Compiladores

Prof. Rodrigo Martins
rodrigo.martins45@francomontoro.com.br

Cronograma da Aula

- Aula Inaugural
- Apresentação da disciplina
- Conceitos Iniciais

Ementa

- Linguagens e tradutores.
- Compiladores e Interpretadores.
- A estrutura de um compilador.
- Análise Léxica.
- Análise Sintática.
- Representação Intermediária.
- Análise Semântica.
- Geração e Otimização de Código. Interpretadores.
- Laboratório com ferramentas de auxílio à construção de compiladores.

Objetivos

- Apresentar ao aluno as diferentes técnicas e ferramentas na construção de compiladores ou interpretadores para uma linguagem de alto nível.

- Capacitar o aluno no desenvolvimento de analisadores léxicos e sintáticos, utilizando técnicas de análise semântica, de geração e de otimização de códigos.

- Ao final da disciplina, o aluno estará capacitado a entender e construir um compilador para uma linguagem de alto nível.

Critérios de Avaliação

- T1 – Lista de Exercícios 1 (30%)
- P1 - Avaliação Bimestral 1 (70%)

- T2 - Lista de Exercícios 2 (30%)
- P2 – Avaliação Bimestral 2 (70%)

- Média Final – (MB1 + MB2) / 2

- Média final maior ou igual a 7,0 (sete) implicará em aprovação sem exame final;
- Média final igual ou superior a 4,0 (quatro) e inferior a 7,0 (sete) dependerá de aprovação em exame final;
- Média final de aproveitamento inferior a 4,0 (quatro) implicará em reprovação;
- A aprovação em exame final será obtida se a média aritmética da média final de aproveitamento com a nota do exame final for igual ou superior a 5,0 (cinco).

Metodologias de Trabalho

- Material exposto em sala de aula (Apresentações);
- Indicação de Sites sobre o conteúdo (Artigos);
- Pesquisas;
- Uso de metodologias ativas no desenvolvimento de projetos.

Bibliografia Básica

- GRUNE, D. et. al. Projeto Moderno de Compiladores, Campus, Rio de Janeiro, 2001.

- Louden, Kenneth. Compiladores: princípios e práticas. Thomson Pioneira, 2004.

Bibliografia Complementar

- PRICE, A. M. A. e TOSCANI, S. S. Implementação de Linguagens de Programação: Compiladores, Sagra Luzzatto, Porto Alegre, 2ª ed, 2001.

- AHO, A. V. Compiladores. Princípios, Técnicas e Ferramentas, Editora LTC, 1996.

O que é um Compilador?

- É um programa que traduz um código fonte escrito em uma linguagem de programação de alto nível para código de máquina executável pelo computador.

- O processo de compilação é realizado em etapas, que envolvem análise léxica, análise sintática, análise semântica, geração de código intermediário, otimização de código e geração de código final.

O que é um Compilador?

- O compilador recebe um arquivo de código fonte escrito em uma linguagem de programação, analisa e valida a sua sintaxe, identifica possíveis erros semânticos, gera um código intermediário que possa ser otimizado e, finalmente, gera um código de máquina que possa ser executado diretamente pelo computador.

Qual a importância dos compiladores para a programação?

- São muito importantes para a programação, pois permitem que os programadores escrevam programas em linguagens de alto nível, que são mais fáceis de entender e de escrever do que o código de máquina diretamente.

- Os compiladores fazem a tradução do código fonte para código de máquina, que é a linguagem que o computador entende e executa.

Algumas das principais importâncias dos compiladores para a programação

Produtividade

- Os compiladores permitem que os programadores escrevam programas em linguagens de alto nível, que são mais fáceis de entender e de escrever do que o código de máquina diretamente.

- Isso aumenta a produtividade do programador, pois ele pode se concentrar na lógica do programa e não precisa se preocupar com detalhes da arquitetura do computador.

Algumas das principais importâncias dos compiladores para a programação

Portabilidade

- Os compiladores permitem que um mesmo programa seja executado em diferentes plataformas e sistemas operacionais, desde que haja um compilador disponível para a linguagem de programação utilizada.

