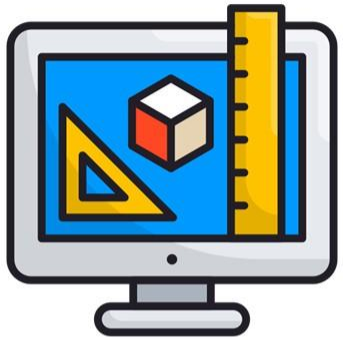


Tecnologia da Informação

COMPILADORES

Prof. Alessandro Borges

Especialista em Gestão de Projetos

alessandroborgesoficial

Compiladores

Introdução

Prof. Alessandro Borges

Especialista em Gestão de Projetos

alessandroborgesoficial

Objetivos de Aprendizagem

- Explicar interpretação e compilação de linguagens de programação.
- Definir a estrutura de um compilador.
- Demonstrar as fases do processo de compilação.

Pense nisso!

VOCÊ JÁ SE QUESTIONOU?

Em um mundo onde os sistemas de software estão cada vez mais complexos e interconectados, como você garantiria que um aplicativo não apenas funcione, mas também seja capaz de evoluir, se adaptar a novas tecnologias e atender às necessidades futuras dos usuários? Qual o papel da arquitetura de software nesse processo?

A arquitetura de software garante que um aplicativo seja adaptável, evolutivo e capaz de atender às necessidades futuras, funcionando como a planta de um sistema complexo.

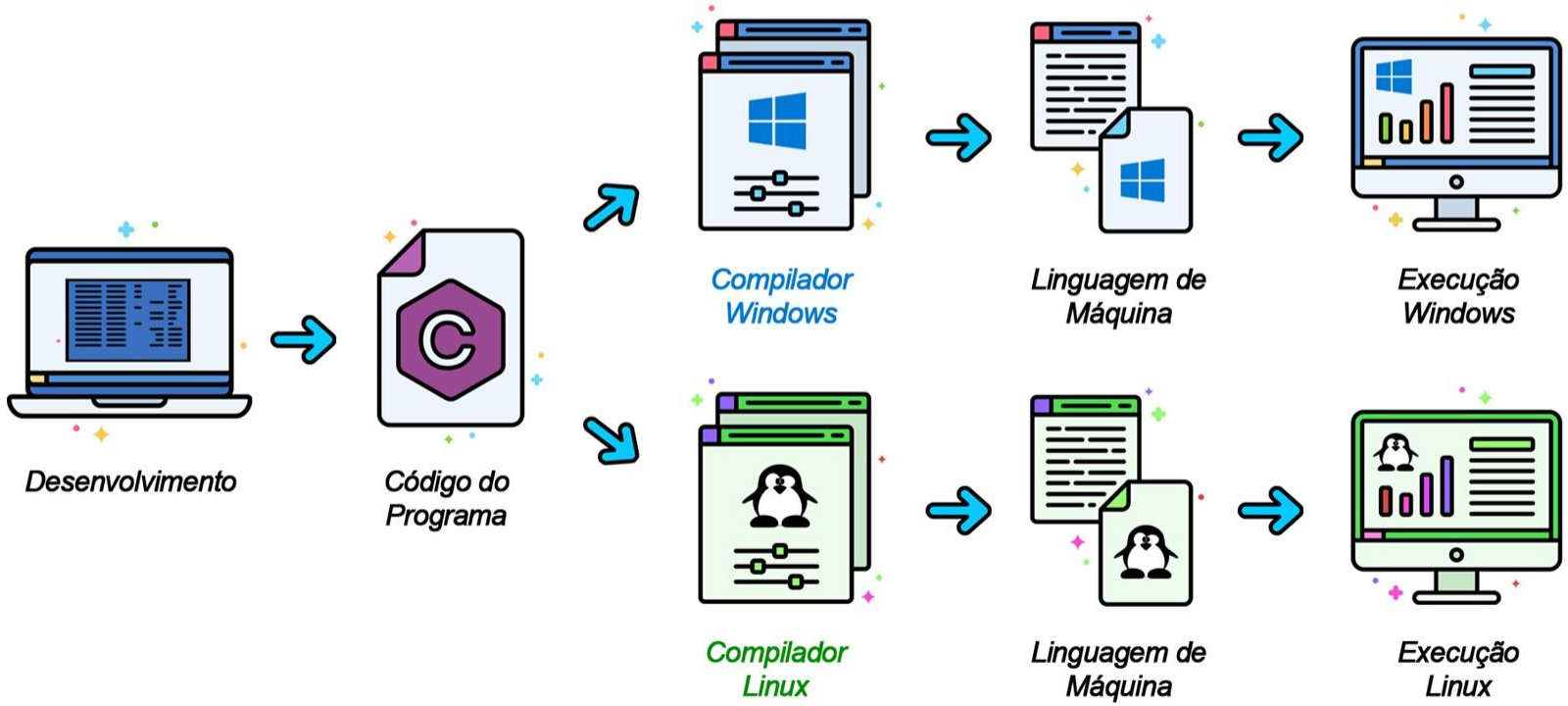
Processo de
Compilação

Compilação é a conversão de um código construído em uma linguagem de alto nível próxima ao que o programador entende em código de máquina, sendo destinado especificamente a um processador ou sistema operacional. O compilador traduz o programa inteiro em código de máquina de uma só vez, criando um arquivo executável que pode ser utilizado várias vezes, sendo assim pouco prático, porém mais rápido.

Exemplo: Implementações em Linguagem C.

O que é Linguagem de Máquina?

Conjunto de instruções de um software em linguagem de baixo nível, que são executadas diretamente pelo processador ou sistema operacional específico, definidos durante o processo de compilação. Por ser executada diretamente e ser específica é mais rápida que o bytecode.



Compilando um Programa em C

```c
#include <stdio.h>

int main() {
    // Declara e inicializa a variável 'numero' com o valor 10 em uma única linha
    int numero = 10;
    printf("O valor da variável número é: %d\n", numero);
    return 0;
}
```

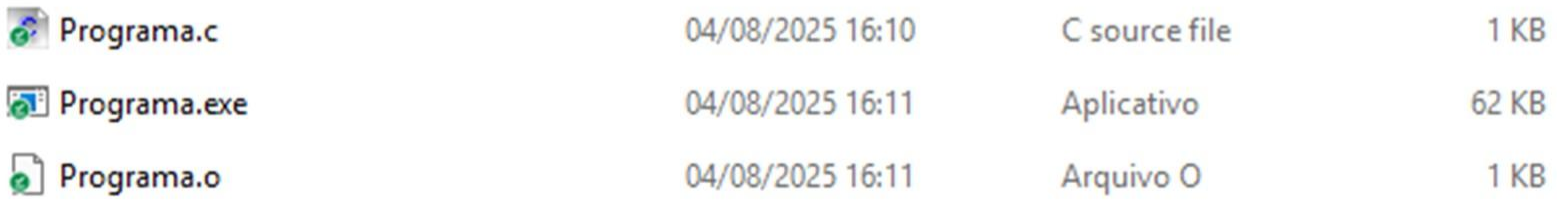
Resultado (Prompt):

```
O valor da variável número é: 10
```

Principais Arquivos Gerados Pela criação de um Programa em C

1. **Formato .c (Código Fonte):** Arquivo que o programador escreve de forma legível as instruções, funções e a lógica que definem o que o software deve fazer.
2. **Formato .o I Formato .obj (Object File):** Contém o código-fonte de um único arquivo (.c) já traduzido para código de máquina.

   É um arquivo intermediário que ainda não é executável, pois referências a funções e variáveis externas ainda não foram resolvidas.
3. **Formato .exe (Executável):** Arquivo pronto para ser executado diretamente pelo sistema operacional. Ele é o resultado do processo de linkagem, que combina um ou mais arquivos objeto (.o) com as bibliotecas necessárias em um único pacote coeso.



Introdução

Processo Básico de Compilação

O processo de build de um programa em linguagens como C e C++, é orquestrado por toolchains (conjuntos de ferramentas) como o **GCC** e o **Clang**, possuindo 4 etapas clássicas:

- **GCC - GNU Compiler Collection:** Suíte de ferramentas de programação de código aberto, que permite compilar em diversas linguagens além do C e C++.
- **Clang:** Compilador Front-End de código aberto para a infraestrutura LLVM, focado nas linguagens da família C (C, C++, Objective-C) e famoso por suas mensagens de erro detalhadas, sendo o compilador padrão para todo o ecossistema de desenvolvimento da Apple.

Introdução

Processo Básico de Compilação

1. **Pré-processamento:** Recebe e modifica o código fonte com base em diretivas (comandos que começam com #), inserindo o código das diretivas diretamente no arquivo:

   - **Expansão dos #include:** Busca e insere os códigos das bibliotecas em um arquivo (.i ou .ii) que será gerado temporariamente.
   - **Controle de blocos:** Permite remover ou incluir blocos de código antes da compilação, atendendo situações específicas, usando diretivas como: #if, #ifdef (if defined), #ifndef (if not defined), #else e #endif.

- **Entrada:** Código fonte (meu_programa.c).
- **Saída:** Arquivo de texto temporário (meu_programa.i) com seu código "expandido", ou seja, com todas as substituições já feitas.

Introdução

Processo Básico de Compilação

2. Compilação: O compilador traduz o código expandido (meu_programa.i) para uma linguagem de baixo nível chamada **Assembly** (meu_programa.s), estando dividido em duas etapas importantes:

- **Front-End:** Análise léxica, sintática e semântica, gerando o IR.
- **Back-End:** Otimização do IR e tradução para a linguagem de montagem da arquitetura alvo.

- **Entrada:** O arquivo temporário de código puro (meu_programa.i).
- **Saída:** Um arquivo de texto contendo código Assembly (meu_programa.s).

Introdução

Processo Básico de Compilação

3. Montagem (Assembly): O processo de Montagem é a última e mais simples tarefa do Back-End, onde o programa montador **traduz o código Assembly para código de máquina binário** — a linguagem nativa do processador (os 0s e 1s). Esse processo **gera o arquivo objeto (.o – padrão Linux ou .obj – padrão Windows)**, que contém as instruções do programa, mas as localizações de funções externas (como printf) são deixadas como referências simbólicas para serem resolvidas na etapa final.

