Math 29: Coding and Machines

April 11th, 2022

1 Codes

An important feature of register machines is that their building blocks are incredibly simple. While we can build very complex algorithms using register machines (and therefore also using Turing machines), it is very easy to describe a single node in our machine. It is either a start node, a stop node, an addition node, or a subtraction node. Each node is responsible for advancing program flow to at most two separate nodes.

From this point forward, for a given object x, we will use $\#x$ to represent the code for x. For example, $\#(x,y) = \frac{1}{2}(x+y)(x+y+1)+y$.

In order to describe a register machine, we have to describe each node, and where those nodes go. Suppose M is a register machine with n non-Start/Stop nodes N_i, each one given an index i in 1, 2, …, n, with N_1 being the node connected to the Start node. (We are reserving 0 as the index for the Stop node.) Then the code for M will be

$$\#M = \prod_{i=1}^{n} p_i^{\#N_i}$$

where p_i is the i-th prime and $\#N_i$ is the code for the i-th node, which we shall define below.

If N_i is an addition node R_j^+ with output node N_k, then

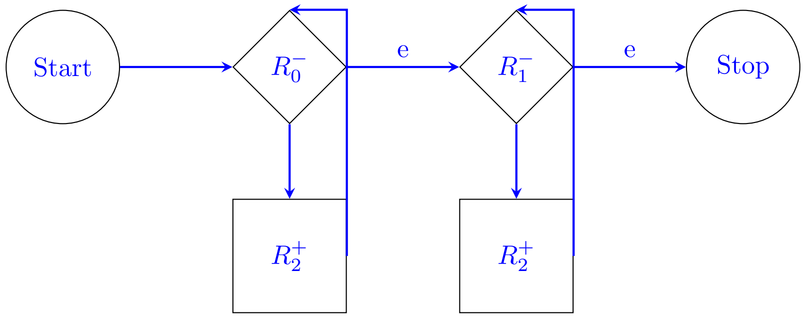
$$\#N_i = 3^j * 5^k$$

If N_i is a subtraction node R_j^- with output node N_k and empty output node N_l, then

$$\#N_i = 2 * 3^j * 5^k * 7^l$$

Example 1. *A code for the addition register machine M given in a previous lecture is* $3^{2*5^2*7^3} * 5^{3^2*5} * 7^{2*3*5^4} * 11^{3^2*5^3}$.

Proof: Recall that the addition machine is



There are four non-Start/Stop nodes. The one connected to the start node is N_1, and it is a subtraction node for R_0. Its output node is N_2, and its empty node is N_3. Therefore $\#N_1 = 2 * 3^0 * 5^2 * 7^3 = 2 * 5^2 * 7^3$.

N_2 is an addition node for R_2 whose output node is N_1, so $\#N_2 = 3^2 * 5^1 = 3^2 * 5$.

N_3 is a subtraction node for R_1. Its output node is N_4, and its empty node is the Stop node. Therefore $\#N_3 = 2 * 3^1 * 5^4 * 7^0 = 2 * 3 * 5^4$.

N_4 is an addition node for R_2 whose output node is N_3, so $\#N_4 = 3^2 * 5^3$.

Therefore,

$$\#M = 3^{2*5^2*7^3} * 5^{3^2*5} * 7^{2*3*5^4} * 11^{3^2*5^3}$$

We can also code Turing machines. Each instruction $\langle q_i, r_j, a_k, q_s \rangle$ can be coded in the form $2^i * 3^j * 5^k * 7^s$, where $0 = r_0 = a_0$, $1 = r_1 = a_1$, $* = r_2 = a_2$, $L = a_3$, and $R = a_4$. Then the code for the given Turing machine would be

$$\#N = \prod_{j=1}^{n} p_i^{\#I_j}$$

where the I_j's are the instructions.

Technically, in both our coding for the register machines and the coding for the Turing machines, a single machine has multiple codes based on how we number the nodes/instructions. This issue does not impact any of our practical uses of codes for machines, but can be avoided entirely if we so choose by giving formal rules by which nodes and instructions need to be numbered, which would then give each machine a unique way to number its components. As discussed in section 3.4 of the textbook, we can even make our codings surjective if we so choose.

