COMP0017
Computability and Complexity Theory

Fabio Zanasi
http://www0.cs.ucl.ac.uk/staff/F.Zanasi/

Lecture seven

Previously on COMP0017

- Turing Machines
- Decidable and recognisable problems/languages
- The Church-Turing thesis
- Evidences of the Church-Turing thesis:
  - Variations of the definition of Turing machine
  - Other models of computation

In this lecture

We discuss a key concept of computer science: the possibility of a **universal** Turing machine.

Towards universal machines

A little experiment

Pick a word processor program, say Microsoft Word.

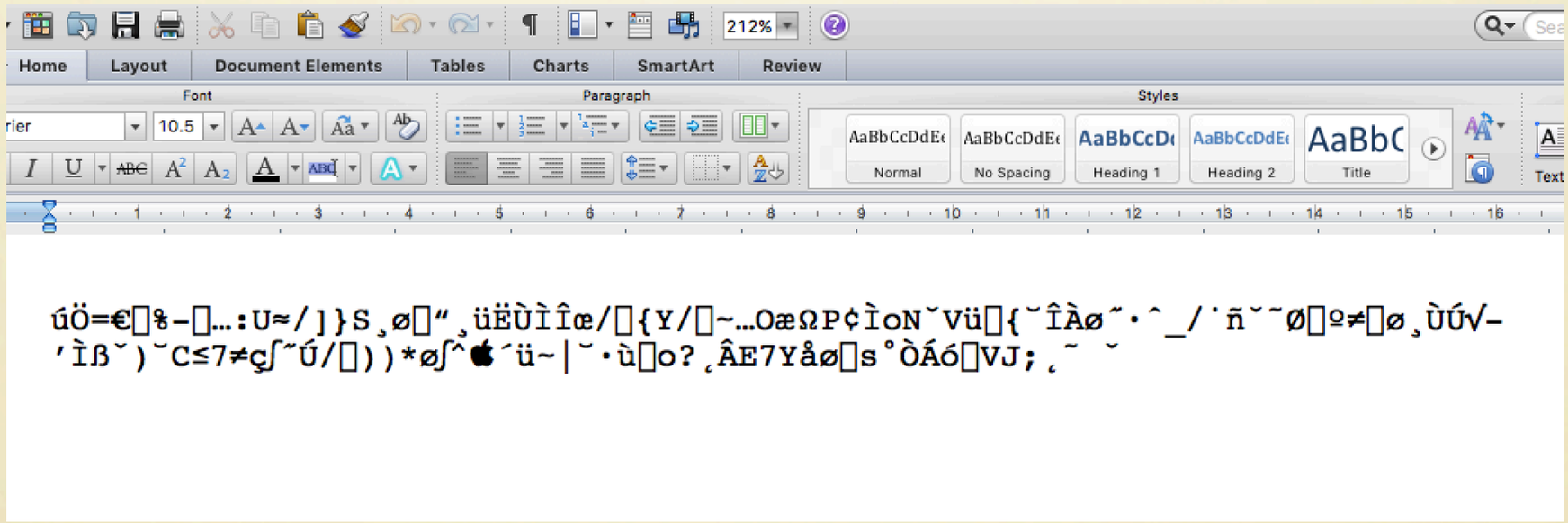
Let’s now open a *.doc file with Microsoft Word.

Let’s then open a *.exe file with Microsoft Word.

A little experiment

What do we observe?

Unlike with *.doc files, Microsoft Word does not produce a meaningful output with *exe files.



úÖ=€□%-□…:U≈/]}S¸ø□"¸üËÙÌÎœ/□{Y/□~…OæΩP¢ÌoN˜Vü□{˜ÎÀø˜·^_/·ñ˜˜Ø□º≠□ø¸ÙÚ√-
'Ìß˜)˜C≤7≠ç∫˜Ú/□))*ø∫^´ü~|˜·ù□o?¸ÂE7Yåø□s°ÒÁó□VJ;¸˜ ˜

A little experiment

What do we observe?

However, the important observation is that in principle Microsoft Word **can run** on *.exe files.

It can even run **on itself** (WINWORD.exe).

