

MAC0316 - CONCEITOS

VALORES E TIPOS

Os dados são caracterizados por três aspectos básicos: valores, tipos e variáveis.

VALORES

São representações simbólicas de conceitos. Se por exemplo, o conceito representado é a temperatura que marca um termômetro, o número indicado pela coluna de mercúrio seria um valor numérico.

TIPOS

Os valores relevantes para a resolução de um problema são classificados segundo algum critério, e uma classe de valores recebe o nome de um tipo.

Por exemplo, se todos os valores relevantes para resolver um problema são numéricos, os valores podem ser classificados segundo o seu tipo como naturais, inteiros e reais.

ANOTACÕES: valor é uma representação simbólica e um tipo é uma classe de valores.

VALORES E TIPOS EM UM PROGRAMA

Um programa é composto por mais de uma expressão, por isso os valores dos dados precisam ser registrados para passar de uma expressão para outra.

Os valores precisam ser repassados de uma expressão para outra dentro do programa. Os registros desses valores servem como repositório para passagem de valores, e eles são efetuados em variáveis.

Os valores são elementos que podem ser analisados, armazenados, atualizados e transmitidos durante a execução.

Os tipos devem representar os conjuntos de valores elementares que se deseja tratar nas linguagens de programação. Estes são representados por valores juntamente com as operações (pelos regras unificadoras) para tratamento dos mesmos.

Para efeitos de estudo dos conceitos, os tipos são divididos em dois grandes grupos:

- tipos primitivos
- tipos compostos

Além destes, podemos definir tipos em termos deles próprios, os tipos recursivos.

TIPOS PRIMITIVOS

Os tipos primitivos usam aqueles cujos valores são "átomicos", ou seja, não podem ser desmembrados em valores mais simples. A implementação de um tipo primitivo em um computador é feita necessariamente "fora" da linguagem de programação.

• TIPOS PRIMITIVOS NAS LPs

FORTRAN: foi criada para resolver problemas científicos. Os tipos primitivos usam relacionados com problemas numéricos. O foco era em números reais, variações de precisão, números inteiros e números complexos.

COBOL: LP criada para processamento de dados comerciais. Possui cadeias de caracteres como tipo primitivo.

APL E MATLAB: LPs voltadas para a resolução de problemas matemáticos relacionados com a álgebra de matrizes - têm matrizes como um tipo de dado primitivo.

• TIPOS PRIMITIVOS NUMÉRICOS

Alguns tipos numéricos primordiais para o tratamento de valores tanto em linguagens dedicadas à resolução de problemas numéricos quanto ao processamento de dados comerciais, aparecem na maioria das linguagens atuais.

Os tipos inteiro e real, por exemplo, aparecem com nomenclatura diferente nas várias linguagens de programação atual.

• TIPOS PRIMITIVOS NÃO-NUMÉRICOS

Nem todos os problemas a serem resolvidos computacionalmente possuem natureza exclusivamente numérica, e por isso tipos que representem valores não numéricos devem também ser provados pelas linguagens. Os valores mais comuns são os valores booleanos (0 e 1), agrupados no tipo booleano, e os caracteres que representam símbolos padrão, agrupados no tipo caracteres.

TIPOS PRIMITIVOS ENUMERADOS

- Alguns problemas requerem a construção de novos valores.
- Construção de novos conjuntos de valores definidos pelos usuários, através da enumeração.
- Em algumas linguagens de programação, tais como Pascal e C, conjunto de dados primitivos podem também ser criados pela enumeração de seus elementos.

Exemplo 3.8 – Em Pascal, por exemplo, podemos criar um tipo (conjunto de valores) para representar os meses do ano:

```
type MesesP = (jan, fev, mar, abr, mai, jun,
                jul, ago, set, out, nov, dez);
```

□

Exemplo 3.9 – De forma análoga, podemos ter em C este novo tipo definido por:

```
enum MesesC {jan, fev, mar, abr, mai, jun,
             jul, ago, set, out, nov, dez};
```

- Os elementos que aparecem na enumeração são tratados como valores.

