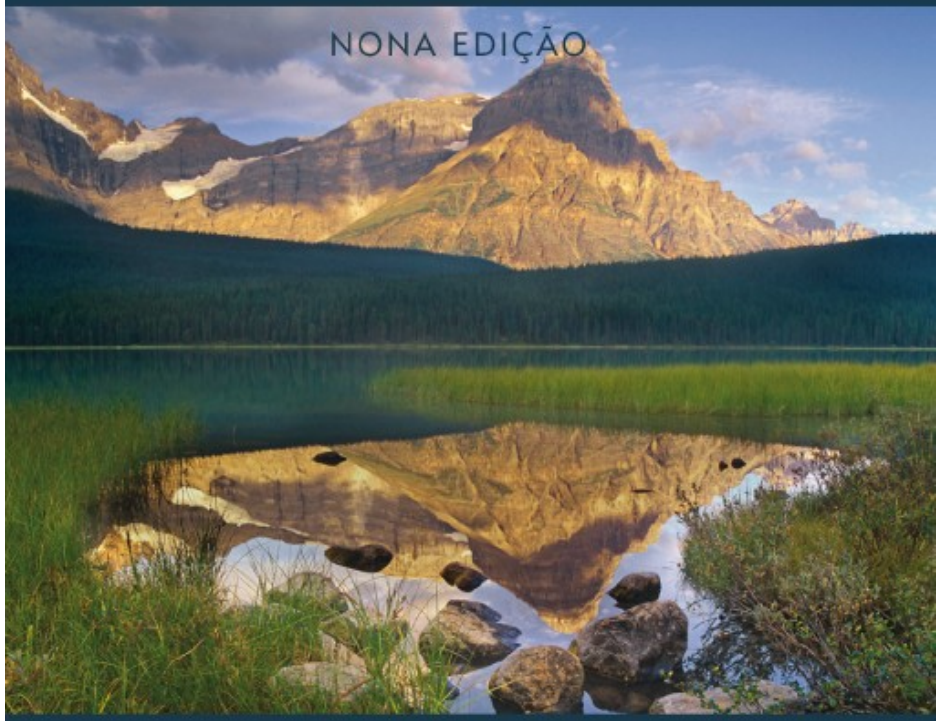
CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

NONA EDIÇÃO

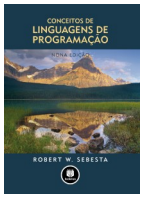


ROBERT W. SEBESTA

bookman

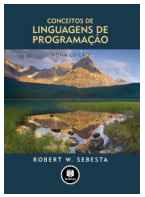
Capítulo 6

Tipos de Dados



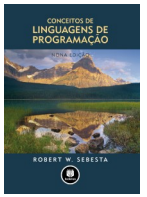
Tópicos do Capítulo 6

- Introdução
- Tipos de dados primitivos
- Cadeias de caracteres
- Tipos ordinais definidos pelo usuário
- Tipos de matrizes
- Matrizes associativas
- Registros
- Uniões
- Ponteiros e referências



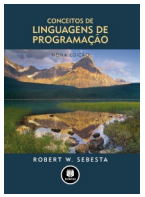
Introdução

- Um *tipo* de dados define uma coleção de valores de dados e um conjunto de operações pré-definidas sobre eles
- Um *descritor* é a coleção de atributos de uma variável
- Um *objeto* representa uma instância de um tipo de dados definido pelo usuário
- Uma questão de projeto para todos os tipos de dados: Que operações são fornecidas para variáveis do tipo e como elas são especificadas?



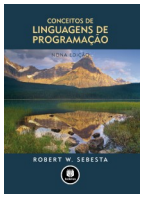
Tipos de dados primitivos

- Praticamente 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
- Alguns dos tipos primitivos são meramente reflexos de hardware
- Outros requerem apenas um pouco de suporte externo ao hardware para sua implementação



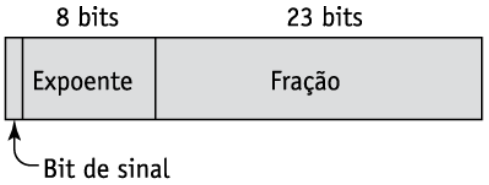
Tipos de dados primitivos: inteiro

- Quase sempre um reflexo exato do hardware, logo o mapeamento é simples
- Muitos computadores suportam diversos tipos de tamanhos inteiros
- Java inclui quatro tamanhos inteiros com sinal: `byte, short, int, long`

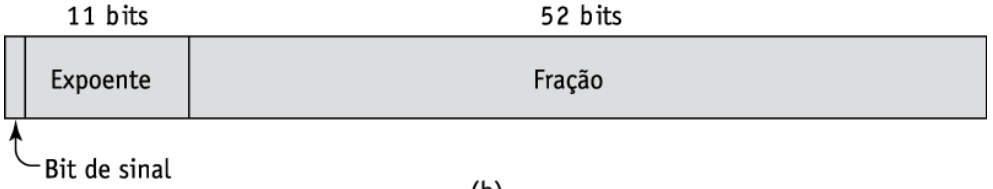


Tipos de dados primitivos: ponto flutuante

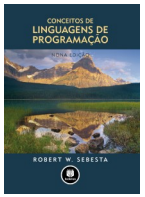
- Modelam números reais, mas as representações são apenas aproximações
- Linguagens para uso científico suportam pelo menos dois tipos de ponto flutuante (por exemplo, `float` e `double`)
- IEEE Padrão de Ponto Flutuante 754



(a)



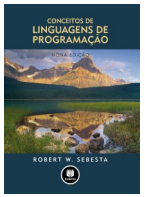
(b)



Tipos de dados primitivos: complexo

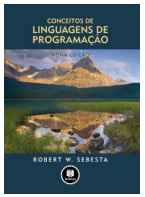
- Algumas linguagens suportam um tipo de dados complexo – por exemplo, Fortran e Python
- Valores complexos são representados como pares ordenados de valores de ponto flutuante
- Literal complexo (em Python):

  `(7 + 3j)`, onde `7` é a parte real e `3` é a parte imaginária



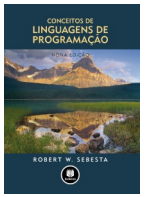
Tipos de dados primitivos: decimal

- Para aplicações de sistemas de negócios
  - Essencial para COBOL
  - C# tem um tipo de dados decimal
- Decimais codificados em binário (BCD)
- *Vantagem*: precisão
- *Desvantagens*: faixa de valores restrita, gasto de memória



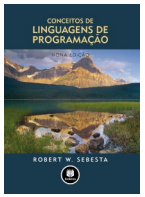
Tipos de dados primitivos: booleanos

- Mais simples de todos
- Faixa de valores: dois elementos, um para “verdadeiro” e um para “falso”
- Poderiam ser representados por bits, mas são armazenados em bytes
  - Vantagem: legibilidade



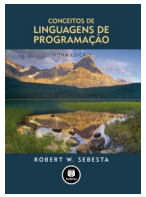
Tipos de dados primitivos: caractere

- Armazenados como codificações numéricas
- Codificação mais usada: ASCII
- Uma alternativa, conjunto de caracteres de 16 bits: Unicode (UCS-2)
  - Inclui caracteres da maioria das linguagens naturais
  - Originalmente usado em Java
  - C# e JavaScript também suportam Unicode
- Unicode 32 bits (UCS-4)
  - Suportada por Fortran, começando com 2003



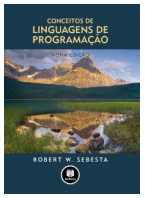
Cadeias de caracteres

- Valores são sequências de caracteres
- Questões de projeto:
  - As cadeias devem ser apenas um tipo especial de vetor de caracteres ou um tipo primitivo?
  - As cadeias devem ter tamanho estático ou dinâmico?