A little experiment

What we learned

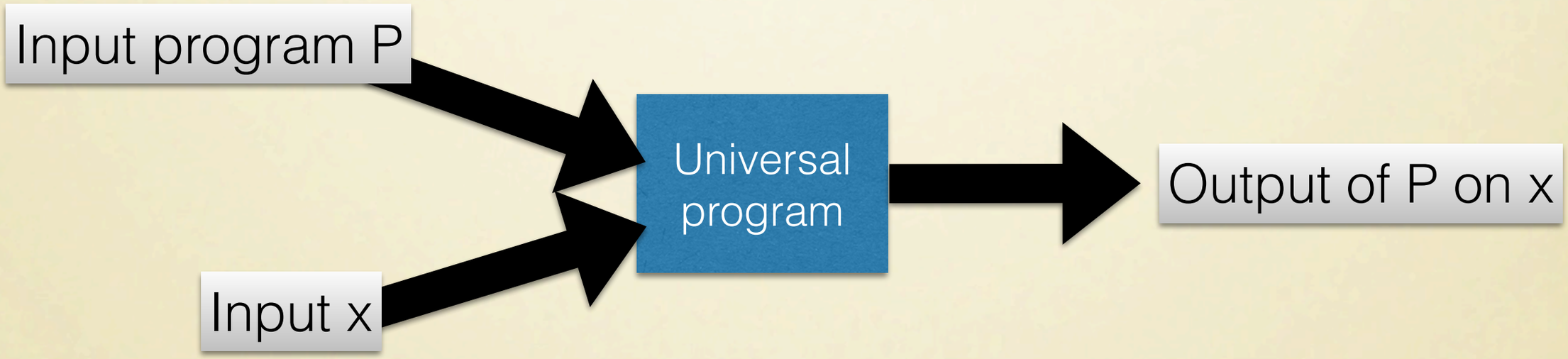
1. Any program can be run with any file as input.
   (although the resulting output will be garbage unless the input was intentionally produced to work with the program.)

2. Programs are just files. Thus a program can run with another program as input.

3. A program can be run using its own file as the input.

Universal programs

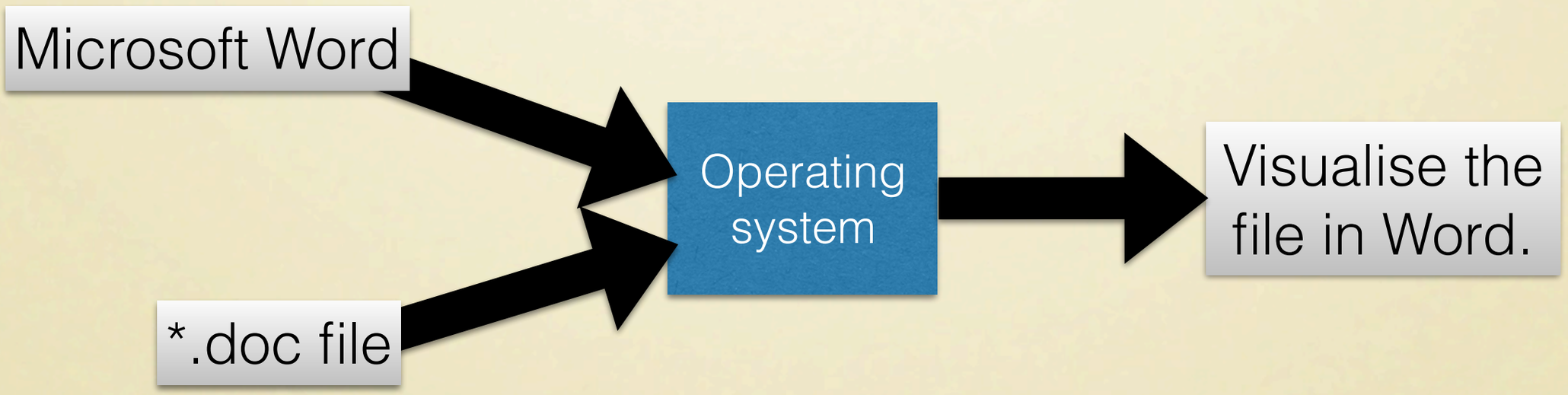
A universal program is one that treats other programs as meaningful inputs and runs them.



