# Linguagens de Programação: Avaliando uma Linguagem de Programação

Prof. Arnaldo Candido Junior

UNESP – IBILCE

São José do Rio Preto, SP

# Formas de Classificar

- Quanto ao paradigma: imperativo, orientado a objetos, funcional, lógico
- Quanto ao nível: baixo, médio, alto
- Quanto à geração: 1ª, 2ª, 3ª, 4ª, 5ª
- Quanto ao método de execução: compilada, interpretada o híbrida

# Nível

- **Baixo nível**: próximo da máquina. Exemplo: Assembly
- **Médio nível**: recursos próximos de máquina e recursos amigáveis ao programador. Exemplo: C
- **Alto nível**: recursos amigáveis ao programador. Exemplo: Python
  - A grande maioria das linguagens de programação está nesta categoria

# Geração

- **1ª geração**: código de máquina. Exemplo:
  - 0010 0001 0110 1100
- **2ª geração**: linguagem de montagem
  - Comandos binários foram substituídos por **mnemônicos**
  - Transformando o exemplo anterior em algo como: ADD R1, TOTAL
  - Vantagem: muito eficientes

# Geração (2)

- **3ª geração**: linguagem orientada a usuário.
  - Projetadas para programadores projetarem sistemas de acordo com o perfil do usuário alvo
  - Buscam resolver problemas nos nichos principais: Científico (Fortran), comercial (Cobol), Inteligência Artificial (Lisp), …
  - Algumas buscam ser multi-nicho (Ada)
  - Vantagem: fáceis de se usar (programadores)

# Geração (3)

- **4ª geração**: linguagem orientada à aplicação
  - Buscam atender aplicações específicas
  - Exemplos: Matlab, SQL, VHML, VBA
  - Vantagens: amigáveis ao usuário
    - Considerando que é um usuário com um perfil bastante técnico

# Geração (4)

- **5ª geração**: linguagem de representação de conhecimento
  - Mais voltado a Inteligência Artificial: ontologias, processamento de língua natural
  - Como codificar conhecimento sobre o mundo real para apoiar a tomada de decisão
- Importante: geração posterior != geração melhor. 3ª geração, no momento, tem mais aplicações

# Critérios de avaliação das linguagens

- **1. Legibilidade** (readability): é fácil de ler um programa na linguagem?
- **2. Escritabilidade** (writability - facilidade de escrita): é fácil escrever um programa?
- **3. Confiabilidade** (reliability): programas escritos na linguagem tendem a ser confiáveis (livres de erros?)
- **4. Custo** (cost): os programas são eficientes?

# Critérios: legibilidade

- **Simplicidade**: é fácil aprender a linguagem?
- **Tipos de dados**: a linguagem oferece os tipos de dados adequados para os problemas que se propõe a resolver?
- **Projeto de sintaxe** (syntax design): a sintaxe da linguagem é intuitiva? As palavras reservadas são adequadas?

# Critérios: legibilidade (2)

- **Ortogonalidade**: os conceitos (blocos de construção) da linguagem são independentes uns dos outros?
  - É possível um array para cada tipo de variável?
  - É possível um ponteiro para cada tipo de variável?

# Critérios: facilidade de escrita

- **Simplicidade e ortogonalidade**: também afetam a escritabilidade. Existem muitas formas diferentes de se implementar um mesmo trecho de código?
  - Existe chance dos programadores se confundirem e utilizarem as construções de forma incorreta?
- **Expressividade**: é possível fazer operações complexas com um programa simples e enxuto?

# Critérios: facilidade de escrita (2)

- **Suporte a abstração**: é possível usar estruturas/operações ignorando-se detalhes de implementação?
  - Abstrair é observar características comuns de um grupo (ou um todo)
  - Exemplo de abstração comuns em programação: uma lista de “coisas” (inteiros, strings, structs, etc)

# Confiabilidade

- **Aliasing** (múltiplos nomes): a mesma variável (mesma posição de memória) pode ter nomes diferentes?
  - Ex.: ponteiros. Mudança em um ponteiro repercute em todos que apontam para a mesma área.
- **Manipulação de exceções**: a linguagem oferece recursos para tratar excessões que ocorrem durante o decorrer do programa?

