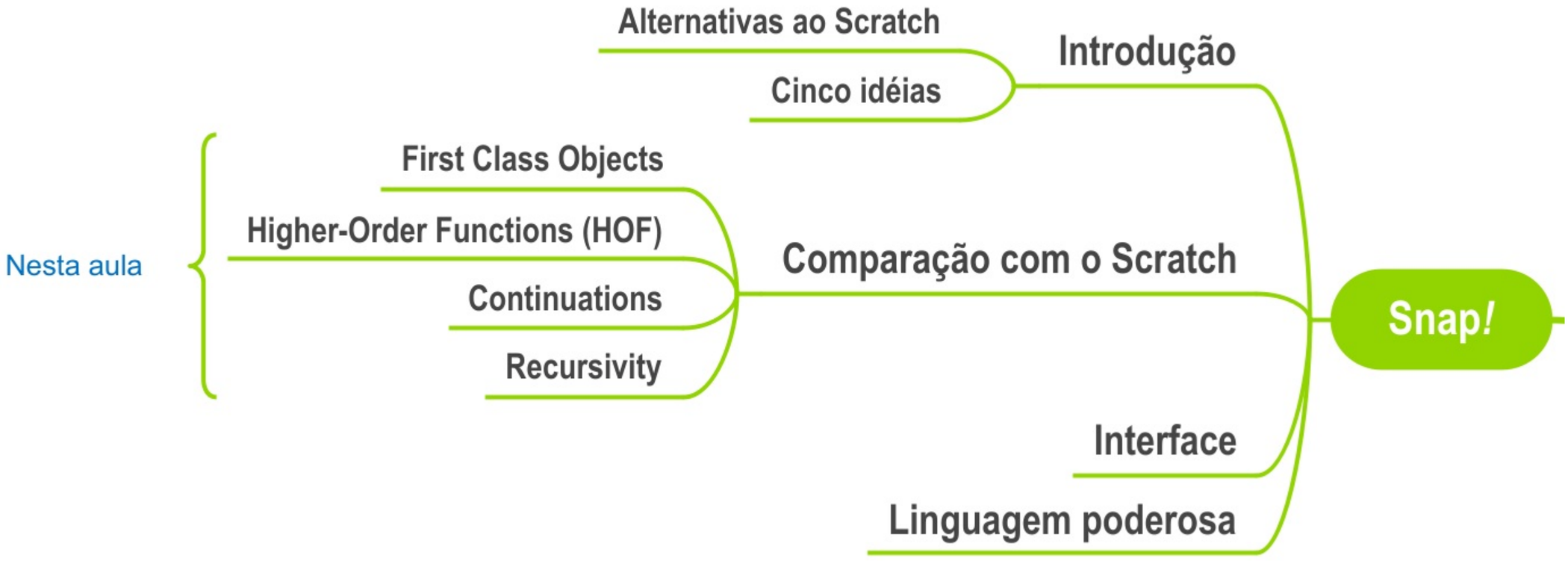
FUNDAMENTOS DA PROGRAMAÇÃO

Conceitos avançados



Conceitos avançados

Quando comparamos o Scratch com o Snap*!*, na aula passada, vimos que o Snap*!* tem funcionalidades muito avançadas que o Scratch não possui. Especialmente o Snap*!* apresenta:

- **First Class Objects (objetos de 1ª classe)**
- **Higher-Order Functions (funções de ordem superior)**
- **First Class Continuations (continuações de 1ª classe)**
- **Recursivity (recursividade)**
- **...**

Objetivo: apenas apresentar esses conceitos... não se preocupe se você não entender tudo agora!

Objetos de 1ª Classe

Foto: adaptada do seminário "Strachey 100", da Universidade de Oxford, para comemorar o centenário de nascimento de Christopher Strachey.
(https://www.cs.ox.ac.uk/strachey100/)

Christopher S. Strachey (1916 - 1975)

- Cientista da computação britânico
- Pioneiro no projeto de linguagens de programação
- Um dos fundadores da semântica denotacional
- Criador do conceito de "time-sharing"
- Criador da linguagem CPL (Combined PL)
- Formalização dos L e R values em expressões
- Pioneiro em vídeo games

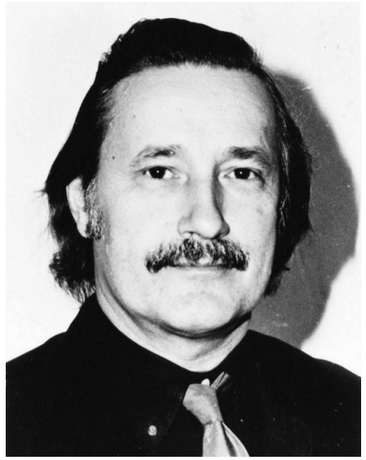
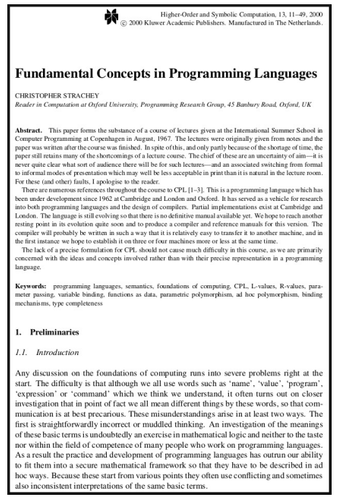


Foto: IEEE Computer Society: Computer Pioners
(https://history.computer.org/pioneers/strachey.html)



Higher-Order and Symbolic Computation, 13, 11–49, 2000
© 2000 Kluwer Academic Publishers. Manufactured in The Netherlands.

Fundamental Concepts in Programming Languages

CHRISTOPHER STRACHEY
Reader in Computation at Oxford University, Programming Research Group, 45 Banbury Road, Oxford, UK

Abstract. This paper forms the substance of a course of lectures given at the International Summer School in Computer Programming at Copenhagen in August, 1967. The lectures were originally given from notes and the paper was written after the course was finished. In spite of this, and only partly because of the shortage of time, the paper still retains many of the shortcomings of a lecture course. The chief of these are an uncertainty of aim—it is never quite clear what sort of audience there will be for such lectures—and an associated switching from formal to informal modes of presentation which may well be less acceptable in print than it is natural in the lecture room. For these (and other) faults, I apologise to the reader.

There are numerous references throughout the course to CPL [1–3]. This is a programming language which has been under development since 1962 at Cambridge and London and Oxford. It has served as a vehicle for research into both programming languages and the design of compilers. Partial implementations exist at Cambridge and London. The language is still evolving so that there is no definitive manual available yet. We hope to reach another resting point in its evolution quite soon and to produce a compiler and reference manuals for this version. The compiler will probably be written in such a way that it is relatively easy to transfer it to another machine, and in the first instance we hope to establish it on three or four machines more or less at the same time.

The lack of a precise formulation for CPL should not cause much difficulty in this course, as we are primarily concerned with the ideas and concepts involved rather than with their precise representation in a programming language.

Keywords: programming languages, semantics, foundations of computing, CPL, L-values, R-values, parameter passing, variable binding, functions as data, parametric polymorphism, ad hoc polymorphism, binding mechanisms, type completeness

1. Preliminaries

1.1. Introduction

Any discussion on the foundations of computing runs into severe problems right at the start. The difficulty is that although we all use words such as 'name', 'value', 'program', 'expression' or 'command' which we think we understand, it often turns out on closer investigation that in point of fact we all mean different things by these words, so that communication is at best precarious. These misunderstandings arise in at least two ways. The first is straightforwardly incorrect or muddled thinking. An investigation of the meanings of these basic terms is undoubtedly an exercise in mathematical logic and neither to the taste nor within the field of competence of many people who work on programming languages. As a result the practice and development of programming languages has outrun our ability to fit them into a secure mathematical framework so that they have to be described in ad hoc ways. Because these start from various points they often use conflicting and sometimes also inconsistent interpretations of the same basic terms.

Objetos de 1ª Classe

O que são objetos de 1ª classe? Para que um objeto qualquer em uma linguagem de programação seja considerado de 1ª classe, essas cinco propriedades obrigatoriamente devem estar presentes:

1) Pode ser o valor de uma variável

2) Pode ser elemento de um agregado (array, lista, ...)

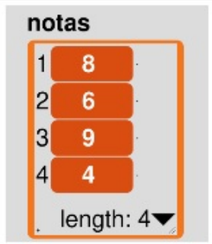
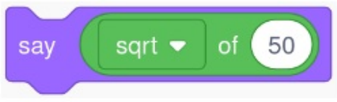
3) Pode existir sem um nome, pode ser anônimo

4) Pode ser argumento de um subprograma

5) Pode ser retorno de um subprograma

Foto: Javier Quiroga, no Unsplash
(https://unsplash.com/photos/red-and-white-led-signage-IuGDNXawS9M)

Números são objetos de 1ª classe em, praticamente, todas as linguagens

Propriedade	Snap!	Scratch
Valor de variável		
Elemento de agregado		
Usado sem nome		
Pode ser argumento		
Pode ser retorno		

Arrays são objetos de 1ª classe?

Depende da linguagem!

- Arrays são 1ª classe em: Snap!, Lisp, Scheme...

