**O que é pensamento computacional?**

**É a habilidade de mobilizar os conhecimentos para solucionar os problemas com eficiência, a partir do**[**uso da tecnologia**](https://blog.conexia.com.br/rendimento-escolar/). Para isso, é preciso dominar os fundamentos da computação, bem como explorá-los de forma crítica, criativa e estratégica.

Assim, o pensamento computacional permite encontrar soluções não apenas no que se refere à apropriação da tecnologia, mas também em relação às demais áreas do conhecimento.

Mais que uma aposta para o futuro, a cultura digital está entre as 10 competências gerais da Base Nacional Comum Curricular ([BNCC](http://basenacionalcomum.mec.gov.br/a-base)). Com isso, fica evidente como a tecnologia e os recursos digitais têm papel necessário nas escolas.

O pensamento computacional tem alguns pilares. São eles:

* **decomposição**— se refere à habilidade de decompor um problema grande em partes menores, o que facilita a sua compreensão e resolução;
* **abstração**— é a capacidade de encontrar o que é mais relevante em uma situação, filtrando o que pode ser deixado de lado;
* **reconhecimento de padrões**— o pensamento computacional, em grande medida, funciona por padrões. Assim, é importante reconhecer repetições e similaridades para buscar soluções;
* **pensamento algorítmico**— está ligado ao uso da lógica e da racionalidade para criar soluções, a partir da sistematização.

**Qual a importância do pensamento computacional?**

Como vimos, a tecnologia é uma realidade na sociedade contemporânea e [a escola não pode ficar de fora](https://blog.conexia.com.br/transformacao-na-educacao/) dessa nova característica. Isso porque ajuda a desenvolver habilidades essenciais, além de ser um recurso importante para captar mais a atenção dos estudantes.

A seguir, conheça alguns dos benefícios que o pensamento computacional proporciona.

**Autonomia**

Aulas mais ativas é a característica de uma proposta de educação que encoraja o pensamento computacional. Efetivamente, quando as crianças e os adolescentes têm contato com atividades mais práticas, deixam de apenas receber conhecimento e passam a explorar ativamente suas possibilidades.

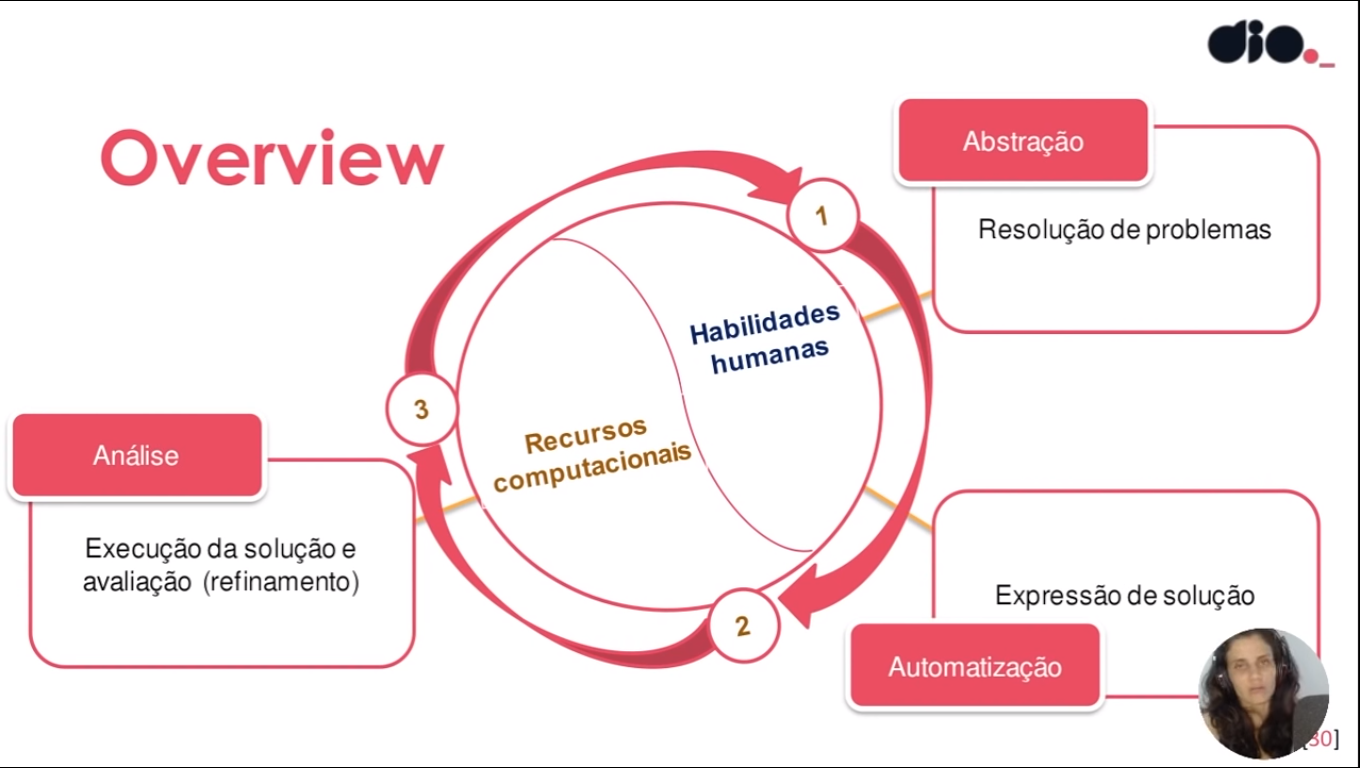
Isso contribui para uma postura mais autônoma, incentivando os [alunos a serem protagonistas](https://blog.conexia.com.br/aluno-protagonista/) em seu processo de busca pelo conhecimento.

