Introduction to Programming paradigms

different perspectives (to try) to solve problems

15 September 2014,
Introduction to Information Systems

Giovanni Sileno g.sileno@uva.nl
Leibniz Center for Law
University of Amsterdam

“Mechanical” computing

Pascal: Pascaline ~ 1650

Helping his father (tax accountant of Normandy, appointed by Richelieu), Pascal invented a machine for *mechanic calculation*, performing **addition** and **subtraction**.

Blaise Pascal

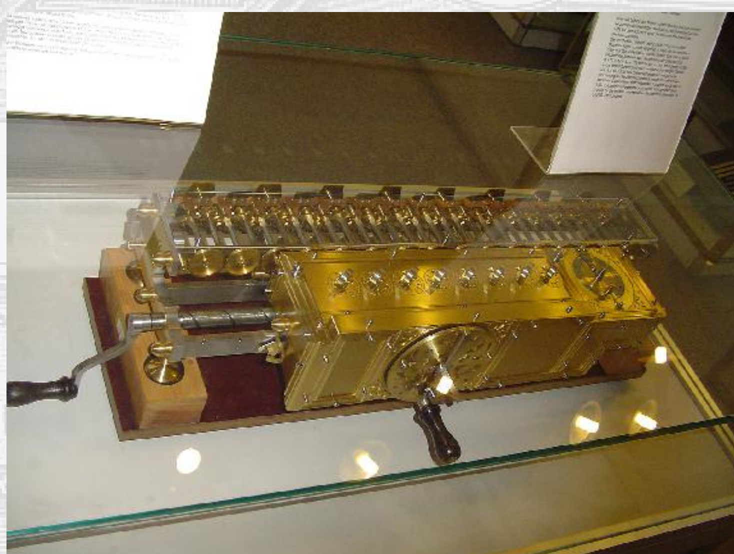
Schickard: Calculating Clock ~1625

Before him, Schickard had already invented an "artithmetic instrument", but unfortunately he was not able to publicly present a full working copy.

Wilhelm Schickard

Leibniz: Stepped Reckoner ~1680

Influenced by the Pascaline, Leibniz proposed a mechanic calculator performing all four operations: **addition**, **subtraction**, **multiplication** and **division**.

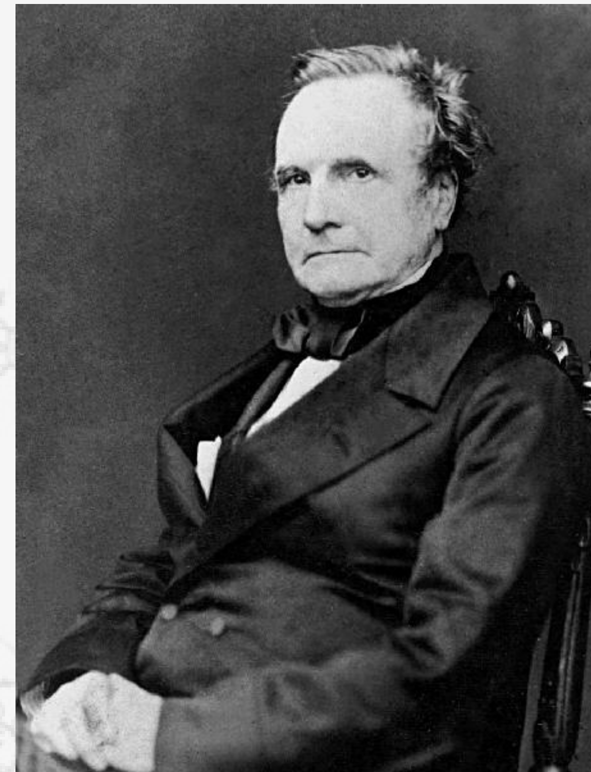


Gottfried Wilhelm von Leibniz

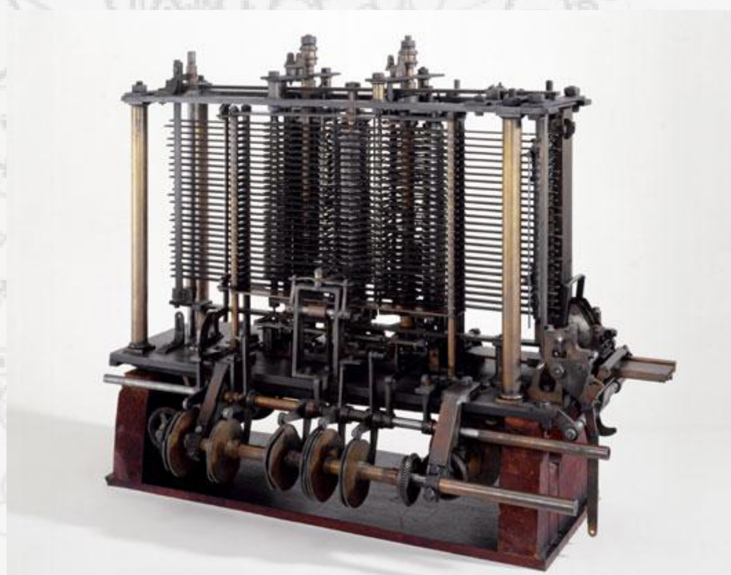
Leibniz Center for Law

Babbage: Analytical Machine ~1840

Extending a project for a *difference engine* to calculate polynomial functions, Babbage proposed a **general purpose** calculator, using *punched cards*.



Charles Babbage



Ada Lovelace ~1840

Collaborating with Babbage, Ada Lovelace is said to be the *first programmer* and pioneer of computer science

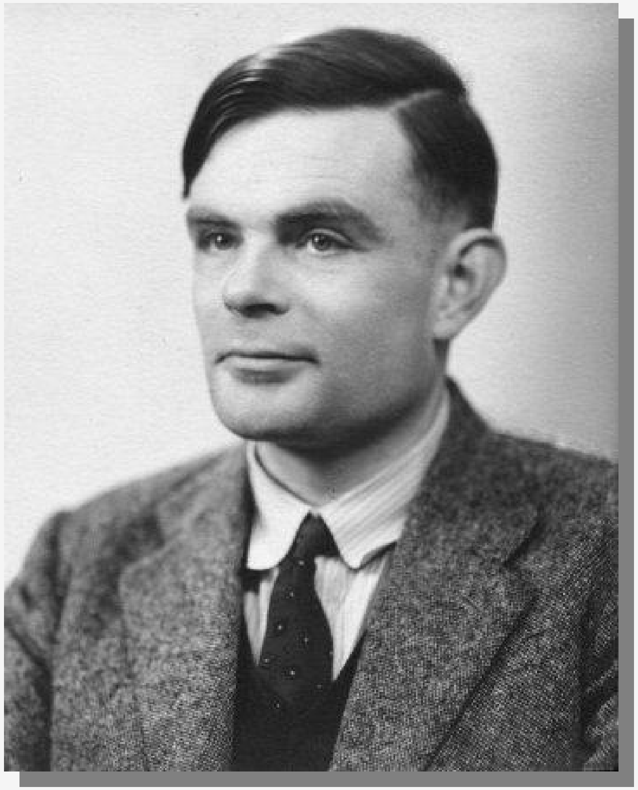
"[..] **developping** and tabulating any function whatever [..] the engine [is] the material expression of any indefinite function of any degree of generality and complexity [..]"

Ada Byron Lovelace

Turing: The Turing Machine ~1936

The formal proof of her intuition arrives only one century later, using an hypothetical device consisting of:

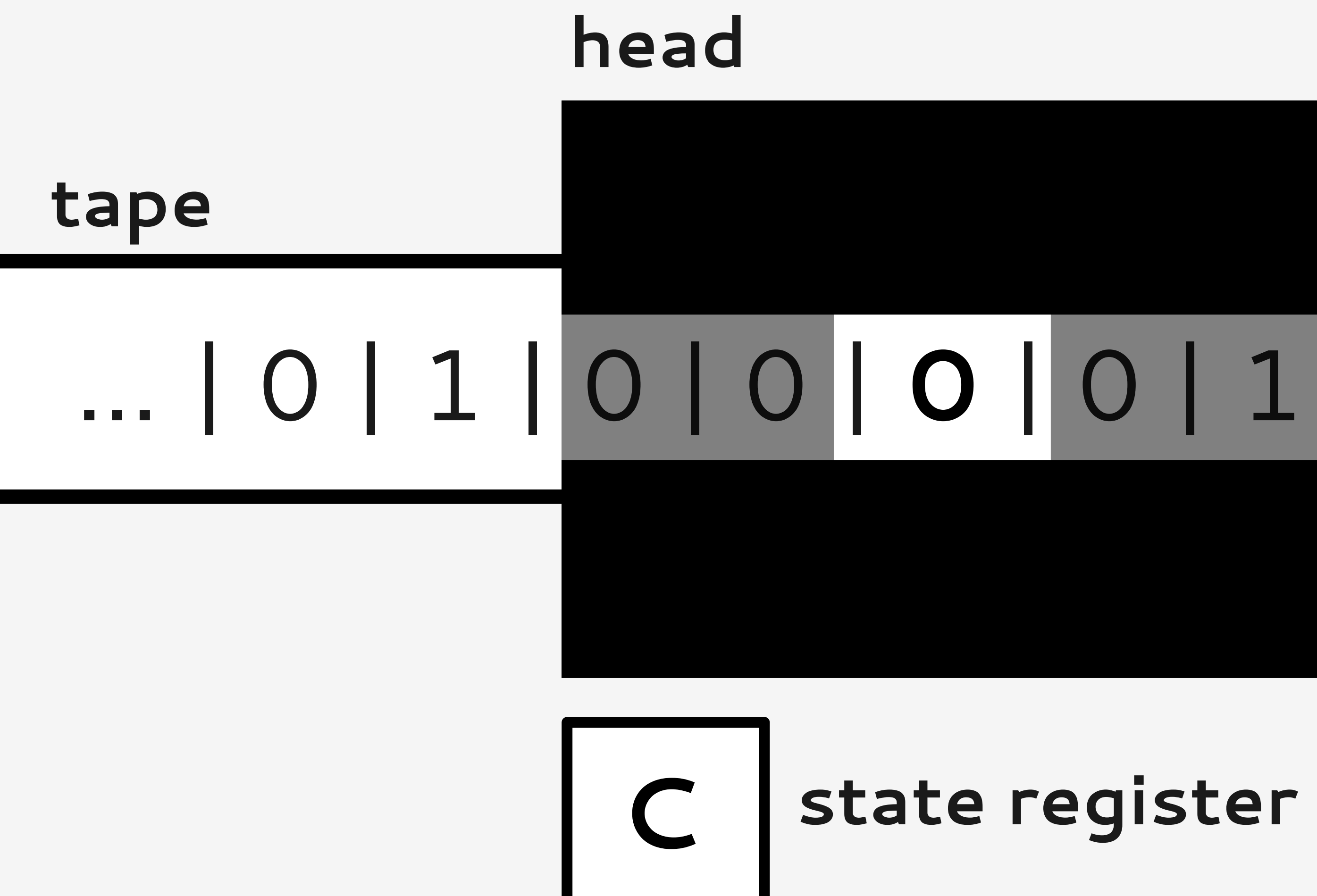
- a **tape**
- a **head**, which can
  - read/write the tape
  - move along the tape
- a **state** register
- an **action table**



Alan Turing

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	Write 1 Move R → B
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

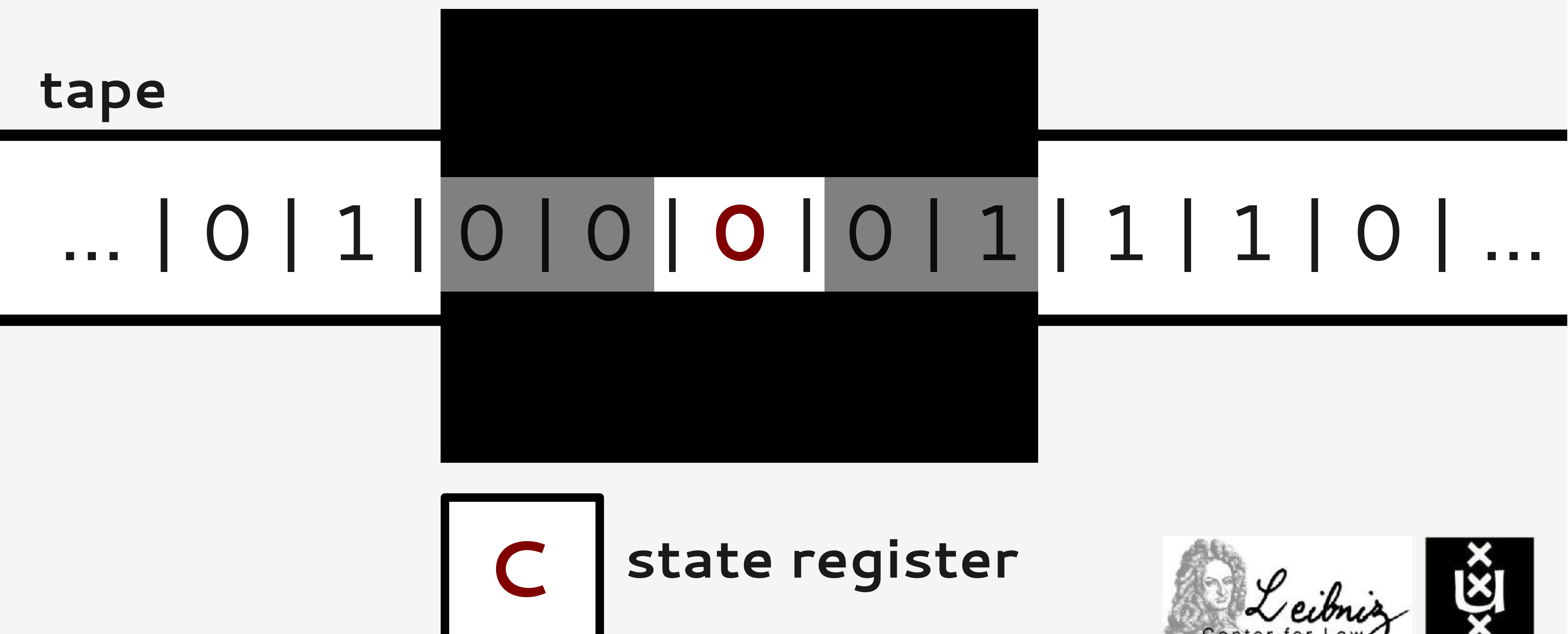


[reading instruction]

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	**Write 1 Move R → B**
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

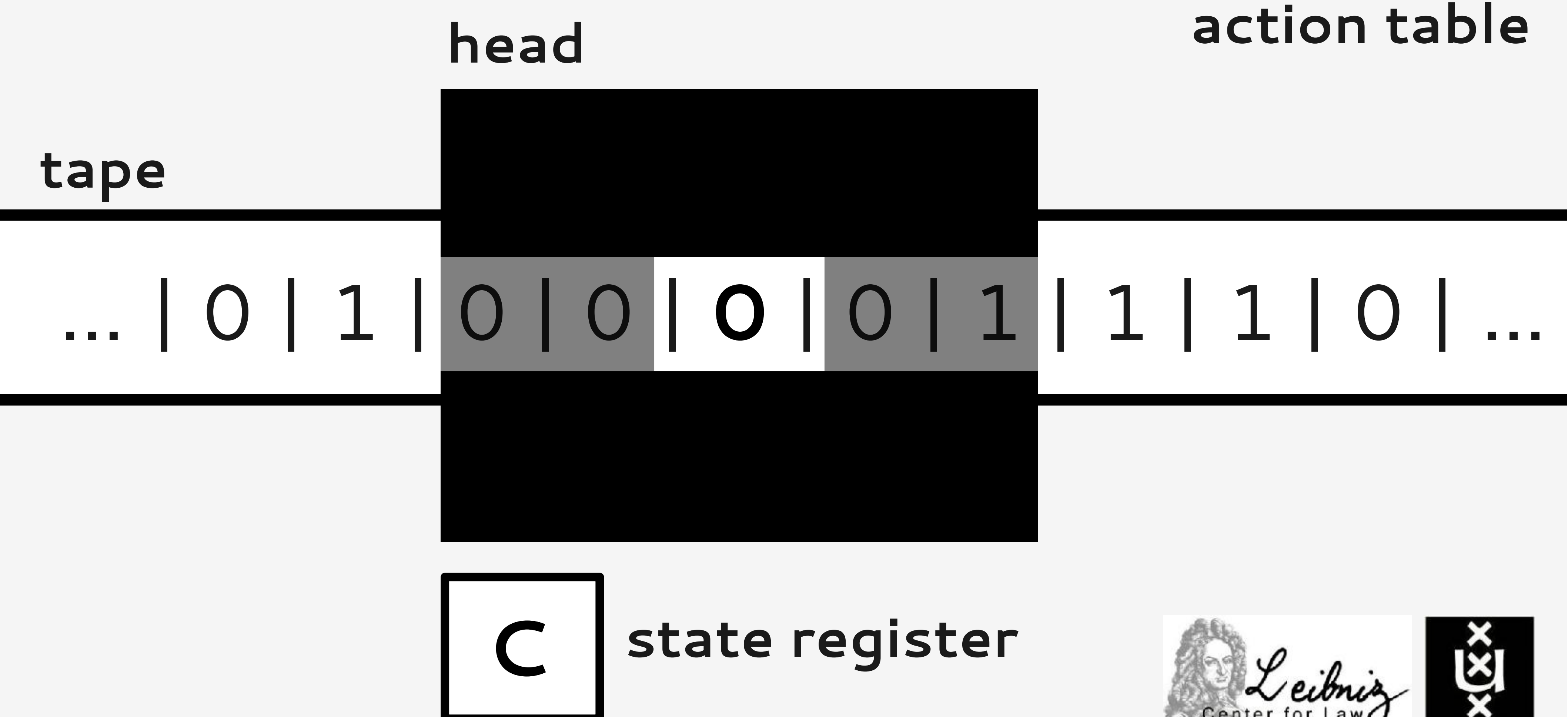
head



[executing instruction 0, C]

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	**Write 1** **Move R** **→ B**
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

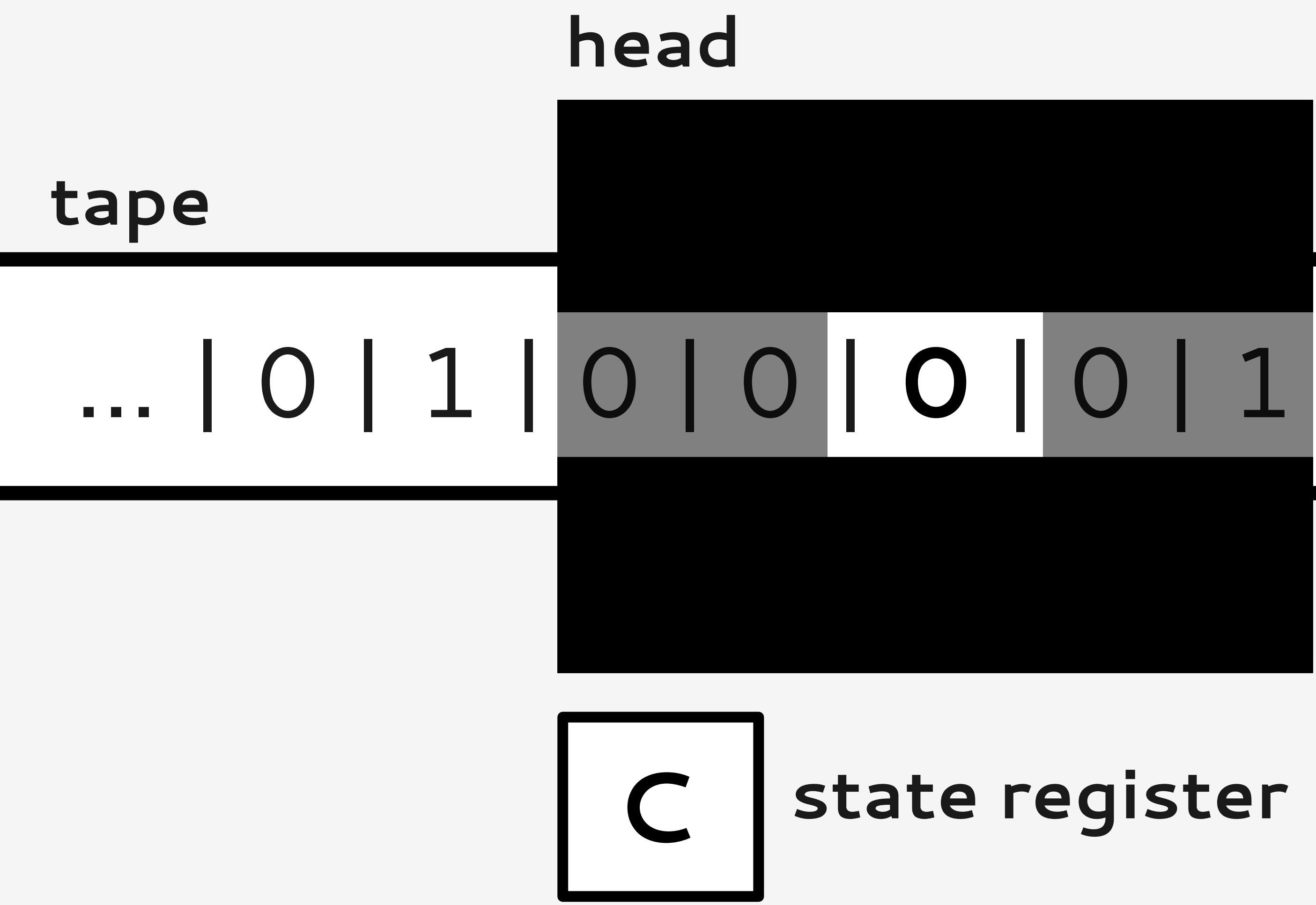


[executing instruction 0, C]

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	**Write 1** **Move R** **→ B**
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

Write 1...



