MAKING MINDS

A Constructive Introduction to the Computational Theory of Mind

Gabriel Greenberg

9.20.2026

"I never satisfy myself until I can make a mechanical model of a thing. If I can make a mechanical model, I can understand it."

— Lord Kelvin, 1884

Table of Contents

Unit 1: COMPUTATION

Unit 2: LEVELS

Unit 3: COGNITION

Unit 1

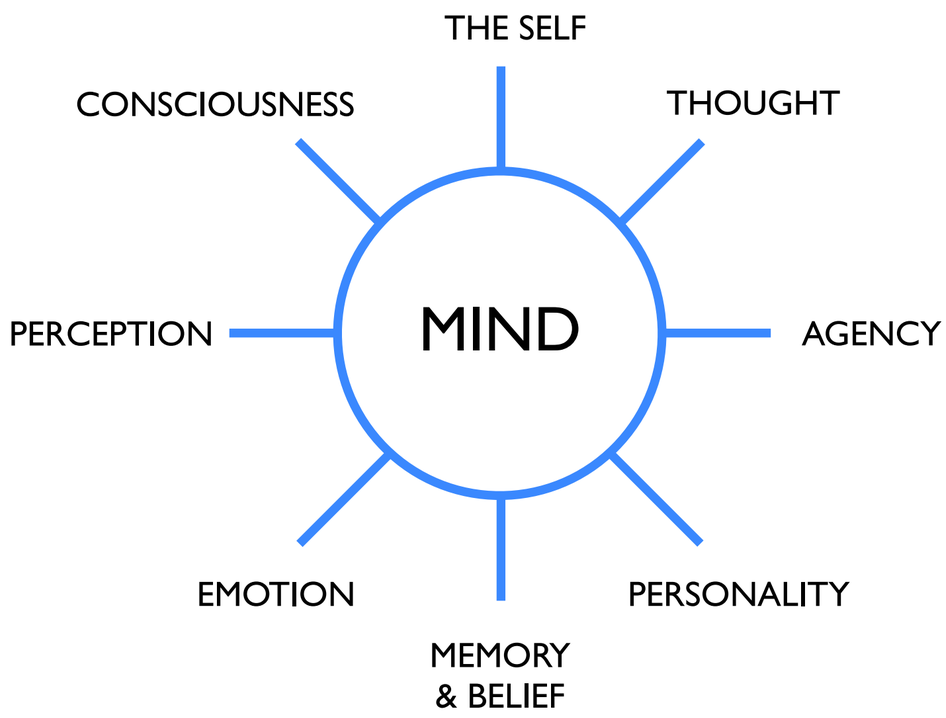
COMPUTATION

1. The Cognition-Body Problem

The Mind

What is a mind? What is *your* mind? There is no simple definition. The mind is the unity of **psychological** capacities, processes, and states.

Mental states are states of awareness, thinking, perceiving, feeling, hoping, imagining lying on the beach, and so on.



Dualism & Physicalism

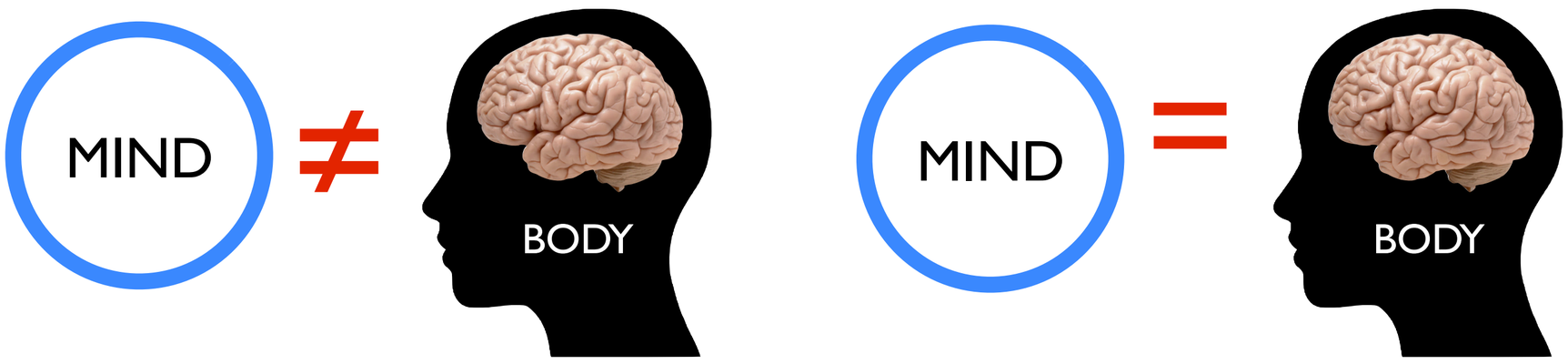
How does your own inner mental states fit into the objective physical universe? How do we reconcile mind and nature? The debate between two answers to this question is one of the deepest metaphysical debates in the history of philosophical thought.

Dualism

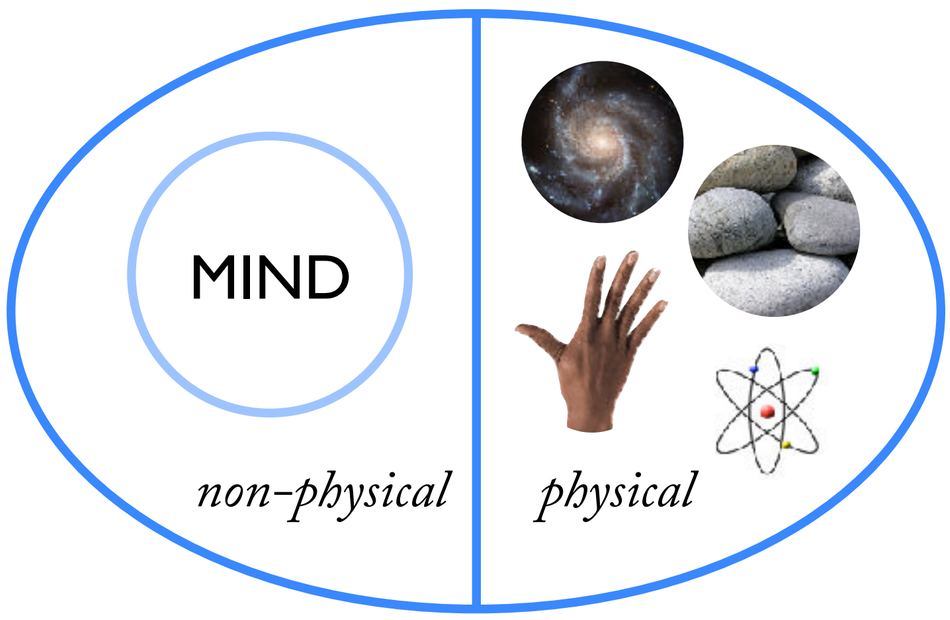
The mind is distinct from the body. No mental state is realized by a physical state.

Physicalism

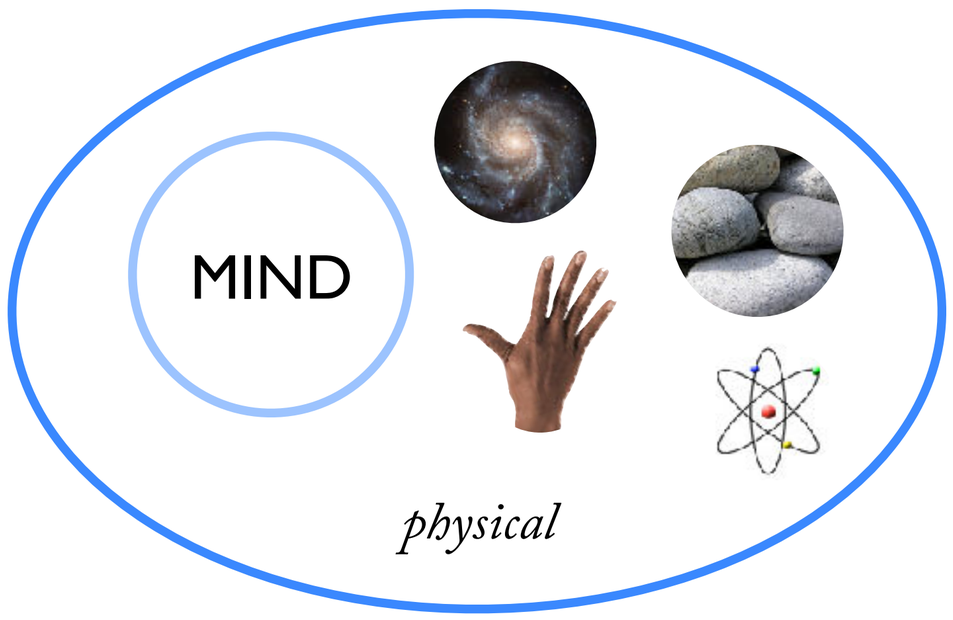
The mind is (part of) the body. Every mental state is realized by a physical state.



Add: motivations for physicalism.



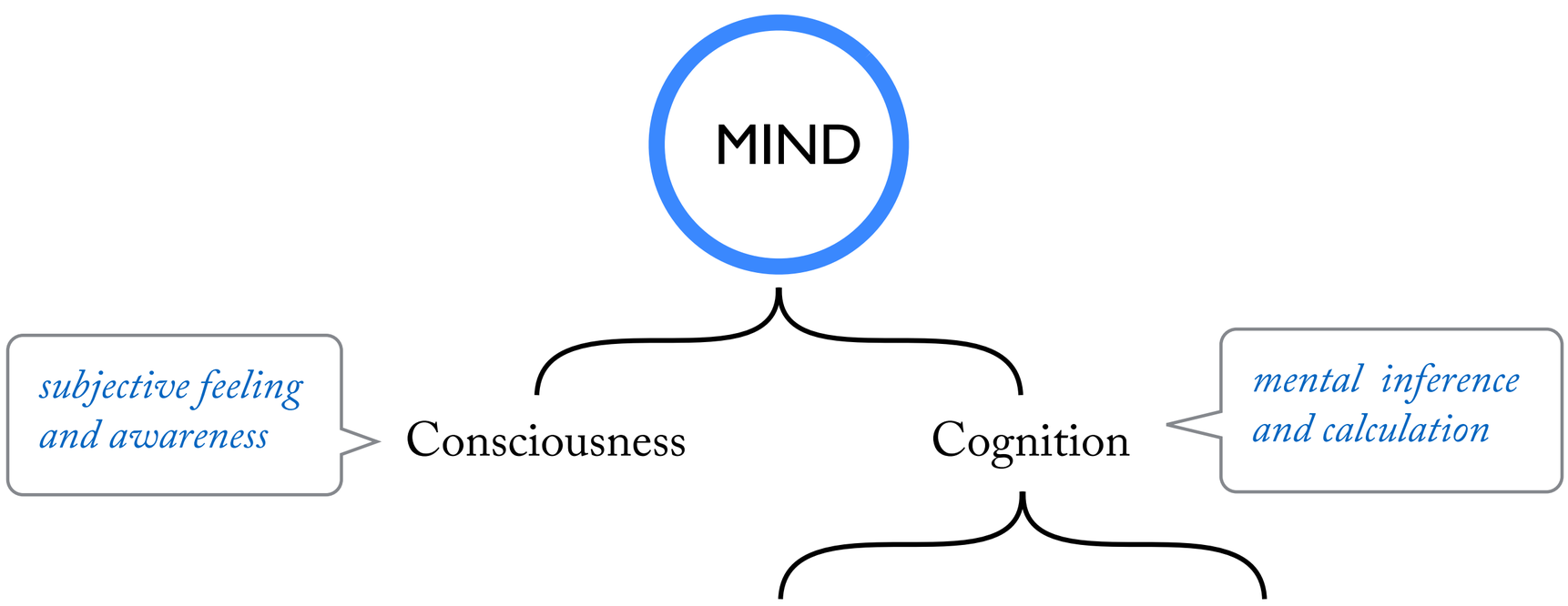
Reality according to **Dualism**



Reality according to **Physicalism**

Cognition & Consciousness

Two fundamental properties seem to distinguish minds from the rest of the non-mental universe. One is **consciousness**. The other is **cognition**.



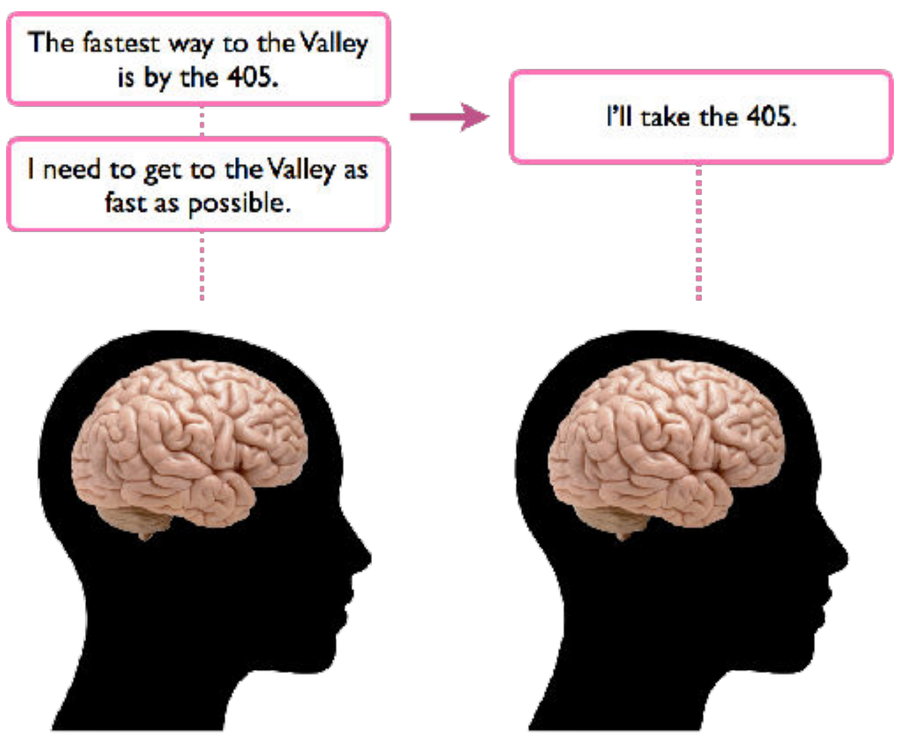
Contentful State:
States possessing content.

content

Rationality:
Rule-governed transitions between contentful states.

content → content

A system has **cognition** if it undergoes rational transitions between contentful states.



A system undergoes **rational transitions** when it finds rational solutions to problems.

The **content** of a state is whatever it is "about" or "of".

Individual words have particular individuals or properties as contents. Entire sentences have whole situations as their contents.

Examples of Mental States with Content

Belief: A believes *he is on Mars.*
Wish: B wishes *she could visit Mars.*
Know: C knows *she owns a G-Class.*
Think: D thinks *it's Tuesday.*
Learn: E learns *a dance move.*
See: F sees *a bicycle.*

Examples of Rationality

Logic: All X are Y. Q is X. → Q is Y.
Math: 3 + 4 → 3 + 4 = 7
Navigation: Maze → Path
Decision-Making: I want to see a movie. I like action movies. → I'll see an action movie.
Perception: [Light image] → It's a duck!
Prediction: Initial conditions → final conditions.

The Cognition-Body Problem

Leibniz (1646-1716) famously challenged the idea cognition could emerge from purely physical elements. His objection is known as **The Windmill Argument:**

> One is obliged to admit that *perception* and what depends upon it is *inexplicable on mechanical principles*, that is, by figures and motions. In imagining that there is a machine whose construction would enable it to think, to sense, and to have perception, one could conceive it enlarged while retaining the same proportions, so that one could enter into it, just like into a windmill. Supposing this, one should, when visiting within it, find only parts pushing one another, and never anything by which to explain a perception. Thus it is in the simple substance, and not in the composite or in the machine, that one must look for perception.
>
> Leibniz 1714, *Monadology*

The argument has more or less the following form:

The Windmill Argument

Premise 1: Basic physical elements do not have cognition.

Premise 2: If a set of objects do not have cognition, then nothing composed of those objects alone has cognition.

Therefore: Nothing composed purely from basic physical elements has cognition.

Dualism is one conclusion to draw from this argument. If the premises are true, the argument implies that cognition is not physical. If it isn't physical, cognition must be the activity of a non-physical mind or soul.

Panpsychism is another possible reaction: deny Premise 1, and insist that even basic physical elements have cognition. In fact this was Leibniz' own view. (He called these basic minds "monads".) We won't discuss this interesting idea in this course.

Physicalism (or materialism) is the view that the mind (including cognition) *is* physical. Physicalists accept Premise 1, but deny Premise 2. They believe that non-cognitive parts *can* give rise to a complex whole which has genuine cognition.

The constructive challenge: physicalism is a promising *idea*, but how could you actually *make* a thinking thing from material parts? Where would you even begin?

The Computational Theory of Mind (CTM) is one version of physicalism. It seeks to understand the mind as the computational expression of the brain. CTM is more than just a philosophical idea, it is also the foundational assumption of cognitive science. The purpose of this course is to investigate CTM's answer to the cognition-body problem primarily as a philosophical and logical idea, and secondarily as an empirical hypothesis.

The Trajectory of this Course

This course is a sustained physicalist response to Leibniz' challenge. Over 10 weeks we will try to see how far we can get, starting with the simplest non-thinking parts, and building towards more and more complex systems. The end goal is to design a organism whose physical construction is completely clear to us, but which seems to exhibit limited cognition. Our path to that destination is the computational approach to the mind.

The Approach of this Class

- One foot in computer science.
- One foot in cognitive science.
- One foot in philosophy.

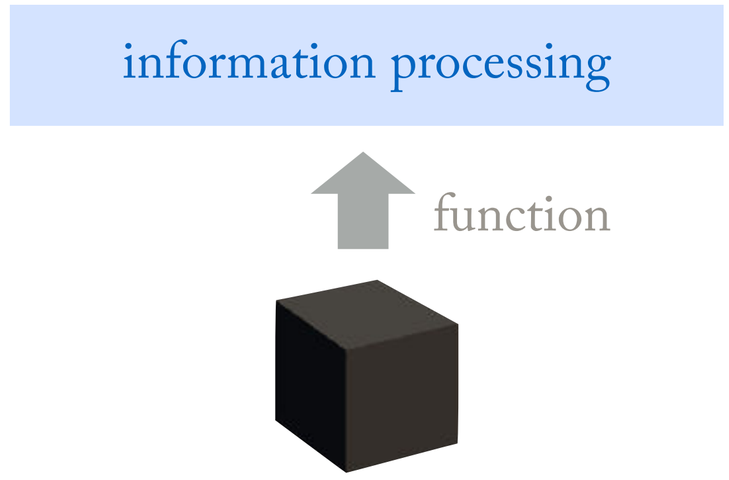
Course outline:

- Unit 1: What is computation?
- Unit 2: Levels of abstraction, Levels of power
- Unit 3: Computation as cognition

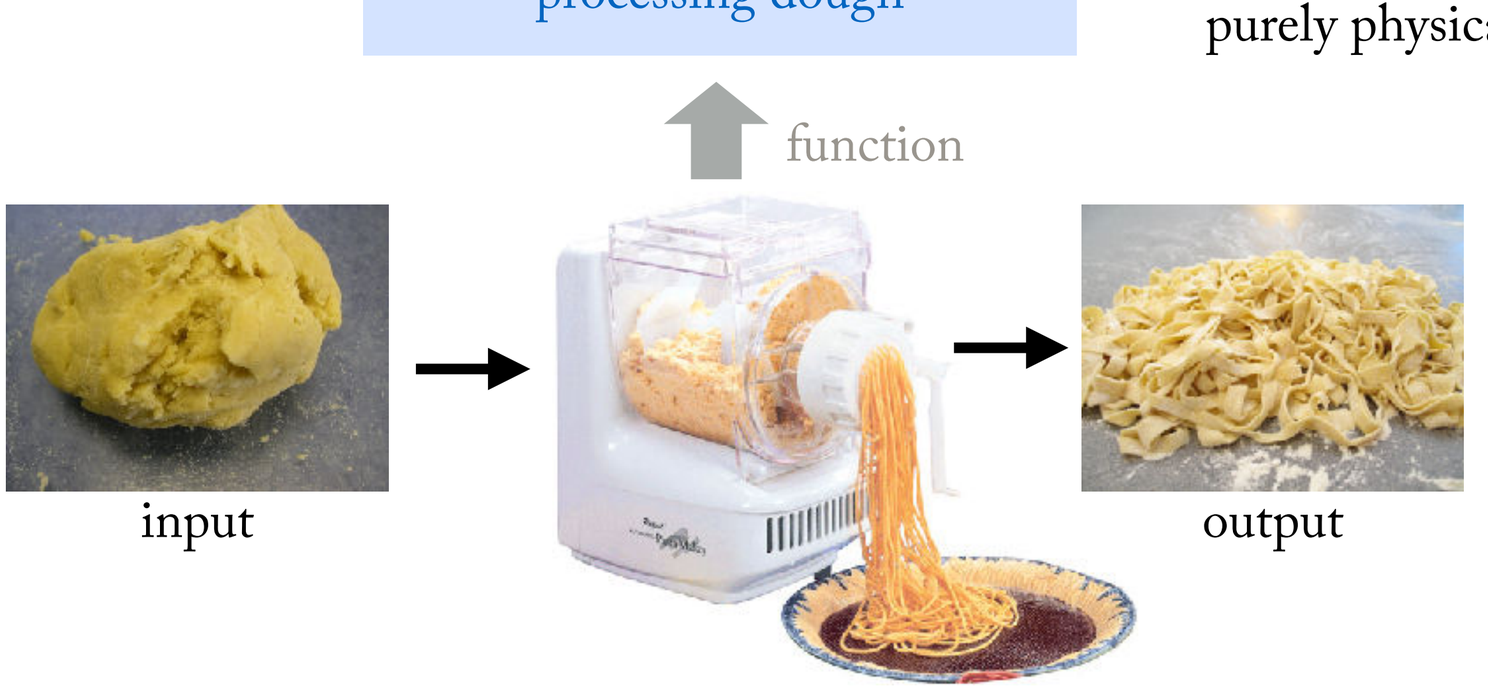
2. What is a computer?

1. Information Processing

A computer is a machine that has the function of **processing information**.

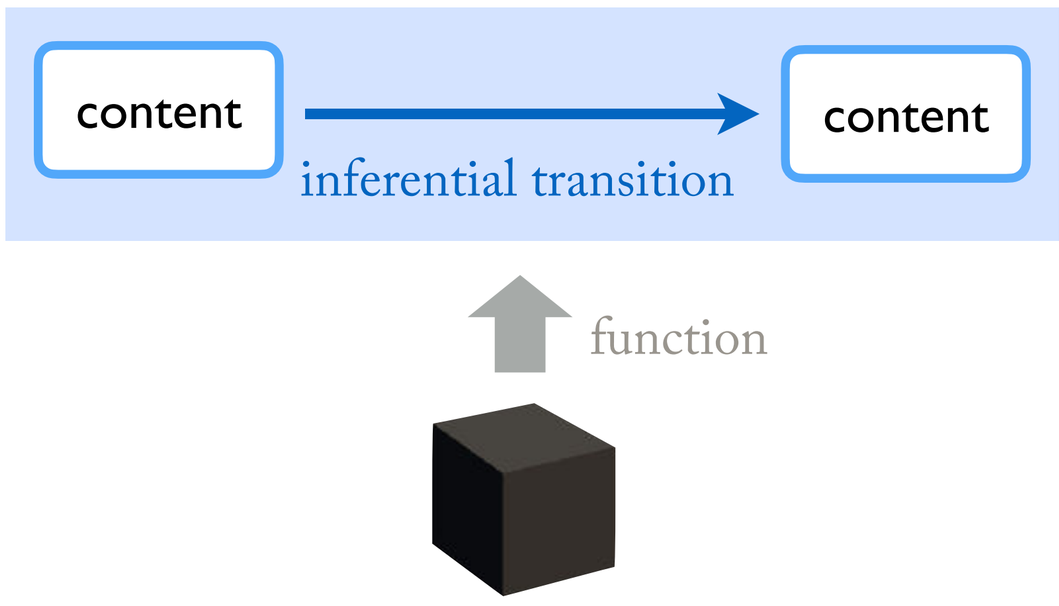


Like any other machine, it has a definite function…



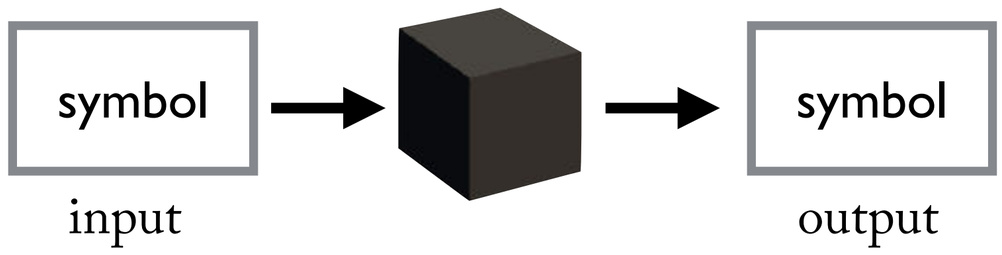
… but the function of a computer isn't something that can be described in purely physical terms.

To process information is to perform **inferential transitions** between **contentful states.**



2. Symbol Processing

A computer takes physical signals as **input** and produces physical signals as **output**.

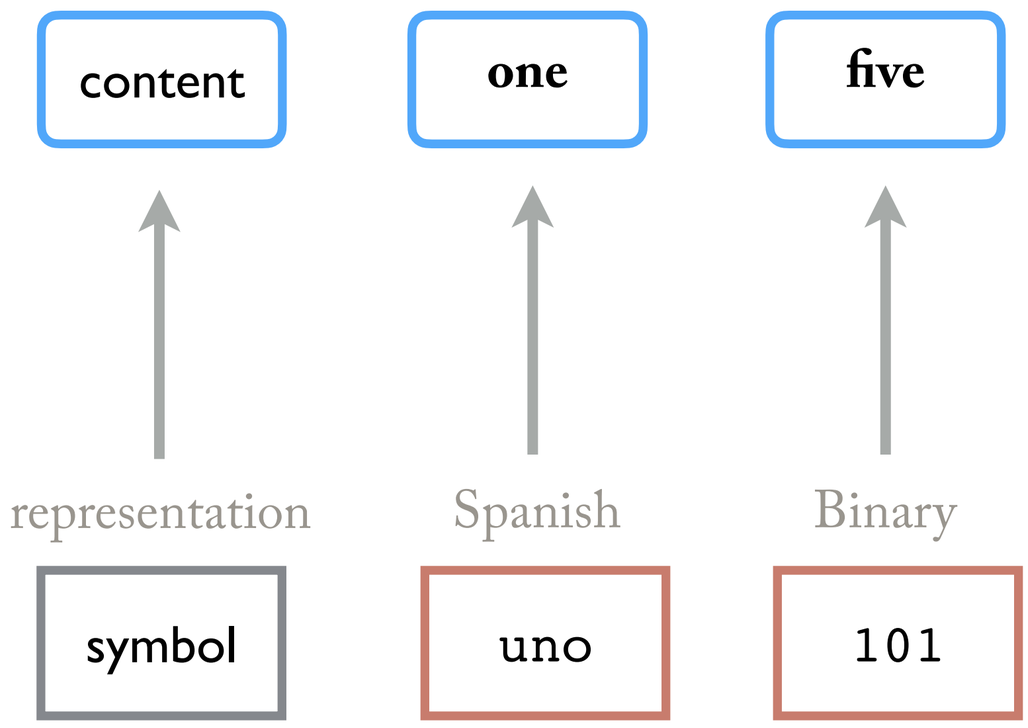


Symbols are physical objects or patterns. A computer takes symbols as inputs, processes symbols, and produces symbols as outputs.

A **representation** is a physical symbol that is *about* something or expresses a meaning.

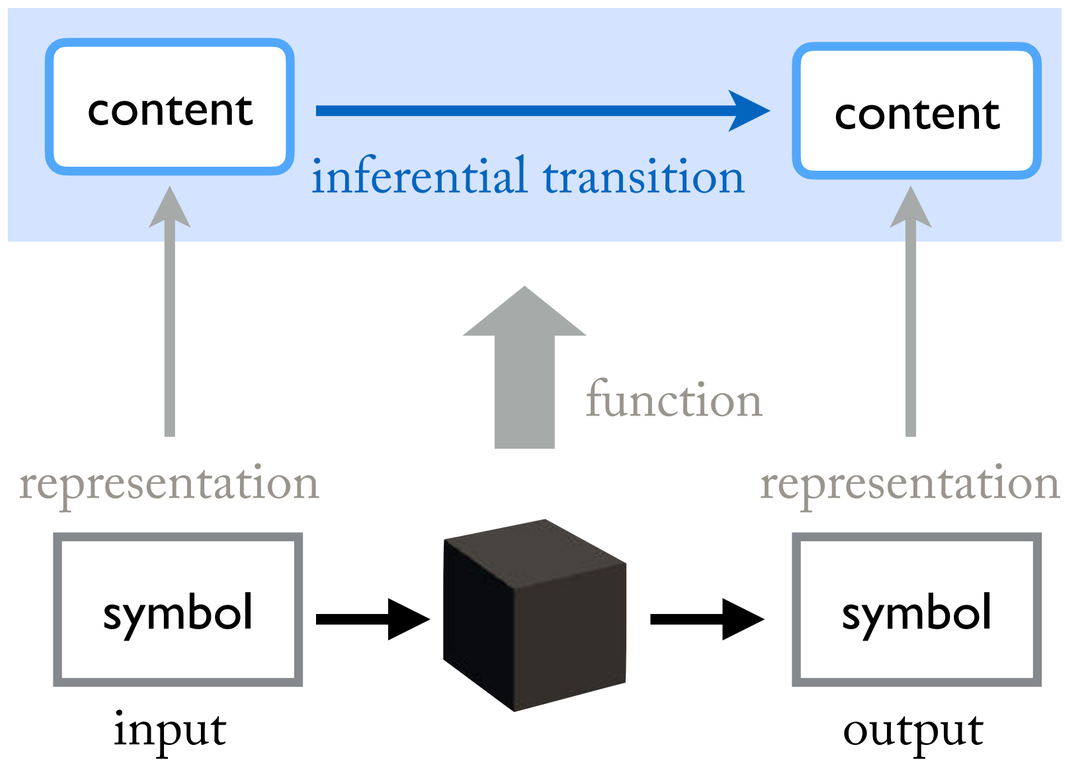
What a representation is about or what it means is its **content**. Symbols **represent** their contents.

Both **external** symbols, like words or pictures, and **internal** symbols, like computer code, can express content

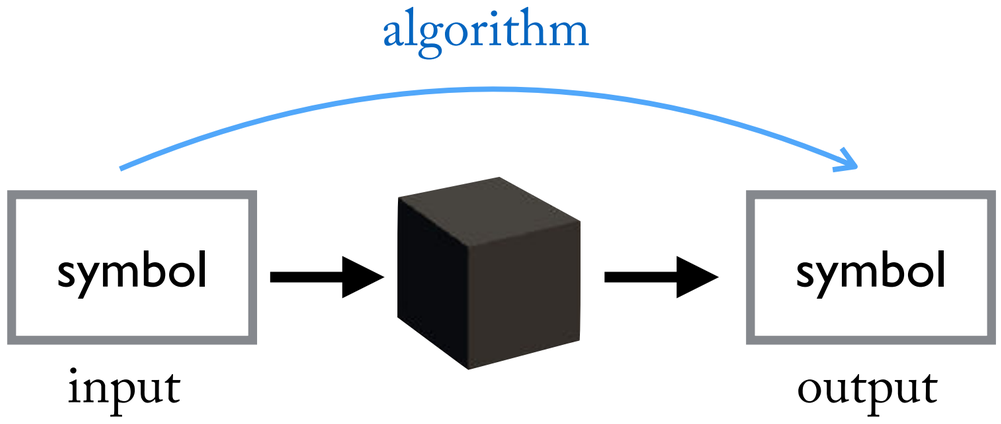


For a computer to perform a inferential transition, its inputs and outputs must represent the initial and final contents of the inferential transition.

The representations processed by the computer must **mirror** the inferential transition.



In order to produce representations which mirror inferential transitions, computers follow **algorithms:** mechanical procedures for manipulating symbols.

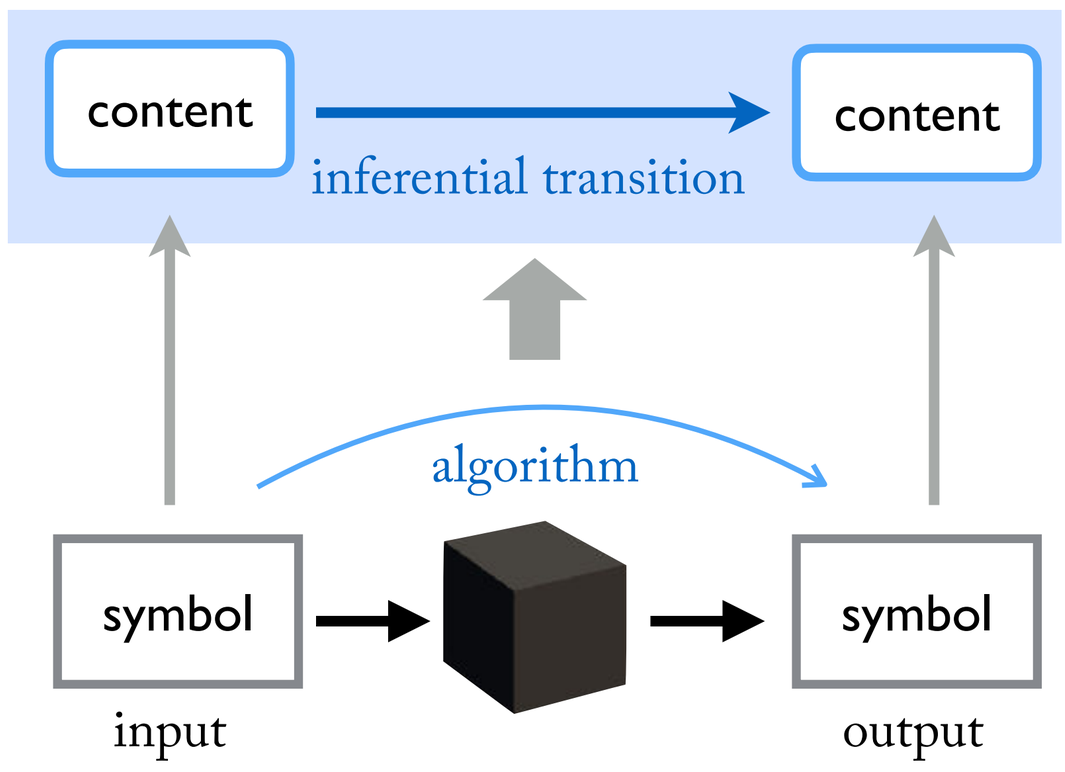


Computation

A computer processes information *by* processing symbols.

A computer functions to perform inferential operations by manipulating representational symbols according to an algorithm.

Computation as embodied logic.



Computational Theory of Mind (CTM)

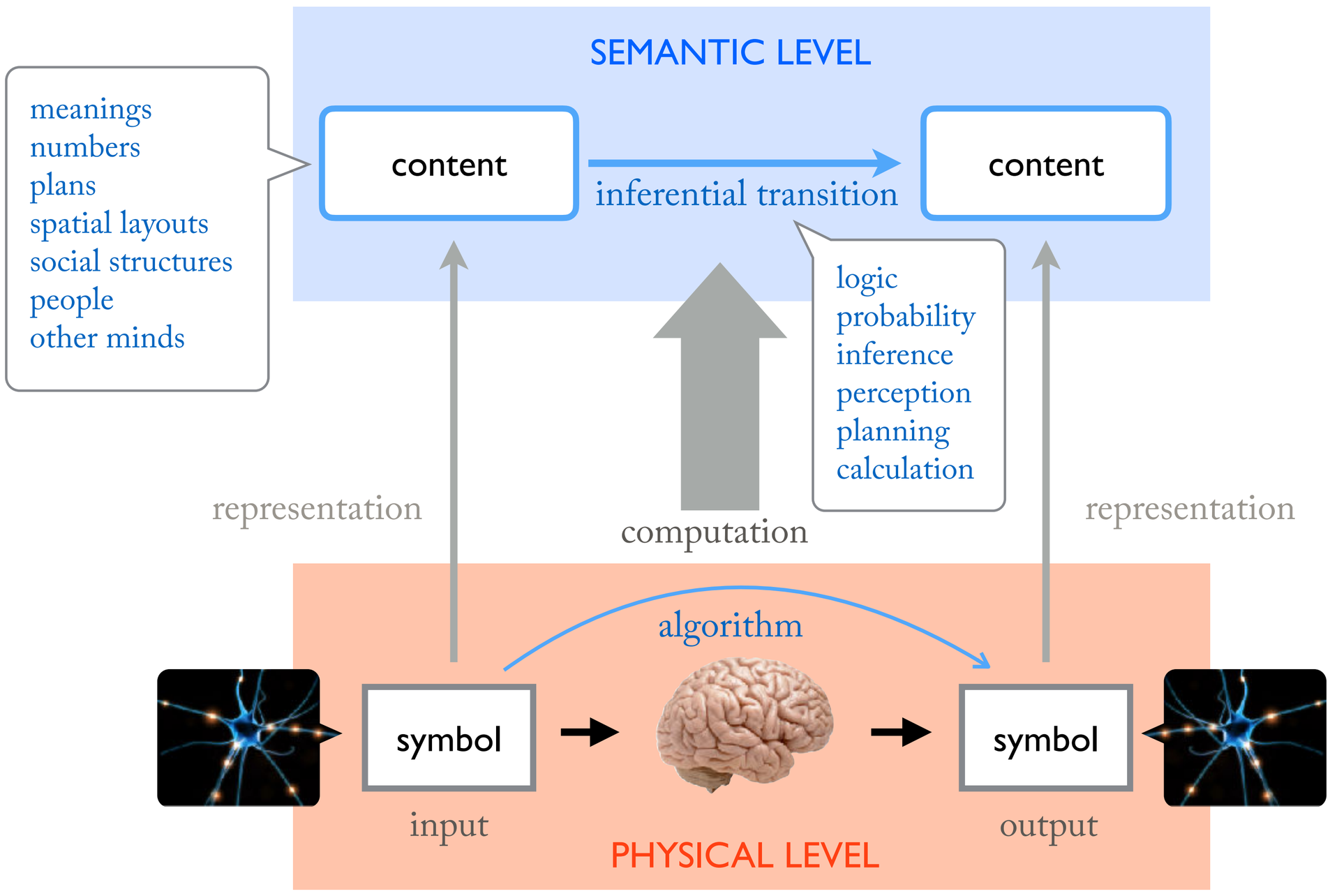
Computational Theory of Mind (v. 1)

1. The brain is a naturally occurring computer or information processor.
2. Cognition arises from the computational activity of the brain.

According to CTM, the brain is a type of computer.

Information processing
The brain has the function of processing information, or preforming inferential operations. This is a biological function, not a design function.

Symbol processing
The brain takes symbols as inputs, it processes symbols, and produces symbols as outputs. Neural symbols in the brain represent mental contents.



Computational Theory of Mind: the brain computes cognitive functions by manipulating symbolic representations in the brain according to biologically grounded algorithms.

Further Notes

- How CTM answers Leibniz Challenge
  Analogy with computers ≈ the brain is a computer, the mind is an algorithm.
- Computers are purely physical.
- They have thought-like states, and intelligence-like processes.
- They can cause and be caused by physical events.
- The physical domain and the semantic domain
- Natural and artificial computers.

- **CTM and Cognitive Science**
  - Cognitive science isn't philosophically committed.
  - But it relies on CTM as a working hypothesis.
  - CTM is a conjecture, not a conclusion.

- **Unit 1 outline:**
  - Logic circuits
  - Functions
  - Representation
  - Computation
  - CTM

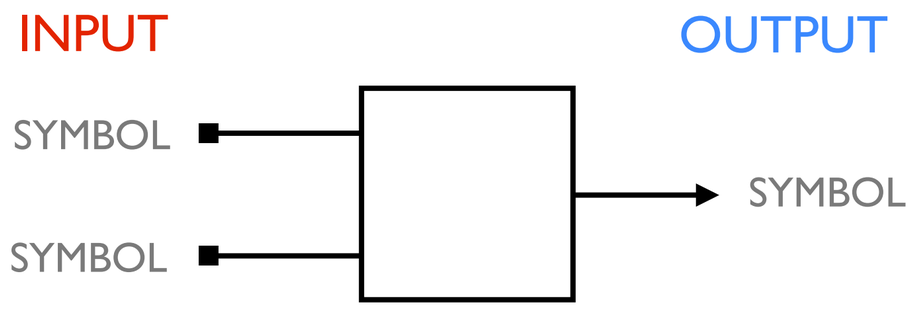
3. Logic Gates & Combinatorial Circuits

Logic Gates

Logic gates are the basic building blocks of most human-made computers. By combining logic gates into complex networks, we can ultimately achieve complete computational power.

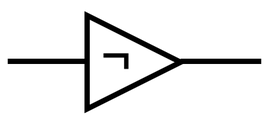
A single logic gate is actually a very, very simple computer. It takes one or two symbols as input (depending on the gate), and produces a single symbol as output.

There are hundreds of millions of logic gates in a normal desktop processor.



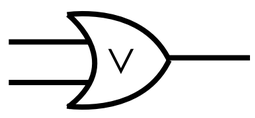
There are three types of logic gates. Each is defined by its input-output profile:

INVERTER



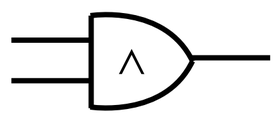
IN	OUT
0	1
1	0

OR-GATE



IN 1	IN 2	OUT
0	0	0
0	1	1
1	0	1
1	1	1

AND-GATE



IN 1	IN 2	OUT
0	0	0
0	1	0
1	0	0
1	1	1

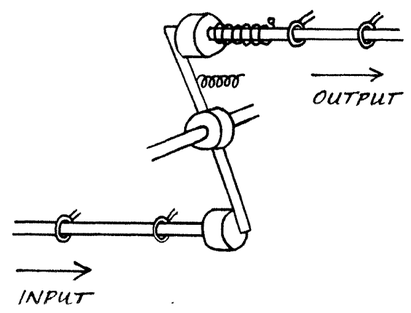
To understand logic gates, don't think of the input and output symbols as actual "1"s and "0"s. They are abstract values that are defined simply as opposites.

The logic gate itself is a kind of abstract machine which matches inputs with outputs.

IN	OUT
0	1
1	0

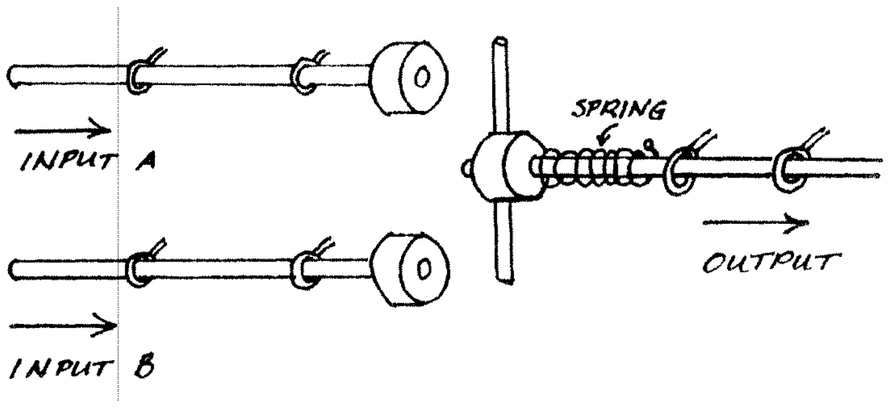
Thought of as abstract machines, logic gates work "instantaneously" and never make mistakes or wear out.

Physical implementations of logic gates take time to process, make mistakes, and are subject to entropy.

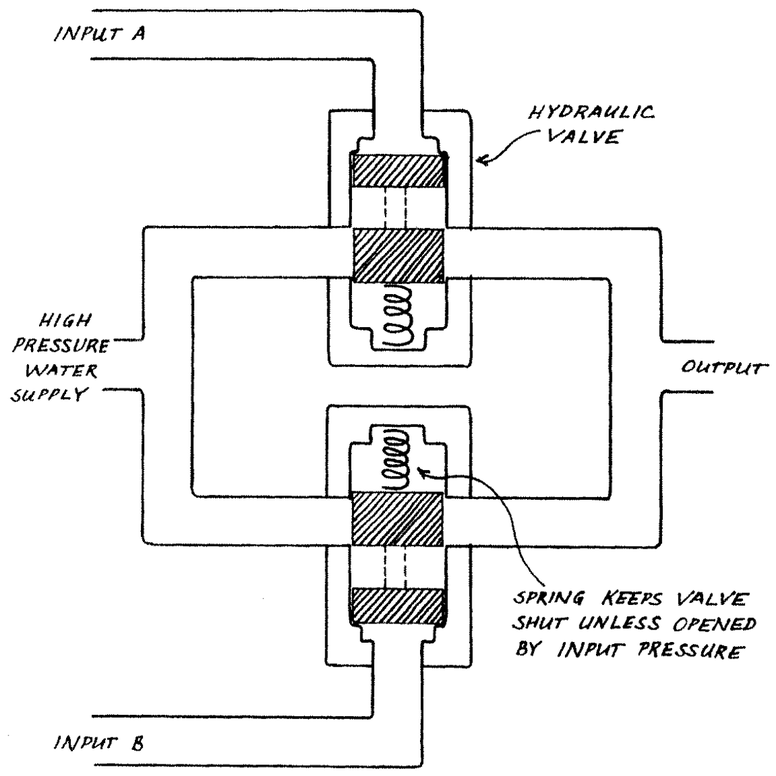


From Physics to Logic

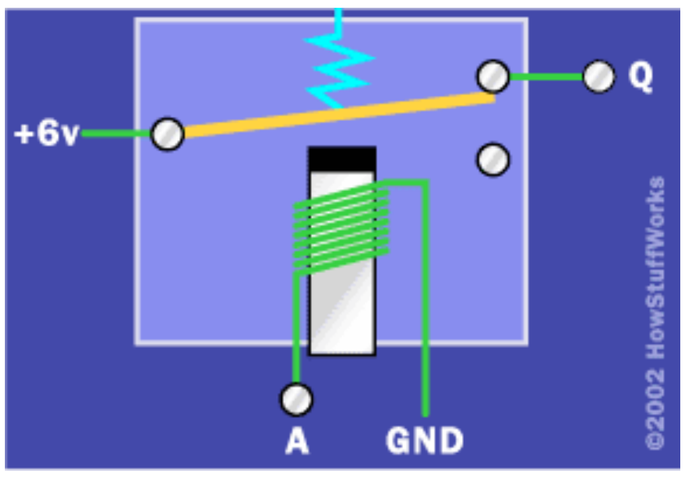
Logic gates can be implemented using any number of different physical designs and physical materials.



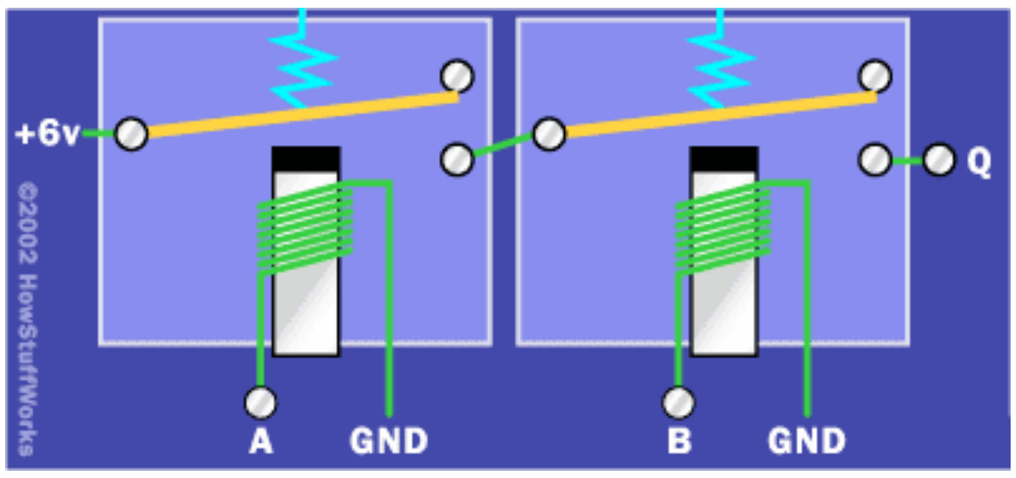
Physical implementation of an OR-gate with a stick and spring mechanism.



Physical implementation of an OR-gate with hydraulic valves.



Physical implementation of a NOT-gate with electromagnets.



Physical implementation of an AND-gate with electromagnets.

Logic gates themselves can be thought of as abstractions from concrete machines.

They abstract away from the internal description of a machine by specifying only inputs and outputs.

And they abstract away from the external description by specifying inputs and outputs only as "0" and "1".

IN	OUT
0	1
1	1

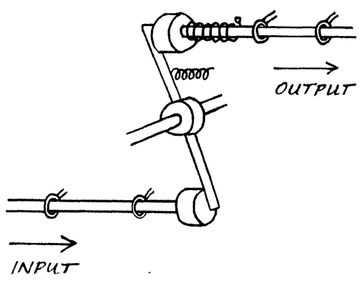
A NOT-gate.

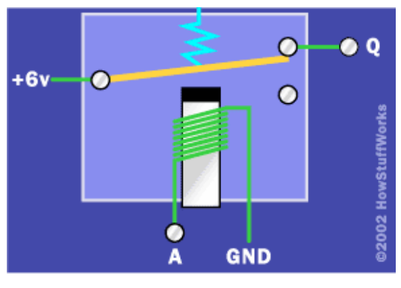
Abstraction over external mechanisms.

IN	OUT
→	←
←	→

IN	OUT
	↯
↯	

Abstraction over internal mechanisms.

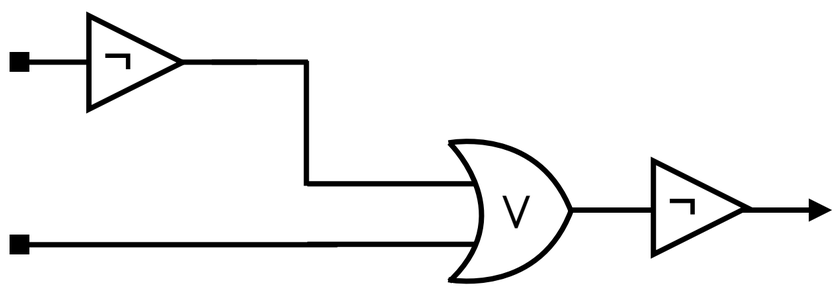




Physical implementations of a NOT-gate.

Combinatorial Circuits: concept

Logic gates can be combined together into a circuit, called a **combinatorial circuit** or **CC.** Here's an example:



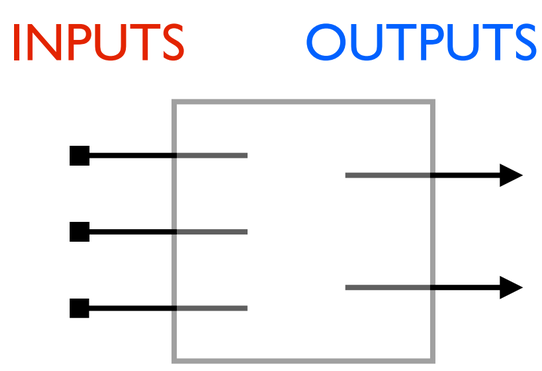
Combinatorial circuits are made up of **logic gates** connected by **links.** The links represent pathways that symbols may flow through.

In a combinatorial circuit, we abstract from time. Symbols pass "instantaneously" from inputs, through the network of links and logic gates, to the final output.

Input/output control. In a factory-made computer, the inputs are determined by the user, while the outputs are determined by the machine (given the input). In a "natural" computer, like the brain of an animal, the inputs are determined by the natural environment, but the outputs are still determined by the machine.

Combinatorial Circuits: definition

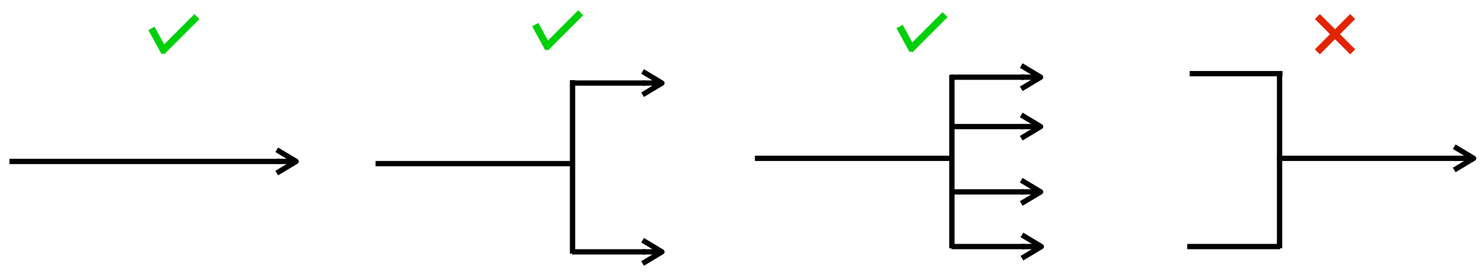
Input and output. A combinatorial circuit can have any number of inputs and any number of outputs.



For clarity, mark links which are connected to the input with small squares. Mark links connected to the output with small arrows.

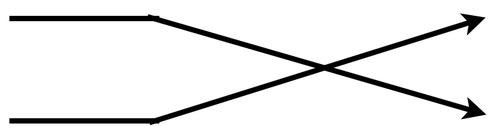
Each input leads either into a logic gate or directly to an output.

Links. Logic gates are connected to one another by **links.** Links may split any number of times. When a links splits, it carries the same symbol to both of its branches. Links may split, but they can't merge, because how would you combine a 0 and a 1?

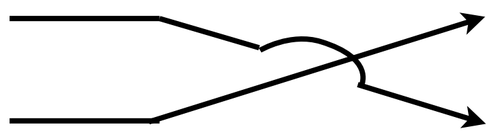


When drawing your circuit diagrams, the link-*lines* may sometimes cross. This is OK, it doesn't say anything about the circuit itself, which is an abstract network of logic gates. Line crossing is just a side effect of drawing in two dimensions.

So the following diagram is fine:



But for visual clarity, introduce a "hop":

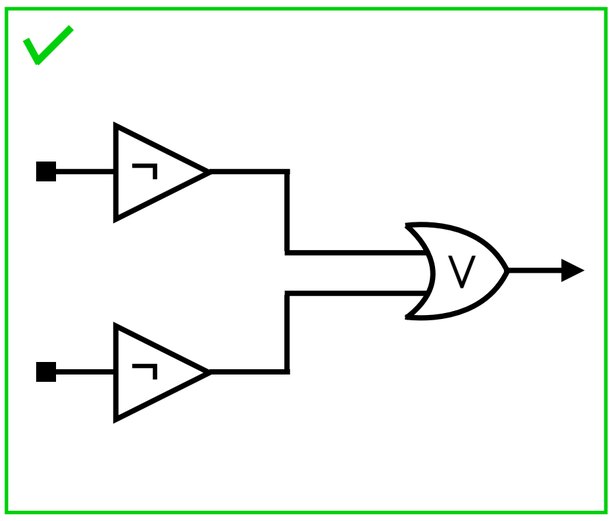


CC Definition. A combinatorial circuit can now be officially defined as follows:

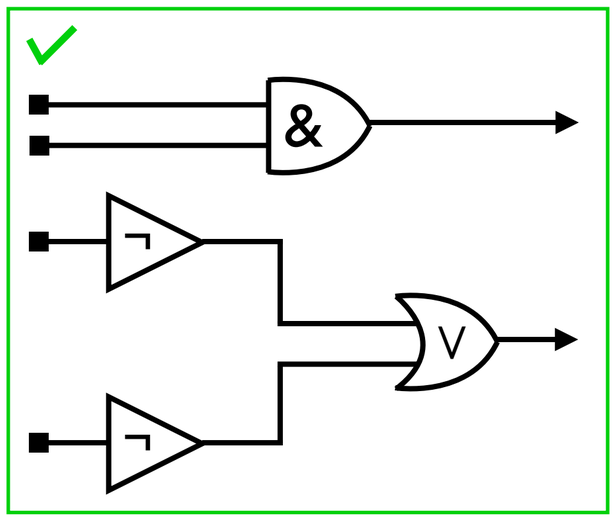
A **combinatorial circuit** is any network of logic gates and links where:

- there are no loops;
- there are no merged links;
- every free end is either an input or output.

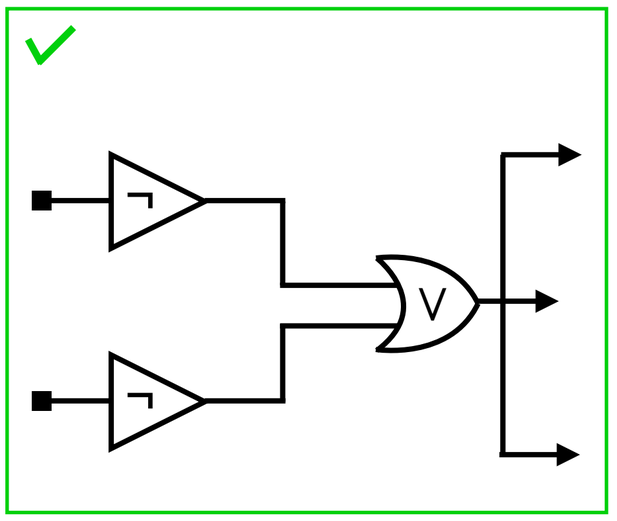
A standard case.



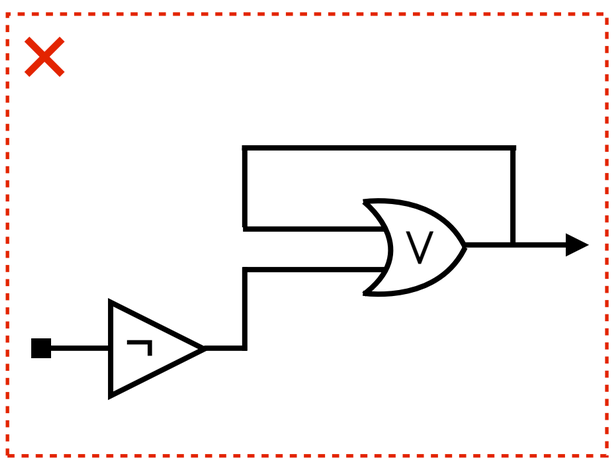
There may be disconnected parts.



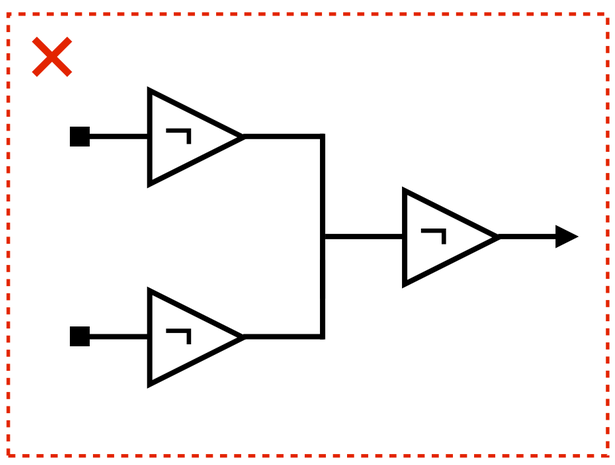
There may be more outputs than inputs.



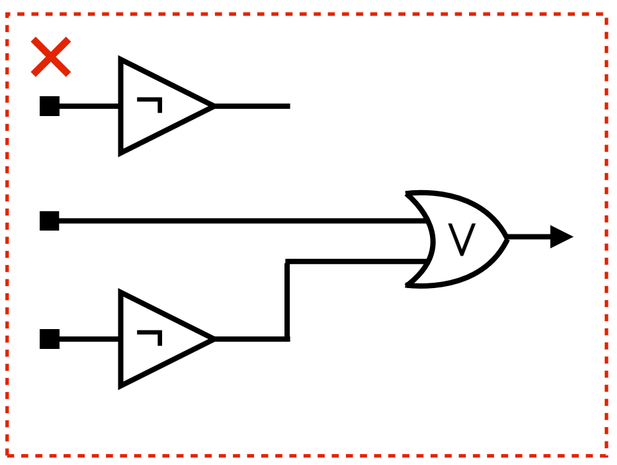
No loops allowed.



No merged links allowed.

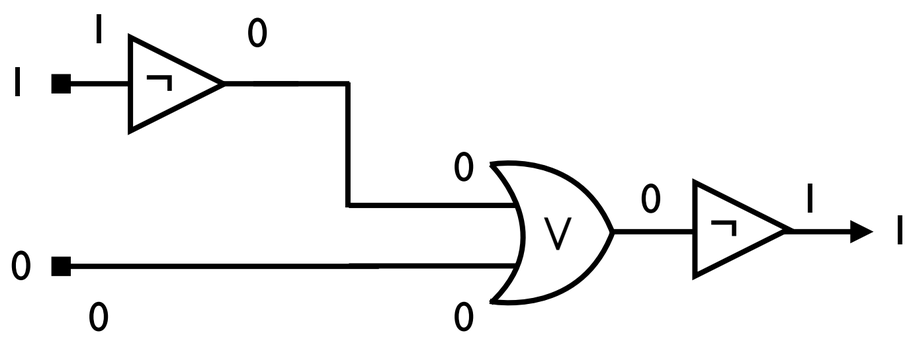


No free ends (except in/out).

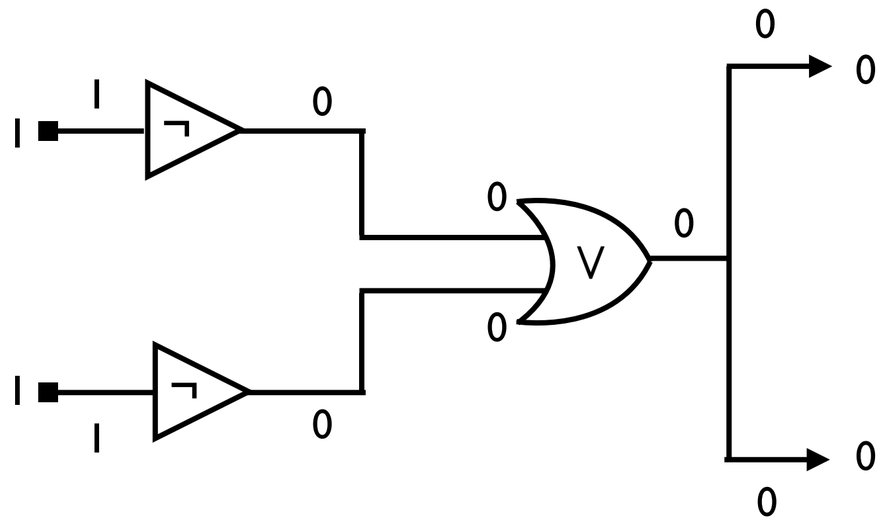


Combinatorial Circuits: Flow of information

Here are two sample circuits and their input-output tables. Each circuit is labeled with one possible set of inputs, as well as their values as they propagate through the circuit.



IN 1	IN 2	OUT
I	0	I
I	I	0



IN 1	IN 2	OUT 1	OUT 2
I	0	I	I
I	I	0	0

In-class Problems

1. Not

IN 1	OUT
0	1
1	0

2. Identity

IN 1	OUT
0	0
1	1

3. And

IN 1	IN 2	OUT
0	0	0
0	1	0
1	0	0
1	1	1

4. Not-And

IN 1	IN 2	OUT
0	0	1
0	1	1
1	0	1
1	1	0

5. Or

IN 1	IN 2	OUT
0	0	0
0	1	1
1	0	1
1	1	1

6. Nor

IN 1	IN 2	OUT
0	0	1
0	1	0
1	0	0
1	1	0

7. All-1

IN 1	OUT
0	1
1	1

8. All-0

IN 1	OUT
0	0
1	0

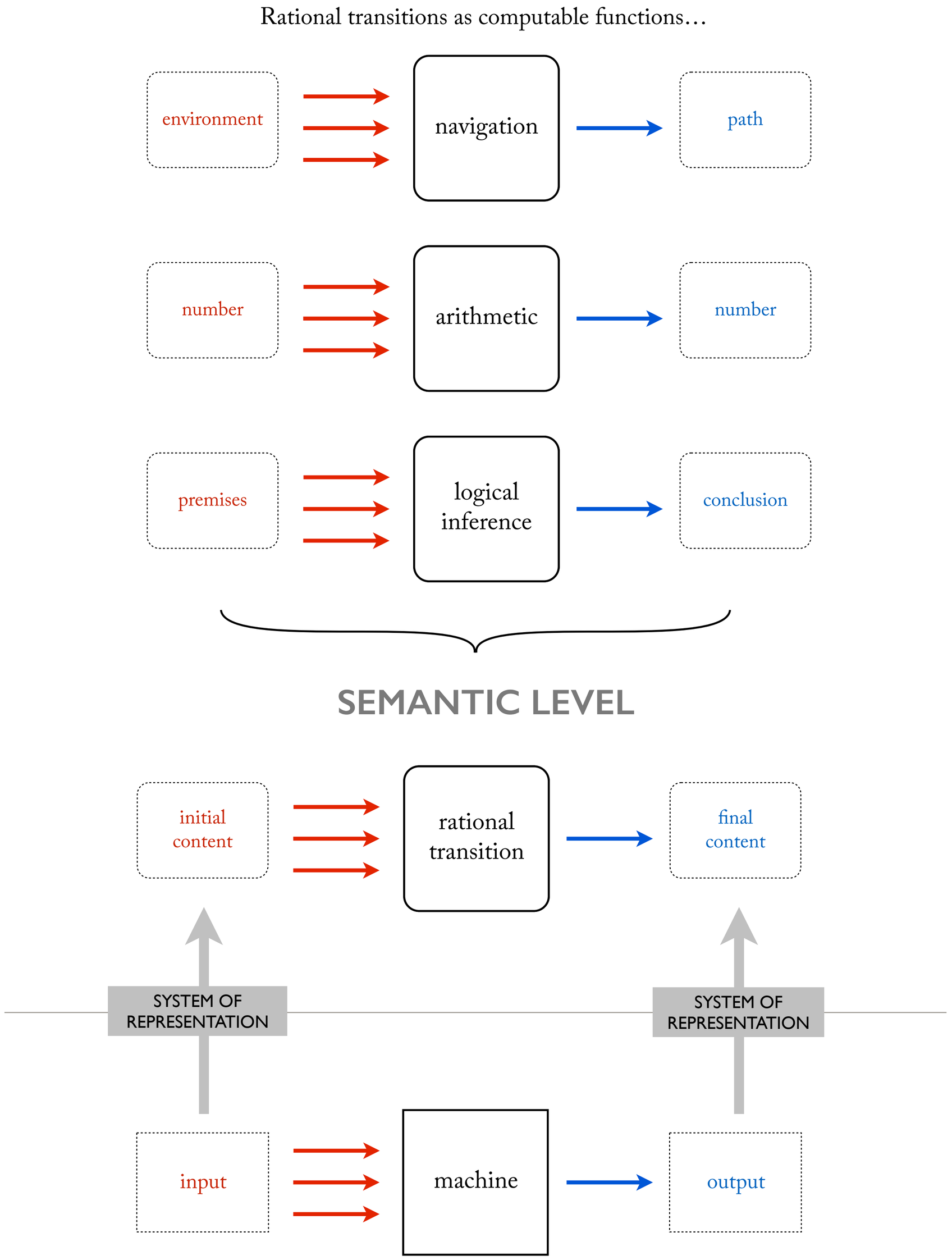
9. Split

IN 1	IN 2	OUT1	OUT2
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

4. Functions

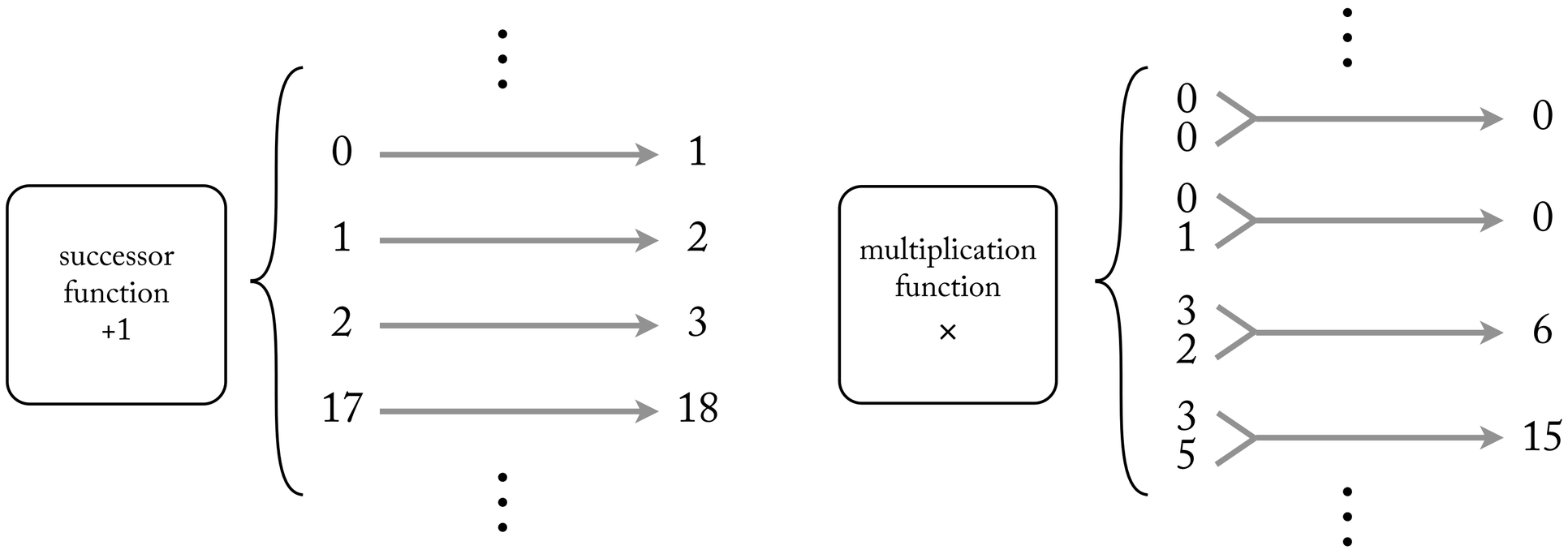
The Structure of Computation

So far we've described computational information processing in terms of rational transitions between informational content. Now we'll use the mathematical concept of a **function** to formalize this idea. Officially, we'll say that a computer **computes a function** when it performs a given rational transition.



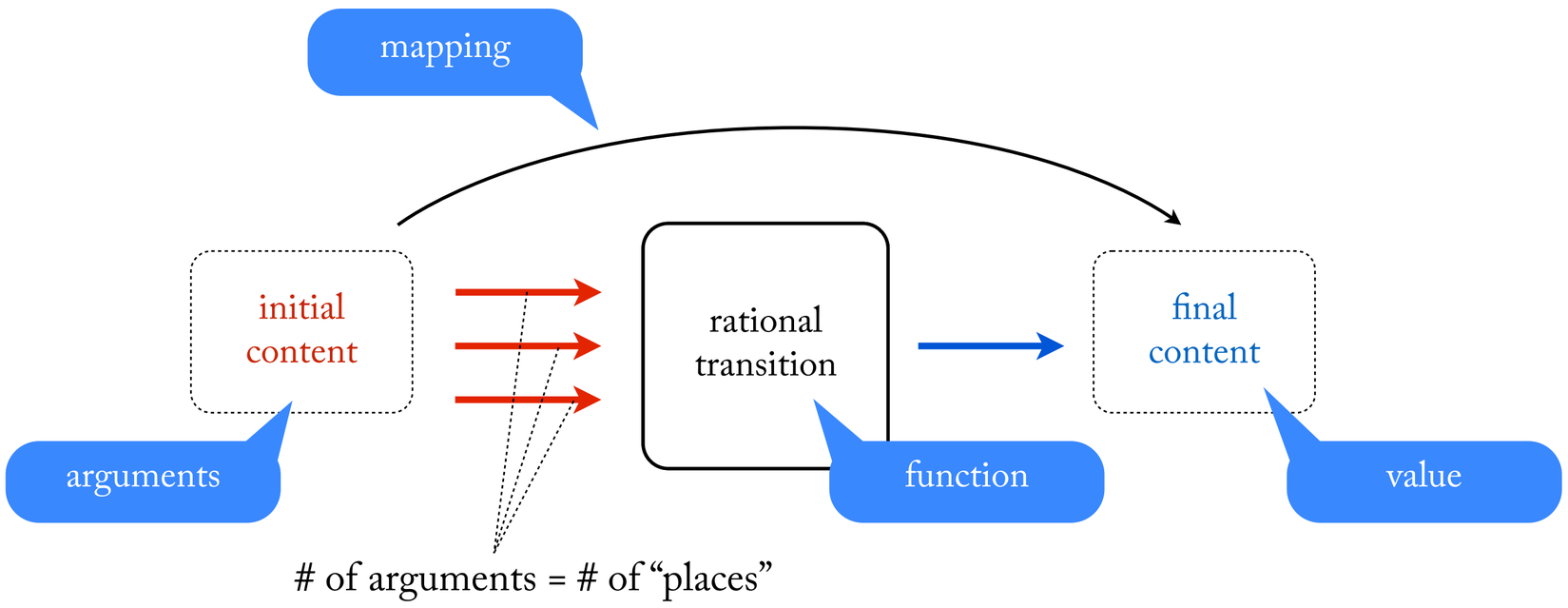
Numerical Functions

Familiar functions correspond to relations between numbers. Examples include *successor*, *addition*, *multiplication*, and so on.



If you think of numbers as abstract objects in Plato's heaven, then functions are abstract and infinite relations *between* those numbers.

The Structure of Functions



Functions are **deterministic**. Given the same arguments, they always return the same value.

Functions can have **many** arguments, but only **one** value.

Multiple arguments go in different **argument places**. What is a function for which argument place matters?

the addition function:

3 + 4 = 7

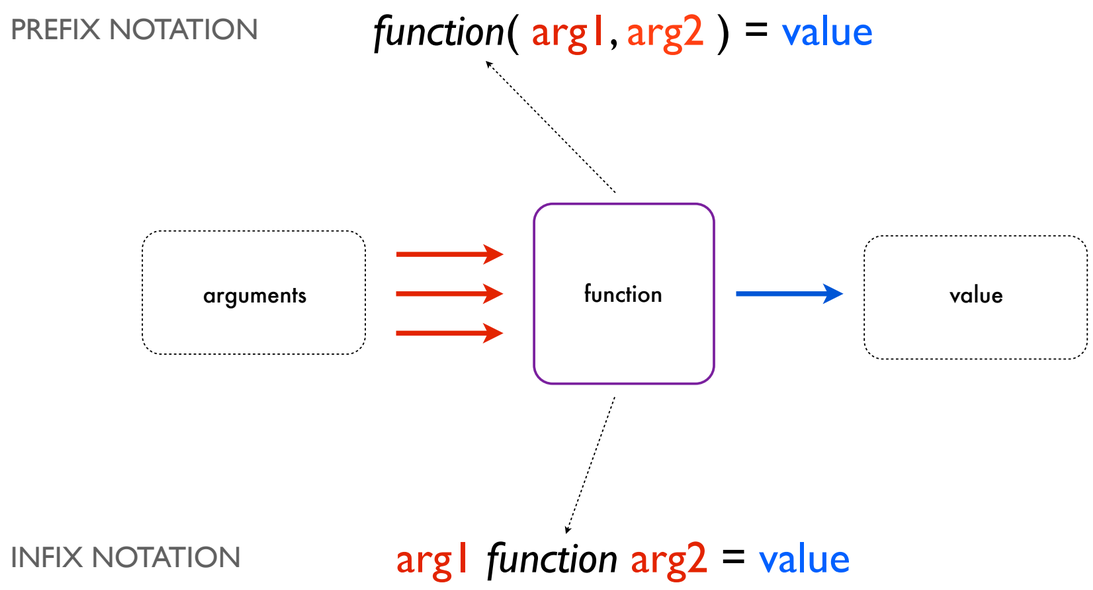
argument 1 argument 2 value

Functions: Terminology

How to **talk** about functions…	The function **takes X** (as an argument) and **returns Y** (as value).
	The function **applied to X, returns Y**.
	The function **maps X to Y.**

Functions: notation

How to **write** about functions...



Prefix notation puts the function symbol on the left side.

" add( ) "	means	*the addition function*
" add(*x*, *y*) "	means	*the addition function applied to arguments x and y*

For example:

" add(3,4) "	means	*the addition function applied to the numbers three and four*
	which is...	*the number seven*

Which is why it is true to say: add(3,4) = 7

Infix Notation (Grade School Notation) puts the function symbol in the middle.

" + "	means	*the addition function*
" x+y "	means	*the addition function applied to arguments x and y*

For example:

" 3+4 "	means	*the addition function applied to the numbers three and four*
	which is...	*the number seven*

Which is why it is true to say: 3+4 = 7

Functions: Composition

Functions can be combined together. This is called the **composition** of functions. Feel free to mix the two styles of notation together as you find convenient.

(3 + 4) × 2 = 14

multiply(add(3,4), 2) = 14

add(3,4) × 2 = 14

These statements are all equivalent, and appropriate use of notation.

Defining a Function

The value of a function may be **undefined** for some arguments.

*divide(*3,0) = undefined *square.root(*-4) = undefined

Most functions are defined for **infinitely many** arguments.

Some functions are defined only for **finitely many** arguments. Call these **finite functions**.

To define an **infinite function** you typically have to **use an equation.**

Example 1
Let s( ) be the 1-place function which satisfies the following condition:
For any number x: $s(x) = x + 1$

Example 2
Let q( ) be the 2-place function which satisfies the following condition:
For any numbers x, y: $q(x,y) = x + (y \times 2)$

To define an **finite function** you typically have to **use a table.** (See below for an exception.)
By convention, the function is undefined for the unlisted argument pairs.

q( ) =

argument 1	argument 2	value
0	0	15
0	1	3
1	0	0
1	1	1
1	2	2

The Identity of Functions

A function is defined by its arguments and its values *only*.
It doesn't matter *how* these are specified.

Example 1

Let f( ) be the function:
for any x: $f(x) = x^2$

Let g ( ) be the function:
for any x: $g(x) = x \times x$

It follows that f( ) = g ( ).

Example 2

Let h( ) be the function:

argument	value
1	2
2	3
3	4

Let j () be the function:
for any x:
$j(x) = x + 1$ if $1 \leq x \leq 3$:
$j(x)$ = undefined otherwise

It follows that h( ) = j( )

Functions vs Machines

In this class it's critical to keep your concept of **function** and your concept of **machine** clearly separated. (Even if, in some of the readings, authors sometimes run them together!)

function

A function is…

- An abstract entity, which often relates numbers.
- Has no temporal or spatial location.
- Up to infinite in size.
- "Instantaneously" relates one number to another.
- Requires no "processing" to do this.

---

machine

A machine is…

- A concrete device, which processes symbols.
- Has a temporal and spatial location.
- Only finite in size.
- Takes time to take an input and produce an output.
- Requires processing to do this.

The Metaphysics of Functions

Mathematical functions are abstract infinite relations between numbers. There is nothing more to a function than the numbers it relates.

You can think of a mathematical function as an infinite "list" of numbers.

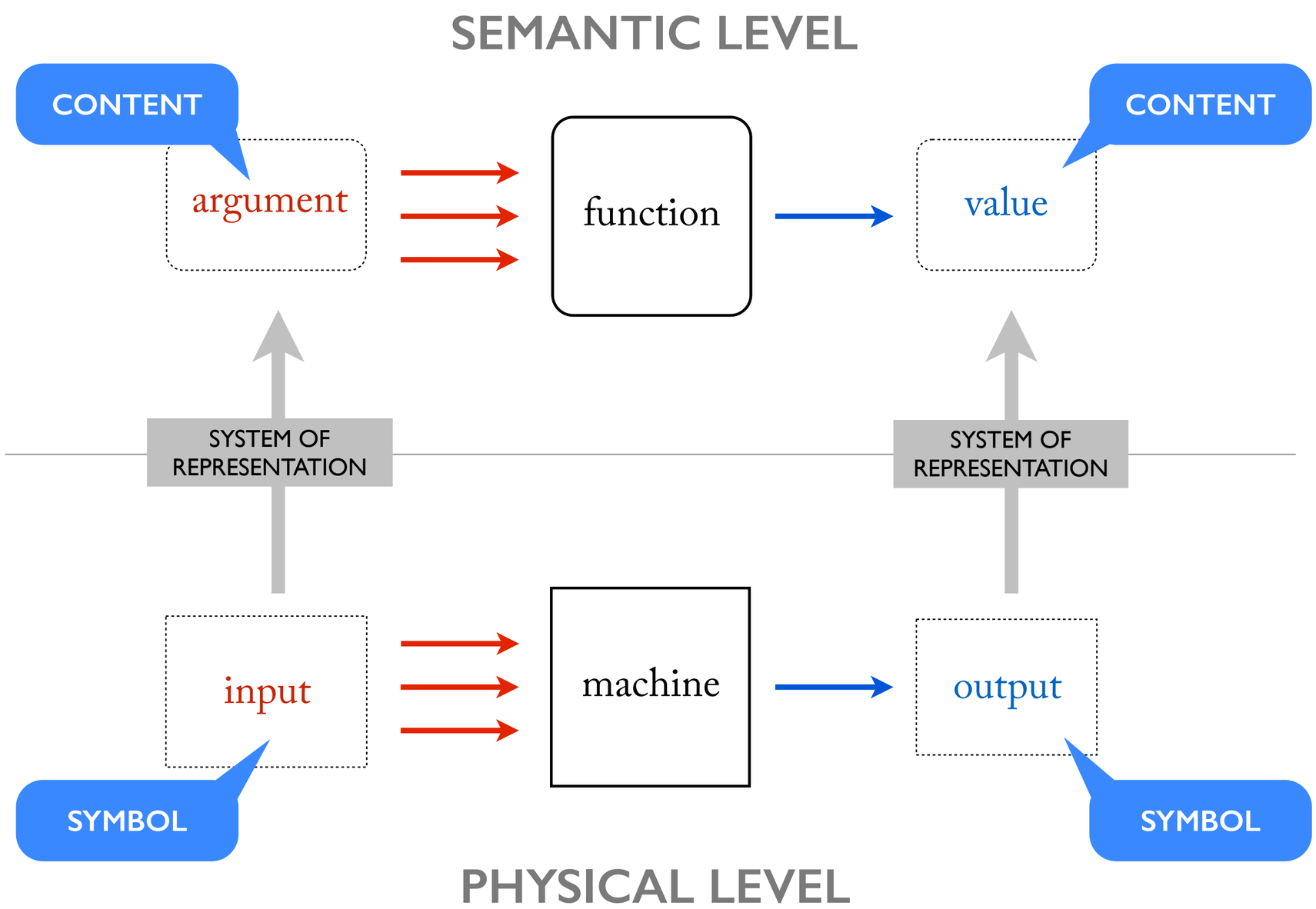
Other kinds of functions relate can relate other kinds of objects (sets, people, places, etc.). But the functions themselves are always abstract relations.

arg 1	arg 2	value
0	0	0
0	1	0
1	0	0
1	1	1
1	2	2
2	0	0
2	1	2
2	2	4
3	0	0

5. Systems of Representation

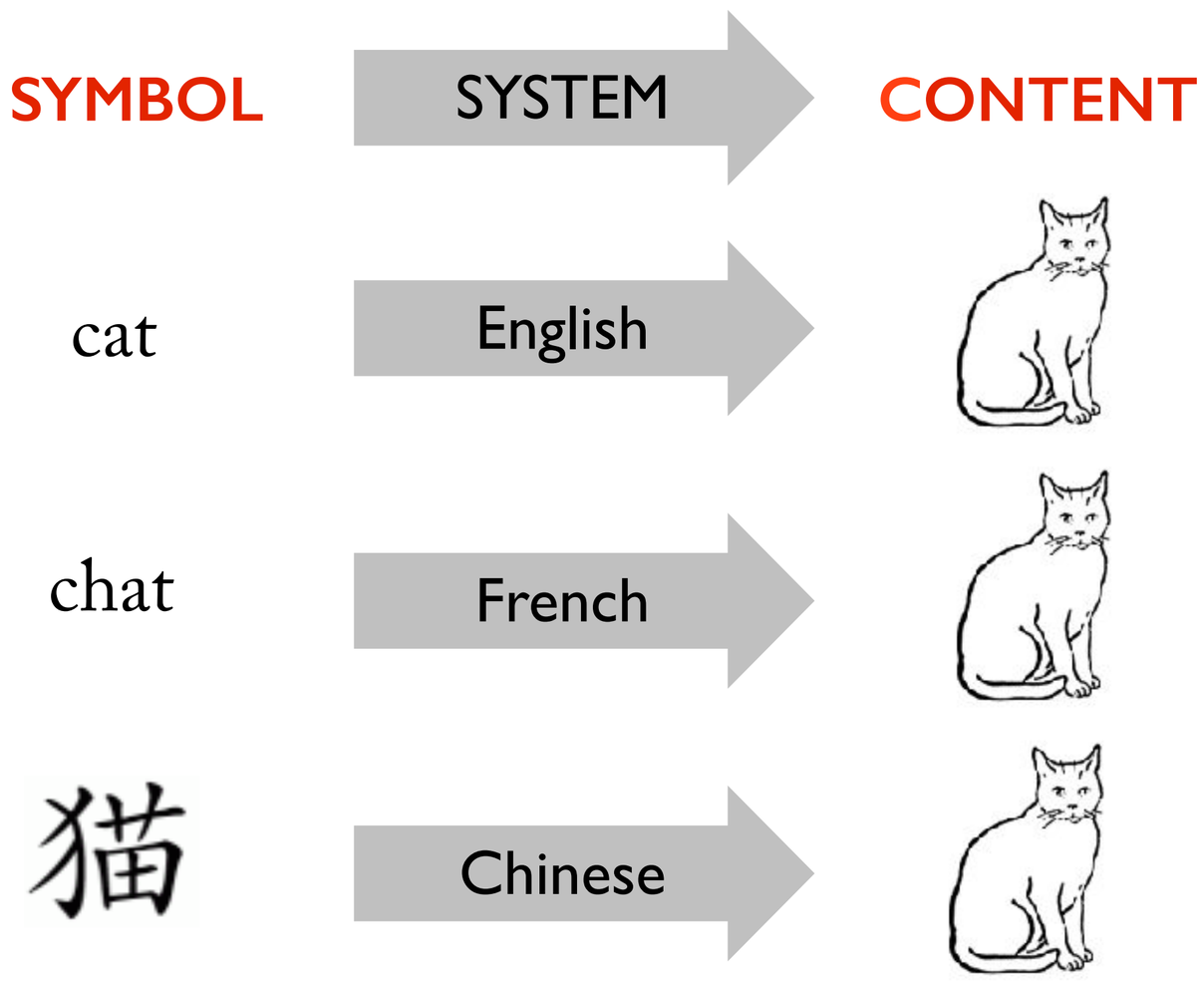
The role of representational systems in computation

Systems of representation associate symbols with contents. In computation, they map inputs to arguments and outputs to values. They are the glue that connects the physical and semantic levels.



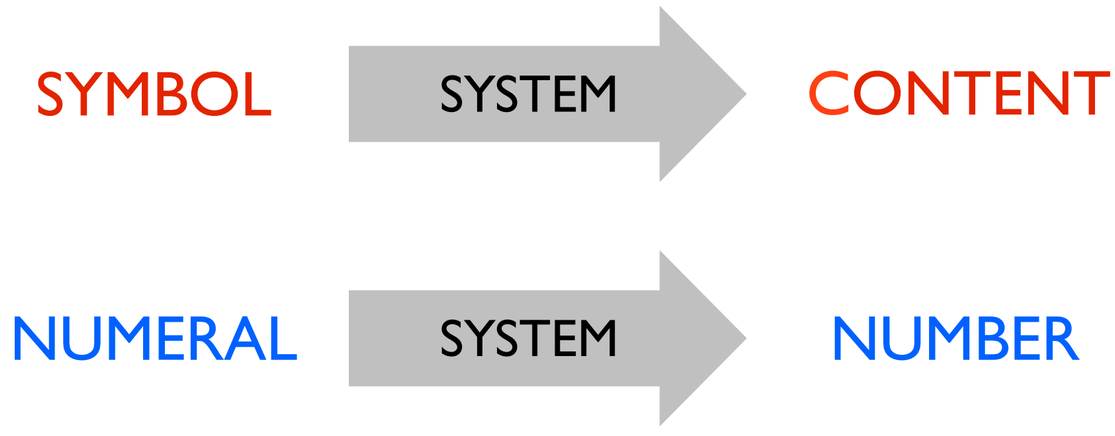
Symbol, content, system

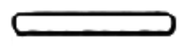
Symbols express contents only relative to a system. The same content can be expressed by different symbols in different systems. Here is an illustration for the case of words in a language.

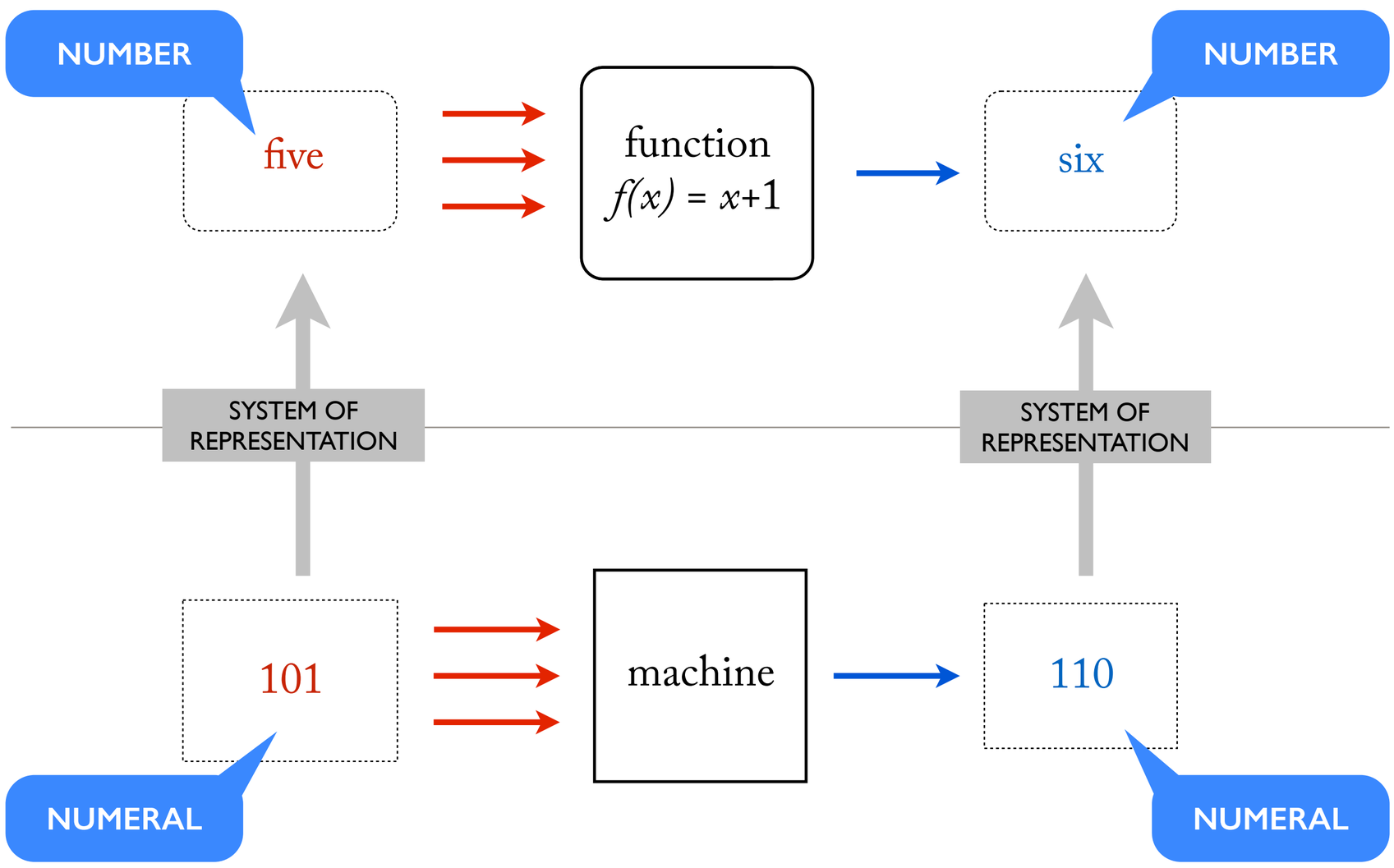


Numeral systems

Numeral systems are systems of representation specifically for numbers. In these systems, the symbols are called **numerals**, and the contents are **numbers**. Numerals are types of physical symbols, while numbers are abstract quantities.



Different numeral systems use….	Roman numerals	Arabic numerals	Mayan numerals	The numbers represented
		0		*zero*
	I	1		*one*
	II	2		*two*
	III	3		*three*
	IV	4		*four*
	V	5		*five*



Anatomy of a Numeral

A **complex** numeral is a made up of a string of **atomic** numerals. Each atomic numeral has a **position** within the complex numeral.

In the Arabic numeral system, the atomic numerals are the numerals 0-9, and all other numerals are constructed from these.

3rd position	2nd position	1st position	
3	8	4	complex symbol

Atomic symbols in the **Indo-Arabic** system

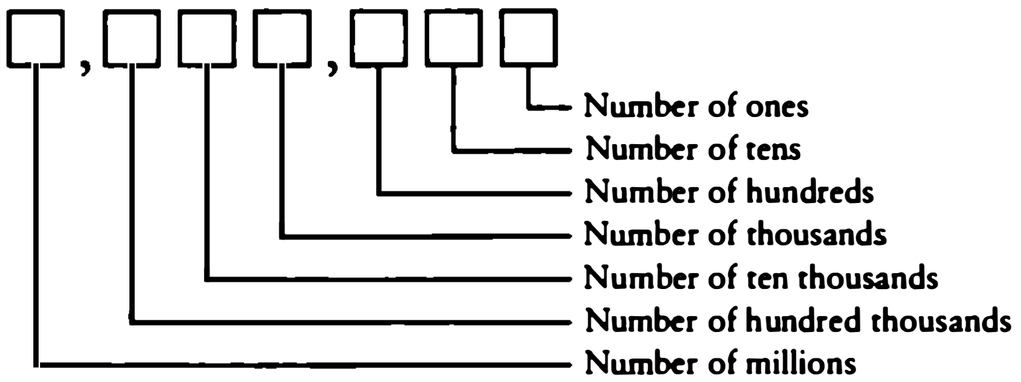
0 1 2 3 4 5 6 7 8 9

Mechanics of a Numeral System

A **positional** numeral system is defined by (i) the values it assigns to each atomic symbol (always starting with zero); (ii) the value of its positions.

The Arabic numeral system is positional, but the Roman system is not.

Positions in the **Indo-Arabic** system

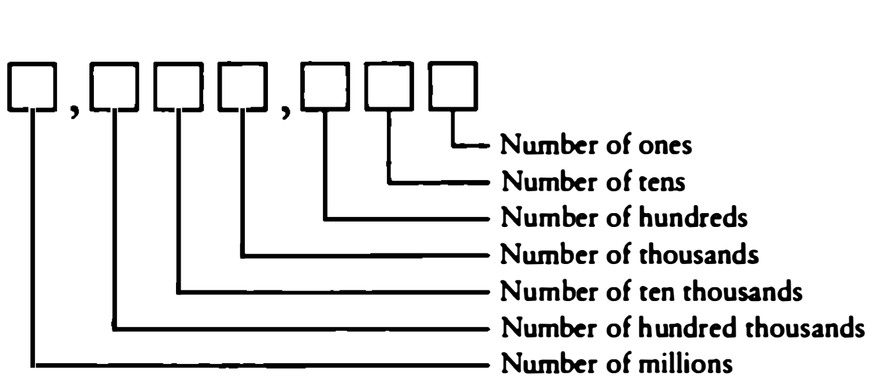


Base

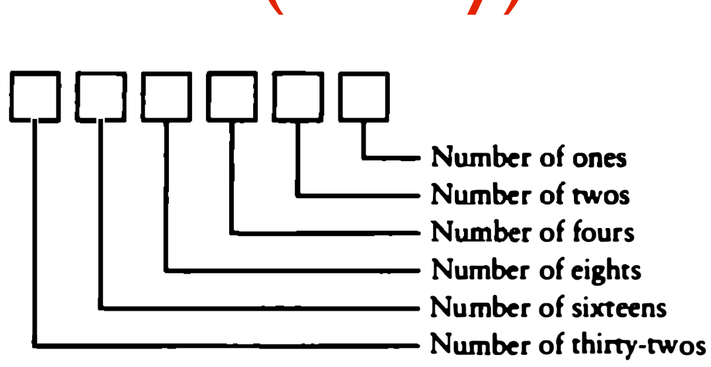
The **base** of a positional system corresponds to the number of atomic symbols. In the Arabic system, there are ten symbols, 0-9, so this system is called **base 10**. In the Binary system, there are only two symbols (0,1), so this system is **base 2**.

The base determines the value of each position.

Base 10



Base 2 (Binary)



Base 10	Base 7	Base 2 (Binary)	Base 1 (Tally)
1	1	1	1
2	2	10	11
3	3	11	111
4	4	100	1111
5	5	101	11111

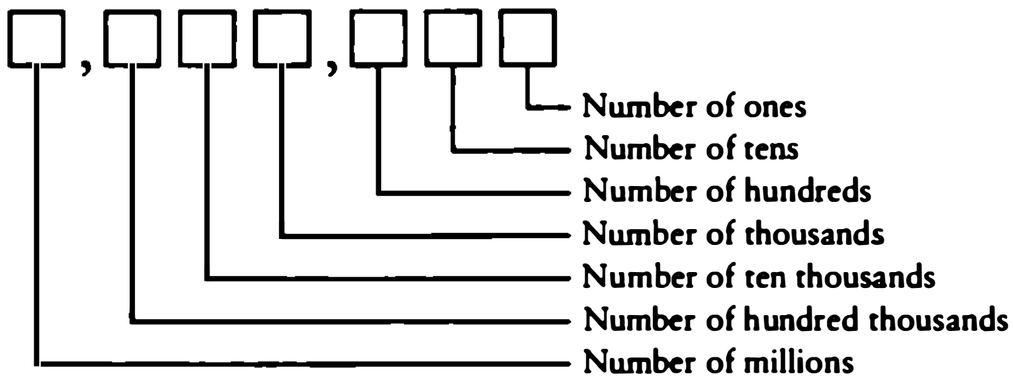
Calculating a Numeral's Value

Decimal System

ten atomic symbols, *base*=10

0 1 2 3 4 5

zero one two three four five …



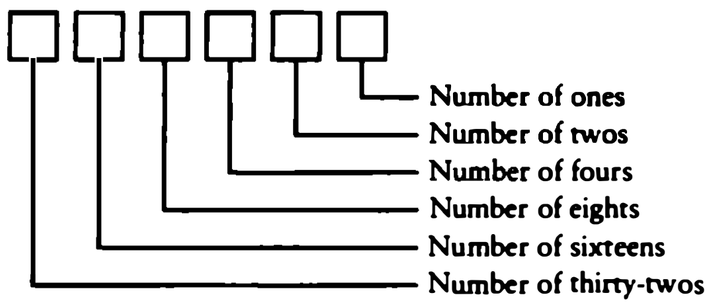
$$111_D = 1_D \times 100 + 1_D \times 10 + 1_D \times 1$$

Binary System

two atomic symbols, *base = 2*

0 1

zero one

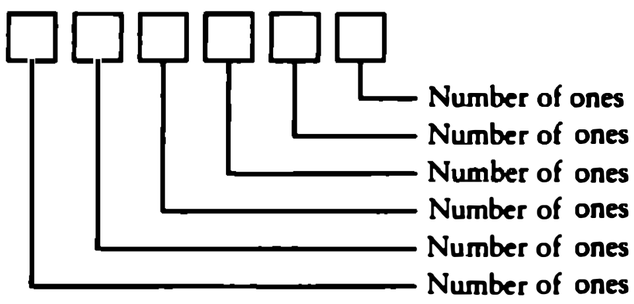


$$111_B = 1_B \times 4 + 1_B \times 2 + 1_B \times 1$$

Tally System

one atomic symbol, *base = 1*

I



$$111_T = 1_T \times 1 + 1_T \times 1 + 1_T \times 1$$

Note: extended tally system with unlimited 0's to the left.

Binary and Tally

In this course, the most important numeral systems will be **tally (T)** and **binary (B).** This is because you can express all numbers in these systems using only two symbols, 0 and 1.

The same set of numerals represent different numbers in the two systems.

11 → Binary → *three*

11 → Tally → *two*

Tally Syntax

Tally systems in common usage take a variety of forms.

(Images from Wikipedia)

For our purposes a **well formed tally numeral** is any string of bits, where the the first n (≤ 0) bits from the right are 1's, and they are followed to the left by any number of 0's.

$$0 \ldots 0\, 1 \ldots 1$$

So the following are well formed:

000
11
0011
1111

But these ones are note.

1000
1101
1110

When designing machine to compute a function in tally, we'll always assume that inputs are well-formed.

Binary Syntax

Tally systems in common usage take a variety of forms.

For our purposes a **well formed tally numeral** is any string of bits, where the the first *n* (≤0) bits from the right are 1's, and they are followed to the left by any number of 0's.

0 . . . 0 1 . . . 1

So the following are well formed:

000
11
0011
1111

000
11
0011
1111

(Images from Wikipedia)

When designing machine to compute a function in tally, we'll assume that all inputs are well-formed. For example, you'l never get the input "1011".

---

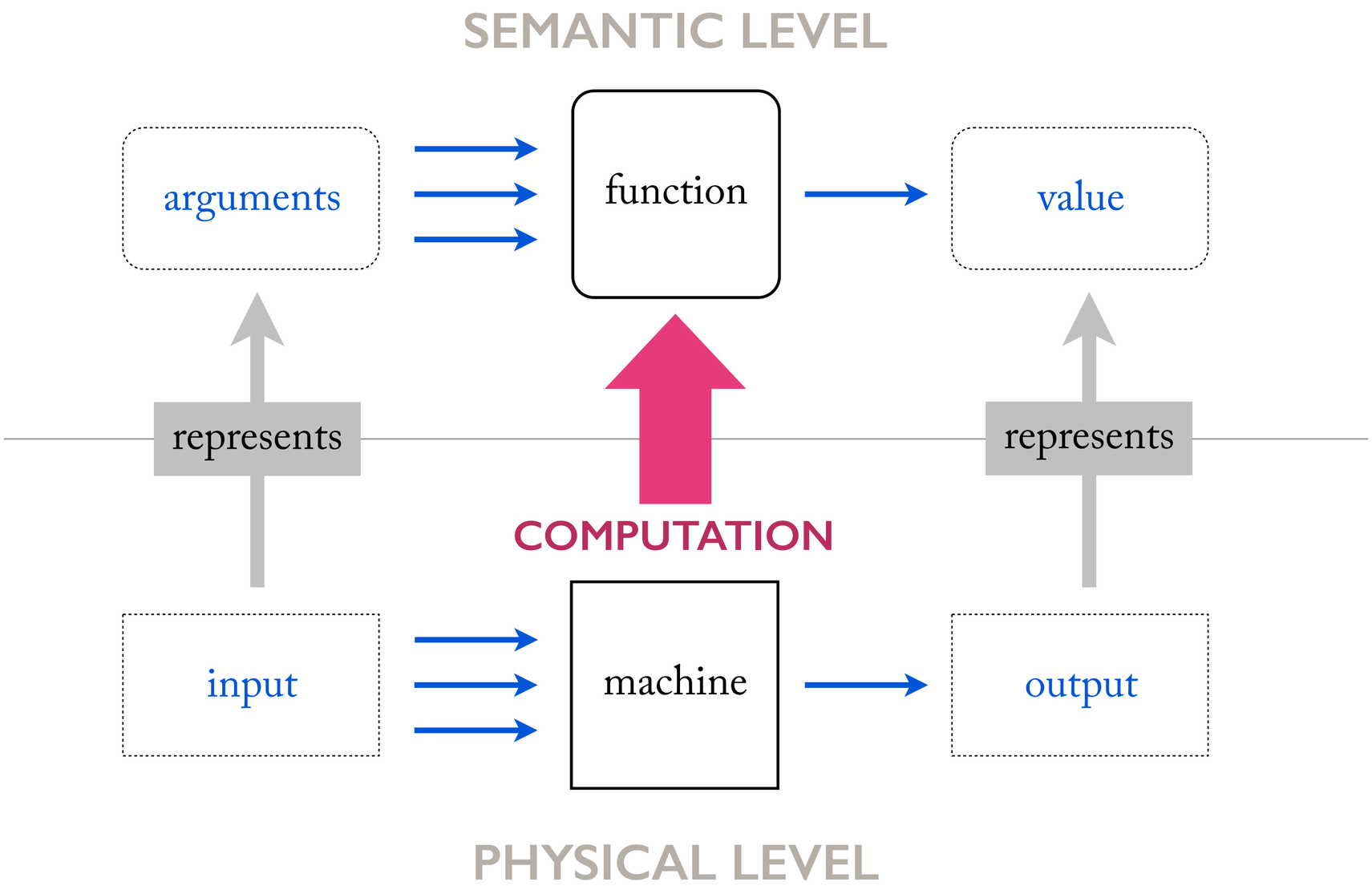
one

6. Defining Computation

Integrate more of Crane reading. Distinguish computing vs. implementing a function.

