Anotações da Aula de Scratch

O que foi visto na aula?

Que aula ein! Pensando em absorver todo o conteúdo apresentado no vídeo da Aula 0, temos aqui algumas anotações:

- Bem vindos(as)!
- O que é ciência da computação?
- Representando números
- Texto
- Imagens, videos e sons
- Algoritmos
- Pseudocodigo
- Scratch

Bem vindo!

Este ano, o curso de Ciência da Computação de Harvard está no **Loeb Drama Center** da Harvard University, onde, graças à colaboração com o **American Repertory Theatre** , tiveram um palco incrível e até mesmo adereços para demonstrações.

Eles transforamaram uma **aquarela** do século 18 **do campus de Harvard feita** por um aluno, Jonathan Fisher, no plano de fundo do palco.

Vinte anos atrás, quando era um estudante de graduação, o professor David J. Malan superou sua própria apreensão, saindo de sua zona de conforto e cursou CS50 (para nós, CC50), descobrindo que o curso era menos sobre programação em si e mais sobre solução de problemas. Na verdade, dois terços dos alunos do CS50 nunca fizeram um curso de ciência da computação antes.

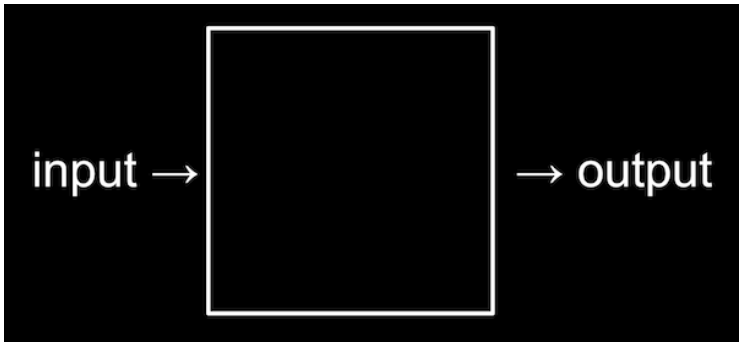
E o mais importante: o que importa neste curso não é tanto onde você termina em relação aos seus colegas, mas onde você termina em relação a si mesmo quando você começou a jornada.

Para começar o curso, vamos recriar um componente de um jogo do **Super Mario**, depois construiremos um aplicativo da web chamado CS50 Finance que permitirá aos usuários comprar e vender ações virtualmente e terminaremos o curso com a criação de seu próprio projeto final.

O que é Ciência da Computação?

A ciência da computação é fundamentalmente sobre resolução de problemas.

Podemos pensar na resolução de problemas como o processo de pegar algumas informações (detalhes sobre nosso problema) e gerar alguns resultados (a solução para nosso problema). A "caixa preta" no meio é a ciência da computação, ou o código que aprenderemos a escrever.



Para começar a fazer isso, precisaremos de uma maneira de representar entradas (inputs) e saídas (outputs), para que possamos armazenar e trabalhar com informações de forma padronizada.

Representando números

Podemos começar com a tarefa de marcar presença, contando o número de pessoas em uma sala. Com a nossa mão, podemos levantar um dedo de cada vez para representar cada pessoa, mas não poderemos contar muito alto. Este sistema é denominado **unário**, onde cada dígito representa um único valor de um.

Provavelmente aprendemos um sistema mais eficiente para representar números, onde temos dez dígitos, de 0 a 9:

```
0 1 2 3 4 5 6 7 8 9
```

- Este sistema é denominado decimal, ou base 10, uma vez que existem dez valores diferentes que um dígito pode representar.

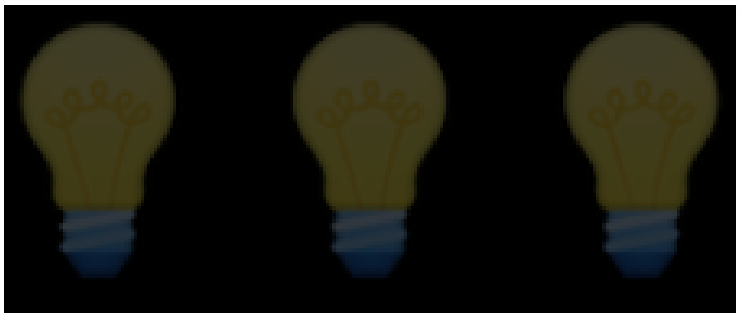
Os computadores usam um sistema mais simples chamado **binário**, ou base dois, com apenas dois dígitos possíveis, **0 e 1**.

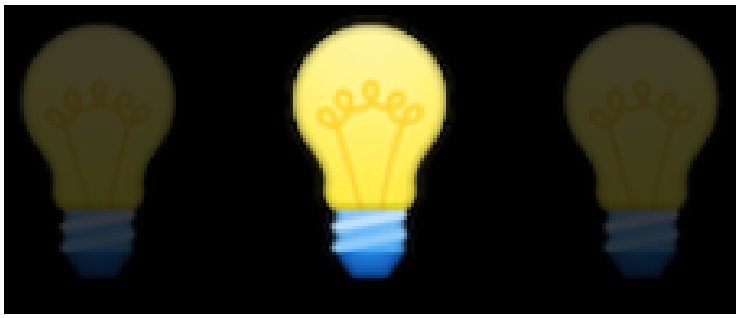
- Cada dígito binário também é chamado de **bit**.

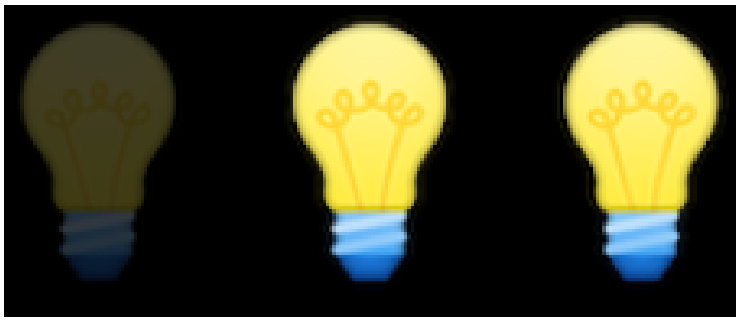
Como os computadores funcionam com eletricidade, que pode ser ligada ou desligada, podemos convenientemente representar um bit ligando ou desligando alguma chave para representar 0 ou 1.

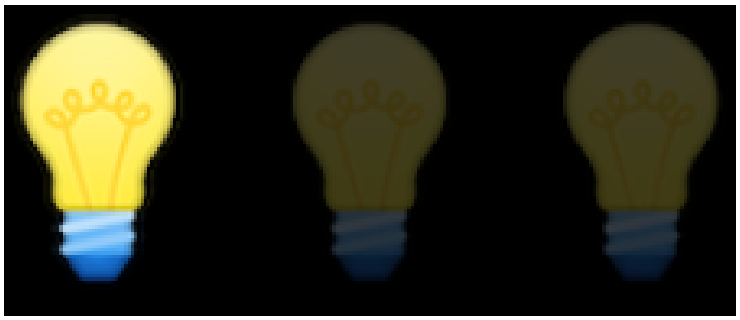
- Com uma lâmpada, por exemplo, podemos ligá-la para contar até 1.

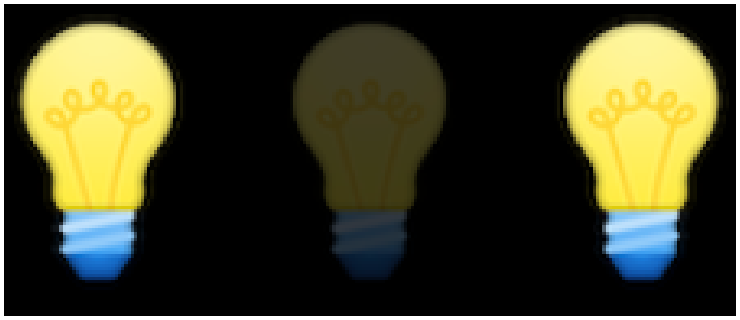
Com três lâmpadas, podemos acendê-las em padrões diferentes e contar de 0 (com as três apagadas) a 7 (com as três acesas):

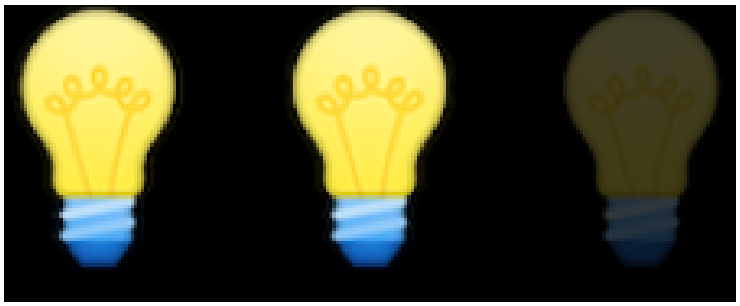


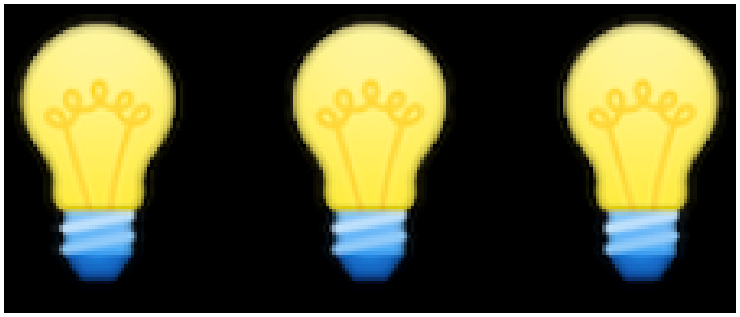












Dentro dos computadores modernos, não existem lâmpadas, mas milhões de pequenos interruptores chamados **transistores** que podem ser ligados e desligados para representar valores diferentes. Por exemplo, sabemos que o seguinte número em decimal representa cento e vinte e três.

```
1 2 3
```

