Compiladores

Fabio Mascarenhas – 2015.2

http://www.dcc.ufrj.br/~fabiom/comp

Introdução

- Compiladores x Interpretadores
  - Offline x Online
  - Um compilador transforma um programa *executável* de uma linguagem fonte para um programa *executável* em uma linguagem destino
  - O programa resultante deve ser, de alguma maneira, *melhor* que o original
  - Um interpretador *executa* um programa executável, produzindo o *resultado* do programa
- A maior parte das técnicas que veremos nessa disciplina servem para ambos

Histórico

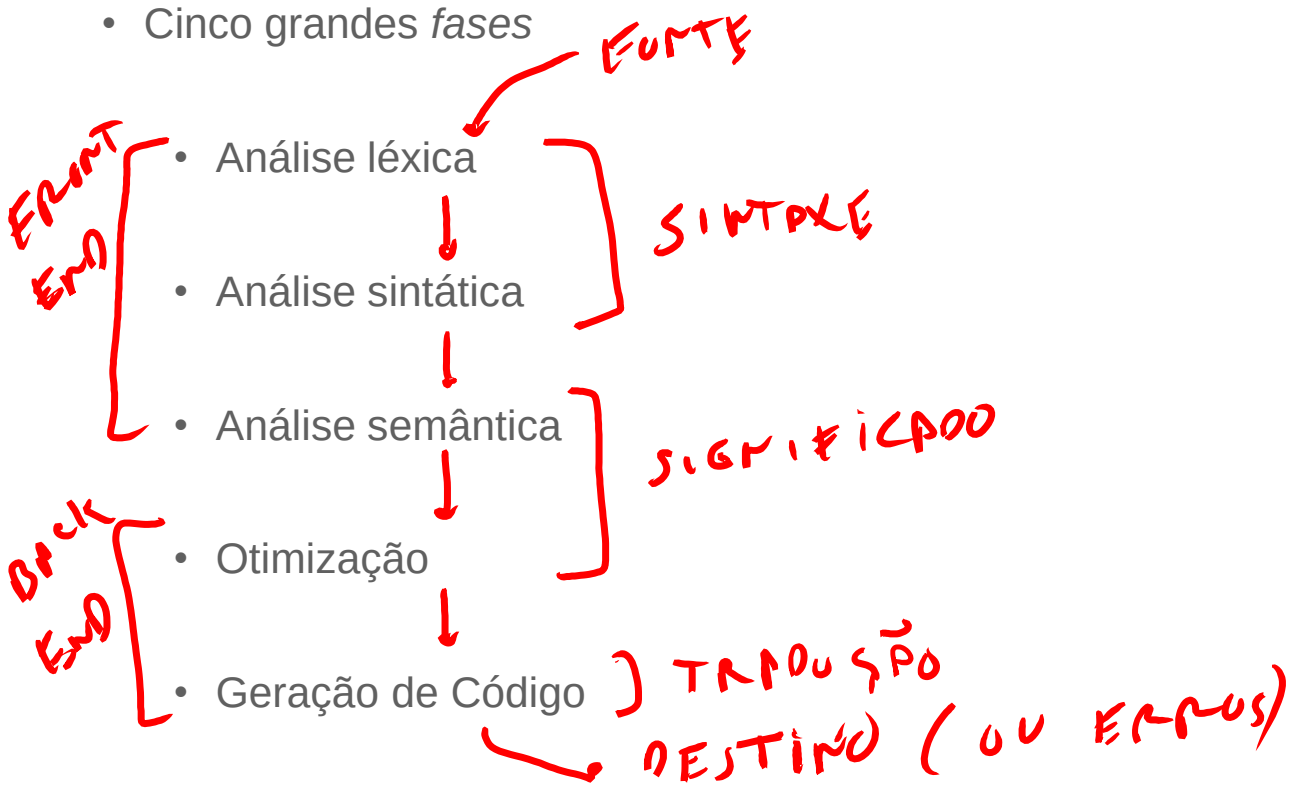
- Inicialmente os computadores eram programados diretamente em linguagem de máquina
  - Não se dava muita importância ao software, ou à produtividade dos programadores!
- Em 1953 John Backus cria na IBM a primeira linguagem de alto nível, *Speedcoding*
  - Interpretada, lenta (10x-20x código de máquina)
- Em 1957 a IBM lança a primeira versão do compilador FORTRAN, o primeiro compilador moderno
  - Gerava código com desempenho similar aos programas escritos diretamente em linguagem de máquina
  - Projeto gerenciado pelo mesmo John Backus, começou em 1954
  - Em 1958 metade dos programas existentes para os mainframes IBM já eram escritos em FORTRAN

