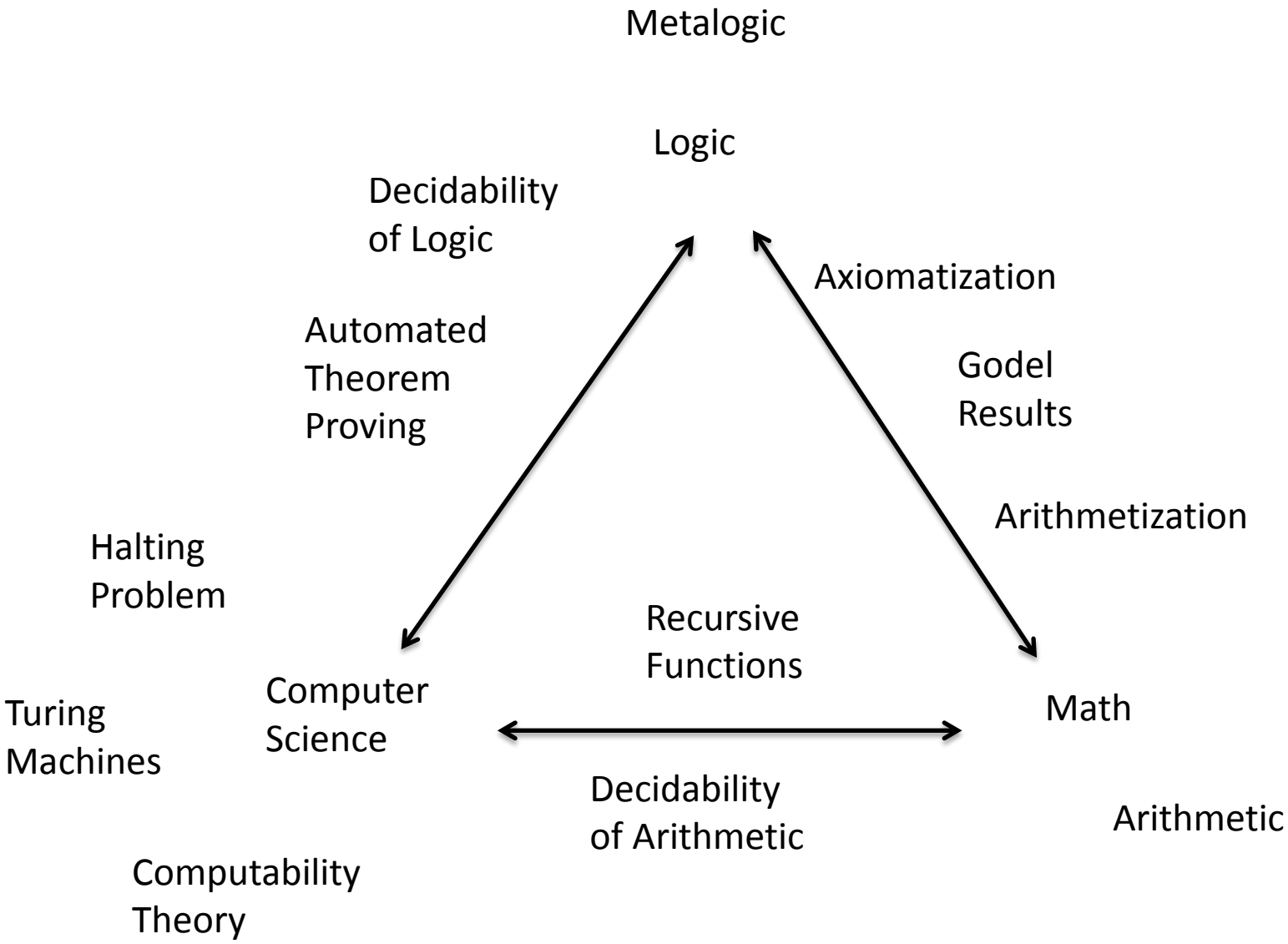
Turing-Machines

Computability and Logic


Metalogic
Logic
Decidability of Logic
Axiomatization
Automated Theorem Proving
Godel Results
Arithmetization
Halting Problem
Recursive Functions
Computer Science
Math
Turing Machines
Decidability of Arithmetic
Arithmetic
Computability Theory

Central Question

- Given any statement about mathematics, is it true or is it false?

