# Construção de compiladores

Profs. Mário César San Felice (e Helena Caseli,
Murilo Naldi, Daniel Lucrédio)
Departamento de Computação - UFSCar

1º semestre / 2018

Tópico 1 - Introdução

# Introdução

# O que são compiladores?

- Resposta “livro-texto”
  - “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

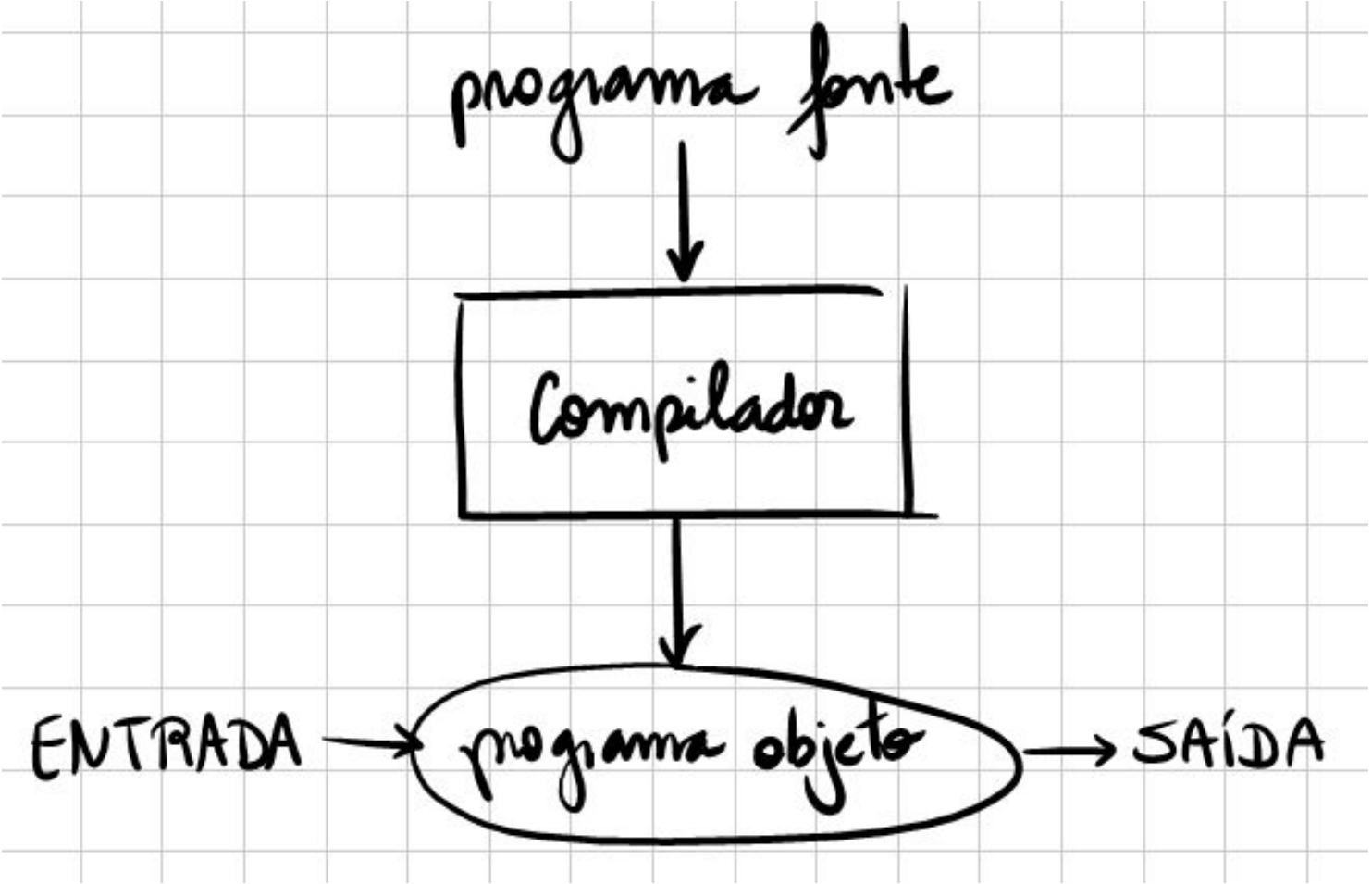
# O que são compiladores?

- Imagine-se tendo que treinar um cachorro
- O ideal:
  - “Quando eu disser JORNAL, atravesse a sala, abra a porta com a pata, pegue o jornal, sem estragá-lo, e me traga”
- Na prática:
  - É necessário treiná-lo com recompensas, palavras curtas, entonação de voz, e paciência
  - Você sabe fazer isso?
    - Pode contratar um treinador com experiência na “linguagem” dos cachorros