# Confiabilidade (2)

- **Legibilidade e facilidade de escrita**: também influenciam este quesito. Programas fáceis de se criar e entender são menos propensos a erros.
- **Tipagem**: quanto à analise de tipos, uma linguagem pode ser:
  - Estática ou dinamicamente tipada
  - Forte ou fracamente tipada

# Confiabilidade (3)

- **Estaticamente tipada** (ex.: C): tipos são checados durante a compilação.
- **Dinamicamente tipada** (ex.: Python): tipos são checados durante a execução.
- **Fracamente tipada** (ex. JavaScript): um tipo é convertido em outro automaticamente.
  - “1” + 3 == “13”
  - “13” - 3 == 10

# Confiabilidade (4)

- **Fortemente tipada** (ex.: C): conversões não são automáticas, salvo casos mais simples.
  - Caso especial: **coerção**
  - Permite conversão em casos simples (ex.: int para float)
- Importante: não há consenso entre os autores e as quatro definições de tipagem apresentadas podem variar

# Custos

- 1. Custo para treinar dos programadores na linguagem
- 2. Custo para escrever programas na linguagem
- 3. Custo do compilador e das plataformas de desenvolvimento (hoje em dia, abertos)
- 4. Custo de executar o programa (equipamento necessário)

# Custos (2)

- 5. Custo do sistema de apoio (bibliotecas, sistema operacional, dependências)
- 6. Custo de confiabilidade. Ex.: piloto automático, planta nuclear, sistema de tomografia:
  - Embutido no desenvolvimento do sistema e no equipamento necessário
- 7. Custo de manutenção do software

# Trade-offs

- Trade-offs (compromissos, equilíbrio, balanço): requisitos conflitantes.
  - Requisitos conflitantes precisam ser conciliados
  - Por exemplo: processador vs memória; disco vs rede
  - Prioridade pode mudar de acordo com o problema.
- Projeto de linguagem de programação: envolve diversos trade-offs (exemplos a seguir...)

# Trade-offs (2)

- **Expressividade** vs **legibilidade**. Exemplos:
  - Expressões regulares: muito poderosas, mas difíceis de ler
  - Perl: muitas formas diferentes de implementar o mesmo trecho de código. Qual escolher?
  - C++: muitos recursos diferentes (herança múltipla, várias estratégias de gerenciamento de ponteiros). Qual usar?

# Trade-offs (3)

- **Confiabilidade** vs **custo de execução**.
  - Exemplo: checar índices em arrays
  - A falta de checagem levou a muitos problemas de segurança na linguagem C
    - Buffer overflow (extravasamento de buffer / estouro de buffer)