- O **3** está na coluna das unidades, o **2** está na coluna das dezenas e o **1** está na coluna das centenas.
- Portanto, **123** é **100 × 1 + 10 × 2 + 1 × 3 = 100 + 20 + 3 = 123**.
- Cada casa de um dígito representa uma potência de dez, pois há dez dígitos possíveis para cada casa. O lugar mais à direita é para 100, o do meio 101 e o lugar mais à esquerda 10^2

```
10² 10¹ 10⁰
 1   2   3
```

Em binário, com apenas dois dígitos, temos potências de dois para cada valor de casa:

```
2² 2¹ 2⁰
#  #  #
```

Equivalente a:

```
4  2  1
#  #  #
```

Com todas as lâmpadas ou interruptores desligados, ainda teríamos um valor de 0:

```
4 2 1
0 0 0
```

Agora, se mudarmos o valor binário para, digamos, **0 1 1**, o valor decimal seria 3, uma vez que somamos o 2 e o 1:

```
4 2 1
0 1 1
```

Se tivéssemos mais lâmpadas, poderíamos ter um valor binário de **110010**, que teria o valor decimal equivalente a 50:

```
32 16  8  4  2  1
 1  1  0  0  1  0
```

Observe que 32 + 16 + 2 = 50 . Com mais bits, podemos contar até números ainda maiores.

Texto

Para representar as letras, tudo o que precisamos fazer é decidir como os números são mapeados para as letras. Alguns humanos, muitos anos atrás, decidiram coletivamente um mapeamento padrão de números em letras. A letra "A", por exemplo, é o número 65, e "B" é 66 e assim por diante. Ao usar o contexto, como quando estamos olhando uma planilha ou um e-mail, diferentes programas podem interpretar e exibir os mesmos bits como números ou texto.

O mapeamento padrão, **ASCII**, também inclui letras minúsculas e pontuação.

Se recebêssemos uma mensagem de texto com um padrão de bits que tivesse os valores decimais **72, 73 e 33**, esses bits seriam mapeados para as letras **HI!**. Cada letra é normalmente representada com um padrão de oito bits, ou um **byte**, então as sequências de bits que receberíamos são **01001000, 01001001 e 00100001.**

- Podemos já estar familiarizados com o uso de bytes como uma unidade de medida para dados, como em megabytes ou gigabytes, para milhões ou bilhões de bytes.

Com oito bits, ou um byte, podemos ter 28 ou 256 valores diferentes (incluindo zero). O valor mais alto que podemos contar seria 255.

Outros caracteres, como letras com acentos e símbolos em outros idiomas, fazem parte de um padrão chamado **Unicode**, que usa mais bits do que ASCII para acomodar todos esses caracteres.

- Quando recebemos um emoji, nosso computador está apenas recebendo um número binário que mapeia para a imagem do emoji baseado no padrão Unicode. Por exemplo, o emoji "rosto com lágrimas de alegria" tem apenas os bits **00000001111101100000010**:

> 😂

Imagem, vídeo e sons

Uma imagem, como a imagem do emoji, é composta de cores. Com apenas bits, podemos mapear números para cores também. Existem muitos sistemas diferentes para representar cores, mas um comum é **RGB**, que representa cores diferentes indicando a quantidade de vermelho, verde e azul dentro de cada cor.

Por exemplo, nosso padrão de bits anterior, **72, 73 e 33** pode indicar a quantidade de vermelho, verde e azul em uma cor. E nossos programas saberiam que esses bits são mapeados para uma cor se abríssemos um arquivo de imagem, em vez de recebê-los em uma mensagem de texto.

Cada número pode ser um byte, com 256 valores possíveis, portanto, com três bytes, podemos representar milhões de cores. Nossos três bytes de cima representariam um tom escuro de amarelo:

Observe este exemplo:

Os pontos, ou quadrados, em nossas telas são chamados de **pixels**, e as imagens são compostas por muitos milhares ou milhões desses pixels também. Então, usando três bytes para representar a cor de cada pixel, podemos criar imagens. Podemos ver os pixels em um emoji se aumentarmos o zoom, por exemplo:

A **resolução** de uma imagem é o número de pixels que existem, horizontalmente e verticalmente, portanto, uma imagem de alta resolução terá mais pixels e exigirá mais bytes para ser armazenada.

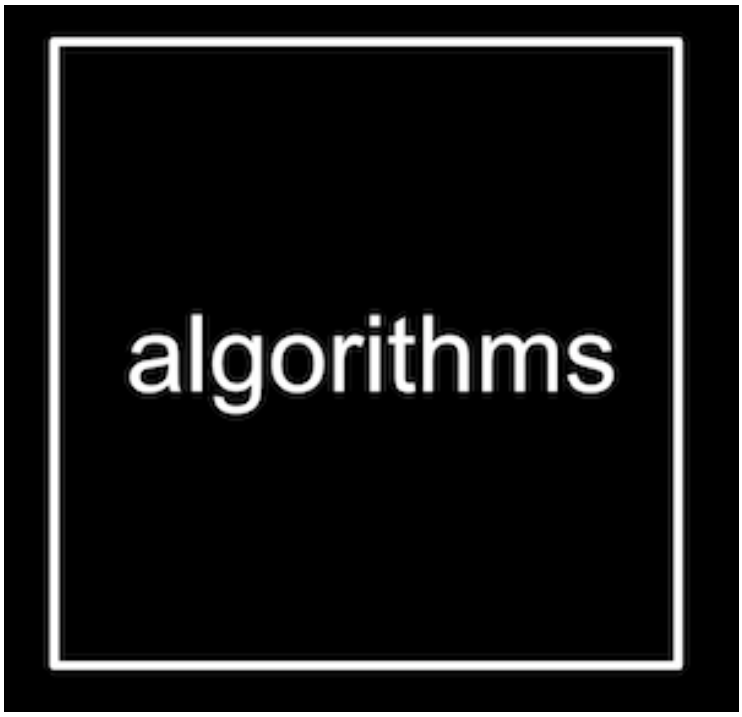
Os vídeos são compostos de muitas imagens, mudando várias vezes por segundo para nos dar a aparência de movimento, como um **flipbook** antigo faria.

A música também pode ser representada com bits, com mapeamentos de números para notas e durações, ou mapeamentos mais complexos de bits para frequências de som em cada momento transcorrido.

Os formatos de arquivo, como JPEG e PNG, ou documentos do Word ou Excel, também são baseados em algum padrão com o qual alguns humanos concordaram, para representar informações com bits.

Algoritmos

Agora que podemos representar inputs e outputs, podemos trabalhar na resolução de problemas. A caixa preta de antes contém **algoritmos**, instruções passo-a-passo para resolver problemas.



Os humanos também podem seguir algoritmos, como receitas para cozinhar. Ao programar um computador, precisamos ser mais precisos com nossos algoritmos para que nossas instruções não sejam ambíguas ou mal interpretadas.

