CEUB

EDUCAÇÃO SUPERIOR

PARADIGMAS DE LINGUAGENS DE PROGRAMAÇÃO

BCC 2024-1 (MÓDULO 1)

ceub.br

AULA 01 - 28/2/2024
PLANO DE ENSINO

PROFESSOR

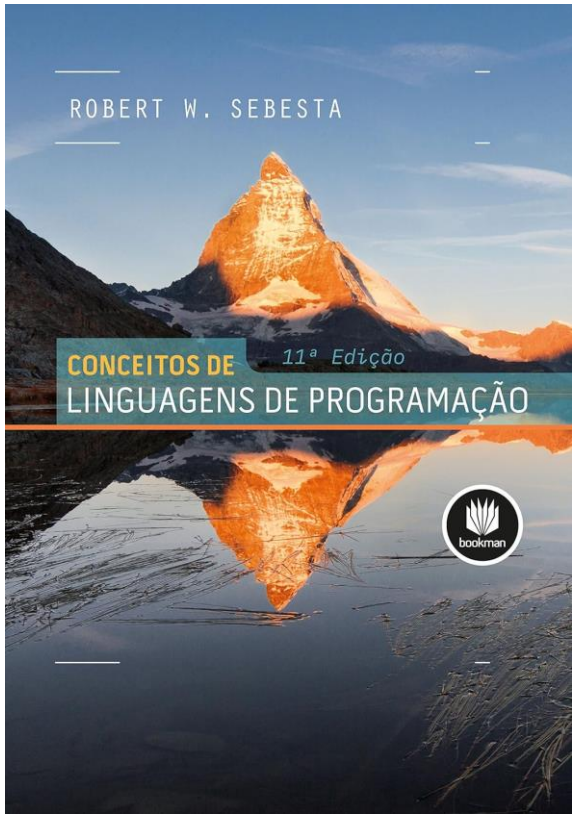
Prof. **Thiago** Neves

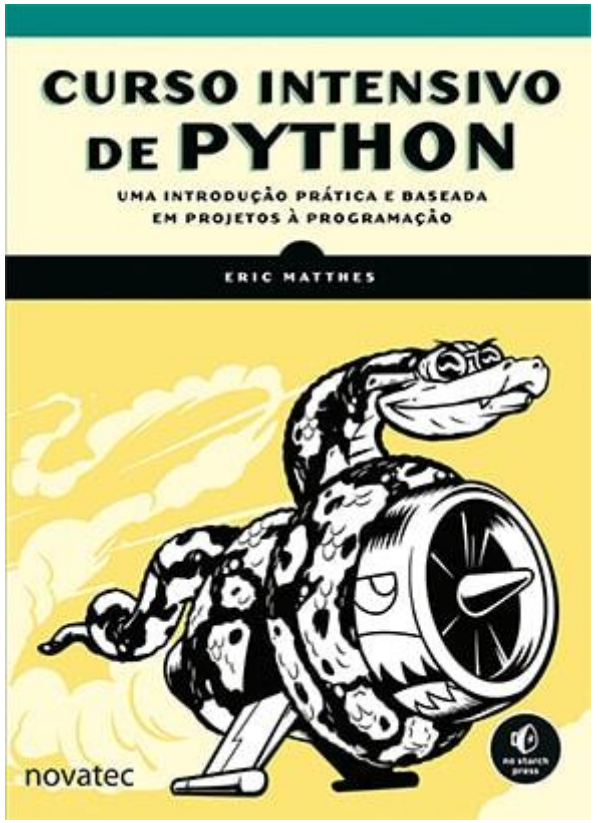
- Bacharel em Ciência da Computação
- MSc. Informática – Redes de Computadores
- 12 anos de experiência na área de TIC
- thiago.neves@ceub.edu.br

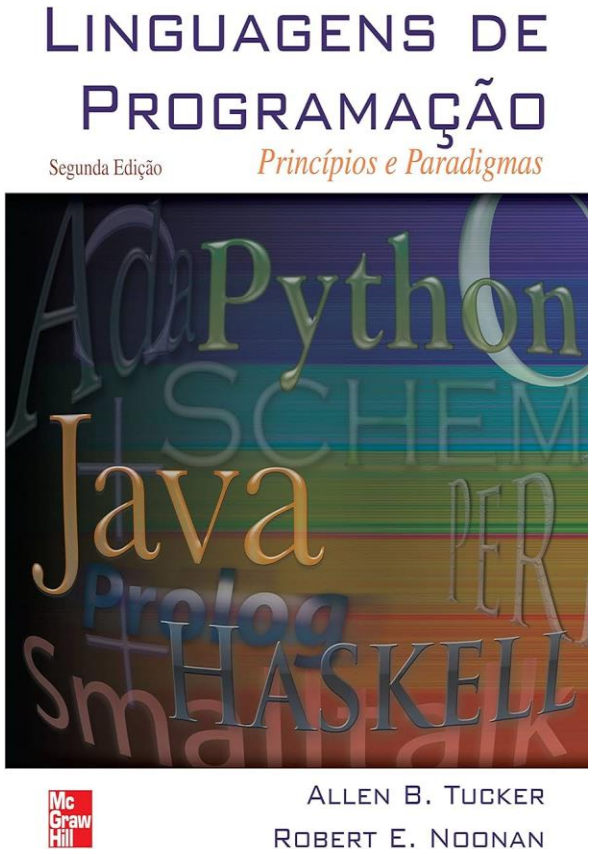
EMENTA DA DISCIPLINA

- Conceitos básicos;
- Metalinguagens;
- Tipos, variáveis, visibilidade, tempo de vida, comandos, estruturas de controle, unidades de programa, gerenciamento de memória, aspectos de implementação e outras construções das linguagens de programação;
- Linguagens imperativas;
- Linguagens orientadas a objetos;
- Linguagens funcionais;
- Linguagens lógicas;
- Linguagens concorrentes;
- Projeto de linguagens: Características de uma boa linguagem de programação;
- Sintaxe; Semântica e Seleção de linguagens para aplicações específicas.

BIBLIOGRAFIA







AVALIAÇÃO 1/5

Avaliação

- Baseado no Regimento Geral do UniCEUB, Título IV, Capítulo I, Subseção V, a apuração do rendimento escolar será feita abrangendo os aspectos de assiduidade e aproveitamento, eliminatórios por si mesmos.

- A assiduidade será verificada pela frequência às aulas e às atividades da disciplina e o aproveitamento será aferido mediante a exigência da assimilação progressiva dos conhecimentos ministrados.

- O aproveitamento nos estudos será traduzido pelas seguintes menções: SS (Superior); MS (Médio Superior); MM (Médio); MI (Médio Inferior); II (Inferior); SR (Sem Rendimento) e RF (Reprovado por Falta) e a menção final representará o julgamento global do aproveitamento nos estudos dentro de suas respectivas proporções de equivalência.

AVALIAÇÃO 2/5

Frequência

- Como a disciplina possuiu 75 h/a, o aluno que obtiver mais de 18 h/a de ausência estará reprovado por falta (RF) ainda que apresente menção final apta para aprovação. Cada encontro equivale a 3 (três) presenças.

- O acesso a cada Unidade de Aprendizagem (UA) representa 3 (três) horas de presença na disciplina, totalizando assim 15 horas. Na ausência de acesso(s) às UA, o estudante receberá falta na mesma proporção.

- Os sábados são considerados dias letivos pelo UniCEUB, dessa forma poderão ser utilizados para a antecipação ou reposição de aulas, apresentação de trabalhos ou aulas práticas.

AVALIAÇÃO 3/5

Critérios de Avaliação

- A menção final será baseada na média simples das 3 (três) maiores notas dentro de 4 (quatro) avaliações parciais.
- Não haverá avaliações substitutivas ou de reposição em nenhuma hipótese.
- Casos excepcionais deverão ser tratados com o professor.

  - Menção 1: Prova escrita com o conteúdo do Módulo 1.
  - Menção 2: Prova escrita com o conteúdo do Módulo 2.
  - Menção 3: Prova escrita com o conteúdo do Módulo 3.
  - Menção 4: Trabalho prático de programação.

AVALIAÇÃO 4/5

Avaliação por meio das Unidades de Aprendizagem (UA)

- As Unidades de Aprendizagem - UA´s são previamente avaliadas e selecionadas pelo(a) docente, constando uma trilha de conhecimento onde os estudantes seguirão um roteiro permeando pelo assunto apresentado, de maneira que tenham a capacidade de descrever, compreender e aplicar o conhecimento discutido na UA.

- Por sua vez, a avaliação do conteúdo das Unidades de Aprendizagem será realizada, ao longo do semestre, pelo(a) docente responsável pela disciplina, de forma a integrar o conteúdo das Unidades de Aprendizagem no ementário e nas suas avaliações. Não haverá lançamento de uma menção específica da UA.

AVALIAÇÃO 5/5

Da frequência das UA´s (Unidades de Aprendizagem)

- A frequência será lançada pelo setor institucional Lab Class e registrada por meio do acesso e realização das atividades previstas em cada Unidade de Aprendizagem, até dia 4 de junho de 2024.

- Para cada Unidade acessada, serão computadas 3 horas-aula, e para que o(a) estudante tenha o cômputo total da carga-horária do objeto educacional, 15 ou 30 horas, a depender da carga-horária da disciplina, deverá realizar todas as atividades previstas nas UA ´s até o dia 4 de junho de 2024.

PROGRAMAÇÃO DAS AULAS

Módulo 1

- Aula 1: 28/2/2024
- Aula 2: 06/3/2024
- Aula 3: 13/3/2024
- Aula 4: 20/3/2024
- Aula 5: 27/3/2024
- Prova 1: 03/4/2024

Módulo 2

- Aula 6: 10/4/2024
- Aula 7: 17/4/2024
- Aula 8: 24/4/2024
- Aula 9: 08/5/2024
- Prova 2: 15/5/2024

Módulo 3

- Aula 10: 22/5/2024
- Aula 11: 29/5/2024
- Aula 12: 05/6/2024
- Aula 13: 12/6/2024
- Prova 3: 19/6/2024

Trabalho prático

- Apresentação: 26/6/2024

PARADIGMAS DE LINGUAGENS DE PROGRAMAÇÃO

O que é um paradigma?

Modelo que serve de padrão; normas: paradigmas políticos, sociais.

Norma que já está estabelecida: a escola segue paradigmas tradicionais.

Paradigmas é sinônimo de: padrão, exemplo, normas, modelos.

Paradigmas de Linguagens de programação?

Padrões/Exemplos/Normas/Modelos de Linguagens de Programação.

PARADIGMAS DE LINGUAGENS DE PROGRAMAÇÃO

CEUB
EDUCAÇÃO SUPERIOR

TIOBE Index for February 2024 - https://www.tiobe.com/tiobe-index/

Feb 2024	Feb 2023	Change	Programming Language	Ratings	Change
1	1		Python	15.16%	-0.32%
2	2		C	10.97%	-4.41%
3	3		C++	10.53%	-3.40%
4	4		Java	8.88%	-4.33%
5	5		C#	7.53%	+1.15%
6	7	↑	JavaScript	3.17%	+0.64%
7	8	↑	SQL	1.82%	-0.30%
8	11	↑	Go	1.73%	+0.61%
9	6	↓	Visual Basic	1.52%	-2.62%
10	10		PHP	1.51%	+0.21%
11	24	⇈	Fortran	1.40%	+0.82%
12	14	↑	Delphi/Object Pascal	1.40%	+0.45%
13	13		MATLAB	1.26%	+0.27%
14	9	⇊	Assembly language	1.19%	-0.19%
15	18	↑	Scratch	1.18%	+0.42%

QUESTÕES – AULA 1

Q1 Que argumentos você pode dar **a favor da ideia de uma única linguagem** para todos os domínios de programação?

Q2 Que argumentos você pode dar **contra a ideia de uma única linguagem** para todos os domínios de programação?

QUESTÕES – AULA 1

RESPOSTA Q1

- Isso reduziria dramaticamente os custos de treinamento em programação e a compra e manutenção de compiladores;
- Simplificaria a contratação de programadores.

RESPOSTA Q2:

- A linguagem necessariamente seria extensa e complexa;
- Os compiladores seriam caros e dispendiosos de manter;
- Provavelmente a linguagem não seria muito boa para nenhum domínio de programação, nem em eficiência de compilador nem na eficiência do código gerado. Mais importante ainda, não seria fácil de usar, porque, independentemente da área de aplicação, a linguagem incluiria muitos recursos e construções desnecessárias e confusas (destinadas a outras áreas de aplicação).
- Diferentes usuários aprenderiam diferentes subconjuntos, tornando a manutenção difícil.

AULA 02 - 06/3/2024

PRELIMINARES

EVOLUÇÃO DAS LINGUAGENS

Tópicos

- Motivos para estudar conceitos de linguagens de programação
- Domínios de programação
- Critérios de avaliação de linguagens
- Influências no design de linguagens
- Categorias de linguagens
- Compromissos de design de linguagem
- Métodos de implementação
- Ambientes de programação

Motivos para estudar conceitos de linguagens de programação

- Aumento da capacidade de expressar ideias
- Melhoria no embasamento para escolher linguagens adequados
- Aumento da habilidade de aprender novas linguagens
- Melhor compreensão da importância da implementação
- Melhor uso das linguagens já conhecidas
- Avanço geral da computação
  - A melhor linguagem geralmente é escolhida?
  - EX: FORTRAM vs ALGOL 60.

Domínios de programação

Aplicações científicas
- Grandes quantidades de cálculos em ponto flutuante; uso de matrizes
- Fortran

Aplicações comerciais
- Produção de relatórios, uso de números decimais e caracteres
- COBOL

Inteligência artificial
- Manipulação de símbolos em vez de números; uso de listas encadeadas
- LISP, PROLOG

Programação de sistemas
- Necessidade de eficiência devido ao uso contínuo
- C

Software Web
- Coleção eclética de idiomas: marcação (por exemplo, HTML), script (por exemplo, PHP), de propósito geral (por exemplo, Java)

Qual a linguagem é utilizada para Natural Language processing (NLP)?

Critérios de avaliação de linguagens

- Readability (Legibilidade): a facilidade com que os programas podem ser lidos e compreendidos
- Writability (Facilidade de escrita): a facilidade com que uma linguagem pode ser usada para criar programas
- Reliability (Confiabilidade): conformidade com as especificações (ou seja, execução conforme suas especificações)
- Cost: o custo total final

Critérios de avaliação: Readability

Simplicidade geral
- Um conjunto gerenciável de características e construções
- Multiplicidade mínima de características.
- Sobrecarga mínima de operadores

Ortogonalidade
- Um conjunto relativamente pequeno de construções primitivas pode ser combinado de um número relativamente pequeno de maneiras. EX: Incremento de variáveis.
- Toda combinação possível é legal

Tipos de dados
- Tipos de dados pré-definidos adequados

Considerações de sintaxe
- Formas de identificadores: composição flexível
- Palavras especiais e métodos de formação de declarações compostas
- Forma e significado: construções autoexplicativas, palavras-chave significativas

Critérios de avaliação: Writability

Simplicidade e ortogonalidade

- Poucas construções, um pequeno número de primitivas, um conjunto pequeno de regras para combiná-las

Expressividade

- Um conjunto de maneiras relativamente convenientes de especificar operações
- Força e número de operadores e funções pré-definidas

Critérios de avaliação: Reliability

Verificação de tipo

- Teste para erros de tipo

Tratamento de exceção

- Interceptar erros em tempo de execução e tomar medidas corretivas

Aliasing

- Presença de dois ou mais métodos distintos de referenciamento para o mesmo local de memória

Legibilidade e facilidade de escrita

- Um idioma que não suporta maneiras "naturais" de expressar um algoritmo exigirá o uso de abordagens "não naturais", e consequentemente, reduzirá a confiabilidade.

Critérios de avaliação: Cost

- Treinamento de programadores para usar a linguagem
- Escrita de programas (proximidade com aplicações específicas)
- Compilação de programas
- Execução de programas
- Sistema de implementação de linguagem: disponibilidade de compiladores gratuitos
- Confiabilidade: baixa confiabilidade leva a custos elevados
- Manutenção de programas

Critérios de avaliação: Others

Portabilidade

- A facilidade com que os programas podem ser movidos de uma implementação para outra

Generalidade

- A aplicabilidade a uma ampla gama de aplicações

Bem-definido

- A completude e precisão da definição oficial da linguagem

Influências no design de linguagens

Arquitetura de Computadores

- As linguagens são desenvolvidas em torno da arquitetura de computadores predominante, conhecida como arquitetura de von Neumann

Metodologias de Design de Programas

- Novas metodologias de desenvolvimento de software (por exemplo, desenvolvimento de software orientado a objetos) levaram a novos paradigmas de programação e, por extensão, a novas linguagens de programação

Influência da Arquitetura de Computadores

Arquitetura de computador bem conhecida: Von Neumann

Linguagens imperativas, mais dominantes, devido aos computadores Von Neumann

- Dados e programas armazenados na memória
- A memória é separada da CPU
- Instruções e dados são transmitidos da memória para a CPU
  - Base para linguagens imperativas
  - Variáveis modelam células de memória
  - Instruções de atribuição modelam a transmissão
  - Iteração é eficiente

The von Neumann Architecture

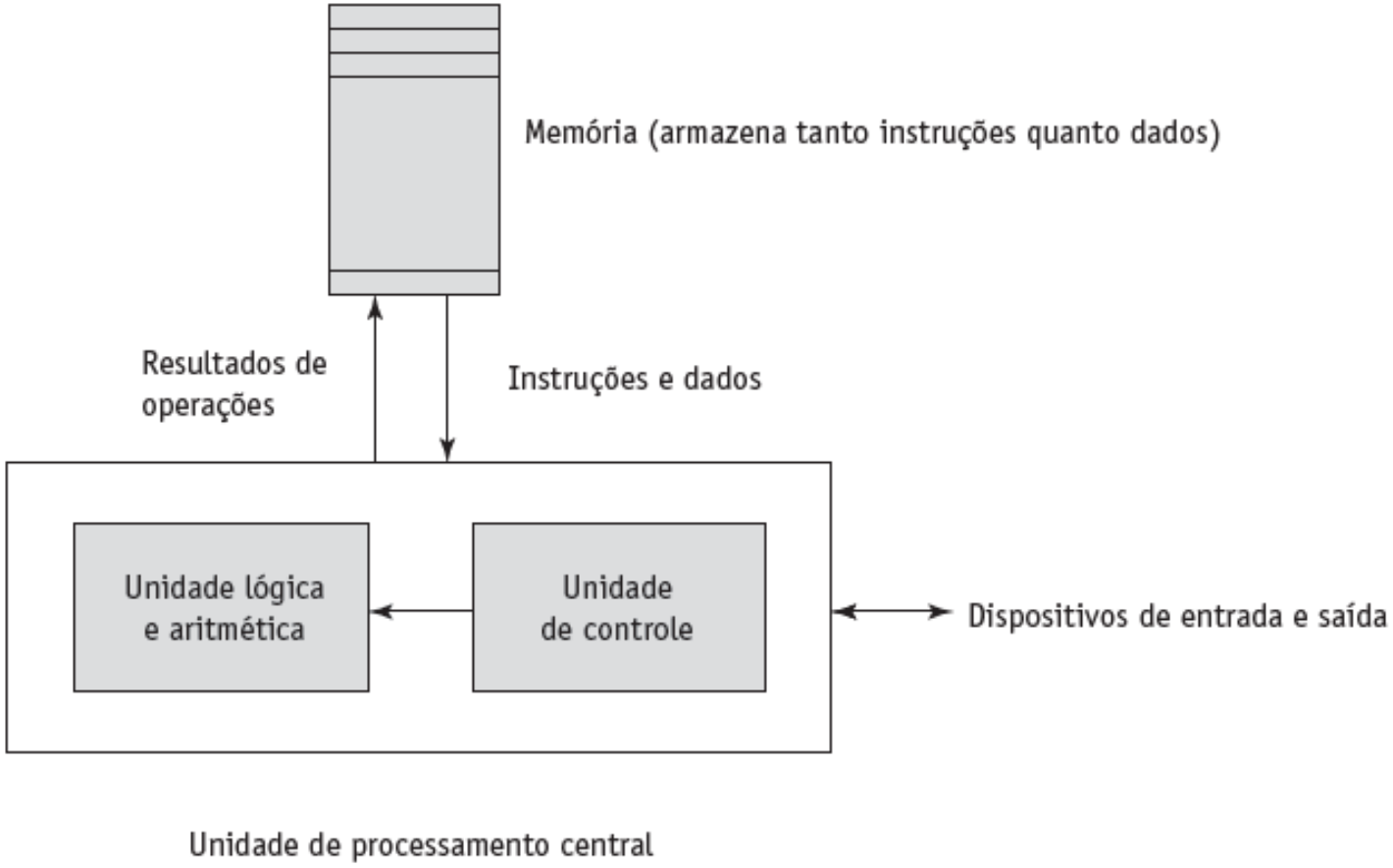


FIGURA 1.1
A arquitetura de computadores von Neumann.

