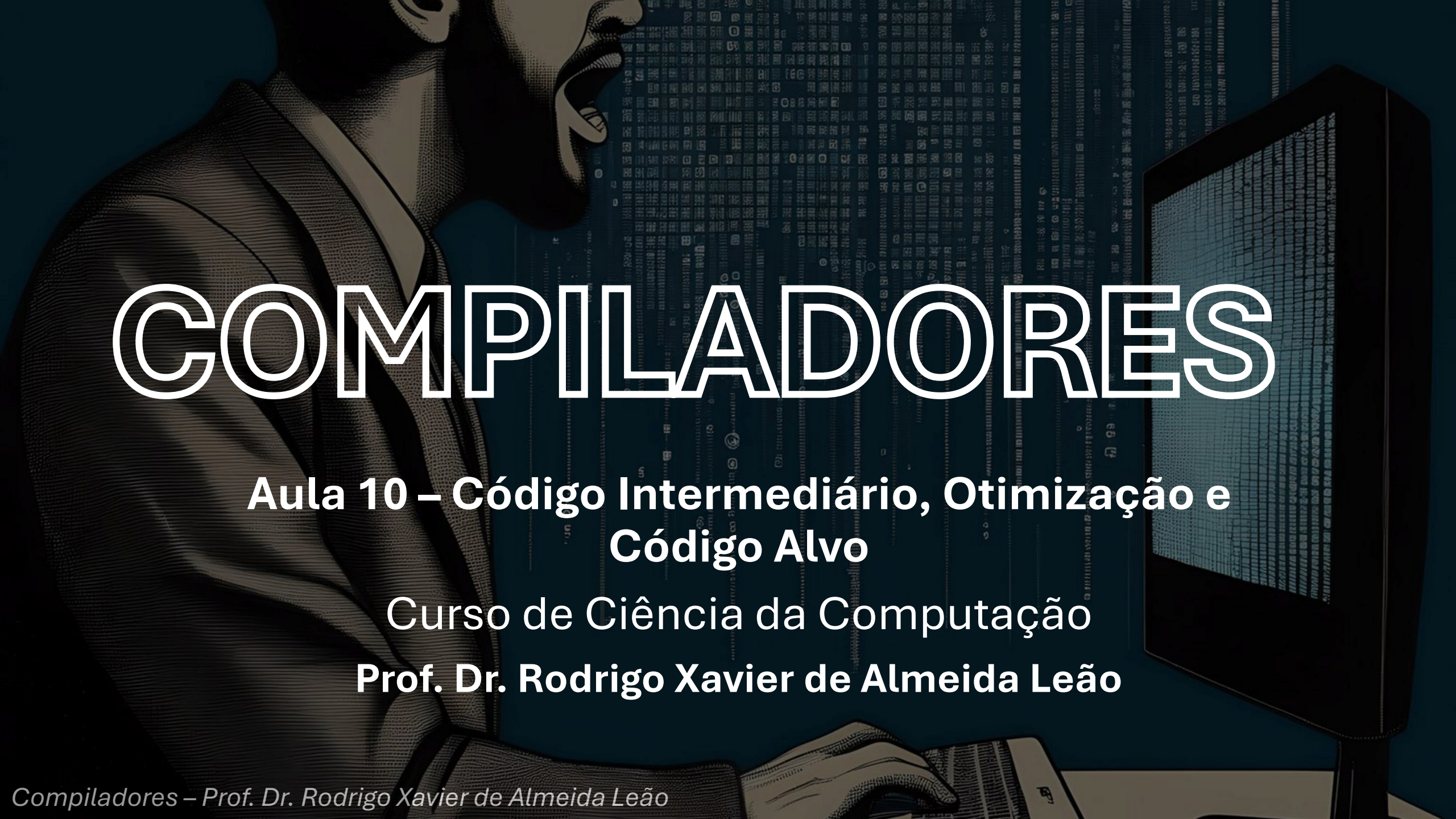
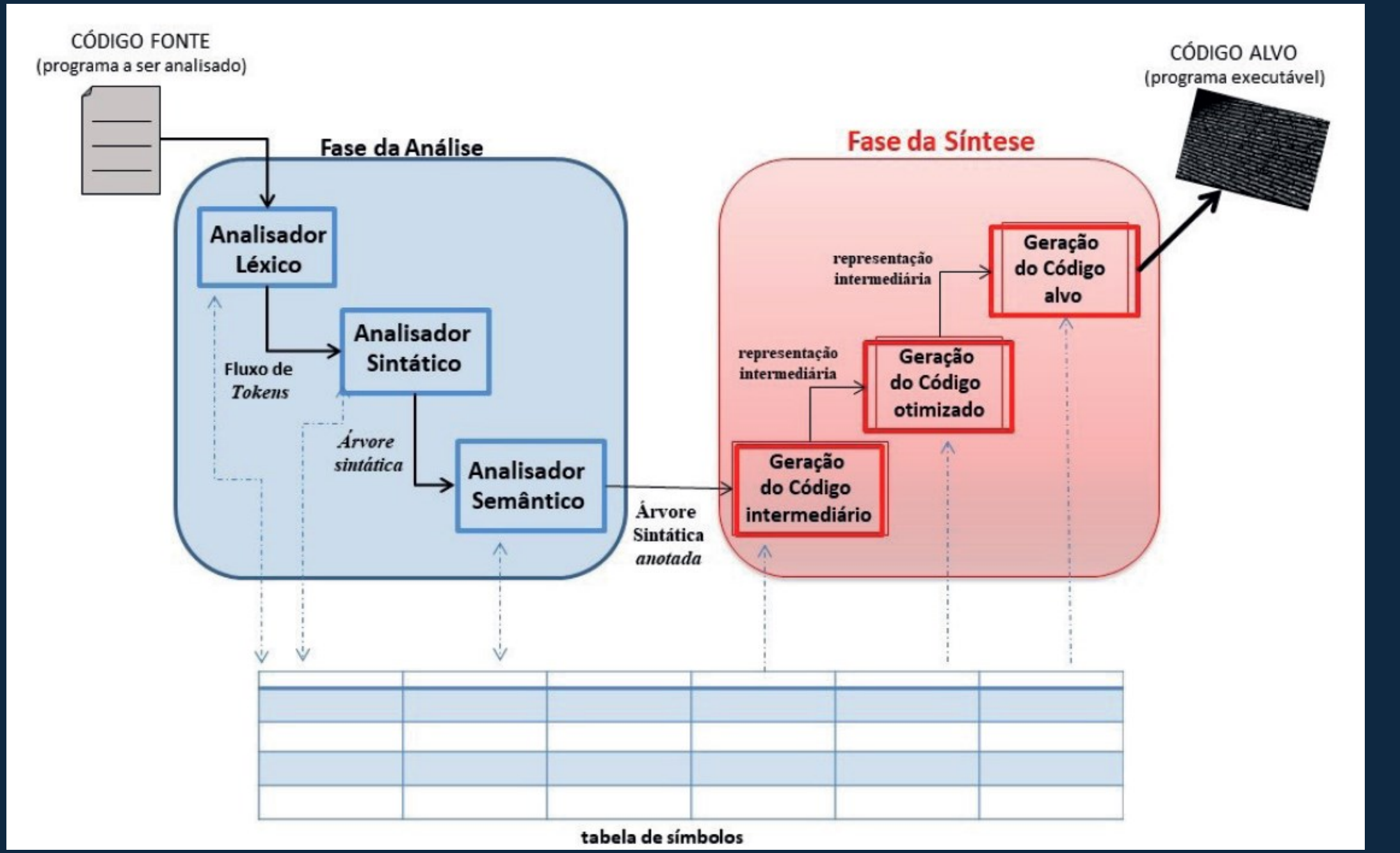


