Algorithms and Computers

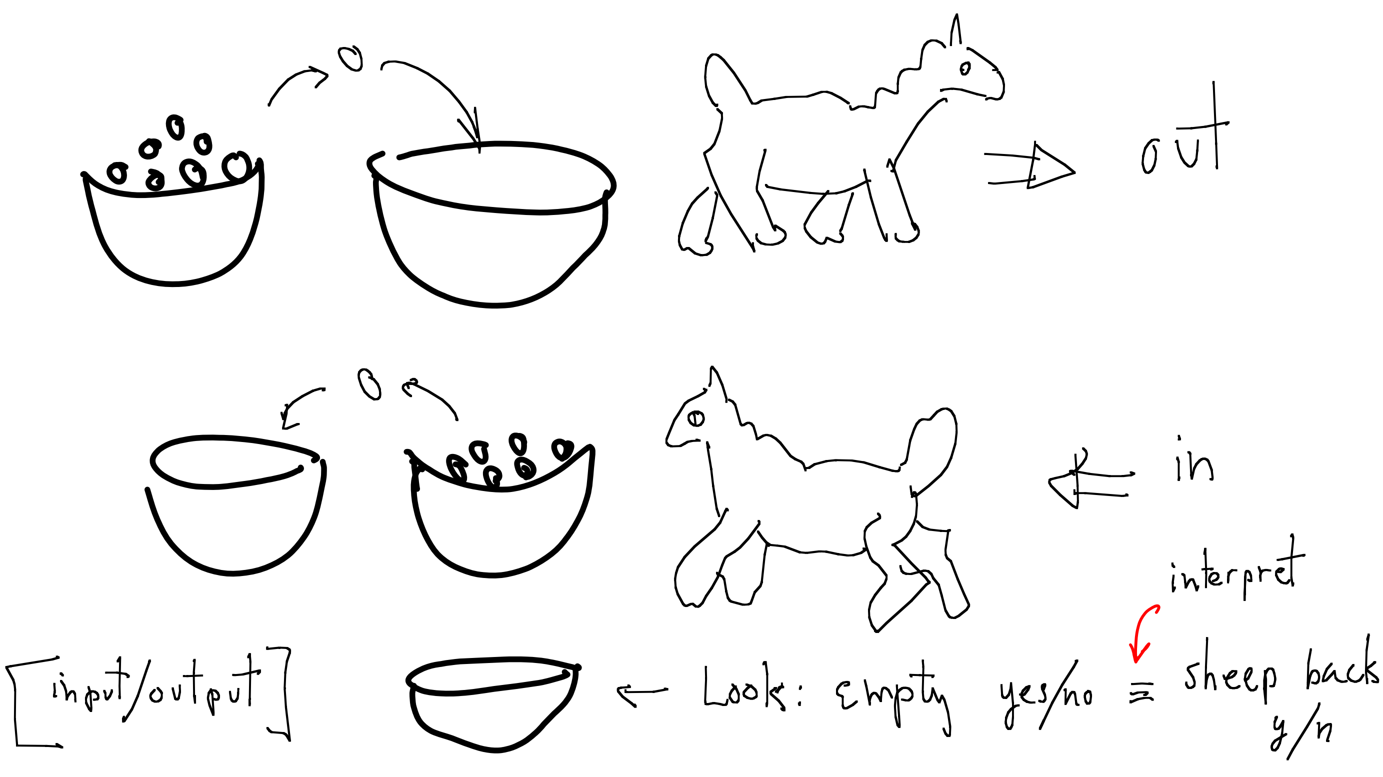
We would like to have:
-- A simple concept of computation/computing/computers
Why?
--- 1. When we build one, we can tell what we want: can it do what it is supposed to do?
--- 2. When we see one, we can recognize it (eg. is a QM machine a computer?)
--- 3. When we look at a complex system, we can identify its fundamental structure: abstraction.
--- 4. We can define what we mean by an algorithm (ie., TM that always halts).

Computer Science: The science of computing machines and what they can do?
--- Interesting provided there are interesting machines
--- What's interesting: big, fast, cheap (compared w/ value of what we want to do)
Big: lots of data
Fast: quick results (quick enough)
Cheap: to build, to run, to use.
-- Are there other "computing machines" out there?

