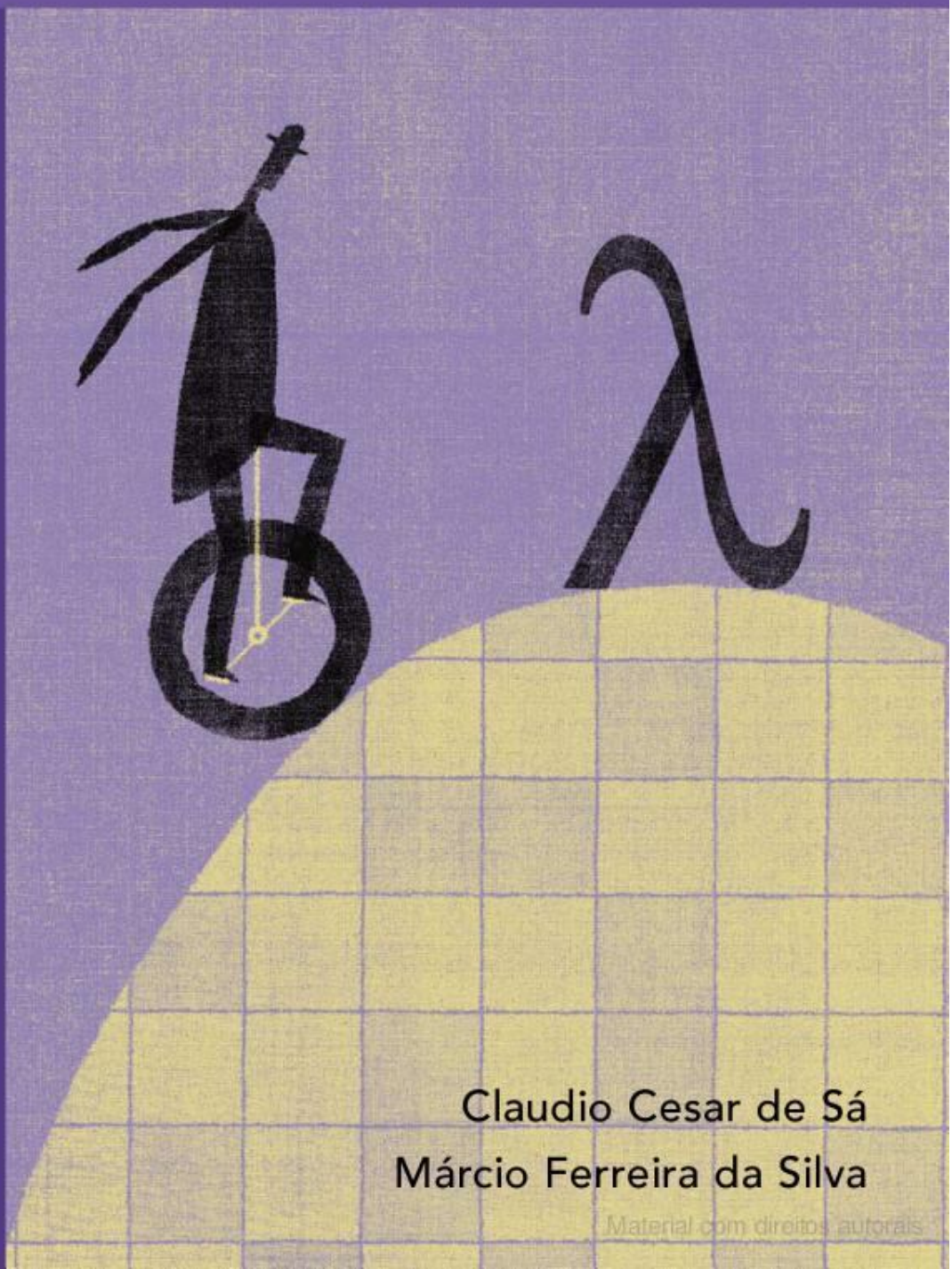
Haskell

Uma Abordagem Prática



Claudio Cesar de Sá
Márcio Ferreira da Silva

novatec

Sumário

1 Introdução **23**

1.1 O que é o Haskell? 23

1.1.1 Características do Haskell 24

1.2 Paradigmas de Linguagens de Programação 25

1.2.1 O Paradigma Funcional 26

1.2.2 Uma Visão Crítica 26

1.3 Histórico 27

1.4 Objetivos deste Livro 28

2 Matemática Funcional **29**

2.1 Iniciando em Haskell 29

2.2 Visualizando Funções em Blocos 32

2.3 Exercício Resolvido 34

2.4 Exercícios Propostos 37

3 Uma Visão Funcional da Indução **39**

3.1 Recursão 39

3.2 Exemplos 41

3.2.1 Fatorial 41

3.2.2 Fibonacci 42

3.3 Perigos da Recursão 43

3.4 Casamento de Padrões 45

3.5 Avaliação Preguiçosa 47

3.6 Exercícios Propostos 49

4 Tuplas **51**

4.1 Apresentação às Tuplas 51

4.2 Tipos Compostos com Tuplas 55

4.3 Exercício Resolvido 57

4.4 Exercícios Propostos 61

5 Listas **63**

5.1 Noções sobre Listas 63

5.2 Listas em Haskell 64

5.2.1 Exemplos 67

5.3 A Função Observe 72

5.4 Compreensão de Lista (*List Comprehension*) 74

5.5 Outros Exemplos 75

5.6 *Strings* 80

5.6.1 Conversões entre Tipos 81

5.6.2 Caracteres Especiais 81

5.7 Avaliação Preguiçosa em Listas 82

5.8 Exercícios Propostos 83

6 Tipos de Dados **87**

6.1 Tipos Básicos 87

6.1.1 Booleanos 89

6.1.2 Inteiros 90

6.1.3 Caracteres 93

6.1.4 Reais 96

6.2 Tipos Sinônimos 99

6.3 Tipos Estruturados 99

6.4 Operadores 101

6.5 Exercícios Propostos 102

7 Classes **105**

7.1 Polimorfismo . . . 105

7.1.1 Polimorfismo *ad hoc* . . . 106

7.2 Classes Derivadas . . . 109

7.3 Hierarquia de Classes e Tipos . . . 109

7.3.1 Classe Enum . . . 110

7.3.2 Classe Show . . . 111

7.3.3 Classe Num . . . 112

7.4 Mônadas . . . 115

7.4.1 Efeitos Colaterais . . . 116

7.4.2 Classe Monad . . . 117

7.5 Exercícios Propostos . . . 119

8 Lambda Cálculo **121**

8.1 Breve Histórico . . . 121

8.2 Fundamentação das Linguagens Funcionais . . . 123

8.3 Abstração . . . 124

8.4 Sintaxe . . . 127

8.4.1 Convenções na Sintaxe . . . 129

8.5 Variáveis Livres e Ligadas . . . 131

8.5.1 Variáveis Livres . . . 131

8.5.2 Variáveis Ligadas . . . 131

8.5.3 Complemento . . . 132

8.6 Regras de λ-redução (ou λ-conversão) . . . 132

8.6.1 α-redex (ou α-redução) . . . 133

8.6.2 β-redex (ou β-redução) . . . 135

8.6.3 η-redex (ou η-redução) . . . 136

8.7 Substituição . . . 137

8.7.1 Cuidados com a Substituição . . . 138

8.8 Lambda em Haskell . . . 138

8.8.1 Exemplo de Teste de Liberdade . . . 141

8.9 Considerações . . . 142

8.9.1 Escolha na Seqüência da Redução . . . 142

8.9.2 Analogias Computacionais 143
8.10 Exercícios Propostos 144

9 Generalização **147**
9.1 Funções Genéricas sobre Listas 147
9.1.1 Motivação ao Mapeamento 149
9.1.2 Motivação à Filtragem 149
9.1.3 Motivação à Redução 150
9.2 Funções de Alta-Ordem 151
9.2.1 Mapear 151
9.2.2 Filtrar 153
9.2.3 Reduzir 155
9.3 Análise 158
9.4 Exercícios Propostos 159

10 Figuras Textos **163**
10.1 Tipos e Funções 163
10.2 Funções sob Figuras de Caracteres 164
10.2.1 Considerações Iniciais 164
10.2.2 Manipulando Figuras 165
10.3 Função de Composição de Funções 175
10.3.1 Fundamentos 175
10.3.2 Exemplos 177
10.4 Resumo 180
10.5 Exercícios Propostos 181

11 Vetores **183**
11.1 Criação e Manipulação de Matrizes 183
11.2 Exemplos 186
11.2.1 Histograma 186
11.2.2 Determinante de uma Matriz 187
11.3 Exercícios Propostos 188

12 Árvores **191**

12.1 Noções sobre Árvores 191
12.2 Árvores Binárias 192
12.3 Exemplo 194
12.4 Exercícios Propostos 195

13 Algoritmos de Ordenação **197**

13.1 Introdução 197
13.2 Ordenação por Seleção 198
13.3 Ordenação por Inserção 199
13.4 *MergeSort* 201
13.5 *Quicksort* 202
13.6 Exercícios Propostos 203

14 Elementos Não-Funcionais **205**

14.1 Compilando Códigos-Fonte 205
14.1.1 Modo Interpretado 205
14.1.2 Modo Compilado 207
14.2 Expressões Condicionais e *Layout* 208
14.3 Blocos de Comandos 212
14.4 Listas de Ações 216
14.5 Comentários 217
14.5.1 Exemplo de Comentários 217
14.6 Blocos de Repetição 218
14.6.1 Exemplo 1 218
14.6.2 Exemplo 2 219
14.7 Módulos 220
14.8 Resumo do Capítulo 221

15 Entradas, Saídas, e Seqüências de Ações **223**

15.1 Funções de Escrita 223

15.2 Funções de Leitura 224

15.2.1 Exemplos 225

15.2.2 Um `while` 227

15.3 Arquivos 227

15.3.1 Operações com *handles* sobre Arquivos 228

15.3.2 Operações Diretas sobre Arquivos 230

15.4 Tratamento de Exceções 232

15.5 Exemplo 233

15.5.1 Execução do Código 238

15.6 Exercícios Propostos 239

16 Problemas **241**

16.1 Motivação aos Problemas da IA 241

16.2 Problema dos Vasos 242

16.2.1 Descrição do Problema 242

16.2.2 Implementação e Código 244

16.3 Quebra-Cabeça 248

16.3.1 Descrição 248

16.3.2 Implementação e Código 249

16.4 Sobre as Implementações 254

16.5 Exercícios Propostos 254

17 Reflexões Finais **255**

17.1 Com Direito à *Macarronada*? 255

17.2 Requisitos de um Programa 256

17.2.1 Requisitos para Avaliar um Programa 256

17.2.2 Programando com Elegância 259

17.3 O Mundo dos Blocos Funcionais 260

17.3.1 As Armadilhas 262

17.4 Epílogo 265

A Ambientes de Programação **267**

A.1 Interpretador Hugs 268

A.1.1 Repositório para *Download* 268

A.1.2 Procedimento de Instalação 268

A.1.3 Ambiente de Execução 268

A.1.4 Opções do Programa 269

A.1.5 Comandos do Hugs 269

A.1.6 Como Conseguir Ajuda 271

A.2 Compilador Haskell Glasgow 271

A.2.1 Repositório para *Download* 271

A.2.2 Procedimento de Instalação 271

A.2.3 Construindo um Executável 272

A.2.4 O Interpretador do GHC 273

A.2.5 Opções do GHCi 274

A.2.6 Como Conseguir Ajuda 274

A.3 Editores 274

B Funções Complementares **277**

Apresentação

A utilização de linguagens funcionais modernas no ensino de programação, em cursos de ciência da computação e similares, pode trazer grandes benefícios aos estudantes. Noções importantes, como indução matemática e recursividade, são introduzidas naturalmente. Tenho observado esses benefícios desde o ano de 2001, quando adotei o Haskell como linguagem básica para uma disciplina de programação funcional, em cursos de graduação.

Passei então a me corresponder com interessados na linguagem Haskell, trocando material. O professor Claudio Cesar de Sá foi uma das pessoas com quem me correspondo, desde 2002. É uma satisfação poder observar um dos resultados do seu trabalho, que se concretiza na obra *Haskell: uma abordagem prática*.

Tenho grande prazer de poder escrever a Apresentação deste livro, constatando que se trata de uma obra muito bem-formulada e adequada ao ensino de programação funcional. O texto introduz os conceitos de forma didática, permitindo que mesmo estudantes de áreas não diretamente ligadas à computação possam tirar proveito do material. Mas o livro, não se prende apenas aos conceitos elementares apresenta também as características mais avançadas de linguagens funcionais, adequadamente implementadas em Haskell, como a avaliação preguiçosa e as funções de alta-ordem. Pode ser considerada a obra mais completa relacionada à linguagem Haskell publicada na língua portuguesa até esta data. Recomendo sua utilização como livro-texto em disciplinas de programação funcional, tanto em cursos de graduação quanto de pós-graduação.

Vladimir Oliveira Di Iorio
Departamento de Informática
Universidade Federal de Viçosa - MG

Dedicatórias

À Maria Luiza, pelo equilíbrio próximo.
Aos nossos filhos, por torná-lo distante,
mas necessário, em qualquer processo.
Claudio Cesar de Sá

Aos meus pais pelo apoio irrestrito em
todos os momentos de minha vida.
À Priscila que soube tão bem compreender os meus
momentos de ausência em função deste trabalho.
Aos amigos, que sempre acreditaram no meu trabalho.
Márcio Ferreira da Silva

Tocando em Frente

Ando devagar porque já tive pressa
levo esse sorriso porque já chorei demais
Hoje me sinto mais forte, mais feliz quem sabe
Só levo a certeza de que muito pouco eu sei, eu nada sei

Conhecer as manhas e as manhãs
o sabor das massas e das maçãs
É preciso amor pra poder pulsar
É preciso paz pra poder sorrir
É preciso a chuva para florir
. . . . . .