The structure of computation

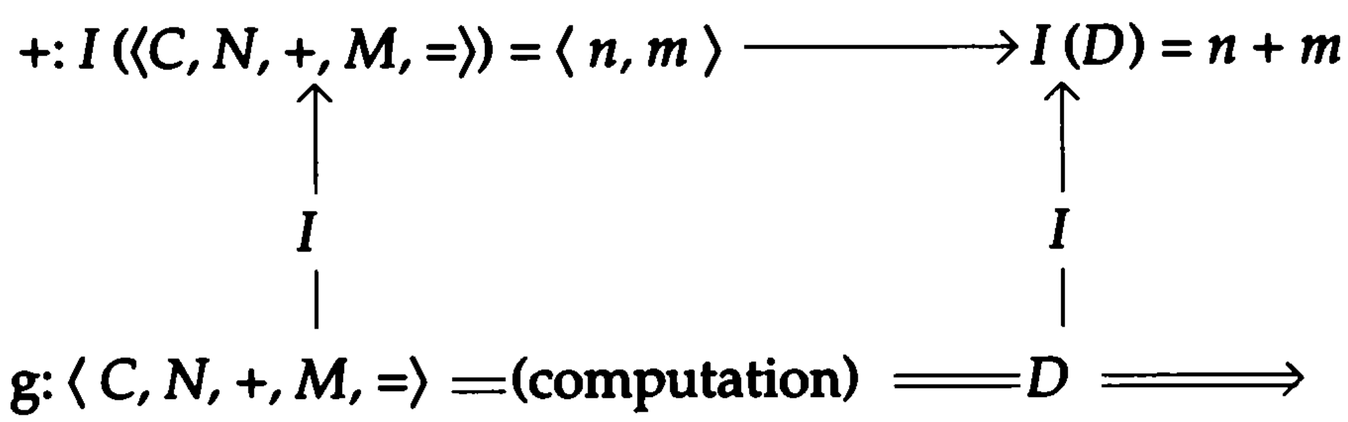
Computation is a relation between a **machine** and a **function** that is brought about when the machine is designed to physically **mirror** or **simulate** the abstract argue-value profile of the function. In this section, we will define this relationship formally.



The "tower bridge" model of computation

This way of thinking about computation goes back to Turing (1936), but Cummins (1991) gives an especially clear statements of the essential idea:

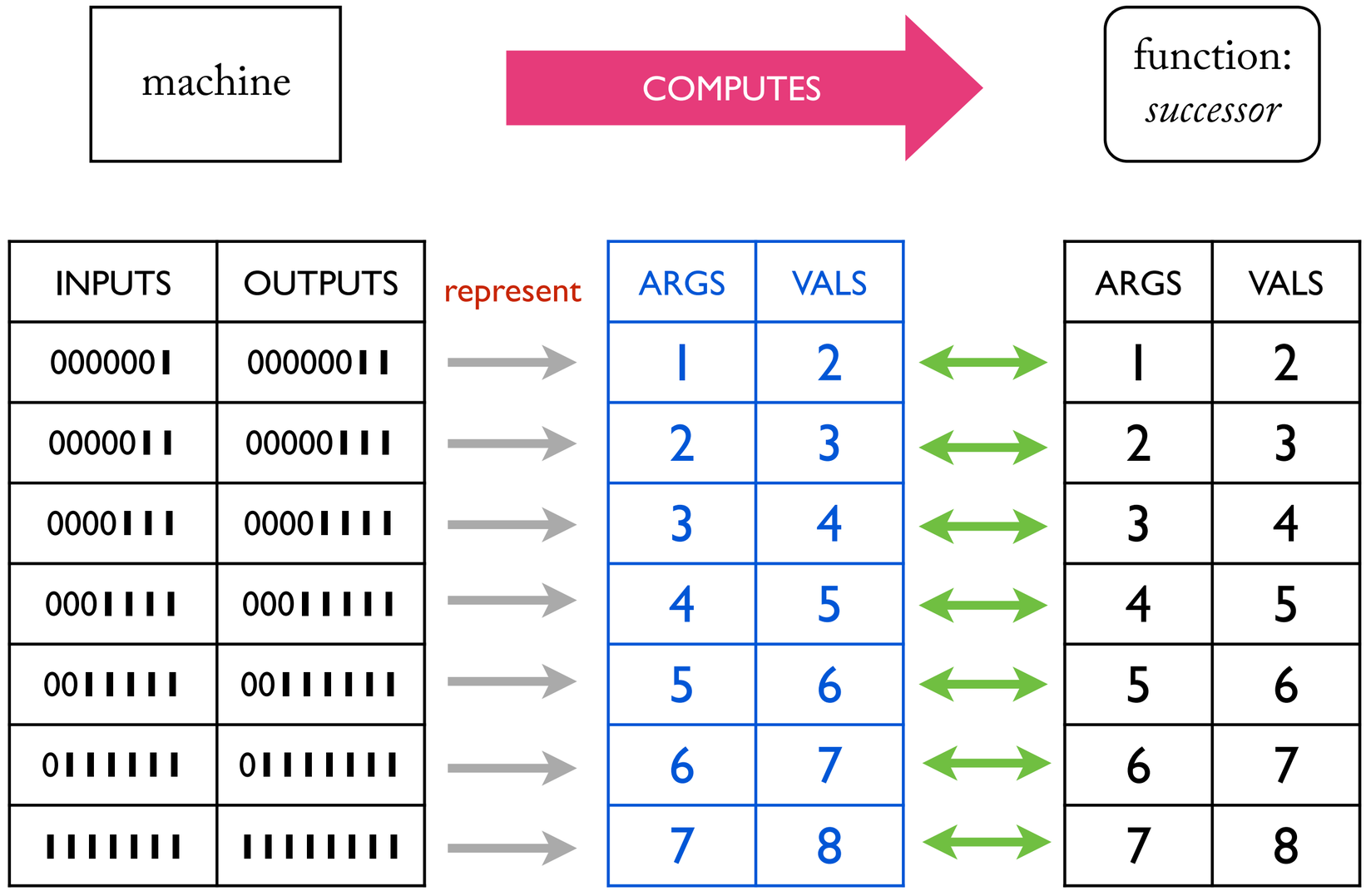
> **The following is a useful picture of this whole conception of adding machines. I call it the Tower Bridge picture because it reminds me of London's Tower Bridge.**



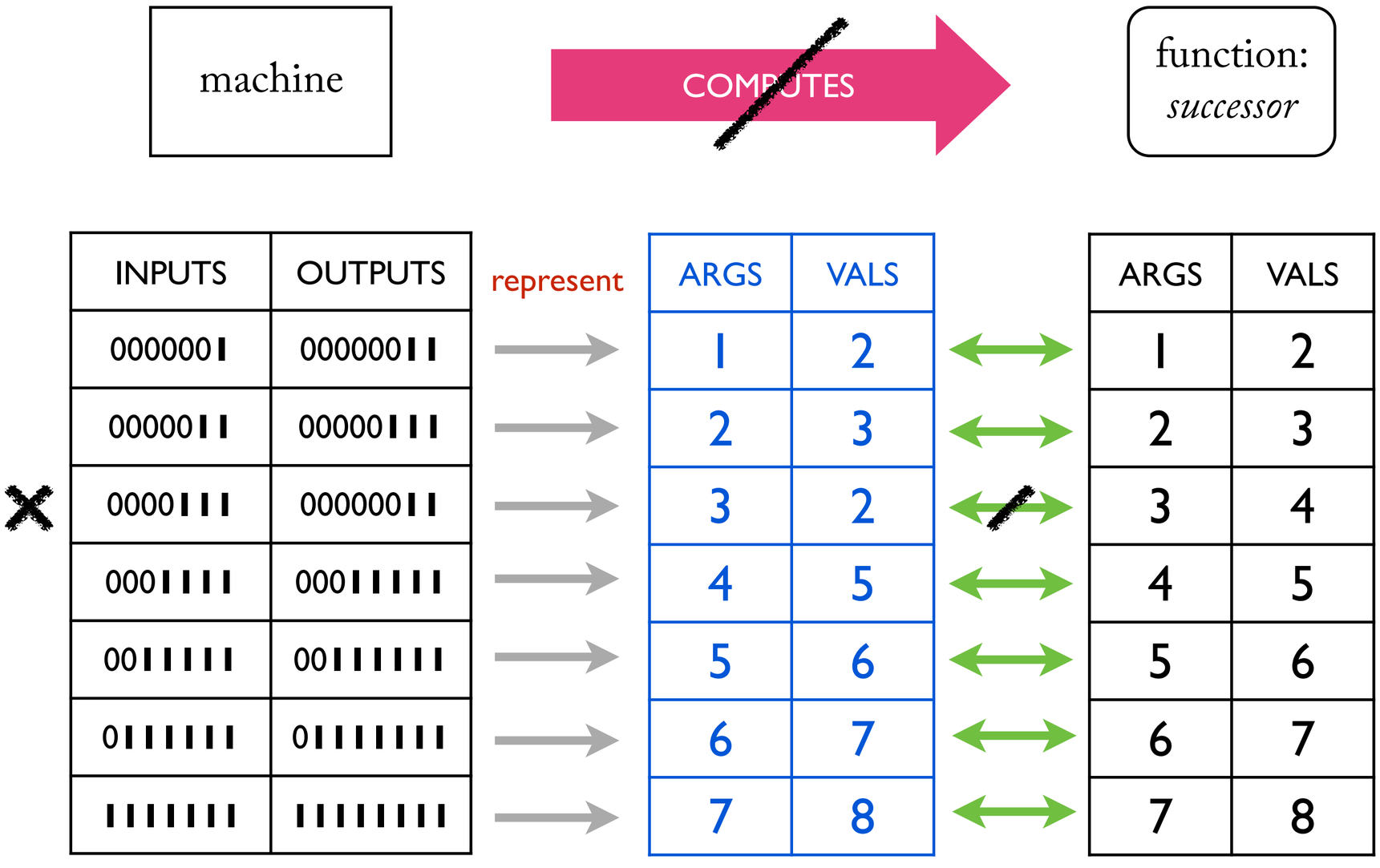
The top span pictures the function instantiated: +, in our present case. It takes a pair of numbers onto their sum. The bottom span corresponds to the function satisfied (called simply *g*). It takes a quintuple of button pressings onto a display. The vertical arrows correspond to interpretation: $I(\langle C, N, +, M, = \rangle)$ is the interpretation of $(\langle C, N, +, M, = \rangle)$, namely $\langle n, m \rangle$, the pair of numbers represented by *N* and *M*. $I(D)$ is the interpretation of the display, which will be the sum of *n* and *m* if the thing works right. Under interpretation, the bottom span is revealed as an instantiation of the top span; computation is revealed as addition.

Examples of computation

The following machine successfully computes the successor function, under the tally representation system. The machine's input-output profile, when interpreted in tally, perfectly matches the argument-value profile of the successor function (so far).



The following machine does *not* computes the successor function under the tally representation system. The machine's input-output profile, when interpreted in tally, does *not* perfectly match the argument-value profile of the successor function.

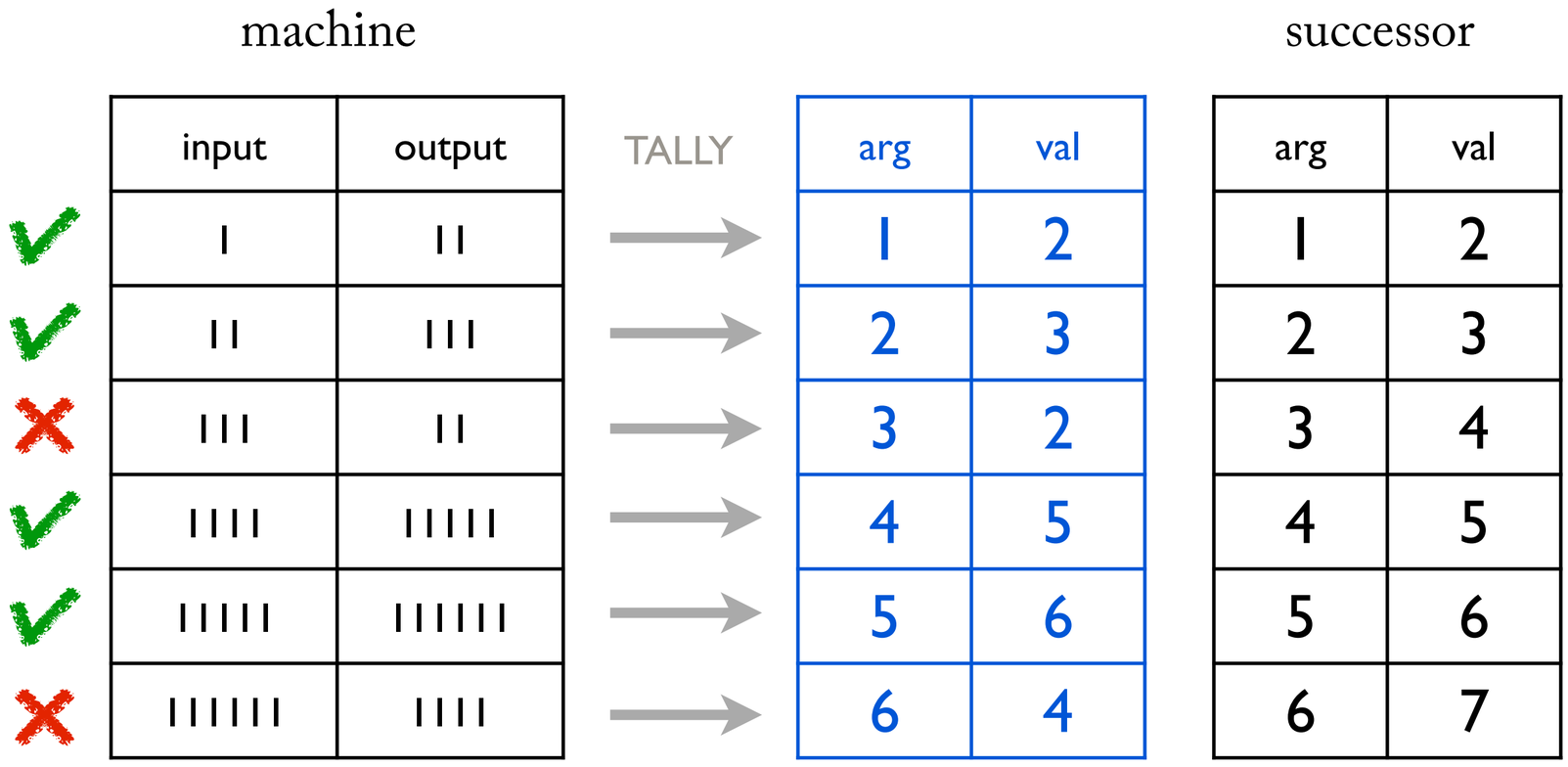


Computing a value

Suppose I ask you to compute the sum of 3 and 4. Clearly, I am asking you to compute the value of the addition function as applied to the arguments 3 and 4. Let's formalize this idea of **computing the value** using the tools we've developed so far.

> **Computing the value** (informal)
>
> Machine M **computes the value** of function *f* applied to argument *x* under representational system *R* if and only if:
>
> if M takes as global input a symbol string which represents *x*,
> then M produces as global output a symbol string which represents *f(x)*.

Suppose I build a machine with the following input output profile. For which arguments does it compute successor under tally?



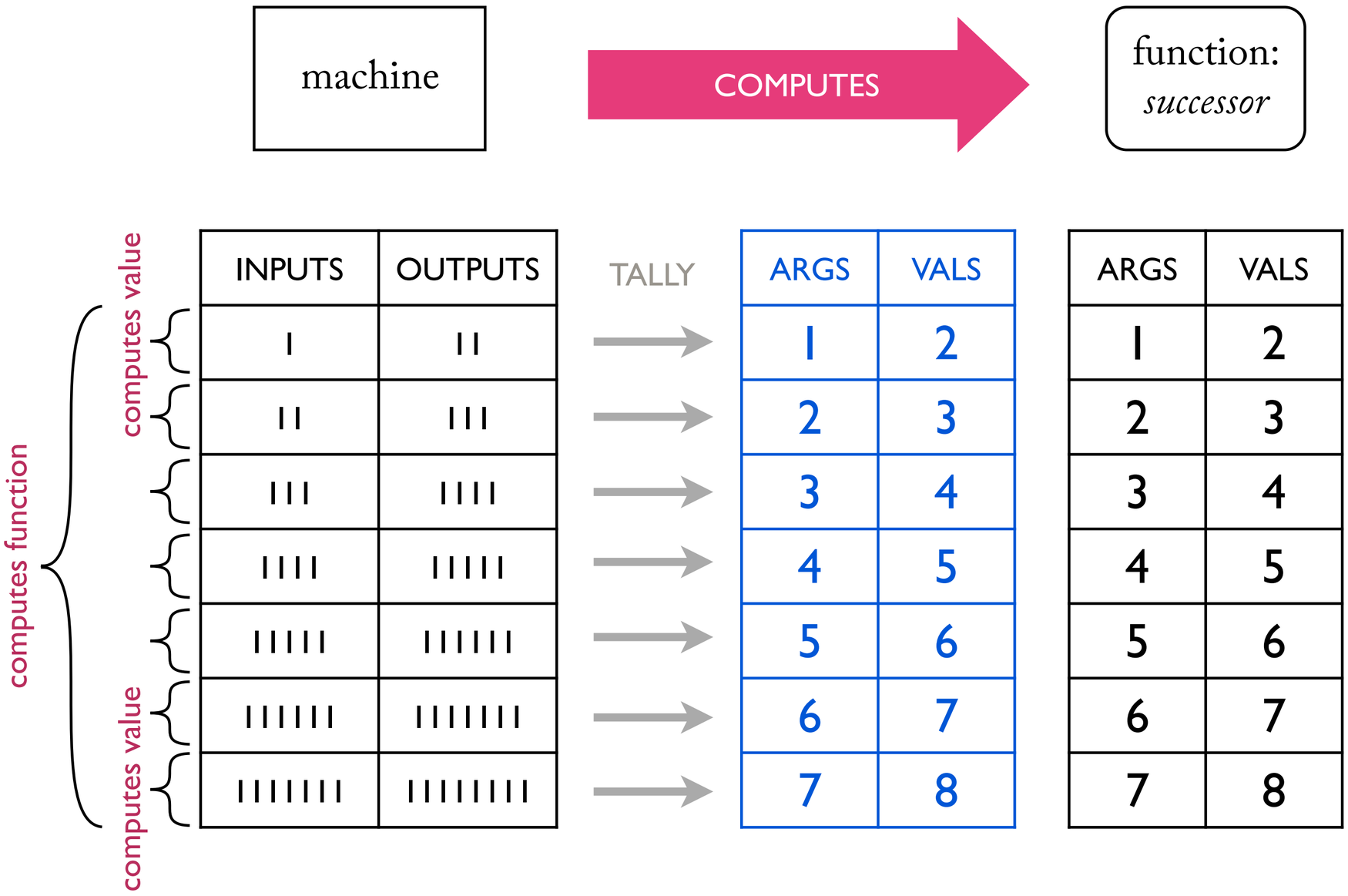
M computes the value of successor applied to 1, 2, 4, 5, and 7.
M does *not* compute the value of successor applied to 3 and 6.

We can define the idea of computing a value more precisely as follows. As an exercise, identify the correspondences between each clause of this definition, and the arrows in our standard diagram of computation.

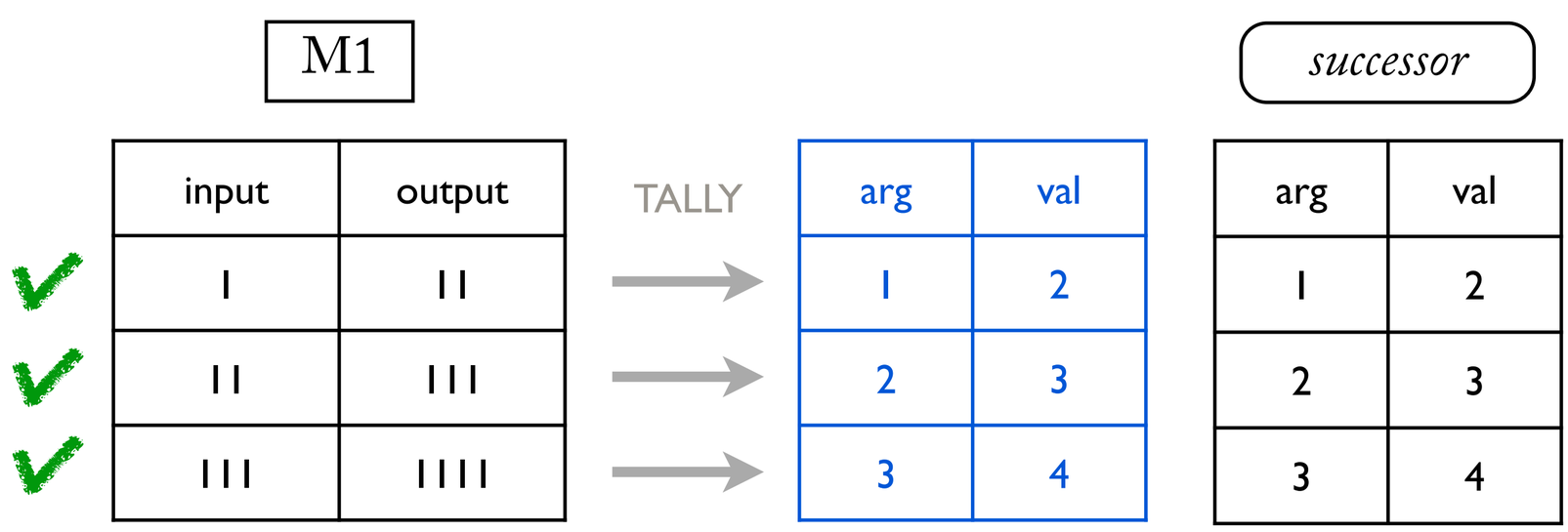
> **Computing the value** (formal)
>
> Machine M **computes the value** of function *f* applied to argument *x* under system R if and only if:
>
> (1) there is a value *y* such that $f(x)=y$;
> (2) there is a symbol string S_1 such that S_1 represents *x* under R;
> (3) there is a symbol string S_2 such that S_2 represents *y* under R;
> (4) if M takes S_1 as a global input, then M produces S_2 as a global output.

Computing a function

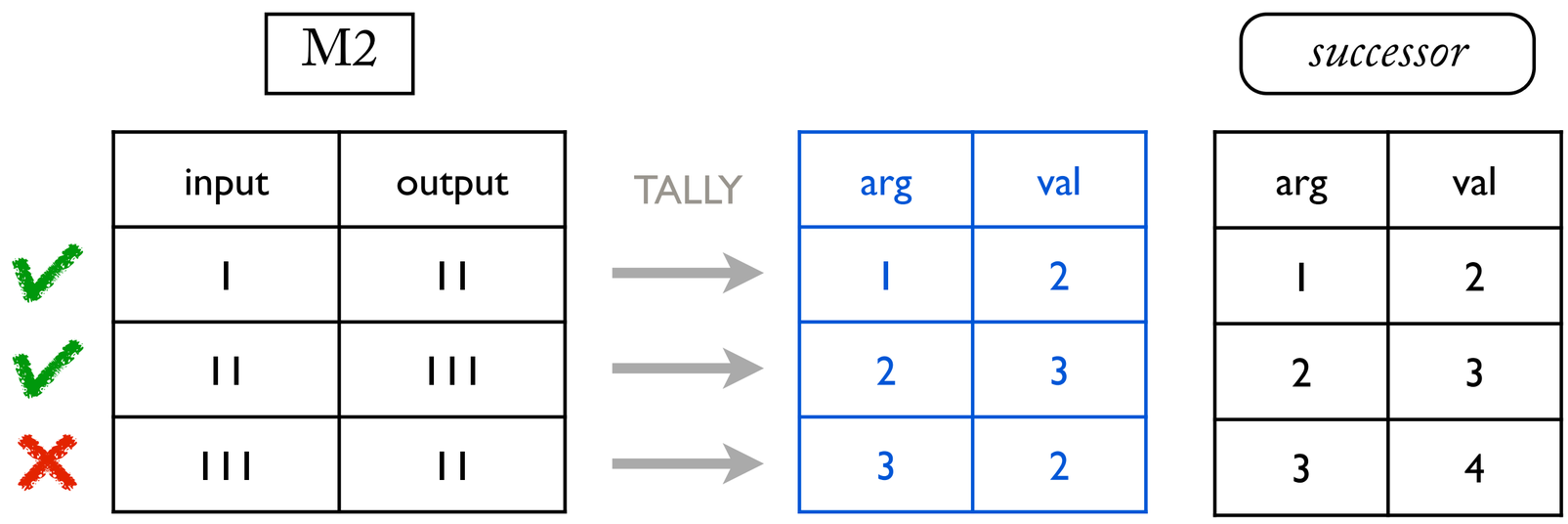
Machine M **computes the function** *f* under system R if and only if:
for every possible argument *x* for function *f*:
M computes the value of *f* applied to *x* under R.



M1 computes successor under tally (so far).



M2 does **not** compute successor under tally.



Bounded Functions

A **bounded** function is a function which is defined for only a finite range of arguments. To name a bounded function, we write:

$f[0\text{-}n]$ e.g. "*successor*[0-3]"

Note: this names a function that is only defined for the *arguments* 0 through *n* (*not* the values). The same notation applies whether the function is 1-place, 2-place, or more. For example the following two tables are equivalent to the named function in each case.

successor[0-2]

arg	val
0	1
1	2
2	3

addition[0-1]

arg1	arg2	val
0	0	0
0	1	1
1	0	1
1	1	2

A machine **computes a bounded function** when it computes that function's values for all *defined* arguments. The values it computes for undefined arguments are irrelevant for now.

Notation for functionally describing computers

We will often want to refer compactly to a computer in terms of what function it computes, and what system of representation it computes it under. This is actually a deep and important form of **functional abstraction**, as we'll see in a few weeks.

I will say that a machine computes

$f[0\text{-}n]$ S

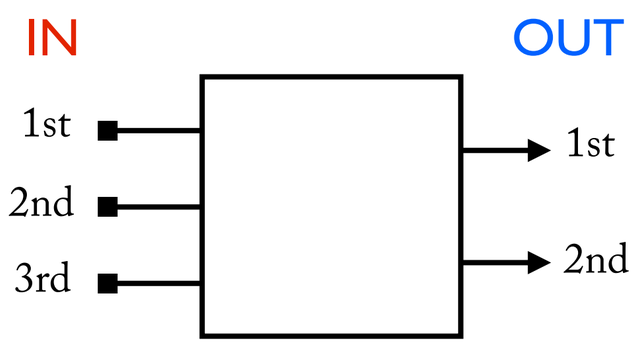
where "$f[0\text{-}n]$" is a (bounded) function, and S is a representational system. Here I will normally use **T** to stand for tally, and **B** to stand for binary. So I will often ask you to solve problems such as:

Design a machine which computes $+1[0\text{-}9]$ B.
Design a machine which computes $2(x+3)$ T.

I'll sometimes omit variables in the definition of the function, if context makes their role clear, as in the first machine described above.

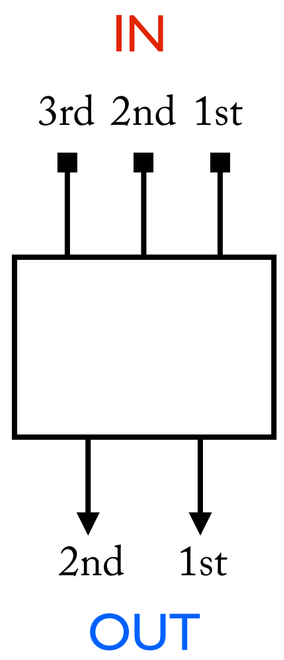
Inputs and Outputs

Input and output. A CC can have any number of inputs and any number of outputs. All the inputs are read at the same time (**synchronically**) and all the outputs are produced at the same time. By convention, the inputs and outputs are ordered from top-to-bottom, or from right-to left (following Arabic).

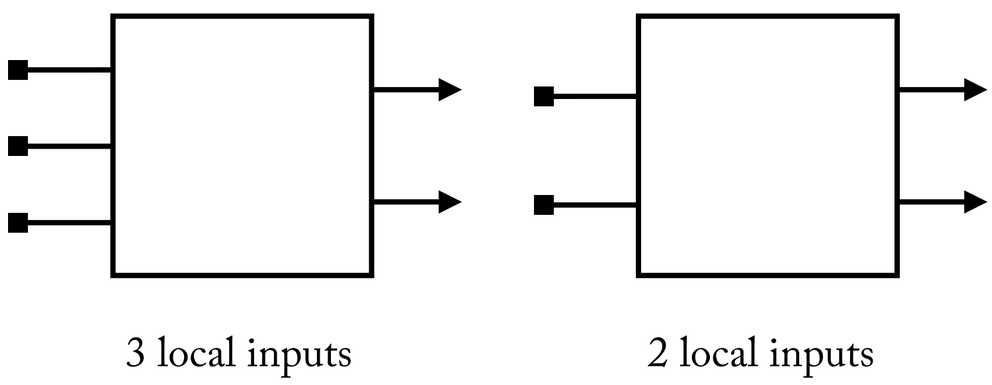


The positions here correspond to positions in a complex numeral:

1 1 0
3rd 2nd 1st



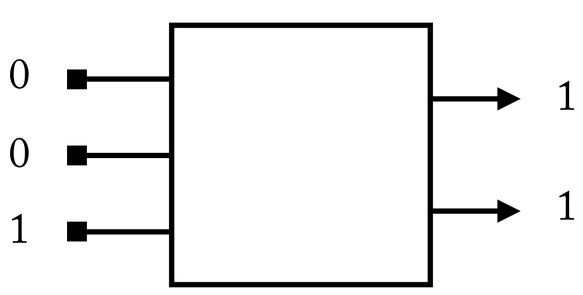
Local input. The **local** inputs to a circuit correspond to the number of input-wires the machine has. Each local input receives one symbol at a given time. A single symbol is called an **atomic** symbol.



3 local inputs 2 local inputs

Global input. The **global** input to the machine is a collection of its local inputs. Where each local input carries an atomic symbol, a global input is the aggregate or **string** of atomic symbols. This string itself is called a **complex** symbol. The **order of local inputs** within a global input corresponds to the **order of symbols** in the input string.

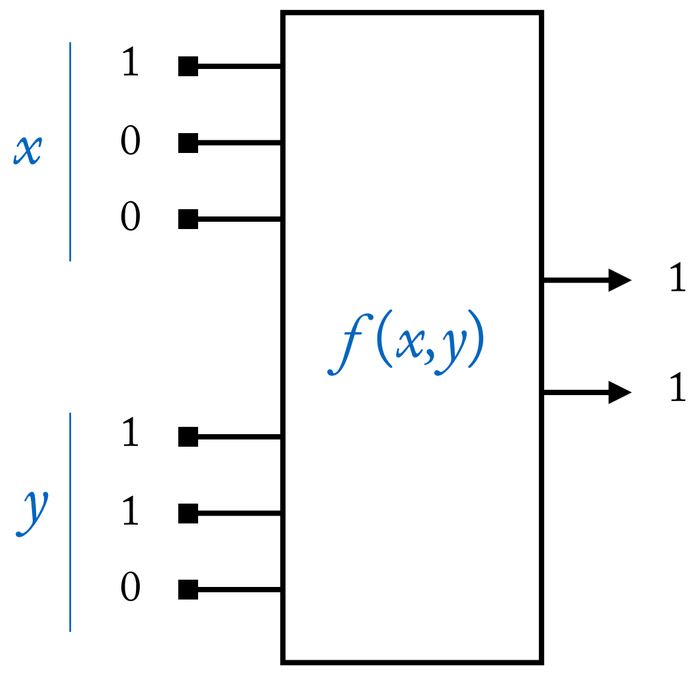
In a CC, the global input is read at one time, by reading each local input synchronically. Later we'll look at circuits where the global input is read over time, **diachronically**.



In the example on the left:
The local inputs are "0", "0", and "1".
The global input is the string "100".

Multiple global inputs. The number of global inputs corresponds to the number of arguments in the function being computed. Each global input represents one argument.

When designing a machine that computes a function with multiple argument places, indicate clearly which global input goes with which argument. The ordering convention again is top-to-bottom, or right-to-left.



In-class Problems

1. Suppose M1 computes x+1[0-4] T
   a. How many local inputs and outputs must M1 have?
      How many global inputs and outputs does M1 have?
   b. Make a global input-output table for M1. (Restrict your table to possible inputs and outputs.)
   c. Design a circuit for M1.
2. M2 computes x+1[0-5] T. Design a circuit.
3. M3 computes x+1[0-1] B. M3 has 1 local input and 2 local outputs.
   Make a global input-output table. Design a circuit.
4. M4 computes x+1[0-2] B. M3 has 2 local inputs and 2 local outputs.
   Make a global input-output table. Design a circuit.
5. M5 computes x+1[0-3] B. Make a global input-output table. Design a circuit.

Tip: for all these problems, turn your circuit 90° so the inputs and outputs are vertically aligned, the same way you would set up a long addition problem.

7. Computation as a 3-way Relation

The structure

Computation is a relation between a **machine** M, a **system of representation** R, and a **function** *f*.

Given any two, you can solve for the third.
M + R = *f*?
M + *f* = R?
f + R = M?

Most of the problems I will assign are of the third kind.

The three elements are logically independent. (R)

The same M can compute different functions *f* and *f** relative to different systems of representation R and R*.

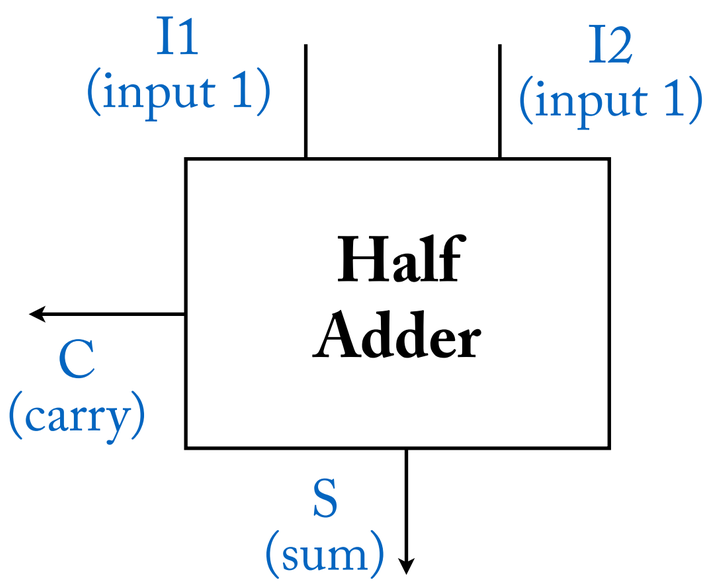
The same function *f* can be computed by different machines M and M* relative to different systems of representation R and R*.

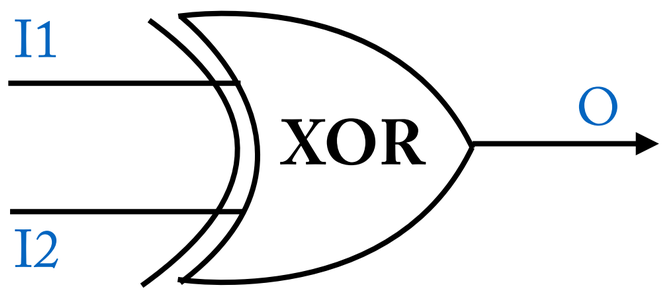
(Section on Representation: choice of representational system matters. Tie in to Newell)

8. Functional Abstraction

Diagramming abstraction

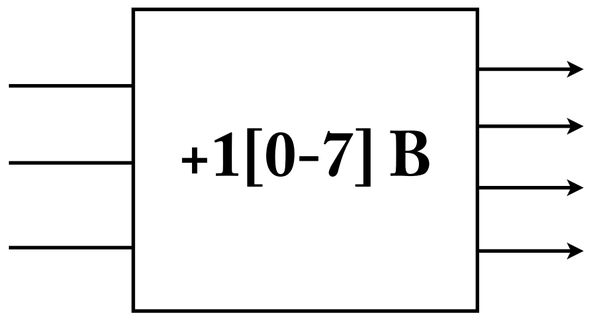
Once you've shown how to build an XOR and Half-Adder from logic gates you can go ahead treat these like logic gates themselves. I'll call these **boxed** circuits. That is, you can use elements like these in your circuits:





By the way: if you use a Half-Adder, it is very important to label your "carry" and "sum" outputs.

In general, once you have defined a circuit for a given function (say, binary +1[0-7]), you can use that circuit later as part of other circuits. For example:



9. Function Composition

It's a familiar fact that functions can be "stacked together" or **composed**. For example, I could define *f* as:

$$f(x) = (x + 3) \bullet 2$$

Here I've defined *f* in terms a +3 function and •2 function. Put another way, I've defined *f* as the composition of +3 and •2.

Because functions are identified only by their arguments and values, the same function can be composed in different ways. In fact, "a composition" of a function is really just a set of instructions to the reader for how to identify that function. (It is very much like a computer program for your mind to run.) The same function can be specified using different sets of instructions. For example, consider the function *g* where:

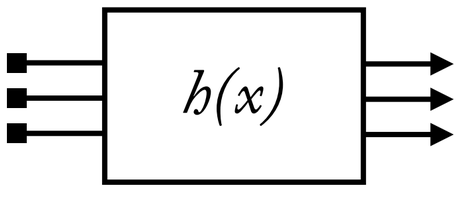
$$g(x) = x+7 = (x+5)+2$$

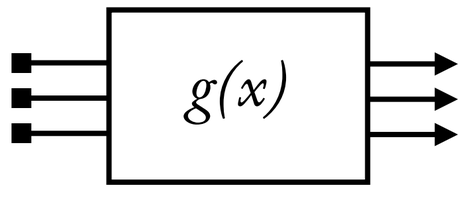
Here we describe *g* in two different ways, by composing different functions. Can you think of others we might have used?

Machine Composition

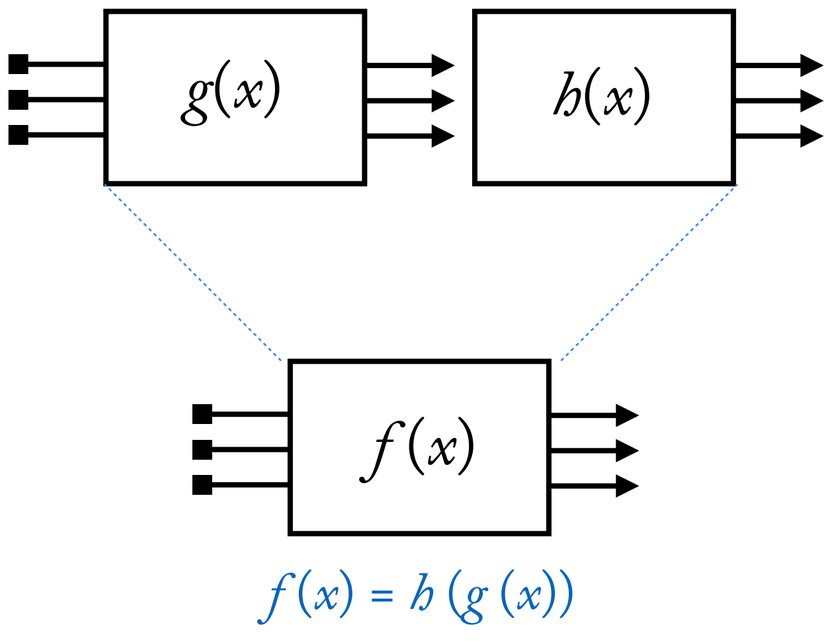
Any time a function f can be defined in terms of the composition of functions g and h—then a *machine* which computes f can be designed by *putting together* a machines which computes g and a machine which computes h. (Actually, this is only true for certain ways of describing machines. But it holds for the circuit-based descriptions we're considering right now.)

For example, suppose: $f(x) = h(g(x))$. And suppose you have a CC that computes h and a CC that computes g:





Then you can construct a machine that computes f by *combining the machines themselves*. Like this:

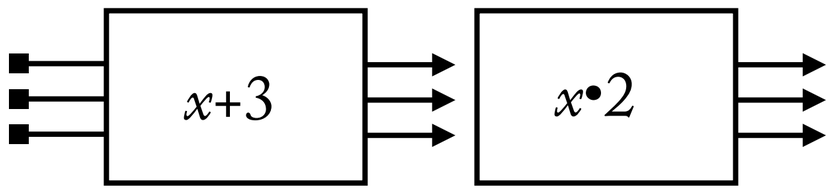


You'll have to be careful about the number of inputs and outputs lining up properly in each case. In the illustration here, I've abstracted away from this (important!) detail.

Concretely, applied to the example we started with, suppose:

$$f(x) = (x + 3) \bullet 2$$

And suppose you have machines that computes $x + 3$ and $x \bullet 2$. Then you can design a machine which computes f by putting together your $x + 3$ machine and your $x \bullet 2$ machine as follows:



You'll want to think about and experiment with these principles to make sure they work for both tally and binary, how to deal with multiple argument places, and other engineering questions that may arise.

Just as there are different ways of composing the same function, there are different ways of combining machines to compute the same function. If you have machines that compute +1, +2, and +5, can you describe different machines for computing $x + 7$?

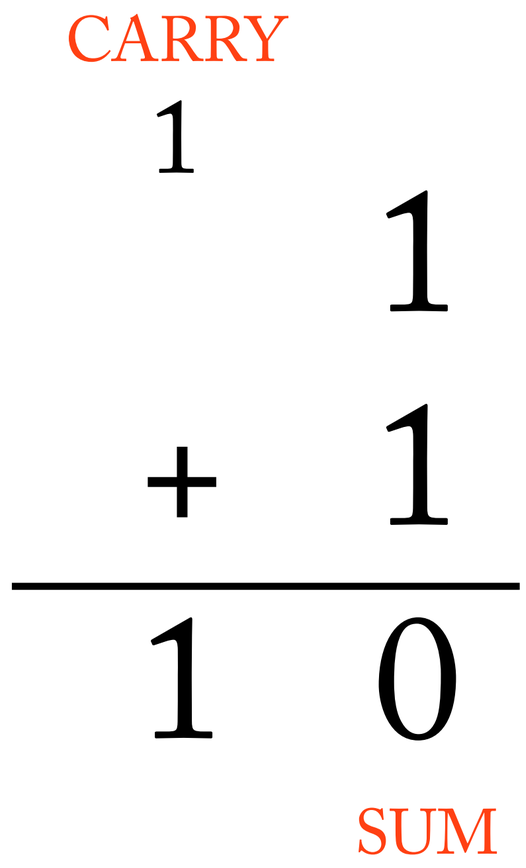
Note! When composing functions, and putting machines together, *order matters*. (Recall the order of operations PEMDAS from middle school arithmetic!) For example, $2 \bullet (3+x) \neq 3+(2 \bullet x)$. This means that order also matters in the way you put together your machines to derive a more complex machine. I'll leave it to you to determine exactly *how* order matters here.

10. The Story of Binary Successor

The Problem: Binary Successor

For computer scientists, the computation of binary successor is the basis for nearly all more sophisticated computations.

Intro: Long Addition in the Simplest Case

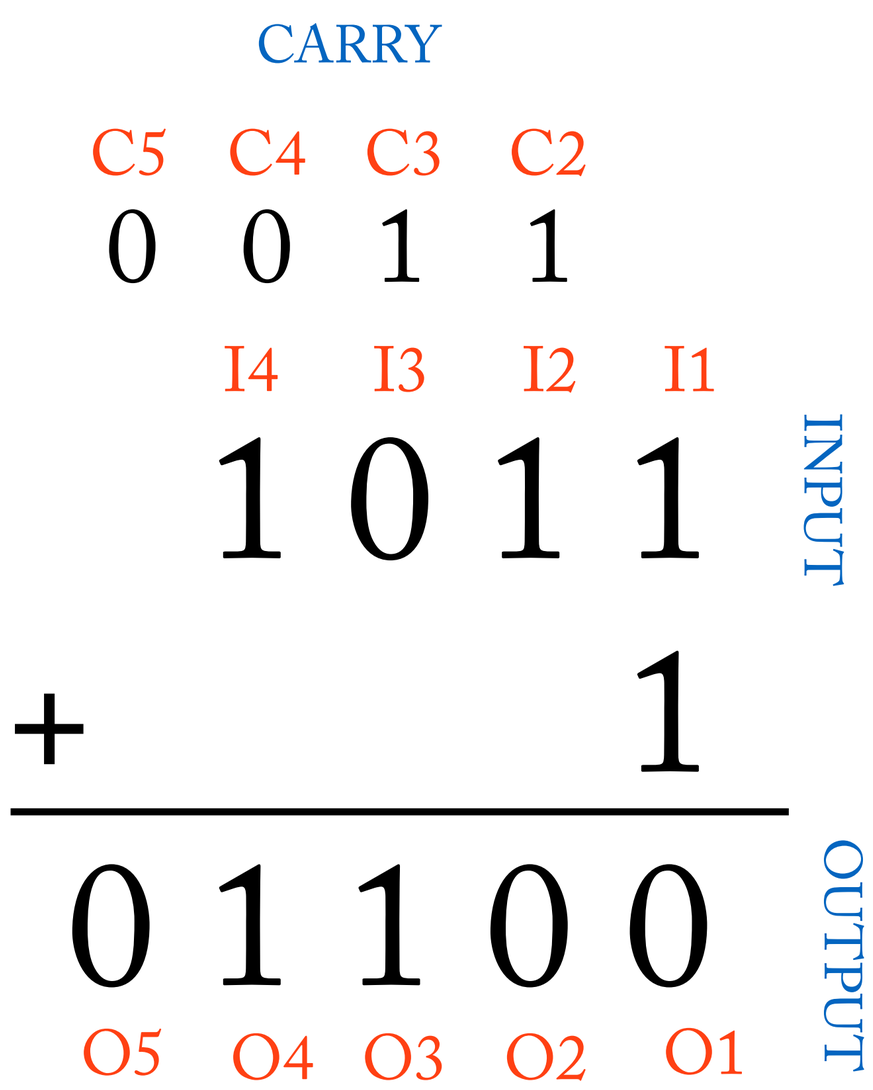


A *bit* = a single binary digit (or numeral).

The result of the initial sum is two bits long, which we break into parts.

The *first* bit is called the **SUM** bit (or S bit). The *second* bit is called the **CARRY** bit (or C bit).

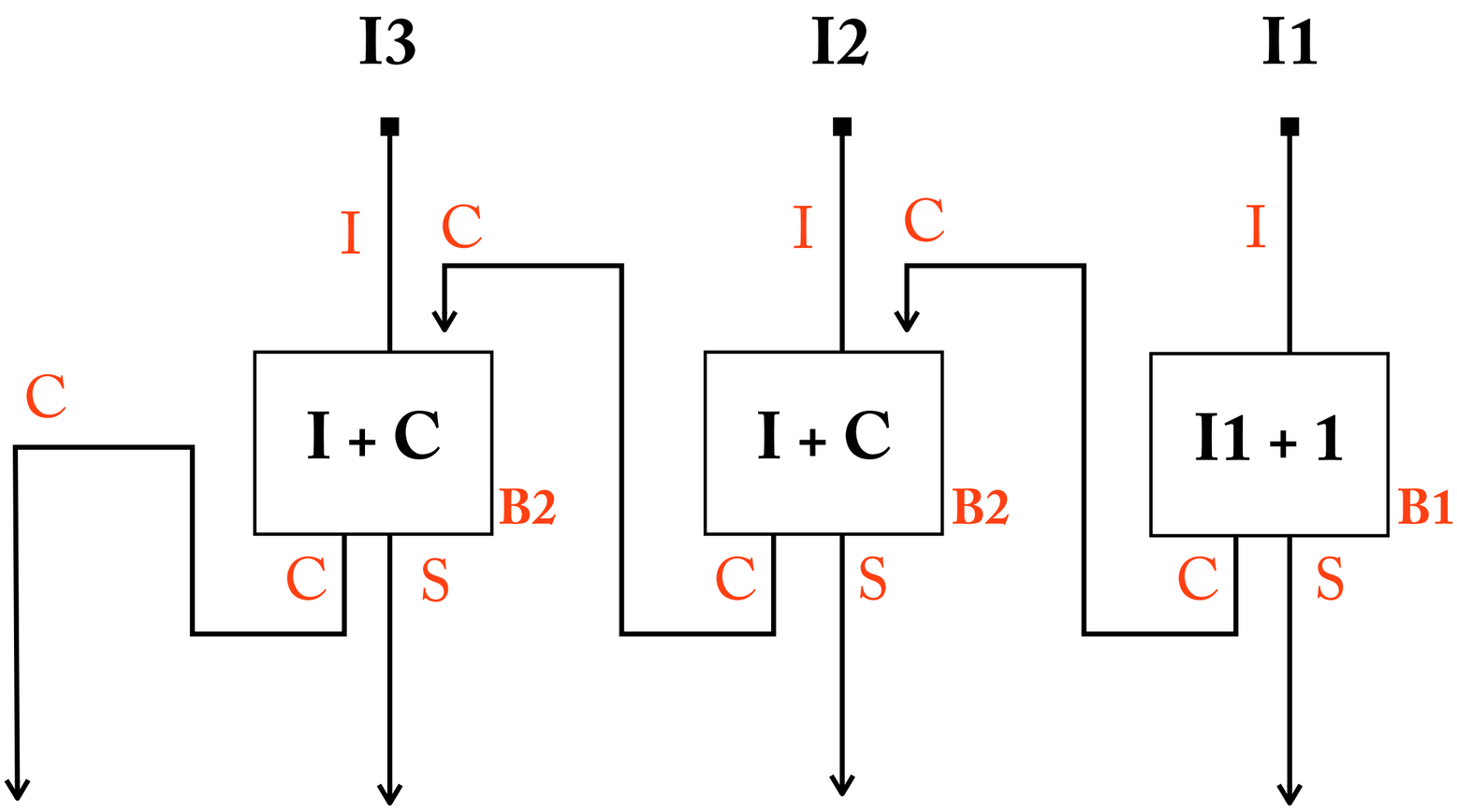
Step 1. Identify a method (carrying)



Binary successor algorithm

1. Compute I1 + 1.
   Results in a SUM bit and a CARRY bit.
2. O1 = SUM.
3. Compute I_{NEXT} + CARRY.
   Results in a new SUM bit and a new CARRY bit.
4. O_{NEXT} = SUM.
5. Repeat from 3, as long as there are inputs.
6. When there are no inputs left, O_{FINAL} = CARRY.

Step 2. Break it into parts



We could keep doing this forever...!

Step 3. Analyze the parts

Box 1: I1 +1

Input	Sum	Carry
1	0	1
0	1	0

Box 2: I +C

Input	Carry (IN)	Sum	Carry
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

Next you must design a circuit corresponding to each box….

Interlude: Half-Adder

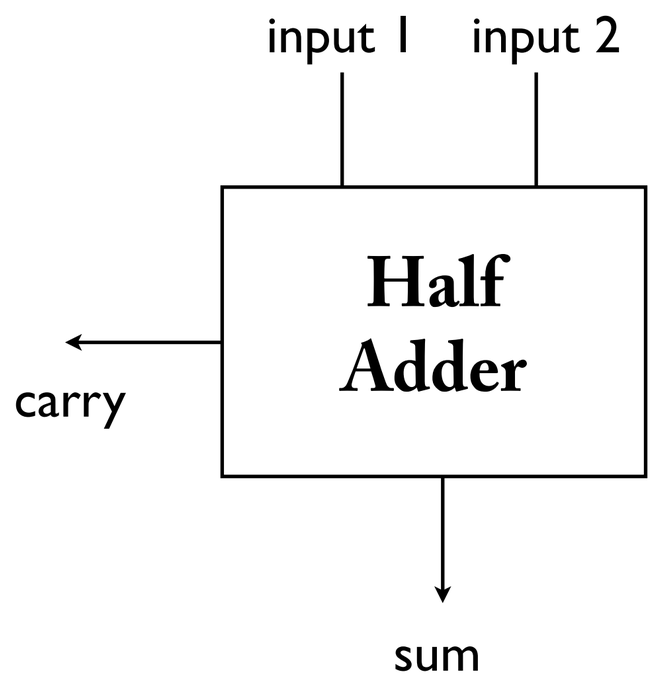
The Box 2 circuit is especially useful and important. It performs a key action: it takes any two bits as inputs, and outputs a SUM bit and a CARRY bit.

This mini-circuit is known as a "half adder". Half adders form the heart of any addition-based computation.

It is only *half* an adder, because it only adds two bits, rather than two bits *plus* a carry bit.

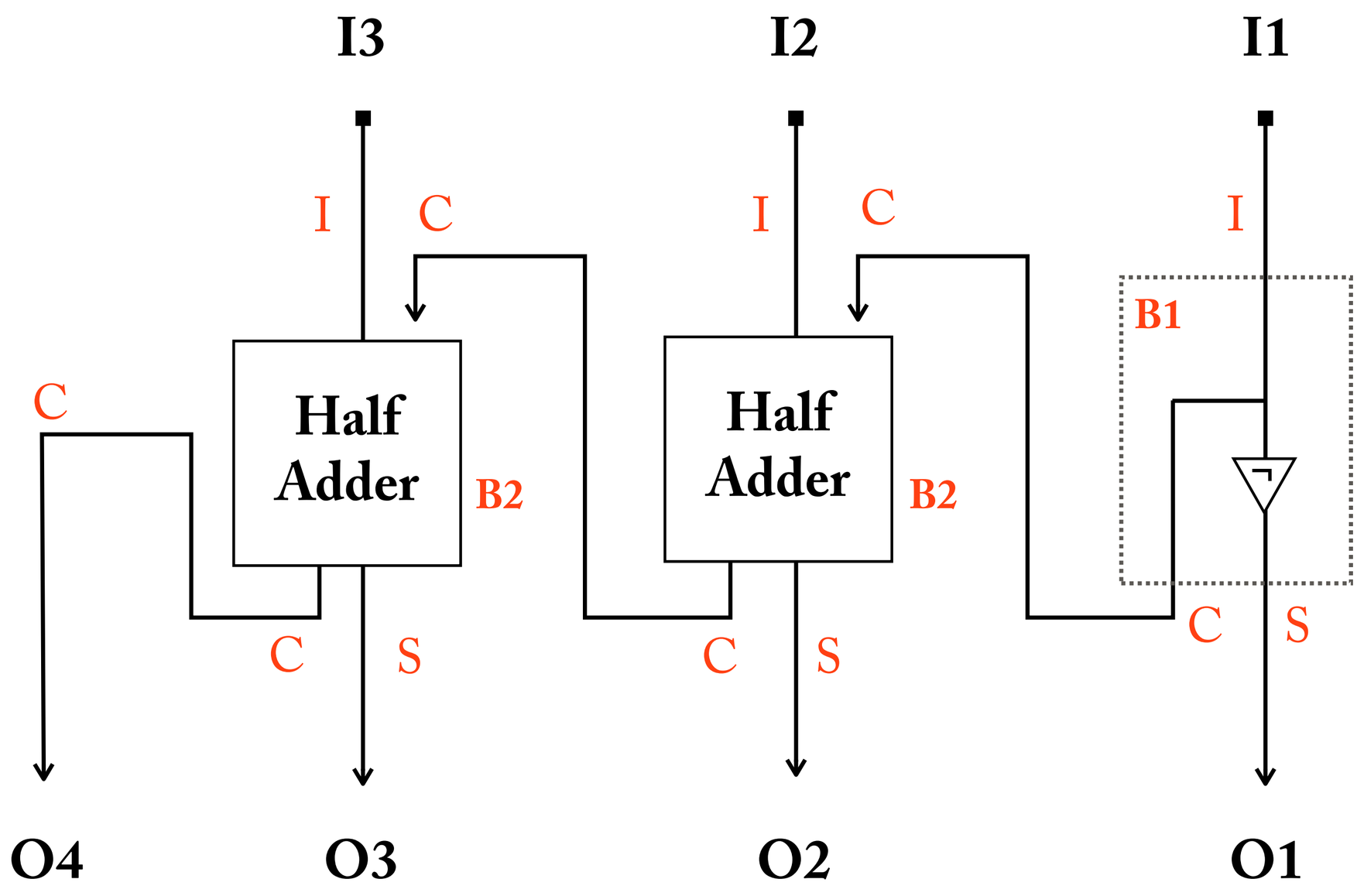
NOTE: in order to use half adders in your homework solutions, you'll first have to show how a half adder is built from logic gates.

Schematically, it looks like this:



Always label your SUM and CARRY outputs!

Step 4: A schematic solution



11. Introducing Turbots

Embodied Computation

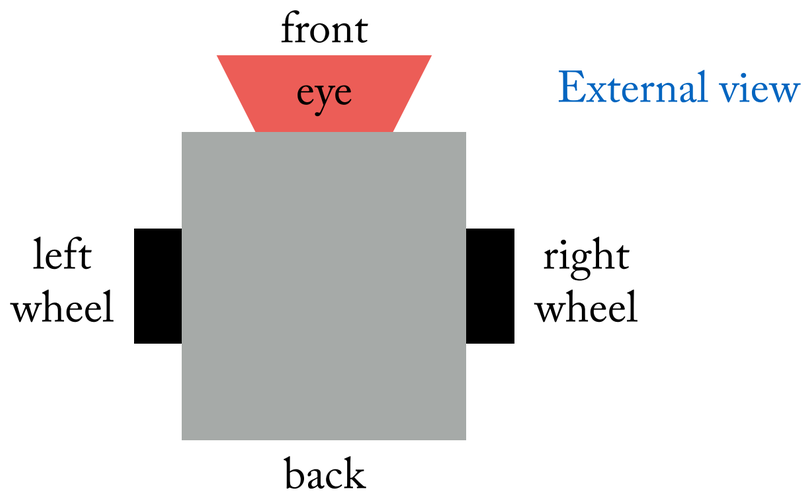
The central fact about cognition is that it allows an organism to interact meaningfully with its environment. Cognition mediates between an organism's ability to sense the environment and its ability to act on it. We can't claim to be studying cognition by only focusing on mathematical computation, even if that is an important part of the story. In fact, many believe that for a computer to exhibit *genuine* mental representation, it must be part of a fully embodied organism that causally interacts with the physical world.

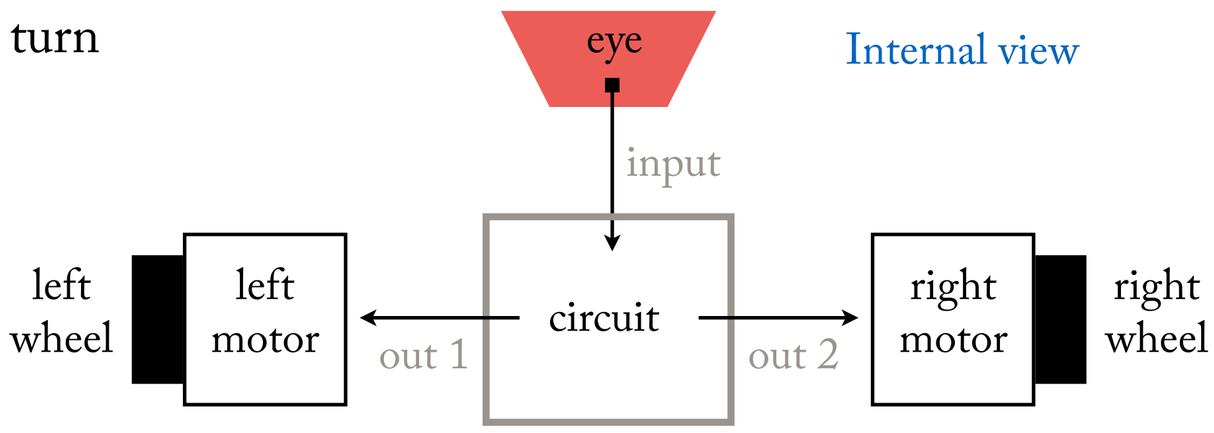
For these reasons, we will now begin to investigate computational problems of navigation for organisms capable of perception and action, alongside the more abstract, traditional study of mathematical computation.

Turbots

A **turbot** (from "Turing" + "robot") is a simple robot with a single electronic eye, two wheels powered by two motors, and an internal circuit which processes signals from the eye, and issues signals to the two motors.

For perception, turbots are limited to seeing the object immediately in front of them; for action, they can only turn left or right, go straight, or stop.



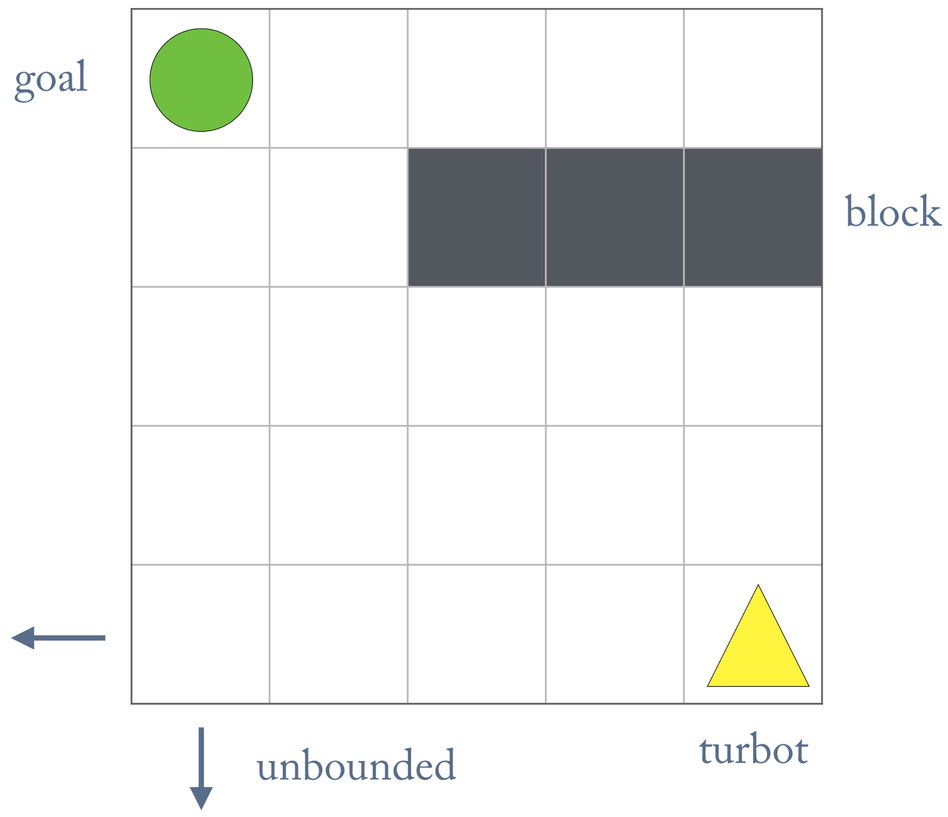


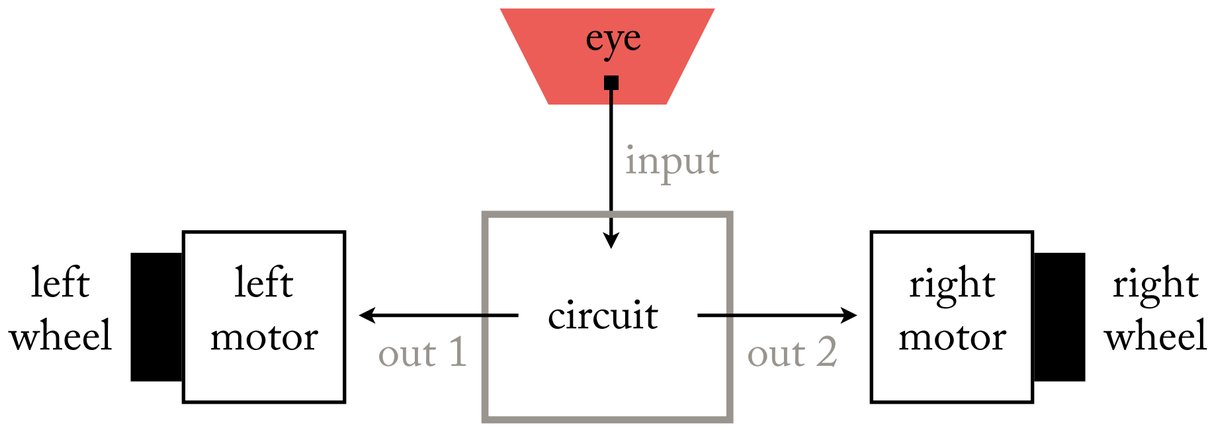
Environment

Turbots live an unbounded grid-world that includes the turbot, unmovable blocks, and (typically) a goal.

The position and orientation of the turbot is indicated by the yellow triangle.

For navigation problems, the yellow triangle indicates the turbot's starting point. Typically, navigation is "completed," and the problem solved, when the turbot *crosses over* the goal. It does not need to stop on the goal. (Think of Pac-Man.)





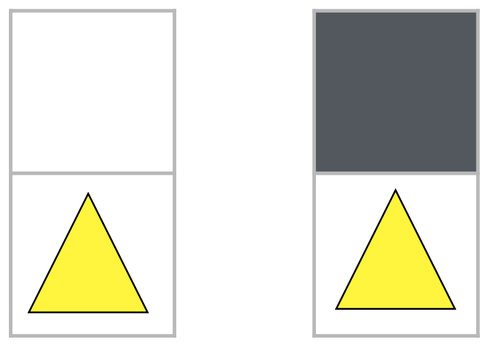
Internal view of a turbot

Progression of time

During each unit of time, or **cycle**, the turbot "instantly" (1) receives a new input; (2) processes it; (3) produces an output; (4) acts on that output. At the end of each cycle the circuit is "wiped clean" of any information, but the circuit itself persists through time, and the process begins again. (By contrast, for CC's that compute mathematical functions, we only consider their operation at a single moment in time, for a single input.)

Input: perception

The input to the turbot comes through the eye. At each cycle, the eye can only see the cell in front of the turbot at that time. And it can only detect the absence (input = 0) or presence (input = 1) of a block.



empty
input = 0

block
input = 1

Turbot perception

empty → 0
block → 1

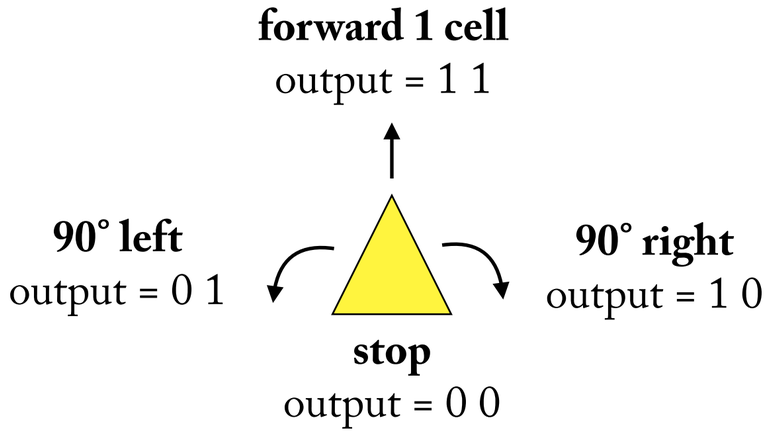
Output: action

The output from the turbot goes through the two motors and on to the wheels. An output of 1 to a motor will turn it on for 1 cycle, while an output of 0 will turn it off for 1 cycle.

If both motors receive a 1, the turbot moves forward 1 cell. If both motors receive a 0, the turbot stops in the current cell.

If the *left* motor receives a 1 and the *right* motor receives a 0, the turbot will rotate 90° *to the right*, but it will not change cells. (This makes mechanical sense if you think about it.) The opposite is also true: if the *right* motor receives a 1 and the *left* motor a 0, the turbot as a whole will rotate 90° *to the left*.

The only way to turn 180° is to apply two 90° turns over two cycles.



Turbot action

1 1 → forward 1 cell
0 1 → 90° left
1 0 → 90° right
0 0 → stop

A Simple Example

Consider the following problem:

> Design a turbot which moves straight ahead until it reaches a block then stops.

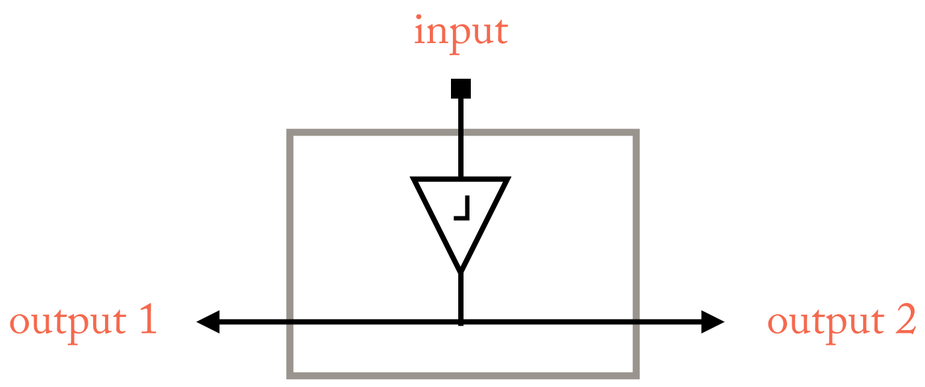
To satisfy this description, the turbot must repeatedly apply the following rule:

> If perception = **empty**, then action = **move forward**.
> If perception = **block**, then action = **stop**.

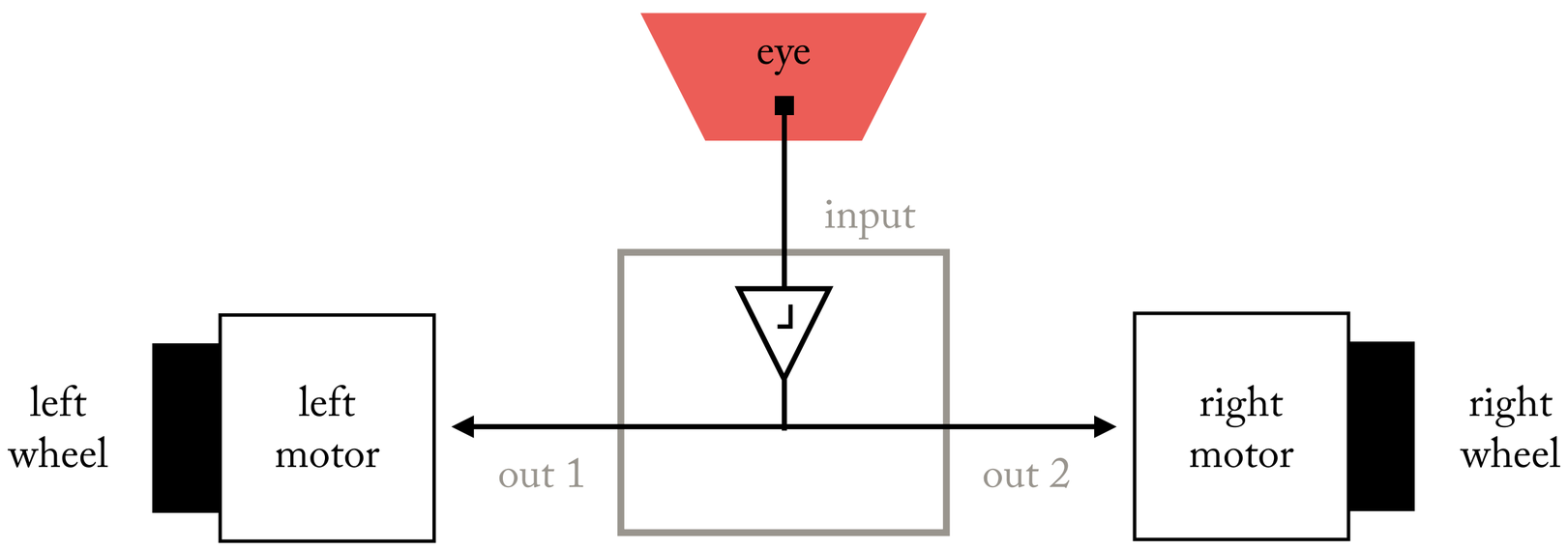
In terms of inputs and outputs:

> If IN = **0**, then OUT = **1 1**.
> If IN = **1**, then OUT = **0 0**.

It is now trivial design such a circuit:



Installed into a turbot, this circuit solves the problem we started with.



12. Real and Ideal Computers

1. Symbols and numbers.
2. Error and accuracy.
3. Finity and infinity.
4. Actuality and possibility.
5. Computing a function vs. instantiating (or satisfying) a function.

   “Humans, in particular , routinely satisfy functions by computing them.”

   (Cummins 1991, p. 91)

“But + is a function whose arguments and values are numbers; and whatever numbers are, they are not states or processes or events in any physical system. How, then, can a physical system be described by +? How can a physical system traffic in numbers and hence add? The answer, of course, is that numerals— i .e., representations of numbers — can be states of a physical system, even if the numbers themselves cannot. A physical system adds by trafficking in numerals and hence, indirectly, in the numbers those numerals represent.” (Cummins 1991, p. 89)

“From an explanatory point of view, what interpretation provides is a link between mere state crunching (button-pressing-to-display transitions) and addition: Under interpretation , the state transitions of the system are revealed as adding. You can get a feel for the explanatory role of interpretation by imagining that the device is an archaeological discovery. Perhaps you know from documents that the thing is an adding machine; however, looking at it, you have no idea, at first, how to give inputs or read outputs. You can “make it go,” let’s say, by turning a handle, but this is just more or less random manipulation until you find an interpretation that reveals the state transitions of the device as sum calculations and hence establishes the representational status and contents of those states. To know how to use it is to know how to see it as simulating +, and that means knowing how to “make it go” and how to interpret it.” (Cummins 1991, p. 94)

13. Craik's computational theory of mind

Craik's Hypothesis on the Nature of Cognition

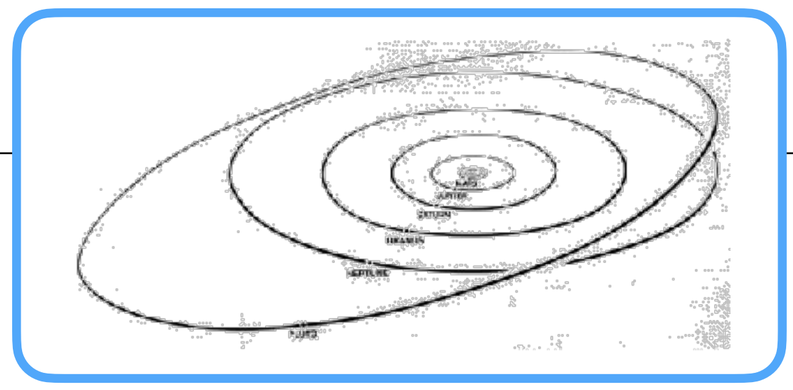
Writing in 1943, on the cusp of what would soon be recognized as the "cognitive revolution," Craik envisioned a mechanistic theory of the mind:

> "My hypothesis then is that thought models, or parallels, reality— that its essential feature is not 'the mind', 'the self', 'sense-data', nor propositions but symbolism, and that this symbolism is largely of the same kind as that which is familiar to us in mechanical devices which aid thought and calculation." (57)

There are two parts to this hypothesis: (i) that "thought models, or parallels, reality"; and (ii) that this modeling is achieved through "symbolism" analogous to that of calculating machines.

Physical Models of Reality

Let's start with the observation that we often use physical models to help understand other phenomena. Like words or pictures these models represent external events. But they do more than that. They change and evolve to match the way the represented system changes and evolves.



> The physical process which it is desired to predict is *imitated* by some mechanical device or model which is cheaper, or quicker, or more convenient in operation. (51)

Thought as a Model of Reality

Craik proposes that thought may have a similar function of modeling external processes.

> Human thought has a definite function; it provides a convenient small-scale model of a process so that we can, for instance, design a bridge in our minds and know that it will bear a train passing over it instead of having to conduct a number of full-scale experiments; and the thinking of animals represents on a more restricted scale the ability to represent, say, danger before it comes and leads to avoidance instead of repeated bitter experience. (59)

But he notes that this can't be achieved by actually *duplicating* the external object within.

> This process of reasoning has produced a final result similar to that which might have been reached by causing the actual physical processes to occur (e.g. building the bridge haphazard and measuring its strength or compounding certain chemicals and seeing what happened); but it is also clear that this is not what has happened; the man's mind does not contain a material bridge or the required chemicals (51)

Symbolic Models of Reality

The key insight is to realize that modeling can be achieved by "relation-structures" involving symbols.

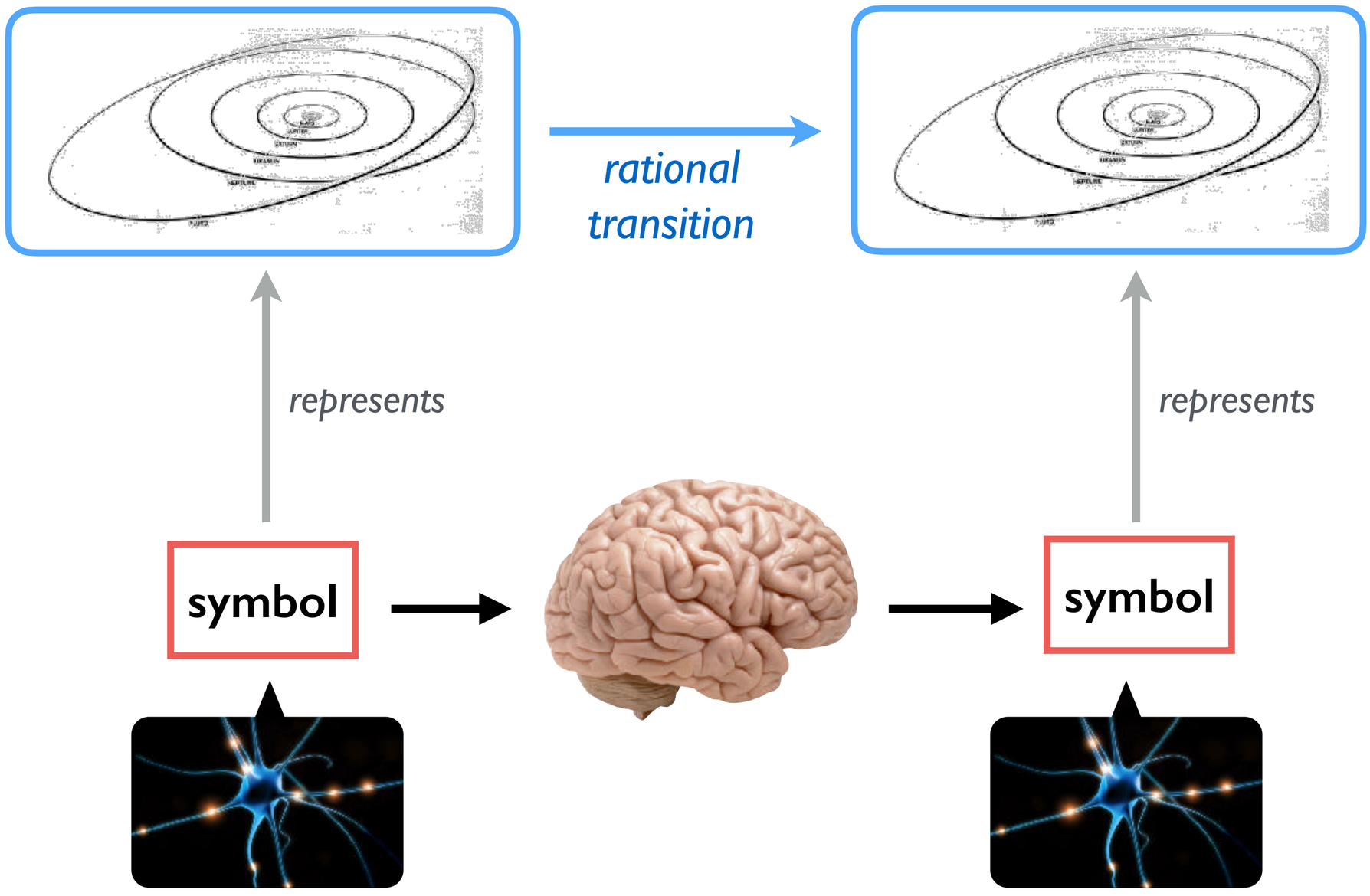
> By a model we thus mean any physical or chemical system which has a similar relation-structure to that of the process it imitates. By 'relation-structure' I do not mean some obscure non-physical entity which attends the model, but the fact that it is a physical working model which works in the same way as the process it parallels, in the aspects under consideration at any moment. Thus, the model need not resemble the real object pictorially. (51)

This leads to Craik's core hypothesis:

> "The process of thought, reduced to its simplest terms, is as follows…
> (1) 'Translation' of external process into words, numbers or other symbols,
> (2) Arrival at other symbols by a process of 'reasoning', deduction, inference, etc., and
> (3) 'Retranslation' of these symbols into external processes (as in building a bridge to a design) or at least recognition of the correspondence between these symbols and external events (as in realising that a prediction is fulfilled)." (50)

And the central analogy between minds and computers:

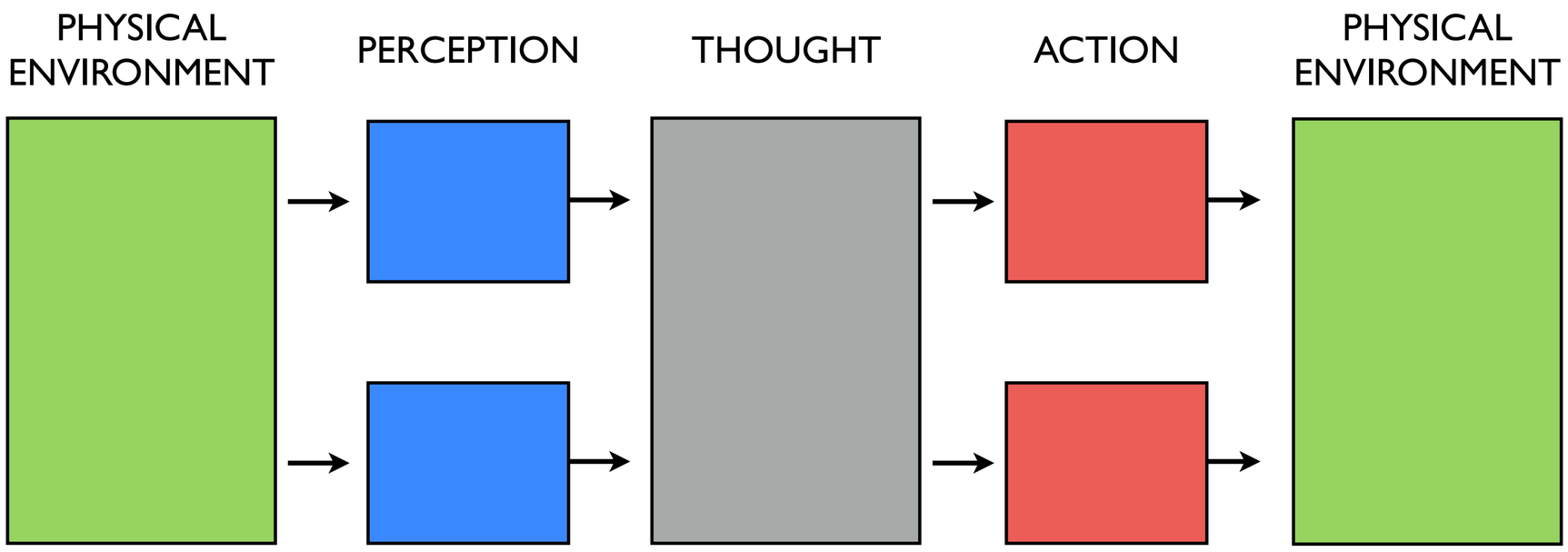
> I have tried… to indicate what I suspect to be the fundamental feature of neural machinery— its power to parallel or model external events— and have emphasised the fundamental role of this process of paralleling in calculating machines. (52)

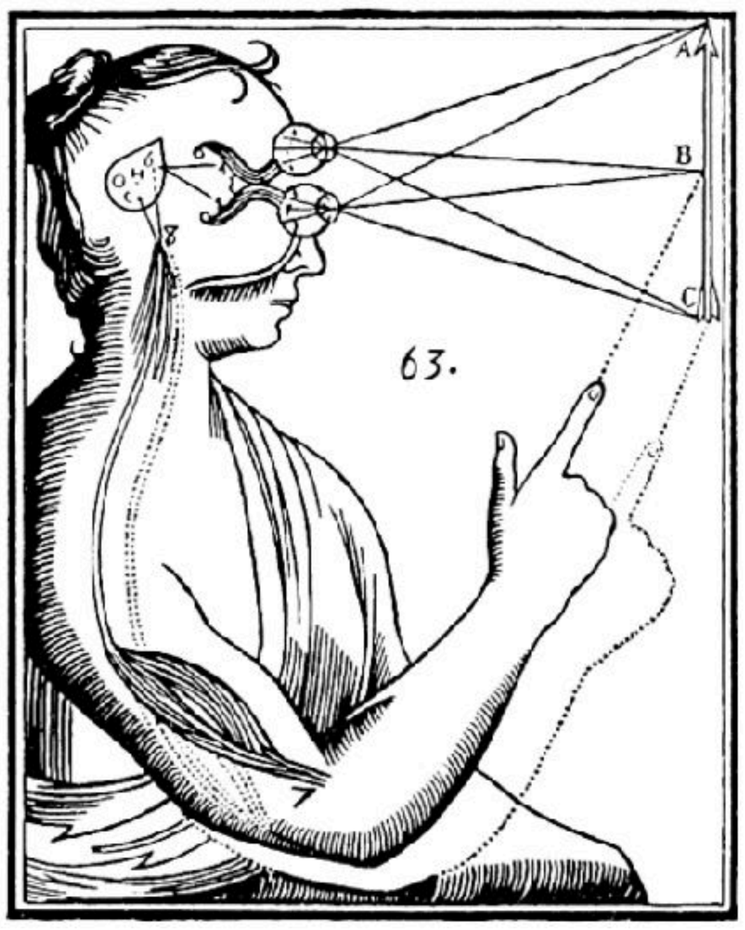


Mind as Input/Output Machine

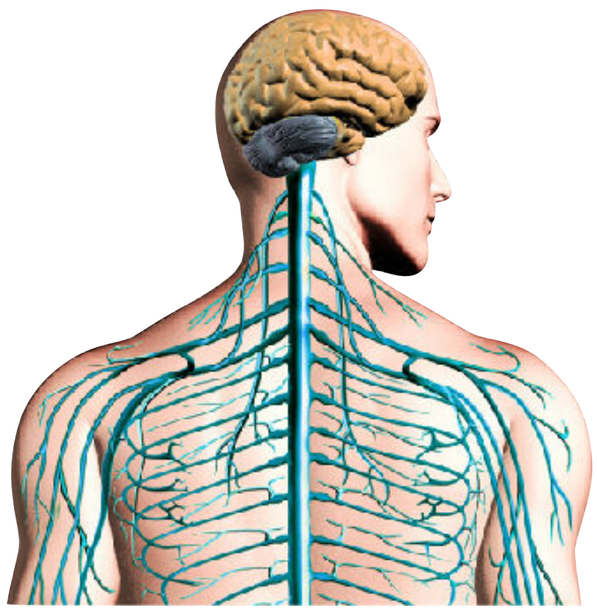
The process of thought, reduced to its simplest terms, is as follows…
(1) 'Translation' of external process into words, numbers or other symbols,
(2) Arrival at other symbols by a process of 'reasoning', deduction, inference, etc., and
(3) 'Retranslation' of these symbols into external processes. (50)

This three-part division of cognition is as old as Descartes, and it is the foundation of contemporary cognitive science:

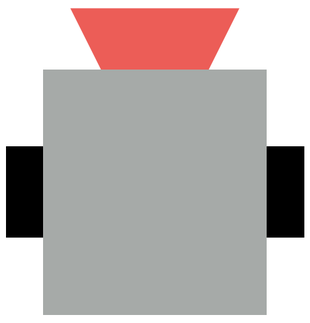




For Descartes, the mind was made up of the *passions*, *thinking*, and *actions* of an immaterial soul.



Neuroscience recognizes a central nervous system with inputs from the sensory systems and outputs to the motor systems.



Even in the humble turbot, simple perception and action abilities are mediated by logic circuits.

14. Artificial and Natural Computers

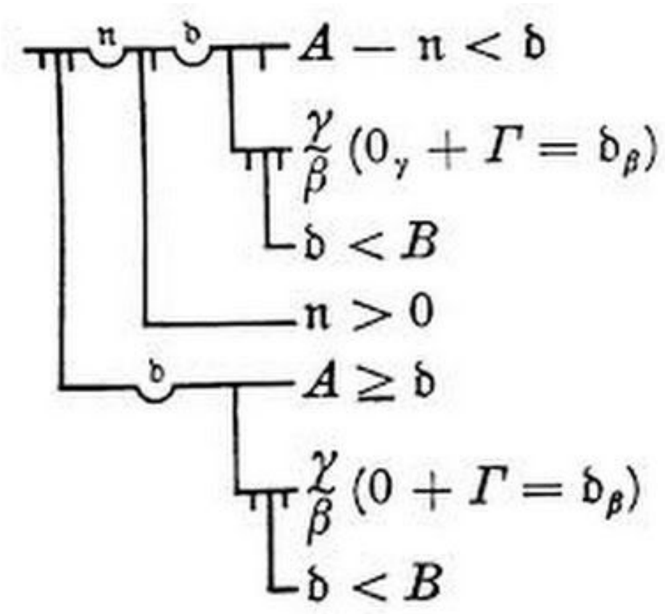
The Invention of Computers

Why create a computer? Why isn't human cognition enough? Why appeal to computation at all? Speed. Accuracy. Reliability. Scalability. Cost. Humans have developed automated computing machines for the same reason we have created any number of automated devices, from the egg beater to the steam engine.

Leibniz himself imagined a machine that would be faster and more reliable than human calculators.

> "If controversies were to arise, there would be no more need of disputation between two philosophers than between two accountants. For it would suffice to take their pencils in their hands, and say to each other: *Calculemus*—Let us calculate."
>
> — Leibniz, 1684

Meanwhile, logicians from Aristotle to Frege, attempted to formalize the laws of thought.



Frege's 1894 "Begriffsschrift" or *Concept-script.*

In the 20th century, the bridge was finally made between logic and physical systems. The promise of human-run computation was finally achieved. .

Table I. Analogue Between the Calculus of Propositions and the Symbolic Relay Analysis

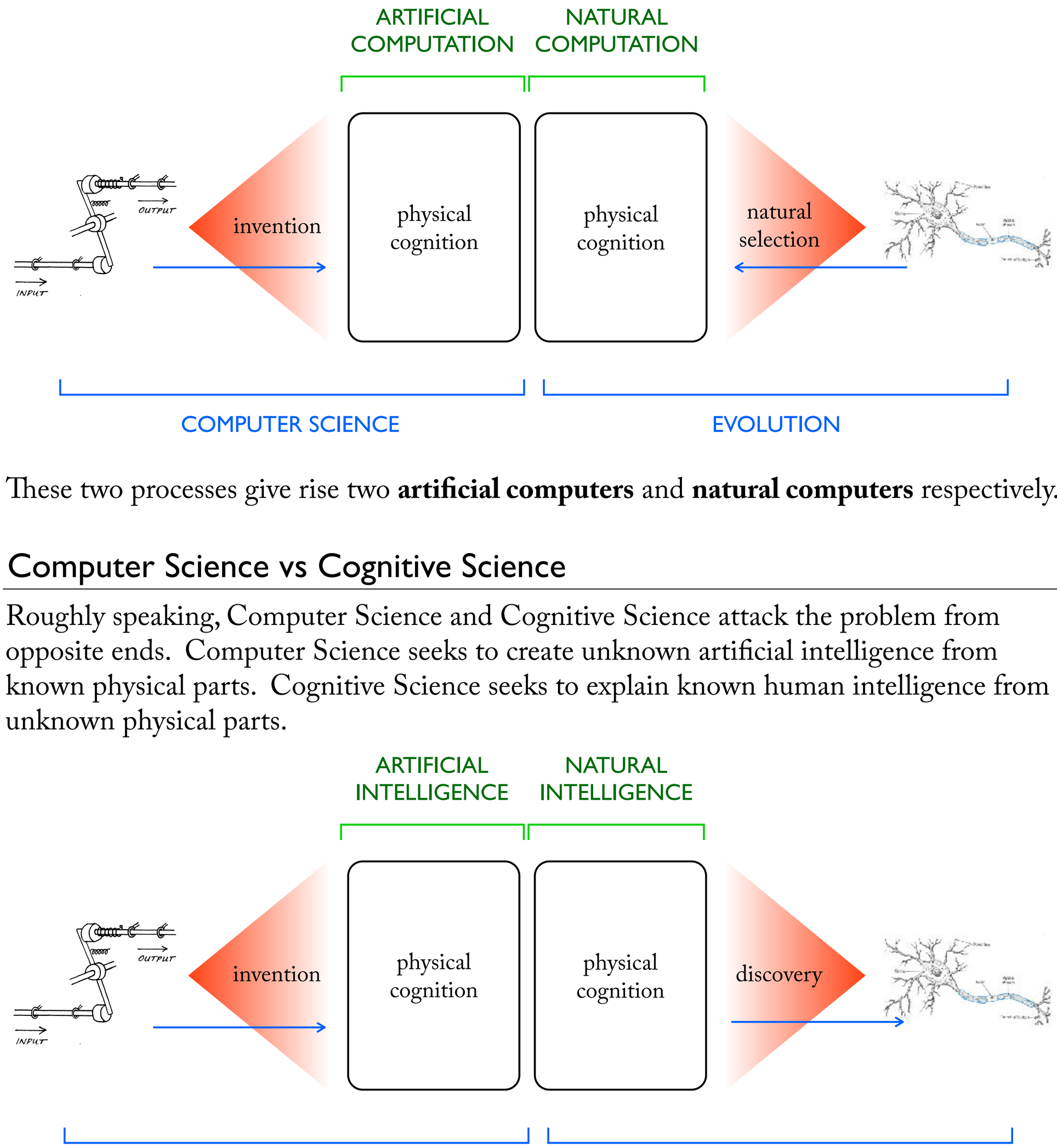
Symbol	Interpretation in Relay Circuits	Interpretation in the Calculus of Propositions
X	The circuit X	The proposition X
0	The circuit is closed	The proposition is false
1	The circuit is open	The proposition is true
$X + Y$	The series connection of circuits X and Y	The proposition which is true if either X or Y is true
$X\ Y$	The parallel connection of circuits X and Y	The proposition which is true if both X and Y are true
X'	The circuit which is open when X is closed and closed when X is open	The contradictory of proposition X
=	The circuits open and close simultaneously	Each proposition implies the other

Shannon 1938 "A symbolic analysis of relay and switching circuits"

> "Logic began as a way to understand the laws of thought. It then helped create machines that could reason according to the rules of deductive logic. Today, deductive and inductive logic are being combined to create machines that both reason and learn. What began, in Boole's words, with an investigation "concerning the nature and constitution of the human mind," could result in the creation of new minds—artificial minds—that might someday match or even exceed our own." Dixon

Computer Science vs Nature

Human Computer Science and Nature represent two processes by which physical cognition has developed.



These two processes give rise two **artificial computers** and **natural computers** respectively.

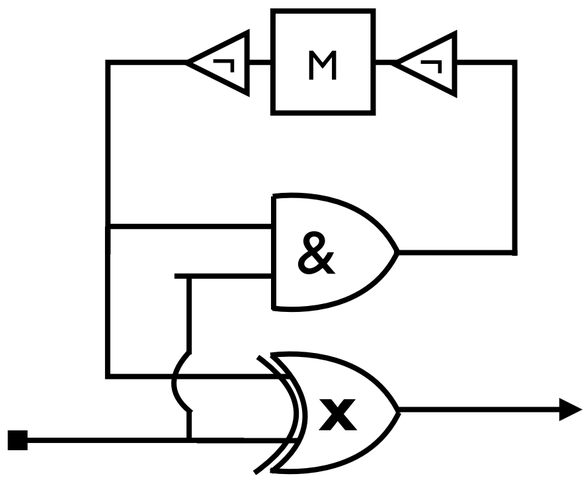
Computer Science vs Cognitive Science

Roughly speaking, Computer Science and Cognitive Science attack the problem from opposite ends. Computer Science seeks to create unknown artificial intelligence from known physical parts. Cognitive Science seeks to explain known human intelligence from unknown physical parts.

"Psychology is engineering in reverse. In forward-engineering, one designs a machine to do something; in reverse-engineering, one figures out what a machine was designed to do." Pinker (1997, 21)

Making Minds

In this class, we are straddling the divide between computer science and cognitive science, artificial and natural. We are using artificial computers as a model for what natural selection could have produced. And we are using the invented machines of computer science as a model for the discovered machines of cognitive science.



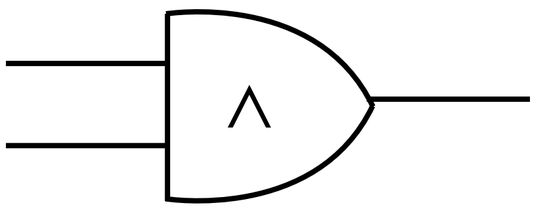
Artificial Computers

Designed. Artificial computers are designed by humans.

Assigned functions. The purpose or function of an artificial computer is what it was assigned by its human designer or user.

User interpretation. The meanings of symbols in an artificial computer are (+/-) determined by the interpretive choices of a human user.

Hardware. Modern artificial computers are built from digital logic gates in an electrical medium.



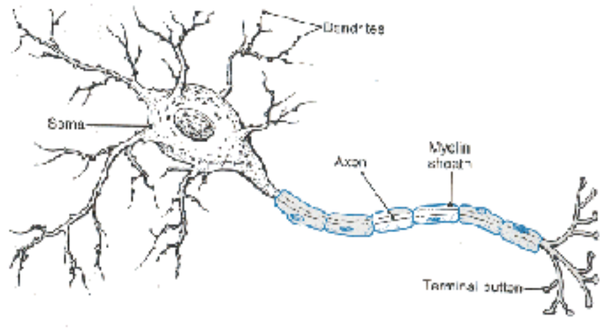
Natural Computers

Evolved. Natural computers are the product of natural selection.

Selected functions. The function of a natural computer is what it was selected to do as a result of an evolutionary process.

Natural representation. The meanings of symbols in a natural computer are determined by the organism's causal interactions with the environment.

Wetware. Natural computers are built from neurons in an electro-chemical medium.



"The mind is a system of organs of computation, designed by natural selection… The mind is what the brain does; specifically, the brain processes information, and thinking is a kind of computation."
Pinker (1997, 21)

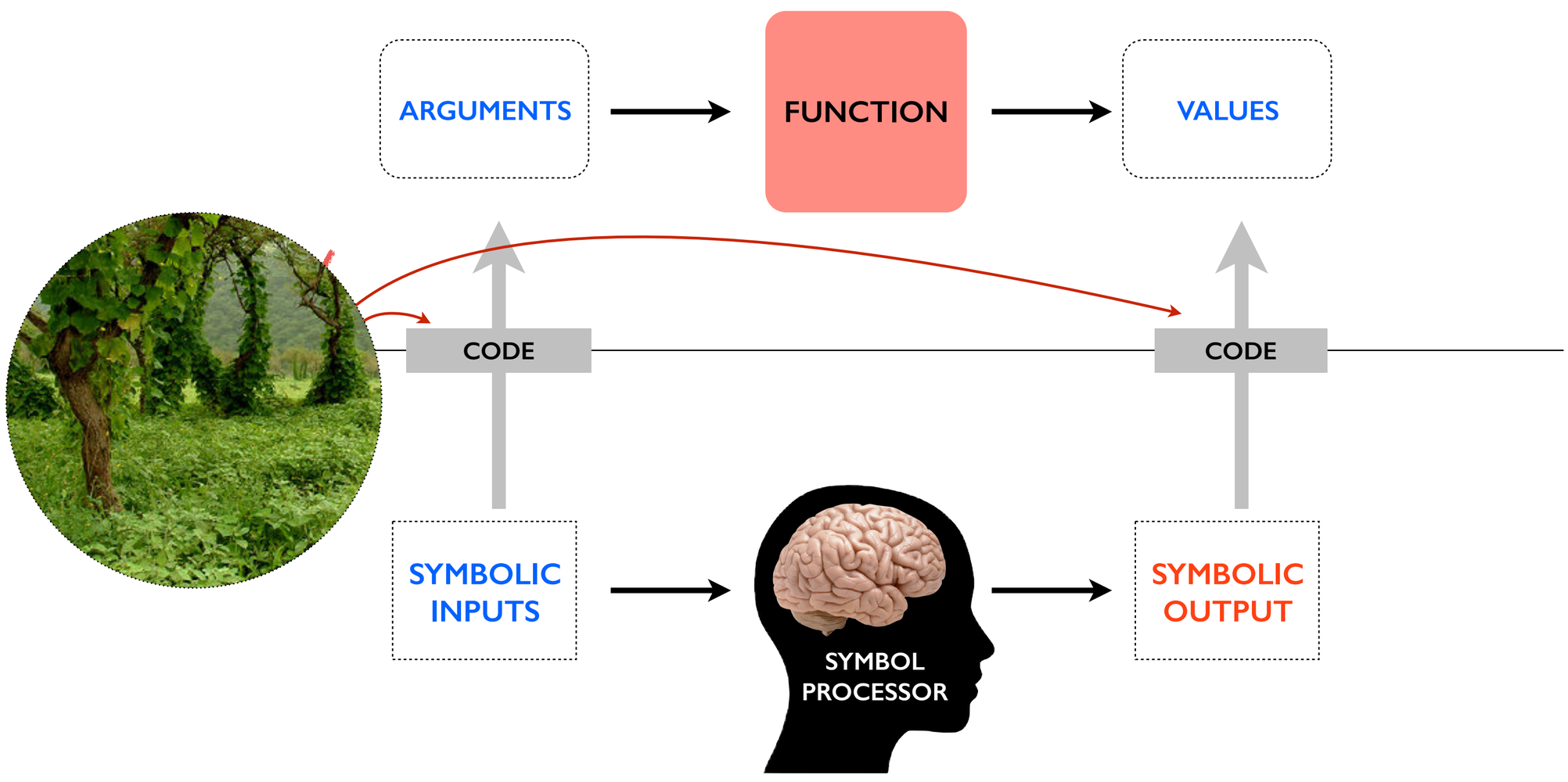
Original vs Derived Intentionality

Original intentionality. Symbols in an artificial computer are given their contents by stipulation or convention. But what gives the contents to the symbols in a natural computer? The computer concept doesn't resolve this at all, but without content there is no representation, and no computation.

> "The question naturally arises of course as to how mental symbols – mental representations – acquire their contents or meanings. The numerals of various numeral systems such as Arabic numerals and Roman numerals derive their meanings from the linguistic conventions of communities. The current along the input and output wires of an electronic computer circuit derive their meanings from our intentional stipulative assignments; as do the keys of hand-calculators and the displays on their screen. If intentionality is to be explained by appeal to mental representations, then we need an account of their meaning that makes no appeal to linguistic conventions or semantic intentions, for these presuppose intentionality… Suffice it to note that the problem of how mental representations acquire their meanings is perhaps the deepest problem that any reductive theory of intentionality faces." (McG)

From plants to informavores: plants have evolved to be responsive to information in the environment, but they do not need to store or process that information. More complex creatures take information in from the environment, processes it, and act on the basis of this processing. For these creatures, information-processing is a biological function.

Externalism: the philosophical idea that the content of your mental states depends in part on the internal state of your brain (what symbols you are processing), but also on your relationship to the external environment (how those symbols carry external information).



15. The Computational Theory of Mind

Start with motivations for CTM- quotes from Haugeland.

Reorient towards: is CTM possible? Focus on problem issues: emotion, consciousness, intelligence.

CTM Defined

CTM v.1: "The central tenet of the theory is that a mind is a computer." (McGlaughlin)

> "Three famous philosophical dilemmas stood in the way [of a materialist theory of mind]: (i) the metaphysical problem of mind interacting with matter; (ii) the theoretical problem of explaining the relevance of meanings, without appealing to a question-begging homunculus; and (iii) the methodological issue over the empirical testability (and, hence, respectability) of "mentalistic" explanations. The computational idea can be seen as slicing through all three dilemmas at a stroke; and this is what gives it, I think, the bulk of its tremendous gut-level appeal. (Haugeland 1981)

Issue 1: is this claim a metaphor, a methodology, or a literal identification?

Comment: In this class, it is meant literally: the mind is a computer just the way the heart is a pump. That is its actual biological function. But this is a philosophical conjecture. It cognitive science, computationalism is treated more like a methodological assumption than a world-view.

Issue 2: how could it be a literal claim, given that our brains are obviously *not* "computers" like laptops or phones. Isn't the target claim clearly metaphorical?

Comment: this is just a verbal confusion. The technical concept of a "computer" is that of a system whose function is to perform cognitive tasks (rational transitions) that relate cognitive contents by mechanically manipulating symbols which function to represent those contents—i.e. an information processor. This definition is neutral about the physical composition of the system or its history. It's in this sense that we mind is literally a computer.

Issue 3: is the claim that the *mind* is a computer, or that the *brain* is a computer?

Comment: Really, this is a claim about the brain, since computers are *embodied* systems. Instead, the mind is better thought of as what the brain *does*. Perhaps we should say, the mind *emerges* from a computer (the brain).

Issue 4: is the idea that the mind is a *computer* or that it's a *computer program*?

Comment: if the brain is a *computer*, perhaps the mind is a *computer program*? This might be half right, in the sense that a computer executes (or performs) a computer program. But as we'll see, the concept of a "program" refers specifically to instructions that are distinct from the hardware they are stored in. But much of the mind emerges from biological hardware. The more general concept is that of an *algorithm*.

Issue 5: is this a claim about all aspects of the mind, including consciousness and emotion?

Comment: While consciousness may be a computational state of the brain, there are few computationalists (even methodological computationalists) who believe that computation can *explain* phenomenology. In the end, computationalism is really about the cognitive mind. Emotion remains an unsettled case.