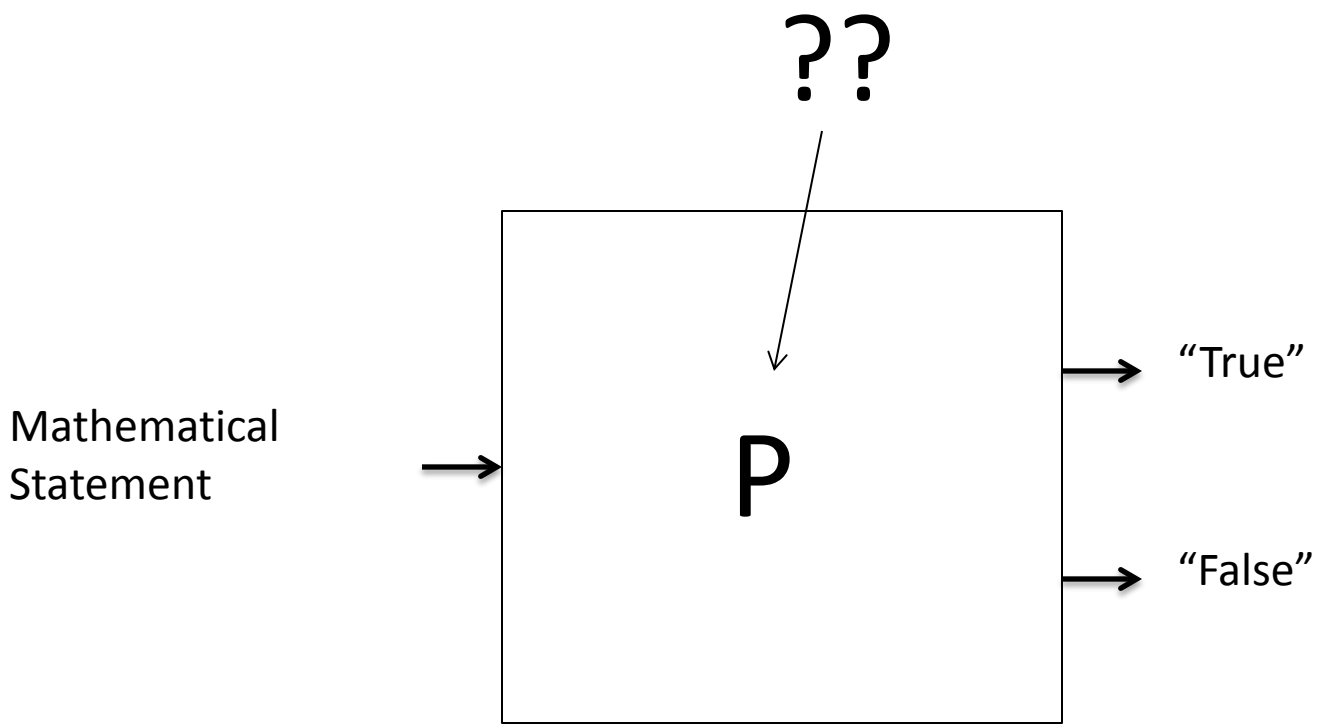
- Example:
  - Goldbach Conjecture: Every even number greater than 2 is the sum of two prime numbers.
  - True or false?

Decision Problem

- The *decision problem* for mathematics is to find a way to decide whether some statement about mathematics is true or false.

- Each domain has their own decision problem(s).

Decision Procedures



P being a 'procedure' can be understood as P being some kind of *systematic method*, or *algorithm*

Soundness and Completeness

- A decision procedure P is *sound* with regard to some domain D iff:
  - For any statement S about domain D:
    - If P returns “True”, then S is true
    - If P returns “False”, then S is false
- A decision procedure P is *complete* with regard to some domain D iff:
  - For any statement S about domain D:
    - If S is true, then P returns “True”
    - If S is false, then P returns “False”

The Axiomatic Method

- Define a set of axioms A_D that capture basic truths about some domain D, and see whether some statement S about domain D follows from A_D