- Isso aumenta a portabilidade do programa, pois ele pode ser distribuído para diferentes sistemas sem a necessidade de modificações significativas.

Algumas das principais importâncias dos compiladores para a programação

Eficiência

- Os compiladores geram um código de máquina otimizado, que é executado mais rapidamente pelo computador.

- Isso aumenta a eficiência do programa, o que é especialmente importante em programas que precisam processar grandes quantidades de dados ou que precisam ser executados em tempo real.

Algumas das principais importâncias dos compiladores para a programação

Detecção de erros

- Os compiladores podem detectar erros de programação antes da execução do programa, o que ajuda a economizar tempo e esforço na fase de depuração.

- Isso é especialmente importante em programas grandes e complexos, que podem conter muitos erros difíceis de serem detectados manualmente.

Algumas das principais importâncias dos compiladores para a programação

Detecção de erros

- Os compiladores podem detectar erros de programação antes da execução do programa, o que ajuda a economizar tempo e esforço na fase de depuração.

- Isso é especialmente importante em programas grandes e complexos, que podem conter muitos erros difíceis de serem detectados manualmente.

Exemplos de linguagens compiladas

- **C:** é uma linguagem compilada, e os compiladores C são amplamente disponíveis para várias plataformas, incluindo Windows, Linux, macOS e outras.

- **C++:** é compilada em código de máquina para ser executado em diferentes plataformas.

Exemplos de linguagens compiladas

- **Pascal**: é compilada em código de máquina para ser executada em diferentes plataformas.

- **Cobol**: É uma linguagem de programação de negócios que é amplamente usada em mainframes para processar grandes quantidades de dados. COBOL é compilada em código de máquina para ser executada em diferentes plataformas.

O que é um interpretador?

- É um programa que lê e executa um código fonte diretamente, linha por linha, sem a necessidade de uma etapa prévia de compilação.

- O interpretador analisa o código fonte e executa as instruções nele contidas, sem gerar um código de máquina separado.

O que é um interpretador?

- Os interpretadores podem ser mais lentos do que os programas compilados, uma vez que precisam analisar o código fonte linha por linha antes de executá-lo.

- Isso pode ser um problema em programas que precisam processar grandes quantidades de dados ou que precisam ser executados em tempo real. Além disso, os interpretadores podem ser menos seguros do que programas compilados, pois os erros de programação só são detectados durante a execução do programa.

Exemplos de linguagens interpretadas

- **Lisp**: uma das linguagens de programação mais antigas ainda em uso, usada em inteligência artificial, processamento de linguagem natural, entre outros.

Exemplos de linguagens interpretadas

- **Tcl**: uma linguagem de programação fácil de aprender e usar, usada em automação de tarefas, desenvolvimento web, entre outros.

Exemplos de linguagens interpretadas

- **BASIC**: uma linguagem de programação popular em microcomputadores na década de 1970 e 1980, ainda usada em sistemas legados e em sistemas educacionais.

Exemplos de linguagens interpretadas

- **Perl**: uma linguagem de programação de alto nível usada em sistemas Unix e em desenvolvimento web.

Sistemas de Interpretação Híbridos

- Os sistemas de interpretação híbridos são sistemas que combinam técnicas de interpretação e compilação para executar programas.

- A ideia por trás desse tipo de sistema é combinar as vantagens dos sistemas de interpretação e dos sistemas de compilação para criar um sistema mais eficiente e flexível.

Sistemas de Interpretação Híbridos

- Os sistemas de interpretação híbridos combinam os benefícios dos sistemas de interpretação e de compilação.
- Esses sistemas utilizam técnicas de compilação para gerar um código intermediário otimizado, que é então interpretado pelo sistema.
- O código intermediário pode ser gerado de forma rápida, permitindo que o sistema seja mais flexível e adaptável às mudanças no código fonte.
- Além disso, o código intermediário pode ser otimizado antes da execução, o que pode resultar em uma execução mais rápida do que a interpretação de um código fonte sem otimizações.

Exemplos de linguagens com sistemas de Interpretação Híbridos

- **Java**: uma linguagem de programação de alto nível e orientada a objetos, que é compilada para bytecode e depois interpretada por uma máquina virtual Java (JVM).

