothing$ then it is c.e. by definition. Otherwise, pick one $p \in A$ (e.g. the least one), and define the following function:

$$f(\langle x,y\rangle) := \begin{cases} x & \text{if } R(x,y) \\ p & \text{otherwise} \end{cases}$$

Then f is clearly computable, it is total because the standard pairing function is surjective, and clearly $\mathrm{ran}(f) = A$. □

Intuitively, you can think of c.e. sets as follows: if we want to test whether a given x is in A or not, then there is an algorithm such that if $x \in A$ then it halts and confirms $x \in A$, but if $x \notin A$ then it does not halt, and does not confirm $x \in A$. Thus, the difference between computable and c.e. sets is the following:

A **Computable**	A **c.e.**
$x \in A \to$ the algorithm halts and confirms $x \in A$	$x \in A \to$ the algorithm halts and confirms $x \in A$
$x \notin A \to$ the algorithm halts and confirms $x \notin A$	—

We can make this intuition precise:

4.4. Lemma. *If A is computable then A is c.e.*

Proof. Consider the algorithm "run χ_A; if the output is 0, enter an infinite loop." If φ_e is the function computed by this algorithm, then clearly $A = W_e$. □

4.5. Complementation Theorem (Post). *A is computable iff both A and $\overline{A}$ are c.e.*

Proof. If A is computable then both A and $\overline{A}$ are computable so by Lemma 4.4 both A and $\overline{A}$ are c.e.

Conversely, suppose A and $\overline{A}$ are c.e. Suppose f enumerates A and g enumerates B. Then we define a function h by

$$h(0) := f(0)$$
$$h(1) := g(0)$$
$$h(2) := f(1)$$

$$h(3) := g(1)$$
$$\dots$$

Now consider the following algorithm: "given input x, find $\mu i(h(i) = x)$; if i is even, output 1, if i is odd, output 0". Since $\mathrm{ran}(h) = \mathrm{ran}(f) \cup \mathrm{ran}(g) = A \cup \overline{A} = \mathbb{N}$, we know that $\mu i(h(i) = x)$ is defined for all x. Then it is clear that the algorithm computes the characteristic function χ_A of A. □

4.2. "Dynamic" Enumerations

Each c.e. set W_e can be seen as a union of finite, computable approximations:

$$W_{e,0} \subseteq W_{e,1} \subseteq W_{e,2} \subseteq \dots$$

such that $W_e = \bigcup_{s \in \omega} W_{e,s}$. We can define $W_{e,s} := \mathrm{dom}(\varphi_{e,s})$, or equivalently, if W_e is enumerated by f, use $W_{e,s} := \{f(0), f(1), \dots, f(s)\}$. It doesn't matter which definition we use, but in each case the enumeration of W_e does not really depend on the enumeration of W_i, for $i \neq e$. What we would like to do instead, is to fix some **simultaneous enumeration** h of all the W_e's, in such a way that at each step precisely one new element is added to precisely one W_e.

4.6. Definition. A function h is a **simulatenous computable enumeration (s.c.e.)** of all c.e. sets if it is totally computable and

$$\mathrm{ran}(h) = \{\langle x, e\rangle \mid x \in W_e\}$$

Since such an h clearly exists (e.g. let $h(\langle n, e\rangle) := \langle f_e(n), e\rangle$ where f_e enumerates W_e) we fix, from now on, an s.c.e. h.

Then we define $W_{e,s} := \{x \mid \exists t \leq s\ (h(t) = \langle x, e\rangle)\}$. ◁

If X is a c.e. set, we write X_s to denote $W_{e,s}$ where $W_e = X$. Now we can **compare** the enumerability of different c.e. sets.

4.7. Definition. Let X, Y be two c.e. sets. Then

1. $X \setminus Y := \{x \mid \exists s\ (x \in X_s \wedge x \notin Y_s)\}$, the set of those x which are enumerated in X before they may eventually be enumerated in Y.
2. $X \searrow Y := (X \setminus Y) \cap Y = \{x \mid \exists s\ (x \in X_s \wedge x \notin Y_s \wedge \exists t > s(x \in Y_t))\}$, the set of those x which are first enumerated in X and at a later stage in Y. ◁

Another useful characterization of this is the following: suppose X is a c.e. set. Then we can define a function $m_X : \mathbb{N} \longrightarrow \mathbb{N}$ by $m_X(x) := \mu s\ (x \in X_s)$, i.e. the first stage s at which x "appears in" X. Moreover, since the relation "$x \in X_s$" is computable, the function m_X is actually partially computable. It follows from the definitions that if X and Y are two c.e. sets and $x \in X \cap Y$, then either $m_X(x) < m_Y(x)$ or $m_Y(x) < m_X(x)$, but **not** $m_X(x) = m_Y(x)$.