The von Neumann Architecture

- Fetch-execute-cycle (on a von Neumann architecture computer)

```
initialize the program counter
repeat forever
  fetch the instruction pointed by the counter
  increment the counter
  decode the instruction
  execute the instruction
end repeat
```

Influências das Metodologias de Programação

- Década de 1950 e início dos anos 1960: Aplicações simples;
  - preocupação com eficiência da máquina
- Final dos anos 1960: Eficiência humana tornou-se importante;
  - legibilidade, melhores estruturas de controle
  - Programação estruturada
  - Projeto de cima para baixo e refinamento passo a passo
- Final dos anos 1970: Orientado a processo para orientado a dados
  - Abstração de dados
- Meio dos anos 1980: Programação orientada a objetos
  - Abstração de dados + herança + polimorfismo

Categorias de linguagens

Imperativo
- As principais características são variáveis, declarações de atribuição e iteração
- Incluem linguagens que suportam programação orientada a objetos
- Incluem linguagens de script
- Incluem as linguagens visuais
- Exemplos: C, Java, Perl, JavaScript, Visual BASIC .NET, C++

Funcional
- O principal meio de fazer cálculos é aplicar funções a parâmetros dados
- Exemplos: LISP, Scheme, ML, F#

Lógico
- Baseado em regras (as regras são especificadas sem ordem particular)
- Exemplo: Prolog

Híbrido de marcação/programação
- Linguagens de marcação estendidas para suportar alguma programação
- Exemplos: JSTL, XSLT

Language Design Trade-Offs

Reliability vs. cost of execution

- Exemplo: Java exige que todas as referências aos elementos de uma matriz sejam verificadas quanto ao índice adequado, o que leva a um aumento nos custos de execução

Readability vs. writability

- Exemplo: APL fornece muitos operadores poderosos (e um grande número de novos símbolos), permitindo que cálculos complexos sejam escritos em um programa compacto, mas ao custo de uma legibilidade ruim

Writability (flexibility) vs. reliability

- Exemplo: Os ponteiros em C++ são poderosos e muito flexíveis, mas são pouco confiáveis

Métodos de implementação

Compilação

- Programas são traduzidos para linguagem de máquina; inclui sistemas JIT
- Uso: Aplicações comerciais de grande porte

Interpretação Pura

- Programas são interpretados por outro programa conhecido como um interpretador
- Uso: Programas pequenos ou quando eficiência não é uma questão

Sistemas de Implementação Híbridos

- Um compromisso entre compiladores e interpretadores puros
- Uso: Sistemas pequenos e médios quando eficiência não é a principal preocupação

Layered View of Computer

- O sistema operacional e a implementação da linguagem são sobrepostos à interface de máquina de um computador.

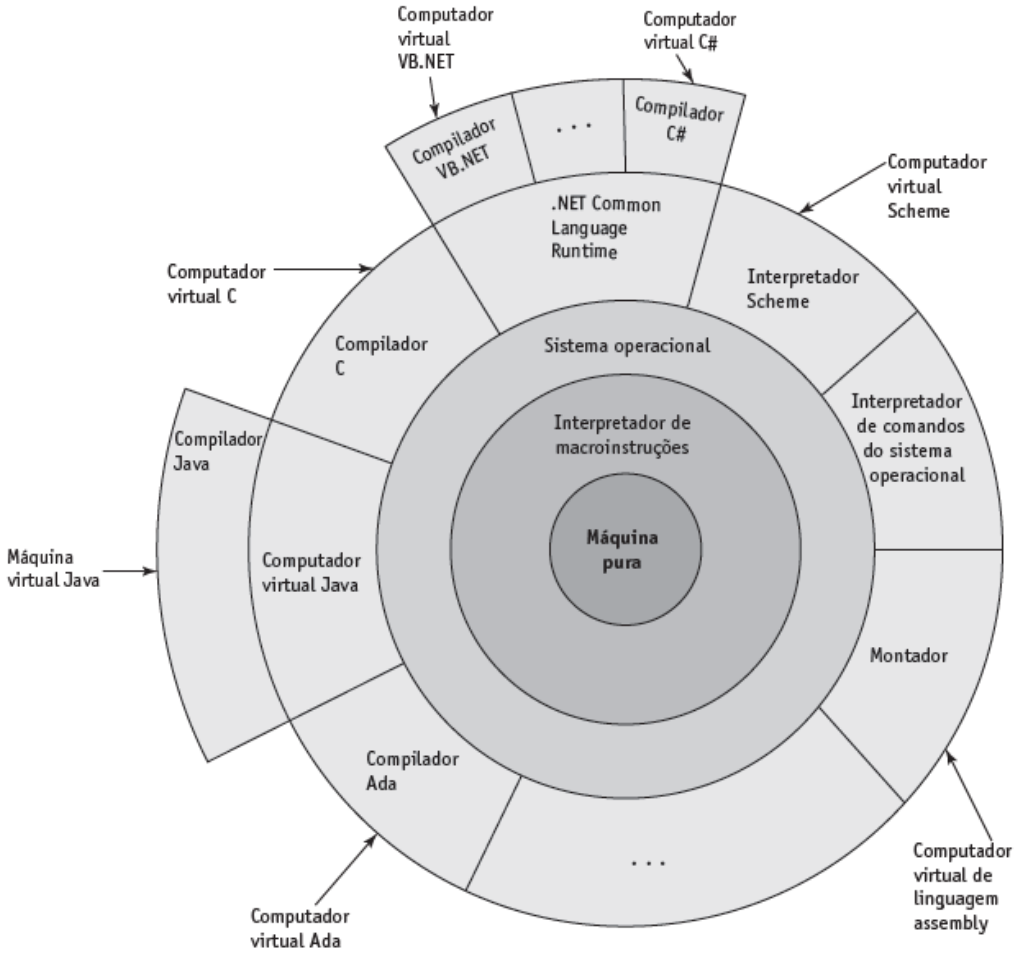


FIGURA 1.2
Interface em camadas de computadores virtuais, fornecida por um sistema de computação típico.

Compilation

Programa de alto nível (linguagem de origem) em código de máquina (linguagem de máquina)

- Tradução lenta, execução rápida

O processo de compilação possui várias fases:

- análise léxica: converte caracteres no programa-fonte em unidades léxicas
- análise sintática: transforma unidades léxicas em árvores de análise que representam a estrutura sintática do programa
- análise semântica: gera código intermediário
- geração de código: gera o código de máquina

The Compilation Process

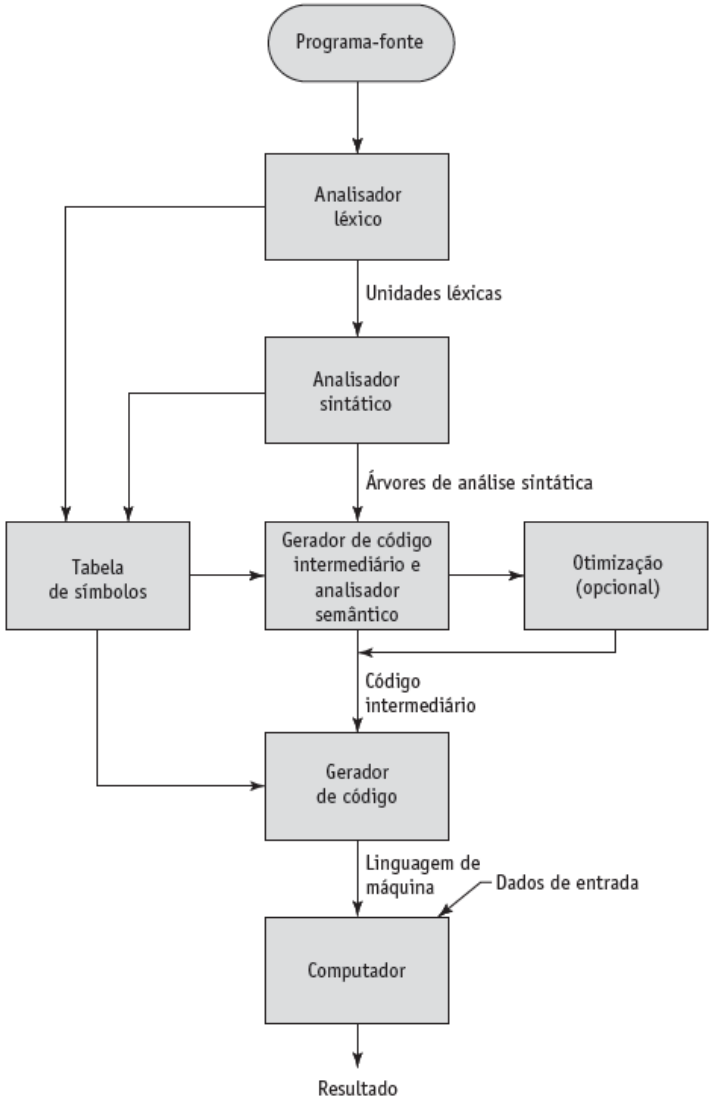


FIGURA 1.3
O processo de compilação.

Von Neumann Bottleneck

- A velocidade de conexão entre a memória de um computador e seu processador determina a velocidade do computador
- Frequentemente, as instruções do programa podem ser executadas muito mais rapidamente do que a velocidade da conexão; assim, a velocidade da conexão resulta em um gargalo
- Conhecido como gargalo de von Neumann; é o principal fator limitante na velocidade dos computadores

Pure Interpretation

- Sem tradução
- Implementação mais fácil de programas (erros de tempo de execução podem ser exibidos facilmente e imediatamente)
- Execução mais lenta (10 a 100 vezes mais lenta do que programas compilados)
- Frequentemente requer mais espaço
- Agora raro para linguagens de alto nível tradicionais
- Significativo retorno com algumas linguagens de script da Web (por exemplo, JavaScript, PHP)

Pure Interpretation Process

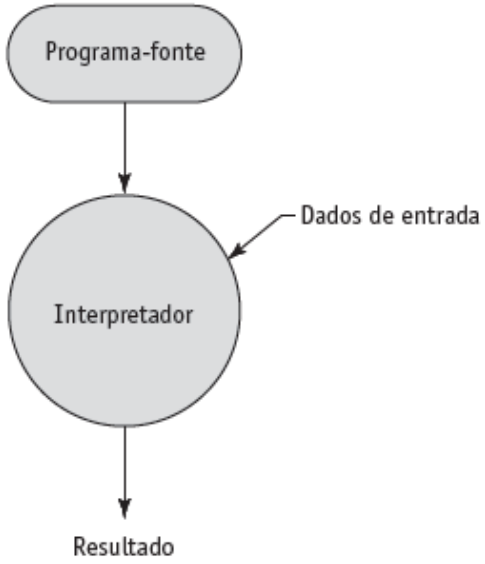


FIGURA 1.4
Interpretação pura.

Hybrid Implementation Systems

- Um compromisso entre compiladores e interpretadores puros
- Um programa em linguagem de alto nível é traduzido para uma linguagem intermediária que permite fácil interpretação
- Mais rápido do que interpretação pura
- Exemplos
  - Programas Perl são parcialmente compilados para detectar erros antes da interpretação
  - As implementações iniciais do Java eram híbridas; a forma intermediária, o byte code, fornece portabilidade para qualquer máquina que tenha um interpretador de byte code e um sistema em tempo de execução (juntos, esses são chamados de Máquina Virtual Java)

Hybrid Implementation Process

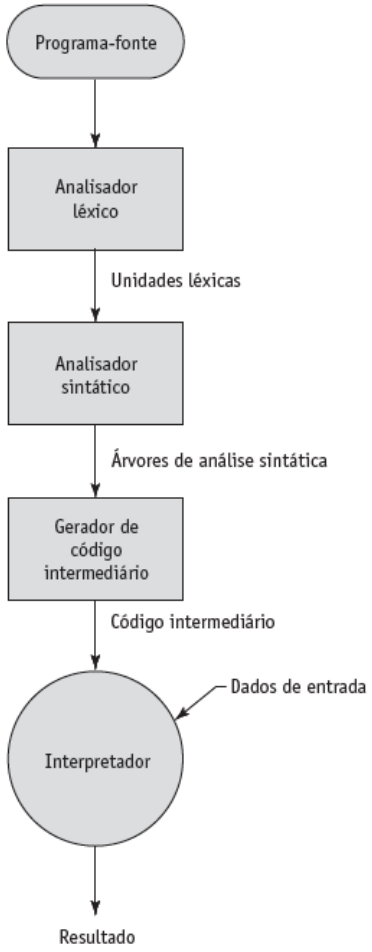


FIGURA 1.5
Sistema de implementação híbrido.

Just-in-Time Implementation Systems

- Inicialmente, traduzir programas para uma linguagem intermediária
- Então, compilar a linguagem intermediária dos subprogramas em código de máquina quando são chamados
- A versão em código de máquina é mantida para chamadas subsequentes
- Sistemas JIT são amplamente utilizados para programas Java
- Linguagens .NET são implementadas com um sistema JIT
- Essencialmente, sistemas JIT são compiladores atrasados

Preprocessors

- As macros (instruções) do pré-processador são comumente usadas para especificar que o código de outro arquivo deve ser incluído
- Um pré-processador processa um programa imediatamente antes de ser compilado para expandir macros de pré-processador incorporadas
- Um exemplo conhecido: pré-processador de C
  - expande #include, #define e macros similares

Ambientes de programação

- Uma coleção de ferramentas usadas no desenvolvimento de software
- UNIX
  - Um sistema operacional e coleção de ferramentas mais antigo
  - Atualmente frequentemente utilizado através de uma GUI (por exemplo, CDE, KDE ou GNOME) que roda sobre o UNIX
- Microsoft Visual Studio .NET
  - Um ambiente visual grande e complexo
  - Usado para construir aplicações Web e aplicações não Web em qualquer linguagem .NET
- NetBeans
  - Relacionado ao Visual Studio .NET

Resumo

O estudo das linguagens de programação é valioso por várias razões:

- Aumenta nossa capacidade de usar diferentes construções;
- Nos permite escolher linguagens de forma mais inteligente;
- Facilita a aprendizagem de novas linguagens.

Os critérios mais importantes para avaliar linguagens de programação incluem:

- Legibilidade, capacidade de escrita, confiabilidade, custo

As principais influências no design de linguagens foram a arquitetura de máquinas e metodologias de desenvolvimento de software.

Os principais métodos de implementação de linguagens de programação são: compilação, interpretação pura e implementação híbrida.

Evolução das Linguagens 1/3

1. Plankalkül de Zuse
2. Programação de Hardware Mínimo: Pseudocódigos
3. O IBM 704 e o Fortran
4. Programação Funcional: Lisp
5. O Primeiro Passo Rumo à Sofisticação: ALGOL 60
6. Automatização de Registros Empresariais: COBOL
7. Os Primórdios do Compartilhamento de Tempo: Basic

Evolução das Linguagens 2/3

8. Tudo para Todos: PL/I
9. Duas Linguagens Dinâmicas Iniciais: APL e SNOBOL
10. Os Primórdios da Abstração de Dados: SIMULA 67
11. Design Ortogonal: ALGOL 68
12. Alguns Descendentes Iniciais dos ALGOLs
13. Programação Baseada em Lógica: Prolog
14. O Maior Esforço de Projeto da História: Ada

Evolução das Linguagens 3/3

15. Programação Orientada a Objetos: Smalltalk
16. Combinação de Recursos Imperativos e Orientados a Objetos: C++
17. Uma Linguagem Orientada a Objetos Baseada em Imperativo: Java
18. Linguagens de Script
19. A Linguagem Principal da .NET: C#
20. Linguagens Híbridas de Marcação/Programação

1. Zuse's Plankalkül (Cálculo de programa)

- Projetado em 1945, mas não publicado até 1972
- Nunca implementado
- Estruturas de dados avançadas
  - ponto flutuante, matrizes, registros
- Invariantes

1. Zuse’s Plankalkül (Cálculo de programa)

Plankalkül Syntax

- An assignment statement to assign the expression A[4] + 1 to A[5]

```
      |   A + 1 => A
V     |   4         5          (subscripts)
S     |   1.n       1.n        (data types)
```

2. Minimal Hardware Programming: Pseudocodes

- O que estava errado em usar código de máquina?
  - Baixa legibilidade
  - Baixa modificabilidade
  - Codificação de expressões era tediosa
  - Deficiências da máquina – sem indexação ou ponto flutuante

2.1 Pseudocodes: Short Code

- Short Code developed by Mauchly in 1949 for BINAC computers
  - Expressions were coded, left to right
  - Example of operations:

```
01 -    06 abs value    1n (n+2)nd power
02 )    07 +            2n (n+2)nd root
03 =    08 pause        4n if <= n
04 /    09 (            58 print and tab
```

2.2 Pseudocodes: Speedcoding

- Speedcoding, desenvolvido por Backus em 1954 para o IBM 701
  - Pseudo operações para funções aritméticas e matemáticas
  - Branching condicional e incondicional
  - Registradores de auto-incremento para acesso a matrizes
  - Lento!
  - Apenas 700 palavras restantes para o programa do usuário

2.3 Pseudocodes: Related Systems

O Sistema de Compilação UNIVAC

- Desenvolvido por uma equipe liderada por Grace Hopper
- Pseudocódigo expandido em código de máquina

David J. Wheeler (Universidade de Cambridge)

- desenvolveu um método de uso de blocos de endereços relocáveis para resolver o problema de endereçamento absoluto

3. IBM 704 and Fortran

- Fortran 0: 1954 – não implementado
  - IBM mathematical FORmula TRANslating system
- Fortran I: 1957
  - Projetado para o novo IBM 704, que tinha registradores de índice e hardware de ponto flutuante
  - Isso levou à ideia de linguagens de programação compiladas, porque não havia lugar para esconder o custo da interpretação (nenhum software de ponto flutuante)
  - Ambiente de desenvolvimento
    - Os computadores eram pequenos e pouco confiáveis
    - As aplicações eram científicas
    - Não havia metodologia ou ferramentas de programação
    - A eficiência da máquina era a preocupação mais importante

3.1 Design Process of Fortran

- Impacto do ambiente no design do Fortran I
  - Não há necessidade de armazenamento dinâmico
  - Necessidade de bom tratamento de matrizes e laços de contagem
  - Sem tratamento de strings, aritmética decimal ou entrada/saída poderosa (para software empresarial)

3.2 Fortran I Overview

Primeira versão implementada do Fortran

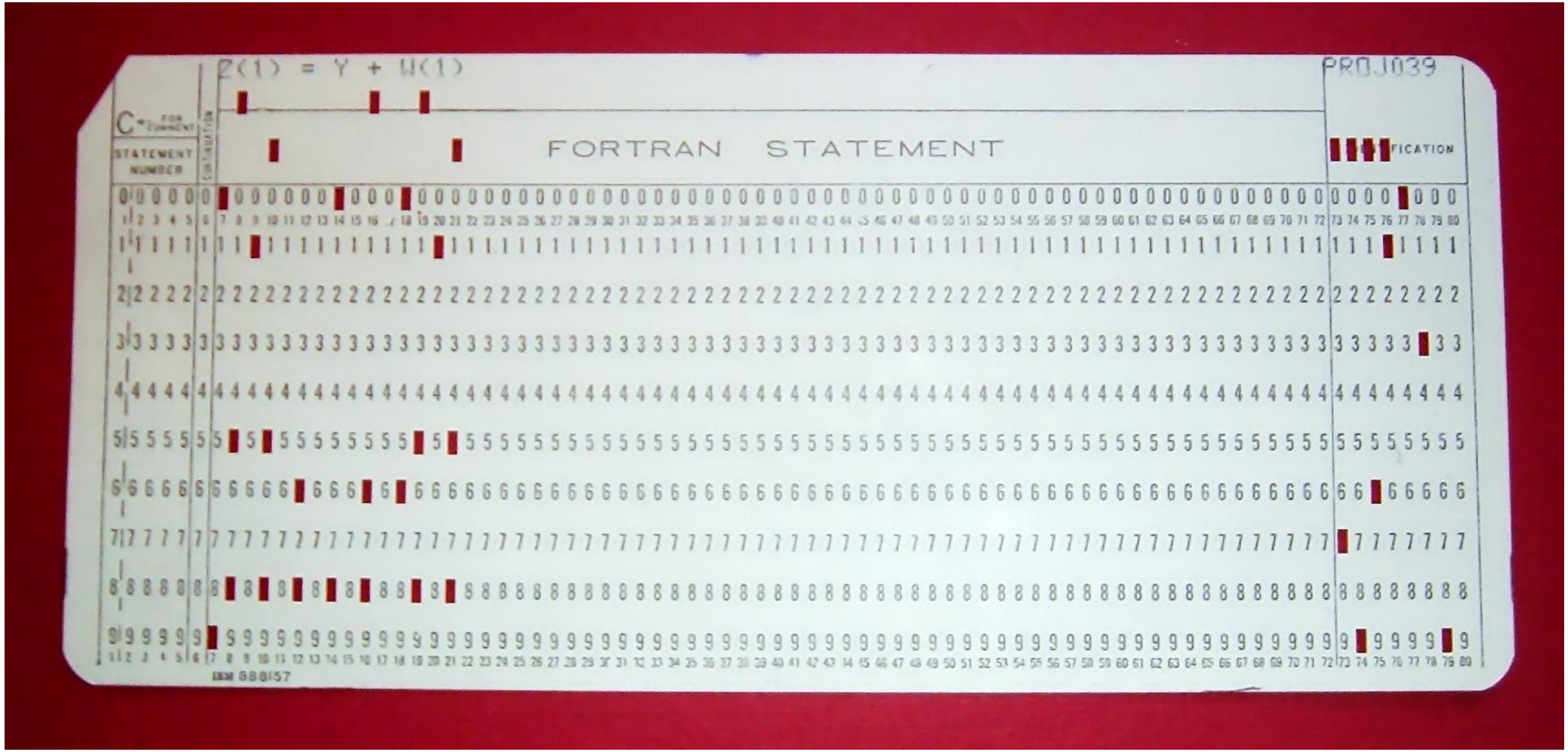
- Nomes podiam ter até seis caracteres
- Laço de contagem pós-teste (DO)
- Entrada/saída formatada
- Subprogramas definidos pelo usuário
- Declaração de seleção de três vias (IF aritmético)
- Sem declarações de tipagem de dados

