

Programe em Hope

LÓGICA DE PROGRAMAÇÃO FUNCIONAL

PROGRAMAÇÃO FUNCIONAL NA PRÁTICA



JOSÉ A. ALONSO JIMÉNEZ
JOSÉ AUGUSTO N. G. MANZANO

LÓGICA DE PROGRAMAÇÃO FUNCIONAL

Programe em HOPE

PROGRAMAÇÃO FUNCIONAL NA PRÁTICA

Revisão 1.0 – 26/04/2021



José A. Alonso Jiménez
José Augusto N. G. Manzano

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

Alonso Jiménez, José A.

Lógica de programação funcional [livro eletrônico] : programa em Hope : programação funcional na prática / José A. Alonso Jiménez, José Augusto Navarro Garcia Manzano. -- 1. ed. -- São Paulo : Grupo de Lógica Computacional, 2021.
PDF

Bibliografia

ISBN 978-65-00-21446-8

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

21-63512

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

José Augusto N. G. Manzano

ORCID: 0000-0001-9248-7765

LÓGICA DE PROGRAMAÇÃO FUNCIONAL PROGRAME EM HOPE

PROGRAMAÇÃO FUNCIONAL NA PRÁTICA

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

Temas de "Programación funcional" (curso 2019-20)

<https://www.cs.us.es/~jalonso/cursos/i1m/temas.php>

Autor: José A. Alonso Jiménez

Edição: Grupo de Lógica Computacional
Departamento de Ciências da Computação e Inteligência Artificial
Universidade de Sevilla – Espanha

Ano: 2020

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.

Esta obra está licenciada, assim como a original base, sob a licença:
Reconhecimento-NãoComercial-CompartilharIgual 2.5 Espanha Creative Commons.

É permitido:



- copiar, distribuir e comunicar publicamente a obra;
- fazer obras derivadas.

Sob as seguintes condições:

(BY) Reconhecimento: devem ser dados os créditos da obra de forma específica aos autores.

(NC) Não-comercial: não pode utilizar esta obra com finalidades comerciais.

(SA) Compartilhar sob a mesma licença: Se você alterar ou transformar este trabalho, ou gerar um trabalho derivado, você só pode distribuir o trabalho gerado sob uma licença idêntica à esta.

- Ao reutilizar ou distribuir a obra, você deve cumprir os termos expressos a esta obra.
- Algumas dessas condições podem não se aplicar se a permissão for obtida a partir do proprietário do direito autoral, neste caso, José A. Alonso Jiménez.

Este é um resumo do texto legal (licença completa). Para ver uma cópia desta licença, visite o endereço: <https://creativecommons.org/licenses/by-nc-sa/2.5/es/deed.pt> ou envie uma carta para Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

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 própria linguagem.

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 como ocorre na programação imperativa.

Este livro vem complementar os poucos livros de programação funcional publicados no Brasil relacionados ao estudo da lógica de programação funcional em si e não, necessariamente, sobre determinada linguagem de programação. Para tornar o estudo atraente a dinâmica de apresentação deste livro é usar pouca teoria e lançar mão de ações práticas sobre a linguagem Hope. Não com o objetivo de ensinar Hope, mas de colocar em ação os detalhes apresentados e incentivar você a estudar outras obras do assunto.

Além deste livro você pode ter acesso ao livro "Pense em Hope" também em parceria entre os professores José A. Alonso Jiménez da Universidade de Sevilla e José Augusto Manzano do Instituto Federal de São Paulo.

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, teste todos os exemplos, tente entender o que cada exemplo mostra.

O capítulo 1 faz introdução apresentando um breve histórico dos antecedentes que fundamentam a programação funcional e a linguagem Hope.

No capítulo 2 é descrito o acesso ao ambiente de programação e seu uso a partir de ações interativas e de aplicações de funções básicas. Apresenta-se algumas regras iniciais a serem seguidas em todo o estudo.

O capítulo 3 apresenta os tipos de dados suportados pela linguagem que são em médias os tipos mais populares encontrados na área da programação funcional.

No capítulo 4 tem-se contato com o desenvolvimento de funções por composição, condição e correspondência de padrões, além do uso de funções lambda.

O capítulo 5 apresenta como é possível operar compreensões de listas a partir das limitações da linguagem Hope. Um dos destaques do capítulo é o desenvolvimento de funções criptográficas baseada na cifra de César.

No capítulo 6 você tem contato com a principal estrutura de programação de uma linguagem funcional: a recursão. São apresentados os efeitos de recursão numérica, em listas sobre argumentos e recursões múltiplas além da apresentação sobre a heurística ser considerada para a definição de funções recursivas.

O capítulo 7 mostra o uso de funções especializadas, descreve o que são funções de ordem superior e apresenta diversos exemplos de processamento de listas, composição de funções e efeitos de dobra à esquerda e a direita. O capítulo ainda mostra um estudo de caso sobre a simulação de transmissão de cadeias.

No capítulo 8 é apresentado os recursos de definição e customização de tipos de dados. Nesta etapa você aprende como criar seus próprios tipos de dados buscando-se ter um código de programa concizante com a estrutura de dados em uso.

O capítulo 9 apresenta detalhes sobre o tema: avaliação preguiçosa descrevendo como essa forma de processamento é realizada em uma linguagem funcional como Hope.

No capítulo 10 você tem contato com a criação e gerenciamento das próprias bibliotecas de funcionalidades baseadas em funções customizadas. É nesta etapa que você aprende como criar arquivos de módulo como é o caso do componente "**mylist.hop**" que acompanhe esta obra.

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

1. INTRODUÇÃO À PROGRAMAÇÃO FUNCIONAL

1.1.	Antecedentes históricos	9
1.2.	Linguagem Hope	10
1.3.	Paradigma declarativo funcional	12

2. INTRODUÇÃO À PROGRAMAÇÃO COM HOPE

2.1.	Acesso ao ambiente operacional	17
2.2.	Ações interativas	19
2.3.	Aplicação de funções	22
2.4.	Scripts Hope	23
2.5.	Atribuição de nomes e comentários	25

3. TIPOS DE DADOS

3.1.	Tipos básicos	27
3.2.	Tipos compostos	27
3.3.	Tipos funções	28
3.4.	Inferência de tipos	29
3.5.	Parcialização	30
3.6.	Polimorfismo	31

4. DEFINIÇÃO DE FUNÇÕES

4.1.	Por composição	33
4.2.	Por condição	33
4.3.	Por correspondência de padrões	34
4.4.	Expressões lambda	36

5. DEFINIÇÕES DE LISTAS POR COMPREENSÃO

5.1.	Geradores	39
5.2.	Guardas	39
5.3.	A função "zip"	40
5.4.	Compreensão de cadeias	40
5.5.	Cifra de César	41

6. FUNÇÕES RECURSIVAS

6.1.	Recursividade numérica	45
6.2.	Recursividade sobre listas	46
6.3.	Recursão sobre vários argumentos	48
6.4.	Recursão múltipla	49
6.5.	Heurística para as definições recursivas	49

7. FUNÇÕES ESPECIALIZADAS

7.1. Funções de ordem superior	53
7.2. Processamento de listas	53
7.3. Função de dobra à direita (foldr)	55
7.4. Função de dobra à esquerda (foldl)	57
7.5. Composição de funções	58
7.6. Estudo de caso: Codificação binária e transmissão de cadeias	58

8. TIPOS DE DADOS CUSTOMIZADOS

8.1. Declaração de tipos	63
8.2. Definição de tipos	64
8.3. Definição de tipos recursivos	67

9. AVALIAÇÃO PREGUIÇOSA

9.1. Estratégias de avaliação	71
9.2. Terminação	72
9.3. Número de reduções	72
9.4. Estruturas infinitas	73
9.5. Programação modular	75

10. MÓDULOS OU BIBLIOTECAS

10.1. Estrutura de um módulo	77
10.2. Definição de prioridade de funções	82
10.3. Encapsulamento de funções	83

APÊNDICES

A. Pedido e autorização	85
B. Método de Pólya para resolução de problemas	87
C. Instalação, entrada e saída do ambiente	91
D. Resumo funções predefinidas Hope	93
E. Palavras reservadas Hope	95
F. Módulo: mylist.hop	97

REFERÊNCIAS BIBLIOGRÁFICAS	105
----------------------------------	-----

ÍNDICE REMISSIVO	107
------------------------	-----

CAPÍTULO 1 - Introdução à programação funcional

A programação funcional caracteriza-se por ser um paradigma voltado a codificação de programas a partir da definição de funções. A ideia central de *funções* na programação é a mesma encontrada na Ciência Matemática que determina que uma função é a relação entre dois conjuntos abrangendo todos os elementos do primeiro conjunto e associando-os somente a um elemento do segundo conjunto. Este estilo de programação proporciona como vantagem o desenvolvimento de programas concisos e de fácil manutenção. Esses programas podem fazer uso de: potentes tipos de dados na representação de estruturas; operações com ações de recursividade; compreensões de conjuntos na forma de listas e/ou tuplas; aplicações de funções de ordem superior, além de avaliações preguiçosas, entre outras vantagens (O'SULLIVAN B., STEWART D. & GOERZEN J., 2008; RUIZ, et al., 2004; BIRD & WADLER, 1998; HUTTON, 2007). Assim sendo, leia atentamente este capítulo sem realizar em computador os exemplos de códigos indicados, deixe isso a partir do próximo capítulo.

1.1 - Antecedentes históricos

Por incrível que pareça, o início desta história começa no final da década de 1920, desenrolando-se durante a década de 1930. Em 1927 Haskell Brooks Curry e Moses Schönfinkel apresentam a teoria da lógica combinatória (FURRUGIA, 2021), 1931 Kurt Gödel apresenta a teoria das funções recursivas (ZACH, 2007), 1936 Alan Turing apresenta sua máquina teórica fundamentando diversas operações computacionais (RENDELL, 2016) e Alonzo Church desenvolve a teoria do famoso *cálculo lambda* que tornou-se a base de constituição e fundamentação das linguagens funcionais (BARENDREGT & BARENDSEN, 2000). Em 1936 Stephen Cole Kleene apresenta a teoria sobre funções computáveis (HOFSTRA & SCOTT, 2020).

Durante a década de 1940 a maior contribuição foi a de Emil Leon Post que em 1943 apresenta a ideia de uma máquina de estado finito (SURHONE, 2010). Algum tempo depois, durante a década de 1950 ocorrem dois eventos muito importantes: 1954 surgimento da primeira linguagem de programação imperativa (semiestruturada) de alto nível conhecida como FORTRAN (FORMula TRANslator) criada por John Backus (BACKUS, 1998) e em 1958 o surgimento da primeira linguagem declarativa funcional chamada Lisp (List processing) criada por John McCarthy (MC-CARTHY, 1978). Só para dar ideia, o paradigma conhecido como *programação orientada a objetos* surge em 1965 (OWE; KROGDAHL & LYCHE, 2004).

No período da década de 1960 surgiu a primeira linguagem de *programação funcional pura* chamada ISWIN apresentada em 1965 por Peter Landin (LANDIN, 1966). As linguagens da categoria de programação funcional puras são baseadas totalmente no uso de funções e tipos de dados. Usam para a tomada de decisões, além do tradicional "if / then / else", o efeito de correspondência de padrões. Para ações de repetições fazem uso de ações recursivas. A propósito, existem linguagens de programação funcional denominadas impuras, sendo normalmente linguagens que atendem ao paradigma funcional em conjunto com outros paradigmas de programação. Essas linguagens lançam mão de ações de repetição baseadas em laços, deixando o uso de recursividade em segundo plano.

Na década de 1970 ocorrem importantes eventos ao mundo da programação funcional, surgem as linguagens funcionais puras: SASL de 1976 por David Turner (SOMMERVILLE, 1977 & LÉVÉNEZ, 2021), FP de 1978 por John Backus (BACKUS, 1978); ML de 1973 por Robin Milner (MILNER, 1978); NPL (1977) de Rod Burstall e John Darlington (HOWE, 1985); Hope de 1978 por Rod M. Burstall, David B. Macqueen e Donald T. Sannella (GABRIÉLS, GERRITS, KOOIJMANS, 2007).

Já na década de 1980 surge Miranda de 1985 por David Turner (MICHAELSON, 2011) e em 1987 desenvolve-se um comitê para a definição do que se conhece hoje como linguagem Haskell, o qual publica no ano de 2003 o documento "Haskell Report" (JIMÉNEZ, 2019).

As linguagens funcionais impuras começam a surgir a partir da década de 1980 tendo como exemplares as linguagens: Standard ML de 1983 (MACQUEEN; HARPER & REPPY, 2020), Caml de 1985 (INRIA, 2005), OCaml de 1996 (SMITH, 2006), F# de 2005 (FANCHER, 2014) e Elixir de 2011 (THE ELIXIR TEAM, 2014).

Atualmente encontra-se a programação funcional em tão importante estágio de uso que linguagens de programação tradicionais que operam sob o paradigma imperativo como: C++, C#, Java, Rust, entre outras estão dando suporte a esse paradigma ou estão sendo desenvolvidas com a capacidade de realizarem operações funcionais.

1.2 - Linguagem Hope

A linguagem Hope foi apresentada em 1978 na *Universidade de Edimburgo* por Rod Burstall, Donald Sannella e John Darlington caracterizando-se por ser uma pequena linguagem de programação baseada na composição de funções, sendo ideal para o ensino das técnicas de programação funcional a iniciantes, por ser uma linguagem descontraída, divertida e fácil de usar. Sua maior influência veio da linguagem NPL de Rod Burstall e John Darlington em seu trabalho na transformação de programas apresentado à Universidade de Edimburgo e da linguagem ISWIN de Peter Landin (LANDIN, 1966; DPL, 2000). Segundo o sítio "CodeLani - <https://codelani.com/languages/hope.html>" Hope é uma linguagem de programação classificada com grua de importância de 25% das linguagens de programação existentes.

As linguagens NPL e Hope foram as primeiras linguagens a utilizarem tipos de dados algébricos, correspondência de padrões, funções anônimas e outros recursos funcionais. Um dos padrões mais envolventes da NPL é a avaliação de compreensões de listas. No entanto Hope não adotou este recurso, o que é lamentável de um lado, mas interessante de outro ao forçar que o estudante tenha que abstrair as operações de compreensão, buscando outras formas de solução. No entanto, o efeito de compreensão de listas ressurgiu a partir da linguagem ML da Universidade de Edimburgo e contemporânea a Hope, escrita por Robin Milner (MACQUEEN, HARPER & REPPY, 2020) e vem sendo mantido a partir de então em diversas linguagens de programação funcional.

Um programa Hope é composto a partir de uma ou mais funções. Cada função pode ser formada por expressões aritméticas (ou algébricas), expressões lógicas, por conjuntos de equações simples e/ou por recursão. Os tipos de dados primitivos suportados pela linguagem incluem: números, símbolos, cadeias, caracteres, listas, tuplas e dados polimórficos. As listas são formadas por elementos de mesmo tipo, enquanto que as tuplas podem ser heterogêneas. Hope aceita, além do uso de tipos de dados primitivos (tipos de dados fortes) o uso de tipos de dados derivados (DPL, 2000; BAILEY, 1990). Hope é uma linguagem funcional que segue o modelo "*top-down*", isso significa que as funções devem ser constituídas na ordem em que serão usadas: primeiro a função base e posteriormente função que usará a função base. É pertinente salientar que quando Hope foi lançado a definição de tipos de dados derivados não era uma prática comum usada nas linguagens de programação da época, como ocorre nos dias de hoje.

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, pelo que tudo indica, não se tornou disponível de forma pública. Um tempo depois a *British Telecom* em parceria com o *Imperial College* de Londres (ICL) implementou uma versão do interpretador Hope em 1986 chamada *IC-HOPE* escrito por Victor Wu Way Hung, J. Kewley, T. Thomson e outros. A primeira implementação da linguagem usou avaliação ansiosa (*eager*

evaluation)¹, posteriormente mudou-se para avaliação preguiçosa (*lazy evaluation*)² mais avançada e eficiente (DPL, 2000). Detalhes a este respeito são fornecidos ao longo desta obra.

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 (tendo trabalhado anteriormente no ICL) 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 e adaptado a partir de Sharbarshin, escrito no ambiente *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 Hope (e não a linguagem em si) são ferramentas simples podendo apresentar bugs de funcionamento que no geral não atrapalham em demasia o uso operacional do ambiente de programação, bem como a linguagem.

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. Basta que ao final deste estudo você estude o código do arquivo de módulo "**mylist.hop**".

Os scripts apresentados nesta obra 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 *Notas, tópicos e códigos da "Informática (2019-20)"*.

¹ Forma de avaliação na qual uma expressão é avaliada quando encontrada tendo seu resultado vinculado a uma variável, sendo o estilo mais comum encontrado nas linguagens de programação.

² Forma de avaliação na qual o resultado da expressão é atrasado até o ponto em que for considerado suficientemente necessário.

1.3 - Paradigma declarativo funcional

A programação funcional caracteriza-se por ser um modelo de programação baseado na concepção e avaliação de funções matemáticas a partir do fornecimento de certos argumentos. Este estilo de programação pertence ao ramo da *programação declarativa*, podendo assim ser referenciado como: *paradigma declarativo funcional*. Nesse paradigma os programas focam o resultado do problema sem indicar o que efetivamente o computador deve fazer a cada etapa de seu processamento, diferente-mente do modelo usado na *programação imperativa* que se preocupa no como fazer.

A programação imperativa é um estilo de programação em que os programas são escritos por instru-ções que especificam (detalhadamente) como se deve calcular certo resultado. Como exemplo, ob-serve o código imperativo a seguir codificado em *PORTUGUÊS ESTRUTURADO* e *Pascal* que tem por finalidade calcular e apresentar a soma dos "n" primeiros números inteiros.

programa soma		program soma;
var		var
n, contador, total : inteiro		n, contador, total : integer;
início		begin
leia n		readln(n);
contador <- 0		contador := 0;
total <- 0		total := 0;
repita		repeat
contador <- contador + 1		contador := contador + 1;
total <- total + contador		total := total + contador;
até_que (contador = n)		until (contador = n);
escreva total		writeln(total);
fim		end.

Quando executado o programa imperativo chamado "**soma**" com o fornecimento de uma entrada com valor "**4**" ter-se-á como evolução do cálculo para obtenção da soma os seguintes passos:

CONTADOR	TOTAL
0	0
1	1
2	3
3	6
4	10

Agora veja, como exemplo, um código do mesmo problema escrito de acordo com a estrutura funci-onal na linguagem Hope. Observe o tamanho do programa funcional em relação ao programa impe-rativo.

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

Quando executada a função "**soma**" fornecendo o valor numérico "**4**" ter-se-á como evolução do cálculo para obtenção do resultado os seguintes passos:

```
soma 4
= sum (1..4)      [definição de soma]
= sum [1,2,3,4]   [definição de soma]
= 1 + 2 + 3 + 4   [definição de sum]
= 10              [definição de "+"]
```

Uma função Hope é obrigatoriamente declarada a partir de um cabeçalho (protótipo da função) identificada a partir da cláusula "**dec**" que *solicita* a reserva de espaço de trabalho na memória do computador para que a função possa ser operacionalizada. Neste caso, pela instrução "**dec soma : num -> num;**", em que "**soma**" é o nome de identificação da função, ":" é um símbolo lido como "*para o argumento*" (ou "*para os argumentos*" quando houver mais de um valor), o "**num**" antes operador de retorno "->" representa o argumento de entrada, "**num**" após "->" representa o valor de resposta da função. O operador de retorno "->" é um símbolo lido como "*retorne*". Desta forma a instrução "**soma : num -> num;**" deve ser lida como "*função [soma] para o argumento [:] do tipo (num) de modo que retorne [->] um resultado do tipo (num)*".

A partir da declaração do protótipo da função faz-se a definição da parte operacional da função com a cláusula "---" definida com a instrução "**soma n <= sum (1..n);**", em que "**n**" é o argumento formal de entrada de valor na função definido à esquerda do símbolo de asserção "<=" que possui à direita a definição da ação operacional a ser executada como sendo o somatório (função "**sum**") de "**1**" até "**n**".

OBS.:

É comum que linguagens funcionais ofereçam um conjunto interno de funções prontas para facilitar o uso de operações triviais. Hope não é diferente. No entanto, o conjunto de funcionalidades internas da linguagem Hope é bastante limitado. Por este motivo o estudo deste livro deve ser operacionalizado com o carregamento do arquivo de módulo "**mylist.hop**", onde se encontra entre diversos recursos as definições das funcionalidades "**sum**" e "**(1..n)**".