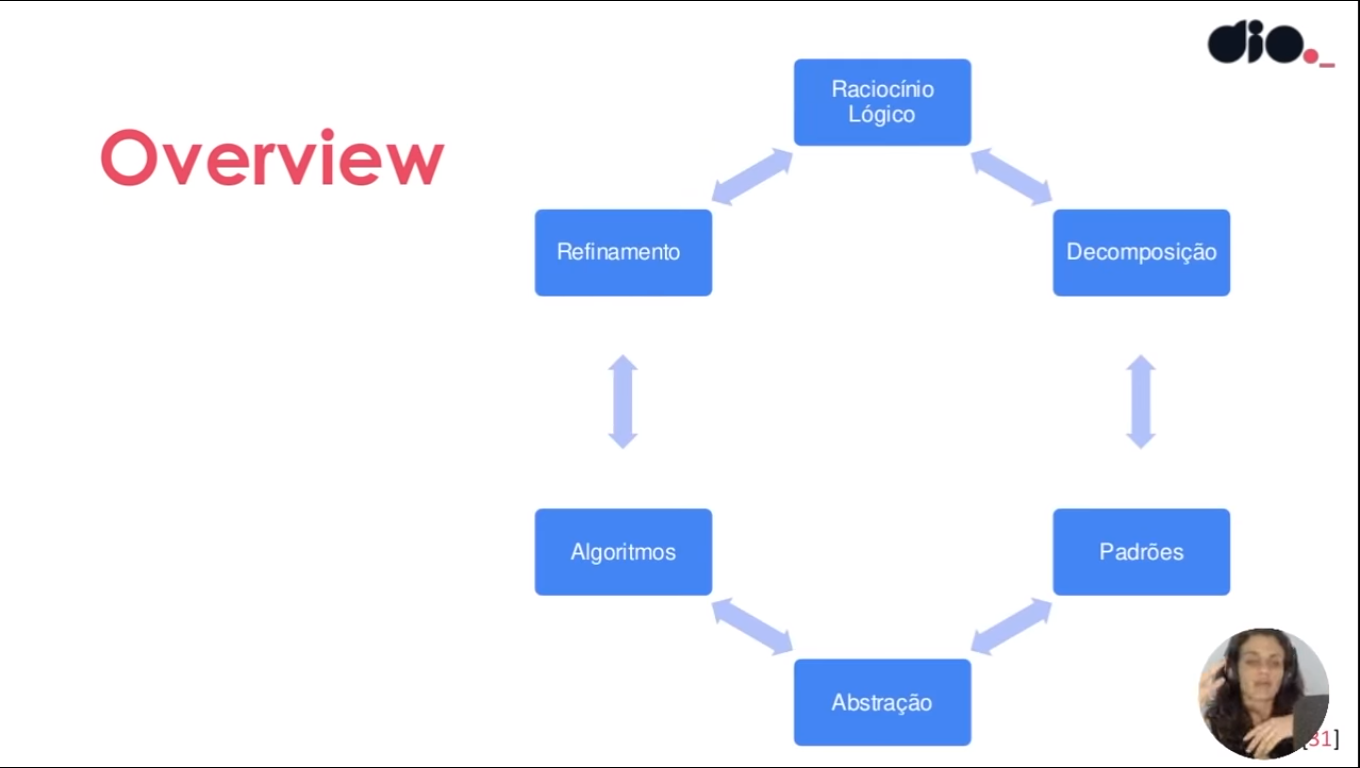
**Criatividade**

Como você pôde perceber, nesse contexto de aula, o estudante é convidado a buscar soluções para problemas de diversas áreas. Assim, precisa mobilizar sua criatividade para encontrar os melhores caminhos, o que representa um ótimo estímulo.

**Interdisciplinaridade**

O pensamento computacional pode ser explorado por diversas matérias e é muito saudável que os professores busquem a convergência dos conhecimentos. Os recursos tecnológicos beneficiam uma proposta interdisciplinar e ajudam a aprofundar a aprendizagem.





**Estrutura Condicional Simples**

A Estrutura Condicional Simples executa um comando ou vários comandos se a condição for verdadeira. Se a condição for falsa, a estrutura é finalizada sem executar os comandos. O comando que define a estrutura é representado pela palavra SE.