3.3 Fortran I Overview (continued)

Primeira versão implementada do FORTRAN

- Sem compilação separada
- Compilador lançado em abril de 1957, após 18 anos de esforço de trabalho
- Programas com mais de 400 linhas raramente eram compilados corretamente, principalmente devido à baixa confiabilidade do 704
- O código era muito rápido
- Rapidamente tornou-se amplamente utilizado

3.3 Fortran I Overview (continued)



3.3 Fortran I Overview (continued)

3.4 Fortran II

Distribuído em 1958

- Compilação independente
- Corrigiu os bugs

3.5 Fortran IV

Evoluído durante 1960–62

- Declarações de tipo explícitas
- Declaração lógica de seleção
- Nomes de subprogramas poderiam ser parâmetros
- Padrão ANSI em 1966

3.6 Fortran 77

Tornou-se o novo padrão em 1978

- Manipulação de strings de caracteres
- Declaração lógica de controle de loop
- Declaração IF-THEN-ELSE

3.7 Fortran 90

As mudanças mais significativas em relação ao Fortran 77

- Módulos
- Matrizes dinâmicas
- Ponteiros
- Recursão
- Declaração CASE
- Verificação de tipo de parâmetro

3.8 Latest versions of Fortran

- Fortran 95 – adições relativamente menores, além de algumas exclusões
- Fortran 2003 – suporte para POO, ponteiros para procedimentos, interoperabilidade com C
- Fortran 2008 – blocos para escopos locais, co-matrizes, Do Concurrent

3.8 Latest versions of Fortran

```
! Programa de exemplo do Fortran 95
!  Entrada:   Um inteiro, List_Len, onde List_Len é menor do
!             que 100, seguido por valores inteiros List_Len
!  Saída:     O número de valores de entrada que são maiores
!             do que a média de todos os valores de entrada
Implicit none
Integer Dimension(99) :: dInt_List
Integer :: List_Len, Counter, Sum, Average, Result
Result= 0
Sum = 0
Read *, List_Len
If ((List_Len > 0) .AND. (List_Len < 100)) Then
! Lê os dados de entrada em um vetor e calcula sua soma
   Do Counter = 1, List_Len
      Read *, Int_List(Counter)
      Sum = Sum + Int_List(Counter)
   End Do
! Calcula a média
   Average = Sum / List_Len
! Conta os valores que são maiores do que a média
   Do Counter = 1, List_Len
      If (Int_List(Counter) > Average) Then
         Result = Result + 1
      End If
  End Do
! Imprimir o resultado
   Print *, 'Number of values > Average is:', Result
Else
   Print *, 'Error - list length value is not legal'
End If
End Program Example
```

3.9 Fortran Evaluation

- Compiladores altamente otimizados (todas as versões anteriores a 90)
- Tipos e armazenamento de todas as variáveis são fixados antes do tempo de execução
- Mudou drasticamente para sempre a forma como os computadores são utilizados

4. Functional Programming: Lisp 1958

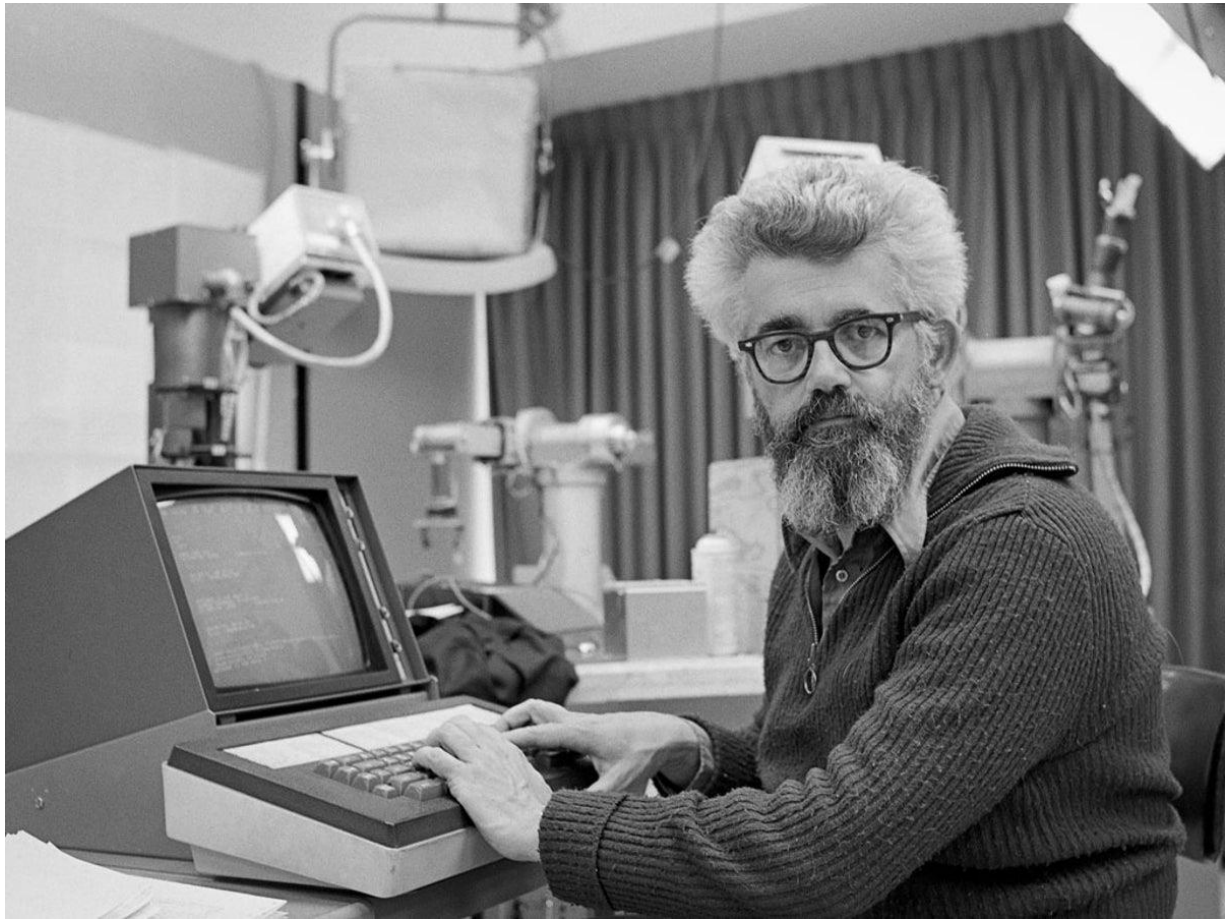
Linguagem de Processamento de LIStas

- Projetada no MIT por McCarthy

A pesquisa em IA precisava de uma linguagem para

- Processar dados em listas (em vez de matrizes)
- Cálculo simbólico (em vez de numérico)
- Apenas dois tipos de dados: átomos e listas
- Sintaxe baseada no cálculo lambda

4. Functional Programming: Lisp 1958



4. Representation of Two Lisp Lists

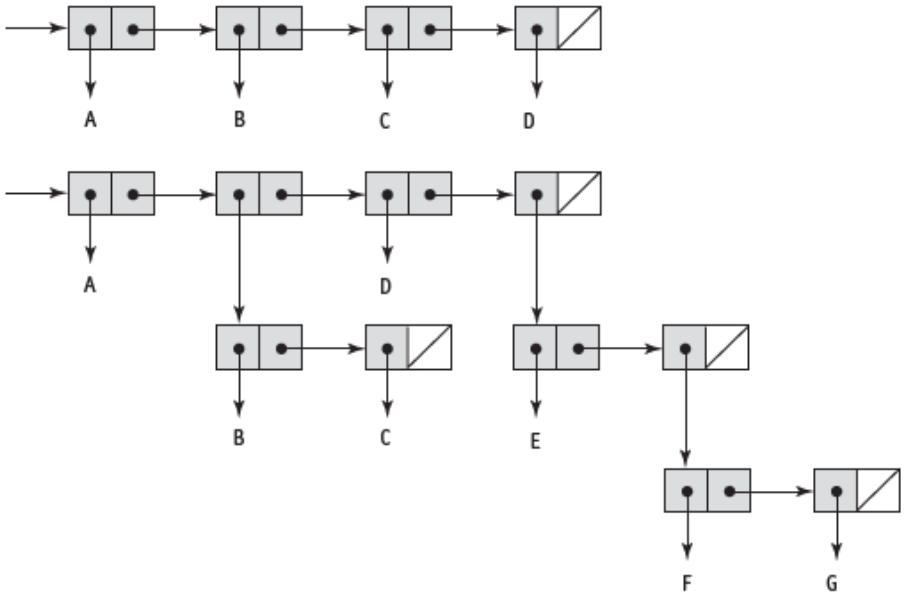


FIGURA 2.2
Representação interna de duas listas Lisp.

Representing the lists `(A B C D)`
and `(A (B C) D (E (F G)))`

4. Representation of Two Lisp Lists

```
; Função de exemplo em LISP
; O código a seguir define uma função de predicado em LISP
; que recebe duas listas como argumentos e retorna True
; se as duas listas forem iguais, e NIL (false) caso contrário
(DEFUN equal_lists (lis1 lis2)
(COND
((ATOM lis1) (EQ lis1 lis2))
((ATOM lis2) NIL)
((equal_lists (CAR lis1) (CAR lis2))
(equal_lists (CDR lis1) (CDR lis2)))
(T NIL)
)
)
```

4. Lisp Evaluation

- Pioneirismo na programação funcional
  - Sem necessidade de variáveis ou atribuições
  - Controle via recursão e expressões condicionais
- Ainda a linguagem dominante para IA
- Common Lisp e Scheme são dialetos contemporâneos de Lisp
- ML, Haskell e F# também são linguagens de programação funcional, mas utilizam uma sintaxe muito diferente

4. Scheme

- Desenvolvido no MIT no meio da década de 1970
- Pequeno
- Uso extensivo de escopo estático
- Funções como entidades de primeira classe
- Sintaxe simples (e tamanho pequeno) tornam-no ideal para aplicações educacionais

4. Common Lisp

- Um esforço para combinar características de vários dialetos de Lisp em uma única linguagem
- Grande, complexo, usado na indústria para algumas aplicações de grande porte

5. ALGOL 60

Ambiente de desenvolvimento

- O FORTRAN havia (mal) chegado para IBM 70x
- Muitas outras linguagens estavam sendo desenvolvidas, todas para máquinas específicas
- Nenhuma linguagem portátil; todas eram dependentes da máquina
- Nenhuma linguagem universal para comunicar algoritmos

ALGOL 60 foi o resultado de esforços para projetar uma linguagem universal

5.1 Early Design Process

ACM e GAMM se reuniram durante quatro dias para o projeto (de 27 de maio a 1 de junho de 1958)

Objetivos da linguagem

- Próxima da notação matemática
- Boa para descrever algoritmos
- Deve ser traduzível para código de máquina

5.2 ALGOL 58

- O conceito de tipo foi formalizado
- Nomes poderiam ter qualquer comprimento
- Matrizes poderiam ter qualquer número de subscripts
- Parâmetros eram separados por modo (entrada e saída)
- Subscripts eram colocados entre colchetes
- Declarações compostas (início ... fim)
- Ponto e vírgula como separador de declarações
- O operador de atribuição era :=
- A estrutura if tinha uma cláusula else-if
- Sem E/S - "tornaria dependente da máquina"

5.3 ALGOL 58 Implementation

- Não pretendia ser implementado, mas variações dele foram (MAD, JOVIAL)
- Embora a IBM estivesse inicialmente entusiasmada, todo o suporte foi abandonado até meados de 1959

5.4 ALGOL 60 Overview

ALGOL 58 modificado em uma reunião de 6 dias em Paris

Novas características

- Estrutura de bloco (escopo local)
- Dois métodos de passagem de parâmetros
- Recursão de subprograma
- Matrizes dinâmicas de pilha
- Ainda sem E/S e sem manipulação de strings

5.5 ALGOL 60 Evaluation

Sucessos

- Foi a forma padrão de publicar algoritmos por mais de 20 anos
- Todas as linguagens imperativas subsequentes são baseadas nela
- Primeira linguagem independente de máquina
- Primeira linguagem cuja sintaxe foi formalmente definida (BNF)

5. ALGOL 60 Evaluation (continued)

Fracasso

- Nunca amplamente utilizado, especialmente nos EUA
- Motivos
  - Falta de E/S e o conjunto de caracteres tornavam os programas não portáteis
  - Muito flexível – difícil de implementar
  - Fortran muito estabelecido
  - Descrição formal de sintaxe
  - Falta de suporte da IBM

6. Computerizing Business Records: COBOL

Ambiente de desenvolvimento

- A UNIVAC estava começando a usar o FLOW-MATIC
- A Força Aérea dos Estados Unidos estava começando a usar o AIMACO
- A IBM estava desenvolvendo o COMTRAN

O UNIVAC I foi o primeiro computador comercial fabricado e comercializado nos Estados Unidos

6. COBOL Historical Background

- Baseado no FLOW-MATIC
- Recursos do FLOW-MATIC
  - Nomes de até 12 caracteres, com hifens embutidos
  - Nomes em inglês para operadores aritméticos (sem expressões aritméticas)
  - Dados e código eram completamente separados
  - A primeira palavra em cada declaração era um verbo

6. COBOL Design Process

- Primeira Reunião de Design (Pentágono) – Maio de 1959
- Objetivos de design
  - Deve parecer como um inglês simples
  - Deve ser fácil de usar, mesmo que isso signifique que será menos poderoso
  - Deve ampliar a base de usuários de computadores
- Não deve ser influenciado pelos problemas atuais dos compiladores
- Os membros do comitê de design eram todos de fabricantes de computadores e filiais do Departamento de Defesa
- Problemas de design: expressões aritméticas? subscripts? Brigas entre fabricantes

6. COBOL Evaluation

Contribuições

- Primeira facilidade de macro em uma linguagem de alto nível
- Estruturas de dados hierárquicas (registros)
- Declarações de seleção aninhadas
- Nomes longos (até 30 caracteres), com hifens
- Divisão de dados separada

6. COBOL: DoD Influence

- Primeira linguagem exigida pelo DoD
  - teria falhado sem o DoD
- Ainda a linguagem de aplicativos de negócios mais amplamente utilizada

7. BASIC

- Projetado por Kemeny & Kurtz em Dartmouth
- Objetivos de Design:
  - Fácil de aprender e usar para estudantes não científicos
  - Deve ser "agradável e amigável"
  - Rápido retorno para trabalhos de casa
  - Acesso gratuito e privado
- Tempo do usuário é mais importante do que tempo de computador
- Dialeto popular atual: Visual Basic
- Primeira linguagem amplamente utilizada com compartilhamento de tempo

8. Everything for Everybody: PL/I

- Projetado pela IBM e SHARE
- Situação computacional em 1964 (ponto de vista da IBM)
  - Computação científica
    - Computadores IBM 1620 e 7090
    - FORTRAN
    - Grupo de usuários SHARE
  - Computação empresarial
    - Computadores IBM 1401, 7080
    - COBOL
    - Grupo de usuários GUIDE

8.1 PL/I: Background

- Até 1963
  - Os usuários científicos começaram a precisar de E/S mais elaboradas, como o COBOL tinha; usuários empresariais começaram a precisar de ponto flutuante e matrizes para Sistemas de Informação Gerencial (MIS)
  - Parecia que muitas empresas começariam a precisar de dois tipos de computadores, linguagens e equipe de suporte – muito custoso
- A solução óbvia
  - Construir um novo computador para realizar ambos os tipos de aplicativos
  - Projetar uma nova linguagem para realizar ambos os tipos de aplicativos

8.2 PL/I: Design Process

- Projetado em cinco meses pelo Comitê 3 X 3
  - Três membros da IBM, três membros da SHARE
- Conceito inicial
  - Uma extensão do Fortran IV
- Inicialmente chamado de NPL (Nova Linguagem de Programação)
- Nome alterado para PL/I em 1965

8.3 PL/I: Evaluation

- Contribuições do PL/I
  - Primeira concorrência em nível de unidade
  - Primeiro tratamento de exceção
  - Recursão selecionável por switch
  - Primeiro tipo de dados ponteiro
  - Primeiras seções transversais de matrizes
- Preocupações
  - Muitos novos recursos foram mal projetados
  - Muito grande e muito complexo

9. Two Early Dynamic Languages: APL and SNOBOL

- Caracterizada por tipagem dinâmica e alocação dinâmica de armazenamento
- Variáveis não têm tipo definido
  - Uma variável adquire um tipo quando recebe um valor
- O armazenamento é alocado para uma variável quando ela recebe um valor

9.1 APL: A Programming Language

- Projetada como uma linguagem de descrição de hardware na IBM por Ken Iverson por volta de 1960
  - Altamente expressiva (muitos operadores, tanto para escalares quanto para arrays de várias dimensões)
  - Programas são muito difíceis de ler
- Ainda em uso; mudanças mínimas

9.2 SNOBOL

- Projetada como uma linguagem de manipulação de strings nos Laboratórios Bell por Farber, Griswold e Polensky em 1964
- Operadores poderosos para correspondência de padrões de strings
- Mais lenta do que outras linguagens alternativas (e portanto não mais usada para escrever editores)
- Ainda utilizada para certas tarefas de processamento de texto

10. Os Primórdios da Abstração de Dados: SIMULA 67

- Projetada principalmente para simulação de sistemas na Noruega por Nygaard e Dahl
- Baseada no ALGOL 60 e SIMULA I
- Principais contribuições
  - Corrotinas – um tipo de subprograma
  - Classes, objetos e herança

11. Orthogonal Design: ALGOL 68

- A partir do desenvolvimento contínuo do ALGOL 60, mas não sendo um superset dessa linguagem
- Fonte de várias novas ideias (embora a própria linguagem nunca tenha alcançado um uso generalizado)
- O design é baseado no conceito de ortogonalidade
  - Alguns conceitos básicos, além de alguns mecanismos de combinação

11.1 ALGOL 68 Evaluation

Contribuições

- Estruturas de dados definidas pelo usuário
- Tipos de referência
- Arrays dinâmicos (chamados de flex arrays)

Comentários

- Menos uso do que o ALGOL 60
- Teve forte influência em linguagens subsequentes, especialmente Pascal, C e Ada

11.2 Pascal - 1971

- Desenvolvida por Wirth (um ex-membro do comitê ALGOL 68)
- Projetada para ensinar programação estruturada
- Pequena, simples, nada realmente novo
- O maior impacto foi no ensino de programação
  - Desde meados da década de 1970 até o final da década de 1990, foi a linguagem mais amplamente usada para o ensino de programação

11.3 C - 1972

- Projetada para programação de sistemas (nos Laboratórios Bell por Dennis Ritchie)
- Evoluiu principalmente do BCLP e B, mas também do ALGOL 68
- Conjunto poderoso de operadores, mas verificação de tipo fraca
- Inicialmente difundida através do UNIX
- Embora projetada como uma linguagem de sistemas, tem sido usada em muitas áreas de aplicação

13. Programação Baseada em Lógica: Prolog 1972

- Desenvolvido por Comerauer e Roussel (Universidade de Aix-Marseille), com ajuda de Kowalski (Universidade de Edimburgo)
- Baseado em lógica formal
- Não procedimental
- Pode ser resumido como sendo um sistema de banco de dados inteligente que usa um processo de inferência para inferir a veracidade de consultas dadas
- Comparativamente ineficiente
- Poucas áreas de aplicação

13. Programação Baseada em Lógica: Prolog 1972

A base de dados de um programa Prolog consiste em dois tipos de sentenças: fatos e regras. Exemplos de sentenças factuais são:

```
mother(joanne, jake).
father(vern, joanne).
```

Essas sentenças afirmam que joanne é a mãe (mother) de jake, e que vern é o pai (father) de joanne.