Almir Sater

Prefácio

Uma das formas de pensar do ser humano é utilizando o conceito de *indução*. A indução visa analisar casos simples, para uma, duas, três ... instâncias, e tenta generalizá-las sobre uma *regra mais ampla* [1]. Essa regra torna-se um conceito abragente e generaliza as instâncias mais simples. Trata-se de uma atividade típica da inteligência humana. Neste sentido, há a idéia de se construir programas de computadores, em que uma regra conceitual é aplicada sucessivamente para os casos mais simples, até que o caso base retorne um dado conhecido. Este livro apresenta tal conceito aplicado a uma linguagem de computador.

A concepção deste livro foi incentivada pela ausência de um material didático destinado a profissionais que desejam aprender a programar desde um nível elementar até um nível intermediário. O presente texto é destinado a estudantes de cursos de programação, da área de computação e informática, e demais áreas das ciências exatas. Além de cientistas e profissionais interessados em aprender uma linguagem de programação revolucionária, embora seu embasamento matemático seja sustentado desde a década de 1930. Contudo, um dos pontos de partida da linguagem Haskell está nas funções matemáticas, ensinadas nas escolas de ensino fundamental e médio brasileiras. Esse fato incrementa a perspectiva de uso dessa linguagem.

Entre os diversos paradigmas de linguagens de programação mais conhecidos destacam-se o procedural ou seqüencial, a orientação a objetos, o lógico, o funcional, asregras de produção, etc. Sob o paradigma funcional há várias linguagens de programação. Dentre elas, o Haskell constitui-se em uma alternativa de concepções modernas e matematicamente fundamentadas [2].

A linguagem Haskell destaca-se pela sua clareza na codificação, reuso do código e pela pouca ou quase nenhuma exigência de conhecimento prévio de programação. Além disso, a linguagem é de código aberto, multiplataforma e em franca expansão de uso.

Atualmente, outros títulos sobre Haskell estão disponíveis apenas em língua inglesa, os quais iniciam com assuntos complexos, em que leigos em computação ficariam sem saber o que fazer nem por onde começar. Este livro visa preencher e contemplar a questão didática dos títulos existentes, principalmente no idioma inglês.

Como um dos atrativos da linguagem Haskell é seu rápido aprendizado [3], este livro objetiva ir ao encontro dessa premissa, ao contrário de outras linguagens de programação, nas quais uma quantidade substancial há de ser conhecida até que uma experimentação possa ser efetivada. Em Haskell, com poucos elementos sobre

a linguagem e conhecimentos de programação, um leigo potencializa e concretiza interessantes desenvolvimentos [4].

Estruturação do Livro

Este livro difere em parte de uma seqüência esperada dos capítulos de publicações sobre linguagens de programação, cujos conteúdos são apresentados de modo gradativo em uma complexidade crescente. Ao final da apresentação de um conceito, esse é exemplificado, discutida a sua abrangência e correlacionado com o próximo assunto a ser apresentado.

Haskell: uma abordagem prática contrapõe essa apresentação seqüencial e se movimenta em uma *espiral*. Essa *espiral* reflete os saltos para a frente e para trás que o texto segue, com o objetivo de não sobrecarregar o leitor com todos os detalhes do ponto apresentado. Assim, há muitas notas de rodapé, bem como indicações de pontos à frente, onde um dado assunto é detalhado. Ora, em um caso contrário, indicações de onde tal assunto já foi apresentado, e ali vale a lembrança na conexão com o referido tema.

Essa *espiral* reproduz a idéia do que é o Haskell: *uma linguagem declarativa e não-seqüencial*. Assim, o objetivo é destacar como o Haskell funciona em uma conexão direta com exemplos. Oportunamente, esta é a estratégia de apresentação dos assuntos: *suas definições são imediatamente seguidas por exemplos, e esses executados*. Essa tríade de uma explicação do conteúdo apresentado, sua implementação em linhas código e sua execução, é o argumento para justificar o título deste livro: *Haskell: uma abordagem prática*.

Em geral, tais exemplos são ora simples ora bizarros, mas com o objetivo de destacar um dado ponto. A natureza não-seqüencial do Haskell é propositadamente refletida neste livro, pois a proposta é tornar o texto o mais atraente possível para o leitor iniciante em programação, tornando-o apto, de imediato, a construir programas em Haskell.

A seguir apresenta-se um resumo dos conteúdos de cada capítulo:

- ✓ **Capítulo 1**: Apresenta a linguagem Haskell, seu contexto, histórico e motivações.
- ✓ **Capítulo 2**: Alguns exemplos da geometria espacial são implementados, a fim de que o leitor imediatamente experimente esta linguagem.
- ✓ **Capítulo 3**: O conceito *recursão* é ilustrada com exemplos, como sendo a base de todo e qualquer programa funcional.
- ✓ **Capítulo 4**: Uma estrutura chamada de *tuplas* vai permitir que uma função retorne mais de um valor encapsulado. Essa estrutura nativa do Haskell é particular em relação às demais linguagens imperativas.
- ✓ **Capítulo 5**: Mostra uma estrutura dinâmica de armazenamento: as *listas*. Tal estrutura é largamente usada ao longo do livro.
- ✓ **Capítulo 6**: Discute os tipos primários de dados do Haskell, embora seu uso tenha sido feito quando necessário em capítulos anteriores. O adiamento deste

assunto na seqüência do livro tem o objetivo de não sobrecarregar o leitor com detalhes que não afetem o aprendizado inicial desta linguagem. Contudo, a partir do Capítulo 2, este sexto pode ser lido em paralelo.

- ✓ **Capítulo 7**: A linguagem Haskell utiliza um esquema de dedução de tipos de dados, o qual é hierarquizado em um conceito de *classes* de tipos. Parte dessa hierarquia é estudada neste capítulo, haja vista que a mesma é definida a partir dos tipos primários.
- ✓ **Capítulo 8**: Apresenta o formalismo matemático do paradigma desta linguagem. Este capítulo demanda um conhecimento elementar de matemática.
- ✓ **Capítulo 9**: Discute a existência de três funções genéricas que, quando combinadas como argumentos em outras funções, ampliam o nível de abstração de programação. Isto conduz às *funções de alta-ordem*.
- ✓ **Capítulo 10**: Reescrevendo exemplos e resolvendo alguns exercícios de [5], aqui é ilustrado o uso das funções de alta-ordem aplicado a figuras bidimensionais em formato caracter.
- ✓ **Capítulo 11**: Esta linguagem suporta o conceito de *matrizes*. No caso, deve-se incluir uma *biblioteca* ou *módulo* específico para um acesso às funções prontas sobre matrizes.
- ✓ **Capítulo 12**: Apresenta o uso de estruturas mais sofisticadas como *árvores*, seguindo por exemplos introdutórios sobre o assunto.
- ✓ **Capítulo 13**: Alguns métodos clássicos de ordenação são implementados e discutidos em estruturas lineares do tipo listas.
- ✓ **Capítulo 14**: Como se utiliza um interpretador e um compilador Haskell e como o mesmo suporta as questões não-funcionais da linguagem. Alguns tópicos foram usados em capítulos anteriores, mas aqui está detalhado. O início deste capítulo deve ser consultado logo, em paralelo ao Capítulo 2.
- ✓ **Capítulo 15**: O suporte e a manipulação a *arquivos* do tipo textos, finalizando com um exemplo de um sistema de cadastro de pessoas.
- ✓ **Capítulo 16**: Dois problemas clássicos da inteligência artificial são discutidos e resolvidos, mostrando a versatilidade do Haskell como uma linguagem de propósitos gerais.
- ✓ **Capítulo 17**: Aqui, busca-se resgatar e sumarizar conhecimentos de experimentos sobre esta linguagem, como ferramenta de ensino para estudantes que estão se iniciando na área. Este capítulo enfatiza as dificuldades que muitos neófitos apresentam no aprendizado de uma linguagem de programação. Mais especificamente, a experiência reporta alguns semestres de ensino desta linguagem na graduação. Seus resultados podem ser encontrados em: `http://geocities.yahoo.com.br/lpg3udesc`. Há uma discussão sobre pontos do que é resolver um problema *versus* a atividade de programar.
- ✓ **Apêndice A**: Neste apêndice são discutidos dois interpretadores e um compilador para Haskell. Como instalar nos ambientes MS-Windows e Linux, como compilar, etc. Complementa o Capítulo 14.
- ✓ **Apêndice B**: Algumas funções sobre listas estão prontas na biblioteca-padrão do Haskell: a `Prelude.hs`. O uso das mesmas no modo interpretado é ilustrado. Este apêndice ter sido um guia útil aos estudantes. Contudo, encoraja-se

que, a cada função pronta existente, o leitor implemente a sua própria função equivalente.

A ordem dos capítulos busca apresentar o conhecimento focado na construção imediata de programas. Os detalhes, que eventualmente poderiam tornar-se confuso ao leitor, são referenciados. Em muitas partes do livro, alguns conceitos são revisitados, ora antecipados, mas esta repetição de conteúdo é proposital com a finalidade de tornar claro os detalhes fundamentais da linguagem. Praticamente todos conteúdos dos capítulos foram devidamente experimentados e testados em cursos sobre linguagens funcionais, procurando sempre diversificar a abordagem dos exercícios propostos.

Orientação para a Leitura do Livro

A proposta do livro é tornar o leitor apto e confortável em uma iniciação a desenvolver seus exemplos, sem precisar conhecer os detalhes (e são muitos!) que envolvem a linguagem Haskell. Essa é a razão de que alguns conteúdos precisaram ser explicados fora de sua ordem. Entretanto, estas *rápidas* antecipações foram devidamente referenciadas para que os leitores pudessem ir diretamente em frente, deslocando-se para outras partes do livro. Em sentido contrário, a referência a um exemplo anterior também é referenciado quando o mesmo tiver pertinência com o conteúdo apresentado.

Assim, uma idéia de como o livro pode ser lido é dirigida a perfis de leitores:

- ✓ Para os iniciantes em programação: leiam os capítulos na seqüência de apresentação. Contudo, capítulos 6 e 17 aconselha-se a leitura dos memos qualquer momento, ou em paralelo a partir do Capítulo 1. Opcionalmente, o Capítulo 8, que apresenta os principais formalismos sobre Cálculo Lambda (*Lambda Cauculus*) pode ser omitido.
- ✓ Para aqueles que já conhecem alguma linguagem de programação: leiam primeiramente os capítulos 1 e 17 para se contextualizarem sobre a linguagem Haskell. Em seguida, sigam a seqüência dos capítulos, omitindo as partes já conhecidas das outras linguagens. Os *saltos* para algum conteúdo em particular é incentivado sem muita perda de conteúdo. Eventualmente, o Capítulo 8 pode ser omitido.
- ✓ Para os matemáticos: leiam o Capítulo 1 para uma visão geral, e, em seguida, o Capítulo 8. Contudo, a destreza e a sofisticação de se programar em Haskell passa obrigatoriamente por este capítulo matemático. Opcionalmente, o Capítulo 17, para uma conclusão mais aprimorada sobre o texto.

Agradecimentos

Desde agosto de 2002, quando tomamos conhecimento da linguagem Haskell, muito se construiu e contraímos um débito de gratidão com várias pessoas. Possivelmente estamos sendo injustos com algumas que aqui não são citadas, mas, para elas também, fica o nosso profundo agradecimento.

- ✓ Inicialmente, a Michel Gagnon o qual nos apresentou e incentivou a usar a linguagem Haskell. Suas notas de aula, consolidadas em uma apostila [6], foram um marco inicial para construção deste livro.
- ✓ A Vladimir Oliveira Di Iorio (Universidade Federal de Viçosa - MG) que nos cedeu as transparências de seu curso sobre linguagens funcionais, a partir das quais adaptamos muitos exemplos para esta publicação.
- ✓ A Simon Thompson (University of Kent - UK), por ter autorizado o uso de suas notas de aula `http://www.haskell.org/haskellwiki/Education`, em que alguns exercícios foram resolvidos e são apresentados no Capítulo 10.
- ✓ A Hal Daumé III (`http://www.isi.edu/~hdaume/`) pela autorização de uso de seu tutorial de Haskell, *Yet Another Haskell Tutorial* [7]. Esse texto encontra-se em código LaTeX (`http://www.latex-project.org/`) e alguns exemplos e macros deste documento foram reusados.
- ✓ A Guilherme Bittencourt, pelas correções e sugestões sobre Cálculo Lambda (Capítulo 8). A Fernando Deeke Sasse pela paciência de uma primeira leitura, pelo menos até a metade do texto, e sugestões no estilo de redação.
- ✓ Aos monitores e voluntários da disciplina de Linguagem de Programação III, do curso de bachalerado em Ciência da Computação da Universidade do Estado de Santa Catarina (UDESC), em especial a Pedro Gabriel de Figueiredo Rosa e a Patrícia Retore, que sempre mantiveram a disciplina dentro de um clima produtivo com o uso de parte deste material.
- ✓ Ao Grupo de Computação Cognitiva Aplicada (`http://www2.joinville.udesc.br/~coca`) do Departamento de Ciência da Computação (DCC) da UDESC, pela atmosfera e pelo incentivo.
- ✓ Aos professores do Departamento de Ciência da Computação da UDESC.
- ✓ À empresa Softexpert pelo suporte oferecido ao Márcio, e à UDESC pelo apoio aos autores.