TIPOS COMPOSTOS

- Conjuntos de dados onde cada um dos seus elementos pode ser desmembrado em valores mais simples.
- (produto cartesiano, união disjunta, mapeamentos e conjuntos potência)

TIPO COMPOSTO: PRODUTO CARTESIANO

O produto cartesiano de dois conjuntos corresponde a todos os pares formados por valores destes conjuntos, de forma que o primeiro elemento do par pertence ao primeiro conjunto e o segundo elemento do par pertence ao segundo conjunto.

Exemplo 3.12 – As datas do ano, denotadas pelo par (mês, dia), podem ser representadas em Pascal pelo produto cartesiano (record) de mês (MesesP, Exemplo 3.8), pelo dia (DiasP, Exemplo 3.10):

```
type DataP = record
  m: MesesP
  d: DiasP
end;
```

TIPO COMPOSTO: UNIÃO DISJUNTA

Um caso especial da união de conjuntos é a chamada união disjunta. Nesta, o conjunto resultante possui todos os elementos dos dois conjuntos originais, e além disso podemos distinguir o conjunto origem de cada elemento; isto é, do primeiro ou do segundo conjunto.

- $S + T = \{ \text{prim } x \mid x \in S \} \cup \begin{matrix} \text{um valor} \\ \{ \text{seg } y \mid y \in T \} \end{matrix}$
- livre
→ por vez
- $\#(S + T) = \#S + \#T$

União disjunta discriminada: exemplo

- A = {jan, fev, mar}
- B = {1, 2, 3}
- A + B = {prim jan, seg 1,
prim fev, seg 2,
prim mar, seg 3}
- $\#(A + B) = \#A + \#B = 3 + 3 = 6$

TIPO COMPOSTO: MAPEAMENTOS

O mapeamento de dois conjuntos resulta em um terceiro conjunto de pares de elementos, onde o primeiro elemento é originário do primeiro conjunto e o segundo é originário do segundo conjunto.

- A = {jan, fev, mar}
- B = {1, 2, 3}
- C = A → B = {(jan, 1), (fev, 2), (mar, 3)}
= {jan → 1, fev → 2, mar → 3}
- D = A → B = {(jan, 3), (fev, 1), (mar, 3)}
= {jan → 3, fev → 1, mar → 3}
- ...
- $\#(A \rightarrow B) = (\#B)^{\#A} = 3^3 = 27$

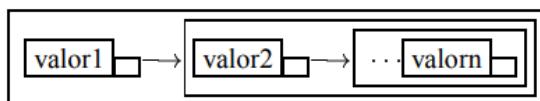
TIPO COMPOSTO: CONSUNTOS POTÊNCIA

Matematicamente, o conjunto formado por todos os possíveis subconjuntos de um dado conjunto é chamado de conjunto potência.

- A = {jan, fev, mar}
- $P(A) = \{\emptyset, \{\text{jan}\}, \{\text{fev}\}, \{\text{mar}\}, \{\text{jan, fev}\}, \{\text{jan, mar}\}, \{\text{fev, mar}\}, \{\text{jan, fev, mar}\}\}$
- $\#(P(A)) = (2)^{\#A} = 2^3 = 8$

TIPOS RECURSIVOS

- Um tipo recursivo é um tipo de dado para valores que podem conter outros valores do mesmo tipo. Isto indica que uma lista de α ou é uma lista vazia ou um elemento α (a cabeça da lista) seguido de uma lista de α (a cauda da lista).
- De forma geral, os tipos recursivos são compostos de valores que possuem o próprio tipo como valor, estes são definidos sem termos deles próprios.



VARIÁVEIS

- Os valores são agrupados em tipos para que determinemos um tratamento uniforme.
- Os valores são armazenados em **variáveis**, as quais permitem que os mesmos sejam usados em diversos pontos do programa.