# 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

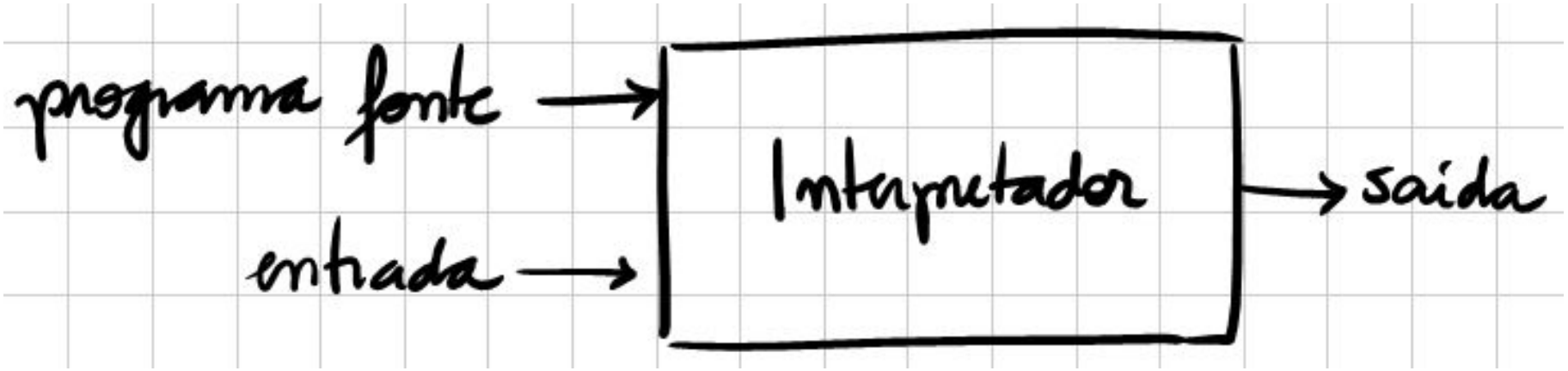
# Compiladores



# 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

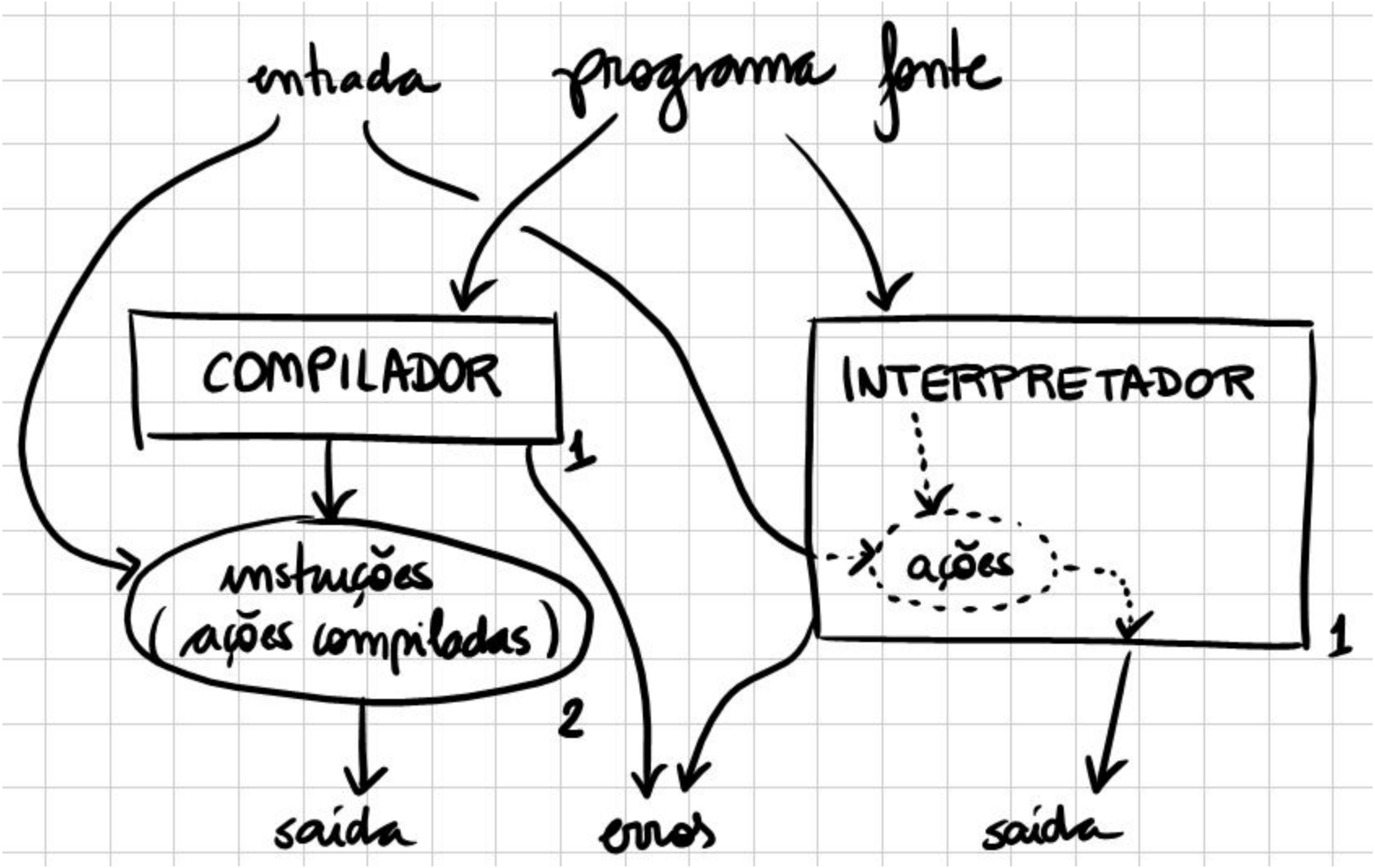
# Interpretadores



# Interpretadores e compiladores

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

# Interpretadores vs compiladores



# Interpretadores vs compiladores

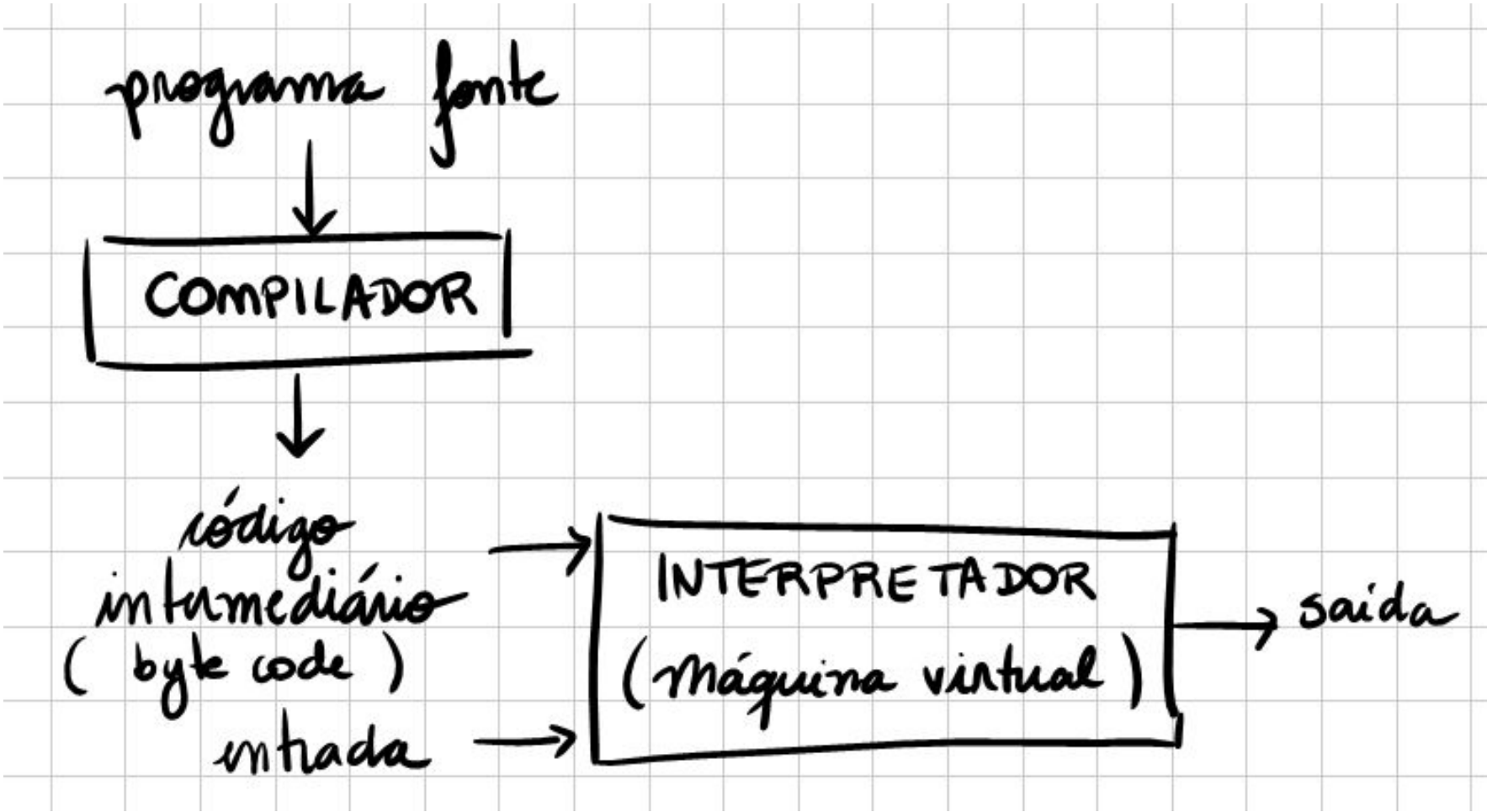
- Interpretadores e compiladores podem ser utilizados com um mesmo propósito
- Cada um tem vantagens/desvantagens
- Compilador:
  - Mapeamento entrada/saída mais rápido
    - Traduz 1 vez para executar n vezes
  - Na tradução, perde-se informação
    - Dificulta o diagnóstico de erros
- Interpretador
  - Mapeamento entrada/saída mais lento
    - Precisa “traduzir” toda vez que é executado
  - O diagnóstico de erros é normalmente melhor
  - Mais flexível

# Interpretadores vs compiladores

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

# Interpretadores vs compiladores

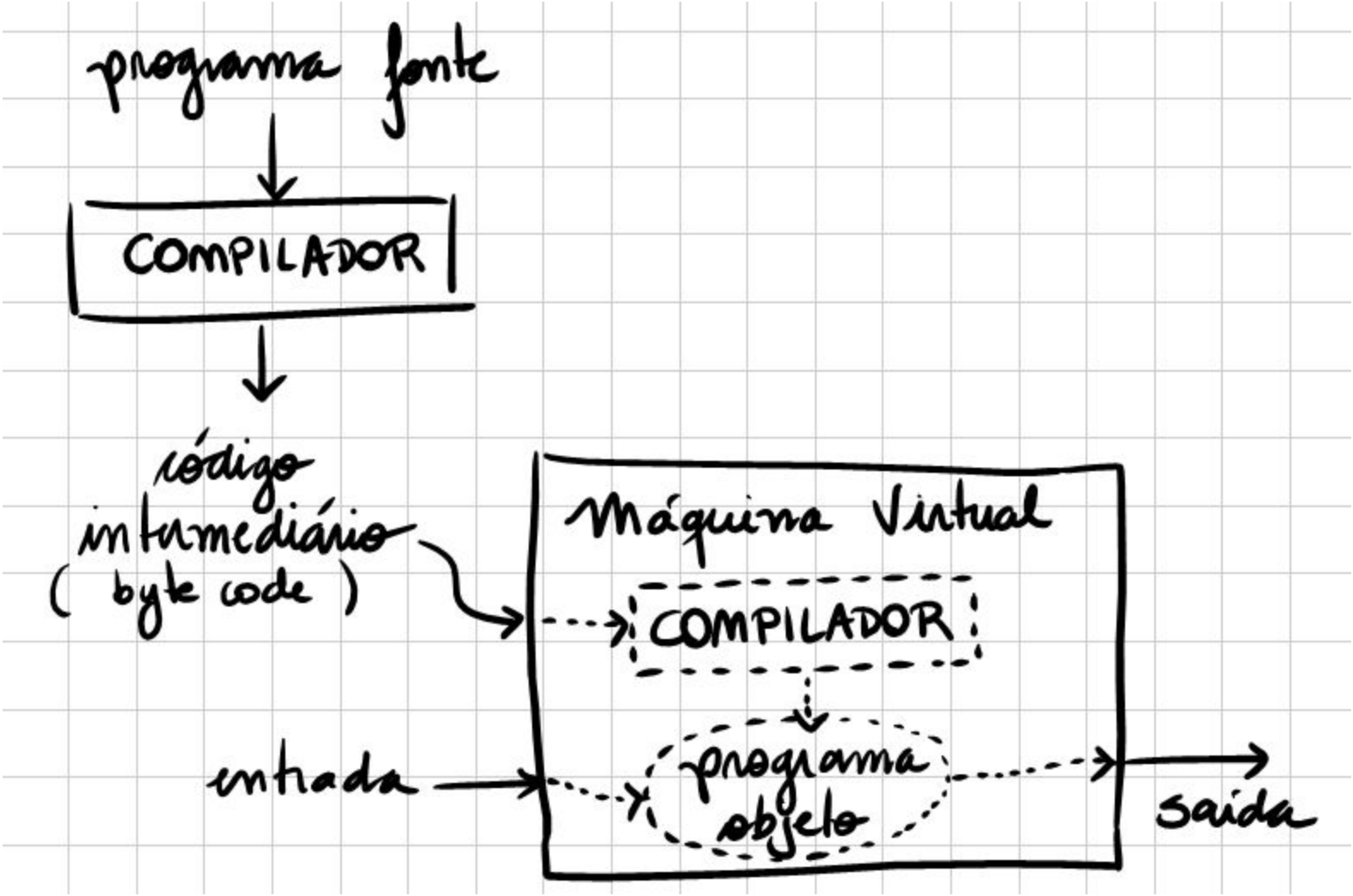
## Abordagem híbrida



# Interpretadores vs compiladores

- Quando o mapeamento entrada/saída é:
  - Demorado
  - Interativo
  - Complexo
- Pode-se realizar a compilação “na hora”
  - Just-In Time (JIT)