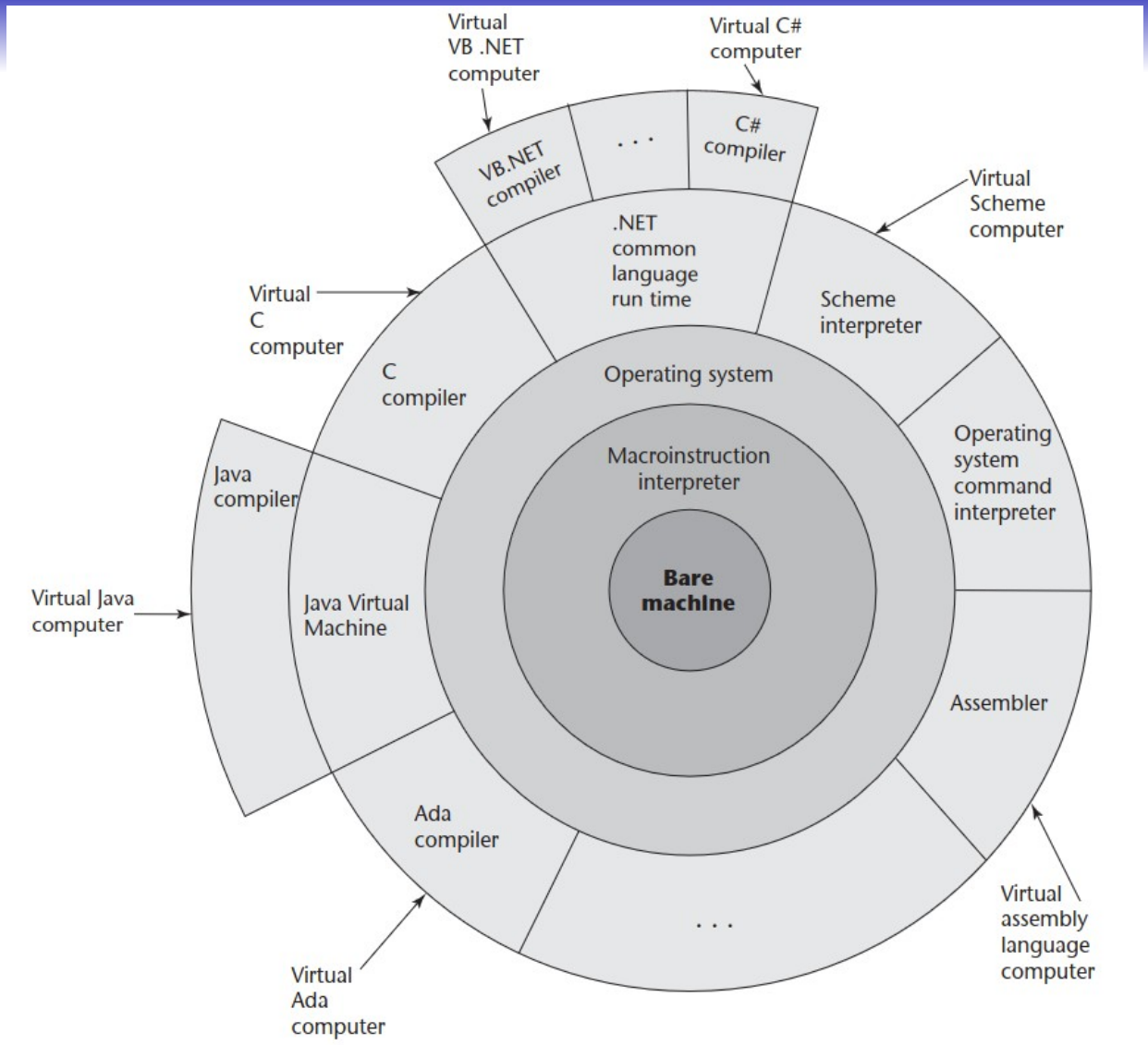
# Trade-offs (4)

- **Escritabilidade** vs **custo de execução**.
  - Existem muitas formas diferentes de manipular ponteiros em C++
  - Alternar entre as estratégias de manipulação pode ser confuso e introduzir erros no programa

# Métodos de implementação de linguagens

- **Métodos de implementação**: compilação, interpretação, híbrido
  - Um método de implementação é a estratégia da linguagem para rodar o código
  - Como traduzir o programa fonte em instruções executáveis pela máquina?
- Um sistema de implementação de uma linguagem envolve diferentes softwares: S.O., bibliotecas, frameworks e dependências de modo geral

