Paradigma Imperativo

Eduardo Feitosa
efeitosa@icomp.ufam.edu.br

Máquina de Turing

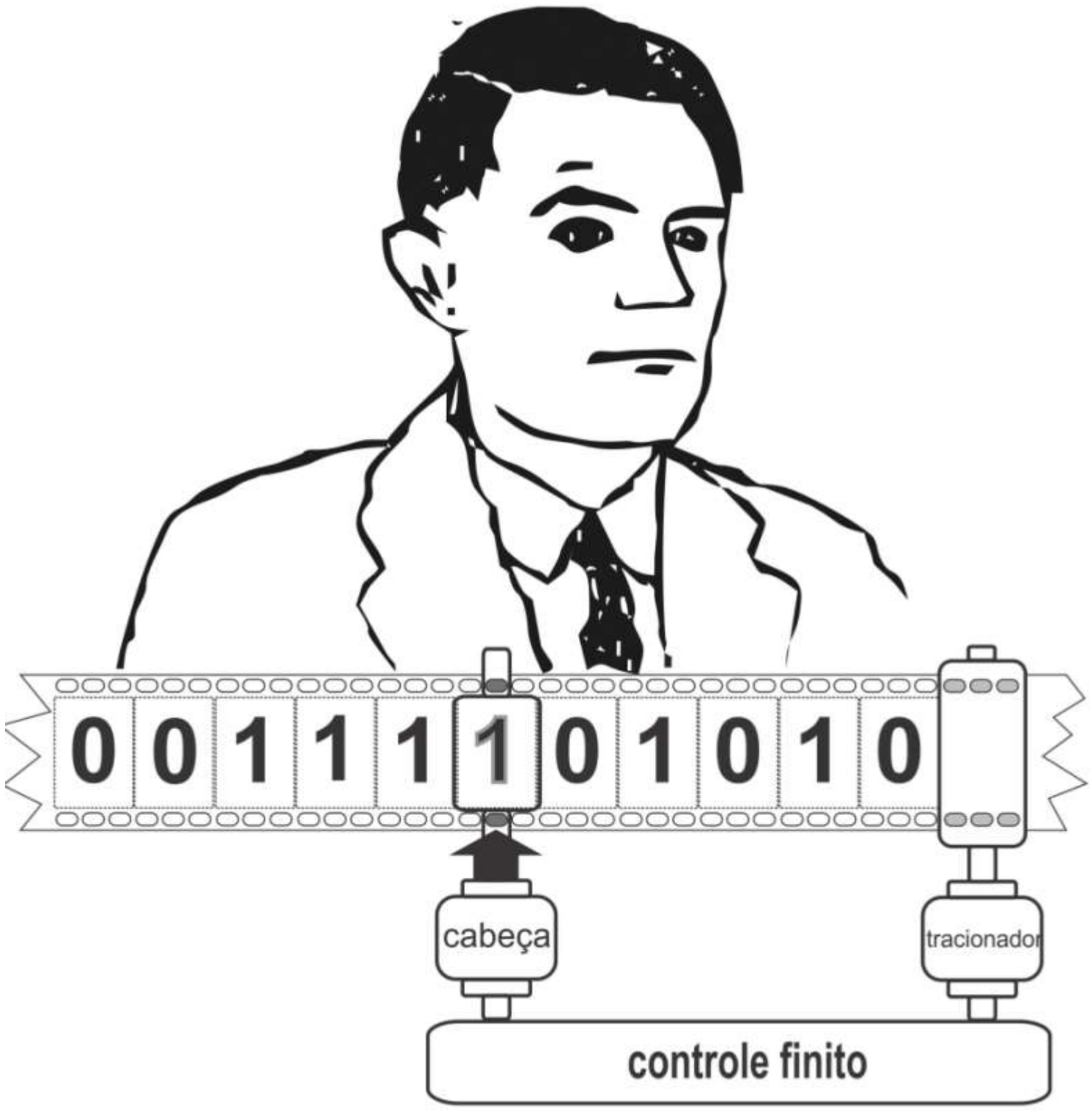
Máquina de Turing

- Modelo matemático proposto por Alan Turing por volta de 1936
  - Aproximadamente seis anos antes do Colossus
- Objetivo
  - Formalizar o conceito de **algoritmo**
  - Solucionar o desafio do "problema de decisão" (*Entscheidungsproblem*) que havia sido proposto por David Hilbert
  - Definir os limites da computabilidade

Máquina de Turing

- As máquinas de Turing são máquinas de estados que possuem
  - Uma fita de tamanho infinito
  - Uma cabeça de leitura e escrita
  - Um conjunto finito de estados internos
    - Inicial, de aceitação e de rejeição
  - Um conjunto finito de regras de transição

Máquina de Turing



Algoritmos e Máquina de Turing

- Uma máquina de Turing que sempre **para** para qualquer entrada é um "modelo de algoritmo"

- Todo algoritmo possui uma máquina de Turing equivalente que **para** para qualquer entrada

- Portanto, algoritmos e máquinas de Turing são conceitos equivalentes

Linguagens e Máquina de Turing

- Máquinas de Turing, por sua vez, são equivalentes a gramáticas livres de contexto

- Uma máquina de Turing pode ser usada para
  - Descrever uma linguagem
  - Verificar se uma cadeia pertence à linguagem

- Portanto, um algoritmo pode ser usado para
  - Descrever uma linguagem
  - Verificar se uma cadeia pertence à linguagem

Máquina de Turing Universal

- Uma Máquina de Turing pode ser entrada de outra máquina de Turing

- Uma **máquina de Turing universal** U é uma máquina de Turing que aceita como entrada
  - A descrição de uma máquina de Turing M
  - Uma cadeia de entrada w

- Simula a execução de M com a cadeia de entrada w
  - Um computador é "equivalente" a uma máquina de Turing universal (com memória finita)

Analogamente

- Podemos pensar em um computador como algo que se comporta mais ou menos como uma máquina de Turing universal
  - A memória é equivalente à fita
  - A memória contém o programa a ser executado e também os dados do programa
  - O hardware do computador faz o papel da cabeça da máquina de Turing, com suas próprias transições e estados para executar os nossos códigos, manipular a memória, fazer operações de E/S etc.

