



Conceitos Básicos I

Linguagens de Programação

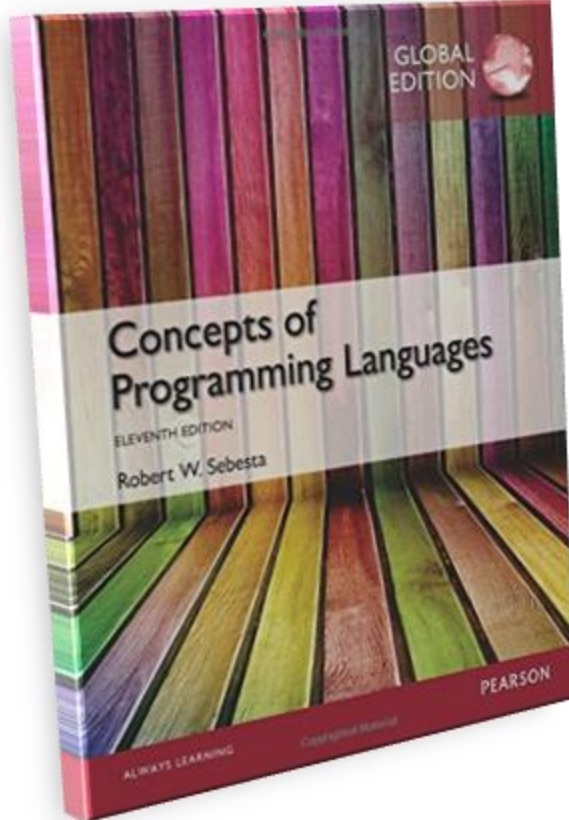
Prof. Ausberto S. Castro V.
ascv@uenf.br

Bibliografia

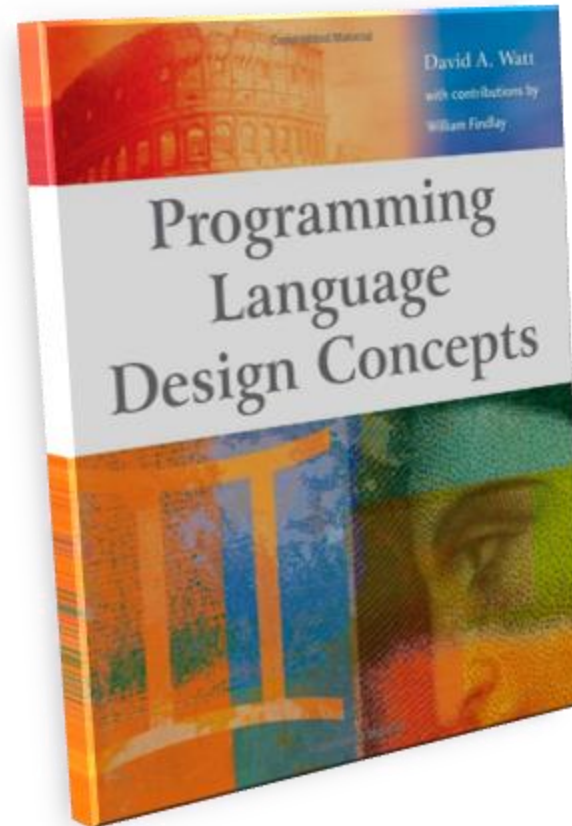


Bibliografia Complementar

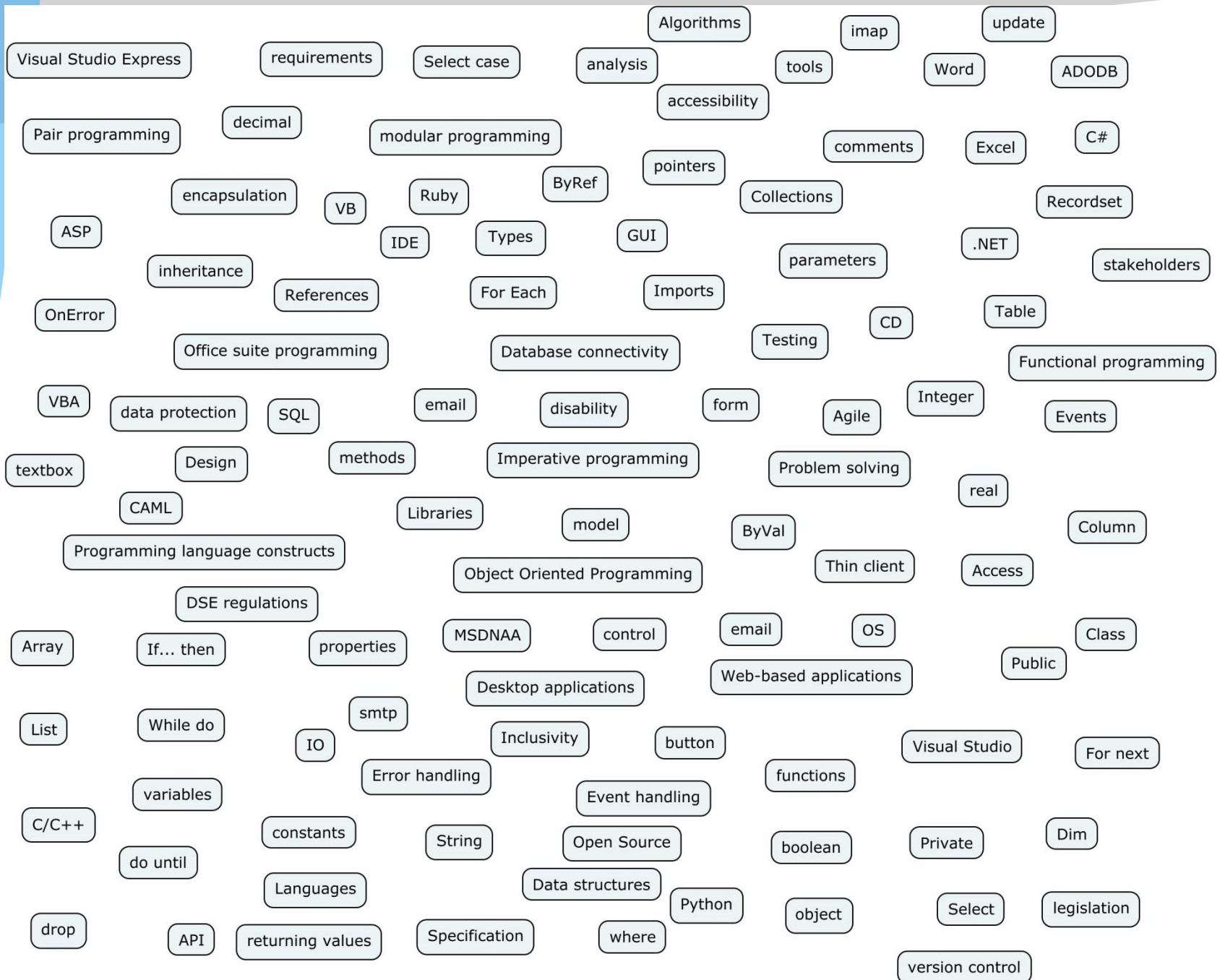
Pearson Education Limited; 11 edition
2016



Wiley; 1 edition (May 21, 2004)



<http://www.levenez.com/lang/>



Conceitos em Programação

- ❖ Linguagem
- ❖ Valores e tipos
- ❖ Variáveis
- ❖ Funções
- ❖ Recursividade
- ❖ Listas
- ❖ Correção
- ❖ Complexidade
- ❖ Estado
- ❖ Concorrência
- ❖ Paralelismo
- ❖ HOP
- ❖ Abstração



Linguagem de Programação



é um artefato !



