

Construção de Compiladores

Prof. Dr. Daniel Lucrédio

DC - Departamento de Computação

UFSCar - Universidade Federal de São Carlos

Tópico 01 - Introdução

Referências bibliográficas

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. Compiladores: Princípios, Técnicas e Ferramentas (2a. edição). Pearson, 2008.

Kenneth C. Loudon. Compiladores: Princípios E Práticas (1a. edição). Cengage Learning, 2004.

Terence Parr. The Definitive Antlr 4 Reference (2a. edição). Pragmatic Bookshelf, 2013.

“um programa que recebe como entrada um programa em uma linguagem de programação – a linguagem fonte – e o traduz para um programa equivalente em outra linguagem – a linguagem objeto”

Aho, Lam, Sethi e Ullman (2008). Compiladores: princípios, técnicas e ferramentas

O que são compiladores?

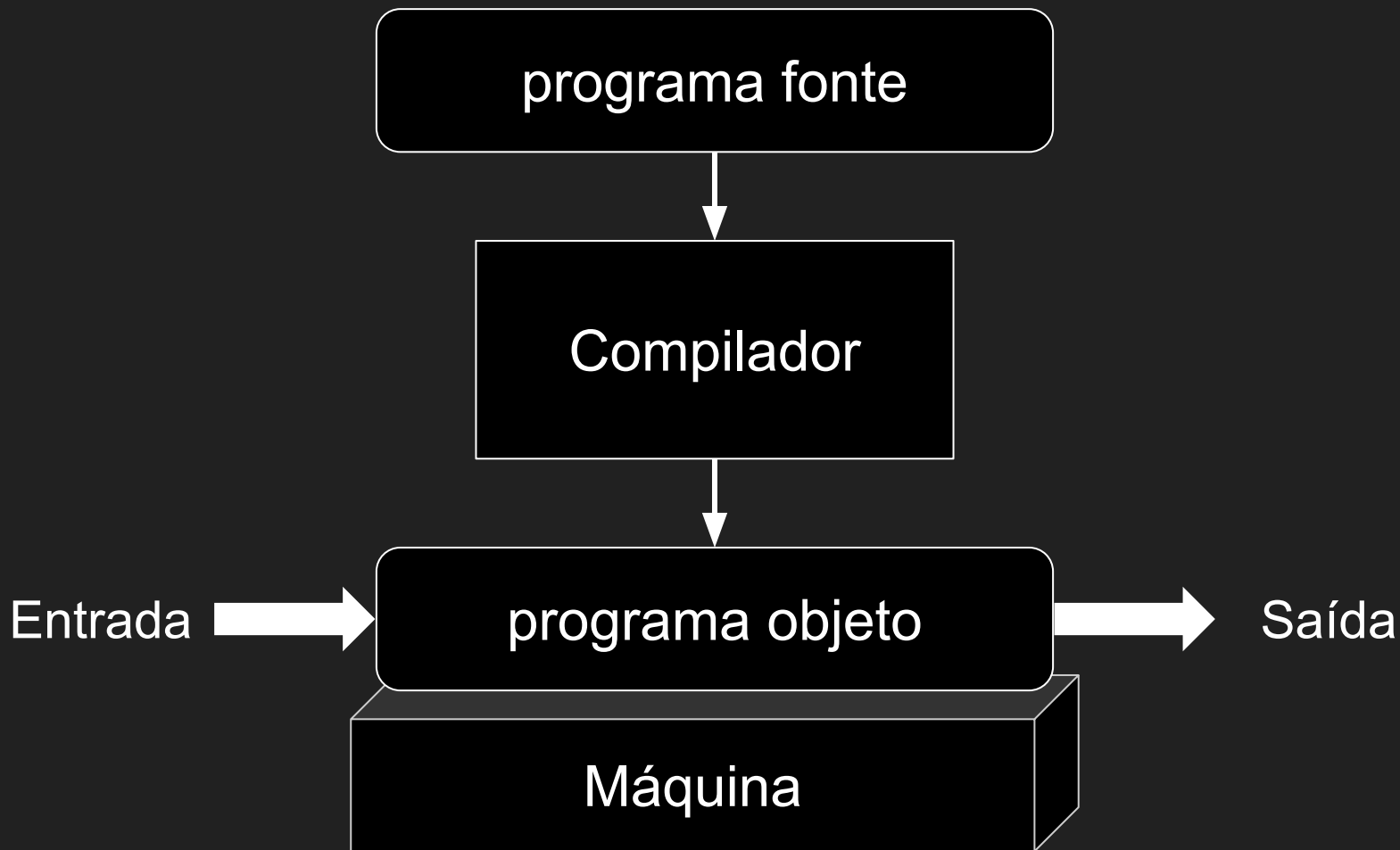
- Compilador é um processador de linguagem
 - Objetivo é TRADUZIR
 - Entrada: Linguagem de programação (C, C++, Java)
 - Saída: Linguagem de máquina
 - De forma que o computador consiga “entender”
- Por que existem?
 - A linguagem de máquina é muito “ruim” de programar
 - Dizemos que é “de baixo nível”
 - Fica difícil raciocinar sobre ela





O que são compiladores?

- O compilador é basicamente isso:
 - Alguém (software) que consegue traduzir nossos desejos em linguagem que o computador entende
 - O compilador entende a linguagem de alto nível
 - E a traduz para uma linguagem de baixo nível
 - Seu trabalho então termina
- De forma que podemos “ensinar” ao computador alguma tarefa
 - E ele faz a tarefa - sozinho

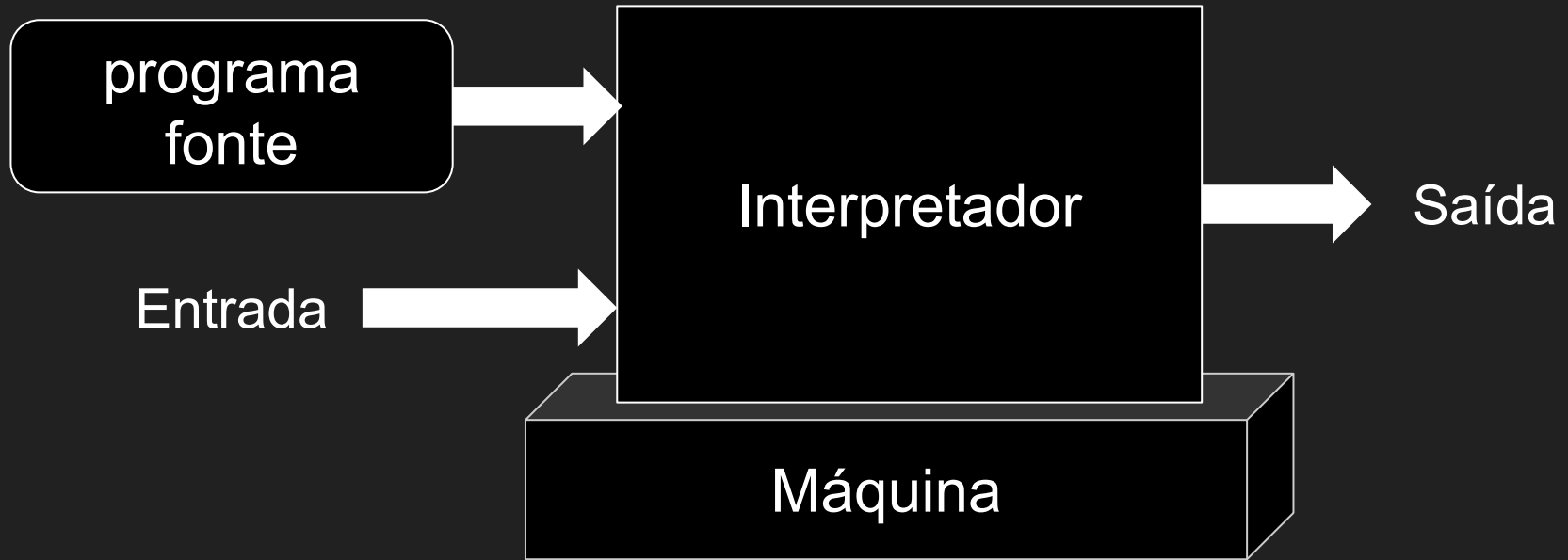


Sobre compiladores

- Também possuem uma função “extra”
- Ajudar o “programador”
 - Apontando erros no programa fonte
- Ex:
 - `if (a > 10 a += 3;`
 - Falta um fecho parêntese depois do 10
 - Como você sabe disso?
 - Como uma máquina pode saber disso?
- Não é uma tarefa simples