Paradigma Imperativo

Introdução

- A arquitetura dos computadores exerceu grande efeito sobre o projeto das LPs.
  - A maioria delas, nos últimos 35 anos, foi projetada em torno da "máquina de von Neumann".

- Nesta arquitetura, tanto os dados como os programas são armazenados na mesma memória e a CPU é separada da memória.
  - Instruções e dados devem ser transportados, ou transmitidos, da memória para a CPU.
  - Os resultados das operações realizadas na CPU devem ser devolvidos para a memória.

Introdução

- LPs criadas para esta arquitetura são chamadas de imperativas ou procedurais, e compartilham as seguintes características:
  - As variáveis modelam as células de memória;
  - Os comandos de atribuição são baseados nas operações de transferência dos dados e instruções;
  - Os operandos das expressões são passados da memória para a CPU, e o resultado da expressão é passado de volta para a célula de memória, representada pelo lado esquerdo do comando de atribuição;

Introdução

- A execução é sequencial de instruções;
- A forma iterativa de repetição é o método mais eficiente desta arquitetura;
- A iteração é rápida em computadores com este tipo de arquitetura porque as instruções são armazenadas em células adjacentes da memória. Essa eficiência desencoraja o uso da recursão para repetição.

Introdução

- É interessante notar que algumas vezes as linguagens de programação imperativas são também chamadas de procedurais, mas isto não tem relação com o conceito de **procedimento**.

- O paradigma imperativo de linguagens de programação pode ser encontrado, por exemplo, nas linguagens Fortran, Cobol, Basic, Pascal, Modula-2, C e Ada.

Paradigma Imperativo

- O paradigma imperativo de programação é o mais antigo de todos os paradigmas de programação.

- Baseia-se no modo de funcionamento do computador, ou seja, é influenciado pela arquitetura do computador.

- Isto é refletido na execução sequencial baseada em comandos e no armazenamento de dados alterável, conceitos que são baseados na maneira como computadores executam os programas no nível de linguagem de máquina.

Paradigma Imperativo

- O termo “imperare” em Latim significa “comandar”.

- O paradigma imperativo foi predominante nas LP, pois tais linguagens são mais fáceis de traduzir para uma forma adequada para execução na máquina.

- Um programa desenvolvido a partir deste modelo, por exemplo nas linguagens C e Modula-2, consiste em uma sequência de modificações no armazenamento de dados na memória do computador.

Paradigma Imperativo

- As linguagens imperativas são caracterizadas por três conceitos:
  - **Variáveis** - mantém o estado de um programa imperativo e são associadas com localizações de memória que correspondem a um endereço e um valor de armazenamento.
    - O valor da variável pode ser acessado e alterado direta ou indiretamente, através de um comando de atribuição ou de um comando de leitura.

Paradigma Imperativo

- As linguagens imperativas são caracterizadas por três conceitos:
  - **Atribuições** - O comando de atribuição introduz uma dependência de ordem no programa: o valor de uma variável é diferente antes e depois de um comando de atribuição.
  - **Sequência** - O resultado do processamento de um programa depende da ordem na qual os comandos são escritos e executados, ou seja, da sequência na qual os comandos estão escritos.

Paradigma Imperativo

- As funções de linguagens imperativas são descritas como **algoritmos** que especificam como processar um intervalo de valores, a partir de um valor de domínio, com uma série de passos prescritos.

- A repetição, ou laço, é usada extensivamente para processar os valores desejados.
  - Laços são usados para varrer uma sequência de localizações de memória, tal como vetores, ou para acumular um valor em uma variável específica.
  - Por essas características, têm sido chamadas de “baseadas em comandos” ou “orientadas a atribuições”.

Comparativo entre LP Imperativas

	FORTRAN	PASCAL
- valores e tipos - expressões	- tipos: integer, real, double precision, complex, logical; vetor: dimension <nome> (dim1, dim2, dim3), real, integer - constantes lógicas: .true., .false. - operadores: **, *, /, +, -, .ge., .gt., .le., .lt., .eq., .ne., .not., .and., .or.	- tipos: simples: boolean, integer, char, real; estruturados: array, file, record, set - expressões: boolean: and, or, not, xor integer: +, -, *, div, mod, =, >=, <>. abs, sqr, trunc, round string: var a: string; a = 'abc'; file: type arq = file of integer;

- comandos
e seqüências
	if (<exp>) then
	...
	end if
	if (<exp>)
	...
	else if ( ) then
	...
	else
	...
	end if
	- comandos I/O: open, close, read, write, print, rewind, endfile
	- goto, continue, pause, stop
		- comandos simples: write, writeln, read, clrscr, gotoxy, delay, readkey, upcase
		if-then, if-then-else
		case <exp> of
		case <op1>:
		case <op2>:
		else
		end
		for x := <inic> to|downto <fim> do
		while <cond> do
		begin
		...
		end;