- **Entrada:** O arquivo de código Assembly **(meu_programa.s)**.
- **Saída:** Um arquivo objeto **(meu_programa.o)**, que contém o código de máquina ainda não "ligado".

Introdução

Processo Básico de Compilação

4. Linkagem (Linking): O linker combina um ou mais arquivos objeto (.o) com o código das bibliotecas necessárias. Sua função é resolver as referências simbólicas pendentes, conectando as chamadas a funções como printf às suas definições reais nas bibliotecas. O processo resulta em um único arquivo executável, com todo o código necessário para o programa rodar.

- **Entrada:** Um ou mais arquivos objeto (meu_programa.o).
- **Saída:** O arquivo executável final (meu_programa.exe).

Processo de
INTERPRETAÇÃO

A interpretação executa em tempo real, linha a linha, o código do software por um programa chamado interpretador que em seguida é executado pelo sistema operacional ou processador. Esse método é mais prático, pois é possível executar programas em diferentes sistemas, bastando para isso um interpretador compatível, porém é mais lento, pois sempre que executado precisa ser interpretado.

Exemplo: Implementações em HTML (Hipertext Markup Language).

Introdução

Processo Hibrido

A linguagem Java utiliza uma abordagem híbrida, muito semelhante à do Python, mas com passos mais explícitos para o desenvolvedor. O processo combina tanto a compilação quanto a interpretação.

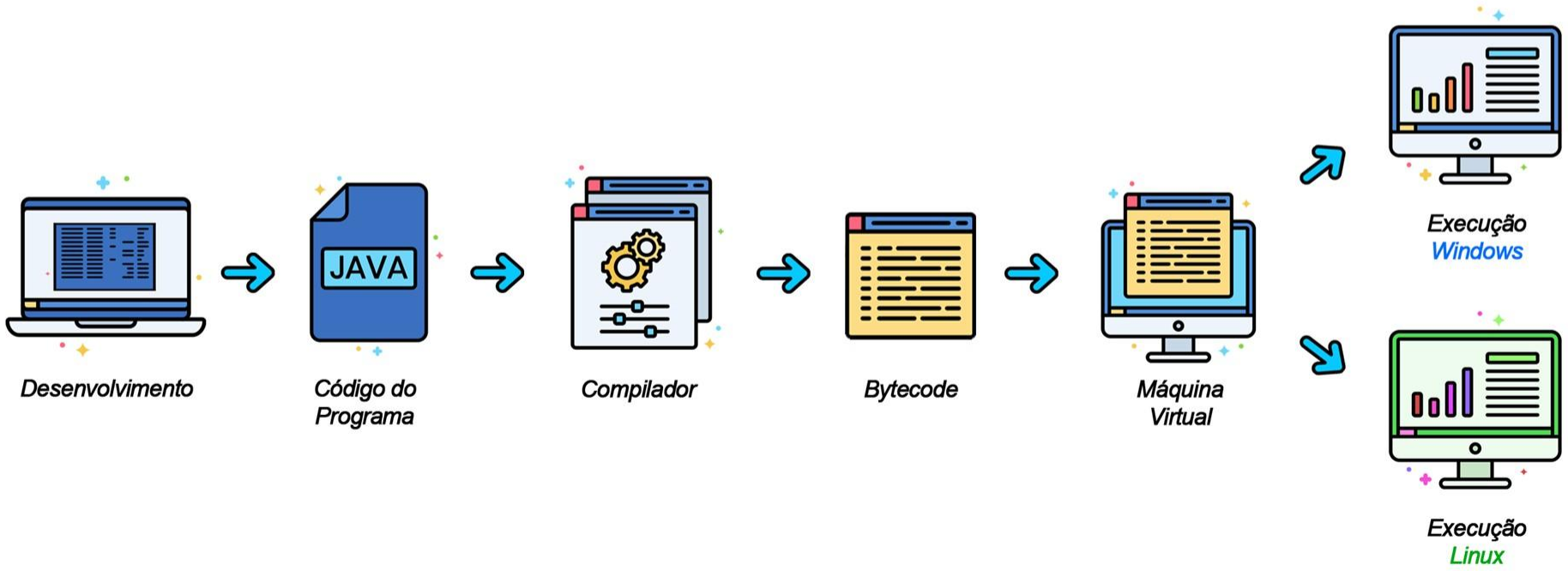
O funcionamento é o seguinte:

1. **Compilação para Bytecode:** Primeiro, o código-fonte escrito em um arquivo .java é compilado por um programa chamado javac (o compilador Java). O resultado desse processo não é um código de máquina executável, mas sim um arquivo .class que contém bytecode. Esse bytecode é um código intermediário e independente de plataforma.
2. **Execução pela Máquina Virtual Java (JVM):** Em seguida, esse bytecode é interpretado pela Máquina Virtual Java (JVM). A JVM atua como um tradutor, lendo as instruções do bytecode e as convertendo em comandos de máquina nativos para o sistema operacional e o processador em que está sendo executada.

Introdução

Processo Hibrido

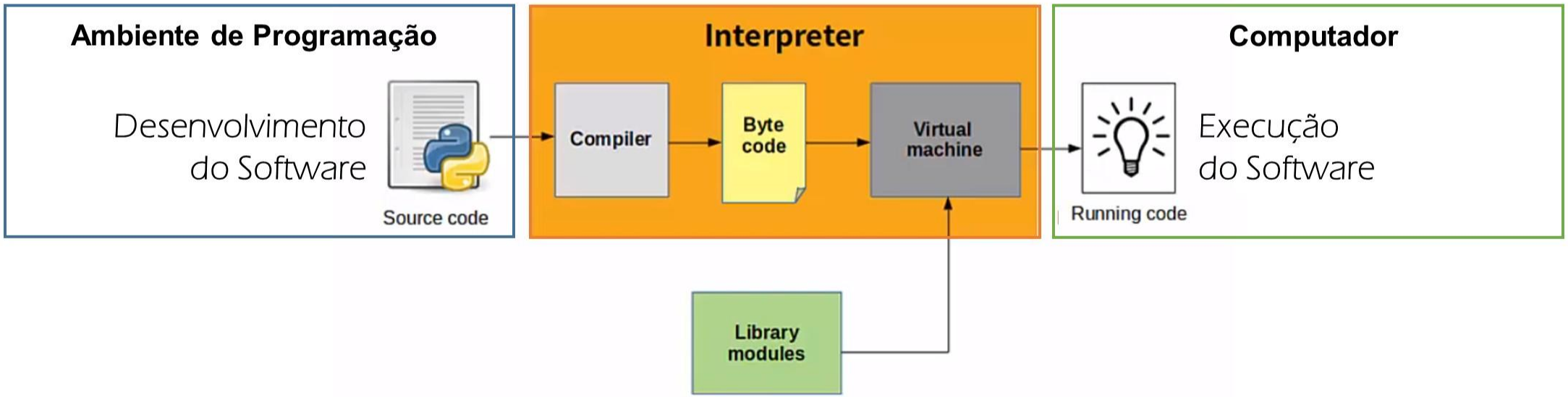
Para otimizar o desempenho, a JVM frequentemente utiliza um compilador JIT (Just-In-Time). Ele identifica as partes do bytecode que são executadas com mais frequência e as compila para código de máquina nativo em tempo de execução, fazendo com que essas partes do programa rodem de forma muito mais rápida.



Introdução

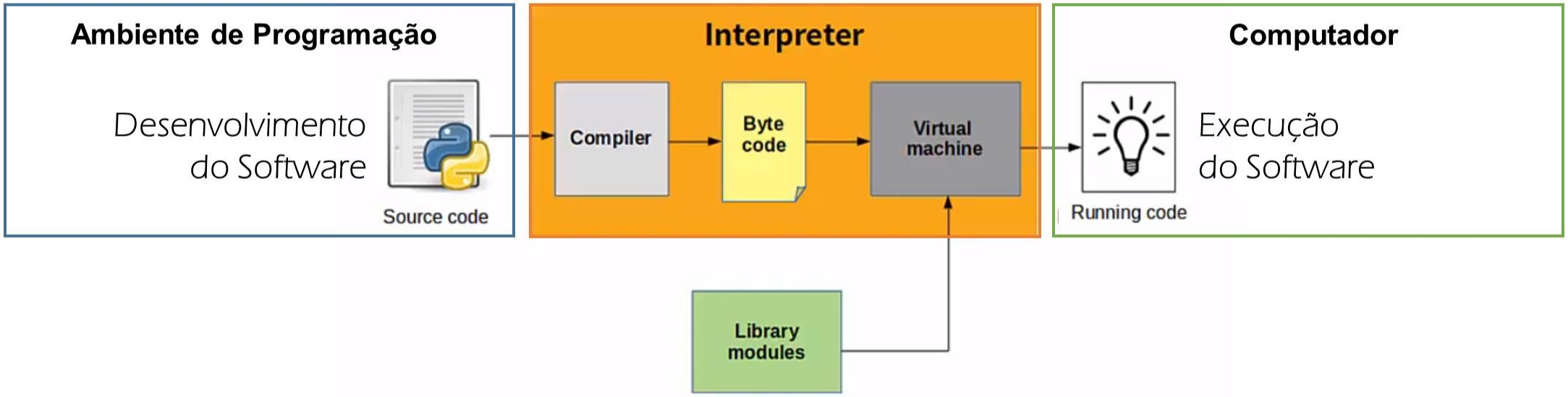
O Python e Seu Processo

O Python é uma linguagem híbrida, porém informalmente considerada interpretada, pois suas principais implementações utilizam um software interpretador que compila o código-fonte utilizado no desenvolvimento de um programa em bytecode que é interpretado em tempo real por uma máquina virtual durante a execução do programa.



O que é Bytecode?

