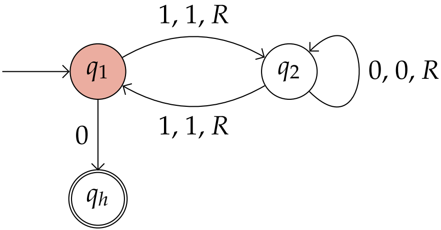
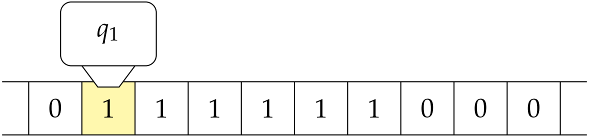
PHIL 222
Philosophical Foundations of Computer Science
Week 3, Tuesday

Sept. 10, 2024

Turing Machines: What is a Function? (cont'd)

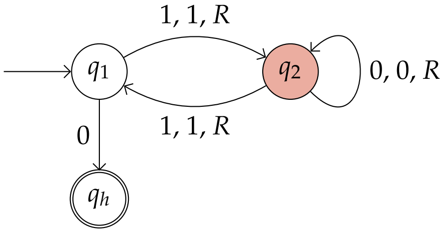
Consider the following Turing machine.

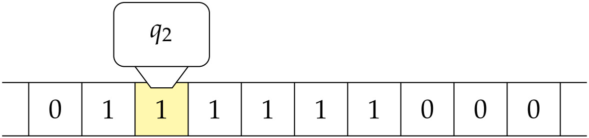




$t = 0$

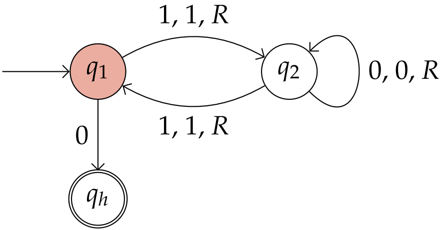
Consider the following Turing machine.

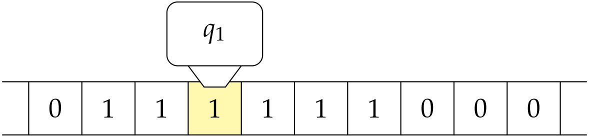




$t = 1$

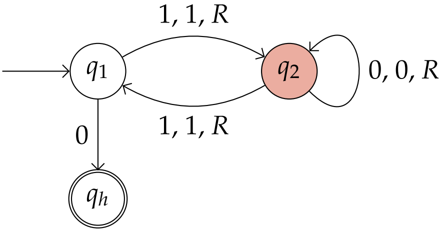
Consider the following Turing machine.

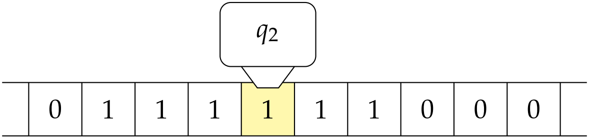




$t = 2$

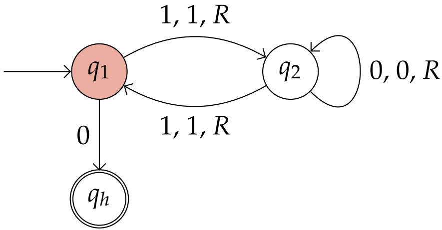
Consider the following Turing machine.

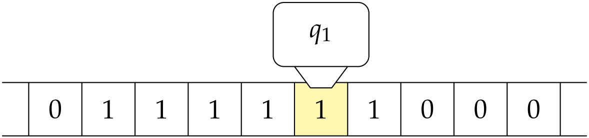




$t = 3$

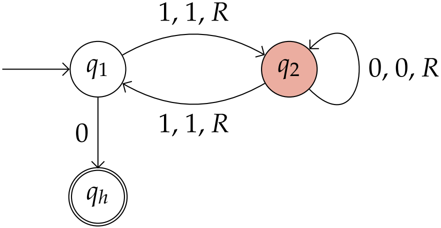
Consider the following Turing machine.

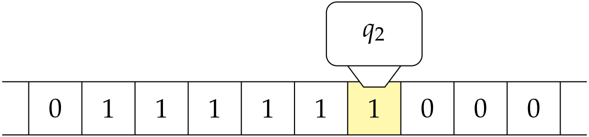




$t = 4$

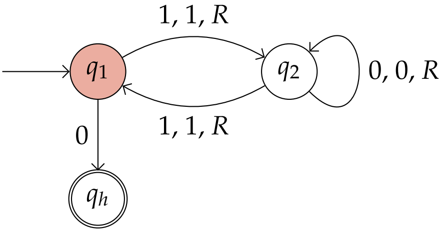
Consider the following Turing machine.

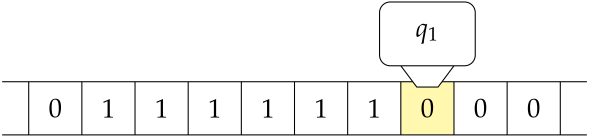




$t = 5$

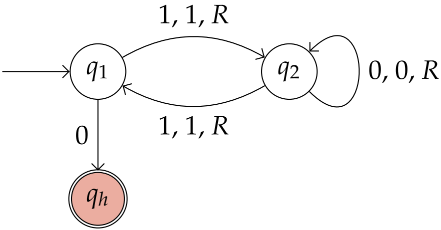
Consider the following Turing machine.

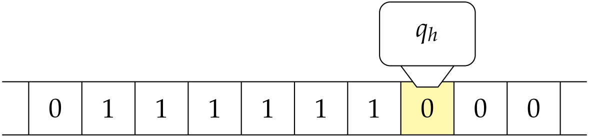




$t = 6$

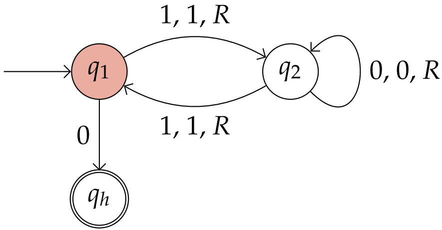
Consider the following Turing machine.

