

PROGRAMAÇÃO FUNCIONAL USANDO HASKELL

Programação Funcional

Usando Haskell

Francisco Vieira de Souza

- Licenciado em Matemática (1978) e Engenheiro Civil (1982) pela Universidade Federal do Piauí, Mestre (1994), Doutor (2000) e Pós-doutor (2011) em Ciência da Computação pelo Centro de Informática da Universidade Federal de Pernambuco.
- Professor do Departamento de Matemática (1986-1987), fundador e professor Titular do Departamento de Informática e Estatística, atualmente Departamento de Computação da Universidade Federal do Piauí, a partir de 1987.

UFPI/CCN/DC
Teresina-Piauí
Setembro de 2018

Copyright ©2017, Departamento de Computação,
Centro de Ciências da Natureza,
Universidade Federal do Piauí,
Todos os direitos reservados.

A reprodução do todo ou parte deste trabalho somente
será permitida para fins educacionais e de pesquisa
e com a expressa autorização do autor.

Copyright ©2017, Departamento de Computação,
Centro de Ciências da Natureza,
Universidade Federal do Piauí,
All rights reserved.

Reproduction of all or part of this work only
will be permitted for education or research use
and with expressed permission of the author.

Apresentação

Este livro representa a compilação de vários tópicos da programação funcional desenvolvidos por vários pesquisadores de renome neste campo e tem como objetivo servir de guia aos profissionais da Informática que desejam ter um conhecimento inicial sobre o paradigma funcional de forma geral e, em particular, sobre a programação funcional usando Haskell. A escolha desta linguagem deve-se ao fato de que Haskell tem se tornado a linguagem funcional padrão do discurso, com vários interpretadores e compiladores desenvolvidos para ela, além de várias ferramentas de análises de programas nela codificados (*profiles*).

Para atingir este objetivo, acreditamos que o estudo deva ser acompanhado de algum conhecimento, mesmo que mínimo, sobre a fundamentação e as formas como estas linguagens são projetadas e implementadas. Este conhecimento proporciona ao leitor uma visão das principais características e propriedades do paradigma funcional. Em particular, é importante verificar que as técnicas utilizadas na compilação das linguagens imperativas não se mostraram adequadas na compilação de linguagens funcionais porque o código gerado sempre apresentou um desempenho fraco e abaixo da crítica.

Em 1978, John Backus defendeu o paradigma funcional como o que oferecia a melhor solução para a chamada “*crise do software*”. As linguagens funcionais são apenas uma sintaxe mais cômoda para o λ -cálculo. David Turner [56] mostrou, em 1979, que a Lógica Combinatorial poderia ser estendida de forma a possibilitar a implementação eficiente de linguagens funcionais. Esse trabalho provocou uma corrida em direção à pesquisa nesta área, gerando uma variedade de técnicas de implementação destas linguagens.

Dentre estas técnicas, uma que tem sido adotada, com resultados promissores, consiste na utilização do λ -cálculo como linguagem intermediária entre a linguagem de alto nível e a linguagem de máquina. Os programas codificados em alguma linguagem funcional de alto nível são traduzidos para programas em λ -cálculo e estes são traduzidos para programas em linguagem de máquina. Neste caso, o λ -cálculo desempenha um papel semelhante ao que a linguagem Assembly exerce, como linguagem de montagem, na compilação de linguagens imperativas. Esta metodologia tem dado certo, uma vez que já se conhecem técnicas eficientes de tradução de programas em λ -cálculo para programas executáveis, faltando apenas uma tradução eficiente de programas codificados em uma linguagem funcional de alto nível para programas em λ -cálculo.

Este livro tem início em seu Capítulo introdutório mostrando as características das linguagens funcionais, destacando suas vantagens em relação às linguagens de outros paradigmas que utilizam atribuições destrutivas. Em seguida, é feita uma introdução ao λ -cálculo. Apesar do caráter introdutório, achamos ser suficiente para quem quer dar os primeiros passos em direção à aprendizagem desta técnica. Os Capítulos subsequentes se referem todos à Programação Funcional usando Haskell.

Por ser uma primeira edição, o livro contém erros e sua apresentação didático-pedagógica pode e deve ser revista. Neste sentido, agradecemos críticas construtivas que serão objeto de análise e reflexão e, por isto mesmo, muito bem-vindas.

Teresina-Pi, Março de 2017.
Francisco Vieira de Souza
E-mail: vieira.ufpi@gmail.com

Sumário

1	Introdução	1
1.1	Princípios das linguagens de programação	3
1.2	O processo de implementação das linguagens de programação	6
1.3	A implementação de linguagens funcionais	7
1.4	Máquinas abstratas	8
1.5	Por que escolher Haskell?	9
1.6	Haskell na Indústria	10
1.7	A comunidade de Haskell	11
1.8	Resumo	11
1.9	Composição deste livro	12
2	Programação funcional	15
2.1	Introdução	15
2.2	Computabilidade de funções	16
2.3	Análise de dependências	17
2.4	Funções e expressões aplicativas	18
2.5	Independência da ordem de avaliação	19
2.6	Transparência referencial	22
2.7	Interfaces manifestas	23
2.8	Definição de funções	24
2.8.1	Definições explícitas e implícitas de variáveis	24
2.8.2	Funções totais e funções parciais	26
2.8.3	Definições explícitas e implícitas de funções	26
2.8.4	Definições de funções por enumeração	27
2.8.5	Definição de funções por intencionalidade	28
2.8.6	Definição de funções por composição	28
2.8.7	Definição de funções por casos	29
2.8.8	Definição de funções por recursão	29
2.9	Resumo	31
2.10	Exercícios propostos	31

3	λ-cálculo	33
3.1	Introdução	33
3.2	λ -expressões	34
3.3	A sintaxe do λ -cálculo	35
3.3.1	Funções e constantes predefinidas	35
3.3.2	λ -abstrações	36
3.3.3	Aplicação de função e curificação	36
3.4	A semântica operacional do λ -cálculo	37
3.4.1	Ocorrências livres ou ligadas	38
3.4.2	Combinadores	39
3.5	Regras de conversão entre λ -expressões	41
3.5.1	α -conversão	41
3.5.2	β -conversão	41
3.5.3	η -conversão	42
3.5.4	Convertibilidade de λ -expressões	42
3.5.5	Captura	43
3.5.6	Provando a conversibilidade	45
3.5.7	Uma nova notação	45
3.6	Ordem de redução	46
3.7	Funções recursivas	48
3.8	Algumas definições	50
3.9	Resumo	51
3.10	Exercícios propostos	52
4	Programação funcional em Haskell	53
4.1	Introdução	53
4.1.1	Outras implementações	54
4.2	Primeiros passos	55
4.2.1	O interpretador ghci	57
4.2.2	Identificadores em Haskell	58
4.3	Funções em Haskell	59
4.3.1	Construindo funções	60
4.3.2	Avaliação de funções em Haskell	62
4.3.3	Casamento de padrões (patterns matching)	62
4.4	Tipos de dados em Haskell	63
4.4.1	O tipo inteiro (Int ou Integer)	63
4.4.2	O tipo ponto flutuante (Float ou Double)	67
4.4.3	O tipo booleano (Bool)	68

4.4.4	O tipo caractere (Char)	69
4.4.5	O tipo cadeia de caracteres (String)	70
4.4.6	Metodologias de programação	71
4.4.7	Os tipos de dados estruturados de Haskell	74
4.4.8	Escopo	75
4.4.9	Expressões condicionais	76
4.4.10	Avaliação em Haskell	77
4.5	Projeto de programas	79
4.5.1	Provas de programas	80
4.6	Resumo	83
4.7	Exercícios propostos	83
5	O tipo Lista	85
5.1	Introdução	85
5.2	Funções sobre listas	86
5.3	Pattern matching revisado	90
5.4	Compreensões e expressões ZF (Zermelo-Fraenkel)	93
5.5	Funções de alta ordem	98
5.5.1	A função map	99
5.5.2	Funções anônimas	101
5.5.3	As funções fold e foldr	102
5.5.4	A função filter	103
5.6	Polimorfismo	104
5.6.1	Tipos variáveis	105
5.6.2	O tipo mais geral	105
5.7	Indução estrutural	106
5.8	Composição de funções	108
5.8.1	Composição normal de funções	109
5.8.2	Composição avançada	111
5.8.3	Esquema de provas usando composição	111
5.9	Aplicação parcial	113
5.9.1	Seção de operadores	114
5.10	Algoritmos de ordenação	121
5.10.1	Ordenação por inserção	121
5.10.2	Ordenação por seleção	122
5.10.3	Ordenação por trocas	123
5.10.4	Ordenação otimizada	123
5.11	Resumo	125

5.12	Exercícios propostos	126
6	Tipos de dados algébricos	129
6.1	Introdução	129
6.2	Classes de tipos	129
6.2.1	Fundamentação das classes	130
6.2.2	Funções que usam igualdade	131
6.2.3	Assinaturas e instâncias	132
6.2.4	Classes derivadas	133
6.2.5	As classes predefinidas em Haskell	134
6.3	Tipos algébricos em Haskell	136
6.3.1	Como se define um tipo algébrico?	137
6.3.2	Tipos recursivos	140
6.3.3	Tipos algébricos polimórficos	141
6.4	Árvores	143
6.4.1	Árvores binárias	143
6.4.2	Funções sobre árvores binárias	143
6.4.3	Árvores binárias aumentadas	145
6.4.4	Árvores de buscas binárias	147
6.4.5	Árvores heap binárias	149
6.4.6	Árvores Rosas	152
6.4.7	Árvores AVL	157
6.5	Tratamento de exceções	162
6.5.1	Valores fictícios	163
6.5.2	Tipos de erros	164
6.6	Lazy evaluation	166
6.6.1	Expressões ZF (revisadas)	168
6.6.2	Dados sob demanda	170
6.6.3	Listas potencialmente infinitas	170
6.7	Provas sobre tipos algébricos	172
6.8	Resumo	173
6.9	Exercícios propostos	174
7	Tipos de dados abstratos	175
7.1	Introdução	175
7.2	Módulos em Haskell	176
7.2.1	Controles de exportação	177
7.2.2	Importação de módulos	177
7.2.3	Controles de importação	177

7.3	Modularidade e abstração de dados	178
7.3.1	O que é mesmo um tipo abstrato de dados?	178
7.4	Eficiência de programas funcionais	179
7.4.1	O desempenho da função reverse	179
7.5	Implementação de tipos abstratos de dados em Haskell	180
7.5.1	O tipo abstrato Pilha	181
7.5.2	O tipo abstrato de dado Fila	183
7.5.3	O tipo abstrato Set	186
7.5.4	O tipo abstrato Tabela	187
7.6	Arrays	188
7.6.1	A criação de arrays	189
7.6.2	Utilizando arrays	190
7.7	Resumo	190
7.8	Exercícios propostos	191
8	Programação com ações em Haskell	193
8.1	Introdução	193
8.1.1	Ser ou não ser pura: eis a questão	194
8.2	Entrada e Saída em Haskell	195
8.2.1	Operações de entrada	196
8.2.2	Operações de saída	196
8.2.3	O comando do	197
8.3	Arquivos, canais e descritores	199
8.3.1	A necessidade dos descritores	200
8.3.2	Canais	202
8.4	Gerenciamento de exceções	203
8.5	Resumo	209
A	Algumas funções padrões	215
B	Compilação e execução de programas em Haskell	221
B.1	Introdução	221
B.2	Baixando e instalando o GHC	221
B.3	Compilando em GHC	221
B.3.1	Passando parâmetros para um programa executável	222
B.3.2	Diretivas de compilação	222

Capítulo 1

Introdução

*“Recently Haskell was used in an experiment here
at Yale in the Medical School.
It was used to replace a C program that controlled a heart-lung machine.
In the six months that it was in operation,
the hospital estimates that probably a dozen lives
were saved because the program was far more robust than C program,
which often crashed and killed the patients. ”*
(Paul Hudak in [16])

Até os anos 80, os programas de computadores eram pequenos porque os problemas que eles resolviam computacionalmente também eram pequenos. Este tipo de programação ficou conhecido como “*programming in the small*”. A melhoria de desempenho verificada nos computadores possibilitou que problemas bem maiores e desafiantes pudessem ser solucionados por eles, o que não era possível ser realizado com as máquinas anteriores. Como consequência, os programas passaram a ser bem maiores e mais complexos que os anteriores, caracterizando um novo tipo de programação que ficou conhecido como “*programming in the large*”.

Este novo tipo de programação trouxe, como consequência, a necessidade de novos métodos de construção de software. Por serem grandes, os programas passaram a ser construídos, não apenas por uma única pessoa, mas por grupos ou times de pessoas, possivelmente trabalhando paralelamente em locais distintos. Foi necessário desenvolver novos métodos de trabalho em grupo que proporcionassem a construção segura e eficiente destes *softwares*. Neste caso, a palavra chave passou a ser “*produtividade*”. Este movimento ficou conhecido como a “*crise do software dos anos 80*” e a solução encontrada para resolvê-lo ficou conhecida como “*programação estruturada*”.

Para entender e analisar as similaridades existentes entre a programação estruturada e a programação funcional, faz-se necessário conhecer os fundamentos nos quais se baseiam estas duas formas de se construir programas. Vamos iniciar lembrando os princípios nos quais se baseia a programação estruturada. **Hoare enumerou seis princípios fundamentais da estruturação de programas** [30]:

1. **Transparência de significado.** Este princípio afirma que o significado de uma expressão, como um todo, pode ser entendido em termos dos significados de suas sub-expressões. Assim, o significado da expressão $E + F$ depende simplesmente dos significados das sub-expressões E e F , independente das complicações de cada uma delas.
2. **Transparência de propósitos.** Textualmente, Hoare argumenta: “o propósito de cada parte consiste unicamente em sua contribuição para o propósito do todo”. Assim, em

$E + F$, o único propósito de E é computar o número que será o operando esquerdo do operador “+”, valendo o mesmo para o operando F . Isto significa que seu propósito não inclui qualquer efeito colateral.

