Recursion in Mathematics and Programming

Summation

Compute summation

$$\sum_{k=1}^{n} = 1 + 2 + \ldots + n.$$

It's recursive definition:

$$\begin{aligned} \textbf{sum}(0) &= 0 \\ \textbf{sum}(n) &= \textbf{sum}(n-1) + n \qquad \text{(for } n \geq 1) \end{aligned}$$

Source code "`sum.jl`".

Summation

Evaluation of the expression $\mathbf{sum}(4)$

$$
\begin{aligned}
\mathbf{sum}(4) \quad &\Rightarrow \quad \mathbf{sum}(3) + 4 \\
&\Rightarrow \quad (\mathbf{sum}(2) + 3) + 4 \\
&\Rightarrow \quad ((\mathbf{sum}(1) + 2) + 3) + 4 \\
&\Rightarrow \quad (((\mathbf{sum}(0) + 1) + 2) + 3) + 4 \\
&\Rightarrow \quad (((0 + 1) + 2) + 3 + 4) \\
&\Rightarrow \quad ((1 + 2) + 3) + 4 \\
&\Rightarrow \quad (3 + 3) + 4 \\
&\Rightarrow \quad 6 + 4 \\
&\Rightarrow \quad 10
\end{aligned}
$$

Summation

$$\begin{array}{c} \mathbf{sum}(4) \\ | \\ \mathbf{sum}(3) \\ | \\ \mathbf{sum}(2) \\ | \\ \mathbf{sum}(1) \\ | \\ \mathbf{sum}(0) \end{array}$$

To compute $\mathbf{sum}(4)$, the function **sum** was called 5 times.

And to compute $\mathbf{sum}(n)$, the function **sum** will be called $n+1$ times.

Fibonacci numbers

Recursive definition of the Fibonacci numbers:

$$\begin{aligned}\mathbf{fib}(0) &= 0\\ \mathbf{fib}(1) &= 1\\ \mathbf{fib}(n) &= \mathbf{fib}(n-1) + \mathbf{fib}(n-2) \qquad (\text{for } n \geq 2)\end{aligned}$$

Source code "*fib.jl*".

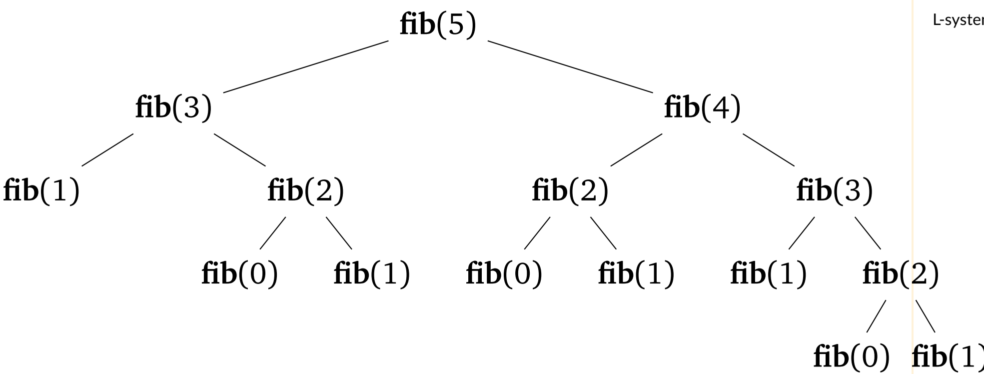
Fibonacci numbers

Evaluation of the expression $\mathbf{fib}(5)$

$$
\begin{aligned}
\mathbf{fib}(5) \quad &\Rightarrow \quad \mathbf{fib}(3) + \mathbf{fib}(4) \\
&\Rightarrow \quad (\mathbf{fib}(1) + \mathbf{fib}(2)) + (\mathbf{fib}(2) + \mathbf{fib}(3)) \\
&\Rightarrow \quad (1 + (\mathbf{fib}(0) + \mathbf{fib}(1))) + ((\mathbf{fib}(0) + \mathbf{fib}(1)) + (\mathbf{fib}(1) + \mathbf{fib}(2))) \\
&\Rightarrow \quad (1 + (0 + 1)) + ((0 + 1) + (1 + (\mathbf{fib}(0) + \mathbf{fib}(1)))) \\
&\Rightarrow \quad (1 + 1) + (1 + (1 + (0 + 1))) \\
&\Rightarrow \quad 2 + (1 + (1 + 1)) \\
&\Rightarrow \quad 2 + (1 + 2) \\
&\Rightarrow \quad 2 + 3 \\
&\Rightarrow \quad 5
\end{aligned}
$$

How many times did we call function **fib** to compute $\mathbf{fib}(5)$?

Fibonacci numbers



To compute **fib**(n), we need to do approximately 2^n function calls.

Exponential running time, $O(2^n)$, too slow to be practical.

