PHIL 222
Philosophical Foundations of Computer Science
Week 3, Thursday

Sept. 12, 2024

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:

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

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:

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

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:

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

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

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

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).

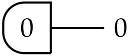
The Basic Six

So what functions would be obviously computable?

The Basic Six

So what functions would be obviously computable?

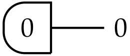
- The zero:



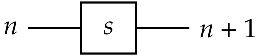
The Basic Six

So what functions would be obviously computable?

- The zero:



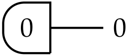
- The successor:



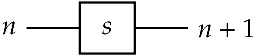
The Basic Six

So what functions would be obviously computable?

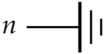
- The zero:



- The successor:



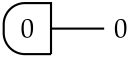
- The discarding:



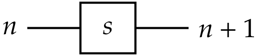
The Basic Six

So what functions would be obviously computable?

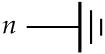
- The zero:



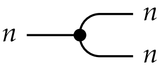
- The successor:



- The discarding:



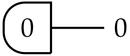
- The duplication:



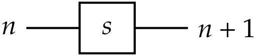
The Basic Six

So what functions would be obviously computable?

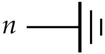
- The zero:



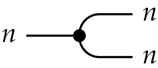
- The successor:



- The discarding:



- The duplication:

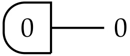


- The identity:

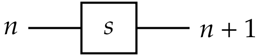
The Basic Six

So what functions would be obviously computable?

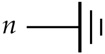
- The zero:



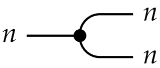
- The successor:



- The discarding:

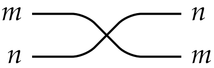


- The duplication:



- The identity:

- The swap:



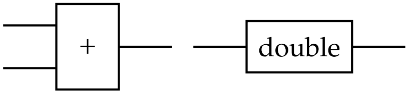
Composition

Boxes can be composed both serially and parallelly.

Composition

Boxes can be composed both serially and parallelly.

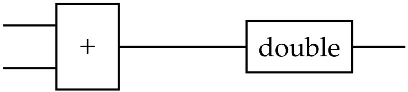
E.g.,

- 

Composition

Boxes can be composed both serially and parallelly.

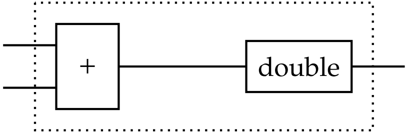
E.g.,

- 

Composition

Boxes can be composed both serially and parallelly.

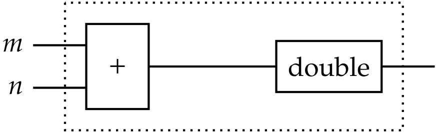
E.g.,

- 

Composition

Boxes can be composed both serially and parallelly.

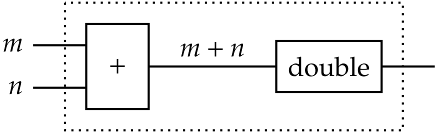
E.g.,

- 

Composition

Boxes can be composed both serially and parallelly.

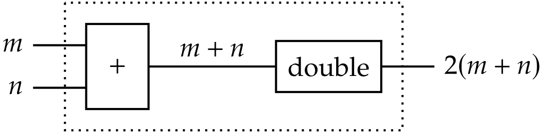
E.g.,

- 

Composition

Boxes can be composed both serially and parallelly.

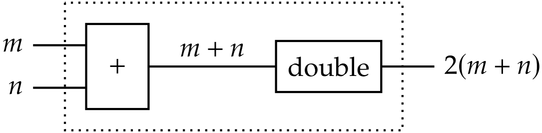
E.g.,

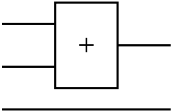
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

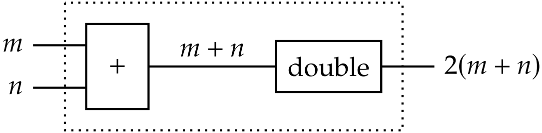
- 

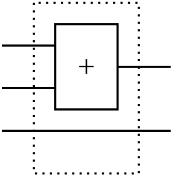
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