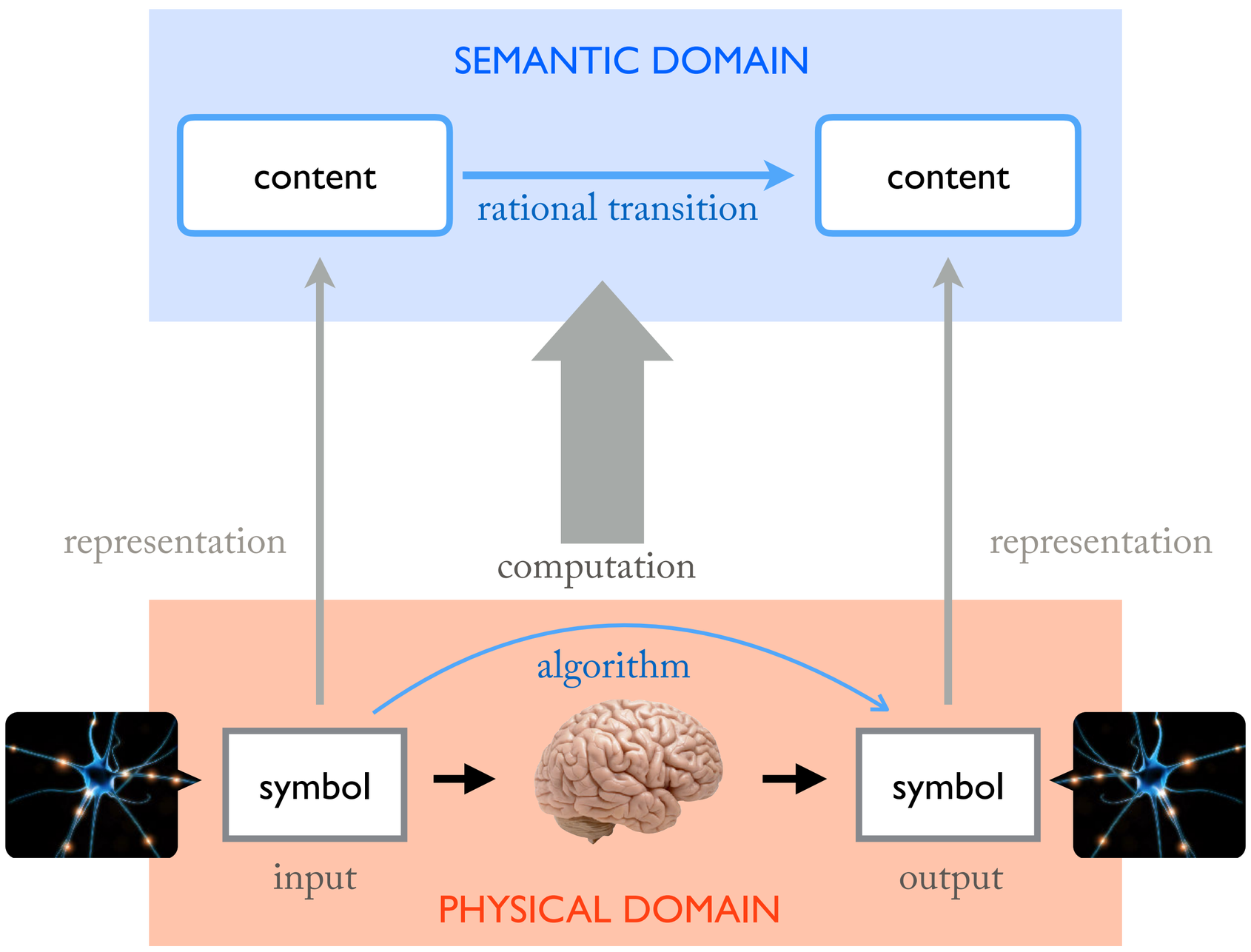
Issue 6: what is meant by "the mind" anyway, given that we have many different kinds of mental states, which constantly changing over time?

Comment: the concept of *mind* is not particularly clear one. It's probably better to talk about mental *capacities*, mental *states*, and mental *processes*. (The mind, roughly, is the unity of these elements.) The idea is that mental states are computational states of the brain, and mental processes are computational processes of the brain.

Issue 7: how could the brain be a computer, since no one designed a brain, or assigned it an interpretive code?

Comment: the brain wasn't designed, but it does have an overall *biological* function, and its parts have many specific biological functions. States of the brain function to carry information about the world (to represent); and structures in the brain function to process these states (or, compute).

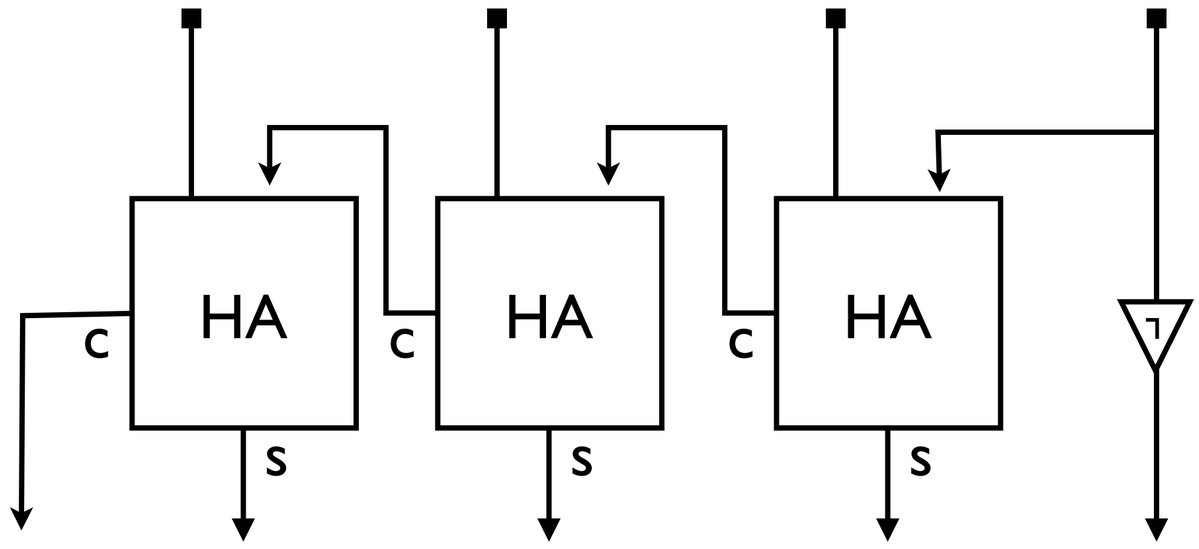
Computationalism v. 2: the brain is a natural computer, or information processor. The biological function of the brain is to perform a range of cognitive functions by carrying out mechanical transformations on physical symbols which compute those functions. Cognitive states are computational states of the brain. Cognitive processes are computational processes of the brain.



Computational Theory of Mind: the brain computes cognitive functions by manipulating symbolic representations in the brain according to biologically grounded algorithms.

16. Lessons and Limits of Combinatorial Computation

5 Fundamental Lessons of CC computation



+1[0-15] B combinatorial circuit

Lesson 1: The physical realization of a cognition.
+1B shows us definitively that an entirely physical system can performs cognitive operations, via the method of symbolic simulation.

Lesson 2: Information processing as functional description.
Hammers have the function of pounding nails, and hearts have the function of pumping blood. +1B has an *information processing* function, which goes beyond the way it manipulates the physical world. "It computes +1" is the informational-level description of the machine. Compare +1B and +1T: different physical functions, same information processing function.

Lesson 3: Computing a function vs implementing a function.
All things *implement* the laws of physics, but only some things *compute* the laws of physics. To compute is to *represent* a function, but not to actually conform with it. +1B can compute the laws of *adding-apples-to-a-basket*, but it does not *implement* these laws. (By the way, do the planets compute their own orbit? When humans walk, do they compute or implement their trajectory?)

Lesson 4: Computation does not require an explicitly represented algorithm.
+1B computes a function not by "following" a set of rules, which would require an inner-agent to do the following, but simply by *embodying* a set of rules in its functional architecture. To compute a function is to

Lesson 5: Competence vs performance.
+1B shows how a physical system can have certain cognitive *capacities*— i.e. the +1B machine has the capacity to compute +1, for a range of arguments, even if it is never actually used for this. It also illustrates the contrast between the capacities you have *competence* for, and your actual performance. Because of entropy, friction, and the rest, how a machine actually performs can pull apart from its competence.

Computation beyond Combinatorial Circuits

1. **Multi-functionality.** Natural cognitive systems compute many different functions. Our +1B machine computes only one.

2. **Learning.** Natural cognitive systems change their behavior in response to experience. The behavior of our +1B machine is fixed for eternity.

3. **Non-arithmetical functions.** Natural cognitive systems perform non-arithmetical functions, including decision making, action-planning, prediction, navigation. What can a CC do beyond +1B?

4. **Logical and probabilistic inference.** Among non-arithmetical functions some employ language-like reasoning, especially logic and inference. +1B is stuck in a world of integers.

5. **Perception and action.** Natural cognitive systems exist within an environment. They gather information from that environment (perception), and they change that environment (action). +1B has no obvious causal connection to the natural order.

6. **Biological basis.** Brains are built from neurons, not logic gates; neurons are put together to form neural nets, not combinatorial circuits. So +1B fails to capture the physical and functional basis of natural computation.

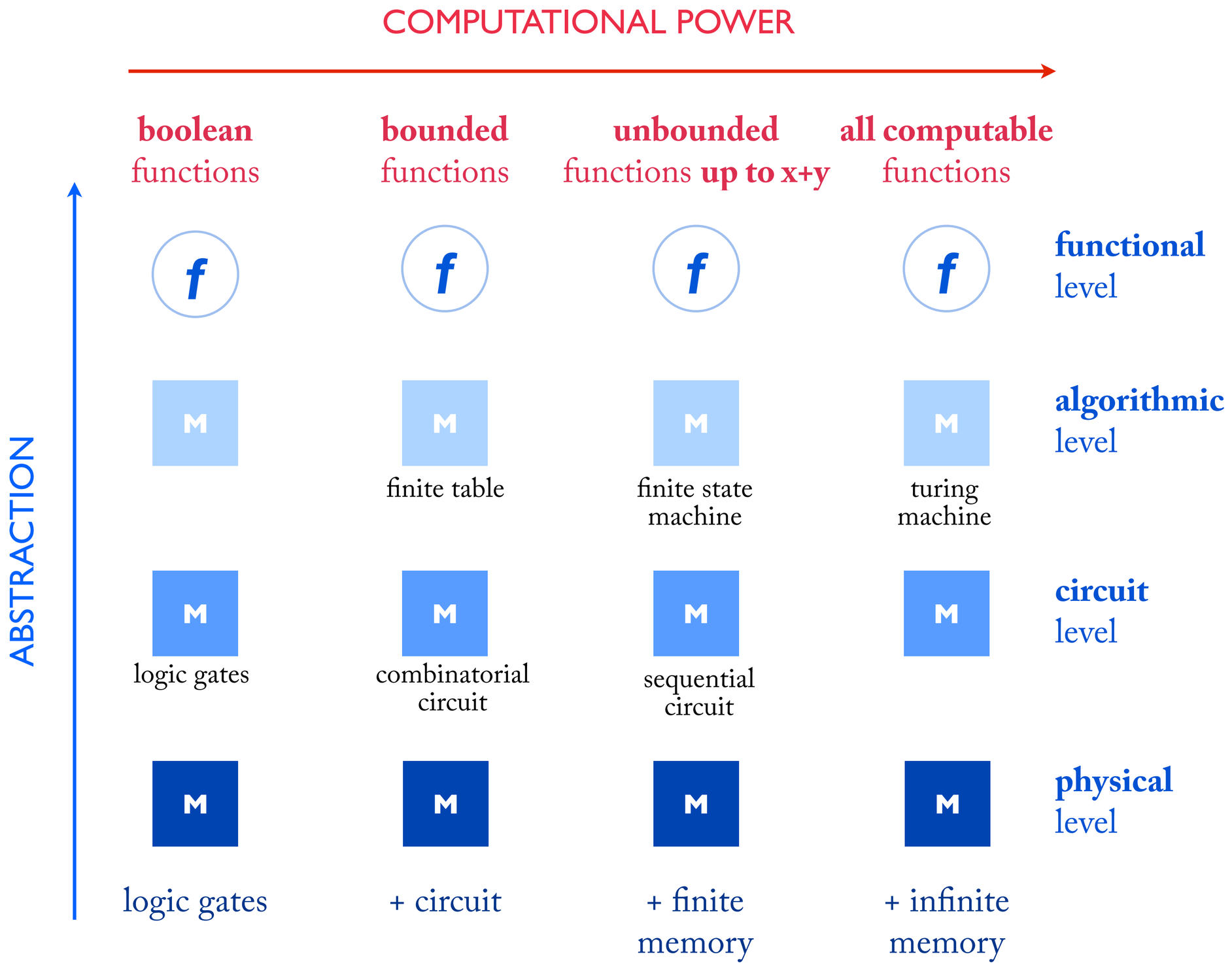
7. **Emotion and consciousness.** +1B is incapable of emotion or consciousness. (Right?) It is not clear, from the current standpoint, what would even be required to achieve this.

8. **Original functionality.** Natural cognitive systems have the biological function of performing certain information-processing tasks, as a result of natural selection. The +1B machine's only function was stipulated in a problem set.

Unit 2

LEVELS

In Unit 2, we explore the dimensions of computation. We'll look at levels of abstraction and levels of computational power.



17. Memory

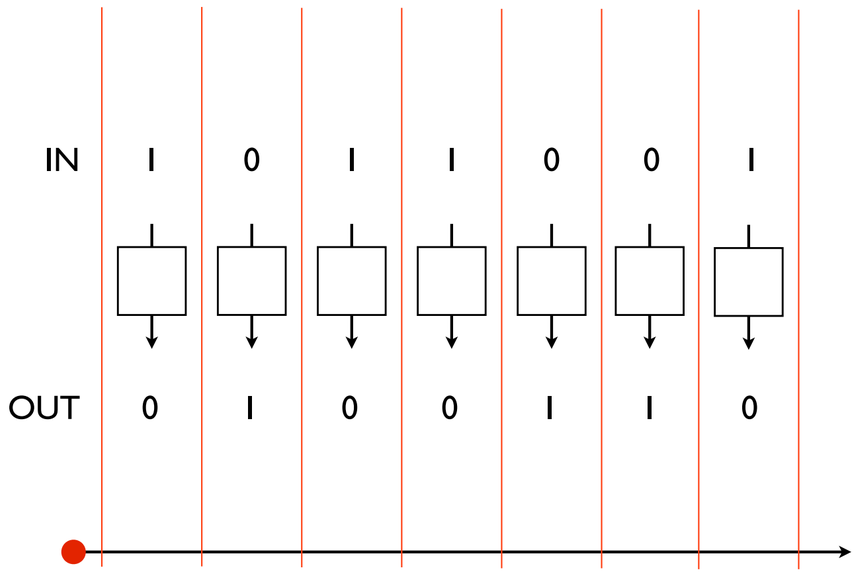
To do: add notes on STM deficits, HM, and the concept of "state memory" vs. "stored memory".

Harnessing the Power of Time

At the end of Unit 1, we saw some of the limitations of combinatorial circuits. For one thing, the size of the inputs they can process is limited by the size of the circuits. In a small and finite amount of space, only small and finite numerals can be processed. For another, combinatorial circuits always respond in the same way to the same input, no matter what has happened in the past. They are incapable of learning.

To overcome both problems, we must find a way to harness the power of time. If we can look back in time, we can condition our behavior on what has happened in the past. And we can access the unlimited store of time available to us for the purposes of computation. Memory, of course, is what will allow our little computers to master time.

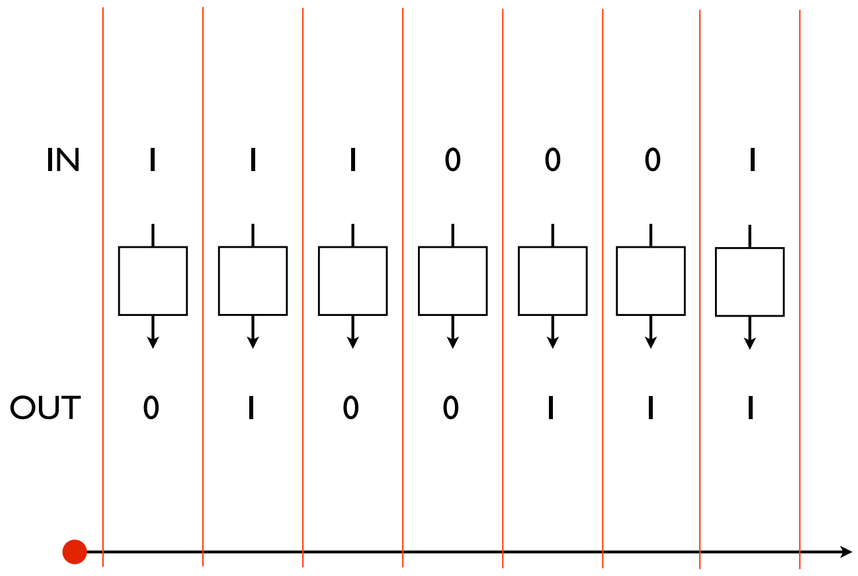
History Independent Response



This machines always responds to the same input in the same way, independent of its past responses.

No memory required.

History Dependent Response



This machines responds to the same input in different ways, depending on its past responses.

Memory required.

Memory

Memory is **information carried forward in time**. The chalkboard is a form of memory; marks are carried forward in time until they are erased. A seesaw is a primitive form of memory too; it stores information about which end was pushed down most recently.

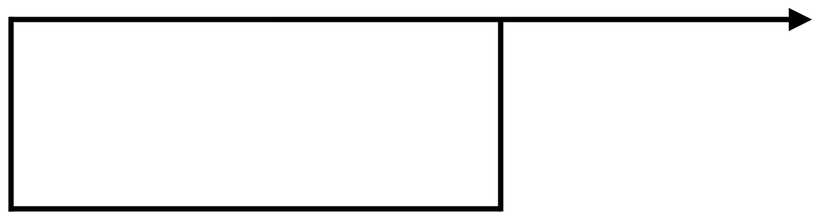
18. Physical Memory

Constructing a Memory Register

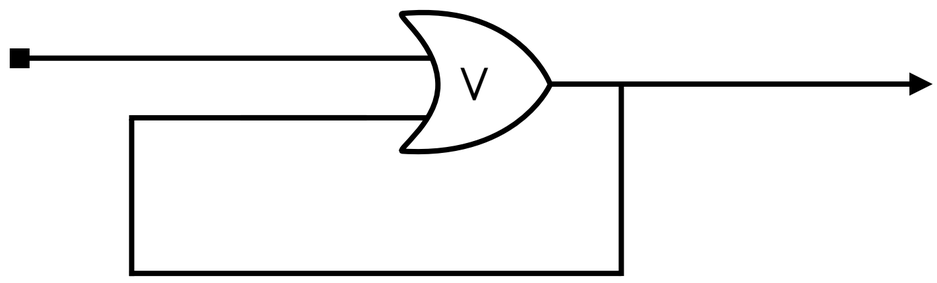
We can easily carry information forward through time in a stable medium like chalk on a chalkboard. But how could we store information in a dynamic medium like the flow of electric current, or electro-chemical interactions of neurons? How can we get the information stored in an electric current to "sit still"? We'll now explore how memory can be implemented in an electronic circuit. The details here won't matter much for the overall narrative, but if we are out to demonstrate the possibility of physical cognition, we better prove to ourselves first that memory itself can can physically implemented.

Warning! In this section I'm going to use the familiar-looking circuit diagrams to represent **electronic circuits**, and not **logic circuits**. This means that 1's and 0's represent states of an electric current (**on** and **off**), and lines represent **wires**, not just information pathways. In an electronic circuit, the resting state for all wires is off, so the default input signal to any logic gate is 0. Also, in electronic circuit, you can create loops, though their behavior can be unpredictable! I've abstracted away from voltage source and ground.

The core idea is that information could be stored by locking an electrical current into a **loop**, which would sustain it over time. The circuit below embodies this idea. The signal stored in the loop is being repeatedly delivered to the output wire at right.

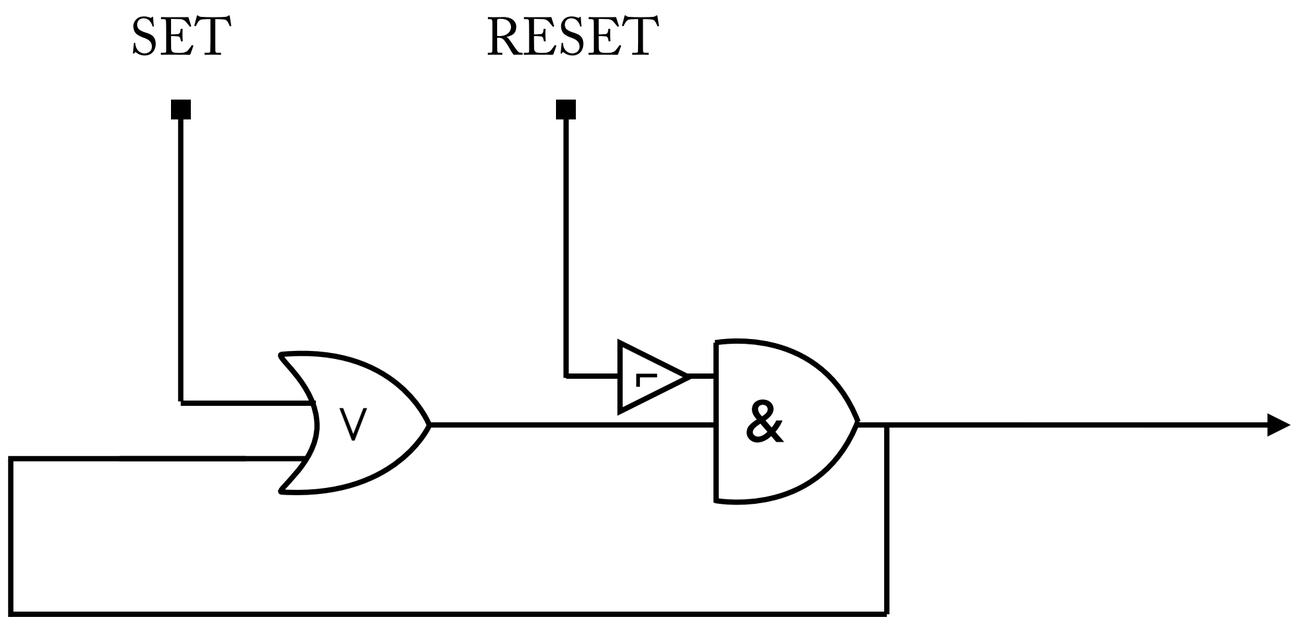


But how do we get information into the loop in the first place? A circuit like the one below does the trick. As long as the input signal is 0, the output is 0. But as soon as the circuit receives a single 1, it will output 1's forever. This is a form of memory— it carries information forward in time about whether a 1 has ever been inputted.



But it is not very useful. It's like a seesaw that always gets stuck. We want a memory that can store whatever information we like, the way a blackboard allows us to do! What we need is a way of selectively breaking the loop, and resetting the circuit at will. We want something that, at one time, can take a bit and keep reproducing it (like the circuit above), but at another time, can also start afresh if its convenient.

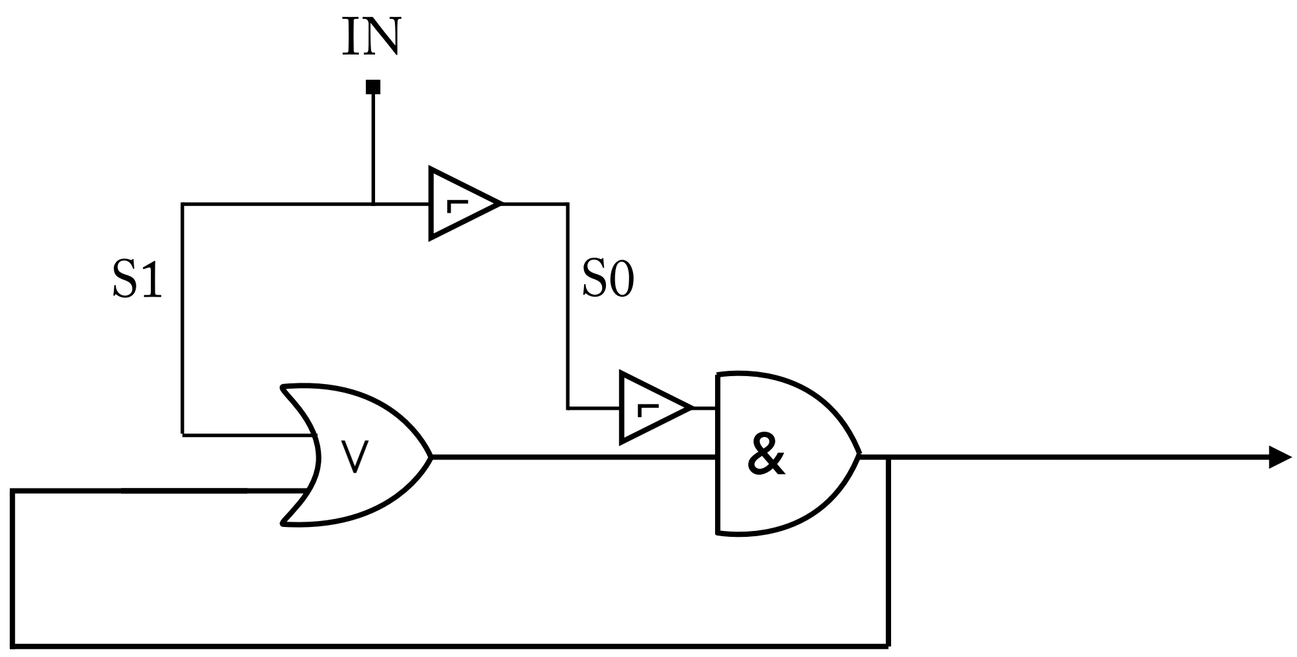
Here is an idea: introduce an AND-gate after the OR-gate. That way we can block the loop when we want to. For conceptual clarity, we'll add a NOT-gate just before the AND-gate.



To think about this circuit, let's say that, by default, RESET=0 and SET=0. In the initial "resting state", the circuit as a whole will store a 0. If at some point SET=1 for a moment, then the circuit will switch over into an "active state", in which case it stores a 1, even when SET=0 again. When you are ready to return to the resting set, let RESET=1 for a moment.

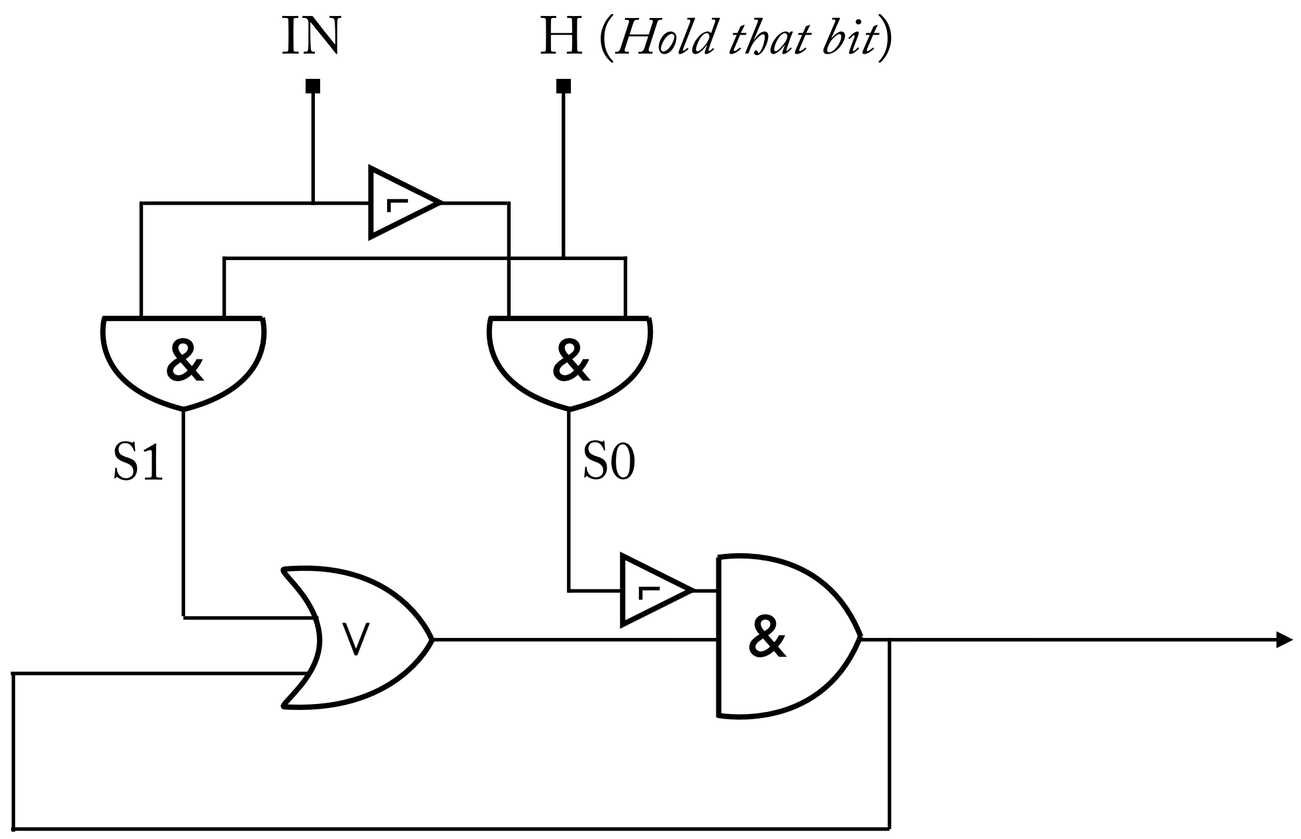
In a way, the resting state stores a 0, and the active state stores a 1. The glimmer of real memory! Can we use the two different states to store an arbitrary bit? Suppose we had 1's trigger the active state, and 0's trigger the resting state?

The natural next step is to hookup an input wire to the S1/S0 wires so that 1's activate the S1 side (and not the S0 side) and 0's activate the S0 side (and not the S1 side).



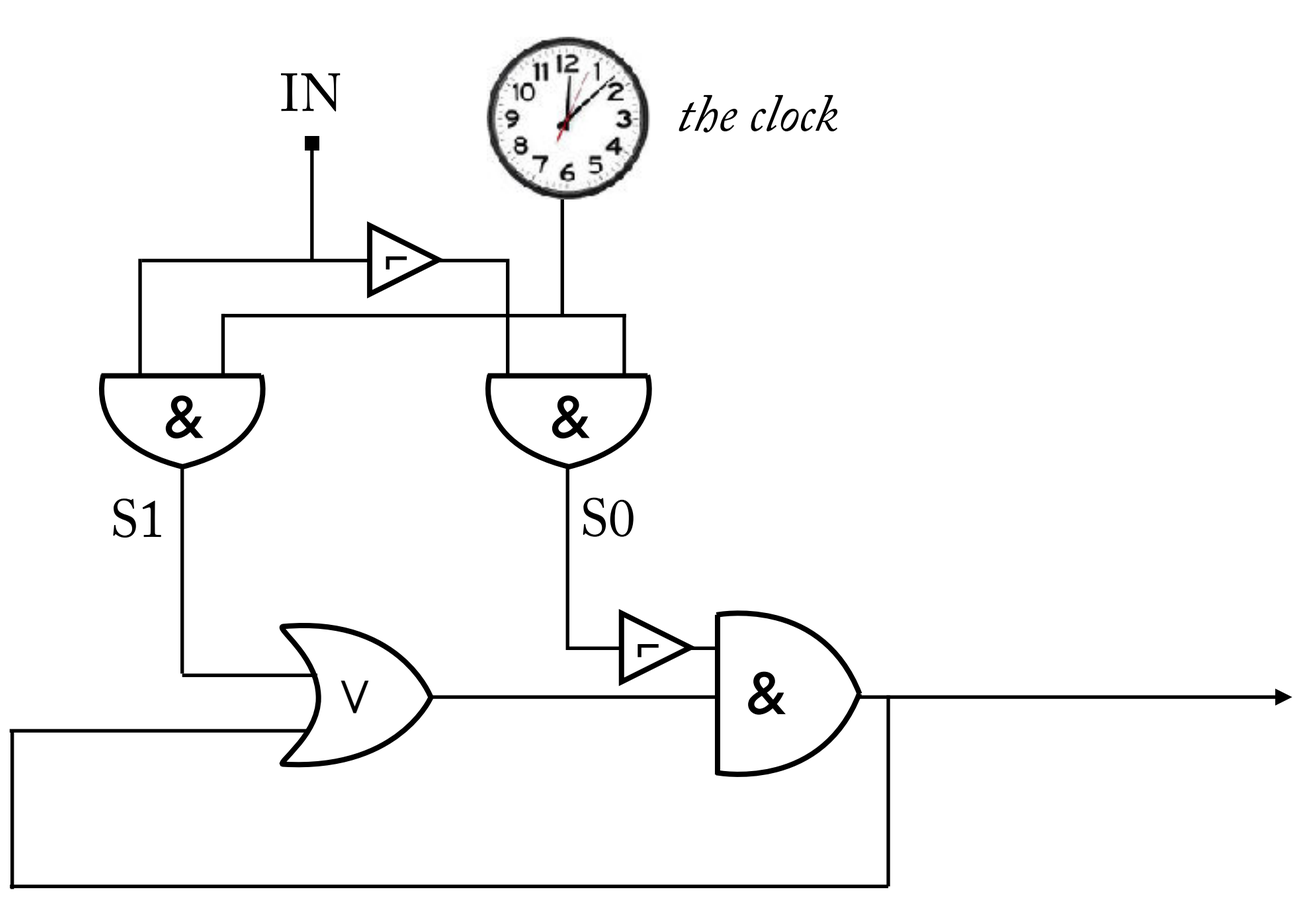
This is a good start, but our resulting circuit doesn't actually remember anything. The problem is that there is always *something* coming along as input (even if it's just the OFF signal).

So we are going introduce a control mechanism which we'll call "Hold that Bit" (H). The idea is that you activate H when you want to capture and store the current input bit, but if you are happy with the bit you've got, then leave H off. This circuit is known as a **flip-flop.**

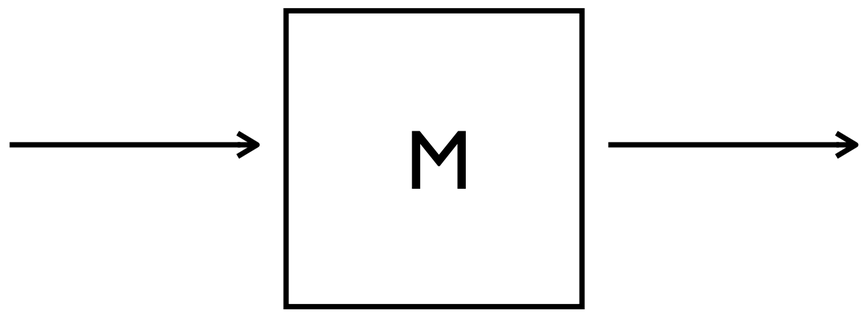


Our memory is like a box. Sending a 1 through H is equivalent to opening the box; whatever the current input bit is, that bit is stored in the box. Sending a 0 to H is equivalent to closing the box; the current input bit is now irrelevant, only what's in the box matters.

There's one last step. After all, when does H get activated? Does the user somehow have to turn on each memory when they want to use it? No. For this we'll use **a clock**. A clock in this sense doesn't *store* the time, instead it marks out **ticks** at regular intervals, like a metronome. We'll connect the clock up directly to H. The idea is that, at each tick, H is activated, and the loop is either set to 1 or set to 0; after each tick, H is deactivated and the loop continues to store and output whatever bit it was last set to.



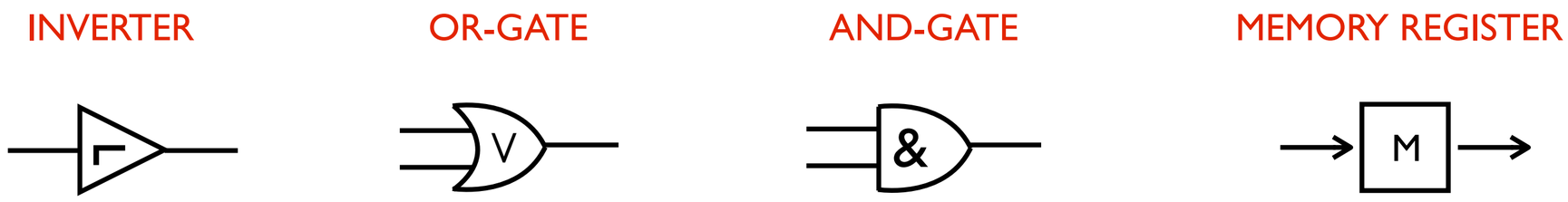
The result is what we'll call a **memory register**: a type of self-contained circuits that can store a single bit. In the future, we'll want to abstract away from all this physical circuitry, in order to focus on the logic. So I'll think of each memory as 1-input 1-output box which outputs a bit at the beginning of each clock cycle and inputs a new bit at the end of each clock cycle. We'll hide the input from the clock, with the assumption that the clock is hooked up in the same way to every memory register.



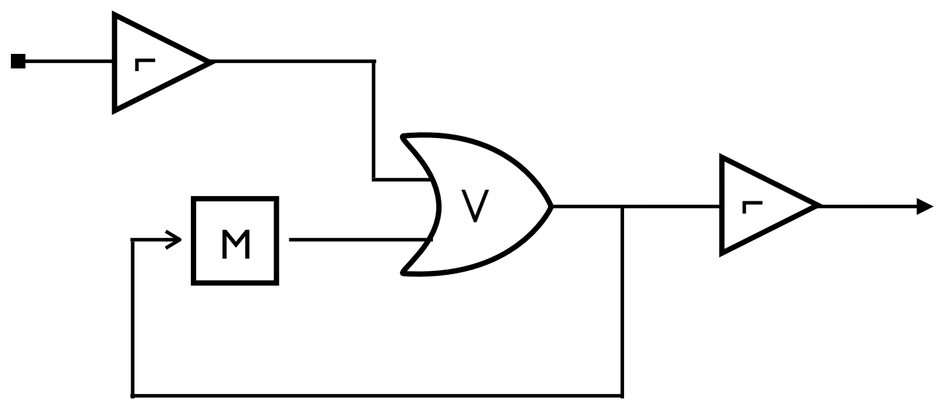
19. Machines in Time: Sequential Circuits

Sequential Circuits = Combinatorial Circuits + Memory

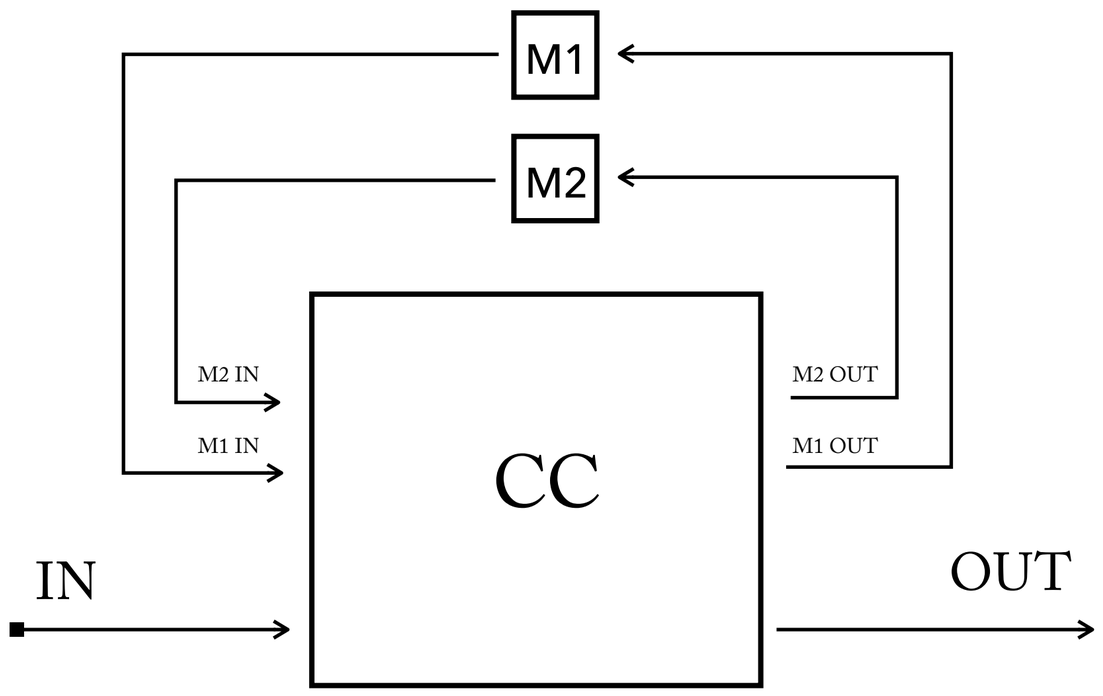
A **sequential circuit** (SC) is a combinatorial circuit combined with a finite number of memory registers. Our circuits now consists of four basic parts.



The structure of a sequential circuit is just like a combinatorial circuit, except that **loops are allowed**, but **only when they are interrupted by a memory register.** Like this:

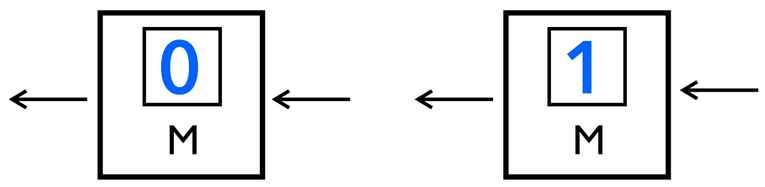


In a **standard layout,** the memory registers are kept separate from the combinatorial circuit. The inputs to the circuit as a whole are aligned with inputs to the CC from memory; the outputs of the circuit as a whole are aligned with the output from the CC to memory. This design that makes the conceptual relationship between CC and memory visually clear. (You don't have to use standard layout when designing circuits.)

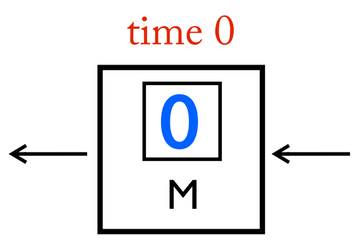


Memory Registers

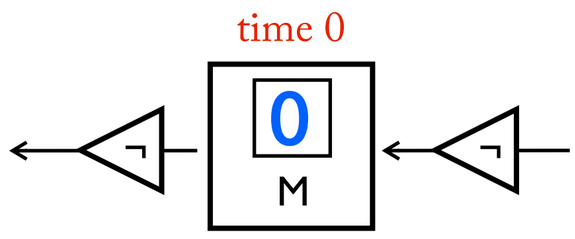
Each memory register can store exactly one 1-bit.



We'll assume our memory registers are **0-first**:
at time 0 they always begins by storing a 0.



But you can build a **functional 1-first** memory with a circuit that uses a 0-fist memory, but simulates the behavior of 1-first memory. It outputs a 1 on the first cycle



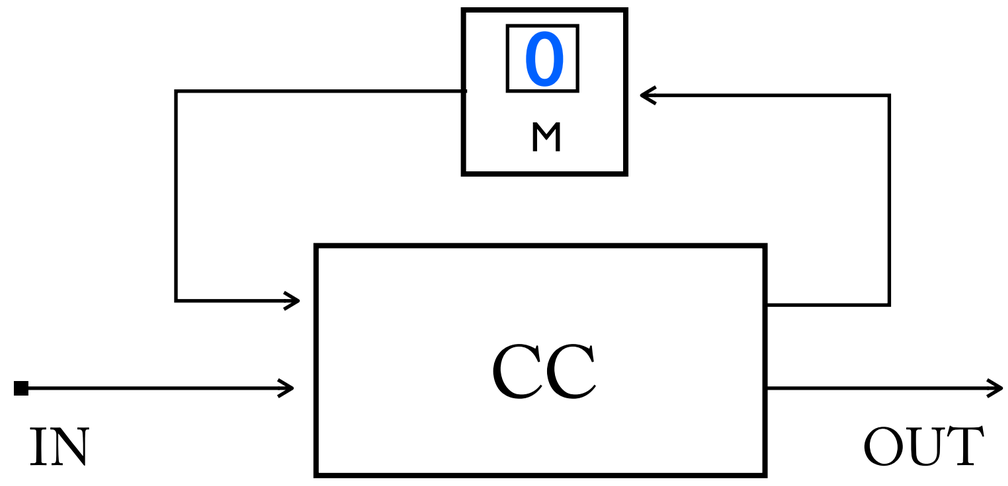
The Clock

Machine time is governed by an abstract clock, which goes through discreet cycles. At the beginning of each cycle, inputs are read, and output are written or displayed. As before, within a cycle, the machine works instantaneously.

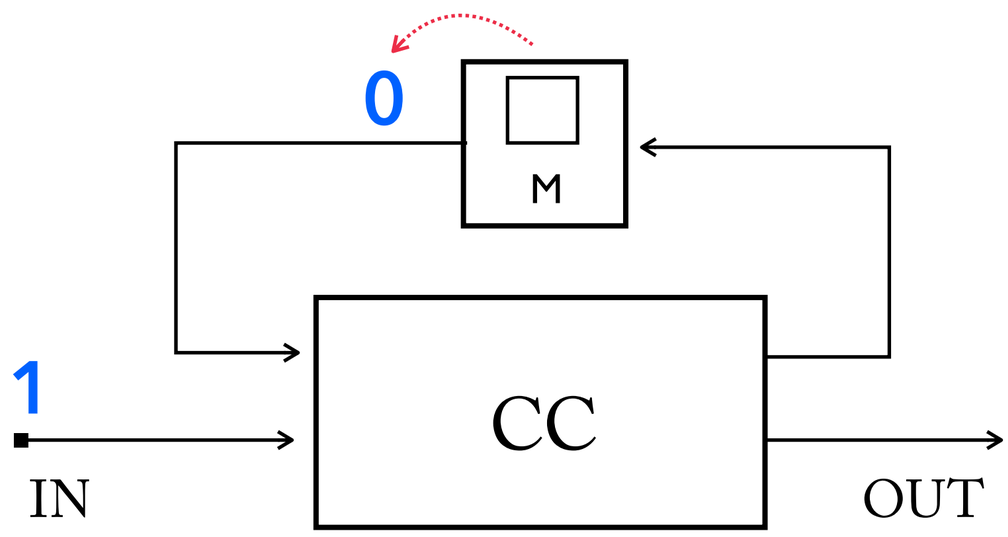
What makes memory registers *memory*— in other words, what makes them carry information forward through time— is that they preserve the bit they carry after each cycle of the clock.

Cycles of a Sequential Circuit

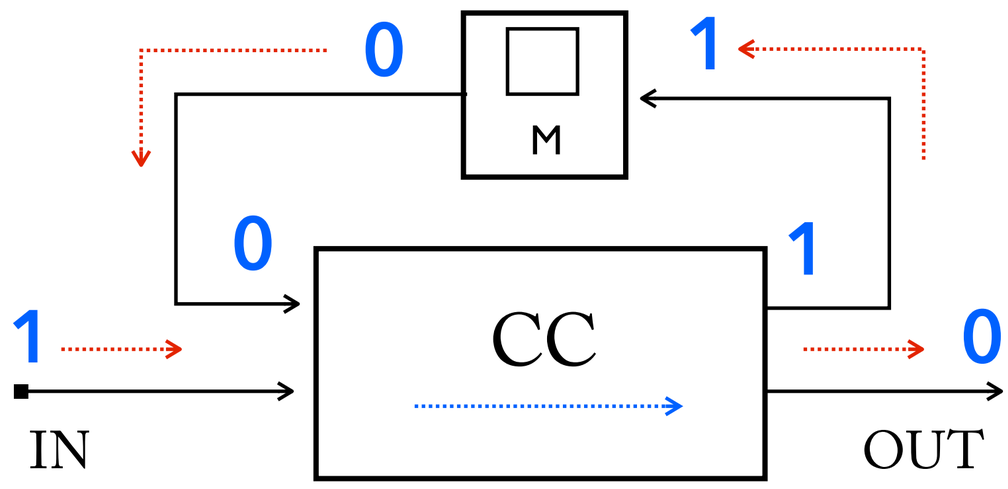
1. In the beginning, the circuit is quiet and a bit is stored in each memory register.



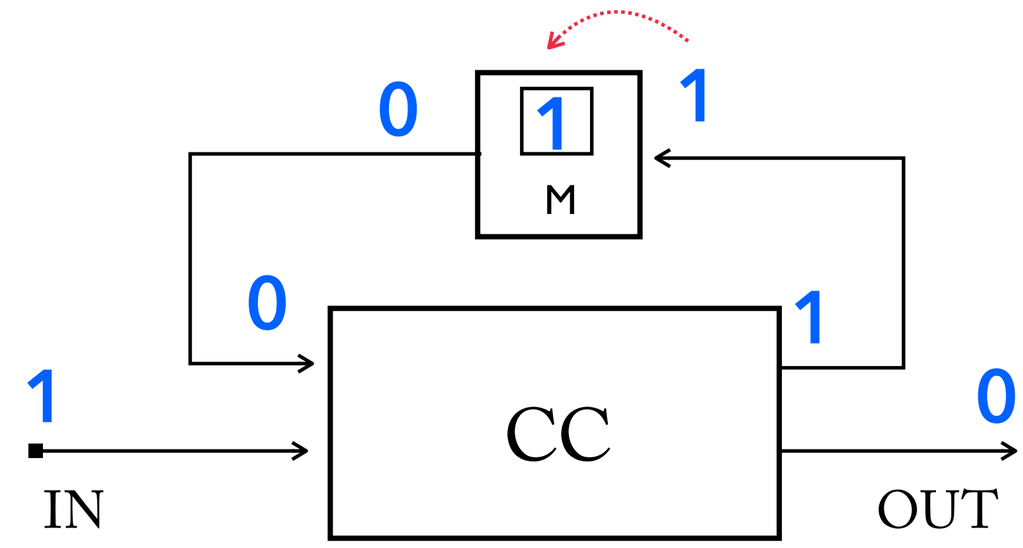
2. Input is received from each local input and from each memory register.



3. Run the circuit, produce outputs.

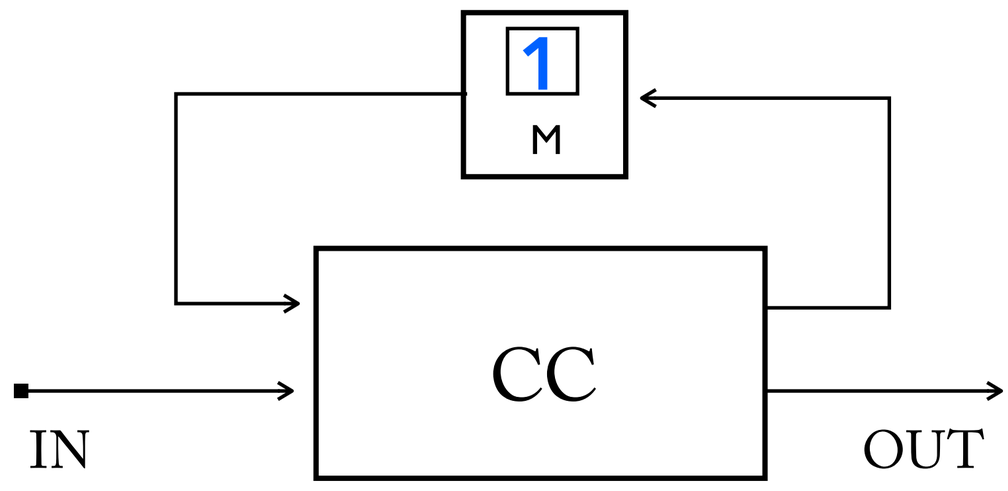


4. Record new bits into memory registers.



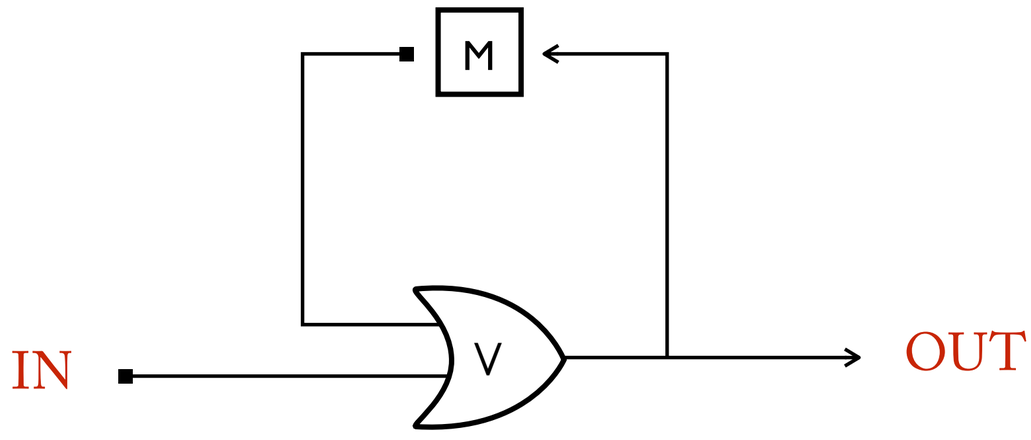
5. Clear the circuit of all information except for the memory circuits.

Initiate the next cycle.



Memoirs of a Sequential Circuit

To keep track of the behavior of a sequential circuit, we'll typically have to keep *two* kinds of tables: one **global input-output table**, and a set of **sequential local input-output tables.**



Global input-output table

IN	OUT
000	000
001	111
010	
100	100
100	
101	111
…	…

Sequential local input-output tables

…	t5	t4	t3	t2	t1	
…	0	0	0	0	0	IN
…	0	0	0	0	0	OUT

…	t5	t4	t3	t2	t1	
…	0	0	0	0	1	IN
…	1	1	1	1	1	OUT

…	t5	t4	t3	t2	t1	
…	0	0	0	1	1	IN
…	1	1	1	1	1	OUT

…	t5	t4	t3	t2	t1	
…	0	0	1	0	1	IN
…	1	1	1	1	1	OUT

What does this circuit *do*? At which point in the sequence does this machine exhibit distinctively sequential history-dependent behavior?

In-Class Problems

Design SC's which solve the following problems.

1. **Temporal AND:**
   Output = 1 iff the current input = 1 **and** the last input = 1.

2. **Temporal OR:**
   Output = 1 iff the current input =1 **or** the last input = 1.

3. **Trip-wire**
   Output = 0 iff there is or ever was an input = 0.

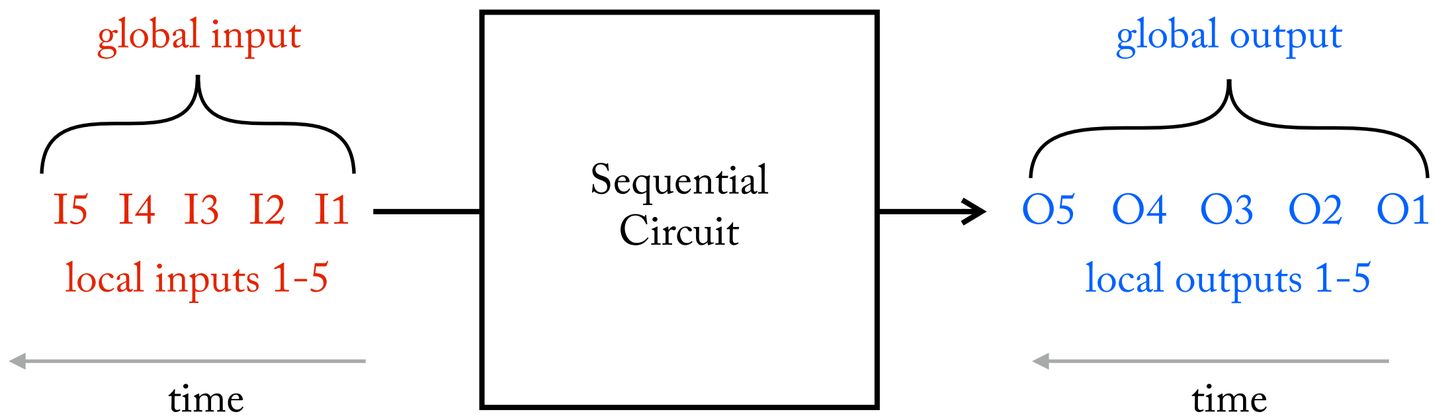
4. **Pattern Finder:**
   Output = 1 as long as the input alternates 1's and 0's (e.g. "101010…"). If the pattern is ever broken, output = 0 forever after.

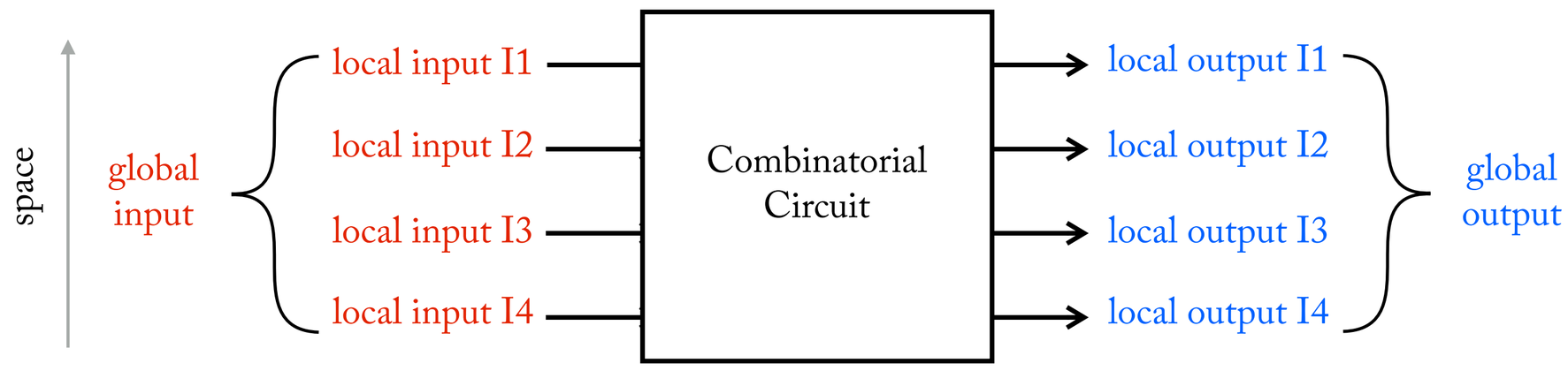
5. **ON/OFF switch:**
   Starts in the OFF state. If you push a button, it goes ON. If you push the button again, it goes OFF. (**Hint:** think in tables.)

20. Computing in Time

Inputs and Outputs

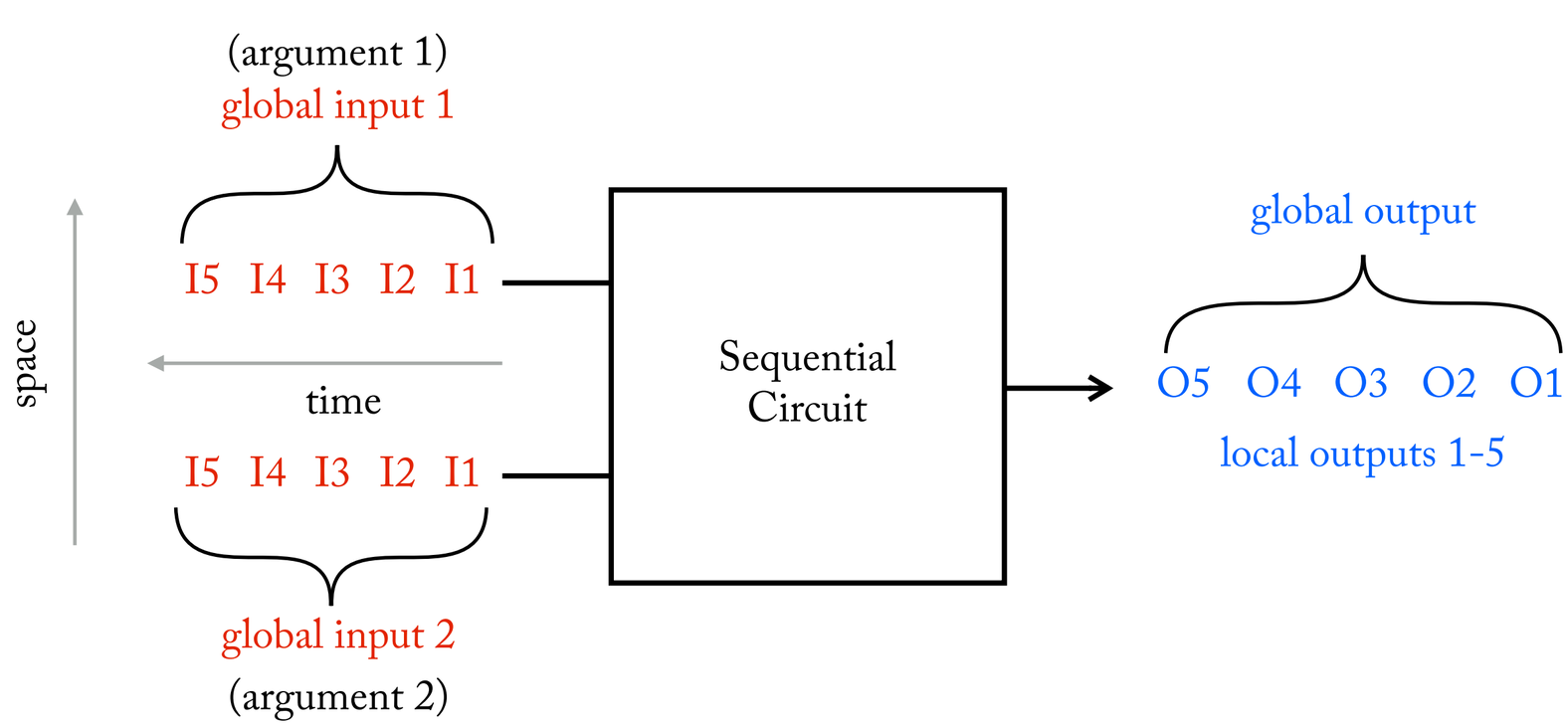
We now introduce the concept of **input at a time**, as well as the overall **input over time**. The input at a time corresponds to an SC's **local input**, the input over time corresponds to an SC's **global input**. So, for SC's, arguments and values are represented by complex symbols over time.



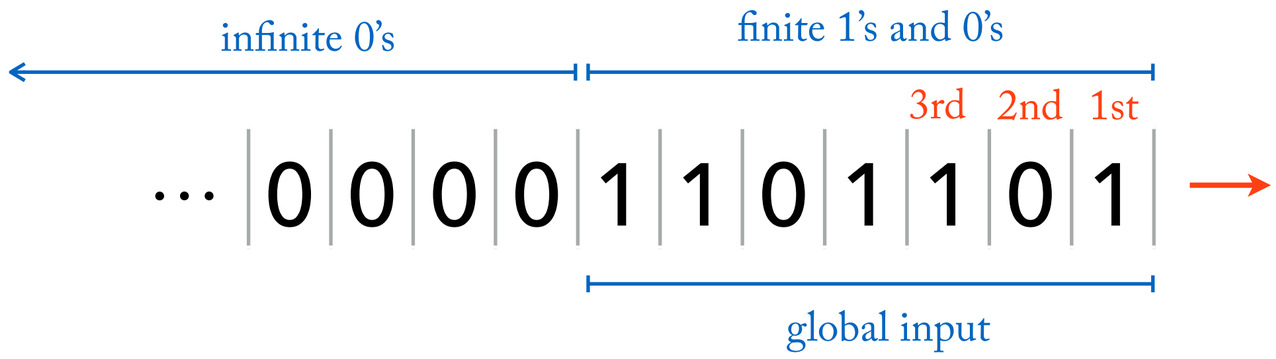


SC's do with time what CC's do with space. The major difference is that space is finite, but time is (for all practical purposes) unlimited.

By the way: SC's can have more than one global input. When performing computation, each argument-place in the computed function corresponds to a distinct global input in the circuit.

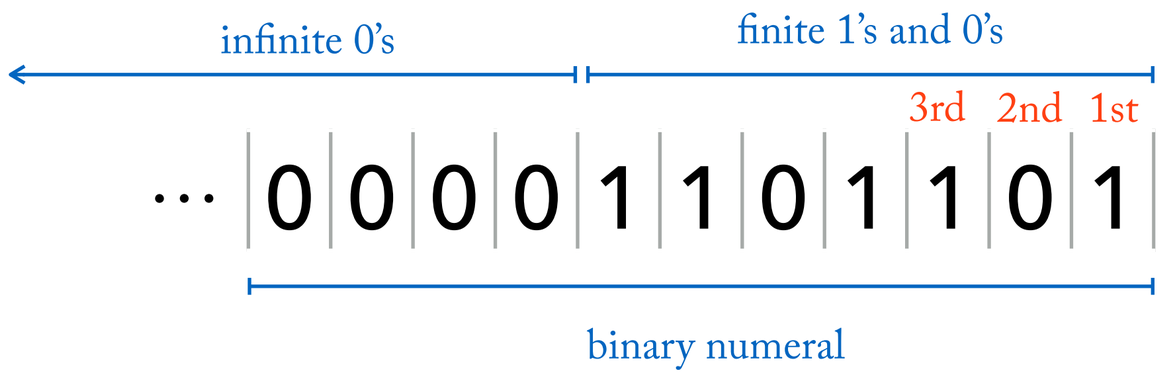


Computing with Sequential Circuits



On any one occasion, we'll imagine that an SC goes on forever. It receives an infinite number of digits over an infinite amount of time. We'll assume that the input always begins with a finite number of 1's and 0's followed by an infinite number of 0's.

When an SC computes a numerical function, the global inputs and outputs are those finite sequences of local inputs/outputs at the point when all remaining local inputs *and* outputs are 0's.



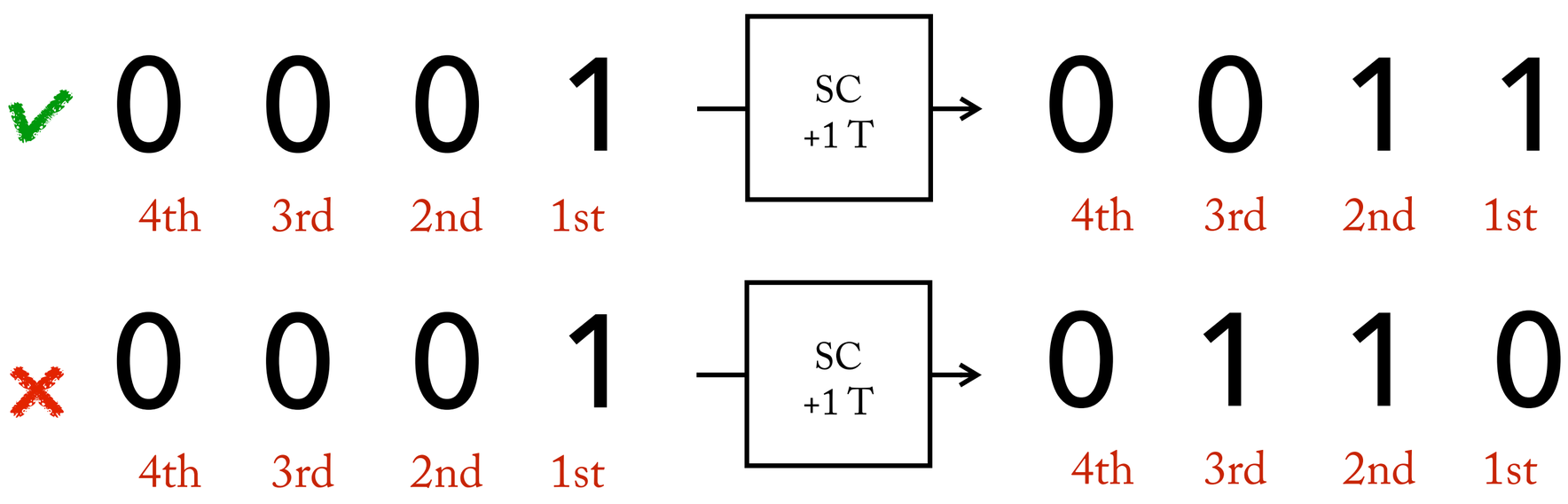
Numerals in Tally:

- Entered as a finite number of consecutive 1's followed by an infinite number of 0's. E.g. 00001, 00011, 00111, 01111, etc.
- 0 will count as a numeral in Tally (representing zero).

Numerals in Binary:

- Made up of a finite number of 0's and 1's followed by an infinite number of 0's.

Computing a function with SC's is just like computing a function with CC's. Only now, the inputs arrive in temporal order, rather than spatial order.



+1 T: One more time with feeling….

Remember our temporal input-output table for +1 T. It's a temporal version of the +1 T CC. It always produces a 1 first and then delays every other bit by 1 cycle.

+1 T where x=0

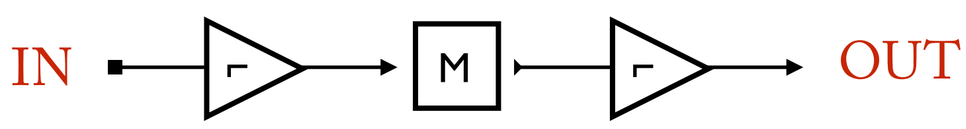
t5	t4	t3	t2	t1	
…	0	0	0	0	IN
…	0	0	0	1	OUT

+1 T where x=2

t5	t4	t3	t2	t1	
…	0	0	1	1	IN
…	0	1	1	1	OUT

We can always delay a bit by 1 cycle by introducing a memory in the middle of a wire.

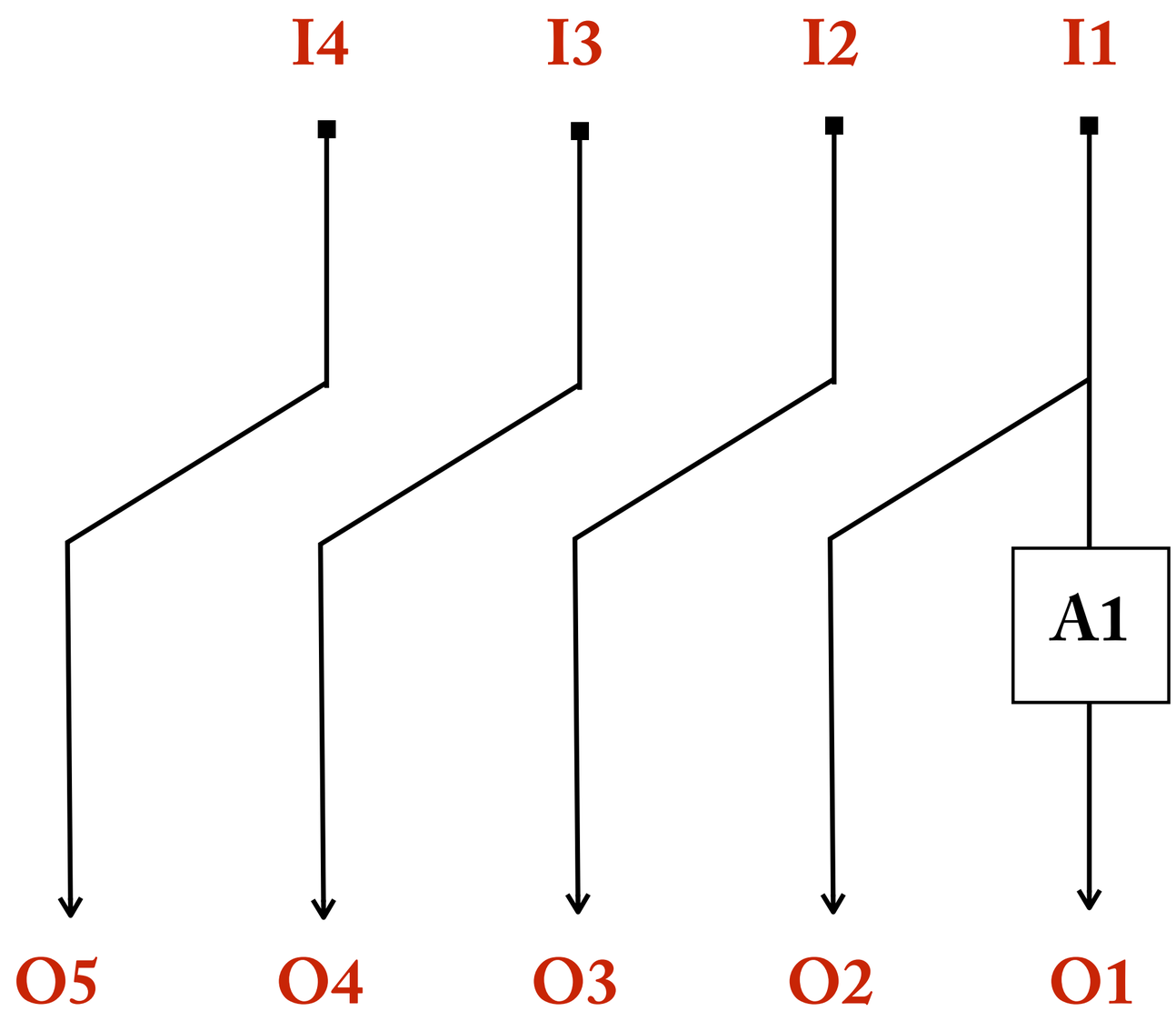
Of course for +1 T, we need the first bit to be a 1. So then it would make sense to use a **1-first memory**, and use *that* to delay the incoming bit.



Voila! +1 T SC

If you squint in just the right way, you should be able to see that our solution for SC +1 T is the same as our solution to CC +1 T.

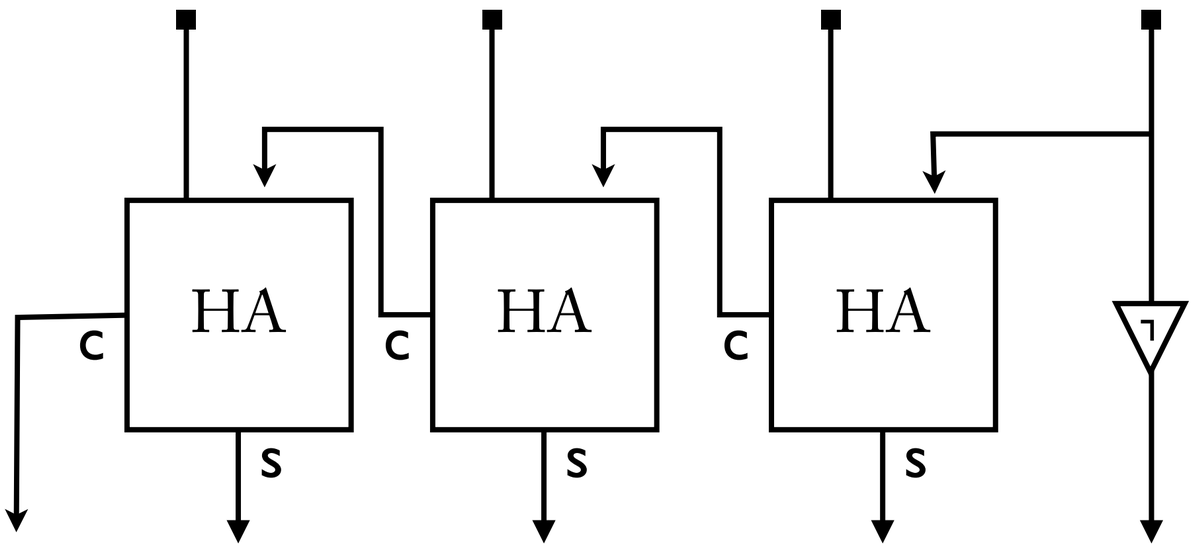
The CC just pushed each bit one position over in *space*. The SC pushes each bit one position over in *time*.



Binary +1

How do you compute Binary +1? Intuitively, by computing +1 on the first digit, outputting the sum, then adding the carry to the second digit, outputting the sum, then adding the carry to the third digit, outputting the sum, and so on…

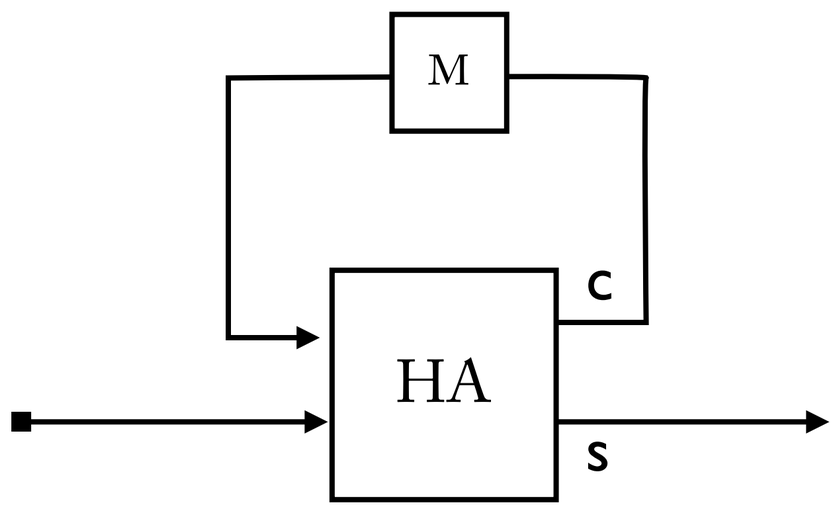
We've already developed a sub circuit for adding a carry to an input, outputting the sum, and computing the next carry. This is the half-adder! Before, we strung half-adder's together *in space*, so that the carry from one was the input to the next.



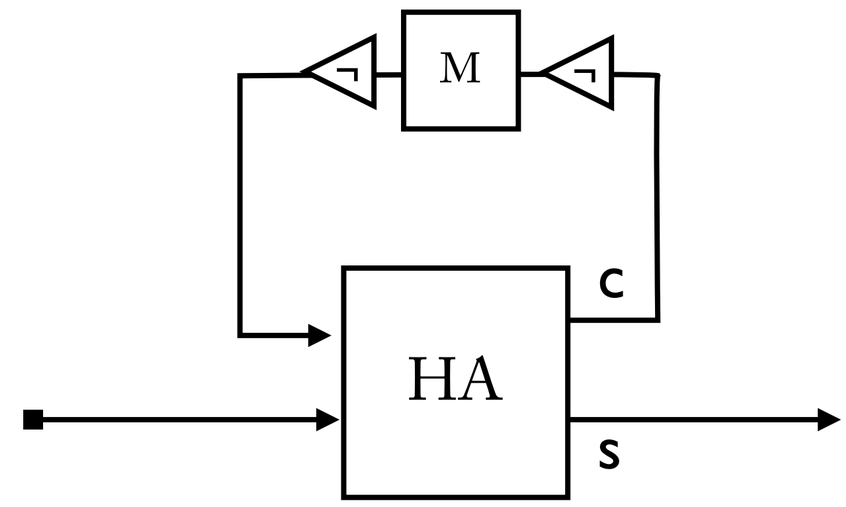
+1 B CC

Now we are going to string half-adder's together *in time*. The idea is that, at each cycle, you store the carry digit in memory, and then use that memory as input on the next cycle. Something like the circuit at right.

The only problem with this is, it doesn't have any mechanism for adding *the first 1*. (Remember how we handled this with inverters for the combinatorial circuit?)



One way to solve this problem would be to start the memory off with a 1 in it— but then have it behave like a normal memory after that. Then the memory will push a 1 out on the first cycle, this will get inputted into the half-adder, and the result will be effectively adding 1 to your binary digit. This is exactly wha the function 1-first memory does. Here you go:



+1 B SC

Prove to yourself that this works! And meditate (deeply) on the relationship between this binary +1 machine, and our old combinatorial binary +1 machine. What one does in space, the other does in time!

In-Class Problems

Design SC's which solve the following problems.

1. **+2 T**

2. **+2 B**
   Challenge: what is the smallest circuit, in terms of numbers of logic gates, that you can think of that computes this function?

3. **2*x* B**

4. **2*x*+1 B**

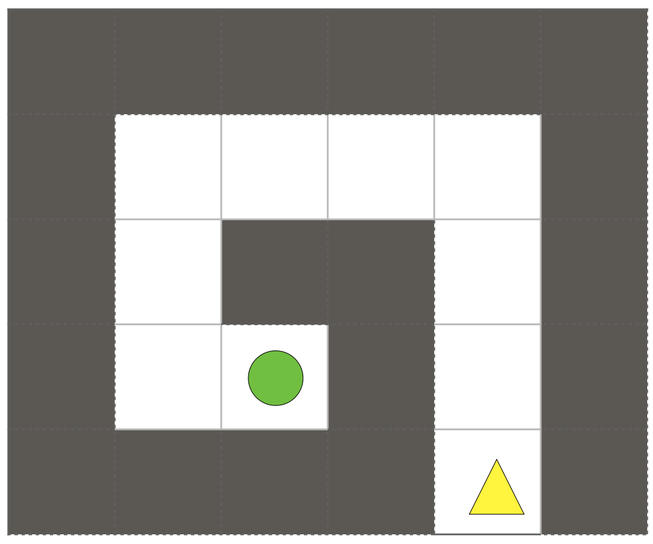
5. **Come up with an SC that computes a two-place function.** Be creative— it doesn't have to be a familiar function; define your own if you like.

6. **Pattern Finder 2**
   Output = 1 as long as current digit and the previous 3 conform to the pattern "1100" or "0011".

21. Computational Power

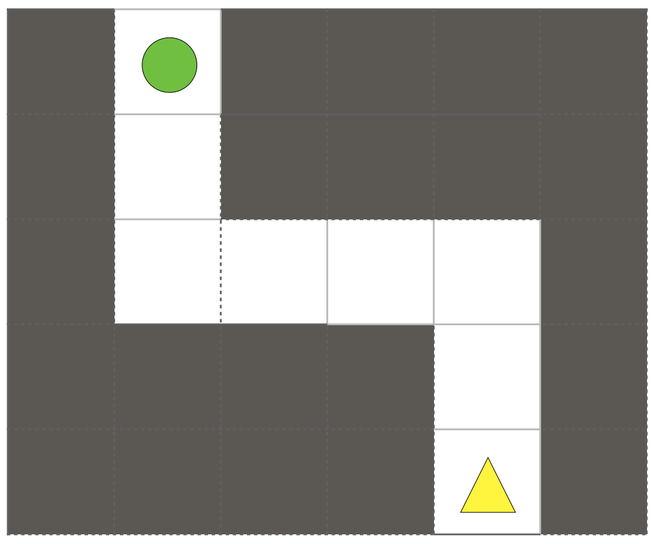
Differences in Computational Power: CCs vs SCs

Spiral



CC computable.
SC computable.

Zig-Zag



Not CC computable.
SC computable.

Differences in Power

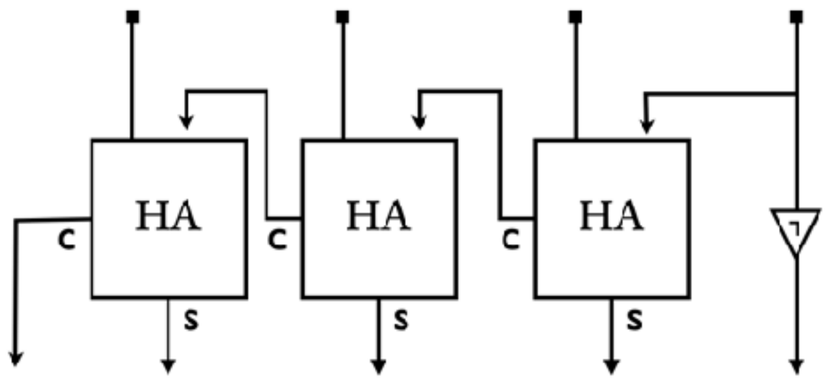
Zig-Zag is not CC computable because it requires different responses to the same input at different times. But CC's are history-independent..

Zig-Zag is SC computable, because SC memory allows the machine to enter into different input-output states over time, depending on its history.

The Memory Moral

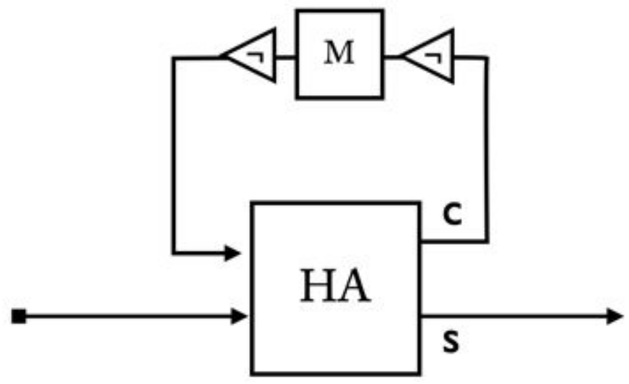
Memory allows a machine to display flexible responses to inputs. As a consequence, memory increases computational power.

Binary +1 [0-15]



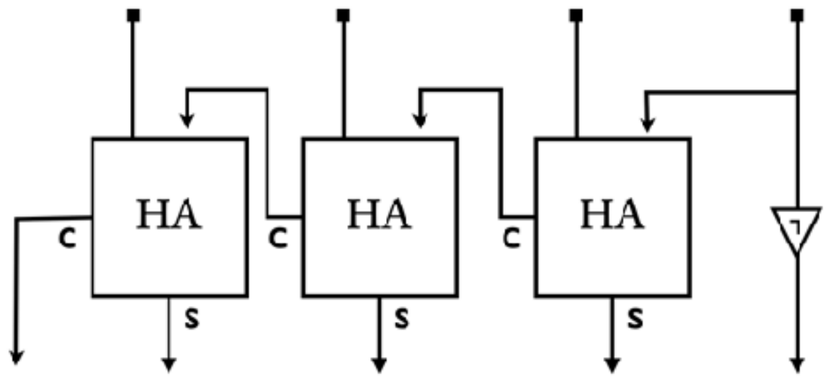
CC computable.
SC computable.

Binary +1



Not CC computable.
SC computable.

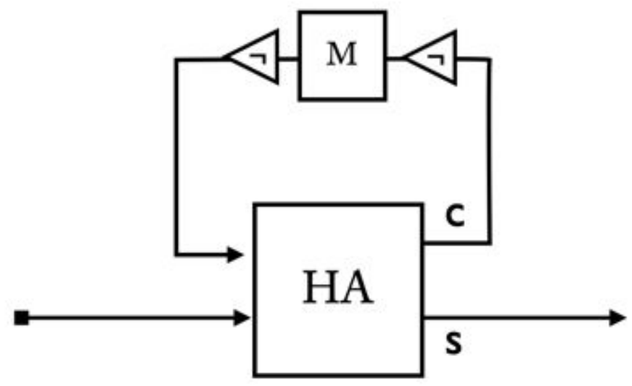
CC Powers and Limitations



CC Binary +1 [0-15]

- CC's have no memory or history responsiveness: they always respond in the same way to the same input.
- The input size for any given CC is fixed by its internal mechanisms.
- CC's can compute *any* bounded function.
- CC's compute *only* bounded functions.
- Practically: the higher the bound, the bigger the machine.

SC Powers and Limitations



SC Binary +1

- SC's contain memory and are history responsive: they can respond in different ways to the same input.
- They input size for any given SC is the infinite (or extendable) dimension of time.
- SC's can compute *any* bounded function.
- SC's compute *some* unbounded functions.
- Practically: for many functions, the higher the bound, the machine stays the same. (For other functions, this isn't true.)

"Infinite ends, finite means."

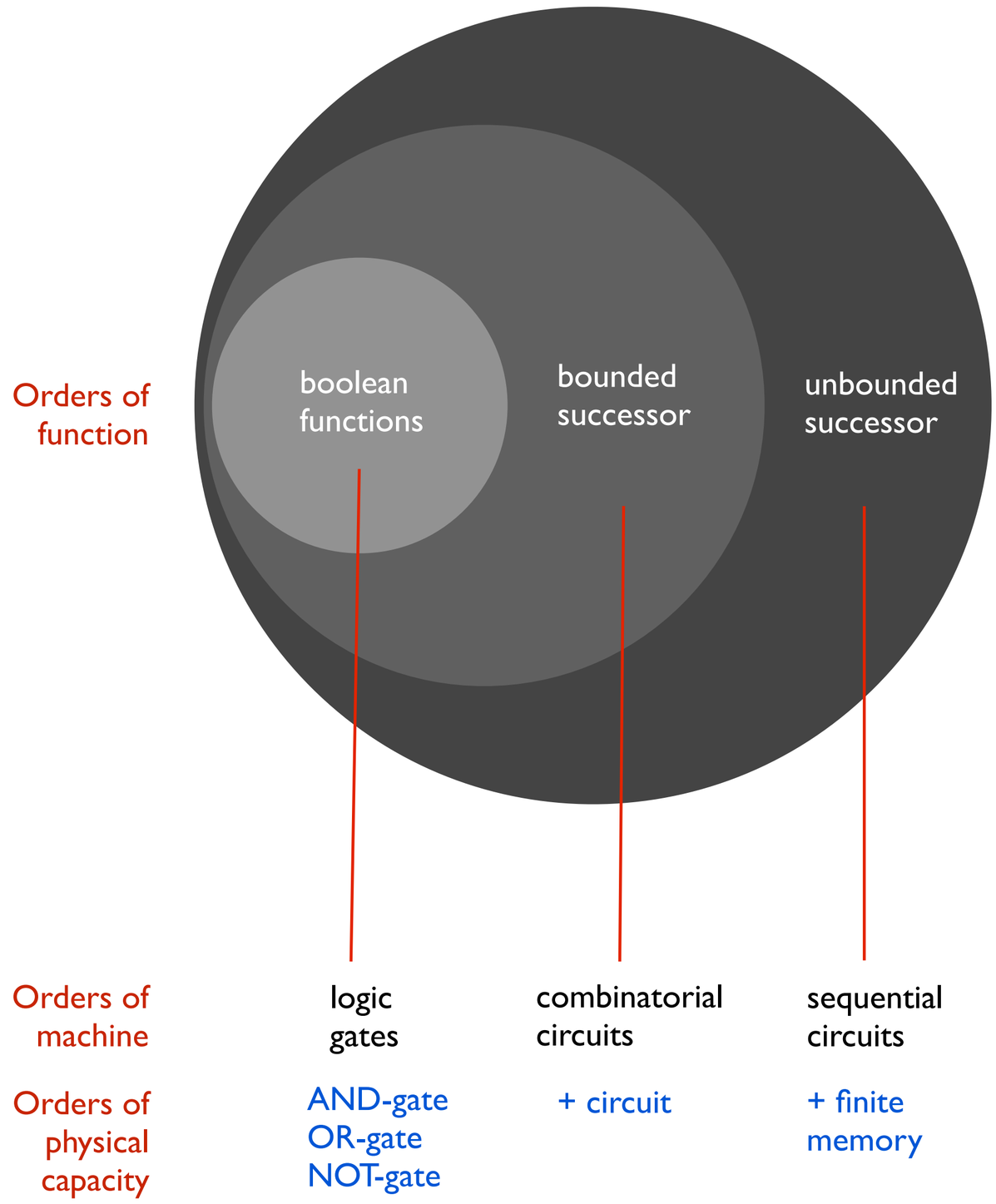
CC Implementation

- Fast.
- Limited in scope.
- The larger the problem, the larger the machine.
- Good for problems with fixed size.
- Compare: perception.

SC Implementation

- Slow.
- Unlimited in scope.
- The larger the problem, the machine stays the same.
- Good for problems with unpredictable size.
- Compare: thought.

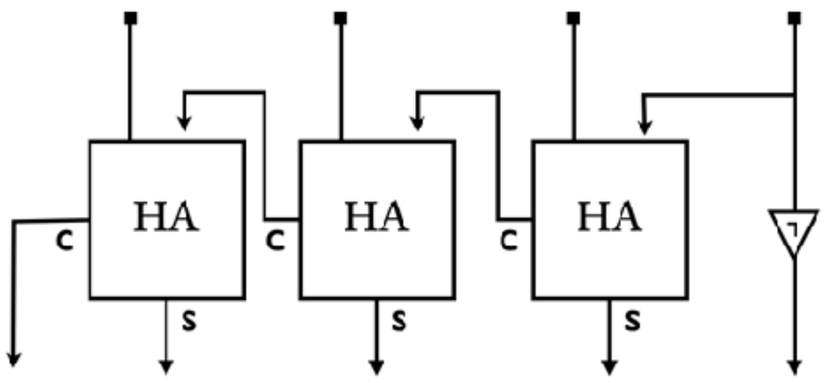
Orders of Computational Power



Definition: Computational Power
A machine M1 has greater computational power than a machine M2 iff

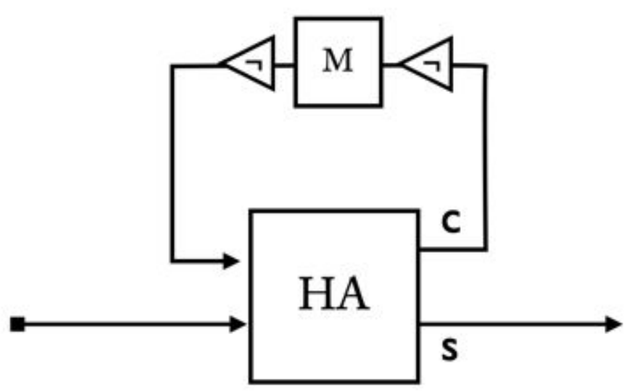
(a) for any function *f* and any code *C*:
if M1 can compute *f* under C, then M2 can compute *f* under C

(b) there some function *f* and code *C* such that:
M1 can compute *f* under *C*, but *M2* cannot compute *f* under *C*.



Binary +1 [0-15]

CC computable.
SC computable.

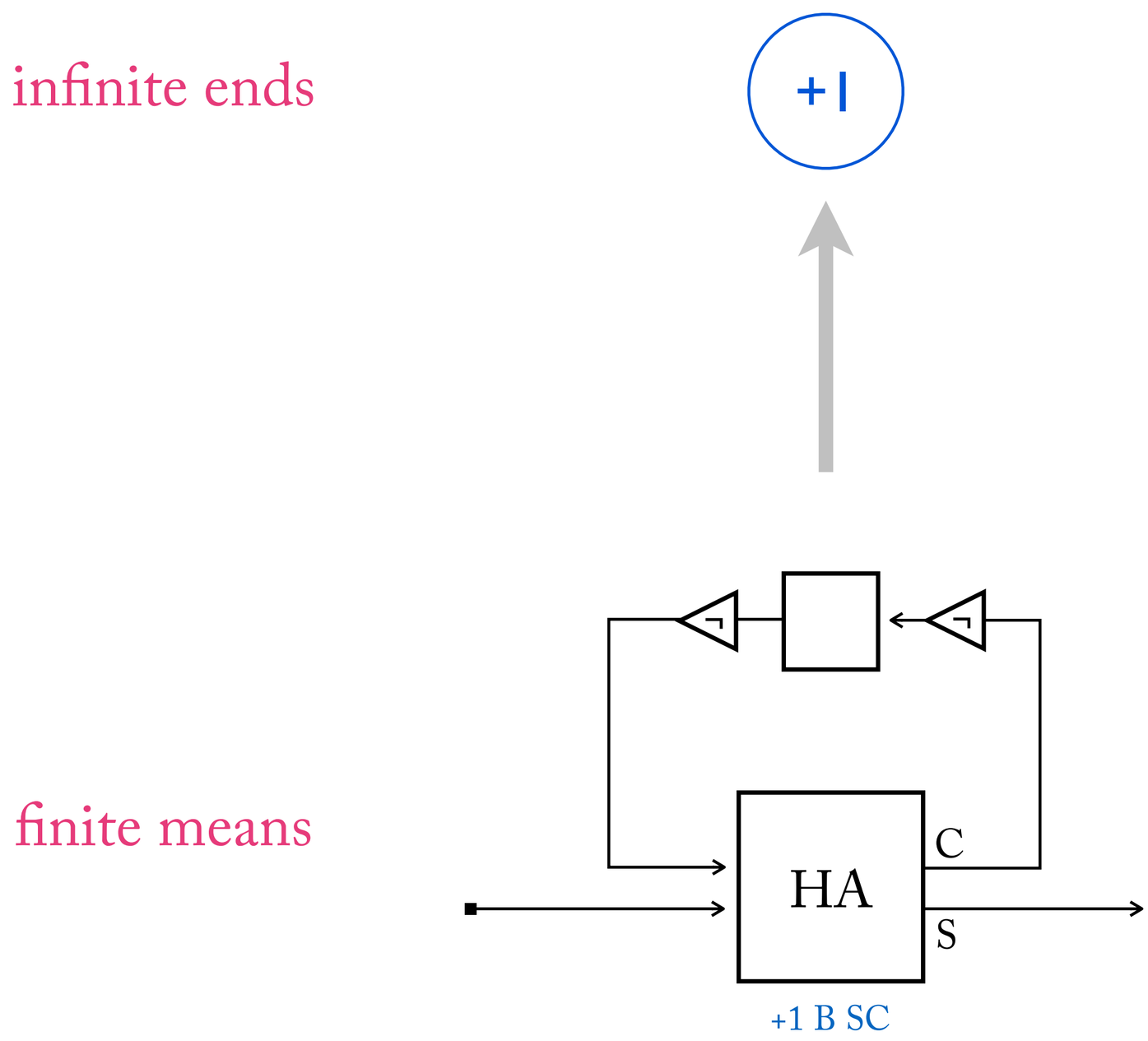


Binary +1

Not CC computable.
SC computable.

For language is quite peculiarly confronted by an an unending and truly boundless domain, the essence of all that can be thought. It must therefore make infinite employment of finite means, and is able to do so through the power who produces identity of language and thought.

— Humbolt, *On Language*, 1836



What allows SC's to overcome their own finite physical limitations? They exploit the potential infinity of time.

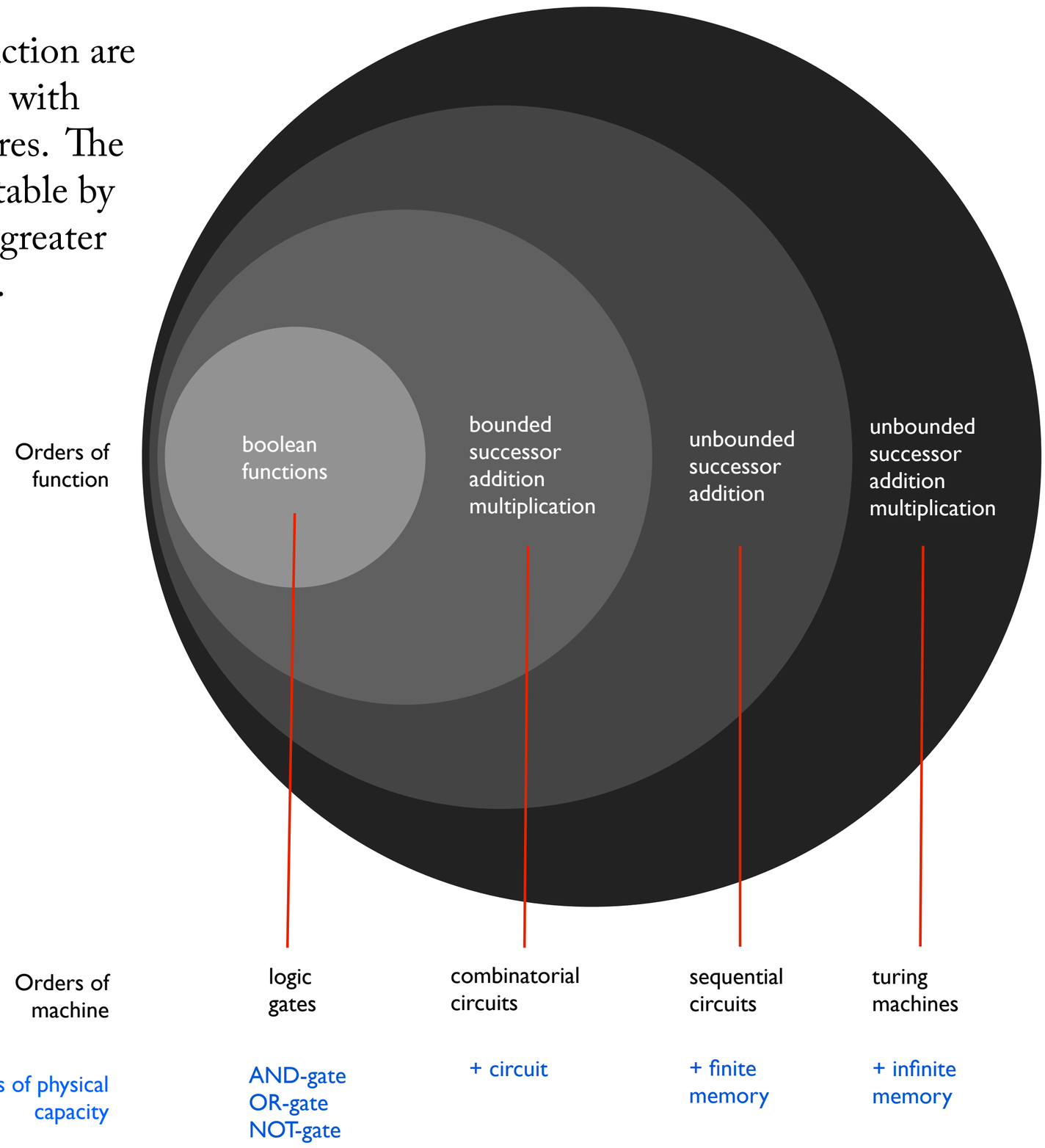
Memory isn't just necessary for achieving infinite potential. It seems to be necessary for *coherence* too.

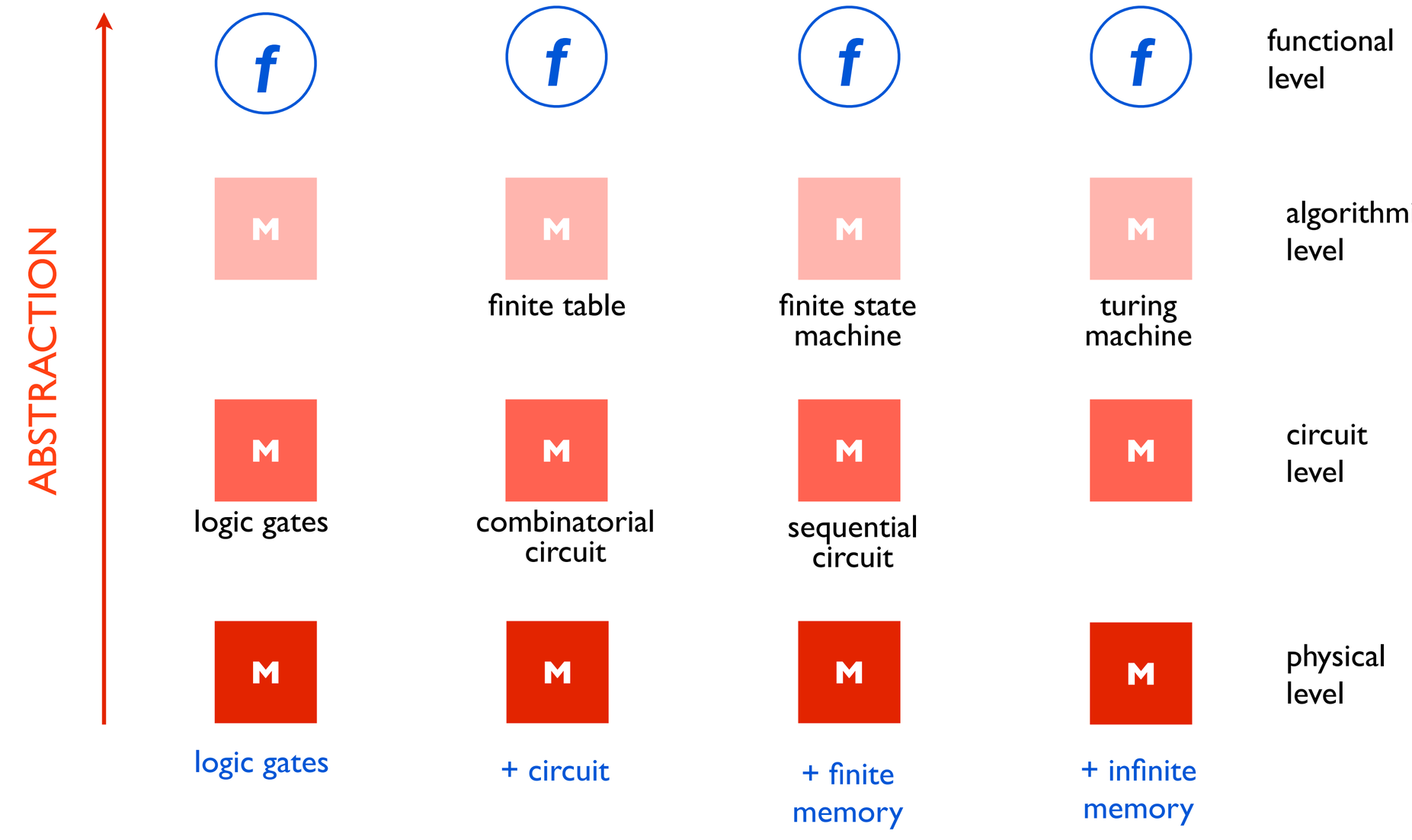
Language. I'd like to buy a carne asada taco.

Argument. I think, therefore I am.

A Map of Power and Abstraction

Different classes of function are computed by machines with different physical features. The more functions computable by a kind of machine, the greater its *computational power*.





22. The Method of State Analysis

Solving SC Computation Problems

The way that circuits with memory work can be unintuitive, and solving SC computation problems is sometimes difficult. What follows is a step-by-step method for turning SC problems into discrete sequences of CC problems. To do this, we'll have to conceptualize SCs in terms of abstract *states*— a concept will examine in greater depth in the coming chapters. Analyzing SC's in terms of states gives you a distinctive understanding of their core mechanism.