The Hope

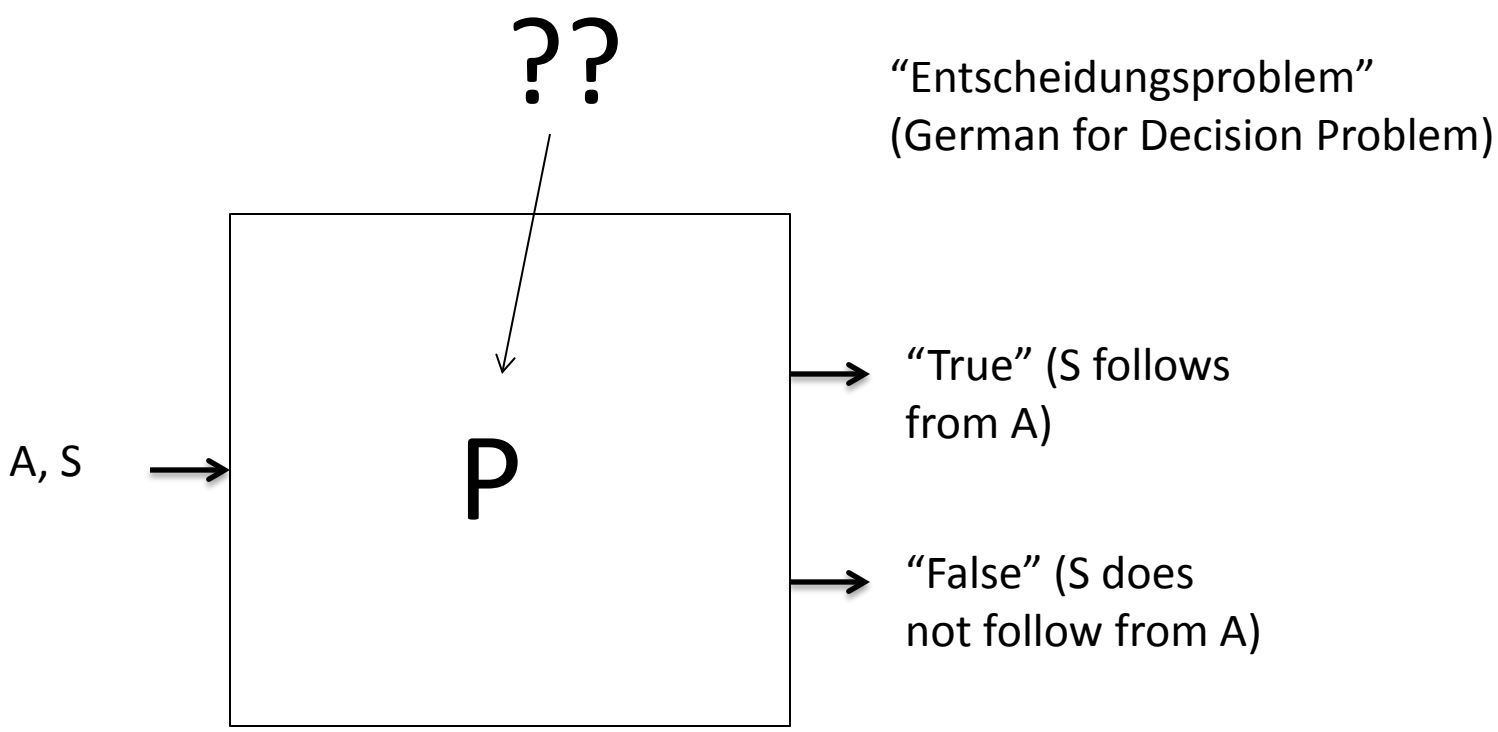
1. There exists a set of axioms A such that all, and only, mathematical truths are logical consequences of A
2. There exists a decision procedure P for deciding whether or not some mathematical statement S is a logical consequence of A

The Really Exciting Prospect

- If 1 and 2 are true, and assuming we can implement P on a computer (which, assuming such a P exists, is quite likely, since it's all formal logic!):

- Proving or disproving mathematical claims can be completely automated!

Decision Procedure for Logical Consequence



Alan Turing, 1936

- Published “On Computable Numbers, with an application to the Entscheidungsproblem”
  - … his ‘Turing-machines’ paper!
- Argued that the Entscheidungsproblem is unsolvable
  - There is no sound and complete systematic method for deciding whether some (first-order logic) argument is valid or not
  - (combined with Godel’s work on the arithmetization of logic, this implies that there is also no decision procedure for arithmetic, let alone all of mathematics)
- Required Turing to consider *all* possible ‘systematic methods’, i.e. ‘computations’ or ‘symbol-manipulating algorithms’ … but *how*?