# JIT

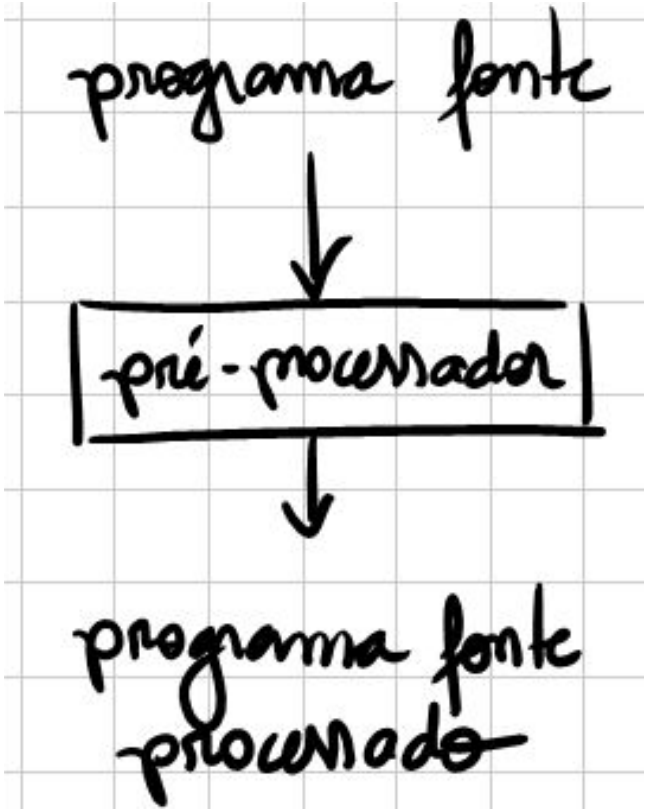

programa fonte
COMPILADOR
código intermediário ( byte code )
Máquina Virtual
COMPILADOR
entrada
programa objeto
saída

# Compiladores vs interpretadores

- Ao longo da disciplina, estudaremos “compiladores”
  - Afinal, o nome da disciplina é “construção de compiladores 1”
  - Mas grande parte das técnicas empregadas são as mesmas para a construção de interpretadores
    - Então, de certa forma, você também está cursando “construção de interpretadores 1”
    - Mas infelizmente não vale crédito!

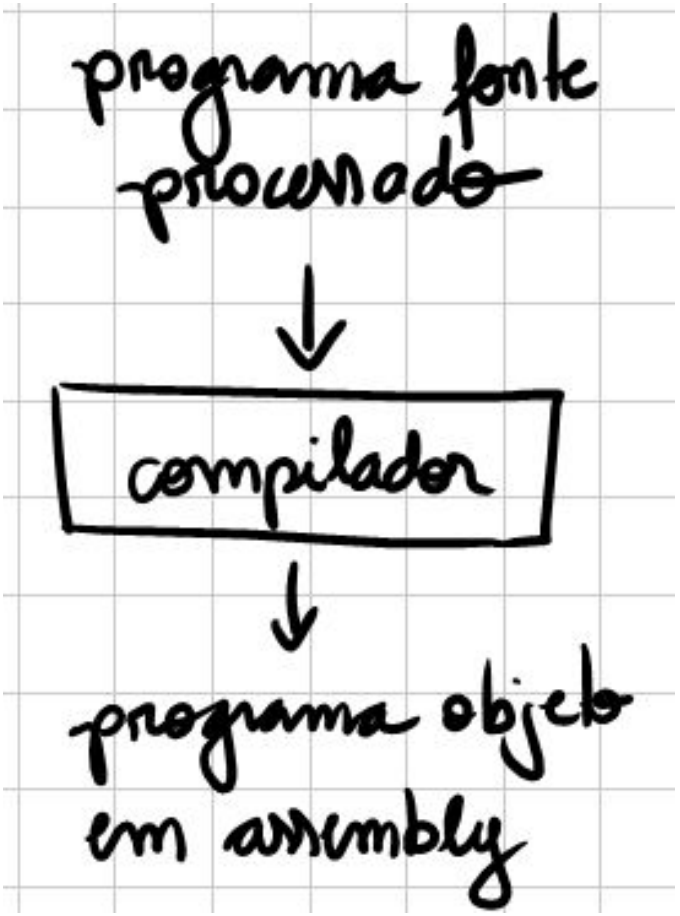
# Outros programas relacionados

- **Pré-processador**:
  - Coleta diversas partes de um programa
  - Expande macros



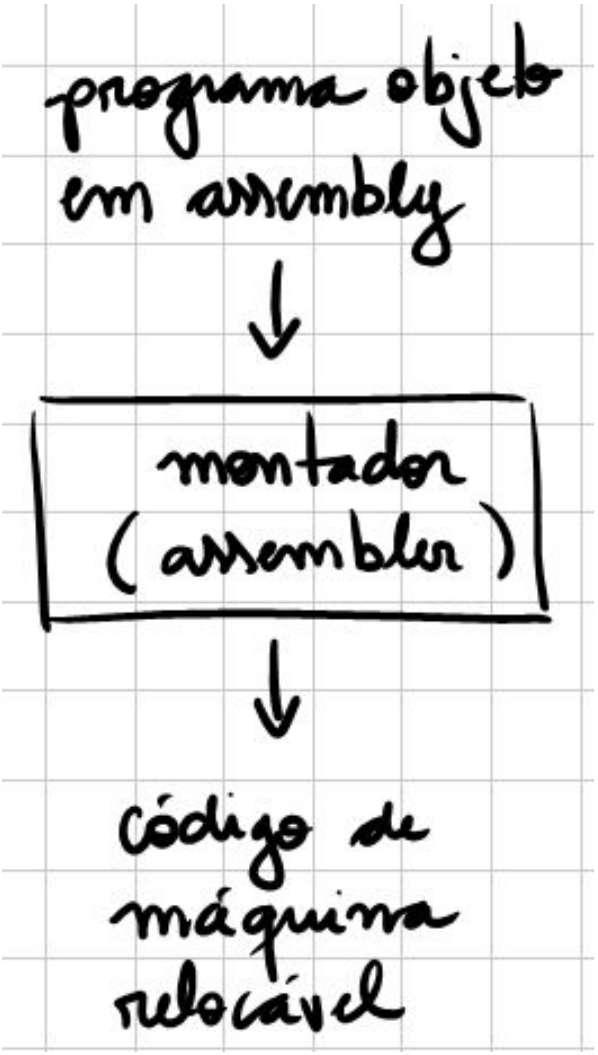
# Outros programas relacionados

- **Compilador**:
  - Produz um programa em uma linguagem simbólica (assembly)
    - Mais fácil de ser gerada
    - Mais fácil de ser depurada
    - Suficientemente próxima da linguagem de máquina



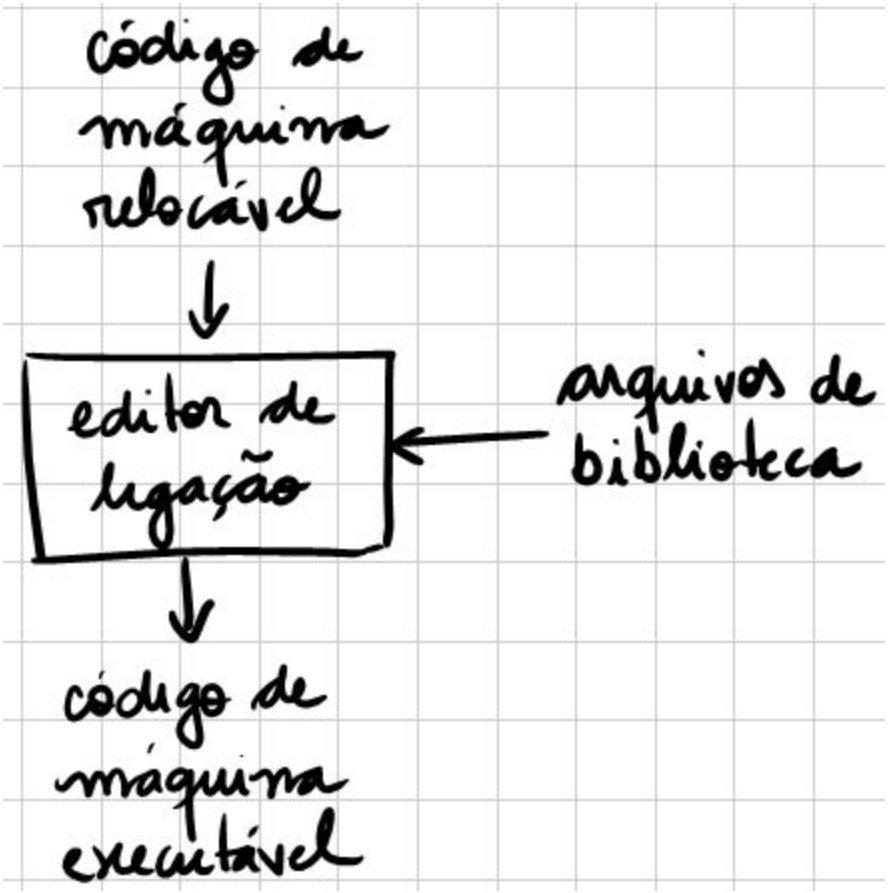
# Outros programas relacionados

- **Montador** (assembler):
  - Produz código de máquina relocável
    - Ou seja, que pode ser movido na memória
  - Pedaços de um programa, mas com endereços “flexíveis”



# Outros programas relacionados

- **Editor de ligação** (linker):
  - “Junta” os pedaços de programa relocáveis em um único “executável”
  - Faz a ligação entre diversos pedaços (ou bibliotecas)
  - Resolve endereços de memória externos
    - Referências entre diferentes arquivos



