

Paradigmas de Linguagens de Programação

Nomes, Vinculações, Verificação de Tipos e Escopos

Cristiano Lehrer, M.Sc.

Nomes (1/6)

- Um **nome** é uma *string* de caracteres usada para identificar alguma entidade de um programa.
- O termo **identificador** é utilizado muitas vezes de forma intercambiável com nome.
- Principais questões de projeto para nomes:
 - Qual o tamanho máximo de um nome.
 - Caracteres de conexão podem ser usados em nomes.
 - Os nomes fazem distinção entre maiúsculas e minúsculas.
 - As palavras especiais são palavras reservadas ou palavras-chave.

Nomes (2/6)

- Tamanho:
 - As primeiras linguagens de programação usavam nomes de um único caractere:
 - Notação matemática utiliza um caractere para incógnitas!
 - **FORTRAN I** → 6 caracteres.
 - **FORTRAN 90** e **C** → 31 caracteres.
 - **ADA** → sem restrição de tamanho.
 - **C++** → sem restrição de tamanho:
 - Limitações impostas pelos implementadores do compilador!
 - **Java** → sem restrição de tamanho.

Nomes (3/6)

- Conectores:
 - O sublinhado ou *underscore* () é o conector normalmente utilizado.
 - Pascal, Modula-2 e FORTRAN 77 não permitem.
 - Outras linguagens permitem.

Nomes (4/6)

- Distinção entre letras maiúsculas e minúsculas:
 - Vantagem → aumentam a gama de nomes possíveis:
 - `Soma`, `soma` e `SOMA` são identificadores diferentes.
 - Desvantagem → perda de legibilidade, uma vez que nomes similares são diferentes.
 - `C`, `C++` e `Java`:
 - Nomes distinguem letras maiúsculas e minúsculas.
 - Método `Java` para converter um `String` em um valor inteiro:
 - `Integer.parseInt(String)`
 - Lembrar a grafia correta do comando → problema de capacidade da escrita:
 - `Integer.ParseInt(String)` ou `Integer.parseint(String)`
 - Os nomes em outras linguagens não fazem distinção entre letras maiúsculas e minúsculas.

Nomes (5/6)

- Palavras especiais:
 - Uma **palavra-chave** é uma palavra especial somente em certos contextos:
 - As palavras especiais em **FORTRAN** são palavras-chave:
 - **REAL** seguido de um nome indica a declaração de uma variável:
 - REAL soma
 - **REAL** seguido de um operador de instrução, é uma variável:
 - REAL = 3.4
 - Tornam a legibilidade da linguagem de programação pobre!

Nomes (6/6)

- Palavras especiais:

- Uma **palavra reservada** é uma palavra especial e não pode ser usada como um nome definido pelo usuário:
 - Aumentam a legibilidade e a segurança da linguagem.
 - Utilizado pela maioria das linguagens de programação.
 - Exemplo em **Java**:
 - `int goto = 0, for = 0; // Instrução inválida`
 - `int Goto = 0, For = 0; // Instrução válida`
 - Lembre-se que o **Java** faz distinção entre as letras maiúsculas e minúsculas!

Variáveis (1/5)

- Uma **variável** de programa é uma abstração de uma célula ou de um conjunto de células de memória do computador.
- Variáveis podem ser caracterizadas como um sêxtuplo de atributos:
 - Nome
 - Endereço
 - Valor
 - Tipo
 - Tempo de vida
 - Escopo

Variáveis (2/5)

- O **endereço** de uma variável é o mesmo da memória à qual ela está associada:
 - Uma variável pode ter diferentes endereços em diferentes lugares no programa, conforme o exemplo em [Java](#):

```
public void sub1(){int x = 0;}
```

```
public void sub2(){int x = 2;}
```

- Uma variável pode ter diferentes endereços em diferentes tempos durante a execução do programa, conforme o exemplo em [Java](#):

```
public void fatorial(int n){  
    if (n <= 1) return 1  
    else return n * fatorial(n - 1);  
}
```

Variáveis (3/5)