Universal programs

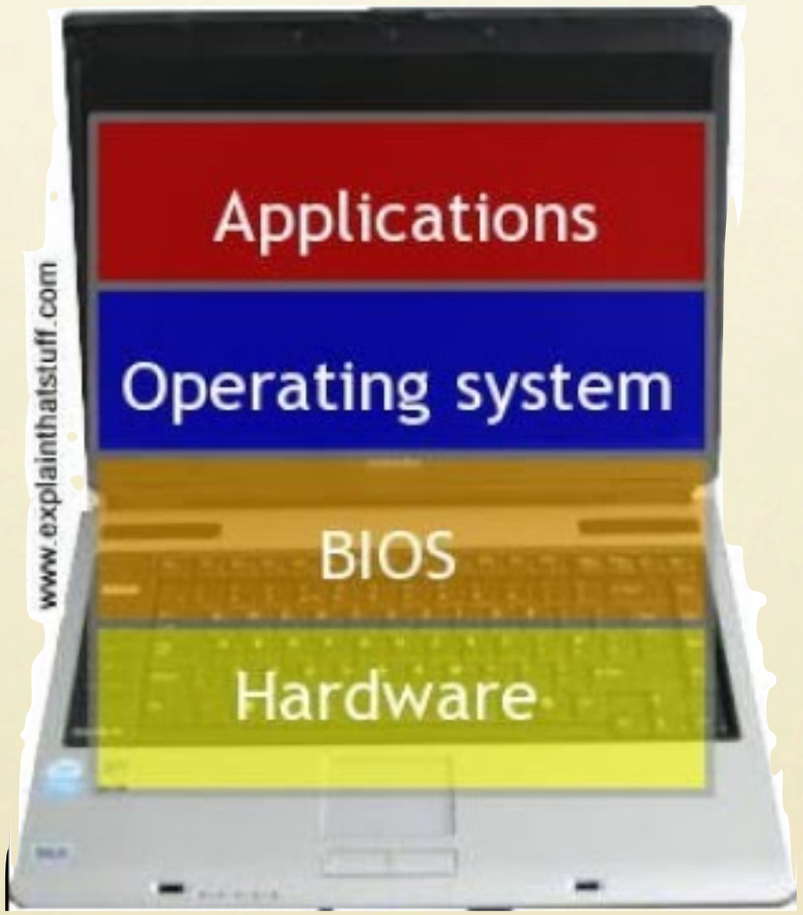
Operating systems (Windows, OS-X, Linux, …) are examples of universal programs.

When you click on a file, the operating system selects a program to run that file.