COMPILADORES

Aula 10 – Código Intermediário, Otimização e Código Alvo

Curso de Ciência da Computação

Prof. Dr. Rodrigo Xavier de Almeida Leão


CÓDIGO FONTE
(programa a ser analisado)
CÓDIGO ALVO
(programa executável)
Fase da Análise
Fase da Síntese
Analisador Léxico
Analisador Sintático
Analisador Semântico
Fluxo de Tokens
Árvore sintática
Árvore Sintática anotada
representação intermediária
representação intermediária
Geração do Código alvo
Geração do Código otimizado
Geração do Código intermediário
tabela de símbolos

Arquitetura de Computadores

1. **Conjunto de Instruções (ISA - Instruction Set Architecture)**:

   Define as instruções que a CPU pode executar, incluindo operações aritméticas, lógicas, de controle de fluxo e de memória.

   Exemplo: x86, ARM, MIPS, RISC-V.

   Compiladores devem gerar código que seja compatível com o ISA da máquina de destino.

2. **Registros**:

   Unidades de armazenamento rápidas dentro da CPU.

   A quantidade e o tipo de registros (propósito geral, ponto flutuante, etc.) afetam a alocação de registros feita pelo compilador.

4. **Pipeline e Superescalaridade**:

   Arquiteturas modernas usam pipelines e podem ser superescalares, permitindo a execução de múltiplas instruções em paralelo.

   Compiladores devem otimizar o código para maximizar a eficiência do pipeline e minimizar stalls.

5. **Caches e Hierarquia de Memória**:

   A organização da hierarquia de memória (caches, memória principal, etc.) influencia a geração de código.

   Compiladores devem tentar otimizar o uso da memória cache para melhorar o desempenho.