O PAPEL DAS VARIÁVEIS NOS PROGRAMAS

- Na execução de um programa, uma variável é um objeto identificado por um 'nome fantasia' que contém um valor (o seu conteúdo), o qual pode ser consultado ou modificado tantas vezes quanto necessário.
- É importante salientar que para cada variável de um programa temos associado:
 - um identificador (o 'nome fantasia')
 - um endereço da(s) célula(s) de memória onde o valor é armazenado
 - o conteúdo (valor) na endereço de memória.

identificador → endereço de memória [conteúdo]

ARMAZENAMENTO E ACESSO A VALORES

As variáveis podem tanto armazenar um conteúdo atômico quanto composto.

VARIÁVEIS SIMPLES

As variáveis simples demoram espaços de memória que podem ter seus conteúdos armazenados e acessados ativamente nas linguagens de programação.

Não há como armazenar apenas parte dos valores.

- **TIPOS PRIMITIVOS**: os valores dos tipos primitivos são sempre referidos como elementos atômicos, mesmo que sejam armazenados internamente na máquina por elementos que possam ser eventualmente desmembrados.

Da mesma forma que para os tipos numéricos, os valores caracteres e booleans também são armazenados em variáveis simples.

Os tipos enumerados da linguagem Pascal também têm seus valores armazenados em variáveis simples.

Outro tipo primitivo comum nas linguagens de programação são os apontadores, os quais têm como valores endereços de memória donde se encontram os conteúdos (valores) desejados.

- **TIPOS COMPOSTOS**: A maioria dos tipos compostos tem seus valores tratados de forma vetorial e por isso armazenados em variáveis compostas.
Os conjuntos, contudo, como implementados em Pascal e Modula-3, são tratados como um único elemento e não existe acesso direto aos seus componentes (ao i-ésimo componente, por exemplo).
- **TIPOS RECURSIVOS**: Linguagens que embedem listas como tipo predefinido (linguagens funcionais) não fazem acesso e armazenamento vetorial dos seus elementos. Nesse contexto, as listas são tratadas como elementos únicos, não podemos ter acesso ao i-ésimo elemento da lista, exceto via operações sucessivas.

VARIÁVEIS COMPOSTAS

As variáveis compostas possuem componentes que podem ser tratados vetorialmente.

- produto cartesiano
- união disjunta
- mapeamentos
- Tipos recursivos

AS VARIÁVEIS E SUA EXISTÊNCIA

As variáveis também são classificadas quanto à sua existência ao longo da execução do programa.

Uma variável existe desde o momento em que é associada (vinculada) a uma célula de memória (a chamada alocação) da variável, e cessa sua existência quando o dado espaço de memória é disponibilizado (desalocação).

• VARIÁVEIS GLOBAIS E LOCAIS

- As variáveis globais são aquelas vinculadas às células de memória antes da execução do programa e assim permanecem até que a execução do programa se encerre.

- O tempo de vida das variáveis **locais** está relacionado aos blocos de execução nos quais elas estão inseridas.

- **VARIÁVEIS INTERMITENTES (HEAP).**

As variáveis intermitentes são criadas e destruídas em tempo de execução.

O que difere substancialmente estas variáveis das locais é o controle explícito da sua existência pelos comandos de criação e destruição, enquanto as variáveis locais têm sua existência relacionada a blocos de execução.

- **VARIÁVEIS PERSISTENTES.**

- O armazenamento de dados não é restrito apenas ao tempo de execução dos programas.
- Dados já previamente processados ou simplesmente armazenados.
- Arquivos, bancos de dados como meio de armazenamento.
- Variáveis do tipo arquivo e operações predefinidas.

VINCULAÇÕES E VERIFICAÇÃO DE TIPOS

De forma geral, o termo vinculação se refere à associação entre elementos.

A vinculação de um tipo a uma variável, por exemplo, associa aquela variável aos possíveis valores que ela pode assumir.

