

# Compiladores

## Introdução à Compilação

**Prof. Edson Alves**

Faculdade UnB Gama

# Sumário

1. Introdução
2. Análise do programa fonte
3. Fases de um compilador
4. Agrupamento de fases
5. Ferramentas

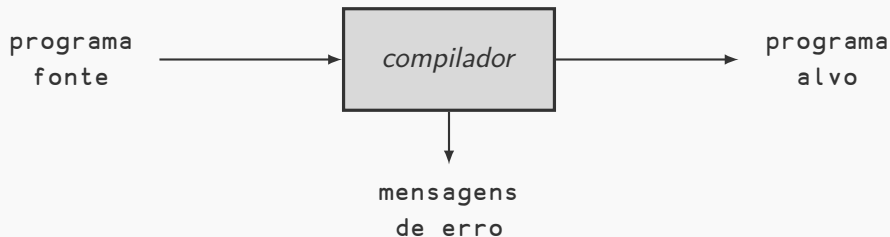
# Histórico

- ▶ Os primeiros compiladores surgiram na década de 50
- ▶ Não há registros preciso de qual foi o primeiro compilador
- ▶ Os primeiros compiladores lidavam com a tradução de fórmulas aritméticas (FORTRAN – *Formula Translator*)
- ▶ Os compiladores eram considerados programas difíceis de se escrever
- ▶ O primeiro compilador Fortran levou 18 homens-ano para ser escrito (1 homen-ano  $\approx$  2.080 horas)
- ▶ Embora continue não sendo uma tarefa não trivial, a escrita de compiladores se beneficiou dos avanços da área desde então

# Definição

## Definição de compilador (informal)

Um compilador é um programa que lê um programa escrito em uma linguagem (linguagem fonte) e o traduz para uma outra linguagem (linguagem alvo).



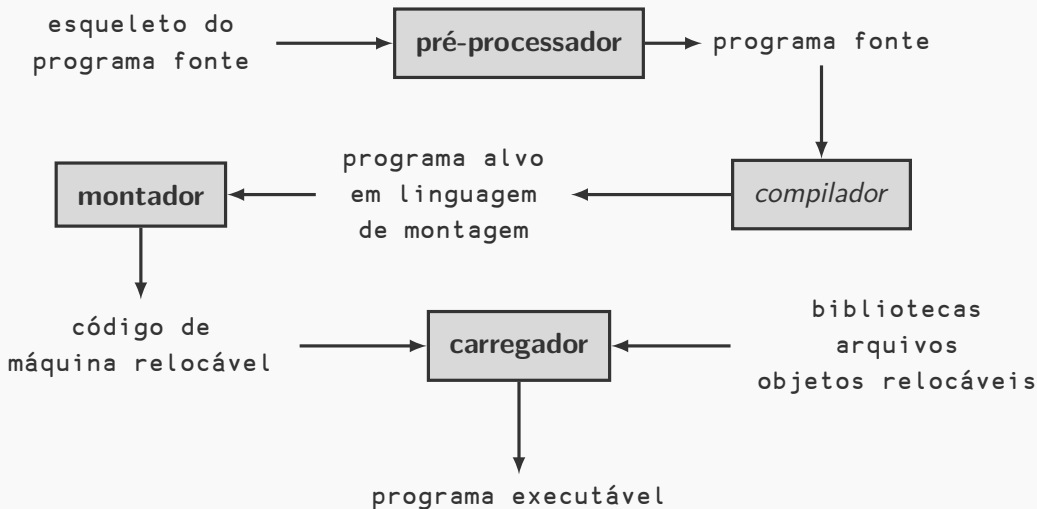
# Características dos compiladores

- ▶ O processo de compilação deve identificar e relatar possíveis erros no programa fonte
- ▶ Em geral, as linguagens fonte são linguagens de programação tradicionais (C/C++, Java, Python, etc)
- ▶ As linguagens alvo podem ser tanto linguagens tradicionais quanto linguagens de máquina
- ▶ Os compiladores podem ser classificados de diversas formas, dependendo de seu objetivo ou como foi construído (de uma passagem, múltiplas passagens, depuradores, etc)