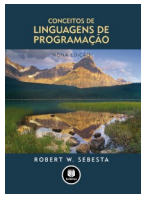
Cadeias e suas operações

- Operações comuns:
  - Atribuição
  - Comparação (=, > etc.)
  - Concatenação
  - Referência a subcadeias
  - Casamento de padrões



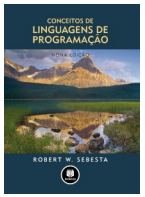
Cadeias de caracteres em certas linguagens

- C e C++
  - Não são definidas como primitivas
  - Usam matrizes de caracteres **`char`** e uma biblioteca de funções que fornecem operações
- SNOBOL4
  - Primitivas
  - Várias operações, incluindo casamentos de padrões
- Fortran e Python
  - Tipo primitivo com atribuição e diversas operações
- Java
  - Primitiva via classe `String`
- Perl, JavaScript, Ruby e PHP
  - Incluem operações de casamento de padrões pré-definidas, usando expressões regulares



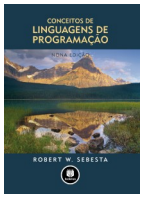
Opções de tamanho de cadeias

- *Estático*: COBOL, classe pré-definida `String`
- *Dinâmico limitado*: C and C++
  - Nessas linguagens, um caractere especial é usado para indicar o fim da cadeia de caracteres
- *Dinâmico*: SNOBOL4, Perl, JavaScript
- Ada 95 suporta as três opções



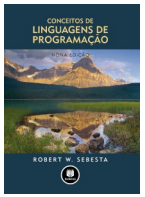
Avaliação

- Importantes para a facilidade de escrita
- Como tipos primitivos de tamanho estático não são custosos, por que não tê-los em uma linguagem?
- Tamanho dinâmico é interessante, mas vale o custo?

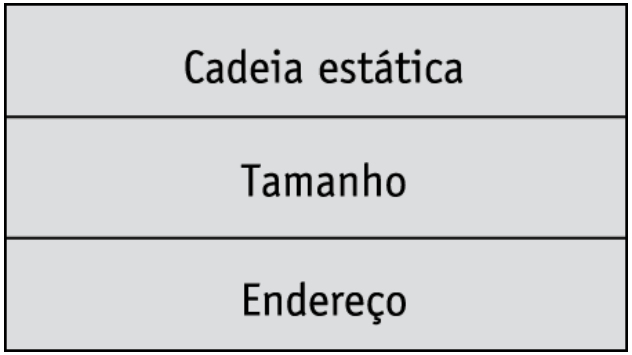


Implementação de cadeias de caracteres

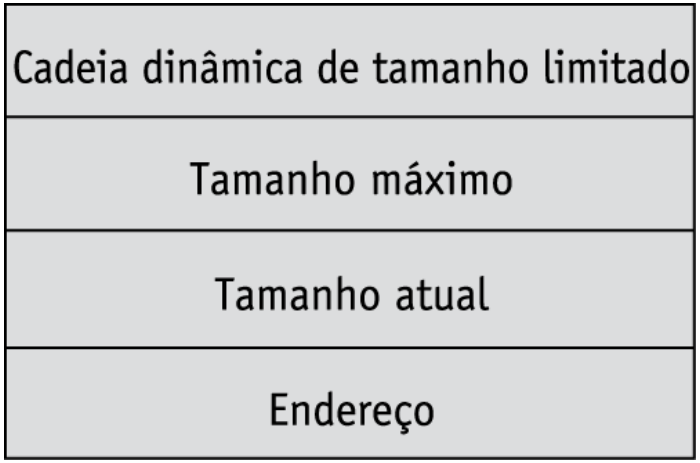
- Tamanho estático: descritor em tempo de compilação
- Tamanho dinâmico limitado: pode necessitar de um descritor em tempo de execução (mas não em C e C++)
- Tamanho dinâmico: necessita de descritor em tempo de execução; alocação/liberação é o maior problema de implementação



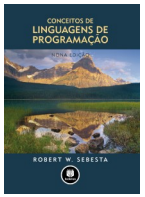
Descritores em tempo de compilação e de execução



Descritor em tempo de compilação para cadeias estáticas

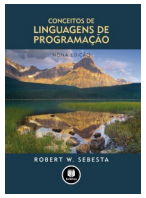


Descritor em tempo de execução para cadeias dinâmicas de tamanho limitado



Tipos ordinais definidos pelo usuário

- Um tipo ordinal é um no qual a faixa de valores possíveis pode ser facilmente associada com o conjunto de inteiros positivos
- Exemplos de tipos primitivos ordinais em Java
  - `integer`
  - `char`
  - `boolean`

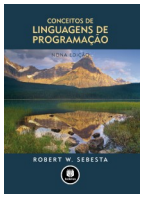


Tipos enumeração

- Todos os valores possíveis, os quais são constantes nomeadas, na definição
- Exemplo em C#

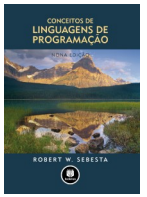
```
enum days {mon, tue, wed, thu, fri, sat, sun};
```

- Questões de projeto
  - Uma constante de enumeração pode aparecer em mais de uma definição de tipo? Se pode, como o tipo de uma ocorrência de tal constante é verificado no programa?
  - Os valores de enumeração são convertidos para inteiros?
  - Existem outros tipos que são convertidos para um tipo enumeração?



Avaliação de tipos enumeração

- Melhora a legibilidade, por exemplo, não precisa codificar uma cor como um número
- Melhora a confiabilidade, por exemplo, compilador pode verificar:
  - operações (não permitir que as cores sejam adicionadas)
  - Nenhuma variável de enumeração pode ter um valor atribuído fora do intervalo definido
  - Ada, C# e Java 5.0 fornecem melhor suporte para enumeração do que C++, porque variáveis do tipo enumeração nessas linguagens não são convertidas para tipos inteiros

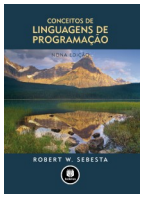


Tipos subfaixa

- Uma subsequência contígua de um tipo ordinal
  - Exemplo: 12.18 é uma subfaixa do tipo inteiro
- Projeto de Ada

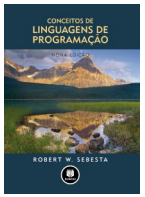
```
type Days is (mon, tue, wed, thu, fri, sat, sun);
subtype Weekdays is Days range mon..fri;
subtype Index is Integer range 1..100;

Day1: Days;
Day2: Weekday;
Day2 := Day1;
```



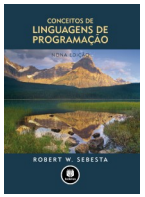
Avaliação do tipo subfaixa

- Melhora a legibilidade
  - Torna claro aos leitores que as variáveis de subtipos podem armazenar apenas certas faixas de valores
- Melhora a confiabilidade
  - A atribuição de um valor a uma variável de subfaixa que está fora da faixa especificada é detectada como um erro



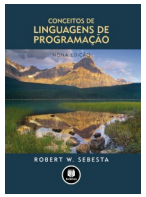
Implementação de tipos ordinais definidos pelo usuário

- Tipos enumeração são implementados como inteiros
- Tipos subfaixas são implementados como seus tipos ancestrais, exceto que as verificações de faixas devem ser implicitamente incluídas pelo compilador em cada atribuição de uma variável ou de uma expressão a uma variável subfaixa



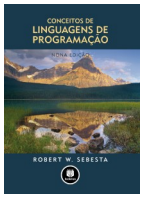
Tipos matrizes

- Uma matriz é um agregado homogêneo de elementos de dados no qual um elemento individual é identificado por sua posição na agregação, relativamente ao primeiro elemento



Questões de projeto de matrizes

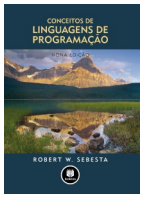
- Que tipos são permitidos para índices?
- As expressões de índices em referências a elementos são verificadas em relação à faixa?
- Quando as faixas de índices são vinculadas?
- Quando ocorre a liberação da matriz?
- As matrizes multidimensionais irregulares ou retangulares são permitidas?
- As matrizes podem ser inicializadas quando elas têm seu armazenamento alocado?
- Que tipos de fatias são permitidas?