Um exemplo de uma regra é

```
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

14. O Maior Esforço de Projeto da História: Ada

- Um enorme esforço de design, envolvendo centenas de pessoas, muito dinheiro e cerca de oito anos
- Sequência de requisitos (1975–1978)
  - (Espantalho, Homem de Madeira, Homem de Lata, Homem de Ferro, Homem de Aço)
- Nomeado Ada em homenagem a Augusta Ada Byron, a primeira programadora

14. Ada Evaluation

Contribuições

- Pacotes – suporte para abstração de dados
- Tratamento de exceção – elaborado
- Unidades de programa genéricas
- Concorrência – através do modelo de tarefas

Comentários

- Design competitivo
- Incluiu tudo o que se sabia na época sobre engenharia de software e design de linguagem
- Os primeiros compiladores eram muito difíceis; o primeiro compilador realmente utilizável veio quase cinco anos após a conclusão do design da linguagem

14. Ada 95

- Ada 95 (iniciada em 1988)
  - Suporte para POO através de derivação de tipo
  - Mecanismos de controle aprimorados para dados compartilhados
  - Novos recursos de concorrência
  - Bibliotecas mais flexíveis
- Ada 2005
  - Interfaces e interfaces de sincronização
- A popularidade sofreu porque o DoD não exige mais seu uso, mas também devido à popularidade do C++

15. Programação Orientada a Objetos: Smalltalk

- Desenvolvido no Xerox PARC, inicialmente por Alan Kay, mais tarde por Adele Goldberg
- Primeira implementação completa de uma linguagem orientada a objetos (abstração de dados, herança e vinculação dinâmica)
- Pioneirismo no design de interface gráfica do usuário
- Promoveu a POO

16. Combinação de Recursos Imperativos e Orientados a Objetos: C++

- Desenvolvido nos Laboratórios Bell por Stroustrup em 1980
- Evoluiu a partir de C e SIMULA 67
- Recursos para programação orientada a objetos, parcialmente retirados do SIMULA 67
- Uma linguagem grande e complexa, em parte porque suporta tanto programação procedural quanto orientada a objetos
- Cresceu rapidamente em popularidade, juntamente com a POO
- Padrão ANSI aprovado em novembro de 1997
- Versão da Microsoft: MC++
  - Propriedades, delegados, interfaces, sem herança múltipla

16.1 A Related OOP Language

- Objective-C (projetado por Brad Cox – início dos anos 1980)
  - C com suporte para POO baseado em Smalltalk
  - Utiliza a sintaxe de chamada de método do Smalltalk
  - Usado pela Apple para programas de sistemas

17. Uma Linguagem Orientada a Objetos Baseada em Imperativo: Java

Desenvolvido na Sun no início da década de 1990

- C e C++ não eram satisfatórios para dispositivos eletrônicos embutidos

Baseado em C++

- Significativamente simplificado (não inclui struct, union, enum, aritmética de ponteiros e metade das coerções de atribuição de C++)
- Suporta apenas POO
- Possui referências, mas não ponteiros
- Inclui suporte para applets e uma forma de concorrência

17.1 Java Evaluation

- Eliminou muitos recursos inseguros do C++
- Suporta concorrência
- Bibliotecas para applets, GUIs, acesso a banco de dados
- Portátil: conceito de Máquina Virtual Java, compiladores JIT
- Amplamente utilizado para programação na Web
- O uso aumentou mais rápido do que qualquer linguagem anterior
- Versão mais recente, 21, lançada em 2023

18. Linguagens de Script

Perl
- Projetado por Larry Wall – lançado pela primeira vez em 1987
- As variáveis são tipadas estaticamente, mas declaradas implicitamente
- Três espaços de nomes distintos, denotados pelo primeiro caractere do nome de uma variável
- Poderoso, mas um pouco perigoso
- Ganhou amplo uso para programação CGI na Web
- Também usado como substituto para a linguagem de administração de sistemas UNIX

JavaScript
- Começou na Netscape, mas depois se tornou uma joint venture da Netscape e da Sun Microsystems
- Uma linguagem de script incorporada em HTML do lado do cliente, frequentemente usada para criar documentos HTML dinâmicos
- Totalmente interpretado
- Relacionado ao Java apenas por meio de sintaxe similar

PHP
- PHP: Pré-processador de Hipertexto, projetado por Rasmus Lerdorf
- Uma linguagem de script incorporada em HTML do lado do servidor, frequentemente usada para processamento de formulários e acesso a banco de dados por meio da Web
- Totalmente interpretado

18. Scripting Languages for the Web

Python
- Uma linguagem de script interpretada orientada a objetos
- Verificação de tipo, mas tipagem dinâmica
- Usado para programação CGI e processamento de formulários
- Dinamicamente tipado, mas verificado por tipo
- Suporta listas, tuplas e hashes

Ruby
- Projetado no Japão por Yukihiro Matsumoto (também conhecido como "Matz")
- Começou como um substituto para Perl e Python
- Uma linguagem de script puramente orientada a objetos
- – Todos os dados são objetos
- A maioria dos operadores são implementados como métodos, que podem ser redefinidos pelo código do usuário
- Totalmente interpretado

18. Scripting Languages for the Web

Lua

- Uma linguagem de script interpretada orientada a objetos
- Verificação de tipo, mas tipagem dinâmica
- Usado para programação CGI e processamento de formulários
- Dinamicamente tipado, mas verificado por tipo
- Suporta listas, tuplas e hashes, todos com sua única estrutura de dados, a tabela
- Facilmente extensível

19. A Linguagem Principal da .NET: C#

- Parte da plataforma de desenvolvimento .NET (2000)
- Baseado em C++, Java e Delphi
- Inclui ponteiros, delegados, propriedades, tipos de enumeração, um tipo limitado de digitação dinâmica e tipos anônimos
- Está evoluindo rapidamente

20. Linguagens Híbridas de Marcação/Programação

- XSLT
  - Linguagem de Transformação de Folhas de Estilo Extensível (XSTL): transforma documentos XML para exibição
  - Construções de programação (por exemplo, laços)

- JSP
  - Páginas de Servidor Java: uma coleção de tecnologias para suportar documentos Web dinâmicos
  - JSTL, uma biblioteca JSP, inclui construções de programação na forma de elementos HTML

Genealogy of Common Languages

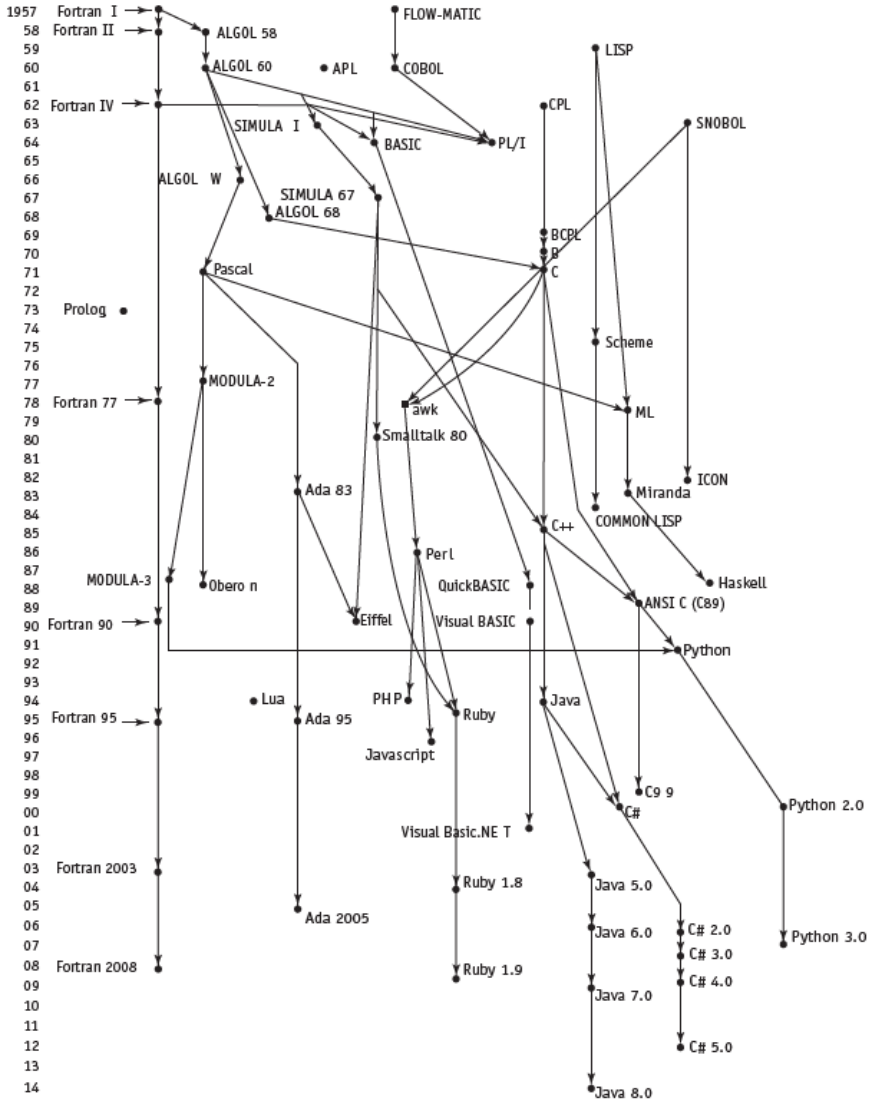


FIGURA 2.1
Genealogia das principais linguagens de programação de alto nível.

Resumo

- Desenvolvimento, ambiente de desenvolvimento e avaliação de um número de importantes linguagens de programação
- Perspectiva sobre questões atuais no design de linguagens

QUESTÕES – AULA 2

Q1. Descreva a maior motivação da IBM para desenvolver PL/I.

Q2. Quais são os argumentos a favor e contra a ideia de uma linguagem sem tipos?

Q3. Dê duas razões para que a interpretação pura seja um método de implementação aceitável para diversas das linguagens de scripting recentes.

QUESTÕES – AULA 2

Resposta Q1.

A principal motivação para o desenvolvimento do PL/I foi fornecer uma única ferramenta para centros de computação que precisam suportar tanto aplicações científicas quanto comerciais. A IBM acreditava que as necessidades das duas classes de aplicações estavam se fundindo, pelo menos em algum grau. Eles sentiram que a solução mais simples para um provedor de sistemas, tanto de hardware quanto de software, era fornecer um único sistema de hardware executando uma única linguagem de programação que servisse tanto aplicações científicas quanto comerciais.

Resposta Q2.

O argumento a favor das linguagens sem tipo é sua grande flexibilidade para o programador. Literalmente qualquer local de armazenamento pode ser usado para armazenar qualquer tipo de valor. Isso é útil para linguagens muito de baixo nível usadas para programação de sistemas. A desvantagem é que a verificação de tipo é impossível, então é inteiramente responsabilidade do programador garantir que as expressões e atribuições estejam corretas.

Resposta Q3.

Uma situação na qual a interpretação pura é aceitável para linguagens de script é quando a quantidade de computação é pequena, para a qual o tempo de processamento será negligenciável. Outra situação é quando a quantidade de computação é relativamente pequena e é feita em um ambiente interativo, onde o processador frequentemente está ocioso devido à lenta velocidade das interações humanas.

AULA 03 - 13/3/2024
Descrição da sintaxe e da semântica

Tópicos

- Introdução
- O Problema Geral de Descrever a Sintaxe
- Métodos Formais de Descrever a Sintaxe
- Gramáticas de Atributos

Introdução

- Sintaxe: a forma ou estrutura das expressões, declarações e unidades de programa.
- Semântica: o significado das expressões, declarações e unidades de programa.
  - EX: while (expressão) sentença
- Sintaxe e semântica fornecem a definição de uma linguagem.
  - Usuários de uma definição de linguagem.
    - Outros designers de linguagem.
    - Implementadores.
    - Programadores (os usuários da linguagem).

O Problema Geral de Descrever Sintaxe: Terminologia

- Uma sentença é uma sequência de caracteres sobre um determinado alfabeto.
- Uma linguagem é um conjunto de sentenças.
- Um lexema é a unidade sintática de nível mais baixo de uma linguagem (por exemplo, *, soma, iniciar).
- Um token é uma categoria de lexemas (por exemplo, identificador).

index = 2 * count + 17;

Lexemas	Tokens
index	identificador
=	sinal de igualdade
2	literal inteiro
*	operador de multiplicação
count	identificador
+	operador de adição
17	Literal inteiro
;	ponto e vígula

Definição Formal das Linguagens

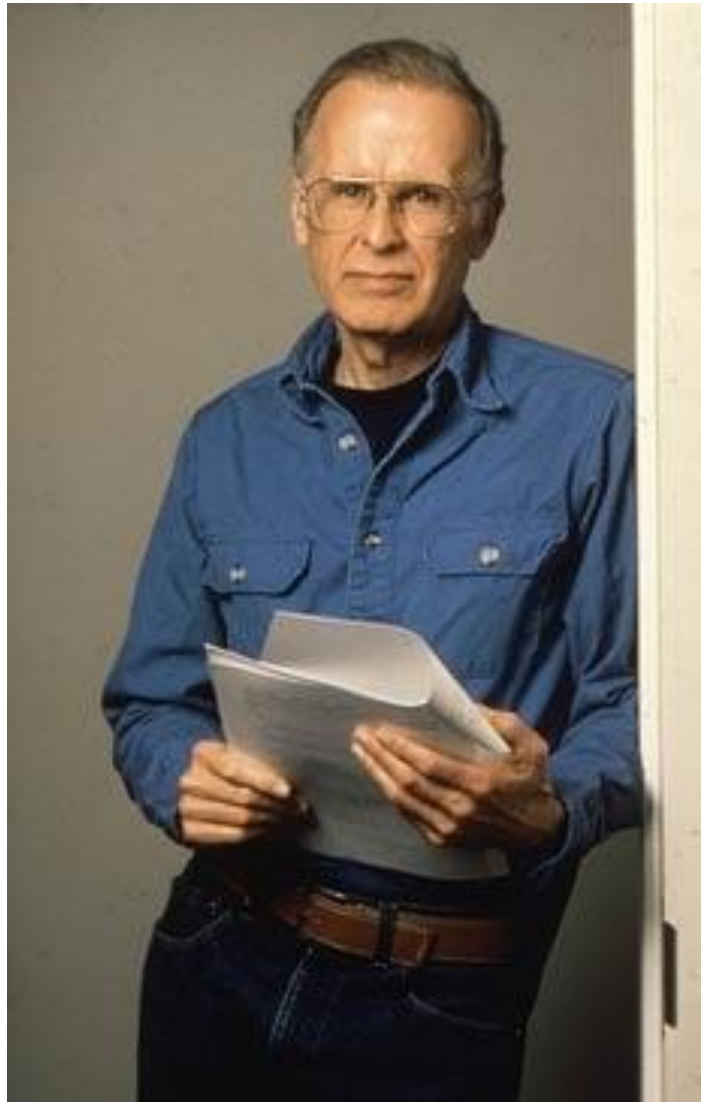
- Reconhecedores
  - Um dispositivo de reconhecimento lê sequências de entrada sobre o alfabeto da linguagem e decide se as sequências de entrada pertencem à linguagem.
  - Exemplo: parte de análise sintática de um compilador

- Geradores
  - Um dispositivo que gera sentenças de uma linguagem
  - Pode-se determinar se a sintaxe de uma determinada sentença é correta sintaticamente comparando-a com a estrutura do gerador.

BNF and Context-Free Grammars

- Context-Free Grammars
  - Desenvolvido por Noam Chomsky em meados da década de 1950
  - Geradores de linguagem, destinados a descrever a sintaxe de línguas naturais
  - Define uma classe de linguagens chamadas linguagens livres de contexto
- Backus-Naur Form (1959)
  - Inventado por John Backus para descrever a sintaxe do Algol 58
  - BNF é equivalente a gramáticas livres de contexto

John Backus



BNF Fundamentals

- Em BNF, abstrações são usadas para representar classes de estruturas sintáticas – elas agem como variáveis sintáticas (também chamadas de símbolos não-terminais ou apenas terminais).
- Terminais são lexemas ou tokens.
- Uma regra tem um lado esquerdo (LHS), que é um não-terminal, e um lado direito (RHS), que é uma sequência de terminais e/ou não-terminais.
- Não-terminais frequentemente são colocados entre colchetes angulares.
- EX:

  <assign> -> <var> = <expression>

  Total = subtotal1 + subtotal2

BNF Fundamentals (continued)

- Exemplos de regras BNF:
  <ident_list> → identificador | identificador, <ident_list>
  <if_stmt> → if <logic_expr> then <stmt>

- Gramática: um conjunto finito não vazio de regras.

- Um símbolo inicial é um elemento especial dos não-terminais de uma gramática.

BNF Rules

- Uma abstração (ou símbolo não-terminal) pode ter mais de um lado direito.

<stmt> → <single_stmt>| begin <stmt_list> end

Describing Lists

- Listas sintáticas são descritas usando recursão.

  <ident_list> → ident

  | ident, <ident_list>

- Uma derivação é uma aplicação repetida de regras, começando com o símbolo inicial e terminando com uma sentença (todos os símbolos terminais).

Um Exemplo de Gramática

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

Em Exemplo de Derivação

```
<program> => <stmts> => <stmt>
                     => <var> = <expr>
                     => a = <expr>
                     => a = <term> + <term>
                     => a = <var> + <term>
                     => a = b + <term>
                     => a = b + const
```

Questão 1