5. Recursion Theorem

5.1. Basics

5.1. Recursion Theorem (Kleene). *For every total computable function f there is an n such that $\varphi_n = \varphi_{f(n)}$.*

Proof (quite detailed).

- Consider the function ψ defined as follows:

$$\psi(x,y) := \begin{cases} \varphi_{\varphi_x(x)}(y) & \text{if } \varphi_x(x)\downarrow \\ \uparrow & \text{otherwise} \end{cases}$$

  Clearly, ψ is computable, since the following algorithm computes it: "run φ_x on input x; if it halts with output w, run φ_w on input y; if it halts output the result." Equivalently, if φ_U is the function corresponding to the Universal Turing Machine, then ψ can be written as $\psi(x,y) := \varphi_U(\varphi_U(x,x),y)$.

- Now by the S-m-n theorem there is a total, computable function d such that

$$\varphi_{d(x)}(y) = \psi(x,y)$$

  namely, d is defined by $d(x) := S^1_1(e,x)$ where e is the index of ψ.

- Since f and d are both total computable functions, $f \circ d$ is also a total computable function. Let z be its index, i.e.

$$\varphi_z = f \circ d$$

  Note that since φ_z is total, we have that in particular $\varphi_z(z)\downarrow$, so $\psi(z,y) = \varphi_{\varphi_z(z)}(y)$ for all y.

- But then, what can we say about the function $\varphi_{d(z)}$? For every y, we get

$$\varphi_{d(z)}(y) = \psi(z,y) = \varphi_{\varphi_z(z)}(y) = \varphi_{(f\circ d)(z)}(y)$$

  and so

$$\varphi_{d(z)} = \varphi_{f(d(z))}$$

  and hence $d(z)$ is the fixed point n that we were looking for. □

Basically, the intuitive consequence of the Recursion Theorem is that, when defining a computable function φ_n, we may use the number n in the definition. For example,

1. There is an n such that $W_n = \mathrm{dom}(\varphi_n) = \{n\}$
2. There is an n such that $\mathrm{ran}(\varphi_n) = \{n\}$

Let's quickly see how we can formally do this:

1. Define
$$\psi(x,y) := \begin{cases} 1 & \text{if } x = y \\ \uparrow & \text{if } x \neq y \end{cases}$$
By the S-m-n theorem there is a computable f such that $\varphi_{f(x)}(y) = \psi(x,y)$. Let n be the fixed point of f. Then for all y
$$\varphi_n(y) = \varphi_{f(n)}(y) = \psi(n,y)$$
so $W_n = \{n\}$.
2. Define $\psi(x,y) := x$. Then by the S-m-n theorem there is a computable f such that $\varphi_{f(x)}(y) = \psi(x,y)$. Let n be the fixed point of f. Then for all y we have $\varphi_n(y) = \varphi_{f(n)}(y) = \psi(n,y) = n$.

The second situation has some interesting applications to practical programming: since we can see n as representing the **code of** φ_n, the function φ_n can be said to "output its own code". There are people who try doing the same in real, existing programming languages. For example, this program in C apparently outputs its own code:

```
char x[]="char x[]=%c%s%c;%cint main() {printf(x,34,x,34,10,10);return 0;}%c";
int main() {printf(x,34,x,34,10,10);return 0;}
```

Such programs are also called **Quines**. The Recursion Theorem guarantees that Quines exist in every programming language which has the full computational power (i.e. equivalent to Turing machines).[4]

[4] If you are interested in this concept, try consulting e.g.
<http://www.madore.org/~david/computers/quine.html> or
<http://en.wikipedia.org/wiki/Quines>.
For examples of Quines in many languages, see <http://www.nyx.net/~gthompso/quine.htm>.