

# Algoritmia Básica: Construção e Representação de Algoritmos

1ª edição: Novembro/2001

RICARDO J. MACHADO

Email: [rmac@dsi.uminho.pt](mailto:rmac@dsi.uminho.pt)  
URL: <http://www.dsi.uminho.pt/~rmac>



Universidade do Minho

Departamento de Sistemas de Informação

## Sumário

- 1. Construção de Algoritmos**
- 2. Representação de Algoritmos**

# 1. Construção de Algoritmos (1/50)

- génese -

© RMAC XI-2001

## ■ Origem da noção de algoritmo

- a noção de algoritmo é essencial na programação de sistemas informáticos
- a palavra "algoritmo" tem origem no nome do matemático árabe/persa (Mohammed *al-Khowârizmî*) do século IX que definiu as regras passo-a-passo para as operações básicas da aritmética
- em Latin, o nome transformou-se em *Algorismus*, de onde "algoritmo" derivou directamente
- historicamente, o primeiro algoritmo não trivial foi inventado (entre 400 e 300 a.c.) pelo matemático grego Euclides
- esse primeiro algoritmo é o da determinação do máximo divisor comum de dois números inteiros e é designado por "algoritmo de Euclides"

3

# 1. Construção de Algoritmos (2/50)

- o algoritmo -

© RMAC XI-2001

## ■ Noção de algoritmo

- na ciência da computação, um algoritmo pode ser considerado  
*uma sequência não ambígua de instruções elementares conducentes à solução de um dado problema*  
ou  
*um conjunto de instruções que pode ser executado mecanicamente numa quantidade finita de tempo e que resolve algum problema*
- um algoritmo pode também referir um processo resolvente de um problema em qualquer outro domínio distinto da computação

4

# 1. Construção de Algoritmos (3/50)

- o exemplo da culinária -

## ■ Exemplo #1

- imagine-se uma cozinha onde se encontra
  - ingredientes alimentares (farinha, ovos, açúcar, chocolate, ...)
  - utensílios de culinária (panelas, talheres, formas, ...)
  - aparelhos de cozinha (forno, micro-ondas, batedeira, ...)
  - um cozinheiro (ser humano)
- a confeitaria é um processo que
  - *produz* um bolo
  - *a partir de* ingredientes
  - *pelo* cozinheiro
  - *com o auxílio dos* utensílios e dos aparelhos
  - *de acordo com* uma receita

© RMAC XI-2001

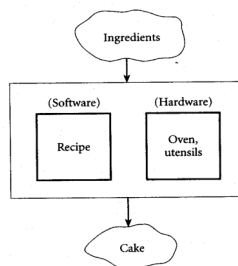
5

# 1. Construção de Algoritmos (4/50)

- o exemplo da culinária -

## ■ Analogia com a área da informática

- os ingredientes constituem a entrada (*input*) do processo, o bolo consiste na saída (*output*) do processo e a receita corresponde ao *algoritmo*
- as receitas estão incorporadas no *software* e os utensílios e aparelhos representam o *hardware*



© RMAC XI-2001

6

# 1. Construção de Algoritmos (5/50)

- o exemplo da culinária -

© RMAC XI-2001

## ■ A receita e as operações finitas

- tal como os computadores que só executam operações binárias, também o cozinheiro com o auxílio dos utensílios e aparelhos está limitado a realizar um conjunto finito e bem definido de operações, tais como misturar, bater, aquecer, cozer, ferver, provar, abrir o forno, fechar o forno, medir o tempo, medir quantidades ...
- o segredo do sucesso está na receita (o algoritmo que *transforma* ingredientes em saborosos bolos) e não no forno ou no cozinheiro
- claro que
  - os utensílios têm que ser adequados, pois é difícil "caçar um leão com uma fisga" ou "matar mosquitos com uma caçadeira"
  - o cozinheiro tem que estar mais ou menos dentro do assunto, caso contrário nem sequer percebe a terminologia da área ("claras em castelo", "banho-Maria", ...)

7

# 1. Construção de Algoritmos (6/50)

- o exemplo da culinária -

© RMAC XI-2001

## ■ A receita da mousse de chocolate\*

- cozinhado: *6 taças de mousse de chocolate*
- ingredientes: *uma tablete de chocolate, 2 colheres de sopa com água, 1/4 de uma chávena de pequeno almoço com açúcar, 6 ovos, ...*
- preparado: *Derreter em lume brando o chocolate juntamente com as duas colheres de água. Depois de derretido, misturar 2 colheres de sopa de açúcar e adicionar a manteiga aos poucos de cada vez. Deixar tudo ao lume. Bater as 6 gemas de ovos durante 5 minutos. Juntá-las lentamente ao chocolate. Mexer devagar se o chocolate ainda não estiver todo derretido. Misturar rum e baunilha. Bater as 6 claras dos ovos em castelo. Antes que as claras fiquem em castelo, juntar 2 colheres de sopa de açúcar. Juntar as claras ao chocolate e misturar bem. Deitar a mousse nas taças individuais. Levar ao frigorífico durante 4 horas. Servir com chantilly, se desejado.*

\* Sinclair, Malinowski (1978), *French Cooking*. Weathervane Books, p. 73

8

# 1. Construção de Algoritmos (7/50)

- o exemplo da culinária -

© RMAC XI-2001

## ■ Instruções básicas ou não

- uma das instruções básicas presente no texto da receita é

*"misturar 2 colheres de sopa de açúcar"*

- porque é que a receita não diz

*"medir 1/2 colher de sopa de açúcar, deitá-lo no chocolate derretido, misturar bem, medir mais 1/2 colher de sopa de açúcar, deitá-lo no chocolate derretido, misturar bem, ..." ?*

- ou mais pormenorizadamente ainda

*"medir 20,75 gramas de açúcar, deitá-lo no chocolate derretido, usar uma colher para realizar movimentos circulares ao misturar o preparado, ..." ?*

- ou ainda

*"... deslocar o braço em direcção ao preparado, num ângulo de 14º, a uma velocidade de 400mm/s ..." ?*

9

# 1. Construção de Algoritmos (8/50)

- o exemplo da culinária -

© RMAC XI-2001

## ■ Instruções básicas ou não (cont.)

- a resposta está no hardware

- neste caso, o hardware (cozinheiro + utensílios + aparelhos) sabe como *"misturar 2 colheres de sopa de açúcar"* e não precisa de mais pormenores

- mas será que sabe *"preparar uma mistura de chocolate com açúcar e manteiga"*?

- se sim, a receita reduzir-se-ia a uma única instrução que diria simplesmente *"preparar mistura de chocolate"* ou mesmo *"preparar mousse de chocolate"*

- esta receita seria ideal, uma vez que é sintética, clara e precisa, não contém erros e garante (ou talvez não!) a produção do *output* tal e qual como o desejado

10

# 1. Construção de Algoritmos (9/50)

## ■ Nível de detalhe das instruções básicas

- obviamente que o nível de detalhe é uma assunto de primordial importância quando se pretende decidir que instruções básica cada passo do algoritmo deve conter
- as acções que cada passo do algoritmo refere que devem ser realizadas **TÊM** que estar de acordo com as capacidades do hardware que vai executar as instruções
- adicionalmente, as acções devem ser entendíveis pelos humanos, uma vez que
  - os algoritmos são escritos por humanos
  - os humanos têm que estar convencidos de que os algoritmos resolvem correctamente os problemas para os quais foram construídos
  - os humanos podem ter que vir a modificá-los no futuro

© RMAC XI-2001

11

# 1. Construção de Algoritmos (10/50)

- o exemplo do produto de dois números -

## ■ Exemplo #2

- o cálculo manual do produto de dois números inteiros é um problema algorítmico
- se se pretender multiplicar 528 por 46, deve-se
  - multiplicar 6 (unidades de 46) por 8 (unidades de 528), resultando 48
  - escrever as unidades do resultado anterior obtido (8)
  - memorizar as dezenas do resultado anterior obtido (4)
  - multiplicar 6 (unidades de 46) por 2 (dezenas de 528), resultando 12
  - adicionar ao resultado anterior (12) as dezenas obtidas na primeira multiplicação (4), resultando 16
  - escrever as unidades do resultado anterior obtido (6)
  - memorizar as dezenas do resultado anterior obtido (1)
  - multiplicar 6 (unidades de 46) por 5 (centenas de 528), resultando 30
  - adicionar ao resultado anterior (30) as dezenas obtidas na primeira multiplicação (1), resultando 31
  - ...

© RMAC XI-2001

12

# 1. Construção de Algoritmos (11/50)

- o exemplo do produto de dois números -

## ■ Exemplo #2

- porquê escrever no algoritmo "*multiplicar 6 ... por 8 ...*"?
- porque não escrever "*adicionar 8 vezes o número 6*"?
- porque não resolver o problema de uma só vez escrevendo "*multiplicar 528 por 46*"?
- porque é permitido conceber uma instrução básica como sendo "*multiplicar 6 ... por 8 ...*" e a instrução "*multiplicar 528 por 46*" já não é considerada básica e como tal não permitida no algoritmo?
- porque se assume que o hardware (o autómato que vai executar o algoritmo), neste caso um humano porque se disse que seria manual, é capaz de executar 6x8 directamente, mas 528x46 já não
- ou seja, 528x46 deve ser refinado em instruções mais simples

© RMAC XI-2001

13

# 1. Construção de Algoritmos (12/50)

- o exemplo da lista telefónica -

## ■ Exemplo #3

- a tarefa de encontrar um nome numa lista telefónica pode ser encarada como um problema com um objectivo concreto que, por sua vez, pode ser visto como uma sequência de sub-objectivos que podem ser atingidos através do cumprimento de determinados passos de resolução
- de uma forma *ad-hoc* e seguindo uma abordagem empírica (experiência do quotidiano) parece razoável começar por subdividir a resolução do problema em dois passos
  1. encontrar a página da lista que contém o último apelido do nome
  2. encontrar, na página determinada no passo 1, o nome procurado
- é importante observar a forma imperativa como é habitual descrever as instruções contidas nos passos de um algoritmo

© RMAC XI-2001

14

# 1. Construção de Algoritmos (13/50)

## ■ Fluxo de um algoritmo

- a execução de um algoritmo segue um fluxo por defeito que consiste na sequência pelos quais os passos aparecem ordenados
- se nada for dito em contrário na descrição de um determinado passo, o passo que lhe segue é o que está colocado imediatamente a seguir na lista de passos que compõem o algoritmo
- para alterar, num determinado passo, o fluxo principal definido pela ordenação dos passos, deve ser dada uma instrução explícita na descrição desse passo no sentido de indicar em que condições e para que passo deve “saltar” a execução do fluxo do algoritmo