- **C#**: uma linguagem de programação de alto nível e orientada a objetos, que é compilada para código intermediário e depois interpretada pelo Common Language Runtime (CLR).

Exemplos de linguagens com sistemas de Interpretação Híbridos

- **PHP**: embora seja geralmente considerada uma linguagem de programação interpretada, o PHP pode ser compilado em bytecode usando o Zend Engine e, em seguida, interpretado.

- **Ruby**: semelhante ao PHP, o Ruby pode ser compilado em bytecode usando o mecanismo YARV e, em seguida, interpretado.

Exemplos de linguagens com sistemas de Interpretação Híbridos

- **JavaScript**: enquanto a maioria das implementações do JavaScript é interpretada, o Google V8 é uma implementação que compila o código JavaScript para código de máquina.

- **Lua**: uma linguagem de script leve, que pode ser compilada em bytecode e, em seguida, interpretada por uma máquina virtual Lua.

- **Swift**: uma linguagem de programação de alto nível e orientada a objetos, que pode ser compilada para código intermediário e, em seguida, interpretada pelo Swift Playground.

Exemplos de linguagens com sistemas de Interpretação Híbridos

- **Python**: é uma linguagem de programação interpretada. Isso significa que o código fonte em Python é interpretado diretamente pelo interpretador Python, em vez de ser compilado para código de máquina executável.

- A implementação da linguagem Python inclui um compilador JIT (Just-in-Time), que pode otimizar o código em tempo de execução para melhorar o desempenho.

- Além disso, existem ferramentas como o Cython e o Numba, que permitem que parte do código Python seja compilado em código de máquina para melhorar ainda mais o desempenho em determinadas situações.

Interpretador X Compilador

- A principal diferença entre interpretadores e compiladores é o processo de conversão do código fonte para código de máquina.

- O compilador converte todo o código fonte de uma vez só em um arquivo executável ou em um código intermediário que é posteriormente convertido em um arquivo executável.

- Já o interpretador lê cada linha do código fonte em tempo real e a converte em código de máquina e executa imediatamente.

Interpretador X Compilador

Outras diferenças incluem:

- Compiladores geralmente geram um arquivo executável, enquanto interpretadores executam o código diretamente a partir do código fonte.

- O tempo de execução de um programa compilado geralmente é mais rápido do que o tempo de execução de um programa interpretado.

Interpretador X Compilador

Outras diferenças incluem:

- Interpretadores tendem a ser mais fáceis de depurar, pois permitem que o programador execute o código linha por linha.

- Compiladores permitem que um programa seja distribuído em diferentes plataformas sem a necessidade de um ambiente de execução específico. Já o interpretador depende do ambiente de execução específico em que está sendo executado.

As diferentes fases do processo de compilação

Análise léxica

- A primeira fase do processo de compilação é a análise léxica. Nesta fase, o código fonte é dividido em tokens (palavras-chave, identificadores, números, símbolos etc.) pelo analisador léxico.

- O analisador léxico produz uma lista de tokens, que é a entrada para a próxima fase.

As diferentes fases do processo de compilação

Análise sintática

- A segunda fase do processo de compilação é a análise sintática. Nesta fase, a lista de tokens produzida na fase anterior é analisada pelo analisador sintático para determinar se o código fonte está sintaticamente correto e produzir uma árvore sintática que representa a estrutura gramatical do programa.

As diferentes fases do processo de compilação

Análise semântica

- A terceira fase do processo de compilação é a análise semântica. Nesta fase, o compilador verifica se o código fonte está semanticamente correto e produz uma tabela de símbolos que contém informações sobre as variáveis, funções e outras entidades do programa.

As diferentes fases do processo de compilação

Otimização de código

- A quarta fase do processo de compilação é a otimização de código. Nesta fase, o compilador realiza várias transformações no código fonte para melhorar o desempenho e a eficiência do programa resultante. Isso pode incluir a eliminação de código redundante, a substituição de operações demoradas por operações mais rápidas e outras técnicas de otimização.

