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).

BIG IDEA: Define computation (automatic procedure)

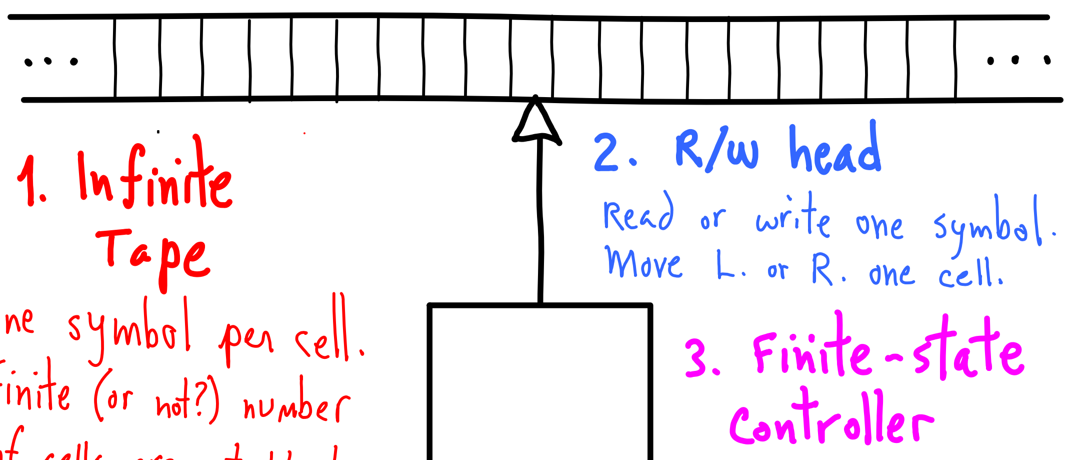
Church-Turing Thesis:

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

(Efficiently?)

Can't prove, but works so far.

TM model of Computation



1. Infinite Tape

One symbol per cell. Finite (or not?) number of cells are not blank.

2. R/W head

Read or write one symbol. Move L. or R. one cell.

3. Finite-state Controller

In one state at a time. Finite number of states. Changes state deterministically, next state depends on symbol read and current state. Specific start state.

4. Finite symbol set, Σ.

For example,

$\Sigma = \{0, 1\}$

or

$\Sigma = \{\#, 0, \ldots, 9, a \ldots z\}$

(# ← blank)

5. User manual

Defines:

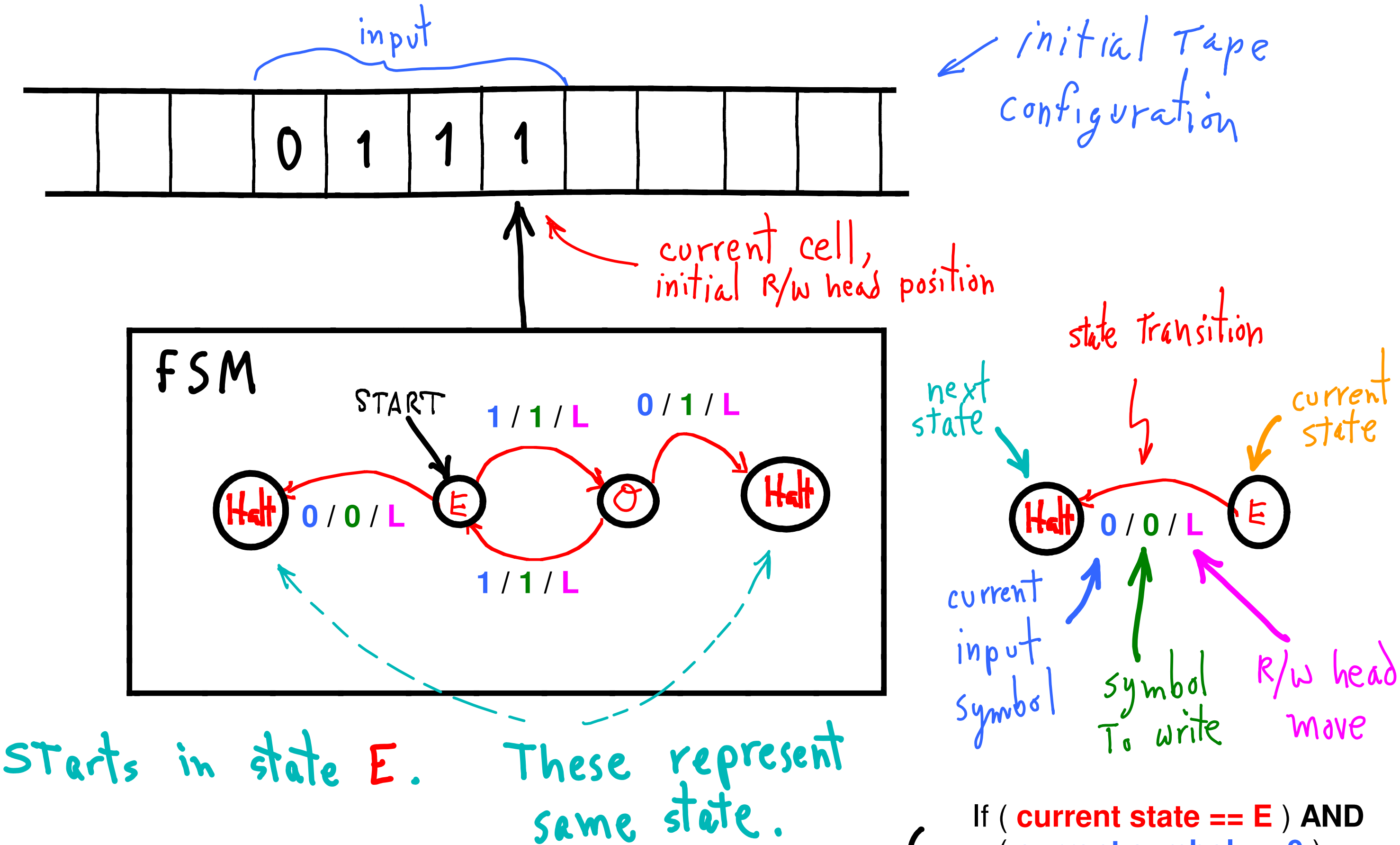
1. Proper input
2. Proper output
3. Interpretation
4. Halting states
5. Start state

Operation

1. Read one symbol
2. Depending on state and symbol,
 --- write one symbol
 --- move R/W head one cell L. or R.
 --- change state
3. Repeat, or if current state is a "halting state", stop operating.

Finite-State Machine Controller

State Transition diagrams



a state-transition rule →

If (**current state == E**) **AND**
(**current symbol == 0**)
then
(**write symbol == 0**)
(**move R/W head L.**)
(**new state == Halt**)

What does it do?

On this specific input?

On any general input?

Alternate representation of FSM:

(start state = E)

(state, input)	(output, move, next state)
(E, 1)	(1, L, O)
(E, 0)	(0, L, Halt)
(O, 1)	(1, L, E)
(O, 0)	(1, L, Halt)

Same information as diagram.

→ An encoding of the FSM.

	code
State E	1
State O	11
State Halt	111
Symbol "0"	1
Symbol "1"	11
move L	1
move R	11