```
<assign> -> <id> = <expr>
<id> -> A | B | C
<expr> -> <id> + <expr>
            | <id> * <expr>
            | (<expr>)
            | <id>
```

Validar a sentença: A = B * (A + C)

Validar a sentença: A = A * (B + C * A))

Validar a sentença: B = C * (A * C + B)

Validar a sentença: A = A * (B + (C))

Questão 1

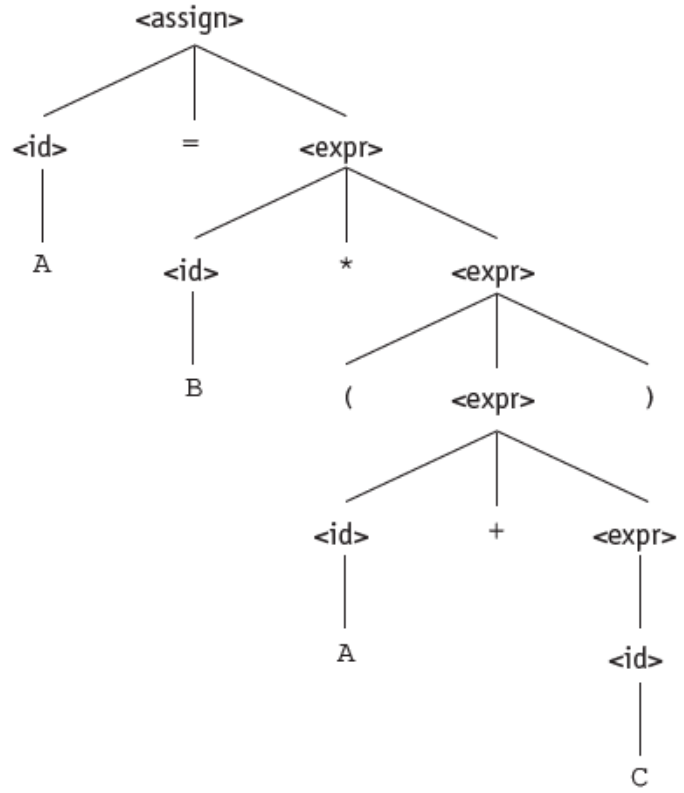


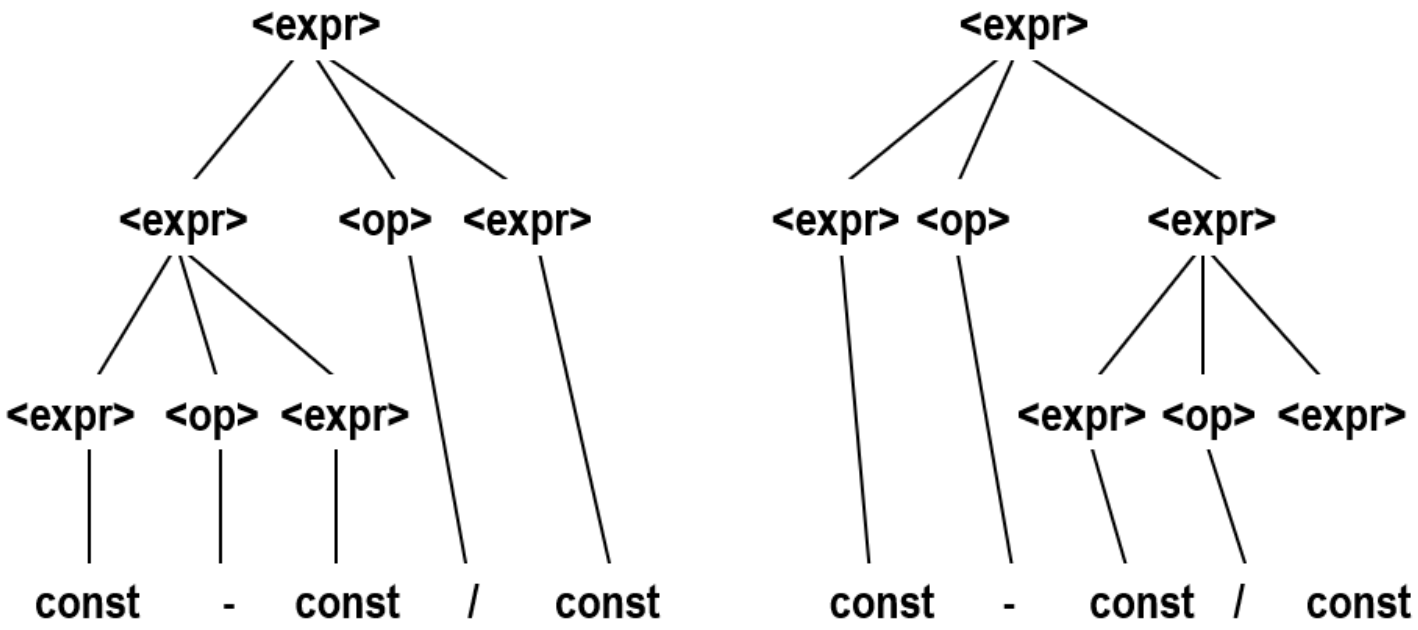
FIGURA 3.1
Uma árvore de análise sintática para a sentença simples `A = B * (A + C).`

Ambiguity in Grammars

- Uma gramática é ambígua se e somente se ela gera uma forma sentencial que possui duas ou mais árvores de análise distintas.

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const
<op> → / | -

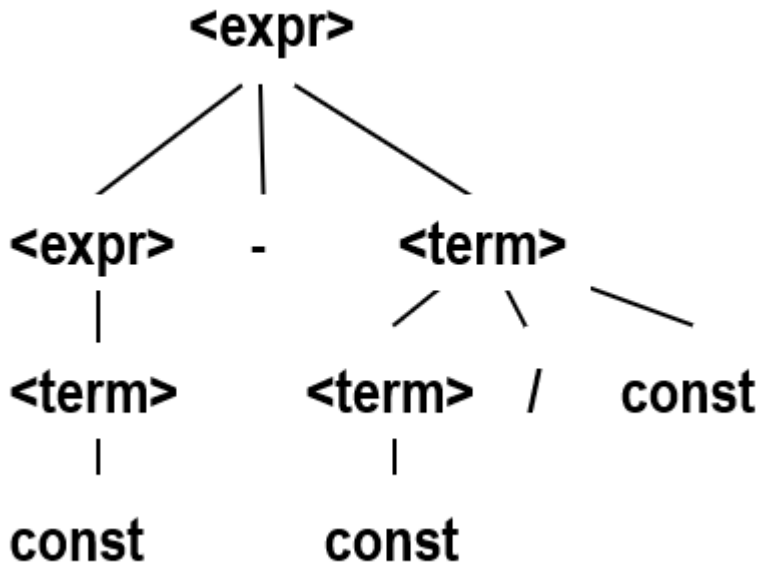


An Unambiguous Expression Grammar

Se utilizarmos a árvore de análise para indicar os níveis de precedência dos operadores, não podemos ter ambiguidade.

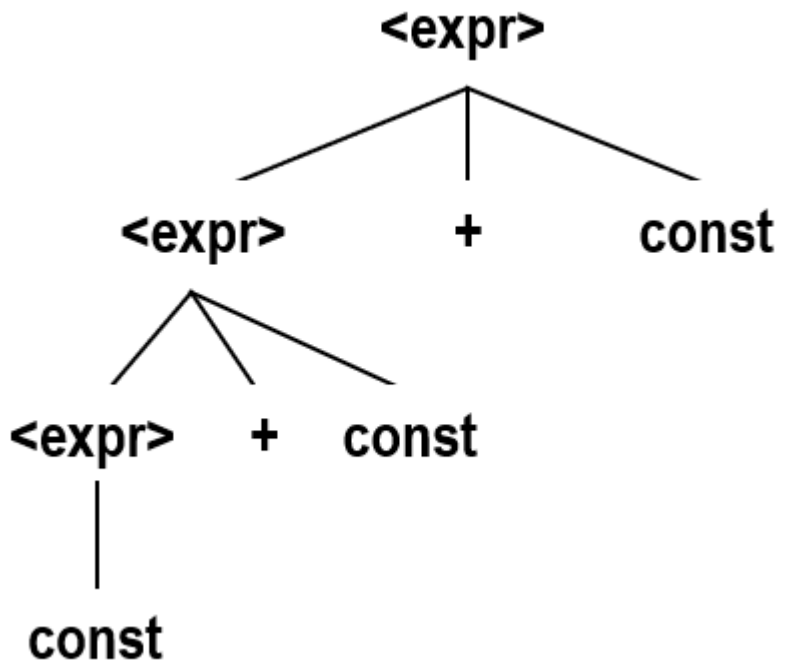
<expr> → <expr> - <term> | <term>

<term> → <term> / const | const



Associativity of Operators

- A associatividade dos operadores também pode ser indicada por uma gramática.
- <expr> → <expr> + <expr> | const (ambígua)
- <expr> → <expr> + const | const (não ambígua)



Questão 2

<assign> -> <id> = <expr>

<id> -> A | B | C

<expr> -> <expr> + <term> | <term>

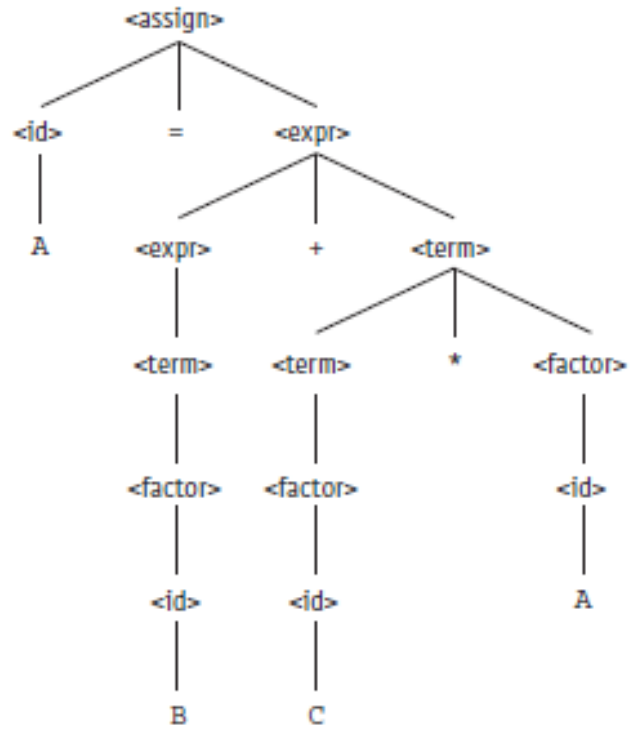
<term> -> <term> * <fator> | <fator>

<fator> -> (<expr>) | <id>

Validar a sentença: A = B + C * A
Validar a sentença: A = (A + B) * C
Validar a sentença: A = B + C + A
Validar a sentença: A = A * (B + C)
Validar a sentença: A = B * (C * (A + B))

Questão 2

<assign> => <id> = <expr>
=> <id> = <expr> + <term>
=> <id> = <expr> + <term> * <factor>
=> <id> = <expr> + <term> * <id>
=> <id> = <expr> + <term> * A
=> <id> = <expr> + <factor> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <term> + C * A
=> <id> = <factor> + C * A
=> <id> = <id> + C * A
=> <id> = B + C * A
=> A = B + C * A



Unambiguous Grammar for Selector

Java if-then-else grammar

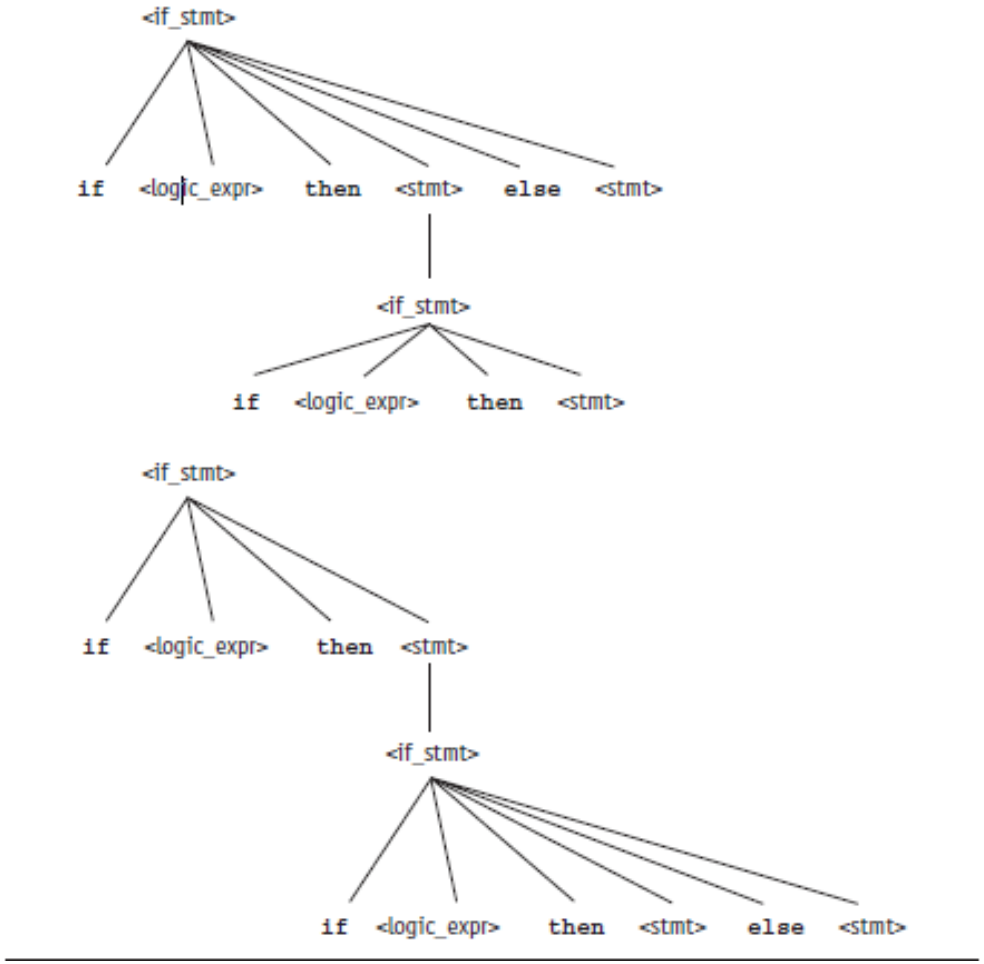
<if_stmt> -> if (<logic_expr>) <stmt>
| if (<logic_expr>) <stmt> else <stmt>

Se tivéssemos também <stmt> → <if_stmt>, essa gramática seria ambígua. A forma sentencial mais simples que ilustra essa ambiguidade é

if <logic_expr> then if <logic_expr> then <stmt> else <stmt>

```
if (done == true)
   then if denom == 0
   then quotient = 0;
   else quotient = num / denom;
```

Unambiguous Grammar for Selector



Unambiguous Grammar for Selector

An unambiguous grammar for `if-then-else`

```
<stmt> -> <matched> | <unmatched>
<matched> -> if (<logic_expr>) <stmt>
             | a non-if statement
<unmatched> -> if (<logic_expr>) <stmt>
              | if (<logic_expr>) <matched> else
                   <unmatched>
```

Extended BNF

- Partes opcionais são colocadas entre colchetes [ ]
- <proc_call> → ident [(<expr_list>)]

- Partes alternativas das partes direitas das regras são colocadas dentro de parênteses e separadas por barras verticais
- <term> → <term> (+|–) const

- Repetições (0 ou mais) são colocadas dentro de chaves { }
- <ident> → letter {letter|digit}

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
```

Semântica Estática

- Gramáticas livres de contexto (GLCs) não podem descrever toda a sintaxe das linguagens de programação
- Categorias de construções que são problemáticas:
  - Livres de contexto, mas complicadas (por exemplo, tipos de operandos em expressões)
  - Não livres de contexto (por exemplo, variáveis devem ser declaradas antes de serem usadas)

Attribute Grammars

- Gramáticas de atributos (AGs) têm adições às GLCs para carregar algumas informações semânticas nos nós da árvore de análise

- Valor primário das AGs:
  - Especificação de semântica estática
  - Projeto de compiladores (verificação de semântica estática)

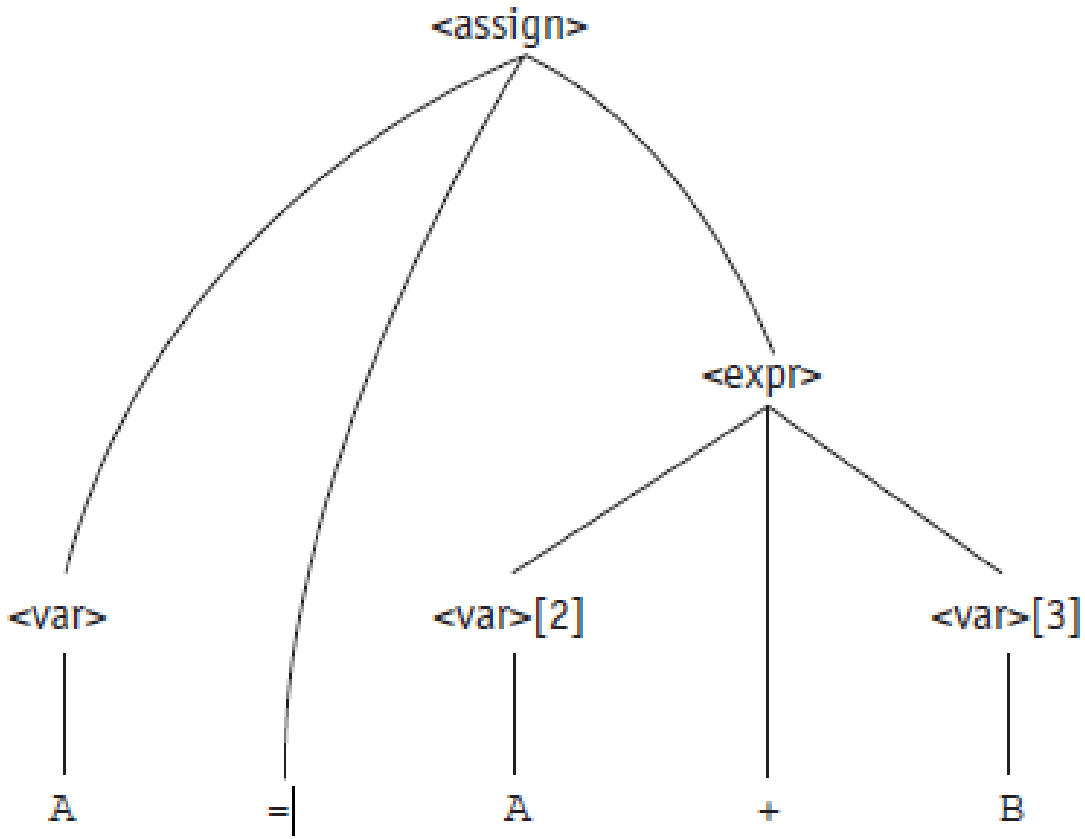
Attribute Grammars

- Os únicos nomes de variáveis são A, B e C.
- O lado direito das atribuições podem ser uma variável ou uma expressão a forma de uma variável adicionada a outra. As variáveis podem ser do tipo inteiro ou real.
- Quando existem duas variáveis no lado direito de uma atribuição, elas não precisam ser do mesmo tipo.
- O tipo da expressão quando os tipos dos operandos não são o mesmo é sempre real.
- Quando os tipos são os mesmos, o da expressão é o mesmo dos operandos.
- O tipo do lado esquerdo da atribuição deve casar com o do lado direito.
- Então, os tipos dos operandos do lado direito podem ser mistos, mas a atribuição só é válida se o alvo e o valor resultante da avaliação do lado direito são do mesmo tipo.

<assign> → <var> = <expr>
<expr> → <var> + <var> | <var>
<var> → A | B | C

Attribute Grammars

A = A + B



Attribute Grammars

Uma gramática de atributos para sentenças de atribuição simples

1. Regra sintática: <assign> → <var> = <expr>
   Regra semântica: <expr>.expected_type ← <var>.actual_type

2. Regra sintática: <expr> → <var>[2] + <var>[3]
   Regra semântica: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and (<var>[3].actual_type = int)
   then int
   else real
   Predicado: <expr>.actual_type == <expr>.expected_type

3. Regra sintática: <expr> → <var>
   Regra semântica: <expr>.actual_type ← <var>.actual_type
   Predicado: <expr>.actual_type == <expr>.expected_type

4. Regra sintática: <var> → A | B | C
   Regra semântica: <var>.actual_type ← look-up(<var>.string)

A função lookup busca um dado nome de variável na tabela de símbolos e retorna o tipo de dessa variável.

Resumo

- BNF e gramáticas livres de contexto são meta-linguagens equivalentes
  - Bem adequadas para descrever a sintaxe de linguagens de programação
- Uma gramática de atributos é um formalismo descritivo que pode descrever tanto a sintaxe quanto a semântica de uma linguagem

QUESTÕES – AULA 3

Q1. Escreva descrições EBNF para: Uma sentença de cabeçalho de definição de classe em Java.

Q2. Prove que a seguinte gramática é ambígua:

<S> → <A>

<A> → <A> + <A> | <id>

<id> → a | b | c

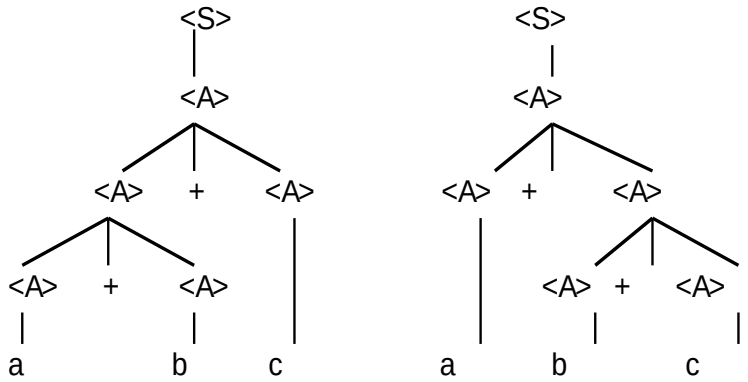
QUESTÕES – AULA 3

Resposta Q1.

<class_head> → {<modifier>} **`class`** <id> [**`extends`** class_name]
[**`implements`** <interface_name> {, <interface_name>}]

<modifier> → **`public`** | **`abstract`** | **`final`**

Resposta Q2.



AULA 04 - 20/3/2024
NOMES, VINCULAÇÕES E ESCOPOS

NOMES, VINCULAÇÕES E ESCOPOS

- Introdução
- Nomes
- Variáveis
- O Conceito de Binding
- Escopo
- Escopo e Tempo de Vida
- Ambientes de Referência
- Constantes Nomeadas

Introdução

Linguagens imperativas são abstrações da arquitetura de von Neumann.

- Memória
- Processador

Variáveis são caracterizadas por atributos

- Para projetar um tipo, é necessário considerar escopo, tempo de vida, verificação de tipo, inicialização e compatibilidade de tipo.

Nomes

Questões de design para nomes:

- Os nomes são sensíveis a maiúsculas e minúsculas?
- Existem palavras especiais que são reservadas ou palavras-chave?

Nomes

Comprimento:

- Se forem muito curtos, não podem ser conotativos.
- Exemplos de linguagens:
  - C99: sem limite, mas apenas os primeiros 63 são significativos; além disso, os nomes externos são limitados a um máximo de 31.
  - C# e Java: sem limite, e todos são significativos.
  - C++: sem limite, mas os implementadores frequentemente impõem um.

Nomes

Caracteres especiais:

- PHP: todos os nomes de variáveis devem começar com o símbolo do dólar ($).
- Perl: todos os nomes de variáveis começam com caracteres especiais ($, @, %), que especificam o tipo da variável.
- Ruby: os nomes de variáveis que começam com @ são variáveis de instância; aqueles que começam com @@ são variáveis de classe.

Nomes

Sensibilidade a maiúsculas e minúsculas:

- Desvantagem: legibilidade (nomes que se parecem são diferentes).
- Nomes nas linguagens baseadas em C são sensíveis a maiúsculas e minúsculas.
- Nomes em outras linguagens não são.
- Pior em C++, Java e C# porque os nomes pré-definidos são mistos (por exemplo, IndexOutOfBoundsException).

Nomes

Palavras especiais:

- Um auxílio para legibilidade; usado para delimitar ou separar cláusulas de declaração.
- Uma palavra-chave é uma palavra que é especial apenas em certos contextos.
- Uma palavra reservada é uma palavra especial que não pode ser usada como um nome definido pelo usuário.
- Problema potencial com palavras reservadas: Se houver muitas, muitas colisões ocorrem (por exemplo, COBOL tem 300 palavras reservadas!).

Variáveis

- Uma variável é uma abstração de uma célula de memória.
- Variáveis podem ser caracterizadas como um sexteto de atributos:
  - Nome
  - Endereço
  - Valor
  - Tipo
  - Tempo de vida
  - Escopo

Atributos de Variáveis

- Nome – nem todas as variáveis têm nomes.
- Endereço – o endereço de memória com o qual está associado.
  - Uma variável pode ter endereços diferentes em momentos diferentes durante a execução.
  - Uma variável pode ter endereços diferentes em lugares diferentes em um programa.
  - Se dois nomes de variáveis podem ser usados para acessar a mesma localização de memória, eles são chamados de aliases (apelidos).
  - Os aliases são criados através de ponteiros, variáveis de referência, uniões em C e C++.
  - Os aliases são prejudiciais para a legibilidade (os leitores do programa devem lembrar de todos eles).

Exemplo

main.c

```c
// Online C compiler to run C program online
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *i = malloc(sizeof(int));
    int *j;

    *i = 11111111;
    printf("%d\n", i);
    printf("%d\n", *i);

    j = i;
    printf("%d\n", *j);

    return 0;
}
```

Atributos de Variáveis

- Tipo – determina o intervalo de valores das variáveis e o conjunto de operações definidas para valores desse tipo; no caso de ponto flutuante, o tipo também determina a precisão.
- Valor – o conteúdo da localização com o qual a variável está associada.
  - O l-value de uma variável é seu endereço.
  - O r-value de uma variável é seu valor.
- Célula de memória abstrata – a célula física ou coleção de células associadas a uma variável.

The Concept of Binding (Vinculação)

- Um binding é uma associação entre uma entidade e um atributo, como entre uma variável e seu tipo ou valor, ou entre uma operação e um símbolo.
- O tempo de binding (Tempo de vinculação) é o momento em que um binding ocorre.

Possible Binding Times

- Tempo de design da linguagem – associa símbolos de operadores a operações. Ex: (*) multiplicação.
- Tempo de implementação da linguagem – associa o tipo de ponto flutuante a uma representação. Ex: (float) faixa de valores.
- Tempo de compilação – associa uma variável a um tipo em C ou Java. Ex: float A;
- Tempo de carregamento – associa uma variável estática em C ou C++ a uma célula de memória. Ex: float A = 10.10;
- Tempo de execução – associa uma variável local não estática a uma célula de memória.

Static and Dynamic Binding

- Um binding é estático se ocorrer pela primeira vez antes do tempo de execução e permanecer inalterado durante a execução do programa.
- Um binding é dinâmico se ocorrer pela primeira vez durante a execução ou puder mudar durante a execução do programa.

Type Binding (Vinculações de tipo)

- Como um tipo é especificado?
- Quando ocorre o binding?
- Se for estático, o tipo pode ser especificado por meio de uma declaração explícita ou implícita.