As diferentes fases do processo de compilação

Geração de código

- A quinta e última fase do processo de compilação é a geração de código. Nesta fase, o compilador produz o código de máquina executável a partir da árvore sintática e da tabela de símbolos geradas nas fases anteriores.

As linguagens formais e como elas são utilizadas em compiladores

- Linguagens formais são conjuntos de regras e símbolos que são usados para descrever e definir uma linguagem, independentemente da sua semântica ou significado.

- Essas regras e símbolos são definidos de forma matemática e lógica, o que permite que elas sejam precisas e inequívocas.

As linguagens formais e como elas são utilizadas em compiladores

- Em compiladores, as linguagens formais são usadas para descrever a gramática da linguagem de programação que está sendo compilada.

- A gramática de uma linguagem de programação é uma descrição formal da sua sintaxe, ou seja, das regras que determinam a estrutura das instruções da linguagem.

As linguagens formais e como elas são utilizadas em compiladores

- As linguagens formais também são usadas para criar as tabelas de símbolos, que contêm informações sobre as variáveis, funções e outras entidades do programa.

- As tabelas de símbolos são usadas para verificar se o código fonte está semanticamente correto e para associar os símbolos do código fonte aos endereços de memória correspondentes no código de máquina gerado pelo compilador.

Conjuntos de regras das linguagens formais

- Os conjuntos de regras das linguagens formais podem ser expressos em diferentes notações, mas as mais comuns são as expressões regulares, as gramáticas formais e as máquinas de estado finito.

- Expressões regulares:

  - [0-9]+: qualquer sequência de dígitos.
  - a(b|c)*d: qualquer sequência que começa com "a", termina com "d" e tem zero ou mais ocorrências das letras "b" ou "c" entre elas.
  - .*: qualquer sequência de caracteres.

Conjuntos de regras das linguagens formais

- Gramáticas formais:
  - Gramática livre de contexto para expressões aritméticas simples:

```
<expression> ::= <term>
               | <expression> + <term>
               | <expression> - <term>
<term>       ::= <factor>
               | <term> * <factor>
               | <term> / <factor>
<factor>     ::= <number>
               | <variable>
               | ( <expression> )
```

Conjuntos de regras das linguagens formais

- A gramática do slide anterior define que uma expressão aritmética pode ser composta de um termo ou de uma expressão somada ou subtraída de outro termo. Um termo pode ser composto de um fator ou de um termo multiplicado ou dividido por outro fator. Um fator pode ser um número, uma variável ou uma expressão entre parênteses.

- Com base nessa gramática, é possível construir árvores sintáticas para expressões aritméticas na linguagem C, o que pode ser útil em um compilador para verificar a correção da sintaxe de um programa e gerar código de máquina correspondente.

Exemplo 1

- Exemplo prático de como um compilador funciona, mostrando como um programa em uma linguagem de programação é convertido em código de máquina.