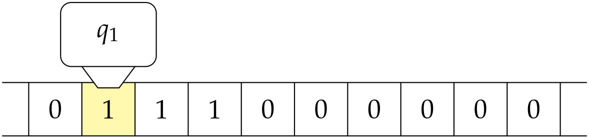




$t = 7$

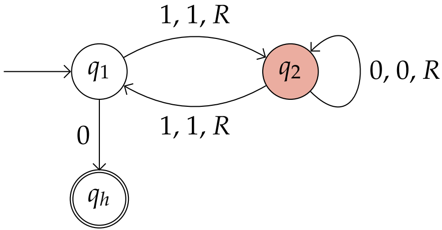
Consider the following Turing machine.

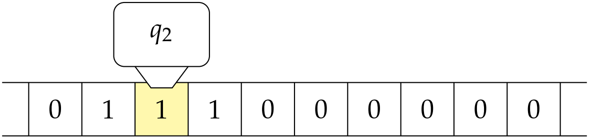




$t = 0$

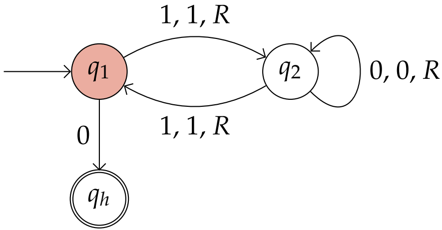
Consider the following Turing machine.

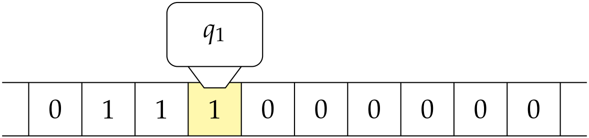




$t = 1$

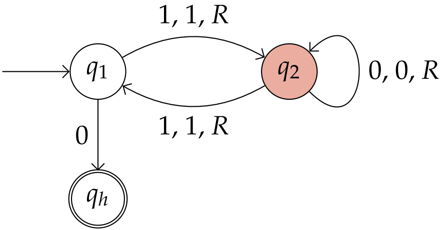
Consider the following Turing machine.

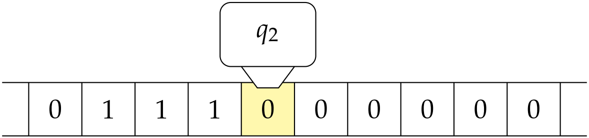




$t = 2$

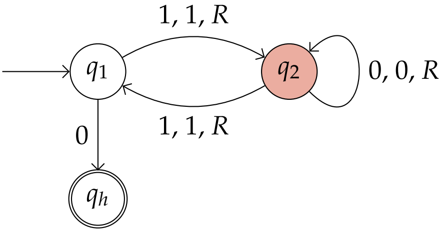
Consider the following Turing machine.

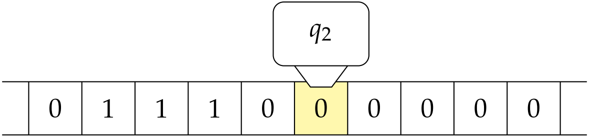




$t = 3$

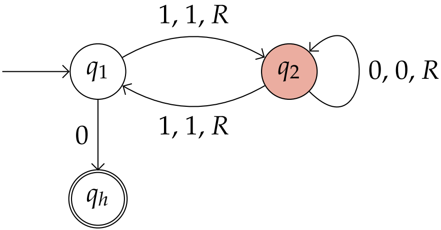
Consider the following Turing machine.

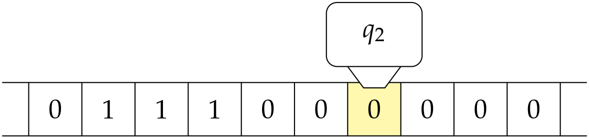




$t = 4$

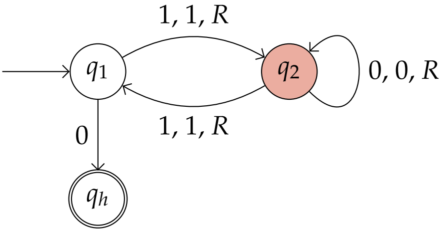
Consider the following Turing machine.

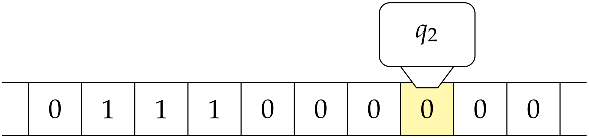




$t = 5$

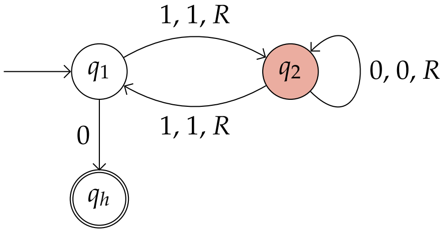
Consider the following Turing machine.

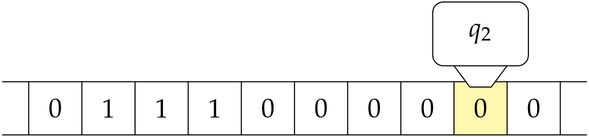




$t = 6$

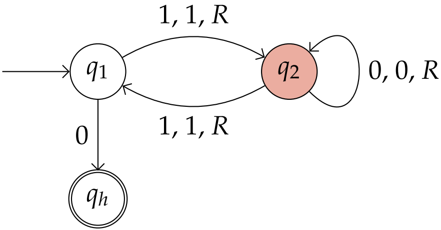
Consider the following Turing machine.

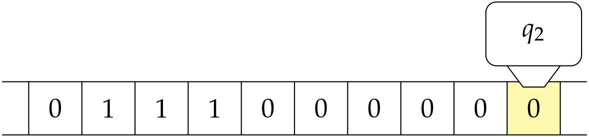




$t = 7$

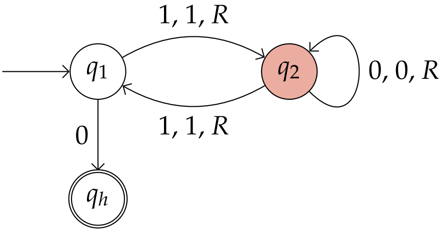
Consider the following Turing machine.