Improved recursive Fibonacci

Evaluation of the expression **fib_loop_rec**(5)

$$
\begin{aligned}
\textbf{fib\_loop\_rec}(5) &\Rightarrow \textbf{loop}(1,0,1)\\
&\Rightarrow \textbf{loop}(2,1,1)\\
&\Rightarrow \textbf{loop}(3,1,2)\\
&\Rightarrow \textbf{loop}(4,2,3)\\
&\Rightarrow \textbf{loop}(5,3,5)\\
&\Rightarrow 5
\end{aligned}
$$

Just $n+1$ function calls to compute **fib_loop_rec**(n).

This is a linear time $O(n)$ algorithm, we get a real improvement!

A simple calculator language

We want to describe a computer that can do the following:

Input		Output
55	$\Rightarrow$	55
$(\ 1\ +\ 2\ )$	$\Rightarrow$	3
$(\ 12\ +\ (\ (\ 5\ \times\ 2\ )\ \times\ 7\ )\ )\ )$	$\Rightarrow$	82

A simple calculator language

$$100$$

$$(\ 5 + (\ 74 - 15\ )\ )$$

$$(\ (\ 12 + 5\ ) \times (\ 5 - 7\ )\ )\ )$$

The syntax of the language:

$$
\begin{array}{lcl}
<expr> & ::= & \textit{strings of digits}\text{, represent integer numbers} \\
 & | & (\ <expr> + <expr>\ ) \\
 & | & (\ <expr> - <expr>\ ) \\
 & | & (\ <expr> \times <expr>\ )
\end{array}
$$

A simple calculator language

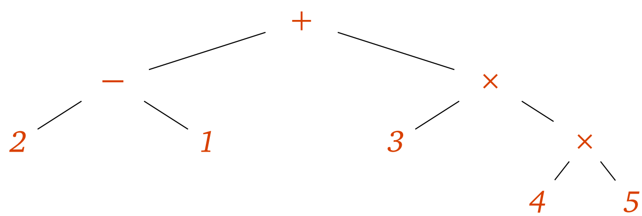
Simple examples

Evaluating expressions

Recursion in Mathematics

L-systems

$$(\ (\ 2 - 1 \ ) \ + \ (\ 3 \times \ (\ 4 \times 5 \ ) \ ) \ )$$



The evaluating function looks at the root of the tree only and desides what to do

$$
\begin{aligned}
& eval\big[\ (\ E_1 \ + \ E_2 \ ) \ \big] \\
\Rightarrow \quad & eval\big[E_1\big] + eval\big[E_2\big]
\end{aligned}
$$

A simple calculator language

Evaluation function:

$$
\begin{aligned}
eval\big[\ (\ E_1\ +\ E_2\ )\ \big] &= eval\big[E_1\big] + eval\big[E_2\big] \\
eval\big[\ (\ E_1\ -\ E_2\ )\ \big] &= eval\big[E_1\big] - eval\big[E_2\big] \\
eval\big[\ (\ E_1\ \times\ E_2\ )\ \big] &= eval\big[E_1\big] \times eval\big[E_2\big] \\
eval\big[n\big] &= n
\end{aligned}
$$

A simple calculator language

$$
\begin{aligned}
& eval[\ (\ \ (\ 2 - 1\ )\ \ +\ \ (\ 3 \times\ \ (\ 4 \times 5\ )\ \ )\ \ )\ ] \\
\Rightarrow\ & eval[\ (\ 2 - 1\ )\ ]\ \ +\ \ eval[\ (\ 3 \times\ \ (\ 4 \times 5\ )\ \ )\ ] \\
\Rightarrow\ & (eval[2] - eval[1]) + (eval[3] \times eval[\ (\ 4 \times 5\ )\ ]) \\
\Rightarrow\ & (2 - 1) + (3 \times (eval[4] \times eval[5])) \\
\Rightarrow\ & 1 + (3 \times (4 \times 5)) \\
\Rightarrow\ & 1 + (3 \times 20) \\
\Rightarrow\ & 1 + 60 \\
\Rightarrow\ & 61
\end{aligned}
$$

Recursion in Mathematics

Some prominent examples of recursion

- *Euclid's algorithm*
  a algorithm for computing the greatest common divisor.
- *Newton's method*
  a recursive method for finding successively better approximations to the roots of a real-valued function.
- Complex objects such as fractals can be defined using recursive definition.

Consider a recurrence

Given a number c,

$$z_0 = 0$$
$$z_n = z_{n-1}^2 + c$$

What sequences of numbers can be generated with this recurrence? The result, of course, depends on the value of c.