Código Intermediário

A tradução para um código intermediário, como veremos nesta seção, traz uma grande vantagem para o projetista do compilador, pois não haverá a necessidade de escrever um compilador que traduza diretamente a representação da árvore sintática anotada para o código de máquina, o que é um processo altamente complexo e, se assim o fizéssemos, seria necessário escrever um compilador para cada tipo de arquitetura. Entretanto, se subdividimos a fase de síntese, criando um código intermediário que não precisa estar atrelado a uma determinada arquitetura, quando for preciso gerar o código final para diversas plataformas, apenas a tradução do código intermediário precisará ser feita, o que é um processo muito mais simples.

Seja:

M a linguagem de máquina;

A uma representação intermediária;

L uma linguagem de alto nível qualquer;

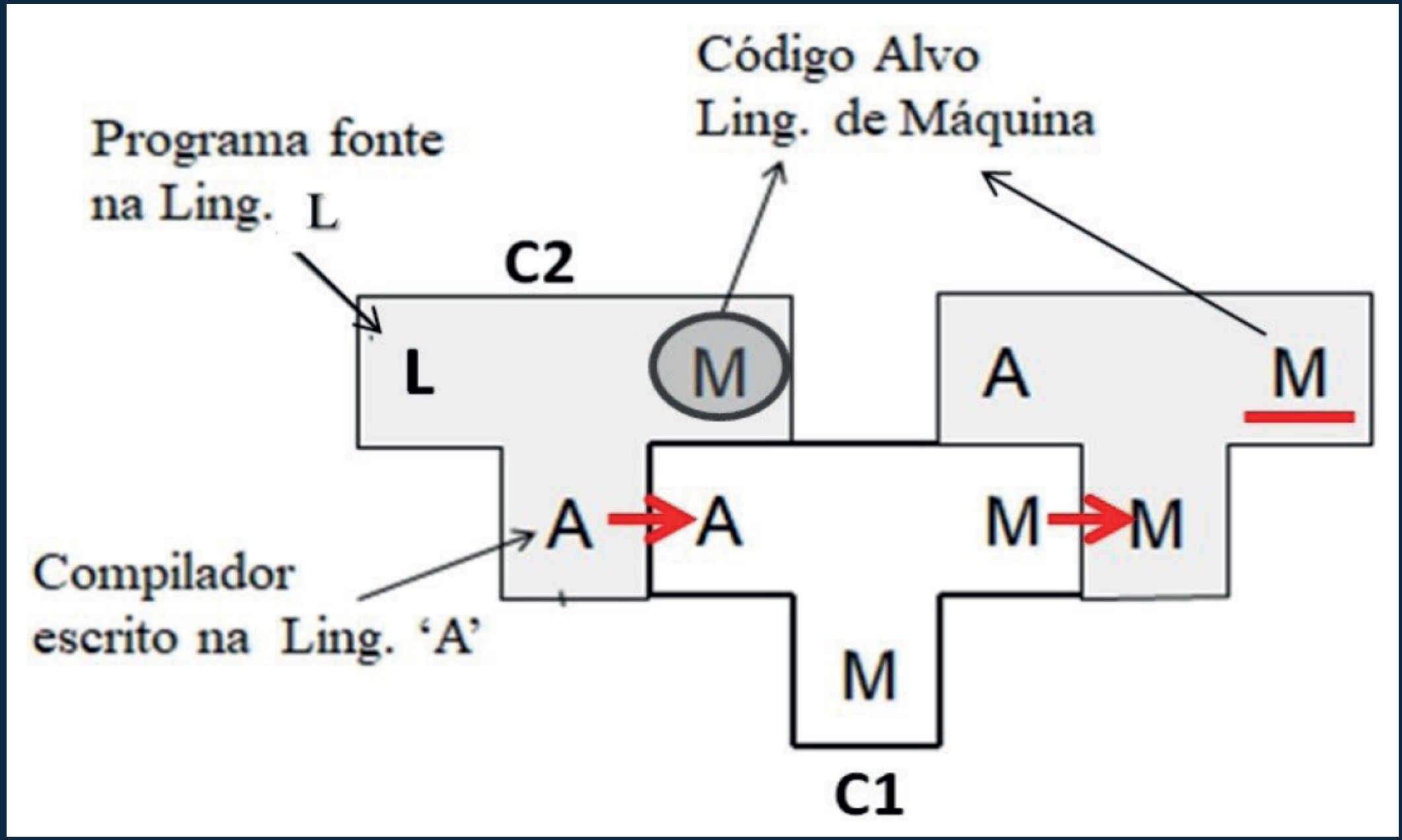
C1 o compilador de A, escrito em M e gera código alvo M; e

C2 o compilador de L, escrito em A.

Como podemos gerar o código alvo M para o compilador C2?

Você se lembra do processo ***Bootstrapping***?

Por meio do processo de *Bootstrapping*, é possível escrever um compilador em qualquer linguagem de programação a partir da existência de dois compiladores.