==
Concept of computation

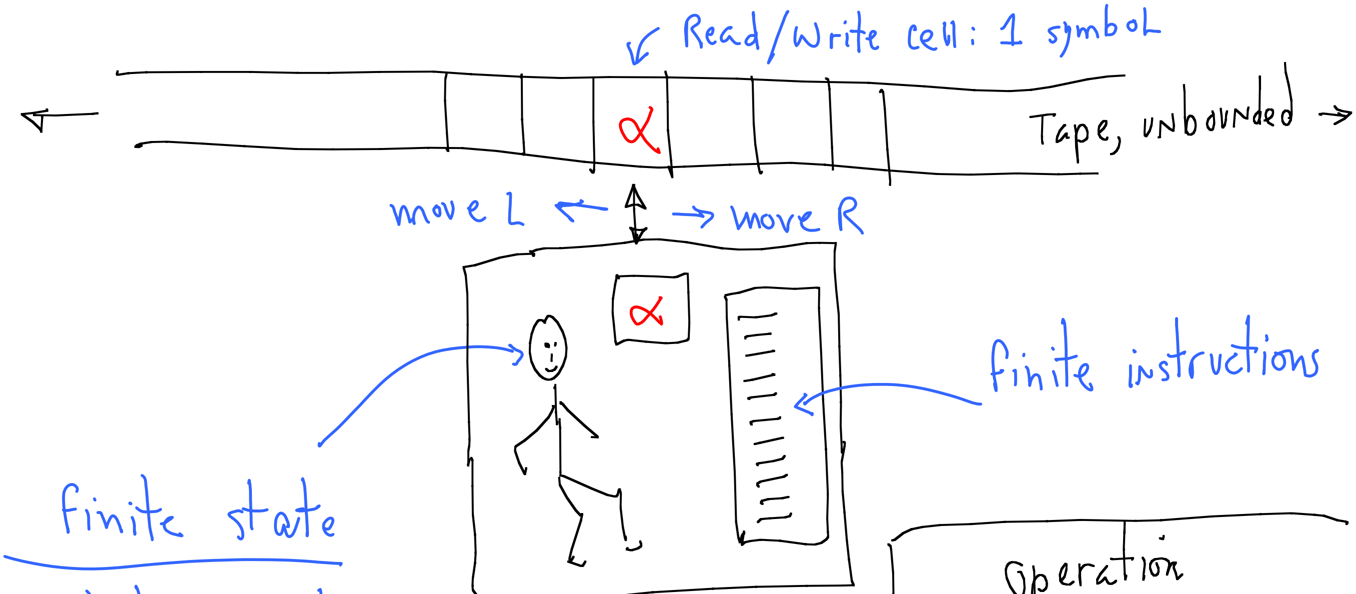
---- Information (what is it?)
---- Information transformation, results.
---- Answers questions that are interesting to us.

---- Symbolic computation: abstraction into symbols, manipulation of symbols.



BIG IDEA: Define computation (automatic procedure)

History: numbers, arithmetic, geometry (planets), abacus, Pascal, Babbage, others, ... Turing.



finite state

- what symbol
- what step in instructions

Finite set of symbols

e.g. {0, 1} or { #, a, b, c }

sufficient "blank"

