

C005

# Linguagens de Programação e Compiladores



Prof. MSc. Renzo P. Mesquita  
[renzo@inatel.br](mailto:renzo@inatel.br)

# Bem-vindos ao curso de C005 (Linguagens de Programação e Compiladores)

*BRAINSTORM*



# Bem-vindos ao curso de C005 (Linguagens de Programação e Compiladores)

## *Critérios de Avaliação*

$$NP1 = PV1 * 0.70 + LE1 * 0.30;$$

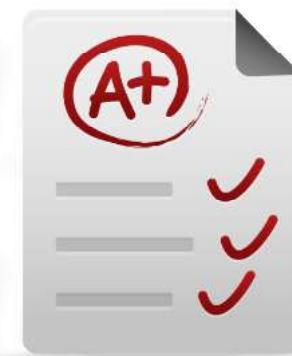
$$NP2 = PV2 * 0.50 + LE2 * 0.2 + SE * 0.3;$$

*Obs: NP1 (Cap.1 e Cap.2) e NP2 (Cap.3 e Seminários).*

$$NPA = (NP1 + NP2)/2;$$

Resumo:

- Duas Provas;
- Listas de exercícios;
- Um Seminário de Apresentação.



# Bem-vindos ao curso de C005 (Linguagens de Programação e Compiladores)

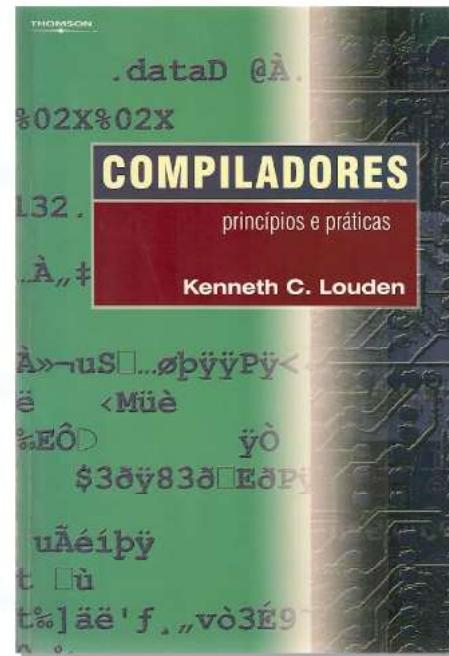
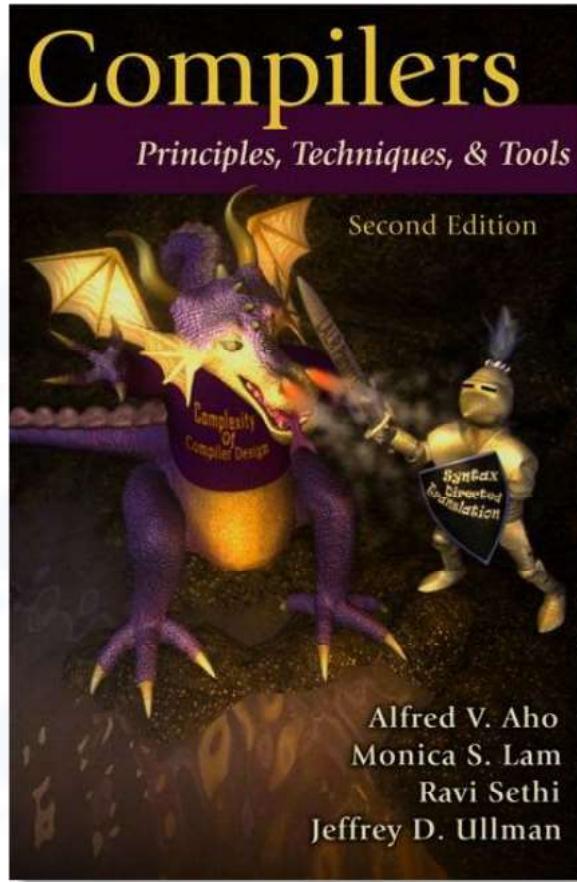
## *Critérios de Aprovação*

```
...
if(NPA >= 60)
{
    escreve("APROVADO!"); NFA=NPA;
}
else if(NPA < 30)
{
    escreve("REPROVADO!"); NFA=NPA;
}
else
{
    calculaNP3(NPA);
}
...
```

```
calculaNP3(int npa)
{
    NFA = (npa + NP3)/2;
    if(NFA >= 50)
    {
        escreve("APROVADO!");
    }
    else
    {
        escreve("REPROVADO!");
    }
}
```

# Bem-vindos ao curso de C005 (Linguagens de Programação e Compiladores)

*Material de Apoio*



# Bem-vindos ao curso de C005 (Linguagens de Programação e Compiladores)

*A Ferramenta Kahoot*

# Kahoot!

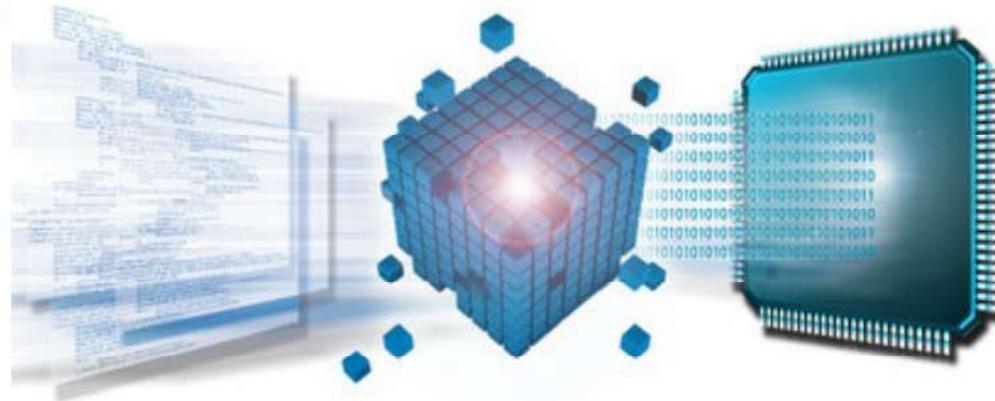


- Quiz de Perguntas Online;
- Pode ser jogado em qualquer dispositivo conectado à Internet;
- Vamos utilizá-lo para revisarmos pontos chaves em algumas aulas;
- VAMOS TESTAR?

**<https://kahoot.it/>**

**C005**  
**Linguagens de Programação e  
Compiladores**

Cap.1 - Introdução às Linguagens de Programação  
e Compiladores



Prof. MSc. Renzo P. Mesquita  
[renzo@inatel.br](mailto:renzo@inatel.br)

# Objetivos

- Discutir conceitos gerais e importantes sobre as linguagens de programação;
- Apresentar as diferentes formas de tradutores de Linguagens de Programação;
- Apresentar uma visão geral da estrutura de um compilador típico;



# **1. Introdução às Linguagens de Programação e Compiladores**

- 1.1. Evolução das Linguagens de Programação;*
- 1.2. Razões para estudar Linguagens de Programação;*
- 1.3. Propriedades Desejáveis das LP's;*
- 1.4. Paradigmas das Linguagens de Programação;*
- 1.5. Linguagens de Domínio Específico;*
- 1.6. Compiladores e Interpretadores;*
- 1.7. Aplicações da construção de Compiladores;*

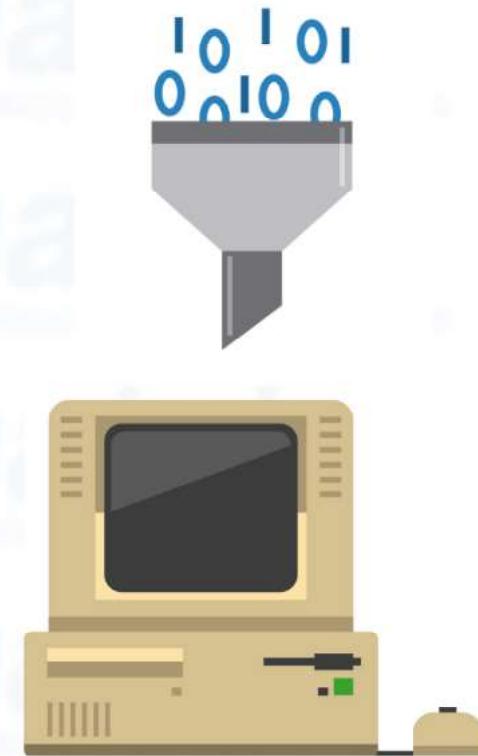


## 1.1 Evolução das Linguagens de Programação

Os primeiros computadores eletrônicos apareceram na década de 40 e eram programados em linguagem de máquina, diretamente por sequências de 0s e 1s. Nem é preciso fazer força para entender como esse tipo de programação era lento, cansativo e passível de erros.

