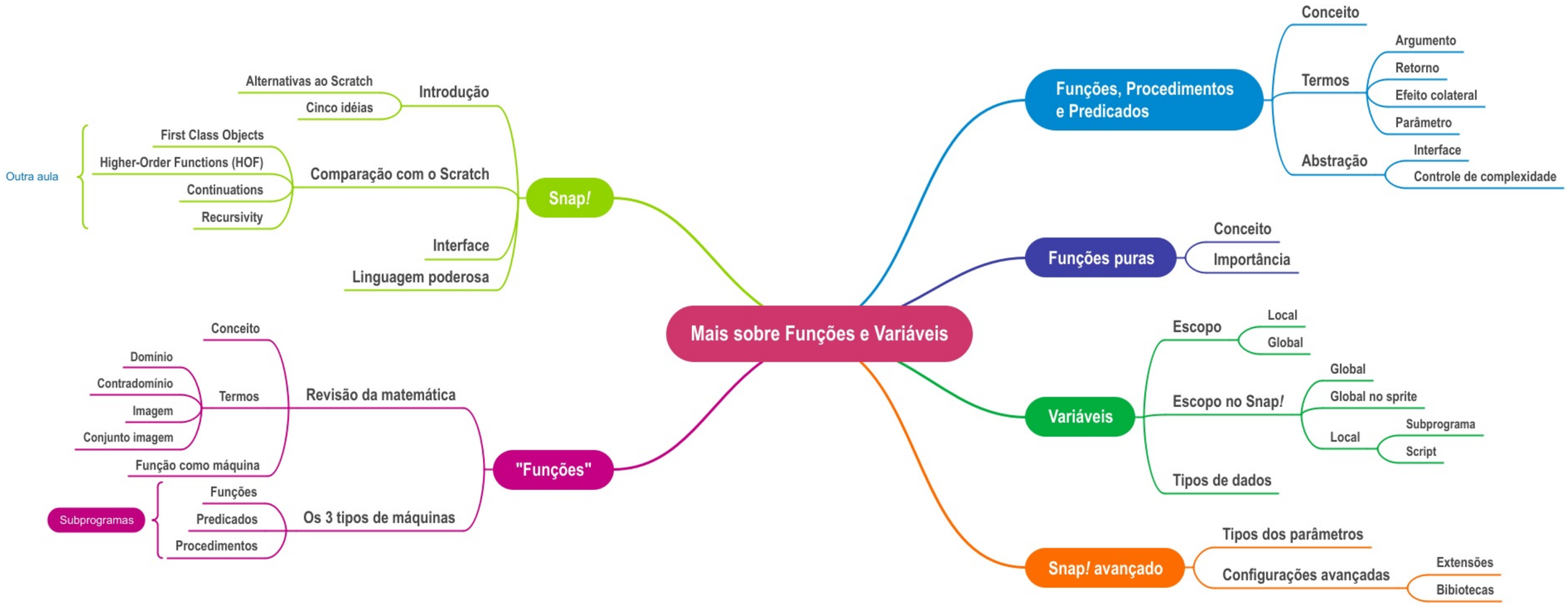
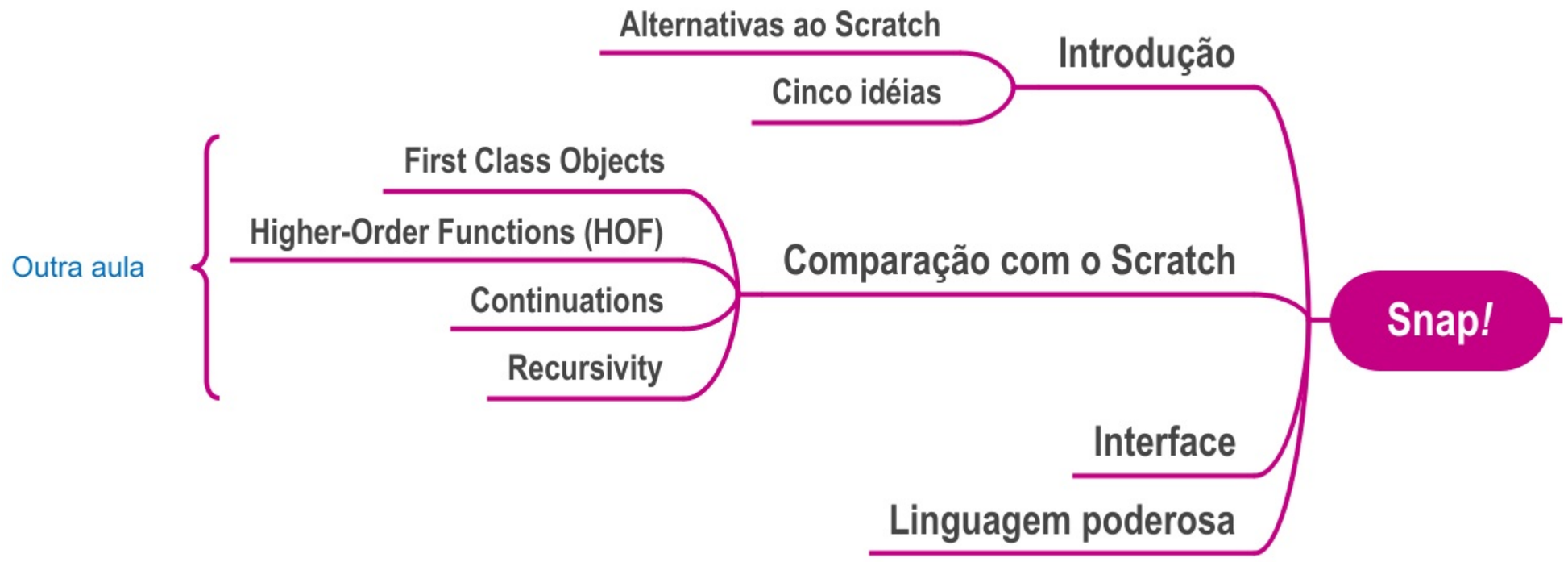
FUNDAMENTOS DA PROGRAMAÇÃO

Mais detalhes sobre Funções e Variáveis



Snap!


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

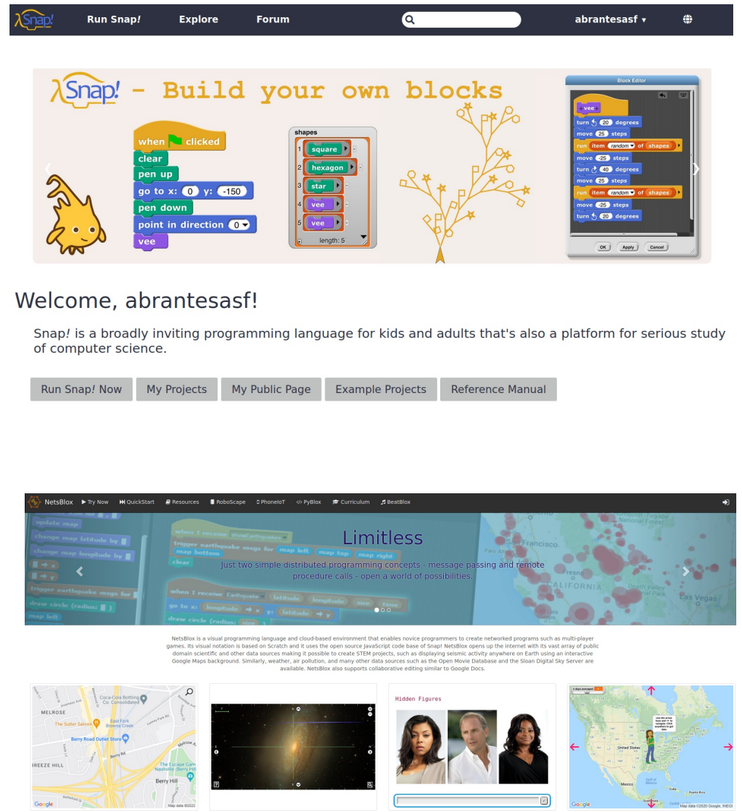
Existem diversas alternativas ao Scratch

O sucesso do MIT com a linguagem Scratch inspirou diversos outros projetos semelhantes. Dois interessantes:

- **Snap*!***

  É uma reimplementação do Scratch, feita por Jens Möning e Brian Harvey, que acrescentou muitas funcionalidades avançadas, sendo particularmente voltado para cursos de ciência da computação.

  https://snap.berkeley.edu

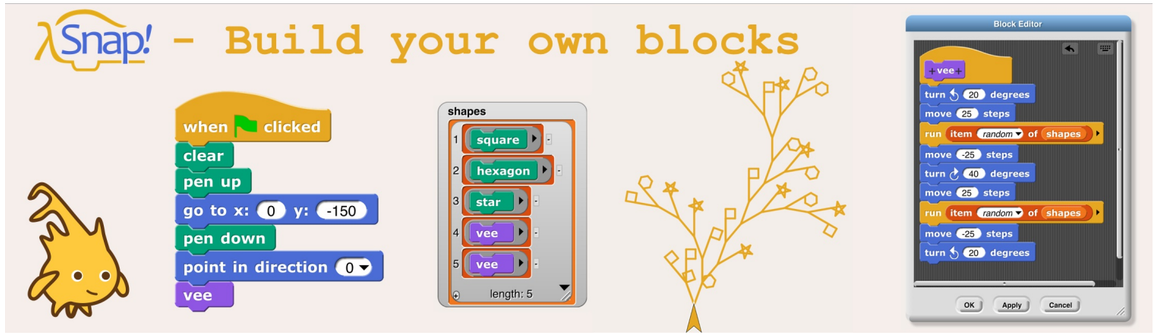


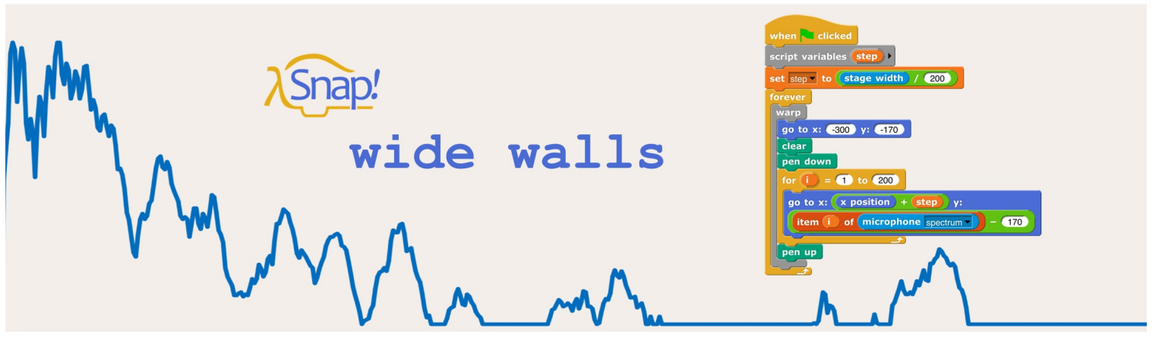
- **NetsBlox**

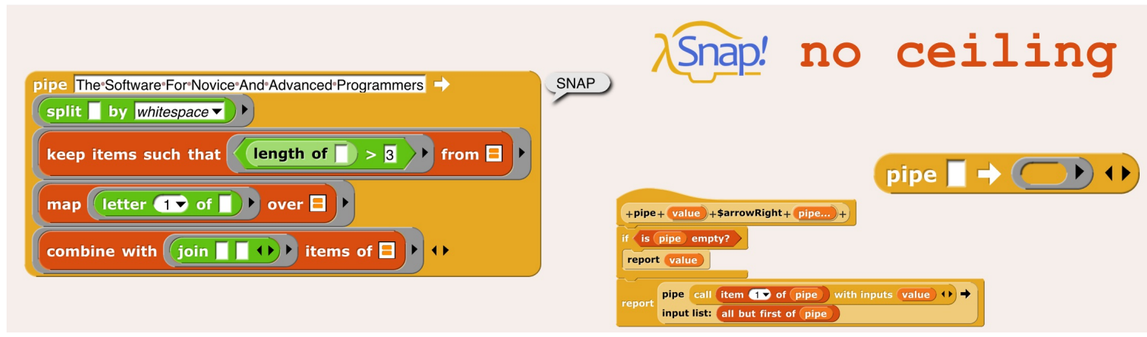
  É uma extensão do Snap*!* que permite utilizar programação distribuída e criar programas para ambiente em rede, como a Internet. Criado por Brian Broll, Jens Möning e outros.

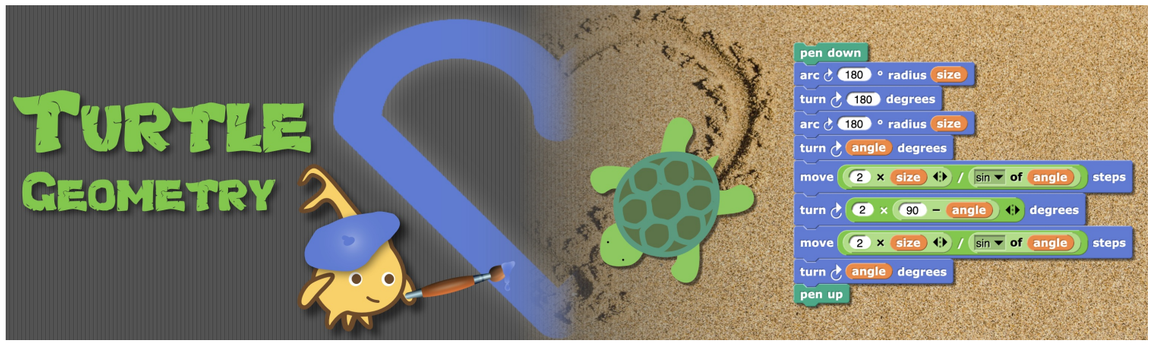
  https://netsblox.org

Snap! Cinco idéias…







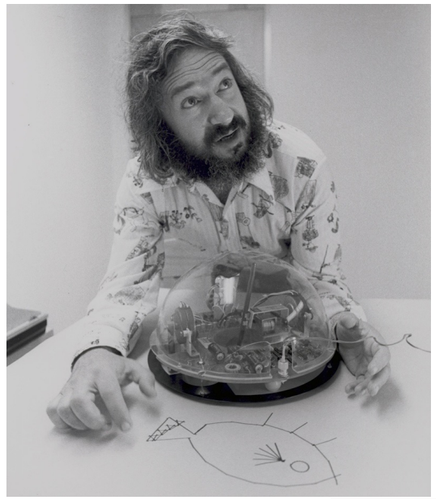


Imagens retiradas do site oficial do Snap!, em (https://snap.berkeley.edu/)

Snap! Cinco idéias...

Logo (1967):

- **Seymour Papert, Wallace Feurzeig e Cynthia Solomon**
- **Dialeto Lisp, voltado para crianças**
- **Em 1969: especificação da tartaruga**
- **Em 1970: primeira turtle para a Logo**



https://roamerrobot.tumblr.com/post/23079345849/the-history-of-turtle-robots

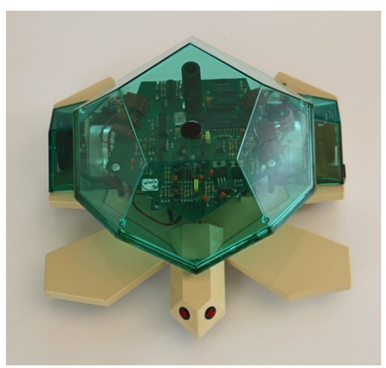
Logo Foundation
(https://el.media.mit.edu/logo-foundation)

Papert com alunos da 4ª série
(https://www.blackhistory.mit.edu/archive/seymour-papert-and-turtle-ca-1968)

Seymour Papert: WikiPedia (https://en.wikipedia.org/wiki/File:Seymour_Papert.jpg)
Wallace Feurzeig: Wikipedia (https://en.wikipedia.org/wiki/File:Wally_Feurzig.jpg)
Chintya Solomon: Wikipedia (https://en.wikipedia.org/wiki/File:Cynthia_Solomon.jpg)

Irwin, 1ª Turtle Wireless
(https://roamerrobot.tumblr.com/post/23079345849/the-history-of-turtle-robots)

Logo Foundation
(https://el.media.mit.edu/logo-foundation)



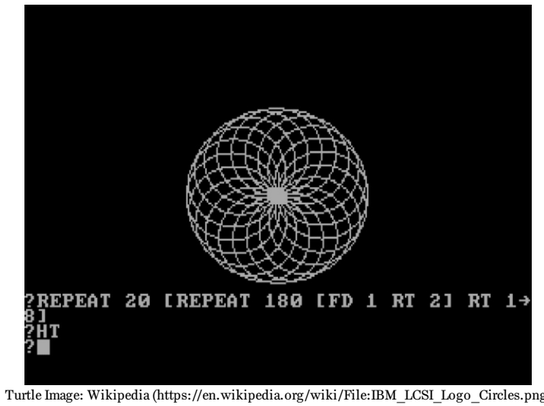
Valiant (https://roamerrobot.tumblr.com/post/23079345849/the-history-of-turtle-robots)

Cynthia com alunos da 1ª série
(https://roamerrobot.tumblr.com/post/23079345849/the-history-of-turtle-robots)

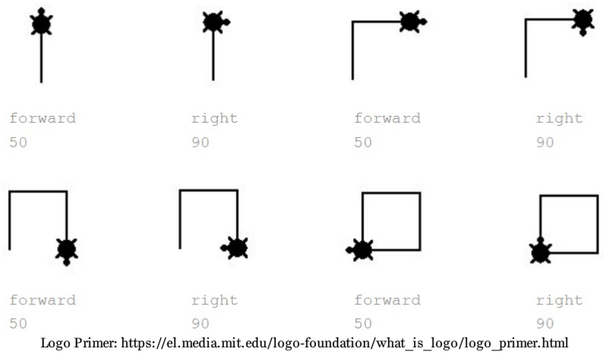
Snap! Cinco idéias...

