1. Expressões e Sentenças de Atribuição (Slides 06)

Expressões são o meio fundamental para especificar **computações** em uma linguagem de programação.

Expressões Aritméticas

As expressões aritméticas utilizam operadores (unários, binários ou ternários) e operandos.

- **Ordem de Avaliação de Operadores**: Para evitar ambiguidade, as linguagens seguem regras de **precedência** e **associatividade**. Por exemplo, multiplicação e divisão têm precedência maior que adição e subtração, e a maioria dos operadores é associativa à esquerda (avalia da esquerda para a direita). Os parênteses podem ser usados para alterar explicitamente a ordem de avaliação.
- **Ordem de Avaliação de Operandos**: A ordem de avaliação dos operandos pode gerar resultados distintos se a função operando tiver **efeitos colaterais** (modificar um parâmetro por referência ou uma variável global). Linguagens como Java impõem ordem fixa (esquerda para a direita) para garantir previsibilidade, enquanto C/C++ permitem flexibilidade, priorizando a eficiência.
- **Operadores Sobrecarregados**: Um mesmo operador pode ser usado para mais de um propósito (ex: `+` para inteiros e *float*). Linguagens como C++, C#, Ada e Python permitem a sobrecarga de operadores definida pelo usuário para melhorar a legibilidade.
- **Conversões de Tipos**:
    - **Alargamento (*widening*)**: Converte para um tipo que pode representar todos os valores do tipo original (mais segura, ex: `int` → `float`).
    - **Estreitamento (*narrowing*)**: Converte para um tipo que não pode representar todos os valores (insegura, ex: `float` → `int`).
    - Podem ser **implícitas** (coerção) ou **explícitas** (*cast*). Linguagens como F#, ML e Ada não permitem coerção implícita para aumentar a segurança de tipos.

Expressões Relacionais e Booleanas

Expressões relacionais comparam dois operandos e resultam em um valor **booleano** (verdadeiro/falso). Os operadores relacionais têm precedência menor que os aritméticos. Expressões booleanas consistem em variáveis, constantes e expressões relacionais, combinadas com operadores booleanos (AND `&&`, OR `||`, NOT `!`).

- **Avaliação em Curto-Circuito**: O valor da expressão pode ser determinado sem avaliar todos os operandos (ex: em `A && B`, se `A` for falso, `B` não é avaliado).

Sentenças de Atribuição

A essência das linguagens imperativas é o papel dominante das sentenças de atribuição, que causam o **efeito colateral** de alterar o **estado do programa**.

- **Formas Comuns**: As formas incluem atribuições simples (`=` ou `:=`), compostas (`+=`, `-=`) e unárias (`++`, `--`).
- **Atribuição como Expressão**: Em C, C++ e Java, a atribuição retorna um valor, permitindo o uso de atribuições em expressões (ex: `while ((ch = getchar()) != EOF)`).
- **Atribuições Múltiplas**: Linguagens como Perl, Ruby, Lua e Python suportam a atribuição de múltiplos alvos e fontes (ex: `$a, $b = $b, a).
- **Linguagens Funcionais**: Nesses paradigmas, não existe "atribuição" no sentido imperativo; identificadores são **nomes vinculados a valores imutáveis** (bindings), e não variáveis mutáveis.

---

2. Estruturas de Controle no Nível de Sentença (Slides 07)

O **fluxo de controle** define a ordem de execução das instruções. As **sentenças de controle** são estruturas que permitem escolher caminhos alternativos e repetir execuções.

Sentenças de Seleção

Permitem escolher entre dois ou mais caminhos de execução.

- **Seleção de Dois Caminhos** (`if-else`):
  - A **expressão de controle** deve ser booleana nas linguagens modernas (C99 e C++ aceitam aritméticas/booleanas; Java apenas booleana).
  - **Aninhamento de Seletores**: A maioria das linguagens imperativas associa a cláusula `else` ao `if` mais próximo. Linguagens como Python, Ruby e Lua usam indentação ou palavras-chave de fechamento (`end`) para desambiguar.
- **Seleção Múltipla** (`switch`/`case`):
  - São generalizações do `if-else` para melhorar a legibilidade.
  - **Switch em C**: Usa valores constantes discretos e permite a execução de múltiplos casos seguidos (*fall-through*), a menos que seja usado o `break`.
  - **Estruturas *else-if* e `elif`**: Usadas quando a seleção é baseada em expressões booleanas e não em valores discretos. A estrutura `elif` (Python) é mais legível do que `if` aninhados.

    - **Linguagens Funcionais**: Usam formas baseadas em expressões, como o `COND` do Scheme, que avalia pares (predicado, expressão) sequencialmente.