Computations

- A computation is a symbol-manipulation algorithm.
  - Example: long division.
- Not every algorithm is a computation
  - Example: furniture assembly instructions

A COMPUTER WANTED.

WASHINGTON, May 1.—A civil service examination will be held May 18 in Washington, and, if necessary, in other cities, to secure eligibles for the position of computer in the Nautical Almanac Office, where two vacancies exist—one at $1,000, the other at $1,400..

The examination will include the subjects of algebra, geometry, trigonometry, and astronomy. Application blanks may be obtained of the United States Civil Service Commission.

The New York Times
Published: May 2, 1892

Computers

- A ‘computer’ is something that computes, i.e. something that performs a computation, i.e. something that follows a systematic procedure to transform input strings into output strings.
- Humans can take the role of a computer.
  - Of course, this does require that the human is able to understand and perform the operations, i.e. that the human can indeed execute the algorithm.
  - If this is so, we call the computation ‘effective’.
- Modern computers have mechanized the process of computation
  - Are there mechanical computations that no longer can be performed by humans, i.e. that are no longer effective?

The Scope and Limits of Effective Computation

- In his 1936 paper, Turing tried to figure out what the basic elements are of any effective computation … or at least to what elements any kind of computation can always be reduced to.

States and Symbols

- Take the example of multiplication: we make marks on any place on the paper, depending on what other marks there already are, and on what ‘stage’ in the algorithm we are (we can be in the process of multiplying two digits, adding a bunch of digits, carrying over).
- So, when going through an algorithm we go through a series of stages or states that indicate what we should do next (we should multiply two digits, we should write a digit, we should carry over a digit, we should add digits, etc).

A Finite Number of Abstract States

- The stages we are in vary between the different algorithms we use to solve different problems.
- However, no matter how we characterize these states, what they ultimately come down to is that they indicate what symbols to write based on what symbols there are.
- Hence, all we should be able to do is to be able to discriminate between different states
  - what we call them is completely irrelevant!
- Moreover, although an algorithm can have any number of stages defined, since we want an answer after a finite number of steps, there can only be a finite number of such states.

A Finite Set of Abstract Symbols

- Next, Turing pointed out that the symbols are abstract as well: whether we use ‘1’ to represent the number 1, or ‘☺’ to do so, doesn’t matter.
- All that matters is that different symbols can be used to represent different things.
  - What actual symbols we use is irrelevant!
- Also, while we can use any number of symbols, any finite computation will only deal with a finite number of symbols. So, all we need is a finite set of symbols.

A String of Symbols

- While we can write symbols at different places (e.g. in multiplication we use a 2-dimensional grid), symbols have a discrete location on the paper. These discrete locations can be numbered.
- Or, put another way: we should be able to do whatever we did before by writing the symbols in one big long (actually, of arbitrarily long size) string of symbols.

Reading, Writing, and Moving between Symbols

- During the computation, we write down symbols on the basis of the presence of other symbols. So, we need to be able to read and write symbols, but we also need to get to the right location to read and write those symbols.
- With one big long symbol string, however, we can get to any desired location simply by moving left or right along this symbol string, one symbol at a time.

The Scope and Limits of Effective Computation VI

- Turing thus obtained the following basic components of effective computation:
  - A finite set of states
  - A finite set of symbols
  - One big symbol string that can be added to on either end
  - An ability to move along this symbol string (to go left or right)
  - An ability to read a symbol
  - An ability to write a symbol
- This is … a Turing–machine!