Universal programs

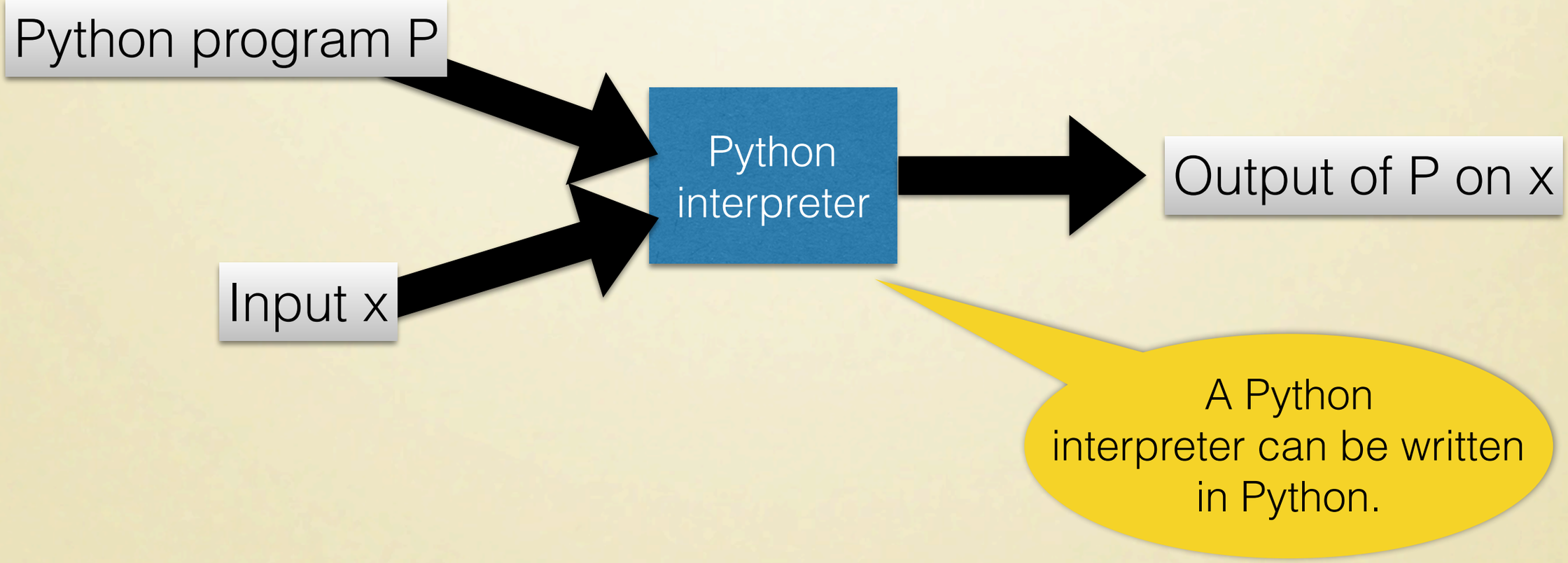
Actually, modern computer architectures are nothing but layers of universal computational devices.



Universal programs

An **interpreter** is another example of universal program.

```
Fabios-MBP:~ Fabio$ python program.py input
```



A bit of history

All of these concepts are descendants of Turing's intuition of a **Universal Turing machine** (introduced in his 1936 paper).

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions

A bit of history

Throughout human history, computational devices were mostly designed to do **one thing** (run a single program, accomplish a single task).

General-purpose machines are a modern invention (by Turing, and to some extent C. Babbage). The key advancement was to understand that there is no intrinsic difference between programs and their data.

A **universal machine** is one that is in principle capable of accomplishing any algorithmic task (it is **programmable**).

Self-reflection

Last week we saw how URMs can be coded as input data for a Turing machine.

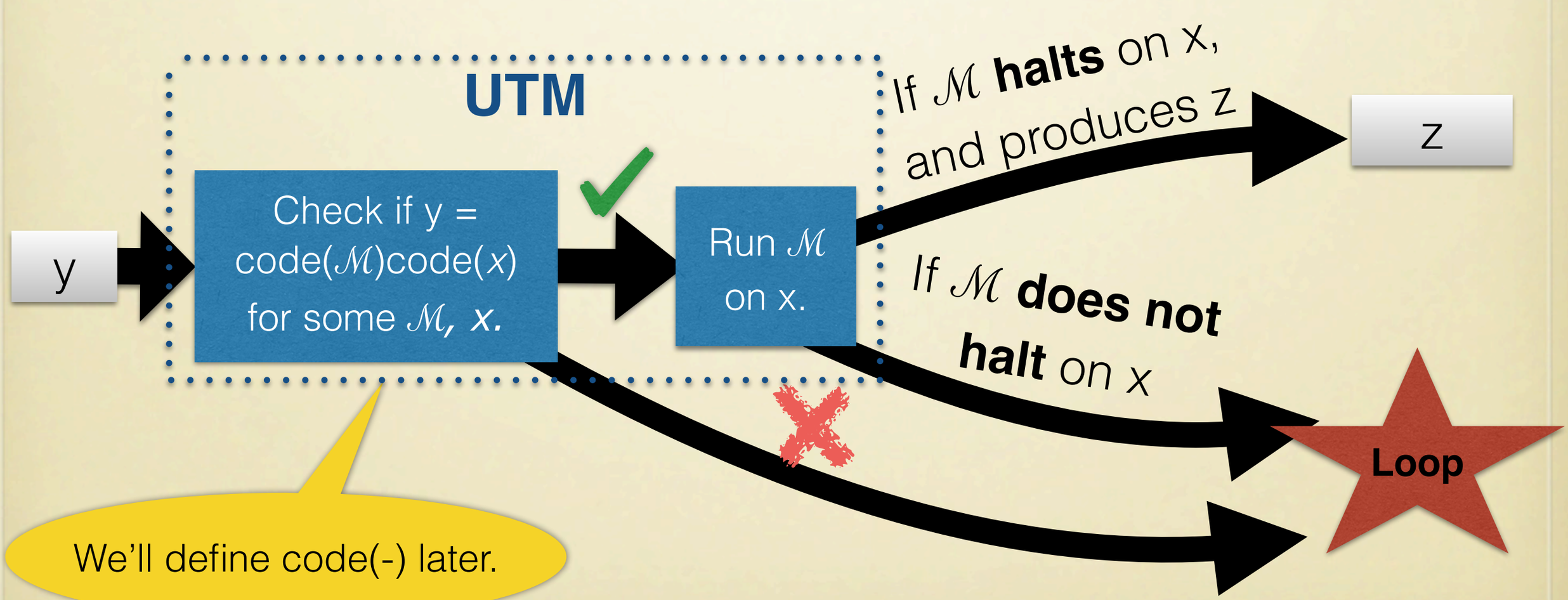
Now we will see that Turing machines themselves can be coded as input data for other Turing machines.