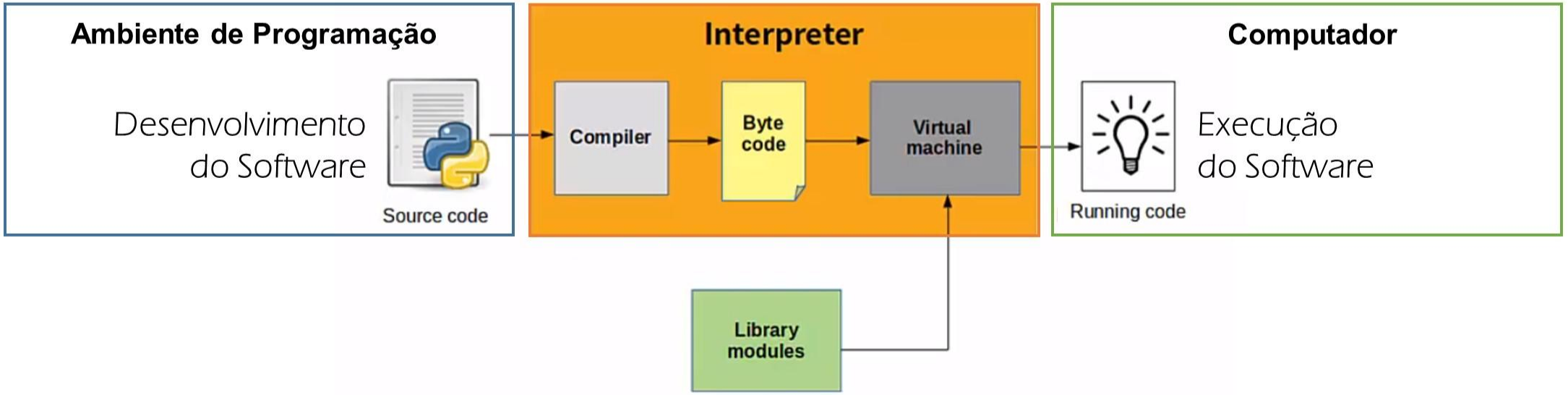
Bytecode é um conjunto de instruções compiladas, porém executadas por uma máquina virtual, software que utiliza recursos do computador para simular um computador virtual e que permite a execução do programa independentemente do processador ou sistema operacional. O bytecode é mais portável e seguro que a linguagem de máquina.



Introdução

O Interpretador

O Interpretador do Python compila o código-fonte de uma implementação em Bytecode antes da interpretação em si. A interpretação para código de máquina ocorre em tempo real, linha a linha conforme a execução vai progredindo, essa interpretação é realizada por uma máquina virtual conhecida como PVM (Python Virtual Machine) contida no próprio interpretador.



Introdução

Arquivos Padrões Python

Arquivo .PY

Um arquivo "**.py**" é o formato padrão usado ao ser gerado um arquivo de programa ou script escrito em Python. Os arquivos "**.py**" são frequentemente usados para programar servidores da Web e outros sistemas de computadores administrativos.

Programa.py

Arquivo .PYC

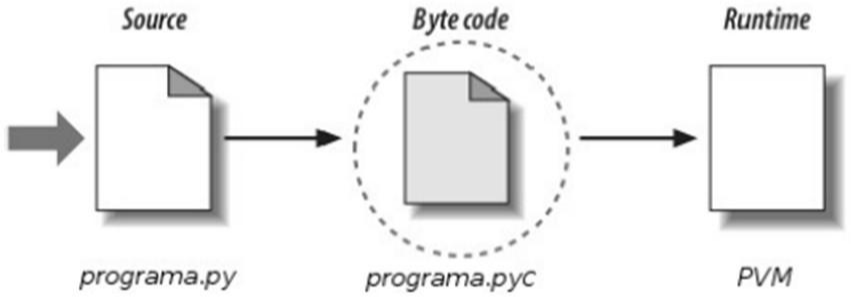
Os arquivos "**.pyc**" são usados pela linguagem de programação Python. Um arquivo "**.pyc**" é um arquivo executável que contém **bytecode** compilado para uma determinada arquitetura.

Programa.pyc

Dica Rápida

Como funciona a interpretação e execução no Python!

O interpretador do Python está embutido na Implementação (CPython ou PyPy), sendo executado automaticamente por ela, compilando em **bytecode** e executando.



Podemos compilar manualmente um arquivo padrão "**.py**" para um arquivo "**.pyc**", porém na maioria das vezes esse é um processo desnecessário.

Exemplo:

```
>>> py_compile.compile('programa.py')
'__pycache__/programa.cpython-34.pyc'
```

Compiladores

Programa Objeto

Prof. Alessandro Borges

Especialista em Gestão de Projetos

alessandroborgesoficial

Objetivos de Aprendizagem

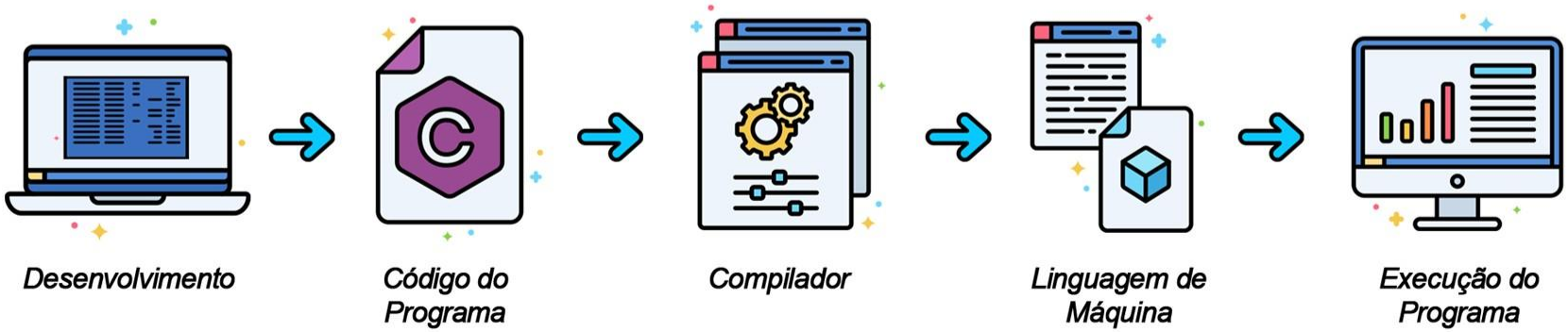
- Entender um Programa Objeto
- Identificação das Etapas Básicas da Compilação.
- Demonstrar as fases do processo de compilação.

Introdução

O Que é Um Programa Objeto?

Um programa objeto (meu_programa.o ou meu_programa.obj) é um arquivo que contém código de máquina. Ele é o resultado da tradução de um código-fonte, que pode ser tanto uma linguagem de alto nível como C/C++ (processada pelo compilador e depois enviada para o montador), quanto um código em linguagem de baixo nível como Assembly (processada diretamente pelo montador).

No entanto, este código de máquina ainda não é um programa executável completo. Ele é um produto intermediário, contendo o código traduzido de uma única unidade de compilação (geralmente um arquivo .c ou .cpp), mas com referências a elementos externos ainda não resolvidas.



Programa Objeto

Características Principais

1. **Código de Máquina Relocável:** Os endereços de memória para funções e variáveis no Programa Objeto ainda não são finais. Eles são escritos de forma relativa, permitindo que um outro programa, o **ligador (linker)**, decida mais tarde onde alocar esse código na memória final do programa.

2. **Referências Simbólicas:** Capacidade de registrar todos os símbolos (funções ou variáveis) que são utilizados pelo código no arquivo, mas cuja definição se encontra em uma unidade de compilação externa (outro arquivo objeto ou uma biblioteca).

Uma referência simbólica é um marcador que o compilador cria em um arquivo objeto para um símbolo (como a função printf) que é usado no código, mas definido externamente.

Este marcador funciona como uma diretiva para o linker, instruindo-o a encontrar o endereço final deste símbolo em outros arquivos ou bibliotecas e a conectar todas as chamadas a ele durante a fase de ligação.

3. **Tabela de Símbolos:** Cada programa objeto contém uma tabela de símbolos. Esta tabela lista todas as funções e variáveis globais que são definidas neste arquivo e que podem ser usadas por outros arquivos (símbolos exportados), bem como os símbolos que ele precisa de outros arquivos (símbolos importados).

Programa Objeto

Estrutura de Um Arquivo Objeto

Embora o formato exato varie entre sistemas operacionais (como ELF em Linux ou COFF em Windows), um arquivo objeto geralmente contém as seguintes seções:

1. **Cabeçalho (Header):** Contém metadados sobre o arquivo, como o tamanho e a posição das outras seções e a arquitetura da máquina de destino (ex: x86-64).
2. **Seção de Código (.text):** Armazena as instruções de máquina compiladas do seu código-fonte.
3. **Seção de Dados (.data):** Contém as variáveis globais e estáticas que são inicializadas com um valor.
4. **Seção BSS (.bss):** Reserva espaço para variáveis globais e estáticas que não são inicializadas (ou inicializadas com zero).
5. **Tabela de Símbolos (.symtab):** Mapeia os nomes de símbolos para seus locais dentro do arquivo.
6. **Informações de Relocação:** Contém as instruções para o ligador sobre como modificar os endereços de memória quando o código for combinado com outros arquivos.

C

```c
#include <stdio.h>

// Para popular a seção .data
int bonus_global = 5;

// Para popular a seção .bss
int resultado_final_global;

int main() {
    // Variáveis locais (vivem na pilha/stack em tempo de execução)
    int a = 10;
    int b = 20;

    // Operação de soma
    resultado_final_global = a + b + bonus_global;

    // Exibição do resultado
    printf("O resultado final é: %d\n", resultado_final_global);

    return 0;
}
```

Exemplo de Código em C

Imagine um pequeno código-fonte chamado soma.c que ilustrar quais tipos de dados concretos são armazenados em cada uma das seções de um arquivo objeto.

Estrutura De Um Arquivo Objeto

Quando o compilador processa soma.c e gera o arquivo objeto soma.o, o conteúdo será organizado da seguinte forma:

1. Seção de Código (.text)

Esta seção armazena as instruções de máquina para a função main. Não será código C, mas sim o código de máquina (representado aqui de forma simplificada como Assembly) que o processador executa.