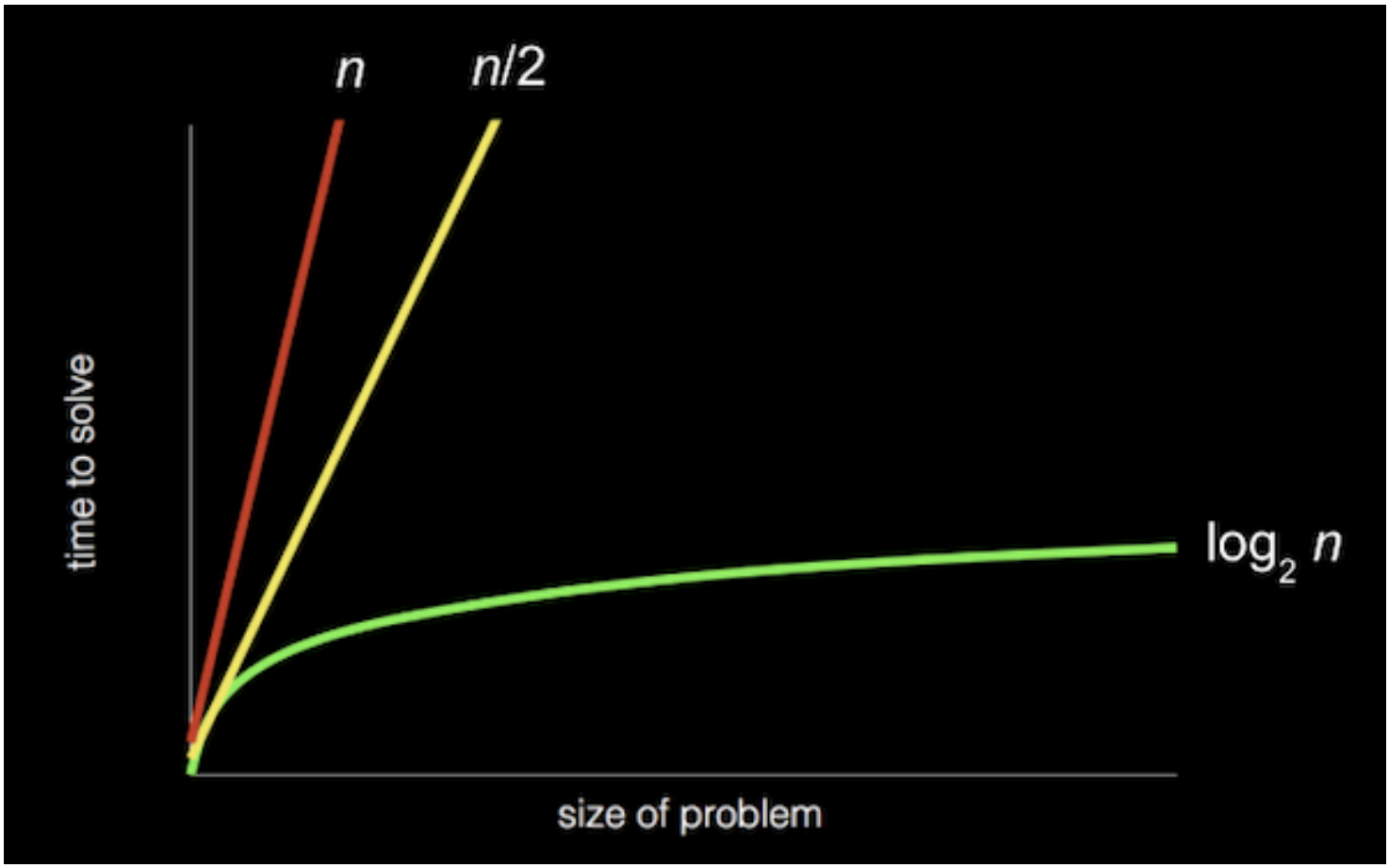
Podemos ter um aplicativo em nossos telefones que armazena nossos contatos, com seus nomes e números de telefone classificados em ordem alfabética. O equivalente "old-school" pode ser uma lista telefônica, uma cópia impressa de nomes e números de telefone.

Nossa contribuição para o problema de encontrar o número de alguém seria a lista telefônica e um nome a ser procurado. Podemos abrir o livro e começar da primeira página, procurando um nome uma página de cada vez. Este algoritmo estaria **correto**, já que eventualmente encontraremos o nome que buscamos se ele estiver no livro.

Podemos folhear o livro duas páginas por vez, mas esse algoritmo não estará correto, pois podemos pular a página com nosso nome nela. Podemos consertar esse bug, ou engano, voltando uma página se formos longe demais, pois sabemos que a lista telefônica está classificada em ordem alfabética.

Outro algoritmo seria abrir a lista telefônica ao meio, decidir se nosso nome estará na metade esquerda ou na metade direita do livro (porque o livro está em ordem alfabética) e reduzir o tamanho do nosso problema pela metade. Podemos repetir isso até encontrar nosso nome, dividindo o problema pela metade a cada vez. Com 1.024 páginas para começar, precisaríamos apenas de 10 etapas de divisão ao meio antes de termos apenas uma página restante para verificar. Podemos ver isso visualizado em uma **animação de dividir uma lista telefônica ao meio repetidamente**, em comparação com a **animação de pesquisar uma página por vez**.

Na verdade, podemos representar a eficiência de cada um desses algoritmos com um gráfico:



Nossa primeira solução, pesquisar uma página por vez, pode ser representada pela linha vermelha: nosso tempo para resolver aumenta linearmente à medida que o tamanho do problema aumenta. *n* é um número que representa o tamanho do problema, portanto, com n páginas em nossas listas telefônicas, temos que realizar até n etapas para encontrar um nome.

A segunda solução, pesquisar duas páginas por vez, pode ser representada pela linha amarela: nossa inclinação é menos acentuada, mas ainda linear. Agora, precisamos apenas de (aproximadamente) *n / 2* etapas, já que viramos duas páginas de cada vez.

Nossa solução final, dividindo a lista telefônica ao meio a cada vez, pode ser representada pela linha verde, com uma relação fundamentalmente diferente entre o tamanho do problema e o tempo de resolvê-lo: **logarítmica**, já que nosso tempo de resolução aumenta cada vez mais lentamente conforme o tamanho do problema aumenta.

Em outras palavras, se a lista telefônica fosse de 1.000 para 2.000 páginas, precisaríamos apenas de mais uma etapa para encontrar nosso nome. Se o tamanho dobrasse novamente de 2.000 para 4.000 páginas, ainda precisaríamos de apenas mais uma etapa. A linha verde é rotulada log2 *n*, ou log base 2 de *n*, já que estamos dividindo o problema por dois em cada etapa.

Quando escrevemos programas usando algoritmos, geralmente nos preocupamos não apenas com o quão corretos eles são, mas também com o quão bem projetados eles são, considerando fatores como eficiência.

Pseudocódigo

Podemos escrever **pseudocódigo**, que é uma representação de nosso algoritmo em inglês preciso (ou alguma outra linguagem humana):

```
1 Pegue a lista telefônica
2 Abra no meio da lista telefônica
3 Olhe para a página
4 Se a pessoa estiver na página
5    Ligar para pessoa
6 Caso contrário, se a pessoa estiver mais para o início do livro
7    Abrir no meio da metade esquerda do livro
8    Volte para a linha 3
9 Caso contrário, se a pessoa estiver mais para o final do livro
10   Abrir no meio da metade direita do livro
11   Volte para a linha 3
12 Caso contrário
13   Desistir
```

- Com essas etapas, verificamos a página do meio, decidimos o que fazer e repetimos. Se a pessoa não estiver na página e não houver mais páginas sobrando no livro, paramos. E esse caso final é particularmente importante para lembrar. Quando outros programas em nossos computadores esquecem esse caso final, eles podem travar ou parar de responder, uma vez que encontraram um caso que não foi contabilizado, ou continuar a repetir o mesmo trabalho continuamente nos bastidores, sem fazer nenhum progresso.

Algumas dessas linhas começam com verbos ou ações. Começaremos chamando estas *funções:*

```
1 Pegue a lista telefônica
2 Abra no meio da lista telefônica
3 Olhe para a página
4 Se a pessoa estiver na página
5    Ligar para pessoa
6 Caso contrário, se a pessoa estiver mais para o início do livro
7    Abrir no meio da metade esquerda do livro
8    Volte para a linha 3
9 Caso contrário, se a pessoa estiver mais para o final do livro
10   Abrir no meio da metade direita do livro
11   Volte para a linha 3
12 Caso contrário
13   Desistir
```

Também temos ramificações que levam a caminhos diferentes, como bifurcações na estrada, que chamaremos de *condições:*

```
1 Pegue a lista telefônica
2 Abra no meio da lista telefônica
3 Olhe para a página
4 Se a pessoa estiver na página
5    Ligar para pessoa
6 Caso contrário, se a pessoa estiver mais para o início do livro
7    Abrir no meio da metade esquerda do livro
8    Volte para a linha 3
9 Caso contrário, se a pessoa estiver mais para o final do livro
10   Abrir no meio da metade direita do livro
11   Volte para a linha 3
12 Caso contrário
13   Desistir
```

E as perguntas que decidem para onde vamos são chamadas de *expressões booleanas* , que eventualmente resultam em um valor de sim ou não, verdadeiro ou falso:

```
1 Pegue a lista telefônica
2 Abra no meio da lista telefônica
3 Olhe para a página
4 Se a pessoa estiver na página
5    Ligar para pessoa
6 Caso contrário, se a pessoa estiver mais para o início do livro
7    Abrir no meio da metade esquerda do livro
8    Volte para a linha 3
9 Caso contrário, se a pessoa estiver mais para o final do livro
10   Abrir no meio da metade direita do livro
11   Volte para a linha 3
12 Caso contrário
13   Desistir
```

Por último, temos palavras que criam ciclos, onde podemos repetir partes de nosso programa, chamadas *loops:*


```
1 Pegue a lista telefônica
2 Abra no meio da lista telefônica
3 Olhe para a página
4 Se a pessoa estiver na página
5    Ligar para pessoa
6 Caso contrário, se a pessoa estiver mais para o início do livro
7    Abrir no meio da metade esquerda do livro
8    Volte para a linha 3
9 Caso contrário, se a pessoa estiver mais para o final do livro
10   Abrir no meio da metade direita do livro
11   Volte para a linha 3
12 Caso contrário
13   Desistir
```


Scracth

Podemos escrever programas com os blocos de construção que acabamos de descobrir:

- Funções
- Condições
- Expressões Booleanas
- Loops

E também descobriremos recursos adicionais, incluindo:

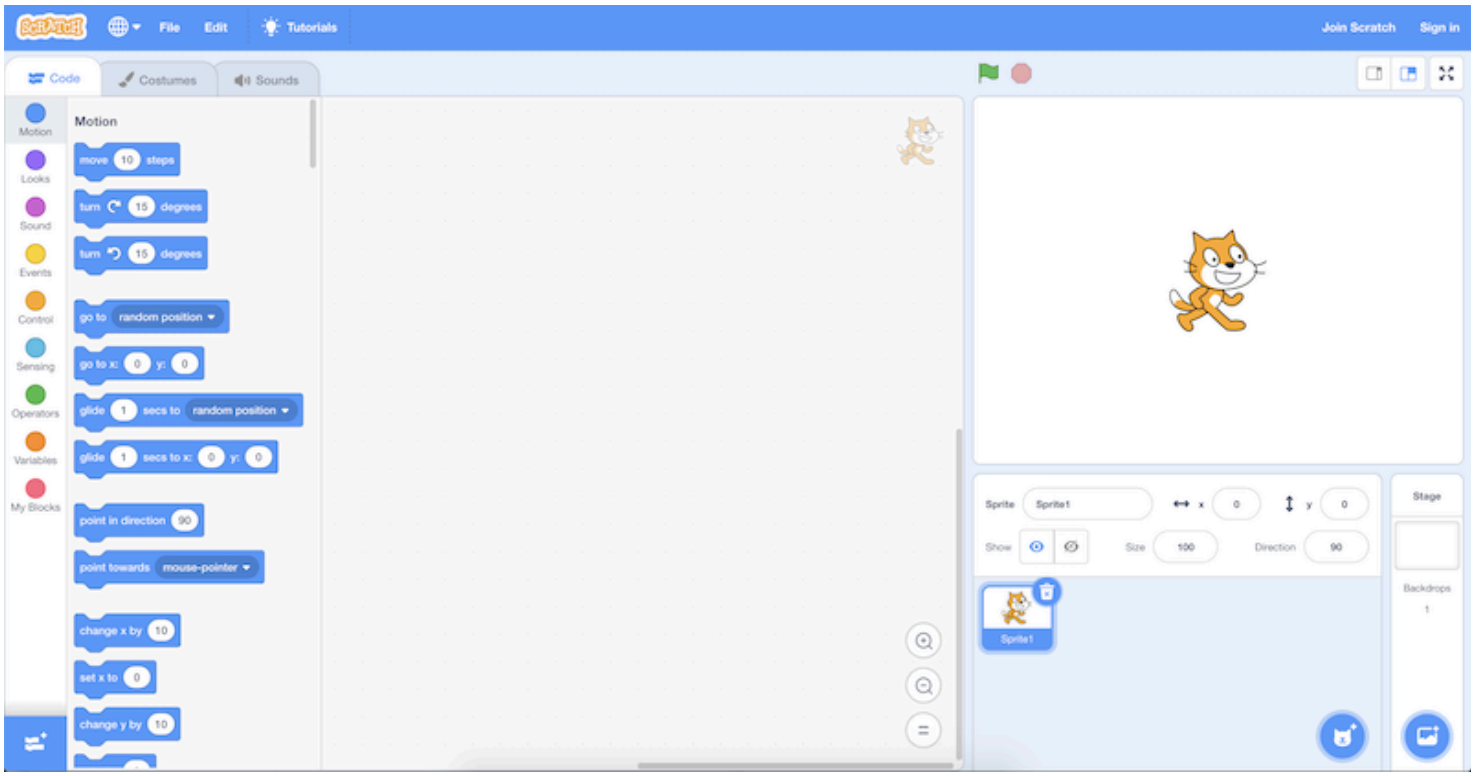
- Variáveis
- Threads
- Eventos
- ...

Antes de aprendermos a usar uma linguagem de programação baseada em texto chamada C, usaremos uma linguagem de programação gráfica chamada **Scratch**, onde arrastaremos e soltaremos blocos que contêm instruções.

Um programa simples em C que imprime "hello, world" seria assim:

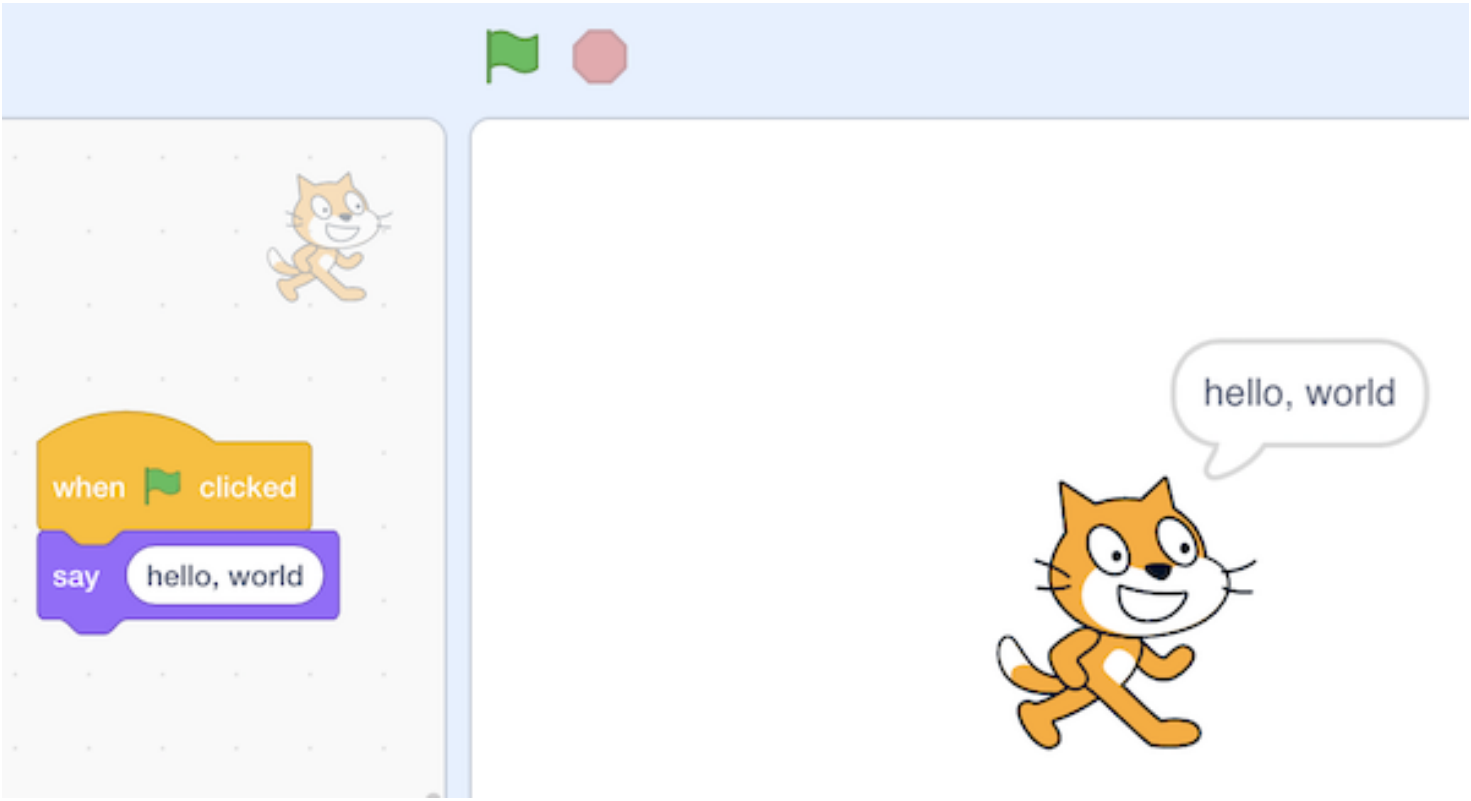
```c
#include <stdio.h>

int main(void)
{
    printf("hello, world\n");
}
```

Há muitos símbolos e sintaxe, ou a organização desses símbolos, que precisaríamos entender. O ambiente de programação do Scratch é um pouco mais amigável:



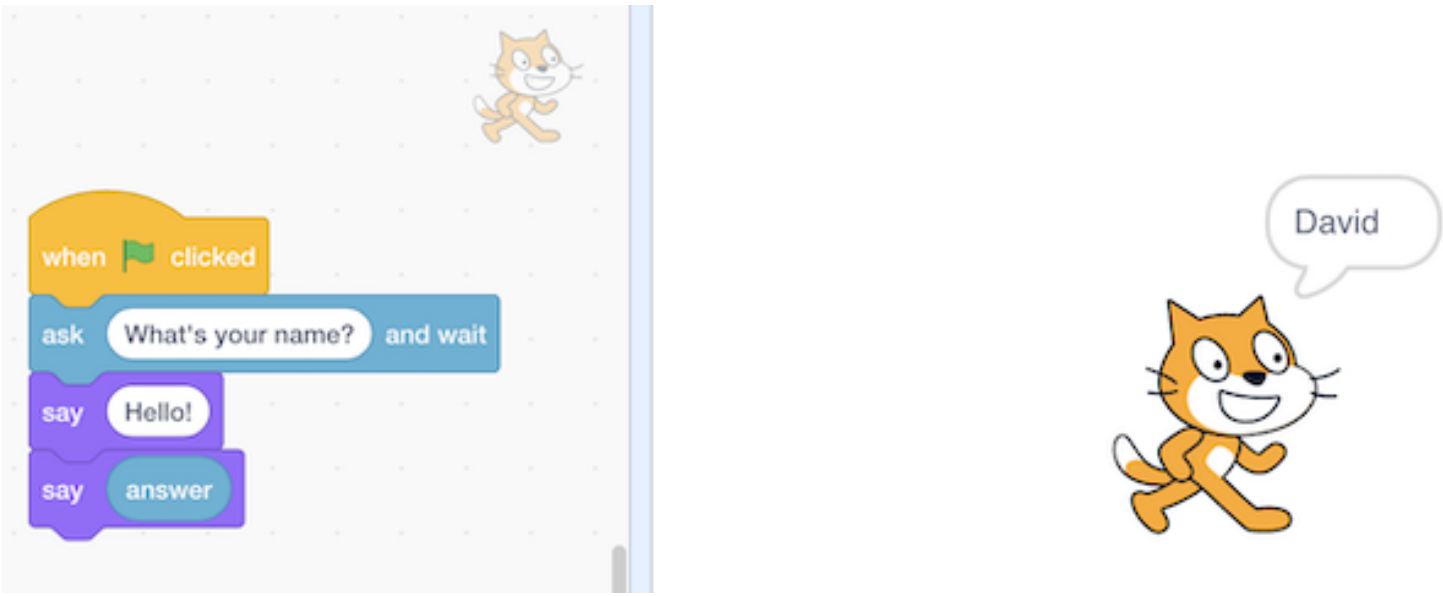
- Na parte superior direita, temos um palco que será exibido pelo nosso programa, onde podemos adicionar ou alterar fundos, personagens (chamados de sprites no Scratch) e mais.
- À esquerda, temos peças de quebra-cabeça que representam funções, variáveis ou outros conceitos, que podemos arrastar e soltar na nossa área de instruções no centro.
- Na parte inferior direita, podemos adicionar mais personagens para o nosso programa usar.

Podemos arrastar alguns blocos para fazer o Scratch dizer "hello, world":



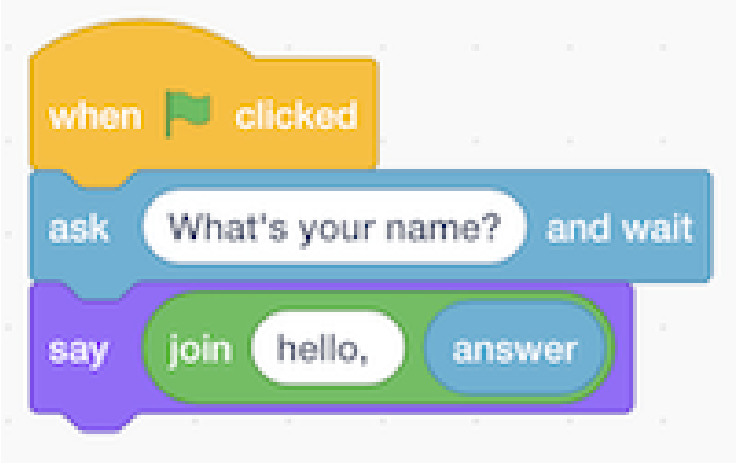
O bloco "when bandeira verde clicked" se refere ao início do nosso programa (já que há uma bandeira verde acima do palco que podemos usar para iniciá-lo), e abaixo dele encaixamos um bloco "say" (dizer) e digitamos "hello, world". E podemos descobrir o que esses blocos fazem explorando a interface e experimentando.

Também podemos arrastar o bloco "ask and wait", com uma pergunta como “What’s your name? (Qual é o seu nome?)", e combiná-lo com um bloco "say" para a resposta (**answer**):



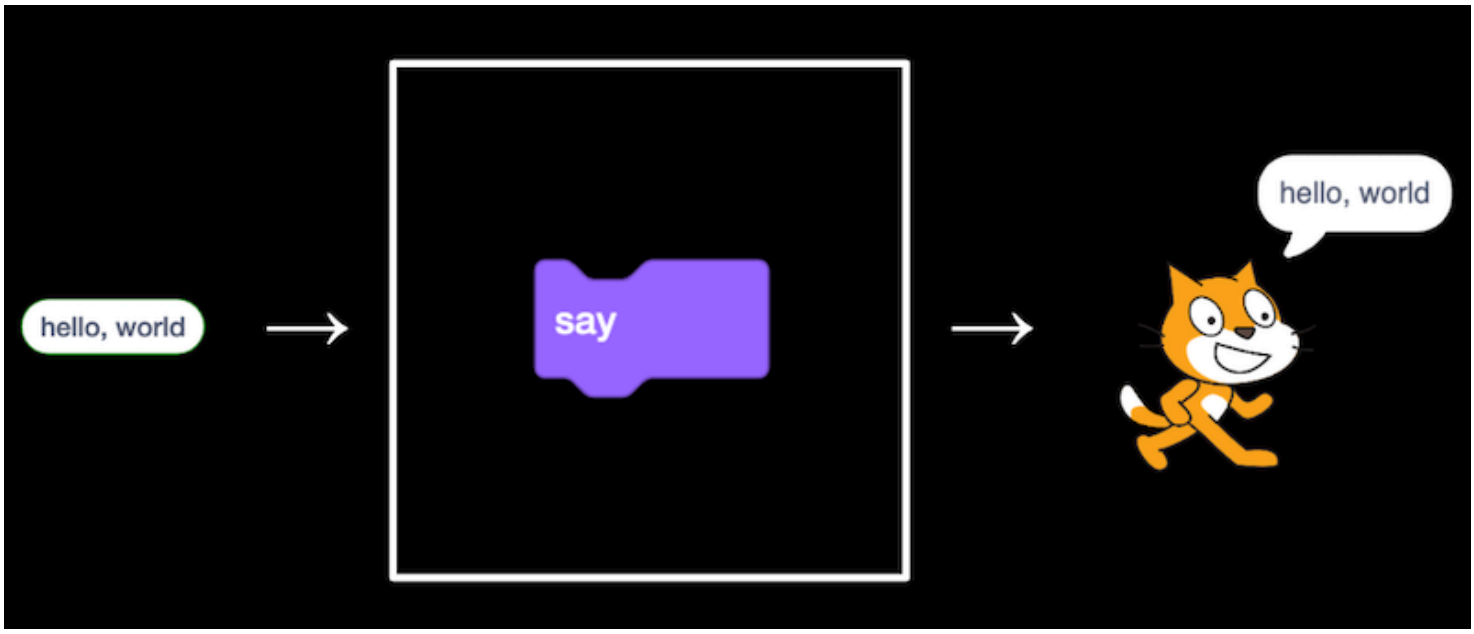
O bloco "**answer**" é uma variável, ou valor, que armazena o que o usuário do programa digita, e podemos colocá-lo em um bloco "say" arrastando e soltando também.

Mas não esperamos depois de dizer "Hello" com o primeiro bloco, então podemos usar o bloco "join" (unir) para combinar duas frases para que nosso gato possa dizer "hello, David":

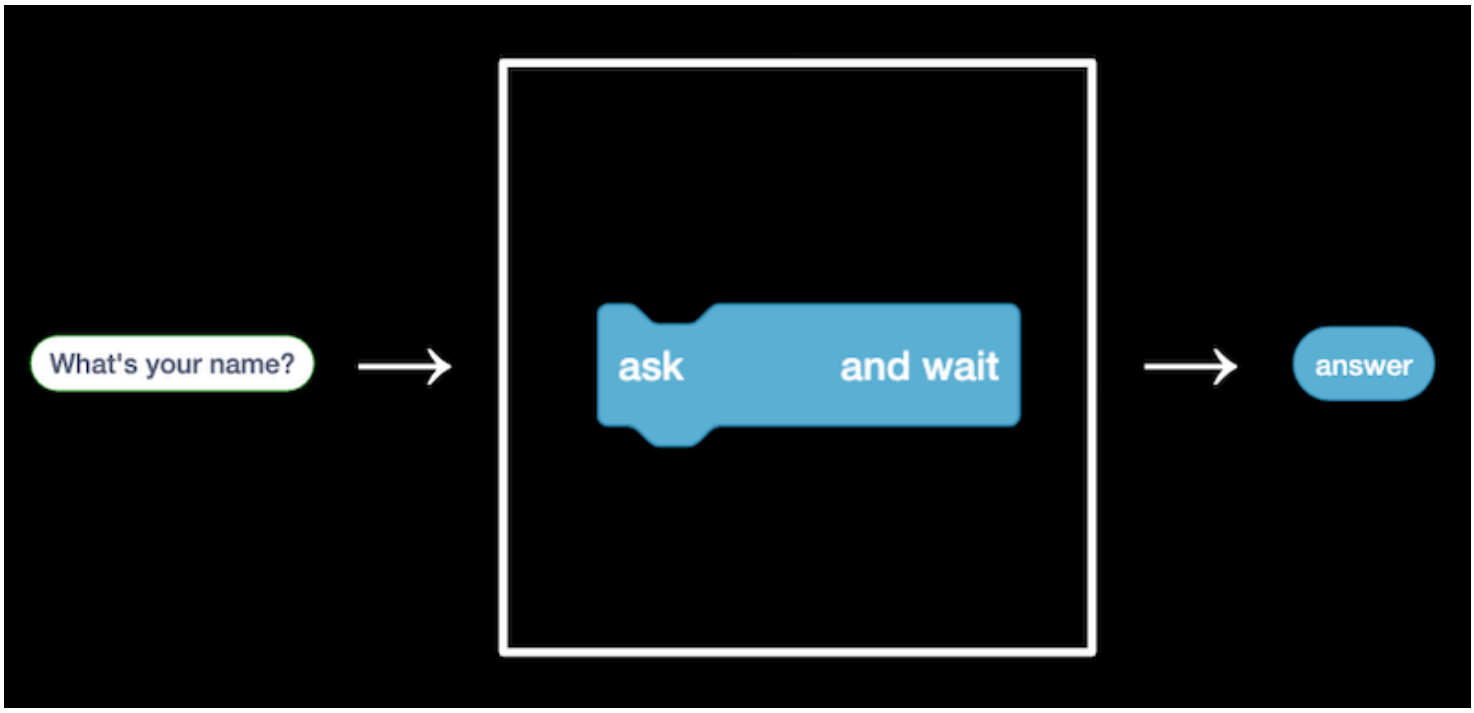


Quando tentamos aninhar blocos, ou colocá-los um dentro do outro, o Scratch nos ajuda expandindo os lugares onde eles podem ser usados.