Logo (1967):

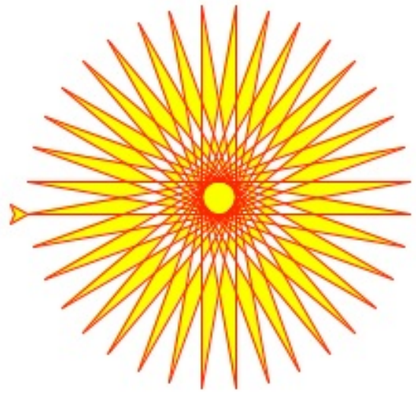
- Evoluiu para o Turtle Graphics: gráficos vetoriais criados com um cursor (a "tartaruga") sobre um plano cartesiano.



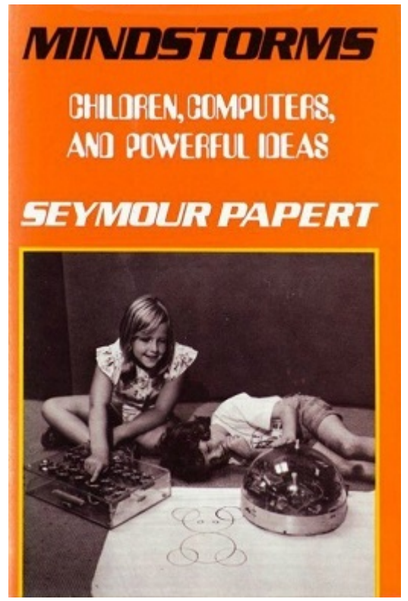
Turtle Image: Wikipedia (https://en.wikipedia.org/wiki/File:IBM_LCSI_Logo_Circles.png)



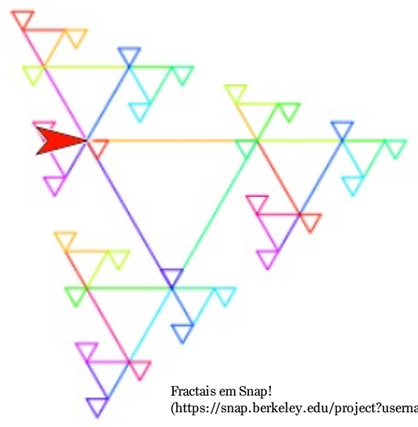
Logo Primer: https://el.media.mit.edu/logo-foundation/what_is_logo/logo_primer.html



Turtle em Python:
https://docs.python.org/3/library/turtle.html



https://el.media.mit.edu/logo-foundation/resources/books.html



Fractais em Snap!
(https://snap.berkeley.edu/project?username=chsapcsp106&projectname=Fractal%20Art)

Snap*!* x Scratch

Scratch foi pensado para ser simples e atrativo para crianças e, apesar de ter funcionalidades importantes, foi simplificado com a remoção de alguns conceitos fundamentais para a computação e a inclusão de alguns limites. Já o Snap*!* foi pensado para retirar esses limites e incluir conceitos fundamentais para o ensino de ciência da computação, como por exemplo:

- **First Class Objects (objetos de 1ª classe)**

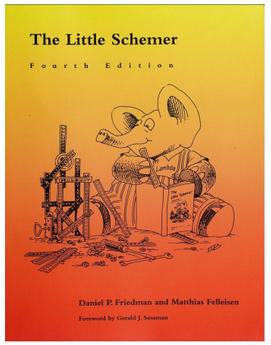
- **Higher-Order Functions - HOF (funções de ordem superior)**

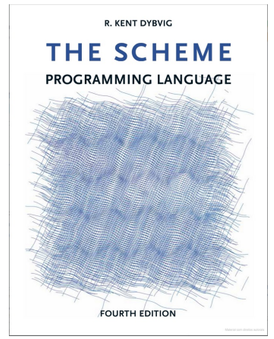
- **First Class Continuations (continuações de 1ª classe)**

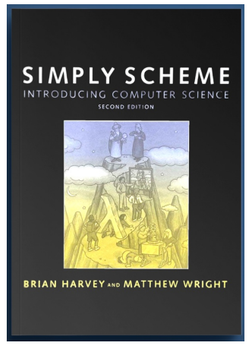
- **Recursividade**

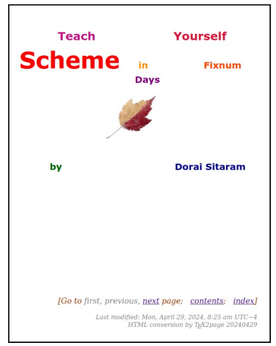
- **...**

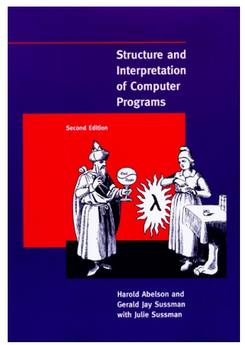
Não julgue um livro pela capa! Snap! é uma linguagem poderosa!

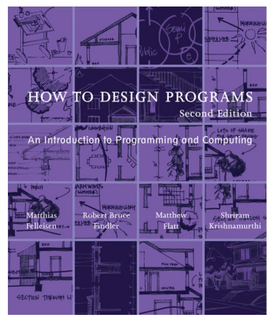


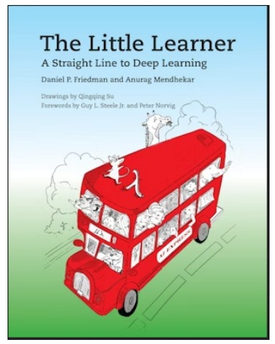


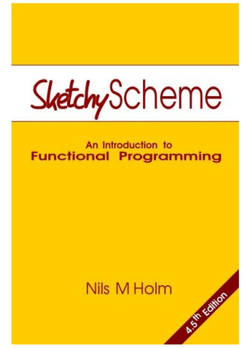


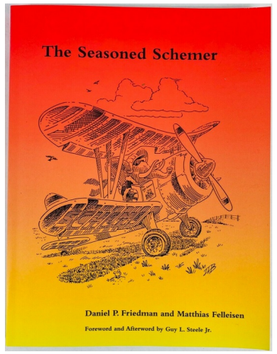


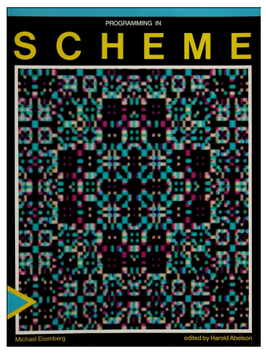


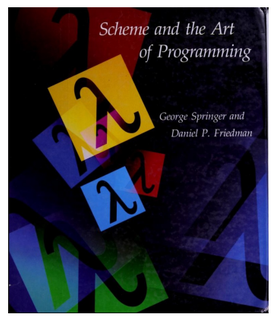


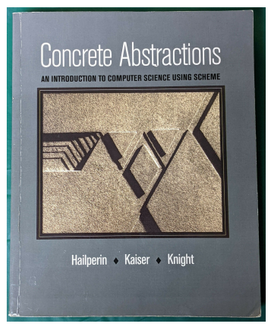


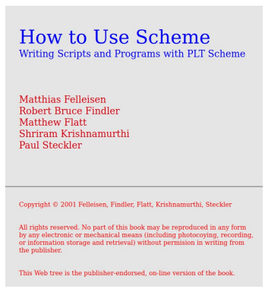


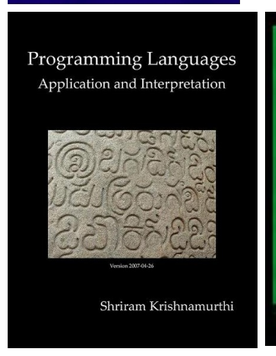


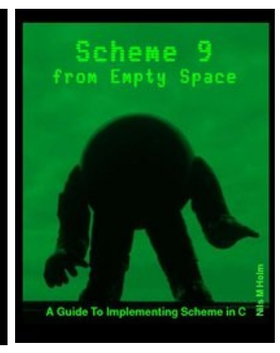


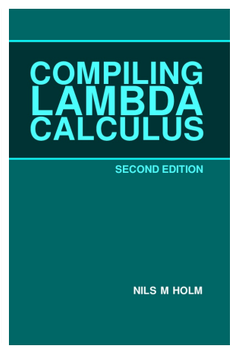


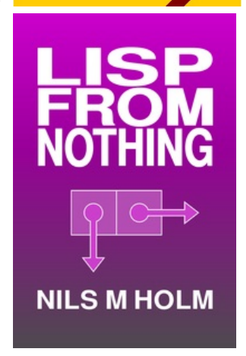


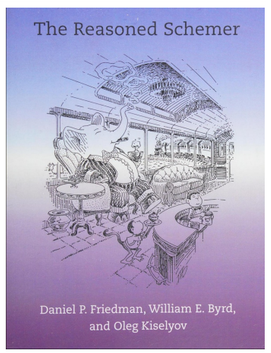


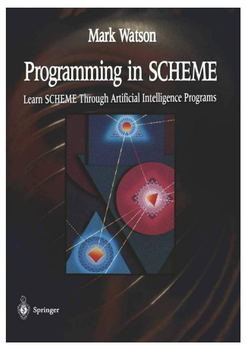


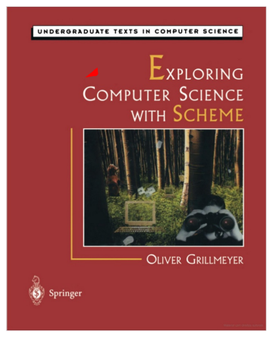


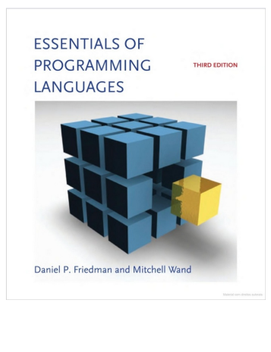


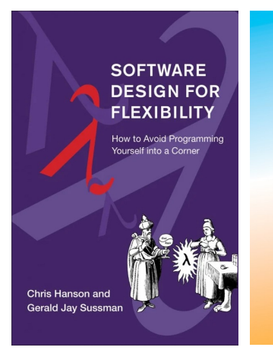


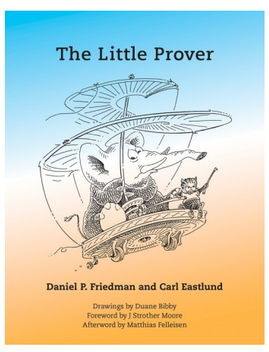








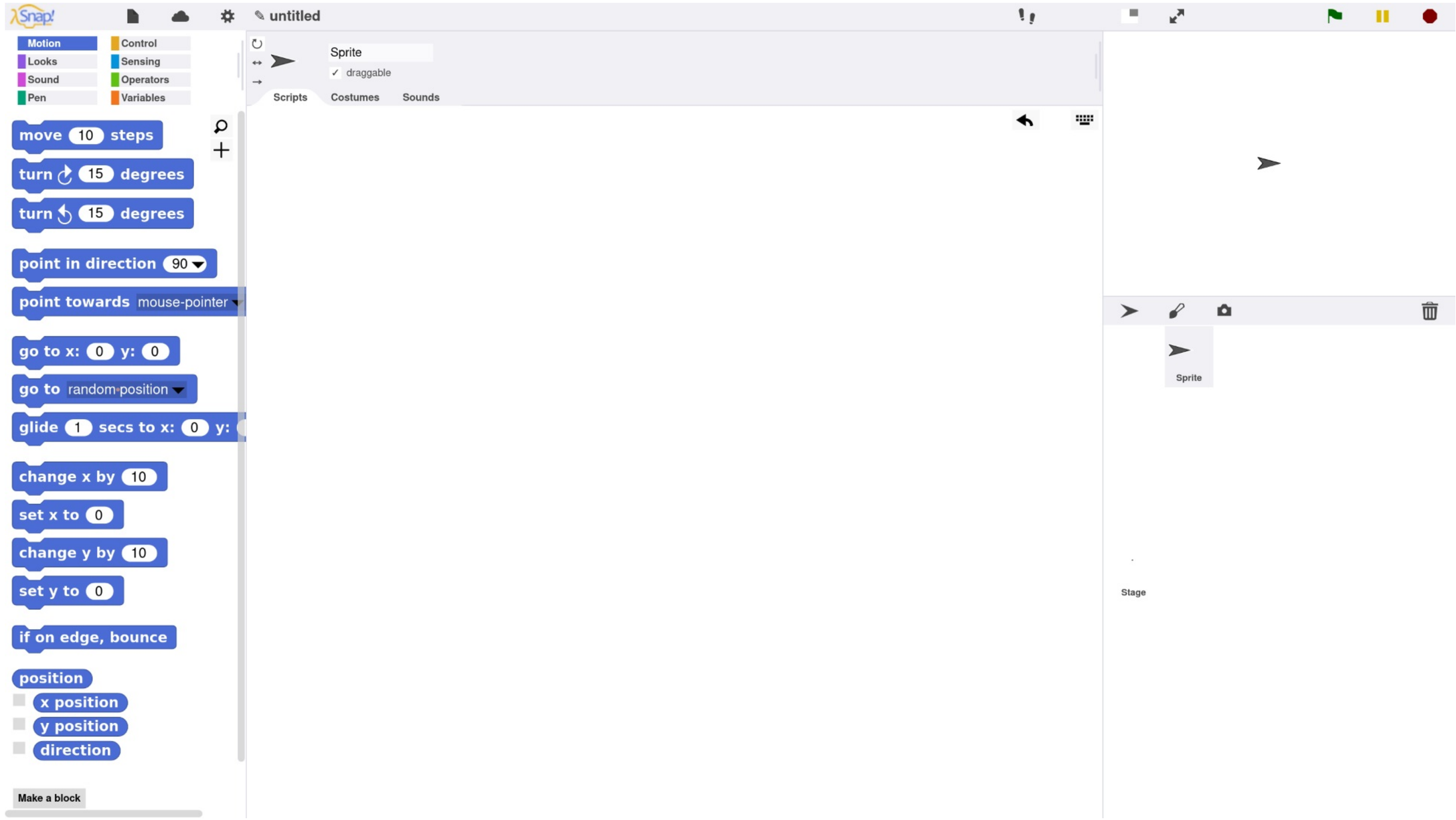




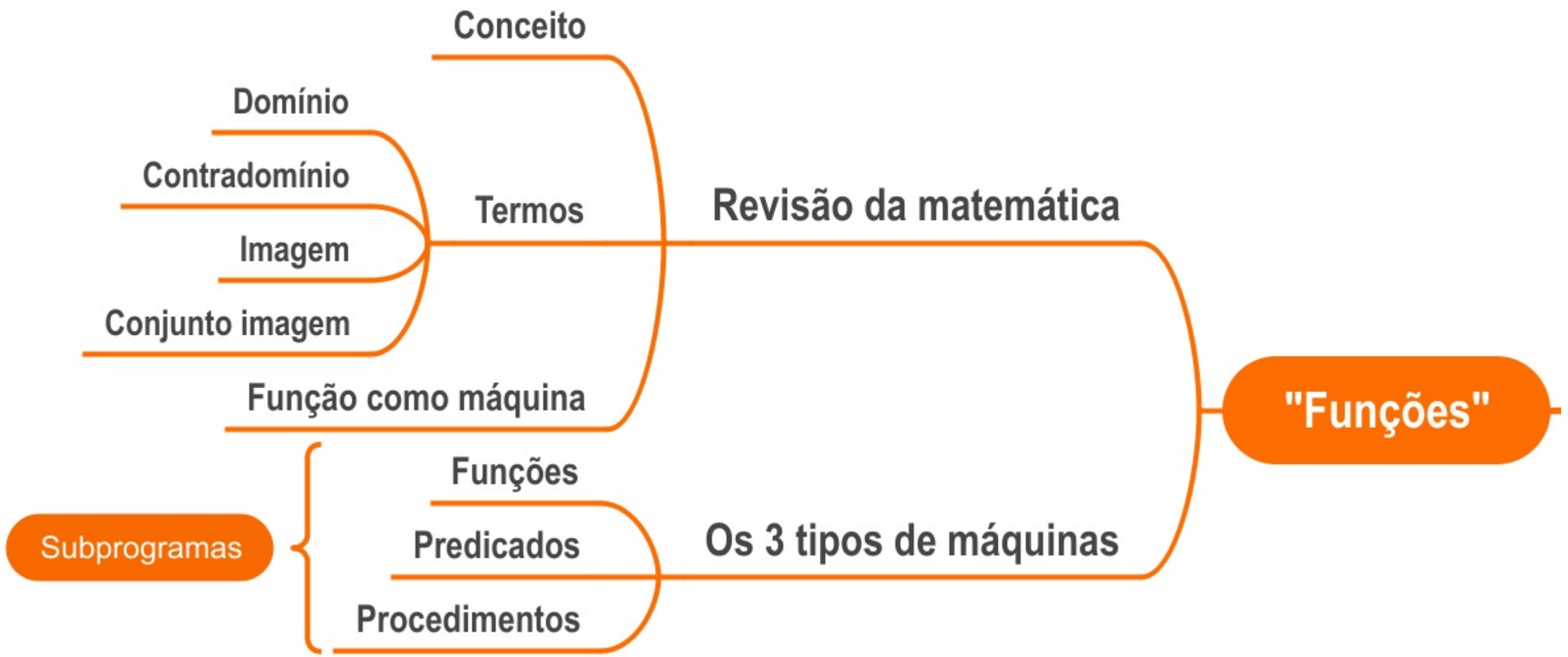
Os blocos são disfarces para facilitar o aprendizado!