[executing instruction 0, C]

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	**Write 1** **Move R** **→ B**
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

Write 1...

head

tape

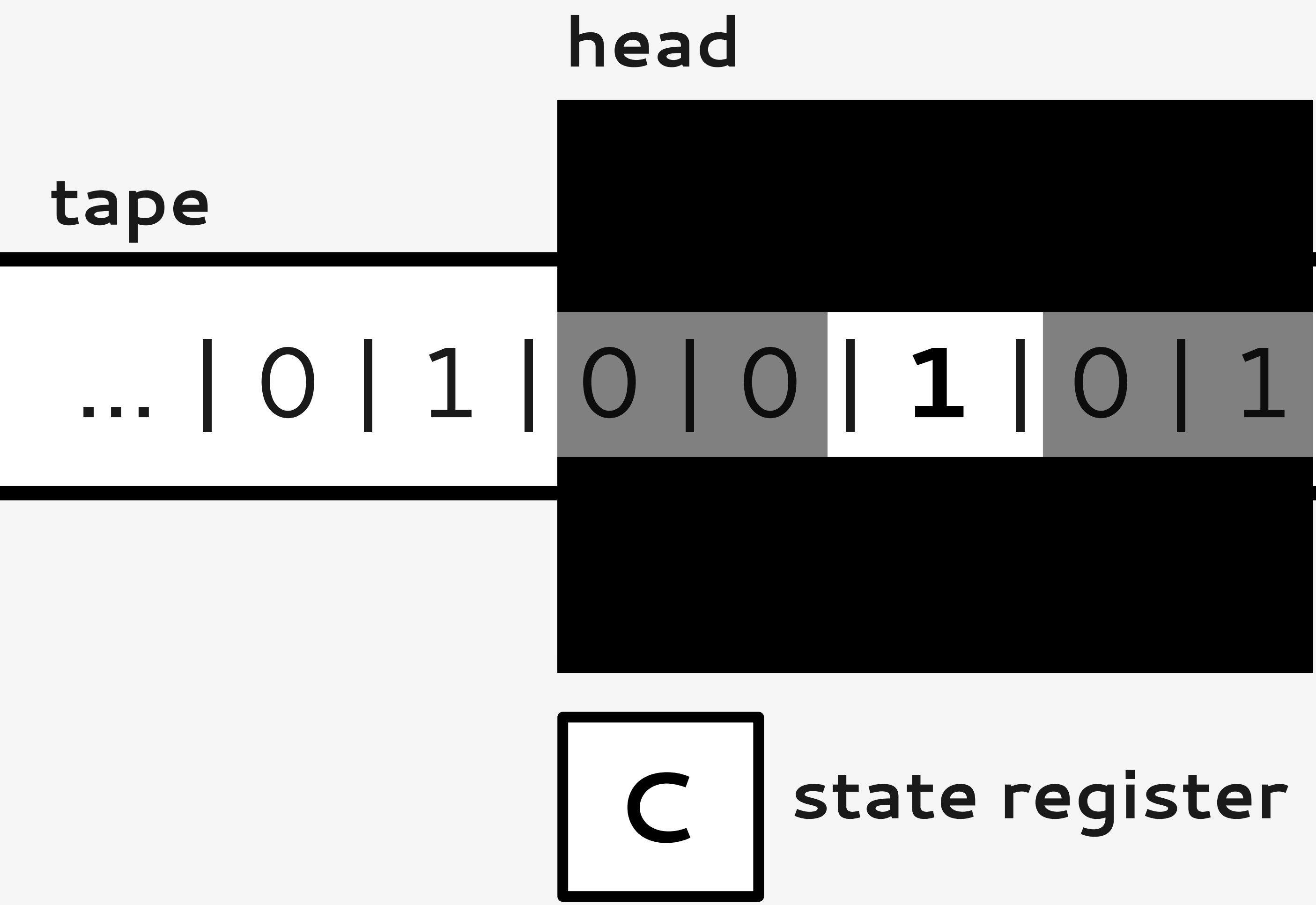
... | 0 | 1 | 0 | 0 | **1** | 0 | 1 | 1 | 1 | 0 | ...

C **state register**

[executing instruction 0, C]

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	~~Write 1~~ **Move R** **→ B**
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

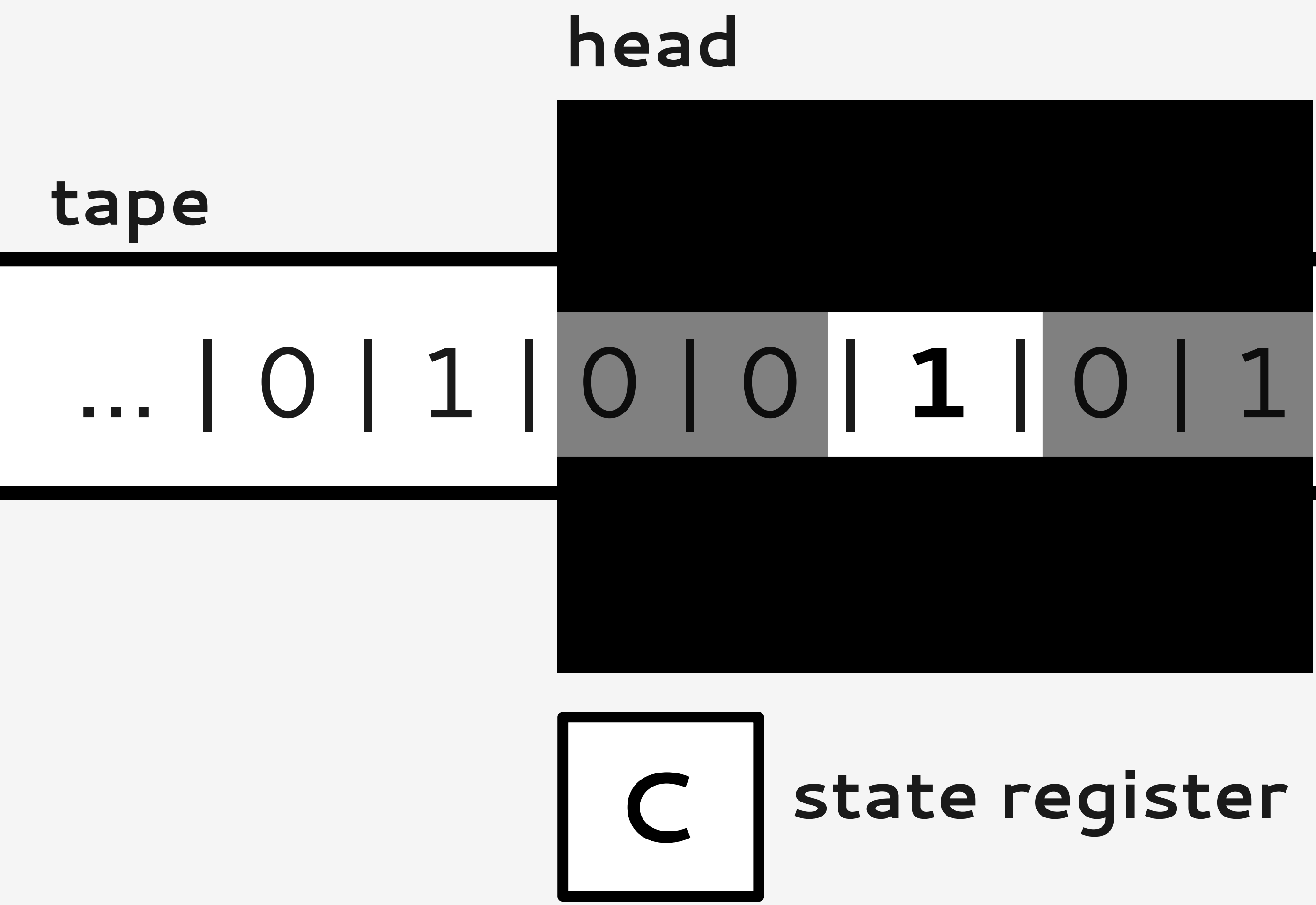


[executing instruction 0, C]

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	~~Write 1~~ **Move R** **→ B**
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

Move R...



[executing instruction 0, C]

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	~~Write 1~~ **Move R** **→ B**
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

Move R...

head

tape

... | 0 | 1 | 0 | 0 | 1 | **0** | 1 | 1 | 1 | 0 | ...

C **state register**

[executing instruction 0, C]

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	~~Write 1~~ ~~Move R~~ **→ B**
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

head

tape

... | 0 | 1 | 0 | 0 | 1 | **0** | 1 | 1 | 1 | 0 | ...

C **state register**

[executing instruction 0, C]

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	~~Write 1~~ ~~Move R~~ **→ B**
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

head

tape

… | 0 | 1 | 0 | 0 | 1 | **0** | 1 | 1 | 1 | 0 | …

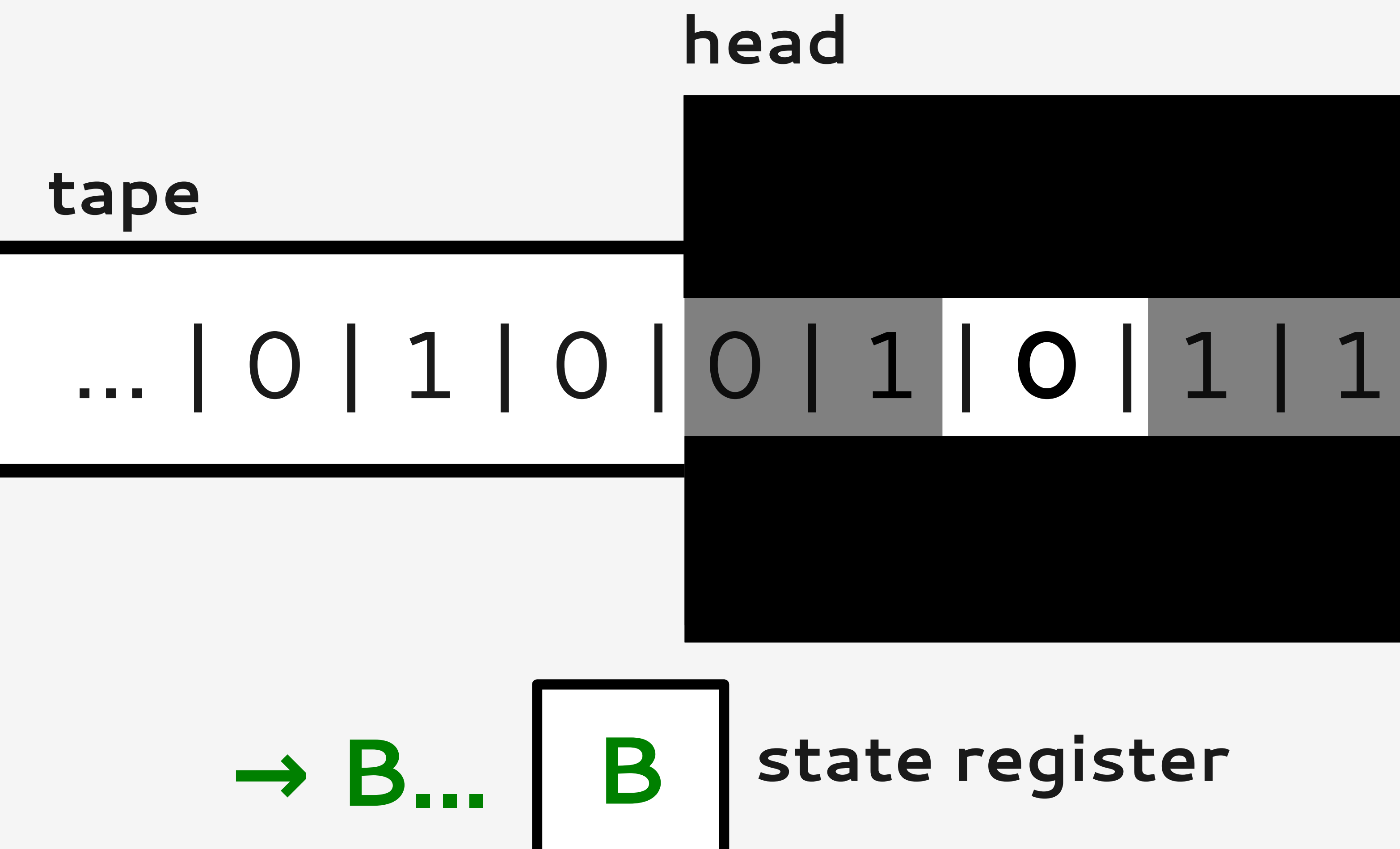
→ B… **C** **state register**

Leibniz Center for Law

[executing instruction 0, C]

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	~~Write 1~~ ~~Move R~~ **→ B**
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

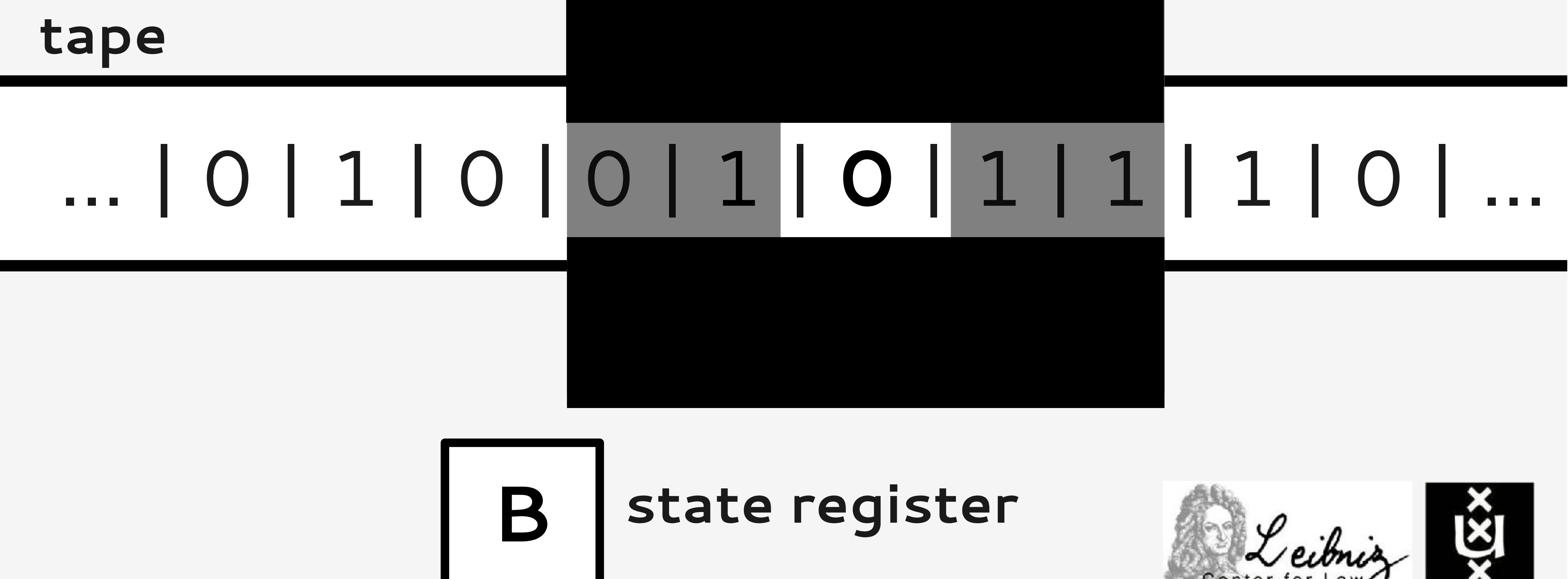
action table



	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	~~**Write 1**~~ ~~**Move R**~~ ~~**→ B**~~
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

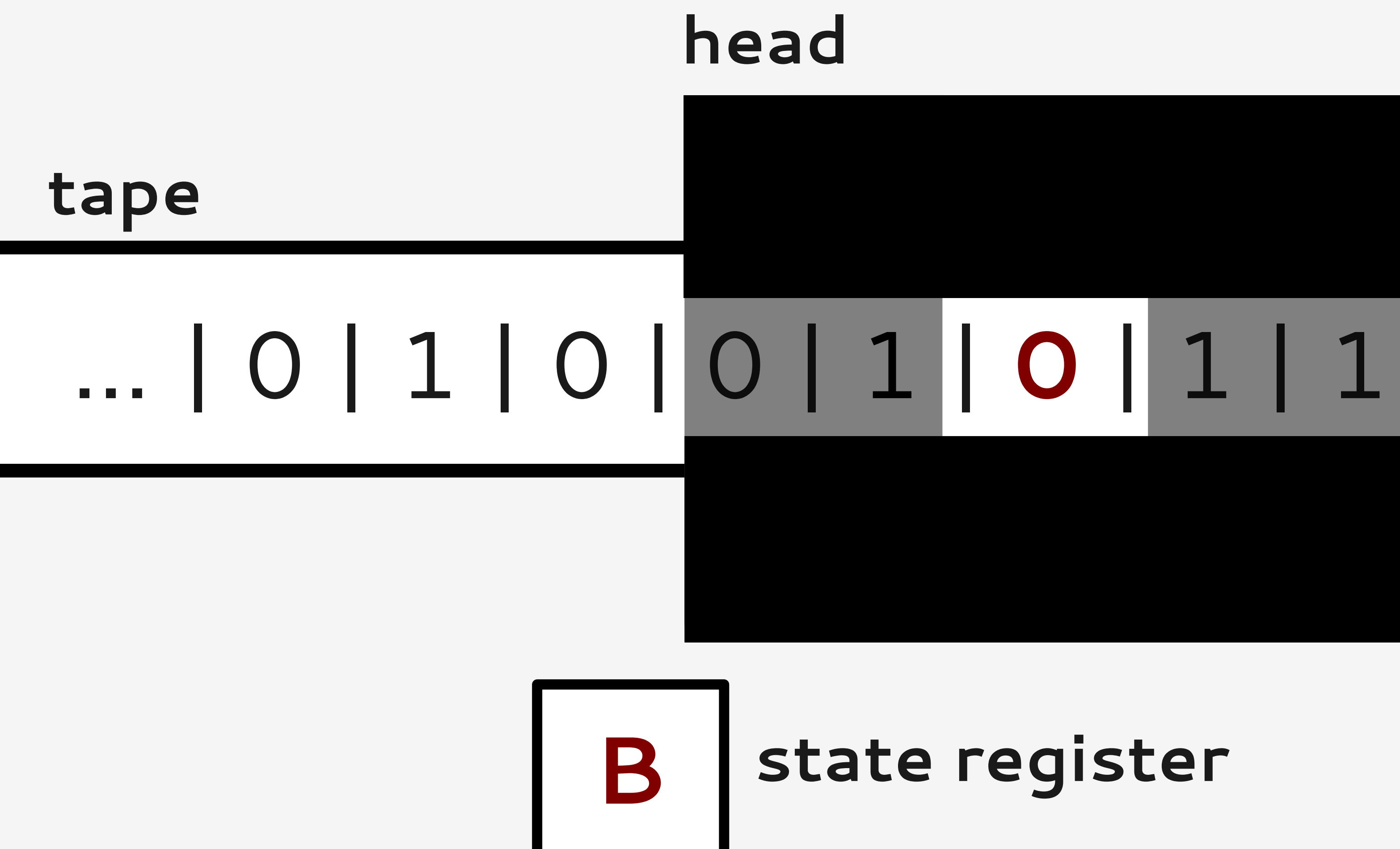
head



[reading instruction] etc. etc.

	A	B	C
0	Write 1 Move R → B	**Write 0 Move L → C**	Write 1 Move R → B
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table



	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	Write 1 Move R → B
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

I/O
peripheral

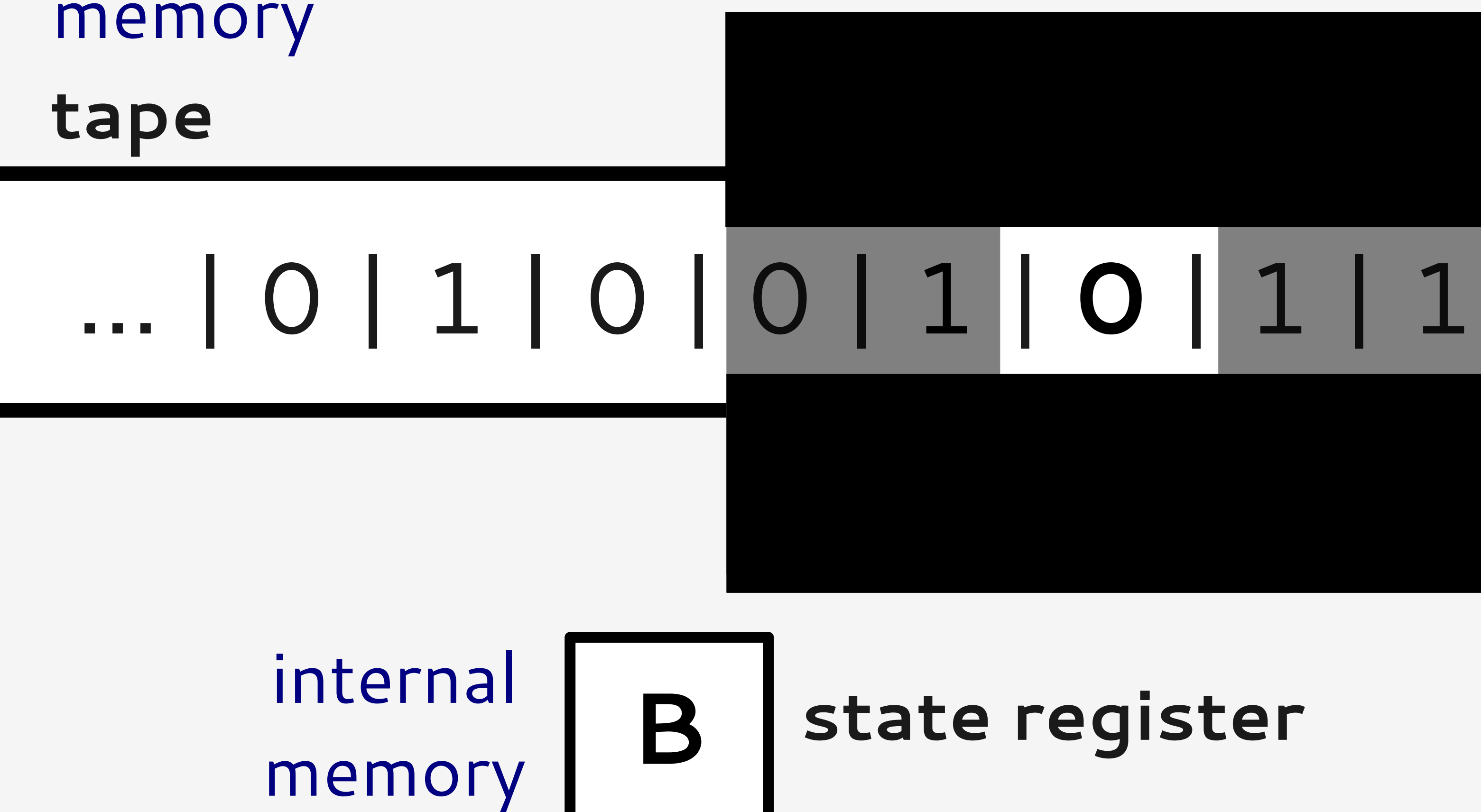
external
memory

head

action table

program

tape



internal
memory

state register

+ something which is operating!