- Se duas ou mais variáveis podem ser usadas para acessar a mesma posição de memória, elas são chamadas **alias**es:
 - A criação de *alias*es é um estorvo para a legibilidade:
 - Permite que uma variável tenha seu valor alterado por uma atribuição para uma variável diferente.
 - Torna a verificação de programas mais difícil.
 - *Alias*es são criados, principalmente, com ponteiros e referências, conforme o exemplo em **Java**:

```
Object o = new Object();
```

```
Object p = o; // apontam para o mesmo objeto
```

Variáveis (4/5)

- O **tipo** de uma variável determina a faixa de valores que ela pode ter e o conjunto de operações definidas para os valores do tipo:
 - Por exemplo, variáveis do tipo `int` em `Java` possuem uma faixa de valores igual a -2^{31} a $2^{31} - 1$ e operações aritméticas para adição (+), subtração (-), multiplicação (*), divisão inteira (/), divisão real (/) e resto da divisão (%).

Variáveis (5/5)

- O **valor** de uma variável é o conteúdo da célula ou das células de memória associadas à variável:
 - Célula de memória abstrata:
 - Célula física ou coleção de células físicas associadas com uma variável.
 - Facilita o entendimento, uma vez que fisicamente as células de memória possuem um tamanho fixo:
 - Atualmente, em 32 bits ou 64 bits.

Conceito de Vinculação (1/5)

- Uma **vinculação** é uma associação, como, por exemplo, entre um atributo e uma entrada ou entre uma operação e um símbolo:
 - O momento em que uma vinculação desenvolve-se é chamado de **tempo de vinculação**.
 - Vinculação e tempo de vinculação são conceitos proeminentes na semântica das linguagens de programação.

Conceito de Vinculação (2/5)

- Possíveis tempos de vinculação:
 - Tempo de projeto da linguagem:
 - Associando símbolos de operações:
 - Associar o símbolo de `*` a operação de multiplicação.
 - Tempo de implementação da linguagem:
 - Associando o intervalo de valores aos tipos de dados:
 - Associar o intervalo de -2^{31} a $2^{31} - 1$ ao tipo `int` em `Java`.
 - Tempo de compilação:
 - Associando uma variável a um tipo de dados:
 - `int a; // código em C ou em Java`

Conceito de Vinculação (3/5)

- Possíveis tempos de vinculação – continuação:
 - Tempo de ligação (*link*):
 - Associando uma chamada a uma biblioteca da linguagem:
 - `printf("Hello World!"); // código em C`
 - Tempo de carregamento:
 - Associando uma variável estática a uma célula de memória:
 - `public static int total = 10; // código em Java`
 - Tempo de execução (*run-time*):
 - Associando uma variável local a uma célula de memória:
 - `public void sub1(){int x = 0;} // código em Java`

Conceito de Vinculação (4/5)

- Exemplo de vinculações e seus tempos de vinculação para o código C a seguir:

```
int count;
```

```
count = count + 5;
```

- Conjunto dos tipos possíveis para `count`: vinculado no tempo de projeto.
- Tipo de `count`: vinculado no tempo de compilação.
- Conjunto de valores possíveis de `count`: vinculado no tempo de projeto do compilador.
- Valor de `count`: vinculado no tempo de execução com esta instrução.
- Conjunto dos significados possíveis para o símbolo do operador `+`: vinculado no tempo de definição da linguagem.
- Significado do símbolo do operador `+` nessa instrução: vinculado no tempo de compilação.
- Representação interna do literal `5`: vinculada no tempo de projeto do compilador.

Conceito de Vinculação (5/5)

- Vinculação de atributos a variáveis:
 - Uma vinculação é **estática** se ocorrer antes do tempo de execução e permanecer inalterada ao longo da execução do programa.
 - Se a vinculação ocorrer durante a execução ou puder ser modificada no decorrer da execução de um programa, será chamada **dinâmica**.

Vinculações de Tipos (1/3)

- Declarações de variáveis:

- Uma **declaração explícita** é uma instrução em um programa que lista nomes de variáveis e especifica que elas são de um tipo particular:

- `int a, b, c; // código em C ou em Java`

- Uma **declaração implícita** é um meio de associar variáveis a tipos por convenções padrão em vez de por instruções de declaração:

- Em **FORTRAN**, um identificador que aparece em um programa que não é explicitamente declarado é implicitamente declarado de acordo com a seguinte convenção.