- Arrays são 2ª classe em: Scratch, C, ...

valor de variável

elemento de agregado

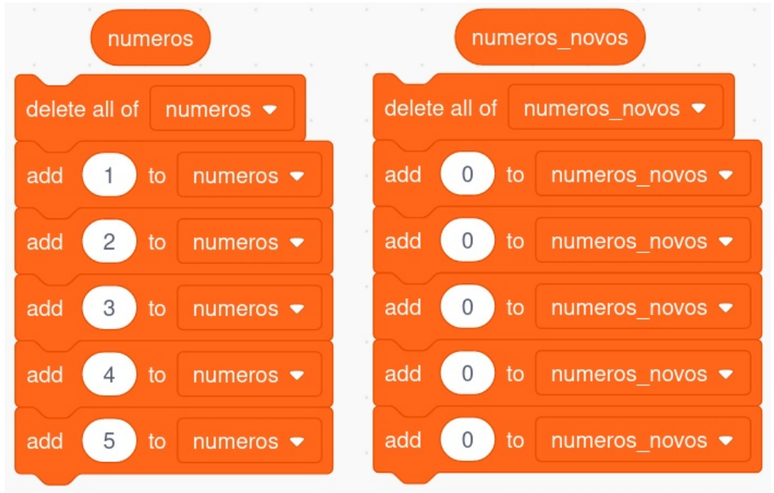
utilizado de forma anônima

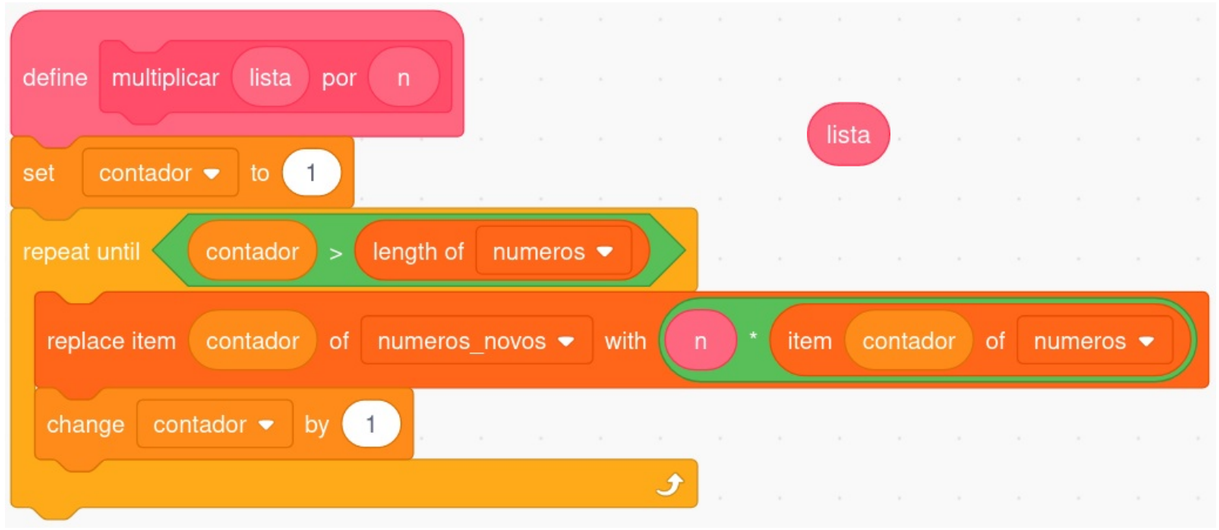
utilizado como argumento

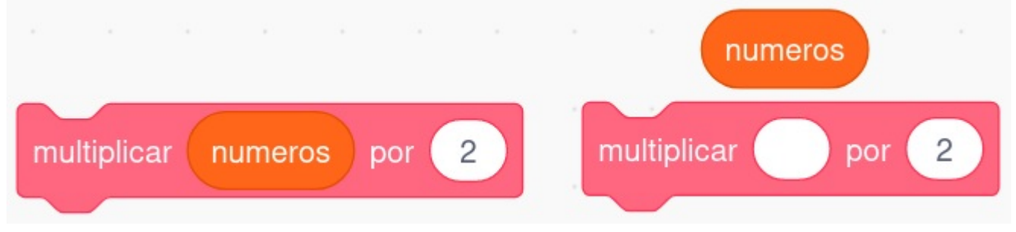
pode ser retornado

Arrays são objetos de 1ª classe?

Problema: criar uma função para multiplicar todos os elementos de um array por um número qualquer, e retornar os resultados em outro array. Versão Scratch:

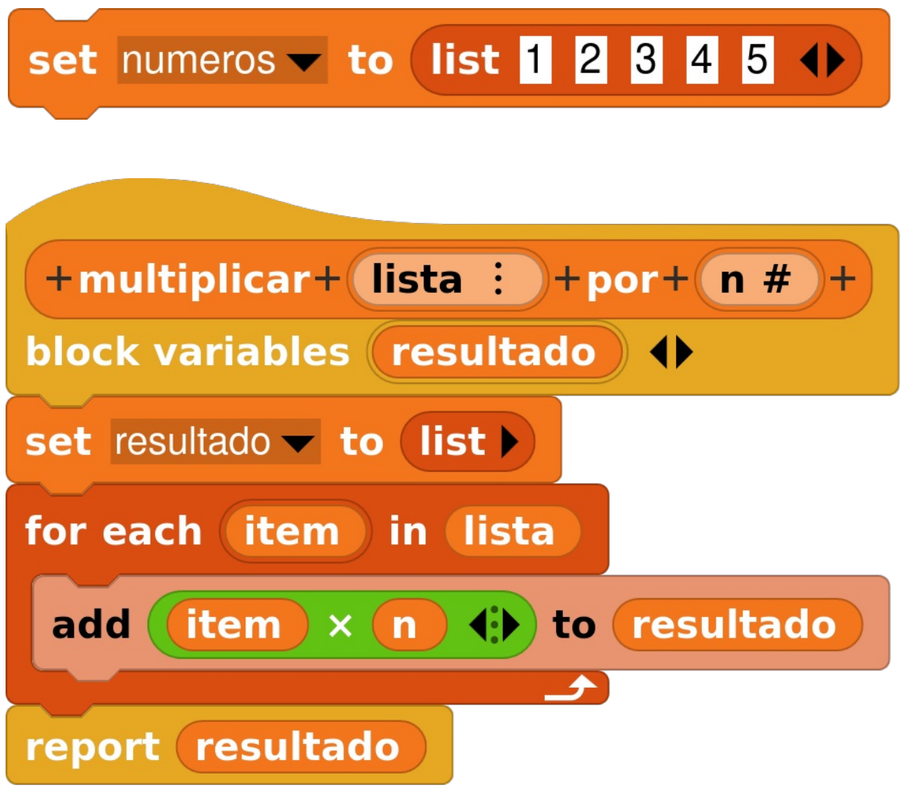


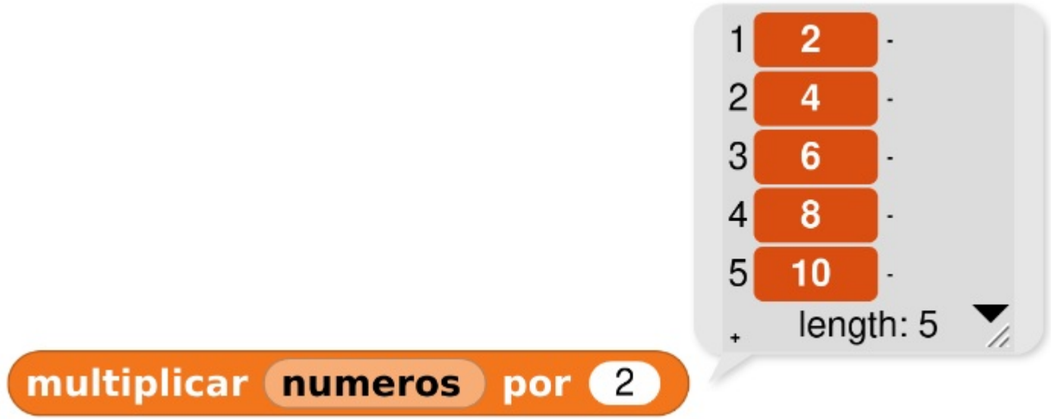


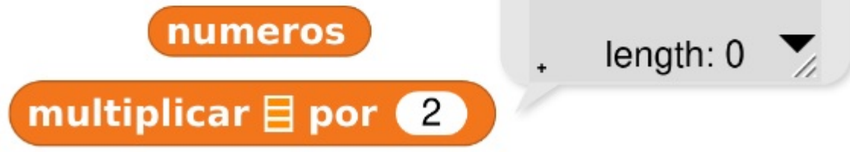


Arrays são objetos de 1ª classe?