- declarações - procedimentos e funções	- declaração var.: <tipo> <id> - funções: function <id> (<parâm.>) - proc.: procedure <id> (<p.>)	- declaração var.: var <id>: <tipo>; - procedimentos e funções: procedure <id> (<parâm.>); function <id> (<parâm.>): <tipo>;
	versões anteriores a FORTRAN 90: somente letras maiúsculas; outras versões traduzem para maiúsculas durante a compilação	não é case sensitive

Comparativo entre LP Imperativas

	C	ADA
- valores e tipos - expressões	- tipos: char, int, float, double, struct, union - operadores: -, +, *, /x, %, ++, >, >=, =, !=, &&, ||, !, & bit a bit: &, |, ^, ~, >>, <<	- tipos: array: <id> array(x..y) of <tipo> integer, natural, positive, character, boolean, string: type string is array(x..y) of character; float - operadores: =, /=, >, >=, +, -, abs, **, and, or, xor, not

- comandos e seqüências	if (<exp.>) <comandos>
	else <comandos>
	for (inic; cond; incremento)
	
	switch (<exp>){
	case <op1>: ...; break;
	case <op2>: ...; break;
	default: ...;
	}
	while (<cond.>){ ....;}
	do
	....
	while (<cond.>);
	return <exp>, goto <tótulo>, break
		if <cond> then
		elsif <cont> then
		else
		end if;
		
		case <exp> is
		<alt. 1>
		<alt. 2>
		end case;
		
		when <lista escolha> => <com.>
		when <others> => <comandos>
		
		while <cond.> loop
		<comandos>
		end loop;
		for <id> in <interv> loop
		<comandos>
		end loop

- declarações - procedimentos e funções	- declaração var.: <tipo> <lista variáveis>; - constantes: const <tipo> <id> = <valor>; - funções: <tipo> <id> (<parâm>)	- declaração var.: <id>: <tipo>; - procedimentos: procedure <id> (<parâm>) is - funções: function <id> (<p.>) return <tipo>
	é case sensitive	

Exemplos de LPs Imperativas

Python: uma Linguagem Simples

- Projetada por Guido van Rossum em 1990

- Projetada para ser simples
  - Princípio norteador: “deve haver um jeito óbvio (e preferencialmente apenas um) de se fazer alguma coisa em Python”

- Desenvolvimento contínuo
  - PEP (*Python Enhancement Proposal*)
  - Propostas da comunidade para acrescentar funcionalidades ou corrigir problemas do Python

Python: algumas PEPs

- PEP 8: estilo de código
  - Define um estilo de código comum para todos os programadores
  - Usar espaços ou TAB para indentar?
  - Onde usar linhas em branco?

- PEP 20: o “Zen” do Python
  - Define os princípios norteadores da linguagem

- PEP 3099
  - Coisas que não vão mudar no Python 3000

Python: Fluxo sequencial

```
#!/usr/bin/python3

def fib(n):
    if n <= 1:
        return 1
    else:
        return fib(n - 1) + fib(n - 2)

s = input("Digite um n: ")
x = int(s)
fx = fib(x)
print("Fibonacci de {x} e' {fx}")
```

Perl: uma Linguagem expressiva

- Projetada por Larry Wall em 1987

- Projetada para substituir awk e sed

- Designada para ser prática (fácil de usar, eficiente, completa) em vez de elegante

- Muito influenciada por comandos de terminal
  - Chamadas de função não exigem parênteses

Perl: uma Linguagem expressiva

- Perl possui três tipos de dados
  - **$x** é um escalar
    - Pode ser uma string, um inteiro, um ponto flutuante ou uma referência
  - **@x** é um vetor
    - Pode conter qualquer elemento escalar
    - Equivalente às listas de Python
  - **%x** é uma hash
    - Contém entradas do tipo chave => valor
    - Equivalente aos dicionários de Python

Perl: Fluxo sequencial

```
#!/usr/bin/perl

@vetor = (1, 2, 3, 4);
$numel = $#vetor;
print "O vetor contem $numel elementos:\n";
print @vetor;
print "\n";

$primeiro = $vetor[0];
print "O primeiro elemento da lista eh $primeiro\n";

$primeiro = shift @vetor;
print "O primeiro elemento da lista era $primeiro:\n";
print @vetor;
print "\n";
```

Perl: Fluxo sequencial

```perl
#!/usr/bin/perl

@vetor = (1, 2, 3, 4);
$numel = $#vetor;
print("O vetor contem $numel elementos:\n");
print(@vetor);
print("\n");

$primeiro = $vetor[0];
print("O primeiro elemento da lista eh $primeiro\n");

$primeiro = shift(@vetor);
print("O primeiro elemento da lista era $primeiro:\n");
print(@vetor);
print("\n");
```

O mesmo código, com os parênteses opcionais

C: uma linguagem “genérica”

- A linguagem C foi criada por Dennis Ritchie e Brian Kernighan em 1972

- Uma linguagem de baixo nível, baseada em B, com tipos de dados
  - Apesar de não impor muitas restrições

- Programação estruturada
  - Todos os programas contêm uma função chamada main()
  - A partir do seu ponto de entrada, a execução segue o fluxo sequencial

C: uma linguagem “genérica”

- C foi projetada para ser compilada
- Todas as variáveis devem ser declaradas
  - Os tipos das variáveis devem ser conhecidos em tempo de compilação
  - As variáveis não podem mudar de tipo durante a execução
  - Tipagem estática
  - Tipagem relativamente fraca
  - Mais fraca que Java, mais forte que JavaScript

C: uma linguagem “genérica”

- Atualmente a linguagem C é um padrão ISO
  - International Organization for Standardization ou Organização Internacional para Padronizações
  - ISO/IEC 9899:2011 (C11)