A título de curiosidade veja os códigos das funcionalidades "**sum**" e "**(1..n)**" definidas no arquivo de módulo "**mylist.hop**":

```
dec .. : 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);

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

Não se preocupe em entender logo de imediato os códigos aqui apresentados.

Note que uma função é uma aplicação que toma um ou mais argumentos e devolve um resultado. O retorno do resultado deverá sempre ser o mesmo se informado os mesmos parâmetros. Na programação funcional as funções devem seguir a regra: *mesmos argumentos, mesmo resultado* não importando quantas vezes a função é executada e avaliada.

Na linguagem Hope as funções são definidas mediante o uso de **equações** formadas pelo **nome da função**, pelas **identificações dos argumentos** e o **corpo** que especifica como o calcula do valor é obtido a partir dos argumentos fornecidos.

Veja outros exemplos de definição de funções em Hope. Atente para os códigos das funções *dobro*, *somatorio* e *classifica*.

Atente para os detalhes de operação da função "**dobro**":

```
dec dobro : num -> num;
--- dobro x <= x + x;
```

Veja um exemplo da avaliação da execução da função "**dobro**" na memória:

```
dobro 3
= 3 + 3 [define dobro]
= 6     [define "+"]
```

Exemplo de avaliação ansiosa:

```
dobro (dobro 3)
= dobro (3 + 3) [define dobro]
= dobro 6      [define "+"]
= 6 + 6        [define dobro]
= 12           [define "+"]
```

Exemplo de avaliação preguiçosa:

```
dobro (dobro 3)
= (dobro 3) + (dobro 3) [define dobro]
= (3 + 3) (dobro 3)    [define dobro]
= 6 + (3 + 3)          [define "+"]
= 6 + (dobro 3)        [define dobro]
= 6 + 6                [define "+"]
= 12                   [define "+"]
```

Veja um exemplo da avaliação da execução da função "**somatorio**" na memória a partir do uso de recursividade sobre listas. A função "**somatorio lista**" calcula a soma de todos os elementos da lista informada. Ao ser executada a função "**somatorio [2,3,7];**" o resultado retornado será "**12**".

Definição:

```
dec somatorio : list num -> num;
--- somatorio [] <= 0;
--- somatorio (x :: xs) <= x + somatorio xs;
```

Avaliação:

```
somatorio [2,3,7]
= 2 + somatorio [3,7] [define somatorio]
= 2 + (3 + somatorio [7]) [define somatorio]
= 2 + (3 + (7 + sum [])) [define somatorio]
= 2 + (3 + (7 + 0))      [define somatorio]
= 12                     [define "+"]
```

Observe um exemplo de classificação de elementos de uma lista utilizando o efeito de compreensão. É pertinente salientar que Hope não possui internamente ferramental para operar compreensões. Desta forma, considere a função "**comp**" definida no arquivo de módulo "**mylist**" que efetua a compreensão simples de listas.

```
dec 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);
```


A partir da função "**comp**" é possível criar a função "**classifica**" que efetua a classificação crescente dos elementos de uma lista. A função "**classifica(x)**" a seguir é baseada no algoritmo de classificação rápida: *quick sort*.

Exemplo:

```
classifica [4,6,2,3]; == [2,3,4,6]
classifica "deacb";  == "abcde"
```

Definição:

```
dec classifica : list alpha -> list alpha;
--- classifica []      <= [];
--- classifica (x :: xs) <= classifica menores <> [x] <> classifica maiores
                        where menores == comp (xs, \a => a <= x)
                        where maiores == comp (xs, \b => b > x);
```

Avaliação do exemplo com listas de compreensão:

classifica [4,6,2,3]	
= classifica [2,3] <> [4] <> classifica [6]	[define classifica]
= classifica [] <> [2] <> classifica [3] <> [4] <> classifica [6]	[define classifica]
= [] <> [2] <> classifica [3] <> [4] <> classifica [6]	[define classifica]
= [2] <> classifica [3] <> [4] <> classifica [6,5]	[define "<>"]
= [2] <> classifica [] <> [3] <> [] <> [4] <> classifica [6]	[define classifica]
= [2] <> [] <> [3] <> [] <> [4] <> (classifica [6])	[define classifica]
= [2] <> [3] <> [4] <> classifica [6]	[define "<>"]
= [2,3] <> [4] <> classifica [6]	[define "<>"]
= [2,3,4] <> classifica [6]	[define "<>"]
= [2,3,4] <> classifica [] <> [6] <> classifica []	[define classifica]
= [2,3,4] <> classifica [] <> [6] <> classifica []	[define classifica]
= [2,3,4] <> [] <> [6] <> []	[define classifica]
= [2,3,4,6]	[define "<>"]

Uma característica importante do paradigma funcional são as operações baseadas sobre *objetos imutáveis* (chamados estranhamento, por alguns, de *variáveis imutáveis*). Quando o valor de um objeto (ou variável) é definido na memória, este não poderá ser alterado. Em vez disso, cópias do objeto são definidas na memória. Esta operação aumenta o nível de segurança dos dados processados e simplifica a depuração dos programas.

ANOTAÇÕES

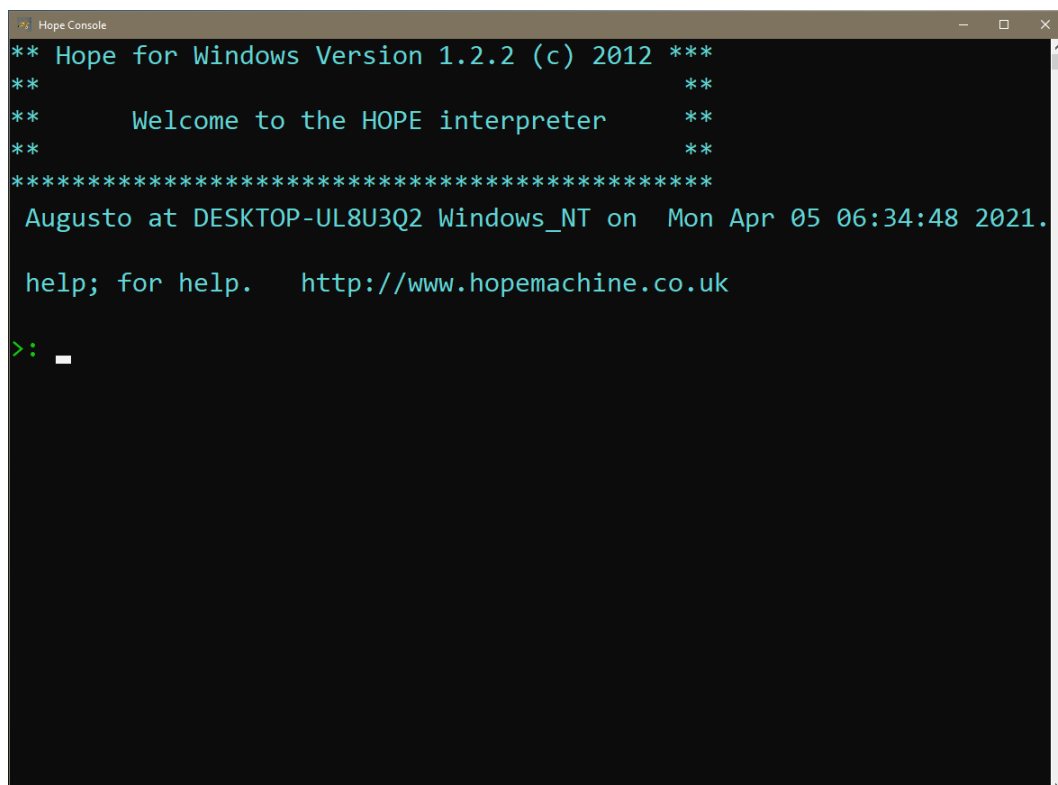
CAPÍTULO 2 - Introdução à programação com Hope

É importante saber diferenciar a linguagem Hope de seu ambiente interativo de funcionamento. A linguagem é constituída por um conjunto de cláusulas que são as palavras-chave que representam os comandos da linguagem como: "**dec**", "**---**", "**num**", "**list**", "**alpha**", "**truval**", "**where**", "**if**", "**then**", "**else**" entre outros que são apresentados ao longo deste estudo. Já o ambiente é o programa que você chama em seu computador, é a imagem da janela aberta em seu sistema operacional, podendo ser o ambiente "Hope for Windows" ou "Hopeless" para o sistema operacional Windows ou ainda o ambiente "hope" no sistema operacional Linux. O conjunto linguagem e ambiente formam o que se chama de *ecossistema hope*. Com o objetivo de diferenciar graficamente a linguagem do ambiente será usada para a linguagem a grafia "Hope" com a primeira letra em maiúsculo e para o ambiente ou ecossistema a grafia "*hope*" em minúsculo e itálico. É importante ressaltar que os programas funcionais podem ser avaliados manualmente (como no capítulo anterior) e que as linguagens funcionais avaliam automaticamente os programas funcionais (BIRD & WADLER, 1998; BAILEY, 1985 & 1990).

2.1 - Acesso ao ambiente operacional

Para se ter acesso ao ambiente *hope* é importante que o software de gerenciamento da linguagem esteja instalado e devidamente configurado para acesso. Caso você não o tenha em seu computador é necessário primeiro baixar o programa e proceder sua instalação e configuração, dependendo do sistema operacional em uso. As informações de como fazer isso encontram-se no "*Apêndice C - Instalação, entrada e saída do ambiente*".

A partir do pressuposto de que você tem acesso ao ambiente *hope* efetue sua chamada como orientado no "*Apêndice C*". O visual do ambiente "Hopeless" para o sistema operacional Windows é semelhante ao visual do ambiente para o sistema operacional Linux. A maior diferença encontra-se no visual do programa "Hope for Windows" que é colorizado. As figuras 2.1, 2.2 e 2.3 mostram a aparência dos ambientes nos sistemas operacionais Windows e Linux (Fedora 33).



```
Hope Console
** Hope for Windows Version 1.2.2 (c) 2012 ***
**
**      Welcome to the HOPE interpreter      **
**                                          **
*****
Augusto at DESKTOP-UL8U3Q2 Windows_NT on  Mon Apr 05 06:34:48 2021.

help; for help.  http://www.hopemachine.co.uk

>: _
```

Figura 2.1 - Programa "Hope for Windows" no sistema operacional Windows.

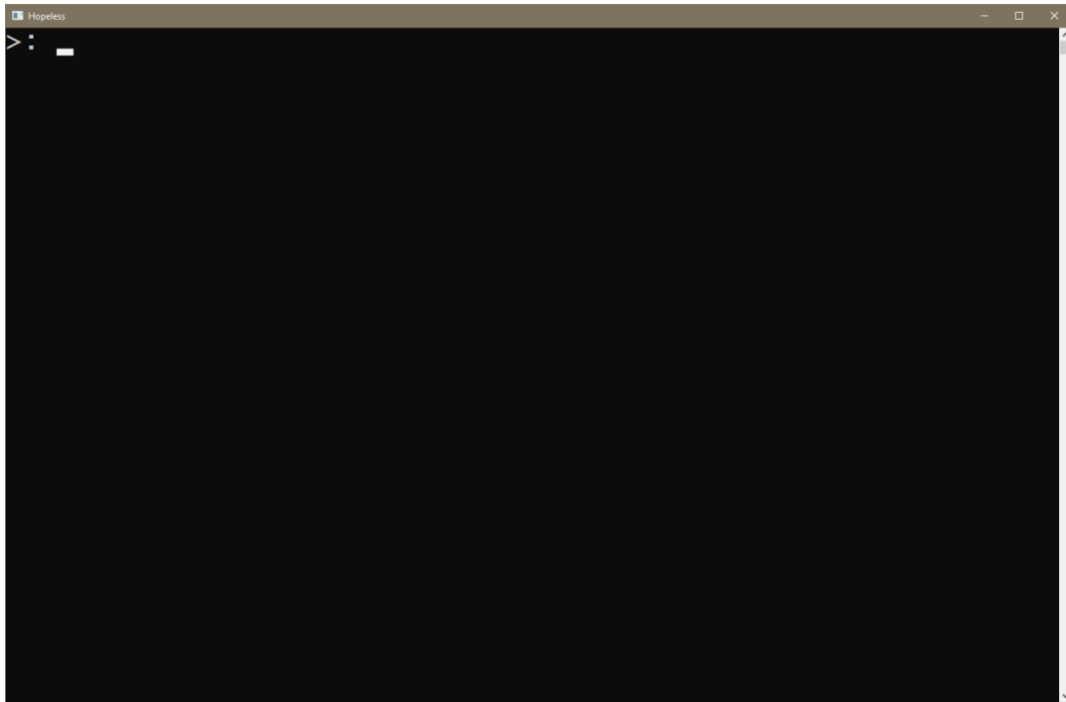


Figura 2.2 - Programa "Hopeless" no sistema operacional Windows.

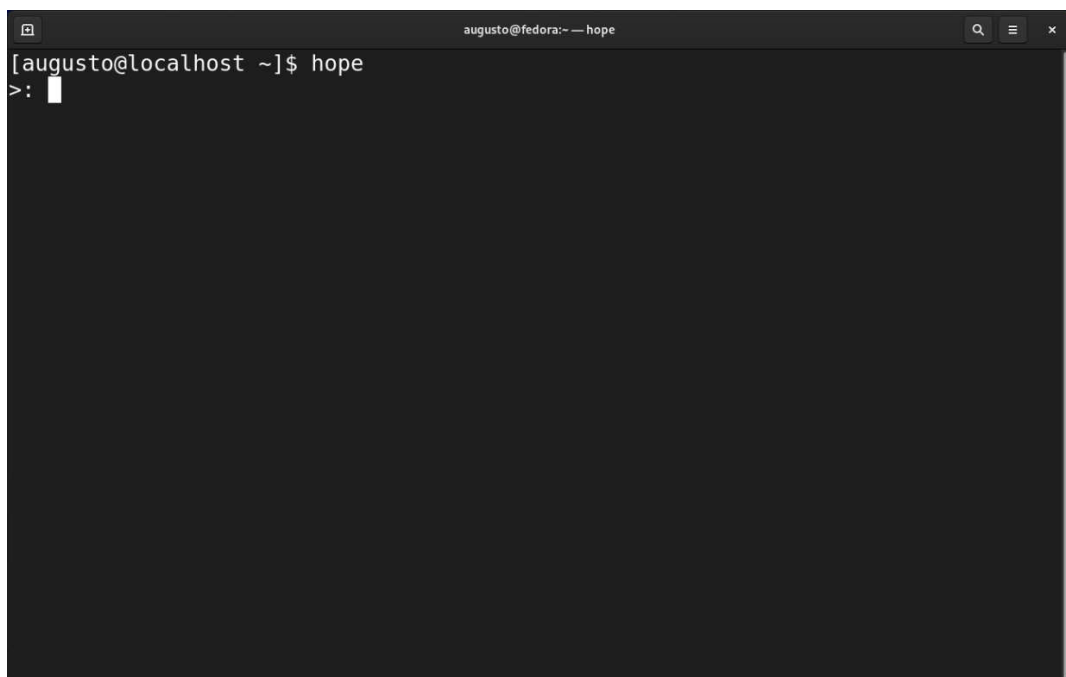


Figura 2.3 - Programa "Hope" no sistema operacional Linux (Fedora 33).

Para encerrar o ambiente basta executar no *prompt* ">:" o comando de saída **"exit;"** e acionar a tecla **"<Enter>"**. A propósito o *prompt* ">:" indica que o ambiente está pronto para uso.

Um cuidado a ser considerado, o indicativo de uso do comando **"help;"** apresentado na tela do programa "Hope for Windows" somente funciona nos programas "Hope for Windows" e "Hopeless" do sistema operacional Windows (Figura 2.4) não sendo aplicado a versão usada em Linux. No entanto a expressão **"<expression>"** e os comandos **"write"**, **"display"** e **"save"**, além de **"exit"** funcionam em qualquer ambiente de qualquer sistema operacional.



```

>: help;
Hope for Windows 1.2.2
<expression>;
write <expression>;
write <expression> to "file";
display;
save filename;
exit;
>: _

>: help;
Hopeless v0.5
<expression>;
write <expression>;
write <expression> to "file";
display;
save filename;
exit;
>: _

```

Figura 2.4 – Uso do comando "help;".

Os programas para o gerenciamento do ambiente *hope* aqui apresentados possuem como origem o mesmo código <https://github.com/dmbaturin/hope>. O código fonte utilizado para o ambiente *hope* do Linux é o mesmo para os ambientes destinados ao Windows, sendo "Hopeless" baseado no código do Linux e "Hope for Windows" baseado sobre o "Hopeless".

Após o acesso ao ambiente interativo *hope* é importante sempre fazer o carregamento do módulo adicional "**mylist.hop**" para maior comodidade operacional. Isto se faz necessário devido à ausência no ambiente *hope* de um módulo para essas ações que seja carregado automaticamente. Os detalhes sobre este módulo são encontrados no "Apêndice F - Módulo: *mylist.hop*". Para fazer uso do módulo "**mylist.hop**" basta executar no *prompt* do ambiente *hope* a instrução "**uses mylist;**", lembrando que o módulo "**mylist.hop**" deverá estar devidamente posicionado em seu sistema (veja o "Apêndice F").

2.2 - Ações interativas

Ao ser acessado o ambiente apresenta-se o *prompt* ">:" com a indicação do *cursor* piscando a sua frente. A partir deste momento pode ser informado a declaração e definição de funções, bem como fazer uso interativo do ambiente como se fosse uma calculadora. É possível realizar operações aritméticas e lógicas entre valores numéricos e listas de elementos numéricos ou alfanuméricos.

Ações aritméticas

As operações aritméticas são realizadas a partir do uso dos operadores: "+" (adição), "-" (subtração), "*" (multiplicação), "/" (divisão com quociente real), "**div**" (divisão com quociente inteiro) e "**mod**" (resto de divisão entre números inteiros).

Veja alguns exemplos desse tipo de operação. Os indicativos ">:" e ">>" não devem ser escritos no *prompt*. Informe tão somente o que estiver indicado após ">:". Os resultados das operações são apresentados, para maior legibilidade, em negrito.

```

>: 2+3;
>> 5 : num

>: 2-3;
>> -1 : num

>: 2*3;
>> 6 : num

>: 7/2;
>> 3.5 : num

>: 7 div 2;
>> 3 : num

```

```
>: 7 mod 2;
>> 1 : num
```

Além da definição de operações aritméticas básicas a linguagem fornece um conjunto interno de funções internas, prontas para o uso, destacando-se:

abs(x)	valor absoluto (sempre positivo) de "x";
acos(x)	arco cosseno de "x";
acosh(x)	arco cosseno hiperbólico de "x";
asin(x)	arco seno de "x";
asinh(x)	arco seno hiperbólico de "x";
atan(x)	arco tangente de "x";
atan2(x,y)	arco tangente do coeficiente dos argumentos "x" e "y";
atanh(x)	arco tangente hiperbólico de "x";
ceil(x)	arredondamentode "x" para o próximo inteiro acima;
cos(x)	cosseno de "x";
cosh(x)	cosseno hiperbólico de "x";
exp(x)	exponencial natural de "x";
floor(x)	arredondamentode "x" para o próximo inteiro abaixo;
hypot(x,y)	comprimento da hipotenusa de um triângulo retângulo de "x" e "y";
log(x)	logaritmo natural de "x" na base "e";
log10(x)	logaritmo natural de "x" na base 10;
pow(x, y)	potencia de "x" elevado a "y";
sin(x)	seno de "x";
sinh(x)	seno hiperbólico de "x";
sqrt(x)	raiz quadrada de "x";
tan(x)	tangente de "x";
tanh(x)	tangente hiperbólico de "x".

O uso dessas funções é bastante simples, bastando indicar o nome da função desejada e passar como argumento o valor necessário a sua ação para ver o resultado retornado.

```
>: pow(2,3);
>> 8 : num

>: abs(0-3);
>> 3 : num

>: log(2);
>> 0.693147180559945 : num

>: ceil(2.1);
>> 3 : num

>: floor(2.9);
>> 2 : num

>: sqrt(25);
>> 5 : num
```

As funções e os operadores aritméticos podem ser operacionalizados de forma combinada.

```
>: pow(2,3) + ceil(2.1) - 1;
>> 10 : num
```

A precedência das operações aritméticas na linguagem Hope segue o mesmo padrão estabelecido na Ciência Matemática.

```
>: 2*pow(10,3);  
>> 2000 : num  
  
>: 2+3*4;  
>> 14 : num
```

A relação de associabilidade da linguagem Hope é mesma estabelecida pela Ciência Matemática.

```
>: pow(2,pow(3,4)); !!! 2^3^4  
>> 2.41785163922926e+024 : num  
  
>: pow(2,(pow(3,4))); !!! 2^(3^4)  
>> 2.41785163922926e+024 : num  
  
>: 2-3-4;  
>> -5 : num  
  
>: (2-3)-4;  
>> -5 : num
```

Em relação ao uso de listas os cálculos realizados sobre essas estruturas podem gerar respostas peculiares, como: seleção de elemento, eliminação de elemento, obtenção do tamanho da lista, obtenção da soma dos elementos de uma lista, obtenção do produto dos elementos de uma lista, concatenação de duas listas, inversão dos elementos de uma lista.

Selecionar o primeiro elemento de uma lista não vazia.

```
>: head [1,2,3,4,5];  
>> 1 : num  
  
>: head ['h','o','p','e'];  
>> 'h' : char  
  
>: head "hope";  
>> 'h' : char
```

Eliminar o primeiro elemento de uma lista não vazia.

```
>: tail [1,2,3,4,5];  
>> [2, 3, 4, 5] : list num  
  
>: tail ['h','o','p','e'];  
>> "ope" : list char  
  
>: tail "hope";  
>> "ope" : list char
```

Selecionar o " n "-ésimo elemento de uma lista (iniciando em 0):

```
>: index (2, [1,2,3,4,5]);  
>> 3 : num
```

Selecionar os " n " primeiros elementos de uma lista:

```
>: take (3, [1,2,3,4,5]);  
>> [1, 2, 3] : list num
```

Eliminar os "n" primeiros elementos de uma lista:

```
>: drop (3, [1,2,3,4,5]);
>> [4, 5] : list num
```

Calcular o tamanho de uma lista:

```
>: length [1,2,3,4,5];
>> 5 : num
```

Calcular a soma de uma lista de números:

```
>: sum [1,2,3,4,5];
>> 15 : num
```

Calcular o produto de uma lista de números:

```
>: product [1,2,3,4,5];
>> 120 : num
```

Concatenar duas listas:

```
>: [1,2,3] <> [4,5];
>> [1, 2, 3, 4, 5] : list num
```

Inverter uma lista:

```
>: reverse [1,2,3,4,5];
>> [5, 4, 3, 2, 1] : list num
```

Há ainda certas operações aritméticas que podem resultar em erros quando são submetidas ao ambiente *hope*. É importante conhecê-las para evitá-las.