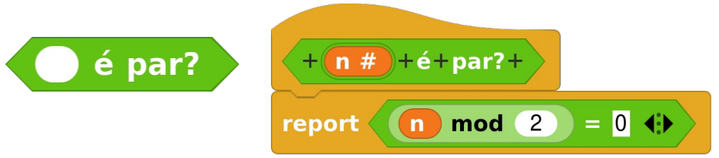
Problema: criar uma função para multiplicar todos os elementos de um array por um número qualquer, e retornar os resultados em outro array. Versão Snap!:







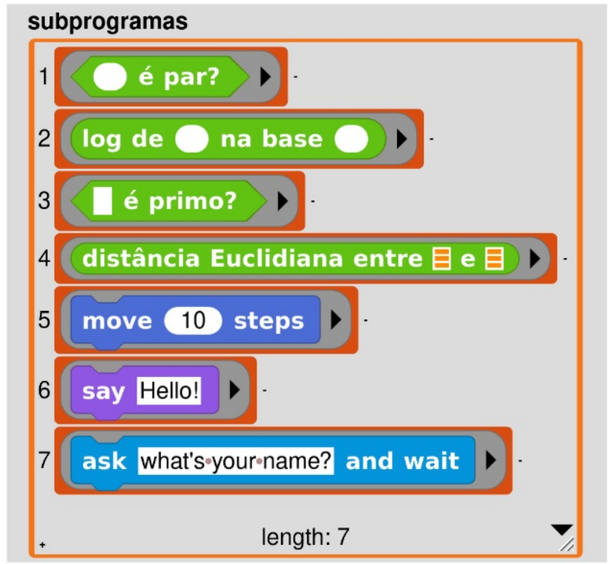
Subprogramas são objetos de 1ª classe?



Depende da linguagem!

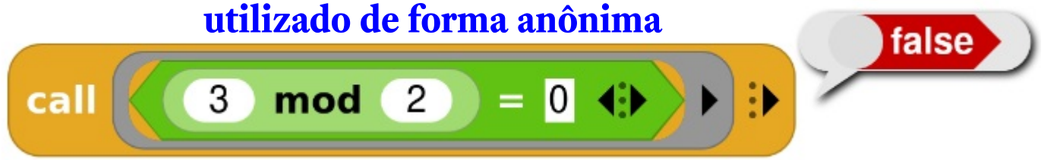
- São 1ª classe em: Snap!, Lisp, Scheme...
- Conceito difícil de entender inicialmente
- Fundamental para HOF!

valor de variável



elemento de agregado

utilizado de forma anônima



utilizado como argumento

pode ser retornado

Continuações são objetos de 1ª classe?

Depende da linguagem!

- São 1ª classe em: Snap*!*, Scheme, ...

- Conceito MUITO difícil de entender!

- Serão citadas brevemente, nesta aula.

No Snap*!* tudo é de primeira classe

Em geral, qualquer objeto em qualquer linguagem de programação deveria ser de 1ª classe. Na prática não é assim. Em Snap*!*: "Everything First Class".

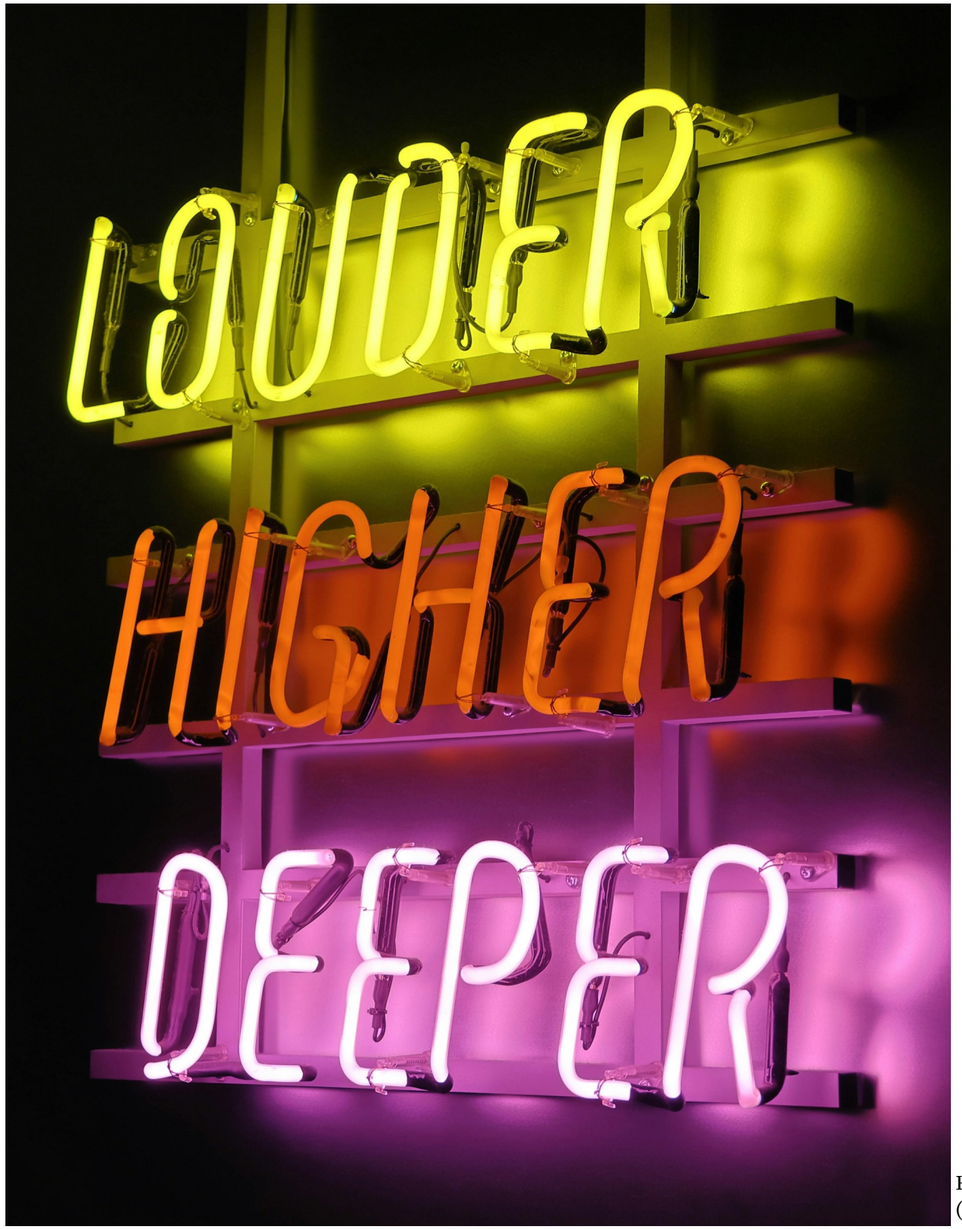
Por que é importante ter objetos de 1ª classe?

- **Arrays (listas): permitem criar qualquer outra estrutura de dados (árvores, heaps, tabelas hash, dicionários, ...)**
- **Subprogramas: permite criar novas estruturas de controle, criar HOF, objetos, ...**
- **Continuações: permitem criar mecanismos de controle (catch/throw, threads, ...)**

No Snap*!* também temos como objetos de 1ª classe:

- **Sprites: são objetos de 1ª classe com herança (permitem POO)**
- **Costumes: são objetos de 1ª classe**
- **Sons: são objetos de 1ª classe**
- **... são objetos de 1ª classe**

Funções de Ordem Superior: Higher-Order Functions (HOF)



Uma HOF é uma função que faz, pelo menos, uma das duas coisas abaixo:

- **Recebe um ou mais subprogramas (função, procedimento ou predicado) como argumento**
- **Retorna um subprograma (função, procedimento ou predicado)**

Permitem criar programas sofisticados, com alto grau de abstração e generalização!

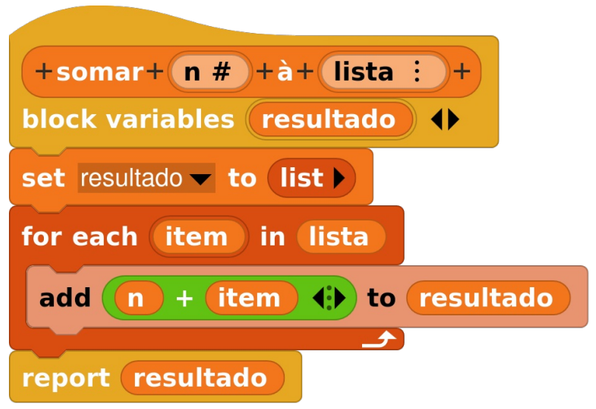
Obs.:

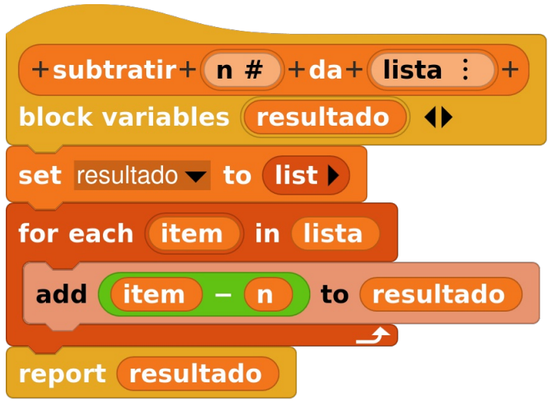
- **Também são chamados de HOP**
- **Funções que não são HO são ditas "first-order" (FO)**

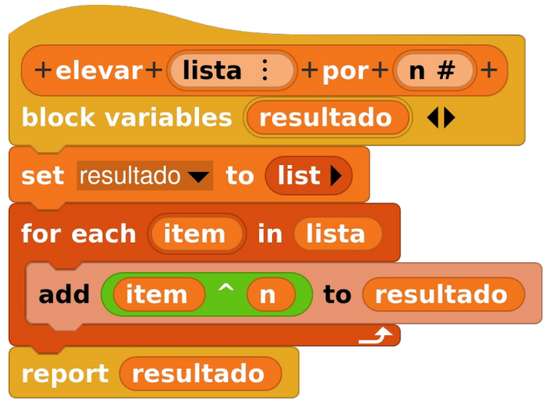
Foto: Tim Depickere, no Unsplash
(https://unsplash.com/photos/a-neon-sign-that-says-liquor-higher-and-deeper-IsO0WKIOkXg)

