Compiladores

CIÊNCIA DA COMPUTAÇÃO

Prof. Dr. João Paulo Aramuni

UNIVERSIDADE FUMEC

Sumário

* **Conceitos Básicos de Compiladores**
  * Introdução e Conceituação Geral
  * Tradutores, Compiladores e Interpretadores
  * Estrutura de um compilador
    * Compilação
    * Passos
    * Fases da compilação

Introdução e Conceituação Geral

* Linguagens de programação são notações para se descrever computações para pessoas e para máquinas.

* O mundo conforme o conhecemos depende de linguagens de programação, pois todo o software executando em todos os computadores foi escrito em alguma linguagem de programação.

Introdução e Conceituação Geral

* Antes que possa rodar, um programa primeiro precisa ser traduzido para um **formato** que lhe permita ser executado por um computador.

* Os sistemas de software que fazem essa tradução são denominados *compiladores*.

Introdução e Conceituação Geral

* O estudo da escrita de compiladores é interdisciplinar.

* Abrange linguagens de programação, arquitetura de máquina, teoria de linguagem, algoritmos e engenharia de software.

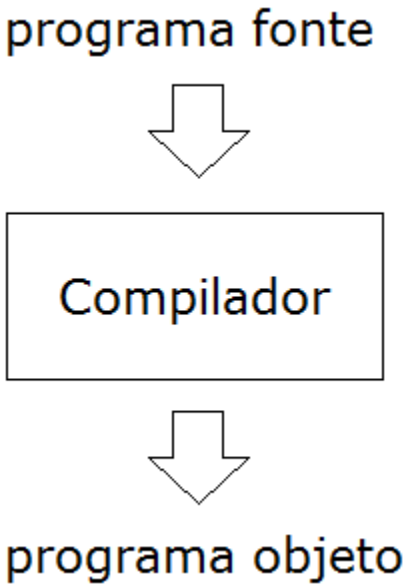
Tradutores, Compiladores e Interpretadores

* Colocando de uma forma bem simples, um compilador é 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***.

* Um papel importante do compilador é relatar quaisquer erros no programa fonte detectados durante esse processo de tradução.

Tradutores, Compiladores e Interpretadores

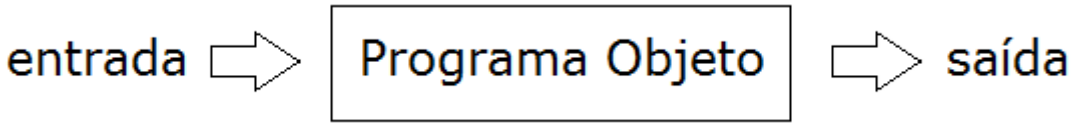
* Um compilador, Fig. 1:



Tradutores, Compiladores e Interpretadores

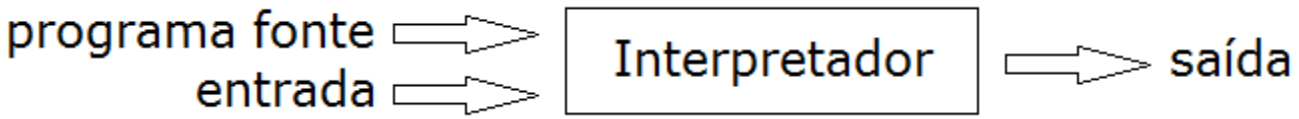
* Se o programa objeto for um programa em uma linguagem de máquina executável, poderá ser chamado pelo usuário para processar entradas e produzir saída.

* Executando o programa objeto, Fig. 2:



Tradutores, Compiladores e Interpretadores

* Um *interpretador* é outro tipo comum de processador de linguagem. Em vez de produzir um programa objeto como resultado da tradução, um interpretador executa diretamente as operações especificadas no programa fonte sobre as entradas fornecidas pelo usuário, Fig. 3:



Tradutores, Compiladores e Interpretadores

* O programa objeto em linguagem de máquina produzido por um compilador normalmente é muito mais rápido no mapeamento das entradas para saídas do que um interpretador.

* Porém, um interpretador frequentemente oferece um melhor diagnóstico de erro do que um compilador, pois executa o programa fonte instrução por instrução.

Tradutores, Compiladores e Interpretadores

* Em um interpretador, o programa conversor recebe a primeira instrução do programa fonte, confere para ver se está escrita corretamente, converte-a em linguagem de máquina e então ordena ao computador que execute esta instrução.

* Depois repete o processo para a segunda instrução, e assim sucessivamente, até a última instrução do programa fonte. Quando a segunda instrução é trabalhada, a primeira é perdida, isto é, apenas uma instrução fica na memória em cada instante.

Tradutores, Compiladores e Interpretadores

* Se este programa fonte for executado uma segunda vez, novamente haverá uma nova tradução, comando por comando, pois os comandos em linguagem de máquina não ficam armazenados para futuras execuções.

* Neste método, o programa conversor recebe o nome de *interpretador*.

* Exemplos de linguagens interpretadas: Perl, PHP, ShellScript...

Tradutores, Compiladores e Interpretadores

* Nos compiladores, programa conversor recebe a primeira instrução do programa fonte, a confere para ver se está escrita corretamente, a converte para linguagem de máquina em caso afirmativo e passa para a próxima instrução, repetindo o processo sucessivamente até a última instrução do programa fonte.

* Caso tenha terminado a transformação da última instrução do programa fonte e nenhum erro tenha sido detectado, o computador volta à primeira instrução, já transformada para linguagem de máquina e a executa. Passa à instrução seguinte, a executa, etc., até a última.

Tradutores, Compiladores e Interpretadores

* Se este programa for executado uma segunda vez, não haverá necessidade de uma nova tradução, uma vez que todos os comandos em linguagem binária foram memorizados em um novo programa completo.

* Neste método, o programa conversor recebe o nome de *compilador*.

* Exemplos de linguagens compiladas: Delphi, C++, Java...

Tradutores, Compiladores e Interpretadores

* **Vantagem:** Neste processo a execução fica mais rápida em relação à anterior, pois se economiza o tempo de retradução de cada instrução a cada nova execução.

* **Desvantagem:** A cada modificação introduzida no programa fonte é necessária uma nova tradução completa para obter um novo programa objeto, o que torna o processo mais dificultoso na fase de desenvolvimento, quando muitas modificações são feitas.

Tradutores, Compiladores e Interpretadores

* Um compilador que traduz linguagem de alto nível para outra linguagem de alto nível é chamada de *tradutor* de fonte para fonte ou conversor de linguagem.

* Um programa que traduz uma linguagem de programação de baixo nível para uma linguagem de programação de alto nível é um *descompilador*.

* Exemplo de descompilador para Java: **JAD** Decompiler: https://goo.gl/A27Yi7

Tradutores, Compiladores e Interpretadores

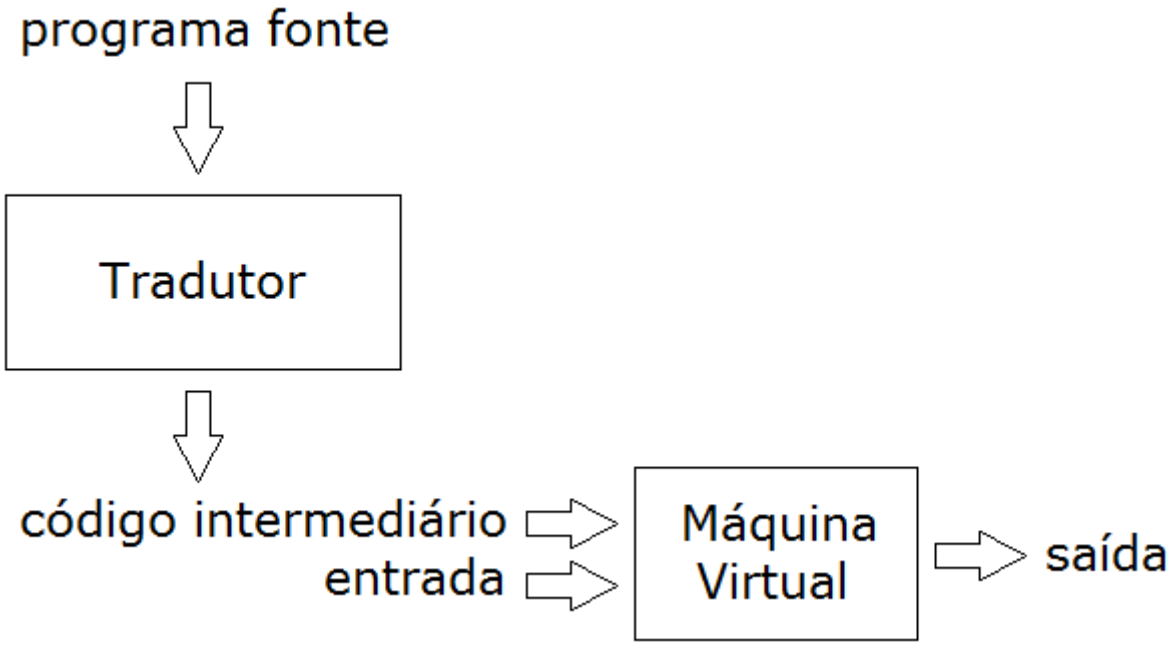
* Os processadores da linguagem Java combinam compilação e interpretação.

* Um programa fonte em Java pode ser primeiro compilado para uma forma intermediária, chamada bytecodes. Os bytecodes (ou códigos de bytes) são então interpretados por uma máquina virtual.

* Como um benefício dessa combinação, os bytecodes compilados em uma máquina podem ser interpretados em outra máquina, talvez por meio de uma rede.

Tradutores, Compiladores e Interpretadores

* Um compilador híbrido, Fig. 4:



Tradutores, Compiladores e Interpretadores

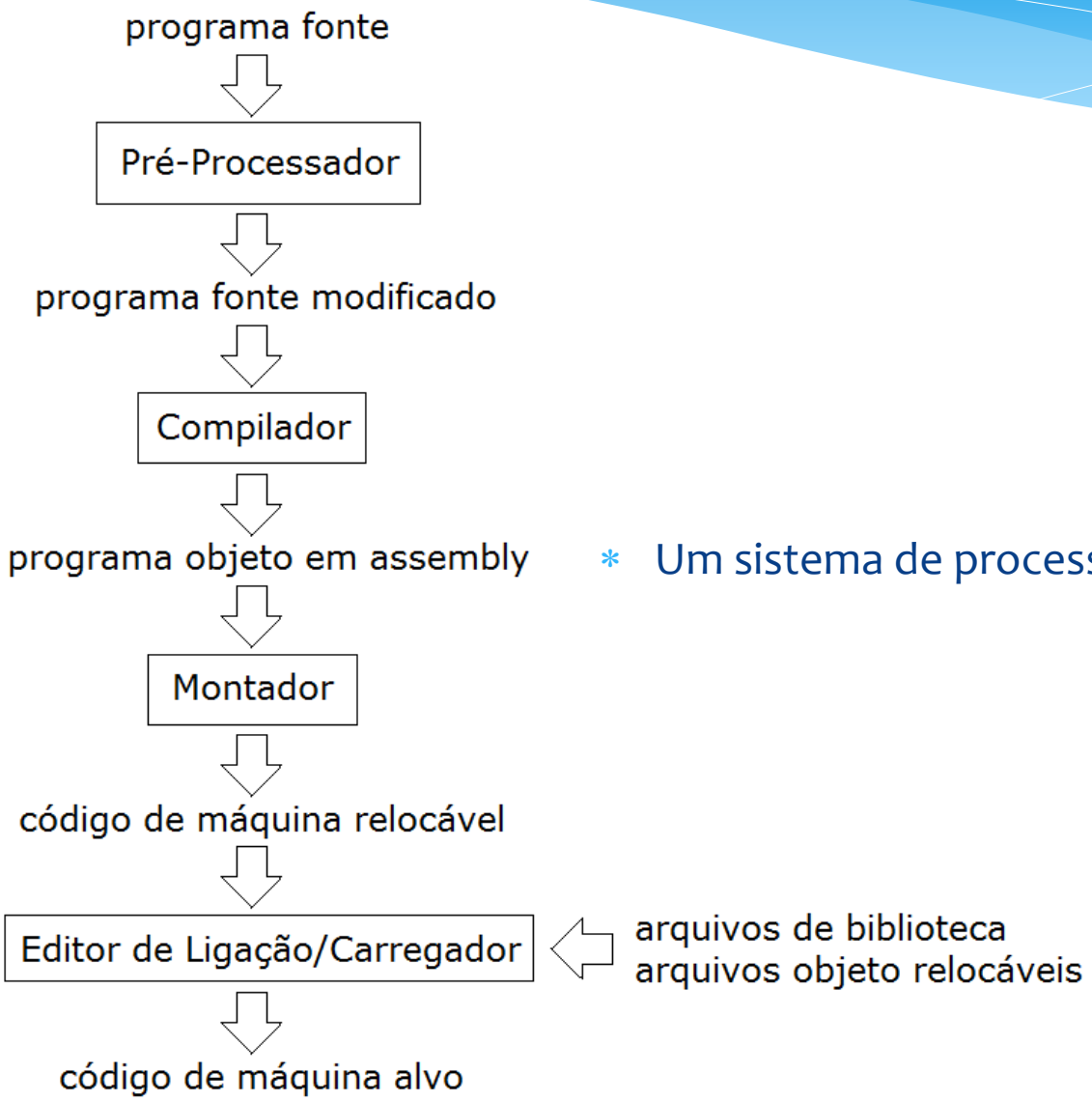
* A fim de conseguir um processamento mais rápido das entradas para as saídas, alguns compiladores Java, chamados compiladores *just-in-time*, traduzem os bytecodes para uma dada linguagem de máquina imediatamente antes de executarem o programa intermediário para processar a entrada.

Tradutores, Compiladores e Interpretadores

* Além de um compilador, vários outros programas podem ser necessários para a criação de um programa objeto executável.

* Um programa fonte pode ser subdividido em módulos armazenados em arquivos separados.

* A tarefa de coletar o programa fonte às vezes é confiada a um programa separado, chamado pré-processador.



* Um sistema de processamento de linguagem, Fig. 5.

Tradutores, Compiladores e Interpretadores

* O compilador recebe na entrada o programa fonte modificado e pode produzir como saída um programa em uma linguagem simbólica, conhecida como *assembly*, considerada mais fácil de ser gerada como saída e mais fácil de depurar.

* A linguagem simbólica é então processada por um programa chamado montador (*assembler*), que produz código de máquina relocável (código em que as referências externas e endereços de memória não estão decididos, ou seja, são endereços simbólicos) como sua saída.

Tradutores, Compiladores e Interpretadores

* Programas grandes normalmente são compilados em partes, de modo que o código de máquina relocável pode ter de ser ligado a outros arquivos objeto relocáveis e a arquivos de biblioteca para formar o código que realmente é executado na máquina.

* O editor de ligação (*linker*) resolve os endereços de memória externos, onde o código em um arquivo pode referir-se a uma localização em outro arquivo.

* O carregador (*loader*) reúne então todos os arquivos objeto executáveis na memória para a execução.

Exercícios

1) Qual é a diferença entre um compilador e um interpretador?

2) Quais são as vantagens de (a) um compilador em relação a um interpretador e (b) um interpretador em relação a um compilador?

3) Que vantagens existem em um sistema de processamento de linguagem no qual o compilador produz linguagem simbólica em vez de linguagem de máquina?

4) Um compilador que traduz uma linguagem de alto nível para outra linguagem de alto nível é chamado tradutor de fonte para fonte. Que vantagens existem em usar C como linguagem objeto para um compilador?

5) Descreva algumas das tarefas que um programa montador precisa realizar.

Estrutura de um compilador
Compilação

* Até este ponto, tratamos um compilador como uma caixa-preta que mapeia um programa fonte para um programa objeto semanticamente equivalente.

* Se abrirmos um pouco essa caixa, veremos que existem duas partes nesse mapeamento: *análise* e *síntese*.

Estrutura de um compilador
Passos

* A parte de *análise* subdivide o programa fonte em partes constituintes e impõe uma estrutura gramatical sobre elas. Depois, usa essa estrutura para criar uma representação intermediária do programa fonte.

* Se a parte de análise detectar que o programa fonte está sintaticamente mal formado ou semanticamente incorreto, então ele precisa oferecer mensagens esclarecedoras, de modo que o usuário possa tomar a ação corretiva.

Estrutura de um compilador
Passos

* A parte de análise também coleta informações sobre o programa fonte e as armazena em uma estrutura de dados chamada ***tabela de símbolos***, que é passada adiante junto com a representação intermediária para a parte de síntese.

Estrutura de um compilador
Passos

* A parte de *síntese* constrói o programa objeto desejado a partir da representação intermediária e das informações na tabela de símbolos.

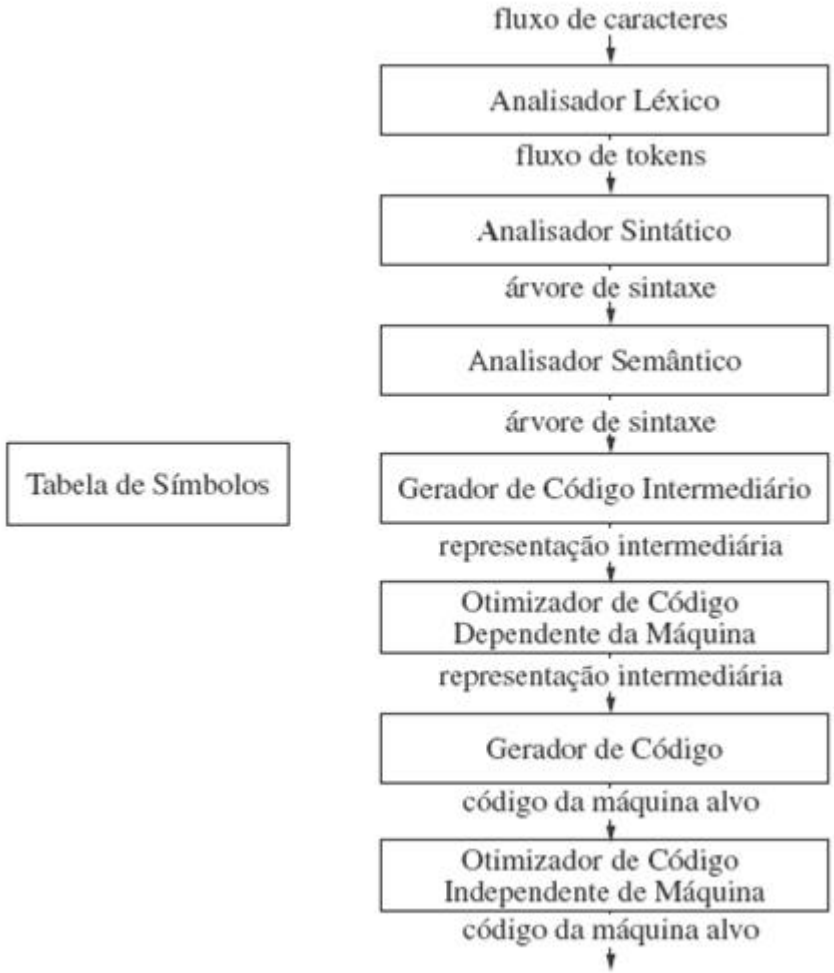
* A parte de análise normalmente é chamada de ***front-end*** do compilador; a parte de síntese é o ***back-end***.

Estrutura de um compilador
Fases da compilação

* Se examinarmos o processo de compilação detalhadamente, veremos que ele é desenvolvido como uma sequencia de ***fases***, cada uma transformando uma representação de programa fonte em outra.

* A figura a seguir exibe a decomposição típica de um compilador em fases.

Estrutura de um compilador



* Fases de um compilador, Fig. 6.

Estrutura de um compilador
Fases da compilação

* Na prática, várias fases podem ser agrupadas, e as representações intermediárias entre essas fases agrupadas não precisam ser construídas explicitamente.

* A tabela de símbolos, responsável pelo armazenamento das informações sobre todo o programa fonte, é utilizada por todas as fases do compilador.

Estrutura de um compilador
Fases da compilação

* Alguns compiladores possuem uma fase de otimização independente de máquina entre o *front-end* e o *back-end*.

* A finalidade dessa fase de otimização é realizar transformações na representação intermediária, de modo que o *back-end* possa produzir um programa objeto melhor do que teria produzido a partir de uma representação intermediária não otimizada.

* Como esta etapa é **opcional**, uma das duas fases de otimização mostradas na figura anterior pode ser omitida.

Estrutura de um compilador
Fases da compilação

* A fase de ***análise*** de um compilador subdivide um programa fonte em partes constituintes e produz uma representação interna para ele, chamada código intermediário.

* A fase de ***síntese*** traduz o código intermediário para o programa objeto.

Estrutura de um compilador
Fases da compilação

* A análise é organizada em torno da "sintaxe" da linguagem a ser compilada. A ***sintaxe*** de um linguagem de programação descreve a forma apropriada dos seus programas, enquanto a ***semântica*** da linguagem define o que seus programas significam; ou seja, o que cada programa faz quando é executado.

* Para especificar a sintaxe, veremos uma notação bastante utilizada, chamada ***gramáticas livres de contexto***.

Estrutura de um compilador
Fases da compilação

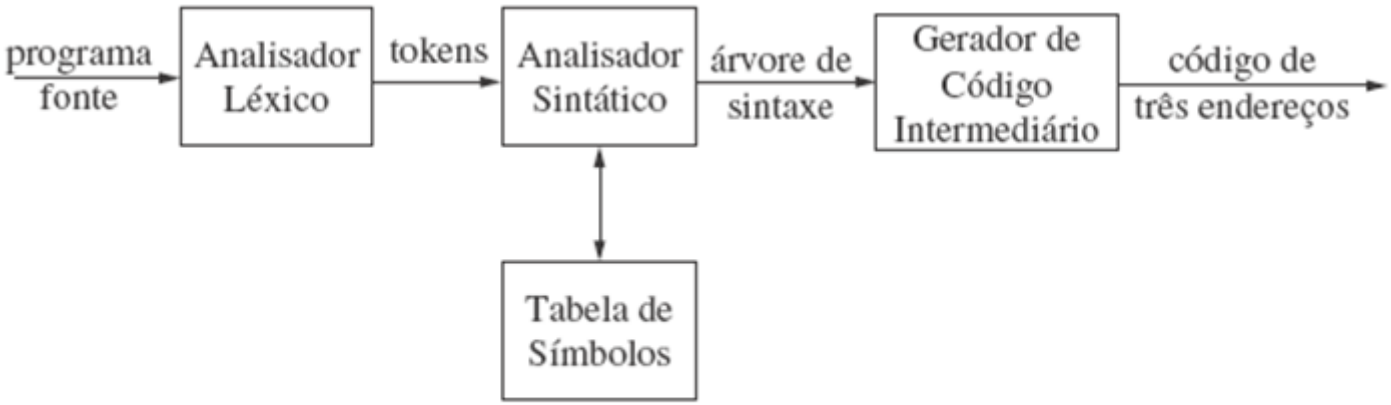
* Com as notações atualmente disponíveis, é muito mais difícil descrever a semântica de uma linguagem do que a sintaxe.

