C) A nondeterministic Turing machine

Just like a nondeterministic finite state acceptor, a nondeterministic Turing machine may proceed according to different possibilities, so its computation is a tree, where each branch corresponds to a different possibility. The transition mapping of such a nondeterministic Turing machine is given by $t : S \times \tilde{A}^k \rightarrow P(S \times \tilde{A}^k \times \{L, R\})$ where:

- P shows we have different possibilities on how to processed.

Theorem:

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

Idea of the Proof:

We construct a deterministic Turing machine that simulates the nondeterministic one by trying out all possible branches. If it finds an accept state on one of these computational branches, it accepts the input, otherwise it loops.

Corollary:

A language L is Turing-recognisable ⇔ some nondeterministic Turing machine recognises L.

Proof:

"⇒" A deterministic Turing machine is a nondeterministic one, so this direction is obvious.
"⇐" follows from the previous theorem.

D) Enumerators

As we saw, a Turing-recognisable language is called in some textbooks a recursively enumerable language. The term comes from a variant of a Turing machine called an **enumerator**.

Loosely, an enumerator is a Turing machine with an attached printer. The enumerator prints out the language L it accepts as a sequence of strings. Note that the enumerator can print out the strings of the language in any order and possibly with repetitions.

Theorem:

A language L is Turing-recognisable ⇔ some enumerator enumerates (outputs) L.

Proof:

"⇐" Let E be the enumerator. We construct the following Turing machine M:

M = on input w

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appears in the output of E, accept w.

Thus, M accepts exactly those strings that appear on E's list and no others, hence exactly L.

"⇒" Let M be a Turing machine that recognises L. We would like to construct an enumerator E that outputs L. Let A be the alphabet of L, i.e. $L \subset A^*$.

In the unit on countability, we proved A^* is countably infinite (note that the alphabet A is always assumed to be finite), so A^* has an enumeration as a sequence $A^* = \{w_1, w_2, \ldots\}$

E = Ignore the input

1. Repeat the following for i = 1, 2, 3, …
2. Run M for i steps on each input $w_1, w_2, \ldots, w_i$
3. If any computations accept, print out the corresponding w_j.

Every string accepted by M will eventually appear on the list of E, and once it does, it will appear infinitely many times because M runs from the beginning on each string for each repetition of step 1.

Note that each string accepted by M is accepted in some finite number of steps, say k steps, so this string will be printed on E's list for every $i \geq k$. (*q.e.d*)

Moral of the Story:

The single-tape Turing machine we first introduced is as powerful as any variants we can think of.

Algorithms

Task:
Use Hilbert's 10^{th} problem to give an example of something that is Turing-recognisable but not Turing-decidable.
We saw that the continuum hypothesis of Cantor was the 1^{st} of Hilbert's 23 problems in 1900 at the International Congress of Mathematicians.

Hilbert's 10^{th} Problem:
Find a procedure that tests whether a polynomial in 2 variables (x and y) with integer coefficients (2, -1, -1) that has integer roots $p(1, 1) = 2.1^2 - 1.1 - 1^2 = 0$ so $x = 1 = y$, $1 \in Z$ is a solution.
Hilbert's problem asked how to find integer roots via a set procedure.