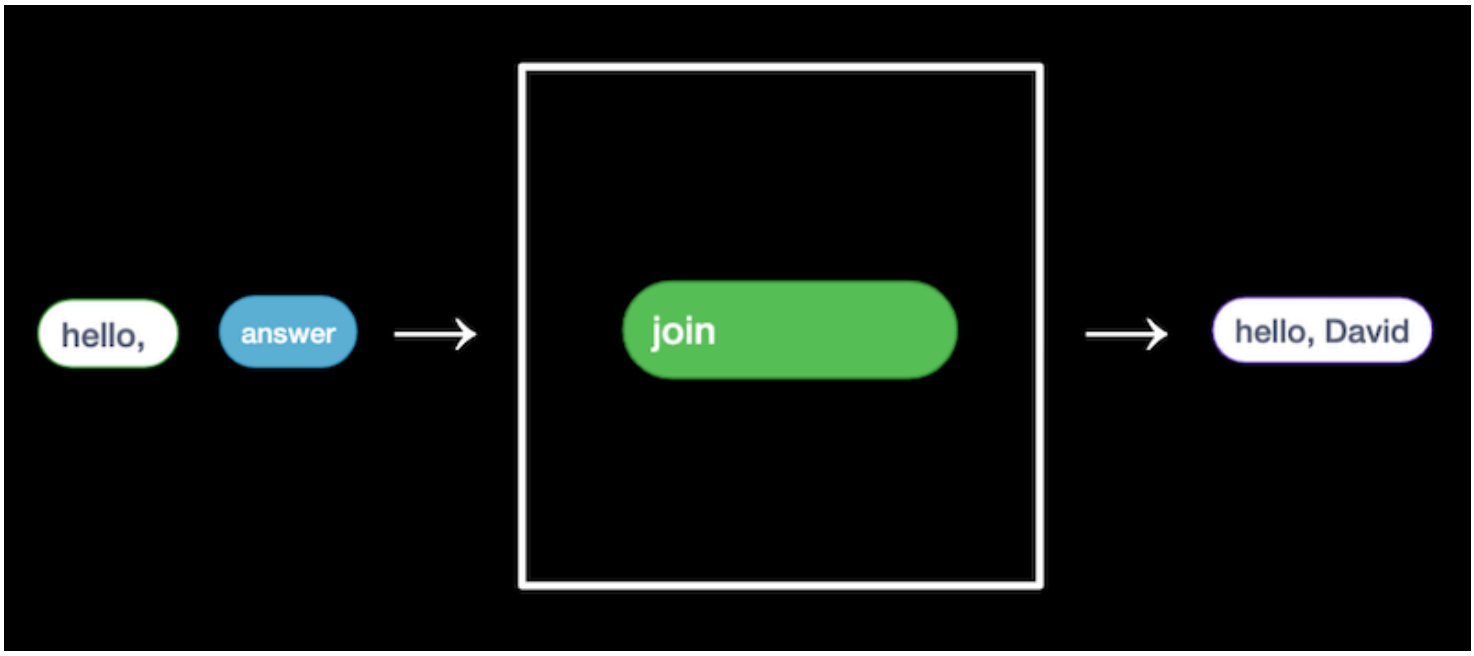
Na verdade, o bloco "say" em si é como um algoritmo, onde fornecemos uma entrada de "hello, world" e ele produziu a saída do Scratch (o gato) "dizendo" essa frase:



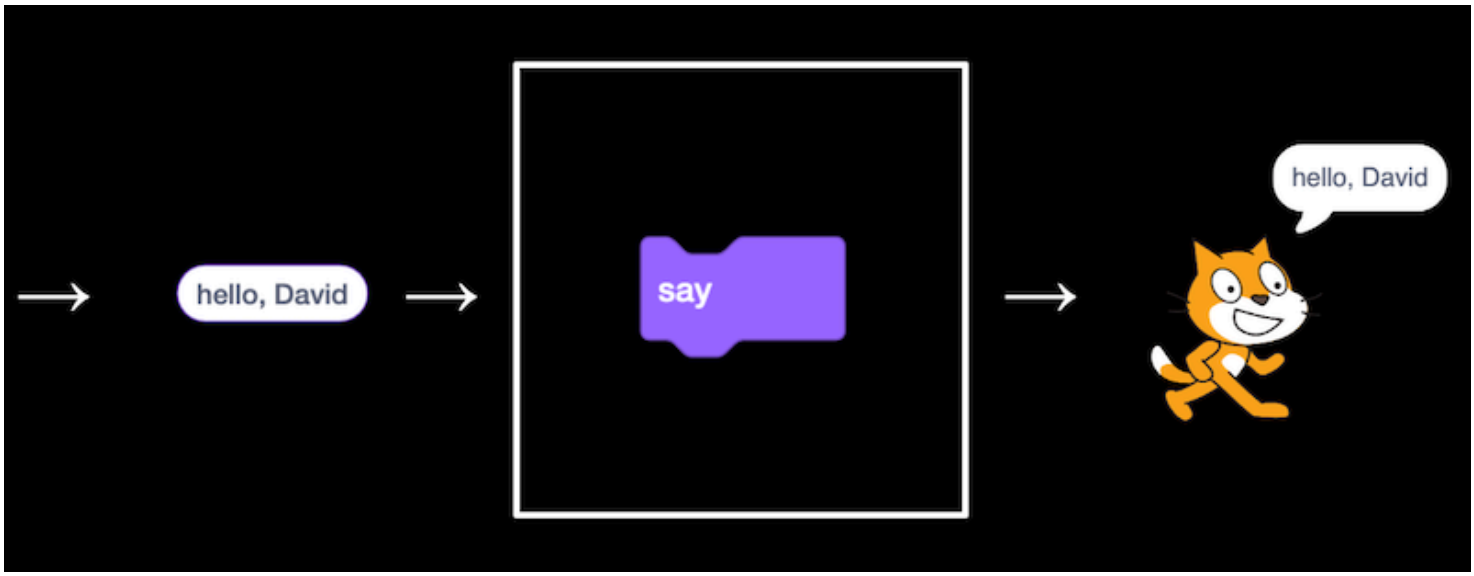
O bloco "ask" também recebe uma entrada (a pergunta que queremos fazer) e produz a saída do bloco "resposta":



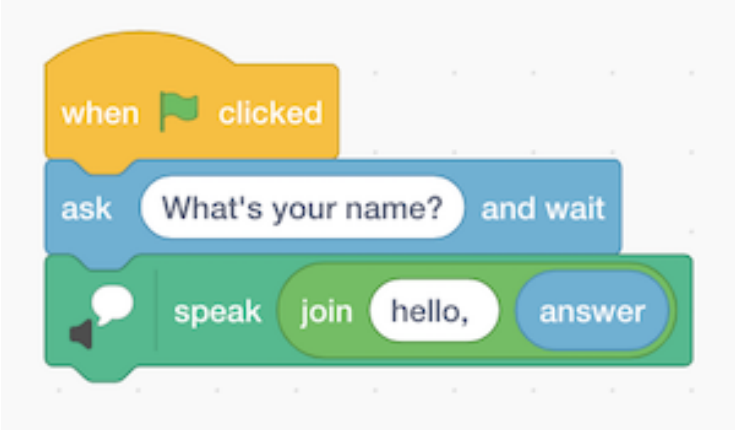
Podemos então usar o bloco "resposta" junto com nosso próprio texto, "hello, ", como duas entradas para o algoritmo de unir...



