Desenhando com com Turtle Graphics

APRENDA E ENSINE **MATEMÁTICA**
E **PROGRAMAÇÃO** COM O **Blockly**,
A LINGUAGEM DE BLOCOS DO GOOGLE

UFPR

Helio H. Monte Alto

DESENHANDO COM TURTLE GRAPHICS

Aprenda e ensine Matemática e Programação com o Blockly, a linguagem de blocos do Google

Helio H. Monte-Alto

2024

Elaboração e Organização:

Prof. Dr. Helio Henrique Lopes Costa Monte-Alto

Apoio:

UFPR
UNIVERSIDADE FEDERAL DO PARANÁ

UFPR
JANDAIA DO SUL

Sumário

Apresentação

Este livro propõe uma abordagem de ensino cujo objetivo principal é auxiliar professores que queiram utilizar linguagens de programação do tipo Logo (também conhecidas como *Turtle Graphics*, ou Gráficos de Tartaruga) em sala de aula, de modo a introduzir conceitos matemáticos importantes de **geometria** e **álgebra**, e, ao mesmo tempo, estimular e desenvolver o **pensamento computacional** de estudantes da Educação Básica. No entanto, o estudante individual de nível superior pode também tirar proveito do material apresentado, especialmente o estudante de Ciência da Computação que deseja conhecer novas perspectivas de **aprendizagem de algoritmos e programação**.

A fim de proporcionar ao leitor a possibilidade de explorar a abordagem utilizando tecnologias que sejam mais adequadas a diferentes faixas etárias e níveis de conhecimento e habilidades com computadores, o material apresentado é dividido em múltiplos volumes distintos, com conteúdo teórico similar, porém utilizando diferentes tecnologias. Dois volumes estão sendo lançados inicialmente: o primeiro utiliza uma ferramenta inovadora e lúdica, baseada em uma linguagem de blocos no estilo do Scratch, o **Blockly Turtle**. O segundo utiliza uma das linguagens de programação mais populares nos dias atuais: o **Python**. Na introdução de cada volume, uma breve introdução da tecnologia utilizada é apresentada a fim de direcionar a escolha mais adequada para cada leitor. Todos os volumes estarão disponíveis no Acervo Digital - Biblioteca Temática: REA/PEA UFPR[1], bastando pesquisar pelo nome do livro (“Desenhando com Turtle Graphics”) e/ou pelo nome do autor.

O conteúdo do livro inclui, além do tutorial básico mais focado no ensino de geometria e álgebra, alguns exemplos mais avançados, como os que envolvem o desenho de estrelas, espirais e fractais. Esse conteúdo introduz conceitos um pouco mais avançados de Algoritmos e Estruturas de Dados, como recursão, *strings*, listas e pilhas. Dessa forma, o estudante pode se aprofundar para além do conteúdo mais focado na transferência de aprendizado entre Matemática e Computação, que é a proposta principal e basilar deste livro.

[1] https://acervodigital.ufpr.br/

Agradecimentos

Parte do conteúdo deste livro teve como ponto de partida uma apostila de programação para professores de Matemática desenvolvida para o Curso de Capacitação de Professores de Matemática para o Ensino com o Apoio de Programação de Computadores, realizado entre setembro e dezembro de 2016 na Universidade Federal do Paraná - Setor Palotina, de autoria do presente autor com a colaboração dos seguintes coautores: Rafael Garcia Cerci, Cassiele Thais dos Santos, Júlio Cezar da Silva Ferreira, e dos professores Marcos Antonio Schreiner e Eliana Santana Lisbôa. Aquela apostila apresentava um tutorial de forma similar ao apresentado neste livro, mas de forma mais simplificada e utilizando tecnologias diferentes. Fica, portanto, meu agradecimento aos demais autores daquela apostila, que foi o ponto de partida para a elaboração deste livro de forma mais completa, revisada e expandida.

O que é *Turtle Graphics*?

Turtle Graphics (Gráficos de Tartaruga) é um método de programação de desenhos vetoriais que utiliza um cursor relativo (a "tartaruga") sobre um plano cartesiano. Esse método é a base da linguagem de programação Logo, originalmente concebida por Seymour Papert[2] como uma linguagem que controla um "robô tartaruga" com o objetivo educacional de introduzir o pensamento computacional para crianças, sendo o precursor do que atualmente é conhecido como Robótica Educacional. Papert descreve a tartaruga como um "objeto-para-se-pensar-com" (do inglês, *object-to-think-with*), no sentido de ser um objeto que pode ser programado de forma tão intuitiva que a pessoa consegue se colocar no lugar dele, como se ela mesma estivesse desenhando com uma grande caneta sobre uma superfície. Assim, o "robô tartaruga" projetado por Papert utiliza uma pequena caneta retrátil, acoplada ao robô, que permite desenhar sobre uma superfície enquanto o robô se move. A linguagem Logo, por sua vez, contém comandos específicos para movimentar o robô, que são usados para construir algoritmos (sequências de instruções) capazes de desenhar diversas formas geométricas e desenhos. Essas instruções são então transmitidas para o robô, que as executa, realizando os movimentos e, consequentemente, desenhando com a caneta sobre a superfície — o próprio chão ou uma mesa forrada com algum tipo de papel, dependendo do tamanho do robô.

Após os trabalhos originais de Papert, a ideia se disseminou, e diversos softwares, que emulam o comportamento da tartaruga na tela do computador de forma totalmente virtual, foram produzidos. Tais ferramentas de software não se utilizam de um dispositivo robótico para executar as instruções de desenho em uma superfície física, porém, permitem, de forma prática, realizar o mesmo tipo de pensamento computacional e criativo. A grande vantagem é a possibilidade de se utilizar o *Turtle Graphics* sem a necessidade de qualquer dispositivo adicional, bastando ter à disposição um computador com acesso à Internet ou com o software específico instalado. Exemplos notáveis de tais ferramentas são: o KTurtle, lançado sob a Licença Pública Geral GNU e parte da KDE Software Compilation 4 (um conjunto de aplicações para o sistema operacional GNU/Linux); o módulo *turtle* da linguagem de programação Python; o Blockly Turtle, uma linguagem de blocos específica para *Turtle Graphics*; e o Scratch, também uma linguagem de blocos, que, embora não seja específica para atividades baseadas em *Turtle Graphics*, é fortemente inspirada nele e permite a realização de desenhos usando esse método.

[2] PAPERT, Seymour. Mindstorms: Children, Computers, and Powerful Ideas (2nd ed.). New York: Basic Books. ISBN 0-465-04674-6. OCLC 794964988. 1993

Qual a relação entre *Turtle Graphics*, Computação e Matemática?

O *Turtle Graphics*, além de estimular o pensamento computacional e criativo, pode ser utilizado, seguindo uma sequência didática adequada, para o ensino de conceitos de Matemática, especialmente aqueles relacionados à geometria e álgebra.

A **geometria** acaba sendo utilizada naturalmente durante o desenvolvimento de algoritmos com o *Turtle Graphics* pela própria natureza deste, que consiste na realização de desenhos vetoriais. No entanto, é possível introduzir conceitos como polígonos, ângulos e plano cartesiano de forma dirigida e fácil de ser aplicada em sala de aula. A primeira parte do conteúdo deste livro é baseada principalmente nessa premissa.

O ensino de **álgebra** provém de outros recursos que estão mais relacionados a linguagens de programação em geral, mas que acabam combinando perfeitamente com o uso da geometria permitido pelo *Turtle Graphics*. A ideia é principalmente explorar a transferência de aprendizagem sobre **funções** em linguagens de programação para **funções matemáticas**. Para exemplificar brevemente como isso funciona: o(a) estudante aprende primeiro como construir uma função/procedimento que desenha um triângulo equilátero na tela baseado no tamanho do lado do triângulo desejado; e então, mostramos que a mesma lógica é utilizada na definição de funções matemáticas cujo domínio e imagem se restringem somente a números — como, por exemplo, $f(x) = 2x$. Essa abordagem já foi estudada por diversos pesquisadores com excelentes resultados, como é o caso da equipe do *Bootstrap*[3], um currículo de ensino de matemática e computação desenvolvido na Universidade de Brown, nos Estados Unidos.

Além disso, o ensino de álgebra é também aumentado por meio da explicação sobre o conceito de **variáveis**, assim como a necessidade de se construir **fórmulas**, como é o caso do cálculo da circunferência de um círculo tendo como parâmetro o raio do círculo que se deseja desenhar. Pode-se explorar também a plotagem de **gráficos** de funções, o que permite um aprofundamento ainda maior no conceito de funções matemáticas e **plano cartesiano**.

Portanto, o principal objetivo deste livro é apresentar um currículo, ou sequência didática, que permita ao menos dar um norte ao educador que pretenda utilizar *Turtle Graphics* para ensinar Computação ao mesmo tempo em que ensina Matemática (ou vice-versa), contribuindo assim para o desenvolvimento do pensamento computacional e um maior engajamento, interesse e compreensão do conteúdo matemático tangido pela abordagem.

[3] SCHANZER, Emmanuel et al. Transferring skills at solving word problems from computing to algebra through Bootstrap. In: Proceedings of the 46th ACM Technical symposium on computer science education. 2015. p. 616-621.

Por que utilizar o Blockly Turtle?

A ferramenta utilizada neste livro é o Blockly Turtle, um ambiente de desenvolvimento de Turtle Graphics baseado no Blockly[4], uma biblioteca do Google para criar linguagens de programação baseadas em blocos, ideais para iniciantes. O Blockly (mais especificamente, uma variante chamada Scratch Blocks) é também a base do desenvolvimento da mais recente versão da famosa linguagem de blocos Scratch[5]. O Blockly Turtle permite realizar os comandos e construções de código disponíveis em outras ferramentas de Turtle Graphics, porém de forma mais lúdica por ser realizada por meio da programação em blocos. A ferramenta em questão, por ser originalmente[6] de licença de código aberto (*open source*), foi adaptada e melhorada no decorrer da elaboração deste material, incluindo-se as seguintes funcionalidades:

1. Conversão do código para as linguagens Javascript (ECMAScript) e Python. A conversão para esta última é especialmente interessante, pois utiliza o módulo *turtle* do Python, bastando copiar o código para um ambiente de desenvolvimento na linguagem Python e executá-lo para obter os mesmos resultados nessa linguagem.

2. Possibilidade de salvar e carregar projetos feitos na ferramenta. Isso permite continuar projetos iniciados sem a necessidade de fazer qualquer tipo de conta ou cadastro, bastando que se salve o arquivo em algum meio de armazenamento, como um pen drive, e-mail ou serviços de armazenamento em nuvem, para que possa ser carregado na ferramenta mais tarde.

3. Inclusão de novos blocos, tais como: blocos de manipulação de textos (*strings*), necessários à atividade envolvendo a geração de fractais; o bloco "vai para" (*"go to"*), que permite fazer a tartaruga se mover diretamente a um ponto específico da tela; o bloco "aponte para", que permite definir exatamente o ângulo de direção da tartaruga; e os blocos responsáveis por pegar os valores atuais das coordenadas x e y e do ângulo de direção da tartaruga.

Pelos motivos acima, sugere-se a utilização da ferramenta disponível por meio da URL https://helioh2.github.io/turtle, na qual está disponível essa versão adaptada.

Algumas das vantagens de se utilizar o Blockly Turtle são as seguintes:

1. A programação em blocos não envolve a escrita de textos com comandos precisos, como ocorre nas linguagens de programação tradicionais. Isso pode ser interessante principalmente ao se trabalhar com crianças, que possuem maior dificuldade de digitação. Evita-se também a ocorrência constante de erros de digitação que geram erros

[4] https://developers.google.com/blockly?hl=pt-br

[5] https://scratch.mit.edu/

[6] A ferramenta original, de autoria principal do engenheiro de software Neil Fraser, do Google, pode ser acessada por meio da seguinte URL: https://blockly.games/turtle

de execução do código, o que é bem comum em linguagens de programação textuais. Com isso, busca-se gerar menos frustração no estudante e melhorar o fluxo e andamento da aula.

2. Embora a programação em blocos não seja textual, é possível a conversão do código em blocos para um código textual em uma linguagem de programação tradicional. Em nosso Blockly Turtle há dois botões que, ao pressioná-los, permitem converter o código em blocos para a linguagem Javascript e para a linguagem Python, respectivamente. Com isso, o aluno mais avançado pode visualizar como seu código em blocos seria escrito em uma dessas linguagens de programação. O código em Python gerado possui ainda a vantagem de poder ser copiado e colado em um ambiente de desenvolvimento em Python e executado, pois é gerado de modo a funcionar com o módulo *turtle* que já vem embutido nessa linguagem.

3. O Blockly Turtle é uma aplicação web, o que significa que basta ter um computador conectado à Internet para sua utilização, não necessitando da instalação de nenhum programa adicional. No entanto, é importante ressaltar que ele pode não funcionar em navegadores (browsers) mais antigos, como o Internet Explorer, que acompanhava o sistema operacional Windows em versões anteriores. No entanto, acreditamos que a ferramenta poderá ser usada sem problemas na maioria dos computadores mais recentes, dos últimos 10 a 15 anos, que já possuem instalados navegadores mais modernos, como o Microsoft Edge, o Google Chrome e o Mozilla Firefox.

Quando não utilizar o Blockly Turtle:

1. Em máquinas muito antigas que não possuem um navegador compatível ou que possam apresentar lentidão devido ao processamento gráfico necessário à execução da ferramenta. Portanto, sugere-se testar as máquinas que serão utilizadas de antemão a fim de verificar se não haverá nenhum problema.

2. Em turmas de alunos com mais idade (adolescentes ou jovens adultos) para os quais pretende-se ensinar principalmente programação, e para os quais é, portanto, mais interessante o uso de uma linguagem de programação de propósito geral e que poderia ser futuramente utilizada em projetos práticos e no mercado de trabalho. Nesses casos, sugere-se diretamente o uso da linguagem Python com o módulo *turtle*, apresentada em outro volume deste material, também disponibilizado no Acervo Digital - Biblioteca Temática: REA/PEA UFPR.

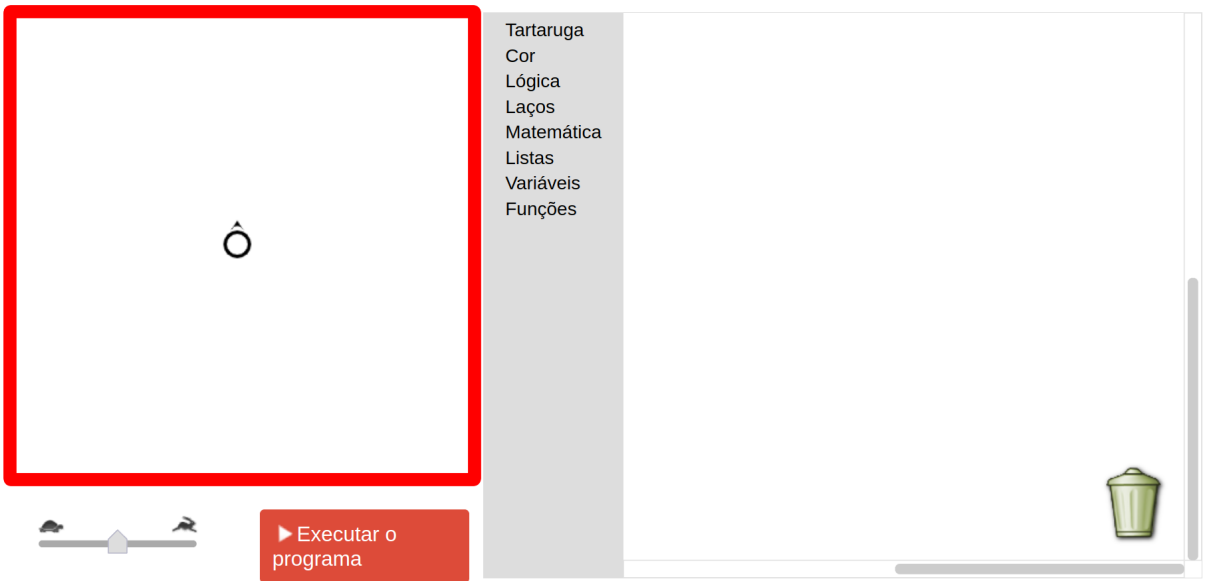
Preparação do ambiente/máquina

1. Acesse um dos seguintes sites (preferencialmente o primeiro):
    a. https://helioh2.github.io/turtle
    b. https://blockly.games/turtle?lang=pt-br&level=10
    c. https://bpp.rs/blockly/apps/turtle/index.html?lang=pt-br
2. Em qualquer dos casos, é possível escolher o idioma "Português Brasileiro" no canto superior direito da tela.

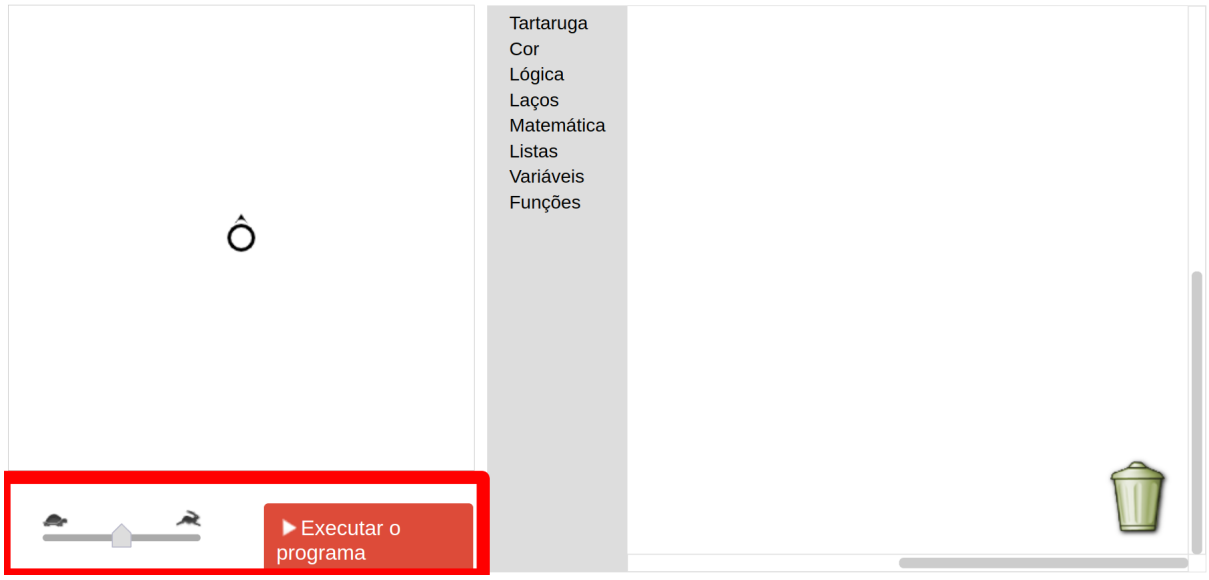
Introdução ao ambiente

Recomenda-se deixar o site aberto e traduzido para Português (Brasileiro). Também recomenda-se remover o bloquinho já existente (*"avançar"*) na área em branco à direita. Para isso, basta clicar na parte verde do bloco e arrastar com o mouse para a lixeira, conforme a imagem abaixo.

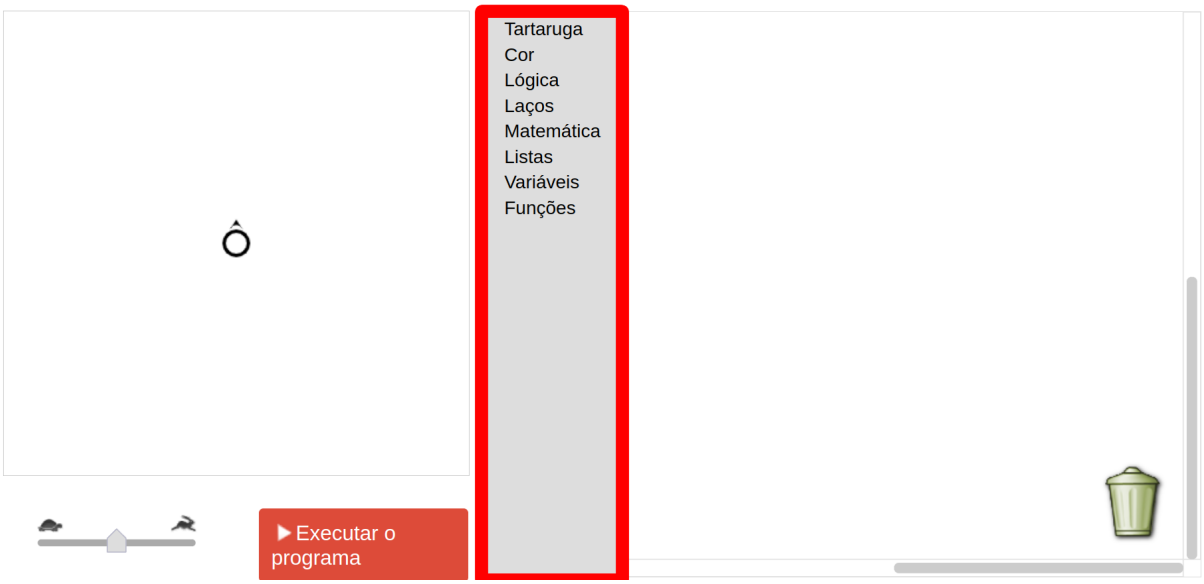
A interface do Blockly Turtle é composta basicamente por quatro áreas. A área em branco do lado esquerdo é a área em que a tartaruga executará seus movimentos. Note que a tartaruga, representada pelo círculo e uma setinha, está apontando para cima.



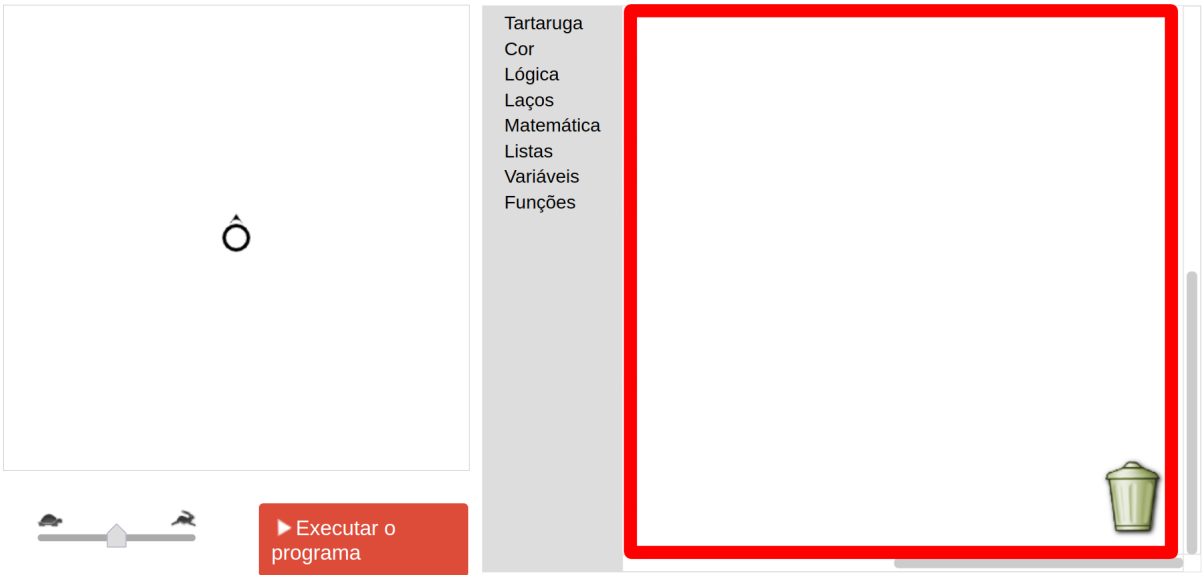
A área no canto inferior esquerdo contém o botão "Executar o programa", e um *slider* para definir a velocidade da tartaruga. Abaixo desse menu, se você estiver usando o site https://helioh2.github.io/turtle, haverá dois botões de conversão para código textual, que veremos mais adiante.



A área no centro consiste no menu de blocos. Em breve iremos interagir com esse menu.



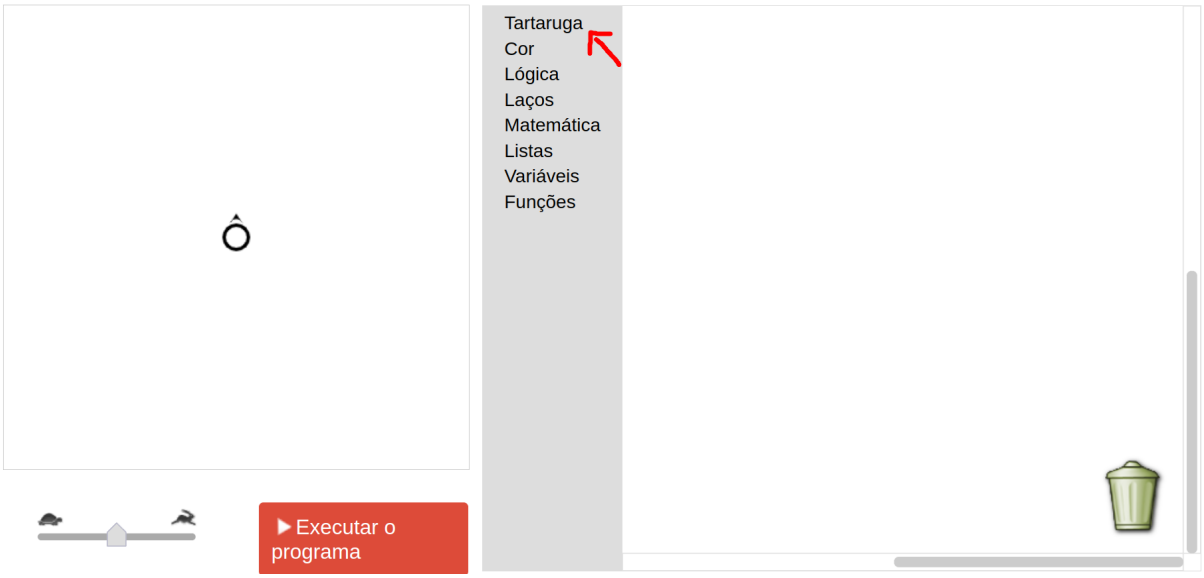
E, por fim, a área em branco do lado direito é a área de programação, onde inserimos os blocos de comandos para construir o nosso programa. Nessa área também há uma lixeira para onde podemos arrastar os blocos de comando que não usaremos mais.

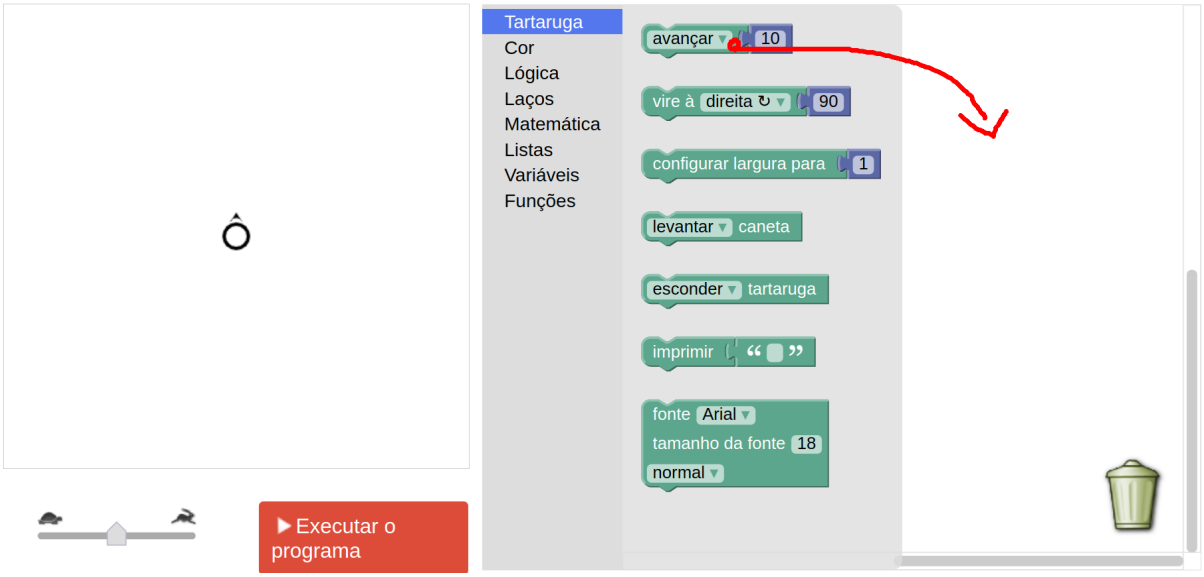


Além dessas áreas, há também uma área superior que contém algumas opções: uma caixa de texto para dar um nome ao projeto, um botão que permite salvar/baixar o projeto, um botão que permite carregar um projeto salvo, e a caixa de seleção de idioma. Em breve veremos a utilidade das opções relacionadas a salvar e carregar projetos.

Inserindo os primeiros blocos

Peça que os alunos acessem o menu **Tartaruga** e, em seguida, que arrastem o bloco *"avançar"* para a área de programação, à direita.





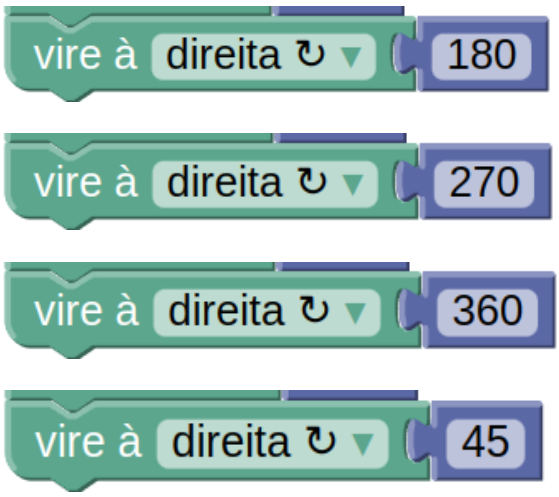
Explique que esse comando ordena que a tartaruga se mova para frente. Em seguida, clique no botão “Executar o programa”. Veja a tartaruga em ação.

Peça para os alunos alterarem o número na frente do bloco avançar. Por exemplo, troque para 100 e peça que executem novamente o programa. Para executar novamente, basta clicar no botão vermelho “Reiniciar”, e ele então mudará novamente para “Executar o programa”. Clique então novamente nele.

Mostre também que é possível fazer a tartaruga girar. Peça para os alunos encaixarem, logo abaixo do comando de avançar, o comando vire à direita, que também se encontra no menu Tartaruga. Mostre que é possível encaixar o novo comando simplesmente arrastando o bloco logo abaixo do comando já existente:

Peça para os alunos clicarem em “Reiniciar”, e em seguida “Executar o Programa”.

Pergunte a eles: *"O que aconteceu?"*. Peça para eles alterarem o ângulo na frente do bloco vire à direita e continue perguntando o que acontece:

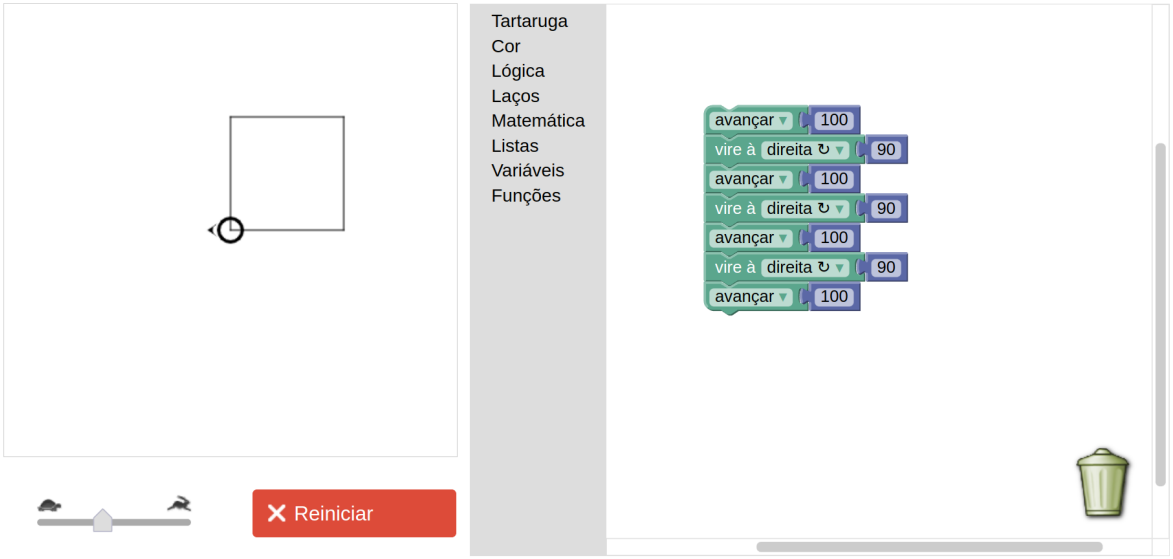


Agora pergunte a eles: *"E se eu quisesse que a tartaruga girasse para a esquerda?"*. Mostre que é possível alterar o sentido da rotação da tartaruga clicando-se em direita dentro do bloco e alterando-o para esquerda.

Desenhando um quadrado

Agora proponha um desafio aos alunos. Peça-lhes que tentem criar um quadrado usando os comandos aprendidos. Mostre na lousa como se faz um quadrado, fazendo uma analogia entre seu giz/caneta e a tartaruga. Dê alguns minutos para eles tentarem desenhar um quadrado.

Elogie e, talvez, recompense os alunos que conseguiram desenhar o quadrado. Parabenize também os que tentaram mas não conseguiram, tentando estimulá-los a não desanimar e continuar se esforçando. Mostre uma possível solução/algoritmo (sem utilizar *loop*/laço de repetição):

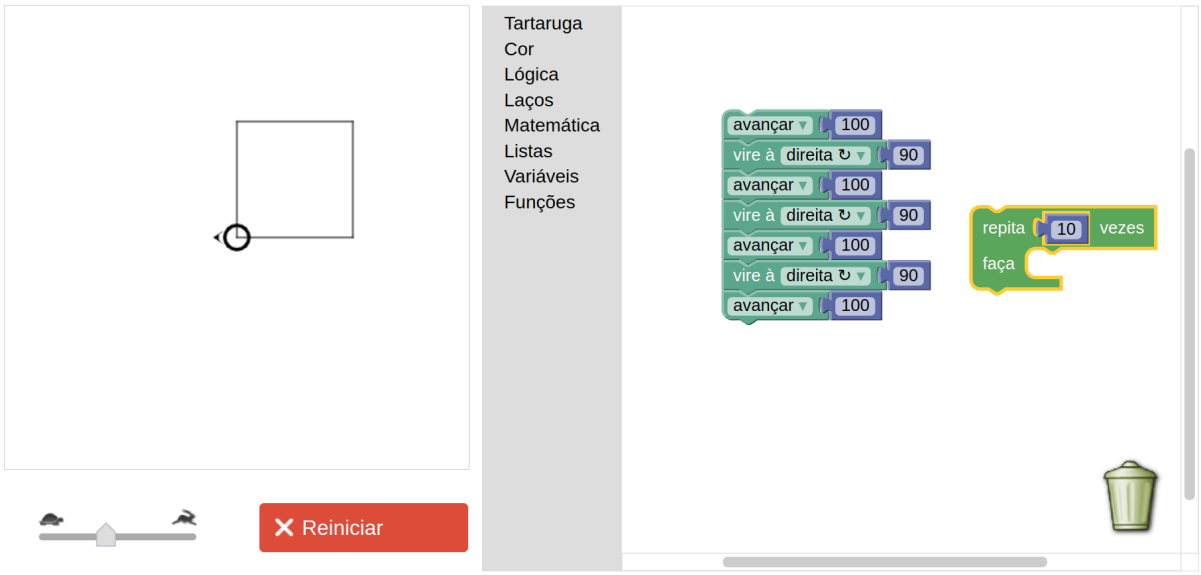


Se achar conveniente, ao mostrar o exemplo funcionando, diminua a velocidade da tartaruga por meio do *slider* no canto inferior esquerdo, ao lado do botão de execução.

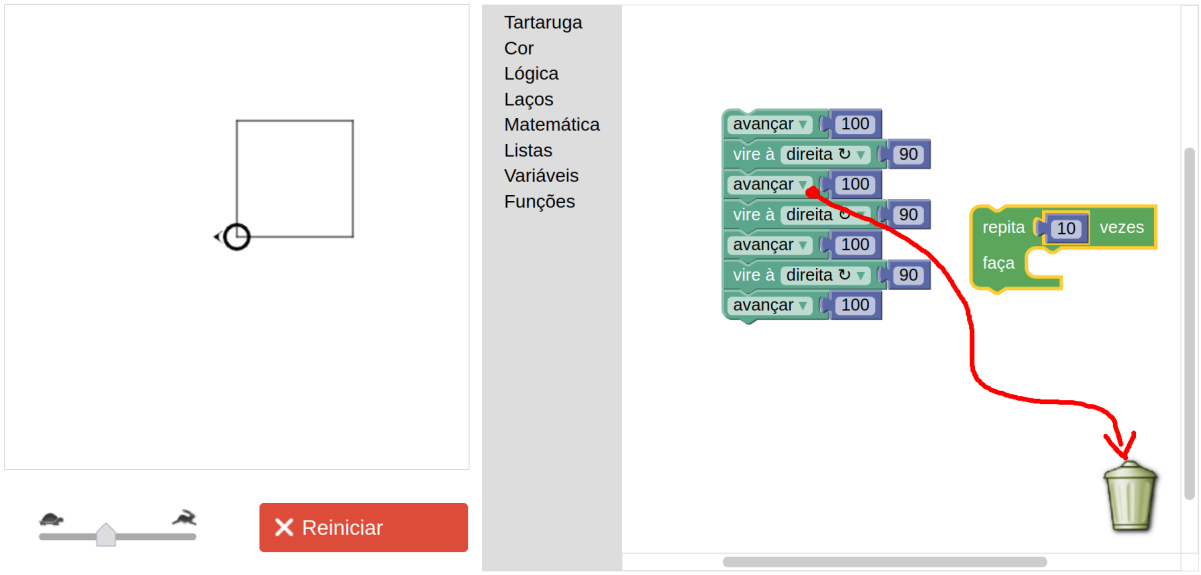
Agora, instigue os alunos para que analisem a solução quanto ao fato de ter que ficar repetindo os mesmos comandos várias vezes para desenhar o quadrado. *"Vocês perceberam que tivemos que repetir várias vezes o* avançar 100 *e o* vire à direita 90*? Quantas vezes repetimos esses comandos?"*. Junto com os alunos, conte quantas vezes foram repetidos os comandos avançar 100 e vire à direita 90. Você deverá chegar, junto a eles, à seguinte conclusão: o comando avançar 100 é repetido 4 vezes, e o comando vire à direita 90 é repetido 3 vezes.

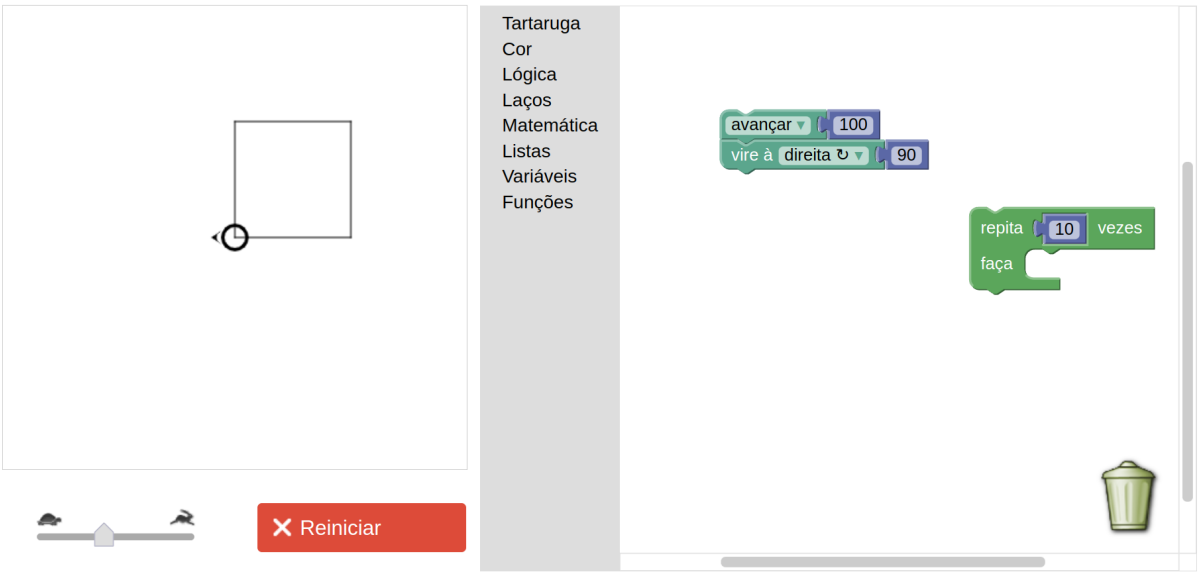
Em seguida, pergunte-lhes: *"Mas o que acontece se adicionarmos mais um* vire à direita 90 *no final. Muda alguma coisa?"*. Peça aos alunos para repetirem a sequência de comandos realizadas anteriormente, adicionando mais este comando (vire à direita 90), com a finalidade de demonstrar que não mudou nada além do fato de a tartaruga ter girado mais uma vez por 90 graus no final. O quadrado continua sendo desenhado. Agora discuta com eles visando chegar à seguinte conclusão: a sequência avançar 100 e vire à direita 90 é repetida exatamente 4 vezes. Hora de introduzir o comando repita.