Exemplo de dado armazenados

```
; Código para a função main()
push    rbp             ; Prepara o frame da função
mov     rbp, rsp
mov     DWORD PTR [rbp-4], 10    ; Move o valor 10 para a variável local 'a'
mov     DWORD PTR [rbp-8], 20    ; Move o valor 20 para a variável local 'b'

; Lógica da soma
mov     edx, DWORD PTR [rbp-4]   ; Carrega 'a'
mov     eax, DWORD PTR [rbp-8]   ; Carrega 'b'
add     eax, edx                 ; Soma a + b
add     eax, DWORD PTR bonus_global[rip] ; Adiciona o bônus global
```

```
; Código para a função main()
push    rbp                             ; Prepara o frame da função
mov     rbp, rsp
mov     DWORD PTR [rbp-4], 10           ; Move o valor 10 para a variável local 'a'
mov     DWORD PTR [rbp-8], 20           ; Move o valor 20 para a variável local 'b'

; Lógica da soma
mov     edx, DWORD PTR [rbp-4]          ; Carrega 'a'
mov     eax, DWORD PTR [rbp-8]          ; Carrega 'b'
add     eax, edx                        ; Soma a + b
add     eax, DWORD PTR bonus_global[rip]            ; Adiciona o bônus global
mov     DWORD PTR resultado_final_global[rip], eax  ; Guarda o resultado final

; Preparação para o printf
mov     esi, eax                        ; Move o resultado para o argumento do printf
lea     rdi, .LC0[rip]                  ; Carrega o endereço da string de formato
call    printf                          ; Chama a função 'printf' (aqui haverá uma entrada de relocação)
mov     eax, 0                          ; Prepara o valor de retorno 0
pop     rbp                             ; Restaura o frame
ret                                     ; Retorna da função main
```

Estrutura De Um Arquivo Objeto

2. Seção de Dados (.data)

Esta seção armazena os valores das variáveis globais e estáticas que já possuem um valor inicial.

Exemplo de dado armazenado:

Para a variável `int bonus_global = 5;`:

A seção .data conterá a representação binária do número 5. Em um sistema de 32 bits, isso seria armazenado como a sequência de 4 bytes: `05 00 00 00` (em hexadecimal, little-endian).

A string de formato `"O resultado final é: %d\n"` também é armazenada aqui (ou em uma seção de dados somente leitura, `.rodata`).

Estrutura De Um Arquivo Objeto

3. Seção BSS (.bss)

Esta seção não armazena dados, apenas registra o espaço necessário para variáveis globais e estáticas não inicializadas. É uma otimização para não desperdiçar espaço no arquivo objeto com zeros.

Exemplo de dado armazenado:

Para a variável int resultado_final_global;:

A seção .bss não contém o valor 0. Em vez disso, ela registra que o símbolo resultado_final_global precisa de um espaço de 4 bytes (tamanho de um int), que deverá ser preenchido com zeros pelo sistema operacional quando o programa for carregado na memória.

Estrutura De Um Arquivo Objeto

4. Tabela de Símbolos (.symtab)

Esta é a "lista de referências" ou o índice do arquivo. Ela mapeia os nomes de variáveis e funções para suas localizações.

Exemplo de dado armazenado (de forma simplificada):

Símbolo (Nome)	Onde está definido?	Informação Adicional
`bonus_global`	Seção `.data`	Definido neste arquivo (símbolo global).
`resultado_final_global`	Seção `.bss`	Definido neste arquivo (símbolo global).
`main`	Seção `.text`	Definido neste arquivo (símbolo global, é uma função).
`a`, `b`	Não aparecem aqui	São variáveis locais, invisíveis fora da função `main`.
`printf`	**INDEFINIDO**	Símbolo externo. Precisa ser encontrado pelo ligador.

Estrutura De Um Arquivo Objeto

5. Informações de Relocação

A seção de informações de relocação consiste em um conjunto de diretivas para o **linker**. Cada diretiva especifica um local no código de máquina que depende de um **endereço simbólico**, o qual deve ser resolvido para seu **endereço virtual absoluto** durante a fase de ligação.

Exemplo Ilustrativo

- **O Compilador:** Ao ver a instrução **call printf**, ele não sabe o endereço de **printf**.
- **Entrada de Relocação:** O compilador gera uma instrução de chamada incompleta (ex: **call 0**) e cria uma entrada na Tabela de Relocação solicitando o preenchimento da chamada.
- **O Linker:** Quando o linker junta todos os arquivos, ele descobre que o endereço de printf é, por exemplo, 0x401080. Ele então lê a Entrada de Relocação, vai até a posição X do código de máquina e completa a instrução, inserindo o endereço correto.

Compiladores

Processo de Compilação

Prof. Alessandro Borges

Especialista em Gestão de Projetos

alessandroborgesoficial

Objetivos de Aprendizagem

- Entender um Programa Objeto
- Identificação das Etapas Básicas da Compilação.
- Demonstrar as fases do processo de compilação.

Processo Geral

Estrutura Geral

Front-End e Back-End

Os compiladores modernos dividem o processo de compilação em duas partes principais, o Front-End e o Back-End, conectadas por uma representação de código neutra chamada Código Intermediário (IR - Intermediate Representation).

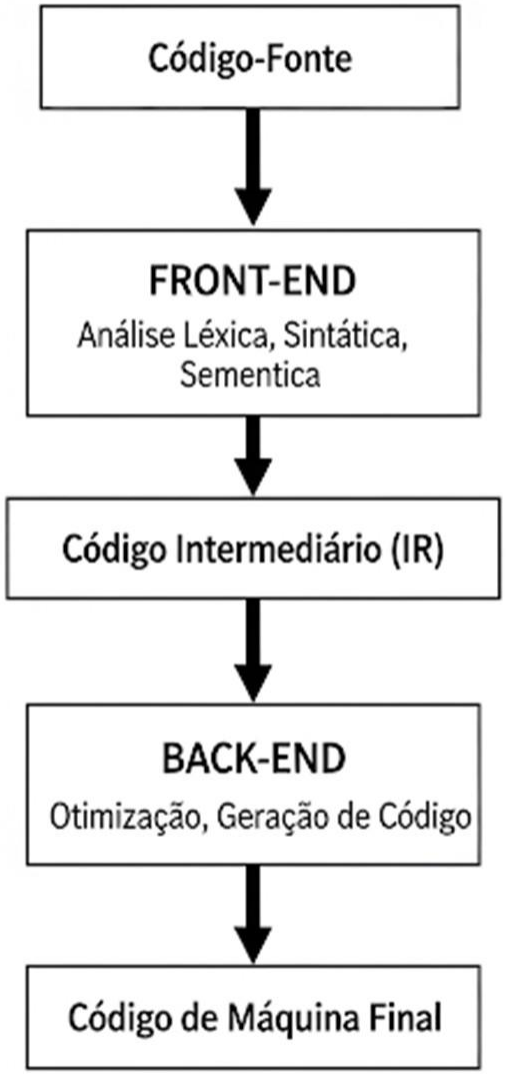
- **Front-End (O quê?):** Compreende a linguagem de programação de origem (Código-Fonte) e realiza análise léxica, sintática e semântica.
  - ❑ Gera o Código Intermediário.
  - ❑ É dependente da linguagem de origem.(um front-end de C++ é diferente de um de Rust).
- **Back-End (Como?):** É a parte que gera o código para a arquitetura de destino.
  - ❑ Otimiza e traduz o Código Intermediário para o código de máquina específico de um processador (como x86-64, ARM, etc.).
  - ❑ É dependente da arquitetura.

Portabilidade

A separação em Front-End e Back-End é genial porque permite, por exemplo, que um único Back-End para a arquitetura x86-64 seja usado por múltiplos Front-Ends (de C++, Rust, Swift, etc.). Da mesma forma, um único Front-End de C++ pode gerar código para múltiplos Back-Ends (x86-64, ARM, etc.), tornando a linguagem muito mais portátil.

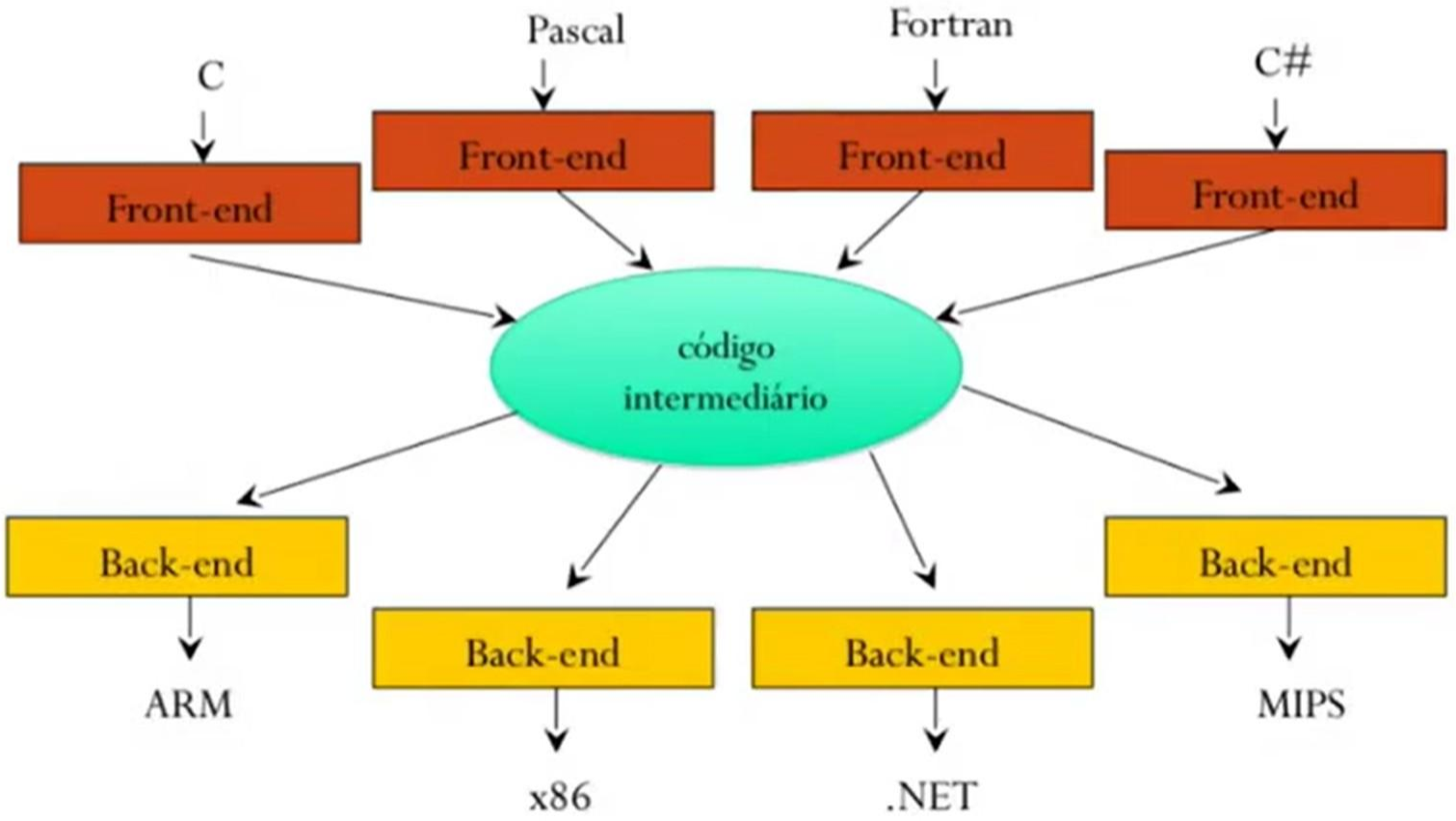
Um Código Intermediário pode ser lido por qualquer Back-End dentro do mesmo ecossistema ou projeto de compilador.

