Pense em Hope

LÓGICA DE PROGRAMAÇÃO FUNCIONAL

EXERCÍCIOS DE PROGRAMAÇÃO FUNCIONAL COM HOPE

JOSÉ A. ALONSO JIMÉNEZ
MARÍA JOSÉ HIDALGO DOBLADO
JOSÉ AUGUSTO NAVARRO GARCIA MANZANO

LÓGICA DE PROGRAMAÇÃO FUNCIONAL

Pense em HOPE

EXERCÍCIOS DE PROGRAMAÇÃO FUNCIONAL COM HOPE

Revisão 1.5 – 20/04/2021
Revisão 1.4 – 17/04/2021
Revisão 1.3 – 13/04/2021
Revisão 1.2 – 10/04/2021
Revisão 1.1 – 05/04/2021
Revisão 1.0 – 26/03/2021

ISBN 978-65-00-19581-1

José A. Alonso Jiménez
María José Hidalgo Doblado
José Augusto Navarro Garcia Manzano

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Alonso Jiménez, José Antonio
Lógica de programação funcional : pense em Hope [livro eletrônico] : Pense em Hope : exercícios de programação funcional com Hope / José Antonio Alonso Jiménez, María José Hidalgo Doblado, José Augusto Navarro Garcia Manzano. -- 1. ed. -- São Paulo : Grupo de Lógica Computacional, 2021.
PDF

Bibliografia
ISBN 978-65-00-19581-1

1. Algoritmos de computadores 2. Exercícios 3. Hope (Linguagem de programação de computador) 4. Processamento de dados 5. Programação funcional (Computação) I. Doblado, María José Hidalgo. II. Manzano, José Augusto Navarro Garcia. III. Título.

21-60268 CDD-005.13

Índices para catálogo sistemático:

1. Linguagens de programação : Computadores : Processamento de dados 005.13

Maria Alice Ferreira - Bibliotecária - CRB-8/7964

José A. Alonso Jiménez ORCID: 0000-0002-8101-1830

María José Hidalgo Doblado ORCID: 0000-0002-3067-9363

José Augusto N. G. Manzano ORCID: 0000-0001-9248-7765

Revisão de scripts: José A. Alonso Jiménez
Diagramação, composição e capa: José Augusto N. G. Manzano
Imagens obtidas a partir dos sítios: Canvas (canva.com) e gratispng (gratispng.com).

LÓGICA DE PROGRAMAÇÃO FUNCIONAL
PENSE EM HOPE

EXERCÍCIOS DE PROGRAMAÇÃO FUNCIONAL COM HOPE

Este livro está fundamentado, adaptado e derivado a partir da obra:

Piensa en Haskell: Ejercicios de programación funcional con Haskell

Autores: José A. Alonso Jiménez
María José Hidalgo Doblado

Edição: Grupo de Lógica Computacional

Departamento de Ciências da Computação e Inteligência Artificial
Universidade de Sevilla – Espanha

Ano: 2012

OBS.:

Este livro é distribuído gratuitamente apenas em meio digital eletrônico no formato PDF.
Desejando imprima-o frente e verso em folha de papel A4 e faça sua encadernação.

Alguns scripts de funções em Hope foram adaptados e ajustados para o devido funcionamento de acordo com o que fora planejado para execução em Haskell. Outros foram adaptados devido a certas características operacionais limitantes da linguagem Hope.

Para melhor aproveitamento deste material garanta ter em mãos o arquivo **mylist.hop**, indicado no apêndice desta obra instalado em seu sistema.

Se estiver em uso o sistema operacional **Windows** copie o arquivo **mylist.hop** para a mesma pasta onde se encontra o arquivo executável **hope.exe** ou **hopeless.exe**, dependendo do ambiente utilizado. Testes realizados no Windows 10 (x64).

Se estiver em uso o sistema operacional **Linux** copie o arquivo **mylist.hop** junto ao diretório **Pasta pessoal**. Testes realizados com GNOME (Ubuntu 20.04 LTS, Fedora 33 OpenSUSE 15.2 Leap).

Para fazer uso do arquivo **mylist.hop** execute dentro do ambiente ***hope*** a instrução "***uses mylist;***".

O arquivo **mylist.hop** é distribuído em conjunto com este livro.

CARTA AO ESTUDANTE

Olá, estudante de programação funcional.

Esperamos que você esteja bem e com excelente animo para desenvolver habilidades na arte da programação de computadores dentro deste fascinante contexto. A programação funcional é um ramo da programação que segue um princípio declarativo de concepção dos códigos de programas. Neste modelo nos preocupamos no que fazer e não em como fazer. Deixamos a preocupação de como um programa fazer para o modelo imperativo, tanto estruturado quanto orientado a objetos.

Este livro veio da busca de material em língua portuguesa para o desenvolvimento de habilidades computacionais voltadas a programação funcional pura. Livros em idioma português no Brasil que deem suporte a aprendizagem da "lógica funcional pura" são raros e quase inexistentes.

O autor e professor José Augusto N. G. Manzano do Instituto Federal de São Paulo tomou conhecimento do material gratuito e disponível na grande rede mundial sobre o tema no idioma espanhol. Assim, entrou em contato com um dos autores do livro *Piensa em Haskell* e solicitou autorização para escrever um livro derivado e adaptado para a linguagem funcional Hope em português. A autorização foi prontamente dada pelo professor José A. Alonso Jiménez da Universidade de Sevilla que também acompanhou o desenvolvimento deste texto.

Este é um livro de exercícios com respostas para auxiliar a amplitude mental de como trabalhar os princípios lógicos da programação funcional. A linguagem Hope foi escolhida por ser simples e de fácil uso, servindo de suporte ao aprendizado da lógica funcional e não da linguagem. Linguagens de programação são meras ferramentas e assim devem sempre serem consideradas. A essência profissional da programação de computadores é desenvolver a capacidade mental sob o aspecto de certo paradigma de programação e não focar suas habilidades em ferramentas.

Este livro deve ser utilizado após a apresentação em aula dos detalhes operacionais da lógica funcional e da realização de alguns exercícios de aprendizagem e fixação de modo que proporcione maior visão do que é apresentado. Estude-o com calma, execute cada exercício com tranquilidade, tente entender o que de fato é apresentado e o que ocorre internamente na memória do computador.

O capítulo 1 faz introdução apresentando brevemente a linguagem Hope e ofertando dicas de como buscar a resolução de problemas a partir do método de Pólya.

No capítulo 2 são apresentadas definições e o desenvolvimento de funções elementares de cunho simplificado.

Já no capítulo 3 é dado atenção a uma série de funções definidas por recursão, sendo essa uma das atividades mais importantes na programação funcional.

O capítulo 4 tem seu foco relacionado as ações na compreensão de dados em listas de forma simulada pelo fato da linguagem Hope não trabalhar diretamente com este conceito.

No capítulo 5 tem-se o uso de recursão e compreensão usados em alguns casos simultaneamente. É uma oportunidade de ver detalhes da programação funcional sendo usados em conjunto.

O capítulo 6 apresenta outro importante princípio da programação funcional que é o uso de funções de ordem superior: quando uma função é usada como argumento de outra função.

No capítulo 7 é mostrado o uso de listas infinitas. O recurso para listas infinitas em Hope é bastante limitado. No entanto, atende adequadamente o que é mais importante ao uso deste princípio.

O capítulo 8 demonstra exemplos de aplicação de operações realizadas sobre conjuntos do ponto de vista matemático.

Além dos capítulos são apresentados nos apêndices algumas informações complementares.

Esperamos que este conteúdo possa lhe ser bastante útil.

Com grande abraço.
Os autores.

SUMÁRIO

CAPÍTULO 1 - Introdução

Este livro é uma introdução a prática da programação funcional utilizando-se como meio de estudo e apresentação a linguagem Hope. Neste capítulo são mostrados alguns detalhes e justificativas iniciais pela escolha da linguagem funcional Hope, usada na realização dos exercícios indicados e outros detalhes essenciais importantes a este estudo.

1.1 - Linguagem Hope, sua origem e legado

A linguagem Hope pode ser considerada uma ferramenta um tanto obscura, mas muito importante na evolução da programação funcional e consequentemente das linguagens funcionais atuais. Foi apresentada em 1978 como trabalho de transformação de programas junto a *Universidade de Edimburgo* por Rod Burstall, Donald Sannella e John Darlington sendo uma linguagem de programação experimental de cunho acadêmico, ou seja, não teve seu uso registrado de forma ampla nas áreas comercial e/ou científica, mas usada no universo da educação. O curioso desta linguagem é o fato de ter sido uma ferramenta que apresentou diversos elementos operacionais usados na atualidade em todas as linguagens funcionais puras existentes. A linguagem incorpora uso de módulos (bibliotecas com funcionalidades) e tipos fortes de dados que dão ao usuário condições de poder criar novos tipos de dados (BAILEY, 1990).

Hope é uma evolução da linguagem NPL sendo contemporânea a linguagem ML, também da Universidade de Edimburgo, anterior a linguagem Miranda (inclusive influenciando esta) e por conseguinte anterior a Haskell advinda de Miranda. Hope com NPL foram as duas primeiras linguagens a apresentar tipos de dados algébricos, correspondência de padrões, funções anônimas e muitos outros recursos. O interpretador da linguagem NPL efetuava (não se tem hoje conhecimento de alguma ferramenta operacional da linguagem em uso e nem da data de seu lançamento) a avaliação de cada linha de seu código da esquerda para a direita no sentido de que as condições a esquerda da correspondência de padrões a partir das variáveis definidas fossem associadas após o operador de asserção a certa operação. Um dos padrões mais envolventes da linguagem NPL era a avaliação de compreensões de conjuntos. No entanto Hope acabou abrindo mão deste recurso, o que é de certa forma lamentável, mas o efeito de compreensão de conjuntos na forma de listas ressurgiu a partir de ML e Miranda e se mantém até os dias atuais.

Apesar de Hope não ter recursos para a criação de compreensões de listas possui sintaxe leve, minimalista e simples de fácil manuseio, principalmente para iniciantes, se mostrando muito adequada para o uso escolar. Por este motivo é que a linguagem Hope foi escolhida para ser usada na adaptação dos scripts do livro *Piensa em Haskell*.

O fato de Hope não possuir recursos para a compreensão de listas faz com que soluções nesse sentido tenham que ser repensadas “fora da caixa”. Ao mesmo tempo que essa falta cria certo inconveniente, também proporciona rica oportunidade de se desenvolver soluções que venham a atender questões com outras óticas.

Na atualidade tem-se em mãos para os sistemas operacionais padrões POSIX e WIN ferramentas de interpretação para a linguagem Hope. A primeira implementação de um interpretador Hope ocorreu na Universidade de Edimburgo e não se tornou disponível de forma pública. Um tempo depois a *British Telecom* em parceria com o *Imperial College* de Londres implementou uma versão do interpretador Hope em 1986 chamada *ICHOPE* escrito por Victor Wu Way Hung.

No ano de 1985 foi publicado na revista *BYTE*, volume 10, número 8 o artigo “A HOPE TUTORIAL” de Roger Bailey mencionando um interpretador para os computadores padrão IBM-PC voltado ao sistema operacional PC-DOS 2.0.

Após o artigo publicado por Roger Bailey, entre os anos de 1998 e 1999 o professor Ross Paterson da *Universidade City* em Londres manteve a distribuição de um interpretador Hope escrito em linguagem C para sistemas operacionais padrão UNIX que se tornou referência de uso no sistema operacional FreeBSD e passou a ser usado como base para outros interpretadores descendentes.

Entre os anos de 2007 e 2012, Alexander A. Sharbarshin estendeu diversas funcionalidades do interpretador Hope escrito por Ross Paterson chamando seu produto como *Hopeless* sendo disponibilizado para os sistemas operacionais macOS (somente para a plataforma PowerPC) e Windows com o uso do programa *Cigwin*, atualizado em 2010 a partir do endereço **http://shabarshin.com/funny**.

O sistema operacional Windows possui uma versão do interpretador denominada *Hope for Windows* desenvolvido por Marco Alfaro lançada em 2012 a partir do código fonte do Professor Ross Paterson, escrito no ambiente de desenvolvimento *Visual Studio*, tendo como última data de atualização o ano de 2014 (**http://hopelang.blogspot.com**). O interpretador *Hope for Windows* poderá ser adquirido a partir do endereço: **http://www.hope.manzano.pro.br**.

Além do sistema operacional Windows há a possibilidade de uso da linguagem junto a sistemas operacionais padrão POSIX, como Linux. Neste caso é necessário obter os arquivos e proceder sua compilação e instalação a partir de **https://github.com/dmbaturin/hope**, atualizado em 2014.

Cabe ressaltar que apesar de funcional os interpretadores (e não a linguagem em si) *Hopeless* e *Hope for Windows* são ferramentas simples podendo apresentar bugs de funcionamento que no geral não atrapalham em demasia seu uso operacional.

É importante considerar que Hope possui algumas desvantagens adicionais além do que foi exposto que necessitam ser levadas em consideração, destacando-se o fato de não ser muito conhecida, de possuir pouquíssima documentação disponível e de ter a ausência de bibliotecas ou módulos com funcionalidades prontas para a manipulação de listas, como: head, tail, init, last, drop, zip e etc.

A ausência de bibliotecas com funcionalidades básicas pode se tornar uma vantagem técnica, muito interessante, pois obriga o usuário da linguagem a desenvolver esse ferramental servindo esta ocorrência de grande incentivo e oportunidade ao aprendizado e fixação de diversas técnicas de programação usadas no universo funcional.

No sentido de manter este livro o mais próximo possível do trabalho dos autores José A. Alonso Jiménez e María José Hidalgo Doblado é apresentada em seu apêndice a biblioteca de operação suplementar **mylist.hop** que para ser usada basta ser chamada no *prompt* do ambiente de programação a partir da instrução “**uses mylist;**”, desde que o interpretador e a biblioteca estejam no mesmo diretório ou pasta.

Os scripts aqui apresentados foram implementados a partir da adaptação dos códigos originalmente escritos em Haskell e transcritos na linguagem Hope por tradução ou transliteração dependendo do caso. Em certos momentos, devido a características da linguagem Hope tornou-se necessário realizar adaptações e ajustes mais radicais a fim de acomodar os objetivos deste trabalho frente ao trabalho original *Piensa em Haskell*.

https://en.wikipedia.org/wiki/Hope_(programming_language)
https://en.wikipedia.org/wiki/NPL_(programming_language)
http://www.stolyarov.info/static/hope/ref_man/ref_man.html

1.2 - Método de Pólya para a resolução de problemas matemáticos

Para resolver um problema é necessário (POLYA, 1978, p. 19):

Passo 1: Entender o problema

- Qual é a incógnita? Quais são os dados?
- Qual é a condição? A condição é suficiente para determinar a incógnita? É suficiente? Redundante? Contraditória?

Passo 2: Configurar um plano

- Você já se deparou com um problema semelhante? Ou você já viu o mesmo problema colocado de forma ligeiramente diferente?
- Você conhece algum problema relacionado a este? Você conhece algum teorema que pode ser útil? Olhe atentamente para o desconhecido e tente se lembrar de um problema que lhe é familiar e que seja semelhante.
- Aqui está um problema relacionado ao seu e que você já resolveu. Você pode usá-lo? Você pode usar seu resultado? Você pode usar o método deles? Te faz falta introduzir um elemento auxiliar para poder utilizá-lo?
- Você pode expor o problema de outra maneira? Você pode colocá-lo de outra forma novamente? Use suas definições.
- Se você não conseguir resolver o problema proposto, tente resolver algum problema semelhante primeiro. Você pode imaginar um problema análogo um pouco mais acessível? Um problema mais geral? Um problema mais específico? Um problema análogo? Você pode resolver parte do problema? Considere apenas parte da condição: descarte a outra parte; Até que ponto o desconhecido está determinado agora? Como isso pode variar? Você pode deduzir quaisquer elementos que sejam úteis dos dados? Você consegue pensar em alguns outros dados apropriados para determinar o desconhecido? Você pode mudar o desconhecido? Você pode mudar o desconhecido, os dados ou ambos se assim for necessário para que fiquem mais próximos um do outro?
- Você usou todos os dados? Você usou toda a condição? Você considerou todas as noções essenciais sobre o problema?

Passo 3: Executar o plano

- Ao exercitar seu plano de solução, verifique cada uma das etapas.
- Você pode ver claramente que a etapa está correta? Você pode provar isso?

Passo 4: Examinar a solução obtida

- Você pode verificar o resultado? Você pode raciocinar?
- Você pode obter o resultado de forma diferente? Você pode ver tudo de uma única vez? Você pode usar o resultado ou método em algum outro problema?

1.3 - Método de Pólya para a resolução de problemas de programação

Para resolver um problema é necessário (THOMPSON, 1996 adaptado de POLYA, 1978):

Passo 1: Entender o problema

- Quais são os argumentos? Qual é o resultado? Qual é o nome da função? Qual é o seu tipo?

- Qual é a especificação do problema? A especificação pode ser satisfeita? É insuficiente? Redundante? Contraditória? Quais restrições são assumidas em argumentos e nos resultados?

- Você pode dividir o problema em partes? Pode ser que seja útil desenhar diagramas com exemplos de argumentos e resultados.

Passo 2: Desenhar o programa

- Você já se deparou com um problema semelhante? Ou você já viu o mesmo problema colocado de forma ligeiramente diferente?

- Você conhece algum problema relacionado a este? Você conhece alguma função que pode ser útil? Avalie atentamente a sua experiência e tente se lembrar de um problema que lhe é familiar e que possui o mesmo tipo ou lhe seja semelhante.

- Você conhece algum problema familiar com especificações semelhantes?

- Aqui está um problema relacionado ao seu que você já resolveu. Você pode usar isso? Você pode usar seu resultado? Você pode usar o mesmo método de solução? Você precisa apresentar qualquer função auxiliar para usá-lo?

- Se você não conseguir resolver o problema proposto, tente resolver algum problema primeiro que seja similar. Você pode imaginar um problema análogo ou um pouco mais acessível? Um problema mais geral? Um problema mais específico?

- Você pode resolver parte do problema? Você pode deduzir qualquer elemento útil dos dados? Você consegue pensar em quaisquer outros dados apropriados para determinar o desconhecido? Você pode mudar o desconhecido? Você pode mudar o desconhecido, os dados ou ambos se necessário para que fiquem mais próximos um do outro?

- Você usou todos os dados? Você usou todas as restrições dos dados? Você considerou todos os requisitos da especificação?

Passo 3: Escrever o programa

- Conforme você escreve o programa verifique cada uma das etapas e funções auxiliares.

- Você pode ver claramente se cada etapa ou função auxiliar está correta?

- Você pode escrever o programa em etapas? Você pensa sobre os diferentes casos em que o problema está dividido. Em particular você pensa sobre os diferentes casos dos dados? Você pode pensar em calcular os casos de forma independente e juntá-los para obter um resultado final.

- Você pode pensar na solução para o problema dividindo-o em problemas menores com dados mais simples e juntando-os as soluções parciais para obter a solução final sem dificuldades por meio de recursão?

- Em seu projeto você pode usar problemas mais gerais ou mais específicos? Escreva as soluções para esses problemas. Eles podem servir como um guia para a solução do problema original ou de sua solução?

- Você pode se apoiar em outros problemas que já resolveu? Eles podem ser usados? Podem ser modificados? Você pode orientar a solução do problema original?

Passo 4: Examinar a solução obtida

- Você pode verificar a operação do programa a partir do uso de uma coleção de argumentos?

- Você pode verificar as propriedades do programa?

- Você pode escrever o programa de uma maneira diferente?

- Você pode usar o programa ou o método de desenvolvimento em algum outro programa?

ANOTAÇÕES

CAPÍTULO 2 - Definições elementares de funções

Neste capítulo são propostos exercícios de funções com definições elementares (não recursivas). Esses exercícios estão baseados nos 4 primeiros capítulos (ALONSO & MANZANO, 2021), além de outros exercícios acrescidos e fundamentados nos capítulos 5, 6 e 7 da mesma obra e de Alonso (2019).

2.1 - Média de 3 números

(**Exercício 2.1.1**) Definir a função *media3* tal que [media3(x, y, z);] seja a média aritmética dos números x, y e z. Por exemplo,

```
media3(1,3,8);  .....................  4
media3(0−1,0,7);  ...................  2
media3(0−3,0,3);  ...................  0
```

Solução:

```
media3 : num # num # num -> num;
media3 (x, y, z) <= (x + y + z) / 3;
```

2.2 - Soma de reais de uma coleção de moedas

(**Exercício 2.2.1**) Definir a função *somaMoedas* tal que [somaMoedas(a, b, c, d, e);] seja a soma em reais (R$) correspondentes aos valores: “a” para R$ 1,00, “b” para R$ 2,00, “c” para R$ 5,00, “d” para R$ 10 ,00 e “e” para R$ 20,00. Por exemplo,

```
somaMoedas(0,0,0,0,1); ..............   20
somaMoedas(0,0,8,0,3); ..............  100
somaMoedas(1,1,1,1,1); ..............   38
```

Solução:

```
somaMoedas : num # num # num # num # num -> num;
somaMoedas (a, b, c, d, e) <= 1 * a + 2 * b + 5 * c + 10 * d + 20 * e;
```

2.3 - Volume de uma esfera

(**Exercício 2.3.1**) Definir a função *volumeEsfera* tal que [volumeEsfera(r);] seja o volume da esfera de raio “r”. Por exemplo,

```
volumeEsfera(10);  ..................  4188.79020478639
volumeEsfera(5);  ...................   523.598775598299
volumeEsfera(6.5);  .................  1150.34650998946
```

Indicação: Usar o valor de “pi” a partir da operação - *acos(0) * 2*, se necessário.