# 1. Construção de Algoritmos (14/50)

- o exemplo da lista telefónica -

## ■ Refinamento do passo 1

1. encontrar a página da lista que contém o último apelido do nome
  - 1.1. colocar o dedo (marcador  $d$ ) ao acaso na lista
  - 1.2. abrir a lista na folha apontada pelo marcador  $d$  (o dedo)
  - 1.3. o último apelido do nome está contido numa das páginas (esquerda ou direita) em que a lista foi aberta? se sim, ir para o passo 2
  - 1.4. o último apelido do nome precede a página esquerda? se sim, colocar o marcador  $d$  atrás (antes) da página esquerda, senão colocar o marcador  $d$  à frente (depois) da página direita
  - 1.5. ir para o passo 1.2



# 1. Construção de Algoritmos (15/50)

- o exemplo da lista telefónica -

© RMAC XI-2001

## ■ Refinamento do passo 2

2. encontrar, na página determinada no passo 1, o nome procurado

2.1. delimitar, na página encontrada no passo 1, a região de busca do último apelido

2.2. procurar, dentro da região de busca delimitada, o nome completo

- no refinamento do passo 2, assumiu-se que não existem nomes com o mesmo apelido localizados em páginas diferentes da lista, o que é uma suposição pouco expedita, uma vez que, na prática, não é garantido que tal aconteça sempre (acontece muito raramente!)
- tal como já referido anteriormente, a tarefa de detalhar (refinar ou reificar) cada um dos passos consiste na obtenção de descrições compostas por instruções elementares (básicas) e não ambíguas, passíveis de serem executadas automaticamente

17

# 1. Construção de Algoritmos (16/50)

© RMAC XI-2001

## ■ Regras de refinamento

- um algoritmo como este, relativamente bem estruturado e já com algum detalhe funcional, seria muito difícil de ser executado automaticamente, tal e qual como está
- a automatização (computação) do processo (tarefa de encontrar um nome numa lista telefónica) obriga a
  - eliminar as suposições não plausíveis, ou seja, retractar a realidade da forma mais fidedigna possível
  - utilizar instruções executáveis por um autómato (máquina artificial que funciona segundo as regras de um determinado modelo de computação)

18

# 1. Construção de Algoritmos (17/50)

- o exemplo da lista telefónica -

## ■ Refinamento do passo 1

- no caso do passo 1, dever-se-ia, por exemplo
  - conceptualizar um índice da posição corrente da página na lista
  - 1.1.1. escolher uma posição  $n$  ao acaso no intervalo  $[1, N]$ , em que  $N$  é o número total de páginas úteis da lista
  - 1.1.2. tornar o marcador  $d$  igual a  $n$  (atribuir à variável  $d$  o valor  $n$ )
  - eliminar instruções não formais ("mal" formuladas, porque ambíguas), tais como "colocar o marcador atrás"
  - 1.4. o último apelido do nome precede a página esquerda?
    - se sim, tornar  $n$  igual a  $(n + 1) / 2$  (actualização do valor de  $n$ ),
    - senão tornar  $n$  igual a  $(N + n) / 2$
  - 1.5. ir para o passo 1.1.2

© RMAC XI-2001

19

# 1. Construção de Algoritmos (18/50)

- o exemplo da lista telefónica -

## ■ Exemplo #3

- seguindo as recomendações anteriormente expostas, torna-se possível obter uma descrição algorítmica adequada e suficientemente detalhada para transformar um problema com um carácter inerentemente não numérico numa abordagem à sua resolução com um suporte numérico, possibilitando a sua computação
- lembrando a tese de Church-Turing, qualquer problema é resolúvel por um processo algorítmico se e só se for também resolúvel por uma máquina de Turing, pelo que uma função diz-se *computável* se puder ser avaliada numa MT para quaisquer dados válidos com um número finito de passos
- o algoritmo construído para o exemplo #3 poderia designar-se de "Algoritmo da lista\_telefonica"

© RMAC XI-2001

20

# 1. Construção de Algoritmos (19/50)

- o exemplo da cálculo do MDC -

© RMAC XI-2001

## ■ Exemplo #4

- o problema numérico da determinação do máximo divisor comum de dois números inteiros  $m$  e  $n$  diferentes de 0 pode ser resolvido com o seguinte algoritmo
  1. ler  $m$  e  $n$
  2. se  $m < n$ , então tornar  $min = m$ , senão tornar  $min = n$
  3. tornar  $mdc = min$
  4. tornar  $r1 = m \bmod mdc$  e  $r2 = n \bmod mdc$
  5. se  $r1 = 0 \wedge r2 = 0$ , então devolver como resultado o valor de  $mdc$  e terminar, senão tornar  $mdc = mdc - 1$  e ir para o passo 4
- nesta solução, assumiu-se que para o autómato que vai executar uma implementação do algoritmo a instrução " $n \bmod mdc$ " é básica

21

# 1. Construção de Algoritmos (20/50)

- o exemplo da lista de funcionários -

© RMAC XI-2001

## ■ Exemplo #5

- admita-se que é fornecida uma lista com registos de funcionários de uma determinada empresa
- o registo de cada um dos funcionários contém o seu nome e o valor do seu salário, para além de outras informações
- pretende-se calcular a soma dos salários de todos os funcionários da empresa
- um possível algoritmo é o seguinte
  1. criar e iniciar a zero um contador
  2. percorrer a lista de registos e adicionar ao contador o valor do salário de cada um dos funcionários
  3. depois de chegar ao final da lista, devolver ao exterior o valor obtido para o contador

22

# 1. Construção de Algoritmos (21/50)

- o exemplo da lista de funcionários -

## ■ Variável acumuladora/contadora

- de facto, se as acções constantes de cada um dos três passos do algoritmo apresentado forem consideradas instruções elementares, o algoritmo resolve o problema proposto, i.e., calcula a soma dos salários de todos os funcionários
- o contador utilizado para acumular (somar) os valores dos salários é designado de *variável*, uma vez que pode ser visto como um "caixote" inicialmente vazio (contém o valor zero) e cujo conteúdo (valor acumulado) vai mudando ao longo da execução do algoritmo
- ou seja, depois do passo 2 do algoritmo ter sido executado para o
  - primeiro funcionário da lista, a variável contadora apresenta o valor do salário do primeiro funcionário
  - segundo funcionário da lista, a variável contadora apresenta o valor da soma dos salários do primeiro e do segundo funcionários
  - ...

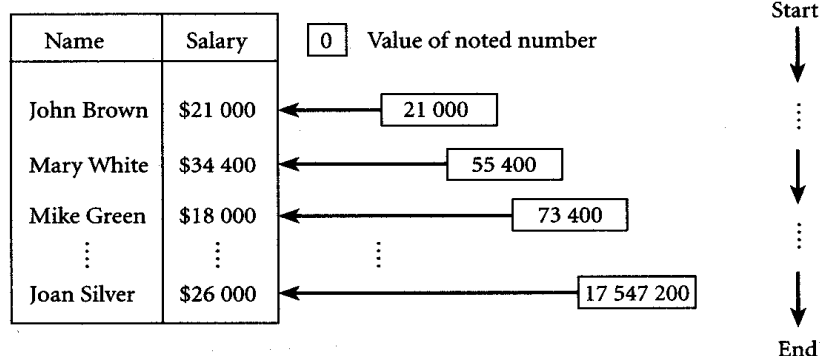
© RMAC XI-2001

23

# 1. Construção de Algoritmos (22/50)

- o exemplo da lista de funcionários -

## ■ Variável acumuladora/contadora (cont.)



© RMAC XI-2001

24

# 1. Construção de Algoritmos (23/50)

- o exemplo da lista de funcionários -

## ■ Algoritmo fixo vs entradas variáveis

- é importante constatar que o texto do algoritmo do exemplo #5 é pequeno e constante no seu tamanho à medida que a lista de funcionários aumenta, se a empresa em causa contratar mais colaboradores
- ou seja, apesar do processo que o algoritmo descreve aumentar linearmente com o tamanho da lista de valores de entrada (registos de funcionários) o algoritmo propriamente dito mantém-se imutável
  - exemplo: das empresas, uma com 10 funcionários e outra com 1 milhão (!), poderão utilizar exactamente o mesmo algoritmo para calcular a soma dos salários dos seus funcionários e exigindo ambas uma única variável contadora, apesar do processo ser muito mais rápido a executar para o primeiro caso do que para o segundo
- esta imutabilidade do algoritmo exige um grau de generalidade elevado para admitir uma variabilidade nas entradas

© RMAC XI-2001

25

# 1. Construção de Algoritmos (24/50)

- requisitos -

## ■ Requisitos do problema algorítmico

- genericamente, o desenvolvimento de software é efectuado para cumprir um determinado caderno de encargos que formaliza os requisitos do cliente
- especificamente, a construção de uma solução para um problema cuja resolução se pretende algorítmica exige a leitura de um documento que, supostamente, descreve o conjunto de requisitos do problema
- só pela compreensão dos requisitos se torna possível apreender a informação necessária para iniciar o processo de construção algorítmica da solução para o problema proposto
- o documento não é mais do que uma representação textual, gráfica, matemática, simbólica ou outra dos requisitos do problema

© RMAC XI-2001

26

# 1. Construção de Algoritmos (25/50)

- requisitos -

© RMAC XI-2001

## ■ Definição #1 (*ad-hoc*)

- qualquer coisa que um cliente pretende
- qualquer coisa que necessita ser concebida

## ■ Definição #2 (norma IEEE 610-93)

- uma condição ou uma capacidade de que alguém necessita para resolver um problema ou atingir um objectivo
- uma condição ou uma capacidade que deve ser verificada ou possuída por um sistema ou por um componente de um sistema para satisfazer um contracto, uma norma, uma especificação, ou outro qualquer documento formalmente imposto
- uma representação documentada de uma condição ou uma capacidade, no âmbito dos dois itens anteriores