- Se o identificador iniciar-se com uma das letras **I, J, K, L, M** ou **N** ele será implicitamente declarado como do tipo **INTEGER**.
 - Caso contrário, será implicitamente declarado como do tipo **REAL**.

Vinculações de Tipos (2/3)

- Vinculação dinâmica de tipos:
 - A variável é vinculada ao tipo quando lhe é atribuído um valor em uma instrução de atribuição:
 - Por exemplo, em um programa **APL**:
 - `LIST = 10.2 5.1 0.0 // torna LIST é um array de reais`
 - `LIST = 47 // torna LIST uma variável escalar inteira`
 - Vantagem:
 - Flexibilidade (unidades de programa genéricos).
 - Desvantagens:
 - Alto custo (verificação dinâmica de tipos e interpretação).
 - Detecção de erro de tipagem pelo computador é difícil.

Vinculações de Tipos (3/3)

- Inferência de tipos:
 - Ao invés de comandos de assinalamento, tipos são determinados a partir do contexto de referência.
 - Exemplos em [ML](#):
 - **fun** `circumf(r) = 3.14159 * r * r;`
 - Argumento real e resultado real.
 - **fun** `vezes10(x) = 10 * x;`
 - Argumento inteiro e resultado inteiro.
 - **fun** `quadrado(x) = x * x;`
 - Inválido, o tipo para o operador `*` não pode ser inferido.
 - **fun** `quadrado(x) : int = x * x;`
 - Argumento inteiro e resultado inteiro.

Vinculações de Armazenamento (1/5)

- A célula de memória à qual uma variável é vinculada deve, de alguma maneira, ser tomada de um pool de memória disponível:
 - Este processo é chamado de **alocação**.
- **Desalocação** é o processo de devolver uma célula de memória desvinculada de uma variável ao pool de memória disponível.
- O **tempo de vida** de uma variável é o tempo durante o qual esta é vinculada a uma localização de memória específica:
 - Inicia-se quando a variável é vinculada a uma célula específica.
 - Encerra-se quando a variável é desvinculada dessa célula.

Vinculações de Armazenamento (2/5)

- **Variáveis estáticas** são aquelas vinculadas a células de memória antes que a execução do programa inicia-se e que assim permanecem até que a execução do programa encerre-se:
 - Vantagem:
 - Eficiência (endereçamento direto).
 - Suporte a subprogramas **sensíveis à história**, ou seja, variáveis que retenham valores entre as execuções do subprograma.
 - Desvantagem:
 - Falta de flexibilidade (sem recursão).
 - Exemplo em linguagens de programação:
 - Todas as variáveis **FORTRAN 77**, variáveis do **C**, **C++** e **Java** precedidas por **static**.

Vinculações de Armazenamento (3/5)

- **Variáveis stack-dinâmicas** são aquelas cujas vinculações de armazenamento criam-se a partir da elaboração de suas instruções de declaração, mas cujos tipos são estaticamente vinculados:
 - A **elaboração** dessas declarações refere-se ao processo de alocação e de vinculação de armazenamento indicado pela declaração, o qual se desenvolve quando a execução atinge o código em que ela está anexada.
 - Vantagem:
 - Permite recursão, uma vez que conserva o valor armazenado.
 - Desvantagem:
 - Sobrecarga de alocação e desalocação.
 - Subprogramas não podem ser sensíveis à história.
 - Referências ineficientes (endereçamento indireto).
 - Exemplos em linguagens de programação:
 - Variáveis locais em **Pascal** e em subprogramas em **C**.
 - Variáveis de tipos primitivos declaradas em métodos em **Java**.

Vinculações de Armazenamento (4/5)

- **Variáveis heap-dinâmicas explícitas** são células de memória sem nome (abstratas) alocadas e desalocadas por instruções explícitas em tempo de execução, especificadas pelo programador:
 - Referenciado através de ponteiros ou referências.
 - Vantagem:
 - Permite gerenciamento de armazenamento dinâmica.
 - Desvantagem:
 - Ineficiente e não confiável.
 - Exemplos em linguagens de programação:
 - Objetos dinâmicos em C++ (**new** e **delete**).
 - Todos os objetos em Java.