- A linguagem C foi base da linguagem C++
  - C++ também é hoje um padrão ISO
  - ISO/IEC 14882:2017 (C++17)

C++: linguagem “para a todos conquistar”

- C++ cresceu muito desde sua versão inicial em 1985

- Sofreu influência de muitas linguagens
  - Ada, ALGOL 68, C, CLU, ML, Simula, …

- É um “monstro” multi-paradigma
  - Procedimental
  - Orientado a objetos
  - Funcional
  - Paralelo
  - Genérico

C++: linguagem “para a todos conquistar”

```cpp
#include <iostream>

using std::cout;
using std::cin;

int main()
{
    int n, i, j;
    cout << "Digite um numero\n";
    cin >> n;
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++)
            cout << '*';
        cout << '\n';
    }
    return 0;
}
```

Expressões e Sentenças de Atribuição

Introdução

- **Expressões** são os meios fundamentais de especificar computações em uma LP.
  - É crucial entender tanto a sintaxe quanto a semântica de expressões das linguagens.
  - Isso é um exemplo de **ortogonalidade**

- Para entender a avaliação de expressões, é necessário conhecer as ordens de avaliação de operadores e operandos.
  - A ordem de avaliação de operadores de expressões é ditada pelas regras de associatividade e de precedência da linguagem.

Introdução

- A ordem de avaliação dos operandos em expressões pode não ser mencionada no projeto da linguagem.
  - Programadores escolhem a ordem, o que leva à possibilidade de os programas produzirem resultados diferentes em implementações diferentes.

- A essência das LPs imperativas é o papel dominante das sentenças de atribuição.
  - A finalidade é causar o efeito colateral de alterar os valores de variáveis, ou o estado, do programa.
  - Parte integrante das LPs imperativas é o conceito de variáveis cujos valores mudam durante a execução.

Expressões Aritméticas

- Avaliação de expressões aritméticas foi a motivação para o desenvolvimento da primeira linguagem de programação;

- Expressão aritmética consiste de:
  - operadores e operandos,
  - parênteses
  - chamadas de função

- Exemplo
  - **Double A = (3.2 + 8.4) / B;**

Expressões Aritméticas

- Considerações de projeto:
  - Quais são as regras de precedência de operadores?
  - Quais são as regras de associatividade de operadores?
  - Qual é a ordem de avaliação dos operandos?
  - Existem restrições quanto aos efeitos colaterais da avaliação dos operandos?
  - A linguagem permite sobrecarga de operadores definidas pelo usuário?

Expressões Aritméticas

- Operadores:
  - Unário
    - possui apenas um operando
    - **Ex: A ++**
  - Binário
    - possui dois operandos
    - **Ex: A * B**
  - ternário
    - possui 3 operandos
    - (condição) ? Verdadeiro : Falso
    - **Ex: (A < B) ? 1 :**

Ordem de Avaliação de Operadores

- Como deve ser avaliada a seguinte expressão:
  - a + b * c** d** e / f

- Para isso temos:
  - Regra de Precedência
  - Regra de Associatividade
  - Parênteses
  - Expressões Condicionais

Regra de Precedência de Operadores

- **Regra de Precedência de Operadores** para avaliação de expressões definem a ordem na qual operadores adjacentes de diferentes níveis de precedência são avaliados

- Níveis de precedência típicos
  - parênteses
  - Operadores unários
    - ** (se a linguagem suporta exponenciação)
    - *, /
    - +, -
    - ==, > , < , <= , >=

Regra de Precedência de Operadores

- Uma expressão típica em C

```
int numeros[] = {1, 3, 6, 16, 32, 42};
int *ptr = &numeros[3];
```

- A linguagem assegura que o operador **&** tem menor precedência que o operador **[]** para que a expressão acima signifique o "endereço do quarto elemento"

Regra de Precedência de Operadores

- Uma expressão típica em Perl

```
$nome_completo =~ /\s*(\w+)\s+(\w+)/ or die "No match";
$nome = $1;
$sobrenome = $2;
```

- O operador **or** tem menor precedência que os outros operadores
  - O comando **die** só é executado se a expressão à esquerda retornar um valor falso

Regra de Precedência de Operadores

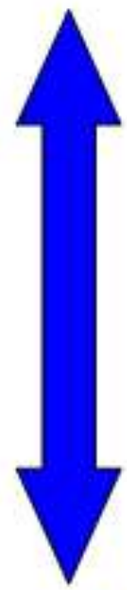
- As regras de precedência da linguagem influenciam muito na facilidade de escrita e na facilidade de leitura

- Algumas regras são intuitivas ou devem ser internalizadas pelos programadores
  - &estrutura->vetor[10]

- Em outros casos, é melhor utilizar parênteses
  - terceiro_bit = (num >> 2) & 1

Regra de Precedência de Operadores

MAIS ALTA



MAIS BAIXA

FORTRAN	PASCAL	C	Ada
**	*, /, **div, mod**	++ , -- (pós-fixo)	**, **abs**
*, /	+ , - (todos)	++ , -- (prefixo)	*, /, **mod**
+, - (todos)		+ , - (unário)	+, - (unário)
		*, / , %	+, - (binário)
		+, - (binário)	

Regra de Associatividade

- Regras de associatividade definem a ordem em que os operadores adjacentes com mesmo precedência são avaliados (Ex: A + B – C + D)

- Regras de associatividade comuns
  - Da esquerda para a direita,
  - exceto **, Qual é da direita para a esquerda a esquerda
    - B + A ** C

- Regras de precedência e de associatividade podem ser substituídas pelo uso de parênteses
  - Ex: (A + B) * C