"Snap! é Scheme disfarçada de Scratch."

Interface: basicamente a mesma do Scratch, com mais funcionalidades

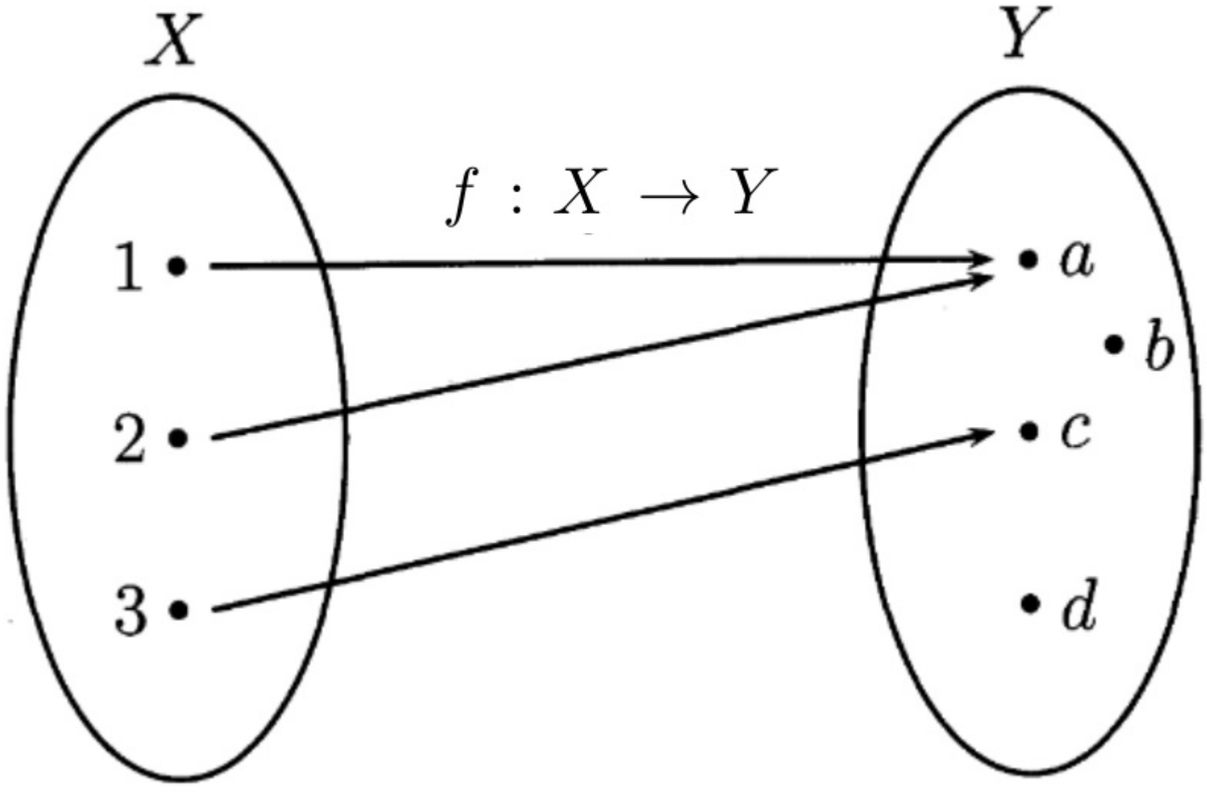


"Funções"


"Funções"
Revisão da matemática
Conceito
Termos
Domínio
Contradomínio
Imagem
Conjunto imagem
Função como máquina
Os 3 tipos de máquinas
Funções
Predicados
Procedimentos
Subprogramas

Revisão de matemática: conceito de função

É uma regra que associa a cada elemento de um conjunto X um único elemento de um conjunto Y. Todos os elementos de X devem estar associados a um único elemento de Y:



MUNIZ NETO, A. C. Tópicos de Matemática Elementar, Volume 3: Introdução à Análise. 2. ed.
Rio de Janeiro: Sociedade Brasileira de Matemática, 2012. v. 3. (Coleção do Professor de Matemática, v. 3).

Revisão sobre Funções

Abrantes Araújo Silva Filho

Revisão: 2023-06-15

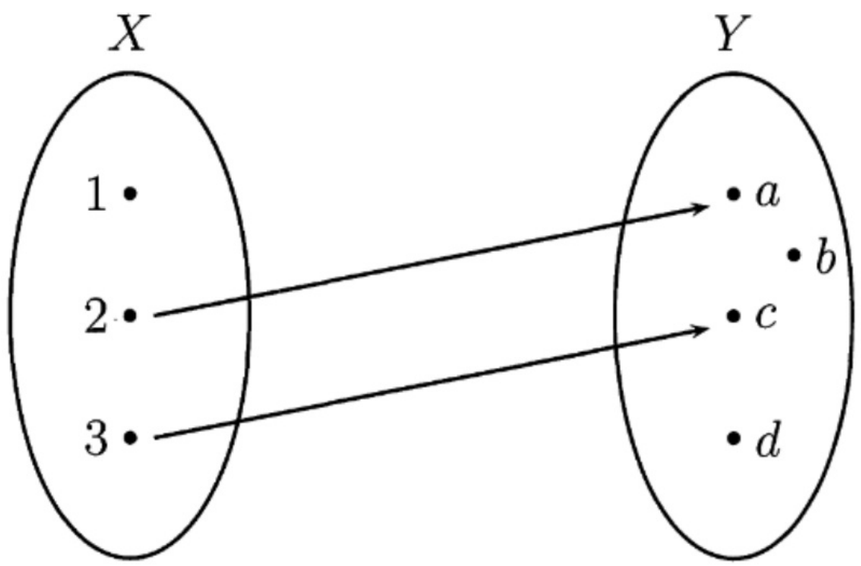
Resumo

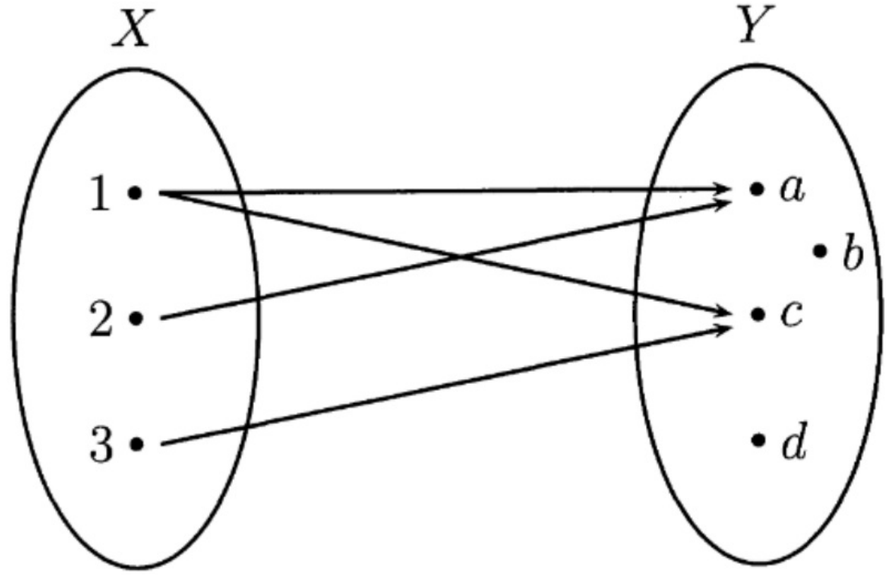
Breve revisão sobre os conceitos fundamentais sobre funções: conceito, domínio, contradomínio, imagem, conjunto imagem.

Sumário

Revisão de matemática: conceito de função

Se algum elemento de X não estiver associado a um elemento de Y, ou se um elemento de X estiver associado a mais de um elemento de Y, não temos uma função.

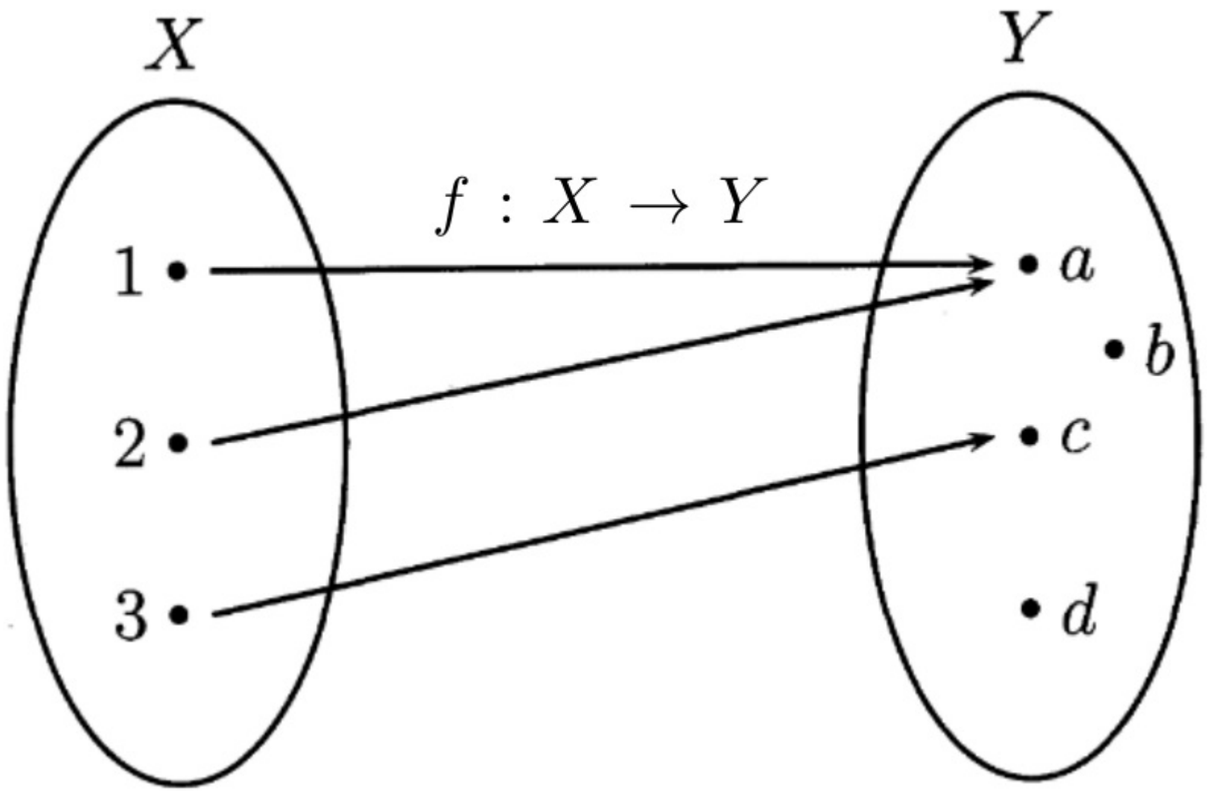




MUNIZ NETO, A. C. Tópicos de Matemática Elementar, Volume 3: Introdução à Análise. 2. ed.
Rio de Janeiro: Sociedade Brasileira de Matemática, 2012. v. 3. (Coleção do Professor de Matemática, v. 3).

Revisão de matemática: domínio da função

O domínio de uma função são os valores de X para os quais as operações indicadas pela regra da função são possíveis. O domínio não inclui os valores de X para os quais as operações indicadas pela regra da função não são possíveis.

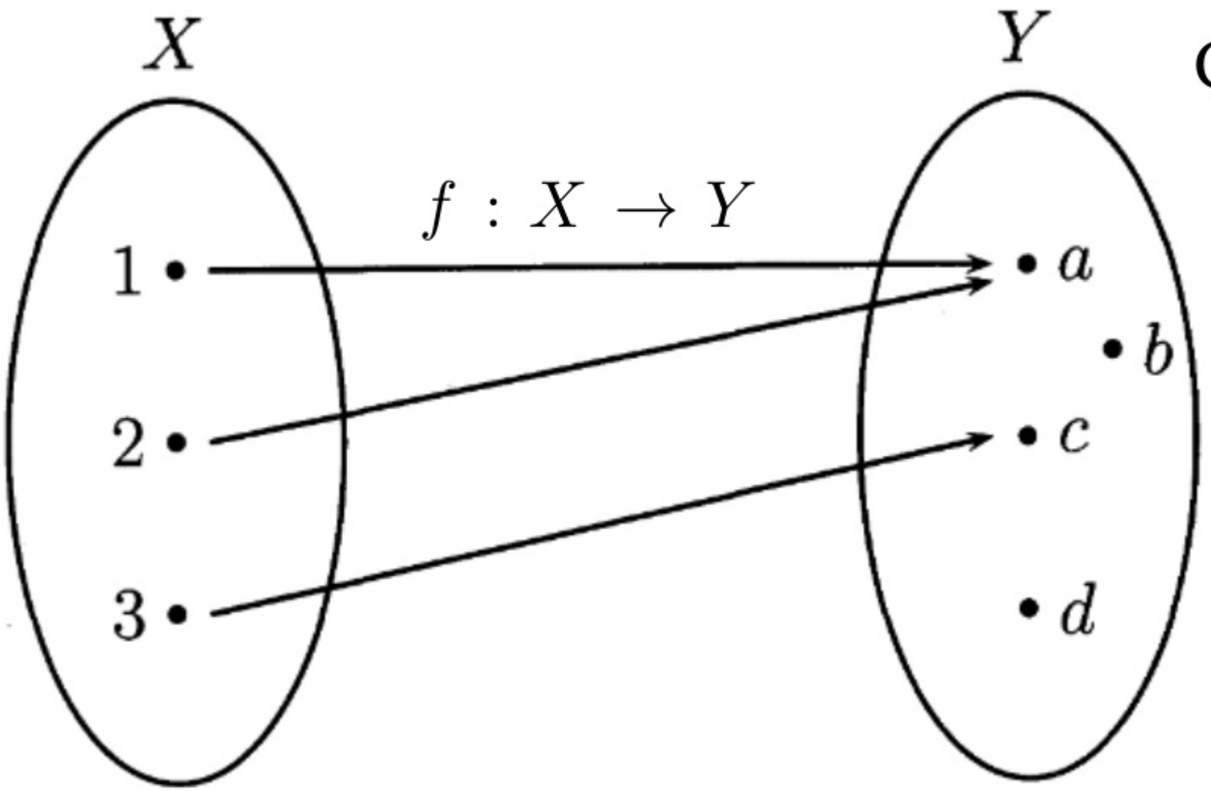


Domínio da função: $X = \{1, 2, 3\}$

MUNIZ NETO, A. C. Tópicos de Matemática Elementar, Volume 3: Introdução à Análise. 2. ed.
Rio de Janeiro: Sociedade Brasileira de Matemática, 2012. v. 3. (Coleção do Professor de Matemática, v. 3).

Revisão de matemática: contradomínio da função

O contradomínio de uma função são todos os valores possíveis de Y. O contradomínio corresponde a tudo o que a função pode produzir como saída.