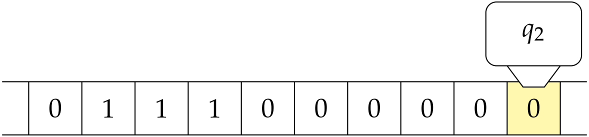




$t = 8$

Consider the following Turing machine.





$t = 8$

Some Turing machines, depending on the input, may never halt!

If the Turing machine on the previous slide computes any function, it is the following partial function:

- $\mathrm{IsEven}_{\mathrm{semi}} : n \mapsto \begin{cases} n & \text{if } n \text{ is even,} \\ \text{undefined} & \text{if } n \text{ is odd.} \end{cases}$

If the Turing machine on the previous slide computes any function, it is the following partial function:

- $\mathrm{IsEven}_{\mathrm{semi}} : n \mapsto \begin{cases} n & \text{if } n \text{ is even,} \\ \text{undefined} & \text{if } n \text{ is odd.} \end{cases}$

Thus, some Turing machines may fail to compute total functions. But we regard every Turing machine as computing some function or other, so we enter

Definition. We say that a partial function is **Turing computable** to mean that there is a Turing machine that computes it (or that can be interpreted as computing it . . .).

Turing Machines:
What Turing Machines Can Do

Composition.

Many functions can be obtained by composing others: e.g.,

$$(m, n) \mapsto 2(m + n)$$

is the composition of

$$+ : (m, n) \mapsto m + n \qquad \text{and} \qquad \text{double} : k \mapsto 2k,$$

Composition.

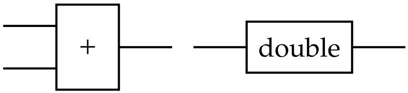
Many functions can be obtained by composing others: e.g.,

$$(m, n) \mapsto 2(m + n)$$

is the composition of

$$+ : (m, n) \mapsto m + n \qquad \text{and} \qquad \text{double} : k \mapsto 2k,$$

as in



Composition.

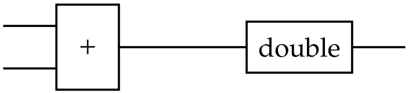
Many functions can be obtained by composing others: e.g.,

$$(m, n) \mapsto 2(m+n)$$

is the composition of

$$+ : (m, n) \mapsto m+n \qquad \text{and} \qquad \text{double} : k \mapsto 2k,$$

as in



Composition.

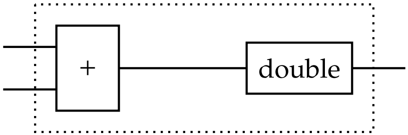
Many functions can be obtained by composing others: e.g.,

$$(m, n) \mapsto 2(m + n)$$

is the composition of

$$+ : (m, n) \mapsto m + n \qquad \text{and} \qquad \text{double} : k \mapsto 2k,$$

as in



Composition.

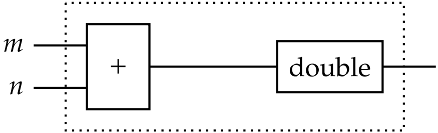
Many functions can be obtained by composing others: e.g.,

$$(m, n) \mapsto 2(m + n)$$

is the composition of

$$+ : (m, n) \mapsto m + n \qquad \text{and} \qquad \text{double} : k \mapsto 2k,$$

as in



Composition.

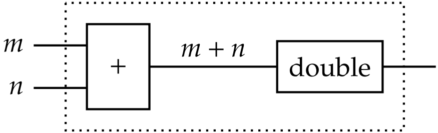
Many functions can be obtained by composing others: e.g.,

$$(m, n) \mapsto 2(m + n)$$

is the composition of

$$+ : (m, n) \mapsto m + n \qquad \text{and} \qquad \text{double} : k \mapsto 2k,$$

as in



Composition.

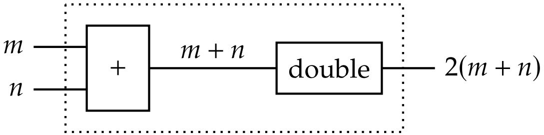
Many functions can be obtained by composing others: e.g.,

$$(m, n) \mapsto 2(m + n)$$

is the composition of

$$+ : (m, n) \mapsto m + n \qquad \text{and} \qquad \text{double} : k \mapsto 2k,$$

as in



Composition.

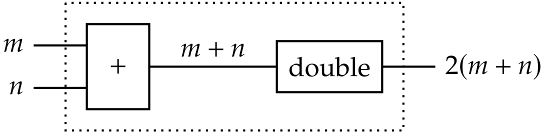
Many functions can be obtained by composing others: e.g.,

$$(m, n) \mapsto 2(m + n)$$

is the composition of

$$+ : (m, n) \mapsto m + n \qquad \text{and} \qquad \text{double} : k \mapsto 2k,$$

as in



We write $g \circ f$ for the composition "first f and then g"
(because $g \circ f(m, n) = g(f(m, n))$).

Composition.

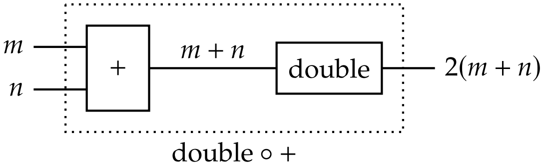
Many functions can be obtained by composing others: e.g.,

$$(m, n) \mapsto 2(m + n)$$

is the composition of

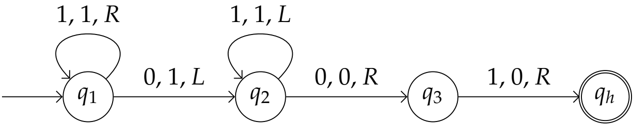
$$+ : (m, n) \mapsto m + n \qquad \text{and} \qquad \text{double} : k \mapsto 2k,$$

as in

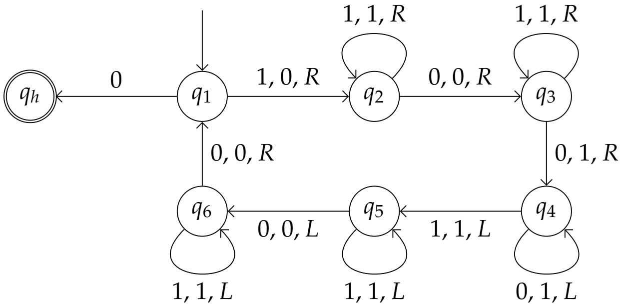


We write $g \circ f$ for the composition “first f and then g”
(because $g \circ f(m, n) = g(f(m, n))$).