27

# 1. Construção de Algoritmos (26/50)

© RMAC XI-2001

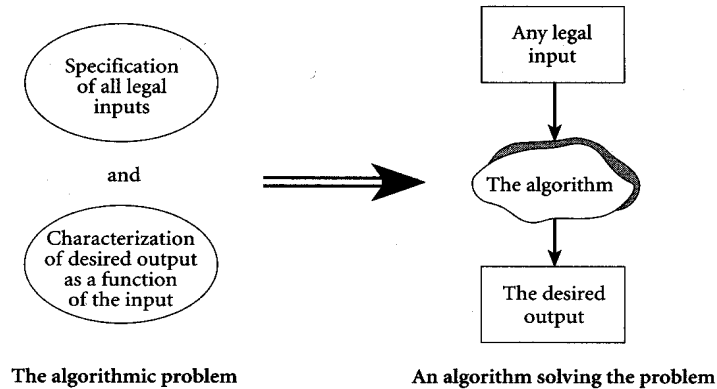
## ■ Problema algorítmico

- um problema algorítmico é o problema para o qual o algoritmo a conceber deve descrever uma resolução
- a descrição de um problema algorítmico (conjunto de requisitos do problema) deve incluir, obrigatoriamente
  - uma definição precisa dos conjuntos de dados de entrada (*input*) válidos, i.e., que o algoritmo deve ser capaz de tratar (sub-conjunto dos  $x \in \Gamma$  do modelo de computação MT)
  - uma caracterização precisa das saídas do processo (*output*) em função das entradas (sub-conjunto dos  $z \in \Gamma$  do modelo de computação MT)

28

# 1. Construção de Algoritmos (27/50)

## ■ Problema algorítmico (cont.)



© RMAC XI-2001

29

# 1. Construção de Algoritmos (28/50)

- exemplos de problemas algorítmicos -

## ■ Problema #1

- **input:** dois números inteiros  $j$  e  $k$
- **output:** o número  $j^2 + 3.k$ 
  - observações:
    - este é um *problema aritmético*
    - este problema exige, unicamente, a realização de alguns cálculos aritméticos sobre dois valores numéricos de entrada
    - este é um problema simples, porque os dados a manipular correspondem exactamente aos valores numéricos recebidos do exterior

© RMAC XI-2001

30

# 1. Construção de Algoritmos (29/50)

- exemplos de problemas algorítmicos -

## ■ Problema #2

- **input:** um número inteiro positivo  $k$
- **output:** a soma de todos os números inteiros pertencentes a  $[1, k]$ 
  - observações:
    - este é um *problema aritmético*
    - este problema é mais complexo do que o problema #1, uma vez que exige a realização de cálculos aritméticos, não exclusivamente sobre valores de entrada, mas também sobre valores calculados a partir dos de entrada
    - exige, portanto, a manipulação de vários elementos numéricos, cuja cardinalidade depende do valor do próprio  $k$

© RMAC XI-2001

31

# 1. Construção de Algoritmos (30/50)

- exemplos de problemas algorítmicos -

## ■ Problema #3

- **input:** um número inteiro positivo  $k$
- **output:** "sim", se  $k$  for primo e "não", em caso contrário
  - observações:
    - este é um *problema de decisão*
    - este problema exige uma resposta relativamente a uma propriedade de um dado de entrada
    - lembrando, um número primo é um número inteiro positivo que pode só ser dividido com resto 0 por ele próprio e pela unidade
    - a resolução deste problema exige o recurso a operações aritméticas, mas o seu **output** é não numérico, uma vez que é uma de duas respostas, ou "sim", ou "não"

© RMAC XI-2001

32



# 1. Construção de Algoritmos (31/50)

- exemplos de problemas algorítmicos -

## ■ Problema #4

- **input:** uma lista *L* de palavras em português
- **output:** a lista *L* ordenada por ordem alfabética crescente
  - observações:
    - este é um *problema de ordenação*
    - este problema exige a utilização de um conjunto de critérios para estabelecer uma sequência ordenada de uma colecção de dados recebidos do exterior
    - tal como o problema #2, este problema também tem que manipular um número variável de elementos (palavras) que depende da lista *L* recebida do exterior

© RMAC XI-2001

33

# 1. Construção de Algoritmos (32/50)

- exemplos de problemas algorítmicos -

## ■ Problema #5

- **input:** dois textos em português
- **output:** uma lista com as palavras comuns aos dois textos
  - observações:
    - este é um *problema de procura*
    - este problema exige a pesquisa de palavras chave em conjuntos de dados não ordenados
    - tal como o problema #4, este problema também exige a manipulação de palavras e não de valores numéricos
    - assume-se que um texto é uma *string* composta por palavras, espaços vazios e símbolos de pontuação e que uma palavra é uma *string* simples delimitada por espaços vazios ou por símbolos de pontuação

© RMAC XI-2001

34

# 1. Construção de Algoritmos (33/50)

- exemplos de problemas algorítmicos -

## ■ Problema #6

- **input:** um mapa com cidades, com distâncias associadas a cada segmento de estrada e com duas das cidades presentes no mapa associadas às letras *A* e *B*
- **output:** a descrição do percurso mais curto entre as cidades *A* e *B*
  - observações:
    - este é um *problema de otimização*
    - este problema exige o recurso a técnicas de pesquisa de várias soluções sobre as quais tem que haver uma decisão que determine a solução que minimiza uma determinada função de custo
    - a função de custo no caso deste problema calcula a distância entre duas cidades separadas por um ou mais segmentos de estrada

© RMAC XI-2001

35

# 1. Construção de Algoritmos (34/50)

- exemplos de problemas algorítmicos -

## ■ Problema #7

- **input:** um mapa com cidades, com distâncias associadas a cada segmento e número inteiro positivo *k*
- **output:** "sim", se for possível definir um percurso que passe por todas as cidades e cuja distância total do percurso não seja superior a *k* e "não", em caso contrário
  - observações:
    - este é um *problema de decisão com procura*
    - este problema exige, tal como o problema #6, a determinação de um caminho mínimo, não entre dois pontos, mas entre todos os pontos (cidades)
    - este problema não exige uma otimização, mas unicamente uma decisão se "sim" ou "não" é possível encontrar um caminho mais pequeno do que um determinado limite pré-estabelecido

© RMAC XI-2001

36

# 1. Construção de Algoritmos (35/50)

- exemplos de problemas algorítmicos -

© RMAC XI-2001

## ■ Problema #8

- **input:** um programa  $P$  escrito na linguagem  $C$ , que lê (*input*) um número inteiro  $x$  e que devolve (*output*) um número inteiro  $y$ , e um número inteiro  $k$
- **output:** o número  $2.k$ , se o programa  $P$  devolver  $y$  sempre com o valor de  $x^2$ , o número  $3.k$ , em caso contrário
  - observações:
    - este é um problema sobre algoritmos
    - este problema exige uma decisão que depende do comportamento observado de um programa  $P$  (um outro algoritmo já implementado numa linguagem de programação executável)

37

# 1. Construção de Algoritmos (36/50)

© RMAC XI-2001

## ■ Solução algorítmica

- um problema algorítmico diz-se *resolvido*, quando for encontrada uma solução algorítmica (algoritmo) *apropriada*
- um algoritmo diz-se *apropriado*, quando tem a capacidade de fornecer/gerar/calcular saídas (*outputs*) correctas para todas e quaisquer entradas (*inputs*) válidas que sejam utilizadas para executar o algoritmo
- um algoritmo que só funcione correctamente para alguns conjuntos de entradas válidas não é um algoritmo suficientemente correcto, porque está *incompleto*
- ou seja, um algoritmo *incompleto* está somente habilitado a resolver o problema algorítmico para alguns casos particulares do problema genérico (aqueles casos particulares que possuem as entradas para as quais o algoritmo é capaz de gerar resultados correctos)

38

# 1. Construção de Algoritmos (37/50)

## ■ Solução para o problema algorítmico #1

- computar  $j^2 + 3.k$  é um problema trivial, desde que a soma e a multiplicação sejam consideradas instruções elementares

## ■ Solução para o problema algorítmico #2

- determinar os números inteiros compreendidos no intervalo  $[1, k]$  é, também, um problema trivial, desde que seja possível recorrer a um contador (variável acumuladora)

## ■ Solução para o problema algorítmico #3

- a determinação se um dado número  $k$  é primo pode ser realizada
  - dividindo-o por todos os números inteiros compreendidos no intervalo  $[2, k - 1]$
  - responder “não”, se, pelo menos, uma das divisões der resto 0
  - responder “sim”, se nenhuma das divisões der resto 0

# 1. Construção de Algoritmos (38/50)

## ■ Outra solução para o problema algorítmico #3

- a solução anterior pode ser melhorada (otimizada relativamente ao tempo de execução), efectuando divisões por todos os números inteiros compreendidos no intervalo  $[2, \sqrt{k}]$ , evitando realizar divisões por múltiplos de números já testados
- *Nota:* qualquer uma das soluções apresentadas aqui podem ser optimizadas recorrendo a técnicas mais avançadas de construção de algoritmos, no entanto, nesta parte, a exemplificação de soluções segue uma abordagem minimal, no sentido em que considera um algoritmo apropriado aquele que se apresenta completo, independentemente da sua eficiência

## ■ Solução para os problemas algorítmicos #4 e #5

- a ordenação e a procura de palavras ser realizada recorrendo a um dos inúmeros algoritmos de ordenação disponíveis na literatura

# 1. Construção de Algoritmos (39/50)

## ■ Solução para os problemas algorítmicos #6 e #7

- os problemas #6 e #7 podem ambos ser resolvidos determinando todos os percursos possíveis entre as cidades e calcular as respectivas distâncias totais, ou seja
  - as distâncias de todos os percursos entre as cidades  $A$  e  $B$
  - as distâncias de todos os percursos que passem por todas as cidades
- uma vez que o número de cidades é finito, o número de percursos também é finito, pelo que se torna possível conceber um algoritmo que execute aquelas acções
- esta abordagem deve ser realizada tendo o cuidado de não “esquecer” nenhum percurso válido, nem de considerar segmentos de percurso desnecessários
- *Nota:* estas duas soluções carecem, obviamente, de uma abordagem optimizada, caso contrário o tempo de execução dos algoritmos atinge facilmente escalas inaceitáveis

© RMAC XI-2001

41

# 1. Construção de Algoritmos (40/50)

- características dos algoritmos -

## ■ Entradas

- um algoritmo tem zero ou mais entradas (habitualmente acessíveis do exterior através de instruções de leitura)
- a quantidade de entradas tem que ser especificada previamente
  - exemplo: o algoritmo MDC tem duas entradas  $m$  e  $n$ , ambas do conjunto dos números inteiros positivos (sem o zero)

