HASKELL

Notas de Aula de Aspectos
de Linguagem de Programação

Professor:
Rodrigo Augusto Barros Pereira Dias

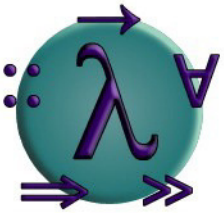
Versão 5.5
2007-2

S U M Á R I O

1 – Introdução à Programação Funcional

Na década de 1980, um comitê foi organizado com o objetivo de construir uma linguagem funcional de programação padronizada com uma semântica não-rígida. Haskell (http://www.haskell.org), em homenagem ao lógico *Haskell Curry*, foi o resultado dessas deliberações.

Haskell Curry (*1900-1982*)



Logotipo dalinguagem Haskell

O último padrão semi-oficial desta linguaguem é *Haskell 98*, destinado a especificar uma versão mínima e portável da linguagem para o ensino e como base para futuras extensões. A linguagem continua a evoluir rapidamente, com Hugs e GHC representando os padrões atuais.

Características interessantes do Haskell incluem o suporte a funções recursivas e tipos de dados, casamento de padrões, list comprehensions e guard statements. A combinação destas características pode fazer com que a construção de funções que seriam complexas em uma linguagem procedimental de programação tornem-se uma tarefa quase trivial em Haskell.

A linguagem é, em 2002, a linguagem funcional sobre a qual mais pesquisa está sendo realizada. Muitas variantes tem sido desenvolvidas: versões paralelizáveis do *MIT* e *Glasgow*, ambas chamadas *Parallel Haskell*, outras versões paralelas e distribuidas chamadas *Distributed Haskell* (anteriormente *Goffin*) e *Eden*, uma versão chamada *Eager Haskell* e várias versões orientadas a objetos: *Haskell++*, *O'Haskell* e *Mondrian*. A versão educacional do Haskell chamada *Gofer* foi desenvolvida por *Mark Jones*, ela é oferecida pelo *HUGS*, o *Haskell User's Gofer System*.

As seguintes implementações do Haskell estão totalmente (ou quase) de acordo com o padrão Haskell 98 e são distribuídas sob licenças open source ou free software:

Hugs - É um interpretador que oferece rápida utilizaçlão dos programas e razoável velocidade de execução. Dispõe de uma simples biblioteca gráfica. Hugs é ideal para pessoas que estão aprendendo os básicos de Haskell. É a mais portável e peso leve das implantações.

GHC - O *Glasgow Haskell Compiler* compila para código nativo de diferentes arquitecturas e pode também compilar para C. É provavelmente o compilador Haskell mais popular, e possui bibliotecas bastante úteis que somente trabalharão com ele.

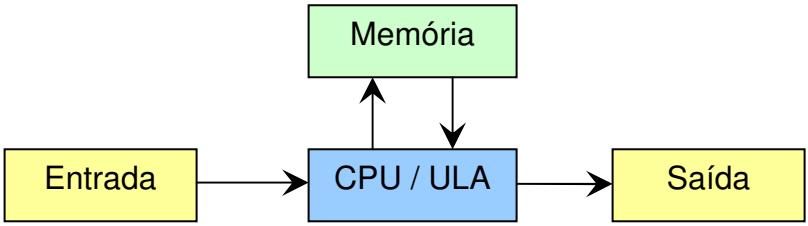
Nhc98 - Outro compilador de bytecodes, mas mais rápido que Hugs. Nhc98 foca na minimização do uso de memória, e é uma boa escolha para máquinas antigas e/ou lentas.

Helium - É um novo dialecto do Haskell, o foco é na facilidade de aprendizado. Atualmente carece de classes de tipo, tornando-o incompatível com muitos programas Haskell.

1.1. Principais Conceitos

Os seguintes conceitos são muitas vezes utilizados nos livros adotados e a sua compreensão é essencial na compreensão da forma de funcionamento de um computador e na compreensão do que é e de como funciona uma linguagem de programação.

1.1.1. Máquina de von Neumann: Abstração do computador moderno. É utilizada para ilustrar o funcionamento de um computador. É constituída por uma memória capaz de guardar informação devidamente endereçada; uma CPU que executa instruções de controle do estado da máquina; uma ULA que executa operações sobre os dados em memória e uma unidade de entrada/saída, para interagir com o exterior. Como na imagem a seguir:



A programação da unidade de controle da *máquina de von Neumann* pode apenas ser efetuada com um conjunto muito limitado e simples de instruções. Uma hipótese de um conjunto de instruções base para a *máquina de von Neumann* poderia ser:

- Soma e subtração de um endereço de memória na posição i, com o acumulador:
  - A:= A+M[i]; A:=A-M[i];
- Soma e subtração do valor endereçado pelo endereço de memória na posição i, com o acumulador:
  - A:= A+ [M[i]]; A:=A-[M[i]];

- Multiplicação e divisão do acumulador por 2:
  - A:= A*2; A:= A div 2;
- Mover o conteúdo do acumulador de e para memória:
  - A:=M[i]; M[i] :=A;
- Executar as instruções situadas a partir do endereço i de memória:
  - Goto i (PC := i)
- Se o valor do acumulador for maior ou igual à zero, executar as instrução situadas a partir do endereço de memória i:
  - if(A>=0) goto i

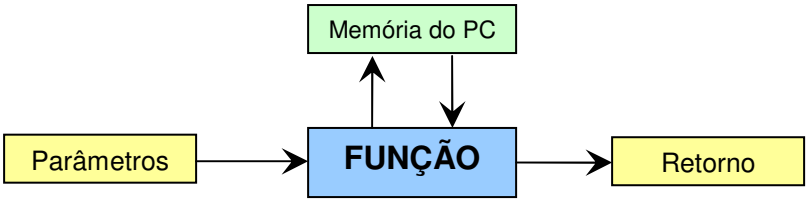
Todos os programas executados num computador têm de ser codificados num conjunto de instruções similares a estas. Os programas assim codificados dizem-se em linguagem máquina ou código máquina.

1.1.2. Linguagem de programação: Linguagens formais utilizadas na descrição de mecanismos abstratos. Têm como objetivo descrever e comunicar um processo computacional. Podem ser utilizadas para estudar os algoritmos e para defini-los de modo não ambíguo.

1.1.3. Compilador: Um compilador é um programa, capaz de ler um arquivo contendo um programa escrito numa linguagem de programação e gerar um programa em linguagem máquina.

1.1.4. Interpretador: Tal como o compilador, um interpretador analisa sintática e semanticamente uma dada linguagem de programação, no entanto, em vez de converter o programa para linguagem máquina, ele executa o programa que lhe é dado, passo a passo.

1.1.5. Funções: O conceito de função aqui tangido não é o original proveniente da matemática, mas sim o que se desenvolveu com as linguagens de programação moderna. Principalmente no paradigma funcional, uma função é um micro programa ultra especializado com entra e saídas bem definidos a fim de solucionar um problema. Se for necessário esquematizar o funcionamento de uma função teremos um diagrama muito semelhante à Máquina de von Neumann:



1.2. Tipologia das linguagens

1.2.1. Por níveis: Disposição hierárquica, segundo o seu nível, que denota um índice da facilidade do programador enunciar a solução dos seus problemas.

- *Linguagens de baixo nível:* O código máquina e a sua representação em Assembler.
- *Linguagens de alto nível:* algorítmicas, imperativas, prescritivas, procedurais, etc. Papel determinante do cálculo numérico. Exs: Fortran, Pascal e Simula.
- *Linguagens de muito alto nível:* Idealmente, não algorítmicas, declarativas, etc. Voltadas para o processamento simbólico. Exs: SQL, Haskell, ProLog, e LisP.

1.2.2. Por árvores genealógicas:

Desenha-se a rede das linguagens de programação destacando as suas ligações implícitas e explícitas. As principais árvores dividem-se pelos paradigmas da programação: imperativo, orientado à objetos, funcional e lógico.

1.2.3. Gerações

As linguagens agrupam-se de acordo com um processo de mutações e seleções tecnológicas num processo descontínuo, geralmente classificasse em 5 gerações:

- 1ª Geração: Linguagem binária, linguagens máquina e Assembler.
- 2ª Geração: Utilizando compiladores: Cobol, Basic, RPG, Fortran, Algol.
- 3ª Geração: Linguagens procedimentais: Pascal, Fortran 8X, C, Simula.
- 4ª Geração: Geradores de programas: ADA, ou linguagens de interrogação: SQL.
- 5ª Geração: Linguagens de especificação de problemas: LisP, ProLog.

1.2.4. Outras formas de classificação

- Classificação por domínios de aplicação:
  - Científicas: ALGOL, BASIC, FORTRAN;
  - Comerciais: COBOL.
- Estilo algorítmico:
  - Processamento de listas: LISP
  - Gerais (ou fins múltiplos): Algol68, Pascal, Simula67
  - Inteligência Artificial: ProLog
  - Ciências Sociais: SPSS.
  - Planejamento de espaços: ProLog
  - Programação de Sistemas: C

- Classificação por modos de processamento:
    - Interativa: o programador pode efetuar alterações ao programa enquanto este é executado. Exs: Basic, APL, LisP, ProLog.
    - Por lotes. Exs: Fortran, Cobol, Algol, Pascal.
- Classificação por categorias:
    - Fim geral: PL/1, Pascal, Basic, Fortran, C.
    - Fim específico: Cobol, LisP, ProLog.
- Classificação por número de utilizadores: Sobre o número de utilizadores que houve/há numa linguagem num momento, ou pela percentagem do código existente da linguagem.

1.3. Principais Paradigmas

1.3.1. Programação Imperativa: Linguagens que permitem descrever a resolução de um problema através de uma série de tarefas elementares que o computador pode compreender e executar. A seqüência de comandos define o procedimento a seguir e permite a mudança de estado de variáveis de atribuição. Seus principais problemas são não serem adequadas à computação em paralelo e a baixa produtividade na programação. Ex.: Fortran, Cobol, Pascal e C.

1.3.2. Programação Orientada à Objetos: O programa é organizado em função de objetos, que contêm não só as estruturas de dados mas também as funções que sobre eles agem. A comunicação entre objetos é feita através de mensagens. Os objetos são normalmente organizados numa rede podendo os objetos mais especializados herdar as propriedades dos objetos mais genéricos. Este paradigma está particularmente bem adaptado à construção de interfaces gráficas. Ex.: SmallTalk, Simula, C++ e o Java.

1.3.3. Programação Funcional: Descrevem o conhecimento de um problema através da declaração de funções. As funções são aplicadas recursivamente ou por composição e têm como resultado valores. A parte algorítmica e procedimental é suprimida: modela-se apenas as formulações matemáticas da computação. Ex.: LisP, APL e Haskell.

1.3.4. Programação Lógica: O problema é descrito em função de afirmações e de regras sobre os objetos. Procurou-se possibilitar a total supressão da parte algorítmica e procedimental: cabe ao interpretador encontrar os caminhos ou processos de resolução do problema. A programação em lógica é uma subdivisão do paradigma funcional. Programas e dados aparecem em conjunto. Ex.: ProLog.

1.4. Quais linguagens ensinar?

- *Linguagens Funcionais:* Haskell, SML, Scheme, Erlang, LisP, Java ($^{+}/_{-}$)
- *Linguagens Imperativas:* Fortran, C, C++, Visual C++, Cobol, Visual basic, Assembly, Java ($^{+}/_{-}$)

Destacam-se os seguintes pontos negativos com relação à tradicional abordagem de lecionar a arte de construção de algoritmos através do paradigma imperativo:

1. Os alunos têm uma considerável dificuldade em implementar imperativamente as especificações de soluções de problemas que são a eles passados, não tendo tanta dificuldade em especificar, em alto nível a solução de um problema;
2. As linguagens imperativas são sintaticamente extensas e semanticamente mal definidas, o que dificulta o seu aprendizado pelos não-iniciados;
3. Paradigma de *von Neumann* é um modelo muito distante dos modelos matemático aos quais os pupilos estão acostumados;
4. Necessidade de gerência automática de memória faz com que o ensino de estruturas de dados seja dificultado;
5. A fatia de tempo dedicada ao seu ensino de funções não é suficiente para que o aluno o absorva completamente, fazendo com que o ensino de disciplinas na área de *Engenharia de Software* fique demasiadamente abstrato.

1.4.1. Usos industriais de linguagens funcionais

- Intel (verificação de microprocessadores)
- Hewlett Packard & Ericsson (telecom)
- Carlstedt Res.&Tech. (escalonamento de tripulação)
- Legacys & Hafnium (ferramenta Y2K)
- Shop.com (comércio eletrônico)
- Motorola (geração de testes)
- Thompson (rastreamento por radar)
- O AutoCad foi programado em LISP

1.4.2. Muitas razões para usar Haskell:

1. Linguagem de propósitos gerais;
2. De muito alto nível;
3. Expressiva;
4. Concisa (sintaxe simples);
5. Muito boa na combinação de componentes;
6. Programador não gerencia memória (não há ponteiros)

A *única razão para não usar Haskell* é que programas em Haskell rodam mais lentamente do que programas correspondentes em linguagens imperativas. Bem, há algumas outras (como programação em tempo real, que é bem complicado em Haskell), mas são muito específicas.

A proposta de adotar uma linguagem funcional como primeira não é nova, uma das universidades pioneiras nesta abordagem é o MIT com a linguagem funcional Scheme, mas esta proposta está em consonância com o que é adotado em várias instituições estrangeiras de excelência, por exemplo, vários departamentos de computação no Reino Unido e Austrália adotam Haskell ou Miranda como primeira linguagem como também alguns departamentos americanos.

1.4.3. Áreas para programação em Haskell

1. Compilação;
2. Simulação (Teoria das Filas, Grafos, Pesq. Operacional...);
3. Programação Paralela e Distribuída;
4. Computação Gráfica;
5. Internet;
6. Interfaces Gráficas para Usuários;
7. Banco de Dados...

1.4.4. Como programar à la Haskell em C

Você não pode usar os seguintes recursos de C: Comandos de repetição; condicional (a não ser que o "then" sempre tenha um "else"); comandos de atribuição e ponteiros.

Você só pode usar: define's; funções; return's e operadores aritméticos e lógicos.

1.5. Programação Funcional

Uma função é uma maneira de computar um resultado a partir de seus argumentos:

f (x) = sen (x) / cos (x) ← (Função gerando número a partir de ângulo)

Um programa funcional computa sua saída como uma função da sua entrada.

1.5.1 Valores e Expressões

Um valor é um dado: 2, 4, 3.14159, "Rodrigo", [1,2,3] . . .
Uma expressão computa um valor: 2 + 2, 2 * pi * r . . .
Expressões combinam valores usando funções e operadores.

1.5.2. Operações

Operadores são sempre explícitos: b ^ 2 - 4 * a * c
Não podem ser escritos como: b2 - 4ac
Multiplicação (*) mantém a maior precedência que a subtração (-).

1.5.3. Definições e Tipos

Uma definição fornece um nome a um valor

```
b area :: Int a
area = 41 * 37 c
```

a - Tipos especificam que espécie de valor *area* é.
b - Nomes começam com uma **letra minúscula** e são feitos de letras e dígitos.
c - Uma expressão diz como o valor é computado.

1.5.4. Definições de Funções

Uma definição de função especifica como o resultado é computado a partir de seus argumentos:

```
area :: Int -> Int -> Int d
area a b e = a*b f
```

d - Tipos de funções especificam os tipos de seus argumentos e do seu resultado.
e - Os argumentos são nomes dados após o nome da função.
f - O corpo especifica como o resultado é computado.

* **Notação importante:** Os argumentos de uma função não precisam (e muitas vezes não podem) vir entre parênteses. Exemplo:

```
media :: Float -> Float -> Float
media x y = (x + y) / 2.0
```

Chamadas:	*Retorno:*	
media 2 3	2.5	
media (2+2) (3*3)	6.5	←Eles servem apenas para agrupar

Contraste o uso de parênteses em Haskell com o uso em C. No C, os parênteses são recursos sintáticos utilizados em definições e chamadas de funções, servindo apenas para separar tanto argumentos formais como reais. Em Haskell uma função (*Ex.: f (x,y) = x*) tem apenas um argumento, a saber, o par (x,y), um elemento do produto cartesiano dos domínios aos quais x e y pertencem.

Um programa funcional consiste, basicamente, de definições de funções (colocadas em um arquivo chamado script).