Interpretadores

- Existe outro tipo de processador de linguagem
 - Chamado de interpretador
 - Como o compilador, o interpretador entende a linguagem de alto nível
 - Mas ele mesmo executa as tarefas
 - É como se o treinador de cachorros fosse buscar o jornal
- Outra forma de ver
 - O interpretador traduz o programa fonte diretamente em ações



Compiladores

vs

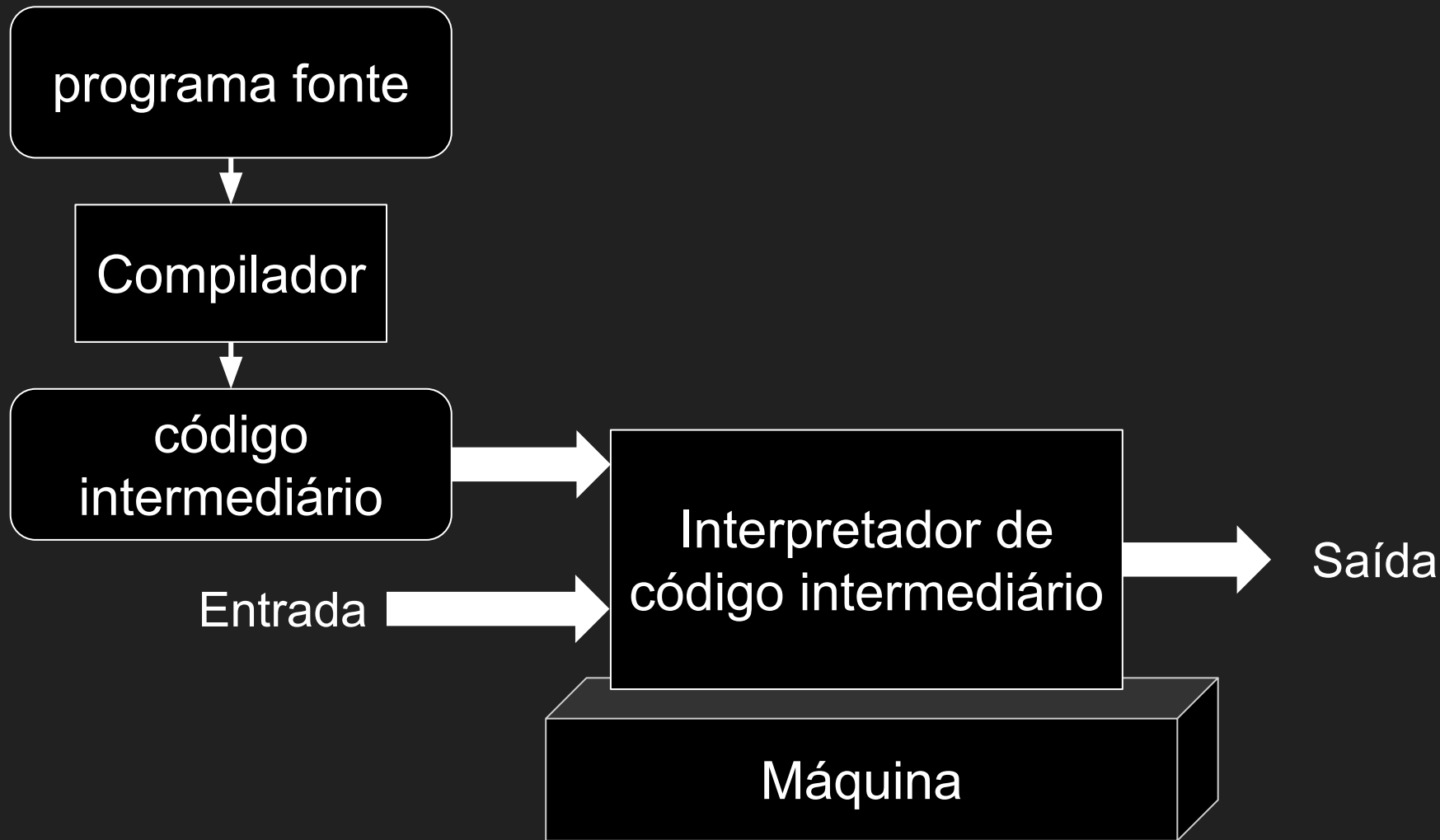
Interpretadores

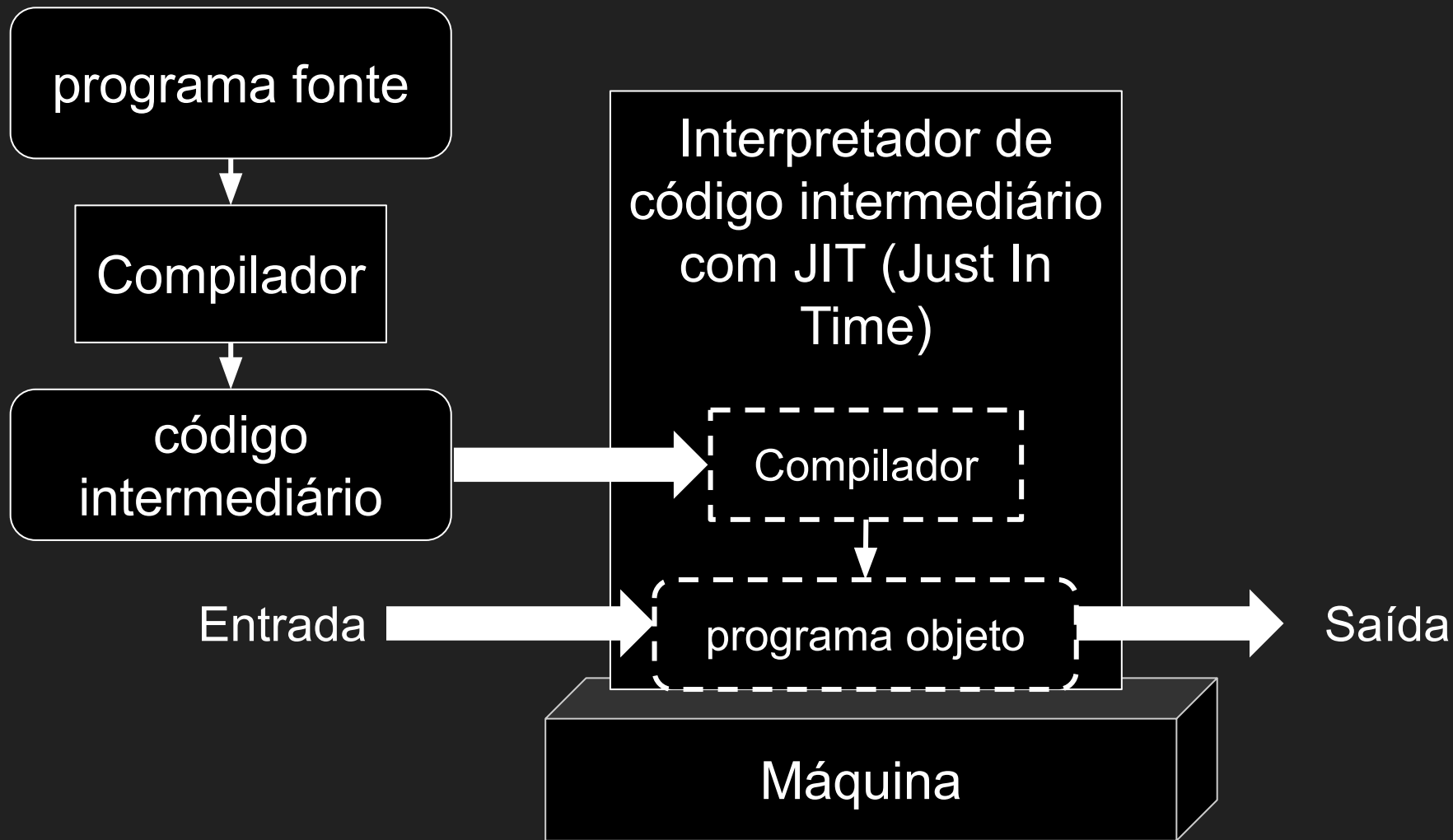
- Mapeamento entrada/saída mais rápido
 - Traduz 1 vez para executar n vezes
- Na tradução, perde-se informação
 - Dificulta o diagnóstico de erros