Matrizes e índices

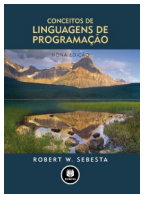
- *Indexar* (ou subscrever) é um mapeamento de índices para elementos

  `nome_matriz (lista_valores_índices)` → `elemento`
- Sintaxe de índices
  - FORTRAN, PL/I e Ada usam parênteses
    - Ada explicitamente usa parênteses para mostrar a uniformidade entre as referências matriz e chamadas de função, pois ambos são mapeamentos
  - A maioria das outras linguagens usa chaves



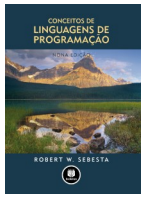
Tipos de índices de matrizes

- FORTRAN, C: apenas inteiros
- Ada: inteiro ou enumeração
- Java: apenas tipos inteiros
- Verificação de faixas de índices
  - C, C++, Perl e Fortran não especificam faixas de índices
  - Java, ML e C# especificam faixas de índices
  - Em Ada, o padrão é exigir a verificação de faixas de índice, mas pode ser desligada



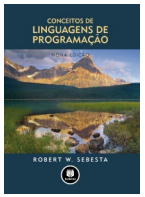
Vinculações de índices e categorias de matrizes

- Estática: é uma na qual as faixas de índices são vinculadas estaticamente e a alocação de armazenamento é estática (antes do tempo de execução)
  - Vantagem: eficiência (sem alocação dinâmica)
- *Dinâmica da pilha fixa*: é uma na qual as faixas de índices são vinculadas estaticamente, mas a alocação é feita em tempo de elaboração da declaração
  - Vantagem: eficiência de espaço



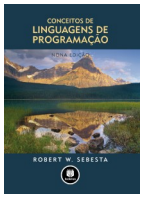
Vinculações de índices e categorias de matrizes

- *Dinâmica da pilha*: é uma na qual tanto as faixas de índices quanto a alocação de armazenamento são vinculadas dinamicamente em tempo de elaboração
  - Vantagem: flexibilidade (o tamanho da matriz não precisa ser conhecido até ela ser usada)
- *Dinâmica do monte fixa*: similar a uma dinâmica da pilha fixa, no sentido de que as faixas de índices e a vinculação ao armazenamento são fixas após este ser alocado (mas a partir do monte, em vez de a partir da pilha, e em tempo de execução)



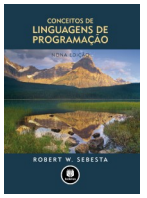
Vinculações de índices e categorias de matrizes

- *Dinâmica do monte*: é uma na qual a vinculação das faixas de índices e da alocação de armazenamento é dinâmica e pode mudar qualquer número de vezes durante seu tempo de vida
  - Vantagem: flexibilidade (matrizes podem crescer e encolher durante a execução de um programa)



Vinculações de índices e categorias de matrizes

- Matrizes C e C++ que incluem o modificador `static` são estáticas
- Matrizes C e C++ sem o modificador `static` são dinâmicas da pilha fixas
- C e C++ também fornecem matrizes dinâmicas do monte fixas
- C# inclui uma segunda classe de matrizes, `ArrayList`, que fornece matrizes dinâmicas da pilha
- Perl, JavaScript, Python e Ruby suportam matrizes dinâmicas do monte



Inicialização de matrizes

- Algumas linguagens fornecem os meios para inicializar matrizes no momento em que seu armazenamento é alocado
  - Exemplo em C, C++, Java, C#

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

  - Cadeias de caracteres em C e C++

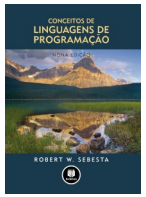
```
char name [] = “freddie”;
```

  - Matrizes de cadeias em C e C++

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

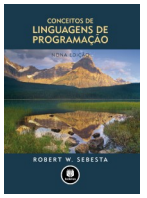
  - Inicialização de objetos String em Java

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



Matrizes heterogêneas

- Uma *matriz heterogênea* é uma em que os elementos não precisam ser do mesmo tipo
- Suportadas por Perl, Python, JavaScript e Ruby

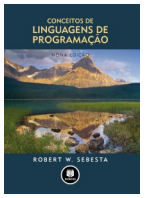


Inicialização de matrizes

- Linguagens baseadas em C
  - `int list [] = {1, 3, 5, 7}`
  - `char *names [] = {“Mike”, “Fred”,“Mary Lou”};`
- Ada
  - `List : array (1..5) of Integer :=`
    `(1 => 17, 3 => 34, others => 0);`
- Python
  - Compreensões de lista

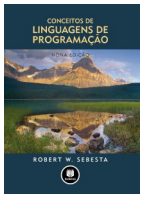
```
list = [x ** 2 for x in range(12) if x % 3 == 0]
```

puts `[0, 9, 36, 81]` in `list`



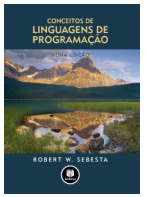
Operações de matrizes

- APL é a linguagem de processamento de matrizes mais poderosa já desenvolvida. As quatro operações aritméticas básicas são definidas para vetores (matrizes de dimensão única) e para matrizes, bem como operadores escalares
- Ada permite atribuição de matrizes, mas também concatenação
- Python fornece atribuição de matrizes, apesar de ser apenas uma mudança de referência. Python também suporta operações para concatenação de matrizes e para verificar se um elemento pertence à matriz
- Ruby também fornece concatenação de matrizes
- Fortran inclui operações *elementais* porque elas ocorrem entre pares de elementos de matrizes



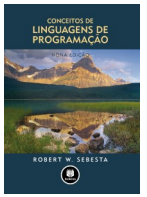
Matrizes retangulares e irregulares

- Uma matriz retangular é uma multidimensional na qual todas as linhas e colunas têm o mesmo número de elementos
- Na matriz irregular, o tamanho das linhas não precisa ser o mesmo
  - Possíveis quando as multidimensionais são matrizes de matrizes
- C, C++ e Java suportam matrizes irregulares
- Fortran, Ada e C# suportam matrizes retangulares (C# também suporta irregulares)



Fatias

- Uma fatia é alguma subestrutura de uma matriz; nada mais do que um mecanismo de referência
- Fatias são úteis apenas em linguagens que têm operações de matrizes



Exemplos de fatias

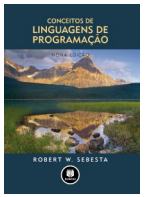
- Fortran 95

```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3) :: Cube
```

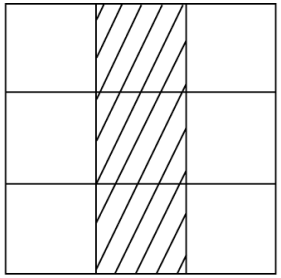
`Vector (3:6)` é uma matriz de quatro elementos