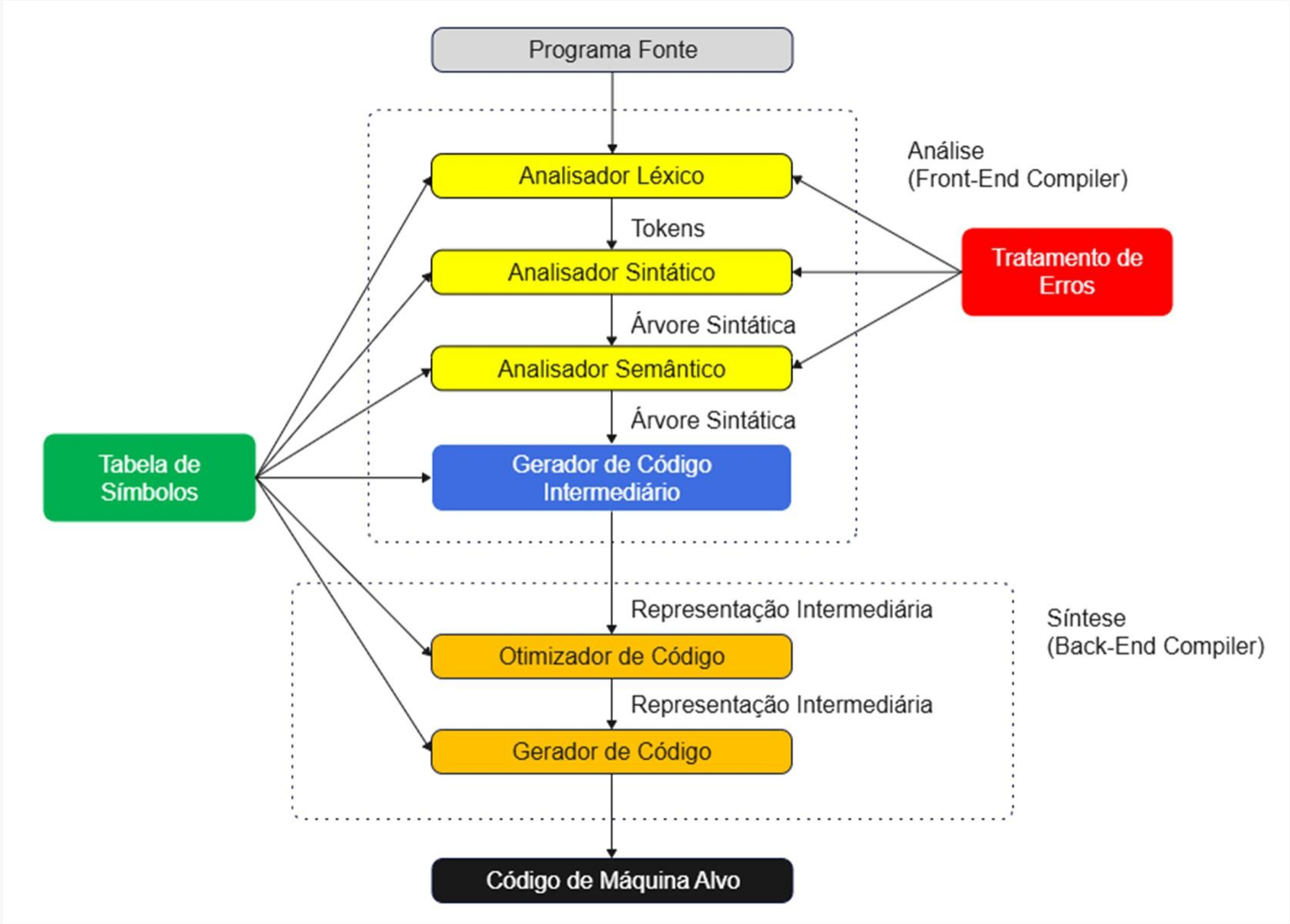
LLVM (Low Level Virtual Machine): Framework para a construção de compiladores e outras ferramentas relacionadas à manipulação de código.



Processo de Compilação

- Duas etapas
  - Análise (Front-End)
  - Síntese (Back-End)




Programa Fonte
Analisador Léxico
Tokens
Analisador Sintático
Árvore Sintática
Analisador Semântico
Árvore Sintática
Gerador de Código Intermediário
Tabela de Símbolos
Tratamento de Erros
Análise (Front-End Compiler)
Representação Intermediária
Otimizador de Código
Representação Intermediária
Gerador de Código
Síntese (Back-End Compiler)
Código de Máquina Alvo

Compiladores

Processo Front-End

Prof. Alessandro Borges

Especialista em Gestão de Projetos

alessandroborgesoficial

Objetivos de Aprendizagem

- Entender um Programa Objeto
- Identificação das Etapas Básicas da Compilação.
- Demonstrar as fases do processo de compilação.

Front-End

Fases de Análise

Análise Léxica

Contexto Humano

A análise léxica é o processo mental e automático de reconhecer palavras individuais a partir de um fluxo de letras ou sons. Nosso cérebro agrupa esses estímulos brutos em unidades de significado, as palavras, ignorando o que não é essencial, como os espaços entre elas. Essa segmentação é o passo fundamental que nos permite, em seguida, compreender a gramática e o sentido de uma frase.

Contexto Computacional

A análise léxica é a primeira e fundamental fase no processo de compilação de um programa de computador. Sua principal função é converter uma **sequência de caracteres** (o código-fonte) em uma sequência de unidades lógicas chamadas **tokens**.

O software que realiza essa tarefa é chamado de analisador léxico, lexer ou scanner.

Front-End

Fases de Análise

Análise Léxica

A **Análise Léxica (Scanning)** lê o código-fonte caractere por caractere e o agrupa em uma sequência de "tokens" (as menores unidades de significado da linguagem).

- **Entrada:** Uma sequência de caracteres, o código fonte:

  `resultado = a + 10;`

- **Saída:** Uma sequência de tokens. Cada token representa uma unidade léxica e seu tipo, como: literais, palavra-chave, identificador, operador e símbolos de pontuação.

  `IDENTIFICADOR(resultado)`, `OPERADOR(=)`, `IDENTIFICADOR(a)`, `OPERADOR(+)`, `NÚMERO(10)`, `PONTUAÇÃO(;)`

Front-End

Fases de Análise

Análise Léxica

A **Análise Léxica (Scanning)** lê o código-fonte caractere por caractere e o agrupa em uma sequência de "tokens" (as menores unidades de significado da linguagem).

Exemplos de Verificação Léxica

1. Caractere Inválido ou Desconhecido: `int valor = novo_valor@2;`
2. String ou Comentário Não Terminado: `char* saudacao = "Olá, mundo;`
3. Número Malformado: `float valor = 3.14.159;`

Dica Rápida!

O Que São Tokens?

Tokens são as menores unidades com significado para a linguagem de programação e na prática sua estrutura pode contém um tipo e, opcionalmente, um lexema. Eles podem ser:

- **Palavras-chave:** (`if` , `for` , `while` , `class` , `int`).
- **Identificadores:** Nomes de variáveis, funções, classes (por exemplo, `soma`).
- **Operadores:** (`+` , `-` , `*` , `=` , `!=`).
- **Literais:** Valores constantes, como números (`10` , `3.14`) ou strings (`"hello"`).
- **Símbolos de pontuação:** (`{` , `}` , `(` , `)` , `;`).

```java
public static void main (String [] args){
    int a = 10, b = 4;
    float c = a / b;
    System.out.print(c);
}
```

Tabela de Símbolos

1	
2	
3	
4	
5	
6	
7	
8	
9	

Lista de Tokens

Front-End

Fases de Análise

Análise Sintática

Contexto Humano

A análise sintática é o processo de verificar se uma sequência de palavras (já reconhecidas) forma uma frase gramaticalmente correta. Ela verifica se as palavras estão na ordem e estrutura que a gramática exige.

Exemplo: Sujeito, verbo, objeto.

Contexto Computacional

É o processo de verificar como os tokens se encaixam para formar "frases" estruturadas e hierárquicas, de acordo com as regras gramaticais da linguagem, gerando como resultado uma Árvore de Sintaxe Abstrata (AST - Abstract Syntax Tree).

- **Entrada:** A sequência de tokens da fase anterior.
- **Saída:** Uma AST (Abstract Syntax Tree).

Front-End

Fases de Análise

Análise Sintática

Contexto Computacional

É o processo de verificar como os tokens se encaixam para formar "frases" estruturadas e hierárquicas, de acordo com as regras gramaticais da linguagem, gerando como resultado uma Árvore de Sintaxe Abstrata (AST - Abstract Syntax Tree).