3. **Independência das partes.** Este princípio apregoa que os significados de duas partes, não sobrepostas, podem ser entendidos de forma completamente independente, ou seja, na avaliação de uma expressão $E + F$, a expressão E pode ser entendida independentemente da expressão F e vice versa. Isto acontece porque o resultado computado por um operador depende apenas dos valores de suas entradas.
4. **Aplicações recursivas.** Este princípio se refere ao fato de que as expressões aritméticas são construídas pela aplicação recursiva de regras uniformes. Isto significa que se nós sabemos que E e F são expressões, então sabemos que $E + F$ também é uma expressão.
5. **Interfaces pequenas.** As expressões aritméticas têm interfaces pequenas, porque cada operação aritmética tem apenas uma saída e apenas uma ou duas entradas. Além disso, cada uma das entradas e saídas é um valor simples. Assim, a interface entre as partes é clara, pequena e bem controlada.
6. **Estruturas manifestas.** Este princípio se refere ao fato de que os relacionamentos estruturais entre as partes de uma expressão aritmética seja óbvio. Uma expressão é uma subexpressão de uma outra expressão se estiver textualmente nela circunscrita. Também duas expressões não estão estruturalmente relacionadas se elas não se sobrepuzarem de alguma forma.

Estas características são verdadeiras em relação à programação estruturada, no entanto, nem sempre elas são assim entendidas. A este respeito, John Hughes [17] fez algumas reflexões, analisando a importância da programação estruturada, fazendo um paralelo com as linguagens funcionais. Em sua análise, ele cita que quando se pergunta a alguém o que é programação estruturada, normalmente se tem, uma ou mais, das seguintes respostas:

- é uma programação sem **gotos**,
- é uma programação onde os blocos não têm entradas ou saídas múltiplas ou
- é uma programação onde os programas nela escritos são mais tratáveis matematicamente.

Apesar de todas as respostas acima serem corretas, no que se refere à caracterização deste tipo de programação, elas não são conclusivas. Aludem ao que a programação estruturada não é, mas não dizem o que realmente é a programação estruturada. Na realidade, uma resposta coerente para a pergunta sobre a programação estruturada pode ser: “**programas estruturados são programas construídos de forma modular**”.

Esta é uma resposta afirmativa e que atinge o cerne da questão. A construção de programas modulares é responsável pela grande melhoria na construção de software, sendo a técnica responsável pelo notório aumento de produtividade de *software* que ultimamente tem se verificado. E por que a modularidade é tão determinante? A resposta é imediata: na modularidade, os problemas são decompostos em problemas menores e as soluções para estes subproblemas são mais fáceis de serem encontradas. No entanto, estas pequenas soluções devem ser combinadas para representar uma solução para o problema original como um todo. Modula II, Ada, Pascal, C, C++, Standard ML, Haskell, Java, Eiffel e todas as modernas linguagens de programação, independente do paradigma ao qual pertençam, foram projetadas ou adaptadas depois para serem modulares.

O aumento de produtividade na programação estruturada se verifica porque:

- módulos pequenos podem ser codificados mais facilmente e mais rapidamente,
- módulos de propósito geral podem ser reutilizados, ou seja, maior produtividade e
- os módulos podem ser testados e compilados de forma independente, facilitando muito a depuração dos programas.

Vamos agora conhecer a programação funcional para podermos estabelecer uma comparação entre ela e a programação estruturada. John Hughes também comenta uma situação semelhante, quando se pergunta a alguém sobre o que é uma linguagem funcional. Para ele, normalmente, se tem como resposta uma ou mais das seguintes alternativas:

- é uma linguagem onde os programas nela codificados consistem inteiramente de funções,
- é uma linguagem que não tem *side effects*,
- é uma linguagem em que a ordem de execução é irrelevante, ou seja, não precisa analisar o fluxo de controle,
- é uma linguagem onde pode-se substituir, a qualquer tempo, variáveis por seus valores (transparência referencial) ou
- é uma linguagem cujos programas nela escritos são mais tratáveis matematicamente.

Estas respostas também não são conclusivas, da mesma forma que as respostas dadas à questão sobre o que é programação estruturada. E qual seria uma resposta afirmativa e definitiva, neste caso? Usando a resposta anterior como base, pode-se afirmar que “as linguagens funcionais são altamente modularizáveis”.

Esta é uma resposta afirmativa, precisando apenas de um complemento para que ela fique completa. Este complemento se refere às características que as linguagens funcionais apresentam e que são responsáveis por esta melhoria na modularidade dos sistemas. Estas características podem ser sumarizadas na seguinte observação: “a programação funcional melhora a modularidade, provendo módulos menores, mais simples e mais gerais, através das funções de alta ordem e do mecanismo de *lazy evaluation*”.

Estas duas características, verificadas apenas nas linguagens funcionais, é que são responsáveis pela grande modularidade por elas proporcionada. Dessa forma, as linguagens funcionais são altamente estruturadas e, portanto, representam uma solução adequada para a tão propalada “crise do software” dos anos 80.

1.1 Princípios das linguagens de programação

Em 2004, Philip Wadler¹ escreveu o artigo “*Why no one uses functional languages*”, onde ele comenta sobre a pouca utilização das linguagens funcionais na Indústria e ambientes comerciais [59]. Para ele, dizer que ninguém usa linguagem funcional, é um exagero. As chamadas telefônicas no Parlamento Europeu são roteadas por um programa escrito em Erlang, a linguagem funcional da Ericsson. A rede Cornell distribui CDs virtuais usando o sistema Esemble, escrito em CAML e a Polygram vende CDs na Europa utilizando Natural Expert, da Software AG. As linguagens Erlang (www.erlang.se) e ML Works de Harlequin (www.harlequin.com) apresentam um extensivo ambiente de suporte ao usuário. Além disso, as linguagens funcionais são mais

¹Philip Wadler trabalha nos grupos de ML e de Unix na Bell Labs. Ele é co-autor das linguagens Haskell e GJ. Além de vários artigos publicados, ele também é co-editor da revista Journal of Functional Programming.

adequadas à construção de provadores de teoremas, incluindo o sistema HOL que foi utilizado na depuração do projeto de multiprocessadores da linha HP 9000.

Ainda segundo Wadler, as linguagens funcionais produzem um código de máquina com uma melhoria de uma ordem de magnitude e, nem sempre, estes resultados são mostrados. Normalmente, se mostram fatores de 4. Mesmo assim, um código que é quatro vezes menor, quatro vezes mais rápido de ser escrito ou quatro vezes mais fácil de ser mantido não pode ser jogado fora. Para ele, os principais fatores que influenciam na escolha de uma linguagem de programação são:

- **Compatibilidade.** Atualmente, os sistemas não são mais construídos monoliticamente, como eram no passado. Os programas se tornaram grandes (*programming in the large*) e agora eles devem ser escritos de forma modular por programadores em tempos e locais possivelmente distintos, devendo serem ligados através de interfaces bem definidas. É necessário acabar com o isolamento das linguagens funcionais e incorporar a elas facilidades para a comunicação entre programas funcionais e programas codificados em outras linguagens pertencentes a outros paradigmas. A Indústria da Computação está começando a distribuir padrões como CORBA e COM para suportar a construção de software a partir de componentes reutilizáveis. Algumas linguagens funcionais já apresentam facilidades para a construção de grandes softwares, com formas bem adequadas para a definição de módulos, apesar de algumas delas ainda não oferecem estas facilidades. Atualmente, os programas em Haskell já podem ser empacotados como um componente COM e qualquer componente COM pode ser chamado a partir de Haskell.
- **Bibliotecas.** Muitos usuários escolheram Tcl, atraídos, principalmente, pela biblioteca gráfica Tk. Muito pouco da atratividade de Java tem a ver com a linguagem em si, e sim com as bibliotecas associadas, usadas na construção de gráficos, banco de dados, interfaceamento, telefonia e servidores. Apesar de ainda não existirem muitas bibliotecas gráficas para as linguagens funcionais, muito esforço tem sido feito nesta direção, nos últimos tempos. Haskell tem Fudgets, Gadgets, Haggis, HOpenGL e Hugs Tk. SML/NJ tem duas: eXene e SML Tk. Haskell e ML têm, ambas, um poderoso sistema de módulos que tornam suas bibliotecas fáceis de serem construídas, já tendo a biblioteca Edison com estruturas de dados eficientes, construída por Okasaki [33] e mantida por Robert Dockins. Haskell ainda tem HSQL com interfaces para uma variedade de Bancos de Dados, incluindo MySQL, Postgres, ODBC, SQLite e Oracle. Haskell ainda tem Happy, um gerador de parsers LALR, similar ao yacc, atualmente estendido para produzir parsers LR para gramáticas ambíguas.
- **Portabilidade.** Inegavelmente C e C++ têm sido preferidas em muitos projetos. No entanto, muito desta preferência não se deve ao fato de C gerar um código mais rápido que o código gerado pelas linguagens funcionais, apesar de, normalmente, se verificar esta diferença de desempenho. Na realidade, esta preferência se deve mais à portabilidade inegável de C. Sabe-se que os pesquisadores em Lucent preferiam construir a linguagem PRL, para Banco de Dados, usando SML, mas escolheram C++, porque SML não estava disponível no mainframe Amdahl, onde deveria ser utilizada. Por outro lado, as técnicas de implementação de linguagens utilizando máquinas abstratas têm se tornado muito atrativas para linguagens funcionais [26] e também para Java. Isto se deve muito ao fato de que escrever a máquina abstrata em C a torna muito mais fácil de ser portada para uma grande variedade de arquiteturas.
- **Disponibilidade.** Alguns compiladores são muito difíceis de serem instalados. Por exemplo, a instalação de **GHC** (*Glasgow Haskell Compiler*) era considerado uma aventura para quem tentasse fazê-lo. Ainda existem poucas linguagens funcionais comerciais e isto torna

difícil um código estável e um suporte confiável. Além do mais, as linguagens funcionais estão em permanente desenvolvimento e, portanto, estão sempre em transformações. No entanto, este quadro tem se modificado, pelo menos em relação a Haskell. Em 1998, foi adotado o padrão Haskell98 que permanece inalterado até o momento, apesar de continuar a serem incorporadas novas extensões à sua biblioteca. Atualmente, a instalação dos compiladores Haskell é uma tarefa possível de ser feita sem qualquer trauma estando disponível para várias plataformas.

- **Empacotamento.** Muitas linguagens funcionais seguem a tradição de LISP, de sempre realizar suas implementações através do loop *read-eval-print*. Apesar da conveniência, é essencial desenvolver habilidades para prover alguma forma de conversão de programas funcionais em programas de aplicação *standalone*. Muitos sistemas já oferecem isto, no entanto, incorporam o pacote de runtime completo à biblioteca e isto implica na exigência de muita memória.
- **Ferramentas.** Uma linguagem para ser utilizável necessita de ferramentas para depuração e profiler. Estas ferramentas são fáceis de serem construídas para linguagens estritas, no entanto, são muito difíceis de serem construídas para linguagens *lazy*, onde a ordem de avaliação não é conhecida a priori. Verifica-se uma exceção em Haskell, onde muitas ferramentas de profiler já estão disponíveis e muitas outras estão em pleno desenvolvimento.
- **Treinamento.** Para programadores imperativos é muito difícil programar funcionalmente. Uma solução imperativa é mais fácil de ser entendida e de ser encontrada em livros ou artigos. Uma solução funcional demora mais tempo para ser criada, apesar de muito mais elegante. Por este motivo, muitas linguagens funcionais atuais provêem um escape para o estilo imperativo. Isto pode ser verificado em ML que não é considerada uma linguagem funcional pura, porque permite atribuições destrutivas. Haskell é uma linguagem funcional pura, mas consegue imitar as atribuições das linguagens imperativas utilizando uma teoria funcional complexa que é a semântica de ações, implementadas através de mônadas²
- **Popularidade.** Se um gerente escolher uma linguagem funcional para ser utilizada em um projeto e este falhar, provavelmente ele será crucificado. No entanto, se ele escolher C ou C++ e não tiver sucesso, tem a seu favor o argumento de que o sucesso de C++ já foi verificado em inúmeros casos e em vários locais. Este quadro também vem se modificando em relação às linguagens funcionais, principalmente Haskell, onde ela tem se popularizado muito nos últimos anos.
- **Desempenho.** Há uma década atrás, os desempenhos dos programas funcionais eram bem menores que os dos programas imperativos, mas isto tem mudado muito ultimamente. Hoje, os desempenhos de muitos programas funcionais são melhores ou pelo menos estão em “pés de igualdade” com seus correspondentes em C. Isto depende da aplicação. Java tem uma boa aceitação e, no entanto, seu desempenho é muito inferior a C, na grande maioria das aplicações. Na realidade, existem linguagens com alto desempenho que não são muito utilizadas e existem linguagens com desempenho mediano com alta taxa de utilização. Desempenho é um fator importante, mas não tem se caracterizado como um fator decisivo na escolha de uma linguagem.

Para continuar esta viagem pelo mundo da programação funcional, tentando entender sua fundamentação teórica, é necessário conhecer também como as linguagens são implementadas. Inicialmente, será analisada a implementação de linguagens tradicionais e depois esta análise se particulariza para o caso da implementação das linguagens funcionais.

²A semântica de ações é um tema tratado no Capítulo 7. Os mônadas são estruturas matemáticas da teoria dos grupos e são tratados no Capítulo 8 deste livro.

1.2 O processo de implementação das linguagens de programação

Programar é modelar problemas do mundo real ou imaginário através de uma linguagem de programação, onde o resultado seja um programa que seja executado em um computador. Para isso, os problemas devem ser modelados em um nível de abstração bem mais alto através de especificações formais, feitas utilizando uma linguagem de especificação formal. Existem várias linguagens de especificação formal: Lotos, Z, VDM, Redes de Petri, entre outras. A escolha de uma delas está diretamente ligada ao tipo da aplicação e à experiência do programador. Por exemplo, para especificar dispositivos de hardware é mais natural se usar Redes de Petri, onde pode-se verificar a necessidade de sincronização e podem ser feitas simulações para a análise de desempenho ou detectar a existência, ou não, de inconsistências.