- Mapeamento entrada/saída mais lento
 - Precisa traduzir toda vez que é executado
- O diagnóstico de erros é normalmente melhor
 - Mais flexível

Interpretadores e compiladores

- É possível combinar as duas abordagens
 - E buscar os benefícios de ambas
- Ex: Java, .NET
 - Usam compilação + interpretação
 - Abordagem híbrida





Compiladores vs interpretadores

- Na disciplina, muitos conceitos são comuns
- Foco é no processo de tradução
 - Compreensão da linguagem fonte
 - Detecção de erros

Compiladores e Linguagens Formais

Compiladores

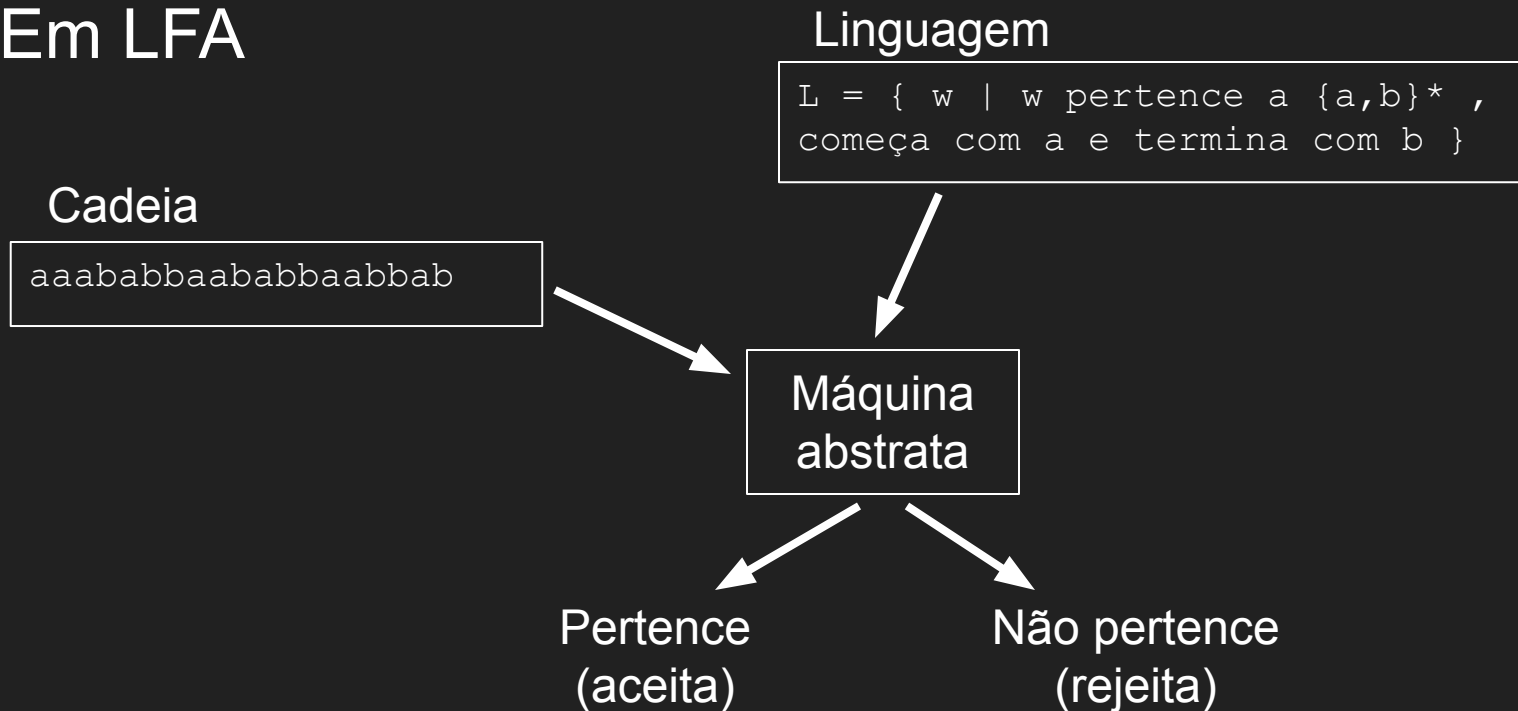
- Normalmente pensamos em linguagens de programação
 - Programa fonte = algoritmo
 - Programa objeto = executável
- Mas compiladores podem ser utilizados em outros contextos
 - SQL
 - Programas SQL não são algoritmos
 - Mas ainda assim existe um compilador (ou interpretador)
 - HTML
 - O compilador (interpretador) no navegador lê o programa (página) HTML e o traduz em ações
 - Que desenham uma página
 - Latex

Resumindo

- Conceito de linguagem
 - É o mesmo que visto em LFA
 - Linguagens são descrições formais de problemas
- Mas aqui, o objetivo é fazer com que o computador entenda a **semântica** da linguagem
 - Ou seja: não basta decidir se uma cadeia faz parte ou não da linguagem
 - É necessário “entender” o que significa a cadeia
 - E traduzi-la para as ações desejadas!

LFA x compiladores

- Em LFA



LFA x compiladores

- Trabalharemos com linguagens livres de contexto
 - Linguagens livres de contexto são bons modelos de programas que tipicamente queremos escrever
- Portanto, precisamos de um ... PDA
 - Modelo abstrato simples (AF com uma pilha)
 - “Fácil” de implementar

LFA x compiladores

- Em LFA

Cadeia

aaabaababbaaaaabaabaabbababb

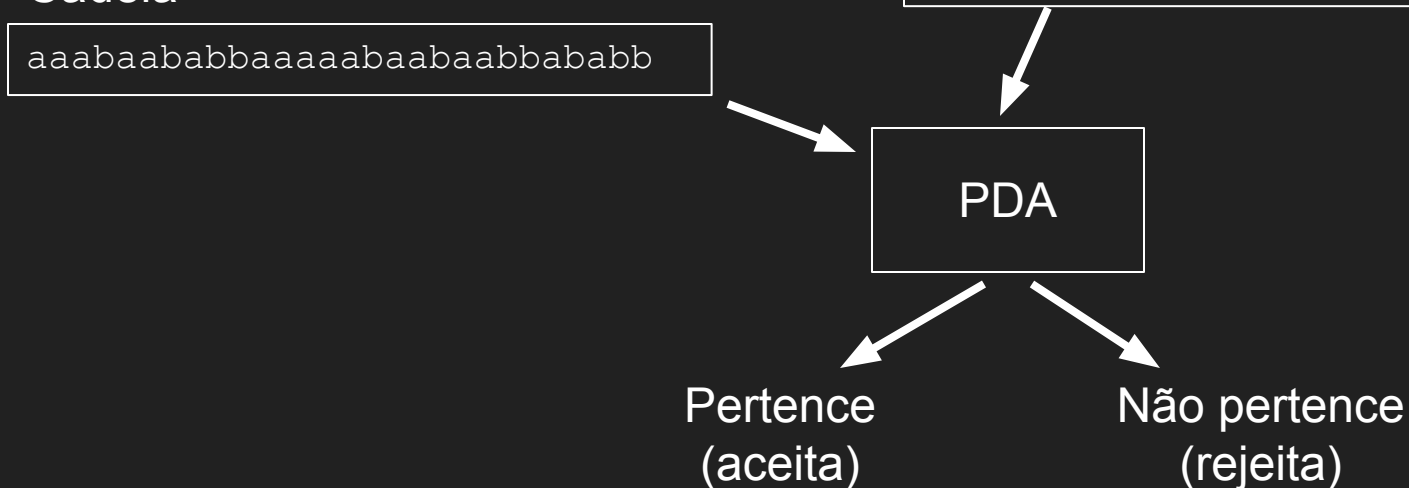
Linguagem (gramática)