Sentenças de Iteração (Laços)

Fazem uma sentença ou coleção de sentenças ser executada repetidamente (*laço*).

- **Controle da Iteração**: Pode ser por **contadores** (ex: `for`) ou por **condições lógicas** (ex: `while`).
    - **Pré-teste**: A condição é testada antes da execução do corpo (ex: `while`, `for`).
    - **Pós-teste**: A condição é testada depois da execução do corpo (ex: `do...while`).
- **Laços Controlados por Contador**: Têm uma **variável de laço**, valor inicial, final e tamanho do passo.
    - **`for` em C**: Estrutura flexível que permite múltiplas inicializações e atualizações.
    - **`for` em Python**: Itera sobre os elementos de um **objeto iterável** (lista, *range*, etc.). A cláusula `else` opcional é executada se o laço terminar normalmente (sem `break`).
    - **Linguagens Funcionais Puras**: A repetição é controlada por **recursão**, pois não possuem variáveis mutáveis.
- **Laços Controlados Logicamente**: São mais gerais e controlados por uma expressão booleana (ex: `while`, `do...while`).
- **Mecanismos de Controle Posicionados pelo Usuário**:
    - **`break`**: Encerra o laço completamente.
    - **`continue`**: Pula o restante do corpo do laço e volta ao topo.
    - Linguagens como Java e Perl permitem `break` rotulados para sair de múltiplos laços aninhados.
- **Iteração Baseada em Estruturas de Dados**: Usa um **iterador** (implícito ou explícito) para percorrer coleções. Linguagens como Java (a partir da versão 5.0), C# e Python usam a sintaxe `foreach` ou `for...in`.

Desvio Incondicional

- **`goto`**: Transfere o controle de execução para uma posição especificada (rótulo). É a sentença mais poderosa, mas seu uso sem restrições pode prejudicar a legibilidade e a confiabilidade do programa (crítica de Dijkstra). Linguagens como Java, Python e Ruby não possuem `goto`.

---

3. Subprogramas (Slides 08)

Subprogramas são conjuntos de sentenças que definem **computações parametrizadas**, sendo o principal recurso de **abstração de processos**.

Fundamentos

- **Tipos**: **Procedimentos** (não retornam valor) e **Funções** (retornam valor). As funções são semanticamente modeladas como funções matemáticas.
- **Características Comuns**: Ponto de entrada único, a unidade chamadora é suspensa (existe apenas um subprograma em execução por vez), e o controle retorna ao chamador ao término.
- **Componentes**:
    - **Cabeçalho**: Especifica o nome, o tipo e a lista de **parâmetros formais**.
    - **Corpo**: Define as ações.
    - **Protocolo**: Perfil dos parâmetros formais mais o tipo de retorno.
- **Acesso a Dados**:
    - **Variáveis não locais**: Acesso direto, que pode reduzir a confiabilidade.
    - **Passagem de Parâmetros**: Mais flexível; os dados são acessados como nomes locais ao subprograma.
- **Parâmetros**: Os nomes declarados são **parâmetros formais**; as expressões passadas na chamada são **parâmetros reais**.
    - **Posicionais**: Vinculados pela ordem na lista.
    - **Palavra-chave (*keyword*)**: O nome do formal é explicitado na chamada (ex: Python).
    - **Valores Padrão (*default*)**: Usados quando nenhum argumento real é passado.
    - **Número Variável de Parâmetros**: Linguagens como C (`printf`), Python (`*args`, `**kwargs`) e C# (`params`) permitem listas de argumentos de tamanho variável.
- **Efeitos Colaterais Funcionais**: Ocorrem quando uma função modifica um parâmetro passado por referência ou uma variável global. **Funções puramente funcionais** (ex: Haskell) não possuem variáveis mutáveis e, portanto, não produzem efeitos colaterais.

Ambientes de Referenciamento Local

- **Variáveis Locais**: Definidas dentro de subprogramas.
    - **Dinâmicas da Pilha**: Criadas a cada chamada e destruídas ao final. Permitem recursão. C, C++, Java, C# e Python as usam por padrão.
    - **Estáticas**: Criadas uma única vez e mantêm seu valor entre chamadas. Não suportam recursão.
- **Subprogramas Aninhados**: Definidos dentro de outros subprogramas. Permitem uma hierarquia de escopos e acesso a variáveis não locais dos subprogramas envolventes (escopo estático). Permitidos em descendentes de ALGOL 60, e em linguagens modernas como JavaScript, Python e Ruby.