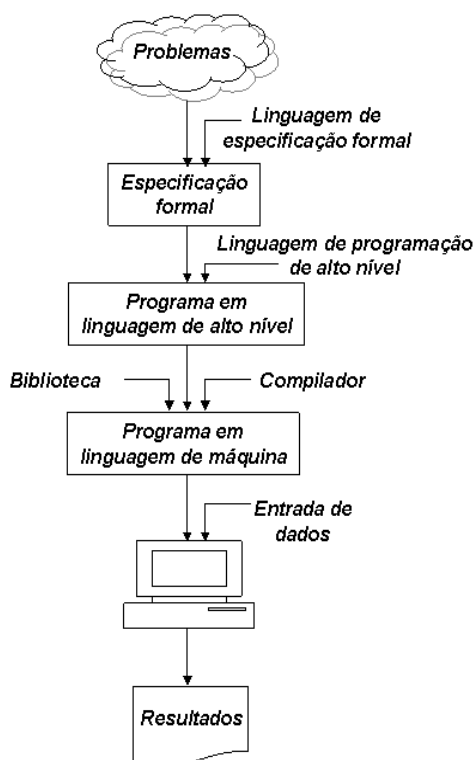


Figura 1.1: Esquema de solução de problemas através de computadores.

Muitas ferramentas gráficas já existem e são utilizadas na análise de desempenho de dispositivos especificados formalmente e podem ser feitos protótipos rápidos para se verificar a adequabilidade das especificações às exigências do usuário, podendo estes protótipos ser parte integrante do contrato de trabalho entre o programador e o contratante. Mais importante que isto, a especificação formal representa uma prova da correção do programa e que ele faz exatamente o que foi projetado para fazer. Isto pode ser comparado com a garantia que um usuário tem quando compra um eletrodoméstico em uma loja. Esta garantia não pode ser dada pelos testes, uma vez que eles só podem verificar a presença de erros, nunca a ausência deles. Os testes representam uma ferramenta importante na ausência de uma prova da correção de um programa, mas não representam uma prova. O uso de testes requer que eles sejam bem projetados e de forma objetiva, para que tenham a sua existência justificada. Com a especificação

pronta, ela deve ser implementada em uma linguagem de programação.

A linguagem de programação a ser escolhida depende da aplicação. Em muitos casos, este processo é tão somente uma tradução, dependendo da experiência do programador com a linguagem de programação escolhida. No caso das linguagens funcionais, este processo é muito natural, uma vez que as especificações formais são apenas definições implícitas de funções, restando apenas a tradução destas definições implícitas para definições explícitas na linguagem funcional escolhida³. Resta agora a tradução destes programas em linguagens de alto nível para programas executáveis em linguagem de máquina, sendo este o papel do compilador. Este processo está mostrado na Figura 1.1.

A compilação de programas codificados em linguagens imperativas normalmente é feita em duas etapas [2]. Na primeira delas, é feita uma tradução do programa escrito em linguagem de alto nível para um programa codificado em linguagem intermediária, chamada de linguagem de montagem (Assembly). Na etapa seguinte, é feita a tradução do programa em linguagem de montagem para o programa executável, em linguagem de máquina. Este processo está mostrado na Figura 1.2.

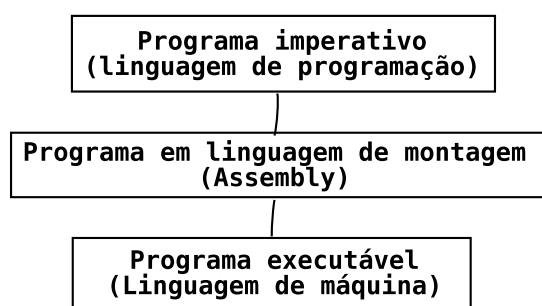


Figura 1.2: Esquema de compilação das linguagens imperativas.

Esta mesma metodologia foi tentada por alguns pesquisadores na tradução de programas codificados em linguagens funcionais para programas em linguagem de máquina, mas os resultados não foram animadores [27]. Os códigos executáveis gerados eram todos de baixo desempenho. Por este motivo, os pesquisadores da área de implementação de linguagens funcionais se viram obrigados a buscar outras alternativas de compilação para estas linguagens.

1.3 A implementação de linguagens funcionais

Como já mencionado anteriormente, as linguagens funcionais apresentam características importantes, como as **funções de alta ordem**, **polimorfismo** e **lazy evaluation**. Estas características são proporcionadas por particularidades apresentadas pela **programação aplicativa** ou **funcional**. Estas particularidades são: a transparência referencial, a propriedade de Church-Rosser, a independência na ordem de avaliação e as interfaces manifestas, que serão objeto de estudo no Capítulo 1 deste livro. Para que estas particularidades estejam presentes, é necessário que, durante a execução, muitas estruturas permaneçam ativas na heap para que possam ser utilizadas mais tarde. Estas características têm dificultada a criação de códigos executáveis enxutos e de bons desempenhos.

Por este motivo, outras técnicas de implementação foram pesquisadas. Uma que tem apresentado resultados promissores consiste na tradução de programas codificados em linguagens funcionais para programas em uma linguagem intermediária, como na tradução das linguagens

³Estas formas de definição de funções serão mostradas no Capítulo 1 deste livro.

imperativas, mas utilizando λ -cálculo⁴ como linguagem intermediária, em vez da linguagem Assembly [26]. Já existem métodos eficientes de tradução de programas codificados no λ -cálculo para programas em linguagem de máquina. Dessa forma, o problema agora se restringe à tradução dos programas escritos nas linguagens funcionais para programas codificados em λ -cálculo. Este processo está mostrado na Figura 1.3.

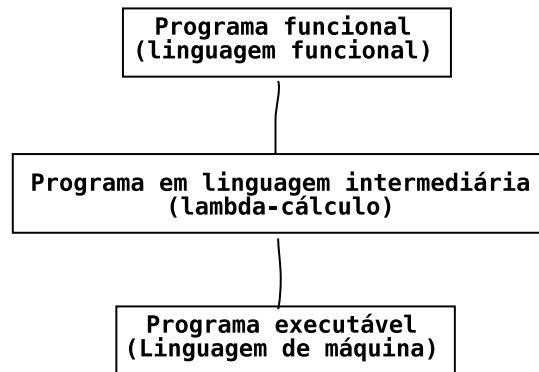


Figura 1.3: Um esquema de compilação das linguagens funcionais.

A escolha do λ -cálculo como linguagem intermediária entre as linguagens funcionais e a linguagem de máquina se deve a dois fatores [37]:

1. o λ -cálculo é uma linguagem simples, com poucos construtores sintáticos e semânticos e
2. o λ -cálculo é uma linguagem suficientemente poderosa para expressar todos os programas funcionais.

1.4 Máquinas abstratas

Uma técnica que tem sido usada com sucesso na tradução de programas codificados em linguagens funcionais para programas codificados em λ -cálculo de alto desempenho tem sido a transformação das expressões iniciais em linguagens funcionais em expressões equivalentes em λ -cálculo. Estas λ -expressões iniciais, possivelmente, não apresentem um desempenho muito bom, mas serão transformadas em λ -expressões mais simples, até atingir uma forma normal, se ela existir. O resultado desta técnica foi um sistema que ficou conhecido na literatura como máquinas abstratas. A primeira máquina abstrata foi **SECD**(**S**ta**C**k, **E**nvironment, **C**ode and **D**ump) desenvolvida por Peter Landin em 1964 [24]. Ela usa a pilha **S** para avaliar as λ -expressões codificadas, utilizando o ambiente **E**.

Uma otimização importante na máquina **SECD** foi transferir alguma parcela do tempo de execução para o tempo de compilação. No caso, isto foi feito transformando as expressões com notação infixa para a notação polonesa reversa que é adequada para ser executada em pilha, melhorando o ambiente de execução. Para diferenciar da máquina **SECD**, esta máquina foi chamada de **SECD2**.

Em 1979, David Turner [56] desenvolveu um processo de avaliação de expressões em **SASL** (uma linguagem funcional de sua autoria) usando o que ficou conhecida como a máquina de combinadores. Ele utilizou os combinadores **S**, **K** e **I** do λ -cálculo para representar expressões, em vez de λ -expressões, utilizando para isto um dos combinadores acima citados, sem variáveis

⁴ λ -cálculo é uma teoria de funções que será vista no Capítulo 2 deste livro.

livres [11]. Turner traduziu diretamente as expressões em **SASL** para combinadores, sem passar pelo estágio intermediário das λ -expressões. A máquina de redução de Turner foi chamada de **Máquina de Redução SK** e utilizava a redução em grafos como método para sua implementação. Esta máquina teve um impacto muito intenso na implementação de linguagens aplicativas, pelo ganho em eficiência, uma vez que ela não utilizava o ambiente da máquina **SECD**, mas tirava partido do compartilhamento que conseguia nos grafos de redução.

Um outro pesquisador que se tornou famoso no desenvolvimento de máquinas abstratas foi Johnsson, a partir de 1984, quando ele deu início a uma série de publicações [20, 21, 22] sobre este tema, culminando com sua Tese de Doutorado, em 1987 [23], onde ele descreveu uma máquina abstrata baseada em supercombinadores que eram combinadores abstraídos do programa de usuário para algumas necessidades particulares. A máquina inventada por Johnsson ficou conhecida pela **Máquina G** e se caracterizou por promover uma melhoria na granularidade dos programas, que era muito fina na máquina de redução de Turner. Uma otimização importante desta máquina foi a utilização de uma segunda pilha na avaliação de expressões aritméticas ou outras expressões estritas.

Várias outras máquinas abstratas foram construídas com bons resultados. Entre elas podem ser citadas a máquina **GMC** e a máquina Γ , idealizadas por Rafael Lins, da Universidade Federal de Pernambuco [45].

Além destas, uma que tem se destacado com excelentes resultados é a máquina Γ CMC, também idealizada por Rafael Lins [26], onde um programa codificado em **SASL** é traduzido para um programa em Ansi C. Este processo está mostrado na Figura 1.4.

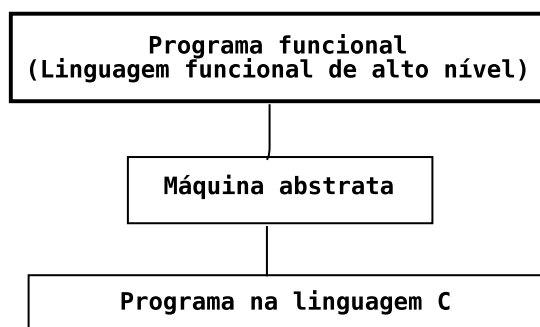


Figura 1.4: O esquema de compilação de linguagens funcionais adotado em Γ CMC.

A escolha da linguagem C se deve ao fato de que os compiladores de C geram códigos reconhecidamente portáteis e eficientes. A máquina Γ CMC é baseada nos combinadores categóricos que se fundamentam na Teoria das Categorias Cartesianas Fechadas, recentemente utilizada em diversas áreas da Computação, sendo hoje um tema padrão do discurso, nos grandes encontros e eventos na área da Informática [46].

1.5 Por que escolher Haskell?

Resumidamente, para ser bem utilizada, uma linguagem deve suportar trabalho interativo, possuir bibliotecas extensas, ser altamente portátil, ter uma implementação estável e fácil de ser instalada, ter depuradores e profilers, ser acompanhada de cursos de treinamentos e já ter sido utilizada, com sucesso, em uma boa quantidade de projetos.

Para Philip Wadler [59], todos estes requisitos já são perfeitamente atendidos por algumas linguagens funcionais, por exemplo Haskell. Para ele, o que ainda existe é um preconceito injustificável por parte de alguns programadores de outros paradigmas de programação, apesar

de que este quadro tem se modificado vertiginosamente ao longo dos últimos anos.

Na realidade, Haskell oferece aos usuários três características principais: novidade, poder e prazer [34]:

- **Novidade.** Haskell, provavelmente, seja diferente de qualquer linguagem utilizada pelo leitor. por oferecer uma nova forma de pensar sobre software. Em Haskell, se enfatiza o uso de funções que tomam valores imutáveis como entrada e produzem novos valores como saída, ou seja, sendo dadas as mesmas entradas, as funções devolvem sempre os mesmos resultados.
- **Poder.** Como um código puro não tem envolvimento com o mundo externo, porque as funções respondem apenas a entradas visíveis, é possível se estabelecer propriedades sobre seus comportamentos, que devem ser sempre verdadeiros, podendo, estes comportamentos, serem testados de forma automática. Na linguagem também são utilizadas técnicas tradicionais para se testar códigos que devem interagir com arquivos, redes ou hardware exóticos. No entanto, este código impuro é muito menor do que seria encontrado em um programa codificado em uma linguagem tradicional. Isso aumenta a nossa confiança de que o código seja sólido.
- **Prazer.** Acredita-se ser fácil entender a programação em Haskell e isto torna possível construir programas pequenos e em um curto espaço de tempo. Para se programar em Haskell, são importadas muitas idéias da Matemática abstrata para o campo prático, o que tornam esta programação prazerosa.

Resumidamente, Haskell é uma linguagem que representa uma novidade, onde o leitor é incentivado a pensar em programação a partir de um ponto de vista diferente dos já experimentados, representando uma nova perspectiva. Haskell é uma linguagem poderosa, porque seus programas são pequenos, rápidos e seguros. Finalmente, Haskell é uma linguagem que proporciona prazer em se programar com ela, porque em seus programas são aplicadas técnicas de programação adequadas para resolver problemas do mundo real.

1.6 Haskell na Indústria

Baseado em [34], serão mostradas a seguir alguns exemplos de grandes sistemas codificados em Haskell:

- Os projetos ASIC e FPGA (Lava, produtos da Bluespec, Inc.).
- Software para composição musical (Hscore).
- Compiladores e ferramentas relacionadas a eles (GHC).
- Controle de revisão distribuída (Darcs).
- Middlewere para Web (HAppS, produtos da Galois, Inc.).

Em termos de companhias que já utilizam Haskell, indicamos o leitor visitar a página wiki (http://www.haskell.org/haskellwiki/Haskell_in_industry). Lá, entre outras, estão citadas:

- ABN AMRO. Um banco internacional que usa Haskell na avaliação de riscos e protótipos de seus derivativos financeiros.
- Anygma. Uma companhia que desenvolve ferramentas para a criação de multimídia.