$S \rightarrow SaA \mid SbB \mid aaa$
 $A \rightarrow aba$
 $B \rightarrow bab$

PDA

Pertence
(aceita)

Não pertence
(rejeita)



LFA x compiladores

- Em compiladores

Linguagem (gramática)

```
S → Expr;  
Expr → Expr + Expr |  
       Expr - Expr...  
...
```

Cadeia (programa)

```
int a = 2;  
int b = 3  
if(a > b) {  
    System.out.pribtln  
        ("A é maior do que B");  
}
```

Compilador

Pertence
(aceita)
+ ações

Não pertence
(rejeita)
+ erros

```
mov R1, #43F2  
mov R2, #AA3F  
sub R1, R2, R3  
jnz #45FF  
...
```

```
Linha 2: faltou ";"  
Linha 4: não existe "pribtln"
```

LFA x compiladores

- Portanto, um compilador é essencialmente um PDA
- Ele usa uma pilha e estados para reconhecer as cadeias
 - Análise sintática
- E traduz (ou executa) para “ações semânticas”
 - Definidas sobre as regras da linguagem
 - Ex:
 - $\text{Expr} \rightarrow \text{Expr} + \text{Expr} \{ \text{ação de soma} \} \mid \text{Expr} - \text{Expr} \{ \text{ação de subtração} \}$
- Mas tem (sempre tem) um problema

Linguagens não livres de contexto

Considere a seguinte cadeia, em uma linguagem de programação típica:

```
String numero = 0;  
if (nmero > 0) {  
    System.out.println("Nunca vai entrar aqui");  
}
```

Irá acusar erro aqui, pois a variável
nmero não foi declarada

Em algumas LP, variáveis precisam ser declaradas antes de serem utilizadas

Linguagens não livres de contexto

Considere a seguinte cadeia:

```
String numero = 0;  
if (nmero > 0) {  
    System.out.println  
}
```

Em algumas LP, variáveis

É o mesmo caso da linguagem
 $\{ww \mid w \text{ em um alfabeto com mais de um símbolo}\}$

Não é uma linguagem livre de contexto!!

Prova? => Lema do bombeamento!

ca:

ilizadas

Linguagens não livres de contexto

- Outros exemplos: declaração de pacotes, macros, chamada de funções, etc.
- Ou seja, gramáticas livres de contexto não conseguem impor todas as restrições de uma linguagem de programação típica
- Portanto, fica a pergunta: podemos usar um PDA?
 - Refraseando: precisamos de um autômato mais poderoso?
 - Uma máquina de Turing com fita limitada?
 - Um PDA com duas pilhas?
 - Mas o PDA simples é tão ... simples!!
 - Eu queria MUITO usar um PDA simples

Hierarquia de Chomsky

Hierarquia	Gramáticas	Linguagens	Autômato mínimo
Tipo-0	Recursivamente Enumeráveis ou irrestritas	Recursivamente Enumeráveis	Máquinas de Turing
Tipo-1	Sensíveis ao contexto	Sensíveis ao contexto	MT com fita limitada
Tipo-2	Livres de contexto	Livres de contexto	Autômatos de pilha (Push-Down Automata - PDA)
Tipo-3	Regulares (Expressões regulares)	Regulares	Autômatos finitos

E se...

```
int a = 2;  
int b = 3;  
if(a > b) {  
    System.out.println("A é maior do que B");  
}
```



```
TIPO NOME = CONSTANTE;  
TIPO NOME = CONSTANTE;  
if(NOME > NOME) {  
    CLASSE.MEMBRO.METODO(CONSTANTE_STR);  
}
```



```
NOME1 = a  
NOME2 = b  
CONSTANTE1 = 2  
CONSTANTE2 = 3  
CLASSE1 = System  
MEMBRO1 = out  
METODO1 = println  
CONSTANTE_STR1 =  
    "A é maior do que B"
```

E se...

```
int a = 2;  
int b = 3;  
if(a > b) {  
    System.out.println("A é maior do que B");  
}
```

Não é livre
de
contexto!



```
TIPO NOME = CONSTANTE;  
TIPO NOME = CONSTANTE;  
if(NOME > NOME) {  
    CLASSE.MEMBRO.METODO(CONSTANTE_STR);  
}
```

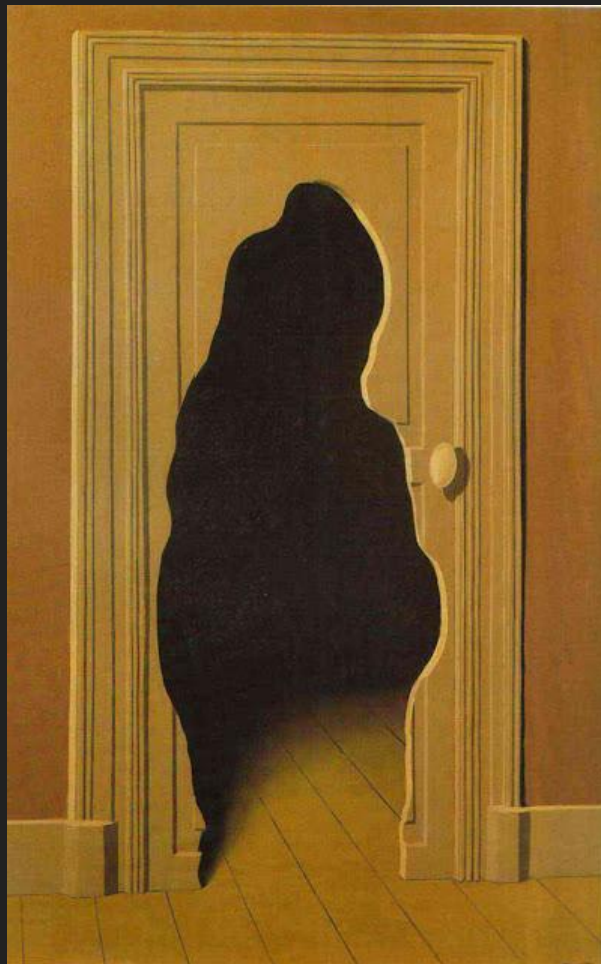


```
NOME1 = a  
NOME2 = b  
CONSTANTE1 = 2  
CONSTANTE2 = 3  
CLASSE1 = System  
MEMBRO1 = out  
METODO1 = println  
CONSTANTE_STR1 =  
"A é maior do que B"
```

É livre de
contexto!

Compiladores e CFLs

- Portanto, a resposta é:
 - Sim, podemos usar um PDA simples
- Mas é um PDA “turbinado”
 - Usando um truque para transformar uma linguagem não livre de contexto em uma linguagem livre de contexto
 - Desprezando nomes e valores
 - Mas armazenando a informação em outro lugar (tabela de símbolos)
- Quando o PDA simples terminar o trabalho dele
 - Fazemos verificações adicionais envolvendo os nomes



Rene Magritte. La réponse imprévue. 1933.