*Como criar formas de programação mais inteligíveis às pessoas a fim de melhorar a produtividade na criação de softwares?*

Criando as Linguagens de mais ALTO NÍVEL;



# 1.1 Evolução das Linguagens de Programação

Atualmente, existem centenas de Linguagens de Programação. Elas podem ser classificadas de diversas maneiras, e uma delas, se diz respeito às Gerações:

## 1<sup>a</sup> Geração

São as linguagens de máquina. Ex: 0 e 1;

## 2<sup>a</sup> Geração

São as linguagens de montagem, conhecidas como Assembly;  
Ex: Liguagem de um Processador específico;

## 3<sup>a</sup> Geração

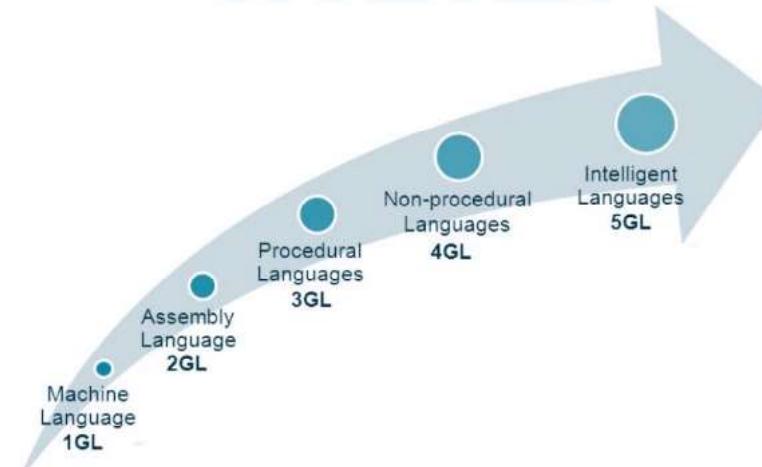
São as linguagens de Alto Nível procedurais, ou seja, que seguem uma sequência lógica (How to do). Ex: C++, Java, Python etc;

## 4<sup>a</sup> Geração

São as linguagens para aplicações específicas não procedurais (What to do).  
Ex: SQL, Cypher, MQL etc;

## 5<sup>a</sup> Geração

São linguagens baseadas em lógicas matemáticas, e que ajudam a dar suporte na programação de Inteligências Artificiais. Ex: Prolog, Mercury etc.



## 1.2. Razões para estudar Linguagens de Programação

1. Capacidade de resolver soluções computacionais para problemas;
2. Maior habilidade para escolhermos Linguagens de Programação apropriadas para casos específicos;
3. Maior habilidade para aprender novas Linguagens de Programação;
4. Maior capacidade para projetar novas Linguagens de Programação;



## 1.3. Propriedades desejáveis das LP's



*Vamos dar uma rápida olhada em cada uma delas?*

## 1.3. Propriedades desejáveis das LP's

### 1.3.1. Legibilidade

- Facilidade para ler e entender um programa;
- Melhorar a tarefa de manutenção dos programas;

*Fatores que favorecem a legibilidade:*

- Simplicidade;
- Recursos para estruturação de dados e controle;

*Fatores que prejudicam a legibilidade:*

- Uso extensivo de "goto's" - permitem a programação não estruturada;
- Estrutura de dados não adequadas. Ex: if-else sem o begin if e end if;
- Sobrevida de operadores - usar o mesmo símbolo com significados diferentes. Ex: Usar o símbolo '\*' para denotar tanto multiplicação como ponteiros;
- etc.



## 1.3. Propriedades desejáveis das LP's

### 1.3.2. Redigibilidade

- Facilidade para escrever o programa, permitindo ao programador se **concentrar nos algoritmos centrais** do programa sem se preocupar com aspectos não relevantes;

*Fatores que favorecem a redigibilidade:*

- Simplicidade;
- Suporte para **abstração** - Abstração de processo (Ex: Subprograma) e Abstração de dados (Ex: Classes);

*Fatores que prejudicam a redigibilidade:*

- Construções muito complexas;
- **Falta de recurso para abstração**;
- Muitas construções primitivas;



## 1.3. Propriedades desejáveis das LP's

### 1.3.3. Confiabilidade

- Um programa é confiável se ele se comportar de acordo com suas especificações sob todas as condições;

*Fatores que favorecem a Confiabilidade:*

- **Verificação de tipos** - seja em tempo de compilação ou em tempo de execução;
- **Tratamento de exceções** - capacidade do programa interceptar erros durante a execução e tomar medidas corretivas;

*Fatores que prejudicam a Confiabilidade:*

- Permitir ações perigosas - não verificar intervalos de índices de arrays, aritmética de ponteiros, **compatibilidade de tipos**, entre outros.
- Recursos pobres para escritas dos programas;



## 1.3. Propriedades desejáveis das LP's

### 1.3.4. Eficiência

- Está relacionada com o tempo de execução de um programa;
- Algumas aplicações exigem que a execução seja rápida (Ex: Aplicações de Tempo Real);
- Em geral, fatores que melhoram a confiabilidade, abstração e legibilidade dos programas, diminuem a eficiência;
- Linguagens de Programação que requerem verificação de tipos em tempo de execução são menos eficientes.



# 1.3. Propriedades desejáveis das LP's

## 1.3.5. Ortogonalidade

- Capacidade de uma Linguagem de Programação de permitir ao programador combinar seus conceitos básicos sem que se produzam efeitos anômalos nessa combinação;
- Quanto menor o número de exceções, maior a Ortogonalidade;
- O Programador consegue prever com segurança o comportamento de uma determinada combinação de conceitos;

### *Fatores que favorecem a Ortogonalidade:*

- Número pequeno de construções primitivas que podem ser combinadas de forma regular;

### *Fatores que prejudicam a Ortogonalidade:*

- Número alto de exceções às regras de linguagem;
- Operadores que não podem ser aplicados a qualquer tipo de operandos;



## 1.3. Propriedades desejáveis das LP's

### 1.3.6. Reusabilidade

- Propriedade de utilizar o mesmo código para várias aplicações;
- Ligada a recursos de abstração da linguagem: subprogramas com parâmetros, bibliotecas, classes, API's, entre outros;
- Quanto **mais reusável** for um código, **maior** será a **produtividade** de programação.



## 1.3. Propriedades desejáveis das LP's

### 1.3.7. Modificabilidade

- Facilidade de alterar o programa sem implicações em outras partes do mesmo;

*Fatores que favorecem a Modificabilidade:*

- Uso de **constantes simbólicas**. Ex: final (Java), const (C);
- Separação entre interface gráfica e lógica de negócio;
- Tipos abstratos de dados;



## 1.3. Propriedades desejáveis das LP's

### 1.3.8. Portabilidade

- Propriedade dos programas escritos em uma Linguagem de Programação se comportarem da mesma maneira independente do compilador, sistema operacional ou hardware utilizado;