```
>: 1 div 0;
>> run-time error - attempt to divide by zero

>: head [];
>> user error - Empty list!

>: tail [];
>> user error - Empty list!

>: index (5, [2,3]);
>> user error - Index out of range!
```

2.3 - Aplicação de funções

Funções são a base da programação funcional e neste paradigma o conceito aplicado sobre a programação é, basicamente, o mesmo aplicado na Ciência Matemática. No entanto, existem pontos a serem considerados.

A notação de uma função no contexto matemático é representada por parênteses e a multiplicação usa justaposição ou espaços. Por exemplo, a função "**f(a,b) + cd**" representa a soma do valor de "f" aplicado a "a" e "b" somado ao produto de "c" por "d".

A notação de uma função no contexto computacional, precisamente na linguagem Hope, é representada por parênteses quando houver mais de um argumento, espaços e a multiplicação é indicada com o uso do símbolo "*". Por exemplo, a função " $f(a,b) + c * d$ " representa a soma do valor de " f " aplicado a " a " e " b " somado ao produto ("*") de " c " por " d ".

Assim como os operadores possuem prioridade na ação de seus cálculos as funções em Hope também possuem essa característica. A aplicação de funções tem maior prioridade que quaisquer outros operadores. Por exemplo, a expressão " $f a + b$ " representa a expressão matemática " $f(a) + b$ ".

Exemplos de expressões Hope e expressões matemáticas.

EXPRESSÕES	
MATEMÁTICA	HOPE
$f(x)$	$f\ x$
$f(x,y)$	$f\ (x,\ y)$
$f(g(x))$	$f\ (g\ x)$
$f\ (x,\ g(y))$	$f\ (x,\ (g\ y))$
$f\ (x)g(y)$	$f\ x * g\ y$

2.4 - Scripts Hope

Em Hope os usuários podem escrever suas próprias funções. As funções do usuário são definidas na forma de scripts, que são codificações de textos compostos por uma sucessão de funções definidas pelo usuário. Os scripts Hope podem ser armazenados em arquivos de módulos identificados com a extensão ".hop". Assim sendo, há duas formas de se fazer a codificação de funções em Hope: a primeira diretamente no ambiente (forma rápida) e a segunda fora do ambiente (forma adequada). A primeira forma apesar de prática é limitada e muito restritiva, sendo a segunda mais conveniente.

Se você estiver usando o ambiente *hope* no sistema operacional Linux informe no *prompt* do ambiente o comando "**edit**" seguido de um nome de arquivo de módulo sem a extensão ".hop" com ponto-e-vírgula. Por exemplo, para criar um arquivo chamado "**teste**" use a instrução "**edit teste;**". Na sequência é aberto automaticamente o editor de textos padrão do sistema. No caso do *Fedora Linux* abre-se o editor de textos "**nano**" e no *Ubuntu Linux* e *openSuse Leap Linux* abre-se o editor de texto "**vim**".

A partir do editor aberto escreva o código ou os códigos das funções necessárias, grave o conteúdo editado e encerre o editor. Assim que o editor é finalizado o controle é devolvido ao ambiente *hope*. Após a definição inicial do arquivo de módulo basta executar, por exemplo, "**uses teste;**" para colocar a função ou funções definidas em uso. A partir daí é só solicitar a edição do arquivo, preceder mudanças, gravar alterações e ao voltar ao ambiente *hope* e usar automaticamente o que foi editado. O uso de "**uses teste;**" é necessário apenas na primeira operação da instancia do ambiente. Esse mecanismo torna o uso do ambiente e editor de texto no Linux extremamente dinâmico.

O editor de texto "**nano**" é simples e intuitivo. Após escrever o código da função ou funções desejadas é pertinente realizar a gravação de seu conteúdo. Para tanto, acione as teclas de atalho "**<Ctrl> + <o>**" e confirme com "**<Enter>**". Para sair do editor e voltar ao ambiente *hope* acione as teclas de atalho "**<Ctrl> + <o>**".

O editor de textos "**vim**" é simples mas não muito intuitivo. Assim, dentro do editor para informar comandos de operação use inicialmente o símbolo dois pontos ":" seguido de um código de operação. Veja a seguir alguns códigos básicos:

```
:i      - insere texto antes da posição atual do cursor;
:a      - insere texto depois da posição atual do cursor;
:o      - adiciona linha em branco abaixo da linha onde se encontra o cursor;
:O      - adiciona linha em branco acima da linha onde se encontra o cursor;
:q      - encerra execução do editor de textos;
<Esc>   - muda para o modo visualização.
```

No editor de texto "**vim**" acione o comando **":a"** entre o código ou códigos das funções desejadas. Acione a tecla "**<Esc>**" para mudar o modo de visualização e voltar ao modo de comando e informe o comando **":w"** para gravar o conteúdo editado. Na sequência para voltar ao ambiente *hope* informe o comando **":q"**.

No sistema operacional Windows infelizmente a dinâmica é outra e não é agradável. Para seus usuários o que pode ser feito, de forma paliativa, é abrir o editor de textos padrão do sistema (normalmente o **Bloco de Notas**) ou outro de sua preferência, escrever na primeira linha a instrução **"uses mylist;"** e em seguida codificar as funções. Gravar o arquivo de módulo com a extensão **".hop"** no mesmo local onde o programa **"hope.exe"** ou **"hopeless.exe"** se encontram, chamando este arquivo no ambiente *hope* com o comando **"uses"**. Caso necessite atualizar o arquivo **".hop"** este não poderá ser recarregado na memória após aberto e não poderá ser regravado. Neste caso, deve-se encerrar o ambiente, abri-lo novamente e fazer nova chamada do arquivo de módulo em uma nova instância do ambiente na memória.

A partir dessa visão inicial prepare-se para escrever seu primeiro arquivo de módulo Hope. Abra o ambiente *hope* em seu sistema operacional preferido e execute inicialmente no *prompt* do ambiente a instrução **"uses mylist;"**.

Se estiver em uso o sistema Linux execute no *prompt* do ambiente a instrução **"edit exemplo;"** para que o editor seja aberto. Se for a distribuição Fedora não há necessidade de fazer nada, mas se for o Ubuntu ou openSUSE Leap acione o comando **":a"**. Para o sistema Windows abra o **"Bloco de Notas"**.

Na primeira linha do arquivo de módulo **"exemplo.hop"**, tanto no Windows como no Linux coloque a instrução **"uses mylist"** e acione **"<Enter>"** e informe as definições **"dobro"** e **"quadruplo"** seguintes:

```
dec dobro : num -> num;
--- dobro x <= x + x;

dec quadruplo : num -> num;
--- quadruplo x <= dobro (dobro x);
```

Grave as funções definidas como comentado e a partir do ambiente *hope* execute a instrução **"uses exemplo;"**. Depois, efetue como teste as seguintes execuções:

```
>: quadruplo 10;
>> 40 : num

>: take (dobro 2, [1,2,3,4,5,6]);
>> [1, 2, 3, 4] : list num
```

No sistema operacional Linux volte ao modo de edição **"edit exemplo;"** e no Windows com o **"Bloco de Notas"** aberto ponha o cursor na última linha de texto e acrescente as seguintes definições:

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

dec media : list num -> num;
--- media ns <= sum ns div length ns;
```

Grave o conteúdo atual e volte ao ambiente *hope*. Se estiver em uso o Linux nada precisará ser feito, mas se estiver em uso o Windows feche o ambiente *hope* aberto e chame-o novamente, executando em seguida a instrução "**uses mylist, exemplo;**" e efetue como teste as seguintes execuções:

```
>: fatorial (dobro 2);
>> 24 : num

>: dobro (media [1,5,3]);
>> 6 : num
```

2.5 - Atribuição de nomes e comentários

A definição de nomes para as funções segue algumas regras. Hope é muito democrática nesse sentido, mas o mesmo não se pode dizer de outras linguagens. Desta forma, é sempre melhor assumir regras mais rígidas para evitar eventuais problemas. Assim sendo, nunca inicie o nome de uma função com caractere maiúsculo, mas se necessário use-o a partir do segundo caractere do nome, por exemplo "**sProduto**", "**somaQuadrado**", "**somaQuadradoDoProduto**", etc.

As palavras-chave reservadas da linguagem não podem ser usadas como nomes de funções. Nunca use as palavras:

```
---, \, abstype, char, data, dec, display, edit, else, end, error, exit, help, id, if,
in, infix, infixr, infixl, lambda, let, letrec, list, module, mu, nonop, num, private,
pubconst, pubfun, pubtype, read, save, then, trual, tuple, type, typevar, use, uses,
where, whererec, write.
```

Acostume-se a escrever os nomes dos argumentos de listas usando "**s**" como sufixo. Por exemplo:

```
ns      - representa uma lista de números;
xs      - representa uma lista de elementos;
css     - representa uma lista de lista de caracteres.
```

Em Hope o layout do texto do programa (o recuo, ou seja, a endentação) delimita definições de continuidade de uma instrução quando esta é muito extensa. As endentações estabelecem as partes do código com deslocamento da margem esquerda menor ou igual ao início da definição de sua linha anterior.

Exemplo:

```
--- teste (x, n) <= if x = n then a else b
                      where a == true
                      where d == false;
```

Conselhos:

- Comece as definições das funções na primeira coluna;
- Use espaços em branco para determinar o recuo nas definições (não use "<Tab>").

Os arquivos de módulos podem conter linhas de comentários. Os comentários são declarados após o uso do símbolo "!". Pode-se fazer uso da quantidade de símbolos "!" necessárias a definições de certo comentário. Seguem duas sugestões:

```
!! linha de comentário para o estabelecimento de
!! cabeçalhos de texto.
```

```
! Linha de comentário simples
```

Outras formas de uso dos comentários poderão ser definidas para melhor organizar a documentação interna do código de um arquivo de módulo.

OBS.:

Ao final de cada capítulo é pertinente efetuar a gravação dos scripts utilizados, por exemplo com o nome "**cap_N_prog.hop**", onde "**N**" é o número do capítulo em foco.

Evite usar a instrução "**save nome;**", pois em alguns casos, por razões desconhecidas, o arquivo pode ser corrompido, o que impedirá seu carregamento posterior com a instrução "**uses nome;**". Esses efeitos foram detectados junto aos capítulos 6 (uso da função definida "**| |**") e 8 (no uso de todas as instruções com o comando "**data**").

Para gravar o conteúdo de cada capítulo utilize a cláusula "**edit nome;**", exceto no Windows que deverá ser feito precariamente fora do ambiente e diretamente em um editor de textos de sua preferência como comentado.

CAPÍTULO 3 - Tipos de dados

Um tipo de dado, na programação de computadores, é uma forma de se dizer ao computador como se deseja usar sua memória, além de ser uma maneira de estabelecer coleções de valores relacionados. Nem todas as linguagens de programação operam com essa característica: algumas gerenciam os dados na memória automaticamente e outras não. As linguagens que não gerenciam os dados e seus tipos em memória, ao contrário do que se pode pensar, dão maior liberdade ao desenvolvimento de programas, uma vez que permitem a você ajustar, de forma segura, exatamente o que se deseja e se necessita (BIRD & WADLER, 1998; BAILEY, 1985 & 1990).

3.1 - Tipos básicos

A linguagem Hope, assim como outras linguagens funcionais, oferece um conjunto de tipos básicos de dados que atendem de forma geral as necessidades mais específicas encontradas em diversas situações computacionais. Os tipos de dados básicos fornecidos pela linguagem são:

- **num** - tipo de dado usado para representar valores numéricos inteiros ou reais com ponto flutuante, como '1', '0', '1.5', '99' e '-4';
- **char** - tipo de dado usado para representar valores alfabéticos e/ou alfanuméricos delimitados entre aspas simples, como 'x', 'A', '3' e '\$';
- **truval** - tipo de dado usado para representar valores lógicos, como 'true' e 'false'.

A partir desse conjunto mínimo de tipos é possível representar os valores essenciais a programação de computadores para o estabelecimento, em especial, de operações aritmética e lógicas.

3.2 - Tipos compostos

Os tipos de dados básicos podem atender a diversas necessidades no uso de dados, mas na programação funcional são insuficientes, pois este estilo de programação necessita operar dados na forma de conjuntos como descritos no estudo da "teoria de conjuntos" na Ciência Matemática. Neste sentido, Hope oferece os dados compostos:

- **tuple** - tipo de dado usado para representar conjuntos de dados heterogêneos delimitados entre parênteses e separados por vírgula, como **(1,2)** que retorna $(1, 2) : num \# num$ e **(1, 'A', true)** que retorna $(1, 'A', true) : num \# char \# truval$;
- **list** - tipo de dado usado para representar conjuntos de dados homogêneos delimitados entre colchetes e separados por vírgula, como **[1,2,3]** que retorna $[1, 2, 3] : list\ num, ['a', 'b', 'c']$ que retorna $"abc" : list\ char$ e **[true, false]** que retorna $[true, false] : list\ truval$.

Há linguagens funcionais que apresentam outros tipos de dados compostos, mas no geral "list" e "tuple" atendem bem as necessidades de manipulação de dados.

Hope não opera diretamente com dados do tipo *cadeia de caracteres*, conhecidos como "string", mas os aceita de forma indireta a partir da definição de listas de caracteres fundamentada sobre o tipo "list char" para uma entrada ['a', 'b', 'd'], ou o tipo "list (list char)" para uma entrada ["livro", "hope"].

3.3 - Tipos funções

Uma função dentro do contexto da programação funcional é uma aplicação de valores de certo tipo que pode, inclusive, retornar valores de outro tipo. O identificador $T_1 \rightarrow T_2$ representa o tipo de uma função que aplica valores do tipo T_1 em valores do tipo T_2 . Veja alguns exemplos de funções que recebem um tipo de argumento e devolvem um tipo igual ou diferente ao tipo indicado no argumento de entrada.

```
>: not;
>> not : truval -> truval

>: isDigit;
>> isDigit : char -> truval

>: odd;
>> odd : num -> truval

>: abs;
>> abs : num -> num
```

Uma função pode operar com mais de um argumento, se assim for estabelecido. Neste caso, o conjunto de argumentos deve ser indicado entre parênteses, na forma de tupla. Veja como exemplo o código da função "**soma2**" que opera o uso de múltiplos argumentos (mais de um argumento) a partir da recepção de dois argumentos "**x**" e "**y**", retornando o resultado da soma entre os argumentos "**x**" e "**y**". Por exemplo, **soma2 (2,3)** é igual a "5".

```
dec soma2 : num # num -> num;
--- soma2 (x, y) <= x + y;
```

Veja na primeira linha da função o estabelecimento de sua assinatura, a definição do protótipo para que o computador reserve os espaços de memória necessários a operação. Veja a definição dos argumentos de entrada "**num # num**" após o símbolo ":". Note que o formato "**num # num**" é o indicativo do uso de uma tupla de dois componentes numéricos.

Na segunda linha a indicação dos argumentos "**x**" e "**y**" deve obrigatoriamente ser declarada dentro de parênteses "**(x, y)**" antes do símbolo de asserção "<=" devido a indicação "**num # num**" na primeira linha da função.

Além de operar com mais de um argumento, uma função pode operar mais de um valor. Considere a função "**deZeroA**" que apresente os valores de "0" até um limite fornecido. Por exemplo, **deZeroA 5** retornará a lista "**[0,1,2,3,4,5]**".

```
dec deZeroA : num -> list num;
--- deZeroA n <= (0..n);
```

Veja que a função "**deZeroA**" tem em sua primeira linha a definição da recepção de um argumento do tipo "**num**" e a devolução de uma resposta do tipo "**list num**" que permite voltar como resposta mais de um valor, neste caso, do tipo "**list**".

Existem linguagens funcionais em que a definição da linha de protótipo é opcional, não sendo este o caso de Hope. Mesmo nas linguagens em que essa definição é opcional sugere-se sempre fazê-la por ser uma atitude além de conveniente, extremamente educada.

3.4 - Inferência de tipos

A inferência de tipos ocorre quando a linguagem de programação consegue automaticamente detectar o tipo de dado informado ou o tipo de retorno de uma expressão estabelecida. Essa é uma característica, comum, encontrada em linguagens funcionais. Uma forma de detectar tipos de dados em Hope é fazer uso da função "id" que identifica e retorna o tipo do dado informado. Observe algumas situações:

```
>: id 2;
>> 2 : num

>: id 1.5;
>> 1.5 : num

>: id true;
>> true : truval

>: id 'a';
>> 'a' : char

>: id "abelha";
>> "abelha" : list char

>: id [1,2,3];
>> [1, 2, 3] : list num

>: id ["livro","hope"];
>> ["livro", "hope"] : list (list char)

>: id (1,2,3);
>> (1, 2, 3) : num # num # num

>: id (1,'a',false);
>> (1, 'a', false) : num # char # truval

>: "texto" = "texto";
>> true : truval

>: "texto" = "testo";
>> false : truval

>: id not;
>> not : truval -> truval

>: id not (not false);
>> false : truval

>: dobro (media [1,5,3]);
>> 6 : num
```

As linguagens que inferem os tipos de dados usados são ferramentas que operam com os chamados **tipos seguros** e Hope é uma linguagem de tipos seguros. Normalmente, nas linguagens que possuem tipos seguros os erros de tipo não ocorrem durante a avaliação. Mas, a inferência de tipo não elimina todos os erros durante a avaliação. Por exemplo,

```
>: id 1 div 0;
run-time error - attempt to divide by zero
```

3.5 - Parcialização

A parcialização no contexto da programação funcional se refere ao termo "*currificação*" oriundo do inglês em "*currying*" que permite estabelecer a característica de uma função com mais de um argumento poder ser interpretada como funções que tomam um argumento e devolvem outra função com um argumento a menos. Veja um exemplo de parcialização.

```
dec soma3 : num -> num -> num;
--- soma3 x y <= x + y;
```

Veja que na primeira linha da função está definida a recepção de apenas um valor que devolve outro valor que é usado para devolver um terceiro valor "**num -> num -> num**". Note que não está em uso o símbolo "#" e, assim sendo, seus elementos não são operacionalizados entre parênteses.

A função "**soma3**" toma o valor numérico "**x**", retorna o valor "**x**" pegando em seguida o valor "**y**" e aplicando a operação entre "**x**" e "**y**" a fim de obter o resultado da soma. Por exemplo,

```
>: soma3;
>> soma3 : num -> num -> num

>: soma3 2;
>> soma3 2 : num -> num

>: soma3 2 5;
>> 7 : num
```

Observe agora um exemplo de parcialização com o uso de três argumento:

```
dec mult : num -> num -> num -> num;
--- mult x y z <= x * y * z;
```

A função "**mult**" toma o valor numérico "**x**", retorna o valor "**x**" pegando em seguida o valor "**y**" e aplicando a operação entre "**x**" e "**y**" a fim de obter o resultado parcial da multiplicação. Após essa etapa intermediária a função pega o resultado parcial de "**x**" e "**y**" e aplica este junto ao valor "**z**" de modo a obter o resultado da multiplicação de "**x**", "**y**" e "**z**".

```
>: mult;
>> mult : num -> num -> num -> num

>: mult 2;
>> mult 2 : num -> num -> num

>: mult 2 3;
>> mult 2 3 : num -> num

>: mult 2 3 7;
>> 42 : num
```

As funções que operam seus argumentos de um em um, como as funções "**soma3**" e "**mult**", são funções do tipo *curried* (currificadas). Este estilo de ação pode ser aplicado de forma parcial. Por exemplo:

```
>: (soma3 2) 3;
>> 5 : num
```


As ações de currficação podem ser utilizadas na definição de funções parciais. Por exemplo,

```
dec suc : num -> num;
--- suc <= soma3 1;
```

Em que a função "**suc**" retoma o sucessor de um valor numérico informado. Veja que a função recebe um argumento e aplica esse argumento sobre a função "**soma linha**" que ao pegar o valor informado efetua a soma deste com o valor "**1**" estabelecido.

```
>: suc 3;
>> 4 : num
```

Um detalhe importante a ser considerado é que as funções em Hope com múltiplos argumentos são, normalmente, definidas na forma de tuplas a partir do uso do símbolo "#", a menos que seja explicitamente declarado que os argumentos devem ser currficados.

3.6 - Polimorfismo

Um determinado tipo de dado é polimórfico quando este pode assumir qualquer forma de tipo suportado pela linguagem. Desta forma, não é necessário, basicamente, se preocupar se o dado será "**num**" ou "**char**", excetuando-se, em parte, o tipo "**truval**" mais usado como tipo de retorno de função. No entanto, de uma forma mais rara o tipo "**truval**" poderá também ser usado como argumento de entrada para alguma função.