Compiladores e CFLs

- Sintaxe (forma) vs semântica (significado)
 - Compilador precisa lidar com ambos
 - Mas até onde vai a sintaxe?
 - Onde começa a semântica?
- `int a = "Alo mundo";`
 - Aqui tem um erro sintático ou semântico?
- Lembrando do ensino fundamental
 - Verbo transitivo direto PEDE objeto direto
- No mundo das LPs
 - Variável inteira PEDE constante inteira
 - Uma variável deve ter sido declarada antes de ser usada

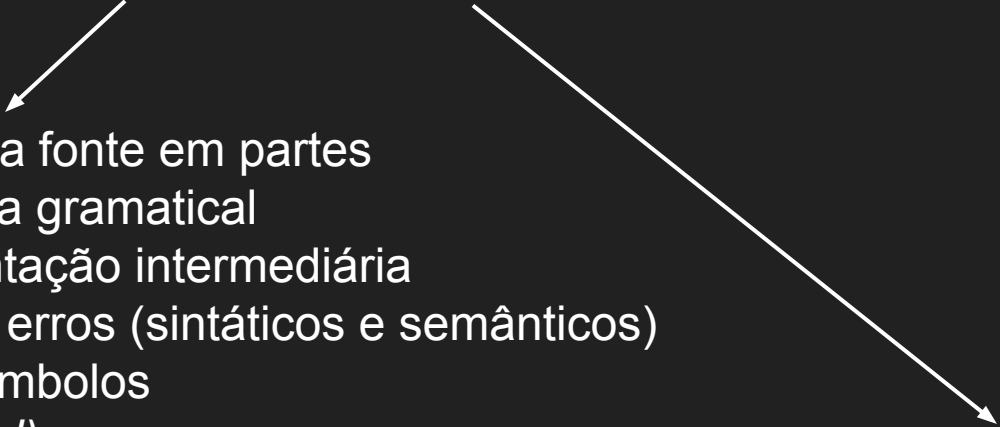
Compiladores e CFLs

- Em compiladores:
 - Tudo que está na gramática (livre de contexto) é sintático
 - O resto é considerado semântico
- Motivo: o uso de PDAs simples
 - Ou seja, adotamos o ponto de vista das linguagens livres de contexto, por praticidade
- Faz sentido, pois em LFA, temos:
 - Árvore de análise sintática
 - Somente com elementos da gramática

Estrutura de um compilador

Estrutura de um compilador

Duas etapas: análise e síntese



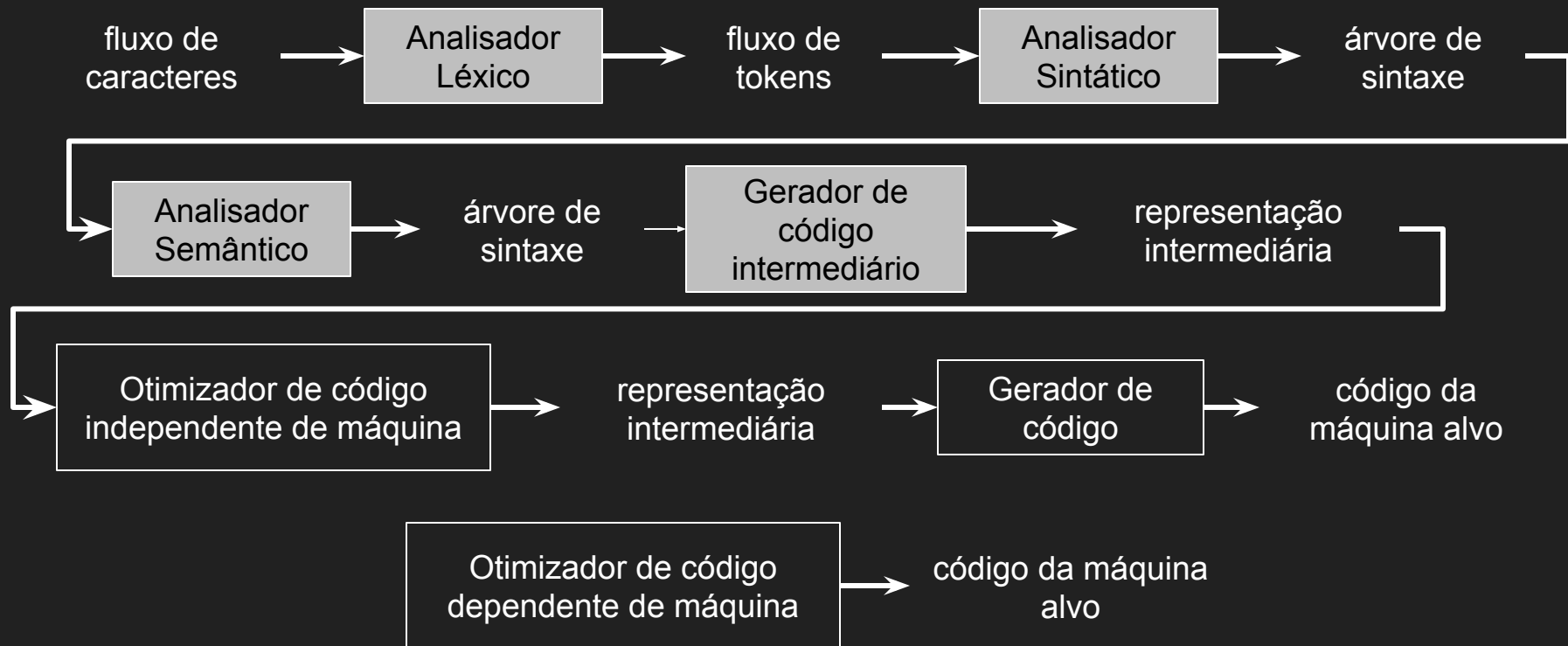
Quebrar o programa fonte em partes
Impor uma estrutura gramatical
Criar uma representação intermediária
Detectar e reportar erros (sintáticos e semânticos)
Criar a tabela de símbolos
(front-end)

Construir o programa objeto
com base na representação intermediária
e na tabela de símbolos
(back-end)

Fases de um compilador

front-end

back-end



Fases de um compilador

- **Análise léxica** (scanning)

- Lê o fluxo de caracteres e os agrupa em sequências significativas
 - Chamadas lexemas
- Para cada lexema, produz um token

<nome-token, valor-atributo>

-
- Identifica o tipo do token
 - Símbolo abstrato, usado durante a análise sintática
 - Aponta para a tabela de símbolos (quando o token tem valor)
 - Necessária para análise semântica e geração de código

Fases de um compilador

Exemplo: análise léxica

Espaços são normalmente descartados

```
position = initial + rate * 60
```

Lexema	Token
position	<id,1>
=	<=>
initial	<id,2>
+	<+>
rate	<id,3>
*	<*>
60	<num,4>

entrada	lexema	tipo	...
1	position	int	...
2	initial	int	...
3	rate	float	...
4	60	int	...