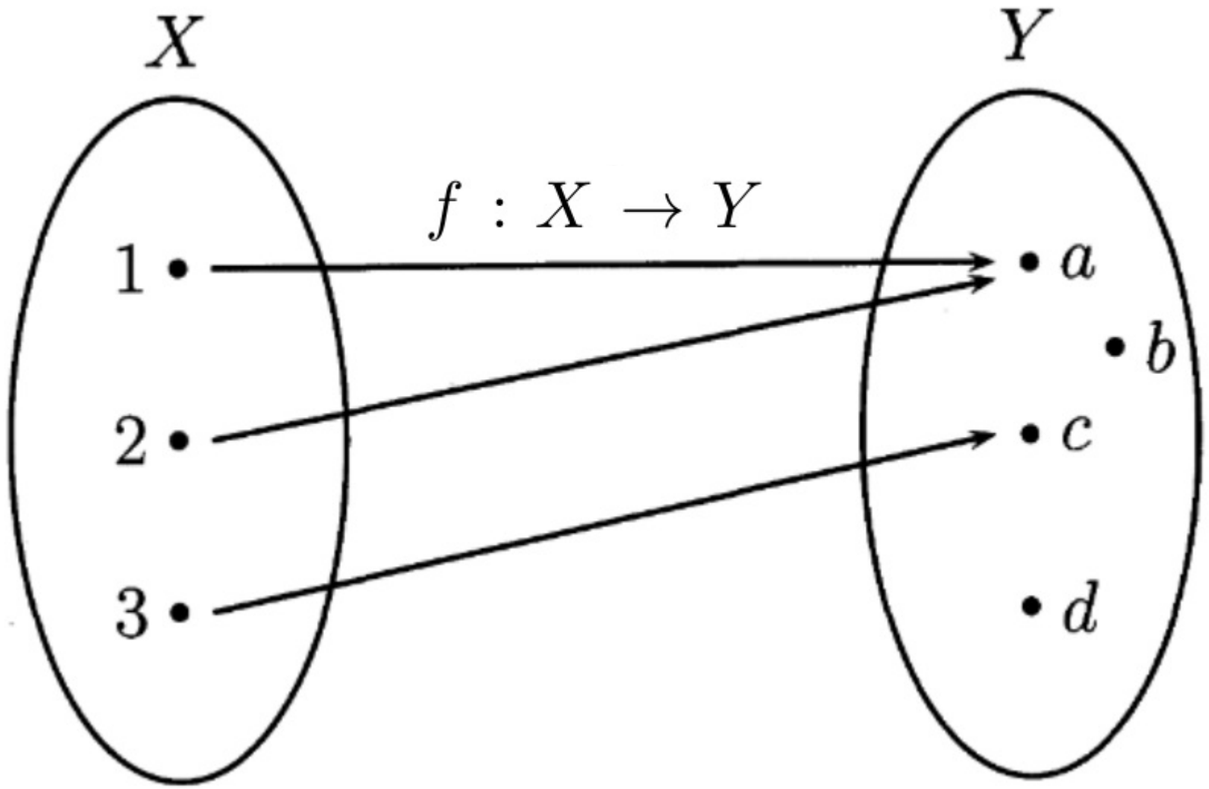


Contradomínio da função: $Y = \{a, b, c, d\}$

MUNIZ NETO, A. C. Tópicos de Matemática Elementar, Volume 3: Introdução à Análise. 2. ed.
Rio de Janeiro: Sociedade Brasileira de Matemática, 2012. v. 3. (Coleção do Professor de Matemática, v. 3).

Revisão de matemática: imagem de x

Uma imagem corresponde a um valor específico de y que está associado a um valor de x pela regra da função. Se x é um elemento de X, o único y de Y associado à x é denominado de imagem de x pela função f.



$$y = f(x)$$

Imagem de 1: $y = f(1) = a$

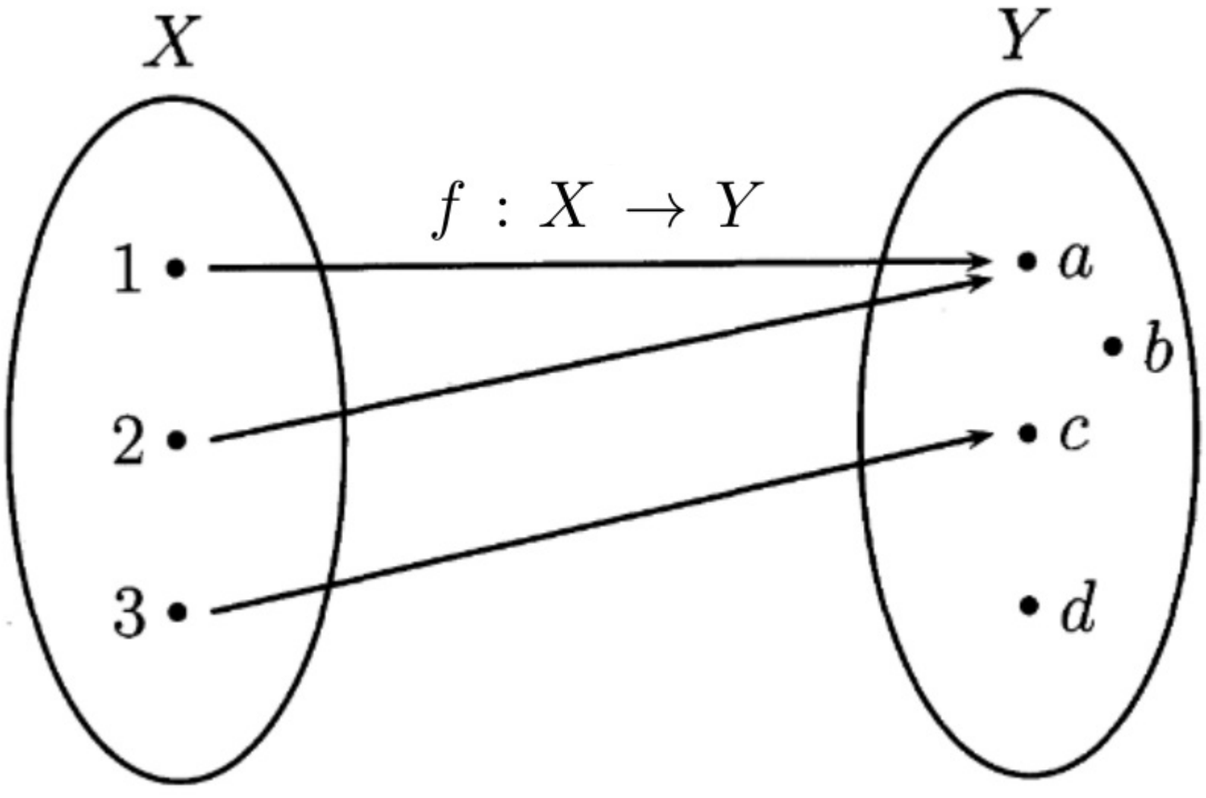
Imagem de 2: $y = f(2) = a$

Imagem de 3: $y = f(3) = c$

MUNIZ NETO, A. C. Tópicos de Matemática Elementar, Volume 3: Introdução à Análise. 2. ed.
Rio de Janeiro: Sociedade Brasileira de Matemática, 2012. v. 3. (Coleção do Professor de Matemática, v. 3).

Revisão de matemática: conjunto imagem

O conjunto imagem corresponde ao conjunto de todas as imagens obtidas pela função.



$$y = f(x)$$

Conjunto imagem da função: $\{a, c\}$

$$\text{Im}(f) \subseteq Y$$

MUNIZ NETO, A. C. Tópicos de Matemática Elementar, Volume 3: Introdução à Análise. 2. ed.
Rio de Janeiro: Sociedade Brasileira de Matemática, 2012. v. 3. (Coleção do Professor de Matemática, v. 3).

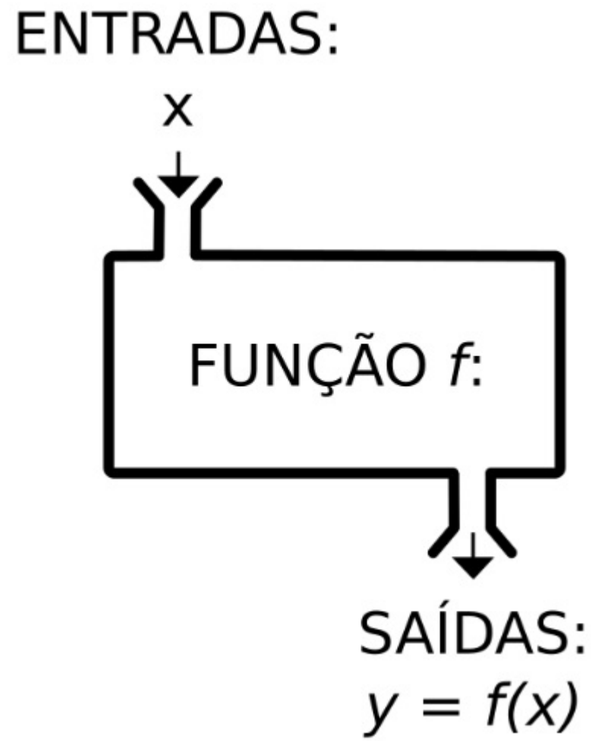
Função na programação: domínio e contradomínio

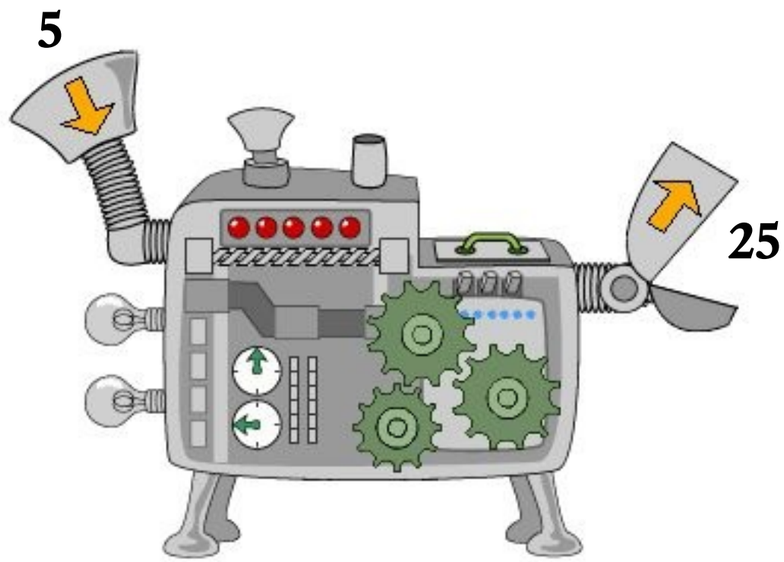
Na matemática costumamos pensar apenas em números mas, na programação, o domínio e o contradomínio podem ser muitas coisas diferentes. Exemplos:

Função	Domínio	Contradomínio
	núm. reais positivos	núm. reais positivos
	palavras, números, frases, ...	inteiro não negativo
	palavras, números, frases, ...	boolean (true ou false)
	booleanos	boolean (true ou false)
	palavras, números, frases	letra ou número

Função na programação: é uma máquina abstrata

Uma função pode ser entendida como uma máquina abstrata que recebe entradas e produz saídas. O funcionamento interno da máquina não é importante para quem está utilizando, basta saber como utilizar. ABSTRAÇÃO!





http://hzsd.ca/learningcenter/Library/Math%20Resources/00F4AD41-011EDEB3.111/040510_104650_0.gif?src=.PNG

Os 3 tipos de "máquinas": subprogramas

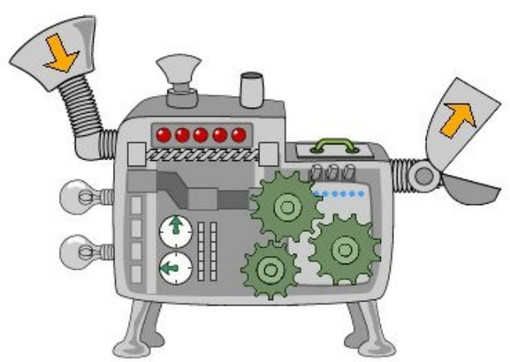
"reporter"

"predicate"

"command"

Subprogramas são blocos de ações que executam como uma unidade, escondendo seus detalhes internos (abstração). Existem três tipos principais de subprogramas:

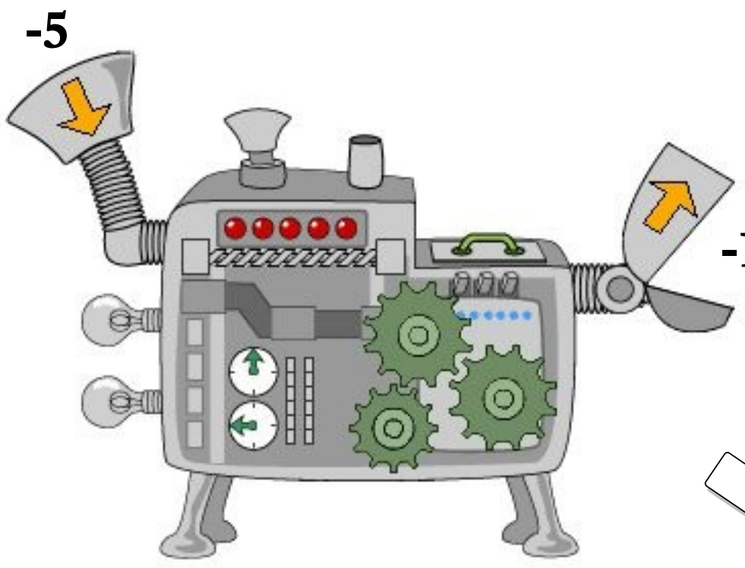
- **funções**
- **procedimentos**
- **predicados**



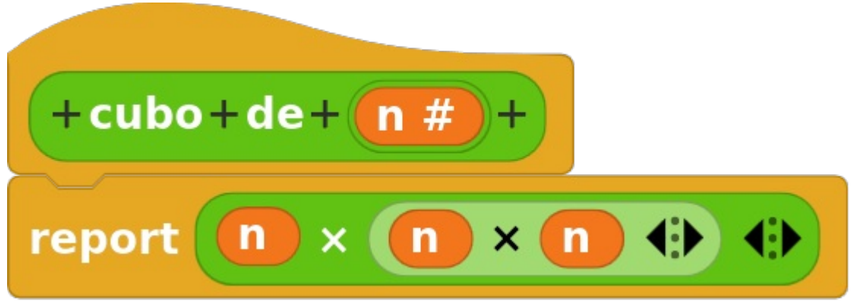
http://hzsd.ca/learningcenter/Library/Math%20F
00F4AD41-011EDEB3.111/040510_104650_0.gif

Como criar nossas próprias máquinas?

$$f : X \to Y \mid x, y \in \mathbb{R} \wedge y = f(x) = x^3$$



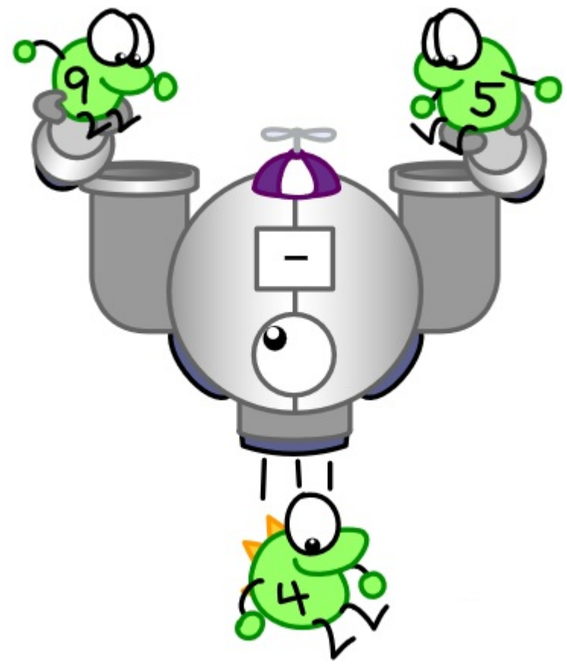
http://hzsd.ca/learningcenter/Library/Math%20Resources/00F4AD41-011EDEB3.111/040510_104650_0.gif?src=.PNG



Função como máquina abstrata: regras básicas

Regras obrigatórias da função como máquina:

- **Recebe 0 ou mais entradas**
- **Produz 1 única saída**



Computer Science Illustrated (https://csillustrated.berkeley.edu)

Outras regras desejáveis (para funções puras):