Claudio Cesar de Sá
Márcio Ferreira da Silva

Capítulo 1

Introdução

Este capítulo apresenta um contexto da linguagem Haskell. Trata-se de uma linguagem de programação para propósitos gerais, destinada a construir programas para computadores. Essa tarefa de geração de códigos pode ser realizada por qualquer linguagem de programação. Nesta visão, o Haskell é uma proposta real e estável. Objetiva-se aqui apresentar o Haskell, sua abrangência em relação às outras linguagens, suas finalidades, etc. Esses tópicos são cobertos em textos específicos tais como: [8, 9, 10, 2]. Um histórico da linguagem [11, 7] finaliza o capítulo.

1.1 O que é o Haskell?

O Haskell é uma linguagem de programação funcional com tipos de dados bem-definidos [12]. Os modelos para as estruturas de dados são baseados na Teoria dos Conjuntos, enquanto as operações são descritas com o uso de funções entre os vários tipos de conjuntos. Por exemplo, seja o conceito de domínio e imagem, mapeados por números inteiros e ilustrados na Figura 1.1.

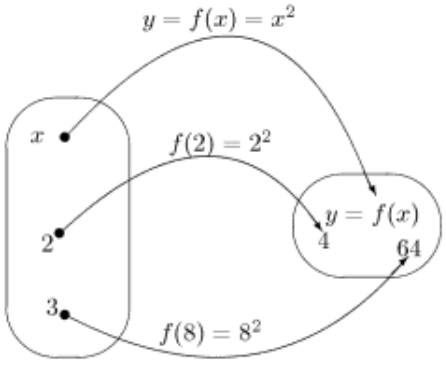


Figura 1.1 – Mapeamento domínio *versus* imagem.

■ OBSERVAÇÃO ■ A notação utilizada neste livro é padronizada pelo código-fonte do programa, salvo em um arquivo-texto qualquer, e em seguida a sua execução em um dos ambientes do Haskell em modo interpretado. O interpretador mais conhecido é o HUGS (Hugs e Winhugs), o qual está descrito na parte A.1 do Apêndice A. Contudo, outros interpretadores e compiladores para Haskell estão descritos na parte A.2 do Apêndice A. A operacionalidade de compilar e interpretar o código Haskell é também detalhada na seção 14.1. A codificação dos programas é feita em um editor de texto qualquer, orientado a caractere. Os requisitos e as vantagens desses editores também estão na parte A.3 do Apêndice A. Em um exemplo, edite e salve o código em um arquivo ASCII, tal como:

```
quadrado x = x*x
```

Carregue para execução no interpretador e avalie a função como:

```
Main> quadrado    2
4
```

A execução é chamada pelo nome da função e os valores de seus argumentos. O *prompt* dos interpretadores é dado pela seqüência `Main>` ou `Prelude>` , pontos esses detalhados nas seções 14.1, 14.2 e nas partes A.1, e A.2 do Apêndice A. Este pequeno exemplo, mostra a linguagem Haskell com atributos de legibilidade, rapidez e simplicidade para iniciantes em programação.

Segundo se observa, o código em Haskell é quase idêntico à função matemática. Os elementos comuns a serem identificados entre eles são ilustrados pela Figura 2.2.

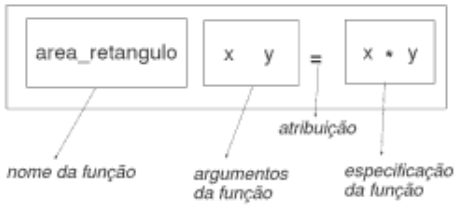


Figura 2.2 – Os elementos de uma função em Haskell.

A seguir, é apresentado o cálculo da área de um círculo que é dada pelo valor do π (pi) vezes o quadrado de seu raio:

$$A_{circulo}\ \ (r) = \pi.r^2$$

Logo, a área do círculo é uma função com o argumento r, representando o raio do círculo, multiplicado pelo valor de π (pi). O valor π, que é um número irracional, é igual a $3,14159...$ O valor do raio é dado pelo segmento de reta $\overline{AB}$, ou seja, o módulo da distância entre os pontos **A** e **B**. Para tanto, é necessário que sejam fornecidos os valores de **A** e de **B**. Simplificando-se, é utilizado o valor do raio diretamente, visto que a distância entre os dois pontos é também exemplificada neste capítulo.

Figura 2.3 – Um círculo e seu raio.

Em Haskell, este conceito matemático é representado por duas funções:

```
pi = 3,14159
area_circulo r = pi * r * r
```

Execução:

```
Main> area_circulo 3
28,27431
```

A primeira função é a **pi**, que não apresenta nenhum argumento, somente retorna o seu valor definido 3, 14159. A função `area_circulo` é definida com o argumento `r`, e utiliza o valor da função `pi` para calcular a área do círculo.

Em outro exemplo, o volume de um paralelepípedo onde há de três argumentos passados como parâmetro para uma única função. Tal exemplo é dado pela Figura 2.4.

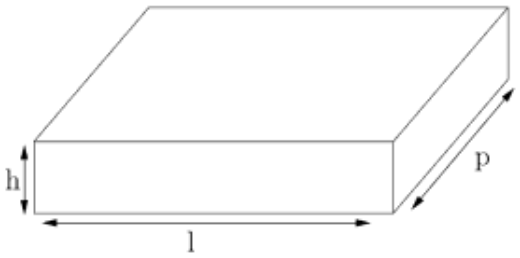


Figura 2.4 – Medidas de um paralelepípedo.

O volume paralelepípedo, como mostra a Figura 2.4, é descrito pela função matemática:

$$V\_paralelepipedo \;\; (h, l, p) = h.l.p$$

O volume desse paralelepípedo é dado pelo produto da base (l), largura (p) e altura (h). Em Haskell, essa notação matemática é representada por:

```
volume_paralelepipedo h l p = h * l * p
```

Execução:

```
Main> volume_paralelepipedo 3 5 2
30
```

Os exemplos apresentados constituem a base para a compreensão de funções mais complexas, as quais serão vistas nos próximos capítulos. Na seção seguinte, as funções são apresentadas em uma notação em blocos, procurando elucidar o funcionamento de funções simples e as que derivam das mesmas.

2.2 Visualizando Funções em Blocos

Com base nos exemplos, evidencia-se que uma função tem o conceito embutido de encapsulamento. Procura-se sintetizar todo o conhecimento que foi visto nas figuras, a fim de se estabelecer uma notação gráfica e um estilo de programação bem próprio do Haskell que é utilizado nos capítulos subseqüentes. Para os iniciantes, a notação gráfica apresentada é útil; já para os programadores avançados, a mesma fica em um nível natural de como se constrói um programa.

Assim, uma função é uma fórmula que efetua um tipo de cálculo específico. Para utilizar uma função basta fornecer, se necessário, os valores apropriados para a realização desse cálculo. Uma função possui um nome que a identifica e uma lista de argumentos (parâmetros) que vem após o nome, identificadora dos valores sobre os quais a função se aplica. Veja o exemplo da função `soma` a seguir:

```
soma x y = x + y
```

Execução:

```
Main> soma 1 2
3
```

A função `soma`, em que os valores `x` e `y` são passados como parâmetro, retorna como saída um inteiro.

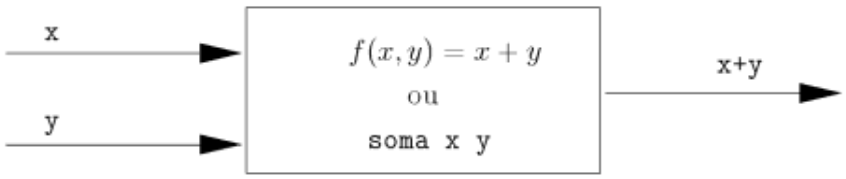


Figura 2.5 – Um bloco funcional, representando uma função

Quanto ao reuso de código, trata-se de uma das características da linguagem Haskell, reusando a função `soma` para o cálculo da média aritmética de dois valores. O exemplo seguinte utiliza a função `soma`, definida por `soma_3`, pois soma três números inteiros $x + y + z$. Assim tal função é dada por:

```
soma_3  x  y  z =  x + y + z
```

Observando-se `soma_3`, é possível observar que a função "+" se repete em dois lugares. Reusando tal função, `soma_3` pode ser reescrita como:

```
soma_3  x  y  z =  x + (soma y  z)
```

ou

```
soma_3  x  y  z =  (soma x y) +  z
```

Substituindo a função "+" por `soma`, tem-se:

```
soma_3  x  y  z =  soma x (soma y  z)
```

ou

```
soma_3  x  y  z =  soma (soma x y)  z
```

A execução de qualquer uma das versões de `soma_3` é dada por:

```
Main> soma_3  4  1  3
8
Main>
```

Este conceito de reuso de funções é um dos pontos fortes do Haskell. Os conceitos apresentados nesta seção são resumidos graficamente na Figura 2.6.

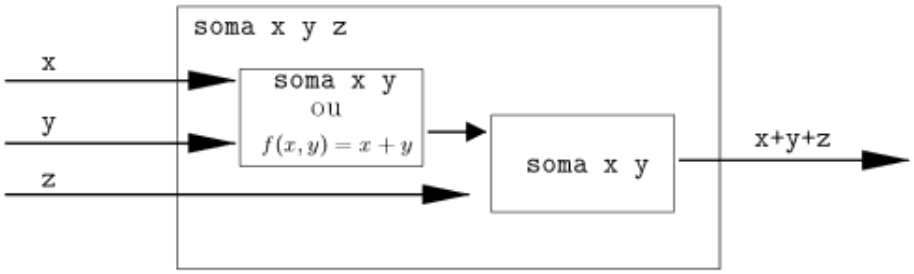


Figura 2.6 – Compondo funções para formar novas funções

O reuso de funções pode ser bem ilustrado por funções que calculam a média de três e quatro valores. A partir da idéia anterior, várias combinações podem ser feitas para a obtenção do resultado esperado. Assim, tem-se:

```
media_3 x y z = (soma_3 x y z) / 3
media_4 x y z w = soma_3 (soma x y) z w / 4
```

Execução:

```
Main> media_3  9.1  8.2  7.3
8.2
Main> media_4  9.1  8.2  7.3  6.4
7.75
```

Considere agora o problema do cálculo da hipotenusa (h) de um triângulo-retângulo, dado na Figura 2.7.

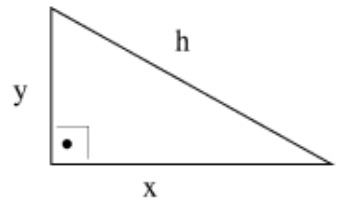


Figura 2.7 – Um triângulo-retângulo.

O valor de h é dado por:

$$h = \sqrt{x^2 + y^2}$$

Um diagrama funcional para esta fórmula pode ser apresentado na Figura 2.7, na qual o código Haskell equivalente para esta fórmula é dado por:

```
hipotenusa x y = sqrt (soma (x*x) (y*y))
```

Execução:

```
Main> hipotenusa 34 67
75.1332150250473
Main>
```

2.3 Exercício Resolvido

O problema a seguir descreve o cálculo da distância entre dois pontos. Os pontos $A(x_1, y_1)$ e $B(x_2, y_2)$ definem uma reta.

$$d_{AB}(x_1, y_1)(x_2, y_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Definida a função geral da reta, para o cálculo da distância entre dois pontos, são mostradas particularidades da função, que ocorrem devido à disposição da reta paralela a um dos eixos onde os pontos se encontram.

1. Se a reta que passa pelos pontos **A** e **B** (segmento **AB**) for paralela ao eixo das abscissas(x), deve-se aplicar a fórmula: $d_{AB} = |x_2 - x_1|$. Neste caso tem-se $y_1 = y_2$.

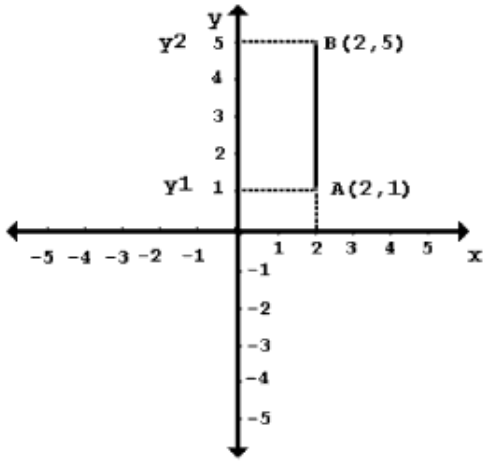


Figura 2.8 – Reta paralela ao eixo y.

Em Haskell, o código desta função é praticamente idêntico à formulação matemática:

```
-- Toda linha em Haskell é comentada a partir da seqüência "--"
d_AB :: Int -> Int -> Int  -- Aqui o tipo da função é especificada
d_AB x1 x2 = abs (x2 - x1)
```

Execução:

```
Main> d_AB 2 5
3
```

■ OBSERVAÇÕES ■ Neste código, foram adicionados dois novos elementos à sintaxe dos programas em Haskell:

- ✓ A seqüência "−−" que torna uma linha comentada a partir da sua direita no código-fonte. Esse detalhe de comentar códigos é apresentado na seção 14.5.
- ✓ Haskell é uma linguagem fortemente tipada. Mas, felizmente, a mesma procura *adivinhar* os tipos de dados utilizados no programa, liberando o programador de tal tarefa. Assim, na segunda linha do código, introduz-se a tipagem da função `d_AB` e de seus argumentos. Tal assunto é abordado no Capítulo 6.