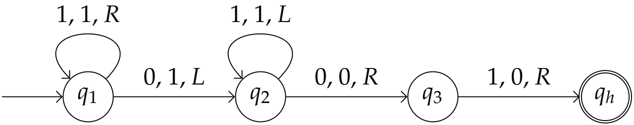
+ is computed by

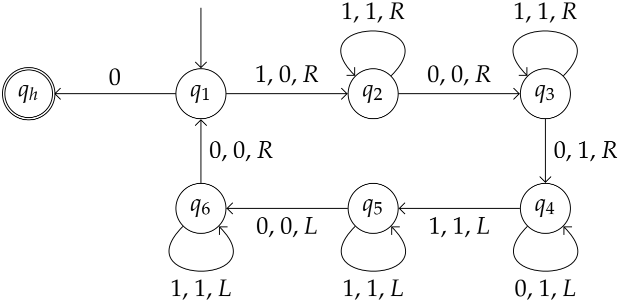


and double is computed by

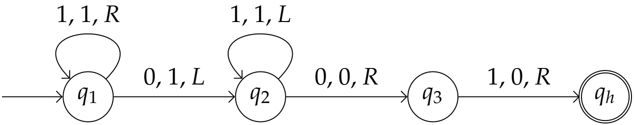


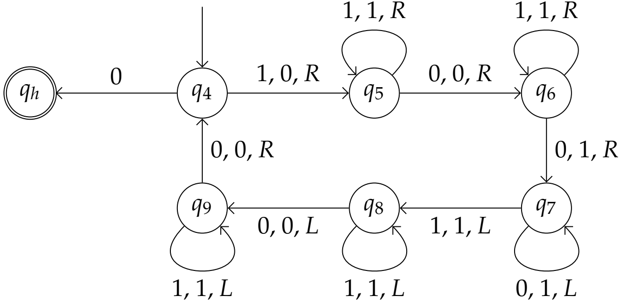
Then the composition double $\circ$ + is computed by



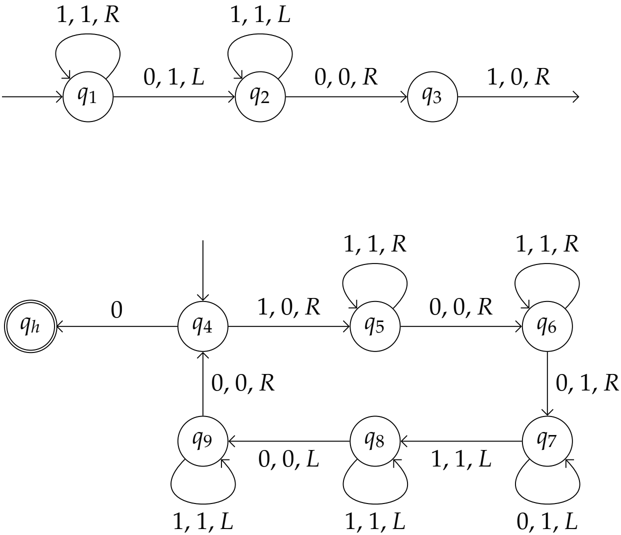


Then the composition double ∘ + is computed by

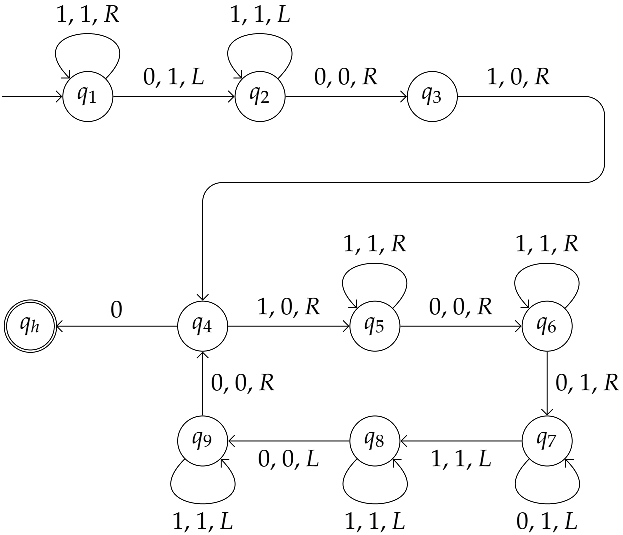




Then the composition double $\circ$ + is computed by

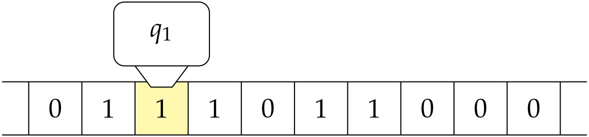


Then the composition double ∘ + is computed by



Coding and a universal Turing machine.

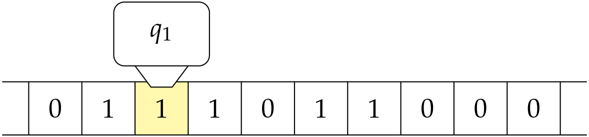
The state of a Turing machine and a tape combined



can be represented by

Coding and a universal Turing machine.