Funções simples são usadas para definir outras mais complexas que são, por sua vez, utilizadas para definir outras funções ainda mais complexas e assim sucessivamente.

Finalmente, definimos uma função para computar a saída do programa como um todo a partir de seus valores de entrada.

Charada:

Qual é o próximo elemento da seqüência a seguir:

2 – Introdução à Linguagem Haskell

2.1. Usando o intepretador Hugs

Agora mostraremos como utilizar o sistema Hugs que interpreta programas escritos em Haskell, cuja home-page oficial é: *http://www.haskell.org*. Nela, além do Hugs e outros interpretadores e compiladores, você encontrará tudo sobre Haskell.

2.1.1 Um primeiro programa em Haskell

Um script em Haskell deve ser um texto plano sem nenhuma formatação adicional. Dessa maneira a extenção do arquivo é irrelevante, apesar de a maioria utilizar o padrão ".hs" nos nomes dos scripts, isto é indiferente.

Atente também para identação em Haskell que deve ser feita com "*espaços*", e não com "*tabs*". Na maioria dos casos um tab é interpretado como um único caractere, gerando uma identação equivocada. Se você está em ambiente *Windows* e utilizando o "*Bloco de Notas*", aconselha-se que que a fonte utilizada seja uma fonte size-fixed, como courrier ou terminal, para evitar interpretações erradas de identação.

Começamos por fornecer um primeiro script em Haskell:

PrimeiroScript.hs

```
-- Função constante:
tamanho :: Int
tamanho = 12+13

-- Função para calcular o quadrado de um inteiro:
quadrado :: Int -> Int
quadrado n = n * n

-- Função para dobrar um inteiro:
dobro :: Int -> Int
dobro n = 2 * n

-- Exemplo usando tamanho, quadrado e dobro:
exemplo :: Int
exemplo = dobro (tamanho - quadrado (2*2))
```

Ou podemos fornecer o script no estilo "*literate programming*" (atente para o sufixo lhs no nome), onde as linhas de código tem um ">" na frente e o restante é comentário. Veja um exemplo na página seguinte:

PrimeiroScript.lhs

```
Função constante:
>       tamanho :: Int
>       tamanho = 12+13
Função calcula o quadrado de um inteiro:
>       quadrado :: Int -> Int
>       quadrado n = n * n
Função para dobrar um inteiro:
>       dobro :: Int -> Int
>       dobro n = 2 * n
Um exemplo usando tamanho, quadrado e dobro:
>       exemplo :: Int
>       exemplo = dobro (tamanho - quadrado (2*2))
Os comentários vem sem o ">" no começo.
O que é bom para pequenos scripts bem comentados.
```

2.1.2. Iniciando Hugs no Unix

No prompt do Unix, digite

hugs PrimeiroScript.hs <ENTER>

Aparecerão várias mensagens e finalmente o seguinte prompt:

Main>

O interpretador Hugs irá, então, avaliar as expressões digitadas no prompt, incluindo aquelas presentes no PrimeiroScript.hs.

2.1.3. Iniciando Hugs no Windows

Trivialmente como qualquer aplicativo Windows, o duplo clique no ícone do arquivo resolve o problema. Alternativamente pode-se abrir o arquivo utilizando os atalhos normais de "*menu -> abrir*" ou arrastar o arquivo para dentro da área do interpretador.

Observação importante: o interpretador *WinHugs* não interpreta arquivos localizados em drives de rede.

2.2. Avaliando expressões no Hugs

```
Main> dobro 32 - quadrado (tamanho - dobro 3)
-297
Main> dobro 320 - quadrado (tamanho - dobro 6)
471
Main> 30 - 2 + 4
32
Main> (*) 8 - 2
4
Main> div 13 2
6
```

2.2.1. Tabela de precedência

Caso não se deseje utilizar parênteses para definir a ordem de avalição de uma expressão, existe uma tabela de precedência da linguagem, em que os níveis mais altos tem preferência, e esta diminui ao descer a tabela. Mas lembre-se que sempre que possível de que *é aconselhável a utilização de parenteses* para explicitar a ordem de avaliação.

Nível	*Preferência à esquerda*	*Preferência à direita*
9	!!	.
8		^ ^^ **
7	* / div mod %	
6	+ -	
5		: ++
4	== /= < <= >= > elem notElem	
3		&& (e lógico)
2		|| (ou lógico)
1	>> >>=	=<<
0		$ $! seq

2.2.2. Operadores novos :

!! (exclamação dupla) – Recebe uma lista e um inteiro e retorna o número daquela posição na lista. A lista começa indexada em 0, assim, [1,2,3] !! 1, retornaria 2.
. (ponto) – Faz composição de funções. Note que a segunda função é ativada antes da primeira. Um exemplo: (abs**.**snd) (-1,-3) = 3
^ - Potência de inteiro. Exemplo: 2 ^ 3 = 8
^^ - Potência de ponto flutuante. Exemplo: 2.5 ^^ 2 = 6.25
****** - O expoente da potência pode ser fracionário, permitindo cálculo de raizes. Exemplo: 4 ** 0.5 = 2
div – Retorna o resultado de divisão inteira
mod - Retorna o resto da divisão inteira
% - Tira a simplificação da fração. Exemplo 3 % 9 = 1 % 3
>> - Omissão de saída e IO.
>>= - Redirecionamento de IO, neste caso, saída para arquivo.
=<< - Redirecionamento de IO, neste caso, entrada de um arquivo.
$ - Operador assosiativo direito infixo de funções (*f $ x = f x*). Útil em estilo de passagem contínua.
**$!** - Igual ao *$*, mas: (*f $! x = x ´seq´ f x*)

seq – Avalia o 1º argumento antes de retornar o segundo, usualmente para aumentar a performance e evitar avaliação preguisoça desnecessária. Força a avaliação da função.

2.2.3. Alguns comando do interpretador Hugs

No prompt você pode digitar os seguintes comandos. Note que eles sempre devem vir precedidos de ":" (dois pontos), caso contrário será considerado uma função.

:?	Help de todos os comandos
:l <nome_arq>	Carrega módulos de um arquivo específico
:l	Apaga todos os arquivos carregados, exceto o *prelude.hs*
:a <nome_arqs>	Lê módulos adicionais
:r	Repete último comando de "load" (*recompila...*)
:p <nome_arq>	Usa um arquivo de projeto
:e <nome_arq>	Edita um arquivo
:e	Edita o último módulo
:m <modulo>	Carrega módulo para expressões de avaliação
:t <expressao>	Imprime o tipo de uma expressão
:s <opcoes>	Configura as opções da linha de comando
:s	Help nas opções da linha de comando
:n [pat]	Lista os nomes no escopo atual
:i <nomes>	Descreve os objetos nomeados
:b <modulos>	Procura nomes definidos em <modulos>
:f <nome>	Edita o conteúdo do módulo para a definição do nome
:!command	Escapa para o *shell* do sistema operacional
:cd dir	Muda o diretório
:gc	Força *Garbage Colection* (limpa a memória)
:v	Imprime a versão do Hugs
:q	Sai do interpretador Hugs

2.3. Tipos básicos do Haskell

Temos o hábito de pensar em funções cujos argumentos e resultados são números. Em Haskell, usualmente trabalhamos com tipos de valores bem mais ricos. Vamos dar uma olhada em alguns dos tipos pré-definidos de Haskell, tais como: *inteiros, reais, caracteres, booleanos e agregadores*.

2.3.1. Números inteiros

O Haskell possui dois tipos para tratar os inteiros, um com precisão limitada e outro de precisão infinita (declarada dinâmicamente conforma a necessidade).

As três operações básicas da aritmética (*soma* (+), *subtração* () e *multiplicação* (*)), são implementadas e válidas para todos os tipos numéricos. Mas existem exceções que são válidas só para os inteiros:

div	div 7 2 ➔ 3	Divisão inteira
mod	mod 7 2 ➔ 1	Resto de divisão inteira
gcd	gcd 27 9 ➔ 9	MDC entre dois números
lcm	lcm 8 3 ➔ 24	MMC entre dois números
^	2 ^ 3 ➔ 8	Potência de inteiro

2.3.1.1 Int (números inteiros de precisão finita

Números inteiros entre -2^{31} (-2.147.483.648) até $2^{31} - 1$ (2.147.483.647) ➔ (..., -3, -2, -1, 0, 1, 2, 3, ... :: Int)

2.3.1.2. Integer (números inteiros de precisão infinita)

Números inteiros entre -? e +?. As operações válidas são as mesmas que para inteiros de precisão finita, mas permitindo maior elasticidade de valores, como se segue:

```
2 * 40000000000000000000000000000000→
80000000000000000000000000000000
```

2.3.2. Números reais.

Ou de ponto flutuante, assim como o inteiro, possui dois tipos para tratar os valores reais, um com precisão limitada e outro de precisão infinita. Também mantém as três operações básicas da aritmética, e suas próprias exceções que são válidas só para os reais, como segue na tabela:

/	9 / 2 ➔ 4.5	Divisão real
pi	pi ➔ 3.14159	Retorna a constante pi
sqrt	sqrt 16 ➔ 4.0	Calcula a raiz quadrada
exp	exp 8 ➔ 2980.96	Calcula exponencial
log	Log 15 ➔ 2.70805	Calcula logaritmo
sin	sin 15 ➔ 0.650288	Seno de um ângulo em radianos
cos	cos 30 ➔ 0.154251	Coseno de um ângulo em radianos
tan	tan 25 ➔ -0.133526	Tangente de um ângulo em radianos
^^	7.2 ^^ 3 ➔ 373.248	Potência de real
**	8 ** (1/3) ➔ 2.0	Expoente pode ser real

2.3.2.1. Float (números reais de precisão finita)

Números reais com seis dígitos significativos (+ ou -) entre 2^{15} (-32.768,0) até 2^{15} - 1 (32.767,0) ➔ (1.5, 0.425, 3.14159 :: Float)

2.3.2.2. Double (números reais de precisão infinita)

Mantendo a analogia com os inteiros, são válidas as mesmas operações do Float, mas com liberdade total de tamanho significativo.

2.3.3. Char (caracteres do teclado)

O tipo nativo do Haskell para representação de caracteres utiliza os 256 valores da tabela ASCII (variando de 0 a 255) para representar e controlar seus caracteres. Um Char deve ser expresso entre ‘’ (aspas simples ou *piclis*).

A seguir uma tabela com funções próprias para Char:

ord	retorna o valor ASCII do caracter
chr	retorna o caracter do valor ASCII
toUpper	converte de minúscula p/ maiúscula
toLower	converte maiúscula de p/ minúscula
isUpper	verifica se é maiúscula
isLower	verifica se é minúscula
isSpace	verifica se é caractere de espaço
isDigit	verifica se é dígito
isAlpha	verifica se é alfabético
isAlphaNum	verifica se é alfa-numérico
isPrint	verifica se é caractere “imprimível”
isControl	verifica se é caractere de controle
isAscii	verifica se o caractere tem um código ASCII válido

2.3.4. Listas (são expressas entre colchetes – [ ])

Lista de valores podem ser de vários tipos, sendo que uma lista pode conter ***N*** elementos do mesmo tipo, e vem sempre envolvidas por colchetes, neste exemplo são listas de inteiros:

```
[1,2,3], [2] :: [Int]
```

Estudaremos as listas mais profundamente em breve, por enquanto algumas exemplos de operações envolvendo listas:

Como você adicionaria 4 ao final da lista [1,2,3]?

```
[1,2,3] ++ [4] → [1,2,3,4]
```

Obs.: Note que não é 4 e sim [4], pois o “++” combina duas listas e 4 não é uma lista, é um inteiro.

Como você pegaria o primeiro elemento de uma lista?

```
head [1,2,3] → 1
```

E se tivesse de pegar o último elemento da lista?

```
last [1,2,3] → 3
```

2.3.5. Strings (são listas de caracteres - [Char])

É o tipo de um pedaço de texto, pode ser quaisquer caracteres envolvidos por aspas duplas (“ ”). O Char é um único caractere envovido por aspas simples (‘ ’).

```
“Hello!” :: String        ‘X’ :: Char
```

Operações válidas para listas, também valem para Strings:

"Hello" ++ " World" → "Hello World"

No exemplo anterior concatenamos duas Strings.

Mas Existem funções que são específicas para strings, veja:

show (2+2) → "4"

A função "*show*" converte um tipo qualuer em String. Assim como as listas, estudaremos as Strings em profundidade mais á frente. **Obs.**: Note que "2+2" não é igual a "4". "2+2" é uma string com três caracteres, "4" é uma string com um caractere. E elas não são o mesmo texto.

2.3.6. Tuplas (são expressas entre parênteses – ( ))

Servem para agrupar elementos de tipos diferentes, mas diferentemente das listas não podem ter um comprimento arbitrário, todos os elementos de uma tupla sempre devem casar com a definição da mesma, como por exemplo:

(Int, Char)	(65, 'A')	Uma tupla contendo um inteiro e um caracter

2.3.7. Bool (lógicos Booleanos)

Uma condição em Haskell (e como na maioria das linguagens) é ou verdadeira ou falsa. Em Haskell os valores que representam os booleanos são constantes e são representados por True (representando verdade) e False (representando não verdade).

Note que ambos são expressos com a primeira letra maiúscula e não são escritos entre aspas (pois não são String).

Tabela das expressões lógicas:

==	5 == 8 → False	Igualdade
/=	4 /= 1 → True	Diferença
>	2 > 3 → False	Maior
<	2 < 3 → True	Menor
>=	3 >= 5 → False	Maior ou igual
<=	2 <= 3 → True	Menor ou igual
not	not False → True	Negação (inverte um Bool)
||	Flase || True → True	Ou lógico
&&	True && False → False	E lógico

2.3.7.1. Funções retornando booleanos

Como usual, funções podem retornar resultados booleanos:

```
emOrdem :: Int -> Int -> Int -> Bool
emOrdem x y z = (x <= y) && (y <= z)
```

Esta função verifica se os valores x, y e z por ela recebidos foram informados em ordem crescente, considerando empates.

2.3.7.2. Definições por Casos

Freqüentemente, programas precisam tomar decisões e computar diferentes resultados em diferentes casos. Como por exemplo: Defina uma função "*maior*" para retornar o maior entre os seus dois argumentos x e y:

- Se x <= y então maior x y deve ser y
- Se x > y então maior x y deve ser x

```
maior :: Int -> Int -> Int
maior x y | x <= y = y
maior x y | x >[a] y  = x
```

a - Uma guarda: uma expressão do tipo Bool, um teste lógico. Se a guarda retirnar True a equação se aplica.

A definição pode ser otimizada se os casamentos para as tags forem os mesmos, evitando-se de se reescrevê-los, basta que se mantenham os "pipes" das guardas alinhados. Exemplo:

```
max :: Int -> Int -> Int
max x y| x <= y = y
       | x > y  = x
```

Seria escrita matematicamente como:

$$\max(x,y) = \begin{cases} y, & x <= y \\ x, & x > y \end{cases}$$

Uma outra otimização seria a utilização do operando *otherwise*, que equivale a "em outra condição":

```
max :: Int -> Int -> Int
max x y| x <= y = y
       | otherwise = x
```

Atenção: O operando "*otherwise*" serve como opção padrão caso a avaliação da condição em questãonão se enchaixe em nenhum dos casos anteriores, por isso ela deve ser a última a ser definida.

Charada :

Você se encontra do lado de fora de uma sala **totalmente** fechada com três interruptores eletrécicos á sua frente. Como você deve proceder para descobrir qual dos três interruptores é o que acende a luz do interior da sala, sabendo que ao abrir a porta da sala para verificar o estado da lâmpada não pode mais sair da sala e/ou modificar o estado dos interruptores.

Dado importante: a distância do teto desta sala para o solo é de 1 metro e 80 centímetros.

3 – Avaliações de Expressões Lógicas e Aritméticas

3.1. Avaliação com Guardas

Para avaliar uma chamada de função, avalie cada guarda, uma por vez, de cima para baixo, até que uma seja verdadeira, então, troque a chamada com uma cópia do corpo da função que segue a guarda verdadeira.

```
max 4 2        ??      4 <= 2      False
   ↓           ??      4 > 2       True
   4
```

```
max :: Int -> Int -> Int
max x y | x <= y = y
        | x > y  = x
```

A função max está correta?