Métodos de Passagem de Parâmetros

- **Modelos Semânticos**:
    - **Modo de Entrada (*in mode*)**: Recebe dados, mas não devolve alterações.
    - **Modo de Saída (*out mode*)**: Envia dados de volta, sem receber valores iniciais.
    - **Modo de Entrada e Saída (*inout mode*)**: Recebe valor inicial e retorna o valor atualizado.
- **Modelos de Implementação**:
    - **Passagem por Valor**: O valor do parâmetro real é copiado para o formal. Implementa a semântica de *in mode*. Eficiente para tipos escalares, mas ineficiente para grandes estruturas (custo da cópia).
    - **Passagem por Referência**: É transmitido o endereço da célula de memória do parâmetro real. Implementa a semântica de *inout mode*. É eficiente em tempo e espaço (sem cópia), mas o acesso é indireto (mais lento) e pode criar **apelidos** (*aliases*), prejudicando a confiabilidade.
    - **Passagem por Atribuição (Python/Ruby)**: O valor do parâmetro formal é atribuído ao real. Objetos mutáveis (listas) podem ser alterados, mas imutáveis (inteiros) resultam em novo objeto.
- **Linguagens Comuns**:
    - **C**: Passagem por valor (a referência é simulada por ponteiros).
    - **C++**: Permite passagem por valor, por referência (com `&`) e por ponteiro.
    - **Java**: Todos os parâmetros são passados por valor. Objetos são passados por valor da referência, podendo o objeto ser alterado.
    - **C#**: Passagem por valor (padrão) ou por referência (com `ref`).

Tópicos Avançados

- **Subprogramas Sobrecarregados**: Subprogramas com o mesmo nome, mas perfis de parâmetros (número, ordem ou tipos) diferentes. O compilador tenta identificar a versão correta.
- **Fechamentos (*Closures*)**: Uma função aninhada mais o ambiente de referência onde foi criada. Permite que a função "lembre" e acesse variáveis do escopo externo (variáveis não locais), mesmo após o escopo ter sido encerrado.

---

4. Tratamento de Exceções (Slides 09)

Exceção é qualquer evento não usual, errôneo ou não, detectável por *hardware* ou *software*, que exija **processamento especial**.

Conceitos Básicos

- **Tratamento de Exceção**: O processamento especial feito por um **tratador de exceção**.

- **Levantar/Lançar Exceção (*Raise/Throw*)**: Quando o evento ocorre.
- **Vantagem**: Evita código redundante de verificação manual e facilita a propagação de erros, permitindo reuso de tratadores e isolamento da lógica de erro.

Questões de Projeto

- **Tipos de Exceção**: A linguagem deve permitir exceções **predefinidas** (levantadas pelo sistema, ex: divisão por zero) e **definidas pelo usuário** (levantadas explicitamente no código).
- **Vinculação ao Tratador**: Como a exceção é associada a um tratador (local ou a nível de unidade).
- **Continuação (*Resumption*)**: Após o tratador terminar, a execução continua após o ponto onde ocorreu a exceção, ou o programa/unidade é encerrado (*termination*).

Tratamento em Linguagens Comuns

Linguagem	Estrutura	Mecanismo de Lançamento	Tipos de Exceção	Continuação	Cláusula de Limpeza	Observações
C++	`try`/`catch`	`throw [expressão]`	Apenas definidas pelo usuário	**Término** (após `catch`, continua após o `try`)	Não possui	A busca por `catch` segue a ordem sequencial; o tipo da expressão `throw` determina o tratador.
Java	`try`/`catch`	`throw new Exception()`	Predefinidas e definidas pelo usuário; verificadas (*checked*) e não verificadas (*unchecked*)	**Término** (após `catch`, continua após o `try`)	`finally`	O `throws` na assinatura do método obriga o tratamento ou declaração de exceções verificadas. Inclui a classe `Throwable` como base.

Python	`try`/ `except`/ `else`/ `finally`	`raise Exception`	Todas são objetos, derivam de `BaseException`.	**Término** (após `except`, continua após o `try`)	`finally`	A cláusula `else` é executada se nenhuma exceção for levantada.
Ruby	`begin`/ `Rescue`/ `else`/ `ensure`/ `end`	`raise`	A maioria herda de `StandardError`.	**Término**	`ensure` (equivalente ao `finally`)	Permite `retry` no final do tratador.

Sentença `assert`

Usada para **programação defensiva**. Verifica se uma condição é verdadeira, lançando uma `AssertionError` se for falsa. Pode ser desativada em produção.

---

5. Programação Orientada a Objetos (Slides 10)

Linguagens OO aplicam o princípio da **abstração** a coleções de tipos de dados, com foco em **Herança** e **Vinculação Dinâmica**.