USER Manual : correct tape input/output

Operation

A. read symbol

B. from present state figure

1. Symbol to write
2. next state
3. Direction to move

C. write symbol

D. change state

E. Move tape L/R one cell

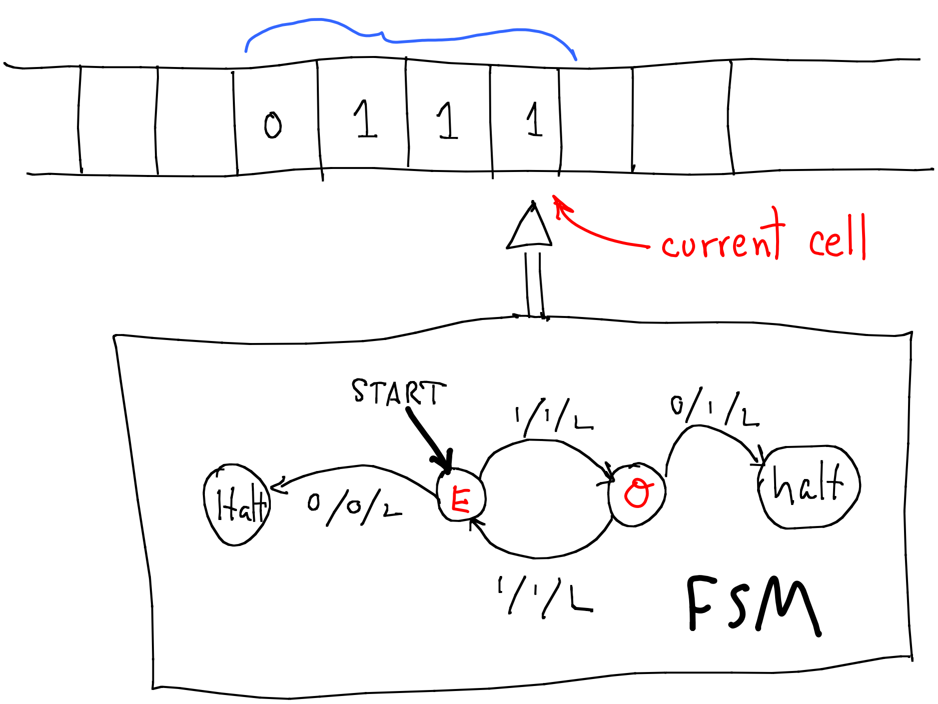
Church-Turing Thesis:

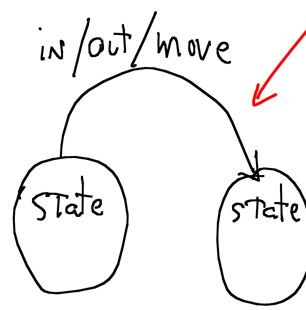
Any computation can be done by some Turing Machine (TM).

(Efficiently?)

Can't prove, but works so far.

State Transition diagrams





Alternate representations of FSM:

==================================
(start state)
(state, input) (output, move, next state)

(Even)
(Even, 1) (1, L, Odd)
(Even, 0) (0, L, Halt)
(Odd, 1) (1, L, Even)
(Odd, 0) (1, R, Halt)

==================================
A "unary" encoding of the above:

```
00100
  1  0 11 0 11 0  1  0  11  00
  1  0  1  0  1  0  1  0 111 00
 11 0 11 0 11 0  1  0   1   00
 11 0  1  0 11 0 11 0 111 00
00
```

Alphabet, aka symbol set

$\Sigma = \{0, 1, ※ ? \}$

functional view

$state^* = f(state, in)$

$out = g(state, in)$

$move = h(state, in)$

4

What does it do?

See LC3 FSM inputs & states

alternate description of FSM

OR use per-state func.s

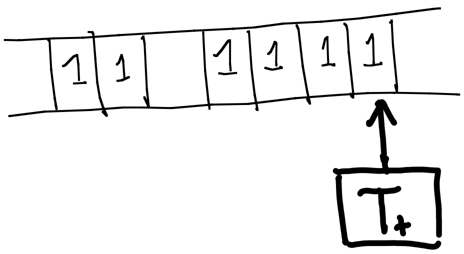
i = state

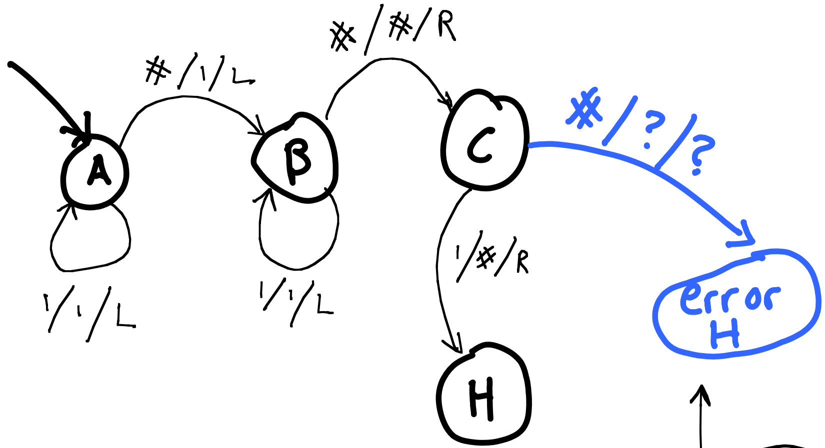
$f_{state_i}(in) = state_j$

$g_{state_i}(in) = output$

$h_{state_i}(in) = move$

Unary Adder | T_+





User Manual: Start with two numbers coded in unary on tape separated by a single blank, and RW-head positioned on rightmost 1 of righthand number. Machine halts with RW-head positioned at leftmost 1 of unary-coded result.

all possible inputs must be handled

Q. Does this work for input of 0 in one or both numbers?

Q. What is state A's purpose? B's? describe in words.

What else?