The state of a Turing machine and a tape combined

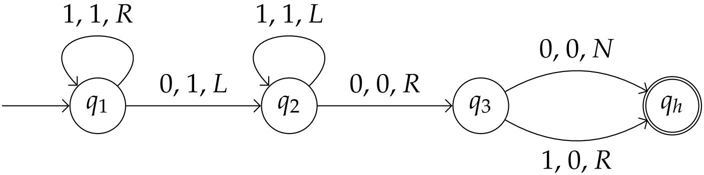


can be represented by

$$1q_1 11011$$

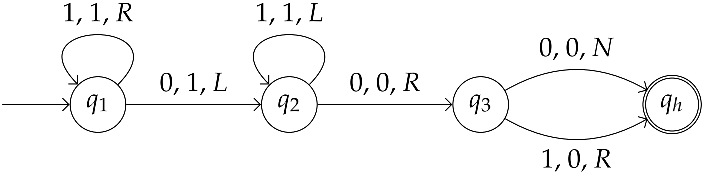
(called a “complete configuration” or “instantaneous description”).

A Turing machine is represented by a flowchart ("state diagram"),



but can also be represented by

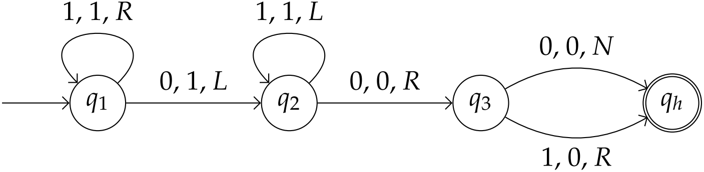
A Turing machine is represented by a flowchart ("state diagram"),



but can also be represented by

	0	1
q_1	$1, L, q_2$	$1, R, q_1$
q_2	$0, R, q_3$	$1, L, q_2$
q_3	$0, N, q_h$	$0, R, q_h$

A Turing machine is represented by a flowchart (“state diagram”),

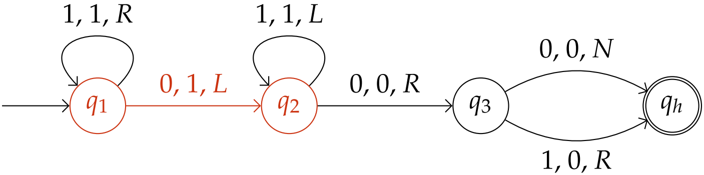


but can also be represented by

	0	1
q_1	$1, L, q_2$	$1, R, q_1$
q_2	$0, R, q_3$	$1, L, q_2$
q_3	$0, N, q_h$	$0, R, q_h$

or $\quad ;q_1 01 L q_2; q_1 11 R q_1; q_2 00 R q_3; q_2 11 L q_2; q_3 00 L q_h; q_3 10 R q_h$

A Turing machine is represented by a flowchart ("state diagram"),



but can also be represented by

	0	1
q_1	$1, L, q_2$	$1, R, q_1$
q_2	$0, R, q_3$	$1, L, q_2$
q_3	$0, N, q_h$	$0, R, q_h$

or $\quad ; q_1 01 L q_2 ; q_1 11 R q_1 ; q_2 00 R q_3 ; q_2 11 L q_2 ; q_3 00 L q_h ; q_3 10 R q_h$

Now let's take another look at
`https://turingmachine.io/`

What you just saw is a computer simulating Turing machines — indeed, a computer that can simulate any Turing machine!

What you just saw is a computer simulating Turing machines — indeed, a computer that can simulate any Turing machine!

More precisely, it takes in two inputs,

1. a code representing a Turing machine M,
2. a code representing an input x for Turing machines,

and outputs the result of feeding x to M.

What you just saw is a computer simulating Turing machines — indeed, a computer that can simulate any Turing machine!

More precisely, it takes in two inputs,

1. a code representing a Turing machine M,
2. a code representing an input x for Turing machines,

and outputs the result of feeding x to M.

Turing's idea: There is a Turing machine that can do this!

What you just saw is a computer simulating Turing machines — indeed, a computer that can simulate any Turing machine!

More precisely, it takes in two inputs,

1. a code representing a Turing machine M,
2. a code representing an input x for Turing machines,

and outputs the result of feeding x to M.

Turing's idea: There is a Turing machine that can do this!

Definition. A "universal Turing machine" is a Turing machine that does the computation above — i.e., that takes in a combination of a code of a Turing machine M and another input x as input, and outputs the result of feeding x to M.

E.g., feed it with codes of

$$;q_1 01 L q_2; q_1 11 R q_1; q_2 00 R q_3; q_2 11 L q_2; q_3 00 L q_h; q_3 10 R q_h$$

and $q_1 111011$; then it outputs 11111.

Church-Turing Thesis.

Now let's get back to this:

Definition. We say that a function is "Turing computable" to mean that there is a Turing machine that computes it.

Church-Turing Thesis.

Now let's get back to this:

Definition. We say that a function is "Turing computable" to mean that there is a Turing machine that computes it.

It follows that a function is

$$\text{Turing computable} \iff \text{a universal Turing machine can compute it.}$$

Church-Turing Thesis.

Now let's get back to this:

Definition. We say that a function is "Turing computable" to mean that there is a Turing machine that computes it.

It follows that a function is

Turing computable $\iff$ a universal Turing machine can compute it.

It seems correct to say that anything Turing computable is computed mechanically or effectively. How about the converse? Does it hold?

Church-Turing Thesis.

Now let's get back to this:

Definition. We say that a function is "Turing computable" to mean that there is a Turing machine that computes it.

It follows that a function is

$$\text{Turing computable} \iff \text{a universal Turing machine can compute it.}$$

It seems correct to say that anything Turing computable is computed mechanically or effectively. How about the converse? Does it hold?

"Yes" is called **the Church-Turing thesis**:
Any mechanical / effective method can be carried out by some Turing machine, and therefore by the universal Turing machine.

Turing Machines:
What Turing Machines Cannot Do

Advice

We are going to see the first technical “theorem” of this course.

- Its content may be relevant to the final (true / false) exam.
- Its proof may be tough, but do not worry too much: you will not be tested for the understanding of the proof.

But it will be nice to understand the proof. If you find it hard and want my help to understand it,

- *Come to see me in office hours & appointments!*

Uncomputable numbers.

There are (even mathematical) things Turing computers cannot do . . . .

Uncomputable numbers.

There are (even mathematical) things Turing computers cannot do . . . .