Fundamentos

- **Classe/Objeto**: A **classe** é a implementação de um Tipo Abstrato de Dado (TAD) ; o **objeto** é sua instância.
- **Tipos Abstratos de Dados (TAD)**: Unem dados e operações (métodos) em uma unidade encapsulada.
    - **Encapsulamento**: Une dados e operações em uma unidade sintática.
    - **Ocultação de Informação**: Restringe o acesso aos dados internos, expondo apenas os **métodos públicos**.
    - **Níveis de Acesso**: `public`, `private` e `protected` (visível apenas para subclasses).
- **Herança**: Permite que uma nova classe (subclasse) herde dados e operações de uma classe existente (superclasse).

    - **Reuso e Especialização**: A subclasse pode adicionar ou modificar (*sobrescrever*) métodos e variáveis.
    - **Herança Simples vs. Múltipla**: Herança simples deriva de uma classe pai; múltipla, de duas ou mais. Linguagens como Java e C# a proíbem em classes, mas a simulam com **Interfaces**.
- **Vinculação Dinâmica (*Dynamic Dispatch*)**: O método a ser executado é determinado em **tempo de execução**, permitindo **polimorfismo**.
    - **C++**: Requer a palavra-chave `virtual` para vinculação dinâmica (caso contrário, é estática em tempo de compilação).
    - **Java**: Todas as chamadas de métodos são dinamicamente vinculadas por padrão, exceto para métodos `final`, `static` ou `private`.
    - **Classes Abstratas**: Classes que contêm métodos abstratos (sem corpo), que devem ser implementados pelas classes derivadas. Em C++, o método é declarado como *virtual pura* (`=0`).

Questões de Projeto

- **Exclusividade dos Objetos**: Smalltalk e Ruby tratam *todos* os dados como objetos; C++, Java e C# mantêm tipos primitivos separados por questões de eficiência.
- **Subclasses vs. Subtipos**: O **Princípio de Substituição de Liskov** afirma que objetos da subclasse devem poder ser usados onde a classe pai é esperada, sem alterar o comportamento do programa.
- **Alocação de Objetos**:
    - **C++**: Objetos podem ser estáticos, dinâmicos da pilha ou dinâmicos do *heap* (liberação explícita com `delete`).
    - **Java, C#, Ruby, Smalltalk**: Objetos são dinâmicos do *heap* e a liberação é implícita (coleta automática de lixo).
- **Inicialização**: Em C++, Java, C# e Ruby, construtores são chamados implicitamente.

---

6. Linguagens de Programação Funcional (Slides 12)

O paradigma funcional, inspirado nas **funções matemáticas**, surgiu como alternativa às linguagens imperativas.

Características Principais

- **Modelo**: Baseado na **avaliação de expressões** e funções, e não em comandos ou alteração de estado.
- **Funções Matemáticas**: Mapeamento entre um **Domínio** (entrada) e uma **Imagem** (saída).

    - **Ausência de Efeitos Colaterais**: Funções matemáticas (e puras) não dependem de valores externos e, para o mesmo elemento do domínio, sempre produzem o mesmo elemento da imagem.
    - **Ausência de Estado/Variáveis Mutáveis**: Não existe o conceito de variável que modela uma posição de memória.
- **Cálculo Lambda**: Sistema formal (Alonzo Church) que é a base teórica das linguagens funcionais modernas.
- **Funções de Ordem Superior (*Higher-Order Functions*)**: Funções que recebem uma ou mais funções como parâmetro, ou retornam uma função como resultado (ou ambos).
    - **Composição Funcional**: Combina duas funções para formar uma terceira (o resultado de uma é a entrada da outra).
    - **Aplicar-para-todos (`map`)**: Recebe uma função e a aplica a cada elemento de uma lista, retornando uma nova lista com os resultados.
- **Imutabilidade**: Em linguagens funcionais puras, criam-se **ligações imutáveis** (*bindings*). Uma vez definido, um nome nunca muda de valor.
- **Transparência Referencial**: Uma função pura sempre retorna o mesmo resultado para as mesmas entradas e não causa efeitos colaterais. Isso torna a semântica mais simples e a execução **determinística**.
- **Repetição**: É expressa por **recursão**, não por iteração (laços).
- **Conciliação**: Linguagens funcionais como Haskell tendem a ser mais concisas que as imperativas. Essa ausência de estado facilita a **programação concorrente** e o paralelismo.

Evolução

- **Lisp**: A primeira linguagem funcional pura, projetada no final da década de 1950.
- **Modernas**: Linguagens como ML, Haskell, OCaml e F# ampliaram o alcance do paradigma.
- **Híbridas**: Muitas linguagens funcionais modernas (e imperativas, como Python e C#) incluem recursos de ambos os paradigmas (ex: variáveis mutáveis e funções *lambda*).