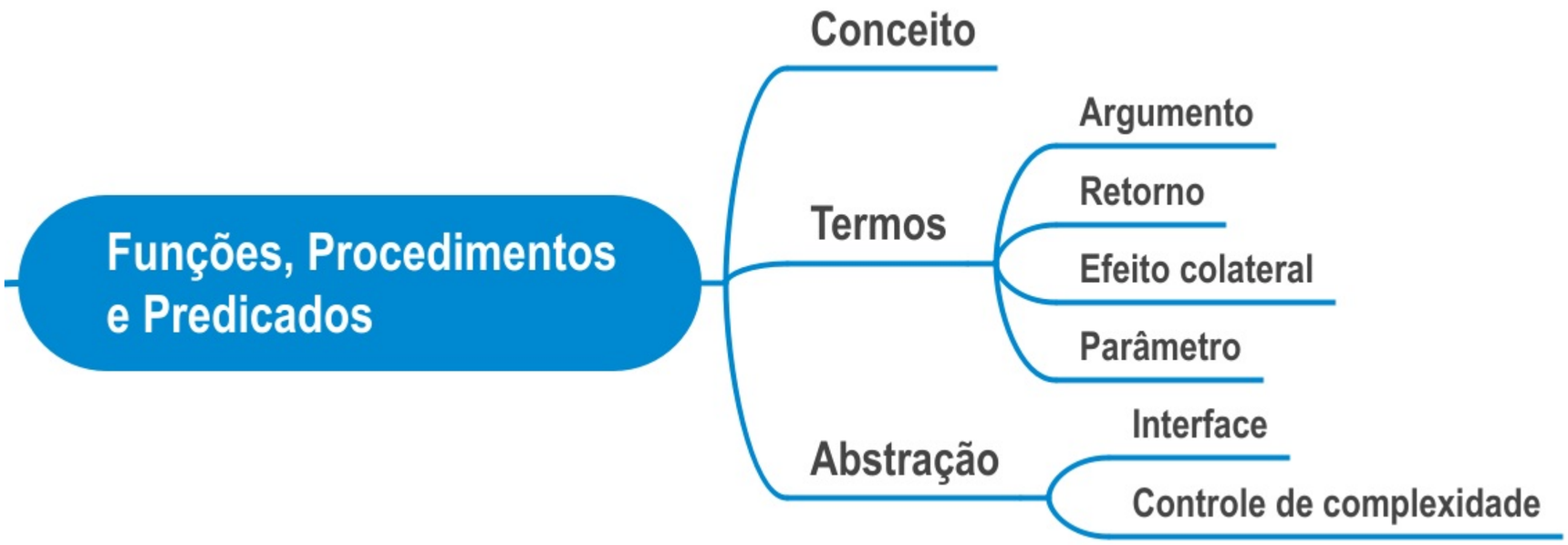
- **Determinística (as mesmas entradas devem produzir a mesma saída, ou seja a saída deve ser uma função apenas das entradas)**

- **Sem estado (não deve depender de história prévia)**

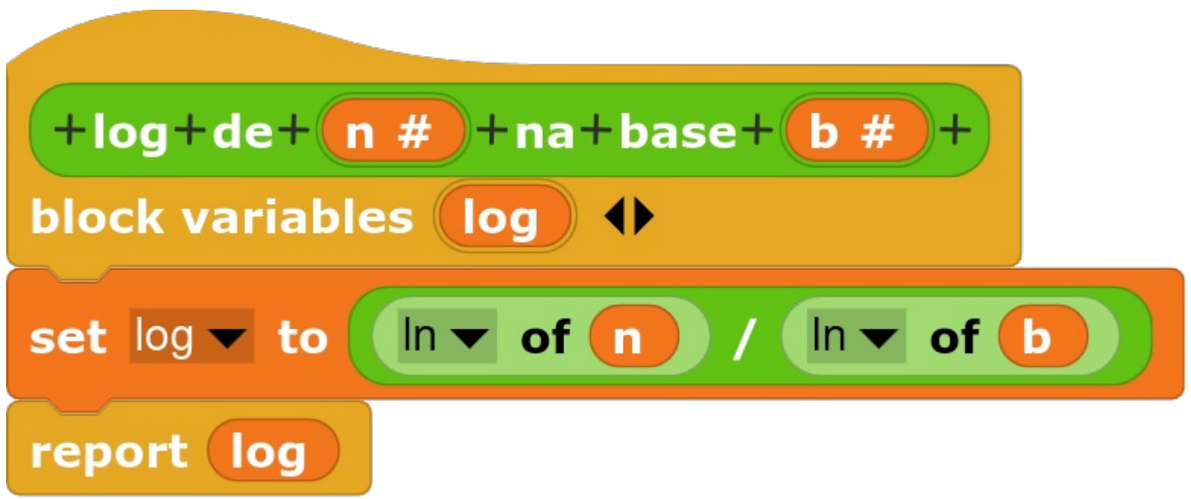
- **Sem mutação (nenhuma variável do ambiente é modificada)**

- **Sem efeitos colaterais (nada mais ocorre, só o retorno)**

Subprogramas em detalhes: funções, procedimentos e predicados



Blocos de Funções



Funções:

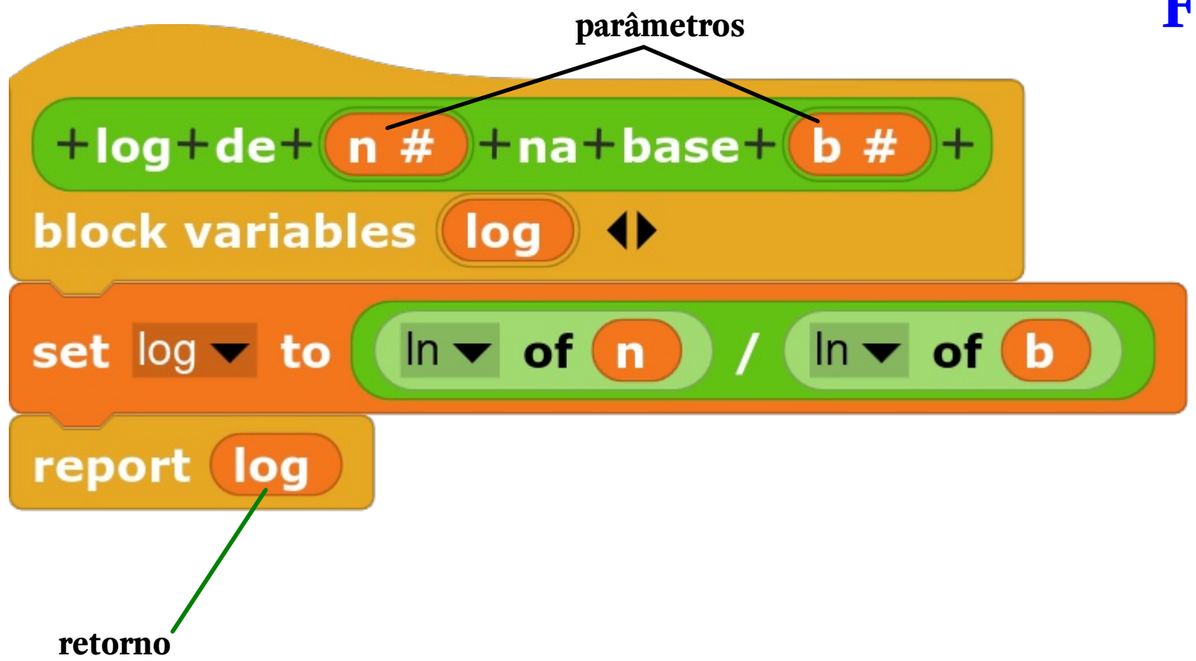
São blocos que recebem 0 ou mais entradas, executam uma ou mais ações escondendo seus detalhes internos, e retornam um único valor. Só temos que saber como usar.

- **Recebem 0 ou mais argumentos**
- **Podem realizar uma ou mais ações**
- **Podem ter efeito colateral**
- **Retornam um único valor**

Obs.: o retorno encerra a função!

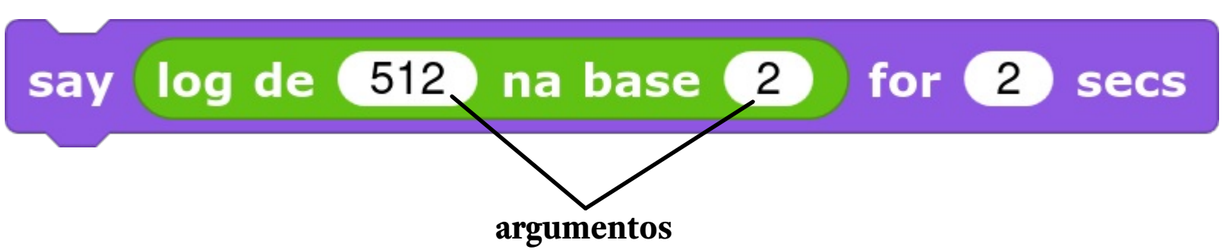
Snap*!* chama as funções de reporter.

Blocos de Funções



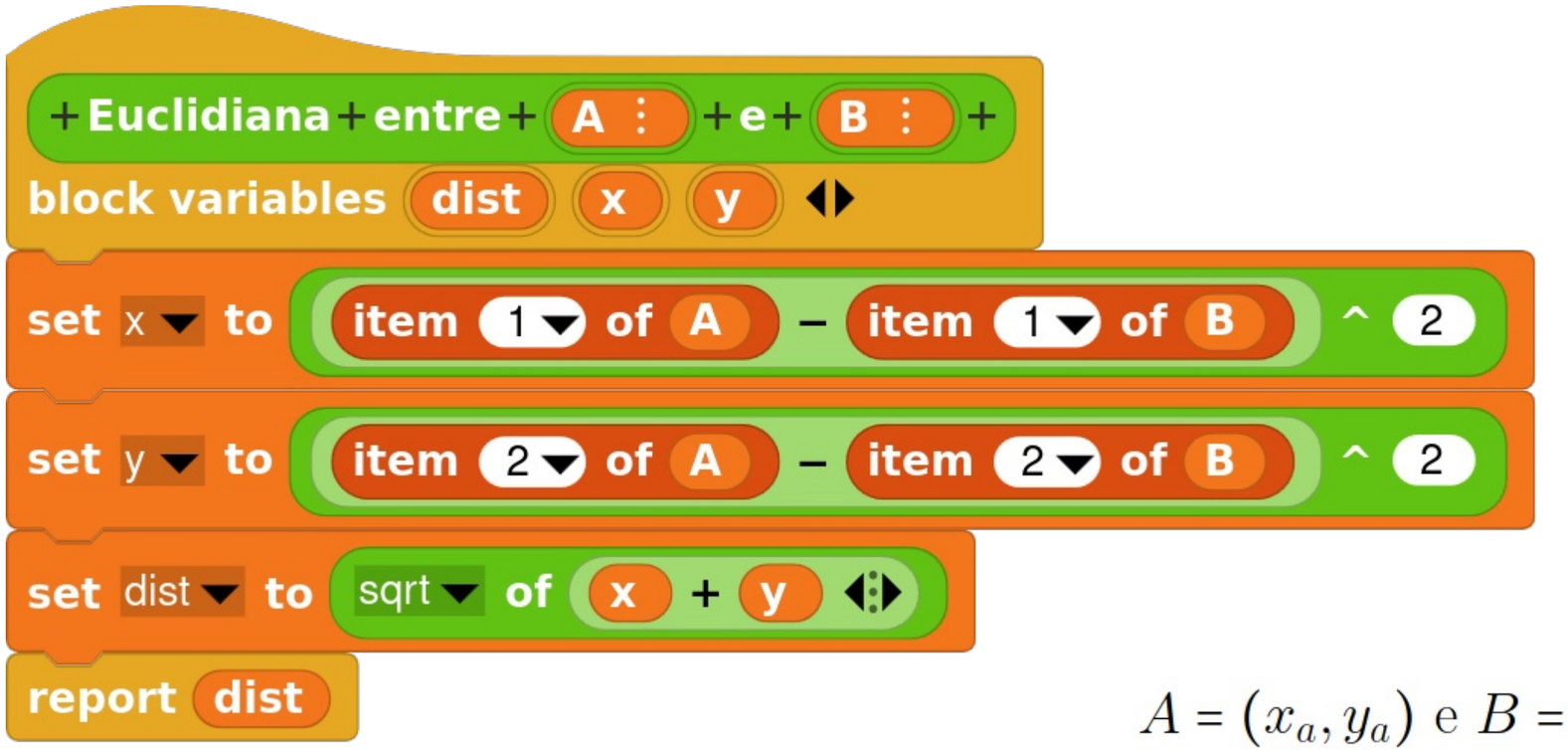
Funções:

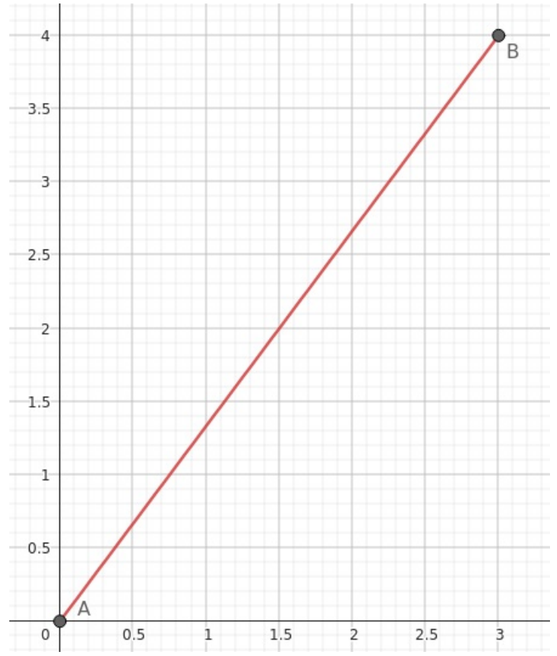
- **Argumento**: aquilo que passamos para dentro da função (entrada).
- **Parâmetros**: são as **variáveis internas** da função, que **receberão os valores dos argumentos**. Os parâmetros são utilizados no **corpo** da função.
- **Efeito colateral**: algo que a função realiza (gráfico, entrada/saída, ...) sem conexão direta ao retorno.
- **Retorno**: é o que a função retorna, devolve, para quem chamou. Uma função sempre tem um retorno (saída).



Exemplo de função: Distância Euclidiana

Calcular a distância Euclidiana entre dois pontos. Entradas: duas listas contendo, respectivamente, as coordenadas (x, y) de cada ponto. Saída: a distância Euclidiana.