* Portanto, para especificar a semântica, usaremos descrições informais e exemplos sugestivos.

* Além de especificar a sintaxe de uma linguagem, uma gramática livre de contexto pode ser usada para auxiliar na tradução dos programas.

Estrutura de um compilador
Fases da compilação

* **Modelo de um *front-end* de compilador**, Fig. 7.



Um modelo de *front-end* de compiladores.

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
* **Análise Sintática**
* **Análise Semântica**
* **Geração de Código Intermediário**
* **Otimização de Código**
* **Geração de Código**

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
* **Análise Sintática**
* **Análise Semântica**
* **Geração de Código Intermediário**
* **Otimização de Código**
* **Geração de Código**

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
  * A primeira fase de um compilador é chamada *análise léxica* ou *leitura* (*scanning*).
  * O analisador léxico lê o fluxo de caracteres que compõem o programa fonte e os agrupa em sequencias significativas, chamadas *lexemas*. Para cada lexema, o analisador léxico produz como saída um token no formato:

(*nome-token*, *valor-atributo*)

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
  * Em seguida o analisador léxico passa o token para a fase subsequente, a análise sintática.

  * Um analisador léxico permite que um tradutor trate as construções de múltiplos caracteres como identificadores, que são escritos como sequências de caracteres, mas são tratados como unidades chamadas tokens durante a análise sintática.

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
  * Em um token, o primeiro componente, *nome-token*, é um símbolo abstrato que é usado durante a análise sintática, e o segundo componente, *valor-atributo*, aponta para uma entrada na tabela de símbolos referente a esse token.
  * A informação da entrada da tabela de símbolos é necessária para a análise semântica e para a geração de código.

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
  * Exemplo: Suponha que um programa fonte contenha o comando de atribuição:

```
variavel1 = variavel2 + variavel3 * 60
```

Estrutura de um compilador

* **Análise Léxica**
  * Os caracteres nessa atribuição poderiam ser agrupados nos seguintes lexemas e mapeados para os seguintes tokens passados ao analisador sintático:

  * 1. **variavel1** é um lexema mapeado em um token <**id**, 1>, onde **id** é um símbolo abstrato que significa *identificador* e **1** aponta para a entrada da tabela de símbolos onde se encontra **variavel1**. A entrada da tabela de símbolos para um identificador mantém informações sobre o identificador, como seu nome e tipo.

  * 2. O símbolo de atribuição **=** é um lexema mapeado para o token **<=>**. Como esse token não precisa de um valor de atributo, omitimos o segundo componente. Poderíamos ter usado qualquer símbolo abstrato, como **atribuir** para o nome do token, mas, por conveniência de notação, escolhemos usar o próprio lexema como nome do símbolo abstrato.

```
variavel1 = variavel2 + variavel3 * 60
```

Estrutura de um compilador

* **Análise Léxica**
  * 3. **variavel2** é um lexema mapeado em um token <**id**, 2>, onde **2** aponta para a entrada da tabela de símbolos onde se encontra **variavel2**.
  * 4. **+** é um lexema mapeado para o token **<+>**.
  * 5. **variavel3** é um lexema mapeado em um token <**id**, 3>, onde o valor **3** aponta para a entrada da tabela de símbolos onde se encontra **variavel3** .
  * 6. ***** é um lexema mapeado para o token **<*>**.
  * 7. **60** é um lexema mapeado para o token **<60>**.

variavel1 = variavel2 + variavel3 ∗ 60

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
  * Os espaços que separam os lexemas são descartados pelo analisador léxico.

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
  * Comando de atribuição antes da análise léxica:

    variavel1 = variavel2 + variavel3 * 60

  * Representação do comando de atribuição APÓS a análise léxica como uma sequencia de tokens:

    <**id**, 1> <=> <**id**, 2> <+> <**id**, 3> <*> <60>

  * Nessa representação, os nomes de token =, + e * são símbolos abstratos para os operadores de atribuição, adição e multiplicação, respectivamente.

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
* **Análise Sintática**
* **Análise Semântica**
* **Geração de Código Intermediário**
* **Otimização de Código**
* **Geração de Código**

Estrutura de um compilador
Fases da compilação

* **Análise Sintática**
  * A segunda fase do compilador é a *análise sintática*.
  * O analisador sintático utiliza os primeiros componentes dos tokens produzidos pelo analisador léxico para criar uma representação intermediária tipo árvore, que mostra a estrutura gramatical da sequencia de tokens.

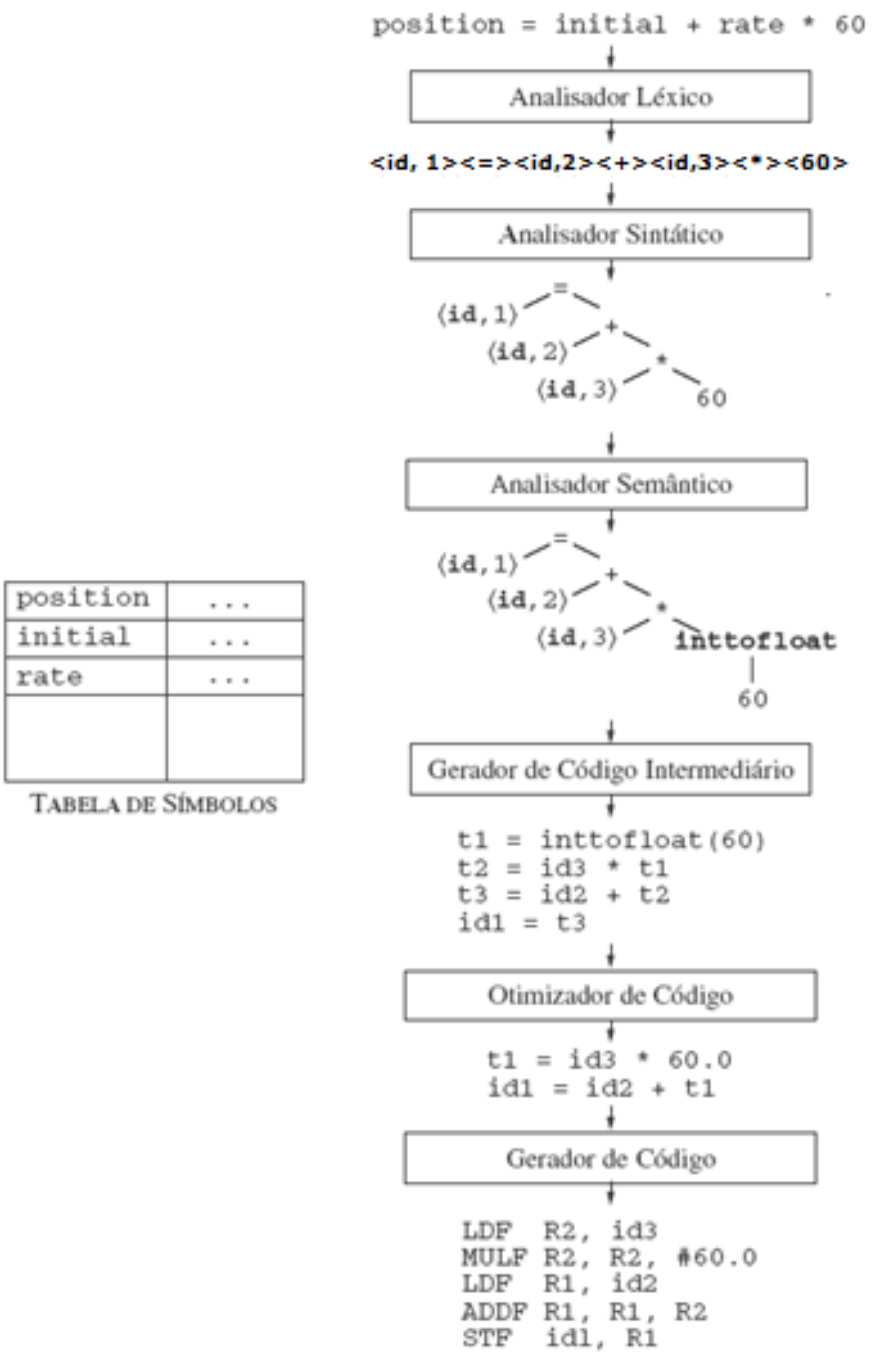
Estrutura de um compilador
Fases da compilação

* **Análise Sintática**
  * Uma representação típica é uma *árvore de sintaxe* em que cada nó interior representa uma operação, e os filhos do nó representam os argumentos da operação.

  * Uma árvore de sintaxe para o fluxo de tokens,

    <**id**, 1> <=> <**id**, 2> <+> <**id**, 3> <*> <60>

    aparece como saída do analisador sintático da figura a seguir:



position	...
initial	...
rate	...

TABELA DE SÍMBOLOS

* Tradução de uma instrução de atribuição, Fig. 8.

Estrutura de um compilador
Fases da compilação

* **Análise Sintática**
  * Essa árvore mostra a ordem em que as operações do comando de atribuição:

    ```
    variavel1 = variavel2 + variavel3 * 60
    ```

    deve ser realizada.

Estrutura de um compilador
Fases da compilação

* **Análise Sintática**
  * A árvore possui um nó interior rotulado com *, com <**id**, 3> como seu filho da esquerda e o inteiro 60 como seu filho da direita. O nó <**id**, 3> representa o identificador variavel3.
  * O nó rotulado com * torna explícito que devemos primeiro multiplicar o valor de variavel3 por 60.
  * O nó rotulado com + indica que devemos somar o resultado dessa multiplicação com o valor da variavel2.

Estrutura de um compilador
Fases da compilação

* **Análise Sintática**
  * A raiz da árvore, rotulada com =, indica que devemos armazenar o resultado dessa adição em uma localização associada ao identificador variavel1.

  * Essa ordem das operações é consistente com as convenções normais da aritmética, que nos dizem que a multiplicação tem maior precedência que a adição, e por isso a multiplicação deve ser realizada antes da adição.

Estrutura de um compilador
Fases da compilação

* As fases subsequentes do compilador utilizam a **estrutura gramatical** para auxiliar na análise do programa fonte e para gerar o programa objeto.

* Na AULA04_COMPILADORES_ANALISE_SINTATICA, usaremos as **gramáticas livres de contexto** para especificar a estrutura gramatical das linguagens de programação e discutiremos os algoritmos para a construção automática de analisadores sintáticos eficientes, a partir de certas classes de gramáticas.

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
* **Análise Sintática**
* **Análise Semântica**
* **Geração de Código Intermediário**
* **Otimização de Código**
* **Geração de Código**

Estrutura de um compilador
Fases da compilação

* **Análise Semântica**
  * O *analisador semântico* utiliza a árvore de sintaxe e as informações na tabela de símbolos para verificar a consistência semântica do programa fonte com a definição da linguagem.
  * Ele também reúne informações sobre os tipos e as salva na árvore de sintaxe ou na tabela de símbolos, para uso subsequente durante a geração de código intermediário.