Solução:

```
volumeEsfera : num -> num;
volumeEsfera r <= 4 / 3 * pi * pow (r, 3);
```

2.4 - Área de uma coroa circular

(**Exercício 2.4.1**) Definir a função *areaDeCoroaCircular* tal que [areaDeCoroaCircular(r1,r2);] seja a área de uma coroa circular de raio interno "r1" e raio externo "r2". Por exemplo,

```
areaDeCoroaCircular(1,2); ...........   9.42477796076938
areaDeCoroaCircular(2,5); ...........  65.9734457253857
areaDeCoroaCircular(3,5); ...........  50.2654824574367
```

Solução:

```
areaDeCoroaCircular : num # num -> num;
areaDeCoroaCircular (r1, r2) <= pi * (pow (r2, 2) - pow (r1, 2));
```

2.5 - Último dígito de um número

(**Exercício 2.5.1**) Definir a função *ultimoDigito* tal que [ultimoDigito(x);] seja o último dígito do número "x". Por exemplo,

```
ultimoDigito(325); ..................  5
ultimoDigito(128); ..................  8
ultimoDigito(15); ...................  5
```

Solução:

```
ultimoDigito : num -> num;
ultimoDigito x <= x mod 10;
```

2.6 - Máximo de três valores

(**Exercício 2.6.1**) Definir a função *maxTres* tal que [maxTres(x,y,z);] seja o máximo de "x", "y" e "z". Por exemplo,

```
maxTres(6,2,4); .....................  6
maxTres(6,7,4); .....................  7
maxTres(6,7,9); .....................  9
```

Solução:

```
maxTres : num # num # num -> num;
maxTres (x, y, z) <= max (x, max (y, z));
```

2.7 - Disjunção exclusiva

A disjunção exclusiva "xor" entre duas condições é verificada se uma é verdadeira e a outra é falsa.

(**Exercício 2.7.1**) Definir a função *xor1* que calcule a disjunção exclusiva da tabela verdade. Use 4 equações, uma para cada linha da tabela. Por exemplo,

```
xor1(1=1,2=2); ......................  false
xor1(1=1,1=2); ......................  true
xor1(1=2,2=2); ......................  true
xor1(1=2,2=1); ......................  false
```

Solução:

```
xor1 : truval # truval -> truval;
xor1 (true, true)   <= false;
xor1 (true, false)  <= true;
xor1 (false, true)  <= true;
xor1 (false, false) <= false;
```

(**Exercício 2.7.2**) Definir a função *xor2* que calcule a disjunção exclusiva da tabela verdade e uso de padrões. Use 2 equações, uma para cada valor do primeiro argumento.

Solução:

```
xor2 : truval # truval -> truval;
xor2 (true, y)  <= not y;
xor2 (false, y) <= y;
```

(**Exercício 2.7.3**) Definir a função *xor3* que calcule a disjunção exclusiva da tabela verdade a partir da disjunção (or), conjunção (and) e negação (not). Use 1 equação.

Solução:

```
xor3 : truval # truval -> truval;
xor3 (x, y) <= (x or y) and not (x and y);
```

(**Exercício 2.7.4**) Definir a função *xor4* que calcule a disjunção exclusiva da tabela verdade a partir da desigualdade (/=). Use 1 equação.

Solução:

```
xor4 : truval # truval -> truval;
xor4 (x, y) <= x /= y;
```

2.8 - Rotação de listas

(**Exercício 2.8.1**) Definir a função *rota* tal que [rota(xs);] seja a lista obtida colocando o primeiro elemento de "xs" no final da lista. Por exemplo,

```
rota([3,2,5,7]); ....................  [2,5,7,3]
rota([9,7,6,5]); ....................  [7,6,5,9]
rota([8,2,9,7]); ....................  [2,9,7,8]
```

Solução:

```
rota : list num -> list num;
rota xs <= tail xs <> [head xs];
```

(**Exercício 2.8.2**) Definir a função *rota'* tal que [rota'(n,xs);] seja a lista obtida colocando os primeiros "n" elementos de "xs" no final da lista. Por exemplo,

```
rota'(1,[3,2,5,7]); .................  [2,5,7,3]
rota'(2,[3,2,5,7]); .................  [5,7,3,2]
rota'(3,[3,2,5,7]); .................  [7,3,2,5]
```

Solução:

```
rota' : num # list num -> list num;
rota' (n, xs) <= drop (n, xs) <> take (n, xs);
```

2.9 - Faixa de uma lista

(**Exercício 2.9.1**) Definir a função *mostraMenorMaior* tal que [mostraMenorMaior(xs);] seja a lista formada pelo menor e maior elementos de “xs”. Por exemplo,

```
mostraMenorMaior([3,2,7,5]); ........  [2,7]
mostraMenorMaior([9,7,6,5]); ........  [5,9]
mostraMenorMaior([8,2,6,7]); ........  [2,8]
```

Solução:

```
mostraMenorMaior : list num -> list num;
mostraMenorMaior xs <= [minimum xs, maximum xs];
```

2.10 - Reconhecimento de palíndromos

(**Exercício 2.10.1**) Definir a função *palindromo* tal que [palindromo(xs);] verifique se “xs” é um palíndromo, ou seja, se ler “xs” da esquerda para a direita é o mesmo que ler da direita para a esquerda. Por exemplo,

```
palindromo([3,2,5,2,3]); ............  true
palindromo([3,2,5,6,2,3]); ..........  false
```

Solução:

```
palindromo : list num -> truval;
palindromo xs <= xs = reverse xs;
```

2.11 - Elementos internos de uma lista

(**Exercício 2.11.1**) Definir a função *interior* tal que [interior(xs);] seja a lista obtida eliminando os extremos da lista “xs”. Por exemplo,

```
interior([2,5,3,7,3]); ..............  [5,3,7]
interior(2..7); .....................  [3,4,5,6]
```

Solução:

```
interior : list num -> list num;
interior xs <= tail (init xs);
```

2.12 - Final de uma lista

(**Exercício 2.12.1**) Definir a função *final* tal que [final(n,xs);] seja a lista formada pelos “n” finais elementos de “xs”. Por exemplo,

```
final(3,[2,5,4,7,9,6]); .............  [7,9,6]
final(2,[2,5,4,7,9,6]); .............  [9,6]
```

Solução:

```
final : num # list num -> list num;
final (n, xs) <= drop (length xs - n, xs);
```

2.13 - Segmentos de uma lista

(**Exercício 2.13.1**) Definir a função *segmento* tal que [segmento(m,n,xs);] seja a lista dos elementos de “xs” compreendidos entre as posições “m” e “n”. Por exemplo,

```
segmento(3,4,[3,4,1,2,7,9,0]); ......  [1,2]
segmento(3,5,[3,4,1,2,7,9,0]); ......  [1,2,7]
segmento(5,3,[3,4,1,2,7,9,0]); ......  nil (equivale a [])
```

Solução:

```
segmento : num # num # list num -> list num;
segmento (m, n, xs) <= drop (m - 1, take (n, xs));
```

2.14 - Extremos de uma lista

(**Exercício 2.14.1**) Definir a função *extremos* tal que [extremos(n,xs);] seja a lista formada pelos “n” primeiros elementos de “xs” e os “n” finais elementos de “xs”. Por exemplo,

```
extremos(3,[2,6,7,1,4,5,8,9,3]); ....  [2,6,7,8,9,3]
extremos(2,[2,6,7,1,4,5,8,9,3]); ....  [2 6,9,3]
```

Solução:

```
extremos : num # list num -> list num;
extremos (n, xs) <= take (n, xs) <> drop (length xs - n, xs);
```

2.15 - Mediana de 3 números

(**Exercício 2.15.1**) Definir a função *mediana* tal que [mediana(x,y,z);] seja o número mediano dos três números “x”, “y” e “z”. Por exemplo,

```
mediana(3,2,5); .....................  3
mediana(2,4,5); .....................  4
mediana(2,6,5); .....................  5
mediana(2,6,6); .....................  6
```

Solução 1:

```
mediana : num # num # num -> num;
mediana (x, y, z) <= x + y + z - minimum ([x, y, z]) - maximum ([x, y, z]);
```

Solução 2:

```
mediana' : num # num # num -> num;
mediana' (x, y, z) <= if a >= x and x >= b then x else
                      if a >= y and y >= b then y else x + y + z - a - b
                      where a == minimum ([x, y, z])
                      where b == maximum ([x, y, z]);
```

2.16 - Igualdade e diferença de 3 elementos

(**Exercício 2.16.1**) Definir a função *tresIguais* tal que [tresIguais(x,y,z);] verifique se os elementos "x", "y" e "z" são iguais. Por exemplo,

```
tresIguais(4,4,4); ..................  true
tresIguais(4,3,4); ..................  false
```

Solução:

```
tresIguais : num # num # num -> truval;
tresIguais (x, y, z) <= x = y and y = z;
```

(**Exercício 2.16.2**) Definir a função *tresDiferentes* tal que [tresDiferentes(x,y,z);] verifique se os elementos "x", "y" e "z" não são iguais. Por exemplo,

```
tresDiferentes(3,5,2); ..............  true
tresDiferentes(3,5,3); ..............  false
```

Solução:

```
tresDiferentes : num # num # num -> truval;
tresDiferentes (x, y, z) <= x /= y and x /= z and y /= z;
```

2.17 - Igualdade e diferença de 4 elementos

(**Exercício 2.17.1**) Definir a função *quatroIguais* tal que [quatroIguais(x,y,z,u);] verifique se os elementos "x", "y", "z" e "u" são iguais. Por exemplo,

```
quatroIguais(5,5,5,5); ..............  true
quatroIguais(5,5,4,5); ..............  false
```

Solução:

```
quatroIguais : num # num # num # num -> truval;
quatroIguais (x, y, z, u) <= x = y and tresIguais (y, z, u);
```

2.18 - Propriedade triangular

Os comprimentos dos lados de um triângulo não podem ser qualquer valor. Para que um triângulo possa ser construído, a propriedade triangular deve ser respeitada, ou seja, o comprimento cada lado tem que ser menor que a soma dos outros dois lados.

(**Exercício 2.18.1**) Definir a função *triangulo* tal que [triangulo(a,b,c);] verifique se "a", "c" e "c" correspondem a regra da propriedade trinagular. Por exemplo,

```
triangulo(3,4,5); ...................  true
triangulo(30,4,5); ..................  false
triangulo(3,40,5); ..................  false
triangulo(3,4,50); ..................  false
```

Solução:

```
triangulo : num # num # num -> truval;
triangulo (a, b, c) <= a < b + c and b < a + c and c < a + b;
```

2.19 - Divisão segura

(**Exercício 2.19.1**) Definir a função *divisaoSegura* tal que [divisaoSegura(x,y);] seja “x/y” se o “y” não for zero e 9999 caso contrário. Por exemplo,

```
divisaoSegura(7,2); .................  3.5
divisaoSegura(7,0); .................  9999
```

Solução:

```
divisaoSegura : num # num -> num;
divisaoSegura (_, 0) <= 9999;
divisaoSegura (x, y) <= x / y;
```

2.20 - Tipo de um triângulo

(**Exercício 2.20.1**) Definir a função *tipoTriangulo* tal que [tipoTriangulo(a,b,c);] mostre se possível os tipos de triângulos formados (equilátero, isósceles e escaleno) pelos lados “a”, “b” e “c”. Por exemplo,

```
tipoTriangulo(3,3,3); ...............  "Triangulo equilatero"
tipoTriangulo(3,3,2); ...............  "Triangulo isosceles"
tipoTriangulo(3,4,5); ...............  "Triangulo escaleno"
tipoTriangulo(1,3,4); ...............  Medidas nao formam triangulo!
```

Solução:

```
tipoTriangulo : num # num # num -> list char;
tipoTriangulo (a, b, c) <= if triangulo (a, b, c)
                           then if a = b and b = c
                                then "Triangulo equilatero"
                                else if a = b or a = c or c = b
                                     then "Triangulo isosceles"
                                     else "Triangulo escaleno"
                           else error ("Medidas nao formam triangulo!");
```

2.21 - Norma de um vetor

(**Exercício 2.21.1**) Definir a função *modulo* tal que [modulo(x,y);] represente o vetor “v” como |v| a partir da obtenção da distância entre os pontos “x” e “y”. Por exemplo,

```
modulo(3,4); ........................  5
modulo(2.5,6.3); ....................  6.77790528113222
```

Solução:

```
modulo : num # num -> num;
modulo (x, y) <= sqrt (pow (x, 2) + pow (y, 2));
```

2.22 - Retângulo de área máxima

(**Exercício 2.22.1**) As dimensões dos retângulos podem ser representadas em pares de valores como (5,3), em que 5 é a base e 3 é a altura. Defina a maior área de um retângulo a partir da função *maiorRetangulo* tal que [maiorRetangulo(r1,r2)] represente o retângulo com a maior área entre “r1” e “r2”. Por exemplo,

```
maiorRetangulo((4,6),(3,7)); ........  (4,6)
maiorRetangulo((4,6),(3,8)); ........  (4,6)
maiorRetangulo((4,6),(3,9)); ........  (3,9)
```

Solução:

```
maiorRetangulo : (num # num) # num # num -> num # num;
maiorRetangulo ((a, b), c, d) <= if a * b >= c * d then (a, b) else (c, d);
```

2.23 - Pontos de um plano

Os pontos de um plano podem ser representados por um par de números que são suas coordenadas:

1 - Quadrante de um ponto
2 - Intercambio de coordenadas
3 - Ponto simétrico
4 - Distância entre pontos
5 - Ponto médio entre dois pontos

(**Exercício 2.23.1**) Definir a função *quadrante* tal que [quadrante(p);] seja o quadrante do ponto “p”, assumindo que “p” não faz parte dos eixos. Por exemplo,

```
quadrante(3,5); .....................  1
quadrante(0-3,5); ...................  2
quadrante(0-3,0-5); .................  3
quadrante(3,0-5); ...................  4
```

Solução:

```
quadrante : num # num -> num;
quadrante (x, y) <= if x > 0 and y > 0 then 1 else
                    if x < 0 and y > 0 then 2 else
                    if x < 0 and y < 0 then 3 else
                    if x > 0 and y < 0 then 4 else 0;
```

(**Exercício 2.23.2**) Definir a função *intercambio* tal que [intercambio(p);] seja a obtenção de um ponto a partir da troca das coordenadas do ponto “p”. Por exemplo,

```
intercambio(2,5); ...................  (5,2)
intercambio(5,2); ...................  (2,5)
```

Solução:

```
intercambio : num # num -> num # num;
intercambio (x, y) <= (y, x);
```

(**Exercício 2.23.3**) Definir a função simetricoH tal que [simetricoH(p);] seja o ponto simétrico de “p” em relação ao eixo horizontal. Por exemplo,

```
simetricoH(2,5); ....................  (2,-5)
simetricoH(2,0-5); ..................  (2,5)
```

Solução:

```
simetricoH : num # num -> num # num;
simetricoH (x, y) <= (x, 0 - y);
```

(**Exercício 2.23.4**) Definir a função *distancia* tal que [distancia(p1,p2);] seja a distância entre os pontos “p1” e “p2”. Por exemplo,

```
distancia((1,2),(4,6)); .............  5
```

Solução:

```
distancia : (num # num) # num # num -> num;
distancia ((x1, y1), x2, y2) <= sqrt (pow (x1 - x2, 2) + pow (y1 - y2, 2));
```

(**Exercício 2.23.5**) Definir a função *pontoMedio* tal que [pontoMedio(p1,p2);] seja o ponto médio entre os pontos “p1” e “p2”. Por exemplo,

```
pontoMedio((0,2),(0,6)); ............  (0,4)
pontoMedio((0-1,2),(7,6)); ..........  (3,4)
```

Solução:

```
pontoMedio : (num # num) # num # num -> num # num;
pontoMedio ((x1, y1), x2, y2) <= ((x1 + x2) / 2, (y1 + y2) / 2);
```

2.24 - Números complexos

Os números complexos podem ser representados por pares de números, por exemplo, o número 2 + 5i pode ser representado pelo par (2,5).

1 - Soma de números complexos
2 - Produto de números complexos
3 - Conjugado de um número complexo

(**Exercício 2.24.1**) Definir a função *somaComplexo* tal que [somaComplexo(x,y);] seja a soma dos números complexos “x” e “y”. Por exemplo,

```
somaComplexo((2,3),(5,6)); ..........  (7,9)
somaComplexo((1,2),(1,3)); ..........  (2,5)
```

Solução:

```
somaComplexo : (num # num) # num # num -> num # num;
somaComplexo ((a, b), c, d) <= (a + c, b + d);
```

(**Exercício 2.24.2**) Definir a função *produtoComplexo* tal que [produtoComplexo(x,y);] seja a multiplicação dos números complexos “x” e “y”. Por exemplo,

```
produtoComplexo((2,3),(5,6)); .......  (-8,27)
```

Solução:

```
produtoComplexo : (num # num) # num # num -> num # num;
produtoComplexo ((a, b), c, d) <= (a * c - b * d, a * d + b * c);
```

(**Exercício 2.24.3**) Definir a função *conjulgado* tal que [conjulgado(z);] seja o conjulgado do número complexo “z”. Por exemplo,

```
conjulgado(2,3); ...................  (2,-3)
```

Solução:

```
conjulgado : num # num -> num # num;
conjulgado (a, b) <= (a, 0 - b);
```

2.25 - Intercalação de pares

(**Exercício 2.25.1**) Definir a função *intercala* tal que [intercala(xs,ys);] receba duas listas “xs” e “ys” com dois elementos cada e devolva uma lista com quatro elementos, construída intercalando os elementos de “xs” e “ys”. Por exemplo,

```
intercala([1,4],[3,2]); ............  [1,3,4,2]
```

Solução:

```
intercala : list num # list num -> list num;
intercala ([x1, x2], [y1, y2]) <= [x1, y1, x2, y2];
```

2.26 - Permuta cíclica de uma lista

(**Exercício 2.26.1**) Definir a função *ciclo* tal que [ciclo(xs);] receba uma lista “xs” e permute ciclicamente os seus elementos passando o último elemento da lista para seu início. Por exemplo,

```
ciclo([2,5,7,9]); ..................  [9,2,5,7]
ciclo([]); .........................  nil (para [])
ciclo([2]); ........................  [2]
```

Solução:

```
ciclo : list num -> list num;
ciclo [] <= [];
ciclo xs <= last xs :: init xs;
```

2.27 - O maior número de 2 dígitos a partir de dois dígitos fornecidos

(**Exercício 2.27.1**) Definir a função *maiorNumero* tal que [maiorNumero(x,y);] seja o maior número entre dois dígitos que podem ser construídos a partir dos dígitos “x” e “y”. Por exemplo,

```
numeroMaior(2,5); ...................  52
numeroMaior(5,2); ...................  52
```

Solução:

```
numeroMaior : num # num -> num;
numeroMaior (x, y) <= a * 10 + b
                      where a == max (x, y)
                      where b == min (x, y);
```

2.28 - Quantidade de raízes de uma equação quadrática (2º grau)

(**Exercício 2.28.1**) Definir a função *numeroDeRaizes* tal que [numeroDeRaizes(a,b,c);] represente o número de raízes reais para uma equação ax^2 + bx + c = 0. Por exemplo,

```
numeroDeRaizes(2,0,3); ..............  0
numeroDeRaizes(4,4,1); ..............  1
numeroDeRaizes(5,23,12); ............  2
```

Solução:

```
numeroDeRaizes : num # num # num -> num;
numeroDeRaizes (a, b, c) <= if d < 0 then 0 else
                            if d = 0 then 1 else 2
                            where d == pow (b, 2) - 4 * a * c;
```

2.29 - Raízes de uma equação quadrática (2º grau)

(**Exercício 2.29.1**) Definir a função *raizes* tal que [raizes(a,b,c);] devolva na forma de lista os valores das raízes reais da equação ax^2 + bx + c = 0. Por exemplo,

```
raizes1(1,0-2,1); ...................  [1,1]
raizes1(1,3,2); .....................  [-1,-2]
raizes2(1,0-2,1); ...................  [1,1]
raizes2(1,3,2); .....................  [-1,-2]
raizes2(1,2,3); .....................  Nao existem raizes reais
```

Solução 1:

```
raizes1 : num # num # num -> list num;
raizes1 (a, b, c) <= [(0 - b + d) / t, (0 - b - d) / t]
                     where d == sqrt (pow (b, 2) - 4 * a * c)
                     where t == 2 * a;
```

Solução 2:

```
raizes2 : num # num # num -> list num;
raizes2 (a, b, c) <=
    if d >= 0 then [(0 - b + e) / (2 * a), (0 - b - e) / (2 * a)]
    else error ("Nao existem raizes reais")
    where e == sqrt d
    where d == pow (b, 2) - 4 * a * c;
```

2.30 - Área de um triângulo mediante a fórmula de Herón

(**Exercício 2.30.1**) Em geometria, a fórmula descoberta por Herón de Alexandria, diz que a área de um triângulo cujos lados medem “a”, “b” e “c” é a raiz quadrada de s*(s - a)*(s - b)*(s - c), onde “s” é o semiperímetro a partir de “s = (a + b + c) / 2”. Definir a função *area* tal que [area(a,b,c);] é a área de um triângulo com lados “a”, “b” e “c”. Por exemplo,

```
area(3,4,5); ........................  6
```

Solução:

```
area : num # num # num -> num;
area (a, b, c) <= sqrt (s * (s - a) * (s - b) * (s - c))
                 where s == (a + b + c) / 2;
```

2.31 - Números racionais como pares de inteiros

Os números racionais podem ser representados por pares de números inteiros. Por exemplo, o número 2/5 pode ser representado pelo par (2,5).