Para isso, acesse o menu **Laços** e arraste o comando repita 10 vezes faça, conforme ilustrado abaixo.

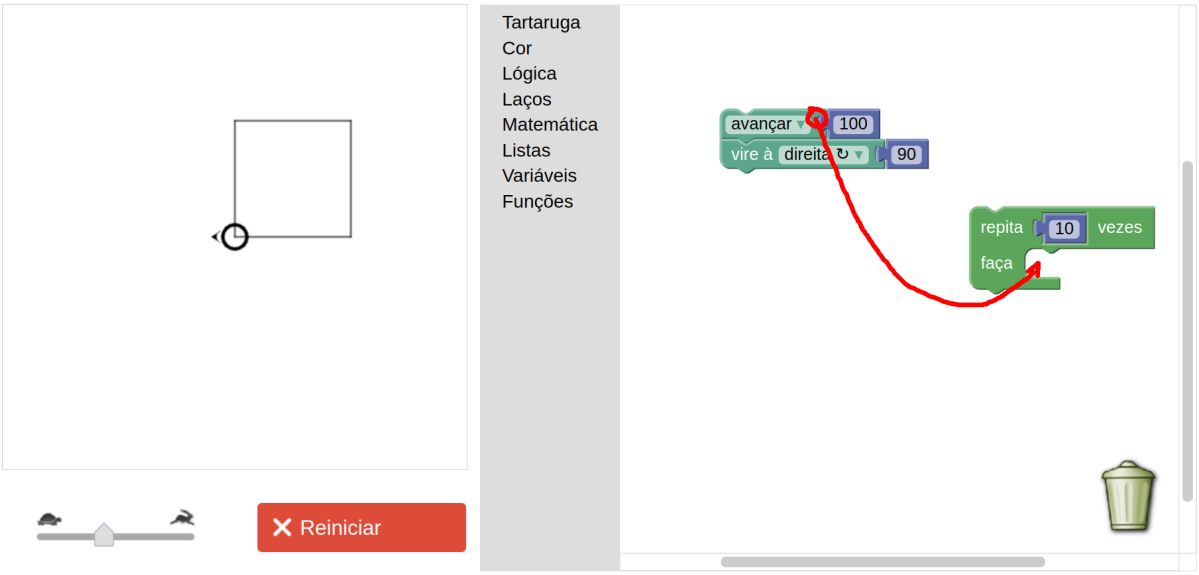


Perceba que, ao arrastar o comando para uma área em branco, ele fica separado do resto do código (programa). Perceba também que o bloco do repita possui um encaixe diferente, de modo que é possível encaixar outros comandos "dentro" dele. Faça o seguinte: desencaixe os últimos 5 (ou 6, caso você tenha colocado um último vire à direita, conforme discutido) blocos da sequência de comandos que desenhava o quadrado, e arraste esses blocos para a lixeira. Fazendo isso, restará apenas a sequência de comandos contendo um único avançar e vire à direita. Veja ilustração abaixo:





Agora, arraste os dois blocos que sobraram para dentro do repita (assegure-se de pegar com o mouse no primeiro bloco, para não desencaixá-los um do outro. Se acabar desencaixado os dois sem querer, basta encaixá-los novamente). Veja a ilustração abaixo:



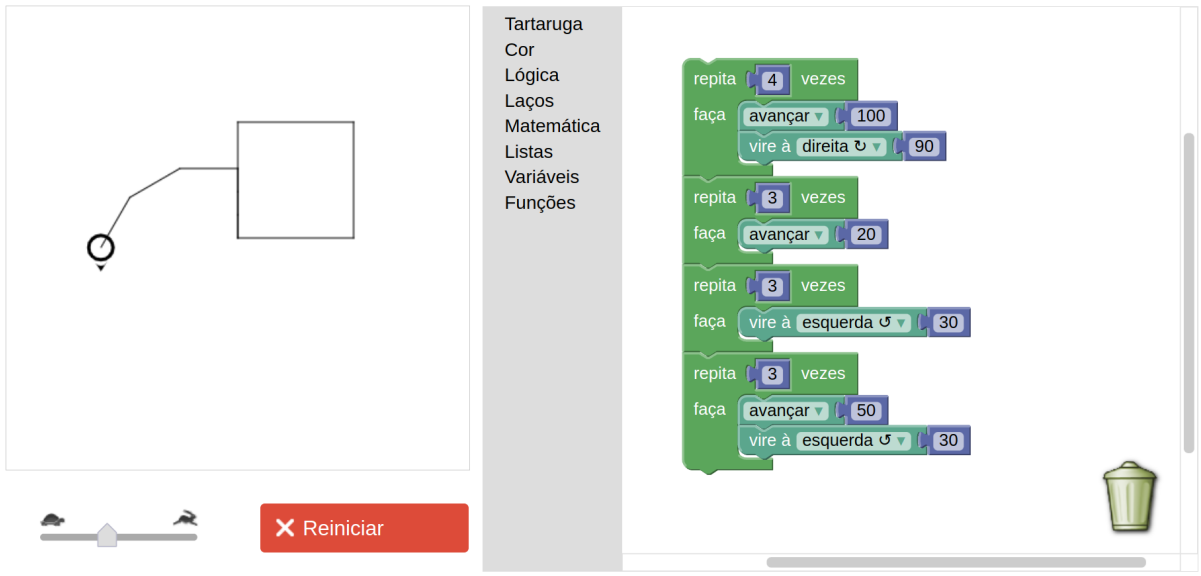
Arraste o bloco repita mais para o centro da tela.

Note que o comando repita diz que repetirá 10 vezes. Mas, como discutido anteriormente, precisamos repetir os dois comandos de avançar e vire apenas quatro vezes. Portanto, altere o valor de 10 para 4 no comando repita.

Enfim, execute o programa clicando em "Reiniciar" e "Executar o programa", e observe junto com os alunos a tartaruga executando as instruções.

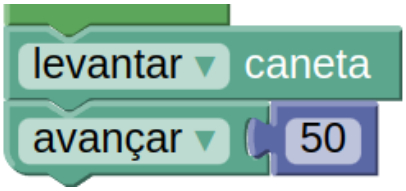
O comando é auto-explicativo, mas explique brevemente como funciona o repita.

No entanto, é sempre mais fácil fazer mais exemplos para eles entenderem o raciocínio. Peça para eles encaixarem outros comandos logo abaixo (**cuidado: não dentro!** Veja as imagens abaixo) do comando repita:

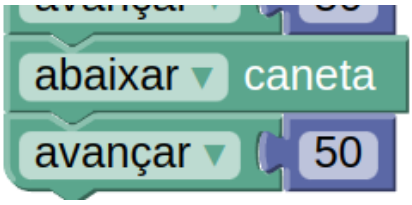


Aqui é uma boa hora para introduzir também os comandos levantar/abaixar caneta. Explique que a tartaruga tem uma caneta na mão, e por isso ela desenha quando anda. Normalmente, ela

anda com a caneta encostada no “chão” (aqui simulado pela tela), o que faz ela desenhar. No entanto, ela também pode levantar a caneta para que possa andar sem desenhar nada no “chão”. Peça para eles testarem os seguintes blocos:



Pergunte o que aconteceu. Agora peça para eles colocarem, logo em seguida:



> **Obs:** o abaixar fica no mesmo bloco de comando do levantar, basta clicar na parte verde-clara do bloco e aparecerá a opção de alterá-lo para abaixar.

Pergunte o que aconteceu. Deverá ficar evidente para eles que levantar caneta faz a tartaruga levantar a caneta (deixando de desenhar), e abaixar caneta faz ela abaixar a caneta (voltando a desenhar).

Peça para os alunos salvarem seus programas, pois em seguida iremos fazer um novo programa do zero.

Para isso, se estiver usando o site https://helioh2.github.io/turtle, escreva um nome para o projeto na caixa de texto na parte superior da tela (por exemplo, “quadrado”) e use o botão  para fazer o download do arquivo. Esse arquivo pode ser usado mais tarde para restaurar o projeto por meio do botão  . Recomenda-se também fazer upload do arquivo por e-mail (envio de anexo) ou por meio de algum serviço de nuvem, para não perder o projeto salvo.

Se estiver usando o site https://blockly.games/turtle, é possível salvar o código feito clicando-se no botão . Uma URL será gerada. Copie-a em algum lugar para que possa ser recuperada mais tarde. Para restaurar o projeto mais tarde, basta colar a URL na barra de endereço do navegador.

Opcional: Convertendo para código textual

Se você estiver usando a ferramenta por meio do site https://helioh2.github.io/turtle, é possível converter o código em blocos para código em linguagem de programação textual.

Esta é uma boa oportunidade para explicar aos alunos que a programação geralmente é feita de modo textual, e que os bloquinhos que estamos utilizando é só uma forma de termos um primeiro contato mais divertido com a programação. Explique também que todos os aplicativos de celular, programas de computador e jogos são programas, e foram escritos utilizando um código de programação textual. Uma ideia para exemplificar isso é mostrar o código-fonte de uma página qualquer da web. Abra, por exemplo, a página inicial do site da sua instituição de ensino em uma nova aba do navegador, clique com o botão direito do mouse em qualquer parte em branco da página, e procure a opção "Mostrar código-fonte" (a forma como isso está escrito depende de navegador para navegador). Será aberta uma janela/aba contendo o código em HTML[7] da página atual. Se quiser mostrar um pouquinho mais, feche a janela/aba do código-fonte, volte à página, clique com o botão direito e escolha a opção "Inspecionar". Uma parte diferente será aberta dentro do navegador. Dentro dessa parte, provavelmente haverá algumas abas (*"Console"*, *"Elements"*, *"Source"*, etc.). Clique na aba *"Source"* (ou "Fonte"). Ali você verá uma estrutura de diretórios (pastas) contendo os diversos arquivos de código-fonte que estão sendo executados para a página funcionar como ela funciona. Mostrando isso, você explica que cada página da Internet, assim como cada aplicativo de celular e os jogos que os alunos jogam no computador ou em *consoles* de videogame, são todos feitos de códigos de programação que outras pessoas escreveram.

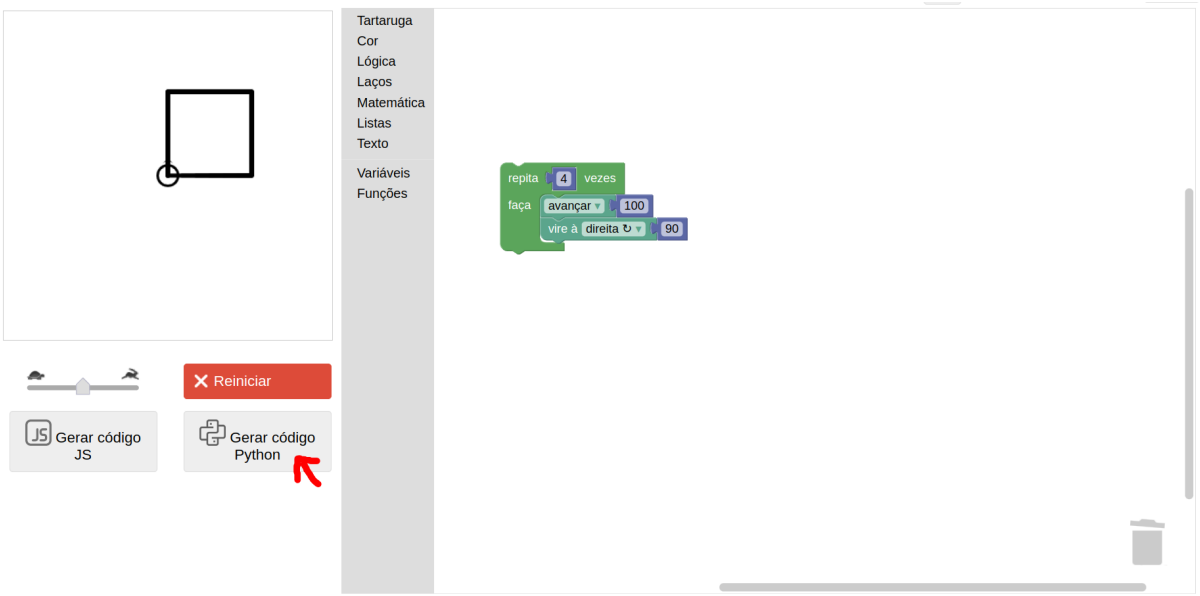
Voltando ao Blockly Turtle, peça para os alunos clicarem no botão "Gerar Código JS", logo abaixo da área em que se encontra o botão "Executar o programa". Uma janelinha *pop-up* abrirá dentro da página do Blockly Turtle, contendo o código em Javascript que é executado toda vez que clicamos no botão "Executar o programa". Explique que o Javascript, também chamado de ECMAScript, é a linguagem de programação que é executada dentro do navegador, e que todas as páginas da Internet provavelmente executam algum tipo de código nessa linguagem. Explique, portanto, que o código em blocos que criamos é sempre convertido para esse código em Javascript para que possa fazer a tartaruga executar os comandos que pedimos. Se você tinha mostrado os códigos após clicar em "Inspecionar" no navegador, você deve ter visto alguns códigos em Javascript ali, que são os arquivos terminados com a extensão ".js". Se não, volte para a aba da página de sua instituição e procure por esses arquivos para mostrá-los para os alunos. Explique que, por mais que o código Javascript pareça "assustador", um programa complexo é geralmente desenvolvido por uma equipe de várias pessoas e/ou com um longo tempo de desenvolvimento, que vão desde meses até anos. Exemplifique isso com alguns dados sobre o tempo de desenvolvimento de alguns jogos famosos. Abaixo são apresentados alguns exemplos.

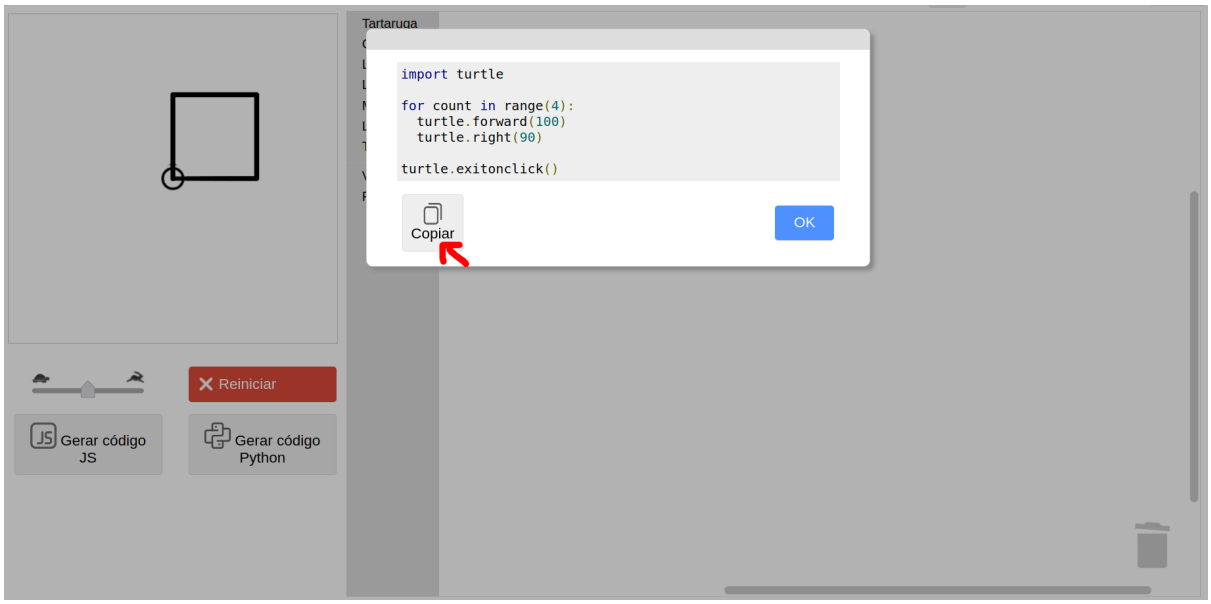
[7] HTML (*Hypertext Markup Language*) é a linguagem de marcação usada para *design* e construção das páginas Web (popularmente chamadas, no Brasil, de "páginas da Internet", ou *"sites"*).

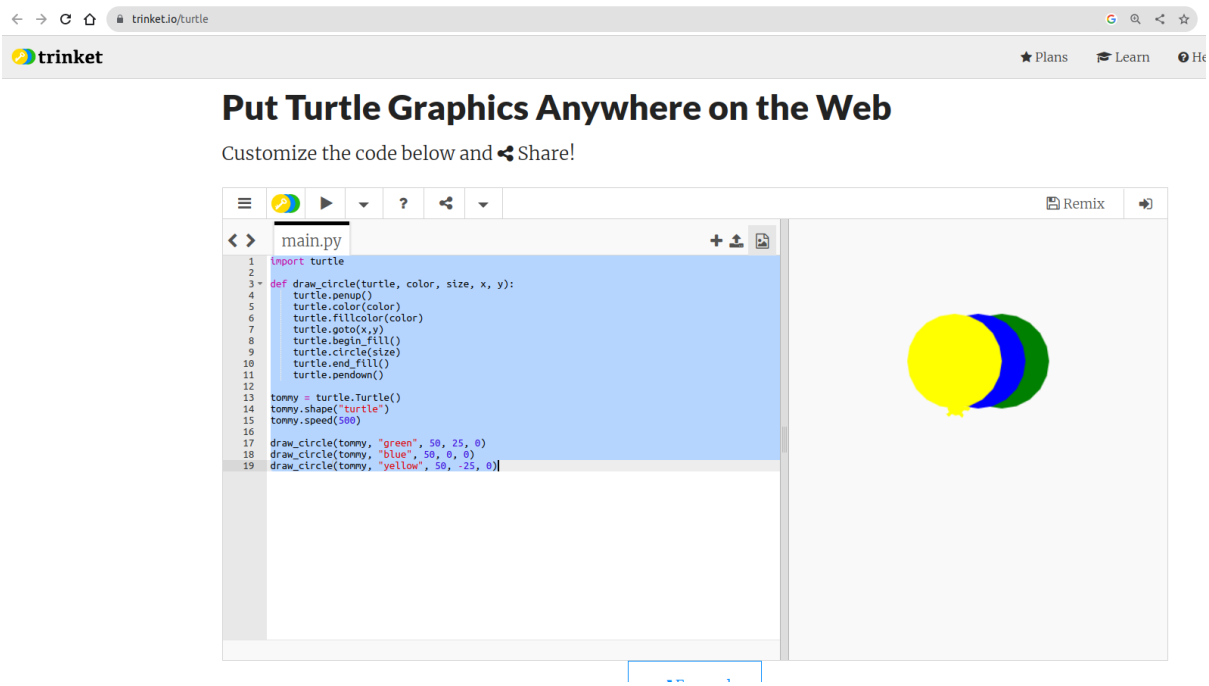
Jogo	**Tempo de desenvolvimento**	**Quantidade de pessoas na equipe**	**Orçamento**
Call of Duty: Modern Warfare 2	2 anos	500+	Mais de 50 milhões de dólares
The Legend of Zelda: Breath of the Wild	5 anos	300	Aprox. 120 milhões de dólares
Grand Theft Auto V	5 anos	1000	Aprox 265 milhões de dólares

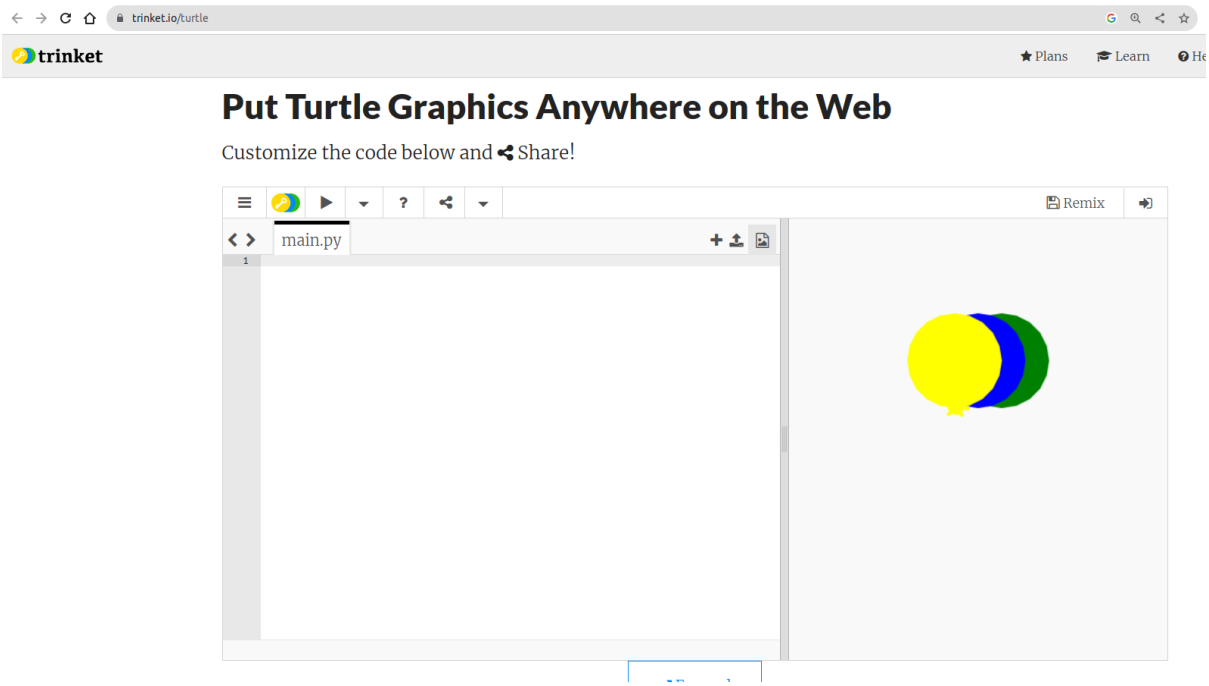
Peça para os alunos fecharem a janelinha do código clicando em "Ok", e peça para clicarem no outro botão "Gerar Código Python". Explique que o Python é outra linguagem de programação muito utilizada nos dias atuais, como em análise de dados e desenvolvimento de sistemas em geral. Uma particularidade interessante do código em Python gerado é que ele é compatível com o módulo *turtle*, que permite programar a tartaruguinha nessa linguagem. Peça para eles clicarem no botão copiar e, em seguida, acessar o site https://trinket.io/turtle ou https://www.pythonsandbox.com/turtle (caso nenhum desses esteja disponível, pesquise na Internet por "python turtle online"). Esses sites abrem um ambiente de desenvolvimento em que é possível escrever código em Python e executar o *turtle*. Peça para os alunos apagarem todo o código que estiver predefinido nesses sites, e colarem o que foi copiado do Blockly Turtle (explique como faz para colar, seja clicando com o botão direito e escolhendo a opção "Colar", seja usando o comando de teclado "Ctrl+V"). O código então será colado no editor, e pode ser executado clicando-se no botão de "play", representado por um triângulo deitado.

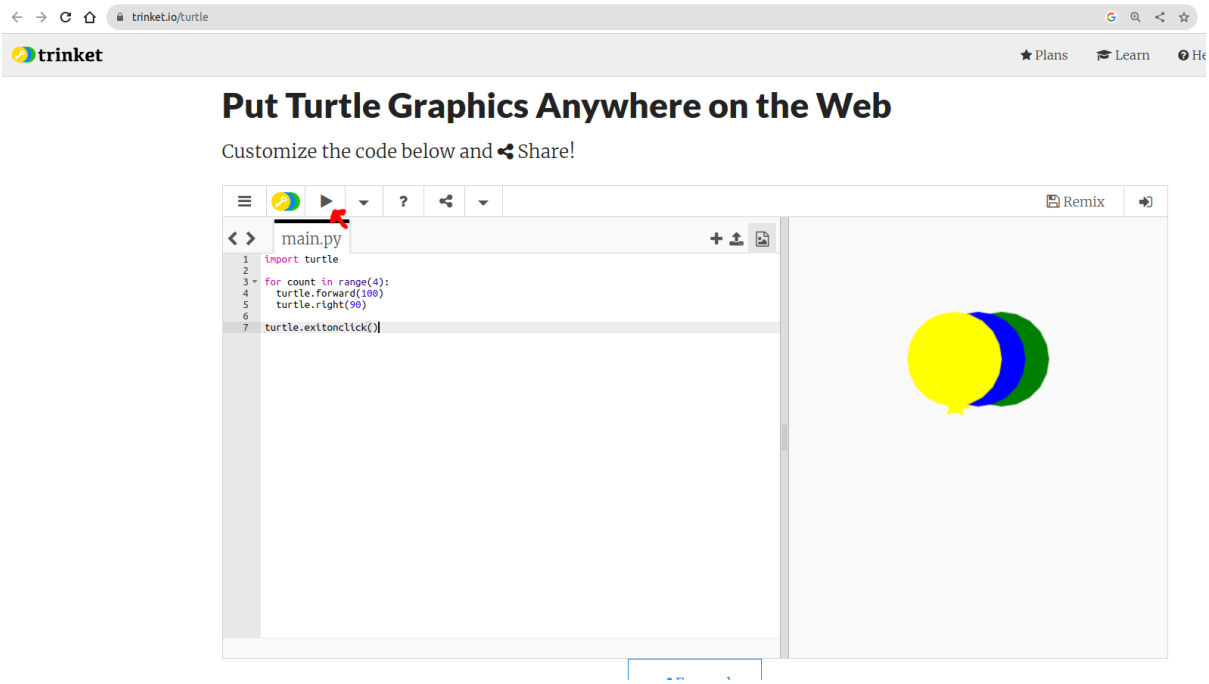
Abaixo é mostrado passo a passo o processo de gerar o código em Python, copiá-lo, abrir o editor de Python, colar o código, e executá-lo.

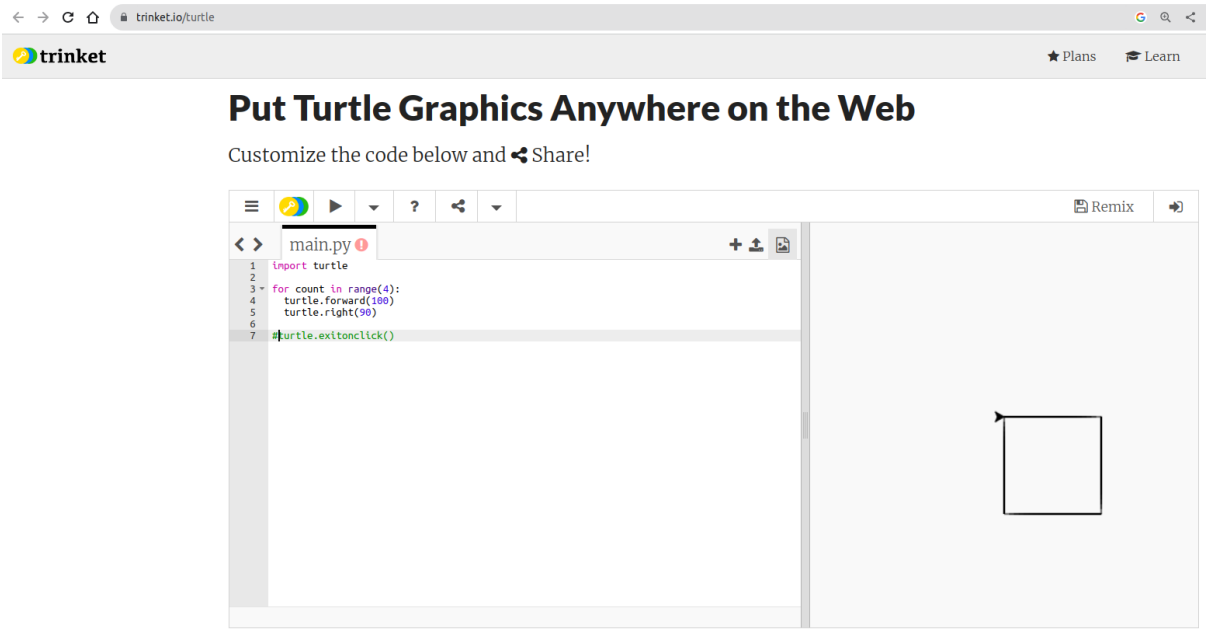

Tartaruga
Cor
Lógica
Laços
Matemática
Listas
Texto
Variáveis
Funções
repita 4 vezes
faça avançar 100
vire à direita 90
Reiniciar
Gerar código JS
Gerar código Python


Tartaruga
import turtle
for count in range(4):
turtle.forward(100)
turtle.right(90)
turtle.exitonclick()
Copiar
OK
Reiniciar
Gerar código JS
Gerar código Python


trinket.io/turtle
trinket
Plans
Learn
Put Turtle Graphics Anywhere on the Web
Customize the code below and Share!
Remix
main.py
import turtle
def draw_circle(turtle, color, size, x, y):
turtle.penup()
turtle.color(color)
turtle.fillcolor(color)
turtle.goto(x,y)
turtle.begin_fill()
turtle.circle(size)
turtle.end_fill()
turtle.pendown()
tommy = turtle.Turtle()
tommy.shape("turtle")
tommy.speed(500)
draw_circle(tommy, "green", 50, 25, 0)
draw_circle(tommy, "blue", 50, 0, 0)
draw_circle(tommy, "yellow", 50, -25, 0)


trinket.io/turtle
trinket
Plans
Learn
Put Turtle Graphics Anywhere on the Web
Customize the code below and Share!
Remix
main.py

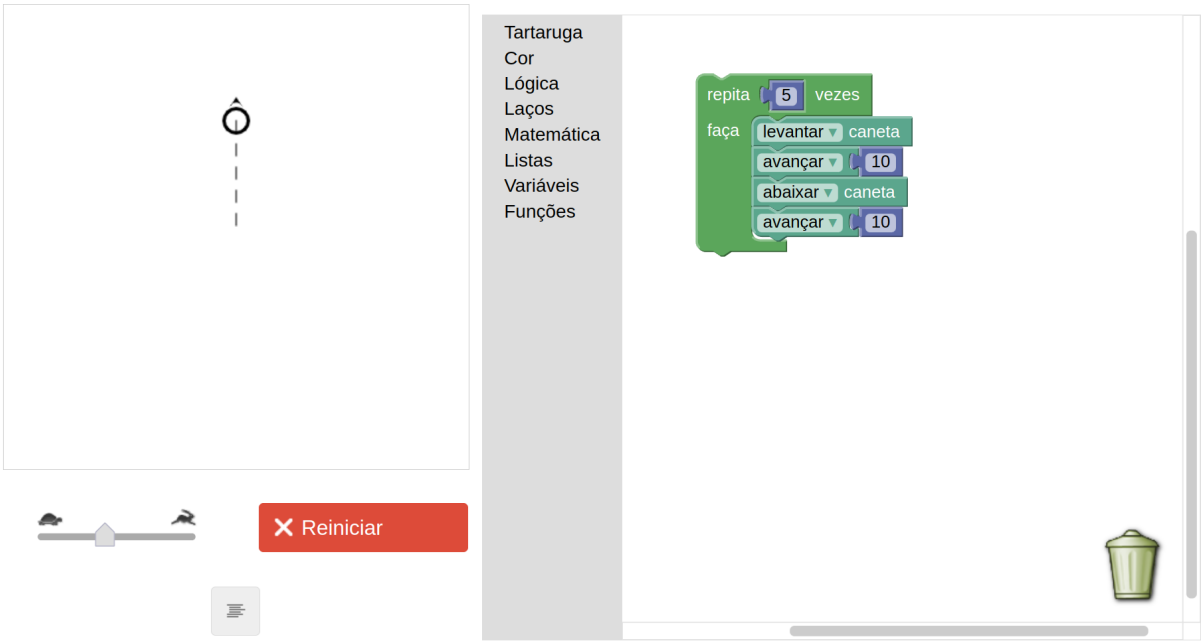




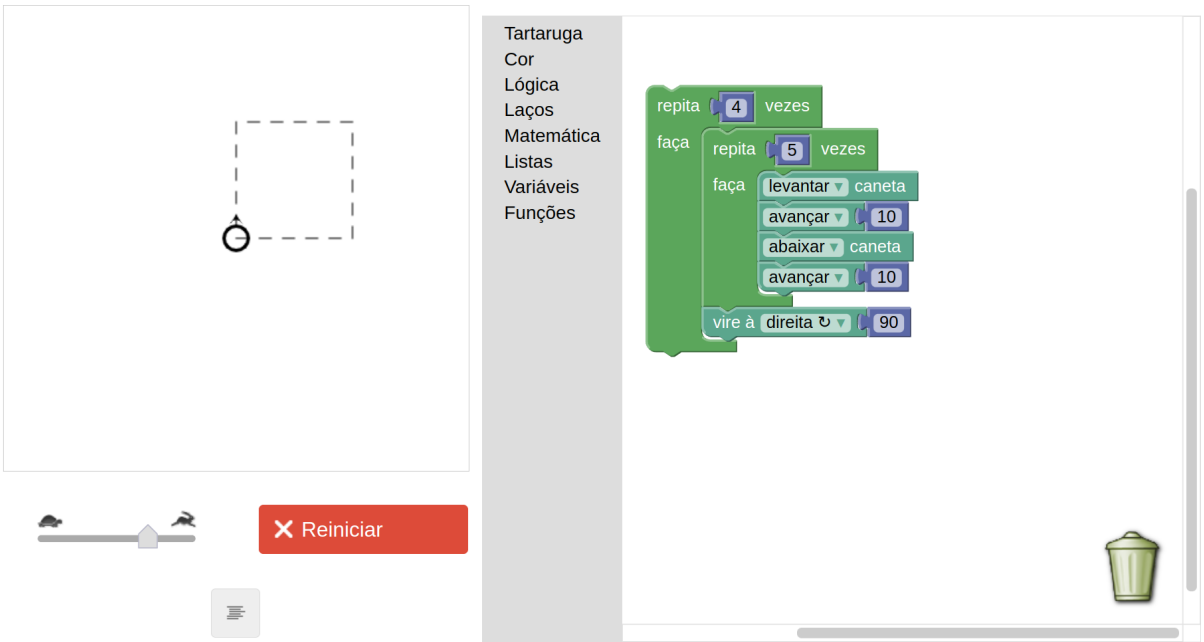
Linhas tracejadas

Agora, proponha um desafio: *"E se eu quiser que a tartaruga faça uma linha tracejada? Quero que ela desenhe uma linha tracejada com cinco traços"*. Mostre na lousa o que você quer fazer. Dê alguns minutos para os alunos tentarem por si próprios. Dê a dica de que eles podem usar o repita para ficar mais fácil.

Elogie aos que conseguiram e aos que se esforçaram, e mostre a solução:



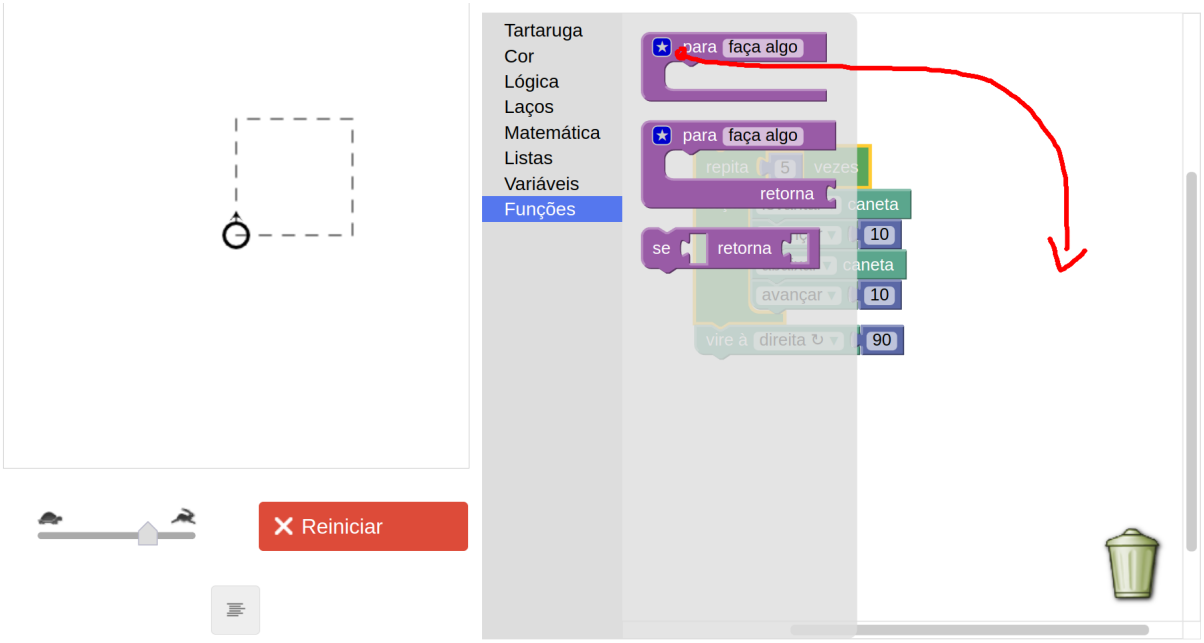
Agora pode ser interessante lançar mais um desafio: peça aos alunos que desenhem um quadrado feito com as linhas tracejadas. Na lousa, mostre o que você quer fazer, deixando clara a ideia de usar os mesmos comandos para linhas tracejadas repetindo-os 4 vezes e girando 90 graus cada vez. Dê alguns minutos para eles tentarem. Se alguém conseguir, elogie e estimule os demais. Dê a solução:

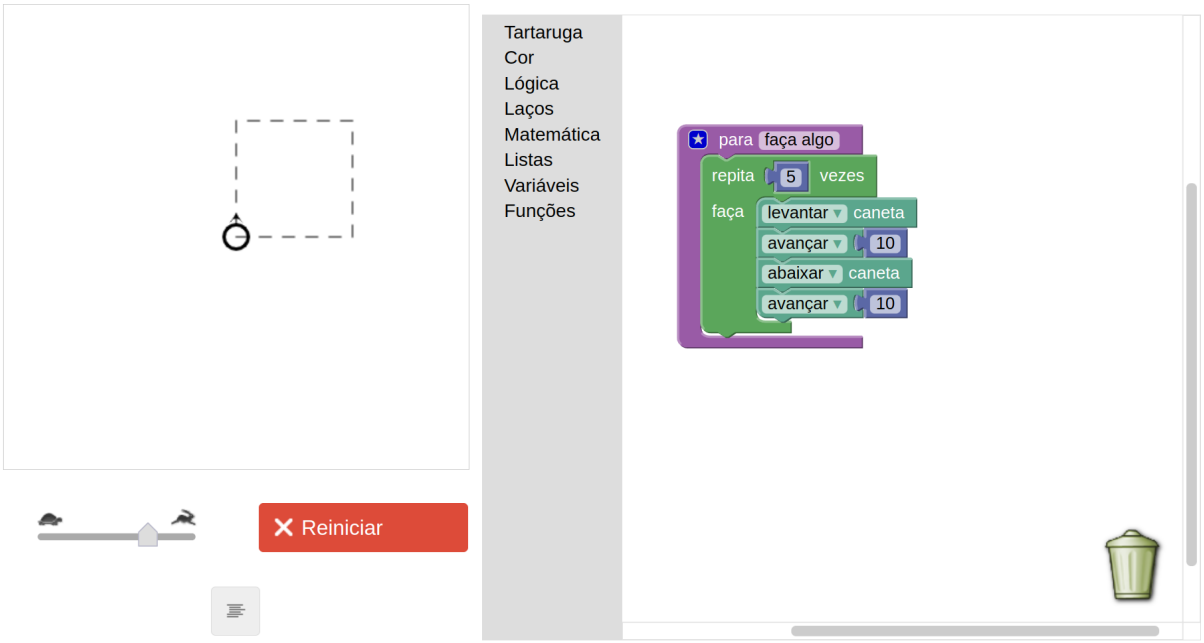


Explique que cada repita tem seu próprio escopo e que, neste caso, temos dois repitas aninhados (um dentro do outro). Indague os alunos de modo similar ao que segue: *"Vocês perceberam que tivemos que repetir de novo todos aqueles comandos para fazer a linha tracejada? E se tivesse uma maneira da tartaruga* ***aprender*** *como fazer a linha tracejada de uma vez por todas, e depois só pedir para ela desenhar de novo sem ter que repetir todos os comandos?"*.