*Fatores que favorecem a Portabilidade:*

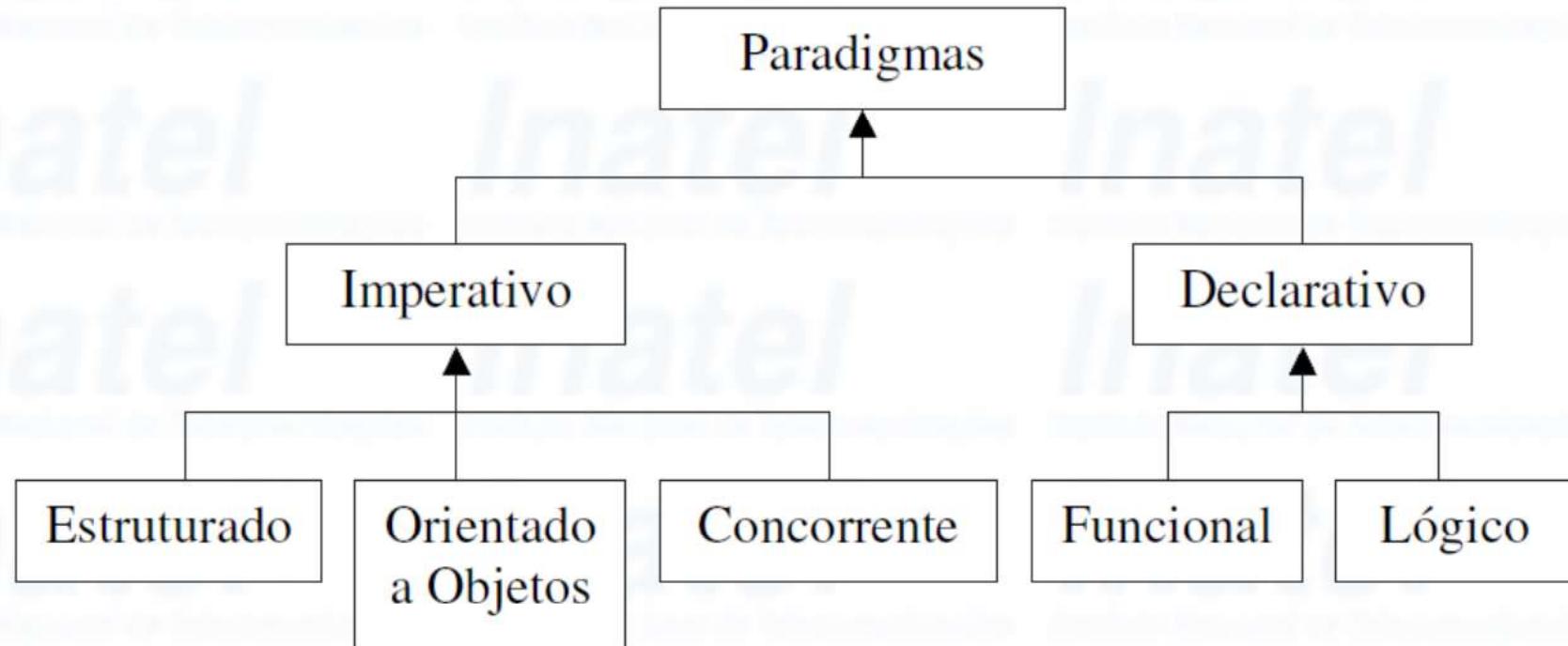
- Implementação híbrida (Ex: Java);
- Padronização da especificação da linguagem desde o seu projeto (Pode prejudicar o desempenho);



# 1.4 Paradigmas das LPs

*Dá-se o nome de Paradigma a um conjunto de características que servem para categorizar um grupo de linguagens.*

Existem diversas classificações, são elas:



*Vamos dar uma rápida olhada em cada uma delas?*

# 1.4. Paradigmas das LPs

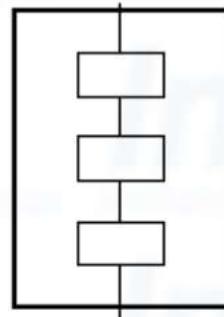
## 1.4.1) Imperativo

- Computação vista como um processo que realiza mudanças de estado;
- Especifica como um processamento deve ser feito para o computador;
- Variáveis podem possuir diferentes valores a cada momento;
- Subdividida nos paradigmas Estruturado, Orientado a Objetos e Concorrente.

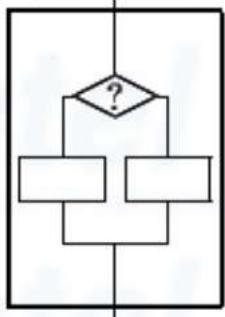
### A) Estruturado

- Facilita a compreensão de programas, comparada à programação com código de máquina;
- Um programa é composto por blocos aninhados de comandos;
- Utiliza de três mecanismos básicos: sequência, seleção e iteração;
- Ex: PASCAL e C são linguagens que adotam o paradigma estruturado.

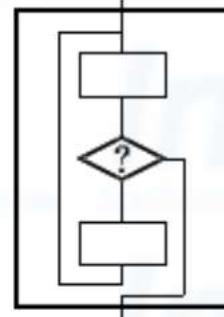
Sequência



Seleção



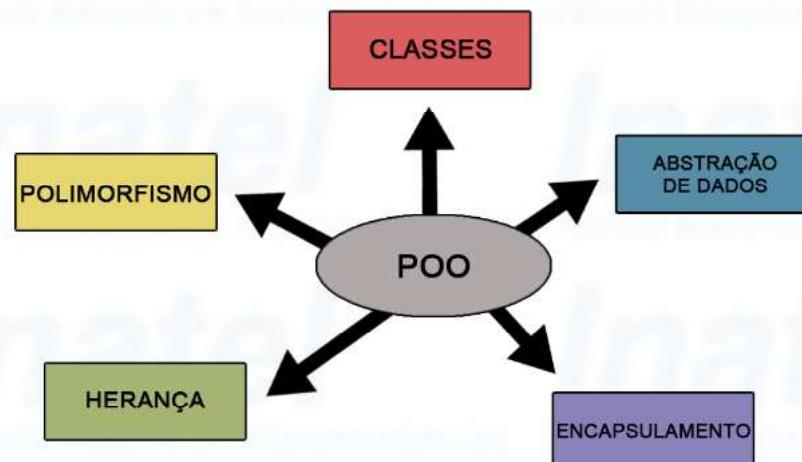
Iteração



# 1.4. Paradigmas das LPs

## B) Orientado a Objetos

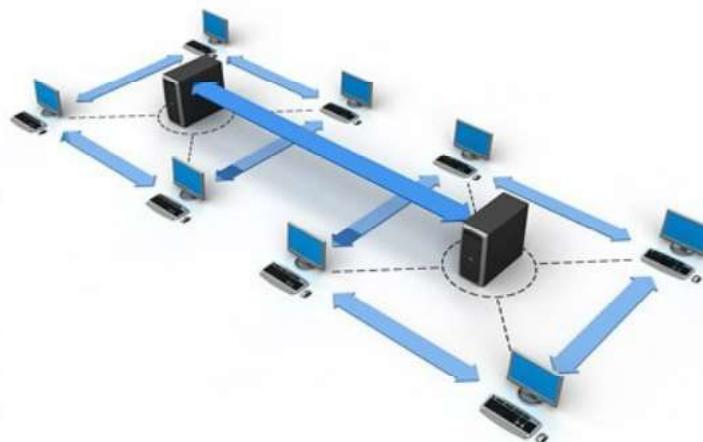
- Paradigma visto como uma evolução do paradigma estruturado;
- Oferece conceitos que torna mais rápido e confiável o processo de desenvolvimento de software;
- Enfoque nas **abstrações de dados**, ao invés de enfoque no controle de execução dos programas (paradigma estruturado);
- **Classes** são abstrações que definem uma estrutura de dados e **Objetos** são instâncias de classes;
- Ex: SMALLTALK, C++, JAVA, entre outras.



## 1.4. Paradigmas das LPs

### C) Concorrente

- Oferece a possibilidade de **vários processos executando simultaneamente** (suporte a Threads) concorrendo por recursos;
- Pode utilizar uma ou várias unidades de processamento (Em um mesmo computador ou distribuídas geograficamente);
- Ex: ADA, JAVA, C# entre outros.



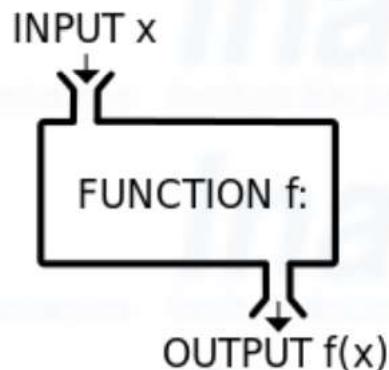
# 1.4. Paradigmas das LPs

## 1.4.2) Declarativo

- Linguagens que especificam diretamente ao computador **O QUE** deve ser feito;
- **O programador descreve o problema** a ser resolvido e esta descrição é usada para encontrar uma ou mais soluções ao problema automaticamente;
- Subdividida nos paradigmas Funcional e Lógico.

### A) Funcional

- Operam **apenas sobre funções definidas**, as quais recebem uma lista de valores e retornam um valor;
- Trabalha com composição de funções e chamada recursiva de funções.
- Ex: LISP, HASSELL, entre outras.