VINCULAÇÕES

- associação de entidades de um programa (tal como variáveis, funções, etc) a atributos relativos àquelas entidades no contexto do programa (um valor, um tipo, uma estrutura ou ainda uma abstração).
- Cada entidade do programa só pode ser processada quando estes atributos são definidos de forma exata, e esta definição exata dos atributos chamamos de vinculação.
- Vinculação é, portanto, um conceito fundamental relativo ao projeto de linguagens de programação:
 - as entidades existentes: o que pode ser vinculado
 - os atributos necessários: de que forma
 - tempo em que cada vinculação ocorre: quando estes elementos podem ser vinculados.
- TEMPO DE VINCULAÇÃO
 - Quando as vinculações são realizadas
 - ESTÁTICA: uma variável é dita **estática** se ocorrer antes da execução e permanecer inalterada ao longo da execução do programa. As variáveis têm seus tipos definidos em tempo de compilação.
 - DINÂMICA: uma variável é dita **dinâmica** se ocorrer em tempo de execução ou se for modificada ao longo da execução. As operações realizadas sobre as variáveis podem também ser verificadas com relação aos tipos em tempo de compilação.
- ESCOPO

Quais os elementos da linguagem que são visíveis nas várias partes dos programas.

SISTEMAS DE TIPOS

• MONOMORFISMO

- Uma linguagem tem um sistema de tipos monomórficos quando cada elemento declarado na linguagem possui um único tipo.
- A maioria das linguagens de programação imperativas.
- Um exemplo de tipos monomórficos são as constantes, variáveis, parâmetros e resultados de funções nas linguagens Pascal e C.
 - cada um destes elementos possui um tipo determinado na própria declaração, o qual não pode ser modificado em tempo de compilação ou execução

Exemplo 5.3 – Uma estrutura de pilha de inteiros e outra de caracteres podem ser definida na linguagem Pascal, por exemplo:

```
type stackint = ^ intnode;
    intnode = record
        elem : Integer;
        next : ^stackint;
    end;

type stackchar = ^ charnode;
    charnode = record
        elem : Char;
        next : ^stackchar;
    end;
```

- Duas estruturas de pilha precisaram ser definidas, uma para caracteres e outra para inteiros.
- A linguagem não permite que a estrutura seja definida para um tipo genérico.

• SOBRECARGA

- A sobrecarga é o uso de um mesmo identificador para operações diferentes, ou seja um mesmo identificador pode denotar comportamentos distintos.
- Os operadores de soma (+) e subtração (-), por exemplo, são usados tanto para números inteiros quanto reais na maioria das linguagens.

• SOBRECARGA INDEPENDENTE DE CONTEXTO

- A abstração a ser aplicada depende dos tipos dos argumentos, os quais devem corresponder aos parâmetros da abstração.

- EXEMPLO: operador de divisão na linguagem C (" / ") : a divisão de dois números inteiros terá como resultado um inteiro, independentemente se a variável à qual a expressão é atribuída é do tipo inteiro ou real.

• SOBRECARGA DEPENDENTE DO CONTEXTO

- a abstração a ser aplicada (função / operador) depende não só do tipo dos operandos a serem aplicados, mas também do tipo do resultado esperado na expressão onde está sendo aplicado.
- EXEMPLO: na linguagem ADA, podemos sobrecarregar o operador de divisão (" / ") e este depende não apenas dos operandos, mas também do tipo da variável para quem o resultado da divisão está sendo atribuído.

```

...
function "//" (m, n :Integer) return Float is
begin
    return Float(m) / Float(n);
end;

...
a, b, n : Integer;
x: Float;
...
n = a / b;      (1) {divisão de reais -
                     uso do operador "//" predefinido}
x = a / b;      (2) {divisão de inteiros -
                     uso do operador "//" novo}

```

• POLIMORFISMO

- O polimorfismo é uma propriedade que tem de uma única abstração ser usada para uma família de tipos; a abstração funciona de uma única forma, independentemente do tipo em uso. E, para isso, os tipos usados na aplicação de tal abstração estão relacionados.
- NÃO CONFUNDIR COM SOBRECARGA : na sobrecarga, usamos um mesmo identificador para exercer um conjunto de abstrações; e estas abstrações não precisam estar necessariamente relacionadas.