With a little more thought we can build:

--- Tu-: A unary subtractor
--- Tb+: A binary adder
--- Tb-: A binary subtractor
...

But, this is boring, whenever I want to do something new, I have to build a new machine.

BIG IDEA: Make a TM simulator (call it UTM)

--- UTM simulates any other machine A, if we put a description of A on UTM's input tape and layout A's input tape in a simulated tape encoding on UTM's tape.

--- Turing demonstrated one, and a way of describing machines (see below).

1.a. Pattern match A's current state w/ current-state part of rule.
1.b. If match, go to 2; otherwise, advance to next rule and go to 1.
2.a. Find the current location of the simulated RW-head,
    pattern match cell content with rule's input symbol.
2.b. If matched, go to 3. Otherwise, advance to next rule and go to 1.a.
3. Copy output symbol to current simulated tape cell.
4. Copy next-state symbol to current-state area.
5. Move simulated head as needed. Go to 1.

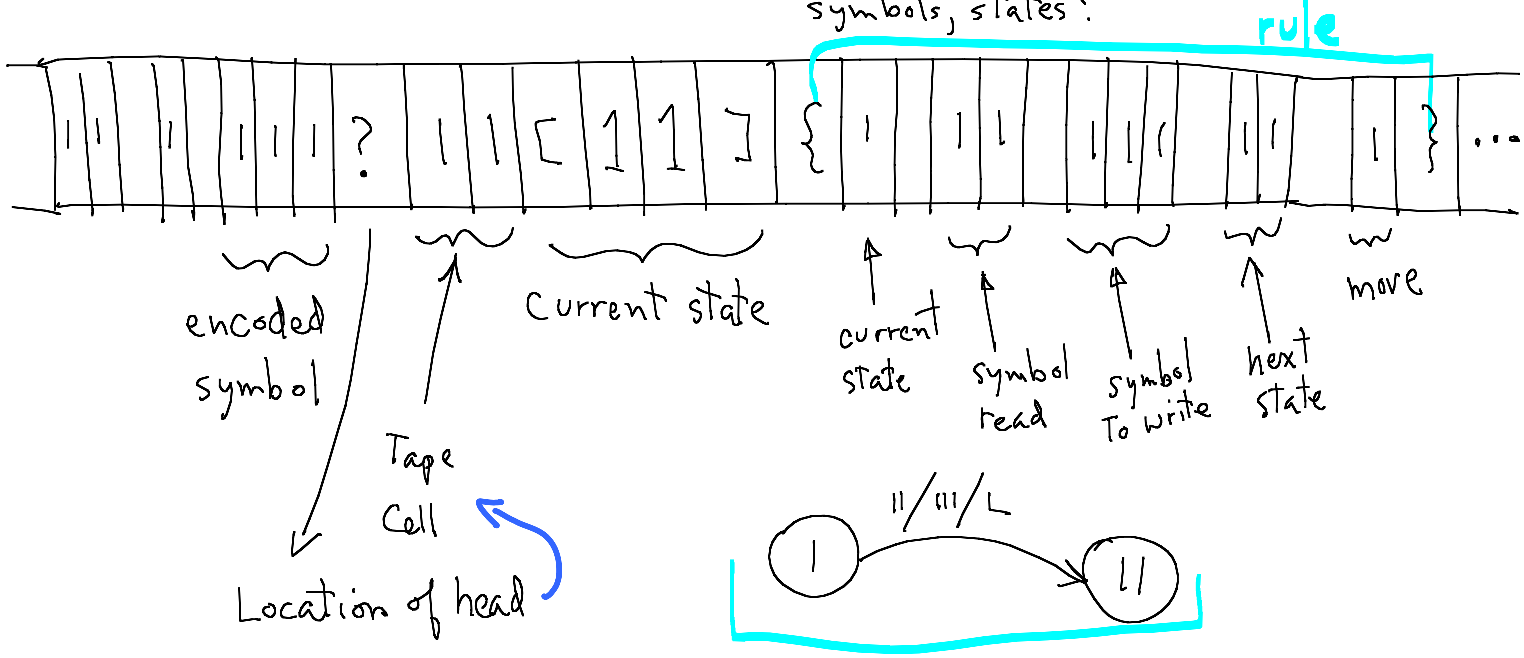
Built using some basic TMs: Tcopy, Tmatch, Tshifttape, ...