Exemplos de Verificação

1. Ponto e Vírgula Faltando: `int x = 5`
2. Parênteses, Chaves ou Colchetes Desbalanceados: `if (x > 5 { printf("maior"); }`
3. Operadores Mal Posicionados: `int y = 5 + * 10;`
4. Palavras-chave em Contextos Inválidos: `int for = 10;`

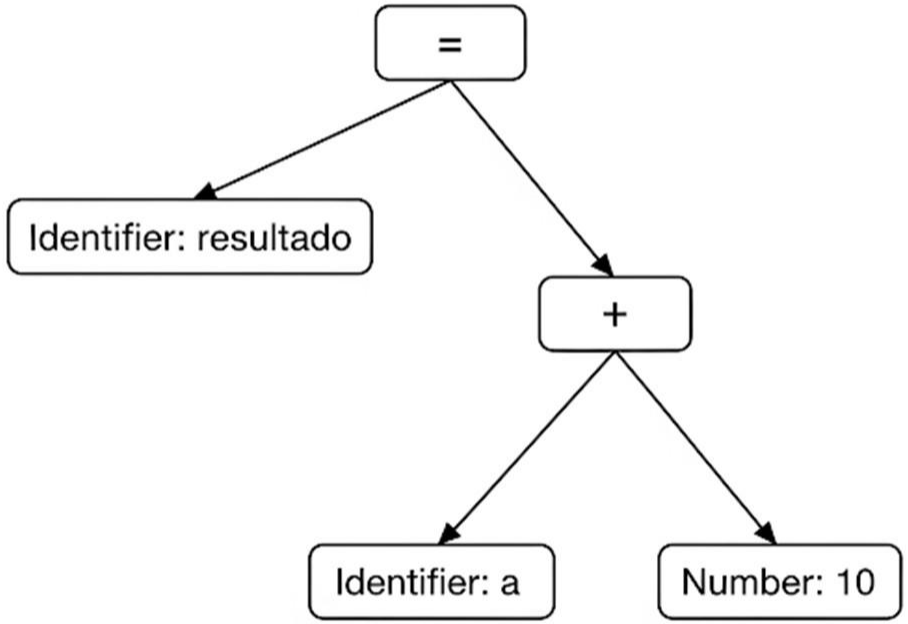
Exemplo de Aplicação
AST - Abstract Syntax Tree

Demonstração de como uma Árvore de Sintaxe Abstrata (AST - Abstract Syntax Tree) modelando uma simples instrução de atribuição.

```
resultado = a + 10;
```

O compilador organiza essa instrução em árvore para entender a relação entre os operadores e operandos.

```
= (resultado, + (a, 10))
```

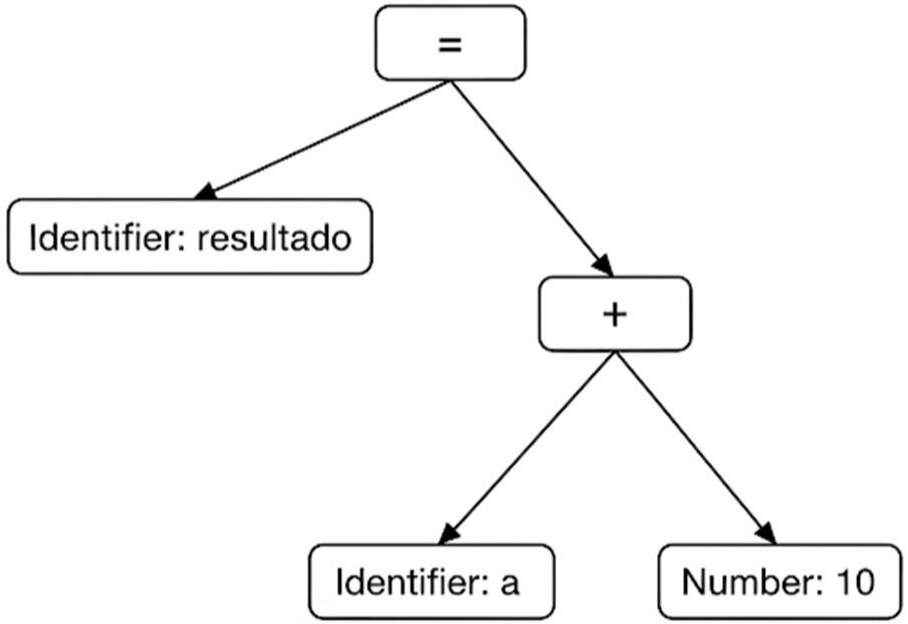


- **A Raiz (Root):** O nó no topo da árvore é o operador de atribuição (=). Isso mostra que a operação principal é atribuir um valor a algo.

Exemplo de Aplicação
AST - Abstract Syntax Tree

- **Os Galhos da Atribuição (Branches):** Uma operação de atribuição sempre tem duas partes:
  - ❑ **Ramo da Esquerda (Destino):** Indica a variável Identifier: resultado que receberá o valor final.
  - ❑ **Ramo da Direita (Origem):** Indica uma outra operação, a Adição (+), possuindo uma outra sub-árvore.
- **As Folhas (Leaves):** Valores concretos ou as variáveis que não precisam de mais nenhuma decomposição.

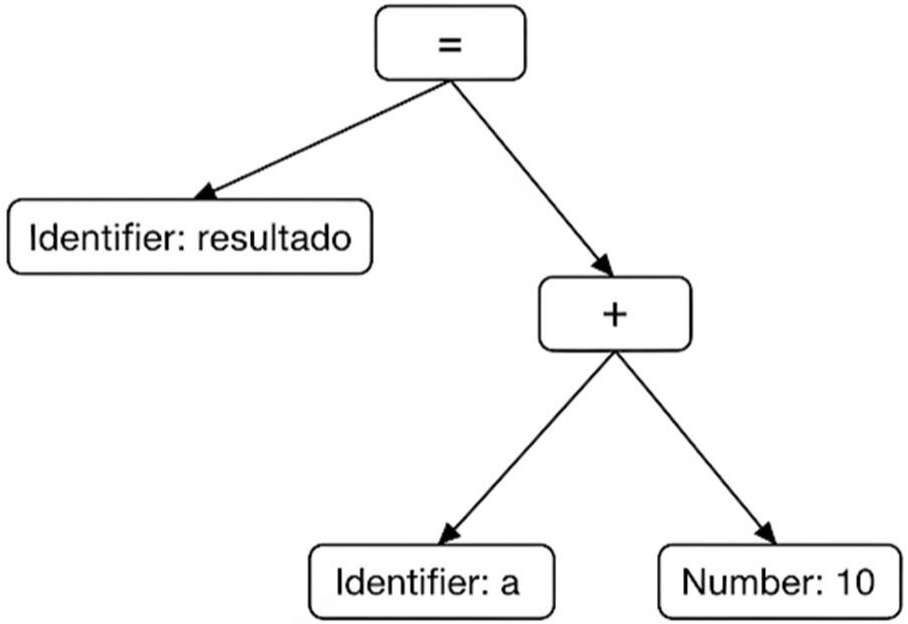
Identifier: resultado Identifier: a Number: 10



Percurso em Profundidade

Depth-First Search – DFS Pós - Ordem

O algoritmo mais usado por compiladores na leitura das Árvores de Sintaxe Abstrata é o Percurso em Profundidade (Depth-First Search - DFS) em Pós-Ordem (Post-Order DFS). Esse algoritmo é inerentemente recursivo (usa estrutura em pilha). Ele explora o mais fundo possível ao longo de cada galho antes de retroceder (backtracking).

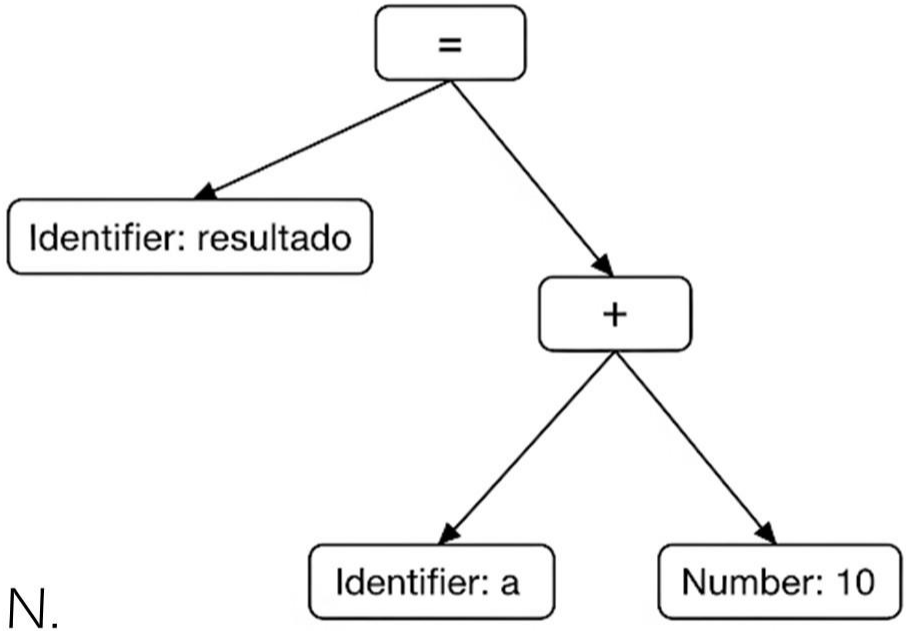


- ❑ **Varredura:** O algoritmo começa sua varredura pela raiz até as folhas.
- ❑ **Processamento:** O algoritmo processa as folhas primeiro, de baixo para cima, até a raiz.

Percurso em Pós-Ordem DFS

Depth-First Search Pós - Ordem

O percurso em pós-ordem é um algoritmo de percurso em profundidade (Depth-First Search - DFS) para estruturas de dados em árvore. Sua característica definidora é que um nó é processado somente após todos os seus nós descendentes terem sido completamente percorridos e visitados.



A ordem de operações para um dado nó N é:

1. Percorrer recursivamente a sub-árvore esquerda de N.
2. Percorrer recursivamente a sub-árvore direita de N.
3. Visitar (processar) o próprio nó N.

Front-End

Fases de Análise

Análise Sintática

Contexto Computacional

É o processo de verificar como os tokens se encaixam para formar "frases" estruturadas e hierárquicas, de acordo com as regras gramaticais da linguagem, gerando como resultado uma Árvore de Sintaxe Abstrata (AST - Abstract Syntax Tree).