1 - Forma reduzida de um número racional
2 - Soma de dois números racionais
3 - Produto de dois números racionais
4 - Igualdade de números racionais

(**Exercício 2.31.1**) Definir a função *formaReduzida* tal que [formaReduzida(x);] seja a forma reduzida do número racional “x”. Por exemplo,

```
formaReduzida(4,10); ................  (2,5)
```

Solução:

```
formaReduzida : num # num -> num # num;
formaReduzida (a, b) <= (a div c, b div c) where c == gcd (a, b);
```

(**Exercício 2.31.2**) Definir a função *somaRacional* tal que [somaRacional(x,y);] seja a soma dos números racionais “x” e “y”. Por exemplo,

```
somaRacional((2,3),(5,6)); ..........  (3,5)
```

Solução:

```
somaRacional : (num # num) # num # num -> num # num;
somaRacional ((a, b), c, d) <= formaReduzida (a * d + b * c, b * d);
```

(**Exercício 2.31.3**) Definir a função *produtoRacional* tal que [produtoRacional(x,y);] seja a multiplicação dos números racionais “x” e “y”. Por exemplo,

```
produtoRacional((2,3),(5,6)); .......  (5,9)
```

Solução:

```
produtoRacional : (num # num) # num # num -> num # num;
produtoRacional ((a, b), c, d) <= formaReduzida (a * c, b * d);
```

(**Exercício 2.31.4**) Definir a função *igualdadeoRacional* tal que [igualdadeRacional(x,y);] seja a verificação se os números racionais “x” e “y” são iguais. Por exemplo,

```
igualdadeRacional((6,9),(10,15)); ...  true
igualdadeRacional((6,9),(11,15)); ...  false
```

Solução:

```
igualdadeRacional : (num # num) # num # num -> truval;
igualdadeRacional ((a, b), c, d) <=
                   formaReduzida (a, b) = formaReduzida (c, d);
```

2.32 - Múltiplo de "n" sobre "m"

(**Exercício 2.32.1**) Definir a função *multiplo* tal que [multiplo(n,m);] informe se dado número “n” é múltiplo de “m”. Por exemplo,

```
multiplo(8,4); ......................  true
multiplo(5,2); ......................  false
multiplo(9,3); ......................  true
multiplo(9,6); ......................  false
```

Solução:

```
multiplo : num # num -> truval;
multiplo (n, m) <= n mod m = 0;
```

1.33 - Média dos elementos de uma lista

(**Exercício 1.33.1**) Definir a função *media* tal que [media([xs]);] apresente o valor da média aritmética dos elementos de uma lista “xs”. Por exemplo,

```
media([1,2,3]); .....................  2
media([1,1.5,2.75,3]); ..............  2.0625
```

Solução:

```
media : list num -> num;
media xs <= sum xs / length xs;
```

ANOTAÇÕES

Capítulo 3 – Definições por recursão

Este capítulo apresenta exercícios com definições por recursão. Os exercícios apresentados estão baseados nos capítulos 5 e 7 (ALONSO & MANZANO, 2021), além de exercícios acrescidos e outros exercícios referenciados no capítulo 8 da obra indicada.

3.1 - Potência de exponente natural

(**Exercício 3.1.1**) Definir por recursão a função *potencia* tal que [potencia(x,n);] seja o valor de "x" elevado a "n". Por exemplo,

```
potencia(2,3); ......................  8
```

Solução:

```
potencia : num # num -> num;
potencia (x, 0) <= 1;
potencia (x, n) <= x * potencia (x, n - 1);
```

3.2 - Repetição (replicar) de um elemento

(**Exercício 3.2.1**) Definir por recursão a função *repetir* tal que [repetir(n,x);] seja o valor de "x" repetido "n" vezes na lista. Por exemplo,

```
repetir(3,2); .......................  [2,2,2]
```

Solução:

```
repetir : num # num -> list num;
repetir (0, _)     <= [];
repetir (n + 1, x) <= x :: repetir (n, x);
```

3.3 - Duplo fatorial de um número

(**Exercício 3.3.1**) O duplo fatorial de um número "n" se define por

```
0!! = 1
1!! = 1
n!! = n * (n - 2) * ... * 3 * 1, si "n" é impar
n!! = n * (n - 2) * ... * 4 * 2, si "n" é par
```

Por exemplo,

```
8!! = 8 * 6 * 4 * 2     = 384
9!! = 9 * 7 * 5 * 3 * 1 = 945
```

Definir por recursão a função *duploFatorial* tal que [duploFatorial(n);] seja o duplo fatorial de "n". Por exemplo,

```
duploFatorial(8); ...................  384
duploFatorial(9); ...................  945
```

Solução:

```
duploFatorial : num -> num;
duploFatorial 0 <= 1;
duploFatorial 1 <= 1;
duploFatorial n <= n * duploFatorial (n - 2);
```

3.4 - Algoritmo de Euclides para o máximo divisor comum

(**Exercício 3.4.1**) Dados dois números naturais, “a” e “b”, é possível calcular seu maior divisor comum usando o Algoritmo de Euclides. Este algoritmo resumi-se na seguinte fórmula:

$$mdc(a,b) = \begin{cases} a, & se\, b = 0 \\ mdc(b, a\ módulo\ b), & se\, b > 0 \end{cases}$$

Definir por recursão a função *mdc* tal que [mdc(a,b);] seja o máximo divisor comum de “a” e “b” calculado mediante o Algoritmo de Euclides. Por exemplo,

```
mdc(30,45); .........................  15
```

Solução:

```
mdc : num # num -> num;
mdc (a, 0) <= a;
mdc (a, b) <= mdc (b, a mod b);
```

3.5 - Menor número divisível por uma sequência de números

(**Exercício 3.5.1**) Definir por recursão a função *menorDivisivel* tal que [menorDivisivel(a,b);] seja o menor número divisível entre “a” e “b”. Por exemplo,

```
menorDivisivel(2,5); ................  60
```

Obs. Usar a função *lcm*, tal que “lcm(x,y);” é o mínimo múltiplo comum de “x” e “y”.

Solução:

```
menorDivisivel : num # num -> num;
menorDivisivel (a, b) <= if a = b
                         then a
                         else lcm (a, menorDivisivel (a + 1, b));
```

(**Exercício 3.5.2**) Definir a constante *euler5* tal que [euler5] seja o menor número divisível pelos números de 1 a 20 e calcular seu valor.

Solução:

```
euler5 : num;
euler5 <= menorDivisivel (1, 20);
```

O cálculo é:

```
>: euler5;
>> 232792560 : num
>:
```

3.6 - Número de passos para resolver o problema da torre de Hanoi

(**Exercício 3.6.1**) Três hastes de platina são encontradas em um templo hindu. Em uma delas (primeira haste), há 64 anéis de ouro de raios diferentes, colocados do mais alto ao mais baixo. O trabalho dos monges do templo é passar todos os anéis da primeira para a terceira haste, usando a segunda como haste auxiliar, com as seguintes condições:

- Apenas um anel pode ser movido a cada etapa.
- Não poderá haver um anel de diâmetro maior sobre um de diâmetro menor.

Reza a lenda que quando todos os anéis estiverem na terceira haste, será o fim do mundo.

Para 64 anéis é retornado o valor 18.446.744.073.709.551.615.

Definir a função *numPassosHanoi* tal que [numPassosHanoi(n);] seja o número de passos necessários para movimentar "n" anéis. Por exemplo,

```
numPassosHanoi(2); ..................    3
numPassosHanoi(7); .................. 127
numPassosHanoi(64); .................    1.84467440737096e+019
```

Solução: Sejam "A", "B" e "C" as três hastes. A estratégia recursiva é a seguinte:

- Caso base (n = 1): o disco é movido de "A" para "C".
- Caso indutivo (n = m + 1): move-se "m" discos de "A" para "C"; move-se o disco de "A" para "B" e move-se "m" discos de "C" para "B".

Por tanto,

```
numPassosHanoi : num -> num;
numPassosHanoi 1       <= 1;
numPassosHanoi (n + 1) <= 1 + 2 * numPassosHanoi n;
```

3.7 - Conjunção de uma lista

(**Exercício 3.7.1**) Definir por recursão a função *and'* tal que [and'(xs);] verifiquie se todos os elementos de "xs" são verdadeiros. Por exemplo,

```
and'([1+2<4,2::[3] = [2,3]]); .......  true
and'([1+2<3,2::[3] = [2,3]]); .......  false
```

Solução:

```
and' : list truval -> truval;
and' []        <= true;
and' (b :: bs) <= b and' bs;
```

3.8 - Pertence a uma lista

(**Exercício 3.8.1**) Definir por recursão a função *elem'* tal que [elem'(x,xs);] verifique se “x” pertence a lista “xs”. Por exemplo,

```
elem'(3, [2,3,5]); ..................  true
elem'(4, [2,3,5]); ..................  false
elem''(3, [2,3,5]); .................  true
elem''(4, [2,3,5]); .................  false
```

Solução 1:

```
elem' : num # list num -> truval;
elem' (x, [])      <= false;
elem' (x, y :: ys) <= if x = y then true else elem' (x, ys);
```

Solução 2:

```
elem'' : num # list num -> truval;
elem'' (x, [])      <= false;
elem'' (x, y :: ys) <= x = y or elem'' (x, ys);
```

3.9 - Último elemento de uma lista

(**Exercício 3.9.1**) Definir por recursão a função *last'* tal que [last'(xs);] seja o último elemento de “xs”. Por exemplo,

```
last'([2,3,5]); .....................  5
```

Solução:

```
last' : list num -> num;
last' ([x])     <= x;
last' (_ :: xs) <= last' xs;
```

3.10 - Concatenação de uma lista

(**Exercício 3.10.1**) Definir por recursão a função *concat'* tal que [concat'([xss]);] seja uma lista obtida a partir da concatenação das listas de “xss”. Por exemplo,

```
concat'([1..3,5..7,8..10]); .........  [1,2,3,5,6,7,8,9,10]
concat'([[1,2],[3,4,5],[6]]); .......  [1,2,3,4,5,6]
```

Solução:

```
concat' : list (list num) -> list num;
concat' []          <= [];
concat' (xs :: xss) <= xs <> concat' xss;
```

3.11 - Seleção de um elemento

(**Exercício 3.11.1**) Definir por recursão a função *seleciona* tal que [seleciona(xs,n);] seja o enésimo elemento de “xs”. Por exemplo,

```
seleciona([2,3,5,7], 2); ............  5
```

Solução:

```
seleciona : list num # num -> num;
seleciona (x :: _, 0)  <= x;
seleciona (_ :: xs, n) <= seleciona (xs, n - 1);
```

3.12 - Seleção dos primeiros elementos

(**Exercício 3.12.1**) Definir por recursão a função *take'* tal que [take'(n,xs);] seja a lista dos primeiros “n” elementos de “xs”. Por exemplo,

```
take'(3, 4..12); ....................  [4,5,6]
take'(2, [9,7,5,3]); ................  [9,7]
```

Solução:

```
take' : num # list num -> list num;
take' (0, _)       <= [];
take' (_, [])      <= [];
take' (n, x :: xs) <= x :: take' (n - 1, xs);
```

3.13 - Intercalação (comparação) de média aritmética

(**Exercício 3.13.1**) Definir por recursão a função *refinada* tal que [refinada(xs);] seja a lista da intercalação entre cada dois elementos consecutivos de “xs” e a média aritmética entre esses valores, posicionada entre os valores. Por exemplo,

```
refinada([2,7,1,8]); ................  [2,4.5,7,4,1,4.5,8]
refinada([2]); ......................  [2]
refinada([]); .......................  nil (para [])
```

Solução:

```
refinada : list num -> list num;
refinada xs             <= xs;
refinada (x :: y :: zs) <= x :: (x + y) / 2 :: refinada (y :: zs);
```

3.14 - Classificação de listas com mesclagem de elementos

1 - Junção de elementos entre duas listas
2 - Separação de elementos de uma lista em duas listas
3 - Classificação por mesclagem de listas
4 - Validação de lista classificada
5 - Classificação por mesclagem de permutação
6 - Determinação de permutações

(**Exercício 3.14.1**) Definir por recursão a função *mescla* tal que [mescla(xs,ys);] seja a junção dos elementos das listas classificadas “xs” e “ys”. Por exemplo,

```
mescla([2,5,6],[1,3,4]); ............  [1,2,3,4,5,6]
mescla([2,6,5],[3,1,4]); ............  [2,3,1,4,6,5]
```

Solução:

```
mescla : list num # list num -> list num;
mescla ([], ys)          <= ys;
mescla (xs, [])          <= xs;
mescla (x :: xs, y :: ys) <= if x =< y
                             then x :: mescla (xs, y :: ys)
                             else y :: mescla (x :: xs, ys);
```

(**Exercício 3.14.2**) Definir por recursão a função *metade* tal que [metade(xs);] seja formada por um par de listas contendo a metade dos elementos de “xs” divididos de modo que o comprimento entre as listas seja de no máximo um elemento. Por exemplo,

```
metade([2,3,5,7,9]); ................  ([2,3],[5,7,9])
metade([2,3,5,6,7,9]); ..............  ([2,3,5],[6,7,9])
```

Solução:

```
metade : list num -> list num # list num;
metade xs <= splitAt (length xs div 2, xs);
```

(**Exercício 3.14.3**) Definir por recursão a função *classifMescla* tal que [classifMescla(xs);] seja a lista “xs” classificada a partir da junção de seus elementos (se a lista estiver vazia retorna-se o valor vazio; se a lista possuir um só elemento retornará este elemento, mas se a lista tiver elementos distintos estes deverão ser separados em duas listas para que em seguida essas listas sejam combinadas de modo que resultem uma única lista com seus elementos classificados de forma crescente). Por exemplo,

```
classifMescla([5,2,3,1,7,2,5]); .....  [1,2,2,3,5,5,7]
classifMescla([5,6,7,3,4,2,1]); .....  [1,2,3,4,5,6,7]
```

Solução:

```
classifMescla : list num -> list num;
classifMescla []    <= [];
classifMescla ([x]) <= [x];
classifMescla xs    <= mescla (classifMescla ys, classifMescla zs)
                       where (ys, zs) == metade xs;
```

(**Exercício 3.14.4**) Definir por recursão a função *classificada* tal que [classificada(xs);] verifique se “xs” é uma lista classificada. Por exemplo,

```
classificada([2,3,5]); ..............  true
classificada([2,5,3]); ..............  false
```

Solução:

```
classificada : list num -> truval;
classificada []            <= true;
classificada ([_])         <= true;
classificada (x :: y :: xs) <= x =< y and classificada (y :: xs);
```

(**Exercício 3.14.5**) Definir por recursão a função *apaga* tal que [apaga(x,xs);] seja a lista resultante após a remoção (apagamento) da primeira ocorrência de “x” na lista “xs”. Por exemplo,

```
apaga(1, [1,2,1]); ..................  [2,1]
apaga(3, [1,2,1]); ..................  [1,2,1]
```

Solução:

```
apaga : num # list num -> list num;
apaga (x, [])      <= [];
apaga (x, y :: ys) <= if x = y then ys else y :: apaga (x, ys);
```

(**Exercício 3.14.6**) Definir por recursão a função *ehPermut* tal que [ehPermut(xs,ys);] verifique se “xs” é permutação de “ys”. Por exemplo,

```
ehPermut([1,2,1],[2,1,1]); ..........  true
ehPermut([1,2,1],[1,2,2]); ..........  false
```

Solução:

```
ehPermut : list num # list num -> truval;
ehPermut ([], [])      <= true;
ehPermut ([], y :: ys) <= false;
ehPermut (x :: xs, ys) <= elem (x, ys) and ehPermut (xs, apaga (x, ys));
```

3.15 - Lista com elementos replicados

(**Exercício 3.15.1**) Definir por recursão a função *replicar* tal que [replicar(n,x);] seja a lista formada por “n” cópias do elemento “x”. Por exemplo,

```
replicar(3,true);  ..................  [true,true,true]
replicar(4,5);  .....................  [5,5,5,5]
replicar(0,”abelha”);  ..............  nil (para [])
```

Solução:

```
replicar : num # alpha -> list alpha;
replicar (0, _) <= [];
replicar (n, x) <= x :: replicar (n - 1, x);
```

3.16 - Lista dos multiplicadores de um número

(**Exercício 3.16.1**) Definir por recursão a função *multiplicadores* tal que [multiplicadores([x],n);] seja a lista formada pelos multiplicadores da lista indicada em “[x]” a partir do número “n” estabelecido. Por exemplo,

```
multiplicadores([1,2,3,4],2); .......  [1,2]
multiplicadores([1,2,3,4],3); .......  [1,3]
multiplicadores([1,2,3,4],4); .......  [1,2,4]
```

Solução:

```
multiplicadores : list num # num -> list num;
multiplicadores ([], n)      <= [];
multiplicadores (x :: xs, n) <= if n mod x = 0
                                then x :: multiplicadores (xs, n)
                                else multiplicadores (xs, n);
```

3.17 - Listagem dos dígitos de um número

(**Exercício 3.17.1**) Definir por recursão a função *digitos* tal que [digitos(n);] gere uma lista formada pelos dígitos que compões o número "n". Por exemplo,

```
digitos(320274); ....................  [3,2,0,2,7,4]
digitos(123456); ....................  [1,2,3,4,5,6]
```

Solução:

```
digitos' : num -> list num;
digitos' n <= if n < 10
              then [n]
              else n mod 10 :: digitos' (n div 10);

digitos : num -> list num;
digitos n <= reverse (digitos' n);
```

3.18 - Soma dos dígitos de um número

(**Exercício 3.18.1**) Definir por recursão a função *somaDigitos* tal que [somaDigitos(n);] seja a soma dos dígitos que forma o número "n". Por exemplo,

```
somaDigitos(3); .....................   3
somaDigitos(2454); ..................  15
somaDigitos(20045); .................  11
```

Solução:

```
somaDigitos : num -> num;
somaDigitos n <= if n < 10
                 then n
                 else n mod 10 + somaDigitos (n div 10);
```

3.19 - Indicar se o dígito informado faz parte do número

(**Exercício 3.19.1**) Definir a função *temDigito* tal que [*temDigito*(n,x);] verifique se o dígito "n" faz parte do número "x". Por exemplo,

```
temDigito(4, 1041); .................  true
temDigito(3, 1041); .................  false
```

Solução:

```
temDigito : num # num -> truval;
temDigito (n, x) <= elem (n, digitos x);
```

3.20 - Quantidade de dígitos de um número

(**Exercício 3.20.1**) Definir a função *numeroDeDigitos* tal que [*numeroDeDigitos*(x);] mostre a quantidade de dígitos que forma o número "x". Por exemplo,

```
numeroDeDigitos(34047); .............  5
numeroDeDigitos'(34047); ............  5
```

Solução 1:

```
numeroDeDigitos : num -> num;
numeroDeDigitos x <= length (digitos x);
```

Solução 2:

```
numeroDeDigitos' : num -> num;
numeroDeDigitos' x <= length (num2str x);
```

3.21 - Número correspondente a partir dos dígitos de uma lista

(**Exercício 3.21.1**) Definir por recursão a função *listaNumero* tal que [*ListaNumero*([x]);] seja o número formado a partir dos elementos da lista "x". Por exemplo,

```
listaNumero([5]); ...................     5
listaNumero([1,3,4,7]); ............. 1347
listaNumero([0,0,1]); ...............    1
listaNumero([1,0,0]); ...............  100
```

Solução:

```
listaNumero' : list num -> num;
listaNumero' ([x]) <= x;
listaNumero' (x :: xs) <= x + 10 * listaNumero' xs;

listaNumero : list num -> num;
listaNumero xs <= listaNumero' (reverse xs);
```

3.22 - Primeiro dígito de um número

(**Exercício 3.22.1**) Definir por recursão a função *primeiroDigito* tal que [*primeiroDigito*(n);] retorne o primeiro dígito do número "n". Por exemplo,

```
primeiroDigito(425); ................ 4
```

Solução:

```
primeiroDigito : num -> num;
primeiroDigito n <= if n < 10 then n else primeiroDigito (n div 10);
```

3.23 - Dígitos invertidos de um número

(**Exercício 3.23.1**) Definir a função *inversoNum* tal que [*inversoNum*(n);] retorne o número "n" informado de forma invertida. Por exemplo,

```
inversoNum(42578); .................. 87524
inversoNum(203); ....................   302
```

Solução:

```
inversoNum : num -> num;
inversoNum n <= listaNumero (reverse (digitos n));
```