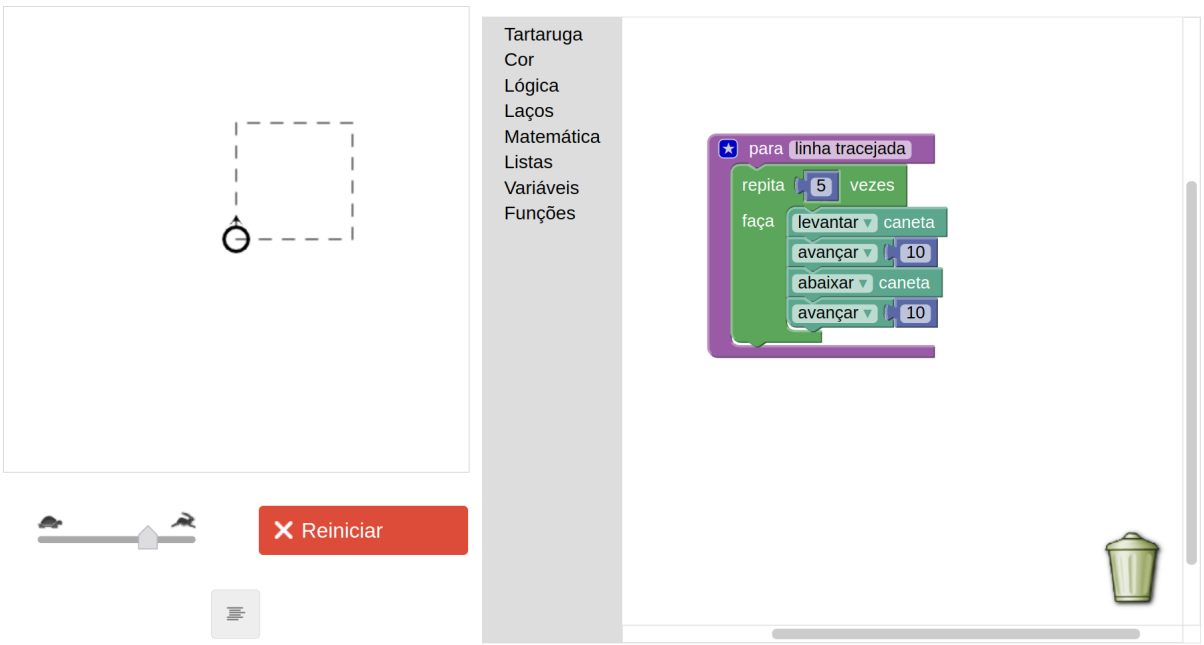
Faça uma dinâmica com os alunos: convide um(a) aluno(a) para ficar de pé, e faça de conta que ele(a) é um robô, e que você dará os comandos a ele(a). Peça para ele(a) andar alguns passos à frente, depois girar o corpo, e depois dar mais alguns passos até ficar de frente para a porta da sala. Depois, peça para voltar de onde saiu, e diga algo assim: *"Agora eu quero que você faça de novo os mesmos comandos, mas quero também que você aprenda/memorize esses comandos."*. Repita os "comandos" para que ele(a) chegue até a porta e diga: *"Muito bem, agora você aprendeu o comando 'vá até a porta'. Sempre que eu te pedir 'vá até a porta', você deve repetir os comandos exatamente como fez. Vamos testar?"*. E então, peça para ele(a) voltar ao ponto inicial e diga: *"Fulano(a), 'vá até a porta'"*. O(a) aluno(a) então deve "executar" novamente os comandos, sem você explicar os passos que deve seguir.

Esta dinâmica tem a intenção de facilitar a compreensão do conceito de **criar uma função** na linguagem, isto é, **fazer a tartaruga aprender um novo comando**. Explique que, da mesma forma que o(a) colega "aprendeu" como "ir até a porta", também podemos fazer a tartaruga aprender sequências de comandos. Mostre que vamos fazer a tartaruga aprender a desenhar uma linha tracejada, como feito anteriormente. Para isso, arrastamos o bloco de definição de função (para 'faça algo'), do menu **Funções**, conforme mostrado abaixo:

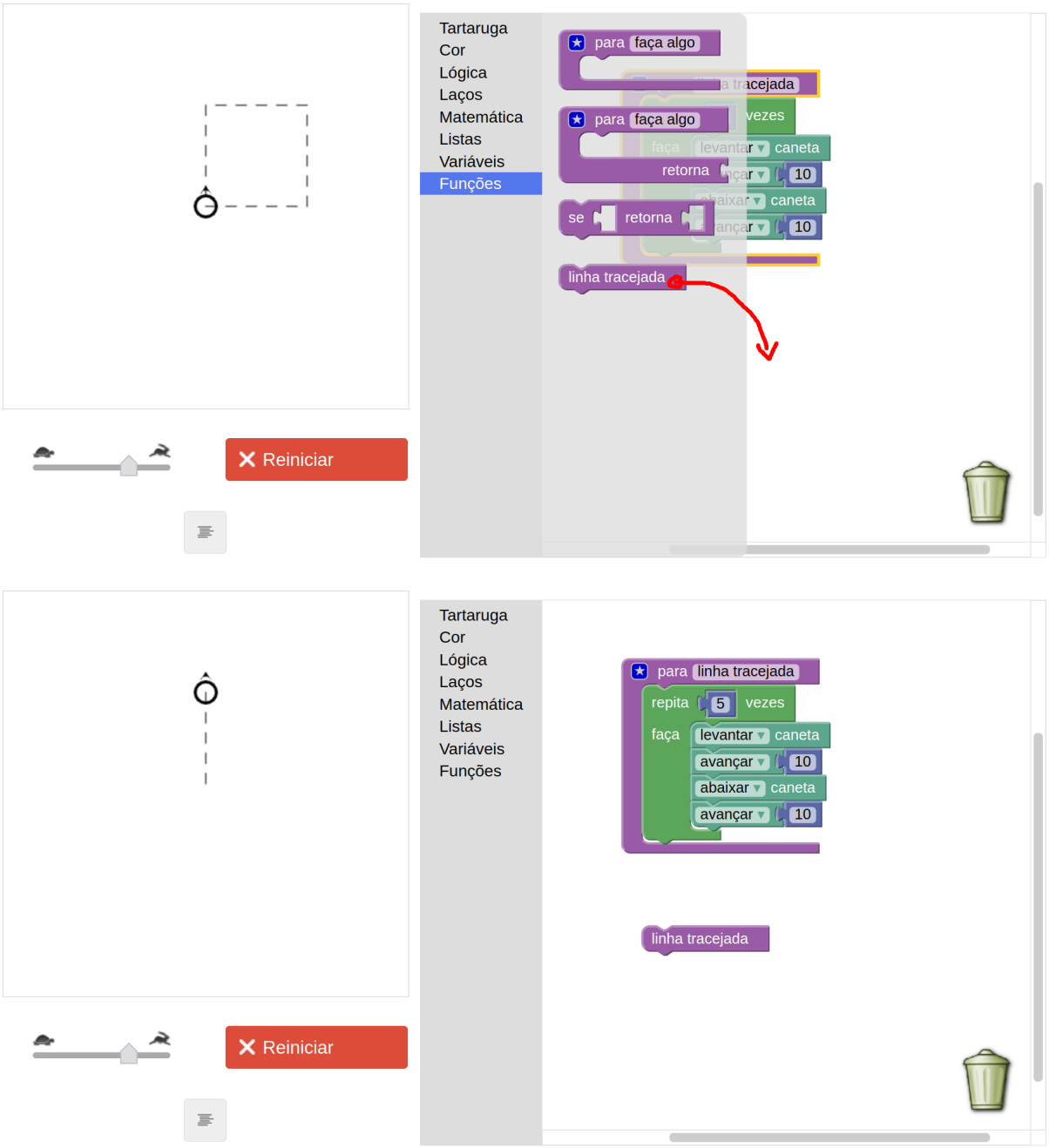




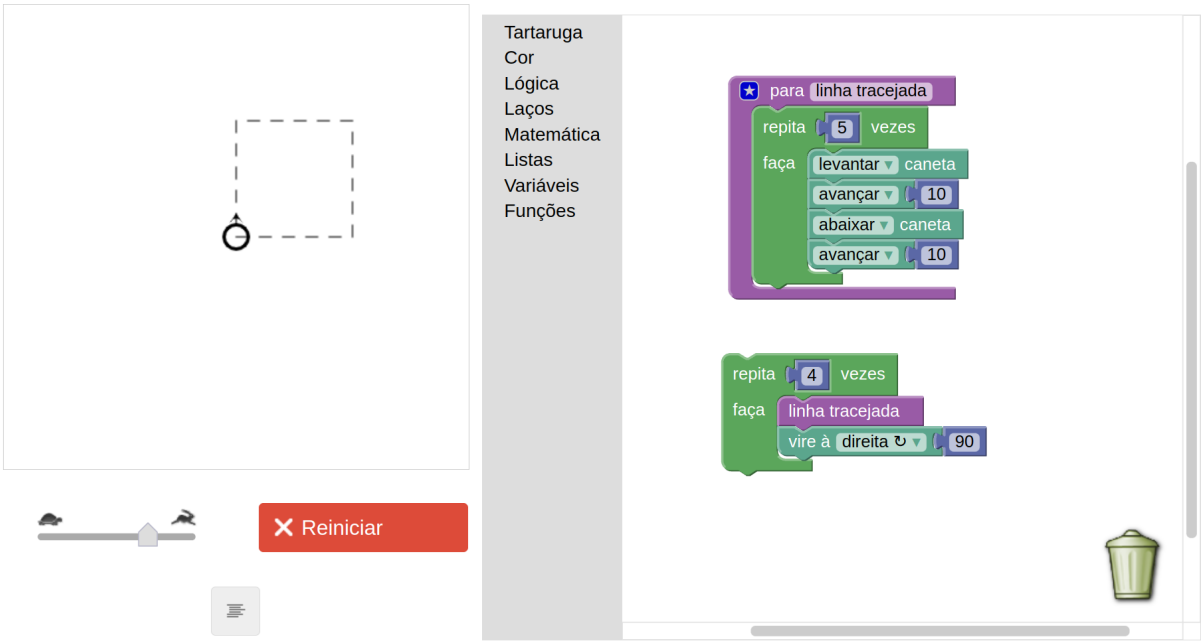
Altere o nome da função de faça algo para linha tracejada. Explique que esse será o nome do comando que o programa aprenderá, e que usaremos esse nome sempre que quisermos desenhar uma linha tracejada.



Diga: *"Agora vamos testar o novo comando da tartaruga"*. Para isso, acesse novamente o menu Funções, mas agora escolhendo a função chamada linha tracejada. Arraste-a para qualquer lugar na tela (Atenção: não arraste para junto da própria definição da função, mas sim como um bloco separado). Veja abaixo:



Peça aos alunos que insiram várias vezes o comando linha tracejada, para que entendam que a tartaruga aprendeu um novo comando. Agora lance um **desafio**: *"Façam um quadrado com linhas tracejadas usando o novo comando que ensinamos para a tartaruga"*. **Espere** alguns minutos e dê a solução:

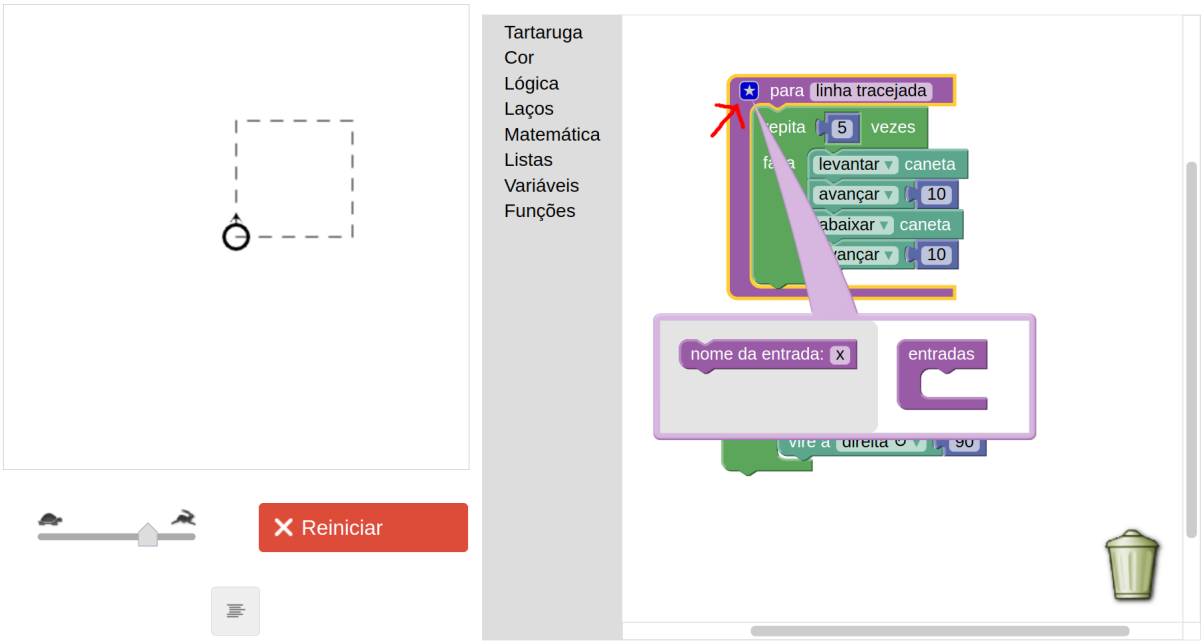


Opcional (neste estágio do tutorial): Na próxima parte do tutorial, os alunos aprenderão sobre polígonos e mais sobre funções, inclusive algo muito importante na Matemática: que as funções podem ter parâmetros de entrada. No entanto, se desejar, é possível introduzir a ideia de parâmetros de entrada neste momento, fazendo algumas modificações à função linha tracejada.

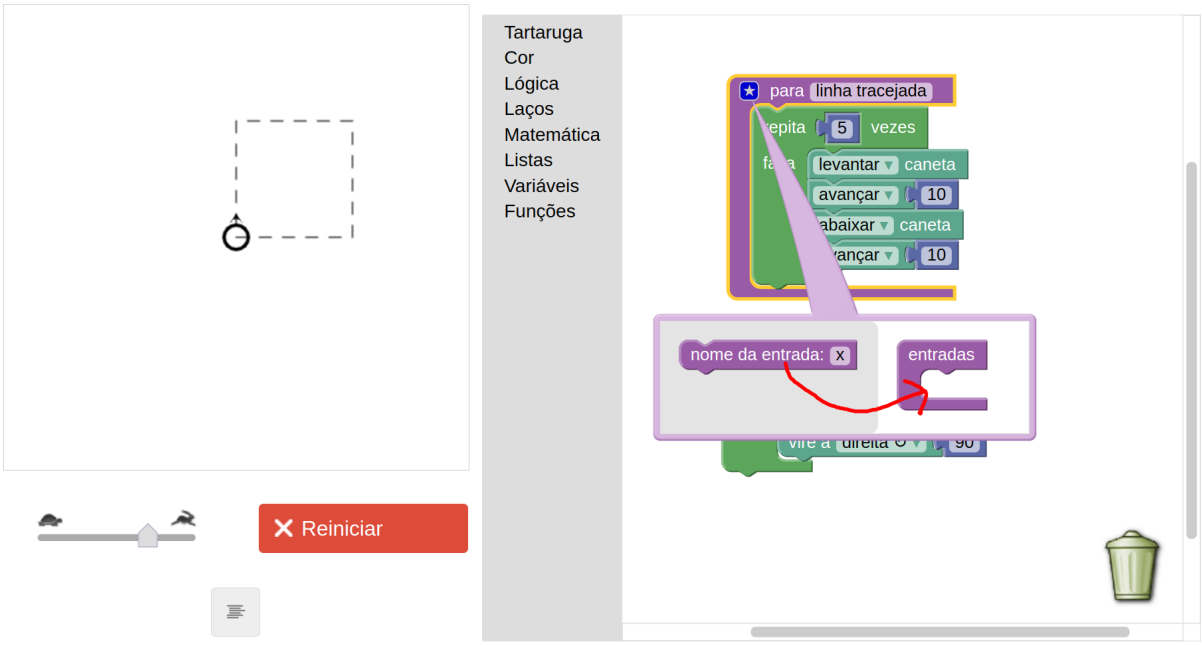
Para isso, pergunte aos alunos o seguinte: *"E se quiséssemos ensinar o programa a desenhar linhas tracejadas, mas de modo que pudéssemos dizer a ele quantos traços devem ser feitos?"* Dê exemplos na lousa de chamadas[8] à função linha tracejada, mas passando para ela a quantidade de traços. Por exemplo, escreva na lousa: linha tracejada 6, e desenhe uma linha com 6 traços. Em seguida, escreva na lousa linha tracejada 7, e desenhe uma linha com 7 traços. Faça quantos exemplos achar necessário. Em seguida, mostre como podemos modificar nossa função linha tracejada no Blockly Turtle para que receba a quantidade de traços. Para isso:

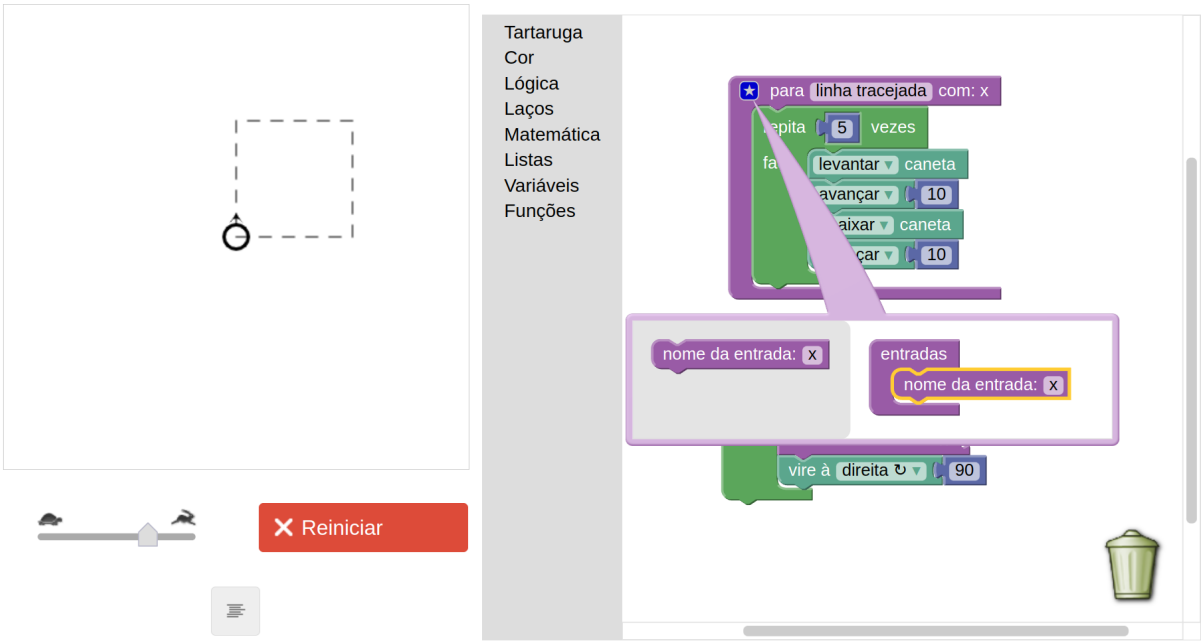
Clique no botão azul em forma de estrela no bloco de definição da função.

[8] "Chamada a uma função" é o termo que se usa quando colocamos para executar uma função que definimos, como fizemos ao arrastar o bloco *"linha tracejada"* à área de programação.

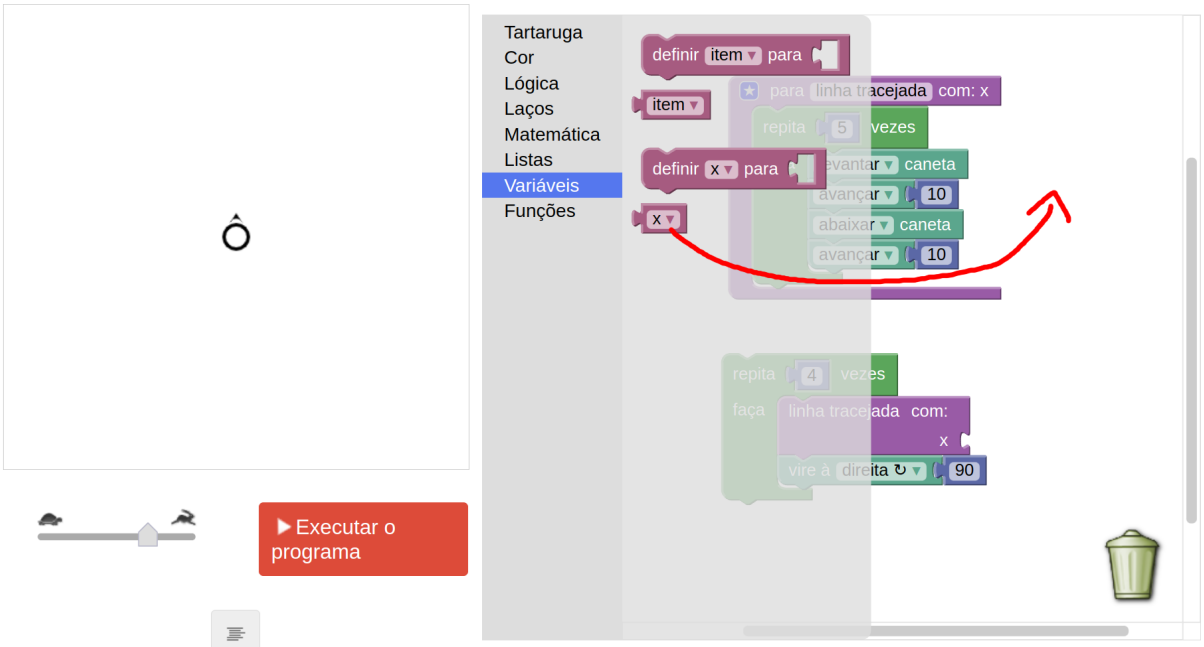


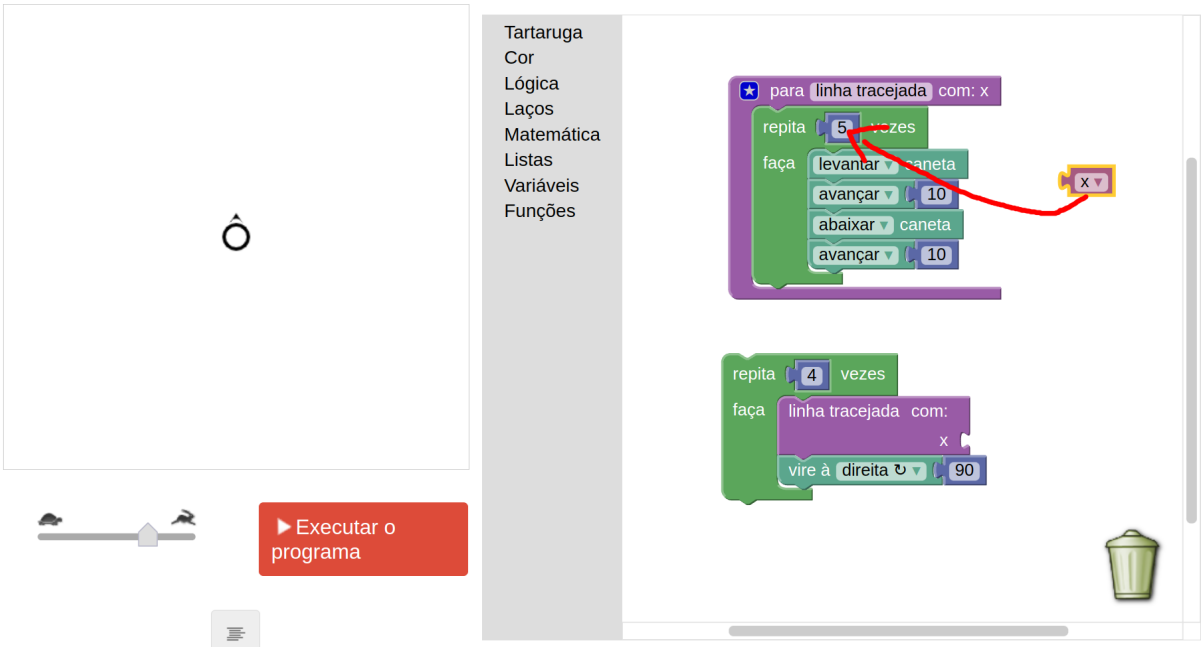
Aparecerá uma caixinha com um bloco que define o nome da entrada (o nome da variável de entrada). Deixe o nome x e explique que esse x será usado dentro da função para se referir à quantidade de traços. Arraste esse bloco de nome da entrada para dentro do bloco entradas, do lado direito da caixinha.



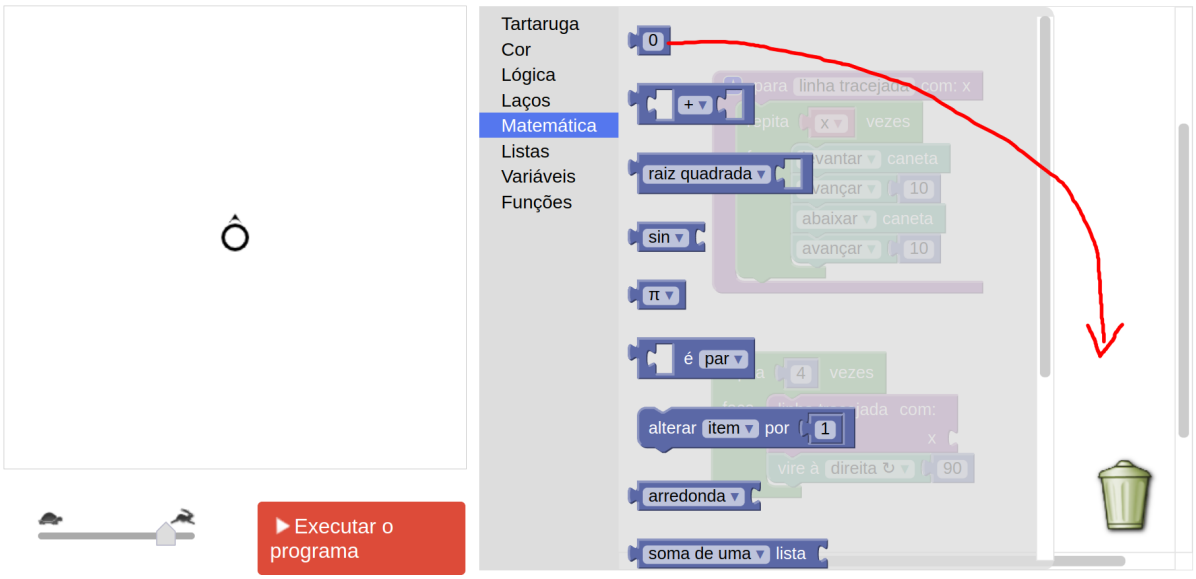


Para fechar a caixinha das variáveis de entrada, basta clicar novamente na estrela azul. Agora, já será possível acessar a variável x por meio do menu **Variáveis**. Arraste-a para o bloco repita dentro da função, substituindo o número 5 (quantidade de repetições de traço).





Não esqueça de alterar também a chamada da função, que agora requer que você encaixe um valor de entrada para x (você pode criar valores numéricos por meio do menu **Matemática**, como ilustrado abaixo).




Tartaruga
Cor
Lógica
Laços
Matemática
Listas
Variáveis
Funções
para linha tracejada com: x
repita x vezes
faça levantar caneta
avançar 10
abaixar caneta
avançar 10
repita 4 vezes
faça linha tracejada com: x
vire à direita ↻ 90
0
Executar o programa


Tartaruga
Cor
Lógica
Laços
Matemática
Listas
Variáveis
Funções
para linha tracejada com: x
repita x vezes
faça levantar caneta
avançar 10
abaixar caneta
avançar 10
repita 4 vezes
faça linha tracejada com: x
vire à direita ↻ 90
7
Executar o programa


Tartaruga
Cor
Lógica
Laços
Matemática
Listas
Variáveis
Funções
para linha tracejada com: x
repita x vezes
faça levantar caneta
avançar 10
abaixar caneta
avançar 10
repita 4 vezes
faça linha tracejada com: x
vire à direita ↻ 90
7
Executar o programa

Altere na chamada da função o valor do parâmetro (variável) passado, colocando diferentes valores, como 7, 3, 5, etc. Veja na última imagem acima como fica com "x=7".

Seguindo a mesma linha de raciocínio, é possível também definir um segundo parâmetro: a largura de cada traço. Basta repetir os mesmos passos para criar um parâmetro chamado larg (por exemplo). Veja abaixo como ficaria:

Faça testes com diferentes valores para x e larg, deixando claro o quanto é interessante parametrizar funções, pois permite criar diferentes formas de linhas tracejadas com pouco esforço.

Polígonos regulares e ângulos

Abra uma nova aba ou peça para os alunos jogarem na lixeira os blocos feitos até agora. Se estiver usando o Blockly Turtle pelo site https://helioh2.github.io/turtle, clique em  para fazer o *download* do projeto, a fim de salvá-lo para poder restaurá-lo depois, se desejar. Se estiver usando o Blockly Turtle pelo site https://blockly.games/turtle, é possível salvar o projeto clicando no botão com formato de elo de corrente ( ). Uma URL (endereço web) será gerada, que pode ser copiada para o e-mail, de modo que bastará abrir novamente essa URL (copiando-a para a barra de navegação de seu browser) mais tarde para restaurar o projeto.

Vamos agora fazer os alunos brincarem um pouco com diferentes polígonos regulares. Explique que o quadrado é um polígono de 4 lados, mas que existem outras formas geométricas com diferentes quantidades de lados. Pergunte aos alunos se eles conhecem algum outro polígono. Provavelmente dirão "triângulo", ou talvez até mesmo "hexágono", etc. Diga que vamos desenhar um triângulo, e faça junto com eles:

`repita 3 vez` (questione, *"por que 3"*?)

`faça [avançar 50, vire à direita 120]` (diga que em breve você vai explicar o porquê de ser 120)

Pergunte: *"Por que 'repita 3 vezes'?"*. *"E quanto ao ângulo, por que 120?"* Deixe em mistério por enquanto e explique que logo entenderão o porquê do 120 para o ângulo do triângulo, e o porquê do 90 para o ângulo do quadrado.

Peça para executarem. Agora lance um **desafio**: *"E se eu quiser fazer um hexágono? Um hexágono tem 6 lados. Como eu poderia começar?"* Deixe eles começarem e ajude somente aqueles que não entenderam. Logo, alguns deles vão tentar colocar '`repita 6 vezes [avançar`

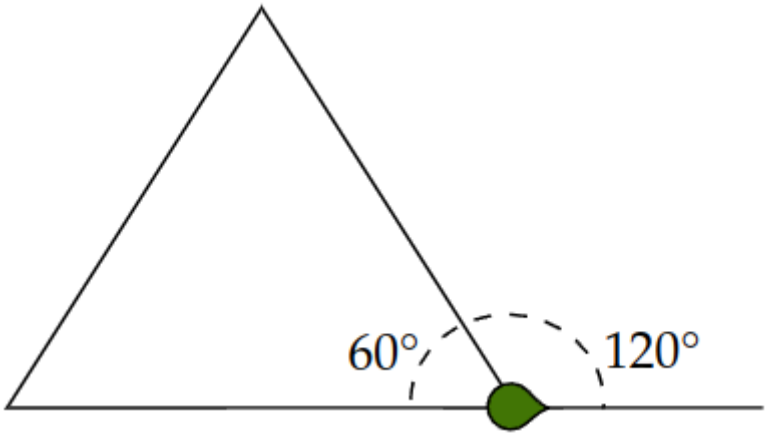
50, vire à direita ...'], mas provavelmente não conseguirão inserir o ângulo correto. Escreva na lousa o código para cada um dos polígonos que já conseguiram fazer:

```
repita 4 vezes [avançar 50, vire à direita 90] → Quadrado (desenhe o quadrado)
```

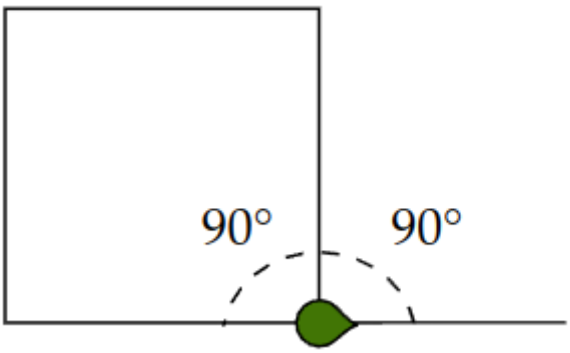
```
repita 3 vezes [avançar 50, vire à direita 120] → Triângulo (desenhe o triângulo)
```

```
repita 6 vezes [avançar 50, vire à direita ???] → Hexágono (desenhe o hexágono)
```

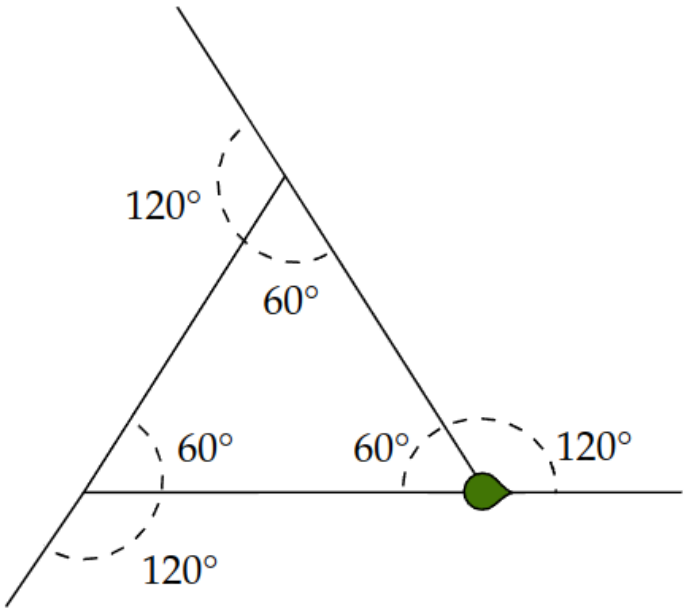
Neste ponto os alunos talvez fiquem curiosos para descobrirem qual o ângulo correto para o hexágono. É uma oportunidade para, na lousa, explicar como funcionam os ângulos de um polígono. Por exemplo, desenhe um triângulo e mostre quantos graus a tartaruga tem que girar após traçar um lado. Com isto, é possível explicar aos alunos a questão dos **ângulos suplementares** e dos **ângulos internos** de um polígono regular. Mostre que os ângulos internos de cada ponta do triângulo têm 60º, e que a tartaruga tem que girar 120º, pois 180º - 60º = 120º, que é o ângulo suplementar de 60º.



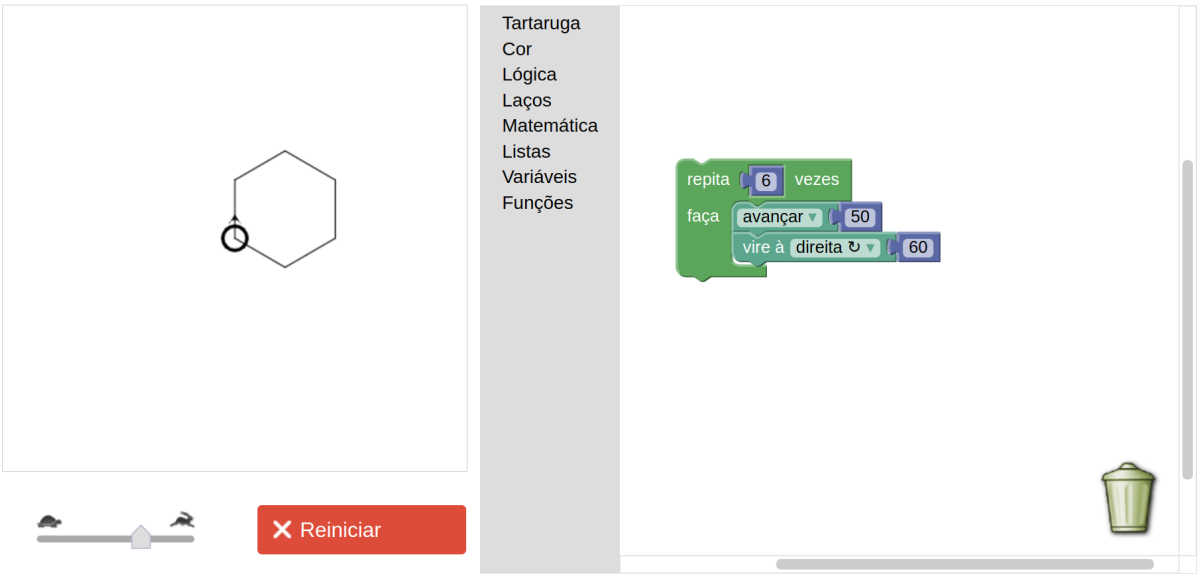
Faça o mesmo para o quadrado, mostrando que o ângulo suplementar de 90º é 90º.



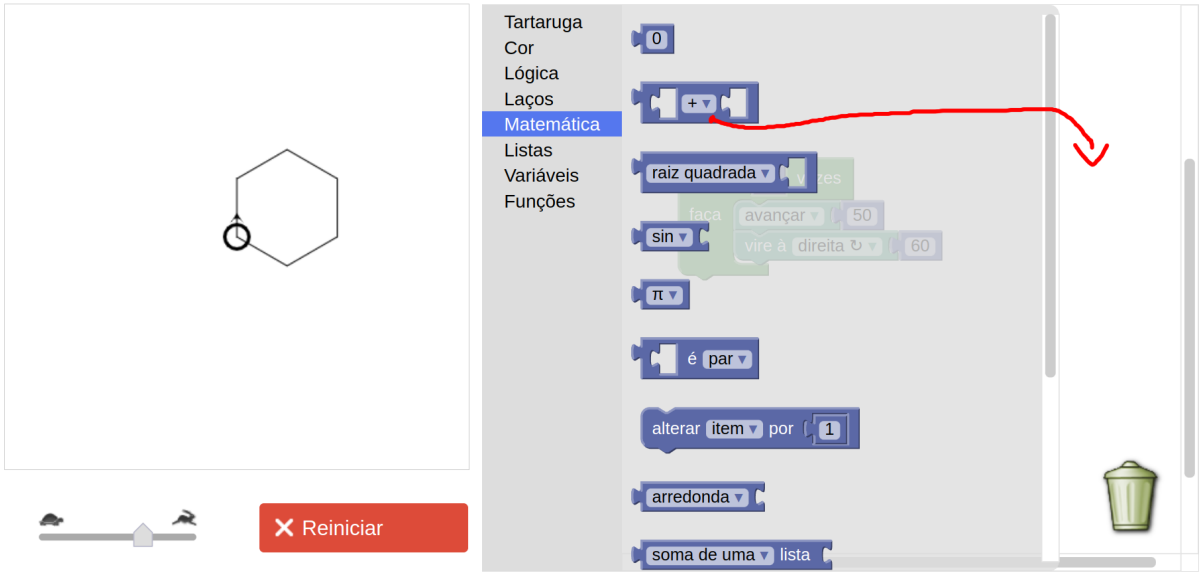
Mostre também que a soma dos ângulos internos sempre totaliza 180º, e que se somarmos os ângulos suplementares dos ângulos internos teremos 360º (por exemplo, 120 + 120 + 120, no caso do triângulo). Assim, você pode chegar à conclusão de que para 4 lados temos que girar 90º, pois 360º/4 = 90º, e que para 3 lados temos que girar 120º, pois 360º/3 = 120º.