- Ruby suporta fatias com o método `slice`

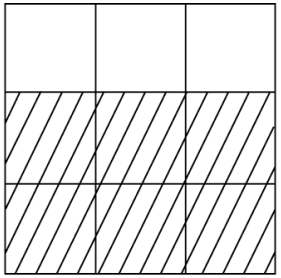
`list.slice(2, 2)` retorna o terceiro e o quarto elementos de `list`



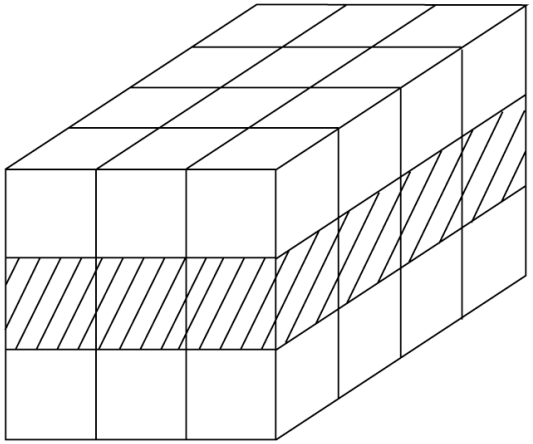
Exemplos de fatias em Fortran 95



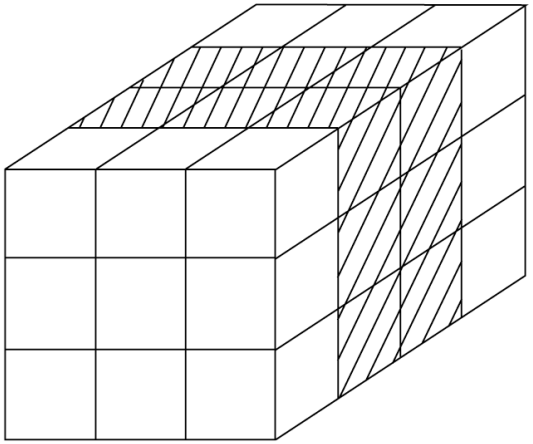
MAT (1:3, 2)



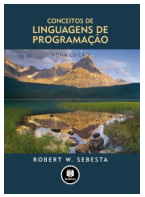
MAT (2:3, 1:3)



CUBE (2, 1:3, 1:4)



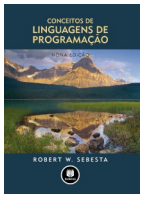
CUBE (1:3, 1:3, 2:3)



Implementação de matrizes

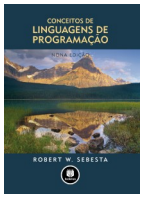
- Função de acesso mapeia expressões subscritas para um endereço na matriz
- Função de acesso para para list:

  endereço(list[k]) = endereço (list[0])
  + k * tamanho_do_elemento)



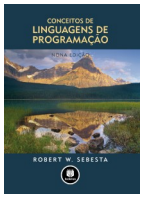
Acessando matrizes multidimensionais

- Duas maneiras:
  - Ordem principal de linhas – usada na maioria das linguagens
  - Ordem principal de coluna – usada em Fortran

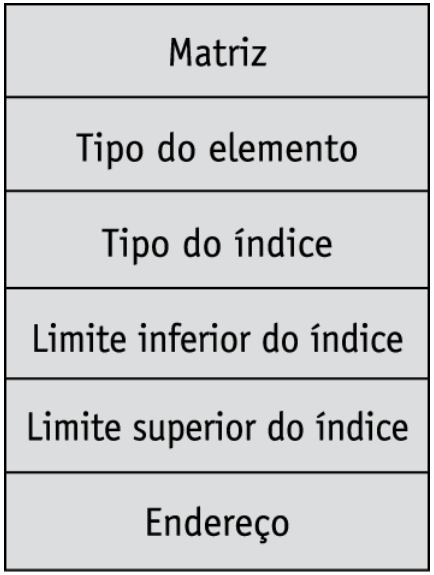


Localizando um elemento em uma matriz multidimensional

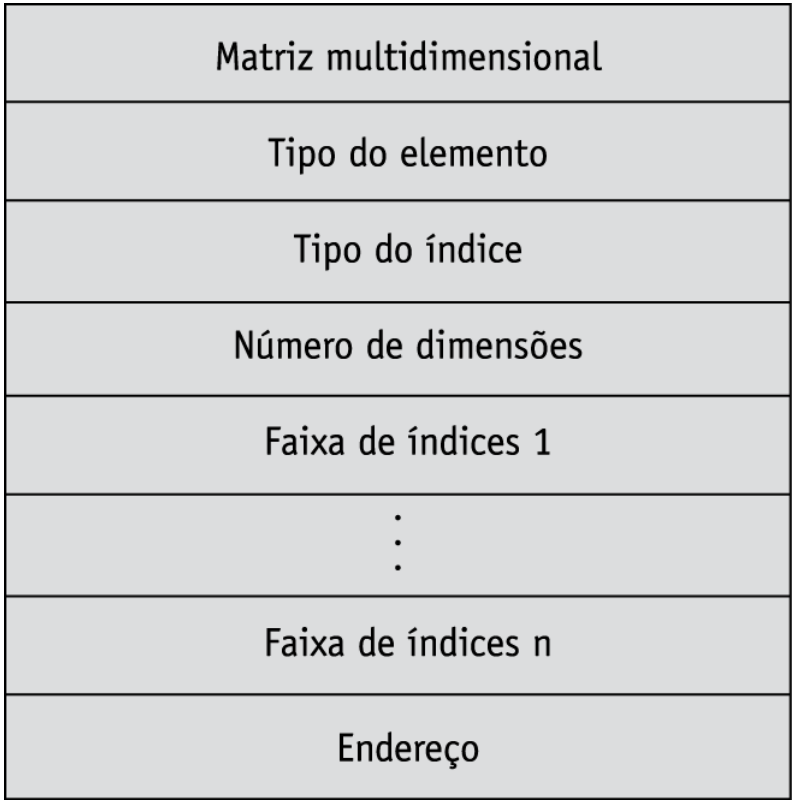
- Formato geral
  - Localização (a[I,j]) = endereço de a [li_linha,li_coluna] + (((I – li_linha) * n) + (j – li_col)) * tamanho_do_elemento



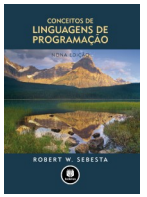
Descritores em tempo de compilação



Matriz de uma dimensão

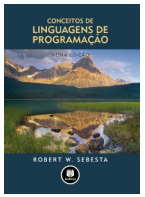


Matriz multidimensional



Matrizes associativas

- Uma *matriz associativa* é uma coleção não ordenada de elementos de dados indexados por um número igual de valores chamados de *chaves*
  - Chaves definidas pelo usuário devem ser armazenadas
- Questões de projeto:
  - Qual é o formato das referências aos seus elementos?
  - O tamanho é estático ou dinâmico?
- Suportadas diretamente em Perl, Python, Ruby e Lua
  - Em Lua, suportadas por tabelas



Matrizes associativas em Perl

- Nomes começam com `%`; literais são delimitados por parênteses

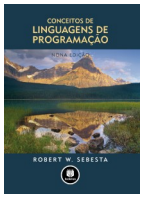
```
%hi_temps = ("Mon" => 77, "Tue" => 79, “Wed” => 65, …);
```

- Índices são escritos com colchetes e chaves

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

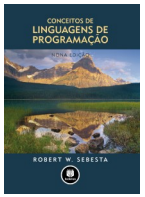
  - Elementos podem ser removidos com `delete`

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



Registros

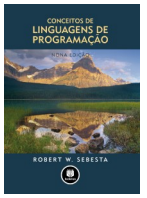
- Um *registro* é um agregado de elementos de dados no qual os elementos individuais são identificados por nomes
- Questões de projeto:
  - Qual é a forma sintática das referências a campos?
  - Referências elípticas são permitidas?