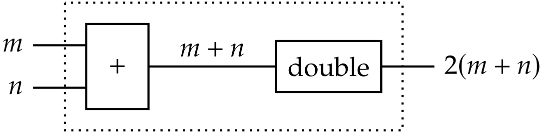
- 

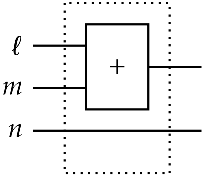
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

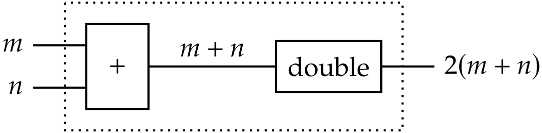
- 

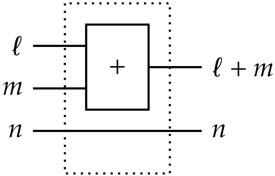
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

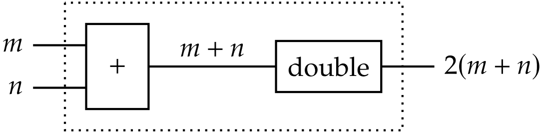
- 

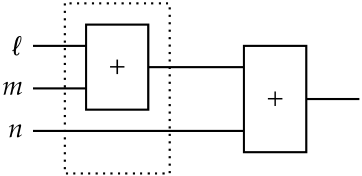
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

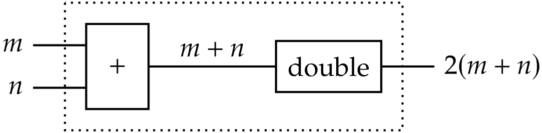
- 

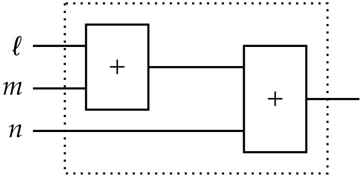
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

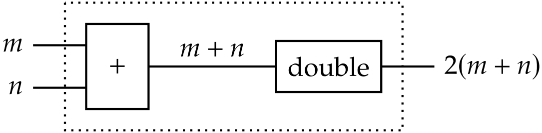
- 

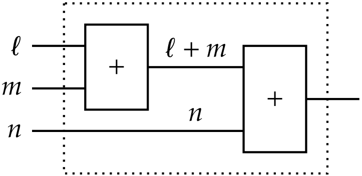
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

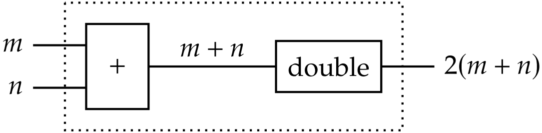
- 

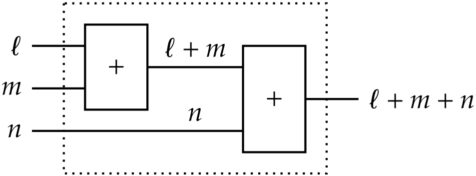
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

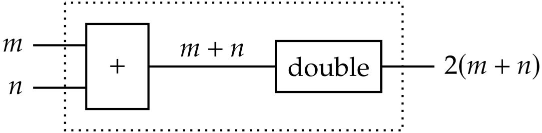
- 

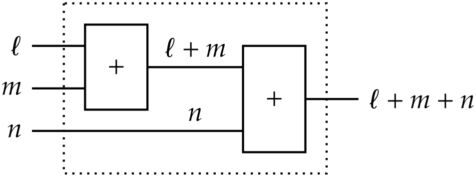
- 

Composition

Boxes can be composed both serially and parallelly.

E.g.,

- 

- 

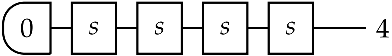
If parts are computable so is their composition!

E.g.,

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

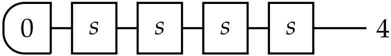
E.g.,

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



E.g.,

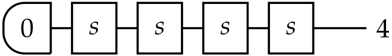
- The number 4 (i.e. the function that takes no input and outputs 4) is computable because it can be built as

  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.

