Tape machines are $\approx$ Markov algorithms in a Turing machine style. These "machines" apply productions to strings, so tape machines are also like formal grammars. Here I focus on *deterministic* tape machines that map strings to strings, *if* the calculation halts or returns. Note that the "tape" is just an "instantaneous description" of the state of the machine. It is just a string, a finite sequence of "letters" from its formal alphabet.

Tape machines are equivalent to Turing machines, but they are a little more human-friendly. If we restrict our alphabet to 1 and 0 and encode a second human-friendly alphabet in these two symbols in an intuitive way, then tape machines are very easily understood as partial recursive functions from N to N, mapping base-2 representations to base-2 representations.

Each tape machine is defined by what I call a **Markov function**. The tape is updated by this function. Here I will focus on alphabet of decimal digits, since this illustrates that Markov functions treat numbers in unusual ways: what they do is usually hard to express in terms of addition and multiplication.

For instance, we define the Markov function f by

$$f := 29 \to 3; 2 \to 7; 72 \to 9$$

.

Note that these productions are *substring replacement rules.* To compute $f(x)$, we apply only the first applicable production to x, if there is one. We also always apply that production just once, at the leftmost (earliest) opportunity. If no production applies, then $f(x) = x$. Note that f is defined above in a "human friendly alphabet." We can also express it in a "machine code" of just 0s amd 1s.

With this definition, we have $f(129) = 13$. Note that we did **not** apply the production $2 \to 7$. This is because we had an opportunity to apply $29 \to 3$, which comes first in the list.

Also $f(202) = 702$, because we replace only the *first* 2 and leave any others alone.

Let $f^n(x)$ be the composition of f with itself n times. So $f^2(322) = f(f(322)) = f(372) = 39$. Using this f^n notation, it's easy to do a bounded computation, but what we are *most* interested in is a continuing application of f until no production is possible. This gives us, more or less, a Markov algorithm on numerals, which may not terminate.

We let $t_0 = x$. Then $t_{n+1} = f(t_n)$. The algorithm terminates if and when $t_{n+1} = f(t_n)$. In other words, the algorithm continues until a fixed point for f is discovered among the t_i. If this happens, then we can define $f^*(x) = t_n$. So f^* is the partial recursive function defined in terms of the total recursive "step" function f.

For instance, let's define $f = 2 \to 3; 3 \to 4;$. Then the attempted calculation of $f^*(2)$ gives us the finite tape sequence $2 \to 3 \to 4 \to 4$. We stop whenever we see the same n immediately next to itself. So $f^*(2) = 4$. In the usual language, 2 is in the domain of f^*.

Let's see at least one non-terminating calculation. Let $f = 2 \to 3; 3 \to 2;$. Then the attempted calculation of $f^*(3)$ gives us $3 \to 2 \to 3 \to 2 \to ...$ This is an obvious infinite loop. Indeed, any repetition in the t_i that involves a cycle of length 2 or more indicates a non-terminating computation.[1] We can use $f^*(3) = \infty$ or $f^*(3) \nearrow$ to indicate this situation.

For instance, let the program step function be $f = f_0| \to \mathit{OO}f_0; f_0 \to |$.[2] Our convention is to start with the particular state symbol f_0 at the beginning of the tape. An easy approach is just to stick with f_i for names of "states", which is to say for all symbols that aren't 0 and 1. The tape is just an always-finite string, without "infinite blanks" stretching out in one or both directions in many treatments of Turing machines. The productions allow for what amounts to insertion and deletion.

Let's try to compute $f^*(|||)$.

$$f_0||| \to \mathit{OO}f_0|| \to \mathit{OOOO}| \to \mathit{OOOOOO}f_0 \to \mathit{OOOOOO}| \to \mathit{OOOOOO}|.$$

So $f^*(|||) = \mathit{OOOOOO}|$, because we've found a fixed point. There's no rule against using productions that operate on O and $|$, but the "Turing machine style" basically comes from using "letter symbols" to "do the work."

MACHINE CODE

We encode O as 10. We encode $|$ as 110. We encode f_0 as 1110 and f_1 as 11110 and so on. So the tape $f_0O|$ is encoded as 111010100. Note that this encoding prevents leading zeros, so the tape is always a binary representation. Indeed the tape is always an even number.

We can encode programs — the Markov step function f — with the additional encoding of $\to$ as 0. We can actually use 0 for the semicolon

[1]We could of course declare that the program terminates when such a cycle is finally found, but we still don't have a output.

[2]Here I use "stylized" ones and zeros, which helps us distinguish them from state-symbol subscripts.

between instructions also.

This means that we encode $f_0| \to Of_1; f_0O \to f_1$ as

$$1110110010111100111010.$$

Here we also avoid leading zeros, and any step function (which is after the essence of tape machine program) is just an even number expressed in binary.

We can implement tape machines in C express machine code in memory by operating on individual bits. A universal tape machine can be constructed with less than 200 productions. The input tape for this universal machine includes both the program to simulate and the input to simulate this program on. We can use a 0 at the end of the program part of the tape to do this, since the input will not have leading zeros.

If we operate in this "machine code," which is easily translated into "assembly," then our tape machines are "immediately" partial numerical functions from $N \to N$. Not all numbers are programs, but it is easy to detect which ones are, so it is also easy to enumerate all tape machines.

One advantage of this approach is that we can view tape machines as functions on strings and numbers at the same time, more easily than we can in other models of computation. Markov functions on the usual decimal digits also allow for the easy creation of "strange" numerical functions that are also obviously computable. Addition and multiplication, which children learn as string algorithms, are generalized in a vivid way.

We can also consider tape machines that are interpreted to map rational numbers to rational numbers, and so on. By working in the rationals, we could get some very strange graphs. If the calculation does not converge in a fixed number of steps, we don't graph that point or declare it equal to 0, etc.