Funções de Ordem Superior: Higher-Order Functions (HOF)

Retornando ao problema de multiplicar todos os elementos de um array: e se eu quisesse dividir? Ou somar? Ou subtrair? Ou calcular uma potência? Solução comum:

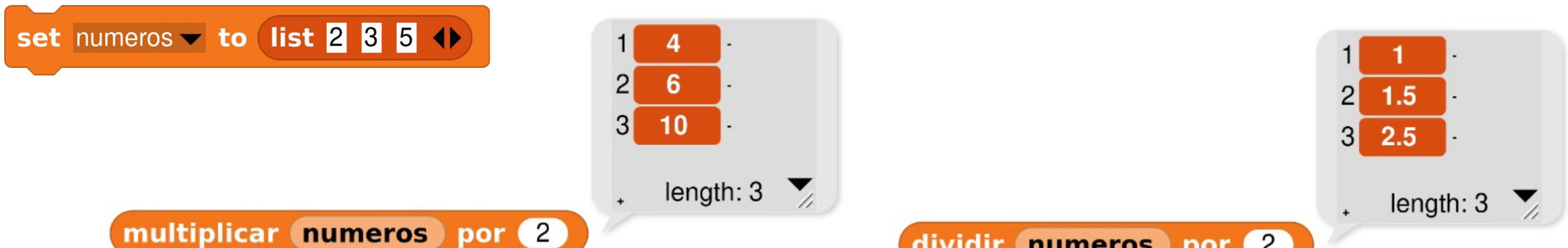


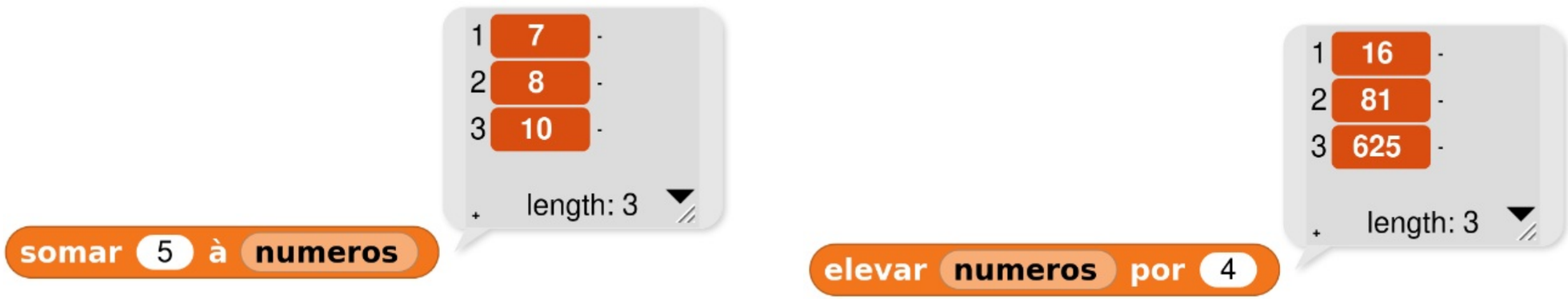




Funções de Ordem Superior: Higher-Order Functions (HOF)

Retornando ao problema de multiplicar todos os elementos de um array: e se eu quisesse dividir? Ou somar? Ou subtrair? Ou calcular uma potência? Solução comum:



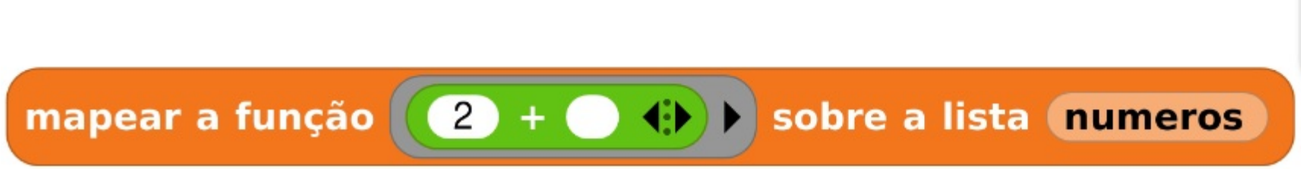


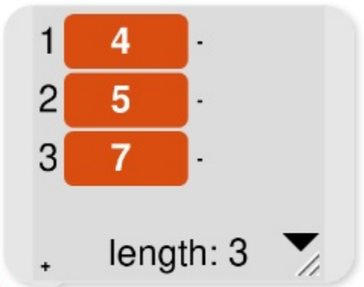
Funções de Ordem Superior: Higher-Order Functions (HOF)

Retornando ao problema de multiplicar todos os elementos de um array: e se eu quisesse dividir? Ou somar? Ou subtrair? Ou calcular uma potência? Solução HOF:

Funções de Ordem Superior: Higher-Order Functions (HOF)

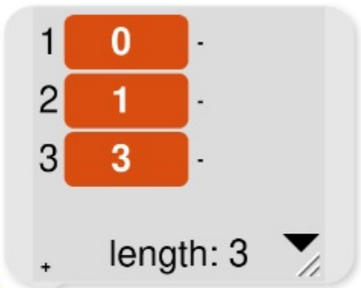
Retornando ao problema de multiplicar todos os elementos de um array: e se eu quisesse dividir? Ou somar? Ou subtrair? Ou calcular uma potência? Solução HOF:

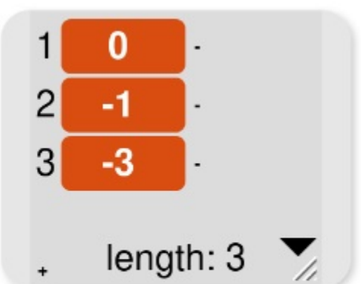




Funções de Ordem Superior: Higher-Order Functions (HOF)

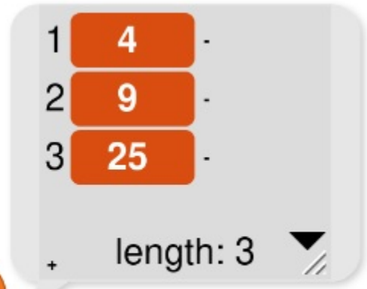
Retornando ao problema de multiplicar todos os elementos de um array: e se eu quisesse dividir? Ou somar? Ou subtrair? Ou calcular uma potência? Solução HOF:

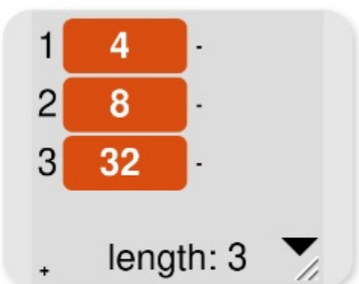




Funções de Ordem Superior: Higher-Order Functions (HOF)

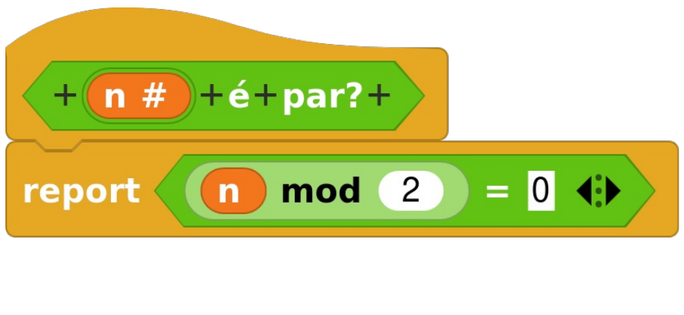
Retornando ao problema de multiplicar todos os elementos de um array: e se eu quisesse dividir? Ou somar? Ou subtrair? Ou calcular uma potência? Solução HOF:





Funções de Ordem Superior: Higher-Order Functions (HOF)

Retornando ao problema de multiplicar todos os elementos de um array: e se eu quisesse dividir? Ou somar? Ou subtrair? Ou calcular uma potência? Solução HOF:

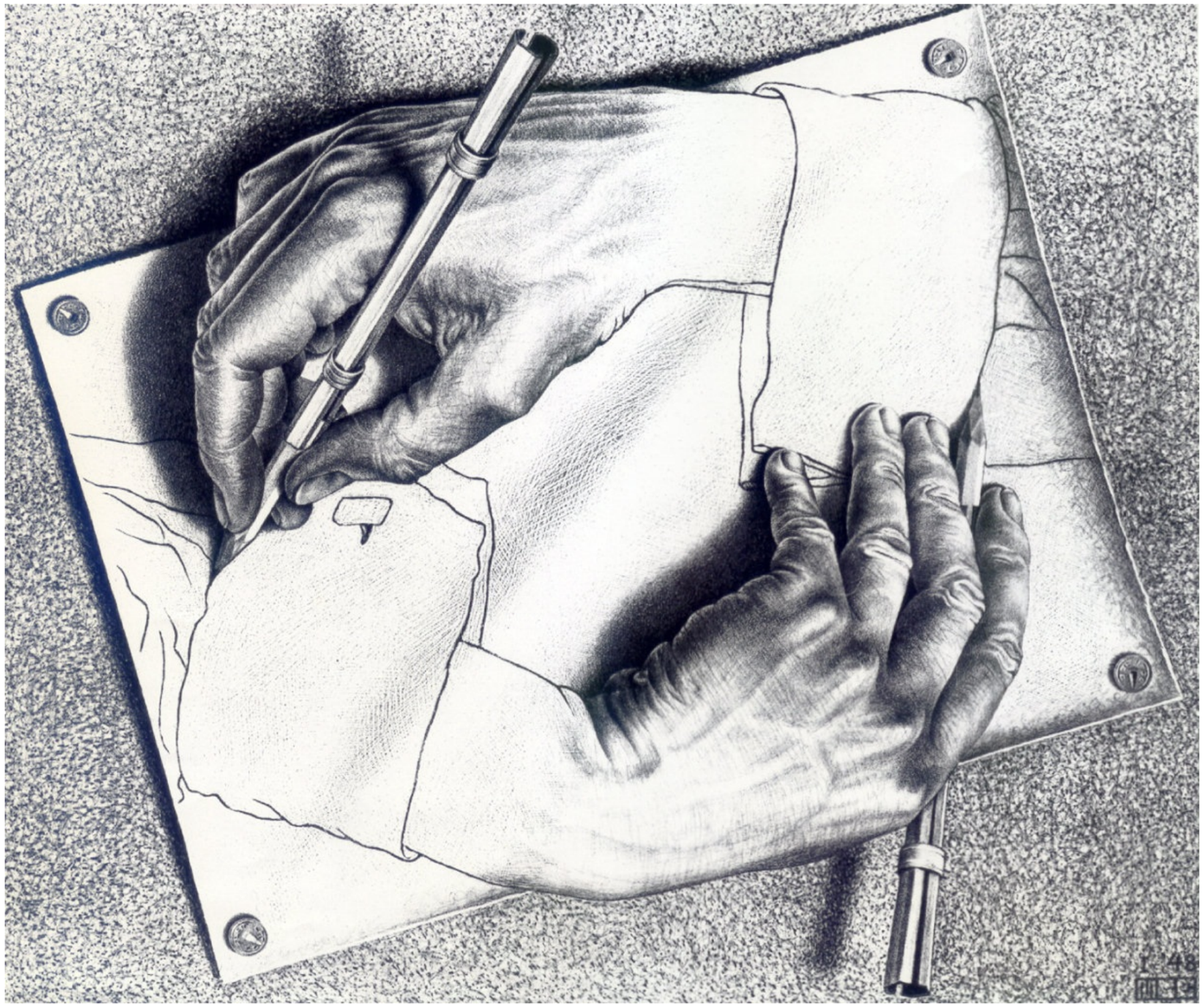


Funções de Ordem Superior: Higher-Order Functions (HOF)

Função x Dado:

HOF mostram que a distinção entre dado e função não é tão simples como pode parecer. Funções SÃO dados que podem ser passados e/ou retornados!

Recursividade



Drawing Hands, por Maurits Cornelis Escher, 1948 (https://www.wikiart.org/en/m-c-escher/drawing-hands)

Uma função é dita recursiva quando ela chama a si mesma para o desenvolvimento de algum processo computacional.

É uma técnica sofisticada de programação, que nos permite solucionar problemas complexos.

Não é fácil de entender no começo...

Recursividade

Para entender o que é recursividade, vamos considerar um exemplo simples: o cálculo do fatorial de um número "n" qualquer. Sabemos que:

$$\text{Para } n \in \mathbb{Z}, \text{ com } n \geq 0:$$

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

$$0! = 1$$

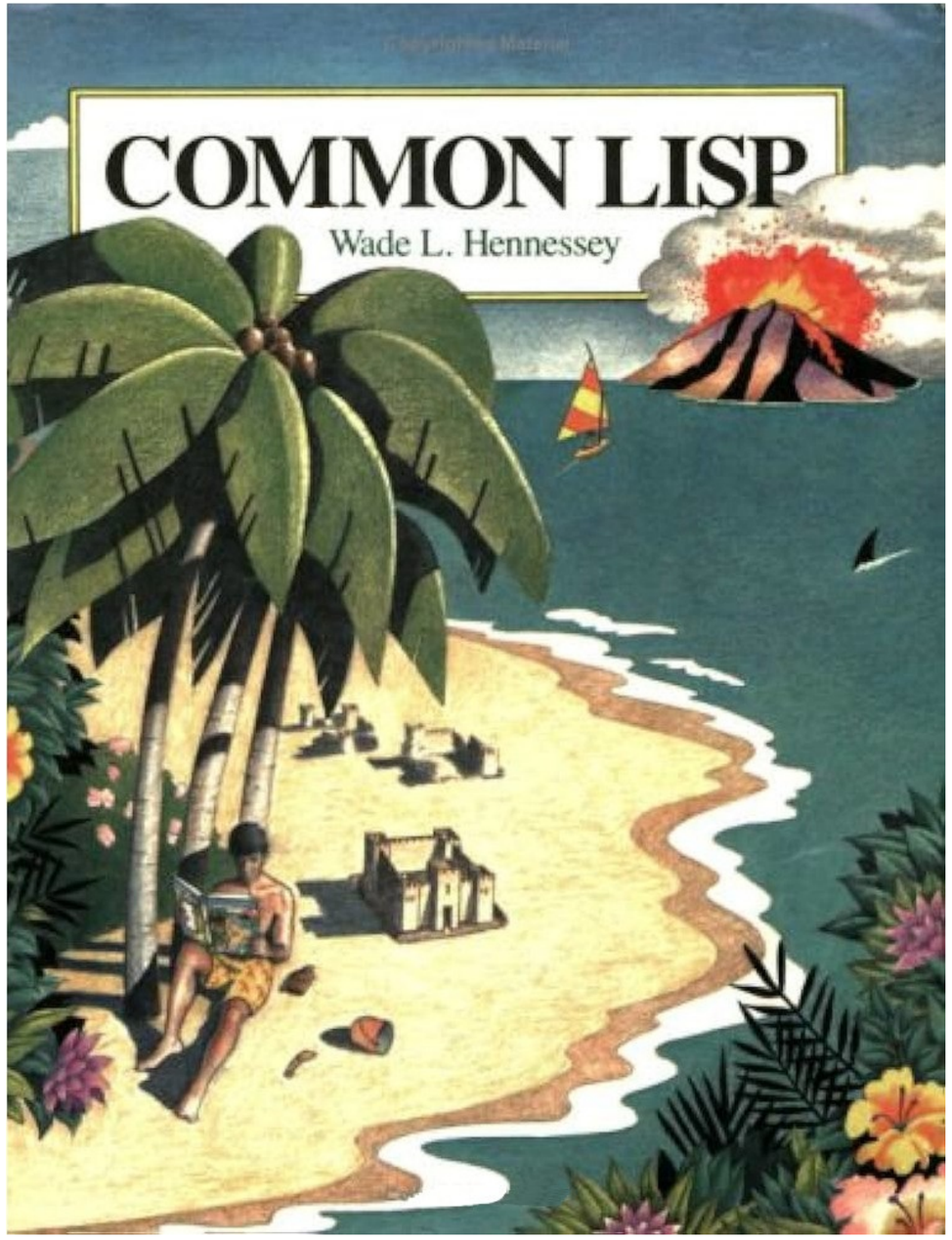
Por exemplo:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Como criar uma função para calcular o fatorial de um número "n" qualquer?

Foto: Alf van Beem, na Wikipedia
(https://commons.wikimedia.org/wiki/File:Droste_Cacao_Alcalinise_blikje,_foto4.JPG