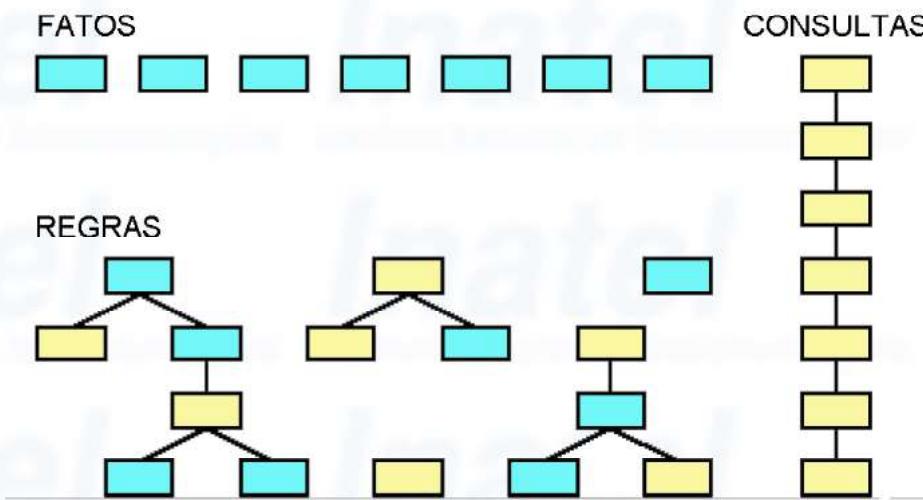
### Ex: Linguagem LISP

(plus 6 9) ;soma 6 + 9  
(plus (plus 3 4) (plus 6 7)) ; soma 3+4 com 6+7

# 1.4. Paradigmas das LPs

## B) Lógico

- Faz uso da lógica matemática;
- Geralmente utilizam de uma base de conhecimento (Fatos);
- Empregada em problemas que possam ser resolvidos por dedução ou inferência;
- Muito utilizada na área de Inteligência Artificial.
- Ex: PROLOG.



*Ex: Linguagem PROLOG*

```
gosta(joão,flores).  
gosta(joão,maria).  
gosta(paulo,maria)
```

```
?- gosta(joão,X)
```

## 1.4. Paradigmas das LPs

### 1.4.3) Linguagens de Programação Multiparadigma

As vezes um só paradigma não é capaz de resolver todos os problemas, o que fazer neste caso? Usar uma linguagem Multiparadigma.

- É uma linguagem que suporta mais de um paradigma de programação;
- Geralmente utilizam de frameworks para permitir o uso de diversos paradigmas de forma mais clara.
- Ex: SCALA.



## 1.5. Linguagens de Domínio Específico

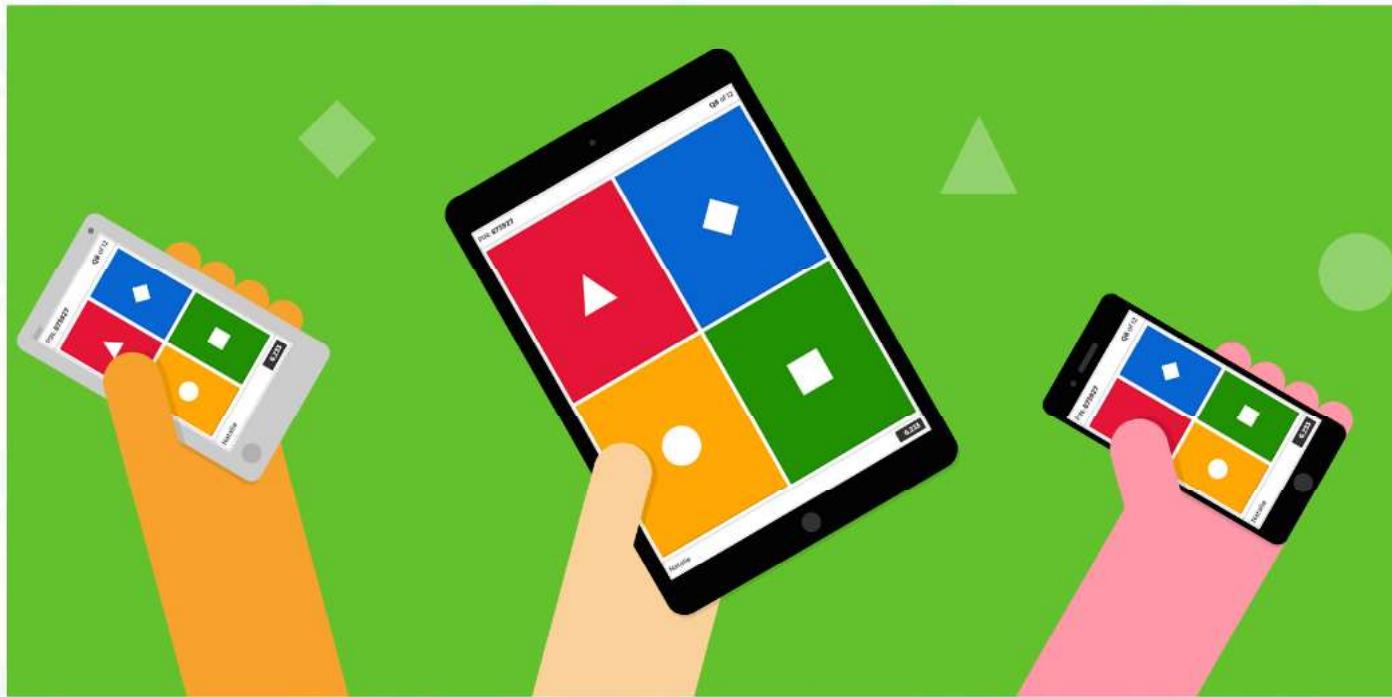
Os paradigmas e linguagens citadas anteriormente são denominadas **GPL (general-purpose languages)**, ou linguagens de uso geral;

*E o que são Linguagens de Domínio Específico?*

- Também chamada de **DSL (domain-specific language)** é um tipo de linguagem dedicada e especializada para um domínio de aplicação específica;
- Ex: HTML (Criação de páginas), VHDL (Design de Hardware), SQL (Definição e manipulação de banco de dados), entre outras.
- DSL's muito simples, usadas geralmente para um único tipo específico de aplicação, são denominadas **mini-linguagens**;

HORA DO

# Kahoot!



ACESSE: <https://kahoot.it>

## 1.6. Compiladores e Interpretadores

*Os programas escritos nas diversas linguagens de programação de alto nível precisam ser traduzidos para uma linguagem de mais baixo nível (código de máquina) para que possam ser executados pelo computador.*

Existem dois métodos básicos utilizados na tradução de códigos escritos em linguagem de alto nível para linguagem de máquina:

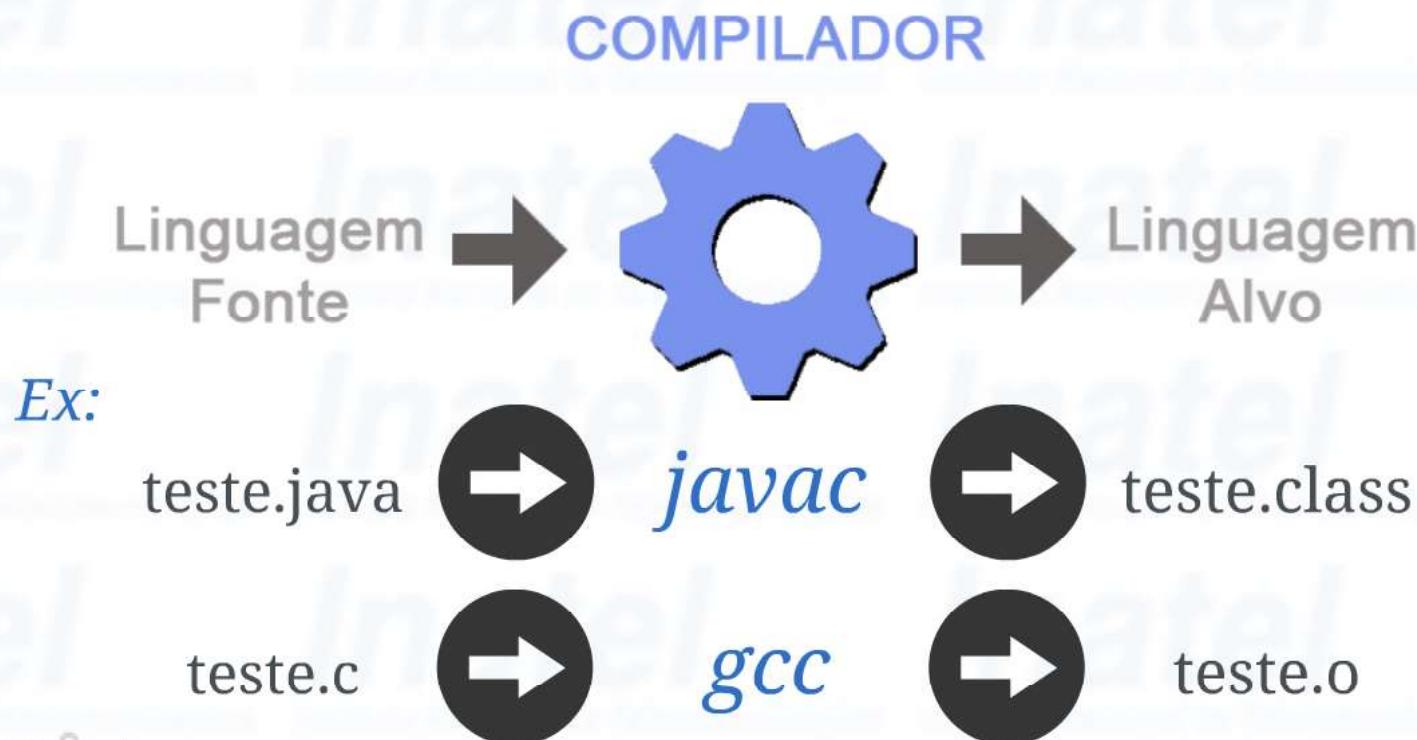
- 1) Compiladores;
- 2) Interpretadores;



# 1.6. Compiladores e Interpretadores

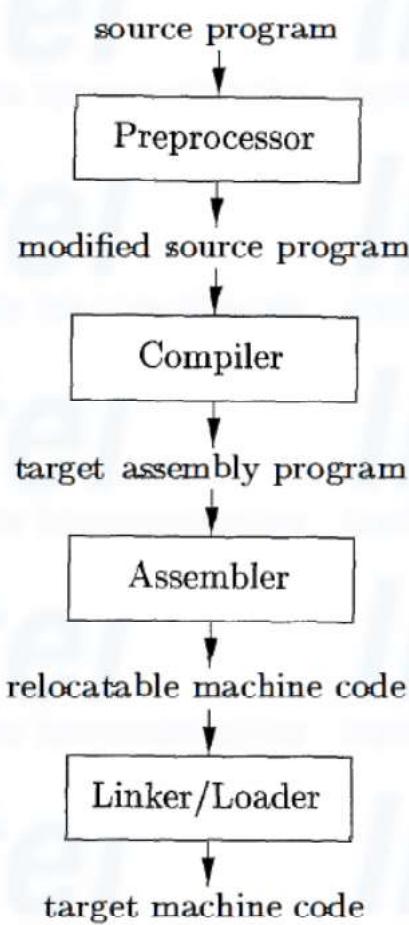
## 1.6.1 Compiladores

Compilador é um programa que recebe como entrada um programa em uma linguagem de programação (*Source Language*) e o traduz em um programa equivalente em outra linguagem, denominado programa objeto (*Target Language*).



# 1.6. Compiladores e Interpretadores

Além de um compilador, vários outros programas podem ser necessários para criação de um programa objeto executável.



## A) Pré-Processador (Preprocessor)

Programa separado ativado pelo compilador antes do início da tradução para realização de operações simples. Algumas funções: Apagar comentários, incluir outros arquivos (Ex: #include na linguagem C), executar substituições de macros etc.

## C) Montador (Assembler)

Após o compilador receber o programa modificado enviado pelo pré-processador e gerar uma linguagem assembly (linguagem objeto), um programa chamado Montador (Assembler) produz um código de máquina relocável.

## D) Linker/Loader

Programas grandes normalmente são compilados em partes. O Linker combina um ou mais arquivos objetos gerados pelo compilador em um arquivo executável. O Loader reúne os arquivos executáveis na memória para execução.

# 1.6. Compiladores e Interpretadores

## *Bibliotecas (Libraries)*

Arquivos auxiliares que possuem diversas funções específicas. Podem ser de dois tipos:

### *1) Bibliotecas Estáticas*

São inseridas antes de utilizar um Compilador específico. Ex:  
Arquivos de extensão .h no C, import de jars no Java etc.

### *2) Bibliotecas Dinâmicas*

São inseridas em uma das fases do Compilador. Ex: arquivos de extensão .lib, .dll (dynamic linking library) etc.

(Observação: As Bibliotecas dinâmicas são dependentes dos sistemas operacionais e as estáticas podem ser compiladas juntamente com o código do programa)



# 1.6. Compiladores e Interpretadores

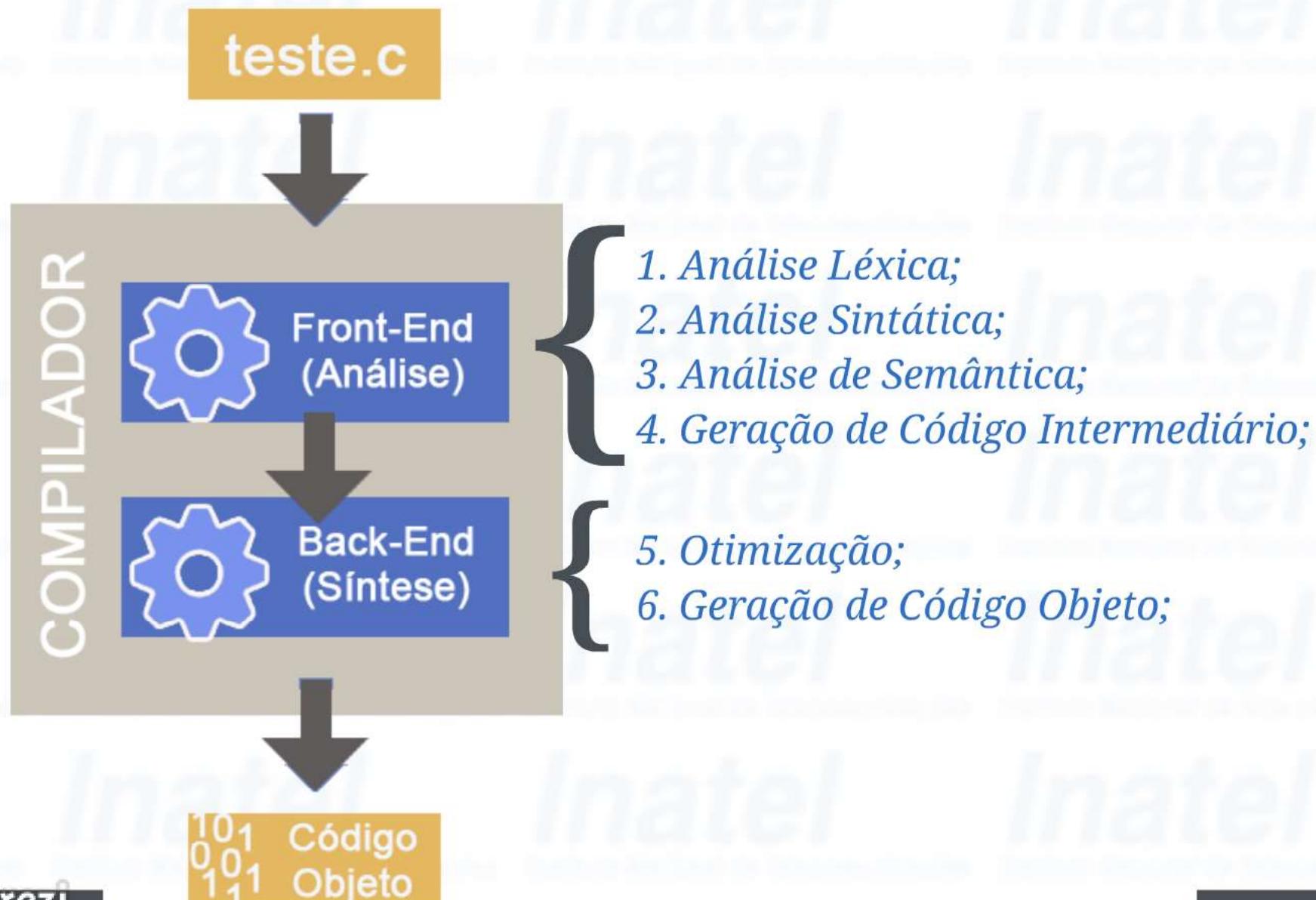
## 1.6.1.1 Fases de um Compilador

Até agora, tratamos o Compilador como uma "caixa preta" que mapeia um programa fonte para um programa objeto. O Compilador internamente é subdividido em módulos, onde cada módulo possui uma função específica. São eles:

1. *Análise Léxica, Leitura ou Scanning (Lexemas)*
2. *Análise Sintática (Árvore Sintática)*
3. *Análise Semântica (Tipos Corretos)*
4. *Geração de Código Intermediário (Código Intermediário)*
5. *Otimização (Otimização de Código)*
6. *Geração de Código Alvo (Linguagem Alvo)*

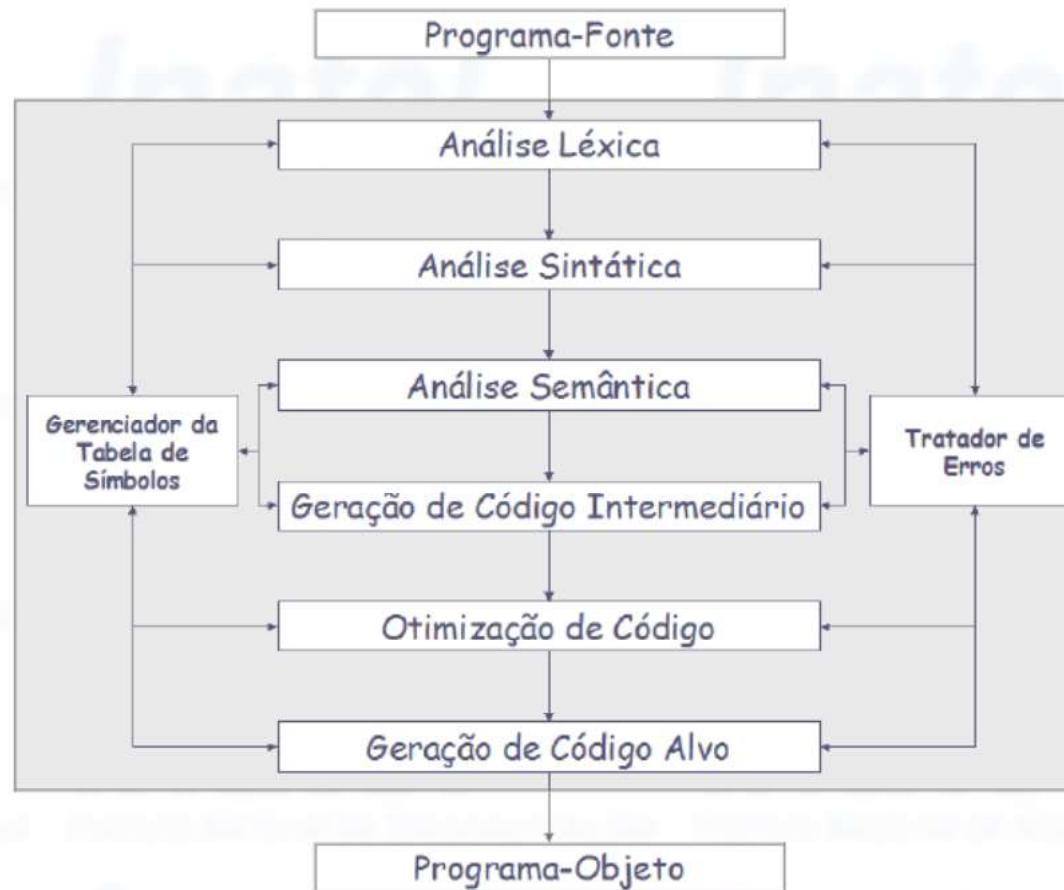
*Como cada um destes módulos se relacionam?*

## 1.6. Compiladores e Interpretadores



# 1.6. Compiladores e Interpretadores

## *Fases de um Compilador*



(Observação: O Gerenciador da Tabela de Símbolos e o Tratador de Erros são submódulos comuns a todas as fases de compilação)

# 1.6. Compiladores e Interpretadores

## *Tabela de Símbolos*

Estrutura de dados responsável por **armazenar informações de todos os identificadores (variáveis e palavras-chave)** durante a compilação.

Informações que podem ser armazenadas em uma tabela de dados:

- Tipos de dados;
- Tamanho dos tipos de dados;
- Escopos;
- Entre outras informações relevantes.

Como a tabela de símbolos é acessada com bastante frequência, o **acesso a ela deve acontecer de forma eficiente**. Algumas estruturas rápidas para implementá-la:

- Tabelas Hash;
- Árvore Binária de Busca;
- Pilhas, Filas, entre outros.

variável 1	...
variável 2	...
variável 3	...

*Tabela de Símbolos*

## 1.6. Compiladores e Interpretadores

### *Tratador de Erros*

Parte do Compilador responsável em tratar erros em qualquer uma das fases de compilação. Utiliza de duas estratégias básicas:

#### *1) Abortar*

Parar a compilação em qualquer uma das fases caso seja encontrado algum erro.

#### *2) Recuperar*

Tentar continuar o processo de compilação, desconsiderando algum fato.



# 1.6. Compiladores e Interpretadores

## 1.6.1.1 Análise Léxica

O fluxo de caracteres que compõem o programa fonte é lido e agrupado em sequências. As principais tarefas dessa fase são:

- Abrir o programa feito na linguagem fonte;
- Ler caracter a caracter do código, buscando identificar sequências significativas (Lexemas);
- Para cada Lexema, é verificado a que classe ele pertence (se é uma variável, palavra reservada, um número etc.) para que o mesmo possa se tornar um Token;
- Fechar o arquivo;

### *Lexemas*

Sequência de caracteres que obedece a um padrão (regra);

### *Token*

Um Lexema que já foi tratado (já foi alocado em uma classe de símbolos) e que poderá ser usado pelas próximas fases do Compilador.

*Vamos ver um pequeno exemplo?*

## 1.6. Compiladores e Interpretadores

Suponha que o programa fonte contenha o seguinte comando de atribuição:

*position = initial + rate \* 60;*

**position = initial + rate \* 60**



Lexical Analyzer



**(id, 1) (=) (id, 2) (+) (id, 3) (\*) (60)**

Tabela de Símbolos

1	position
2	initial
3	rate

(Representação do comando após Análise Léxica)

*Mas quais são as regras para se criar os Tokens?*

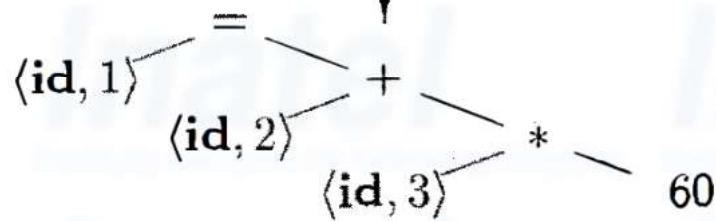
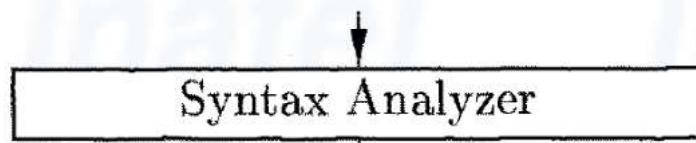
*Veremos isso em detalhes no Cap.2*

# 1.6. Compiladores e Interpretadores

## 1.6.1.1.2 Análise Sintática

O Analisador Sintático utiliza os Tokens produzidos pelo Analisador Léxico para criar uma representação intermediária do tipo árvore (Árvore Sintática), que mostra uma interação válida entre uma sequência de Tokens.