Uma função é considerada polimórfica se seu retorno é do tipo polimórfico. Por exemplo, a função "**length**" do módulo "**mylist**" é polimórfica:

```
>: length;
>> length : list alpha -> num
```

Veja que a função "**length**" possui o argumento de tipo "**list**" qualificado como "**alpha**". Isto significa que para qualquer dado "**alpha**" a função retorna a quantidade de elementos da lista informada.

O qualificador "**alpha**" da função "**length**" caracteriza-se por ser uma *variável de tipo* que deve ser definida com seus nomes sob o uso de caracteres minúsculos, mesmo que a linguagem não possua essa restrição. Veja alguns exemplos no uso da função "**length**":

```
>: length [1, 4, 7, 1];
>> 4 : num

>: length ["Jan", "Fev", "Mar"];
>> 3 : num

>: length [reverse, tail];
>> 2 : num
```

Exemplos de funções polimórficas definidas no arquivo de módulo "**mylist**":

```
>: fst;
>> fst : alpha # beta -> alpha

>: fst (1, 'x');
>> 1 : num
```

```

>: fst (true,"Hoje");
>> true : truval

>: snd;
>> snd : alpha # beta -> beta

>: snd (1,'x');
>> 'x' : char

>: snd (true,"Hoje");
>> "Hoje" : list char

>: head;
>> head : list alpha -> alpha

>: head [2,1,4];
>> 2 : num

>: head ['b','c'];
>> 'b' : char

>: take;
>> take : num # list alpha -> list alpha

>: take (3, [3,5,7,9,4]);
>> [3, 5, 7] : list num

>: take (2, ['l','o','l','a']);
>> "lo" : list char

>: take (2, "lola");
>> "lo" : list char

>: zip;
>> zip : list alpha # list beta -> list (alpha # beta)

>: zip ([3,5], "lo");
>> [(3, 'l'), (5, 'o')] : list (num # char)

>: id;
>> id : alpha -> alpha

>: id 3;
>> 3 : num

>: id 'x';
>> 'x' : char

```

CAPÍTULO 4 - Definição de funções

O principal motivo pelo qual o *paradigma declarativo funcional* é assim conhecido é o fato de toda a engenharia da programação estar fundamentada no uso de *funções*. Neste sentido, essas funções são definidas a partir de algumas estratégias de concepção a serem percebidas (BIRD & WADLER, 1998; BAILEY, 1985 & 1990).

4.1 - Por composição

A definição por composição é a forma mais simples de se estabelecer o desenvolvimento de certa função. Veja os exemplos de definição das funções **"isDigit"**, **"even"** e **"splitAt"** do arquivo de módulo **"mylist"**.

Decidir se um caractere é um dígito:

```
dec isDigit : char -> truval;
--- isDigit c <= '0' =< c and c <= '9';
```

Decidir se um número (inteiro) é par:

```
dec even : num -> truval;
--- even n <= n mod 2 = 0;
```

Dividir uma lista em seu "*n*"-ésimo elemento

```
dec splitAt : num # list alpha -> list alpha # list alpha;
--- splitAt (n, xs) <= (take (n, xs), drop (n, xs));
```

4.2 - Por condição

A definição por condição é a forma pela qual uma função retorna um de uma série de valores a partir da ocorrência de certa condição. Veja os exemplos de definição da função **"abs"** similar a função homônima padrão interna do arquivo de módulo **"Standard"** e a função **"signum"** do arquivo de módulo **"mylist"**.

Calcular o valor absoluto (com condição):

```
dec abs : num -> num;
--- abs n <= if n >= 0 then n else 0-n;
```

Calcular o sinal de um número (com condições aninhadas):

```
dec signum : num -> num;
--- signum n <= if n < 0 then 0-1 else
                if n = 0 then 0 else 1;
```

4.3 - Por correspondência de padrões

A correspondência de padrões (*pattern matching*) é uma forma de uso de decisões indiretas em que certo valor que esteja sob determinada condição pode ser executado de modo a obter informações a partir deste valor. Neste contexto podem ser realizadas as correspondências de padrões sobre constantes, variáveis, listas, tuplas e valores.

A partir da definição de constantes como padrões veja os exemplos de definição das funções **"not"** e **"and"** do arquivo de módulo **"Standard"**.

Para calcular a negação:

```
dec not : truval -> truval;
--- not true <= false;
--- not false <= true;
```

Para calcular a conjunção (com valores):

```
dec      and : truval # truval -> truval;
--- false and p <= false;
--- true  and p <= p;
```

Não se preocupe com o uso da cláusula **"infix"** para as definições dos dois próximos operadores: **"and1"** e **"and2"**. Em momento mais oportuno isso será explicado. Atente agora, apenas aos exemplos das funções **"and1"** e **"and2"** apresentados.

Para calcular a conjunção (com variáveis anônimas):

```
infix    and1 : 2;
dec      and1 : truval # truval -> truval;
--- true and1 true <= true;
--- _    and1 _    <= false;
```

Para calcular a conjunção (com variáveis):

```
infix    and2 : 2;
dec      and2 : truval # truval -> truval;
--- true and2 x <= x;
--- _    and2 _ <= false;
```

A partir da definição de tuplas como padrões veja os exemplos das funções **"fst"** e **"snd"** do arquivo de módulo **"mylist"**.

Calcular o primeiro elemento de um par de valores:

```
dec fst : (alpha # beta) -> alpha;
--- fst (x,_) <= x;
```

Calcular o segundo elemento de um par de valores:

```
dec snd : (alpha # beta) -> beta;
--- snd (_,y) <= y;
```

A partir da definição de lista como padrões veja os exemplos de definição das funções definidas "**teste1**" e "**teste2**", além das funções "**null**", "**head**", "**tail**" e "**pred**" do arquivo de módulo "**mylist**". A função "**teste1 xs**" testa se a lista "**xs**" é uma lista de três caracteres em que o primeiro seja iniciado com o caractere "**a**":

```
dec teste1 : list char -> truval;
--- teste1 ['a', _, _] <= true;
--- teste1 _          <= false;
```

Construção de listas com o operador "**cons**" representado pelo símbolo "::":

```
| [1,2,3] = 1::[2,3] = 1::(2::[3]) = 1::(2::(3::[]));
```

A função "**teste2 xs**" testa se a lista "**xs**" é uma lista de caracteres iniciada com o caractere "**a**":

```
dec teste2 : list char -> truval;
--- teste2 ('a'::_) <= true;
--- teste2 _      <= false;
```

Decidir se uma lista é vazia:

```
dec null : list alpha -> truval;
--- null []          <= true;
--- null (_::_)      <= false;
```

Apresentar o primeiro elemento de uma lista:

```
dec head : list alpha -> alpha;
--- head []          <= error ("Empty list!");
--- head (x :: _)    <= x;
```

Apresentar a cauda de uma lista (todos os valores, exceto a cabeça):

```
dec tail : list alpha -> list alpha;
--- tail []          <= error ("Empty list!");
--- tail (_ :: xs)    <= xs;
```

A partir da definição de valores como padrões veja os exemplos de definição da função "**pred**" do arquivo de módulo "**mylist**".

Calcular o predecessor de um número:

```
dec pred : num -> num;
--- pred 0      <= 0;
--- pred (n+1) <= n;
```

Observações sobre o uso do padrão "**n+k**" como "**n+1**":

- O estilo "**n+k**" é apenas igualado a números maiores ou iguais a "**n**";
- Há que escrevê-lo entre parênteses.

4.4 - Expressões lambda

As funções podem ser definidas sem nomeá-las usando expressões lambda. É importante considerar que funções dessa categoria não ficam armazenadas na memória e devem ser usadas imediatamente a sua definição. No entanto, funções lambda podem ser associadas a variáveis e assim mantidas na memória.

Exemplo de avaliação de uma expressão lambda:

```
>: (\x => x + x) 3;
>> 6 : num
```

Definição da função "**soma4**" sem o uso de expressão lambda:

```
dec soma4 : num # num -> num;
--- soma4 (x, y) <= x + y;
```

Uso de expressões lambda para destacar a parcialização (currificação) sobre, por exemplo, a definição da função "**soma5 x y**" com o uso de expressão lambda associada a variável "**soma5**":

```
dec soma5 : num -> num -> num;
--- soma5 <= \x => \y => x + y;
```

É importante ressaltar que em se tratando de programação funcional todos os elementos operacionalizados no ambiente de um operador aritmético, a um valor, a uma função, a uma constante, a uma variável são independentemente de qualquer coisa: *funções*.

Uso de expressões lambda em funções como resultados em que "**const x y**" é "**x**".

Definição da função "**const**" do arquivo de módulo "**mylist**" sem o uso de expressão lambda:

```
dec const : alpha -> beta -> alpha;
--- const x _ <= x;
```

Definição da função "**const**" com uso de expressão lambda:

```
dec const' : alpha -> beta -> alpha;
--- const' x <= \_ => x;
```

Uso de expressões lambda em funções com apenas um argumento como na função "**impares**" que apresenta uma lista de números ímpares dos "**n**" primeiros números ímpares.

Definição da função "**impares**" sem o uso de expressão lambda:

```
dec calcImpar : num -> num;
--- calcImpar x <= 2*x+1;

dec impares : num -> list num;
--- impares n <= map ((0..n-1), calcImpar);
```

Definição da função "**impares**" com uso de expressão lambda:

```
dec impares' : num -> list num;  
--- impares' n <= map ((0..n-1), \x => 2*x+1);
```

ANOTAÇÕES

CAPÍTULO 5 - Definições de listas por compreensão

É sabido que os conjuntos podem ser representados de três maneiras básicas: por enumeração (descrição), por compreensão e diagramas. Em Hope é possível usar apenas as formas por enumeração e compreensão representados por listas ou tuplas. A enumeração caracteriza-se por distribuir os elementos separados por vírgula e a compreensão caracteriza-se por indicar uma propriedade comum e exclusiva de seus elementos (BIRD & WADLER, 1998; BAILEY, 1985 & 1990; DIAS, 2020).

5.1 - Geradores

A definição de um conjunto por compreensão em Matemática segue o modelo:

$$\{x^2 \mid x \in \{2, 3, 4, 5\}\} = \{4, 9, 16, 25\}$$

Diferentemente de outras linguagens de programação funcional Hope não possui internamente recursos para definir compreensões. Neste sentido, usa-se alguma função especializada para fazer a simulação dessa operação, destacando-se, neste sentido, a função **"map"** pertencente ao arquivo de módulo **"mylist"**:

```
>: map (\x => pow (x,2), (2..5));
>> [4, 9, 16, 25] : list num
```

Para o contexto apresentado a operação **"(2..5)"** é denominada como sendo o recurso gerador da operação desejada.

Devido a limitação da linguagem para realizar operações de compreensões com duas ou mais listas torna-se necessário ter que escrever um conjunto de funções que deem suporte essas operações. Por se tratar de dinâmica mais avançado e fugir do escopo proposto neste trabalho este tema não é aqui tratado.

5.2 - Guardas

As listas geradas por ações de compreensões podem ter a definição de **guardas** com o objetivo de restringir os valores operacionalizados.

Exemplo de guarda a partir da função **"filter"** do arquivo de módulo **"mylist"**:

```
>: filter (\x => even (x), (1..10));
>> [2, 4, 6, 8, 10] : list num

>: filter (even, (1..10));
>> [2, 4, 6, 8, 10] : list num
```

Neste exemplo a guarda da operação é a função **"even"** do arquivo de módulo **"mylist"**.

A função **"fatores"** é a lista dos fatores do número **"n"**. Por exemplo,

```
>: fatores 30;
>> [1, 2, 3, 5, 6, 10, 15, 30] : list num
```

Para obter o resultado considere o código da função **"fatores"** a seguir:

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

A função "**primo**" verifica se "**n**" é primo. Por exemplo,

```
>: primo 30;
>> false : truval

>: primo 31;
>> true : truval
```

Para obter o resultado considere o código da função "**primo**" a seguir:

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

A função "**primos**" apresenta uma lista de valores primos até o limite estabelecido em "**n**". Por exemplo,

```
>: primos 31;
>> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31] : list num
```

Para obter o resultado considere o código da função "**primos**" a seguir:

```
dec primos : num -> list num;
--- primos n <= filter (primo, (2..n));
```

5.3 - A função "zip"

A função "**zip (xs, ys)**" é a lista obtida combinando-se os elementos das listas "**xs**" e "**ys**". Por exemplo,

```
>: zip (['a','b','c'], [2,5,4,7]);
>> [('a', 2), ('b', 5), ('c', 4)] : list (char # num)
```

A função "**adjacentes (xs)**" é a lista de pares de elementos adjacentes na lista "**xs**". Por exemplo,

```
| adjacentes [2,5,3,7] = [(2, 5), (5, 3), (3, 7)]
```

Para obter o resultado considere o código da função "**adjacentes**" a seguir:

```
dec adjacentes : list alpha -> list (alpha # alpha);
--- adjacentes xs <= zip (xs, tail xs);
```

5.4 - Compreensão de cadeias

As cadeias são na programação funcional listas de caracteres. Por exemplo,

```
>: "abc" = ['a','b','c'];
>> true : truval
```

A expressão

```
| "abc"
```

é equivalente a

```
| ['a','b','c']
```

pois ambas as ações devolvem como resposta a saída

```
| "abc" : list char
```

As funções sobre listas se aplicam a cadeias de caracteres:

```
>: length ("abcde");  
>> 5 : num  
  
>: reverse ("abcde");  
>> "edcba" : list char  
  
>: "abcde" <> "fg" ;  
>> "abcdefg" : list char
```

As operações de compreensão podem ser aplicadas sobre listas que tratam conjuntos de caracteres. A função "**minusculas(xs)**" retorna uma cadeia formada apenas pelos caracteres de "**xs**" que sejam minúsculos, desprezando qualquer caractere maiúsculo. Por exemplo,

```
|>: minusculas "EsteTextoFicaSemMaisculos";  
>> "steextoicaemaisculos" : list char
```

Para obter o resultado considere o código da função "**minusculas**" a seguir:

```
dec minusculas : list char -> list char;  
--- minusculas xs <= filter (\x => elem (ord(x), (ord('a')..ord('z'))), xs);
```

A função "**ocorrencias(x, xs)**" retorna o número de ocorrências do caractere "**x**" na cadeia "**xs**". Por exemplo,

```
|>: ocorrencias ('n', "Jimenez & Manzano");  
>> 3 : num
```

Para obter o resultado considere o código da função "**ocorrencias**" a seguir:

```
dec ocorrencias : char # list char -> num;  
--- ocorrencias (x, xs) <= length (filter (\x' => x = x', xs));
```

5.5 - Cifra de César

A cifra de César é um mecanismo de criptografia bastante simples, mas muito divertido de ser implementado em diversas linguagens de programa e em Hope não é diferente. Nesse mecanismo (algoritmo) de criptografia de textos, oficialmente, cada letra do alfabeto original é deslocada para outra letra 3 posições a sua frente.

A codificação da mensagem

```
| "em todas as medidas"
```

fica cifrada como

```
| "hp wrgdv dv phlgdv"
```

No entanto, é possível abstrair este mecanismo e definir a mudança do deslocamento das letras da mensagem "**n**" posições.

Desta forma, o deslocamento de 5 posições para a mensagem

```
| "em todas as medidas"
```

fica cifrada como

```
| "jr ytifx fx rjinifx"
```

A decodificação de um texto criptografado com deslocamento "**n**" é obtido codificando-o com deslocamento negativo de "**n**", ou seja, "**-n**", que em Hope deve ser indicado como "**0-n**".

A partir da ideia do que deve acontecer atente para os passos a seguir na elaboração das ações de codificação e decodificação da operação de criptografia.

Antes de mais nada é importante conhecer as aplicações básicas das funções "**ord**" e "**chr**" do arquivo de módulo "**Standard**".

A função "**ord**" tem por objetivo retornar o código ASCII³ do caractere informado

```
| ord 'a' = 97
| ord 'b' = 98
| ord 'A' = 65
```

A função "**chr**" tem por objetivo retornar o caractere do código ASCII informado

```
| chr 97 = 'a'
| chr 98 = 'b'
| chr 65 = 'A'
```

Para facilitar e simplificar esta atividade usar-se-á apenas caracteres minúsculos. Desta forma as letras de "**a**" até "**z**" a partir do código ASCII são representadas internamente no computador na faixa de valores de "**97**" até "**122**". No entanto, considere que esses valores estejam definidos na faixa de "**0**" até "**26**", sendo necessário para isso a definição de duas funções auxiliares a esta operação, sendo "**let2int**" e "**int2let**".

A função "**let2int**" tem por objetivo retornar o valor cardinal do caractere informado.

```
| let2int 'a' = 0
| let2int 'b' = 1
| let2int 'z' = 25
```

Observe o código da função "**let2int**":

```
dec let2int : char -> num;
--- let2int c <= ord c - ord 'a';
```

³ American Standard Code Interchange Information.

A função **"int2let"** tem por objetivo retornar o caractere do valor cardinal informado.

```
int2let 0 = 'a'
int2let 1 = 'b'
int2let 25 = 'z'
```

Veja o código da função **"int2let"**:

```
dec int2let : num -> char;
--- int2let n <= chr (ord 'a' + n);
```

A partir dessas funções básicas tem-se o necessário para desenvolver as funções que efetivarão os deslocamentos dos caracteres da mensagem.

Considere a função **"desloca(n, c)"** que tem por objetivo efetuar o deslocamento de um caractere **"c"** certa quantidade **"n"** de caracteres no alfabeto, desde que esse deslocamento esteja na faixa de **"0"** até **"25"**.

```
>: deslocar (3, 'a');
>> 'd' : char

>: deslocar (3, 'y');
>> 'b' : char
```

Atente para o código da função **"desloca"**:

```
dec deslocar : num # char -> char;
--- deslocar (n, c) <= if elem (ord c, ord 'a' .. ord 'z')
                        then int2let ((let2int c + n) mod 26)
                        else c;
```

OBS.:

Este livro, como já é sabido, é uma adaptação da obra *"Temas de programación funcional"* de José A. Alonso Jiménez da Universidade de Sevilla em que é usada a linguagem funcional Haskell. A adaptação da obra visa traduzir ou transliterar os programas Haskell para Hope. No entanto, apesar de ambas as linguagens pertencerem ao mesmo paradigma computacional há certos detalhes devido a características particulares de Hope em relação a Haskell que não permitem certas adaptações ou a adaptação tem um custo/benefício maior do que a didática aplicada e desejada. Nesses casos mais particulares optou-se por omitir os scripts com maior grau de incompatibilidade.

Observe a função **"recompor(n, c)"** que tem por objetivo efetuar a reposição deslocamento de um caractere **"c"** certa quantidade **"n"** de posições no alfabeto, desde que esse deslocamento esteja na faixa de **"0"** até **"25"**.

```
>: recompor (3, 'd');
>> 'a' : char

>: recompor (3, 'b');
>> 'y' : char
```

Observe o código da função "**recompor**":

```
dec recompor : num # char -> char;
--- recompor (n, c) <= if let2int c >= 0 and let2int c <= n - 1
    then int2let (26 + (let2int c - n))
    else if elem (ord c, ord 'a' .. ord 'z')
    then int2let ((let2int c - n) mod 26)
    else c;
```

A função "**codifica(n, xs)**" a seguir devolve a mensagem informada no argumento "**xs**" criptografada a partir da cifra de César com o deslocamento indicado em "**n**".

```
dec codifica : num # list char -> list char;
--- codifica (n, xs) <= map (xs, \x => deslocar (n, toLower x));
```

Veja o resultado da operação da função "**codifica**":

```
>: codifica (3, "atacar base inimiga ao amanhecer");
>> "dwdfdu edvh lqlpljd dr dpdqkhfhu" : list char
```

A função "**decodifica(n, xs)**" seguinte devolve a mensagem criptografada no argumento "**xs**" em sua forma original ao estabelecer o mesmo argumento de deslocamento "**n**" usado para criptografar a mensagem.

```
dec decodifica : num # list char -> list char;
--- decodifica (n, xs) <= map (xs, \x => recompor (n, x));
```

Note o resultado da operação da função "**codifica**":

```
>: decodifica (3, "dwdfdu edvh lqlpljd dr dpdqkhfhu");
>> "atacar base inimiga ao amanhecer" : list char
```

As operações com criptografia podem ser muito mais sofisticadas que este exemplo, mesmo para a cifra de César. Há diversos detalhes que estão sendo omitidos por fugirem da proposta desta obra.

CAPÍTULO 6 - Funções recursivas

Recursividade é a capacidade que uma função possui de chamar a si mesma. Uma função recursiva não pode chamar a si mesma infinitamente. Desta forma, é necessário que a função tenha a definição de uma condição de encerramento (aterramento) que é a solução mais simples e menor para o problema avaliado (BIRD & WADLER, 1998; BAILEY, 1985 & 1990).

6.1 - Recursividade numérica

Recursão numérica: A função "fatorial"

Um algoritmo clássico para a apresentação do efeito de recursão é o do cálculo do fatorial de um valor numérico natural. Observe o código para o cálculo da fatorial a partir de dois exemplos de funções **"fatorial2"** e **"fatorial3"**.