- Amgen. Uma companhia de biotecnologia que cria modelos matemáticos e outras aplicações complexas.
- Bluespec. Um projeto de software ASIC e FPGA.
- Eaton. Usa Haskell no projeto e verificação de sistemas de veículos hidráulicos híbridos.

A nosso ver, o que existe mesmo é a falta de informação e conhecimento do que realmente é programação funcional e quais as suas vantagens e desvantagens em relação à programação em outros paradigmas. Este livro tenta prover subsídios para que seus usuários possam iniciar um processo de discussão sobre o tema. Para desencadear este processo, vamos começar fazendo uma comparação entre a programação funcional e a programação estruturada.

1.7 A comunidade de Haskell

Existem muitas formas pelas quais leitores e usuários podem entrar em contacto com outros programadores usuários de Haskell, a fim de tirar dúvidas, saber o que outras pessoas estão falando sobre o tema ou até mesmo apenas para manter contacto social. Podem ser citadas:

- A principal fonte de recursos é o site oficial de Haskell: <http://www.haskell.org/>. Apresenta muitos links para várias comunidades e atividades relacionadas ao tema.
- Listas de discussão em Haskell: http://haskell.org/haskellwiki/Mailing_lists. O mais interessante é o Haskell-cafe, onde profissionais e admiradores de vários níveis se encontram.
- Para um *chat* de tempo real, o canal IRC Haskell: <http://haskell.org/haskellwiki/IRC/channel>. nomeado por `#Haskell`.
- Muitos grupos de usuários, *workgroups* acadêmicos: <http://haskell.org/haskellwiki/User.-groups>.
- O jornal semanal de Haskell: <http://sequence.complete.org/>. Mostra um sumário semanal das atividades da comunidade de Haskell, com vários links para listas de discussões.
- O Haskell Communities and Activities Report: <http://haskell.org/communities/>. Coleciona informações sobre pessoas que usam Haskell e o que elas estão fazendo.

1.8 Resumo

Este Capítulo introdutório foi feito na tentativa de apresentar a Programação Funcional como uma alternativa importante na escolha de uma linguagem de programação, enumerando suas principais características, mostrando suas principais vantagens sobre as outras. As vantagens da programação sem atribuições em relação à programação com atribuições são similares às da programação sem gotos em relação à programação com gotos. Resumidamente, são elas:

- os programas são mais fáceis de serem entendidos,
- os programas podem ser derivados mais sistematicamente e
- é mais fácil de serem feitas inferências sobre eles.

As vantagens da programação funcional sobre a programação imperativa podem ser resumidas nos seguintes argumentos:

1. A programação funcional conduz a uma disciplina que melhora o estilo.
2. A programação funcional encoraja o programador a pensar em níveis mais altos de abstração, através de mecanismos como funções de **alta ordem**, **lazy evaluation** e **polimorfismo**.
3. A programação funcional representa um paradigma de programação para a computação massivamente paralela, pela ausência de atribuições, pela independência da ordem de avaliação e pela habilidade de operar estruturas complexas de dados.
4. A programação funcional é uma aplicação da Inteligência Artificial.
5. A programação funcional é importante na criação de especificações executáveis e na implementação de protótipos com uma rigorosa fundamentação matemática. Isto permite verificar se as especificações estão corretas, ou não.
6. A programação funcional está fortemente acoplada à Teoria da Computação.

Esta fundamentação matemática tem como vantagem, primeiramente a constatação de que essas linguagens têm um embasamento teórico e depois, sendo matemático, torna estas linguagens mais tratáveis e mais fáceis de terem seus programas provados.

1.9 Composição deste livro

Este livro é composto desta Introdução, 8 (oito) Capítulos, as referências bibliográficas consultadas e dois Apêndices. Nesta Introdução, é analisada a importância das linguagens funcionais e a necessidade de estudar o λ -cálculo, justificando sua escolha como linguagem intermediária entre as linguagens funcionais e o código executável. Além disso, mostra-se porque as linguagens funcionais aumentam a modularidade dos sistemas através das funções de alto nível e do mecanismo de avaliação preguiçosa.

O Capítulo 1 é dedicado à fundamentação das linguagens funcionais, abordando as principais diferenças entre elas e as linguagens de outros paradigmas. O mundo das linguagens de programação é dividido entre o mundo das expressões e o mundo das atribuições, evidenciando as vantagens do primeiro mundo em relação ao segundo.

No Capítulo 2, é introduzido o λ -cálculo, sua evolução histórica e como ele é usado nos dias atuais. A teoria é colocada de maneira simples e introdutória, dado o objetivo do livro.

No Capítulo 3, inicia-se a programação em Haskell. São mostrados seus construtores e uma série de exemplos, analisando como as funções podem ser construídas em Haskell. São mostrados os tipos de dados primitivos adotados em Haskell e os tipos estruturados mais simples que são as tuplas. No Capítulo, também são mostrados os esquemas de provas de programas, juntamente com vários exercícios, resolvidos ou propostos.

O Capítulo 4 é dedicado às listas em Haskell. Este Capítulo se torna necessário, dada a importância que este tipo de dado tem nas linguagens funcionais. Neste Capítulo, são mostradas as compreensões ou expressões ZF e a composição de funções como uma característica apenas das linguagens funcionais, usadas na construção de funções. Um tema importante e que é discutido neste Capítulo se refere às formas de provas da corretude de programas em Haskell, usando indução estrutural sobre listas. No Capítulo são mostrados vários exemplos resolvidos e, ao final, são colocados vários exercícios à apreciação do leitor.

O Capítulo 5 é dedicado aos tipos de dados algébricos. Inicialmente, são mostradas as **type class** como formas de incluir um determinado tipo de dados construído pelo usuário em uma classe de tipos que tenham funções em comum, dando origem à sobrecarga como forma

de polimorfismo. Neste capítulo são mostradas algumas estruturas de dados importantes que podem ser construídas usando os tipos algébricos, a exemplo das especificações de expressões em BNF e das árvores. O Capítulo apresenta um estudo sobre o tratamento de exceções e como ele é feito em Haskell. Além disso, o capítulo apresenta um estudo sobre o mecanismo de avaliação lazy que Haskell utiliza, possibilitando a construção de listas potencialmente infinitas. O Capítulo termina com a apresentação de formas de provas de programas em Haskell que utilizam tipos algébricos.

O Capítulo 6 é dedicado ao estudo da modularização de programas codificados em Haskell. O Capítulo é iniciado com o estudo dos módulos e como eles são construídos em Haskell, juntamente com os mecanismos de importação e exportação destas entidades. No Capítulo, são mostrados como estes módulos são compilados separadamente e como eles são linkados para formarem um código executável. As diversas ferramentas de execução de código executável que o sistema dispõe para a depuração de código.

No Capítulo 7 é feito um estudo sobre os tipos de dados abstratos. Inicialmente é feito um estudo simplificado sobre a eficiência em programas funcionais e em seguida são mostrados os tipos de dados abstratos e as formas como eles são construídos em Haskell. Todos os tipos abstratos mostrados são analisados quanto à eficiência, tentando formas alternativas de construí-los levando em consideração este fator. O Capítulo termina, como todos os outros, com um Resumo de seu conteúdo.

O Capítulo 8 é dedicado às operações de entrada e saída em Haskell, evidenciando o uso de arquivos ou dispositivos como saída ou entrada de dados. Este processo em Haskell é feito através do mecanismo de “ações”, cuja semântica representa o conteúdo principal do Capítulo.

O Capítulo 9 contempla as referências bibliográficas consultadas durante sua elaboração juntamente com algumas outras indicações que devem ser analisadas por quem deseja conhecer melhor o paradigma funcional.

O livro apresenta finalmente dois apêndices com a intenção de torná-lo autocontido. O Apêndice A é dedicado às principais funções já constantes do arquivo Prelude.hs e que podem ajudar o usuário na construção de novas funções.

O livro termina com o Apêndice B, dedicado à compilação de programas codificados em Haskell, utilizando o compilador GHC (Glasgow Haskell Compiler) que gera código nativo. Ele se tornou necessário dadas as diferenças entre a compilação de programas imperativos e a compilação de programas funcionais. São mostradas as várias formas de utilização de parâmetros buscando construir programas executáveis adequados aos objetivos do usuário. Também são mostradas as diversas formas de execução adotadas, como a geração de *traces* ou dados estatísticos sobre o programa, que podem ser analisados pelo usuário.

Capítulo 2

Programação funcional

*“We can now see that a lazy implementation
based on suspensions, we can treat every function in the same way.
Indeed, all functions are treated as potentially non-strict
and their arguments is automatically suspended.
Later, is and when it is needed, it will be ussuspended (strictly evaluated).”*
(**Antony D. T. Davie** in [11])

2.1 Introdução

A programação funcional teve início antes da invenção dos computadores eletrônicos. No início do século XX, muitos matemáticos estavam preocupados com a fundamentação matemática; em particular, queriam saber mais sobre os conjuntos infinitos. Muito desta preocupação aconteceu por causa do surgimento, no final do século XIX, de uma teoria que afirmava a existência de várias ordens de infinitos, desenvolvida por George Cantor (1845-1918) [30]. Muitos matemáticos, como Leopold Kronecker (1823-1891), questionaram a existência destes objetos e condenaram a teoria de Cantor como pura “enrolação”. Estes matemáticos defendiam que um objeto matemático só poderia existir se, pelo menos, em princípio, pudesse ser construído. Por este motivo, eles ficaram conhecidos como “*construtivistas*”.

Mas o que significa dizer que um número, ou outro objeto matemático, seja construtível? Esta idéia foi desenvolvida lentamente, ao longo de muitos anos. Guiseppe Peano (1858-1932), um matemático, lógico e lingüista, escreveu “*Formulaire de Mathématique*” (1894-1908), onde mostrou como os números naturais poderiam ser construídos através de finitas aplicações da função sucessor. Começando em 1923, Thoralf Skolen (1887-1963) mostrou que quase toda a teoria dos números naturais poderia ser desenvolvida construtivamente pelo uso intensivo de definições recursivas, como as de Peano. Para evitar apelos questionáveis sobre o infinito, pareceu razoável chamar um objeto de construtível se ele pudesse ser construído em um número finito de passos, cada um deles requerendo apenas uma quantidade finita de esforço. Assim, nas primeiras décadas do século XX, já existia considerável experiência sobre as definições recursivas de funções sobre os números naturais.

A cardinalidade (quantidade de elementos) dos conjuntos finitos era fácil ser conhecida, uma vez que era necessário apenas contar seus elementos. Mas, para os conjuntos infinitos, esta técnica não podia ser aplicada. Inicialmente, era necessário definir o que era realmente um conjunto infinito. Foi definido que um conjunto era infinito se fosse possível construir uma correspondência biunívoca entre ele e um subconjunto próprio de si mesmo. Foram definidos os conjuntos infinitos enumeráveis, caracterizados pelos conjuntos infinitos para os quais fosse

possível construir uma correspondência biunívoca com o conjunto dos números naturais, \mathbf{N} . Assim, todos os conjuntos infinitos enumeráveis tinham a mesma cardinalidade, que foi definida por \aleph_0 ¹. Assim, as cardinalidades do conjunto dos números inteiros, \mathbf{Z} , e dos números racionais, \mathbf{Q} , também são \aleph_0 , uma vez que é possível construir uma correspondência biunívoca entre \mathbf{Z} e \mathbf{N} e entre \mathbf{Q} e \mathbf{N} . Os conjuntos infinitos para os quais não fosse possível estabelecer uma correspondência biunívoca entre eles e \mathbf{N} foram chamados de infinitos não enumeráveis.

Dado um conjunto \mathbf{A} , o conjunto das partes de \mathbf{A} , denotado por $\mathcal{P}(\mathbf{A})$, é o conjunto cujos elementos são todos os subconjuntos de \mathbf{A} , incluindo ele próprio e o conjunto vazio. Um resultado que garante a existência de cardinais infinitos é o fato de que o conjunto das partes de um conjunto tem sempre cardinalidade maior que a deste. Este fato ficou conhecido como Teorema de Cantor, aqui descrito sem demonstração.

Teorema de Cantor. *Seja \mathbf{A} um conjunto e $\mathcal{P}(\mathbf{A})$ o conjunto das partes de \mathbf{A} . Então a cardinalidade de \mathbf{A} , denotada por $\#\mathbf{A}$, é menor que a cardinalidade de $\mathcal{P}(\mathbf{A})$, ou seja, $\#\mathbf{A} < \#\mathcal{P}(\mathbf{A})$.*

Prova-se que o conjunto das partes de \mathbf{N} é equipotente (tem a mesma cardinalidade) ao conjunto dos reais, \mathbf{R} . Considerando que 2^k representa o cardinal do conjunto das partes de conjuntos com cardinalidades k , então 2^{\aleph_0} é a cardinalidade do conjunto dos números reais, \mathbf{R} .

Mas, o que isto tem a ver com a Computação? Na realidade, prova-se que existem \aleph_0 programas que podem ser definidos em uma linguagem de programação como C, Java, Haskell, ou outra. Além disso, existem 2^{\aleph_0} funções de \mathbf{N} para \mathbf{N} . Logo, conclui-se que existem infinitas funções que não podem ser representadas algorítmicamente, ou seja, que não são computáveis. Isto significa que a quantidade de funções computáveis é muito menor que a quantidade de funções não computáveis. Este é um resultado, no mínimo, inusitado.

2.2 Computabilidade de funções

Na década de 1930, existiram muitas tentativas de formalização do construtivismo, procurando caracterizar o que era computabilidade efetiva, ou seja, procurava-se saber o que realmente podia ser computado. Uma das mais famosas tentativas foi a definição de Turing sobre uma classe de máquinas abstratas, que ficaram conhecidas como “máquinas de Turing”, que realizavam operações de leitura e escritas sobre uma fita de tamanho finito. Outra técnica, baseada mais diretamente nos trabalhos de Skolen e Peano, consistia no uso de “funções recursivas gerais”, devida a Gödel. Uma outra técnica, com implicação importante na programação funcional, foi a criação do λ -cálculo, desenvolvido por Church e Kleene, no início da década de 1930. Outra noção de computabilidade, conhecida como “Algoritmos de Markov”, também foi desenvolvida nesta mesma época. O mais importante foi que todas estas noções de computabilidade foram provadas serem equivalentes. Esta equivalência levou Church, em 1936, a propor o que ficou conhecida como a Tese de Church², onde ele afirmava que “uma função era computável se ela fosse primitiva recursiva” [9].