- Cada nó interior representa uma operação, e os filhos do nó representam os argumentos da operação.

$$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$$


*Mas como é possível criar interações válidas entre os Tokens?*

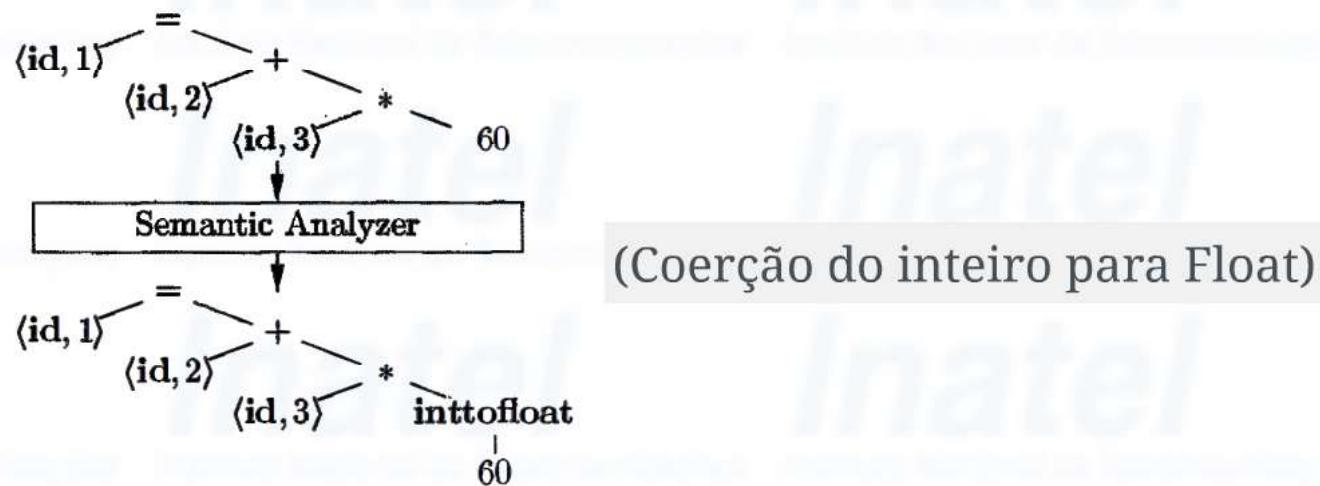
*Veremos isso em detalhes no Cap.3*

# 1.6. Compiladores e Interpretadores

## 1.6.1.1.3 Análise Semântica

O Analisador Semântico utiliza a árvore sintática e as informações da tabela de símbolos para verificar a consistência semântica do programa fonte com a definição da linguagem.

- Uma parte importante da Análise Semântica é a **verificação de tipo**, em que o compilador verifica se cada operador possui operandos compatíveis.
- Realiza o processo de **coerção** (ou parse automático dos operandos quando possível).



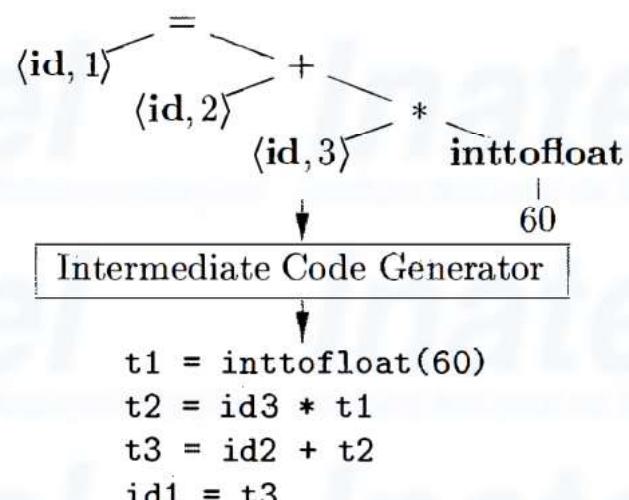
# 1.6. Compiladores e Interpretadores

## 1.6.1.1.4. Geração de Código Intermediário

Na tradução de um programa fonte para um código objeto, o compilador pode produzir uma ou mais representações intermediárias que podem ter diversas formas.

As representações intermediárias devem possuir duas propriedades importantes: ser facilmente produzida e ser facilmente traduzida para a máquina alvo.

- A Representação Intermediária mais utilizada é a chamada **Código de Três Endereços (Three Code Address)** que consiste de uma operação, dois operandos de entrada e um operando de saída.



Árvore sintática transformada em um código de três endereços.

# 1.6. Compiladores e Interpretadores

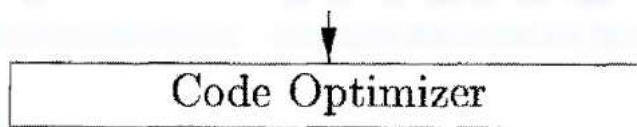
## 1.6.1.1.5. Otimização de Código

São realizadas transformações no código intermediário com objetivo de produzir um código melhor, mais rápido e que consuma menos energia (independente da arquitetura da máquina).

Algumas técnicas de Otimização aplicadas:

- Inibição de variáveis declaradas e não utilizadas;
- Otimização de Laços;
- Inibição de segmentos de códigos não relevantes, entre outras;

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```



```
t1 = id3 * 60.0
id1 = id2 + t1
```

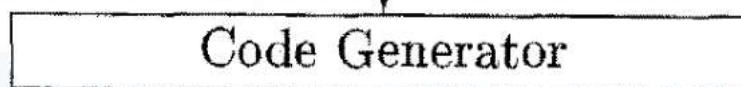
# 1.6. Compiladores e Interpretadores

## 1.6.1.1.6. Geração de Código Alvo

Nesta etapa, é gerada a saída do programa da linguagem fonte em linguagem Assembly.

Vale lembrar que cada linguagem Assembly é específica para uma Arquitetura específica de computador, e que após esta fase, o Assembler (ou Montador) entra em ação, produzindo o Código de Máquina (Instruções formadas por 0s e 1s).

```
t1 = id3 * 60.0  
id1 = id2 + t1
```



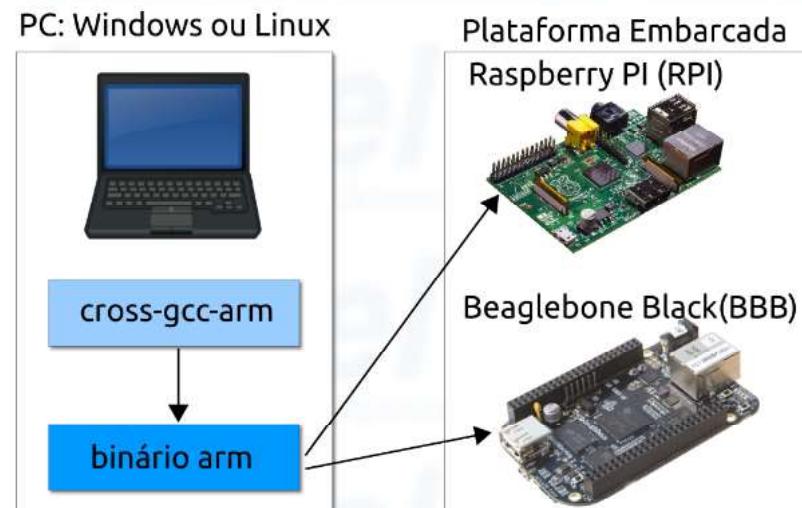
```
LDF R2, id3  
MULF R2, R2, #60.0  
LDF R1, id2  
ADDF R1, R1, R2  
STF id1, R1
```

# 1.6. Compiladores e Interpretadores

## 1.6.1.2. Compilação Cruzada

A compilação cruzada é uma técnica usada para compilar programas de uma plataforma Y a partir de uma plataforma X, onde X é diferente de Y.

Com a compilação cruzada, é possível compilar programas dentro de um PC, para rodarem em outra plataforma destino como por exemplo, o Android ou qualquer plataforma Embarcada, onde os recursos de memória e armazenamento são mais limitados.