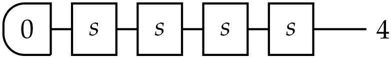
- The function

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

  is computable because it can be built as

E.g.,

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

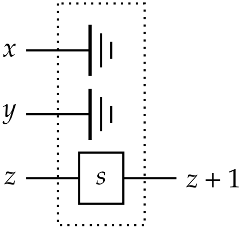


  Indeed, every natural number (as a function) is computable.

- The function

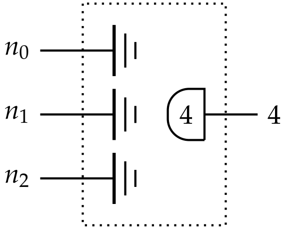
$$g : (x, y, z) \mapsto z + 1$$

  is computable because it can be built as

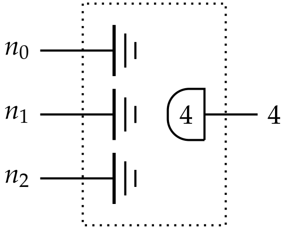


- The constant function $4 : \mathbb{N}^3 \longrightarrow \mathbb{N} :: (n_0, \ldots, n_2) \mapsto 4$ is computable because it can be built as

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



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

Recursive Functions:
Primitive Recursion

Recursion

Let's begin with the simplest case: computing

$$\mathrm{exp2} :: n \mapsto 2^n.$$

Recursion

Let's begin with the simplest case: computing

$$\text{exp2} :: n \mapsto 2^n.$$

$$2^0 = 1$$

Recursion

Let's begin with the simplest case: computing

$$\text{exp2} :: n \mapsto 2^n.$$

$$\begin{aligned} 2^0 &= & 1 \\ 2^1 &= 2^0 \times 2 = 1 \times 2 = & 2 \end{aligned}$$

Recursion

Let's begin with the simplest case: computing

$$\text{exp2} :: n \mapsto 2^n.$$

$$\begin{aligned} 2^0 &= & 1 \\ 2^1 &= 2^0 \times 2 = 1 \times 2 = & 2 \\ 2^2 &= 2^1 \times 2 = 2 \times 2 = & 4 \end{aligned}$$

Recursion

Let's begin with the simplest case: computing

$$\text{exp2} :: n \mapsto 2^n.$$

$$\begin{aligned}
2^0 &= & 1 \\
2^1 &= 2^0 \times 2 = 1 \times 2 = & 2 \\
2^2 &= 2^1 \times 2 = 2 \times 2 = & 4 \\
2^3 &= 2^2 \times 2 = 4 \times 2 = & 8
\end{aligned}$$

Recursion

Let's begin with the simplest case: computing

$$\text{exp2} :: n \mapsto 2^n.$$

$$\begin{aligned}
2^0 &= & & & 1 \\
2^1 &= 2^0 \times 2 &= 1 \times 2 &= 2 \\
2^2 &= 2^1 \times 2 &= 2 \times 2 &= 4 \\
2^3 &= 2^2 \times 2 &= 4 \times 2 &= 8
\end{aligned}$$

This is to compute exp2(n) by

- **base step**: start with 1, and
- **inductive step**: apply "double" n times to the previous value.

Recursion

Let's begin with the simplest case: computing

$$\text{exp2} :: n \mapsto 2^n.$$

$$\begin{aligned} 2^0 &= & & & 1 \\ 2^1 &= 2^0 \times 2 &= 1 \times 2 &= 2 \\ 2^2 &= 2^1 \times 2 &= 2 \times 2 &= 4 \\ 2^3 &= 2^2 \times 2 &= 4 \times 2 &= 8 \end{aligned}$$