## Criação do programa executável

- ▶ Além do compilador, outros programas podem ser usados na criação do programa executável
- ▶ Antes de ser passado para o compilador, o programa alvo pode ser pré-processado (por exemplo, o pré-processador da linguagem C processa as diretivas como `#include` e `#define`)
- ▶ Após a compilação, o programa alvo pode demandar processamento adicional para a construção do executável (novamente no caso da linguagem C, temos o montador e o *linkeditor*)

## Exemplo de fluxo de geração de um programa executável

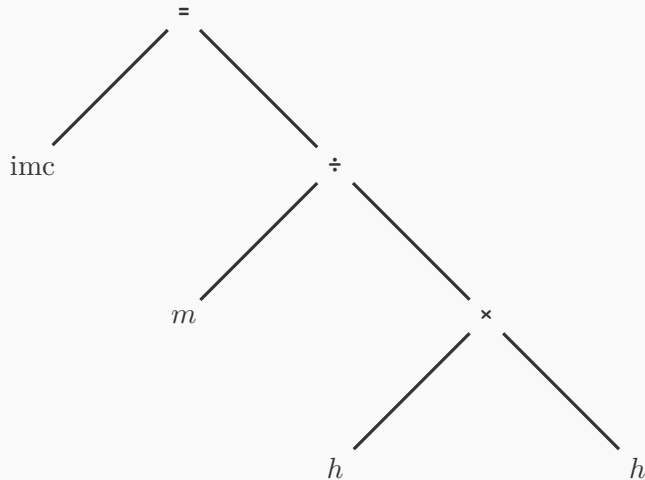


# Análise e síntese

- ▶ A compilação é composta por duas partes: análise e síntese
- ▶ A análise divide o programa fonte em partes constituintes e as organiza em uma representação intermediária
- ▶ Em geral, a representação intermediária consiste em uma árvore sintática, onde cada nó representa uma operação e cada filho representa um operando
- ▶ A síntese constrói o programa alvo a partir desta representação intermediária



## Exemplo de árvore sintática



**Figura:** Árvore sintática da fórmula  $imc = m/h^2$ .

# Análise do programa fonte

A análise é composta por três fases:

1. *análise linear*: o fluxo de caracteres que compõem o programa alvo é lido, da esquerda para direita, e agrupado em *tokens* (sequência de caracteres com significado coletivo)
2. *análise hierárquica*: os *tokens* são ordenados hierarquicamente em coleções aninhadas com significado coletivo
3. *análise semântica*: verificação que garante que os componentes do programa se combinam de forma significativa

# Análise léxica

Em um compilador, a análise linear também é denominada análise léxica ou esquadrinhamento. Por exemplo, no enunciado

$$F = 1.8 * C + 32$$

a análise léxica identificaria os seguintes *tokens*:

1. o identificador `F`
2. o símbolo de atribuição `=`
3. a constante em ponto flutuante `1.8`
4. o símbolo de multiplicação `*`
5. o identificador `C`
6. o símbolo de adição `+`
7. a constante inteira `32`

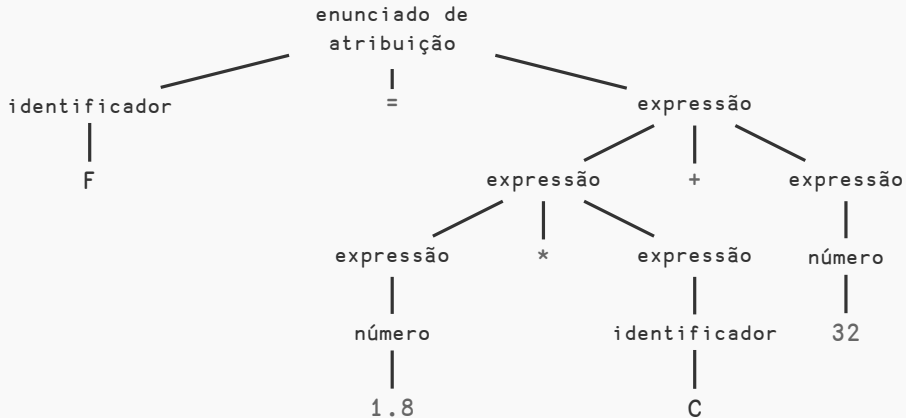
# Análise sintática

A análise hierárquica também é denominada análise sintática ou gramatical. Ela agrupa os *tokens* hierarquicamente, em geral em uma árvore gramatical

A estrutura hierárquica pode ser definida por meio de regras recursivas.