TM Description

How do we "describe" some M for UTM?

How many symbols do we need?

How many simulated machine symbols, states?



ENCODING

$$\Sigma = \{a, b, c, \dots\} \Rightarrow \{1, 11, 111, \dots\}$$

$$S = \{START, A, B, C, \dots\} \Rightarrow \{1, 11, 111, 1111, \dots\}$$

--- Uses a FIXED SYMBOL SET

--- BUT, Can encode any size symbol set.

Can encode characters, numbers, strings, images, ...

--- Has FIXED NUMBER OF STATES,

--- BUT, simulates machines w/ any number of states.

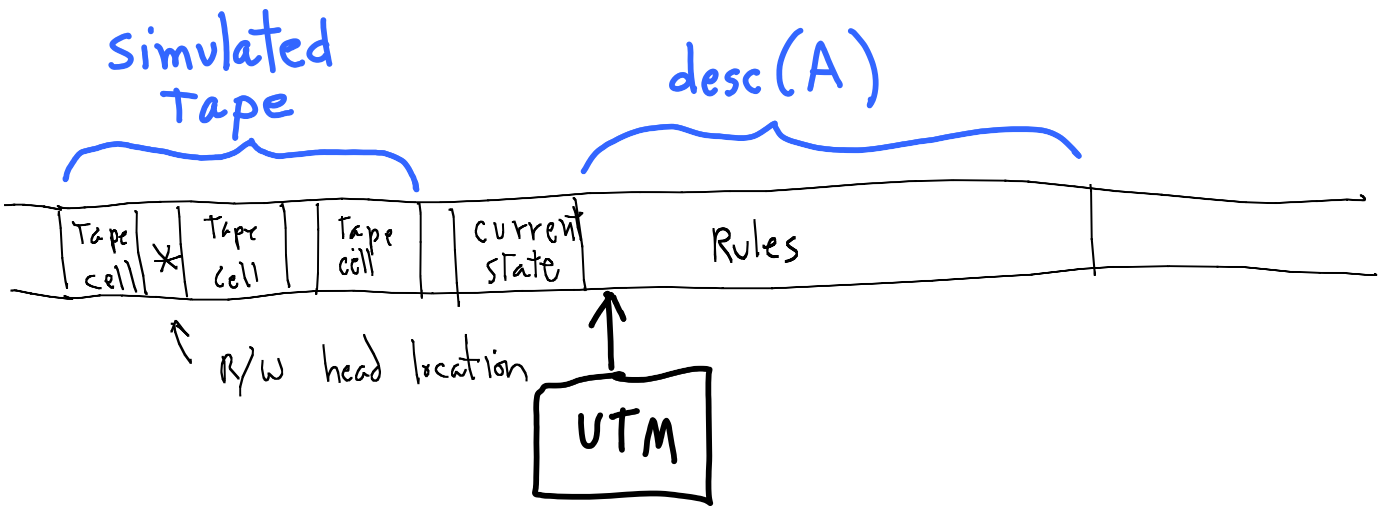
--- Uses a BOUNDED AREA OF TAPE,

--- BUT, relocates to expand simulated tape as needed.

--- Uses PATTERN MATCHING, not states to determine a match between state or symbol codes.
(Using states to count would limit number of simulated symbols/states possible.)

--- Any universal machine could be used, with larger symbol sets (say, binary integers), can encode more economically.