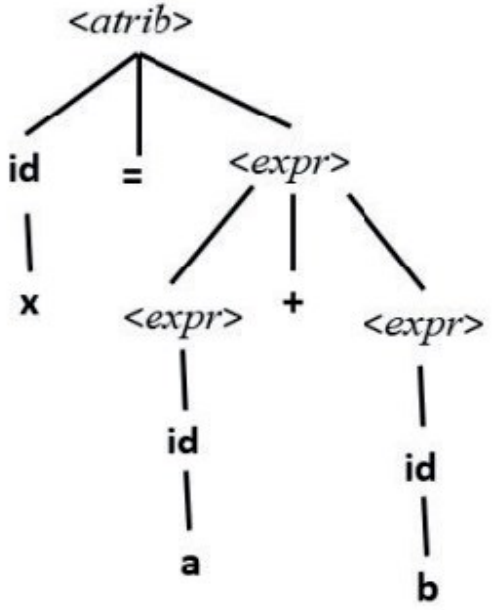

Código Alvo
Ling. de Máquina
Programa fonte
na Ling. L
C2
L
M
A
M
A
A
M
M
Compilador
escrito na Ling. 'A'
M
C1

FORMAS DO CÓDIGO INTERMEDIÁRIO

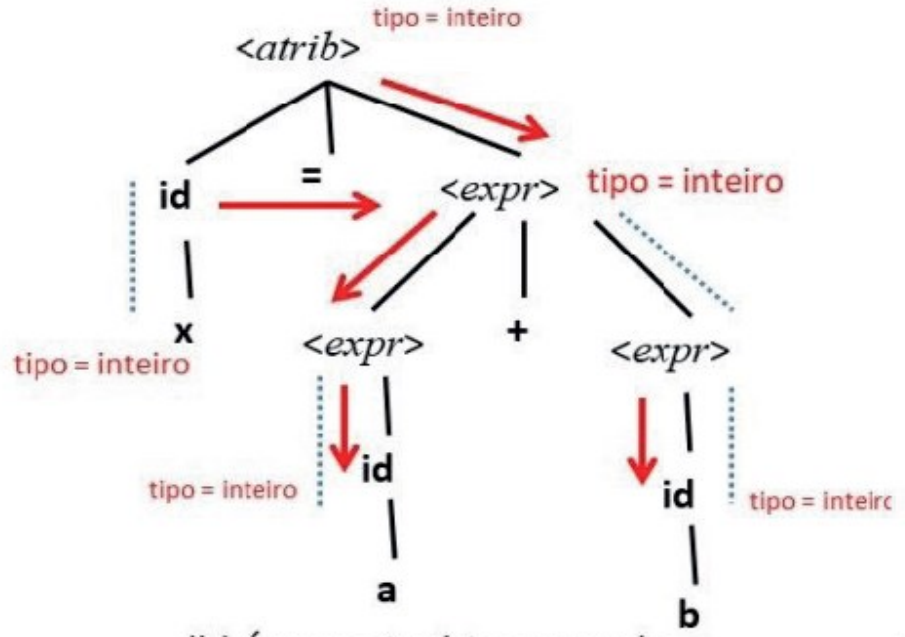
Figura 4.3 | Representações de árvores

gramática <atrib> → id = <expr>
<expr> → <expr> + <expr> | id

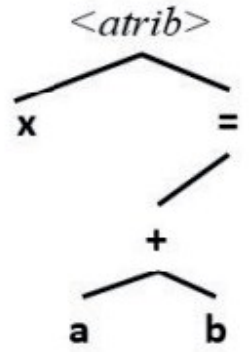
sentença
x = a + b



(a) – árvore de derivação



(b) Árvore sintática anotada



(C) Árvore Sintática Abstrata (AST)

notação pós-fixa: x a b + =

No nosso exemplo, o compilador precisará simplificar a instrução A * (B + C) / D. A notação pós-fixa insere a operação depois das variáveis ou número. No caso da expressão infixa A * (B + C) / D, a notação pós-fixa para essa expressão será A B C + * D / . Parece muito complicado à primeira vista, mas para a máquina será fácil, pois os valores são armazenados em uma pilha e, ao encontrarem o operador, sempre dois valores serão desempilhados e o cálculo será feito. Isso simplifica as instruções e não é necessário o uso de parênteses. Veja na Figura 4.4 o esquema de funcionamento da pilha e acompanhe seu funcionamento pelo algoritmo a seguir.

expressão pós-fixa `3 4 2 * 1 5 - / +`:

`3` -> Pilha: `3`

`4` -> Pilha: `3 4`

`2` -> Pilha: `3 4 2`

`*` -> Pop `4` e `2`, push `8` -> Pilha: `3 8`

`1` -> Pilha: `3 8 1`

`5` -> Pilha: `3 8 1 5`

`-` -> Pop `1` e `5`, push `-4` -> Pilha: `3 8 -4`

`/` -> Pop `8` e `-4`, push `-2` -> Pilha: `3 -2`

`+` -> Pop `3` e `-2`, push `1` -> Pilha: `1`

Esse algoritmo lê a sentença de entrada A * (B + C) / D e escreve essa sentença na notação pós-fixa, a qual permite uma conversão mais fácil para o código alvo (executável). O algoritmo para conversão deverá:

1. Criar uma pilha vazia;
2. Ler os *tokens* da sentença de entrada, neste caso temos apenas variáveis, operadores e parênteses, e, a cada *token*,:
    a. Se encontrar um identificador, colocar diretamente na saída;
    b. Se encontrar um OPERADOR (+, -, * ou /):
        i. Enquanto a pilha não estiver vazia e no topo existir um operador com prioridade >= ao encontrado, desempilhar o operador e colocá-lo na saída;

ii. Em seguida, empilhar o operador encontrado na pilha;

c. Se encontrar um abre-parêntese '(', empilhar;

d. e encontrar um fecha-parêntese ')':

i. Enquanto não encontrar o abre-parêntese '(' na pilha, desempilhar o operador e colocá-lo na saída, exceto o abre-parêntese, que deverá ser apenas desempilhado;

3. Ao concluir a varredura da sentença, certificar-se de esvaziar a pilha, desempilhando os elementos diretamente na saída.

Figura 4.4 | Teste do algoritmo para a expressão A * (B + C) / D; conversão de uma notação infixa para pós-fixa

	ENTRADA	prioridade	PILHA	SAIDA	acoes do algoritmo com relacao a entrada
1	A		**vazia**	A	(2.a) escreve direto na saida
2	*	3	*	A	(2.b) pilha vazia empilha
3	(	1	(*	A	(2.c) sempre empilha
4	B			AB	(2.a) escreve direto na saida
5	+	2	+ (*	AB	(2.b) empilha, pois topo da pilha com prioridade menor
6	C			ABC	(2.a) escreve direto na saida
7	)		*	ABC+	(2.d) desempilha e escreve na saida até achar (
8	/	3	/	ABC+*	(2.b) prioridade igual. Desempilha * e empilha /
9	D			ABC+*D	(2.a) escreve direto na saida
10	vazio		**vazia**	**ABC+*D/**	(3) esvazia a pilha e escreve na saida

Converter uma expressão aritmética para código de três endereços (TAC - Three Address Code) envolve a decomposição da expressão em uma série de instruções simples, cada uma contendo no máximo três operandos (dois operandos e uma variável de destino). O código de três endereços é uma forma intermediária usada em compiladores para facilitar a geração de código de máquina.

Código fonte alto nível	Código em 3 endereços

`x = a * ( b + 40 )`	`t0 = int(40)`
	`t1 = b + t0`
	`t2 = a * t1`
	`x = t2`

Expressão Pós-Fixa: `` `3 4 2 * 1 5 - / +` ``

Código de Três Endereços:

1. `` `t1 = 4 * 2` `` // Multiplicação
2. `` `t2 = 1 - 5` `` // Subtração
3. `` `t3 = t1 / t2` `` // Divisão
4. `` `t4 = 3 + t3` `` // Adição

Portabilidade:

O código intermediário é mais abstrato do que o código de máquina e pode ser gerado de maneira independente da arquitetura de hardware. Isso facilita a portabilidade, permitindo que o mesmo código-fonte seja compilado para diferentes plataformas de hardware com menor esforço.

Simplificação do Processo de Compilação:

A geração de código intermediário simplifica o design do compilador. Em vez de criar um compilador específico para cada par linguagem-alvo/hardware, os desenvolvedores de compiladores podem criar um único front-end que gera código intermediário e vários back-ends que traduzem o código intermediário para diferentes arquiteturas de hardware.

Otimização:

O código intermediário oferece uma representação mais fácil de analisar e transformar, o que é ideal para otimizações. Técnicas de otimização como eliminação de redundâncias, reordenação de instruções e melhor uso dos registradores podem ser aplicadas mais facilmente no código intermediário do que no código de máquina ou no código-fonte.

Facilidade de Geração de Código:

Geração de código de máquina diretamente do código-fonte pode ser complicada e propensa a erros. Ao introduzir um estágio intermediário, o processo de geração de código é dividido em partes menores e mais manejáveis, onde cada parte pode ser otimizada e verificada separadamente.

Modularidade:

A introdução de um código intermediário modulariza o processo de compilação. Isso significa que diferentes partes do compilador podem ser desenvolvidas, testadas e melhoradas de forma independente. Por exemplo, o front-end do compilador (que trata da análise léxica e sintática) pode ser separado do back-end (que trata da geração de código).