Histories and States

Over the course of the history of a machine, it will enter into various fixed states, that often repeat. Consider various sequences for the SC tripwire, and its two characteristic states.

…	t5	t4	t3	t2	t1	
…	0	0	1	0	0	IN
…	1	1	1	0	0	OUT

tripped state un-tripped state

…	t5	t4	t3	t2	t1	
…	0	0	1	1	0	IN
…	1	1	1	1	0	OUT

tripped state un-tripped state

These enduring states can be further analyzed into instantaneous states of the machine that repeat in time.

…	t5	t4	t3	t2	t1	
…	0	0	1	1	0	IN
…	1	1	1	1	0	OUT

tripped state tripped state tripped state tripped state un-tripped state

The idea behind state analysis is to describe a sequential circuit in terms of the characteristic inputs and outputs of a state over time, and the conditions under which one state leads to another, or simply repeats.

Example 1: trip-wire

Design a **trip-wire** which sets off an "alarm" (outputs constant 1's) as soon as it reads a 1 for the first time.

Step 1: Sequential input-output table. Start by making a left-to-right table of local inputs and outputs over time. (A top-to-bottom table will also work, but the horizontal layout will be helpful when we get to numerical computation.). You may want to make a few of these, for variation.

One sequence…

…	t5	t4	t3	t2	t1	
…	0	0	0	0	0	IN
…	0	0	0	0	0	OUT

Another sequence…

…	t5	t4	t3	t2	t1	
…	0	0	1	0	0	IN
…	1	1	1	0	0	OUT

Yet another…

…	t5	t4	t3	t2	t1	
…	0	0	1	1	0	IN
…	1	1	1	1	0	OUT

Step 2: State description. Consider how many "states" the circuit has, or overall ways of responding to input and output. (We'll define this idea more precisely later.) In this case, the machine has two states— before the first 1, and after the first 1. Write these down in words.

State 0

If IN=0, OUT= 0. Repeat State 0.

If IN=1, OUT=1. Go to State 1.

State 1

If IN=0, OUT= 1. Repeat State 1.

If IN=1, OUT=1. Repeat State 1.

As a rule, start with State 0, and count up from there. In general, each state description will have the following schematic form:

State X

If IN=A, then OUT= B. Go to state Y.

If IN=C, then OUT=D. Go to State Z.

Step 3: State tables. Next, make construct an input-output table for each state. But when you do this, include extra input and output columns for memory, M_{IN} and M_{OUT}. M_{IN} is the bit coming *in* from memory, and M_{OUT} is to the bit going *out* into memory.

A single memory register has two states (0 and 1). So for the current problem, we can assume that State 0 is defined by a 0 in memory, and State 1 is defined by a 1 in memory. Since M_{IN} is the digit coming in from memory at the beginning of a new cycle, State 0 corresponds to $M_{IN} = 0$, and State 1 corresponds to $M_{IN} = 1$. In the same way, an instruction to move State 0 corresponds to $M_{OUT} = 0$, and an instruction to move State 1 corresponds to $M_{OUT} = 1$.

State 0

If IN=0, OUT= 0. Repeat State 0.
If IN=1, OUT=1. Go to State 1.

State 0

M_{IN}	IN	OUT	M_{OUT}
0	0	0	0
0	1	1	1

State 1

If IN=0, OUT= 1. Repeat State 1.
If IN=1, OUT=1. Repeat State 1.

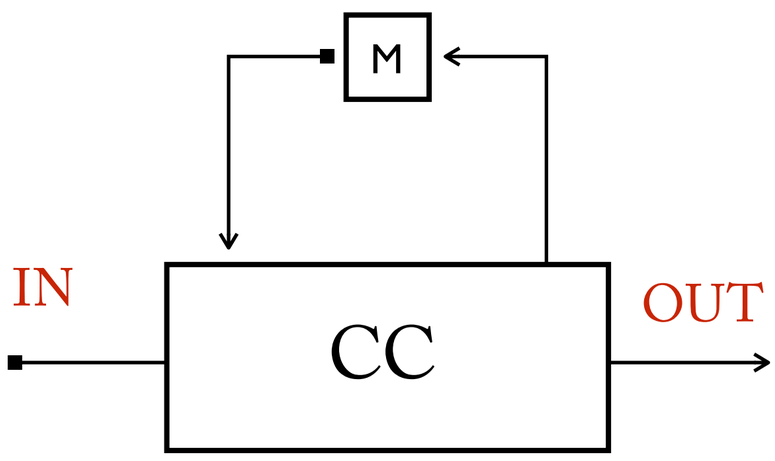
State 1

M_{IN}	IN	OUT	M_{OUT}
1	0	1	1
1	1	1	1

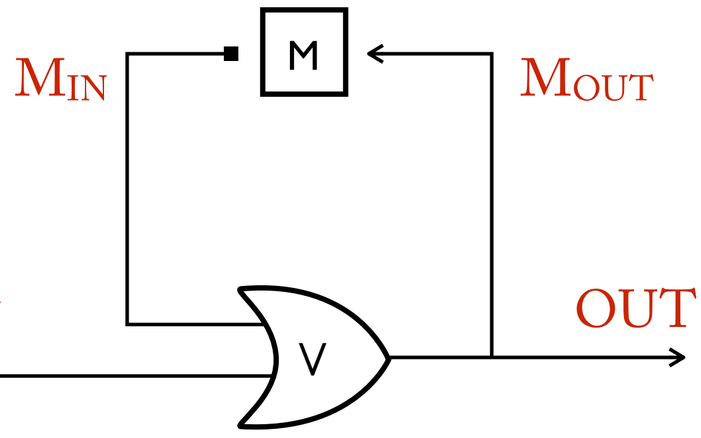
Step 4: CC design. Assemble these states into a single input output table, and design a CC which computes this table. And design a CC to solve it.

M_{IN}	IN	OUT	M_{OUT}
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

Step 5: SC design. Convert your CC design into a SC, follwing the schematic layout of an SC:



SC schematic layout



SC tripwire

+1 T: Step by step

Design an SC which computes +1 T.

Step 1. Start by making a global-input output table, the way you might for any numerical function.

IN	OUT
0	1
1	11
11	111
111	1111
…	…

Remember you can always add 0's to the left, without changing the global inputs and outputs.

IN	OUT
0000	0001
0001	0011
0011	0111
0111	1111
…	…

To get a sense for what's going on here, make a left-to-right table of local inputs and outputs over time. You will want to look at a few different variations. Key cases to always check for are x=0 and x≠0.

+1 T where x=0

t5	t4	t3	t2	t1	
…	0	0	0	0	IN
…	0	0	0	1	OUT

+1 T where x=2

t5	t4	t3	t2	t1	
…	0	0	1	1	IN
…	0	1	1	1	OUT

Step 3 & 4. Divide the machine up into states. This step isn't formulaic; you'll have to think about how the computation intuitively works. Then make an I/O table for each state.

State 0

If IN=0, OUT= 1. Go to State 1.
If IN=1, OUT=1. Repeat State 0.

State 1

If IN=0, OUT= 0. Repeat State 1.
If IN=1… **never happens!**

State 0

I	M→	O	→M
0	0	1	1
1	0	1	0

State 1

I	M→	O	→M
0	1	0	1
1	1		

Now combine these two state descriptions so it looks like a familiar input-output table:

I	M→	O	→M
0	0	1	1
1	0	1	0
0	1	0	1
1	1		

Step 5. Design a circuit which handles the table from Step 4.

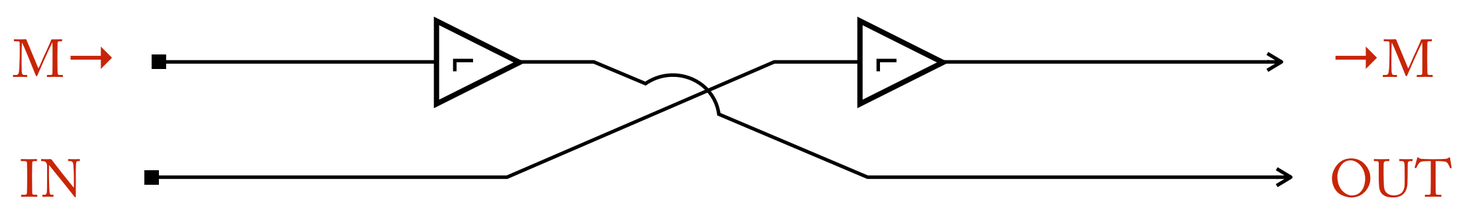
I	M→	O	→M
0	0	1	1
1	0	1	0
0	1	0	1
1	1		

Typically it's useful to solve for the O and →M columns separately. Here I'd note two patterns:

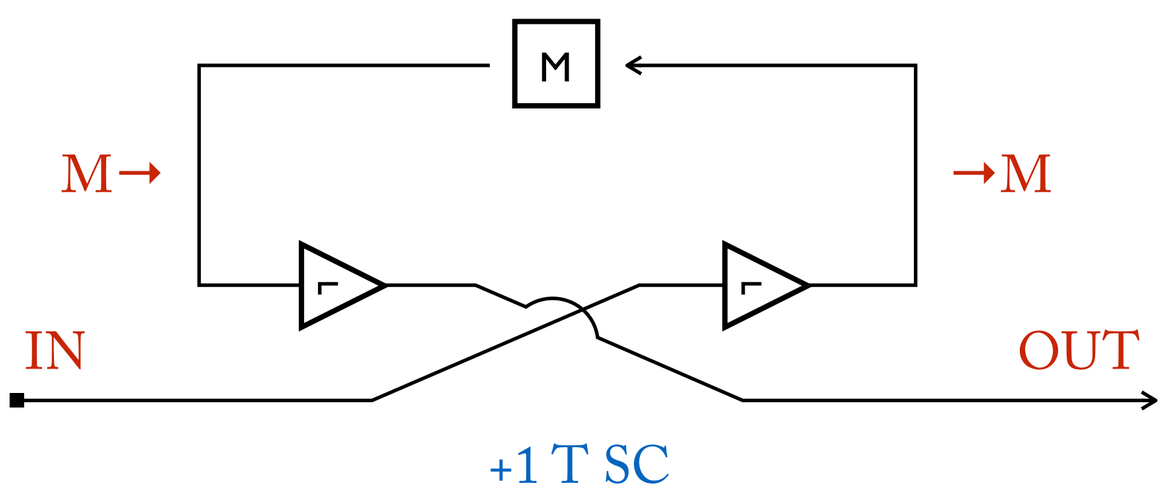
(1) Inverting the M→ value always gives you the O value.

(2) Inverting the I value always gives you the →M value.

We can draw this up as a simple circuit:



We now interpret "M→" as *reading from memory*, and we interpret "→M" as *writing to memory*. The result is our final circuit.



23. State Abstraction

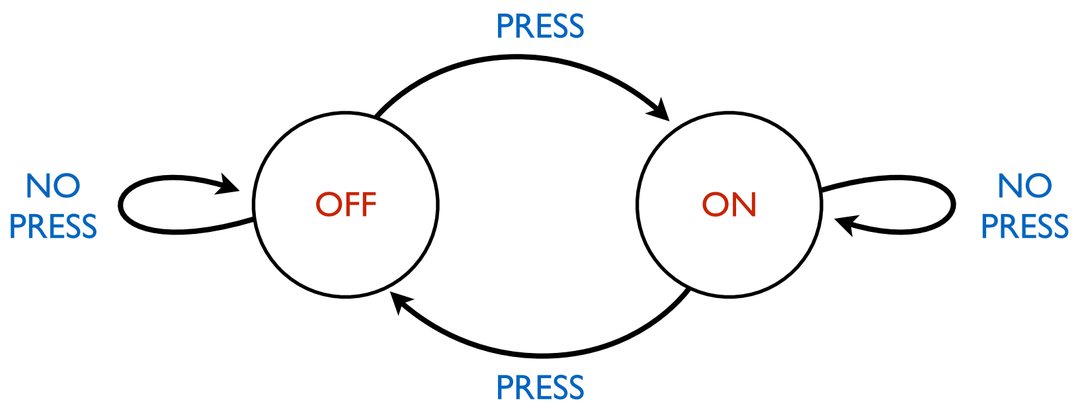
The Concept of State

The concept of a state is at once one of the most profound and one of the most trivial ideas we will explore in this class. On the trivial end, identifying a machine's states is little more than some tedious book-keeping. (And that'll be the focus of this lesson.)

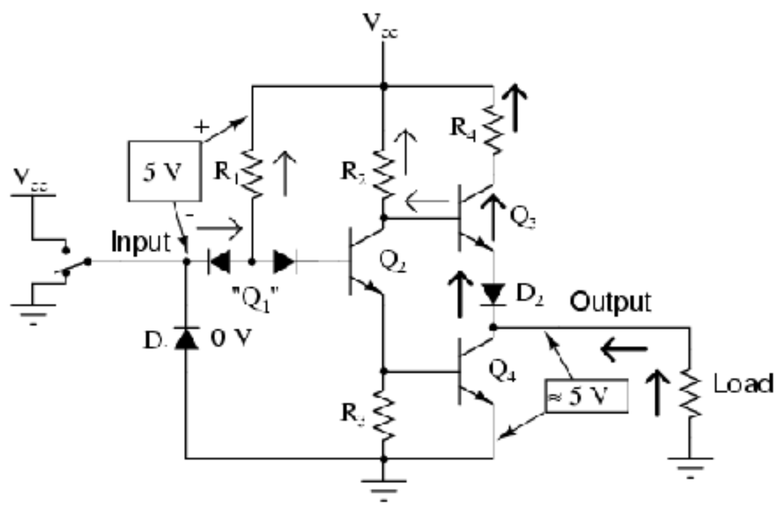
On the profound end, once defined, the concept of a machine state provides us with fundamental insights about the relationship between low-level physics and high-level cognition. The philosopher Hillary Putnam even thought that the concept of machine states solved (or dissolved!) the mind-body problem. You'll have to judge for yourself.

A **state** is the overall configuration of a given system. Given an input, a system's state determines its current output and what state it will be in next. **State abstraction** leaves aside particular details of circuitry, and describes system purely in terms of inputs, outputs, and state transitions.

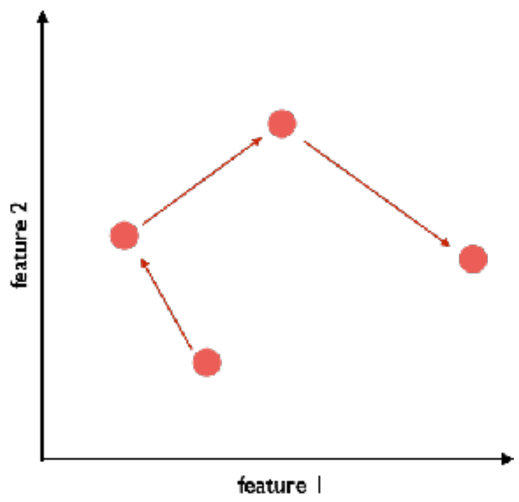
For example, here is a diagram which gives the state abstraction of the on-off switch. It describes the machine entirely in terms of its overall dispositions to produce inputs in response to outputs.



ON/OFF Switch



Whereas a machine as a complicated internal configuration at any given point…



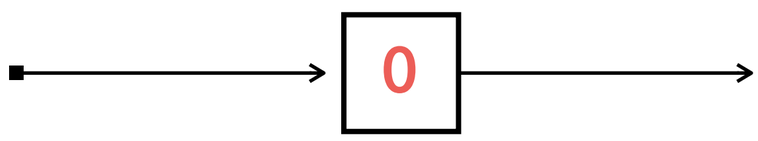
We can imagine a **state space** of all possible internal configurations. Each state is a point in this space.

States of an SC

In the case of sequential circuits, a state is the total "input-output disposition" of the machine at a given moment. It includes how the machine will respond to all possible inputs at a time. It also includes what states the current state will lead to. It **doesn't** include the actual circuity of the machine.

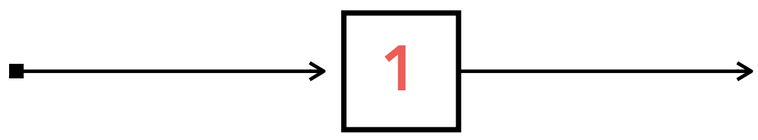
Note: there is an ambiguity about whether two different memory settings, with same next state, and same I/O should be counted as "same state"

The total input-output disposition of a machine is determined by the internal circuitry of the machine and the configuration of its memory registers at a time. All machines start in the "initial state," where every memory register is set to 0. This determines one set of responses to possible inputs. Other settings of the memory registers result in different responses to possible inputs, hence different states.



State 0

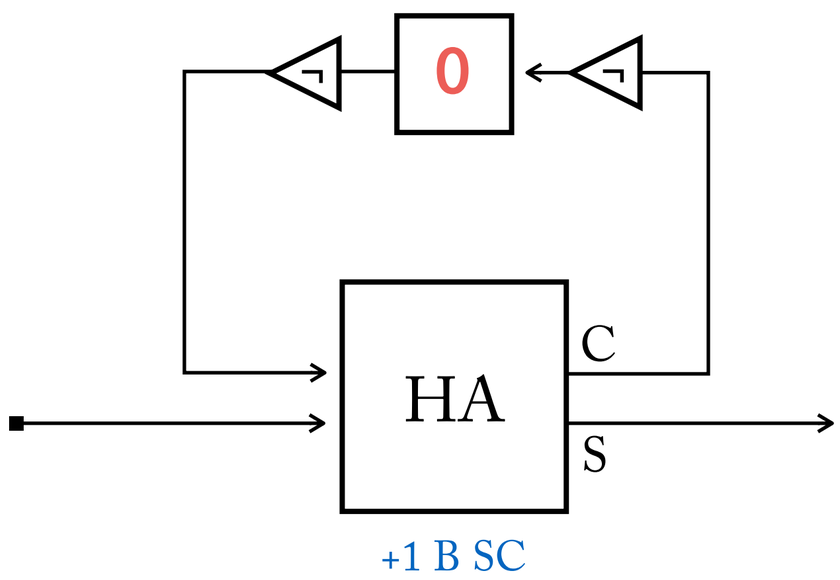
IN	OUT
1	0
0	0



State 1

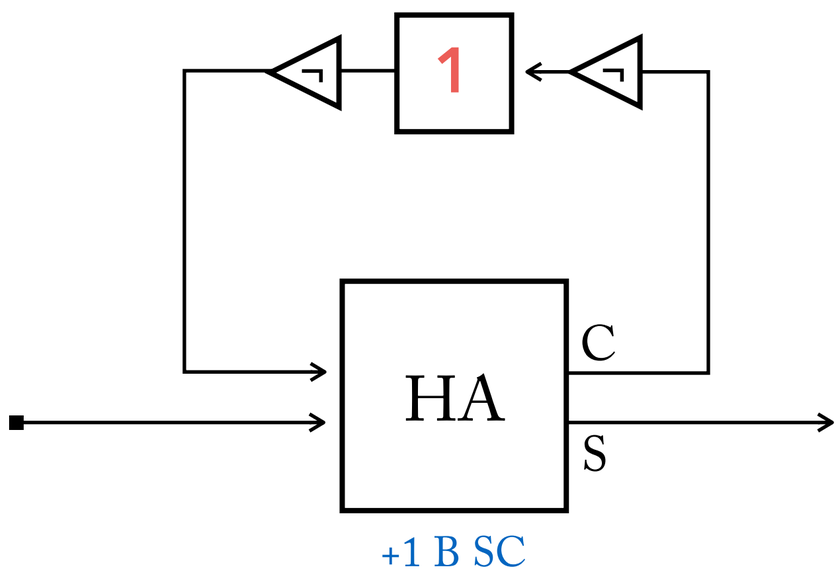
IN	OUT
1	1
0	1

Here's a more interesting example:



State 0

IN	OUT
1	0
0	1



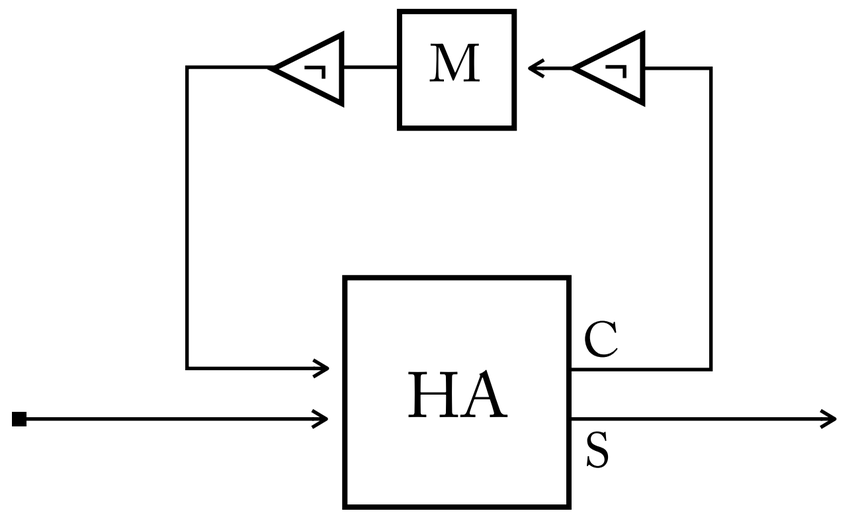
State 1

IN	OUT
1	1
0	0

But there is more to a state that just input and output. There is also its **relation to other states**…

State Tables

Step 0
Design a sequential circuit which computes a specific function.



Step 1
Make a table of all possible inputs from memory and inputs from the user.

If there are more memory registers, or more inputs from the user, there will be more rows in the table.

M→	IN	OUT	→M
0	0	1	1
0	1	0	0
1	0	0	1
1	1	1	1

Step 2
Fill out the table with all of the outputs to the user and all of the outputs to memory.

Step 3
Add a column at the left for "**current state**" and a column on the right for "**next state**".

Think of the input from M→ as describing the current *total memory configuration* of the machine.

CURRENT STATE	M→	IN	OUT	→M	NEXT STATE
S_0	0	0	1	1	A
S_0	0	1	0	0	S_0
A	1	0	0	1	A
A	1	1	1	1	A

Step 4
For each distinct possible configuration of memory, enter a distinct state name in the "**current state**" column. For the configuration where all memories are set to 0, that state should be called S_0. After that you can call the states anything you like.

Step 5
Using the state names chosen in Step 4, fill out the "**next state**" column. The value of →M determines the next value of M→, which determines the identity of the next state.

Step 6
Merge the cells in the "**current state**" column that correspond to the same state.

CURRENT STATE	M→	IN	OUT	→M	NEXT STATE
S_0	0	0	1	1	A
	0	1	0	0	S_0
A	1	0	0	1	A
	1	1	1	1	A

Step 7
Erase the memory columns.

… and you've got yourself a **state table**!

CURRENT STATE	IN	OUT	NEXT STATE
S_0	0	1	A
	1	0	S_0
A	0	0	A
	1	1	A

Step 8 (Optional)
Eliminate unused states: eliminate any state that either:
(1) cannot be reached by finite steps, beginning with S_0;
(2) will not be reached by finite steps, beginning with S_0, given the representational code (binary, tally) intended for this machine.

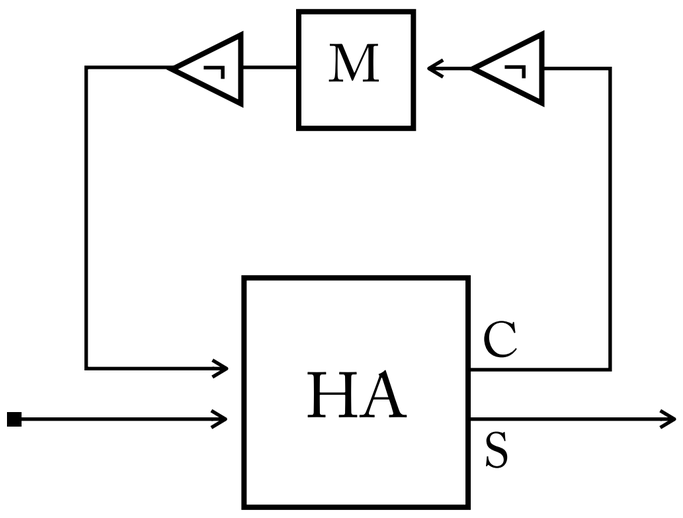
Note: this step is not *strictly* required, but it is best practice.

For purposes of illustration, here is a table for a machine with two memory registers, before the memory columns have been erased.

STATE	M1	M2	IN	OUT	NEXT
S_0	0	0	1	1	S_0
			0	1	A
A	0	1	1	0	S_0
			0	1	C
B	1	0	1	0	B
			0	0	C
B	1	1	1	1	C
			0	1	S_0

The grey row is eliminated by Step 8(1).

From State Tables to State Diagrams



+1 B SC

+1 B SC determines the following state table.

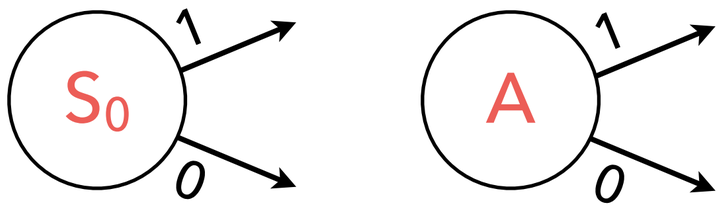
STATE	IN	OUT	NEXT
S_0	0	1	A
	1	0	S_0
A	0	0	A
	1	1	A

+1 B State Table

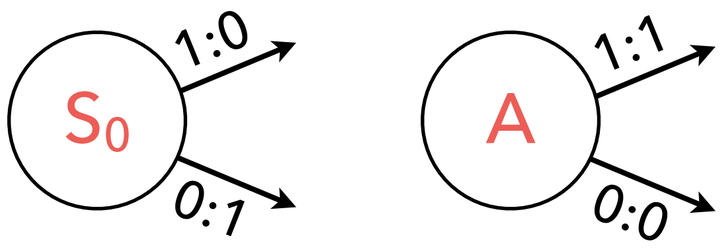
To construct a state diagram from a state table…

Step 1
For each state of the machine draw a circle, and label it with the name of the state.

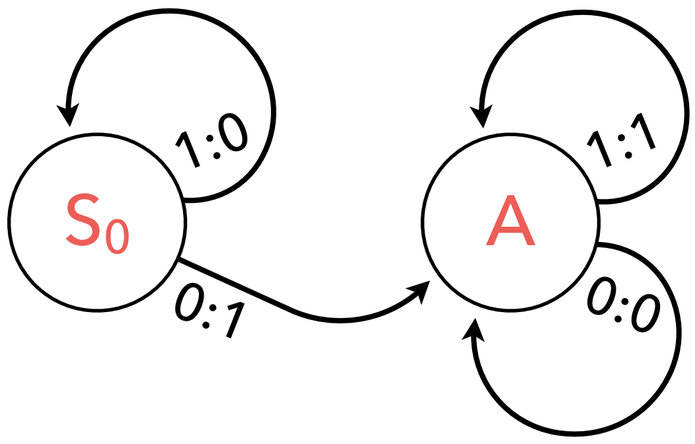
Step 2
For each circle draw up to two arrows exiting it, one labeled "1", the other "0" corresponding to the two possible inputs for that state.



Step 3
For each arrow, add the notation "input : output", corresponding to the input and output associated with the given state transition.



Step 4
For those inputs which lead to a next state, connect the arrow to that state.

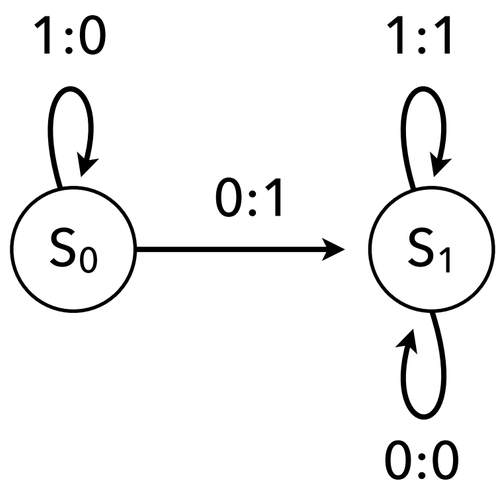


Note: you'll often find diagrams with some of the I/O arrows omitted because they don't arise in computation for a given problem.

STATE	IN	OUT	NEXT
S_0	0	1	A
	1	0	S_0
A	0	0	A
	1	1	A

+1 B State Table

equivalent descriptions



+1 B State Diagram

24. The Concept of Abstraction

Abstraction, the Very Idea

Abstraction is the transition from a more specific (**low level**) description of some system to a more general (**high level**) description of that system . Abstraction always involves both a loss of information and a gain in generality. But abstractions are not arbitrary deletions of detail. Instead, they capture **"real patterns"** in lower-level phenomena.

For example, consider the following two systems made up of circles.

We can describe each system in terms of the relative location of colored **circles**. Or we can describe each as conforming to the shape of a **square**.

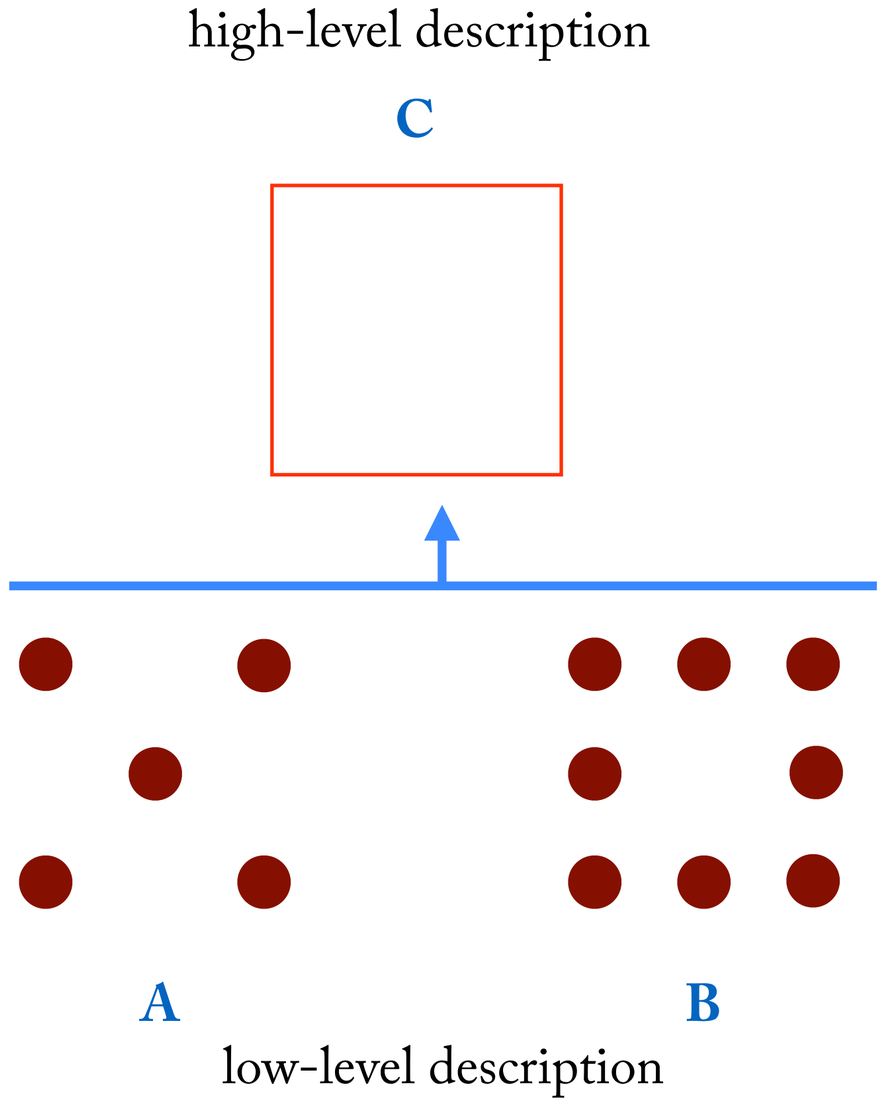
The relationship between the lower level and higher level descriptions here has many of the signature feature of abstraction in general.

Multiple realizability is the hallmark of abstraction. A description X is multiply realizable iff there is more than one (more) concrete description that can realize X. C is realized by both A and B.

Loss of detail. The high-level description C leaves out the composition and distribution of the circles that make up A and B.

Addition of deep structure. C explicitly introduces a level of structure, a "real pattern," which is present in both A and B, but explicit in neither.

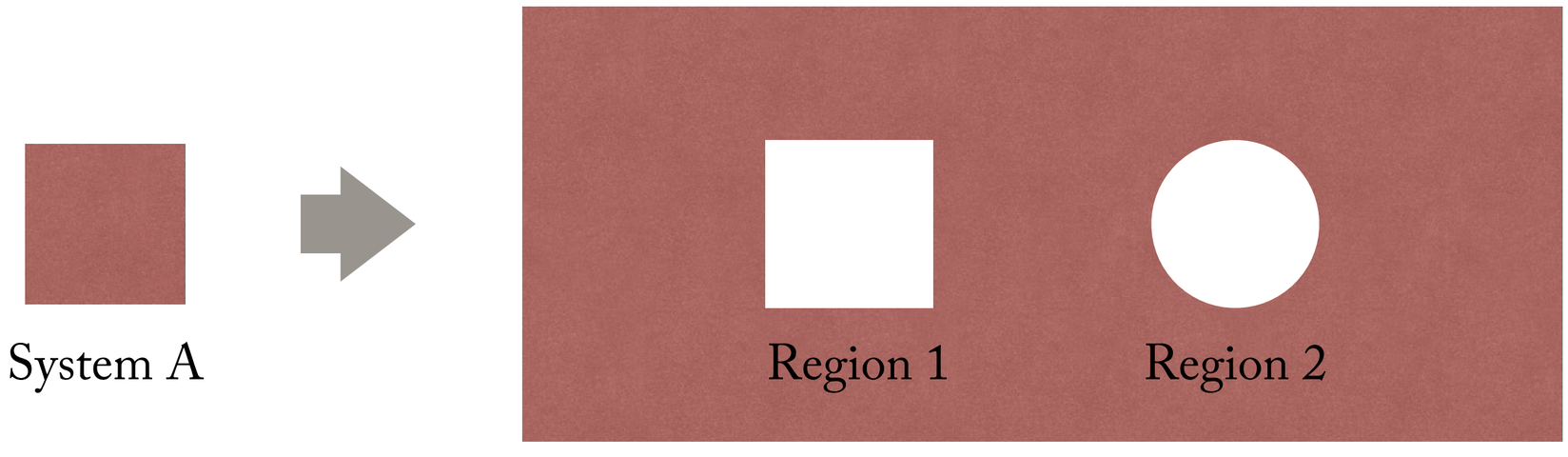
Autonomy. We can understand and reason about a system as an instance of C, without knowledge of whether it is A or B.



Abstraction as Explanation

Putnam's puzzle (from "Philosophy and our Mental Life," 1975):

> "Suppose we have a very simple physical system - a board in which there are two holes, a circle one inch in diameter and a square one inch high, and a cubical peg one-sixteenth of an inch less than one inch high. We have the following very simple fact to explain:
>
> *the peg passes through the square hole, and it does not pass through the round hole.*" (p. 295)



Consider two possible explanations of this fact.

> **Explanation 1:** microphysical deduction.
> System A is rigid lattice of atoms where atom a_1 is in position k_1 with features f_1, a_2 is in position k_2 with features f_2, … (and so on for every atom in A). It follows from the laws of quantum mechanics that, among all possible trajectories of A through space-time, A passes through R1 in some, but never passes through R2
>
> **Explanation 2:** geometrical deduction.
> A is a square with sides just under 1 inch across, and diagonal just under $\sqrt{2}$ inches. Since R1 is a square shape with sides 1 inch across, A passes through R1. Since R2 is 1 inch across at its widest point, and A is $\sqrt{2}$ inches across at its widest point, and $\sqrt{2}>1$, A does not pass through R2.
>
> **Verdict:** clearly E2 is the superior explanation; it's not clear that E1 is an explanation at all. But why? (p. 295)

In defense of the high-level explanation: **relevance.**

- "In this explanation certain *relevant structural features of the situation* are brought out."
- "*The ultimate constituents don't matter…only the higher level structure matters.*"
- "The explanation at the higher level brings out the relevant geometrical relationships. The lower level explanation conceals those laws. Also notice that the higher level explanation applies to a much more interesting class of systems."

In defense of the high-level explanation: **generality.**

- "The same explanation will go in any world (whatever the microstructure) in which those *higher level structural features* are present. In that sense *this explanation is autonomous*."
- "The higher level explanation is far more general, which is why it is *explanatory*."
- "An explanation is superior if it is more general."

The Moral: abstraction plays an essential role in any successful scientific explanation.

Functional Abstraction

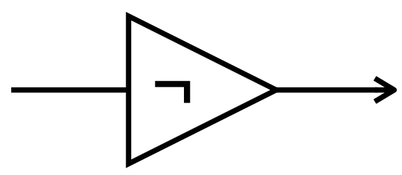
There are many different kinds of abstraction. We may describe a planet as an irregularly shaped solid, as a ball, or as point. We may describe a person as loose affiliation of particles, as a coordinated system of cells, as an animal, or as a rational agent.

In cognitive science and computational theory, **functional abstraction** dominates. This is a kind of abstraction that focuses in on a system's *function* or *purpose*, and abstracts away from details of how it achieves that function. Still there are many kinds of functional abstraction. We've already encountered several!

Functional Abstraction 1: Input-output Abstraction

Logic gates themselves can be thought of as functional abstractions which make types of inputs and outputs relevant and let everything else fade away.

They abstract away from the internal description of a machine by specifying only inputs and outputs. And they abstract away from the external description by specifying inputs and outputs only as "0" and "1".



A NOT-gate.

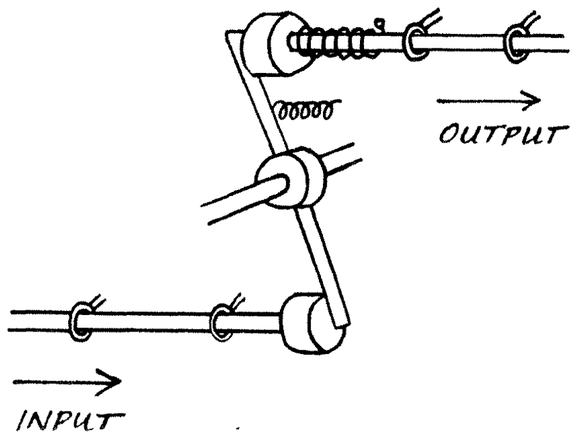
IN	OUT
0	1
1	1

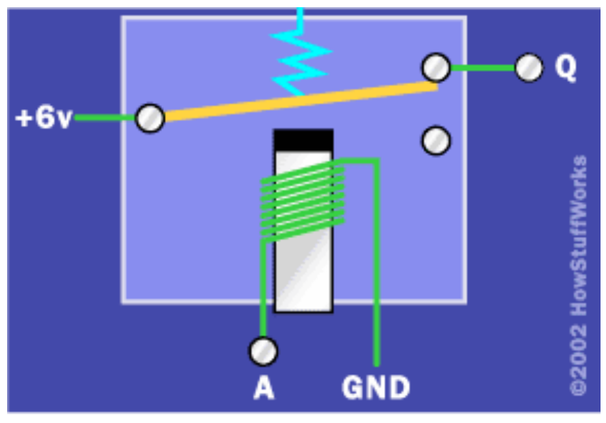
Abstraction over external mechanics.

IN	OUT
→	←
←	→

IN	OUT
⚡	⚡
⚡	⚡

Abstraction over internal mechanics.

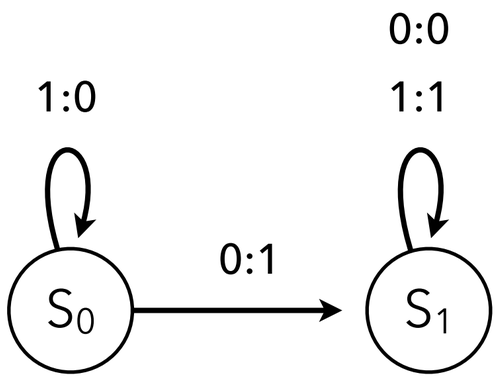




Physical implementations of a NOT-gate.

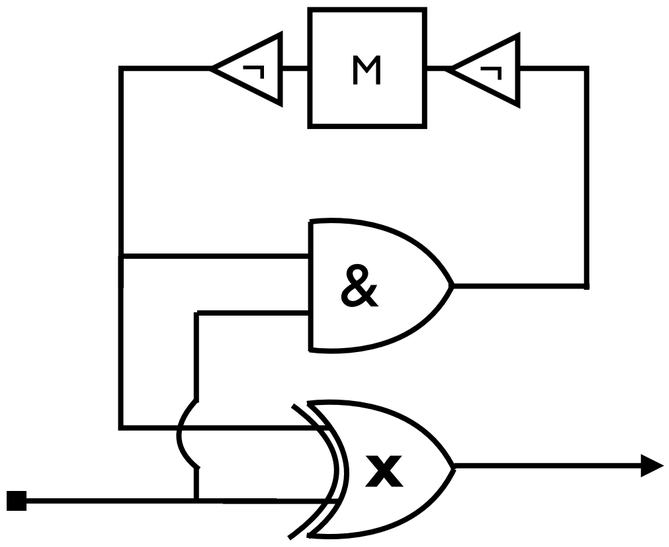
Functional Abstraction II: Piecewise Abstraction

Functional Abstraction III: State Abstraction

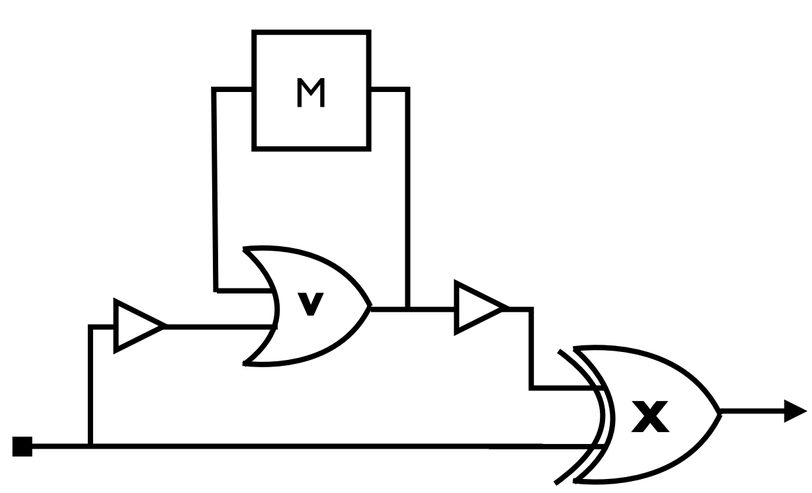


+1 B FSM

Abstraction over internal machinery.



+1 B SC



+1 B SC

In state abstraction we find all of the central hall marks of abstraction itself.

Multiple realizability: Multiple SC's realize the same FSM.

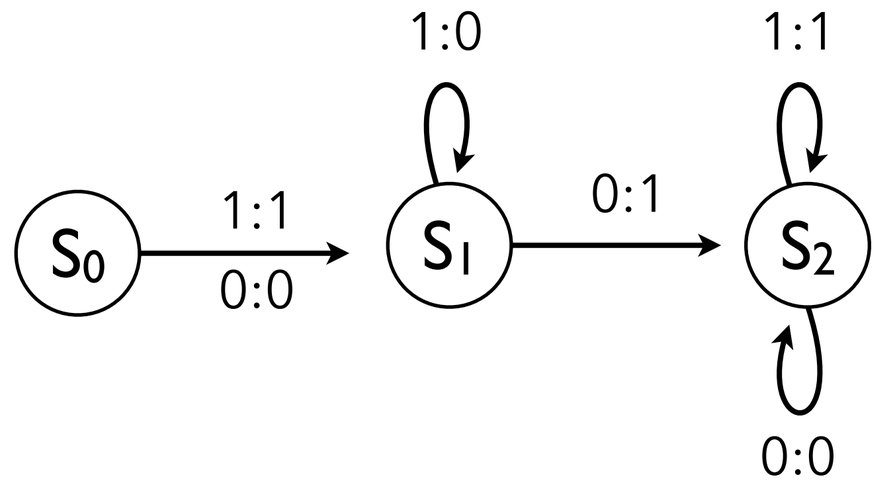
Loss of detail. The FSM leaves out details of circuitry.

Addition of deep structure. The FSM reveals the inner procedure that each SC is following.

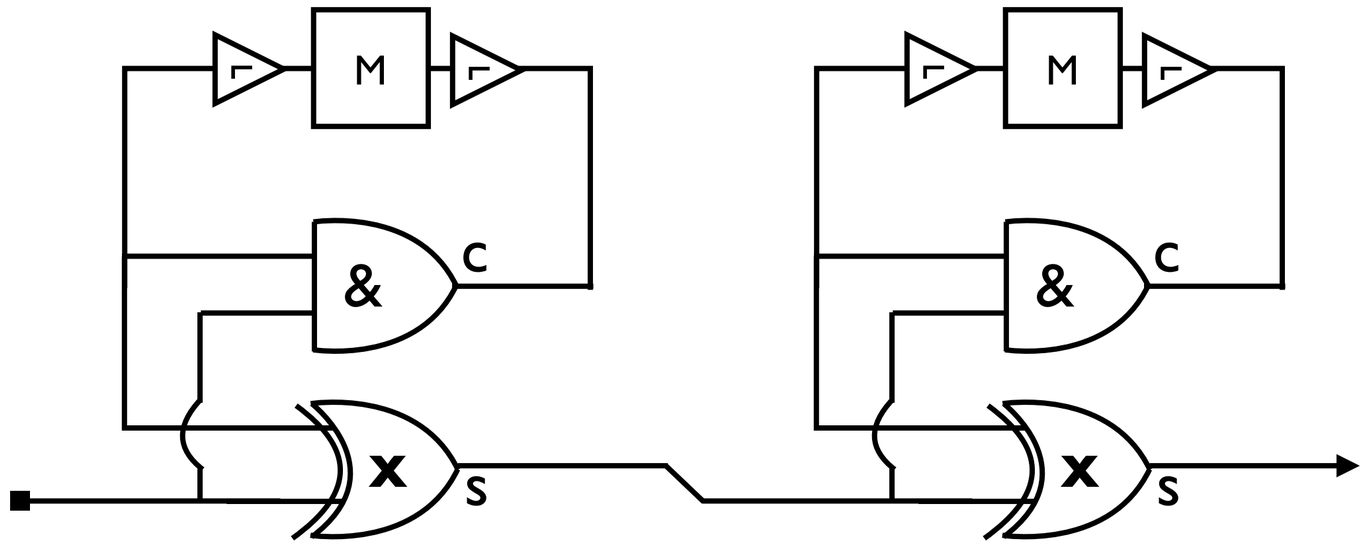
Autonomy. We can understand and reason about an FSM without understanding SC's, and visa versa.

State Machines and Sequential Circuits

FSMs describe pathways through overall configurations of the system. As the system changes, it flows through the state diagram. Each node represents a *temporal stage* in the machine's history.



SCs describe pathways of information. Information flows through the circuit as computation is carried out. Each nodes represents a *spatial* part of the machine.



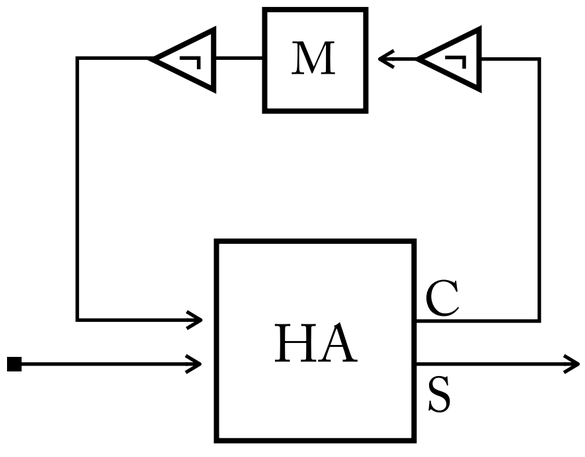
Both information flow diagram and state flow diagram describe systems, but in different ways.

We can think of a human as consisting of a complex circuit. Or we can think of them as implementing a complex algorithm and moving from state to state. Here we see the seeds of the contrast between neuroscience and psychology.

State abstraction marks an important kind of independence from the physical level. They are the first indication that computations can be carried out by neurons.

State Abstraction and the Mind-Body Problem

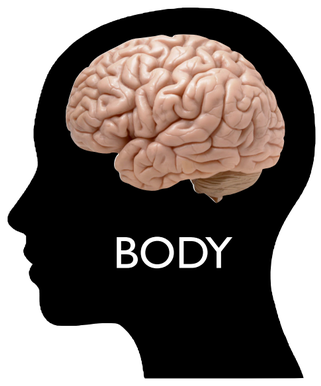
Consider now two very different descriptions of the same machine:

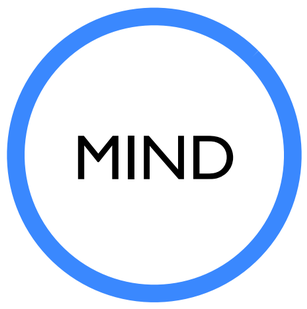


+1 B SC

STATE	IN	OUT	NEXT
S_0	0	1	A
	1	0	S_0
A	0	0	A
	1	1	A

+1 B State Table





Putnam (1967) "The Nature of Mental States" argued that state abstraction was the key to unlocking the mind-body problem. He thought the SC-State Table relationship was essentially equivalent to the body-mind relationship.

Since there is nothing mysterious about the way that SC's determine State Tables, there should be nothing mysterious about body-mind relationship. The implicit idea is that *mental states* are just *machine states*, for the right sort of human machine.

"I shall, in short, argue that pain is not a brain state, in the sense of a physical-chemical state of the brain (or even the whole nervous system), but another *kind* of state entirely. I propose the hypothesis that pain, or the state of being in pain, is a functional state of a whole organism." (Putnam 1967)

"Even though octopus and mammal are examples of parallel (rather than sequential) evolution, for example, virtually identical structures (physically speaking) have evolved in the eye of the octopus and in the eye of the mammal, notwithstanding the fact that this organ has evolved from different kinds of cells in the two cases. Thus it is at least possible that parallel evolution, all over the universe, might *always* lead to *one and the same* physical 'correlate' of pain… Thus any animal that we count as capable of these various states will at least seem to have a certain rough kind of functional organization."

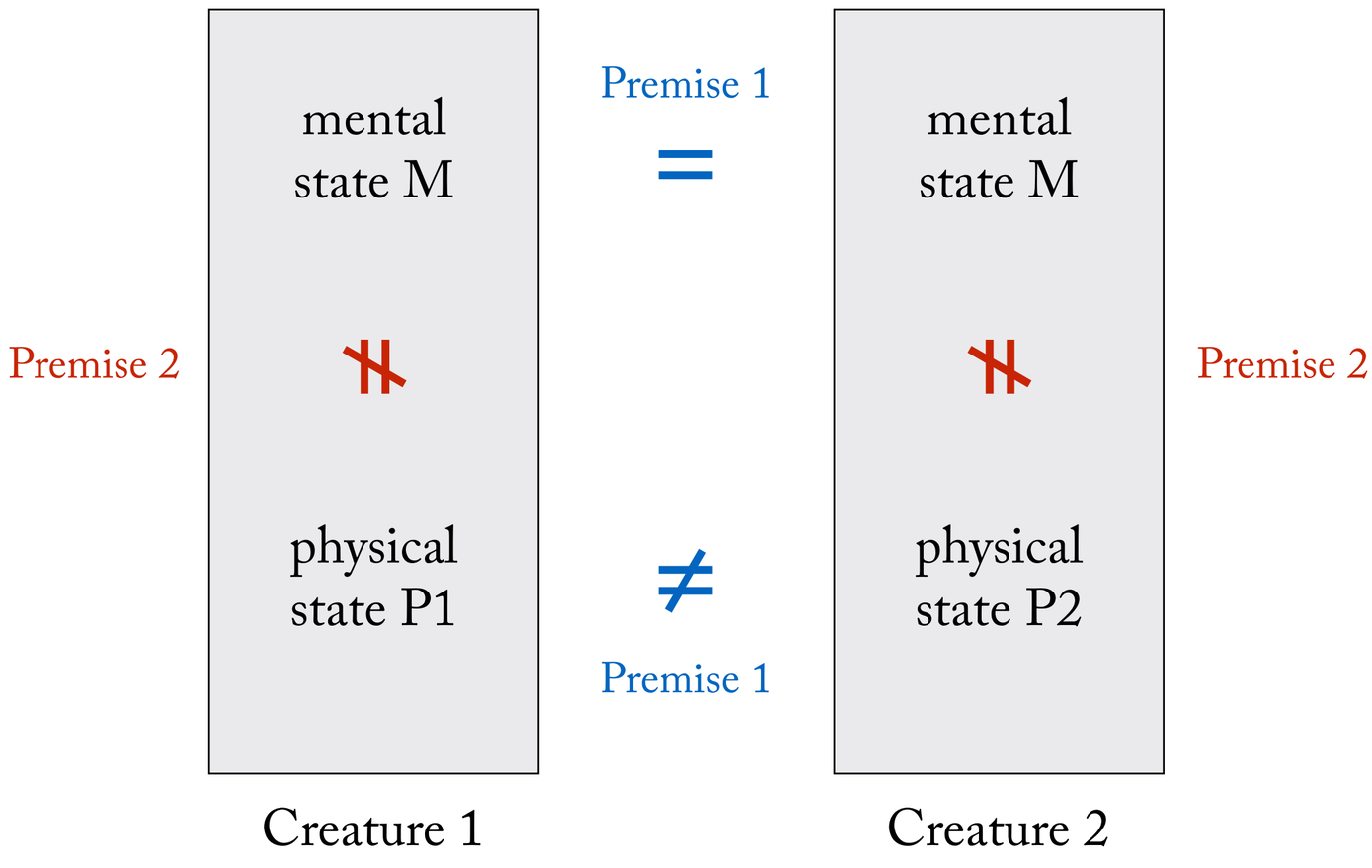
An Argument for Dualism?

1. In many cases, the same mental state can be physically realized in different ways by different creatures. Examples: beliefs, thoughts, perceptions, emotions.

2. Identity is transitive. Therefore: the mental states from Premise 1 cannot be identical with the physical states they are realized by.

3. According to physicalism, mental states are identical with physical states.

4. Therefore, physicalism is false.

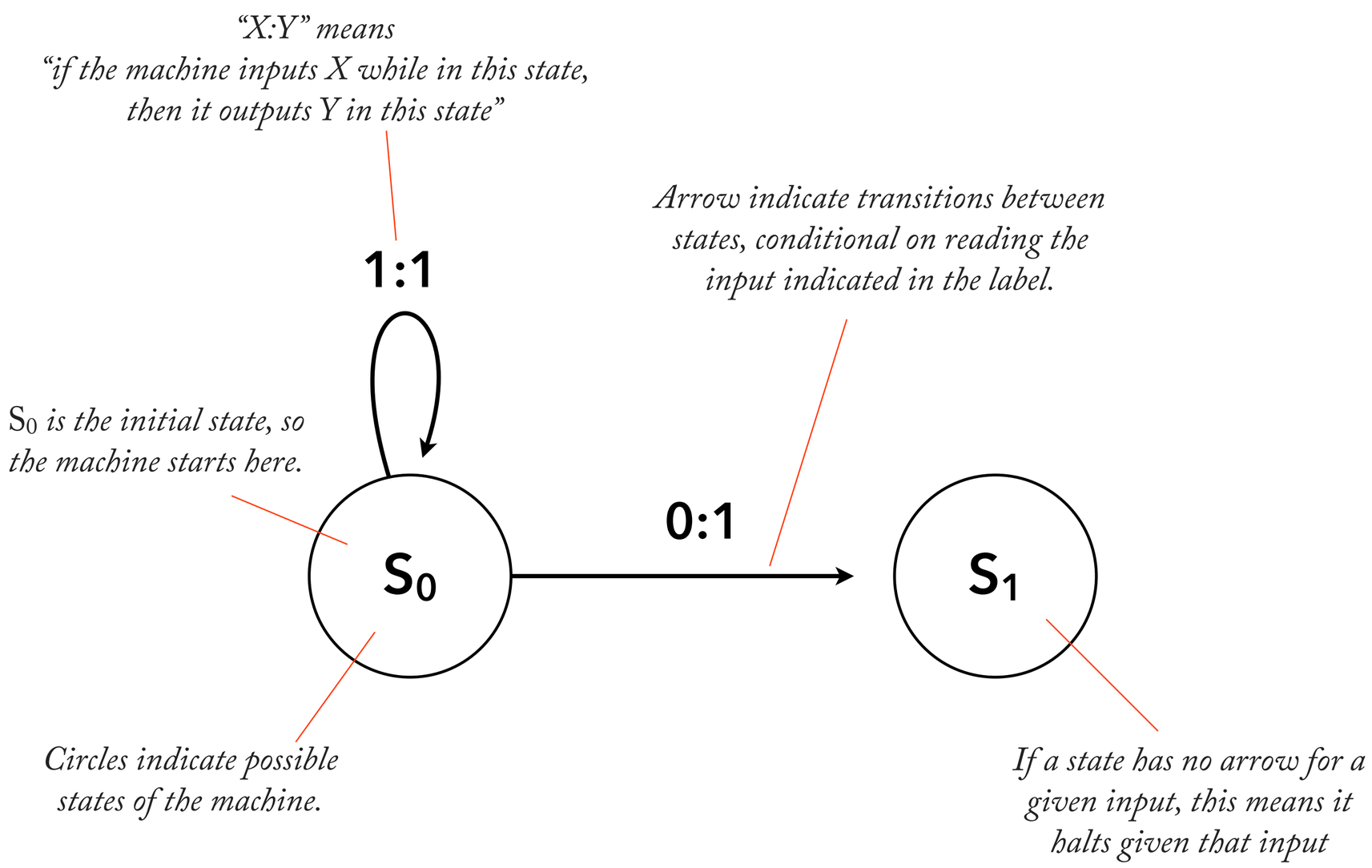


"Anyone who wishes, then, to argue on this basis for the existence of the soul will have to be prepared to hug the souls of Turing machines to his philosophical bosom!" (Putnam 1960)

25. Finite State Machines

Reading FSM Diagrams

You can get a feel for how FSM diagrams work by studying the following simple example:



This diagram represents the following rule:

In state S_0,
if input = 1, then output = 1, and return to S_0;
if input = 0, then output = 1, and move to S_1.

In state S1, halt on any input.

Constructing FSM Diagrams

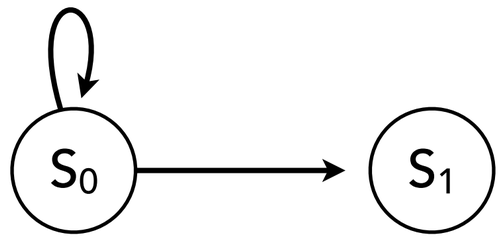
FSM diagrams can be constructed from the following elements.

Nodes: circles associated with names.

- Each node represents a state.
- The initial state is always named "S_0".

Links: one-directional arrows connecting circles to circles.

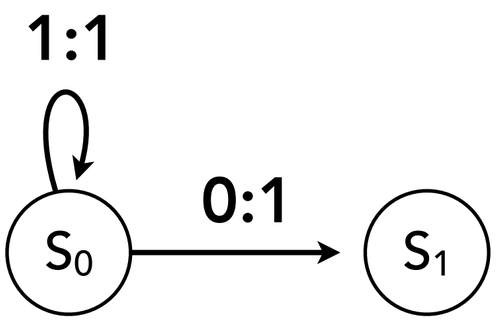
- There are at most two lines leaving each circle.*
- There may be zero lines leaving a circle.
- Circles can be connected to themselves.
- The lines can cross.



*More precisely, there are as many lines leaving each circle as possible inputs. If the machine can read *n* symbols, there will be up to *n* links for each node.

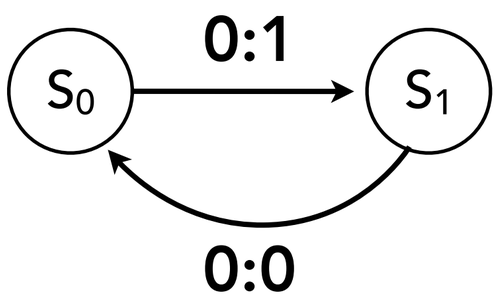
Labels: each line is labeled with with an expression of the form "*input* : *output*".

- Possible inputs: 1, 0
- Possible outputs: 1, 0.
- **Note!** For each circle, there can be at most one arrows labeled with input 1, and one arrows labeled with input 0.
- A common variation writes the output where we've put the state name, and labels the line only with the input.



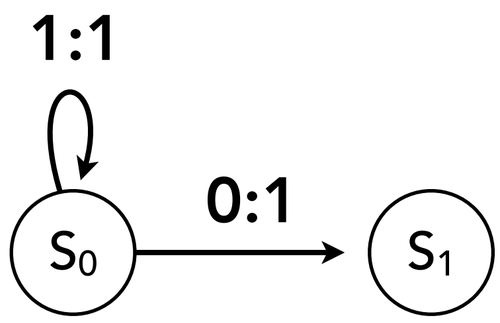
Loops: any kind of loop is acceptable:

- Loops from a state to itself.
- Loops among a groups of states.



Halting:

- A machine halts in a given state, given an input, if there are no arrows leaving that state for that input.
- If there are *no* lines leaving a state, the machine will halt in that state no matter what.



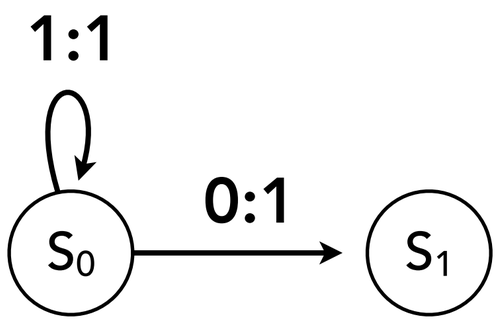
Computing with Finite State Machines

Computation for FSMs is defined in the same way as it is for SCs. Local inputs and outputs are processed one bit at a time, and the global input and output corresponds to the sequences of inputs and outputs which is reached when all further input and output bits will be 0s.

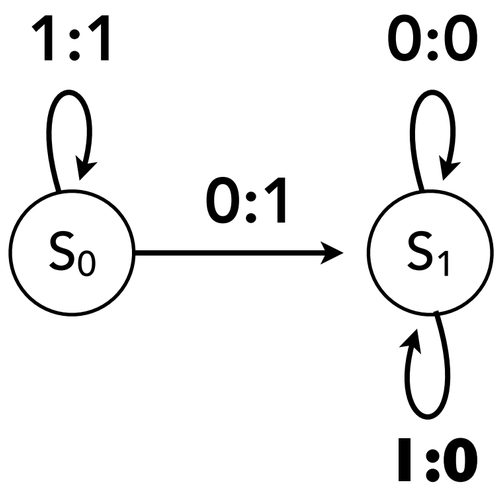
Note on halting:

- A machine **halts** in a given state, given an input, if there are no arrows leaving that state for that input.
- When computing a function, an FSM should never halt. Instead, it should end computation by outputting 0's indefinitely.

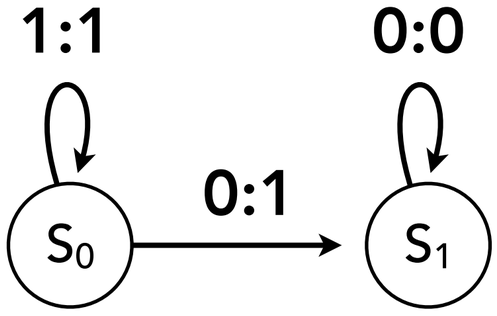
So **don't** design your FSM this way.



Do ensure that the final state produces a continuous output of 0's:



To compute tally successor…

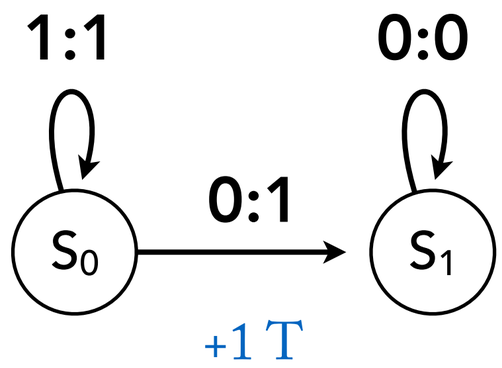


+1 T

Example: +1 T

This FSM at right computes successor in tally.

The timeline below shows the step-by-step progression of the machine for a sample input.

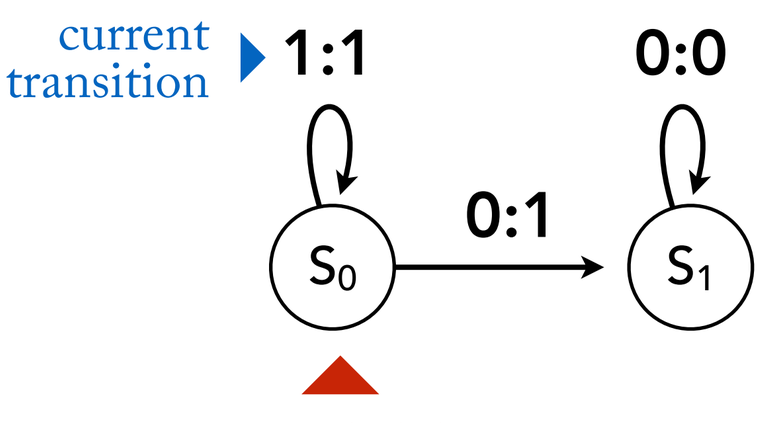


IN: 0 0 0 1 1 — time — OUT: 1

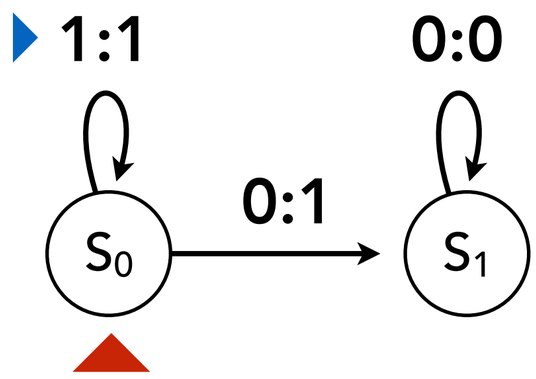
current input

current transition

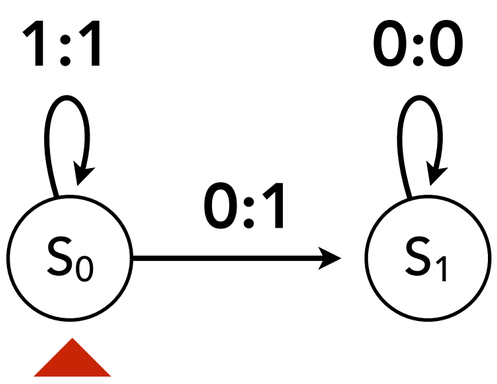
current state



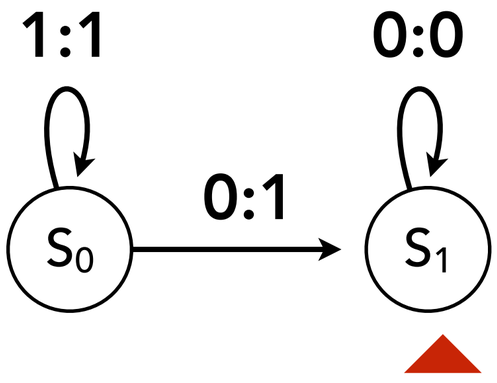
IN: 0 0 0 1 1 — OUT: 1 1



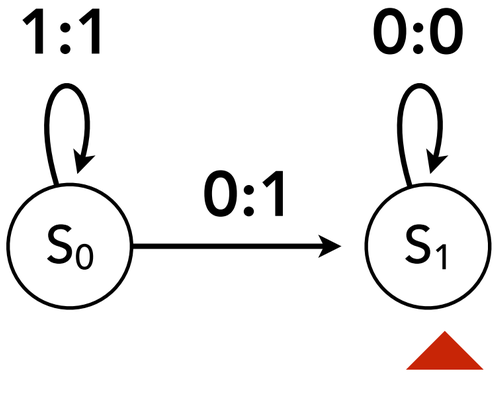
IN: 0 0 0 1 1 — OUT: 1 1 1



IN: 0 0 0 1 1 — OUT: 0 1 1 1



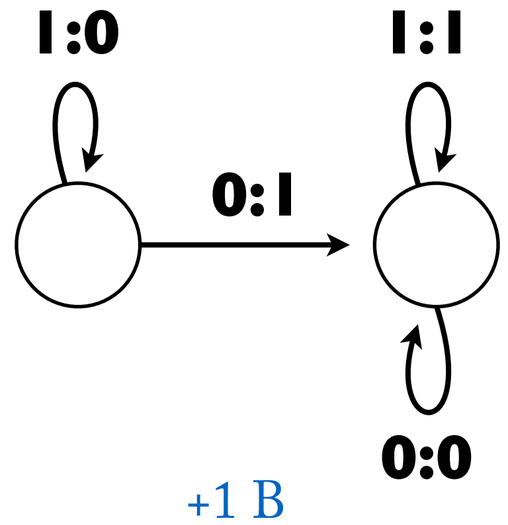
IN: 0 0 0 1 1 — OUT: 0 0 1 1 1



Example: +1 B

This FSM at right computes successor in binary.

The timeline below shows the step-by-step progression of the machine for a sample input.



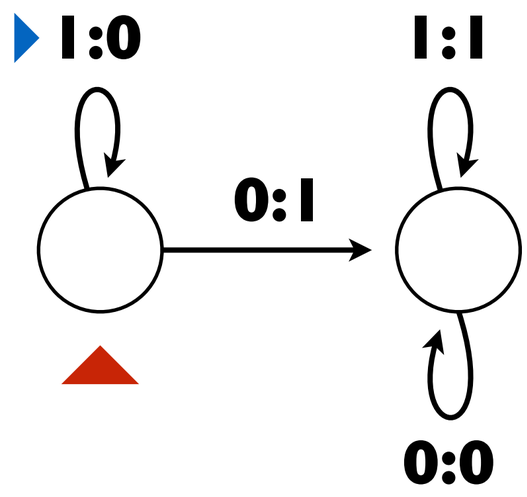
+1 B

IN

0 0 1 1

OUT

0

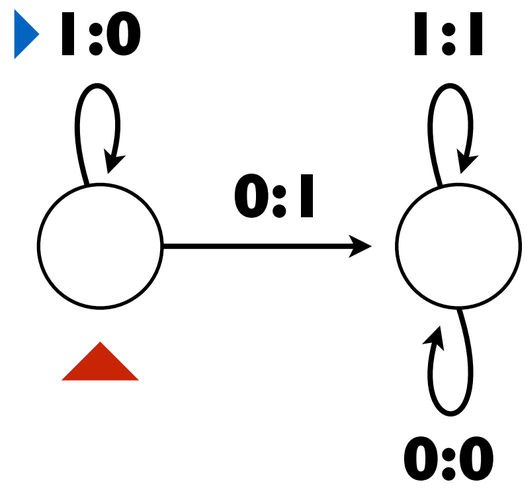


IN

0 0 1 1

OUT

0 0

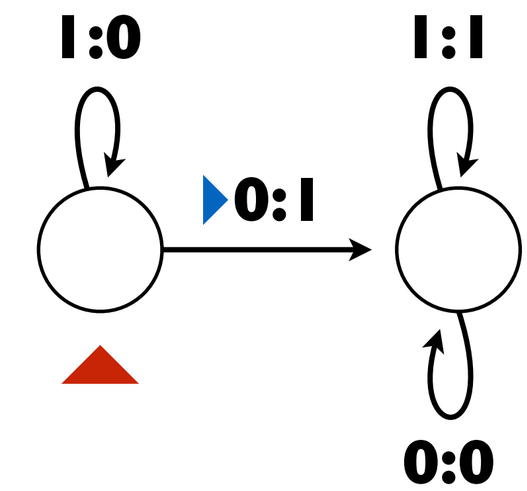


IN

0 0 1 1

OUT

1 0 0

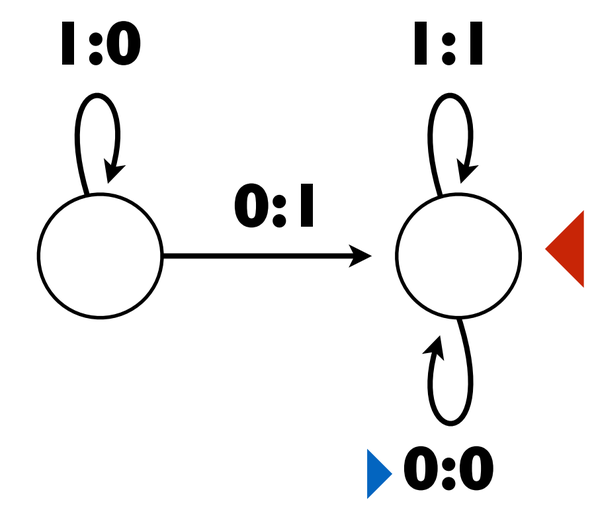


IN

0 0 1 1

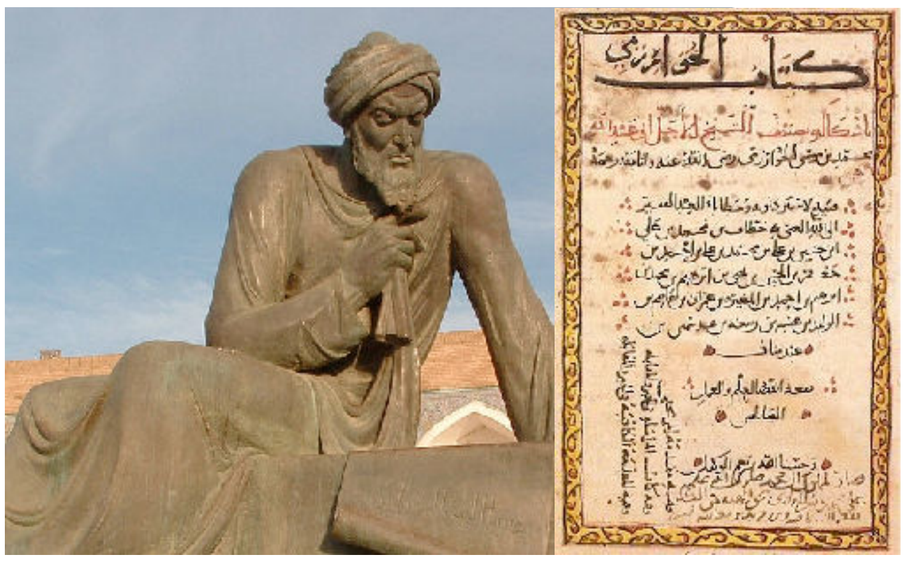
OUT

0 1 0 0



26. What is an Algorithm?

The Algorithm Concept



Muhammad ibn Mūsā al-Khwārizmī (708-850), Baghdad

Ada Lovelace (1815-1852), London

Alan Turing (1912-1954), Wilmslow, England

An **algorithm** is a *procedure* such that:

1. The procedure can be divided into finitely many steps.
2. The transition from one step to another is completely determinate.

A **program** is a *description* of an algorithm in a given computational language. But a machine can **implement** an algorithm in its physical circuitry, without a program. This is the case of finite state machines that are based on sequential circuits.

Steps in the algorithm are the same as states in an FSM. States and state transitions are the basic units out of which algorithms are built.

To do *X*

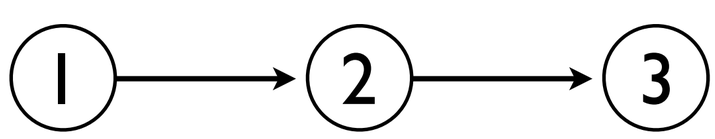
1. [redacted]
2. [redacted]
3. [redacted]

To do *X*

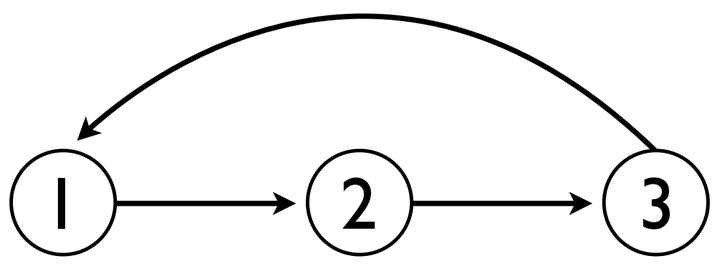
1. [redacted]
2. [redacted]
3. [redacted]

and repeat State 1.

Running an algorithm on a given sequence of inputs involves a certain number of transitions. This is the total number of state transitions that occur until the input is fully processed.



In this case, there are only three states, and only three possible steps.



In this case, there are only three states, but potentially infinite number of transitions.

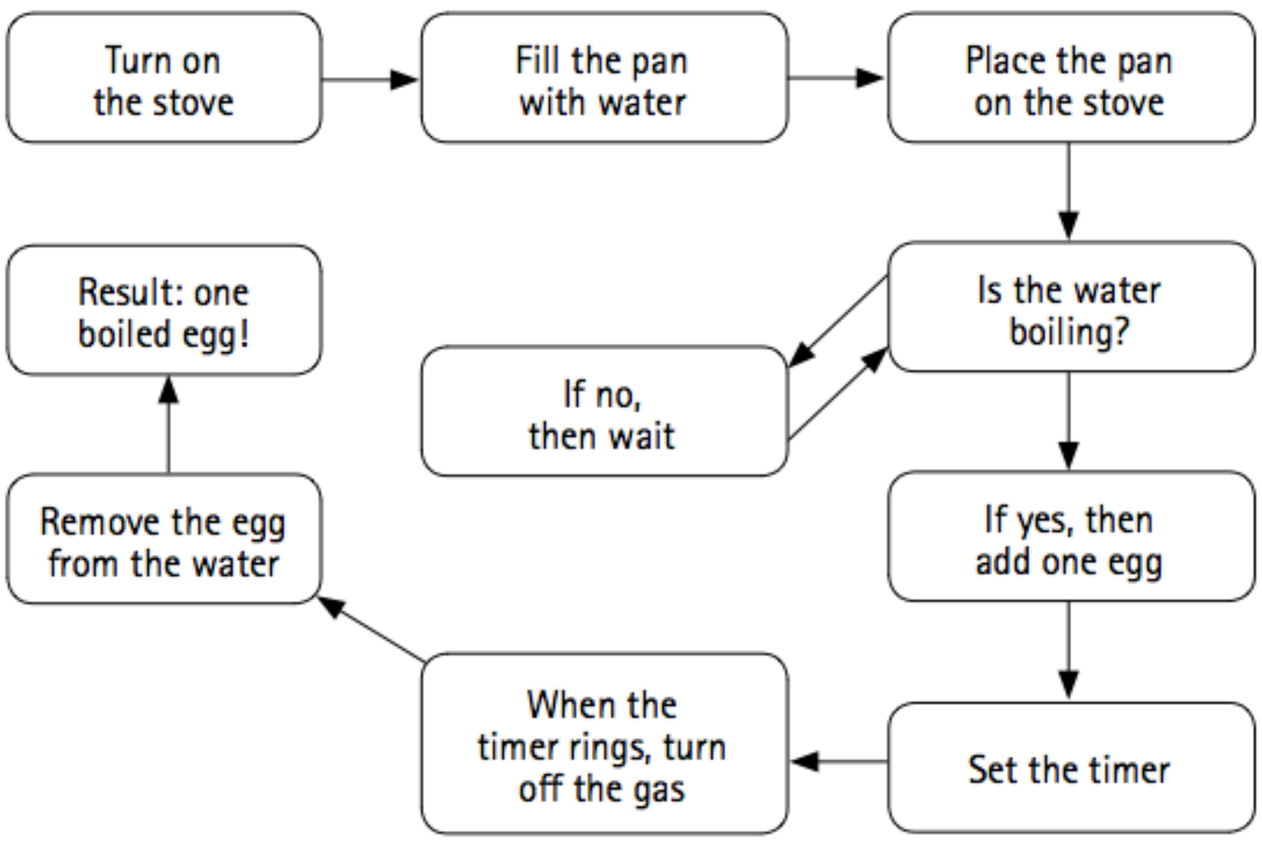
Algorithms and Local Logics

Following an algorithm succeeds as an evolutionary strategy to the extant that it can mirror a relevant process in the world.

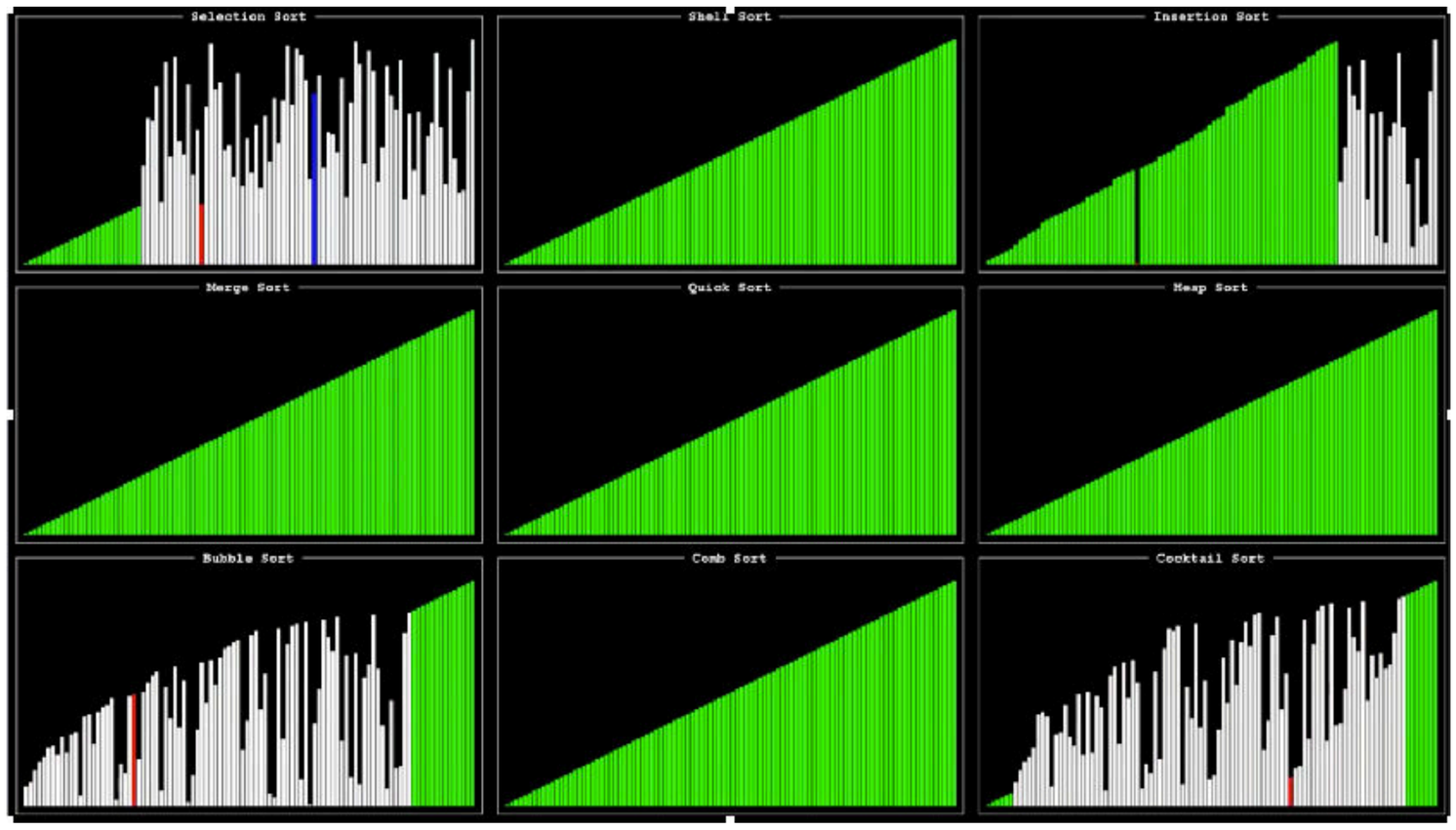
Perhaps an ideal being could simulate everything in the world by implementing algorithms that perfectly implemented the laws of physics. The laws of physics are a **global logic**, a set of rules that describes all situations.

We succeed to the extent that we can identify constrained situations governed by **local logics**. Examples of local logics are the …. laws of psychology, heuristics.

1 Turn on the stove
2 Fill the pan with water
3 Place the pan on the stove
4 When the water boils, add one egg, and set the timer
5 When the timer rings, turn off the gas
6 Remove the egg from the water
7 Result: one boiled egg.



Sorting



Math

How to Do Long Addition

237
+ 116

Step 1
Write the numbers you wish to add, one underneath the other.

Step 2
Add up the numbers in the right-most column.

7 + 6 = 13

Step 3
Check if the answer from the previous Step is 9 or less:
- If Yes, write the number below the column and move to Step 4.
- If No, the answer will have two digits.
  - Write the digit on the right below the column.
  - Write the digit on the left above the column to the left of the current column. This is carrying. Move to Step 4.

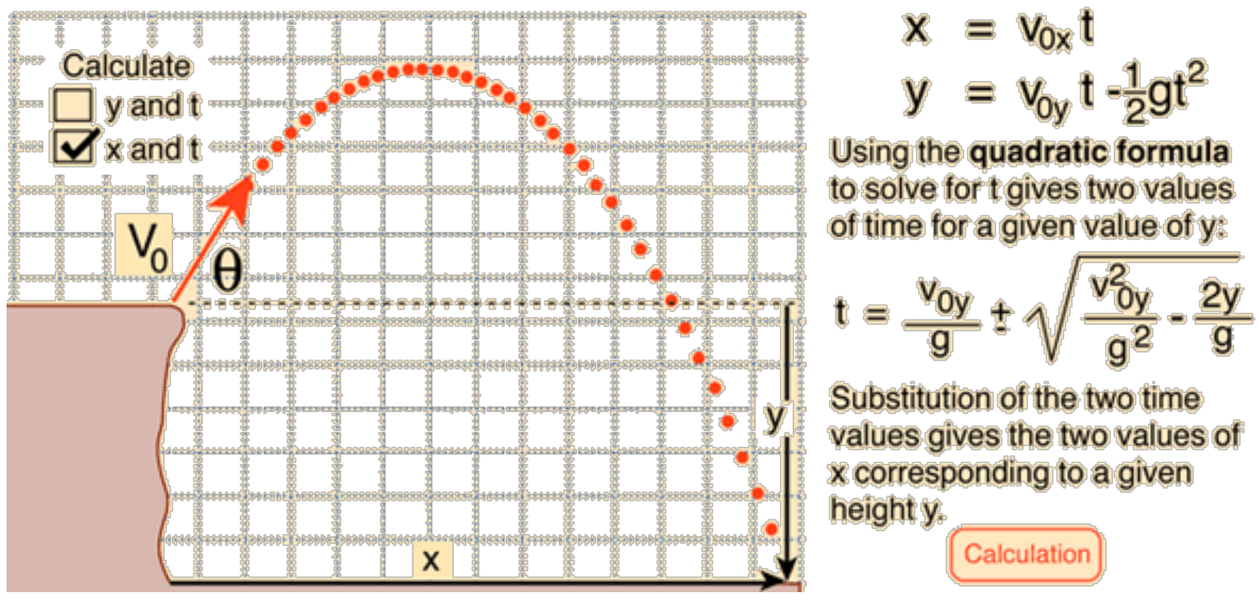
```
   1
 2 3 7
+1 1 6
-----
   3
```

Step 4
Move to the column to the left. Add up the numbers in this column, including any numbers above the two lines.

```
   1
 2 3 7
+1 1 6
-----
 3 5 3
```

Step 5
Go to Step 3 and repeat until all columns have been added.

Physics



$$x = v_{0x} t$$
$$y = v_{0y} t - \frac{1}{2} g t^2$$

Using the **quadratic formula** to solve for t gives two values of time for a given value of y:

$$t = \frac{v_{0y}}{g} \pm \sqrt{\frac{v_{0y}^2}{g^2} - \frac{2y}{g}}$$

Substitution of the two time values gives the two values of x corresponding to a given height y.

Calculation

27. Levels of Abstraction

The Concept of Level

Descriptions of complex systems can be divided into **levels**.

Upper levels are **abstractions** of lower levels.
Lower levels are **realizations** of upper levels.

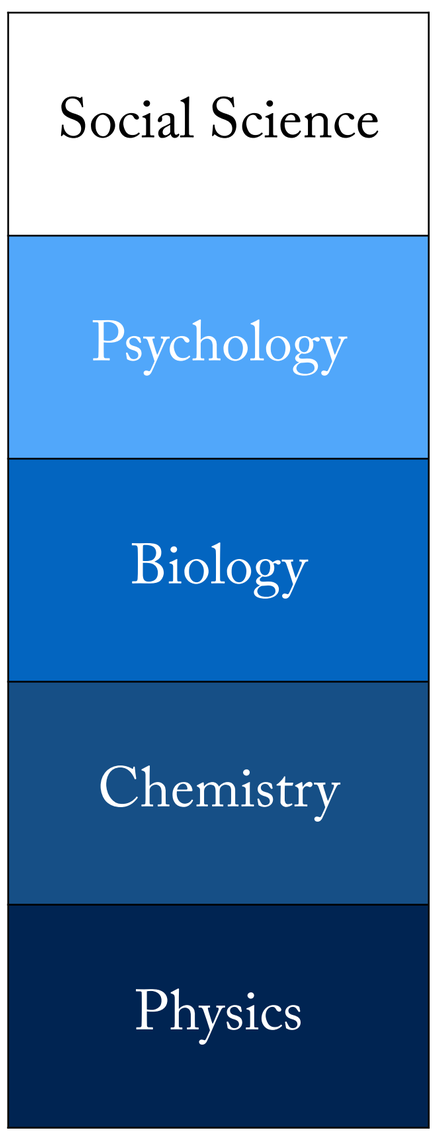
Upper levels capture **real patterns** that emerge from lower levels.

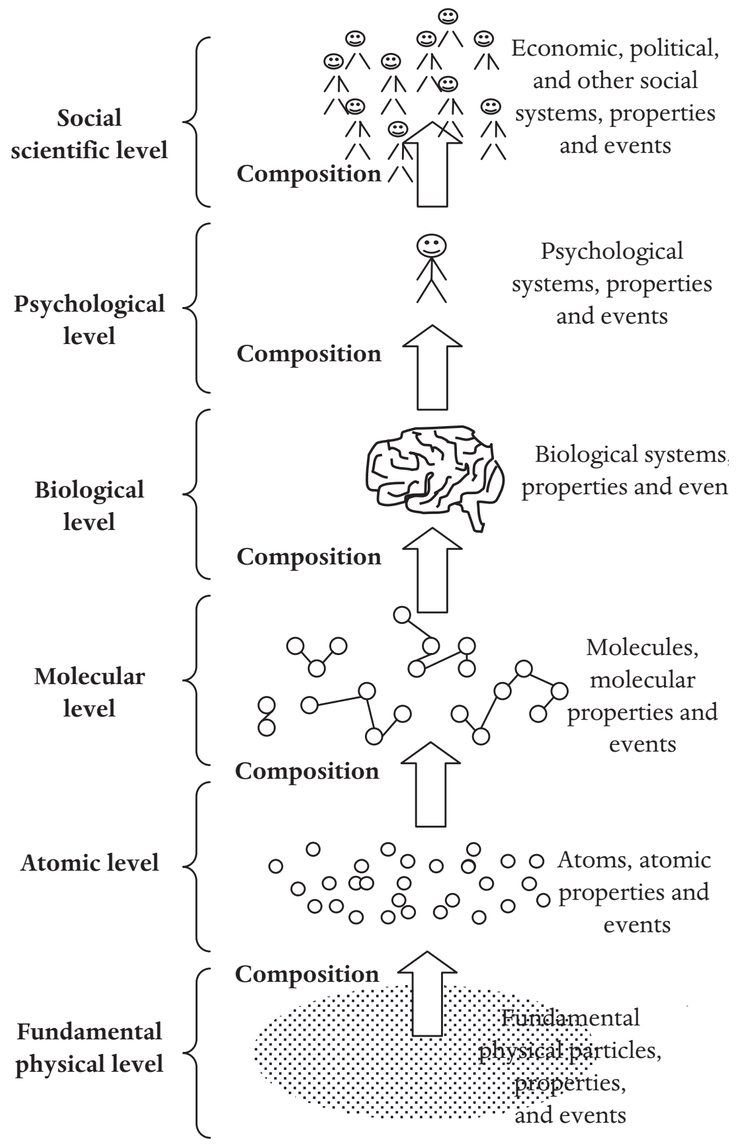
Levels in Nature

The natural sciences as a whole are organized into **levels of abstraction**, also known as **levels of explanation.**

Every level is **necessary**. A complete explanation of reality requires the contributions of all levels, not just the bottom level of physics.

Every level is **autonomous**. Levels of explanation can be studied at least in part without knowledge of the levels above and below.

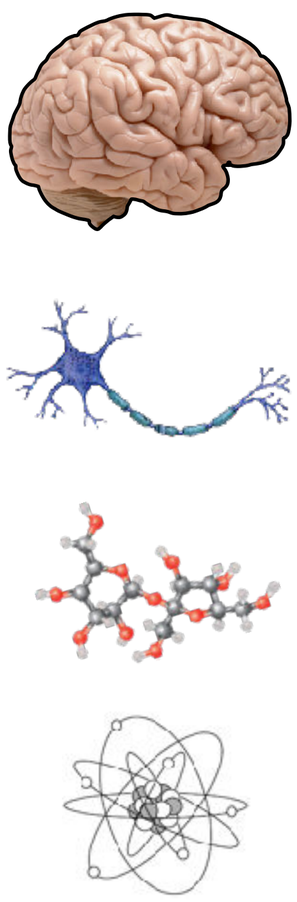




Levels in the Nervous System

The brain itself is a complex system which must be studied at many levels simultaneously.

> "If one hopes to achieve a full understanding of a system as complicated as a nervous system… then one must be prepared to contemplate different kinds of explanation at different levels of description that are linked, at least in principle, into a cohesive whole, even if linking the levels in complete detail is impractical." (Marr 1982)



Neural network

Biology

Chemistry

Physics

Marr's Three Levels

Marr proposed an influential three-level view of cognitive science where each level aims to answer a different kind of explanatory question.

Computational theory	Representation and algorithm	Hardware implementation
What is the goal of the computation, why is it appropriate, and what is the logic of the strategy by which it can be carried out?	How can this computational theory be implemented? In particular, what is the representation for the input and output, and what is the algorithm for the transformation?	How can the representation and algorithm be realized physically?

Figure 1–4. The three levels at which any machine carrying out an information-processing task must be understood.

An example: the humble cash register

Computational Level

"The most abstract [level] is the level of what the device does and why. What it does is arithmetic, so our first task is to master the theory of addition. Addition is a mapping, usually denoted by + from pairs of numbers into single numbers."

Algorithmic Level

"In order that a process shall actually run, however, one has to realize it in some way and therefore choose a representation for the entities that the process manipulates. The second level of the analysis of a process, therefore involves choosing two things: (1) a *representation* for the input and the for the output of the process and (2) a *algorithm* by which the transformation may actually be accomplished… If the first of our levels specifies what and why, this second level specifies *how*. For addition, we much choose Arabic numerals for the representations, and for the algorithm we could follow the usual rules about adding the least significant digits first and "carrying" if the sum exceeds 9.

Hardware Level

"This brings us to the third level, that of the device in which the process is to be realized physically… The child who methodically adds two numbers from right to left, carrying a digit when necessary, may be using the same algorithm that is implemented by the wires and transistors of the cash register in the neighbor-hood supermarket, but the physical realization of the algorithm is quite different in these two cases.

Two features of levels:

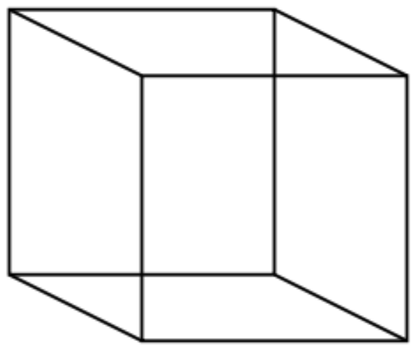
Multiple Realizability: upper levels can be realized in multiple ways at lower levels.

Methodological Independence: one can gain knowledge about different levels more or less independently.

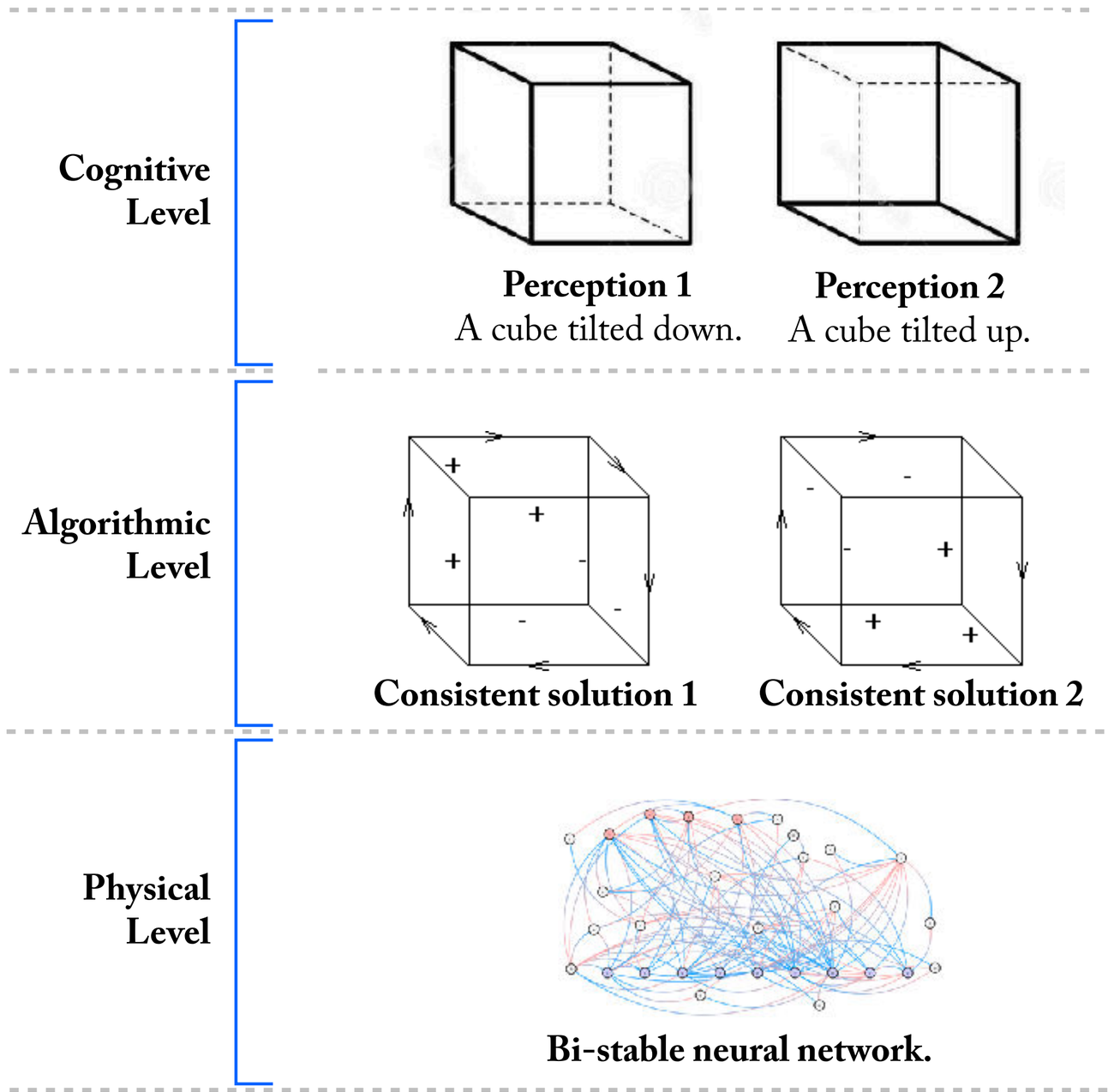
The Cognitive-Ecological Level

Marr called the top level of his hierarchy the "computational" level, with an eye towards *what* is computed. But this is confusing to us, since "computation" evokes algorithms and circuits. Instead, I'll call the top level the **cognitive-ecological level.** It is the level where the **cognitive task** is specified and where the **ecological rationale** for performing that task is explained.

> "It is the top level, the level of computational theory, which is critically important from an information-processing point of view. The reason for this is that the nature of the computations that underlie perception depends more upon the computational problems that have to be solved than upon the particular hardware in which their solutions are implemented. To phrase the matter another way, an algorithm is likely to be understood more readily by understanding the nature of the problem being solved than by examining the mechanism (and the hardware) in which it is embodied."



"The ambiguity of the Necker cube seems to demand a different kind of explanation. To be sure, part of the explanation of its perceptual reversal must have to do with a bistable neural network (that is, one with two distinct stable states) somewhere inside the brain, but few would feel satisfied by an account that failed to mention the existence of two different but perfectly plausible three-dimensional interpretations of this two-dimensional image."



Cognition

Computation

Neuroscience

Physics

The Physicalist Worldview

Marr's "computational" level is the level at which we ascribe content and rationality to an agent. Modern physicalists take this idea as the basis for their theory of the mind: cognition emerges from computation, just as biology emerges from chemistry. Mental states are entirely natural phenomena, just like cells or molecules. The natural world is a unified hierarchy of levels of abstraction.

Levels of Abstraction

Multiple Realizability

Cognitive-Ecological Level

"At one extreme, the top level, is the abstract computational theory of the device, in which the performance of the device is characterized as a mapping from one kind of information to another, the abstract properties of this mapping are defined precisely, and its appropriateness and adequacy for the task at hand are demonstrated."

"If one hopes to achieve a full understanding of a system as complicated as a nervous system… then one must be prepared to contemplate different kinds of explanation at different levels of description that are linked, at least in principle, into a cohesive whole."

+1

Functions

Semantic Abstraction

Algorithmic Level

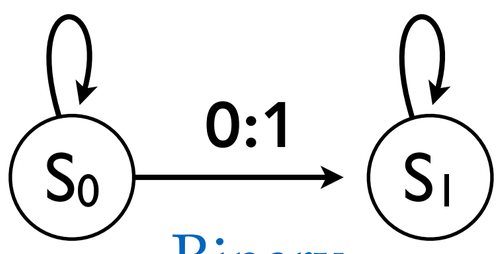
"The second level of analysis of a process, therefore, involves choosing two things: (1) a representation for the input and for the output of the process and (2) an algorithm by which the transformation may actually be accomplished."