This kind of self-reflecting encoding is sometimes called **Gödelization**, after Kurt Gödel. He introduced it to encode statements about arithmetic within arithmetic itself, a key step of his *incompleteness theorem* (1931).

This idea is still very present in modern programming (interpreters, higher order functions, object oriented, …)

The universal Turing machine

- The UTM takes as input a string y.
- First, it checks whether y is the encoding of a TM $\mathcal{M}$ and of a string *x* in the alphabet Σ_I of $\mathcal{M}$.
- If so, the UTM simulates the action of $\mathcal{M}$ on *x*



What it means to simulate $\mathcal{M}$

Running the Universal Turing Machine on input y = code($\mathcal{M}$)code(x) should yield the same outcome as running $\mathcal{M}$ on input x.

UTM does not halt on code($\mathcal{M}$)code(x)

$\mathcal{M}$ does not halt on x

UTM halts on code($\mathcal{M}$)code(x) with output z.

$\mathcal{M}$ halts on x with output z.

Existence of a Universal Turing Machine

Theorem A Universal Turing Machine exists.

Proof We are going to explicitly construct one (in the same way as Turing did).

ON COMPUTABLE NUMBERS, WITH AN APPLICATION TO THE ENTSCHEIDUNGSPROBLEM

By A. M. Turing.

[Received 28 May, 1936.—Read 12 November, 1936.]

The "computable" numbers may be described briefly as the real numbers whose expressions as a decimal are calculable by finite means. Although the subject of this paper is ostensibly the computable *numbers*, it is almost equally easy to define and investigate computable functions

Encoding Turing Machines

Encoding Turing machines

We first focus on the definition of code(-).

It will translate a TM into a string over the alphabet {0,1}.

Disclaimer: there are many perfectly acceptable ways of doing this.

Encoding Turing machines

$$\mathcal{M} = \langle\, \Sigma,\, Q,\, q_0,\, H,\, \delta \,\rangle$$

We first introduce a few conventions.

- the states in Q are ordered as q_0, q_1, q_2, … with q_0 initial.
- We order symbols that may appear in the definition of δ

$$\sigma_0 = \sqcup \qquad \sigma_1 = \triangleright \qquad \sigma_2 = \rightarrow \qquad \sigma_3 = \leftarrow$$

and order the remaining symbols from Σ as σ_4, σ_5, …

We can then encode states and symbols as strings of 1s:

$$\text{code}(q_i) = \underbrace{11\ldots1}_{i+1 \text{ times}} \qquad \text{code}(\sigma_i) = \underbrace{11\ldots1}_{i+1 \text{ times}}$$