Vinculações de Armazenamento (5/5)

- Variável heap-dinâmicas implícita são vinculadas ao armazenamento do *heap* somente quando lhe são atribuídos valores:
 - Vantagem:
 - Flexibilidade.
 - Desvantagem:
 - Ineficiente, porque todos os atributos são dinâmicos.
 - Perca na detecção de erros.
 - Exemplos em linguagens de programação:
 - Todas as variáveis em [APL](#).

Verificação de Tipos (1/4)

- **Verificação de tipo** é a atividade de assegurar que os operandos de um operador sejam de tipos compatíveis:
 - Um tipo **compatível** é aquele válido para o operador ou com permissão, nas regras da linguagem, para ser convertido pelo código gerado pelo compilador para um tipo válido:
 - Essa conversão automática é chamada de **coerção**.
 - Um **erro de tipo** é a aplicação de um operador a um operando de tipo impróprio.

Verificação de Tipos (2/4)

- Se todas as vinculações de tipo forem estáticas, quase todas as verificações de tipo podem ser estáticas.
- Se as vinculações de tipo forem dinâmicas, a verificação de tipo deve ser dinâmica.
- Uma linguagem de programação é **fortemente tipificada** se erros de tipificação são sempre detectados:
 - Vantagem de tipificação forte:
 - Permite a detecção de uso inadequado de variáveis que resultam em erros de tipos.

Verificação de Tipos (3/4)

- Tipificação forte em linguagens de programação:
 - **FORTRAN** não é fortemente tipificado:
 - Relação entre parâmetros reais e formais não é verificada quanto ao tipo.
 - **C** e **C++** não são fortemente tipificados:
 - Verificação dos tipos de parâmetros pode ser evitada.
 - Uniões não são verificadas.
 - **Java** é fortemente tipificado.

Verificação de Tipos (4/4)

- Regras de coerção de uma linguagem têm um efeito importante sobre o valor da verificação de tipos:

- Podem enfraquecê-la consideravelmente:

- C

```
float a = 5.7;
```

```
int b = a; // permitido
```

- Java

```
float a = 5.7;
```

```
int b = a; // não permitido
```

Compatibilidade de Tipos (1/5)

- A ideia da compatibilidade de tipos foi definida quando se estudou a questão da verificação de tipos.
- Há dois métodos diferentes de compatibilidade de tipos:
 - Compatibilidade de nome.
 - Compatibilidade de estrutura.

Compatibilidade de Tipos (2/5)

- **Compatibilidade de nome** significa que duas variáveis têm tipos compatíveis somente se estiverem na mesma declaração ou em declarações que usam o mesmo nome de tipo:
 - Fácil de implementar, mas altamente restritivo:
 - Considerando que o **Pascal** usasse compatibilidade estrita de nomes de tipos:

```
type indextype = 1..100; {um tipo subfaixa}
var count: integer;
    index: indextype;
```
 - As variáveis **count** e **index** não são compatíveis!

Compatibilidade de Tipos (3/5)

- **Compatibilidade de estrutura** significa que duas variáveis têm tipos compatíveis se os seus tipos tiverem estruturas idênticas:
 - Mais flexível, mas (mais) difícil de implementar.
 - Considerando que o **Pascal** usasse compatibilidade estrita de estrutura de tipos:

```
type celsius = real;
```

```
    fahrenheit = real;
```

- As variáveis desses dois tipos são compatíveis, de modo que podem ser misturadas em expressões, o que, certamente, é indesejável nesse caso.

Compatibilidade de Tipos (4/5)

- **Pascal:**

- Não utiliza nem completamente pelo nome, nem pela estrutura:

type

```
tipo1 = array [1..10] of integer;
```

```
tipo2 = array [1..10] of integer;
```

```
tipo3 = tipo2;
```

- **tipo1** e **tipo2** não são compatíveis:
 - A compatibilidade de tipo de estrutura não é usada.
- **tipo2** é compatível com **tipo3**:
 - A compatibilidade de tipo por nome não é estritamente usada.