Exemplos de Verificação

1. Ponto e Vírgula Faltando: `int x = 5`
2. Parênteses, Chaves ou Colchetes Desbalanceados: `if (x > 5 { printf("maior"); }`
3. Operadores Mal Posicionados: `int y = 5 + * 10;`
4. Palavras-chave em Contextos Inválidos: `int for = 10;`

Processo Geral

Front-End (Fases de Análise)

1. Análise Léxica (Scanning): Lê o código-fonte caractere por caractere e o agrupa em uma sequência de "tokens" (as menores unidades de significado da linguagem).

- **Entrada:** `resultado = a + 10;`
- **Saída:** Uma sequência de tokens como: `IDENTIFICADOR(resultado)`, `OPERADOR(=)`, `IDENTIFICADOR(a)`, `OPERADOR(+)`, `NÚMERO(10)`, `PONTUAÇÃO(;)`

2. Análise Sintática (Parsing): É o processo de verificar como os tokens se encaixam para formar "frases" estruturadas e hierárquicas, de acordo com as regras gramaticais da linguagem, gerando como resultado uma Árvore de Sintaxe Abstrata (AST - Abstract Syntax Tree).

- **Entrada:** A sequência de tokens da fase anterior.
- **Saída:** Uma AST (Abstract Syntax Tree).

Processo Geral

Front-End (Fases de Análise)

3. Análise Semântica: Percorre a AST para verificar o significado lógico do código. É aqui que os erros de tipo são pegos.

A frase "A cadeira chutou a bola" é sintaticamente perfeita, mas semanticamente não faz sentido (cadeiras não chutam).

- ❑ **Entrada:** A AST.
- ❑ **Saída:** A AST validada e anotada com informações de tipo.

Exemplos de verificação:

- Uma variável foi declarada antes de ser usada?
- Você está tentando somar um número com uma string? ("olá" + 5)
- Uma função foi chamada com o número correto de argumentos?

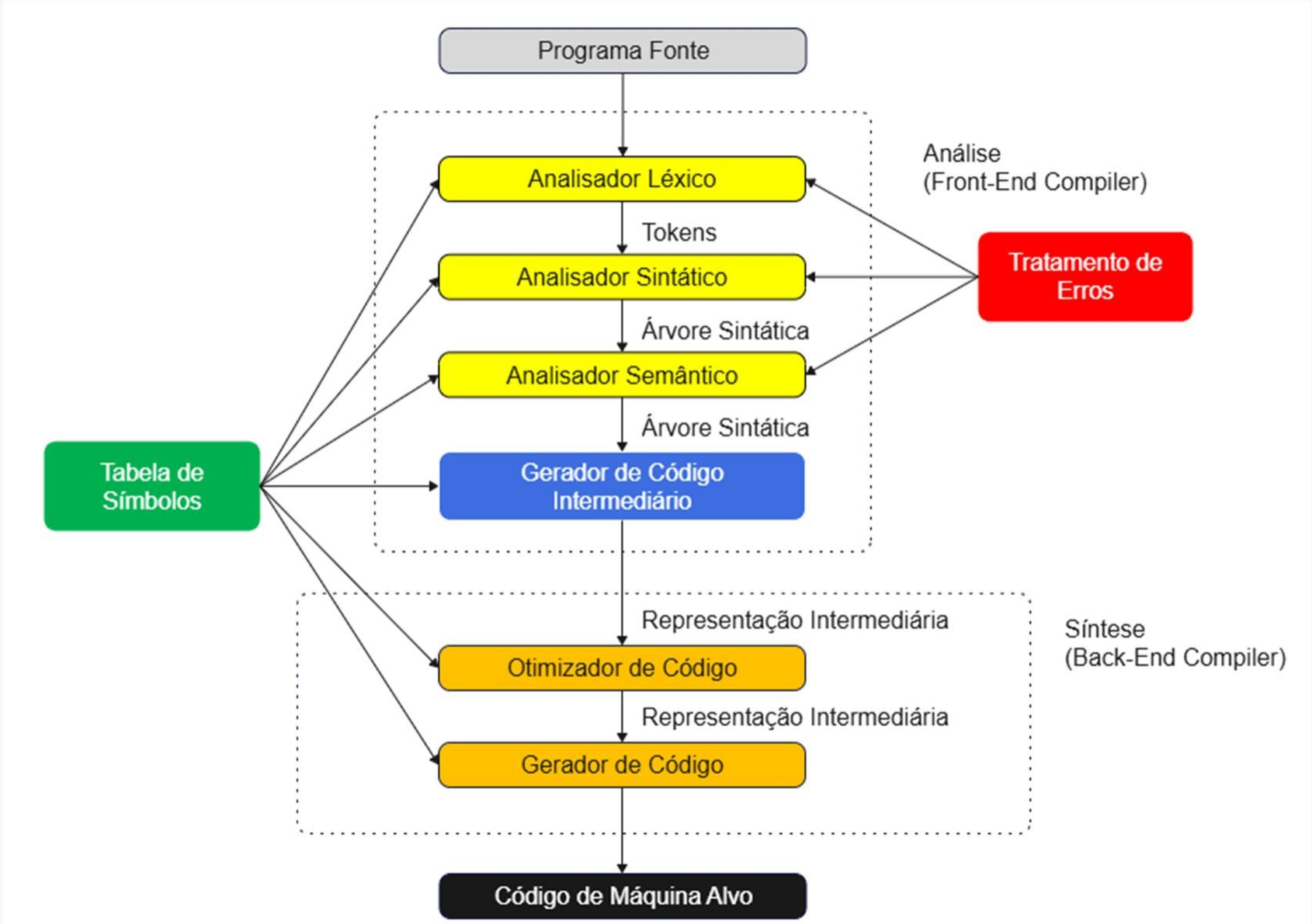
Processo Geral

Front-End (Fases de Análise)

4. Geração de Código Intermediário (IR): Esta é a última fase do Front-End. Ela traduz a AST validada para uma representação de código mais simples, genérica e independente da máquina.

- **Entrada:** A AST validada.
- **Saída:** O Código Intermediário (IR).

A principal responsabilidade do **Front-End** é ler e entender completamente a linguagem de programação de origem. Ele realiza as análises léxica, sintática e semântica para garantir que o código seja válido. O **Código Intermediário (IR)** é o **produto final** dessa análise, uma representação completa e validada do programa que é independente da máquina de destino. A geração do IR é o último ato do Front-End antes de seu trabalho ser considerado concluído.



Compiladores

Processo Back-End

Prof. Alessandro Borges

Especialista em Gestão de Projetos

alessandroborgesoficial

Objetivos de Aprendizagem

- Entender um Programa Objeto
- Identificação das Etapas Básicas da Compilação.
- Demonstrar as fases do processo de compilação.

Compilação

Front-End (Fases de Síntese)

O Back-End de um compilador é a parte do processo que transforma a representação lógica e abstrata do seu programa em algo concreto, otimizado e executável por um processador.

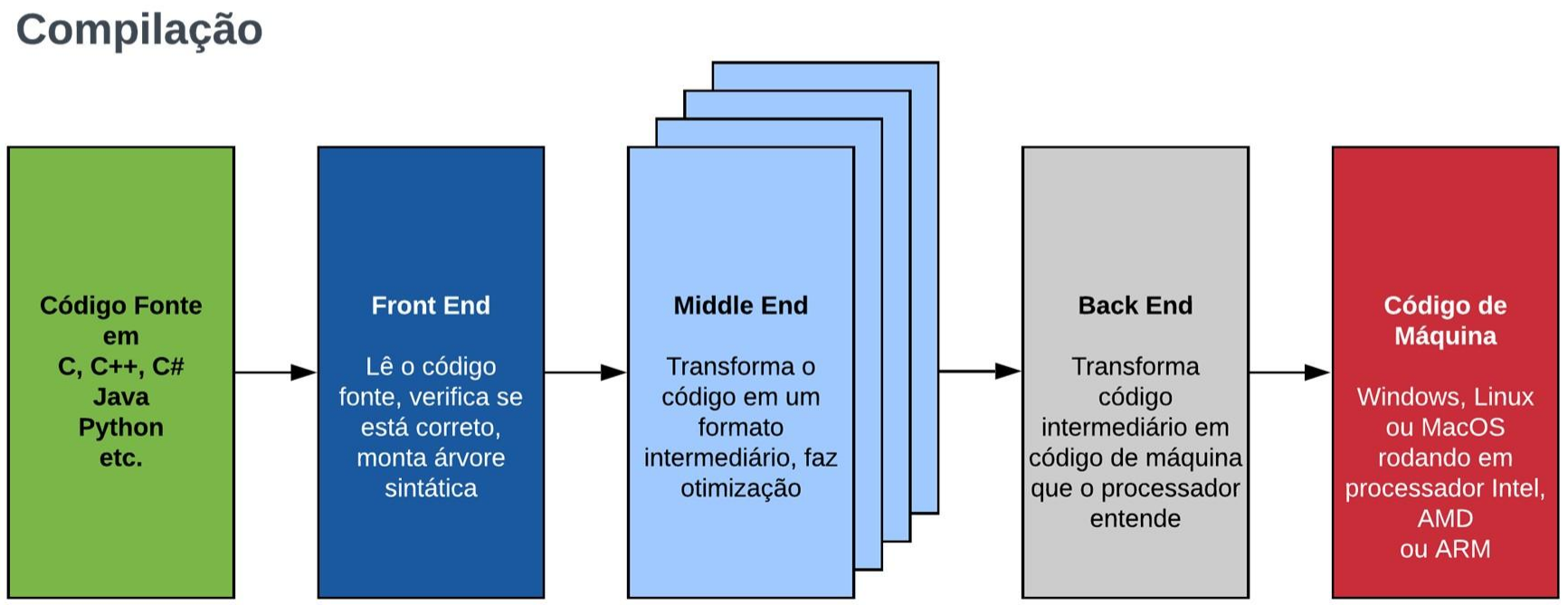
Enquanto o Front-End se preocupa em entender a linguagem de programação, o Back-End se preocupa em dominar a arquitetura do hardware.

O Back-End (ou fase de síntese) é o conjunto de fases de um compilador que recebe uma representação intermediária (IR) do programa, otimiza-a e a traduz para o código de máquina específico da arquitetura de destino.