1. qualquer identificador é uma expressão
2. qualquer número é uma expressão
3. se  $E_1$  e  $E_2$  são expressões, também são expressões  $E_1 + E_2$  e  $E_1 * E_2$
4. se  $I$  é um identificador e  $E$  uma expressão, então  $I = E$  é um enunciado

## Exemplo de árvore gramatical



**Figura:** Árvore gramatical do enunciado  $F = 1.8 * C + 32$

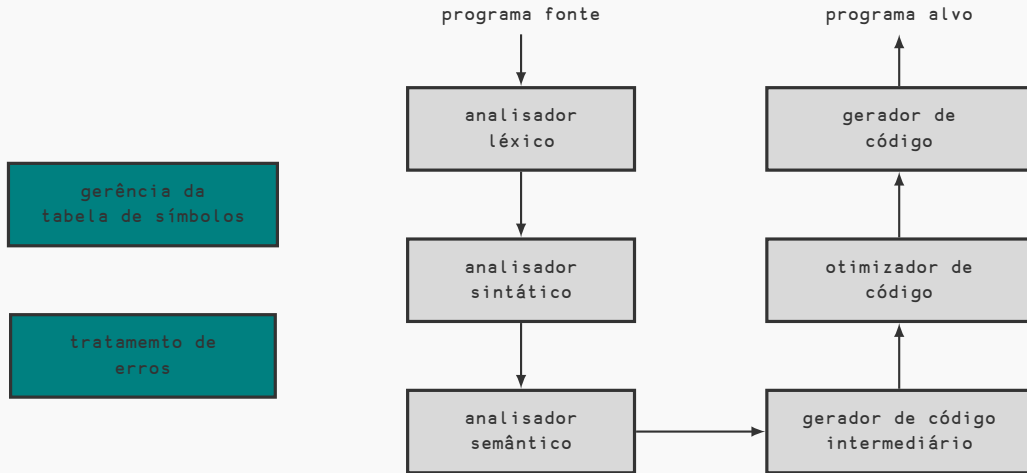
# Análise semântica

- ▶ A análise semântica verifica potenciais erros semânticos no programa alvo
- ▶ Ela usa a árvore da análise sintática para identificar operadores e operandos das expressões e enuncados
- ▶ Ela também faz a verificação de tipos
- ▶ Caso os tipos dos operandos não sejam compatíveis com os tipos esperados pelos operadores, esta análise ou retorna um erro ou adicionar uma promoção (ou conversão) de tipo, a depender da linguagem alvo
- ▶ Por exemplo, na expressão à direita do enunciado  $F = 1.8 * C + 32$ , o operando à esquerda da soma tem tipo ponto flutuante e o da direita tipo inteiro: o valor 32 deve ser promovido para ponto flutuante ou deve ser sinalizado um erro de tipo

# As fases de um compilador

- ▶ Conceitualmente, o compilador opera em fases
- ▶ Cada fase manipula o programa fonte e entrega o resultado para a próxima fase
- ▶ Na prática, algumas fases podem ser agrupadas, e a representação intermediária entre elas pode ser não se construída explicitamente
- ▶ As primeiras fases estão relacionadas à análise do programa fonte, as últimas estão relacionadas à síntese (construção do programa alvo)
- ▶ Duas atividades interagem com todas as fases: a gerência da tabela de símbolos e o tratamento de erros