**Pseudocódigo:**

1. INICIO ALGORITMO

2. SE condição ENTAO

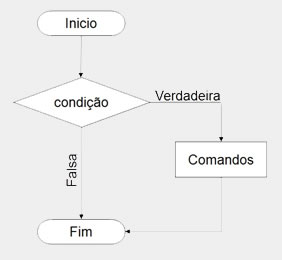
3.       comando

4. FIM SE

5. FIM ALGORITMO

*O comando somente será executado se a condição for verdadeira*

**Fluxograma:**

[](http://fabrica.ms.senac.br/wp-content/uploads/2013/06/imagem2.jpg)

**Estrutura Condicional Composta**

A Estrutura Condicional Composta segue o mesmo princípio da Estrutura Condicional Simples, com a diferença de que quando a condição não é satisfeita, será executado o outro comando. O comando que define a estrutura é representado pelas palavras SE e SE NÃO.

**Pseudocódigo:**

1. INICIO ALGORITMO

2. SE condição ENTAO

3.      comando  A

4. SENAO

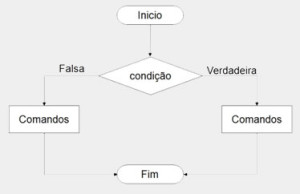
5.     comando B

6. FIM SE

7. FIM ALGORITMO

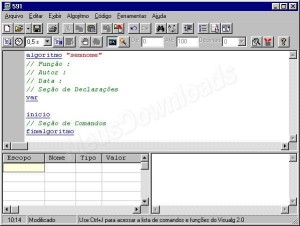
*Se a condição for verdadeira, então o comando A será executado. Caso ela seja falsa, apenas o comando B será executado.*

Fluxograma:

[](http://fabrica.ms.senac.br/wp-content/uploads/2013/06/imagem3.jpg)

**Ferramentas:**

Uma dica para quem está começando no mundo dos algoritmos é utilizar uma boa ferramenta para auxiliar no seu aprendizado. Eu indico o [Visualg](http://pt.wikipedia.org/wiki/Visualg), que é uma ferramenta fácil de ser utilizada por ser em português e ela permite que você execute seus programas.

[](http://fabrica.ms.senac.br/wp-content/uploads/2013/06/imagem4.jpg)

**Conclusão:**

As Estruturas Condicionais são muito utilizadas no dia-a-dia de nós programadores, ela vem nos auxiliar na tomada de decisões (por exemplo: Fazer login de usuário, se o login e senha estiverem corretos, é liberado o acesso se não, é emitido  uma mensagem de erro). Por isso, é de extrema importância dominarmos este assunto.

# O que são Vetores e Matrizes (arrays)

**Vetores** e **Matrizes** são estruturas de dados muito simples que podem nos ajudar muito quando temos muitas variáveis do mesmo tipo em um algoritmo. Imagine o seguinte problema: Você precisa criar um algoritmo que lê o nome e as 4 notas de 50 alunos, calcular a média de cada aluno e informar quais foram aprovados e quais foram reprovados.

Conseguiu imaginar quantas variáveis você vai precisar? Muitas né? Vamos fazer uma conta rápida: 50 variáveis para armazenar os nomes dos alunos, (4 \* 50 = ) 200 variáveis para armazenar as 4 notas de cada aluno e por fim, 50 variáveis para armazenar as médias de cada aluno. 300 variáveis no total, sem contar a quantidade de linhas de código que você vai precisar para ler todos os dados do usuário, calcular as médias e apresentar os resultados. Mas eu tenho uma boa notícia pra você. Nós não precisamos criar 300 variáveis! Podemos utilizar **Vetores** e **Matrizes** (também conhecidos como **ARRAYs**)!

## O QUE SÃO VETORES E MATRIZES?

**Vetor** (**array** uni-dimensional) é uma variável que armazena várias variáveis do mesmo tipo. No problema apresentado anteriormente, nós podemos utilizar um vetor de 50 posições para armazenar os nomes dos 50 alunos.

**Matriz** (**array** multi-dimensional) é um **vetor** de **vetores**. No nosso problema, imagine uma matriz para armazenar as 4 notas de cada um dos 50 alunos. Ou seja, um vetor de 50 posições, e em cada posição do vetor, há outro vetor com 4 posições. Isso é uma matriz.

Cada item do vetor (ou matriz) é acessado por um número chamado de **índice**.

Vamos representar os vetores e matrizes graficamente para facilitar o entendimento do conceito.

vetor e matriz
(array)

Podemos ver na imagem acima que cada posição do vetor é identificado por um número (chamado de índice), no caso da matriz são dois números (um na vertical e um na horizontal).

## VETORES E MATRIZES NA PRÁTICA!

Tomando o exemplo apresentado no início deste artigo, vamos implementar o algoritmo utilizando o [**Visualg**](http://www.dicasdeprogramacao.com.br/linguagem-de-programacao-para-iniciantes/). Nele, vamos usar algumas estruturas básicas já apresentadas aqui, tais como A [**estrutura de repetição PARA**](http://www.dicasdeprogramacao.com.br/estrutura-de-repeticao-para/) e a [**estrutura de decisão SE-ENTÃO-SENÃO**](http://www.dicasdeprogramacao.com.br/estrutura-de-decisao-se-entao-senao/). (Neste algoritmo vamos reduzir o número de alunos de 50 para 5, para facilitar a visualização do resultado.)