2. Se a reta que passa pelos pontos **A** e **B** (segmento **AB**) for paralela ao eixo das ordenadas (y), deve-se aplicar a fórmula: $d_{AB} = |y_2 - y_1|$. Neste caso tem-se $x_1 = x_2$.

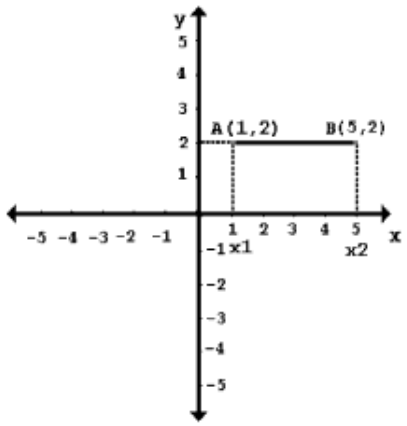


Figura 2.9 – Reta paralela ao eixo x.

Em Haskell, o conceito desta função é escrito da mesma maneira:

```
d_AB :: Int -> Int -> Int
d_AB y1 y2 = abs (y2 - y1)
```

Execução:

```
Main> d_AB 2 (-5)
7
```

3. Se a reta que passa pelos pontos **A** e **B** (segmento de reta **AB**), não é paralela a nenhum eixo, é aplicada a equação geral da reta. Ver Figura 2.10.

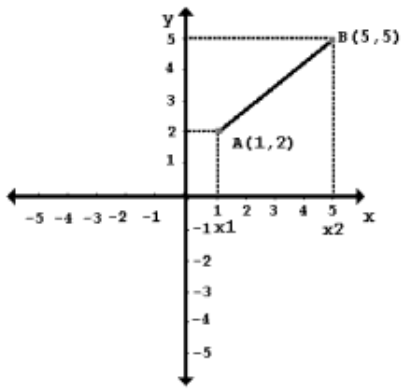


Figura 2.10 – Reta inclinada em relação aos dois eixos cartesianos.

A distância entre **AB** depende das diferenças entre abscissas e ordenadas de tal forma que, ao se aplicar o teorema de Pitágoras no $\triangle ABC$, deduz-se que:

$$d_{AB}^2 = d_{AC}^2 + d_{BC}^2$$

$$d_{AB}^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2$$

$$d_{AB} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

A partir dessa definição matemática do problema, uma parte do código já foi apresentada. Assim, o problema por completo é reescrito por uma única função, a qual é dada por:

```
d_AB x1 y1  x2 y2 | x1 == x2 = abs (y2-y1) -- Condição 1
                  | y1 == y2 = abs (x2-x1) -- Condição 2
                  | otherwise = sqrt ( (x2-x1)^2 + (y2-y1)^2 ) -- Condição 3
```

Execução:

```
Main> d_AB 1 2 5 2
4.0
Main> d_AB 1 3 1 4
1.0
Main> d_AB 1 2 5 5
5.0
```

Esta função possui quatro argumentos, `x1`, `y1`, `x2`, e `y2`, que representam os dois pontos A e B no espaço cartesiano bidimensional. Neste código, a função `d_AB` faz uso das funções `abs` e `sqrt`, as quais estão definidas na biblioteca `Prelude.hs` do Haskell. Esta biblioteca contém várias funções básicas, além de ser carregada automaticamente por default, em qualquer interpretador Haskell.

> ■ OBSERVAÇÃO ■ Há um novo símbolo neste último código: "|". Este símbolo é conhecido como *guardas*, e separa o nome da função e seus argumentos de uma parte condicional à execução do corpo da função. Isto é, após uma guarda "|", há uma condição lógica a ser verificada. Caso esta condição tenha uma resposta lógica verdadeira, a parte seguinte ao sinal de "=" é executado. Caso a condição lógica seja falsa, uma próxima condicional, dada pela guarda subseqüente, é testada, e assim sucessivamente. Esta estrutura com guardas na função `d_AB` tem uma analogia direta com as estruturas `if-then` e `case` da programação imperativa. O conceito de guarda é detalhado na seção 3.3.

2.4 Exercícios Propostos

1. Fornecidos três valores, a, b e c, escreva uma função que retorne quantos dos três são iguais. A resposta pode ser 3 (todos iguais), 2 (dois iguais e o terceiro diferente) ou 0 (todos diferentes).
2. Fornecidos três valores, a, b e c, elaborar uma função que retorne quantos desses três números são maiores que o valor médio entre eles.
3. Escrever uma função `potencia_2` que retorne o quadrado de um número (x^2).
4. Reutilizando a função `potencia_2`, construir uma função `potencia_4` que retorne o seu argumento elevado à quarta potência.

5. Implemente em Haskell a função do or-exclusivo, a qual é dada por:

$$a \otimes b = (a \ \vee \ b) \wedge \sim (a \ \wedge \ b)$$

■ OBSERVAÇÃO ■ Os conectivos lógicos encontram-se na seção 6.1.1.

6. Escrever duas funções, `x_maior` que retorne o maior e `x_menor` que retorne menor valor real, das raízes de uma equação do 2^{o} grau. A expressão genérica é dada por:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Capítulo 3

Uma Visão Funcional da Indução

Este capítulo trata de um problema comum ao uso de uma linguagem funcional: *o problema da recursividade*. A visão via *indução matemática* é uma das mais apropriadas, pois a mesma proporciona uma *escadaria* típica da recursividade [1].

A indução matemática é uma definição generalizada para uma conjunto de objetos que tenham em comum uma propriedade. Para se demonstrar que este conjunto exibe tal propriedade, deve-se conhecer o caso trivial. Tal caso corresponde normalmente a `n=0`, `n=1`, etc., que são os valores-base ou triviais. O passo seguinte é demonstrar que, para um `n` genérico, a propriedade indutiva mantém-se a partir de `n-1`. Assim, o termo `n` dessa propriedade indutiva é validada, considerando que o caso anterior, `n-1`, era também verdadeiro.

O conceito dessa idéia indutiva é a base de toda a programação em Haskell, mas, esses passos matemáticos, são deixados a cargo do computador. Para tais cálculos, uma *escadaria* de um caso `n` que se deseja demonstrar é construída, do fim para o início, até o caso trivial ou conhecido dessa regra recursiva.

Inicialmente, este capítulo define a recursão em detalhes, apresenta exemplos, cuidados e perigos da mesma. Em seguida, o conceito de *casamento de padrões* para o Haskell é definido de forma genérica e irrestrita. O capítulo é encerrado com um conceito particular das linguagens funcionais, que é a *avaliação preguiçosa* (*lazy evaluation*). Um exemplo é usado para exibir esta característica.

3.1 Recursão

O clássico exemplo da recursão, por meio de uma idéia indutiva, é o da soma dos `n` primeiros inteiros. Isto é, como somar `1+2+3+4+..  +(n-1)+n`, representado por `soma(n)`. Exemplificando: `soma(5)=1+2+3+4+5=15`. Assim, para resolver o problema

da soma de `1` até `n` é utilizado o seguinte raciocínio:

$$
\begin{array}{rcl}
 & & 1 \quad + \quad 2 \; + \; 3 \; + 4 \quad + \ldots + n \\
soma\ 1 & = & \underbrace{1} \\
soma\ 2 & = & \underbrace{(soma\ 1) \quad + \quad 2} \\
soma\ 3 & = & \quad \underbrace{(soma\ 2) \quad + \quad 3} \\
soma\ 4 & = & \quad\quad (soma\ 3) \quad + \quad 4 \\
 & & \vdots \\
soma\ n & = & \quad\quad\quad (soma(n-1)) \quad + \quad n
\end{array}
$$

Sob uma definição matemática desta soma de inteiros entre `1` e `n`, é dada por:

$$
soma(n) = \begin{cases} 1 & : \quad n = 1 \\ soma(n-1) + 1 & : \quad n > 1 \end{cases}
$$

Como se observa, há uma soma repetitiva entre o último número com a soma acumulada até o número antecessor. Sendo assim, a última linha descrita é a generalização da função `soma`, a qual é a *regra indutiva geral*, dada por:

$$
soma \quad n = soma \quad (n-1) \quad + \quad n
$$

Tendo o problema generalizado, basta descrever o aterramento, ou seja a última ação da função soma, que, no caso, é a `soma 1`. Em seguida, basta adicionar a última linha do problema (a linha generalizada), pois essa se encarrega recursivamente de resolver o problema. Neste caso o código fica descrito como:

```
soma 1 = 1
soma n = soma (n-1) + n
```

A execução deste programa é dado por:

```
Main> soma 4
10
Main> soma 2
3
```

Considerando a fórmula genérica do caso em questão, determina-se quem é o aterramento ou termo inicial do processo indutivo, e em seguida traduz o código matemático na notação do Haskell. Para compreender como o interpretador Haskell executa tal código, é feita uma *escadaria da recursividade* [1], que representa as sucessivas chamadas recursivas que a linguagem Haskell realiza, dada por:

```
soma 4
= (soma 3 )  + 4
= ((soma 2 ) + 3) + 4
= (((soma 1 ) + 2) + 3) + 4
```

Agora, o interpretador resolve o problema da prioridade com base nos parênteses, que, neste caso, é a soma dos números:

```
= (((1) + 2) + 3) + 4
= ((3) + 3) + 4
= (6) + 4
= 10
```

No retorno da chamada, os elementos anteriores são visitados a partir da regra geral, em que o objeto `n` possui uma relação ao seu antecessor `(n-1)`, chegando, assim, ao resultado esperado 10.

3.2 Exemplos

3.2.1 Fatorial

Um exemplo análogo ao anterior é o fatorial de um número inteiro. A exemplo da soma dos `n` inteiros, para o caso do fatorial é dado por:

$$
\begin{array}{lcl}
fatorial\ n & = & 0 * 1 * 2 * 3 * 4 * ... * n \\
fatorial\ 0 & = & \underbrace{1} \\
fatorial\ 1 & = & \underbrace{(fatorial\ 0) * 1} \\
fatorial\ 2 & = & \underbrace{(fatorial\ 1) * 2} \\
fatorial\ 3 & = & \underbrace{(fatorial\ 2) * 3} \\
fatorial\ 4 & = & (fatorial\ 3) * 4 \\
 & . & \\
 & . & \\
 & . & \\
fatorial\ n & = & (fatorial(n-1)) * n
\end{array}
$$

Sob uma definição matemática, o número fatorial de um inteiro é indutivamente dado por:

$$
fatorial(n) = \begin{cases} 1 & : \quad n = 0 \\ fatorial(n-1) * n & : \quad n \geq 1 \end{cases}
$$

O fatorial é o número corrente multiplicado pelo fatorial do seu antecessor. Assim como o apresentado na última linha do problema a generalização fica:

$$
fatorial\ \ n = fatorial\ \ (n-1) \ * \ n
$$

Ao deduzir a fórmula generalizadora do fatorial, é necessário definir o aterramento para o caso do fatorial. Sabe-se que `fatorial(0) = 1` . Em seguida, a linha `fatorial(0)=1` é adicionada antes da fórmula generalizadora `fatorial n = fatorial (n-1) * n`. Logo, em Haskell, este código é dado como:

```
fatorial 0 = 1
fatorial n = fatorial (n-1) * n
```

Execução:

```
Main> fatorial 2
2
Main> fatorial 5
120
```

A *escadaria da recursividade* da função `fatorial` segue a mesma lógica da função `soma`, tal como:

```
fatorial 5
= (fatorial 4) * 5
= ((fatorial 3 )  * 4 ) * 5
= (((fatorial 2 ) * 3) * 4) * 5
= ((((fatorial 1 ) * 2) * 3) * 4) * 5
```

Efetivamente, o cálculo é expresso em símbolos, tais como: `fatorial 4`, `fatorial 3`, ..., até `fatorial 1`. O cálculo numérico é realizado seguindo a ordem implícita dos parênteses, não existindo nenhuma outra pendência de símbolos na expressão. Assim, a multiplicação entre os números no cálculo de `fatorial 5` é dado por:

```
= ((((1) * 2) * 3) * 4) * 5
= (((2) * 3) * 4) * 5
= ((6) * 4) * 5
= (24) * 5
= 120
```

Finalmente, o resultado esperado é `120`, o qual é obtido após sucessivas substituições de símbolos por números, e ao final operar apenas com números. Este conceito é conhecido como *avaliação preguiçosa* – e é retomado na seção 3.5.

3.2.2 Fibonacci

Os números de Fibonacci são definidos da seguinte forma: o primeiro é `1`, o segundo também é `1`, o n-ésimo número é definido como a soma dos dois números anteriores. Sob uma definição matemática, o número Fibonacci é indutivamente dado por:

$$fib(n) = \begin{cases} 0 & : \quad n = 0 \\ 1 & : \quad n = 1 \\ fib(n-1) + fib(n-2) & : \quad n > 1 \end{cases}$$

Qual é a função que calcula o n-ésimo número de Fibonacci? Uma alternativa é programar a função de modo recursivo. A implementação é trivial e existe uma tradução direta da definição dos números de Fibonacci:

```
fib 0 = 0
fib 1 = 1
fib n = soma (fib (n-1))  (fib (n-2))    --  Equivalente a: fib (n-1) + fib (n-2)
```

Execução:

```
ain> fib 7
13
Main> fib 8
21
Main> fib 9
34
```

Cada vez que a função `fib` é chamada, a dimensão do problema se reduz em uma unidade (de `n` para `n-1`), mas são feitas duas chamadas recursivas. Isto dá origem a uma explosão no número de chamadas sobre ela própria, em que um dado termo da expressão é calculado várias vezes, em instâncias diferentes.