START STATE

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
```

end of description — current state — current symbol — output — move — next state

That "unary" encoding can be a binary number:

0010010110110101100101010101110011011011010100110101101101110000

still has same information

A functional view of FSM

state	symbol	F = next-state
E	0	Halt
E	1	O
O	0	Halt
O	1	E
Halt	0	Halt
Halt	1	Halt

next-state function

Global functions

F : { **states** } X { **symbols** } ===> { **states** }

G : { **states** } X { **symbols** } ===> { **symbols** }

H : { **states** } X { **symbols** } ===> { **L, R** }

Local functions

Each state has 3 functions.

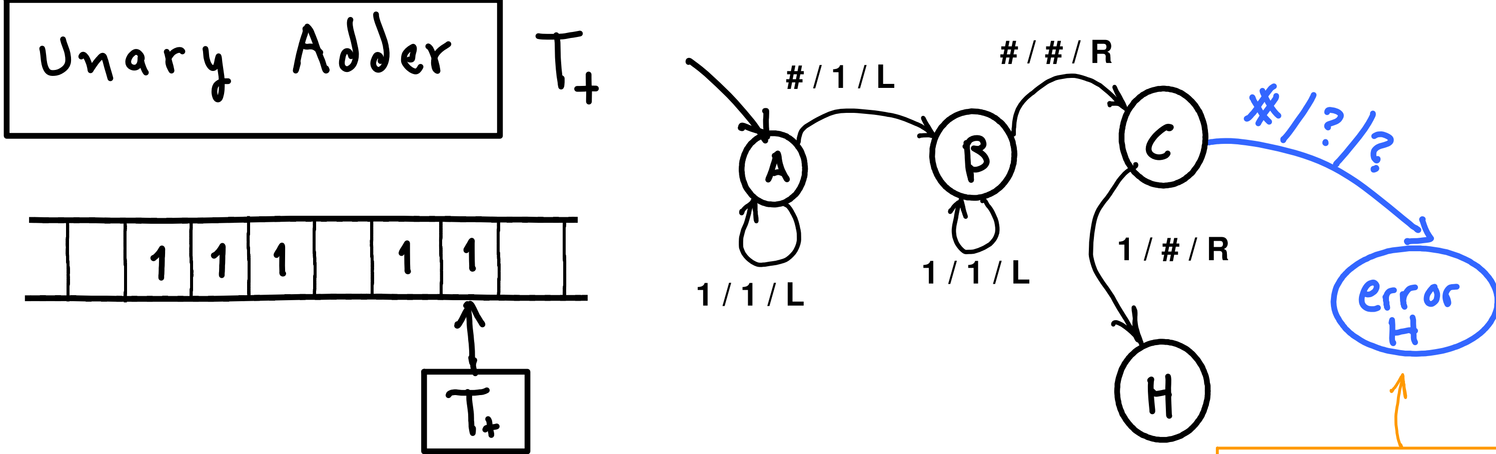
E.g., functions for state e

F_e : { **symbols** } ===> { **states** }

G_e : { **symbols** } ===> { **symbols** }

H_e : { **symbols** } ===> { **L, R** }

(We can describe these with our programming language.)