	A	B	C
0	Write 1 Move R → B	Write 0 Move L → C	Write 1 Move R → B
1	Write 0 Move L → A	Write 1 Move R → B	Write 1 Move N HALT

action table

program

I/O peripheral

head

external memory

tape

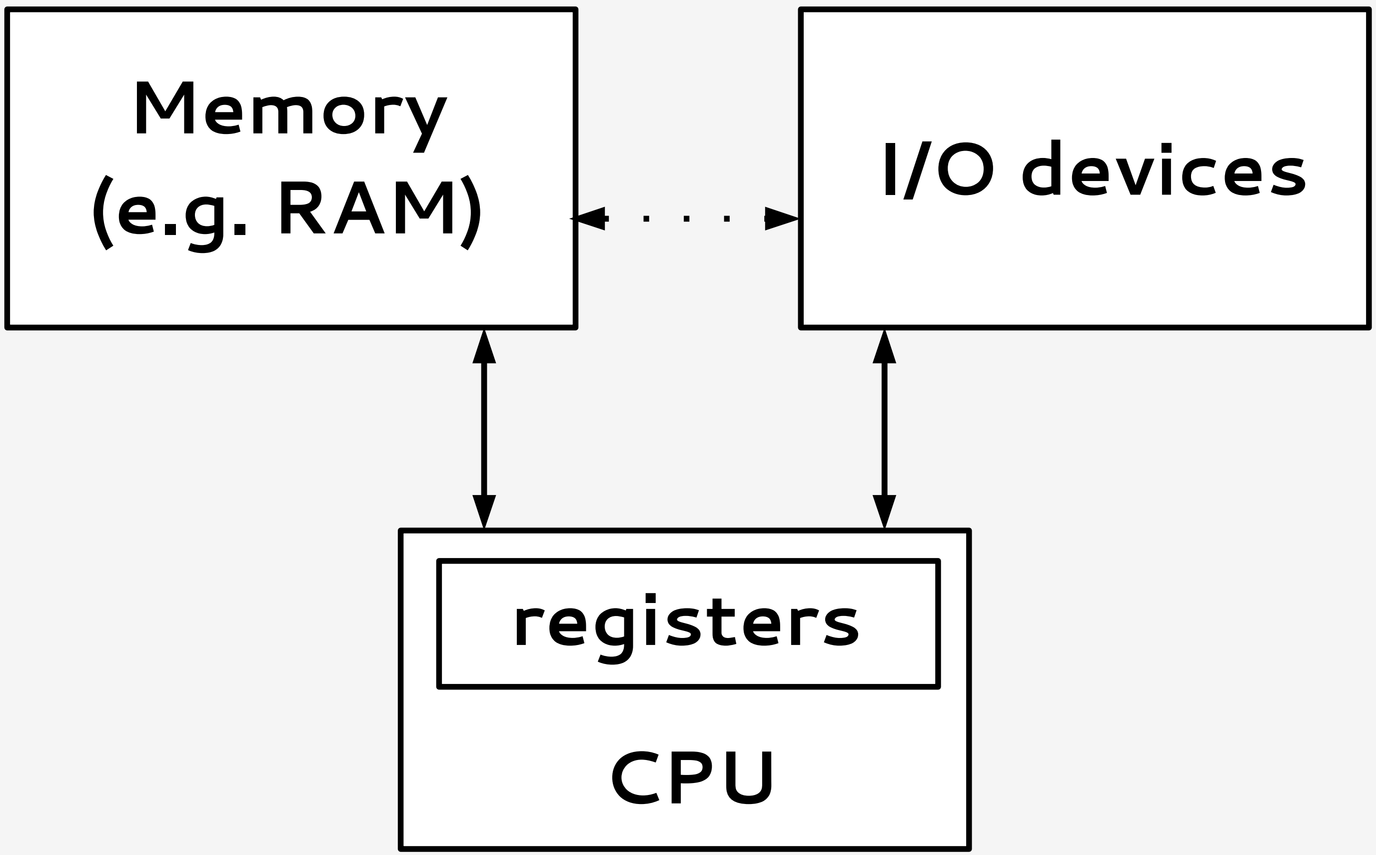
… | 0 | 1 | 0 | 0 | 1 | **0** | 1 | 1 | 1 | 0 | …

internal memory

B

state register

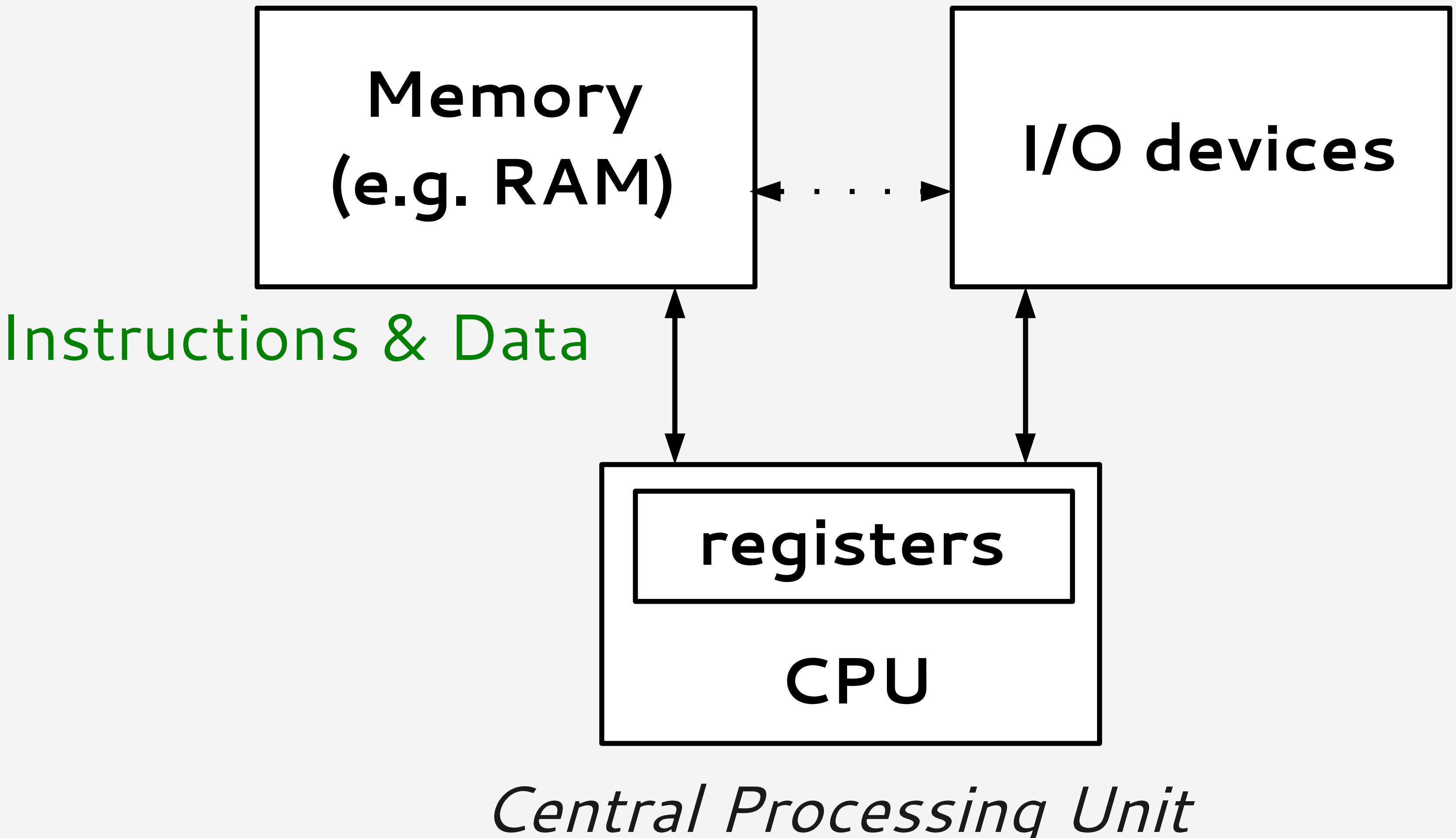
A modern computer (roughly)



Central Processing Unit

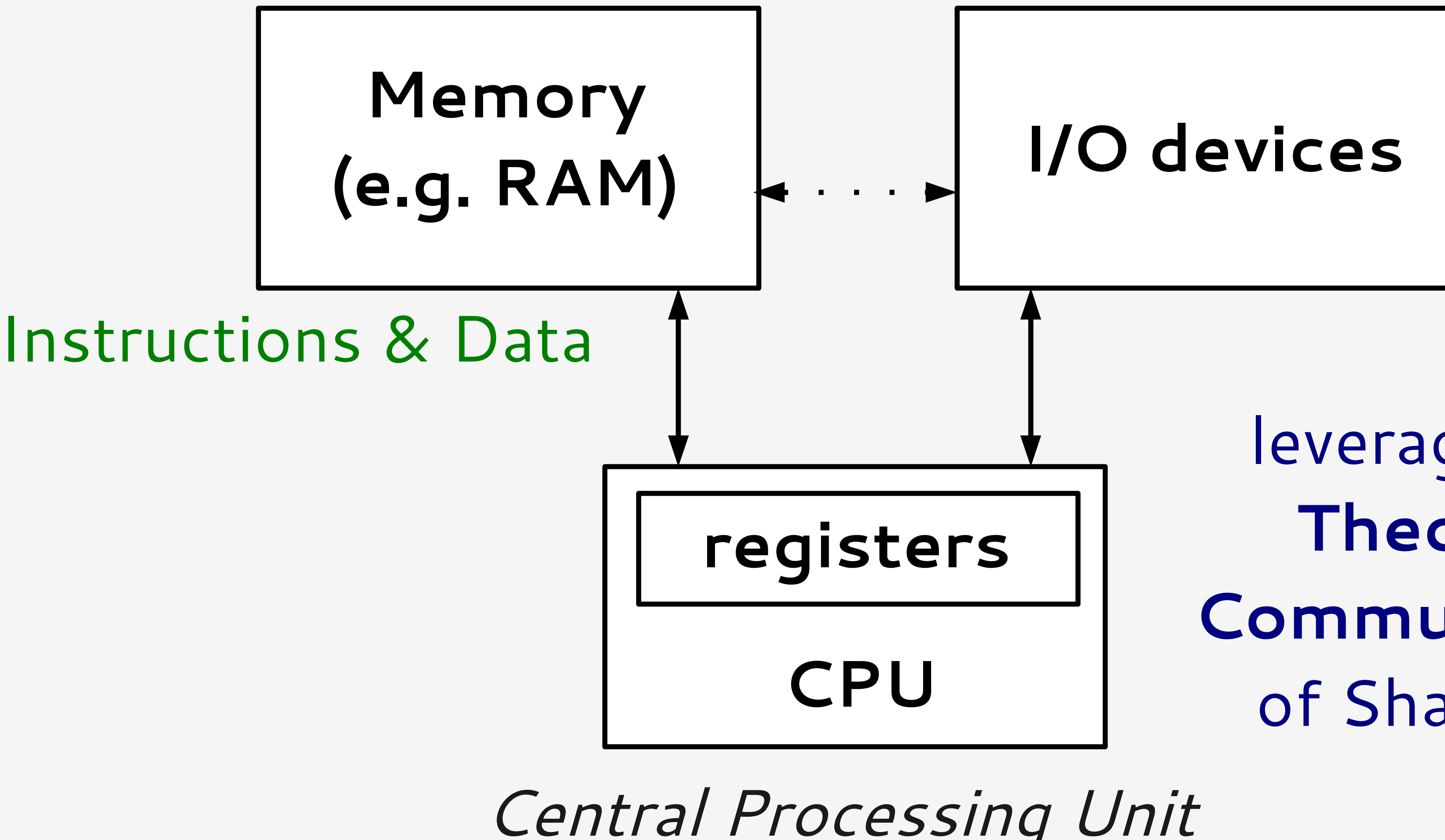
~ Von Neumann architecture

A modern computer (roughly)



~ Von Neumann architecture

A modern computer (roughly)

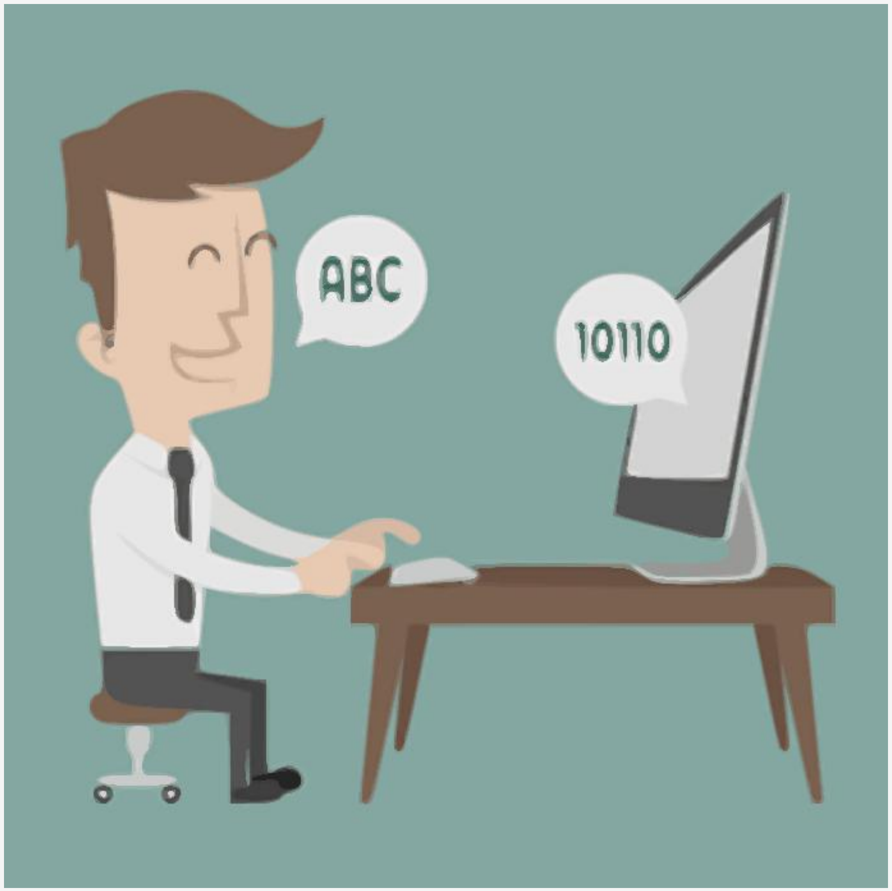


leveraging the **Theory of Communication** of Shannon!!!

~ Von Neumann architecture

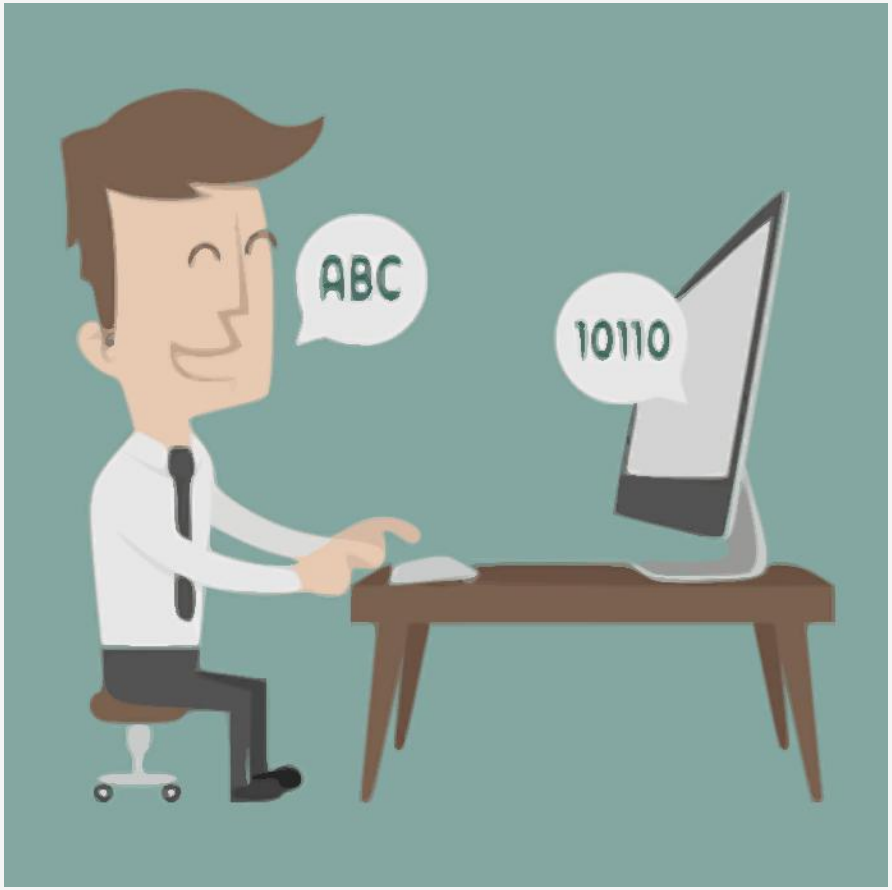
Programming computers

Machines as symbol handlers



Starting from the Pascaline, computing machines respond to the need to displace tedious, repetitive (symbolic) work.

Machines as symbol handlers

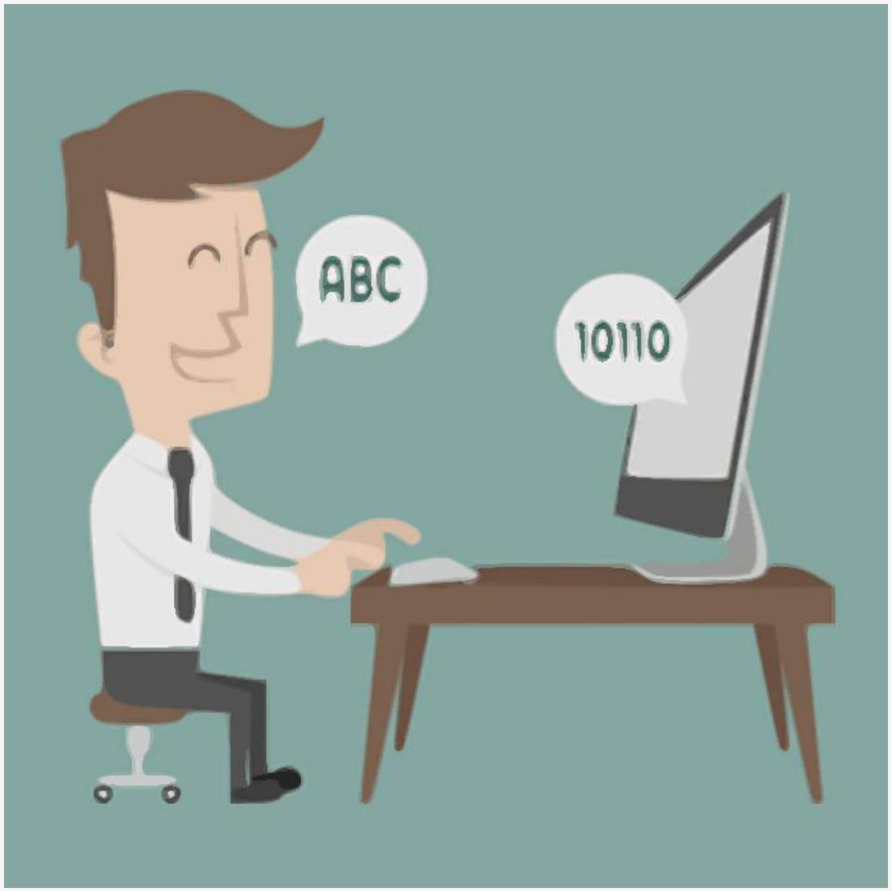


Starting from the Pascaline, computing machines respond to the need to displace tedious, repetitive (symbolic) work.

PS. "*A physical symbol system has the necessary and sufficient means for general intelligent action*" Allen Newell and Herbert A. Simon, Computer Science as Empirical Inquiry: Symbols and Search (1976) → (G.O.F.) **Artificial Intelligence**

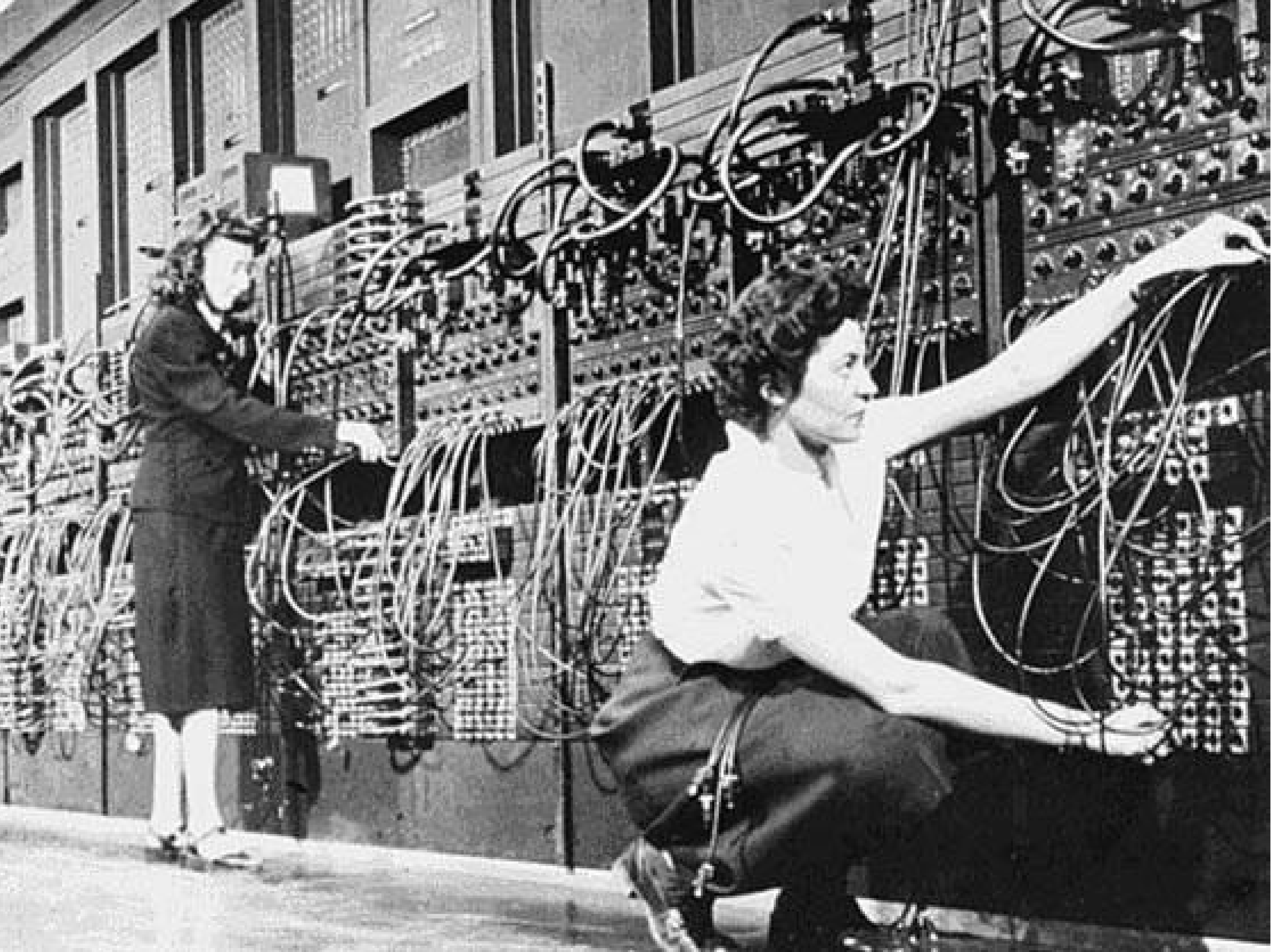
Leibniz Center for Law

Machines as symbol handlers



Starting from the Pascaline, computing machines respond to the need to displace tedious, repetitive (symbolic) work.