Consider a recurrence

$$z_0 = 0$$
$$z_n = z_{n-1}^2 + c$$

Let us test different values for c.

$$\begin{aligned}
\text{when } c = 0 : &\quad 0 \to 0 \to 0 \to 0 \to 0 \to \cdots \\
\text{when } c = -1 : &\quad 0 \to (-1) \to 0 \to (-1) \to \cdots \\
\text{when } c = -2 : &\quad 0 \to (-2) \to 2 \to 2 \to 2 \to \cdots
\end{aligned}$$

$$\begin{aligned}
\text{when } c = 1 : &\quad 0 \to 1 \to 2 \to 5 \to 26 \to \cdots \infty \\
\text{when } c = -3 : &\quad 0 \to (-3) \to 6 \to 33 \to \cdots \infty
\end{aligned}$$

Define set M

$$
\begin{aligned}
\text{when } c = 0: &\quad 0 \to 0 \to 0 \to 0 \to 0 \to \cdots \\
\text{when } c = -1: &\quad 0 \to (-1) \to 0 \to (-1) \to \cdots \\
\text{when } c = -2: &\quad 0 \to (-2) \to 2 \to 2 \to 2 \to \cdots
\end{aligned}
$$

$$
\begin{aligned}
\text{when } c = 1: &\quad 0 \to 1 \to 2 \to 5 \to 26 \to \cdots \infty \\
\text{when } c = -3: &\quad 0 \to (-3) \to 6 \to 33 \to \cdots \infty
\end{aligned}
$$

We are interested in all c, such that the corresponding sequence z_n **does not go to infinity**.

Let's say that all such numbers c belong to the set M.

Define set M

$$
\begin{aligned}
\text{when } c = 0: &\quad 0 \to 0 \to 0 \to 0 \to 0 \to \cdots \\
\text{when } c = -1: &\quad 0 \to (-1) \to 0 \to (-1) \to \cdots \\
\text{when } c = -2: &\quad 0 \to (-2) \to 2 \to 2 \to 2 \to \cdots
\end{aligned}
$$

$$
\begin{aligned}
\text{when } c = 1: &\quad 0 \to 1 \to 2 \to 5 \to 26 \to \cdots \infty \\
\text{when } c = -3: &\quad 0 \to (-3) \to 6 \to 33 \to \cdots \infty
\end{aligned}
$$

We are interested in all c, such that the corresponding sequence z_n **does not go to infinity**.

Let's say that all such numbers c belong to the set M.

Numbers c and z_n are *complex numbers*. So, we need to quickly learn how to add and multiply them.

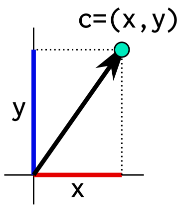
Quick intro to complex numbers

The set of complex numbers $\mathbb{C}$ is an extension of the set of real numbers $\mathbb{R}$.

Any complex number can be represented by a pair of real numbers

$$(x, y) \qquad x, y \in \mathbb{R}$$

x is the *real part*,
y is the *imaginary part*.



Alternative notation. We can represent the pair as a sum of its real and imaginary part

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

$i = \sqrt{-1}$ is the *imaginary unit*.

Quick intro to complex numbers

Addition

$$(a, b) + (c, d) = (a + c, b + d)$$

Multiplication

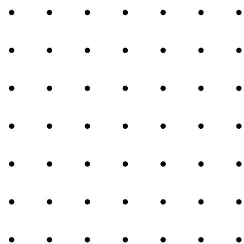
$$(a, b) \cdot (c, d) = (ac - bd, bc + ad)$$

Particularly,

$$i^2 = i \cdot i = (0, 1) \cdot (0, 1) = (0 - 1, 0 + 0) = (-1, 0) = -1$$

How to draw the set M?

It's impossible to check all numbers c. But we can take a rectangular grid of points around the origin, $(0,0)$. We are going to mark all those points that belong to the set M.



$z_n = (x_n, y_n)$ will go to infinity if one of its components get large, $\sqrt{x_n^2 + y_n^2} \geq 2$. This condition is used practically to compute the membership of c in the set.

Source code "`mset.jl`".

L-systems

An L-system or Lindenmayer system is a parallel *rewriting system*.

It consists of

- an alphabet of symbols
- an initial string (called an axiom) to start construction
- a collection of production rules that expand each symbol into some larger string of symbols

Example:

$$\begin{aligned} \textit{Alphabet:} \quad & \{A, B\} \\ \textit{Initial string:} \quad & A \\ \textit{Production rules:} \quad & A \longrightarrow AB \\ & B \longrightarrow A \end{aligned}$$

L-systems

Example:

Alphabet: $\{A, B\}$

Initial string: A

Production rules: $A \longrightarrow AB$

$B \longrightarrow A$

Start rewriting:

$$n = 0 : A$$
$$n = 1 : AB$$
$$n = 2 : ABA$$
$$n = 3 : ABAAB$$
$$n = 4 : ABAABABA$$
$$n = 5 : ABAABABAABAAB$$
$$n = 6 : ABAABABAABAABABAABABA$$

L-systems

Koch curve:

Alphabet:	$\{F, L, R\}$
Initial string:	F
Production rules:	$F \longrightarrow F\ L\ F\ R\ F\ R\ F\ L\ F$
	$L \longrightarrow L$
	$R \longrightarrow R$

When F stands for moving forward, and L and R are the commands for turning left and right

Source code "`lsys.jl`".