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    float num1, num2, num3, media;
    printf("Digite o primeiro numero: ");
    scanf("%f", &num1);
    printf("Digite o segundo numero: ");
    scanf("%f", &num2);
    printf("Digite o terceiro numero: ");
    scanf("%f", &num3);
    media = (num1 + num2 + num3) / 3.0;
    printf("A media dos numeros eh %.2f\n", media);
    return 0;
}
```

Exemplo 1

- O processo de compilação desse programa pode ser dividido nas seguintes etapas:

- **Análise léxica:** O código fonte é analisado pelo analisador léxico, que divide o programa em tokens.

- Por exemplo, as palavras-chave **include**, **int**, **float**, **return** e **scanf** são reconhecidas como **tokens**, bem como os nomes de variáveis, operadores e outros símbolos no programa.

Exemplo 1

- **Análise sintática:**

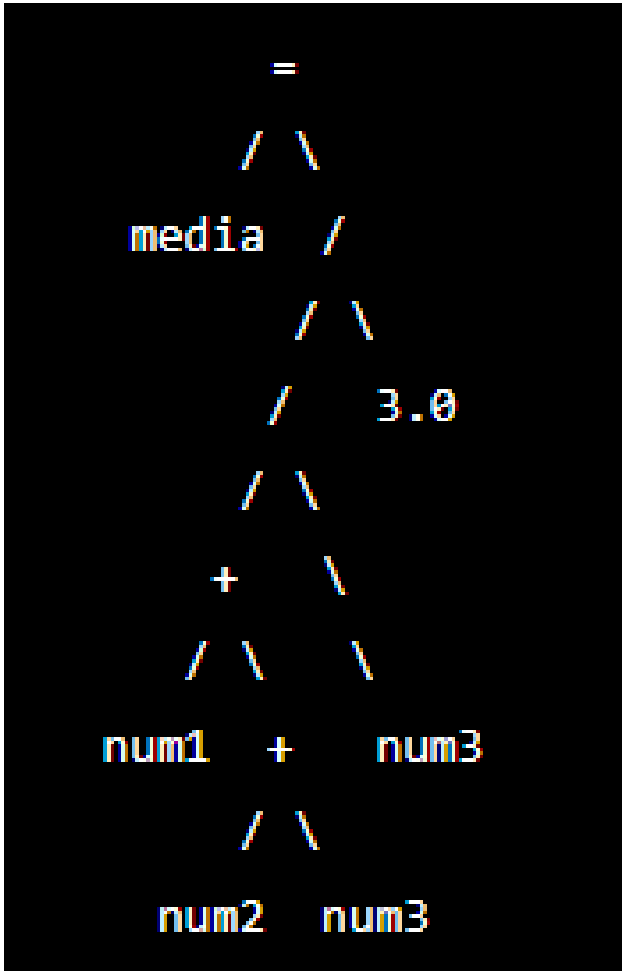
- O analisador sintático usa as regras da gramática da linguagem C para verificar se o código fonte está sintaticamente correto.

- Isso envolve a construção de uma árvore sintática que representa a estrutura do programa.

Exemplo 1

- Nesta árvore, o operador principal é o sinal de igual **(=)**, que é a raiz da árvore. O operando esquerdo é a variável **"media"**. O operando direito é a subexpressão **(num1 + num2 + num3) / 3.0**.

- A subexpressão é representada por um nó com o operador de divisão **(/)** e seus operandos são a subexpressão da esquerda e a constante **3.0**. A subexpressão da esquerda é representada por um nó com o operador de adição **(+)** e seus operandos são as variáveis **num1, num2 e num3**.



Exemplo 1

- **Análise semântica:**

- O compilador verifica se o programa está semanticamente correto, ou seja, se as variáveis são declaradas antes de serem usadas e se as operações são aplicáveis aos tipos de dados corretos.

Exemplo 1

- **Otimização de código:**

- O compilador realiza várias transformações no código para melhorar o desempenho e a eficiência do programa resultante.

- Isso pode incluir a eliminação de código redundante e a simplificação de expressões.

Exemplo 1

- **Geração de código:** O compilador gera o código de máquina a partir do programa em C.

- O código de máquina é uma sequência de instruções em linguagem de baixo nível que pode ser executada diretamente pelo processador.

- Por exemplo, a linha **scanf("%f", &num1)** pode ser traduzida para a instrução de máquina correspondente que lê um valor em ponto flutuante da entrada padrão e armazena o valor na memória na posição de endereço de **num1**.

Exercícios

1) Escreva a diferença entre uma linguagem de programação compilada e uma linguagem de programação interpretada. Forneça exemplos para cada uma dessas categorias.

2) Escreva a importância de seguir uma gramática formal quando se desenvolve um compilador e como ela ajuda a garantir a correta compilação de um programa.

3) Como um compilador traduz um programa em linguagem de alto nível em código de máquina? Explique o processo passo a passo.

Exercícios

4) Explique como a análise sintática é usada no processo de compilação e por que é um passo importante na tradução de um programa de linguagem de alto nível para código de máquina.

5) Descreva a diferença entre análise léxica e análise sintática em um compilador e explique por que ambos são necessários.

Referências desta aula

- GRUNE, D. et. al. Projeto Moderno de Compiladores, Campus, Rio de Janeiro, 2001.

- Louden, Kenneth. Compiladores: princípios e práticas. Thomson Pioneira, 2004.

FIM

OBRIGADO

RODRIGO