## ■ Saídas

- um algoritmo tem uma ou mais saídas (habitualmente disponibilizadas para o exterior através de instruções de escrita)
- no caso de algoritmos cujo objectivo é efectuar uma dada acção (tocar uma campainha, abrir uma porta, ...), consideram-se como saídas os conjuntos de dados que desencadeiam tais acções
  - exemplo: no algoritmo MDC a saída corresponde ao valor de  $mdc$

© RMAC XI-2001

42

# 1. Construção de Algoritmos (41/50)

- características dos algoritmos -

## ■ Finitude

- a execução de um algoritmo deve terminar sempre num número finito de passos
  - exemplo: o algoritmo MDC termina obrigatoriamente no passo 5, uma vez que o valor corrente da variável *mdc* é decrescente e, no pior caso (*m* e *n* primos entre si), obtém-se  $mdc = 1$  e  $r1 = r2 = 0$

## ■ Definibilidade

- todos os passos de um algoritmo devem possuir um significado preciso e não ambíguo
- num algoritmo devem ser rejeitadas formulações do tipo
  - "se *n* for grande" (especificação qualitativa)
  - "escrever qualquer coisa" (domínio indefinido)
  - "para o maior inteiro *n* tal que  $x^n + y^n = z^n$ , com *x*, *y*, *z* inteiros ..." (solução matemática desconhecida)

© RMAC XI-2001

43

# 1. Construção de Algoritmos (42/50)

- características dos algoritmos -

## ■ Eficácia

- os passos de um algoritmo devem conduzir, efectivamente, à solução do problema proposto
- no entanto, por vezes, não basta que o algoritmo seja eficaz, pois levantam-se questões de eficiência (a tratar no final deste capítulo)
- a eficiência depende do método de resolução adoptado
- na ausência de um conhecimento aprofundado sobre métodos resolventes (sub-)óptimos (algoritmos eficientes) dever-se-á recorrer às definições matemáticas (tal como o algoritmo do exemplo #4), ou a heurísticas ("manuais") de resolução
  - exemplo: um algoritmo mais eficiente (menor número de operações realizadas) do que o apresentado para o cálculo de MDC poderia ser o já referido algoritmo de Euclides que reduz sucessivamente o problema do  $mdc(m, n)$  ao problema do  $mdc(n, m \bmod n)$

© RMAC XI-2001

44

# 1. Construção de Algoritmos (43/50)

## ■ Formalização da noção de algoritmo

- um algoritmo é um processo
  - *discreto* (sequência de acções indivisíveis)
  - *determinístico* (para cada passo da sequência e para cada conjunto válido de dados corresponde uma e só uma acção)
  - *finito* (ou seja, o algoritmo deve terminar para quaisquer que sejam os dados iniciais pertencentes aos domínios pré-definidos)
- um algoritmo é estruturalmente constituído por passos, ordenados segundo as regras básicas do fluxo de execução dos algoritmos, sendo cada passo descrito à custa de uma ou mais instruções elementares para o nível de abstracção (refinamento) adoptado

# 1. Construção de Algoritmos (44/50)

- tipos de instruções algorítmicas -

## ■ Leitura e escrita de dados

- corresponde a instruções de entrada (*input*) e a saída (*output*) de dados do algoritmo

## ■ Atribuição de valores

- corresponde a instruções conducentes à actualização de valores de dados, formuladas em linguagem natural na forma "tornar ...", "passar a ser igual ..."
- por vezes, em linguagens algorítmicas baseadas em linguagem natural, utiliza-se a seta ( $\leftarrow$ ) para representar atribuições
- exemplo: no algoritmo da lista telefónica a atribuição "tornar  $n$  igual a  $(N + n) / 2$ " poderia ser representada por " $n \leftarrow (N + n) / 2$ "

# 1. Construção de Algoritmos (45/50)

- tipos de instruções algorítmicas -

## ■ Teste de condições

- corresponde a instruções que verificam se uma dada condição (proposição Booleana simples ou composta) é satisfeita ou não, com base nos valores correntes das variáveis envolvidas na proposição
- tendo em conta a resposta obtida (a condição ou é avaliada como *verdadeira*, ou como *falsa*), uma determinada acção (instrução) deve ser executada
- estas instruções, na sua forma mais simples, são, em linguagem natural, formuladas da seguinte forma:

se *condição* é verdadeira, então executar *acção\_1*,  
senão executar *acção\_2*

ou, de uma forma mais sintética:

se *condição*, então *acção\_1*, senão *acção\_2*

© RMAC XI-2001

47

# 1. Construção de Algoritmos (46/50)

- tipos de instruções algorítmicas -

## ■ Salto

- corresponde a instruções que redireccionam a sequência de execução dos passos do algoritmo, contrariando as regras básicas de fluxo
- em linguagem natural, são, normalmente, formuladas na forma "ir para o passo ..."

## ■ Paragem

- corresponde a instruções que indicam que o fluxo de execução do algoritmo chegou ao fim
- em linguagem natural, são, normalmente, formuladas na forma "terminar"

© RMAC XI-2001

48



# 1. Construção de Algoritmos (47/50)

© RMAC XI-2001

## ■ Meta-algoritmo (algoritmo para construir algoritmos)

1. ler, atentamente, o texto do problema algorítmico
  - 1.1. procurar o significado de todas as palavras e expressões
  - 1.2. compreender clara e completamente o problema descrito
  - 1.3. eliminar detalhes supérfluos
2. especificar simbolicamente os dados do problema algorítmico
  - 2.1. retirar do texto do enunciado, e representar simbolicamente por compreensão, os conjuntos de dados de entrada (*input*)
  - 2.2. retirar do texto do enunciado, e representar simbolicamente por compreensão, os conjuntos de dados de saída (*output*)
  - 2.3. retirar do texto do enunciado, e representar simbolicamente, as relações entre os conjuntos de dados de saída e os de entrada

49

# 1. Construção de Algoritmos (48/50)

© RMAC XI-2001

## ■ Meta-algoritmo (algoritmo para construir algoritmos)

3. determinar que acções devem ser descritas pelo algoritmo, recorrendo aos princípios da engenharia de software
  - 3.1. decidir que problema algorítmico vai ser tratado, tendo em conta os princípios da generalidade e da antecipação à mudança
  - 3.2. identificar as acções a descrever pelo algoritmo, recorrendo aos princípios da desagregação do problema e da modularidade
  - 3.3. especificar o algoritmo, recorrendo ao princípio do rigor e formalismo
  - 3.4. verificar se as acções identificadas no passo 3.2 deram origem a instruções elementares, tendo em conta os princípios da abstracção e da incrementalidade;  
se sim, então ir para o passo 4,  
senão decidir o nível de detalhe a adoptar e ir para o passo 3.2

50

# 1. Construção de Algoritmos (49/50)

© RMAC XI-2001

## ■ Meta-algoritmo (algoritmo para construir algoritmos)

### 3.3. especificar o algoritmo, recorrendo ao princípio do rigor e formalismo

3.3.1. decidir que linguagem (natural, simbólica, de modelação) vai ser utilizada para descrever as acções do algoritmo

3.3.2. agrupar as acções em passos discretos de execução

3.3.3. ordenar os passos segundo um fluxo essencialmente sequencial de execução das acções

3.3.4. identificar quais as violações (saltos) ao fluxo sequencial de execução das acções

3.3.5. descrever, ordenadamente, as acções recorrendo à linguagem adoptada

51

# 1. Construção de Algoritmos (50/50)

© RMAC XI-2001

## ■ Meta-algoritmo (algoritmo para construir algoritmos)

### 4. executar, manualmente, o algoritmo

4.1. seleccionar um conjunto de entradas válidas

4.2. aplicar o conjunto de entradas ao algoritmo resultante da última execução do passo 3.3

4.3. verificar a validade dos dados de saída;  
se dados de saída válidos então terminar,  
senão identificar incorrecção e ir para o passo 1

52

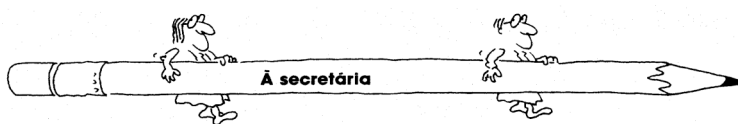


## exercícios

1. Considere as regras de funcionamento e de avaliação especificadas no documento da aula de apresentação da disciplina de Programação Estruturada.
  - a) À luz do modelo de computação das máquinas de Turing, indique
    - a.1) O conjunto  $x$  das entradas do sistema
    - a.2) O conjunto  $z$  das saídas do sistema
  - b) Escreva, em linguagem natural, um algoritmo não refinado para a determinação da classificação final para um aluno
  - c) Refine o algoritmo construído na alínea anterior até a um nível de detalhe em que só existam instruções elementares (ou básicas)
  - d) Estenda o algoritmo construído na alínea anterior, de forma a que sejam determinadas as classificações finais de todos os alunos oficialmente inscritos na disciplina no ano lectivo corrente

© RMAC XI-2001

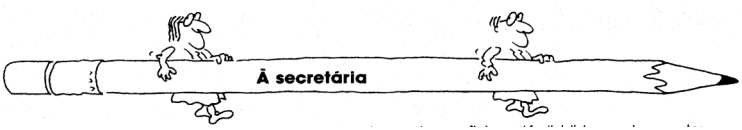
53



1. b)
  1. calcular frequência
  2. calcular avaliação teórica
  3. calcular avaliação prática
  4. calcular classificação à disciplina
  5. calcular classificação final
1. c)
  1. calcular frequência
    - 1.1. calcular  $\#(AC) = \#(ACt) + \#(ACtp) + \#(ACp)$
    - 1.2. se  $\#(AC)/AD \geq 2/3 \wedge \#(ACt)/ADt \geq 2/3 \wedge \#(ACtp)/ADtp \geq 2/3 \wedge \#(ACp)/ADp \geq 2/3$ , então tornar  $freq = TRUE$ , senão tornar  $freq = FALSE$
  2. calcular avaliação teórica
    - 2.1. se  $f1 \geq 8/20 \wedge prescindi\_f2 = FALSE$ , então tornar  $at = (f1 + f2)/2$ , senão tornar  $at = ex$

© RMAC XI-2001

54



1. c) (cont.)