Isto significa que, já na década anterior à década da invenção do computador eletrônico, muitos matemáticos e lógicos já haviam investigado, com profundidade, a computabilidade de funções e identificado a classe das funções computáveis como a classe das funções primitivas recursivas.

Em 1958, John McCarthy investigava o uso de operações sobre listas ligadas para implementar um programa de diferenciação simbólica. Como a diferenciação é um processo recursivo,

¹ \aleph é a primeira letra do alfabeto hebraico, conhecida por Aleph.

²Na realidade, não se trata de uma tese, uma vez que ela nunca foi provada. No entanto, nunca foi exibido um contra-exemplo, mostrando que esta conjectura esteja errada.

McCarthy sentiu-se atraído a usar funções recursivas e, além disso, ele também achou conveniente passar funções como argumentos para outras funções. McCarthy verificou que o λ -cálculo provia uma notação conveniente para estes propósitos e, por isto, resolveu usar a notação de Church em sua pesquisa. Ainda em 1958, no MIT, foi iniciado um projeto com o objetivo de construir uma linguagem de programação que incorporasse estas idéias. O resultado ficou conhecido como LISP 1, que foi descrita por McCarthy, em 1960, em seu artigo “*Recursive Functions of Symbolic Expressions and Their Computation by Machine*” [30]. Neste artigo, ele mostrou como vários programas complexos podiam ser expressos por funções puras operando sobre estruturas de listas. Este fato é caracterizado, por alguns pesquisadores, como o marco inicial da programação funcional.

No final da década de 1960 e início da década de 1970, um grande número de cientistas da Computação começaram a investigar a programação com funções puras, chamada de “programação aplicativa”, uma vez que a operação central consistia na aplicação de uma função a seu argumento. Em particular, Peter Landin (1964, 1965 e 1966) desenvolveu muitas das idéias centrais para o uso, notação e implementação das linguagens de programação aplicativas, sendo importante destacar sua tentativa de traduzir a definição de Algol 60, uma linguagem não funcional, para o λ -cálculo. Baseados neste estudo, Strachey e Scott construíram um método de definição da semântica de linguagens de programação, conhecido como “semântica denotacional”. Em essência, a semântica denotacional define o significado de um programa, em termos de um programa funcional equivalente.

No entanto, a programação aplicativa, que havia sido investigada por um reduzido número de pesquisadores nos anos de 1960 e 1970, passou a receber uma atenção bem maior após 1978, quando John Backus, o principal criador do FORTRAN, publicou um paper onde fez severas críticas às linguagens de programação convencionais, sugerindo a criação de um novo paradigma de programação. Ele propôs o paradigma chamado de “programação funcional” que, em essência, é a programação aplicativa com ênfase no uso de funcionais, que são funções que operam sobre outras funções. Muitos dos funcionais de Backus foram inspirados em **APL**, uma linguagem imperativa projetada na década de 1960, que provia operadores poderosos, sem atribuição, sobre estruturas de dados. A partir desta publicação, o número de pesquisadores na área de linguagens de programação funcional tem aumentado significativamente.

2.3 Análise de dependências

Segundo MacLennan [30], as linguagens de programação se enquadram em um de dois mundos: o mundo das expressões (aritméticas, relacionais, booleanas, etc) e o mundo das atribuições. No primeiro caso, o único objetivo é encontrar o valor de uma expressão através de um processo de “avaliação”. Já no mundo das atribuições, o processo predominante é a “alteração” de alguma coisa, sendo dividido em dois tipos. No primeiro tipo, se altera o fluxo da execução de um programa, usando comandos de seleção, como **if**, **for**, **while**, **repeat**, **goto** e chamadas a procedimentos. No segundo tipo, o que se altera é o estado da memória (principal ou secundária) do computador.

No mundo das atribuições, a ordem em que as coisas são feitas tem importância fundamental. Por exemplo, a seqüência

$$\begin{aligned} i &= i + 1; \\ a &= a * i; \end{aligned}$$

tem um efeito diferente se forem consideradas as mesmas instruções em ordem invertida, ou seja:

$$\begin{aligned} a &= a * i; \\ i &= i + 1; \end{aligned}$$

Analisemos a expressão $z = (2 * a * y + b) * (2 * a * y + c)$. As duas sub-expressões entre parênteses do lado direito do sinal de atribuição contêm uma sub-expressão em comum ($2 * a * y$) e qualquer compilador, com um mínimo de otimização, transformaria este fragmento de código, da seguinte forma:

```
t = 2 * a * y;
z = (t + b) * (t + c);
```

No mundo das expressões, é seguro promover esta otimização porque qualquer sub-expressão, como $2 * a * y$, tem sempre o mesmo valor, em um mesmo contexto. Analisemos agora, uma situação similar no mundo das atribuições.

Sejam as expressões

```
y = 2 * a * y + b; e z = 2 * a * y + c;
```

onde também verificamos uma sub-expressão em comum ($2 * a * y$). Se for realizada a mesma fatoração anterior, teremos:

```
t = 2 * a * y;    y = t + b; z = t + c;
```

Esta otimização altera o valor da variável z , porque os valores de y são diferentes nas duas ocorrências da sub-expressão $2 * a * y$. Portanto, não é possível realizar esta otimização. Apesar da análise de dependência entre sub-expressões poder ser feita por um compilador, ela requer técnicas sofisticadas de análise de fluxo. Tais análises, normalmente são caras para serem realizadas e difíceis de serem implementadas corretamente. No mundo das expressões, não existem atualizações destrutivas de variáveis e, portanto, a variável t não poderia ter mais de um valor. De forma resumida, é fácil e seguro realizar estas otimizações nas expressões e difícil e inseguro de serem feitas no mundo das atribuições. Fica clara a vantagem das expressões sobre as atribuições. O propósito da programação funcional é estender as vantagens das expressões para as linguagens de programação.

2.4 Funções e expressões aplicativas

Vamos continuar analisando o mundo das expressões. Elas são estruturalmente simples porque são compostas de aplicações de operações aritméticas a seus argumentos. Estas operações são **funções puras**, ou seja, são mapeamentos matemáticos de entradas para saídas. Isto significa que o resultado de uma operação depende apenas de suas entradas. Além disso, uma expressão construída a partir de funções puras e constantes tem sempre o mesmo valor. Por exemplo, seja a função pura f definida da seguinte forma:

$$f(u) = (u + b)(u + c)$$

em um contexto em que $b = 3$ e $c = -2$. A função f é pura porque é definida em termos de funções puras, que são as operações aritméticas de adição e multiplicação. Vamos considerar a avaliação de $f(3ax)$ em um contexto em que $a = 2$ e $x = 3$. Como a avaliação de expressões aritméticas independe da ordem de avaliação, assim também será a avaliação de f . Isto significa que o argumento ($3ax$) de f pode ser avaliado antes ou depois da substituição; o resultado será sempre o mesmo. Se ele for avaliado antes, a sequência de avaliações será:

$$f(3ax) = f(3 * 2 * 3) = f(18) = (18 + b) * (18 + c) = \dots = 21 * 16 = 336$$

Por outro lado, se a avaliação de $3ax$ for deixada para ser feita após a substituição, a sequência de avaliações será:

$$f(3ax) = (3ax + b) * (3ax + c) = (3 * 2 * x + b) * (3ax + c) = (6 * x + b) * (3ax + c) = \dots = 336$$

No campo das linguagens de programação, estas duas seqüências de avaliações correspondem aos métodos de passagem de parâmetros por valor e por nome, respectivamente. No primeiro

caso, a avaliação é dita “estrita” e, no segundo, ela é dita “não estrita”, em relação a seu argumento x . A passagem de parâmetros por valor, normalmente, é mais eficiente porque o argumento é avaliado apenas uma vez. No entanto, o que é mais importante é o fato de que o valor da expressão é o mesmo, independente da ordem de avaliação adotada. Apesar da ordem de avaliação escolhida não ter influência sobre o valor final da expressão, ela pode influenciar sobre o término, ou não, do processo de avaliação. Por exemplo, a avaliação da função $f(x) = 1$ para $x = 1/a$, em um contexto em que a seja igual a zero, tem diferença nas duas ordens de avaliação. Se $1/a$ for avaliado antes da substituição, a avaliação será indefinida e não termina. Se for deixada para ser feita depois da substituição, o resultado será 1.

A programação aplicativa ou funcional é freqüentemente distingüida da programação imperativa que adota um estilo que faz uso de imperativos ou ordens, por exemplo, “troque isto!”, “vá para tal lugar!”, “substitua isto!”, e assim por diante. Ao contrário, o mundo das expressões envolve a descrição de valores. Por este motivo, o termo “programação orientada por valores”.

Na programação aplicativa, os programas tomam a forma de expressões aplicativas. Uma expressão deste tipo é uma constante ($2, \pi, e$, etc) ou é composta totalmente de aplicações de funções puras a seus argumentos, que também são expressões aplicativas. Em BNF, as expressões aplicativas são definidas da seguinte forma:

```
<EA> ::= <id> (<EA>, ...)  
        | <literal>  
        | <id>
```

A estrutura aplicativa se torna mais clara se as expressões forem escritas na forma pré-fixa, ou seja, $sum(prod(prod(2, a), x), b)$ em vez da forma usual, infixa, $2ax + b$.

A programação aplicativa tem um único construtor sintático que é a aplicação de uma função a seu argumento. Na realidade, este construtor é tão importante que, normalmente, é representado de forma implícita, por justaposição, em vez de explicitamente, através de algum símbolo. Desta forma, $senx$ significa a aplicação da função sen ao argumento x .

2.5 Independência da ordem de avaliação

Pelo que foi visto na seção anterior, para entender as vantagens do mundo das expressões e as fontes de suas propriedades, é necessário investigar a ordem de avaliação nas expressões aritméticas. Avaliar alguma coisa significa encontrar seu valor. Assim, podemos avaliar a expressão aritmética “ $5 * 4 + 3$ ” encontrando seu valor que, no caso, é 23.

Podemos também avaliar a expressão $(3ax + b)(3ax + c)$? A resposta é não; a menos que sejam conhecidos os valores de a, b, c e x . Para entender como isto é feito, vamos mostrar como uma expressão é avaliada, através da construção de uma árvore de avaliação, conforme pode ser visto na Figura 2.1, onde as variáveis ficam nas folhas e as operações nos nós internos.

O valor desta expressão depende do contexto em que ela é avaliada. Por exemplo, vamos avaliá-la em um contexto em que $a = 2, b = 3, c = -2$ e $x = 3$. A avaliação pode ser iniciada em vários pontos mas, por motivo de regularidade, ela será feita da esquerda para a direita. Nesta estrutura, cada operação em um nó interno depende apenas dos resultados das operações dos nós abaixo dele, na árvore. A avaliação de uma sub-árvore afeta apenas a árvore acima dela, não influenciando os resultados das sub-árvores que estejam a sua esquerda ou a sua direita. Colocando todos os operadores de forma explícita, a expressão se transforma em:

$$(3 * a * x + b) * (3 * a * x + c)$$

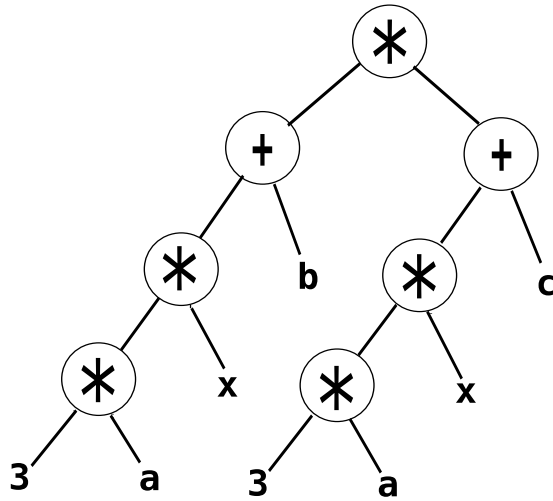


Figura 2.1: Árvore representativa da expressão $(3ax + b)(3ax + c)$.

Para realizar a primeira operação (a multiplicação $3 * a$) é necessário saber o valor de a que, neste contexto, é 2. Substituindo este valor na expressão, ela se torna

$$(3 * 2 * x + b) * (3 * 2 * x + c)$$

Observe que o a da segunda sub-expressão também foi substituído por causa da árvore utilizada. O processo de avaliação continua, substituindo-se a expressão inicial por uma nova expressão. Neste caso, por

$$(6 * x + b) * (6 * x + c)$$

A sequência completa de todos os passos realizados no processo de avaliação é mostrada a seguir, onde a flexa dupla (\Rightarrow) significa a transformação de expressões.

$$\begin{aligned} &(3 * a * x + b) * (3 * a * x + c) \\ &(3 * 2 * x + b) * (3 * 2 * x + c) \\ &(6 * x + b) * (6 * x + c) \\ &(6 * 3 + b) * (6 * 3 + c) \\ &(18 + b) * (18 + c) \\ &(18 + 3) * (18 + c) \\ &21 * (18 + c) \\ &21 * (18 + (-2)) \\ &21 * 16 \\ &336 \end{aligned}$$

Observe que se a avaliação tivesse sido iniciada pela sub-árvore direita da árvore inicial, o resultado seria o mesmo, ou seja, qualquer ordem de avaliação produziria o resultado 336. Isto ocorre porque, na avaliação de uma expressão pura³, a avaliação de uma subexpressão não afeta o valor de qualquer outra subexpressão porque não existe qualquer dependência entre elas.

É fácil entender esta independência da ordem de avaliação, observando a árvore de avaliação. A avaliação é iniciada com a colocação de alguns valores nas folhas. Os nós internos são avaliados em qualquer ordem, sob demanda, podendo ser avaliados em paralelo. Cada operação depende apenas de suas entradas que são os valores dos nós filhos. Este processo de avaliação pode ser visto na Figura 2.2

Este processo é conhecido como “decoração”, onde cada nó interno é decorado com o valor da aplicação da operação aos valores dos nós abaixo dele. A avaliação termina quando a raiz

³Uma expressão é dita pura quando não realiza qualquer operação de atribuição, explícita ou implícita.