Linguagem

- ❖ **Definição**
- ❖ **Cada linguagem de programação é um artefato: tem sido conscientemente projetado**
 - Por uma pessoa: Pascal (N. Wirth)
 - Por um grupo de pessoas : Ada (Projetos do DoD)
- ❖ **Características**
 - *Deve ser universal*: solução para qualquer problema
 - *Implementável* em um computador: compilador, interpretador
 - Implementação *eficiente aceitável*: tempo, espaço
 - **Eficiente**: o efeito desejado
 - **Aceitável**: mínimo de tempo e espaço

Valores e tipos

❖ Valor

- Algo que pode ser
 - avaliado,
 - armazenado,
 - incorporado em uma estrutura de dados,
 - passado como argumento de um procedimento ou função,
 - retornado como resultado de uma função
- Uma ENTIDADE que existe durante uma computação

❖ Tipo

- CONJUNTO de valores
 - “ v é um valor de tipo T ” = $v \in T$
 - “A expressão E é do tipo T ” = “o resultado de avaliar E será um valor do tipo T ”

Valores e tipos

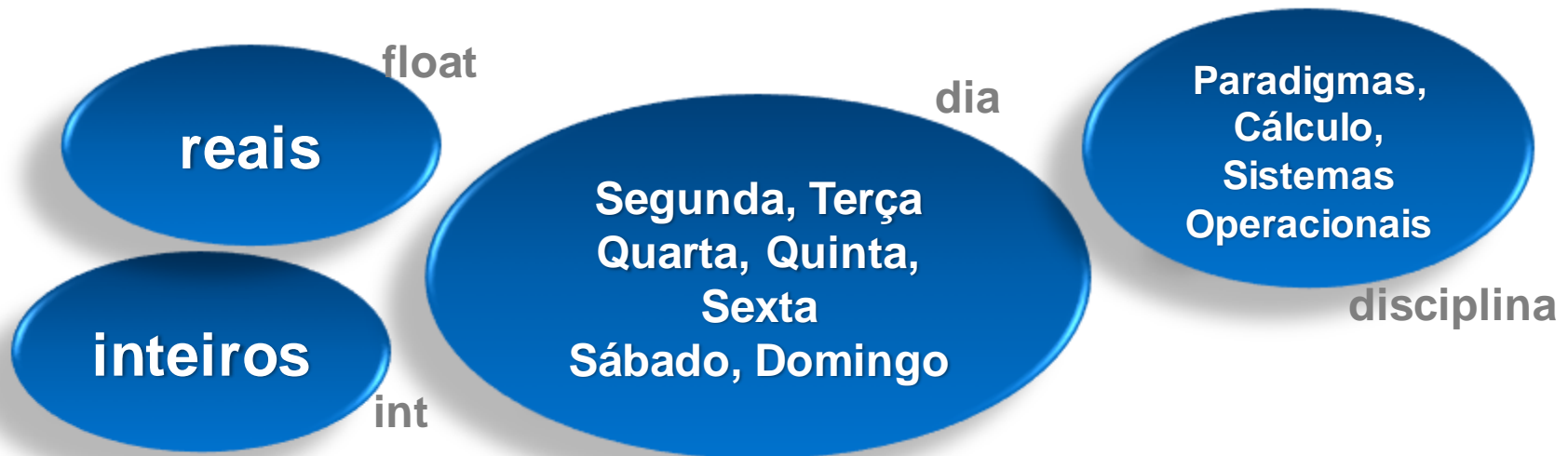
2845	
2846	
2847	73
2848	
2849	

$$K = x^2 + z^3 + 73y;$$

ArmazenaDados('Paradigmas',2008,73,'CC');

X = calculaPeso(Alt,Larg,A); // X=73

código	nome	endereço	idade	salário	peso
5772	Marino		73		



Variáveis

- ❖ **São atalhos para valores (atribuição)**

`x = 71`

`nome = "UNASP"`

`y := 24.7`

`M = [1 2 3; 4 5 6]`

- ❖ **Devem ser declaradas (na maioria da linguagens)**

- Processo de criação de variáveis: reserva de memória

`int x, y`

`REAL a, b, c`

`float k[20], z`

`x, y, j : INTEGER`

- ❖ **Tem um identificador**

- Letra + caracteres permitidos

- ❖ **Ocupa um espaço na memória**

- Pode ser acessado em qualquer momento da execução

Expressões

- ❖ É uma *frase de um programa* que pode ser *avaliada* para obter um *valor*

- 368 $2 * x$ `'UENF'` $k := a + b$

- ❖ **Tipos de expressões**

- Literais
- Agregados
- Chamadas de função
- Expressões condicionais
- Acesso a constantes e variáveis

Expressões

❖ Literais

- A mais simples expressão: indicam explicitamente um valor
- `279` `3.1415` `'m'` `'Brasil'`

❖ Agregados

- Constrói um valor composto a partir de valores componentes
- `(23, x*2.6)` `(31, mes, ano)`

❖ Chamadas de função

- Calcula um valor aplicando uma função a um argumento
- `raizcubica(x)` `altura(h)` `area(a,h)`

Expressões

❖ Expressão condicional

- Contém duas subexpressões das quais somente uma será avaliada
- `if (a > b) then max := a else max := b`
- `max = (a > b) ? a : b`

❖ Acesso a constantes

- `const pi = 3.1415`
- `const nome = 'Pedro Lopes'`

❖ Acesso a variáveis

- `var peso : real`
- `int codigo;`
- `struct cadastro tabela;`

Funções

- ❖ **Segmentos de programa**
 - Conjuntos de instruções que executam UMA tarefa definida
- ❖ **Utilizam parâmetros (como entradas)**
- ❖ **Devolvem um valor (no nome)**

```
(define soma
  (lambda (x y)
    (begin
      (newline)
      (display "A soma = ")
      (+ x y)
    )
  )
)
```

```
function soma (a,b : real) :
real
begin
    soma := a + b;
end
```

```
int fatorial (int n)
{
  If ( n=0 ) { fatorial =1}
  else
    { fatorial = n*
fatorial(n-1) }
}
```

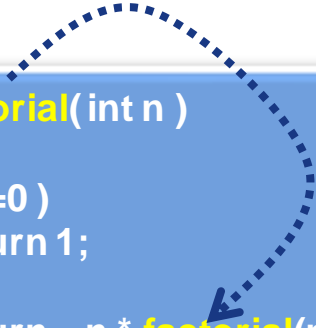
Recursividade

❖ Recursão

- Recursão é um *método de programação* no qual uma função pode chamar a si mesma

❖ Recursividade

- A recursividade nas linguagens de programação envolve a ***definição de uma função*** que pode invocar a si própria.



```
int factorial(int n)
{
    if( n==0 )
        return 1;
    else
        return  n * factorial(n-1);
}
```

Linguagem C



```
(define comprimento
  (lambda (lista)
    (if (null? lista)
        0
        (+ (comprimento (cdr lista)) 1))
  )
)
```