This is to compute exp2(n) by

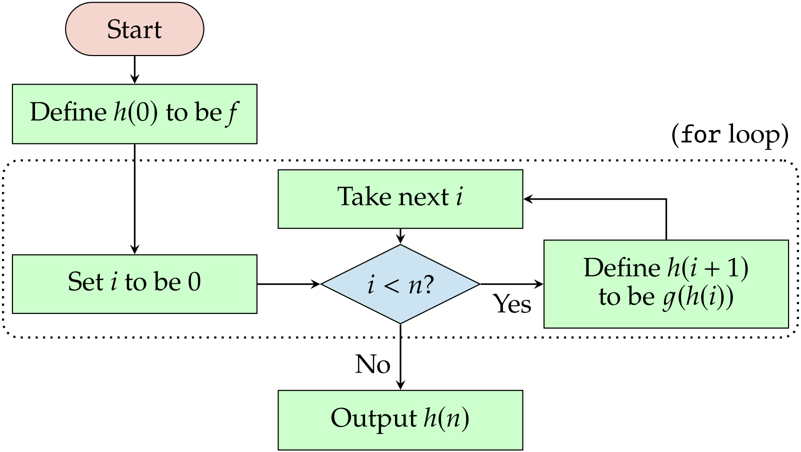
- **base step**: start with 1, and
- **inductive step**: apply "double" n times to the previous value.

We can regard this as the following program.

```
exp2(0) := 1
for i in (0, ..., n-1):
    exp2(i+1) := double(exp2(i))
return exp2(n)
```

```
h(0) := f
for i in (0, ..., n-1):
    h(i+1) := g(h(i))
return h(n)
```

```
h(0) := f
for i in (0, ..., n-1):
    h(i+1) := g(h(i))
return h(n)
```



A slightly more complicated case:

$$\text{factorial} :: n \mapsto n! = 1 \times 2 \times 3 \times \cdots \times n.$$

A slightly more complicated case:

$$\text{factorial} :: n \mapsto n! = 1 \times 2 \times 3 \times \cdots \times n.$$

$$0! = \qquad\qquad 1$$

A slightly more complicated case:

$$\text{factorial} :: n \mapsto n! = 1 \times 2 \times 3 \times \cdots \times n.$$

$$\begin{aligned} 0! &= && 1 \\ 1! &= 0! \times 1 = 1 \times 1 &&= 1 \end{aligned}$$

A slightly more complicated case:

$$\text{factorial} :: n \mapsto n! = 1 \times 2 \times 3 \times \cdots \times n.$$

$$\begin{aligned} 0! &= && && 1 \\ 1! &= 0! \times 1 &&= 1 \times 1 &&= 1 \\ 2! &= 1! \times 2 &&= 1 \times 2 &&= 2 \end{aligned}$$

A slightly more complicated case:

$$\text{factorial} :: n \mapsto n! = 1 \times 2 \times 3 \times \cdots \times n.$$

$$\begin{aligned} 0! &= & & & 1 \\ 1! &= 0! \times 1 &= 1 \times 1 &= 1 \\ 2! &= 1! \times 2 &= 1 \times 2 &= 2 \\ 3! &= 2! \times 3 &= 2 \times 3 &= 6 \end{aligned}$$

A slightly more complicated case:

$$\text{factorial} :: n \mapsto n! = 1 \times 2 \times 3 \times \cdots \times n.$$

$$\begin{aligned}
0! &= & & & 1 \\
1! &= 0! \times 1 &= 1 \times 1 &= 1 \\
2! &= 1! \times 2 &= 1 \times 2 &= 2 \\
3! &= 2! \times 3 &= 2 \times 3 &= 6 \\
4! &= 3! \times 4 &= 6 \times 4 &= 24
\end{aligned}$$

A slightly more complicated case:

$$\text{factorial} :: n \mapsto n! = 1 \times 2 \times 3 \times \cdots \times n.$$

$$\begin{aligned} 0! &= & 1 \\ 1! &= 0! \times 1 = 1 \times 1 = 1 \\ 2! &= 1! \times 2 = 1 \times 2 = 2 \\ 3! &= 2! \times 3 = 2 \times 3 = 6 \\ 4! &= 3! \times 4 = 6 \times 4 = 24 \end{aligned}$$

- **base step**: start with 1, and
- **inductive step**: in the ith iteration, apply the function $- \times (i+1)$ to the previous value $i!$ — iterate this n times.

New: for each $i < n$, the ith iteration of the ind. step also depends on i.