Tabela de símbolos

```
<id,1> <=> <id,2> <+> <id,3> <*> <num,4>
```

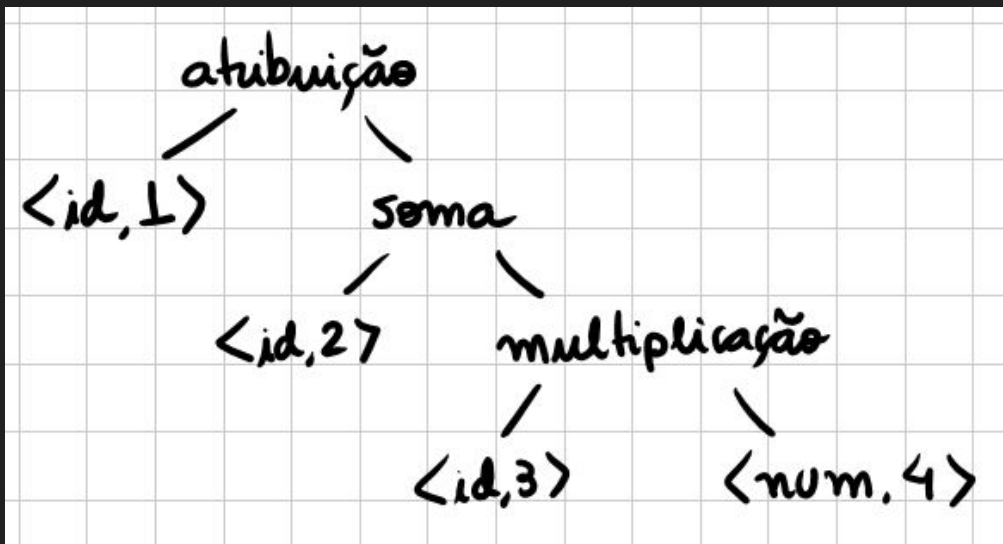
Fases de um compilador

- **Análise sintática (parsing)**
 - Usa os tokens produzidos pelo analisador léxico
 - Somente o primeiro “componente”
 - (ou seja, despreza os aspectos não-livres-de-contexto)
 - Produz uma árvore de análise sintática
 - Representa a estrutura gramatical do fluxo de tokens
 - As fases seguintes utilizam a estrutura gramatical para realizar outras análises e gerar o programa objeto

Fases de um compilador

Exemplo: análise sintática

`<id,1> <=> <id,2> <+> <id,3> <*> <num,4>`



Fases de um compilador

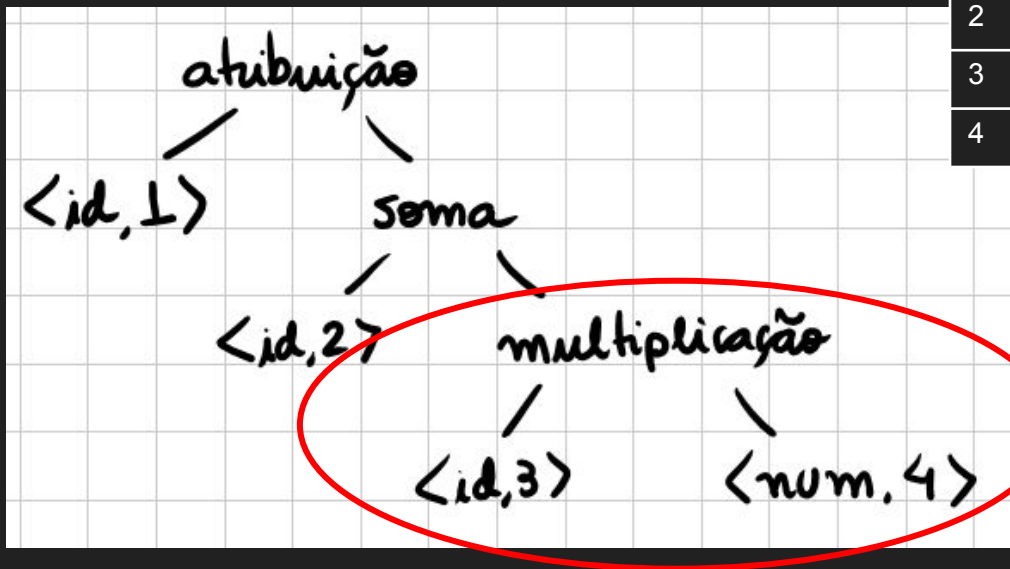
- **Análise semântica**

- Checa a consistência com a definição da linguagem
- Coleta informações sobre tipos e armazena na árvore de sintaxe ou na tabela de símbolos
- Checagem de tipos / coerção (adequação dos tipos) são tarefas típicas dessa fase

Fases de um compilador

Exemplo: análise semântica

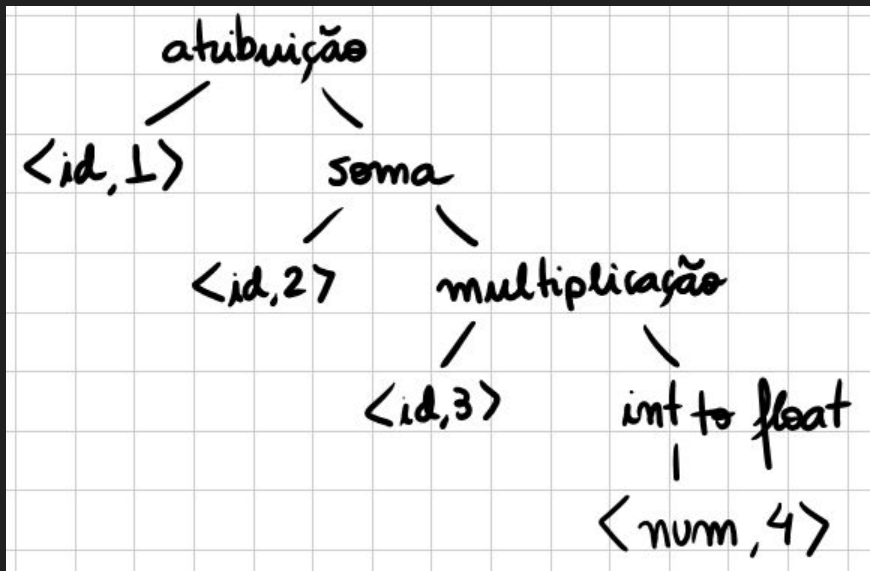
entrada	lexema	tipo	...
1	position	int	...
2	initial	int	...
3	rate	float	...
4	60	int	...



Tipos incompatíveis

Fases de um compilador

Exemplo: coerção



entrada	lexema	tipo	...
1	position	int	...
2	initial	int	...
3	rate	float	...
4	60	int	...

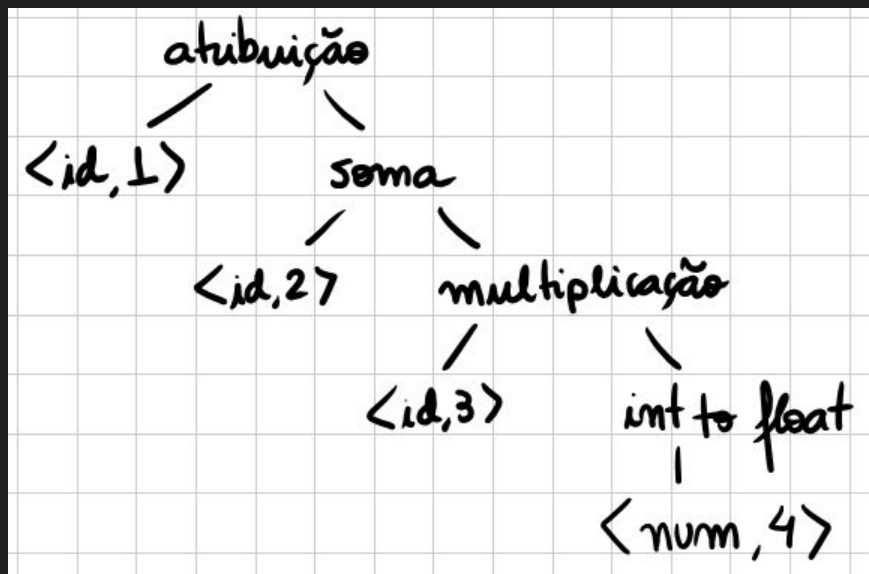
Fases de um compilador

- **Geração de código intermediário**

- Muitos compiladores geram uma representação intermediária, antes de gerar o código de máquina
- Motivos:
 - Fácil de produzir
 - Fácil de converter em linguagem de máquina
- Exemplos:
 - Árvore de sintaxe
 - Código de três endereços