**Preste muita atenção no modo como é criado o Vetor e a Matriz e também a forma como é acessada cada posição.**

algoritmo "MediaDe5Alunos"

// Função : Calcular a média das notas de 10 alunos e apresentar quem foi aprovado ou reprovado

// Autor : Gustavo

// Seção de Declarações

var

nomes: vetor [1..5] de caractere

notas: vetor [1..5,1..4] de real

medias: vetor [1..5] de real

contadorLoop1, contadorLoop2: inteiro

inicio

//Leitura dos nomes e as notas de cada aluno

PARA contadorLoop1 DE 1 ATE 5 FACA

ESCREVA("Digite o nome do aluno(a) número ", contadorLoop1, " de 5: ")

LEIA(nomes[contadorLoop1])

PARA contadorLoop2 DE 1 ATE 4 FACA

ESCREVA("Digite a nota ", contadorLoop2, " do aluno(a) ", nomes[contadorLoop1], ": ")

LEIA(notas[contadorLoop1, contadorLoop2])

FIMPARA

//CÁLCULO DAS MÉDIAS

medias[contadorLoop1] := (notas[contadorLoop1, 1] + notas[contadorLoop1, 2] + notas[contadorLoop1, 3] + notas[contadorLoop1, 4]) / 4

FIMPARA

//APRESENTAÇÃO DOS RESULTADOS

PARA contadorLoop1 DE 1 ATE 5 FACA

SE medias[contadorLoop1] >= 6 ENTAO

ESCREVAL("O aluno(a) ", nomes[contadorLoop1], " foi aprovado com as notas (", notas[contadorLoop1, 1], ", ", notas[contadorLoop1, 2], ", ", notas[contadorLoop1, 3], ", ", notas[contadorLoop1, 4], ") e média: ", medias[contadorLoop1])

SENAO

ESCREVAL("O aluno(a) ", nomes[contadorLoop1], " foi reprovado com as notas (", notas[contadorLoop1, 1], ", ", notas[contadorLoop1, 2], ", ", notas[contadorLoop1, 3], ", ", notas[contadorLoop1, 4], ") e média: ", medias[contadorLoop1])

FIMSE

FIMPARA

fimalgoritmo

Repare que os arrays (vetores ou matrizes) aliados a estrutura de repetição PARA é um ótimo recurso para algoritmos que precisam de muitas variáveis do mesmo tipo.

Se você é iniciante em programação este algoritmo pode parecer complexo para você, se houver dúvidas não se acanhe em deixar um comentário aí em baixo que vamos te ajudar. Pra aprender programação (e qualquer outra coisa) o mais importante é a força de vontade!

Um resultado do algoritmo acima pode ser observado a seguir:

*Digite o nome do aluno(a) número 1 de 5: Gustavo*

*Digite a nota 1 do aluno(a) Gustavo: 9*

*Digite a nota 2 do aluno(a) Gustavo: 10*

*Digite a nota 3 do aluno(a) Gustavo: 9,5*

*Digite a nota 4 do aluno(a) Gustavo: 8*

*Digite o nome do aluno(a) número 2 de 5: João*

*Digite a nota 1 do aluno(a) João: 5*

*Digite a nota 2 do aluno(a) João: 6*

*Digite a nota 3 do aluno(a) João: 4,5*

*Digite a nota 4 do aluno(a) João: 7*

*Digite o nome do aluno(a) número 3 de 5: Pedro*

*Digite a nota 1 do aluno(a) Pedro: 7*

*Digite a nota 2 do aluno(a) Pedro: 8,5*

*Digite a nota 3 do aluno(a) Pedro: 6*

*Digite a nota 4 do aluno(a) Pedro: 7*

*Digite o nome do aluno(a) número 4 de 5: Luciana*

*Digite a nota 1 do aluno(a) Luciana: 10*

*Digite a nota 2 do aluno(a) Luciana: 7*

*Digite a nota 3 do aluno(a) Luciana: 7,5*

*Digite a nota 4 do aluno(a) Luciana: 8*

*Digite o nome do aluno(a) número 5 de 5: Augusto*

*Digite a nota 1 do aluno(a) Augusto: 5*

*Digite a nota 2 do aluno(a) Augusto: 5,5*

*Digite a nota 3 do aluno(a) Augusto: 7,5*

*Digite a nota 4 do aluno(a) Augusto: 6*

*O aluno(a) Gustavo foi aprovado com as notas ( 9, 10, 9.5, 8) e média: 9.125*

*O aluno(a) João foi reprovado com as notas ( 5, 6, 4.5, 7) e média: 5.625*

*O aluno(a) Pedro foi aprovado com as notas ( 7, 8.5, 6, 7) e média: 7.125*

*O aluno(a) Luciana foi aprovado com as notas ( 10, 7, 7.5, 8) e média: 8.125*

*O aluno(a) Augusto foi aprovado com as notas ( 5, 5.5, 7.5, 6) e média: 6*

*\*\*\* Fim da execução.\ \*\*\* Feche esta janela para retornar ao Visualg.*

## CONCLUSÃO

Como você pode perceber nesse artigo, Vetores e Matrizes são, na verdade, a mesma coisa: **array** a diferença é que o vetor é um array de apenas 1 dimensão e a matriz é um array de 2 (ou mais) dimensões.