Curiosamente, esta função de Fibonacci tem uma definição não-recursiva, porém aproximada, dada por:

$$fib(n) = \frac{1}{\sqrt{5}}(\phi_1^n - \phi_2^n)$$

onde:

$$\phi_1 = \frac{1+\sqrt{5}}{2} \qquad \text{e} \qquad \phi_2 = \frac{1-\sqrt{5}}{2}$$

Esta função como todas as outras, podem ser provadas pela indução matemática. Há vários livros de matemática e sítios na internet, que tratam do assunto. Por exemplo, detalhes dessa demonstração para o caso dos números de Fibonacci podem ser encontrados em: `http://www.brpreiss.com/books/opus4/html/page73.html`. Adicionalmente, há uma análise da taxa de crescimento desta função, demonstrando a explosão combinatorial que a mesma exibe.

3.3 Perigos da Recursão

As funções recursivas em Haskell merecem uma atenção especial. Como se observa, uma função recursiva é uma função que chama a ela mesma, e quase todas as construções em Haskell são recursivas, pois necessitam de algum tipo de repetição. Praticamente em toda a função recursiva existe o conceito de *aterramento*, o qual deve sempre vir antes da linha que faz a chamada da função recursiva geral. Caso se inverta a seqüência dessas linhas, uma seqüência infinita de chamadas vai efetivamente ocorrer, se os critérios de parada não estejam bem-definidos. Uma função bem-definida, é aquela que possui um número de passos finitos, em que seus `n` aterramentos sempre se encontram antes da função recursiva geral.

O exemplo[1], a seguir, descreve a função `mult_7`, cujo objetivo é retornar o número de múltiplos de `7` encontrados no intervalo de `0` até um número informado pelo usuário.

```
mult_7 3 = 0
mult_7 2 = 0
mult_7 5 = 0
mult_7 6 = 0
```

[1] Este exemplo tem uma motivação que vai ao encontro do conteúdo da seção 3.4.

```
mult_7 4 = 0
mult_7 1 = 0
mult_7 7 = 1
mult_7 x = 1 + mult_7 (x-7)
```

Execução:

```
Main> mult_7 9
1
Main> mult_7 18
2
Main> mult_7 21
3
Main> mult_7 3
0
```

A ordem dos aterramentos na função `mult_7` não tem importância. Caso seja colocado um e apenas um dos aterramentos depois da função recursiva geral, em algumas ocasiões a função recursiva geral sempre chama a si mesma causando chamadas recursivas infinitas. O interpretador/compilador retorna o erro *stack overflow*, que é o estouro de pilha. No caso, a capacidade de se armazenarem dados na memória disponível para o interpretador é sobrecarregada. Em seguida, alguns exemplos de função malformada:

```
multiplo_7 3 = 0
multiplo_7 5 = 0
multiplo_7 6 = 0
multiplo_7 4 = 0
multiplo_7 1 = 0
multiplo_7 7 = 1
multiplo_7 x = 1 + multiplo_7 (x-7)
multiplo_7 2 = 0
```

Execução:

```
Main> multiplo_7 4
0
Main> multiplo_7 10
1
Main> multiplo_7 9

ERROR: Control stack overflow
```

```
multiplo_7 3 = 0
multiplo_7 2 = 0
multiplo_7 5 = 0
multiplo_7 4 = 0
multiplo_7 1 = 0
multiplo_7 7 = 1
multiplo_7 x = 1 + multiplo_7 (x-7)

Main> multiplo_7 8
1
Main> multiplo_7 15
```

```
2
Main> multiplo_7 13

ERROR: Control stack overflow
```

No primeiro exemplo, no qual o aterramento `multiplo_7 2 = 0` é colocado depois da função recursiva geral `multiplo_7 x = 1 + multiplo_7 (x-7)`, a função continua funcionando. Mas quando se tem um caso em que existe uma chamada do aterramento `multiplo_7 2 = 0`, a função recursiva geral chama a ela mesma infinitas vezes, causando estouro de pilha. Já no segundo exemplo ocorre o mesmo erro ao chamar o aterramento `multiplo_7 6 = 0`, pois o mesmo foi omitido.

Uma solução alternativa e prática para essa solução é a utilização de *guardas* ('|'), onde o exemplo anterior deve ser reescrito da seguinte maneira:

```
multi_7 7= 1
multi_7 x |(x>=1) && (x<=6) = 0
          |otherwise        = 1 + multi_7 (x-7)
```

Execução:

```
Main> multi_7 70
10
Main> multi_7 1
0
Main> multi_7 4
0
Main> multi_7 7
1
```

Uma *guarda* é uma condição lógica a ser verdadeira, para que a definição da função seja executada. Caso esta condição lógica não seja verdadeira, a próxima guarda é verificada. Na função `multi_7` a condição aplicada é `(x>=1) && (x<=6)`. A condição `otherwise` aceita qualquer condição, ou seja, qualquer condição não-satisfeita nas guardas anteriores é aplicada à condição `otherwise`. As guardas são montadas da seguinte maneira:

```
função parametro_1 parametro_2 ...
       |guarda_1   = expressão_1
       |guarda_2   = expressão_2
       ...
       |otherwise = expressão_n
```

Uma função pode possuir `n` parâmetros, `n` guardas e suas respectivas expressões a serem realizadas, e um `otherwise`, que abrange as demais condições existentes e não declaradas.

3.4 Casamento de Padrões

Observando-se os exemplos da seção 3.3, é possível verificar inicialmente que há muitas linhas de código duplicadas para função `multi_7`. Os parâmetros da função `multi_7`

são diferenciados por algum valor entre 1 e 7, mas que, ao final, com a introdução das *guardas*, a parte condicional da função, esta ficou bastante resumida em sua notação. A variável x na função `multi_7` fez um *casamento* condicional com valores de 1 a 6, retornando 0, enquanto, para os demais valores, recursivamente segue-se na busca de mais múltiplos de 7.

Uma pergunta: *esta duplicação de linhas de código poderia ser evitada*? A resposta é *sim*, e o conceito para esta equivalência é o de *casamento de padrão*. Em um exemplo com objetivos didáticos, o conceito de casamento padrão é exemplificado pelo código a seguir:

```
f x y z | (x == 7) = 10
f x y z | (y == 8) = 20
f x y z | (z == 9) = 30
        | otherwise = 0
```

Sob uma notação com várias condicionais na mesma linha:

```
g 7 y z  = 10
g x 8 z  = 20
g x y 9  = 30
g x y z  | (x /= 7) || (y /= 8) || (z /= 9)  = 0
```

A questão é: *Como tornar estas linhas equivalentes, uma vez que as variáveis são indepentes entre si?* Há uma variável conhecida como *anônima*, a qual é simbolizada por "_" (em inglês é chamado de *underscore*), também conhecido como *coringa* ou *wild card*. Usando esta variável anônima, "_", o código anterior pode ser reescrito da seguinte maneira:

```
h  7 _ _   =  10
h  _ 8 _   =  20
h  _ _ 9   =  30
h  _  _ _  =  0
```

A execução das funções `f`, `g` e `h` são dadas por:

```
Main> f 7 8 9
10
Main> f 17 8 9
20
Main> f 17 18 9
30
Main> f 17 18 19
0
Main> g 7 8 9
10
Main> g 17 8 9
20
Main> g 17 18 9
30
Main> g 17 18 19
0
Main> h 7 8 9
10
```

```
Main> h 17 8 9
20
Main> h 17 18 9
30
Main> h 17 18 19
0
Main>
```

Assim, a função ativada e executada é aquela que se ajusta com um dos padrões no argumento, mas sem considerar o conteúdo quando o argumento for "_". Os identificadores são vinculados aos valores identificados para avaliação do lado direito da equação.

Os padrões permitidos são constantes, tais como inteiros, valores booleanos, tuplas, nomes de parâmetros formais, etc. Os padrões não-permitidos são expressões aritméticas, relacionais ou lógicas. Os nomes dos identificadores devem ser diferentes. Alguns outros exemplos dessas definições são dadas por:

```
iguais :: Int -> Int -> Bool
iguais x x = True
iguais x y = False

ERRO - variavel repetida em padrao!!! (Repeated variable ``x'' in  pattern)

-- Solução:

iguais :: Int -> Int -> Bool
iguais x y | x == y = True
iguais x y = False
```

Aqui um "_" pode ser usado quando um valor não interessa para avaliação. Por exemplo, a definição do operador lógico `ou` (`or`):

```
ou :: Bool -> Bool -> Bool
ou False False = False
ou   _     _   = True
```

Neste caso, somente quando as variáveis passadas como parâmetro forem `False`, o resultado é `False`. Em qualquer outro caso, a variável anônima "_" assume o resultado `True`.

3.5 Avaliação Preguiçosa

A linguagem Haskell utiliza uma estratégia na avaliação das funções denominada *avaliação preguiçosa* (*lazy evaluation*), cujo fundamento é que não se avalia nenhuma subexpressão ou função até que seu valor seja reconhecido como necessário. Este conceito é ilustrado pelo exemplo a seguir:

```
dobro, triplo :: Int -> Int
menor, f :: Int -> Int -> Int

menor x y
```

```
   | x < y = x
   | otherwise = y

dobro x = x + x
triplo x = 3 * x
f a b  =  (dobro (triplo (menor a  b) ) )
```

Ao se executar a função `f`, a sua avaliação real é dada por:

```
f 9 8 = (dobro (triplo (menor  9  8) ) )
      = (triplo (menor  9  8)) + (triplo (menor  9  8))
      = (3 * (menor  9  8)) + (triplo (menor  9  8))
      = (3 * (menor  9  8)) + (3 * (menor  9  8))
      = (3 * 8) + (3 * (menor  9  8))
      = (3 * 8) + (3 * 8)
      = 24 + (3 *  8)
      = 24 + 24
      = 48
```

O exemplo mostra o conceito da avaliação preguiçosa, na qual tudo é tratado como *representação simbólica* de expressões e funções, adiando tanto quanto possível a sua avaliação. Se um argumento não for utilizado, este não será avaliado. Eventualmente, o mesmo nem vem ser calculado, economizando, assim, recursos computacionais.

Esta representação em símbolos para funções e expressões é evidente para usuários, mas não para máquinas. Por exemplo, as expressões:

$$1 + 2 + 4! == 4! + 2 + 1 == 2 + 4! + 1$$

Alguém avaliaria imediatamente que, pela propriedade de comutatividade da adição, tais expressões são equivalentes. Uma máquina, que não esteja programada para avaliar a comutatividade da adição em expressões aritméticas, deve calcular cada uma das expressões para, então, validar que o resultado da expressão é 27 nas três avaliações.

A avaliação preguiçosa é utilizada na implementação de funções *não-estritas* (*non-strict*), que recebem argumentos não-computados, só os calculando quando necessário. Já as funções de alta-ordem[2] recebem funções como argumentos e as retornam durante a avaliação de uma expressão.

Este tipo de avaliação traz muitas vantagens, inclusive a habilidade de representar potencialmente estruturas de dados infinitas, como o conjunto dos números inteiros[3]. A avaliação preguiçosa implica que funções tanto podem receber argumentos não-avaliados como também retornar resultados parcialmente não-avaliados.

Em resumo, as avaliações parciais ou preguiçosas são uma característica de linguagens não-procedurais, presentes em linguagens funcionais como o Haskell. Nos exemplos ao longo deste livro, observe o procedimento da avaliação preguiçosa sob funções e expressões.

[2] Assunto apresentado no Capítulo 9.
[3] Assunto apresentado no Capítulo 5.

3.6 Exercícios Propostos

1. Calcular a soma entre dois números n_1 e n_2 incluindo e excluindo os limites.
2. Dados dois números n_1 e n_2, encontrar os múltiplos de n_3 que se encontram nesse intervalo.
3. Utilizando a função da soma, faça uma função que calcule a multiplicação entre dois números quaisquer, considerando números positivos e negativos.

   ```
   mult (-4) 5
   ```

4. Seja a expansão e^x definido pela série de Taylor

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + ...$$

   (a) Calcule a soma da série para `n=10` termos (não é o `x` da série) e teste para vários `x`.
   (b) Considere que o valor analítico de e^x seja dado pela função `exp x`. Para um erro (valor_Analitico - valor_Da_Serie) menor que `0,001`, quantos `n` termos da série são necessários?

5. Implemente a função `mod` (função que retorna o resto de uma divisão de inteiros). Obviamente, não pode ser utilizada a função `mod` do interpretador. Exemplo:

   ```
   Main> mod2 11 4
   3
   ```

   Procure implementar a partir do algoritmo de Euclides.
6. Seja a seqüência:

$a_1 = \sqrt{6}$
$a_2 = \sqrt{6 + \sqrt{6}}$
$a_3 = \sqrt{6 + \sqrt{6 + \sqrt{6}}}$
$a_4 = ...$

   (a) Encontre a forma recursiva para a_{n+1};
   (b) Encontre a soma dos 10 primeiros termos.