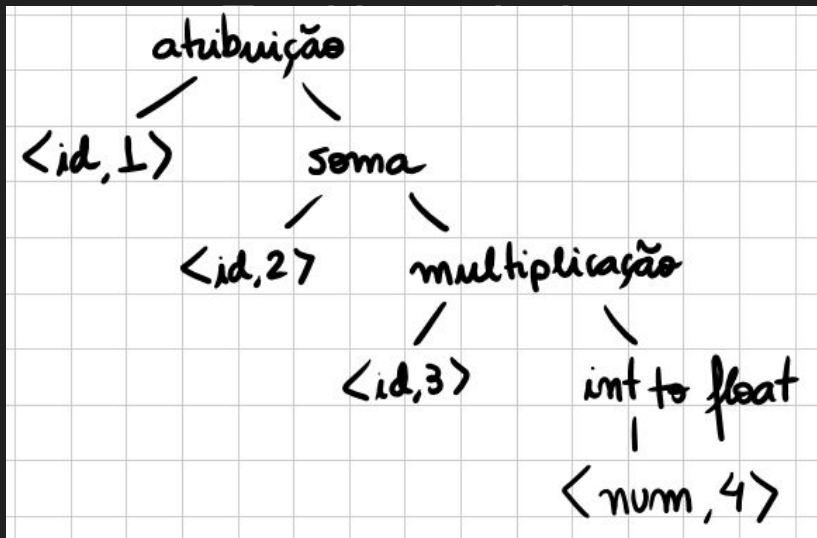
Fases de um compilador

- Exemplo: geração de código intermediário
 - Árvore de análise sintática já é um código intermediário



Fases de um compilador

- Exemplo: geração de código intermediário
 - Código de três endereços
 - Facilita geração de código objeto



```
num4 = 60
t1 = inttofloat(num4)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Fases de um compilador

- **Otimização de código**

- Tenta melhorar o código intermediário para produzir melhor código final
 - Mais rápido
 - Menor
 - Consome menos energia
 - Etc.

- Independentes x dependentes de máquina

- Quanto mais otimizações, mais lenta é a compilação

- Porém, existem algumas otimizações simples, que levam a grandes melhorias

Fases de um compilador

- Exemplo: otimização de código
 - Conversão “inttofloat” durante a compilação
 - Pode-se eliminar t3, pois é usado apenas uma vez

```
num4 = 60
```

```
t1 = inttofloat(num4)
```

```
t2 = id3 * t1
```

```
t3 = id2 + t2
```

```
id1 = t3
```

```
t1 = id3 * 60.0
```

```
id1 = id2 + t1
```

Fases de um compilador

- **Geração de código**

- Recebe como entrada uma representação intermediária do programa fonte
- Mapeia em uma linguagem objeto
- Seleciona os registradores ou localizações de memória para cada variável
- Tradução do código intermediário em sequências de instruções de máquina
 - Que realizam a mesma tarefa

Fases de um compilador

- Exemplo: geração de código
 - Código de máquina
 - Uso de registradores e instruções de máquina

```
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
```

Importante: é necessário lidar com endereços (não feito aqui)

Fases de um compilador

- **Gerenciamento da tabela de símbolos**
 - Fase “guarda-chuva”
 - Essencial: registrar nomes (variáveis, funções, classes, etc) usados no programa
 - Coletar informações sobre cada nome (tipo, armazenamento, escopo, etc)

Manipulação de erros

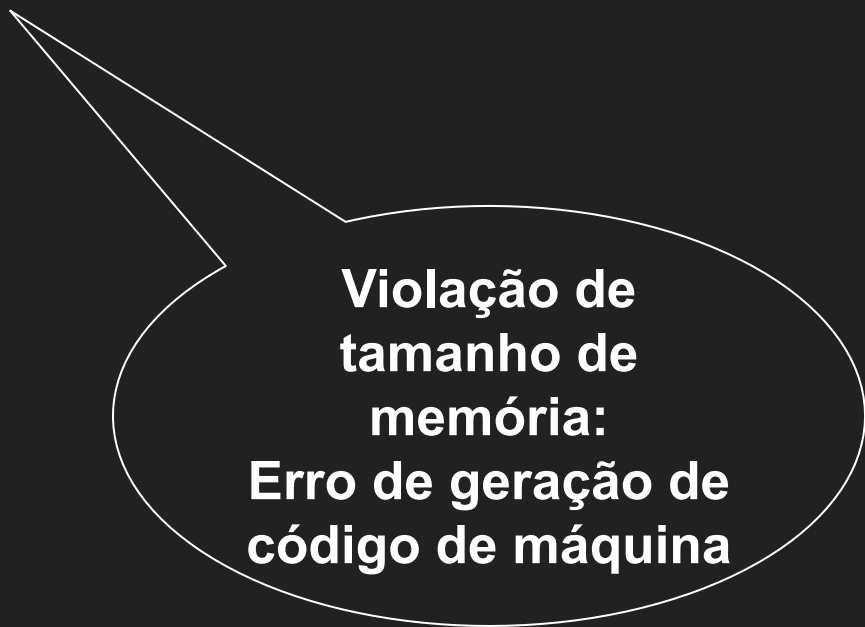
- Cada fase pode detectar diferentes erros
- Dependendo da gravidade, é possível que o compilador se “recupere” e continue lendo
 - Ou mesmo ignore o erro (ex: HTML)
- Em outros casos (na maioria), um erro desencadeia outros
- Mas, acredite, o compilador faz o melhor possível!

```
int main()
{
    int i, a[1000000000000000];
    float j@;

    i = "1";
    while (i<3
        printf("%d\n", i);
    k = i;
    return (0);
}
```

```
int main()
{
    int i, a[1000000000000000];
    float j@;

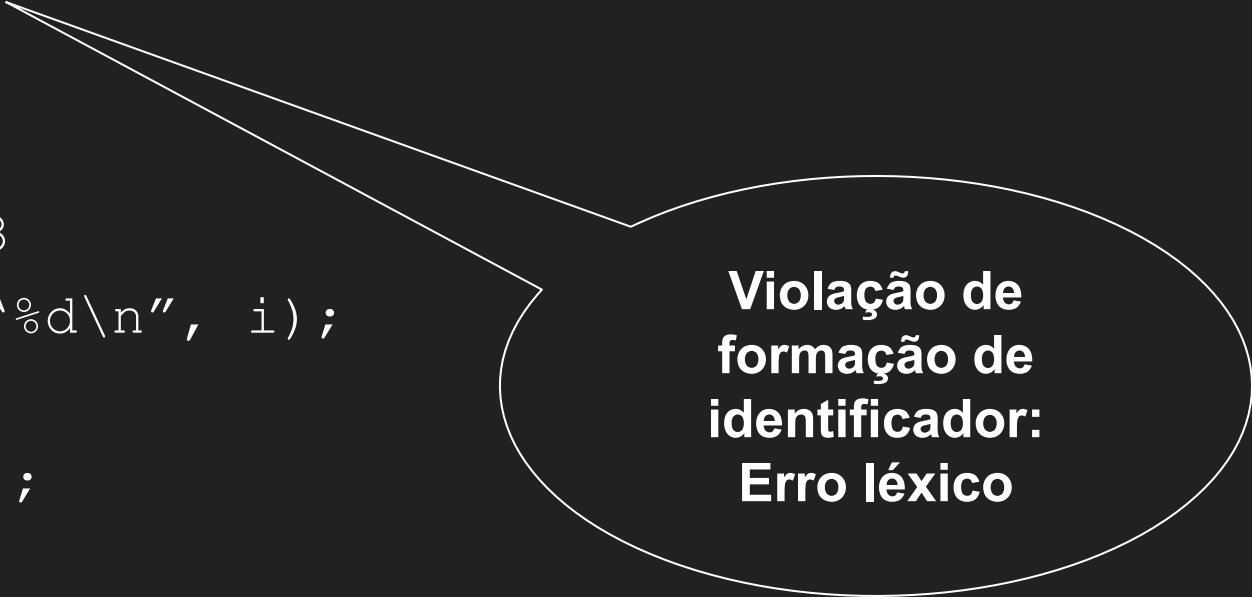
    i = "1";
    while (i<3
        printf("%d\n", i);
    k = i;
    return (0);
}
```



**Violação de
tamanho de
memória:
Erro de geração de
código de máquina**