... cuja saída podemos passar como entrada para o bloco "dizer":

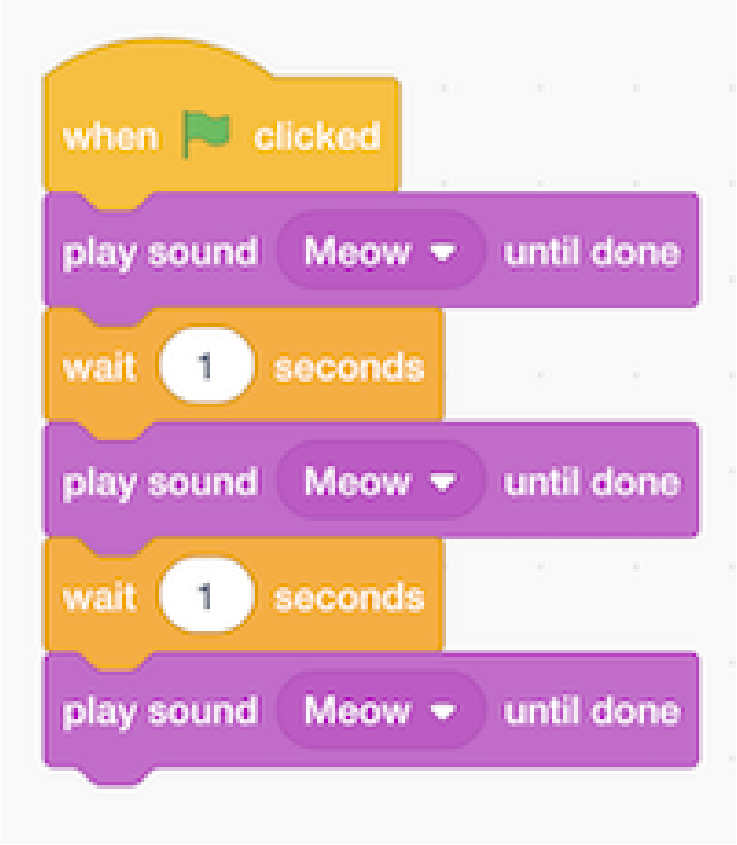


Na parte inferior esquerda da tela, vemos um ícone para extensões, e uma delas é chamada Texto para Fala. Depois de adicioná-la, podemos usar o bloco "speak" para ouvir nosso gato falar:



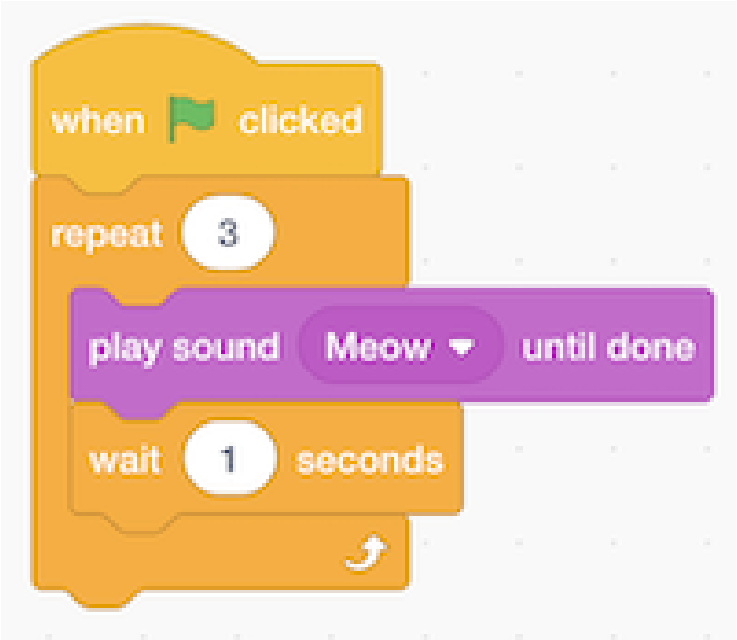
A extensão Texto para Fala, graças à nuvem, ou servidores de computador na internet, está convertendo nosso texto em áudio.

Podemos tentar fazer o gato dizer "miau":



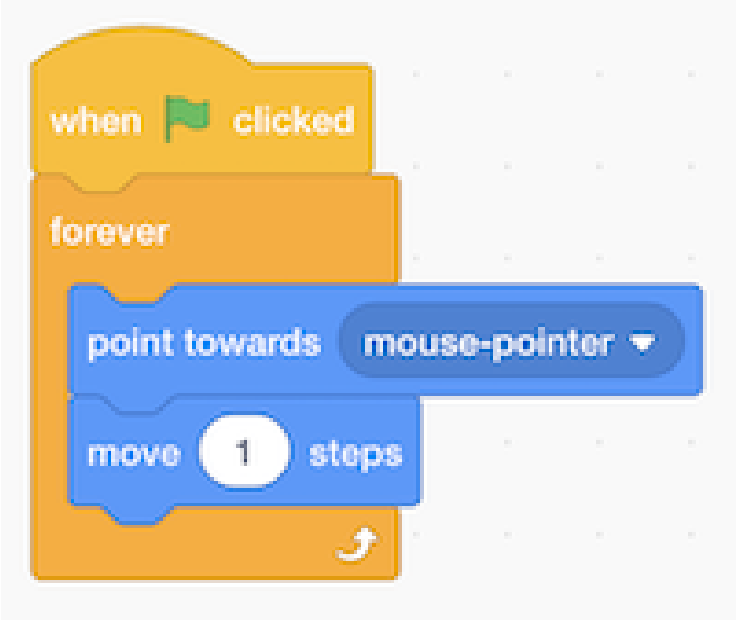
Podemos fazer ele dizer "miau" três vezes, mas agora estamos repetindo blocos várias vezes.

Vamos usar um loop, ou um bloco "repeat":

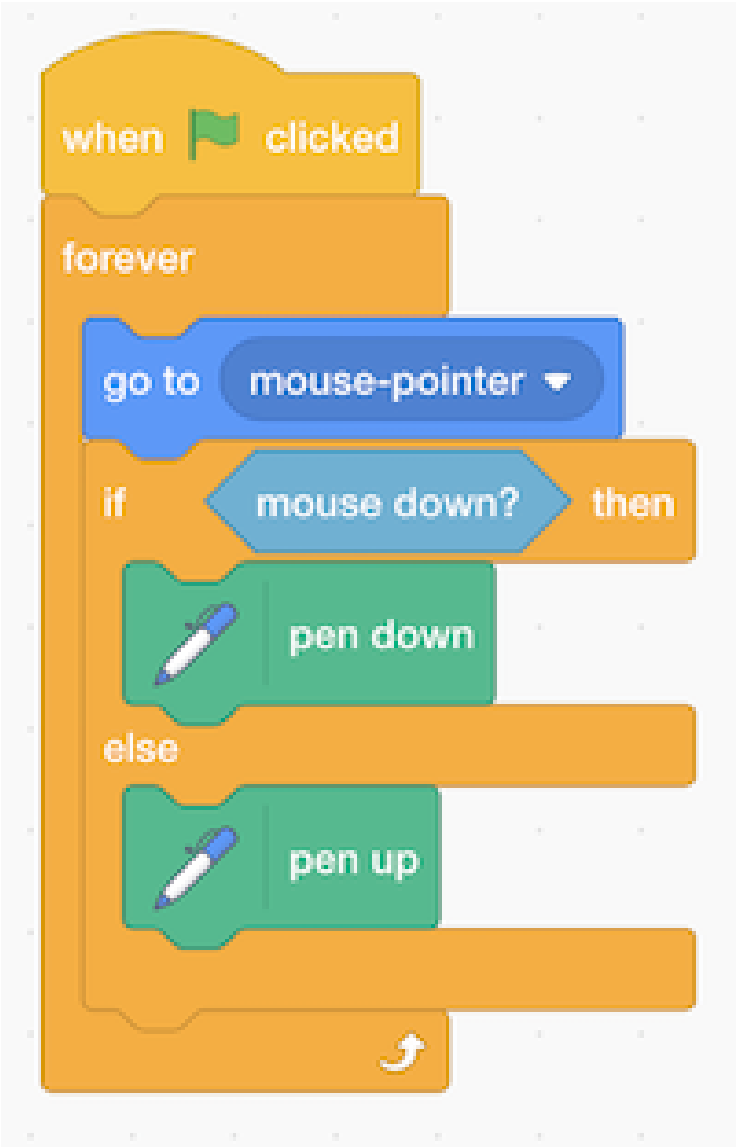


Agora nosso programa alcança os mesmos resultados, mas com menos blocos. Podemos considerá-lo com um design melhor: se houver algo que quisermos mudar, precisaríamos mudar em apenas um lugar em vez de três.

Podemos fazer o gato apontar para o mouse e mover-se em sua direção:



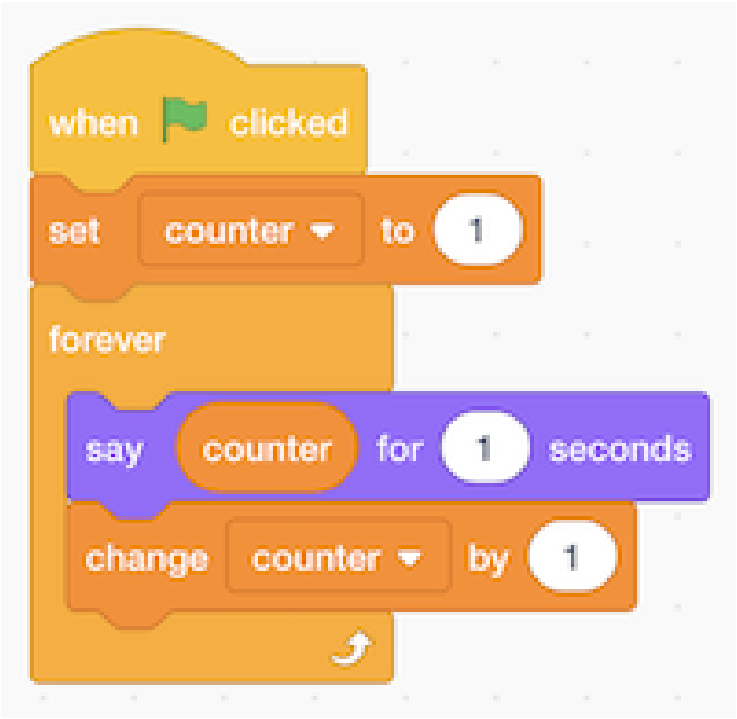
Experimentamos a extensão Caneta, usando o bloco "pen down" com uma condição:



Aqui, movemos o gato para o ponteiro do mouse, e se o mouse estiver clicado, ou pressionado, colocamos a "pen down" (caneta para baixo), o que desenha. Caso contrário, colocamos a caneta para cima. Repetimos isso muito rapidamente, várias vezes, então acabamos com o efeito de desenhar sempre que mantemos o mouse pressionado.