Estrutura de um compilador
Fases da compilação

* **Análise Semântica**
  * Uma parte importante da análise semântica é a *verificação de tipo*, em que o compilador verifica se cada operador possui operandos compatíveis.

* Exemplo: x = 2 + “teste”;

Estrutura de um compilador
Fases da compilação

* **Análise Semântica**
  * Outro exemplo: Muitas linguagens de programação exigem que um índice de arranjo seja um inteiro, portanto o compilador precisa informar um erro de tipo se um número de ponto flutuante for usado para indexar um arranjo.

Estrutura de um compilador
Fases da compilação

* **Análise Semântica**
  * A especificação da linguagem pode permitir algumas conversões de tipos chamadas de ***coerções***.

  * Por exemplo, um operador aritmético binário pode ser aplicado um par de inteiros ou a um par de números de ponto flutuante. Se o operador for aplicado **a um número de ponto flutuante e a um inteiro**, o compilador pode converter ou coagir o inteiro para um número de ponto flutuante.

Estrutura de um compilador
Fases da compilação

* **Análise Semântica**
  * Como neste exemplo, Fig. 9:

```
float x;
float y = 2.0;
int z = 3;
x = y + z;

// z é coagido para float.
// z = 3.0 e x = 5.0
```

Estrutura de um compilador
Fases da compilação

* **Análise Semântica**
  * Essa coerção aparece na Fig. 8 “Tradução de uma instrução de atribuição”.
  * Suponha que **variavel1**, **variavel2** e **variavel3** tenham sido declaradas como números de ponto flutuante, e que o lexema **60** tenha a forma de um inteiro.
  * O verificador de tipos no analisador semântico da figura descobre que o operador ***** é aplicado a um número de ponto flutuante **variavel3** e a um inteiro **60**.

Estrutura de um compilador
Fases da compilação

* **Análise Semântica**
  * Nesse caso, o inteiro pode ser convertido em um número de ponto flutuante. Na Fig. 8, observe que a saída do analisador semântico tem um nó extra para o operador **inttofloat**, o qual converte explicitamente seu argumento inteiro em um número de ponto flutuante.

  * A verificação de tipo e a análise semântica são discutidas nas aulas: AULA05 e AULA06.

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
* **Análise Sintática**
* **Análise Semântica**
* **Geração de Código Intermediário**
* **Otimização de Código**
* **Geração de Código**

Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * No processo de traduzir um programa fonte para um código objeto, um compilador pode produzir uma ou mais representações intermediárias, as quais podem ter diversas formas.

  * As árvores de sintaxe denotam uma forma de representação intermediária; elas normalmente são usadas durante as análises sintática e semântica.

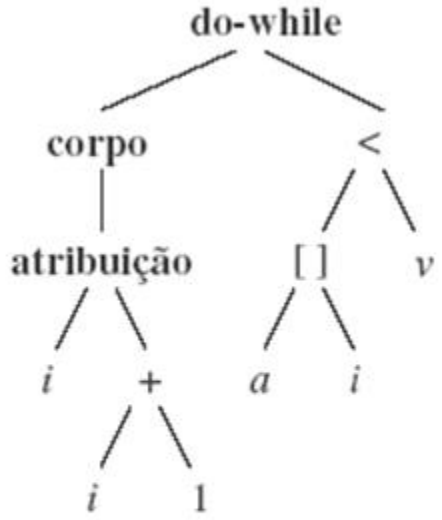
Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * As árvores sintáticas abstratas, ou simplesmente árvores sintáticas, representam a estrutura sintática hierárquica do programa fonte.

  * No modelo da Fig. 7, o analisador sintático produz uma árvore sintática, que é depois traduzida em um código de três endereços. Alguns compiladores combinam a análise sintática e a geração de código intermediário em um único módulo.

Estrutura de um compilador

* **Geração de Código Intermediário**
  * A Fig. 10, abaixo, ilustra duas formas de código intermediário.



(a)

```
1:  i=i+1
2:  t1 = a[i]
3:  if t1 < v goto 1
```

(b)

Código intermediário para "`do i = i + 1; while ( a[i] < v);`".

Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * A raiz da árvore sintática abstrata da Fig. 10(a) representa um *loop* do-while completo. O filho à esquerda da raiz representa o corpo do *loop*, que consiste apenas na atribuição $i = i + 1;$. O filho à direita da raiz representa a condição $a[i] < v$.

Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * Outra forma de representação intermediária muito comum, mostrada na Fig. 10(b), é uma sequência de instruções de "três endereços".

  * A razão para o nome "três endereços" dessa forma de código intermediário advém de instruções no formato $x = y$ **op** z, onde **op** é um operador binário, y e z são os endereços para os operandos, e x é o endereço para o resultado da operação.

Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * Uma instrução de três endereços executa no máximo uma operação, normalmente um cálculo, uma comparação ou um desvio.

Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * Depois das análises sintática e semântica do programa fonte, muitos compiladores geram uma representação intermediária explícita de baixo nível ou do tipo linguagem de máquina, que podemos imaginar como um programa para uma máquina abstrata.

  * Essa representação intermediária deve ter duas propriedades importantes: ser facilmente produzida e ser facilmente traduzida para a máquina alvo.

Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * Na AULA06, iremos considerar a fundo a forma intermediária chamada de ***código de três endereços***, que, como vimos, consiste em uma sequência de instruções do tipo assembler com três operandos por instrução.

  * Cada operando pode atuar como um registrador.
    * Um registrador é um local interno à CPU, onde os dados que foram buscados na memória são armazenados. O registrador é um circuito lógico que tem a finalidade de reter a curto prazo um conjunto de bits.

Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * A saída do gerador de código intermediária na Fig. 8 consiste em uma sequência de instruções ou código de três endereços:

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

Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * Vários pontos precisam ser observados em relação aos códigos de três endereços.

  * Primeiro, cada instrução de atribuição de três endereços possui no máximo um operador do lado direito. Assim, essas instruções determinam a ordem em que as operações devem ser realizadas; a multiplicação precede a adição no programa fonte como visto anteriormente.

Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * Segundo, o compilador precisa gerar um nome temporário para guardar o valor computado por uma instrução de três endereços.

  * Terceiro, algumas "instruções de três endereços", como a primeira e última na sequência, possuem menos de três operandos:

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

Estrutura de um compilador
Fases da compilação

* **Geração de Código Intermediário**
  * Na AULA06 serão apresentadas as principais representações intermediárias usadas nos compiladores.

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
* **Análise Sintática**
* **Análise Semântica**
* **Geração de Código Intermediário**
* **Otimização de Código**
* **Geração de Código**

Estrutura de um compilador
Fases da compilação

* **Otimização de Código**
  * A fase de otimização de código independente das arquiteturas de máquina faz algumas transformações no código intermediário com o objetivo de produzir um código objeto melhor.
  * Normalmente, melhor significa mais rápido, mas outros objetivos podem ser desejados, como um código menor ou um código objeto que consuma menos energia.

Estrutura de um compilador
Fases da compilação

* **Otimização de Código**
  * Por exemplo, um algoritmo direto gera o código intermediário visto anteriormente, usando instrução para cada um dos operandos da representação de árvore produzida pelo analisador semântico.

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

Estrutura de um compilador
Fases da compilação

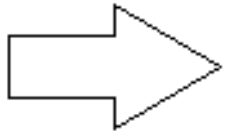
* **Otimização de Código**
  * Uma boa estratégia para gerar um código aberto é usar um algoritmo simples de geração de código intermediário seguido de otimizações.

  * Nesta abordagem, o otimizador pode deduzir que a conversão do valor inteiro 60 para ponto flutuante pode ser feita de uma vez pode todas durante a compilação, de modo que a operação **inttofloat** pode ser eliminada do código substituindo-se o inteiro 60 pelo número de ponto flutuante 60.0.

Estrutura de um compilador
Fases da compilação

* **Otimização de Código**
  * Além do mais, t3 é usado apenas uma vez na atribuição de seu valor para id1, portanto o otimizador pode eliminá-lo também transformando o código intermediário em uma sequência de código menor:

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



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

Estrutura de um compilador
Fases da compilação

* **Otimização de Código**
  * O número de otimizações de código realizadas por diferentes compiladores varia muito.
  * Aqueles que exploram ao máximo as oportunidades de otimizações são chamados "***compiladores otimizadores***".
  * Quanto mais otimizações, mais tempo é gasto nessa fase. Mas existem otimizações simples que melhoram significativamente o tempo de execução do programa objeto sem atrasar muito a compilação.

Estrutura de um compilador
Fases da compilação

* **Otimização de Código**
  * As aulas AULA07 e AULA08 discutirão em detalhes as otimizações independentes e dependentes da arquitetura de máquina.

Estrutura de um compilador
Fases da compilação

* **Análise Léxica**
* **Análise Sintática**
* **Análise Semântica**
* **Geração de Código Intermediário**
* **Otimização de Código**
* **Geração de Código**

Estrutura de um compilador
Fases da compilação

* **Geração de Código**
  * O gerador de código recebe como entrada uma representação intermediária do programa fonte e o mapeia em uma linguagem objeto.

  * Se a linguagem objeto for código de máquina de alguma arquitetura, devem-se selecionar os registradores ou localizações de memória para cada uma das variáveis usadas pelo programa. Depois, os códigos intermediários são traduzidos em sequências de instruções de máquina que realizam a mesma tarefa.

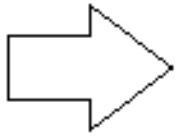
Estrutura de um compilador
Fases da compilação

* **Geração de Código**
  * Um aspecto crítico da geração de código está relacionado à cuidadosa atribuição dos registradores às variáveis do programa.