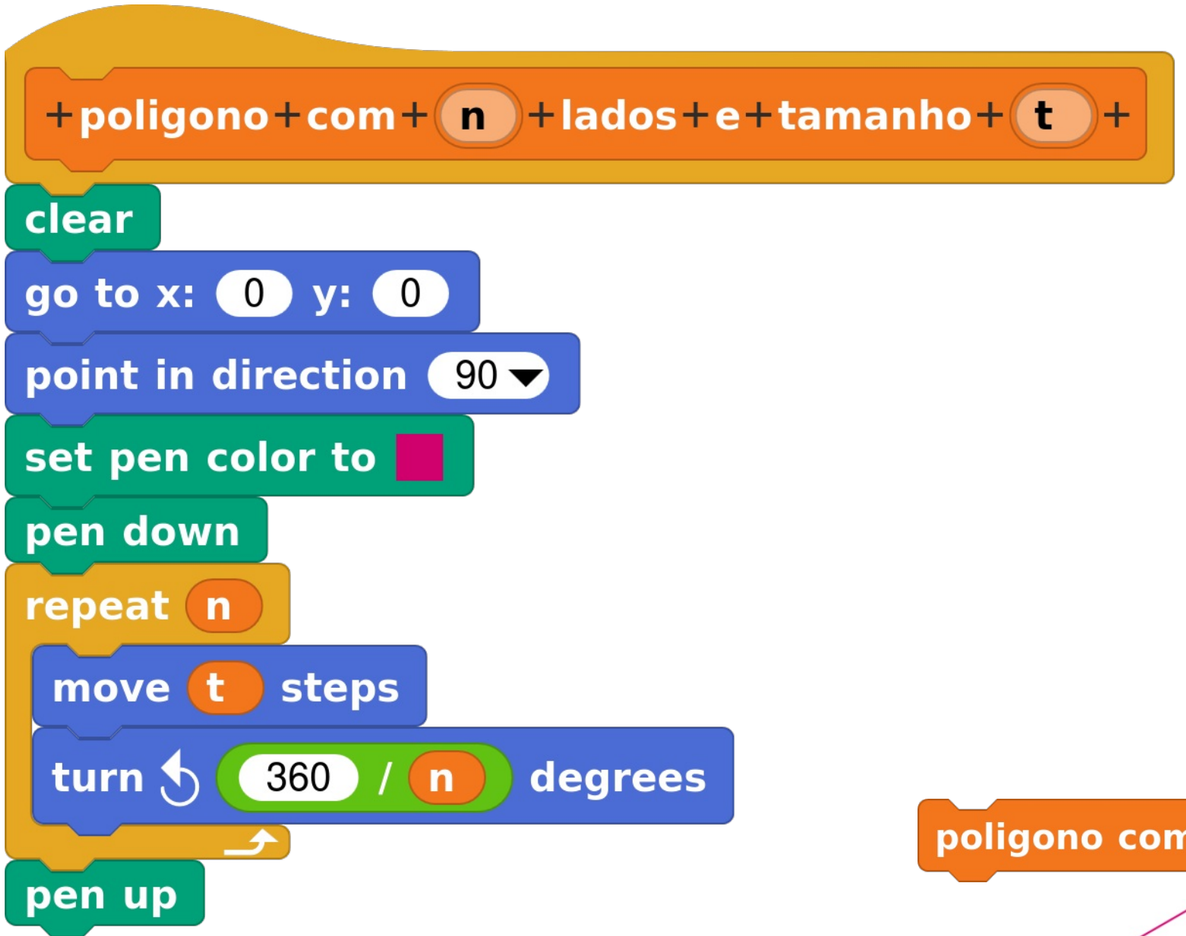




$$A = (x_a, y_a) \text{ e } B = (x_b, y_b)$$

$$d = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

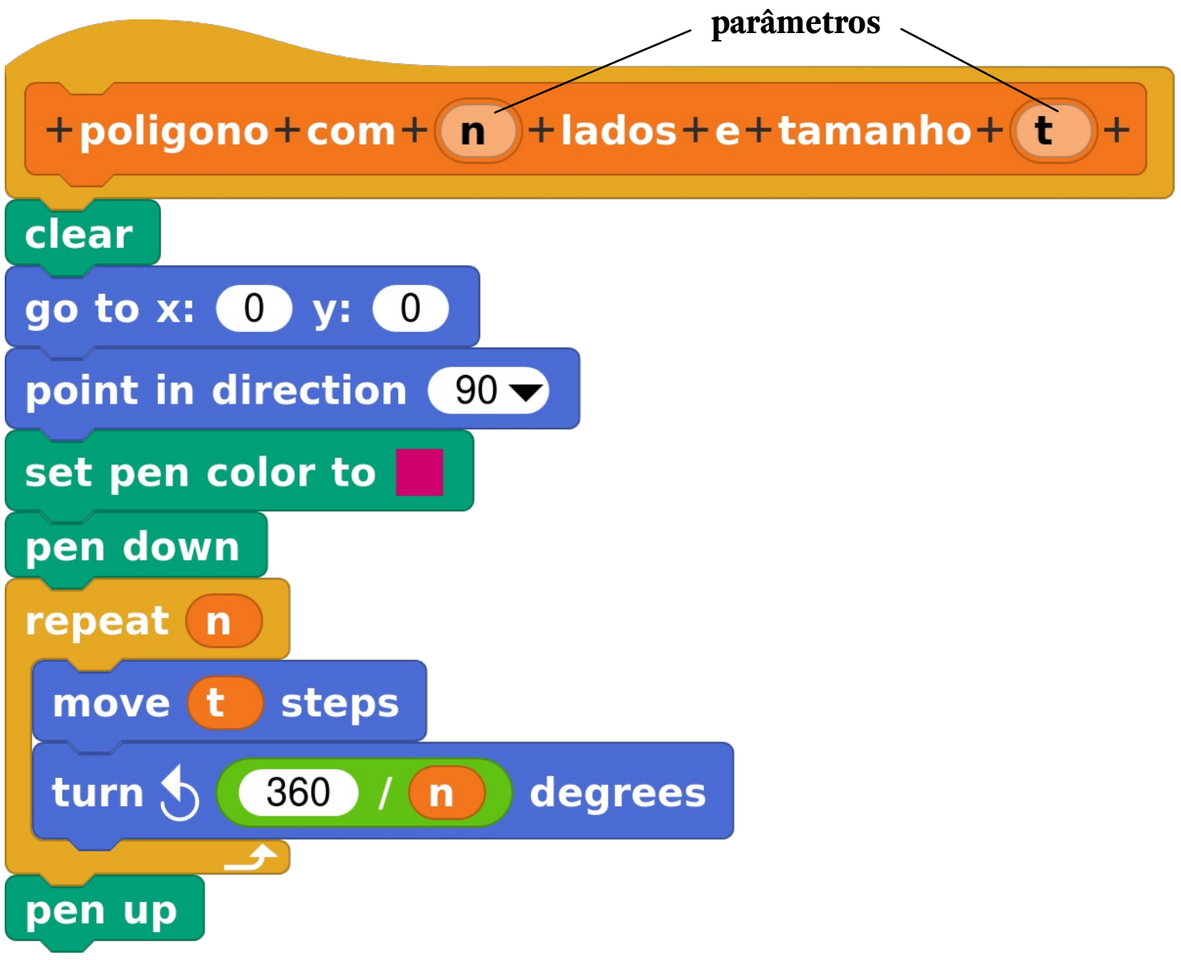
Blocos de Procedimentos



Procedimentos:
São blocos que executam uma ou mais ações, escondendo seus detalhes internos, mas não têm retorno. São usados pelos efeitos colaterais. Snap*!* chama os procedimentos de comandos.

- **Recebem 0 ou mais argumentos**
- **Podem realizar uma ou mais ações**
- **Produzem efeito colateral**
- **Não retornam nenhum valor**

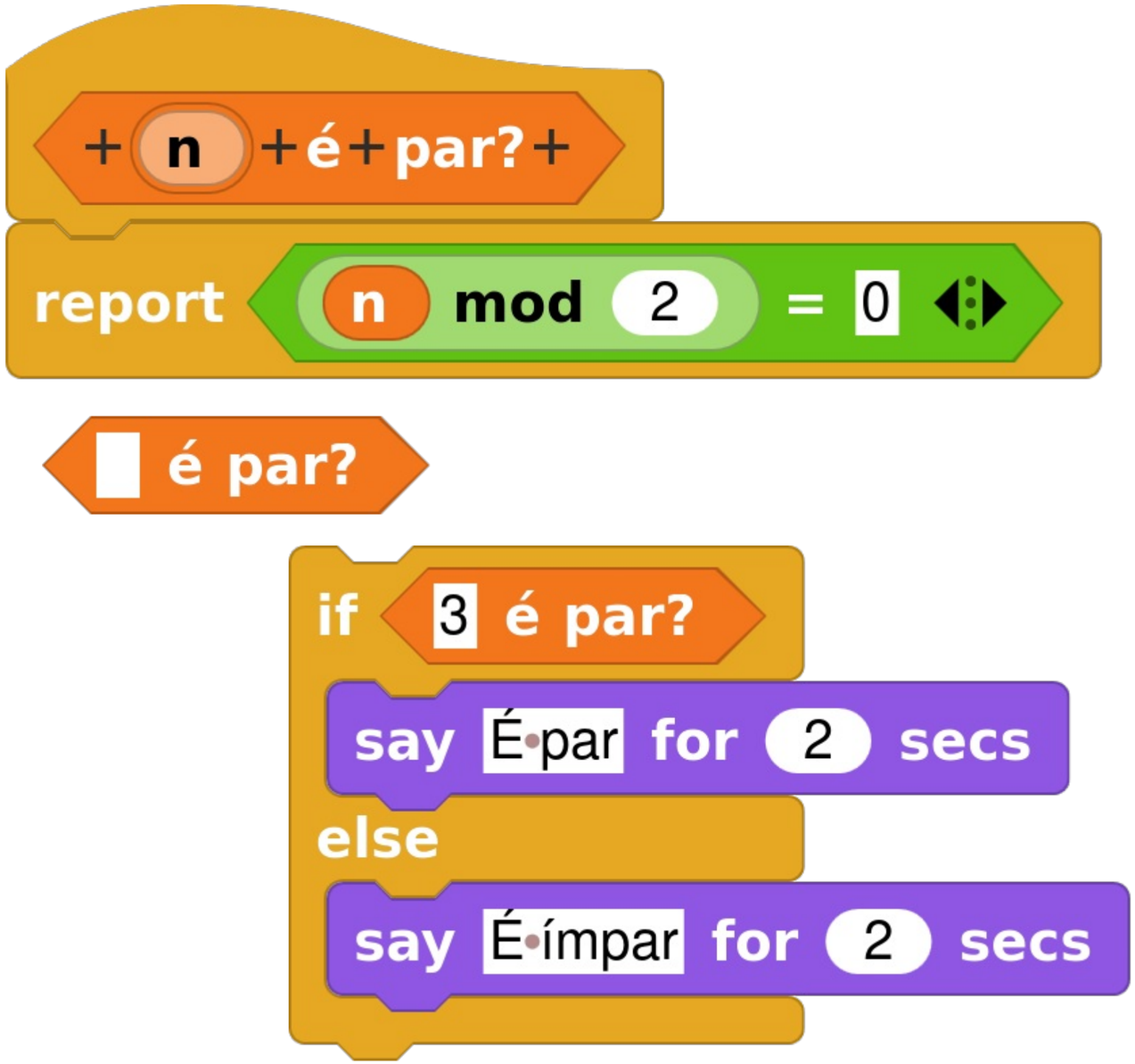
Blocos de Procedimentos



Procedimentos:

- **Argumento**: aquilo que passamos para dentro do procedimento (entrada).
- **Parâmetros**: são as **variáveis internas** do procedimento que **receberão os valores dos argumentos**. Os parâmetros são utilizados no **corpo** do procedimento.
- **Efeito colateral**: algo que o procedimento realiza (gráfico, entrada/saída, ...).
- **Retorno**: não tem, um procedimento não retorna nada para quem chamou. Não tem nenhuma saída.

Blocos de Predicados



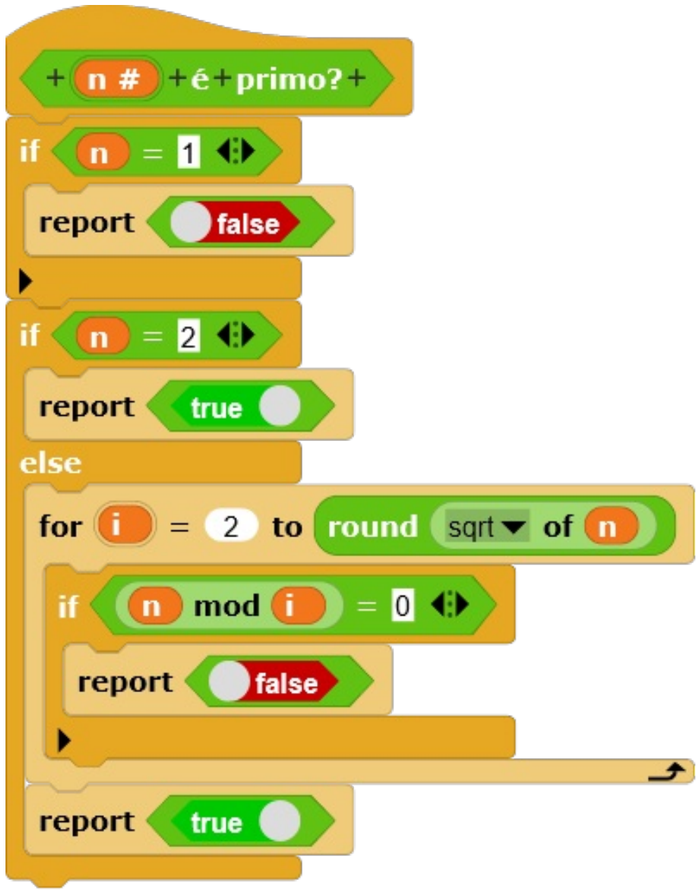
Predicados:
São funções que somente retornam VERDADEIRO (true) ou FALSO (false). Snap*!* chama aos predicados de predicate.

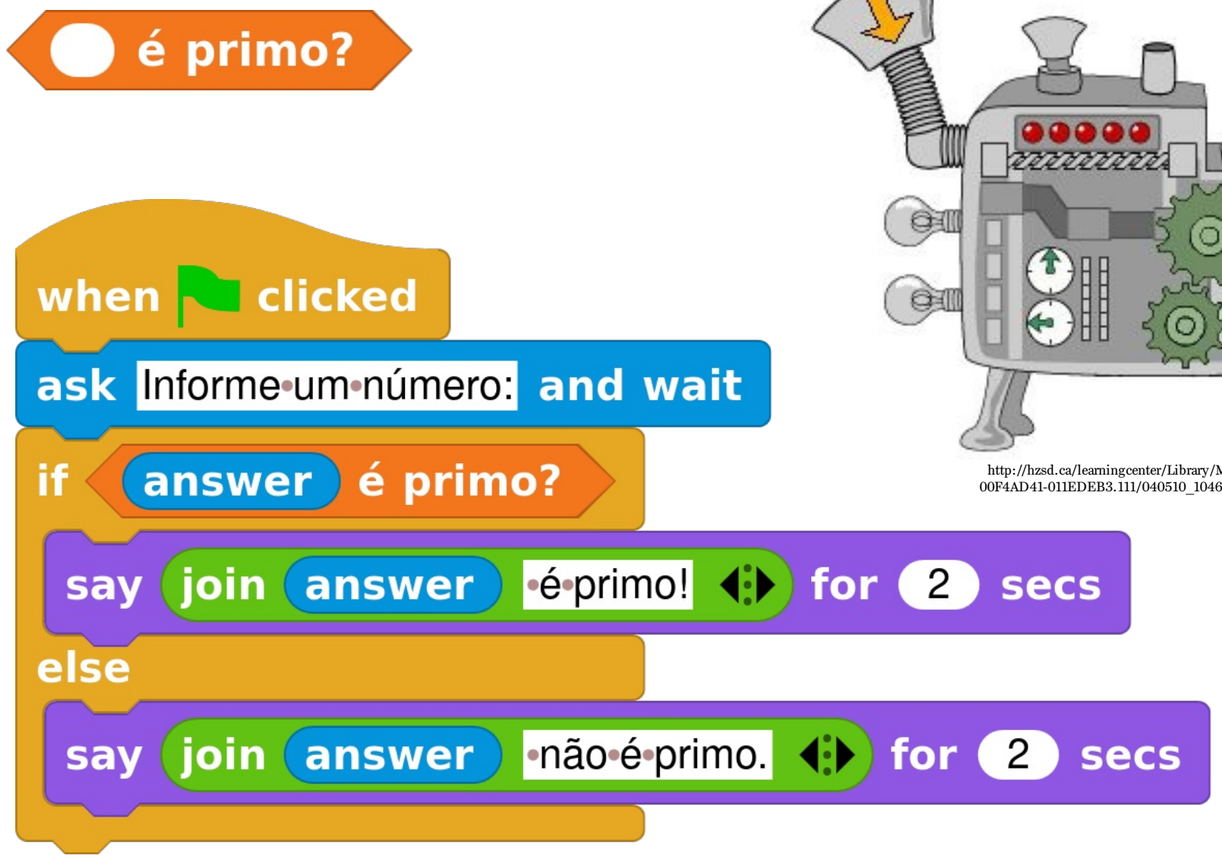
- **Recebem 0 ou mais argumentos**
- **Podem realizar uma ou mais ações**
- **Podem ter efeito colateral**
- **Retornam true ou false**

Obs.: o retorno encerra o predicado.

Subprogramas (procedimentos, funções ou predicados) x abstração

Subprogramas são fundamentais para a abstração e controle da complexidade na resolução de problemas complexos! Servem como blocos básicos de construção de nossos programas, permitem generalizar soluções e podem ser combinados para criar coisas mais complexas e sofisticadas.





http://hzsd.ca/learningcenter/Library/Math%20Resources/00F4AD41-011EDEB3.111/040510_104650_0.gif?src=.PNG

Subprogramas são fundamentais para abstração!

Acima da barreira da abstração: o usuário não se importa em como a distância Euclidiana é calculada, assume que o cálculo é correto. Só precisa saber como usar a função, p. ex.: quais devem ser as entradas?

Interface (barreira da abstração)

Abaixo da barreira da abstração: o programador cria a função de modo a garantir que ela funcionará corretamente, conforme o esperado, desde que utilizada do modo correto (entradas corretas). O programador não se importa em como e para que o usuário utilizará a função.

Subprogramas são fundamentais para abstração!

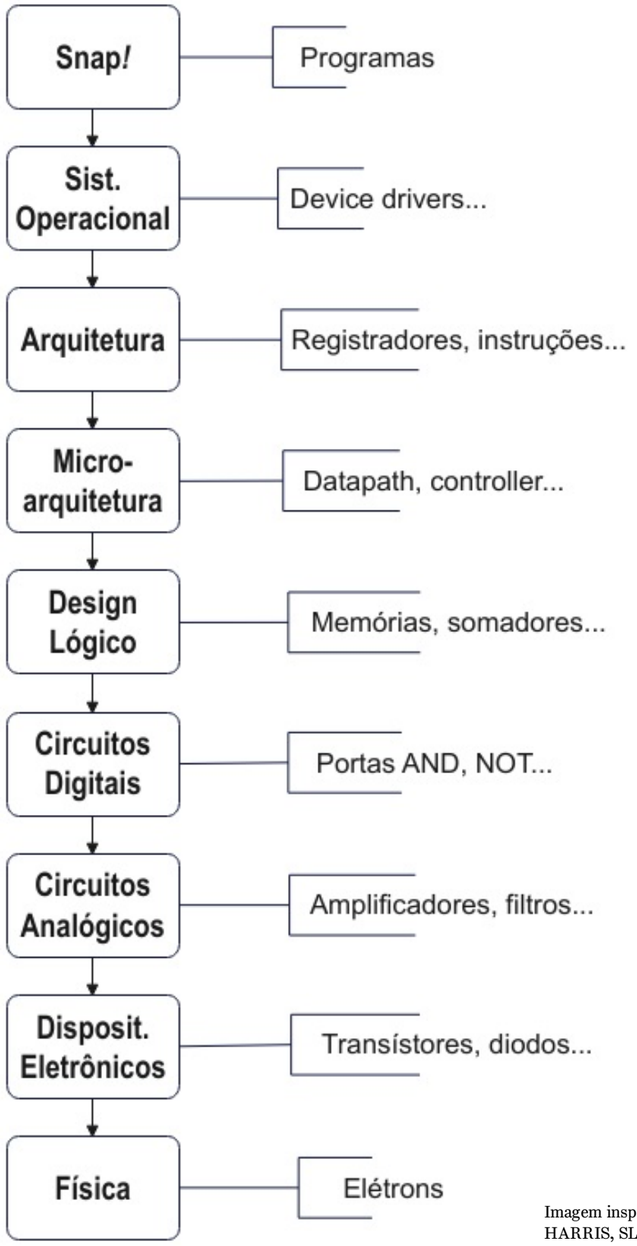
A interface (barreira da abstração) é o "contrato" entre quem utilizará a função e quem criou a função, e é essa barreira que esconde os detalhes internos. O contrato que a interface cria estabelece que a função funcionará de modo correto, sem que o usuário precise se preocupar com os detalhes internos, desde que esse usuário utilize a função de acordo com especificações (entradas, valores, etc.)