Solução 1:

```
dec fatorial2 : num -> num;
--- fatorial2 0      <= 1;
--- fatorial2 (n + 1) <= (n + 1) * fatorial2 n;
```

Solução 2:

```
dec fatorial3 : num -> num;
--- fatorial3 0 <= 1;
--- fatorial3 n <= n * fatorial3 (n - 1);
```

Cálculo da função **"fatorial2"**:

```
fatorial3 3 = 3 * (fatorial2 2)
            = 3 * (2 * (fatorial2 1))
            = 3 * (2 * (1 * (fatorial2 0)))
            = 3 * (2 * (1 * 1))
            = 3 * (2 * 1)
            = 3 * 2
            = 6
```

Recursão numérica: A função de produto "prod"

Código para a função que efetua o produto por soma de dois valores numéricos. Observe o código para o cálculo da função **"prod"** (não se preocupe com o uso da cláusula **"infix"** neste momento):

```
infix prod : 6;
dec prod : num # num -> num;
--- m prod 0      <= 0;
--- m prod (n + 1) <= m + (m prod n);
```

Cálculo da função **"prod"**:

```
3 prod 2 = 3 + (3 prod 1)
          = 3 + (3 + (3 prod 0))
          = 3 + (3 + 0)
          = 3 + 3
          = 6
```

6.2 - Recursividade sobre listas

Recursão sobre listas: A função "produto"

Veja o algoritmo para a apresentação do produto de uma lista de valores numéricos. Observe o código para o cálculo da função **"produto"**:

```
dec produto : list num -> num;
--- produto []      <= 1;
--- produto (n :: ns) <= n * produto ns;
```

Cálculo da função **"produto"**:

```
produto [7,5,2] = 7 * (produto [5,2])
                = 7 * (5 * (produto [2]))
                = 7 * (5 * (2 * (produto [])))
                = 7 * (5 * (2 * 1))
                = 7 * (5 * 2)
                = 7 * 10
                = 70
```

Recursão sobre listas: A função "tamanho"

Veja o algoritmo para a apresentação da longitude, do tamanho, de uma lista. Observe o código para o cálculo da função **"tamanho"**:

```
dec tamanho : list alpha -> num;
--- tamanho []      <= 0;
--- tamanho (_ :: xs) <= 1 + tamanho xs;
```

Cálculo da função **"tamanho"**:

```
tamanho[2,3,5] = 1 + (tamanho[3,5])
               = 1 + (1 + (tamanho[5]))
               = 1 + (1 + (1 + (tamanho[])))
               = 1 + (1 + (1 + 0))
               = 1 + (1 + 1)
               = 1 + 2
               = 3
```

Recursão sobre listas: A função "inverte"

Veja o algoritmo para a apresentação dos elementos de uma lista de forma invertida. Observe o código para o cálculo da função **"inverte"**:

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


Cálculo da função "tamanho":

```

inverte[2,5,3] = (inverte[5,3]) <> [2]
               = ((inverte[3]) <> [5]) <> [2]
               = (((inverte[]) <> [3]) <> [5]) <> [2]
               = ([] <> [3]) <> [5] <> [2]
               = ([3] <> [5]) <> [2]
               = [3,5] <> [2]
               = [3,5,2]

```

Recursão sobre listas: A função "||" (concatenação)

Veja o algoritmo para a apresentação da concatenação de duas listas em uma só lista. Observe o código para o cálculo da função "||":

```

infix || : 5;
dec (||) : list alpha # list alpha -> list alpha;
---      [] || ys <= ys;
--- (x :: xs) || ys <= x :: (xs || ys);

```

Cálculo da função "||":

```

[1,3,5] || [2,4] = 1 :: ([3,5] || [2,4])
                 = 1 :: (3 :: ([5] || [2,4]))
                 = 1 :: (3 :: (5 :: ([ ] || [2,4])))
                 = 1 :: (3 :: (5 :: [2,4]))
                 = 1 :: (3 :: [5,2,4])
                 = 1 :: [3,5,2,4]
                 = [1,3,5,2,4]

```

OBS.:

O código da função "||" pode não ser executado, mesmo estando escrito corretamente. Isso ocorre, por uma falha desconhecida. No Windows o ambiente Hope pode ser encerrado abruptamente após a tentativa de execução da função. No Linux é apresentada mensagem de erro indicando que não há memória suficiente. O problema acontece quando os parênteses da terceira linha "(x :: xs)" da lista "x :: xs" e "(xs || ys)" da expressão "x :: (xs || ys)" são inexplicavelmente removidos. Neste caso, uma forma de acertar este problema é após efetuar a gravação do módulo, sair do ambiente, abrir o arquivo do módulo em um editor de texto externo e colocar de volta os parênteses removidos.

Recursão sobre listas: Inserção classificada

A função "inserir(e, xs)" insere um elemento "e" na lista "xs" a frente do primeiro elemento de "xs" maior ou igual a "e". Por exemplo,

```

>: inserir (5, [2,4,7,3,6,8,10]);
>> [2,4,5,7,3,6,8,10] : list num

```

Observe o código para o cálculo da função "inserir":

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

Cálculo da função "inserir":

```
inserir 4 [1,3,5,7] = 1::(inserir 4 [3,5,7])
                   = 1::(3::(inserir 4 [5,7]))
                   = 1::(3::(4::(5::[7])))
                   = 1::(3::(4::[5,7]))
                   = [1,3,4,5,7]
```

Recursão sobre listas: Classificação crescente por inserção (insert sort)

A função "**clasIns(xs)**" classifica de forma crescente por inserção os elementos da lista "**xs**". Por exemplo,

```
>: clasIns [2,4,3,6,3];
>> [2,3,3,4,6] : list num
```

Observe o código para o cálculo da função "**clasIns**":

```
dec clasIns : list alpha -> list alpha;
--- clasIns []      <= [];
--- clasIns (x :: xs) <= inserir (x, clasIns xs);
```

Cálculo da função "**clasIns**":

```
clasIns[7,9,6] = inserir 7 (inserir 9 (inserir 6 []))
               = inserir 7 (inserir 9 [6])
               = inserir 7 [6,9]
               = [6,7,9]
```

6.3 - Recursão sobre vários argumentos

Recursão sobre vários argumentos: A função "zip"

A função "**zip(xs, ys)**" da arquivo de módulo "**mylist**" tem por finalidade combinar de forma alterada os elementos de duas listas em uma lista de pares de tuplas.

```
dec zip : list alpha # list beta -> list (alpha # beta);
--- zip ([], _)      <= [];
--- zip (_, [])      <= [];
--- zip (x :: xs, y :: ys) <= (x, y) :: zip (xs, ys);
```

Cálculo da função "**zip**":

```
zip [1,3,5] [2,4,6,8] = (1,2) : (zip [3,5] [4,6,8])
                     = (1,2) : ((3,4) : (zip [5] [6,8]))
                     = (1,2) : ((3,4) : ((5,6) : (zip [] [8])))
                     = (1,2) : ((3,4) : ((5,6) : []))
                     = [(1,2),(3,4),(5,6)]
```

Recursão sobre vários argumentos: A função "drop"

A função "**drop(n, xs)**" do arquivo de módulo "**mylist**" tem por finalidade eliminar os elementos iniciais da lista "**xs**" indicados em "**n**".

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

Cálculo da função "**zip**":

<pre>drop 2 [5,7,9,4] drop 1 [7,9,4] drop 0 [9,4] [9,4]</pre>	<pre>drop 5 [1,4] drop 4 [4] drop 1 [] []</pre>
---	---

6.4 - Recursão múltipla*Recursão múltipla: A função "fibonacci"*

A sequência de Fibonacci é formada pelos valores numéricos: "0, 1, 1, 2, 3, 5, 8, 13, 21, ...". Seus dois primeiros termos são 0 e 1 e o restante são obtidos adicionando os dois anteriores.

A função "**fibonacci(n)**" indica pelo argumento "**n**" o "**n**"-ésimo termo que determina o final da sucessão da série de Fibonacci. Por exemplo,

```
>: fibonacci 8;
>> 21 : num
```

Observe o código para o cálculo da função "**fibonacci**":

```
dec fibonacci : num -> num;
--- fibonacci 0      <= 0;
--- fibonacci 1      <= 1;
--- fibonacci (n + 2) <= fibonacci n + fibonacci (n + 1);
```

Recursão múltipla: Classificação rápida (quick sort)

A função "**clasRap(xs)**" classifica de forma crescente de forma rápida os elementos da lista "**xs**".

```
dec clasRap : list alpha -> list alpha;
--- clasRap []      <= [];
--- clasRap (x :: xs) <= clasRap menores <> [x] <> clasRap maiores
                        where menores == comp (xs, \a => a <= x)
                        where maiores == comp (xs, \b => b > x);
```

6.5 - Heurística para as definições recursivas*Aplicação do método: A função "produto"*

Passo 1: Definir o tipo.

```
dec clasRap : list alpha -> list alpha;
```

Passo 2: Enumerar os casos.

```
dec produto : list num -> num;
--- produto []      <=
--- produto (n :: ns) <=
```

Passo 3: Definir o caso mais simples (aterramento).

```
dec produto : list num -> num;
--- produto []      <= 1;
--- produto (n :: ns) <=
```

Passo 4: Definir os demais casos.

```
dec produto : list num -> num;
--- produto []      <= 1;
--- produto (n :: ns) <= n * produto ns;
```

Passo 5: Generalizar e simplificar.

```
dec produto : list num -> num;
--- produto xs <= foldr ((*), 1, xs);
```

Aplicação do método: A função "drop"

Passo 1: Definir o tipo.

```
dec drop : num # list alpha -> list alpha;
```

Passo 2: Enumerar os casos.

```
dec drop : num # list alpha -> list alpha;
--- drop (0, [])      <=
--- drop (0, xs)      <=
--- drop (n, [])      <=
--- drop (n, x :: xs) <=
```

Passo 3: Definir os casos mais simples (aterramento).

```
dec drop : num # list alpha -> list alpha;
--- drop (0, [])      <= [];
--- drop (0, xs)      <= xs;
--- drop (n, [])      <= [];
--- drop (n, x :: xs) <=
```

Passo 4: Definir os demais casos.

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

Passo 5: Generalizar e simplificar.

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

Aplicação do método: A função "init"

Passo 1: Definir o tipo.

```
dec init : list alpha -> list alpha;
```

Passo 2: Enumerar os casos.

```
dec init : list alpha -> list alpha;
--- init (x :: xs) <=
```

Passo 3: Definir os casos mais simples (aterramento).

```
dec init : list alpha -> list alpha;
--- init (x :: xs) <= if null xs
                        then []
                        else
```

Passo 4: Definir os demais casos.

```
dec init : list alpha -> list alpha;
--- init (x :: xs) <= if null xs
                        then []
                        else x :: init xs;
```

Passo 5: Generalizar e simplificar.

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

ANOTAÇÕES

CAPÍTULO 7 - Funções especializadas

As funções especializadas são recursos que aumentam a potencialidade das operações na programação funcional. Normalmente essas funções são definidas de forma genérica e utilizadas para o tratamento de diversas operações comuns a diversas necessidades (BIRD & WADLER, 1998; BAILEY, 1985 & 1990).

7.1 - Funções de ordem superior

Uma função de ordem superior é capaz de tomar uma função como seu argumento devolvendo uma função como resultado.

A função "**duasVezes(f, x)**" é o resultado de aplicar "**f**" a "**f x**". Por exemplo,

```
>: duasVezes ((*3), 2);
>> 18 : num

>: duasVezes (reverse, [2,5,7]);
>> [2, 5, 7] : list num
```

Observe o código para o cálculo da função "**duasVezes**":

```
dec duasVezes : (alpha -> alpha) # alpha -> alpha;
--- duasVezes (f, x) <= f (f x);
```

Usos das funções de ordem superior

- Definição de padrões de programação.
 - Aplicação de uma função a todos os elementos de uma lista.
 - Filtragem de listas por propriedades.
 - Padrões de recursão sobre listas.
- Definição de linguagem específica de domínio
 - Linguagem para processamento de mensagens.
 - Analisadores sintáticos.
 - Procedimentos de entrada/saída.
- Uso de propriedades algébricas de funções de ordem superior para raciocinar sobre programas.

7.2 - Processamento de listas

O processamento de listas é uma maneira de estender as funcionalidades das ações definidas para compreensões de listas.

A função "map"

A função "**map(f, xs)**" retorna uma lista aplicando "**f**" a cada elemento de "**xs**". Por exemplo,

```
>: map ((*2), [3,4,7]);
>> [6, 8, 14] : list num

>: map (sqrt, [1,2,4]);
>> [1, 1.4142135623731, 2] : list num
```

Definição da função "**map**" por recursão (existente no arquivo de módulo "**mylist**"):

```
map : (alpha -> beta) # list alpha -> list beta;
map (_, [])      <= [];
map (f, x :: xs) <= f x :: map (f, xs);
```

A função "filter"

A função "**filter(p, xs)**" retorna a lista dos elementos de "**xs**" que cumprem a propriedade "**p**". Por exemplo,

```
>: filter (even, [1,3,5,4,2,6,1]);
>> [4, 2, 6] : list num

>: filter ((>3), [1,3,5,4,2,6,1]);
>> [5, 4, 6] : list num
```

Definição da função "**filter**" por recursão (existente no arquivo de módulo "**mylist**"):

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

Funções "all", "any", "takeWhile" e "dropWhile" do arquivo de módulo "mylist"

A função "**all(p, xs)**" verifica se todos os elementos de "**xs**" cumprem a propriedade "**p**". Por exemplo,

```
>: all (odd, [1,3,5]);
>> true : truval

>: all (odd, [1,3,6]);
>> false : truval
```

A função "**any(p, xs)**" verifica se algum elemento de "**xs**" cumpre a propriedade "**p**". Por exemplo,

```
>: any (odd, [1,3,5]);
>> true : truval

>: any (odd, [2,4,6]);
>> false : truval
```


A função "**takeWhile(p, xs)**" mostra a lista "**xs**" sem os elementos iniciais que atendem o predicado "**p**". Por exemplo,

```
>: takeWhile(even, [2,4,6,7,8,9]);
>> [2, 4, 6] : list num
```

A função "**dropWhile(p, xs)**" mostra a lista dos elementos iniciais de "**xs**" que atendem o predicado "**p**". Por exemplo,

```
>: dropWhile(even, [2,4,6,7,8,9]);
>> [7, 8, 9] : list num
```

7.3 - Função de dobra à direita (foldr)

Esquema básico de recursão sobre listas

Exemplos de funções recursivas disponibilizadas no arquivo de módulo "**mylist**" (algumas):

```
sum []      <= 0;
sum (x :: xs) <= x + sum xs;

last ([x])  <= x;
last (x :: xs) <= last xs;

map (_, []) <= [];
map (f, x :: xs) <= f x :: map (f, xs);

drop (0, xs) <= xs;
drop (n, []) <= [];
drop (n, x :: xs) <= drop (n - 1, xs);
```

Esquema básico de recursão sobre listas:

```
f []      <= valor;
f (x :: xs) <= x <operação> f xs;
```

O padrão para a função "foldr"

Redefinições com o padrão de dobra "**foldr**":

```
sum      = foldr ((+), 0, [1,2,3,4]) => 10
product = foldr ((*), 1, [1,2,3,4]) => 24
```

Definição da função "**foldr**" no arquivo de módulo "**mylist**":

```
dec foldr : (alpha # beta -> beta) # beta # list alpha -> beta;
--- foldr (f, v, [])      <= v;
--- foldr (f, v, x :: xs) <= f (x, foldr (f, v, xs));
```

Visão recursiva de funcionamento da função "foldr"

Cálculo com a função "sum":

sum ([2,3,5]) = foldr ((+), 0, [2,3,5])	[definição de sum]
= foldr ((+), 0, 2::(3::(5::[])))	[notação de lista]
= 2+(3+(5+0))	[subst. (::) por (+) e "[]" por "0"]
= 10	[aritmética]

Cálculo com a função "product":

product ([2,3,5]) = foldr ((*), 1, [2,3,5])	[definição de product]
= foldr ((*), 1, 2::(3::(5::[])))	[notação de lista]
= 2*(3*(5*1))	[subst. (::) por (*) e "[]" por "1"]
= 30	[aritmética]

Cálculo com a função "foldr(f, v, xs)":

Substituir em "xs" os "(:)" por "f" e "[]" por "v".

Definição de tamanho (longitude) de lista com a função "foldr"

Exemplo do cálculo da longitude de uma lista:

longitude ([2,3,5])	
= longitude 2::(3::(5::[]))	
= 1+(1+(1+0))	[Substituições]
= 3	

Substituições:

- os "(:)" por "(_, n => 1+n), 0";
- o "[]" por "0".

Definição da função "longitude" (clone da função "length") usando a função "foldr"

```
dec longitude : list alpha -> num;
-- longitude xs <= foldr ((\_, n => 1+n), 0, xs);
```

Definição de inversão de lista com a função "foldr"

Exemplo de cálculo com inversão de lista:

inversa ([2,3,5])	
= inversa 2 :: (3 :: (5 :: []))	
= (([] <> [5]) <> [3]) <> [2]	[Substituições]
= [5,3,2]	

Substituições:

- os "(:)" por "(\x, xs => xs <> [x])";
- o "[]" por "[]".

Definição da função "**inversa**" (clone da função "**reverse**") usando a função "**foldr**"

```
dec inversa : list alpha -> list alpha;
--- inversa xs <= foldr ((\x, xs => xs <> [x]), [], xs);
```

Definição de concatenação com a função "foldr"

Exemplo de cálculo com concatenação:

```
| conc ([2,3,5], [7,9])
= cons 2 :: (3 :: (5 :: []), [7,9])
=      2 :: (3 :: (5 :: [7,9]))      [Substituições]
=      [2,3,5,7,9]
```

Substituições:

- os "(::)" por "(::)";
- o "[]" por "ys".

Definição da função "**conc**"

```
dec conc : list alpha # list alpha -> list alpha;
--- conc (xs, ys) <= foldr ((::), ys, xs);
```

7.4 - Função de dobra à esquerda (foldl)

Definição de soma de lista com acumuladores

Definição da função "**somac**" com acumuladores:

```
dec somacAux : num # list num -> num;
--- somacAux (v, [])      <= v;
--- somacAux (v, x :: xs) <= somacAux (v + x, xs);

dec somac : list num -> num;
--- somac xs <= somacAux (0, xs);
```

Cálculo com a função "**somac**":

```
| somac [2,3,7] = somacAux (0, [2,3,7])
                  = somacAux (0+2, [3,7])
                  = somacAux (2, [3,7])
                  = somacAux (2+3, [7])
                  = somacAux (5, [7])
                  = somacAux (5+7, [])
                  = somacAux (12, [])
                  = 12
```

Padrão de definição de recursão com acumulador

Padrão de definição (generalização da função "**somacAux**"):

```
| f (v, [])      = v
| f (v, x :: xs) = f (v + x, xs)
```

Definições com o padrão de dobra "**foldl**":

```
sum      = foldl ((+), 0, [1,2,3,4]) => 10
product = foldl ((*), 1, [1,2,3,4]) => 24
```

Definição da função "**foldl**" no arquivo de módulo "**mylist**":

```
dec foldl : (alpha # beta -> alpha) # alpha # list beta -> alpha;
--- foldl (f, v, [])      <= v;
--- foldl (f, v, x :: xs) <= foldl (f, f (v, x), xs);
```

7.5 - Composição de funções

Definição da composição das funções a partir do arquivo de módulo "**Standard**":

```
dec o : (beta -> gamma) # (alpha -> beta) -> alpha -> gamma;
--- (f o g) x <= f(g x);
```

Uso de composição para simplificar definições.

Definição sem composição:

```
even n      = not (even n)
duasVezes (f, x) = f (f x)
```

Definição com composição:

```
even n      = (not o even) n
duasVezes f  = (f o f) x
```

7.6 - Estudo de caso: Codificação binária e transmissão de cadeias

Os objetivos desta proposta são:

- definir uma função que converta uma cadeia de uma lista de zeros e uns junto de outra função que realize a conversão oposta;
- simular a transmissão da cadeia mediante zeros e uns.

Os números binários são representados mediante listas de bits em ordem inversa. Um bit é um número "zero" ou "um". Por exemplo, o número "1101" é representado por "[1,0,1,1]".

Para facilitar a representação de dados binários está sendo definido um tipo de dado customizado chamado "**bit**" definido a partir do tipo "**num**".

```
type bit == num;
```

Conversão de bases: decimal para binário

Para realizar a conversão de um número decimal em binário considere a função "**bin2int(x)**" que retorna o número decimal do binário "**x**". Veja o código:

```
dec bin2int : list bit -> num;
--- bin2int xs <= foldr (\x,y => x + 2 * y, 0, xs);
```

Veja o uso da função "**bin2int**" com a apresentação do número invertido:

```
>: bin2int [1,0,1,1];
>> 13 : num

>: bin2int [0,1,0,1];
>> 10 : num
```

Conversão de bases: binário para decimal

Para realizar a conversão de um número binário em decimal considere a função "**int2bin(x)**" que retorna o número binário do decimal "**x**". Veja o código:

```
dec int2bin : num -> list bit;
--- int2bin n <= if n < 2
                    then [n]
                    else n mod 2 :: int2bin (n div 2);
```

Veja o uso da função "**int2bin**":

```
>: int2bin 13;
>> [1, 0, 1, 1] : list bit

>: int2bin 10;
>> [0, 1, 0, 1] : list bit
```

Por exemplo,

```
int2bin 13 = 13 mod 2 :: int2bin (13 div 2)
           = 1 :: int2bin (6 div 2)
           = 1 :: (6 mod 2 :: int2bin (6 div 2))
           = 1 :: (0 :: int2bin 3)
           = 1 :: (0 :: (3 mod 2 :: int2bin (3 div 2)))
           = 1 :: (0 :: (1 :: int2bin 1))
           = 1 :: (0 :: (1 :: (1 :: int2bin 0)))
           = 1 :: (0 :: (1 :: (1 :: [])))
           = [1,0,1,1]
```

A partir da composição básica dos recursos que garantem a conversão de bases entre binários e decimais passa-se a parte de codificação de uma mensagem texto em sequência binária e a decodificação da sequência binária em forma de texto.