# Compiladores e LFA

# Compiladores

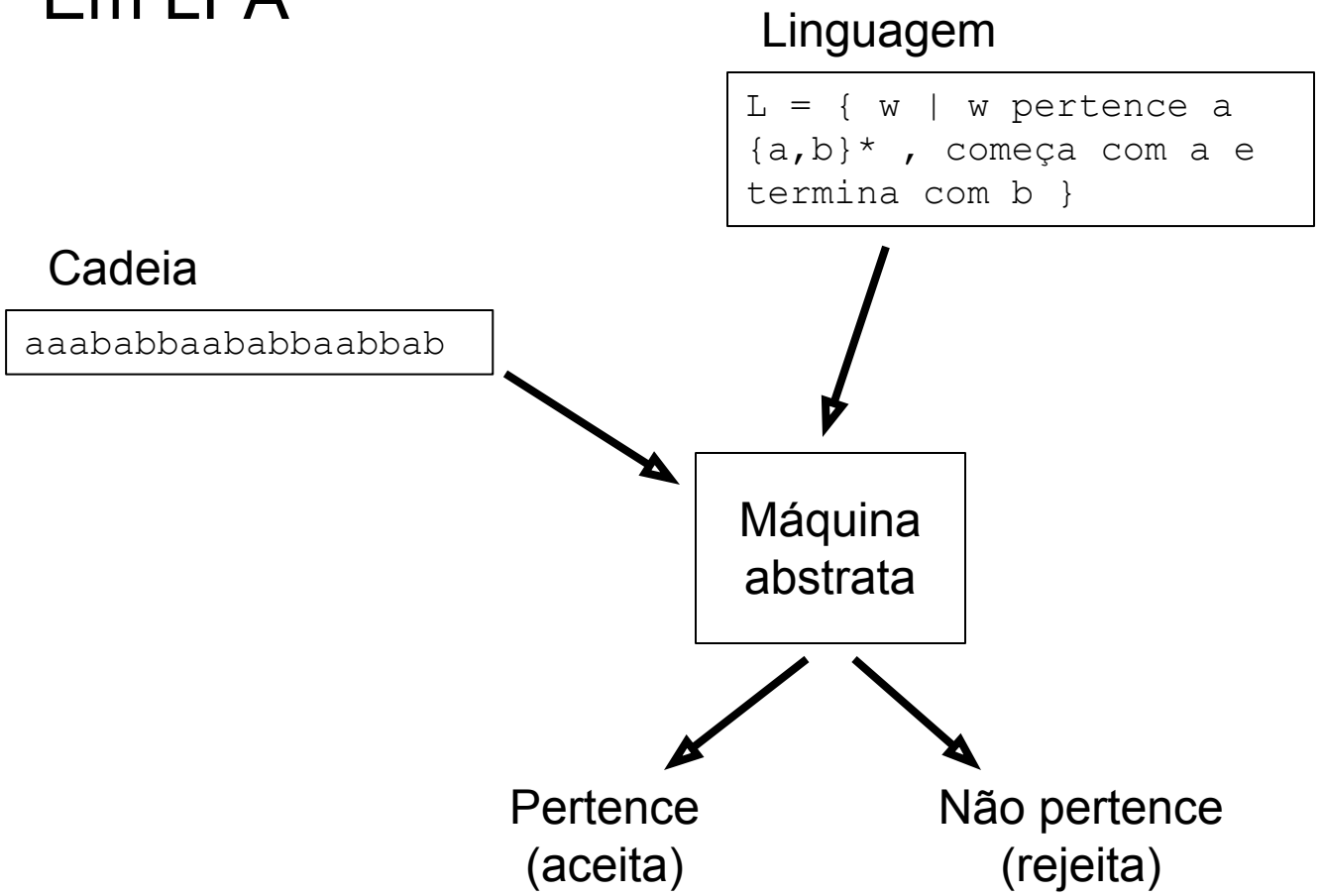
- Normalmente pensamos em linguagens de programação
  - Programa fonte = algoritmo
  - Programa objeto = executável
- Mas compiladores podem ser utilizados em outros contextos
  - SQL
    - Programas SQL não são algoritmos
    - Mas ainda assim existe um compilador (ou interpretador)
  - HTML
    - O 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