3. calcular avaliação prática

3.1. calcular  $tp1g = 0.6 \times esp1 + 0.4 \times rel1$

3.2. calcular  $tp2g = 0.6 \times esp2 + 0.4 \times rel2$

3.3. calcular  $ap = ((tp1i + tp1g)/2 + (tp2i + tp2g)/2)/2$

4. calcular classificação à disciplina

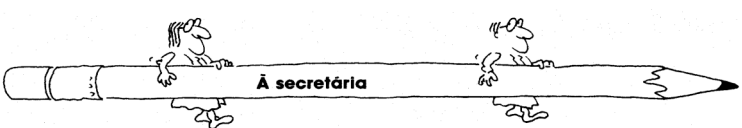
4.1. calcular  $value = 0.45 \times at + 0.5 \times ap + 0.05 \times ac$

4.2. se  $(8/20 > value \geq 9.5/20) \vee (value < 18.5/20)$ ,  
então tornar  $cd = value$  e ir para 5

4.3. se  $(8/20 \leq value < 9.5/20) \vee (value \geq 18.5/20)$ ,  
então tornar  $cd = oral$

© RMAC XI-2001

55



1. c) (cont.)

5. calcular classificação final

5.1. se  $aluno\_ordinario = TRUE \wedge (freq = FALSE \vee \exists tp < 9/20)$ ,  
então tornar  $cf = \text{"não admitido"}$  e terminar

5.2. se  $aluno\_ordinario = FALSE \wedge \exists tp < 9/20$ ,  
então tornar  $cf = \text{"não admitido"}$  e terminar

5.3. se  $at < 8/20 \vee cd < 9.5/20$ , então tornar  $cf = \text{"reprovado"}$ ,  
senão tornar  $cf = \text{"aprovado com" } ROUND(cd) \text{ "valores"}$

© RMAC XI-2001

56

## 2. Representação de Algoritmos (1/40)

- fundamentos -

### ■ Princípios de representação

- os algoritmos devem ser representados recorrendo a uma linguagem que garanta o rigor e/ou formalismo das acções descritas
- numa primeira abordagem à solução algorítmica, a linguagem utilizada deve ser relativamente fácil de compreender para não se tornar, ela própria, um entrave e um outro problema a resolver, para além do da construção de uma solução algorítmica para o problema algorítmico proposto
- apesar do desejo de simplicidade, a linguagem utilizada não deve ser facilitista, no sentido e que permite uma abordagem imprecisa e ambígua
- antes pelo contrário, a linguagem utilizada deve garantir o suporte a todos os princípios da engenharia de software, tal como sugerido pelo meta-algoritmo anteriormente apresentado

© RMAC XI-2001

57

## 2. Representação de Algoritmos (2/40)

- fundamentos -

### ■ Decisões de concepção e implementação

- uma vez que um algoritmo deve ser visto como uma solução conceptual para o problema algorítmico proposto, todas as *decisões de concepção e implementação* devem ser deixadas para mais tarde, aquando da execução de fases de desenvolvimento mais avançadas em que o objectivo principal consiste na produção de um sistema que resolva, efectivamente, o problema proposto, de uma forma automática e com suporte computacional
- desta forma, a linguagem utilizada para representar o algoritmo não deve impor ou promover a introdução de decisões de concepção ou implementação prematuramente, tal como sucede, habitualmente, quando se recorre a linguagens de programação para descrever algoritmos

© RMAC XI-2001

58

## 2. Representação de Algoritmos (3/40)

- linguagens algorítmicas -

### ■ Caracterização

- uma linguagem que consiga responder a todas estas exigências e que possua as propriedades enunciadas designa-se de *linguagem algorítmica*
- uma linguagem algorítmica deve suportar a descrição de algoritmos, mas não de soluções finais (programas de computador)
- um algoritmo totalmente refinado, com instruções elementares puramente numéricas e computável não corresponde ainda a um programa de computador, porque falta introduzir as decisões de concepção e implementação
- ou seja, uma linguagem algorítmica deve suportar a descrição do "*o quê?*", mas não do "*como?*", porque esta deve ser suportada somente aquando da adopção de uma linguagem de programação para produzir soluções finais (programas executáveis)

© RMAC XI-2001

59

## 2. Representação de Algoritmos (4/40)

- linguagens algorítmicas -

### ■ Tipos de linguagens algorítmicas

- é possível conceber vários tipos de linguagens algorítmicas, tendo em conta as exigências e as características do problema algorítmico proposto
  - *linguagem natural*, quando as acções são descritas textualmente, recorrendo a uma linguagem natural (português, inglês, etc.)
  - *diagramas*, quando as acções são descritas graficamente, recorrendo a grafos dirigidos, tais como os diagramas de estados e os fluxogramas
  - *linguagens de modelação*, quando as acções são descritas textualmente, recorrendo a linguagens artificiais propositadamente concebidas para descrever modelos abstractos de sistemas, sem decisões de implementação
  - *especificação formal*, quando as acções são descritas simbolicamente, recorrendo a linguagens formais essencialmente matemáticas

© RMAC XI-2001

60

## 2. Representação de Algoritmos (5/40)

- fluxogramas -

© RMAC XI-2001

### ■ Caracterização

- os fluxogramas são, também, chamados de diagramas de fluxo de actividades, uma vez que possibilitam a representação de sequências de actividades geridas por um fluxo de controlo (que representa a sequência de passos do algoritmo)
- os fluxogramas possuem as seguintes propriedades
  - o fluxo de controlo entre as actividades é representado à custa de arcos dirigidos que unem actividades
  - a transição entre duas actividades é activada logo que a actividade precedente esteja completa
  - a evolução do fluxo de execução não deve depender de estímulos externos

61

## 2. Representação de Algoritmos (6/40)

- fluxogramas -

© RMAC XI-2001

### ■ Caracterização (cont.)

- os fluxogramas utilizam símbolos diferentes para cada tipo de instrução algorítmica
- por vezes, em representações relaxadas
  - o símbolo de início de algoritmo (*start*) é o mesmo do de conclusão (*termination*)
  - o de entrada e saída de dados (*input/output*) é o mesmo do de acção (*process*)
- o símbolo de conector (*connector*) é utilizado quando se pretende evitar o desenho de um arco que atravessa uma mesma página de uma ponta a outra, ou que descreve um percurso graficamente complicado
- o símbolo de conector inter-página (*off-page connector*) é utilizado quando o fluxograma de um mesmo algoritmo necessita de ser representado em mais do que uma página

62

## 2. Representação de Algoritmos (7/40)

- fluxogramas -

### ■ Caracterização (cont.)

- o símbolo de decisão (*decision*) permite representar testes de condições; as duas únicas respostas possíveis para cada teste de decisão (verdadeiro ou falso) dão origem a arcos que devem sair pelos dois extremos laterais opostos do diamante
- o símbolo de sub-rotina (*sub-routine*) permite, à custa dos princípios da modularidade e da abstracção, representar passos não refinados, mas cuja descrição pormenorizada surge, algures, noutra descrição fora do fluxograma em causa
- os fluxogramas devem ser representados com clareza, de forma a serem rigorosos na representação do algoritmo e a evitarem qualquer tipo de ambiguidade na sua interpretação
- no desenho de cada fluxograma deve garantir-se que a sequência principal de execução das acções do algoritmo é de cima para baixo e da esquerda para a direita

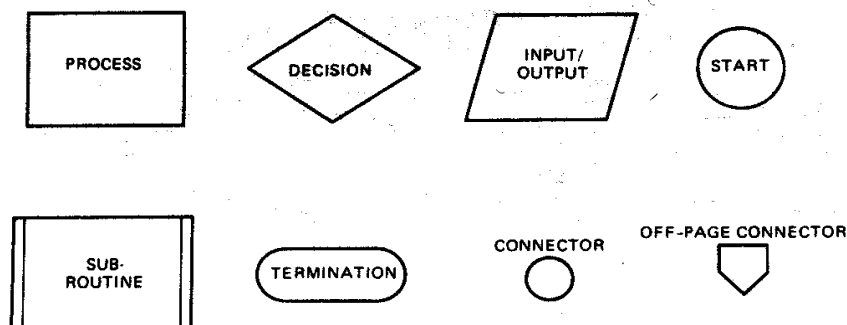
© RMAC XI-2001

63

## 2. Representação de Algoritmos (8/40)

- fluxogramas -

### ■ Símbolos elementares dos fluxogramas



© RMAC XI-2001

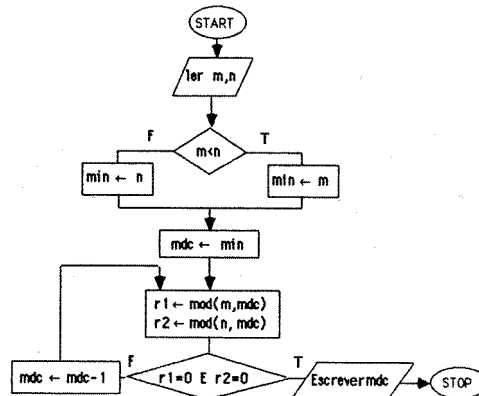
64



## 2. Representação de Algoritmos (9/40)

- fluxogramas -

### ■ O fluxograma do algoritmo do MDC



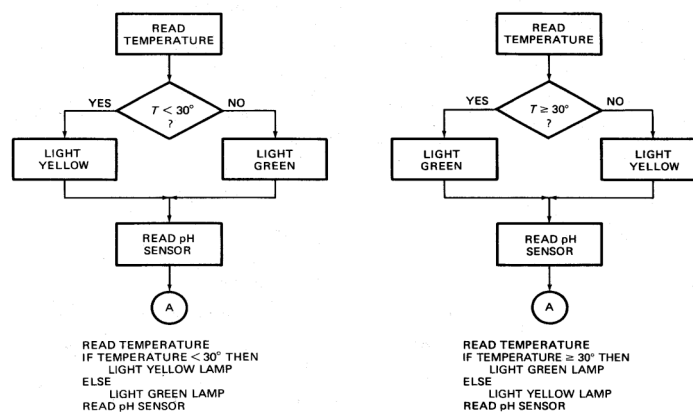
© RMAC XI-2001

65

## 2. Representação de Algoritmos (10/40)

- fluxogramas -

### ■ Fluxogramas equivalentes para decisão



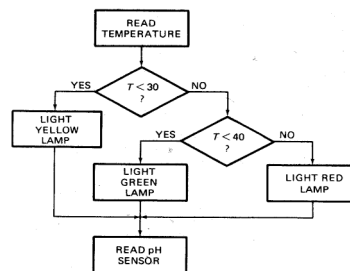
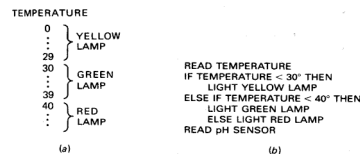
© RMAC XI-2001