**Array** é uma das estruturas de dados mais simples que existe e uma das mais utilizadas também. Acho que todas as linguagens de programação têm **arrays**, pelo menos ainda não conheço uma linguagem que não tem. Porém, os índices podem mudar dependendo da linguagem, algumas começam os índices do array com 1 e outras com 0, essa é a grande diferença que geralmente encontramos entre linguagens. No caso das linguagens que começam os arrays com o índice 0, o último elemento do array recebe o índice (<tamanho do array> - 1).

Gostou de conhecer os Arrays (Vetores e Matrizes)?

**1. Introdução ao uso de funções**

Se você precisar de um *algoritmo* com um objetivo muito bem definido (e.g. um *algoritmo* para encontrar o menor valor dentro uma *lista* dada) é melhor **encapsulá-lo** em um bloco de modo a poder invocá-lo sempre que precisar desse *algoritmo*. Geralmente todas as linguagens de programação possibilita esse *encapsulamento* e isso é feito na forma de uma função ou de um procedimento (um *procedimentos* é análogo à uma função, entretanto nele **não** existe qualquer devolução). A grande vantagem dessa abordagem é deixar o seu código melhor organizado, além de permitir que a função seja utilizada em outros trechos do código sem a necessidade de codificá-la novamente!

Para ampliar as vantagens do uso de *funções* as *linguagens de programação* permitem o *aninhamento* de funções (funções dentro de funções) e a declaração de variáveis locais.

**3. Por que usar o conceito de função?**

Assim, implementar códigos com objetivos específicos (como computar o *cosseno* de qualquer valor) apresentam três grandes vantagens:

1. Facilita o desenvolvimento (*desenvolvimento modular*): implementa-se uma *particular unidade*, um trecho menor, concentrando-se nele, até que ele esteja funcionando com alto grau de *confiabilidade*;
2. Organização: o código fica melhor organizado e portanto mais fácil de ser mantido;
3. Reaproveitamento: sempre que precisar aplicar o código encapsulado em qualquer outro trecho de código (ou noutro código), pode-se utilizar aquele que já foi implementado e é *confiável*.

**4. Sobre a definição (seus parâmetros) e uso de funções em *Portugol***

Assim, agrupar trechos com objetivos específicos, implementando-os na forma de uma *função* ajuda bastante no desenvolvimento e na organização de códigos em programação. Vejamos isso inicialmente uma pseudo-linguagem de programação, em *Portugol*.

Do ponto de vista prático, a estrutura básica de uma função em uma linguagem de programação está representada abaixo, com a *declaração da função* e sua *lista de parâmetros formais*, seguido de sua invocação (quando providenciamos os *parâmetro efetivos*).

**Entrada e Saída**

Embora um computador possa dar solução a certos problemas muitas vezes mais rápido do que um ser humano, quais problemas do mundo real um **algoritmo** que não pode receber e nem apresentar dados pode resolver?

Por exemplo, é possível para um [**algoritmo**](https://www.devmedia.com.br/o-que-e-algoritmo/40708) informar nome e idade de um usuário, mas quais dados informar? Ele só saberá informar se receber esses dados. E o usuário só saberá que dados informar se o algoritmo pedir. Solucionamos esse impasse por meio da entrada e saída de dados.

A entrada/saída de dados compõem esse processo que consiste em receber os dados do mundo real, por meio do que é digitado ou da ação do mouse, por exemplo. Após isso, mostramos a informação processada ao usuário, o que pode acontecer por meio da tela, uma impressão ou ainda a gravação em um arquivo.

A seguir conheceremos o sistema de input/output de dados e como funciona a lógica, por isso, nesse primeiro momento não se preocupe com a escrita do código, pois dependendo da linguagem a grafia do comando altera.

### Saída dos dados

Os algoritmos precisam interagir com o usuário e para isso podemos utilizar o comando escreva() para imprimir alguma coisa na tela. Esse é um comando muito simples e portanto será o primeiro que aprenderemos nessa mecânica de entrada/saída de dados.

Por exemplo, podemos mostrar uma mensagem na tela utilizando o seguinte comando:

escreva (“Olá mundo!”)

Ao executar o comando, na tela apareceria a mesma mensagem que vemos na **Figura 1**.

Figura 1 Ola mundo

### Entrada de dados

Para que o usuário interaja com o algoritmo, este precisa ler o dado por meio do comando leia.

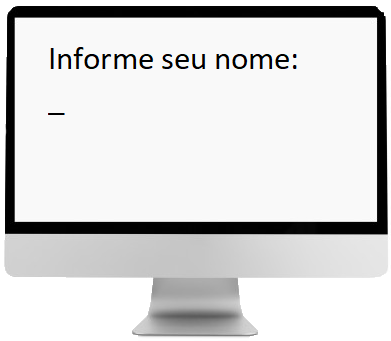
Contudo, assim como quando lemos alguma informação e guardamos ela no nosso cérebro, o computador também guardará o dado lido na memória. Lembrando do conceito de [**variáveis**](https://www.devmedia.com.br/o-que-sao-variaveis/40728), o dado informado será armazenado em uma variável.