N bit integers $\Rightarrow$ 2^N symbols, 32-bit words $\Rightarrow$ 4G symbols



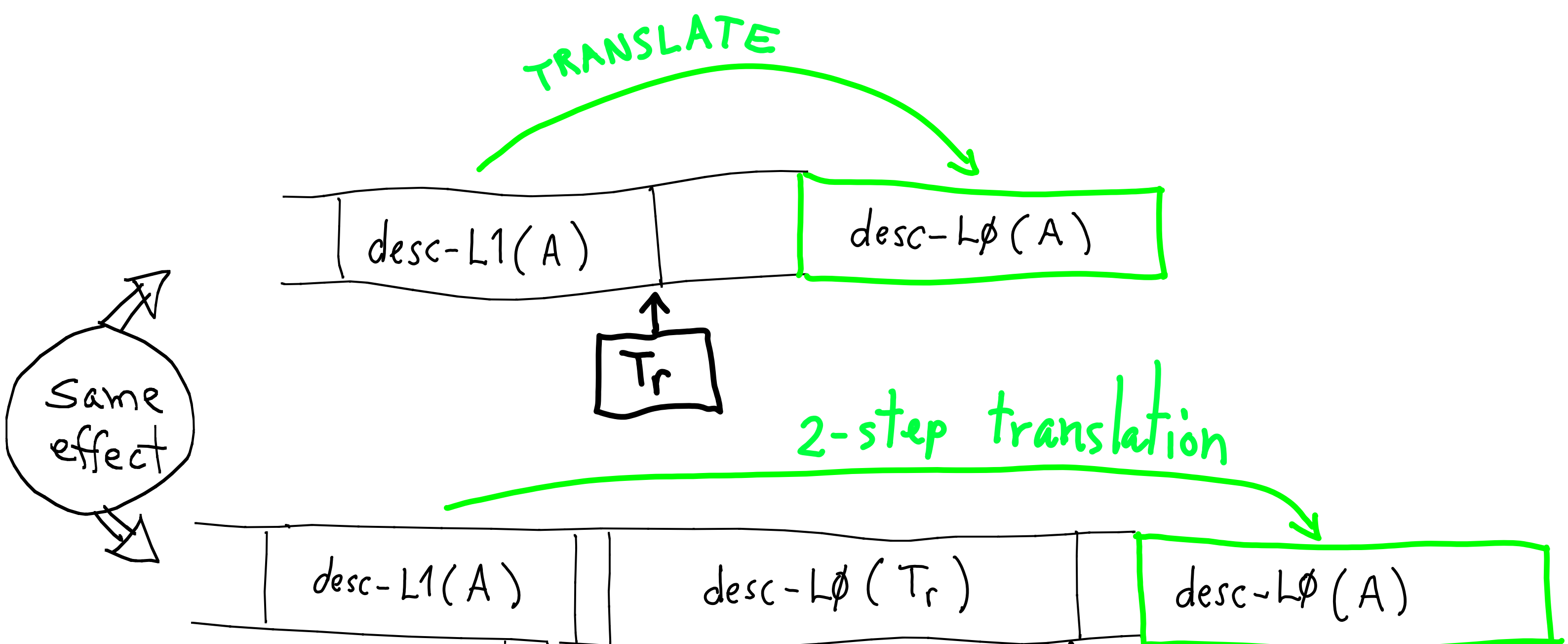
Programmability and Translation

Can simulate any TM, no matter the size of its symbol set, the number of states, or the number of rules (UTM uses pattern matching, not counting via UTM states).

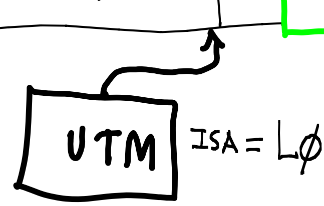
The description is written in the encoding for that UTM (there can be more than one UTM, each using a different encoding for machines/tape/symbols...)

The description language is the "machine language" for that UTM.

Q. Can we write a language translator that can be simulated by UTM?
--- Call UTM's machine language, L0. Suppose there is another language, L1, that can be used to describe TMs. Is there a translator machine T10?



Can we do this another level deeper? That is, start with desc-L2(A), simulate Tr-2-1 to get desc-L1(A), then use Tr? What language is desc(Tr-2-1) written in?



BIG IDEA: machine descriptions as input data.

--- Translate between descriptions: C++ => C => ASM => machine language (ISA)

--- Ask questions about Algorithms/Procedures/TMs using desc(M):

Given machine M and input x, will xM ever halt? (read "xM" as "x operated on by M").

what about

- ☐ 2-way infinite tapes?
- ☐ 2d tapes?
- ☐ RAM tape?
- ☐ Multiple R/W heads?
- ☐ Alphabet (Symbol set)?

No difference in computational capability!
(Maybe faster, that's all)

$\Rightarrow$ 2 symbols (min) $\{0,1\}$ or $\{\#, 0, 1\}$

Why not use REALLY HUGE symbol sets?

32-bit word => 4 Giga-symbol (4 Billion)

64-bit word => 16 Exa-symbol (16 Billion Billion)

[See "The Myth of the Turing Machine", C. Eliasmith]
[See "Turing Machines", Stanford Encylopedia of Philosophy]