1:0

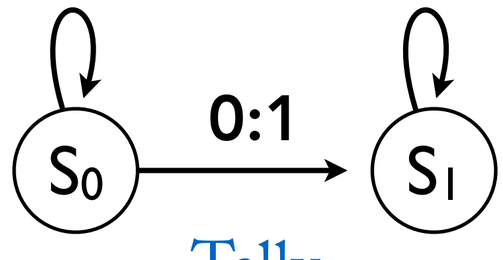
0:0
1:1

1:1

0:0



Binary

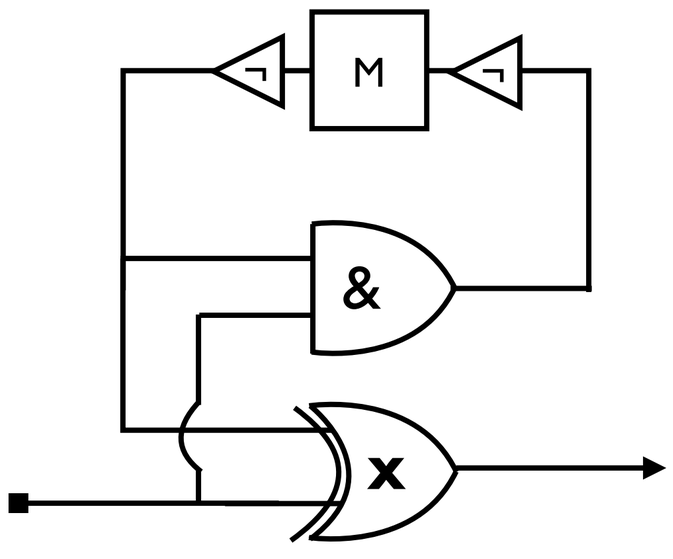


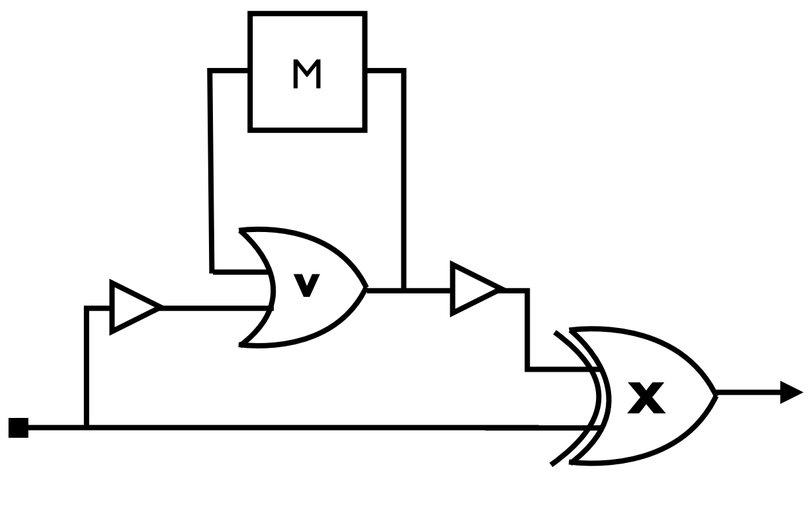
Tally

State Machines

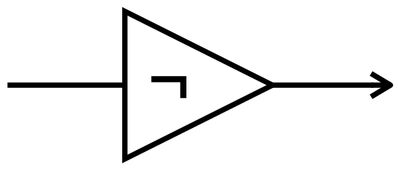
State Abstraction

Mechanical Level





Logic Circuits

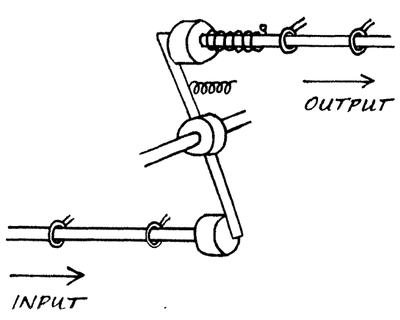


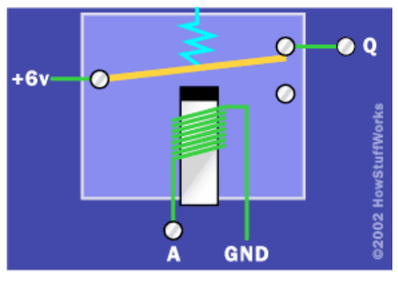
Logic Gates

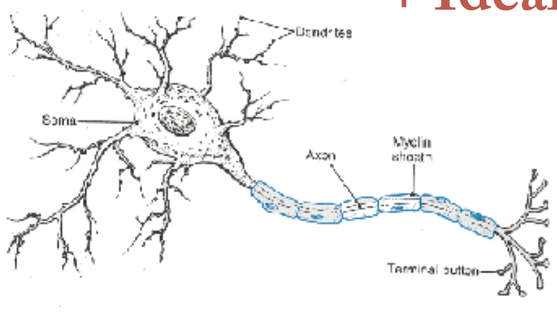
Mechanical Abstraction

+ Idealization

Physical Level





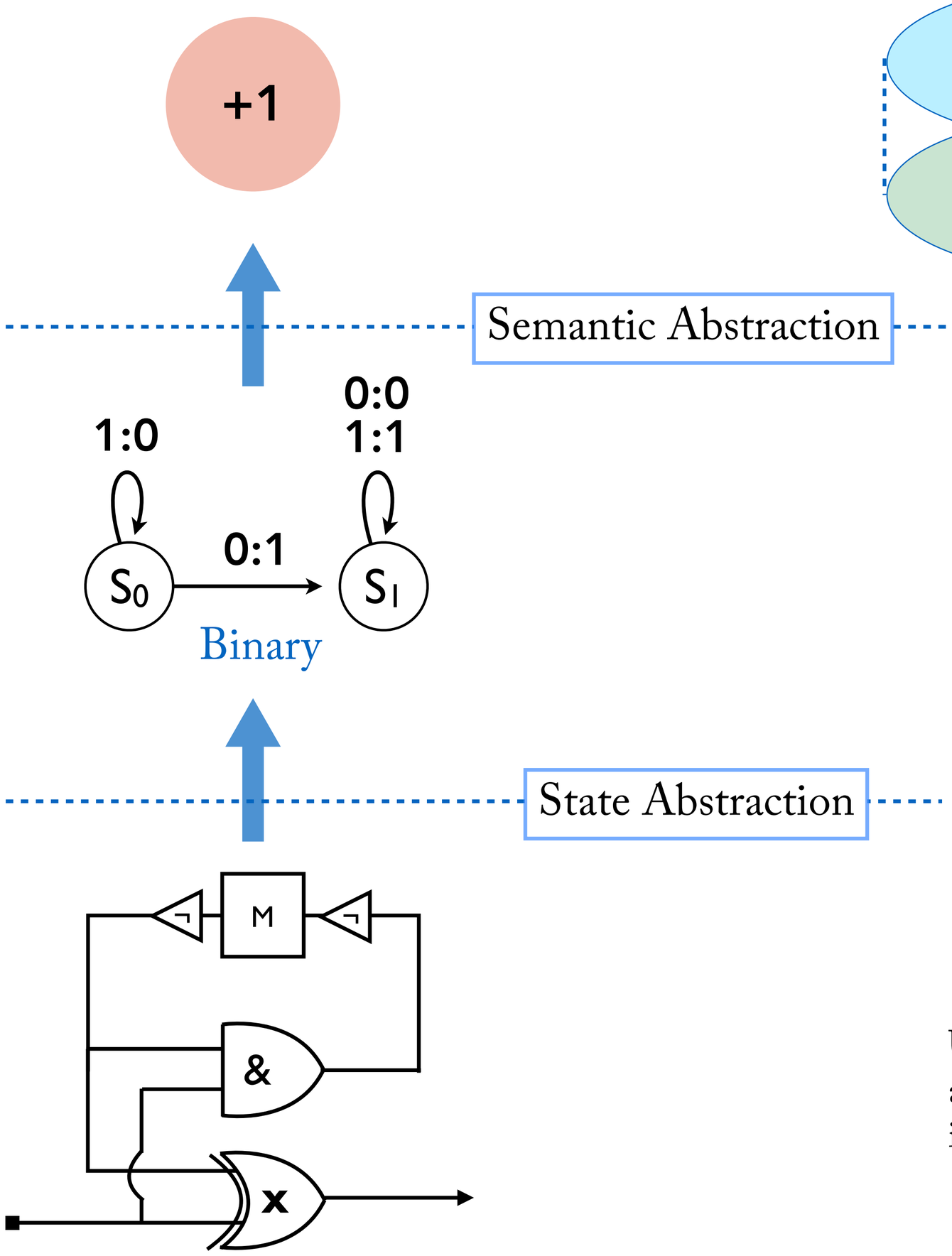


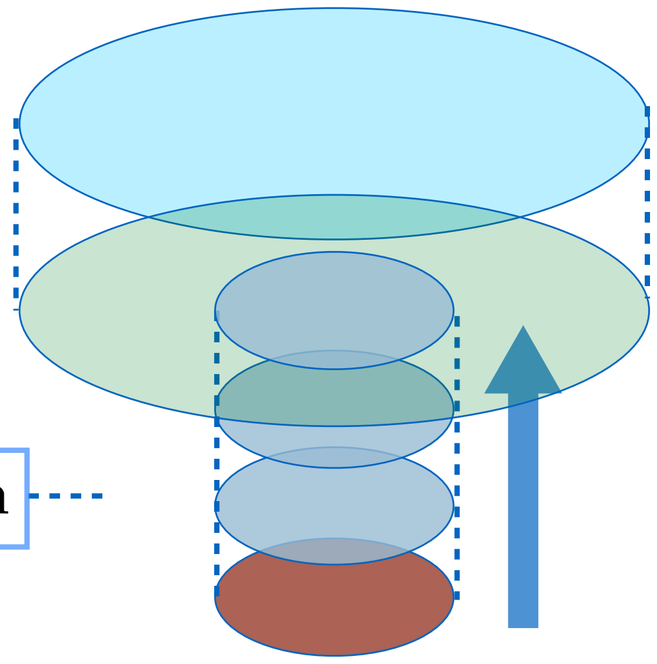
Physical Machines

"How can the representation and algorithm be realized physically?"

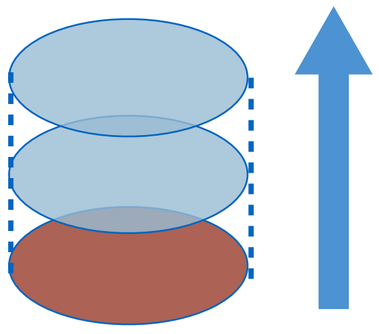
Semantic Abstraction

What are the conditions when we would say an organism computes +1? When it counts some *features of its environment*, e.g. the number of fruits in the tree or the number of steps back to the nest.





Outward abstraction abstraction over the informational relations between a system and the environment.



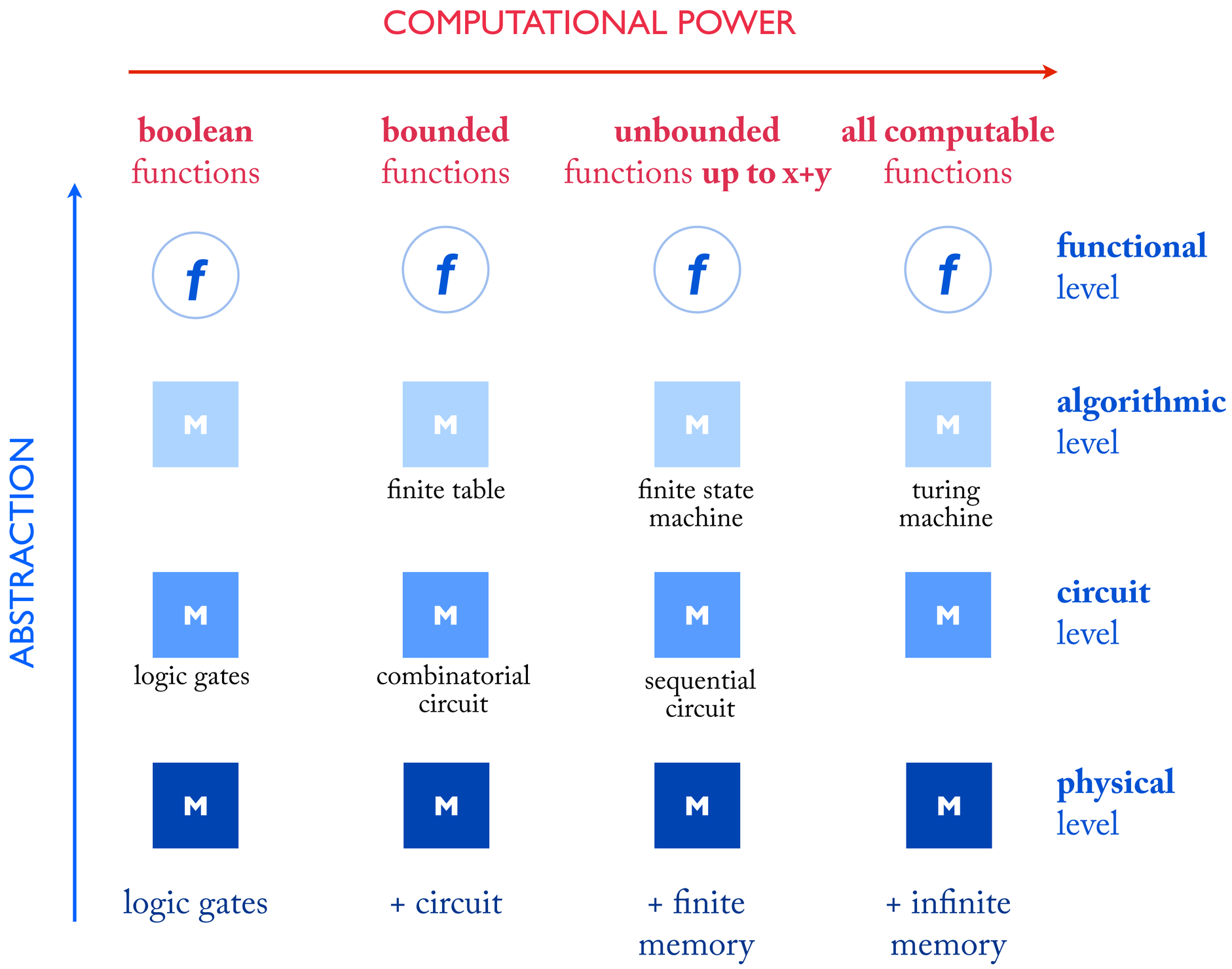
Upward abstraction: abstraction over a system's internal mechanisms.

The central idea here is that semantic description is just another form of functional abstraction. This brings representation and rationality into dialogue with the rest of scientific explanation.

Rational transition as abstraction. To say that a machine computes +1 is to say that, under functional abstraction, it can be characterized in terms of the +1 function. Computation is not like a metaphysical chain between the machine and the function.

Content as abstraction. The same is true for representation. Representation isn't an abstract chain (to be broken or maintained) between a symbol and its content. Content is a way of abstractly characterizing a symbol. We can describe the same physical system as a pattern of electric pulses, as the bit sequence 101, or as the number five. Each belongs to a different level of description.

In Unit 2, we've explored levels of computation: levels of abstraction and levels of computational power.



Unit 3

COGNITION

28. Turing Machines

Turing's Idea

Turing's problem: can all mathematical proofs be described mechanically? What kind of mechanical tools would be needed to generate any given proof?

This problem arose because of Turing's interest in the **Entscheidungsproblem**, or decision problem: the question of whether there is a general algorithm for determining whether any given mathematical claim is provable.

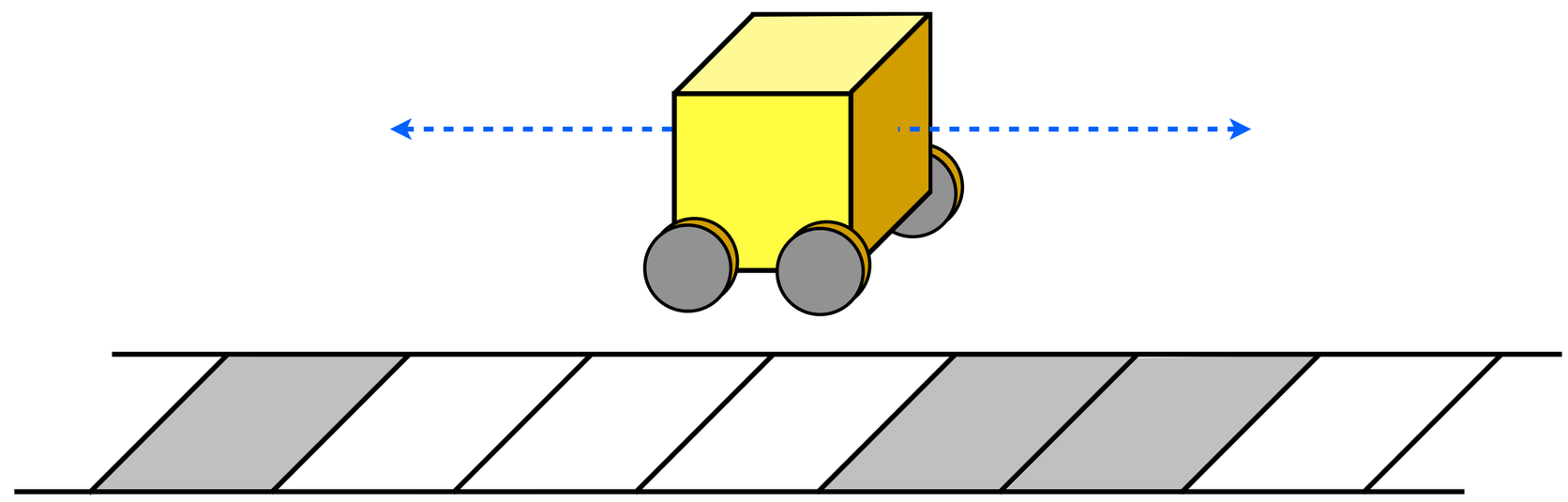
Turing reasoned that the tools you need to solve any given math problem are:

- A piece of paper.
- A stock of mathematical symbols.
- A pencil that writes.
- A human controller.

Now simplify each of these to their simplest mechanical form:

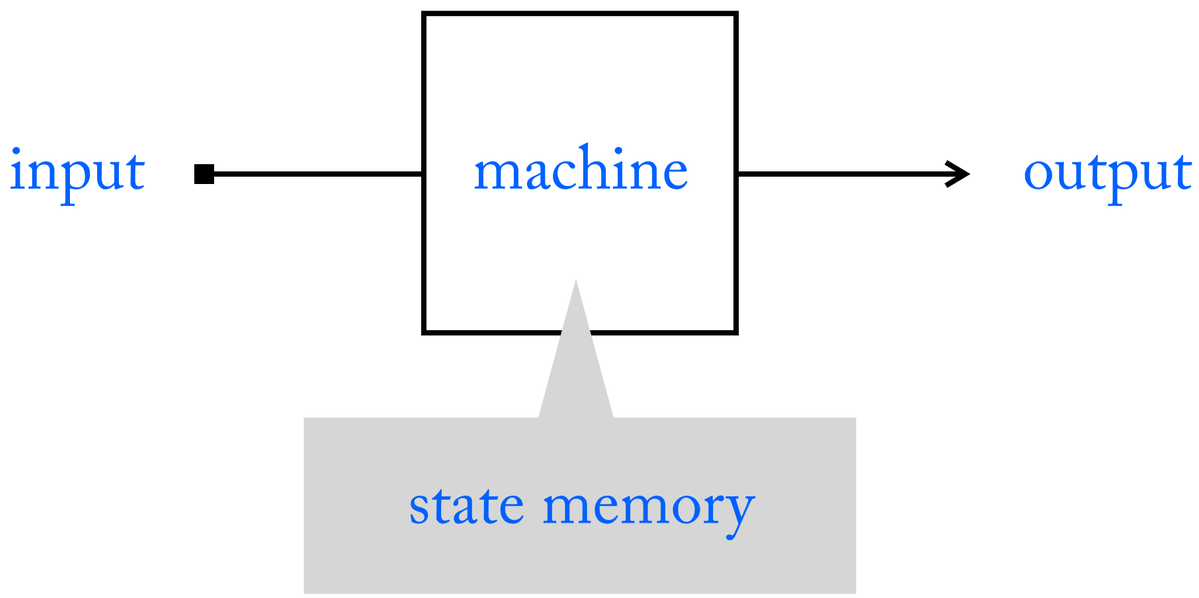
- The paper becomes a 1-dimensional tape.
- The mathematical symbols become 0's and 1's.
- The actions of the pencil become write a 0, write a 1, move left, move right.
- The controller becomes a finite state machine that can read 0's and 1's,

A machine which computes in this way is a Turing Machine (TM).

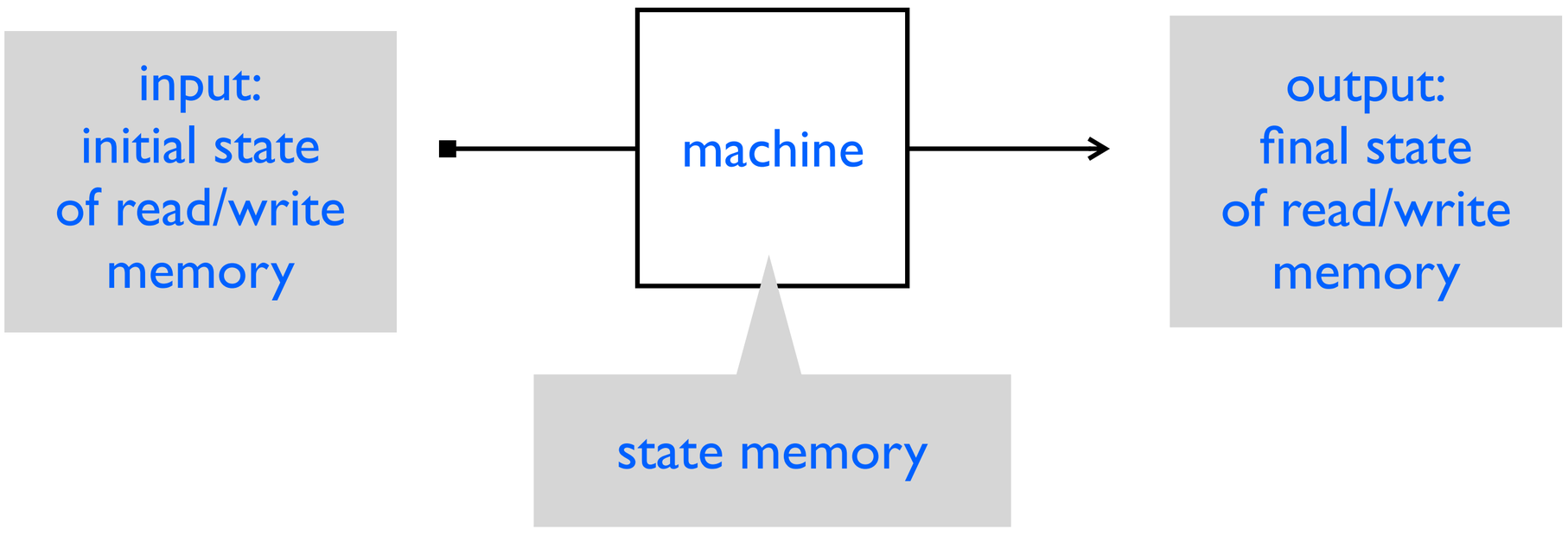


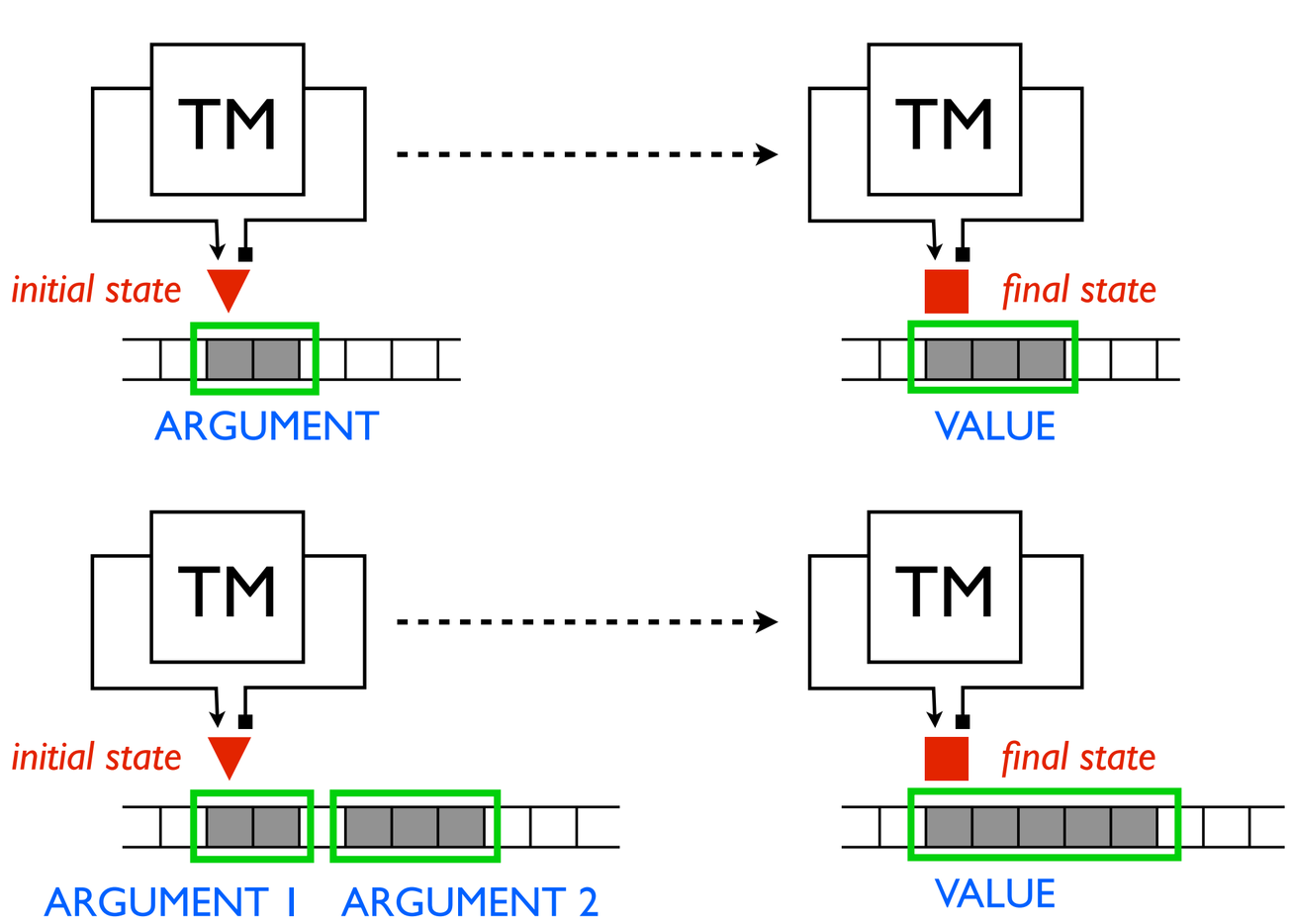
Computing with Memory

FSM: State Memory.
State memory is: (1) short-term; (2) fixed by the architecture of the system.

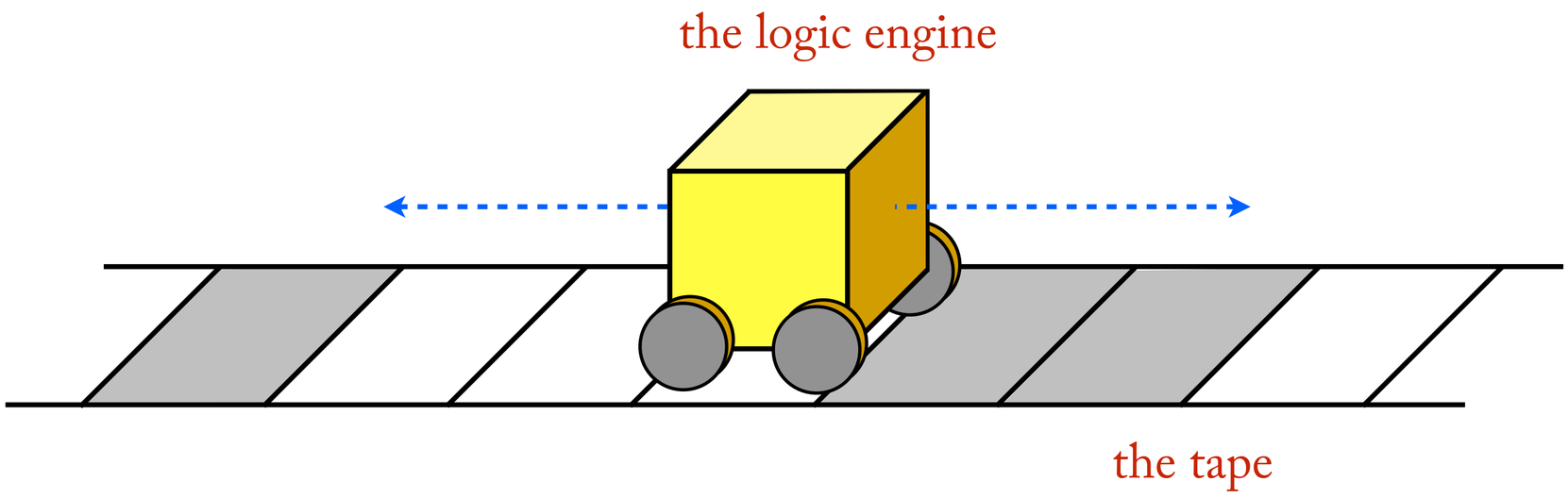


TM: State Memory + Read/Write Memory
Read/write memory is: (1) stable; (2) changeable; (3) extendable within a system.





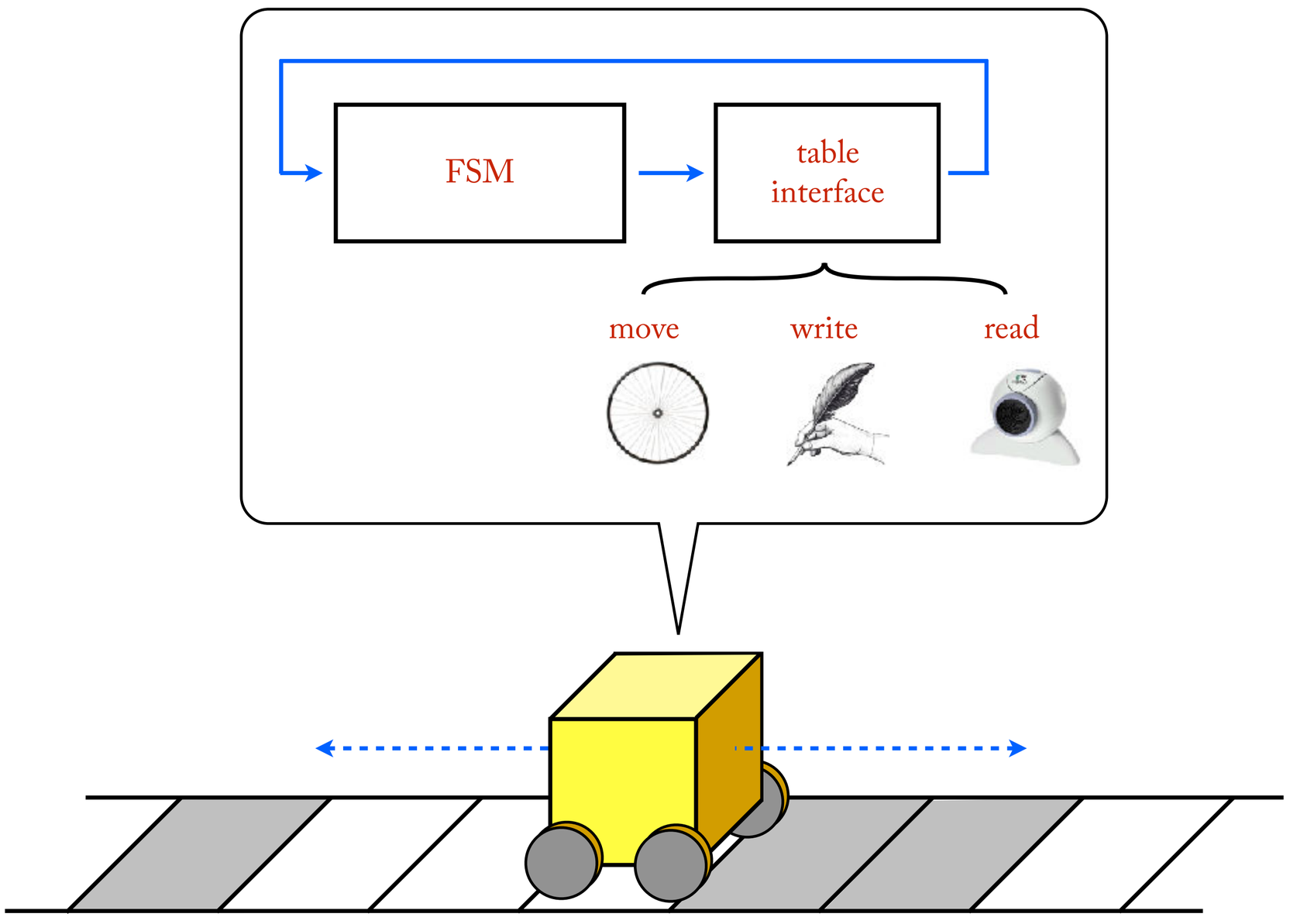
Turing Machine Architecture



A Turing Machine has a two-part architecture:
(i) **A logic engine** (a finite state machine).
(ii) **A read-write memory** (a tape).

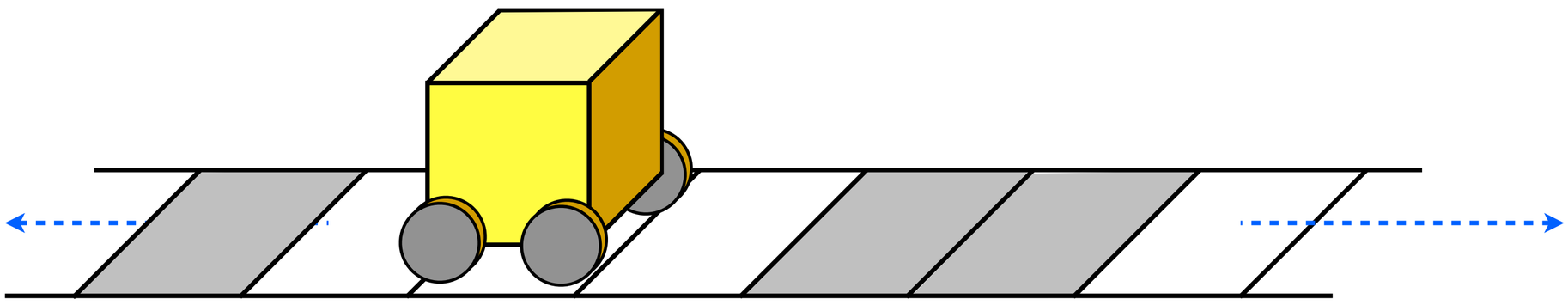
The logic engine is a finite state machine, realized by a sequential circuit. The tape is an infinite strip of graph paper covered with 0's and 1's which the logic engine reads from and writes onto.

In order to work on the tape, the FSM inside the TM is connected to a tape-interface. This interface allows the FSM to move left and right on the tape, write onto it, and read from it. The whole system is connected together in a loop so that the machine calculates, then interacts with the tape, then repeats.



Each reading of the tape becomes the input to the circuit. Each output of the circuit becomes an action of the machine on the tape. The behavior of the machine as a whole depends on both the content of the tape and the content of the memory registers in the sequential circuit.

The Tape



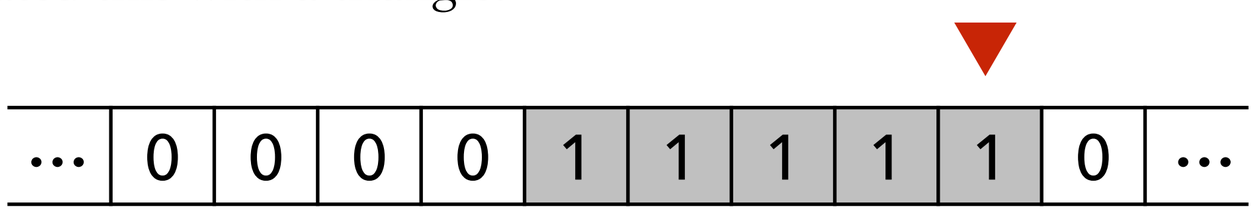
The **turing tape** has the following properties:

- The tape is infinitely long.
- It is divided into cells.
- Every cell is either filled-in (1) or empty (0).
- There are **infinitely** many empty cells (0's) on the tape.
- There are **finitely** many filled-in cells (1's) on the tape.

Machine-Tape Interface

At any given time, the machine occupies exactly one cell of the tape. This is called the **position** of the machine at that time.

We'll indicated this with a triangle:



The machine's abilities are very limited. It can only perform "1-cell" operations: it can only write on, move across, or read one cell at a time.

Actions

The machine can only perform 1-cell actions:

1. **R** = move right one cell.
2. **L** = move left one cell.
3. **1** = make the current cell filled-in.
4. **0** = make the current cell empty.
5. **Halt**

Perceptions

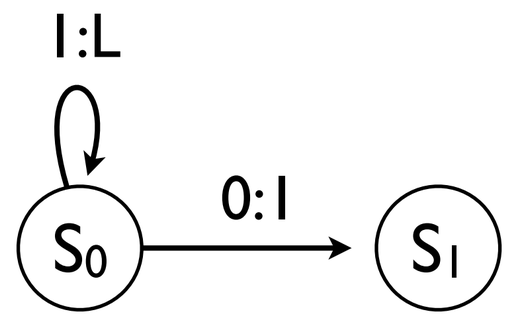
The machine can read only two 1-cell states of the tape:

1. **1** = the current cell is filled-in.
2. **0** = the current cell is empty.

Turing Machine Diagrams

TM diagrams are just like FSM diagrams, where the possible inputs are (for now) 1 or 0, and the possible outputs are R, L, 1, or 0.

This FSM can be realized by a sequential circuit with one input, and several outputs which control a motor.



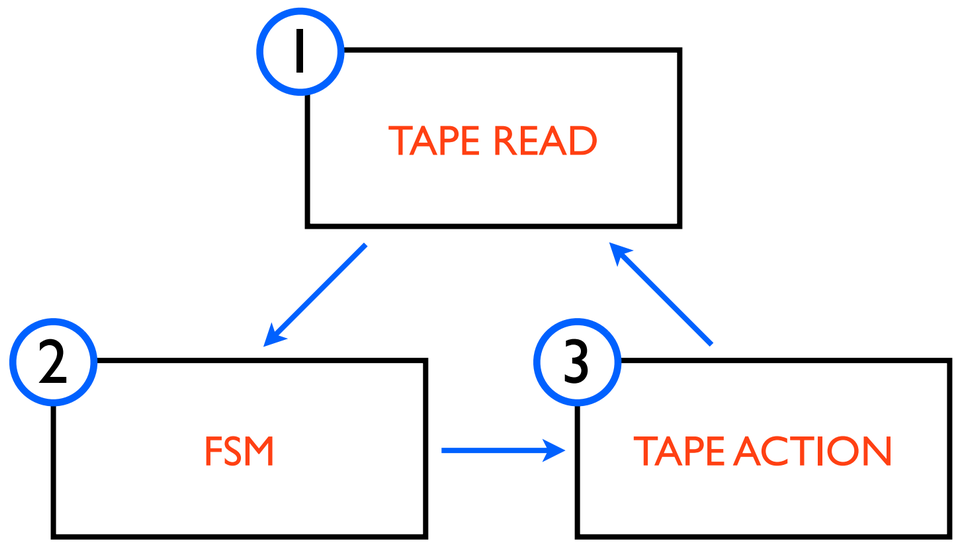
This machine above reads 1's and moves left, until it reads a 0, then writes a 1.

Operation in Time

Order of operations

At each time cycle, the machine undergoes a sequence of actions:

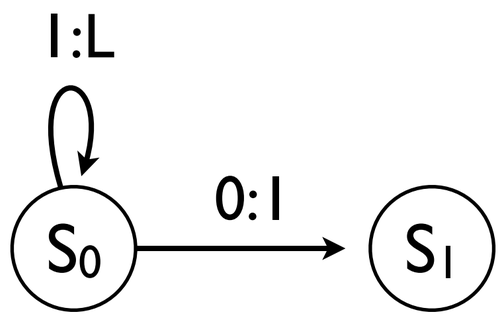
1. Read tape
2. Run circuit
3. Execute tape action (or halt)
4. Repeat



Halting:

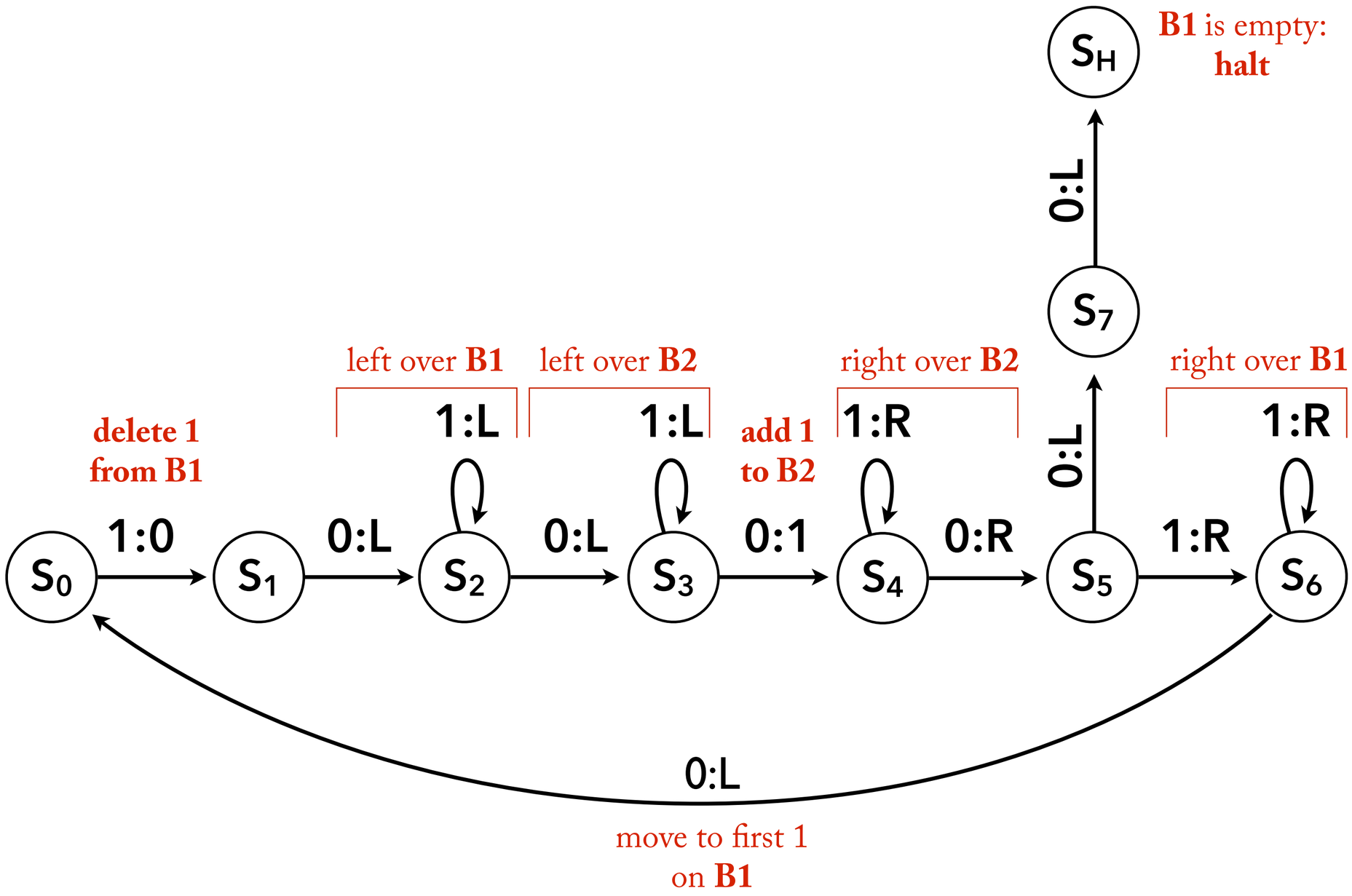
When a machine halts, computation ends. There is no going back.

- Machines do not have to halt. (They can go on forever.)
- A machine halts in a given state, if it is given an input and there are no arrows leaving that state for that input.
- If there are no lines leaving a state, the machine will halt in that state.

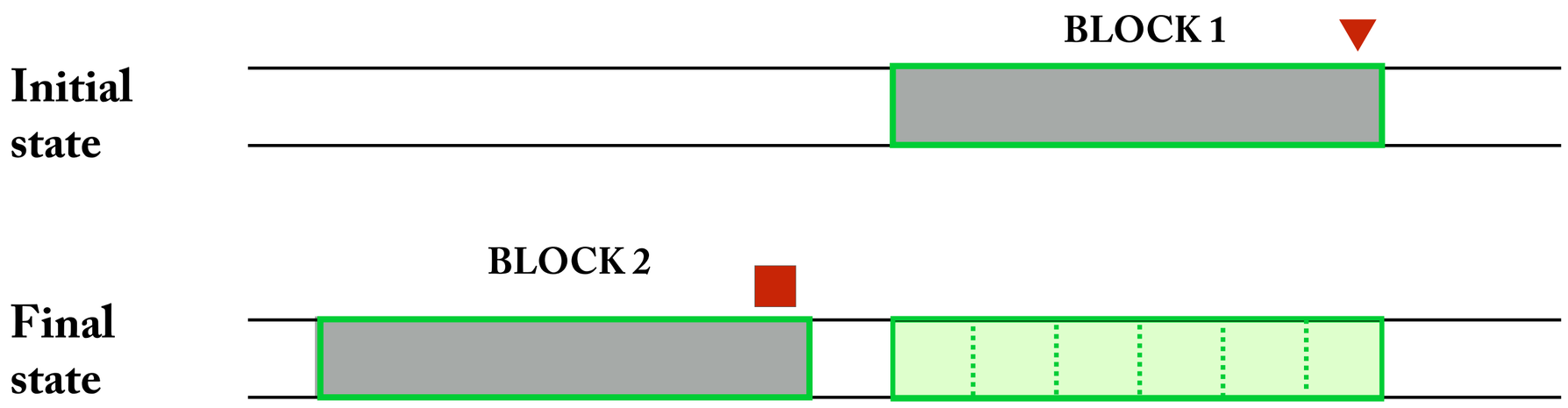


This machine halts in State 1.

Example: Cut and Paste



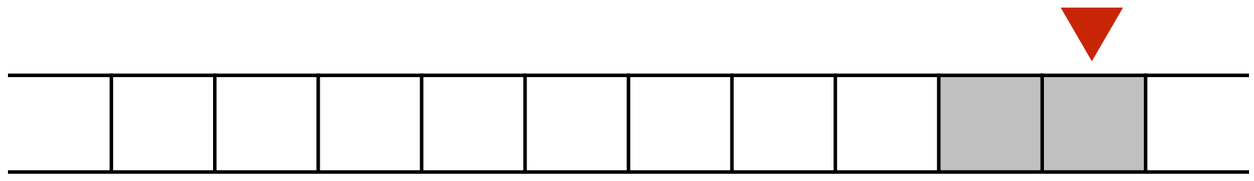
The goal is to shift a block of 1's from the Block 1 position to the Block 2 position, where Block 2 is to the left of the original Block 1 position, separated by 1 cell.



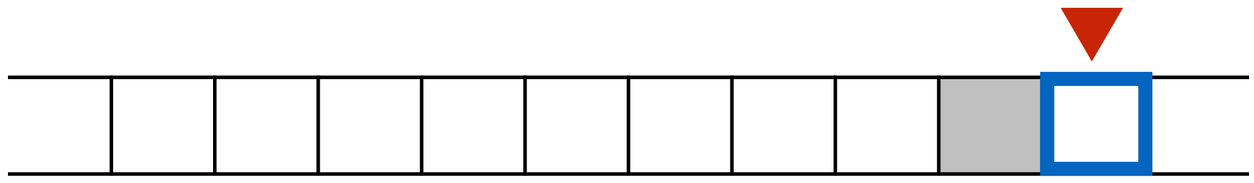
Cut-and-paste:

a) Remove the first 1 at the right edge of B1.
b) Move to the left edge of B1, then move two cells to the left.
c) Skip left over any 1's in B2 (initially there are none), and add a 1 at the end of B2.
d) Go back to B1.
e) Check if it is empty.
   If it is empty, move to B2 and halt.
   If it is not empty, go to the right edge of B1 and return to (a).

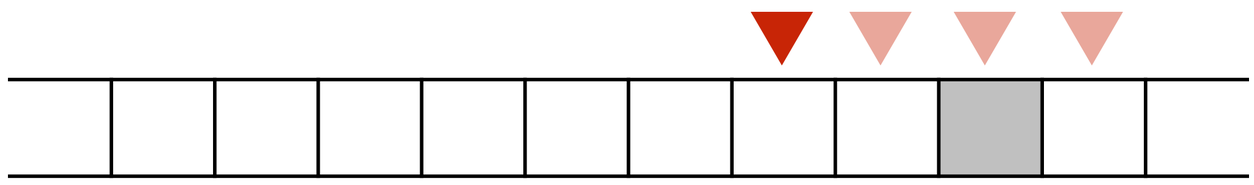
Standard position.



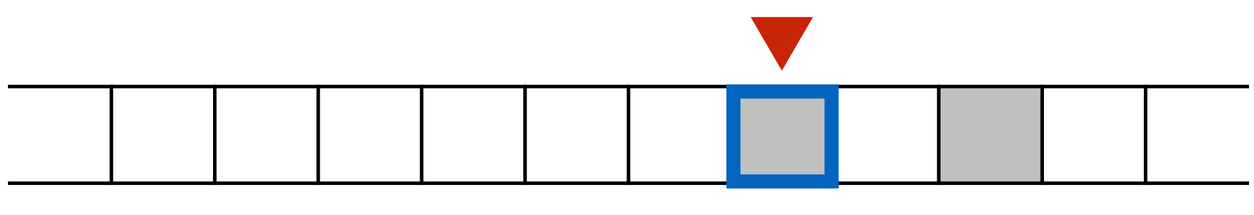
Delete 1 from B1.



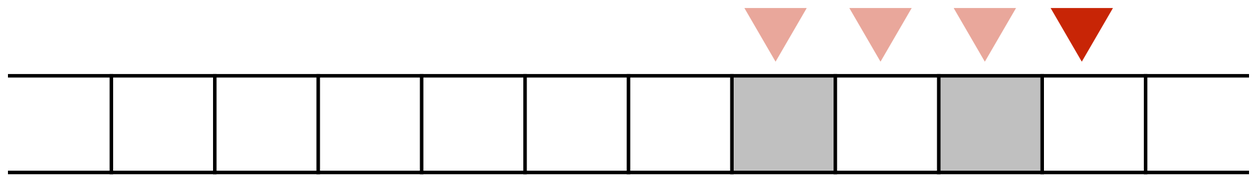
Move left to B2.



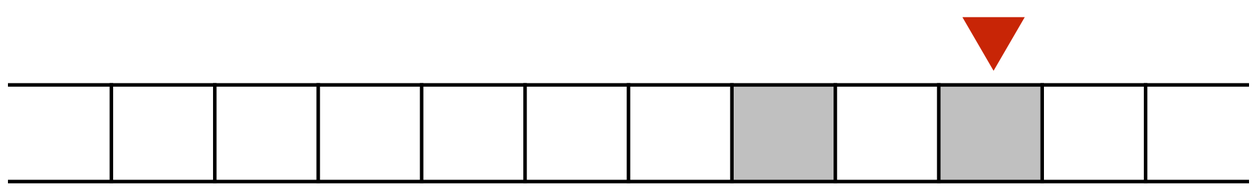
Add 1 to B2.



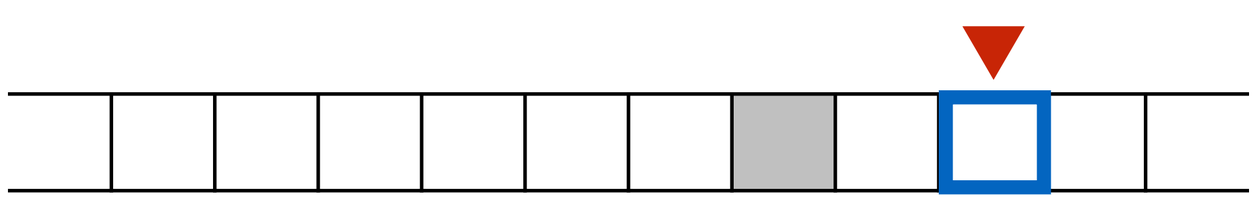
Move right to B1.



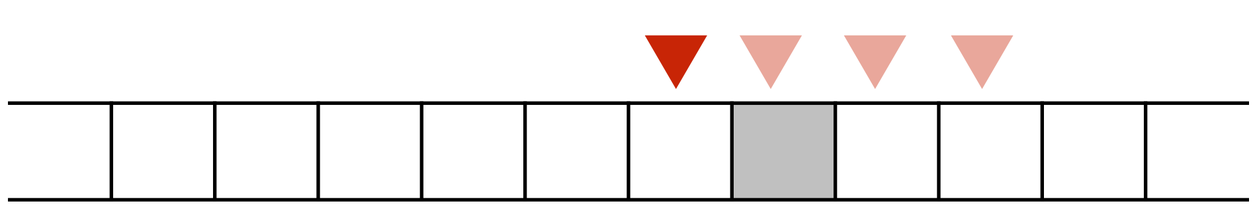
Move to first 1 of B1.



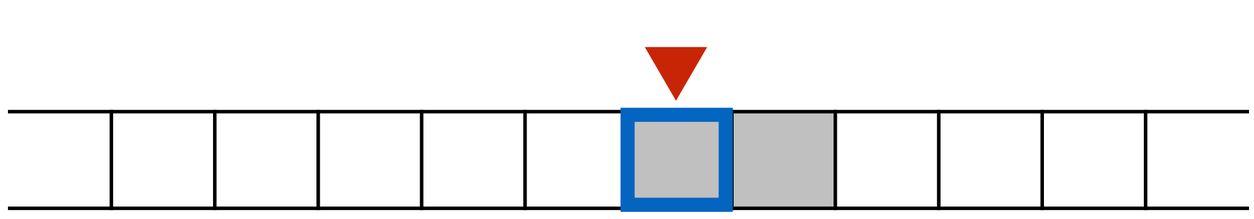
Delete 1 from B1.



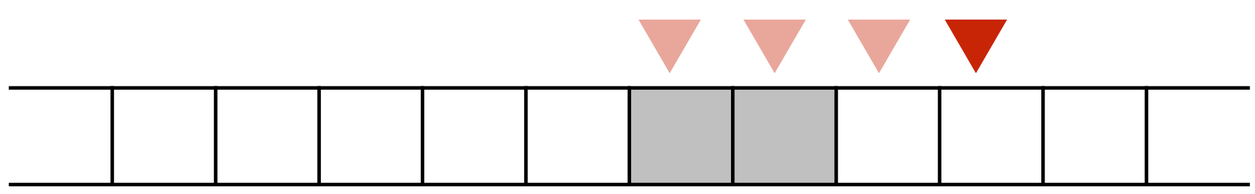
Move left to B2.



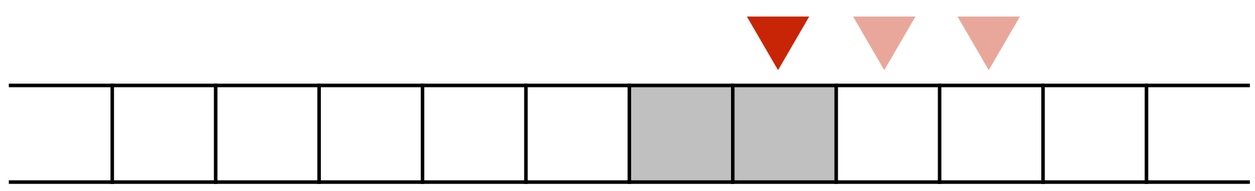
Add 1 to B2.



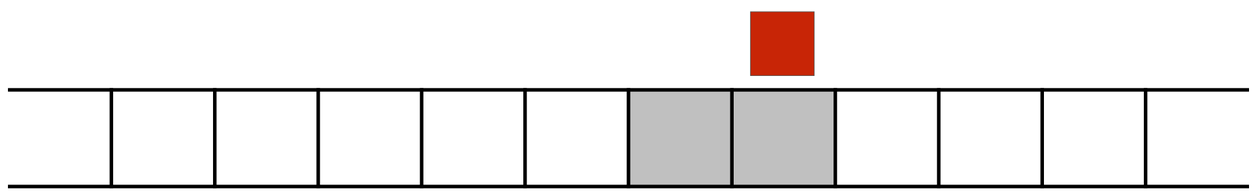
Move right to B1.



If B1 is empty, move to B2.



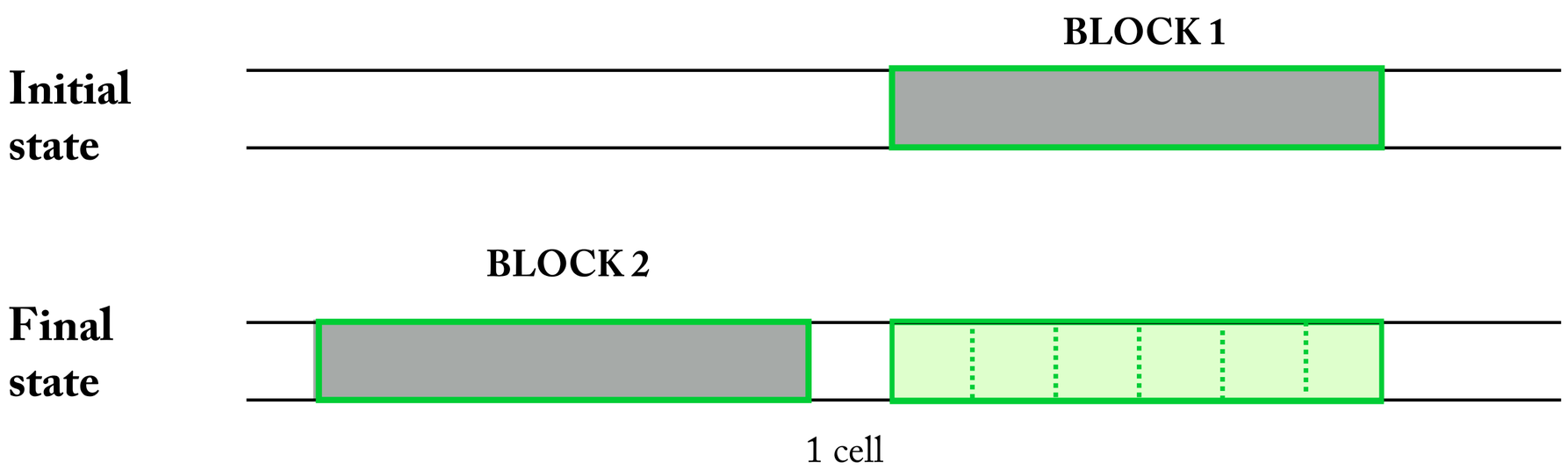
Halt.



In-Class Problems

Start on the right most cell (the "right edge") of the block in all problems.

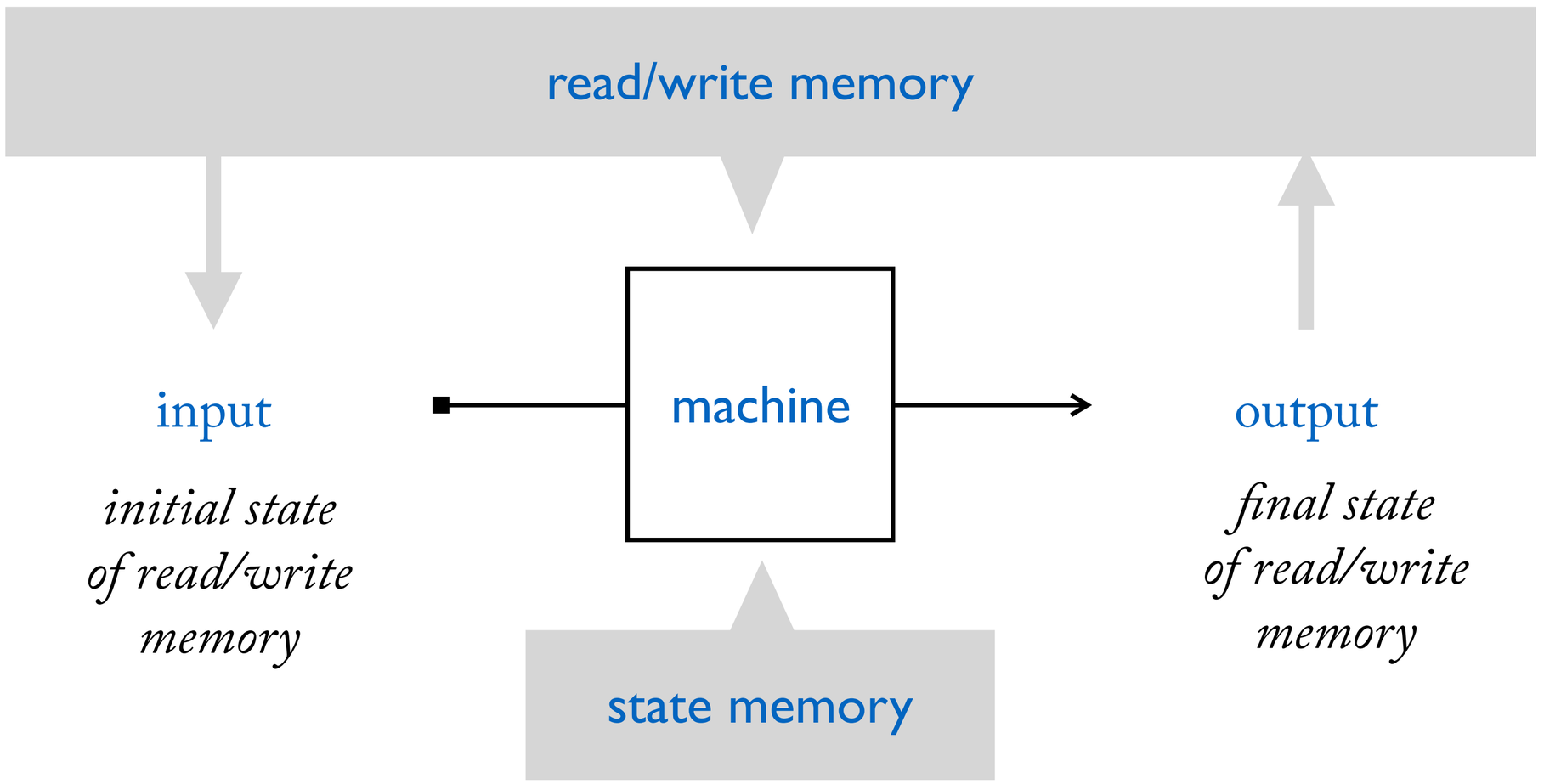
1. **To the left gap:** Move left over a block of 1's, and halt on the first blank space to the left.
2. **To the left gap and back:** Move left over a block of 1's until you reach the gap, then return to the starting point and halt.
3. **To the left gap and +1:** Move left over a block of 1's, add a 1 at the end, then halt .
4. **To the left gap, +1, and back:** Move left over a block of 1's, add a 1 at the end, then return to the starting point and halt.
5. **Infinite expander:** Add a 1 at the left end of a block, then add a 1 at the right end of a block, and repeat forever.
6. **Cut-and-paste:** shift a block of 1's from the Block 1 position to the Block 2 position, where Block 2 is to the left of the original Block 1 position, separated by 1 cell.



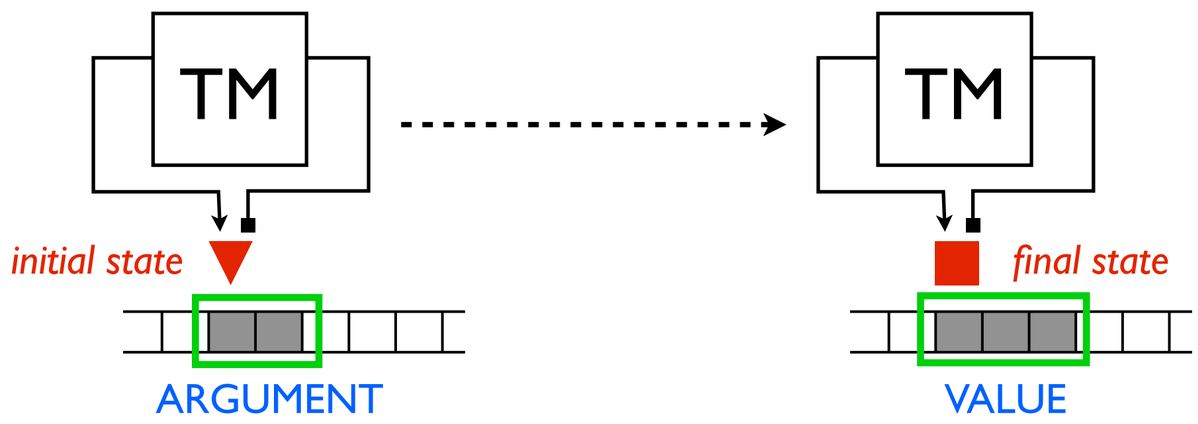
29. Computing with Turing Machines

Computing with Read/Write Memory

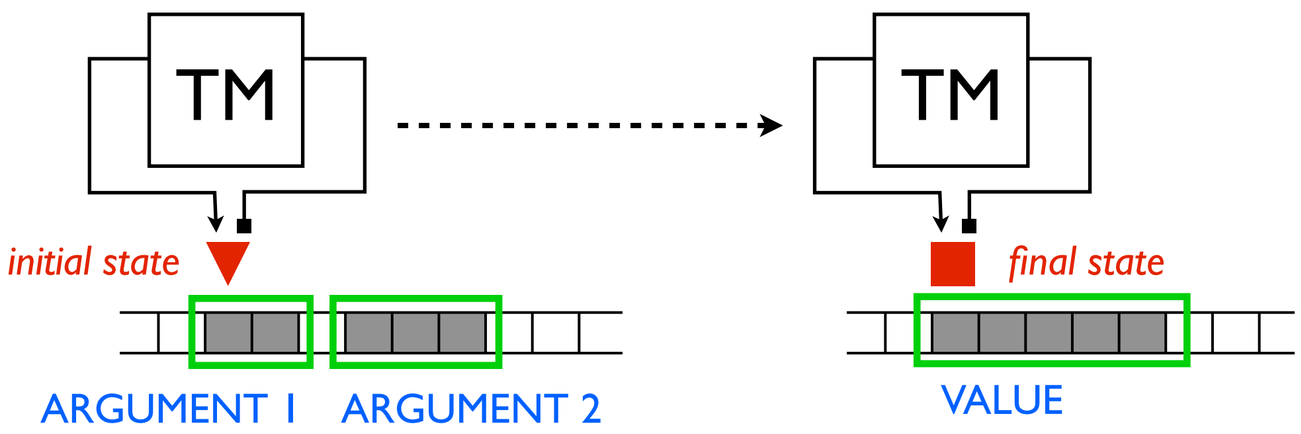
For TMs, the read/write memory serves a dual role. It functions as the input and output of the machine. But it also functions as "scratch paper" that the machine uses in the course of executing its algorithm.



Computing a one place function:

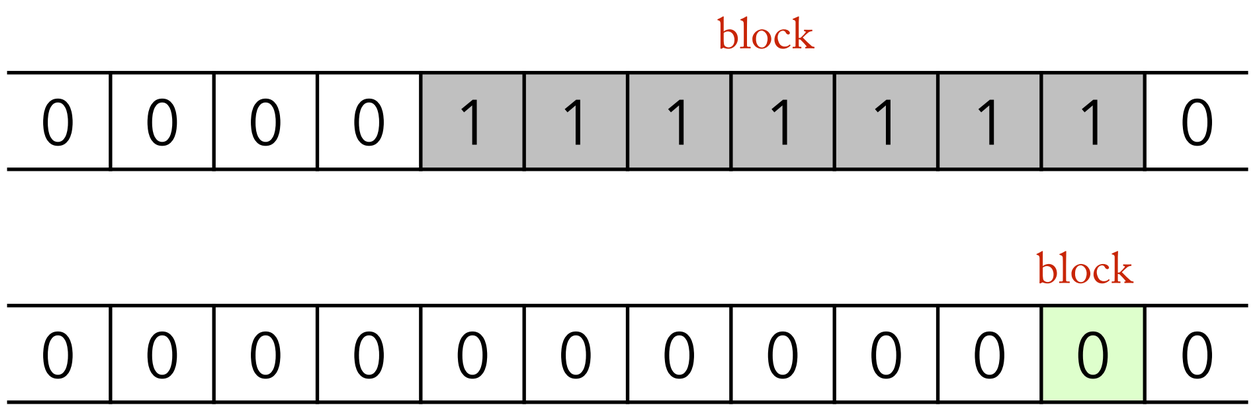


Computing a two place function:



Definition of a tally block

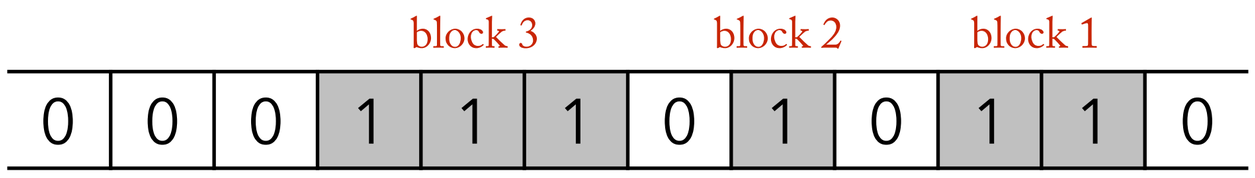
A tally block is either (a) one or more consecutive 1's; or (b) a single empty cell, that serve to represent the argument to a function.



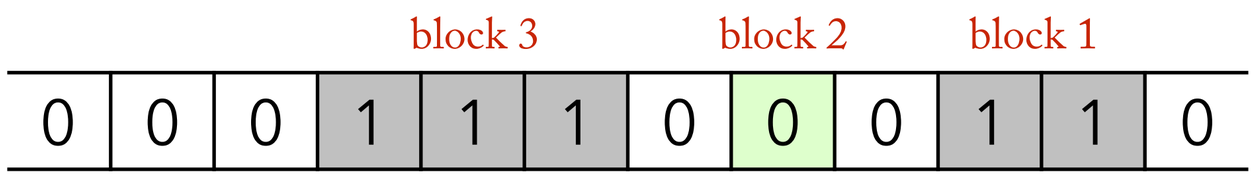
Definition of a tally input

The initial state of a tape counts as **an input** for computing a function *f* iff:

Condition 1: the tape contains a series of blocks, each separated by a single 0.



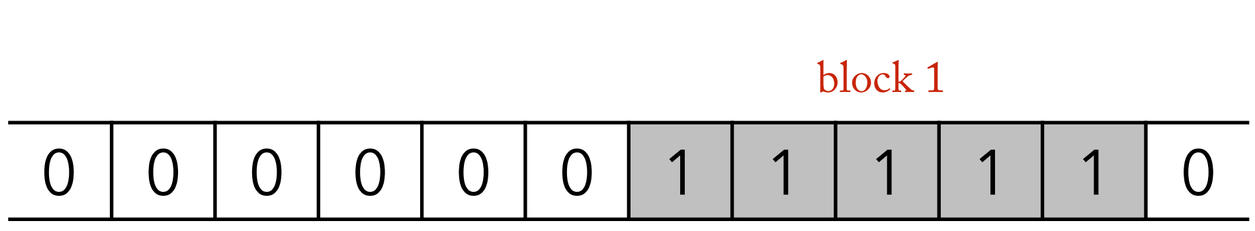
If a block is empty, it still takes up one cell.



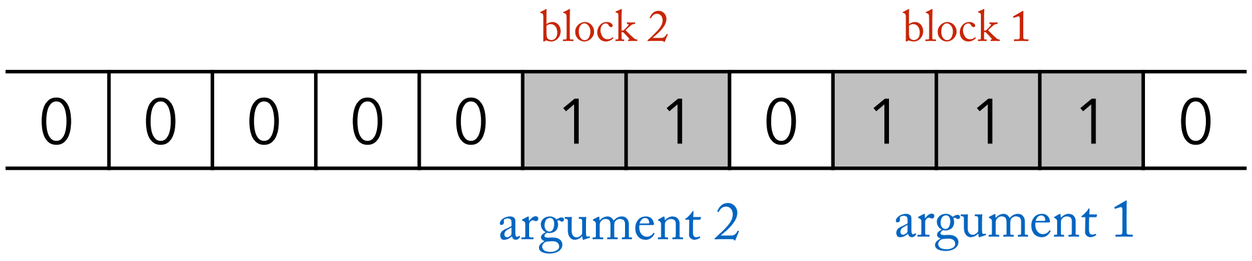
The initial blocks may be located anywhere on the tape.

Condition 2: there are as many blocks on the tape as there are arguments for the function being computed.

$f(x) = x + 1$



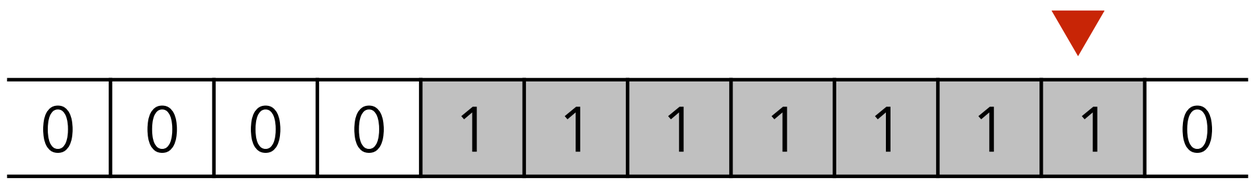
$f(x,y) = x + y$



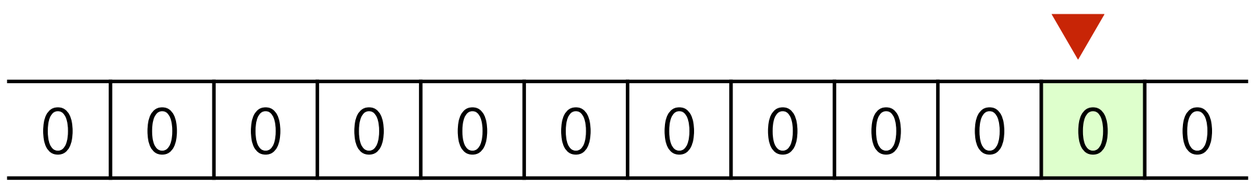
Counting from the right, the first block on the tape corresponds to the first argument of the function computed, the second block corresponds to the second argument, and so on.

Note: if the machine in question is a sub-routine of a larger machine, there may be other blocks on the tape that are irrelevant to its operation.

Condition 3: the machine starts in **standard position.** For tally, standard position is defined as **the right most cell of the right most block**.



Even if the block is empty.



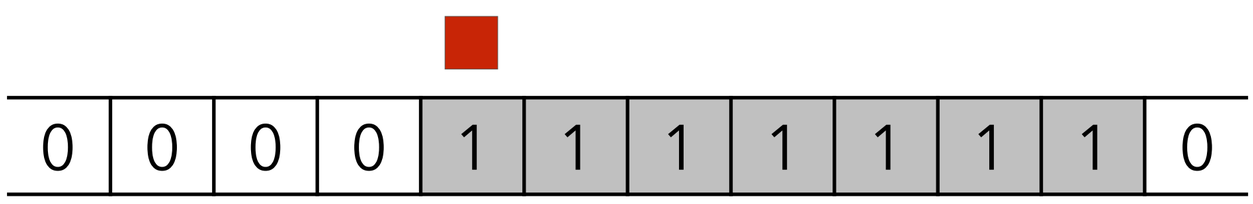
Note: If all blocks are empty, it doesn't matter where on the tape the machine starts.

Definition of a tally output

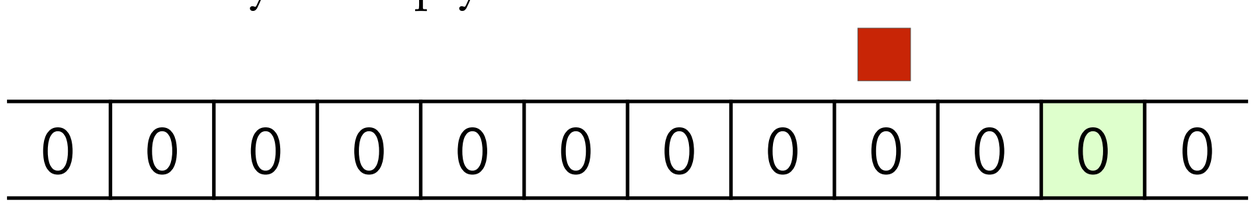
The initial state of a tape counts as **an output** for computing a function *f* iff:

Condition 1: the machine halts at some point.

Condition 2: when the machine halts, there is only a single block left on the tape.



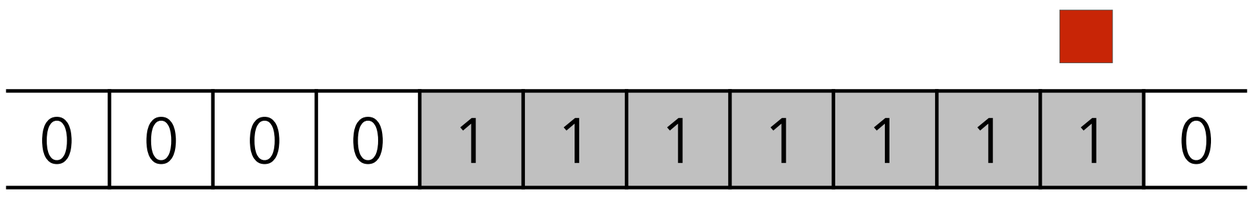
The final block may be empty:



The final block may be located anywhere on the tape.

In some circumstances, a third condition will be added. You can assume it does *not* apply, unless explicitly required.

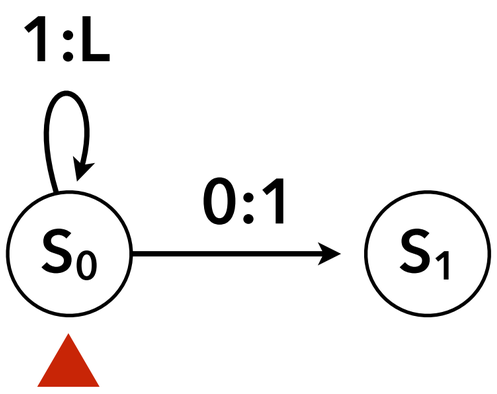
(Condition 3) The machine halts in standard position.



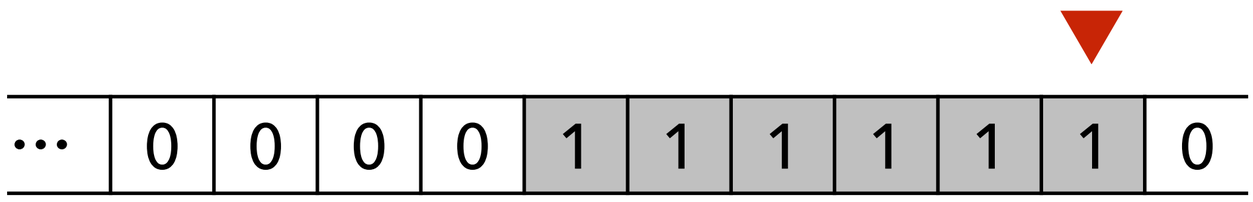
Note: standard position here is the right most cell of the right most block. It is *not* necessarily the same standard position you started in.

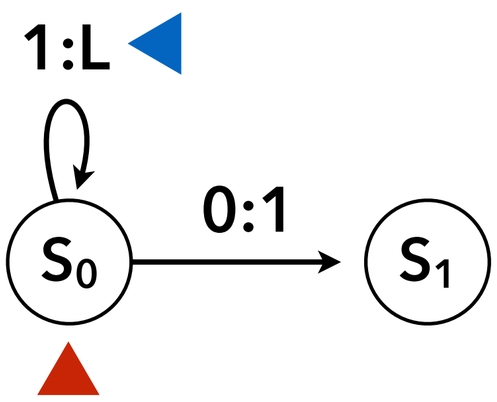
Note: if the final block is empty, then Condition 3 is satisfied by halting anywhere.

Example: +1 T

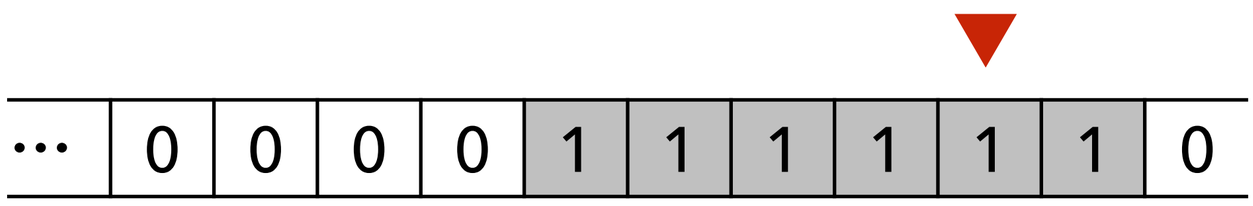


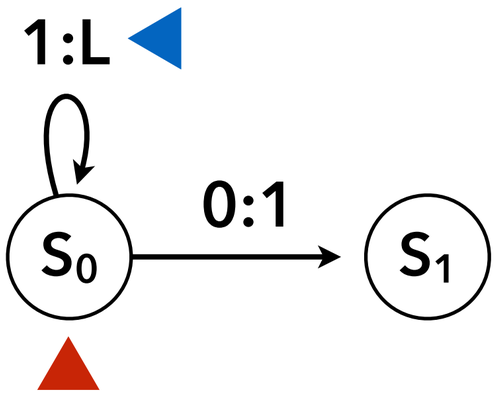
Start with S_0 in the machine, and standard position on the tape.



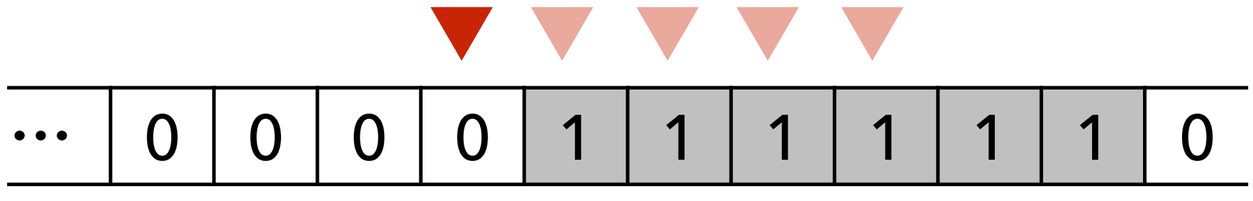


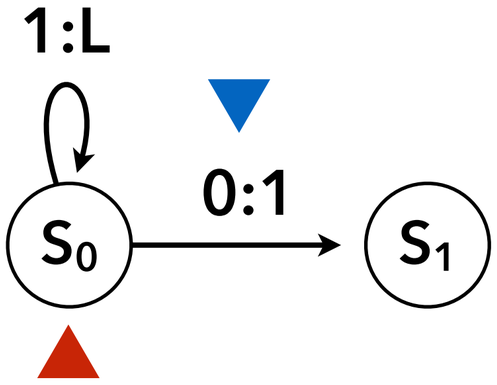
If you read a 1, move left one cell.



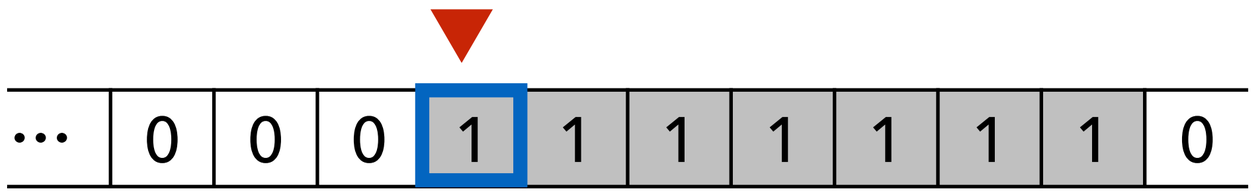


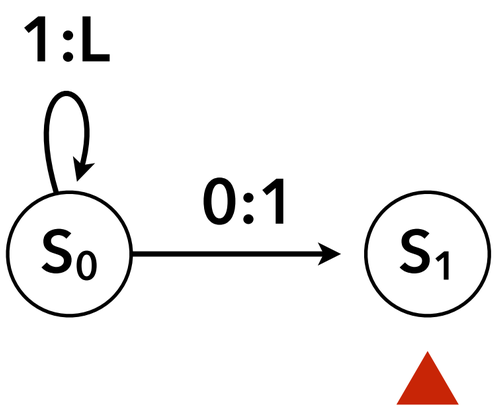
This operation repeats five more times….



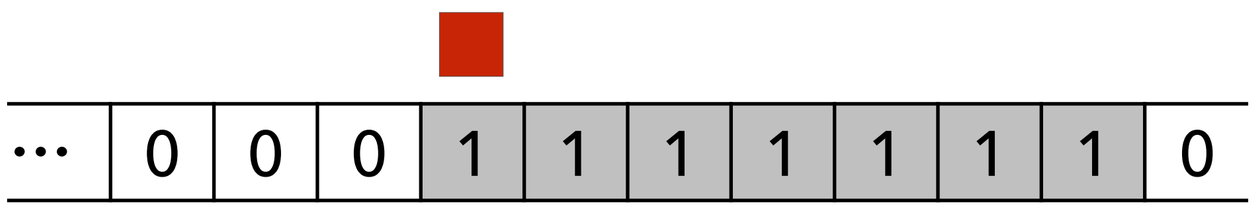


If you read a 0, write a 1.





There are no conditions for exiting S_1, so the machine halts.

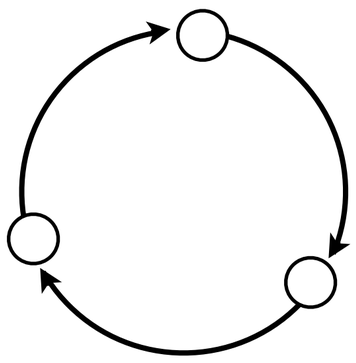


30. Recursion

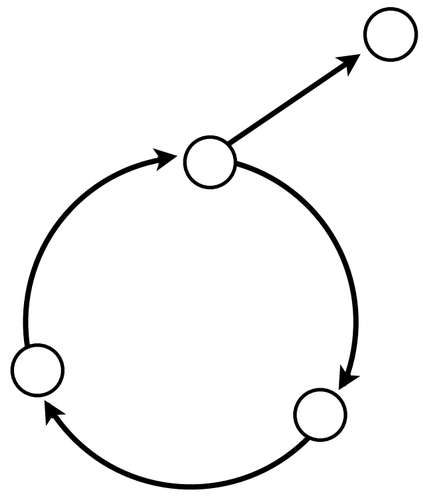
The Concept of Recursion

A recursive algorithm repeats some process with slight modifications until some goal is reached. "Recursive" means repeating. Recursive algorithms occur not only in computation, but also in natural processes.

A finite sequence of states.



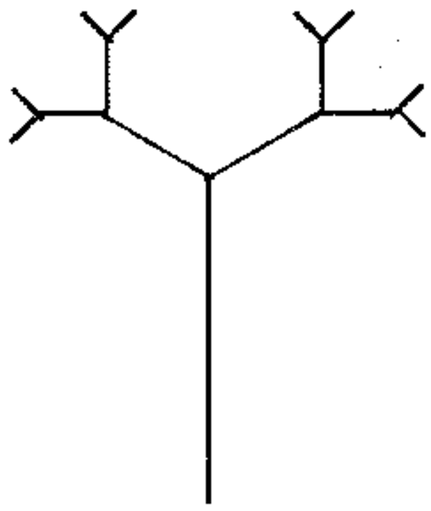
A loop of states which repeat infinitely.



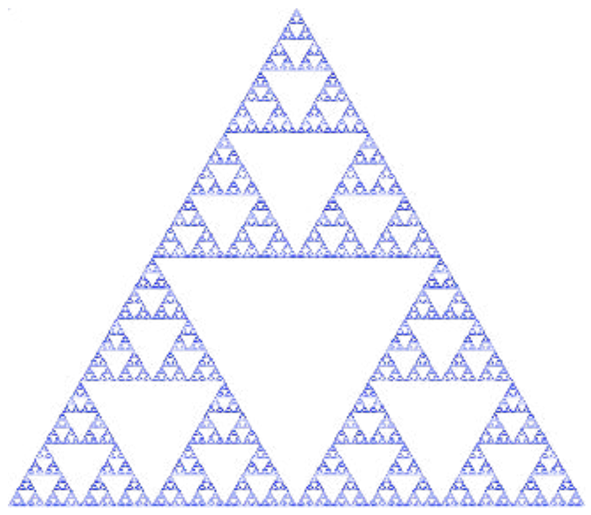
A loop which repeats a finite number of times until an exit condition is reached.

Recursive algorithms can be used to generate drawings with elements that repeat, but also change a little bit, after each loop.

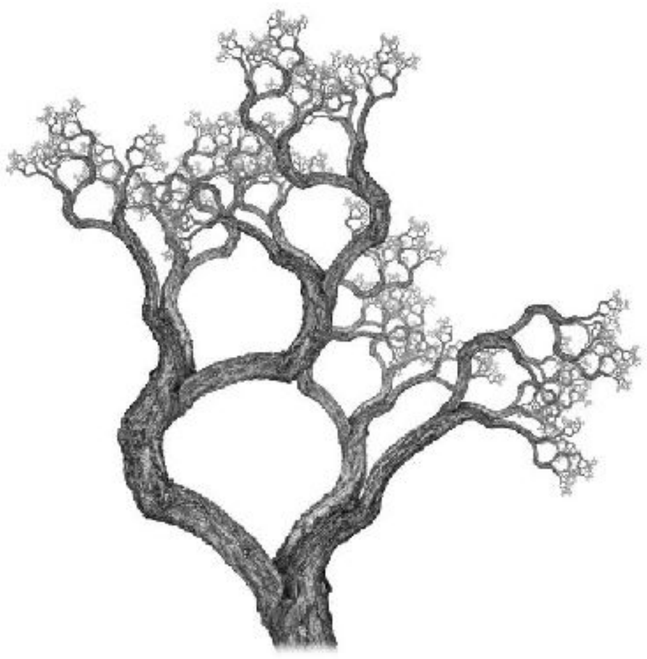
```
TO TREE :SIZE
FORWARD :SIZE
IF :SIZE <1 STOP ELSE TWO-
TREES SIZE/2
BACK :SIZE
END
```



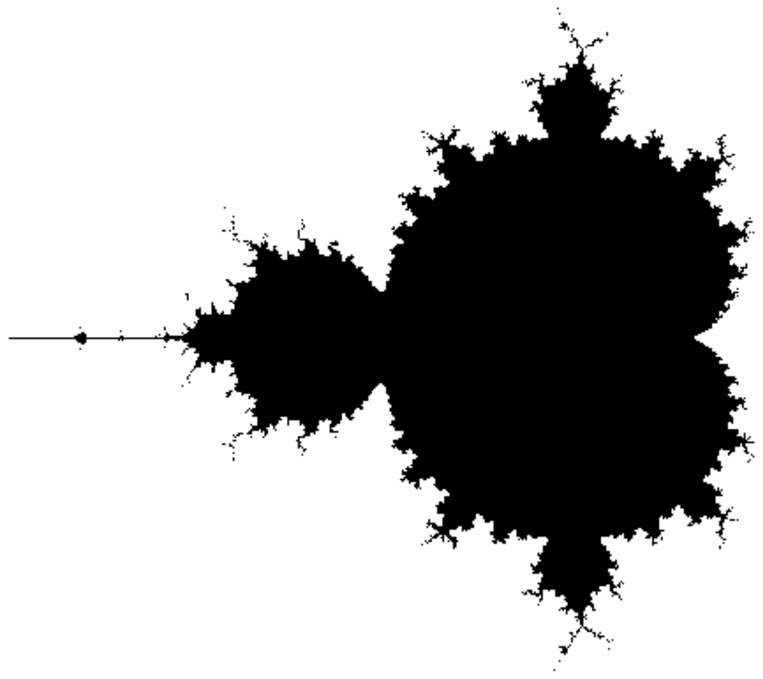
This program written in LOGO describes an algorithm for drawing simple trees.



How would you generate Sierpinksi's triangle?



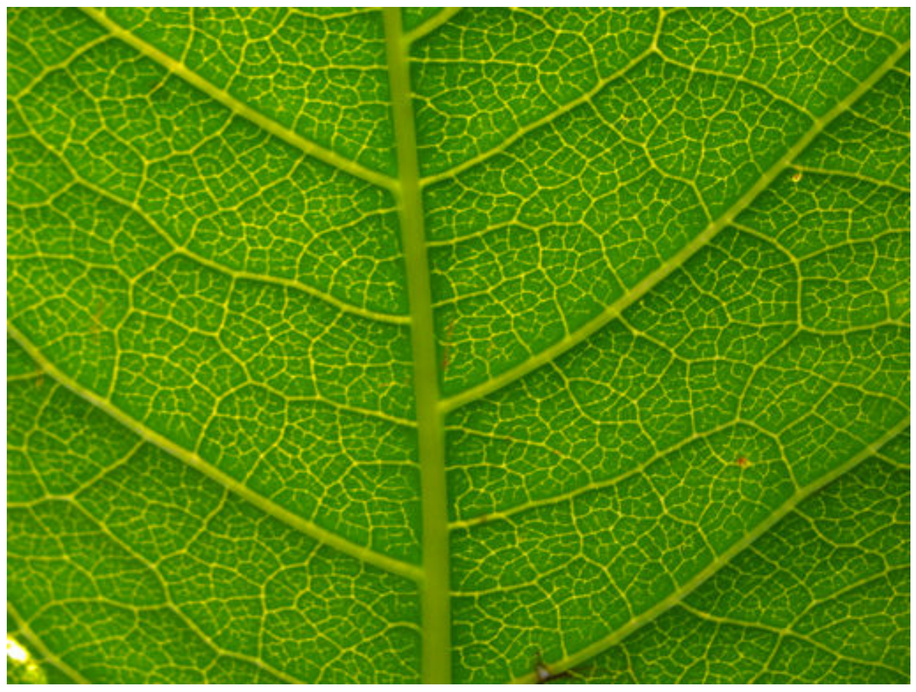
The same technique can be used to draw more realistic-looking trees…



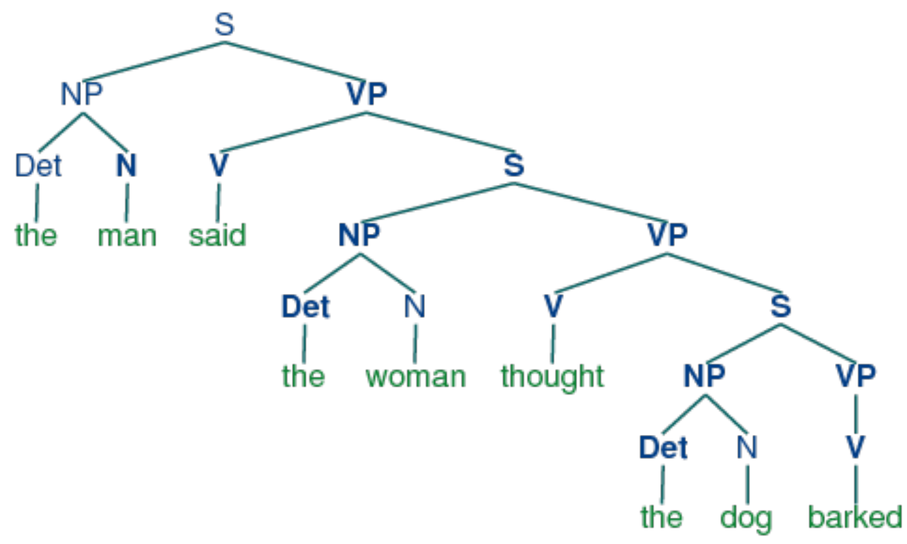
… or bizarre and complex fractals.

Recursion in Nature

The natural world is filled with recursive structures. Consider a tree: how are the branches generated? How is each leaf on a branch generated? How is each cell is a leaf generated?



Cell growth works by repeating the same kind of operation over and over, with slight modifications made for a single kind of structure (an organ, a leaf), and larger modifications.



Famously, language is also thought to have a recursive structure. Sentences are formed by recursively combining parts of speech.

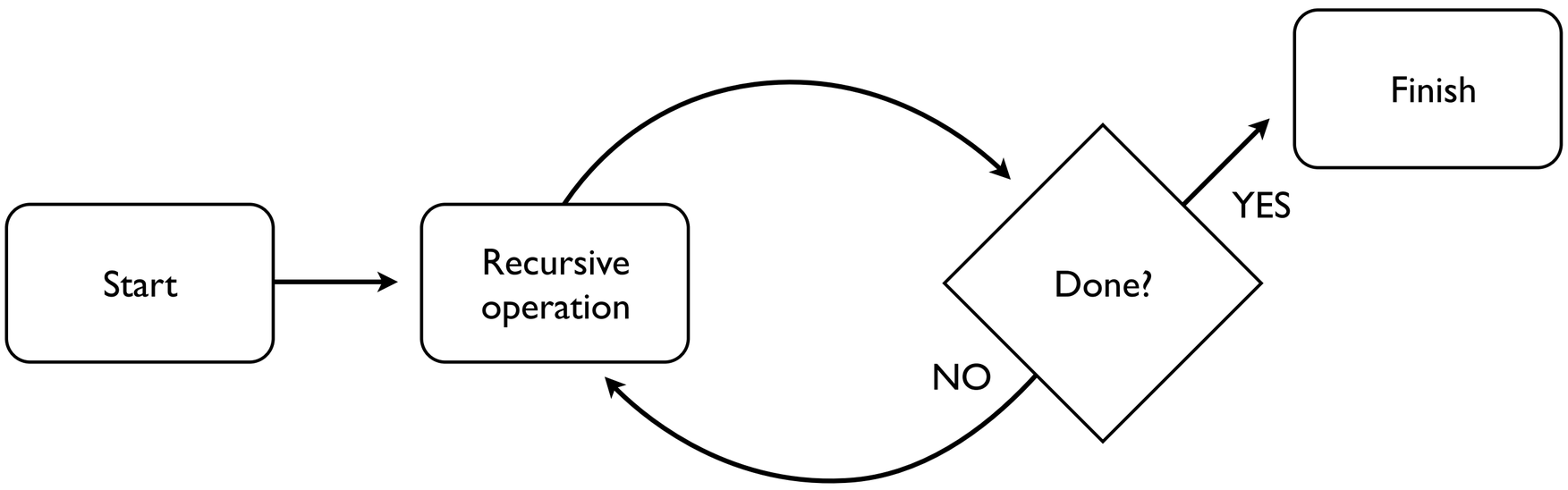
Both cellular growth and sentence structures are driven by recursive operations. But do they have the same root cause?

Perhaps language is recursive *because* biological growth is recursive. Somehow the biological structure of the brain requires it. Perhaps. A more plausible alternative is that both recursive for the same reason. Recursive algorithms are the central mechanism for generating indefinite complexity from a finite base of instructions.

The same pressures which made biology recursive are also what make language recursive, and it is the same pressure which makes computer programs recursive. All face the same kinds of ecological pressure: infinite ends, finite means.

The Architecture of Recursion

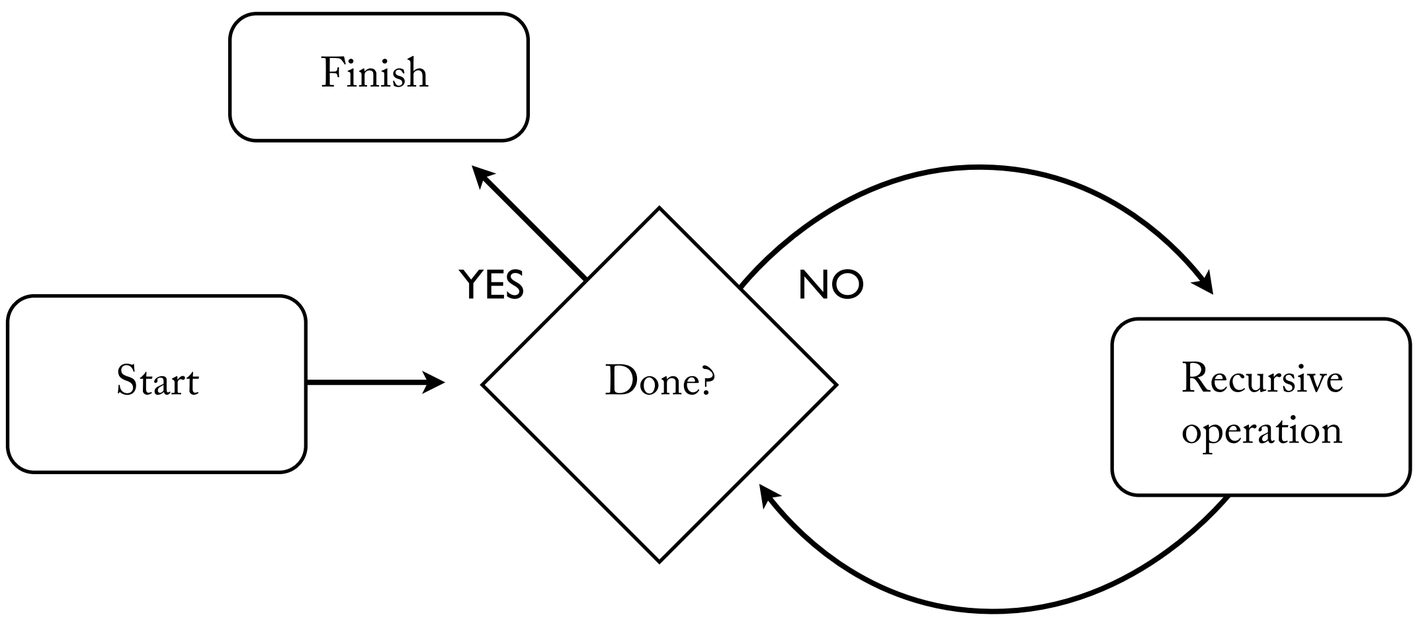
Recursive algorithms have a characteristic architecture. Here is a typical design:



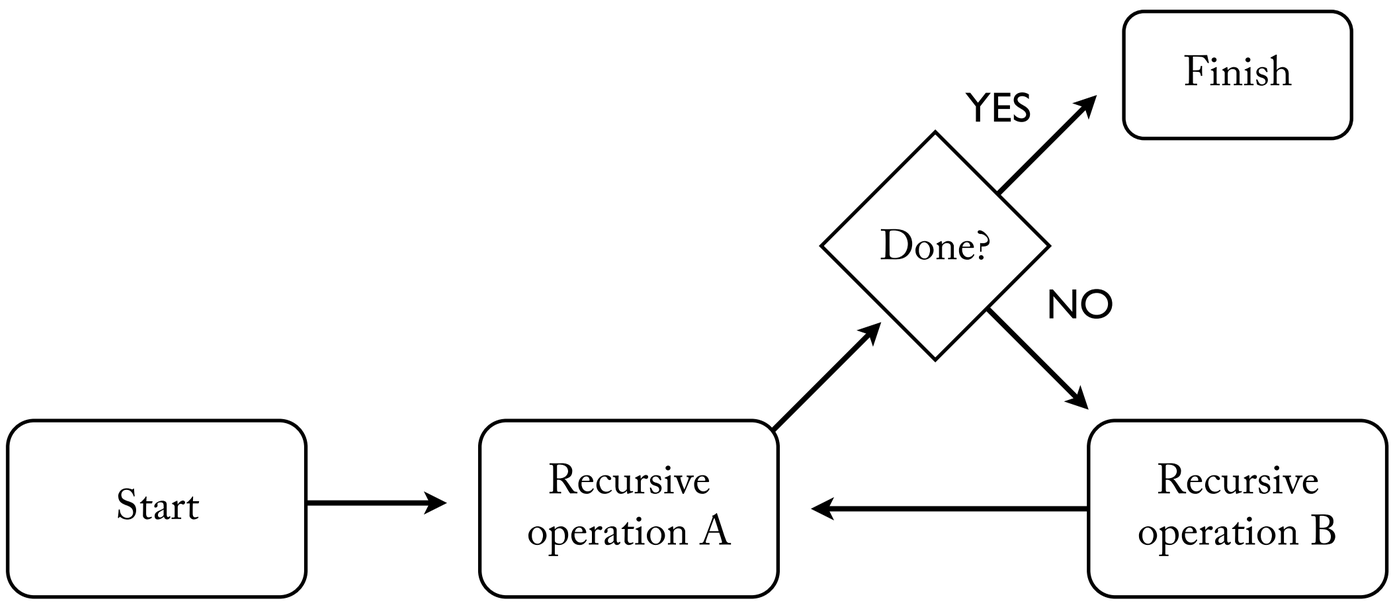
A recursive algorithm has three key elements:

(1) a **recursive operation** which is a repeated;
(2) a way to **test** whether the process should be repeated again;
(3) an **exit operation** when the process is complete.

The three elements may appear in nearly any order. You'll have to find the order best suited for the problem.



Often the recursive operation will itself involve several discrete operations, and these can be re-arranged as well. The underlying conceptual structure is the same.



Note: these diagrams are **flow charts**, a common system for describing algorithms. See the Section "Flow Charts" for more about creating and reading these diagrams.

Case Study: 2*x* T

The goal is to design a TM computes $2x$ in Tally. For present purposes assume $x \geq 1$. Here it is instructive to remember what multiplication really means. Multiplying x by 2 is equivalent to adding 2 to 0, x times. For example:

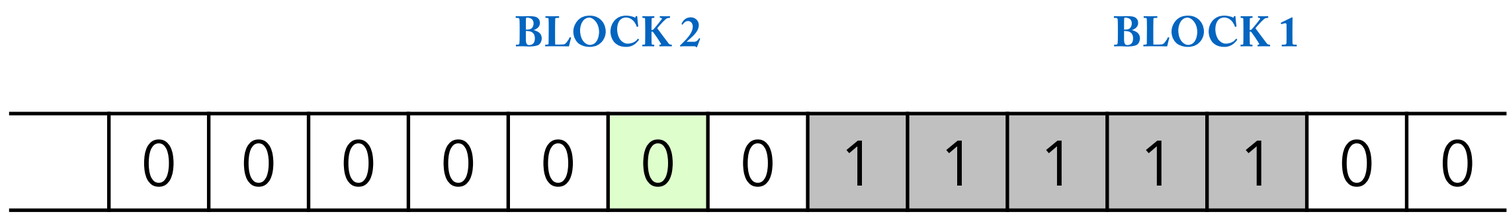
$$\begin{aligned} 2 \bullet 3 &= 0 + 2 \\ &= 2 + 2 \\ &= 4 + 2 \\ &= 6 \end{aligned}$$

3 times

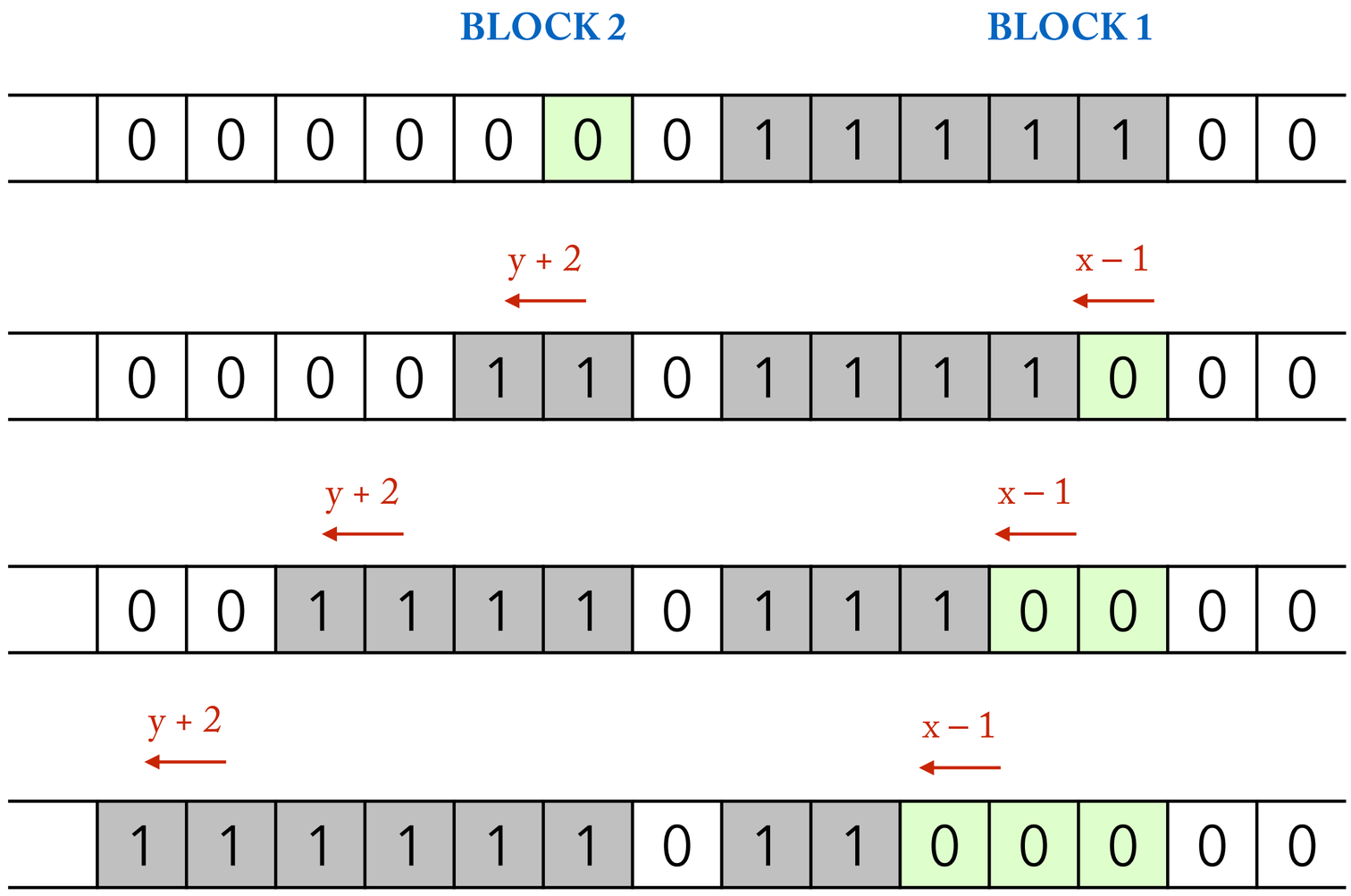
$$\begin{aligned} 2 \bullet 5 &= 0 + 2 \\ &= 2 + 2 \\ &= 4 + 2 \\ &= 6 + 2 \\ &= 8 + 2 \\ &= 10 \end{aligned}$$

5 times

To approach this problem, let us imagine two blocks on the tape. Block 1 (B1) represents the argument x. Block 2 (B2) represents the value $2x$. B1 starts with a value, but B2 starts empty.