Regra de Associatividade

- Em FORTRAN
  - Alguns operadores unários associam-se da direita para esquerda
- Em APL
  - Todos os operadores têm precedência iguais
  - São associativos da direita para a esquerda

Linguagem	Regra de Associatividade
FORTRAN	Esquerda : *, / , + , - Direita : **
Pascal	Esquerda : Todos
C	Esquerda : ++ pós-fixo, -- pós-fixo, * , / , % , + binário , - binário Direita : ++ prefixo, -- prefixo, + unário , - unário
C++	Esquerda : * , / , % , + binário , - binário Direita : ++, -- , + unário , - unário
Ada	Esquerda : todos, exceto ** Direita : **

Expressões Condicionais

- Operador ternário ? :

- C, C++ e Java

Exemplo:

res = (cont == 0)? 0 : soma/cont

Equivalente – if-then-else

if (cont == 0)

res = 0

else res = soma /cont

Ordem de avaliação de operandos

- **Variáveis**
  - Buscar seu valor na memória

- **Constantes:**
  - Algumas vezes é necessário buscar na memória;
  - Outras vezes a constante está na própria instrução de máquina.

- **Expressões parêntizadas**
  - avaliar todos operandos primeiro antes que seu valor possa ser usado como operando

Efeitos Colaterais

- Ocorre quando uma função altera um de seus parâmetros
- ou uma variável não local
- <u>Exemplo:</u> Quando uma função é chamada em uma expressão e altera outro operando da expressão:
  - a = 10;
    - /*a função fun( ) retorna o valor do argumento divido por 2 e modifica o parâmetro*/
  - b = a + fun(a);
  - Se o valor de a for buscado primeiro a = 10 + 5 = 15
  - Mas se o segundo operando for avaliado primeiro, o valor do primeiro será 5 + 5 = 10

Efeitos Colaterais

- Variável Global

```
Int a = 5;
Int fun1( ) {  a = 17;  return 3;  }
Int fun2( ) {  a = a + fun1( );   }
void main( ) {  fun2( );   }
```

- O valor computado em fun2( ) depende da ordem de avaliação dos operandos na expressão
  - a + fun1 ( )

Efeitos Colaterais

- Possíveis Soluções
  - O projetista da linguagem poderia impedir que a avaliação da função afetasse o valor das expressões.
  - Declarar, na definição da linguagem, que os operandos devem ser avaliados em uma ordem particular
  - Exigir que os programadores garantam esta ordem

- Rejeitar os efeitos colaterais é difícil e elimina a otimização do programador

Sobrecarga de Operadores

- Usar um operador para mais do que um propósito

- Exemplo
  - + para adição de quaisquer operandos de tipo numérico
  - int e float

- Em Java (+) para concatenar cadeias.
- Em C:
  - A = B * C // Multiplicação
  - A = * ref; // Referência

Sobrecarga de Operadores

- Alguns representam problemas em potencial
  - Perda da capacidade de detectar erros
  - Omissão de um operador
  - Podem ser evitados pela introdução de novos símbolos

- Exemplo:
  - media = soma / cont; // int ou float
  - div para divisão de inteiros no Pascal

Sobrecarga de Operadores

- C++ e Ada permitem que programador defina a sobrecarga de operadores.

- Problema potencial:
  - Programadores podem definir sobrecarga de operadores sem sentido;
  - **Legibilidade** pode ficar comprometida.

Conversões de Tipo

- Uma conversão de estreitamento transforma um valor para um tipo que não pode armazenar todos os valores do tipo original
  - **float para int**

- Uma conversão de alargamento transforma um valor para um tipo que pode incluir, pelo menos, aproximações de todos os valores do original
  - **int para float**

- Uma expressão de modo misto é aquela que possui operandos de tipos diferentes

Conversões para Expressões

- Desvantagem de conversão:
  - Diminui poder do compilador na detecção de erros
  - Na maioria das linguagens, todos os tipos numéricos são convertidos em expressões, usando coerção de alargamento
  - No Ada e Modula-2, praticamente, não é permitida conversão em expressões

Conversão de Tipo Explícita

- Chamada de *casting* em linguagens baseadas em C

- Exemplos
  - C: (int) numero
  - Ada: Float (soma)

- Obs: a sintaxe em Ada é similar a chamada de funções

Erros em Expressões

- Erros em Expressões (causados por):
  - Limitações aritméticas:
    - Ex. Divisão por zero
  - Limitações da aritmética computacional:
    - Ex. overflow de inteiros
    - Overflow de ponto flutuante

Expressões Relacionais

- Possui dois operandos e um operador relacional

- Este Compara os valores de seus dois operandos
  - Seu valor é booleano

- Os símbolos de operadores variam bastante entre linguagens

Expressões Relacionais

Operação	**Ada**	**Java**	**FORTRAN 90**
Igual	=	==	.EQ. ou ==
Diferente	/=	!=	.NE. ou <>
Maior que	>	>	.GT. ou >
Menor que	<	<	.LT. ou <
Maior que ou igual	>=	>=	.GE. ou >=
Menor que ou igual	<=	<=	.LE. ou <=