Programação é um processo muito sujeito a erros; programas, raramente, estão corretos "de primeira". Uma boa parte do custo de desenvolvimento de software é oriunda do processo de busca e correção de erros.

É essencial que software seja testado: tente-o em um grande número de entradas e verifique se as saídas estão corretas.

Escolhendo dados de teste:

Dados de teste devem ser escolhidos cuidadosamente, incluindo casos "difíceis" que podem induzir a fracassos.

A função max deveria ser testada, no mínimo, com x<y, x == y, x >y e possíveis combinações de argumentos positivos e negativos;

Escolha exemplos de teste suficientes de tal forma que todos os casos em seu programa sejam utilizados pelo menos uma vez.

Especificações

O que queremos dizer com "max está correto"?

Uma especificação formula propriedades que esperamos que max satisfaça:

Propriedade: x <= max x y ou y <= max x y

Mas, porque formular especificações? Elas nos ajudam a deixar mais claro o que max tem que fazer, podendo ajudar em testes, permitindo-nos provar que programas estão corretos.

Especificações e Testes

Podemos definir uma função para checar se propriedades são válidas:

```
prop_Max :: Int -> Int -> Bool
prop_Max x y =  x <= max x y && y <= max x y
```

Se prop_Max sempre retorna True, então a especificação é satisfeita. Podemos testar max em várias entradas sem precisar inspecionar os resultados à mão.

Da definição de max:

```
x <= y → max x y = y
x > y  → max x y = x
```

Teorema: x <= max x y

Prova: Considere dois casos:

Caso x <= y: y = max x y, logo x <= max x y

Caso x > y: max x y = x e x <= x,
logo x <= max x y

Métodos Formais

Provas são custosas e também sujeitas a erros, mas podem garantir correção, logo, testar abrangentemente é o método mais comum atualmente.

Clientes de software críticos de segurança demandam provas atualmente;

Provas de correção serão cada vez mais importantes graças a ferramentas automáticas para provas e demanda por software de qualidade. Assim, defina:

abs x para retornar o valor absoluto de x (e.g. abs 2 = 2, abs (-3) = 3)
sign x para retornar 1 se x for positivo e -1 se x for negativo.

Enuncie (e prove) uma propriedade relacionando abs e sign.

```
abs :: Int -> Int
abs x | x <= 0 = -x
      | x > 0  = x

sign :: Int -> Int
sign x | x < 0  = -1
       | x > 0  = 1
       | x == 0 = 0[a]
```

a - Você considerou este caso? Ele pode ser também escrito como sign 0 = 0

Propriedade: x == sign x * abs x

3.2. Comandos

Um comando para escrever "Hello!" em *meuarquivo:*

```
writeFile "meuarquivo" "Hello!" :: IO ()
        O tipo de comandos que não produzem valores
readFile "meuarquivo" :: IO String
        O tipo de comando que produz uma string
```

Se *meuarquivo* contém "Hello!", readFile "meuarquivo" é igual a "Hello!"? *NÃO!*

O resultado de uma função depende apenas de seus *argumentos*; "Hello!" não pode ser computado a partir de "meuarquivo". O readFile é um comando para ler um arquivo e "Hello !" é um pedaço de texto constante. O efeito de um comando pode ser diferente, dependendo de *quando* for executado.

3.2.1. Combinando comandos

```
do      conteudo <- readFile "meu arquivo"
        writeFile "meuoutroarquivo" conteudo
```

Notas:

- A indentação neste caso é obrigatória
- O *do* combina dois ou mais comandos em seqüência
- O conteúdo é o nome a string produzida
- O readFile é o comando que produz uma string. Tipo: IO String
- O writeFile escreve o conteúdo da string, isto é, "Hello!" para *meuoutroarquivo*

3.3. Funções

```
dobro :: Int -> Int
dobro x = x + x
```

dobro 2 → 4 *(é a chamada de função)*
dobro *(sem argumentos - é um valor de função)*

3.4. Composição de Funções

Composição de funções, nada mais é que um operador de funções.

```
quadruplo :: Int -> Int
quadruplo = dobro . dobro
```

quadruplo 2 → (dobro . dobro) 2
→ dobro (dobro 2)
→ dobro 4
→ 8

3.5. A função map

A função map, mapeia para cada elemento da lista o que outra função faria para um elemento isoladamente, respeitando o tipo da lista é claro. É uma função com uma função como argumento e como resultado.

```
dobros :: [Int] -> [Int]
dobros = map dobro
```

dobros [1,2,3] → [dobro 1, dobro 2, dobro 3]
→ [2, 4, 6]

3.6. Funções de Alta Ordem

A habilidade de computar funções é uma das forças de Haskell, grandes partes de um programa podem ser automaticamente computadas pelo sistema ao invés de programadas à mão. Este é um tópico um pouco mais avançado ao qual retornaremos várias vezes.

3.7. Um Exemplo Juntando Tudo (*estudo de caso Linux*)

Vamos tentar montar um enviador amigável de email.

Defina um comando para enviar email que procura pelo endereço correto automaticamente. Para isto, devemos armazenar endereços em um arquivo, pois é fácil de modificar e vários usuários podem compartilhar o programa.

Arquivo: endereços

Rodrigo Dias	rodrigo.dias@esatcio.br
Juarez Muylaert	jamf@estacio.br
Bruno Bazzanella	bazzanella@estacio.br
Sandra Mariano	srhm@estacio.br

Podemos reutilizar alguns componentes (de SO's sérios, baseados no Unix), como por exemplo:

grep	para procurar um endereço
emacs	para editar a mensagem
mail	para enviar a mensagem

Nosso plano agora é enviar um email para o Juarez. Assim teremos:

- *grep Juarez enderecos > recipiente*
  Produz "Juarez Muylaert jamf@estacio.br" no arquivo de nome recipiente
- *readFile "recipiente" : o endereço é a última "palavra"*
- *emacs msg*
  Cria o arquivo msg com a mensagem
- *mail endereco < msg*
  Envia o conteúdo do arquivo msg para o endereço

  Mas como podemos rodar outro programa?
- system "emacs msg" :: IO Int

  Um comando que executa uma string como um comando Shell, o resultado produzido é um código do tipo "exit".

E como extrair o endereço eletrônico? Reutilizando uma função padrão de Haskell:

- words :: String -> [String]

Assim, teremos:

```
words "Juarez Muylaert   jamf@estacio.br" → ["Juarez", "Muylaert",
                                              "jamf@estacio.br"]
```

No fim, juntando tudo teremos:

```
email :: String -> IO Int
email nome =
    do  system ( "grep " ++ nome ++ "endereco>recipiente")
        recipiente <- readFile "recipiente"
        system ("emacs msg")
        system ("mail" ++ last (words recipiente) ++ " <msg")
```

3.8. Definições Revisitadas

Seja a definição

```
dobro :: Int -> Int
dobro x = 2 * x
```

Ela faz uma assertiva verdadeira sobre a função definida: seja x o inteiro que for, dobro x e 2 * x são iguais, ou seja, fornece uma maneira de computar chamadas da função. Mas, dada a definição:

```
x :: Int
x*x = 4
```

x é igual a 2? *NÃO!*

Esta não é uma definição válida em Haskell. Faz 1 assertiva verdadeira sobre x, mas não fornece uma maneira para computar x.

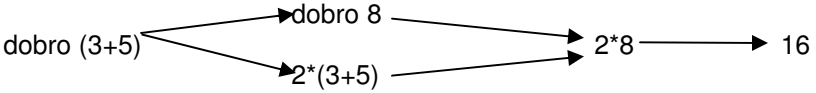
3.9. Computando com Definições

Uma chamada de função é computada pela troca da chamada com uma cópia do corpo com os nomes dos argumentos trocados pelos argumentos reais (isto é, os utilizados na chamada). Como segue em exemplo:

dobro 8 → 2 * 8 → 16	dobro :: Int -> Int dobro x = 2 * x

3.10. Ordem de Avaliação

Pode haver mais do que um modo de avaliar uma expressão



Você pode usar qualquer ordem de avaliação; todas fornecem o mesmo resultado.

3.11. Observações sobre Operadores e Funções

- Operadores infixos podem ser escritos antes dos seus argumentos com a utilização de parênteses como em:

  (+) :: Int -> Int -> Int

  de tal forma que

  (+) 2 3 = 2 + 3

  Esta conversão é necessária mais tarde em funções de alta ordem.

- Podemos converter funções em operadores com a utilização do acento grave ` (crase), como em:

  2 `max` 3 = max 2 3

  para aquelas funções binárias, ou seja, com dois argumentos.

- Operadores do tipo "Faça você mesmo" podem ser criados com combinações dos seguintes símbolos:

  ! # $ % & * + . / < = > ? \ ^ | : - ~

- Um operador não pode começar por ':'

- Algumas combinações de símbolos tabém são reservadas:

  .., ::, =>, =, @, \, |, ^, <- e ->

- Você pode declarar a associatividade e precedência de seus novos operadores (veja Prelude.hs e descubra como!).

- Aplicações de funções são dadas escrevendo o nome da função na frente de seu(s) argumento(s):

  f v1 v2 ... vn

  Isto tem maior precedência do que qualquer outro operador.

  Portanto, f n+1 é interpretada como f n mais 1, ou seja, f (n) +1, e não f aplicada a (n+1), f (n+1).

Na dúvida, use parênteses envolvendo cada argumento da função.

- De modo similar, como '-' é tanto um operador prefixo como infixo, há escopo para confusão.

  f -12 será interpretada como 12 subtraído de f ao invés de f aplicada a -12; a solução é colocar parênteses:

  f (-12)

3.12. Exemplo de Criação de Operadores : Ou Exclusivo

O ou explusivo é aquele em que se tem verdade como resposta apenas quando um dos operados é verdade e o outro falso, caso ambos sejam verdadeiro ou ambos falsos o resultado é falso. Assim, fica como exemplo a criação do operando **&|**, o & vem do "e lógico" (&&) e o | vem do "ou lógico" (||).

```
(&|) :: Bool -> Bool -> Bool
(&|) x y | x == y = False
         | otherwise = True
```

Charada :

Indiana Jones precisa de sua ajuda.

Ele se encontra em uma caverna e percebeu que pode desarmar uma armadilha se explodir uma dinamite, mas ele precisa de exatos 45 segundos de vantagem antes da explosão (ou a armadilha pega ele).

O problema é que todos os 3 pavios que ele possui foram projetados para queimar em exatos **30 segundos**. Devido a sua experiência com explosivos e armadilhas Indy sabe que não pode apenas cortar um pavio no meio e emendar em outro, pois os pavios queimam de maneira irregular (num pavio de 15cm por exemplo pode-se queimar 14 cm em 10 segundos e demorar os outros 20 segundos queimando o último centímetro).

Como nosso herói conseguirá garantir a confecção de um pavio que queime em exatos 45 segundos?

Dado importante: Indiana Jones não usa relógio de pulso ou de outra espécie quaquer e não há nenhum relógio de parede na caverna.

Anotações :

4 – Recursividade

Problema: defina fat :: Int -> Int
fat n = 1 * 2 * ... * n
O que podemos fazer se já soubermos o valor de fat (n-1)?
Então fat n = 1 * 2 * ... * (n-1) * n

= fat (n-1) * n

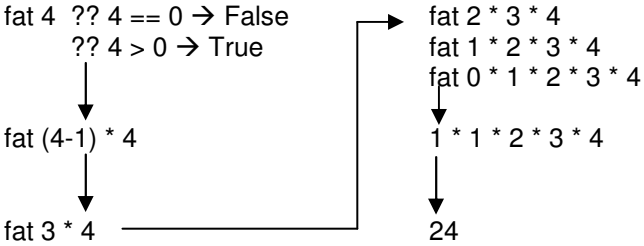
Uma Tabela de Fatoriais

n	fat n	
0	1	(É sabido que fat 0 = 1)
1	1	(Logo fat 1 = 1 * 1 = fat 0 * 1)
2	2	(Logo fat 2 = 1 * 2 = fat 1 * 2)
3	6	(Logo fat 3 = 2 * 3 = fat 2 * 3)
4	24	. . .

Uma Definição Recursiva de Fatorial

```
fat :: Int -> Int
fat 0          = 1              (Caso base)
fat n | n > 0 = fat (n-1) * n   (Caso recursivo)
```

4.1. Avaliando Fatoriais



4.2. Recursão Primitiva

Defina: f n em termos de f (n-1), para n > 0
f 0 separadamente

Pergunta mais importante: E se eu já souber o valor de f (n-1), posso computar f n a partir dele? *Sim.*

Defina uma função potencia de tal forma que potencia x n = x*...*x (n vezes)

Naturalmente, potencia x n == x ^ n, mas você deve definí-la sem utilizar ^. Veremos na página seguinte:

```
potencia :: Int -> Int
potencia x 0  = 1                    (Não esqueça o caso base)
potencia x n | n > 0 = potencia x (n-1) * x
                       (Dado que isto é igual a (x * x * ... * x) * x
                                                 n-1 vezes
```

4.3. Recursão Geral

E se soubermos os valores de f x para todo x menor do que n? Podemos computar f n a partir deles? Pro exemplo:

x ^ (2 * n) == (x * x) ^ n
x ^ (2 * n + 1) == (x * x) ^ n * x

```
pot2 :: Int -> Int -> Int
pot2 x 0 = 1                -- Caso base ainda necessário
pot2 x n  | n `mod` 2 == 0 = pot2 (x*x) (n `div` 2)   -- 2 casos recursivos
          | n `mod` 2 == 1 = pot2 (x*x) (n `div`2) * x
```

Comparando as versões

Primeira Versão	Segunda Versão
potencia 3 5 potencia 3 4 * 3 potencia 3 3 * 3 * 3 potencia 3 2 * 3 * 3 * 3 potencia 3 1 * 3 * 3 * 3 * 3 potencia 3 0 * 3 * 3 * 3 * 3 * 3 1 * 3 * 3 * 3 * 3 * 3 243	potencia 3 5 potencia 9 2 * 3 potencia 81 1 * 3 potencia 81 0 * 81 * 3 1 * 81 * 3 243
Seis chamadas Cinco multiplicações	Quatro chamadas Três multiplicações

4.4. Agora um exemplo mais difícil – Número Primo

Defina primo :: Int -> Bool de tal forma que primo n é True se n é um número primo. E se soubermos se (n-1) é primo? E se soubermos se cada número menor é primo?

4.4.1. Generalize o Problema
n é primo *significa* Nenhum k na faixa 2 <= k < n divide n.
Generalize: troque 2 por uma label (uma tag) . . .
Defina *fatora m n == True* se algum k na faixa m<=k<n divide n.
Logo primo n = not (fatora 2 n)
(not x é True se x é False e vice-versa)

4.4.2. Decomposição Recursiva

Problema: Algum k na faixa m<=k<n divide n?

E se soubermos se qualquer k em uma faixa menor divide n?

Algum k na faixa m<=k<n divide n se m divide n, ou algum k na faixa m+1<=k<n divide n.

4.4.3. Solução Recursiva

```
divide :: Int -> Int -> Bool
divide m n = n `mod` m == 0

fatora :: Int -> Int -> Bool
fatora m n | m == n = False
           | m < n = divide m n || fatora (m+1) n  (Não há k na faixa n<=k<n)
                                                   (x || y é True se x é True or y é True)
```

O que está ficando menor?

A faixa m<=k<n contém n-m elementos. Chame isto de tamanho do problema.

```
fatora m n | m == n = False      -- Caso base: n-m == 0
           | m < n = divide m n || fatora (m+1) n
                                 -- Recursão: n-(m+1) == (n-m)-1)
```

O tamanho do problema fica menor em cada chamada, até que torne-se zero e a recursão termina.

4.5. Lições

A recursão permite-nos decompor um problema em subproblemas menores do mesmo tipo.

O problema mais geral pode ser mais fácil de ser resolvido recursivamente porque as chamadas recursivas podem fazer mais coisas.

Para garantir terminação defina um tamanho de problema que deve ser maior do que zero nos casos recursivos e diminua de uma unidade, pelo menos, em cada chamada recursiva.

Problema :

Crie uma função (ou funções) que dado um número inteiro com uma quantidade qualquer de dígitos, retorne este número com seus dígitos invertidos.

Note que a função deve receber e retornar um inteiro. Desta maneira a solução seguinte não é válida:

```
inverte :: Integer -> String
inverte n = reverse (show n)
```

Pois o retorno seria uma string contendo os caracteres dos dígitos invertidos. Da mesma maneira **não se deve** transformar o número numa lista de dígitos.