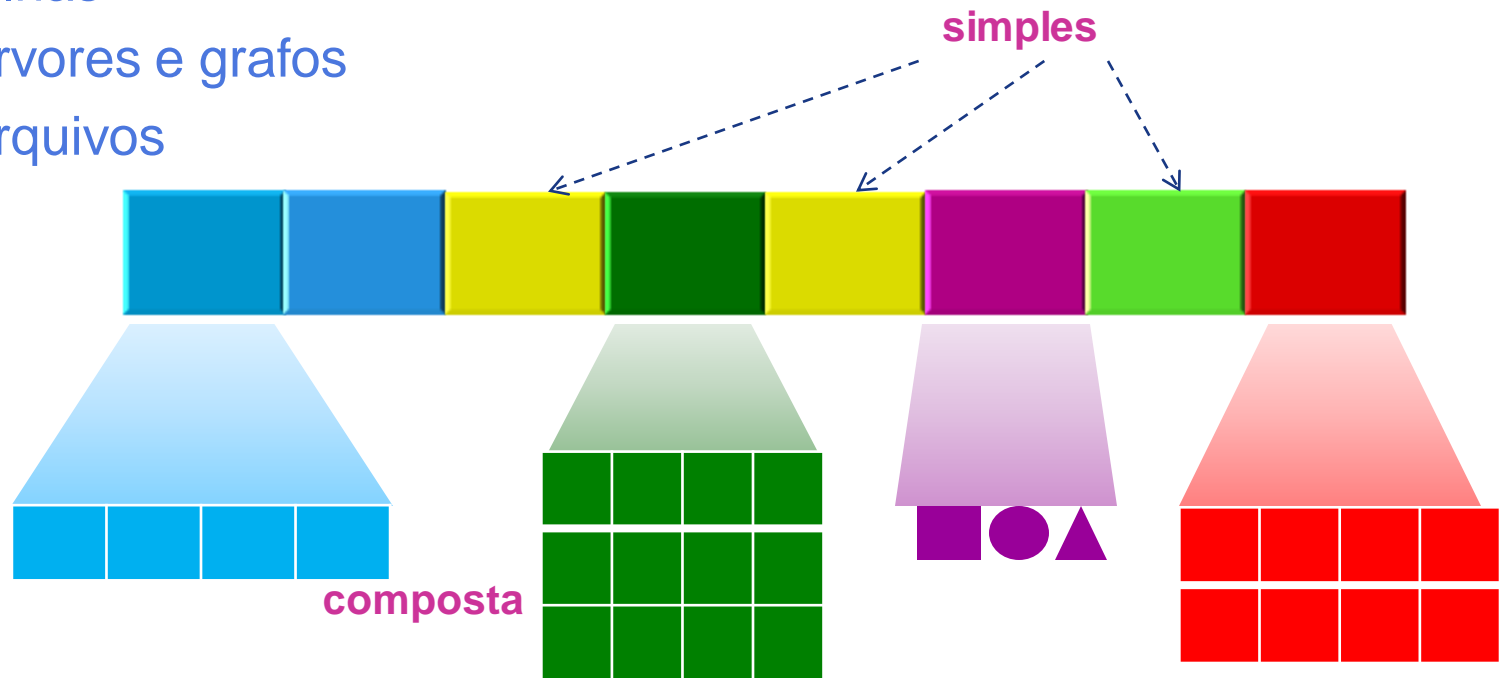
Linguagem Scheme

Estruturas de Dados

❖ Simples e compostas (primitivas e estruturadas)

❖ Principais

- Primitivas: int, float, string
- Vetores e matrizes (arrays)
- Listas
- Pilhas
- Árvores e grafos
- Arquivos



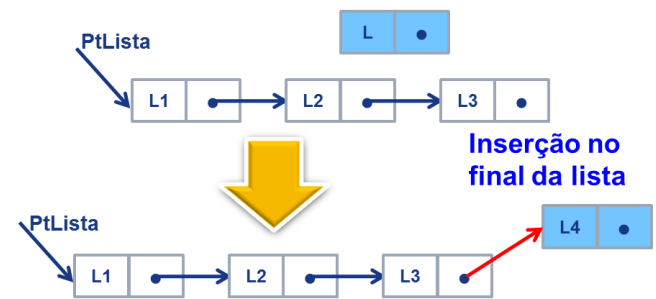
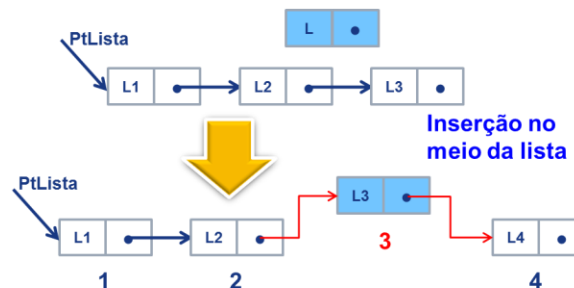
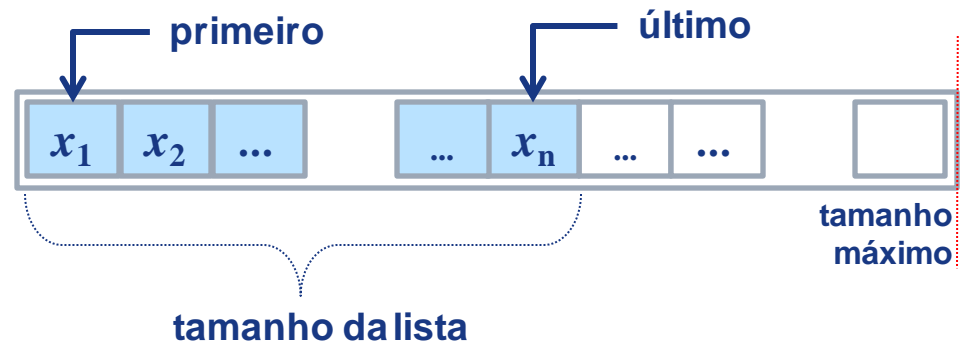
Listas

❖ Estrutura de dados indexada (ordenada)

- FIFO: First Input, First Output

❖ Operações:

- Criar lista
- Primeiro elemento
- Resto da lista
- Comprimento da lista
- Inserir elemento
- Consultar lista
- Testar lista (vazia, cheia)



Listas

Lista = []

Comprimento = 6

Lista = [1 3 5 7 8 9]

primeiro

resto

```
(define lista (list 2 5 9 0 4) )  
(car lista)  
(cdr lista)  
(length lista)  
(append lista (list 3 0 1 0 6) )  
(member 7 lista)  
(null? lista)
```

lista = [2 5 9 0 4]

lista = [2 5 9 0 4 3 0 1 0 6]

Correção

- ❖ Um programa é *correto* se ele faz o que deveria fazer (não o que o programador queria fazer)
- ❖ Como provar correção?
 - Definir corretamente o que o programa deveria fazer
 - Especificação correta do programa
 - Usar um modelo matemático (exato, preciso)
 - Semântica da linguagem (axiomático, operacional, denotacional)
 - Mostrar que o programa satisfaz o modelo matemático