66

## 2. Representação de Algoritmos (11/40)

### - fluxogramas -

#### ■ Decisões em cascata



© RMAC XI-2001

67

## 2. Representação de Algoritmos (12/40)

### - especificação formal -

#### ■ Especificação algébrica em IPS

**Spec:** Sequence;  
**extend** Natural, Primitivesort by  
**Sorts:** Note;  
**Operations:**  
 empty :  $\rightarrow$  Note;  
 addf :  $\text{Word} \times \text{Note} \rightarrow \text{Note}$ ;  
 tail :  $\text{Note} \rightarrow \text{Note}$ ;  
 head :  $\text{Note} \rightarrow \text{Word}$ ;  
 cat :  $\text{Note} \times \text{Note} \rightarrow \text{Note}$ ;  
 length :  $\text{Note} \rightarrow \text{Nat}$ ;  
 eq :  $\text{Note} \times \text{Note} \rightarrow \text{Bool}$ ;  
**Variables:**  
 u, v : Note;  
 x, y : Word;  
**Axioms:**  
 length(empty) = 0;  
 length(addf(x, u)) = 1 + length(u);  
 eq(empty, empty) = true;  
 eq(empty, addf(x, u)) = false;  
 eq(addf(x, u), empty) = false;  
 eq(u, v) = (head(u) == head(v))  $\wedge$  eq(tail(u), tail(v));  
 head(addf(x, u)) = x;  
 tail(empty) = empty;  
 tail(addf(x, u)) = u;  
 cat(empty, u) = u;  
 cat(u, empty) = u;  
 cat(addf(x, u), v) = addf(x, cat(u, v));

© RMAC XI-2001

68

## 2. Representação de Algoritmos (13/40)

- especificação formal -

### ■ Especificação com conjuntos e lógica de predicados em Z

```

Allocate
ΔComputerSystem
pid? : PROCESS_ID
rtype? : ResourceType

pid? ∈ dom processes
rtype? ∈ dom resources
(∃ rins : ResourceInstance |
  rins ∈ resources rtype? ∧ rins.status = Free •
  allocated' = allocated ∪ {rins.id ↦ pid?} ∧
  (resources' = resources ⊕ {r : ResourceType | r ∈ dom resources •
    if r = rtype? then r ↦ set_status(resources r, rins.id)
    else r ↦ resources r}) ∧
  waiting_queues' = waiting_queues ∧
  processes' = processes)
¬ (∃ rins : ResourceInstance •
  rins ∈ resources rtype? ∧ rins.status = Free) ⇒
  resources' = resources ∧
  allocated' = allocated ∧
  (waiting_queues' = waiting_queues ⊕
    {rtype? ↦ (waiting_queues rtype?) ∩ {pid?}}) ∧
  (processes' = processes ⊕
    {pid? ↦ (μ ps : ProcessStructure |
      ps.status = Waiting ∧
      ps.resources_needed =
        (processes pid?).resources_needed)})

```

© RMAC XI-2001

69

## 2. Representação de Algoritmos (14/40)

- OCL -

### ■ UML vs. OCL

- UML (*unified modeling language*) é uma linguagem de modelação genérica, normalizada pela OMG (*object management group*), que é utilizada para especificar, visualizar, construir e documentar artefactos de sistemas, tipicamente, baseados em software
- dentro da norma UML, existe a linguagem OCL (*object constraint language*) que foi especialmente desenvolvida para especificar invariantes, pré-condições, pós-condições, bem como outros tipos de restrições aos modelos descritos em UML
- OCL foi desenvolvida com o intuito de especificar restrições de uma forma formal, mas sem recorrer a pesados artefactos matemáticos, tornando-se numa linguagem de modelação precisa, mas de fácil compreensão
- a linguagem OCL é originária da IBM, mais especificamente do método *Syntropy* dedicado à modelação de negócio

© RMAC XI-2001

70

## 2. Representação de Algoritmos (15/40)

- OCL -

© RMAC XI-2001

### ■ Caracterização

- OCL não é uma linguagem de programação, pelo que, na sua semântica original, não permite
  - representar fluxos de controlo
  - invocar processos
  - activar operações *non-query*
- OCL é, tipicamente, utilizada para
  - especificar invariantes sobre classes e tipos
  - especificar invariantes de tipo sobre estereótipos
  - descrever pré- e pós-condições sobre operações e métodos
  - descrever guardas
  - descrever percursos de navegação sobre associações
  - especificar restrições sobre operações
- OCL é uma linguagem baseada em expressões, ou seja, quando uma expressão OCL é avaliada devolve sempre um valor

71

## 2. Representação de Algoritmos (16/40)

- OCL -

© RMAC XI-2001

### ■ Caracterização (cont.)

- em OCL é garantido que as expressões não possuem efeitos colaterais, ou seja, não conseguem realizar alterações ao modelo
- OCL é uma linguagem tipada, pelo que todas as expressões OCL possuem um tipo e as relações entre expressões só é possível serem estabelecidas se as expressões devolverem valores do mesmo tipo
  - *exemplo:* não é possível comparar um inteiro com uma string
- pelo facto da OCL ser uma linguagem de modelação, nem todas as expressões são, garantidamente, executáveis
- como linguagem de modelação, a OCL encontra-se completamente desprovida de instruções que introduzam decisões de implementação

72

## 2. Representação de Algoritmos (17/40)

- OCL -

### ■ OCL\*

- não fossem as limitações de não permitir a representação de fluxos de controlo nem de permitir efeitos colaterais, a linguagem OCL possuiria um conjunto de características adequadas para ser utilizada como linguagem de modelação algorítmica
- apesar dessa limitação (semântica) original, a linguagem OCL permite introduzir aos iniciados nestes assuntos, de uma forma acessível, vários conceitos úteis na programação de sistemas informáticos
- por esta razão e também como forma de ir adiantando o trabalho da disciplina de LP e SI1, PE adopta a linguagem OCL\* como linguagem "oficial" de modelação algorítmica
- a linguagem OCL\* distingue-se, unicamente, da OCL pelo facto de libertar a restrição de não permitir a representação de fluxos de controlo nem da existência de efeitos colaterais

© RMAC XI-2001

73

## 2. Representação de Algoritmos (18/40)

- tipos básicos em OCL -

### ■ Tipo *Boolean*

- toda e qualquer informação que pode assumir apenas dois valores diferentes
  - exemplo: *true* e *false*

### ■ Operações sobre o tipo *Boolean*

Operation	Notation	Result type
or	a or b	Boolean
and	a and b	Boolean
exclusive or	a xor b	Boolean
negation	not a	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implies	a implies b	Boolean
if then else	if a then b else b' endif	Type of b and b'

© RMAC XI-2001

74

## 2. Representação de Algoritmos (19/40)

- tipos básicos em OCL -

### ■ Tipo *Integer*

- toda e qualquer informação numérica que pertença ao conjunto dos números naturais
  - exemplos: 10 e 16738856

### ■ Tipo *Real*

- toda e qualquer informação numérica que pertença ao conjunto dos números reais
  - exemplo: 10,25 e -56774,8897
- **Nota #1:** uma vez que OCL é uma linguagem de modelação, não existem restrições no suporte à representação de dados, pelo que a noção de número máximo representável não existe
- **Nota #2:** o tipo de dados *Integer* é um subconjunto do tipo de dados *Real*

© RMAC XI-2001

75

## 2. Representação de Algoritmos (20/40)

- tipos básicos em OCL -

### ■ Operações sobre os tipos *Integer* e *Real*

Operation	Notation	Result type
equals	a = b	Boolean
not equals	a <> b	Boolean
less	a < b	Boolean
more	a > b	Boolean
less or equal	a <= b	Boolean
more or equal	a >= b	Boolean
plus	a + b	Integer or Real
minus	a - b	Integer or Real
multiplication	a * b	Integer or Real
division	a / b	Real
modulus	a.mod(b)	Integer
integer division	a.div(b)	Integer
absolute value	a.abs	Integer or Real
maximum of a and b	a.max(b)	Integer or Real
minimum of a and b	a.min(b)	Integer or Real
round	a.round	Integer
floor	a.floor	Integer

© RMAC XI-2001

76

## 2. Representação de Algoritmos (21/40)

- tipos básicos em OCL -

### ■ Tipo *String*

- toda e qualquer informação composta por caracteres (tem que iniciar por uma letra)

- exemplos: `'DSI'` `'programação estruturada'` `'LIGues_2'`

### ■ Operações sobre o tipo *String*

Operation	Expression	Result type
concatenation	<code>string.concat(string)</code>	String
size	<code>string.size</code>	Integer
to lower case	<code>string.toLower</code>	String
to upper case	<code>string.toUpper</code>	String
substring	<code>string.substring(int,int)</code>	String
equals	<code>string1 = string2</code>	Boolean
not equals	<code>string1 &lt;&gt; string2</code>	Boolean

© RMAC XI-2001

77

## 2. Representação de Algoritmos (22/40)

- tipos não básicos em OCL -

### ■ Tipo *Enumeration*

- toda e qualquer informação que o utilizador pretenda definir, por enumeração, para ir de encontro com as suas necessidades
- a definição de tipos de dados por enumeração permite
  - a posterior declaração de variáveis pertencentes a esse tipo
  - a utilização dos enumerados em expressões OCL (desde que precedidos do caracter #)
  - a declaração de módulos que retornam valores do tipo definido
- sintaxe

`enum {value1, value2, value3, ..., valuen}`

- exemplo: `gender: enum {male, female}`  
`gender = #male implies title = 'Mr.'`

© RMAC XI-2001

78

## 2. Representação de Algoritmos (23/40)

- tipos não básicos em OCL -

© RMAC XI-2001

### ■ Supertipo *Collection*

- toda e qualquer informação representável por uma colecção de dados
- *Collection* é um supertipo abstracto, ou seja
  - por ser *abstracto*, não pode originar instâncias de dados
  - por ser um *supertipo*, dá origem a tipos mais refinados, nomeadamente ao tipo *Set*, ao tipo *Bag* e ao tipo *Sequence*
- a conceptualização de um supertipo abstracto permite que sejam definidas uma única vez várias operações genéricas aplicáveis indistintamente a qualquer um dos 3 subtipos
- em OCL, qualquer operação sobre instâncias de subtipos de *Collection* não modificam a colecção, mas podem originar uma outra colecção