Definition. A real number x is called computable if there is a computable function $f : \mathbb{N} \to \mathbb{Z}$ that approximates it, in the sense that

$$\frac{f(n)}{10^n} = x \text{ rounded down to } n \text{ decimal places.}$$

E.g., π is computable: there is a Turing computable function f s.th.

$$f(0) = 3, \qquad f(2) = 314, \qquad f(10) = 31415926535, \qquad \text{etc.,}$$

$$\text{i.e.,} \qquad \frac{f(0)}{10^0} = 3, \qquad \frac{f(2)}{10^2} = 3.14, \qquad \frac{f(10)}{10^{10}} = 3.1415926535, \qquad \text{etc.}$$

Uncomputable numbers.

There are (even mathematical) things Turing computers cannot do . . . .

Definition. A real number x is called computable if there is a computable function $f : \mathbb{N} \to \mathbb{Z}$ that approximates it, in the sense that

$$\frac{f(n)}{10^n} = x \text{ rounded down to } n \text{ decimal places.}$$

E.g., π is computable: there is a Turing computable function f s.th.

$$f(0) = 3, \qquad f(2) = 314, \qquad f(10) = 31415926535, \qquad \text{etc.,}$$

i.e., $$\frac{f(0)}{10^0} = 3, \qquad \frac{f(2)}{10^2} = 3.14, \qquad \frac{f(10)}{10^{10}} = 3.1415926535, \qquad \text{etc.}$$

But the overwhelming majority of real numbers are not computable.

Theorem. There are Turing uncomputable real numbers.

Theorem. There are Turing uncomputable real numbers.

Proof. Remember that we can code all the Turing machines; let's code them by natural numbers. Then we can order all the Turing machines by comparing their natural-number codes.

Theorem. There are Turing uncomputable real numbers.

Proof. Remember that we can code all the Turing machines; let's code them by natural numbers. Then we can order all the Turing machines by comparing their natural-number codes. So let's "enumerate" all the Turing machines M_n that compute a real number x_n:

$$M_0, M_1, M_2, \ldots.$$

Theorem. There are Turing uncomputable real numbers.

Proof. Remember that we can code all the Turing machines; let's code them by natural numbers. Then we can order all the Turing machines by comparing their natural-number codes. So let's "enumerate" all the Turing machines M_n that compute a real number x_n:

$$M_0, M_1, M_2, \dots .$$

Then write $x_{n,0}$ for the integer part of x_n and $x_{n,m}$ ($m > 0$) for the digit in the mth decimal place of x_n. Visually, we have the following table:

$$\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & \cdots \\
\hline
x_0 & x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} & \cdots \\
x_1 & x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} & \cdots \\
x_2 & x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} & \cdots \\
x_3 & x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

$$
\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & \cdots \\
\hline
x_0 & x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} & \cdots \\
x_1 & x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} & \cdots \\
x_2 & x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} & \cdots \\
x_3 & x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}
$$

$x_{n,0}$ is the integer part of x_n, and
$x_{n,m}$ ($m > 0$) is the digit in the mth decimal place of x_n.

$$
\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & \cdots \\
\hline
x_0 & x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} & \cdots \\
x_1 & x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} & \cdots \\
x_2 & x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} & \cdots \\
x_3 & x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}
$$

$x_{n,0}$ is the integer part of x_n, and
$x_{n,m}$ ($m > 0$) is the digit in the mth decimal place of x_n.

Now define a real number d with the integer part $d_0 = x_{0,0} + 1$ and

$$
d_m = \begin{cases} x_{m,m} + 1 & \text{if } x_{m,m} = 0, \\ x_{m,m} - 1 & \text{if } 1 \leqslant x_{m,m} \leqslant 9 \end{cases}
$$

in the mth decimal place.

$$
\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & \cdots \\
\hline
x_0 & x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} & \cdots \\
x_1 & x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} & \cdots \\
x_2 & x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} & \cdots \\
x_3 & x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}
$$

$x_{n,0}$ is the integer part of x_n, and
$x_{n,m}$ ($m > 0$) is the digit in the mth decimal place of x_n.

Now define a real number d with the integer part $d_0 = x_{0,0} + 1$ and

$$
d_m = \begin{cases} x_{m,m} + 1 & \text{if } x_{m,m} = 0, \\ x_{m,m} - 1 & \text{if } 1 \leqslant x_{m,m} \leqslant 9 \end{cases}
$$

in the mth decimal place. Then d differs from every x_n, because $d_n \neq x_{n,n}$ (and because d has no 9 in any decimal place).

$$
\begin{array}{c|ccccc}
 & 0 & 1 & 2 & 3 & \cdots \\
\hline
x_0 & x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} & \cdots \\
x_1 & x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} & \cdots \\
x_2 & x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} & \cdots \\
x_3 & x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}
$$

$x_{n,0}$ is the integer part of x_n, and
$x_{n,m}$ ($m > 0$) is the digit in the mth decimal place of x_n.

Now define a real number d with the integer part $d_0 = x_{0,0} + 1$ and

$$
d_m = \begin{cases} x_{m,m} + 1 & \text{if } x_{m,m} = 0, \\ x_{m,m} - 1 & \text{if } 1 \leqslant x_{m,m} \leqslant 9 \end{cases}
$$

in the mth decimal place. Then d differs from every x_n, because $d_n \neq x_{n,n}$ (and because d has no 9 in any decimal place). Therefore d is not Turing computable. □

Recursive Functions

Advice

This is a technical part of the course that will be covered in the technical exercises and the midterm exam.

If anything here does not "click" in your mind,

- *Come to see me in office hours & appointments!*

You are only expected to learn what the rule of the game is like. It is absolutely natural if it does not "click" in your mind for the first time you see it. But to resolve that situation, you need interactive help. Please, please help me help you.

For the next model of computation,

let's shift our focus from "What computation looks like?"

to "What can be computed?"

In particular, "What functions of natural numbers are computable?"

For the next model of computation,

let's shift our focus from "What computation looks like?"

to "What can be computed?"

In particular, "What functions of natural numbers are computable?"