Church-Turing Thesis

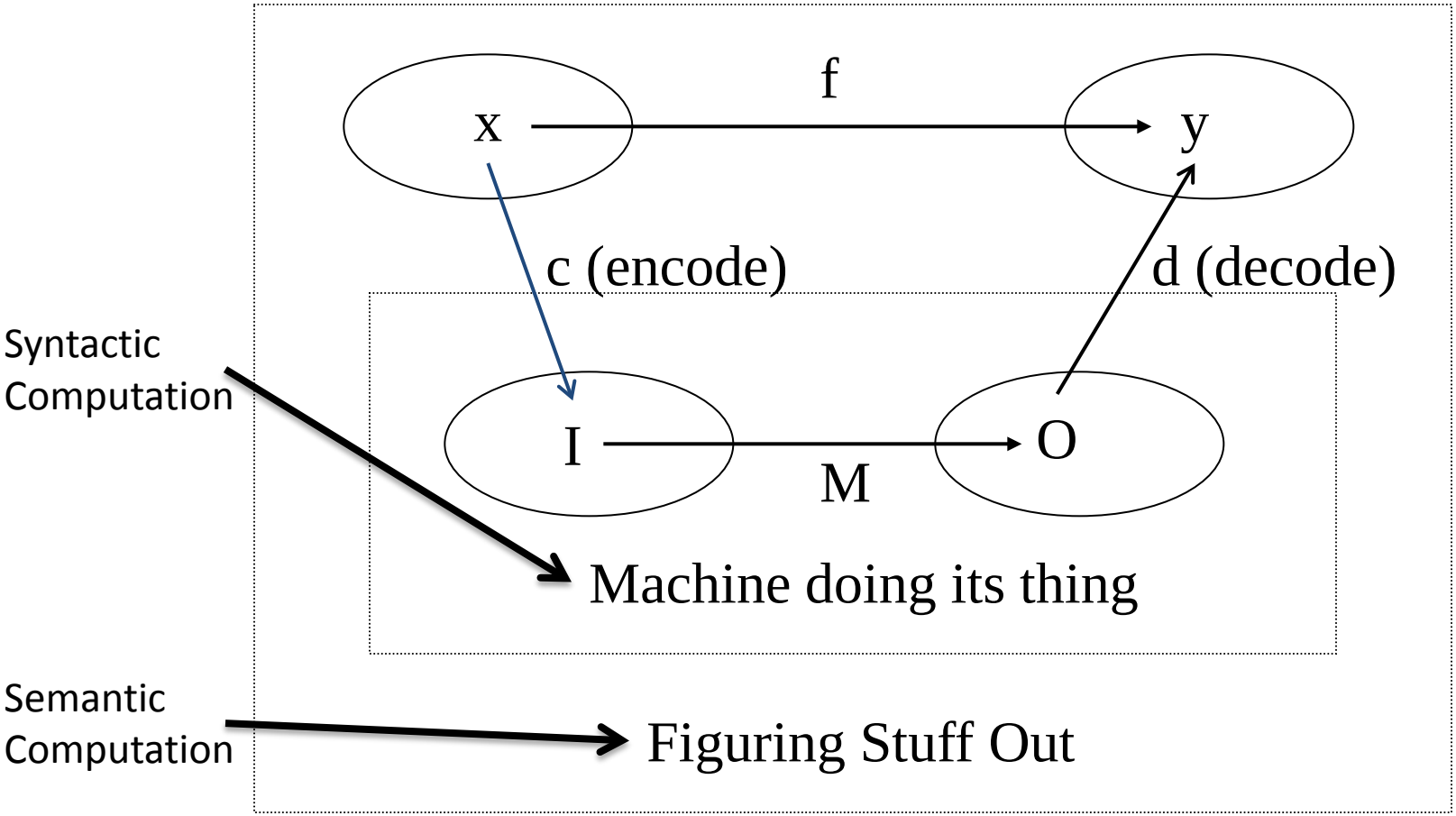
- The Church-Turing Thesis is that anything that we can figure out using any kind of systematic method can be figured out using a Turing-machine.
- In short: *anything that is effectively computable is Turing-computable*
- Note: we can't mathematically prove this Thesis, since we can't mathematically prove whether the mathematical definition of Turing-computability captures the conceptual notion of computability … but we can provide (and have!) lots of evidence in its favor: so far, we have not a single example of a systematic method that we can follow that figures things out that Turing-machines cannot!

Turing Machines Demo

‘Computing Things’ Using a Turing-Machine

- Remember that we want computations to ‘figure something out’, i.e. we want to ‘compute something’. E.g:
  - Compute the answer to some question
  - Compute the solution to a problem
- The things we want to compute can often (always) be stated as functions (e.g. question → answer. Or: problem → solution)
- So: we have a machine that transforms input symbol strings into output symbol strings, and we want to use that to compute a function from some domain to some other domain.
  - But that means that to compute a function, we need to interpret what the machine is doing. See next slide!