A slightly more complicated case:

$$\text{factorial} :: n \mapsto n! = 1 \times 2 \times 3 \times \cdots \times n.$$

$$\begin{aligned} 0! &= && && 1 \\ 1! &= 0! \times 1 &&= 1 \times 1 &&= 1 \\ 2! &= 1! \times 2 &&= 1 \times 2 &&= 2 \\ 3! &= 2! \times 3 &&= 2 \times 3 &&= 6 \\ 4! &= 3! \times 4 &&= 6 \times 4 &&= 24 \end{aligned}$$

- **base step**: start with 1, and
- **inductive step**: in the ith iteration, apply the function - $\times\, (i + 1)$ to the previous value $i!$ — iterate this n times.

New: for each $i < n$, the ith iteration of the ind. step also depends on i.

```
factorial(0) := 1
for i in (0, ..., n-1):
    factorial(i+1) := multiply(factorial(i), i+1)
return factorial(n)
```

A case with another type of complication:

$$\mathrm{add} :: (x, n) \mapsto x + n.$$

A case with another type of complication:

$$\text{add} :: (x, n) \mapsto x + n.$$

Let's take $x = 10$ for instance.

A case with another type of complication:

$$\text{add} :: (x, n) \mapsto x + n.$$

Let's take $x = 10$ for instance.

$$10 + 0 = \qquad 10$$

A case with another type of complication:

$$\text{add} :: (x, n) \mapsto x + n.$$

Let's take $x = 10$ for instance.

$$\begin{aligned} 10 + 0 &= & 10 \\ 10 + 1 &= (10 + 0) + 1 = 10 + 1 = & 11 \end{aligned}$$

A case with another type of complication:

$$\text{add} :: (x, n) \mapsto x + n.$$

Let's take $x = 10$ for instance.

$$\begin{aligned} 10 + 0 &= && 10 \\ 10 + 1 &= (10 + 0) + 1 = 10 + 1 &&= 11 \\ 10 + 2 &= (10 + 1) + 1 = 11 + 1 &&= 12 \end{aligned}$$

A case with another type of complication:

$$\text{add} :: (x, n) \mapsto x + n.$$

Let's take $x = 10$ for instance.

$$\begin{aligned}
10 + 0 &= & 10 \\
10 + 1 &= (10 + 0) + 1 = 10 + 1 = & 11 \\
10 + 2 &= (10 + 1) + 1 = 11 + 1 = & 12 \\
10 + 3 &= (10 + 2) + 1 = 12 + 1 = & 13
\end{aligned}$$

A case with another type of complication:

$$\text{add} :: (x, n) \mapsto x + n.$$

Let's take $x = 10$ for instance.

$$\begin{aligned} 10 + 0 &= && 10 \\ 10 + 1 &= (10 + 0) + 1 = 10 + 1 &&= 11 \\ 10 + 2 &= (10 + 1) + 1 = 11 + 1 &&= 12 \\ 10 + 3 &= (10 + 2) + 1 = 12 + 1 &&= 13 \end{aligned}$$

- **base step**: start with x, and
- **inductive step**: in the ith iteration, apply the successor function to the previous value $\text{add}(x, i)$ — iterate this n times.

New: add takes a parameter x as an argument in addition to n.

A case with another type of complication:

$$\text{add} :: (x, n) \mapsto x + n.$$

Let's take $x = 10$ for instance.

$$\begin{aligned} 10 + 0 &= & 10 \\ 10 + 1 &= (10 + 0) + 1 = 10 + 1 = & 11 \\ 10 + 2 &= (10 + 1) + 1 = 11 + 1 = & 12 \\ 10 + 3 &= (10 + 2) + 1 = 12 + 1 = & 13 \end{aligned}$$

- **base step**: start with x, and
- **inductive step**: in the ith iteration, apply the successor function to the previous value add(x, i) — iterate this n times.