7. Implementar a fórmula que indica de quantas maneiras é possível escolher **n** objetos de uma coleção original de **m** objetos, onde $m \geq n$

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}$$

8. Construa uma função que retorne o MMC (Mínimo Múltiplo Comum) entre três números inteiros.

   ```
   Exemplo:
   Main mmc 2 4 3
   12
   ```

9. Construa uma função que calcule a raiz quadrada inteira de um número inteiro. Implemente uma função que, a partir de um número fornecido pelo usuário, calcule o valor inteiro.

```
Exemplo:
Main> raiz 65
 8
```

10. Construa a função de Ackermann, a qual é definida por:
    1. $a(m, n) = 1$ se $m = 0$
    2. $a(m, n) = a(m - 1, 1)$ se $m \neq 0$ e $n = 0$
    3. $a(m, n) = a(m - 1, a(m, n - 1))$ se $m \neq 0$ e $n \neq 0$

```
Exemplo:
Main> acker (2, 2)
 7
```

11. Construa uma função que retorne o Máximo Divisor Comum entre dois números inteiros, e, caso contrário, retorne 0.

Capítulo 4

Tuplas

Este capítulo trata sobre funções que manipulam um conjunto de dados heterogêneos de modo encapsulado. Tal conjunto heterogêneo de dados representa uma relação entre seus dados em face de sua *posicionalidade* na estrutura. Uma tupla estabelece que objetos não-homogêneos de um conjunto estejam estruturados sob uma forma desejada. Assim, uma *estrutura relacional* de objetos heterogêneos define uma tupla.

O número de objetos nesta relação define o tamanho da tupla. Por exemplo, uma tupla com dois objetos é chamada de *par* ou tupla-2. Com três objetos é uma *tripla* ou tupla-3, e assim sucessivamente. Além de definir e exemplificar as tuplas em Haskell, o capítulo é finalizado com um exercício que acumula todo o conhecimento descrito até o momento.

4.1 Apresentação às Tuplas

As *tuplas* em Haskell permitem a definição e o uso de tipos de dados heterogêneos sob uma estrutura relacional. Por exemplo, para desenhar uma figura qualquer no monitor de vídeo, definido em um espaço cartersiano de duas dimensões, é necessário referenciar a posição do desenho na tela por um par de pontos: (x, y). Sob esta mesma notação, basta indicar este par de pontos por coordenadas entre parênteses, e o conteúdo separado, por uma vírgula. Essa é uma estrutura de dados cuja notação é diretamente suportada pelo Haskell.

```
Prelude> (1,3)
(1,3)
```

Neste exemplo, foi utilizado um par de inteiros, `1` e `3`. Esse par não necessita ser de um único tipo de conteúdo. Muito pelo contrário. Os pares heterogêneos são os de maior interesse prático. Os problemas típicos da área envolvem itens como nomes, idade, peso, etc., logo, uma variedade de tipos. Por exemplo, seja uma tupla-4 que tenha conteúdo como nome, idade, peso e esporte predileto de uma pessoa:

```
Prelude> ("Fernanda", 18, 65.23, "Tenis")
("Fernanda",18,65.23,"Tenis")
```

Essa tupla-4 é estruturada com os seguintes tipos de dados:

1º. campo	2º. campo	3º. campo	4º. campo
Fernanda	18	65.23	Tenis
⇓	⇓	⇓	⇓
String	Int	Float	String

O primeiro campo desta tupla-4 é uma `String`. Esse tipo de dados é uma lista[1] de caracteres.

> ■ OBSERVAÇÃO ■ O tipo `String` é definido como um conjunto seqüencial de caracteres textuais da tabela ASCII, tal como uma palavra, frase, etc. Esse tipo de dado é apresentado na seção 5.6, onde o mesmo é definido como uma lista ou seqüência de caracteres.

O segundo campo, destinado à idade, é do tipo inteiro. No terceiro campo, um valor real ou de ponto flutuante, para o caso de um conteúdo como peso de uma pessoa.

A principal aplicação das tuplas é definir uma função que receba ou retorne mais de um valor e, geralmente, de tipo heterogêneo. Apesar de suas vantagens, a tupla tem uma estrutura estática, ou seja, uma vez criada, não pode ser modificada. Este fato fica evidente nos exemplos deste capítulo.

No caso particular das tuplas-2 ou par binário, a linguagem Haskell tem duas funções pré-definidas: `fst` e `snd`. A função `fst` (do inglês *first*) extrai o primeiro elemento da tupla. A função `snd` (do inglês *second*) extrai o segundo elemento da tupla. Exemplo:

```
Prelude> fst (5, "bom dia")
5
Prelude> snd (5, "bom dia")
"bom dia"
```

As tuplas com um número maior de elementos podem ser definidas com três, quatro, cinco ou até `n` elementos. Uma tripla, uma quádrupla podem ser definidas respectivamente por:

```
Prelude> (1,2,3)
(1,2,3)
Prelude> (1,2,3,4)
(1,2,3,4)
```

[1] As listas de caracteres são tratadas no Capítulo 5.

```
5
Prelude> 2+3*4
14
Prelude> 2 / 3 * 4
2.66666666666667
Prelude> (2 / 3) * 4
2.66666666666667
Prelude> (*) ((/) 2 3)  4    -- em uma notação prefixa
2.66666666666667
```

Operadores Relacionais

Após a apresentação dos números inteiros e dos booleanos, é possível definir com precisão os operadores relacionais. Esses operadores retornam uma saída do tipo `Bool` (Booleano), descritos na Tabela 6.3.

Tabela 6.3 – Operadores relacionais.

Operador	Descrição	Exemplos
`>`	Maior do que	`3 > 2 = True, (>) 3 3 = False`
`>=`	Maior e igual a	`3 >= 2 = True, (>=) 3 3 = True`
`<`	Menor do que	`3 < 2 = False, (<) 3 3 = False`
`<=`	Menor e igual a	`3 <= 2 = False, (<=) 3 3 = True`
`==`	Igual a	`3 == 2 = False, (==) 3 3 = True`
`/=`	Diferente de	`3 /= 3 = False, (/=) 3 2 = True`

Todos os operadores relacionais da Tabela 6.3, para o caso dos inteiros, têm a seguinte tipagem: `Int -> Int -> Bool`. Para o caso do tipo `Bool`, seguem `Bool -> Bool -> Bool`.

6.1.3 Caracteres

Em geral, pessoas e computadores comunicam-se via um teclado como entrada e a tela como saída, os quais são baseados em seqüências de caracteres. Os caracteres da linguagem Haskell são simplesmente todos os caracteres do código ASCII, acrescidos de alguns aqueles: tabulação ($\backslash t$), nova linha ($\backslash n$), barra invertida ($\backslash\backslash$), apóstrofo (aspa simples) ($\backslash'$), e aspas duplas ($\backslash''$).

Em Haskell os caracteres são escritos entre aspas simples. Do mesmo modo que um caracter numeral obrigatoriamente deve ser escrito entre aspas simples, a fim de que se entenda que ele é um caracter. Caso contrário, é tratado como um número inteiro. Um conjunto de caracteres é denominado pelo tipo `String` e são apresentados na seção 5.6.

■ OBSERVAÇÃO ■ Um erro comum, é tratar que `a` e `'a'` como iguais. Existe uma diferença entre ambos, pois `a` pode ser o nome de uma função ou de uma variável, e `'a'` é um caracter literal do tipo `Char`.
Do mesmo modo, o número deve ser escrito como `'3'`, para que seja visto como caractere; caso contrário é tratado como um número.

Para trabalhar com caracteres utilizado-se o valor numérico da tabela ASCII ou o próprio caractere. Para tanto, é necessário conhecer os valores da tabela ASCII. Mas, felizmente não é preciso que se memorizem os valores, ou que se faça uma tabela para consulta. Para tanto, a biblioteca `Char.hs` possui duas funções definidas para a conversão de valores e caracteres. Em algumas versões do Haskell, tais funções já se encontram na biblioteca `Prelude.hs`. Essas funções são a `ord` e o `char`, exemplificadas por:

ord :: Char -> Int

A função converte o caractere para seu respectivo valor numérico na tabela ASCII. Exemplo:

```
Prelude> ord 'a'
97
```

chr :: Int -> Char

A função converte um valor para seu respectivo caractere na tabela ASCII. Exemplo:

```
Prelude> chr 97
'a'
```

Com estas duas funções são construídas outras funções sobre caracteres. Por exemplo, transformar as letras minúsculas em maiúsculas, já que a diferença entre elas é sempre `32`, pois:.

```
Prelude> ord 'a' - ord 'A'
32
Prelude> ord 'b' - ord 'B'
32
```

Ao utilizar a função `ord` para os valores correspondentes na tabela ASCII de `'a'` até `'z'` são devolvidos os valores `97` até `122`, respectivamente. Analogamente, de `'A'` até `'Z'` são retornados os valores de `65` a `90` respectivamente. Com base nesta diferença, as letras são transformadas de minúsculas para maiúsculas, e vice-versa. Para tanto, um valor inicial representa a diferença entre dois caracteres iguais, um minúsculo e outro maiúsculo, dados por:

```
dif :: Int

dif = 32
 -- ou
dif = ord 'a' - ord 'A'
```

Então, funções que transformem letras minúsculas em maiúsculas, e o contrário, são dadas por:

```
import Char

dif :: Int
dif = ord 'a' - ord 'A'  -- 32

maiusc_Min :: Char -> Char
maiusc_Min ch = chr ((ord ch) + dif)

minusc_Mai :: Char -> Char
minusc_Mai ch = chr ((ord ch) - dif)
```

Execução:

```
Main> maiusc_Min 'H'
'h'
Main> maiusc_Min 'x'
'\152'
Main> minusc_Mai 'a'
'A'
Main> minusc_Mai 'H'
'('
Main>
```

Propositadamente alguns resultados acontecem. As funções apresentadas são simples e sem nenhuma restrição sobre suas entradas. Assim, ao se digitar certos caracteres, o programa executa a função como for possível, eventualmente ocasionando erros. No caso da uma entrada como `'x'` já era um caractere minúsculo, tendo ocorrido a conversão além do último caractere válido, que era a letra `'z'`.

Para resolver esta dificuldade, o uso das guardas é mandatório. A partir destas funções, uma nova função `conv` é definida, reusando-as a fim de resolver essa falha no programa:

```
conv x
   | x >= 'a' && x <= 'z' = minusc_Mai x
   | x >= 'A' && x <= 'Z' = maiusc_Min x
   | otherwise = error " não é uma letra entre A e z  !!! "
```

Execução:

```
Main> conv 'x'
'X'
Main> conv 'H'
'h'
Main> conv '4'

Program error:  não é uma letra entre A e z !!!
```

As funções `chr` e `ord` são encontradas na biblioteca `Char.hs`. Algumas outras interessantes são dadas por:

Tabela 6.4 – Funções sobre caractere.

Função	Descrição	Exemplos
`isLower`	Função que retorna verdade caso o caractere digitado for minúsculo, do contrário retorna falso.	`isLower 'a' = True isLower 'A' = False`
`isUpper`	Função que retorna verdade caso o caractere digitado for maiúsculo, do contrário retorna falso.	`isUpper 'a' = False isUpper 'A' = True`
`toLower`	Função que recebe um caractere, e o converte em um caractere minúsculo equivalente.	`toLower 'A' = 'a'`
`toUpper`	Função que recebe um caractere, e converte o mesmo em um caractere maiúsculo equivalente.	`toUpper 'a' = 'A'`
`isDigit`	Verifica se o caractere é um dígito (de `'0'` à `'9'` entre aspas simples).	`isDigit '1' = True isDigit 'a' = False`
`digitToInt`	Transforma um caractere (de `'0'` à `'9'` entre aspas simples), em um número inteiro.	`digitToInt '1' = 1`
`intToDigit`	Faz o inverso da função `digitToInt` transforma um número inteiro, em um caractere.	`intToDigit 1 = '1'`

■ OBSERVAÇÃO ■ No início do arquivo, importar o módulo com estas funções:

```
import Char
```

6.1.4 Reais

Os números reais vêm para completar o conjunto numérico em Haskell. Para o uso de números com partes fracionárias, em Haskell são representados por números com pontos flutuantes, os quais são do tipo `Float`. O Haskell também possui um limite no tipo `Float`. Assim um tipo que dobra a precisão do `Float` chamado de `Double`.

Algumas funções/operadores que fazem parte do domínio dos reais são encontrados nas tabelas 6.5 e 6.6.

Sejam M e N λ-termos e x uma variável. A substituição das ocorrências livres de x em M por N (representada por $M\ [x \leftarrow N]$) é uma operação com maior precedência, pois se trata de uma aplicação, e também associativa à esquerda, sendo definida recursivamente[11] por:

$$\begin{array}{rcll} z[x \leftarrow N] & = & \begin{cases} N & se \quad x \equiv z \\ z & se \quad x \not\equiv z \end{cases} & \\ (E\ \ F)[x \leftarrow N] & = & (E[x \leftarrow N]\ \ F[x \leftarrow N]) & \\ (\lambda y.E)[x \leftarrow N] & = & \lambda z.((E[y/z])[x \leftarrow N]) & \text{sendo } z \text{ uma variável nova} \end{array}$$

Tal definição não leva a resultado nenhum, pois a variável nova pode ser dada por qualquer termo. A esse respeito nota-se que a escolha de diferentes variáveis novas nos conduz a resultados α-equivalentes [46]. A relação de α-equivalência pode ser melhor vista na seção 8.6.1.