RESUMO:

- MONOMORFISMO: cada elemento declarado na linguagem possui um único tipo.
- SOBRECARGA: um mesmo identificador para exercer um conjunto de abstrações
- POLIMORFISMO: uma única abstração é usada para uma família de tipos

VERIFICAÇÃO DE TIPOS

- A verificação de tipos é a atividade de garantir que os operandos sejam de tipos compatíveis.
- Um tipo é equivalente (ou compatível) quando ele é válido para o operador ou tem permissão, segundo as regras da linguagem, para ser convertido automaticamente para um tipo válido (coerção de tipos).
- Um erro de tipo é tipicamente o uso inadvertido de operadores com operandos.

EQUIVALENCIA DE TIPOS

EQUIVALÊNCIA NOMINAL

- A forma mais simples de compatibilidade de tipos pode ser definida pela equivalência de nomes.
Dizemos que duas variáveis têm compatibilidade de nomes, se elas são declaradas com o mesmo tipo (mesmo nome do tipo).

Definição 5.3 Dados dois tipos de dados T e T' , eles são ditos **nominalmente equivalentes**, se e somente se $T = T'$ (mesmo nome, definido no mesmo local).

EQUIVALÊNCIA ESTRUTURAL

redução dos tipos para os elementos base

Definição 5.4 Dados dois tipos de dados T e T' , eles são ditos **estruturalmente equivalentes**, escrito por $T \equiv T'$, se e somente se T e T' têm o mesmo conjunto de valores.

Exemplo 5.11 – Suponha uma linguagem com os tipos definidos em termos de produto cartesiano, união disjunta e mapeamentos. A equivalência de tipos (T e T') pode ser verificada como segue:

- T e T' são tipos primitivos. Então, $T \equiv T'$ se e somente se T e T' são idênticos.
- $T = A \times B$ e $T' = A' \times B'$. Então, $T \equiv T'$ se e somente se $A \equiv A'$ e $B \equiv B'$.
- $T = A + B$ e $T' = A' + B'$. Então, $T \equiv T'$ se e somente se $A \equiv A'$ e $B \equiv B'$ ou $A \equiv B'$ e $B \equiv A'$.
- $T = A \rightarrow B$ e $T' = A' \rightarrow B'$. Então, $T \equiv T'$ se e somente se $A \equiv A'$ e $B \equiv B'$.
- para quaisquer outros casos, T e T' não são equivalentes.

- EQUIVALENCIA DE DECLARAÇÃO
- Tipos definidos por nomes de outros tipos.

Exemplo 5.12 – Suponha agora um tipo inteiro definido a partir de outro, o que é o caso de Int2 abaixo:

```
type Int1 = Integer;
    Int2 = Int1;
...
```

neste caso, as variáveis definidas por quaisquer dos tipos acima possuem tipos equivalentes. □

• COERÇÃO DE TIPOS

- mapeamento de valores de um dado tipo para valores de um outro tipo automaticamente.

• INFERRÊNCIA DE TIPOS

Exemplo 5.13 – Uma função que calcula a circunferência de um círculo, dado o raio, pode ser definida como:

```
fun circunf(r) = 3.14 * r * r;
```

Na função acima, o tipo de *r* não foi declarado, mas mesmo assim pode ser inferido.

• LINGUAGENS FORTEMENTE TIPIFICADAS

- Podem reconhecer todos os erros de tipo.
- O programador é advertido dos problemas relativos a tipos dos seus programas, seja em tempo de compilação ou execução.

EXPRESSÕES E COMANDOS

- transformação dos dados originais em novos dados produzidos como resultado dos programas.

O PROGRAMA COMO MÁQUINA ABSTRATA

- estados primordiais do programa:
 - estado inicial (nenhuma transformação)
 - estado final (após transformações)
- em linguagens funcionais: os dados de entrada representam o estado inicial, o processamento da função representa o programa, e o resultado da função representa o estado final

EXPRESSÕES

- papel: transformar valores em um programa (dados)
- são os elementos de transformação de dados em um programa: a partir de valores, uma transformação é aplicada e um valor resultado é produzido.