Lembre-se que o comando leia sempre guardará o valor em uma variável.

Como exemplo, se quisermos saber o nome do usuário basta utilizar os seguintes comandos e veremos a mesma coisa da **Figura 2**:

escreva (“Informe o seu nome:”)

leia (nome)

**Figura 2**. Execução do comando leia

Repare que na primeira linha temos o comando de saída escreva, que pede ao usuário que informe o seu nome. Sabemos disso por conta dos parênteses ;)

Já na segunda linha temos o comando de entrada leia, que vai pegar o dado informado e armazenar na variável nome. Assim como no comando escreva, tudo o que estiver entre parênteses será armazenado.

Se quisermos saber o nome e o sobrenome do usuário separado por duas variáveis, devemos utilizar os seguintes comandos:

escreva (“Informe o seu nome e sobrenome:”)

leia (nome, sobrenome)

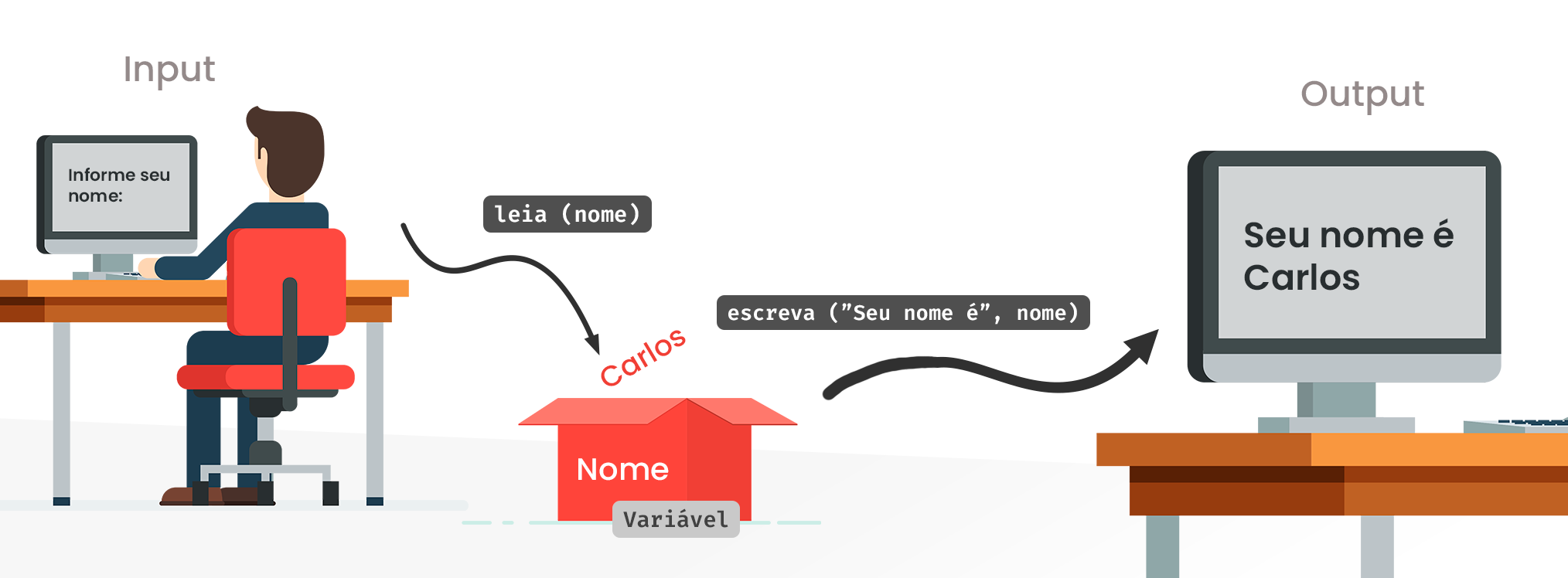
Nesse caso, é possível ler múltiplas variáveis de uma vez, separando-as variáveis por vírgula.

### Retornando dados

O comando escreva também pode retornar o valor de variáveis ao usuário, juntamente com outras informações que queremos passar.

é”, nome)

Na **Figura 3** vemos como funciona o esquema, considerando que o usuário digitaria Carlos no momento da execução.

**Figura 3**. Execução dos comandos leia e escreva

Na linha 3 passamos a mensagem que queremos entre aspas e o algoritmo nos retorna o valor armazenado na variável nome. A sintaxe utilizada com a impressão de texto (entre "") se faz necessária para dar clareza ao que está sendo impresso.

### Exemplo de algoritmo

Com base nos conhecimentos adquiridos até aqui, agora você consegue escrever seu primeiro algoritmo completo! Então, vamos juntos resolver o seguinte problema:

*Uma loja de doces quer saber quais pessoas gostam de sorvete ou não. Para saber isso quer pedir o nome e a opinião.*  
Como criar esse algoritmo?