Explicit/Implicit Declaration

- Uma declaração explícita é uma instrução do programa usada para declarar os tipos de variáveis.
- Uma declaração implícita é um mecanismo padrão para especificar os tipos de variáveis por meio de convenções padrão, em vez de declarações explícitas.
- EX:
  - Var sum = 0;
  - Var total = 0.0;
  - Var name = “João”;
- Linguagens como Basic, Perl, Ruby, JavaScript e PHP oferecem declarações implícitas.
  - Vantagem: facilidade de escrita (uma conveniência menor).
  - Desvantagem: confiabilidade (menos problemas com Perl).

Explicit/Implicit Declaration (continued)

Algumas linguagens usam inferência de tipos para determinar os tipos de variáveis com base no contexto.

- Em C#, uma variável pode ser declarada com 'var' e um valor inicial. O tipo é determinado pelo valor inicial.
- Visual Basic 9.0+, ML, Haskell e F# utilizam inferência de tipos. O contexto em que a variável aparece determina seu tipo.

Dynamic Type Binding

- Vinculação de Tipo Dinâmico (JavaScript, Python, Ruby, PHP e C# (limitado))
- Especificado por meio de uma instrução de atribuição, por exemplo, em JavaScript:

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

  - Vantagem: flexibilidade (unidades de programa genéricas)
  - Desvantagens:
    - Alto custo (verificação dinâmica de tipo e interpretação)
    - Detecção de erros de tipo pelo compilador é difícil.

Variable Attributes (continued)

- Ligações de Armazenamento e Tempo de Vida
  - Alocação – obtenção de uma célula de um pool de células disponíveis
  - Desalocação – colocação de uma célula de volta no pool
- O tempo de vida de uma variável é o tempo durante o qual ela está vinculada a uma célula de memória específica.

Categories of Variables by Lifetimes

- Estático – vinculado a células de memória antes do início da execução e permanece vinculado à mesma célula de memória durante toda a execução, por exemplo, variáveis estáticas em funções de C e C++.
  - Vantagens: eficiência (endereçamento direto), suporte a subprogramas sensíveis ao histórico.
  - Desvantagem: falta de flexibilidade (ausência de recursão).

Categories of Variables by Lifetimes

- Dinâmica de Pilha – Ligações de armazenamento são criadas para variáveis quando suas declarações são elaboradas. (Uma declaração é elaborada quando o código executável associado a ela é executado).
- Se escalar, todos os atributos, exceto o endereço, são vinculados estaticamente.
  - Variáveis locais em subprogramas C (não declaradas como estáticas) e métodos Java.
- Vantagem: permite recursão; conserva armazenamento.
- Desvantagens:
  - Overhead de alocação e desalocação.
  - Subprogramas não podem ser sensíveis ao histórico.
  - Referências ineficientes (endereçamento indireto).

Categories of Variables by Lifetimes

- Explícita dinâmica de heap – Alocada e desalocada por diretivas explícitas, especificadas pelo programador, que têm efeito durante a execução.
- Referenciada apenas por meio de ponteiros ou referências, por exemplo, objetos dinâmicos em C++ (via new e delete), todos os objetos em Java.
- Vantagem: fornece gerenciamento de armazenamento dinâmico.
- Desvantagem: ineficiente e pouco confiável.

Categories of Variables by Lifetimes

- Dinâmica implícita de heap – A alocação e desalocação são causadas por instruções de atribuição.
  - Todas as variáveis em APL; todas as strings e arrays em Perl, JavaScript e PHP.
  - Ex: H = [74, 84, 86, 90, 71];
- Vantagem: flexibilidade (código genérico).
- Desvantagens:
  - Ineficiente, porque todos os atributos são dinâmicos.
  - Perda de detecção de erros.

Variable Attributes: Scope

- O escopo de uma variável é o conjunto de instruções sobre o qual ela é visível.
- As variáveis locais de uma unidade de programa são aquelas declaradas dentro dessa unidade.
- As variáveis não locais de uma unidade de programa são aquelas visíveis na unidade, mas não declaradas lá.
- As variáveis globais são uma categoria especial de variáveis não locais.
- As regras de escopo de uma linguagem determinam como as referências aos nomes são associadas às variáveis.

Scope (continued)

```
function Big() {
   function sub1() {
        var x = 7;
        sub2();
   }
   funcion sub2() }
        var y = x;
   }
   var x = 3;
   sub1();
}
```

Blocks

- Um método de criar escopos estáticos dentro de unidades de programa – de ALGOL 60.
- Exemplo em C:

```
- void sub() {
        int count;
        while (...) {
          int count;
          count++;
          ...
        }
        …
      }
```

- Note: legal in C and C++, but not in Java and C# – too error-prone

The LET Construct

- Na maioria das linguagens funcionais, inclui-se algum tipo de construção let.
- Um construtor let tem duas partes:
  - A primeira parte associa nomes a valores.
  - A segunda parte utiliza os nomes definidos na primeira parte.
- In Scheme:

```
(LET (
     (top (+ a b) )
     (bottom (- c d) )
     )
     (/ top botton)
)
```

Declaration Order

- Em C99, C++, Java e C#, é permitido que as declarações de variáveis apareçam em qualquer lugar onde uma instrução possa aparecer.
  - Em C99, C++ e Java, o escopo de todas as variáveis locais é da declaração até o final do bloco.
  - Em C#, o escopo de qualquer variável declarada em um bloco é todo o bloco, independentemente da posição da declaração no bloco.
  - No entanto, uma variável ainda deve ser declarada antes de ser utilizada.

Declaration Order (continued)

- In C++, Java, e C#, variáveis podem ser declaradas dentro de um `for`
  - O escopo das variáveis fica dentro do for

Global Scope

- C, C++, PHP e Python suportam uma estrutura de programa que consiste em uma sequência de definições de função em um arquivo.
  - Essas linguagens permitem que declarações de variáveis apareçam fora das definições de função.

- Em C e C++, existem declarações (apenas atributos) e definições (atributos e armazenamento).
  - Uma declaração fora de uma definição de função especifica que ela está definida em outro arquivo.
  - Ex: extern int sum;

Global Scope (continued)

- Em PHP, os programas são incorporados em documentos de marcação HTML, em qualquer número de fragmentos, com algumas declarações e algumas definições de função.
- O escopo de uma variável declarada (implicitamente) em uma função é local para a função.
- O escopo de uma variável declarada implicitamente fora de funções é da declaração até o final do programa, mas pula quaisquer funções intermediárias.
- Variáveis globais podem ser acessadas em uma função através da matriz $GLOBALS ou declarando-as como globais.
- Em Python, uma variável global pode ser referenciada em funções, mas só pode ser atribuída dentro de uma função se tiver sido declarada como global dentro da função

Global Scope (continued)

```
day = "Monday"
        def tester():
                print "The global day is:", day
tester()
-----------------------------------
day = "Monday"
def tester():
    print "The global day is:", day
    day = "Tuesday"
    print "The new value of day is:", day
tester()
-----------------------------------
day = "Monday"
        def tester():
        global day
        print "The global day is:", day
        day = "Tuesday"
        print "The new value of day is:", day
tester()
```

Evaluation of Static Scoping

Funciona bem em muitas situações, mas apresenta alguns problemas:

- Na maioria dos casos, é possível ter acesso demais.
- Conforme um programa evolui, a estrutura inicial é destruída e variáveis locais frequentemente se tornam globais; subprogramas também tendem a se tornar globais em vez de serem aninhados.

Dynamic Scope (Escopo Dinâmico)

- Com base nas sequências de chamadas das unidades de programa, e não em seu layout textual (temporal versus espacial).
- As referências às variáveis são conectadas às declarações através de uma busca de retorno pela cadeia de chamadas de subprogramas que forçou a execução até este ponto.

Scope Example

```
function big() {
                        function sub1()
                           var x = 7;
                        function sub2() {
                          var y = x;
                        }
                      var x = 3;
                    }
```

big calls sub1
sub1 calls sub2
sub2 uses x

- Escopo estático:
- Referência a x em sub2 é para o x de big.

- Escopo dinâmico:
- Referência a x em sub2 é para o x de sub1.

Scope Example

- Avaliação do Escopo Dinâmico:
  - Vantagem: conveniência.
  - Desvantagens:
    - Enquanto um subprograma está em execução, suas variáveis são visíveis para todos os subprogramas que ele chama.
    - Impossível fazer verificação estática de tipo.
    - Baixa legibilidade – não é possível determinar estaticamente o tipo de uma variável.

Scope and Lifetime

- O escopo e o tempo de vida são às vezes conceitos intimamente relacionados, mas são diferentes.
- Considere uma variável estática em uma função em C ou C++.

```
void printheader() {
...
} /* Fim de printheader */

void compute() {
    int sum;
    ...
    printheader();
} /* Fim de compute */
```

Referencing Environments

- O ambiente de referência de uma instrução é a coleção de todos os nomes que são visíveis na instrução.
- Em uma linguagem de escopo estático, são as variáveis locais mais todas as variáveis visíveis em todos os escopos envolventes.
- Um subprograma está ativo se sua execução já começou, mas ainda não terminou.
- Em uma linguagem de escopo dinâmico, o ambiente de referência é composto pelas variáveis locais mais todas as variáveis visíveis em todos os subprogramas ativos.

Referencing Environments

```
void sub1() {
  int a, b;
  ...  <--------------- 1
 }  /* Fim de sub1 */
void sub2() {
  int b, c;
  ...  <--------------- 2
  sub1;

 }  /* end of sub2 */
void main() {
  int c, d;
  ...  <--------------- 3
  sub2();
 }  /* Fim de main */
```

Ponto	*Ambiente de referenciamento*
1	a e b de sub1, c de sub2, d de main, (c de main e b de sub2 estão ocultas)
2	b e c de sub2, d de main, (c de main está oculta)
3	c e d de main

Named Constants

- Uma constante nomeada é uma variável que está associada a um valor apenas quando está vinculada ao armazenamento.
- Vantagens: legibilidade e modificabilidade. Usadas para parametrizar programas.
- A vinculação de valores a constantes nomeadas pode ser estática (chamadas de constantes manifestas) ou dinâmicas.
- Em C++ e Java, expressões de qualquer tipo são vinculadas dinamicamente.
- Em C#, existem dois tipos, readonly e const:
  - Os valores das constantes nomeadas const são vinculados em tempo de compilação.
  - Os valores das constantes nomeadas readonly são vinculados dinamicamente.

Resumo

- Sensibilidade a maiúsculas e minúsculas e a relação dos nomes com palavras especiais representam questões de design de nomes.
- Variáveis são caracterizadas pelos sextetos: nome, endereço, valor, tipo, duração, escopo.
- Vinculação é a associação de atributos com entidades do programa.
- Variáveis escalares são categorizadas como: estáticas, dinâmicas de pilha, dinâmicas explícitas de heap, dinâmicas implícitas de heap.
- Tipagem forte significa detectar todos os erros de tipo.

QUESTÕES – AULA 4

Q1. Assuma que o seguinte programa Ada foi compilado e executado usando regras de escopo estático. Que valor de X é impresso no procedimento Sub1? Sob regras de escopo dinâmico, qual o valor de X impresso no procedimento Sub1?

```
procedure Main is
  X : Integer;
  procedure Sub1 is
    begin  -- de Sub1
    Put(X);
    end;  -- de Sub1
  procedure Sub2 is
    X : Integer;
    begin  -- de Sub2
    X := 10;
    Sub1
    end;  -- de Sub2
  begin  -- de Main
  X := 5;
  Sub2
  end;  -- de Main
```

QUESTÕES – AULA 4

Q2.

Considere o programa:

```
procedure Main is
  X, Y, Z : Integer;
  procedure Sub1 is
    A, Y, Z : Integer;
    procedure Sub2 is
     A, B, Z : Integer;
     begin  -- de Sub2
    ...
     end;  -- de Sub2
    begin  -- de Sub1
    ...
    end;  -- de Sub1
  procedure Sub3 is
    A, X, W : Integer;
    begin  -- de Sub3
    ...
    end;  -- de Sub3
  begin  -- de Main
  ...
  end;  -- de Main
```

Liste todas as variáveis, com as unidades de programa onde elas estão declaradas, visíveis nos corpos de `Sub1`, `Sub2` e `Sub3`, assumindo que o escopo estático esteja sendo usado.

QUESTÕES – AULA 4

Q3.

Considere o programa em C:

```
void fun(void) {
  int a, b, c; /* definição 1 */
 ...
  while (...) {
    int b, c, d; /*definition 2 */
    ... <-------------- 1
    while (...) {
     int c, d, e; /* definition 3 */
     ... <------------- 2
     }
    ... <-------------- 3
   }
  ... <---------------- 4
 }
```

Para cada um dos quatro pontos marcados nessa função, liste cada variável visível, com o número da sentença de definição que a define.

QUESTÕES – AULA 4

Resposta Q1.

Static scoping: x is 5
Dynamic scoping: x is 10

Resposta Q2.

Variable	Where Declared
In sub1:	
a	sub1
y	sub1
z	sub1
x	main
In sub2:	
a	sub2
b	sub2
z	sub2
y	sub1
x	main
In sub3:	
a	sub3
x	sub3
w	sub3
y	main
z	main

QUESTÕES – AULA 4

Resposta Q3.

Point 1:	a	1
	b	2
	c	2
	d	2
Point 2:	a	1
	b	2
	c	3
	d	3
	e	3

Point 3: same as Point 1

Point 4:	a	1
	b	1
	c	1

AULA 05 - 27/3/2024
Tipos de dados

Tópicos

- Introdução
- Tipos de Dados Primitivos
- Tipos de Dados de Sequência de Caracteres
- Tipos de Enumeração
- Tipos de Arranjo
- Arranjos Associativos
- Tipos de Registro
- Tipos de Tupla
- Tipos de Lista
- Tipos de União
- Tipos de Ponteiro e Referência
- Verificação de Tipo
- Tipagem Forte
- Equivalência de Tipo
- Teoria e Tipos de Dados

Introdução

- Um tipo de dado define uma coleção de valores de dados e um conjunto de operações predefinidas sobre esses valores.
- Um descritor é a coleção dos atributos de uma variável.
- Um objeto representa uma instância de um tipo de dado definido pelo usuário (dados abstratos).
- Uma questão de design para todos os tipos de dados: Quais operações estão definidas e como elas são especificadas?

Primitive Data Types

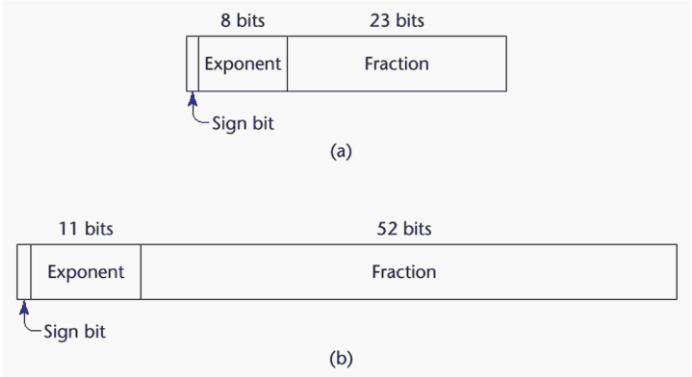
- Quase todas as linguagens de programação fornecem um conjunto de tipos de dados primitivos.
- Tipos de dados primitivos: aqueles não definidos em termos de outros tipos de dados.
- Alguns tipos de dados primitivos são simplesmente reflexos do hardware.
- Outros requerem apenas um pouco de suporte não relacionado ao hardware para sua implementação.

Primitive Data Types: Integer

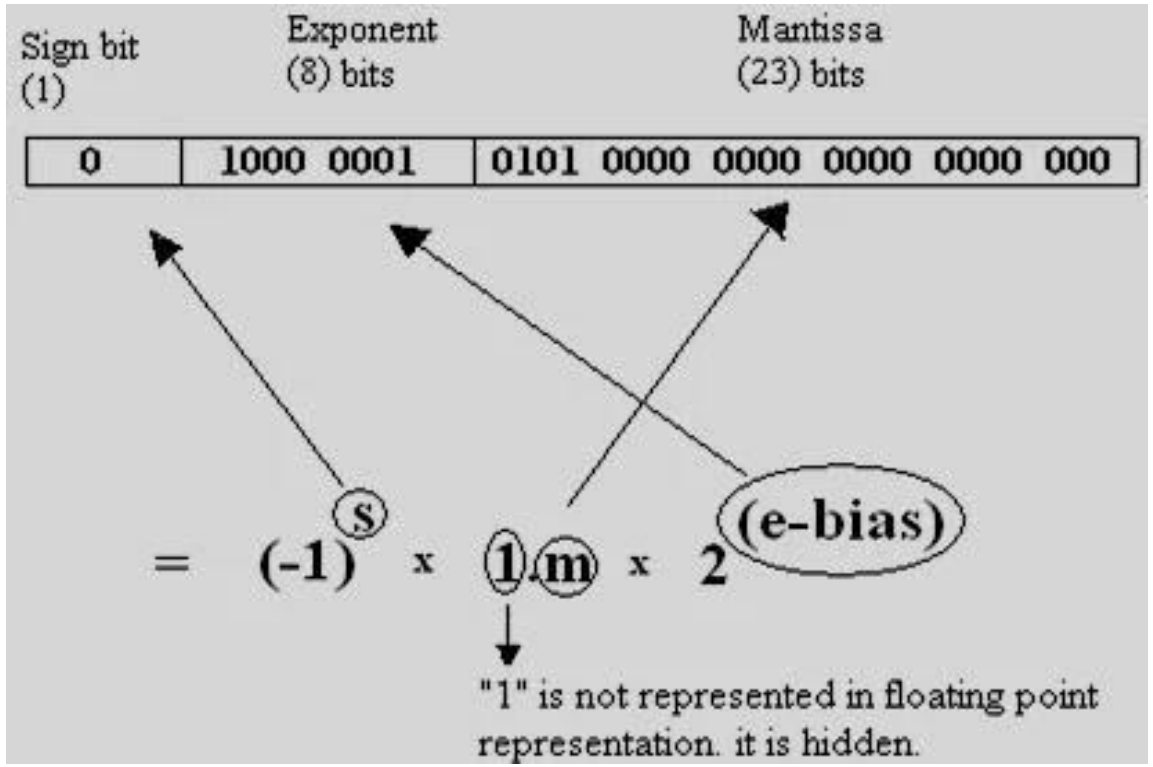
- Quase sempre é um reflexo exato do hardware, então o mapeamento é trivial.
- Pode haver até oito tipos diferentes de inteiros em uma linguagem.
- Tamanhos de inteiros assinados em Java: byte, short, int, long.

Primitive Data Types: Floating Point

- Modelam números reais, mas apenas como aproximações.
- Linguagens para uso científico suportam pelo menos dois tipos de ponto flutuante (por exemplo, float e double; às vezes mais).
- Normalmente são exatamente como o hardware, mas nem sempre.
- Ponto flutuante IEEE
  - Padrão 754



Primitive Data Types: Floating Point



$$\text{bias} = 2^{(8-1)} - 1$$
$$= 127$$

$$\text{Decimal value} = (-1)^0 \times 1.0101\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{(129-127)}$$
$$= 1.0101\ 0000\ 0000\ 0000\ 0000\ 000 \times 2^{(2)}$$
$$= 101.01\ 0000\ 0000\ 0000\ 0000\ 000 \text{ (Binary fraction)}$$

↓ Convert binary fraction to decimal

$$= (1x2^2)+(0x2^1)+(1x2^0) . +(0x2^{-1}) + (1x2^{-2}) + (0x2^{-3}) \ldots\ldots\ldots (0x2^{-21})$$
$$= (4+0+1+0+0.25)$$
$$= 5.25$$

$$\text{Decimal Equivalent Value} = (-1)^s \times 1.m \times 2^{(e-bias)}$$

$$\text{bias} = 2^{(e-1)} - 1$$

https://www.rfwireless-world.com/Tutorials/floating-point-tutorial.html

Primitive Data Types: Complex

- Algumas linguagens suportam um tipo complexo, por exemplo, C99, Fortran e Python.
- Cada valor consiste em dois floats, a parte real e a parte imaginária.
- Forma literal (em Python):
- (7 + 3j), onde 7 é a parte real e 3 é a parte imaginária.

Primitive Data Types: Decimal

- Para aplicações empresariais (dinheiro)
  - Essencial para COBOL
  - C# oferece um tipo de dado decimal
- Armazena um número fixo de dígitos decimais, em forma codificada (BCD)
- Vantagem: precisão
- Desvantagens: alcance limitado, desperdício de memória

Primitive Data Types: Boolean

- O mais simples de todos
- Intervalo de valores: dois elementos, um para "verdadeiro" e outro para "falso"
- Pode ser implementado como bits, mas frequentemente como bytes
  - Vantagem: legibilidade

Primitive Data Types: Character

- Armazenado como codificações numéricas
- Codificação mais comumente usada: ASCII
- Uma alternativa, codificação de 16 bits: Unicode (UCS-2)
  - Inclui caracteres da maioria das línguas naturais
  - Originalmente usado em Java
  - C# e JavaScript também suportam Unicode
- Unicode de 32 bits (UCS-4)
  - Suportado por Fortran, a partir de 2003

Primitive Data Types: Character

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Character String Types

- Valores são sequências de caracteres
- Questões de design:
  - É um tipo primitivo ou apenas um tipo especial de array?
  - O comprimento das strings deve ser estático ou dinâmico?