Encoding Turing machines

$$\mathcal{M} = \langle\, \Sigma,\, Q,\, q_0,\, H,\, \delta \,\rangle$$

- We encode a single tuple t = $\langle\, q_i,\, \sigma_n,\, q_j,\, \sigma_m \,\rangle$ of δ as:

$$\text{code}(t) = \text{code}(q_i)0\text{code}(\sigma_n)0\text{code}(q_j)0\text{code}(\sigma_m)0$$

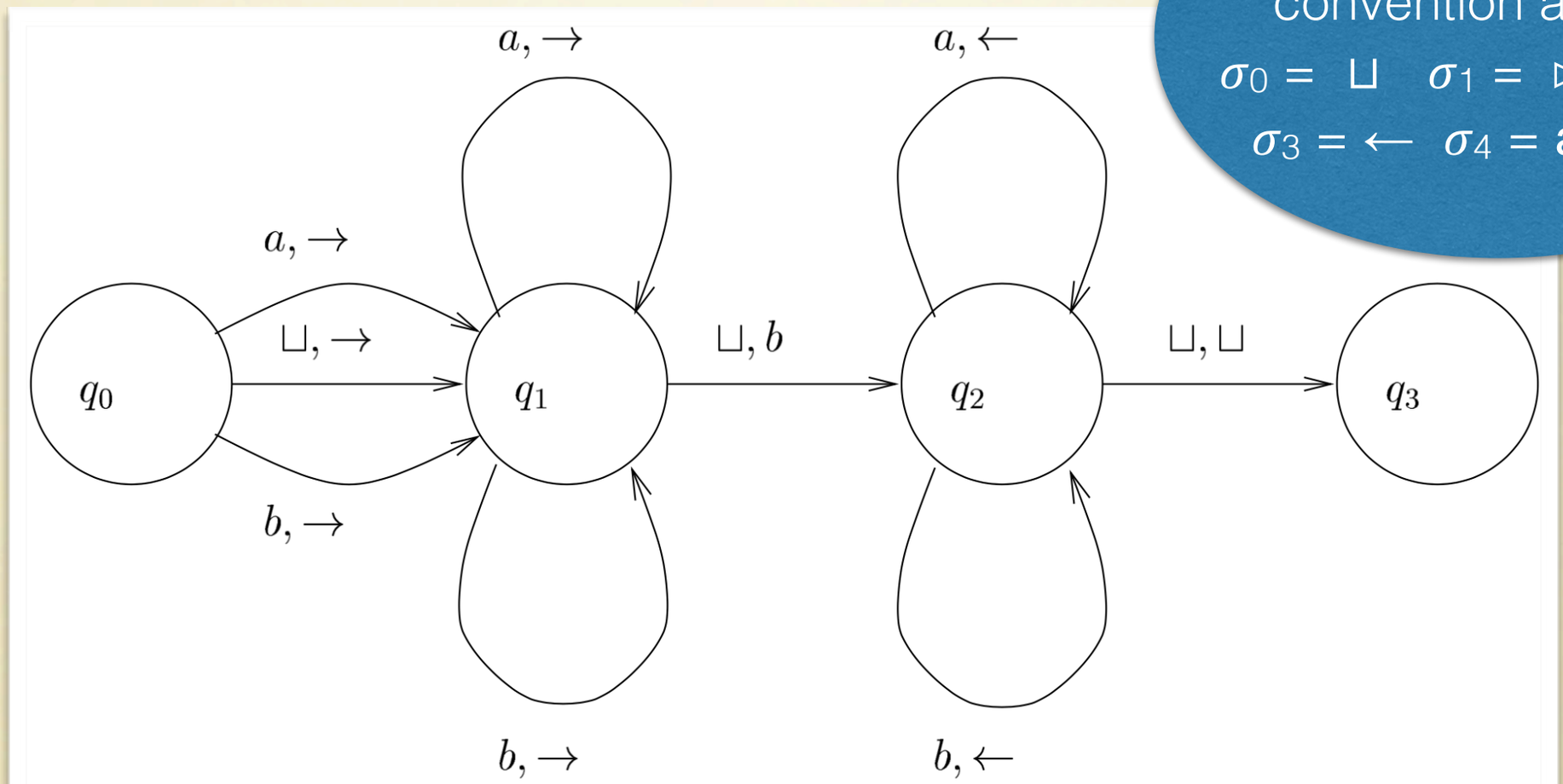
- And the whole of $\delta = \{t_1, t_2, \ldots, t_k\}$ as

$$\text{code}(\delta) = \text{code}(t_1)0\text{code}(t_2)0\ldots0\text{code}(t_k)0$$

- Halting states in *H* can be inferred as those on which δ *å*is not defined (= they never occur in second position in a tuple).