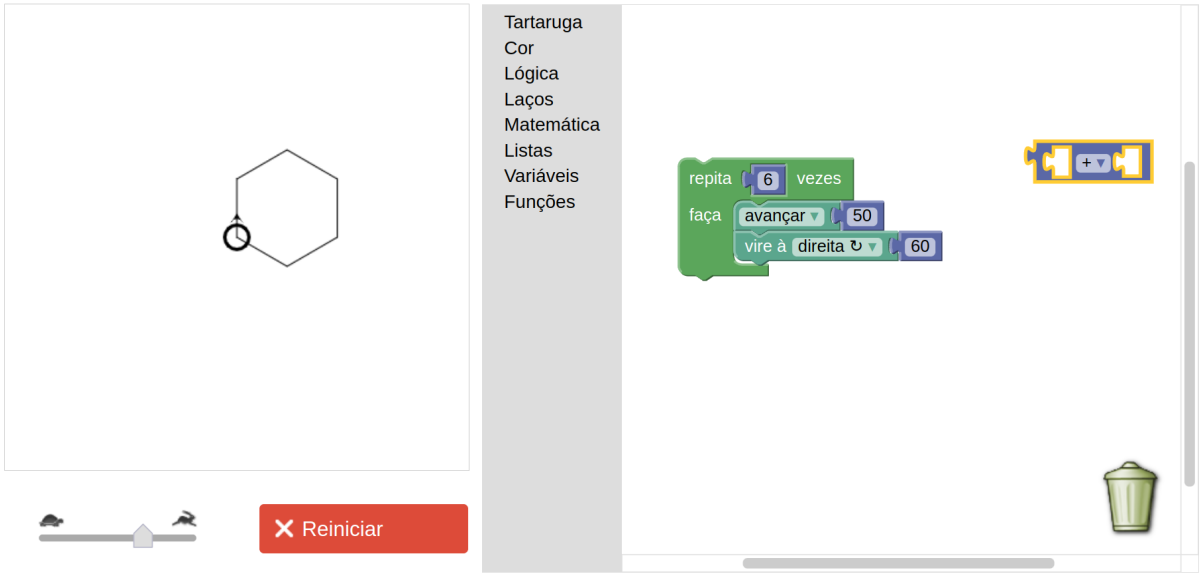


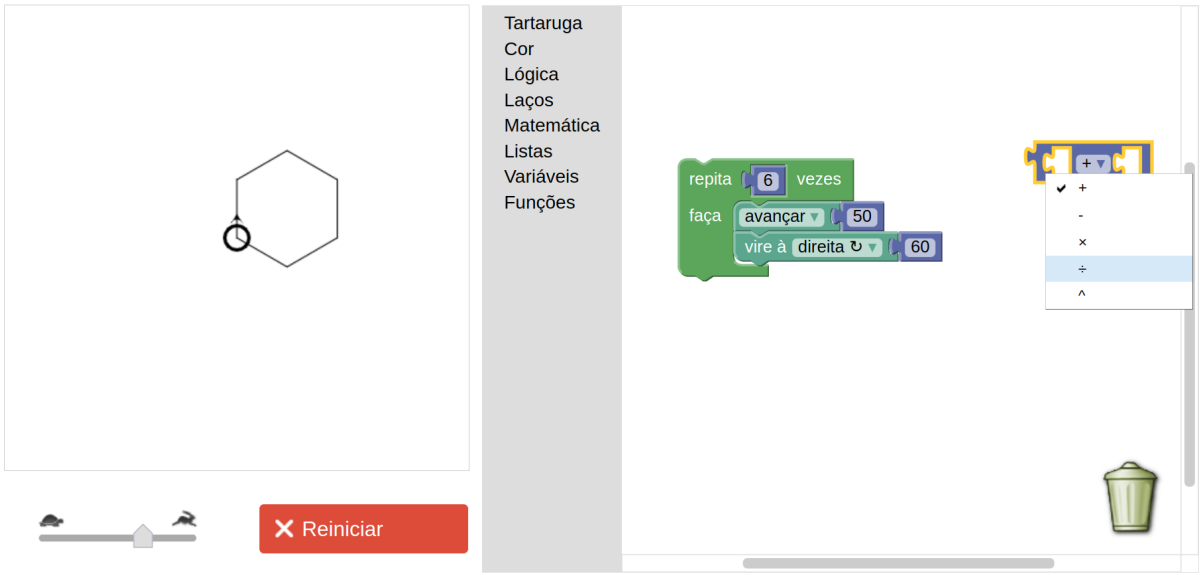
Tendo evidenciado esse padrão/fórmula ($360/lados$), peça aos alunos que calculem o ângulo que a tartaruga tem que girar para o hexágono. Deve-se chegar à conclusão que é 360º/6 = 60º. Feito isso, complete o código e peça que os alunos executem:

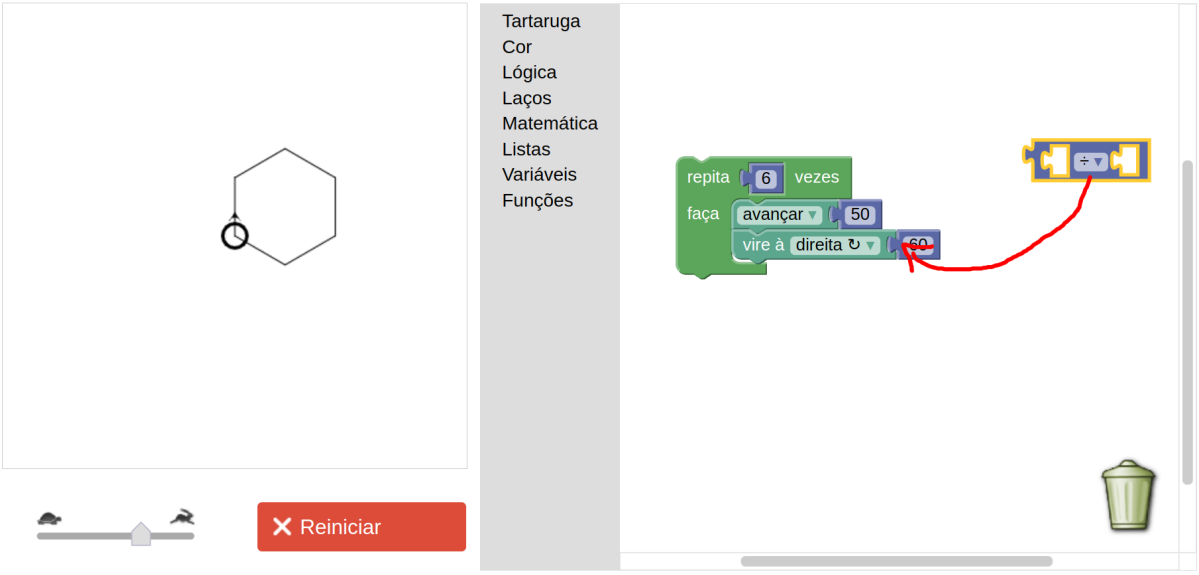


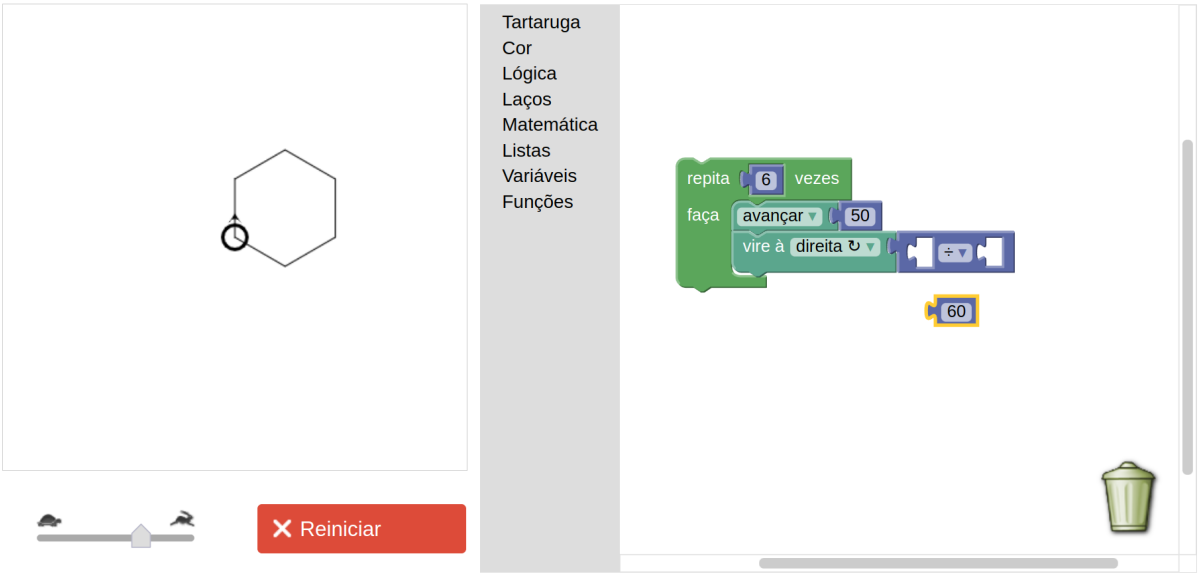
Pode ser interessante também mostrar que é possível, na programação, colocar um bloco de cálculo em vez do valor já calculado. Por exemplo, em vez de colocar vire à direita 60, podemos colocar vire à direita 360/6 (360 dividido por 6). Para isso, basta encaixar, no lugar do número 60, o bloco de operação aritmética encontrado no menu Matemática, escolhendo-se o operador "÷" (divisão). Veja abaixo:

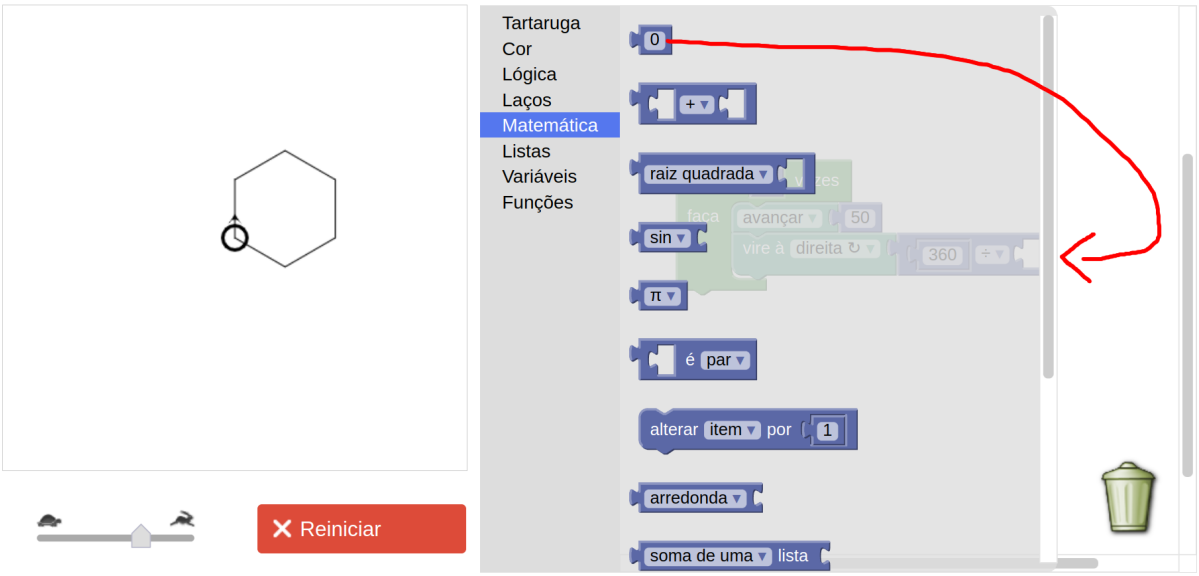

Tartaruga
Cor
Lógica
Laços
Matemática
Listas
Variáveis
Funções
0
+
raiz quadrada
sin
π
é par
alterar item por 1
arredonda
soma de uma lista
faça
avançar 50
vire à direita 60
Reiniciar

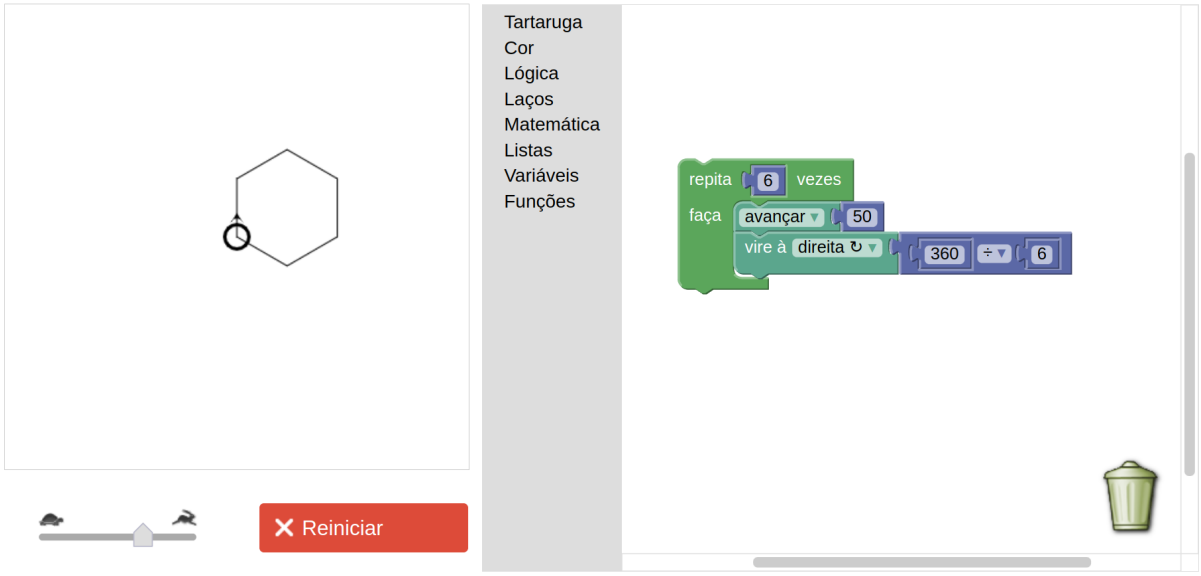

Tartaruga
Cor
Lógica
Laços
Matemática
Listas
Variáveis
Funções
repita 6 vezes
faça
avançar 50
vire à direita 60
+
Reiniciar


Tartaruga
Cor
Lógica
Laços
Matemática
Listas
Variáveis
Funções
repita 6 vezes
faça
avançar 50
vire à direita 60
+
-
×
÷
^
Reiniciar


Tartaruga
Cor
Lógica
Laços
Matemática
Listas
Variáveis
Funções
repita 6 vezes
faça avançar 50
vire à direita 60
÷
Reiniciar


Tartaruga
Cor
Lógica
Laços
Matemática
Listas
Variáveis
Funções
repita 6 vezes
faça avançar 50
vire à direita
÷
60
Reiniciar


Tartaruga
Cor
Lógica
Laços
Matemática
Listas
Variáveis
Funções
0
+
raiz quadrada
sin
π
é par
alterar item por 1
arredonda
soma de uma lista
Reiniciar



Uma vez que isso tenha sido compreendido, lance o desafio para os alunos fazerem o pentágono, o heptágono e o octógono. Elogie e recompense os que conseguirem desenhar mais. Depois, escreva na lousa as soluções dos desafios propostos:

```
repita 4 vezes [avançar 50, vire à direita 90] → Quadrado
repita 3 vezes [avançar 50, vire à direita 120] → Triângulo
repita 6 vezes [avançar 50, vire à direita 60] → Hexágono
repita 5 vezes [avançar 50, vire à direita 72] → Pentágono
repita 7 vezes [avançar 50, vire à direita 51] → Heptágono
repita 8 vezes [avançar 50, vire à direita 45] → Octógono
```

Pergunte: *"E se eu quiser fazer um hexágono maior, o que tenho que fazer?"*. Espere pela resposta e mostre que basta aumentar a distância com que se anda para frente:

```
repita 6 vezes [avançar 100, vire à direita 60] → Hexágono
```

Agora você terá que trabalhar com o aluno a **identificação de padrões**, uma das habilidades relacionadas ao **pensamento computacional**. Comece a raciocinar com os alunos da seguinte forma: *"O que tem de comum nos códigos de todos esses polígonos?"*. Vá termo por termo de cada linha perguntando: *"Isto muda?"*. Por exemplo:

```
repita 4 [avançar 50, vire à direita 90]
repita . . .
repita . . .
. . .
```

Pergunte: *"Dentre todos eles, o 'repita' muda?"*. A resposta será **"não"**.

```
repita 4 vezes [avançar 50, vire à direita 90]

repita 3 vezes . . .

repita 6 vezes . . .

. . .
```

Pergunte: *"Dentre todos eles, esse número muda (se referindo à quantidade de lados)?"*. A resposta deve ser **"sim"**.

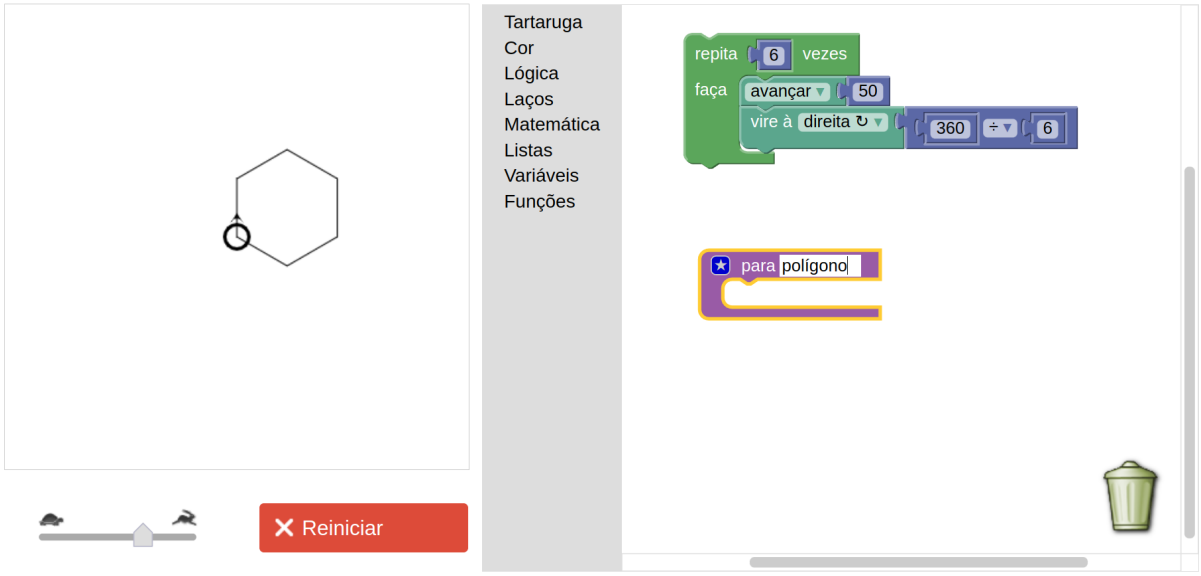
Abaixo de todas as linhas, copie cada termo que não muda, e, no lugar dos termos que mudam, escreva um ponto de interrogação (ou outro símbolo que represente algo que não sabemos, ou que pode acomodar algum valor, como uma "caixinha" ou "quadradinho"). No final, essa última linha ficará assim:

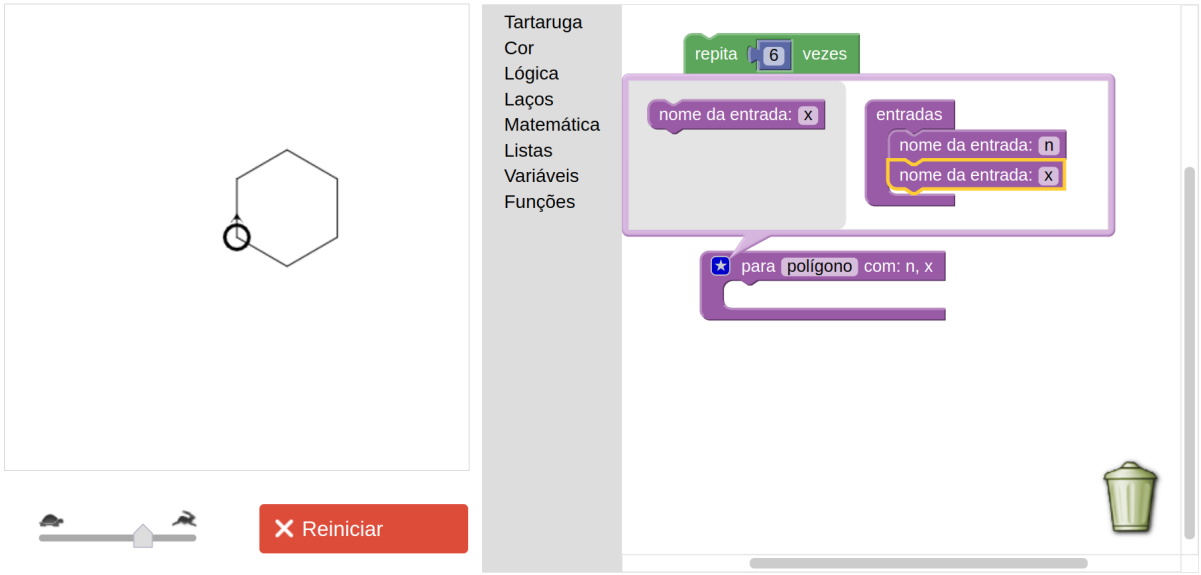
```
repita ? vezes [avançar ?, vire à direita ?]
```

Frente a cada interrogação, questione: *"O que vai nessa interrogação?"*, ou *"O que podemos colocar para substituir a interrogação?"*. Reflita com os alunos, com base nos exemplos de polígonos escritos acima, o que cada número no lugar da interrogação representa. Tente com eles chegar à conclusão de que: a primeira interrogação diz respeito à ***quantidade de lados***; a segunda diz respeito ao ***tamanho dos lados***; e a última diz respeito à operação ***360÷(quantidade de lados)***. ao final, você deve demonstrar para o aluno que aquela expressão vai virar uma **fórmula** para desenhar qualquer polígono, desde que se saiba a quantidade de lados e o tamanho dos lados. Troque as interrogações por letras ou palavras da seguinte forma:

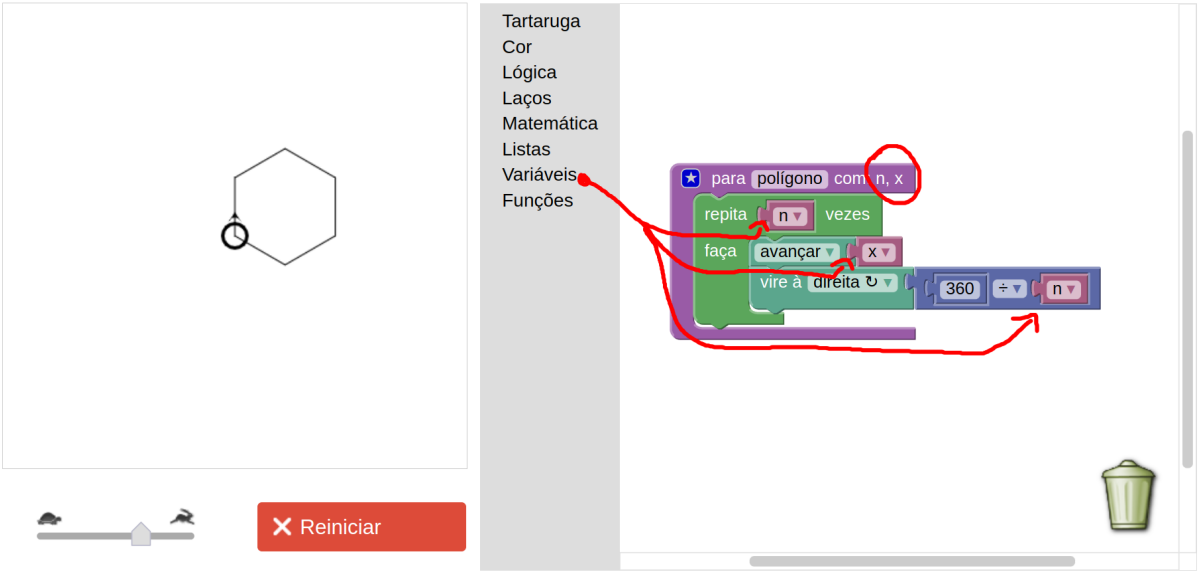
```
repita n vezes [avançar x, vire à direita 360÷n]
```

Aqui temos uma ótima oportunidade de introduzir o conceito de **variáveis**. Explique que uma variável é como se fosse uma caixinha com um nome escrito do lado de fora, mas que, dentro da caixa, tem um número (valor). Neste caso, cabe a nós definir os números que vão ser postos dentro dessas caixinhas. Para isso, vamos precisar definir **uma função** que recebe **parâmetros (variáveis) de entrada**. Em outras palavras, temos que fazer com que a tartaruga **aprenda** um novo comando chamado polígono. No entanto, sempre que formos pedir para ela executar esse comando, temos que informá-la quais são os valores que queremos para essas variáveis de entrada. Para isso, criaremos uma nova **função**, como já foi explicado anteriormente, mas definiremos que essa função recebe 2 (duas) variáveis de entrada: n e x, onde n representa a quantidade de lados do polígono, e x representa o tamanho de cada lado. Para definir as variáveis de entrada, basta clicar na estrela azul no bloco da definição da função (veja também o passo a passo no final da seção **Linhas Tracejadas**).



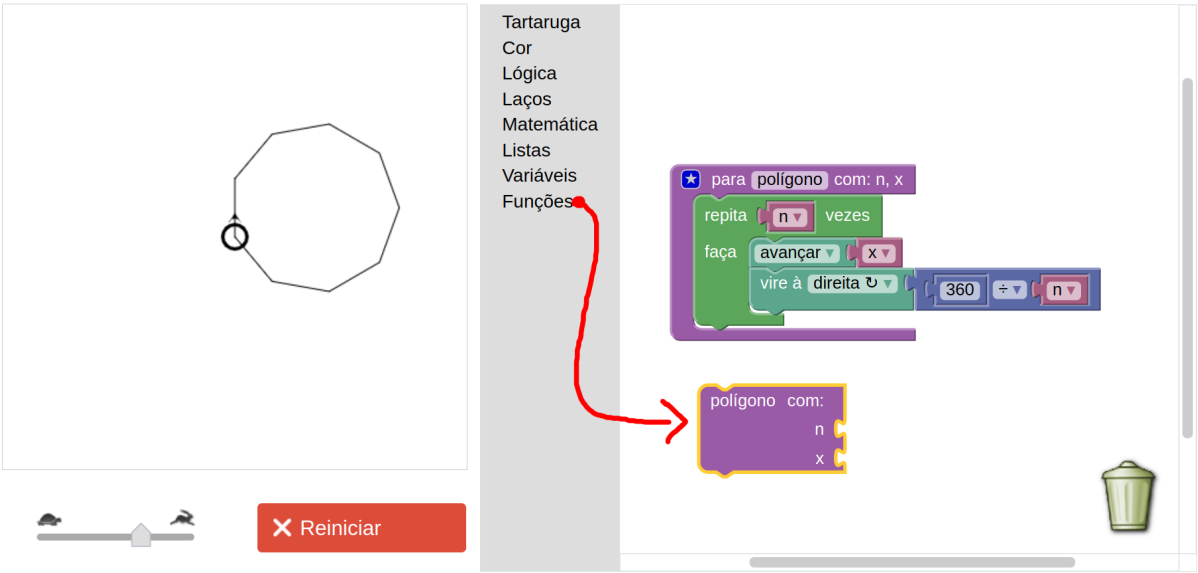


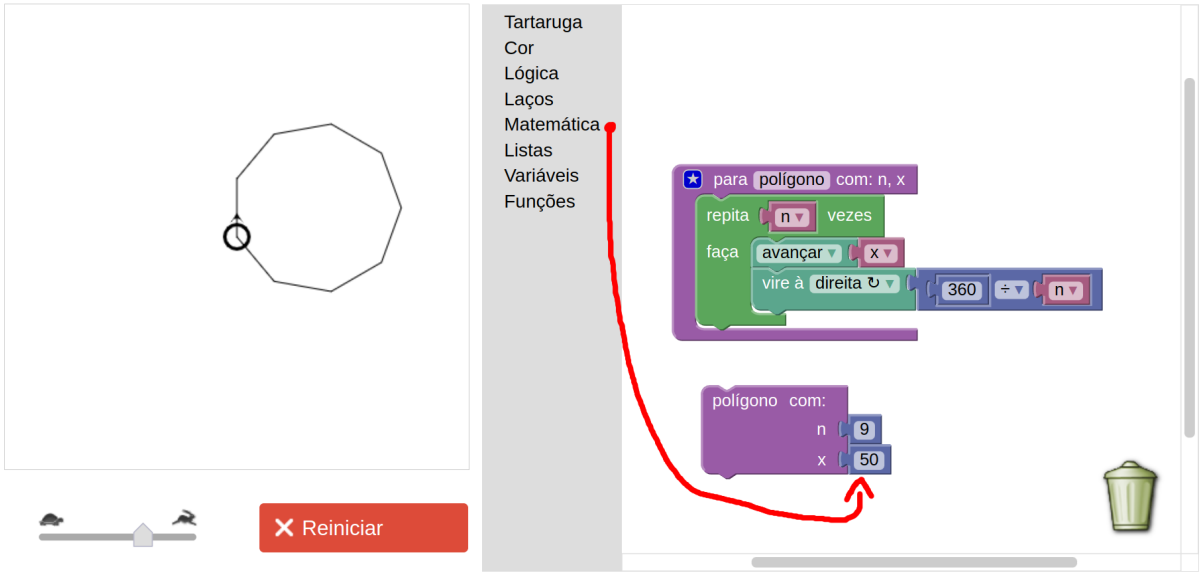
Veja abaixo como deve ficar a definição da função polígono. Note que é necessário arrastar as variáveis x e n para as posições corretas, pegando-as no menu Variáveis.



Explique que as variáveis são também chamadas de **parâmetros**, pois permitem passar valores para dentro da função.

Agora peça a eles que tentem pedir à tartaruga para que desenhe um polígono de 9 lados. Para isso, vá no menu Funções, arraste um bloco polígono para uma área em branco (Atenção: arraste para uma área em branco, separado da definição da função). Em seguida, encaixe os números referentes ao valor da variável (parâmetro) n e x. Valores numéricos podem ser criados a partir do menu Matemática, como já foi explicado. Veja abaixo:





Sempre lembre-se de reexecutar o programa para ter um *feedback*.

Solicite aos alunos que peçam à tartaruga que desenhe polígonos de diferentes lados com a nova função.

Um gancho para introduzir funções matemáticas

Lance mais um **desafio** aos alunos: peça a eles que façam a tartaruga aprender como desenhar um triângulo. Para isto, escreva na lousa alguns exemplos de como criar um triângulo:

polígono com n=3, x=50

polígono com n=3, x=100

polígono com n=3, x=150

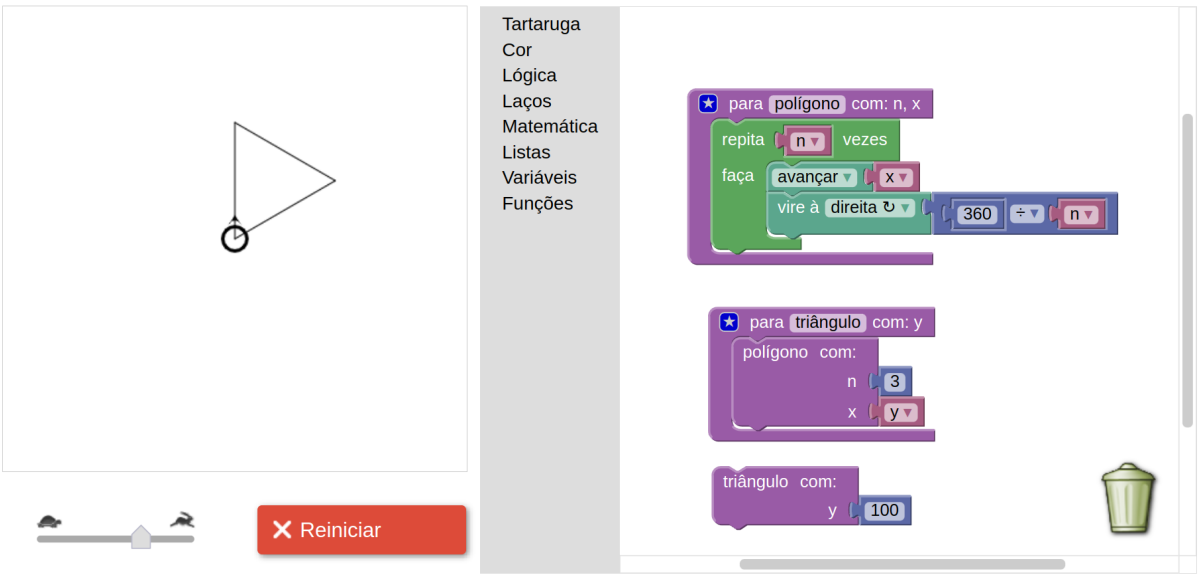
Mais uma vez, faça o procedimento de verificar quais os termos que mudaram e aqueles que permanecem inalterados em cada um dos exemplos. O resultado ficará assim:

polígono com n=3, x=??

Pergunte aos alunos o que representa a interrogação. Deve-se chegar à conclusão de que se trata do **tamanho dos lados**. Atribua um nome a essa "incógnita", por exemplo, y:

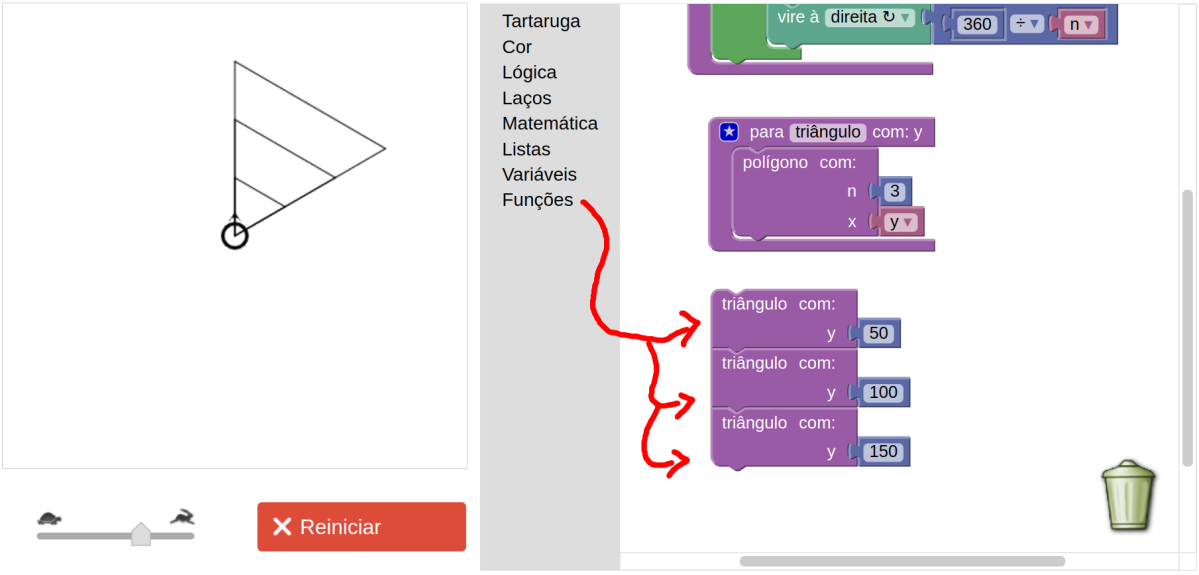
polígono com n=3, x=y

Agora peça aos alunos que escrevam a função triângulo, de modo similar ao que foi feito para o polígono. Espere uns minutos (para eles tentarem) e depois mostre a solução:



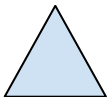
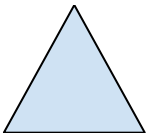
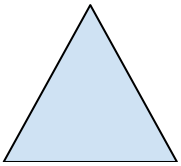
Note que temos duas funções definidas: polígono, que já tínhamos definido anteriormente, e a nova função triângulo, que utiliza a função polígono, mas que só possui um único parâmetro (variável) de entrada: y, que define o tamanho dos lados do triângulo.

Agora solicite aos alunos que peçam à tartaruga que desenhe vários triângulos com diferentes valores para y, bastando arrastar a função triângulo para a área de código:



Assegure-se de que eles percebam que diferentes números que você passa como variável y para a função "triângulo" resultam em triângulos de diferentes tamanhos. Esta é uma oportunidade para iniciar a explicação sobre **funções matemáticas**:

Faça na lousa uma tabela mais ou menos assim:

y	triângulo(y)
20	
40	
60	
80	

Perceba que se trata de uma tabela de função, cujo *domínio* é um número e cuja *imagem* é uma **forma geométrica**! Acreditamos que essa seja uma maneira interessante de explicar como funcionam as funções matemáticas, conforme também constataram autores como Schanzer, Felleisen e Krishnamurthi[9], dentre outros pesquisadores, os quais propuseram uma abordagem para o ensino de álgebra com programação chamada *Bootstrap.*

Você pode mostrar como a ideia é parecida com funções que só envolvem números. Por exemplo *f(x) = 2x*:

x	f(x)
1	2
2	4
3	6
4	8

Desafio 1: como um desafio um pouco mais interessante, peça aos alunos que definam uma função chamada polígono tracejado, desenhando um polígono usando a linha tracejada definida anteriormente. Dica: A ideia é parecida com o quadrado tracejado que fizemos anteriormente — basta trocar o comando avançar pela chamada à função linha tracejada.

[9] SCHANZER, Emmanuel et al. Transferring skills at solving word problems from computing to algebra through Bootstrap. In: Proceedings of the 46th ACM Technical symposium on computer science education. 2015. p. 616-621.

Desafio 2: peça aos alunos para criarem funções para quadrado, hexágono e pentágono, do mesmo modo que fizeram para triângulo.