79

## 2. Representação de Algoritmos (24/40)

- tipos não básicos em OCL -

© RMAC XI-2001

### ■ Tipo *Set*

- é uma colecção que contém instâncias de um tipo válido em OCL, mas que não permite a existência de elementos repetidos, ou seja, corresponde, matematicamente, a um conjunto
  - exemplos: `Set {1, 2, 5, 88}`  
`Set {'maçã', 'laranja', 'morango'}`

### ■ Tipo *Bag*

- é um *Set* que aceita repetições de elementos, ou seja, uma mesma instância de um tipo válido em OCL pode ocorrer mais do que uma vez num *Bag*
  - exemplos: `Bag {1, 2, 5, 2, 88}`  
`Bag {'maçã', 'laranja', 'morango', 'maçã'}`

80



## 2. Representação de Algoritmos (25/40)

- tipos não básicos em OCL -

### ■ Tipo *Sequence*

- é um *Bag* em que os elementos se encontram ordenados segundo um determinado critério

- exemplos: `Sequence {1, 2, 2, 5, 88}`  
critério → ordenação crescente do Bag `{1, 2, 5, 2, 88}`  
`Sequence { 'melão', 'maçã', 'morango' }`  
critério → ordenação por tamanho decrescente

- *Nota*: os *Set* e os *Bag* não possuem uma ordem pré-definida

- quando, numa *Sequence*, existir um número elevado de elementos consecutivos, é possível representá-la por compreensão, utilizando a expressão `limit_1 .. limit_2`

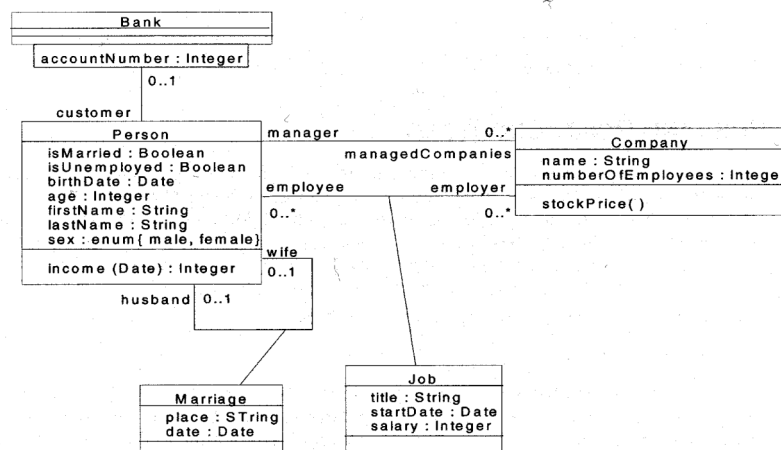
- exemplo: `Sequence {1 .. 10}`  
`Sequence {1 .. (6 + 4)}`  
`Sequence {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}`

© RMAC XI-2001

81

## 2. Representação de Algoritmos (26/40)

- Exemplo em OCL -



© RMAC XI-2001

82

## 2. Representação de Algoritmos (27/40)

- exemplo em OCL -

### ■ Caracterização da "entidade" *Person*

#### – dados

isMarried: Boolean  
isUnemployd: Boolean  
birthDate: Date  
age: Integer  
firstName: String  
lastName: String  
sex: enum {male, female}

#### – operações

income (Date): Integer

© RMAC XI-2001

83

## 2. Representação de Algoritmos (28/40)

- tipos não básicos em OCL -

### ■ Operações sobre o supertipo *Collection*

- *select* permite gerar uma sub-*collection* a partir da selecção dos elementos que numa primeira *collection* cumprem um determinado critério

#### – sintaxe:

*collection* → *select (boolean-expression)*  
*collection* → *select (v | boolean-expression-with-v)*  
*collection* → *select (v: Type | boolean-expression-with-v)*

#### ■ exemplo:

##### Company

self.employee → select (age > 50)  
self.employee → select (p | p.age > 50)  
self.employee → select (p: Person | p.age > 50)

© RMAC XI-2001

84

## 2. Representação de Algoritmos (29/40)

- tipos não básicos em OCL -

© RMAC XI-2001

### ■ Operações sobre o supertipo *Collection* (cont.)

- *reject* permite gerar uma sub-*collection* a partir da selecção dos elementos que numa primeira *collection* não cumprem um determinado critério

- **sintaxe:**

*collection* → *reject (boolean-expression)*

*collection* → *reject (v | boolean-expression-with-v)*

*collection* → *reject (v: Type | boolean-expression-with-v)*

- **equivalente a**

*collection* → *select (v: Type | not (boolean-expression-with-v) )*

- **exemplo:**

Company self.employee → reject (isMarried)

85

## 2. Representação de Algoritmos (30/40)

- tipos não básicos em OCL -

© RMAC XI-2001

### ■ Operações sobre o supertipo *Collection* (cont.)

- *collect* permite gerar uma *collection* a partir de outra *collection*, mas em que os elementos da primeira não pertencem à segunda, ou seja, a *collection* gerada não é uma sub-*collection* da original

- **sintaxe:**

*collection* → *collect (expression)*

*collection* → *collect (v | expression-with-v)*

*collection* → *collect (v: Type | expression-with-v)*

- **exemplo:**

Company

self.employee → collect (birthDate)

self.employee → collect (p | p.birthDate)

self.employee → collect (p: Person | p.birthDate)

86

## 2. Representação de Algoritmos (31/40)

- tipos não básicos em OCL -

© RMAC XI-2001

### ■ Operações sobre o supertipo *Collection* (cont.)

- o resultado de um *collect* é um *Bag* e não um *Set*
- o *Bag* resultante de um *collect* é do mesmo tamanho da *collection* original
- é possível transformar um *Bag* num *Set* (eliminando os elementos repetidos), recorrendo à operação *asSet* válida sobre um *Bag*

#### ■ exemplo:

Company self.employee -> collect (birthDate) -> asSet

- sintaxe simplificada:

*collection.property-name*

#### ■ exemplo:

Company self.employee.birthDate

87

## 2. Representação de Algoritmos (32/40)

- tipos não básicos em OCL -

© RMAC XI-2001

### ■ Operações sobre o supertipo *Collection* (cont.)

- *forAll* permite gerar um resultado cujo valor *Booleano* depende do cumprimento de uma determinada condição por parte de todos os elementos da *Collection*

- sintaxe:

*collection* -> forAll (*boolean-expression*)

*collection* -> forAll (*v* | *boolean-expression-with-v*)

*collection* -> forAll (*v*: *Type* | *boolean-expression-with-v*)

#### ■ exemplo:

Company

self.employee -> forAll (firstName = 'Jack')

self.employee -> forAll (p | p.firstName = 'Jack')

self.employee -> forAll (p: Person | p.firstName = 'Jack')

88

## 2. Representação de Algoritmos (33/40)

- tipos não básicos em OCL -

### ■ Operações sobre o supertipo *Collection* (cont.)

- a operação *forAll* pode ser estendida, permitindo a utilização de mais do que um iterador *v* sobre a mesma *Collection* de entrada

#### ■ exemplo:

##### Company

```
self.employee -> forAll (Person e1, e2 |  
                        e1 < > e2 implies e1.firstName < > e2.firtsName)
```

é equivalente a

##### Company

```
self.employee -> forAll (e1 | self.employee -> forAll (e2 |  
                        e1 < > e2 implies e1.firstName < > e2.firtsName)
```

© RMAC XI-2001

89

## 2. Representação de Algoritmos (34/40)

- tipos não básicos em OCL -

### ■ Operações sobre o supertipo *Collection* (cont.)

- *exists* permite gerar um resultado cujo valor *Booleano* depende da existência de, pelo menos, um elemento na *Collection* que cumpra uma determinada condição

#### - sintaxe:

*collection* -> *exists* (*boolean-expression*)

*collection* -> *exists* (*v* | *boolean-expression-with-v*)

*collection* -> *exists* (*v*: *Type* | *boolean-expression-with-v*)

#### ■ exemplo:

##### Company

```
self.employee -> exists (firstName = 'Jack')
```

```
self.employee -> exists (p | p.firstName = 'Jack')
```

```
self.employee -> exists (p: Person | p.firstName = 'Jack')
```

© RMAC XI-2001

90

## 2. Representação de Algoritmos (35/40)

- tipos não básicos em OCL -

### ■ Operações sobre o supertipo *Collection* (cont.)

- *iterate* permite construir um valor, iterando sobre uma *Collection*

- sintaxe:

*collection* → *iterate* (*elem*: *Type*; *acc*: *Type* = <*expression*> |  
*expression-with-elem-and-acc*)

- a variável *elem* é o iterador sobre a *Collection*
  - a variável *acc* é o acumulador
  - a variável *acc* recebe inicialmente o valor de <*expression*>
  - durante a iteração sobre todos os elementos de *Collection*, e depois da *expression-with-elem-and-acc* ter sido calculada o seu valor é acumulado na variável *acc*
- *iterate* é a operação mais elementar do supertipo *Collection*, uma vez que todas as outras operações podem ser deduzidas a partir dela

© RMAC XI-2001

91

## 2. Representação de Algoritmos (36/40)

- tipos não básicos em OCL -

### ■ Operações sobre o supertipo *Collection* (cont.)

- *collection* → *size*: *Integer*  
post: result = *collection* → *iterate* (*elem*; *acc*: *Integer* = 0 | *acc* + 1)
- *collection* → *count* (*object*: *OclAny*): *Integer*  
post: result = *collection* → *iterate* (*elem*; *acc*: *Integer* = 0 |  
if *elem* = *object* then *acc* + 1 else *acc* endif)
- *collection* → *includes* (*object*: *OclAny*): *Boolean*  
post: result = (*collection* → *count* (*object*) > 0)
- *collection* → *includesAll* (*c2*: *Collection*(*T*)): *Boolean*  
post: result = *c2* → *forAll* (*elem* | *collection* → *includes* (*elem*))
- *collection* → *isEmpty*: *Boolean*  
post: result = (*collection* → *size* = 0)

© RMAC XI-2001

92

## 2. Representação de Algoritmos (37/40)

- tipos não básicos em OCL -

### ■ Operações especificamente sobre o tipo *Set*

```
set -> union (set2: Set(T)): Set(T)
set -> union (bag: Bag(T)): Bag(T)
set = (set2: Set): Boolean
set -> intersection (set2: Set(T)): Set(T)
set -> intersection (bag: Bag(T)): Bag(T)
set - (set2: Set(T)): Set(T)
set -> including (object: T): Set(T)
set -> excluding (object: T): Set(T)
set -> symmetricDifference (set2: Set(T)): Set(T)
set -> count (object: T): Integer
set -> asSequence: Sequence(T)
set -> asBag: Bag(T)
```