* **Passo 1**  
  Como primeiro passo para criar nosso algoritmo, precisamos escrever a palavra início. Ela é obrigatória e mostra que o nosso programa começa ali.
* **Passo 2**  
  Precisamos declarar as variáveis: como queremos saber o nome utilizaremos o tipo cadeia. E para saber se gosta ou não de sorvete vamos pedir que digite **Sim** para sim e **Nao** para não: usaremos também o tipo cadeia.

Começaremos assim então:

Lembre-se que como são do mesmo tipo, as variáveis podem ser declaradas numa única linha.

* **Passo 3**  
  Vamos agora pedir os dados ao usuário. Para isso usaremos o comando escreva para informar o que queremos, e vamos ler os dados informados pelo usuário através do comando leia. Para tornar o programa amigável, vamos agradecer a participação retornando a mensagem com o nome informado.

Vamos concluir a estrutura já criada com esses comandos? Acompanhe o algoritmo completo na **Listagem 1**.

**Listagem 1**. Algoritmo completo

Assim como iniciamos com a palavra início, devemos encerrar obrigatoriamente com fim. Dessa forma, conseguimos deixar claro que o nosso programa conseguiu resolver o problema.

### Conclusão

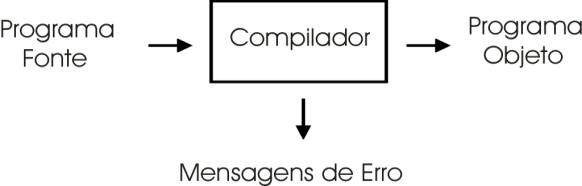
Com o que aprendemos aqui agora podemos escrever algoritmos sem dificuldade.

Aproveite os conhecimentos adquiridos e comece a resolver problemas do seu dia a dia. Até a próxima 😉.

**Analises de código**

Uma breve descrição do funcionamento básico de um compilador será apresentada nesta seção.

Posto de forma simples, um compilador é um programa que lê um programa escrito numa linguagem - a linguagem*-fonte*- e o traduz num programa equivalente numa outra linguagem - a linguagem-*objeto*. Como importante parte desse processo de tradução, o compilador relata a seu usuário a presença de erros no programa-fonte.



### 1.1) O modelo de compilação controlada pela sintaxe

Existem quatro partes básicas na compilação: as análises léxica, sintática, e de contexto, bem como a geração de código. A *análise léxica*serve para reconhecer os elementos sintáticos mais elementares, como por exemplo o número 10 a partir dos dígitos 1 e 0. A*análise sintática* agrupa itens léxicos e elementos sintáticos já reconhecidos formando novos elementos sintáticos, como por exemplo uma expressão aritmética completa, os elementos de um comando de atribuição (lado esquerdo, símbolo de atribuição e o lado direito), etc. A *análise de contexto*associa a certos elementos sintáticos características não sintáticas, como por exemplo o endereço de uma variável na tabela de símbolos do compilador, onde se encontra seu tipo, seu endereço ou codificação no código objeto. A própria construção das tabelas de símbolos do compilador é feita pelo analisador de contexto. A*geração de código*serve para gerar o código objeto.Em geral denomina-se de *análise semântica* o conjunto da análise de contexto e da geração de código.

O analisador sintático pode ser o ativador execução dos outros módulos que fazem as análises léxica e de contexto, bem como a geração de código, ou pode gerar uma estrutura que é empregada pela análise de contexto e pela geração de código. Nesses casos, esse modelo é denominado de *compilação controlada pela sintaxe* (*syntax-oriented compilation*).

Se o código-objeto gerado é um código intermediário, deve-se seguir uma fase de *síntese* do código-objeto definitivo passando-se eventualmente por processos de otimização do código-objeto.

Em geral, o analisador sintático permite a construção de uma *árvore-sintática*, que dá a estrutura sintática detalhada e completa do programa. Cada nó dessa árvore é um elemento sintático, que pode ser *decorado*com os elementos de contexto. Essas árvores sintáticas decoradas permitem o máximo de otimização possível.

### 1.3) A análise léxica

A análise léxica, feitapelo*analisador léxico*(*lexical*scanner) é em geral feita *linearmente*, estritamente da esquerda para a direita. Por exemplo para reconhecer 10 como um número, o analisador léxico reconhece o primeiro dígito, reconhecendo com isso o fato de se iniciar um número, e em seguida reconhece o 0 sem voltar para o 1. Cada elemento reconhecido pelo analisador léxico é denominado *item léxico* (*token*).Na análise léxica do seguinte comando de atribuição, podem ser reconhecidos pelo analisador léxico os seguintes itens léxicos:

montante := depósito\_inicial  \* (1 + taxa\_de\_juros / 100 )

Podem ser agrupados nos seguintes*tokens:*

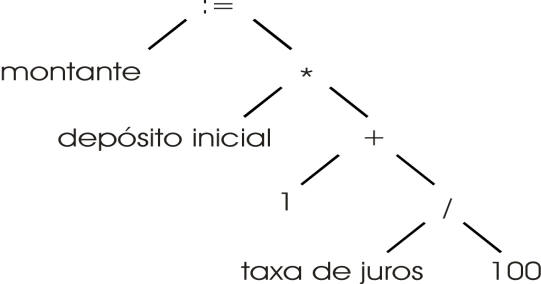
1. O identificador montante  
2. O símbolo de atribuição :=  
3. O identificador depósito\_inicial  
4. O símbolo \*5. O símbolo (  
6. O número 1

etc.