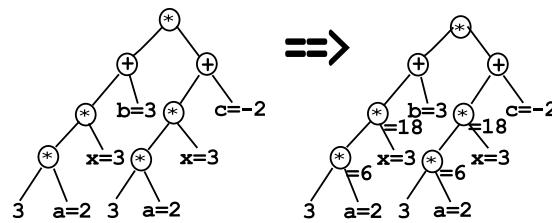


Figura 2.2: Representação do processo de avaliação.

da árvore for decorada com o valor final da expressão. Como afirmado anteriormente, diversos processos podem acontecer em paralelo, na decoração da árvore, desde que seja observada a estrutura de árvore. Isto significa que sempre se chega ao mesmo valor.

Esta propriedade verificada nas expressões puras, ou seja, a independência da ordem de avaliação, é chamada de “*propriedade de Church-Rosser*”. Ela permite a construção de compiladores capazes de escolher a ordem de avaliação que faça o melhor uso dos recursos da máquina. A possibilidade da avaliação ser realizada em paralelo, implica na utilização de multiprocessadores de forma bastante natural.

Por outro lado, as “expressões impuras”, normalmente, não apresentam esta propriedade, conforme pode ser verificado no exemplo a seguir, em C. Seja a expressão $a + 2 * fun(b)$. Ela é pura ou impura? Para responder a isso, devemos verificar a definição de fun. Por exemplo,

```
int fun(int x)
{
    return(x*x);
}
```

Como *fun* não executa qualquer atribuição, a não ser a pseudo-atribuição a fun do valor de retorno da função, ela é uma função pura. Isto significa que, na avaliação da expressão $a + 2 * fun(b)$, pode-se avaliar primeiro a sub-expressão *a* ou $2 * fun(b)$, porque o resultado será o mesmo. No entanto, vamos supor a função *fun1*, definida da seguinte forma:

```
int fun1(int x)
{
    a = a + 1;
    return (x * x);
}
```

Neste caso, *fun1* é uma pseudo-função, porque ela não é uma função pura. Supondo que a variável *a*, mencionada em *fun1*, seja a mesma variável da expressão $a + 2 * fun1(b)$, como *fun1* altera o valor de *a*, o valor de $a + 2 * fun1(b)$ depende de qual operando do operador “+” é avaliado em primeiro lugar. Se, por exemplo, o valor de *a* for zero, caso ele seja avaliado primeiro na expressão $a + 2 * fun1(b)$, o valor final da expressão será $2b^2$, ao passo que se $2 * fun1(b)$ for avaliado primeiro, a expressão final terá o valor $2b^2 + 1$.

Desta forma, para uma mesma linguagem, C, foram mostrados dois exemplos em que, no primeiro, observa-se a independência na ordem de avaliação e, no segundo, esta propriedade não é verificada.

2.6 Transparência referencial

Vamos novamente considerar o contexto de avaliação da seção anterior. Se uma pessoa fosse avaliar manualmente a expressão $(3ax + b)(3ax + c)$ jamais iria avaliar a sub-expressão $3ax$, que neste contexto é 18, duas vezes. Uma vez avaliada esta sub-expressão, o avaliador humano substituiria a sub-expressão $3ax$ por 18, em todos os casos onde ela aparecesse. A avaliação seria feita da seguinte forma:

$t = 3 * a * x$
 $\Rightarrow 3 * 2 * x$
 $\Rightarrow 6 * x$
 $\Rightarrow 6 * 3$
 $\Rightarrow 18$
 $\Rightarrow 18 + b) * (18 + c)$
 $\Rightarrow (18 + 3) * (18 + c)$
 $\Rightarrow 21 * (18 + (-2))$
 $\Rightarrow 21 * 16$
 $\Rightarrow 336$

Isto acontece porque a avaliação de uma mesma expressão, em um contexto fixo sempre dará como resultado o mesmo valor. Para os valores de $a = 2$ e $x = 3$, $3ax$ será sempre igual a 18. Esta técnica de avaliação humana é também utilizada na avaliação de expressões puras. Como a subexpressão $3ax$ ocorre duas vezes, não existe razão para se duplicar a representação na árvore. O valor desta sub-expressão é compartilhado pelas operações dos nós que se encontram acima do nó desta operação.

A rigor, não se tem mais uma estrutura de árvore, e sim um grafo acíclico. No entanto, pode-se decorar o grafo partindo das folhas, da mesma maneira feita antes. Seu desenvolvimento pode ser entendido observando o *grafo de redução* da expressão mostrado na Figura 2.3.

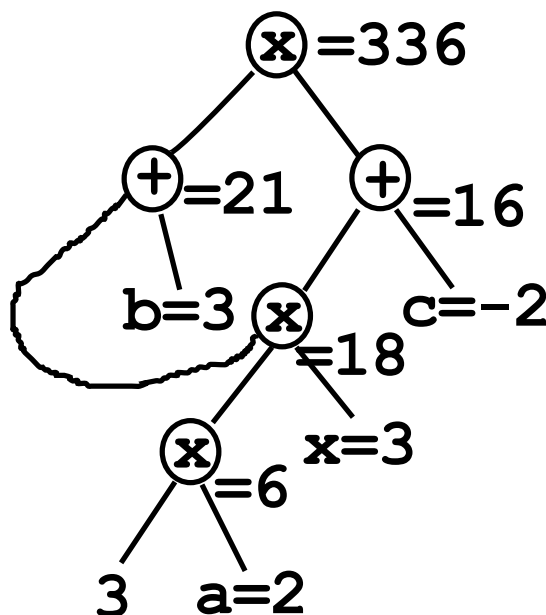


Figura 2.3: Grafo com um nó compartilhado.

Esta propriedade é chamada de “*transparência referencial*”, e significa que, em um contexto fixo, a substituição de sub-expressões por seus valores é completamente independente da expressão envolvente. Portanto, uma vez que uma expressão tenha sido avaliada em um dado

contexto, não é mais necessário avaliá-la, porque seu valor jamais será alterado. De forma mais geral, a transparência referencial pode ser definida como “a habilidade universal de substituir iguais por iguais”. Em um contexto em que $a = 2$ e $x = 3$, sempre pode-se substituir $3ax$ por 18 ou 18 por $3ax$, sem que o valor da expressão envolvente seja alterado. A transparência referencial resulta do fato de que os operadores aritméticos não têm memória e, assim sendo, toda chamada a um operador com as mesmas entradas produz sempre o mesmo resultado.

Mas por que a transparência referencial é importante? Da Matemática, sabemos da importância de poder substituir iguais por iguais. Isto conduz à derivação de novas equações, a partir de equações dadas e a transformação de expressões em formas mais usuais e adequadas para a prova de propriedades sobre elas.

No contexto das linguagens de programação, a transparência referencial permite otimizações como a eliminação de sub-expressões comuns. Por exemplo, dada a definição da pseudo função *fun1*, da seção anterior, é claro que, como *fun1* deixa em *a* o registro do número de vezes que ela é chamada, não se poderia eliminar a subexpressão comum, *fun1(b)*, da expressão

$$(a + 2 * fun1(b)) * (c + 2 * fun1(b))$$

Isto acontece porque a troca do número de vezes que *fun1* é chamada altera o resultado da expressão. Em algumas linguagens de programação, isto complica a eliminação de subexpressões comuns.

2.7 Interfaces manifestas

A evolução da notação matemática ao longo dos anos tem permitido que ela seja utilizada, com sucesso, para exibir muitas propriedades. Uma destas propriedades se refere às interfaces manifestas, ou seja, as conexões de entradas e saídas entre uma sub-expressão e a expressão que a envolve são visualmente óbvias. Consideremos a expressão $4 + 5$. O resultado desta adição depende apenas das entradas para a operação (4 e 5) e elas estão mostradas de forma clara na expressão, ou seja, uma está colocada à esquerda e a outra à direita do operador $+$. Não existem entradas escondidas para este operador.

Vamos imaginar agora a pseudo-função *fun2*, definida da seguinte forma:

```
int fun2(int x)
{
    a = a + 1;
    return (a * x);
}
```

Não existe uma forma de se conhecer o valor de *fun2(5)* sem antes saber o valor da variável não local, *a*. O valor de *a* é atualizado em cada chamada a *fun2*, então *fun2(5)* tem valores diferentes em cada chamada. Esta situação caracteriza a existência de interfaces escondidas para a função *fun2*, tornando difícil, ou mesmo impossível, se prever o comportamento da função.

Consideremos, novamente, a expressão $(2ax + b) * (2ax + c)$. As entradas para o primeiro operador $+$ ($2ax$ e b) estão manifestas. O papel da subexpressão $2ax + b$, dentro da expressão completa, também é manifesto, ou seja, ela representa o argumento esquerdo do operador de multiplicação. Não existem saídas escondidas ou *side effects* na adição. As entradas são as subexpressões $2ax$ e b , em qualquer lado do operador e a saída é facilmente calculada e liberada para a expressão envolvente.

As características das interfaces manifestas podem ser resumidas da seguinte forma: as expressões podem ser representadas por árvores e a mesma árvore representa tanto a estrutura

sintática de uma expressão quanto a forma como os dados fluem na expressão. As subexpressões que se comunicam entre si podem sempre ser colocadas em posições contíguas na árvore ou na forma escrita da expressão. Esta característica não é válida no mundo das atribuições porque as variáveis alteráveis permitem comunicação não local. Em geral, o gráfico do fluxo de dados não é uma árvore e pode ser uma estrutura muito diferente da árvore sintática. Assim, pode ser impossível colocar juntas as partes comunicantes, para que suas interfaces sejam óbvias.

2.8 Definição de funções

A programação funcional usando a linguagem Haskell é o principal objetivo desta Apostila. Isto significa que devemos estudar formas de passagens das funções matemáticas para funções codificadas em Haskell. Do ponto de vista matemático, as funções são definidas sobre conjuntos, domínio e imagem, sem qualquer preocupação como elas são executadas para encontrar um resultado. Já no mundo da computação, as funções são declaradas sobre tipos e levam-se em conta os algoritmos utilizados para implementá-las e a diferença de desempenho entre eles tem importância fundamental. Apesar desta diferença de pontos de vista, o processo de passagem de um mundo para o outro é quase um processo de tradução direta. Nas seções seguintes, as funções serão definidas levando-se em conta o ponto de vista da Matemática e deixamos as definições do ponto de vista computacional para serem feitas a partir do Capítulo 3.

2.8.1 Definições explícitas e implícitas de variáveis

Vamos considerar inicialmente as definições explícitas de variáveis. Uma definição de uma variável é explícita, em uma equação, se ela aparece no lado esquerdo desta equação e não aparece no seu lado direito. Por exemplo, a equação

$$y = 2ax$$

define explicitamente a variável y . As definições explícitas têm a vantagem de poderem ser interpretadas como regras de reescritas que informam como substituir diretamente uma expressão por outra. Por exemplo, a definição anterior de y implica na regra de reescrita $y \Rightarrow 2ax$ informando como eliminar a variável y em uma fórmula, onde ela ocorra. Por exemplo, para eliminar y da expressão $3y^2 + 5y + 1$ aplica-se a regra de reescrita acima, para se obter

$$3y^2 + 5y + 1 \Rightarrow 3(2ax)^2 + 5(2ax) + 1 \Rightarrow 12a^2x^2 + 10ax$$

em que a variável y não ocorre na expressão final. A definição explícita de variáveis pode ser estendida para conjuntos de equações simultâneas. Um conjunto de variáveis é definido explicitamente por um conjunto de equações, se

- as variáveis forem individualmente explícitas e
- as equações puderem ser ordenadas, de forma que nenhuma delas use em seu lado direito uma variável já definida anteriormente na lista.

Por exemplo, o conjunto de equações

$$\begin{cases} y = 2 * a * x \\ x = 2 \\ a = 3 \end{cases}$$

define explicitamente y , x e a . As equações precedentes podem ser convertidas às seguintes

regras de reescrita:

$$\begin{cases} y \Rightarrow 2 * a * x \\ x \Rightarrow 2 \\ a \Rightarrow 3 \end{cases}$$

Estas regras podem ser aplicadas na seguinte ordem: a primeira delas é aplicada até que não exista mais y , em seguida a segunda é aplicada até que não exista mais x e, finalmente, a terceira é aplicada até que não exista mais a . Desta forma teremos a seguinte sequência de reduções:

$$y \Rightarrow 2 * a * x \Rightarrow 2 * a * 2 \Rightarrow 2 * 3 * 2 \Rightarrow 6 * 2 \Rightarrow 12$$

Por outro lado, uma variável é definida implicitamente em uma equação se ela aparecer nos dois lados desta equação. Por exemplo, a equação

$$2a = a + 3$$

define implicitamente a como 3. Para encontrar o valor de a é necessário resolver a equação usando técnicas da álgebra. O processo de solução pode ser visto como uma forma de converter uma definição implícita em uma definição explícita, mais usual, uma vez que uma definição implícita não pode ser convertida diretamente em uma regra de reescrita. A equação anterior, $2a = a + 3$, não informa explicitamente o que deve substituir a na expressão $2ax$, por exemplo. Para encontrar este valor é necessário utilizar as regras da álgebra, que podem não ser triviais. Além disso, as regras de reescrita que resultam de definições explícitas sempre terminam, ou seja, a aplicação repetida das regras de reescritas elimina todas as ocorrências da variável definida.