# Representação típica das fases de um compilador





## Gerenciamento da tabela de símbolos

- ▶ Esta atividade registra os identificadores do programa alvo e identifica seus diversos atributos
- ▶ Exemplos de possíveis atributos de um identificador: nome, tipo, memória e escopo
- ▶ Caso o identificador se refira a um procedimento, dentre seus atributos devem constar a quantidade de seus parâmetros e respectivos tipos, modo de passagem (cópia ou referência) e o tipo do retorno, se houver
- ▶ Os identificadores e seus respectivos atributos são armazenados em uma estrutura denominada tabela de símbolos

## Tratamento de erros

- ▶ A cada fase da compilação podem acontecer um ou mais erros
- ▶ Após a identificação do erro, o compilador deve tratá-lo de alguma maneira e, se possível, continuar o processo em busca de outros erros
- ▶ Abortar a compilação logo no primeiro erro pode diminuir a utilidade do compilador (por exemplo, o prosseguimento da compilação após um erro léxico pode ajudar na geração de uma sugestão de correção para o erro)
- ▶ As análises sintática e semântica podem identificar uma parcela considerável dos erros no programa fonte

## Exemplo da parte da análise do programa fonte

`fahrenheit = 1.8 * celsius + 32`



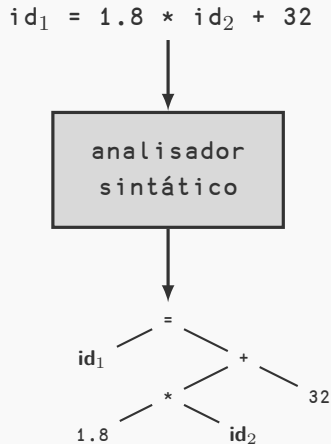
analisador  
léxico



`id1 = 1.8 * id2 + 32`

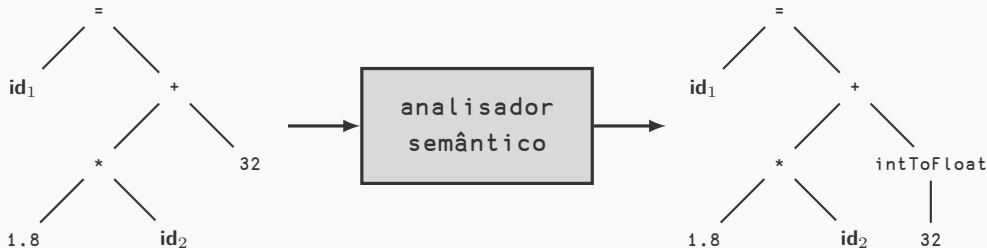
**Figura:** Análise léxica do enunciado `fahrenheit = 1.8 * celsius + 32`

## Exemplo da parte da análise do programa fonte



**Figura:** Análise sintática

## Exemplo da parte da análise do programa fonte

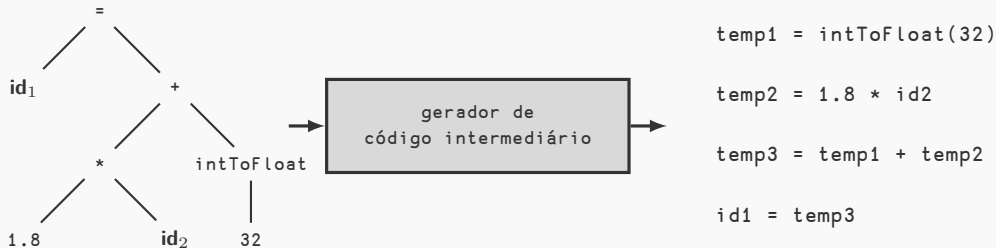


**Figura:** Análise semântica

## Geração de código intermediário

- ▶ A árvore resultante da análise semântica é transformada pelo compilador em um código intermediário
- ▶ Esta representação pode ser entendida como um código para uma máquina abstrata
- ▶ O código intermediário deve ter duas qualidades fundamentais:
  1. deve ser fácil de gerar
  2. deve ser fácil de traduzir para o programa alvo
- ▶ Uma representação possível é o código de três endereços
- ▶ Além de computar expressões, esta representação também precisa tratar dos fluxos de controle e das chamadas de procedimentos