```
int main()
{
    int i, a[1000000000000000];
    float j@;


    i = "1";
    while (i<3
        printf("%d\n", i);
    k = i;
    return (0);
}
```



**Violação de
formação de
identificador:
Erro léxico**


```
int main()
{
    int i, a[1000000000000000];
    float j@;

    i = "1";
    while (i<3
        printf("%d\n", i);
    k = i;
    return (0);
}
```



**Violação de
significado:
Erro semântico**

```
int main()
{
    int i, a[1000000000000000];
    float j@;

    i = "1";
    while (i<3
        printf("%d\n", i);
    k = i;
    return (0);
}
```

**Violação de
formação de
comando:
Erro sintático**

```
int main()
{
    int i, a[1000000000000000];
    float j@;

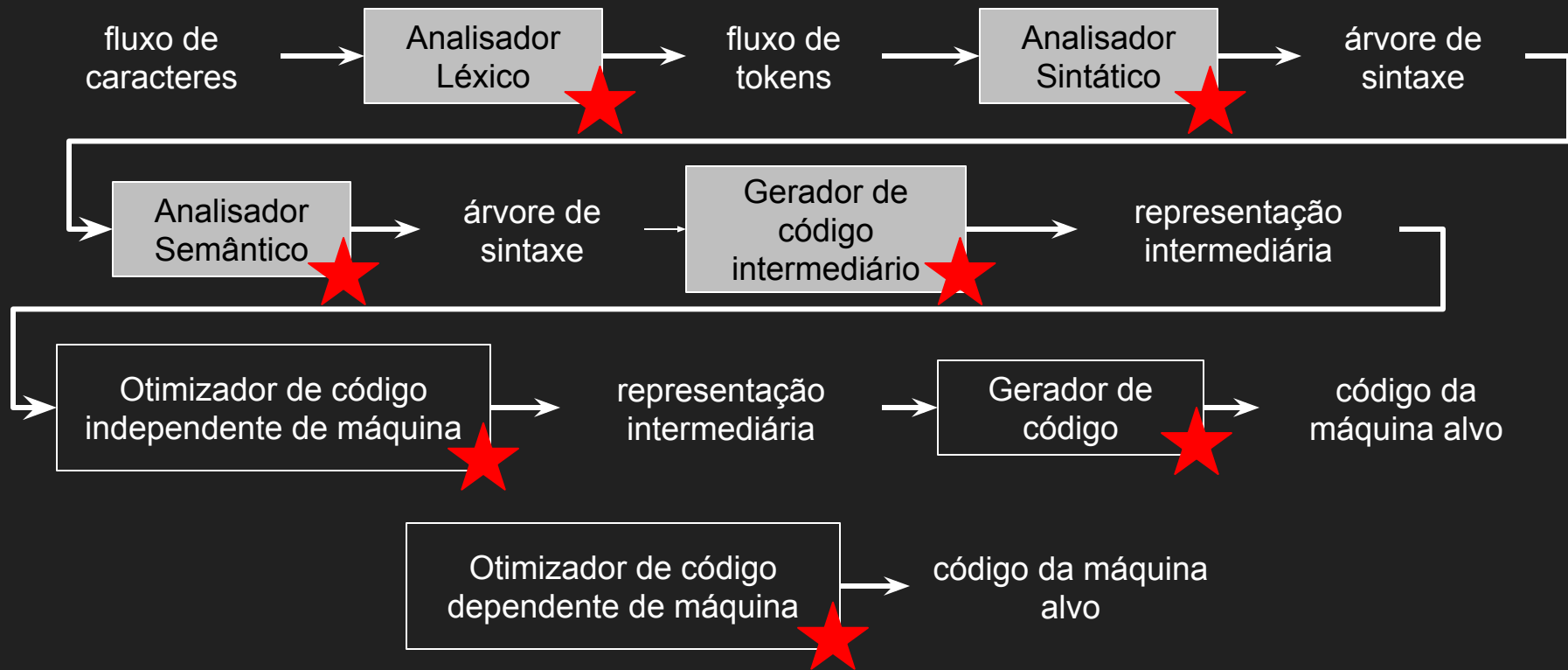
    i = "1";
    while (i<3
        printf("%d\n", i);
    k = i;
    return (0);
}
```

**Violação de
identificadores
conhecidos:
Erro contextual
("semântico")**

Manipulação de erros

front-end

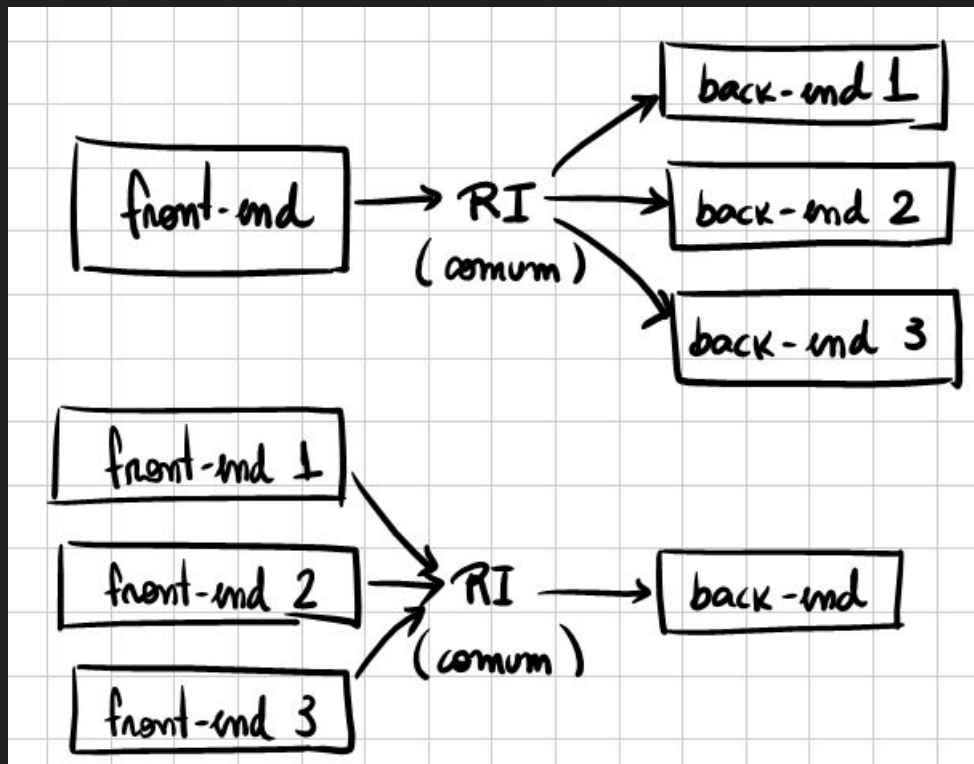
back-end



Agrupamento das fases

- A divisão anterior é apenas lógica
- Pode-se realizar várias fases de uma única vez
- Em uma única passada
 - Imagine que o programa está uma fita VHS
 - Cada passada é um “play” na fita toda
 - Ao fim de cada passada, precisa rebobinar
- Exemplo:
 - Passada 1 = análise léxica, sintática, semântica e geração de representação intermediária (front-end)
 - Passada 2 = otimização (opcional)
 - Passada 3 = geração de código específico de máquina (back-end)

Agrupamento de fases



Resumo

Nesta disciplina o foco estará no front-end

Análise léxica + sintática + semântica

Um pouco de geração de código / interpretação



Suficiente para muitas aplicações
práticas

Fim