3.24 - Verificar se um número é capicúa

(**Exercício 3.24.1**) Definir a função *capicua* tal que [capicua(n);] verifique se o número "n" possui os mesmo ditos da direita para a esquerda e da esquerda para a direita tendendo ao centro. Por exemplo,

```
capicua(1234); ......................  false
capicua(1221); ......................  true
capicua(4); .........................  true
capicua(12321); .....................  true
```

Solução:

```
capicua : num -> truval;
capicua n <= n = inversoNum n;
```

3.25 - Produto dos dígitos de um número

(**Exercício 3.25.1**) Definir a função *produto* tal que [produto(n);] retorne o resultado da multiplicação dos dígitos que formam o número "n". Por exemplo,

```
produto(327); .......................  42
produto(123); .......................   6
```

Solução:

```
produto : num -> num;
produto n <= product (digitos n);
```

3.26 - Primitivo de um número

(**Exercício 3.26.1**) O primitivo de um número é obtido a partir da multiplicação dos dígitos que compõe esse número até que reste um só dígito, o qual é chamado de primitivo do número inicial. Considerando o número 327 decompondo-o na multiplicação de 3 x 2 x 7 obtém-se 42, que por sua vez sendo decomposto na multiplicação de 4 x 2 tem-se ao final o valor 8. Deste modo, 8 é primitivo de 327. Assim sendo, definida por recursão a função a *primitivo* tal que [primitivo(n);] apresente o primitivo do número "n". Por exemplo,

```
primitivo(327); .....................  8
primitivo(123); .....................  6
```

Solução:

```
primitivo : num -> num;
primitivo n <= if n < 10 then n else primitivo (produto n);
```

3.27 - Números com média igual aos seus dígitos

(**Exercício 3.27.1**) Dois números são equivalentes se a média de seus dígitos forem iguais. Por exemplo, 3205 e 41 são equivalentes já que:

$$\frac{3+2+0+5}{4} = \frac{4+1}{2}$$

Definir a função *equivalentes* tal que [equivalentes(x,y);] verifique se os números "x" e "y" são equivalentes. Por exemplo,

```
equivalentes(3205,41); ..............  true
equivalentes(3205,25); ..............  false
```

Solução:

```
equivalentes : num # num -> truval;
equivalentes (x, y) <= media (digitos x) = media (digitos y);
```

3.28 - Intervalo numérico entre valores de uma lista

(**Exercício 3.28.1**) Definir a função *entre* tal que [entre(m,n);] seja uma lista de números entre "m" e "n". Por exemplo,

```
entre(2,5); .........................  [2,3,4,5]
entre(1,6); .........................  [1,2,3,4,5,6]
```

Solução:

```
entre : num # num -> list num;
entre (m, n) <= if m > n then [] else m :: entre (m + 1, n);
```

3.29 - Limite numérico dentro de certa faixa

(**Exercício 3.29.1**) Definir por recursão a função *naFaixa* tal que [naFaixa(a,b,[xs]);] seja uma lista dos elementos de "xs" maiores ou iguais a "a" e menores ou iguais a "b". Por exemplo,

```
naFaixa(5,10,1..15); ................  [5,6,7,8,9,10]
naFaixa(10,5,1..15); ................  []
naFaixa(5,5,1..15); .................  [5]
```

Solução:

```
naFaixa : num # num # list num -> list num;
naFaixa (a, b, [])      <= [];
naFaixa (a, b, x :: xs) <= if a =< x and x =< b
                           then x :: naFaixa (a, b, xs)
                           else naFaixa (a, b, xs);
```

3.30 - Substituição de impar pelo próximo par

(**Exercício 3.30.1**) Definir por recursão a função *substitImpar* tal que [substitImpar([xs]);] seja uma lista em que cada número impar seja substituído pelo próxima número par. Por exemplo,

```
substitImpar([2,5,7,4]); ............  [2,6,8,4]
```

Solução:

```
substitImpar : list num -> list num;
substitImpar []        <= [];
substitImpar (x :: xs) <= if odd x
                          then (x + 1) :: substitImpar xs
                          else x :: substitImpar xs;
```

3.31 - Expansão da fatoração de um número

(**Exercício 3.31.1**) Definir por recursão a função *expansao* tal que [expansao([(a,b)]);] seja a expansão da fatoração de uma lista de tuplas “(a,b)”. Por exemplo,

```
expansao([(2,2),(3,1),(5,1)]); ......  60
```

Solução:

```
expansao : list (num # num) -> num;
expansao []           <= 1;
expansao ((x, y) :: zs) <= pow (x, y) * expansao zs;
```

3.32 - Soma dos dígitos de uma cadeia

(**Exercício 3.32.1**) Definir por recursão a função *somaDigitosCadeia* tal que [somaDigitosCadeia([texto]);] seja a soma dos dígitos existentes na cadeia “texto”. Usar para a operação as funções Nota: *isDigit* y *digitToInt*. Por exemplo,

```
somaDigitosCadeia("12345"); .........  15
somaDigitosCadeia("SE 2431 X"); .....  15
```

Solução:

```
somaDigitosCadeia : list char -> num;
somaDigitosCadeia []        <= 0;
somaDigitosCadeia (x :: xs) <= if isDigit x
                               then digitToInt x + somaDigitosCadeia xs
                               else somaDigitosCadeia xs;
```

3.33 - Capitalização de uma cadeia

(**Exercício 3.33.1**) Definir a função *maiuscInicial* tal que [maiuscInicial([texto]);] seja a apresentação do primeiro caractere da cadeia “texto” em maiúsculo e os demais caracteres da cadeia em minúsculo. Por exemplo,

```
maiuscInicial("aUgusto"); ...........  "Augusto"
maiuscInicial("sEviLLa"); ...........  "Sevilla"
```

Existem muitas formas de resolver esta proposta de operação. Foi optado uma maneira simples a partir da definição de duas funções especializadas, sendo a primeira destinada a escrever toda a cadeia em letras maiúsculas e a segunda em escrever toda a cadeia em letras minúsculas.

```
tudoMaiusculo : list char -> list char;
tudoMaiusculo []        <= [];
tudoMaiusculo (x :: xs) <= toUpper x :: tudoMaiusculo xs;

tudoMinusculo : list char -> list char;
tudoMinusculo []        <= [];
tudoMinusculo (x :: xs) <= toLower x :: tudoMinusculo xs;
```

A partir das funções *tudoMaiusculo* e *tudoMinusculo* fica fácil escrever uma função que capitalize apenas o primeiro caractere de uma cadeia.

Solução:

```
maiuscInicial : list char -> list char;
maiuscInicial []        <= [];
maiuscInicial (x :: xs) <= toUpper x :: tudoMinusculo xs;
```

3.34 - Número de zeros finais

(**Exercício 3.34.1**) Definir por recursão a função *zeros* tal que [zeros([n]);] seja a quantidade de zeros que aparecem no final do número “n”. Por exemplo,

```
zeros(30500); .......................  2
zeros(30501); .......................  0
```

Solução:

```
zeros : num -> num;
zeros n <= if n mod 10 = 0 then 1 + zeros (n div 10) else 0;
```

3.35 - Aplicação iterada de função

(**Exercício 3.35.1**) Definir por recursão a função *potenciaFunc* tal que [PotenciaFunc(n,f,x);] seja o resultado da aplicação da função “f” por “n” sobre “x”. Por exemplo,

```
potenciaFunc(3,(*10),5); ............  5000
potenciaFunc(4,(+10),5); ............    45
```

Solução:

```
potenciaFunc : num # (alpha -> alpha) # alpha -> alpha;
potenciaFunc (0, _, x) <= x;
potenciaFunc (n, f, x) <= potenciaFunc (n - 1, f, f x);
```

3.36 - Soma dos primeiros números impares

(**Exercício 3.36.1**) Definir por recursão a função *somaImpares* tal que [somaImpares (n);] seja a soma dos primeiro “n” números impares. Por exemplo,

```
somaImpares(5); .....................  35
```

Solução:

```
somaImpares : num -> num;
somaImpares 0 <= 0;
somaImpares n <= 2 * n + 1 + somaImpares (n - 1);
```

3.37 - Um mais a soma das potências de dois

(**Exercício 3.37.1**) Definir por recursão a função *somaPotenciasde2mais1* tal que [somaPotenciasde2mais1(n);] seja igual a $1 + 2^0 + 2^1 + 2^2 + ... 2^n$. Por exemplo,

```
somaPotenciasde2mais1(3); ...........  16
```

Solução:

```
somaPotenciasde2mais1 : num -> num;
somaPotenciasde2mais1 0 <= 2;
somaPotenciasde2mais1 n <= pow (2, n) + somaPotenciasde2mais1 (n - 1);
```

3.38 - Lista de números impares

(**Exercício 3.38.1**) Definir a função *impares* tal que [impares([xs});] verifique se determinada lista "xs" informada para a função é formada por elementos impares. Por exemplo,

```
impares([1,3,5]); ...................  true
impares([1,3,5,6]); .................  false
```

Solução:

```
impares : list num -> truval;
impares n <= odd (listaNumero n);
```

3.39 - Inversão de tuplas

(**Exercício 3.39.1**) Definir a função *inverteTuplas* tal que [inverteTuplas([(a,b)])] apresente uma lista de tuplas com seus elementos invertidos. Dica: usar como apoio as funções *snd* e *fst*. Por exemplo,

```
inverteTuplas([(1,2),(3,4)]); .......  [(2,1),(4,3)]
inverteTuplas'([(1,2),(3,4)]); ......  [(2,1),(4,3)]
```

Solução 1:

```
inverteTuplas : list (num # num) -> list (num # num);
inverteTuplas []       <= [];
inverteTuplas (x :: xs) <= (snd x, fst x) :: inverteTuplas xs;
```

Solução 2:

```
inverteTuplas' : list (num # num) -> list (num # num);
inverteTuplas' []            <= [];
inverteTuplas' ((x, y) :: xs) <= (y, x) :: inverteTuplas' xs;
```

(**Exercício 3.39.2**) Definir a função *tuplasInvertidas* tal que [*tuplasInvertidas*([(a,b)])] apresente uma lista de tuplas com seus elementos invertidos em sentido oposto a ordem original. Por exemplo,

```
tuplasInvertidas([(1,2),(3,4)]); ....  [(4,3),(2,1)]
```

Solução:

```
tuplasInvertidas : list (num # num) -> list (num # num);
tuplasInvertidas [] <= [];
tuplasInvertidas xs <= reverse (inverteTuplas' xs);
```

Capítulo 4 - Definições por compreensão

Este capítulo apresenta exercícios com definições por compreensão. Os exercícios apresentados estão baseados nos capítulos 5 e 6 (ALONSO & MANZANO, 2021), além de alguns acréscimos e subtrações implementadas a partir das situações onde compreensões operam com mais de uma lista.

OBS.:

Para uso de alguns dos eventos deste capítulo em linguagem Hope é necessário ter em mãos funções que realizem a simulação das ações de compreensão, pois a linguagem não possui de forma direta recurso para realizar ações de compreensão. No entanto, nem todos os exercícios propostos na obra original puderam ser portados e neste caso esses exercícios foram deixados no capítulo anterior para serem resolvidos com ações de recursão.

Considere para os exercícios deste capítulo a definição de algumas funções auxiliares para a simulação de ações de compreensão a partir de funções recursivas:

```
compNum : (alpha -> beta) # list alpha -> list beta;
compNum (funcao, []) <= [];
compNum (funcao, x :: xs) <= funcao x :: compNum (funcao, xs);

compTri : (num -> list num) # list num -> list (list num);
compTri (funcao, []) <= [];
compTri (funcao, x :: xs) <= funcao x :: compTri (funcao, xs);
```

4.1 - Soma dos quadrados dos primeiros n números

(**Exercício 4.1.1**) Definir por compreensão a função *somaDosQuadrados* de tal modo que [somaDosQuadrados(n);] seja a soma dos quadrados dos "n" primeiros números, ou seja, 1^2 + 2^2 + ... + n^2 a partir da operação: ∑ [x ^ 2 | x <- [1..n]]. Por exemplo,

```
somaDosQuadrados(3); ................      14
somaDosQuadrados(100); ..............  338350
```

Solução:

```
somaDosQuadrados : num -> num;
somaDosQuadrados n <= sum (compNum (\x => pow (x, 2), 1 .. n));
```

4.2 - Triângulos aritméticos

1 - Somatório dos valores de 1 até "n"
2 - Linha de um triângulo aritmético
3 - Calculo para formação de um triângulo aritmético

(**Exercício 4.2.1**) Definir a função *soma* de modo que [soma(n);] seja a soma dos "n" primeiros números. Por exemplo,

```
soma(3); ............................     6
soma(100); ..........................  5050
```

Solução 1:

```
soma : num -> num;
soma n <= sum (1 .. n);
```

Solução 2:

```
soma' : num -> num;
soma' n <= (1 + n) * n div 2;
```

(**Exercício 4.2.2**) Um triângulo aritmético é formado pela estrutura numérica:

```
 1
 2  3
 4  5  6
 7  8  9 10
11 12 13 14 15
16 17 18 19 20 21
```

Definir a função *linha* de tal modo que [linha(n);] seja a enésima linha de certo triângulo aritmético. Por exemplo,

```
linha(4); ..........................  [7,8,9,10]
linha(5); ..........................  [11,12,13,14,15]
```

Solução:

```
linha : num -> list num;
linha n <= soma (n - 1) + 1 .. soma n;
```

(**Exercício 4.2.3**) Definir a função *triangArit* tal que [triangArit(n);] seja um triângulo aritmético de altura "n" a partir da operação: [linha m | m <- [1..n]]. Por exemplo,

```
triangArit(3); .....................  [[1],[2,3],[4,5,6]]
triangArit(2); .....................  [[1],[2,3]]
```

Solução:

```
triangArit : num -> list (list num);
triangArit n <= compTri (linha, 1 .. n);
```

4.3 - Números perfeitos

(**Exercício 4.3.1**) Um número inteiro positivo é perfeito quando a soma de seus fatores é igual a ele, excluindo-se o próprio número. Definir por compreensão a função *perfeitos* tal que [perfeitos(n);] é a lista de todos os número perfeitos menores que "n". Por exemplo,

```
perfeitos(500); ....................  [6,28,496]
```

Para solucionar esta questão é necessário possuir uma função que calcule os fatores de determinado valor. Observe a função [perfeitos(n);] que apresenta uma lista dos fatores do número indicado. Para ver os fatores de 15, execute "fatores(15);" e será mostrada a lista "[1,3,5,15]" a partir da operação: [x | x <- [1..n], n mod x = 0].

Solução:

```
fatores : num -> list num;
fatores n <= filter (\x => n mod x = 0, 1 .. n);
```

Após a definição da função fatores passa-se a definição da função [perfeitos(n);] a partir da operação: [x | x <- [1..n], sum (init (fatores x)) = x].

Solução:

```
perfeitos : num -> list num;
perfeitos n <= filter (\x => sum (init (fatores x)) = x, 1 .. n);
```

4.4 - Números amigos

(**Exercício 4.4.1**) Dois números são amigos quando o somatório de cada um de seus fatores (divisores), excetuando-se o próprio valor é igual ao outro.

Definir a função *amigo* tal que [amigo(n);] seja a indicação verdadeira se dado dois números estes forem amigos. Por exemplo,

```
amigos(220,284); ....................  true
amigos(221,283); ....................  false
```

Solução:

```
amigos : num # num -> truval;
amigos (x, y) <= sum (fatores x) - x = y and sum (fatores y) - y = x;
```

4.5 - Números abundantes

Um número natural “n” é chamado de abundante se for menor que a soma de seus divisores, excluindo-se o próprio número. Por exemplo, 12 e 30 são abundantes, mas 5 e 28 não são.

1 - Verificar número abundante
2 - Números abundantes menores que certo limite
3 - Números abundantes pares
4 - Primeiro abundante impar

(**Exercício 4.5.1**) Definir a função *numeroAbundante* tal que [numeroAbundante(n);] verifique se dado número “n” é ou não abundante. Por exemplo,

```
numeroAbundante(5); .................  false
numeroAbundante(12); ................  true
numeroAbundante(28); ................  false
numeroAbundante(30); ................  true
```

Para a realização da operação de verificação se dado valor é ou não abundante é necessário considerar a função *divisores* tal que [divisores(n)] apresente lista dos valores divisores de “n”.

Solução:

```
divisores : num -> list num;
divisores n <= filter (\m => n mod m = 0, 1 .. n);
```

Após a definição da função divisores passa-se a definição da função que validará se número é ou não abundante. Observe a função [numeroAbundante (n);].

Solução:

```
numeroAbundante : num -> truval;
numeroAbundante n <= n < sum (divisores n) - n;
```

(**Exercício 4.5.2**) Definir a função *numerosAbundantesMenores* tal que [numerosAbundantes-Menores(n);] seja uma lista de números abundantes menores e iguais a “n”. Por exemplo,

```
numerosAbundantesMenores(50); .......  [12,18,20,24,30,36,40,42,48]
```

Solução:

```
numerosAbundantesMenores : num -> list num;
numerosAbundantesMenores n <= filter (numeroAbundante, 1 .. n);
```

(**Exercício 4.5.3**) Definir a função *todosPares* tal que [todosPares(n);] verifique se os números abundantes menores ou iguais a “n” são pares. Por exemplo,

```
todosPares(10); .....................  true
todosPares(100); ....................  true
todosPares(1000); ...................  false
```

Solução:

```
todosPares : num -> truval;
todosPares n <= and' (compNum (even, numerosAbundantesMenores n));
```

(**Exercício 4.5.4**) Definir a constante *primerAbundanteImpar* que retorne o valor do primeiro número natural abundante ímpar. Determinar o valor do dito número.

Solução:

```
primerAbundanteImpar : num;
primerAbundanteImpar <= head (filter (
                              odd, numerosAbundantesMenores 140000));
```

O valor 140.000 definido para a função *numerosAbundantesMenores* refere-se a um valor de segurança máximo estabelecido para evitar estouro de memória. O cálculo da operação é:

```
primerAbundanteImpar; ...............  945
```

4.6 - Fatores primos

Número primo caracteriza-se por ser um valor natural divisível por 1 e por ele mesmo. É sabido que os números naturais maiores que 1 podem ser decompostos em fatores. Os fatores primos de um número são os números primos que dividem exatamente esse número. Assim sendo, determine os fatores primos de um determinado número inteiro positivo. Os fatores primos do número 112 são 2, 2, 2, 2 e 7. Neste caso a função deve retornar uma lista apenas com os números [2, 7].

(**Exercício 4.6.1**) Definir a função *fatoresPrimos* tal que [fatoresPrimos(n);] seja uma lista de números que são os fatores primos de “n”. Por exemplo,

```
fatoresPrimos(112); .................  [2,7]
fatoresPrimos(220); .................  [2,5,11]
fatoresPrimos(360); .................  [2,3,5]
```

Solução:

```
checaPrimo : num -> truval;
checaPrimo n <= fatores n = [1, n];

fatoresPrimos : num -> list num;
fatoresPrimos n <= filter (checaPrimo, fatores n);
```

4.7 - Aproximação do número "e"

(**Exercício 4.7.1**) Definir a função *aproxE* tal que [aproxE(n);] é a lista cujos elementos são os termos da sucessão *(1 + 1/m) ^ m*, desde 1 até “n”. Por exemplo,

```
aproxE(1); ..........................  [2]
aproxE(2); ..........................  [2,2.25]
aproxE(3); ..........................  [2,2.25,2.37037037037037]
```

Solução:

```
aproxE : num -> list num;
aproxE n <= compNum(\m => pow (1 + 1 / m, m), 1 .. n);
```

(**Exercício 4.7.2**) Qual é o limite da sucessão (1 + 1/m) ^ m?

Solução:

O limite da sucessão é o número e.

(**Exercício 4.7.3**) Definir a função *errorE* tal que [errorE(x);] é o menor número de termos da sucessão *(1 + 1/m) ^ m* necessários para obter seu limite com um erro menor que “x”. Por exemplo,

```
errorE(0.1); ........................    13
errorE(0.01); .......................   135
errorE(0.001); ......................  1359
```

Solução:

```
errorE : num -> num;
errorE' x <= head (filter (\n => abs (aproxE' n - e) < x, from 1));
```

(**Exercício 4.7.4**) O número “e” pode ser definido a partir da soma da série:

$$\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

Definir a função *aproxE'* tal que [aproxE'(n);] seja a aproximação de “e” obtida a partir da soma dos termos da série até *1/n!* . Por exemplo,

```
aproxE'(10); ........................  2.71828180114638
aproxE'(100); .......................  2.71828182845905
```

Solução:

```
fatorial : num -> num;
fatorial n <= product (1 .. n);

aproxE' : num -> num;
aproxE' n <= 1 + sum (compNum (\k => 1 / fatorial k, 1 .. n));
```

(**Exercício 4.7.5**) Definir a constante “e” como 2,71828459:

Solução:

```
e : num;
e <= 2.71828459;
```

(**Exercício 4.7.6**) Definir a função *errorE'* tal que [errorE'(x);] seja o menor número de termos da série anterior necessária para obter o número “e” com erro menor que “x”. Por exemplo,

```
errorE'(0.1); .......................  3
errorE'(0.01); ......................  4
errorE'(0.001); .....................  6
errorE'(0.0001); ....................  7
```

Solução:

```
errorE' : num -> num;
errorE' x <= head (filter (\n => abs (aproxE' n - e) < x, from 1));
```

4.8 - Aproximação de limite

(**Exercício 4.8.1**) Definir a função *aproxLimSeno* tal que [aproxLimSeno(n);] seja a lista cujos elementos são os termos da sucessão *(sen(1/m))/(1/m)* de 1 até “n”. Por exemplo,

```
aproxLimSeno(1); ....................  [0.841470984807897]
aproxLimSeno(2); ....................  [0.84147098...,0.95885107...]
```

Solução:

```
aproxLimSeno : num -> list num;
aproxLimSeno n <= compNum (\m => sin (1 / m) / (1 / m), 1 .. n);
```

(**Exercício 4.8.2**) Qual é o limite da sucessão *(sen(1/m))/(1/m)*?