Interface (barreira da abstração)

Exemplos de interfaces (contratos):

- **Blocos no Snap!**
- **Pedais no carro**
- **Contratar uma construtora para uma casa**
    - **Contratar mestre de obras**
        - **Contratar pedreiros**
            - **Contratar ajudantes**

Abstrações são fundamentais para o controle da complexidade!



Podemos combinar níveis de abstrações cada vez mais altos para o controle da complexidade.

Se não fosse pelas abstrações, como conseguiríamos construir um computador? Como o progresso tecnológico ocorreria?

Imagem inspirada e adaptada de:
HARRIS, SL & HARRIS, D. Digital Design and Computer Architecture: RISC-V Edition. 1ª ed. Cambridge: Morgan Kaufmann, 2022.

Funções (e predicados) x Procedimentos

Certifique-se de que a diferença entre funções/predicados e prodecimentos está clara para você:

Funções/Predicados

- **retornam um valor**
- **podem ser combinados (composição funcional)**
- **não deveriam ter efeitos colaterais**

Procedimentos

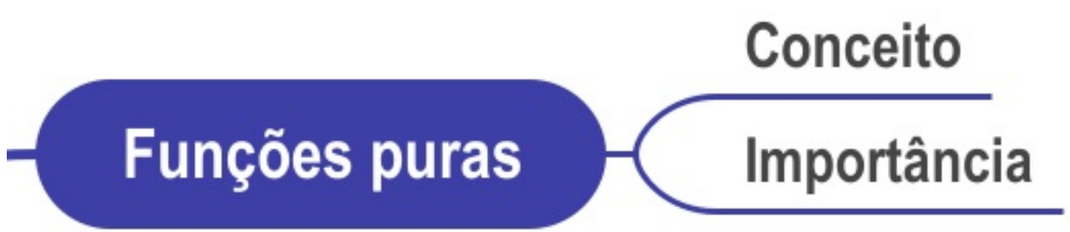
- **não retornam nada**
- **não podem ser combinados (por quê?)**
- **são usados pelos efeitos colaterais**

Por que usar subprogramas? Um resumo!

- Abstração e controle da complexidade
- Generalização
- São blocos básicos a partir dos quais podemos construir coisas complexas
- Podem ser combinados (funções/predicados)

Como saber quais subprogramas criar? PENSAMENTO COMPUTACIONAL!

Funções puras



Funções puras: o que está ocorrendo aqui?

Considere o que está ocorrendo com a função "diga x", abaixo:

Funções puras: conceito

Uma função pura é aquela que tem as seguintes propriedades:

- **Recebe 0 ou mais entradas**
- **Produz 1 única saída**
- **Determinística:**

  As mesmas entradas devem produzir a mesma saída, ou seja, a saída deve ser uma função apenas das entradas.
- **Sem efeitos colaterais:**

  Nada mais ocorre, não pode afetar o "universo" de nenhum outro modo a não ser o retorno. Só o retorno ocorre.
- **Sem mutação:**

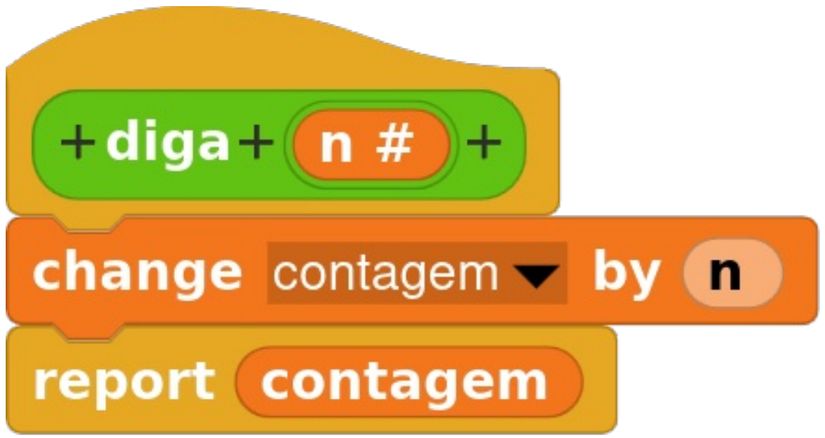
  Não pode alterar o estado de nada no "universo", só o que pertence à própria função, não modifica nenhuma variável do ambiente.
- **Sem estado:**

  Não deve depender de história prévia, do estado prévio de nada no universo nem do seu próprio estado prévio.

Funções puras: o que está ocorrendo aqui?

A função "diga x" não é uma função pura! Quais os problemas?

Imprevisível!
Quebra a interface!
Tem efeito colateral!



Funções puras: por que são importantes?

Se uma função não é pura:

- **comportamento incerto (depende do universo ou do passado)**
- **quebra a barreira da abstração (o usuário precisa saber algo do interior da função para conseguir entender o que está ocorrendo)**
- **aumenta a complexidade (e a complexidade é nossa inimiga)**
- **não é mais determinística (mesmas entradas fornecem saídas diferentes)**

Se a função é pura:

- **comportamento previsível (só depende dos argumentos)**
- **é determinística**
- **não tem efeitos colaterais, não afeta o universo, exceto pelo retorno**
- **não causa mutação**
- **não mantém estado (histórico)**

Debate: até que grau as funções devem ser puras?

Por que é importante evitar efeitos colaterais?

Exemplo retirado de: CS10/fa21. Imagine que você está escrevendo o código do Super Maria Brothers, desse modo:

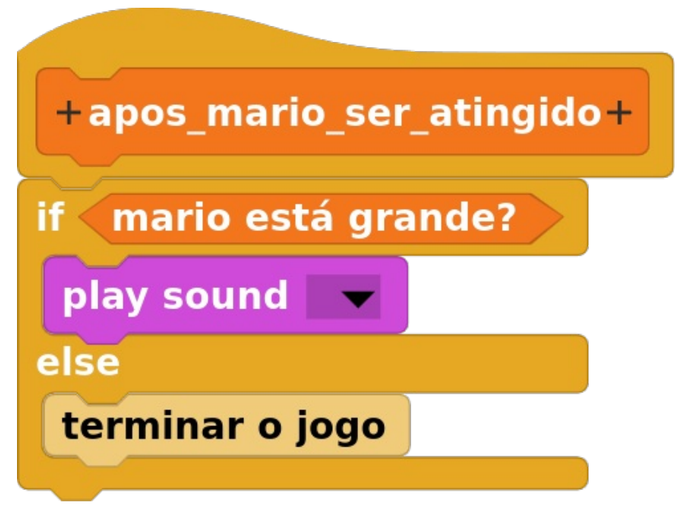
- Quando o Mario consegue um item, ele muda seu estado:
- cogumelo: aumenta de tamanho
- flor de fogo: aumenta de tamanho e permite que atire fogo
- folha: aumenta de tamanho e permite que ele voe

- Quando ele é atingido por um inimigo:
- se estiver grande: perde o item e volta ao tamanho normal
- se estiver normal: perde o jogo

normal **cogumelo** **flor de fogo** **folha**

Imagens do Mario em: https://cs10.org/fa21/

Por que é importante evitar efeitos colaterais?

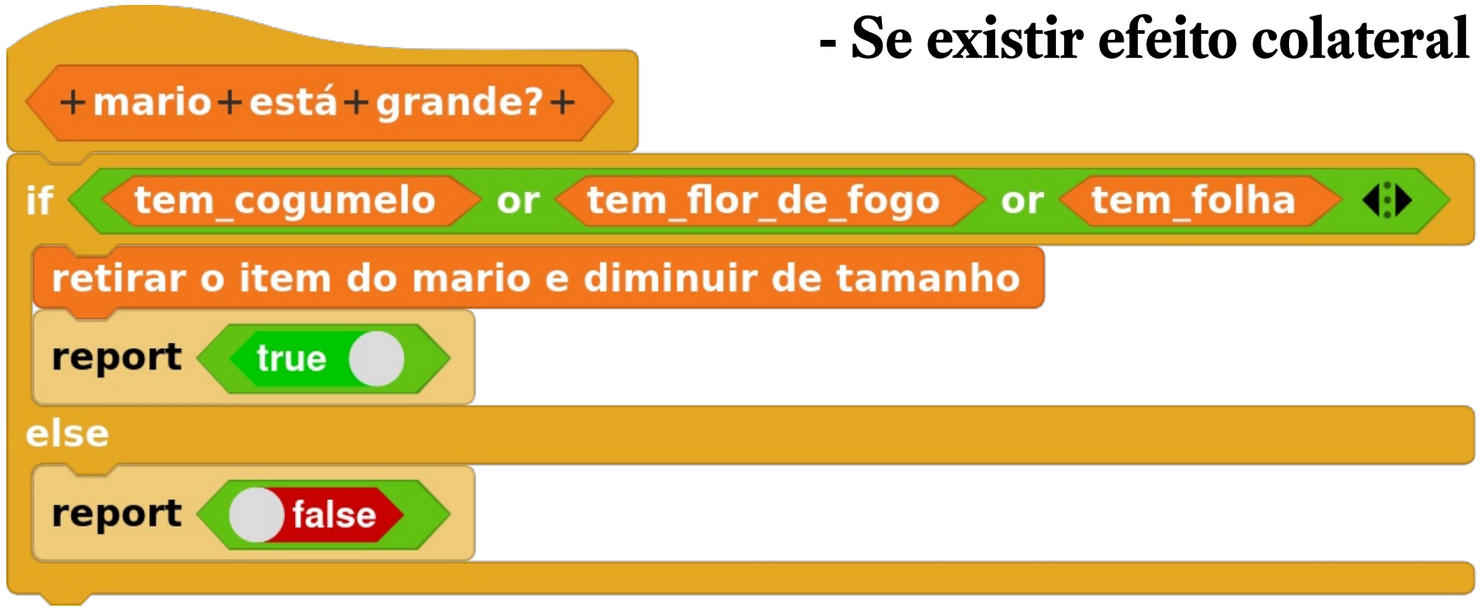


Um aluno implementou a funcionaidade desejada utilizando:

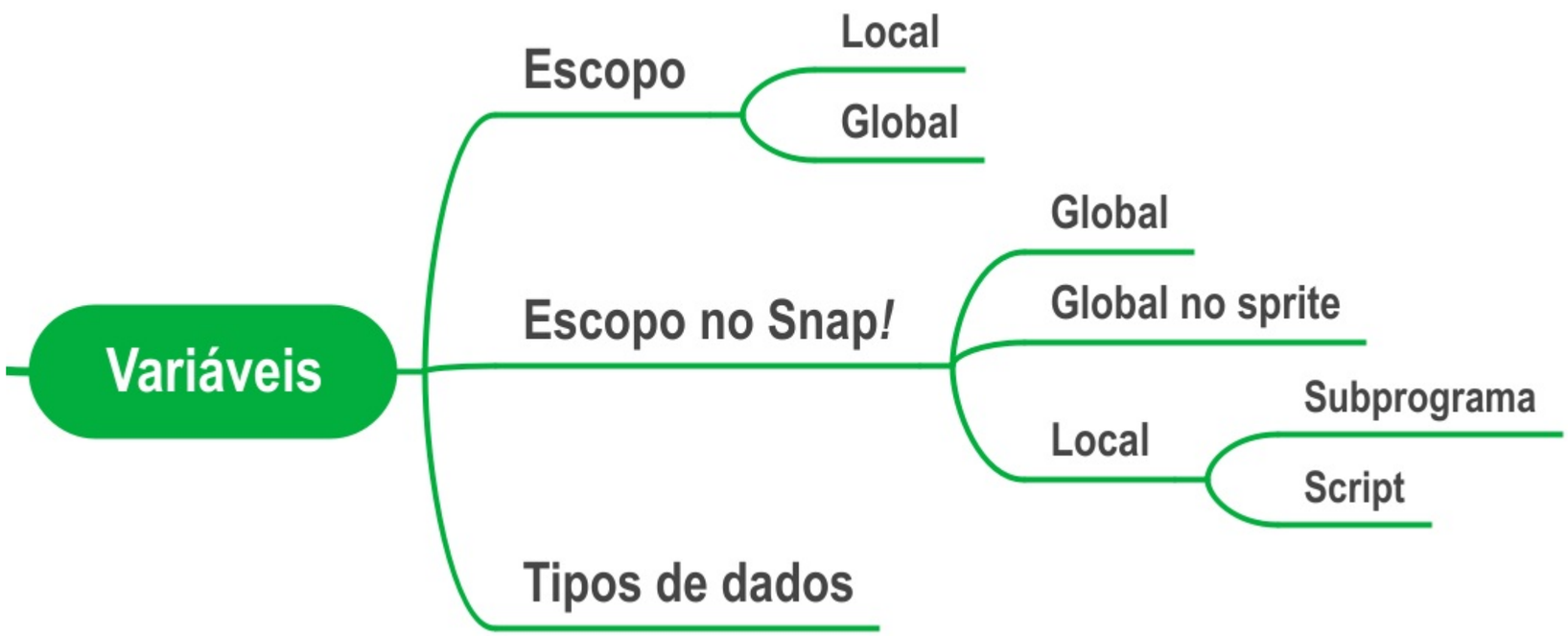
- **procedimento: apos_mario_ser_atingido**
- **predicado: mario_está_grande?**

Analise o código do aluno e responda:

- **O código está correto, implementa a funcionalidade?**
- **Existe efeito colateral nesse código?**
- **Se existir efeito colateral, ele é perigoso ou inocente?**
- **Se existir efeito colateral e ele for perigoso, como corrigir?**



Variáveis


Variáveis
Escopo
Local
Global
Escopo no Snap!
Global
Global no sprite
Local
Subprograma
Script
Tipos de dados

Variáveis: escopo

Uma variável é uma estrutura de dados que nos permite armazenar um valor na memória e, ao mesmo tempo, associá-lo a um nome. Nós usamos o nome da variável para acessar o valor armazenado em diversas partes do programa.

Entretanto, uma variável pode não ser acessível em todas as partes do programa, isso depende do escopo da variável. O escopo de uma variável corresponde ao trecho do programa no qual uma variável é visível e pode ser acessada. Dependendo do escopo, uma variável pode ser:

Em geral:
- global
- local

No Snap!:
- global
- global de sprite
- local:
  - subprogramas
  - script

Variáveis com escopo global

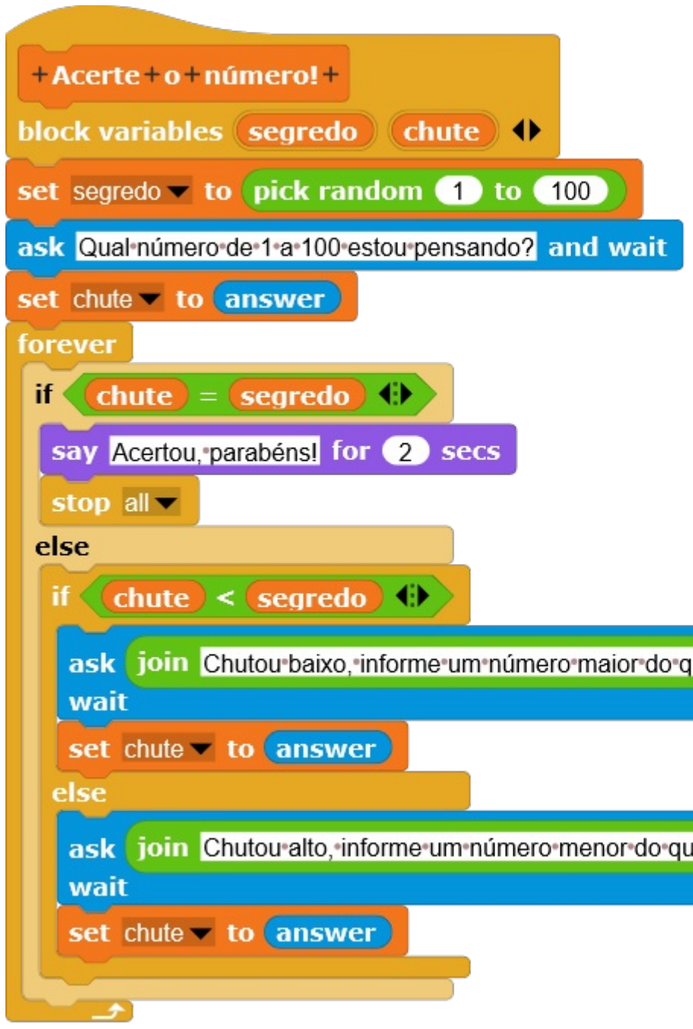
São acessíveis em todos os scripts em todos os sprites.