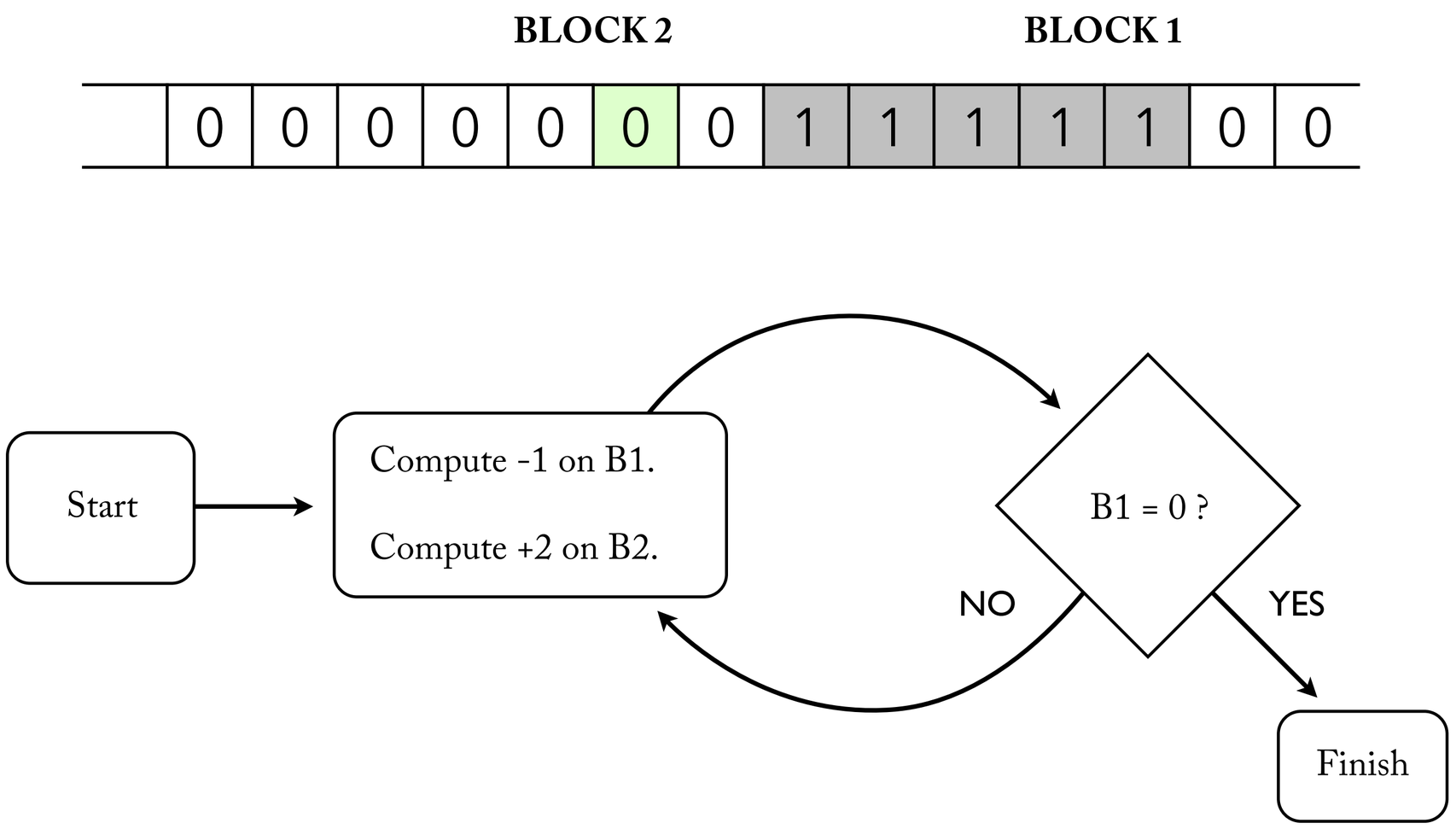


The strategy is this: let us repeatedly remove 1 from B1, and each time we do so, add 2 to B2. In other words, each time we compute x-1 on B1, we'll then compute y+2 on B2. When B1 is empty, B2 should have twice as many cells filled as B1 did originally.

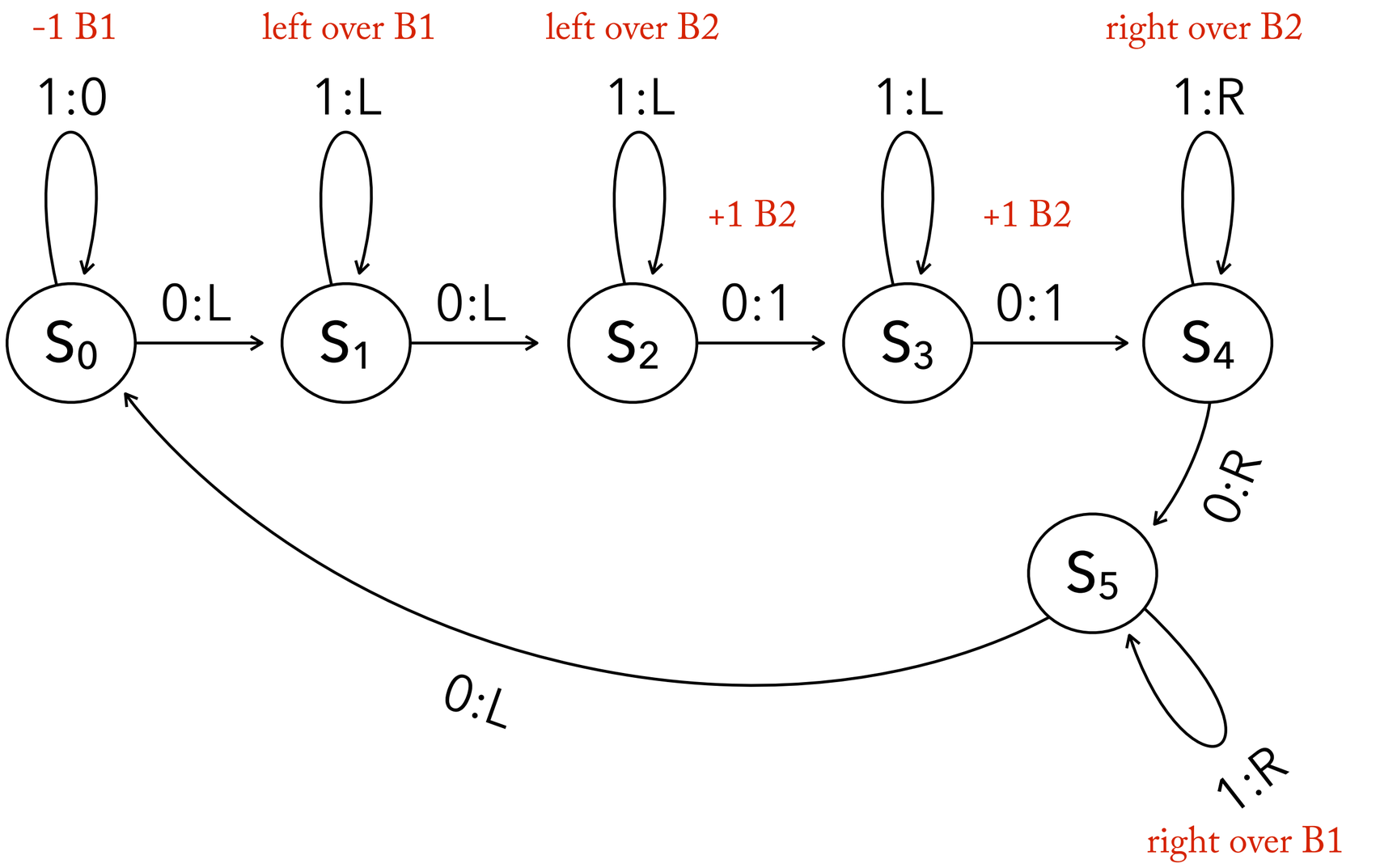


Here B1 functions as **counter**: it tells us how far we are into the comptuation. B2 functions as the **answer.** Nearly all recursive algorithms will include both elements. **Note:** here I remove 1 from the right side of my counter, but the left side would work just as well.

Before drawing up our TM, let us convert our strategy into an explicit flow chart:

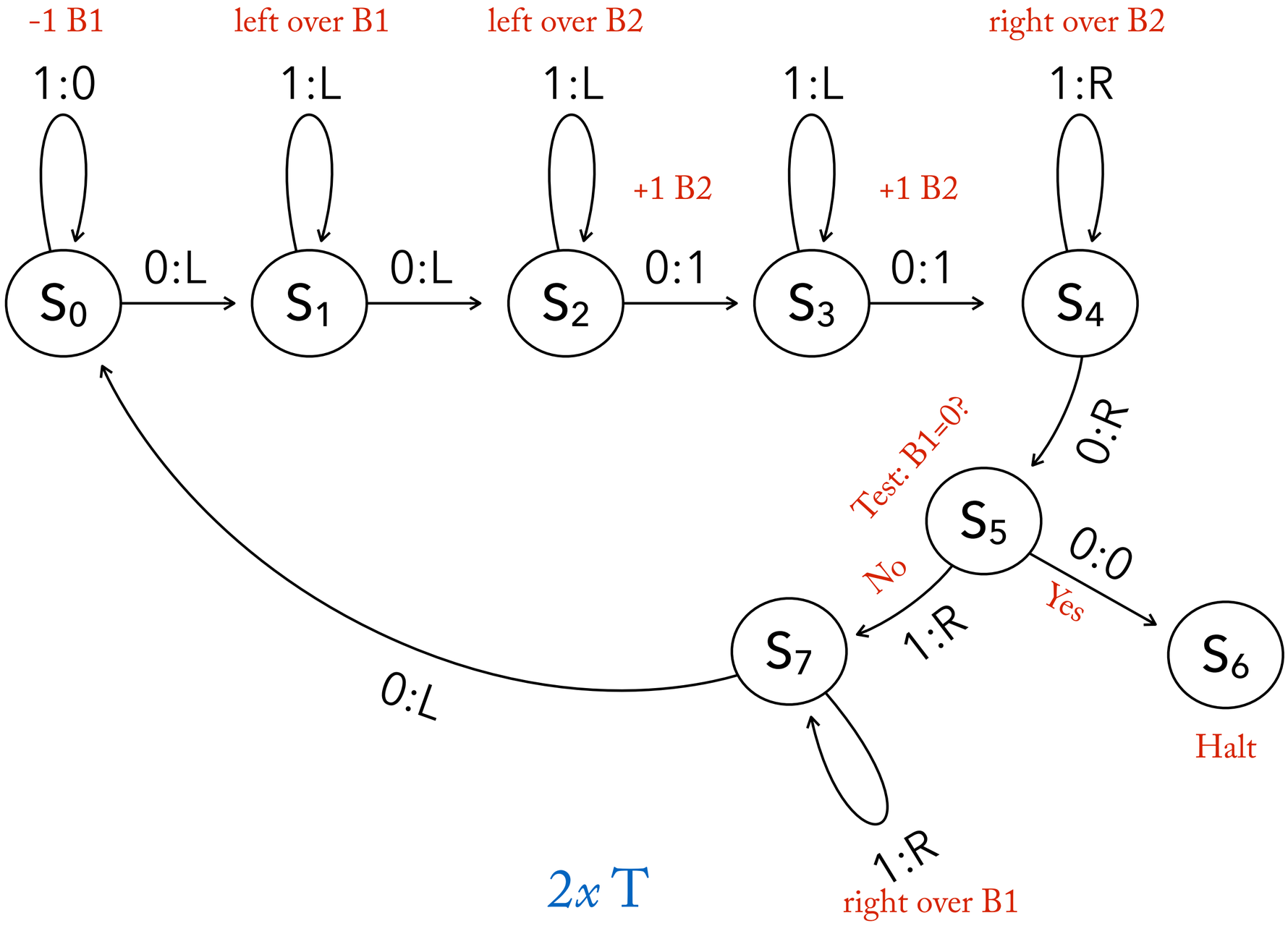


To turn this design into a TM, a good starting place is just to put the critical elements into a loop: (i) -1 from B1, (ii) +1 to B2, (ii) +1 to B2.



The problem with this design is that it contains no **test** or **exit operation**. As a result, it loops indefinitely. Can you predict what will happen to this machine after B1 is empty?

Finally, here is a fully annotated $2x$ Tally TM. It's just like the machine above, but builds in a test to see when B1 is empty, and an explicit exit condition if it is.



Thinking through *x*+*y* T

How can we break addition up into a simple, repeating operation? Remember that addition is just successor applied over and over

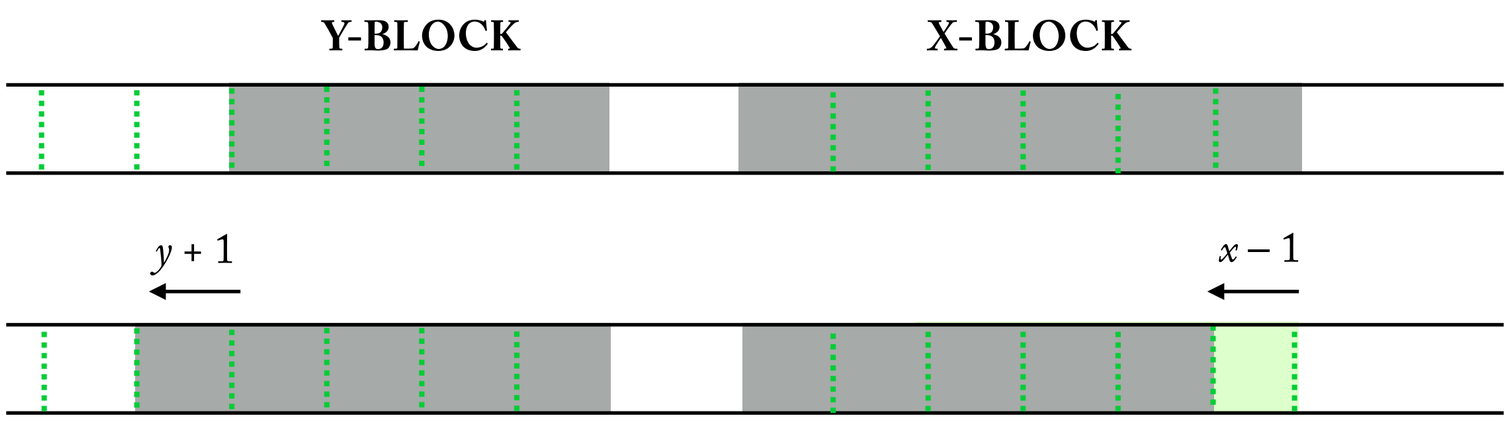
$$\begin{aligned} 2 + 3 &= 2 + 1 \\ &= 3 + 1 \\ &= 4 + 1 \\ &= 5 \end{aligned}$$

3 times

$$\begin{aligned} 4 + 5 &= 4 + 1 \\ &= 5 + 1 \\ &= 6 + 1 \\ &= 7 + 1 \\ &= 8 + 1 \\ &= 9 \end{aligned}$$

5 times

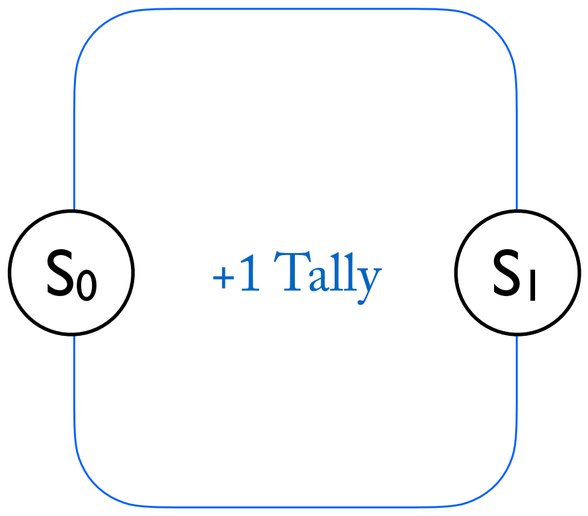
The next step is to work through a strategy. Here's a start…



31. Turing Machine Abstraction and Composition

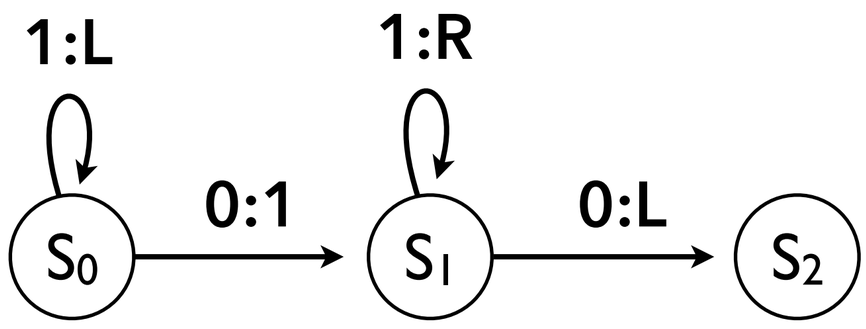
Functional Abstraction for Subroutines

We can apply functional abstraction to TM's which compute specific functions.

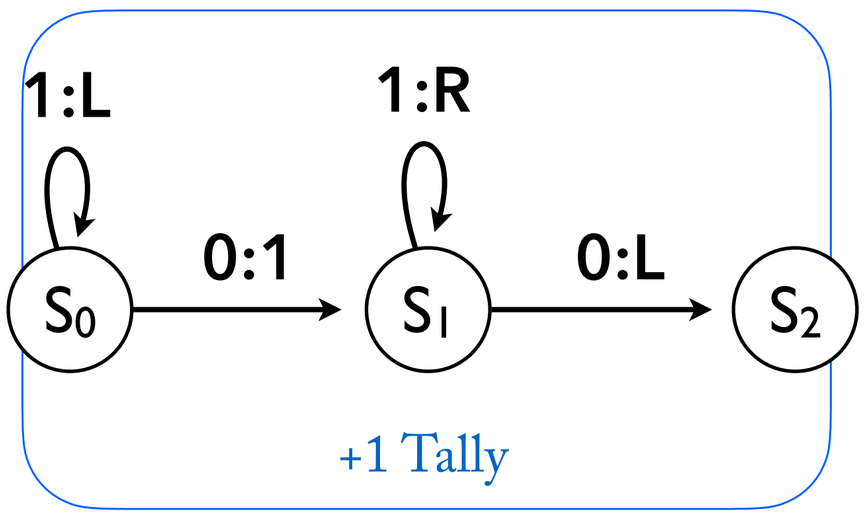


This boxed TM starts in S_0 and halts in S_1. By the time it halts, it will have computed +1 T.

Boxed TMs will be easier to combine into larger machines if they start *and* finish in standard position. For example, this TM computes +1 T and returns to standard position:



We can box this machine it by drawing a frame around it that intersects only with initial state and the halting state:



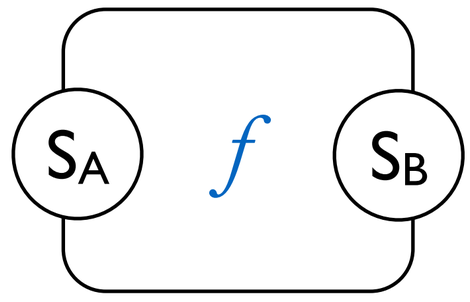
Then we can kick away the internal states:

Constraints on boxing

To prevent technical errors, we have to impose a few constraints on when a boxed machine may be introduced.

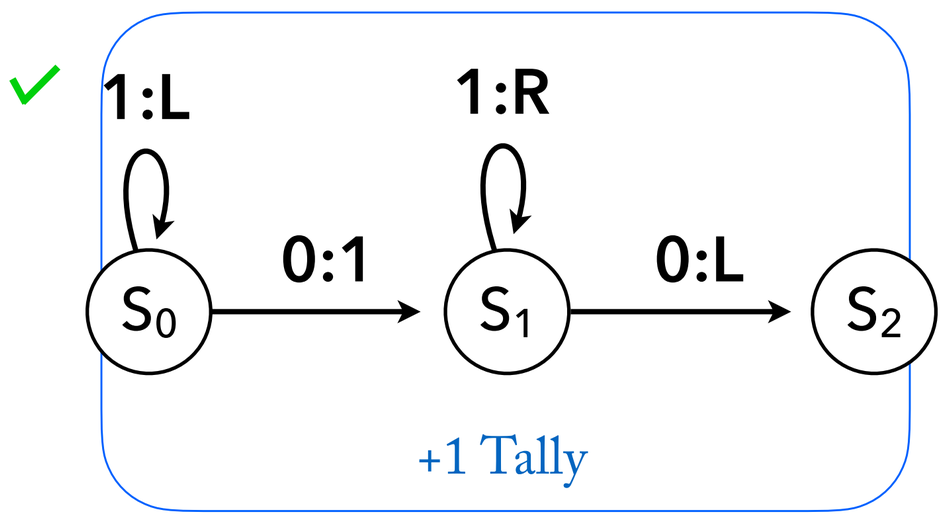
A machine M can be **boxed** only when:

1. M computes a function *f*.
2. S_A is the **initial state** of the machine.
3. S_B is the only **terminal node** of the machine:
   a) M halts in standard position;
   b) M halts in S_B;
   c) M halts in no other state (given the task);
   d) there are no transitions from S_B to other internal states of M.

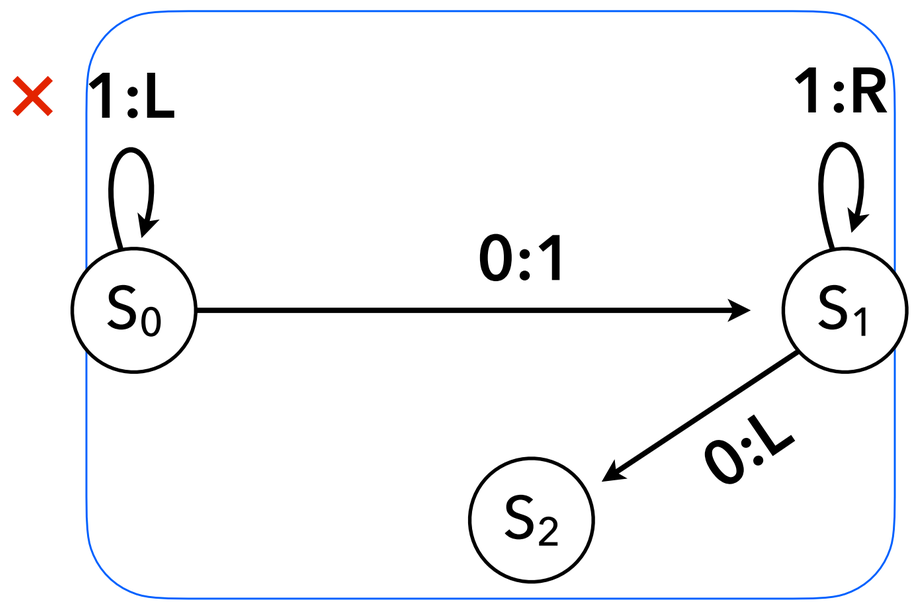


A boxed machine M can only be used in a larger circuit when there are no transitions from S_A to any states outside of M.

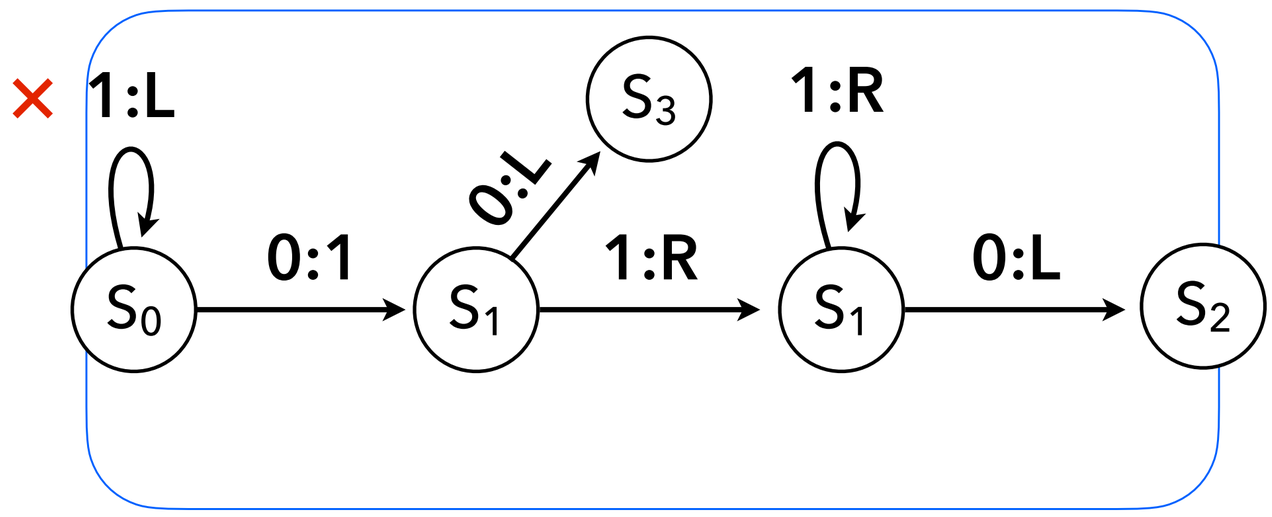
So this is right:



But this is wrong:

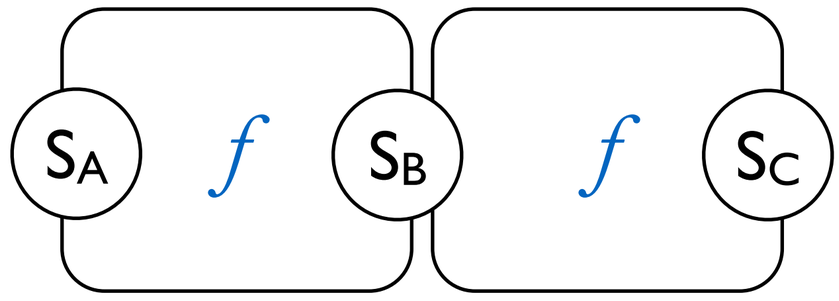


And this is wrong:

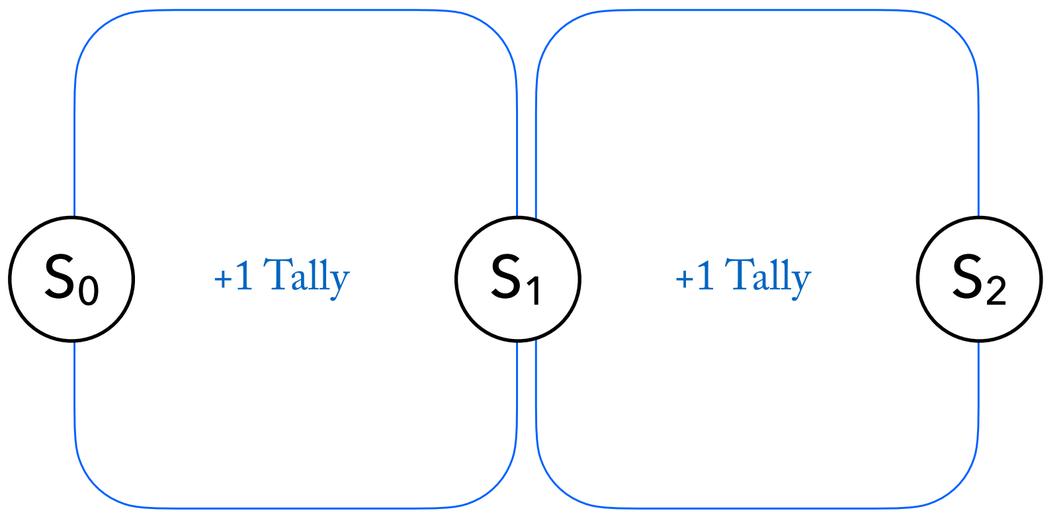


Function Composition

Boxed machines can be composed together to form larger machines:

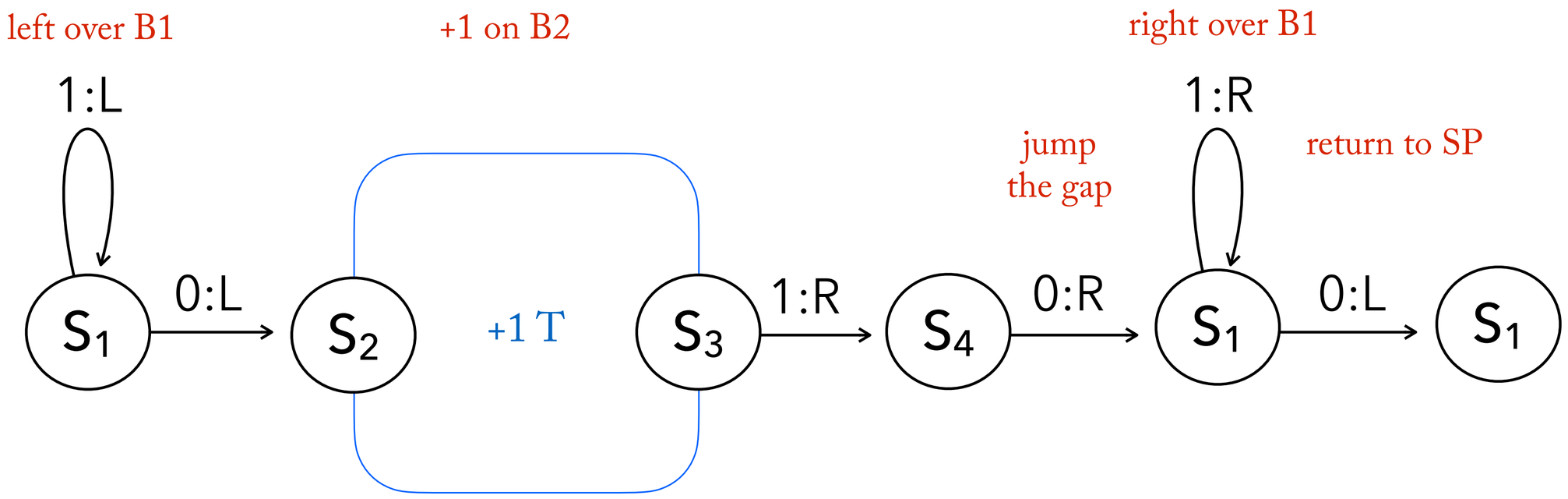


For example, this machine computes +2 T.



Boxed machines can be integrated with unboxed elements as well.

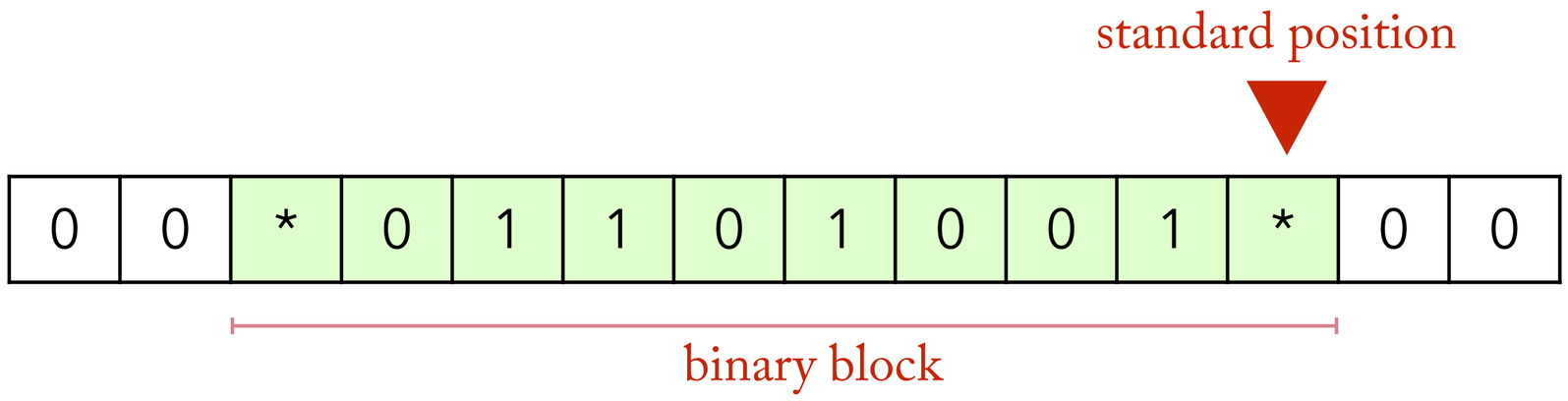
This machines moves left over a first block and then performs +1 T on a second block, then returns to standard position.



32. Binary Turing Machines

Set-up

- We now allow three symbols: 1, 0, *
- Turing Machines can read and write all three symbols, so up to *three* arrows can leave each state.
- A **binary block** is a sequence of one or more 1's and 0's, flanked on both sides by *'s:

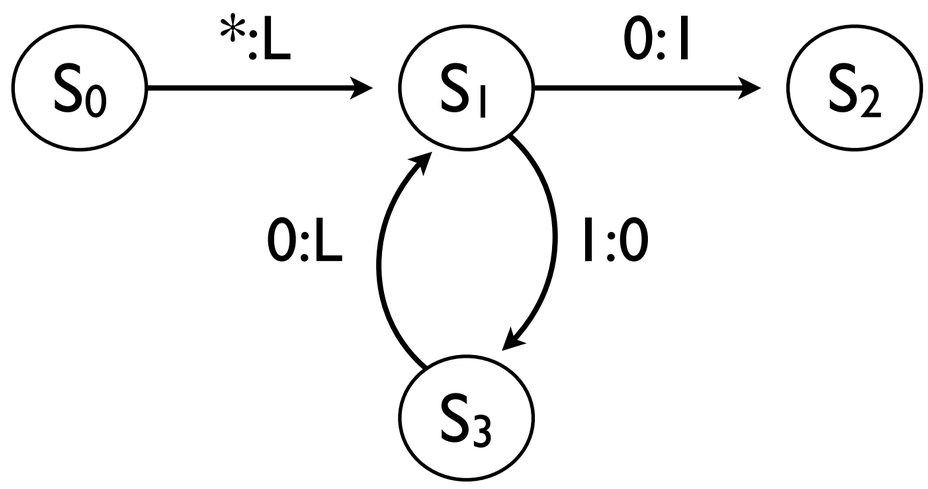


- **Standard position** is defined as the right-most * of the right-most block

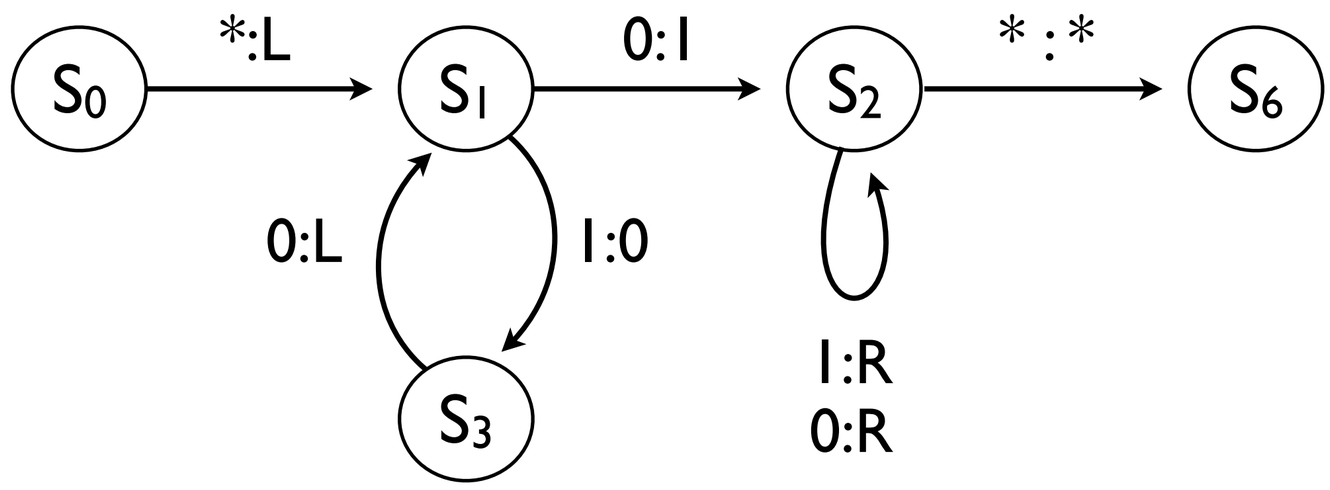
Binary +1

This machine computes the core of Binary +1.

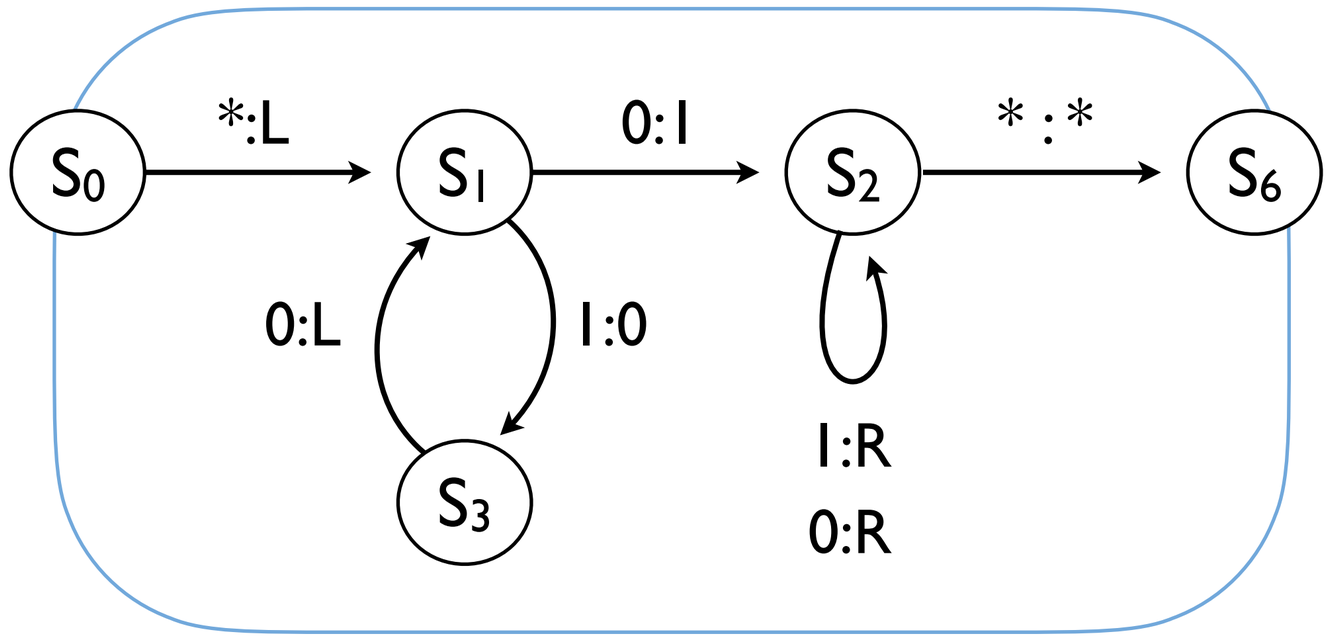
But it doesn't deal with the case where the input's are all 1's, and the left-most * must be overwritten with a 1.



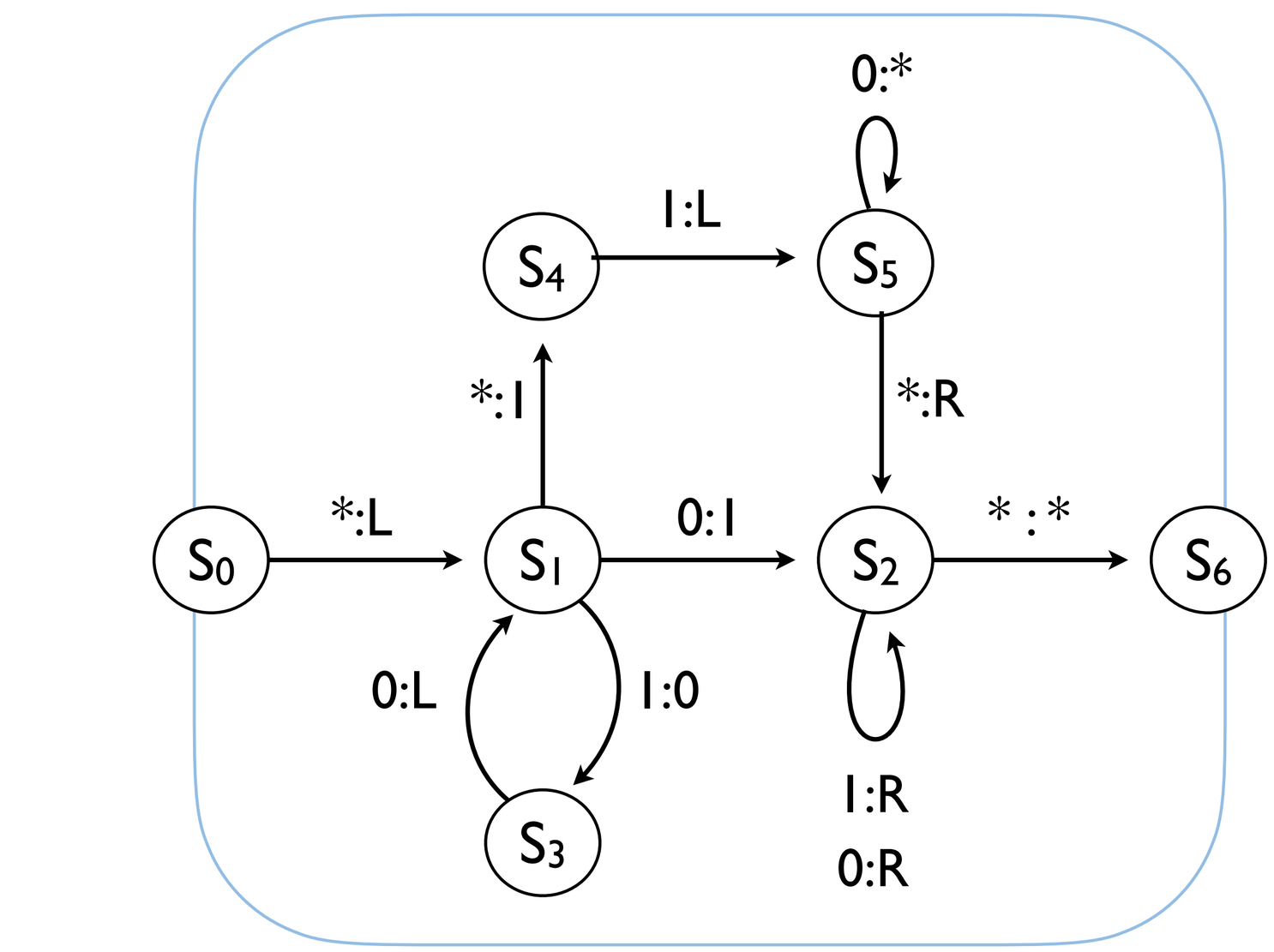
To facilitate boxing, let it return to standard position. Let's also add a terminal node.



This machine can be boxed:



And composed:



When dealing with unbounded binary problems, there is a wrinkle. Suppose the whole binary block is filled with 1's. To compute successor on this numeral, we must overwrite the left-most * with a 1, it then rewrites the * one cell to the left.

33. The Mathematical Tree

The Hyper-operation Sequence

Successor is a primitive function: it is defined in terms of no simpler function.

$$x + 1$$

Addition is defined recursively in terms of successor.

$$x + y = x + \underbrace{1 \ldots + 1}_{y \text{ times}}$$

y copies of 1 added to x

Multiplication is defined recursively in terms of addition.

$$x \times y = \underbrace{x + x \ldots + x}_{y \text{ times}}$$

y copies of x combined by addition

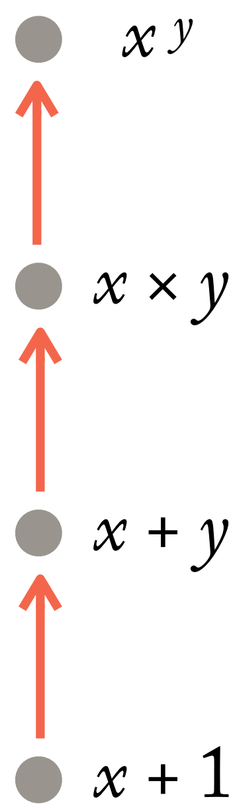
Exponentiation is defined recursively in terms of multiplication.

$$x^y = \underbrace{x \times x \ldots \times x}_{y \text{ times}}$$

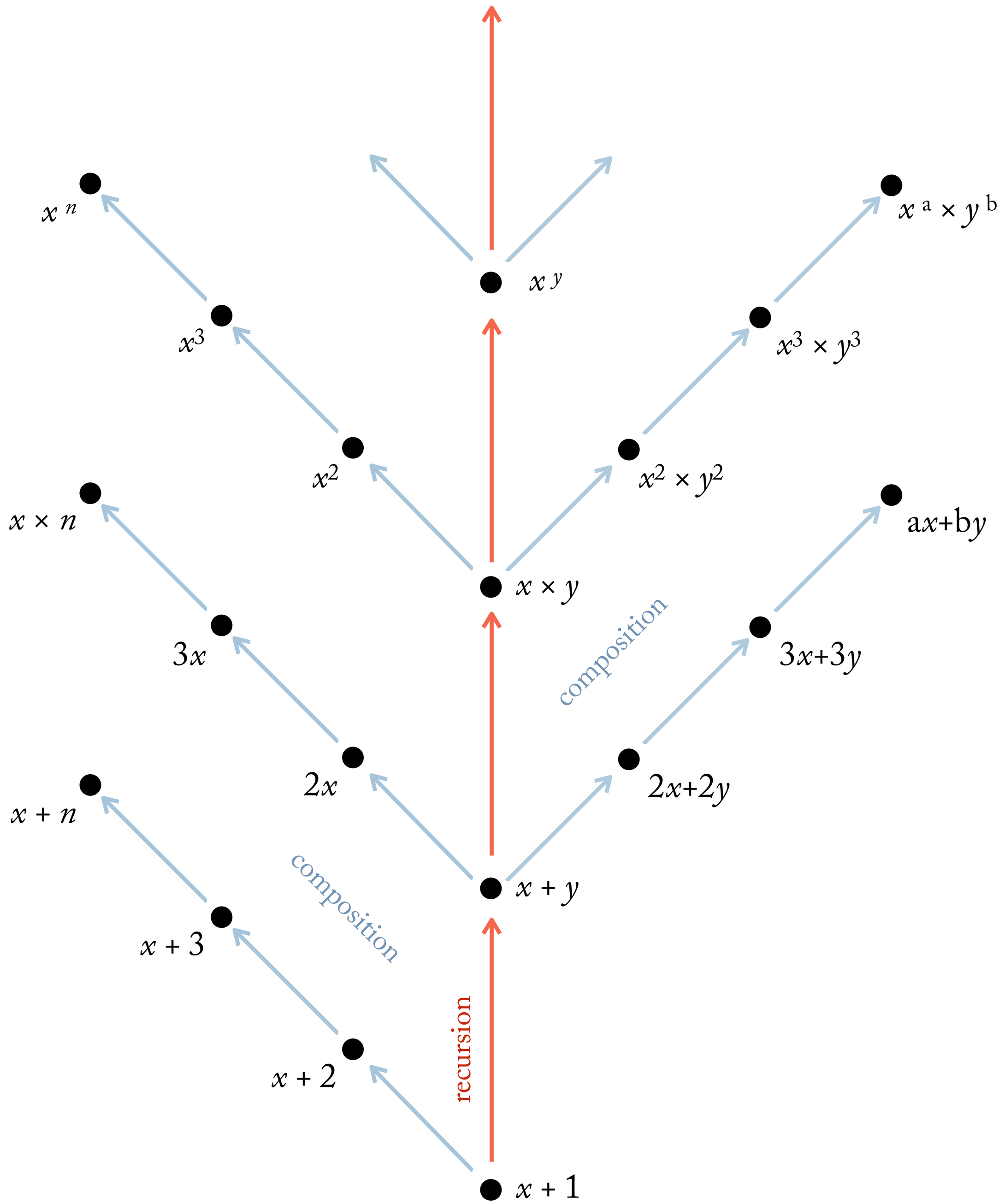
y copies of x combined by multiplication

Through recursion, we can build steadily up from successor to the further reaches of mathematical space.

Each function in the sequence can be computed by an algorithm that relies on a sub-routine for the previous function in the sequence.

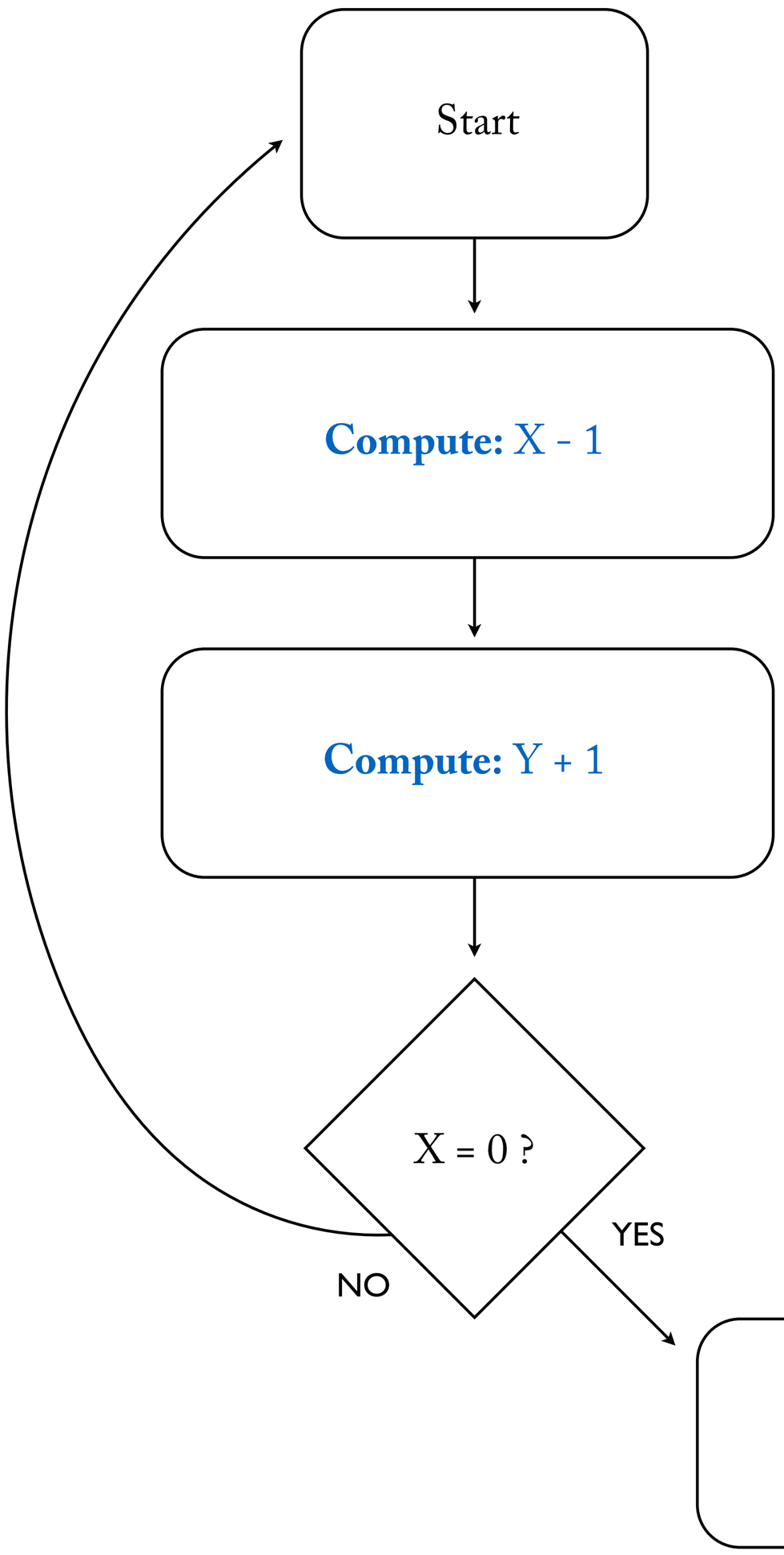


Recursion + Composition = Computable Mathematics



Recursive Addition

Compute $f(x,y) = x + y$



How can we break addition up into a simple, repeating operation?

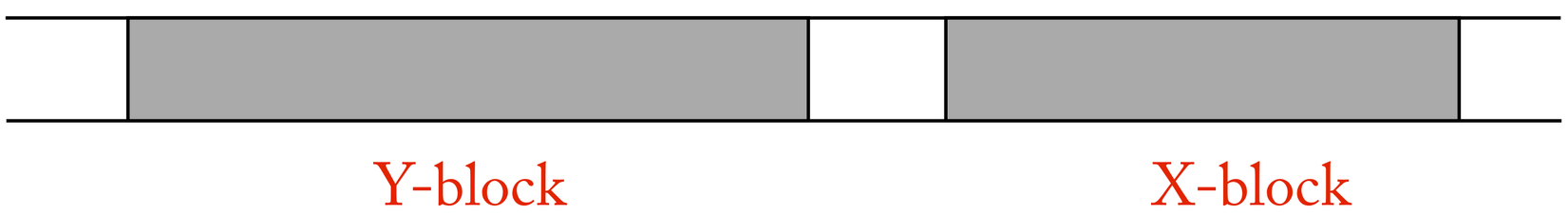
Adding x to y is equivalent to adding 1 to x, y times.

$$
\begin{aligned}
2 + 3 &= 2 + 1 \\
&= 3 + 1 \\
&= 4 + 1 \\
&= 5
\end{aligned}
$$

3 times

$$
\begin{aligned}
4 + 5 &= 4 + 1 \\
&= 5 + 1 \\
&= 6 + 1 \\
&= 7 + 1 \\
&= 8 + 1 \\
&= 9
\end{aligned}
$$

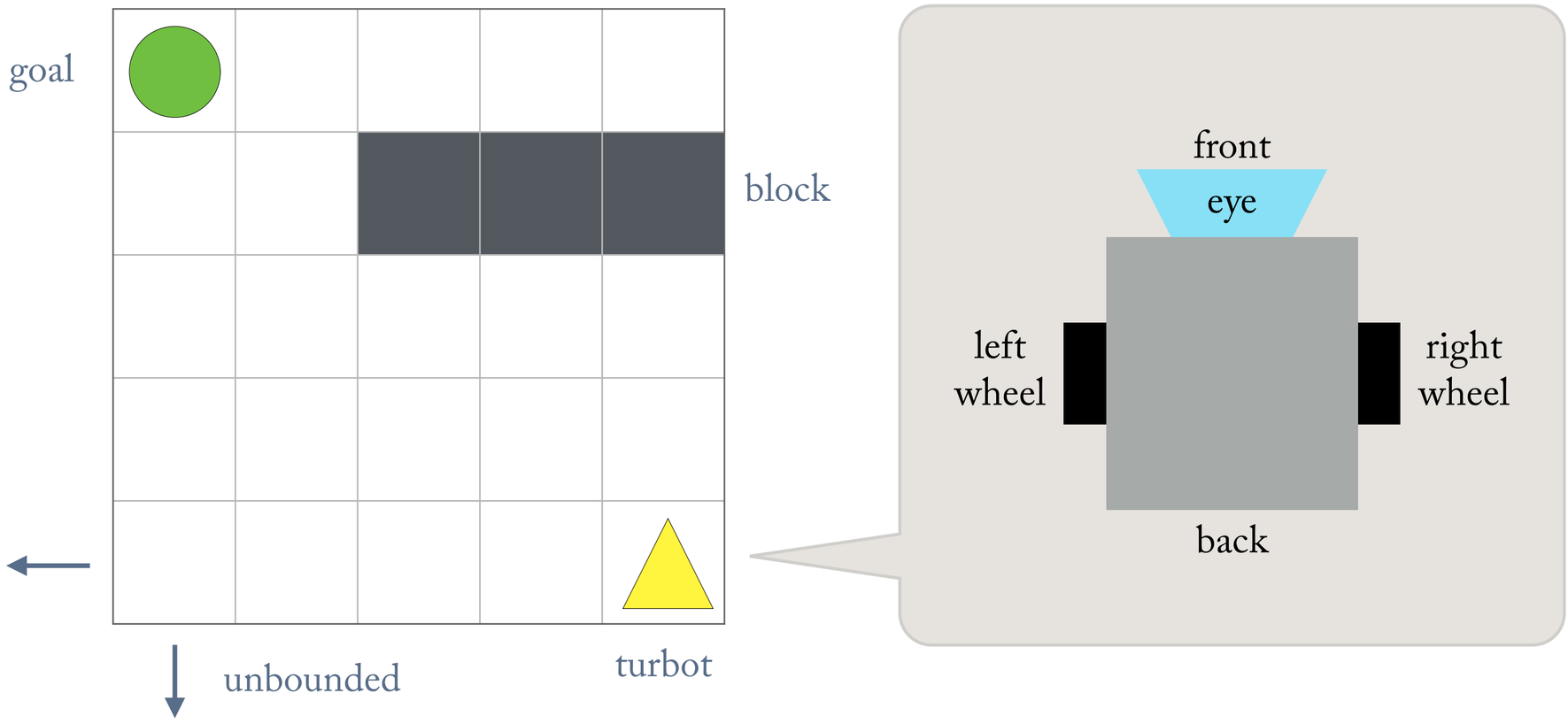
5 times



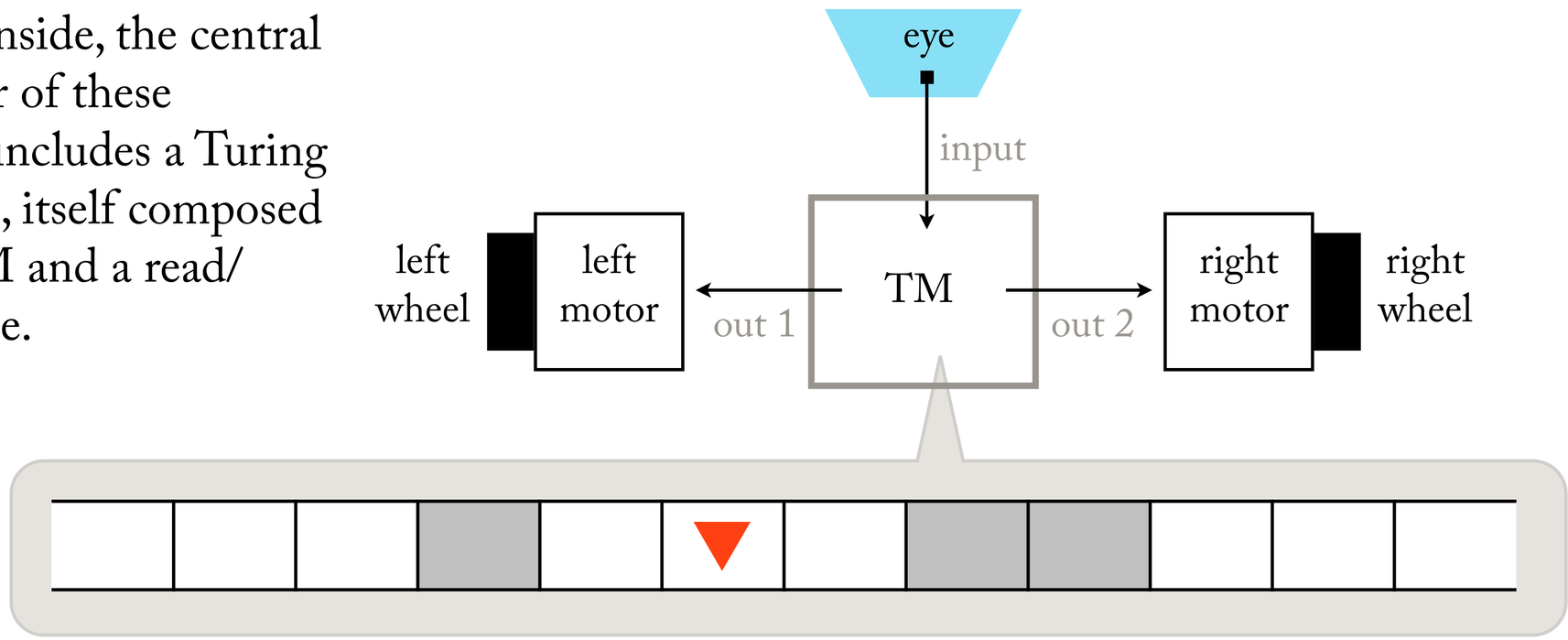
34. Computational Navigation

Turbots as Turing Machines

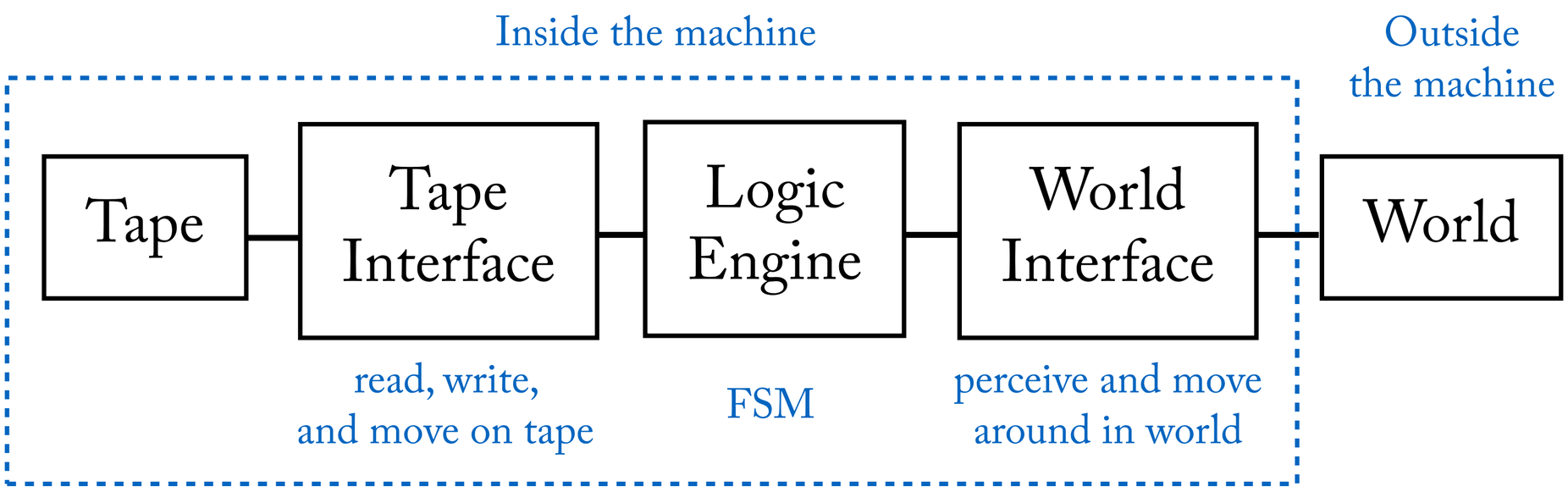
We next introduce Turbots based on Turing Machines. Viewed from the outside, they are just like the other navigational agents we've designed.



On the inside, the central processor of these Turbots includes a Turing Machine, itself composed of a FSM and a read/ write tape.



Turbots negotiate between two worlds: the *internal* world of the tape, and the *external* world of the environment. In this way they are just like you or me.



Turbots: Operation

- Turbots have an *internal* tape, but also act on an *external* world.

- At any moment, a Turbot has both an **internal position** (on the tape) and an **external position** (in the world).

- At each transition, a Turbot can perform either an internal operation, or an external operation, but not both.

- When a Turbot is performing an internal operation, its external position remains unchanged; when it is performing an external operation, its internal position remains unchanged.

- Turbots start with a blank tape (unless otherwise noted).

- Here are the operations available to Turbots:

Internal:

Input	**Output**
0 = read 0	0 = write 0
1 = read 1	1 = write 1
* = read *	* = write *
	R = move right
	L = move left

External:

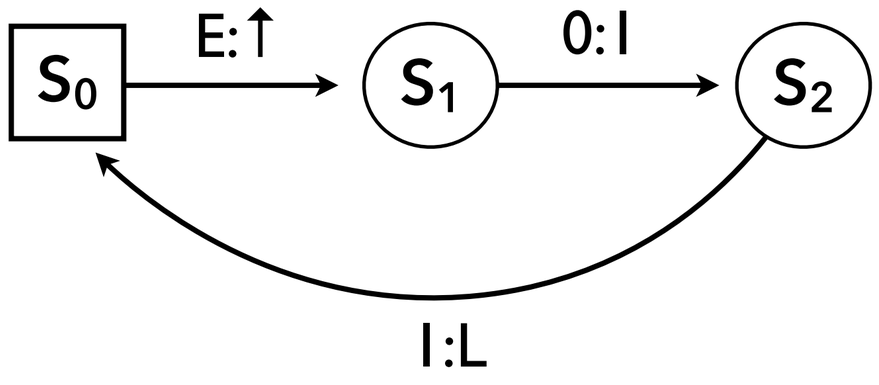
Input	**Output**
B = see block	↑ = move forward
E = see empty	↱ = right turn
F = see food	↰ = left turn

- By convention, states are either internal or external, but not both. External states are **square**, internal states are **circular**.

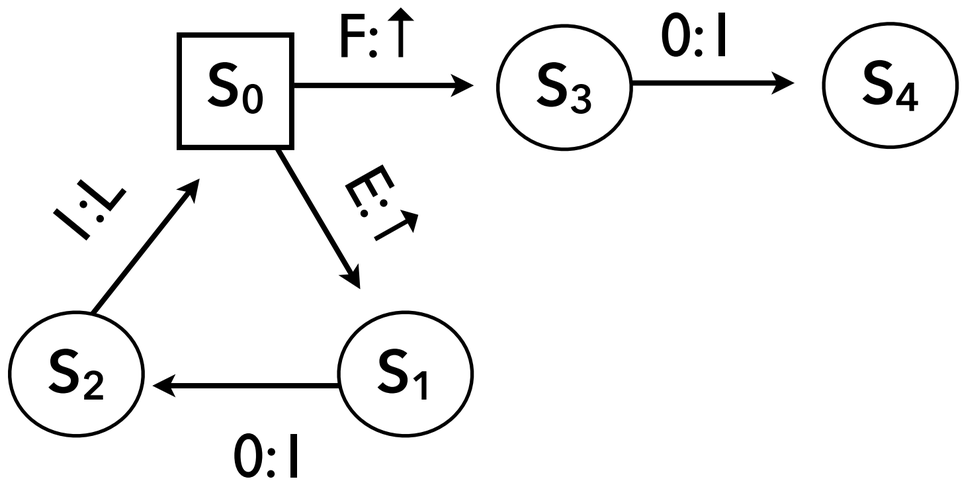
- The Turbot **cannot** pass through blocks. It **can** pass through food.

- Internally, a Turing Machine only perceives the cell it is **currently reading**.
- Externally, a Turbot only perceives the cell **in front of it**.

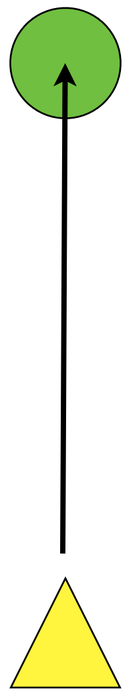
- This turbot moves forward, and records a "1" each time it does so.



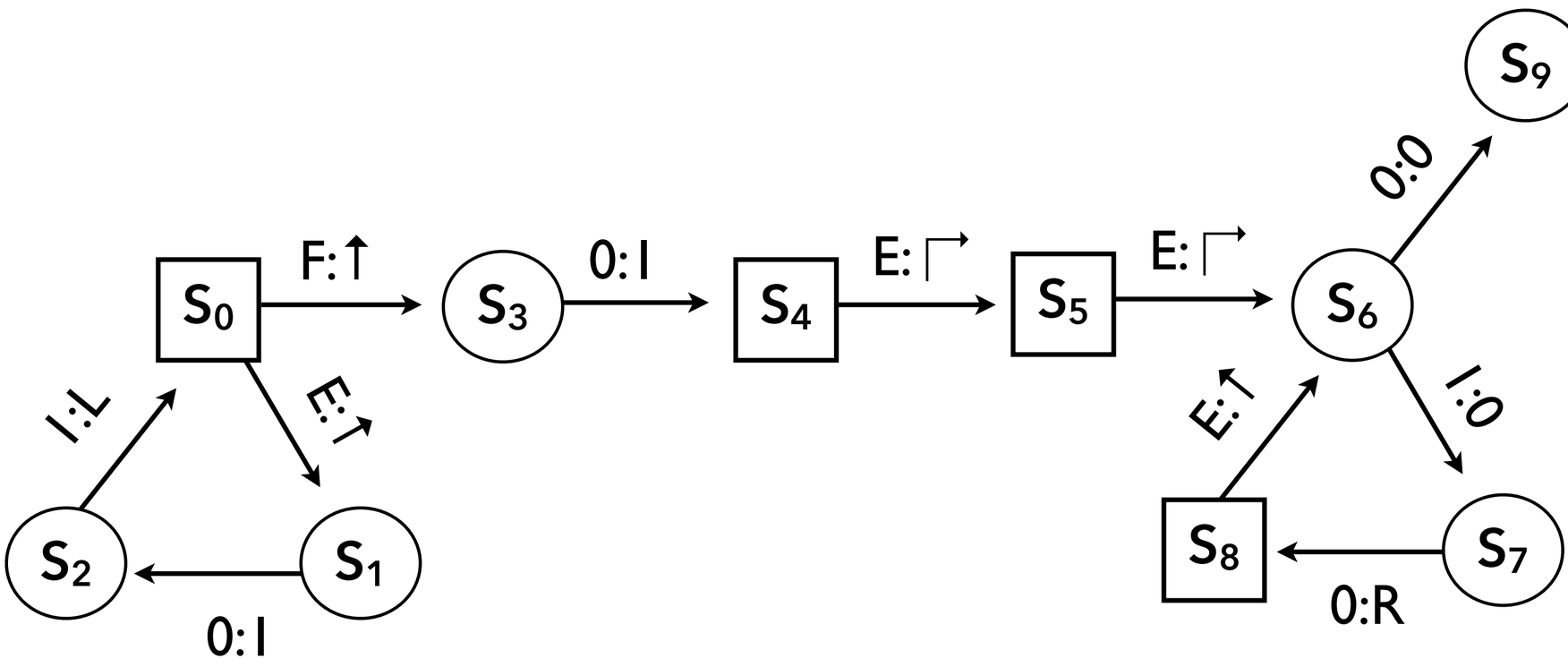
Half-Max Machine



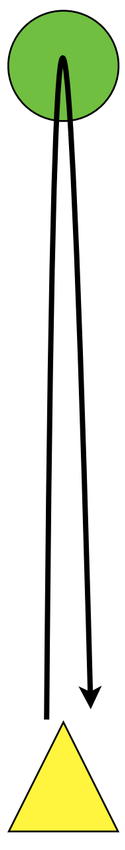
Move to food directly in front of you, recording a "1" each time you move forward, then halt.



Mad-Max Machine



Move to food directly in front of you, but an unknown distance away, then return to the starting point and halt.

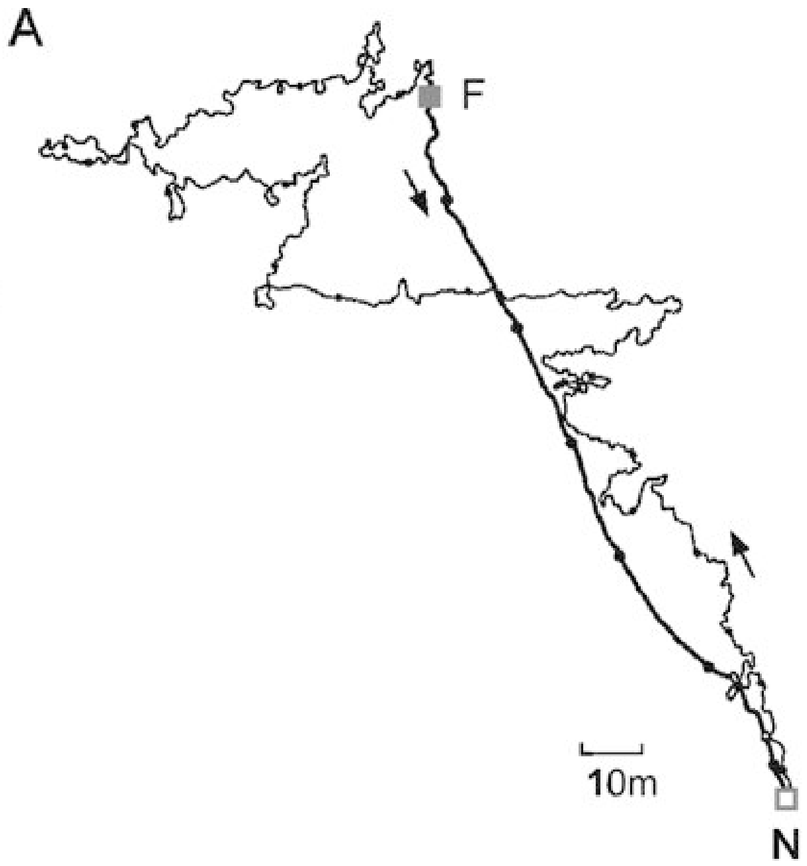


35. The Desert Ant

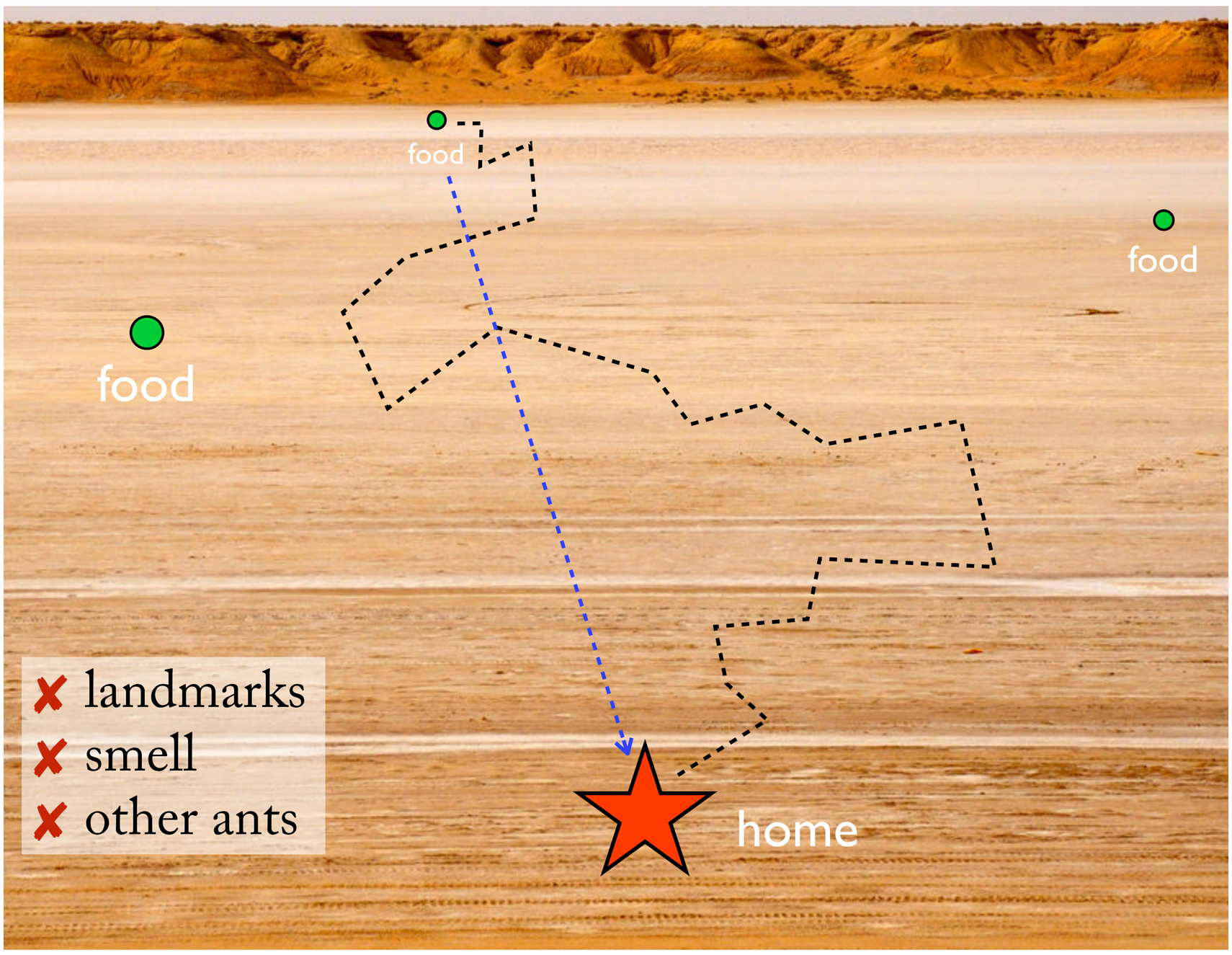
The Ant

250,000 neurons

The Journey

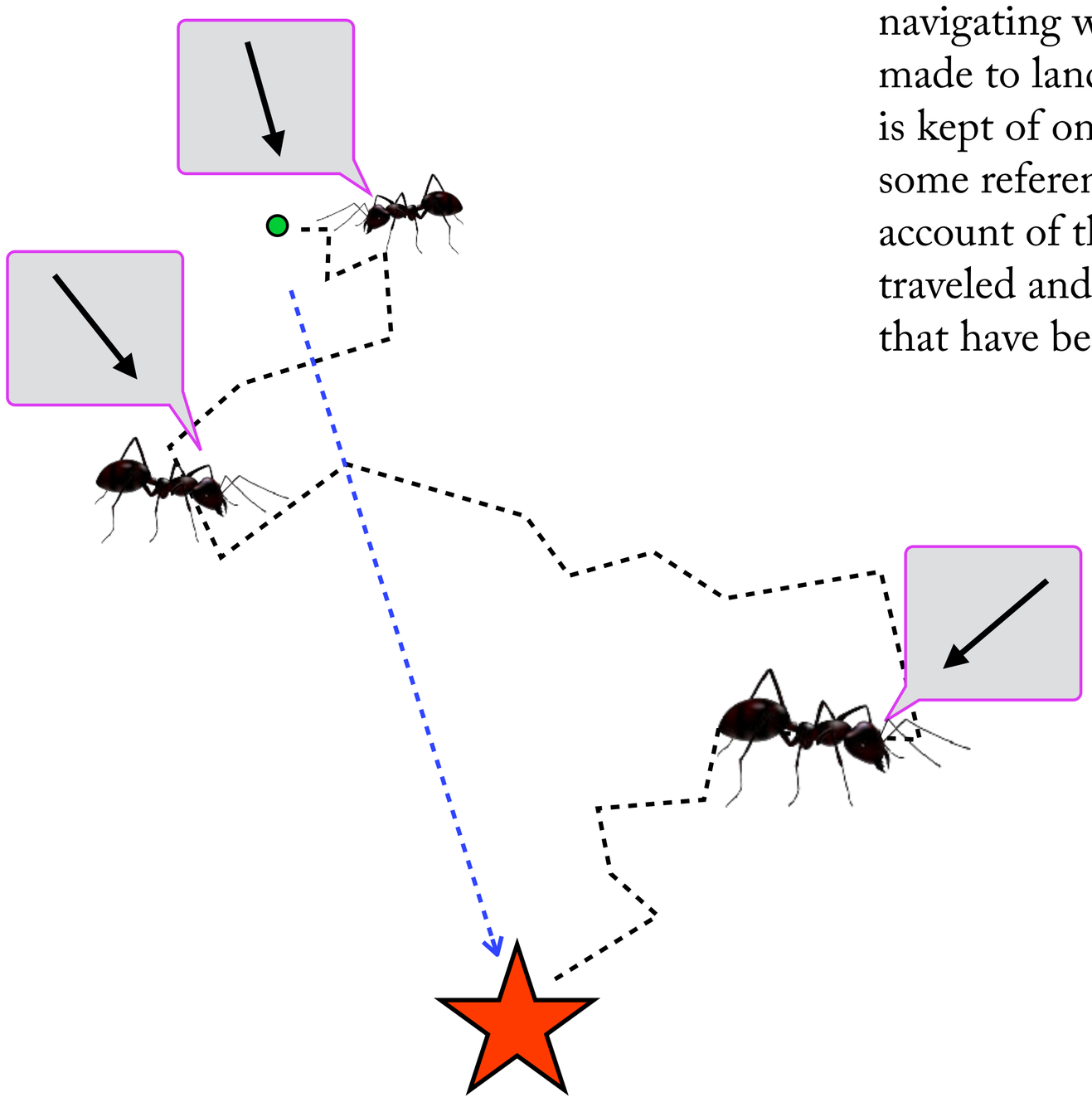


"Scientists observed that instead of retracing their winding route back home, they head straight back to the nest. How does the ant keep track of its current location relative to home? Path integration is the name given to the process thought to be used by animals (human and non-human alike) for what some might call dead reckoning." — SA



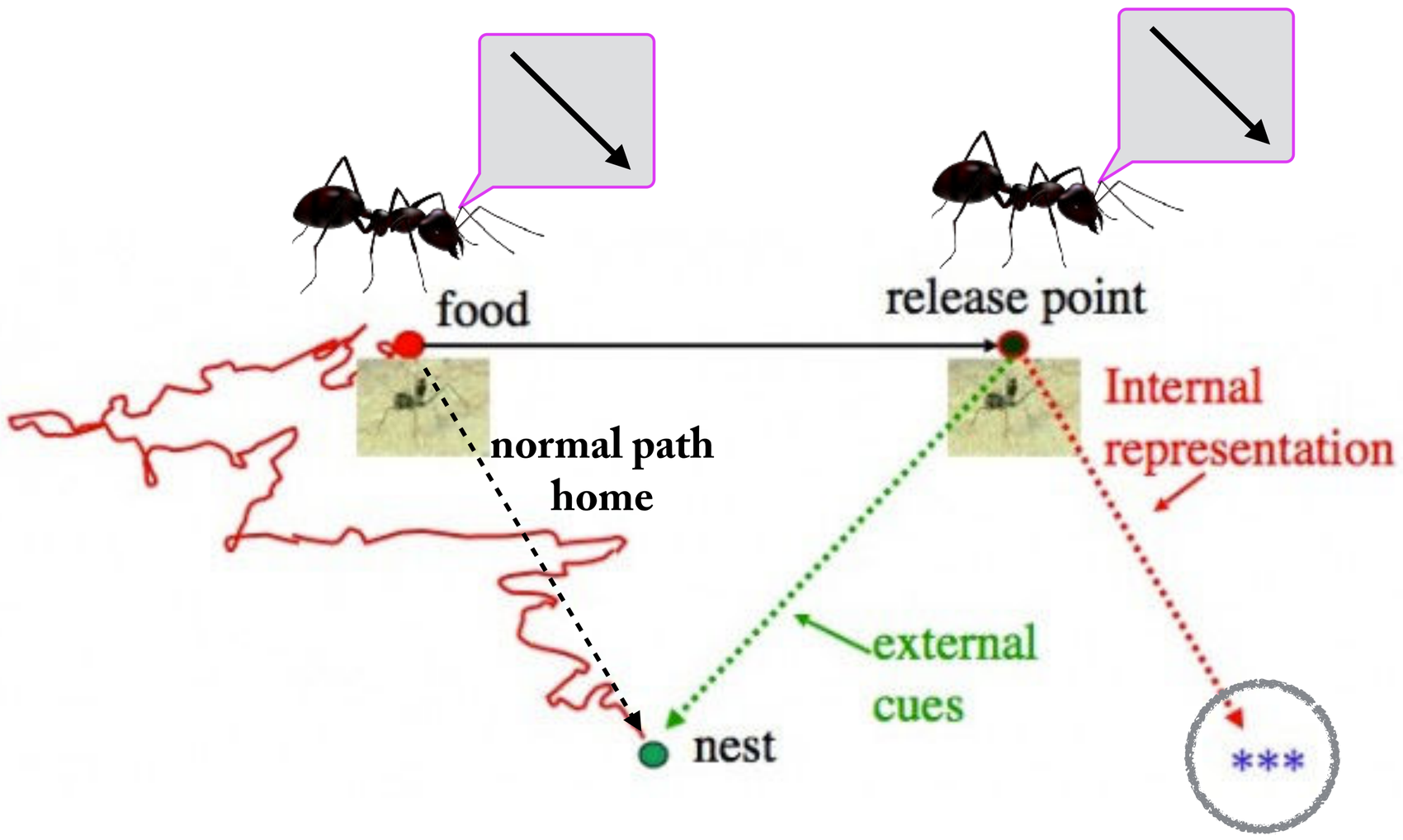
Ant = ~ 0.5 cm
Travel = >150 m
= over 20,000x
their length

The Representational Hypothesis

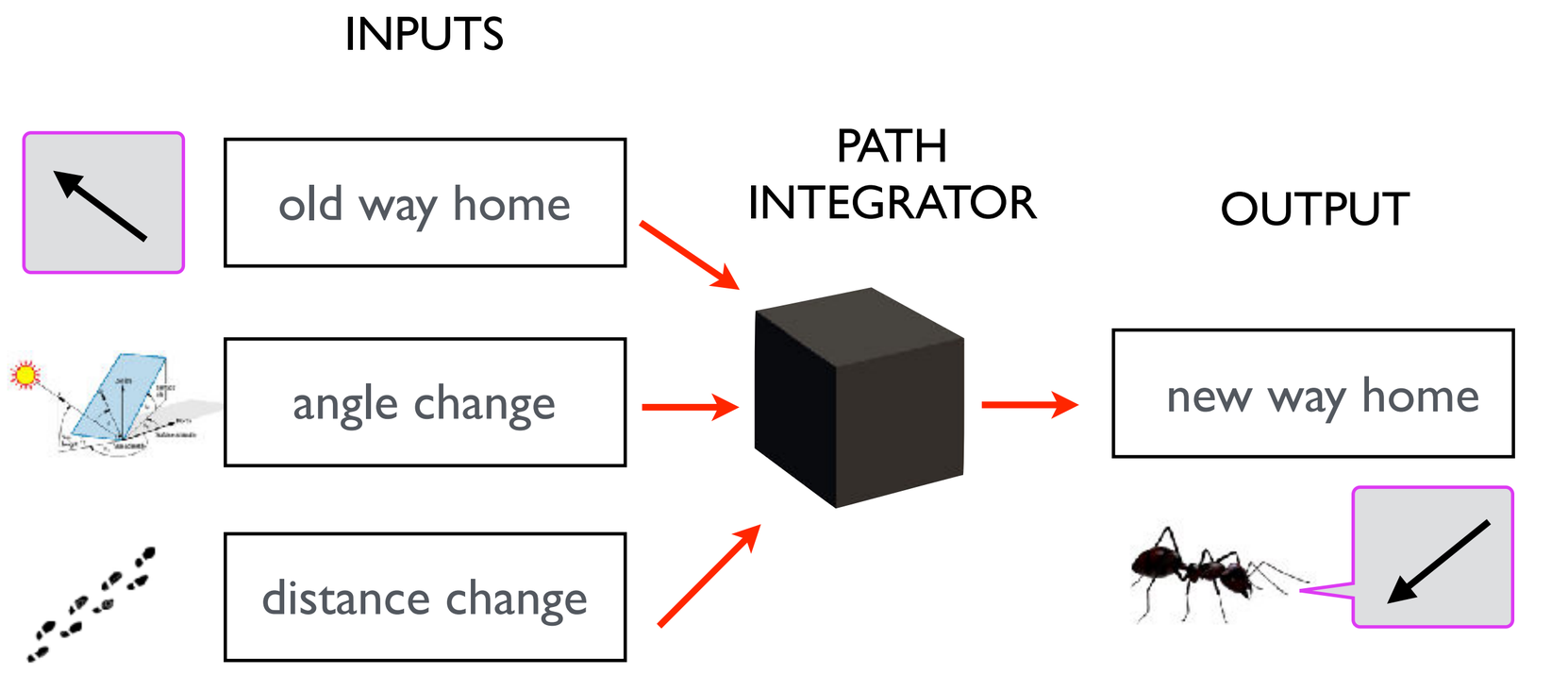


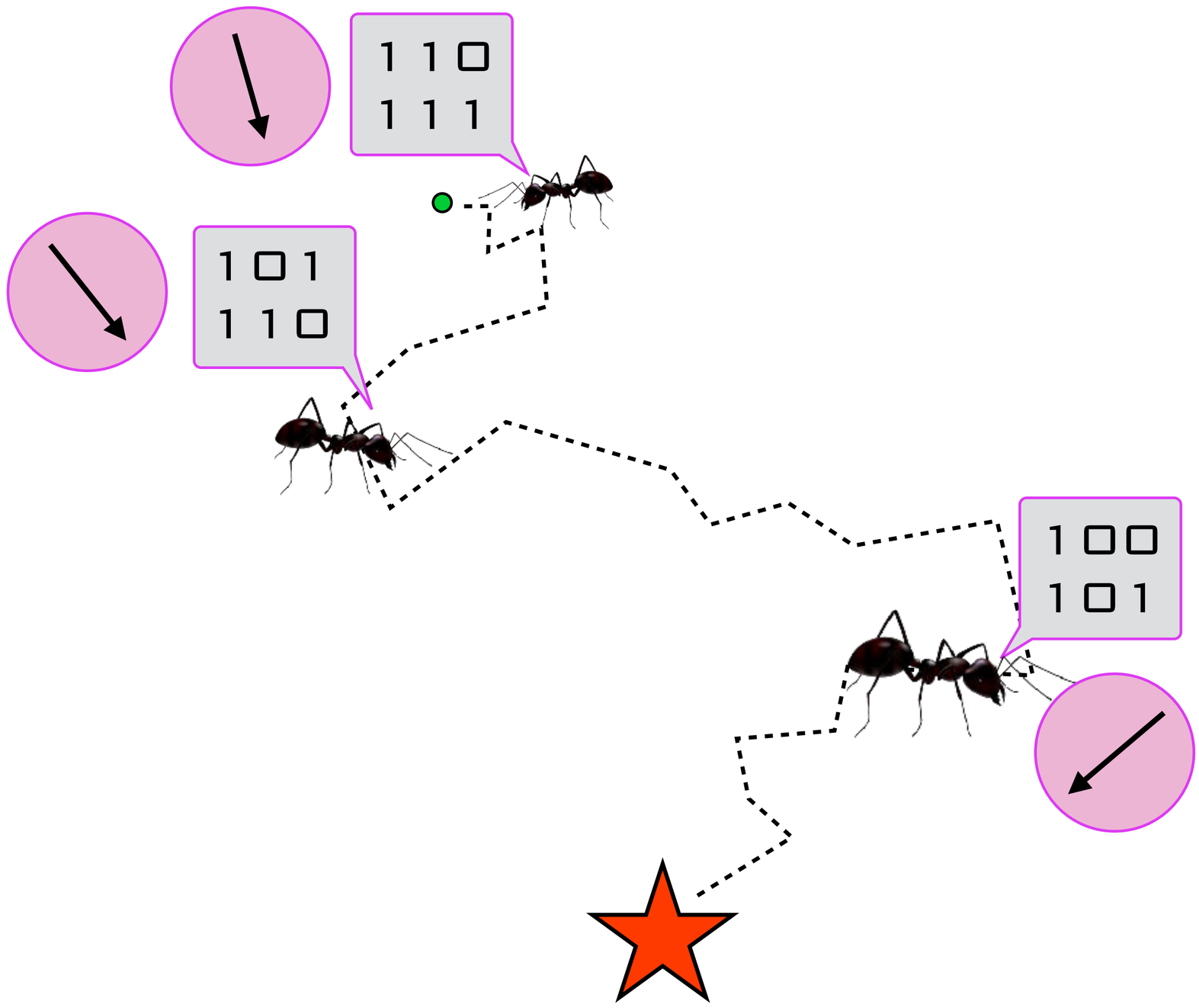
"**Dead reckoning**… is the method of navigating where no reference is made to landmarks. Instead, a record is kept of one's position in respect to some reference point by taking account of the distance that has been traveled and the changes in direction that have been made." Pearce 2008

Testing the Representational Hypothesis



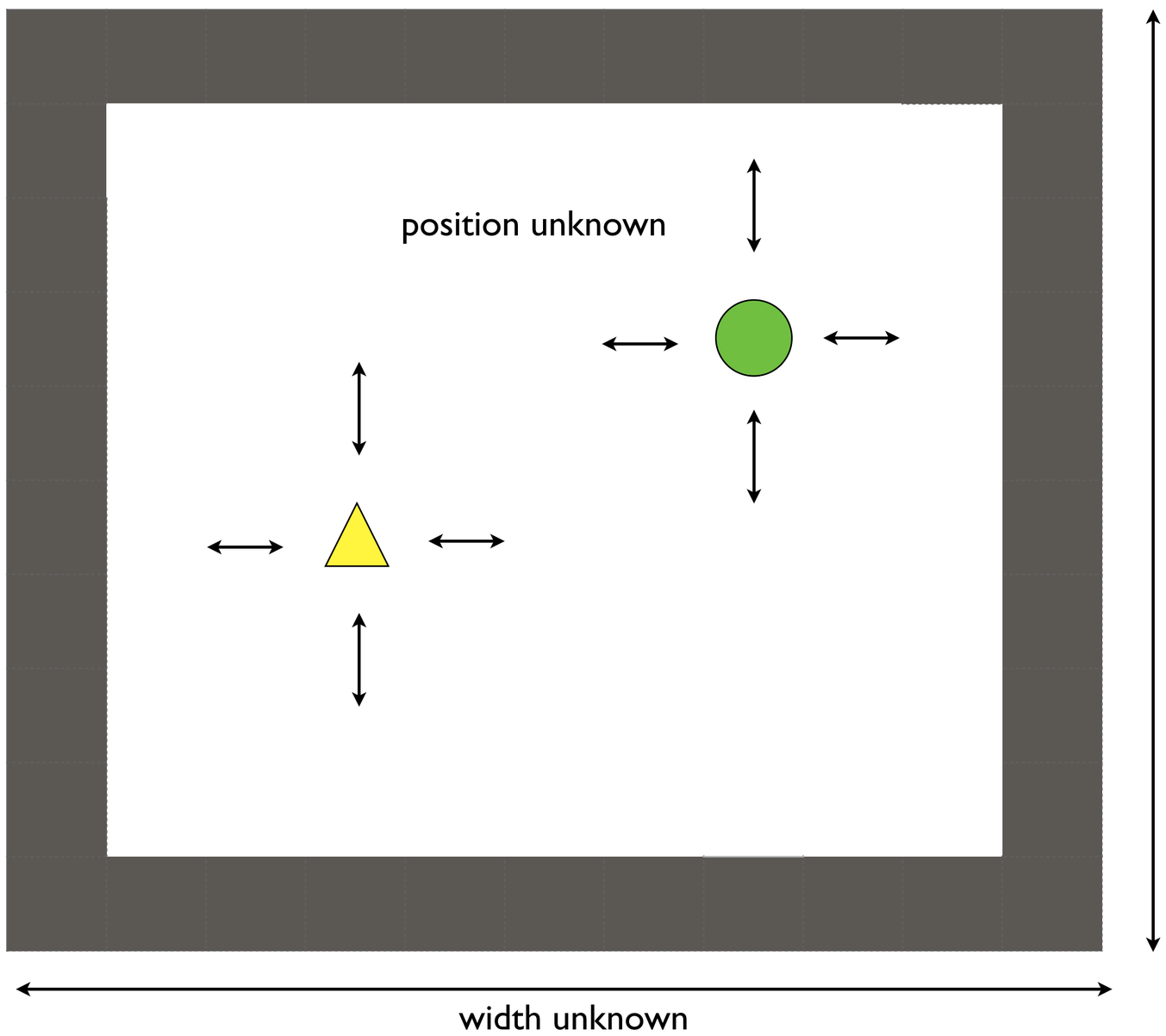

Which way home?
Direction
Distance
Skylight Compass
Step Counter
13914


INPUTS
old way home
angle change
distance change
PATH
INTEGRATOR
OUTPUT
new way home



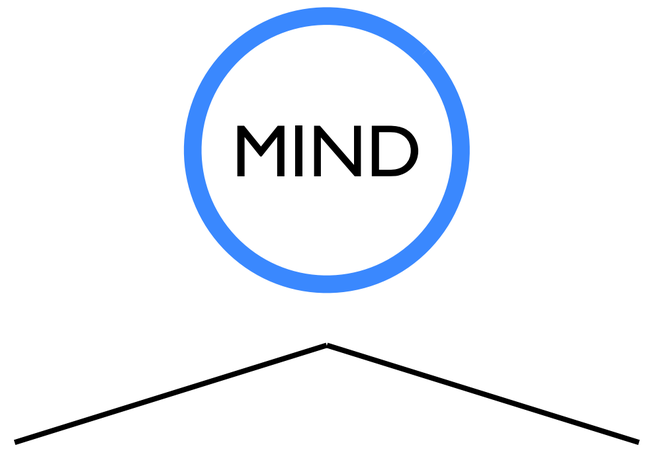
The Desert Ant

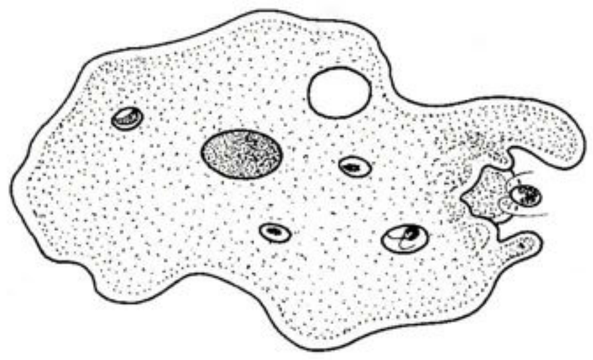
First, search until you find food (are placed over it). Second, return to where you started as efficiently as possible.

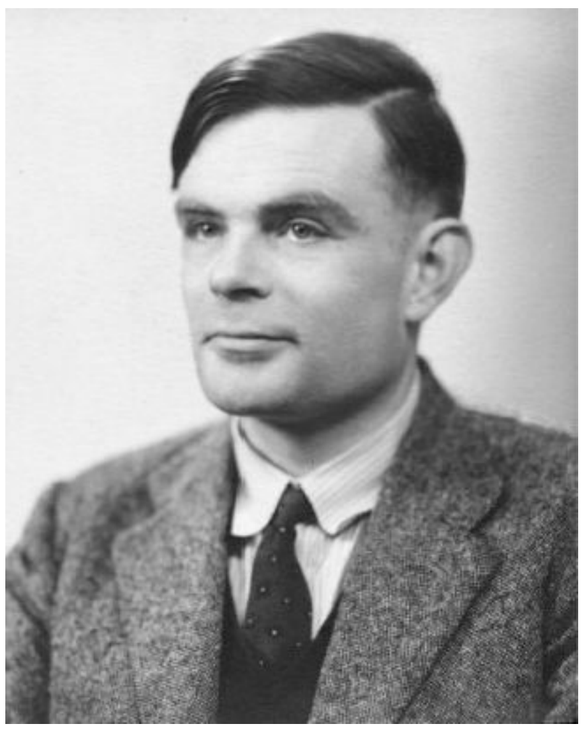


36. Physical Cognition

"A belief … is a map of neighbouring space by which we steer."
—Ramsey 1931



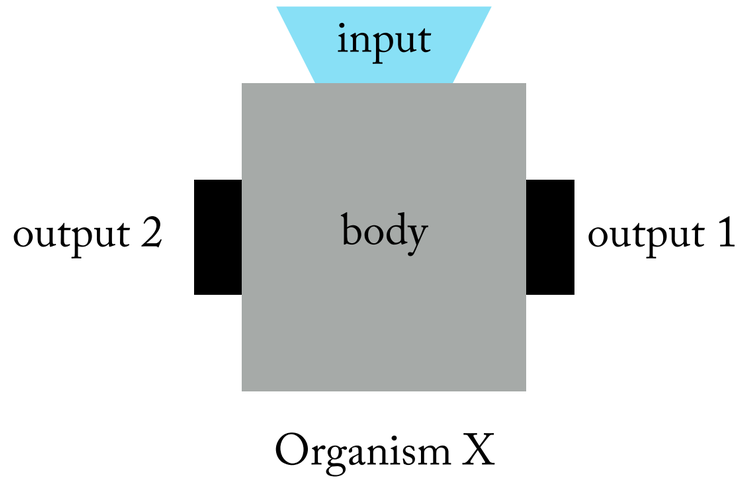




From Object to Animal

Suppose we found a strange object in the woods and attempted to understand its behavior.

After much experimentation, we determine that it is some sort of input-output system. Perhaps it is even an animal of some unknown kind. We'll call it Organism X.



Organism X

To study it's behavior, we schematize it's overall organization into four basic components: an input organ, two output organs, and a body connecting them.

Does Organism X have cognition? Does it think? Have beliefs? Consciousness? To put it to the test, we stimulate the input organ with a series of signals over time and record the output. Here are some snapshots of one of our experiments.

…	t13	t12	t11	t10	t9	t8	t7	t6	t5	t4	t3	t2	t1	
…	0	0	0	0	0	1	0	0	0	0	0	0	0	**IN**
…	11	11	11	01	11	01	11	11	11	11	11	11	10	**OUT**

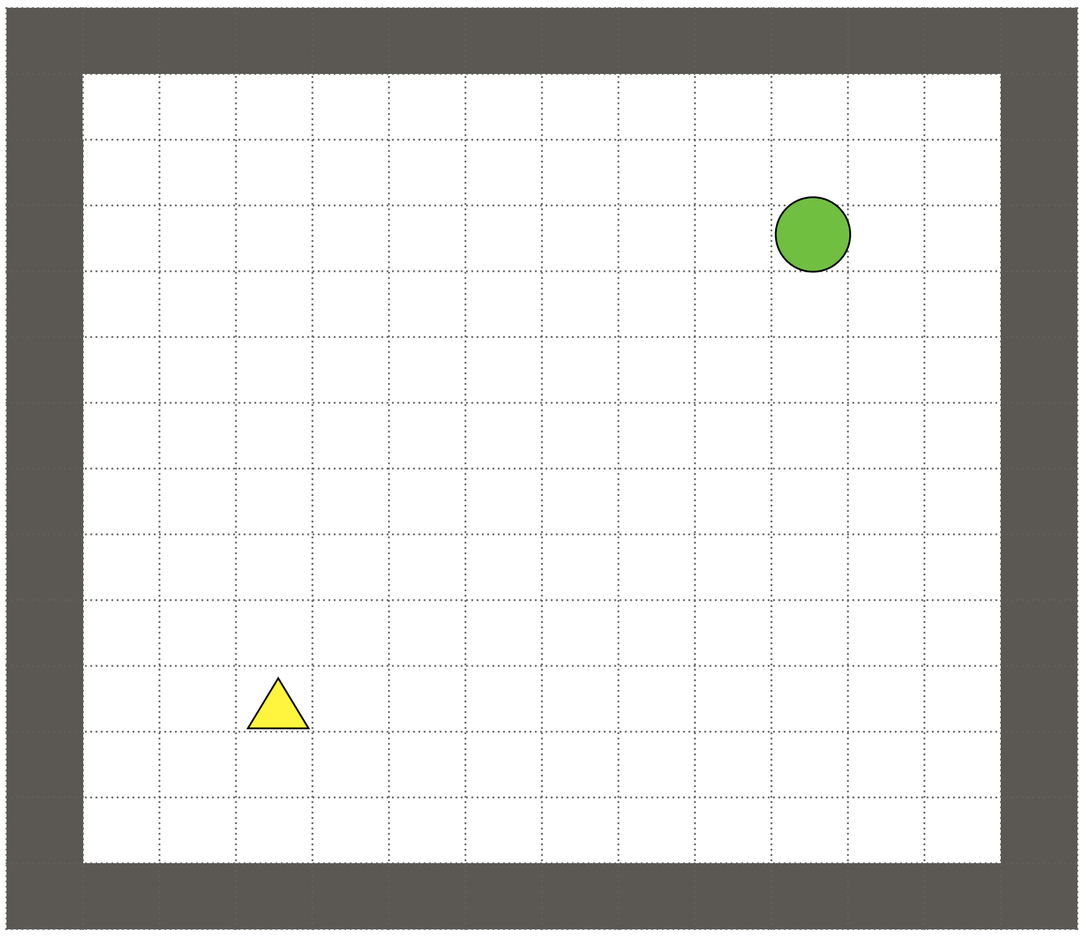
…	t111	t110	t109	t108	t107	t106	t105	t104	t103	t102	t101	t100	…	
…	0	0	0	0	0	0	*	0	0	1	0	0	…	**IN**
…	11	11	11	11	11	01	11	11	11	10	11	11	…	**OUT**

…	t198	t197	t196	t195	t194	t193	t192	t191	t190	t189	t188	t187	…	
…	0	0	0	0	0	0	0	0	0	0	0	0	…	**IN**
…	00	00	00	00	00	00	00	11	11	11	11	11	…	**OUT**

There is certainly something complex about Organism X's reactions. But are they random or rational? Is it trying to tell us something? Do the inputs *mean anything* to it?