Compatibilidade de Tipos (5/5)

- **C:**
 - Usa equivalência estrutural para todos os tipos, exceto para as estruturas (registros e uniões), para as quais o **C** usa equivalência de declaração.
- **C++:**
 - Usa equivalência de nomes.
- As linguagens orientada a objetos, como o **Java** e o **C++**, trazem consigo a questão da compatibilidade de objetos e sua relação com a hierarquia de herança.

Escopo (1/6)

- O **escopo** de uma variável de programa é a faixa de instruções na qual a variável é visível:
 - Uma variável é **visível** em uma instrução se puder ser referenciada nessa instrução.
 - As **variáveis não-locais** de uma unidade ou de um bloco de programa são as visíveis dentro daquela ou deste, mas não são declaradas aí.
 - As regras de escopo de uma linguagem determinam como uma ocorrência particular de um nome está associada à variável.

Escopo (2/6)

- No **escopo estático** (*static scoping*) o escopo de uma variável pode ser determinado estaticamente, ou seja, antes da execução do programa:
 - Conceito introduzido pelo **ALGOL 60** e adotado pela maioria das linguagens de programação.
 - Processo para localizar a declaração de uma variável:
 - Iniciar a pesquisa no escopo atual e ir subindo na hierarquia de escopos até que seja encontrado uma declaração com o nome desejado.
 - Variáveis podem estar escondidas de uma unidade quando existe uma outra variável “mais próxima” com o mesmo nome.

Escopo (3/6)

- Considere o seguinte procedimento em **Pascal**:

```
procedure big;
  var x: integer;
  procedure sub1;
  begin {sub1}
    ...x...           {x declarado em big}
  end {sub1}
  procedure sub2;
  var x: integer;
  begin {sub2}
    ...x...           {x declarado em sub2}
  end; {sub2}
begin {big}
  ...x...             {x declarado em big}
end; {big}
```

```
procedure big;
var x: integer;
```

```
procedure sub1;
```

```
procedure sub2;
var x: integer;
```

Escopo (4/6)

- Muitas linguagens permitem que novos escopos estáticos sejam definidos no meio do código, permitindo que uma seção de código tenha suas próprias variáveis locais cujo escopo é minimizado:
 - Esse tipo de seção de código é chamado de bloco.
 - As instruções **for** em **C++** e **Java** permitem definições de variáveis em suas expressões de inicialização:

```
for(int x = 0; x <= 10; x++)  
{  
    ...  
}
```

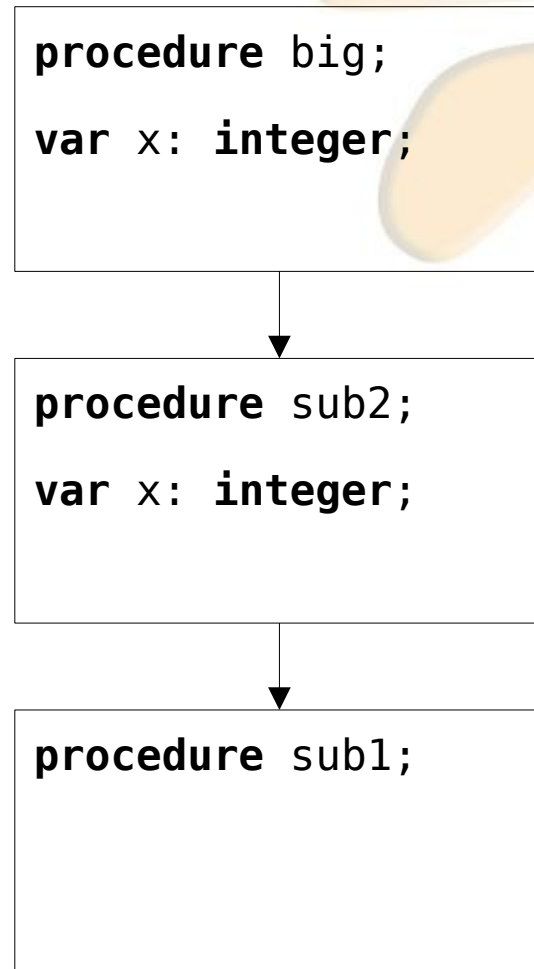
Escopo (5/6)

- O **escopo dinâmico** baseia-se na sequência de chamada de subprogramas, não em suas relações espaciais um com o outro:
 - Escopo das variáveis em **APL**, **SNOBOL4** e nas primeiras versões de **LISP**.
- Os programas em linguagens de escopo estático são de leitura mais fácil, mais confiáveis e sua execução é mais rápida do que programas equivalentes em linguagens de escopo dinâmico!