Otimização

otimização, um código bom. Aqui, cabe uma desmistificação na construção de compiladores: a fase de otimização do código é muito mais complexa que a de geração do código, pois gerar (traduzir) é associar os comandos da linguagem de alto nível às instruções de baixo nível, percorrendo os nós da AST, enquanto otimizar requer a análise de várias propriedades, de acordo com o tipo de estrutura gerada. Vamos analisar quais são estas propriedades, segundo Aho (2007):

1) Uma transformação precisa preservar o significado dos programas – durante o processo de otimização, não podemos comprometer a execução do programa, gerando códigos que não existiam no programa fonte. Por exemplo, ao otimizar o código, é gerada uma passagem que pode em algum momento resultar em um erro que não existia no programa original, logo, essa otimização não deverá ser feita.

2) Uma transformação precisa acelerar o programa que será gerado, **na média**, e por um fator que possa ser mensurável – a eficiência média é o que importará nesse ponto da otimização, pois nem sempre o fato de aumentarmos o número de instruções no processo de conversão de uma instrução da linguagem de alto-nível para o código de máquina, ocupando mais memória, resulta em menor eficiência na média. Esse é um caso no qual se tenta alcançar uma eficiência média, mensurável pela taxa de crescimento (*Big-Theta*).

3) Uma transformação deve valer o esforço – isso significa que o tempo que o compilador irá gastar para gerar o código alvo tem que ser compensado na eficiência do programa gerado. Podemos dizer que não vale fazer a otimização se "a emenda for pior do que o soneto", isto é, se o resultado do esforço que será gasto na otimização for muito maior que o resultado que será obtido com o programa executável gerado.

python

```python
def sum_array(arr):
    sum = 0
    for i in range(len(arr)):
        sum += arr[i]
    return sum

if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5]
    result = sum_array(arr)
    print(f"Sum: {result}")
```

Eliminação de Redundâncias e Funções Embutidas

Usar a função embutida `sum` do Python para somar elementos de uma lista é uma otimização natural:

python Copiar código

```python
if __name__ == "__main__":
    arr = [1, 2, 3, 4, 5]
    result = sum(arr)
    print(f"Sum: {result}")
```

plaintext

```
t1 = 0              # sum = 0
t2 = 0              # i = 0
L1: if t2 >= len(arr) goto L2
t3 = arr[t2]        # t3 = arr[i]
t1 = t1 + t3        # sum += arr[i]
t2 = t2 + 1         # i++
goto L1
L2: return t1
```

E para a versão otimizada com a função embutida:

plaintext

```
t1 = sum(arr)    # Directly use the built-in sum function
return t1
```

```c
#include <stdio.h>

int sum_array(int arr[], int n) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    return sum;
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int result = sum_array(arr, 5);
    printf("Sum: %d\n", result);
    return 0;
}
```

Desdobramento de Loop (Loop Unrolling):

Desdobrar o loop para reduzir a sobrecarga de controle do loop.

```
t1 = 0              // sum = 0
t2 = 0              // i = 0
L1: if t2 >= n goto L2
t3 = arr[t2]        // t3 = arr[i]
t1 = t1 + t3        // sum = sum + arr[i]
t2 = t2 + 1         // i++
goto L1
L2: return t1
```

```
t1 = 0          // sum = 0
t2 = 0          // i = 0
L1: if t2 >= n goto L2
t3 = arr[t2]    // t3 = arr[i]
t1 = t1 + t3    // sum = sum + arr[i]
t2 = t2 + 1     // i++
goto L1
L2: return t1
```

Desdobramento de Loop (Loop Unrolling):

Desdobrar o loop para reduzir a sobrecarga de controle do loop.

```
t1 = 0          // sum = 0
t2 = 0          // i = 0
L1: if t2 >= n goto L2
t3 = arr[t2]    // t3 = arr[i]
t1 = t1 + t3    // sum = sum + arr[i]
t2 = t2 + 1     // i++
if t2 >= n goto L2
t4 = arr[t2]    // t4 = arr[i]
t1 = t1 + t4    // sum = sum + arr[i]
t2 = t2 + 1     // i++
goto L1
L2: return t1
```

Inline de Funções:

Substituir chamadas de função por seu corpo diretamente no código principal para evitar a sobrecarga de chamada de função.