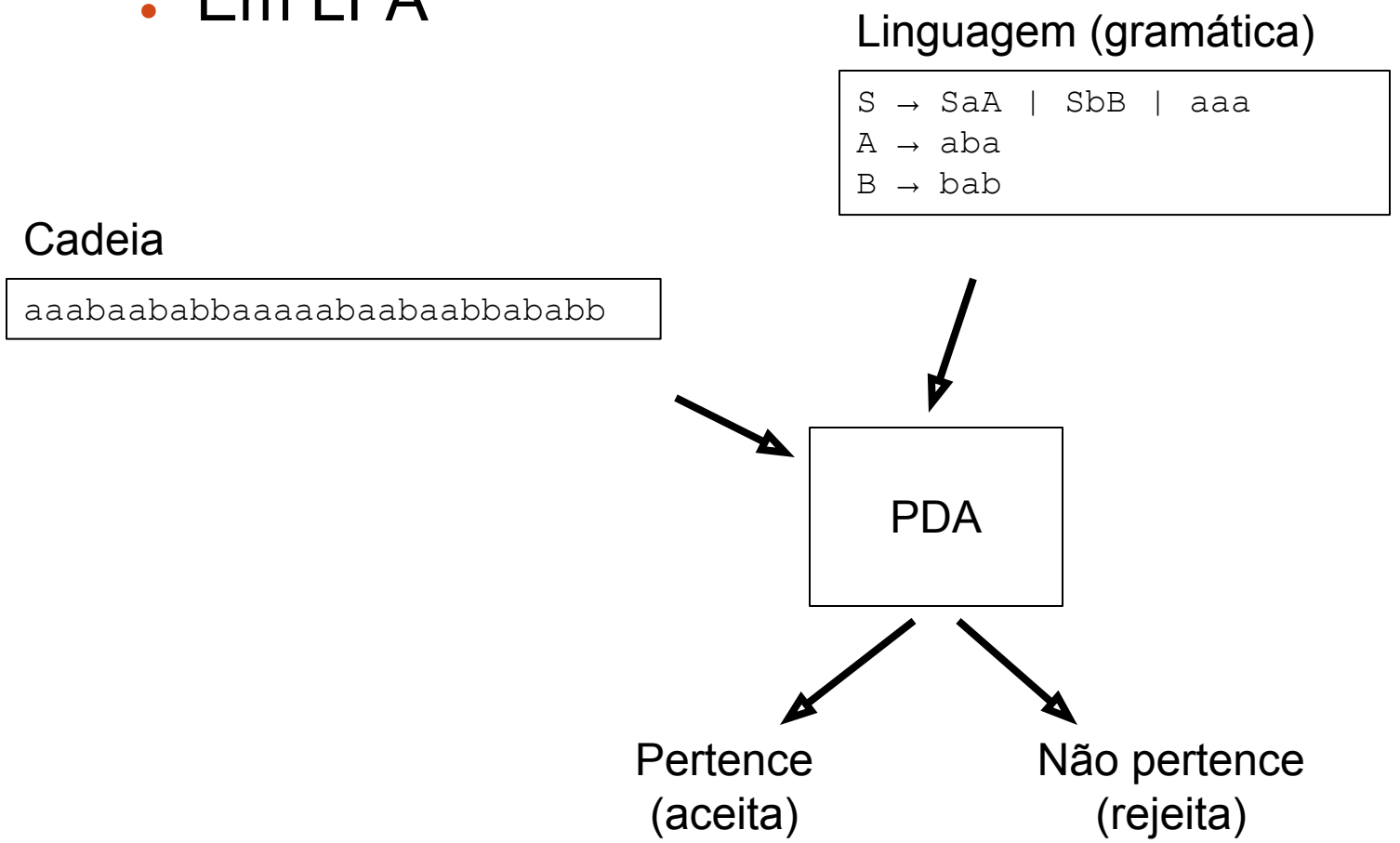


# LFA x compiladores

- 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

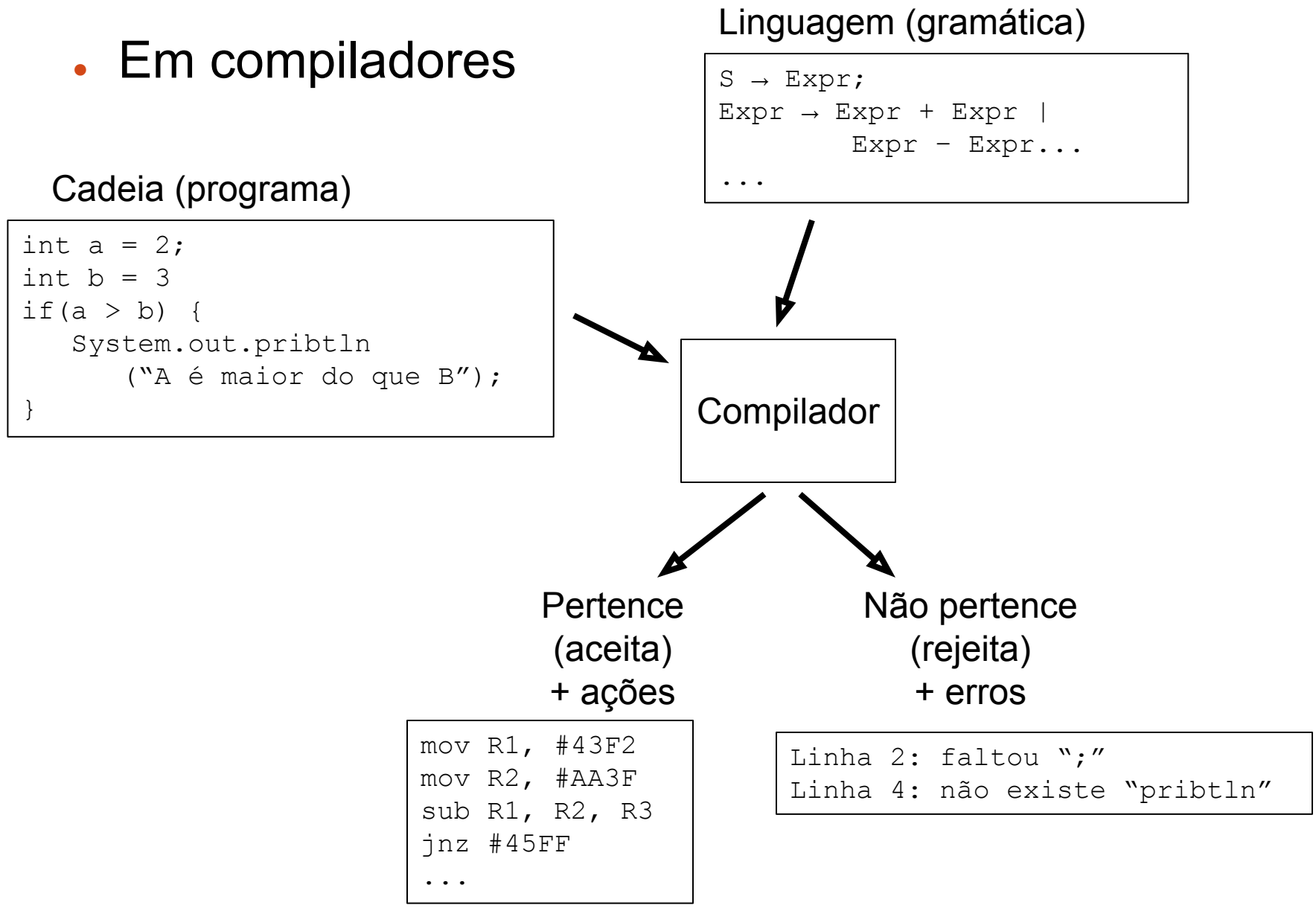
# LFA x compiladores

- Em LFA



# LFA x compiladores

- Em compiladores



# 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:
    - Expr → Expr + Expr { ação de soma } | Expr – Expr { 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

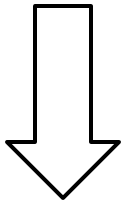
- Em algumas LP, variáveis precisam ser declaradas antes de serem utilizadas
  - É o mesmo caso da linguagem {ww | w em um alfabeto com mais de um símbolo}
  - Não é uma linguagem livre de contexto!!
    - Prova: lema do bombeamento

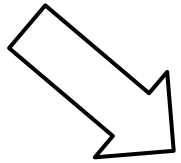
# Linguagens não livres de contexto

- 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

# E se...

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





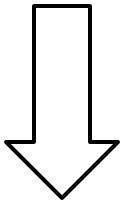
```
TIPO NOME = CONSTANTE;
TIPO NOME = CONSTANTE;
if(NOME > NOME) {
   CLASSE.MEMBRO.METODO(CONSTANTE_STR);
}
```

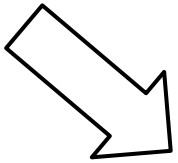
```
NOME1 = a
NOME2 = b
CONSTANTE1 = 2
CONSTANTE2 = 3
CLASSE1 = System
MEMBRO1 = out
METODO1 = println
CONSTANTE_STR1 =
  “A é maior do que B”
```

# 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!





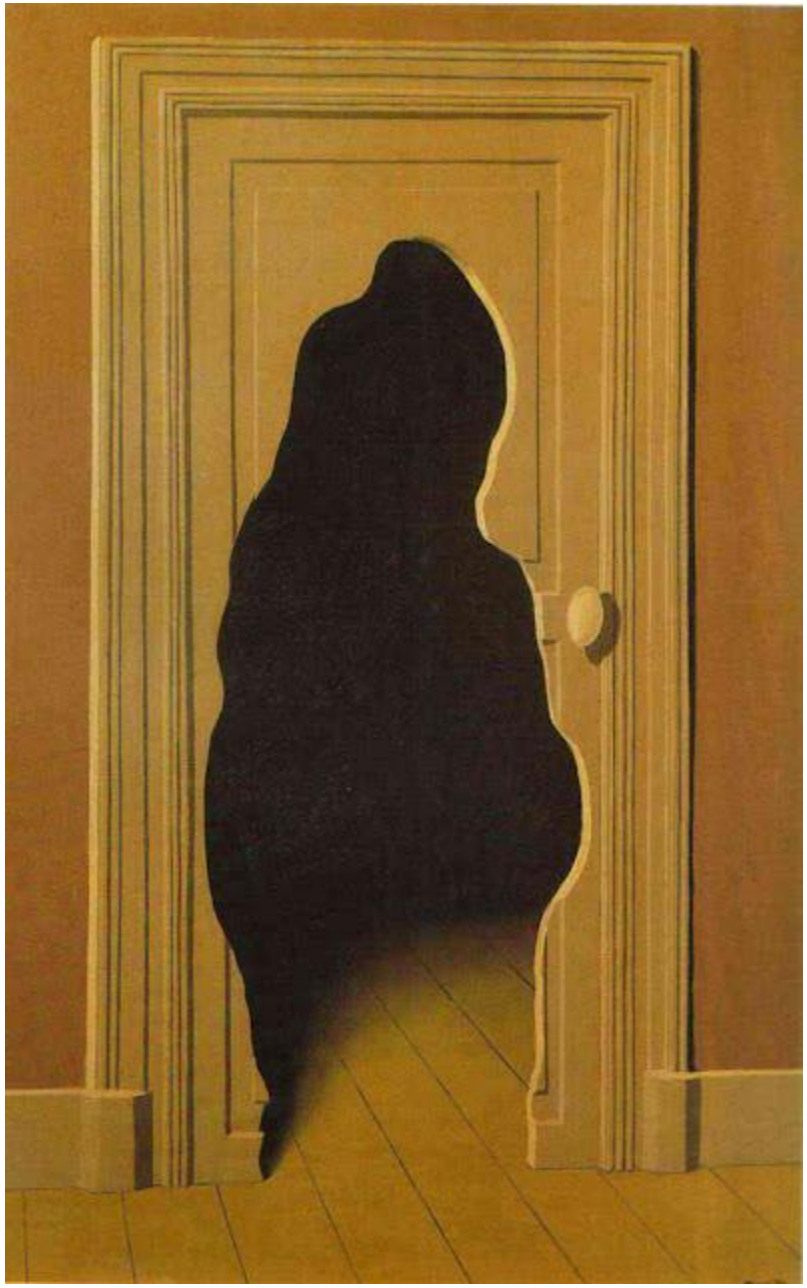
```
TIPO NOME = CONSTANTE;
TIPO NOME = CONSTANTE;
if(NOME > NOME) {
   CLASSE.MEMBRO.METODO(CONSTANTE_STR);
}
```

É livre de contexto!