All possible inputs must be specified **for every state**. But we can agree that unspecified symbols go to "**error**" state.

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.

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

Q. What is the purpose of state **A**? State **B**? State in words.

What else?

With a little more thought we can build:

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

This gets boring. What else can we do?

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.

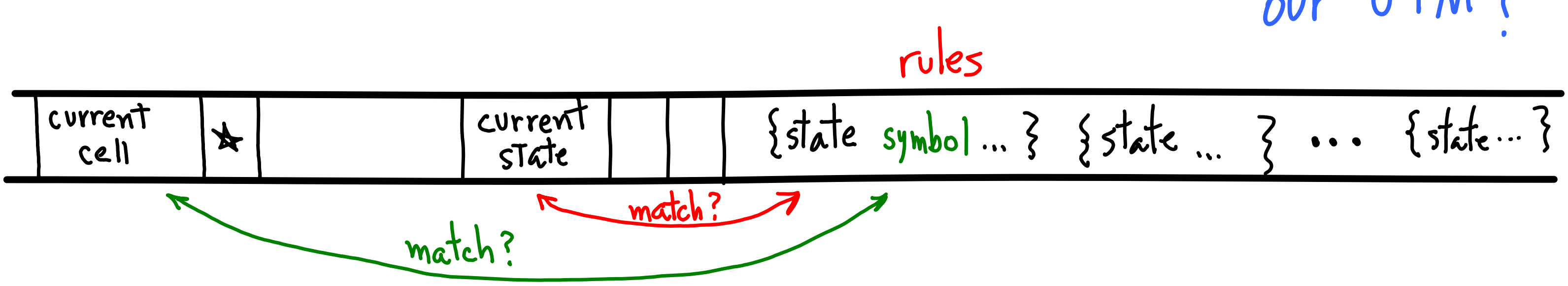
--- Turing demonstrated one, and a way of describing machines (see below).

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.a.
2.a. **Find 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. **Fix** simulated tape. Go to 1.a.

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

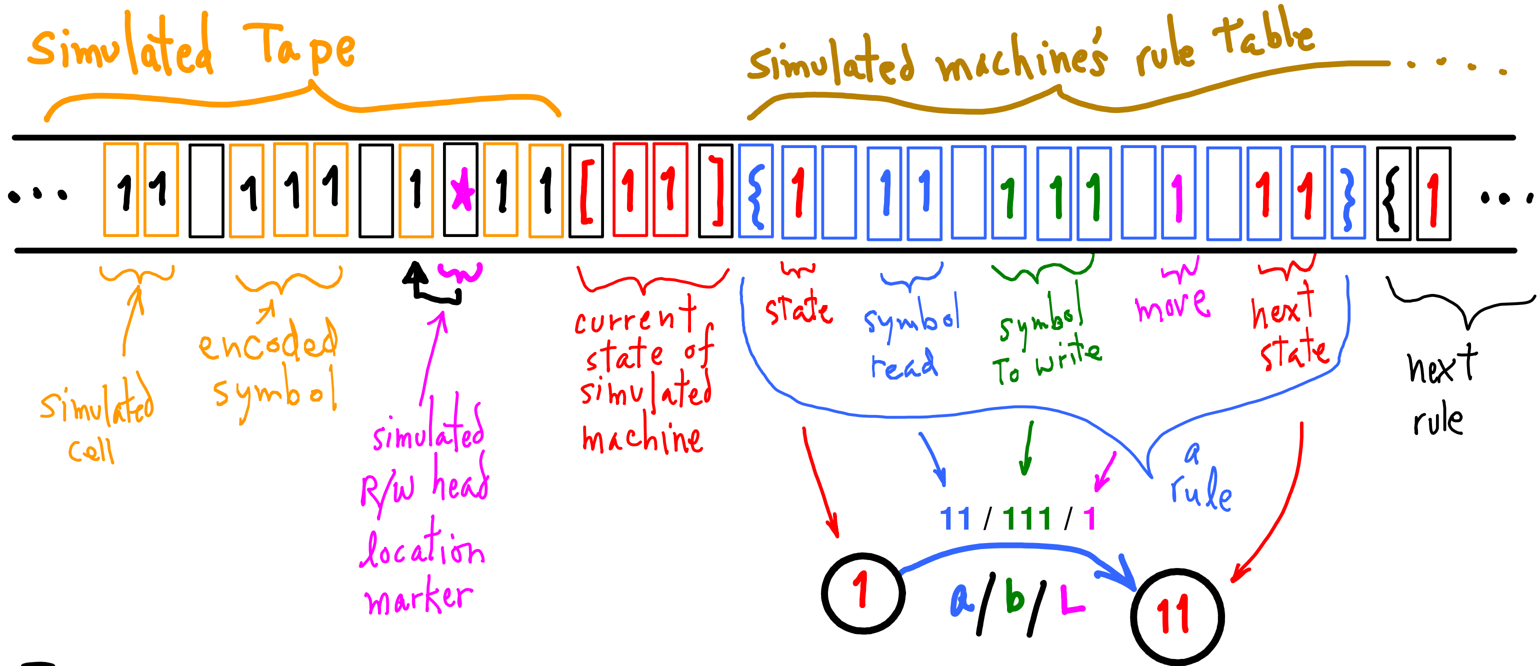
Big Question