Expressões Booleanas

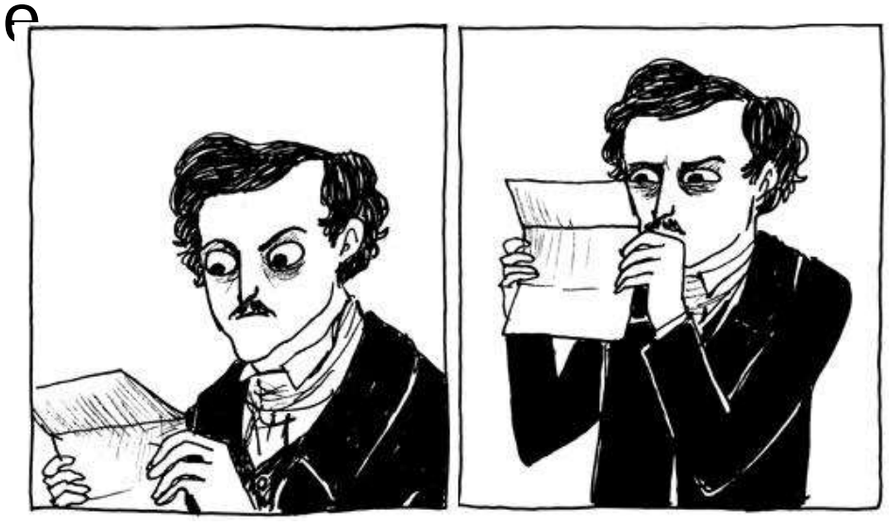
- Operandos são booleanos
- O resultado é booleano

FORTRAN 77	Fortran 90	C	Ada
.AND.	and	&&	and
.OR.	or	||	or
.NOT.	not	!	not
			xor

- Característica do C
  - não possui tipo booleano
  - utiliza o tipo int com 0 para FALSO
  - diferente de zero para VERDADEIRO.
  - A expressão: a < b < c é correta
  - e equivalente a: (a<b) < c

Expressões Booleanas

- Qualquer linguagem “que se preze” oferece operadores booleanos
  - C, C++, Perl, Python, JavaScript etc.
    - A constante “falso” é 0
    - Qualquer outro valor é a constante “verdadeiro”
    - &&, || e ! representam conjunção, disjunção e negação
  - Perl, C++, Python, Pascal etc.
    - A constante “falso” é False e/ou false
    - A constante “verdadeiro” é True e/ou true
    - and, or e not



Operadores Compostos

- É um método abreviado de especificar uma forma de atribuição

- Introduzido em ALGOL; adotado por C

- Exemplo
  - a = a + b
  - É escrito como
  - a += b

Operadores Unários

- Linguagens baseadas em C combinam operações de incremento e de decremento com atribuição

- Exemplos
  - soma = ++ cont (o valor de cont é incrementa em 1, é atribuído a suma)
  - soma = cont++ (atribui a soma , e cont é incrementado)
  - cont++ (cont é incrementado)
  - -cont++ (cont é incremented e depois é transformado em negativos
  - Não (-cont)++

Estruturas de Controle no Nível de Sentença

Níveis de fluxo de controle

- Computações são realizadas por meio da avaliação de expressões e da atribuição dos valores a variáveis

- Para tornar a computação mais flexível e poderosa criou-se:
  - Formas de selecionar entre caminhos alternativos de fluxo de controle (execução da sentença)
  - Execução repetida de sentenças ou de sequência de sentenças chamada de **controle de fluxo**

Níveis de fluxo de controle

- O **controle do fluxo** em um programa ocorre em diversos níveis
  - Dentro das expressões
    - Regras de associatividade
    - Regras de precedência de operadores
  - Entre as unidades de programas
    - Subprogramas
  - Entre as sentenças

Sentenças de controle: Evolução

- Sentenças de controle em FORTRAN I foram baseados diretamente no hardware do IBM 704

- Boa parte da pesquisa e da discussão foi devotada às sentenças de controle nos anos 1960

- **Um resultado importante:** foi provado que todos os algoritmos que podem ser expressos por diagramas de fluxo podem ser codificados em uma linguagem de programação com apenas duas sentenças de controle

Sentenças de controle: Evolução

- Antes, apenas um sentença de controle é necessária, o **goto**, mas esta sentença tem diversos problemas

- O **goto** pode ser substituído por duas sentenças de controle: **seleção** e **repetição**

- Na prática, o aspecto importante não é a quantidade de sentenças de controle, mas a facilidade de leitura e escrita

Estrutura de controle

- Uma **estrutura de controle** é uma sentença de controle e a coleção de sentenças cuja a execução está sob seu controle

- Questão de projeto relativa a todas as estruturas de controle
  - **A estrutura de controle tem múltiplas entradas?**
    - Afetam a legibilidade
    - Ocorrem apenas em LP que incluem *goto* e rótulos (labels) de instruções

Sentenças de Seleção

- Uma **sentença de seleção** provê meios de escolher entre dois ou mais caminhos de execução no programa

- COBOL introduziu o conceito de **IF..ELSE**

- As sentenças de seleção são divididas em duas categorias
  - Duas vias
  - Múltiplas vias

Sentenças de Seleção de duas vias

- As **sentenças de seleção de duas vias** das LPs modernas são bastante semelhantes. A forma geral é:

```
if control_expression
then-clause
else-clause
```

- Questões de projeto
  - Qual é a forma e o tipo da expressão que controla a seleção?
  - Como as cláusulas **then** e **else** são especificadas?
  - Como o significado dos seletores aninhados devem ser especificados?