No entanto, é possível escrever definições implícitas que não terminam, ou seja, nada definem. Considere, como exemplo, a definição implícita

$$a = a + 1$$

Apesar de sabermos que esta equação não tem solução, este fato pode não ser tão óbvio, em casos mais complexos. Se, ingenuamente, interpretarmos esta equação como a regra de reescrita

$$a \Rightarrow a + 1$$

então chegaremos a um não determinismo na seguinte sequência de reduções:

$$2a \Rightarrow 2(a + 1) \Rightarrow 2((a + 1) + 1) \Rightarrow \dots$$

As variáveis também podem ser definidas implicitamente por um conjunto de equações simultâneas. Por exemplo, o conjunto de equações

$$\begin{cases} 2a = a + 3 \\ d - 1 = 3d + a \end{cases}$$

define implicitamente $a = 3$ e $d = -2$. Podemos também ter definições implícitas em que as variáveis não aparecem nos dois lados da mesma equação. Por exemplo, o conjunto de equações

$$\begin{cases} 2a = x \\ x + 1 = a + 4 \end{cases}$$

em que, nem a nem x aparecem nos dois lados de uma mesma equação, define implicitamente a e x . Neste caso, não existe qualquer forma de ordenação destas equações de maneira que as últimas equações não façam uso de variáveis já definidas nas equações anteriores. A forma implícita pode ser observada pela transformação das duas equações em uma só, ou seja,

$$2a + 1 = a + 4$$

Em resumo, uma definição explícita de uma variável informa o valor desta variável, enquanto uma definição implícita estabelece propriedades que apenas esta variável deve apresentar. A determinação do valor de uma variável definida implicitamente exige um processo de solução.

2.8.2 Funções totais e funções parciais

A definição de funções, de acordo com a Matemática, parte das definições de **produto cartesiano** e **relação** entre conjuntos. O *produto cartesiano* entre dois conjuntos **A** e **B**, denotado por $\mathbf{A} \times \mathbf{B}$, é definido como o conjunto de todos os pares ordenados (\mathbf{x}, \mathbf{y}) , onde $\mathbf{x} \in \mathbf{A}$ e $\mathbf{y} \in \mathbf{B}$. Uma *relação* R entre dois conjuntos **A** e **B** é definida como qualquer subconjunto do *produto cartesiano* $\mathbf{A} \times \mathbf{B}$, onde os elementos do conjunto **A** (*conjunto de partida*) que fazem parte da relação formam o **domínio** da relação e os elementos do conjunto **B** (*conjunto de chegada*) que fazem parte da relação constituem a **imagem** da relação.

Já uma **função** entre dois conjuntos **A** e **B** é definida matematicamente como uma relação entre **A** e **B**, com duas características adicionais. A primeira delas é de que todo elemento do conjunto de partida, **A**, faça parte da relação e a segunda é de que nenhum elemento de **A** faça parte de mais de um par na relação. Isto significa que, em uma função, o *conjunto de partida* é o mesmo *conjunto domínio* da função.

Do ponto de vista da Computação, convém registrar que esta definição matemática de função causa algumas dificuldades porque exige que todo elemento do *conjunto de partida* faça parte também do *domínio* da função e isto implica em que nenhum elemento do conjunto de partida fique fora do domínio da função. Imagine uma função definida sobre o conjunto dos inteiros, que é infinito. Neste caso, todo número inteiro teria que fazer parte da função e isto é impossível, porque os computadores são máquinas que podem ter uma memória muito grande, mas será sempre limitada a um maior valor e, portanto, é finita. É impossível se colocar um conjunto infinito dentro de algo finito, sendo esta uma impossibilidade lógica.

Assim sendo, a Computação resolveu chamar a função definida matematicamente como **função total** e resolveu relaxar a exigência de que todo elemento do conjunto de partida fosse parte da função, neste caso, chamando esta relação matemática como *função parcial*. Desta forma, as funções matemáticas são conhecidas como funções totais ou mapeamentos na Computação e a exigência de que o conjunto de partida seja o mesmo conjunto domínio é relaxada para formar as funções parciais.

2.8.3 Definições explícitas e implícitas de funções

Após termos analisado as definições explícitas e implícitas das variáveis, vamos agora considerar como elas são aplicadas ao caso das funções. Por exemplo, as duas equações, a seguir, definem implicitamente a função *implica*.

$$\text{and } [p, \text{ implica } (p, q)] = \text{and } (p, q)$$

$$\text{and } [\text{not } p, \text{ implica } (p, q)] = \text{or } [\text{not } p, \text{ and } (\text{not } p, q)]$$

Estas equações não podem ser usadas explicitamente para avaliar uma expressão como *implica (True, False)*. Usando teoremas da álgebra booleana, estas equações podem ser resolvidas para se chegar à seguinte definição explícita:

$$\text{implica } (p, q) = \text{or } (\text{not } p, q)$$

A definição explícita permite que *implica (True, False)* seja avaliada usando substituição. Uma vantagem da programação funcional é que ela simplifica a transformação de uma definição implícita em uma definição explícita. Isto é muito importante porque as especificações formais de sistemas de softwares, frequentemente, têm a forma de definições implícitas, enquanto as

definições explícitas são, normalmente, fáceis de serem transformadas em programas. Assim, a programação funcional provê uma forma de se passar das especificações formais para programas satisfazendo estas especificações.

Deve ser notado que as definições recursivas são implícitas, por natureza. No entanto, como seu lado esquerdo é simples, ou seja, composto apenas pelo nome da função e seus argumentos, elas podem ser convertidas facilmente em regras de reescritas. Por exemplo, as duas equações seguintes constituem uma definição recursiva de fatorial, para $n \geq 0$.

$$\begin{aligned} fat &: N \rightarrow N \\ fat(n) &= \begin{cases} 1, & \text{se } n = 0 \\ n * fat(n-1), & \text{se } n > 0 \end{cases} \end{aligned}$$

elas podem ser convertidas às seguintes regras de reescritas:

$$\begin{aligned} fat\ n &\Rightarrow n * fat(n-1), & \text{se } n > 0 \\ fat\ 0 &\Rightarrow 1, & \text{se } n = 0 \end{aligned}$$

Estas regras de reescrita nos dizem como transformar uma fórmula contendo *fat*. A realização destas transformações, no entanto, não elimina, necessariamente, a função da fórmula. Por exemplo,

$$2 + fat\ 3 \Rightarrow 2 + 3 * fat\ (3 - 1)$$

No entanto, se a computação termina, então a aplicação repetida das regras de reescrita eliminará *fat* da fórmula, ou seja,

$$\begin{aligned} 2 + fat\ 3 &\Rightarrow 2 + 3 * fat\ (3 - 1) \\ &\Rightarrow 2 + 3 * fat\ 2 \\ &\Rightarrow 2 + 3 * 2 * fat\ (2 - 1) \\ &\Rightarrow 2 + 6 * fat\ 1 \\ &\Rightarrow 2 + 6 * 1 * fat\ (1 - 1) \\ &\Rightarrow 2 + 6 * fat\ 0 \\ &\Rightarrow 2 + 6 * 1 \\ &\Rightarrow 2 + 6 \\ &\Rightarrow 8 \end{aligned}$$


2.8.4 Definições de funções por enumeração

Para a Matemática, uma função é uma associação de valores pertencentes a um conjunto de partida, o domínio, com valores pertencentes a um conjunto de chegada, o contradomínio ou imagem da função. Com esta definição em mente, uma função pode ser representada de duas maneiras. A primeira delas é exibir todos os pares do tipo “(entrada, saída)”, sendo esta uma definição extensionista, também conhecida como definição por enumeração, uma vez que todos os seus elementos são exibidos. É importante caracterizar o domínio e o contradomínio da função.

Por exemplo, a função booleana *not* tem como domínio e contradomínio o conjunto $Bool = \{False, True\}$ e pode ser definida, por extensão ou enumeração, da seguinte forma:

$$\begin{aligned} not &: Bool \rightarrow Bool \\ not\ (True) &= False \\ not\ (False) &= True \end{aligned}$$

De forma similar, as funções *or* (disjunção) e *and* (conjunção) também podem ser definidas por enumeração, da seguinte maneira:

$or : Bool \times Bool \rightarrow Bool$	$and : Bool \times Bool \rightarrow Bool$	
$or (False, False) = False$	$and (False, False) = False$	
$or (False, True) = True$	$and (False, True) = False$	
$or (True, False) = True$	$and (True, False) = False$	
$or (True, True) = True$	$and (True, True) = True$	

2.8.5 Definição de funções por intencionalidade

Não é difícil entender que as definições de funções por enumeração só têm sentido se o domínio e o contradomínio forem finitos e de cardinalidade pequena. A grande maioria das funções não pode ser definida desta forma.

Uma outra maneira de representar uma função é exibir uma propriedade que apenas os elementos desta função a têm. Isto significa apresentar uma regra que informe como cada elemento do conjunto de partida (domínio) deve ser processado para que ele se transforme em um único elemento do conjunto de chegada (imagem). Esta forma de representação de uma função é conhecida como intencionista.

Por exemplo, a função unária f que associa cada número inteiro com a sua quarta potência. O domínio e o contradomínio desta função são \mathbf{Z} e \mathbf{N} , respectivamente. A definição de f é a seguinte:

$$f : \mathbf{Z} \rightarrow \mathbf{N}$$

$$f(n) = n^4$$

Outro exemplo é a função ternária g que associa a cada tripla de números inteiros a soma de seus quadrados. O domínio de g é o produto cartesiano $\mathbf{Z} \times \mathbf{Z} \times \mathbf{Z}$ e seu contradomínio é \mathbf{N} . Desta forma, a função fica definida da seguinte forma:

$$g : \mathbf{Z} \times \mathbf{Z} \times \mathbf{Z} \rightarrow \mathbf{N}$$

$$g(x, y, z) = x^2 + y^2 + z^2$$

2.8.6 Definição de funções por composição

As funções podem ainda ser compostas para formar uma nova função. Como exemplo, a função *implica* pode ser definida da seguinte forma:

$$implica : Bool \times Bool \rightarrow Bool$$

$$implica(x, y) = or(not\ x, y)$$

Neste caso, é necessário saber como as funções a serem compostas, *or* e *not*, são aplicadas. A aplicação de uma função se torna simplesmente um processo de substituição das funções primitivas. Por exemplo, para avaliar a função $implica(False, True)$ é necessário fazer a substituição dos argumentos pelas aplicações das funções primitivas.

$$implica(False, True) = or(not\ False, True) = or(True, True) = True$$

Este processo é independente do domínio, ou seja, é independente de quando se está tratando com funções sobre números, ou funções sobre caracteres, ou funções sobre árvores, ou qualquer outro tipo. Sendo a função f definida por

$$f(x) = h(x, g(x))$$

$$\text{então } f(u(a)) = h(u(a), g(u(a)))$$

independente das definições de g , h , u ou da constante a .

2.8.7 Definição de funções por casos

Frequentemente, a definição de uma função não pode ser expressa pela composição simples de outras funções, sendo necessária a definição da função para vários casos. Por exemplo, a função que retorna o sinal algébrico de uma variável inteira pode ser definida da seguinte forma:

$$\begin{aligned} \text{signal} : Z &\rightarrow Z \\ \text{signal}(x) &= \begin{cases} 1, & \text{se } x > 0 \\ 0, & \text{se } x = 0 \\ -1, & \text{se } x < 0 \end{cases} \end{aligned}$$

De forma similar, a diferença absoluta entre dois números inteiros x e y pode ser definida da seguinte forma:

$$\begin{aligned} \text{difabs} : Z \times Z &\rightarrow Z \\ \text{difabs}(x, y) &= \begin{cases} x - y, & \text{se } x > y \\ y - x, & \text{se } x \leq y \end{cases} \end{aligned}$$

2.8.8 Definição de funções por recursão

Algumas situações existem em que é necessário definir uma função em termos de um número infinito de casos. Por exemplo, a multiplicação de números naturais (\times) pode ser definida por infinitas aplicações da função de adição ($+$). Senão vejamos:

$$m \times n = \begin{cases} 0, & \text{se } m = 0 \\ n, & \text{se } m = 1 \\ n + n, & \text{se } m = 2 \\ n + n + n, & \text{se } m = 3 \\ \vdots & \end{cases}$$

Como não é possível escrever um número infinito de casos, este método de definição de funções só é usual se existir alguma “regularidade” ou algum “princípio de unificação” entre os casos, que permita gerar os casos não definidos, a partir de casos já definidos. Se existir tal princípio de unificação, ele deve ser estabelecido, sendo este o propósito das “definições recursivas”, onde um objeto é definido em termos de si próprio. Por exemplo, uma definição recursiva da multiplicação anterior pode ser:

$$\begin{aligned} \times : N \times N &\rightarrow N \\ m \times n &= \begin{cases} 0, & \text{se } m = 0 \\ n + (m - 1) \times n, & \text{se } m > 0 \end{cases} \end{aligned}$$

Neste caso, uma avaliação de 2×3 é feita por substituição da seguinte forma:

$$2 \times 3 \Rightarrow 3 + (2 - 1) \times 3 \Rightarrow 3 + 1 \times 3 \Rightarrow 3 + 3 + (1 - 1) \times 3 \Rightarrow 6 + (1 - 1) \times 3 \Rightarrow 6 + 0 \times 3 \Rightarrow 6 + 0 \Rightarrow 6$$

A recursão é o método básico de se fazer alguma operação iterativamente.

Exercícios resolvidos

1. Dados dois números naturais, x e y , ambos maiores que zero, defina uma função $\text{mdc}(x, y)$ que dê como resultado o máximo divisor comum entre x e y .

Solução:

$$\begin{aligned} mdc : N^+ \times N^+ &\rightarrow N^+ \\ mdc(x, y) &= \begin{cases} x, & \text{se } x = y \\ mdc(x - y, y), & \text{se } x > y \\ mdc(y, x), & \text{se } x < y \end{cases} \end{aligned}$$

2. Dados dois números naturais, x e y , ambos maiores que zero, defina uma função $div(x, y)$ que dê como resultado a divisão inteira de x por y .

Solução:

$$\begin{aligned} div : N^+ \times N^+ &\rightarrow N \\ div(x, y) &= \begin{cases} 1, & \text{se } x = y \\ 1 + div((x - y), y), & \text{se } x > y \\ 0, & \text{se } x < y \end{cases} \end{aligned}$$

3. Dados dois números naturais, x e y , ambos maiores que zero, defina uma função $mod(x, y)$ que dê como resultado o resto da divisão de x por y .