Solução:

O limite da sucessão é 1.

(**Exercício 4.8.3**) Definir a função *errorLimSeno* tal que [errorLimSeno(x);] seja o menor número de termos da sucessão *(sen(1/m))/(1/m)* necessários para obter seu limite com um erro menor que "x". Por exemplo,

```
errorLimSeno(0.1); ..................   2
errorLimSeno(0.01); .................   5
errorLimSeno(0.001); ................  13
errorLimSeno(0.0001); ...............  41
```

Solução:

```
errorLimSeno : num -> num;
errorLimSeno x <= head (filter (\m => abs (1 - sin (1 / m) / (1 / m)) < x,
                        from 1));
```

4.9 - Cálculo do número π

(**Exercício 4.9.1**) Definir a função *calculaPi* tal que [calculaPi(n);] seja a aproximação do número π calculado mediante a expressão:

$$4 * (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{(-1)^n}{2n+1})$$

Por exemplo,

```
calculaPi(3); .......................  2.8952380952381
calculaPi(300); .....................  3.14491490355885
```

Solução:

```
calculaPi : num -> num;
calculaPi n <= 4 * sum (compNum (\x => pow (0 - 1, x) / (2 * x + 1),
                        0 .. n));
```

(**Exercício 4.9.2**) Definir a função *errorPi* tal que [errorPi(x);] seja o menor número de termos da série:

$$4 * (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{(-1)^n}{2n+1})$$

necessários para obter π com erro menor que "x". Por exemplo,

```
errorPi(0.1); ......................   9
errorPi(0.01); .....................  99
errorPi(0.001); .................... 999
```

Solução:

```
errorPi : num -> num;
errorPi x <= head (filter (\n => abs (pi - calculaPi n) < x, from 1));
```

4.10 - Número de fatores de uma fatoração

Fatoração é a decomposição de um número em fatores primos, ou seja, escrever um número por meio da multiplicação de números primos. Na fatoração usa-se os números primos de modo crescente de acordo com as regras de divisibilidade em razão do termo a ser fatorado. A fatoração do número 60 resulta na obtenção dos valores 2, 2, 3 e 5.

Para gerar o número de fatores de uma fatoração é importante ter em mãos algumas funções que auxiliem a operação.

```
menorFator : num -> num;
menorFator n <= head (filter (\x => multiplo (n, x), from 2));

fatoracao : num -> list num;
fatoracao n <= if n = 1
               then []
               else x :: fatoracao (n div x)
               where x == menorFator n;
```

A função *menorFator* retorna o valor do menor fator primo de um número "n" e a função *fatoracao* apresenta a lista de todos os fatores primos de "n".

(**Exercício 4.10.1**) Definir a função *numerosDeFatoracao* de modo que [numerosDeFatoracao(n);] seja um conjunto formado pelos valores primos dos números da fatoração de "n" iniciados em 1 sem a ocorrência de repetições de valores na lista, Por exemplo,

```
menorFator(60); .................... 2
fatoracao(60); ..................... [2,2,3,5]
numerosDeFatoracao(60); ............ [1,2,3,5]
```

Solução:

```
numerosDeFatoracao : num -> list num;
numerosDeFatoracao n <= 1 :: unique (fatoracao n);
```

Capítulo 5 – Definições por recursão e compreensão

Este capítulo apresenta exercícios com duas definições (uma por recursão e outra por compreensão), se assim for possível de serem implementados em Hope: nem toda a solução recursiva é possível de ser obtida com solução por compreensão. Os exercícios correspondem aos capítulos 5, 6 e 7 (ALONSO & MANZANO, 2021). Algumas subtrações foram definidas a partir das situações onde compreensões operam com mais de uma lista E Alguns exercícios indicados no texto original foram remanejados para os capítulos anteriores (ALONSO, 2019).

5.1 - Cálculo do número π

(**Exercício 5.1.1**) Definir por recursão a função *somaQuadradosR* tal que [somaQuadradoR(n);] seja a soma dos quadrados do números de 1 até “n”. Por exemplo,

```
somaQuadradosR(4); ..................  30
```

Solução:

```
somaQuadradosR : num -> num;
somaQuadradosR 0 <= 0;
somaQuadradosR n <= pow (n, 2) + somaQuadradosR (n - 1);
```

(**Exercício 5.1.2**) Definir por compreensão a função *somaQuadradosC* tal que [somaQuadradoC(n);] seja a soma dos quadrados do números de 1 até “n”. Por exemplo,

```
somaQuadradosC(4); ..................  30
```

Solução:

```
somaQuadradosC : num -> num;
somaQuadradosC n <= sum (compNum (\x => pow (x, 2), 1 .. n));
```

5.2 - Número de blocos em escadas triangulares

(**Exercício 5.2.1**) Você quer formar uma escada com blocos quadrados, de modo que se tenha um número determinado de camadas. Por exemplo, uma escada com três camadas terá a seguinte forma:

```
    XX
  XXXX
XXXXXX
```

Definir por recursão a função *numeroCamadasR* tal que [numeroCamadasR(n);] apresente o número de camadas necessárias para construir a escada triangular com “n” camadas. Por exemplo,

```
numeroCamadasR(1); ..................    2
numeroCamadasR(3); ..................   12
numeroCamadasR(10); .................  110
```

Solução:

```
numeroCamadasR : num -> num;
numeroCamadasR 0 <= 0;
numeroCamadasR n <= 2 * n + numeroCamadasR (n - 1);
```

(**Exercício 5.2.2**) Definir por compreensão a função *numeroCamadasC* tal que [numeroCamadasC(n);] apresente o número de camadas necessárias para construir a escada triangular com "n" camadas. Por exemplo,

```
numeroCamadasC(1); ..................    2
numeroCamadasC(3); ..................   12
numeroCamadasC(10); .................  110
```

Solução:

```
numeroCamadasC : num -> num;
numeroCamadasC n <= sum (compNum ((2 *), 1 .. n)));
```

5.3 - Soma dos quadrados dos ímpares entre os primeiros números

(**Exercício 5.3.1**) Definir por recursão a função *somaQuadImparesR* tal que [somaQuadImparesR(n);] seja a soma dos quadrados do números impares de 1 até "n". Por exemplo,

```
somaQuadImparesR(1); ................   1
somaQuadImparesR(7); ................  84
somaQuadImparesR(4); ................  10
```

Solução:

```
somaQuadImparesR : num -> num;
somaQuadImparesR 1 <= 1;
somaQuadImparesR n <= if odd n
                      then pow (n, 2) + somaQuadImparesR (n - 1)
                      else somaQuadImparesR (n - 1);
```

(**Exercício 5.3.2**) Definir por compreensão a função *somaQuadImparesC* tal que [somaQuadImparesC(n);] seja a soma dos quadrados do números impares de 1 até "n". Por exemplo,

```
somaQuadImparesC(1); ................   1
somaQuadImparesC(7); ................  84
somaQuadImparesC(4); ................  10
```

Solução:

```
somaQuadImparesC : num -> num;
somaQuadImparesC n <= sum (listPow (2,
                           filter (\x => odd (pow (x, 2)), 1 .. n)));
```

5.4 - Quadrados dos elementos de uma lista

(**Exercício 5.4.1**) Definir por recursão a função *quadradosR* tal que [quadradosR([xs]);] seja a lista dos elementos de "xs" ao quadrado. Por exemplo,

```
quadradosR([1,2,3]); ................  [1,4,9]
```

Solução:

```
quadradosR : list num -> list num;
quadradosR []        <= [];
quadradosR (x :: xs) <= pow (x, 2) :: quadradosR xs;
```

(**Exercício 5.4.2**) Definir por compreensão a função *quadradosC* de modo que [quadradosC([xs]);] seja a lista dos elementos de "xs" ao quadrado. Por exemplo,

```
quadradosC([1,2,3]); ................  [1,4,9]
```

Solução:

```
quadradosC : list num -> list num;
quadradosC xs <= compNum (\x => pow (x, 2), xs);
```

5.5 - Números impares de una lista

(**Exercício 5.5.1**) Definir por recursão a função *imparesR* tal que [imparesR([xs]);] seja a lista dos números impares de "xs". Por exemplo,

```
imparesR([1,2,4,3,6]); ..............  [1,3]
imparesR(1..6); .....................  [1,3,5]
```

Solução:

```
imparesR : list num -> list num;
imparesR []        <= [];
imparesR (x :: xs) <= if odd x then x :: imparesR xs else imparesR xs;
```

(**Exercício 5.5.2**) Definir por compreensão a função *imparesC* tal que [imparesC([xs]);] seja a lista dos números impares de "xs". Por exemplo,

```
imparesC([1,2,4,3,6]); ..............  [1,3]
imparesC(1..6); .....................  [1,3,5]
```

Solução:

```
imparesC : list num -> list num;
imparesC xs <= filter (odd, xs);
```

5.6 - Quadrado dos elementos impares

(**Exercício 5.6.1**) Definir por recursão a função *imparesQuadR* de modo que [imparesQuadR([xs]);] seja a lista dos quadrados impares de "xs". Por exemplo,

```
imparesQuadR([1,2,4,3,6]); ..........  [1,9]
```

Solução:

```
imparesQuadR : list num -> list num;
imparesQuadR []        <= [];
imparesQuadR (x :: xs) <= if odd x
                          then pow (x, 2) :: imparesQuadR xs
                          else imparesQuadR xs;
```

(**Exercício 5.6.2**) Definir por compreensão a função *imparesQuadC* de modo que [impares-QuadC([xs]);] seja a lista dos quadrados impares de “xs”. Por exemplo,

```
imparesQuadC([1,2,4,3,6]); ..........  [1,9]
```

Solução:

```
imparesQuadC : list num -> list num;
imparesQuadC xs <= listPow (2, filter (odd, xs));
```

5.7 - Soma dos quadrados dos elementos impares

(**Exercício 5.7.1**) Definir por recursão a função *somaQuadImpR* de modo que [somaQua-dImpR([xs]);] seja a soma da lista dos quadrados impares de “xs”. Por exemplo,

```
somaQuadImpR([1,2,4,3,6]); ..........  10
```

Solução:

```
somaQuadImpR : list num -> num;
somaQuadImpR []        <= 0;
somaQuadImpR (x :: xs) <= if odd x
                          then pow (x, 2) + somaQuadImpR xs
                          else somaQuadImpR xs;
```

(**Exercício 5.7.2**) Definir por compreensão a função *somaQuadImpC* de modo que [somaQua-dImpC([xs]);] seja a soma da lista dos quadrados impares de “xs”. Por exemplo,

```
somaQuadImpC([1,2,4,3,6]); ..........  10
```

Solução:

```
somaQuadImpC : list num -> num;
somaQuadImpC xs <= sum (listPow (2, filter (odd, xs)));
```

5.8 - Metade dos elementos pares

Para auxiliar parte desta operação considere a função *listdiv* tal que [listdiv([xs,n]);] seja uma lista dos números de “xs” divididos por “n”.

```
listdiv : list num # num -> list num;
listdiv ([], _)      <= [];
listdiv (x :: xs, n) <= x / n :: listdiv (xs, n);
```

(**Exercício 5.8.1**) Definir por recursão a função *metadeParesR* de modo que [metadePa-resR([xs]);] seja uma lista resultante dos elementos pares de “xs”. Por exemplo,

```
metadeParesR([0,2,1,8,5,6,9]); ......  [0,1,4,3]
```

Solução:

```
metadeParesR : list num -> list num;
metadeParesR []       <= [];
metadeParesR (x :: xs) <= if even x
                          then x / 2 :: metadeParesR xs
                          else metadeParesR xs;
```

(**Exercício 5.8.2**) Definir por compreensão a função *metadeParesC* de modo que [metadeParesC([xs]);] seja uma lista resultante dos elementos pares de "xs". Por exemplo,

```
metadeParesC([0,2,1,8,5,6,9]); ......  [0,1,4,3]
```

Solução:

```
metadeParesC : list num -> list num;
metadeParesC xs <= listdiv (filter (\x => even x, xs), 2);
```

5.9 - Aproximação do número π

A soma da série

$$\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \ldots$$

é igual a (π ^ 2 / 6). Por tanto, π pode ser aproximado mediante a raiz quadrada de 6 pela soma da série. Por exemplo,

(**Exercício 5.9.1**) Definir por recursão a função *aproximaPiR* tal que [aproximaPi(n);] seja a aproximação de "π" obtido a partir da dos "n" termos da série. Por exemplo,

```
aproximaPiR(4); .....................  2.92261298612503
aproximaPiR(1000); ..................  3.14063805620599
```

Solução:

```
aproximaPiR' : num -> num;
aproximaPiR' 1 <= 1;
aproximaPiR' n <= 1 / pow (n, 2) + aproximaPiR' (n - 1);

aproximaPiR : num -> num;
aproximaPiR n <= sqrt (6 * aproximaPiR' n);
```

(**Exercício 5.9.2**) Definir por compreensão a função *aproximaPiC* tal que [aproximaPiC(n);] seja uma lista resultantes dos elementos pares de "xs". Por exemplo,

```
aproximaPiC(4); .....................  2.92261298612503
aproximaPiC(1000); ..................  3.14063805620599
```

Solução:

```
aproximaPiC : num -> num;
aproximaPiC n <= sqrt (6 * sum (compNum (\x => 1 / pow (x, 2), 1 .. n)));
```

5.10 - Compra com desconto

Certa pessoa somente compra algo quando consegue pechinchar um desconto de 10% e o preço de aquisição (com o desconto) seja menor ou igual a 199.

(**Exercício 5.10.1**) Definir por recursão a função *sovinaR* tal que [sovinaR([ps]);] seja o preço máximo a ser pago por uma compra cuja lista de preços seja “ps”. Por exemplo,

```
sovinaR([45,199,220,399]); ..........  417.6
```

Solução:

```
sovinaR : list num -> num;
sovinaR []        <= 0;
sovinaR (p :: ps) <= if precoComDesconto =< 199
                     then precoComDesconto + sovinaR ps
                     else sovinaR ps
                     where precoComDesconto == p * 0.9;
```

(**Exercício 5.10.2**) Definir por compreensão a função *sovinaR* tal que [sovinaR([ps]);] seja o preço máximo a ser pago por uma compra cuja lista de preços seja “ps”. Por exemplo,

```
sovinaC([45,199,220,399]); ..........  417.6
```

Solução:

```
sovinaC : list num -> num;
sovinaC ps <= sum (filter (\p => p * 0.9 =< 199,
                   compNum (\p => p * 0.9, ps)));
```

5.11 - Expoente da maior potência de um número que divide outro número

(**Exercício 5.11.1**) Definir por recursão a função *maiorExpoenteR* tal que [maiorExpoenteR(a,b);] seja o expoente de maior potência de “a” que divide a “b”. Por exemplo,

```
maiorExpoenteR(2,8); ................  3
maiorExpoenteR(2,60); ...............  2
```

Solução:

```
maiorExpoenteR : num # num -> num;
maiorExpoenteR (a, b) <= if b mod a /= 0
                         then 0
                         else 1 + maiorExpoenteR (a, b div a);
```

(**Exercício 5.11.2**) Definir por recursão a função *maiorExpoenteC* tal que [maiorExpoenteC(a,b);] seja o expoente de maior potência de “a” que divide a “b”. Por exemplo,

```
maiorExpoenteC(2,8); ................  3
maiorExpoenteC(2,60); ...............  2
```

Solução:

```
maiorExpoenteC : num # num -> num;
maiorExpoenteC (a, b) <= head (filter (\x => b mod pow (a, x) /= 0,
                               compNum (\x => x - 1, from 1))) - 1;
```

5.12 - Soma de elementos apenas positivos

(**Exercício 5.12.1**) Definir por recursão a função *somaPositivosR* tal que [somaPositivosR([xs];] seja a soma apenas dos elementos positivos de “xs”. Por exemplo,

```
somaPositivosR([1,0-3,3,6]); ........  10
```

Solução:

```
somaPositivosR : list num -> num;
somaPositivosR []        <= 0;
somaPositivosR (x :: xs) <= if x > 0
                            then x + somaPositivosR xs
                            else somaPositivosR xs;
```

(**Exercício 5.12.2**) Definir por compreensão a função *somaPositivosC* tal que [somaPositivosC([xs];] seja a soma apenas dos elementos positivos de “xs”. Por exemplo,

```
somaPositivosC([1,0-3,3,6]); ........  10
```

Solução:

```
somaPositivosC : list num -> num;
somaPositivosC xs <= sum (filter (lambda x => x > 0, xs));
```

ANOTAÇÕES

Capítulo 6 – Funções de ordem superior

Este capítulo mostra exercícios fundamentados no capítulo 7 (ALONSO & MANZANO, 2021; ALONSO, 2019). Os exercícios com ações de compreensões mais elaborados foram subtraídos devido as características operacionais da linguagem Hope. São usados recursos de compreensão, mapeamento, filtragem, dobras e recursão, além de alguns exercícios relacionados a solução de problemas matemáticos.

6.1 - Segmento inicial verificando certa propriedade

(**Exercício 6.1.1**) Definir por recursão a função *takeWhile'* tal que [takeWhile'(p,[xs]);] seja a lista dos elementos de “xs” até o primeiro elemento que cumpra a propriedade “p”. Por exemplo,

```
takeWhile'((<7),[2,3,9,4,5]); .......  [2,3]
```

Solução:

```
takeWhile' : (alpha -> truval) # list alpha -> list alpha;
takeWhile' (_, [])      <= [];
takeWhile' (p, x :: xs) <= if p x
                           then x :: takeWhile' (p, xs)
                           else [];
```

6.2 - Complementar ao segmento inicial, verificando uma propriedade

(**Exercício 6.2.1**) Definir por recursão a função *dropWhile'* tal que [dropWhile'(p,[xs]);] seja a lista obtida a partir da eliminação dos elementos de “xs” até o primeiro elemento que cumpra a propriedade “p”. Por exemplo,

```
dropWhile'((<7),[2,3,9,4,5]); .......  [9,4,5]
```

Solução:

```
dropWhile' : (alpha -> truval) # list alpha -> list alpha;
dropWhile' (_, [])      <= [];
dropWhile' (p, x :: xs) <= if p x
                           then dropWhile' (p, xs)
                           else x :: xs;
```

6.3 - Concatenação de uma lista

(**Exercício 6.3.1**) Definir por recursão a função *concat* tal que [concat([xss]);] seja a junção dos elementos separados de “xxs” em uma única lista. Por exemplo,

```
concat([[1,3],[2,4,6],[1,9]]); ......  [1,3,2,4,6,1,9]
```

Solução:

```
concat : list (list alpha) -> list alpha;
concat []          <= [];
concat (xs :: xss) <= xs <> concat xss;
```

6.4 - Divisão de uma lista numérica por sua média

(**Exercício 6.4.1**) Dada uma lista numérica “xs” calcular o par “(ys,zs)”, sendo que “ys” contém os elementos de “xs” estritamente menores que a média, enquanto que “zs” contém os elementos estritamente maiores que a média. Por exemplo,

```
divideMedia([6,7,2,8,6,3,4]); .......  ([2,3,4],[6,7,8,6])
```

Para realizar as operações de média lembre-se de usar a função *media* definida anteriormente. Segue para conhecimento seu código.

```
media : list num -> num;
media (xs) <= sum xs / length xs;
```

Definir a função *divideMedia* tal que [divideMedia(xs);] realize a operação indicada.