Sentenças de Seleção de duas vias

- A expressão de controle é especificada entre parênteses se a palavra reservada **then** (ou outro marca sintática) não é usada
  - C - if (expressão) comando; else comando;
  - Pascal - if expressão then comando else comando

- Algumas linguagens permitem expressões aritméticas (C/C++, Python), outras permitem apenas expressões booleanas (Ada, Java, Ruby, C#)

Sentenças de Seleção de duas vias

- Forma da cláusulas **then** e **else**
  - Em Perl, todas as cláusulas precisam ser sentenças compostas
  - A maioria das linguagens permitem sentenças simples e sentenças compostas
    - As linguagens baseadas em C utilizam chaves para formar sentenças compostas
  - Em Fortran 95, Ada e Ruby as cláusulas **then** e **else** são sequências de sentenças.
  - Python usa indentação para isso

```
if x > y:
    x = y
    print("case 1")
```

Seletores Aninhados

- Quando uma construção de seleção é aninhada com a cláusula **then** de outra construção de seleção, não fica claro com qual **if** a cláusula **then** deve ser associada

```
if (sum == 0)
    if (count == 0)
        result = 0;
else
    result = 1;
```

Seletores Aninhados

- Esta construção pode ser interpretada de duas formas

```
if (sum == 0) {
   if (count == 0)
      result = 0;
}
else
   result = 1;
```

```
if (sum == 0) {
   if (count == 0)
      result = 0;
   else
      result = 1;
}
```

Seletores Aninhados

- Em Perl, as cláusulas **then** e **else** têm que ser compostas

- Em Java, a semântica estática específica que a cláusula **else** faz par com a cláusula **then** sem par mais próxima

- Uma alternativa sintática é o uso de uma palavra reservada para marcar o fim do seletor (Fortran 95, Ada, Ruby, Lua)

Seletores Aninhados

- Exemplo em Ruby

```
if sum == 0 then
   if count == 0 then
      result = 0
   else
      result = 1
   end
end
```

```
if sum == 0 then
   if count == 0 then
      result = 0
   end
else
   result = 1
end
```

Seletores Aninhados

- Em linguagens que a indentação é significativa, este problema não existe

- Exemplo em Python

```
if sum == 0:
   if count == 0:
      result = 0
   else:
      result = 1
```

Sentenças de Seleção Múltipla

- Uma construção de **seleção múltipla** permite a seleção de uma entre várias sentenças (ou grupo de sentenças)
- Questões de projeto
  - Qual é a forma e o tipo da expressão que controla a seleção?
  - Como os segmentos selecionáveis são especificados?
  - O fluxo de execução através da estrutura é restrito a incluir apenas um segmento selecionável?
  - Como os valores dos casos são especificados?
  - Como os seletores de valores não apresentados deve ser tratados?

Sentenças de Seleção Múltipla

- Exemplo das linguagens baseadas em C

```
switch (index) {
  case 1:
  case 3: odd += 1;
          sumodd += index;
          break;
  case 2:
  case 4: even += 1;
          sumeven += index;
          break;
  default: printf("Error in switch, index %d\n", index);
}
```

Sentenças de Seleção Múltipla

- Exemplo das linguagens baseadas em C
  - As expressões podem ser do tipo inteiro, caractere ou enumerado
  - Mais que um segmento pode ser executado por vez
  - Nem todos os casos precisam ser especificados

```
switch (index) {
  case 1:
  case 3: odd += 1;
          sumodd += index;
          break;
  case 2:
  case 4: even += 1;
          sumeven += index;
          break;
  default: printf("Error in switch, index %d\n", index);
}
```

Sentenças de Seleção Múltipla

- Exemplo em C#

```
switch (value) {
  case -1:
                    Negatives++;
         break;
  case 0:
         Zeros++;
         goto case 1;
  case 1:
         Positives++;
         break;
  default:
         Console.Writeln("Error in switch\n");
         break;
}
```

Sentenças de Seleção Múltipla

- Exemplo em C#
  - Não é permitido a execução implícita de mais de um segmento
  - As expressões também podem ser strings

```
switch (value) {
  case -1:
                        Negatives++;
           break;
  case 0:
           Zeros++;
           goto case 1;
  case 1:
           Positives++;
           break;
  default:
           Console.Writeln("Error in switch\n");
           break;
}
```

Sentenças de Seleção Múltipla

- Exemplo em Ada

```
case expression is
    when choice list => statement_sequence;
    ...
    when choice list => statement_sequence;
    [when other => statement_sequence;]
end case;
```

- A expressão tem que ser do tipo ordinal
- Apenas um segmento pode ser executado
- Lista de escolhas
  - Um das formas 7, 10..15, 10 | 15 | 20
  - Os valores precisam ser mutuamente exclusivo

Sentenças de Seleção Múltipla

- Exemplo em Ruby (semelhante a **if**s aninhados)

```
leap = case
    when year % 400 == 0 then true
    when year % 100 == 0 then false
    else year % 4 == 0
end
```

- Exemplo em Ruby (semelhante ao **switch**)

```
case int_val
    when -1 then neg_count++
    when 0 then zero_count++
    when 1 then pos_count++
    else puts "Error: int_val is out of range"
end
```

Sentenças de Seleção Múltipla

- Exemplo em Python (seleção múltipla usando if)

```
if count < 10:
    bag1 = True
else:
    if count < 100:
        bag2 = True
    else:
        if count < 1000:
            bag3 = True
```

```
if count < 10:
    bag1 = True
elif count < 100:
    bag2 = True
elif count < 1000:
    bag3 = True
```

Sentenças de Iteração

- Uma **sentença de iteração** é aquela que causa a execução de uma sentença (ou coleção de sentenças) zero, uma ou mais vezes
- Também chamada de laço

- Questões de projeto
  - Como a iteração é controlada?
  - Onde o mecanismo de controle deve aparecer na construção do laço?