Example

Consider $\mathcal{M}$ with $\Sigma = \{a, b, \sqcup, \triangleright\}$, $Q = \{q_0, q_1, q_2, q_3\}$, $H = \{q_3\}$. It appends b to its input string.



We stick to our convention and call

$\sigma_0 = \sqcup$ $\sigma_1 = \triangleright$ $\sigma_2 = \rightarrow$

$\sigma_3 = \leftarrow$ $\sigma_4 = a$ $\sigma_5 = b$

Example

Writing the program δ as set of tuples, we have

$$
\begin{aligned}
t_1 &= (q_0, \sigma_0, q_1, \sigma_2) \\
t_2 &= (q_0, \sigma_1, q_0, \sigma_2) \\
t_3 &= (q_0, \sigma_4, q_1, \sigma_2) \\
t_4 &= (q_0, \sigma_5, q_1, \sigma_2) \\
t_5 &= (q_1, \sigma_0, q_2, \sigma_5) \\
t_6 &= (q_1, \sigma_1, q_1, \sigma_2)
\end{aligned}
$$

$$
\begin{aligned}
t_7 &= (q_1, \sigma_4, q_1, \sigma_2) \\
t_8 &= (q_1, \sigma_5, q_1, \sigma_2) \\
t_9 &= (q_2, \sigma_0, q_3, \sigma_0) \\
t_{10} &= (q_2, \sigma_1, q_2, \sigma_2) \\
t_{11} &= (q_2, \sigma_4, q_2, \sigma_3) \\
t_{12} &= (q_2, \sigma_5, q_2, \sigma_3)
\end{aligned}
$$

Example

The tuples are coded as:

$$\begin{aligned}
\mathsf{code}(t_1) &= 10101101110 \\
\mathsf{code}(t_2) &= 10110101110 \\
\mathsf{code}(t_3) &= 101111101101110 \\
\mathsf{code}(t_4) &= 1011111101101110 \\
\mathsf{code}(t_5) &= 1101011101111110 \\
\mathsf{code}(t_6) &= 1101101101110
\end{aligned}$$

$$\begin{aligned}
\mathsf{code}(t_7) &= 11011111101101110 \\
\mathsf{code}(t_8) &= 11011111101101110 \\
\mathsf{code}(t_9) &= 1110101111010 \\
\mathsf{code}(t_{10}) &= 1110110111101110 \\
\mathsf{code}(t_{11}) &= 11101111101110111110 \\
\mathsf{code}(t_{12}) &= 111011111101110111110
\end{aligned}$$