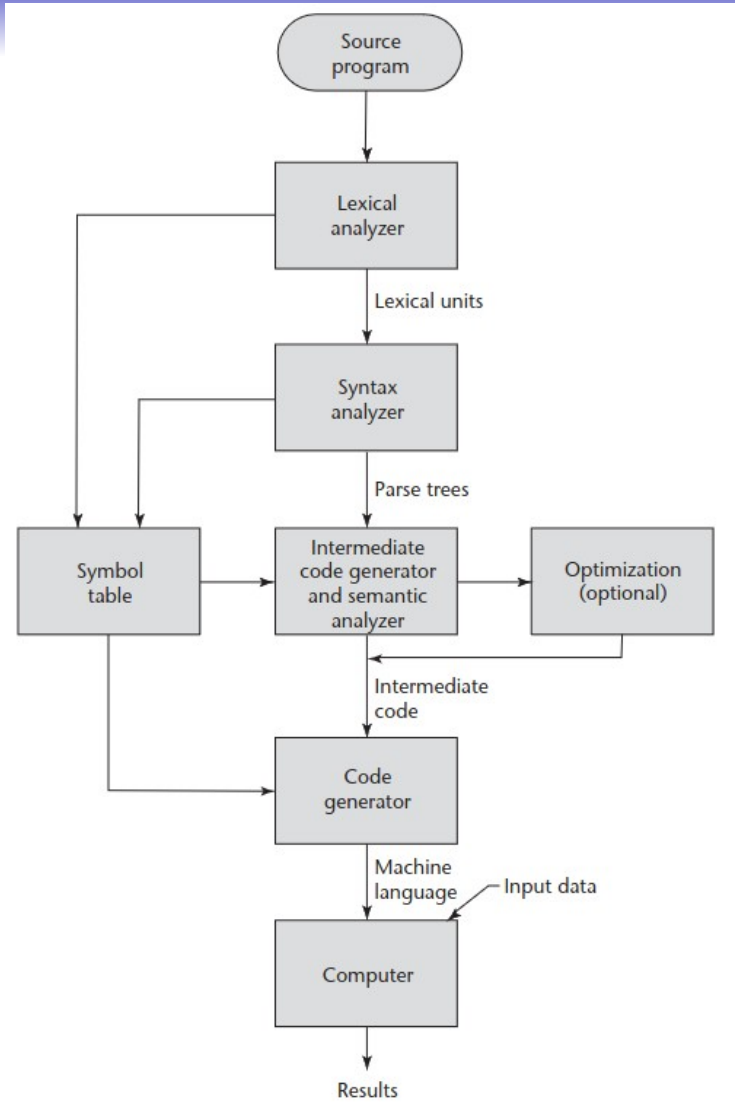
# Métodos de implementação



# Métodos de implementação: compilação

- **Compilação** é a tradução do programa para código de máquina, feita previamente, antes do programa ser usado
- Vantagem: estratégia de execução mais rápida
- Traduz código fonte para código de máquina
- Um compilador é dividido em etapas

# Métodos de implementação: compilação (2)




- Foco a seguir: três analisadores:
  - Léxico
  - Sintático
  - Semântico

# Analisador Léxico

- Analisador léxico: quebra o código fonte em unidades mínimas (tokens)

```
void
main
(
)
{
printf
“olá mundo\n”
;
}
...
```

# Analisador Léxico (2)

- Espaços, quebras de linhas, tabulações: são ignorados pelo analisador léxico
- Tokens processados são enviados para o analisador sintático

# Analisador sintático

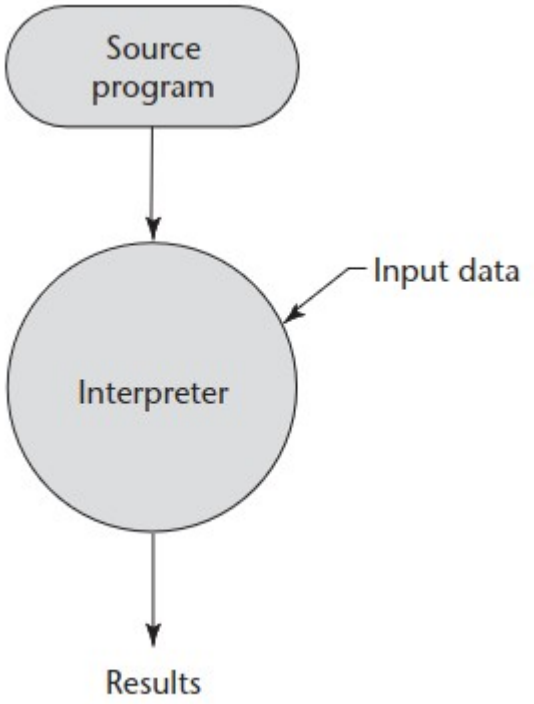
- Verifique se os tokens estão na **ordem** adequada
  - Que respeita a **sintaxe** da linguagem
- Exemplo em C:
  - Depois de um if sempre vem um “(“
  - Toda chave aberta deve ser fechada
  - Todo comando deve terminar em “;”
  - Etc