Character String Types Operations

- Operações típicas:
  - Atribuição e cópia
  - Comparação (=, >, etc.)
  - Concatenação
  - Referência de subcadeia
  - Correspondência de padrões

Character String Type in Certain Languages

- C e C++
  - Não primitivo
  - Usam arrays de char e uma biblioteca de funções que fornecem operações
- SNOBOL4 (uma linguagem de manipulação de strings)
  - Primitivo
  - Muitas operações, incluindo correspondência de padrões elaborada
- Fortran e Python
  - Tipo primitivo com atribuição e várias operações
- Java
  - Primitivo via classe String
- Perl, JavaScript, Ruby e PHP
  - Fornecem correspondência de padrões integrada, usando expressões regulares

Character String Length Options

- Estático: COBOL, classe String do Java
- Com Comprimento Dinâmico Limitado: C e C++
  - Nessas linguagens, um caractere especial é usado para indicar o final dos caracteres de uma string, em vez de manter o comprimento
- Dinâmico (sem máximo): SNOBOL4, Perl, JavaScript

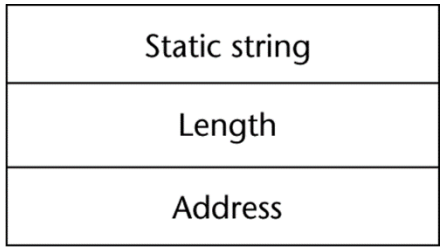
Character String Type Evaluation

- Ajuda à escrevibilidade
- Como um tipo primitivo com comprimento estático, eles são baratos de fornecer – por que não tê-los?
- O comprimento dinâmico é bom, mas vale a pena o custo?

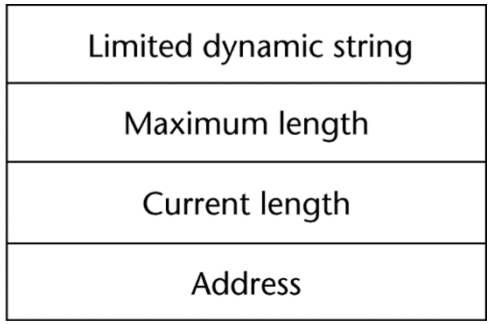
Character String Implementation

- Comprimento estático: descritor em tempo de compilação
- Comprimento dinâmico limitado: pode precisar de um descritor em tempo de execução para o comprimento (mas não em C e C++)
- Comprimento dinâmico: precisa de um descritor em tempo de execução; alocação/desalocação é o maior problema de implementação.

Compile- and Run-Time Descriptors



Compile-time descriptor for static strings



Run-time descriptor for limited dynamic strings

User-Defined Ordinal Types

- Um tipo ordinal é aquele em que o intervalo de valores possíveis pode ser facilmente associado ao conjunto de inteiros positivos.
- Exemplos de tipos ordinais primitivos em Java incluem:
  - inteiro
  - char
  - boolean

Enumeration Types

- Todos os valores possíveis, que são constantes nomeadas, são fornecidos na definição.
- Um exemplo em C# seria:
  - `enum days {mon, tue, wed, thu, fri, sat, sun};`
- Questões de design:
  - Uma constante de enumeração é permitida a aparecer em mais de uma definição de tipo, e se sim, como é verificado o tipo de uma ocorrência dessa constante?
  - Os valores de enumeração são convertidos para inteiro?
  - Qualquer outro tipo é convertido para um tipo de enumeração?

Evaluation of Enumerated Type

- Ajuda à legibilidade, por exemplo, não é necessário codificar uma cor como um número.
- Ajuda à confiabilidade, por exemplo, o compilador pode verificar:
  - operações (não permitir que cores sejam adicionadas)
  - Nenhuma variável de enumeração pode ser atribuída a um valor fora de sua faixa definida.
  - C# e Java 5.0 fornecem um melhor suporte para enumeração do que C++ porque variáveis de tipo enumeração nessas linguagens não são convertidas para tipos inteiros.

Array Types

- Um array é um agregado homogêneo de elementos de dados no qual um elemento individual é identificado pela sua posição no agregado, em relação ao primeiro elemento.

Array Design Issues

- Quais tipos são legais para subscritos?
- As expressões de subscrito em referências a elementos são verificadas quanto ao alcance?
- Quando os intervalos de subscrito são vinculados?
- Quando ocorre a alocação?
- Arrays multidimensionais irregulares ou retangulares são permitidos, ou ambos?
- Qual é o número máximo de subscritos?
- Objetos de array podem ser inicializados?
- São suportados quaisquer tipos de fatias?

Array Indexing

- Indexação (ou subindexação) é um mapeamento de índices para elementos.

  array_name (index_value_list) → an element
- Index Syntax
  - Fortran e Ada usam parênteses.
  - Ada usa explicitamente parênteses para mostrar uniformidade entre referências de array e chamadas de função porque ambas são mapeamentos.
  - A maioria das outras linguagens usa colchetes.

Arrays Index (Subscript) Types

- FORTRAN, C: apenas tipos inteiros
- Java: apenas tipos inteiros
- Verificação de intervalo de índice
  - C, C++, Perl e Fortran não especificam verificação de intervalo
  - Java, ML, C# especificam verificação de intervalo

Subscript Binding and Array Categories

- Estático: os intervalos de subscrito são vinculados estaticamente e a alocação de armazenamento é estática (antes do tempo de execução)
  - Vantagem: eficiência (sem alocação dinâmica)
- Dinâmico de pilha fixa: os intervalos de subscrito são vinculados estaticamente, mas a alocação é feita no momento da declaração
  - Vantagem: eficiência de espaço

Subscript Binding and Array Categories (continued)

- Heap-dynamic fixo: similar ao dinâmico de pilha fixa: a ligação de armazenamento é dinâmica, mas fixa após a alocação (ou seja, a ligação é feita quando solicitada e o armazenamento é alocado do heap, não da pilha).

Subscript Binding and Array Categories (continued)

- Heap-dynamic: a vinculação de intervalos de subscrito e a alocação de armazenamento são dinâmicas e podem mudar qualquer número de vezes.
  - Vantagem: flexibilidade (arrays podem crescer ou encolher durante a execução do programa).

Subscript Binding and Array Categories (continued)

- Em C e C++, arrays que incluem o modificador static são estáticos.
- Arrays em C e C++ sem o modificador static são dinâmicos de pilha fixa.
- C e C++ oferecem arrays dinâmicos de heap fixo.
- C# inclui uma segunda classe de array, ArrayList, que fornece dinâmica de heap fixo.
- Perl, JavaScript, Python e Ruby suportam arrays dinâmicos de heap.

Array Initialization

- Some language allow initialization at the time of storage allocation
  - C, C++, Java, C# example

```
int list [] = {4, 5, 7, 83}
```

  - Character strings in C and C++

```
char name [] = "freddie";
```

  - Arrays of strings in C and C++

```
char *names [] = {"Bob", "Jake", "Joe"];
```

  - Java initialization of String objects

```
String[] names = {"Bob", "Jake", "Joe"};
```

Heterogeneous Arrays

- Um array heterogêneo é aquele em que os elementos não precisam ser do mesmo tipo.
- Isso é suportado por Perl, Python, JavaScript e Ruby.

Array Initialization

- C-based languages
  - `int list [] = {1, 3, 5, 7}`
  - `char *names [] = {"Mike", "Fred", "Mary Lou"};`
- Python
  - List comprehensions

```
list = [x ** 2 for x in range(12) if x % 3 == 0]
   puts [0, 9, 36, 81] in list
```

Arrays Operations

- APL fornece as operações de processamento de arrays mais poderosas para vetores e matrizes, bem como operadores unários (por exemplo, para reverter elementos de coluna).
- Python oferece atribuições de arrays, mas são apenas mudanças de referência. Python também suporta concatenação de arrays e operações de pertencimento de elementos.
- Ruby também fornece concatenação de arrays.

Rectangular and Jagged Arrays

- Um array retangular é um array multidimensional no qual todas as linhas têm o mesmo número de elementos e todas as colunas têm o mesmo número de elementos.
- Uma matriz irregular tem linhas com número variável de elementos.
  - É possível quando arrays multidimensionais aparecem na verdade como arrays de arrays.
- C, C++ e Java suportam arrays irregulares.
- F# e C# suportam arrays retangulares e arrays irregulares.

Slices

- Um slice é alguma subestrutura de um array; nada mais do que um mecanismo de referência.
- Slices só são úteis em linguagens que possuem operações de array.

Slice Examples

- Python

```
vector = [2, 4, 6, 8, 10, 12, 14, 16]
mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

`vector (3:6)` is a three-element array

`mat[0][0:2]` is the first and second element of the first row of `mat`

- Ruby supports slices with the `slice` method

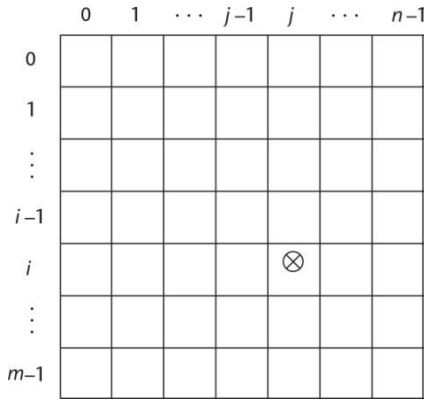
`list.slice(2, 2)` returns the third and fourth elements of `list`

Implementation of Arrays

- A função de acesso mapeia expressões de subscrito para um endereço no array.
- Função de acesso para arrays unidimensionais:

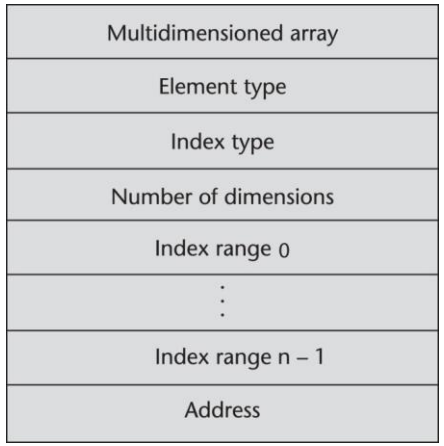
  address(list[k]) = address (list[lower_bound])

  + ((k–lower_bound) * element_size)



Accessing Multi-dimensioned Arrays

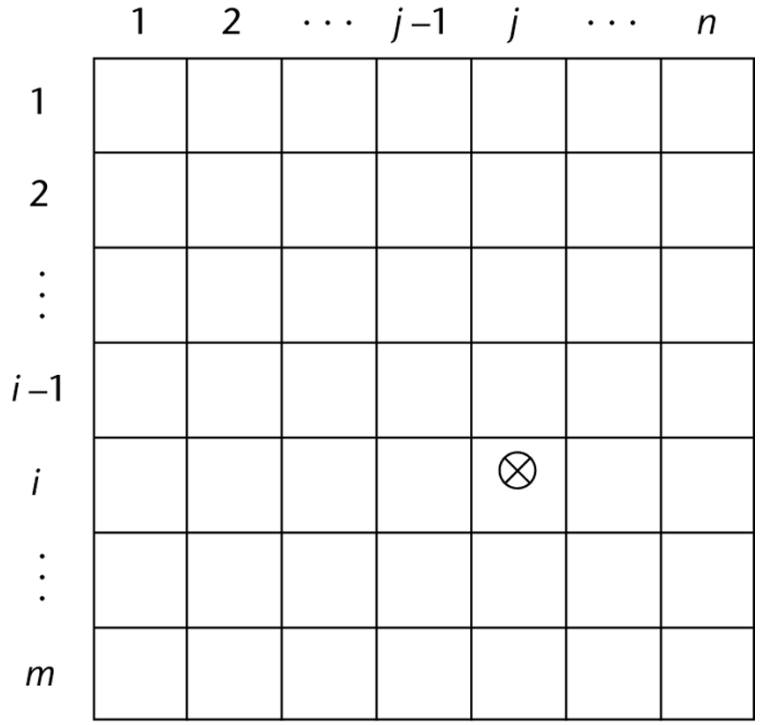
- Duas formas comuns:
  - Ordem principal por linha (por linhas) – usada na maioria das linguagens
  - Ordem principal por coluna (por colunas) – usada em Fortran
  - Um descritor em tempo de compilação para um array multidimensional



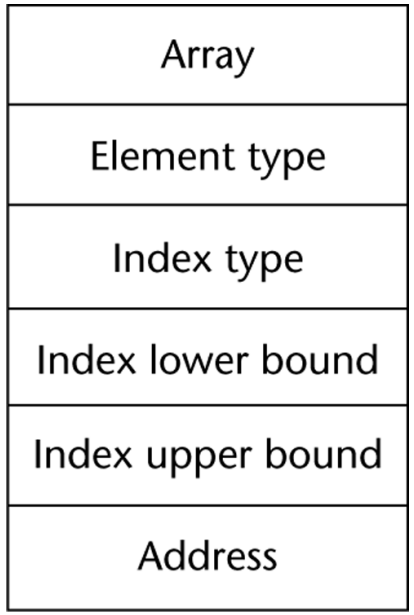
Locating an Element in a Multi-dimensioned Array

General format

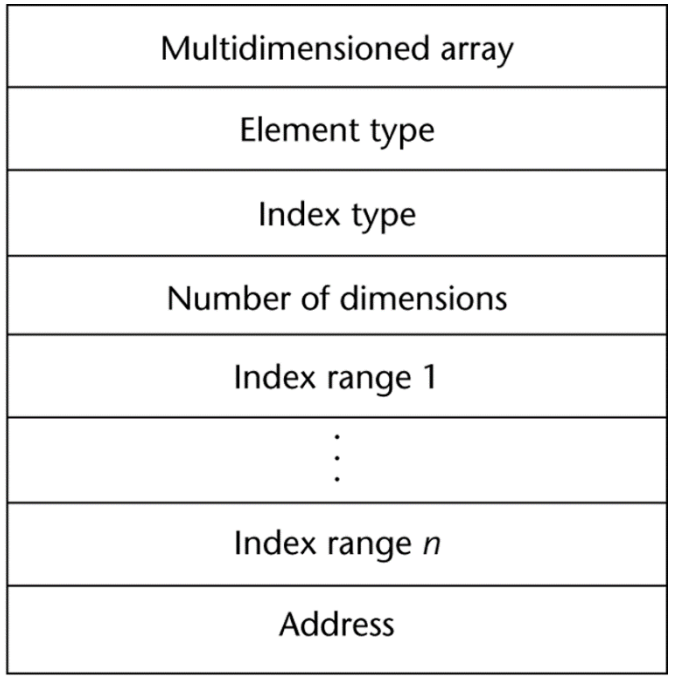
Location (a[I,j]) = address of a [row_lb,col_lb] + (((I - row_lb) * n) + (j - col_lb))
* element_size



Compile-Time Descriptors



Single-dimensioned array



Multidimensional array

Associative Arrays

- Um array associativo é uma coleção desordenada de elementos de dados que são indexados por um número igual de valores chamados chaves.
  - As chaves definidas pelo usuário devem ser armazenadas.
- Questões de design:
- – Qual é a forma das referências aos elementos?
- – O tamanho é estático ou dinâmico?
- Tipo embutido em Perl, Python, Ruby e Lua.
- Em Lua, eles são suportados por tabelas.

Associative Arrays in Perl

- Names begin with `%`; literals are delimited by parentheses

```
%hi_temps = ("Mon" => 77, "Tue" => 79, "Wed" => 65, …);
```

- Subscripting is done using braces and keys

```
$hi_temps{"Wed"} = 83;
```

  - Elements can be removed with **`delete`**

```
delete $hi_temps{"Tue"};
```

Record Types

- Um registro é um agregado possivelmente heterogêneo de elementos de dados no qual os elementos individuais são identificados por nomes.
- Questões de design:
  - Qual é a forma sintática das referências ao campo?
  - São permitidas referências elípticas?

Definition of Records in COBOL

- COBOL usa números de nível para mostrar registros aninhados; outros usam definição recursiva.

- Exemplo em COBOL:

```
01 EMP-REC.
02 EMP-NAME.
   05 FIRST PIC X(20).
   05 MID   PIC X(10).
   05 LAST  PIC X(20).
02 HOURLY-RATE PIC 99V99.
```

References to Records

- Record field references
  1. COBOL
  field_name OF record_name_1 OF ... OF record_name_n
  2. Others (dot notation)
  record_name_1.record_name_2. ... record_name_n.field_name

- Fully qualified references must include all record names

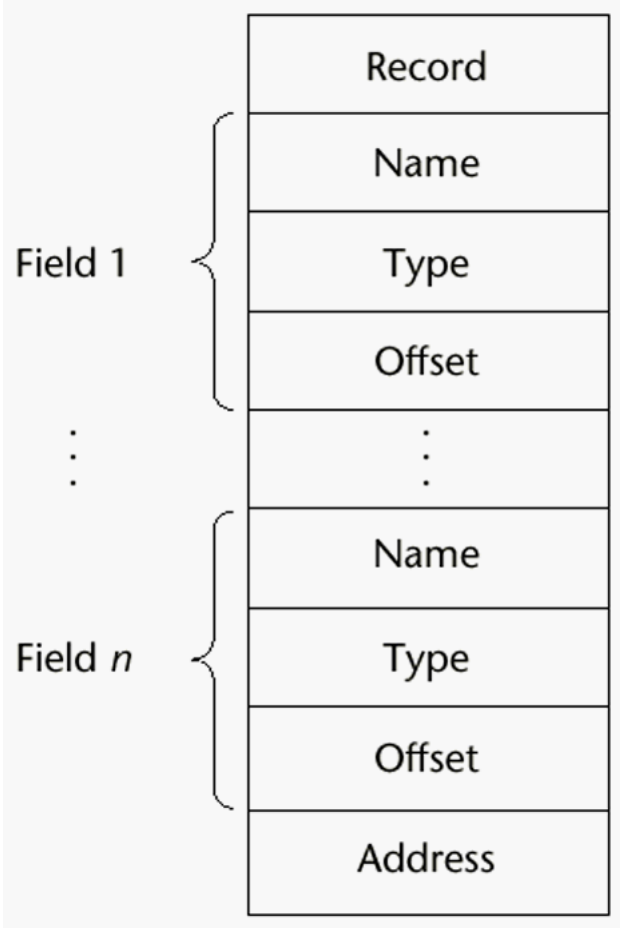
- Elliptical references allow leaving out record names as long as the reference is unambiguous, for example in COBOL
  `FIRST, FIRST OF EMP-NAME`, and `FIRST` of EMP-REC are elliptical references to the employee's first name

Evaluation and Comparison to Arrays

- Registros são usados quando a coleção de valores de dados é heterogênea.
- O acesso aos elementos do array é muito mais lento do que o acesso aos campos do registro, porque os subscritos são dinâmicos (os nomes dos campos são estáticos).
- Subscritos dinâmicos poderiam ser usados com o acesso aos campos do registro, mas isso impediria a verificação de tipo e seria muito mais lento.

Implementation of Record Type

Offset address relative to the beginning of the records is associated with each field



Tuple Types

- Uma tupla é um tipo de dado que é semelhante a um registro, exceto que os elementos não são nomeados.
- Usado em Python, ML e F# para permitir que funções retornem múltiplos valores.
- Python
    - Está intimamente relacionado com suas listas, mas é imutável.
    - Criado com um literal de tupla:
    - `myTuple = (3, 5.8, 'apple')`

    Referenciado com subscritos (começando em 1).

    Concatenado com + e deletado com del.

Tuple Types (continued)

- ML

```
val myTuple = (3, 5.8, 'apple');
```

  - Access as follows:

`#1(myTuple)` is the first element

  - A new tuple type can be defined

```
type intReal = int * real;
```

- F#

```
let tup = (3, 5, 7)
let a, b, c = tup
```

This assigns a tuple to a tuple pattern `(a, b, c)`

List Types

- Listas em Lisp e Scheme são delimitadas por parênteses e não usam vírgulas:
- `(A B C D)` and `(A (B C) D)`
- Dados e código têm a mesma forma:
  - As data, `(A B C)` is literally what it is
  - As code, `(A B C)` is the function `A` applied to the parameters `B` and `C`
- O interpretador precisa saber qual é qual, então se é um dado, nós o citamos com um apóstrofo:
- `'(A B C)` is data

List Types (continued)

- List Operations in Scheme
  - `CAR` returns the first element of its list parameter

    `(CAR '(A B C))` returns `A`
  - `CDR` returns the remainder of its list parameter after the first element has been removed

    `(CDR '(A B C))` returns `(B C)`
  - `CONS` puts its first parameter into its second parameter, a list, to make a new list

    `(CONS 'A (B C))` returns `(A B C)`
  - `LIST` returns a new list of its parameters

    `(LIST 'A 'B '(C D))` returns `(A B (C D))`

List Types (continued)

- List Operations in ML
  - Lists are written in brackets and the elements are separated by commas
  - List elements must be of the same type
  - The Scheme `CONS` function is a binary operator in ML, `::`

    `3 :: [5, 7, 9]` evaluates to `[3, 5, 7, 9]`
  - The Scheme `CAR` and `CDR` functions are named `hd` and `tl`, respectively

List Types (continued)

- F# Lists
  - Like those of ML, except elements are separated by semicolons and `hd` and `tl` are methods of the `List` class
- Python Lists
  - The list data type also serves as Python's arrays
  - Unlike Scheme, Common Lisp, ML, and F#, Python's lists are mutable
  - Elements can be of any type
  - Create a list with an assignment

```
myList = [3, 5.8, "grape"]
```

List Types (continued)

- Python Lists (continued)
  - List elements are referenced with subscripting, with indices beginning at zero

    `x = myList[1]` Sets `x` to `5.8`
  - List elements can be deleted with `del`

    `del myList[1]`
  - List Comprehensions – derived from set notation

    `[x * x for x in range(6) if x % 3 == 0]`

    `range(12)` creates `[0, 1, 2, 3, 4, 5, 6]`

    Constructed list: `[0, 9, 36]`

List Types (continued)

- Haskell's List Comprehensions
  - The original