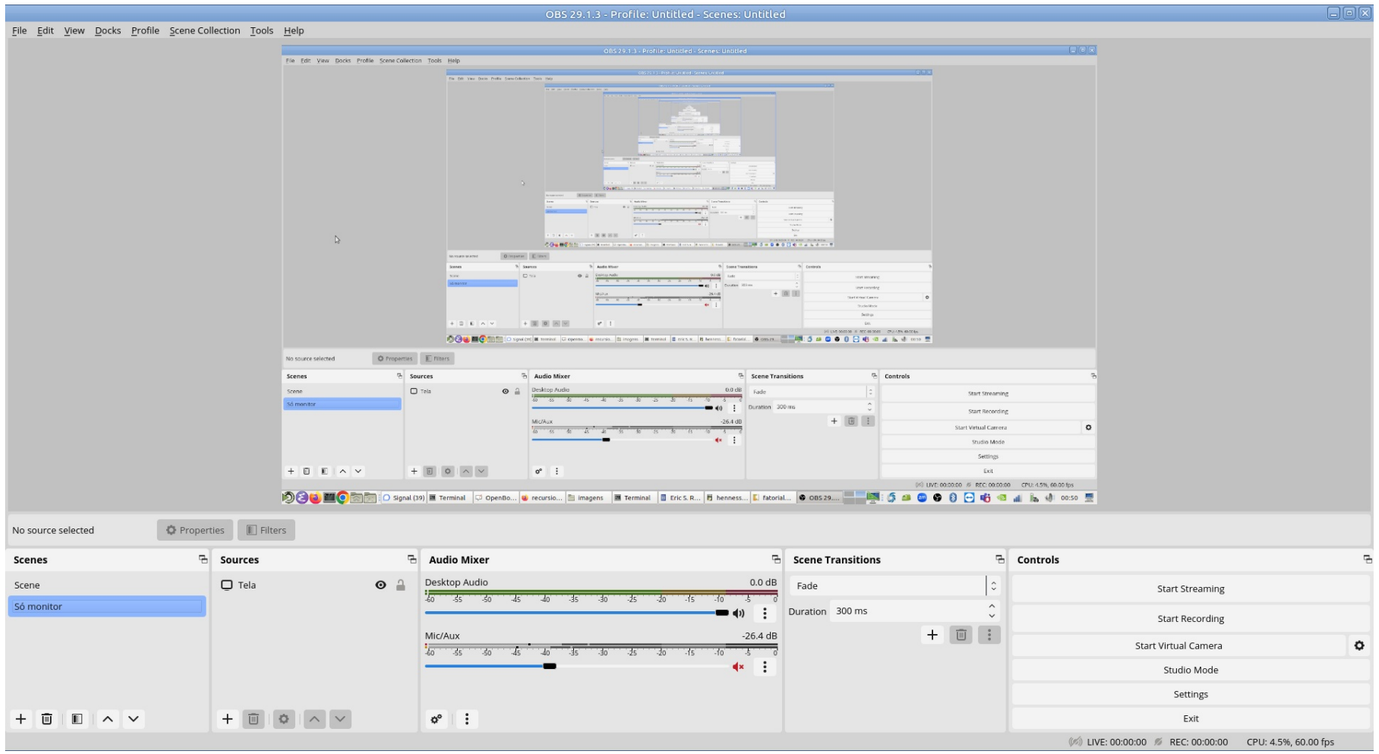
Recursividade



Podemos usar uma estrutura de repetição para implementar a função fatorial. Uma função ou algoritmo que utiliza uma estrutura de repetição é chamada de ITERATIVA:

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

Recursividade



Agora note o seguinte:

$$5! = 5 \times \underbrace{4 \times 3 \times 2 \times 1}_{4!} = 120$$

$$5! = 5 \times 4! = 120$$

$$n! = n \times (n-1)!$$

Uma função ou algoritmo que chama a si mesma é chamada de RECURSIVA:

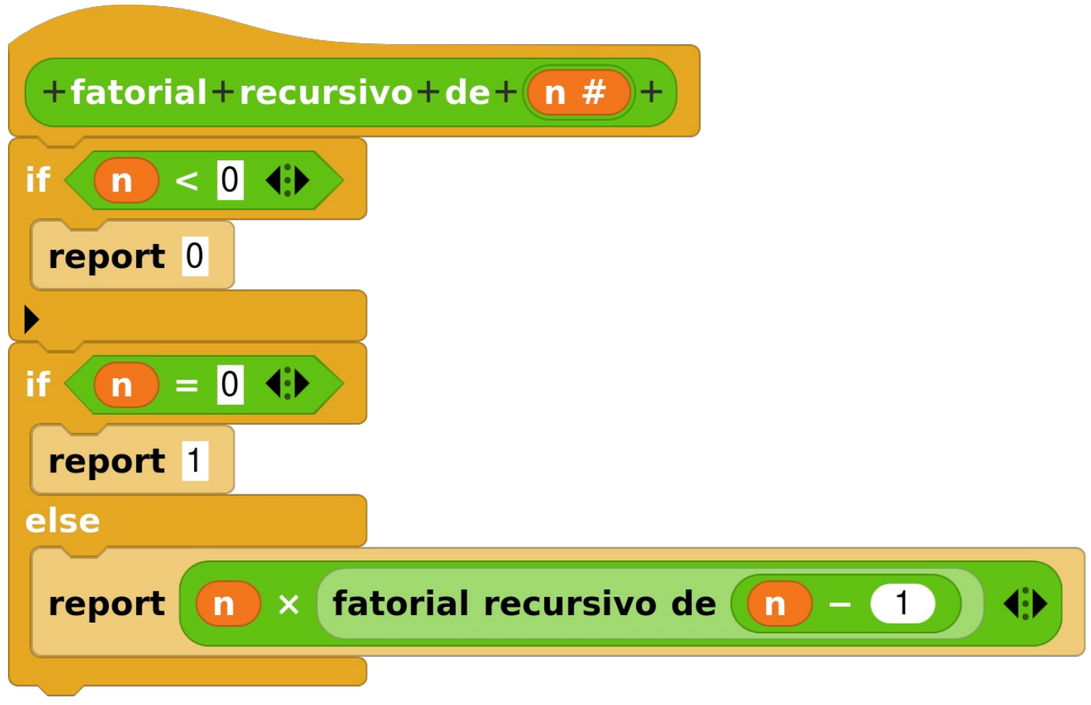
$$\text{Para } n \in \mathbb{Z}, \text{ com } n \geq 0:$$

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \times (n-1)! & \text{se } n > 0 \end{cases}$$

Recursividade

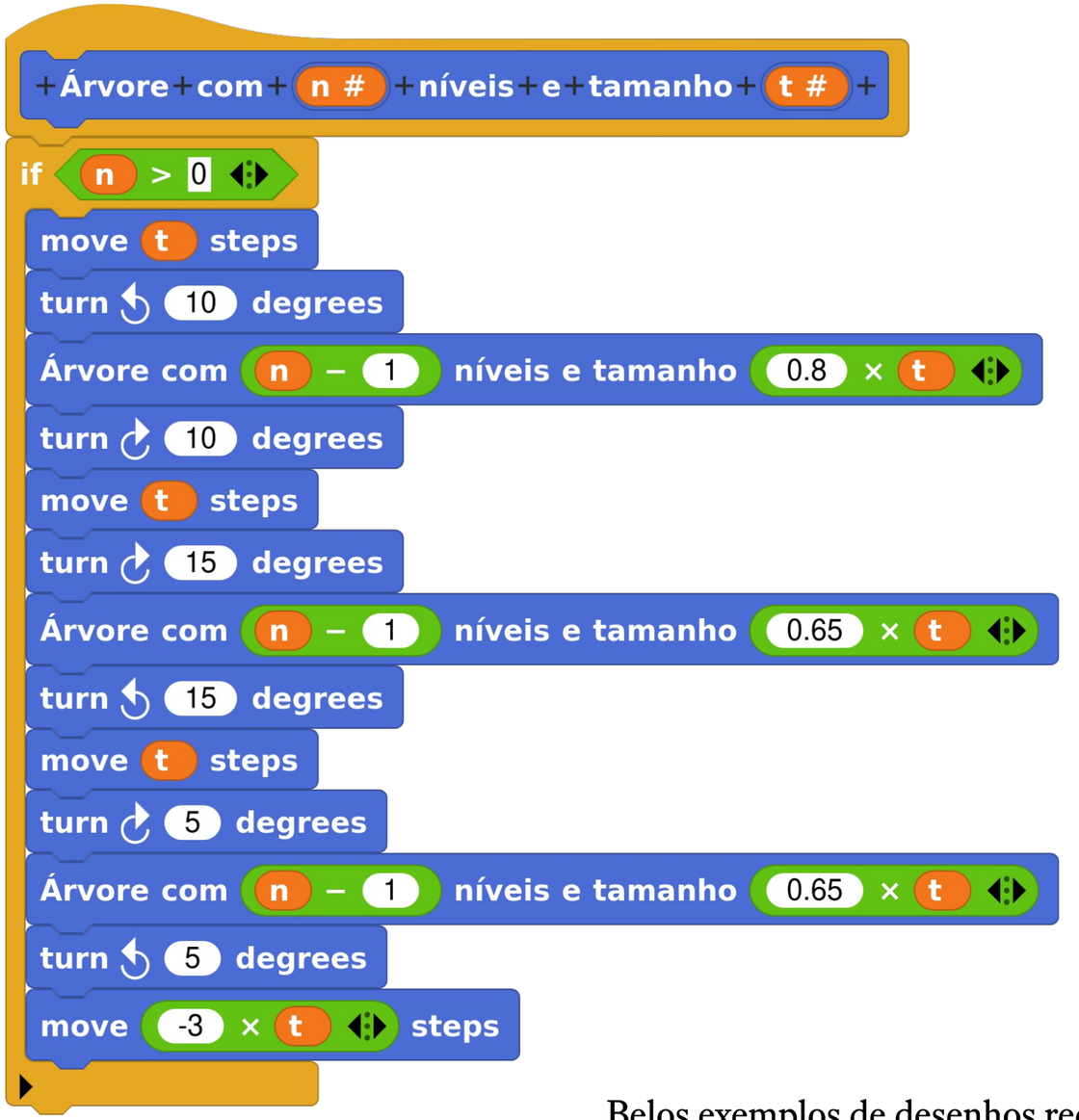
Nevit Dilmen, na Wikipedia (https://commons.wikimedia.org/wiki/File:Droste_1260359-nevit,_corrected.jpg)