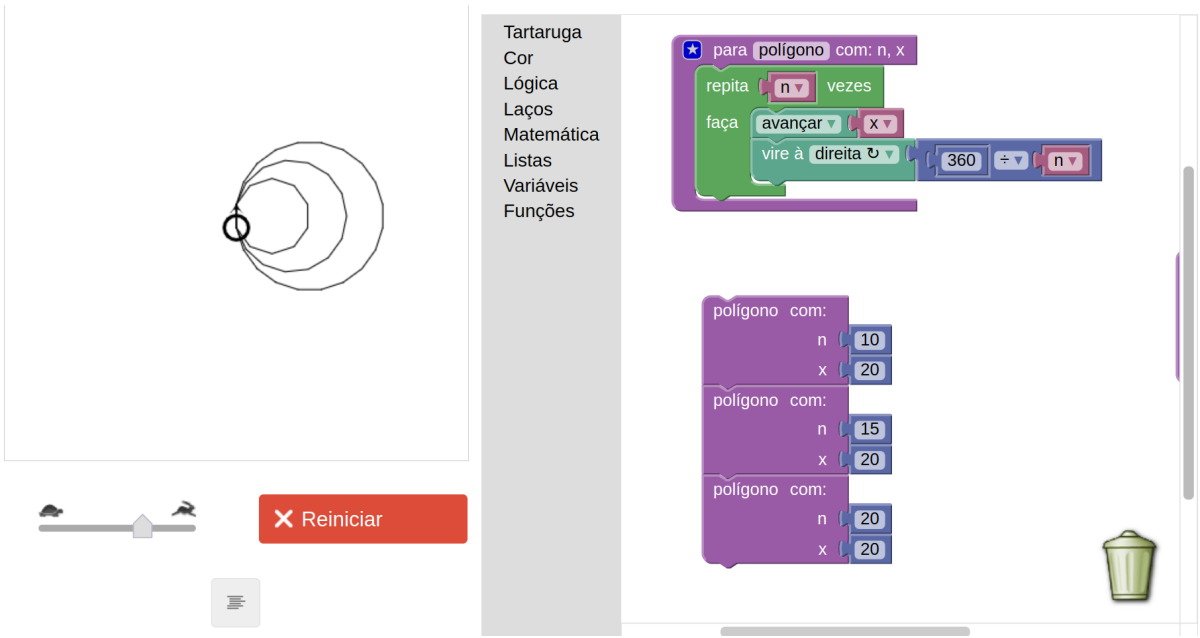
De polígonos para círculos

Peça aos alunos que tentem criar polígonos com mais lados, por exemplo:

polígono com n=10, x=20

polígono com n=15, x=20

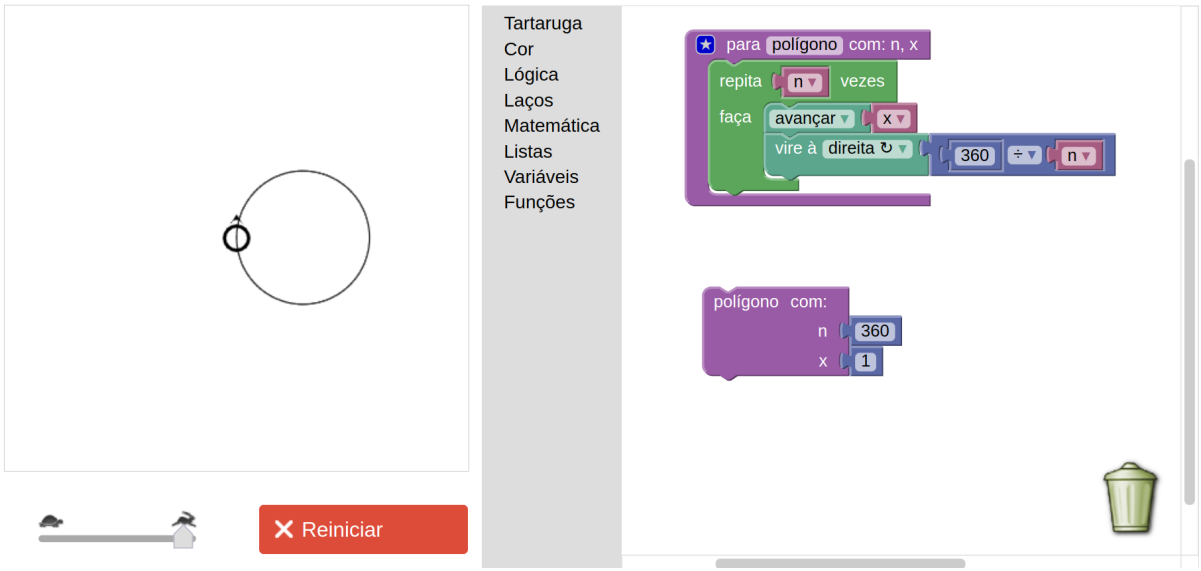
polígono com n=20, x=20



Por si próprios, os alunos começarão a perceber que, quanto mais lados tiver um polígono, mais parecido ele ficará com um círculo! Assim, pergunte: *"Então, se eu quiser fazer um círculo perfeito, preciso fazer o quê?"*. Instigue-os a responderem que basta colocar mais lados. Não reprima se algum aluno disser algo do tipo: *"É só fazer 1 milhão de lados"*. Ele(a) não estará errado, por isso concorde com ele(a) — quanto mais lados, mais perfeito será o círculo. No entanto, deixe claro que levaria muito tempo para desenhar 1 milhão de lados, e por isso temos que escolher um número menor.

Explique a eles que todo círculo é um arco de 360º. Esclareça o que é um arco, desenhando arcos com diferentes ângulos na lousa, e então mostre por que o círculo tem 360º. Portanto, mostre que seria interessante se pudéssemos fazer o círculo por meio de um polígono com 360 lados. Neste ponto, alguns dos alunos já terão tentado fazer o polígono, porém com tamanhos de lados muito grandes. Não reprima esses alunos, apenas explique que os tamanhos dos lados também devem ser bem pequenos, ou o círculo ficará grande demais para caber na tela. Crie com eles o seguinte código:

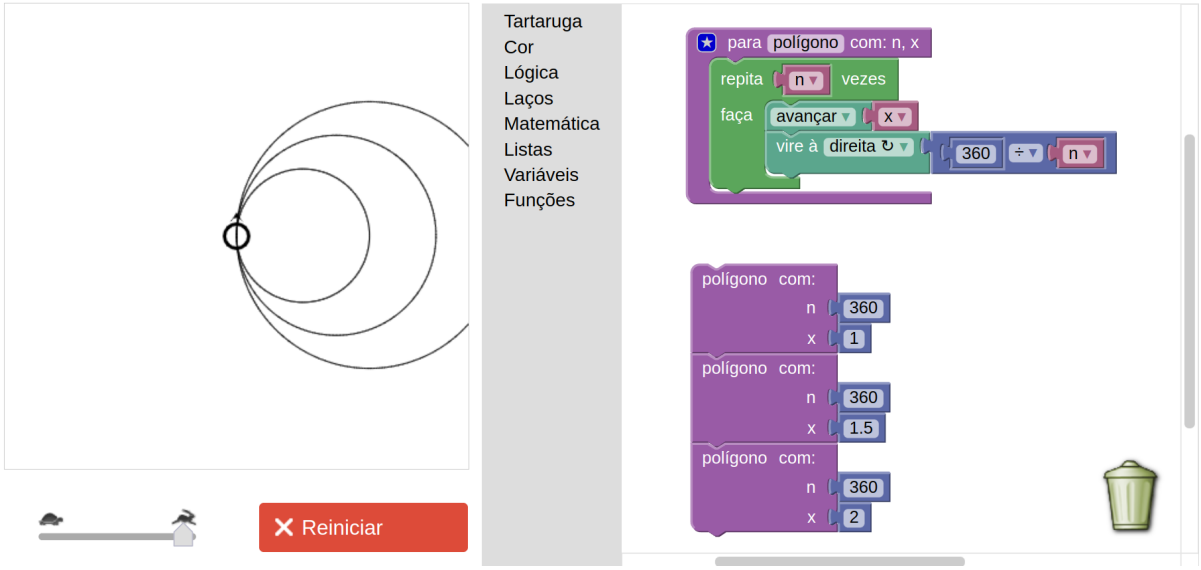
polígono com n=360, x=1



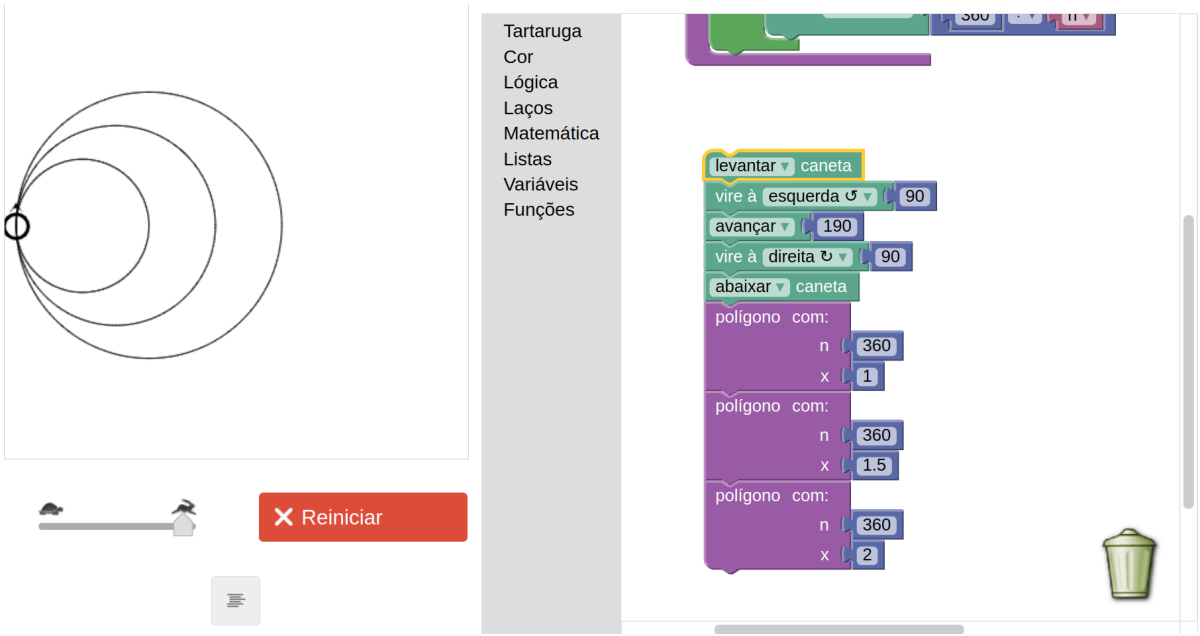
Execute com a velocidade máxima (use o *slider* ao lado do botão de execução). Esse comando desenhará um círculo na tela. Peça a eles para aumentarem aos poucos o tamanho dos lados, visando obter círculos maiores.

polígono com n=360 x=1.5 (explique que 1.5 é "1,5", ou seja, "1 e meio")

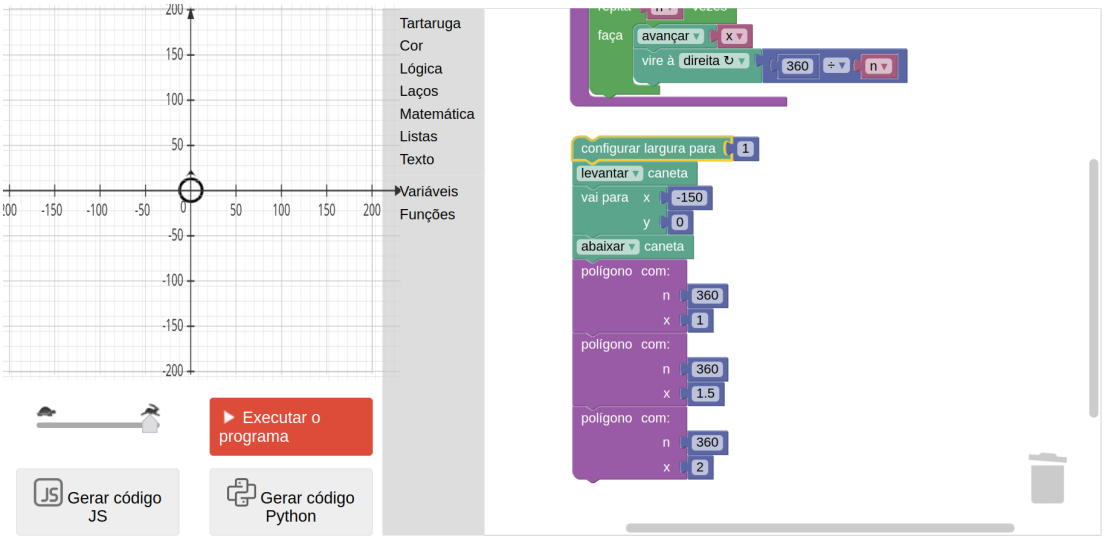
polígono n=360 x=2



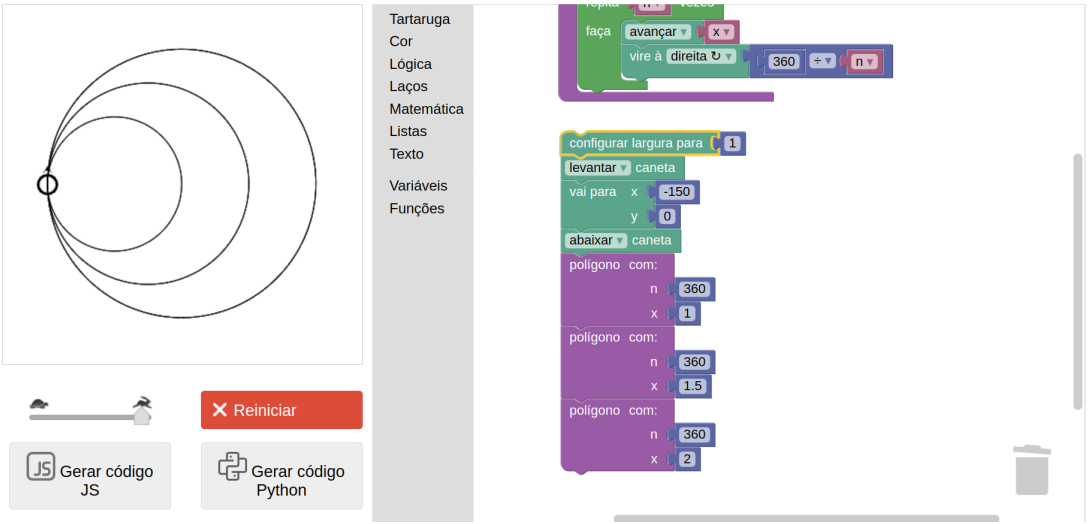
Talvez alguns vão indagar sobre o fato de o círculo maior ser tão grande que "sai da tela". Para corrigir esse problema, adicione um código antes dos blocos que desenham os círculos para levar a tartaruga para a parte mais à esquerda da tela. Isso pode ser feito levantando-se a caneta, fazendo-a girar 90° para a esquerda, avançando 190 passos, girando-se 90º para a direita, e por fim abaixando a caneta. Desse modo, a tartaruga terá mais espaço para desenhar. Veja:



Opcional: Se você estiver usando o site https://helioh2.github.io/turtle, você também pode usar o comando vai para x ... y ..., disponível no menu Tartaruga. Esse comando recebe dois parâmetros, o primeiro sendo a coordenada *x* e o segundo a coordenada *y*. Se for utilizar esse comando, é interessante explicar sobre o **plano cartesiano**, e como o ponto exatamente no meio da tela é o ponto (0,0), e que a tela possui dimensões de 400 x 400 pixels. Portanto, o ponto mais à esquerda é determinado por *x=-200,* o mais à direita por *x=200*, o ponto na parte superior da tela é *y=200* e na parte inferior é *y=-200*. Veja abaixo uma ilustração do plano cartesiano sobreposto à tela:



Se quisermos, portanto, ir para a esquerda da tela, mantendo-se a altura (eixo *y*) centralizada, basta acrescentarmos o comando vai para -150, 0, o que faria a tartaruga se mover imediatamente para o ponto (*x=-150*, *y=0*). Portanto, aqui temos mais um gancho que você pode utilizar para explicar de maneira prática e interessante outro conceito matemático importante: **o plano cartesiano.**



Agora é o momento de explicar que, geralmente, não desenhamos círculos baseados no tamanho do lado do polígono de 360 lados, mas sim com base no **raio do círculo**. Explique que sempre que queremos desenhar um círculo, usamos o raio. Se você tiver um compasso grande que possa ser usado na lousa, é o modo perfeito de explicar. Se você não tiver um compasso grande, mas tiver compassos pequenos, distribua aos alunos e peça a todos que façam um círculo do mesmo tamanho — assim eles perceberão que conseguem fazer sempre o mesmo círculo se souberem o valor do raio.

Outra alternativa é, caso haja apenas um compasso, fazer uma demonstração em um papel sobre uma mesa, chamando os alunos para observarem em volta da mesa. Caso não disponha de um compasso, use a criatividade para demonstrar que é possível desenhar círculos iguais se sabemos o tamanho do raio. Por exemplo, trace duas linhas perpendiculares, isto é, com ângulo de 90º entre elas, uma na vertical e outra na horizontal, e desenhe o arco de 90º em torno dessas linhas; faça o mesmo para os lados opostos de cada linha (formando uma cruz, como no plano cartesiano), e desenhe os outros arcos de 90º até formar uma circunferência. Com isso, demonstre que as 4 linhas que saem do centro têm o mesmo comprimento, que é o raio.

Portanto, explique que o ideal seria se pudéssemos ordenar à tartaruga algo como: “Desenhe um círculo de raio 50”.

> **Opcional:** Aqui pode ser interessante também, se houver espaço na sala, fazer uma **dinâmica de grupo** na qual um aluno (ou o próprio professor) fica parado em pé numa posição que você chamaria de centro, e outro aluno tomaria uma distância de alguns passos deste. Então pediria para o segundo aluno andar em volta do primeiro, de modo que não fique nem mais próximo nem mais distante do centro. Do mesmo modo que a dinâmica de grupo explicada na **Introdução**, você pode fazer de conta que um dos alunos é a tartaruga que recebe instruções. Então, depois de ter aprendido, a turma pode ordenar: “Fulano(a), faça um círculo com raio de 4 passos”, ou “Fulano(a), faça um círculo com raio de 2 passos”, etc.

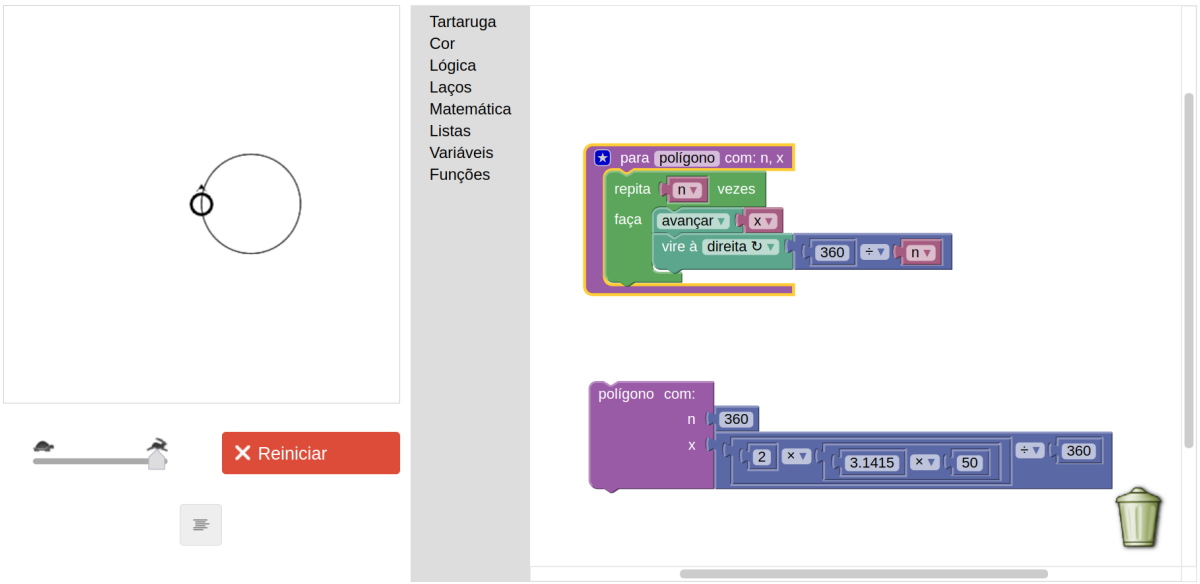
“Mas ainda temos um problema: como podemos descobrir o tamanho do lado do polígono/círculo com base no tamanho do raio?”. Aqui é o momento em que você deve

explicar sobre o comprimento da circunferência, usando a fórmula $2 \times \pi \times r$. Uma forma de explicar isso é usar uma animação, como a que se encontra em http://www.ajudaalunos.com/Quiz_mat/circulo_html/imagens/circulo_pi4.gif.

Assim, mostre que sabemos o quanto temos que fazer a tartaruga andar para que desenhe um círculo: basta calcular *c = 2*pi*raio*. Explique que *pi* é uma constante igual a 3,1415 (o que dá para perceber de modo bem interessante se você mostrar uma animação como a de cima).

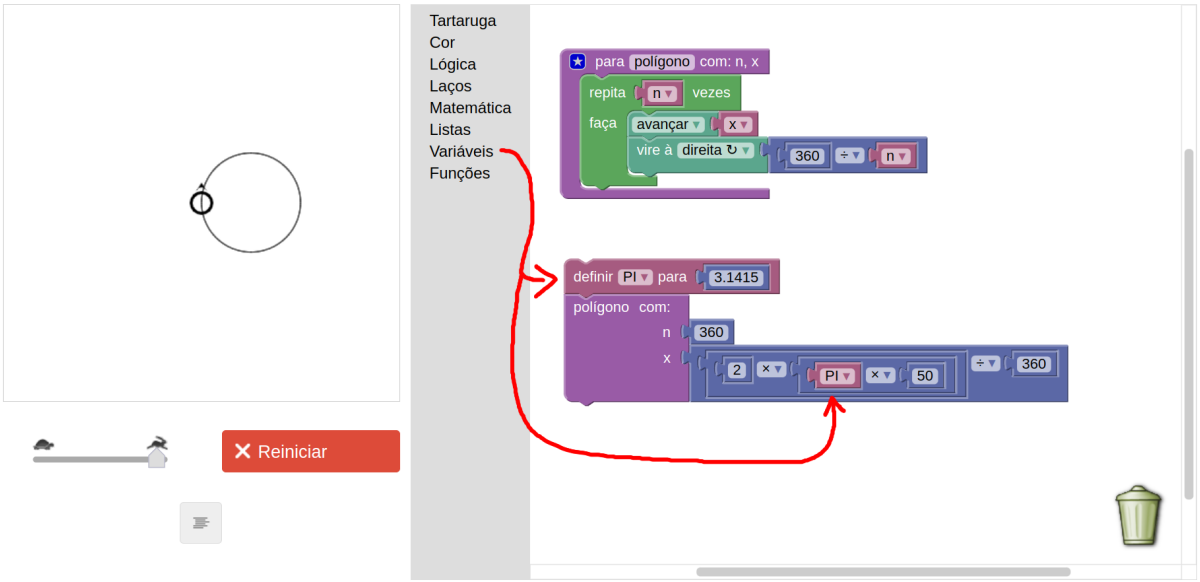
"Mas ainda temos um problema: a fórmula do comprimento da circunferência nos mostra apenas a quantidade total de passos que devem ser desenhados para fazer o círculo, mas nós queremos saber apenas a quantidade de passos em cada lado do polígono". Raciocine com eles para que se chegue à conclusão de que, se temos 360 lados, e o total que a tartaruga deve andar para desenhar o círculo é calculado pelo comprimento da circunferência (*c*), então temos que fazer uma divisão: temos que dividir o valor de *c* por 360. Tendo raciocinado assim com os alunos, desenha um círculo de raio 50 com o seguinte comando:

polígono com n=360 x=(2*3.1415*50)÷360 [use o menu Matemática para montar a expressão aritmética]

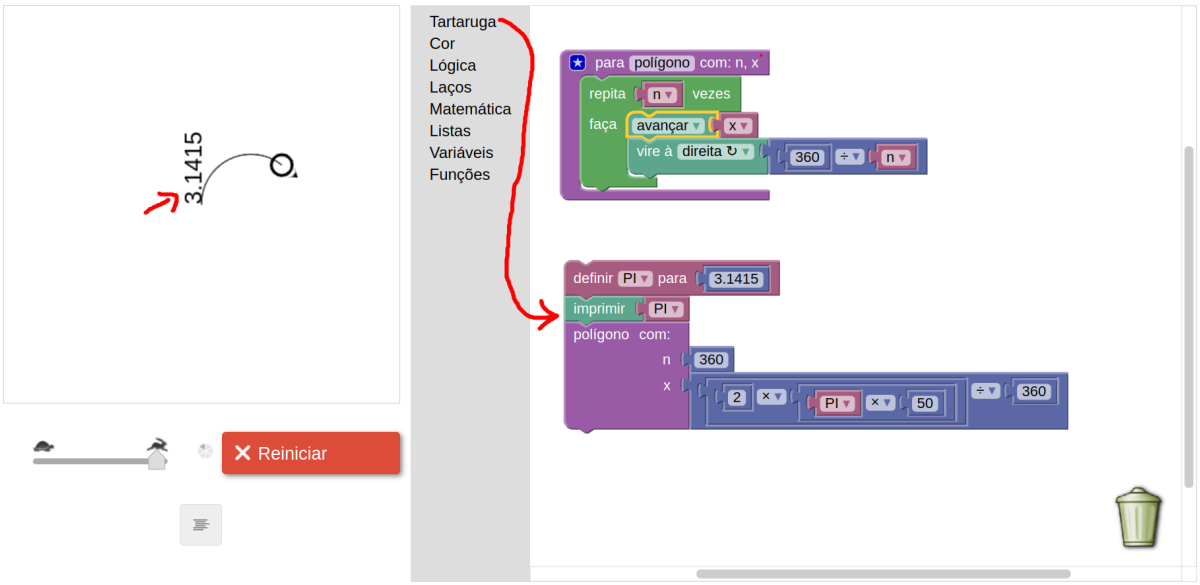


Atenção: não se esqueça de colocar a tartaruga na velocidade máxima por meio do *slider*.

Explique que, nesta linguagem, em vez de ',' (vírgula) usamos '.' (ponto) para números "quebrados"/decimais. Explique também a eles que o procedimento realizado está correto, mas pode ser aprimorado, visando um melhor entendimento. Não seria muito mais interessante se simplesmente pudéssemos chamar o nome PI em vez de 3.1415? Explique como podemos definir variáveis globais (fora de função), e defina a variável PI. Também substitua o 3.1415 na expressão aritmética pelo bloco da nova variável PI (veja o menu Variáveis).



Mostre também que é possível escrever na tela o valor de PI. Basta ir ao menu Tartaruga e incluir, após a definição da variável, um comando "imprimir". Troque o bloco de texto atrelado ao comando por um bloco da variável PI (acessível pelo menu Variáveis).



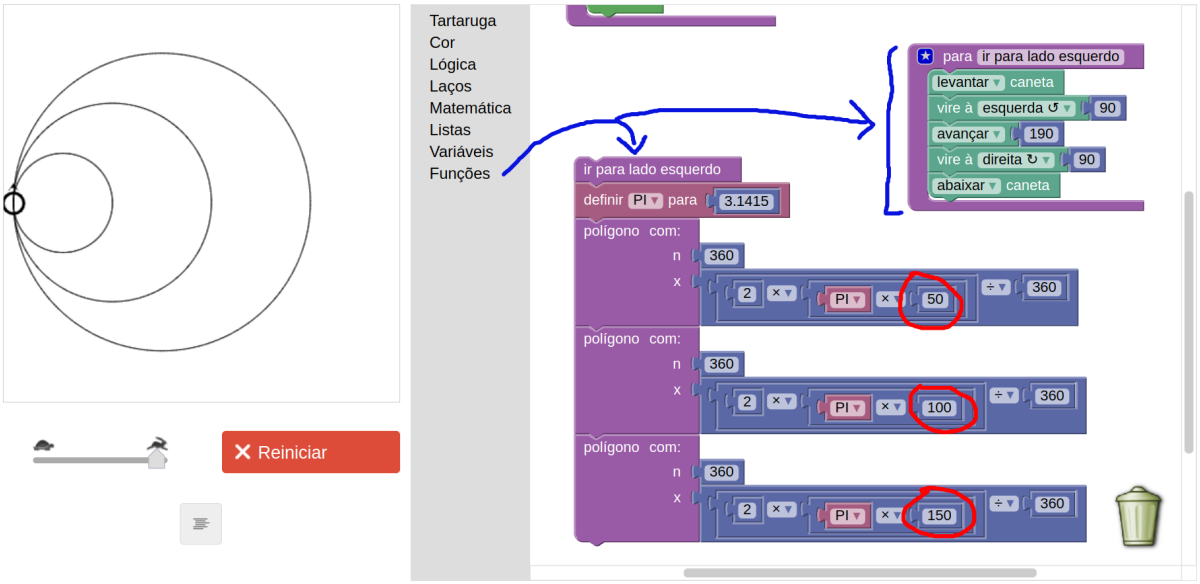
Explique novamente que uma variável é como se fosse uma caixinha com uma letra, ou um nome, no qual podemos colocar um valor dentro. Sempre que queremos colocar um valor em uma variável ("colocar dentro da caixinha"), devemos usar o comando definir, seguido pelo nome da variável. Se quisermos escrever qualquer coisa na tela, inclusive um valor de uma variável, devemos usar o comando imprimir. Note que há uma diferença entre a variável PI, que é uma variável global (ou **constante**), e a variável que é parâmetro de uma função, como já visto na função polígono, triângulo e linha tracejada. A primeira consiste em um valor acessível de qualquer lugar do programa. Já a segunda só é acessível dentro do escopo de uma função.

Voltando à chamada (pedido) da função polígono que está desenhando nosso círculo, questione os alunos sobre a dificuldade em ter que escrever todo esse procedimento quando queremos desenhar um círculo. Pergunte como poderíamos tornar isso mais fácil, tentando lembrá-los de que já foi feito algo parecido antes. Se não lembrarem, pergunte como eles fizeram para desenhar o triângulo sem precisar ficar chamando a função polígono o tempo todo e colocando o número 3 para o número de lados. Assim, lembre-os que podemos **criar funções**, isto é, podemos **ensinar** a tartaruga a executar um conjunto de comandos. Mostre como tinha sido feita a função triângulo da última vez, e escreva o código na lousa. Escreva também na lousa mais alguns exemplos de círculos, demonstrando aquilo que muda e o que fica inalterado:

polígono com n=360 x=(2*PI***50**)÷360

polígono com n=360 x=(2*PI***100**)÷360

polígono com n=360 x=(2*PI***150**)÷360

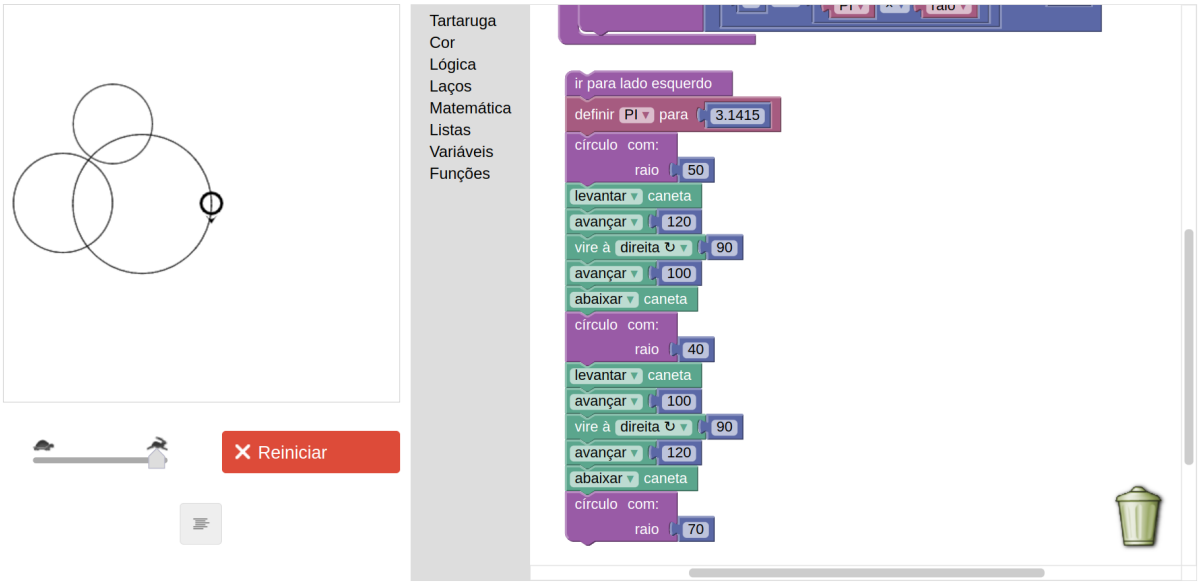


Na figura acima, também foi definida uma função ir para o lado esquerdo, que faz a tartaruga começar do lado esquerdo da tela. A lógica é a mesma da que foi feita algumas páginas atrás. Chame essa função antes de todos os comandos para que a tartaruga tenha espaço na tela para desenhar. Note aqui mais uma vez como a definição de funções torna o programa mais legível e modularizado.

Dê alguns minutos aos alunos para que tentem definir a função círculo. Parabenize os que conseguirem, estimule os que se esforçaram, e então mostre a solução, que consiste na definição da função círculo e na substituição das várias chamadas a polígono feitas anteriormente por chamadas à função círculo.

> **DICA:** é possível aproveitar o código da chamada à função polígono para definir a função círculo. Basta substituir o valor do raio [ex: 50, 100 ou 150] pela variável raio que deverá ser criada como parâmetro de entrada da função círculo.

Mostre para os alunos que é possível desenhar círculos de diferentes tamanhos em lugares diferentes da tela. Basta levantar a caneta, mover a tartaruga para o lugar desejado, abaixar a caneta e então desenhar. Veja um exemplo:



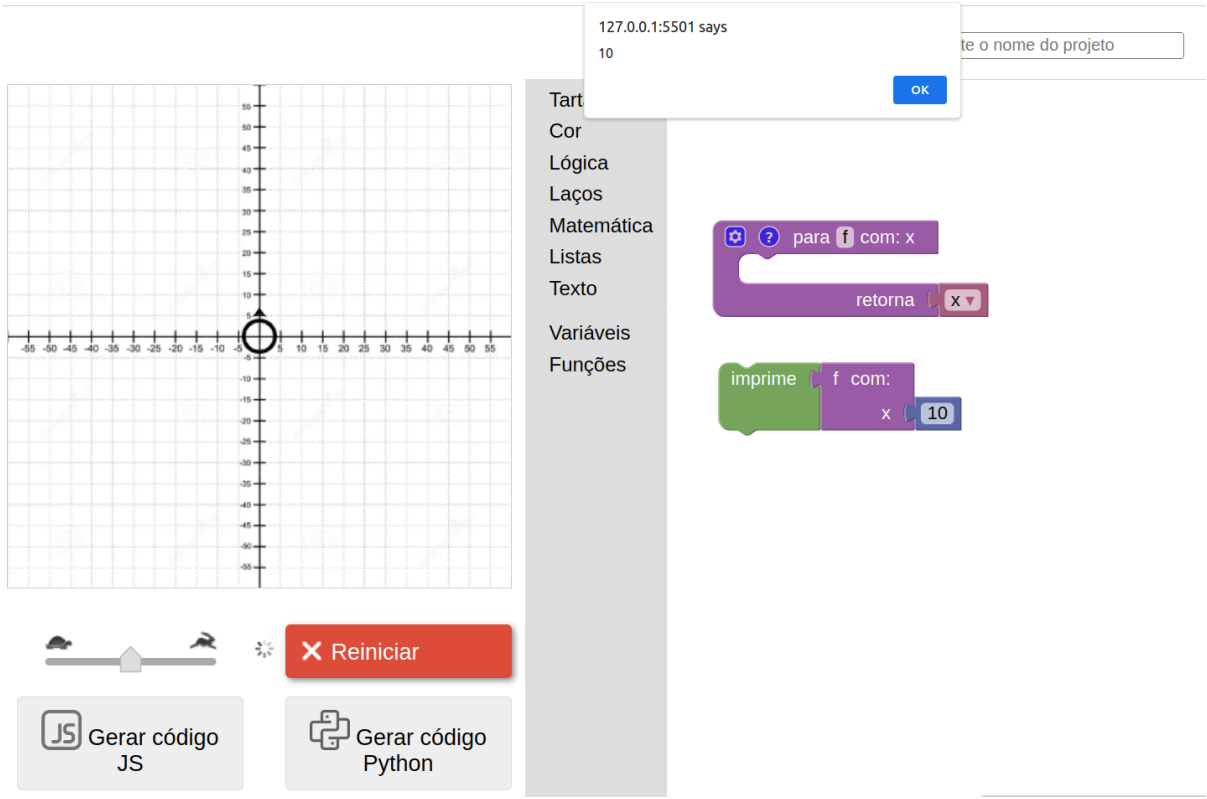
Plotagem de gráficos de funções no plano cartesiano

Uma vez compreendidos inicialmente os conceitos de funções e plano cartesiano, podemos explorar a ideia de plotar gráficos de funções. Essa parte do material só é possível de se realizar se você estiver utilizando o site https://helioh2.github.io/turtle/. Isso porque precisamos do comando vai para, somente disponível nesse site, e também porque somente nele temos o "modo cartesiano", que mostra as linhas e marcações representando o plano cartesiano.

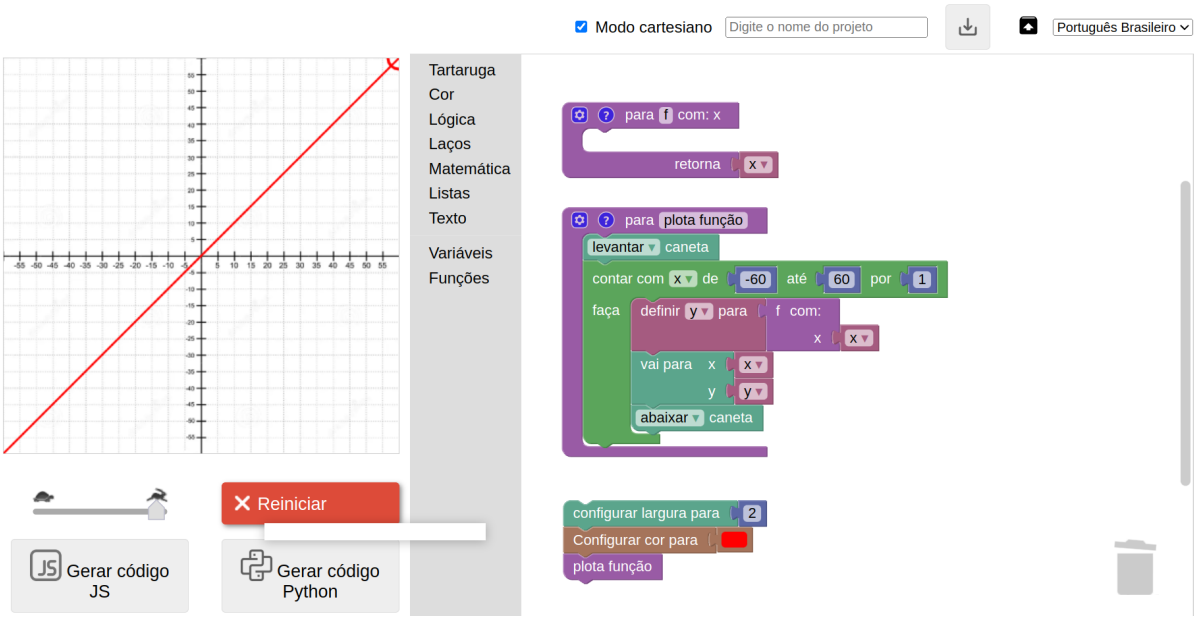
Na parte superior da tela, clique na opção "Modo cartesiano", você notará que aparecerá na tela da tartaruga as linhas dos eixos x e y e as marcações numéricas. É importante notar também que a escala nesse modo cartesiano vai de -60 a 60 no eixo x, e de -60 a 60 no eixo y. Isso é

diferente do modo padrão (sem o plano cartesiano), que, como já vimos no decorrer do livro, vai de -200 a 200 nos eixos x e y. É possível que, em versões futuras do site, seja possível personalizar essas escalas. Se isso ocorrer, essas opções estarão visíveis e acessíveis na parte superior da tela.

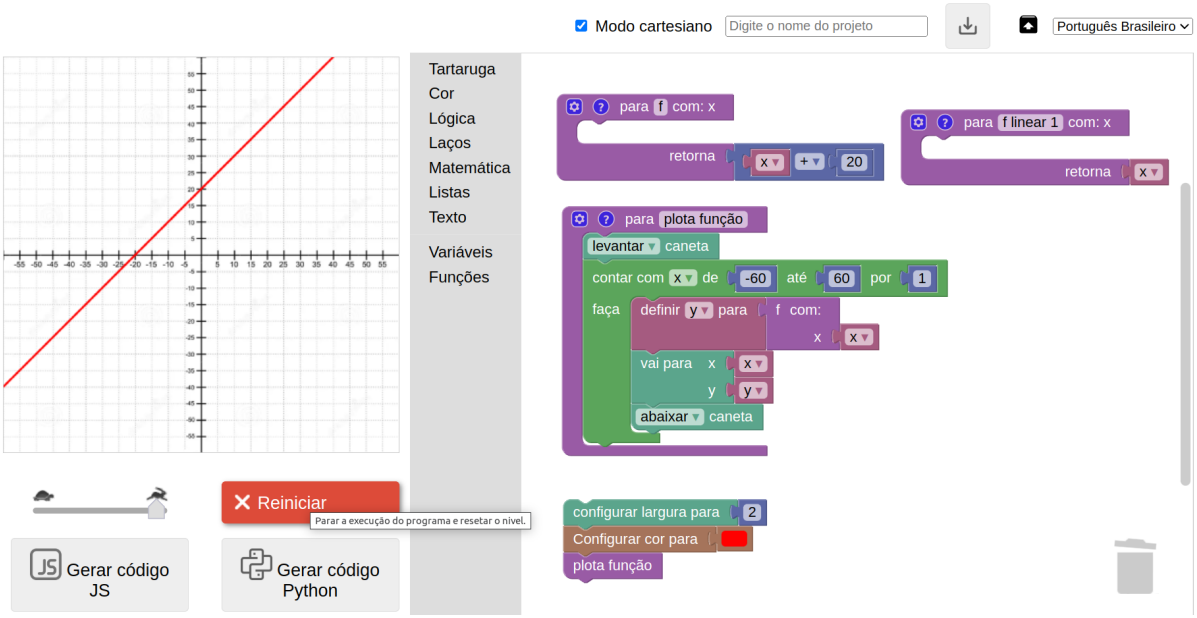
Peça que os alunos criem uma função matemática simples: $f(x) = x$. Para isso, basta definir uma função que retorna chamada f com um único parâmetro x. A função simplesmente retornará o valor de x.