Definição de registros em COBOL

- COBOL usa números de nível para montar uma estrutura hierárquica de registros

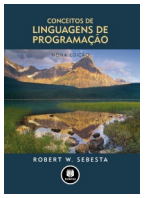
```
01 EMP-REC.
   02 EMP-NAME.
      05 FIRST PIC X(20).
      05 MID   PIC X(10).
      05 LAST  PIC X(20).
   02 HOURLY-RATE PIC 99V99.
```



Definição de registros em Ada

- Estruturas de registro são indicadas de maneira ortogonal

```
type Emp_Rec_Type is record
     First: String (1..20);
     Mid: String (1..10);
     Last: String (1..20);
     Hourly_Rate: Float;
end record;
Emp_Rec: Emp_Rec_Type;
```



Referências a registros

- Referências a campos de registros

  1. COBOL

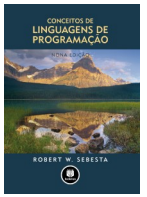
  nome_campo `OF` nome_registro_1 `OF` ... `OF` nome_registro_n

  2. Outros (notação por pontos)

  nome_registro_1.nome_registro_2. ... nome_registro_n.nome_campo

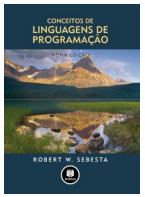
- Referência completamente qualificada deve incluir todos os nomes de registro
- Referência elíptica permite omitir todos os nomes de registros desde que a referência resultante seja não ambígua no ambiente de referenciamento, por exemplo, em COBOL

  `FIRST, FIRST OF EMP-NAME` e `FIRST` de EMP-REC são referências elípticas para o primeiro nome do empregado



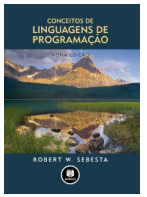
Operações em registros

- Atribuição é comum se os tipos são idênticos
- Ada permite comparações entre registros
- Registros em Ada podem ser inicializados com literais agregados
- COBOL fornece `MOVE CORRESPONDING`
  - Copia um campo do registro de origem especificado para o registro de destino se este tiver um campo com o mesmo nome



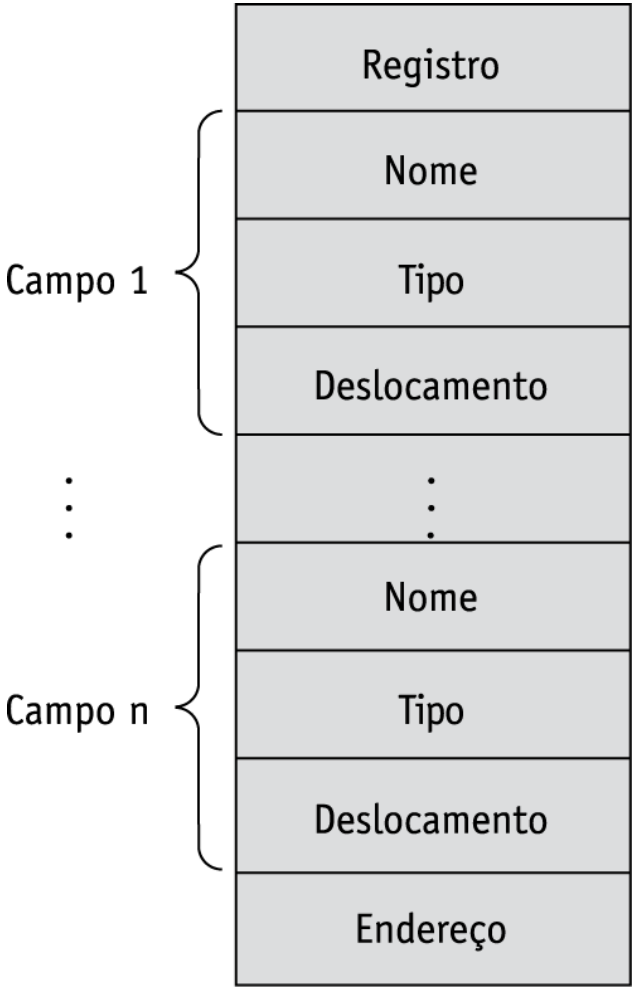
Avaliação e comparação de matrizes

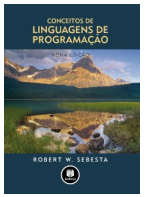
- Registros são utilizados quando a coleta de valores de dados é heterogênea
- Acesso a elementos de matriz são muito mais lentos do que campos de um registro porque os índices são dinâmicos (nomes de campos são estáticos)
- Índices dinâmicos podem ser usados com acessos a campos de registro, mas isso desabilitaria a verificação de tipos e ficaria mais lento



Implementação de registros

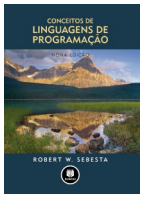
O endereço de deslocamento relativo ao início do registro é associado com cada campo





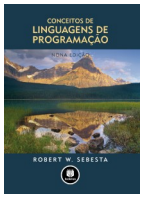
Uniões

- Uma *união* é um tipo cujas variáveis podem armazenar diferentes valores de tipos em diferentes momentos durante a execução de um programa
- Questões de projeto
  - A verificação de tipos deve ser obrigatória?
  - As uniões devem ser embutidas em registros?



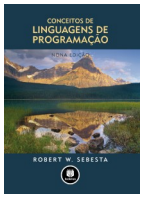
Uniões discriminadas x uniões livres

- Fortran, C e C++ fornecem construções para representar uniões nas quais não existe um suporte da linguagem para a verificação de tipos; as uniões nessas linguagens são chamadas de *uniões livres*
- A verificação de tipos união requer que cada construção de união inclua um indicador de tipo, chamado de *discriminante*
  - Uniões discriminadas são suportadas por Ada