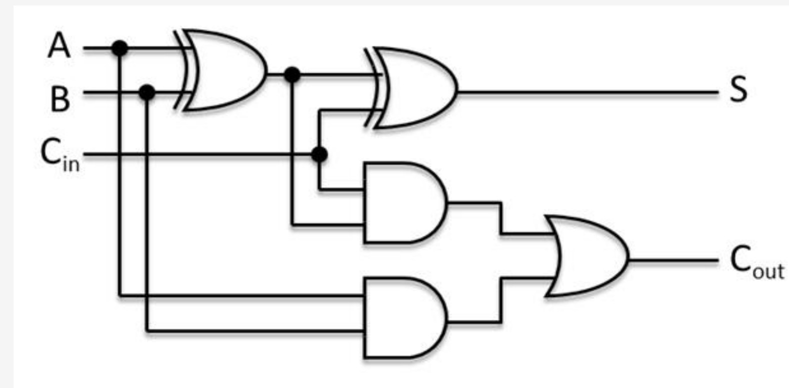
but **how to say to the machine what to do?**

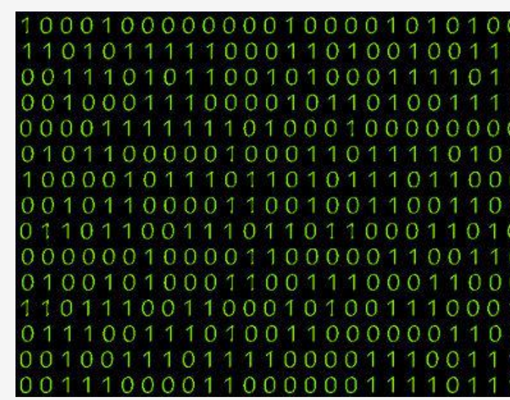


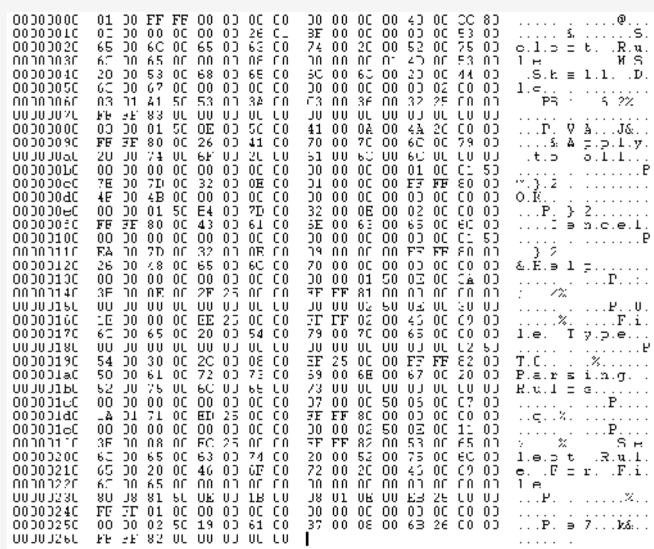
Machine code/instructions

Related to the physical structure of the computer.

- \+ poweful and fast
- − long programs
- − difficult to be written
- − difficult to be revised







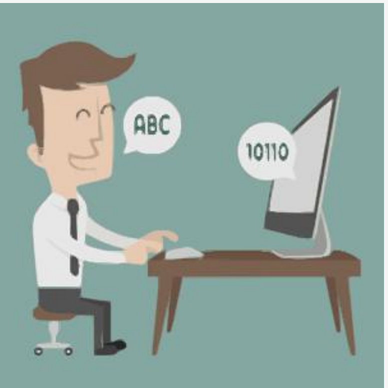
Natural (human) language

It is the language we use in all our communications, learned since our childhood.

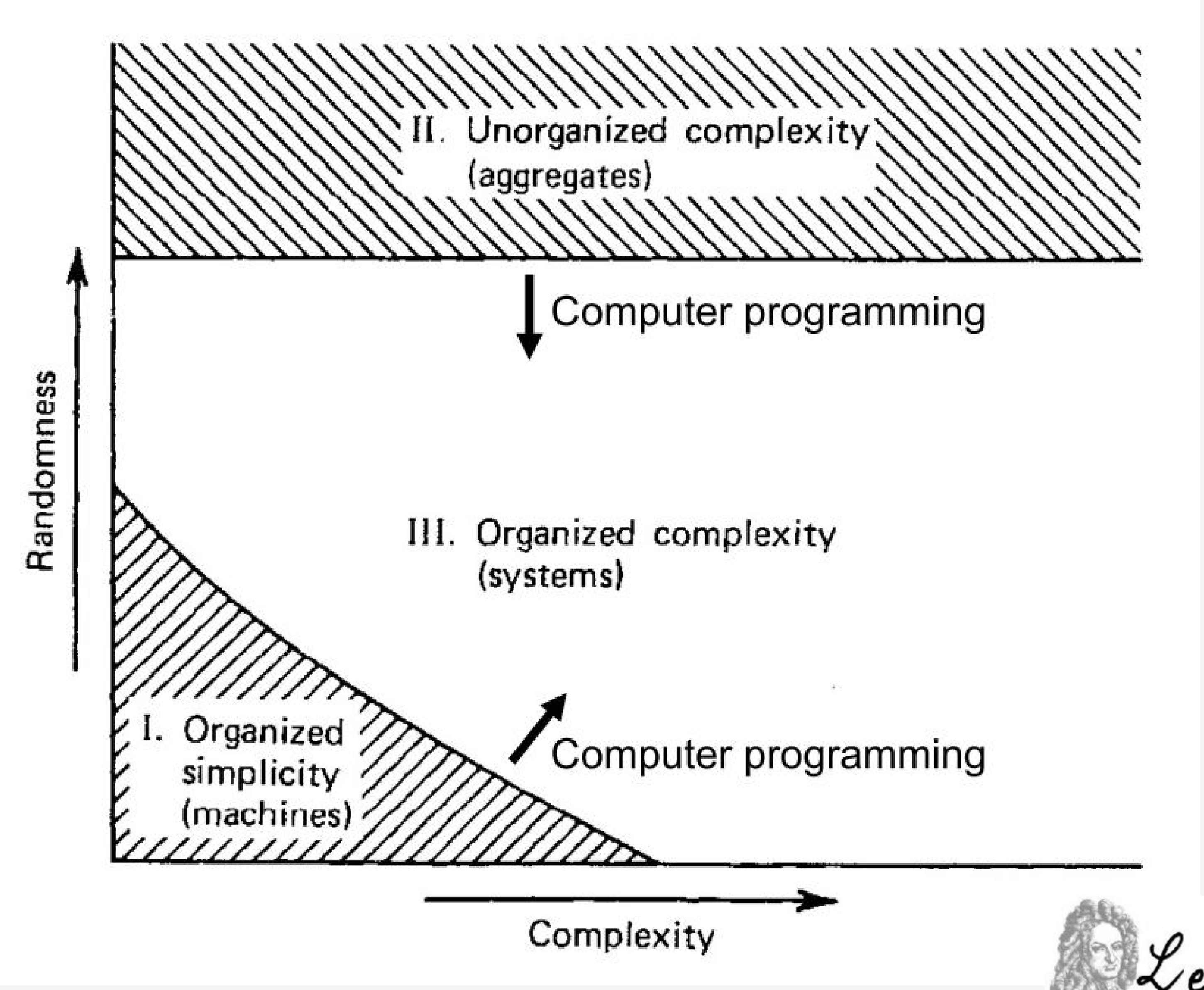
+ Expressively rich.
- Ambiguous, redundant.

Programming languages

A programming language is a language which is *intermediary* between machine code and natural language.



Programming languages

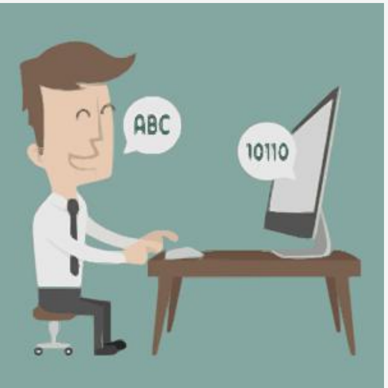


VanRoy 2009,
Weinberg 1977

Programming languages

A programming language is a language which is *intermediary* between machine language and natural language.

- But **what** we have to tell to the machine?



Computing

Computing is no more about computers than astronomy is about telescopes.

Edsger Dijkstra

Problem solving terms

- A *well-defined* problem is usually defined in terms of

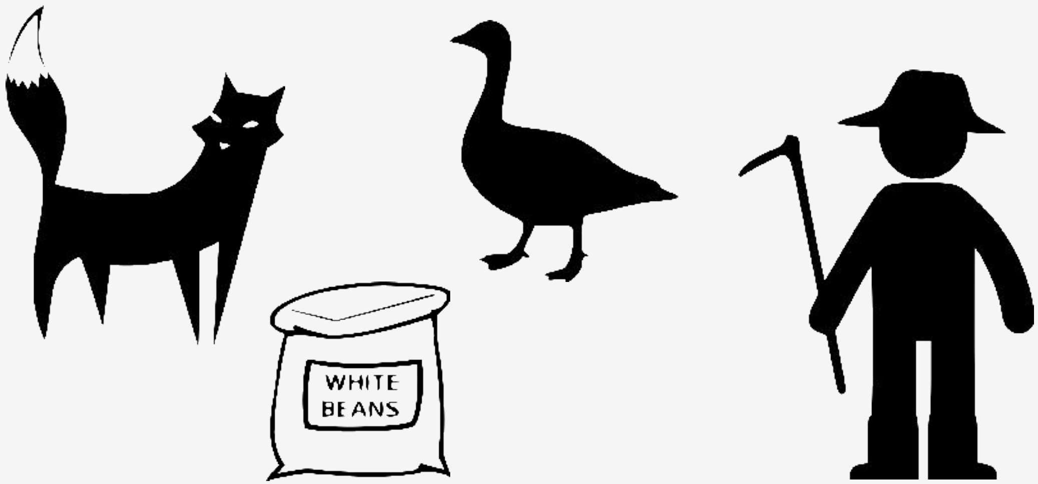
  - an **initial state** or *situation*
  - a **goal state**, i.e. a *desired outcome*,
  - certain **resources** (which put contraints on the possible paths towards the goal).

An ancient puzzle ~ 9th century

- Once upon a time a farmer went to market and purchased a **fox**, a **goose**, and a **bag of beans**. On his way home, the farmer came to the bank of a river. In crossing the river by boat, the farmer could carry only himself and a single one of his purchases – the fox, the goose, or the bag of the beans.

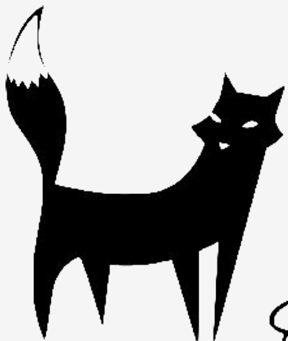
Leibniz Center for Law

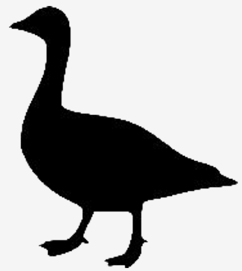
An ancient puzzle ~ 9th century

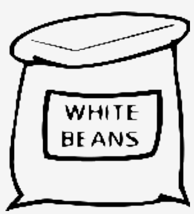


- Once upon a time a farmer went to market and purchased a **fox**, a **goose**, and a **bag of beans**. On his way home, the farmer came to the bank of a river. In crossing the river by boat, the farmer could carry only himself and a single one of his purchases – the fox, the goose, or the bag of the beans.

An ancient puzzle ~ 9th century



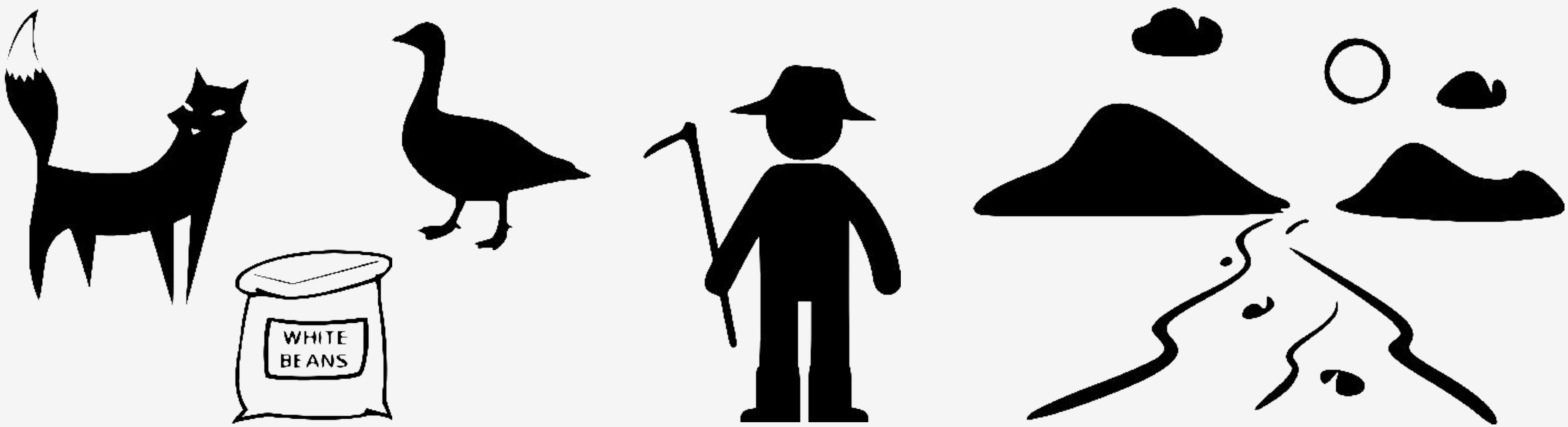




- Once upon a time a farmer went to market and purchased a **fox**, a **goose**, and a **bag of beans**. On his way home, the farmer came to the bank of a river. In crossing the river by boat, the farmer could carry only himself and a single one of his purchases – the fox, the goose, or the bag of the beans.

Leibniz Center for Law

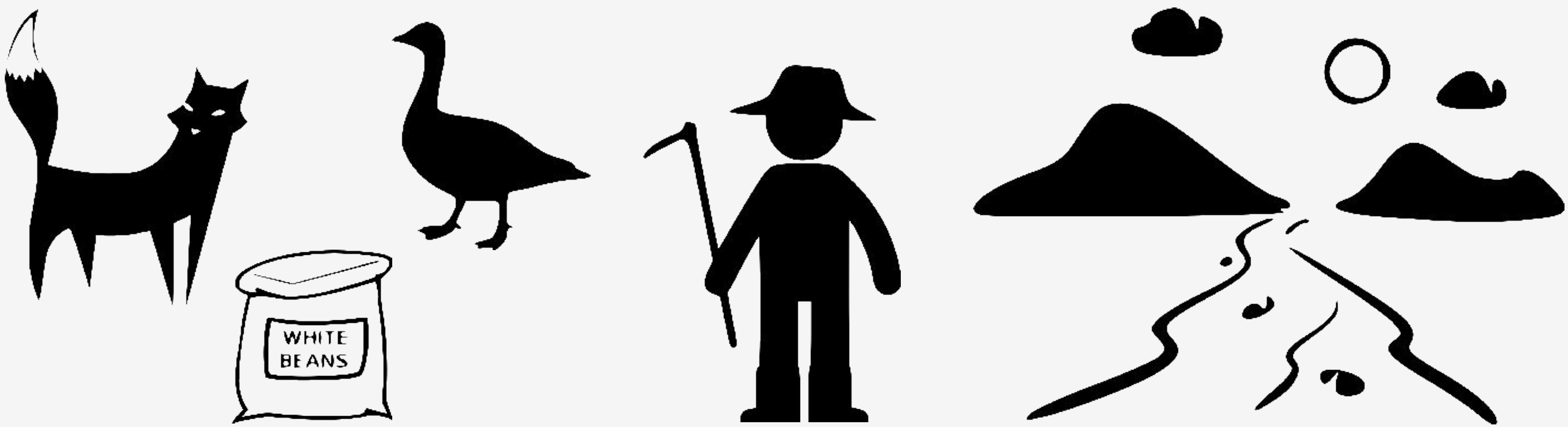
An ancient puzzle ~ 9th century



- If left together, the fox would eat the goose, or the goose would eat the beans.
- The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact. How did he do it?

Leibniz Center for Law

An ancient puzzle ~ 9th century



- What is the goal?
- What is the initial situation?
- Which are the resources/constraints?

Other problems

- How much is 2 * 2 + 4 ?
- Prepare a dish of spaghetti.
- Manage your collection of books.
- Given $f(a, b) = a^2 - b^2$, how much is f(2, 3)?
- Schedule your weekly physical exercises, considering your personal and professional appointments.
- Find the max of 1, 5, 2, 9, 4, 6, 3, 8, 7.
- Order the same sequence.
- Calculate the taxes you have to pay.

Leibniz Center for Law

From *problems* to *solvers*

Problem

↓ Analysis

Problem solving method

↓ Planning

Problem solving task

↓ Execution

Solution

Problem

↓ Analysis

Algorithm

↓ Programming

Program

↓ Execution

Outcome

From *problems* to *solvers*

Problem

↓ Analysis

Problem solving method

↓ Planning

Problem solving task

↓ Execution

Solution

Problem

↓ Analysis

Algorithm

↓ Programming

Program

↓ Execution

Outcome

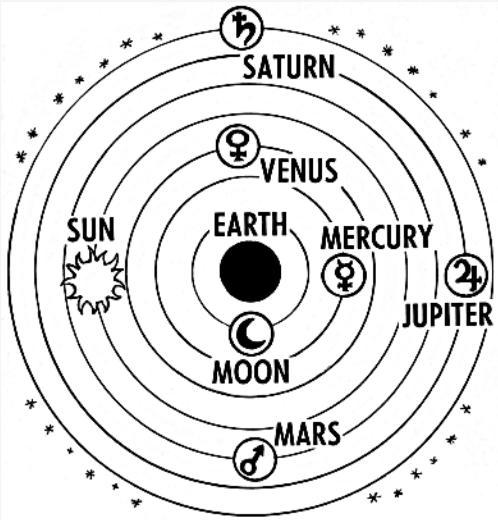
The "real" problem is not programming but finding the path toward the solution.

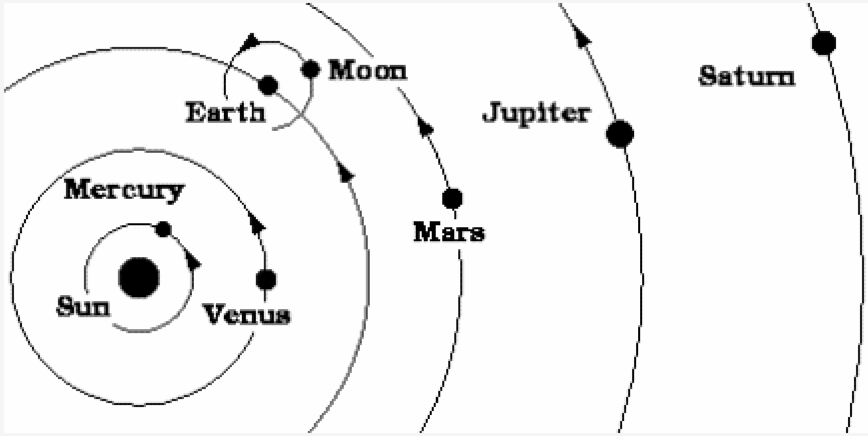
Leibniz Center for Law

Programming paradigms

Paradigms

- In general, a **paradigm** is a *theoretical framework* that guides and aggregates a number of theories, generalizations, and experiences performed within a certain discipline or school.





cf. Thomas Kuhn, The Structure of Scientific Revolutions (1970)

Leibniz Center for Law

Programming paradigms & co.

- Programming paradigm
  - a conceptual framework that serves as visualization, guideline of thoughts and practice for the programming of computers
- Programming technique
  - related to an algorithmic idea for solving a particular class of problems, e.g. *divide et impera* (*divide and conquer)* or *development by stepwise refinement*
- Programming style
  - the way we express ourselves in a computer program, usually related to elegance or lack of elegance

4 “axis” for paradigms

- Imperative vs Declarative
- Procedural vs Object-Oriented
- Sequential vs Concurrent (vs Parallel)
- Static vs Dynamic

Imperative vs Declarative

Imperative vs Declarative

- Imperative:
  - programming focusing on the **sequence of operations** necessary to solve the problem (which in turn usually stays implicit)
- Declarative
  - programming focusing on describing the **problem** (while the sequence of operations to be performed is left implicit)

Imperative programming

Imperative programming

Focus: *how to compute*

Based on instructions, correspondent to actions **commanded** to the machine.

- It assumes that the computer can maintain the changes (the *side-effects*) caused by the computation process.

(states etc...