8.7.1 Cuidados com a Substituição

Existe uma necessidade de salientar a introdução da variável nova antes do processo de substituição. Em seguida, dois exemplos são apresentados, sendo que no primeiro caso ocorre um problema pela falta da substituição da variável nova. Pode-se dizer que ocorreu um problema de *captura* pela abstração (a ocorrência *livre* de uma variável, passa a ser *ligada*), como por exemplo:

$$(\lambda x.y\ x)[y \leftarrow w\ x] \neq \lambda x.(y\ x)[y \leftarrow w\ x] = \ldots = \lambda x.w\ x\ x$$

Como dito antes, sem a introdução da variável nova ocorre o problema da *captura pela abstração*, onde y era uma variável *livre* e se tornou uma variável *ligada*.

$$(\lambda x.y\ x)[y \leftarrow w\ x] = (\lambda z.(y\ x)[x/z])[y \leftarrow w\ x] = \lambda z.(y\ z)[y \leftarrow w\ x] = \ldots = \lambda z = w\ x\ z$$

Já neste segundo caso, percebe-se uma significativa diferença, na qual não ocorre a *captura pela abstração*, e sim o resultado esperado, haja vista que ocorreu uma substituição.

8.8 Lambda em Haskell

Nesta seção é apresentada a relação entre o $\lambda-$cálculo e a linguagem de programação Haskell, mostrando como são atribuídas abstrações lambda em Haskell. A mesma começa introduzindo o conceito de *funções anônimas*, as quais são a essência do $\lambda-$cálculo, pois essas são expressões-λ genéricas.

Para evitar a necessidade de definir funções que são passadas como parâmetro, pode-se usar as *funções anônimas*, isto é, expressões-λ. Por exemplo, uma função **g** em Haskell é expressa da seguinte maneira:

[11] No formalismo matemático utilizado, o sinal "$\leftarrow$" significa, no caso do $\lambda-$cálculo, uma substituição, o que em uma linguagem de programação tradicional é uma atribuição.

```
g x = x / (fatorial x)
```

Para evitar uma declaração da função **g**, abstrai-se a mesma, e, como resultado obtém-se uma abstração anônima:

```
\x -> x / (fatorial x)
```

Execução:

```
Main> g 7
0.00138888888888889
Main>  (\x -> x / (fatorial x)) 7
0.00138888888888889
```

Observações:

1. A sintaxe equivalente para o símbolo λ em Haskell é uma barra invertida "\", pois é o sinal de um teclado convencional, que mais se aproxima ao símbolo λ. Assim, uma expressão qualquer, como λw é reescrita em Haskell como `\w`.
2. O separador "." do $\lambda-$cálculo foi substituído pelo aplicador "$\rightarrow$" no Haskell, que indica qual variável é para ser aplicada sob um argumento a sua direita. No caso do exemplo, foi o 7.

Logo, a expressão **g** é representada em termos do $\lambda-$cálculo, atingindo a *transparência referencial*, que é uma das principais idéias a ser alcançada nas linguagens de programação.

Em um segundo exemplo, seja uma abstração-λ definida por: "$\lambda x.\ (x+1)$" equivalente $\lambda x.\ ((+)\ x\ 1) = \lambda x.\ (+)\ x\ 1$. Essa abstração em Haskell é dada por:

```
inc = \x -> (+) x 1
```

Execução:

```
Main> inc 9
10
Main>  (\x -> (+) x 1)  9      -- equivalente a anterior
10
Main>
```

O código apresentado é interpretado como:

1. `inc` : função identificadora `inc` definida por uma soma;
2. `\x ->` : a abstração da variável `x` na expressão;
3. `(+) x 1` : é expressão resultante da aplicação da função `(+)` aos argumentos `x` e `1`. Por uma questão de rigor matemático: *o argumento* `1` *é aplicado à função* `(+)` *com o argumento* `x`.

Nesse paralelo ao Haskell, verifica-se que alguns dos seus elementos têm suas origens no $\lambda-$cálculo, eis a motivação do estudo dessa teoria. No entanto, no Haskell encontram-se numerosos conceitos que não estão presentes no $\lambda-$cálculo [38]. Como características do Haskell que estão ausentes na teoria do $\lambda-$cálculo, salientando[12] que:

[12] Neste capítulo, é estudado o $\lambda-$cálculo sem tipos, o tipado pode ser encontrado em [22].

1. O Haskell dispõe de um mecanismo de tipos que disciplina *a boa aplicação* das funções aos seus argumentos.
2. Como linguagem de programação moderna, o Haskell dispõe de mecanismos que não são encontrados no $\lambda-$cálculo (por exemplo: casamento de padrões, recursividade explícita com o uso do `do`, `where`, `let`, etc.).

A seguir, uma relação das seções anteriores exemplificadas com Haskell:

Exemplo de α-redex:

Em Haskell, duas equações são α-equivalentes se produzirem os mesmos resultados:

```
Main> (\y -> (+)  y  1) 9.999
10.999
Main> (\x -> (+)  x  1) 9.999
10.999
```

Em um segundo exemplo, função apresenta uma particularidade do α-redex, pois o x interno não é substituído na primeira redução por estar protegido pelo x envolvente. Ou seja, a ocorrência interna de x não é livre no corpo da abstração x externa:

```
Main> (\x -> ( \x -> (+) ( (-) x 1) )  x   2.345)    9.876
11.221
```

Fica a cargo do leitor identificar a ordem de como esta redução ocorreu.

Exemplo de β-redex:

Utilizando a linguagem Haskell para ilustrar a aplicação de uma função a um argumento, que neste caso é aplicar uma expressão $(\backslash x \ -> \ x+1)$ na constante 3, então: $(\backslash x -> \ x+1)\ 3$. O processo de cálculo é detalhado como:

```
(\x -> x + 1)  3  =>  (+)  3  1  =>  4
```

Após o exemplo, fica claro o que acontece quando se aplica a função a um argumento, obtendo-se um resultado pela *substituição* na expressão $(+)\ x\ 1$, da variável x por 3.

Em Haskell há um procedimento análogo:

```
Main> (\k -> (\x -> x^k ) (5) )  (2)
25
Main> ( \k -> \x -> x^k ) 5  2
32
```

A análise das expressões dessas em $\lambda-$cálculo é encontrada na seção 8.6.2.

Exemplo de η-redex:

A seguir uma analogia da regra η-redex do $\lambda-$cálculo com a linguagem Haskell:

```
Prelude> (\x->  (+) 1  x )  4
5
Prelude> ( (+)  1 )  4
5
```

8.8.1 Exemplo de Teste de Liberdade

Em uma prática com a linguagem Haskell, seja uma função que testa se uma variável é livre ou se é ligada, o primeiro passo é a representação da expressão-λ definida na seção 8.4, implementada pela definição de um tipo de dado em Haskell, como:

```
data Lam = Var Int | Comb Lam Lam | Abstr Int Lam
```

Esta descrição é dada por:

1. `Lam`: pode ser construída pelo uso do `Var` seguida por um inteiro, o qual é o número da variável, e na seqüência `| Comb Lam Lam | Abstr Int Lam`.
2. `Var`: representa uma variável e um número inteiro, indicando qual é a variável.
3. `Comb`: permite uma combinação de expressões-λ.
4. `Abstr`: permite se construir uma abstração que, neste caso, foi definida por um inteiro e por uma outra expressão-λ (corpo da abstração).

Exemplificando, uma expressão $\lambda x.\ (\lambda y.\ x\ y)$ é definida pela seguinte notação em Haskell:

```
Abstr 1 (Abstr 2 (Comb (Var 1) (Var 2)))
```

A função que define se uma variável é livre ou não, dá-se em duas partes. A primeira é definir se a variável é livre:

```
data Lam = Var Int | Comb Lam Lam | Abstr Int Lam

testa_liberdade a b|livre_ou_ligada a b = "A variavel eh livre"
                   |otherwise           = "A variavel eh ligada"

livre_ou_ligada x (Var y)            =  x == y
livre_ou_ligada x (Comb lamb_1 lamb_2)  = livre_ou_ligada x lamb_1 ||
                                          livre_ou_ligada x lamb_2
livre_ou_ligada x (Abstr int_var lamb) = (x /=  int_var) &&
                                         (livre_ou_ligada x lamb)
livre_ou_ligada     _   _              = False
```

Sejam algumas expressões-λ como "$(\lambda x.\ y)\ (x)$", "$(x)\ (\lambda x.\ y)$", "$(x)\ (\lambda y.\ y)$" e "$(\lambda x.\ x)\ (\lambda y.\ y)$", a execução desses exemplos[13] é dada por:

[13] Vários desses parênteses poderiam ser omitidos, foram mantidos por questões de legibilidade.

Assim, as funções vistas até aqui são as mesmas, a única diferença é que são aplicadas sobre um novo tipo de entrada: *uma figura*. Assim, uma figura é representada por uma lista bidimensional de caracteres, como por exemplo:

```
losango = ["....#....",
           "...###...",
           "..#####..",
           "...###...",
           "....#...."]
```

O tipo desta matriz é de várias lista de caracteres, ou seja `[String]`, `[[Char]]` ou uma lista do tipo `String`, assim `losango :: [String]` ou `losango :: [[Char]]`. Em resumo, basicamente as funções generalizadoras vistas no Capítulo 9 são aplicadas sob este tipo de dados.

10.2 Funções sob Figuras de Caracteres

10.2.1 Considerações Iniciais

Nos primeiros anos escolares, todos aprendem o alfabeto, pela memorização de objetos, como é o caso das vogais 'a', 'e', 'i', 'o' 'u'. Com a união de algumas vogais e consoantes, tem-se a formação de palavras, tais como Haskell. Logo, a manipulação desses objetos é feita com funções que trabalham com listas de `String`.

Como as letras, essas precisam ser concatenadas em uma ordem para se formar uma palavra, os símbolos também precisam ser dispostos em uma seqüência para formar um objeto. Exemplificando: um objeto está para uma figura assim como uma palavra está para uma frase. Para a concatenação de objetos utiliza-se a função `concatena`, dada por:

```
concatena :: [a] -> [a] -> [a]
concatena []  lista_2  =  lista_2
concatena (a:b) lista_2 = a : concatena b lista_2      --  ou = (a:b)++lista_2
```

Execução:

```
Main> concatena "Hoje eh "  " quinta-feira"
"Hoje eh  quinta-feira"
```

Dessa função que concatena caracter, a caracter formando uma lista de caracteres, pode-se criar funções que, por exemplo, agrupem os objetos para formar uma figura. A função `inverte_1` vista no Capítulo 5 é aqui reutilizada, com uso na manipulação de figuras. Assim:

```
inverte_1 :: [a] -> [a]
inverte_1 []     =  []
inverte_1 (x:xs) =  concatena (inverte_1 xs) [x]
```

Execução:

```
Main> inverte_1  [3 .. 9]
```

```
[9,8,7,6,5,4,3]
Main> inverte_1 " .... #### "
" #### ....
Main> inverte_1 ["programacao", " funcional", " HASKELL"]
[" HASKELL"," funcional","programacao"]
```

Nestes exemplos, inicialmente os objetos da lista numérica foram invertidos. Em seguida, listas de objetos do tipo `String` foram invertidas, pois, a partir desse tipo de dados, é que figuras aqui são construídas.

10.2.2 Manipulando Figuras

Esta seção aborda algumas funções para a manipulação de figuras do tipo caracteres. A fim de aplicar essas funções sob figuras bidimensionais, sugere-se a figura da *cabeça de um cavalo* como estudo de caso. Essa figura foi propositalmente escolhida por ter sido originalmente apresentada no livro do Simon Thompson [5]. Assim, o leitor beneficia-se com um *reforço* sobre assunto, presente em muitos *slides* disponíveis na internet sobre Haskell. Essa cabeça de cavalo é dada por:

```
type Figura = [[Char]]   -- [String]
cavalo :: Figura
cavalo = [
          ".......##..." ,
         ".....##..#.." ,
         "...##.....#." ,
         "..#.......#." ,
         "..#...#...#." ,
         "..#...###.#." ,
         ".#....#..##." ,
         "..#...#....." ,
         "...#...#...." ,
         "....#..#...." ,
         ".....#.#...." ,
         "......##...." ]
```

Execução:

```
Main> cavalo
[".......##...",".....##..#..","...##.....#.","..#.......#.",   ...
"..#...#...#.","..#...###.#.",".#....#..##.","..#...#.....",  ...
"...#...#....","....#..#....",".....#.#....",".....##...."]
Main>
```

Utilizando diretamente a função `cavalo` no Haskell, obtém-se um resultado nada agradável. Na definição da função `cavalo`, as linhas estão dispostas uma sob a outra, sendo difícil identificar a figura. Já utilizando uma função para imprimir cada elemento dessa matriz, as quais são linhas de caracteres, as linhas devem ser impressas uma a uma. Ao final de cada linha ou elemento da matriz, deve-se saltar de linha, para, em seguida, imprimir a próxima linha em questão. Desse modo, uma figura qualquer possa ser identificada. Para que a figura da cabeça da cavalo seja visível, é

argumentos. Na função `inverte_horizontal`, o argumento é omitido, mas inferido a partir da definição de `inverte_1`. Logo, essa função tem um único argumento, pois `inverte_1` tem um argumento na sua definição.