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

# 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

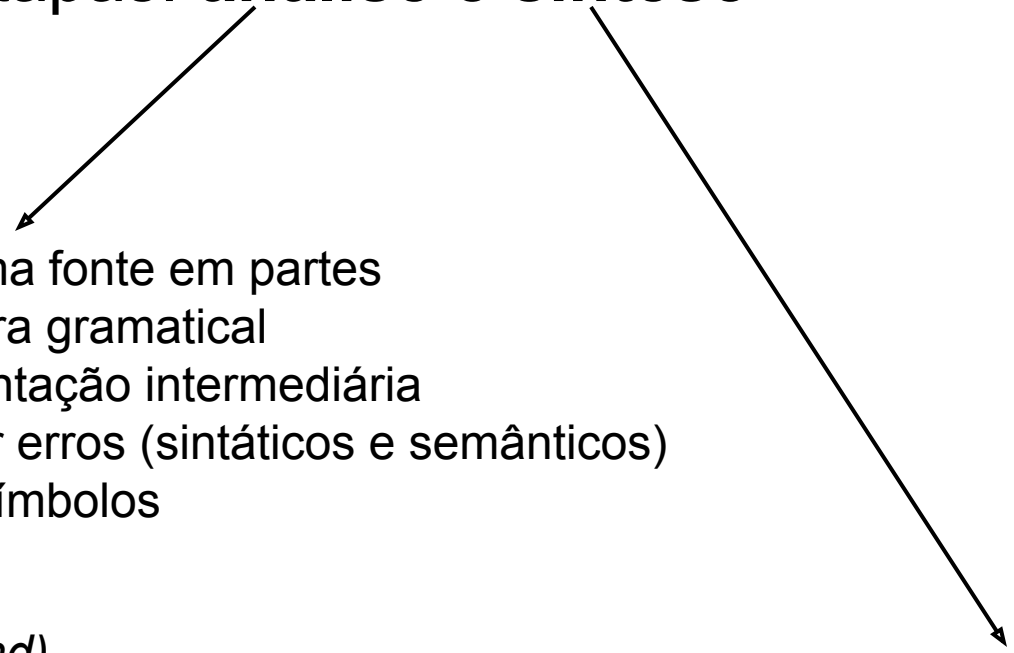
# Compiladores e CFLs

- 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**



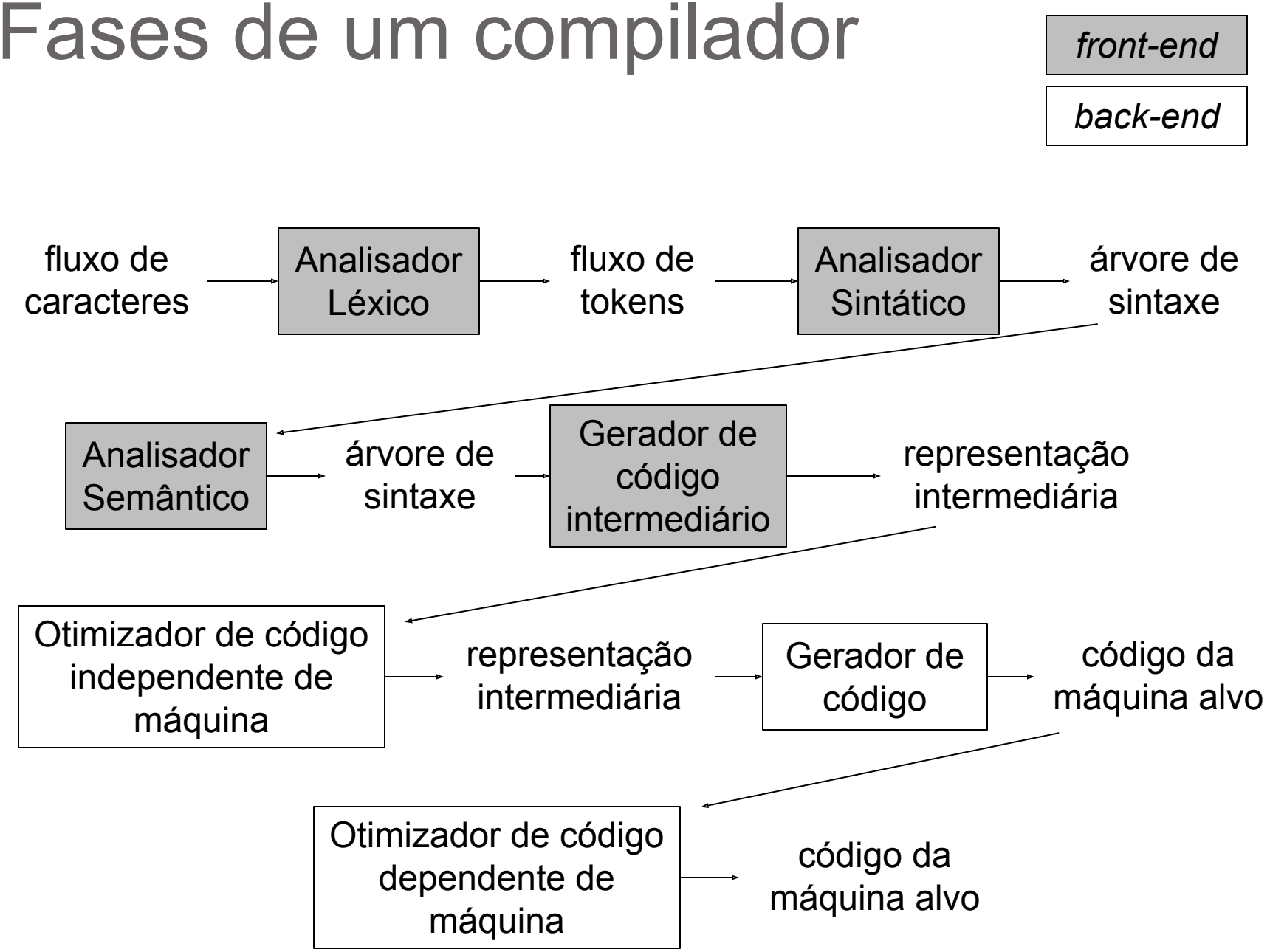
Quebrar o programa fonte em partes
Impor uma estrutura gramatical
Criar uma representação intermediária
Detectar e reportar erros (sintáticos e semânticos)
Criar a tabela de símbolos

*(front-end)*

Construir o programa objeto
com base na representação intermediária
e na tabela de símbolos

*(back-end)*

# Fases de um compilador



# 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

- **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

- **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

- **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

- **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

- **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

- **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)

# Fases de um compilador

- Exemplo: análise léxica

Espaços são 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 | ... |

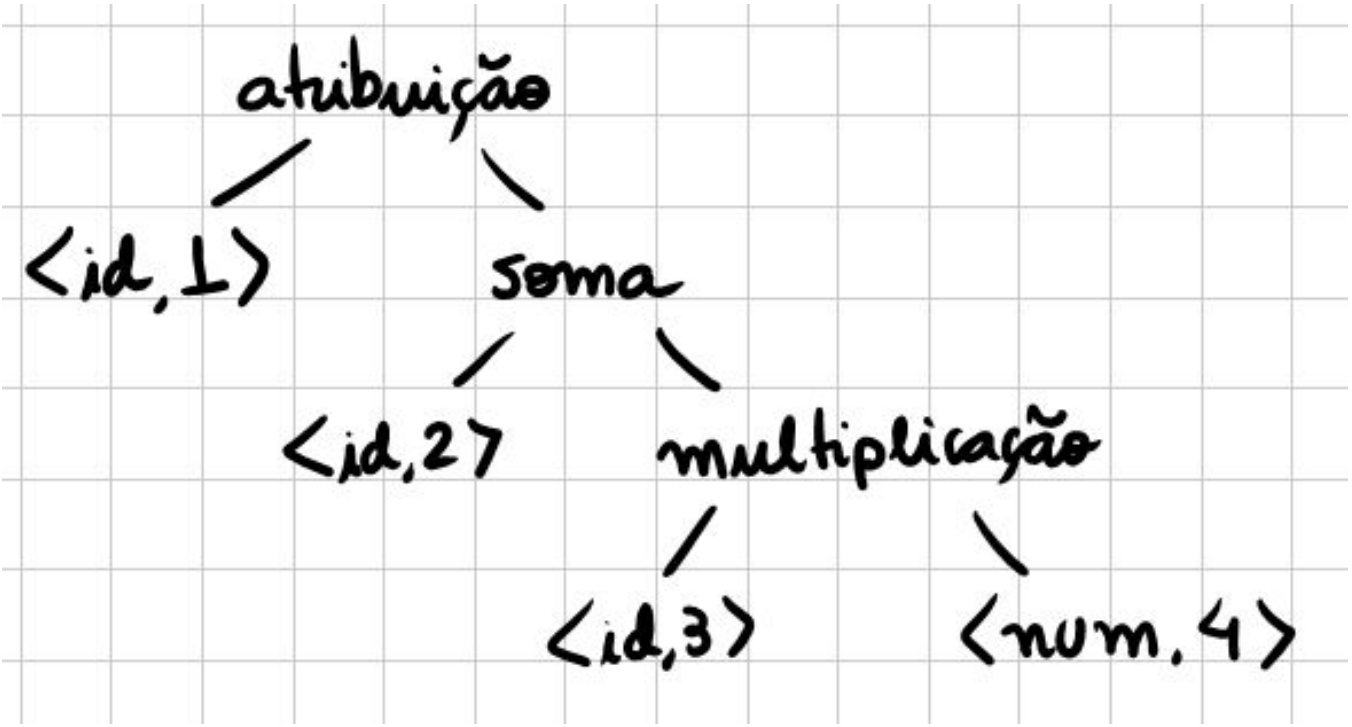
**Tabela de símbolos**

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

# Fases de um compilador

- Exemplo: análise sintática

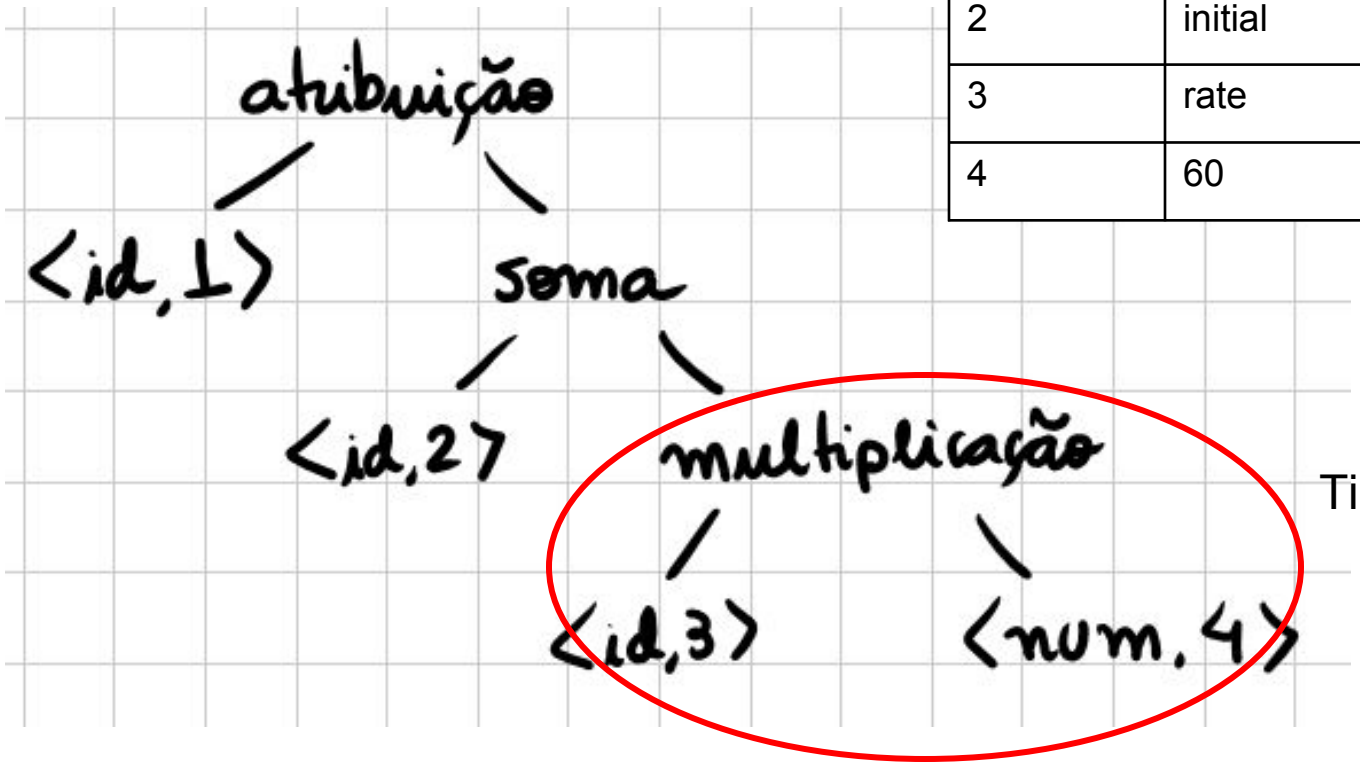
```
<id,1> <=> <id,2> <+> <id,3> <*> <num,4>
```