Escopo (6/6)

- Considere o seguinte procedimento em **Pascal**, e a seguinte chamada de procedimentos:
 - big → sub2 → sub1

```
procedure big;  
  var x: integer;  
  procedure sub1;  
  begin {sub1}  
    ...x...           {x declarado em sub2}  
  end {sub1}  
  procedure sub2;  
    var x: integer;  
  begin {sub2}  
    ...x...           {x declarado em sub2}  
  end; {sub2}  
begin {big}  
  ...x...             {x declarado em big}  
end; {big}
```



Escopo e Tempo de Vida

- Escopo e o tempo de vida de uma variável parecem estar relacionados, mas são conceitos diferentes:
 - Por exemplo, em C e C++, uma variável declarada em uma função usando o especificador `static` está estaticamente vinculada ao escopo dela e, também, estaticamente vinculada ao armazenamento:

```
void sub1() {  
    static int a = 10;  
    ...  
}
```

Ambientes de Referenciamento (1/3)

- O **ambiente de referenciamento** de uma instrução é o conjunto de todos os nomes visíveis na instrução:
 - Numa linguagem de escopo estático, é formado pelas variáveis locais e todas as variáveis visíveis em todos os escopos superiores (mais externos).
 - Numa linguagem de escopo dinâmico, o ambiente de referência é formado pelas variáveis locais e todas as variáveis visíveis em todos os subprogramas ativos:
 - Um subprograma é **ativo** se sua execução já começou mas ainda não terminou.

Ambientes de Referenciamento (2/3)

Ambiente de Referenciamento – Escopo Estático

```
void sub1(){  
  int a, b;  
  ...  
}
```

a e b de sub1

```
void sub2(){  
  int b, c;  
  ...  
  sub1();  
}
```

b e c de sub2

```
void main(){  
  int c, d;  
  ...  
  sub2();  
}
```

c e d de main

Ambientes de Referenciamento (3/3)

Ambiente de Referenciamento – Escopo Dinâmico

```
void sub1(){  
  int a, b;  
  ... ←  
}
```

a e b de sub1, c de sub2, d de main
(c de main e b de sub2 estão ocultos)

```
void sub2(){  
  int b, c;  
  ... ←  
  sub1();  
}
```

b e c de sub2, d de main
(c de main está oculto)

```
void main(){  
  int c, d;  
  ... ←  
  sub2();  
}
```

c e d de main

Constantes Nomeadas

- Uma **constante nomeada** é uma variável vinculada a um valor somente no momento em que ela é vinculada a um armazenamento; seu valor não pode ser mudado pela instrução de atribuição ou por uma instrução de entrada:
 - Vantagens: Legibilidade e modificabilidade.
 - As vinculações de valores a constantes nomeadas podem ser estáticas (chamadas de **constantes manifestas**) ou dinâmicas:
 - Em **Pascal**, estáticas:
 - `const listlen = 100;`
 - Em **C**, **C++** ou **Java**, podem ser estáticas ou dinâmicas:
 - `const float PI = 3.1415; // estático, em C`
 - `const int WIDTH = 100 * lado; // dinâmico, em C`

Inicialização de Variáveis

- A vinculação de uma variável a um valor no momento em que ela é vinculada ao armazenamento é chamada **inicialização**:
 - A inicialização normalmente é realizada na instrução de declaração:
 - `int a = 0; // código em C`
 - Por exemplo, em **Java**, atributos (variáveis definidas na classe) são inicializadas automaticamente com valores padrões, enquanto que variáveis locais (definidas no corpo de um método) devem ser inicializadas explicitamente!