Aula 001 - Apresentação da Disciplina

Visão Geral, Histórico e Introdução aos Compiladores

Prof. Rogério Aparecido Gonçalves
rogerioag@utfpr.edu.br

Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento de Computação (DACOM)
Campo Mourão - Paraná - Brasil

Ciência da Computação

BCC36B - Compiladores

Agenda

1. Plano de Ensino
2. Visão Geral
3. Linguagem de Estudo nesta disciplina: `T++`
4. Próximas Aulas

Plano de Ensino

Visão Geral

Introdução

- Um compilador transforma um programa *executável* de uma linguagem fonte para um programa *executável* em uma linguagem destino
  - Linguagem fonte: C/C++, Java Script, HTML, ...
  - Linguagem destino/alvo: *código de máquina*, i.e., código escrito usando as instruções do computador ou uma máquina virtual
    - Mas por que não gerar para outra linguagem de alto nível?
- O programa resultante deve ser, de alguma maneira, **melhor** que o original

$$\textit{Código Fonte} \rightarrow \textit{COMPILADOR} \rightarrow \textit{Programa Alvo}$$

UTFPR

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 (de 10x à 20x em relação ao 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.

UTFPR

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 pesquisas já foram realizadas...
- Muitas das técnicas que vamos ver nesse curso já são bem antigas (30-40 anos), mas a área ainda muda (principalmente em relação às *otimizações* ou *melhoramento de código*).
- Importante lembrar que os desafios de 40 anos atrás são diferentes dos desafios de hoje!

Por que compiladores? E alguns pontos a destacar...

- É inegável a construção de um compilador para facilitar o trabalho da programação
- Já foi dito na literatura que a linguagem de montagem era tão difícil que se tornava inútil
- Técnicas de melhoria de código (e não técnica de otimização)
  - Nunca temos um código ótimo, apesar de mais eficiente...
- Grande número de trabalhos para automatizar a construção:
  - `Yacc` (Steve Johnson, 1975)
  - `Lex` (Mike Lesk, ~ mesma época)
  - `VMs`
  - `IDEs`

Compiladores X Interpretadores

- Um interpretador *executa* um programa executável, produzindo *resultados* inerentes ao programa fonte.
- As técnicas que iremos aprender é aplicada para **ambos**.
- Um compilador é um sistema de software de grande porte, com muitos componentes e algoritmos internos e interações complexas entre eles.

Programas relacionados a compiladores

- Interpretadores
- Montadores
- Organizadores
- Carregadores
- Pré-processadores
- Editores
- Depuradores
- Geradores de perfil
- Gerenciadores de projetos
- Muitos outros sistemas podem se qualificar como *compiladores*:
  - Programa de composição tipográfica que produz `PostScript` (linguagem para descrever imagens)
    - O programa possui uma especificação executável para produzir outra também executável
  - `Markdown` (uso do `pandoc`)
  - `Latex`

UTFPR

Tradutores

- O código que transforma `PostScript` em pixels é normalmente um `interpretador`, não um compilador

$$\textit{Programa Fonte} \rightarrow \textit{INTERPRETADOR} \rightarrow \textit{Resultados}$$

- Linguagens (geralmente) interpretadas: `Perl`, `Scheme`, `APL`, `Python`, `R`, `Ruby`, etc.
- Algumas linguagens adotam tanto o esquema de compilação quanto de tradução:
  - `Java` é compilada do código-fonte para um formato denominado *bytecode*
  - Aplicativos `Java` são executados na `JVM` utilizando o *bytecode* gerado.
    - Este *bytecode* é interpretado pela `JVM` para gerar resultados.

UTFPR

Princípios fundamentais da compilação

- Temos que ter sempre em mente:
  - *O compilador deve preservar o significado do programa a ser compilado*
    - Exatidão é fundamental! Deve ser fiel ao significado do programa... Caso contrário, por que fazê-lo?
  - *O compilador deve melhorar o programa de entrada de alguma forma perceptível*
    - Deve facilitar o desenvolvimento de alguma forma ou gerar mais segurança: abstração e métodos de acesso respectivamente.
  - Controversas em relação aos motivos de criar um compilador
    - (empresa): Transformou os códigos `Smalltalk` em `Java`.

UTFPR

Principais estruturas de dados de um compilador

- Marcas (*tokens*)
- Árvore Sintática
- Tabela de Símbolos
- Tabela de Literais
- Código Intermediário
- Arquivos temporários

Grandes fases de compilação

- Análise Léxica
- Análise Sintática
- Análise Semântica
- Otimização do código fonte
- Geração de código intermediário
- Otimização do código alvo

Outros aspectos de um compilador

- Análise e síntese
- Frente e fundo (*Frontend* e *Backend*)
- Passadas
- Definição de linguagem e compiladores
- Opções e interfaces de um compilador
- Tratamento de erros

Estrutura mais detalhada do compilador

1. Código fonte $\rightarrow$ `[Sistema de varredura (análise léxica)]`
2. Marcas (*tokens*) $\rightarrow$ `[Analisador sintático]`
3. Árvore sintática $\rightarrow$ `[Analisador semântico]`
4. Árvore anotada $\rightarrow$ `[Otimizador de código-fonte]`
5. Código intermediário $\rightarrow$ `[Gerador de código]`
6. Código-alvo $\rightarrow$ `[Otimizador de código-alvo]`
7. Código-alvo $\rightarrow$ `[Executado]`

UTFPR

Linguagem de Estudo nesta disciplina: T++

Esquemas principais da linguagem: T++

- tipo **inteiro** e **flutuante**;
- deve suportar arranjos (**arrays**) – tipo: identicador[tamanho];
- variáveis locais e globais devem ter um dos tipos especificados;
- tipos de funções pode ser omitidos (quando omitidos viram um **procedimento** e um tipo **void** é devolvido explicitamente;
- linguagem quase fortemente tipificada: nem todos os erros são especificados mas sempre deve ocorrer avisos;
- ...

Exemplos na linguagem T++

```
inteiro: n

inteiro fatorial(inteiro: n)
    inteiro: fat
    se n > 0 então {não calcula se n > 0}
        fat := 1
        repita
            fat := fat * n
            n := n - 1
        até n = 0
        retorna(fat) {retorna o valor do fatorial de n}
    senão
        retorna(0)
    fim
fim

inteiro principal()
    leia(n)
    escreva(fatorial(n))
    retorna(0)
fim
```

Exemplos na linguagem T++

```
flutuante: A[1024]
flutuante: B[1024]
flutuante: C[1024]

somaVetores(inteiro: n)
    inteiro: i
    para i := 0 até n-1 faça
        C[i] := A[i] + B[i]
    fim
fim

inteiro principal()
    inteiro: i
    para i := 0 até 1023 faça
        A[i] := 1
        B[i] := 1
    fim

    somaVetores(1024)

    para i := 0 até 1023 faça
        escreva(C[i])
    fim
```

UTFPR

Leitura Recomendada

Capítulo 1: Introdução

- LOUDEN, Kenneth C. Compiladores: princípios e práticas. São Paulo, SP: Thomson, c2004. xiv, 569 p. ISBN 8522104220.

Próximas Aulas

Próximas Aulas: Análise Léxica

- Expressões regulares
- Autômatos finitos
- Implementação da análise léxica
- Utilização de ferramentas