Solução:

```
divideMedia : list num -> list num # list num;
divideMedia xs <= (filter (\x => x < m, xs),
                   filter (\x => x > m, xs))
                  where m == media xs;
```

6.5 - Lista com elementos consecutivos relacionados

(**Exercício 6.5.1**) Definir por recursão a função *relacionados* tal que [relacionados(r,[xs]);] verifique se todo par “(x,y)” de elementos consecutivo de “xs” cumpre a relação “r”. Por exemplo. Por exemplo,

```
relacionados((<),[2,3,7,9]); ........  true
relacionados((<),[2,3,1,9]); ........  false
```

Solução:

```
relacionados : (alpha # alpha -> truval) # list alpha -> truval;
relacionados (_, _)           <= true;
relacionados (r, x :: y :: zs) <= if x < y
                                  then relacionados (r, y :: zs)
                                  else false;
```

6.6 - Números com dígitos pares

(**Exercício 6.6.1**) Definir por recursão a função *superPar* tal que [superPar(n);] verifique se “n” é um número par tal que todos os seus dígitos são pare. Por exemplo,

```
superPar(426); ......................  true
superPar(425); ......................  false
```

Solução:

```
superPar : num -> truval;
superPar n <= if n < 0 then even n else even (n div 10);
```

(**Exercício 6.6.2**) Definir por filtragem a função *superPar2* tal que [superPar2(n);] verifique se "n" é um número par tal que todos os seus dígitos são pares. Por exemplo,

```
superPar2(426); .....................  true
superPar2(425); .....................  false
```

Solução:

```
superPar2 : num -> truval;
superPar2 n <= filter (even, digitos n) = digitos n;
```

6.7 - Lista de elementos que satisfazem uma propriedade

(**Exercício 6.7.1**) Definir por mapeamento e filtragem a função *filtraAplica1* tal que [filtraAplica1(f,p,[xs]);] seja a lista de elementos numéricos obtida a partir dos elementos de "xs" que atendam ao predicado "p" da função "f". Por exemplo,

```
filtraAplica1((4+), (<3), 1..7); ....  [5,6]
```

Solução:

```
filtraAplica1 : (num -> num) # (num -> truval) # list num -> list num;
filtraAplica1 (f, p, xs) <= map (f, filter (p, xs));
```

(**Exercício 6.7.2**) Definir por recursão a função *filtraAplica2* tal que [filtraAplica2(f,p,[xs]);] seja a lista de elementos numéricos obtida a partir dos elementos de "xs" que atendam ao predicado "p" da função "f". Por exemplo,

```
filtraAplica2((4+), (<3), 1..7); ....  [5,6]
```

Solução:

```
filtraAplica2 : (num -> num) # (num -> truval) # list num -> list num;
filtraAplica2 (f, p, [])      <= [];
filtraAplica2 (f, p, x :: xs) <= if p x
                                 then f x :: filtraAplica2 (f, p, xs)
                                 else filtraAplica2 (f, p, xs);
```

6.8 - Maior e menor elemento de uma lista

(**Exercício 6.8.1**) Definir por recursão a função *maximum'* tal que [maximum'([xs]);] seja o maior elemento existente na lista "xs". Por exemplo,

```
maximum'([3,7,2,5]); ................  7
```

Solução:

```
maximum' : list alpha -> alpha;
maximum' ([x])          <= x;
maximum' (x :: y :: ys) <= max (x, maximum' (y :: ys));
```

(**Exercício 6.8.2**) Definir por recursão a função *minimum'* tal que [minimum'([xs]);] seja o menor elemento existente na lista "xs". Por exemplo,

```
minimum'([3,7,2,5]); ................  2
```

Solução:

```
minimum' : list alpha -> alpha;
minimum' ([x])          <= x;
minimum' (x :: y :: ys) <= min (x, minimum' (y :: ys));
```

6.9 - Inversão de uma lista

(**Exercício 6.9.1**) Definir por recursão a função *inversa1* tal que [inversa1([xs]);] seja o a inversão dos elementos da lista “xs” por concatenação. Por exemplo,

```
inversa1([3,7,2,5]); ................  [5,2,7,3]
```

Solução:

```
inversa1 : list alpha -> list alpha;
inversa1 []        <= [];
inversa1 (x :: xs) <= inversa1 xs <> [x];
```

(**Exercício 6.9.2**) Definir por recursão a função *inversa2* tal que [inversa2([xs]);] seja o a inversão dos elementos da lista “xs” sem o uso de concatenação. Use o operador de construção de listas “::”. Por exemplo,

```
inversa2([3,7,2,5]); ................  [5,2,7,3]
```

Solução:

```
inversa2 : list alpha -> list alpha;
inversa2 [] <= [];
inversa2 xs <= last xs :: inversa2 (init xs);
```

6.10 - Número correspondente da lista na forma decimal

(**Exercício 6.10.1**) Definir por dobra a função *dec2ent* tal que [dec2ent([xs]);] seja o número inteiro correspondente a sua forma decimal. Por exemplo,

```
dec2ent([2,3,4,5]); .................  2345
```

Solução:

```
dec2ent : list num -> num;
dec2ent xs <= foldl (\(a, x) => 10 * a + x, 0, xs);
```

6.11 - Soma dos valores uma lista, aplicados a certa operação

(**Exercício 6.11.1**) Definir por recursão a função *somaComOper* tal que [somaComOper(f,[xs]);] seja a soma dos valores da lista “xs” aplicando a operação da função “f”. Por exemplo,

```
somaComOper((*2),[3,5,10]); .........  36
somaComOper((/10),[3,5,10]); ........   1.8
```

Solução:

```
somaComOper : (alpha -> num) # list alpha -> num;
somaComOper (f, [])      <= 0;
somaComOper (f, x :: xs) <= f x + somaComOper (f, xs);
```

6.12 - Soma das somas das listas em uma lista de listas

(**Exercício 6.12.1**) Definir por recursão a função *sumll* tal que [sumll([xss]);] seja a soma das somas das listas “xss”. Por exemplo,

```
sumll([[1,3],[2,5]]); ...............  11
```

Solução:

```
sumll : list (list num) -> num;
sumll []          <= 0;
sumll (xs :: xss) <= sum xs + sumll xss;
```

6.13 - Lista obtida apagando as ocorrências de certo elemento

(**Exercício 6.13.1**) Definir por recursão a função *apagaOcorrs* tal que [apagaOcorrs(y,[xs]);] seja a lista após remoção das ocorrências de “y” na lista “xs”. Por exemplo,

```
apagaOcorrs(5,[2,3,5,6]); ...........  [2,3,6]
apagaOcorrs(5,[2,3,5,6,5]); .........  [2,3,6]
apagaOcorrs(7,[2,3,5,6]); ...........  [2,3,5,6]
```

Solução:

```
apagaOcorrs : alpha # list alpha -> list alpha;
apagaOcorrs (y, [])      <= [];
apagaOcorrs (y, x :: xs) <= if y = x
                            then apagaOcorrs (y, xs)
                            else x :: apagaOcorrs (y, xs);
```

6.14 - Diferença de duas listas

(**Exercício 6.14.1**) Definir por recursão a função *diferenca* tal que [diferenca([xs],[ys]);] é a diferença entre os conjuntos “xs” e “ys”, ou seja, definir o conjunto dos elementos de “xs” que não pertencem ao conjunto “ys”. Por exemplo,

```
diferenca([2,3,5,6],[5,2,7]); .......  [3,6]
```

Solução:

```
diferenca : list alpha # list alpha -> list alpha;
diferenca (a, [])      <= a;
diferenca ([], b)      <= [];
diferenca (a :: ax, b) <= if member (a, b)
                          then diferenca (ax, b)
                          else a :: diferenca (ax, b);
```

6.15 - A cara e coroa de uma lista

Se denomina *coroa* de “xs” uma sublista não vazia de “xs“ formada por um elemento e os elementos seguintes até o final. Por exemplo, [3,4,5] é a coroa da lista [1,2,3,4,5].

(**Exercício 6.15.1**) Definir por recursão a função *coroa* tal que [coroa([xs]);] seja a lista das coroas da lista “xs”. Por exemplo,

```
coroa([]); ..........................  [nil] (para [[]])
coroa([1,2]); .......................  [[1,2],[2],nil]
coroa([1,2,3]); .....................  [[1,2,3],[2,3],[3],nil]
```

Solução:

```
coroa : list alpha -> list (list alpha);
coroa []       <= [[]];
coroa (x :: xs) <= (x :: xs) :: coroa xs;
```

Se denomina *cara* de “xs” uma sublista não vazia de “xs“ formada pelo primeiro elemento e os elementos seguintes até certo ponto. Por exemplo, [1,2,3] é a cara da lista [1,2,3,4,5].

(**Exercício 6.15.2**) Definir por recursão a função *cara* tal que [cara([xs]);] seja a lista das caras da lista “xs”. Por exemplo,

```
cara([]); ...........................  [nil] (para [[]])
cara([1,2]); ........................  [nil,[1],[1,2]]
cara([1,2,3]); ......................  [nil,[1],[1,2],[1,2,3]]
```

Solução:

```
cara : list alpha -> list (list alpha);
cara []       <= [[]];
cara (x :: xs) <= [] :: compNum (\ys => x :: ys, cara xs);
```

6.16 - Problema de Ullman: subconjunto de tamanho dado e com soma limitada

(**Exercício 6.16.1**) Definir a função *ullman* tal que [ullman(t,k,[xs]);] verifique se “xs” possui um subconjunto com “k” elementos cuja a soma seja menor que “t”. Por exemplo,

```
ullman(9,3,1..10); ..................  true
ullman(5,3,1..10); ..................  false
```

Solução:

```
ullman : num # num # list num -> truval;
ullman (t, k, xs) <= sum (take (k, sort xs)) < t;
```

6.17 - Decomposições de um número como a soma de dois quadrados

(**Exercício 6.17.1**) Definir a função *somaDeDoisQuadrados* tal que [somaDeDoisQuadrados(n);] seja a lista de pares de números, de modo que a soma de seus quadrados seja “n” e o primeiro elemento par seja maior ou igual ao segundo. Por exemplo,

```
somaDeDoisQuadrados(25); ............  [(5,0),(4,3)]
somaDeDoisQuadrados(16); ............  [(4,0)]
somaDeDoisQuadrados(35); ............  nil (para [])
somaDeDoisQuadrados(225); ...........  [(15,0),(12,9)]
```

Solução:

```
sddq : num # num # num -> list (num # num);
sddq (x, y, n) <= if x < y then [] else
                  if pow (x, 2) + pow (y, 2) < n
                     then sddq (x, y + 1, n) else
                  if pow (x, 2) + pow (y, 2) = n
                     then (x, y) :: sddq (x - 1, y + 1, n)
                  else sddq (x - 1, y, n);

somaDeDoisQuadrados : num -> list (num # num);
somaDeDoisQuadrados n <= sddq (ceil (sqrt n), 0, n);
```

6.18 - A identidade de Bézout

(**Exercício 6.18.1**) Definir a função *bezout* tal que [*bezout*(a,b);] seja um par de números “x” e “y” em que “a*x+b*y” é o máximo divisor comum de “a” e “b”. Por exemplo,

```
bezout(21,15); ......................  (-2,3)
```

Sugestão: Use do arquivo de suporte **mylist.hop** a função *quotRem* que retorna um par de números formado pelos quociente inteiro (a div b) e resto (a mod b) da divisão de “x” por “y” a partir do uso da operação: quotRem(a,b).

Solução:

Um exemplo de cálculo é o seguinte:

```
a b q r
36 21  1 15  (1)
21 15  1  6  (2)
15  6  2  3  (3)
 6  3  2  0
 3  0
```

Por tanto,

```
3 = 15 - 6 * 2                     [por (3)]
  = 15 - (21 - 15 * 1) * 2         [por (2)]
  = 21 * (-2) + 15 * 3
  = 21 * (-2) + (36 - 21 * 1) * 3  [por (1)]
  = 36 * 3 + 21 * (-5)
```

Sendo “q” e “r” o quociente inteiro e o resto de “a” entre “b”, “d” é o maior múltiplo comum entre “a” e “b” e “(x, y)” é o valor de (bezout(b,r)). Então,

```
a = bp + r
d = bx + ry
```

Por tanto,

```
d = bx + (a - bp)y
  = ay + b(x - qy)
```

Logo,

```
bezout(a, b) = (y, x - qy)
```

A definição de *bezout* é

```
bezout : num # num -> num # num;
bezout (_, 0) <= (1, 0);
bezout (_, 1) <= (0, 1);
bezout (a, b) <= (y, x - q * y)
                 where (x, y) == bezout (b, r)
                 where (q, r) == quotRem (a, b);
```

6.19 - Solução de uma equação diofantica

(**Exercício 6.19.1**) Neste exercício, verifica-se que a equação diofântica:

$$\frac{1}{x_1} + \frac{1}{x_2} + \cdots + \frac{1}{x_n} = 1$$

tem solução quando para todo n = 1 é possível construir uma lista de inteiros a partir do comprimento de “n” tal que a soma de seus inversos seja 1. Para fazer isso, basta observar que se [x_1, x_2 , ... , x_n] é uma solução, então [2, 2x_1, 2x_2, ..., 2x_n] também o é. Por tanto, defina a função para a equação diofantica em que “n” indica o cumprimento da lista. Por exemplo,

```
diofantica(1); ......................  [1]
diofantica(2); ......................  [2,2]
diofantica(3); ......................  [2,4,4]
diofantica(4); ......................  [2,4,8,8]
diofantica(5); ......................  [2,4,8,16,16]
```

Solução:

```
diofantica : num -> list num;
diofantica 1 <= [1];
diofantica n <= 2 :: map ((2 *), diofantica (n - 1));
```

(**Exercício 6.19.2**) Definir a função *ehDiofantica* tal que [*ehDiofantica*([xs]);] verifique se a soma dos inversos de “xs” é igual a 1. Por exemplo,

```
ehDiofantica([4,2,4]); ..............  true
ehDiofantica([2,3,4]); ..............  false
ehDiofantica(diofantica(5)); ........  true
```

Solução:

```
ehDiofantica : list num -> truval;
ehDiofantica xs <= sum (map ((1 /), xs)) = 1;
```

Capítulo 7 - Listas quase "infinitas", mas limitadas

Este capítulo apresenta exercícios fundamentados no capítulo 10 (ALONSO, 2019). Os exercícios são apresentados com o uso de listas infinitas e de avaliação preguiçosa.

OBS.:

Enquanto a linguagem Haskell opera com o conceito de listas infinitas, sem que seja necessário preocupar-se com seu limite de geração na memória o mesmo não se pode dizer da linguagem Hope.

Em Hope a geração de listas "infinitas" poderá ser realizada numa faixa muito restrita. A quantidade máxima permitida de elementos em uma lista é de aproximadamente 280.000. Por questões de segurança e para evitar estouro de memória e encerramento inesperado do ambiente é aconselhável trabalhar com listas "infinitas" limitadas até 140.000. Apesar dessa limitação, isso não prejudica a aprendizagem do conceito sobre listas infnitas. Listas infinitas em Haskell são definidas a partir da sintaxe "[1..]" e em Hope será definida a partir da função "from(1)" do módulo **mylist.hop**.

O uso de aspas na indicação do termo listas *infinitas* em Hope refere-se ao fato de nesta linguagem ser esse efeito uma simulação.

Um detalhe importante é que a maioria das funções deste capítulo usam estruturas de dados do tipo "num". Em alguns casos se mantém o uso do tipo "alpha".

Em relação as operações com números primos o recurso de geração desses valores está limitado a operar valor primos entre 2 e 5591. Valores maiores ocasionam estouro de memória. No entanto, essa limitação não afeta em absoluto o estudo dos exercícios retratados nesta obra.

7.1 - Lista "infinita" obtida a partir da repetição de um único elemento

(**Exercício 7.1.1**) Definir por compreensão a função *repete* tal que [repete(n);] seja uma lista "infinita" cujos elementos são "n". Por exemplo,

```
repete(5); ..........................  [5,5,5,5,5,5,5,...]
take(3,repete(5)); ..................  [5,5,5]
```

Solução:

```
repete : num -> list num;
repete n <= compNum (\x => n, from 1);
```

(**Exercício 7.1.2**) Definir por recursão a função *repeteFinito* tal que [repeteFinito(n,x);] seja uma lista de "n" elementos iguais a "x". Por exemplo,

```
repeteFinito(3,5); ..................  [5,5,5]
```

Solução:

```
repeteFinito : num # num -> list num;
repeteFinito (0, x) <= [];
repeteFinito (n, x) <= x :: repeteFinito (n - 1, x);
```

(**Exercício 7.1.3**) Definir usando a função *repete* e *take* a função *repeteFinito'* tal que [repeteFinito'(n,x);] seja uma lista de “n” elementos iguais a “x”. Por exemplo,

```
repeteFinito’(3,5); .................  [5,5,5]
```

Solução:

```
repeteFinito' : num # num -> list num;
repeteFinito' (n, x) <= take (n, repete x);
```

7.2 - Potências de um número menor que um limite estabelecido

(**Exercício 7.2.1**) Definir usando *takeWhile'*, *map* e *pow* a função *potenciasMenores* tal que [potenciasMenores(x,y);] seja a lista das potências de “x” menores que “y”. Por exemplo,

```
potenciasMenores(2,1000); ...........  [2,4,8,16,32,64,128,256,512]
potenciasMenores(4,1000); ...........  [4,16,64,256]
```

Solução:

```
potenciasMenores : num # num -> list num;
potenciasMenores (x, y) <= takeWhile' ((< y),
                           map (\a => pow (x, a), from 1));
```

7.3 - Agrupamento de elementos consecutivos

(**Exercício 7.3.1**) Definir usando recursão a função *agrupa* tal que [agrupa(n,[xs]);] seja a lista formada pelas listas de “n” elementos consecutivos da lista “xs”, exceto possivelmente a última lista que pode ter menos de “n” elementos. Por exemplo,

```
agrupa(2,[3,1,5,8,2,7]); ............  [[3,1],[5,8],[2,7]]
agrupa(3,[3,1,5,8,2,7]); ............  [[3,1,5],[8,2,7]]
agrupa(2,[3,1,5,8,2,7,9]); ..........  [[3,1],[5,8],[2,7],[9]]
```

Solução:

```
agrupa : num # list num -> list (list num);
agrupa (n, []) <= [];
agrupa (n, xs) <= take (n, xs) :: agrupa (n, drop (n, xs));
```

7.4 - Conjectura de Collatz

A operação a seguir é considerada aplicável a qualquer número inteiro positivo:

- se o número for par, divide-se o número por 2;
- se o número for ímpar, multiplica-se o número por 3 e em seguida soma-se 1.

A conjectura de Collatz, também chamada de *problema 3n + 1*, estabelece uma sequência numérica (ou trajetória) que a partir de um número natural inicial obedece as duas condições anteriores. Por exemplo, a trajetória de 13 é 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1 infinitamente. Se olharmos para este exemplo, a trajetória de 13 é periódica, isto é, se repete indefinidamente a partir de um determinado momento. A conjectura de Collatz diz que sempre deve-se atingir o número 1 para qualquer número com o qual se começa a trajetória. Exemplos:

- com n = 6, ter-se-á os números 6, 3, 10, 5, 16, 8, 4, 2, 1;
- com n = 11, ter-se-á os números 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1;
- com n = 27, a sucessão possui 112 passos chegando a 9232 antes de chegar a 1, com a sequência de números 27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

(**Exercício 7.4.1**) Definir a função *seguinte* tal que [seguinte(n);] seja o número seguinte na trajetória de Collatz. Por exemplo,

```
seguinte(13); .......................  40
seguinte(40); .......................  20
```

Solução:

```
seguinte : num -> num;
seguinte n <= if even n then n div 2 else 3 * n + 1;
```

(**Exercício 7.4.2**) Definir por recursão a função *collatz* tal que [collatz(n);] seja a definição da trajetória de Collatz de “n” até 1. Por exemplo,

```
collatz(13); ........................  [13,40,20,10,5,16,8,4,2,1]
```

Solução:

```
collatz : num -> list num;
collatz 1 <= [1];
collatz n <= n :: collatz (seguinte n);
```

(**Exercício 7.4.3**) Definir a função *menorCollatzMaior* tal que [menorCollatzMaior(n);] seja o menor número cuja trajetória de Collatz tem mais de “n” elementos. Por exemplo,

```
menorCollatzMaior(100); .............  27
```

Solução:

```
menorCollatzMaior : num -> num;
menorCollatzMaior x <= head (filter (\y => length (collatz y) > x,
                            from 1));
```

(**Exercício 7.4.4**) Definir a função *menorCollatzSupera* tal que [menorCollatzSupera(n);] seja o menor número cuja trajetória de Collatz tem algum elemento maior que “n”. Por exemplo,

```
menorCollatzSupera(100); ............  15
```

Solução:

```
menorCollatzSupera : num -> num;
menorCollatzSupera x <= head (filter (\y => maximum (collatz y) > x,
                              from 1));
```

7.5 - Lista com números primos

Entre os métodos e algoritmos existentes para a obtenção de números primos um, considerado simples, se destaca imensamente sendo conhecido como: Crivo de Eratóstenes. Este método permite encontrar facilmente números primos até certo limite previamente estabelecido a partir da eliminação dos números que não são primos no limite fixado.

Devido a limitações da linguagem Hope o maior limite a ser usado para a geração de números primos segundo o algoritmo do Crivo de Eratóstenes é 5591, lembrando que Hope opera de forma muito limitada com o conceito de listas “infinitas”.

O método baseia-se na criação de lista iniciada no número 2 e encerrada no valor limite estabelecido. Para este recurso use “(2..n)” estabelecido no módulo **mylist.hop**.

Após o estabelecimento da lista basta encontrar todos os múltiplos do número 2, exceto o número 2, e remove-los da lista. O próximo número da lista após o primo anterior (número 2) é primo, ou seja, o número 3. Repita a mesma ação removendo todos os múltiplos do número 3, exceto o 3. O próximo número primo da lista é 5, remova os múltiplos do número 5, exceto o 5 e assim por diante até chegar ao último elemento da lista.

Para remover múltiplos de um número de uma lista considere a função *eliminar*, tal que [eliminar(n,[xs]);] remove de uma lista “xs” os múltiplos do número “n” a partir do apoio de uma ação de filtragem.