Estrutura de um compilador
Fases da compilação

* **Geração de Código**
  * Por exemplo, usando os registradores R1 e R2, o código intermediário visto poderia ser traduzido para o código 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
```

Estrutura de um compilador

* **Geração de Código**
  * O primeiro operando de cada uma das instruções especifica um destino.
  * O **F** em cada uma das instruções nos diz que ela manipula números de ponto flutuante.
  * O código carrega o conteúdo do endereço id3 no registrador R2, depois o multiplica pela constante de ponto flutuante 60.0.
  * O **#** significa que o valor 60.0 deve ser tratado como uma constante imediata.

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

Legenda:
F = **F**loat
LD = Loa**D** (Carrega)
MUL = Multiplica
ADD = Adiciona
ST = Store (Armazena)

Estrutura de um compilador

* **Geração de Código**
  * A terceira instrução move id2 para o registrador R1, e a quarta o soma com o valor previamente calculado no registrador R2.
  * Finalmente, o valor no registrador R1 é armazenado no endereço de id1, portanto o código mostrado implementa corretamente comando de atribuição variavel1 = variavel2 + variavel3 * 60. A AULA07 irá abordar a geração de código.

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

Legenda:
F = **F**loat
LD = Loa**D** (Carrega)
MUL = Multiplica
ADD = Adiciona
ST = Store (Armazena)

Estrutura de um compilador

* **Geração de Código**
  * Comentários:

```
LDF     R2,     id3             // carrega o conteúdo do endereço id3 no registrador R2
MULF    R2,     R2,  #60.0      // multiplica pela constante de ponto flutuante 60.0
LDF     R1,     id2             // carrega o conteúdo do endereço id2 no registrador R1
ADDF    R1,     R1,  R2         // soma o valor contido no registrador R1 com o valor
                                // previamente calculado no registrador R2
STF     id1,    R1              // o valor no registrador R1 é armazenado no endereço id1
```

Estrutura de um compilador
Fases da compilação

* **Geração de Código**
  * Esta discussão sobre geração de código ignorou a importante questão relativa à alocação de espaço na memória para os identificadores do programa fonte.
  * A organização de memória em tempo de execução depende da linguagem sendo compilada.
  * Decisões sobre a alocação de espaço podem ser tomadas em dois momentos: durante a geração de código intermediário ou durante a geração de código.

Estrutura de um compilador

* **Gerenciamento da Tabela de Símbolos**
  * Uma função essencial de um compilador é registrar os nomes de variáveis usados no programa fonte e coletar informações sobre os diversos atributos de cada nome.

  * Esses atributos podem prover informações sobre o espaço de memória alocado para um nome, seu tipo, seu escopo, ou seja, onde seu valor pode ser usado no programa, e, no caso de nomes de procedimento, informações sobre a quantidade e os tipos de seus argumentos, o tipo retornado e o método de passagem de cada argumento, por exemplo, por valor ou por referência.

Estrutura de um compilador

* **Gerenciamento da Tabela de Símbolos**
  * A tabela de símbolos é uma estrutura de dados contendo um registro para cada nome de variável, com campos para os atributos do nome.
  * A estrutura de dados deve ser projetada para permitir que o compilador encontre rapidamente o registro para cada nome e armazene ou recupere dados desse registro também rapidamente.

Estrutura de um compilador

* **O Agrupamento de Fases em Passos**
  * A discussão sobre as fases de um compilador diz respeito à sua organização lógica.
  * Em uma implementação, as atividades de várias fases podem ser agrupadas em um ***passo*** que lê um arquivo de entrada e o escreve em um arquivo de saída.
  * Por exemplo, as fases de análise léxica, análise sintática, análise semântica e geração de código intermediário do *front-end* poderiam ser agrupadas em um passo.

Estrutura de um compilador

* **O Agrupamento de Fases em Passos**
  * A fase de otimização do código poderia ser um passo opcional.
  * Depois, poderia haver um passo para o *back-end*, consistindo na geração de código para determinada máquina alvo.
  * Algumas famílias de compiladores foram criadas em torno de representações intermediárias cuidadosamente projetadas, que permitem que o *front-end* para determinada linguagem fonte tenha uma interface com o *back-end* para uma arquitetura específica na máquina alvo.

Estrutura de um compilador

* **O Agrupamento de Fases em Passos**
  * Com essas famílias, podemos produzir compiladores de diferentes linguagens fonte para uma determinada máquina alvo, combinando diferentes *front-ends* com um único *back-end* para essa máquina alvo.
  * Da mesma forma, podemos produzir compiladores para diferentes máquinas alvo, combinando um único *front-end* com *back-ends* para diferentes máquinas alvo.

Estrutura de um compilador

* **Ferramentas para Construção de Compilador**
  * O projetista de compilador, como qualquer desenvolvedor de software, pode tirar proveito dos diversos ambientes de desenvolvimento de software modernos contendo ferramentas como editores de texto, depuradores, gerenciadores de versão, profilers, ferramentas de testes e assim por diante.

  * Além dessas ferramentas gerais de desenvolvimento de software, outras ferramentas mais especializadas foram desenvolvidas para auxiliar na programação de diversas fases de um compilador.

Estrutura de um compilador

* **Ferramentas para Construção de Compilador**
  * Essas ferramentas utilizam linguagens especializadas para especificar e implementar componentes específicos, e muitas usam algoritmos bastante sofisticados.

  * As ferramentas mais bem-sucedidas são aquelas que ocultam os detalhes de seus algoritmos e produzem componentes que podem ser facilmente integrados ao restante do compilador.

Estrutura de um compilador

* **Ferramentas para Construção de Compilador**

  Algumas das ferramentas mais utilizadas na construção de compiladores:

  1) ***Geradores de analisadores léxicos,*** que produzem analisadores léxicos a partir de uma descrição dos tokens de uma linguagem em forma de expressão regular.

  2) ***Geradores de analisadores sintáticos,*** que produzem automaticamente reconhecedores sintáticos a partir de uma descrição gramatical de uma linguagem de programação.

  3) ***Mecanismos de tradução dirigida por sintaxe,*** que produzem coleções de rotinas para percorrer uma árvore de derivação e gerar código intermediário.

  Para 1 e 2 visite: *http://dinosaur.compilertools.net/*

Estrutura de um compilador

* **Ferramentas para Construção de Compilador**

  Algumas das ferramentas mais utilizadas na construção de compiladores:

  4) ***Geradores de gerador de código***, que produzem um gerador de código a partir de uma coleção de regras para traduzir cada operação da linguagem intermediária na linguagem de máquina para uma máquina alvo.

  5) ***Mecanismos de análise de fluxo de dados***, que facilitam a coleta de informações sobre como os valores são transmitidos de parte de um programa para cada uma das outras partes. A análise de fluxo de dados é uma ferramenta essencial para a otimização de código.

  6) ***Conjuntos de ferramentas para a construção de compiladores***, que oferecem um conjunto integrado de rotinas para a construção das diversas fases de um compilador.

Evolução das Linguagens de Programação

* **Evolução das Linguagens de Programação**
  * Os primeiros computadores eletrônicos apareceram na década de 1940 e eram programados em linguagem de máquina por sequências de 0s e 1s que diziam explicitamente ao computador quais operações deveriam ser executadas e em que ordem.

  * As operações em si eram de muito baixo nível: mover dados de um local para outro, somar o conteúdo de dois registradores, comparar valores e assim por diante.

Evolução das Linguagens de Programação

* **Evolução das Linguagens de Programação**
  * Nem é preciso dizer que esse tipo de programação era lento, cansativo e passível de erros.
  * E, uma vez escritos, tais programas eram difíceis de entender e modificar.

Evolução das Linguagens de Programação

* **Mudança para Linguagens de Alto Nível**
  * O primeiro passo para tornar as linguagens de programação mais inteligíveis às pessoas se deu no início da década de 1950 com o desenvolvimento das linguagens simbólicas ou assembly.

  * Inicialmente, as instruções em uma linguagem simbólica eram apenas representações mnemônicas das instruções de máquina.

Evolução das Linguagens de Programação

* **Mudança para Linguagens de Alto Nível**
  * Mais tarde, foram acrescentadas instruções de *macro* às linguagens simbólicas, para que um programador pudesse definir abreviaturas parametrizadas para sequências de instruções de máquina usadas com frequência.

  * Um passo importante em direção às linguagens de alto nível foi dado na segunda metade da década de 1950, com o desenvolvimento do Fortran para computação científica, do *Cobol* para o processamento de dados comercial, e do *Lisp* para a computação simbólica.

Evolução das Linguagens de Programação

* **Mudança para Linguagens de Alto Nível**
  * A filosofia por trás dessas linguagens era criar construções de alto nível a partir das quais os programadores poderiam escrever com mais facilidade cálculos numéricos, aplicações comerciais e programas simbólicos.

  * Essas linguagens tiveram tanto sucesso que continuam sendo utilizadas até hoje. Nas décadas seguintes, foram projetadas muitas linguagens com recursos inovadores para ajudar a tornar a programação mais fácil, mais natural e mais poderosa.

* **Mudança para Linguagens de Alto Nível**
  * Atualmente, existem milhares de linguagens de programação. Elas podem ser classificadas de diversas maneiras.
  * Uma classificação diz respeito à sua geração, linguagens de:
    * ***Primeira geração*** são as linguagens de máquina;
    * ***Segunda geração*** são as linguagens simbólicas ou de montagem, também conhecidas como assembly;
    * ***Terceira geração*** são as linguagens de alto nível, procedimentais, como Fortran, Cobol, Lisp, C, C++, C# e Java;
    * ***Quarta geração*** são criadas para aplicações específicas, como a linguagem NOMAD para geração de relatórios, SQL para consultas a banco de dados e Postscript para formação de textos;
    * ***Quinta geração*** são as linguagens baseadas em lógica com restrição, como Prolog e OPS5.

Evolução das Linguagens de Programação

* **Mudança para Linguagens de Alto Nível**
  * Outra classificação utilizada denomina ***imperativas*** as linguagens em que um programa especifica ***como*** uma computação deve ser feito.
  * Linguagens como C, C++, C# e Java são linguagens imperativas.
  * Nas linguagens imperativas, existe a noção de estado do programa e mudança do estado provocadas pela execução das instruções.

Evolução das Linguagens de Programação

* **Mudança para Linguagens de Alto Nível**
  * Linguagens funcionais como ML e Haskell, e linguagens de lógica com restrição, como Prolog, normalmente são consideradas linguagens ***declarativas***.

Evolução das Linguagens de Programação

* **Mudança para Linguagens de Alto Nível**
  * O termo ***linguagem de von Neumann*** é aplicado a linguagens de programação cujo modelo computacional se baseia na arquitetura de computador *von Neumann*.
  * Muitas das linguagens de hoje, como Fortran e C, são linguagens de *von Neumann*.

Evolução das Linguagens de Programação

* **Mudança para Linguagens de Alto Nível**
  * John von Neumann
    * 1903 – 1957
  * Um dos mais importantes matemáticos do século XX.
  * Conhecido na computação pela Arquitetura de von Neumann e pela construção do ENIAC.

Evolução das Linguagens de Programação

* **Mudança para Linguagens de Alto Nível**
  * Uma ***linguagem orientada por objeto*** é aquela que admite a programação orientada por objeto, um estilo de programação no qual um programa consiste em uma coleção de objetos que interagem uns com os outros.
  * Simula 67 e Smalltalk são as principais linguagens orientadas por objeto mais antigas. Linguagens como C++, C#, Java e Ruby são linguagens orientadas por objeto mais recentes.

Evolução das Linguagens de Programação

* **Mudança para Linguagens de Alto Nível**
  * ***Linguagens de scripting*** são linguagens interpretadas com operadores de alto nível projetados para "juntar" computações.
  * Essas computações eram originalmente chamadas de ***scripts***. Awk, JavaScript, Perl, Python, Ruby e Tcl são exemplos populares de linguagens de scripting.
  * Programas escritos em linguagens de scripting normalmente são muitos menores do que os programas equivalentes escritos em linguagens como C.

Evolução das Linguagens de Programação

* **Impactos nos Compiladores**
  * Como o projeto de linguagens de programação e os compiladores estão intimamente relacionados, os avanços nas linguagens de programação impõem novas demandas sobre os projetistas de compiladores.

  * Eles têm de criar algoritmos e representações para traduzir e dar suporte aos novos recursos da linguagens.

Evolução das Linguagens de Programação

* **Impactos nos Compiladores**
  * Desde a década de 1940, as arquiteturas de computadores também têm evoluído.
  * Os desenvolvedores de compiladores tiveram não apenas de acompanhar os novos recursos das linguagens, mas também de projetar algoritmos de tradução que tirassem o máximo de proveito das novas **capacidades do hardware.**

Evolução das Linguagens de Programação

* **Impactos nos Compiladores**
  * Os compiladores podem ajudar a difundir o uso de linguagens de alto nível, minimizando o custo adicional da execução dos programas escritos nessas linguagens.
  * Os compiladores também são responsáveis por efetivar o uso das arquiteturas de computador de alto desempenho nas aplicações dos usuários.

Evolução das Linguagens de Programação

* **Impactos nos Compiladores**
  * De fato, o desempenho de um sistema de computação é tão dependente da tecnologia de compilação que os compiladores são usados como uma ferramenta na avaliação dos conceitos arquitetônico antes que um computador seja montado.

Evolução das Linguagens de Programação

* **Impactos nos Compiladores**
  * O projeto de compiladores é desafiador.
  * Um compilador por si só é um programa grande.
  * Além do mais, muitos sistemas de processamento de linguagem modernos tratam de várias linguagens fonte e máquinas alvo dentro de um mesmo arcabouço (framework); ou seja, eles servem como famílias de compiladores, possivelmente consistindo em milhões de linhas de código.

Evolução das Linguagens de Programação

* **Impactos nos Compiladores**
  * Dessa forma, boas técnicas de engenharia de software são essenciais para a criação e evolução dos processadores de linguagem modernos.

Evolução das Linguagens de Programação

* **Impactos nos Compiladores**
  * Um compilador precisa traduzir corretamente um conjunto potencialmente infinito de programas que poderiam ser escritos na linguagem fonte.
  * O problema de gerar código objeto ótimo a partir de um programa fonte é, em geral, **indecidível**; assim, os projetistas de compiladores precisam avaliar as escolhas sobre quais problemas enfrentar e quais heurísticas utilizar para resolver o problema de gerar um código eficiente.

Evolução das Linguagens de Programação

* **Impactos nos Compiladores**
  * Estudar compiladores também é estudar de que forma a teoria encontra a prática.
  * A finalidade dessa disciplina é ensinar a metodologia e as ideias fundamentais usadas no projeto de compiladores.
  * Não é intenção da disciplina ensinar todos os algoritmos e técnicas que poderiam ser usados para a criação de um sistema de processamento de linguagem de última geração. Porém, teremos o conhecimento básico para entender como implementar um compilador de modo relativamente fácil.

Exercício

* **Indique quais dos seguintes termos:**

  a) imperativa

  b) declarativa

  c) von Neumann

  d) orientada por objeto

  e) funcional

  f) terceira geração

  g) quarta geração

  h) de scripting

  Aplicam-se às seguintes linguagens:

Exercício

* **Linguagens:**
  1) C
  2) C++
  3) Cobol
  4) Fortran
  5) Java
  6) Lisp
  7) ML
  8) Perl
  9) Python
  10) VB

A Ciência da Criação de um Compilador

* **A Ciência da Criação de um Compilador**
  * O projeto de compilador é repleto de belos exemplos nos quais problemas complicados do mundo real são solucionados abstraindo-se matematicamente a essência do problema.

  * Esses exemplos servem como excelentes ilustrações de como as abstrações podem ser usadas para solucionar problemas: dado um problema, formule uma abstração matemática que capture suas principais características e resolva-o usando técnicas matemáticas.

A Ciência da Criação de um Compilador

* **A Ciência da Criação de um Compilador**
  * A formulação do problema precisa ser baseada em um conhecimento sólido sobre as características dos programas de computador, e a solução precisa ser validada e refinada empiricamente.

A Ciência da Criação de um Compilador

* **A Ciência da Criação de um Compilador**
  * Um compilador precisa aceitar todos os programas fonte que estão de acordo com a especificação da linguagem; **o conjunto de programas fonte é infinito**, e qualquer programa pode ser muito grande, consistindo em possivelmente milhões de linhas de código.

A Ciência da Criação de um Compilador

* **A Ciência da Criação de um Compilador**
  * Qualquer transformação realizada pelo compilador durante a tradução de um programa fonte precisa preservar a semântica do programa sendo compilado.
  * Logo, os projetistas de compiladores têm influência não apenas sobre os compiladores que eles mesmos criam, mas sobre todos os programas que seus compiladores compilam.

A Ciência da Criação de um Compilador

* **A Ciência da Criação de um Compilador**
  * Essa influência torna a escrita de compiladores particularmente recompensadora.
  * Porém, também torna **desafiador** o desenvolvimento de compiladores.

A Ciência da Criação de um Compilador

* **Modelagem no Projeto e Implementação do Compilador**
  * O estudo dos compiladores é principalmente um estudo de como projetar os modelos matemáticas certos e escolher corretamente os algoritmos, mantendo-se o equilíbrio entre a necessidade de generalização e abrangência *versus* simplicidade e eficiência.

A Ciência da Criação de um Compilador

* **Modelagem no Projeto e Implementação do Compilador**
  * Entre os modelos mais importantes, destacam-se as máquinas de estado finito e expressões regulares, as gramáticas livres de contexto e as árvores.

  * As máquinas de estado finito e expressões regulares são úteis para descrever as unidades léxicas dos programas (palavras-chave, identificadores, etc.) e os algoritmos usados pelo compilador para reconhecer essas unidades.

A Ciência da Criação de um Compilador

* **Modelagem no Projeto e Implementação do Compilador**
  * As gramáticas livres de contexto são utilizadas para descrever a estrutura sintática das linguagens de programação, como o balanceamento dos parênteses ou construções de controle.
  * As árvores são consideradas um importante modelo para representar a estrutura dos programas e sua tradução para código objeto.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * O termo "otimização" no projeto de compiladores refere-se às tentativas que um compilador faz para produzir um código que seja mais eficiente do que o código óbvio.

  * "Otimização" é, portanto, um nome errado, pois não é possível garantir que um código produzido por um compilador seja tão ou mais rápido do que qualquer outro código que realiza a mesma tarefa.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * A otimização do código vem-se tornando cada vez mais **importante** e também mais **complexa** no projeto de um compilador.

  * Mais **complexa** porque as arquiteturas dos processadores se tornaram mais complexas, gerando mais oportunidades de melhorar a forma como o código é executado.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * E mais **importante** porque os computadores maciçamente paralelos exigem otimizações substanciais, ou seu desempenho é afetado por algumas ordens de grandeza.

  * Com a provável predominância de arquiteturas de máquina *multicore* (computadores com chips possuindo maiores quantidades de processadores), os compiladores enfrentarão mais um desafio para tirar proveito das máquinas multiprocessadoras.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * É difícil, se não impossível, construir um compilador robusto a partir de "pedaços".
  * Assim, uma extensa e útil teoria foi criada em torno do problema de otimização de código.
  * O uso de uma base matemática rigorosa permite mostrar que uma otimização está correta e produz o efeito desejado para todas as entradas possíveis.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * Mais adiante, veremos como os modelos baseados em gráficos, matrizes e programas lineares são necessários para auxiliar o compilador a produzir um código “ótimo”.

  * Por outro lado, a teoria pura sozinha é insuficiente. Assim, como para muitos problemas do mundo real, não existem respostas perfeitas. Na verdade quase todas as perguntas que fazemos a respeito da otimização de compiladores são indecidíveis.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * Uma das habilidades mais importantes no projeto de compiladores é a capacidade de formular corretamente o problema que se quer solucionar.

  * Para começar, precisamos de um bom conhecimento do comportamento dos programas, experimentando-os e avaliando-os completamente, a fim de validar nossas intuições.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * As otimizações de um compilador precisam atender os seguintes objetivos de projeto:
    * A otimização precisa ser correta, ou seja, preservar a semântica do programa compilado.
    * A otimização precisa melhorar o desempenho de muitos programas.
    * O tempo de compilação precisa continuar razoável.
    * O esforço de engenharia empregado precisa ser administrável.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * Nunca é demais enfatizar a importância da exatidão. É trivial escrever um compilador que gera código rápido se este código não precisa estar correto!
  * É tão difícil ter compiladores otimizadores corretos que ousamos dizer que nenhum compilador otimizador está livre de erros!
  * Assim, o objetivo mais importante no desenvolvimento de um compilador é que ele seja **correto**.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * O **segundo** objetivo é que o compilador seja eficiente na melhoria do desempenho de muitos programas de entrada.
  * Normalmente, desempenho diz respeito à velocidade de execução do programa. Especialmente em aplicações embutidas, também pode ser necessário diminuir o tamanho do código gerado.
  * E, no caso de dispositivos móveis, também é desejável que o código gerado minimize o consumo de energia.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * Normalmente, as mesmas otimizações que diminuem o tempo de execução reduzem também a energia gasta.
  * Além do desempenho, são importantes ainda os aspectos de usabilidade, como relatório de erros e depuração.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * **Terceiro,** precisamos manter o tempo de compilação pequeno para dar suporte a um ciclo rápido de desenvolvimento e depuração.
  * Fica mais fácil atender este requisito à medida que as máquinas se tornam mais rápidas.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * Normalmente, primeiro desenvolve-se e depura-se um programa sem otimizações.
  * Usando esta estratégia, não apenas reduz-se o tempo de compilação, porém, mais importante, os programas não otimizados se tornam mais fáceis de depurar, pois as otimizações introduzidas por um compilador tendem a obscurecer o relacionamento entre o código fonte e código objeto.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * A ativação de otimizações no compilador às vezes expõe novos problemas no programa fonte.
  * Assim, o teste precisa ser novamente realizado no código otimizado. A necessidade do teste adicional às vezes desencoraja o uso das otimizações nas aplicações, especialmente se seu desempenho não for crítico.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * Finalmente, um compilador é um sistema complexo. Temos de manter o sistema simples para garantir que os custos de engenharia e manutenção do compilador sejam viáveis.
  * Podemos implementar um número infinito de otimizações de programa, e é necessário um esforço incomum para criar uma otimização correta e eficiente.
  * Temos de priorizar as otimizações, implementando apenas aquelas que proporcionam maiores benefícios nos programas fontes encontrados na prática.

A Ciência da Criação de um Compilador

* **A Ciência da Otimização do Código**
  * Assim, estudando os compiladores, aprendemos não apenas a construir um compilador, mas também a metodologia geral para solucionar problemas complexos e problemas abertos.
  * A técnica usada no desenvolvimento de compilador envolve teoria e experimentação.
  * Normalmente, começamos formulando o problema com base em nossas intuições sobre quais são os aspectos importantes.

Aplicações da Tecnologia de Compiladores

* **Aplicações da Tecnologia de Compiladores**
  * O projeto de um compilador não diz respeito apenas a compiladores, e muitas pessoas usam a tecnologia aprendida pelo estudo de compiladores na escola, embora nunca tenham, estritamente falando, nem mesmo escrito parte de um compilador para uma linguagem de programação conhecida.

Aplicações da Tecnologia de Compiladores

* **Aplicações da Tecnologia de Compiladores**
  * A tecnologia de compiladores possui também outras aplicações importantes.
  * Além do mais, o projeto de um compilador tem impacto em várias outras áreas da ciência da computação.
  * A seguir, veremos as interações e aplicações mais importantes dessa tecnologia.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Uma linguagem de programação de alto nível define uma abstração de programação: o programador escreve um algoritmo usando a linguagem, e o compilador deve traduzir esse programa para a linguagem objeto.
  * Em geral, é mais fácil programar em linguagens de programação de alto nível, mas elas são menos eficientes, ou seja, os programas objetos são executados mais lentamente.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Os programadores que usam uma linguagem de baixo nível têm mais controle sobre uma computação e podem, a princípio, produzir código mais eficiente.

  * Infelizmente, os programas feitos desta forma são mais difíceis de escrever e – pior ainda – menos transportáveis para outras máquinas, mais passíveis de erros e mais difíceis de manter. Os compiladores otimizadores dispõem de técnicas para melhorar o desempenho do código gerado, afastando assim a ineficiência introduzida pelas abstrações de alto nível.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Exemplo: A palavra chave ***register*** da linguagem de programação C é um velho exemplo da interação entre a tecnologia de compiladores e a evolução da linguagem.

  * Quando a linguagem C foi criada em meados da década de 1970, considerou-se importante permitir o controle pelo programador de quais variáveis do programa residiam nos registradores.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Esse controle tornou-se desnecessário quando foram desenvolvidas técnicas eficazes de alocação de registradores, e a maioria dos programas modernos não usa mais esse recurso da linguagem.

  * Na verdade, os programas que usam a palavra-chave ***register*** podem perder a eficiência, pois os programadores normalmente não são os melhores juízes em questões de muito baixo nível, como a alocação de registradores.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * A escolha de uma boa estratégia para a alocação de registradores depende muito de detalhes específicos de uma arquitetura de máquina.

  * Tomar decisões sobre o gerenciamento de recursos de baixo nível, como a alocação de registradores, pode de fato prejudicar o desempenho, especialmente se o programa for executado em máquinas diferentes daquela para a qual ele foi escrito.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * A adoção de novas linguagens de programação tem sido na direção daquelas que oferecem maior nível de abstração. Nos anos 80, C foi a linguagem de programação de sistemas predominante.

  * Muitos dos novos projetos iniciados nos 1990 escolheram C++. A linguagem Java, introduzida em 1995, rapidamente ganhou popularidade no final da década de 1990.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Os novos recursos de linguagem de programação introduzidos a cada rodada incentivaram novas pesquisas sobre otimização de compilador.
  * A seguir, veremos um panorama geral dos principais recursos de linguagens de programação que têm estimulado avanços significativos na tecnologia de compilação.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Praticamente todas as linguagens de programação comuns, incluindo C, Fortran e Cobol, admitem que os usuários definam tipos de dados compostos, como arranjos e estruturas, e fluxo de controle de alto nível, como loops e chamadas de procedimento.
  * Se simplesmente traduzirmos diretamente para código de máquina cada construção de alto nível ou operação de acesso, o resultado será ineficaz.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Um conjunto de otimizações, conhecido como ***otimizações de fluxo de dados***, foi desenvolvido para analisar o fluxo de dados de um programa, e remover as redundâncias encontradas nessas construções.

  * Essas otimizações têm-se revelado eficazes, e o código gerado se assemelha ao código escrito em um nível mais baixo por um programa habilidoso.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * A orientação por objeto foi introduzida inicialmente na linguagem Simula em 1967, e incorporada em linguagens como Smalltalk, C++, C# e Java.
  * As principais ideias por trás da orientação por objeto são:
    * **Abstração** de dados e
    * **Herança** de propriedades
  * Ambas consideradas fundamentais para tornar os programas mais modulares e mais fáceis de manter.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Os programas orientados por objeto são diferentes daqueles escritos em várias outras linguagens, pois possuem mais, porém menores, procedimentos (chamados ***métodos*** no contexto de orientação por objeto).

  * Assim, as otimizações presentes no compilador precisam ser eficazes além dos limites de procedimento do programa fonte.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * A "expansão em linha" (do inglês, inlining) de procedimento, que corresponde à substituição de uma chamada de procedimento pelo seu corpo, é particularmente útil neste contexto.
  * Também têm sido desenvolvidas otimizações para agilizar os disparos dos métodos virtuais.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * A linguagem Java possui muitos recursos que tornam a programação mais fácil, e muitos deles foram introduzidos anteriormente em outras linguagens.
  * A linguagem é **segura em termos de tipo**, ou seja, um objeto não pode ser usado como um objeto de um tipo não relacionado.
  * Todos os acessos a arranjos são verificados para garantir que estejam **dentro dos limites do arranjo**.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Java não possui apontadores nem permite aritmética de apontadores. Ela possui uma função primitiva (*built-in*) para a **coleta de lixo**, a qual libera automaticamente a memória das variáveis que não são mais usadas.

  * Embora todos esses recursos facilitem a programação, eles geram um custo adicional no tempo de execução.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Foram desenvolvidas otimizações no compilador para reduzir esse custo adicional, por exemplo, eliminando verificações de limites desnecessárias e alocando na pilha (stack), ao invés de na heap, os objetos que não são acessíveis fora de um procedimento.

  * Algoritmos eficientes também foram desenvolvidos para reduzir o custo adicional atribuído à coleta de lixo.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * Além disso, a linguagem Java é projetada para prover código transportável e móvel.
  * Os programas são distribuídos como bytecode Java, que precisa ser interpretado ou compilado para o código nativo dinamicamente, ou seja, em tempo de execução.

Aplicações da Tecnologia de Compiladores

* **Implementação de Linguagens de Programação de Alto Nível**
  * A compilação dinâmica também tem sido estudada em outros contextos, nos quais a informação é extraída dinamicamente em tempo de execução e usada para produzir um código mais otimizado.
  * Na otimização dinâmica, é importante minimizar o tempo de compilação, pois ele faz parte do custo adicional da execução.
  * Uma técnica muito utilizada é compilar e otimizar apenas as partes do programa que serão executadas com mais frequência.

Fundamentos da Linguagem de Programação

* **Fundamentos da Linguagem de Programação**
  * É importante lembrarmos a terminologia e as diferenças mais importantes que aparecem no estudo das linguagens de programação.

Fundamentos da Linguagem de Programação

* **A Diferença entre Estático e Dinâmico**
  * Um dos aspectos mais importantes ao projetar um compilador para uma linguagem diz respeito às ***decisões*** que o compilador pode tomar sobre um programa.
  * Se uma linguagem utiliza uma política que permite ao compilador decidir a respeito de uma questão, dizemos que a linguagem usa a política ***estática*** ou que a questão pode ser decidida em ***tempo de compilação***.
  * Por outro lado, uma política que só permite que uma decisão seja tomada quando executamos o programa é considerada uma política ***dinâmica***, ou que exige decisão em ***tempo de execução***.

Fundamentos da Linguagem de Programação

* **A Diferença entre Estático e Dinâmico**
  * Uma questão na qual nos concentramos é o escopo das declarações. O escopo de uma declaração de x é a região do programa em que os usos de x se referem a essa declaração.

  * Uma linguagem usa ***escopo estático*** ou escopo léxico se for possível determinar o escopo de uma declaração examinando-se apenas o programa. Caso contrário, a linguagem utiliza ***escopo dinâmico***.

Fundamentos da Linguagem de Programação

* **A Diferença entre Estático e Dinâmico**
  * Com o escopo dinâmico, enquanto o programa é executado, o mesmo uso de x poderia referir-se a qualquer uma dentre as várias declarações diferentes de x.
  * A maioria das linguagens, como C e Java, utiliza ***escopo estático***.

Estático x Dinâmico