## Exemplo de geração de código intermediário



**Figura:** Representação por código de três endereços

# Otimização do código

- ▶ Esta fase procura formas de melhorar o código intermediário, com o intuito de melhorar a performance do código de máquina do programa alvo
- ▶ Algumas otimizações são triviais, outras demandam algoritmos sofisticados, impactando no tempo de compilação
- ▶ As otimizações não devem alterar a semântica do código intermediário
- ▶ As otimizações podem melhorar, além do tempo de execução, o uso de memória do programa alvo



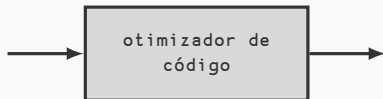
## Exemplo de otimização do código intermediário

```
temp1 = intToFloat(32)
```

```
temp2 = 1.8 * id2
```

```
temp3 = temp1 + temp2
```

```
id1 = temp3
```



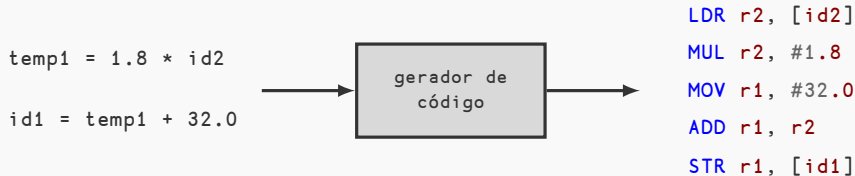
```
temp1 = 1.8 * id2
```

```
id1 = temp1 + 32.0
```

**Figura:** Otimização

## Geração de código

- ▶ A geração de código é a última etapa da compilação
- ▶ Ela produz o programa alvo, em geral em linguagem de máquina relocável ou código de montagem
- ▶ Nesta etapa devem ser atribuídas localizações de memória para as variáveis e também feita a atribuição das variáveis aos registradores



**Figura:** Geração de código em pseudo assembly

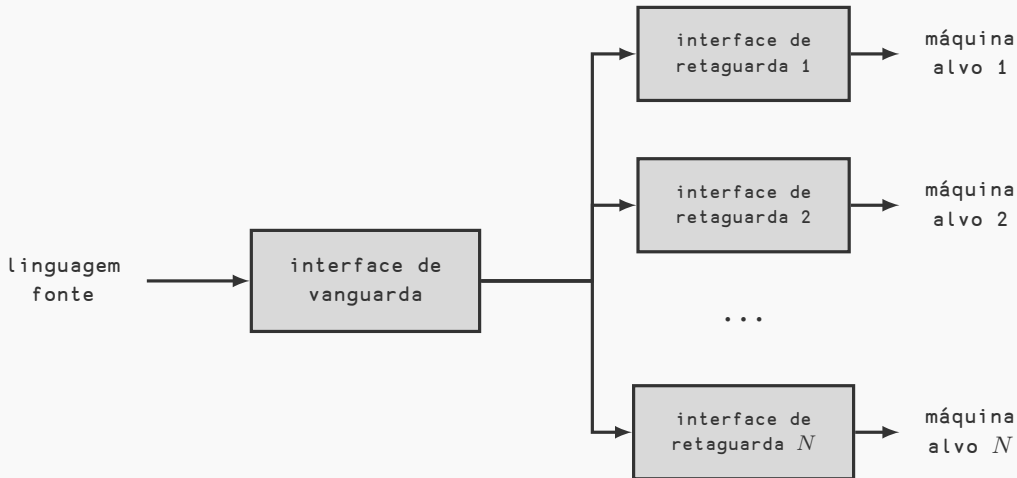
## Interface de vanguarda

- ▶ Na prática, as fases de um compilador são agrupadas em duas interfaces: vanguarda e retaguarda
- ▶ A interface de vanguarda contém as fases que dependem primariamente do programa fonte e que independem da máquina alvo
- ▶ Em geral, ela inclui as fases de análise, a criação da tabela de símbolos e a geração de código intermediário
- ▶ Ela também inclui o tratamento de erros associados a estas fases
- ▶ Embora a otimização faça parte primariamente da interface de retaguarda, é possível aplicar algum nível de otimização na interface de vanguarda