Os espaços em branco são reconhecidos como separadores dos *tokens*e são normalmente eliminados durante a análise léxica

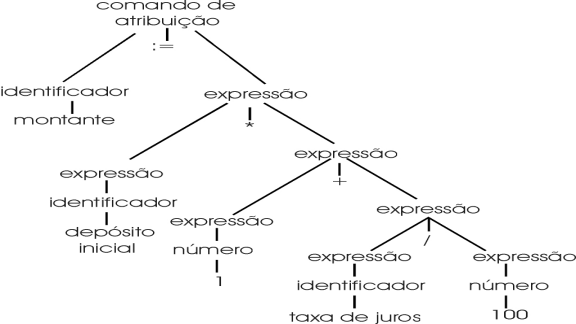
### 1.4) A análise sintática

A ordem da execução dos elementos do comando acima pode ser representada por meio de uma estrutura hierárquica em forma de árvore. No caso, teríamos: 



Nessa figura é mostrado que a divisão é feita antes da soma, tudo que está entre parênteses é agrupado para ser associado à multiplicação etc.

A análise sintática é em geral também uma análise hierárquica, envolvendo o agrupamento dos *tokens* e elementos sintáticos já reconhecidos em novos elementos sintáticos. Esses elementos sintáticos podem ser agrupados sob forma de uma *árvore sintática*, onde aparecem além dos elementos vistos na figura acima, todos os elementos sintáticos. A figura abaixo mostra uma árvore sintática correspondente ao comando visto acima.



A expressão taxa\_de\_juros / 100 é um elemento sintático pois a divisão exige dois argumentos formando um todo com eles.

Em geral não é necessário que o analisador sintático construa uma árvore sintática completa. Por exemplo, reconhecida a expressão com a multiplicação e gerado seu código objeto, não é mais necessário guardar todos os elementos que foram nela reconhecidos, podendo-se eliminá-los da árvore. No entanto, enquanto se analisa o lado direito da atribuição, o trecho da árvore com identificador e := deve ser preservado.

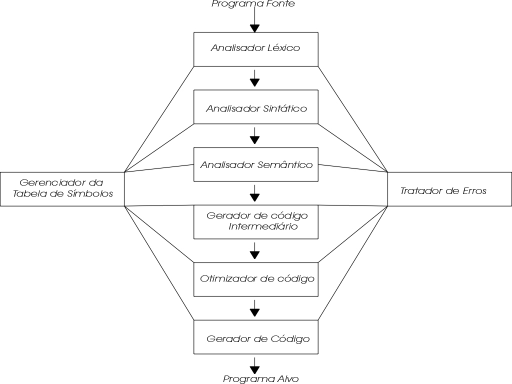
Como dissemos acima os elementos sintáticos de uma árvore sintática podem ser decorados com seu contexto. Por exemplo o identificador do lado esquerdo da atribuição pode ser decorado com o valor do índice apontando para a tabela de símbolos do compilador onde se encontra seu nome (montante), seu tipo, seu endereço no código objeto, etc.

### 1.5) A análise de contexto

A análise de contexto decora os elementos da árvore sintática ou utiliza elementos já decorados. Por exemplo, ao ser reconhecida a expressão que engloba todo o lado direito da atribuição, ela  tem o seu tipo deduzido a partir dos elementos decorados de seu último nível abaixo; antes disso a análise de contexto verifica a compatibilidade dos tipos dos elementos envolvidos naquela expressão.

### 1.6) As fases de um compilador

Um compilador orientado pela sintaxe pode ter uma única *fase*; neste caso todas as análises e geração de código se interligam sendo executadas consecutivamente para cada elemento sintático reconhecido. Uma outra possibilidade é ter-se um compilador de múltiplas fases, gerando-se um código intermediário para cada uma. Cada fase utiliza o código gerado pela fase anterior, transformando-o para a fase seguinte. A figura abaixo mostra um exemplo de múltiplas fases.



Nessa compilação com múltiplas fases, em geral o analisador sintático deve produzir uma árvore sintática

Uma função essencial do compilador é registrar os identificadores usados no programa-fonte e coletar as informações sobre os seus diversos atributos. Uma tabela de símbolos é uma estrutura de dados contendo um registro para cada identificador. A estrutura de dados nos permite encontrar rapidamente cada registro e, igualmente, armazenar ou recuperar dados do mesmo.  
Quando, no programa-fonte, o analisador léxico detecta um identificador, instala-o na tabela de símbolos. No entanto, os atributos do identificador serão inseridos na tabela pelas fases remanescentes.