Can we do this w/ a fixed number of states in our UTM?



TM Description

How do we describe an arbitrary M?
How many symbols does our UTM have?
How many symbols does M have? states?



ENCODING, e.g.,

$\Sigma = \{0, a, b, \dots\}$ $\rightarrow \{1, 11, 111, \dots\}$

States = {Start, A, B, ...} $\rightarrow \{1, 11, 111, \dots\}$

SIMULATING MACHINE (UTM)

--- Uses a **FIXED SYMBOL SET**
--- BUT, **Can encode any size symbol set** (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 and **expands simulated tape as needed**.

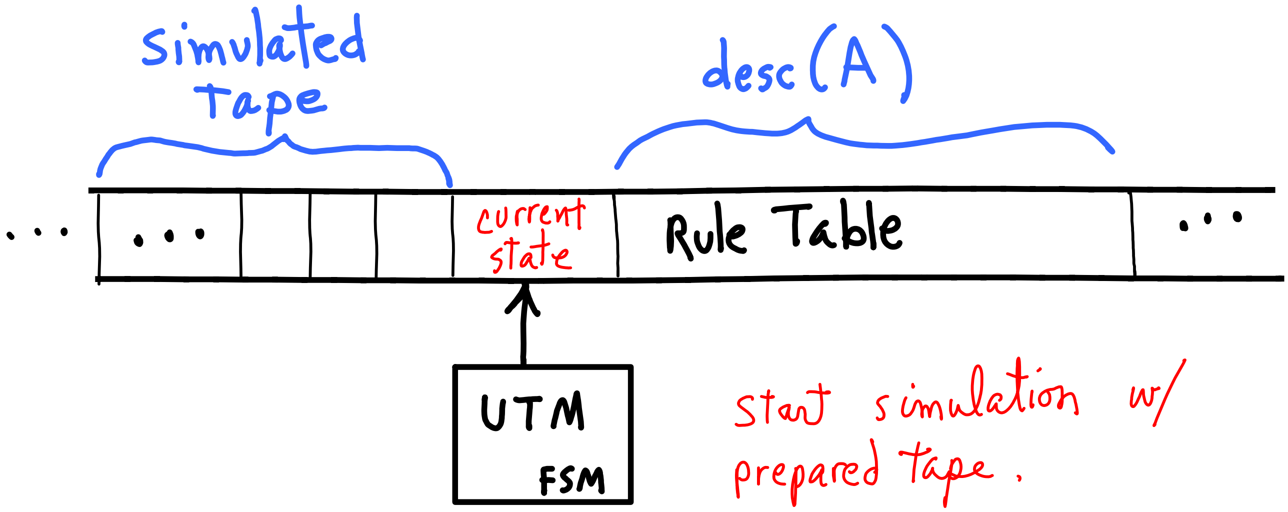
--- Uses **PATTERN MATCHING**, not states to match **state or symbol codes**.
(Using states to count would limit number of simulated symbols/states possible.)

--- A universal machine with a larger symbol sets (say, binary integers), could encode more economically.

n-bit integers $\rightarrow 2^n$ symbols, 32-bit integers $\rightarrow 2^{32}$ symbols