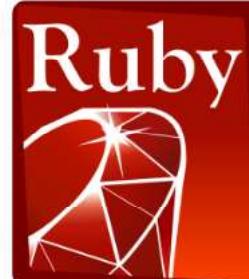
# 1.6. Compiladores e Interpretadores

## 1.6.2 Interpretadores

*Um interpretador é um programa capaz de interpretar as intruções da linguagem fonte (alto nível), executando-as diretamente.*

- Traduz o código fonte linha a linha. As instruções de cada linha são lidas, verificadas e são convertidas diretamente em linguagem de máquina;
- O código fonte não é totalmente traduzido antes de ser executado;
- Geralmente utilizam apenas de recursos da Análise Léxica, Sintática e Semântica;

*Ex:*

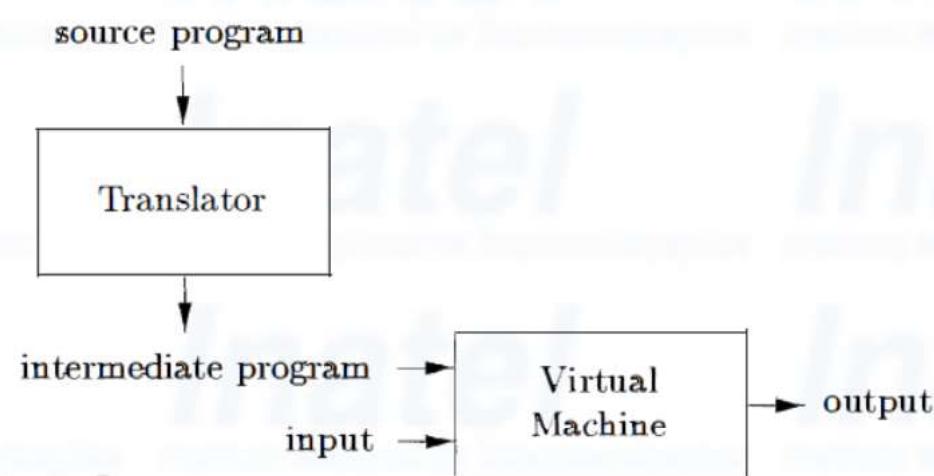


# 1.6. Compiladores e Interpretadores

## 1.6.4. Compiladores Híbridos

Em linguagens híbridas, o compilador tem o papel de converter o código fonte em um código chamado de bytecode, que é uma linguagem de baixo nível.

Um exemplo deste comportamento é o do compilador da linguagem Java que, em vez de gerar código da máquina hospedeira (onde se está executando o compilador), gera um código chamado Java Bytecode que são interpretados por uma Máquina Virtual (Virtual Machine).



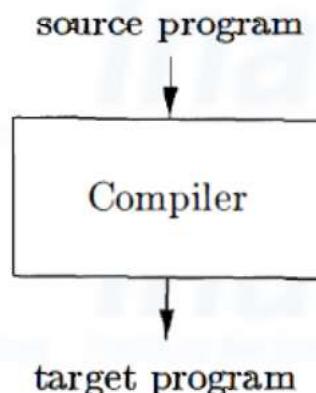
# 1.6. Compiladores e Interpretadores

## 1.6.3. Diferenças entre Compiladores e Interpretadores

### COMPILADOR

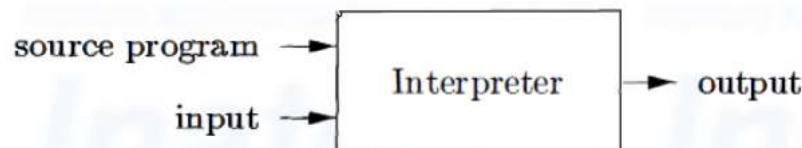


- O código fonte é compilado para uma arquitetura específica.
- Tempo de execução é rápido.
- O código alvo é otimizado para acelerar a execução.
- Quando um programa possui cálculos a serem feitos, como:  $A = (B*C) + (B*C) + (B*C)$  o compilador cria uma variável temporária para armazenar o valor de  $(B*C)$  não necessitando recalcular.



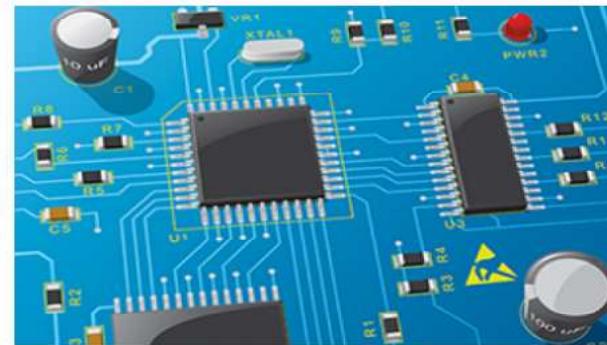
### INTERPRETADOR

- O código fonte é interpretado diretamente por um programa.
- Tempo de execução depende do tamanho do programa.
- A interpretação do código não optimiza o código fonte.
- Quando um programa possui cálculos a serem feitos, como:  $A = (B*C) + (B*C) + (B*C)$  o interpretador recalcula a expressão  $(B*C)$  3 vezes, diminuindo o desempenho.



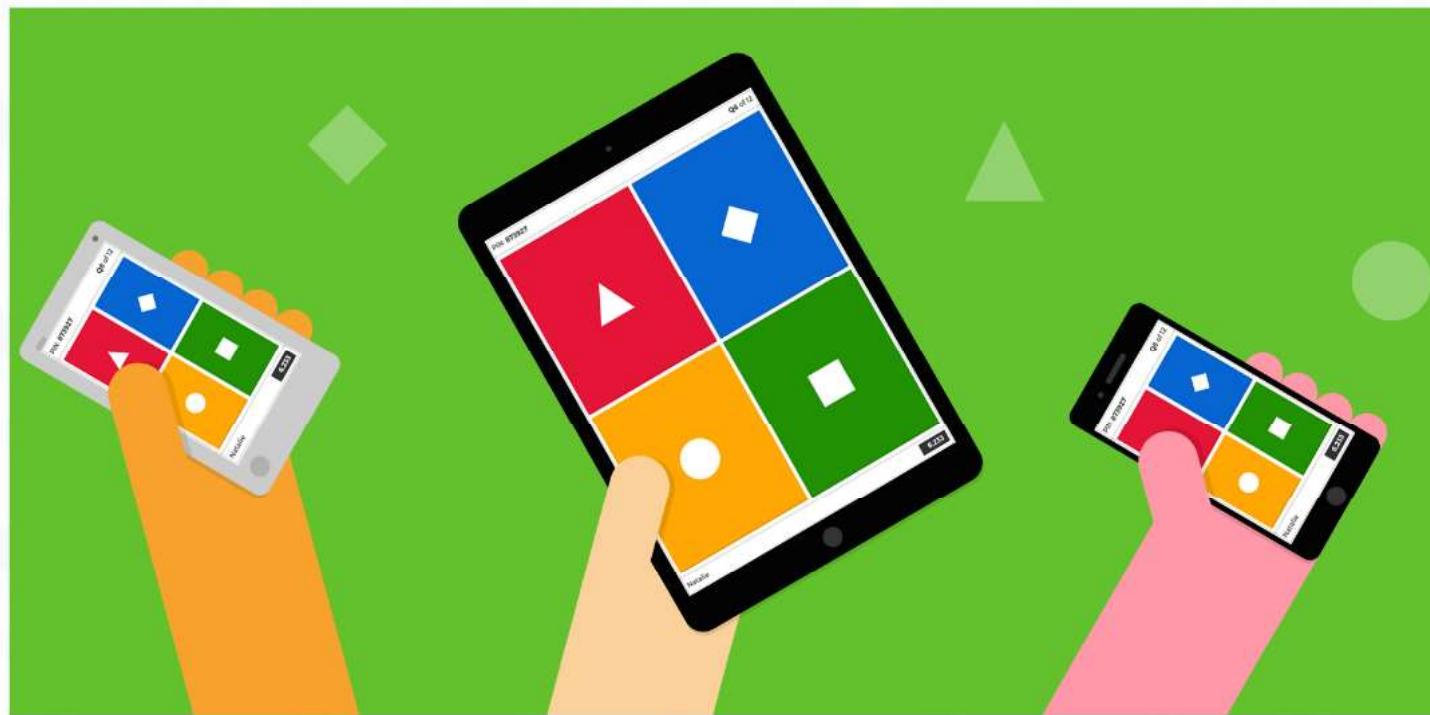
# 1.7. Aplicações da construção de Compiladores

- Construção de diferentes linguagens de programação de Alto Nível;
- Tradutores de documentos (.doc, .java, .c) para outra linguagem;
- Otimizações para arquiteturas de computadores já existentes;
- Projeto de novas arquiteturas de computadores;
- Criação de ambientes simulados (Simuladores);
- Implementar interfaces de comunicação baseadas em softwares com diferentes tipos de hardwares;
- E diversas outras aplicações...



HORA DO

# Kahoot!



ACESSE:

<https://kahoot.it>

**Fim  
do  
Capítulo 1**



**EXERCÍCIOS**