O Scratch também tem diferentes fantasias, ou imagens, que podemos usar para nossos personagens.

Vamos fazer um programa que pode contar:



Aqui, **counter (contador)** é uma variável, cujo valor podemos definir, usar e alterar.

Vemos mais alguns programas, como **bounce**, onde o gato se move para frente e para trás na tela para sempre, virando quando estamos na borda da tela.

Podemos melhorar a animação fazendo o gato mudar para uma fantasia diferente a cada 10 passos em **bounce1**. Agora, quando clicamos na bandeira verde para executar nosso programa, vemos o gato alternar o movimento das pernas.

Podemos até gravar nossos próprios sons com o microfone do nosso computador e reproduzi-los em nosso programa.

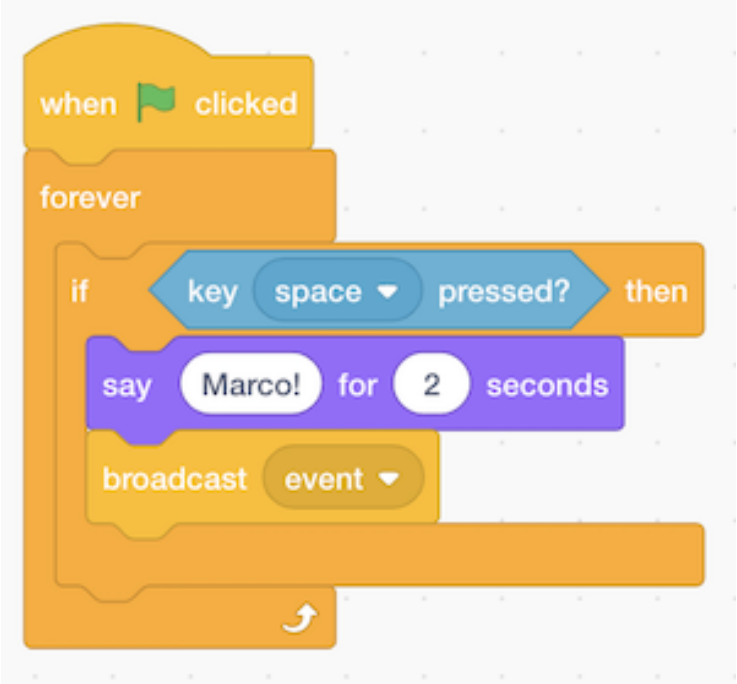
Para construir programas cada vez mais complexos, começamos com cada um desses recursos mais simples e os sobrepomos.

Podemos também fazer o Scratch miar se o tocarmos com o ponteiro do mouse, em **pet0**.

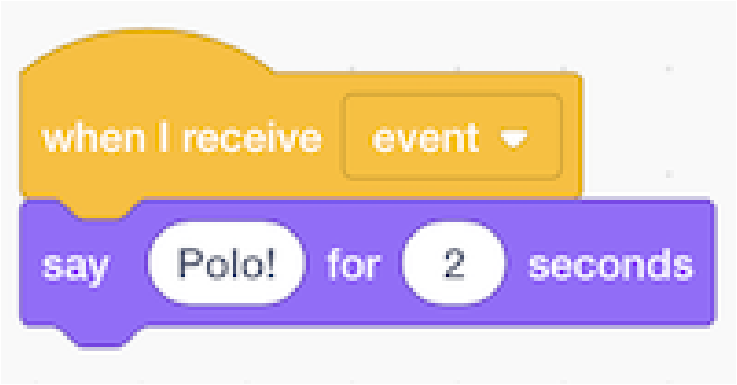
Em **bark**, temos não um, mas dois programas no mesmo projeto Scratch. Ambos esses programas serão executados ao mesmo tempo após a bandeira verde ser clicada. Um deles reproduzirá o som de um leão-marinho se a variável **muted** estiver definida como falsa, e o outro definirá a variável **muted** de verdadeiro para falso, ou de falso para verdadeiro, se a tecla de espaço for pressionada.

Outra extensão observa o vídeo capturado pela webcam do nosso computador e reproduz o som de miado se o vídeo tiver movimento acima de algum limite.

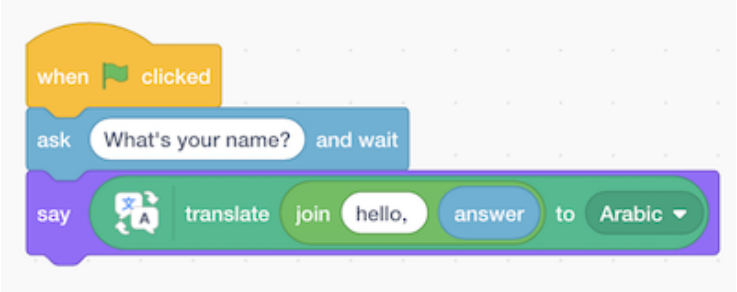
Com vários sprites, ou personagens, podemos ter diferentes conjuntos de blocos para cada um deles:



Para um fantoche, temos esses blocos que dizem "Marco!", e depois um bloco "broadcast event" (transmitir evento). Este "event" é usado para que nossos dois sprites se comuniquem entre si, como enviar uma mensagem nos bastidores. Então, nosso outro fantoche pode simplesmente esperar por esse "event" para dizer "Polo!":

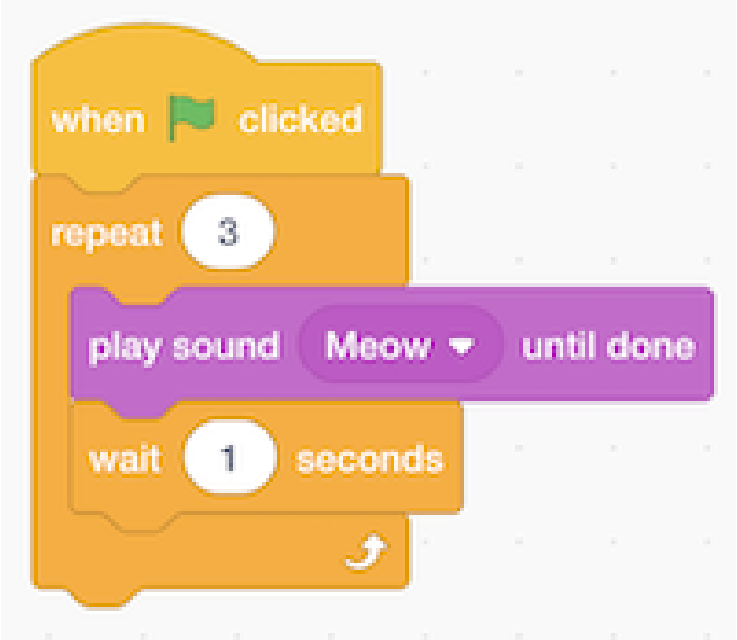


Podemos usar a extensão de **Tradução** para dizer algo em outros idiomas também:



Aqui, a saída do bloco "join" é usada como a entrada para o bloco "translate", cuja saída é passada como entrada para o bloco "say".

Agora que sabemos alguns conceitos básicos, podemos pensar no design, ou qualidade, dos nossos programas. Por exemplo, podemos querer que o gato mia três vezes com o bloco "repeat":



Podemos usar abstração, que simplifica um conceito mais complexo. Nesse caso, podemos definir nosso próprio bloco "miar" no Scratch e reutilizá-lo em outros lugares em nosso programa, como visto em **meow3**. A vantagem é que não precisamos saber como o miado é implementado, ou escrito em código, mas sim usá-lo em nosso programa, tornando-o mais legível.

Podemos até definir um bloco com uma entrada em **meow4**, onde temos um bloco que faz o gato miar um certo número de vezes. Agora podemos reutilizar esse bloco em nosso programa para miar qualquer número de vezes, assim como podemos usar os blocos "translate" ou "say", sem saber os detalhes da implementação, ou como o bloco realmente funciona.

Damos uma olhada em mais alguns exemplos, incluindo **Gingerbread tales remix e Oscartime**, ambos combinando loops, condições e movimento para fazer um jogo interativo.

Oscartime foi feito por David há muitos anos, e ele começou adicionando um sprite, depois um recurso por vez, e assim por diante, até que se somaram ao programa mais complicado.

Um ex-aluno, Andrew, criou Raining Men. Mesmo que Andrew, no final, não tenha seguido a ciência da computação como profissão, as habilidades de resolução de problemas, algoritmos e ideias que aprenderemos no curso são aplicáveis em qualquer lugar.