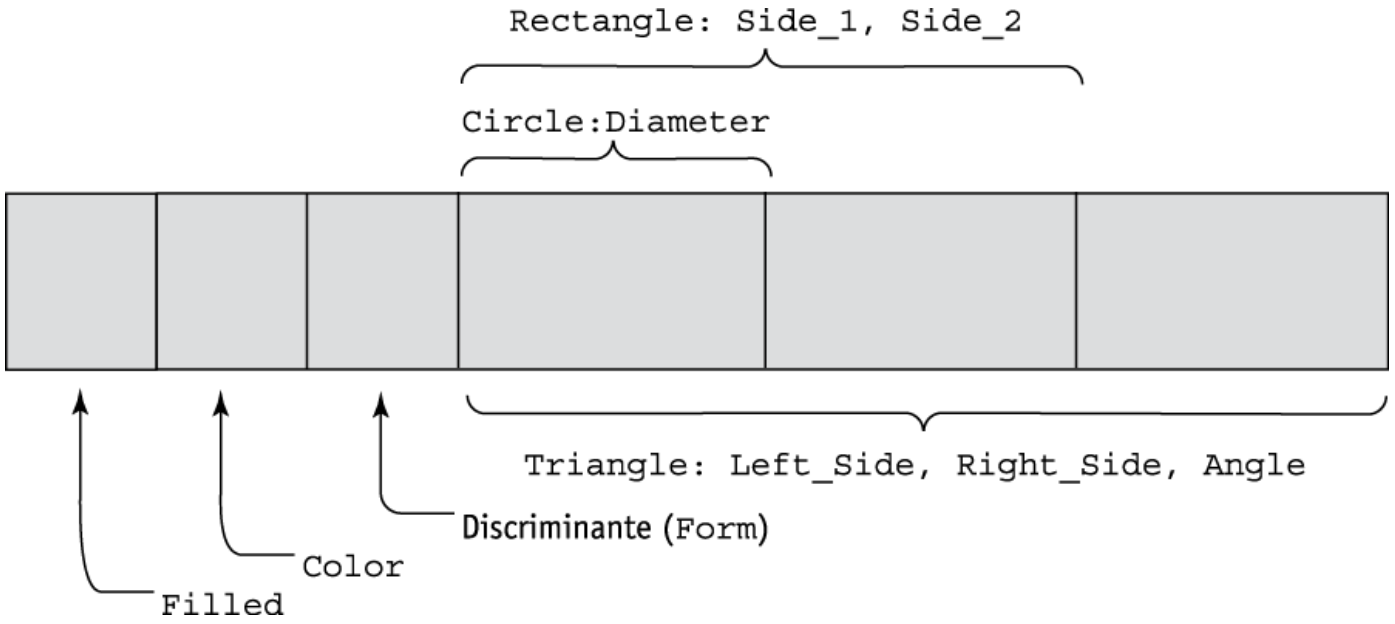


Uniões em Ada

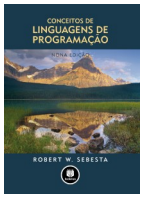
```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form: Shape) is record
 Filled: Boolean;
 Color: Colors;
 case Form is
      when Circle => Diameter: Float;
      when Triangle =>
            Leftside, Rightside: Integer;
            Angle: Float;
      when Rectangle => Side1, Side2: Integer;
 end case;
end record;
```



Uniões em Ada

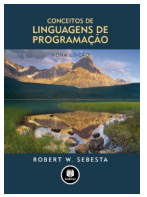


Uma união discriminada de três variáveis do tipo Shape



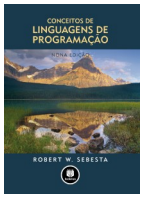
Avaliação de uniões

- Uniões são construções potencialmente inseguras
  - Não permite verificação de tipos
- Java e C# não suportam uniões
  - Reflexo da crescente preocupação com a segurança em linguagens de programação
- Em Ada, podem ser usadas com segurança



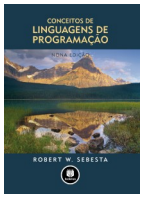
Ponteiros e referências

- Um tipo *ponteiro* é um tipo no qual as variáveis possuem uma faixa de valores que consistem em endereços de memória e um valor especial, *nil*
- Fornecem alguns dos poderes do endereçamento indireto
- Fornecem uma maneira de gerenciar o armazenamento dinâmico
- Um ponteiro pode ser usado para acessar uma posição na área onde o armazenamento é dinamicamente alocado, o qual é chamado de monte (*heap*)



Questões de projeto

- Qual é o escopo e o tempo de vida de uma variável do tipo ponteiro?
- Qual e o tempo de vida de uma variável dinâmica do monte?
- Os ponteiros são restritos em relação ao tipo de valores aos quais eles podem apontar?
- Os ponteiros são usados para gerenciamento de armazenamento dinâmico, endereçamento indireto ou ambos?
- A linguagem deveria suportar tipos ponteiro, tipos de referência ou ambos?

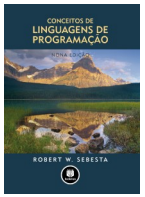


Operações de ponteiros

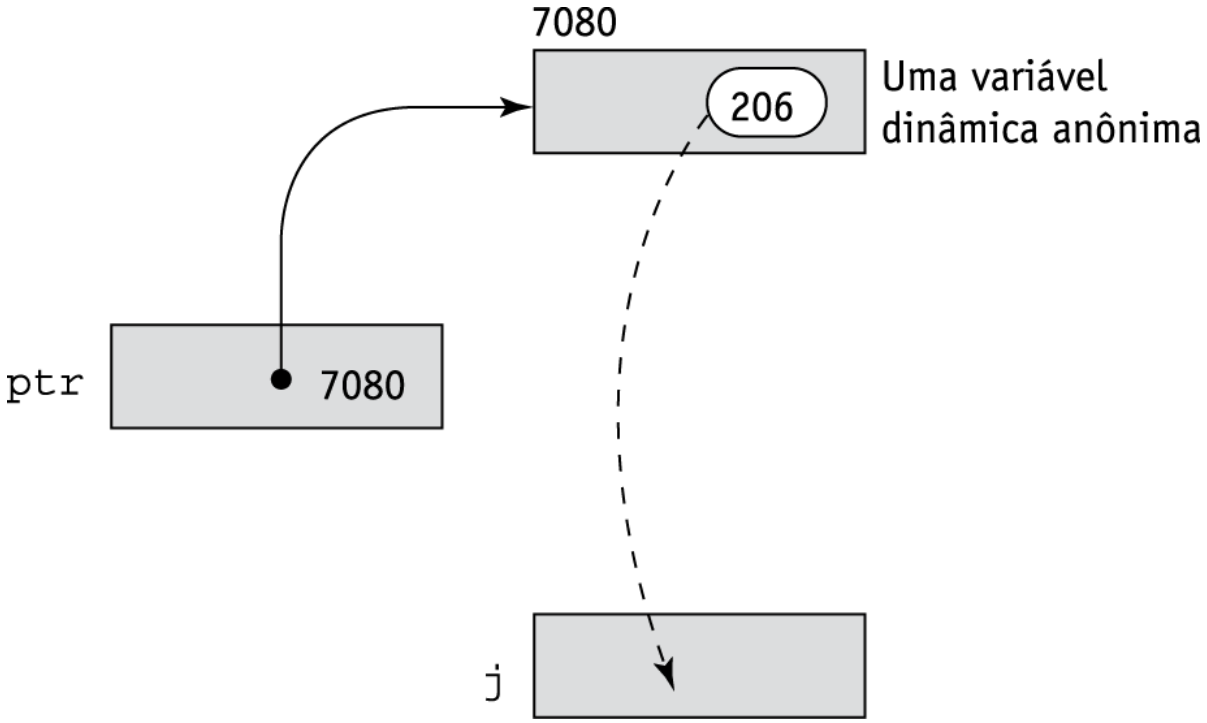
- Duas operações de ponteiros fundamentais: atribuição e desreferenciamento
- Atribuição modifica o valor de uma variável de ponteiro para algum endereço útil.
- Desreferenciamento leva uma referência por meio de um nível de indireção
  - Desreferencimento pode ser explícito ou implícito
  - Em C++, é explicitamente especificado com o asterisco (*)

    `j = *ptr`

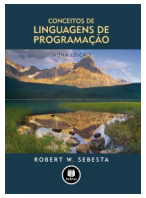
    modifica j para o valor de `ptr`



Atribuição de ponteiro

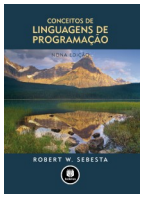


A operação de atribuição j = *ptr



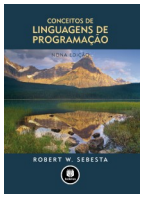
Problemas com ponteiros

- Ponteiros soltos (perigoso)
  - É um ponteiro que contém o endereço de uma variável dinâmica do monte que já foi liberada
- Variáveis dinâmicas do monte perdidas
  - É uma variável dinâmica alocada do monte que não está mais acessível para os programas de usuário (geralmente chamadas de *lixo*)
    - O ponteiro `p1` é configurado para apontar para uma variável dinâmica do monte recém-criada
    - `p1` posteriormente é configurado para apontar para outra variável dinâmica do monte recém-criada
    - A primeira variável dinâmica do monte é agora inacessível, ou perdida. Isso às vezes é chamado de *vazamento de memória*