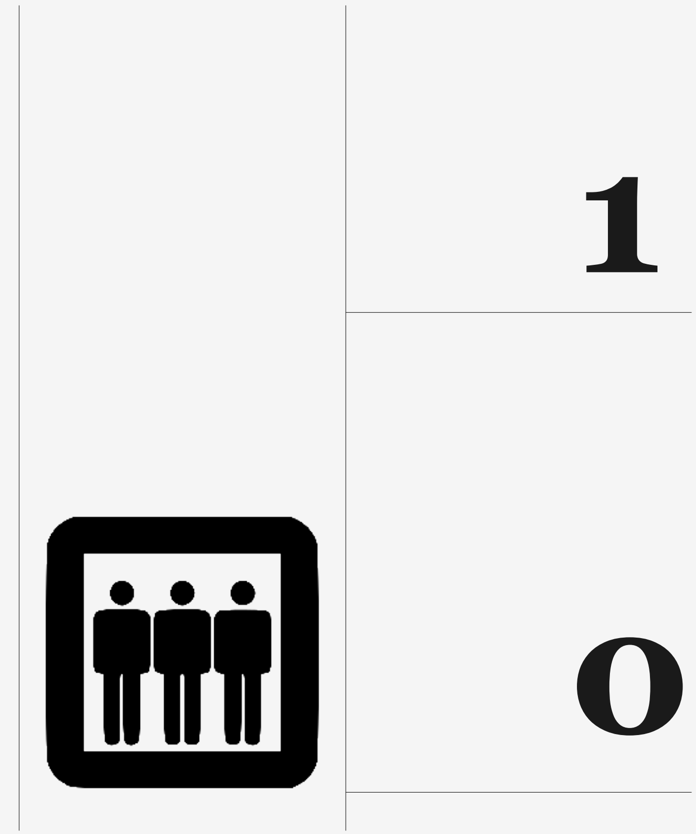
States, Constants and Variables

- State is the ability to maintain *information*, or better, to store value which may change in time.
- A state can be named/labelled. If the referred state does not change, the entity is called **constant**, otherwise it is called **variable**.

  → *an object/process with state is*
  an object/process with memory

States and Transitions

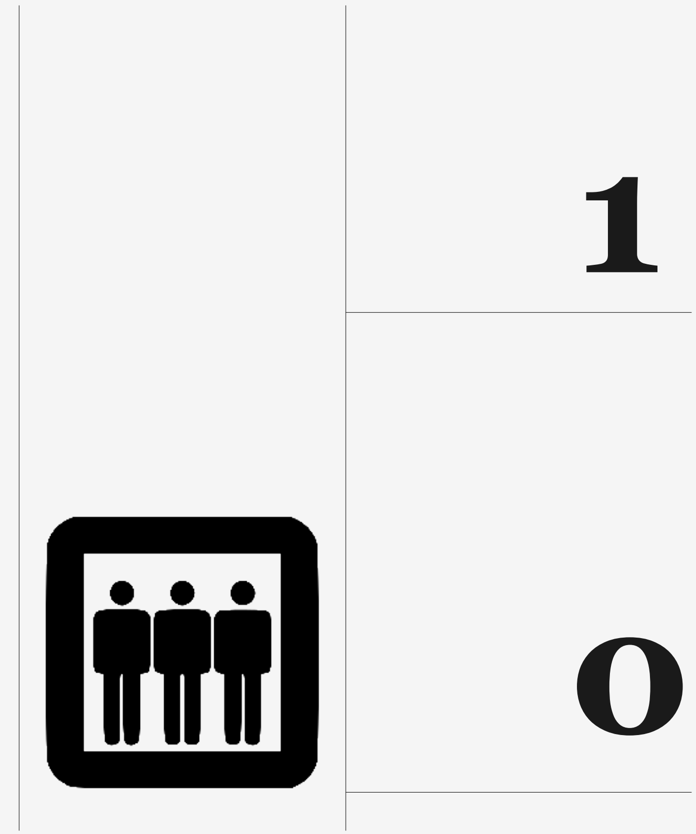
- The life of any object can be described in terms of its **state space** by formulating the possible locations of the object.



Center for Law

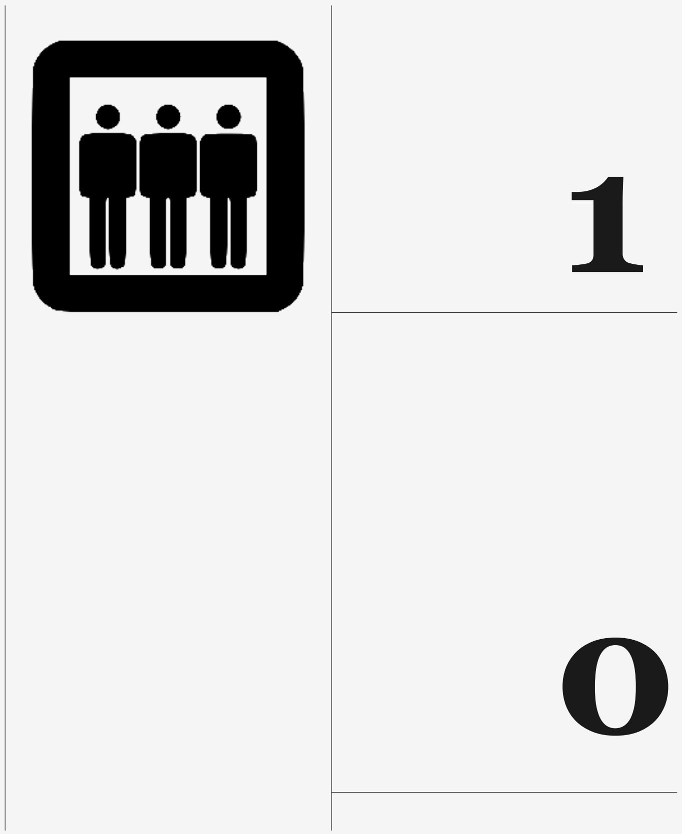
States and Transitions

- The life of any object can be described in terms of its **state space** by formulating the possible locations of the object.
- Ex. 0



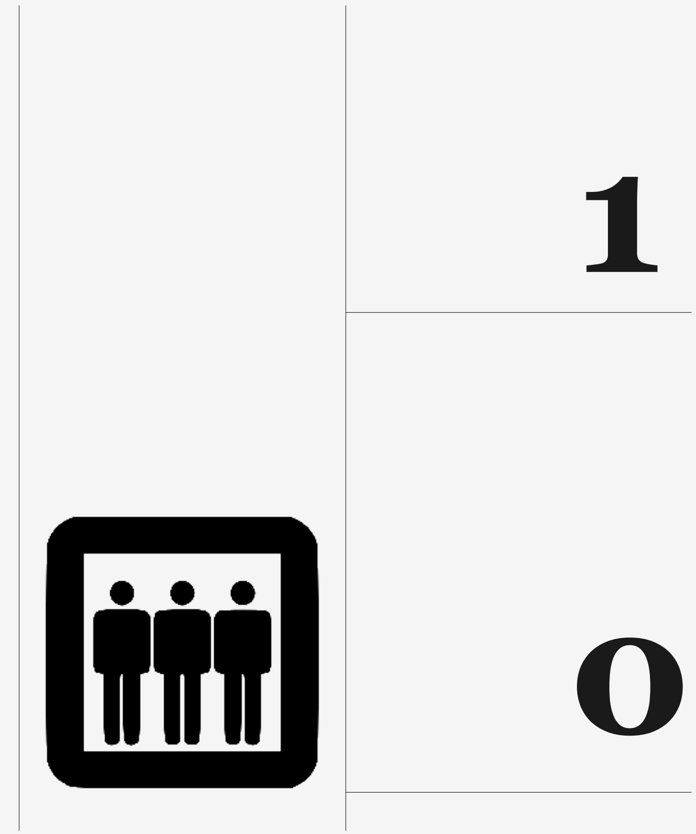
States and Transitions

- The life of any object can be described in terms of its **state space** by formulating the possible locations of the object.
- Ex. 0, 1



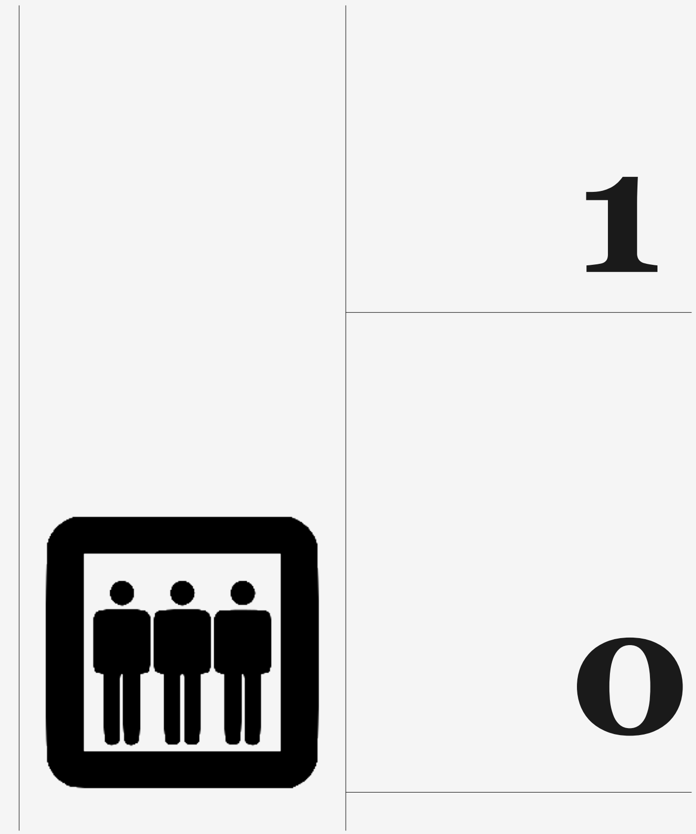
States and Transitions

- The life of any object can be described in terms of its **state space** by formulating the possible locations of the object.
- Ex. 0, 1, 0



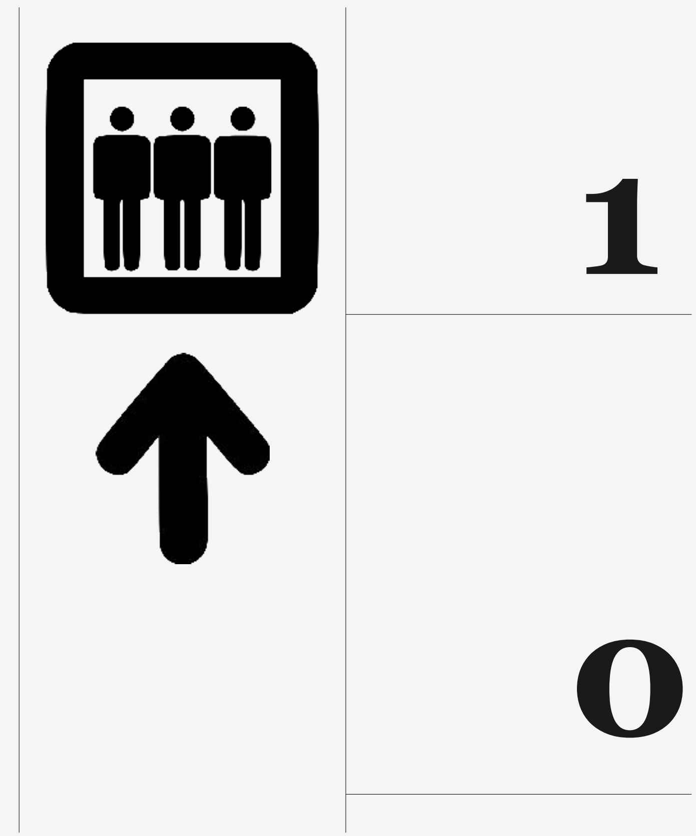
States and Transitions

- The life of any object can be described in terms of its **state space** by formulating the possible locations of the object.
- The dual view is the **transition space** which formulates the distinct events which may change its location.
- Ex.



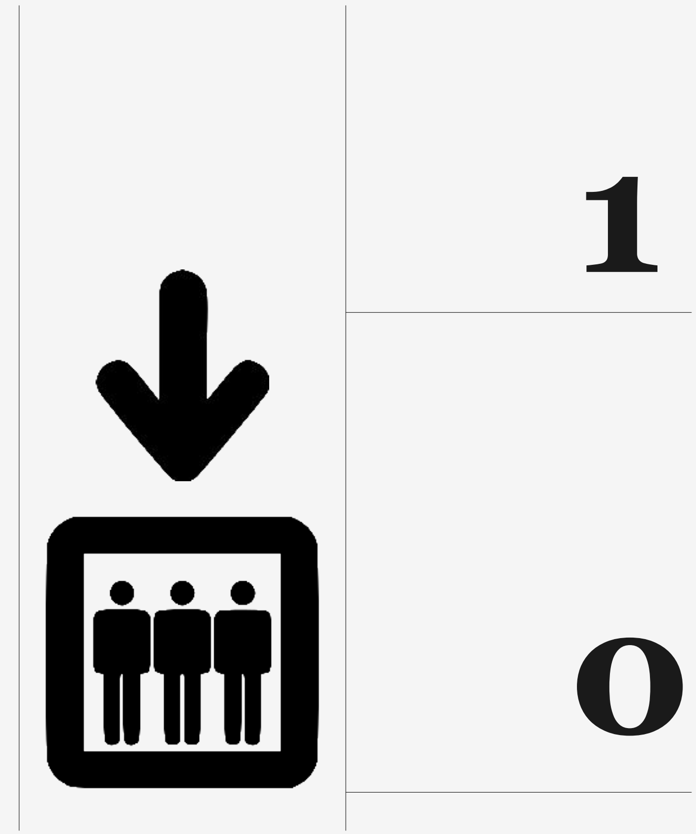
States and Transitions

- The life of any object can be described in terms of its **state space** by formulating the possible locations of the object.
- The dual view is the **transition space** which formulates the distinct events which may change its location.
- Ex. ↑



States and Transitions

- The life of any object can be described in terms of its **state space** by formulating the possible locations of the object.
- The dual view is the **transition space** which formulates the distinct events which may change its location.
- Ex. ↑, ↓



State spaces and data types

- The elementary data components in the computer are bits (or *boolean* values), but we usually program referring to other **data types** as well.

State spaces and data types

- **Types** serve to describe what kind of data is stored.
- Even though all data is represented using 0 and 1, it is not **interpreted** in the same way in the program.

State spaces and data types

- **Types** serve to describe what kind of data is stored.
- Even though all data is represented using 0 and 1, it is not **interpreted** in the same way in the program.
- Usually records **occupy** different amounts of **memory** according to their type:
  - Boolean ~ 1 bit (virtually) → 2 symbols available
  - Char ~ 1 byte
  - Int ~ 2 byte

State spaces and data types

- **Types** serve to describe what kind of data is stored.
- Even though all data is represented using 0 and 1, it is not **interpreted** in the same way in the program.
- Usually records **occupy** different amounts of **memory** according to their type:
  - Boolean ~ 1 bit (virtually) → 2 symbols available
  - Char ~ 1 byte = 8 bit → 2^8 = 256 symbols
  - Int ~ 2 byte = 16 bit → 2^{16} = 65536 symbols

State spaces and data types

- Common data types:
  - Boolean — true, false
  - Char — 'a', 'b', 'z', '3', 'A'
  - Integer — 3, 525, –2643
  - Float — 0.253, 655.34
- and *compositions* of those:
  - Array, List, Map, ..

    e.g. String as composition of Chars

...states etc.)

Imperative programming

Focus: *how to compute*

- Most popular programming languages implement the imperative paradigm:
  - it most closely resembles the actual machine itself, so the programmer thinks in a much closer way to the machine;
  - because of such closeness, it was until recently the only one efficient enough for widespread use.

Imperative programming

- Advantages
  - efficient as close to the machine
  - popular
  - familiar
- Disadvantages
  - a program can be complex to understand, because the *referential transparency* does not hold (due to **side effects**)
  - *abstraction* is more limited
  - **order** is crucial, which is not suited in certain problems

Leibniz Center for Law

Control flow

- The **control flow** basically describes the sequential order in which instructions are evaluated.
- The most common control flow *operators* are:
  - jumps or unconditional branchs (GO TO)
  - conditional branchs and loops (IF .. THEN, WHILE .. DO)
  - subroutine/procedure/function calls, used to pass the computation to external modules and then to continue from the outcome (CALL ... PARAMS ...)

subroutines/procedures.. closures!

A closure is a "packet of work", defined together with certain input parameters.

- Any group of instructions in a program can be transformed into a closure with certain inputs.
- The program can call such *callable units* and the result of their execution is the same as if the instructions were put at the place of the call.
- The use of subroutine however imposes some computational overhead in the call mechanism (the *call stack*).

subroutines/procedures.. closures!

- Disadvantage
  - The use of subroutine imposes some overhead in the call mechanism (the *call stack*).
- Advantages
  - Decomposition
  - Reducing duplicate code within the program
  - Enabling reuse of code across multiple programs
  - Separation of concerns
  - Hiding implementation
  - Improving traceability (debugging)

Divide et impera (divide and conquer)

- Decomposition allows to take a strategic algorithmic approach
- Rather than facing the complete problem, we tackle it down to smaller (and simpler) independent components.

  → Different teams may work on different sub-problems.

Leibniz Center for Law

Declarative programming

Declarative programming

Focus: *what to compute (as desired outcome)*

- It is not concerned about how to do things, but what should be obtained.
  - Languages: domain specific (e.g. HTML), query (SQL), logic (Prolog).

Declarative/Logic programming

Focus: *what to compute (as desired outcome)*

- Various logical assertions about a situation are made, describing all known **facts** and **rules** about the modeled world. Then **queries** are made.
- The role of the computer is to *maintain data* and to perform *logical deduction*.

Algorithm = Logic + Control

"An algorithm can be regarded as consisting of

- a **logic component**, which specifies the *knowledge* to be used in solving problems, and
- a **control component**, which determines the *problem-solving strategies* by means of which that knowledge is used.

The logic component determines the meaning of the algorithm whereas the control component only affects its efficency."

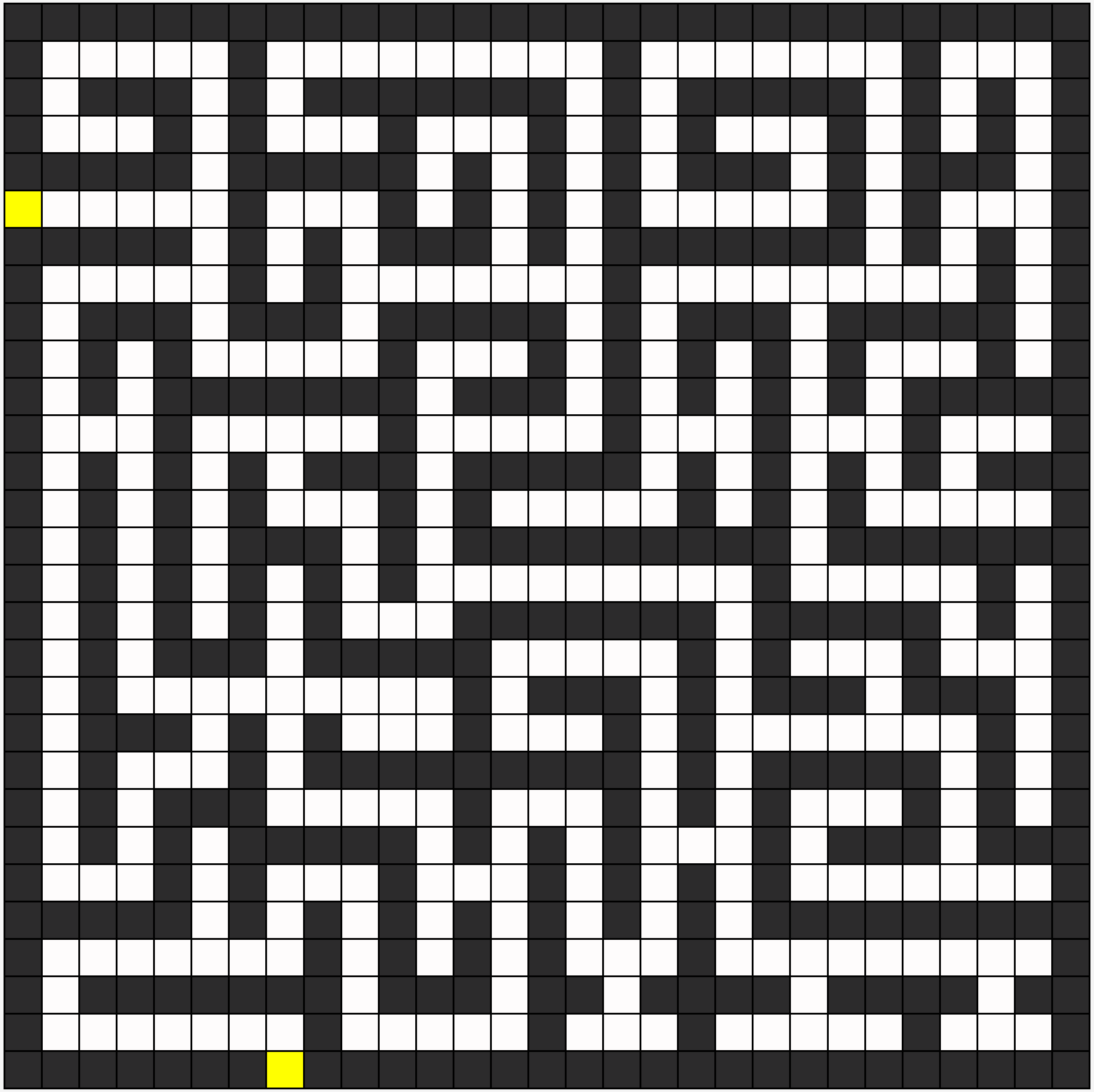
Robert Kowalsky, Algorithm = Logic + Control (1979)