Criação de octetos (bytes)

Um octeto, do inglês *byte*, representa um conjunto de oito elementos. Neste contexto representado pelo tipo "**bit**", ou seja, um conjunto de oito bits.

A função "**criaOcteto(bs)**" é o *byte* correspondente à lista de bits "**bs**", ou seja, serão os oito primeiros elementos de "**bs**" se seu comprimento for maior ou igual a "8", caso contrário a sequência de bits deverá ser ajustada para o tamanho de oito bits adicionando zeros ao final de "**bs**". Observe o código a seguir:

```
dec criaOcteto : list bit -> list bit;
--- criaOcteto bs <= take (8, (bs <> repeat (length bs, 0)));
```

A função "**repeat**" gera uma sequência de zeros de acordo com o tamanho de "**bs**" concatenando essa sequência ao final "**bs**" que tem em seguida suas primeiras 8 posições (o que realmente interessa) extraídas pela função "**take**".

Veja o uso da função "**criaOcteto**":

```
>: criaOcteto [1,0,1,1,0,0,1,1,1,0,0,0];
>> [1, 0, 1, 1, 0, 0, 1, 1] : list bit

>: criaOcteto [1,0,1,1];
>> [1, 0, 1, 1, 0, 0, 0, 0] : list bit
```

Codificação de um texto em formato binário

A função a seguir "**codifica(c)**" codifica a cadeia "**c**" em uma lista de bits a partir da conversão de cada caractere em um número ASCII⁴, convertendo cada caractere em um octeto e concatenando os octetos para obter uma lista de bits. Veja o código seguinte:

```
dec codifica : list char -> list bit;
--- codifica xs <= concat (map (criaOcteto o int2bin o ord, xs));
```

Veja o uso da função "**codifica**":

```
>: codifica "abc";
>> [1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0] : list bit
```

Por exemplo,

```
codifica "abc" = concat (map (criaOcteto o int2bin o ord, "abc"))
                = concat (map (criaOcteto o int2bin o ord, ['a', 'b', 'c']))
                = concat (map (criaOcteto o int2bin o ord 'a',
                               criaOcteto o int2bin o ord 'b',
                               criaOcteto o int2bin o ord 'c'))
                = concat (map (criaOcteto o int2bin o ord [1,0,0,0,0,1,1],
                               criaOcteto o int2bin o ord [0,1,0,0,0,1,1],
                               criaOcteto o int2bin o ord [1,1,0,0,0,1,1]))
                = concat ([1,0,0,0,0,1,1,0], [0,1,0,0,0,1,1,0], [1,1,0,0,0,1,1,0])
                = [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

Separação de lista de bits em octetos

Uma lista de bits, não importando seu tamanho pode ser separada em octetos. Para tanto, considere a função "**separaOctetos**" que retorna uma lista de bits "**bs**" em listas de oito elementos. Veja o código:

```
dec separaOctetos : list bit -> list (list bit);
--- separaOctetos [] <= [];
--- separaOctetos bs <= take (8, bs) :: separaOctetos (drop (8, bs));
```

⁴ A linguagem Hope opera com caracteres no formato ASCII, não aceitando o formato Unicode.

Veja o uso da função "**separaOctetos**":

```
>: separaOctetos [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0];  
>> [[1, 0, 0, 0, 0, 1, 1, 0], [0, 1, 0, 0, 0, 1, 1, 0]] : list (list bit)
```

Decodificação de uma sequência binária em texto

Considere para decodificar uma sequência binária em texto a função "**decodifica(bs)**" que retorna uma cadeia de caracteres a partir da sequência binária "**bs**". Veja o código:

```
dec decodifica : list bit -> list char;  
--- decodifica bs <= map (chr o bin2int, separaOctetos bs);
```

Veja o uso da função "**decodifica**":

```
>: decodifica [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0];  
>> "abc" : list char
```

Por exemplo,

```
decodifica ([1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0])  
= map (chr o bin2int, separaOctetos [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0])  
  
= map (chr o bin2int, [[1,0,0,0,0,1,1,0], [0,1,0,0,0,1,1,0], [1,1,0,0,0,1,1,0]])  
= map (chr o bin2int, [[1,0,0,0,0,1,1,0],  
    (chr o bin2int, [0,1,0,0,0,1,1,0],  
    (chr o bin2int, [1,1,0,0,0,1,1,0])])  
= map (chr 97, chr 98, chr 99])  
= "abc"
```

Transmissão

Os canais de transmissão podem ser representados por funções que transformam cadeias de bits de um ponto de origem, fazem o envio desses bits pelo canal e no ponto de destino decodificam os bits na forma original. É mais ou menos assim que funciona, por exemplo, um protocolo de rede como o TCP/IP⁵.

A função "**transmite(c, t)**" efetua a simulação do envio da mensagem "**t**" pelo canal "**c**" realizando a codificação do conteúdo "**t**" em binário do ponto de origem e reconvertendo para texto o conteúdo transmitido no ponto de destino. Veja o código:

```
dec transmite : (list bit -> list bit) # list char -> list char;  
--- transmite (c, t) <= decodifica (codifica (t));
```

Veja o uso da função "**transmite**":

```
>: transmite (id, "Alonso & Manzano");  
>> "Alonso & Manzano" : list char
```

⁵ Transmission Control Protocol / Internet Protocol.

ANOTAÇÕES

CAPÍTULO 8 - Tipos de dados customizados

Além dos tipos de dados clássicos existentes em Hope, a linguagem permite criar tipos ou definir tipos a partir dos tipos existentes. Os tipos definidos pelo usuário tornam os programas mais claros e ajudam o verificador de tipos a auxiliar o programador (BAILEY, 1985).

8.1 - Declaração de tipos

A declaração de tipos de dados permite que seja definido um novo nome de tipo para um tipo existente no *ecossistema hope* a partir da cláusula **"type"**.

Veja a declaração de sinônimos:

```
>: type string == list char;
>:
```

O tipo **"list char"** pode agora ser referenciado como **"string"**. No entanto, internamente o sinônimo faz referência real ao tipo original associado, ou seja, o tipo **"string"** é um apelido ao tipo **"list char"**. Com essa estratégia é possível definir tipos de dados elucidativos ao contexto da aplicação.

Definição da função **"souString"** que mostra o dado informado como **"string"** e não como **"list char"**:

```
dec souString : string -> string;
--- souString []      <= "";
--- souString (x :: xs) <= x :: souString xs;
```

A função **"souString(txt)"** mostra o conteúdo de **"txt"** como sendo do tipo **"string"**. Por exemplo,

```
>: souString [];
>> nil : string

>: souString "Alonso & Manzano";
>> "Alonso & Manzano" : string
```

É possível criar tipos novos de dados, além de definir sinônimos. Por exemplo, definir um par de números como definição de certa posição cartesiana.

```
>: type pos == num # num;
>:
```

Veja a definição da função **"origem"** que define o par de posições em **"(0,0)"**.

```
dec origem : pos;
--- origem <= (0,0);
```

A função **"origem"** retorna as coordenadas **"(0,0)"**:

```
>: origem;
>> (0, 0) : pos
```

Veja a definição da função "**esquerda(p)**" que mostra a posição à esquerda da posição "**p**".

```
dec esquerda : pos -> pos;
--- esquerda (x, y) <= (x - 1, y);
```

A função "**esquerda**" retorna o valor à esquerda do primeiro valor que forma o par de coordenada:

```
>: esquerda (3,5);
>> (2, 5) : pos
```

Outro recurso de definição de tipo é a possibilidade de se fazer a criação de tipos parametrizados, onde o tipo pode possuir parâmetros.

```
>: type par alpha == alpha # alpha;
>:
```

Veja que o tipo "**par**" possui definido o parâmetro "**alpha**" o qual pode ser usado de modo polimórfico. Observe a função "**multiplica(x, y)**" a seguir que faz uso deste recurso.

```
dec multiplica : par num -> num;
--- multiplica (x, y) <= x * y;
```

A função "**multiplica**" retorna o valor da multiplicação de dois valores definidos a partir do tipo parametrizado "**par num**" como "**num**":

```
>: multiplica (2,5);
>> 10 : num
```

Observe na sequencia o uso do tipo parametrizado "**par alpha**" junto a função "**copia(x)**" que efetua uma cópia do argumento passado retomando-o na forma de tupla.

```
dec copia : alpha -> par alpha;
--- copia x <= (x, x);
```

Veja o uso da função "**copia**" mostrando o efeito no uso de tipos parametrizados:

```
>: copia 1;
>> (1, 1) : par num

>: copia 'a';
>> ('a', 'a') : par char
```

8.2 - Definição de tipos

A definição de tipos difere da declaração de tipos. Na declaração de tipos baseia-se sobre os tipos existentes no *ecossistema hope*. Já a definição de tipos permite que tipos sejam criados do zero a partir da cláusula "**data**".

Um tipo de dados da linguagem nesta categoria é o tipo "**truval**" definido junto ao arquivo de módulo "**Standard**" que caracteriza-se por ser um tipo booleano formado pelos valores "**true**" e "**false**".

```
data truval == false ++ true;
```

O símbolo `"++"` lê-se como `"ou"`.

Os valores `"true"` e `"false"` são os *construtores* do tipo `"truval"`.

A partir da mesma estrutura é possível definir tipos de dados próprios. Veja a definição de um tipo chamado `"mov"` contendo os construtores `"aesquerda"`, `"adireita"`, `"acima"` e `"abaixo"`.

```
>: data mov == aesquerda ++ adireita ++ acima ++ abaixo;
>:
```

Para fazer uso do tipo observe a função `"movimento(m, p)"` que retorna o movimento `"m"` sobre a posição `"p"`. Veja o código:

```
dec movimento : mov # pos -> pos;
--- movimento (aesquerda, (x, y)) <= (x - 1, y);
--- movimento (adireita, (x, y)) <= (x + 1, y);
--- movimento (acima, (x, y)) <= (x, y + 1);
--- movimento (abaixo, (x, y)) <= (x, y - 1);
```

Veja o uso da função `"movimento"`:

```
>: movimento (acima, (2,5));
>> (2, 6) : pos

>: movimento (abaixo, (2,5));
>> (2, 4) : pos

>: movimento (aesquerda, (2,5));
>> (1, 5) : pos

>: movimento (adireita, (2,5));
>> (3, 5) : pos
```

Os tipos de dados definidos podem ser usados com listas. Veja a função `"movimentos(ms, p)"` que aplica o efeito de posicionamento sobre uma lista de movimentos `"ms"` da posição `"p"`. Olhe o código:

```
dec movimentos : list mov # pos -> pos;
--- movimentos ([], p) <= p;
--- movimentos (m :: ms, p) <= movimentos (ms, movimento (m, p));
```

Veja o uso da função `"movimentos"`:

```
>: movimentos ([acima, aesquerda], (2,5));
>> (1, 6) : pos
```

Na sequência considere uma função chamada `"oposto(m)"` que execute o movimento oposto indicado em `"m"`. Veja o código:

```
dec oposto : mov -> mov;
--- oposto aesquerda <= adireita;
--- oposto adireita <= aesquerda;
--- oposto acima <= abaixo;
--- oposto abaixo <= acima;
```

Veja o uso da função "oposto":

```
>: movimento (oposto acima, (2,5));
>> (2, 4) : pos
```

Assim como os tipos declarados possuem parâmetros os tipos definidos também o tem. Veja o exemplo de definição a seguir:

```
data figura == circulo num ++ retangulo (num # num);
```

Veja os tipos dos construtores:

```
>: id circulo;
>> circulo : num -> figura

>: id retangulo;
>> retangulo : num # num -> figura
```

Observe na sequência o uso do tipo como valor da função "quadrado(n)" que retorna o quadrado de "n". Olhe o código:

```
dec quadrado : num -> figura;
--- quadrado n <= retangulo (n, n);
```

Veja o uso da função "quadrado":

```
>: quadrado 5;
>> retangulo (5, 5) : figura
```

Observe na sequência o uso do tipo como argumento da função "area(f)". Veja o código:

```
dec area : figura -> num;
--- area (circulo r)          <= pi * pow (r, 2);
--- area (retangulo (x, y)) <= x * y;
```

Veja o uso da função "area":

```
>: area (circulo 1);
>> 3.14159265358979 : num

>: area (circulo 2);
>> 12.5663706143592 : num

>: area (retangulo (2,5));
>> 10 : num

>: area (quadrado 3);
>> 9 : num
```

Agora é o momento de ver a definição de tipos como parâmetros. Como exemplo, considere o tipo "maybe" que possui os construtores "nothing" e "just alpha".

```
data maybe alpha == nothing ++ just alpha;
```

Para fazer uso do tipo "**maybe**" considere a função "**divisaoSeg(m, n)**" que retorna a divisão de "**m**" por "**n**", caso "**n**" não seja zero ou nada se for o contrário. Veja o código:

```
dec divisaoSeg : num # num -> maybe num;
--- divisaoSeg (_, 0) <= nothing;
--- divisaoSeg (m, n) <= just (m div n);
```

Veja o uso da função "**divisaoSeg**":

```
>: divisaoSeg (6, 3);
>> just 2 : maybe num

>: divisaoSeg (6, 0);
>> nothing : maybe num
```

A função a seguir "**headSeg(xs)**" apresenta a cabeça de uma lista "**xs**" se não for vazia e não mostra nada em caso contrário. Olhe o código:

```
headSeg : list alpha -> maybe alpha;
headSeg [] <= nothing;
headSeg xs <= just (head xs);
```

Veja o uso da função "**headSeg**":

```
>: headSeg [2,3,5];
>> just 2 : maybe num

>: headSeg [];
>> nothing : maybe alpha
```

8.3 - Definição de tipos recursivos

Os tipos de dados definidos com a cláusula "**data**" podem ser recursivos. Um exemplo desta ocorrência é a definição do tipo de dado natural "**nat**".

Os dados naturais são construídos com o zero e a função sucessora "**succ**" do arquivo de módulo "**Standard**".

```
data nat == zero ++ succ nat;
```

Tipos dos construtores:

```
>: zero;
>> zero : nat

>: succ;
>> succ : nat -> nat
```

Exemplos de naturais:

```
>: zero;
>> zero : nat

>: succ zero;
>> succ zero : nat

>: succ (succ zero);
>> succ (succ zero) : nat

>: succ (succ (succ zero));
>> succ (succ (succ zero)) : nat
```

Definições com tipos recursivos

A função "**nat2int(n)**" retorna o número inteiro correspondente ao número natural "**n**". Veja os detalhes:

```
dec nat2int : nat -> num;
--- nat2int zero <= 0;
--- nat2int (succ n) <= 1 + nat2int n;
```

```
>: nat2int (succ (succ (succ zero)));
>> 3 : num
```

A função "**int2nat(i)**" retorna o número natural correspondente ao número inteiro "**i**". Veja os detalhes:

```
dec int2nat : num -> nat;
--- int2nat 0 <= zero;
--- int2nat (n + 1) <= succ (int2nat n);
```

```
>: int2nat 3;
>> succ (succ (succ zero)) : nat
```

A função "**somaNat(m, n)**" retorna a soma dos números naturais "**m**" e "**n**". Veja os detalhes:

```
dec somaNat : nat # nat -> nat;
--- somaNat (zero, n) <= n;
--- somaNat (succ m, n) <= succ (somaNat (m, n));
```

```
>: somaNat ((succ (succ zero)), (succ zero));
>> succ (succ (succ zero)) : nat
```

Tipo recursivo como parâmetro: Listas

Definição do tipo lista:

```
data lista alpha == nulo ++ cons alpha (lista alpha);
```

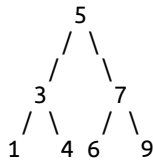
A função "**longitNat(xs)**" retorna o tamanho (longitude) da lista "**xs**". Veja os detalhes:

```
dec longitNat : lista alpha -> num;
--- longitNat nulo <= 0;
--- longitNat (cons _ xs) <= 1 + longitNat xs;
```

```
>: longitNat (cons 2 (cons 3 (cons 5 nulo)));
>> 3 : num
```

Tipo recursivo como parâmetro: Árvores binárias

Exemplo de árvore binária:



Definição do tipo de árvore binária:

```
data arvore == folha num ++ noh arvore num arvore;
```

Representação estrutural da constante "**ehArvore**" baseada na árvore binária indicada para os exemplos:

```
dec ehArvore : arvore;
--- ehArvore <= noh (noh (folha 1) 3 (folha 4)) 5 (noh (folha 6) 7 (folha 9));
```

Definições sobre árvores binárias

A função "**ocorre(m, a)**" verifica se a folha "**m**" ocorre na árvore "**a**". Veja os detalhes:

```
dec ocorre : num # arvore -> truval;
--- ocorre (m, folha n)      <= m = n;
--- ocorre (m, (noh i n d)) <= m = n or ocorre (m, i) or ocorre (m, d);
```

```
>: ocorre (4, ehArvore);
>> true : truval

>: ocorre (10, ehArvore);
>> false : truval
```

A função "**desfolhar(a)**" gera uma lista com as folhas retiradas da árvore "**a**". Veja os detalhes:

```
dec desfolhar : arvore -> list num;
--- desfolhar (folha n)  <= [n];
--- desfolhar (noh i n d) <= desfolhar i <> [n] <> desfolhar d;
```

```
>: desfolhar ehArvore;
>> [1, 3, 4, 5, 6, 7, 9] : list num
```

Uma árvore é classificada se o valor de cada nó for maior que o de sua subárvore esquerdo e menor do que aqueles em sua subárvore direita. A árvore do exemplo está classificada.

A função "**ocorreNaArvoreOrd(m, a)**" verifica se a folha "**m**" ocorre, se existe na árvore "**a**". Veja os detalhes:

```
dec ocorreNaArvoreOrd : num # arvore -> truval;
--- ocorreNaArvoreOrd (m, (folha n))    <= m = n;
--- ocorreNaArvoreOrd (m, (noh i n d)) <= if m = n then true else
                                           if m < n then ocorreNaArvoreOrd (m, i)
                                           else ocorreNaArvoreOrd (m, d);
```

```
>: ocorreNaArvoreOrd (4, ehArvore);
>> true : truval

>: ocorreNaArvoreOrd (10, ehArvore);
>> false : truval
```

Definições de diferentes árvores binárias

Árvores binárias com valores nas folhas:

```
data arvore alpha == folha alpha ++ noh (arvore alpha) (arvore alpha);
```

Árvores binárias com valores nos nós:

```
data arvore alpha == folha ++ noh (arvore alpha) (arvore alpha);
```

Árvores binárias com valores em folhas e nós:

```
data arvore alpha beta == folha ++ noh (arvore alpha beta) (arvore alpha beta);
```


CAPÍTULO 9 - Avaliação preguiçosa

A avaliação preguiçosa (*lazy evaluation*) é uma estratégia operacional existente em diversas linguagens funcionais, possuindo duas propriedades essenciais: a avaliação de uma expressão é suspensa ou atrasada (*delayed*) até que resultado necessário seja calculado; a partir do momento em que o resultado é calculado ele é "memorizado" em *cache*, de modo a ficar disponível caso necessite ser reutilizado, evitando-se desta forma seu recálculo (GEH, 2021).

9.1 - Estratégias de avaliação

A programação funcional pode ser realizada a partir de estratégias diferentes e ser avaliada a partir de diferentes óticas. Veja a seguir exemplos de avaliação com base em parâmetro por valor e por nome.

Para os exemplos a seguir considere a função "**meuMult(x, y)**" que devolve a multiplicação dos argumentos "**x**" e "**y**". Veja os detalhes:

```
dec meuMult : num # num -> num;
--- meuMult (x, y) <= x * y;
```

```
>: meuMult (1+2, 2+3);
>> 15 : num
```

Avaliação passando parâmetros por valor (ou mais internos):

```
meuMult (1+2,2+3)
= meuMult ( 3, 5) [por definição de "+"]
=          3 * 5 [por definição de meuMult]
=          15 [por definição de "*"]
```

Avaliação passando parâmetros por nome (ou outros externos):

```
meuMult (1+2,2+3)
= meuMult (1+3*2+3) [por definição de meuMult]
=          3 * 5 [por definição de "+"]
=          15 [por definição de "*"]
```

Para o exemplo a seguir considere a função "**meuMult' x y**" que devolve a multiplicação dos argumentos "**x**" e "**y**". Veja os detalhes:

```
dec meuMult' : num -> num -> num;
--- meuMult' x <= (\y => x * y);
```

```
>: meuMult' (1+2) (2+3);
>> 15 : num
```

Avaliação com expressão lambda:

```
meuMult' (1+2)      (2+3)
= meuMult' 3        (2+3) [por definição de "+"]
= (\y -> 3 * y) (2+3) [por definição de meuMult']
= (\y -> 3 * y) 5 [por definição de "+"]
= 3 * 5 [por definição de "+"]
= 15 [por definição de "*"]
```

9.2 - Terminação

A linguagem Hope por ser uma ferramenta experimental e minimalista não possui certos recursos definidos como compressões de listas e geração de listas infinitas de forma aprimorada como se tem, por exemplo, na linguagem Haskell. No entanto, é uma excelente linguagem para iniciantes devido a sua leveza e simplicidade. Neste sentido, as gerações de listas infinitas podem ser realizadas desde que estabeleça um limite máximo para evitar o "estouro" da pilha de memória (*stack overflow*).

Processamento ao "infinito" (simulado)

A definição de valores "infinitos" pode, em tese, ser estabelecido a partir da função "inf" que ao ser executada em Hope apresenta mensagem de erro informando estar fora da memória. O mesmo algoritmo em outras linguagens funcionais efetua o processamento, aguardando a interrupção da execução. O efeito em Hope ocorre devido ao mecanismo do ambiente da linguagem armazenar primeiro os valores na pilha de memória para descarregar esses valores ao término da execução. Como a pilha sobrecarregada o processamento é abortado. Mas, há forma de mitigar esse problema que será apresentada em instantes.

Para o processamento típico de valores infinitos pode-se fazer uso do código da função "inf", a qual entra em processamento calculando em memória esses valores, mas não os apresentando. Veja os detalhes:

```
dec inf : num;
--- inf <= 1 + inf;
```

```
>: inf;
run-time error - out of memory
>:
```

Avaliação de infinito (apesar do erro ocorrido):

```
inf
= 1 + inf           [por definição inf]
= 1 + (1 + inf)     [por definição inf]
= 1 + (1 + (1 + inf)) [por definição inf]
= ...              [até ser interrompido com: out of memory]
```

A mensagem "run-time error - out of memory" ocorre quando a ação recursiva atinge o limite máximo que o ambiente da linguagem Hope permite chegar. Veja que esse efeito é controlado pelo ambiente, pelo *ecossistema hope* e não pela linguagem em si.

O arquivo de módulo "mylist" possui a função "from" que gera listas "infinitas" até o valor limite de "140000", sendo este um limite de segurança um pouco menor do que o ambiente consegue suportar. Desta forma, a geração de valores infinitos pode ser simulada de maneira confortável se assemelhando a outras linguagens funcionais.

9.3 - Número de reduções

As ações de redução também podem ser avaliadas a partir do parâmetro por valor ou por nome. Veja a função "quadrado(n)" que apresenta o resultado do quadro do argumento "n":

```
dec quadrado : num -> num;
--- quadrado n <= n * n;
```

```
>: quadrado (1+2);
>> 9 : num
```

Avaliação passando parâmetros por valor:

```
quadrado (1+2)
= quadrado 3 [por definição "+"]
= 3*3 [por definição quadrado]
= 9 [por definição de "*"]
```

Avaliação passando parâmetros por nome:

```
quadrado (1+2)
= (1+2)*(1+2) [por definição quadrado]
= 3*(1+2) [por definição de "+"]
= 3 * 3 [por definição de "+"]
= 9 [por definição de "*"]
```

Avaliação preguiçosa e ansiosa

Na avaliação com passagem de parâmetro por nome os argumentos podem ser avaliados mais vezes do que a passagem por a valor.

Ponteiros podem ser usados para compartilhar valores de expressão.

Avaliação passando parâmetros por nome usando ponteiros para compartilhar valores de expressão é chamada de *avaliação preguiçosa*.

A avaliação pela passagem de parâmetros por valor é chamada de *avaliação ansiosa*.

Avaliação preguiçosa para a função "quadrado":

```
quadrado (1+2)
= n * n com n = 1+2 [por definição quadrado]
= 3 * 3 [por definição de "+"]
= 9 [por definição de "*"]
```

Hope usa avaliação preguiçosa.

9.4 - Estruturas infinitas

Quando se fala em listas infinitas imagina-se uma lista com um valor inicial, um salto de variação de contagem entre o primeiro e segundo elementos estendendo-se infinitamente. E isso é cumprido em diversas linguagens funcionais, exto Hope que terá, para os exemplos deste texto, o limite controlado em "140000" como definido no arquivo de módulo "mylist" junto a função "from".

```
dec from : num -> list num;
--- from (n) <= if n > 140000
    then error ("Maximum limit to infinity: 140000")
    else n .. 140000;
```

Considere a função "uns" que apresente infinitamente uma lista com valores um. Observe o código para a função.

```
dec uns : list num;
--- uns <= 1 :: uns;
```

Avaliação:

```
uns
= 1 :: uns           [por definição uns]
= 1 :: (1 :: uns)    [por definição uns]
= 1 :: (1 :: (1 :: uns)) [por definição uns]
= ...
```

No entanto, ao ser executada a função ao invés de apresentar uma lista com valores "1" é retornado a mensagem "run-time error - out of memory":

```
>: uns;
run-time error - out of memory
>:
```

Isso exige uma mudança de pensamento. Assim sendo, observe o código da função "uns'":

```
dec uns' : list num;
--- uns' <= repeat (140000, 1);
```

O valor "140000" é o estabelecimento do limite de segurança para o ambiente Hope. O argumento "1" é o indicativo de representação de uma lista "infinita" formada por uns.

```
>: uns';
>> [1, 1, 1, 1, 1, 1, ... ] : list num
```

Veja exemplo de avaliação ansiosa para a extração da cabeça de uma lista infinita de uns:

```
head uns'
= head (1 :: uns')           [por definição uns']
= head (1 :: (1 :: uns'))    [por definição uns']
= head (1 :: (1 :: (1 :: uns')) [por definição uns']
= ...
```

Veja exemplo de avaliação preguiçosa para a extração da cabeça de uma lista infinita de uns:

```
head uns'
= head (1 :: uns') [por definição uns']
= 1                [por definição head]
```

Veja exemplo de avaliação Hope:

```
>: head uns';
>> 1 : num
```

9.4 - Programação modular

A avaliação preguiçosa permite separar o controle dos dados.

Para os exemplos considere a função **"take"** definida junto ao arquivo de módulo **"mylist"**:

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

Exemplo de separação do controle (pegue 2 itens) dos dados de uma lista infinita de uns:

```
take 2 uns'
= take 2 (1 :: uns')      [por definição uns']
= 1 :: (take 1 uns')      [por definição take]
= 1 :: (take 1 (1 :: uns')) [por definição uns']
= 1 :: (1 :: (take 0 uns')) [por definição take]
= 1 :: (1 :: [])          [por definição take]
= [1,1]                  [por notação de listas]
```

```
>: take (2, uns');
>> [1, 1] : list num
```

Conclusão das avaliações com estruturas infinitas

Exemplo de lista infinita sem conclusão explícita:

```
>: from 1;
>> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, ..., 140000] : list num
```

Exemplo de lista infinita com conclusão:

```
>: take (3, (from 1));
>> [1, 2, 3] : list num
```

Exemplo de lista limitada sem terminação:

```
>: filter ((=<3), (from 1));
>> [1, 2, 3] : list num
```

Exemplo de lista limitada sem terminação:

```
>: takeWhile ((=<3), (from 1));
>> [1, 2, 3] : list num
```

A peneira de Eratóstenes

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...
  3  5  7  9  11  13  15 ...
    5  7    11  13    ...
      7    11  13    ...
        11  13    ...
          13    ...
```

Definição:

```
dec peneira : list num -> list num;
--- peneira (p :: xs) <= p :: peneira (filter (\x => x mod p /= 0, xs));

dec primos2 : list num;
--- primos2 <= peneira (from 2);
```

Avaliação:

```
>: take (15, primos2);
>> [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47] : list num
```

Cálculo (nomes de funções reduzidos, omissão de "from"):

```
Primos2
= peneira (2..)
= peneira (2:: (3..))
=      (2:: (peneira (fltr (x mod 2 /= 0, (3..))
=      (2:: (peneira (3:: (fltr (x mod 2 /= 0, (4..))
=      (2::3:: (peneira (fltr (x mod 2 /= 0, (x mod 3 /= 0, (4..))
=      (2::3:: (peneira (5:: fltr (x mod 2 /= 0, (x mod 3 /= 0, (6..))
=      (2::3::5:: (peneira (5:: fltr (x mod 2 /= 0, (x mod 3 /= 0, (x mod 5 /= 0 (6..))
= ...
```

CAPÍTULO 10 - Módulos ou bibliotecas

Os módulos ou bibliotecas são em Hope um arquivo em formato texto com a extensão ".hop", o qual consiste em um conjunto de definições de funções. As funções podem ocorrer dentro do arquivo em qualquer ordem, mas os identificadores (os protótipos) devem ser declarados antes de serem usados. O ambiente Hope para ser operado necessita de um arquivo especial chamado "**Standard.hop**" que é carregado automaticamente quando o ambiente é executado. Este arquivo não deve ser modificado sob risco de prejudicar o funcionamento da linguagem (PATERSON, 2000 & BAYLEI, 1985).

10.1 - Estrutura de um módulo

O principal cuidado na elaboração de um arquivo de módulo é levar em consideração o estilo "**top-down**" na definição das funções. Por exemplo, a função "**isAlpha**" do arquivo de módulo "**mylist.hop**" depende da definição antecipada das funções "**isLower**" e "**isUpper**".

```
isAlpha : char -> truval;
isAlpha c <= isLower c or isUpper c;
```

Isso significa dizer que se as funções "**isLower**" e "**isUpper**" estiverem declaradas após "**isAlpha**" ocorrerá um erro no carregamento do arquivo. Imagine um arquivo de módulo chamado "**teste.hop**" contendo as funções como a seguir:

```
isAlpha : char -> truval;
isAlpha c <= isLower c or isUpper c;

isLower : char -> truval;
isLower c <= 'a' =< c and c =< 'z';

isUpper : char -> truval;
isUpper c <= 'A' =< c and c =< 'Z';
```

Na sequência execute a instrução dentro do ambiente "**uses teste1.hop;**" e observe o ocorrido:

```
>: uses teste1;
module teste1, line 2: semantic error - isLower: undefined variable
>:
```

Veja que a mensagem diz que o módulo (module) **teste1** contém um erro na linha "2", pois "**isLower**" é uma variável indefinida, ou seja, não existe antecipadamente na memória o recurso apontado. Observe que a ocorrência de erro parou no primeiro erro encontrado e nem se quer avaliou o restante da expressão da linha "**isAlpha c <= isLower c or isUpper c;**". Isso ocorre devido ao efeito de avaliação preguiçosa que a linguagem utiliza.

A ocorrência desse erro prova que a linguagem Hope segue o estilo "**top-down**" o que, infelizmente, pode limitar algumas tentativas de ações, em se tratando de linguagem funcional. No paradigma imperativo essa característica é muito apreciada, mas em uma linguagem funcional isso pode ser inconveniente.

Se a função "**isLower**" estiver a frente da função "**isAlpha**" e a função "**isUpper**" não estiver a tentativa de carregar o arquivo de módulo na memória apresentará o erro apontando "**isUpper**". Veja esse efeito sobre um arquivo de módulo chamado "**teste1.hop**":

```
isLower : char -> truval;
isLower c <= 'a' =< c and c =< 'z';

isAlpha : char -> truval;
isAlpha c <= isLower c or isUpper c;

isUpper : char -> truval;
isUpper c <= 'A' =< c and c =< 'Z';
```

```
>: uses teste2;
module teste2, line 5: semantic error - isUpper: undefined variable
>:
```

Veja que o erro foi refletido junto a função "isUpper", agora linha "5" que está definida após a função "isAlpha". A comprovação sobre o aspecto "top-down" pode ser verificado a partir de uma estrutura para um arquivo de módulo "teste3.hop":

```
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;
```

```
>: uses teste3;

>: isUpper 'x';
>> false : truval

>: isUpper 'X';
>> true : truval
```

Outro detalhe, é o uso das cláusulas "dec" para declarar uma função e "---" para definir uma função. Essas cláusulas são opcionais na declaração de funções simples, mas obrigatórias na declaração de funções múltiplas, definida na mesma linha. Veja exemplo para o arquivo de módulo "teste4.hop":

```
dec isLower, isUpper, isAlpha : char -> truval;
--- isLower c <= 'a' =< c and c =< 'z';
--- isUpper c <= 'A' =< c and c =< 'Z';
--- isAlpha c <= isLower c or isUpper c;
```

A declaração múltipla de funções é possível nos casos que as funções declaradas possuam a mesma assinatura.

Veja este exemplo para o arquivo de módulo "teste4.hop":

```
dec isLower, isUpper, isAlpha : char -> truval;
--- isAlpha c <= isLower c or isUpper c;
--- isLower c <= 'a' =< c and c =< 'z';
--- isUpper c <= 'A' =< c and c =< 'Z';
```


Observe que ao ser chamado o arquivo de módulo "uses teste5;" não ocorrerá erro de execução. Isso acontece pelo fato da linha de declaração das funções manter a ordem de prioridade entre elas. Uma vez, declarada a ordem antecipada não importa a ordem das definições. As declarações é que devem estar organizadas.

Outra forma válida é a estrutura do arquivo de módulo "teste6.hop" seguinte:

```
dec isLower : char -> truval;
dec isUpper : char -> truval;
dec isAlpha : char -> truval;

--- isAlpha c <= isLower c or isUpper c;
--- isLower c <= 'a' =< c and c =< 'z';
--- isUpper c <= 'A' =< c and c =< 'Z';
```

Esse é um estilo comum encontrado em algumas definições de módulos em Hope que poderá ser expresso como a estrutura para o arquivo de teste7.hop":

```
isLower : char -> truval;
isUpper : char -> truval;
isAlpha : char -> truval;

isAlpha c <= isLower c or isUpper c;
isLower c <= 'a' =< c and c =< 'z';
isUpper c <= 'A' =< c and c =< 'Z';
```

```
>: uses teste7;

>: isAlpha 'a';
>> true : truval

>: isAlpha 'A';
>> true : truval

>: isAlpha '1';
>> false : truval
```

Uma das formas de se fazer a definição de arquivos de módulos é abrir um editor de textos simples e escrever o conteúdo desejado salvo o conteúdo com a extensão ".hop". Outra forma é usar o próprio ambiente para gravar as funções que estejam em memória. Assim sendo, entre os seguintes códigos;

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

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

>: multiplicacao : num # num -> num;
>: multiplicacao (x, y) <= x * y;

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

A figura 10.1, mostra como poderá estar sua tela neste momento após a execução das etapas anteriores.

```

>> uses teste1;
module teste1, line 2: semantic error - isLower: undefined variable
>> uses teste2;
module teste2, line 5: semantic error - isUpper: undefined variable
>> uses teste3;
>> isUpper 'x';
>> false : truval
>> isUpper 'X';
>> true : truval
>> uses teste4;
>> uses teste5;
>> uses teste6;
>> uses teste7;
>> isAlpha 'a';
>> true : truval
>> isAlpha 'A';
>> true : truval
>> isAlpha '1';
>> false : truval
>> soma : num # num -> num;
>> soma (x, y) <= x + y;
>> subtracao : num # num -> num;
>> subtracao (x, y) <= x - y;
>> multiplicacao : num # num -> num;
>> multiplicacao (x, y) <= x * y;
>> divisao : num # num -> num;
>> divisao (x, y) <= x / y;
>>
    
```

Figura 10.1 - Estado das execuções efetivadas.

Se executado "**display**" serão apresentados os detalhes armazenados em memória como indica a figura 10.2.

```

>> true : truval
>> isAlpha 'A';
>> true : truval
>> isAlpha '1';
>> false : truval
>> soma : num # num -> num;
>> soma (x, y) <= x + y;
>> subtracao : num # num -> num;
>> subtracao (x, y) <= x - y;
>> multiplicacao : num # num -> num;
>> multiplicacao (x, y) <= x * y;
>> divisao : num # num -> num;
>> divisao (x, y) <= x / y;
>> display;
uses teste1, teste2, teste3, teste4, teste5, teste6, teste7;

dec soma : num # num -> num;
--- soma (x, y) <= x + y;

dec subtracao : num # num -> num;
--- subtracao (x, y) <= x - y;

dec multiplicacao : num # num -> num;
--- multiplicacao (x, y) <= x * y;

dec divisao : num # num -> num;
--- divisao (x, y) <= x / y;
>>
    
```

Figura 10.2 - Estado dos dados na memória.

Observe que tudo o que foi informado no ambiente ficou registrado exceto o conteúdo carregado dos arquivos de módulo. Neste momento, execute a seguinte instrução:

```
| >> save teste8;
```

Encerre o ambiente Hope e abra o arquivo "**teste8.hop**" em um editor de textos de sua preferência. Retire da primeira linha os indicativos de "**teste1**" e "**teste2**", como apresenta a figura 10.3. Grave a alteração.

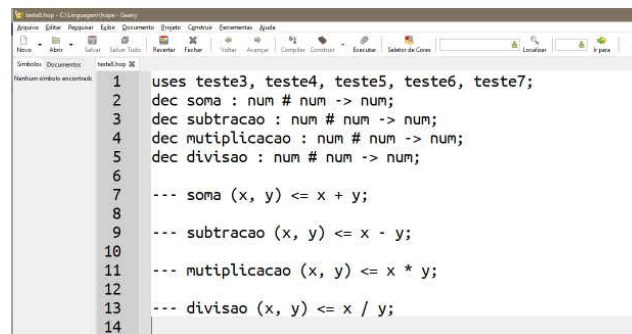


Figura 10.3 - Conteúdo do arquivo de módulo alterado.

Veja na figura 10.3 como o arquivo foi organizado pelo próprio ambiente. Os cabeçalhos das funções foram colocados abaixo da cláusula "**uses**" e o código de cada função é colocado após as declarações das funções. Veja que as funções foram informadas anteriormente sem o uso de "**dec**" e "**---**". No entanto, essas cláusulas foram inseridas, ou seja, não são tão opcionais quanto parecem.

Carregue novamente o ambiente Hope e execute a instrução "**uses teste8;**". Se tudo estiver em ordem nenhum erro será gerado. Neste momento, execute o comando "**display**" e observe que a única instrução na memória é "**uses teste8;**".

Apesar do conteúdo do arquivo de módulo "**teste8**" não ser apresentado este está armazenado na memória. Assim sendo execute os seguintes teste:

```
>: soma (2, 3);
>> 5 : num

>: subtracao (2, 3);
>> -1 : num

>: multiplicacao (2,3);
>> 6 : num

>: divisao (2, 3);
>> 0.6666666666666667 : num
```

Agora lembre-se de que o arquivo de módulo "**teste8**" faz a chamada de outros arquivos de módulo. Isso significa que as funcionalidades "**isAlpha**", "**isLower**" e "**isUpper**" também estão disponíveis.

```
>: isAlpha 'x';
>> true : truval

>: isAlpha 'A';
>> true : truval

>: isAlpha '9';
>> false : truval
```

Veja que a partir desse efeito é possível criar uma estrutura de módulos com diversos níveis de encaimento.

10.2 - Definição de prioridade de funções

Um importante detalhe na programação funcional é que qualquer elemento operacionalizado seja, uma variável, uma constante, um dado como valor e mesmo um operador (aritmético, relacional ou lógico) é considerado uma função. Na programação funcional, tudo é função. Assim sendo, considere a função **"somax"** que tem por objetivo calcular e apresentar o resultado da soma de dois valores numéricos. A função **"somax"** deverá ser usada como um operador aritmético e não como uma função em sua forma tradicional, ou seja, deverá ser usada como **"<valor1> somax <valor2>";**. Para tanto, é necessário fazer uso da cláusula **"infix"** como a seguir:

```
infix somax : 5;
somax : num # num -> num;
(somax) (x, y) <= x + y;
```

A cláusula **"infix"** faz com que uma função que opere com dois argumentos seja usada como operador aritmético binário infix associando sua ação com o dado a sua esquerda. O valor **"5"** indica o nível de prioridade do operador, sendo permitidos o uso de valores entre **"1"** e **"9"**. Um número maior significa maior prioridade na operação. Além da cláusula **"infix"** tem-se a cláusula **"infixr"** que permite declarar certo operador infix à direita. O objetivo desses recursos é tornar o uso simplificado.

Para dar ideia dos níveis de prioridade disponíveis veja a indicação dos operadores aritméticos, relacionais e lógicos definidos no arquivo de módulo **"Standard.hop"**. Quando uma função não tem seu nível de prioridade estabelecido está é automaticamente considerada com o nível mais alto possível.

PRIORIDADE	OPERADOR
1	or
2	and
3	=, /=
4	=<, <, >, >=
5	+
6	*, /, div, mod
7	
8	
9	
-	not

As funções como operadores podem ser definidas tanto com **"infix"** quanto com **"infixr"**. A diferença entre essas declarações afeta apenas a forma como o recurso é analisado e impresso: se um identificador **"@"** for declarado como infix à esquerda a definição **"@(a , b)"** é aplicada como **"a @ b"** priorizando a ação **"a (@ b)"**, caso seja infix à direita a operação **"a @ b"** priorizará ação **"(a @) b"**.

Para fazer referência ao operador **"@"** use **"(@)"** como ocorre em **"(somax) (x, y) <= x + y;"**.

Falta em Hope um operador que facilite a operação de potência. Todas as potências quando necessárias são calculadas com o uso da função **"pow"**. Assim sendo considere a definição da função **"**"**:

```
infixr ** : 7;
** : num # num -> num;
(**) <= pow;
```

Observe o resultado da operação:

```

>: pow (2,3);
>> 8 : num

>: 2 ** 3;
>> 8 : num

```

Grave o conteúdo da memória com o nome "**teste9**" a partir da ação "**save teste9;**".

Saia do ambiente e visualize o conteúdo do arquivo de módulo "**teste9.hop**". Você deverá estar vendo algo semelhante a imagem da figura 10.4.