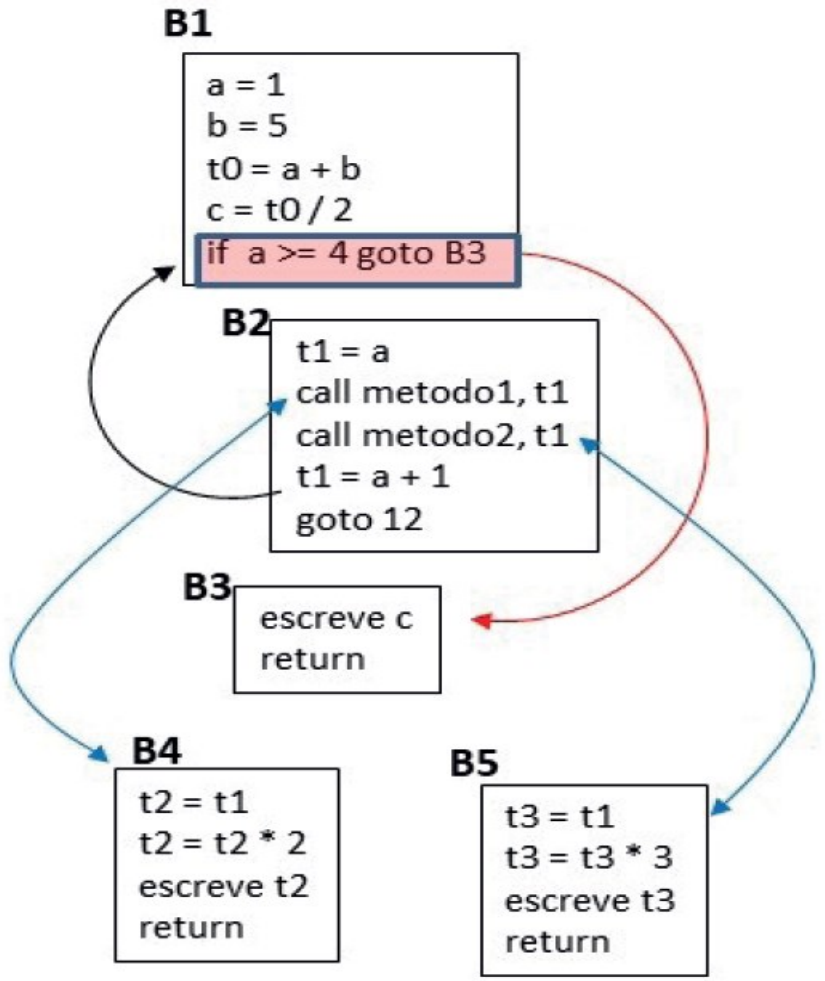
c

```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum += arr[i];
    }
    printf("Sum: %d\n", sum);
    return 0;
}
```

```
public class Exemplo {
    public static void main(String[] args){
        double a, b, c;
        a = 1;
        b = 5;
        c = ( a + b ) / 2;
        while (a < 4 ){
            Exemplo.metodo1(a);
            a++;
        }
    }
    public static void metodo1(double x){
        System.out.println(x*2);
        Exemplo.metodo2(x);
    }
    public static void metodo2(double x){
        System.out.println(x*3);
    }
}
```

Grafos de Fluxo de Controle (CFG) e Blocos Básicos (BB)



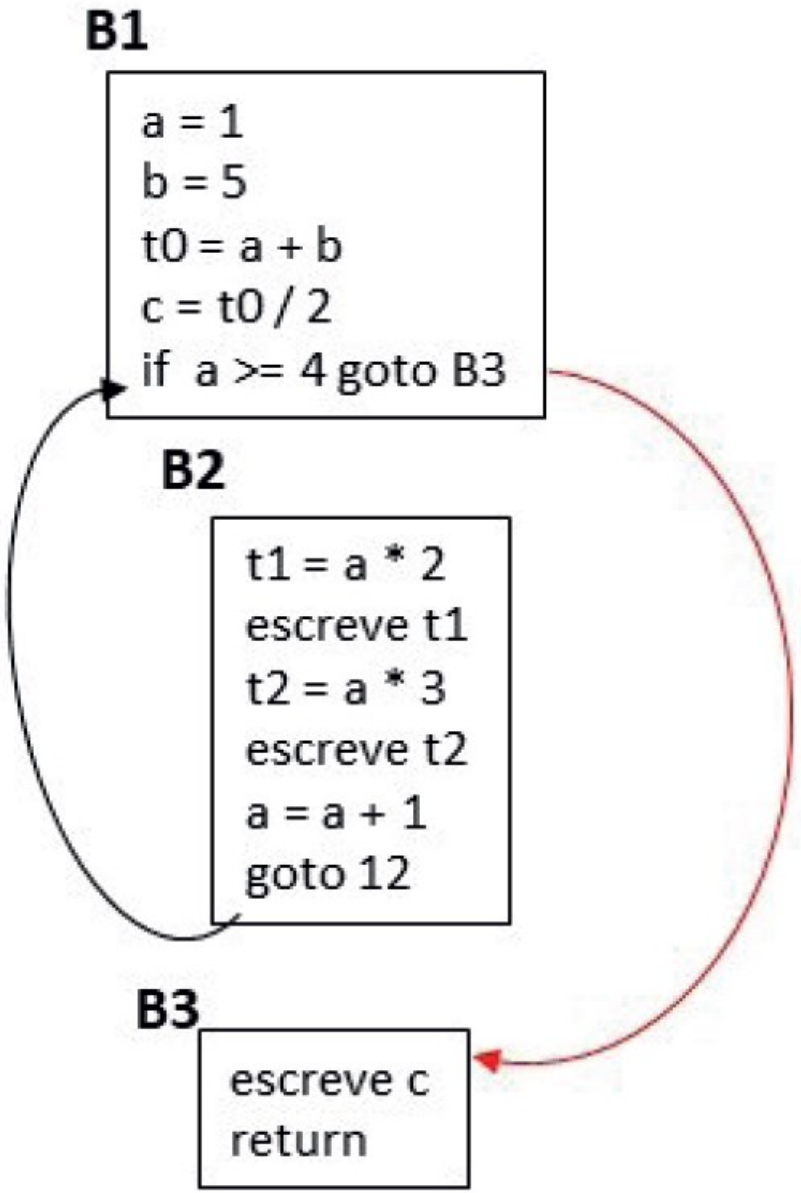
Bloco b4

```
t2 = t1
t2 = t2 * 2
escreve t2
return
```