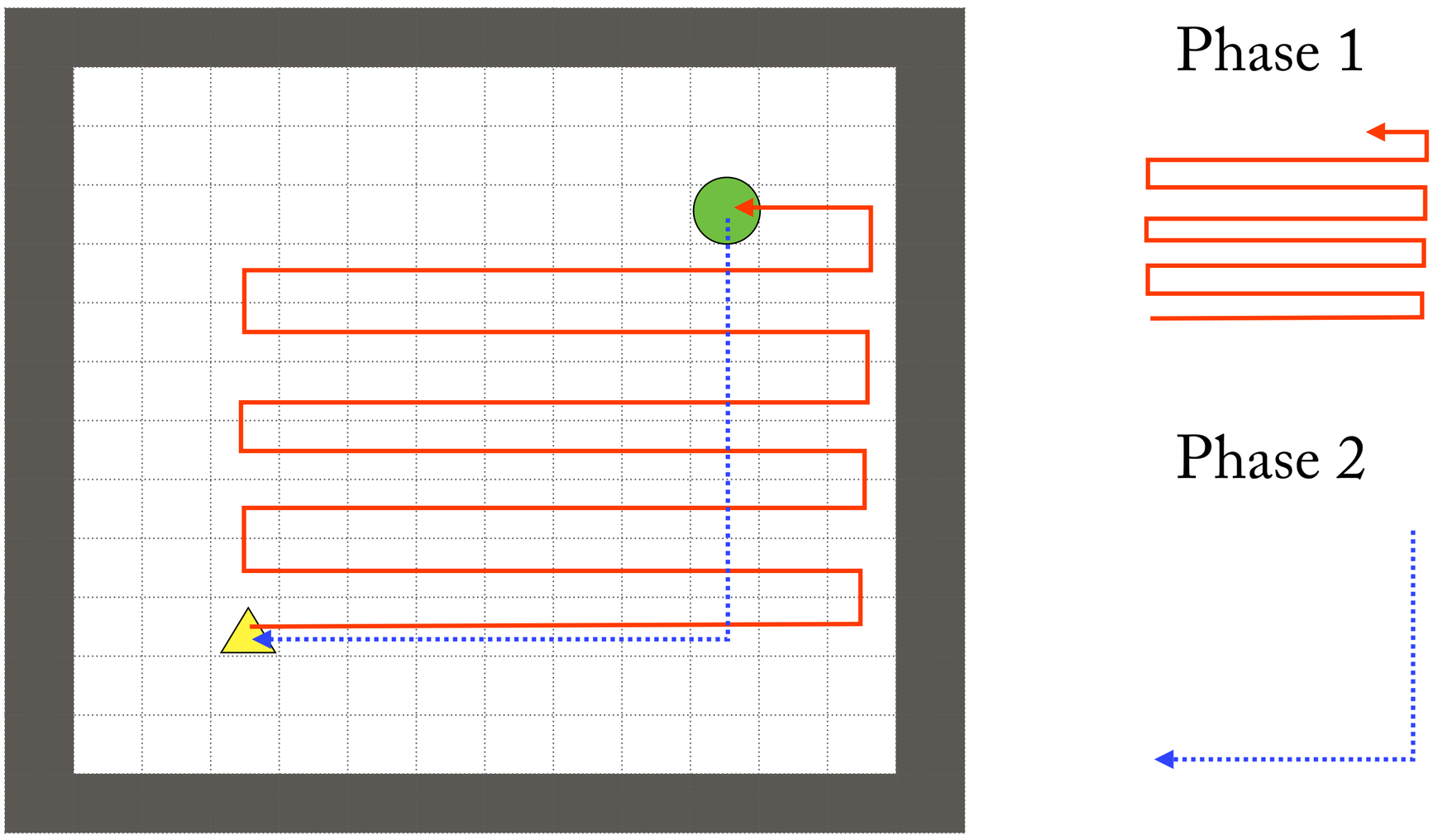
The Ecological View

We consult with a cognitive scientist, who gives us some wise advice: you won't find any answers to your questions unless you look at the **ecological environment** in which Organism X evolved. Only then can we attempt to decode its inputs and outputs.



Consulting with some naturalists, it turns out that X's tend to live in enclosed spaces, where they have single food source— a kind of plant that lives only a single day— and no predators. In other words, the X ecology looks a bit like this.

Observed Behavior



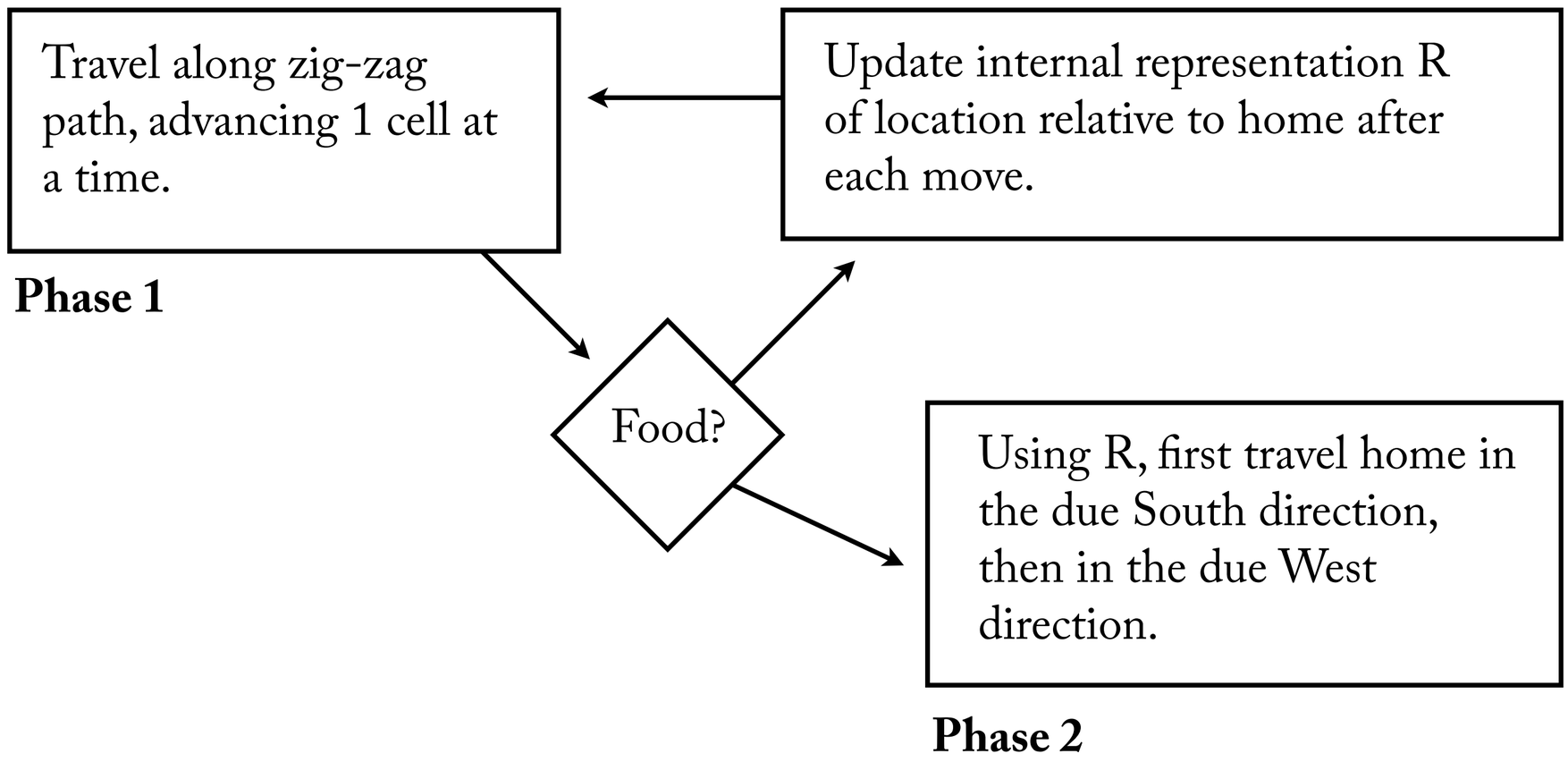
Evolutionary Analysis

Phase 1:
Exhaustive search for food through NE quadrant.

→

Phase 2:
Return home as efficiently as possible.

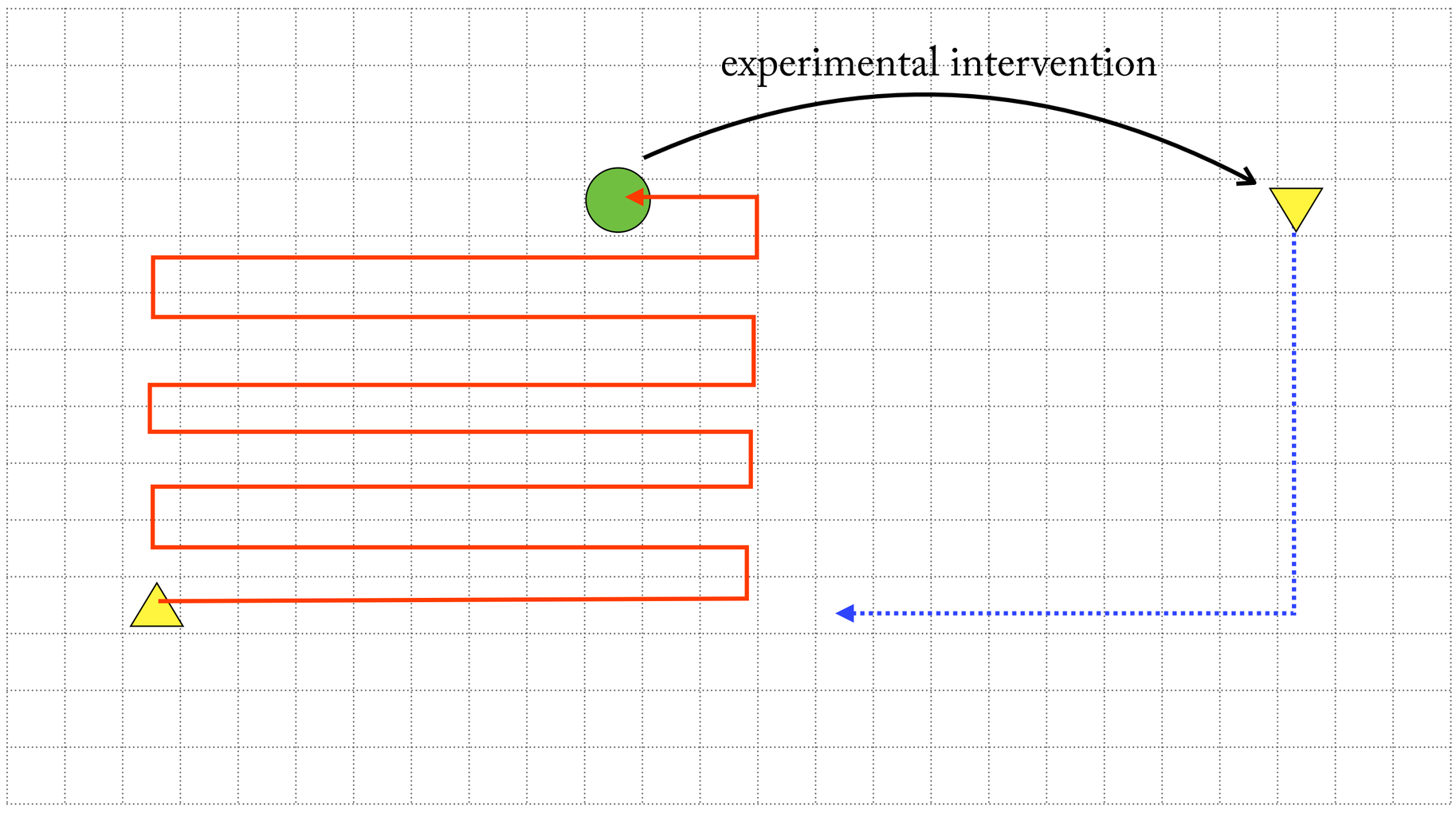
The Representational Hypothesis



"What makes this of interest to cognitive scientists is that it suggests that the brain of an insect is a symbol processing organ. It suggests that the brain uses computation to construct a cognitive map and to compute courses from the positional values stored in that map. The positional data stored on a map together with the computations that utilize those data to set courses constitute a **representation**, a model of the world used to determine courses of action in that world. Thus, the homing abilities of insect might be taken as strong evidence in favor of a computational-representational theory of mind."

Testing the Representational Hypothesis

An experiment with Turbots: evidence for mental representations.



Neuroscience

Next we capture a Turbot, bring it to the lab, and look inside. When we inspect the tape, this is what we find:

*	0	0	0	0	1	*	0	0	0	1	0	*

We capture other Turbots, find variations of our first result.

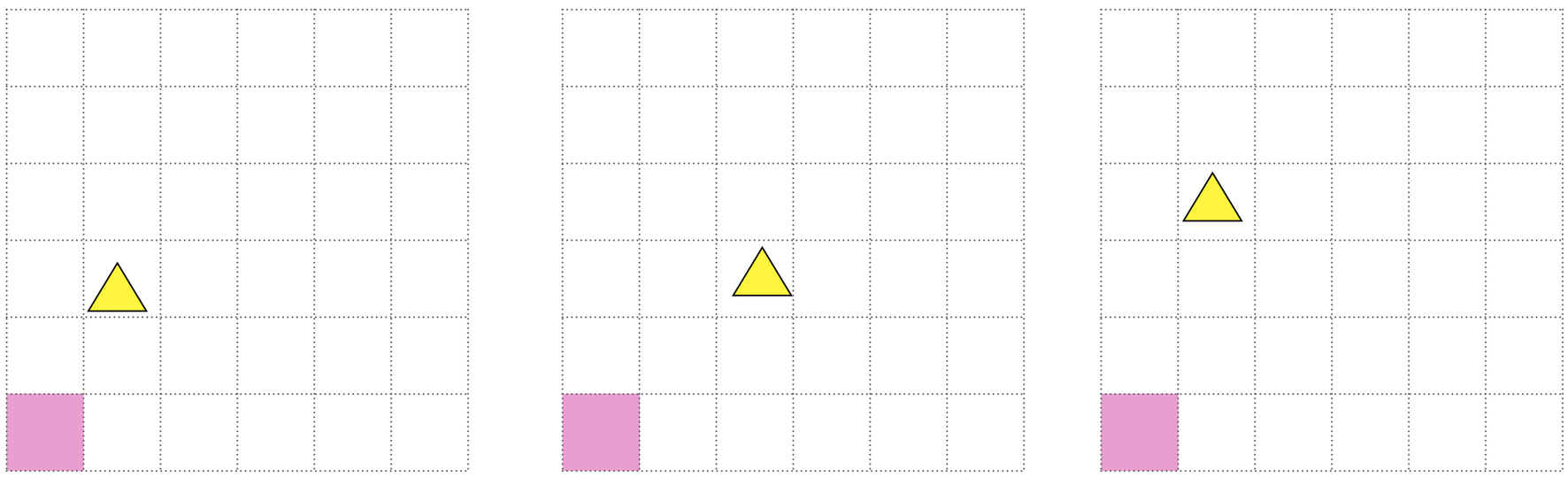
*	0	0	0	0	1	*	0	0	0	1	0	*

*	0	0	0	1	0	*	0	0	0	1	1	*

There are clearly some stable patterns here, but also variation. What does this all mean?

Cognitive Science

After testing several different hypotheses, someone in our lab comes up with the idea that the mark on the tape are correlated to the X- and Y- position of the Turbot. This makes sense from an evolutionary perspective. So we capture three different Turbots at different stages of their search for food.



00001 00010 00010 00010 00001 00011

We still don't really know what these marks *mean*, but we have a clue. When the Turbot moves in the X- direction, the left-hand block changes; when the Turbot moves in the Y-direction, the right-hand block changes. Otherwise they stay the same. This gives rise to the following hypothesis:

*	0	0	0	0	1	*	0	0	0	1	0	*

records X-position records Y-position

Externalism

According to **externalism,** for an internal symbol S to be a genuine mental representation of X, it must satisfy two criteria:

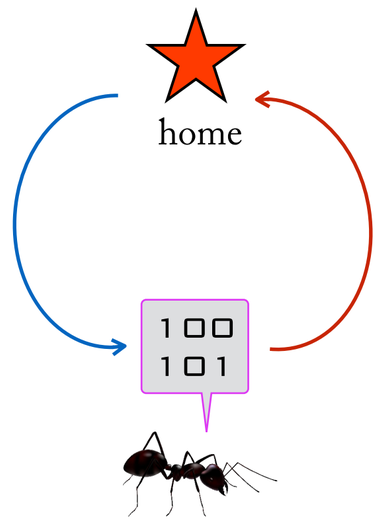
(1) it must function to mirror X in the world;

(2) it must be guided by perception (of X) and guide action (towards X).

When these criteria are met, mental representations are capable of **truth** and **falsity.**

Navigational symbols have meaning for the ant because:

- internal symbols c**arry information** about the direction home.
- internal symbols **guide the action** of the ant with respect to the direction home



Information semantics

Mental Representation

Mirroring

Further study reveals that the symbols in the Turbot memory systematically covary with the physical distance between the turbot and home. In fact, when the system is working properly, the following generalizations hold:

binary(S_x) = the number of steps **east** between **here** and **home**.
binary(S_y) = the number of steps **north** between **here** and **home**.

Mediating action and perception.

Truth and Falsity

This leads the idea that these symbols are **mental representations:** internal symbols which represent the "beliefs" or "cognitive commitments" of the Turbot. At any given time, the Turbot has the "beliefs":

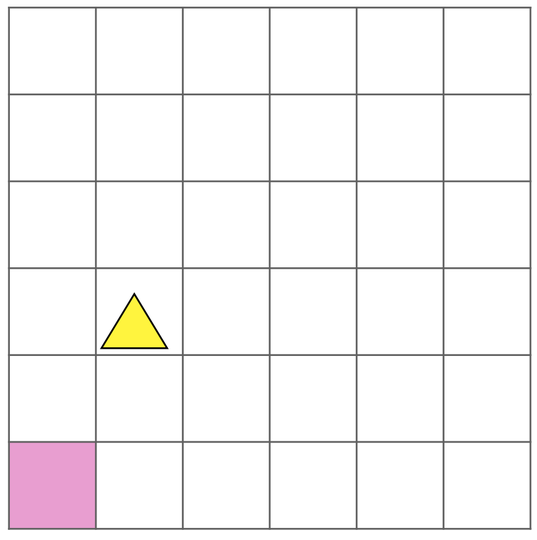
The number of steps **east** between **here** and **home** is binary(S_x).
The number of steps **north** between **here** and **home** is binary(S_y).

These beliefs may be **true** or **false**. As we saw in our original displacement experiment, it is clear that the Turbots follow their internal symbols, *as if* the equations above were true, even when they are not.

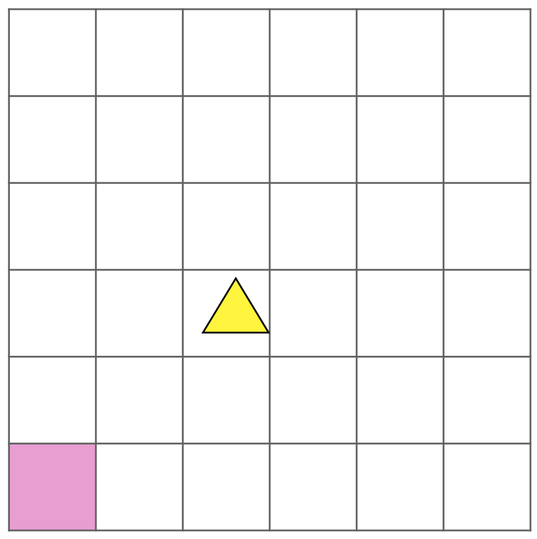
Internal world

External world

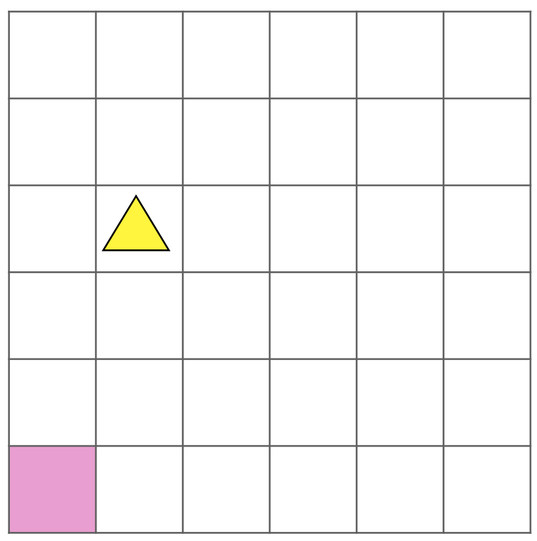
00001 00010



00010 00010



00001 00011



"A *mental representation* is a functioning isomorphism between a set of processes in the brain and a behaviorally important aspect of the world…"

Mental Representation

Further study reveals that the symbols in the Turbot memory systematically covary with the physical distance between the turbot and home. In fact, when the system is working properly, the following generalizations hold:

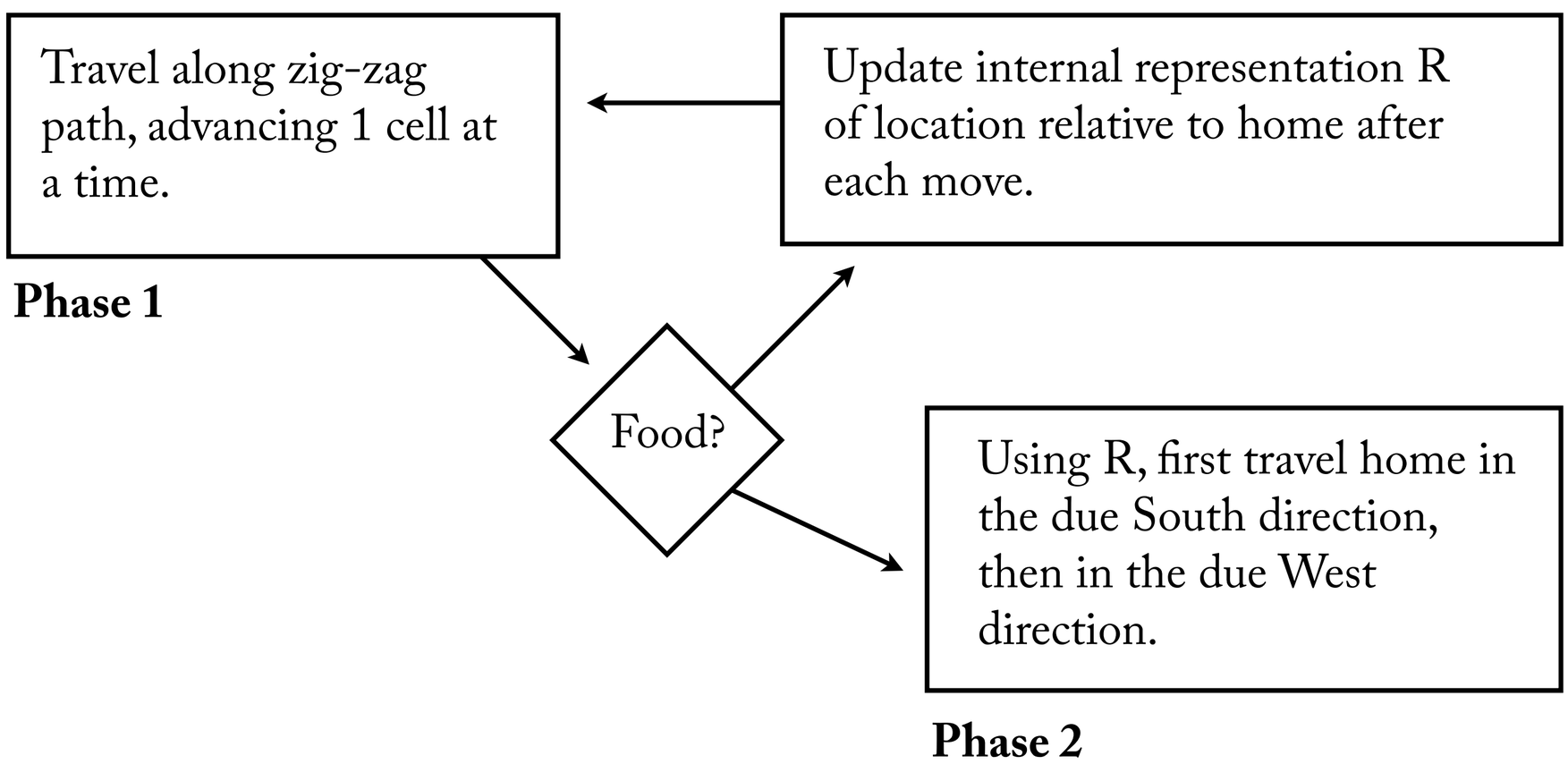
$\text{distance}_x(\text{here, home}) = \text{Bin}(S_x)$
$\text{distance}_y(\text{here, home}) = \text{Bin}(S_y)$

Yet, as we saw in our original displacement experiment, the system is not always working properly. Still, it is clear that the Turbots follow their internal symbols, *as if* the equations above were true, even when they are not.

This leads the idea that these symbols are **mental representations:** internal symbols which represent the "beliefs" or "cognitive commitments" of the Turbot.

Representational content (at a time) = $\text{distance}_x(\text{here, home}) = a$ & $\text{distance}_y(\text{here, home}) = b$

where $a = \text{Bin}(S_x)$ and $b = \text{Bin}(S_y)$



Mirroring the world. These representations can be thought of creating an internal way of "tracking" or "mirroring" an external phenomena.

Truth and falsity. The machine misrepresents the environment when the content of its symbols is false.

"A *mental representation* is a functioning isomorphism between a set of processes in the brain and a behaviorally important aspect of the world…

"Is the brain a symbol-processing system? The rise of cognitive psychology in the sixties reflected the emergence of the (then novel) view that the brain was a symbol-processing system analogous to a computer, and that it used its symbol-processing capacity to construct representations of the environment in which it functioned."

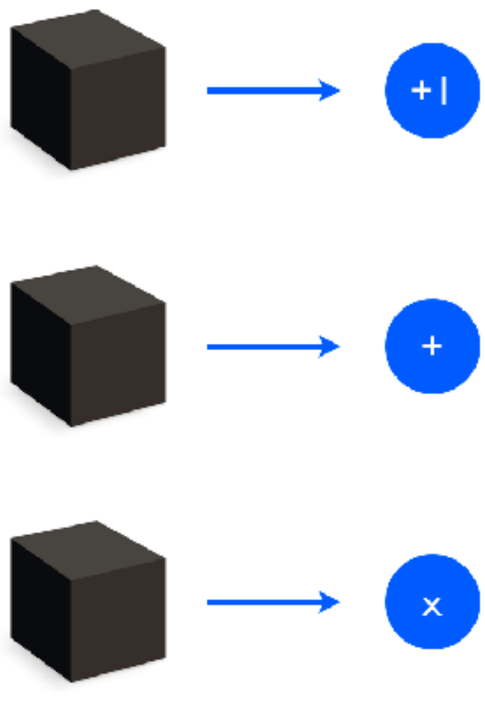
Suppose that we discover some Turbots in the wild, and we attempt to describe them as a cognitive scientist would.

Suppose further that they are the products of natural selection: over millions of cycles of reproduction, mutation, and selective pressure, our turbots have evolved. We assume that:

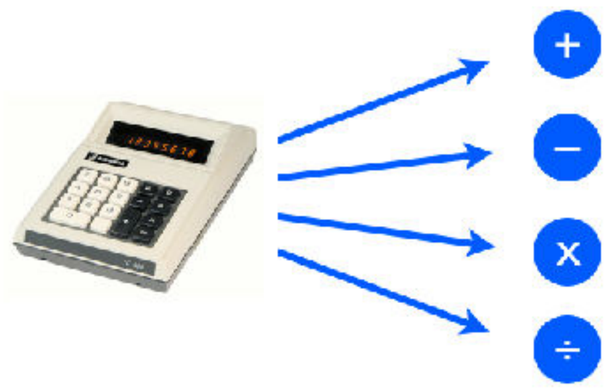
- **The environment** has stayed the same over the course of natural selection (it always involves the surrounding walls, and the placement of food).
- **Reproduction** involves the duplication of machines in similar environments.
- **Variation** between machines involves random permutations of logic gates or tape configurations.
- **Selective pressure**: if a turbot doesn't find food and return home within a certain time span, it dies without having a chance to reproduce.

37. Machines and Programs

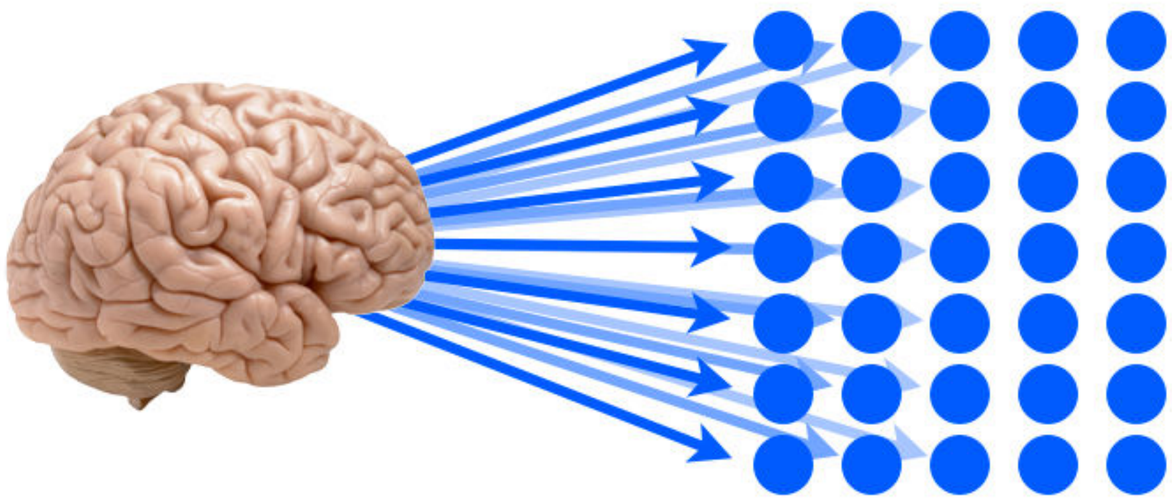
Multi-function Machines



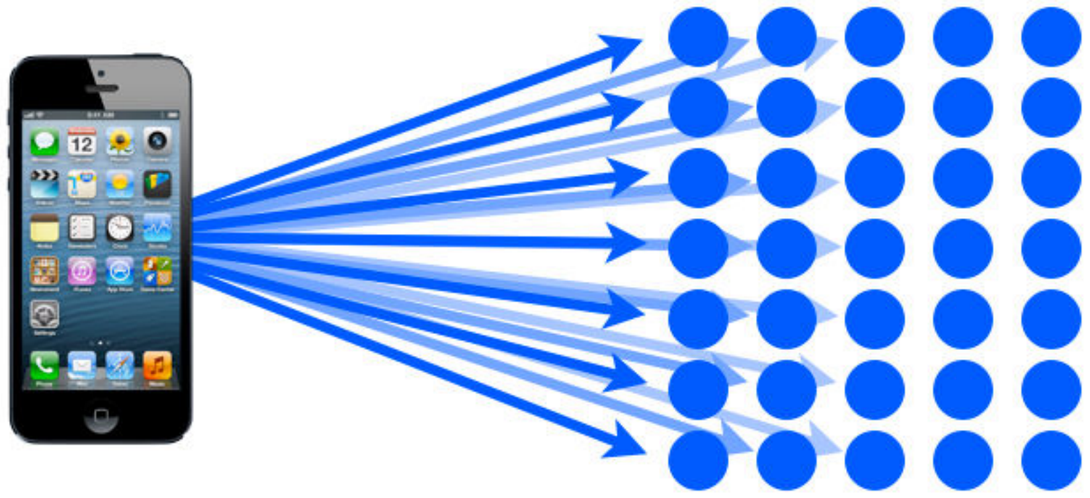
So far, all of the machines we have encountered compute one function each. Of course, it is possible to construct a machine which computes more than one function. A simple calculator computes four functions.



Our brains can compute alot more than four functions. In fact, it seems like there is no practical limit on the the number of functions a brain could compute. It is an *infinite* function machine.

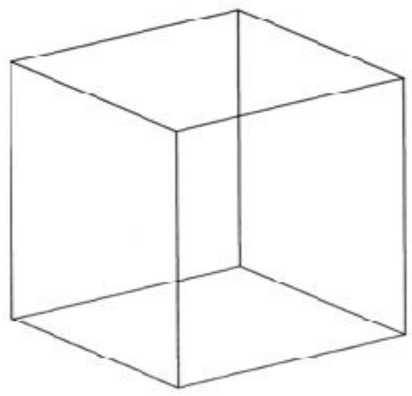


This isn't just true of our brains. It's true of any computer which can read *programs*. Such devices are known as **universal computers**. Your laptop and your phone are universal computers.



Universal Turing Machines: the goal

The goal: Create a Turing Machine which can perform the task of any other Turing Machine. This is called a **Universal Turing Machine** (UTM).



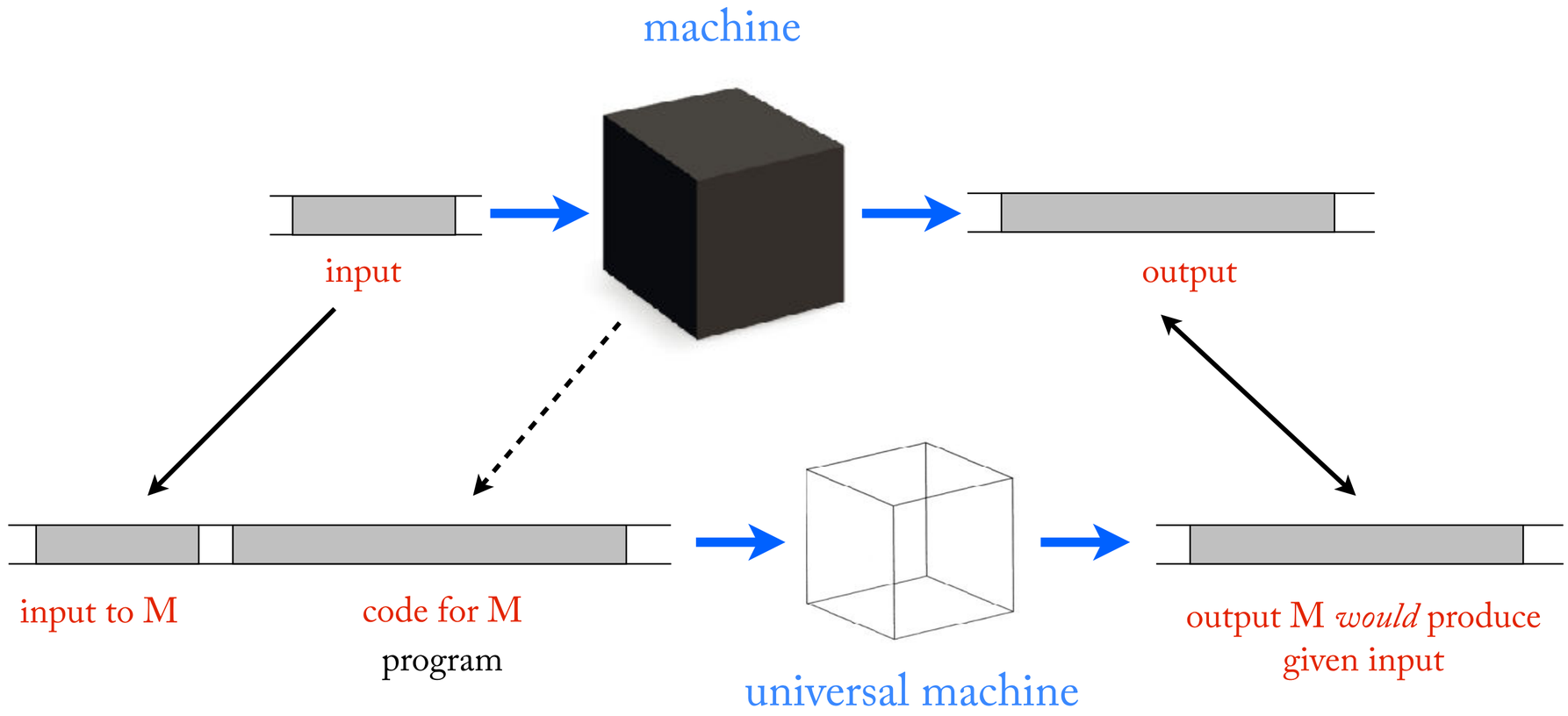
universal machine

Universal Turing Machines: the idea

Strategy for designing a UTM:

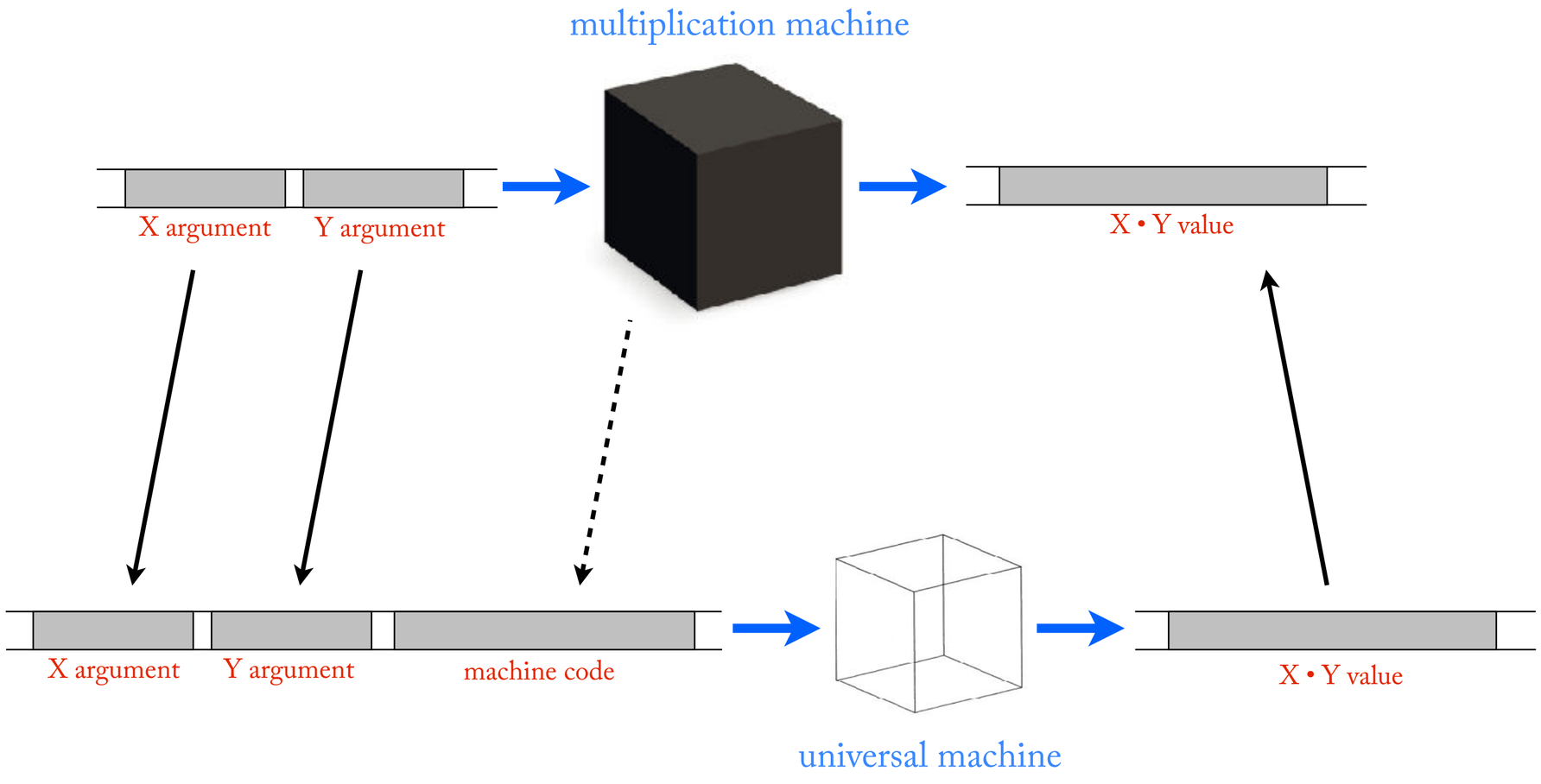
Stage 1: find a method for turning an arbitrary Turing Machine (the simulated machine) into something a Turing Machine can read— i.e. 1's and 0's on tape. This something is called the simulated machine's **code number**.

Stage 2: invent a UTM which takes as input (i) any simulated machine's code number, and (ii) the simulated machine's input; the UTM then outputs the simulated machine's output.



For example, let M be a multiplication machine. It takes two inputs (corresponding to the X and Y arguments) and returns one output (corresponding to the value X•Y). And let's say that M's code number is C.

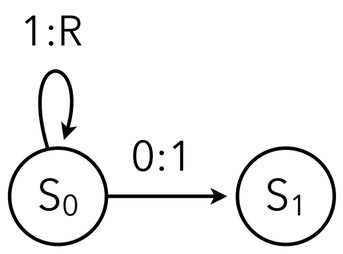
Then if we give a tape which contains, X, Y, *and* C to the UTM, it will return the value X•Y.



Codes Numbers (aka Programs)

Derive a machine's code number in five easy steps.

Step 1. Choose a Turing Machine, and draw up it's state table.



STATE	IN	OUT	NEXT STATE
S_0	1	R	S_0
	0	1	S_1
S_1	1	H	
	0	H	

Step 2. Divide each state cell into rows according to input. Eliminate rows which do not lead to another state.

STATE	IN	OUT	NEXT STATE
S_0	1	R	S_0
S_0	0	1	S_1
S_1	1	H	
S_1	0	H	

Step 3. Replace all symbols with numbers, according to a translation system.

State S_n	⟿	*n*
No state	⟿	0
Read a "0"	⟿	0
Read a "1"	⟿	1
Write a "0"	⟿	0
Write a "1"	⟿	1
Move left (L)	⟿	2
Move right (R)	⟿	3
Halt (H)	⟿	4

STATE	IN	OUT	NEXT STATE
0	1	3	0
0	0	1	1
1	1	4	0
1	0	4	0

Step 4. Represent the table by a sequence of quadruples:

state, in, out, next
state, in, out, next
etc...

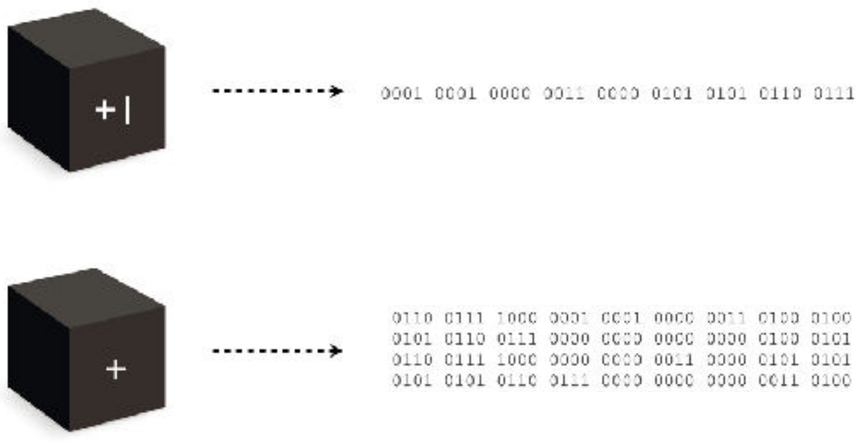
0, 1, 3, 0
0, 0, 1, 1
1, 1, 4, 0
1, 0, 4, 0

Step 5. Represent each number in tally. Represent commas with 0's and line breaks with *s. The result is a string 0's and 1's.

001011100*000010
1*10101111100*100
0111100

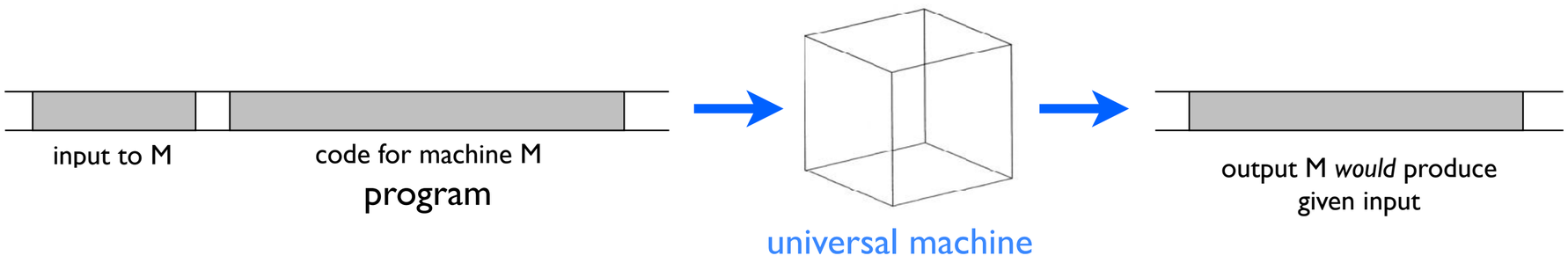
Now every single Turing Machine can be associated with a code number. And every single (well-formed) code number corresponds to a Turing Machine.

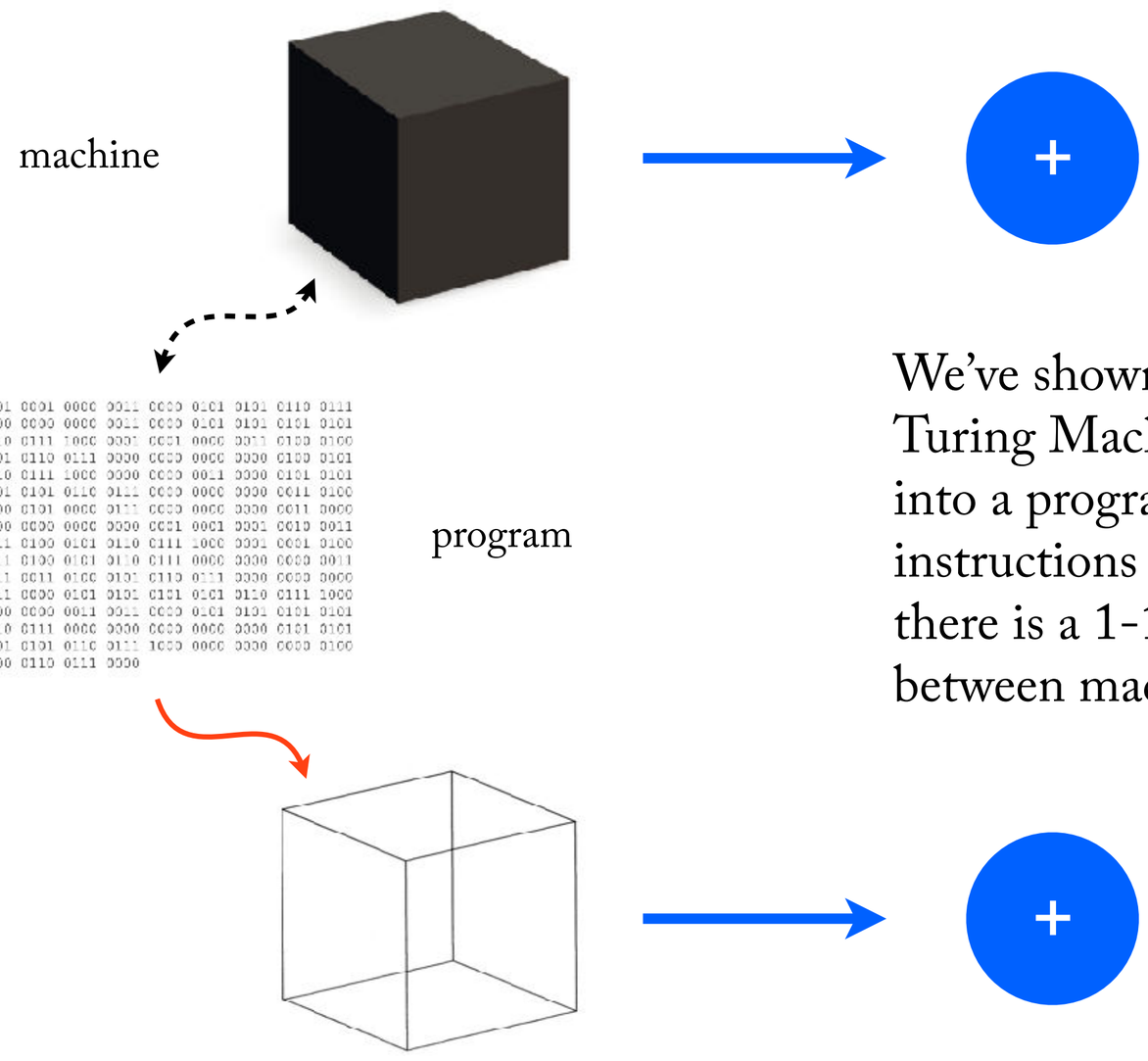
These code numbers will now become the programs which are given to the Universal Machine as input.



Machines as Programs

The code for a machine is also known as its **program**. A program is simply an algorithm written down in a language that a universal machine can interpret.





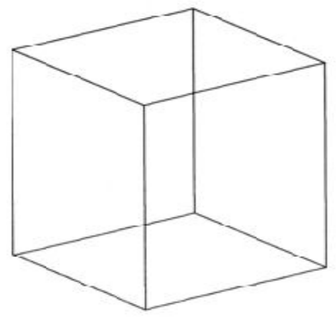
We've shown that any given Turing Machine can be turned into a program which serves as instructions for the UTM. So there is a 1-1 correspondence between machines and programs.

Up until now algorithms have been used to describe machines at a higher level of abstraction. Now they are used as something which is inputted *into* the machine. The universal machine, in effect, *reads* higher levels of abstraction.

38. The Universal Machine

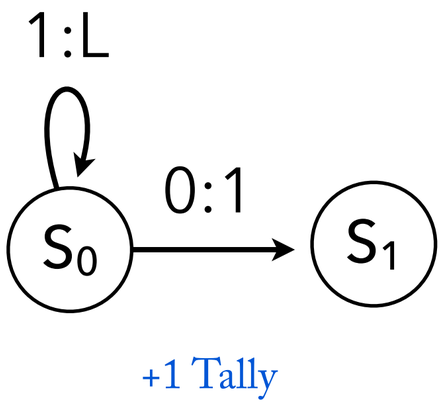
The Idea

The goal: Create a Turing Machine which can perform the task of any other Turing Machine. This is called a **Universal Turing Machine** (UTM).



universal machine

Step 1: Identify a method for translating a machine into a **code number**, which will function as the **program** for that machine.



+1 Tally

STATE	IN	OUT	NEXT
S_0	1	L	S_0
S_0	0	1	S_1
S_1	1	H	
S_1	0	H	

0, 1, 2, 0
0, 0, 1, 1
1, 1, 4, 0
1, 1, 4, 0

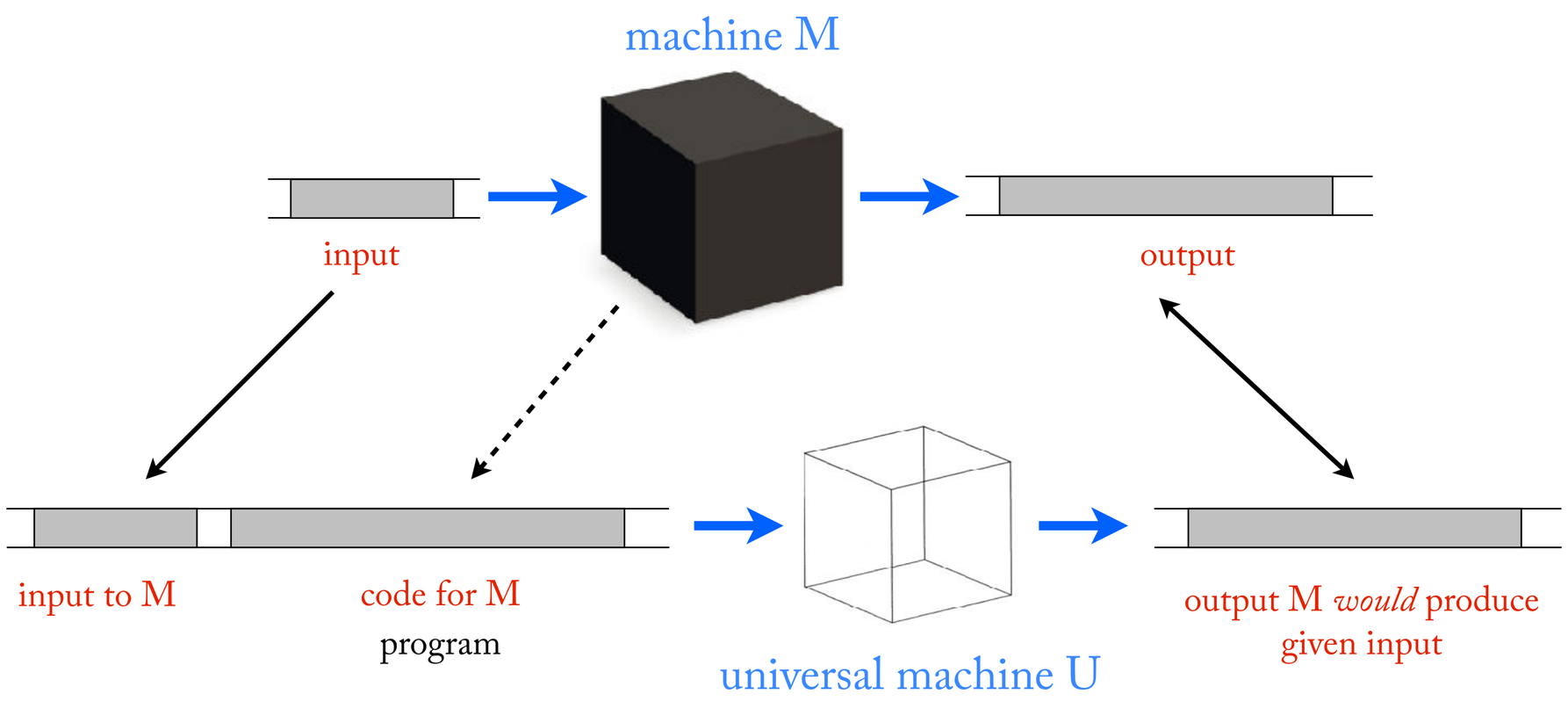
Turing machine → state table → numeral array

* 001011000 * 00001010 * 10101111000 * 10101111000 *

row 1 row 2 row 3 row 4

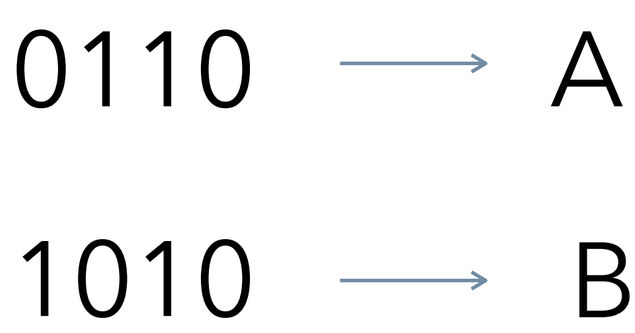
→ code number

Step 2: design a machine U which takes as input (i) any simulated machine M's code number, and (ii) M's input on a given occasion; U then produces the output which M would have produced in response to that input.



Markers

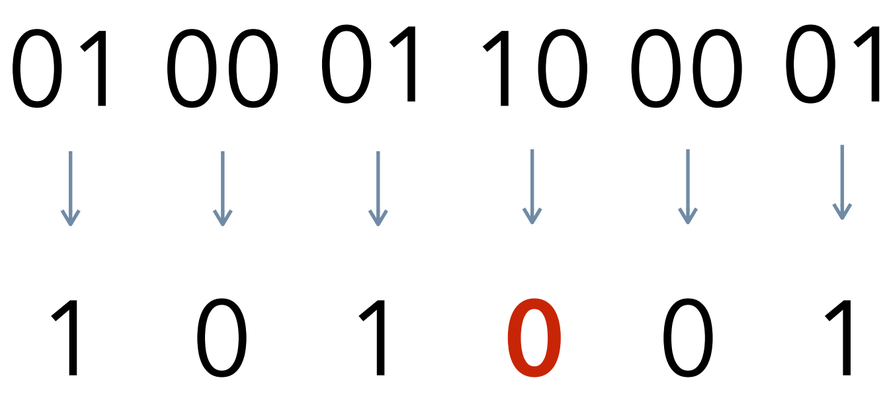
It's often useful to design an algorithm which makes use of more than two symbols. We can do this with 0's and 1's, by making *sequences* of 0's and 1's stand for additional symbols. This is more or less how your computer stores the 26+ letters of the English alphabet.



In particular, it is valuable to introduce a symbol which serves as a **marker** to indicate that a given symbol is in someway different from other symbols. Markers can be used to hold your place while computing, or two mark the end or beginning of a block.

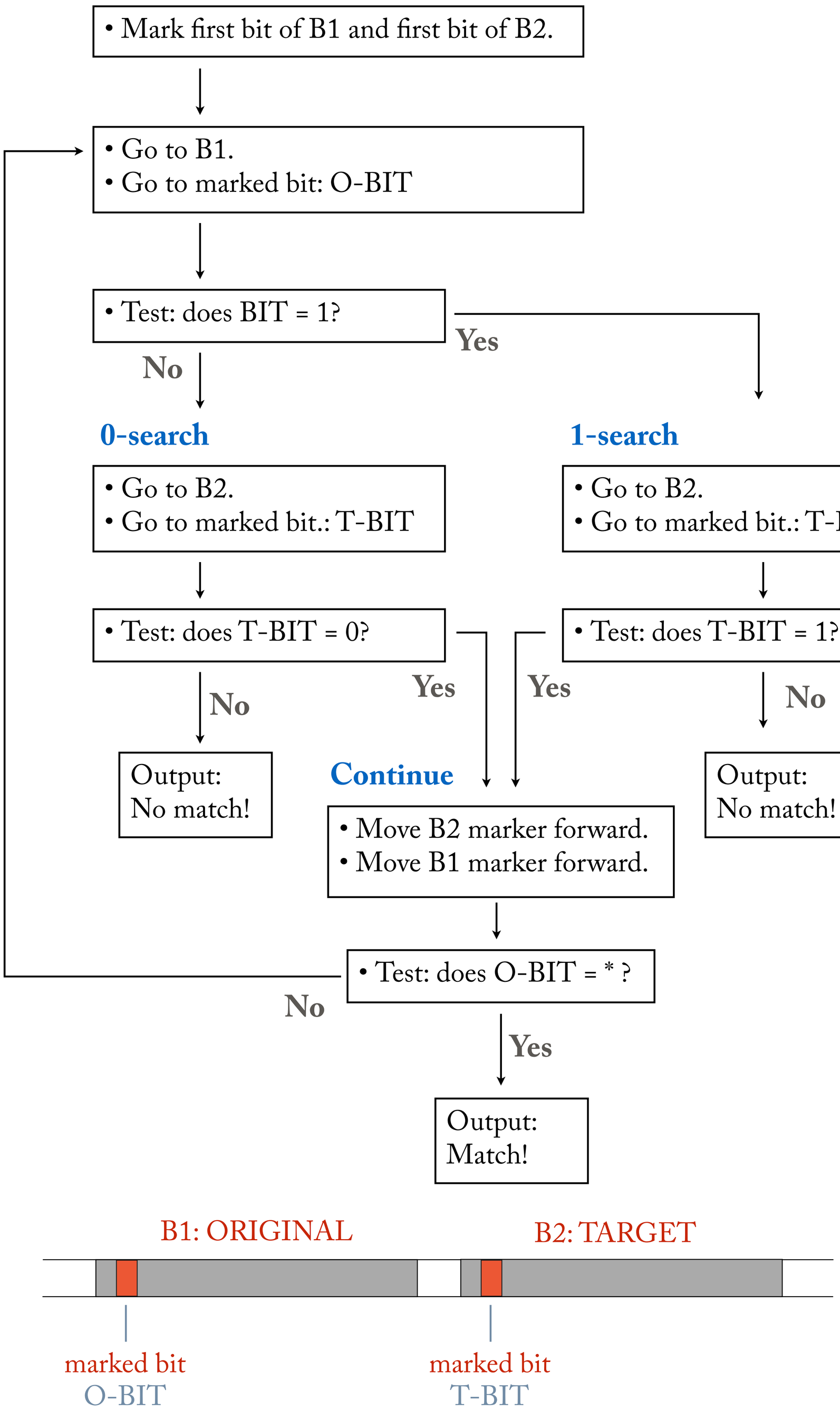
A simple and useful scheme is to allow each *two* bits to represent a symbol. Then the the right hand bit will represent the symbol itself, and the left hand bit will indicate whether it is "marked".

Now we can pick a marked symbol out of a crowd:



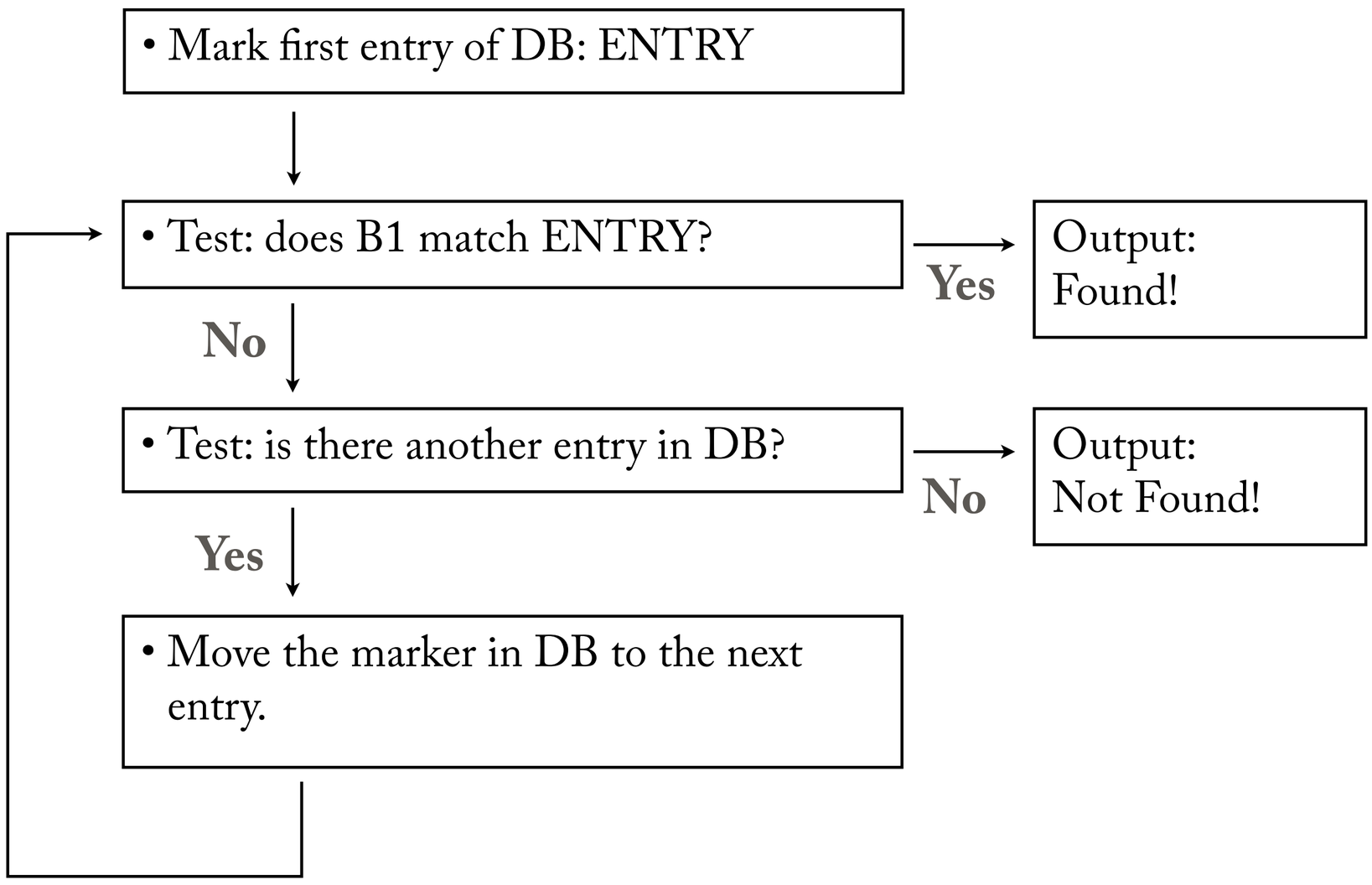
Matching

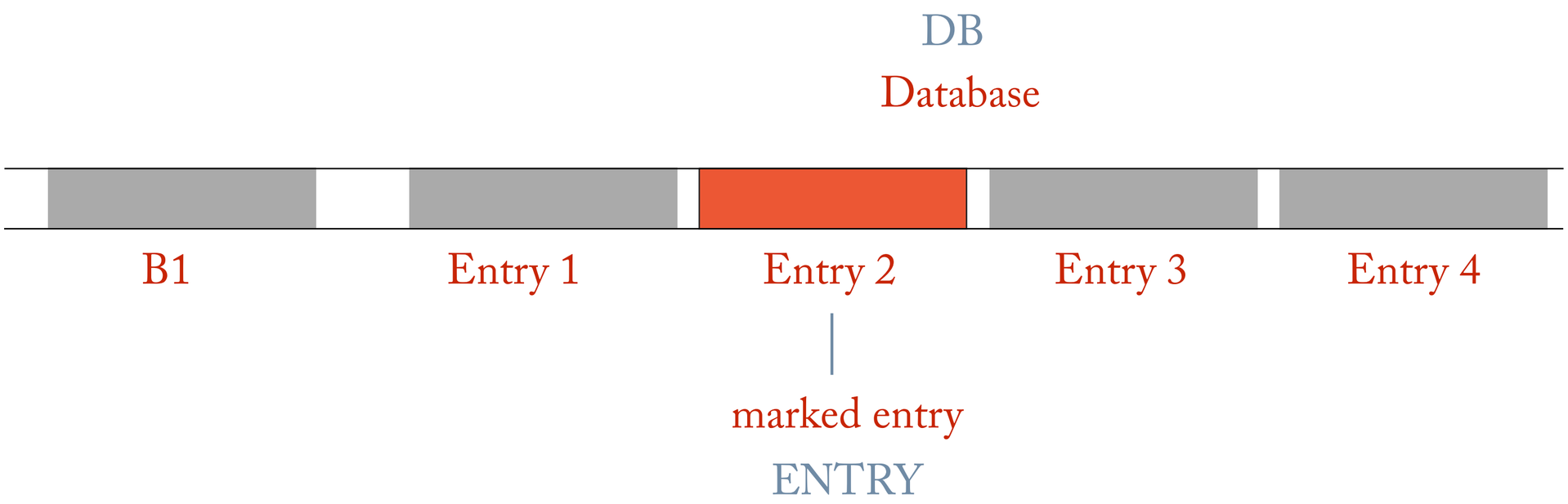
This algorithm compares an original string to a target string, and tests whether the two match or not.



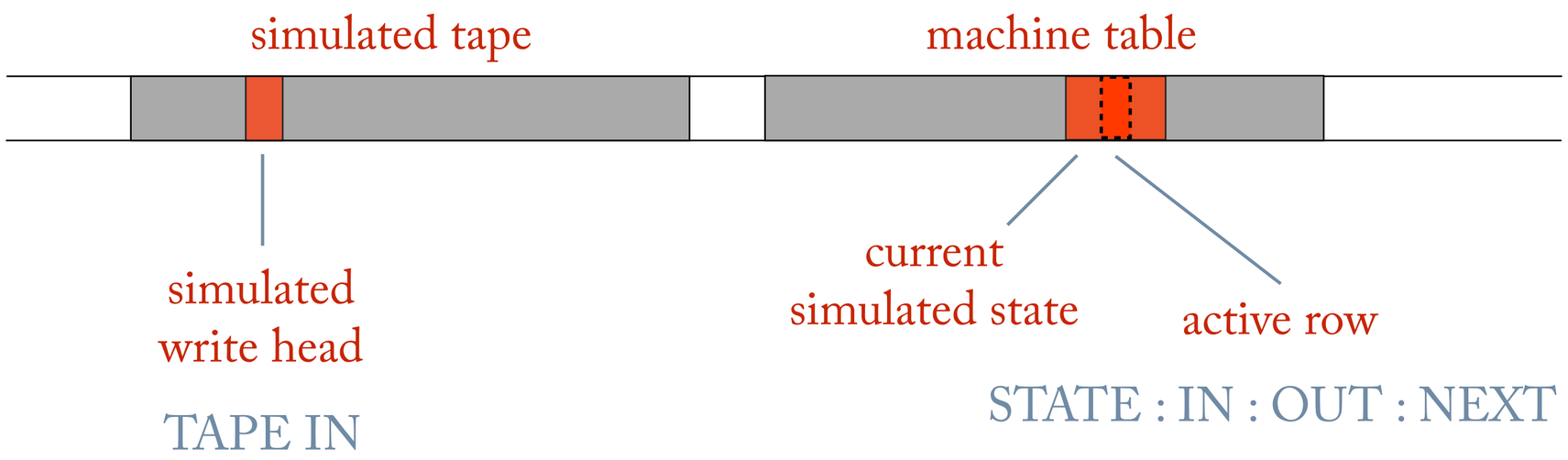
Search

This algorithm searches through a database to find an entry that matches some original sample. It outputs “Found!” or “Not Found!” depending on the result of the search.





Design for a Universal Turing Machine



Initialize + select first state
- Move the simulated write head to the first symbol of the simulated tape.
- Makes the first state of the machine table the current simulated state.

Initialize the simulated tape set the current state to S0.

Select current instruction
- Search the input column of the current simulated state for TAPE IN.
- Make that row the active row: STATE : IN : OUT : NEXT

Search for the current row based on the input from the simulated tape…

Check for final instruction
- Test: does OUT = Halt?

Yes →

Clean up
- Clean up and halt.

No

Perform current instruction
- Perform OUT on the simulated tape, at the position of the simulated write head.

…make the appropriate simulated output: writing or moving on the simulated tape.

Select next state
- Search the state column of the machine table for NEXT.
- Make that state the current simulated state.

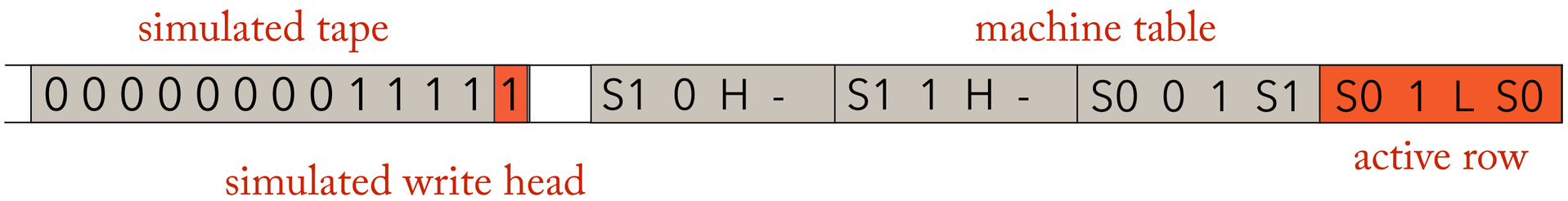
Find the next state named by the current row of the machine table, and make that the new current state.

Operation of a Universal Turing Machine

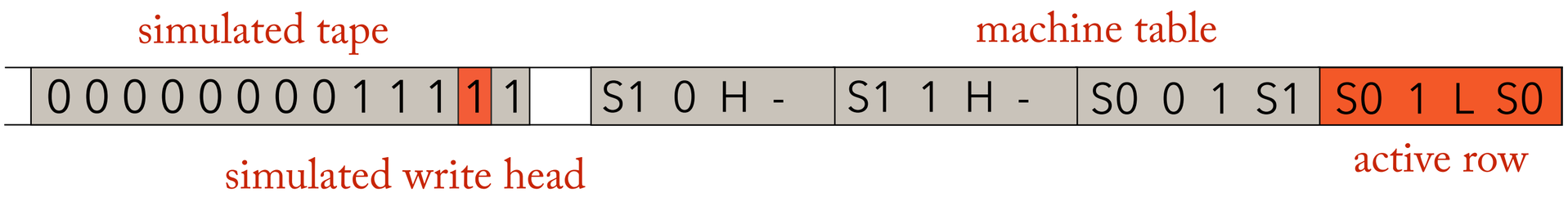
The following sequence illustrates the progress of the UTM, given the following machine table as its program.

STATE	IN	OUT	NEXT
S_0	1	L	S_0
S_0	0	1	S_1
S_1	1	H	
S_1	0	H	

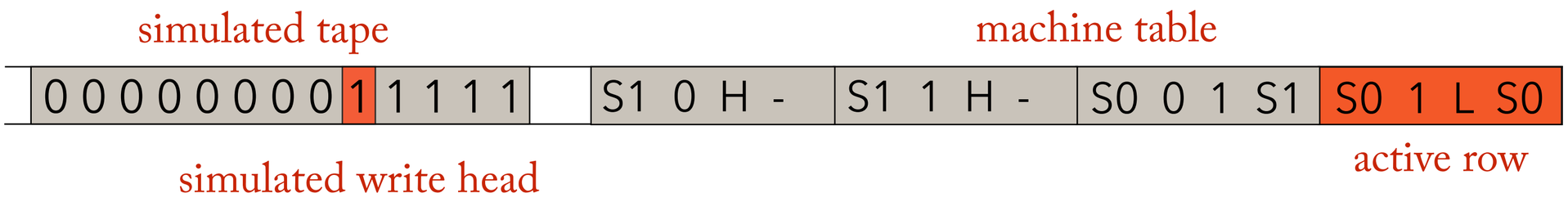
Initial state of the tape:



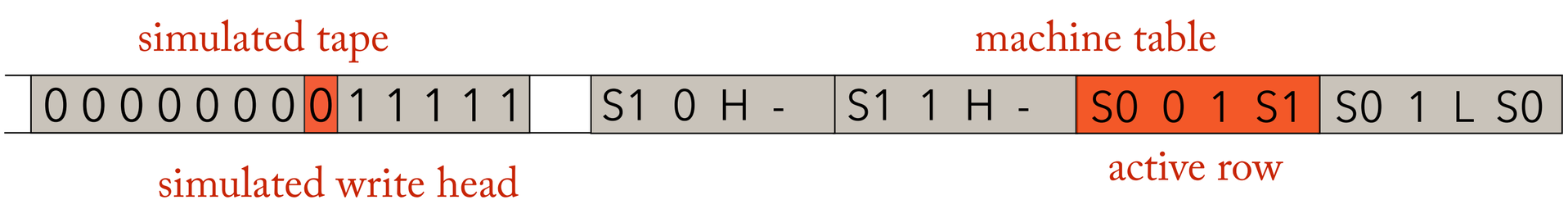
After 1 input/output cycle:



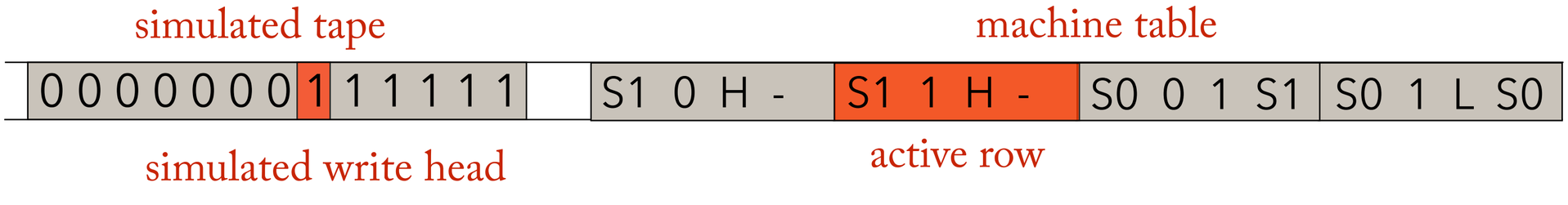
After 4 cycles:



After 5 cycles:



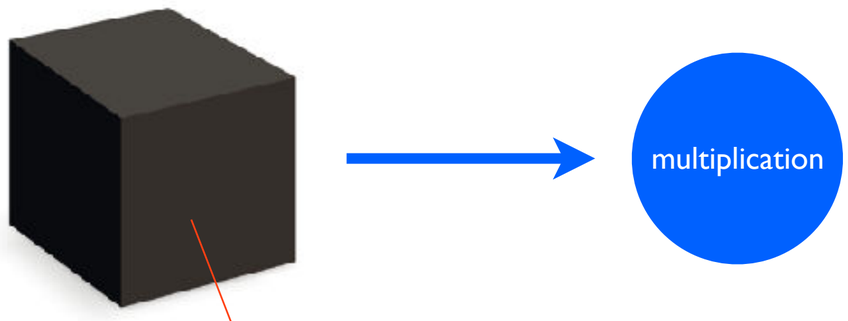
After 6 cycles:



39. Cognitive Architecture

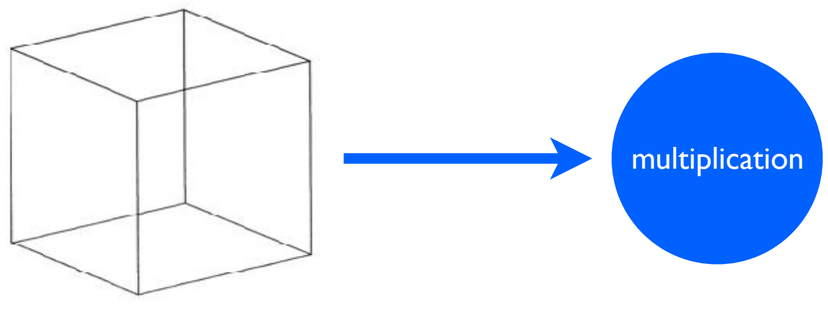
Implicit and Explicit Knowledge

Computes multiplication **without** any representation of a multiplication algorithm.

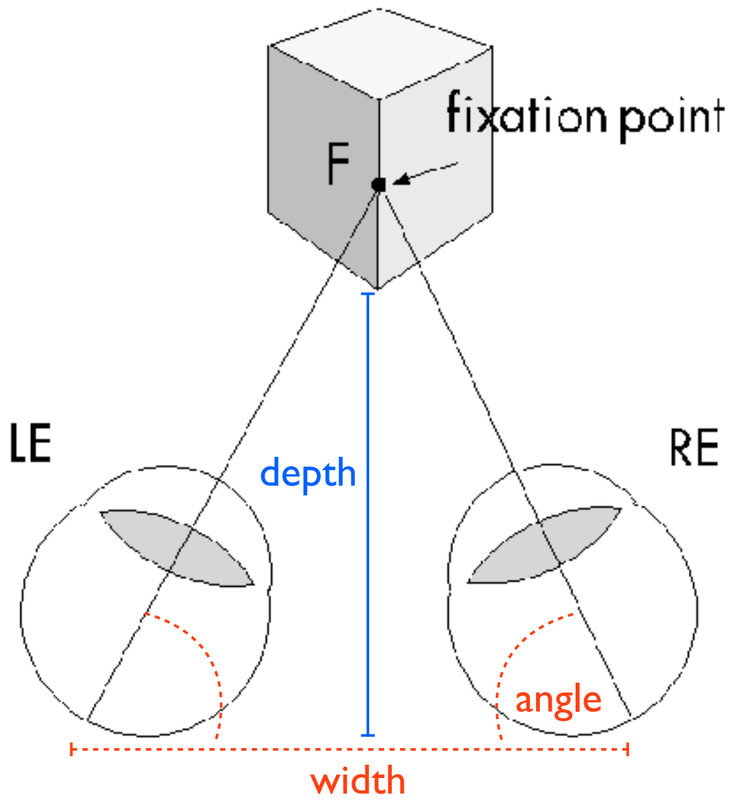


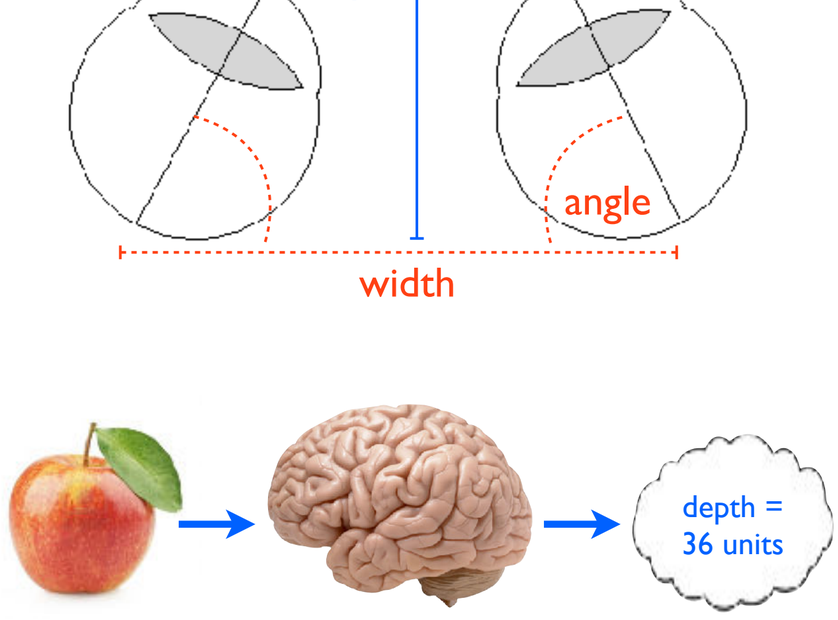
IMPLICIT KNOWLEDGE

Computes multiplication **with** a representation of a multiplication algorithm.

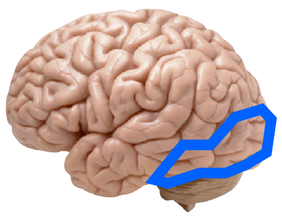


EXPLICIT KNOWLEDGE

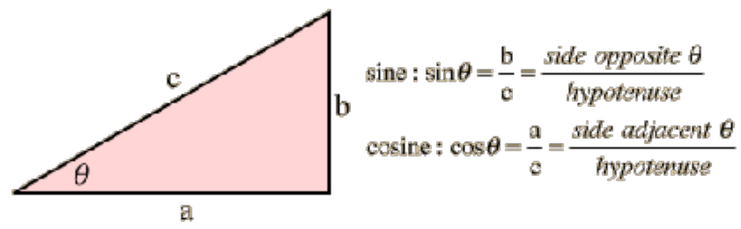




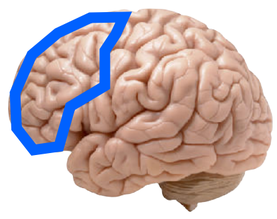
IMPLICIT KNOWLEDGE



Computes *tangent* **without** an explicit representation of a tangent algorithm.

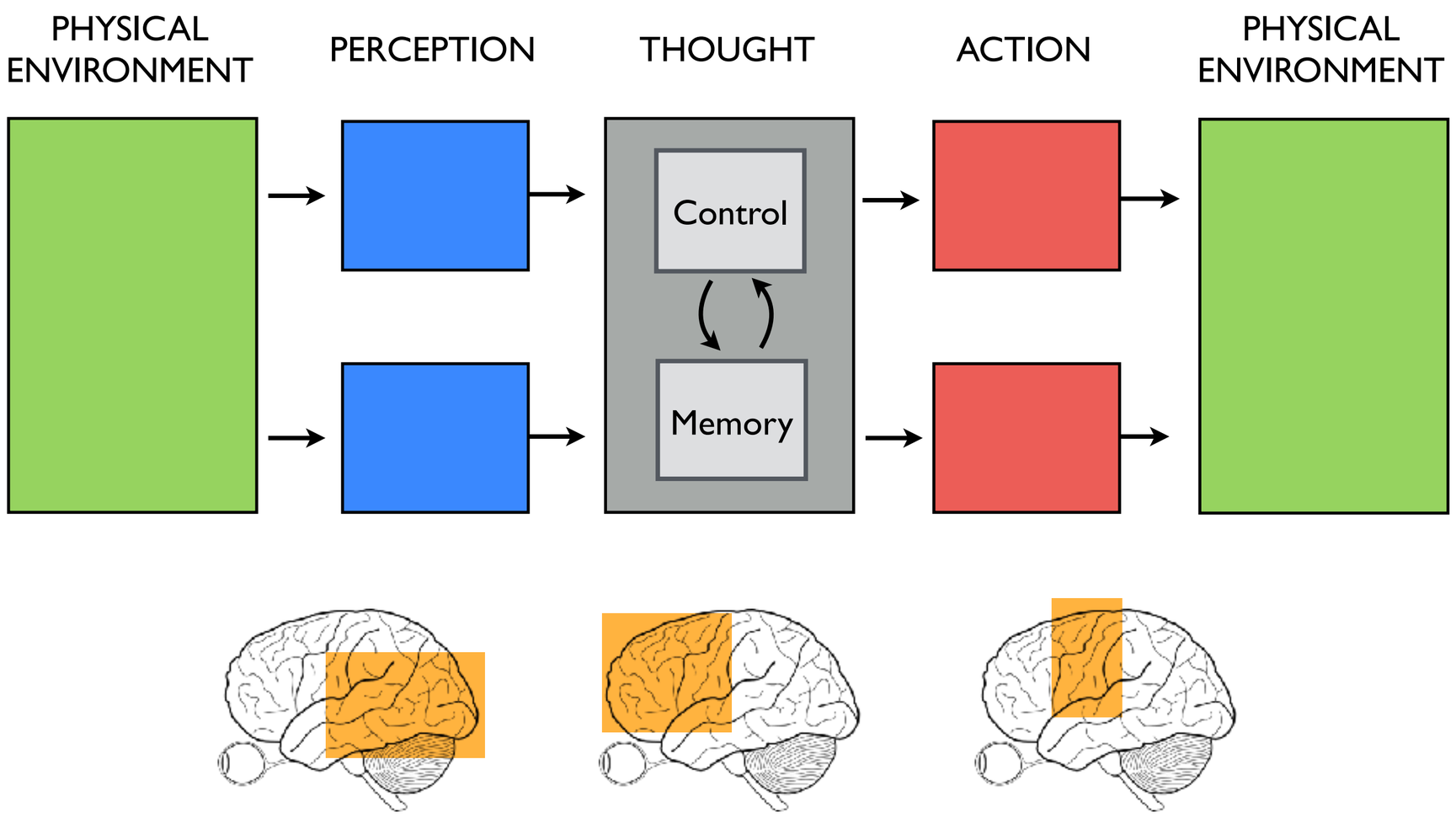


EXPLICIT KNOWLEDGE



Computes *tangent* **with** an explicit representation of a tangent algorithm.

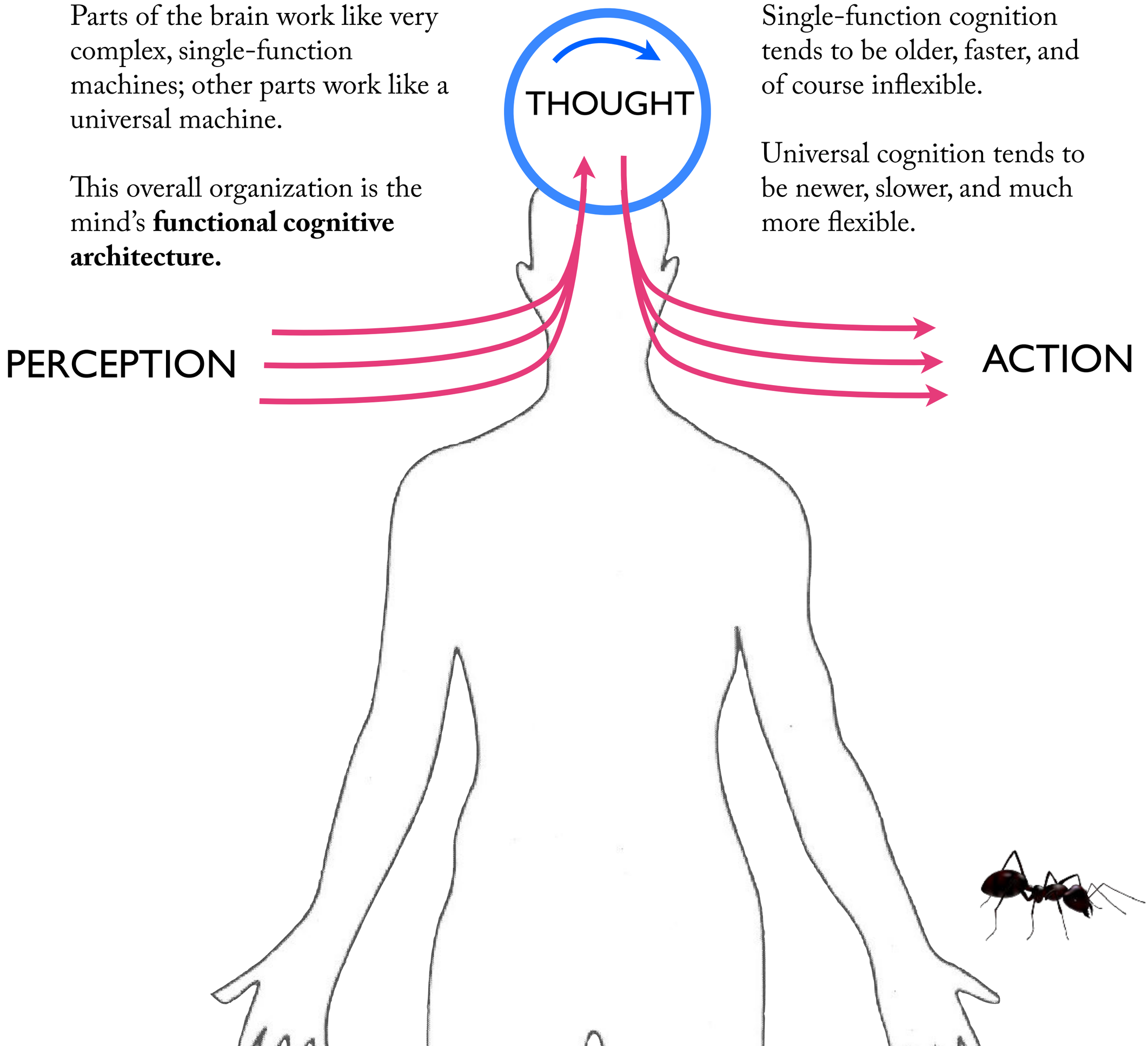
Cognitive Architecture



LOW-LEVEL COGNITION | **HIGH-LEVEL** COGNITION | LOW-LEVEL COGNITION

Parts of the brain work like very complex, single-function machines; other parts work like a universal machine.

This overall organization is the mind's **functional cognitive architecture.**

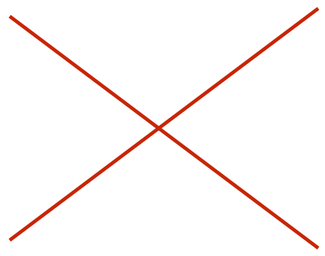


Single-function cognition tends to be older, faster, and of course inflexible.

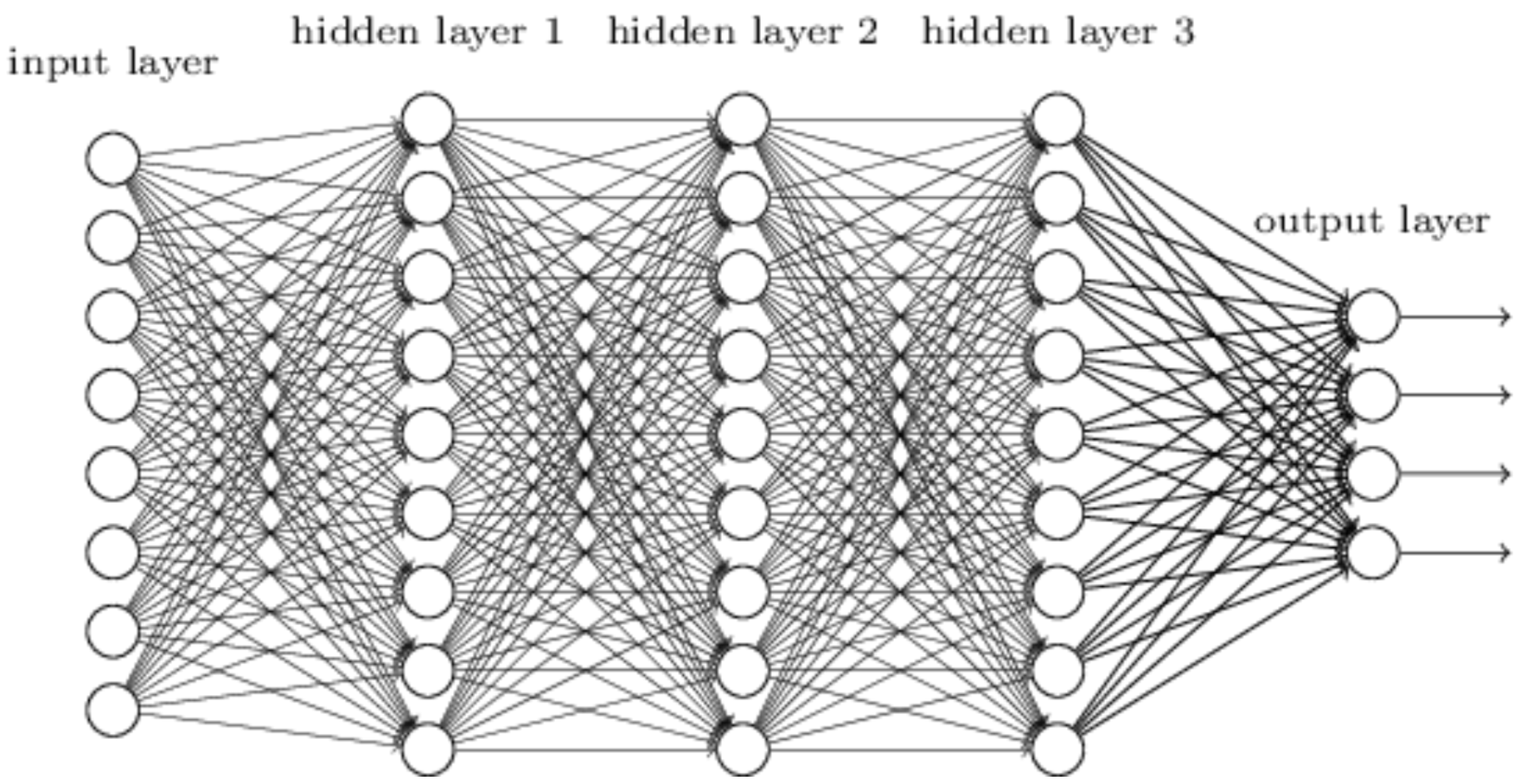
Universal cognition tends to be newer, slower, and much more flexible.

Implicit and Explicit Knowledge

There are two ways to draw the distinction between "implicit" and "explicit" knowledge.

	Implicit Knowledge Encoded in Architecture (not represented)	**Explicit Knowledge** Encoded in Memory (represented)
Implicit Knowledge Unconscious Knowledge	perceptual rules, motor control rules, knowledge how Turbot: Binary + I	implicit bias, blindsight, ? language rules Turbot: x-y coordinates UTM: Binary + I
Explicit Knowledge Consciously Accessible Knowledge		knowledge that, memories, beliefs, intentions, attitudes, reasoning, mental imagery

Some computer architectures, such as "neural nets" (not necessarily what's in our brains), don't draw a sharp distinction between logic engine and memory. Hence they don't draw a sharp distinction between knowledge encoded in *architecture* and knowledge encoded in *representations*.



40. Computation as Consciousness

The Architecture of Human Cognition

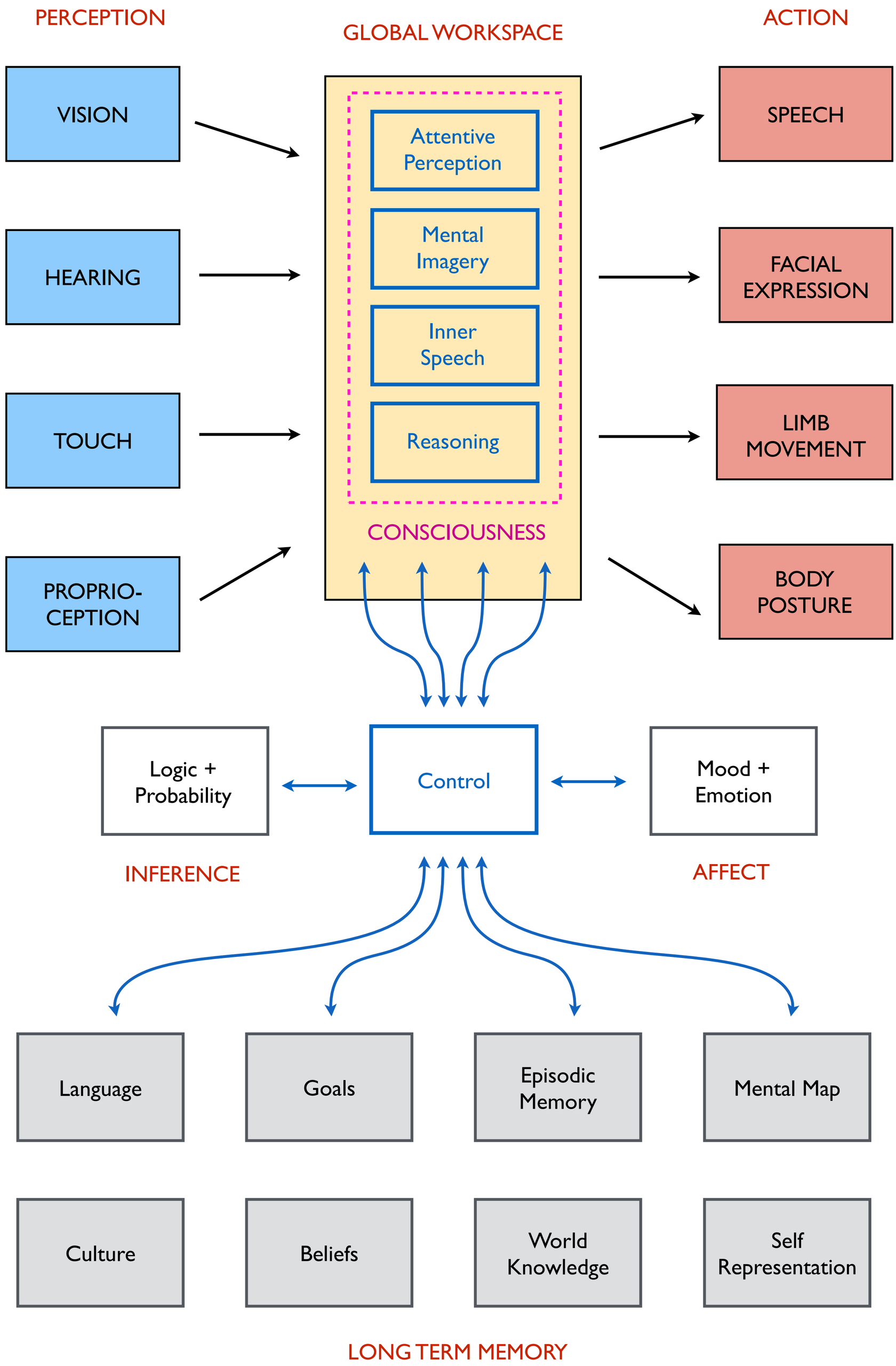
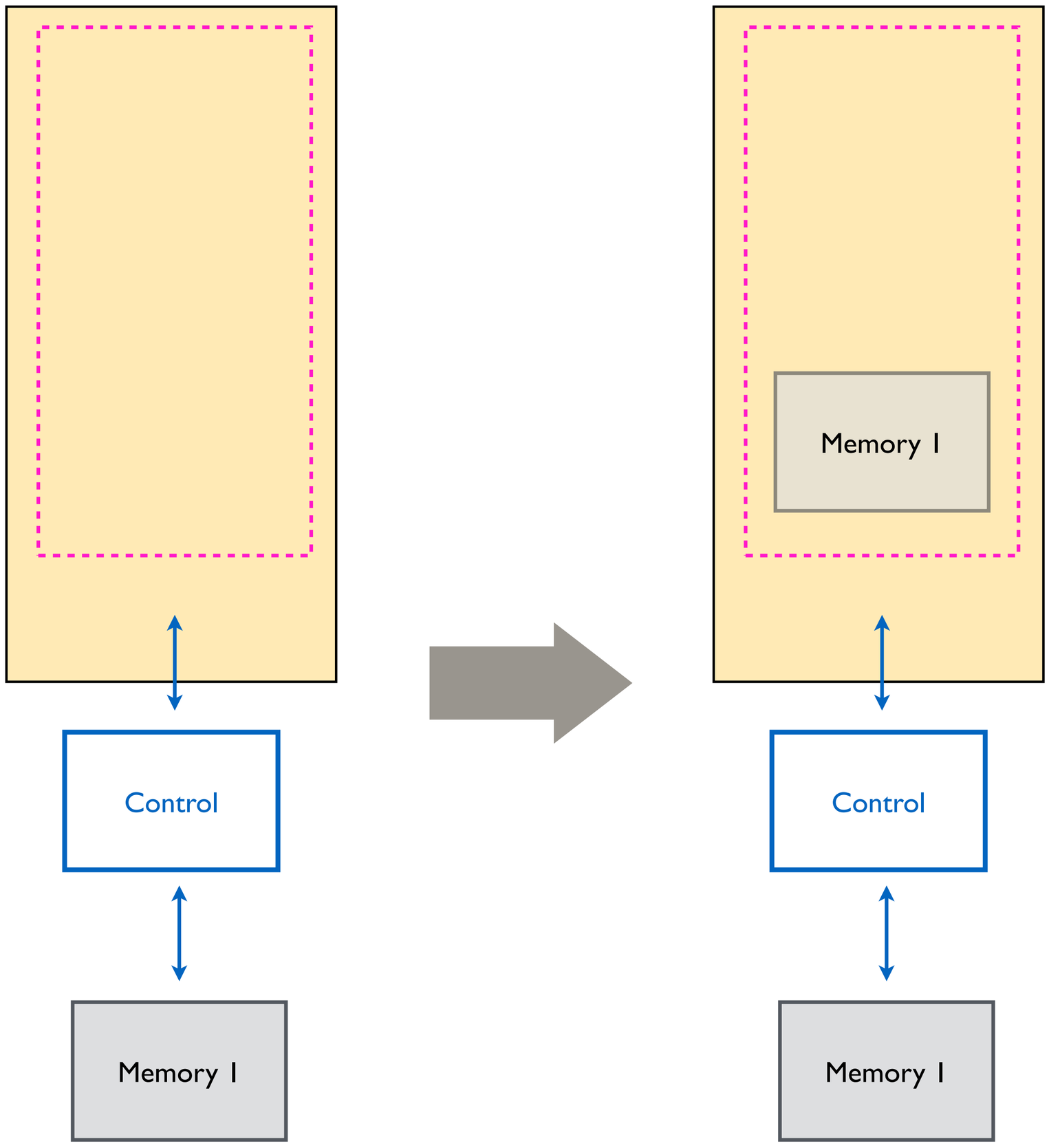


Illustration: recalling a memory.



Consciousness as center of information integration.