```
[n * n | n <- [1..10]]
```

- F#'s List Comprehensions

```
let myArray = [|for i in 1 .. 5 -> [i * i) |]
```

- Both C# and Java supports lists through their generic heap-dynamic collection classes, `List` and `ArrayList`, respectively

Unions Types

- Uma união é um tipo cujas variáveis podem armazenar valores de tipos diferentes em momentos diferentes durante a execução.
- Uma questão de design relacionada é: a verificação de tipo deve ser exigida?

Discriminated vs. Free Unions

- C e C++ fornecem construções de união (union) nas quais não há suporte de linguagem para verificação de tipo; a união nessas linguagens é chamada de união livre (free union).
- A verificação de tipo das uniões requer que cada união inclua um indicador de tipo chamado discriminante. Isso é suportado por ML, Haskell e F#.

Unions in F#

- Defined with a type statement using OR

```
type intReal =
    | IntValue of int
    | RealValue of float;;
```

`intReal` is the new type

`IntValue` and `RealValue` are constructors

To create a value of type `intReal`:

```
let ir1 = IntValue 17;;
let ir2 = RealValue 3.4;;
```

Unions in F# (continued)

- Acessar o valor de uma união é feito com o casamento de padrões

```
match pattern with
    | expression_list1 -> expression1
    | ...
    | expression_listn -> expressionn
```

O padrão pode ser qualquer tipo de dado.

A lista de expressões pode conter curingas (_).

Unions in F# (continued)

Example:

```
let a = 7;;
let b = "grape";;
let x = match (a, b) with
      | 4, "apple" -> apple
      | _, "grape" -> grape
      | _ -> fruit;;
```

Unions in F# (continued)

To display the type of the `intReal` union:

```
let printType value =
  match value with
      | IntVale value -> printfn "int"
      | RealValue value -> printfn "float";;
```

If `ir1` and `ir2` are defined as previously,

`printType ir1` returns `int`

`printType ir2` returns `float`

Evaluation of Unions

- Unions livres são inseguras e não permitem verificação de tipo.
- Java e C# não suportam uniões, refletindo crescentes preocupações com segurança nas linguagens de programação.

Pointer and Reference Types

- Uma variável do tipo ponteiro possui um intervalo de valores que consiste em endereços de memória e um valor especial, nil.
- Isso proporciona o poder do endereçamento indireto e oferece uma forma de gerenciar a memória dinâmica.
- Um ponteiro pode ser usado para acessar uma localização na área onde o armazenamento é criado dinamicamente, geralmente chamada de heap.

Design Issues of Pointers

- Qual é o escopo e a duração de uma variável de ponteiro?
- Qual é a duração de uma variável de heap-dinâmica?
- Os ponteiros são restritos quanto ao tipo de valor para o qual podem apontar?
- Os ponteiros são usados para gerenciamento de armazenamento dinâmico, endereçamento indireto ou ambos?
- A linguagem deve suportar tipos de ponteiro, tipos de referência ou ambos?

Pointer Operations

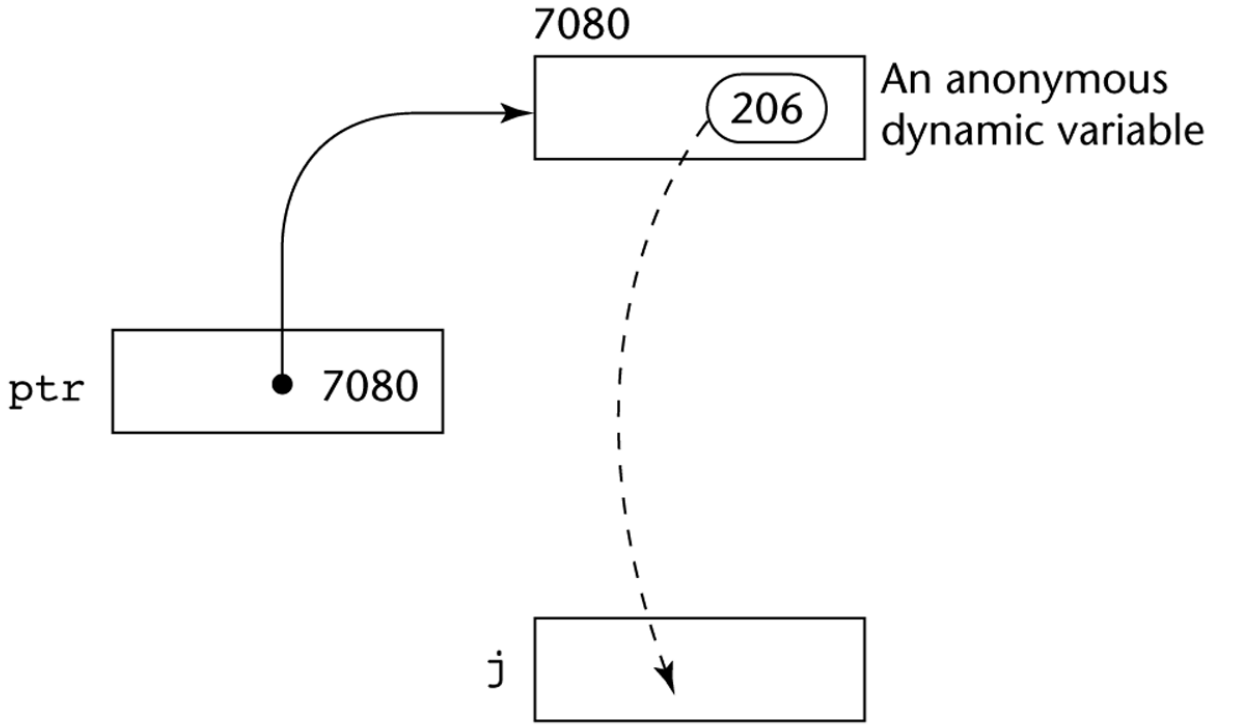
- Duas operações fundamentais: atribuição e desreferenciamento.
- A atribuição é usada para definir o valor de uma variável de ponteiro para algum endereço útil.
- O desreferenciamento retorna o valor armazenado no local representado pelo valor do ponteiro.
- O desreferenciamento pode ser explícito ou implícito.
- Em C++, é usada uma operação explícita via *:

```
j = *ptr
```

define j como o valor localizado em ptr.

Pointer Assignment Illustrated

The assignment operation j = *ptr



Problems with Pointers

- Ponteiros pendentes (perigosos)
  - Um ponteiro aponta para uma variável de heap dinâmica que foi desalocada
- Variável de heap dinâmica perdida
  - Uma variável de heap dinâmica alocada que não está mais acessível ao programa do usuário (frequentemente chamada de lixo)
    - O ponteiro p1 é definido para apontar para uma variável de heap dinâmica recém-criada
    - O ponteiro p1 é posteriormente definido para apontar para outra variável de heap dinâmica recém-criada
    - O processo de perder variáveis de heap dinâmicas é chamado de vazamento de memória

Pointers in C and C++

- Extremamente flexíveis, mas devem ser usados com cuidado.
- Ponteiros podem apontar para qualquer variável, independentemente de quando ou onde foi alocada.
- Usados para gerenciamento de armazenamento dinâmico e endereçamento.
- Aritmética de ponteiros é possível.
- Operadores explícitos de desreferenciamento e endereço.
- O tipo de domínio não precisa ser fixo (void *).

  `void *` can point to any type and can be type checked (cannot be de-referenced)

Pointer Arithmetic in C and C++

```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` is equivalent to `stuff[5]` and `p[5]`

`*(p+i)` is equivalent to `stuff[i]` and `p[i]`

Reference Types

- C++ inclui um tipo especial de ponteiro chamado tipo de referência, que é usado principalmente para parâmetros formais.
  - As vantagens tanto do passagem por referência quanto do passagem por valor.
- Java estende as variáveis de referência de C++ e permite que elas substituam ponteiros completamente.
  - As referências são referências a objetos, em vez de serem endereços.
- C# inclui tanto as referências de Java quanto os ponteiros de C++.

Evaluation of Pointers

- Ponteiros pendentes e objetos pendentes são problemas, assim como a gestão de heap.
- Ponteiros são como comandos de "goto" – eles ampliam o alcance das células que podem ser acessadas por uma variável.
- Ponteiros ou referências são necessários para estruturas de dados dinâmicas – portanto, não podemos projetar uma linguagem sem eles.

Representations of Pointers

- Computadores de grande porte usam valores únicos para acessar a memória.
- Processadores Intel usam segmento e deslocamento (offset).

Dangling Pointer Problem

- Tombstone: uma célula extra no heap que é um ponteiro para a variável dinâmica de heap.
  - A variável de ponteiro real aponta apenas para as lápides.
  - Quando a variável de heap dinâmica é desalocada, a lápide permanece, mas é definida como nil.
  - Custo elevado em tempo e espaço.
- . Trava e chave: Os valores dos ponteiros são representados como pares (chave, endereço).
  - As variáveis de heap dinâmicas são representadas como variáveis mais células para o valor de travamento inteiro.
  - Quando a variável de heap dinâmica é alocada, o valor de travamento é criado e colocado na célula de travamento e na célula de chave do ponteiro.

Heap Management

- Um processo em tempo de execução muito complexo.
- Células de tamanho único versus células de tamanho variável.
- Duas abordagens para recuperar lixo.
  - Contadores de referência (abordagem ávida): a recuperação é gradual.
  - Marca e varredura (abordagem preguiçosa): a recuperação ocorre quando a lista de espaço variável fica vazia.

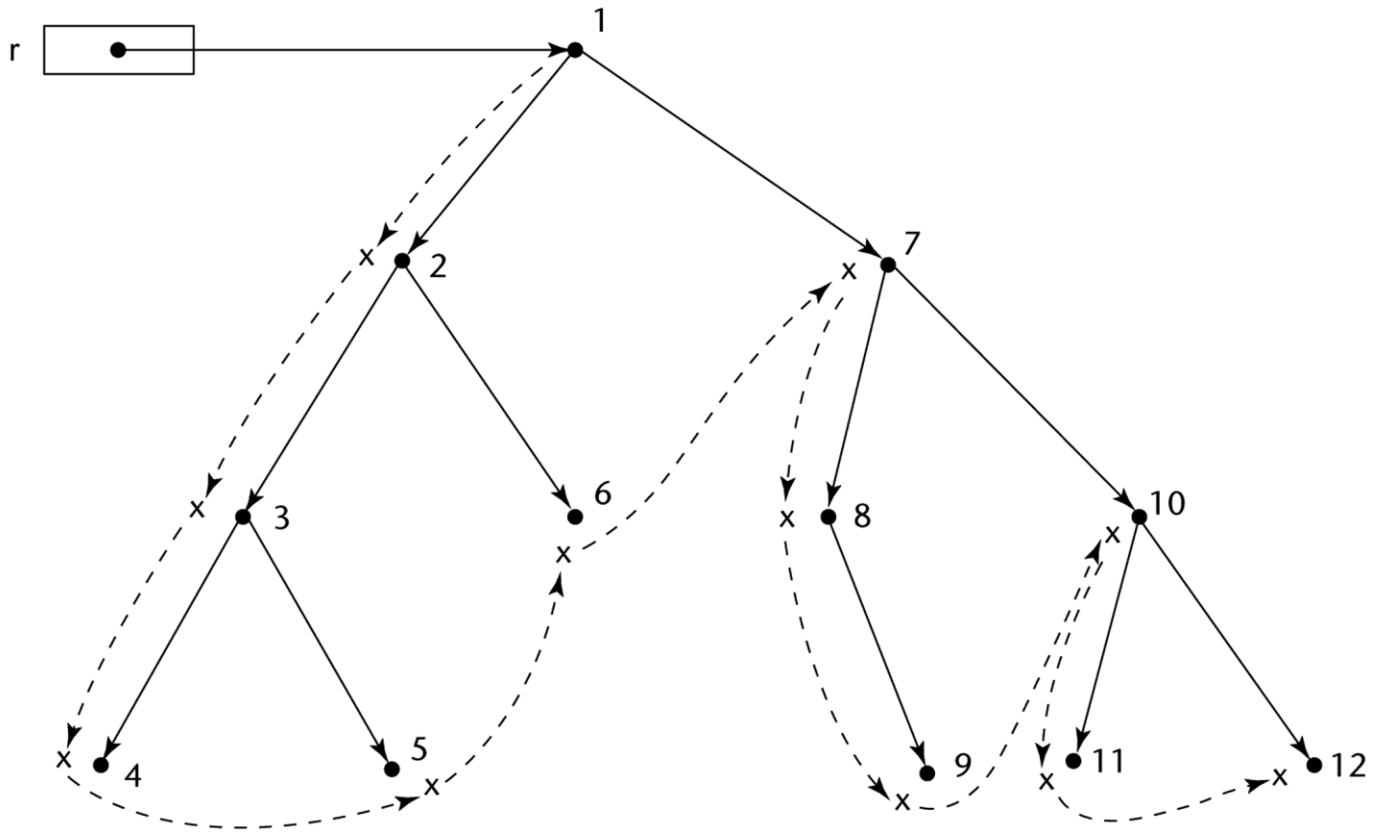
Reference Counter

- Contadores de referência: mantêm um contador em cada célula que armazena o número de ponteiros atualmente apontando para a célula.
  - Desvantagens: espaço requerido, tempo de execução necessário, complicações para células conectadas circularmente.
  - Vantagem: é intrinsecamente incremental, portanto, atrasos significativos na execução do aplicativo são evitados.

Mark-Sweep

- O sistema em tempo de execução aloca células de armazenamento conforme solicitado e desconecta ponteiros das células conforme necessário; então o algoritmo de marcação e varredura começa.
  - Cada célula no heap tem um bit extra usado pelo algoritmo de coleta.
  - Todas as células inicialmente são marcadas como lixo.
  - Todos os ponteiros são rastreados no heap e as células alcançáveis são marcadas como não lixo.
  - Todas as células de lixo são retornadas à lista de células disponíveis.
  - Desvantagens: em sua forma original, era feito muito raramente. Quando feito, causava atrasos significativos na execução do aplicativo. Algoritmos de marcação e varredura contemporâneos evitam isso fazendo-o com mais frequência - chamado de marcação e varredura incremental.

Marking Algorithm



Dashed lines show the order of node_marking

Variable-Size Cells

- Todas as dificuldades das células de tamanho único, além de outras mais.
- Requerido pela maioria das linguagens de programação.
- Se o método mark-sweep for usado, problemas adicionais ocorrem.
  - A configuração inicial dos indicadores de todas as células no heap é difícil.
  - O processo de marcação não é trivial.
  - Manter a lista de espaço disponível é outra fonte de sobrecarga.

Type Checking

- Vamos generalizar o conceito de operandos e operadores para incluir subprogramas e atribuições:

- A verificação de tipo é a atividade de garantir que os operandos de um operador sejam de tipos compatíveis.

- Um tipo compatível é aquele que é legal para o operador ou é permitido, pelas regras da linguagem, ser convertido implicitamente, pelo código gerado pelo compilador, para um tipo legal.
  - Essa conversão automática é chamada de coerção.

- Um erro de tipo ocorre quando um operador é aplicado a um operando de um tipo inadequado.

Type Checking (continued)

- Se todas as vinculações de tipo forem estáticas, praticamente toda verificação de tipo pode ser estática.
- Se as vinculações de tipo forem dinâmicas, a verificação de tipo deve ser dinâmica.
- Uma linguagem de programação é fortemente tipada se os erros de tipo forem sempre detectados.
- Vantagem da tipagem forte: permite a detecção dos usos incorretos de variáveis que resultam em erros de tipo.

Strong Typing

Exemplos de linguagens:

- C e C++ não são totalmente fortemente tipadas: a verificação de tipo de parâmetro pode ser evitada; uniões não são verificadas quanto ao tipo.
- Java e C# são quase fortemente tipadas (devido ao casting explícito).
- ML e F# são fortemente tipadas.

Strong Typing (continued)

- As regras de coerção afetam fortemente a tipagem forte – elas podem enfraquecê-la consideravelmente (como em C++ em comparação com ML e F#).

- Embora Java tenha apenas metade das coerções de atribuição de C++, sua tipagem forte ainda é muito menos eficaz do que a de Ada.

Name Type Equivalence

- A equivalência de tipo por nome significa que duas variáveis têm tipos equivalentes se estiverem em uma mesma declaração ou em declarações que usam o mesmo nome de tipo.
- É fácil de implementar, mas altamente restritiva:
  - Subintervalos de tipos inteiros não são equivalentes a tipos inteiros.
  - Parâmetros formais devem ser do mesmo tipo que seus parâmetros reais correspondentes.

Structure Type Equivalence

- A equivalência estrutural de tipos significa que duas variáveis têm tipos equivalentes se seus tipos tiverem estruturas idênticas.
- É mais flexível, mas mais difícil de implementar.

Type Equivalence (continued)

- Ao considerar o problema de dois tipos estruturados:
  - Dois tipos de registros são equivalentes se forem estruturalmente iguais, mas usarem nomes de campos diferentes?
  - Dois tipos de arrays são equivalentes se forem iguais, exceto pelos subscritos serem diferentes? (por exemplo, [1..10] e [0..9])
  - Dois tipos de enumeração são equivalentes se seus componentes forem soletrados de forma diferente?
  - Com a equivalência estrutural de tipos, você não pode diferenciar entre tipos da mesma estrutura (por exemplo, unidades de velocidade diferentes, ambas float).

Theory and Data Types

- A teoria dos tipos é uma área ampla de estudo em matemática, lógica, ciência da computação e filosofia.
- Dois ramos da teoria dos tipos na ciência da computação:
  - Prático: tipos de dados em linguagens comerciais.
  - Abstrato: cálculo lambda tipado.
- Um sistema de tipos é um conjunto de tipos e as regras que governam seu uso em programas.

Theory and Data Types (continued)

- Um modelo formal de um sistema de tipos é um conjunto de tipos e uma coleção de funções que definem as regras de tipo.
  - Tanto uma gramática de atributos quanto um mapa de tipos poderiam ser usados para as funções.
  - Mapeamentos finitos – modelam arrays e funções.
  - Produtos cartesianos – modelam tuplas e registros.
  - União de conjuntos – modela tipos de união.
  - Subconjuntos – modela subtipos.

Resumo

- Os tipos de dados de uma linguagem são uma grande parte do que determina o estilo e a utilidade dessa linguagem.
- Os tipos de dados primitivos da maioria das linguagens imperativas incluem tipos numéricos, de caracteres e booleanos.
- Os tipos de enumeração e subintervalo definidos pelo usuário são convenientes e contribuem para a legibilidade e confiabilidade dos programas.
- Arrays e registros estão incluídos na maioria das linguagens.
- Ponteiros são usados para flexibilidade de endereçamento e para controlar a gestão dinâmica de armazenamento.

QUESTÕES – AULA 5

Q1. Quais são os argumentos a favor e contra a representação de valores booleanos como bits únicos em memória?

Q2. Como um valor decimal perde espaço em memória?

Q3. Que desvantagens existem no desreferenciamento implícito de ponteiros, mas apenas em certos contextos? Por exemplo, considere a desreferência implícita de um ponteiro para um registro em Ada quando ele é usado para referenciar um campo de um registro.

Q4. Escreva uma pequena discussão sobre o que foi perdido e o que foi ganho na decisão dos projetistas de Java de não incluírem os ponteiros de C++.

Q5. De que forma a verificação de tipos estática é melhor do que a verificação de tipos dinâmica?

QUESTÕES – AULA 5

Resposta Q1.
Variáveis booleanas armazenadas como bits individuais são muito eficientes em termos de espaço, mas na maioria dos computadores o acesso a elas é mais lento do que se fossem armazenadas como bytes.

Resposta Q2.
Valores inteiros armazenados em decimal desperdiçam espaço em computadores de memória binária, simplesmente porque são necessários quatro bits binários para armazenar um único dígito decimal, mas esses quatro bits são capazes de armazenar 16 valores diferentes. Portanto, a capacidade de armazenar seis de cada 16 valores possíveis é desperdiçada. Valores numéricos podem ser armazenados de forma eficiente em computadores de memória binária apenas em bases numéricas que são múltiplos de 2. Se os seres humanos tivessem desenvolvido mãos com um número de dedos que fosse uma potência de 2, esse tipo de problema não ocorreria.

Resposta Q3.
Quando o desreferenciamento implícito de ponteiros ocorre apenas em certos contextos, isso torna a linguagem ligeiramente menos ortogonal. O contexto da referência ao ponteiro determina seu significado. Isso diminui a legibilidade da linguagem e a torna ligeiramente mais difícil de aprender.

Resposta Q4.
A recuperação implícita de armazenamento na heap elimina a criação de ponteiros pendurados por meio de operações explícitas de desalocação, como delete. A desvantagem da recuperação implícita de armazenamento na heap é o custo de tempo de execução da recuperação, muitas vezes quando nem mesmo é necessário (não há escassez de armazenamento na heap).

Resposta Q5.
Uma linguagem que permite muitas coerções de tipo pode enfraquecer o efeito benéfico da tipagem forte ao permitir que muitos erros de tipo potenciais sejam mascarados simplesmente coercendo o tipo de um operando de seu tipo incorreto fornecido na declaração para um tipo aceitável, em vez de relatá-lo como um erro.

PROVA 1 - 03/4/2024

CEUB

EDUCAÇÃO SUPERIOR

ceub.br