Solução:

```
eliminar : num # list num -> list num;
eliminar (n, []) <= [];
eliminar (n, xs) <= filter (\x => x mod n /= 0, xs);
```

A partir da definição de uma função que efetue a remoção de múltiplos de certo número de uma lista é necessário passar a função que deverá aplicar o crivo de remoção a partir de cada número que compões a lista.

Observe a função *crivo*, tal que [crivo([xs]);] efetue a remoção da lista dos números que não são primos deixando ao final apenas os elemento que são primos.

Solução:

```
crivo : list num -> list num;
crivo []        <= [];
crivo (n :: ns) <= n :: crivo (eliminar (n, ns));
```

(**Exercício 7.5.1**) Definir a constante *primos* tal que [primos;] seja a apresentação de uma lista de números primos dentro do limite máximo permitido na linguagem. Por exemplo,

```
take(10,primos); ....................  [2,3,5,7,11,13,17,19,23,29]
```

Solução:

```
primos : list num;
primos <= crivo (2 .. 5591);
```

(**Exercício 7.5.2**) Definir a função *primo* tal que [primo(x);] verifique se "x" é primo. Por exemplo,

```
primo(7); ...........................  true
primo(8); ...........................  false
```

Solução:

```
primo : num -> truval;
primo x <= x = head (dropWhile' ((< x), primos));
```

7.6 - Soma dos números primos truncados

Um número primo é truncado se os números obtidos pela eliminação de algarismos da direita para a esquerda são primos. Por exemplo, 599 é um número primo truncado, pois 599, 59 e 5 são primos. O número 577 é primo não truncado, pois 57 não é primo.

(**Exercício 7.6.1**) Definir a função *primoTruncado* tal que [primoTruncado(x);] verifique se "x" é primo truncado. Por exemplo,

```
primoTruncado(599); .................  true
primoTruncado(577); .................  false
```

Solução:

```
primoTruncado : num -> truval;
primoTruncado x <= if x < 10
                   then primo x
                   else primo x and primoTruncado (x div 10);
```

(**Exercício 7.6.2**) Definir a função *somaPrimosTruncados* tal que [somaPrimosTruncados (n);] seja a soma dos "n" primos truncados. Por exemplo,

```
somaPrimosTruncados(10); ............  249
```

Solução:

```
somaPrimosTruncados : num -> num;
somaPrimosTruncados n <= sum (take (n, filter (primoTruncado, primos)));
```

(**Exercício 7.6.3**) Calcular a soma dos 20 primos truncados.

```
somaPrimosTruncados(20); ............  2551
```

7.7 - Soma dos números primos menores que "n"

(**Exercício 7.7.1**) Definir a função *somaPrimosMenores* tal que [somaPrimosMenores(n);] seja a soma dos números primos menores que "n". Por exemplo,

```
somaPrimosMenores(10); ..............  17
```

Para solucionar esta operação é necessário a definição de uma função que efetue a soma dos números menores que "n". Assim sendo, defina por recursão a função *somaMenores* tal que [somaMenores(n);] seja a soma dos números menores que "n".

Solução:

```
somaMenores : num # list num # num -> num;
somaMenores (n, x :: xs, a) <= if n =< x
                               then a
                               else somaMenores (n, xs, a + x);
```

A partir da função *somaMenores* basta desenvolver a função *somaPrimosMenores* para realizar a soma dos números primos menores que "n". Use para esta ação as funções *primos* e *soma-Menores* desenvolvidas anteriormente.

Solução:

```
somaPrimosMenores : num -> num;
somaPrimosMenores n <= somaMenores (n, primos, 0);
```

(**Exercício 7.7.2**) Definir a função *somaPrimosMenores'* tal que [somaPrimosMenores'(n);] seja a soma dos números primos menores que "n" a partir do apoio das funções *takeWhile'*, *sum* e *primos*. Por exemplo,

```
somaPrimosMenores'(10); .............  17
```

Solução:

```
somaPrimosMenores' : num -> num;
somaPrimosMenores' n <= sum (takeWhile' ((< n), primos));
```

7.8 - A bicicleta de Turing

Reza a lenda que Alan Turing tinha uma bicicleta velha. A bicicleta tinha uma corrente com um elo fraco e um dos raios da roda torto. Quando o raio torto e o elo fraco da corrente se "encontravam" a corrente rompia.

A bicicleta é identificada pelos parâmetros (i, d, n) em que:

- "i" é o número do elo que coincide com o raio torto após começar a andar;
- "d" é o número de elos percorrido a cada evolução da roda;
- "n" é a quantidade de elos da corrente, sendo "n" o elo mais fraco.

Se i = 2, d = 7 e n = 25 então a lista com a indicação do elo que coincide com o raio torto a cada volta é:

```
[2,9,16,23,5,12,19,1,8,15,22,4,11,18,0,7,14,21,3,10,17,24,6, ...]
```

A partir dos parâmetros (i = 2, d = 7, n = 25) fornecidos, sabe-se que a corrente quebrará no momento em que o número de voltas atingir 14. Veja a comprovação.

(**Exercício 7.8.1**) Definir a função *elos* tal que [elos(i,d,n);] seja a lista com o número da volta que atingem o raio torto a cada evolução da roda de uma bicicleta do tipo (i,d,n). Por exemplo,

```
take(10,elos(2,7,25)); ..............  [2,9,16,23,5,12,19,1,8,15]
```

Solução:

```
elos : num # num # num -> list num;
elos (i, d, n) <= map (\j => (i + d * j) mod n, from 0);
```

(**Exercício 7.8.2**) Definir a função *numeroDeVoltas* tal que [numeroDeVoltas(i,d,n);] seja a quantidade de voltas dadas até que a corrente se rompa em uma bicicleta do tipo (i,d,n). Por exemplo,

```
numeroDeVoltas(2,7,25); .............  14
```

Solução:

```
numeroDeVoltas : num # num # num -> num;
numeroDeVoltas (i, d, n) <= length (takeWhile' ((/= 0), elos (i, d, n)));
```

7.9 - Mais fatoriais

(**Exercício 7.9.1**) Definir a função *fatoriais* tal que [fatoriais] apresente os fatoriais dos números naturais tendendo ao infinito. Por exemplo,

```
take(7,fatoriais); ..................  [1,1,2,6,24,120,720]
```

Solução:

```
fatoriais : list num;
fatoriais <= map (fatorial, from 0);
```

(**Exercício 7.9.2**) Definir a função *ehFatorial* tal que [ehFatorial(m)] verifique se dado um número natural “m” este é o resultado de um valor obtido a partir do cálculo de um fatorial. Por exemplo,

```
ehFatorial(120); ....................  true
ehFatorial(20); .....................  false
```

Solução:

```
ehFatorial : num -> truval;
ehFatorial m <= m = head (dropWhile' ((< m), fatoriais));
```

(**Exercício 7.9.3**) Definir a função *posicoesDasFatoriais* tal que [posicoesDasFatoriais(m)] seja a lista dos fatoriais com suas posições. Por exemplo,

```
take(4,posicoesDasFatoriais); .......  [(0,1),(1,1),(2,2),(3,6)]
```

Solução:

```
posicoesDasFatoriais : list (num # num);
posicoesDasFatoriais <= zip (from 0, fatoriais);
```

Capítulo 8 - Operações com conjuntos

Este capítulo apresenta exercícios para a definição de conjuntos representados a partir de listas ordenadas (não significa que os elementos estejam classificados) sem repetições de elementos numéricos (elementos repetidos dentro de um conjunto são considerados o mesmo elemento) encontrados no capítulo 17 (ALONSO, 2019). Mesmo indicando alguns scripts com o uso do tipo de dado *alpha* as funções do capítulo estão projetadas apenas para uso com dados numéricos.

8.1 - Definições operacionais básicas

(**Base 8.1.1**) Para os exercícios deste capítulo são usados os seguintes exemplos de conjuntos:

```
c1 : list num;
c1 <= [0, 1, 2, 3, 5, 7, 9];

c2 : list num;
c2 <= [1, 2, 6, 8, 9];

c3 : list num;
c3 <= 2 .. 100000;

c4 : list num;
c4 <= 1 .. 100000;
```

(**Base 8.1.2**) Definição de conjunto vazio:

```
>: vazio;
>> nil : list num
```

```
vazio : list num;
vazio <= [];
```

(**Base 8.1.3**) Reconhecimento de conjunto vazio:

```
>: estaVazio(c1);
>> false : truval

>: estaVazio(vazio);
>> true : truval
```

```
estaVazio : list num -> truval;
estaVazio xs <= xs = nil;
```

(**Base 8.1.4**) Pertencimento de elemento em um conjunto:

```
>: pertence(3, c1);
>> true : truval

>: pertence(4, c1);
>> false : truval
```

```
pertence : num # list num -> truval;
pertence (_, [])      <= false;
pertence (a, x :: xs) <= a = x or pertence (a, xs);
```

(**Base 8.1.5**) Remoção de elementos replicados em um conjunto:

```
>: unico([1,1,2,2,3,3,4]);
>> [1, 2, 3, 4] : list num

>: unico(c1 <> c2);
>> [5, 3, 7, 0, 2, 6, 8, 1, 9] : list num
```

```
unico : list num -> list num;
unico []       <= [];
unico (x :: xs) <= if member (x, xs) then unico xs else x :: unico xs;
```

(**Base 8.1.6**) Inserção sem repetição de elemento em um conjunto:

```
>: insere(1,c1);
>> [0, 1, 2, 3, 5, 7, 9] : list num

>: insere(4,c1);
>> [0, 1, 2, 3, 4, 5, 7, 9] : list num
```

```
insere : num # list num -> list num;
insere (n, [])      <= [n];
insere (n, x :: xs) <= if n < x then n :: x :: xs else
                       if n = x then x :: xs
                       else x :: insere (n, xs);
```

(**Base 8.1.7**) Remoção de certo elemento de um conjunto:

```
>: remova(1,c1);
>> [0, 2, 3, 5, 7, 9] : list num

>: remova(4,c1);
>> [0, 1, 2, 3, 5, 7, 9] : list num
```

```
remova : num # list num -> list num;
remova (n, [])      <= [];
remova (n, xs) <= filter ((/= n), xs);
```

8.2 - Reconhecimento de subconjunto

(**Exercício 8.2.1**) Definir a função *subConjunto* tal que [subConjunto([c1],[c2]);] verifique se todos os elementos de “c1” pertencem a “c2”. Por exemplo,

```
subConjunto(2..100,1..100); .........  true
subConjunto(1..100,2..100); .........  false
```

Solução:

```
subConjunto : list num # list num -> truval;
subConjunto ([], _)       <= true;
subConjunto (_, [])       <= false;
subConjunto (x :: xs, ys) <= elem (x, ys) and subConjunto (xs, ys);
```

8.3 - Reconhecimento de subconjunto próprio

(**Exercício 8.3.1**) Definir a função *subConjPropio* tal que [subConjPropio([c1],[c2]);] verifique se o conjunto "c1'" é um subconjunto próprio de "c2". Por exemplo,

```
subConjPropio(2..5,1..7); ...........  true
subConjPropio(2..5,1..4); ...........  false
subConjPropio(2..5,2..5); ...........  false
```

Solução:

```
subConjPropio : list num # list num -> truval;
subConjPropio (c1, c2) <= subConjunto (c1, c2) and c1 /= c2;
```

8.4 - Conjunto unitário

(**Exercício 8.4.1**) Definir a função *unitario* tal que [unitario(x);] seja o conjunto "{x}", neste caso representado como "[x]". Dica: usar como apoio as funções *insere* e *vazio*. Por exemplo,

```
unitario(5); .........................  [5]
```

Solução:

```
unitario : num -> list num;
unitario x <= insere (x, vazio);
```

(**Exercício 8.4.2**) Definir a função *unitario'* tal que [unitario'(x);] seja o conjunto "{x}", neste caso representado como "[x]". Dica: realize a operação de maneira direta sem o apoio de qualquer função. Por exemplo,

```
unitario'(5); ........................  [5]
```

Solução:

```
unitario' : num -> list num;
unitario' x <= [x];
```

8.5 - Cardinal de um conjunto

(**Exercício 8.5.1**) Definir a função *cardinal* tal que [cardinal(c);] seja o número de elementos do conjunto "c". Dica: use internamente a lista passada como argumento explícito na ação. Por exemplo,

```
cardinal(c1); ........................  7
cardinal(c2); ........................  5
```

```
cardinal : list num -> num;
cardinal xs <= length xs;
```

(**Exercício 8.5.2**) Definir a função *cardinal'* tal que [cardinal'(c);] seja o número de elementos do conjunto "c". Dica: use internamente a lista passada como argumento implícito na ação. Por exemplo,

```
cardinal' (c1); ....................  7
cardinal' (c2); ....................  5
```

Solução:

```
cardinal' : list num -> num;
cardinal' <= length;
```

8.6 - União de conjuntos

(**Exercício 8.6.1**) Definir a função *uniao* tal que [uniao([c1],[c2]);] seja a união dos elementos existentes nos conjutos "c1" e "c2" com apoio de ação de concatenação. Dica: usar como auxilio as funções *unico* para remover repetições de elementos e *sort* para classificar os elementos existentes no conjunuo resultante. Por exemplo,

```
uniao(c1,c2); ......................  [0,1,2,3,5,6,7,8,9]
cardinal(uniao (c1,c2)); ...........  9
```

Solução:

```
uniao : list num # list num -> list num;
uniao (c1, c2) <= sort (unico (c1 <> c2));
```

(**Exercício 8.6.2**) Definir a função *uniao'* tal que [uniao'([c1],[c2]);] seja a união dos elementos existentes nos conjutos "c1" e "c2" com apoio de uma função auxiliar chamada *junte* tal que [junte([c1],[c2]);] efetue a junção de todos os elementos dos conjuntos "c1" e "c2". Dica: usar como auxilio as funções *unico* para remover repetições de elementos e *sort* para classificar os elementos existentes no conjunuo resultante. Por exemplo,

```
uniao'(c1,c2); .....................  [0,1,2,3,5,6,7,8,9]
cardinal' (uniao' (c1,c2)); ........  9
```

Solução:

```
junte : list num # list num -> list num;
junte ([], [])     <= [];
junte (a, [])      <= a;
junte ([], b)      <= b;
junte (a :: ax, b) <= a :: junte (ax, b);

uniao' : list num # list num -> list num;
uniao' (c1, c2) <= sort (unico (junte (c1, c2)));
```

8.7 - Intersecção de conjuntos

(**Exercício 8.7.1**) Definir a função *interseccao* tal que [interseccao([c1],[c2]);] seja a intersecção dos conjuntos "c1" e "c2". Por exemplo,

```
interseccao(1..7,4..9); ............  [4,5,6,7]
interseccao(c1,c2); ................  [1,2,9]
```

Solução:

```
interseccao : list num # list num -> list num;
interseccao (xs, []) <= [];
interseccao ([], ys) <= [];
interseccao (x :: xs, y :: ys) <=
  if x < y then interseccao (xs, y :: ys) else
  if x = y then x :: interseccao (xs, ys) else
  interseccao (x :: xs, ys);
```

8.8 - Disjunção de conjuntos

(**Exercício 8.8.1**) Definir a função *disjuntos* tal que [disjuntos([c1],[c2]);] verifique se os conjuntos “c1” e “c2” são disjuntos. Por exemplo,

```
disjuntos(2..5,6..9); ...............  true
disjuntos(2..5,1..9); ...............  false
```

Solução:

```
disjuntos : list num # list num -> truval;
disjuntos (c1, c2) <= estaVazio (interseccao (c1, c2));
```

8.9 - Diferenca simétrica de conjuntos

(**Exercício 8.9.1**) Definir a função *difSimetrica* tal que [difSimetrica([c1],[c2]);] seja a diferença simétrica entre os conjuntos “c1” e “c2” são disjuntos. Por exemplo,

```
difSimetrica(c1,c2); ................  [0,3,5,6,7,8]
difSimetrica(c2,c1); ................  [0,3,5,6,7,8]
```

Solução:

```
difSimetrica : list num # list num -> list num;
difSimetrica (c1, c2) <= diferenca (uniao (c1, c2), interseccao (c1, c2));
```

8.10 - Filtragem de conjuntos

(**Exercício 8.10.1**) Definir a função *filtra* tal que [filtra(p,[c]);] seja o conjunto de elementos de “c” que atendam ao predicado “p”. Dica: usar argumentos explícitos. Por exemplo,

```
filtra(even,c1); ....................  [0,2]
filtra(odd,c1); .....................  [1,3,5,7,9]
```

Solução:

```
filtra : (num -> truval) # list num -> list num;
filtra (p, c) <= sort (filter (p, c));
```

(**Exercício 8.10.2**) Definir a função *filtra’* tal que [filtra’(p,[c]);] seja o conjunto de elementos de “c” que atendam ao predicado “p”. Dica: usar argumentos implícitos. Por exemplo,

```
filtra'(even,c1); ..................  [2,0]
filtra'(odd,c1); ...................  [5,1,3,7,9]
```

Solução:

```
filtra' : (num -> truval) # list num -> list num;
filtra' <= filter;
```

8.11 - Aplicação de função sobre os elementos de um conjunto

(**Exercício 8.11.1**) Definir a função *mapConj* tal que [mapConj(f,[c]);] seja o conjunto formado pelos elementos de “c” operacionalizados a partir da função “f”. Por exemplo,

```
mapConj((\x => 6 - x),1..4) .........  [2,3,4,5]
```

Solução:

```
mapConj : (alpha -> alpha) # list alpha -> list alpha;
mapConj (f, c) <= sort (map (f, c));
```

8.12 - Igualdade de conjuntos

(**Exercício 8.12.1**) Definir a função *igualConj* tal que [igualConj([c1],[c2]);] verifique se os conjuntos “c1” e “c2” são iguais. Por exemplo,

```
igualConj(1..4,[1,2,3,4]); ..........  true
igualConj(1..4,[1,2,5,4]); ..........  false
```

Solução:

```
igualConj : list num # list num -> truval;
igualConj (xs, ys) <= subConjunto (xs, ys) and subConjunto (ys, xs);
```

Apêndice A - Módulo de biblioteca "mylist.hop"