> "The ancient theater metaphor of consciousness gives us one plausible way to think about this evidence. A theater combines limited events taking place on stage with a vast audience, just as consciousness involves very limited information that with access to a vast number of unconscious sources of knowledge." (Baars)

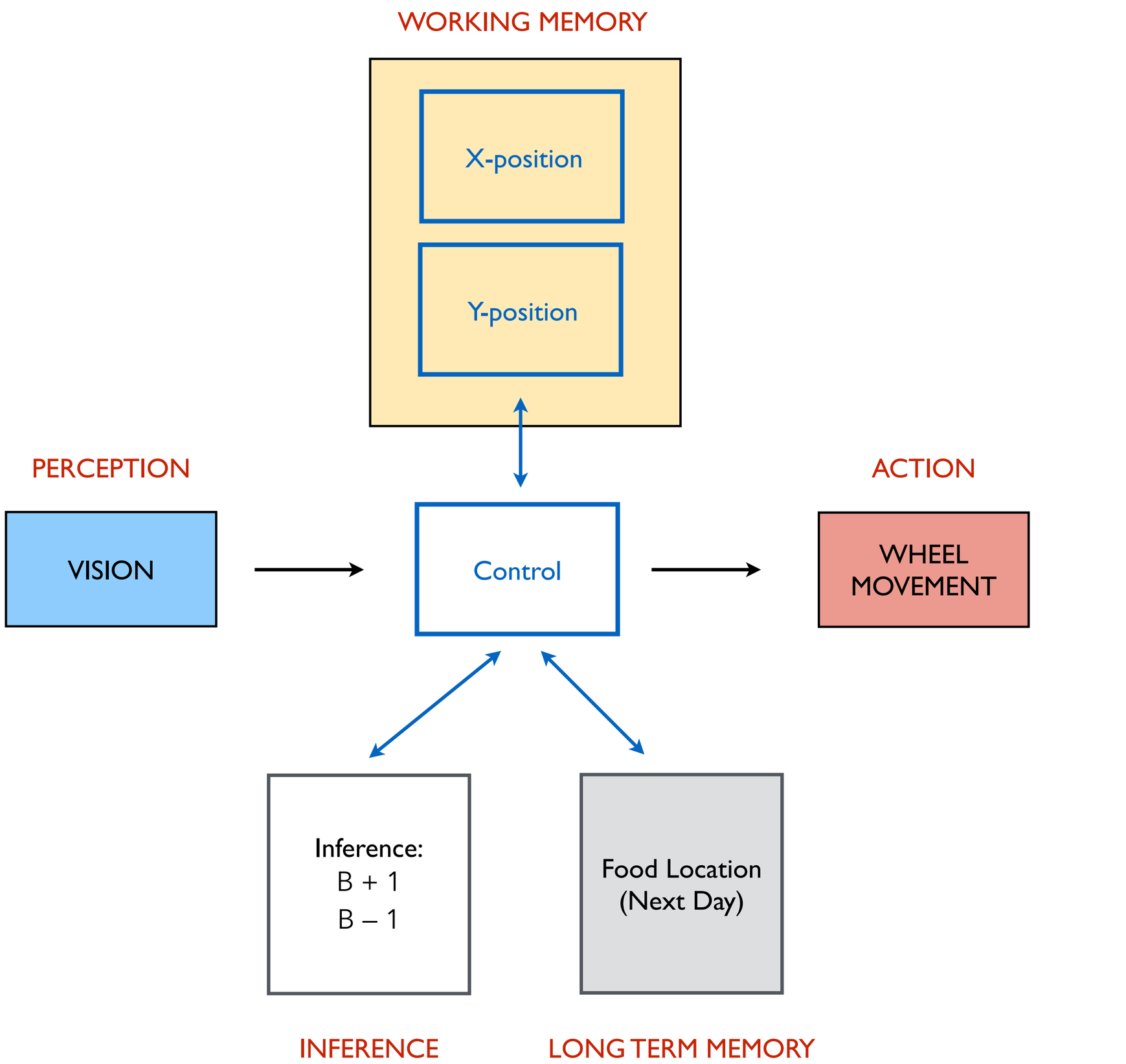
Conscious Cognition

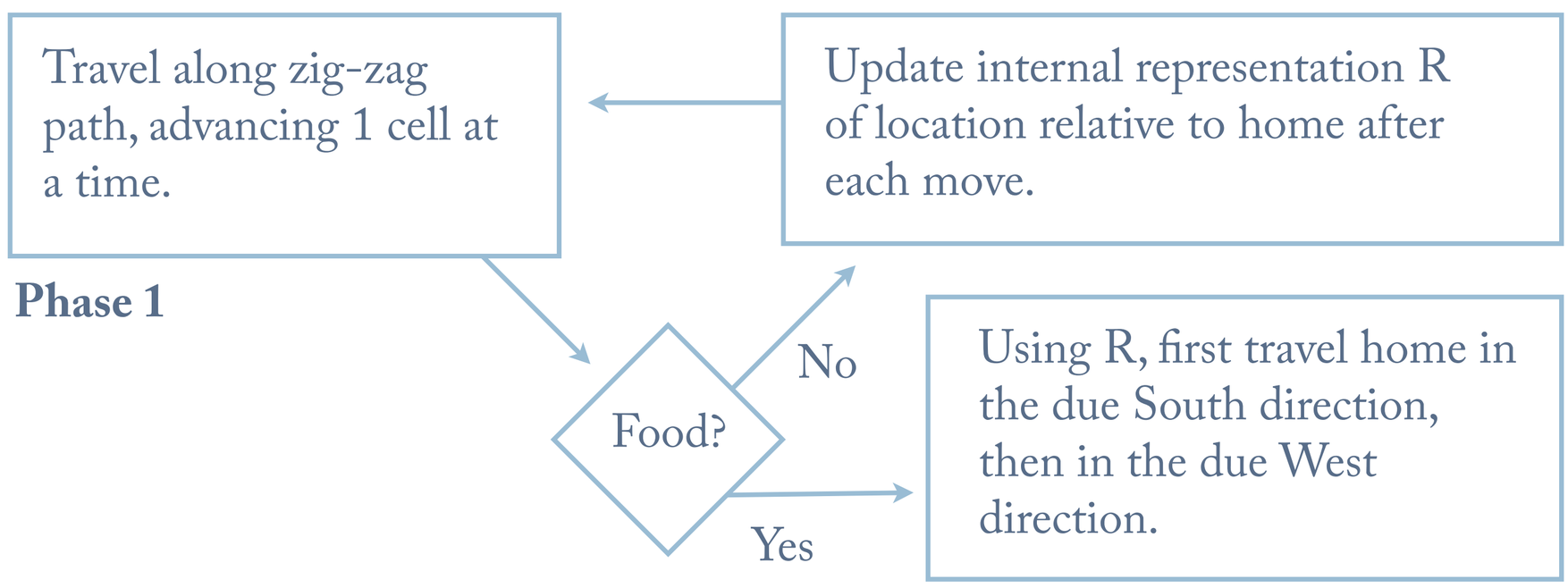
- Limited capacity
- Domain general
- Voluntary
- Informationally open
- Slow
- Adaptable

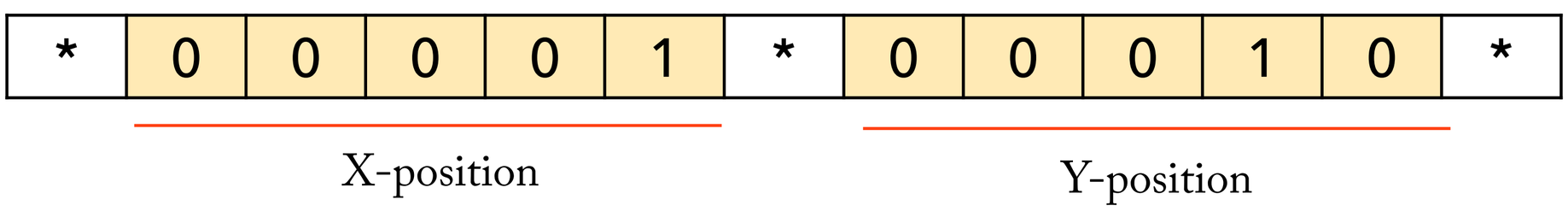
Unconscious Cognition

- Large capacity
- Domain specific
- Automatic
- Informally encapsulated
- Fast
- Fixed function

The Architecture of Turbot Cognition



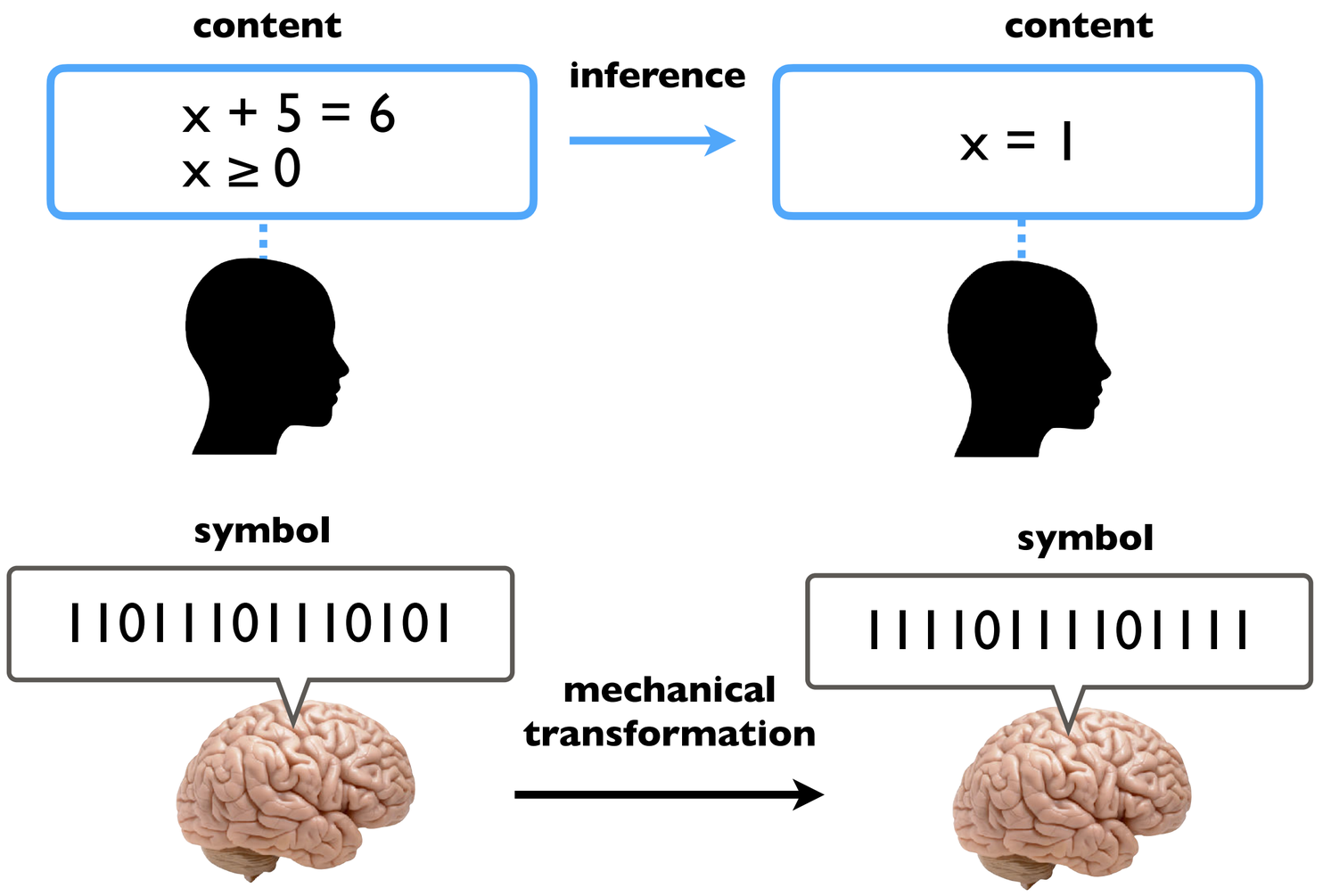


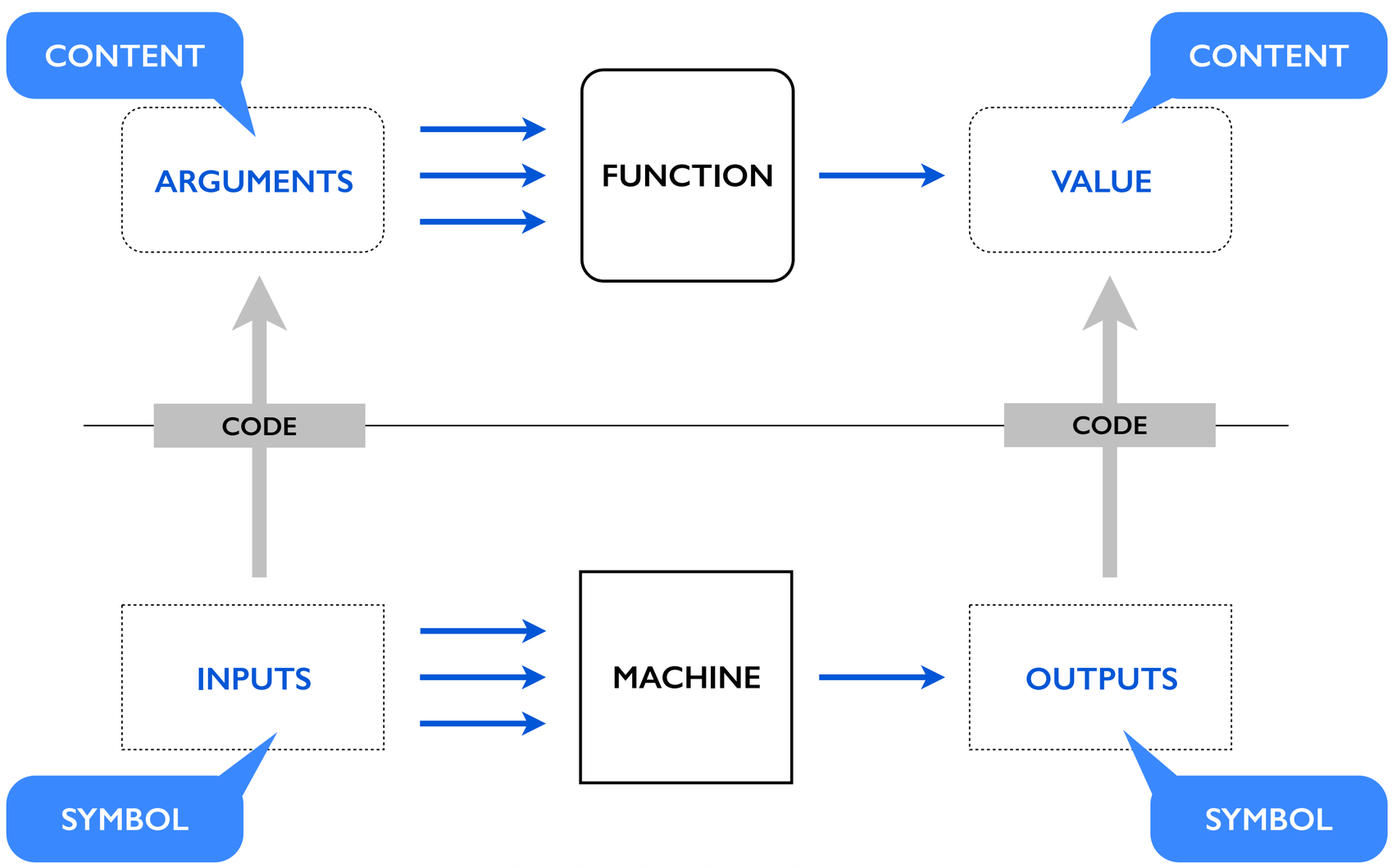


41. Computational Theory of Mind: The Long View

Cognition as Computation

My hypothesis then is that thought models, or parallels, reality—that its essential feature is not 'the mind', 'the self', 'sense-data', nor propositions but symbolism, and that this symbolism is largely of the same kind as that which is familiar to us in mechanical devices which aid thought and calculation.

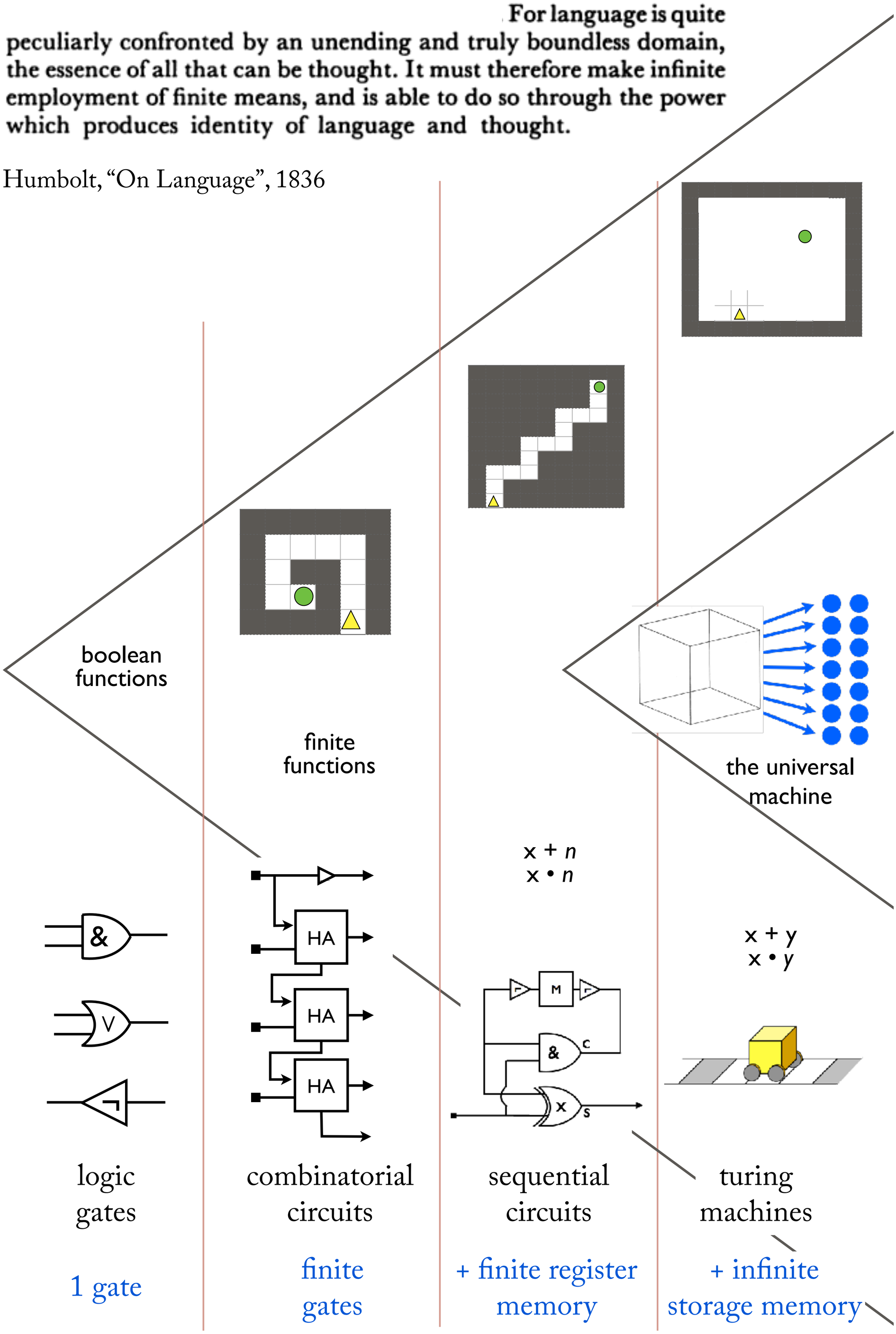




Orders of Computational Power

> For language is quite peculiarly confronted by an unending and truly boundless domain, the essence of all that can be thought. It must therefore make infinite employment of finite means, and is able to do so through the power which produces identity of language and thought.

Humbolt, "On Language", 1836



Levels of Computational Abstraction

"If one hopes to achieve a full understanding of a system as complicated as a nervous system… then one must be prepared to contemplate different kinds of explanation at different levels of description that are linked, at least in principle, into a cohesive whole, even if linking the levels in complete detail is impractical."

Marr, *Vision*, 1982

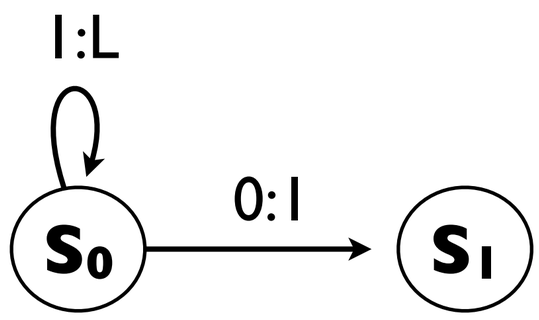
Ecological-Cognitive Level

+1

Functions

Semantic Abstraction

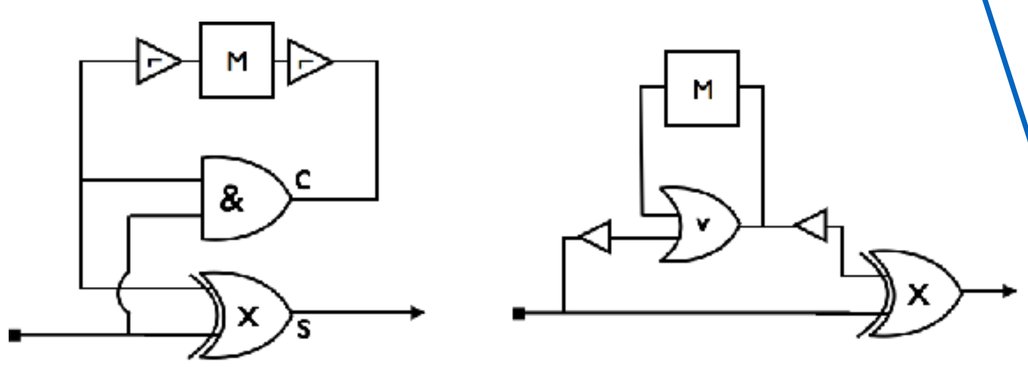
Algorithmic Level



State Machines

State Abstraction

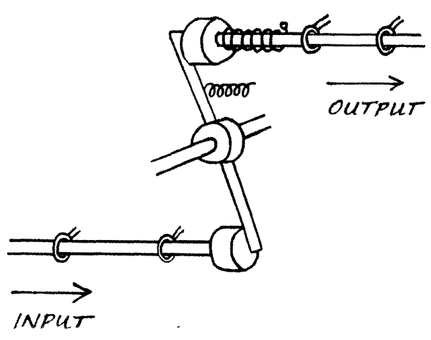
Mechanical Level

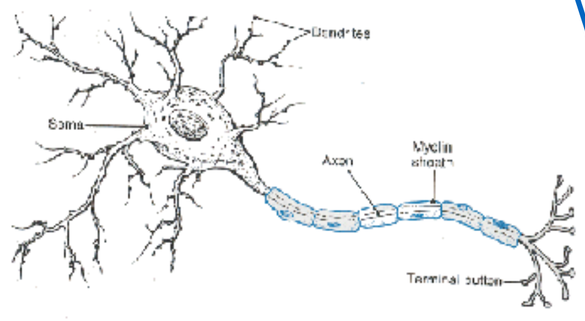


Logic Circuits

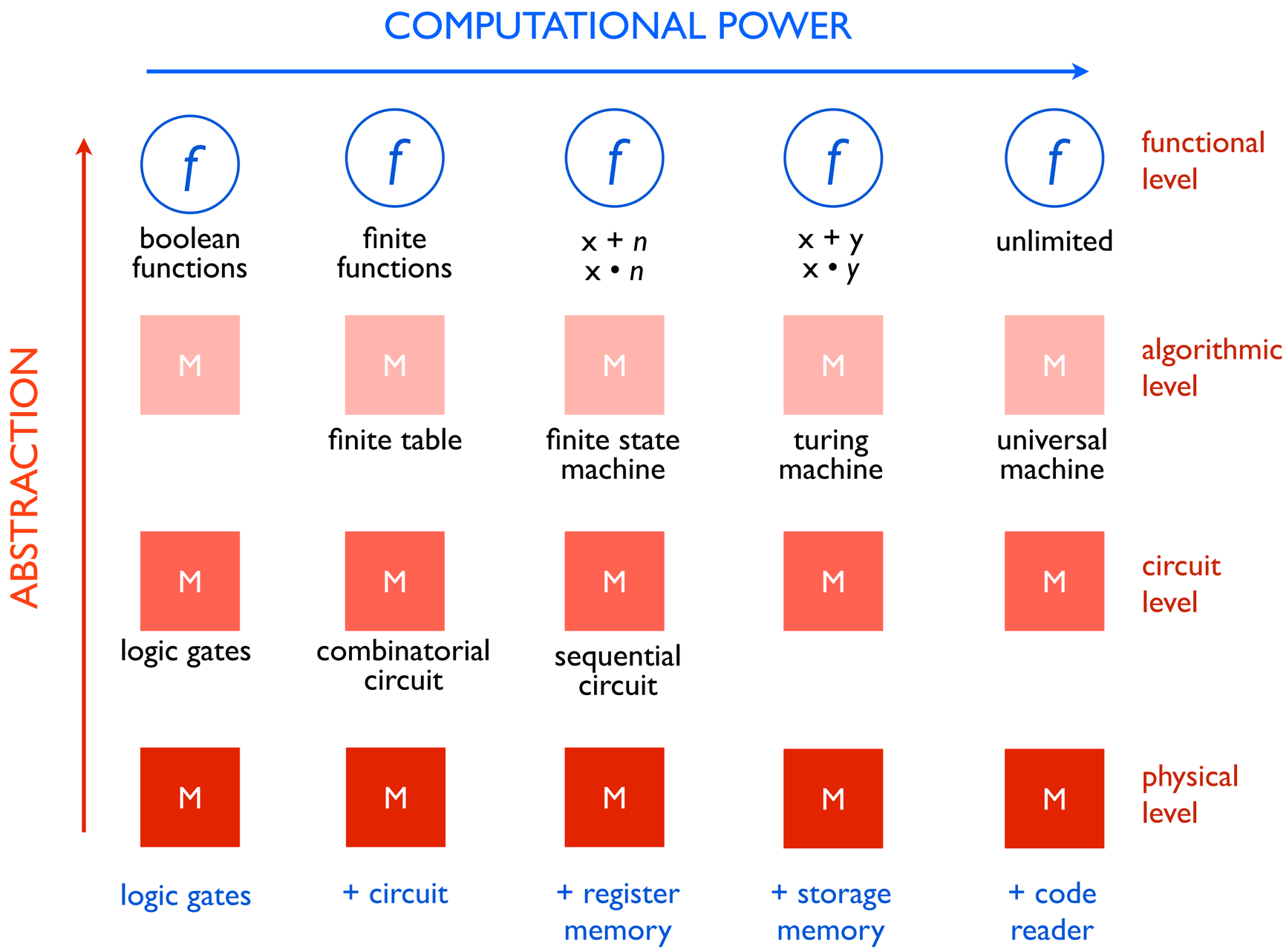
Functional Abstraction

Physical Level

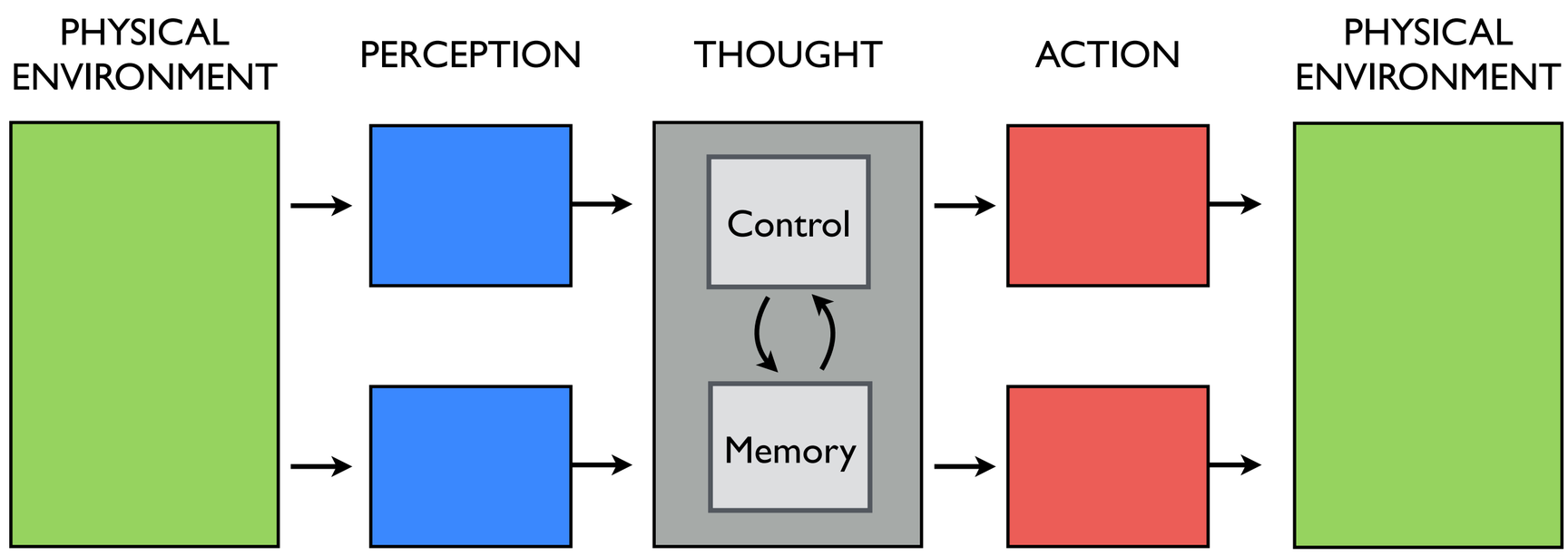


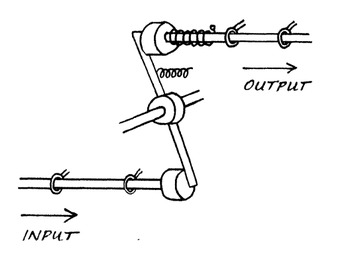


Physical Machines

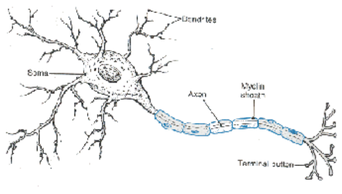


Cognition as Natural Computation





Home is 389 steps away.



I think therefore I am.

Editorial Assistance: Bill Kowalsky, Jon Ma, Christian DeLeon

Problem Sets

HW1. Basics: Circuits, Functions, and Representations

9.30.25

Note: I have put key words in **bold** in the text below. You may look these up (e.g. in the reading, or on Wikipedia) for help or context on a given problem.

I. Combinatorial Circuits

Design **combinatorial circuits** with the following input-output profiles:

1. **NAND**

IN 1	IN 2	OUT
0	0	1
0	1	1
1	0	1
1	1	0

2. Reconstructing OR

IN 1	IN 2	OUT
0	0	0
0	1	1
1	0	1
1	1	1

<< For this problem, do not use OR-gates.

DeMorgan's Law is useful here!

3. Splitting outputs

IN 1	IN 2	OUT1	OUT2
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	0

4. **XOR** (**exclusive OR**)

IN 1	IN 2	OUT
0	0	0
0	1	1
1	0	1
1	1	0

5. Combining circuits

IN 1	IN 2	OUT1	OUT2
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Hint: you might think about the circuits here as systematically analogous to equations in propositional logic, and the input-output tables as analogous to truth-tables. In many cases, you can use your knowledge of logic to work backwards to find the appropriate circuit.

II. Functions

6. Consider the function $m(\cdot)$ defined by the following **equation**:
   for any integers x, y, and z: $m(x, y, z) = 3 + (x \bullet y \bullet z)$
   a. Describe the function with an English sentence using each of the terms "argument", "value", "apply", and "return" correctly.
   b. Describe the function with another English sentence using the phrase "maps ... to".
   c. How many **places** does this function have?
7. Consider the function $k(\cdot)$ that takes two integers x and y as arguments, and returns the **successor** of their product as its value. Define this function with an equation.
8. This is a problem about **function composition**. Let $s(\cdot)$ be the **successor function**. Let $a(\cdot)$ be the **addition function**. Suppose that $p()$ is a 2-place function composed of successor functions and addition functions. Given that $p(1, 2) = 5$, provide two possible definitions of $p()$.
9. Consider the function $j(\,)$ that maps two integers x and y to their product, so long as both x and y are equal to or greater than 0, and equal to or less than two. For any other arguments, it is undefined.
   a. Define $j(\cdot)$ with an equation.
   b. Define $j(\cdot)$ with a table.

   This is a problem about **partial functions**.
10. The **Boolean numbers** are 0 and 1, where 0 and 1 are considered to be the "opposites" of one another. A **Boolean function** is a function which has only Boolean numbers as arguments and values.
    a. Using one equation, define a Boolean function which maps any Boolean number to its opposite.
    b. Provide two different definitions of the Boolean function which maps any Boolean number to itself. This is called the **identity function**.
    c. Provide two different definitions of the Boolean function which maps any Boolean number to the number one. This is called a **constant function**.

III. Representations

11. Represent the numbers zero through ten in **binary**.

12. Invent a new **base-6 counting system**, with all new symbols for the 6 digits (including zero!). List your basic symbols and their meanings. Then represent the number thirty-two in this system.

13. a. Add the binary number 1111 to the binary number 1.
    b. Add the binary number 1011 to the binary number 11001.
    Use one of the methods of addition from the Petzold reading; show your work!

14. Define a function *f* which maps the symbol "○" to the number zero and the symbol "●" to the number one. Use a table or an equation.

15. Suppose we use the symbols in Problem 14 to form a base-2 counting system. Let's call these symbols "dots". A dot can be black or white. Using an equation, define a 3-place function *g* which takes any three dots in a row and maps them onto the numerical value of that string of dots in that base-2 system. You will have to use the function in Problem 14 as part of the definition.

    So, for example, your function should predict that

    g(○, ○, ●) = one g(○, ●, ○) = two and so on…

    Your answer should take the form:
    "for any series of three dots x, y, z: $g(x, y, z)$ = …."

Hint: there are two things going on in Problem 15. It asks you to use your knowledge of functions to define the numerical values of symbols in a base-2 counting system. The method of computing binary values described by Petzold will be of help here. But it is also asking you to keep track of the difference between a *symbol* and its *content*, in this case the difference between "●" and *the number one*. Remember that only numbers, but not symbols, can be added and multiplied! At a deeper level, this problem asks you to give a partial **semantics** for a 2-place representational system. The science of formal semantics extends this kind of analysis to the domain of language.

Challenge problem: define a function which maps strings of 1's and 0's of *arbitrary length* to their numerical values. This is the general problem of semantics for binary.

IV. Synthesis

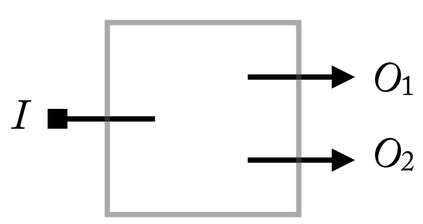
16. Let *tal*(·) be a function which maps strings of tally 1's and 0's to their numerical values. For example, *tal*(0)=zero, *tal*(1)=one, *tal*(00)=zero, *tal*(01)=one, *tal*(11)=two, and so on.

    Consider a machine *M* which, on any one occasion, takes as input a single atomic symbol *I*, and produces two atomic symbols as output, O_1 and O_2, such that, in any context, *successor*(*tal*(*I*)) = *tal*(O_1O_2). Design a combinatorial circuit for machine *M*, with the schematic form below.

17. Let *bin*(·) be a function which maps strings of binary 1's and 0's to their numerical values. For example, *bin*(0)=zero, *bin*(1)=one, *bin*(00)=zero, *bin*(01)=one, *bin*(10)=two, *bin*(11)=three, and so on.

    Consider a machine *N* which, on any one occasion, takes as input a single atomic symbol *I*, and produces two atomic symbols as output, O_1 and O_2, such that, in any context, *successor*(*bin*(*I*)) = *bin*(O_1O_2). Design a combinatorial circuit for machine *N*.

Hint: these two problems look complex, but they are just asking you to apply the intuitive concept of **computation** we have been discussing in the course so far. First figure out what table of input and output symbols would satisfy the equations given. Only then solve for the underlying circuit.



Schematic for *M* and *N*

HW2. Computing with CCs

10.10.25

I. CC Arithmetic

Starting with **the three basic logic gates only**, design CCs that compute the following functions. At any stage you may define a new boxed circuit from a previously established circuit or logic gate. Useful circuits to define in this way include **All-1**, **XOR**, **half-adder**, and (if you wish) **full-adder**. You will also want to introduce boxed circuits for specific mathematical functions as the need arises.

1. Half adder
2. +1 [0-15] B
3. +2 [0-15] B
4. +3 [0-15] B
5. $2x$ [0-15] B
6. $2x$+1 [0-15] B
7. 2(x+1) [0-15] B (only use *one* sub-part which computes +1)
8. x+y [0-15] B (i.e. addition, bounded to 0-15 for each argument)
   Reference**:** Petzold (2000) "A binary adding machine."
9. For any integer n, in principle, could you design a CC that computes +1 [0- n] B? Why or why not? Short paragraph.
10. Could you design a CC that computes +1 B (i.e. unbounded)? Why or why not? Short paragraph.

Challenge problem: prove the following: for any numerical function f and any integer n, there is a CC which computes f[0 - n] B.

On binary addition

Hint 1: Use the grade school algorithm for long addition.

Start by thinking of the problem in terms of long addition, e.g.:

	1	1	Input 1
+	1	0	Input 2
1	0	1	Output

Then set up your machine in the same format:

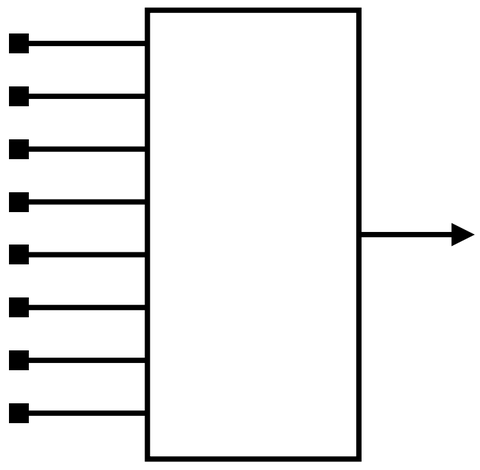
	Digit 4	Digit 3	Digit 2	Digit 1	
	☐	☐	☐	☐	Input 1
	Digit 4	Digit 3	Digit 2	Digit 1	
	☐	☐	☐	☐	Input 2
Digit 5	Digit 4	Digit 3	Digit 2	Digit 1	
☐	☐	☐	☐	☐	Output

When filling in the circuit, think of your machine first in terms of columns (e.g. how does the Output Digit 1 relate to the Input Digit 1's?). Then think about how the columns relate to each other. A key idea here is the "carry digit"– remember how this works in standard long addition, and see if you can apply the same idea to the construction of your machine.

Hint 2: Add twice. When you are doing long addition, note that you really have to do basic addition *twice* for each column. First you have to add the digit from the top row to the digit from the bottom row, and then you have to add the carry to the result. (You can also do these in the opposite order.) The same thing is true in binary, a fact you should convince yourself of before trying to solve the problem. You'll find this idea has a direct correlate in the circuit which computes binary addition.

II. CC Perception

For each of the following problems assume a CC that has 8 inputs and 1 output, like the schematic circuit below. Think of these as computing **perception**: they take as their input an array of stimulations (like hitting the retina) and they output a symbol which represents a specific shape or category.



10. **Edge detector.** Design a machine that obeys the following rule: output 1 iff there is a string of *at least* three consecutive 1's anywhere in the input.

11. **Object detector.** Design a machine that obeys the following rule: output 1 iff there is a string of *exactly* three consecutive 1's anywhere in the input.

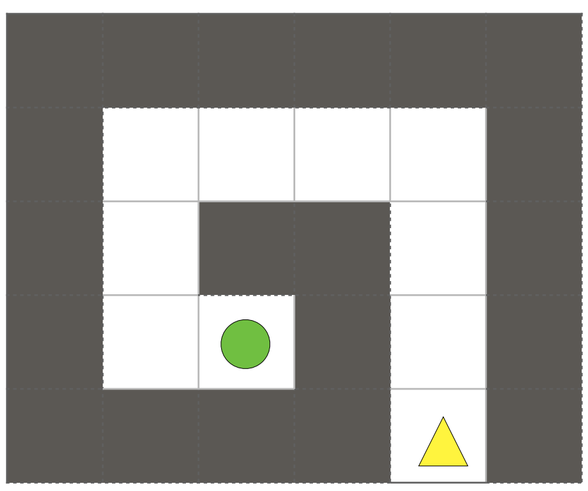
12. **Landmark recognition.** Design a machine that obeys the following rule: output 1 iff the input = 11010111

III. CC Navigation

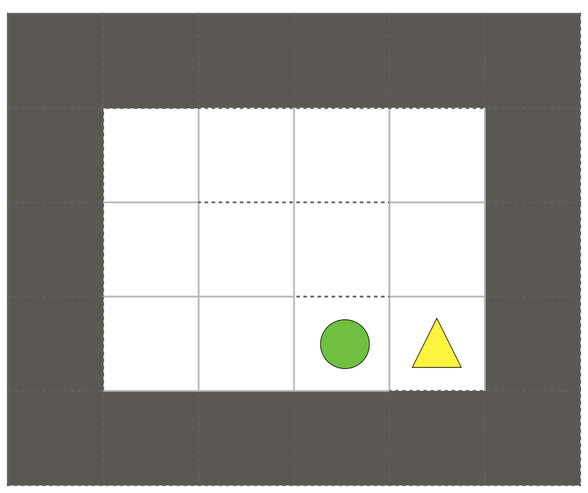
In these navigation tasks, the yellow triangle indicates the turbot's starting position and orientation. Navigation is "completed," and the problem solved, when the turbot *crosses over* the goal. It does not need to stop on the goal. (Think of Pac-Man.)

For each of the following problems, answer the question: can it be completed by a combinatorial circuit embedded in a turbot? If so, give the circuit. If not, explain why in a short paragraph.

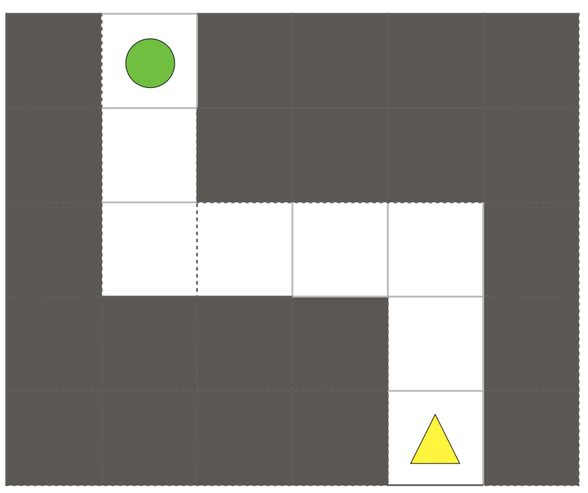
13. **Spiral**



14. **Full circle**



15. **Zig-zag**



IV. Reflection

16. **Representational systems.** Suppose you were designing a calculator for solving a high-stakes problem. (Perhaps "you" are Nature, the "calculator" is the Brain, and the "problem" is Survival.) Would you design it to calculate in binary or in tally? Why? Use specific examples to justify your answer. ~1 paragraph

17. **Embodied computation.** Consider a Turbot like the one in problem 13. What function, if any, does it compute? If it does compute a function, what are the arguments and what is the value of this function? How does this compare to the function computed by the "directions" mode in Google Maps? ~1 paragraph

HW3. Computing with SCs

10.27.25

I. SC Arithmetic

At any stage you may define a new boxed circuit from a previously established circuit. You may assume boxed circuits for **XOR** and **half-adder.** You will also want to introduce boxed circuits for specific mathematical functions as the need arises.

Design SCs that compute the following functions.

1. +1 B
2. +2 B
3. $2x$ B
4. $2x$+1 B [Use one $2x$ circuit and one +1 circuit]
5. $2(x+1)$ B [Use one $2x$ circuit and one +1 circuit]
6. $x+y$ B

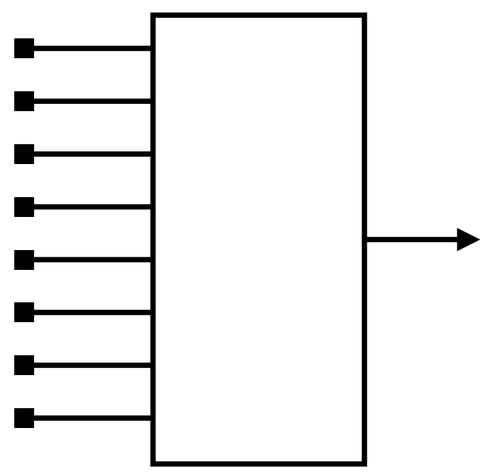
For each of the following functions, is it possible to compute this function with an SC? If so, construct the circuit. If not, explain why in a short paragraph: be as clear and precise as you can.

7. +3 T
8. $2x$ T
9. $x + y$ T
10. $x \bullet y$ B (multiplication)

II. SC Perception

For each of the following problems assume a SC that has 8 inputs and 1 output, like the schematic circuit below. Think of these as computing **perception**: they take as their input an array of stimulations (like hitting the retina) and they output a symbol which represents a specific shape or category.

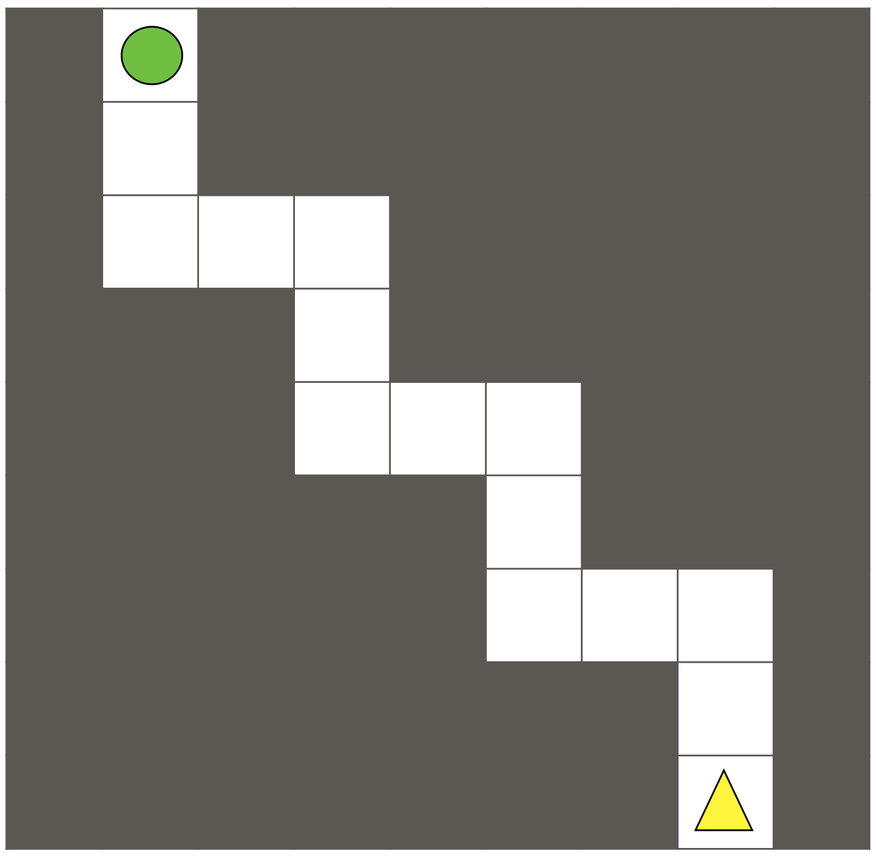
11. **Change detector.** Design a machine that obeys the following rule: output 1 iff the current input differs in any way from the previous input.
12. **Motion detector.** Let an *object image* be a string of exactly three consecutive 1's. Design a machine that obeys the following rule: output 1 iff there is an object image anywhere in the input moving upwards 1 unit per unit of time.



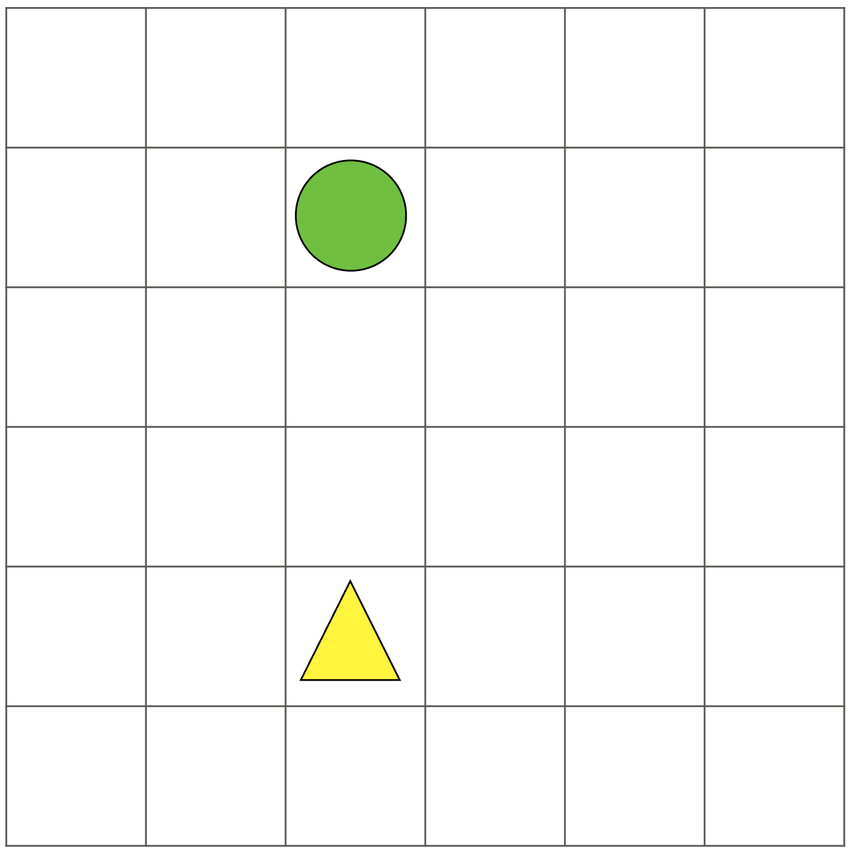
II. SC Navigation

For each of the following navigation tasks: can it be completed by a combinatorial circuit embedded in a turbot? If so, give the circuit. If not, explain why in a short paragraph.

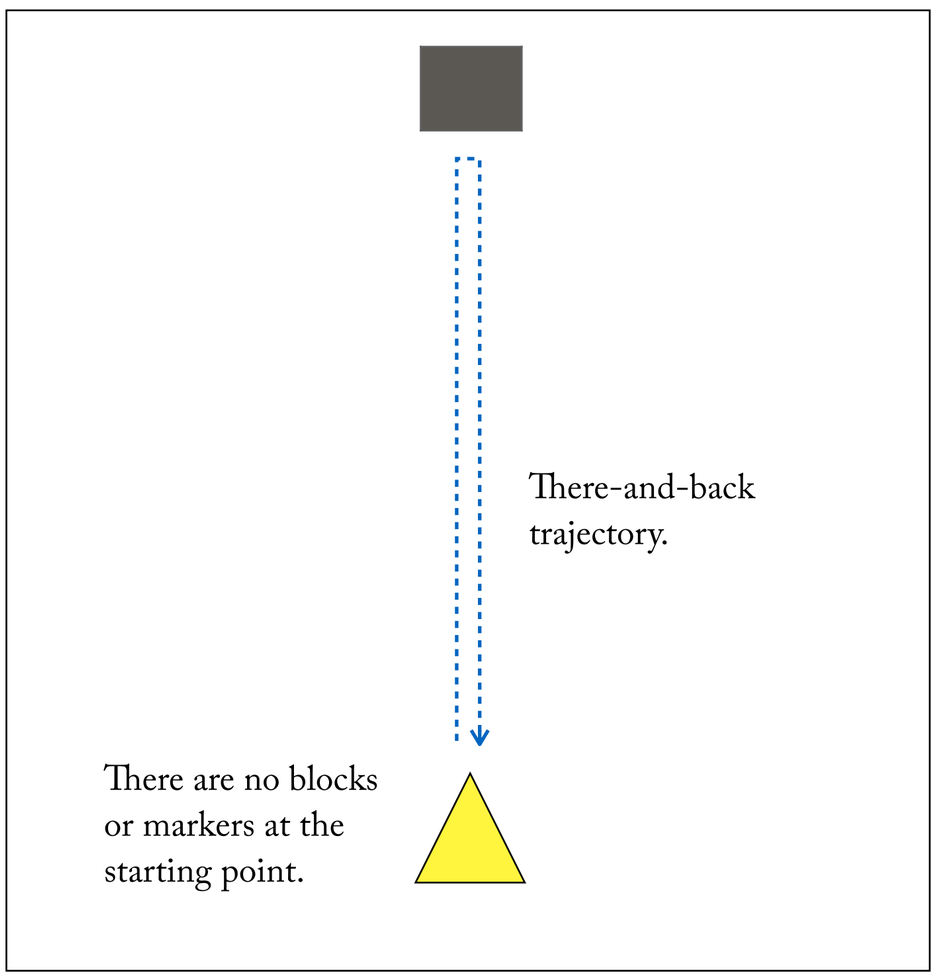
13. **Zig-Zag:** reach the goal and **keep going**.



14. **Three ahead:** reach the goal and **stop**.



15. **Mad Max:** go to a block directly ahead, but an unknown distance away, then return to the starting point and **stop**.



Note 1: SC Turbots have two outputs, one for each wheel.

Note 2: The goal (the green circle) is invisible to the turbot; it reads as an empty block.

III. Reflection

Short paragraph answers.

16. SCs are more computational powerful than CCs: SCs can solve problems that CCs cannot. What physical abilities would you add to an SC if you wanted to make it even more powerful?

17. Could there be an intelligent being with no memory?

HW3. Computing with SCs

2.3.21

I. SC Arithmetic

At any stage you may define a new boxed circuit from a previously established circuit. You may assume boxed circuits for **XOR** and **half-adder.** You will also want to introduce boxed circuits for specific mathematical functions as the need arises.

Design SCs that compute the following functions.

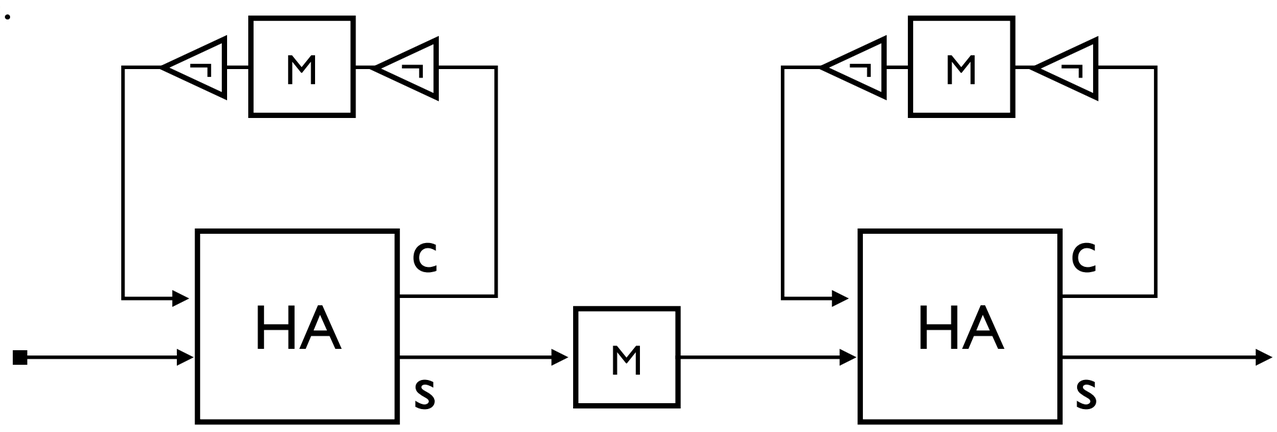
1. +1 T
2. +1 B
3. +2 B
4. $2x$ B
5. $4x$ B
6. $2x$+1 B [Use one $2x$ circuit and one +1 circuit]
7. $2(x+1)$ B [Use one $2x$ circuit and one +1 circuit]
8. $x+y$ B
9. **ON/OFF** switch:
   In the ON state, this circuit outputs 1's; in the OFF state it outputs 0's. It switches between states each time it receives a 1, but keeps the same output if it receives a 0. Start in the OFF state.

For each of the following functions, is it possible to compute this function with an SC? If so, construct the circuit. If not, explain why in a short paragraph: be as clear and precise as you can.

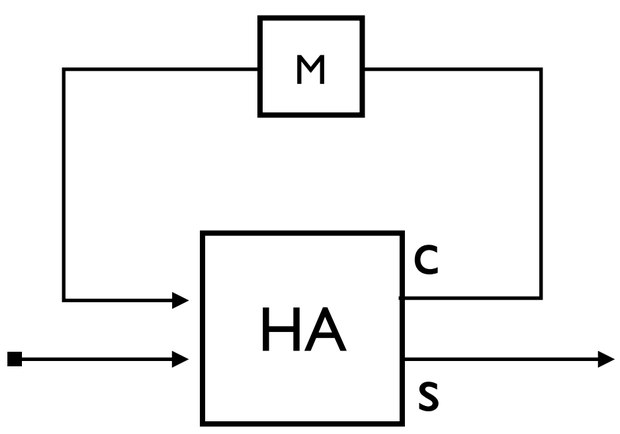
10. +3 T
11. $2x$ T
12. $x \cdot y$ B (multiplication)

What functions do the following circuits compute in Binary?

13.



14.



Add: pattern recognition problems. ab*, aabb*,

add arbitrary pattern recognition (words)- e.g. last three are 1101.

motion detection

make 2x+1, 2(x+1) circuit decoding problems.

explain +1, +2 in class—- have students generalize for HW

Give 2x+1 as on1 neg + M+neg. Ask what it computes in T? in B? how would you decide?

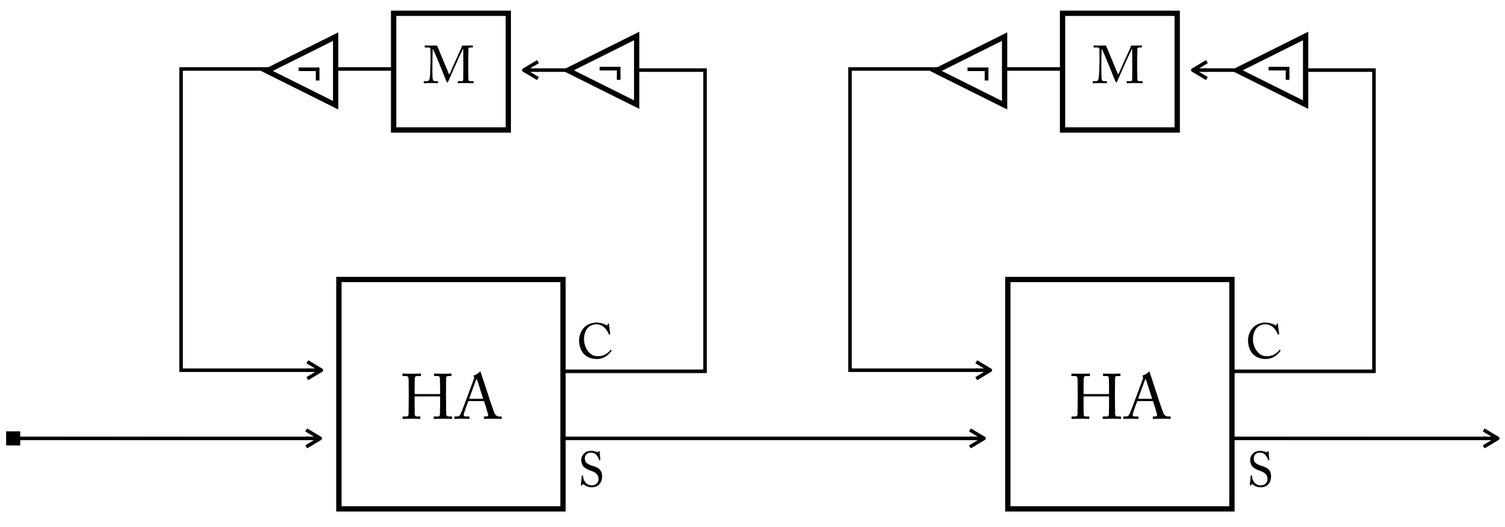
Add discovery problems: here is the input and output over several sequences. What does it do?

HW4. Computing with FSMs

11.11.25

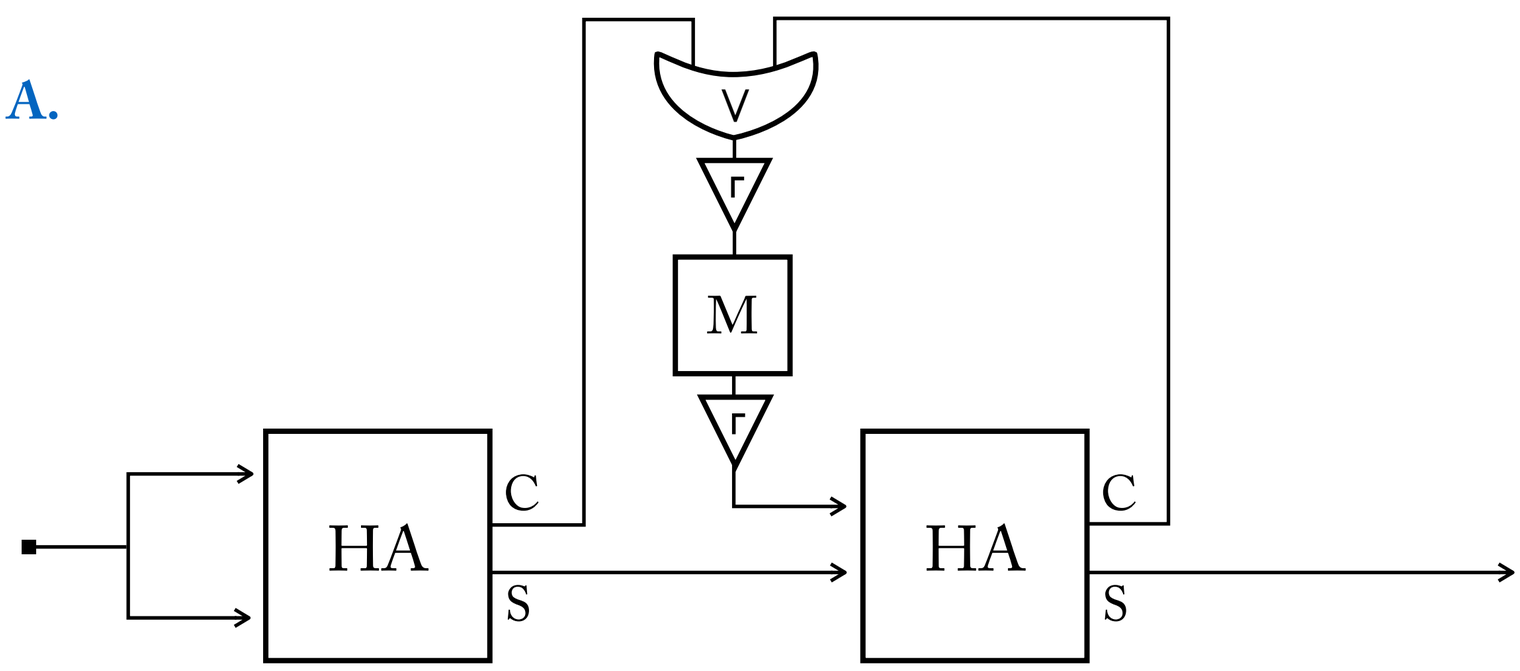
I. State Abstraction

1. **State abstraction.** Construct a state table for the following machine, then convert it to a FSM diagram. What function does it compute in binary?

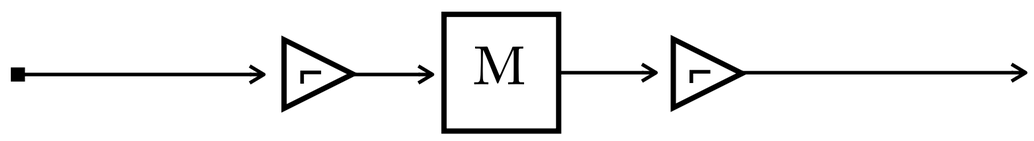


2. **Multiple realizability.** Consider the following two SCs. What function does each compute in binary? Do they implement the same FSM? Why or why not? Show your work.

A.



B.



II. FSM Arithmetic and Beyond

Design FSMs that compute the following functions. (Don't design an SC first to get to your solution.)

3. +1 T

4. +2 T

5. +1 B

6. +2 B

7. +3 B

8. $2x$ B

9. $2x$+1 B

10. $x + y$ B

> **Challenge problem:** prove the following: for any numerical function f and any integer n, there is an FSM which computes f[0 - n] B.

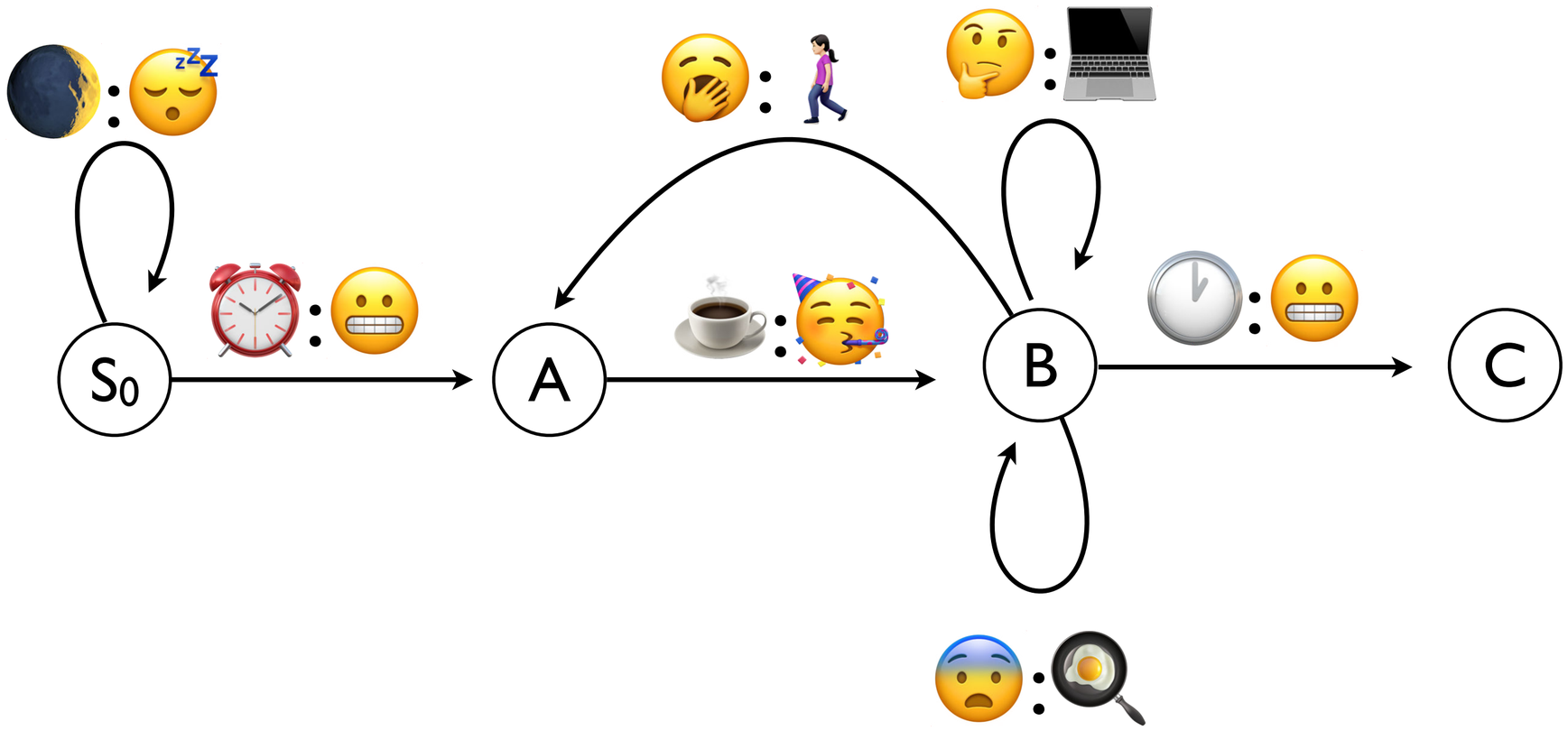
11. **ON/OFF Switch**

    This circuit toggles between outputting 1's or 0's; it switches each time it receives a 1, but keeps the same output if it receives a 0. Start in the off state, where output = 0.

12. **The Daily Grind**

    Design an FSM which captures the cycles and possible paths of your typical morning routine, using only emojis as inputs and outputs.

For example, here is a fragment of my day:



III. FSM Navigation

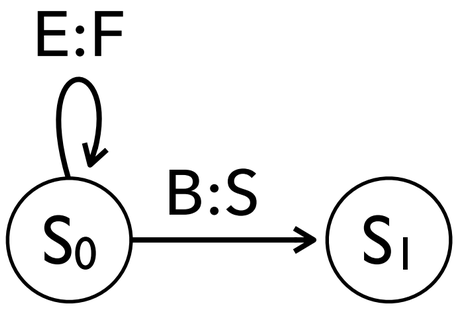
For each of the following navigations tasks, design an FSM which solves it.

Use the following abbreviations:

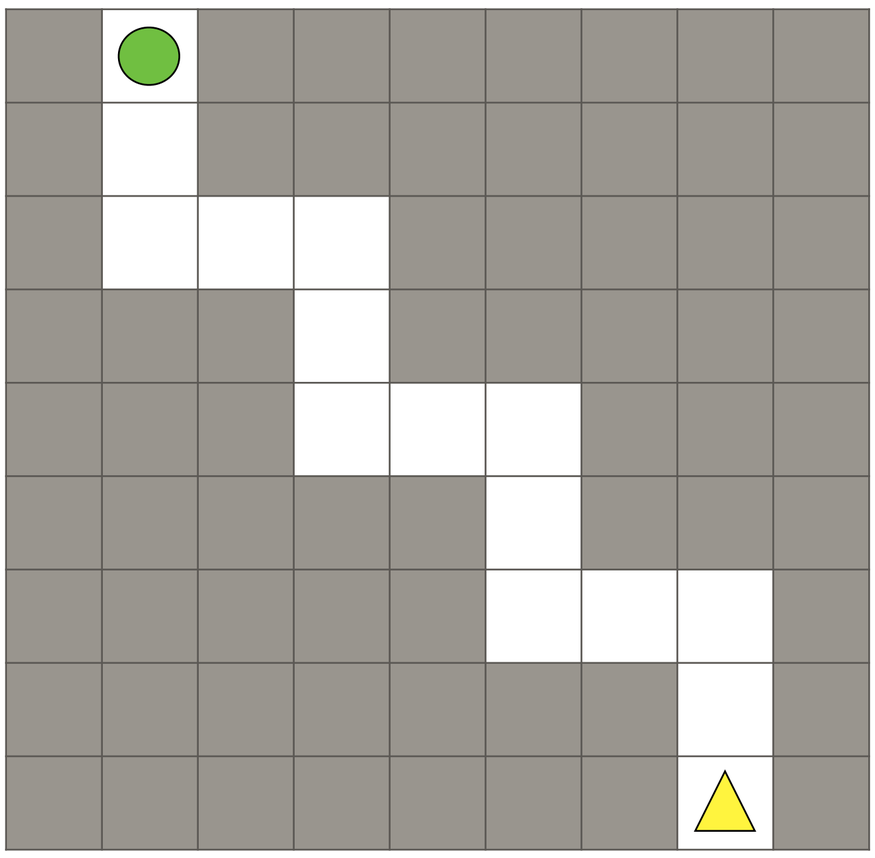
Inputs:
0 = **E** (Empty)
1 = **B** (Block)

Outputs:
11 = **F** (Forward)
00 = **S** (Stop)
10 = **R** (Right)
01 = **L** (Left)

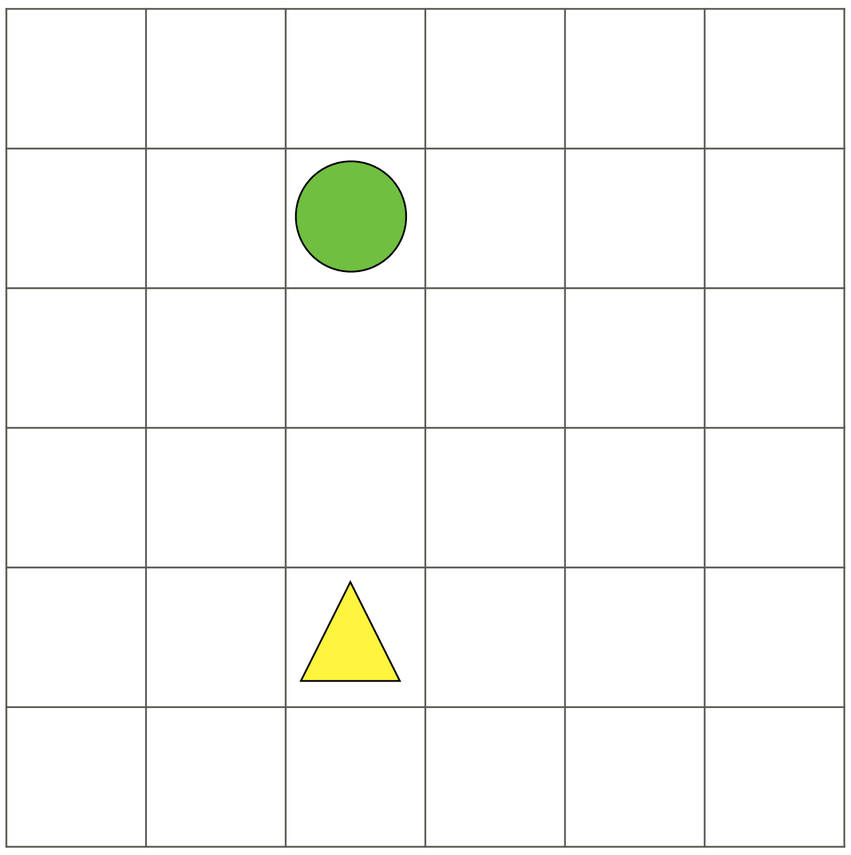
For example, this machine goes forward until it hits a block, then stops:



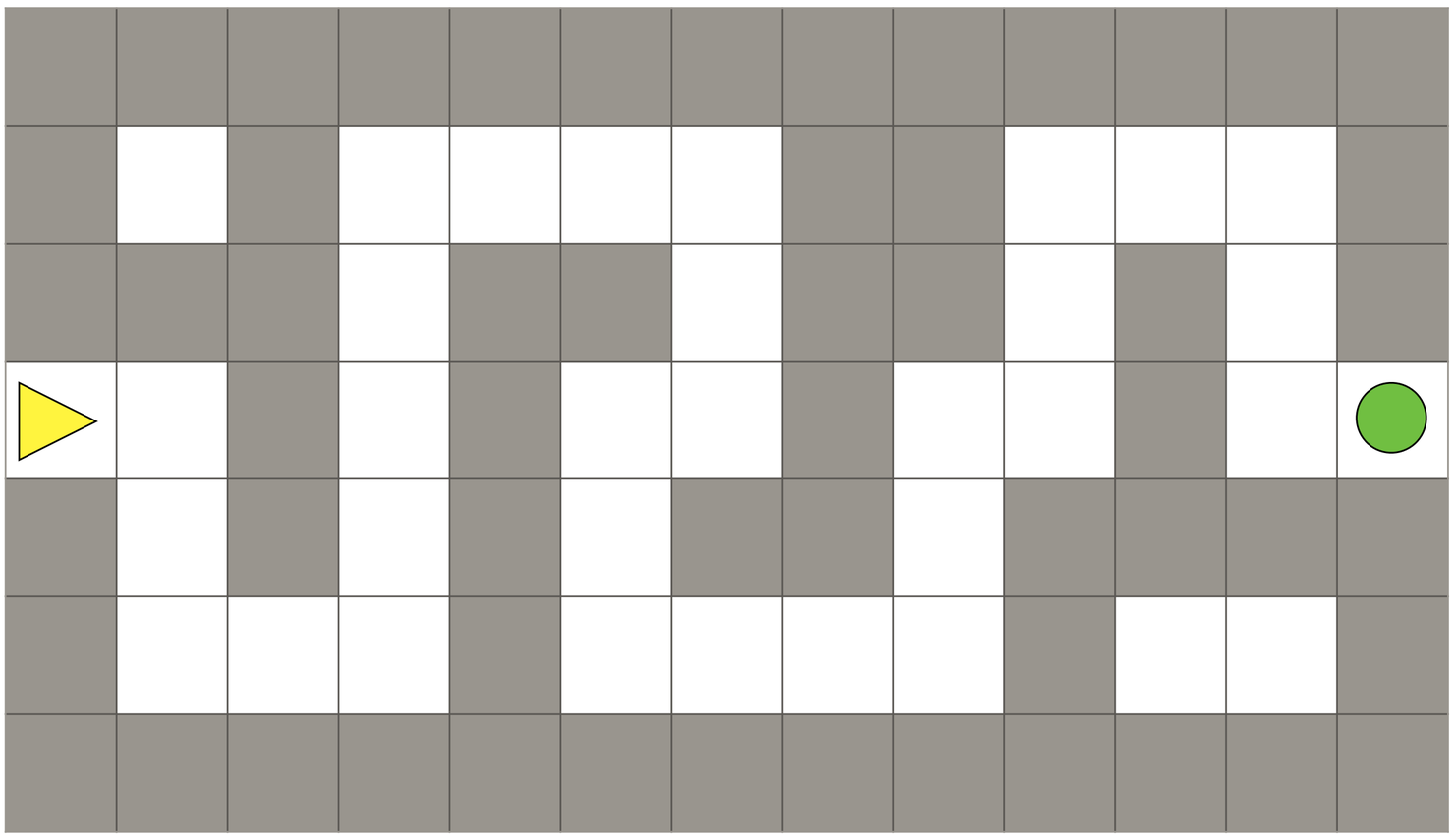
12. **Zig-Zag.** Reach the goal and **keep going**.



13. **Three Ahead.** Reach the goal and **stop**.



14. **Way Finder.** Given *any* non-branching maze, make your way all the way through, and keep going.
A **non-branching maze** is a path that never branches and is never wider than 1-cell across, except on corners where two 1-cell width paths may intersect. Here is an example.*



* This problem was designed by Isaijah Johnson and Jack Schaeffer, Phil 133 W19.

II. FSM Arithmetic and Beyond

Design FSMs that compute the following functions. **Don't** design an SC first to get your solution; **do** think through the FSM directly.

4. +1 T
5. +2 T
6. +1 B
7. +2 B
8. +4 B
9. 2*x* B

Take away problem 9! (Or: replace it with the problem: given the circuit, was is the FSM?)

Caution!

Function composition **does not work** for FSMs. You can't combine two +1 machines to get a +2 machine.

This has to do with the fact that FSM diagrams show states of the *overall* machine, not the flow of information *through* it.

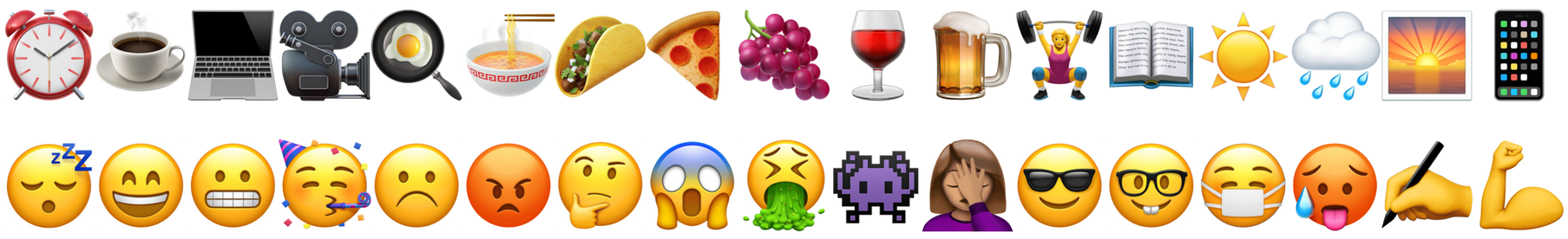
10. **ON/OFF Switch**

This circuit toggles between outputting 1's or 0's; it switches each time it receives a 1, but keeps the same output if it receives a 0. Start in the off state, where output = 0.

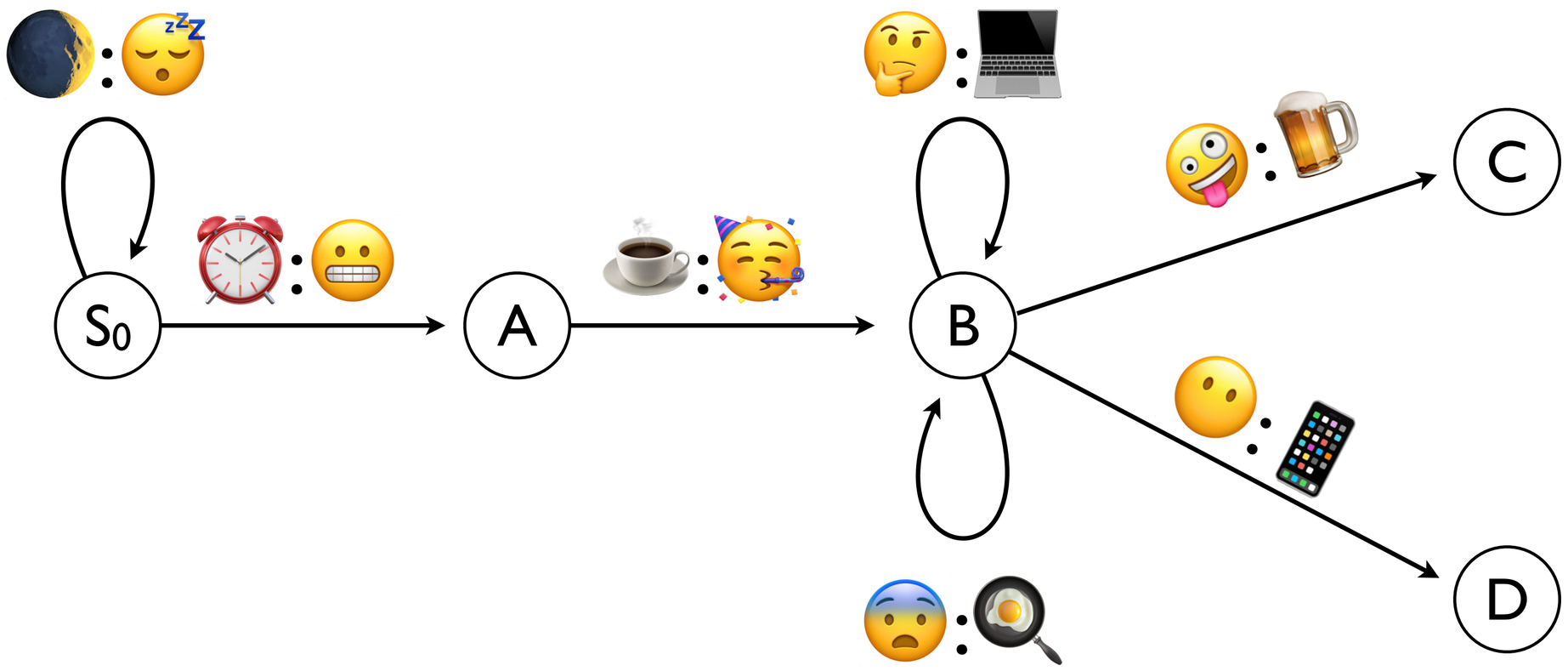
11. **The Daily Grind**

Design an FSM which captures the cycles and possible paths of your typical day during quarantine. Make it one big loop to take you from morning to morning. Feel free to add alternative inputs and outputs as appropriate.

Possible inputs and outputs:



For example, here is a fragment of my day:



HW5. Computing with TMs

2.18.25

Rules

- **Annotation:** all machines must be annotated.
- **Arguments:** Tally problems are defined on arguments $x \geq 1$; for binary problems, $x \geq 0$.
- **Blocks:** blocks are define differently for tally and binary.
  - Tally block = a single block of 1's.
  - Binary block = a block of 0's or 1's, flanked by *s.
- **X- and Y-Blocks:**
  - For 1-place functions, the tape starts with a single block.
    For 2-place functions, the tape starts with two blocks.
  - When a problem involves two arguments x and y:
    - The x-block is on the **right**, and the y-block is on the **left.**
- **Standard Position.**
  - All problems begin in standard position, and must return to standard position.
    - Tally SP = the right most cell of the right most block.
    - Binary SP = the right most * of the right most block.
- **Final configuration:**
  - All problems must end in standard position.
  - All problems must end with a well-formed output: a single block.
- **Boxes**
  - You can use a boxed version of any problem you have solved in an earlier problem, provided it satisfies the criteria for boxing.

I. TM Arithmetic: Tally

Assume $x,y \geq 1$.

1. x+1 T

2. x+3 T.
   Use your solution from (1).

3. $3x$ T.
   Use your solution from (2).

4. x+y T.
   Use your solution from (1).
   For this problem, do not assume that the input blocks will be only separated by 1 cell. They may be separated by any number of cells.

5. $3(x + y)$ T.
   Use your solutions from (3) and (4).

6. $x + 3y$ T.
   Use your solutions from (3) and (4).

II. TM Arithmetic: Binary

Assume $x,y \geq 0$. The following problems have some arbitrary constraints built-in to make them simpler.

7. x+1 B
   • Assume the input block will always have an extra 0 in the left-most cell.

8. x -1 B
   $f(x) = x - 1$ if $x \geq 1$
   $f(x) = 0$ if $x = 0$

9. $x + y$ B
   Use your solutions from (7) and (8).
   • Assume that y-block contain enough 0's padding the left side initially so that the entire block may contain the binary representation of x+y.
   • Don't forget to "clean up" your tape at the end of computation.

III. Algorithmic Design

10. Multiplication: $x \bullet y$

    Using a combination of flow charts, prose, and "snapshots" of the tape, describe a step-by-step *strategy* for computing multiplication. This should not be a complete Turing Machine. (That is a difficult and laborious problem.) But it should be sufficiently detailed that you think someone *could* build a Turing Machine from your instructions, if they knew what they were doing. Your choice of Tally or Binary.

11. Multi-function machine

    Design a machine which can perform different functions depending on the user input (at least successor, addition, multiplication), similar to a calculator. There are many way to do this, so be creative. Use a combination of flow charts, prose, and "snapshots" of the tape; don't design a completed Turing Machine.

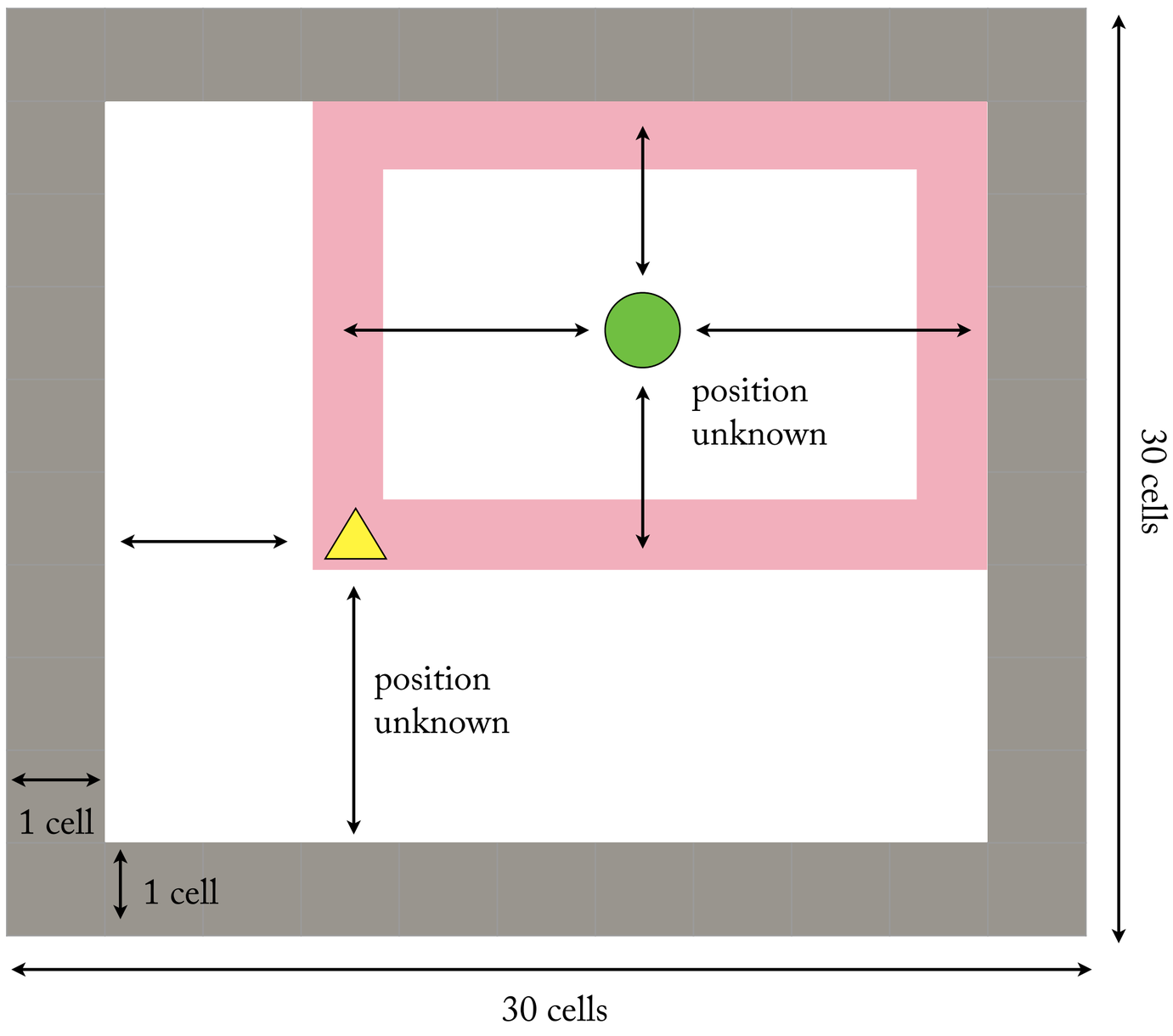
HW6. Navigation with Turbots

2.18.25

The Desert Ant

Goal: Search until you find food (and pass over it), then return to where you started as quickly as possible. You may choose any initial configuration for the tape.

- **Assumption 1:** the food will always appear in the NE quadrant relative to the Turbot's starting position (and not at the same position as the Turbot itself.)
- **Assumption 2:** the world is 30 X 30 cells, surrounded by a wall.
- **Constraint:** use a maximum of 20 cells on the Turing Machine tape.
- **Unknowns:** the starting position of the Turbot in the world is unknown, the position of the food within the NE quadrant of the Turbot is also unknown.



1. Flow-chart.

Design a high-level flow-chart that describes your solution to the Desert Ant Problem. Be sure to describe both the design of the machine and the configuration of the tape.

2. Turbot

Construct a TM that follows flow-chart in in (1) and solves the Desert Ant Problem. Use modular boxed functions when you can. Label all the major parts of your design according to their function.

For very similar functions, it is sufficient to provide one solution, and indicate the obvious change to the other. (For example, if you have one sub-part that handles left turns, and a symmetrical one that handles right turns, you can just design one of these, and describe the other one in words.)

3. The Life Cycle

Suppose that, after solving the Desert Ant Problem, your Turbot must stay at home for a fixed amount of time (say: 20 clock cycles), then return to the food, and the whole cycle must repeat again.

Assuming the food never moves, how would you change your initial design to accommodate this challenge? How would you change the tape? Describe your approach in prose, or with the help of a flow chart, but don't design the actual Turbot.

HW7. Final Project

11.11.25

Outline

Read this entire document carefully.

Length

The assignment should be 6-10 pages in length, ~1.5 spaced. Be sure to cover your topic, but there is no bonus for extra length.

Format

The assignment should be a roughly equal mix of prose and diagrams. (Be creative!) Be sure to explain your diagrams in the text. (Look at the Petzold readings for a good model here.)

Ethos

The point of the assignment is to get you to both

(1) engage with the big philosophical ideas in the course, but also

(2) apply the technical ideas you've learned to specific cases.

Ask not: how little can I do to do well on this assignment? Ask instead: how much knowledge can I effectively deploy in such a short space?

Above all, I would ideally like this to be *interesting to you.* It will show.

Content

- Your final assignment is to write an essay that connects the subject matter of this class to a particular philosophical or scientific theme of your choice. Potential themes are listed on the next page.
- Your paper must address the relationship between your theme and the Computational Theory of Mind.
- At some point in your essay, you should address these questions:
    - What are the key claims of the Computational Theory of Mind?
    - How should we understand the concepts of representation, computation, and algorithm?
    - Is the Computational Theory of Mind plausible? Is the computational approach the right one to explain how physical cognition is possible?
- Whenever possible, illustrate your ideas with application to specific examples.

Possible themes

1. **A topic of your choosing from cognitive science or philosophy**
   Investigate a topic of your choosing from contemporary cognitive science, using the tools and techniques developed in this class. (Cognitive science construed broadly to include: psychology, neuroscience, computer science, linguistics, philosophy.) Show how something similar (perhaps radically simplified) could be analyzed using circuits, FSAs or TMs.

2. **Letter to Leibniz**
   Write a letter to Leibniz responding to his argument that physical cognition is impossible. Was he right? Why or why not? How does the material from the course bare on this question?

3. **The Long View**
   Summarize the whole class, from logic gates and functions to UTM. This is an exercise in efficiently compressing a lot of information into a short space, while carefully choosing what to highlight, and what to skip.

4. **Lessons Learned**
   What are the 3 most important lessons you've learned in this class about computation? What are the 3 most important lessons you've learned in this class about the philosophy of mind? How are these lessons related?

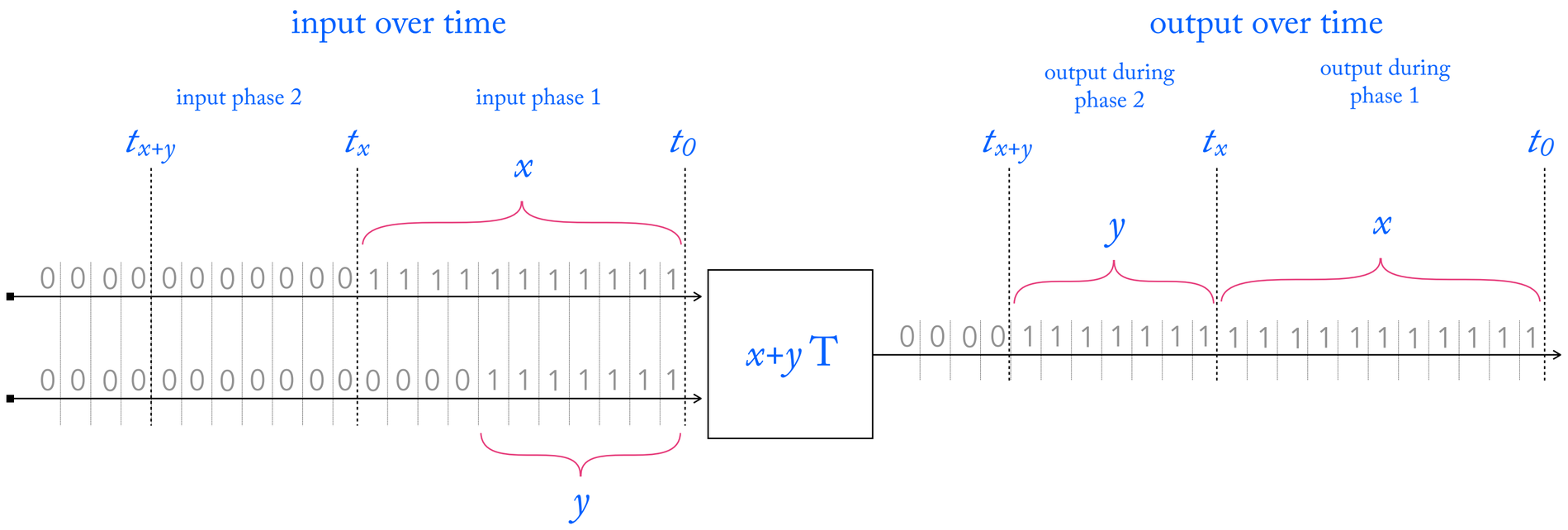
5. **Levels of abstraction**
   Explain the idea of abstraction. Illustrate the four different levels of abstraction discussed in this class, from physics up to functions. Discuss how to shift from one level up to the next. Explain the idea of multiple realizability. Give examples of multiple-realizability at every stage.

6. **Levels of power**
   Explain the idea of computational power. Illustrate the four different levels of power discussed in this class, from logic gates up to Turing Machines. Discuss the physical differences between each type of machine (i.e. different uses of memory). Illustrate the differences in power between each level (i.e. functions that one level can compute but the other can't).

7. **Navigation**
   Come up with an interesting navigation problem that goes beyond the dead-reckoning problem discussed in class. It should have some real-world counterpart. Solve the problem both in outline (as a flowchart) and as a state digram. Discuss the role that representations play in the solution.

8. **Universal Turing Machine**
   Develop a detailed design for a UTM. It need not be an actual Turing Diagram. But the description should break down the operation into small enough parts that you can be confident that someone would be able to actually build it.

The Limits of SCs



Why $x+y$ T can't be computed by an SC.

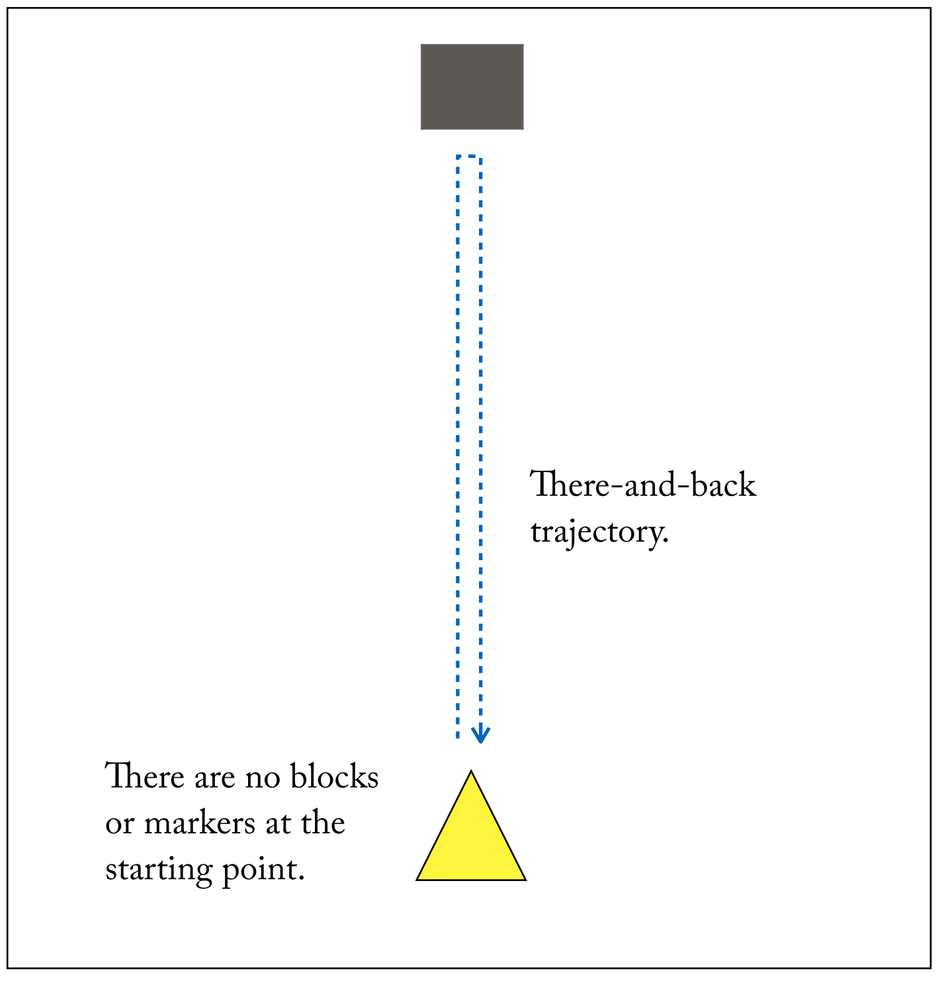
- Suppose for the sake of argument that there were such a machine…
- And suppose for the sake of argument that x is the bigger of x and y (if they're not equal).
- The machine must take in <u>two</u> streams of 1's, one of length x, the other of length y, then outputs <u>one</u> stream of $x+y$ 1's.
- But it can output at most one 1 per unit of time. So during input phase 1, it can output at most x 1's.
- To get the second set of 1's out, it must somehow remember the number y.
- But y can be *any* number.
- The problem is that a given SC only has a fixed number of memories. So an SC can't remember *any* number.
- As a consequence, an SC can't compute $x+y$ T.
- But for any bound n, there is some SC that *can* compute $x+y$ [0-n] T.
- Does this argument apply to $2x$ T? $x+y$ B? $x \bullet y$ B?

An SC can't solve the **Mad Max** problem, because to do so, it would have record the distance it travels *no matter how far it travels*.

But the finite memory of an SC limits how large a distance it can record.

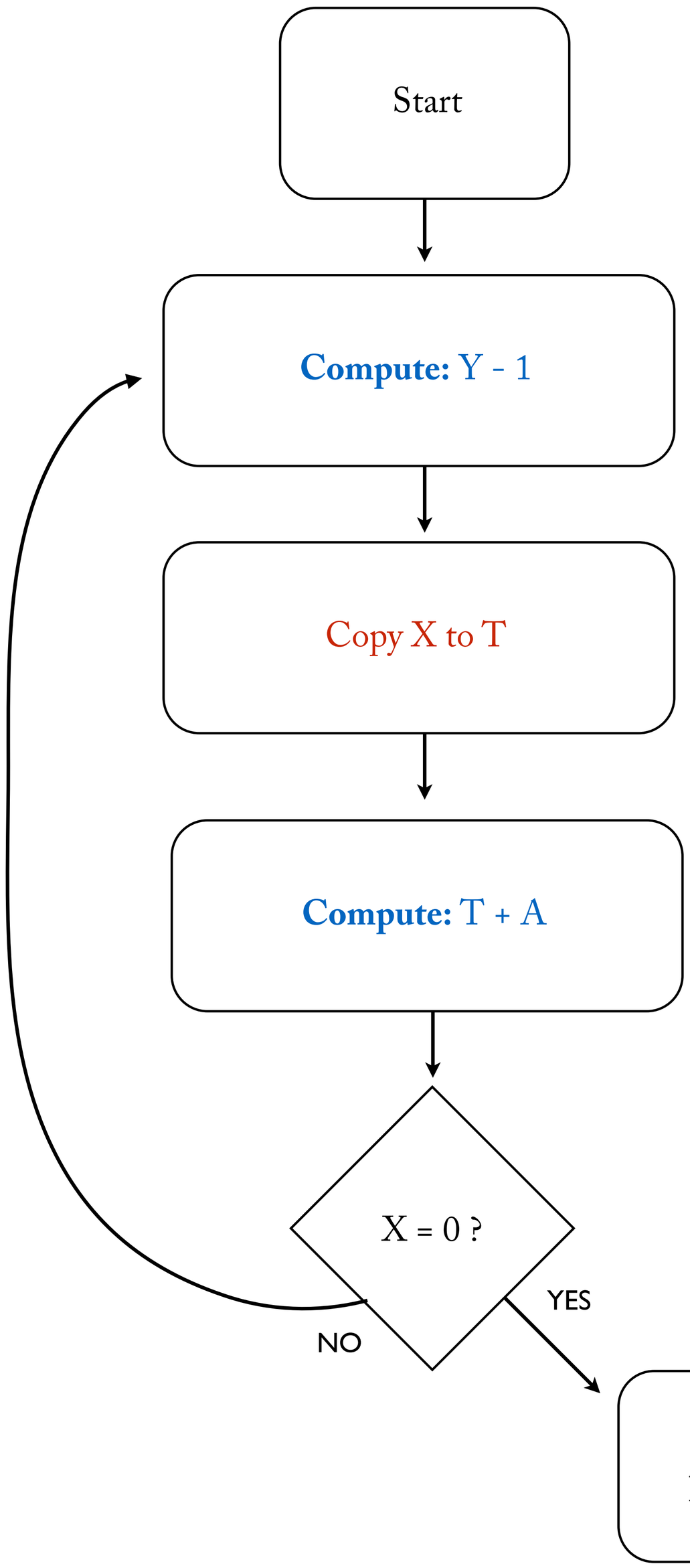
The Memory Moral

The fact that SC's have only fixed and finite memory means that they cannot compute functions which require indefinite *extensions* of memory. To compute T $x+y$ or Mad Max, a machine will need extendable memory.



Recursive Multiplication

Compute $f(x,y) = x \times y$



How can we break multiplication up into a simple, repeating operation?

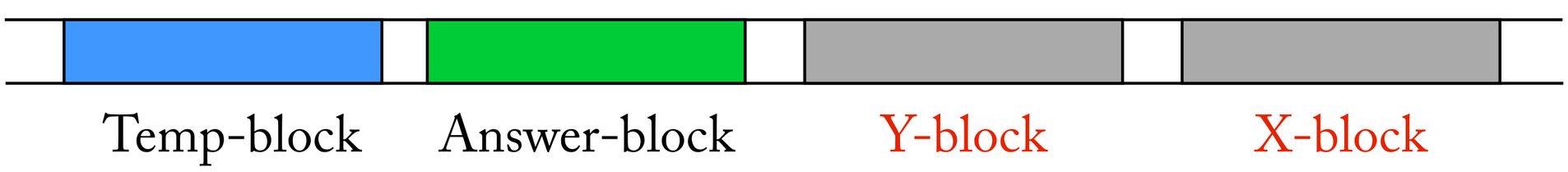
Multiplying x by y is equivalent to adding x to 0, y times.

$2 \times 3 = 0 + 2$
$= 2 + 2$
$= 4 + 2$
$= 6$

3 times

$4 \times 5 = 0 + 4$
$= 4 + 4$
$= 8 + 4$
$= 12 + 4$
$= 16 + 4$
$= 20$

5 times



Extra Problems

1. Represent the number eleven in each of base 1 (tally), base 2 (binary), base 3, base 4, and base 5.