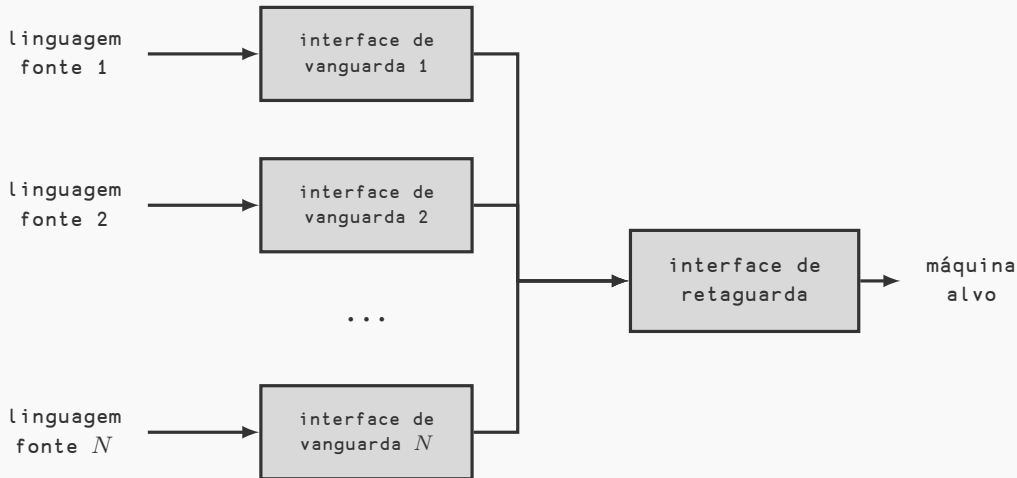
## Interface de retaguarda

- ▶ A interface de retaguarda contém as fases que dependem primariamente da máquina alvo, e independem do programa alvo
- ▶ O ponto de partida é o código intermediário
- ▶ Assim, esta interface contém, em geral, as fases de otimização e geração de código
- ▶ Ela também manipula a tabela de símbolos e trata dos erros associados à estas últimas duas fases
- ▶ No cenário ideal, ambas interfaces são independentes, o que permite fixar uma delas e alterar a outra para obter diferentes compiladores com diferentes objetivos

# Compiladores de uma mesma linguagem para múltiplas máquinas



# Compiladores de múltiplas linguagens para uma mesma máquina



## Ferramentas para a construção de compiladores

- ▶ Sendo o compilador um software, todas as ferramentas úteis no desenvolvimento de um software também serão úteis na construção de um compilador (editor de texto, depuradores, gerenciadores de versão, etc)
- ▶ Existem, entretanto, ferramentas especializadas nas diferentes fases da compilação, as quais podem ser usadas na construção de novos compiladores
- ▶ Os geradores de analisadores gramaticas produzem analisadores sintáticos a partir de uma entrada baseada em uma gramática livre de contexto (Yacc, Bison, etc)
- ▶ Os geradores de analisadores léxicos geram os mesmos a partir de especificações baseadas em expressões regulares (Lex, Flex, etc)

# Ferramentas para a construção de compiladores

- ▶ Os dispositivos de tradução dirigidos pela sintaxe produzem uma coleção de rotinas que percorrem uma árvore gramatical, gerando código intermediário a partir dela
- ▶ Os geradores automáticos de código estipulam regras que traduzem cada operação da linguagem intermediária para a linguagem de máquina alvo
- ▶ Os dispositivos de fluxo de dados atuam na fase de otimização a partir da observação do fluxo de dados entre as diferentes partes de um programa



# Referências

1. **AHO**, Alfred V, **SETHI**, Ravi, **ULLMAN**, Jeffrey D. *Compiladores: Princípios, Técnicas e Ferramentas*, LTC Editora, 1995.
2. GNU.org. [GNU Bison](#), acesso em 23/05/2022.
3. Wikipédia. [Flex \(lexical analyser generator\)](#), acesso em 23/05/2022.