Os scripts aqui apresentados caracterizam-se por serem propostas possíveis de execução a cada necessidade, mas não necessariamente as melhores propostas. Buscou-se manter o máximo de simplicidade possível para estudantes iniciantes na programação funcional.

```
! .*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*. !
!                                                        !
! Scripts auxiliares para a realização de demonstrações e !
! exercícios.                                            !
!                                                        !
! Material de apoio aos livros:                          !
!                                                        !
! * - Programe em Hope  (livro com demonstrações)        !
! * - Pense em Hope     (livro com exercícios resolvidos) !
!                                                        !
! Ambos os livros podem ser adquiridos gratuitamente no   !
! sítio: manzano.pro.br - Selecione "DOWNLOADS" e escolha !
! o material desejado.                                   !
!                                                        !
! .*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*.*. !


!! CONSTANTES PARA USO GERAL
!! ========================

!! a cláusula "dec" é obrigatória somente quando se define
!! mais de uma função na mesma linha.

dec e, pi : num;
--- e  <= exp 1;
--- pi <= acos 0 * 2;


!! FUNÇÕES DE USO GERAL (ABORDAGEM SIMPLES)
!! ========================================

!! Opera listas com faixas numéricas até 140000,
!! (limite de segurança).

infix .. : 4;
.. : num # num -> list num;
n..m <= if n > m
        then []
        else if m > 140000
             then error ("Final limit allowed: 140000")
             else n :: (succ n .. m);

!! Limite máximo permitido para simulação de listas
!! "infinitas" => 140000 (limite de segurança).

from : num -> list num;
from (n) <= if n > 140000
            then error ("Maximum limit to infinity: 140000")
            else n .. 140000;
```

```
const : alpha -> beta -> alpha;
const x _ <= x;

even : num -> truval;
even n <= n mod 2 = 0;

fst : (alpha # beta) -> alpha;
fst (x,_) <= x;

gcd : num # num -> num;
gcd (0, n) <= n;
gcd (m, n) <= gcd (floor n mod floor m, m);

lcm : num # num -> num;
lcm (_, 0) <= 0;
lcm (0, _) <= 0;
lcm (x, y) <= x * y div gcd (x, y);

max : alpha # alpha -> alpha;
max (x, y) <= if x > y then x else y;

min : alpha # alpha -> alpha;
min (x, y) <= if x < y then x else y;

odd : num -> truval;
odd n <= n mod 2 /= 0;

pred : num -> num;
pred 0     <= 0;
pred (n+1) <= n;

quotRem : num # num -> (num # num);
quotRem (a,b) <= (a div b, a mod b);

signum : num -> num;
signum n <= if n < 0 then 0-1 else
            if n = 0 then 0 else 1;

snd : (alpha # beta) -> beta;
snd (_,y) <= y;


!! FUNÇÕES PARA O TRATAMENTO DE CADEIAS (STRINGS)
!! ==============================================

isAscii : char -> truval;
isAscii c <= ord c < 128;

isLower : char -> truval;
isLower c <= 'a' =< c and c =< 'z';

isUpper : char -> truval;
isUpper c <= 'A' =< c and c =< 'Z';

isAlpha : char -> truval;
isAlpha c <= isLower c or isUpper c;

isDigit : char -> truval;
isDigit c <= '0' =< c and c =< '9';
```

```
isAlphaNum : char -> truval;
isAlphaNum c <= isAlpha c or isDigit c;

isHexDigit : char -> truval;
isHexDigit c <= isDigit c or 'a' =< c and c =< 'f' or 'A' =< c and c =< 'F';

isOctDigit : char -> truval;
isOctDigit c <= isDigit c or '0' =< c and c =< '7';

isGraph : char -> truval;
isGraph c <= ' ' =< c and c =< '~';

isControl : char -> truval;
isControl c <= isAscii c and not (isGraph c);

isPunct : char -> truval;
isPunct c <= isGraph c and c /= ' ' and not (isAlphaNum c);

isSpace : char -> truval;
isSpace c <= c = ' ' or c = '\t' or c = '\n';

toLower : char -> char;
toLower c <= if isUpper c then chr (ord c + 32) else c;

toUpper : char -> char;
toUpper c <= if isLower c then chr (ord c - 32) else c;


!! FUNÇÕES DE CONVERSÃO DE DADOS
!! ============================

charToInt : char -> num;
charToInt x <= ord (x);

digitToChar : num -> char;
digitToChar x <= chr (x + 48);

digitToInt : char -> num;
digitToInt x <= ord x - 48;


!! FUNÇÕES PARA MANIPULAÇÃO DE LISTAS
!! ==================================

all : (alpha -> truval) # list alpha -> truval;
all (_, [])      <= true;
all (p, x :: xs) <= p x and all (p, xs);

any : (alpha -> truval) # list alpha -> truval;
any (_, [])      <= false;
any (p, x :: xs) <= p x or any (p, xs);

comp : list alpha # (alpha -> truval) -> list alpha;
comp ([], f) <= [];
comp (x :: xs, f) <= if f x
                     then x :: comp (xs, f)
                     else comp (xs, f);
```

```
concat : list (list alpha) -> list alpha;
concat []        <= [];
concat (x :: xs) <= x <> concat xs;

drop : num # list alpha -> list alpha;
drop (0, xs)      <= xs;
drop (n, [])      <= [];
drop (n, x :: xs) <= drop (n - 1, xs);

dropWhile : (alpha -> truval) # list alpha -> list alpha;
dropWhile (_, [])      <= [];
dropWhile (p, x :: xs) <= if p x
                          then dropWhile (p, xs)
                          else x :: xs;

elem : alpha # list alpha -> truval;
elem (x, [])      <= false;
elem (x, y :: ys) <= x = y or elem (x, ys);

member : alpha # list alpha -> truval;
member <= elem;

filter : (alpha -> truval) # list alpha -> list alpha;
filter (_, [])      <= [];
filter (p, x :: xs) <= if p x
                       then x :: filter (p, xs)
                       else filter (p, xs);

length : list alpha -> num;
length []        <= 0;
length (x :: xs) <= 1 + length xs;

getPos : alpha # list alpha -> num;
getPos (_, [])      <= error ("Element does not exist in the list!");
getPos (n, x :: xs) <= if n = x
                       then length xs
                       else getPos (n, xs);

reverseAux : list alpha # list alpha -> list alpha;
reverseAux([], ys)    <= ys;
reverseAux(x::xs, ys) <= reverseAux(xs, x::ys);

reverse : list alpha -> list alpha;
reverse xs <= reverseAux(xs, []);

find : alpha # list alpha -> num;
find (_, [])      <= error ("Empty list!");
find (n, x :: xs) <= getPos (n, reverse (x :: xs));

foldl : (alpha # beta -> alpha) # alpha # list beta -> alpha;
foldl (f, v, [])      <= v;
foldl (f, v, x :: xs) <= foldl (f, f (v, x), xs);

foldr : (alpha # beta -> beta) # beta # list alpha -> beta;
foldr (f, v, [])      <= v;
foldr (f, v, x :: xs) <= f (x, foldr (f, v, xs));

concat2lst : list alpha # list alpha -> list alpha;
concat2lst (xs, ys) <= foldr ((::), ys, xs);
```

```
head : list alpha -> alpha;
head []       <= error ("Empty list!");
head (x :: _) <= x;

index : num # list num -> num;
index (_, []) <= error "Index out of range!";
index (0, x :: xs) <= x;
index (n, x :: xs) <= index (n - 1, xs);

init : list alpha -> list alpha;
init ([x])     <= [];
init (x :: xs) <= x :: init xs;

insert : alpha # list alpha -> list alpha;
insert (n, [])      <= [n];
insert (n, x :: xs) <= if n =< x
                       then n :: x :: xs
                       else x :: insert (n, xs);

last : list alpha -> alpha;
last ([x])     <= x;
last (x :: xs) <= last xs;

listPow : num # list num -> list num;
listPow (_, [])      <= [];
listPow (n, x :: xs) <= pow (x, n) :: listPow (n, xs);

maximum : list alpha -> alpha;
maximum []             <= error ("Empty list!");
maximum ([x])          <= x;
maximum (x :: y :: xs) <= if x > y
                          then maximum (x :: xs)
                          else maximum (y :: xs);

minimum : list alpha -> alpha;
minimum []             <= error ("Empty list!");
minimum ([x])          <= x;
minimum (x :: y :: xs) <= if x < y
                          then minimum (x :: xs)
                          else minimum (y :: xs);

map : (alpha -> beta) # list alpha -> list beta;
map (_, [])      <= [];
map (f, x :: xs) <= f x :: map (f, xs);

null : list alpha -> truval;
null []     <= true;
null (_::_) <= false;

product : list num -> num;
product xs <= foldl ((*), 1, xs);

range : num # num # num -> list num;
range (i, f, p) <= if i > f
                   then []
                   else i :: range (i + p, f, p);
```

```
reduce : list alpha # (alpha # alpha -> alpha) # alpha -> alpha;
reduce ([], f, n)      <= n;
reduce (x :: xs, f, n) <= f (x, reduce (xs, f, n));

repeat : num # alpha -> list alpha;
repeat (n, a) <= if n = 0
                 then []
                 else a :: repeat (n - 1, a);

sort : list alpha -> list alpha;
sort []        <= [];
sort (x :: xs) <= insert (x, sort xs);


take : num # list alpha -> list alpha;
take (n, [])      <= [];
take (0, xs)      <= [];
take (n, x :: xs) <= x :: take (n - 1, xs);

splitAt : num # list alpha -> list alpha # list alpha;
splitAt (n, xs) <= (take (n, xs), drop (n, xs));

splitAt' : num # list alpha -> list alpha # list alpha;
splitAt' (0, ys)      <= ([], ys);
splitAt' (_, [])      <= ([], []);
splitAt' (n, y :: ys) <= if n < 0
                         then ([], y :: ys)
                         else (y :: a, b)
                         where (a, b) == splitAt' (n - 1, ys);

sum : list num -> num;
sum []        <= 0;
sum (x :: xs) <= x + sum xs;

tail : list alpha -> list alpha;
tail []        <= error ("Empty list!");
tail (_ :: xs) <= xs;

takeWhile : (alpha -> truval) # list alpha -> list alpha;
takeWhile (_, [])      <= [];
takeWhile (p, x :: xs) <= if p x
                          then x :: takeWhile (p, xs)
                          else [];

unique : list alpha -> list alpha;
unique []        <= [];
unique (x :: xs) <= if member (x, xs)
                    then unique xs
                    else x :: unique xs;

zip : list alpha # list beta -> list (alpha # beta);
zip ([], _)            <= [];
zip (_, [])            <= [];
zip (x :: xs, y :: ys) <= (x, y) :: zip (xs, ys);
```

Apêndice B – Resumo funções predefinidas Hope

Neste apêndice são apresentados os recursos padronizados mínimos principais que acompanham a linguagem Hope do ambiente *Hope for Windows* a partir do módulo de biblioteca *Standard.hop*.

Operações aritméticas

`x - y`	é a subtração de "x" com "y"
`x * y`	é a multiplicação de "x" com "y"
`x / y`	é a divisão com quociente real de "x" sobre "y"
`x + y`	é a soma de "x" com "y"
`x div y`	é a divisão com quociente inteiro de "x" sobre "y"
`x mod y`	é o resto da divisão com quociente inteiro de "x" sobre "y"

Funções aritméticas

`abs(x)`	é o valor absoluto (sempre positivo) de "x"
`acos(x)`	é o arco cosseno de "x"
`acosh(x)`	é o arco cosseno hiperbólico de "x"
`asin(x)`	é o arco seno de "x"
`asinh(x)`	é o arco seno hiperbólico de "x"
`atan(x)`	é o arco tangente de "x"
`atan2(x)`	é o arco tangente do coeficiente dos argumentos "x" e "y"
`atanh(x)`	é o arco tangente hiperbólico de "x"
`ceil(x)`	é o arredondamentode "x" para o próximo inteiro acima
`cos(x)`	é o cosseno de "x"
`cosh(x)`	é o cosseno hiperbólico de "x"
`exp(x)`	é o exponencial natural de "x"
`floor(x)`	é o arredondamentode "x" para o próximo inteiro abaixo
`hypot(x,y)`	é o comprimento da hipotenusa de um triângulo retângulo de "x" e "y"
`log(x)`	é o logaritimo natural de "x" na base "e"
`log10(x)`	é o logaritimo natural de "x" na base 10
`pow(x, y)`	é a potência de "x" elevado a "y"
`sin(x)`	é o seno de "x"
`sinh(x)`	é o seno hiperbólico de "x"
`sqrt(x)`	é a raiz quadrada de "x"
`tan(x)`	é a tangente de "x"
`tanh(x)`	é a tangente hiperbólico de "x"

Operações relacionais

`x = y`	verifica se "x" é igual a que "y"
`x /= y`	verifica se "x" diferente de "y"
`x > y`	verifica se "x" é maior que "y"
`x < y`	verifica se "x" é menor que "y"
`x >= y`	verifica se "x" é maior ou igual a "y"
`x =< y`	verifica se "x" igual ou menor que "y"

Operações lógicas

`x and y`	ação de conjunção entre "x" e "y"
`x or y`	ação de disjunção inclusiva entre "x" e "y"
`not x`	ação de negação de "x"

Operações com listas

`xs <> ys`	concatenação das listas "xs" e "ys"
`x :: xs`	é uma lista formada pela cabeça "x" e cauda "xs"

Operações de conversão de dados

`ord('c')`	retorna o código ASCII do caractere "c" informado
`chr(n)`	retorna o caractere do código ASCII numérico "n" informado
`num2str(n)`	retorna como cadeia um valor numérico "n"
`str2num("n")`	retorna como número o conteúdo numérico na forma de cadeia em "n"

Operações diversas

`id(x)`	mostra o identificador de tipo do conteúdo "x" informado
`succ(x)`	mostra o valor sucessor de "x"
`0 - n`	definição do número "n" como negativo

Apêndice C – Pedido e autorização

Neste apêndice são indicados a reprodução do pedido de autorização para derivação do livro *Piensa en Haskell* para o livro *Pense em Hope* e a autorização fornecida.

Pedido

de: augusto(ponto)manzano(arroba)ifsp(ponto)edu(ponto)br
para: jalonso(arroba)us(ponto)es

Assunto: Acerca del libro: Piensa en Haskell

Hola, profesor José A. Alonso Jiménez, mi nombre es José Augusto Navarro García Manzano, vivo y trabajo en Brasil, también soy profesor del Gobierno Federal en IFSP (Instituto Federal de Educación, Ciencia y Tecnología de São Paulo) en la ciudad de Campos do Jordão.

Recientemente hicimos cambios en uno de nuestros cursos de programación de computadoras y estamos adoptando la enseñanza de la lógica de programación basada en la estructura funcional. Nuestro público estará formado por adolescentes y comenzaremos las actividades de estudio de la lógica con una visión filosófica pasando la mirada computacional impregnando los temas: contextualización de la filosofía en la informática - desde el origen hasta la máquina de Turing; datos, información, conocimiento y sabiduría en el contexto epistemológico y computacional; historia y desarrollo de la lógica y sus tipos; Lógica proposicional; modelo funcional; inmutabilidad; conjuntos y sus operaciones; funciones (con nombre y lambda); tipos de datos básicos; la coincidencia de patrones; recursividad (simple y cola); múltiple; divisores; cartografía; filtración; reducción. Iniciaremos el curso con lenguaje Logo para un posicionamiento introductorio y comenzaremos con un lenguaje funcional puro a elegir por el profesor.

Inicialmente, seré el ministro de disciplina. Debido a la poca edad de la audiencia, elijo usar el lenguaje Hope (resultó ser más conveniente con una sintaxis simple) y al final del período presentaré el lenguaje Haskell.

Me enteré de tu libro online "Piensa en Haskell" y tengo la intención de convertirlo en una de las referencias del curso. Aunque he visto la licencia, me gustaría preguntarle oficialmente si tengo su bendición para adaptar los ejercicios escritos en Haskell al lenguaje Hope, para que los dos libros de ejercicios se puedan usar en paralelo en nuestro curso.

Parte de nuestros estudiantes provienen de comunidades desfavorecidas con escasos recursos económicos.

Les agradezco de antemano su atención y pido disculpas por la invasión de su espacio.

Atentamente.

Augusto Manzano.

Autorização

de: jalonso(arroba)us(ponto)es
para: augusto(ponto)manzano(arroba)ifsp(ponto)edu(ponto)br

Assunto: Acerca del libro: Piensa en Haskell

Buenos días Augusto

Estoy encantado de que le guste el libro Piensa en Haskell y, por supuesto, tiene mi permiso para su adaptación a Hope.

También tienes permiso para el resto del material de la asignatura de introducción a la programación funcional.

Para cualquier cosa que necesites, no dudes en ponerte en contacto conmigo.

Atentamente, José A. Alonso.

Apêndice D – Instalação, entrada e saída do ambiente

Neste apêndice são apresentadas as instruções e orientações básicas para uso do ambiente de interpretação da linguagem Hope nos sistemas operacionais Windows 10 e Linux. A versão conhecida da linguagem Hope para macOS é destina a arquitetura PowerPC que não pôde ser verificada por não se ter acesso a equipamentos desta plataforma. Para maiores detalhes adicionais leia o segundo quadro definido na página 4 desta obra.

Linux

Para fazer uso da linguagem Hope no sistema operacional Linux execute os passos a seguir, os quais foram testados igualmente nas distribuições Fedora 33, Ubuntu 20 e OpenSUSE Leap 15:

1. Acesse o endereço: **https://github.com/dmbaturin/hope**;

2. Acione o botão verde "**Code**" e selecione "**Download ZIP**";

3. Confirme para salvar o arquivo em seu sistema;

4. Abra o gerenciador de arquivos do sistema;

5. Selecione e entre no diretório (pasta) "**Downloads**";

6. Selecione o arquivo "**hope-master.zip**" e com o botão de contexto (normalmente botão direito do mouse) escolha a opção de menu "**Abrir com gerenciador de compactação**";

7. Ao ser apresentada a tela do "**Gerenciador de compactação**" selecione a pasta "**hope-master**" e acione o botão "**Extrair**" no canto superior esquerdo;

8. Ao ser aberta a tela "**Extrair**" acione o botão "**Extrair**" no canto superior direito;

9. Concluída a operação de descompactação acione o botão "**Fechar**";

10. Feche a janela do "**Gerenciador de compactação**" acionando o botão "**x**" no canto superior direito;

11. Abra a janela de "**Terminal**" do sistema;

12. Execute na linha de comando "**cd Downloads/hope-master**";

13. Execute na sequencia a instrução "**sudo make install**" e informe a senha do usuário administrador e aguarde o término da operação;

14. Execute "**hope**" - se tudo estiver em ordem será apresentado o prompt "**>:**";

15. Para voltar ao sistema execute o comando "**exit;**".

OBS.:

Eventualmente no Linux pode ocorrer a indicação de que o programa *make* não esteja instalado. Se isso ocorrer confirme sua instalação antes de realizar a instalação do ambiente de programação Hope.

Windows

Para fazer uso da linguagem Hope no sistema operacional *Windows 10* existem duas possibilidades. Uma a partir da proposta do interpreatdor extendido *Hopeless* e outra a partir do interpretador padrão *Hope for Windows* escrita em *Visual Studio*.

Hopeless

1. Acesse o endereço: **http://shabarshin.com/funny/**;

2. Selecione o vinculo (*link*) "**hopeless_cygwin.zip**" no final da página e copie o arquivo para seu sistema;

3. Descompacte o conteúdo do arquivo "**hopeless_cygwin.zip**" em uma pasta chamada "**hopeless**" a partir da raiz de seu disco rígido principal;

4. Abra a pasta "**hopeless**", selecione o arquivo "**hopeless.exe**" e com o botão de contexto (normalmente o botão direito do mouse) selecione a opção de menu "**Criar atalho**";

5. Selecione o arquivo "**hopeless - Atalho**" com o botão de contexto do mouse, escolha a opção de menu "**Recortar**", vá até a área de trabalho dando um clique em qualquer parte e selecionando com o botão de contexto do mouse a opção de menu "**Colar**".

Hope for Windows

1. Acesse o endereço: **http://www.hope.manzano.pro.br/**;

2. Selecione o vinculo (*link*) "**hope.zip (clique aqui)**" e copie o arquivo para seu sistema;

3. Descompacte o conteúdo do arquivo "**hope.zip**" em uma pasta chamada "**hope**" a partir da raiz de seu disco rígido principal;

4. Abra a pasta "**hope**", selecione o arquivo "**hope.exe**" e com o botão de contexto (normalmente o botão direito do mouse) selecione a opção de menu "**Criar atalho**";

5. Selecione o arquivo "**hope - Atalho**" com o botão de contexto do mouse, escolha a opção de menu "**Recortar**", vá até a área de trabalho dando um clique em qualquer parte e selecionando com o botão de contexto do mouse a opção de menu "**Colar**".

Outra forma de acessar qualquer um dos ambientes usados no sistema operacional Windows é abrir a jenala de "**Prompt de comando**", entrar no diretório (pasta) da linguagem e efetuar a chamada do arquivo executável.

Referências bibliográficas

BAILEY, R. **A HOPE Tutorial**: *Using one of the new generation of functional languages*. BYTE, Peterborough, v. 10, n. 8, p. 235-258, aug, 1985.

______. **Functional Programming with Hope (Ellis Horwood Series in Computers and Their Applications)**. Chichester: Ellis Horwood Ltd, 1990.

BURSTALL, R. M., MACQUEEN, D. B. & Sannlla D. T. **HOPE**: *An experimental applicative language*. Dissertação em ciência da computação - Universidade de Endiburgo. Escócia, p. 136-146. 1978.

ALONSO JIMÉNEZ, José A. **Temas de "Programación funcional"**: *curso 2019-20*. Sevilla: Universidad de Sevilla, 2016. Disponível em: http://www.cs.us.es/~jalonso/cursos/i1m-19/temas/2019-20-I1M-temas-PF.pdf. Acesso em: 20 ABR.2021.

______; DOBLADO, Ma. José Hidalgo. **Piensa en Haskell**: *Ejercicios de programación con Haskell*. Sevilla: Universidad de Sevilla, 2012. Disponível em: http://www.cs.us.es/~jalonso/publicaciones/Piensa_en_Haskell.pdf. Acesso em: 4 mar.2021.

______; MANZANO, José Augusto N. G. **Programe em Hope**: *Lógica de programação funcional – programação funcional na prática*. São Paulo: Grupo de Lógica Computacional/Propes Vivens, 2021. Disponível em: https://novo.manzano.pro.br/wp/downloads/. Acesso em: 20 abr.2021.

PATERSON, R. A HOPE interpreter: Reference. 2000. Disponível em: <http://www.stolyarov.info /static/hope/ref_man.pdf>. Acesso em: 5 mar. 2021.

POLYA, G. **Cómo plantear y resolver problemas**. Editorial Trillas: México D. F., 1978.

THOMPSON, Simon. **How to program it**. University of Kent - Computing Laboratory: Canterbury, 1996. Disponível em: https://www.cs.kent.ac.uk/people/staff/sjt/Haskell_craft/HowToProgIt.html. Acesso em: 5 mar.2021.

ANOTAÇÕES

Índice remissivo

Pense em Hope

LÓGICA DE PROGRAMAÇÃO FUNCIONAL

EXERCÍCIOS DE PROGRAMAÇÃO FUNCIONAL COM HOPE

ESTE É UM LIVRO DE EXERCÍCIOS COM RESPOSTAS PARA AUXILIAR A AMPLITUDE MENTAL DE COMO TRABALHAR OS PRINCÍPIOS LÓGICOS DA PROGRAMAÇÃO FUNCIONAL.

A LINGUAGEM HOPE FOI ESCOLHIDA POR SER SIMPLES E DE FÁCIL USO, SERVINDO DE SUPORTE AO APRENDIZADO DA LÓGICA FUNCIONAL E NÃO DA LINGUAGEM EM SI.

LINGUAGENS DE PROGRAMAÇÃO SÃO MERAS FERRAMENTAS E ASSIM DEVEM SEMPRE SEREM CONSIDERADAS. A ESSÊNCIA PROFISSIONAL DA PROGRAMAÇÃO DE COMPUTADORES É DESENVOLVER A CAPACIDADE MENTAL SOB O ASPECTO DE CERTO PARADIGMA DE PROGRAMAÇÃO E NÃO FOCAR SUAS HABILIDADES EM FERRAMENTAS.

ISBN 978-65-00-19581-1
9 786500 195811