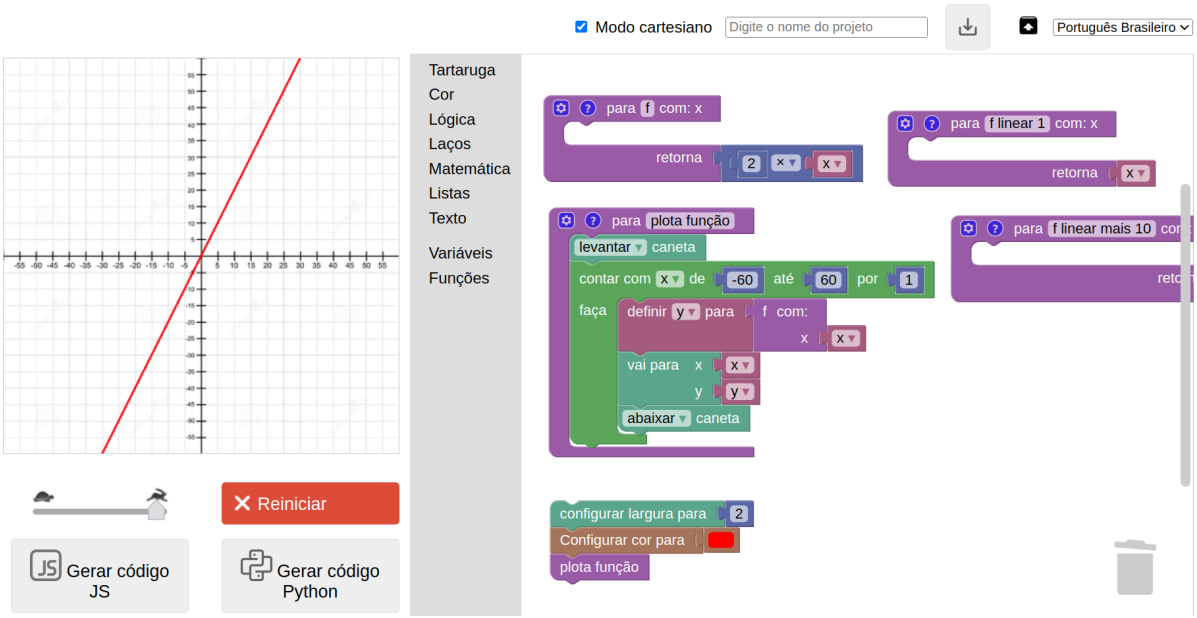
Agora, iremos escrever uma função que plota essa função, chamada plota função. O funcionamento da função plota função é o seguinte: usa-se um laço do tipo contar com... para pegar todos os valores possíveis de x entre -60 e 60, que é a escala do eixo-x do plano cartesiano. Para cada valor de x, então, calculamos o valor de y chamando a função f. Em seguida, usamos o comando vai para para fazer a tartaruga se mover até aquela posição. Também usamos o comando levantar caneta antes do laço, para que a tartaruga se mova até a posição inicial *(x=-60, y=f(x))* sem desenhar na tela, e o comando abaixar caneta dentro do laço para que desenhe a reta/curva resultante.



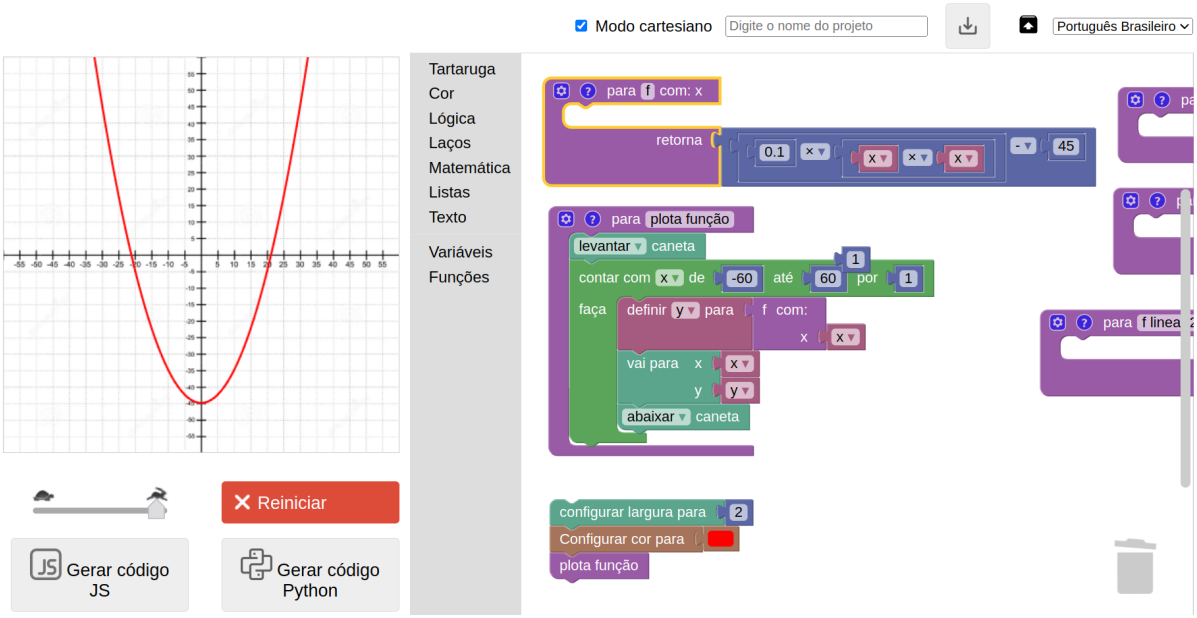
Agora, basta alterar a função f para criar diferentes funções. Sugere-se criar uma cópia da função atual e renomeá-la para poder facilmente refazer a função feita anteriormente. Para isso, clique com o botão direito sobre a definição da função f e escolha a opção "Duplicar". Note que com isso será criada uma função exatamente igual, porém chamada f2. Renomeie essa função para — por exemplo — f linear 1. Então, edite a função f original para representar outra função matemática. É importante lembrar que a função plota função sempre executará a função que possui o nome f. Veja alguns exemplos:



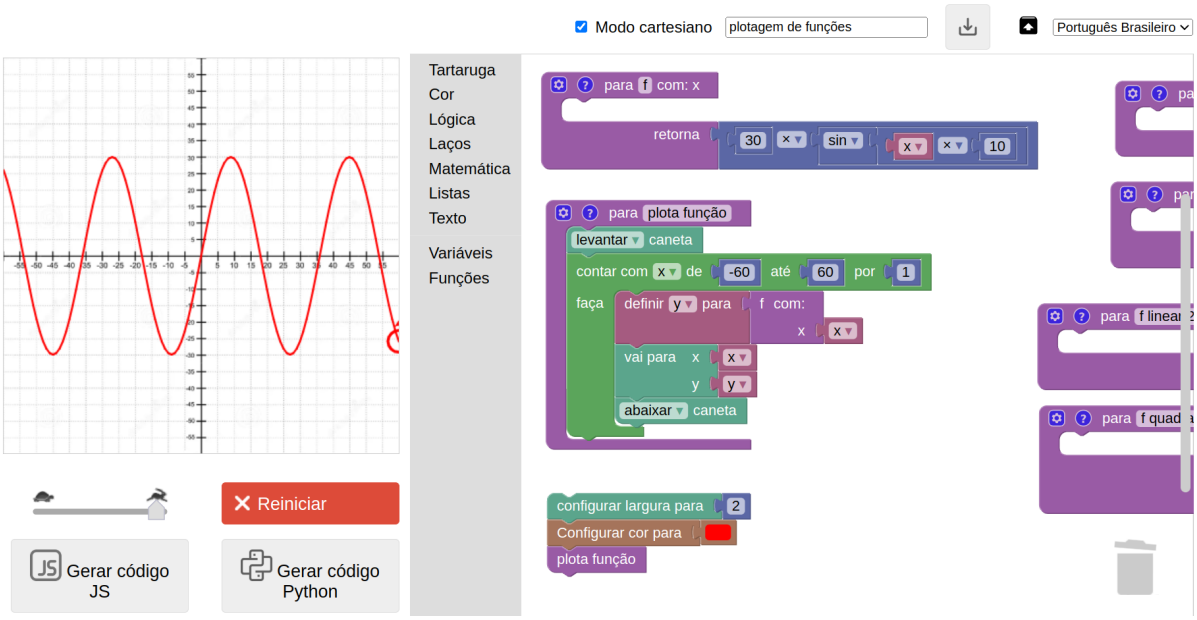
A função f acima é descrita matematicamente como $f(x) = x + 20$, que é uma função de primeiro grau, assim como $f(x) = x$.



A função f acima é descrita matematicamente como $f(x) = 2x$, que também é uma função de primeiro grau, porém com diferente inclinação.



A função f acima é descrita matematicamente como $f(x) = 0{,}1x^2 - 45$, que é uma função de segundo grau, portanto plotada como uma parábola.



A função senoidal acima seria descrita matematicamente como segue:

$$f(x) = 30 \times sen(10x).$$

Os valores constantes multiplicados pelo *seno* (sin) e pelo x foram ajustados para se adequar à escala de nosso plano cartesiano. Fique à vontade para alterar esses valores para criar diferentes funções senoidais.

Fique à vontade para testar também outras funções matemáticas. Algumas outras sugestões: $f(x) = 5$ (função constante); $f(x) = 0{,}001 \times x^3$ (função de terceiro grau).

Parabéns Professor(a)!! Se você seguiu esse plano de aula, temos esperança de que foi bastante proveitoso para os alunos. Conte-nos como foi!!

Indo além: estrelas, espirais e fractais

É possível criar uma grande variedade de desenhos com o Turtle, mas há três tipos de figuras que valem a pena mencionar caso você queira se aprofundar e, talvez, levar para seus alunos. Um desses tipos são as **estrelas**. A lógica do desenho de uma estrela não é muito diferente da lógica do desenho de polígonos regulares: repete-se uma certa quantidade de vezes um movimento para frente e um giro em determinado ângulo.

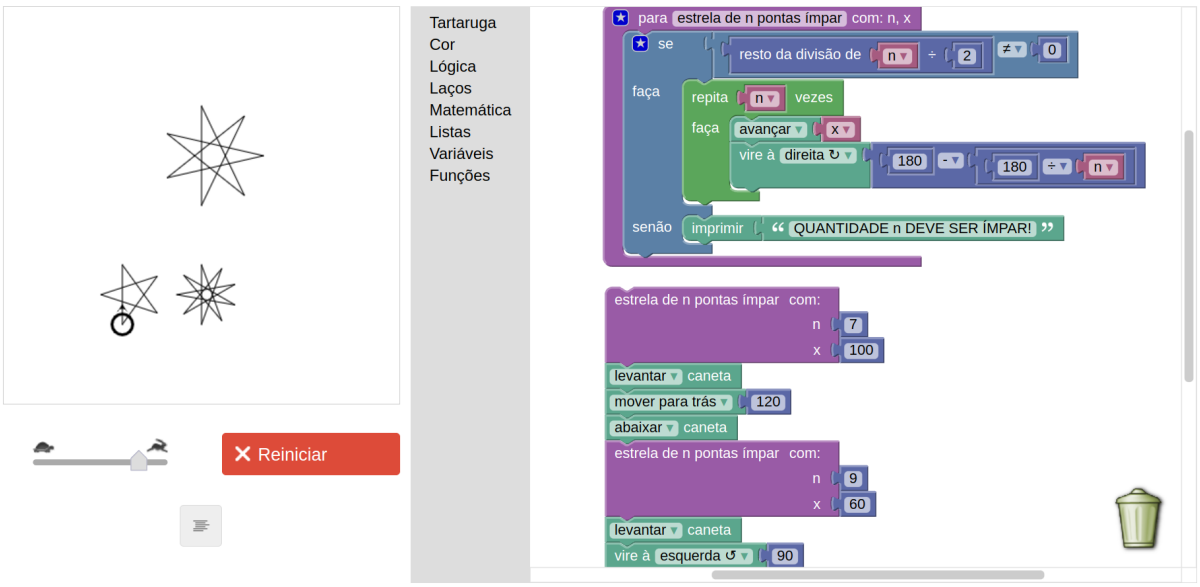
Outro tipo de figura interessante são as **espirais**, que também consistem em repetições de movimentos e giros, mas alterando-se o ângulo de giro e/ou a distância percorrida a cada repetição. É possível desenhar espirais baseadas nos mais diversos polígonos regulares, incluindo espirais circulares.

Por fim, o tipo mais intrigante de figuras são os **fractais**, que consistem em figuras geradas recursivamente, podendo inclusive ser geradas por tempo indeterminado. Muitos fractais simulam padrões presentes na natureza, como o crescimento de plantas e a formação de flocos de neve.

Estrelas

Há dois tipos de estrelas no que diz respeito à lógica de seu desenho: com quantidade ímpar de pontas e com quantidade par de pontas. Vejamos primeiramente como podemos fazer uma estrela com quantidade de pontas ímpar, começando com a estrela de 5 pontas.

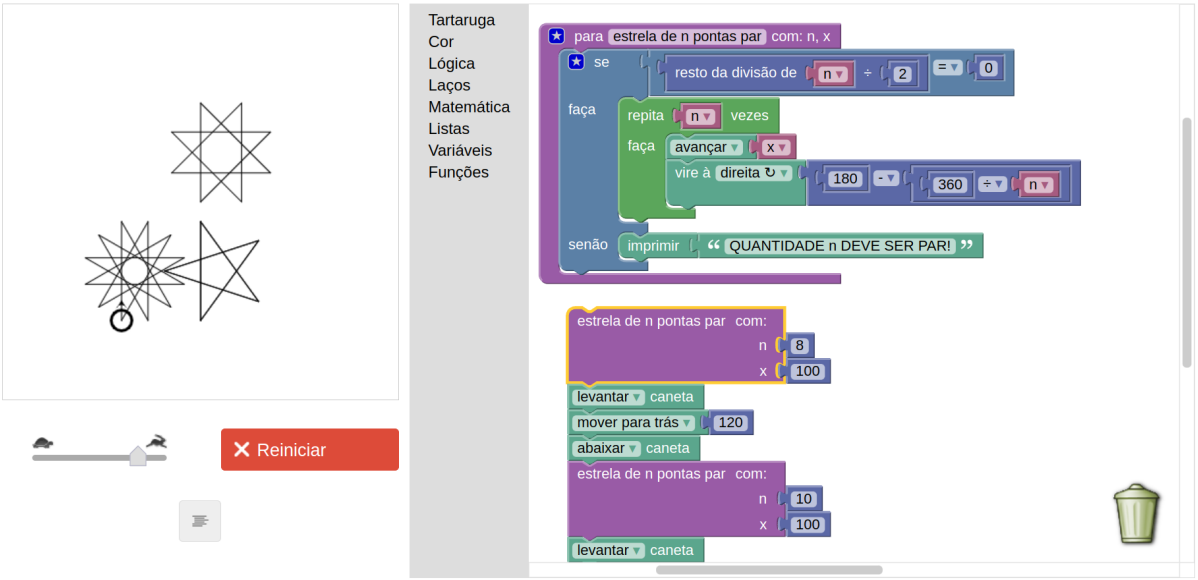
Perceba que a construção lógica é idêntica ao desenho de um pentágono, mas com ângulo de 144°. Mas, de onde tiramos esse ângulo? Basta sabermos que o *ângulo interno* de cada ponta da estrela de 5 pontas tem 36º (que deriva do cálculo *180°/5*). Logo, seu *ângulo suplementar* é de *180° - 36° = 144°* (que é o ângulo que a tartaruga precisa virar para formar as pontas). A partir disso, podemos derivar uma fórmula geral para calcularmos o ângulo suplementar de uma estrela de *n* pontas: *ângulo = 180º - (180° / n)*. Assim podemos testar a fórmula para estrelas com diferentes quantidades ímpares de pontas, e chegamos à solução para qualquer número ímpar maior ou igual a 5.



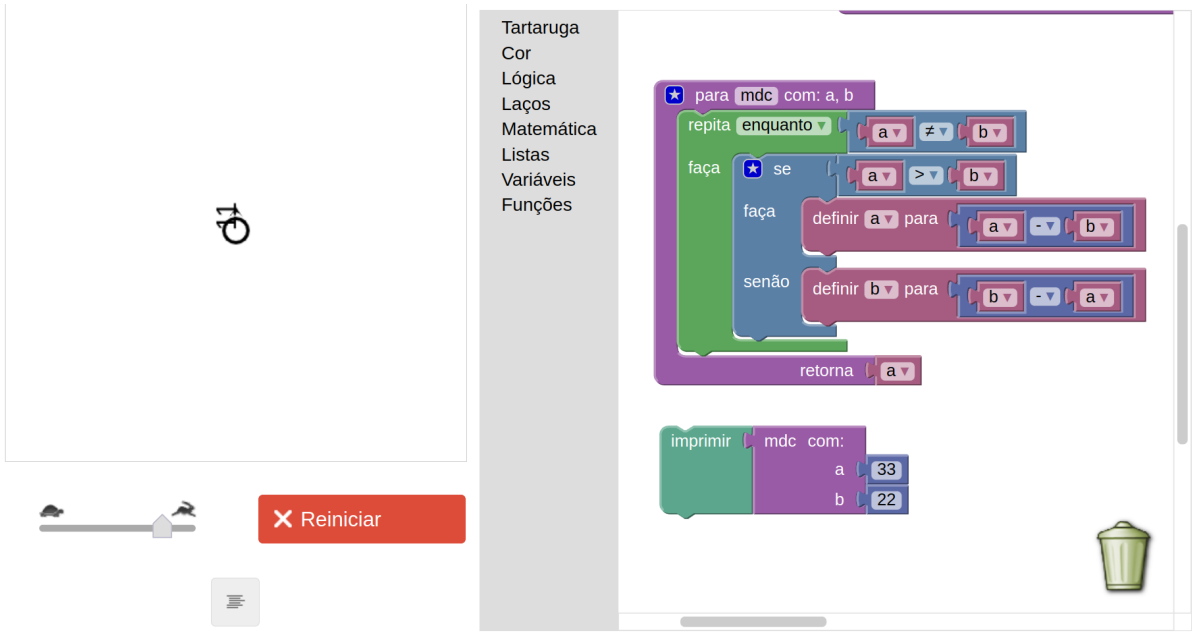
> **Obs:** Perceba que foi feito um comando condicional (se... faça... senão...), disponível no menu Lógica, para verificar se n é de fato um número ímpar. Se não for, a tartaruga escreve: "QUANTIDADE n DEVE SER ÍMPAR!".

Quanto às estrelas com quantidade par de pontas, é um caso um pouco mais complexo. Tome, por exemplo, as estrelas de 6 e 8 pontas. Note que uma forma de desenhar a de 6 pontas é sobrepôr dois triângulos equiláteros rotacionados, e a de 8 pontas pode ser desenhada sobrepondo dois quadrados também rotacionados. No entanto, seguir essa lógica resultaria em estrelas muito "largas" à medida que a quantidade de pontas aumenta.

Alguns casos de estrelas com quantidade par de pontas podem ser desenhadas com uma variante da fórmula para estrelas com pontas ímpares. Basta calcular o ângulo usando *180 - 360/n*. Essa fórmula funciona para a estrela de 8 pontas e de 12 pontas, mas não funciona para de 6 e de 10, por exemplo. Veja abaixo:

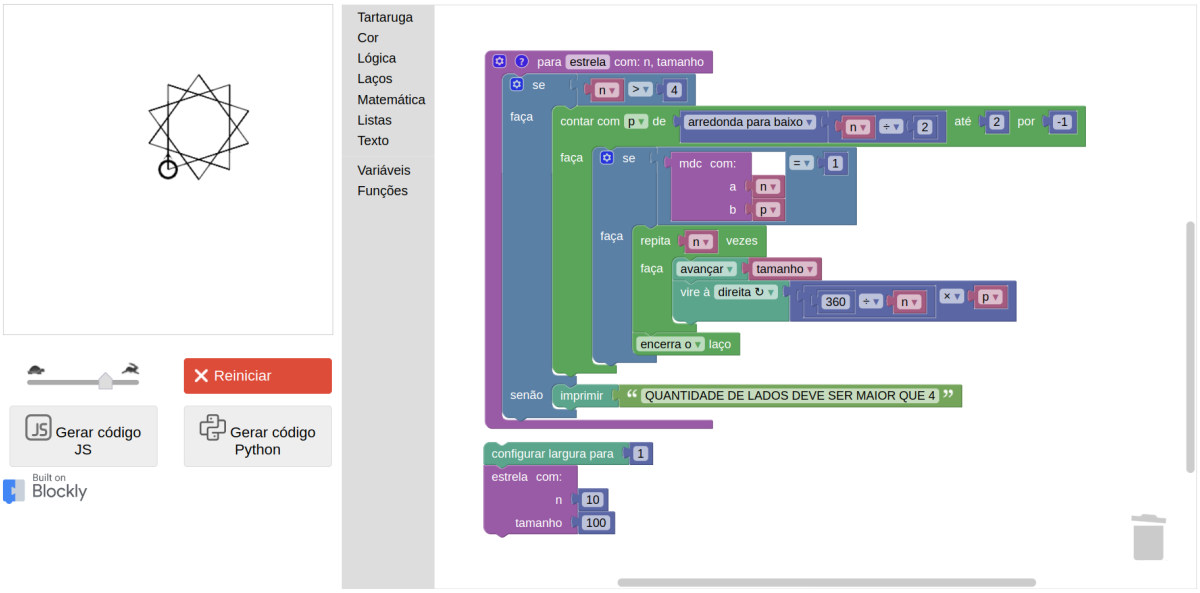


Uma solução mais sofisticada, capaz de desenhar a maioria das estrelas, tanto ímpares como pares, é utilizar o cálculo do Máximo Divisor Comum (MDC) para procurar um número coprimo da quantidade de pontas. Dois números são coprimos entre si se não existe um divisor comum entre eles além do próprio número 1. Uma vez encontrado um coprimo (p) da quantidade de pontas (n) desejada, basta calcular o ângulo como *(360/n) * p*. Portanto, precisaremos primeiramente implementar (codificar) um algoritmo que permita fazer o cálculo do MDC: o algoritmo de Euclides, um dos algoritmos mais antigos, conhecido desde por volta de 300 a.C. O algoritmo de Euclides é bastante simples: ele toma como entrada dois números inteiros (a e b), e, a cada iteração (repetição), subtrai o maior pelo menor e substitui o maior pelo resultado da subtração. Repete-se esse processo até que a=b (o valor de a seja igual o valor de b). Abaixo segue a implementação do algoritmo no Blockly Turtle:

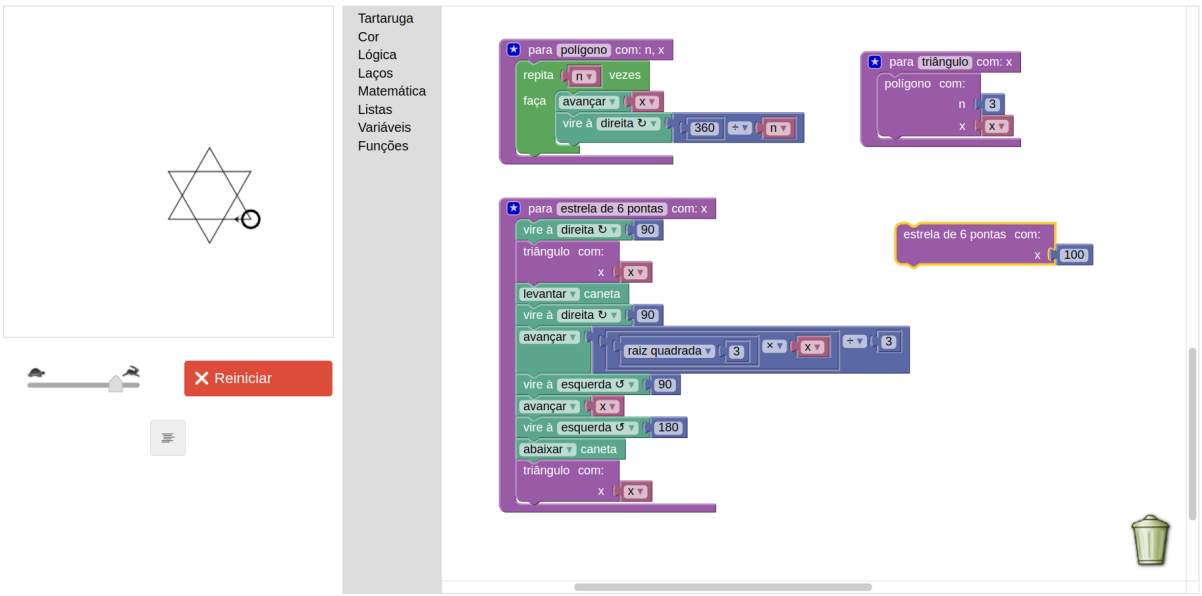


Perceba que a função retorna, ao final, o valor de a (que deverá ser igual a b), que é o resultado do MDC. Quando a função retorna, é possível utilizar o valor de retorno em outros lugares do código, o que é muito útil.

Assim, podemos definir a função estrela da seguinte forma: testa-se cada número entre 2 e *n/2* (quociente da divisão de *n* por 2), de modo a verificar se algum desses número é coprimo de *n*. Para testar todos os números entre 2 e *n*/2, podemos usar o bloco de Laço contar com i de ... até ... por ..., que em outras linguagens é conhecido como o laço do tipo ***"for"*** ou ***"para cada"***. A ideia é inicializar uma variável *p* (chamada de **variável de laço**) com o valor *n/2*, e a cada ciclo do laço (*loop*) decrementar este valor, até chegar ao valor 2. Assim, para cada valor de *p*, calcula-se o MDC de *n* e *p* e verifica-se se o MDC é igual a 1. Se for, significa que *n* e *p* são coprimos entre si, e assim podemos usá-lo na fórmula apresentada. Veja abaixo como fica:



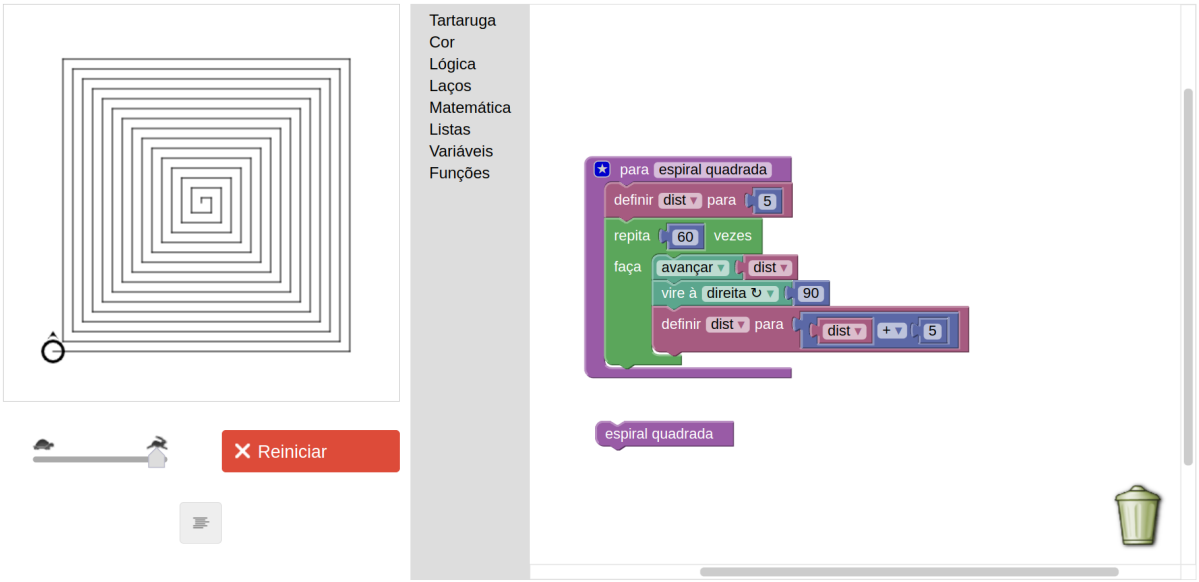
Note que este algoritmo funciona para *n=10,* o que antes não era possível. Porém, ainda temos uma exceção para a qual esse algoritmo também não funciona: quando *n=6*. Esse é um exemplo de caso específico que temos que tratar separadamente. Uma forma de se desenhar uma estrela de 6 pontas é desenhando dois triângulos equiláteros (por exemplo, usando a função *"polígono"* que definimos anteriormente), mas virados em ângulos diferentes. Um exemplo de como isso pode ser feito:



Espirais

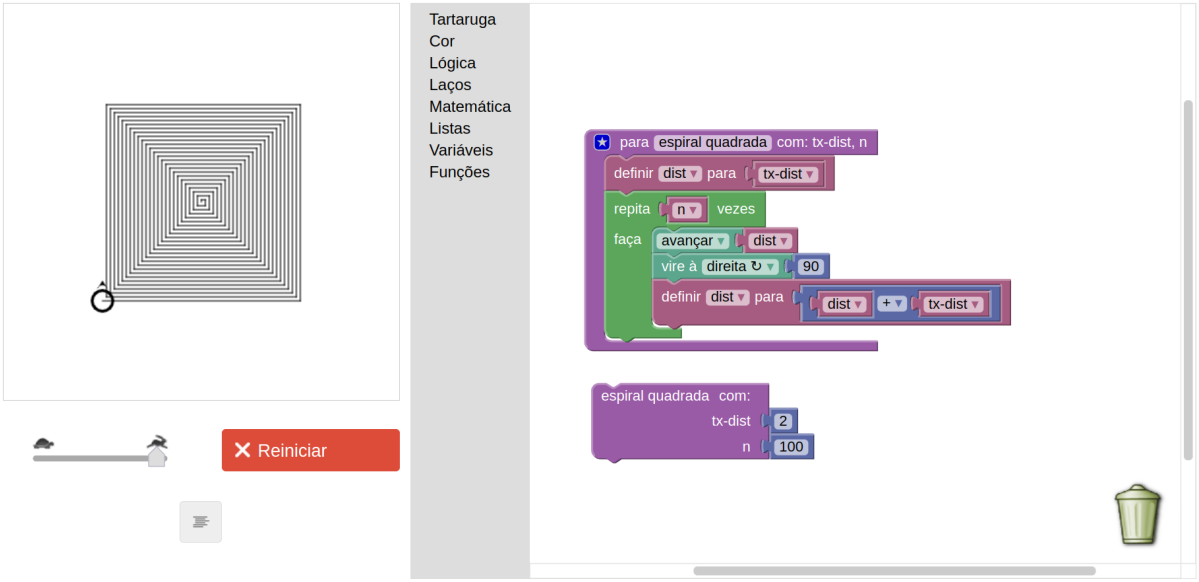
Para construir espirais, é necessário fazer um laço de repetição similar ao que já fizemos para polígonos. No entanto, será necessária a criação de uma **variável de laço** que altere, a cada repetição, a distância a percorrer e, em alguns casos, o ângulo de giro da tartaruga.

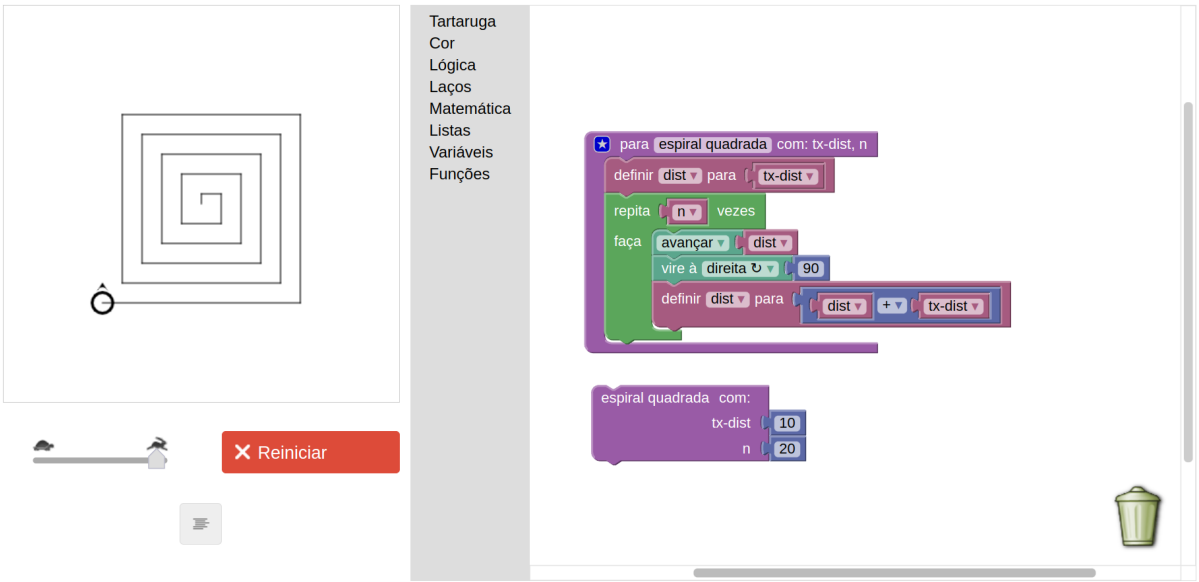
Veja a seguir o exemplo de uma espiral em formato quadriculado:



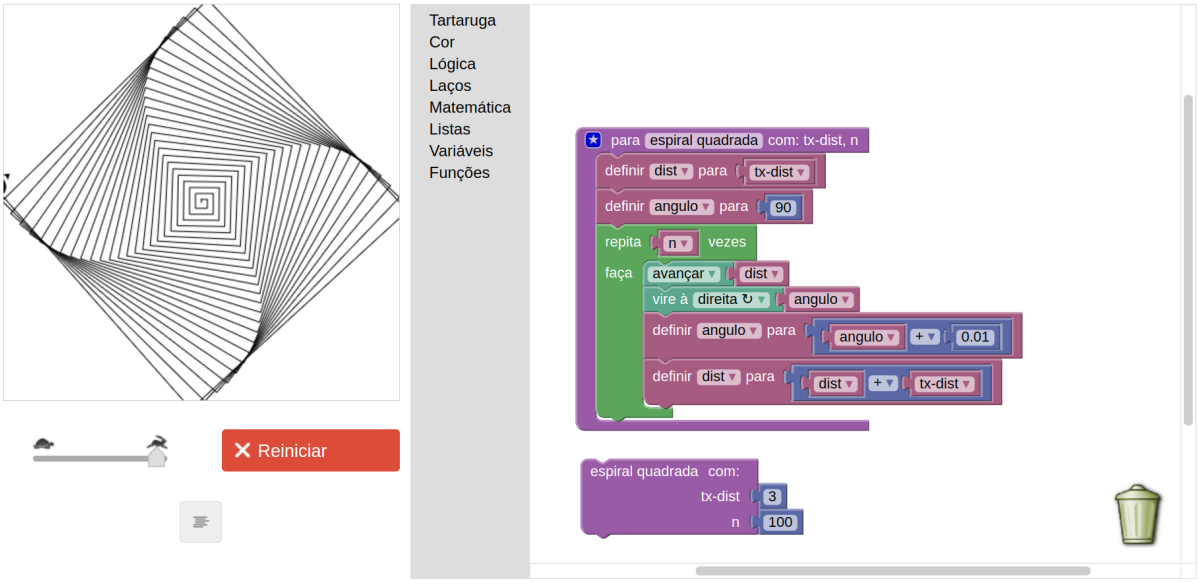
Note que, no início da função, definiu-se uma variável chamada dist. Essa variável é inicializada com o valor 5, e, a cada iteração (dentro do repita), utiliza-se essa variável no comando avançar, e, ao final, incrementa-se em 5 o valor da variável. Isto é, o valor da variável sempre é atualizado a cada iteração somando-se 5 ao seu valor atual. Dessa forma, a cada iteração a tartaruga avança uma distância cada vez maior. Por exemplo, na primeira iteração, ela avançará 5. Na segunda, ela avançará 10. Na terceira, ela avançará 15. E assim por diante, até que a quantidade máxima de iterações definida no repita (60, no caso acima), seja atingida.

É possível também parametrizar essa função para que possamos escolher a "taxa de aumento" e a quantidade de iterações. Essa "taxa de aumento" substituiria o valor 5 no código acima, de modo que, se definirmos um valor menor, a espiral será mais "densa", com espaços menores, e se definirmos um valor maior, será mais "esparsa", com espaços maiores. Veja abaixo essa nova versão e diferentes espirais geradas:

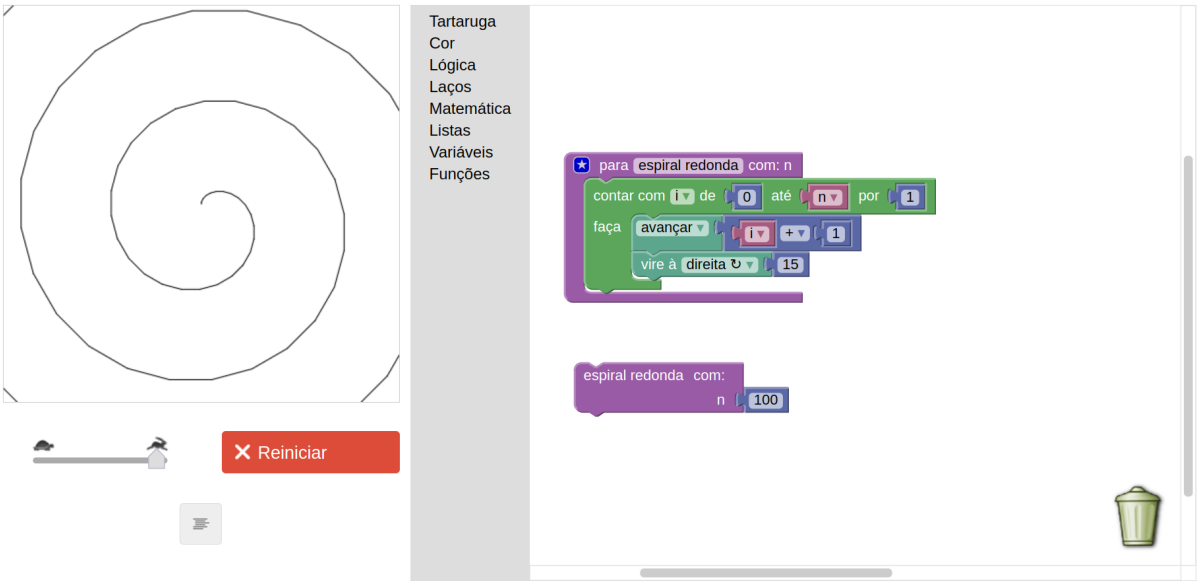




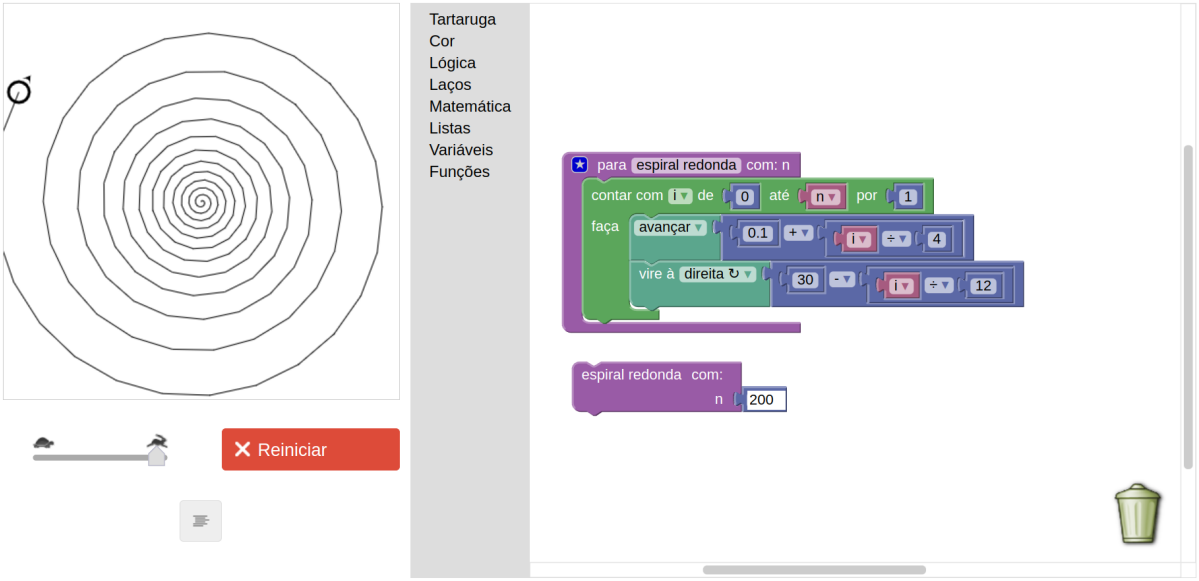
É possível também definir uma variante dessa espiral que faz um efeito de "torção" na espiral. Basta definir mais uma variável angulo, que inicia com o valor 90 e é incrementada a cada repetição por um pequeno valor de ângulo (por exemplo, 0.01). Veja:



É possível também desenhar espirais circulares. A lógica para isso é muito similar à espiral quadriculada. No entanto, fazemos com que a distância percorrida e o ângulo sejam pequenos, e alteramos aos poucos a distância e o ângulo. Uma forma mais flexível de ajustarmos esses valores é utilizando um laço do tipo contar com i de ... até ... por ..., o famoso laço do tipo ***"for"*** ("para cada"). Esse tipo de laço já inclui uma variável de laço (aqui chamada de i) que é incrementada automaticamente a cada iteração. Assim, podemos utilizar esse i como uma forma de calcular uma taxa de crescimento para a distância e o ângulo. Veja abaixo uma primeira tentativa, utilizando a fórmula *i+1* para a distância a se percorrer e 15º fixos de ângulo. Note que, dessa forma, inicialmente a tartaruga percorre a distância de 1 (0+1, pois i=0), e vira 15º. Em seguida, percorre a distância de 2 (1+1, pois i=1), e vira 15º, e assim por diante.