Histórico

- A estrutura geral de um compilador moderno ainda se parece com a do primeiro compilador FORTRAN, embora o interior de todas as partes já tenha mudado desde então
- Uma enorme quantidade de pesquisa e desenvolvimento já foi feita desde então
- Muitas das técnicas que vamos ver nesse curso já são bem antigas (30-40 anos), mas a área ainda vai mudando
- Os desafios e o que era importante há 40 anos são diferentes dos desafios e do que é importante hoje

Estrutura Básica de um Compilador

- Cinco grandes *fases*



- As duas primeiras cuidam da *sintaxe* do programa, as duas intermediárias do seu *significado*, e a última da *tradução* para a linguagem destino

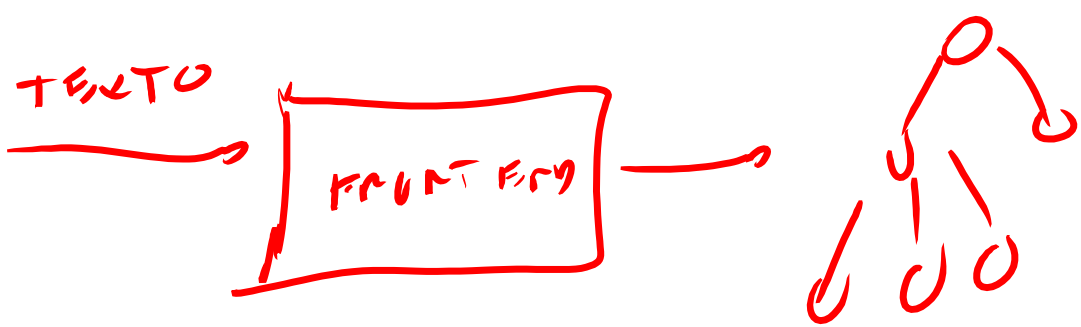
- As três primeiras fases formam o *front-end* do compilador, e as duas outras seu *back-end*

Front-end

- A função do front-end é extrair a *estrutura* do programa, e verificar sua *corretude*

- O front-end produz uma representação do programa como uma *árvore sintática abstrata*

- Caso o programa tenha erros que possam ser detectados em tempo de compilação, o front-end também produz mensagens apontando onde esses erros estão



Análise Léxica

- Primeiro passo do front-end: reconhecer *tokens*
  - Tokens são as palavras do programa
  - O analisador léxico transforma o programa de uma sequência de caracteres sem nenhuma estrutura para uma sequência de tokens
- Ex:
  - if x == y then z = 1; else z = 2;
  - Tokens: if, x, ==, y, then, z, =, 1, ;, else, z, =, 2, ;, EOF

Análise Léxica

- Normalmente o analisador léxico para uma linguagem é produzido mecanicamente a partir uma *especificação léxica* definida por *expressões regulares*