Example

Finally, the Turing machine $\mathcal{M}$ in its entirety is encoded as

code($\mathcal{M}$) =

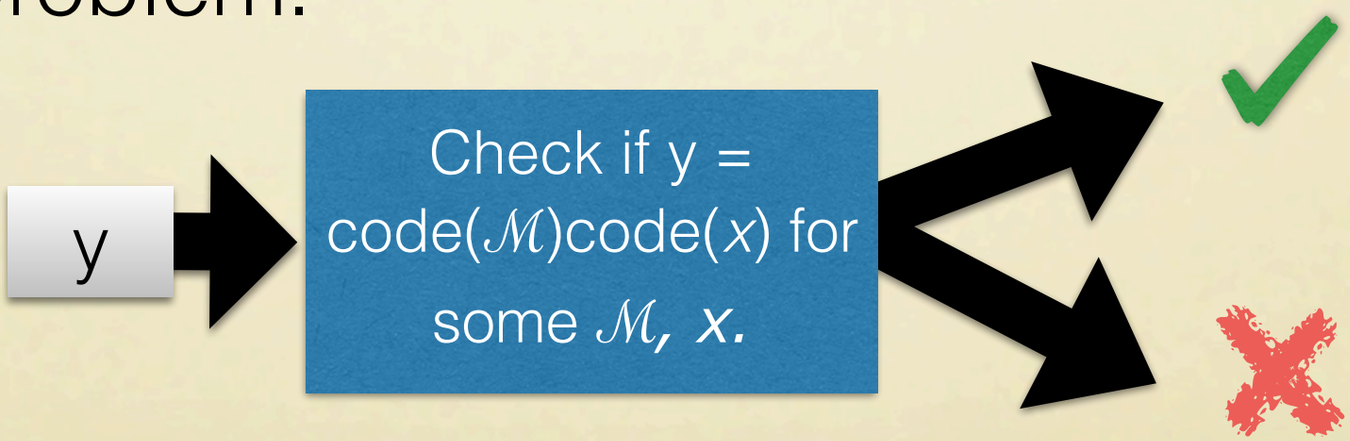
```
10101101110010110101110010111110110
11100101111111011011100110101111011111
11001101101101110011011111011011100
11011111101101110011101011110100111
01101110111001110111111011011110011
10111111011110111100
```

Inputs of $\mathcal{M}$ can be encoded similarly:

code($\sigma_{i1}\sigma_{i2}\ldots\sigma_{in}$) = 00code($\sigma_{i1}$)0code($\sigma_{i2}$)0…0code($\sigma_{in}$).

Observations on the encoding

- It is possible to have two or more machines that do the same job, but correspond to different strings in the encoding (e.g. if they use different states).
- Nonetheless, the coding is 1-to-1, that means, two *different* machines will be encoded as *different* strings.
- Given a string over {0,1}, it's possible to tell whether or not it is the code of some TM. In fact, this is a **decidable** problem.



Constructing the universal Turing machine

Reminder

Running the Universal Turing Machine on input y = code($\mathcal{M}$)code(x) should yield the same outcome as running $\mathcal{M}$ on input x.

UTM does not halt on code($\mathcal{M}$)code(x)		$\mathcal{M}$ does not halt on x
UTM halts on code($\mathcal{M}$)code(x) with output z.		$\mathcal{M}$ halts on x with output z.

Definition of the UTM

The UTM is defined as a three-tapes Turing machine.

Tape 1 will maintain the tape of $\mathcal{M}$ in encoded form.

Tape 2 will maintain code($\mathcal{M}$).

Tape 3 will maintain the current state of $\mathcal{M}$ in encoded form.

Definition of the UTM

To start a simulation of the UTM:

1. Preparation step: check if y = code($\mathcal{M}$)code(x) for some TM $\mathcal{M}$ and input x. If not, loop. If yes, it means tape 1 contains code($\mathcal{M}$)code(x). Go to step 2.
2. Move code($\mathcal{M}$) from tape 1 to tape 2.
3. Shift code(x) on tape 1 to the left and precede it with code(▷)0code(␣)0. Now tape 1 contains the starting tape of $\mathcal{M}$ on input x, in encoded form.
4. Write code(q_0) on tape 3.
5. Place head 1 at encoded ␣ before code(x), head 2 at the start of code($\mathcal{M}$), and head three at the start of code(q_0).

Definition of the UTM

One step of the simulation of $\mathcal{M}$ by the UTM works as follows.

1. Search in code($\mathcal{M}$) for a tuple where the first element matches the state stored on tape 3 and the second part matches the symbol currently scanned by $\mathcal{M}$.
2. Use the fourth element of the tuple to update tape 1 and the position (if necessary) of head 1.
3. Use the third element of the tuple to write the new state that $\mathcal{M}$ will move to on tape 3; or, if $\mathcal{M}$ has reached a halt state, halt.

Final consideration

This construction shows the existence of a universal Turing machine $\mathscr{M}_{\mathsf{U}}$.

Nothing prevents $\mathscr{M}_{\mathsf{U}}$ from receiving itself (in the encoded form code($\mathscr{M}_{\mathsf{U}}$)) as input of a computation!

This observation will be used in the next lecture to prove that some problems are undecidable.