© RMAC XI-2001

93

## 2. Representação de Algoritmos (38/40)

- tipos não básicos em OCL -

### ■ Operações especificamente sobre o tipo *Bag*

```
bag -> union (bag2: Bag(T)): Bag(T)
bag -> union (set: Set(T)): Bag(T)
bag = (bag2: Bag): Boolean
bag -> intersection (bag2: Bag(T)): Bag(T)
bag -> intersection (set: Set(T)): Set(T)
bag -> including (object: T): Bag(T)
bag -> excluding (object: T): Bag(T)
bag -> count (object: T): Integer
bag -> asSequence: Sequence(T)
bag -> asSet: Set(T)
```

© RMAC XI-2001

94

## 2. Representação de Algoritmos (39/40)

- tipos não básicos em OCL -

### ■ Operações especificamente sobre o tipo *Sequence*

sequence → union (sequence2: Sequence(T)): Sequence(T)  
sequence → append (object: T): Sequence(T)  
sequence → prepend (object: T): Sequence(T)  
sequence = (sequence2: Sequence): Boolean  
sequence → subSequence (lower: Integer; upper: Integer): Sequence(T)  
sequence → at (i: Integer): T  
sequence → first: T  
sequence → last: T  
sequence → including (object: T): Sequence(T)  
sequence → excluding (object: T): Sequence(T)  
sequence → count (object: T): Integer  
sequence → asBag: Bag(T)  
sequence → asSet: Set(T)

© RMAC XI-2001

95

## 2. Representação de Algoritmos (40/40)

- tipos em OCL -

### ■ Compatibilidade de tipos

- OCL é uma linguagem tipada, ou seja, exige o cumprimento de um conjunto de regras de compatibilidade para operar com dados de tipos distintos
  - exemplo: não é possível comparar uma String com um Boolean
- uma expressão OCL em que todos os dados pertençam a tipos compatíveis é uma expressão válida
- um tipo *type\_1* é compatível com um *type\_2*, quando uma instância de *type\_1* pode substituir uma instância de *type\_2* em qualquer local em que este é esperado
- a compatibilidade de tipos é transitiva, ou seja, se um tipo *type\_1* for compatível com um tipo *type\_2* e se este for compatível com um tipo *type\_3*, então o tipo *type\_1* é compatível com o tipo *type\_3*

© RMAC XI-2001

96



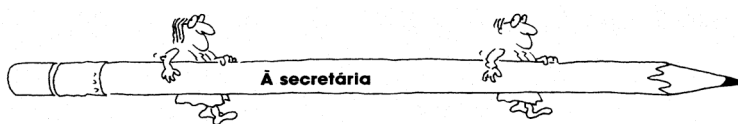


## exercícios

© RMAC XI-2001

1. Considere as regras de funcionamento e de avaliação especificadas no documento da aula de apresentação da disciplina de Programação Estruturada. Tomando como base os 2 algoritmos sugeridos na aula TP anterior (classificação de um aluno e classificação de todos os alunos), represente-os utilizando a notação gráfica de fluxograma.
2. Construa um algoritmo que receba 10 números inteiros fornecidos pelo utilizador e que devolve a soma deles. Represente o algoritmo utilizando a notação gráfica de fluxograma.
3. Construa um algoritmo que receba 10 números inteiros fornecidos pelo utilizador e que devolve o maior inteiro deles. Represente o algoritmo utilizando a notação gráfica de fluxograma.
4. Construa um algoritmo que resolva a seguinte equação pelo método das aproximações sucessivas:  $x^5 + 3x^2 - 10 = 0$ . Represente o algoritmo utilizando a notação gráfica de fluxograma.

97



#### 4. Método das aproximações sucessivas (exemplo)

→ resolver a equação em ordem a x

$$x^5 + 3x^2 - 10 = 0 \Leftrightarrow x = \sqrt[5]{10 - 3x^2}$$

→ arbitrar um valor inicial e uma resolução

$$x = 1 \text{ e } \Delta = 0.00001$$

→ aplicar iterativamente numa próxima equação o valor obtido na anterior

$$x = 1.00000 \Rightarrow x = \sqrt[5]{10 - 3(1.00000)^2} = 1.47577$$

$$x = 1.47577 \Rightarrow x = \sqrt[5]{10 - 3(1.47577)^2} = 1.28225$$

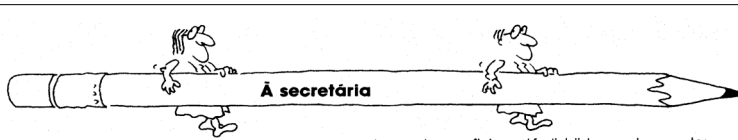
$$x = 1.28225 \Rightarrow x = \sqrt[5]{10 - 3(1.28225)^2} = 1.38344$$

$$x = 1.38344 \Rightarrow x = \sqrt[5]{10 - 3(1.38344)^2} = 1.33613 \quad \dots$$

→ deve manter-se a iteração enquanto se observar uma convergência para o valor da raiz a determinar e tendo em conta a resolução pretendida

© RMAC XI-2001

98



#### 4. Algoritmo em linguagem natural

→ dicionário de símbolos

**X** : o valor de  $x$  a introduzir na equação

**X1** : o valor de  $x$  calculado a partir do de  $X$

**I** : contador de iterações

**N** : número máximo de iterações

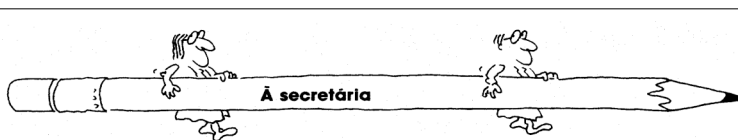
→ algoritmo não refinado

1. ler os valores de  $X$  e  $N$

2. iniciar o valor do contador

3. calcular o valor de  $X1$

4. escrever os valores de  $I$  e  $X1$



#### 4. Algoritmo em linguagem natural

→ algoritmo não refinado (cont.)

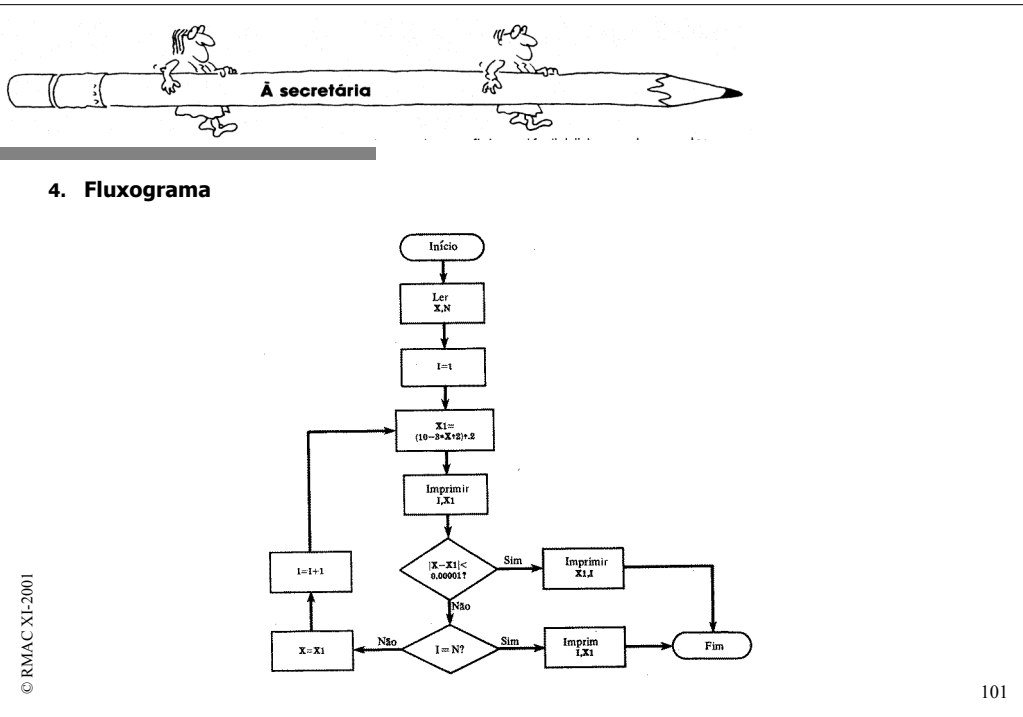
5. testar a resolução do resultado;  
se resolução atingida, então ir para 7, senão continuar

6. testar o valor de  $I$ ;  
se  $I = N$ , então ir para 8,  
senão  $I \leftarrow I + 1$  e  $X \leftarrow X1$  (calculado em 3) e ir para 3

7. escrever os valores finais de  $I$  e  $X1$  e ir para 9

8. escrever os valores finais de  $I$  e  $X1$ , indicando que solução não convergiu suficientemente rápido e ir para 9

9. terminar



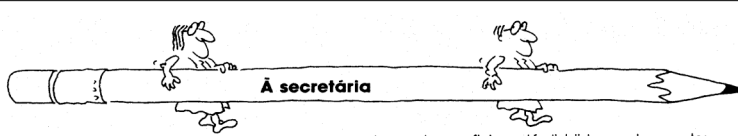
101

## exercícios

1. Diga quais das seguintes expressões em OCL são consideradas válidas.
  - a)  $1 + 2 * 34$
  - b)  $1 + \text{'motocycle'}$
  - c)  $23 * \text{false}$
  - d)  $12 + 13.5$
2. Refine em OCL\* as seguintes operações:
  - a) sequence  $\rightarrow$  excluding (object: T): Sequence(T)
  - b) collection  $\rightarrow$  forAll (expr: OCLexpression): Boolean
3. Construa um algoritmo que calcule a soma de uma sequência de valores numéricos. Represente o algoritmo em OCL\*.
4. Construa um algoritmo que devolva o maior de uma sequência de valores numéricos. Represente o algoritmo em OCL\*.

© RMAC XI-2001

102



- 2.a) sequence  $\rightarrow$  iterate (elem; acc: Sequence(T) = Sequence {} |  
if elem = object then acc  $\rightarrow$  append (elem) endif)
- 2.b) collection  $\rightarrow$  iterate (elem; acc: Boolean = true | acc and expr)
- 3. bag  $\rightarrow$  iterate (elem; acc: T = 0 | acc + elem)
- 4. bag  $\rightarrow$  iterate (elem; acc: T = 0 | if elem > acc then acc = elem)