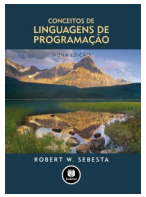
Ponteiros em Ada

- Ponteiros soltos são desabilitados porque objetos dinâmicos podem ser alocados no fim do escopo do tipo ponteiro
- O problema da variável dinâmica do monte perdida não é eliminado (possível com `UNCHECKED_DEALLOCATION`)



Ponteiros em C e C++

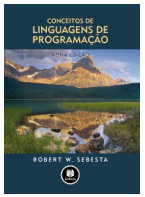
- Extremamente flexíveis, mas devem ser usados com muito cuidado
- Podem apontar para qualquer variável, independentemente de onde ela estiver alocada
- Usado para o gerenciamento de armazenamento dinâmico e endereçamento
- A aritmética de ponteiros é também possível de algumas formas restritas
- C e C++ incluem ponteiros do tipo **void ***, que podem apontar para valores de quaisquer tipos. São, para todos os efeitos, ponteiros genéricos



Aritmética de ponteiros em C e C++

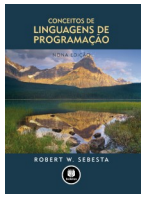
```
float stuff[100];
float *p;
p = stuff;
```

`*(p+5)` é equivalente a `stuff[5]` e `p[5]`
`*(p+i)` é equivalente a `stuff[i]` e `p[i]`



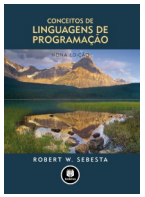
Tipos de referência

- Uma variável de *tipo de referência* é similar a um ponteiro, com uma diferença importante e fundamental: um ponteiro se refere a um endereço em memória, enquanto uma referência se refere a um objeto ou a um valor em memória
- Em Java, variáveis de referência são estendidas da forma de C++ para uma que as permitem substituírem os ponteiros inteiramente
- C# inclui tanto referências de Java quanto ponteiros de C++



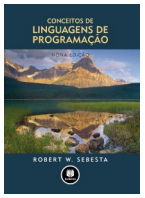
Avaliação

- Ponteiros soltos e lixo são problemas, tanto quanto o gerenciamento do monte
- Ponteiros são como a instrução `goto` - que aumenta a faixa de células que podem ser acessadas por uma variável
- Ponteiros e referências são necessários para estruturas de dados dinâmicas – não podemos projetar uma linguagem sem eles



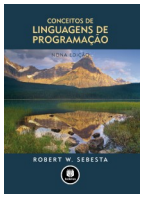
Representação de ponteiros

- Em computadores de grande porte, ponteiros são valores únicos armazenados em células de memória
- Nos primeiros microcomputadores baseados em microprocessadores Intel, os endereços possuem duas partes: um segmento e um deslocamento



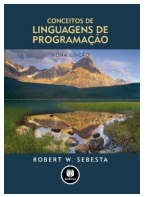
Solução para o problema dos ponteiros soltos

- *Lápides (tombstone)*: célula especial que é um ponteiro para a variável dinâmica do monte
  - A variável de ponteiro real aponta apenas para lápides
  - Quando uma variável dinâmica do monte é liberada, a lápide continua a existir, mas é atribuído a ela o valor nil
  - Caras em termos de tempo e espaço
- *Fechaduras e chaves: os valores de ponteiros são representados como pares ordenados (chaves, endereço*)
  - Variáveis dinâmicas do monte são representadas como o armazenamento da variável mais uma célula de cabeçalho que armazena um valor de fechadura inteiro
  - Quando uma variável dinâmica do monte é alocada, um valor de fechadura é criado e colocado tanto na célula de fechadura na variável dinâmica do monte quanto na célula chave do ponteiro que é especificado na chamada a new



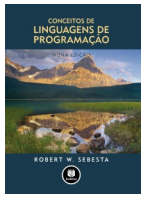
Gerenciamento do monte

- Processo em tempo de execução complexo
- Células de tamanho único × células de tamanho variável
- Duas abordagens para a coleta de lixo
  - Contadores de referências (abordagem ansiosa): a recuperação da memória é incremental e feita quando células inacessíveis são criadas
  - Marcar-varrer (abordagem preguiçosa): a recuperação ocorre apenas quando a lista de espaços disponíveis se torna vazia



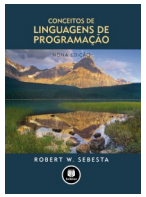
Contador de referência

- Contadores de referência: a recuperação de memória é incremental e é feita quando células inacessíveis são criadas
  - *Desvantagens:* o espaço necessário para os contadores é significativo, algum tempo de execução é necessário e complicações quando uma coleção de células é conectada circularmente
  - *Vantagens*: é intrinsecamente incremental, suas ações são intercaladas com aquelas da aplicação, então ela nunca causa demoras significativas na execução da aplicação