# Fases de um compilador

- Exemplo: análise semântica

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

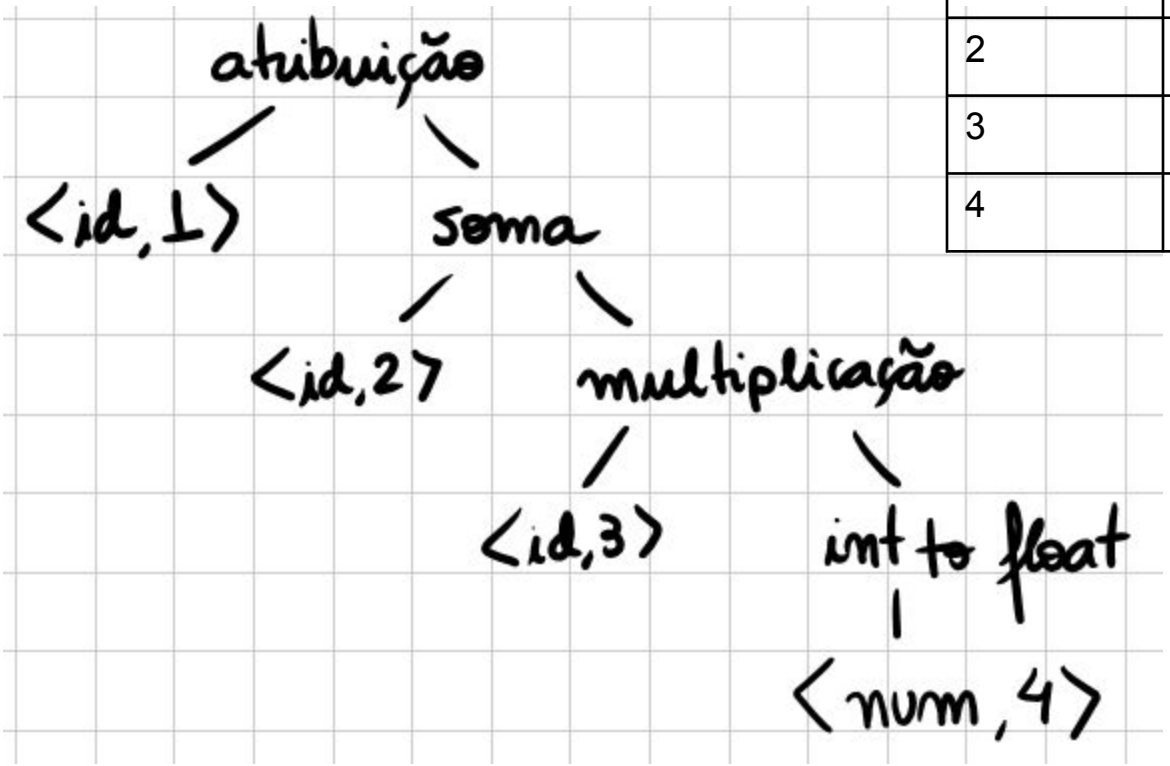



Tipos incompatíveis

# Fases de um compilador

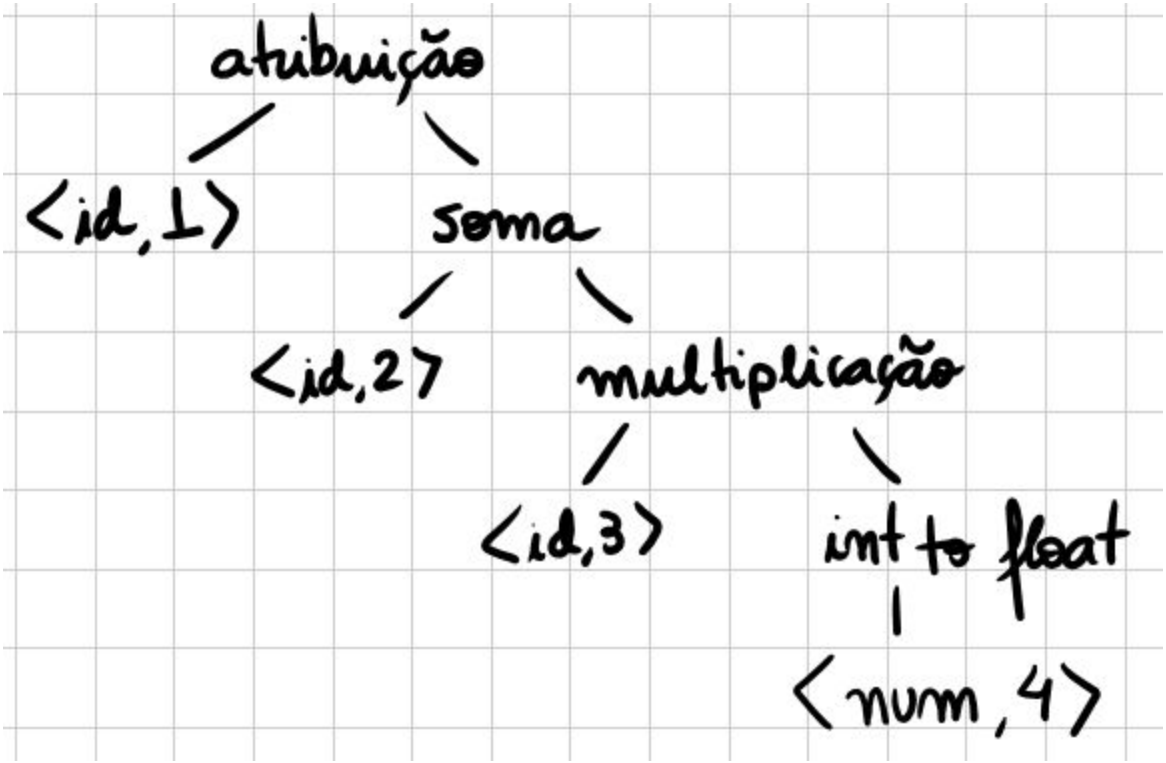
- Exemplo: coerção

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



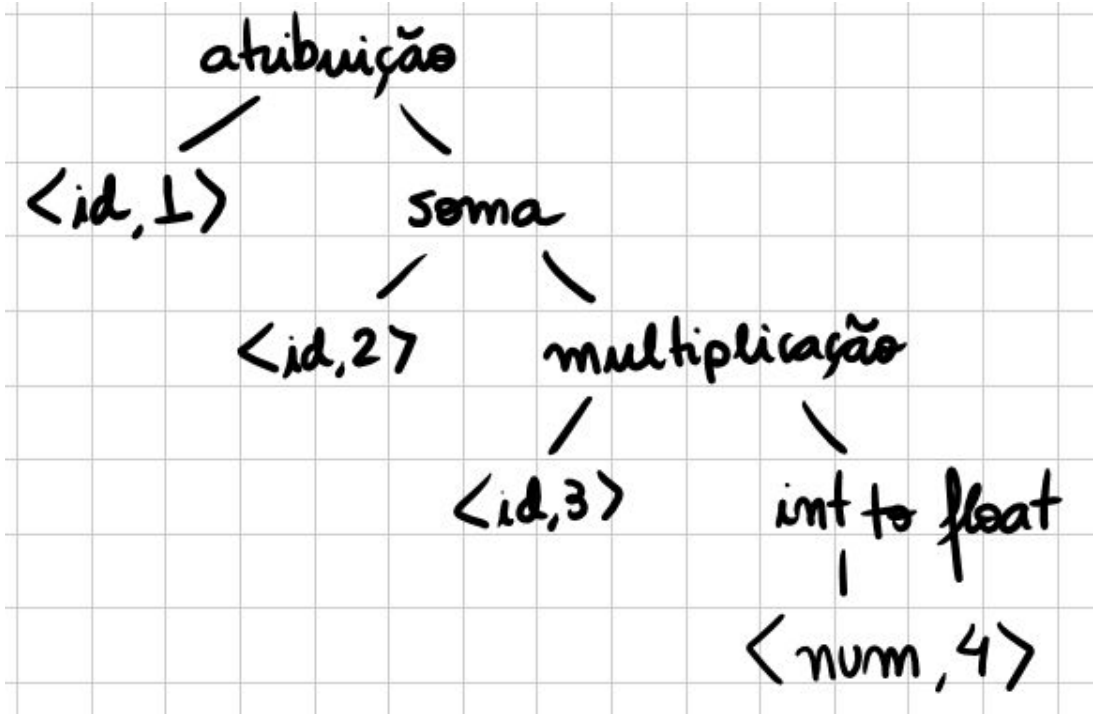
# Fases de um compilador

- Exemplo: geração de código intermediário
  - Árvore de sintaxe já é um código intermediário