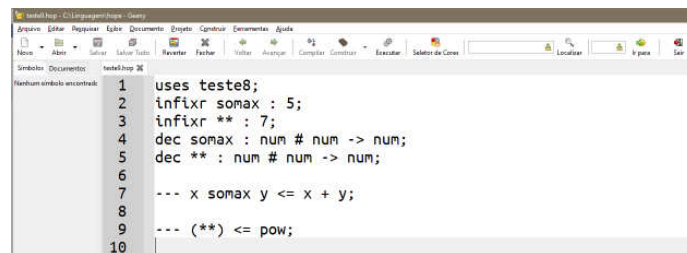


Figura 10.4 - Conteúdo do arquivo de módulo "teste9.hop".

Abra novamente o ambiente e execute "**uses teste9;**" e execute as ações a seguir e todas deverão funcionar pois o arquivo de módulo "**teste9**" chama o arquivo de módulo "**teste8**" que por sua vez chama os arquivos de módulo de "**teste3**" a "**teste7**".

```

>: isAlpha 'A';
>> true : truval

>: soma (3, 4);
>> 7 : num

>: pow (2, 3);
>> 8 : num

>: 2 ** 3;
>> 8 : num

```

Antes do próximo teste encerre a execução do ambiente Hope.

10.3 - Encapsulamento de funções

O efeito de encapsulamento de funções em Hope garante que se faça a separação de uso entre funções públicas e privadas. Este efeito é definido a partir do uso da cláusula "**private**" que garante que todas as definições subsequentes do módulo em uso sejam ocultadas de outros módulos que venham a utilizar esses recursos.

Para efetivar realizar um teste considere a declaração e definição de uma função interface denominada "**fact**" que calcule o fatorial de um valor numérico qualquer a partir da chamada de outra função, neste caso, definida sob a ótica da recursividade de cauda chamada "**factBase**".

Observe os detalhes nas definições de funções que são operacionalizadas com base em encapsulamento público e privado. Assim sendo, crie em um editor de texto de sua preferência um arquivo de módulo, gravando-o no local padrão de seu sistema operacional com o nome "**testeFact.hop**". Para maiores detalhes consulte o Apêndice C.

```

!! Demonstração de encapsulamento
!! Linguagem Hope - Capítulo 10

!  Arquivo de módulo: testeFact
!  =====

!  Seção: pública (modo padrão)
!  -----

    fact : num -> num;

!  Seção: privada
!  -----

    private;

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

    fact n <= factBase (n, 1);
    
```

O uso do símbolo "!" permite definir no arquivo linhas de comentários como elementos de documentação de código.

Abra o ambiente Hope, execute a instrução **"uses testeFact;"** e na sequência efetue os testes:

```

>: fact 5;
>> 120 : num

>: factBase (5, 1);
semantic error - factBase: undefined variable
>:
    
```

Veja que o conteúdo deixado como público é acessado normalmente sem nenhum problema. Em contrapartida, o conteúdo definido como privado não pode ser usado, é como se ele não existisse na memória. O conteúdo privado somente é acessado a partir do uso de alguma função interface que obviamente tenha a indicação de sua declaração em modo público e sua definição em modo privado.

Apêndice A - 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 que estendeu ao desenvolvimento do restante do material incluindo este livro que se apresenta.

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.

Comunicação em 11/04/2021

Assunto: Libro "Progame em Hope"

Buenos días, José A. Jiménez.
Espero encontrarte bien, al igual que el tuyo.

Continúa para su agradecimiento y dada la autorización otorgada el texto del libro "Progame em Hope", también adaptado al lenguaje Hope.

Al igual que el anterior, cuando esté listo, registraré el ISBN y otros trámites.

Un gran abrazo.
Augusto.

Concordância em 12/04/2021

Buenos días Augusto

Me alegro de ver cómo has continuado con la adaptación de mis libros de Haskell a Hope y, por supuesto, cuentas con mi autorización.

Empezaré a revisar detenidamente el libro "Programme en Hope".

Un abrazo,

José A. Alonso

Apêndice B - Método de Pólya para resolução de problemas

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?

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

Apêndice C - 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 destinada 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 sequência 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 interpretador estendido *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 vínculo (*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 vínculo (*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 janela de "**Prompt de comando**", entrar no diretório (pasta) da linguagem e efetuar a chamada do arquivo executável.

Apêndice D - 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

<code>x - y</code>	é a subtração de "x" com "y"
<code>x * y</code>	é a multiplicação de "x" com "y"
<code>x / y</code>	é a divisão com quociente real de "x" sobre "y"
<code>x + y</code>	é a soma de "x" com "y"
<code>x div y</code>	é a divisão com quociente inteiro de "x" sobre "y"
<code>x mod y</code>	é o resto da divisão com quociente inteiro de "x" sobre "y"

Funções aritméticas

<code>abs(x)</code>	é o valor absoluto (sempre positivo) de "x"
<code>acos(x)</code>	é o arco cosseno de "x"
<code>acosh(x)</code>	é o arco cosseno hiperbólico de "x"
<code>asin(x)</code>	é o arco seno de "x"
<code>asinh(x)</code>	é o arco seno hiperbólico de "x"
<code>atan(x)</code>	é o arco tangente de "x"
<code>atan2(x)</code>	é o arco tangente do coeficiente dos argumentos "x" e "y"
<code>atanh(x)</code>	é o arco tangente hiperbólico de "x"
<code>ceil(x)</code>	é o arredondamentode "x" para o próximo inteiro acima
<code>cos(x)</code>	é o cosseno de "x"
<code>cosh(x)</code>	é o cosseno hiperbólico de "x"
<code>exp(x)</code>	é o exponencial natural de "x"
<code>floor(x)</code>	é o arredondamentode "x" para o próximo inteiro abaixo
<code>hypot(x,y)</code>	é o comprimento da hipotenusa de um triângulo retângulo de "x" e "y"
<code>log(x)</code>	é o logaritmo natural de "x" na base "e"
<code>log10(x)</code>	é o logaritmo natural de "x" na base 10
<code>pow(x, y)</code>	é a potência de "x" elevado a "y"
<code>sin(x)</code>	é o seno de "x"
<code>sinh(x)</code>	é o seno hiperbólico de "x"
<code>sqrt(x)</code>	é a raiz quadrada de "x"
<code>tan(x)</code>	é a tangente de "x"
<code>tanh(x)</code>	é a tangente hiperbólico de "x"

Operações relacionais

<code>x = y</code>	verifica se "x" é igual a que "y"
<code>x /= y</code>	verifica se "x" diferente de "y"
<code>x > y</code>	verifica se "x" é maior que "y"
<code>x < y</code>	verifica se "x" é menor que "y"
<code>x >= y</code>	verifica se "x" é maior ou igual a "y"
<code>x <= y</code>	verifica se "x" igual ou menor que "y"

Operações lógicas

<code>x and y</code>	ação de conjunção entre "x" e "y"
<code>x or y</code>	ação de disjunção inclusiva entre "x" e "y"
<code>not x</code>	ação de negação de "x"

Operações com listas

<code>xs <> ys</code>	concatenação das listas "xs" e "ys"
<code>x :: xs</code>	é uma lista formada pela cabeça "x" e cauda "xs"

Operações de conversão de dados

<code>ord('c')</code>	retorna o código ASCII do caractere "c" informado
<code>chr(n)</code>	retorna o caractere do código ASCII numérico "n" informado
<code>num2str(n)</code>	retorna como cadeia um valor numérico "n"
<code>str2num("n")</code>	retorna como número o conteúdo numérico na forma de cadeia em "n"

Operações diversas

<code>id(x)</code>	mostra o identificador de tipo do conteúdo "x" informado
<code>succ(x)</code>	mostra o valor sucessor de "x"
<code>0 - n</code>	definição do número "n" como negativo

Apêndice E - Palavras reservadas Hope

Neste apêndice são apresentadas em ordem alfabética os identificadores chave que formam a estrutura da linguagem Hope (ressaltando que não foram vistos neste livro todos esses comandos, pois isso é assunto para outro momento). É importante ressaltar que nenhuma função pode ser definida utilizando-se esses nomes.

Palavras chave (geral):

Abstype	declaração abstrata de tipo para uso posterior com "type/data";
and	operação lógica de conjunção;
char	tipo de dado caractere;
data	declara tipo de dado derivado;
dec	declaração de função;
display	mostra definições existentes na seção atual de trabalho;
div	operação aritmética de divisão (quociente inteiro);
edit	edita módulo com editor padrão (disponível apenas no padrão POSIX);
else	complemento ao comando "if" para ação falsa;
error	mensagem de erro pelo usuário;
exit	saída do ambiente interativo;
help	modo ajuda (disponível apenas em Windows);
id	identifica tipo de dado de um elemento;
if	estabelece decisão a ser tomada;
in	complemento aos comandos "let" e "letrec";
infix	precedência de operador esquerdo;
infixr	precedência de operador direito;
lambda	função anônima;
let	simplificação de expressões;
letrec	simplificação de expressões (restritivo);
list	tipo de dado lista;
mod	operação aritmética de divisão (resto inteiro);
mu	simplificação de tipos de dados recursivos;
not	operação lógica de negação;
num	tipo de dado número;
or	operação lógica de disjunção;
private	indica componentes privados em um módulo;
read	carrega e mostra o conteúdo de arquivo externo;
save	grava a seção atual em um arquivo de módulo;
then	complemento ao comando "if" para ação verdadeira
truval	tipo de dado lógico (booleano);
tuple	tipo de dado tupla;
type	sinônimo para um tipo de dado específico;
typevar	tipo variável de dado para uso com "dec";
uses	carrega um arquivo de módulo externo;
where	simplificação alternativa de expressões;
whererec	simplificação alternativa de expressões (restritivo);
write	mostra os dados de uma expressão.

Identificadores reservados para compatibilidade entre diferentes implementações Hope:

end	finaliza definição de módulos;
module	define módulos de operação;
nonop	trata como função operadores aritmético/lógico;
pubconst	define constante dentro de módulo;
pubfun	define função dentro de módulo;
pubtype	define tipo de dado dentro de módulo.

Sinônimos:

\	substituto de "lambda";
use	substituto de "uses";
infixrl	substituto de "infixr".

Operadores:

---	definição de função;
-	operação aritmética de subtração (inteiro/real);
!	estabelece linha de comentário em arquivo de módulo;
#	indica a separação de argumentos na forma de tuplas;
()	identifica tuplas ou argumento de funções;
*	operação aritmética de multiplicação (inteiro/real);
/	operação aritmética de divisão (real);
/=	operação relacional (diferente de);
:	defini a assinatura de declaração de uma função, após declaração;
::	efetiva a construção de listas;
;	finaliza linha de instrução ou comando;
[]	constitui listas;
	defini a continuidade de expressão lambda;
+	operação aritmética de adição (inteiro/real);
++	estabelece a atribuição de construtores de tipos definidos;
<	operação relacional (menor que);
<=	defini ação de asserção em funções;
=	operação relacional (igual a);
=<	operação relacional (igual a ou menor que - menor ou igual a);
==	defini tipo de dado em expressões qualificadas;
=>	defini a ação de uma função lambda;
>	operação relacional (maior que);
->	defini ações de saída ou curificação de funções;
>=	operação relacional (maior ou igual a).


```

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);
```

ANOTAÇÕES

Referências bibliográficas

ALONSO JIMÉNEZ, José A. Temas de "Programación funcional": curso 2016-27. Sevilla: Universidad de Sevilla, 2016. Disponível em: <https://www.cs.us.es/~jalonso/publicaciones/2017-Temas-I1M-016-17.pdf>. Acesso em: 30 mar.2021.

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.

BACKUS, J. **Can programming be liberated from the von Neumann style? A functional style and its algebra of programs**. CACM, v. 21, n. 8, pp. 613–641, aug 1978.

_____. **The history of FORTRAN I, II, and III**. IEEE Annals of the History of Computing, v. 20, n. 4, 1998.

BARENDREGT, H., BARENDSEN, E. **Introduction to Lambda Calculus**. 2000. Göteborg: Göteborgs universitet. Disponível em: <http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf>. Acesso em 30 mar. 2021.

BIRD, R., WADLER, P. **Introduction to Functional Programming**. Hemel Hempstead: Prentice Hall Internacional (UK) Ltd., 1988.

BURSTALL, R. M., MACQUEEN, D. B. & SANNILA D. T. **HOPE: An experimental applicative language**. Dissertação em ciência da computação - Universidade de Edimburgo. Escócia, p. 136-146. 1978.

DIAS, M. **A teoria dos conjuntos e representação: resumo de Matemática**. Blog do ENEM, 2020. Disponível em: <https://blogdoenem.com.br/teoria-dos-conjuntos-enem-vestibular/>. Acesso em 6 abr. 2021.

DPL. **Hope**. Dictionary of Programming Languages, 2000. Disponível em: http://cgibin.erols.com/zing/cgi-bin/cep/cep.pl?_key=Hope. Acesso em 5 mar. 2021.

FANCHER, D. **Book of F#: Breaking free with managed functional programming**. San Francisco: No Starch Press, 2014.

FURRUGIA, A. **Combinatory logic: From philosophy and mathematics to computer science**. Msida: University of Malta, 2021. Disponível em: <https://www.um.edu.mt/library/oar/bitstream/123456789/38118/1/Alexander%20Farrugia.pdf>. Acesso em: 30 mar. 2021.

GABRIËLS, R., GERRITS, D., KOOIJMANS, P. **John W. Backus**. Eindhoven: Faculteit Wiskunde & Informatica Technische Universiteit Eindhoven, 2007.

GEH. **Avaliação preguiçosa: Paradigmas de programação**. Grupo de Estudos em Haskell. Disponível em: <https://haskell.pesquisa.ufabc.edu.br/posts/haskell/12.laziness.html>. Acesso em 9 abr. 2021.

HOFSTRA, P., SCOTT, P. **Aspects of categorical recursion theory**. New York: Cornell University, 2020. Disponível em: <https://arxiv.org/pdf/2001.05778.pdf>. Acesso 5 abr. 2021.

HOWE, D. **NPL**. FOLDOC, 1985. Disponível em: <https://foldoc.org/NPL>. Acesso em 30 mar. 2021.
INRIA. **A History of Caml**. Then Caml Language, 2005. Disponível em: <https://caml.inria.fr/about/history.en.html>. Acesso em 30 mar. 2021.

_____. **Temas de "programación funcional"**. Sevilla: Universidad de Sevilla - Depto de Ciencias de la Computación e Inteligencia Artificial, 2019. Disponível em: <<https://www.cs.us.es/~jalonso/cur-sos/i1m/temas/2019-20-I1M-temas-PF.pdf>>. Acesso em 30 mar. 2021.

HUTTON, G. **Programming in Haskell**. Cambridge: Cambridge University Press, 2007.

_____; Dobrado, 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.

LANDIN, P. **The next 700 programming languages**. Communications of the ACM., v. 9, n. 3. mar. 1966. Disponível em: <<https://www.cs.cmu.edu/~crary/819-f09/Landin66.pdf>>. Acesso 5 abr. 2021.

LÉVÉNEZ, É. **Computer Languages History**. Personal site. Disponível em: <<https://www.levenez.com/lang/>>. Acesso em 30 mar. 2021.

MACQUEEN, D., HARPER, R., REPPY, J. **The History of Standard ML**. ACM Program. Lang., v. 4, n. HOPL, 86. jun. 2020. Disponível em: <<https://dl.acm.org/doi/pdf/10.1145/3386336>>. Acesso em 30 mar. 2021.

MC-CARTHY, J. **History of LISP**. ACM SIGPLAN Notices, aug. 1978.

MICHAELSON, G. **An introduction to functional programming through lambda calculus**. New York: Dover Publications, Inc. 2011.

MILNER, R. **A Theory of Type Polymorphism in Programming**. Journal of Computer and System Sciences, v. 17, pp. 348-375, 1978.

O'SULLIVAN B., STEWART D., GOERZEN J. **Real World Haskell**. California: O'Reilly, 2008.

OWE, O., KROGDAHL, S., LYCHE, T. **From Object-Orientation to formal methods: Essays in memory of Ole-Johan Dahl**. New York: Springer, 2004.

PATERSON, R. A HOPE interpreter: Reference. 2000. Disponível em: <http://www.stolyarov.info/static/hope/ref_man.pdf>. Acesso em: 5 mar. 2021.

RENDELL, P. **Turing machine universality of the game of life**. New York: Springer, 2016.

RUIZ, B. C., GUTIÉRREZ, F., P. GUERRERO, GALLARDO, J. E. **Razonando con Haskell**. México: Thompson, 2004.

SMITH, J. B. **Practical OCaml**. New York: Apress, 2006.

SOMMERVILLE, J. F. **An experiment in high-level microprogramming**. St Andrews: University of St Andrews, 1977. Disponível em: <<https://research-repository.st-andrews.ac.uk/handle/10023/13423>>. Acesso em 30 mar. 2021.

SURHONE, L. M. **Tag system**. Mauritius: Betascript Publishing, 2010.

THE ELIXIR TEAM. **Elixir v1.0 released**. Elixir-lang.org, 2014. Disponível em: <<https://elixir-lang.org/blog/2014/09/18/elixir-v1-0-0-released/>>. Acesso em 30 mar. 2021.

THOMPSON, S. **Haskell: The Craft of Functional Programming**. 2. ed. Boston: Addison-Wesley, 1999.

ZACH, R. **Kurt Gödel and Computability Theory**. Calgary: Department of Philosophy - University of Calgary, 2007. Disponível em: <<https://people.ucalgary.ca/~rzach/static/cie-zach.pdf>>. Acesso em 30 mar. 2021.

Índice remissivo

A

abs.....	33
adição	19
alpha	17, 31, 64
and	34
any	54
árvore binária	69
ASCII.....	60
asserção.....	13, 28, 96
aterramento (encerramento).....	45
avaliação	71
avaliação de funções.....	12

B

beta	31
bibliotecas	77
binário para decimal	59
bit	58, 59
byte	59

C

char	27, 28
chr	42
cifra de César.....	41
codificação binária	58
comentários	25, 26
composição	33, 58
compreensão de cadeias	40
concatenação	21, 47, 57, 94
condição	33, 45
conjuntos.....	10, 85
cons (construtores).....	35, 66
contexto computacional.....	23
contexto matemático	22
correspondência de padrões.....	10, 34
criptografia	41
currificação (currying).....	30

D

data	65, 67, 69, 70
dec	17, 78
decodificação	42
definição de tipos.....	64
definição do protótipo.....	28
deslocamento.....	43, 44
display.....	18, 80
div	19, 59
divisão.....	19
dobra à direita.....	55
dobra à esquerda	57
drop	22, 49, 50
dropWhile.....	55

E

edit.....	23, 24
encapsulamento de funções	83
equações	13
estilo de programação.....	27
estratégias de avaliação.....	71
estruturas infinitas.....	73
even	33
exit	18
expressões lambda	36

F

fácil manutenção.....	9
fact	83
false.....	54, 64

F

fatores.....	39
filter	39, 54, 75
final	85, 89
foldl	57, 58
foldr	55, 56, 59
from	72, 75
fst	34
função recursiva.....	45
funções internas (predefinidas)	20, 93

G

geradores.....	39
guardas	39

H

head	21, 35, 74
------------	------------

I

index	21
inferência de tipos	29
infinito.....	72
infix	25, 34, 45, 47, 82, 95, 98
infixr	25, 82, 95, 96
init	51
instalação.....	91
int2let.....	42, 43
isAlpha	77, 78
isDigit	33
isLower.....	77
isUpper	77, 78

J

just	66
------------	----

L

lambda	36
length	22, 31, 41
linhas de comentários	26
list	17, 27
list alpha	67
list bit	59, 61
list char	27, 63
lista de bits	58, 60
listas por compreensão	39

M

map	39, 54
mensagem criptografada	44
mod	19, 59
modo privado / modo público	84
módulos	77
mov	65
multiplicação	19
múltiplos argumentos	31
mylist.hop	11, 19, 33, 35, 97

N

nomes	25
not	34
nothing	66
null	35
num	17, 27, 28, 58
números binários	58

O

octeto	59
odd	54
operações aritméticas	20
ord	42

P

palavras-chave	25
polimorfismo	31
pow	82
pred	35
primo (primos)	40, 76
prioridade de funções	82
product	22

R

recursão	48
recursão múltipla	49
recursividade	45, 46
repeat	60
reverse	22, 41

S

save	18, 26, 80
signum	33
snd	34
soma	93
splitAt	33
Standard	33, 42, 64
string	27, 63
subtração	19
sufixo	25
sum	22, 56

T

tail	21, 35
take	21, 60, 75
takeWhile	55, 75
teoria de conjuntos	27
terminação	72
tipo recursivo	68
tipos básicos de dados	27
tipos de dados	9
tipos funções	28
tipos seguros	29
true	54, 64

T

truval	17, 27, 28, 31, 64
tuple	27
type	58, 63

U

uses	24, 81
------------	--------

W

write	18
-------------	----

Programe em Hope

LÓGICA DE PROGRAMAÇÃO FUNCIONAL

PROGRAMAÇÃO FUNCIONAL NA PRÁTICA

ESTE LIVRO APRESENTA A LINGUAGEM HOPE DE FORMA DINÂMICA E PRÁTICA A PARTIR DE DIVERSOS EXEMPLOS DE APLICAÇÃO DOS 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.