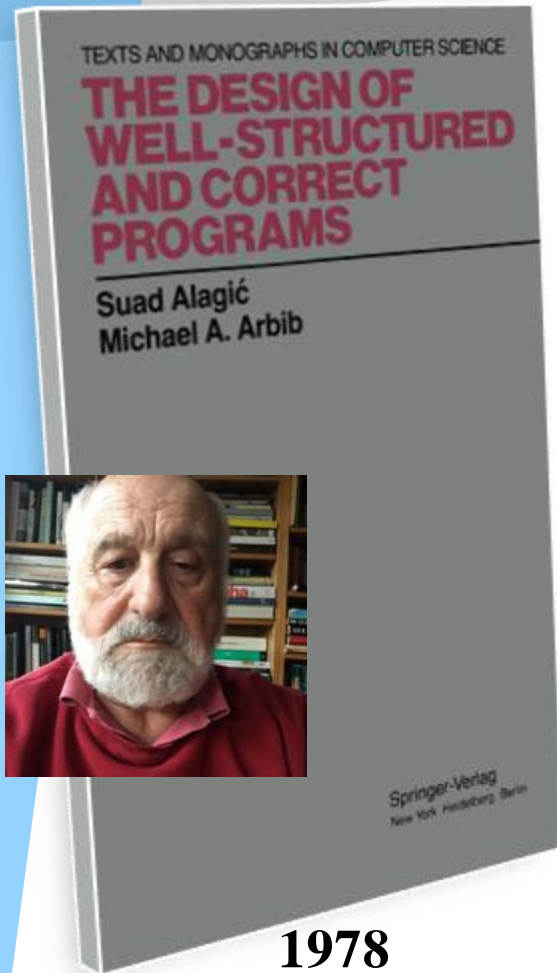
$$x = y + 3 - 5 * y$$

$$x = (y + 3) - 5 * y$$

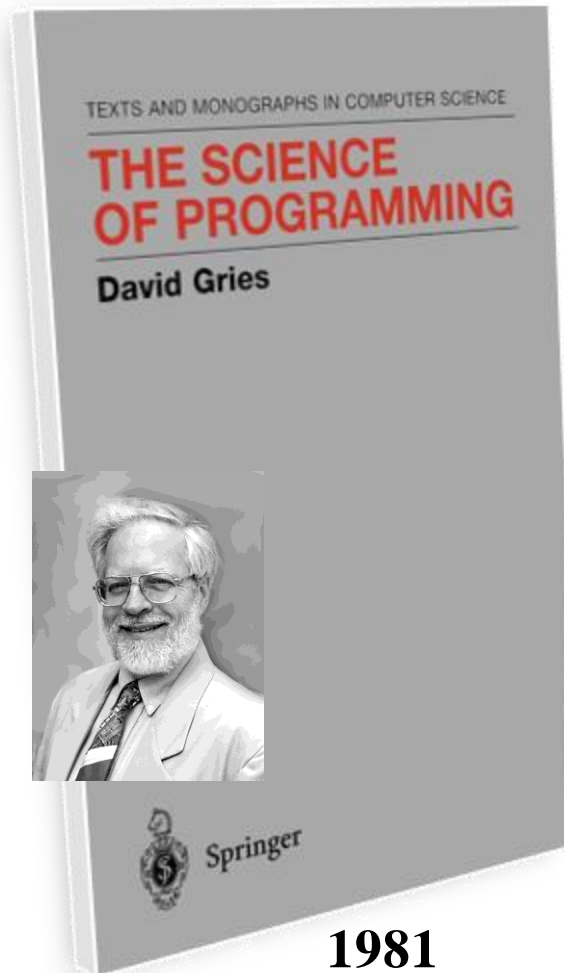
$$x = (y + 3 - 5) * y$$

$$x = y + (3 - 5) * y$$

Correção



1978

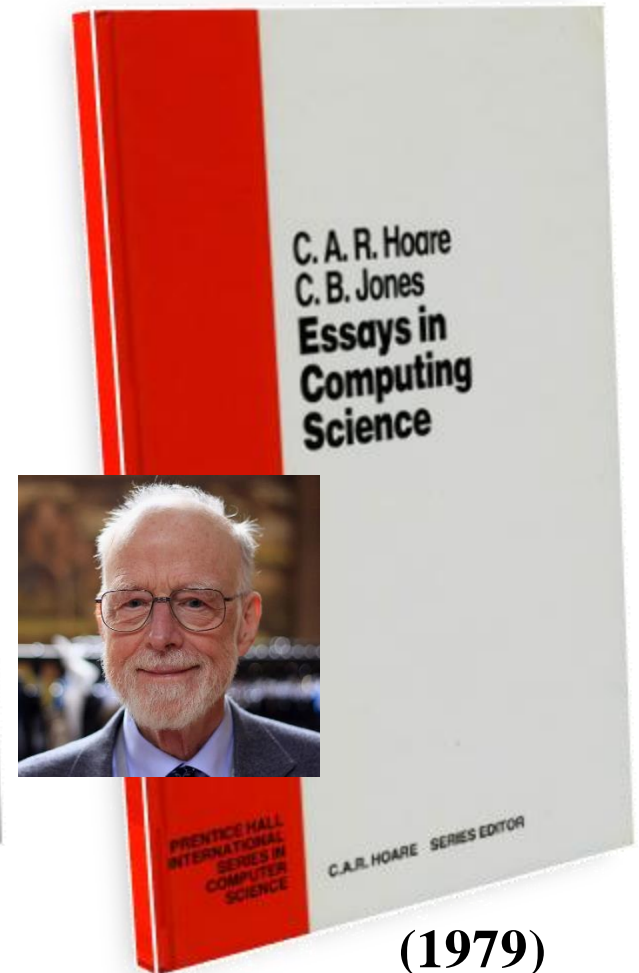


1981

weakest precondition

$Q \Rightarrow wp(S, R)$

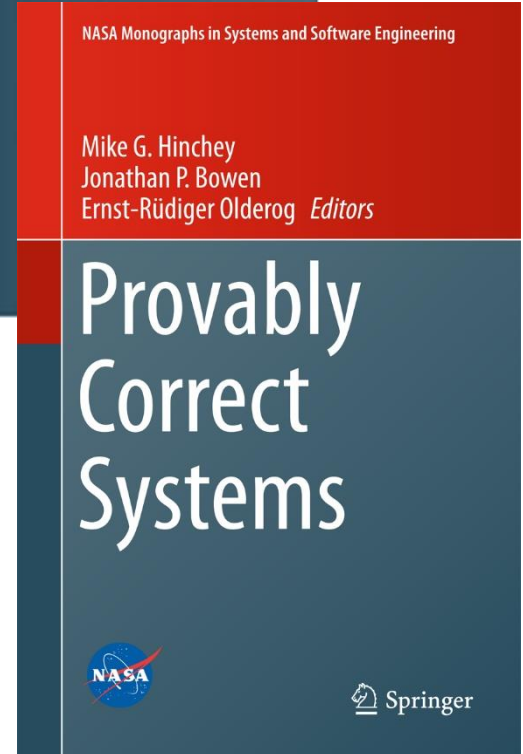
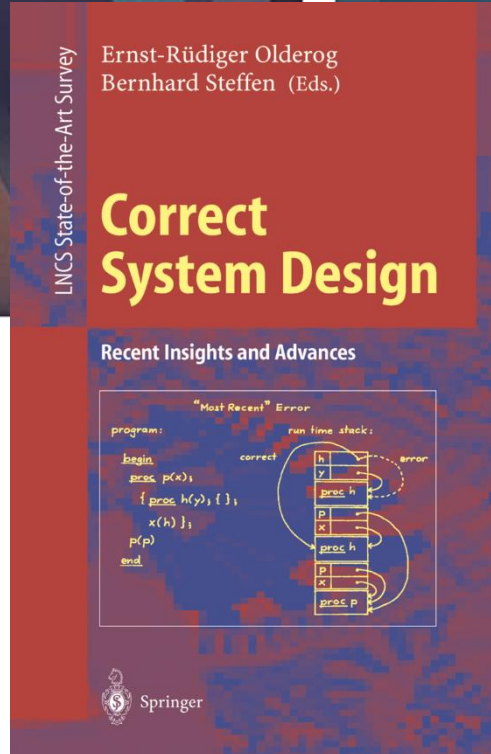
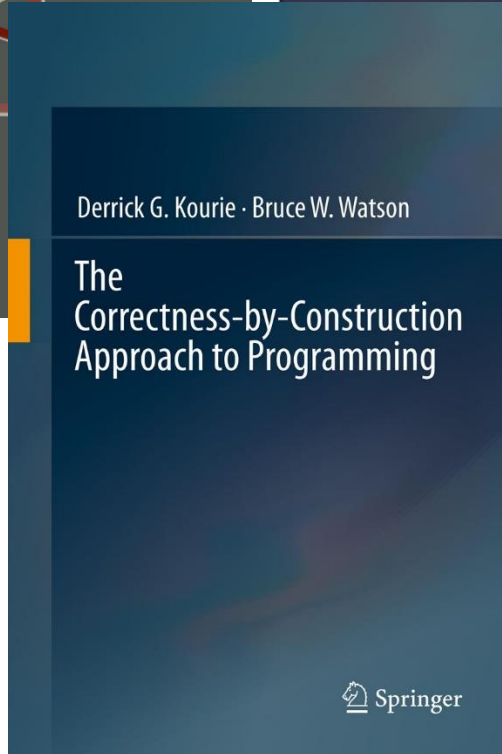
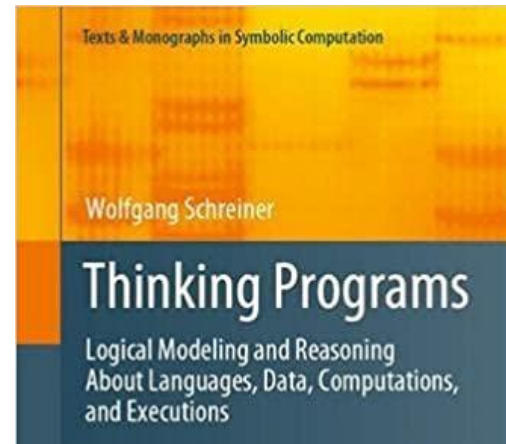
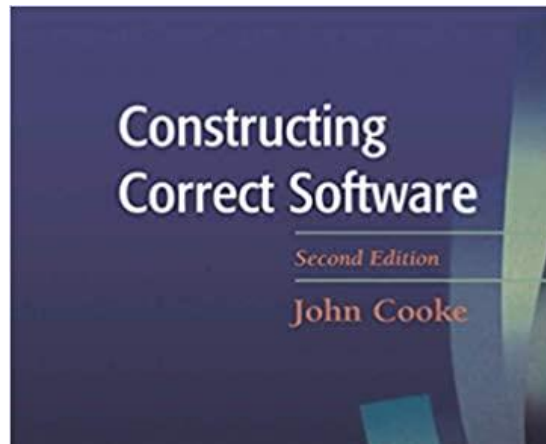
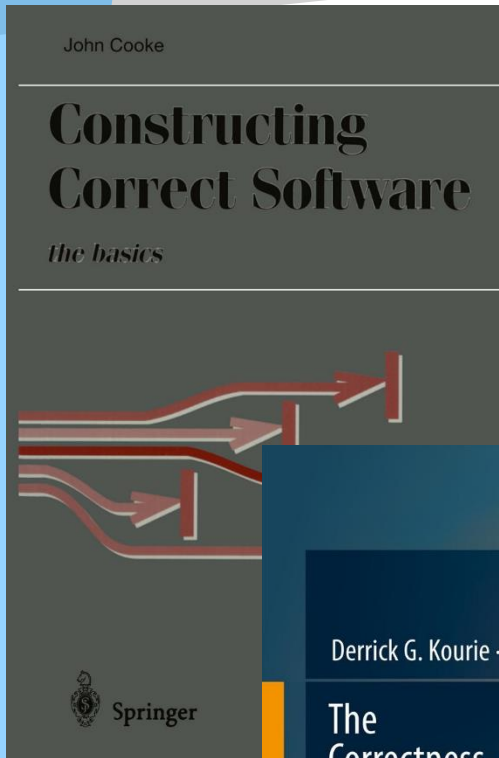
$\{Q\} S \{R\}$