E.g., we have seen Turing machines compute:

- add : $(m, n) \mapsto m + n$.
- double : $n \mapsto 2n$.
- $\mathrm{IsEven}_{\mathrm{semi}} : n \mapsto \begin{cases} n & \text{if } n \text{ is even,} \\ \text{undefined} & \text{if } n \text{ is odd.} \end{cases}$

For the next model of computation,
let's shift our focus from "What computation looks like?"
to "What can be computed?"

In particular, "What functions of natural numbers are computable?"

E.g., we have seen Turing machines compute:

- add : $(m, n) \mapsto m + n$.
- double : $n \mapsto 2n$.
- $\mathrm{IsEven}_{\mathrm{semi}} : n \mapsto \begin{cases} n & \text{if } n \text{ is even,} \\ \text{undefined} & \text{if } n \text{ is odd.} \end{cases}$

But then we know the following are Turing computable, too:

- double $\circ$ add : $(m, n) \mapsto 2(m + n)$.
- $\mathrm{IsEven}_{\mathrm{semi}} \circ \mathrm{double} : n \mapsto 2n$.

For the next model of computation,

let's shift our focus from "What computation looks like?"

to "What can be computed?"

In particular, "What functions of natural numbers are computable?"

E.g., we have seen Turing machines compute:

- $\mathrm{add} : (m, n) \mapsto m + n$.
- $\mathrm{double} : n \mapsto 2n$.
- $\mathrm{IsEven}_{\mathrm{semi}} : n \mapsto \begin{cases} n & \text{if } n \text{ is even,} \\ \text{undefined} & \text{if } n \text{ is odd.} \end{cases}$

But then we know the following are Turing computable, too:

- $\mathrm{double} \circ \mathrm{add} : (m, n) \mapsto 2(m + n)$.
- $\mathrm{IsEven}_{\mathrm{semi}} \circ \mathrm{double} : n \mapsto 2n$.

We start from simple, obviously computable functions, and show more complicated ones to be computable, by showing that they can be built from simple ones by simple operations (e.g. composition).

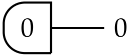
The Basic Six

So what functions would be obviously computable?

The Basic Six

So what functions would be obviously computable?

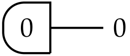
- The zero:



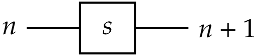
The Basic Six

So what functions would be obviously computable?

- The zero:



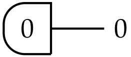
- The successor:



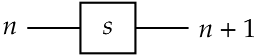
The Basic Six

So what functions would be obviously computable?

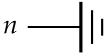
- The zero:



- The successor:



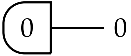
- The discarding:



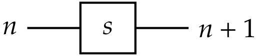
The Basic Six

So what functions would be obviously computable?

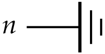
- The zero:



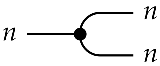
- The successor:



- The discarding:



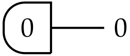
- The duplication:



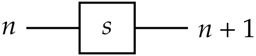
The Basic Six

So what functions would be obviously computable?

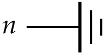
- The zero:



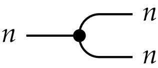
- The successor:



- The discarding:



- The duplication:

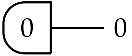


- The identity:

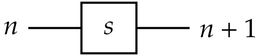
The Basic Six

So what functions would be obviously computable?

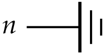
- The zero:



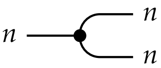
- The successor:



- The discarding:

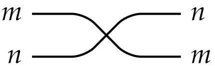


- The duplication:



- The identity:

- The swap:



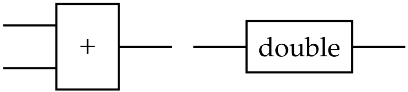
Composition

Boxes can be composed both serially and parallelly.

Composition

Boxes can be composed both serially and parallelly.

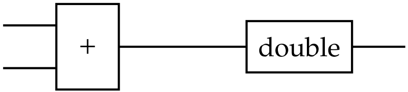
E.g.,

- 

Composition

Boxes can be composed both serially and parallelly.

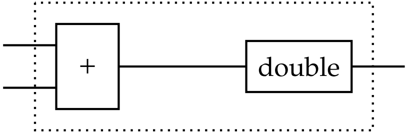
E.g.,

- 

Composition

Boxes can be composed both serially and parallelly.

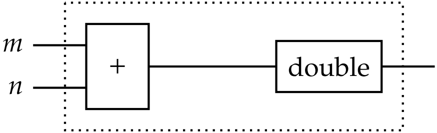
E.g.,

- 

Composition

Boxes can be composed both serially and parallelly.

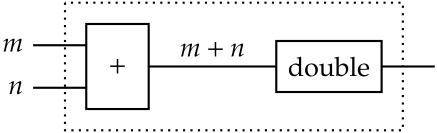
E.g.,

- 

Composition

Boxes can be composed both serially and parallelly.

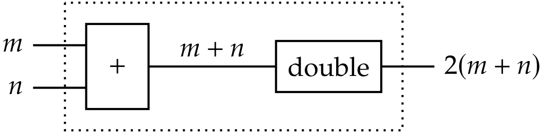
E.g.,

- 

Composition

Boxes can be composed both serially and parallelly.