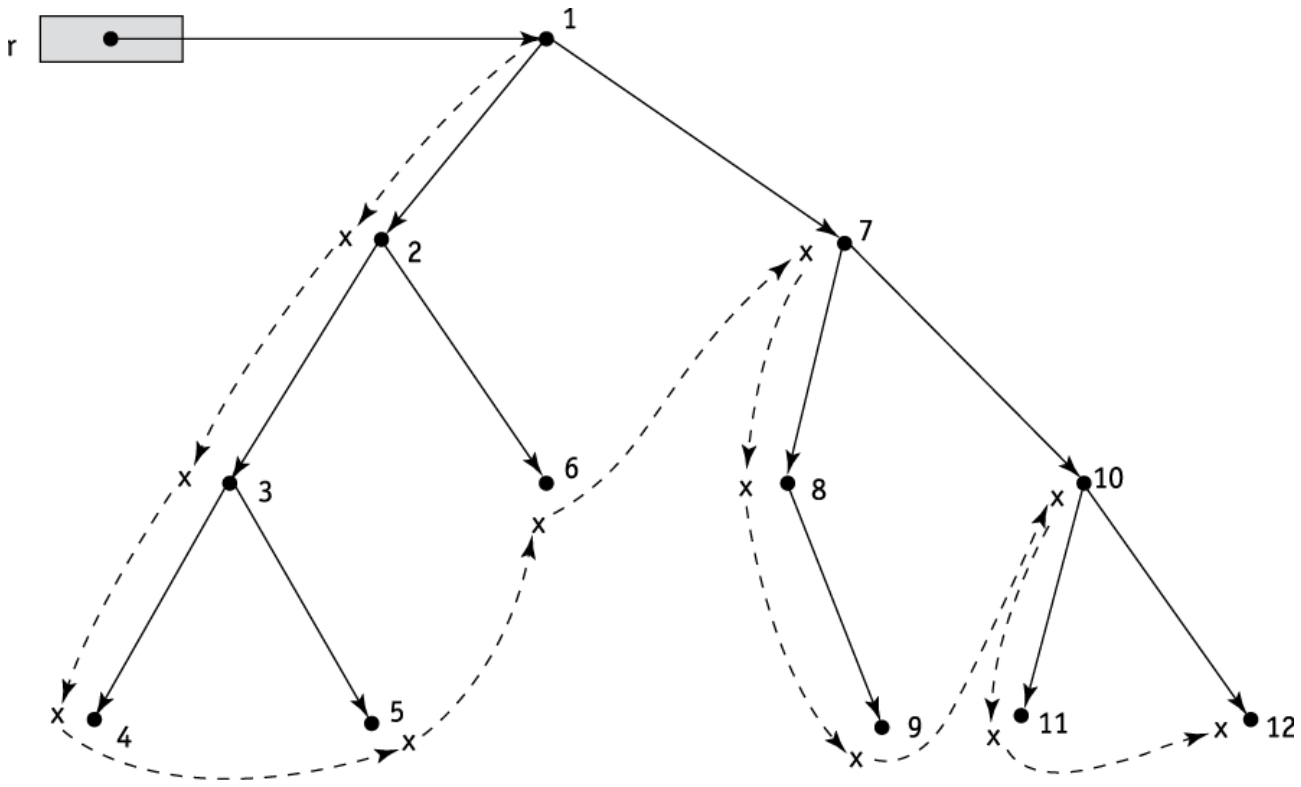


Marcar-varrer

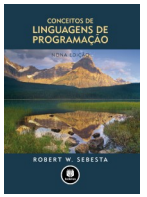
- O sistema de tempo de execução aloca células de armazenamento conforme solicitado e desconecta ponteiros de células conforme a necessidade; marcar-varrer começa
  - Cada célula do monte possui um bit ou campo indicador extra que é usado pelo algoritmo de coleta
  - Todas as células no monte têm seus indicadores configurados para indicar que eles são lixo
  - Cada ponteiro no programa é rastreado no monte, e todas as células alcançáveis são marcadas como não sendo lixo
  - Todas as células retornam para a lista de espaço disponível
  - Desvantagens: na versão original, era realizada com pouca frequência. Quando feita, causava atrasos na execução da aplicação



Algoritmo de marcação

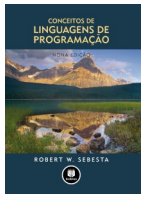


Linhas tracejadas mostram a ordem de marcação dos nós



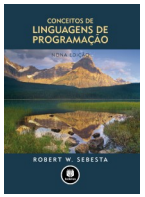
Células de tamanho variável

- Todas as dificuldades de gerenciar um monte para células de tamanho único, com problemas adicionais
- Requeridas pela maioria das linguagens de programação
- Se o marcar-varrer for usado, os seguintes problemas ocorrem
  - A configuração inicial dos indicadores para todas as células no monte para indicar que elas são lixo é difícil
  - O processo de marcação não é trivial
  - Manter a lista de espaços disponíveis é outra fonte de sobrecarga



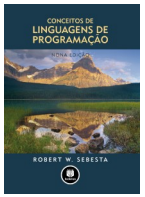
Verificação de tipos

- Conceito de operandos e operadores é generalizado para incluir subprogramas e sentenças de atribuição
- *Verificação de tipos* é a atividade de garantir que os operandos de um operador são de tipos compatíveis
- Um *tipo compatível* é um que ou é legal para o operador ou é permitido a ele, dentro das regras da linguagem, ser implicitamente convertido pelo código gerado pelo compilador (ou pelo interpretador) para um tipo legal
  - Essa conversão automática é chamada de *coerção*
- Um *erro de tipo* é a aplicação de um operador a um operando de um tipo não apropriado



Verificação de tipos (continuação)

- Se todas as vinculações são estáticas, a verificação de tipos pode ser feita praticamente sempre de maneira estática
- Se as vinculações de tipo são dinâmicas, a verificação de tipos deve ser dinâmica
- Uma linguagem de programação é *fortemente tipada* se os erros de tipo são sempre detectados
- Vantagem de tipagem forte: permite a detecção da utilização indevida de variáveis que resultam em erros de tipo

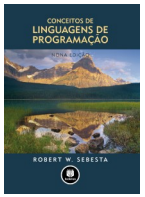


Tipagem forte

Exemplos de linguagens:

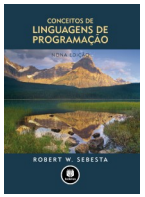
- FORTRAN 95 não é fortemente tipada: parâmetros, `EQUIVALENCE`
- C e C++ também não: ambas incluem tipos união, que não são verificados em relação a tipos
- Ada é quase fortemente tipada

  (Java e C# são similares a Ada)



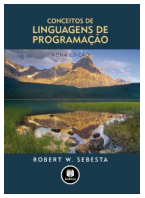
Tipagem forte (continuação)

- As regras de coerção de uma linguagem têm efeito importante no valor da verificação de tipos - eles podem enfraquecer consideravelmente (C++ *versus* Ada)
- Java e C# têm cerca de metade das coerções de tipo em atribuições que C++. Então, sua detecção de erros é melhor do que a de C++, mas não é nem perto de ser tão efetiva quanto a de Ada



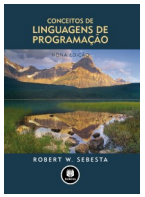
Equivalência de tipos por nome

- *Equivalência de nomes por tipo* significa que duas variáveis são equivalentes se elas são definidas na mesma declaração ou em declarações que usam o mesmo nome de tipo
- Fácil de implementar, mas é mais restritiva:
  - Subfaixas de tipos inteiros não são equivalentes a tipos inteiros
  - Parâmetros formais devem ser do mesmo tipo que os seus correspondentes parâmetros reais



Equivalência de tipos por estrutura

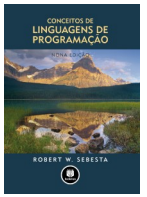
- *Equivalência de tipos por estrutura* significa que duas variáveis têm tipos equivalentes se seus tipos têm estruturas idênticas
- Mais flexível, mas mais difícil de implementar



Equivalência de tipos (continuação)

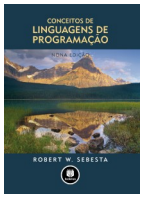
- Considere o problema de dois tipos estruturados:
  - Dois tipos de registros são equivalentes se eles são estruturalmente o mesmo, mas usarem nomes de campos diferentes?
  - Dois tipos de matriz são equivalentes se eles são o mesmo, exceto se os índices são diferentes?

    (por exemplo, [1..10] e [0..9])
  - Dois tipos de enumeração são equivalentes, se seus componentes são escritos de maneira diferente?
  - Com o tipo de equivalência estrutural, não é possível diferenciar os tipos de a mesma estrutura?



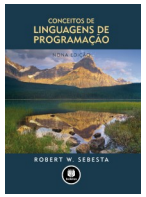
Teoria e tipos de dados

- A teoria de tipos é uma ampla área de estudo em matemática, lógica, ciência da computação e filosofia
- Em ciência da computação, existem dois ramos de teoria de tipos:
  - 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



Teoria e tipos de dados (continuação)

- O modelo formal de um sistema de tipos de uma linguagem de programação consiste em um conjunto de tipos e de uma coleção de funções que definem as regras de tipos da linguagem
  - Tanto uma gramática de atributos quanto um mapa de tipos pode ser usado para as funções
  - Mapeamento finito – modela matrizes e funções
  - Produto cartesiano – modela tuplas e registros
  - União de conjunto – modela tipos de dados 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 de uma linguagem
- Os tipos de dados primitivos da maioria das linguagens imperativas incluem os tipos numéricos, de caracteres e booleanos
- Os tipos de enumeração e de subfaixa definidos pelo usuário são convenientes e melhoram a legibilidade e a confiabilidade dos programas
- Matrizes fazem parte da maioria das linguagens de programação
- Ponteiros são usados para lidar com a flexibilidade e para controlar o gerenciamento de armazenamento dinâmico