Computing Functions



Machines *as* Functions

- We *can*, and it is in fact useful to, treat any Turing-machine M as a function …
- But it would be a function that maps input configurations to output configurations.
- In other words, M : TC → TC

Representations and Configurations

- OK, so to use a Turing-machine to compute a function, we need to specify how we are going to represent the input to the function on the input tape … but we also need to specify where the read/write head is!
  - Indeed, one *could* regard the position of the read/write head as part of the representation, in that the same tape could represent different things, depending on the position of the head on that tape … though this is highly 'irregular'
- Let's call the contents of the tape, together with the position of the read/write head, the (tape) *configuration* of the Turing-machine.
  - Let TC be the set of all possible tape configurations
  - Oh, and let TM be the class of all Turing Machines

Representing Natural Numbers

- From now on, we are going to focus on functions from (any number of) natural numbers to natural numbers. E.g
  - Addition
  - Multiplication
  - Successor
- OK, so we need some representation of numbers!
  - Decimal?
  - Binary?
  - Unary?
  - … all are ok … but some are more ‘computation-friendly’ than others!

‘Standard’ Representation 1

- One ‘standard’ way to encode numbers using a Turing-machine is to use unary representation:
  - To represent the number n, we use a block of n consecutive 1’s
  - If we have multiple numbers, we separate them by 0’s (or blanks)
  - Also, for the input and output, standard practice is to require that the read/write head is on the left-most 1 of the left-most block of 1’s
- So: let us define $[n_1,n_2, \ldots, n_m]$ as the configuration of having a block of n_1 consecutive 1’s on the tape, followed by a 0, followed by a block of n_2 consecutive 1’s, followed by a 0, …, followed by a block of n_m consecutive 1’s, on an otherwise blank (=0) tape, and with the read-write head on the left-most 1 of the left-most block of 1’s

Computing the Successor Function

- As an example, let us use a Turing-machine to compute the successor function, i.e. s(n) = n+1, using the just defined convention.
- That is, we want a Turing-machine that for any n, transforms input configuration [n] into output configuration [n+1]
- OK, this will do (0 is start state, 2 is halting state):
  - <0,1,L,1> (in state 0, if you see a 1, move left and go to state 1)
  - <1,0,1,2> (in state 1, if you see a 0, write a 1, and move to state 2)
- Let us call this machine M_s

More Formally

- Notice that machine M_s does something with *any* input configuration, whether it is of the form [n] or not. But, relevant to what we want to show, we have that in particular:

$$M_s([n]) = [n+1]$$

- Let us define encoding c: N → TC:

$$c(n) = [n]$$

- Finally, let us define decoding c: TC → N:

$$d(T) = \begin{cases} n & if\ T = [n] \\ undefined & otherwise \end{cases}$$

- It is now easy to see that for any n N:

$$d(M(c(n))) = n+1$$

- This is why we can say that machine M_s computes function s

Using M to ‘compute’ f

- OK, so it seems that in general, we have the following definition:
- Machine M computes function $f : X \rightarrow Y$ iff:
  - There exists an encoding c and decoding d such that for all $x \in X$ and $y \in Y$:
    - if $f(x) = y$, then $d(M(c(x))) = y$

Whoops!!

- With this definition, *every* function whose domain is enumerable is computable …. by the null machine! (i.e. the machine that does nothing, i.e. for which M(i) = i)
- Proof: For c, pick any injective function. Now define d as: $d(o) = f(c^{-1}(o))$!
  - (alternative proof: pick c in such a way that it already encodes the answer as part of its encoding of the question … this is called 'Bramming')

‘Reasonable’ Representations

- ‘Brammings’ are not ‘reasonable’ representations!
- We want ‘reasonable’ representations.
- But how to define this?

Transformable Encodings

- “The encoding can be done in may reasonable ways. It does not matter which one we pick, because a Turing machine can always translate one such encoding into another” (Sipser)
- But how do we know this is true? Maybe there is some ‘reasonable’ encoding that is not transformable into other ‘reasonable’ encodings.
- And if we *define* a ‘reasonable‘ encoding as one that is transformable into (…what?! How to avoid a circular definition here?!) … then that puts the cart before the horse!

‘Effective’ Encodings

- Probably the best thing we can do is to say that encodings should be ‘effective’ , i.e. that humans should be able to perform the encoding.

Church-Turing Thesis, Revisited

- We can now a bit more careful about the Church-Turing Thesis:
  - Anything that is effectively figure-out-able, is figure-out-able using a Turing-machine and using effective representation conventions