- **Entrada:** Código Intermediário (IR), gerado pelo Front-End.
- **Saída:** Código de máquina final (geralmente em um arquivo objeto: .o ou .obj).
- **Característica Principal:** É dependente da máquina de destino (sabe se está gerando código para um Intel x86-64, um ARM de celular, etc.), mas independente da linguagem de origem (não importa se o IR veio de um código C++, Rust ou Swift).

O Back-End

Pense no Back-End como um engenheiro especialista em hardware que recebe uma planta universal (o IR) e tem a tarefa de construir o edifício mais eficiente e performático possível usando os materiais e as regras de uma localidade específica (a arquitetura do processador).



Processo Geral

Front-End (Fases de Síntese)

1. Otimização de Código (O "Middle-End"): Esta é uma das fases mais complexas e importantes. O compilador aplica uma série de transformações no Código Intermediário para torná-lo mais rápido e/ou menor, sem alterar seu resultado funcional.

- ❑ **Entrada:** O Código Intermediário.
- ❑ **Saída:** Uma versão otimizada do Código Intermediário.

Exemplos de otimização:

- **Remoção de Código Morto:** Elimina variáveis e trechos de código que nunca são usados.
- **Cálculo de Constantes:** Se o código tem `x = 2 + 3;` , o compilador substitui por `x = 5;`.
- **Inlining de Funções:** Substitui uma chamada de função pequena pelo corpo da própria função para evitar a sobrecarga da chamada.

Processo Geral

Front-End (Fases de Síntese)

2. Geração de Código Final: A fase final. Traduz o Código Intermediário otimizado para a linguagem de montagem (Assembly) ou diretamente para o código de máquina da arquitetura de destino.

- **Entrada:** O Código Intermediário otimizado.
- **Saída:** Código de máquina final, que é então reunido em um arquivo objeto.

1ª Responsabilidade (Geração de Código): Fase onde o IR otimizado é convertido para as instruções específicas da arquitetura de destino.

a) **Seleção de Instruções:** Para cada operação no IR (ex: a = b + c), o compilador deve escolher a melhor sequência de instruções de máquina para realizá-la. Arquiteturas modernas oferecem várias formas de fazer a mesma coisa, e o compilador precisa escolher a mais eficiente.

Processo Geral

Front-End (Fases de Síntese)

2. Geração de Código Final: A fase final. Traduz o Código Intermediário otimizado para a linguagem de montagem (Assembly) ou diretamente para o código de máquina da arquitetura de destino.

- **Entrada:** O Código Intermediário otimizado.
- **Saída:** Código de máquina final, que é então reunido em um arquivo objeto.

1ª Responsabilidade (Geração de Código): Fase onde o IR otimizado é convertido para as instruções específicas da arquitetura de destino.

b) Alocação de Registradores: O IR assume um número infinito de "registradores virtuais". O processador real tem um número muito limitado de registradores físicos (ex: 16 ou 32). Esta tarefa, uma das mais complexas, decide quais variáveis devem residir nesses registradores em cada ponto do programa.

Processo Geral

Front-End (Fases de Síntese)

2. Geração de Código Final: A fase final. Traduz o Código Intermediário otimizado para a linguagem de montagem (Assembly) ou diretamente para o código de máquina da arquitetura de destino.

- **Entrada:** O Código Intermediário otimizado.
- **Saída:** Código de máquina final, que é então reunido em um arquivo objeto.

1ª Responsabilidade (Geração de Código): Fase onde o IR otimizado é convertido para as instruções específicas da arquitetura de destino.

c) **Ordenação de Instruções:** Reorganiza a ordem das instruções de máquina geradas para otimizar o uso do pipeline do processador, garantindo que todas as unidades de execução da CPU estejam ocupadas o máximo de tempo possível.

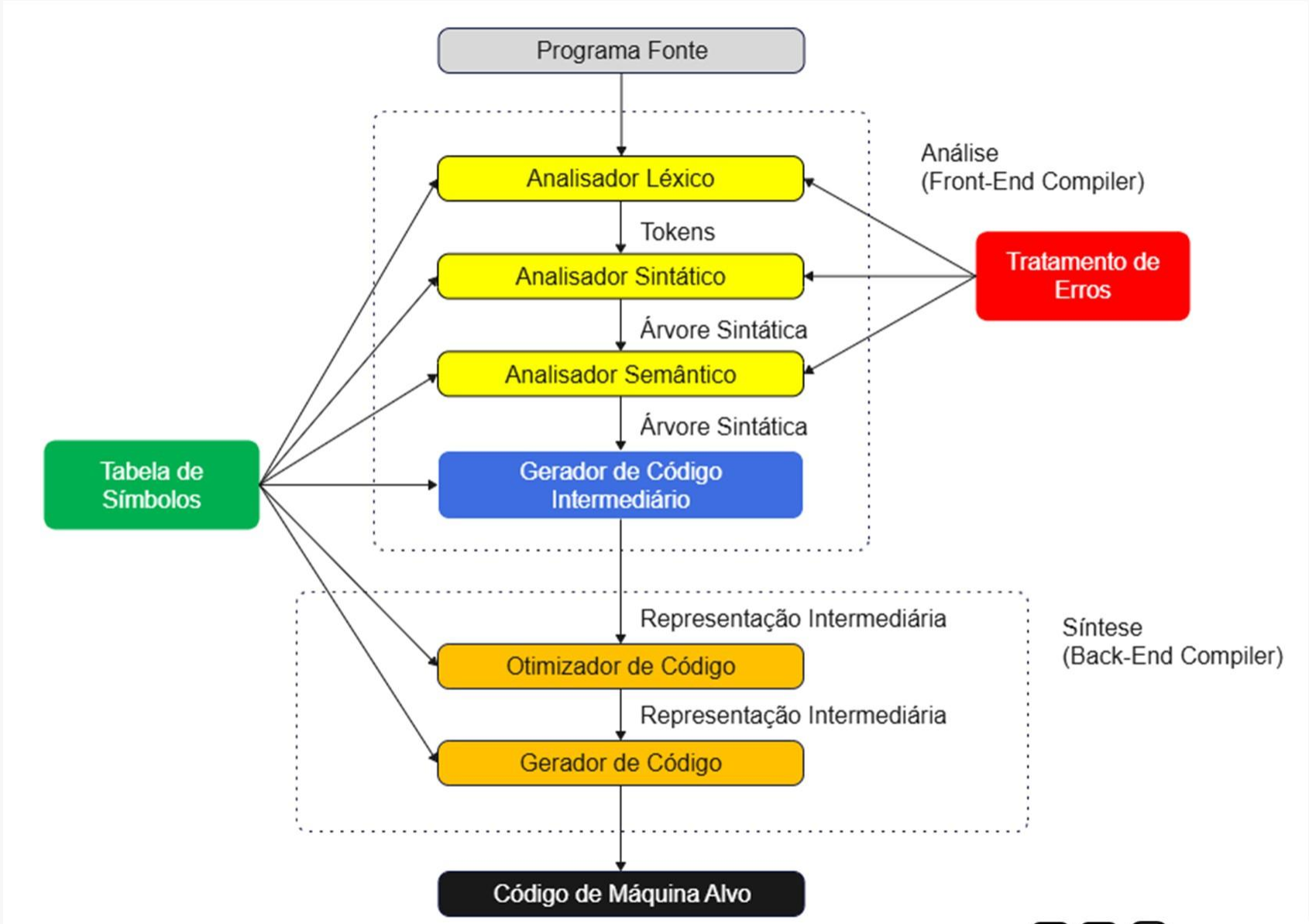
Processo Geral

Front-End (Fases de Síntese)

2ª Responsabilidade (Geração do Código Objeto Final): O Back-End utiliza a sequência final de instruções de máquina e a formata em um arquivo objeto (.o ou .obj). Este arquivo contém:

- O código de máquina binário (na seção .text).
- Dados inicializados (na seção .data).
- Informações para o linker (o programa que une vários arquivos objeto), como a tabela de símbolos e os pontos de relocação.

Em resumo, o Back-End é o responsável por pegar uma representação lógica e universal do seu programa e transformá-la em um conjunto de instruções altamente otimizado e perfeitamente adaptado para extrair o máximo de performance do hardware específico em que ele será executado.



Compiladores

O Fluxo de Trabalho

O desenvolvimento de compiladores modernos com frameworks redefine as responsabilidades. O desenvolvedor foca em projetar a lógica e a sintaxe únicas da linguagem no Front-End. As complexas tarefas de otimização e geração de código final são delegadas ao Back-End do framework.

1. **Foco do Usuário:** Trabalhar no Front-End do Compilador.
    - Definir a Gramática: Criar o arquivo de regras que descreve a estrutura dos tokens (léxico) e das sentenças (sintaxe).
    - Análise Léxica & Sintática: Usar uma ferramenta (como ANTLR), alimentada pela gramática, para transformar o código-fonte em uma AST.
    - Análise Semântica: Escrever um algoritmo de percurso na sua AST, provavelmente usando um Visitor Pattern e verificar tipos, escopo de variáveis, etc.

Compiladores

O Fluxo de Trabalho

2. **Ainda com Usuário:** Criar uma Ponte para o Framework
   - Traduzir a AST (Abstract Syntax Tree) para o Código Intermediário (IR) do LLVM (Framework).
   - Criar algoritmo que percorra a AST e emita as instruções correspondentes em LLVM IR.
3. **O Framework Assume o Controle:** Trabalha no Back-End do Compilador.
   - Recebimento: O LLVM recebe o programa representado como LLVM IR.
   - Otimização: O LLVM executa dezenas de passes de otimização, usando seus próprios algoritmos de percurso e análise sobre o IR.
   - Geração de Código: O Back-End do LLVM pega o IR otimizado e faz o seu próprio percurso final (usando lógicas como Pós-Ordem) para gerar o código de máquina nativo e altamente otimizado para a arquitetura alvo (x86-64, ARM, etc.).

Fontes

- https://www.cadcobol.com.br/
- https://craftinginterpreters.com/chunks-of-bytecode.html
- https://habr.com/ru/articles/191766/
- https://www.cambridge.org/us/features/052182060X/