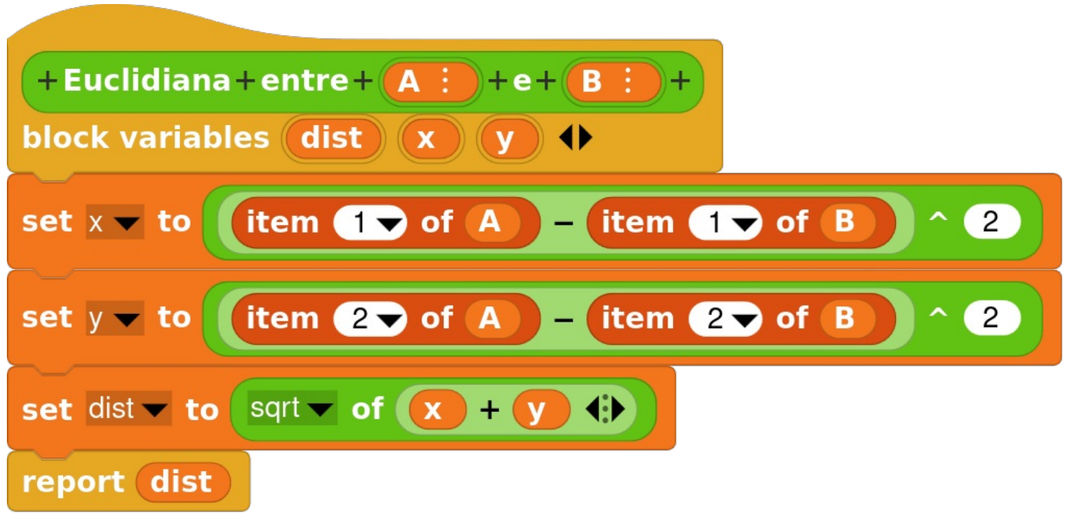
Variáveis com escopo global no sprite

São acessíveis em todos os scripts dentro de um sprite específico.

Variáveis com escopo local, em subprogramas

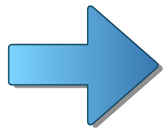
São acessíveis somente dentro do subprograma (função, procedimento ou predicado) onde foram criadas. Correspondem aos parâmetros e as demais variáveis do subprograma.

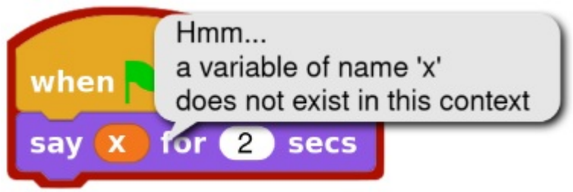




Variáveis com escopo local, em scripts

O Snap*!* conta com outro tipo de variável local, com escopo restrito ao script onde foi criada, mesmo que não seja dentro de um subprograma.





Variáveis: tipos de dados

Conceitualmente, no mundo real, podemos pensar sobre os dados como sendo de um determinado tipo, por exemplo:

- números reais	**1/3, pi, 4, -5.34, 3.7, …**
- números inteiros	**-3, 0, 8, …**
- caracteres	**A, B, 0, 3, @, &, …**
- palavras	**casa, avião, computador, …**
- frases	**a casa pegou fogo**
- booleanos	**verdadeiro, falso**

Variáveis: tipos de dados na programação

Na programação um tipo de dado é um conjunto de valores associado a um conjunto de operações que podem ser executadas sobre esses valores:

tipo de dado = conjunto de valores + conjunto de operações

Ex.: tipo inteiro (int):

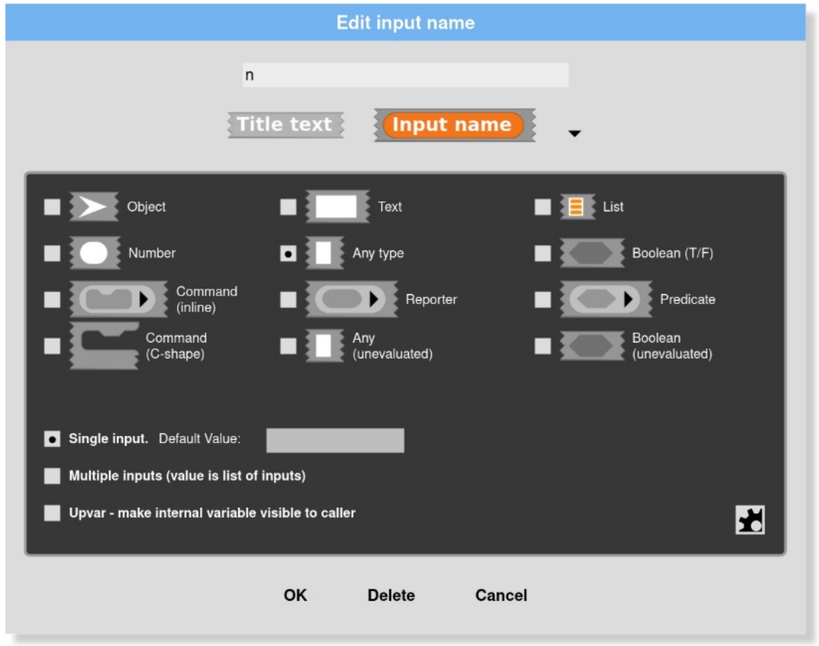
Conjunto de valores	Operações:
...	
-100	
-1	
0	
1	
100	
...	
	
	
	

Variáveis: tipos de dados na programação

Snap*!* tem 4 tipos de dados básicos formalmente definidos, que podem ser utilizados ao se especificar os parâmetros de subprogramas (até a versão atual não se pode especificar tipos de dados em outros locais).

- **número**
- **texto**
- **lista**
- **qualquer**

A forma dos parâmetros mostra o tipo de dados esperado (é possível usar o tipo "errado", mas isso não é um bom estilo de programação):



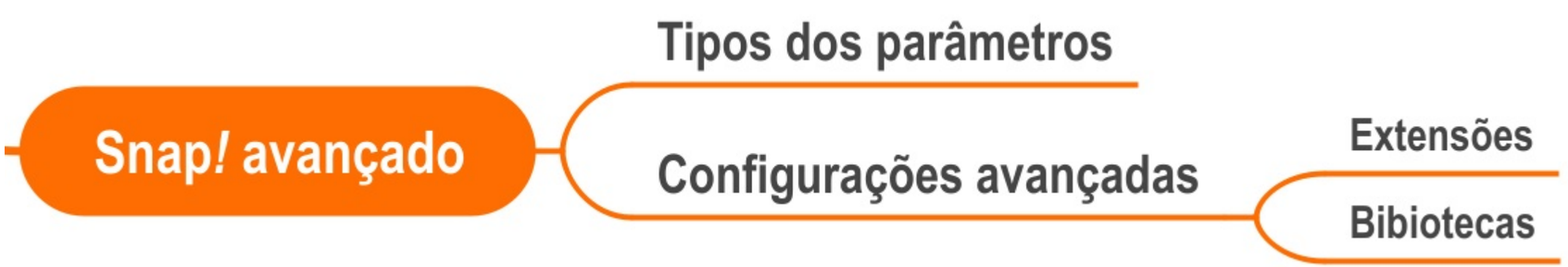
números:
oval

texto:
retângulo largo

lista:
3 retângulos empilhados

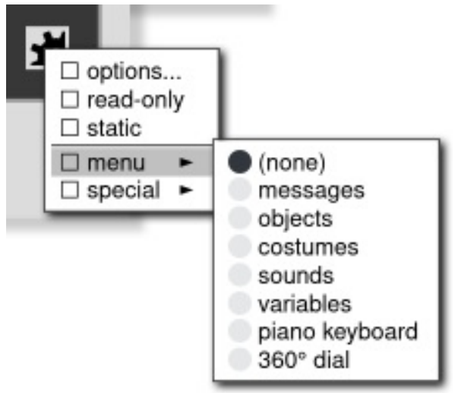
qualquer:
retângulo estreito

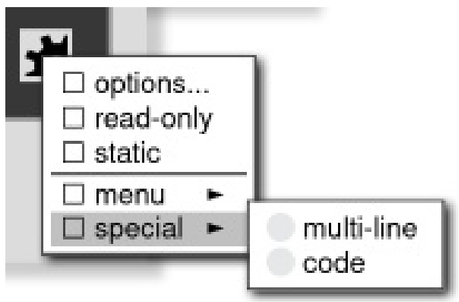
Características avançadas do Snap!



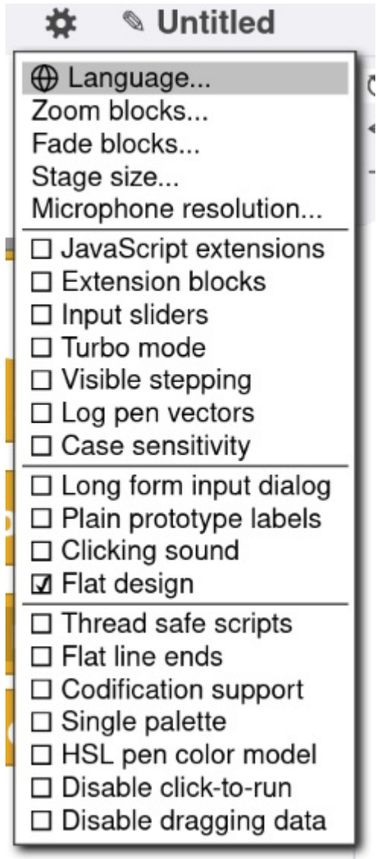
Detalhes ao definir blocos

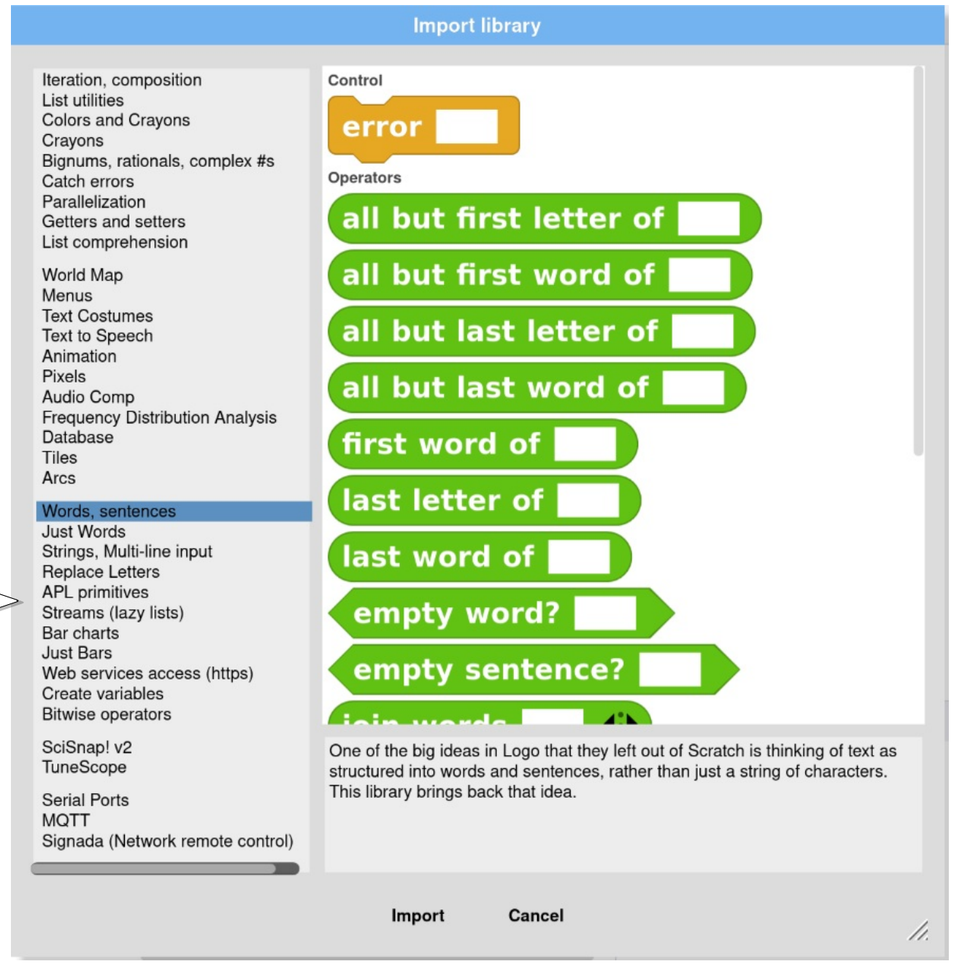
Por padrão, os parâmetros no Snap*!* aceitam qualquer tipo de dado. Você pode explicitamente restringir isso:



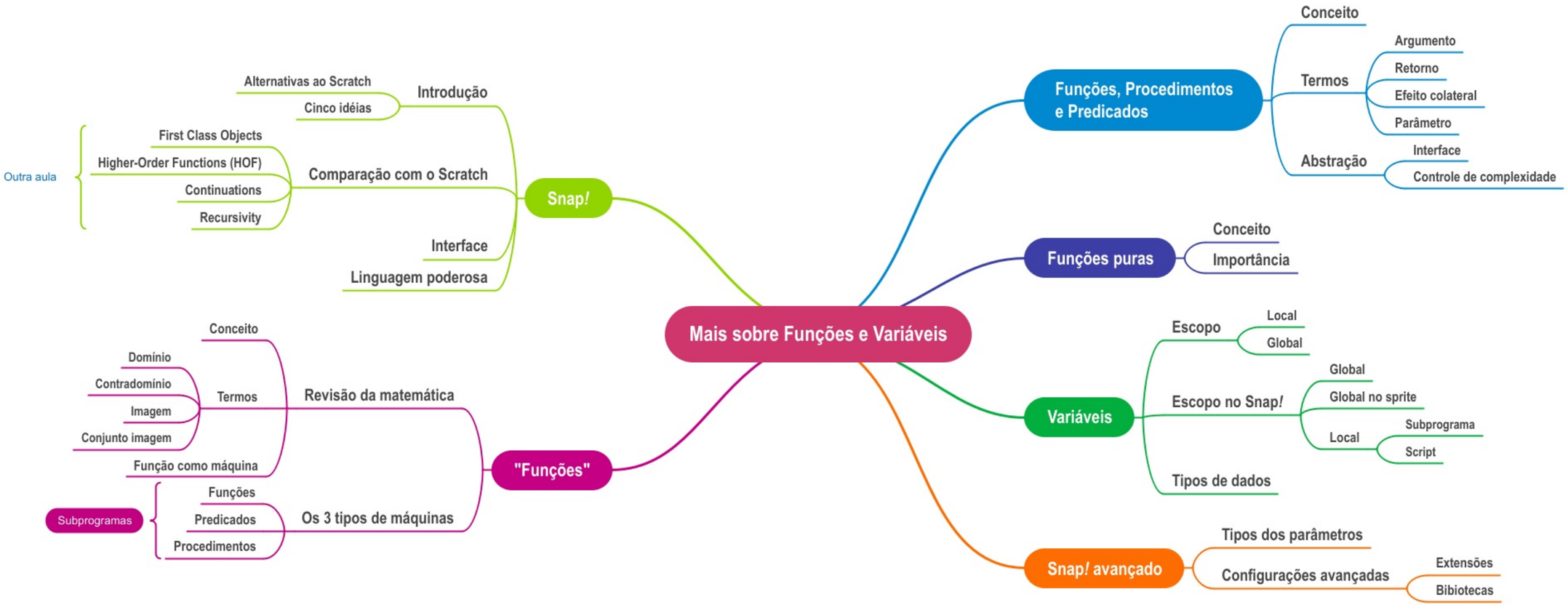


Muitas configurações avançadas





Em resumo


Mais sobre Funções e Variáveis
Snap!
Introdução
Alternativas ao Scratch
Cinco idéias
Comparação com o Scratch
Outra aula
First Class Objects
Higher-Order Functions (HOF)
Continuations
Recursivity
Interface
Linguagem poderosa
"Funções"
Revisão da matemática
Termos
Domínio
Contradomínio
Imagem
Conjunto imagem
Conceito
Função como máquina
Os 3 tipos de máquinas
Subprogramas
Funções
Predicados
Procedimentos
Funções, Procedimentos e Predicados
Conceito
Termos
Argumento
Retorno
Efeito colateral
Parâmetro
Abstração
Interface
Controle de complexidade
Funções puras
Conceito
Importância
Variáveis
Escopo
Local
Global
Escopo no Snap!
Global
Global no sprite
Local
Subprograma
Script
Tipos de dados
Snap! avançado
Tipos dos parâmetros
Configurações avançadas
Extensões
Bibiotecas