Recall that we have shown the basic arithmetic functions are computable: addition, multiplication, subtraction, division, exponentiation, etc. While we have not discussed it, the floor (least natural number less than or equal to) and ceiling (largest natural number greater than or equal to) of natural-base logarithms are as well: given a base $b > 1$ in R_0 and some natural number n, we can start computing $b^0, b^1, b^2, b^3, \ldots$. Since this sequence is unbounded, we can use the fact that less than (and greater than by symmetry) and equality are computable to check when this sequence exceeds or equals n. We can then return the appropriate power of b as required.

1.1 The Church Turing Thesis

Notice the following about the above argument: we did not explicitly build a register machine. We used the fact that we knew certain relations and functions were computable to describe the process of computing something, with knowledge that we'd be able to construct a formal register machine from the described process if we were required to. From this point forward, unless the question asks for us to construct a machine, we will adopt the **Church-Turing Thesis**, which says essentially that "real" computability is exactly the same thing as Turing computablity. (And, by extension, register computability.) The utility of this statement is that we can describe how to compute something in words, equations, or by describing a process. As long as we don't try to use noncomputable information or do infinitely many things, then we can be assured that we'd be able to create the necessary register/Turing machine if we needed to. When asked to show something is computable without being instructed to give a specific machine, proviing code, pseudocode, or describing the process by which you'd compute the desired object is sufficient.

Notice that this is not a formal mathematical statement: “real” computablity, like the terms “trivial” or “natural” when used in mathematics, does not have a formal definition. It is impossible to prove (and, depending on your philosophy, impossible to disprove) the Church-Turing thesis. While a careful mathematician rightly may feel discomfort here, you should think of it as saying “the things which you can ever hope to compute are exactly the things which you can already write a program to compute in your favorite programming language.” (Here we are assuming infinite time and memory: it’s computable if you can write a program that would compute it, even if that program can’t run on any physical computer we have.)

Giving a formal proof that a given programming language, like C++ or Python, is **Turing complete** (i.e. the things it can compute are exactly the same thing as we could compute with Turing machines) is generally tedious. However, the following informal argument is convincing: it is not hard to write Turing machine modules to handle bitwise And and Negation. Computers are just large circuit-boards of binary gates. Any binary gate can be represented by some composition of And and Negation gates (a formal mathematical statement which is true, but we do not prove it here), so for any gate on the circuit board, we can build a Turing machine module to simulate it. However, when our favorite C++ program or Python script is compiled/interpreted, it is turned into machine code that runs through the hardware gates. So all of it can be simulated by a Turing machine that simulates the low-level machine code it produces. (The reason that this is not a formal proof is that many compilers, for example g++, are not Turing-complete despite the language of C++ being so. But it illustrates the main ideas. A similar argument applies to quantum computers as well.)

Here is an example of a valid way to apply the Church-Turing thesis.

Lemma 2. *Given an infinite set $A \subseteq \omega$, its characteristic function is computable if and only if its principle function is computable.*

Proof: Suppose χ_A is computable. Compute $p_A(n)$ for arbitrary n by calculating values of $\chi_A(k)$ for $k = 0, 1, 2, \ldots$. When we find the n-th k such that $\chi_A(k)$ returns 1, halt and return k.

Suppose p_A is computable. Compute $\chi_A(n)$ for arbitrary n by calculating values of $p_A(k)$ for $k = 0, 1, 2, \ldots$. Since A is infinite, eventually we will either see $p_A(k) = n$, in which case we halt and return 1, or we see $p_A(k) > n$ but $p_A(i) < n$ for $i < k$, in which case we halt and return 0.

1.2 Back to Coding

With the (unnecessary but expedient) aid of the Church-Turing thesis, we can describe how register machines can read codes and simulate the behavior of the register machine coded by a number.

We can compute the characteristic function of the prime numbers via the following procedure: given n, check the modulus (remainder) of n by k for each $1 < k < n$. If it is ever 0, return 0 as n is not prime. If it is never 1, n must be prime so we return 1. Then by the above lemma, the principle function for the prime numbers must be computable, so given n there is a computable process which outputs p_n. In particular, given natural numbers n and k greater than 1, we can use repeated division to calculate the largest power of k which divides n.

In particular, given $n = \#M$, we can compute $\#N_i$ for each of M's nodes by figuring out the powers of each prime which divide n. We can also determine if $\#N$ is an addition or subtraction node by checking if it is even, which register it should operate on by checking the power of 3 that divides it, and which node(s) it continues to by doing the same for 5 and 7. **In particular, we can write a program which takes in $n = \#M$ as input and then "runs" M.** We'll explore this idea more next time.