$= 2^2 \cdot 2^{10} \cdot 2^{10} \cdot 2^{10}$

$= 4 \cdot 1k \cdot 1k \cdot 1k = 4G$ symbols



Programmability and Translation

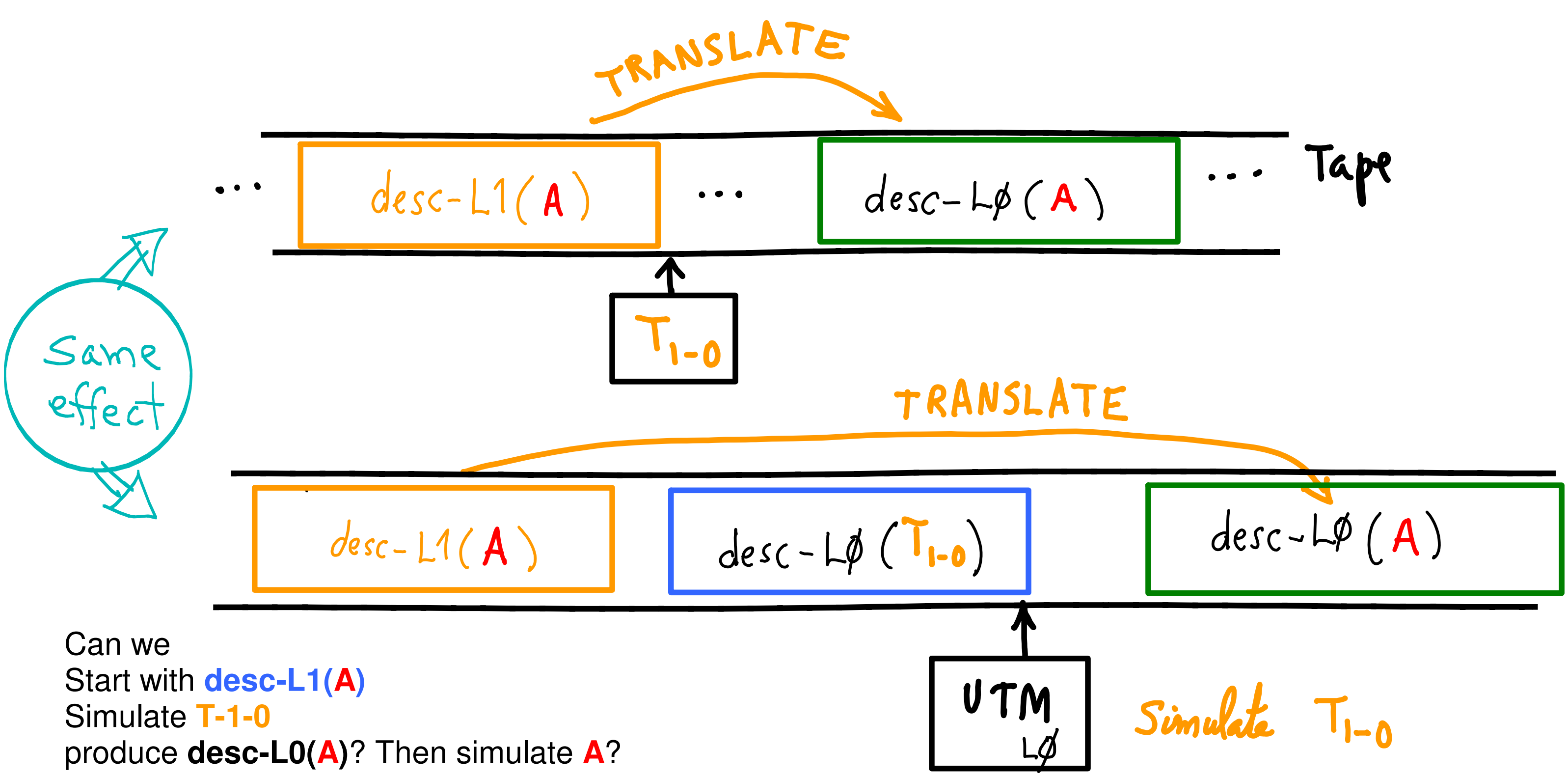
Can simulate any TM, no matter the size of its symbol set, the number of states, or the number of rules (UTM uses pattern matching, not counting via UTM states). Extra credit: prove.

There can be **more than one UTM**, each **using different encodings** for machines/tape/symbols...

The **description language** is the "**machine language**" **for a particular UTM**.

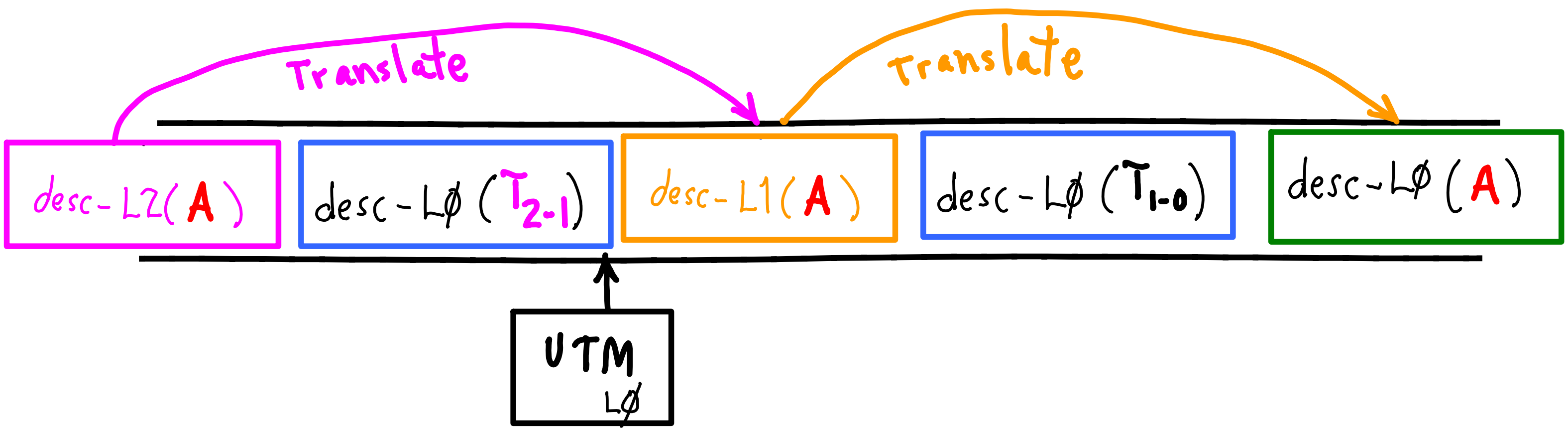
Q. Can we describe a **language translation machine** and **simulate it** using our **UTM**?

Call our UTM's **machine language**, **L0**. Suppose there is **another universal language, L1**, that can be used to describe TMs. **Is there a translator TM, T-1-0, that translates a description of machine A encoded in L1 and produces a description of A encoded in L0**?