# Analisador semântico

- Verifique se as instruções fazem sentido. Exemplos:
  - Não se pode armazenar uma string em um inteiro
  - Não se pode subtrair uma string de outra
  - Não se pode armazenar um array de inteiros em uma variável do tipo ponteiro de real
- Cria uma **tabela de símbolos**

# Interpretação pura

- O código fonte é interpretado em tempo real e traduzido para instruções de máquina sob demanda
- Também usa, em menor grau, analisadores, como nas linguagens compiladas

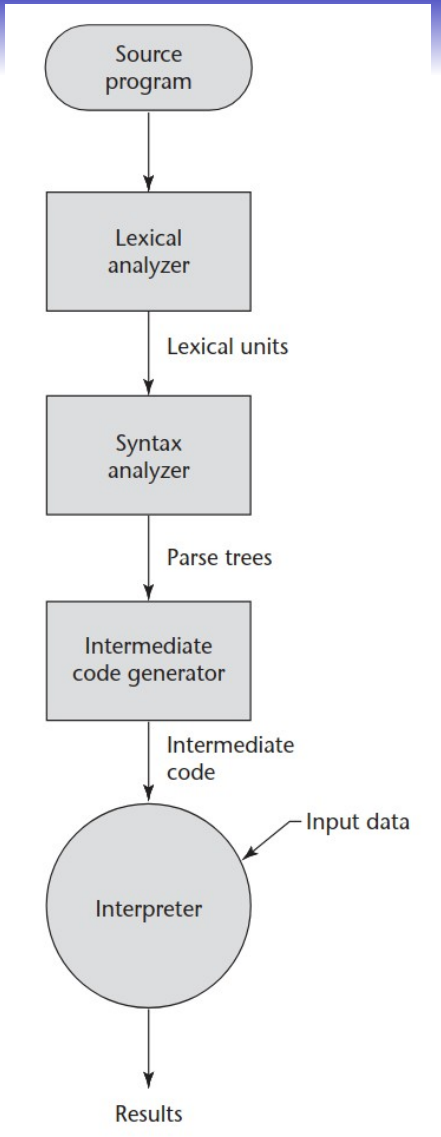


# Interpretação pura (2)

- A principal diferença é o gerador de código adaptado para tempo real e não tem representação intermediário
- Vantagem: mais flexível em algumas construções da linguagem; portável entre arquiteturas e sistemas operacionais
- Desvantagem: mais lenta, de 10 a 100 vezes mais lenta que uma linguagem compilada

# Implementação Híbrida

- O código fonte é convertido em uma representação intermediária
  - Ex.: formato bytecode na linguagem Java
  - Ex. 2: compilação just-in-time na linguagem Java



# Tendências

- As técnicas de programação evoluíram com o passar do tempo
  - Desenvolvedores se tornaram mais experientes
  - Boas práticas foram se consolidando
  - Vantagens e limitações das linguagens pioneiras foram compreendidas

# Tendências (2)

- Linguagens mais modernas foram propostas como possíveis substitutas às mais antigas
  - A seguir...
- Essas linguagens desafiantes conseguirão destronar as consolidadas?
  - Só o tempo dirá…
  - Algumas estão se tornando bem populares

# Tendências (3)

- Principais tendências:
  - Objective C → Swift
  - Java → Kotlin
  - Javascript → TypeScript
  - C++ → Go, Rust e Carbon
  - C → Zig

# Exercício

- Análise a linguagem escolhida quanto aos critérios:
  - Legibilidade, escritabilidade, confiabilidade, custo
  - Método de implementação (compilada, interpretada, híbrida)
  - Indique o ponto forte da linguagem na sua opinião. Indique o ponto fraco. Justifique sua resposta