Execução:

```
Main> imp_cavalo (inverte_horizontal cavalo)
......##....
.....#.#....
....#..#....
...#...#....
..#...#.....
.#....#..##.
..#...###.#.
..#...#...#.
..#.......#.
...##.....#.
.....##..#..
.......##...
................
Main>
```

Sejam as funções `inverte_horizontal` e `inverte_vertical`, que rotacionam uma figura, em Haskell o resultado de uma função é aplicado a outra função. Reusando essas funções (`inverte_vertical` e `inverte_horizontal`), uma nova função é criada para transladar a figura. Da geometria, uma translação é obtida a partir de uma rotação sob o eixo x, seguida por uma rotação no eixo y. Reescrevendo em Haskell:

```
translada_1 :: Figura -> Figura
translada_1 = inverte_vertical . inverte_horizontal
```

Execução:

```
Main>  imp_cavalo (translada_1 cavalo)
....##......
....#.#.....
....#..#....
....#...#...
.....#...#..
.##..#....#.
.#.###...#..
.#...#...#..
.#.......#..
.#.....##...
..#..##.....
...##.......
................
```

O operador ou *função de composição de funções*, representada pelo símbolo ponto (".") na função `translada_1 = inverte_vertical . inverte_horizontal` tem sua origem na definição matemática da composição de funções, isto é: $f(x).g(x) =$

Para controlar qual nó já foi visitado, é criada uma lista de nós visitados[3], a qual guarda a configuração de todos os nós já visitados/explorados. A função `busca` no código faz uma exploração em todos os nós de uma árvore, a qual é formada a partir de uma busca sobre uma lista passada como parâmetro. A função de usuário a ser chamada é a `jogo_8`, cujo argumento é uma lista com os números seqüenciais de 0 a 8, sem repetições. Essa lista de nove números é uma matriz que representa o estado inicial do tabuleiro. A indexação de cada célula na matriz e acessos aos conteúdos são realizados com as funções utilizadas do capítulo 11. Para o exemplo, veja o código-fonte a seguir, estado inicial que é dado pela lista `[0,1,3,5,2,6,4,7,8]`, na qual o `0` indica al posição `(1,1)` na matriz que está vazia, e assim sucessivamente.

A ação primitiva e permitida para o problema no tabuleiro é movimentar as peças, sem que elas ultrapassem os limites de uma matriz 3×3. A configuração final é representada pela função `lista_final`. A função `mat x` transforma a lista passada pelo usuário como um parâmetro para uma matriz. Ou seja, faz uso da função `listArray` que realiza tal conversão.

A seguir, é apresentado o código-fonte completo do programa que resolve o problema do *quebra-cabeça-8*:

```
module JOGO_8 where

import Array

{-- esta funcao, transforma uma lista para a forma matricial--}
mat x = listArray ((1,1),(3,3)) x

{-Funções que retornam somente lista da tupla, ou semente o pai do
nó, ou somente o nó atual respectivamente-}
lista,pai,nos :: ([Int],Int,Int)->[Int]

lista (a,b,c)  = a
pai   (a,b,c)  = b
nos   (a,b,c)  = c

{-estado final-}
lista_final :: [Int]
lista_final = [1,2,3,4,5,6,7,8,0]

{-x = lista que o usuário deseja encontrar a solução-}
jogo_8 :: [Int] -> IO()
jogo_8 x = imprime2 (caminho(busca [(x,0,0)] [(x,0,0)]))
jogo_8 x = busca [(x,0,0)] [(x,0,0)]

{-Função que recebe uma árvore em forma de lista e
devolve somente o ramo da árvore que encontrou a
solução final-}
caminho (a:b) = reverse (imprime (nos a) (a:b))

imprime a [] = []
imprime a (b:c)
                |a == nos(b) = lista b : imprime (pai b) c
                |otherwise  = imprime a c
```

[3]Neste código, esta função é chamada de `lst_nos_visitados`.

```
busca :: [([Int],Int,Int)] -> [([Int],Int,Int)] -> [([Int],Int,Int)]
busca [] lst_nos_visitados = error "fracasso"
busca (no:subarvore) lst_nos_visitados
             | lista(no)  == lista_final = no:lst_nos_visitados
             | otherwise = busca (subarvore ++ visitados)
                                 (visitados ++ lst_nos_visitados)
                           where
                             visitados = (insere 4 (maior (no:subarvore))
                                                    no lst_nos_visitados)

maior x = foldl (\x y -> if x>y then x else y) 0 (map nos x)

{-- faz uma busca em amplitude nó a nó, caso a lista que possui os
filhos estiver vazia, ocorrerá fracasso --}

{-- insere na lista de filhos, os novos filhos que ainda nao estao
na lista de visitados
insere:: Int -> Int-> ([Int],Int,Int)-> [([Int],Int,Int)] -> [([Int],Int,Int)]
--}
insere 0 y z w = []
insere x y (a,b,c) z  = (filho x (encontra (a,c,y+1)) z) ++
                                        (insere (x-1) (y+1) (a,b,c) z)

{-- cria os filhos caso eles nao pertencam a lista de visitados--}

filho 1 ((a,b),c,d,e) y
        | (a+1>3)                           = []
        | (pertence (elems (c // [((a,b),(c!(a+1,b))),
                                    ((a+1,b),0)])) y) == [] = []
        | otherwise                         = [((elems (c // [((a,b),
                                  (c!(a+1,b))),((a+1,b),0)])),d,e)]

filho 2 ((a,b),c,d,e) y
        | b-1<1                             = []
        | (pertence (elems (c // [((a,b),(c!(a,b-1))),
                                    ((a,b-1),0)])) y) == [] = []
        | otherwise                         = [((elems (c // [((a,b),
                                  (c!(a,b-1))),((a,b-1),0)])),d,e)]

filho 3 ((a,b),c,d,e) y
        | b+1>3                             = []
        | (pertence (elems (c // [((a,b),(c!(a,b+1))),
                                    ((a,b+1),0)])) y) == [] = []
        | otherwise                         = [((elems (c // [((a,b),
                                  (c!(a,b+1))),((a,b+1),0)])),d,e)]

filho 4 ((a,b),c,d,e) y
        | (a-1)<1                           = []
        | (pertence (elems (c // [((a,b),(c!(a-1,b))),
                                    ((a-1,b),0)])) y) == [] = []
        | otherwise                         = [((elems (c // [((a,b),
                                  (c!(a-1,b))),((a-1,b),0)])),d,e)]
```

```
-- ---------------/Funcoes Auxiliares/----------------

{--encontra a posicao em coordenadas do 0(zero) na matriz--}

encontra   (x,y,z)
             | (encontra1 x) ==  1 = ((1,1), mat x,y,z)
             | (encontra1 x) ==  2 = ((1,2), mat x,y,z)
             | (encontra1 x) ==  3 = ((1,3), mat x,y,z)
             | (encontra1 x) ==  4 = ((2,1), mat x,y,z)
             | (encontra1 x) ==  5 = ((2,2), mat x,y,z)
             | (encontra1 x) ==  6 = ((2,3), mat x,y,z)
             | (encontra1 x) ==  7 = ((3,1), mat x,y,z)
             | (encontra1 x) ==  8 = ((3,2), mat x,y,z)
             | otherwise           = ((3,3), mat x,y,z)

{-- funcao auxiliar da funcao acima(encontra) --}

encontra1 [] = error "a lista nao possui 0"
encontra1 (a:x)
            |a == 0    = 1
            |otherwise =  1 + encontra1 x

{-esta funcao compara a lista corrente com a lista de visitados,
para ver se a lista corrente pertence a lista de visitados -}

pertence x  [] = x
pertence x (b:y)
               | x == lista (b) =[]
               | otherwise = pertence x y

-- Fuções de impressão --------------------

imprime2 :: [[Int]]->IO()
imprime2 []   = putStr "fim"
imprime2 (a:x)= do
        {
                 implista 0 a;
                 imprime2 x
        }

implista :: Int -> [Int]->IO()
implista x [a] = do
                     {
                  putStr " " ;
                  putStr (show a);
                  putStrLn "|";
                  putStrLn "-------" ;
                     }
implista 0 a   =   do
            {
                putStrLn "-------";
```

```
                putStr "|";
                implista 1 a
            }

implista x (a:b) |x==3 = do
                {
                putStr " " ;
                putStr (show a);
                putStrLn "|";
                putStr "|";
                implista (1) b
                 }

         | otherwise = do
                     {
                   if x==1 then putStr"" else putStr " ";
                   putStr (show a);
                   implista (x+1) b
                     }
```

Sua execução para um estado inicial representado pela lista `[0,1,3,5,2,6,4,7,8]`, é dada por:

```
JOGO_8> jogo_8 [0,1,3,5,2,6,4,7,8]
-------
|0 1 3|
|5 2 6|
|4 7 8|
-------
-------
|1 0 3|
|5 2 6|
|4 7 8|
-------
-------
|1 2 3|
|5 0 6|
|4 7 8|
-------
-------
|1 2 3|
|0 5 6|
|4 7 8|
-------
-------
|1 2 3|
|4 5 6|
|0 7 8|
-------
-------
|1 2 3|
|4 5 6|
```

funções, etc. vão sendo reconhecidas e reproduzidas no texto, geralmente em outra cor de texto ou fundo. O requisito é que o mesmo sempre salve no formato ASCII (formato texto-padrão) e que apresente o número da linha em que o cursor se encontra. Isto é essencial para a tarefa de depuração. Adicionalmente, um editor de texto que apresente as opções a seguir é desejável:

- ✓ Uma sintaxe com realces aos arquivos com extensão `.hs`. Tanto nos ambientes Linux como MS-Windows, há *scripts* prontos para serem incorporados nos editores.
- ✓ Identação dos arquivos de programas.
- ✓ Interação facilitada com os interpretadores Haskell, tais como Hugs e GHCi.
- ✓ Suporte na navegação do código ao longo do arquivo e dos diversos módulo.

No ambiente Linux, editores como: *jed*, *kate*, *nedit*, o *crimson*, etc. apresentam *scripts* específicos para a linguagem Haskell. Há, igualmente, a mesma disponibilidade destes para o ambiente MS-Windows e outros.

```
Main> or [True,True,True]
True
Main> or [True,False,True]
True
Main> or [False,False,False]
False
```

`any` A função `span` recebe uma booleana como parâmetro, e, a partir desta, percorre-se uma dada lista se ocorrer $n \geq 1$ condições verdadeiras, percorrendo cada elemento da lista a função retorna $True$, senão, no caso de não encontrar nenhuma condição verdadeira, retorna $False$.
Seja uma função definida por:

```
funcao x|x=='s'    = True
        |otherwise = False
```

Tem-se:

```
Main> any funcao "desapropriado"
True
Main> any funcao "titulo"
False
```

`all` Idem à `any`, mas, somente se todas as condições forem verdadeiras, a função retornará $True$, mas se houver $n \geq 1$ condições falsas, a função retornará $False$.
Seja uma função definida por:

```
funcao x|x=='s'    = True
        |otherwise = False
```

Tem-se:

```
Main> all funcao "titulo"
False
Main> all funcao "sssssssssss"
True
```

`zip` Esta função recebe como parâmetro duas listas e devolve uma lista de tuplas na qual estão aglutinados o primeiro elemento das duas listas na primeira tupla, o segundo elemento das duas listas na segunda tupla, e assim por diante, até que uma das listas acabe. O conteúdo restante da próxima lista é perdido.
Ex.:

```
Main> zip "hoje" "ontem"
[('h','o'),('o','n'),('j','t'),('e','e')]
```

`zip3` Idem à função `zip`, mas recebe como parâmetro três listas.
Ex.:

```
Main> zip3 "hoje" "ontem" "amanha"
[('h','o','a'),('o','n','m'),('j','t','a'),('e','e','n')]
```

`unzip` Esta função descompacta a lista formada pelo zip3, pegando os primeiros elementos de todas as tuplas e juntando em uma lista. Depois, pega os segundos elementos de todas as tuplas e coloca em uma segunda lista. Estas duas listas são colocadas em uma tupla.
Ex.:

```
Main> unzip [('h','o'),('o','n'),('j','t'),('e','e')]
("hoje","onte")
```

Haskell
Uma Abordagem Prática

Haskell é uma linguagem funcional, de concepção moderna, cuja base é a fundamentação matemática do cálculo lambda. Apresenta conceitos diferenciais como tipagem polimórfica, avaliação preguiçosa, funções de alta-ordem e sobrecarga de funções. Seu ambiente operacional se baseia em um esquema de carregamento de módulos que apresentam funções específicas com um amplo espectro de aplicações. Possui amplo repertório de tipos de dados embutidos, estruturados sob uma hierarquia de tipos de classes, com dedução automática de tipos de dados, o que facilita a aprendizagem para iniciantes em programação, haja vista que os mesmos não precisam ser especificados pelo programador.

Esta linguagem destaca-se pela sua clareza na codificação, reuso do código e pela pouca, ou quase nenhuma, necessidade de conhecimento prévio de programação. Além destes pontos, a linguagem Haskell é de código aberto, com vários compiladores e interpretadores disponíveis, multiplataforma, além de seu uso estar em franca expansão.

Este livro foi incentivado pela ausência de um material didático destinado a profissionais que desejam aprender a programar desde o nível elementar até o avançado. Assim, um dos objetivos desta publicação é ser utilizada em cursos de programação, tanto em informática e computação, quanto nas demais áreas das ciências exatas, sem esquecer dos curiosos da informática e da computação.

Claudio Cesar de Sá é graduado em Engenharia Elétrica pela Udesc (Universidade do Estado de Santa Catarina), mestre em Engenharia Elétrica pela UFPB (Universidade Federal da Paraíba) e doutor pelo ITA (Instituto Tecnológico de Aeronáutica). Lecionou na área de computação em várias universidades e trabalha na Udesc (Universidade do Estado Santa Catarina).

Márcio Ferreira da Silva é bacharel em Ciência da Computação pela Udesc, cursa pós-graduação de Gestão em Tecnologia da Informação e é desenvolvedor de software na Softexpert, em Joinville.

ISBN 85-7522-095-0

9 788575 220955

www.novatec.com.br
novatec editora