# Fases de um compilador

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




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

# Fases de um compilador

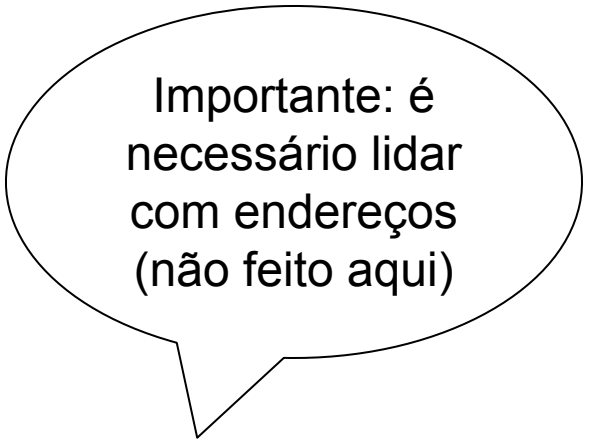
- Exemplo: otimização de código
  - Conversão “inttofloat” durante a compilação
  - Pode-se eliminar t3, pois é usado apenas uma vez

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

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

# Fases de um compilador

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



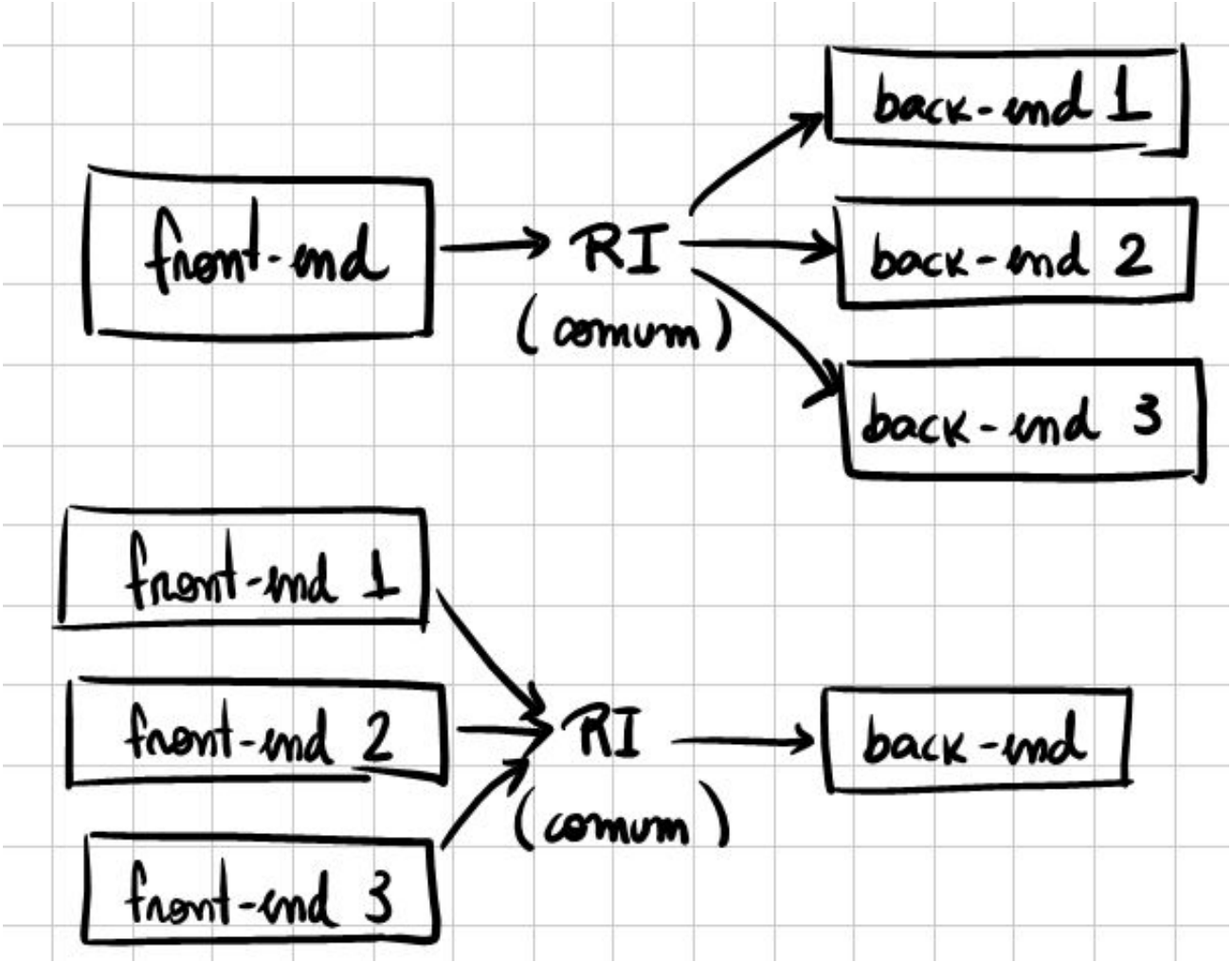

```
t1 = id3 * 60.0
id1 = id2 + t1
```

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

# 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



# 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!

# Manipulação de erros

```
int main()
{
  int i, a[1000000000000];
  float j@;

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

# Manipulação de erros

```
int main()
{
   int i, a[10000000000000];
   float j@;

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

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

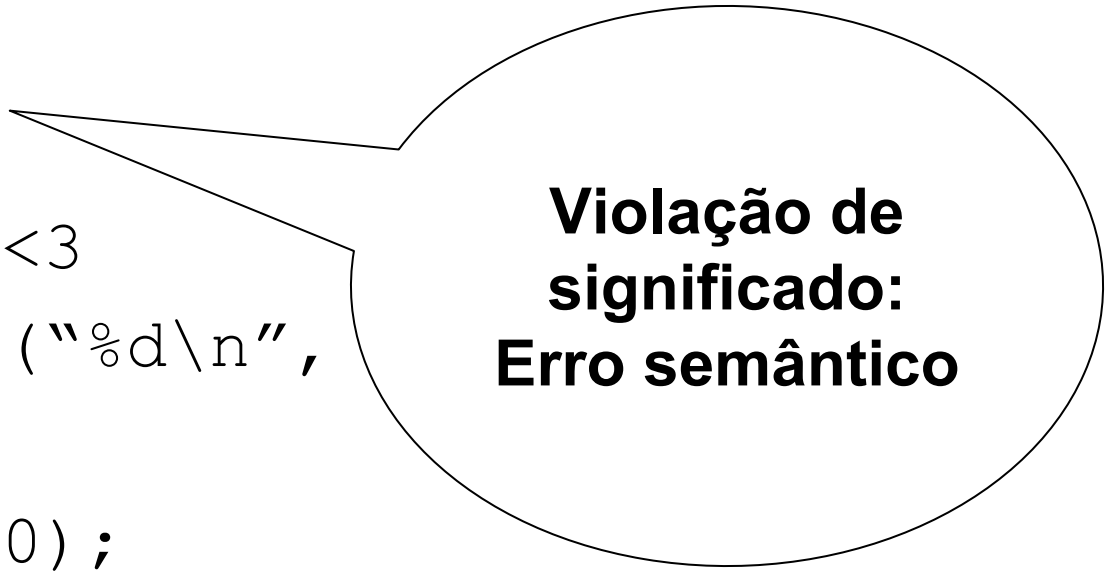
# Manipulação de erros

```
int main()
{
   int i, a[1000000000000];
   float j@;

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

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

# Manipulação de erros

```
int main()
{
   int i, a[1000000000000];
   float j@;

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



# Manipulação de erros

```
int main()
{
   int i, a[1000000000000];
   float j@;

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

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

# Manipulação de erros

```
int main()
{
   int i, a[1000000000000];
   float j@;

   i = "1";
   while (i<3
      printf("
   k = i;
   return (0);
}
```

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

# A ciência da criação de um compilador

# A ciência dos compiladores

- Compiladores são um exemplo de como resolver problemas complicados abstraindo sua essência matematicamente
  - A ciência da computação está por trás
  - Teoria da computação
    - Máquinas de estados finitos
    - Expressões regulares
    - Gramáticas livres de contexto
    - Árvores

# A ciência dos compiladores

- Em compiladores a teoria vira prática, pois questões práticas importantes devem ser consideradas:
  - Corretude
  - Desempenho dos programas
  - Tempo de compilação
  - Facilidade de construção
- Literalmente, o que antes levava semanas passou a levar horas

# A ciência dos compiladores

- Essa ciência permitiu a criação de ferramentas de construção de compiladores
  - Ferramentas específicas para
    - Análise léxica
    - Análise sintática
  - Produzem componentes integráveis entre si
- Na disciplina, focaremos nessas ferramentas

# Aplicações da tecnologia de compiladores

# Aplicações

- Implementação de LPs de alto nível
  - Herança
  - Encapsulamento
- Otimizações para arquiteturas
  - Paralelismo
  - Hierarquias de memórias (cache, múltiplos níveis)

# Aplicações

- Projeto de novas arquiteturas
  - Antes:
    - Primeiro o hardware, depois o compilador
    - No passado, o conjunto de instruções cresceu para facilitar o trabalho do montador
    - CISC (Complex Instruction-Set Computer)
  - Depois:
    - Primeiro o compilador, depois o hardware específico
    - Melhor desempenho, hardware mais simples
    - Melhor “casamento” entre hardware e software
    - RISC (Reduced Instruction-Set Computer)

# Aplicações

- Traduções / transformações
  - Tradução entre duas linguagens no mesmo nível (ou nível parecido)
  - Consulta de banco de dados
    - Interpretador SQL
    - Microsoft LINQ to SQL

# Aplicações

- Ferramentas de produtividade de software
  - Verificação de tipos
  - Verificação de limites
  - Apoio ao programador
    - Quem já usou NetBeans / Eclipse / Visual Studio sabe

# Resumo

- Esse é o objeto do estudo da disciplina
- Vimos:
  - O objetivo dos compiladores
  - Diferenças entre compiladores / interpretadores
  - Relação com LFA
  - Estrutura (etapas e fases) de um compilador
  - Algumas aplicações

# Fim