5 – Tuplas & Listas

5.1. Tuplas

Uma tupla é um valor composto por vários outros valores chamados seus componentes. Uma tupla é defenida listando-se os elementos que a compoem emtre parênteses e separados por vírgulas. Logicamente não existe tupla de um só elemento. Um tipo tupla especifica o tipo de cada componente. Exemplos:

Abstração	Formato	Tipo	Exemplo
Um ponto no plano	(x,y)	(Int, Int)	(1,2)
Um ponto no espaço	(x,y,z)	(Float, Float)	(1.5,2.25)
Uma compra	(“Majorica”,15)	(String, Int)	(“Majorica”,15)
A tupla nula	( )	( )	( )

5.1.1. Definições de Tipos

Podemos nomear tipos com uma definição type:

```
type Compra = (String, Int)
```

Todos os tipos começam com maiúscula. A partir de agora, Compra e (String, Int) são intercambiáveis, são sinônimos. (*Na verdade Compra passa a ser um apelido para a tupla (String,Int)*)

```
majorica :: Compra
majorica = (“Majorica”, 15)
```

5.1.2. Casamento de Padrões

Funções sobre tuplas podem ser definidas por casamento de padrões. Exemplos:

```
nome :: Compra -> String
nome (s,i) = s  --Um padrão que tem que casar com o argumento

preco :: Compra -> Int
preco (s,i) = i
```

5.1.3. Funções de Seleção Padrão para Duplas

```
fst (x,y) = x          (retorna o 1º valor da dupla)
snd (x,y) = y          (retorna o 2º valor da dupla)
```

Mas não podemos definir:

```
select 1 (x,y) = x     (Qual seria o tipo do resultado?)
select 2 (x,y) = y
```

O tipo do resultado de uma função não pode depender dos valores dos argumentos.

5.2. Listas

Uma lista é também um valor composto por outros valores chamados elementos. Exemplos:

- [1,2,3] :: [Int]
- [True, False, True] :: [Bool]
- [ ] :: [Float]
- [(“Majorica”, 15)] :: [Compra]

5.2.1. Tuplas vs. Listas

Uma tupla consiste em um número fixo de componentes de vários tipos, é útil, mas não há muito a saber.

Uma lista consiste em um número arbitrário de elementos todos do mesmo tipo, a lista aparece em diversos contextos. São apoiadas por um rico conjunto de funções padrão e construções.

Listas podem até simplificar programas! Sempre que houver uma seqüência de valores, considere utilizar uma lista.

5.2.2. Listas para Contagem

Freqüentemente precisamos contar:

[1..10] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] *(seqüência contínua)*

[1,3..10] = [1, 3, 5, 7, 9] *(P.A.)*

Exemplo:

```
fat :: Int -> Int
fat n = product [1..n]
--Função padrão para multiplicar todos os elementos de uma lista
```

5.3. Lists Comprehensions

Freqüentemente desejamos fazer a mesma coisa com cada elemento de uma lista. Como no exemplo:

```
dobros :: [Int] -> [Int]
dobros xs =  [ 2 * x | x <- xs]        Para cada elemento x da lista xs
          Adicione 2 * x a lista sendo produzida
dobros [1..3] → [2,4,6]
```

Como somar quadrados usando listas: considere uma lista de comprimento n:

[1, 2, 3, ..., n]

[1, 4, 9, ..., n^2]

1 + 4 + 9 + ... + n^2

```
somaqd :: Int -> Int
somaqd n = sum [i ^2 | i <- [1..n] ]
           Soma todos os elementos de uma lista
```

5.3.1. Filtrando Listas

Uma "list comprehension" pode incluir uma condição que os elementos devem satisfazer. Exemplo:

```
fatores :: Int -> [Int]
fatores n = [ i | i <- [2..n], n `mod` i == 0]
```

Inclui apenas aqueles *i´s* que passam este teste, assim:

fatores 12 [2, 3, 4, 6, 12]

Usando a função fatores, podemos contar o número de fatores...

```
numFatores :: Int -> Int
numFatores n = length (fatores n)
```

*lenght retorna o número de elementos em uma lista

E daí fica fácil testar se um número é primo:

```
ehPrimo :: Int -> Bool
ehPrimo n = fatores n == [n]
```

OU

```
ehPrimo2 :: Int -> Bool
ehPrimo2 n | numFatores n == 1 = True
           | numFatores n > 1 = False
```

5.4. Projetando "List Comprehensions"

Pergunte a você mesmo: Preciso processar uma seqüência de elementos, um por vez? Se a resposta for afirmativa, use uma "list comprehension" !

Comece por escrever um arcabouço com lacunas a serem preenchidas:

1 [| <- ,]

Pergunte a você mesmo: Qual é a lista cujos elementos eu quero processar? Escreva-a na lacuna após a seta à esquerda:

2 [| <- [2..n],]

Pergunte a você mesmo: Como chamarei cada elemento? Preencha com o nome à esquerda da seta:

3 [| i <- [2..n],]

Pergunte a você mesmo: Quero processar cada elemento ou fazer uma seleção a partir deles? Escreva uma vírgula e uma condição ao final da "list comprehension":

4 [| i <- [2..n], n `mod` i == 0]

Pergunte a você mesmo: Quais valores eu quero no resultado? Escreva a expressão antes da barra:

5 [i | i <- [2..n], n `mod` i == 0]

5.5. Listas vs. Recursão

Mesmo problemas que não mencionem listas podem, freqüentemente, ser resolvidos facilmente pelo uso de listas. As construções para listas de Haskell muitas vezes oferecem uma alternativa mais simples para a recursão, mas nem sempre, já a recursão é uma ferramenta poderosa e geral...

Exemplo: *Banco de Dados de Bibliotecas*

Automatizar o arquivo da bibliotecária

Usuário: Rodrigo Dias Empréstimo: The Turning Point	Usuário: Juarez Muylaert Empréstimo : C Completo e Total

Qual é a informação em um cartão de empréstimo?

Nome do usuário : Rodrigo Dias

Título do livro : The Turning Point

```
type Usuario = String
type Livro = String
type Cartao = (Usuario, Livro)
```

Representando o Arquivo de Cartões de Empréstimo : Qual é o conteúdo do arquivo de cartões? Uma seqüência de cartões!

```
Type BD = [Cartao]
arquivo :: BD
arquivo = [("Rodrigo Dias", "The Turning Point"), ("Juarez Muylaert", "C Completo e Total")]
```

Consultando o B.D.: Que informações desejamos extrair?

```
livros :: BD -> Usuario -> [Livro]
usuarios :: BD -> Livro -> [Usuario]
emprestado :: BD -> Livro -> Bool
totEmprestado :: BD -> Usuario -> Int
```

Quais livros peguei emprestado?

```
livros arq "Juarez Muylaert" = ["C Completo e Total"]
livros arquivo "Sandra Mariano" = []    -- Nenhum livro emprestado

livros :: BD -> Usuario -> [Livro]
livros bd pessoa = [ livro | (usuario, livro) <- bd, usuario == pessoa]
```

E se fossem **quantos** livros peguei emprestado?

```
totEmprestado :: BD -> Usuario -> Int
totEmprestado bd pessoa = length (livros bd pessoa)
```

Atualizando o Banco de Dados: Fazer e retornar empréstimos muda o conteúdo do banco de dados. Estas funções usam a informação atual no banco de dados para computar a nova informação no banco de dados. Veja o exemplo na página seguinte:

```
fazEmprestimo :: BD -> Usuario -> Livro -> BD
fazEmprestimo bd usuario livro = [(usuario,livro)] ++ bd

retorneEmprestimo :: BD -> Livro -> BD
retorneEmprestimo bd livro = [ (usuario, livro´) | (usuario, livro´) <- bd, livro´ /= livro ]
```

5.5.1. O Papel das Listas

Listas oferecem um modo natural de computar informações do mundo real, afinal, seqüências abundam!

Listas são fáceis de manipular graças ao suporte de Haskell: uma boa primeira escolha de estrutura de dados.

As listas nem sempre são eficientes: haverão candidatos para troca por estruturas mais rápidas mais tarde...

5.6. Algumas Funções Padrão para Listas

Haskell fornece muitas funções para manipular listas. Vamos dar uma olhada em algumas.

Mas, primeiro, qual é o tipo de length? (Lembrete: length fornece o total de elementos de uma lista).

```
length :: [Int] -> Int
length :: [Float] -> Int
length [Cartao] -> Int
```

length tem muitos tipos!
É polimórfica.
É por isso que é útil.

Para qualquer tipo t, length :: [t] -> Int

Para estas aqui também vale o polimorfismo:

sum :: [X] -> X	product :: [X] -> X
Retorna a soma de uma lista	Retorna o produto de uma lista

5.6.1. Pegando e Largando

- take n xs os primeiros n elementos de xs
- drop n xs todos menos os primeiros n elementos

Exemplos:

- take 3 (fatores 12) [2,3,4]
- drop 3 (fatores 12) [6, 12]

take e drop têm o mesmo tipo. Qual é?

take, drop :: Int -> [a] -> [a]

"a" tem que valer o mesmo tipo em todas as ocorrências. Trata-se de uma variável de tipo, ou seja, uma cujos valores são tipos. Exemplos:

Int -> [Float] -> [Float]
Int -> [String] -> [String]
Não Exemplo: Int -> [Float] -> [String]

5.6.2. O Zipper

Com freqüência queremos combinar elementos correspondentes de duas listas. Veja a seguir:

```
zip [  “Juarez”,    “Bruno”, “Rodrigo” ] [ “Muylaert”, “Bazzanella”, “Dias”] →
        [ (“Juarez”, Muylaert”), (“Bruno”, “Bazzanella”),  (“Rodrigo”, “Dias”)]
```

O Tipo de Zip → zip :: [a] -> [b] -> [(a,b)]

Exemplo:

zip [1,2,3] [“a”, “b”, “c”] → [(1, “a”), (2, “b”), (3,“c”)]

Aqui zip :: [Int] -> [String] -> [(Int, String)]

Questão: Quer percorrer os elementos de duas listas ao mesmo tempo? Use o zip!

Exemplo: *A Posição de x em xs*

Posicao :: a -> [a] -> Int

Exemplo: posicao “b” [“a”, “b”, “c”] 2

Idéia: Marcar cada elemento com sua posição e procurar por aquele que queremos.

```
[“a”, “b”, “c”] → [(“a”,1), (“b”,2), (“c”,3)]
zip [“a”, “b”, “c”] [1, 2, 3] → [(“a”,1), (“b”,2), (“c”,3)]
Use  zip xs [1..length xs]
```

Buscando o elemento certo: Selecione as posições onde o elemento que estamos buscando aparece.

```
posicoes x xs = [pos | (x’,pos) <- zip xs [1..length xs], x’ == x]
        posicoes “b” [“a”, “b”, “c”]  [2]
```

O resultado é uma lista, mas queremos apenas um número...

```
posicao :: a -> [a] -> Int
posicao x xs = head  [pos | (x’,pos) <- zip xs [1..length xs], x’ == x]
```

Seleciona o primeiro elemento da lista de posicoes

Exemplo: *Cálculo de comprimento de caminhos*

1º - Representando um Ponto

type Ponto = (Float, Float) → Coordenadas x e y do ponto.

2º - Representando um Caminho

type Caminho = [Ponto]

caminhoExemplo = [p, q, r, s]

comprimentoCaminho = distancia p q + distancia q r + distancia r s

```
distancia :: Ponto -> Ponto -> Float
distancia (x,y) (x’,y’) = sqrt ((x-x’) ^ 2 + (y-y’) ^ 2)
```

Calculando a distância entre dois pontos pelo teorema de Pitágoras.

Duas Funções Úteis

- init xs todos menos o último elemento de xs
- tail xs todos menos o primeiro elemento de xs

init [p,q,r,s] → [p, q, r]

tail [p,q,r,s] → [q, r, s]

zip [p, q, r] [q, r, s] → [(p,q),(q,r),(r,s)]

Finalmente: *A Função comprimentoCaminho*

```
comprimentoCaminho :: Caminho -> Float
comprimentoCaminho xs = sum [distancia p q | (p,q) <- zip (init xs) (tail xs)]
```

Exemplo: comprimentoCaminho [p, q, r, s, t] → distancia p q + distancia q r + distancia r s + distancia s t

5.7. Principais Funções para Listas

Veja o padrão:

Nome da função	O que ela faz
assinatura	**Exemplo**
++	Junta duas listas.
[a] -> [a] -> [a]	**["a", "b"] ++ ["c"] → ["a","b","c"]**
!!	xs !! n retorna o n-ésimo elemento de xs. Começa do 0.
[a] -> Int -> a	**[14,7,3] !! 1 → 7**
concat	Concatena uma lista de listas em uma lista.
[[a]] -> a	**concat [[2,3],[],[4]] → [2,3,4]**
length	Retorna o comprimento de uma lista.
[a] -> Int	**length [2,2,2,2,3,4,5,90] → 8**
head	Retorna o primeiro elemento de uma lista.
[a] -> a	**head [5,2,3] → 5**
last	Retorna o último elemento de uma lista.
[a] -> a	**last [3,2,3] → 3**
tail	Todos menos o primeiro elemento da lista.
[a] -> [a]	**tail [1,2,3] → [2,3]**
init	Todos menos o último elemento da lista.
[a] -> [a]	**init [1,2,3] → [1,2]**
replicate	Constrói uma lista com n cópias de um item.
Int -> a -> [a]	**replicate 5 "a" → ["a", "a", "a", "a", "a"]**
take	Pega n primeiro elementos da lista.
Int -> [a] -> [a]	**take 5 [1,2,2,3,2,2,2,2] → [1,2,2,3,2]**
drop	Descarta os n primeiros elementos da lista.
Int -> [a] -> [a]	**drop 5 [1,2,2,3,2,2,2,2] → [2,2,2]**
splitAt	Divide a lista em uma posição.
Int -> [a] -> ([a],[a])	**splitAt 3 [2,3,2,1,1] → ([2,3,2],[1,1])**
reverse	Inverte a ordem dos elementos da lista.
[a] -> [a]	**reverse [1,2,3,4,5] → [5,4,3,2,1]**
zip	Transforma um par de listas em uma lista de pares.
[a] -> [b] -> [(a,b)]	**zip [1,2] [3,4] → [(1,3),(2,4)]**

unzip	Transforma uma lista de pares em um par de listas.
[(a,b)] -> ([a], [b])	**unzip [(1,3),(2,4)] → ([1,2], [3,4])**

sum	Soma os elementos de uma lista.
[a] -> a	**sum [1,2,3,4,5] → 15**

product	Multiplica os elementos de uma lista.
[a] -> a	**product [1,2,3,4,5] → 120**

maximum	Retorna o maior elemento de uma lista.
[a] -> a	**maximum [1,3,5,4,2,0] → 5**

minimum	Retorna o menor elemento de uma lista.
[a] -> a	**minimum [4,2,0,5,3,1] → 0**

map	Aplica uma função a cada elemento de uma lista.
f -> [a] -> [f a]	**map (*2) [1,2,3,4] → [2,4,6,8]**

filter	Filtra uma lista baseada numa restrição (função).
f -> [a] -> [f a]	**filter even [0,1,2,3,4,5] → [0,2,4]** **filter odd [0,1,2,3,4,5] → [1,3,5]**

map	Aplica uma função a cada elemento de uma lista.
f -> [a] -> [f a]	**map (*2) [1,2,3,4] → [2,4,6,8]**

words	Separa as palavras de uma frase.
String -> [String]	**words "Haskell é 10" → ["Haskell","é","10"]**

unwords	Monta uma frase apatir de uma lista de palavras
[String] -> String	**unwords ["Haskell","é","10"] → "Haskell é 10"**

5.8. Lições

- Listas modelam qualquer tipo de seqüências do mundo real.
- Listas podem expressar muitos tipos de computações repetidas em programas.
- Construções especiais e um rico conjunto de funções padrão polimórficas permitem-nos manipular listas muito facilmente.