Bloco B4 - otimizado

```
t2 = t1 * 2
escreve t2
return
```

Grafos de Fluxo de Controle Otimizado



Geração do Código Alvo

A geração do código deverá converter a RI em um código que possa ser executado na máquina de destino. As ações desta fase da compilação, conforme definidas por Cooper (2014), são:

1. Seleção de instruções – consiste em selecionar uma sequência de instruções da máquina-alvo que implemente as operações de RI;

2. Escalonamento das instruções – consiste em definir a ordem na qual as operações deverão ser executadas;
3. Alocação de registros – consiste em definir quais valores deverão residir nos registradores em cada ponto do programa.

Código Assembly Sem Otimizações (`-O0`)

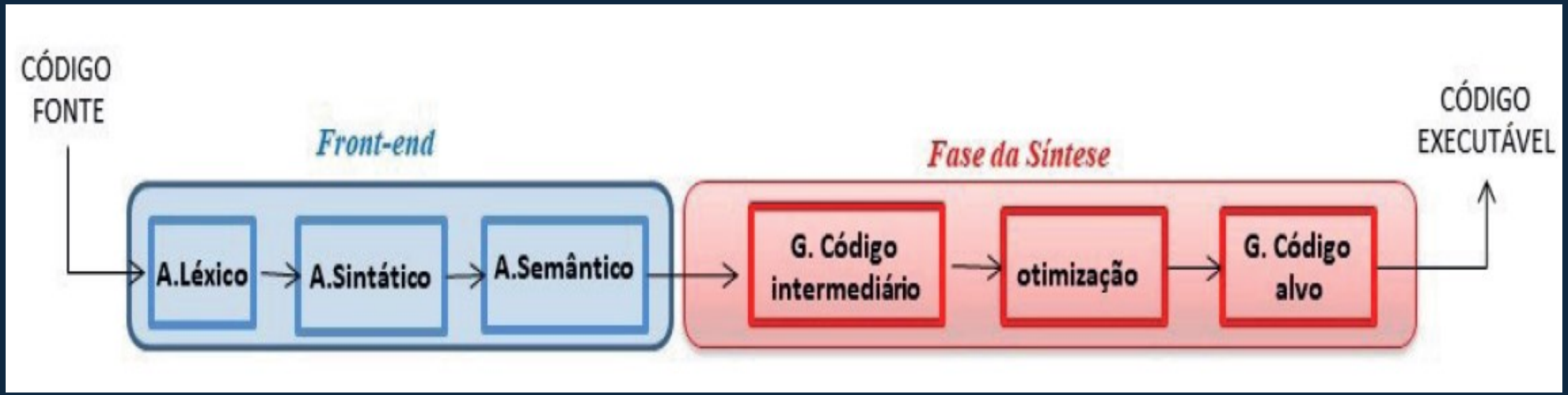
assembly

```
_main:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $32, %rsp
    movl    $0, -4(%rbp)
    movl    $0, -8(%rbp)
    jmp     .L2
.L3:
    movl    -8(%rbp), %eax
    cltq
    movl    arr(,%rax,4), %edx
    addl    %edx, -4(%rbp)
    addl    $1, -8(%rbp)
.L2:
    cmpl    $5, -8(%rbp)
    jl      .L3
    movl    -4(%rbp), %eax
    movl    %eax, -12(%rbp)
    movl    -12(%rbp), %eax
    movl    %eax, %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movl    $0, %eax
    leave
    ret
```

Código Assembly Otimizado (`-O3`)

assembly

```
_main:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    $15, %esi
    movl    $.LC0, %edi
    movl    $0, %eax
    call    printf
    movl    $0, %eax
    popq    %rbp
    ret
```


CÓDIGO FONTE
Front-end
A.Léxico
A.Sintático
A.Semântico
Fase da Síntese
G. Código intermediário
otimização
G. Código alvo
CÓDIGO EXECUTÁVEL