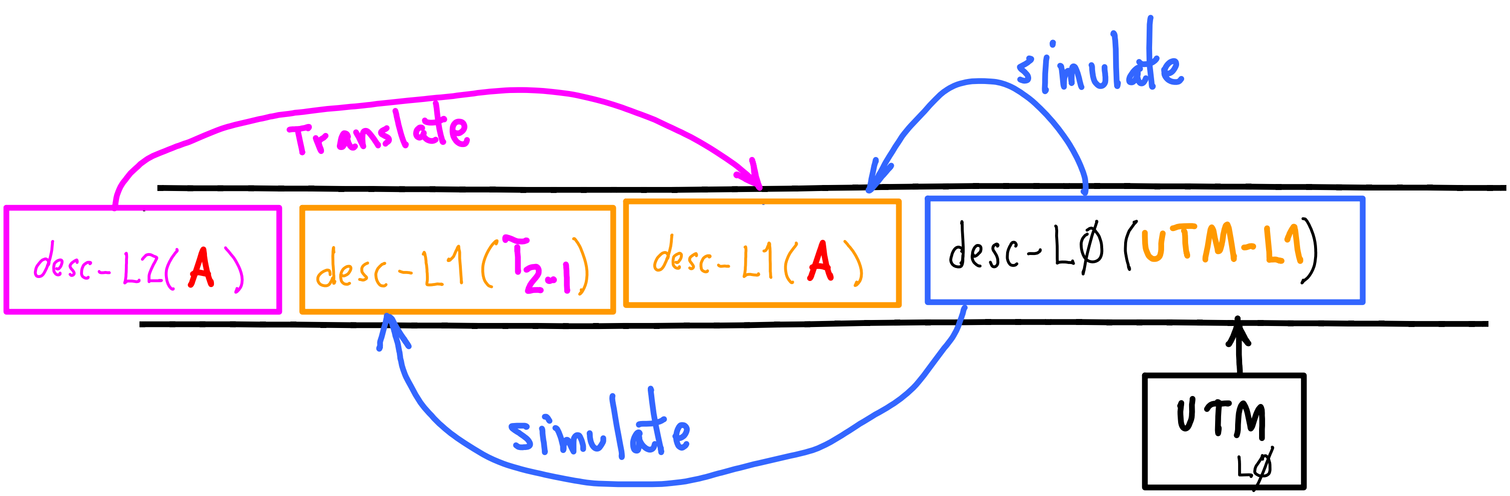


Can we
Start with **desc-L1(A)**
Simulate **T-1-0**
produce **desc-L0(A)**? Then simulate **A**?

How about
--- start with **desc-L2(A)**,
--- simulate **T-2-1** using **desc-L0(T-2-1)**
--- to get **desc-L1(A)**
--- simulate **T-1-0** using **desc-L0(T-1-0)**
--- to get **desc-L0(A)**
--- then simulate **A**?



How about
--- start with **desc-L2(A)**,
--- simulate **T-2-1** using **desc-L1(T-2-1)**
--- by simulating **UTM-L1** using **desc-L0(UTM-L1)**
--- to get **desc-L1(A)**
--- simulate **A** by simulating **UTM-L1 reading desc-L1(A)**



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

- ☐ 2d tapes?
- ☐ RAM tape?
- ☐ multiple R/W heads?
- ☐ alphabet (symbol set)?

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

⇒ 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)