A partir dessa ideia, podemos testar diferentes cálculos de distância e ângulo envolvendo **i**. Por exemplo, podemos somar o valor de **i**, porém dividindo-o por um número positivo maior que 1, para que nossa espiral se torne mais densa. Também podemos incrementar aos poucos o valor do ângulo, para que a espiral se expanda gradualmente. Veja um exemplo, calculando-se a distância usando a fórmula *0.1+(i/4)* para a distância e *30-(i/12)* para o ângulo.

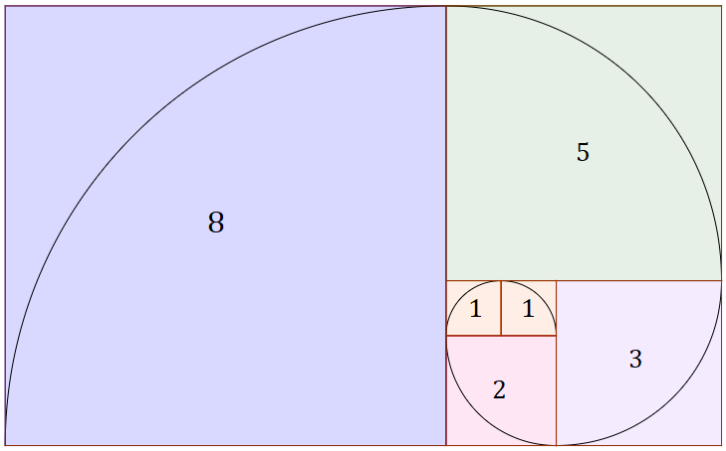


Fique à vontade para testar diferentes fórmulas e veja como ficam as diferentes espirais.

Um tipo muito interessante de espiral que é possível desenhar com o Turtle é a **espiral de Fibonacci**, que é baseada na **sequência de Fibonacci**. Essa conhecida sequência (ou série) consiste em gerar o próximo valor da sequência somando-se os dois valores anteriores. Por exemplo, os 10 primeiros números da sequência de Fibonacci são: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55[10]. Note que a sequência sempre se inicia com "1, 1" e, então, a partir do terceiro elemento, soma-se os dois anteriores. Assim, o terceiro elemento é 2 = 1 + 1, o quarto é 3 = 2 + 1, o quinto é 5 = 3 + 2, e assim por diante. A partir da sequência de Fibonacci, é possível, definindo-se e calculando-se sua função de recorrência, calcular a assim chamada **proporção áurea**. Ou seja, à medida que a sequência de Fibonacci avança em valores cada vez maiores, a proporção entre o último e o penúltimo elemento calculados será de aproximadamente 1,61803 (trata-se de um número irracional, como o *pi*). Note, por exemplo, que se dividirmos 34 (o 9º valor da sequência) por 21 (o 8º), obtermos 1,61904, e se dividirmos 55 (o 10º) por 34 (o 9º), obtemos 1,61764. Quanto mais adiante na sequência, mais essa proporção se aproxima de 1,61803.

A espiral de Fibonacci consiste em desenhar quadrados cujos tamanhos dos lados são os números da sequência, e, dentro de cada quadrado, desenhar um arco de 90° cujo raio é o tamanho do lado do quadrado. Deste modo, concatenando-se esses quadrados de maneira a formar uma espiral, obtemos a espiral de Fibonacci. A figura abaixo mostra um exemplo.

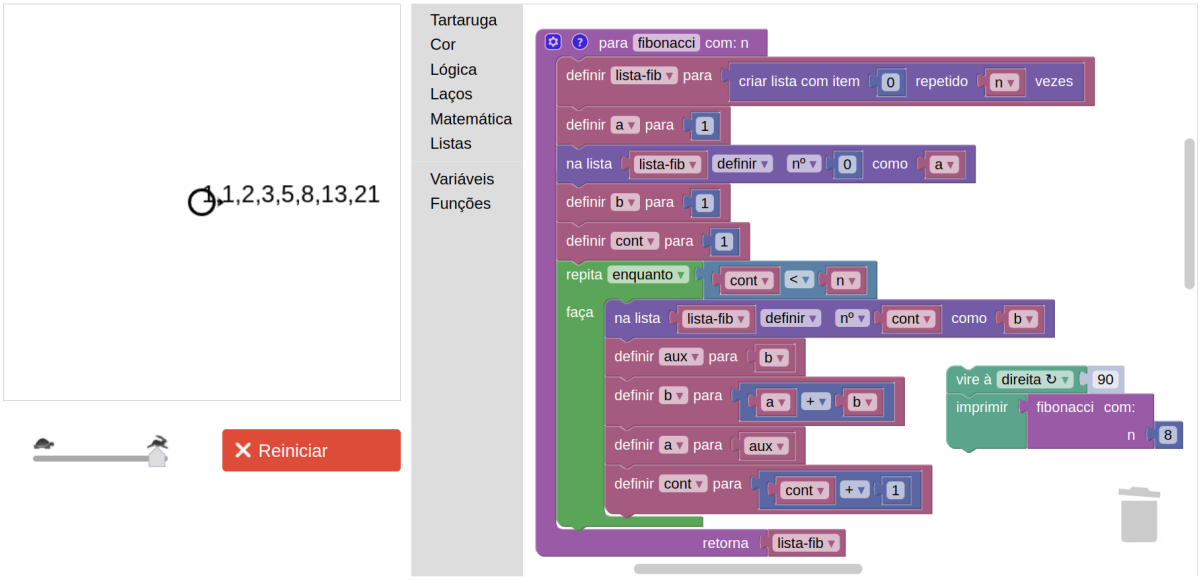
[10] Alguns autores iniciam a sequência com o número 0, seguido por 1. Aqui optou-se por iniciar a sequência com 1, seguido por 1, para facilitar o entendimento da espiral de Fibonacci mais adiante.



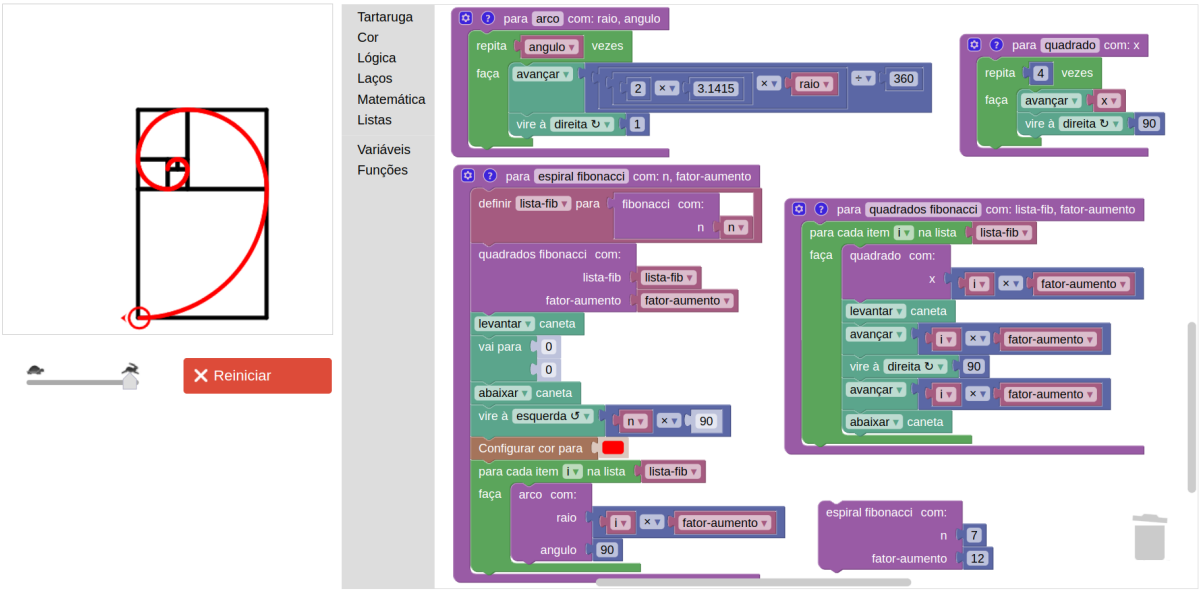
Fonte: https://www.infoescola.com/matematica/sequencia-de-fibonacci/

A sequência de Fibonacci, assim como sua proporção áurea e sua espiral, são encontradas em abundância na natureza, como na concha de caracóis, na cauda do cavalo-marinho, no formato de galáxias espirais, em diferentes proporções do corpo humano, no formato da molécula de DNA, etc. Por esse motivo, alguns consideram a proporção áurea como uma assinatura de Deus na Criação, e está frequentemente associada ao que é belo aos olhos humanos. Por exemplo, diversas obras de arte, obras arquitetônicas, e até composições de música clássica, utilizam essa proporção.

Para desenharmos a espiral, primeiramente é necessário definir uma função que calcula a sequência de Fibonacci. Para isso, utilizam-se os blocos do menu "Listas" para se criar uma lista sequencial com os primeiros `n` números da sequência. Esse `n` é, portanto, a variável de entrada (parâmetro) dessa função. O funcionamento da função consiste em definir, além da lista resultante, duas variáveis `a` e `b`, que, respectivamente, armazenam os valores dos dois últimos valores da sequência calculados até o momento. Ambas as variáveis são inicializadas com o valor 1, e então inicia-se um laço `repita enquanto...`, que realiza a atualização dessas duas variáveis com base na soma de seus valores atuais, e uma variável de laço `cont` que faz a contagem de quantos números da sequência já foram gerados. O laço para de repetir quando `cont` = `n`, o que significa que a quantidade de números especificada na variável `n` terminou de ser gerada. Dentro do laço, a variável `a` recebe o valor de `b`, e `b` recebe o valor de `a`+`b`. Além disso, `b` (o último elemento calculado), é inserido ao final da lista. Dessa forma, ao final da execução desse laço, teremos a lista completa com os primeiros `n` valores. A imagem a seguir mostra a definição dessa função, assim como uma chamada passando-se `n`=*8*. Perceba que foi possível acoplar a chamada da função (à direita de sua definição) ao comando *imprimir*, visto que a chamada da função retorna uma lista que pode ser então escrita na tela.



Uma vez gerada a sequência, podemos utilizá-la para o desenho dos quadrados. Para isso, utilizaremos a mesma função quadrado definida anteriormente (ver seção **Desenhando um Quadrado**). Criaremos também uma função chamada de arco, que recebe um ângulo e um raio, e desenha um arco. Essa função é muito similar à função círculo definida anteriormente (ver seção **De Polígonos para Círculos**), com a diferença que um círculo é um arco de 360°, enquanto a função arco permite desenhar arcos com diferentes ângulos (por exemplo, 90°, que é o ângulo que precisamos para desenhar a espiral de Fibonacci). A partir disso, cria-se duas funções: uma que desenha os quadrados e outra que desenha a espiral. Um exemplo das definições dessas funções é dado abaixo.



Fractais

Uma última classe de figuras que abordaremos neste material são os fractais. Um fractal (do latim *fractu*: fração, quebrado) é uma figura da geometria não clássica que consiste em um padrão que se repete, tal que suas partes menores repetem os traços (a aparência) do todo

completo. De fato, espirais, como a de Fibonacci, também são um tipo de fractal, pois possuem a propriedade de poderem ser desenhadas por um período de tempo indeterminado, seguindo uma regra que define um padrão que se repete. Nesta seção, veremos alguns outros exemplos de fractais e duas formas de desenhá-los: da forma **recursiva** e utilizando **Sistemas-L**, ou Sistemas de Lindenmayer.

A primeira forma introduz um conceito importante na Ciência da Computação e na Matemática: a **recursividade**. Uma função é dita recursiva quando sua definição depende da própria definição da função. Tomemos o problema do cálculo do **fatorial** de um número. Por exemplo:

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

$$3! = 3 \times 2 \times 1$$

$$2! = 2 \times 1$$

$$1! = 1$$

Note que a definição de cada um desses exemplos pode ser feita da seguinte forma:

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

$$1! = 1$$

Assim, percebemos que a fórmula que define a função matemática do fatorial é dada por:

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

Podemos definir isso como uma função da forma convencional, chamando a função de *fat*:

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

Note que a definição da função deve ser feita considerando mais de um caso. O primeiro é chamado de **passo recursivo**, que envolve a chamada (aplicação) da própria função. Perceba que, para calcularmos o fatorial de 4 (4!), precisamos calcular o fatorial de 3 (3!); para calcular o fatorial de 3 (3!), precisamos calcular o fatorial de 2 (2!); e para calcular o fatorial de 2 (2!), precisamos calcular o fatorial de 1 (1!). Assim, em algum momento, a função será chamada para um caso que é conhecido como **caso base**, que é aquele caso que não envolve a necessidade de se aplicar a função recursivamente, geralmente possuindo uma solução (resposta) imediata. É o caso do 1!, que sempre será igual a 1. No momento em que a cadeia de chamadas recursivas à função alcança o caso base, cada chamada é retornada para a anterior que a chamou. Veja um exemplo de um teste de mesa, ou rastreio, da chamada recursiva *fat(4)*:

$$fat(4)$$

$$\rightarrow 4 \times fat(3)$$

$$\rightarrow 3 \times fat(2)$$

$$\rightarrow 2 \times fat(1)$$

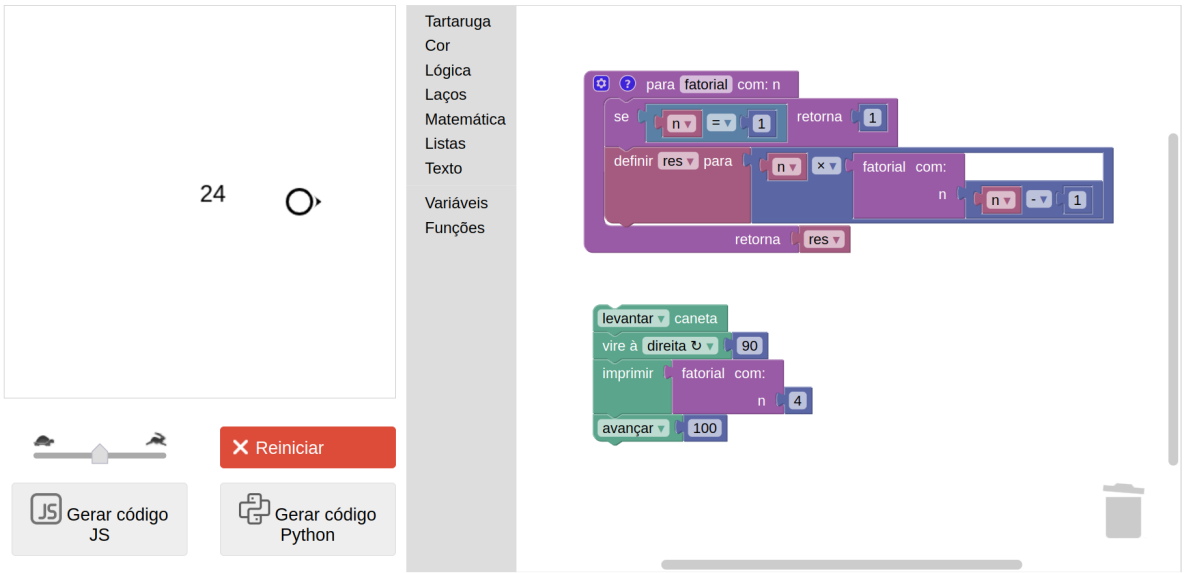
$$2 \times 1 = 2$$

$$3 \times 2 = 6 \leftarrow$$

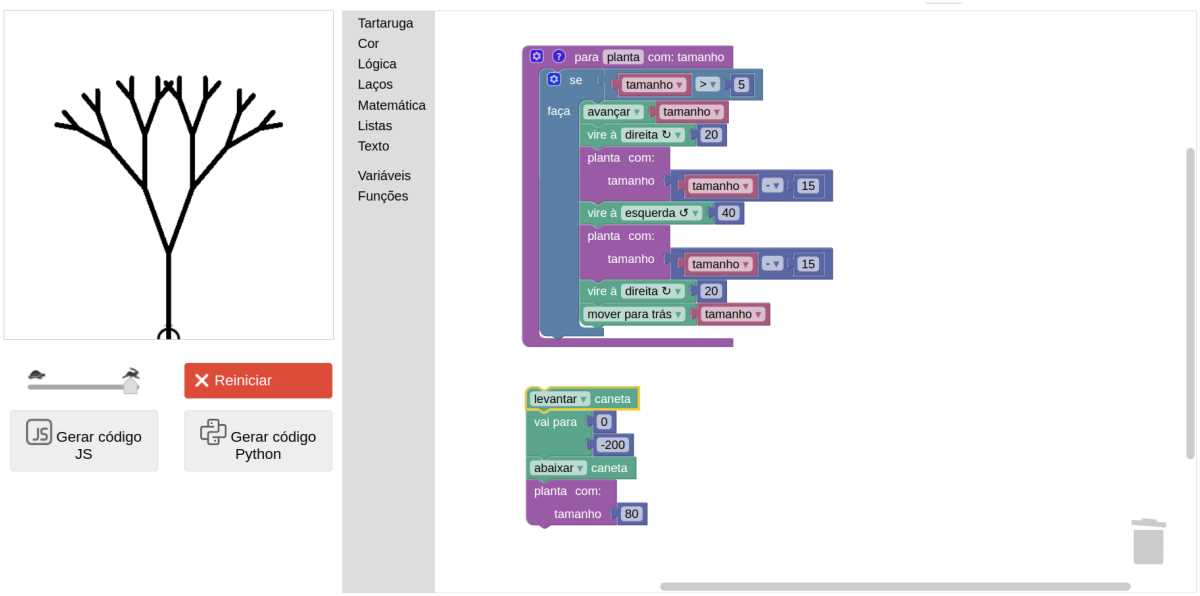
$$4 \times 6 = 24 \leftarrow$$

$$24 \leftarrow$$

Veja abaixo um exemplo de implementação da função *fat*, assim como uma chamada para $n = 4$. Perceba que a função foi definida como uma função que retorna valor, e que o bloco se ... retorna ... se encontra no menu Funções, e não no menu Lógica. Note também que, caso n não seja igual a 1, uma variável chamada res é definida para receber o resultado do cálculo $n \times fat(n-1)$, e que então o valor dessa variável é retornado pela função.



Tendo compreendido a ideia de funções recursivas, tomemos um exemplo de fractal em forma de planta. A ideia é definir uma função chamada planta, que recebe por parâmetro o tamanho do ramo de uma planta, e recursivamente chama a própria função, porém com ramos cada vez menores. Quando o tamanho do ramo alcançar um valor pequeno (e.g. 5), a função para de ser chamada recursivamente. Ou seja, o **caso base** dessa função será quando o *tamanho* do ramo for menor ou igual a 5. Antes e depois de cada chamada recursiva, são desenhadas duas linhas em forma de V, representando um raminho. Veja como fica no Blockly Turtle:



Perceba na definição da função planta como a própria função planta é chamada, no entanto, passando tamanho - *15* por parâmetro. Assim, as chamadas recursivas vão se repetindo até que se alcança um tamanho menor que 5, e então a função termina sua execução.

Embora o uso de recursão para desenhar fractais seja muito interessante, abordaremos agora uma forma alternativa, que é o uso de Sistemas-L. O interessante dessa abordagem é que é possível facilmente adaptar nosso código para desenhar diferentes fractais, como veremos adiante.

> **Atenção:** a partir daqui, é necessário o uso do Blockly Turtle somente por meio do site https://helioh2.github.io/turtle/

Um **Sistema-L**, ou Sistema de Lindenmayer, é um sistema de reescrita paralela por meio de um tipo de gramática formal. Define-se um alfabeto de símbolos a partir dos quais são criadas *strings* (sequências de caracteres, que aqui chamaremos também de *texto*). Caracteres são formas tipográficas como as que encontramos no teclado do computador, tais como letras, números, pontos e sinais, e as *strings* são simplesmente sequências de caracteres, tais como as que usamos para formar palavras e textos. Nos Sistemas-L, essas *strings* são geradas automaticamente por meio de uma **gramática** (também chamada de **conjunto de regras de produção***)*. Dessa forma, inicia-se com uma *string* inicial, também chamada de **axioma**, e a partir de repetidas aplicações das regras da gramática, gera-se uma *string* maior.

Para começar, vamos ver um exemplo simples de conjunto de regras:

A	Axioma
A → B	Regra 1: Mude A para B
B → AB	Regra 2: Mude B para AB

A execução do Sistema-L consiste em iniciar com o axioma e iterativamente aplicar as regras de produção. Assim, sempre que um caractere “A” é encontrado, ele é substituído por “B”, e sempre que um caractere “B” é encontrado, é substituído por “AB”. Supondo que executemos 7 iterações, teremos a seguinte sequência de *strings* geradas:

A (inicial)
B
AB
BAB
ABBAB
BABABBAB
ABBABBABABBAB
BABABBABABBABBABABBAB (final)

Esse Sistema-L de exemplo não serve para muita coisa. Portanto, vejamos um Sistema-L que pode ser utilizado para construir o famoso fractal conhecido como **Triângulo de Sierpinski**:

F−F−F	Axioma
F → F−G+F+G−F	Regra 1: Mude F para F−G+F+G−F
G → GG	Regra 2: Mude G to GG

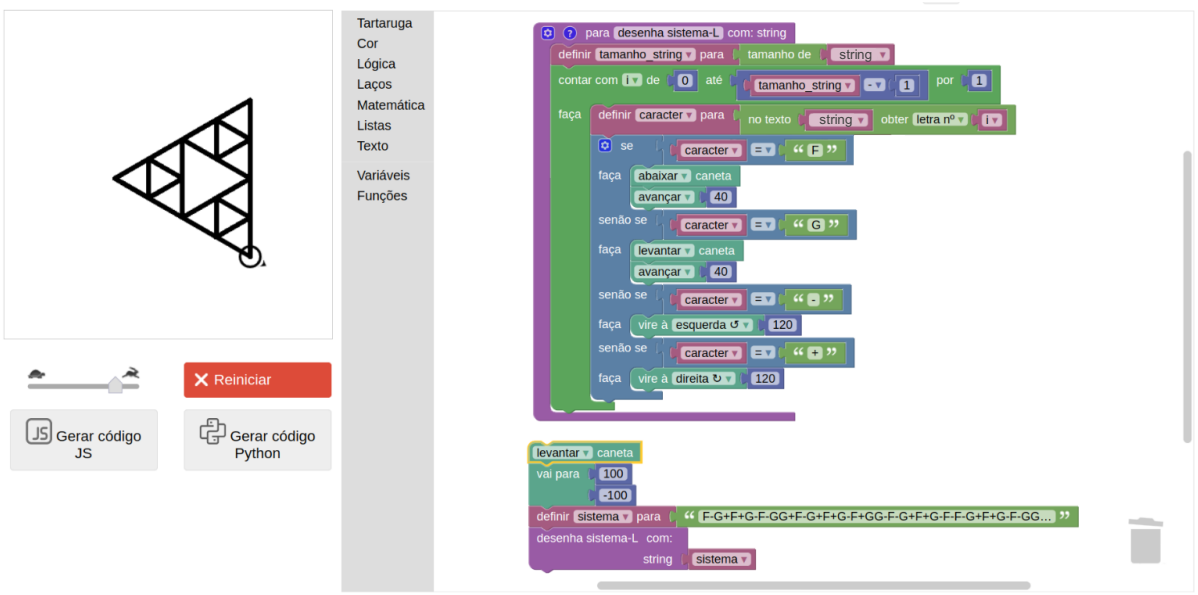
Um exemplo de geração com 2 iterações seria:

F−F−F (inicial)

F−G+F+G−F−F−G+F+G−F−F−G+F+G−F

F−G+F+G−F−GG+F−G+F+G−F+GG−F−G+F+G−F−F−G+F+G−F−GG+F−G+F+G−F+GG
−F−G+F+G−F−F−G+F+G−F−GG+F−G+F+G−F+GG−F−G+F+G−F (final)

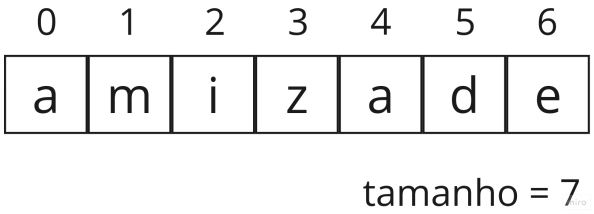
Ok, mas você deve estar pensando: "O que fazemos com esse monte de letrinhas?" Imagine que cada uma dessas letrinhas/caracteres sejam instruções que podemos dar à nossa tartaruga. Por exemplo, quando encontramos um "F", então queremos nos mover 40 passos à frente com a caneta abaixada. Quando encontramos um "G", queremos nos mover 40 passos à frente com a caneta levantada. Quando encontramos um "−" queremos virar à esquerda em um ângulo de 120°. E, por fim, quando encontramos um "+" queremos virar à direita em um ângulo de 120º. Se lermos cada caractere da *string* final, na ordem em que aparecem, e realizarmos os comandos conforme mencionamos, teremos o seguinte desenho:



No exemplo acima, foi criada uma função chamada `desenha sistema-L`, que recebe por parâmetro uma `string`. Perceba que na parte de baixo, criamos uma variável chamada `sistema` que contém a *string* final que escrevemos acima. Em seguida, chamamos a função `desenha sistema-L` passando esse `sistema`. Dentro da função, é feito o seguinte: cria-se uma variável `tamanho_string` que recebe o valor do tamanho da `string`. O comando que recupera o tamanho de uma *string*, assim como todos os demais blocos que realizam operações em *strings*, encontra-se no menu **Texto**. Em seguida, cria-se um laço (*loop*) do tipo `contar com...`, começando do índice 0 até `tamanho_string` - *1*. Isto serve para pegar os índices (ou posições) de cada caractere dentro da *string*.

Nas linguagens de programação em geral, o índice 0 representa a 1ª posição de uma sequência. Portanto, se quisermos pegar o primeiro caractere da *string* (no caso em questão, seria a letra "F"), então pegamos o caractere que está no índice 0. De modo similar, se quisermos pegar o segundo caractere (no nosso caso, o tracinho "−"), devemos pegar o caractere no índice 1, e assim por diante. Supondo que nossa *string* tenha 29 caracteres (como é o caso da *string* gerada após a 1º iteração), então o último caractere será o que se encontra no índice 28. Ora, mas por quê? Porque começamos a contar do índice 0. Veja, por exemplo, a palavra "amor". O índice 0 pegaria o caractere "a", o índice 1 o caractere "m", o índice 2 o caractere "o" e o índice 3 o

caractere “r”. Note que o **tamanho** (isto é, a quantidade de caracteres) da palavra “amor” é 4. Porém, o último caractere da palavra é acessado pelo índice 3, que é exatamente o tamanho da palavra subtraído por um. Veja abaixo uma ilustração de uma *string* e seus índices:

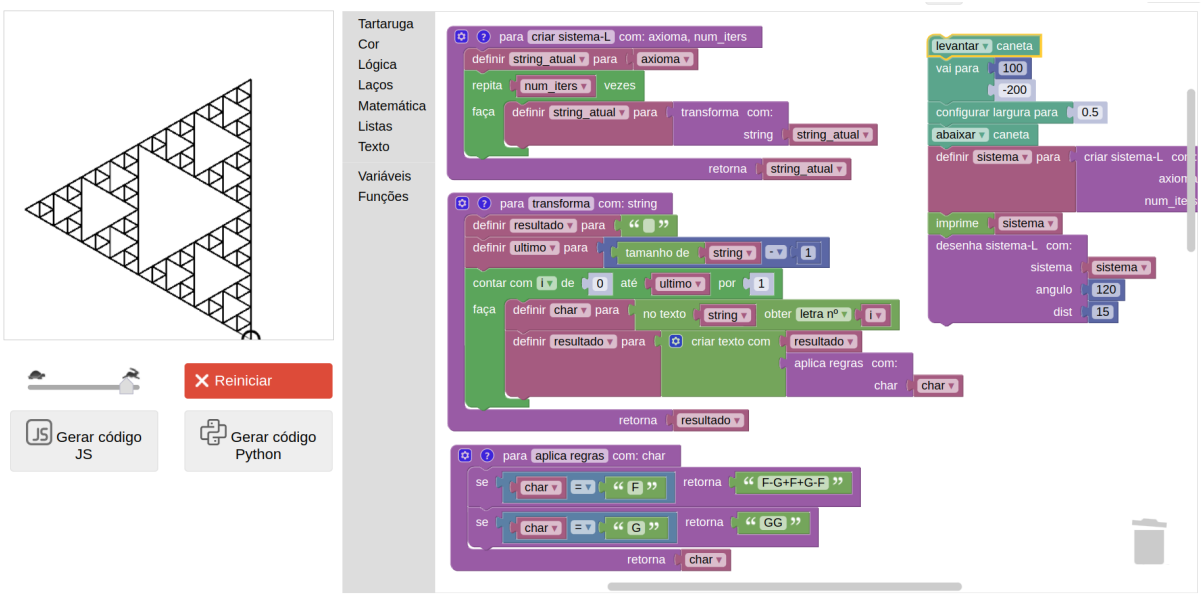


Dentro do laço, cada caractere é acessado pelo seu índice i, por meio da instrução no texto... obter nº... (instrução esta encontrada no menu Texto), e então inicia-se um bloco de condicionais do tipo **se**... faça... senão **se**... faça.... Na frente de cada *“se”*, devemos colocar uma condição. Assim, no primeiro *“se”*, pergunta-se se o caractere é igual a “F”. Se for o caso, abaixa-se a caneta e move-se 40 pixels para frente. O segundo *“se”* pergunta se o caractere é igual a “G”. Se for o caso, levanta-se a caneta e move-se 40 pixels para frente. O terceiro *“se”* pergunta se o caractere é o “−”. Se for o caso, vira-se à esquerda por 120°. E, por fim, o último *“se”* pergunta se o caractere é igual a “+”, em cujo caso vira-se à direita por 120°.

Assim, cada caractere na *string* é lido e interpretado, e uma ação é realizada como resultado. Pense na *string* como uma fita contendo uma sequência de letras, sendo que cada letra é uma instrução. Se for “F”, você anda para frente riscando o chão. Se for “G”, você anda para frente sem riscar o chão. Se for “−”, você vira à esquerda por 120°. E se for “+”, você vira à direita por 120°.

Apenas a título de curiosidade, note que o desenho do triângulo de Sierpinski consiste na repetição de um padrão de triângulos que lembra muito o desenho da *Triforce* dos jogos da franquia *The Legend of Zelda*. Você já jogou algum jogo dessa franquia? Se não, recomendo!

Enfim, vimos como um Sistema-L do tipo “Triângulo de Sierpinski” é capaz de desenhar tal triângulo, mas como podemos fazer para gerar essas *strings* de forma automática? Devemos elaborar um algoritmo que faça n iterações (repita n vezes) a substituição de cada caractere da *string* conforme as regras de produção definidas na gramática. Veja abaixo uma forma organizada de fazer isso:

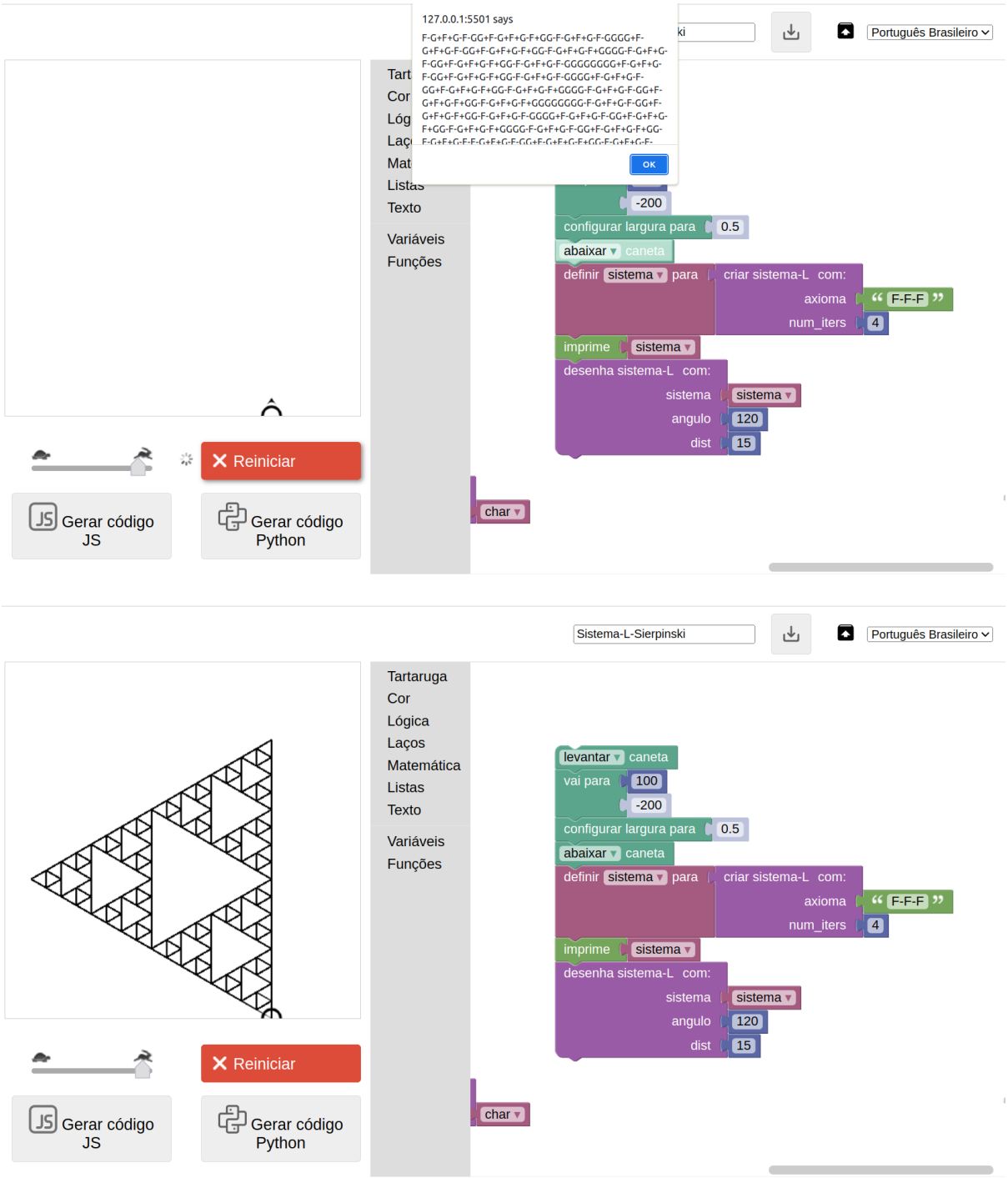


Note que criamos 3 novas funções: criar sistema-L, transforma e aplica regras. A primeira função é a que é chamada para iniciar o processo de construção da *string*. Ela recebe dois parâmetros: o axioma, que é a *string* inicial, e num_iters, que é a quantidade de iterações desejadas. Dentro da função, cria-se uma variável chamada string_atual, que é inicializado com o valor do axioma. Em seguida, inicia-se um laço do tipo repita, para repetir num_iter vezes o seguinte: chama-se a função transforma, passando a string_atual, e atualiza-se a string_atual com o resultado retornado por essa função.

A função transforma é responsável por percorrer a string, isto é, passar por cada caractere, aplicando as regras. Portanto, cria-se uma variável resultado que é inicializada com uma *string* vazia (um texto sem nenhum caractere). Em seguida, define-se um laço do tipo contar com... de... até... por..., que cria uma variável de laço i, que começa com o valor 0 (primeira posição da string) e termina com a posição correspondente ao tamanho da string menos 1 (última posição da string). Dentro do laço, captura-se o caractere na posição i, guardando-o em uma variável chamada char e, em seguida, chama-se a função aplicar regras passando o char.

Por fim chegamos à função aplica regras. Ela recebe por parâmetro um caractere (variável char) e, por meio de condicionais (se... retorna...), define as regras de produção.

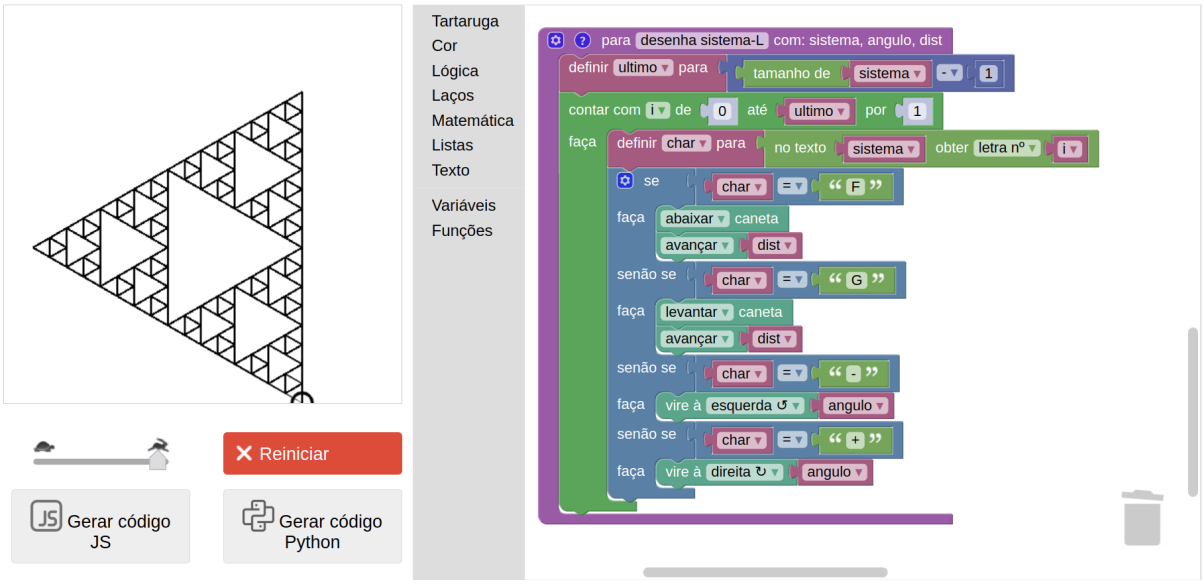
Resumindo, a função criar sistema-L repete várias vezes a função transforma, que, por sua vez, aplica regras para cada caractere da *string*. Ao final, a primeira função retorna como resultado a *string* resultante de se aplicar regras nas várias iterações. As figuras abaixo mostram como podemos imprimir a *string* resultante logo após a chamada a criar-sistema-L, e, em seguida, passar essa *string* para a função desenha sistema-L.