TIPOS DE EXPRESSÕES

EXPRESSÕES: LITERAIS

- denotam um valor fixo de algum tipo
- exemplos:
 - 10
 - 'a'
 - "maria"

EXPRESSÕES: AGREGAÇÃO DE VALORES

- expressões que constroem valores compostos a partir de outros valores mais simples
- exemplos:
 - um C: tamanho e tipo fixos

$$\text{int } n[5] = \{32, 45, 66, 23, 58\}$$

• EXPRESSÕES: APLICAÇÃO DE FUNÇÕES

• F(PA)

• F é o identificador da função

• PA é o parâmetro atual

• exemplos

$$\bullet 5 + 10 = \text{sum}(5, 10)$$

$$\bullet 5 > 10 = \text{maior?}(5, 10)$$

• EXPRESSÕES: CONDICIONAIS

• assumem valores diferentes dependendo da avaliação de uma expressão lógica.

• exemplo

$$\bullet 1, \text{ se } x \leq 0 \text{ ou}$$

$$2, \text{ se } x > 0$$

• EXPRESSÕES: VALORES ASSOCIADOS A IDENTIFICADORES

• acesso ao valor associado a um identificador é na verdade o cálculo de uma expressão.

• constantes: acesso a um valor diretamente

• variável: acesso a um valor que depende do estado do programa

• EXPRESSÕES: AVALIAÇÃO

• o resultado depende da ordem de avaliação

• expressões aritméticas: * e / , + e -

• expressões lógicas: NOT, AND, OR

COMANDOS

• responsáveis pelas mudanças de estados dos programas.

• mudanças em:

• valores armazenados (atribuições)

• controle (instruções compostas)

- TIPOS DE COMANDOS
- COMANDOS: ATRIBUIÇÃO SIMPLES
 - uma expressão é avaliada e o valor resultado é atribuído a uma única variável.
 - exemplos:
 - $x = y$
 - $x = y + 15$
- COMANDOS: ATRIBUIÇÕES MÚLTIPLAS
 - valor resultado da expressão atribuído a diversas variáveis.
 - exemplo: $x, y = 10$
- COMANDOS: ATRIBUIÇÕES INSERIDAS EM EXPRESOES
 - atribuições que são realizadas de forma intermediária ao cálculo de uma expressão.
 - exemplo: $x = a + (y = z / b++)$

ABSTRAÇÕES

- é o princípio pelo qual nos concentramos nos aspectos essenciais de um problema em vez de seus detalhes.
- um programa é uma abstração, mas dentro dele podemos ter outras abstrações

• ABSTRAÇÕES DE PROCESSOS

- uma abstração de processos deve "esconder" os passos computacionais necessários para solucionar o problema.
- FUNÇÕES: abstrações sobre mapeamentos de domínios (e a imagem da função)
- PROCEDIMENTOS: abstrações de comportamentos com efeito colateral

• ABSTRAÇÕES DE TIPOS

• ABSTRAÇÕES DE DADOS

- operações sobre os dados definidos
- os dados só podem ser manipulados com o uso das operações definidas sobre eles.
- EXISTENTES: as operações aplicadas a tipos primitivos não únicas e exclusivamente as predefinidas.
- CRIADAS: agrupamento de dados e abstrações que obrigam os dados a serem tratados de forma exclusivamente por suas operações.
 - módulos / pacotes e classes.

PROGRAMAÇÃO IMPERATIVA

O paradigma imperativo é aquele que expressa o código através de comandos ao computador. Uma outra característica marcante é a mutação de estado (alterar os valores dos objetos). Funciona como uma receita de bolo.

- você tem o controle total do que está sendo executado no programa e dita através de comandos, passo a passo, o que deve ser feito.

PROGRAMAÇÃO FUNCIONAL

O paradigma funcional costuma expressar o código principalmente através de funções e evita a mutação de estado. Você diz como deseja o bolo e ele será feito, portanto tende a ser mais expressivo.