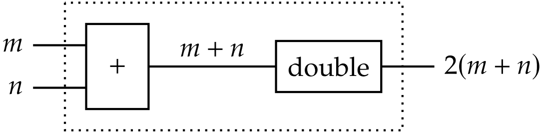
E.g.,

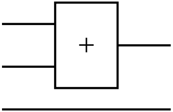
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

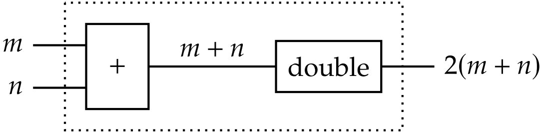
- 

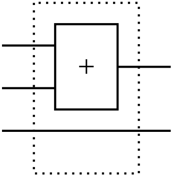
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

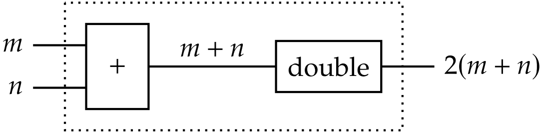
- 

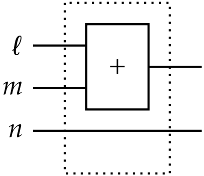
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

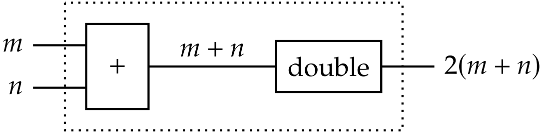
- 

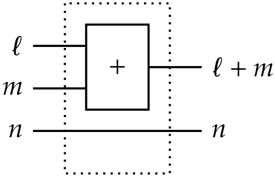
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

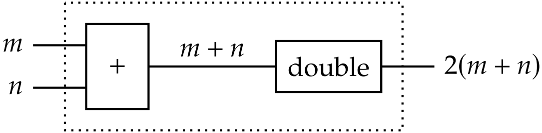
- 

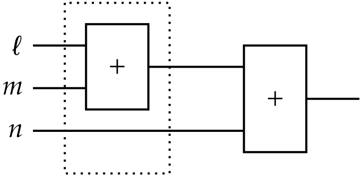
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

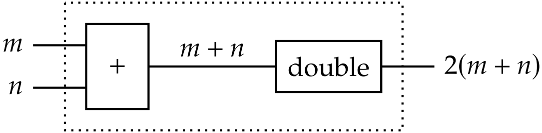
- 

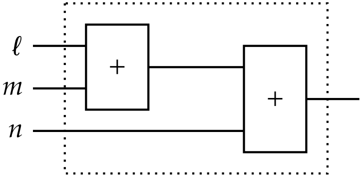
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

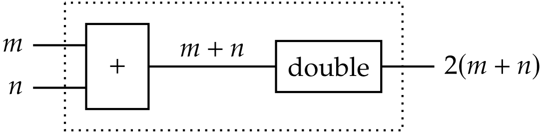
- 

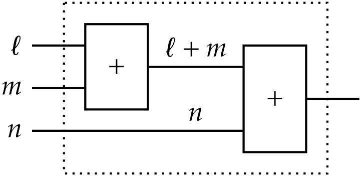
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

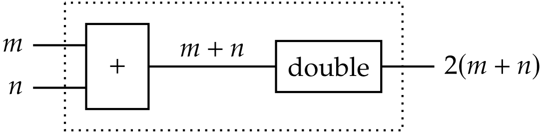
- 

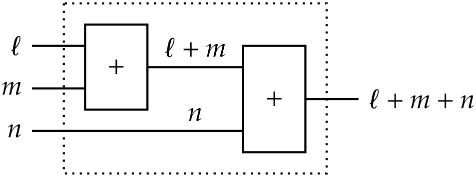
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

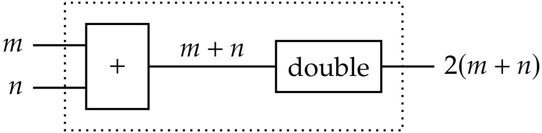
- 

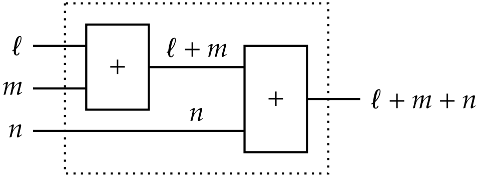
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

- 

- 

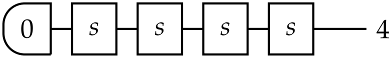
If parts are computable so is their composition!

E.g.,

- The number 4 (i.e. the function that takes no input and outputs 4) is computable because it can be built as

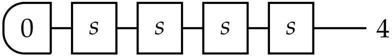
E.g.,

- The number 4 (i.e. the function that takes no input and outputs 4) is computable because it can be built as



E.g.,

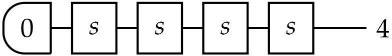
- The number 4 (i.e. the function that takes no input and outputs 4) is computable because it can be built as



Indeed, every natural number (as a function) is computable.

E.g.,

- The number 4 (i.e. the function that takes no input and outputs 4) is computable because it can be built as

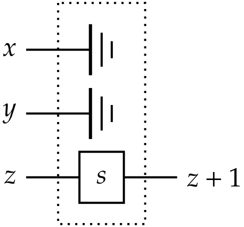


  Indeed, every natural number (as a function) is computable.

- The function

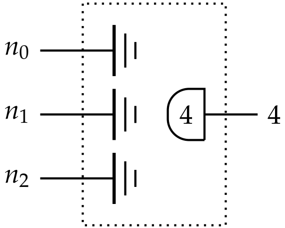
$$g : (x, y, z) \mapsto z + 1$$

  is computable because it can be built as

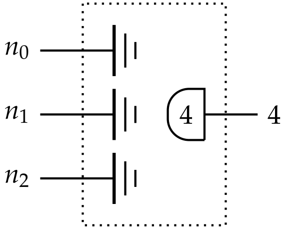


- The constant function $4 : \mathbb{N}^3 \longrightarrow \mathbb{N} :: (n_0, \ldots, n_2) \mapsto 4$ is computable because it can be built as

- The constant function $4 : \mathbb{N}^3 \to \mathbb{N} :: (n_0, \ldots, n_2) \mapsto 4$ is computable because it can be built as



- The constant function $4 : \mathbb{N}^3 \to \mathbb{N} :: (n_0, \ldots, n_2) \mapsto 4$ is computable because it can be built as



Indeed, every constant function $n : \mathbb{N}^k \to \mathbb{N} :: (n_0, \ldots, n_{k-1}) \mapsto n$ is computable.

So how powerful is composition? Can it create a lot of functions?

So how powerful is composition? Can it create a lot of functions?

— No, not really. It cannot even create $+$ or $\times$.

So how powerful is composition? Can it create a lot of functions?

— No, not really. It cannot even create $+$ or $\times$.

Our next order of business is to introduce another way of constructing new functions, and to use it to create $+$ or $\times$.