Leibniz Center for Law

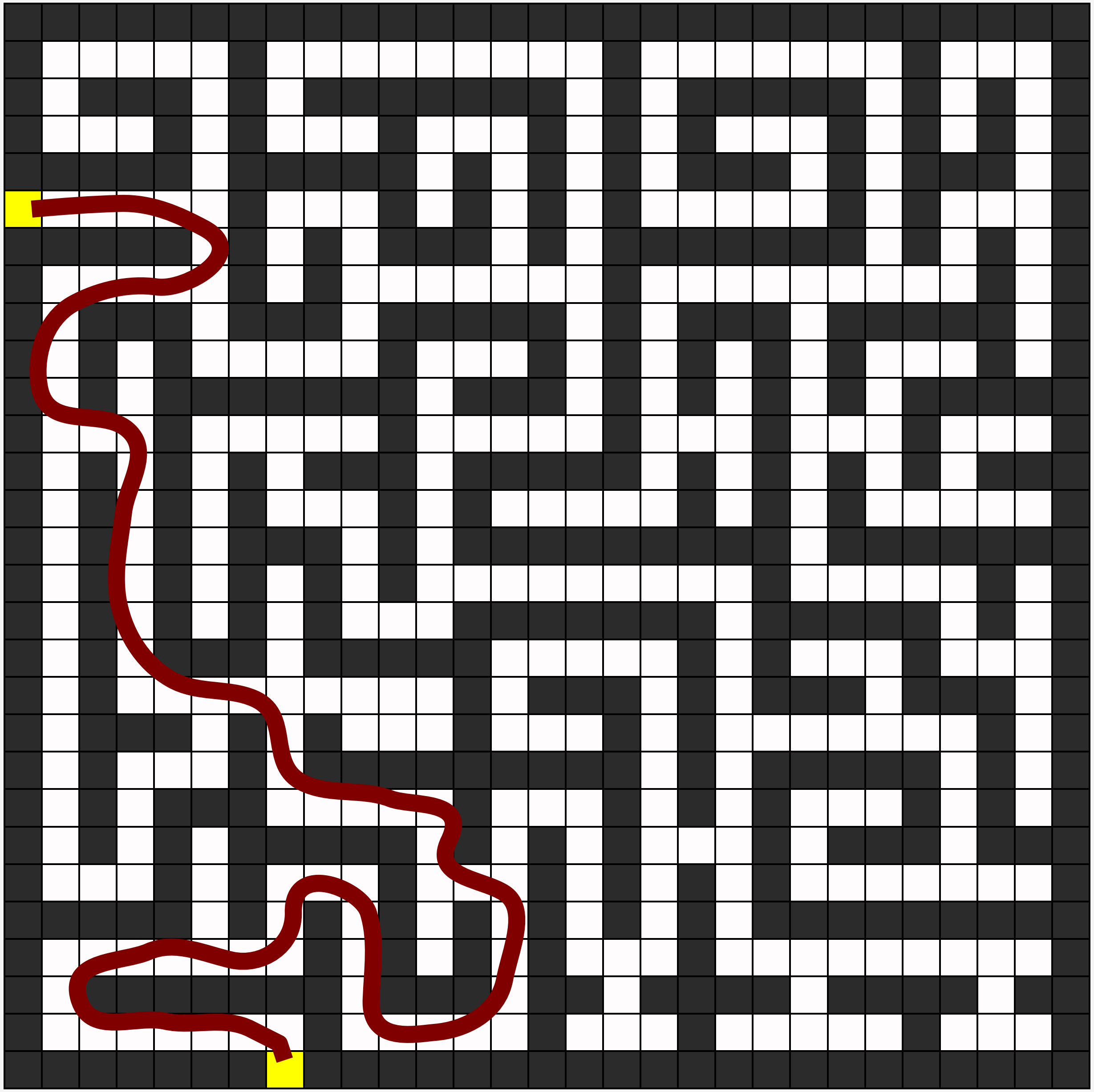
Imperative vs Declarative

- Imperative:
  - **inside-to-outside** approach: all execution alternatives are explicitly specified and new alternatives must be explicitly added
- Declarative
  - **outside-to-inside** approach: constraints implicitly specify execution alternatives as all alternatives that satisfy the constraints; adding new constraints usually means discarding some execution alternatives

Leibniz Center for Law



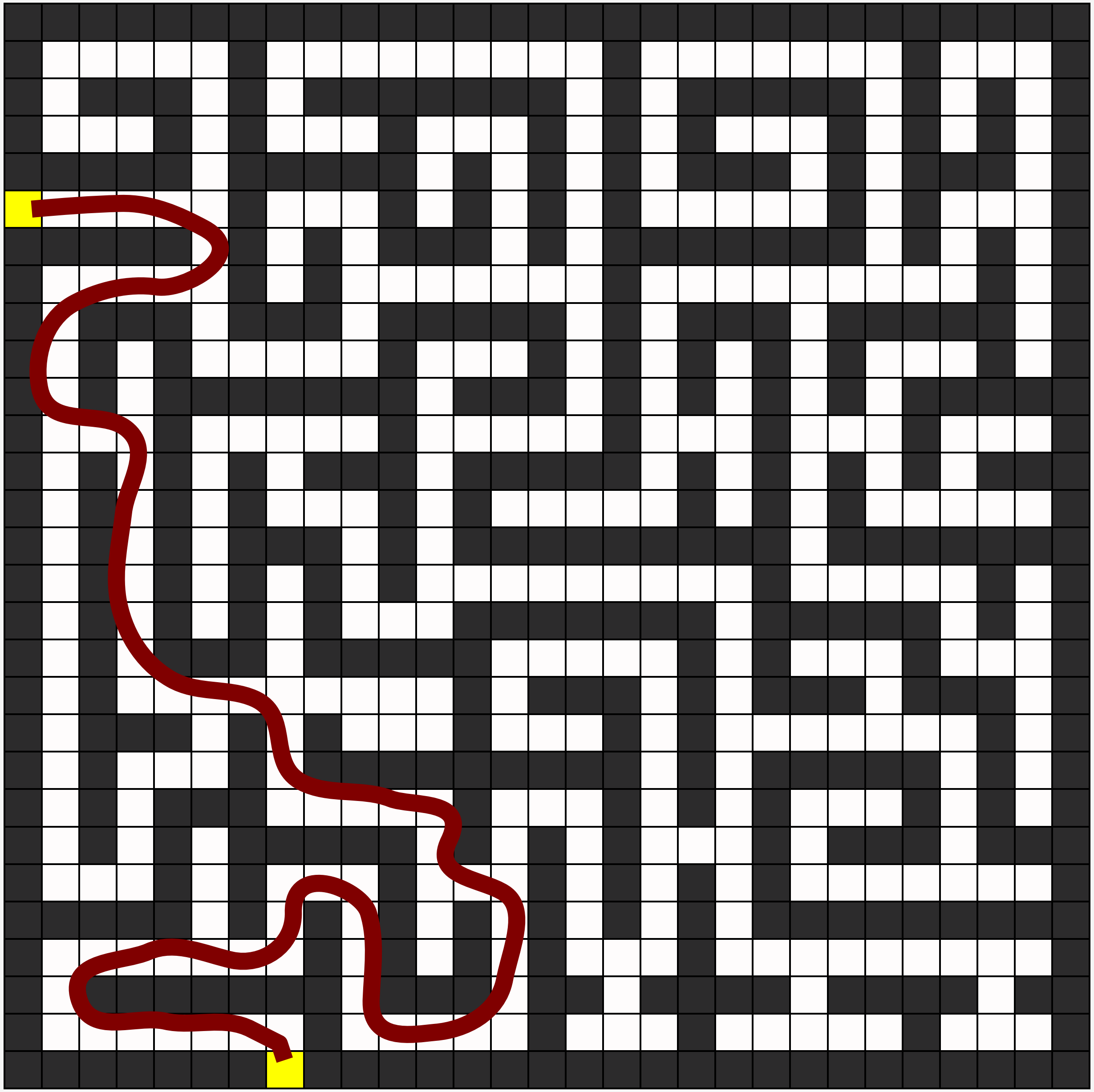
Imperative:
you command
the directions



Imperative:

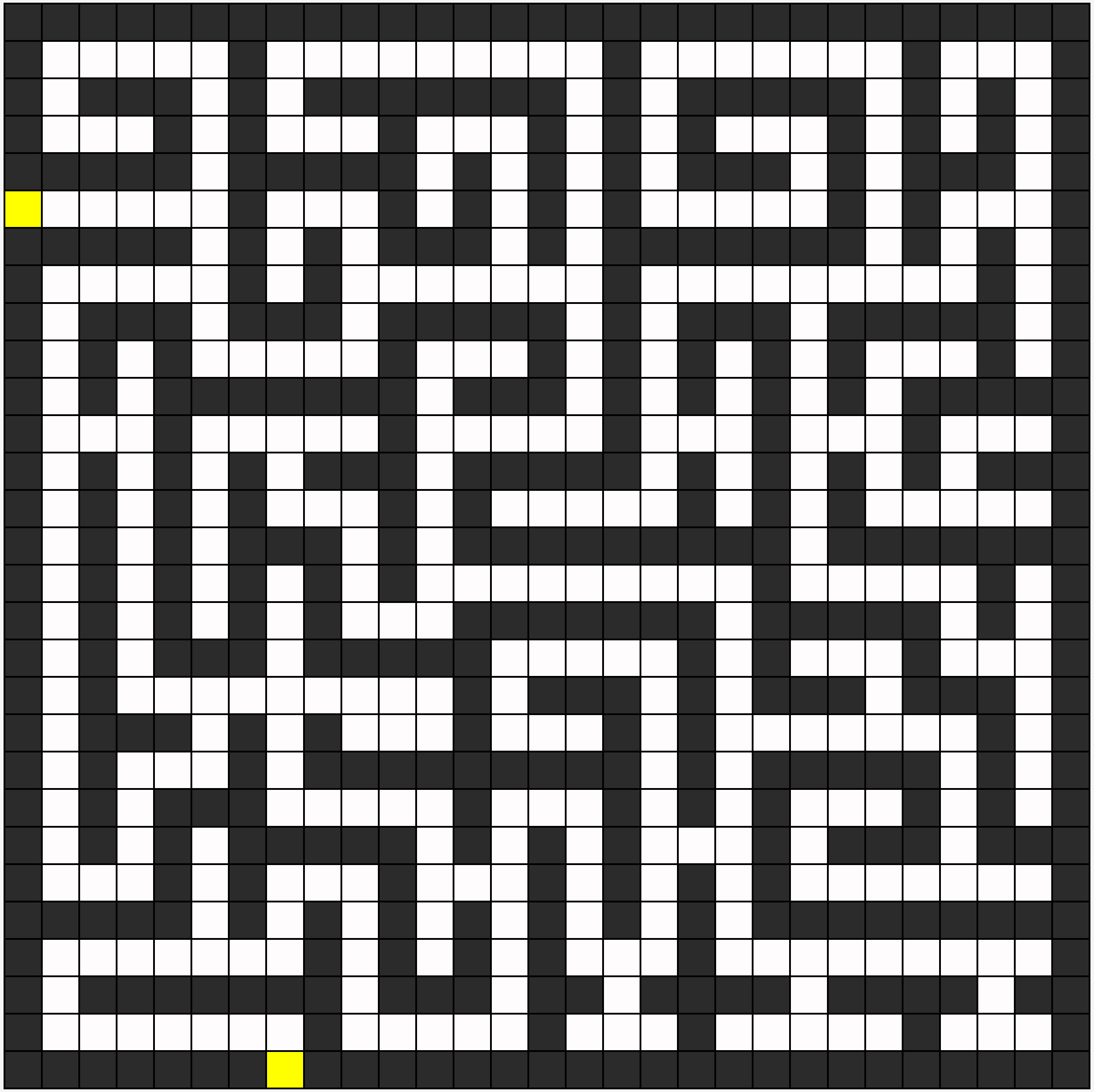
you command

the directions

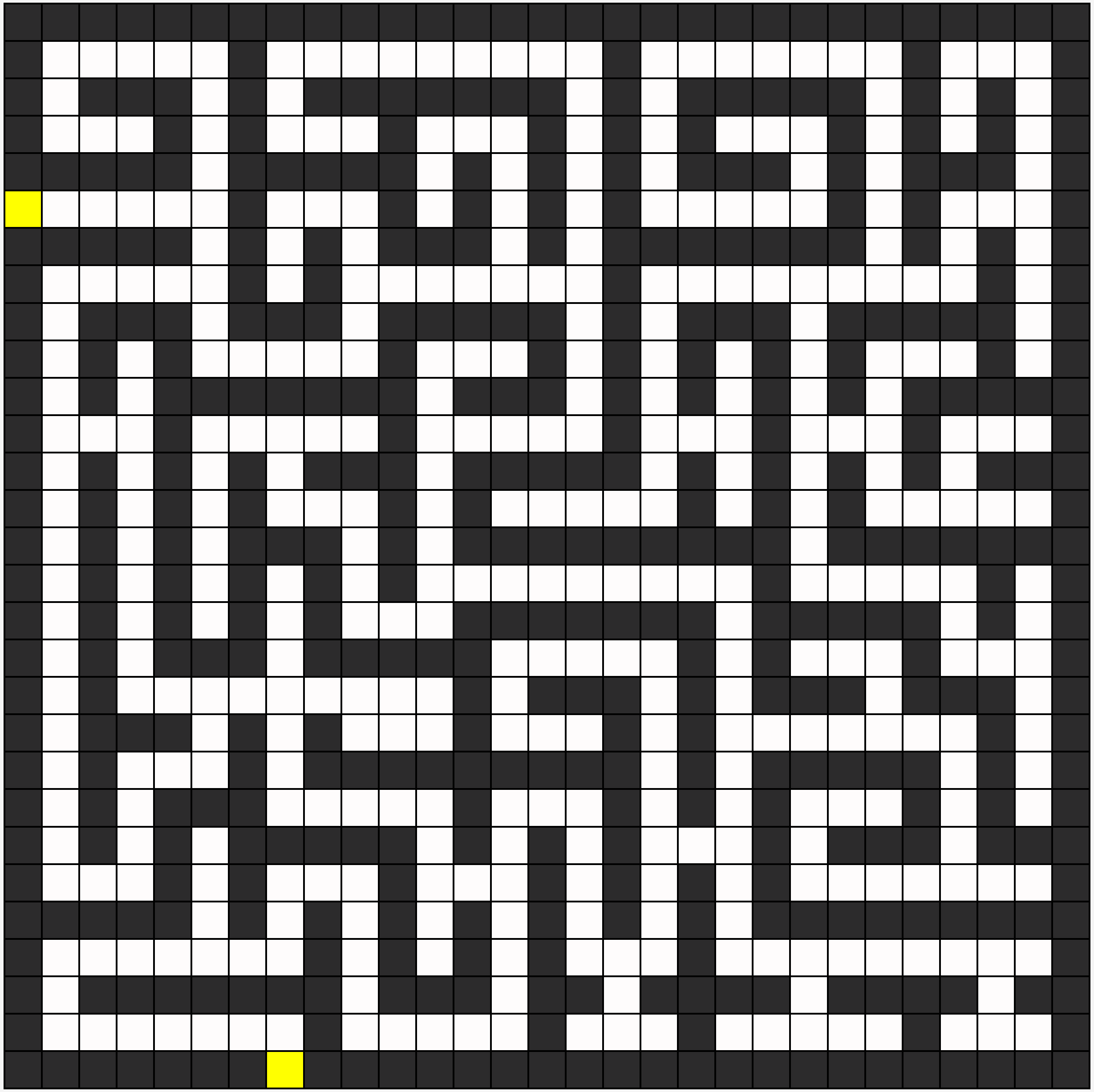


Imperative: you command the directions

- What if the labyinth changes?

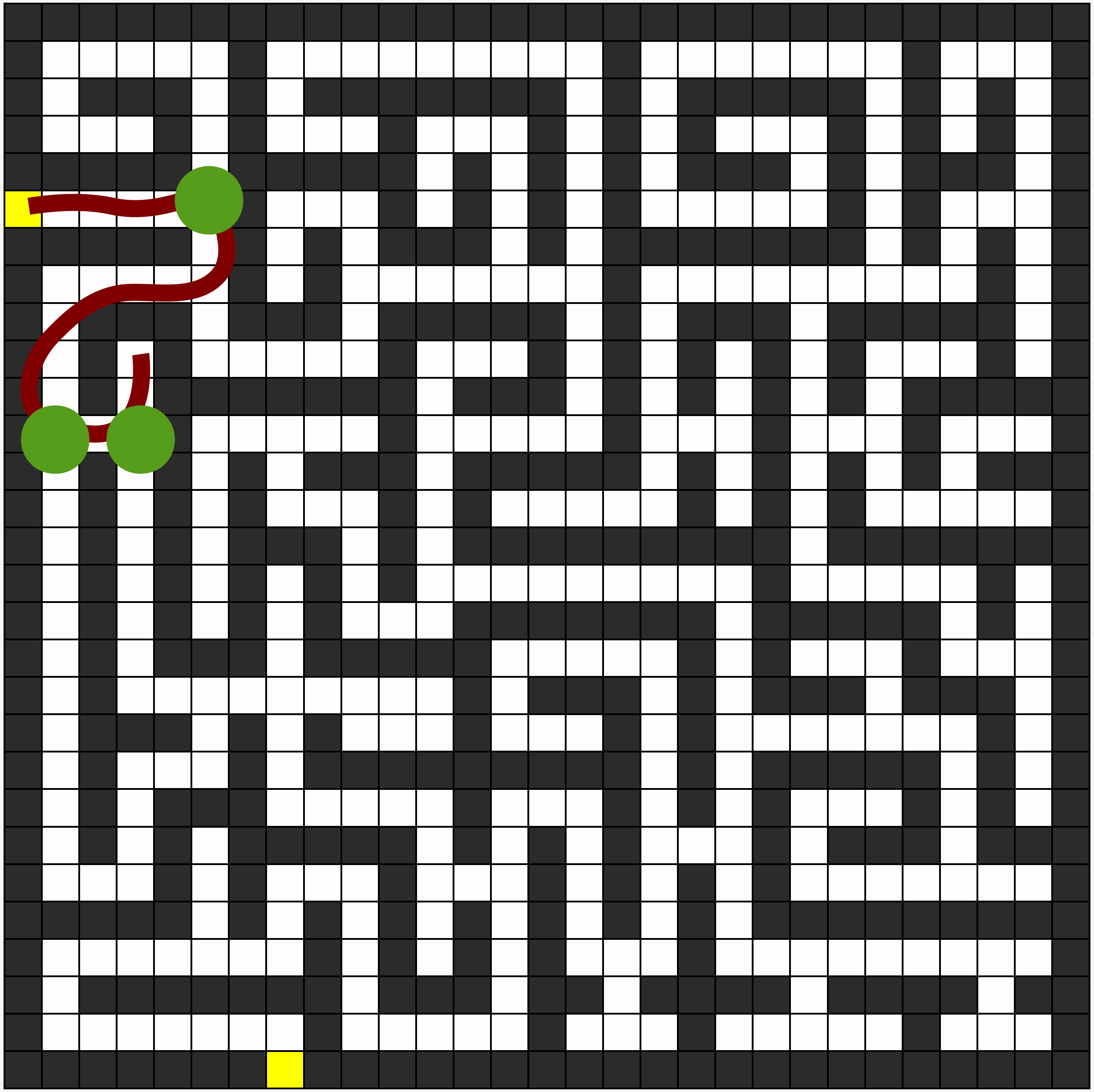


Declarative:
you give just the labyrinth. The computer finds the way.



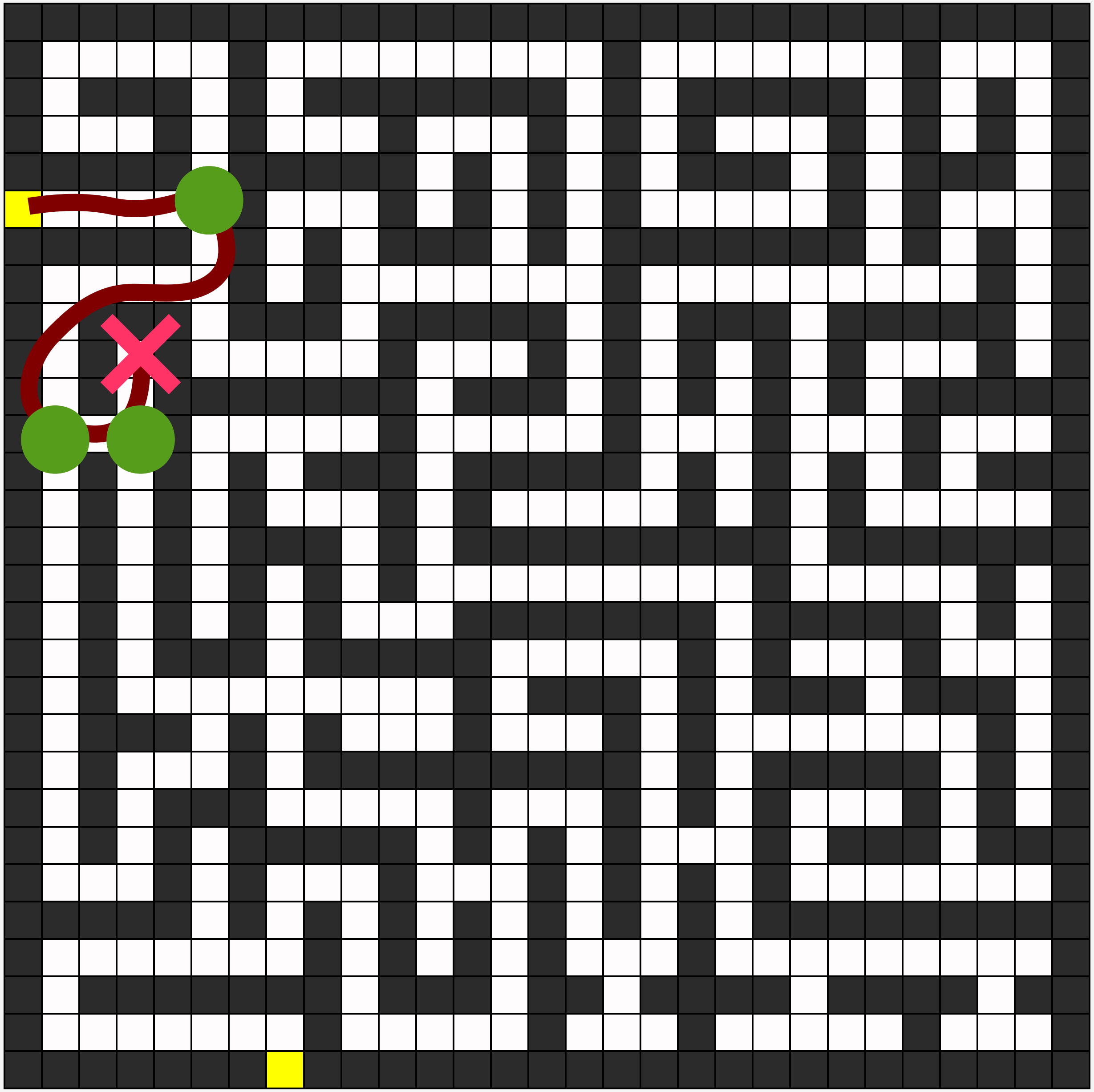
Declarative:
you give just the labyrinth. The computer finds the way.