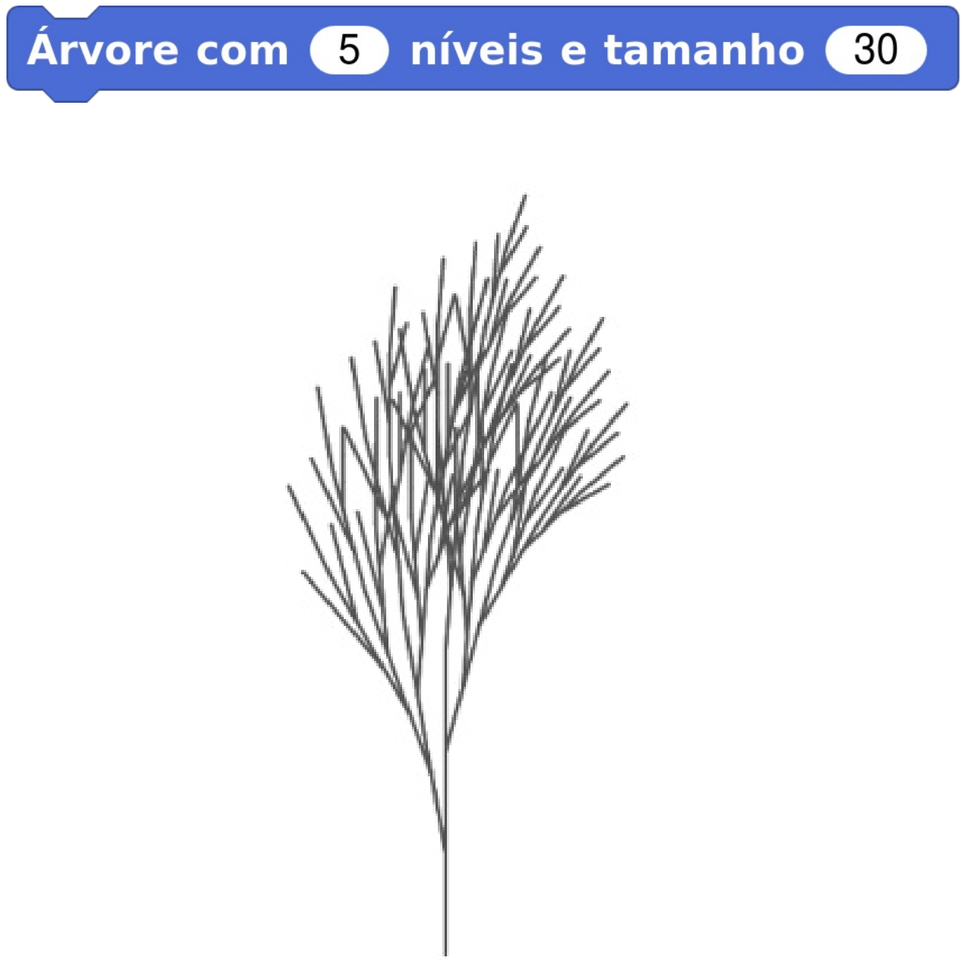
Uma função ou algoritmo que chama a si mesma é chamada de RECURSIVA:



$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \times (n-1)! & \text{se } n > 0 \end{cases}$$

Recursividade





Belos exemplos de desenhos recursivos:
Sarah Bricault, Recursive Drawing: https://bricault.mit.edu/recursive-drawing

Recursividade

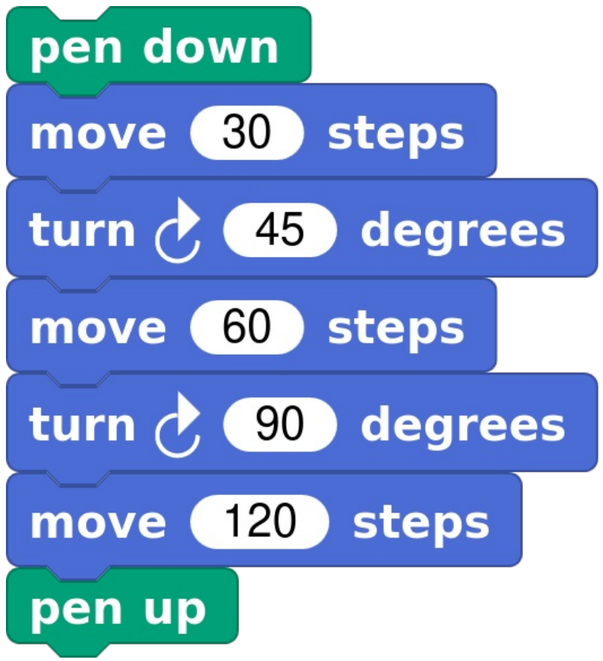
Fechadura de um baú de Henrique VIII, na Wikipedia
(https://commons.wikimedia.org/wiki/File:English_-_Rim_Lock_and_Key_from_a_Chest_Made_for_Henry_VIII_-_Walters_5290,_5291.jpg)

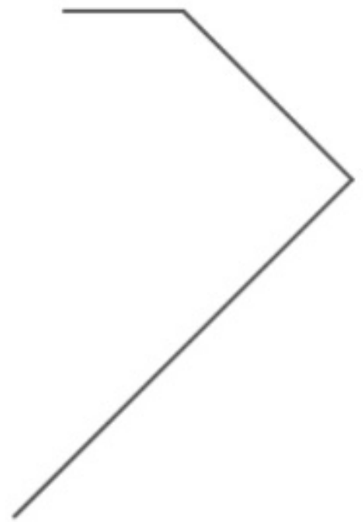
Dominar algoritmos recursivos é uma das chaves que abrem o "baú" de técnicas avançadas, sofisticadas e elegantes da programação.

Não se preocupe se você não entendeu como a recursividade funciona, veremos isso em mais detalhes no curso. No momento o objetivo é que você saiba que recursividade existe e que o Snap*!* nos permite criar funções e algoritmos recursivos.

Continuações

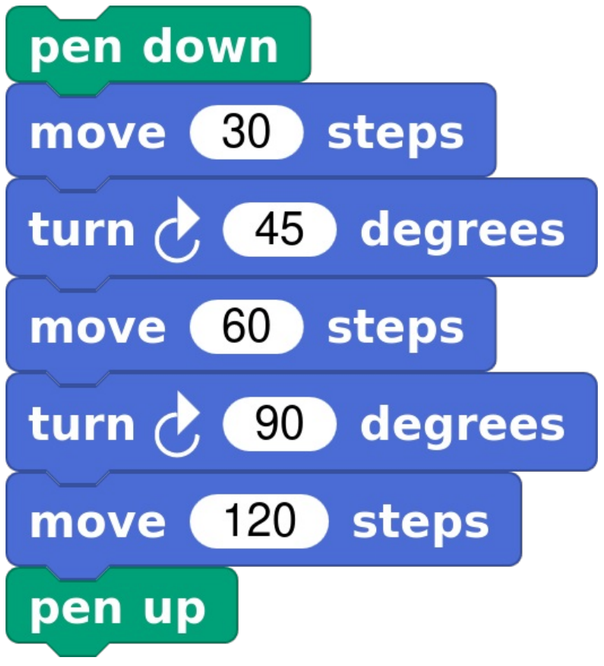
Conceito extremamente avançado e difícil de entender. O objetivo aqui é apenas demonstrar superficialmente a idéia. A continuação de um bloco em um programa corresponde ao processo computacional que será executado após esse bloco terminar.

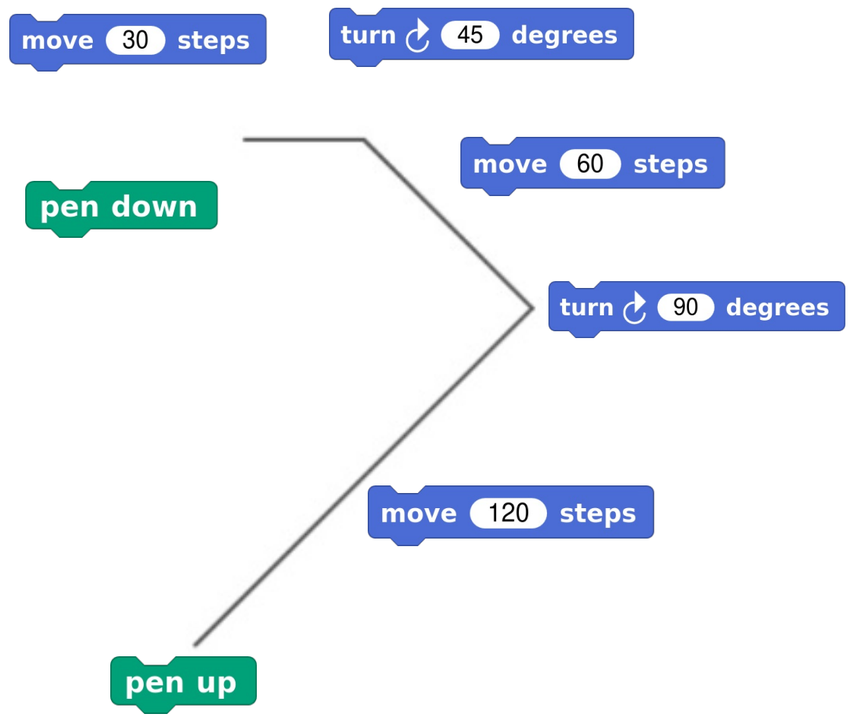




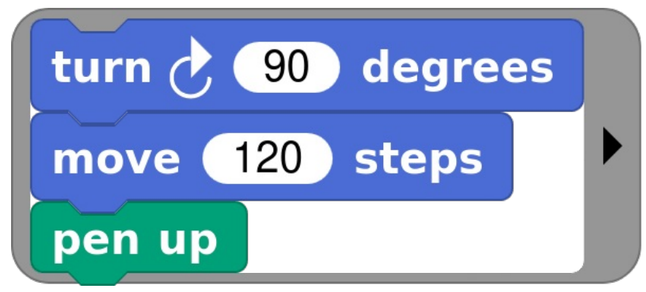
Continuações

Conceito extremamente avançado e difícil de entender. O objetivo aqui é apenas demonstrar superficialmente a idéia. A continuação de um bloco em um programa corresponde ao processo computacional que será executado após esse bloco terminar.