O bloco imprime que usamos aqui é diferente do bloco imprimir do menu Tartaruga, e se encontra no menu Texto. Ele imprime algo em um *pop-up* que deve ser fechado para que o programa continue sua execução.

Note também que a função desenha sistema-L chamada acima está um pouco diferente da que definimos anteriormente. Para tornar a função mais flexível, definimos nela dois parâmetros adicionais: angulo e dist. Isso porque, para cada diferente padrão de fractal, o ângulo de giro muda. No caso do Triângulo de Sierpinski, o ângulo deve sempre ser 120°, mas outros exemplos, como veremos adiante, requerem outros ângulos. Também passamos a distância percorrida a cada avançar ou mover para trás. Com isso, podemos alterar o tamanho do fractal

simplesmente mudando o valor desse parâmetro. Veja a seguir como ficou nossa função `desenha sistema-L`:



Ok, o código ficou bem grande, com várias funções. Mas existe uma grande vantagem em se fazer dessa forma (i.e., usando Sistemas-L). Diferentes fractais podem ser desenhados simplesmente alterando-se: (1) as regras de produção (função `aplica regras`); (2) qual instrução cada caractere ativa (função `desenha sistema-L`); e (3) o ângulo de giro. Portanto, podemos **reutilizar** a maior parte do código, alterando somente esses pontos, e assim desenhar qualquer fractal. Aqui temos mais um conceito importante em Ciência da Computação, especialmente na área de Engenharia de Software: a **reutilização**, ou **reuso**.

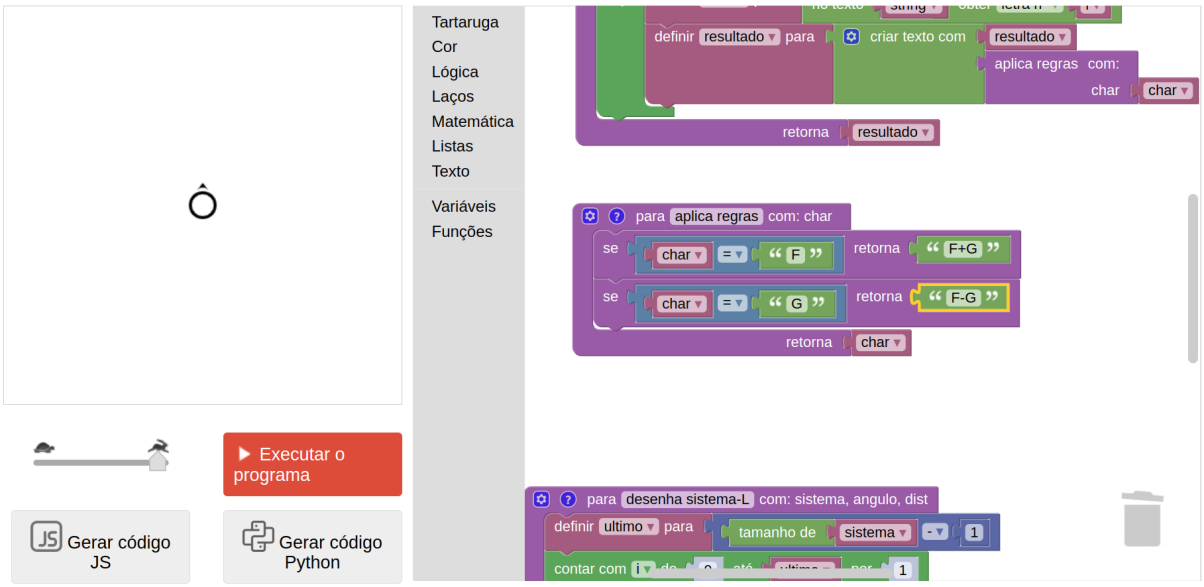
Vamos pegar outro exemplo famoso de fractal, a chamada "Curva de Dragão". Ela é definida como segue:

F	Axioma
F → F+G	Regra 1: Mude F para F+G
G → F-G	Regra 2: Mude G para F-G

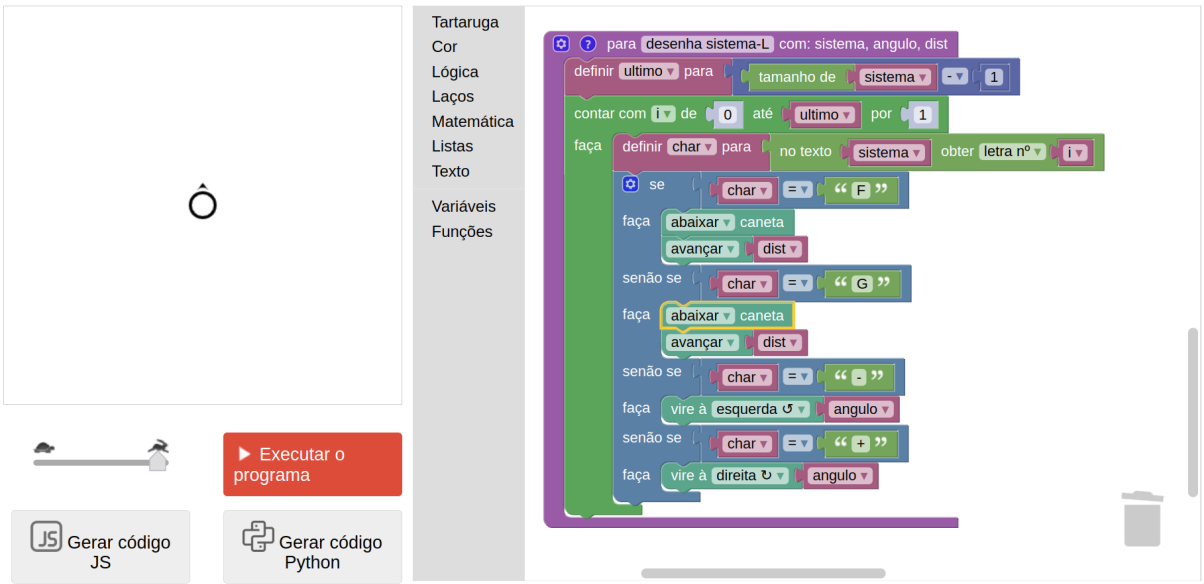
O ângulo de giro deve ser de 90º. Outra diferença aqui é que tanto "F" quanto "G" implicam que se deve avançar desenhando (isto é, com a caneta abaixada). Pronto! Temos todas as informações que precisamos para alterar nosso exemplo anterior para que agora desenhe a Curva de Dragão.

Mas, espere só um pouquinho! Antes de modificarmos nosso código, convém salvar o código atual para que possamos restaurar o código para o Triângulo de Sierpinski quando desejarmos. Para isso, na parte superior da página, basta colocar um nome para o projeto atual (por exemplo, "Sistema L Sierpinski"), e em seguida clicar em  . Isto baixará um arquivo XML que pode ser restaurado mais tarde por meio do botão  .

Uma vez feito isso, vamos alterar a função aplica regras com as novas regras acima.



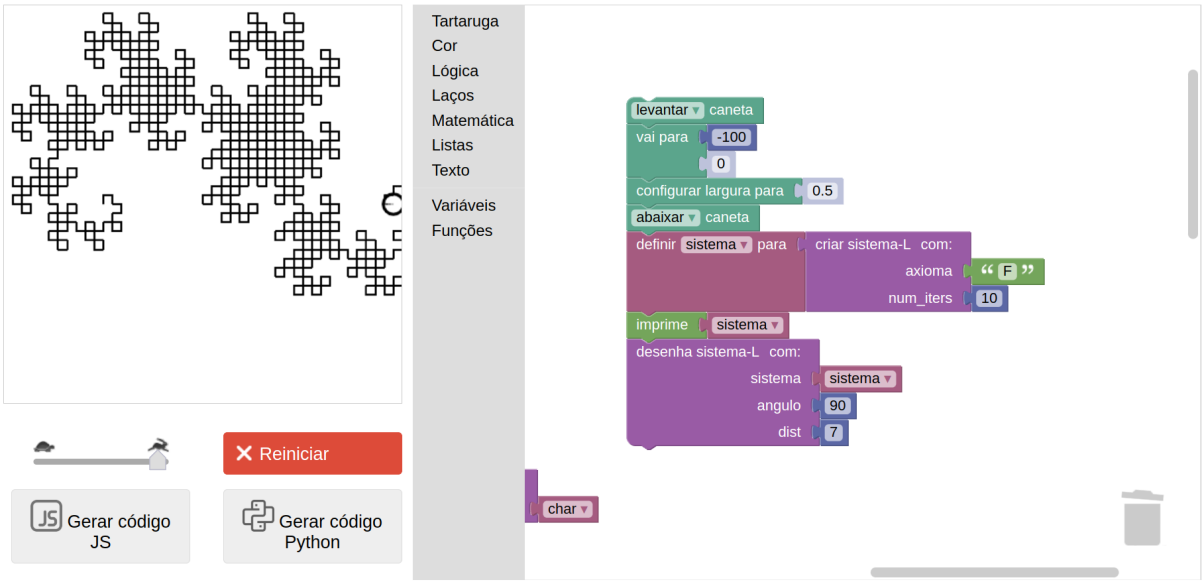
Em seguida, alteramos a função desenha sistema-L para que, quando encontrarmos o caractere "G", ande para frente com a caneta abaixada.



Obs: é possível, e aconselhável, definir um único se... para os dois casos, "F" e "G". Para isso, basta utiliza o bloco ... ou ... disponível no menu

Lógica (essa opção se encontra juntamente com o ... e ..., bastando alterar o "e" para "ou" nas opções do próprio bloco), e adicionar, de cada lado do "ou", as duas condições, ficando assim: se char = 'F' ou char = 'G'. Aqui é possível pegar mais um gancho para um assunto que não tratamos com muito detalhe no conteúdo deste livro, que diz respeito aos operadores lógicos, também conhecidos como *operadores booleanos*.

Em seguida, vamos ao programa principal (o conjunto de blocos que faz as chamadas iniciais), e alteramos o axioma passado para a função criar sistema-L para "F", e alteramos também o angulo na chamada da função desenha sistema-L para 90. Alteramos também dist para um valor menor (7, no exemplo abaixo), e aumentamos o número de iterações (10, no exemplo abaixo), para poder ajustar melhor o tamanho do fractal ao tamanho da tela.



Tente também por conta própria os seguintes Sistemas-L:

- Curva de Koch

F	Axioma
F → F+F−F−F+F	Regra 1: Mude F para F+F−F−F+F

Ângulo: 90º

- Curva de Sierpinski "Ponta de Flecha":

F	Axioma
F → G−F−G	Regra 1: Mude F para G−F−G
G → F+G+F	Regra 2: Mude G to F+G+F

Ângulo: 60°. Tanto “F” quanto “G” implicam em avançar desenhando (caneta abaixada).

Um último exemplo de fractal que colocaremos aqui, usando Sistema-L, será novamente um que se assemelha a uma planta. Anteriormente fizemos um fractal de geração de planta usando recursão. Veremos agora como poderíamos fazer uma planta, um tanto mais complexa e detalhada, usando Sistemas-L. Seja a seguinte configuração:

FX	Axioma
F → C0FF-[C1-F+F]+[C2+F-F]	Regra 1: Mude F para C0FF-[C1-F+F]+[C2+F-F]
X→C0FF+[C1+F]+[C3-F]	Regra 2: Mude X para C0FF+[C1+F]+[C3-F]

Ângulo: 25

Perceba que temos alguns caracteres novos aqui: “X”, “[”,“]”, “C0”, “C1”, “C2” e “C3”.

O “X” não faz nada, servindo apenas como elemento de geração da *string*. Pense nele como um “miolo” que serve apenas para permitir a geração de um tipo de ramo secundário da planta.

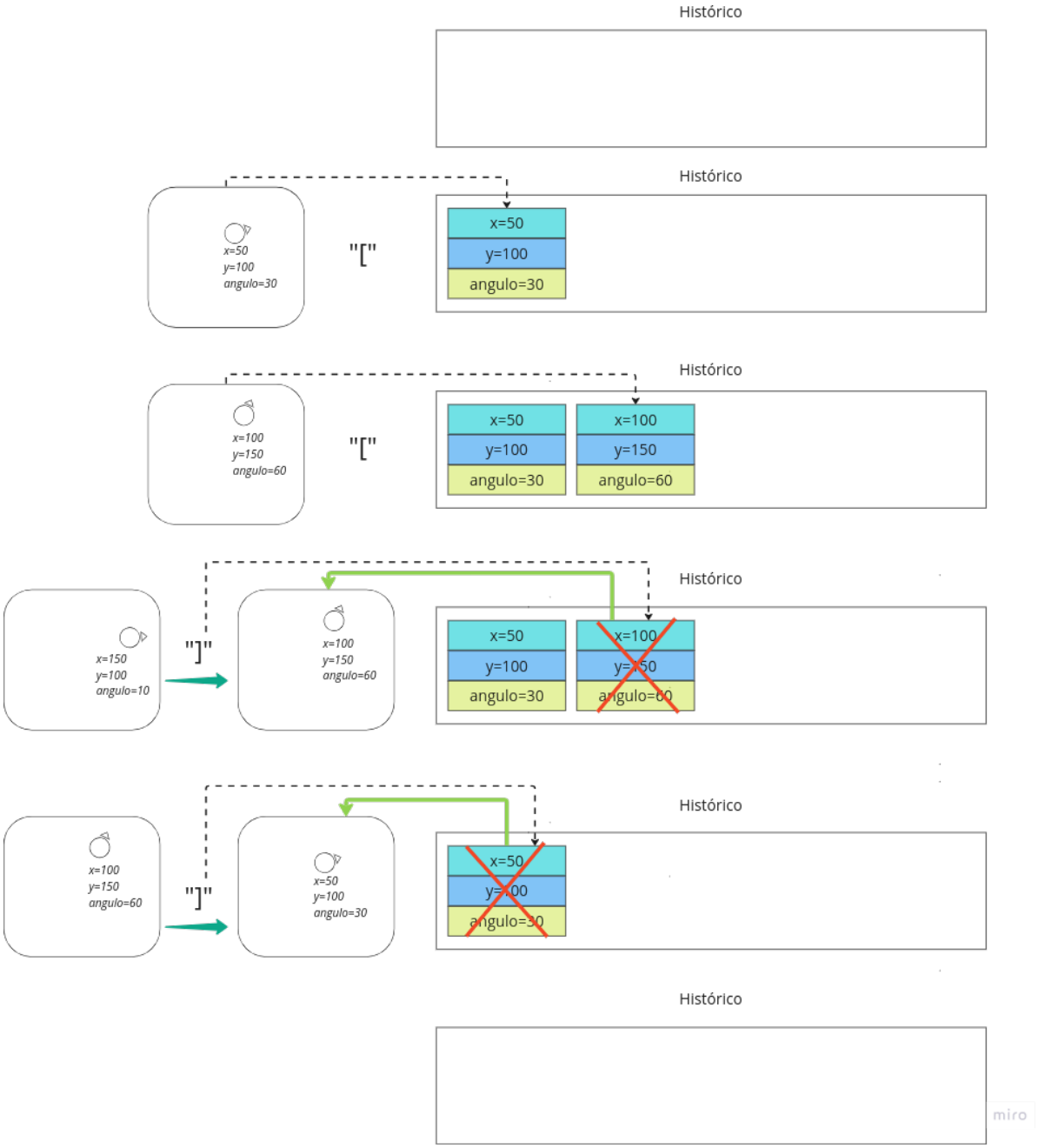
“C0”, “C1”, “C2” e “C3” representam mudanças de cores da linha desenhada pela tartaruga. Isso permitirá desenhar uma planta que possua o caule/galho de cor marrom (“C0”), e folhas em 3 tons de verde diferentes (“C1”, “C2” e “C3”).

O “[” e o “]” são os comandos novos mais importantes. Um “[” indica que as coordenadas *x* e *y,* e o ângulo de *direção* atual da tartaruga, devem ser armazenados em um histórico. Esse histórico é uma lista, e, sempre que encontramos um “[”, inserimos essas informações ao final da lista. Como as informações consistem em um conjunto de valores (*x, y, direção*), elas próprias são agrupados também por meio de uma “mini lista” (que podemos também chamar de **tupla**, ou **registro**), e essa “mini lista” é então inserida ao final do histórico. Ao

encontrarmos um "]", devemos remover os valores do final do histórico (ou seja, a última tupla de valores inserida no histórico), e fazer a tartaruga voltar para a posição *x* e *y* e apontar para o ângulo de *direção* contidos na tupla de valores removida. Dessa forma, será possível desenhar os ramos da planta, que devem retornar ao último ponto de ramificação para completar o desenho.

Esse histórico, em Ciência da Computação, é também conhecido como uma estrutura chamada de **pilha**. Pense em uma pilha de livros. Cada novo livro que você adiciona a essa pilha sempre é inserida no topo da pilha. Quando você remove um livro, deve-se também remover o livro que está no topo, nunca os que estão embaixo. Assim é com nosso histórico: o comando "[" adiciona (*x*, *y*, *direção*) ao final ("topo") da lista, e o comando "]" remove (*x, y, direção*) do final ("topo") da lista.

Abaixo é dada uma ilustração da estrutura e funcionamento desse histórico. O histórico começa vazio, e quando um "[" é lido, grava-se as informações atuais da tartaruga no final. Quando "]" é lido, remove-se as informações do final, que são utilizadas para restaurar o estado da tartaruga.

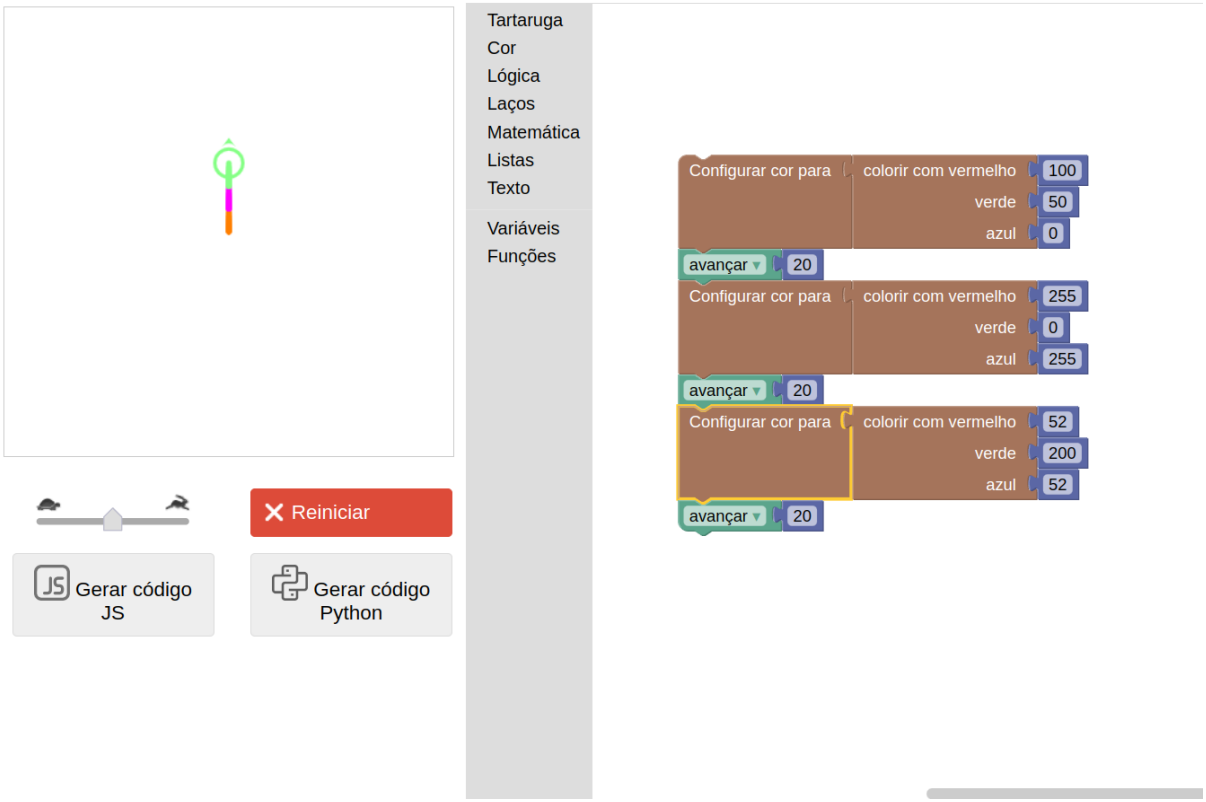


Vamos então ao nosso código em blocos. Como já vimos, basta alterar a função aplicar regras, inserindo as regras desse Sistema-L, e a função desenha sistema-L, que agora fica um pouquinho mais complexa devido à necessidade de se alterar cores e, principalmente, de se tratar com o histórico.

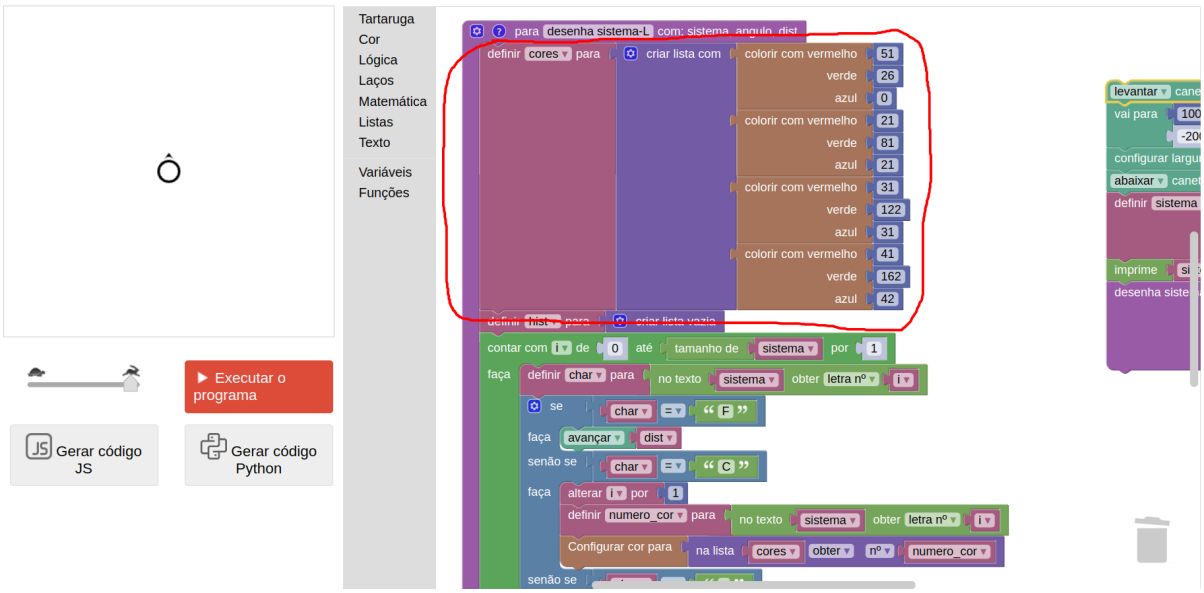
Na função desenha sistema-L, devemos primeiramente definir uma lista de cores, o que facilitará a troca de cores. Como a paleta de cores padrão disponível no menu Cores é muito pequena, o ideal é utilizarmos o bloco colorir com vermelho... verde... azul, que permite definir qualquer cor usando o padrão de cores conhecido como RGB (*red*, *green, blue*). A ideia é combinar diferentes intensidades das cores vermelho, verde e azul para criarmos qualquer cor. Essas intensidades são configuradas por meio de números de 0 a 255, sendo o 0 a menor intensidade da cor, e 255 a maior intensidade. Por exemplo, se configurarmos colorir com

vermelho 255 verde 0 azul 0, teremos uma cor totalmente vermelha vibrante. Se configurarmos colorir com vermelho 0 verde 255 azul 0, teremos um verde vibrante. Se configurarmos colorir com vermelho 255 verde 0 azul 255, teremos um tom rosa/violeta, conhecido como *fuchsia*. Se quisermos a cor preta, basta usar colorir com vermelho 0 verde 0 azul 0, isto é, a total ausência de cores. E se quisermos a cor branca, usamos colorir com vermelho 255 verde 255 azul 255, isto é, a junção de todas as cores, que resulta na cor branca. A brincadeira é muito parecida com a mistura de tintas de cores diferentes. Se você quiser testar mais cores usando o padrão RGB, faça uma busca na Internet por "RGB color picker". Um site recomendado é o https://www.w3schools.com/colors/colors_picker.asp.

Veja um exemplo abaixo do uso de diferentes cores usando o padrão RGB:

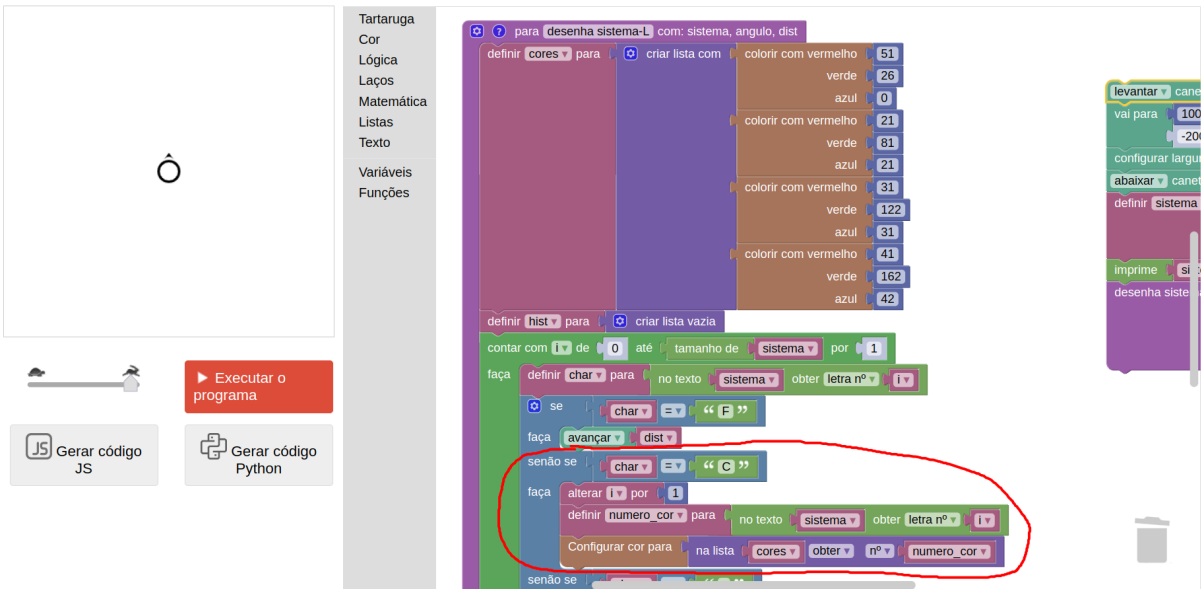


A figura abaixo mostra o trecho da função desenha sistema-L que inclui a definição da lista de cores. Note que usamos o bloco criar lista com, que cria uma lista com valores predefinidos. Assim, o primeiro item da lista será uma cor marrom-escuro, e as demais são diferentes tons de verde, a segunda sendo um verde mais escuro, seguido por dois tons de verde mais claros.

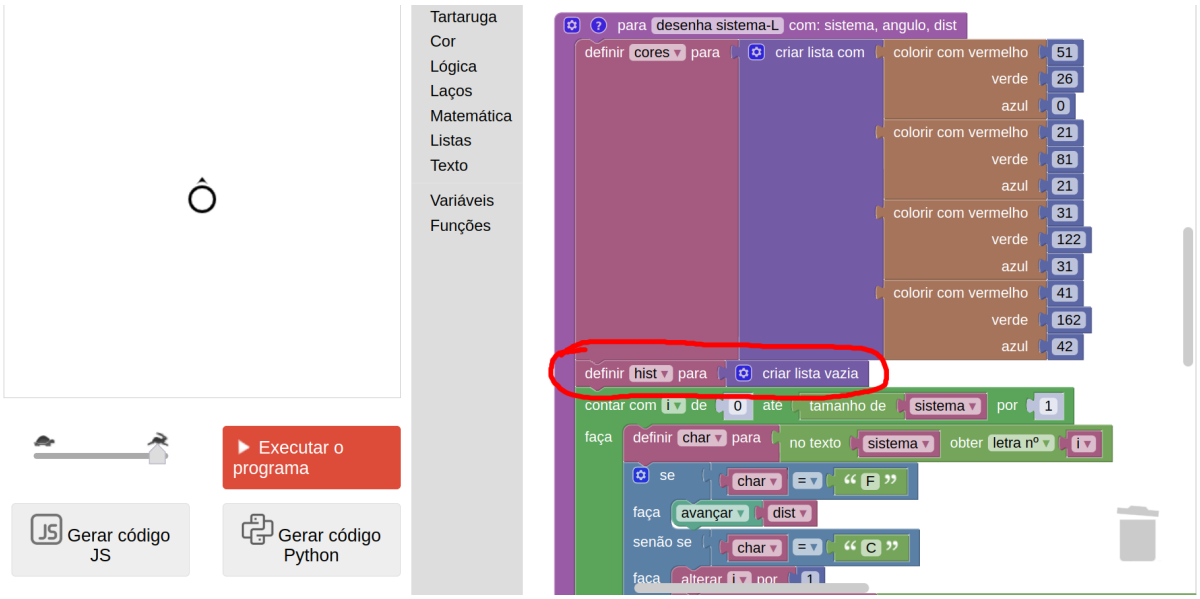


É importante lembrar que os elementos em uma lista podem ser acessados por meio de seus índices, que funcionam da mesma forma que com *strings*, isto é, começando do índice 0. Ou seja, se quisermos pegar a cor marrom (primeira cor), acessamos o índice 0, se quisermos a segunda cor, acessamos o índice 1, a terceira, o índice 2, e a quarta, o índice 3. Lembra que as cores nas regras de produção são representadas como "C0", "C1", "C2" e "C3"? Pois esta é a ideia: vamos utilizar o número que vem na frente da letra "C" para pegarmos o índice da cor na lista de cores. Então se encontrarmos "C" seguido por "0", trocamos a cor da caneta pela cor que está no índice 0 da lista de cores. Se encontrarmos "C" seguido por "1", trocamos pela cor que está no índice 1, e assim por diante. Formidável, não é mesmo? Pois é esse tipo de solução esperta que encanta muitos programadores!

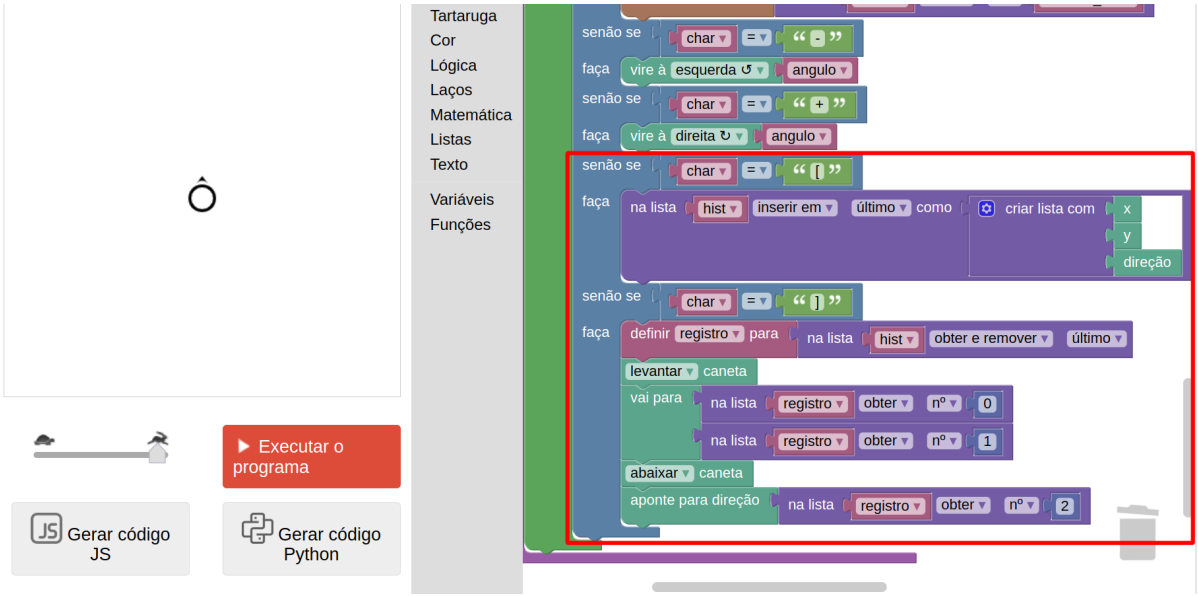
Veja abaixo como fica o trecho que verifica se o caractere atual é o "C". Aumenta-se em 1 o valor de i (que é o índice da *string* que está sendo processada) para pegar o próximo caractere, que será um dos números (0, 1, 2 ou 3); grava-se o número na variável numero_cor, e então usa-se a variável numero_cor como índice para pegar a cor na lista cores.



Vejamos agora a parte relacionada ao tratamento do histórico, direcionada pelos caracteres "[" e "]". Antes de tudo precisamos criar uma lista vazia, que aqui chamaremos por meio da variável hist, representando o histórico. Isso deve ser feito dentro da função desenha sistema-L, antes do laço (contar...), assim como a criação da lista de cores. Veja abaixo:



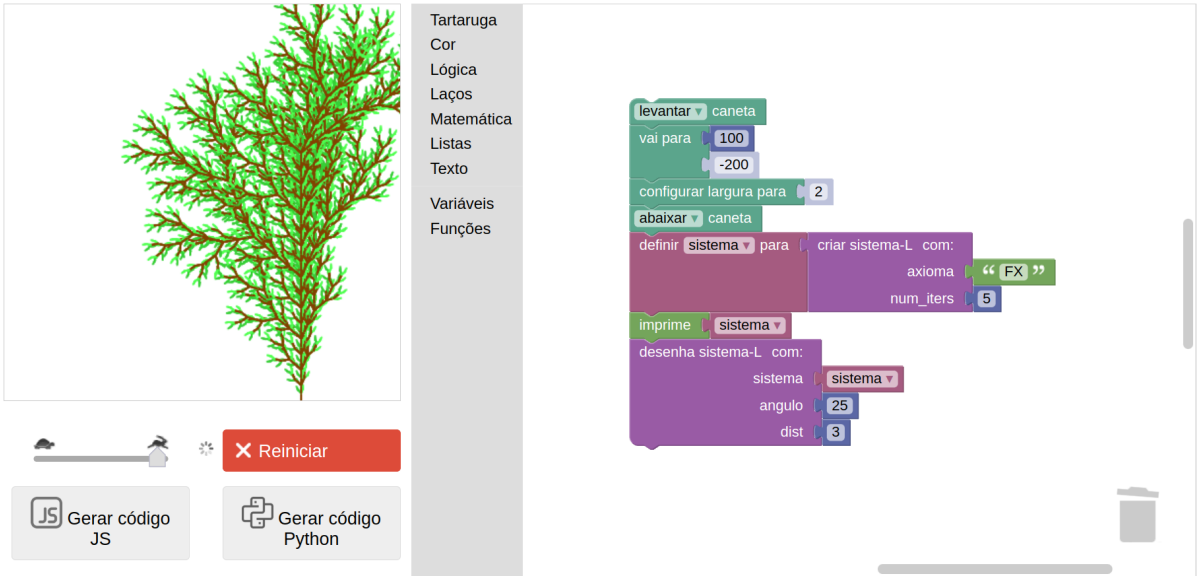
Em seguida, criamos dois novos condicionais senão se no bloco se... existente. Para isso, basta clicar na engrenagenzinha azul ao lado do bloco se..., a qual abrirá um balão onde você consegue configurar a quantidade de senão se. Um dos novos condicionais será para o "[" (guardar no histórico) e o outro para o "]" (recuperar do histórico e remover). Ambos são mostrados abaixo:



Note que, quando o caractere atual é "[", insere-se como último elemento da lista hist (o histórico) uma nova lista contendo x, y e direção (nesta ordem, isto é, x no índice 0, y no índice 1 e direção no índice 2). Esses três blocos (x, y e direção), que podem ser encontrados no menu Tartaruga, dão-nos as coordenadas *x* e *y* atuais da tartaruga, e seu ângulo de *direção*.

Quando o caractere atual é "]", usa-se o comando na lista <hist> obter e remover último para remover e guardar o último (x, y, direção) em uma variável chamada registro. Ou seja, após esse comando, a variável registro conterá aquela lista que havia sido salva dentro de hist anteriormente, quando encontrou-se o caractere "[". Com isso, levantamos a caneta e fazemos a tartaruga ir para a posição *x* igual ao item no índice 0 de registro (isto é, o primeiro elemento, que é o x) e posição *y* igual ao item no índice 1 de registro (segundo elemento, que é o y), e por fim pedimos que a tartaruga aponte para a direção igual ao item no índice 2 (terceiro elemento, que é o ângulo de direção). Com isso, restauramos a tartaruga à posição em que ela estava na última vez em que foi dado o comando "[".

Além disso, modificamos também ligeiramente alguns dos parâmetros no programa principal (conjunto de blocos que inicia tudo), e *voilá*:



Com isso terminamos essa saga. Se quiser brincar com mais exemplos de Sistemas-L, esta página possui alguns muito interessantes: https://www.kevs3d.co.uk/dev/lsystems.

Apêndice I

Dicas diversas de uso do Blockly Turtle

Comandos "Desfazer" e "Refazer"

Clique com o botão direito sobre a tela em branco e escolha a opção "Desfazer". Isso desfaz a última modificação realizada no código (blocos). A opção pode ser usada várias vezes para desfazer várias modificações. Pode-se também utilizar o atalho do teclado "Ctrl+Z".

Clique com o botão direito sobre a tela em branco e escolha a opção "Refazer". Isso refará a última modificação desfeita por meio do comando "Desfazer". Pode-se também utilizar o atalho do teclado "Ctrl+Y".

Comando "Duplicar Blocos"

Clique com o botão direito sobre um bloco e escolha a opção "Duplicar". Isto criará uma cópia exata do bloco. Se o bloco for do tipo que "envolve" outros blocos (por exemplo, os blocos condicionais e os blocos de laço), todo o conteúdo interno é copiado juntamente.

Comando "Limpar Blocos"

Clique com o botão direito sobre a tela em branco e escolha a opção "Limpar Blocos". Isso organizará os conjuntos de blocos, deixando-os alinhados.

Comando "Colapsar Bloco(s)"

Clique com o botão direito sobre um bloco e escolha a opção "Colapsar Bloco". Se fizer isso sobre um bloco que "envolve" (blocos de laço, funções e condicionais), o conjunto de blocos se contrairá em um único bloco, tornando o código mais compacto. O código funcionará normalmente enquanto estiver colapsado/contraído. Para voltar ao normal, clique com o botão direito sobre o bloco colapsado e escolha a opção "Expandir Bloco".

É possível colapsar todos os blocos também clicando em uma parte em branco da tela e escolhendo "Colapsar Blocos". O mesmo pode ser feito para expandi-los clicando em "Expandir Blocos".

Comando "Adicionar comentário"

Clique com o botão direito sobre um bloco e escolha a opção "Adicionar comentário". Se fizer isso, um botão azul com um ponto de interrogação ("?") aparecerá dentro do bloco. Clique nele e abrirá um "balãozinho" em que você pode escrever livremente um comentário sobre aquele bloco, por exemplo, para explicar qual a ideia ou motivo. Isso é muito útil para explicar o que faz uma função, ou um bloco maior de código que possui uma lógica intrincada que você deseja explicar para quem for ler seu código mais tarde.

Cores RGB

Como explicado na seção sobre Fractais, é possível definir diferentes cores para a caneta da tartaruga, e o Blockly Turtle permite a definição de cores usando o padrão RGB por meio do bloco *"colorir com vermelho ... verde ... azul ..."*. É possível definir as cores experimentalmente combinando-se diferentes intensidades dessas três cores. Cada intensidade

pode receber um valor entre 0 e 255. No entanto, muitas vezes é mais conveniente testar cores por meio de uma ferramenta do tipo *Color Picker*, que permite escolher cores por meio de uma interface gráfica intuitiva, retornando ao usuário os valores para as cores RGB. Há várias dessas ferramentas disponíveis na Web, mas sugerimos a seguinte: https://www.w3schools.com/colors/colors_picker.asp.