- For instance, via *trial, error* and **backtracking**



Declarative:
you give just the labyrinth. The computer finds the way.

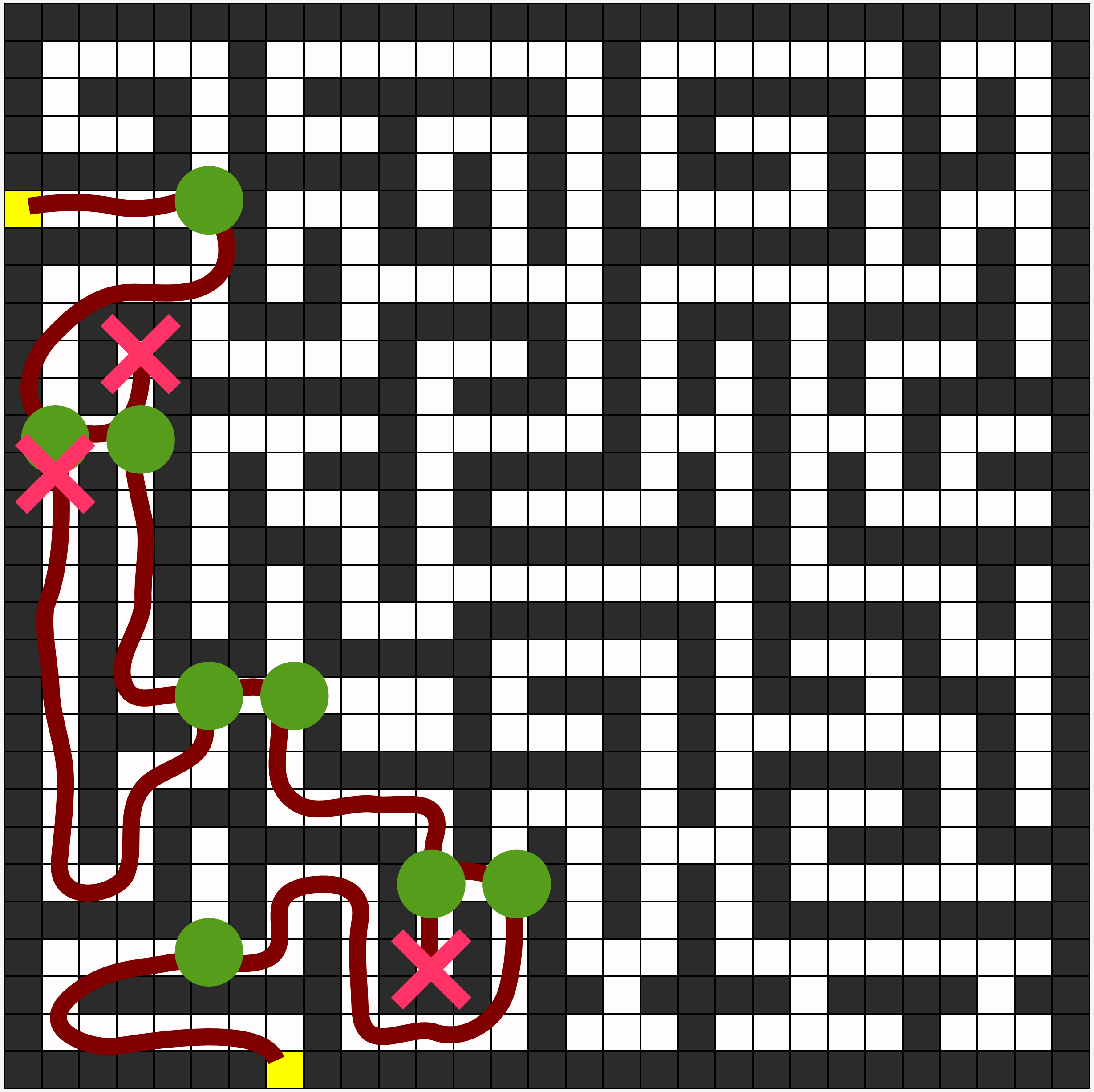
- For instance, via *trial, error* and **backtracking**



Declarative:
you give just the labyrinth. The computer finds the way.

- For instance, via *trial*, *error* and **backtracking**

Leibniz Center for Law



Declarative:
you give just the labyrinth. The computer finds the way.

- For instance, via *trial, error* and **backtracking**

Functional programming

Focus: *computation as mathematical function*

- While functional languages typically do appear to specify *how*, a compiler for a purely functional programming language is free to rewrite the operational behavior of a function, so long as the same result (the *what*) is returned for the same inputs.

Ex. $(a^2 - b^2)$ can be rewritten as $(a - b) * (a + b)$

Center for Law

Functional programming

Focus: *computation as mathematical function*

- While functional languages typically do appear to specify *how*, a compiler for a purely functional programming language is free to rewrite the operational behavior of a function, so long as the same result (the *what*) is returned for the same inputs.

  Ex. $(a^2 - b^2)$ can be rewritten as $(a - b) * (a + b)$

- As declarative programming, it should be transparent in respect to *side-effects*.

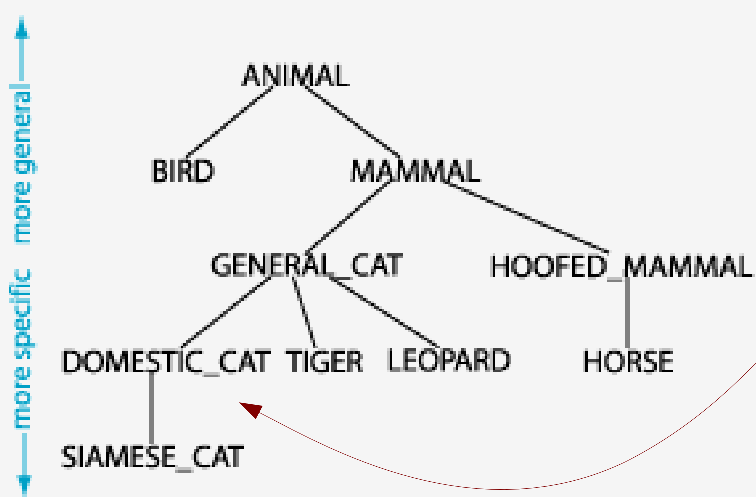
Procedural vs Object-oriented

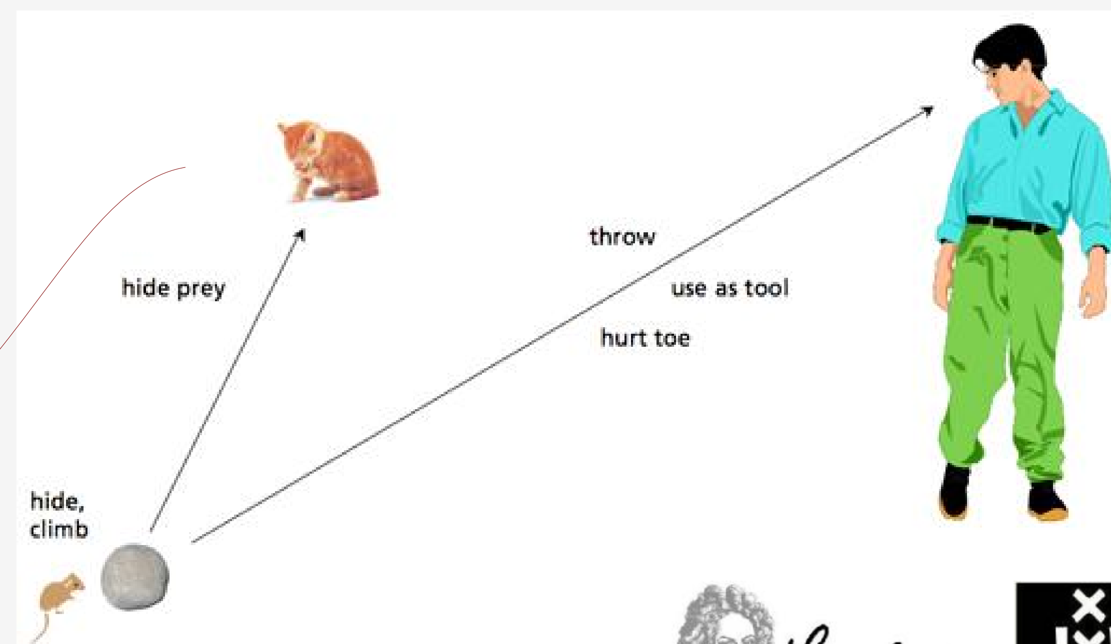
Procedural vs Object-oriented

- Procedural
  - programming focused on *procedures*: blocks of instructions/portions of code related to specific tasks
- Object-oriented
  - programming focused on the (data) objects which are manipulated during the computation

Data types and objects

- The idea of objects grows from associating to data structures the description of possible actions to be performed with, and the description of conceptual relations with other objects (**encapsulation**).





Leibniz Center for Law

Object-oriented programming

Focus: *the object of computation*

- (real–world) objects are modeled as seperate entities having their own **state** (i.e. internal variables or ***attributes***),
- which is modified only by built–in **procedures**, called ***methods***.

→ *what I can do on a object is intrinsic to the object!*

Object-oriented programming

Focus: *the object of computation*

- Objects are organized into **classes**, from which they *inherit* methods and internal variables.
- Objects can integrate other objects, thus enabling *composition*.
- The object-oriented paradigm provides key benefits of *reusable code* and *code extensibility*.

Agent-based programming

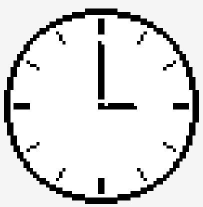
Focus: *computing entities described as concurrent, possibly intentional entities*.

- Ideally, it completes the progression of an imaginary evolution starting from the *instructions*, passing by *objects* and arriving up to *agents*.

Center for Law

physical stance

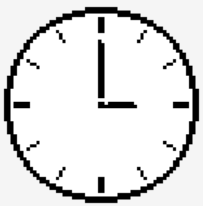
interpreting using
the physical laws



design stance

physical stance

interpretation related to what the entity is supposed to do (i.e. has been designed to do)



sometimes it breaks!

design stance

physical stance

interpretation related to what the entity is supposed to do (i.e. has been designed to do) intentional stance

design stance

physical stance

interpreting an entity as an ***agent***, ascribing him **beliefs, desires, intents** and *enough rationality* to do what he *ought to do* given those beliefs and desires

cf. Daniel Dennett, The Intentional Stance (1987)

intentional stance

design stance

physical stance

interpreting an entity as an ***agent***, ascribing him **beliefs, desires, intents** and *enough rationality* to do what he *ought to do* given those beliefs and desires

cf. Daniel Dennett, The Intentional Stance (1987)

Leibniz Center for Law

intentional stance

design stance

physical stance

interpreting an entity as an ***agent***, ascribing him **beliefs, desires, intents** and *enough rationality* to do what he *ought to do* given those beliefs and desires

cf. Daniel Dennett, The Intentional Stance (1987)

intentional stance

design stance

physical stance

interpreting an entity as an ***agent***, ascribing him **beliefs, desires, intents** and *enough rationality* to do what he *ought to do* given those beliefs and desires

cf. Daniel Dennett, The Intentional Stance (1987)

The intentional stance can be used as reference for the creation of a *user interface*.

Center for Law

The intentional stance can be used as reference for the creation of a *user interface*.

But also as an internal application architecture, via agent-based programming!

The entity is defined by desires and knowledge, and generates intents and performs actions accordingly.

Leibniz Center for Law

Sequential vs Concurrent + (Parallel)

sequential concurrent

behaviour

(processes)

sequential

concurrent

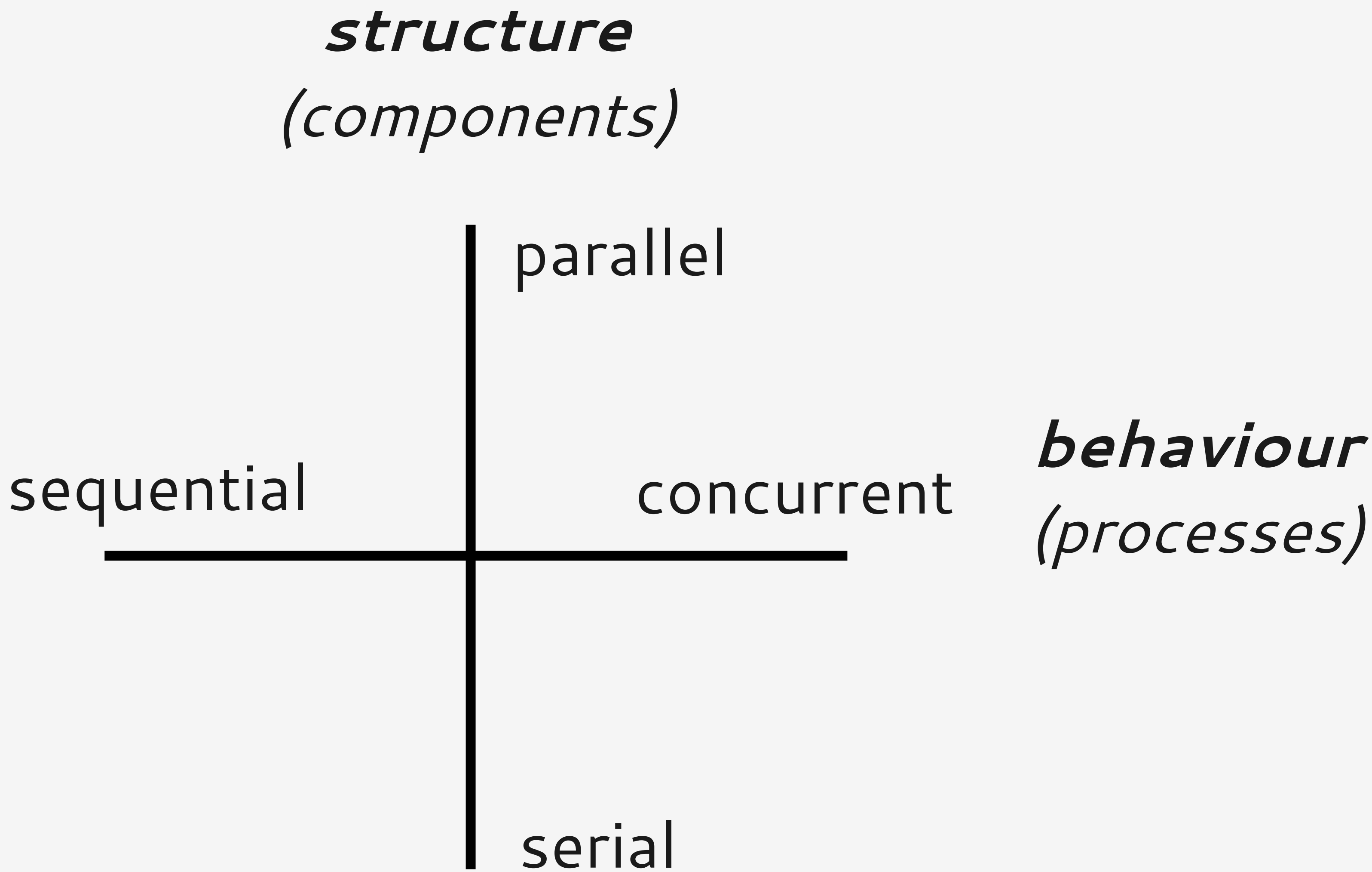
behaviour

(processes)

sequential

concurrent

behaviour
(processes)



structure
(components)
parallel
sequential
concurrent
serial
behaviour
(processes)

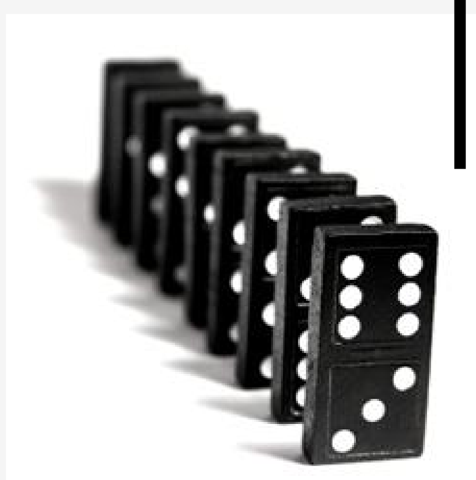
structure
(components)

parallel

sequential

concurrent

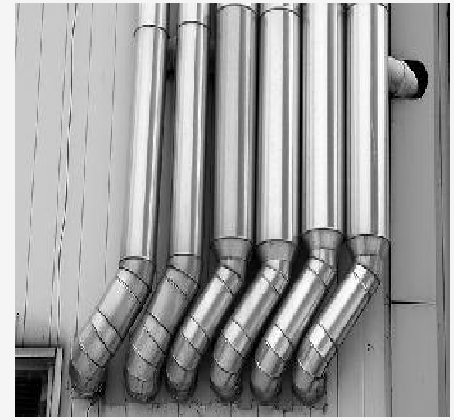
behaviour
(processes)



serial

Leibniz Center for Law

structure
(components)

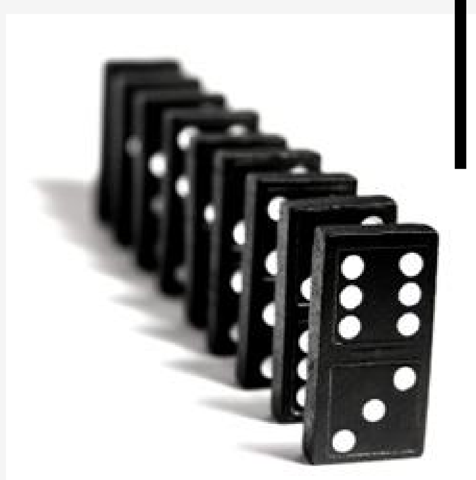


parallel

sequential

concurrent

behaviour
(processes)

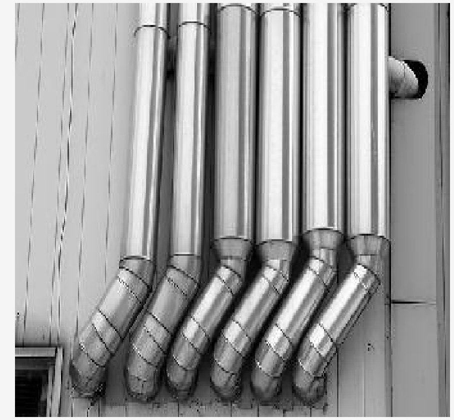


serial

Leibniz Center for Law

structure

(components)



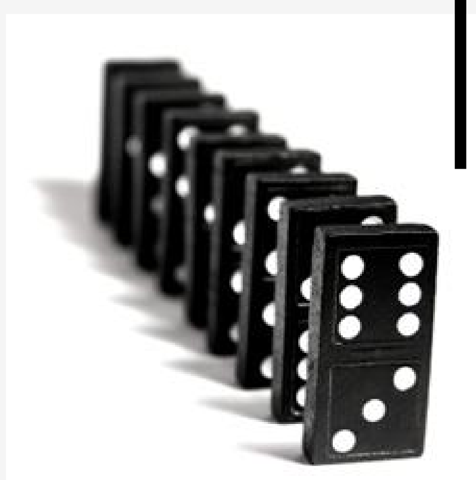
parallel

sequential

concurrent

behaviour

(processes)



serial

Leibniz Center for Law

Sequential, Concurrent, Parallel

- Sequential
  - instructions are executed step by step
- Concurrent
  - execution occurs concurrently, and if it has side-effect over the same components (*race* condition), it produces **non-determinism**
- Parallel
  - determinism is guaranteed if concurrency occurs in separate components (e.g. multicore processors)

Center for Law

Sequential vs Concurrent

- Sequential is the traditional, easier approach.
- It’s fundamentally more difficult to express algorithms in a concurrent/parallel way, but there is the potential of a great increase of performance, exploiting parallel architectures.

Data-flow programming

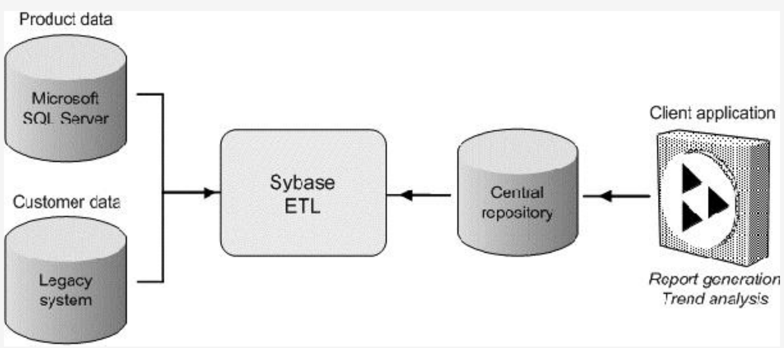
Focus: *how to connect computing devices*

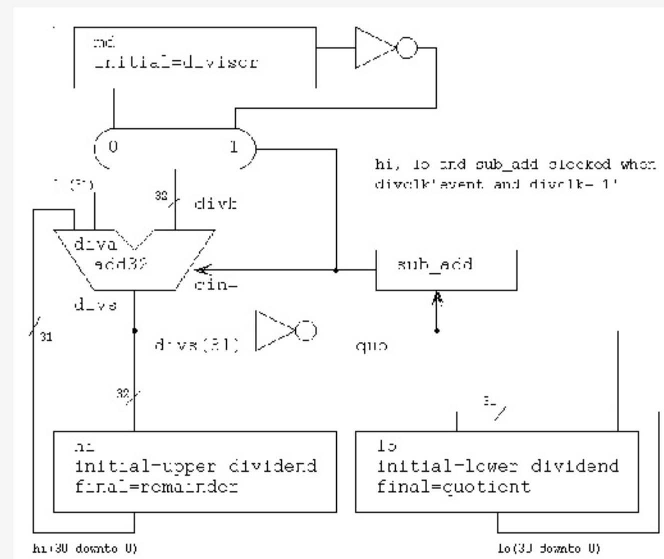
- Rather than how and what to compute, we are concerned about what kind and how data is passed between entities.