(1979)

1989

Programas corretos



Programas corretos

An Axiomatic Basis for Computer Programming

C. A. R. HOARE
The Queen's University of Belfast, Northern Ireland

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming, proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.50, 5.21, 5.23, 5.24

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice of axioms will to some extent depend on the choice of programming language. For illustrative purposes, this paper is confined to a very simple, but not necessarily representative, subset of all current procedures.

2. Computer Arithmetic

The first requirement in valid programming is to know the properties of the which it invokes, for example, add of integers. Unfortunately, in set arithmetic is not the same as the mathematicians, and it is necessary in selecting an appropriate set of axioms displayed in Table I are of axioms relevant to integers. For

* Department of Computer Science
576 Communications of the A

of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y < r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

$$A5 \quad (r - y) + y \times (1 + q)$$

$$= (r - y) + (y \times 1 + y \times q)$$

$$A9 \quad = (r - y) + (y + y \times q)$$

$$A3 \quad = ((r - y) + y) + y \times q$$

$$A6 \quad = r + y \times q \quad \text{provided } y < r$$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to nonnegative numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which 0, 1, 2, and 3 are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite arithmetic satisfies the axiom:

$$A10 \quad \exists x, y \quad (x < y)$$

vardi's insights



Moshe Y. Vardi

DOI:10.1145/3469113

Program Verification: Vision and Reality

IN 1969, TONY HOARE published a classical *Communications* article, "An Axiomatic Basis for Computer Programming." Hoare's article culminated a sequence of works by Turing, McCarthy, Wirth, Floyd, and Manna, whose essence is an association of a proposition with each point in the program control flow, where the proposition is asserted to hold whenever that point is reached.

Hoare added two important elements to that approach. First, he described a formal logic, now called *Hoare Logic*, for reasoning about programs. Second, he offered a compelling vision for the program-verification project: "When the correctness of a program, its compiler, and the hardware of the computer have all been established with mathematical certainty, it will be possible to place great reliance on the results of the program, and predict their properties with a confidence limited only by the reliability of the electronics."

grams in 1977. Clarke and Emerson, and independently, Quella and Sifakis, then built on Pnueli's work and developed, in the early 1980s, *model checking*, an algorithmic technique for checking properties of finite-state programs. That led to Pnueli receiving the ACM A.M. Turing Award in 1996, and Clarke-Emerson-Sifakis receiving the award in 2007. By the mid-1990s, several model checkers had been built and adopted for industrial use by semiconductor and design-automation companies. Industrial temporal logics, such as PSL and SVA, based on Pnueli's work, became industry standards in the early 2000s.

The success of model checking in the semiconductor industry, where post-production error correction is very difficult, points to an important insight that was missing in the early literature on program verification. Program verification is an expensive activity. Navigating the cost-benefit trade-off of program verification is ultimately a business decision.

Isabelle (and similar tools) this social process can be confined to the small logic core. In fact, with the help of proof assistants, formal verification today is even bringing a new standard for rigor in mathematics.

The emergence of cloud computing as the major context for much of today's computing shifts the cost-benefit trade-off of verification, due to its large scale. Because different users of the same cloud platform share hardware resources, security and privacy are of paramount interest. The Automated-Reasoning Group at Amazon Web Service (AWS) has been focusing on the development and use of formal-verification tools at AWS to increase the security assurance of its cloud infrastructure and to help customers secure themselves. At the same time, as the Spectre and Meltdown attacks have demonstrated, the large gap between the logical model (ISA) and the underlying microarchitecture of the X86 microprocessor not only provides side chan-

Em 1969, Hoare escreveu sobre certeza matemática, grande confiança e confiança. Em retrospecto, a esperança de "certeza matemática" foi idealizada, e não totalmente realista, acredito. A verificação pode nos dar grande confiança e segurança, mas a um custo que deve ser justificado pelos benefícios. A implantação de sistemas autônomos com componentes baseados em aprendizado de máquina traz uma nova urgência e entusiasmo para esta importante área de pesquisa.

Moshe Y. Vardi
Rice University, Houston

Complexidade

- ❖ Quantidade de “trabalho” necessário para executar um programa: recursos computacionais utilizados
- ❖ Depende do:
 - Tipo de operações envolvidas
 - Tipos e quantidades de dados envolvidos
- ❖ Tipos de complexidades
 - Temporal
 - tempo necessário para a execução
 - Espacial
 - Quantidade de memória necessária (alocada)

Exercício - Programa em C:
Determinar o Máximo de três números A, B, C

Estado de um programa

❖ Estado

- É uma seqüência de valores, em um determinado tempo, que contém resultados intermediários de uma computação desejada

❖ Tipos de estado

- Implícito (declarativo)
 - Existe na mente do programador
 - Programação recursiva
- Explícito
 - Usado com chamadas de função
 - Utiliza células: lugar onde pode-se colocar conteúdo
 - Células que tem nome, tempo de vida indefinido, conteúdo, que pode ser mudado. Criar, escrever, ler (consultar) conteúdo



Estado de um programa

Executando ...