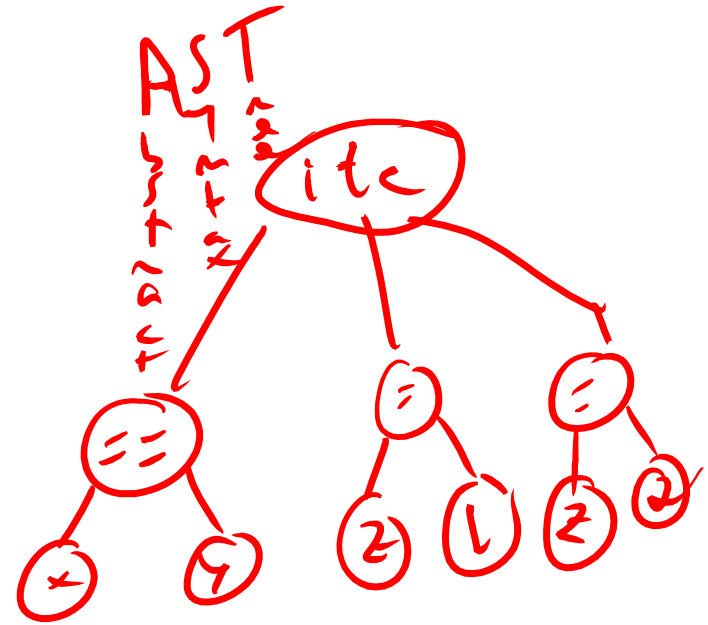
- Um gerador de analisador léxico é um compilador para a sua linguagem de especificação!

- Nesse curso vamos usar o JFlex, que gera analisadores léxicos implementados em Java a partir de uma linguagem de especificação parecida com a do analisador *lex*

- Mas vamos também ver como escrever um analisador léxico “à mão”

Análise Sintática

- O analisador sintático agrupa os tokens em termos sintáticos da linguagem
  - Como sujeito, verbo, objeto, oração, período...
- Ex.: if x == y then z = 1; else z = 2;
  - x == y é uma *expressão relacional*
  - z = 1; e z = 2; são *comandos de atribuição*
  - a frase em si á um *comando if-then-else* composto dessas três partes



Análise Sintática

- Também é comum se gerar mecanicamente um analisador sintático a partir de uma especificação da sintaxe da linguagem, sua *gramática*

- Novamente, um programa gerador de analisadores sintáticos é apenas outro compilador

- O resultado da análise sintática é uma *árvore* representando a estrutura do programa

  - Pode ser *concreta*, codificando toda a estrutura sintática do programa, ou *abstrata*, codificando apenas o essencial

Análise Semântica

- Agora que sabemos a estrutura do programa, podemos tentar entender seu *significado* para detectar erros

- A análise semântica também procura eliminar ambiguidades em relação aos termos do programa

  - Ex: int x = 0; while(x < 10) { int x = 20; print(x); }

- Quando a linguagem permite, a análise semântica também detecta inconsistências entre os *tipos* das variáveis e seus usos

  - Ex: int x = 0; if(x < 5) x = “foo”;

Otimização

- Transformação automática dos programas de modo que eles
  - rodem mais rápido
  - usem menos memória
  - usem menos bateria
  - usem menos a rede
  - em geral, usem menos *recursos* e tenham melhor *desempenho*
- Mistura de análise e síntese, e raramente exato, mas baseado em *heurísticas*
- Requer muita atenção quanto à *corretude*; por ex., x = y * 0 e x = 0 são equivalentes? Depende!

Geração de Código

- Como mapear o programa na linguagem destino

- Dificuldade varia bastante, a depender das características das linguagens fonte e destino, e de quão distantes elas estão

- Ex.: if x == y then z = 1; else z = 2;

  - Em linguagem de máquina x86, assumindo que conseguimos mapear x no registrador EAX, y no EBX e z no ECX, o código acima pode virar:

```
    cmp eax, ebx
    jne l1
    mov ecx, 1
    jmp l2
l1: mov ecx, 2
l2:
```

Uma linguagem de comandos simples

- Tokens: numerais inteiros, identificadores, +, -, (,), =, ;, `print`

- Gramática

```
PROG -> CMD ; PROG
PROG ->
CMD  -> id = EXP
CMD  -> print EXP
EXP  -> EXP + AEXP
EXP  -> EXP - AEXP
EXP  -> AEXP
AEXP -> id
AEXP -> num
AEXP -> ( EXP )
```