New: add takes a parameter x as an argument in addition to n.

```
add(x, 0) := x
for i in (0, ..., n-1):
    add(x, i+1) := add(x, i)+1
return add(x, n)
```

$$\exp :: (x, n) \mapsto x^n.$$

Again let's take $x = 10$ for instance.

$$\exp :: (x, n) \mapsto x^n.$$

Again let's take $x = 10$ for instance.

$$10^0 = \quad 1$$

$$\exp :: (x, n) \mapsto x^n.$$

Again let's take $x = 10$ for instance.

$$\begin{aligned} 10^0 &= & & & 1 \\ 10^1 &= 10^0 \times 10 = & & 1 \times 10 = & 10 \end{aligned}$$

$$\exp :: (x, n) \mapsto x^n.$$

Again let's take $x = 10$ for instance.

$$\begin{aligned} 10^0 &= & & & 1 \\ 10^1 &= 10^0 \times 10 &=& \quad 1 \times 10 &= 10 \\ 10^2 &= 10^1 \times 10 &=& \; 10 \times 10 &= 100 \end{aligned}$$

$$\exp :: (x, n) \mapsto x^n.$$

Again let's take $x = 10$ for instance.

$$\begin{aligned}
10^0 &= & & & 1 \\
10^1 &= 10^0 \times 10 &= & 1 \times 10 &= 10 \\
10^2 &= 10^1 \times 10 &= & 10 \times 10 &= 100 \\
10^3 &= 10^2 \times 10 &= & 100 \times 10 &= 1000
\end{aligned}$$

$$\exp :: (x, n) \mapsto x^n.$$

Again let's take $x = 10$ for instance.

$$\begin{aligned}
10^0 &= & & 1 \\
10^1 &= 10^0 \times 10 = & 1 \times 10 &= 10 \\
10^2 &= 10^1 \times 10 = & 10 \times 10 &= 100 \\
10^3 &= 10^2 \times 10 = & 100 \times 10 &= 1000
\end{aligned}$$

- **base step**: start with 1, and
- **inductive step**: in the ith iteration, apply the function $- \times x$ to the previous value x^i — iterate this n times.

New: each inductive step depends on the parameter x.

$$\exp :: (x, n) \mapsto x^n.$$

Again let's take $x = 10$ for instance.

$$\begin{aligned}
10^0 &= & & 1 \\
10^1 &= 10^0 \times 10 = & 1 \times 10 &= 10 \\
10^2 &= 10^1 \times 10 = & 10 \times 10 &= 100 \\
10^3 &= 10^2 \times 10 = & 100 \times 10 &= 1000
\end{aligned}$$

- **base step**: start with 1, and
- **inductive step**: in the ith iteration, apply the function $- \times x$ to the previous value x^i — iterate this n times.

New: each inductive step depends on the parameter x.

```
exp(x, 0) := 1
for i in (0, ..., n-1):
    exp(x, i+1) := multiply(exp(x, i), x)
return exp(x, n)
```

All the cases so far can be unified by

```
h(x, y, z, 0) := f(x, y, z)
for i in (0, ..., n-1):
    h(x, y, z, i+1) := g(x, y, z, i, h(x, y, z, i))
```

All the cases so far can be unified by

```
h(x, y, z, 0) := f(x, y, z)
for i in (0, ..., n-1):
    h(x, y, z, i+1) := g(x, y, z, i, h(x, y, z, i))
```

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

All the cases so far can be unified by

```
h(x, y, z, 0) := f(x, y, z)
for i in (0, ..., n-1):
    h(x, y, z, i+1) := g(x, y, z, i, h(x, y, z, i))
```

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

If f and g are computable so is h!

All the cases so far can be unified by

```
h(x, y, z, 0) := f(x, y, z)
for i in (0, ..., n-1):
    h(x, y, z, i+1) := g(x, y, z, i, h(x, y, z, i))
```

Definition. We say that a function $h(\bar{x}, n)$ is defined from two other functions $f(\bar{x})$ and $g(\bar{x}, n, k)$ by "primitive recursion" if it satisfies

$$h(\bar{x}, 0) = f(\bar{x}), \qquad h(\bar{x}, s(i)) = g(\bar{x}, i, h(\bar{x}, i)).$$

If f and g are computable so is h!

N.B. Keep track of the numbers of inputs / arguments:

- if f takes n inputs, g takes $n + 2$ inputs, and h takes $n + 1$ inputs.