E1			E2			E3		
1105	3.2	X	1105	11.7	X	1105	3.2	X
1106	128	Y	1106	128	Y	1106	128	Y
1107	16	K	1107	24	K	1107	40	K
1108	32	Idade	1108	32	Idade	1108	32	Idade
1109	25	t2	1109	60	t2	1109	85	t2
1110			1110			1110		

X = 11.7
K = 24
t2 = 60

K = 40
t2 = 85

Estados



Concorrência e Paralelismo

❖ **Processo seqüencial**

- Conjunto totalmente ordenado de eventos, onde cada um muda de estado em um sistema computacional

❖ **Programa seqüencial**

- Texto (programa) que especifica as possíveis mudanças de estado de um processo seqüencial

❖ **Programa concorrente**

- Programa que especifica as possíveis mudanças de estado de dois ou mais processos seqüenciais

❖ **Concorrência**

❖ **Paralelismo**

Concorrência e Paralelismo

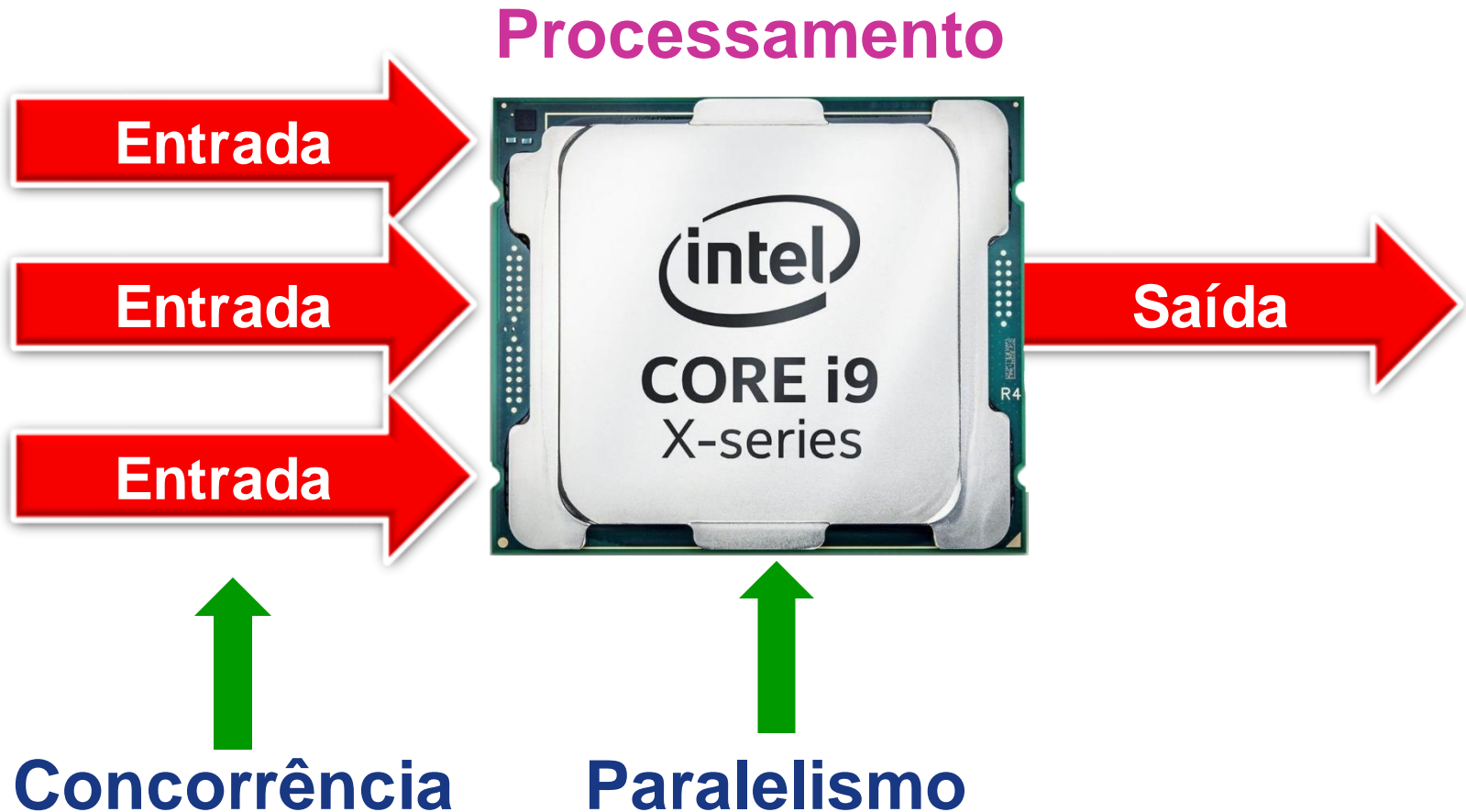
❖ Concorrência

- É propriedade de sistemas computacionais onde vários processos *tentam* ser executados ao mesmo tempo
 - Um ÚNICO processador
- Linguagem JAVA

❖ Paralelismo

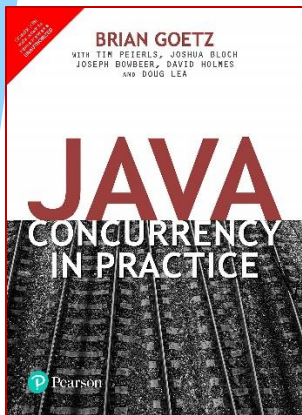
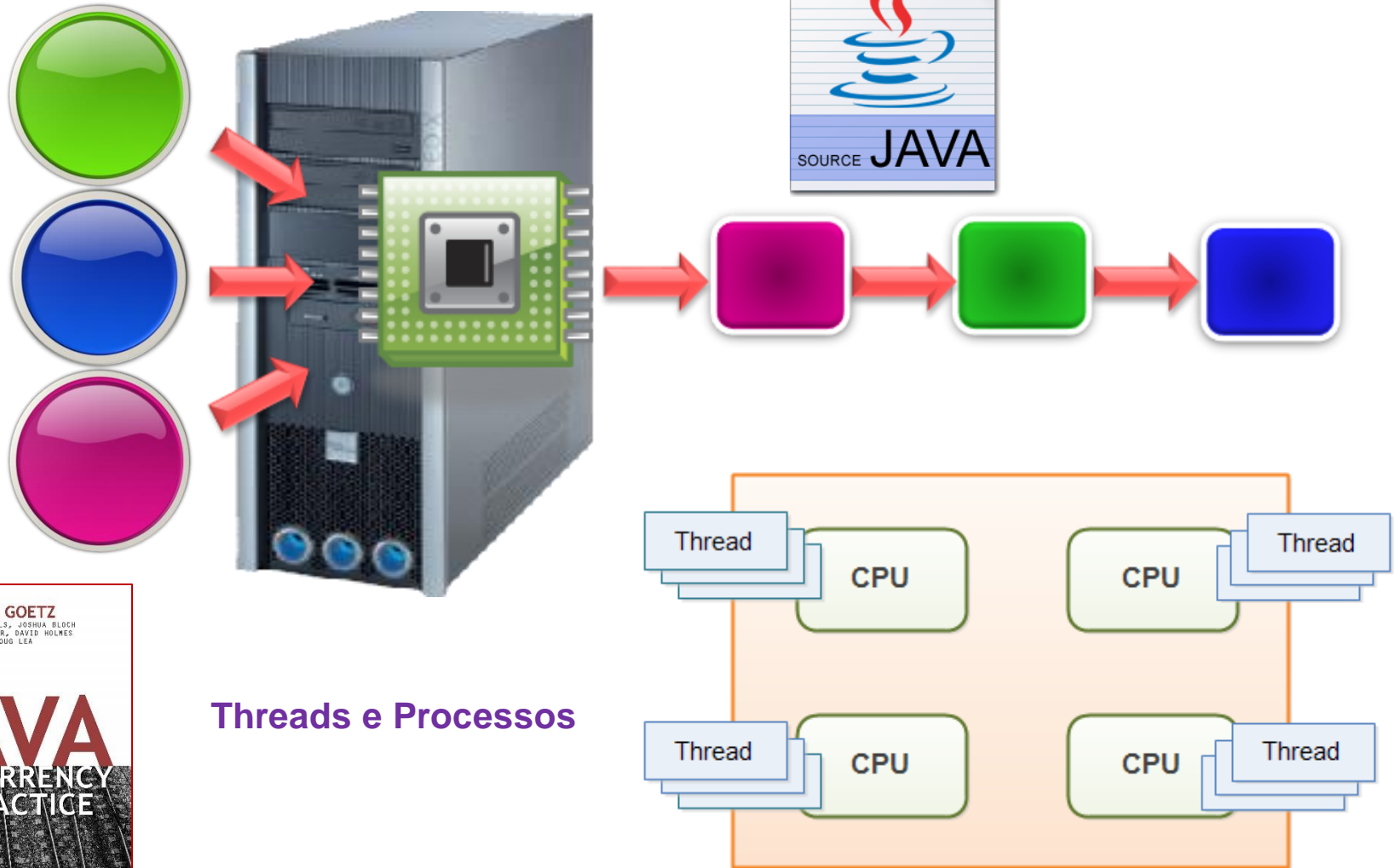
- É propriedade de sistemas computacionais onde vários processos *são executados* ao mesmo tempo
 - MUITOS processadores
- Linguagem FORTRAN 95