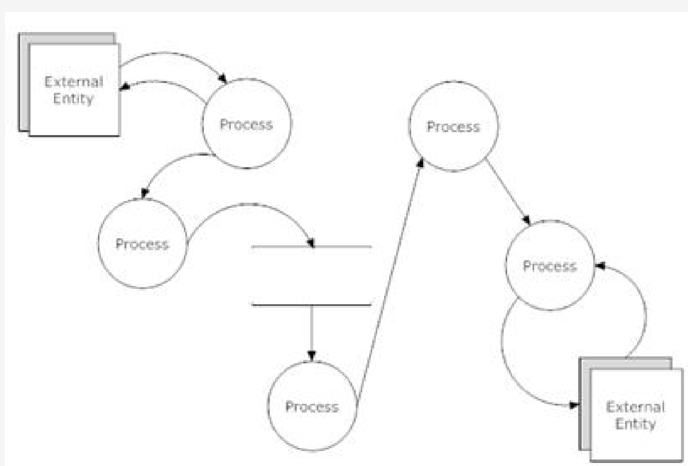
Data-flow programming

Focus: *how to connect computing devices*

- Typical use: electronics, infrastructure design, networks and business modeling, and *spreadsheets*!

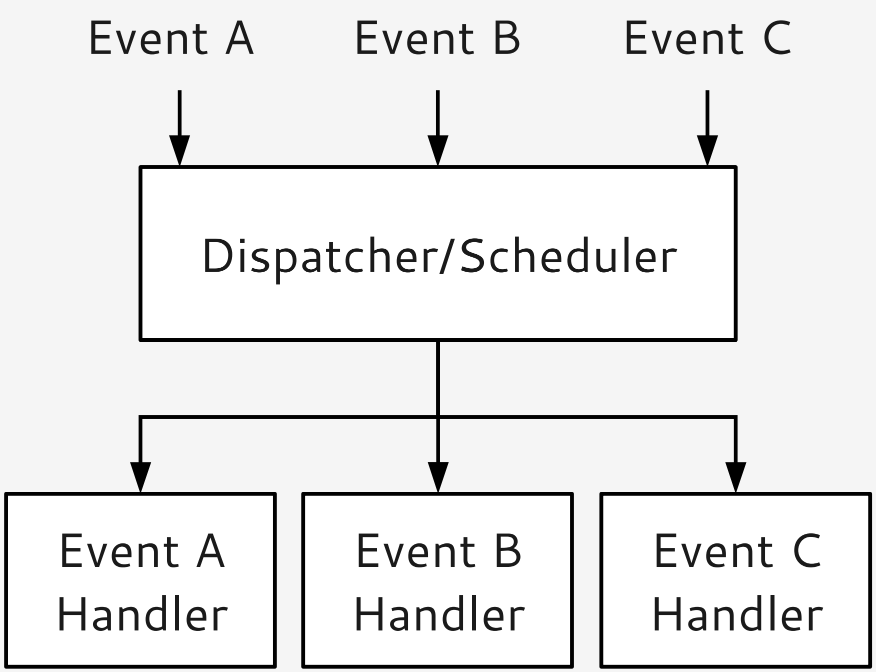






Event-driven programming

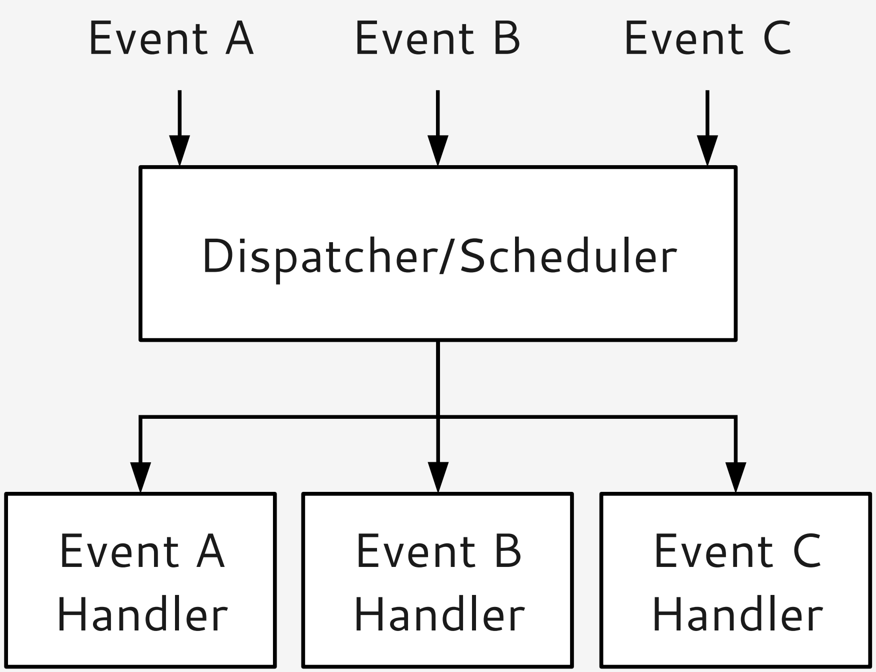
Focus: *computation is asynchronously triggered by events*



- Based on architectures accounting a dispatcher for events, received as messages.

Event-driven programming

Focus: *computation is asynchronously triggered by events*



- Based on architectures accounting a dispatcher for events, received as messages.
- Typical use: user interfaces, rule engines, reactive systems

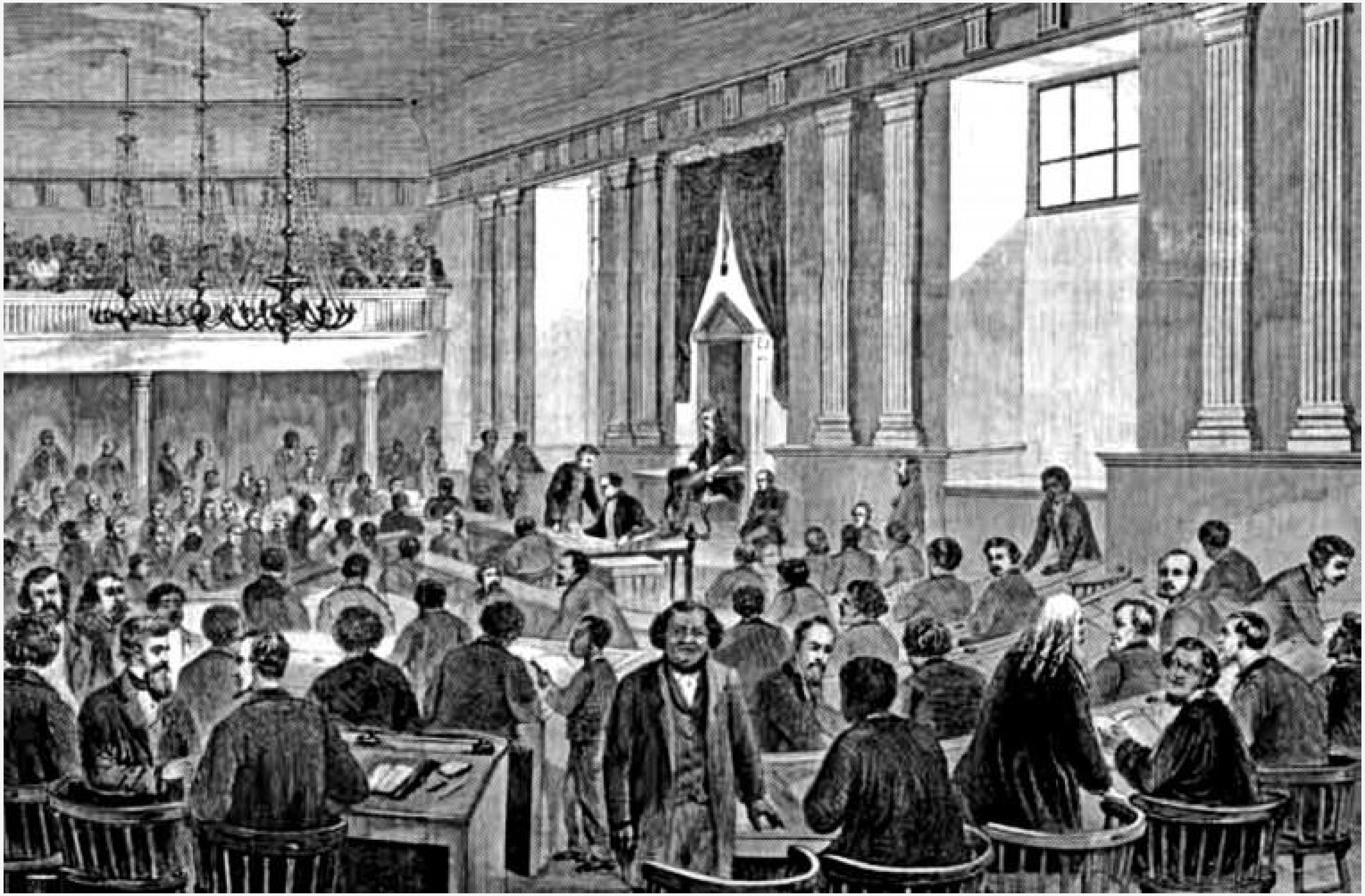
Static vs Dynamic

Static vs Dynamic

- Static
  - Properties are fixed when the program is compiled:
    - types of variables
    - definitions of types
    - code
- Dynamic
  - The same properties can be changed at *run-time*

Static programming in a metaphor..

Dynamic programming in a metaphor..



i.e. some sort of deliberation may change the rules (cf. the game Nomic)

Conclusion

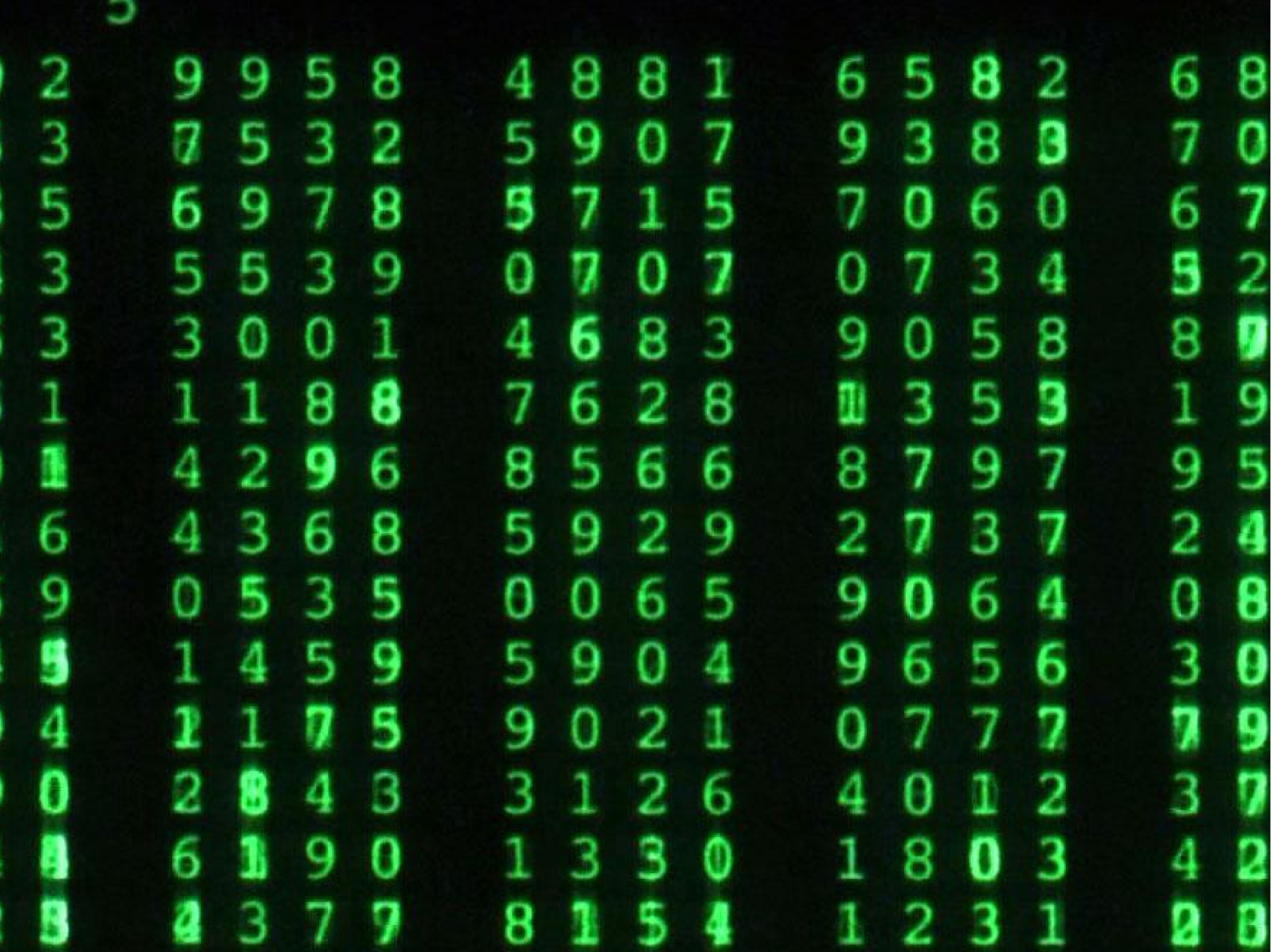
4 “axis” for paradigms

- Imperative vs Declarative
- Procedural vs Object-Oriented
- Sequential vs Concurrent (vs Parallel)
- Static vs Dynamic

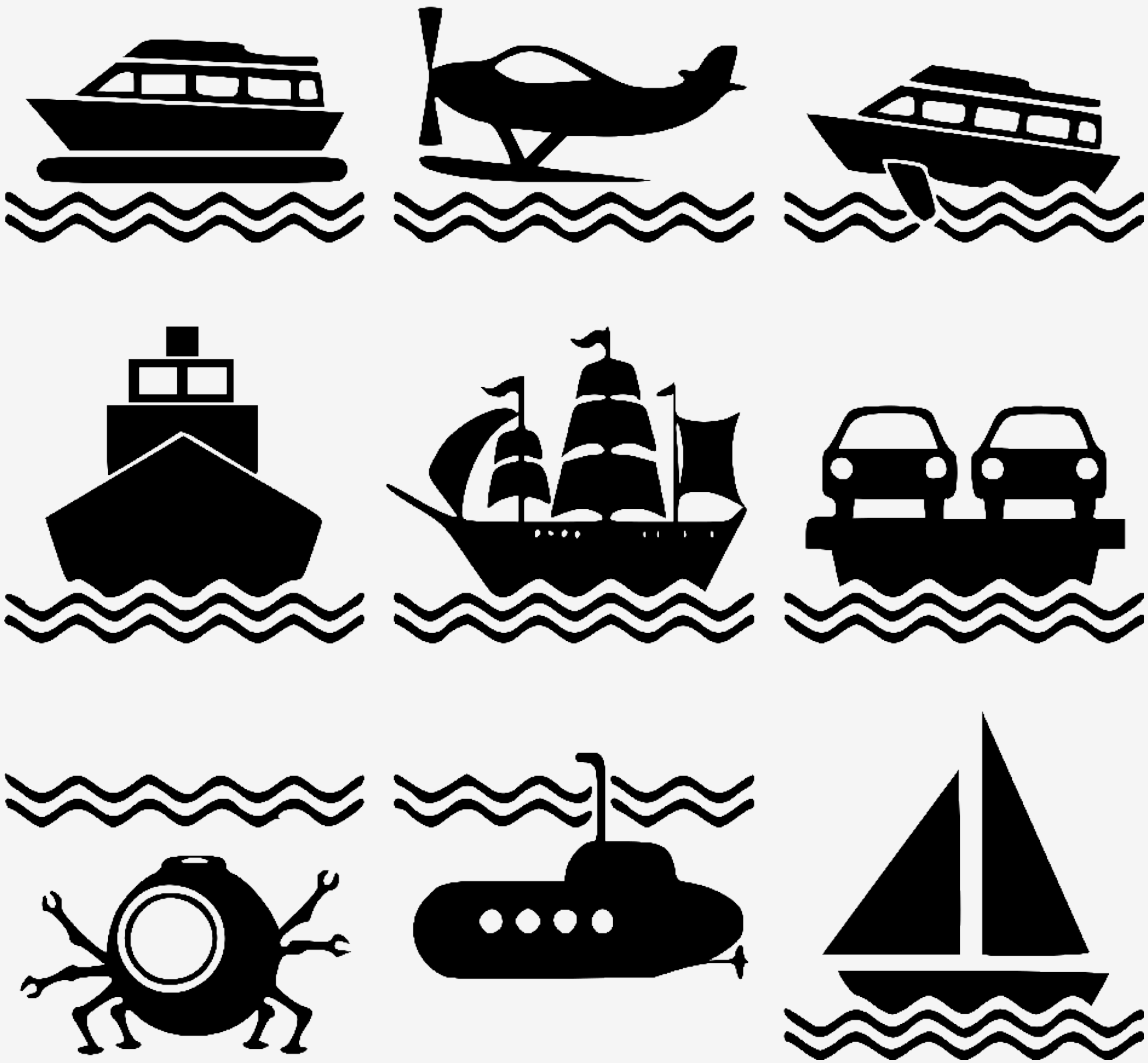
4 “axis” for paradigms

- Imperative vs Declarative
- Procedural vs Object-Oriented
- Sequential vs Concurrent (vs Parallel)
- Static vs Dynamic

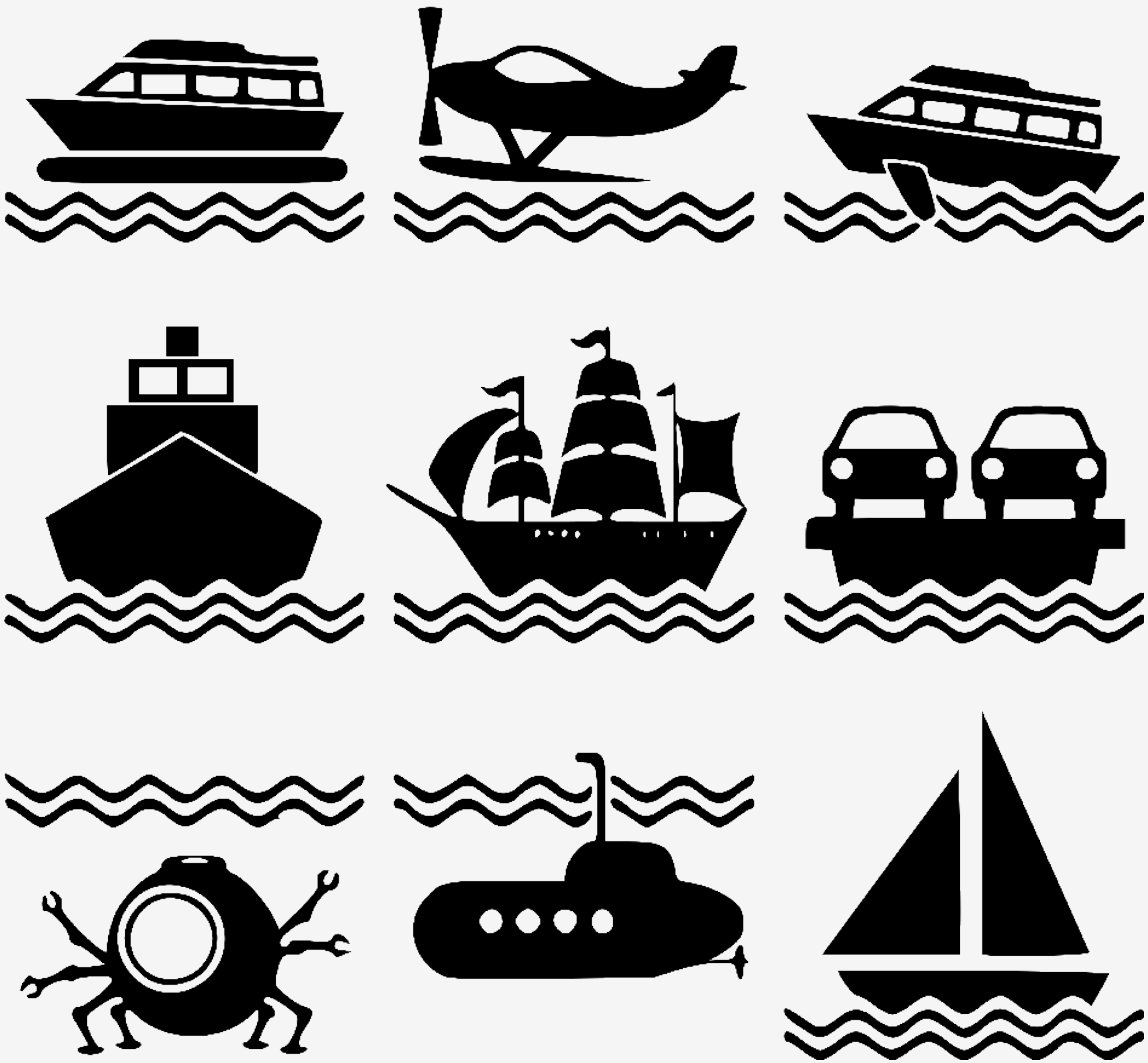
*However, whatever programming choice we make, there is an **ultimate truth**!*


2 9 9 5 8 4 8 8 1 6 5 8 2 6 8
3 7 5 3 2 5 9 0 7 9 3 8 8 7 0
5 6 9 7 8 5 7 1 5 7 0 6 0 6 7
3 5 5 3 9 0 7 0 7 0 7 3 4 5 2
3 3 0 0 1 4 6 8 3 9 0 5 8 8 7
1 1 1 8 8 7 6 2 8 1 3 5 3 1 9
1 4 2 9 6 8 5 6 6 8 7 9 7 9 5
6 4 3 6 8 5 9 2 9 2 7 3 7 2 4
9 0 5 3 5 0 0 6 5 9 0 6 4 0 8
5 1 4 5 9 5 9 0 4 9 6 5 6 3 0
4 1 1 7 5 9 0 2 1 0 7 7 7 7 9
0 2 8 4 3 3 1 2 6 4 0 1 2 3 7
4 6 1 9 0 1 3 3 0 1 8 0 3 4 2
3 4 3 7 7 8 1 5 4 1 2 3 1 0 8

At the end, everything will be always executed as machine code, so why bother?



- Practically speaking, all these navigates in waters...



- Practically speaking, all these navigates in waters...
- But depending of **what we need to do**, one is better than the other!

Building Information Systems

- understand the **problem**
- reflect about the right **representation**
- choose the best suited **paradigm** for the development of the solver

- only then, start **programming**!

Homework

Read the article on BB, mainly sections 2, 4, 5.

Programming Paradigms for Dummies: What Every Programmer Should Know, Peter Van Roy (2009)

- Analyze the ancient puzzle and the other problems in slide 45 with your group and argue which paradigm is the most/less suited, for each of them.
- Consider the Python language. Recognize on which paradigms it is based, providing examples of code.