Charada :

Você se encontra aprisionado em uma sala totalmente fechada, sem janelas e que possui apenas uma porta de saída, sem maçaneta ou fechadura. No centro da sala existe um buraco no solo de + ou - 50cm. de profundidade, no qual encontra-se o dispositivo que abre a porta: é um botão no fundo do buraco que mantém a porta fechada enquanto é pressionado por uma bolinha de pingue pongue, que é do tamanho exato da espessura do buraco, não havendo margem de espaço ao redor da bolinha para as paredes do buraco. Como fazer a bolinha parar de pressionar o buraco e assim poder sair da sala? Você possui apenas os seguintes itens.

Itens: Um jornal do dia de hoje, um lápis preto, um barbante de padaria com + ou - 1m., uma caixa de cereais (pode ser Sucrilhos ou outro de sua preferência), um clipe de metal comum.

Boa sorte e mantenha a calma para não acabar com o ar da sala!

6 – Primeira Lista de Exercícios

1. Escreva uma definição para a função *potenciaDeDois :: Int -> Int* que eleva dois a um inteiro n fornecido como argumento, ou seja, calcula 2^n.

2. Defina uma função *tresDiferentes :: Int -> Int -> Int -> Bool*, de tal forma que o resultado de tresDiferentes m n p é True apenas quando todos os três números m, n e p forem diferentes. Qual é a sua resposta para tresDiferentes 3 4 3?

3. Esta questão é sobre a função *quatroIguais :: Int -> Int -> Int -> Int -> Bool*, que retorna o valor True apenas quando todos os seus quatro argumentos são iguais. Dê uma definição de quatroIguais usando idéias presentes na definição de tresIguais:

```
tresIguais :: Int -> Int -> Int -> Bool
tresIguais m n p = (m == n) && (n == p)
```

Agora dê uma definição de quatroIguais que **use** a função tresIguais definida acima na sua definição.

4. Lembrando que *ord* fornece o código ASCII de um caractere, que *chr* fornece o caractere correspondente a um valor ASCII, defina uma função para converter minúsculas para maiúsculas que não altera caracteres que não sejam minúsculos.

5. Defina a função *caractereParaNumero :: Char -> Int*, que converte um dígito como '8' para o seu valor numérico 8. Os caracteres que não são dígitos serão definidos sempre como 0.

6. Defina uma função *proximaLetra :: Char -> Char*, que recebe como argumento uma letra do alfabeto e retorna como resultado a letra que vem imediatamente após a letra dada como argumento. *Assuma que a letra 'a' vem após a letra 'z'.*

7. Escreva uma função para retornar a média de três inteiros *mediaDeTres :: Float -> Float -> Float -> Float*. Usando esta função defina a função *quantosAcimaDaMedia :: Float -> Float -> Float -> Int*, que retorna quantas de suas entradas são maiores do que sua média.

8. Defina a função *produtoNaFaixa*, que dados dois números naturais m e n retorna o produto *m * (m+ 1) * ... * (n-1) * n*. Sua função deverá retornar 0 quando n for menor do que m.

9. Como fatorial é um caso particular de produtoNaFaixa, escreva uma definição de fatorial que use produtoNaFaixa.

10. Escreva uma definição para a função *somaDeFatoriais :: Int -> Int*, de tal forma que: somaDeFatoriais n = fatorial 0 + fatorial 1 + ... + fatorial (n-1) + fatorial n

11. Forneça três definições para uma função *dobreTodos :: [Int] -> [Int]*, que dobra todos os elementos de uma lista de inteiros, uma recursiva, uma usando list comprehensions e outra usando funções pré-definidas da linguagem.

12. Escreva funções para calcular os seguintes somatórios:

12.A) $1 + 2 + 4 + 8 + 16 + ... + 2^n$

12.B) $1^1 + 2^2 + ... + n^n$

12.C) $1 - 2 + 3 - 4 + 5 - ... + (-1)^{(n+1)} \times n$

13. Escreva uma função para:

13.A) Somar todos os n^{os} naturais que sejam múltiplos de 3 entre 1 e n.
13.B) Somar todos os n^{os} naturais que sejam múltiplos de 3 entre n e m.
13.C) Somar todos os n^{os} naturais que sejam múltiplos de k entre n e m.

14. Defina a função *divisores:: Int -> [Int]*, que retorna a lista de divisores de um inteiro positivo (e a lista vazia para outras entradas).

15. Um número primo n é um número cujos únicos **dois** divisores são 1 e n. Usando divisores defina uma função *ehPrimo :: Int -> Bool* que checa se um inteiro positivo é primo ou não, retornando False se não for primo ou se a entrada não for um número inteiro positivo.

16. Defina a função *emparelhamentos :: Int -> [Int] -> [Int]*, que pega todas as ocorrências de um inteiro n em uma lista. Por exemplo:

emparelhamentos 1 [1,2,1,4,5,1] → [1,1,1]
emparelhamentos 1 [2,3,4,6] → [ ]

17. Usando emparelhamentos defina uma função *elemento :: Int -> [Int] -> Bool*, que é True se o inteiro é um elemento da lista de inteiros e False caso contrário. Por exemplo:

elemento 1 [1,2,1,4,5,1] → True
elemento 1 [2,3,4,6] → False

18. Um inteiro positivo é dito perfeito se a soma dos seus divisores (menos o próprio) for igual ao próprio número. Escreva uma função *ehPerfeito :: Int -> Bool* que retorna True se o argumento for um inteiro perfeito e False caso contrário.

19. Escreva uma função *perfeitosEntre :: Int -> Int -> [Int]*, que retorna como resultado a lista de números perfeitos entre dois inteiros positivos.

20. Escreva uma função para contar quantas vogais estão presentes em uma linha representada por uma string.

21. Faça uma função que determina em uma lista todos os n^{os} múltiplos de 5.

22. Faça uma função que determina em uma lista de n^{os} o maior múltiplo de 5.

23. Reprograme as funções prédefinidas da linguagem:

23.A) head
23.B) tail
23.C) init
23.D) last.

LEMBRE-SE: Não se pode criar uma função com o nome de uma função que já existe. Para isso usaremos neste exercício o padrão: myHead, myTail, myInit e myLast.

24. Escreva a função que meça a distância entre dois pontos conhecendo as suas coordenadas cartesianas *(x,y)*.Considere que definimos o tipo ponto como *type Ponto = (Float, Float)* e a função como, *distancia :: Ponto -> Ponto -> Float*.

25. Escreva uma função que considerando a definição de ponto da questão anterior calcule o comprimento de um caminho. O tipo caminho é definido como *type Caminho = [Ponto]*.

26. Considerar um Banco de Dados com as seguintes informações:

```
type Usuario     = String
type Livro       = String
type Idade       = Int
type Dia         = Int
type Mes         = Int
type Ano         = Int
type Emprestimo  = (Dia, Mes, Ano)
type Cartao      = (Usuario, Idade, Livro,Emprestimo)
type BD          = [Cartao]
```

Afunção a seguir é um exemplo de lista com dados armazenados:

```
exemplo :: BD
exemplo = [  ("Rafael", 37, "Functional Programming", (20,1,2006)),
             ("Leonardo", 40, "The Art of Haskell", (20,2,2006)),
             ("Michelangelo", 36, " ", (9,4,2006)),
             ("Donatelo", 28, "A Arte da Guerra", (5,9,2006)),
             ("Donatelo", 28, "The Art of Haskell", (7, 10, 2005)),
             ("Splinter", 92, "Haskell for Computer Graphics", (10,2,2005)) ]
```

26.A) Escreva uma função que nos dê a relação de livros emprestados por usuário, com assinatura: *livros :: BD -> Usuario -> [Livro]*

26.B) Escreva uma função que determine quantos livros uma dada pessoa pegou emprestado.

26.C) Escreva uma função para determinar os usuários que não tenham livros emprestados.

26.D) Escreva uma função para gerar uma lista de todos os usuários da biblioteca, com suas respectivas idades.

26.E) Escreva uma função para listar, para um dado usuário, as datas de empréstimo de seus livros.

26.F) Escreva uma função que realize um empréstimo da biblioteca.

26.G) Escreva uma função que compute o retorno de um livro à biblioteca.

Anotações :

7 – Programando à lá seqüencial

7.1. Variáveis e Funções

Toda definição de função pode ter dois componentes, escritos com base nas variáveis apropriadas:

Assinatura (que pode ser omitida): associa uma lista de identificadores a tipos que tomam argumentos de um tipo e resultam em outro tipo. Ex.: *dec :: Int -> Int*

Equações: cada uma associa um identificador a uma função que toma como argumento alguns valores e retorna um valor. Ex.: *dec x = x - 1*

Quando existe uma assinatura para uma função, as equações correspondentes devem satisfazer seus tipos. Cada equação segue o seguinte esquema:

```
identificador p1 ... pn corpo
```

O corpo de cada equação pode ser uma expressão (**x+1**) precedida de =, no caso de representar um simples resultado de função (**inc**); ou uma lista de expressões guardadas por condições, no caso de representarem uma seleção entre vários casos diferentes resultantes da aplicação de uma função. Para:

```
fat n | n == 0    = 1
      | otherwise = n * fat (n - 1)
```

7.2. Usando *where* e *let*

Todo corpo de função pode estar acompanhado de uma lista de definições locais iniciada por *where*.

```
incdec :: Int -> (Int,Int)
incdec x = (a, b)
           where a = inc x
                 b = dec x
```

Definições locais devem aparecer todas alinhadas, em uma coluna posterior a do próprio *where*. Elas não são visíveis fora da função. Definições locais com escopo definido também podem ser feitas através de declarações do tipo *let decs in exp*, resultam em uma expressão. Quando *decs* contém mais de uma definição, elas devem ser separadas por ";" e escritas entre chaves. Também podem aparecer aninhadas.

```
incdec :: Int -> (Int,Int)
incdec x = let { a = inc x;
                 b = dec x }
           in (a,b)
```

```
incdec :: Int -> (Int,Int)
incdec x = let a = inc x
           in let b = dec x
              in (a, b)
```

7.3. Uso do *case*

Seleções ente casos também podem ser escritas usando *case exp* seguida por *of* e uma lista de alternativas separadas por **;** e escritas entre chaves. Lembre-se que o retorno de cada um dos possíveis casos avaliados pelo case deve Ter retorno igual ao retorno especificado na assinatura da função principal. Um exemplo seria um programa que dado uma String, retorna o dia da semana correspondente. Veja:

```
diasemana :: String -> Int
diasemana s = case s of {“Dom” -> 1;
                         “Seg” -> 2;
                         “Ter” -> 3;
                         “Qua” -> 4;
                         “Qui” -> 5;
                         “Sex” -> 6;
                         “Sab” -> 7}
```

Avaliando: *diasemana “Qua” => 4.*

Um exemplo similar um pouco mais incrementado seria uma função que simula uma calculadora, recebendo dois operandos e um operador, e escolhendo a ação adequada dependendo do operador. Uma possível implementação segue:

```
calcule :: Int -> Char -> Int -> Int
calcule a o b = case o of {‘+’ -> a + b ;
                           ‘-’ -> a - b ;
                           ‘*’ -> a * b ;
                           ‘/’ -> a `div` b }
```

Avaliando: *calcule 2 ‘+’ 7 => 9.*

7.4. *If – Then – Else*

Admite-se a seleção condicional entre dois valores com base em uma expressão booleana→ **if (exp :: Bool) then (exp :: a) else (exp :: a)**

Exemplo:

```
negate :: Int -> Int
negate x = if (x >= 0) then (-x) else (abs x)
```

Deve ficar claro que sempre que se utilizar esta estruturadeve-se ter um retorno para a condição testada ser verdadeira e outra para caso seja falsa. Por exemplo: função para dizer se um *Int* é par ou ímpar.

```
parimpar :: Int -> String
parimpar x = if (x `mod` 2 == 0) then (“Par”) else (“Ímpar”)
```

Uma outra possibilidade é o aninhamento de mais de um *if-then-else*. Por exemplo: Função para dizer se um *Int* é positivo ou negativo.

```
posneg :: Int -> String
posneg x = if (x > 0) then (“Positivo”)
                      else ( if (x < 0) then (“Negativo”) else (“Zero”) )
```

Excelente exemplo para se verificar a estrurura de ninhos de *if´s*. Pois não se pode esquecer da possibilidade do usuário digitar “*0*”.

7.5. *Do*

Em Haskell é possível se abrir um bloco que será interpretado como programação seqüencial, utilizando-se o *do*, que sempre termina com um retorno final através do comando *return*. Veja:

```
select :: IO Char
select = do  putStr "Quit? [y/n] "
             c <- getChar
             putStr "\n"
             return c
main :: IO ( )
main = do d <- select
          if ( (d == 'y') || (d == 'Y') ) then ( return ( ) )
                                          else ( main )
```

7.6. Polimorfismo

Uma função é polimórfica quando pode aceitar argumentos de tipos diversos, se comportando da mesma forma para qualquer um deles. Como por exemplo a função *map*, que aplica uma outra função à uma lista de qualquer tipo: **map :: (a -> b) -> [a] -> [b]**

Polimorfismo é diferente de sobrecarga pois este último conceito obriga que exista uma implementação diferente para cada tipo de argumento:

```
add :: Bool -> Bool -> Bool
add False False = False
add a     b     = True

add :: Complex -> Complex -> Complex
add (a,b) (c,d) = (a+c,b+d)
```

Sabemos que isto não é possível, pois em Haskell só pode haver uma função com um certo nome em um script.

Além disso, há também o caso das constantes, que podem ser sobrecarregadas (ex. **1 :: Float** e **1 :: Int**) mas não polimórficas.

Polimorfismo e sobrecarga são mecanismos importantes que permitem a escrita de programas complexos e, ainda assim, mais facilmente legíveis.

7.7. Casamento de Padrões

As equações que definem Funções em Haskell são escritas tendo *padrões* como argumentos. Para isto, temos algumas regras:

- Todas as equações que definem uma função devem ser contíguas;
- Número de padrões em cada equação deve ser o mesmo;
- Nenhuma variável pode aparecer mais de uma vez na lista de padrões definindo uma equação.

Padrões são definidos por, dentre outros:

1. Operadores e funções: **+**, **-**, **length**, **reverse**, etc…
2. Constantes
3. Variáveis
4. Estruturas
5. curinga: _ (usar quando argumento é inútil)
6. Padrão nomeado (*as pattern*): **var@(padrao)**
7. Padrão irrefutável: **~padrao**

Exemplos de funções, operadores, constantes e variáveis:

```
fat :: Integer -> [Integer]
fat 0 = 1
fat n = n * fat (n - 1)
```

Exemplo de casamento de estruturas:

```
myLength :: [a] -> Integer
myLength [ ]    = 0                -- equação 1
myLength (c:r)  = 1 + length r     -- equação 2
```

Exemplo de curinga e padrão nomeado:

```
myDrop :: Int -> [a] -> [a]
myDrop _ [ ] = [ ]
myDrop n l@(x:y) | n == 0  =  l
                 | otherwise = drop (n-1) y
```

Exemplo de padrão irrefutável:

```
conc :: [Bool] -> (Bool, [Bool])
conc ~(c:r)      = (c,r)
conc _           = (False, [False])
```

*Note que para esta implementação se chamarmos: *conc [ ]* → *error* (1ª equação). Pois como o padrão é irrefutável ele se vê obrigado a executar esta equação e o erro ocorre por não poder separar a lista entre cabeça e resto.

A função poderia ser escrita corretamente da sequinte maneira:

```
conc :: [Bool] -> (Bool, [Bool])
conc [ ]     = (False, [False])
conc ~(c:r) = (c,r)
```

No processo de avaliação de uma expressão com base em uma equação, procura-se determinar da esquerda para a direita se cada padrão da equação casa com o valor de cada argumento. O resultado pode ser:

- sucesso: a equação é usada na avaliação da função;
- falha: a equação é descartada, passando-se à seguinte, se ela existir;
- erro: quando não foi possível determinar uma equação para computar a função.

Desafios :

1 - Continue estas seqüências lógicas:

S T Q Q __ __ __

2 5 8 __ ← *este elemento não é o 11!*

2 - Corrija a fórmula, colocando apenas um traço:

5 + 5 + 5 = 550

3 - Por favor, escreva qualquer coisa:

_____

4 - Desenhe um retângulo com três linhas:

8 – Sobrecarga de Classes e Tipos

8.1. Introdução

Uma função polimórfica tem uma única definição que funciona sobre valores de muitos tipos diferentes. Uma função sobrecarregada funciona sobre valores de muitos tipos diferentes, mas com definições diferentes para cada um desses tipos. Agora veremos como usar classes de tipos para definir funções sobrecarregadas.