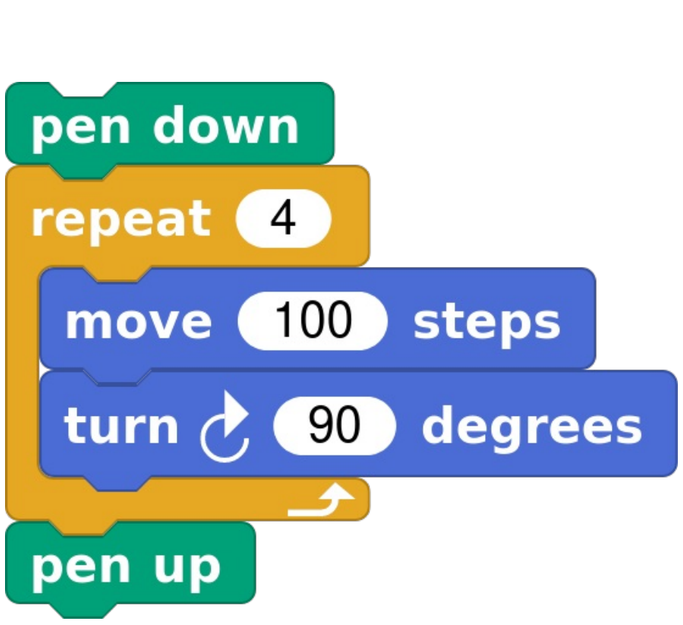


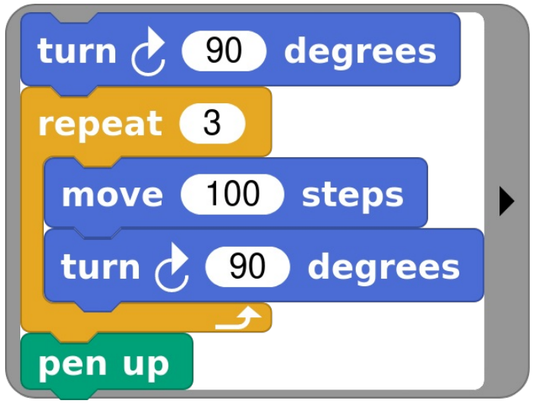
A continuação do bloco "move 60 steps" é:

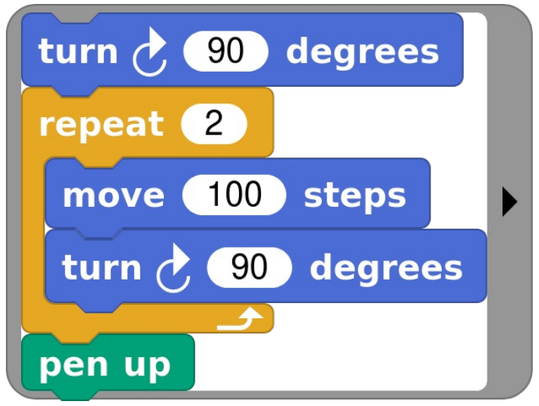


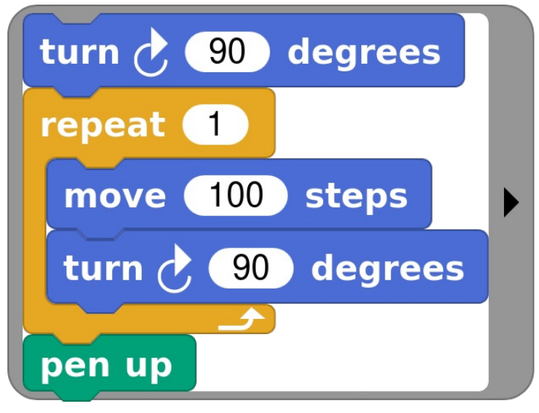
Continuações

Cuidado: o que conta em uma continuação não são os blocos que estão "fisicamente" a seguir mas, sim, o processo computacional que resta para ser executado. Qual será a continuação do bloco "move 100 steps", no programa abaixo?









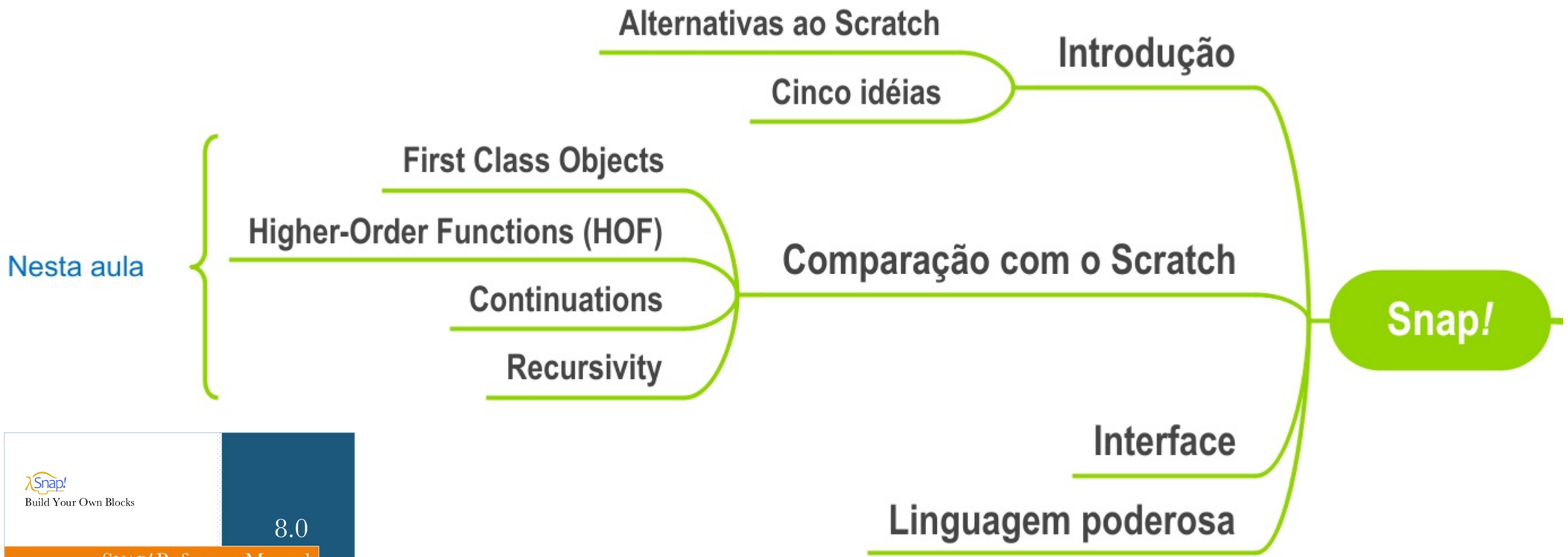
Continuações

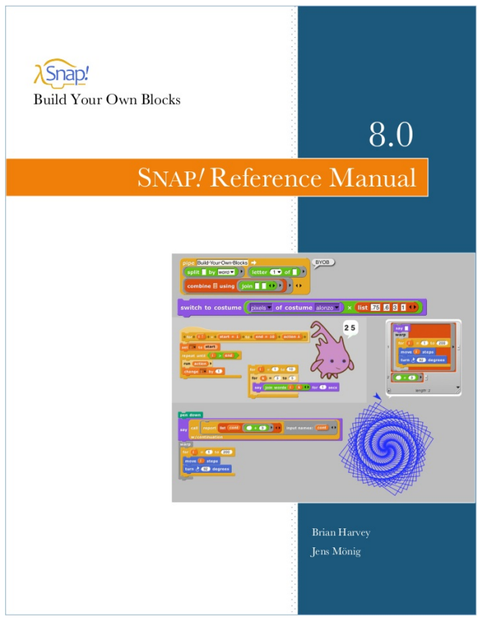
Brian Harvey:

> **"Continuações são o recurso mais desafiador do Snap*!*. Muitos programadores adultos profissionais (em qualquer idioma) seguem suas carreiras sem ter ideia do que é uma continuação. Existem dois tipos de coisas que um procedimento pode fazer com sua continuação:**
>
> **1) Pode invocar a própria continuação para sair de seu próprio código (é assim que criamos CATCH e THROW); e**
>
> **2) Pode colocar sua continuação em uma estrutura de dados externa, uma variável global ou uma lista, e qualquer processo pode reviver esse processo, mesmo depois que o procedimento tiver retornado e todo o script estiver concluído (é assim que criamos MULTITHREAD)."**

Os maiores interessados são pessoas que estão criando ou implementando linguagens de programação, ou que estão resolvendo problemas muito, muito complexos. Não é uma coisa que usamos no dia a dia!

Em resumo


Snap!
Introdução
Alternativas ao Scratch
Cinco idéias
Comparação com o Scratch
First Class Objects
Higher-Order Functions (HOF)
Continuations
Recursivity
Nesta aula
Interface
Linguagem poderosa


Snap!
Build Your Own Blocks
8.0
SNAP! Reference Manual
Brian Harvey
Jens Mönig