```
x: integer;

procedure print_x()
begin
   print(x);
end

procedure p2
x: integer;
begin
   x= 4;
   print_x();
end

begin
   x = 3;
   p2();
end
```

* Se o escopo for ***estático***, o programa **imprime 3**.
* Se o escopo for ***dinâmico*** o programa **imprime 4**.

Estático x Dinâmico

* **Escopo Estático (léxico)**
  * Em linguagens de programação com escopo estático (ou léxico), o escopo é determinado através da estrutura textual do programa. Usando escopo estático (léxico), a vinculação de um nome no ***ambiente*** é determinada da seguinte maneira:

    * Se o nome foi declarado no bloco de execução, aquela vinculação será usada. Caso contrário, se o nome não foi declarado no bloco em execução, ele deve ser buscado nos blocos que o envolvem, do imediatamente envolvente até o mais distante. Se todos os blocos envolventes tiverem sido verificados e a declaração não encontrada, se o nome está no ambiente global, aquela vinculação será usada, caso contrário não há vinculação para aquele nome no ***ambiente***.

Estático x Dinâmico

* **Escopo Estático (léxico)**
  * Pode-se informalmente dizer que o trecho de código onde um nome é visível é o bloco onde foi declarado e todos os blocos aninhados dentro dele, e por este motivo muitas vezes utiliza-se **"escopo léxico"** como sinônimo de **"escopo estático"**.

  * A maioria das linguagens, incluindo C e sua família, utiliza escopo estático. As regras de escopo para C são baseadas na estrutura do programa. Outras linguagens mais modernas, como C++, C# e Java, também oferecem controle explícito sobre escopos, por meio de palavras-chave como ***public***, ***private*** e ***protected***.

Estático x Dinâmico

* **Escopo Dinâmico**
  * Em linguagens de programação com escopo dinâmico, o escopo é determinado através da linha de execução do programa, sendo dependente portanto da ordem de execução das rotinas.

  * Usando escopo dinâmico, a vinculação válida para um nome é a criada mais recentemente durante a execução do programa, baseado em sequências de chamadas de unidades de programas, não no layout textual.

Estático x Dinâmico

* **A Diferença entre Estático e Dinâmico**
  * Como outro exemplo da distinção entre estático e dinâmico, considere o uso do termo ***static*** aplicado aos dados de uma declaração de classe Java.

  * Em Java, uma variável é um ***nome*** que designa uma localização de memória usada para armazenar o ***valor*** de um dado.

Estático x Dinâmico

* **A Diferença entre Estático e Dinâmico**
  * Neste contexto, ***static*** refere-se não ao escopo da variável, mas sim à capacidade de o compilador determinar a localização na memória onde a variável declarada pode ser encontrada.

Estático x Dinâmico

* Uma declaração como:

```
public static int x;
```

torna x uma variável de classe e diz que existe apenas uma cópia de x, não importa quantos objetos dessa classe sejam criados. Além disso, o compilador pode determinar uma localização na memória onde esse inteiro x será mantido. Ao contrário, se "***static***" fosse omitido dessa declaração, cada objeto da classe teria sua própria localização onde x seria mantido, e o compilador não poderia determinar todos esses lugares antes da execução do programa.

Fundamentos da Linguagem de Programação

* **Ambientes e Estados**
  * Outra distinção importante que precisamos fazer ao discutir linguagens de programação é se as mudanças que ocorrem enquanto o programa é executado afetam os valores dos elementos de dados ou afetam a interpretação dos nomes para esses dados.

  * Por exemplo, a execução de uma atribuição como $\boldsymbol{x = y + 1}$ muda o valor denotado pelo nome x. Mais especificamente, a atribuição muda o valor em alguma localização designada para x.

Fundamentos da Linguagem de Programação

* **Ambientes e Estados**
  * Pode não ser tão claro que a localização denotada por x pode mudar durante a execução. Por exemplo, se x não for uma variável (ou "classe") estática, cada objeto da classe tem sua própria localização para uma instância da variável x.

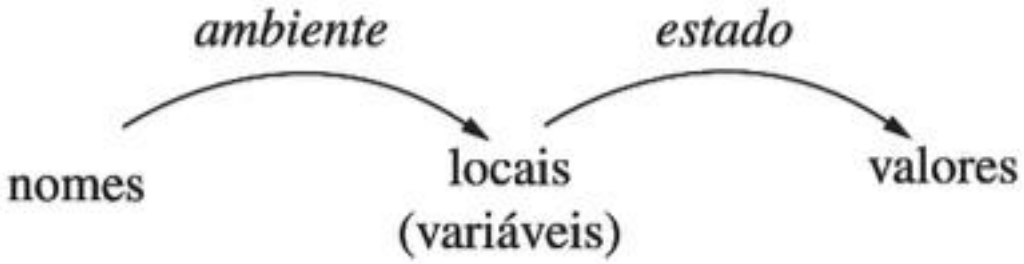
  * Nesse caso, a atribuição para x pode mudar qualquer uma dessas variáveis de "instância", dependendo do objeto ao qual é aplicado um método contendo essa atribuição.

Fundamentos da Linguagem de Programação

* **Ambientes e Estados**
  * A associação dos nomes às localizações na memória (o armazenamento) e depois aos valores pode ser descrita por dois mapeamentos que mudam à medida que o programa é executado.

Ambientes e Estados

* **Ambiente x Estado**
  * O ***ambiente*** é um mapeamento de um nome para uma posição de memória. Como as variáveis se referem a localizações, poderíamos como alternativa definir um ambiente como um mapeamento entre nomes e variáveis.
  * O ***estado*** é um mapeamento de uma posição de memória ao valor que ela contém.



Mapeamento em dois estágios entre nomes e valores.

  * Os ambientes mudam de acordo com as regras de escopo de uma linguagem.

Fundamentos da Linguagem de Programação

* **Nomes, identificadores e variáveis**
  * Embora os termos “nome” e “variável” normalmente se referem à mesma coisa, vamos usá-los cuidadosamente para distinguir entre os nomes usados em tempo de compilação e as localizações em tempo de execução denotadas pelos nomes.
  * Um ***identificador*** é uma cadeia de caracteres, normalmente letras ou dígitos, que se refere a (identifica) uma entidade, como um objeto de dados, um procedimento, uma classe ou um tipo. Todos os identificadores são nomes, mas nem todos os nomes são identificadores.

Fundamentos da Linguagem de Programação

* **Nomes, identificadores e variáveis**
  * Os nomes também podem ser expressões. Por exemplo, o nome $x.y$ poderia designar o campo y de uma estrutura representada por x. Neste contexto, x e y são identificadores, enquanto $x.y$ é um nome, mas não um identificador.
  * Nomes compostos como $x.y$ são chamados de nomes ***qualificados***.

Fundamentos da Linguagem de Programação

* **Nomes, identificadores e variáveis**
  * Uma ***variável*** refere-se a um endereço particular de memória. É comum que o mesmo identificador seja declarado mais de uma vez, sendo que cada declaração introduz uma nova variável.

  * Mesmo que cada identificador seja declarado apenas uma vez, um identificador local a um procedimento recursivo continuará referindo-se a diferentes endereços de memória em diferentes momentos.

Fundamentos da Linguagem de Programação

* **Estrutura de Blocos**
  * Para este tópico vamos considerar as regras de escopo estático para uma linguagem de blocos, onde um ***bloco*** é um agrupamento de declarações e comandos.

  * C utiliza chaves {e} para delimitar um bloco; o uso alternativo ***begin*** e ***end*** para a mesma finalidade teve origem na linguagem Algol.

Fundamentos da Linguagem de Programação

* **Estrutura de Blocos**
  * **Procedimentos x Funções x Métodos**
    * Para evitar dizer "procedimentos, funções ou métodos" toda vez que quisermos falar sobre um subprograma que pode ser chamado, normalmente nos referimos a todos eles como "procedimentos".
    * A exceção é que, quando se fala explicitamente de programas em linguagens como C, que só possuem funções, nos referimos a eles como "funções". Ou, se estivermos discutindo sobre uma linguagem como Java, que possui apenas métodos, também usamos esse termo.

Fundamentos da Linguagem de Programação

* **Estrutura de Blocos**
  * **Procedimentos x Funções x Métodos**
    * Uma função geralmente retorna um valor de algum tipo (o "tipo de retorno"), enquanto um procedimento não retorna nenhum valor.

    * A linguagem C e outras semelhantes, que possuem apenas funções, tratam os procedimentos como funções, mas com um tipo de retorno especial "***void***", que significa nenhum valor de retorno. As linguagens orientadas por objeto, como Java e C++ utilizam o termo "métodos". Estes podem comportar-se como funções ou procedimentos, mas estão associados a uma classe em particular.

Fundamentos da Linguagem de Programação

* **Estrutura de Blocos**
  * Em C, a sintaxe dos blocos é dada por:
    * Bloco é um tipo de comando. Os blocos podem aparecer em qualquer lugar em que outros tipos de comandos (como os comandos de atribuição) podem aparecer.
    * Um bloco é uma sequência de declarações seguida por uma sequência de comandos, todos entre chaves.

Fundamentos da Linguagem de Programação

* **Estrutura de Blocos**
  * Observe que essa sintaxe permite que os blocos sejam aninhados um dentro do outro. Essa propriedade de encaixamento é chamada de ***estrutura de bloco***.

  * A família da linguagem C possui estrutura de bloco, exceto pelo fato de que uma função não pode ser definida dentro de outra função.

Fundamentos da Linguagem de Programação

* **Mecanismos de Passagem de Parâmetros**
  * Todas as linguagens de programação possuem a noção de procedimento, mas elas podem diferir no modo como esses procedimentos recebem seus argumentos.

  * Neste tópico, vamos considerar como ***parâmetros reais*** (ou parâmetros usados na chamada de um procedimento) estão associados aos ***parâmetros formais*** (aqueles usados na definição do procedimento.

Fundamentos da Linguagem de Programação

* **Mecanismos de Passagem de Parâmetros**
  * O mecanismo utilizado determina como o código na sequência de chamada trata os parâmetros.
  * A grande maioria das linguagens utiliza “chamada por valor”, a ou “chamada por referência”, ou ambas.

Fundamentos da Linguagem de Programação

* **Mecanismos de Passagem de Parâmetros**
  * Na ***chamada por valor***, o parâmetro real é avaliado (se for uma expressão) ou copiado (se for uma variável).
  * O valor é armazenado em uma localização pertencente ao parâmetro formal correspondente do procedimento chamado. Esse método é usado em C e Java, e é uma opção comum em C++, bem como na maioria das linguagens.

Fundamentos da Linguagem de Programação

* **Mecanismos de Passagem de Parâmetros**
  * A chamada por valor tem o efeito de que toda a computação envolvendo os parâmetros formais feita pelo procedimento chamado é local a esse procedimento, **e os próprios parâmetros reais não podem ser alterados**.

Fundamentos da Linguagem de Programação

* **Mecanismos de Passagem de Parâmetros**
  * Na ***chamada por referência***, o endereço do parâmetro real é passado ao procedimento chamado como o valor do parâmetro formal correspondente.
  * Os usos do parâmetro formal no código chamado são implementados seguindo-se esse apontador para o local indicado por quem chamou.
  * As mudanças no parâmetro formal, portanto, aparecem como mudanças no parâmetro real.

Fundamentos da Linguagem de Programação

* **Sinônimos**
  * Existe uma consequência interessante de passagem de parâmetros na chamada por referência ou sua simulação, como em Java, onde as referências a objetos são passadas por valor.

  * É possível que dois parâmetros formais se refiram ao mesmo local; tais variáveis são consideradas sinônimas (***aliases***) uma da outra. Como resultado, duas variáveis quaisquer, que correspondem a dois parâmetros formais distintos, também podem tornar-se sinônimos uma da outra.

Fundamentos da Linguagem de Programação

* **Sinônimos**
  * Acontece que entender os sinônimos e os mecanismos que o criam é essencial se um compilador tiver de otimizar um programa.
  * Existem muitas situações em que só podemos otimizar o código se tivermos certeza de que certas variáveis não são sinônimas uma da outra.

Fundamentos da Linguagem de Programação

* **Sinônimos**
  * Por exemplo, poderíamos determinar que $x = 2$ é o único local em que a variável x é atribuída. Nesse caso, podemos substituir um uso de x por um uso de 2;

  * Outro exemplo, substituir $a = x + 3$ pela atribuição mais simples $a = 5$. Mas suponha que existisse outra variável y que fosse um alias de x. Então a atribuição $y = 4$ poderia ter um efeito inesperado ao alterar x. Isso também poderia significar que a substituição de $a = x + 3$ por $a = 5$ seria um erro; o valor apropriado de a poderia ser 7 nesse caso.

Resumo

* ***Processadores de linguagem***: Um ambiente de desenvolvimento de software integrado inclui muitos tipos diferentes de processadores de linguagem, como compiladores, interpretadores, montadores, editores de ligação, carregadores, depuradores e profilers.

* ***Fases do compilador***: Um compilador opera como uma sequência de fases, cada uma transformando o programa fonte de uma representação intermediária para outra.

* ***Linguagens de máquina e linguagem simbólica***: As linguagens de máquina foram as linguagens de programação da primeira geração, seguidas pelas linguagens simbólicas. A programação nessas linguagens era demorada e passível de erro.

Resumo

* ***Modelagem no projeto do compilador***: O projeto do compilador é uma das áreas em que a teoria teve mais impacto na prática. Os modelos mais importantes incluem gramáticas, expressões regulares, árvores e muitos outros.

* ***Otimização de código***: Embora o código não possa ser realmente "otimizado", a ciência de melhorar a eficiência do código é complexa e muito importante. Ela é uma parte importante do estudo da compilação.

* ***Linguagens de alto nível***: Com o passar do tempo, as linguagens de programação assumem cada vez mais tarefas que anteriormente eram responsabilidade do programador, como o gerenciamento de memória, a verificação de consistência de tipo ou a execução paralela do código.

Resumo

* ***Compiladores e arquitetura de computador***: A tecnologia de compiladores influencia a arquitetura de computadores, além de ser influenciada pelos avanços na arquitetura. Muitas inovações modernas na arquitetura dependem da capacidade de os compiladores extraírem dos programas fonte as oportunidades de usar as capacidades do hardware de modo eficaz.

* ***Produtividade e segurança do software***: A mesma tecnologia que permite aos compiladores otimizar seu código pode ser usada para diversas tarefas de análise de programa, desde detectar erros comuns em programas até descobrir que um programa é vulnerável a um dos muitos tipos de intrusões descobertas pelos hackers.

Resumo

* ***Regras de escopo***: O escopo de uma declaração de x é o contexto em que os usos de x se referem a essa declaração, ou seja, é o fragmento do programa em que a declaração tem efeito. Uma linguagem usa escopo estático ou escopo léxico se for possível determinar o escopo de uma declaração examinando apenas o programa. Caso contrário, a linguagem usa o escopo dinâmico.

* ***Ambiente***: O ambiente mapeia um nome para uma localização de memória, enquanto o estado mapeia uma posição de memória ao valor que ela contém.

* ***Estrutura de bloco***: As linguagens que permitem que os blocos sejam encaixados são consideradas como tendo uma estrutura de bloco. Um nome x em um bloco aninhado B está no escopo de uma declaração D de x em um bloco delimitado se não houver outra declaração de x em um bloco entre eles.

Resumo

* ***Passagem de parâmetros***: Os parâmetros são passados por valor ou por referência de um procedimento chamador para o procedimento chamado. Quando grandes objetos são passados por valor, os valores passados são, na realidade, referências aos próprios objetos, resultando em uma efetiva chamada por referência.

* ***Sinônimo***: Quando os parâmetros são (efetivamente) passados por referência, dois parâmetros formais podem referir-se ao mesmo objeto. Essa possibilidade permite que a alteração em uma variável mude outra variável.

Obrigado.

joaopauloaramuni@gmail.com
joaopauloaramuni@fumec.br

UNIVERSIDADE FUMEC
50 Anos