8.2. Por que sobrecarga?

Suponhamos que Haskell não tenha sobrecarregamento e definamos a seguinte função:

```
elemBool :: Bool -> [Bool] -> Bool
elemBool x [ ] = False
elemBool x (y:ys) = (x ==Bool y) || elemBool x ys
```

Temos que escrever uma função $==_{Bool}$ para comparar dois booleanos por igualdade. Suponhamos que queiramos fazer a mesma coisa para inteiros:

```
elemInt :: Int -> [Int] -> Bool
elemInt x [ ] = False
elemInt x (y:ys) = (x ==Int y) || elemInt x ys
```

Temos que escrever uma função $==_{Int}$ para comparar dois inteiros por igualdade. Portanto, toda vez que quisermos escrever uma função deste tipo para uma lista de elementos de outros tipos teremos que escrever uma função para igualdade bastante similar. Uma maneira de solucionar este problema é fazer com que a função igualdade seja passada como parâmetro:

```
elemGeral :: (a -> a -> Bool) -> a -> [a] -> Bool
```

Mas isto seria demasiadamente geral pois *a -> a -> Bool* pode ser qualquer função. A melhor solução é definir uma função que usa a igualdade sobrecarregada:

```
elem :: a -> [a] -> Bool
```

Onde o tipo *a* tem que ser restrito aqueles tipos que possuam uma igualdade. As vantagens desta abordagem são:

- *Reuso*: a definição de elem pode ser utilizada para todos os tipos com igualdade;
- *Legibilidade*: Muito melhor ler == do que $==_{Bool}$ ou $==_{Int}$;

A função de *classes de tipos* é justamente fornecer um mecanismo para "tiparmos" funções como a função *elem* acima.

8.3. Classes de Tipos

Queremos que *elem :: a -> [a] -> Bool* valha para aqueles tipos a que tenham uma operação de igualdade. A coleção de tipos sobre os quais uma função é definida é chamada classe de tipo ou, simplesmente, classe. Assim, o conjunto de tipos sobre os quais == é definido é a *classe igualdade*, **Eq**. Note que as classes que vamos tratar a seguir já existem na linguagem e já vem definidas no arquivo *Prelude.hs*.

8.3.1. Definindo a classe igualdade : Eq

```
class Eq a where
        (==) :: a -> a -> Bool
```

Membros de uma classe de tipo são chamados instâncias.

```
elem :: Eq a => a -> [a] -> Bool
```

Restringimos o tipo a para aqueles sobre os quais foi definida uma operação de igualdade. *Eq a* é chamado contexto.

Lemos `elem :: Eq a => a -> [a] -> Bool` da seguinte maneira:

Se o tipo a estiver na classe *Eq*, isto é, se == estiver definida para o tipo *a*, então elem tem tipo *a -> [a] -> Bool*

8.3.2. Um exemplo de função polimórfica que usa igualdade

```
todosIguais :: Eq a => a -> a -> a -> Bool
todosIguais m n p = (m == n) && (n == p)
```

E se agora definimos

```
sucessor :: Int -> Int
sucessor x = x + 1
```

E tentamos avaliar *todosIguais sucessor sucessor sucessor* ? ? ?

O problema é que em *todosIguais sucessor sucessor sucessor* não temos que *Int -> Int,* o tipo da função sucessor, seja uma instância da *classe Eq*. Em outras palavras, é impossível (em geral) decidir se funções do tipo *Int -> Int* são iguais ou não.

Na verdade, isto não é característico apenas de funções do tipo *Int -> Int*. Em geral, é impossível testar se duas ou mais funções são iguais ou não. Neste caso o sistema responderia com a seguinte mensagem de erro:

```
ERROR: Int -> Int is not na instance of class "Eq"
```

8.4. Assinaturas e Instâncias

Vamos ver mais de perto como declarar classes e suas instâncias. Uma classe como *Eq* é definida como:

```
class Eq a where
        (==), (/=) :: a -> a -> Bool
        x /= y          = not (x == y)  [1]
        x == y          = not (x /= y)  [1]
```

[1]Definições default que são superadas por definições em instâncias.

A instância de Eq para booleanos é definida como:

```
instance Eq Bool where
        True == True    = True
        False == False   = True
        _       == _          = False
```

* Não precisamos definir /= pois a versão default nos serve. Note que == acima será utilizada para Bool e não a versão default na classe *Eq*.

8.5. Classes Derivadas

Classes podem depender (através de *herança*[1]) de outras classes.

```
class Eq a =>[1] Ord a where
        (<), (<=), (>), (>=):: a -> a -> Bool
        max, min                        :: a -> a -> a
        compare          :: a -> a -> Ordering
```

8.6. Algumas das Classes Pré-Definidas de Haskell

Vamos agora dar uma olhada em algumas classes pré-definidas de Haskell. Para maiores detalhes, o estudante pode consultar os arquivos relevantes que vem com o sistema Hugs (ou *WinHugs* dependendo do caso).

8.6.1. Classe Eq *(Equal ou Igual)*

```
class Eq a where
        (==), (/=) :: a -> a -> Bool
        x /= y              = not (x == y)
        x == y              = not (x /= y)
```

8.6.2. Classe Ord *(Ordering ou Ordenação)*

```
class Eq a => Ord a where
        (<), (<=), (>), (>=):: a -> a -> Bool
        max, min                        :: a -> a -> a
        compare          :: a -> a -> Ordering
```

O tipo *Ordering* tem três valores: *LT (lower then)*, *EQ (equal)* e *GT (greather then)* que representam os três possíveis resultados quando comparamos dois elementos na ordenação.

Agora, veremos *Ord* quase completa, ela está sem max e min, para que vovê possa definí-las...

```
class Eq a => Ord a where
        (<), (<=), (>), (>=)            :: a -> a -> Bool
        max, min                        :: a -> a -> a
        compare          :: a -> a -> Ordering
        compare x y      | x == y  = EQ
                         | x <= y  = LT
                         | otherwise        = GT
        x <= y                              = compare x y /= GT
        x < y                               = compare x y == LT
        x >= y                              = compare x y /= LT
        x > y                               = compare x y == GT
```

8.6.3. Classe Enum *(Enumerate ou Enumareda)*

```
class Ord a => Enum a where
        toEnum              :: Int -> a             -- converte Int p/ a
        fromEnum            :: a -> Int             -- converte a p/ Int
        enumFrom            :: a -> [a]             -- [n .. ]
        enumFromThen        :: a -> a -> [a]        -- [n,m..]
        enumFromTo          :: a -> a -> [a]        -- [n..m]
        enumFromThenTo      :: a -> a -> a -> [a]   -- [n,n'..m]
```

Exemplos:

```
ord :: Char -> Int
ord = fromEnum

chr :: Int -> Char
chr = toEnum

succ, pred :: Enum a => a -> a
succ = toEnum . (+1) . fromEnum
pred = toEnum . (subtract 1) . fromEnum
```

8.6.4. Classe Show *(ou Mostrar)*

```
type ShowS = String -> String

class Show a where
        showsPrec :: Int -> A -> ShowS
        show                  :: a -> String
        showList        :: [a] -> ShowS
```

A função mais utilizada na prática é a show que converte um valor em uma string. Ela é bastante utilizada em impressões. Exemplos de instâncias:

```
instance Show Bool where
        show True               = "True"
        show False      = "False"
instance (Show a, Show b) => Show (a,b) where
        show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"
```

8.6.5. Classe Read

Permite que transformemos strings em valores. Para utilizar a classe basta conhecer a função *read :: (Read a) => String -> a* .

É importante ver que em muitos casos o tipo do resultado de read tem que ser especificado. Por exemplo, podemos escrever *(read " 1 ") :: Int* para indicar que queremos que o resultado seja Int.

8.6.6. Classes Numéricas

- Inteiros de precisão limitada → *Int*;
- Inteiros de precisão arbitrária → *Integer*;
- Números de ponto flutuante de precisão simples → *Float*;
- Números de ponto flutuante de precisão dupla → *Double*;
- Números racionais → *Rational*;
- etc.

8.6.7. Classe Num *(Numeric ou Numérica)*

```
class (Eq a, Show a) => Num a where
        (+), (-), (*)       :: a -> a -> a
        negate              :: a -> a
        abs, signum         :: a -> a
        fromInteger         :: Integer -> a
        fromInt             :: Int -> a
        div, mod,           :: a -> a -> a
        quot, rem           :: a -> a -> a
        x - y               = x + negate y
        fromInt             = fromIntegral
```

8.6.8. Outras Classes

Para detalhes de outras classes e dessas que acabamos de ver, olhe o arquivo *Prelude.hs*. Além disso, olhe em outras bibliotecas e na documentação da linguagem.

Anotações :

9 – O Problema da Ordenação

9.1. Conferindo o Pulo do Gato

Vamos estudar o pulo do gato e ver como funções tais como *sum*, *++* e *take* podem ser definidas, a recursão é a chave: ela permite que façamos computações repetidas para processar cada elemento da lista por vez. Apesar de Haskell fornecer um conjunto rico de funções pré-definidas para listas você acabará precisando definir as suas próprias funções.

9.2. Casamento de Padrões sobre Listas

O modo mais básico de separar os elementos de uma lista é através de casamento de padrões:

```
sum [ ] = 0
sum [x] = x
sum [x,y] = x + y
```

Mas como podemos olhar dentro de uma lista de comprimento arbitrário? Uma idéia errada seria: *sum (xs ++ ys) = sum xs + sum ys*

Pois pode casar em mais de uma maneira! Como por exemplo:

```
[1,2,3] == [1] ++ [2,3] == [1,2] ++ [3]
```

Logo, Precisamos de um modo único de representar listas, qualquer lista (não-vazia) xs pode ser escrita como: *xs == [y] ++ ys*

De uma única maneira! Ou seja, uma cabeça e o resto da lista.

Exemplos:
```
[1] == [1] ++ []
[1,2] == [1] ++ ([2] ++ [])
[1,2,3] = [1] ++ ([2] ++ ([3] ++ []))
```

Passaremos a usar uma nova notação:

y : ys → Chamada de *cons*

9.2.1. Padrões *Cons (:)*

Cons (:) é permitido em padrões → x : xs

Onde : *x* casa com o primeiro elemento
xs casa com os elementos restantes

Exemplos:
```
head :: [a] -> a          tail :: [a] -> a
head (x : xs) = x         tail (x : xs) = xs
```

9.2.2. Recursão Primitiva sobre Listas

Toda lista ou casa com [ ] (caso base, a lista vazia) ou com *x:xs* (a estrutura padrão de lista, é o caso recursivo, com uma lista menor em *xs*). Veja exemplos:

sum :: [Int] -> Int sum [] = 0 sum (x : xs) = x + sum xs	length :: [a] -> Int length [] = 0 length (x : xs) = length xs + 1

9.2.3. Checando por um elemento

```
elem x xs        →      retorna True se x é um elemento de xs
elem x ( y : ys) = x == y || elem x ys
elem x [ ]       = False
```

Alternativamente podemos: *elem x xs = or [x == y | y <- xs]*

9.2.4. Definindo ++

```
(++) :: [a] -> [a] -> [a]
[ ] ++ ys = ys
(x : xs) ++ ys = x : (xs ++ ys)
```

9.3. Ordenação por Inserção

A função a seguir que insere um elemento numa lista parte do princípio que a lista passada como parâmetro está previamente ordenada:

```
insira :: (Ord a) => a -> [a] -> [a]
insira x [ ] = [x]
insira x (y : ys)  | x <= y  = x : y : ys
                   | x > y   = y : insira x ys
```

E para ordenar tem-se o sort :

```
sort :: (Ord a) => [a] -> [a]
sort [ ] = [ ]
sort (x : xs) = insira x (sort xs)
```

9.4. Variações em Recursões sobre Listas

A idéia chave é que o caso recursivo expressa o resultado em termos da mesma função sobre uma lista mais curta, e o caso base lida com a menor lista possível. Mas as variações possíveis são muitas...

9.4.1. Definindo init

Lembre-se que a função init retorna a lista sem o último elemento.

```
init :: [a] -> [a]          →      Compare isso a tail (x : xs) = xs
init [x] = [ ]
init (x : xs) = x : init xs
```

9.5. Checando se uma Lista está Ordenada

```
ordenada :: [a] => Bool
ordenada [ ] = True
ordenada [x] = True
ordenada (x : y : xs) = x <= y && ordenada (y : xs)
```

9.5.1. Zip: Recursão sobre dois argumentos

```
zip :: [a] -> [b] -> [(a,b)]
zip [ ] ys = [ ]
zip xs [ ] = [ ]      -- 2 condições de parada
zip (x : xs) (y : ys) = (x,y) : zip xs ys  -- Os 2 argumentos diminuem
```

9.5.2. Definindo take

```
take :: Int -> [a] -> [a]
take 0 (x : xs)    = [ ]
take n [ ]         = [ ]      -- 2 condições de parada
take n (x : xs) = x : (take (n-1) xs)
```

9.6. Merge Sort

Estratégia básica:

- Divida a lista em dois pedaços de tamanhos similares (duas metades);
- Ordene cada metade;
- Combine ("merge") as duas metades ordenadas em uma única lista.

O merge sort é eficiente pois mesclar duas listas já ordenadas é mais rápido que realmente ordenar uma lista. E a seguir, a função merge, que faz a junção ordenada das partes:

```
merge :: (Ord a) => [a] -> [a] -> [a]
merge [ ] ys      = ys
merge xs [ ]      = xs
merge (x : xs) (y : ys)    | x <= y = x : merge xs (y : ys)
                           | x > y   = y : merge (x : xs) ys
```

E a seguir temos a função mergeSort, para fazer as divisões:

```
merge :: (Ord a) => [a] -> [a]
mergeSort [ ] = [ ]
mergeSort [x] = [x]
mergeSort xs | tamanho > 0 = merge (mergeSort esq) (mergeSort dir)
                  where tamanho = length xs `div` 2
                        esq     = take tamanho xs
                        dir     = drop tamanho xs
```

9.7. Quick Sort

Existe possibilidade de se livrar da necessidade de fazer o merge? Sim, se fizer o seguinte:

Escolha um elemento da lista e a dividir em dois pedaços de tamanhos similares, de tal forma que um contenha elementos menores ou iguais ao escolhido e o outro elementos maiores que o escolhido.

- Ordenar cada metade;
- Portanto, como as duas metades estão ordenadas, não é preciso fazer o merge!

```
qS :: (Ord a) => [a] -> [a]
qS [ ]      = [ ]
qS (c:r) = qS [ a | a <- r, a <= c] ++ [c] ++ qS [b | b <- r, b > c]
```

É bonito ou não é???

9.7.1. Definindo reverse: Função utilizada para inverter uma lista. Já é pré-definida na linguagem.

```
reverse :: [a] -> [a]
reverse [ ] = [ ]
reverse (x : xs) = reverse xs ++ [x]
```

9.7.2. Recursão + List Comprehensions

Quando utilizadas em conjunto, recursão e "list comprehensions" podem ser uma ferramenta poderosa. Exemplo: Defina *nub :: [a] -> [a]* , uma função que remove elementos duplicados de uma lista. Veja:

```
nub :: (Eq a) => [a] -> [a]
nub [ ]          = [ ]
nub (x : xs)     = x : nub [ y | y <- xs, y /= x ]
```

Assim: nub [1,2,1,3] → [1,2,3]

9.8. Lições

Recursão funciona bem com listas: reduzimos o problema de computar uma função ao mesmo problema para uma lista mais curta.

Freqüentemente, os casos que consideramos são:

f [ ] = ...	→	Caso base
f (x : xs) = ... f xs	→	Caso recursivo

Algumas vezes *list comprehensions* oferecem soluções mais simples: use o bom senso para definir o mais indicado em cada caso.

Anotações :

10 – Funções de Alta Ordem

10.1. Propósitos

Elaborar a idéia de que funções podem ter outras funções como argumentos e produzir funções como resultados.Fazer uso do poder desta idéia para definir funções gerais que capturam padrões de computação comuns. Veja alguns exemplos:

```
negarTudo :: [Int] -> [Int]
negarTudo [ ] = [ ]
negarTudo (c : r) = negate c : negarTudo r

maiusculaTodos :: String -> String
maiusculaTodos [ ] = [ ]
maiusculaTodos (c : r) = toUpper c : maiusculaTodos r

comprimentos :: [ [ a ] ] -> [ a ]
comprimentos [ ] = [ ]
comprimentos (c : r) = length c : comprimentos r
```

10.1.1. Capturando o padrão comum

```
map :: (a -> b) -> [a] -> [b]
map f [ ] = [ ]
map f (c : r) = (f c) : (map f r)
```

A função pré definida *map*, é uma função que recebe uma função *f* que pode ser aplicada a um elemento de uma lista de elementos. E retorna como saída uma lista com todos os elementos transformados por *f*.