- a função no paradigma funcional é expressada num sentido mais matemático.
- a ideia é fazer o que você quer com base na execução de operações - ou funções - nos objetos

EXEMPLO BOBO

- objetivo: fazer almoço
- na programação imperativa
tenho os ingredientes para fazer o almoço



- na programação funcional
tenho as opções para montar o almoço



HASKELL

funções

• HEAD

devolve o primeiro elemento da lista

```
let x = [1..10]
```

```
print (head x)
```

```
> 1
```

• TAIL

complementa a função head; devolve a lista usen o head (primeiro elemento)

```
let x = [1..10]
```

```
print (tail x)
```

```
> [2,3,4,5,6,7,8,9,10]
```

• LAST

devolve o ultimo elemento da lista

```
let x = [1..10]
```

```
print (last x)
```

```
> 10
```

• INIT

oposto da função tail; retorna a lista usen o ultimo elemento

```
let x = [1..10]
```

```
print (init x)
```

```
> [1,2,3,4,5,6,7,8,9]
```

• NULL

verifica se a lista está vazia

```
let x = [1..10]
```

```
print (null x)
```

```
> False
```

• REVERSE

converte a entrada na ordem inversa

```
let x = [1..50]
```

```
print (reverse x)
```

```
> [50, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

• LENGTH

devolve o tamanho da lista

```
let x = [1..50]
```

```
print (length x)
```

```
> 50
```

• TAKE

usado para criar uma substring a partir de outra string.

```
let x = [1..50]
```

```
print (take 5 (x))
```

```
> [1, 2, 3, 4, 5]
```

• DROP

oposto da função take, gera uma substring

```
let x = [1..50]
```

```
print (drop 5 (x))
```

```
> [6, 7, 8, 9, 50]
```

• MAXIMUM

encontra o elemento com valor máximo

```
let x = [1, 45, 565, 1245, 02, 2]
```

```
print (maximum x)
```

```
> 1245
```

• MINIMUM

encontra o menor valor

let $x = [1, 45, 565, 1245, 02, 2]$

print (minimum x)

> 1

• SUM

soma de todos os elementos da lista

let $x = [1..5]$

print (vsum x)

> 15

• PRODUCT

multiplica todos os elementos da lista

let $x = [1..5]$

print (product x)

> 120

• ELEM

verifica se a lista contém um elemento fornecido

let $x = [1, 45, 565, 1245]$

print (elem 786 (x))

> false

• length :: [a] → Int recebe qualquer lista e retorna um int

length [] = 0 caso base (lista vazia)

length (x; xs) = 1 + length xs

↑
primeiro elemento
restante da lista

RESOLUÇÃO 99 QUESTIONS

problem 1

1 2 3 4

myLast = head. reverse

problem 2

myButLast = head. tail . reverse

problem 3

elementAt :: [a] → Int → a

elementAt (x : xs) i = x só usou o x

elementAt [] _ = erro lista vazia com qualquer index

elementAt (_ : xs) k qualquer elementos com index k.

| k < 1 = erro

| otherwise = elementAt xs (k - 1)

problem 4

myLength :: [a] → Int

myLength [] = 0

myLength (x : xs) = 1 + myLength xs

problem 5

reverse :: [a] → [a]

reverse [] = []

concatenação de listas

reverse (x : xs) = reverse xs ++ [x]

problem 6

cópia de a

palindrome :: (Eq a) => [a] → Bool

palindrome xs = xs == (reverse xs)

RESOLUÇÃO LISTA DE HASKELL

1. $\text{vsubLista} :: \text{Int} \rightarrow [\alpha] \rightarrow [\alpha]$

$\text{vsubLista} = \text{take } i (l)$

2. $\text{vsubLista} :: \text{Int} \rightarrow [\alpha] \rightarrow [\alpha]$

$\text{vsubLista} = \text{drop } i (l)$

3. $\text{maiores} :: \text{Int} \rightarrow [\alpha] \rightarrow [\alpha]$

$\text{maiores} [] = []$

$\text{maiores} (x:xs) = \text{if } x > i \text{ then } x : \text{maiores } xs \text{ else maiores } xs$

4. $\text{produto} :: [\alpha] \rightarrow \text{Int}$

$\text{produto} (x:xs) = x * \text{produto } xs$

ou

$\text{product} (\text{lista})$

5. $\text{maximum} (\text{lista})$