Sentenças de Iteração

- O **corpo** de uma construção iterativa é a coleção de sentenças cuja a execução é controlada pela sentença de iteração

- A sentença de iteração junto com o corpo é chamada de **construção iterativa**

Laços controlados por contador

- Uma sentença de iteração de contagem tem uma variável, chamada **variável do laço**, onde o valor da contagem é mantida
- Existe uma maneira de especificar o valor inicial, o valor final e o passo da variável do laço
- Questões de projeto
  - Qual é o tipo e o escopo da variável do laço?
  - A variável do laço pode ser alterada? Isto altera o controle do laço?
  - Os parâmetros do laço devem ser avaliados uma única vez, ou em cada iteração?

Laços controlados por contador

- Exemplos em Fortran 95

```
Do label var = initial, terminal [, stepsize]

Do var  = initial, terminal [, stepsize]
   ...
End Do
```

  - A variável do laço deve ser do tipo inteiro
  - A variável do laço pode não ser alterada
  - Os parâmetros são avaliados apenas uma vez e podem ser alterados
  - Não é possível saltar para dentro do laço

Laços controlados por contador

- Exemplos em Ada

```
for variable in [reverse] discrete_range loop
    ...
end loop;

Count : Float := 1.35;
for Count in 1..10 loop
    Sum := Sum + Count
end loop;
```

  - O tipo da variável do laço é definida pelo intervalo e ela só existe dentro do laço e não pode ser alterada
  - O intervalo pode ser alterado, mas não afeta a execução do laço
  - Não é possível saltar para dentro do laço

Laços controlados por contador

- Exemplo das linguagens baseadas em C:

```
for (exp1; exp2; exp3)
    loop body

// C
for (count = 1; count <= 10; count++) {
  ...
}

// C99, C++, Java
for (int count = 1; count <= len; count++) {
  ...
}
```

Laços controlados por contador

- Exemplo das linguagens baseadas em C:
  - Não existe uma variável de laço específica
  - exp1 é avaliada apenas uma vez antes da execução do laço
  - exp2 e exp3 são avaliadas a cada iteração
  - Qualquer expressão pode ser alterada

```
for (exp1; exp2; exp3)
    loop body

// C
for (count = 1; count <= 10; count++) {
  ...
}

// C99, C++, Java
for (int count = 1; count <= len; count++) {
  ...
}
```

Laços controlados logicamente

- O controle do laço é baseado em uma expressão booleana (não em um contador)

- Questões de projeto
  - O controle deve ser pré-testado ou pós-testado?
  - O laço controlado logicamente deve ser uma forma especial de laço controlado por contador ou uma sentença separada?

Laços controlados logicamente

- Exemplo das linguagens baseadas em C:

```
while (control_expression)
    loop body

do
    loop body
while (control_expression);
```

- Fortran 95 não tem laço lógico

- Ada tem apenas um laço pré-testado

Laços com controles posicionados pelo usuário

- Em algumas situações é conveniente para o programador escolher o local do controle do laço

- Questões de projeto
  - O mecanismo condicional deve ser parte integral da saída?
  - Apenas um corpo de laço pode ser terminado, ou laços externos também podem ser terminados?

Laços com controles posicionados pelo usuário

- Exemplo em Java

```
while (sum < 1000) {
     getnext(value);
     if (value < 0) continue;
     sum += value;
}

while (sum < 1000) {
     getnext(value);
     if (value < 0) break;
     sum += value;
}
```

Laços com controles posicionados pelo usuário

- Exemplo em Java

```
outerLoop:
for (row = 0; row < numRows; row++) {
     for (col = 0; col < numCols; col++) {
          sum += mat[row][col];
          if (sum > 1000.0) {
               break outerLoop;
          }
     }
}
```

Iteração baseada em estrutura de dados

- A iteração é controlada pelos elementos em uma estrutura de dados
  - A função responsável por percorrer a estrutura de dados é chamada **iterador**
- No início de cada iteração, o iterador é chamado e a cada chamada um valor é retornado (em uma ordem específica)

- Devido a flexibilidade do **for** do C, ele pode ser usado para simular uma iteração definida pelo usuário

```
for (ptr = root; ptr != null; ptr = traverse(ptr))
    ...
```

Iteração baseada em estrutura de dados

- Exemplo em Java

```
List<String> lista = Arrays.asList("casa", "janela",
"pé");
for (String x : lista) {
    System.out.println(x);
}
```

- é equivalente a:

```
List<String> lista = Arrays.asList("casa", "janela", "pé");
{
    Iterator<String> iter = lista.iterator();
    while (iter.hasNext()) {
        String x = iter.next();
        System.out.println(x);
    }
}
```

Iteração baseada em estrutura de dados

- O programador pode implementar as interfaces **Iterable** e **Iterator** para que um tipo possa ser usado no **for** (java)

```
public interface Iterable<T> {
    Iterator<T> iterator();
}

public interface Iterator<T> {
    bool hasNext();
    T next();
}
```

Desvio Incondicional

- Uma sentença de **desvio incondicional** transfere o controle da execução para um local específico no programa

- Muito debatido no final da década de 1960

- **goto** na maioria das linguagens

- É a sentença mais poderosa de controle de fluxo

- Linguagens sem goto: Java, Python, Ruby

Comandos Guardados

- Sugerido por Dijkstra em 1975

- A ideia era dar suporte a uma metodologia que garantisse a corretude durante o desenvolvimento

- Outra motivação era o não determinismo, que às vezes é necessário em programas concorrentes

Comandos Guardados

- Construção de seleção

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + i
fi


if x >= y -> max := x
[] y >= x -> max := y
fi
```

Comandos Guardados

- Construção de laço

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

Paradigma Imperativo

Eduardo Feitosa
efeitosa@icomp.ufam.edu.br