10.1.2. Exemplos revisitados

```
negarTudo x = map negate x
maiusculaTodos x = map toUpper x
comprimentos x = map length x
```

10.2. Seções de Operadores

Um novo modo de expressar funções comuns

- (+2) A função que adiciona 2 ao seu argumento
- (>0) A função que checa se o seu argumento é positivo
- (++"\n") A função que adiciona uma nova-linha ao seu argumento

Em geral :

- (op y) x = x op y
- (x op) y = x op y

10.2.1. Como você usaria hugs para computar :

$1^2 + 2^2 + 3^2 + 4^2 + ... + 20^2$?

Uma solução seria:

```
sum (map (^2) [1..20])  →  2870
```

Digamos agora que em uma lista de números de telefone, representados como strings [“21/4561223”, “24/5222223”, “61/3334455”, ...], como você encontraria os números de Nova Friburgo? Dica: O código de DDD de Nova Friburgo é “24”. Solução:

```
numerosFriburgo :: [String] -> [String]
numerosFriburgo [ ] = [ ]
numerosFriburgo (nr:nrs) | take 3 nr == “24/”     = nr : numerosFriburgo nrs
                         | otherwise              = numerosFriburgo nrs
```

10.3. Outro padrão comum

Dada uma lista mantenha aqueles elementos que satisfaçam alguma propriedade e descarte os outros, como se realizando uma operação de *filtragem*...

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [ ] = [ ]
filter p[1] (c : r)[2]  | p c          = c : (filter p r)[3]
                        | otherwise    = filter p r
```

Obs.: 1 - Uma função descrevendo uma propriedade
2 - Lista de elementos a serem filtrados
3 - Elementos que têm a propriedade p
Propriedade: uma função que retorna um booleano. Exemplos:

```
filter even [1..10]                    →   [2,4,6,8,10]
filter odd [1..10]                     →   [1,3,5,7,9]
filter (<100) (map (^2) [6,7,13,8])    →   [36,49,64]
```

10.4. Map e Filter como “List Comprehensions”

- map f xs = [f x | x <- xs]
- filter p xs = [x | x <- xs, p x]

List comprehensions podem ser explicadas em termos de map e filter, isto é, em termos de funções definidas recursivamente.

10.4.1. Números primos revisitados

Um número n é primo se seus únicos divisores são 1 e n.

- ehDivisor n k = n `mod` k == 0
- ehPrimo n = filter (ehDivisor n) [1..n] == [1,n]

10.5. Composição de Funções

O operador para composição de funções é do tipo:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
(f . g) x = f  (g x)
```

Portanto, *f . g* é a notação de teclado. Na Matemática você escreveria *f o g*.

10.5.1. Números de telefone (*novamente...*)

Vamos definir uma função que pega uma lista de números de telefone como argumento e retorna os números de Nova Friburgo como resultado, com o código da cidade removido. Nova solução:

```
numerosFriburgo = map (drop 3)[2] . (filter ehFriburgo)[1]
                        where ehFriburgo = ((== "24/") . (take 3))[3]
```

Obs.: 1 - Primeiro filtre os números de Friburgo

2 - Depois remova os códigos

3 - A propriedade de ser um número de Friburgo: pegue os primeiros 3 caracteres e então cheque que eles sejam "24/"

10.6. Criando o *Folder*

Uma forma de se criar funções de alta ordem é observar padrões comuns em funções que funcionam de forma semelhante:

```
listSum :: (Num a) => [a] -> [a]
listSum [ ] = 0
listSum (x:xs) = x + listSum xs

listProd :: (Num a) => [a] -> [a]
listProd [ ] = 1
listProd (x:xs) = x * listProd xs
```

Nestas duas funções alguns padrões se repetem:

1. A recursividade tem como caso base a lista vazia;
2. caso recursivo é a aplicação de uma operação matemática sobre os elementos de uma lista;
3. retorno da base de recursão é o elemento neutro da operação em questão.

Assim poderíamos definir a função de alta ordem *folder* como sendo:

```
folder :: (a ->a -> a) -> a -> [a] -> [a]
folder op init [ ] = init
folder op init (x:xs) = x `op` folder op init xs
```

Onde: op → É o operador a ser aplicado na lista

init → É o elemento a ser utilizado como caso base da recursão.

Desta forma podemos re-escrever listSum e listProd como:

```
listSum :: (Num a) => [a] -> [a]
listSum xs = folder (+) 0 xs

listProd :: (Num a) => [a] -> [a]
listProd xs = folder (*) 1 xs
```

11 – Entrada e Saída (I/O)

11.1. Funções para realizar I/O

Em Hasquell o retorno de uma função não é uma ação de saída. E nem os parâmetros de entrada representam açoes de entrada. Para tal fim existem funções próprias para isto.

11.1.1. Funções de Saída

```
putChar :: Char -> IO ()
```

Recebe um tipo *Char* qualquer e retorna uma ação de saída.

```
putStr :: String -> IO ()
```

Recebe uma *String* (lista de *Char*) qualquer e retorna uma ação de saída.

```
putStrLn :: String -> IO ()
```

Semelhante à *putStr*, mas além da ação de saída pula uma linha. Recebe uma *String* (lista de *Char*) qualquer e retorna uma ação de saída.

```
print :: a -> IO ()
```

Esta é uma função mais genérica, pode receber como entrada um tipo qualquer e retorna uma ação de saída.

```
show :: a -> String
```

Transforma qualquer tipo de entrada em uma String na saída, muito útil para realizar conversões genéricas, para o *putStr* ou *putStrLn*.

11.1.2. Funções de Entrada

```
getChar :: IO Char
```

Esta função faz a leitura de exatamente um caractere, normalmente vindo do teclado.

```
getLine :: IO String
```

Semelhante à *getChar*, mas pega mais de um caractere, retornando o conjunto como *String*. De maneira semelhante temos também a função *getContents :: IO String*.

```
read :: String -> a
```

Lê de um tipo *String* para posterior transformação. *Exemplo:* (read (show 26))::Float

11.2. Introdução à Manipulação de Arquivos

```
readFile :: String -> IO String
```

Faz a leitura do conteúdo de um arquivo, cuja localização é passada através de uma *String* no parâmetro de entrada. Retornando uma ação de *IO String*. *Exemplo:* readFile "D:\\temp\\link.txt"

```
writeFile :: String -> String -> IO ()
```

Faz a gravação em um arquivo, cuja localização é passada através de uma *String* no primeiro parâmetro de entrada o conteúdo passado através de outra *String*, passada no segundo parâmetro de entrada. Retornando uma ação de *IO ()*. *Exemplo:* writeFile "D:\\temp\\eu.txt" "É só um exemplo"

Obs.: Caso o arquivo mencionado não exista ele é criado.

```
appendFile :: String -> String -> IO ()
```

Concatena em um arquivo, cuja localização é passada através de uma *String* no primeiro parâmetro de entrada o conteúdo passado através de outra *String*, passada no segundo parâmetro de entrada. Retornando uma ação de *IO ()*. *Exemplo:* appendFile “n.txt” (show (notas turma))

```
>> :: IO a -> IO b
```

Realiza a omissão de um *IO ()* por outro *IO ()* a ser especificado.

Exemplo: writeFile “2t01.txt” (show turma) >> putStr “ok”

```
>>= :: IO a -> (a -> IO b)
=<< :: IO a -> (a -> IO b)
```

Realizam os respectivos redirecionamentos de saída e entrada.

Exemplo: readFile “haskell.tex” >>= putStr

Anotações :

12 – Definindo Novos Tipos

12.1. Por que definir novos tipos?

Usamos Haskell para modelar dados do mundo real. Mas alguns deles são difíceis de serem modelados através dos tipos existentes.

Por exemplo: Dias da semana (Segunda, Terça, Quarta, etc.)

Podemos modelá-los como números:

- Mas o que significa Quarta+Sabado?
- E se dia == 8?

Podemos modelá-los como Strings?

- Mas o que Segunda++Terca significa?
- E se dia == "asdfg"?

12.2. Definindo Novos Tipos com *type*

Em Haskell podemos criar um novo nome de tipo usando a palavra chave **type**. Ele será como um apelido para um tipo que já seja pré - existente na linguagem. A seguir alguns exemplos:

```
type Name = String
type Id = Integer
type Person = (Id, Name)
type Database = [Person]
```

12.3. Definindo Tipos Algébricos com *data*

Ao invés de utilizarmos tipos já existentes definiremos um novo tipo contendo apenas os valores que nós queremos.

```
data Dia = Segunda | Terça | Quarta | Quinta | Sexta | Sábado | Domingo
```

Onde :

- Dia é o nome do tipo
- Os valores do novo tipo; sempre começam com maiúscula

Valores desse tipo são diferentes de todos os outros valores e não podem ser somados, concatenados, etc.

12.4. Funções sobre Tipos Novos

Definimos funções sobre novos tipos através de casamento de padrões:

```
diaDaSemana :: Dia -> Bool
diaDaSemana Segunda = True
diaDaSemana Segunda = True
diaDaSemana Terça = True
diaDaSemana Quarta = True
diaDaSemana Quinta = True
diaDaSemana Sexta = True
diaDaSemana Sábado = False
diaDaSemana Domingo = False
```

12.4.1. Derivando Instâncias de Classes

Como vimos anteriormente, Haskell possui diversas classes pré-definidas (Eq, Ord, Enum, Show, etc.). Quando introduzimos um tipo novo, como Dia, podemos necessitar de igualdade, enumerações, etc. Isto pode ser suprido pelo sistema se pedirmos. Assim:

```
data Dia = Segunda | Terça | Quarta | Quinta | Sexta | Sábado | Domingo
           deriving (Eq, Ord, Enum, Show, Read)
```

Podemos ter, então, *[Quarta .. Sábado]* denotando a lista *[Quarta, Quinta, Sexta, Sábado]* ou, ainda, *show Segunda* que forneceria a string *"Segunda"*.

Mas você sempre pode definir as suas próprias instâncias ao invés de utilizar as default. Como a definição de Bool por exemplo. Booleanos são definidos como:

```
data Bool = True | False
True && p = p
False && p = False
not True = False
not False = True
```

12.4.2. Árvores Binárias de Inteiros

Uma possível definição:

```
data ABI = ABIVazia | ABINo Int ABI ABI
           deriving (Eq, Ord, Show, Read)
```

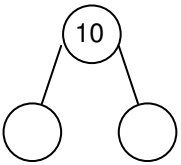
Note a definição recursiva... Para testar:

A) ABINo 10 ABIVazia ABIVazia

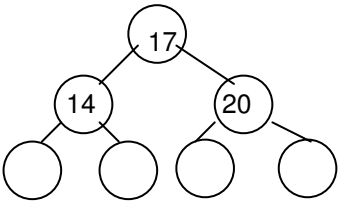
B) ABINo 17 (ABINo 14 ABIVazia ABIVazia) (ABINo 20 ABIVazia ABIVazia)

Onde A e B representam :

A



B



12.4.3. Algumas Funções

```
somaABI :: ABI -> Int
somaABI  ABIVazia = 0
somaABI (ABINo n e d) = n + somaABI e + somaABI d
```

```
prof :: ABI -> Int
prof ABIVazia = 0
prof (ABINo n e d) = 1 + max (prof e) (prof d)

espelho :: ABI -> ABI
espelho ABIVazia = ABIVazia
espelho (ABINo n e d) = ABINo n (espelho d) (espelho e)

nnos :: ABI -> Int
nnos ABIVazia = 0
nnos (ABINo n e d) = 1 + nnos e + nnos d

ocorre :: ABI -> Int -> Int
ocorre ABIVazia  p = 0
ocorre (ABINo n e d) p | n == p       = 1 + ocorre e p + ocorre d p
                       | otherwise  =       ocorre e p + ocorre d p
```

12.5. Modelando Expressões

Vamos projetar um tipo de dados para modelar expressões aritméticas, não os seus valores mas sua estrutura. Uma expressão pode ser: data Expr =

- um número n — Num Int
- uma variável x — | Var String
- uma adição a+b — | Soma Expr Expr
- uma multiplicação a*b — | Mul Expr Expr

Por exemplo:

```
data Expr = Num Int | Var String |  Soma Expr Expr | Mul Expr Expr
```

A expressão:	É representada por:
2	Num 2
2 + 2	Soma (Num 2) (Num 2)
(1 + 2) * 3	Mul (Soma (Num 1) (Num 2)) (Num 3)
1 + 2 * 3	Soma (Num 1) (Mul (Num 2) (Num 3))

Você pode definir uma função *avalia :: Expr -> Int* que avalia uma expressão aritmética? Como por exemplo:

avalia (Soma (Num 1) (Mul (Num 2) (Num 3))) → 7

Não se preocupe com variáveis; cuide apenas dos outros três casos. *Use casamento de padrões:* uma equação para cada caso.

Observação Importante: Temos que envolver Num n com parênteses (e em todo uso de padrões sobre valores de novos tipos!). Caso contrário, a função avalia estaria sendo definida sobre dois argumentos, a saber: Num e n.

```
avalia (Num n)  = n
avalia (Soma a b)        = avalia a + avalia b
avalia (Mul a b) = avalia a * avalia b
```

12.5.1. Formatando Expressões

```
formatExpr :: Expr -> String
formatExpr (Num n) = show n
formatExpr (Var x) = x
formatExpr (Soma a b) = formatExpr a ++ "+" ++ formatExpr b
formatExpr (Mul a b) = formatExpr a ++ "*" ++ formatExpr b
formatExpr (Mul (Num 1) (Soma (Num 2) (Num 3)))      →
"1*2+3"
```

Quais parênteses são necessários?

- 1+(2+3) → Não
- 1+(2*3) → Não
- 1*(2+3) → Sim

Que tipo de expressão pode precisar de parênteses? *Adições*
Quando precisa ser parentisada? *Dentro de multiplicações*

Forneça mais um argumento a formatExpr para dizer em qual contexto seu argumento ocorre. Aí temos:

```
data Contexto = Multiplicacao | QualquerOutro
formatExpr (Soma a b) Multiplicacao = "(" ++ formatExpr (Soma a b) QualquerOutro ++ ")"
formatExpr (Mul a b) _ = formatExpr a Multiplicacao ++ "*" ++ formatExpr b Multiplicacao
```

12.6. Modelando fracassos

Muitas funções podem fracassar em produzir um resultado em algumas ocasiões. Um exemplo: Procurando uma entrada em uma tabela.

```
Type Tabela = [(String,Int)]
busca "y" [("x",1),("y",2)] → Encontra 2
busca "z" [("x",1),("y",2)] → Não Encontra

data TalvezInt = Encontra Int | NaoEncontra
```

Ao invés de definirmos tipos "Talvez" similares, definimos apenas um destes tipos, com um parâmetro de tipo.

```
data Talvez a[1] = Justamente a[2] | Nada[3]
```

Onde: 1 – É o parâmetro de tipo
2 – Representa sucesso
3 – Representa fracasso

Exemplos: 1) Justamente 2, Justamente 3, Nada :: Talvez Int
2) Justamente "ola!", Nada :: Talvez String

12.7. Usando Novos Tipos para Eficiência

Novos tipos podem ser introduzidos para modelar alternativas em programas, mas também para melhorar eficiência. Vamos considerar um importante exemplo: buscar chaves em tabelas.

Uma tabela é uma coleção de chaves e valores associados.

Exemplos: catálogo telefônico, cujas chaves são nomes e cujos valores são números de telefone.

Problema: Dadas uma tabela e uma chave, descubra o valor associado. Dado que uma tabela pode conter qualquer tipo de chaves e valores, defina um tipo parametrizado:

```
type Tabela a b = [(a,b)]
busca :: Ord a => a -> Tabela a b -> Talvez b
busca chave []                    = Nada
busca chave ((c,v):t) | chave == c = Justamente v
                      | otherwise  = busca chave t
```

12.7.1. Buscando chaves rapidamente

O problema é que buscar chaves pesquisando desde o início é lento. Melhor: procure no meio, e então procure para a frente ou para atrás dependendo do que você achou. (Para que isto funcione, assumimos que a tabela esteja ordenada!)