Concorrência e Paralelismo



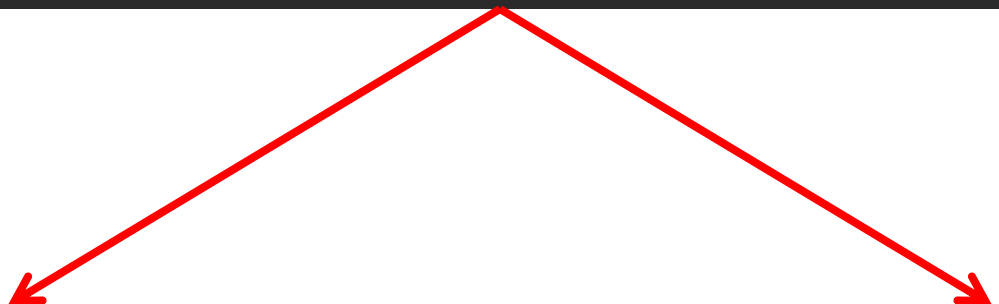
Concorrência e Paralelismo

Concorrência



Concorrência em Java

```
Runnable task = () -> {  
    String threadName = Thread.currentThread().getName();  
    System.out.println("Hello " + threadName);  
};  
  
task.run();  
  
Thread thread = new Thread(task);  
thread.start();  
  
System.out.println("Done!");
```



```
Hello main  
Hello Thread-0  
Done!
```

```
Hello main  
Done!  
Hello Thread-0
```

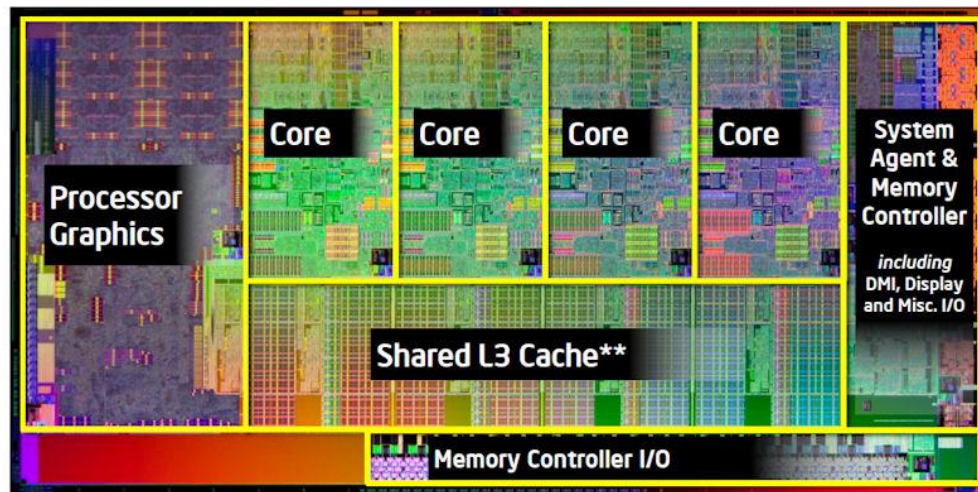
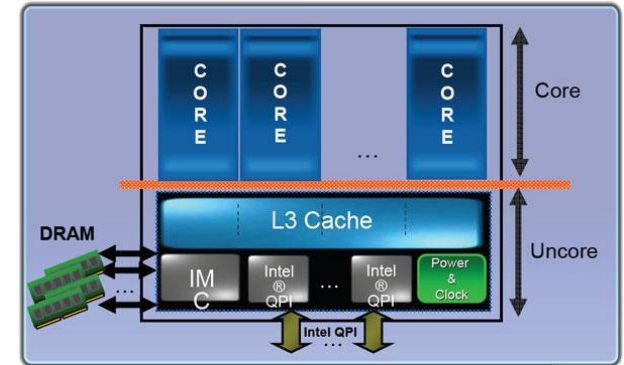
<https://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/>

Concorrência e Paralelismo



Paralelismo

Concorrência e Paralelismo



Concorrência → Paralelismo



Concorrência e Paralelismo

❖ Threads

- Encapsulamento do fluxo de controle em um programa
- Compartilha memória
- Cada uma tem uma prioridade
- A unidade básica de execução de um programa

❖ Problemas

- Não-determinismo: comportamento não reproduzível
- Deadlock (paralisados)
- Dependência de velocidade
- Progresso finito e Inanição (starvation)
 - **inanição** quando um processo nunca é executado ("morre de fome"), pois processos de prioridade maior sempre o impedem de ser executado

Paralelismo em FORTRAN

```
program pi_singlethread
  implicit none
  integer limit
  integer i
  real :: pi=0

  limit = 20000000

  !$OMP PARALLEL    !inicio da paralelizacao do codigo

  !$OMP DO REDUCTION(+:pi) !paralelizar o do
  do i=0, limit
    pi = pi + 4.0 / (4.0*i + 1.0);
    pi = pi - 4.0 / (4.0*i + 3.0);
  enddo
  !$OMP END DO

  !$OMP END PARALLEL

  print*, pi

end
```


Paralelismo em FORTRAN 2018

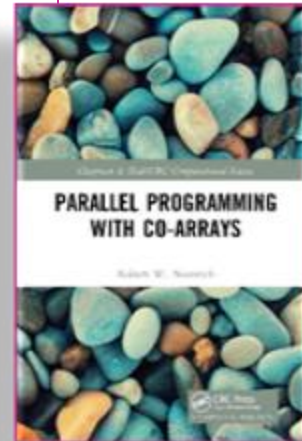
```
program First
  implicit none
  integer, allocatable :: x[:] !----co-array variable--!
  integer :: me,p,y,you       !----normal variables---!

  p = num_images()            !----begin segment 1----!
  me = this_image()           !                        !
  allocate(x[*])               !----end   segment 1----!

  you = me+1                   !----begin segment 2----!
  if(me == p) you = 1          !                        !
  x = me                       !                        !
  sync all                     !----end   segment 2----!

  y = x[you]                   !----begin segment 3----!
  write(*, "('me:  ',i5,' my pal:  ',i5))" me, y !
  deallocate(x)                !----end   segment 3----!

                                !----begin segment 4----!
                                !                        !
                                !                        !
                                !                        !
end program First              !----end   segment 4----!
```



Programação de ordem superior

❖ Higher Order Programming

- Habilidade de usar funções como valores
 - Passar funções como argumentos de outra função
 - Funções podem ser o valor retornado de outras funções
- Utilizado em programação funcional e OO

$g(f)$
 $g(f, h)$

```
(define quadrado  
  (lambda (x) (* x x)  
  )  
)  
  
(define teste-dados '(1 2 3 4 5 6)  
)  
  
( define lista-de-quadrados  
  (map quadrado teste-dados)  
)  
  
(display lista-de-quadrados)
```

← Função quadrado

← Função map

← Função como parâmetro

Abstração

❖ Abstração

- Entidade que incorpora uma computação
- Permite fazer a diferença entre
 - QUE faz uma parte do programa
 - COMO é implementado
- Somente o programador que implementa uma abstração sabe COMO a computação é executada
- Usuários que utilizam a abstração necessitam saber unicamente QUE faz a computação
- Exemplos de abstração:
 - Função e Procedimento

Abstração

❖ Abstração função

- Incorpora uma expressão a ser avaliada. Um valor é retornado
 - $Y = \text{quadrado}(x)$, $k = \text{quadrado}(23)$
 - $X = \text{sqrt}(2x + y)$, $t = \cos(2\pi x + 45)$
- O usuário da função observa somente o resultado e não as etapas de avaliação