12.7.2. Representando Tabelas

Temos que quebrar uma tabela rapidamente em:

- Uma tabela menor de entradas antes da entrada no meio;
- A entrada no meio;
- Uma tabela de entrada após esta.

Uma alternativa seria:

```
data Tab a b = Junte (Tab a b) a b  (Tab a b) | Vazio
```

Bruno	Meier
. . .	. . .
Guto	Nova América
. . .	. . .
. . .	. . .
Zé Geraldo	Niterói

Para buscar uma chave em uma tabela:

- Se a tabela estiver vazia, então a chave não é encontrada;
- Compare a chave com a chave do elemento do meio;
- Se forem iguais, retorne o valor associado;
- Se a chave é menor do que a chave no meio, busque na primeira metade da tabela;
- Se a chave é maior do que a chave no meio, busque na segunda metade da tabela;

Defina:

```
buscar :: Ord a => a => Tab a b => Talvez b
buscar chave Vazio = Nada
buscar chave (Junte esq c v dir) | chave == c    = Justamente v
                                 | chave < c     = buscar chave esq
                                 | chave > c     = buscar chave dir
```

Lembre-se que:

```
data Tab a b = Junte (Tab a b) a b  (Tab a b) | Vazio
```

Para inserir um novo valor precisaremos, também, de uma função para construção de tabelas. Assim, definimos: *" insere :: Ord a => a -> b -> Tab a b -> Tab a b "* para inserirmos um novo valor para uma chave em uma tabela.

Temos que ser cuidadosos para inserir a nova entrada no lugar certo, mantendo a ordenação. Uma idéia é comparar a nova chave com a do meio. Trocar o valor do meio se as chaves forem iguais, caso contrário inserir na primeira ou segunda metade como apropriado:

```
ins chave val Vazio = Junte Vazio chave val Vazio
ins chave val (Junte e c v d) | chave == c = Junte e c val d
                              | chave < c = Junte (ins chave val e) c v d
                              | chave > c = Junte e c v (ins chave val d)
```

Obs.: Muitos esquecem de juntar a nova metade à direita com a metade à esquerda novamente.

Anotações :

13 – Tipos Abstratos de Dados

13.1. Objetivos

Olhar com detalhes o sistema de módulos de Haskell, que permite que definições de funções e outros objetos sejam escondidos ("*hidden*") de outros módulos.

Como isto pode ser feito sobre tipos também, veremos como podemos manipular Tipos Abstratos de Dados (como filas, pilhas e árvores de busca).

Finalmente, veremos algumas aplicações.

13.2. Representação de Tipos

Suponha que queremos construir uma calculadora para expressões aritméticas. Como parte de nossa implementação, modelaremos valores de variáveis através de uma memória. Portanto, a partir de agora discutiremos a confecção de um módulo (através de "module") que contém rotinas e tipos para maniupulação de memórias.

13.2.1. Como representar a memória?

Existem várias maneiras diferentes!

E o que nós vamos fazer com ela?

inicial :: Memoria	*Valor inicial para toda a memória*
valor :: Memoria -> String -> Int	*Fornece o valor de uma variável*
Atualiza :: Memoria -> String -> Int -> Memoria	*Atualiza mem. com novo valor da variável*

Esses são os melhores modelos?

Por exemplo, o primeiro modelo permite que a lista (que é a memória) seja revertida e o segundo que façamos a composição da memória (que é uma função) com outra função! Qual seria um melhor modelo para a memória?

Bem, a resposta é definir um tipo que só tenha sobre ele as operações inicial, valor e atualiza, de tal forma que a representação da memória não seja abusada.

Portanto, esconderemos a informação de como o tipo memória é realmente implementado.

Somente as três operações citadas é que serão permitidas operar objetos do tipo implementando a memória de nossa calculadora.

13.3. O módulo Memória

```
module Memoria
        (Memoria,
          inicial,                  -- :: Memoria
          valor,                    -- :: Memoria -> String -> Int
          atualiza                  -- :: Memoria -> String -> Int -> Memoria
        ) where
data Memoria = Mem [ (Int, String) ] deriving (Eq, Ord, Show, Read)

inicial :: Memoria
inicial = Mem [ ]

valor :: Memoria -> String -> Int
valor (Mem []) v = 0
valor (Mem ((n,w) : mem)) v     | v == w= n
                                | otherwise      = valor (Mem mem) v

atualiza :: Memoria -> String -> Int -> Memoria
atualiza (Mem mem) v n = Mem ((n,v) : mem)
```

13.3.1. Classes: exibindo valores e igualdade

No module Memoria poderíamos ter, por exemplo:

```
instance Eq Memoria where (Mem m1) == (Mem m2) = m1 == m2
instance Show Memoria where show n (Mem m) = show n m
```

Problema: as instâncias não podem ser escondidas! Mas, de qualquer maneira, você pode optar por não tê-las ...

13.4. Filas

```
module Fila
        ( Fila,
          filaVazia,                -- Fila a
           ehFilaVazia,             -- Fila a -> Bool
           insFila,                 -- a -> Fila a -> Fila a
           remFila                  -- Fila a -> (a, Fila a)
        ) where
data Fila a = F [a] deriving (Eq, Ord, Show, Read)
filaVazia = F [ ]

ehFilaVazia (F [ ])       = True
ehFilaVazia _             = False          -- Casa com qualquer coisa!

insFila x (F xs) = F (xs ++ [x])

remFila (F xs)   | not (ehFilaVazia (F xs)) = (head xs, F (tail xs))
                 | otherwise = error “remFila”
```

13.5. Pilhas

```
module Pilha
        ( Pilha,
          pilhaVazia,           -- Pilha a
           ehPilhaVazia, -- Pilha a -> Bool
           push,          -- a -> Pilha a -> Pilha a
           pop                   -- Pilha a -> (a, Pilha a)
        ) where
data Pilha a = P [a] deriving (Eq, Ord, Show, Read)

pilhaVazia = P [ ]

ehPilhaVazia (P [ ])      = True
ehPilhaVazia _ = False

push x (P xs) = P (x : xs)

pop (P xs)      | not (ehPilhaVazia (P xs))      = (head xs, P (tail xs))
                | otherwise                      = error "pop"
```

13.6. Filas Duplas

```
module FilaDupla
        ( Fila,
          filaVazia,            -- Fila a
           ehFilaVazia,         -- Fila a -> Bool
           insFila,             -- a -> Fila a -> Fila a
           remFila              -- Fila a -> (a, Fila a)
        ) where
data Fila a = F [a] [a] deriving (Eq, Ord, Show, Read)

filaVazia = F [ ] [ ]

ehFilaVazia (F [ ] [ ])    = True
ehFilaVazia _              = False

insFila x (F xs ys) = F xs (x:ys)

remFila (F (x:xs) ys)     = (x, F xs ys)
remFila (F [ ] ys)        = remFila (F (reverse ys) [ ])
remFila (F [ ] [ ])       = error "remFila"
```

Obs.: *insFila* e *remFila* são mais "baratas" neste caso...

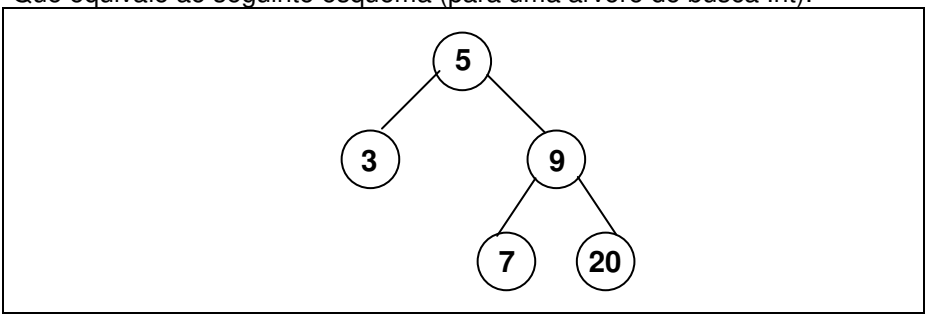
13.7. Árvores de Busca

```
module Arvore
        (Arv,
          vazia,                    -- Arv a
          ehVazia,                  -- Arv a -> Bool
          ehNo,           -- Arv a -> Bool
          arvEsq,                   -- Arv a -> Arv a
          arvDir,         -- Arv a -> Arv a
          valArv,                   -- Arv a -> a
          insArv,                   -- Ord a => a -> Arv a -> Arv a
          remove,                   -- Ord a => a -> Arv a -> Arv a
          minArv                    -- Ord a => Arv a -> Maybe a
) where
data Arv a = Vazia | No a (Arv a) (Arv a) deriving (Eq, Ord, Show, Read)
```

Neste exemplo representamos uma árvore sem ponteiros:

```
No 5  (No 3 Vazia Vazia) (No 9 (No 20 Vazia Vazia) (No 7 Vazia Vazia))
```

Que equivale ao seguinte esquema (para uma árvore de busca Int):



13.7.1. Continuação do módulo Árvore

```
vazia :: Arv a
vazia = Vazia

ehVazia :: Arv a -> Bool
ehVazia Vazia  = True
ehVazia _                 =  False

ehNo :: Arv a -> Bool
ehNo Vazia = False
ehNo _       =  True
```

```
arvEsq :: Arv a -> Arv a
arvEsq Vazia       = error “arvEsq”
arvEsq (No _ e _) = e

arvDir :: Arv a -> Arv a
arvDir Vazia       = error “arvDir”
arvDir (No _ _ d) = d

valArv :: Arv a -> a
valArv Vazia         = error “valArv”
valArv (No v _ _) = v

insArv :: Ord a => a -> Arv a -> Arv a
insArv val Vazia = No val Vazia Vazia
insArv val (No v e d)     | v == val  = No v e d
                          | val > v   = No v e (insArv val d)
                          | val < v   = No v (insArv val e) d

remove :: Ord a => a -> Arv a -> Arv a
remove val (No v e d)   | val < v = No v (remove val e) d
                        | val > v = No v e (remove val d)
                        | ehVazia d = e
                        | ehVazia e = d
                        | otherwise = junta e d

minArv:: Ord a => Arv a -> Maybe a
minArv a          | ehVazia a      = Nothing
                  | ehVazia e      = Just v
                  | otherwise      = minArv e
                        where      e = arvEsq a
                                   v = valArv a

-- junta é uma funcao auxiliar, usada em remove. Ela não eh exportada
junta :: Ord a => Arv a -> Arv a -> Arv a
junta a1 a2 = No mini a1 novaarv
                    where (Just mini) = minArv a2
                              novaarv = remove mini a2
```

14 – Anexo I : Tabelas das Funções

A seguir encontram-se tabelas que resumem algumas das funções mais usadas da linguagem Haskell, o resultado destas tabelas foi obtido mediante a aplicação de um trabalho realizado pelos alunos da Universidade Estácio de Sá, da disciplina de Aspectos de Linguagem de Programação que a cursaram em 2005-1.

Posteriormente no semestre de 2007-1 da mesma Universidade os alunos da mesma disciplina tiveram como tarefa ajustar todos os resultados produzidos pelos alunos do primeiro trabalho para que agora os enunciados dos exercícios não levantassem ambigüidades e que todas as respostas tivessem assinatura, e quando possível que esta fosse polimórfica.

Cada tabela contém no nome da função, e algumas células explicativas. São elas:

- Explicação resumida;
- Assinatura padrão com explicação detalhada da entrada e da saída da função;
- Observações (introduzida no segundo trabalho) sobre peculiaridades da função;
- Programação original da função no *prelude.hs;*
- Programação alternativa desenvolvida pelo aluno;
- Exemplo de utilização da função;
- Exercícios envolvendo a função em questão.

O número de exercícios varia devido a uma série de fatores, entre eles estão o número de alunos que desenvolveu um trabalho sobre a função, exercícios que eram pertinentes, exercícios repetidos, entre outros.

Os gabaritos de todos os exercícios encontram-se no anexo II, logo a seguir deste anexo.

Algumas funções não foram sorteadas e tiveram suas tabelas concluídas pelo próprio professor Rodrigo Dias, as demais contaram com a participação dos alunos que seguem na lista da próxima página. Para elucidar: MAD significa que o aluno cursava a disciplina no Campus Madureira no turno da noite, NAM quer dizer que ele era do Campus Nova América no turno da manhã e NAN que cursou no Campus Nova América no turno da noite. Os códigos 51 e 71 que acompanham as siglas significam, respectivamente, que o aluno estava numa turma do ano/semestre de 2005-1 ou 2007-1.

Cód - Função		Alunos responsáveis
1	|| (ou)	Leonardo Gonçalves (MAD-51)
2	&& (e)	Gleicemara Souza (NAN-51)
3	++	Giselly Lis (NAM-51); Reginaldo Vinhas (NAN-51)
4	!!	Thaís Araújo de Carvalho (NAM-51); Ricardo Pinto (MAD-51)
5	all	Karen Carvalho (NAM-51); Felipe dos Santos (NAN-51)
6	and	Vivian Cristina (MAD-51)
7	any	Emanuel Rodrigues (NAN-51)
8	concat	Paulo Azevedo (NAM-51); Raphael Soares (NAN-51); João de Oliveira (MAD-51)
9	cycle	Ricardo Batista (NAN-51); Felipe da Silva (MAD-51)
10	drop	Glaucilene Cunha (NAN-51)
11	elem	Márcia Maria (NAM-51); José Maurício (MAD-51)
12	filter	Cristiano Passos (NAN-51); Liliene Cardoso (MAD-51)
13	gcd	Alex Sandro (NAM-51)
14	head	Função exemplo do professor
15	init	Sergio Teixeira (NAN-51); Jaqueline Cruz (MAD-51)
16	isAlpha	Luciana Teixeira (MAD-51)
17	isAscii	Rafael Touza (NAM-51); Enzo Benvenuti (NAN-51)
18	isControl	Rafael Atílio (MAD-51)
19	isDigit	Fátima Dantas (NAN-51); Cezar Costa (MAD-51)
20	isLower	Giselle Hamano (MAD-51)
21	isPrint	Camila Oliveira (NAN-51)
22	isSpace	Juliana Lyrio (NAM-51); Fabio Bosísio (NAN-51); Peter Silva (MAD-51)
23	isUpper	Elimara Ferreira (NAM-51); André Mallmann (NAN-51); Leandro Coriolano (MAD-51)
24	last	Sergio Lessa (NAN-51)
25	lcm	Madalena Silva(NAN-51)
26	length	*Não houve aluno no sorteado...*

27	map	Eduardo Alexandre (NAN-51)
28	maximum	Wellington Pralon (NAN-51); Claudio Mayr (MAD-51)
29	minimum	Wellington Oliveira (NAN-51); Carla Vargas (MAD-51)
30	mod	Erika Ramalho (NAM-51); Dayane Fontes (NAN-51); Kelly Cunha (MAD-51)
31	notElem	Nívea Ferreira (NAM-51); Daniel Alves (NAN-51); Vagner Dias (MAD-51)
32	null	Paula Ordonnez (NAN-51); Jairo Cruz (MAD-51)
33	or	Elias Lopes (NAN-51); Sérgio Costa (MAD-51)
34	product	Cristiane Martins (NAN-51); Francisco Alex (MAD-51)
35	replicate	Adriano Cardoso (MAD-51)
36	reverse	Samuel Sarmento (NAN-51); Rogério Leal (MAD-51)
37	span	Rosa Maria (NAM-51); Marcel da Silva (NAN-51)
38	splitAt	Jéssica Cunha (NAM-51); Thiago de Oliveira (NAN-51)
39	sum	Sandro Veras (NAN-51)
40	tail	Rodrigo Pamplona (NAN-51)
41	take	*Não houve aluno no sorteado.*
42	toLower	Luis Neves (NAN-51)
43	toUpper	*Não houve aluno no sorteado.*
44	unwords	Luciana Reis (NAN-51); Felipe Silvany (MAD-51)
45	unzip	Gustavo Dias (MAD-51)
46	words	Marta Pinhão (NAM-51); Cássio Peres (NAN-51); Eduardo Garcia (MAD-51)
47	zip	Aristides Júnior (NAM-51); Raquel Máximo (NAN-51)