❖ Abstração procedimento

- Incorpora um comando e quando chamado, atualiza variáveis
 - $\text{AtualizaCadastro}(c: \text{cadastro})$
 - $\text{OrdenaLista}(L: \text{lista})$
- O usuário do procedimento observa somente as variáveis atualizadas e não as etapas de atualização

Subprogramas

- ❖ **São blocos (trechos) de programa que realizam uma tarefa específica**
 - Facilitam a solução de um problema complexo através de soluções de partes do problema
 - Facilitam a programação estruturada



Cap.9 Subprogramas
Págs. 364-414

Parâmetros

- ❖ **São meios de comunicação entre um programa e um subprograma**
 - Parâmetros formais
 - São os argumentos na *definição* do procedimento
 - Parâmetros atuais (reais)
 - São os argumentos na *chamada* do procedimento
- ❖ **Na comunicação:**
 - Os parâmetros atuais são “substituídos” ou “utilizados” em lugar dos parâmetros formais

Parâmetros: reais e formais

```
procedimento nome ( F1, F2, ..., Fn)
    <ações>           // conjunto de tarefas
fim_procedimento
```

Parâmetros
formais

Na definição!

```
Programa_principal
```

```
...
```

```
<ações>
```

```
[chamar_a] nome(A1, A2, ..., An)
```

```
<ações>
```

```
...
```

```
fim_programa_principal
```

Parâmetros
atuais (reais)

Na chamada!

Parâmetros: reais e formais

```
Function MinhaFuncao(x: integer; y: real): real;  
begin  
    .....  
    .....  
    .....  
end;
```

Parâmetros
Formais: x, y

```
Program P;  
VAR  A,B : real;  
BEGIN  
    .....  
    A := MinhaFuncao(12, 17.6);  
    .....  
    B := MinhaFuncao(70, 328.75);  
    .....  
    .....  
END.
```

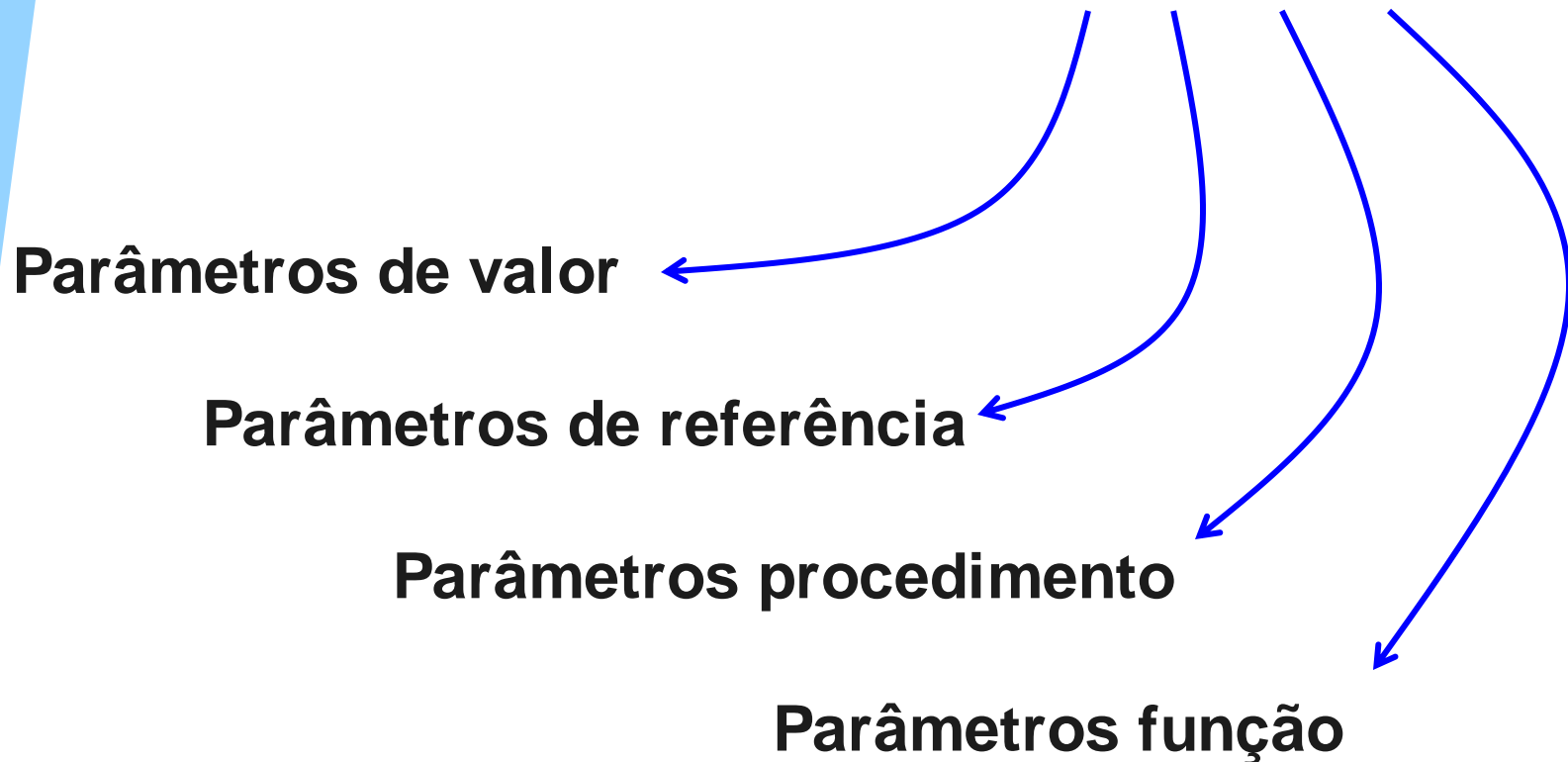
Parâmetros
atuais (reais)

Definição da função

Programa principal

Parâmetros Formais

```
Procedure NomeProcedim( parâmetros formais );
```



Exercício: Pesquisar as 4 definições!

Parâmetros de Valor

- ❖ Chamados *parâmetros de entrada*
- ❖ Envolve *transferência de valor*
 - Transferência de informação entre o parâmetro real e o parâmetro formal
 - São constantes

```
Function MinhaFuncao(x: integer; y:real): real;  
begin  
    .....  
    .....  
    .....  
end;
```

```
Program P;  
VAR  A,B : real;  
BEGIN  
    .....  
A := MinhaFuncao(12, 17.6);  
    .....  
B := MinhaFuncao(70, 328.75);  
    .....  
    .....  
END.
```

12 → x

17.6 → y

x e y são parâmetros de entrada !

Parâmetros de Referência

- ❖ Envolve transferência de endereço (memória) da variável
- ❖ O valor do parâmetro real é *modificado* pela função ou procedimento
- ❖ Agregar a palavra **VAR** na definição

```
Program P;  
VAR  A,B, R : real;  
      K : integer;  
BEGIN  
  .....  
  K := 20;  
  R := 10.8;  
  A := MinhaFuncao(K, R);  
  writeln(K);  
  writeln(R);  
  .....  
END.
```

```
Function MinhaFuncao(x: integer; VAR y: real): real;  
begin  
  .....  
  x := 23;  
  y := 144.5;  
end;
```

20

144.5



K

R

Depois da
execução

20

144.5

23

K

R=y

x

Conceitos numa Linguagem

1. Breve história da linguagem de programação
2. Cinco características básicas da linguagem
3. Como são implementados os valores e tipos?
4. Como são declaradas as variáveis? Regras?
5. Implementação de recursividade
6. Como se implementam listas?
7. Complexidade espacial (memória)
8. Implementa concorrência e paralelismo?
9. Utiliza programação de ordem superior?
10. Tipos de abstração implementados



Prof. Dr. Ausberto S. Castro Vera
Ciência da Computação
UENF-CCT-LCMAT
Campos, RJ

ascv@uenf.br
ausberto.castro@gmail.com