Solução:

$$\begin{aligned} mod : N^+ \times N^+ &\rightarrow N \\ mod(x, y) &= \begin{cases} 0, & \text{se } x = y \\ mod((x - y), y), & \text{se } x > y \\ x, & \text{se } x < y \end{cases} \end{aligned}$$

4. Dados dois números naturais, x e y , ambos maiores que zero, defina uma função $mmc(x, y)$ que dê, como resultado, o mínimo múltiplo comum entre x e y .

Solução:

Para facilitar vamos construir uma função auxiliar, $mmcaux(x, y, k)$, para algum k natural e $k > 0$, de forma que k seja múltiplo de x e y , ao mesmo tempo. O menor valor de k será o máximo entre os valores de x e y .

$$\begin{aligned} mmcaux : N^+ \times N^+ \times N^+ &\rightarrow N^+ \\ mmcaux(x, y, k) &= \begin{cases} k, & \text{se } (mod(k, x) = 0) \& (mod(k, y) = 0) \\ mmcaux(x, y, k + 1), & \text{senão} \end{cases} \end{aligned}$$

Agora pode-se definir o mínimo múltiplo comum usando esta função auxiliar:

$$\begin{aligned} mmc : N^+ \times N^+ &\rightarrow N^+ \\ mmc(x, y) &= mmcaux(x, y, max(x, y)) \end{aligned}$$

onde a nova função max , que retorna o maior valor entre dois números naturais, pode ser definida de forma simples. Deve ser observado que o mínimo múltiplo comum entre dois números também pode ser encontrado através da divisão do seu produto pelo máximo divisor comum entre eles, ou seja, $mmc(x, y) = div(x * y, mdc(x, y))$.

5. Dado um número natural $n > 1$, defina uma função $numdiv(n)$ que dê como resultado o número de divisores de n , incluindo o número 1.

Solução:

De forma similar a que foi feita no exercício anterior, será construída uma função auxiliar, $quantdiv(n, k)$, para algum k natural e $k > 0$.

$$quantdiv : N^+ \times N^+ \rightarrow N$$

$$quantdiv(n, k) = \begin{cases} 0, & \text{se } n < 2k \\ 1 + quantdiv(n, k + 1), & \text{se } mod(n, k) = 0 \\ quantdiv(n, k + 1), & \text{se } mod(n, k) > 0 \end{cases}$$

assim, a função $numdiv$ pode ser definida da seguinte forma:

$$numdiv : N^+ \rightarrow N^+$$

$$numdiv(n) = quantdiv(n, 1)$$

2.9 Resumo

Este Capítulo foi dedicado à fundamentação teórica das linguagens funcionais, buscando inseri-las no contexto matemático. Esta fundamentação tem como vantagem, primeiramente a constatação de que essas linguagens têm um embasamento teórico sólido e depois, sendo este embasamento oriundo da Matemática, torna estas linguagens mais tratáveis matematicamente e mais fáceis de terem seus programas provados.

A parte histórica mostrada no início do Capítulo foi baseada na Apostila de Rafael Lins [27] e no livro de Brainerd [9]. No entanto, para uma viagem interessante pelos meandros da história da Matemática o leitor deve consultar as referências [8, 12].

A análise da ligação existente entre as linguagens funcionais e a Matemática foi, grande parte, baseada no livro de Bruce MacLennan [30]. Ele adota a metodologia de praticar e depois justificar a parte prática através da teoria. Esta metodologia faz com que o livro seja uma referência muito interessante não apenas pelo método utilizado mas, principalmente, pelo conteúdo muito bem escrito. Outra referência importante para este estudo foi o livro de Antony Davie [11] que apresenta um conteúdo próximo da parte teórica do livro de MacLennan. Não menos importante, foi o livro de Chris Okasaki [33] que apresenta uma teoria de linguagens funcionais bastante profunda e objetiva.

2.10 Exercícios propostos

1. Mostre que a definição explícita da função *implica* (anterior) satisfaz a sua definição implícita.
2. Mostre que a definição explícita da função *implica* é a única solução para a sua definição implícita, ou seja, nenhuma outra função booleana satisfaz estas duas equações, apesar de que devem existir outras formas de expressar esta mesma função. Sugestão: usar tabela verdade.
3. Avalie a expressão 3×5 , usando a definição recursiva da multiplicação mostrada nesta seção.

4. Defina, recursivamente, em termos da multiplicação definida nesta seção, a função potência, onde uma base é elevada a um expoente inteiro não negativo.
5. Defina, recursivamente, a exponenciação de números não negativos elevados a uma potência inteira. Sugestão: use a definição condicional e a resposta do exercício anterior.
6. Defina, recursivamente, a adição de inteiros não negativos, em termos das funções *suc* e *pred*, onde $suc(n) = n + 1$ e $pred(n) = n - 1$, para $n \in \mathbb{Z}$.
7. Defina, recursivamente, a adição e a subtração de inteiros quaisquer, em função das funções *pred* e *suc* do exercício anterior.
8. Dado um número natural $n > 0$, n é dito perfeito se a soma de seus divisores, incluindo o número 1, é igual ao próprio n . O primeiro número natural perfeito é o número 6, porque $6 = 1 + 2 + 3$. Defina uma função *eperfeito*(n) que informe se n é, ou não, um número perfeito.
9. Dado um número natural $n > 0$, defina uma função *eprimo*(n) que informe se n é, ou não, um número primo.
10. Um número inteiro M_n é conhecido como *número de Mersenne* em homenagem ao seu mais ilustre estudioso, Marin Mersenne (Oizé, 8 de Setembro de 1588 - Paris, 1 de Setembro de 1648), matemático, teórico musical, padre mínimo, teólogo e filósofo francês. Os estudos matemáticos, em especial a Teoria dos números, notabilizou-o sobretudo por sua contribuição relativa aos chamados *primos de Mersenne*, que são números de Mersenne que também são primos. Um número inteiro não negativo M_n é um *número de Mersenne* se $M_n = 2^n - 1$, para algum número inteiro n não negativo. Assim, a lista dos números de Mersenne são $\{0, 1, 3, 7, 15, \dots\}$. Defina matematicamente uma função que verifique se um dado número inteiro não negativo é, ou não, um número de Mersenne.
11. Utilizando as definições e funções utilizadas no problema anterior, defina matematicamente uma função que mostre a lista dos *primos de Mersenne* que é composta por números de Mersenne que também são primos. Até o presente momento, são conhecidos apenas 46 *primos de Mersenne* mas, nada indica que só existam apenas estes. Para representar o quadragésimo sexto *primo de Mersenne*, M_{46} , em base decimal, são necessárias 3.461 páginas com 50 linhas por página e 75 dígitos por linha.
12. Considere o algoritmo a seguir que gera uma sequência de números naturais não nulos, a partir de um número natural $n > 0$. Se n for par, divida-o por 2. Se n for ímpar, multiplique-o por 3 e some 1. Repita este processo com o novo valor de n , até que ele seja igual a 1, se for possível. Por exemplo, para $n = 22$, a sequência é: 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2 e 1. Para cada n , define-se o *tamanho do ciclo* de n como a quantidade de números da sequência gerada, incluindo o número 1. No exemplo acima, o tamanho do ciclo para $n = 22$ é 16. Defina uma função *tamciclo*(n) que dê como resultado o tamanho do ciclo de n .

Capítulo 3

λ -cálculo

“Type theories in general date back to the philosopher Bertrand Russel and beyond. They were used in the early 1900’s for the very specific purpose of getting round the paradoxes that had shaken the foundations of mathematics at that time, but their use was later widened until they came to be part of the logicians’ standard bag of technical tools, especially in proof-theory.”
(J. Roger Hindley in [13])

3.1 Introdução

Como foi mencionado na Introdução deste trabalho, o λ -cálculo é utilizado como linguagem intermediária entre uma linguagem funcional de alto nível e a linguagem de máquina. Desta forma, o estudo do λ -cálculo se torna importante para se compreender algumas construções dos programas funcionais, sendo este o objetivo deste Capítulo.

O λ -cálculo foi desenvolvido por Alonzo Church no início dos anos 30, como parte de um sistema de Lógica de ordem superior com o propósito de prover uma fundamentação para a Matemática, dentro da filosofia da escola logicista de Peano-Russel [27]. O λ -cálculo é uma teoria que ressurge do conceito de função como uma regra que associa um argumento a um valor calculado através de uma transformação imposta pela definição da função. Nesta concepção, uma função é representada por um grafo, onde cada nó é um par (argumento, valor).

A Lógica Combinatorial foi inventada antes do λ -cálculo, por volta de 1920, por Moses Schönfinkel, com um propósito semelhante ao do λ -cálculo [9]. No entanto, ela foi redescoberta 10 anos depois, em 1930, por Haskell B. Curry, que foi o maior responsável pelo seu desenvolvimento até cerca de 1970. Em 1935, Kleene e Rosser provaram que o λ -cálculo e a Lógica Combinatorial eram inconsistentes, o que provocou um desinteresse acadêmico pelo tema. No entanto, Curry não desistiu de seu estudo e construiu uma extensão à Lógica Combinatorial para fins de seus estudos, conseguindo sistemas mais fracos que a Lógica clássica de segunda ordem. Em 1975, Dana Scott e Peter Aczel mostraram que seu último sistema apresentava modelos interessantes [9]. Em 1941, Church desistiu de sua pesquisa inicial e apresentou uma sub-teoria que lidava somente com a parte funcional. Esta sub-teoria passou a ser chamada de λ -cálculo puro e foi mostrada ser consistente pelo teorema de Church-Rosser [27]. Usando o λ -cálculo, Church propôs uma formalização da noção de computabilidade efetiva pelo conceito de definibilidade no λ -cálculo. Kleene mostrou que ser definível no λ -cálculo é equivalente à recursividade de Gödel-Herbrand. Concomitantemente, Church formulou a sua conjectura associando a recur-

sividade como a formalização adequada do conceito de computabilidade efetiva. Em 1936, Allan Turing modelou a computação automática e mostrou que a noção resultante (computabilidade de Turing) é equivalente à definibilidade no λ -cálculo. Desta forma, o λ -cálculo pode ser visto como uma linguagem de programação e como tal diversos tipos de problemas de programação puderam ser analisados, principalmente os relacionados com chamadas a procedimentos [27].

Curry e seus colegas [10], Barendregt [5] e outros desenvolveram extensivamente os aspectos sintáticos do λ -cálculo, enquanto Scott [43], principalmente reportado por Stoy [54] e Schmidt [42], se dedicou à semântica da notação, o que veio a facilitar a vida dos usuários futuros do λ -cálculo.

3.2 λ -expressões

O λ -cálculo tem apenas quatro construções: variáveis, constantes, aplicações e abstrações. Suponhamos que sejam dadas uma sequência infinita de símbolos distintos chamados de variáveis e uma sequência finita, infinita ou vazia de símbolos distintos, chamados constantes [37]. Quando a sequência de constantes for vazia, o sistema é chamado puro; em caso contrário, é chamado aplicado. O conjunto de expressões, chamadas de λ -expressões, é definido indutivamente da seguinte forma:

- todas as variáveis e constantes são λ -expressões (chamadas de átomos);
- sendo M e N duas λ -expressões, então (MN) é uma λ -expressão, chamada **combinação** ou **aplicação**;
- sendo M uma λ -expressão e x uma variável qualquer, então $(\lambda x.M)$ é uma λ -expressão, chamada **abstração** ou **função**.

As λ -expressões, assim definidas, podem ser formuladas utilizando a notação BNF, da seguinte forma:

$\langle \lambda - exp \rangle$::	$\langle constante \rangle$	- constantes embutidas
		$\langle variavel \rangle$	- nomes de variáveis
		$(\langle \lambda - exp \rangle \langle \lambda - exp \rangle)$	- combinação ou aplicação
		$(\lambda \langle variavel \rangle . \langle \lambda - exp \rangle)$	- abstração ou função.

Apesar das definições acima serem claras, elas são muito rígidas e, em alguns casos, podem surgir dúvidas em relação às regras de escopo, ou seja, até que ponto de uma λ -expressão uma variável tem influência. Assim, as observações a seguir devem ser utilizadas para dirimir estas dúvidas, que surgem, principalmente, por quem está iniciando os primeiros passos no λ -cálculo. Como exemplo, é comum não se reconhecer imediatamente o escopo de uma variável e, nestes casos, elas podem ser de grande valia.

1. As variáveis são representadas por letras romanas minúsculas.
2. As λ -expressões completas são representadas por letras romanas maiúsculas ou por letras gregas minúsculas (exceto α , β , η e λ que têm significados especiais) ou por letras gregas maiúsculas.
3. Apesar da rigidez mostrada na BNF das λ -expressões, os parênteses devem ser usados apenas para resolver ambiguidades [60]. Por exemplo, $(\lambda x.(\lambda y.(\dots(\lambda z.E)\dots)))$ pode ser escrita como $\lambda x \lambda y \dots \lambda z.E$, ou ainda como $\lambda xy \dots z.E$, significando que as abstrações ou funções são associadas pela direita. No entanto, grupos de combinações ou aplicações de termos combinados são associados pela esquerda, ou seja, $E_1 E_2 E_3 \dots E_n$ significa

$(\dots((E_1 E_2) E_3) \dots E_n)$. Além disso, $\lambda a. CD$ representa $(\lambda a.(CD))$ e não $((\lambda a.C)D)$, estabelecendo que o escopo de uma variável se estende até o primeiro parêntese descasado encontrado a partir da variável, ou atingir o final da λ -expressão.

4. A escolha de constantes, de funções embutidas para serem utilizadas na manipulação destas constantes e/ou números e das funções para o processamento de listas é arbitrária.

Exemplos

São exemplos de λ -expressões:

- | | | |
|----|---|--|
| 1. | z | uma variável |
| 2. | 8 | uma constante |
| 3. | zy | uma combinação ou aplicação $((z)(y))$ |
| 4. | $(\lambda a.(ab))$ | uma λ -abstração ou função |
| 5. | $(\lambda y.y)(\lambda a.(ab))$ | uma combinação ou aplicação |
| 6. | $(a(\lambda z.y))a$ | uma combinação ou aplicação |
| 7. | $(\lambda a.\lambda b.\lambda c.(a(\lambda k.k)(\lambda x.x)))$ | uma combinação ou aplicação |

3.3 A sintaxe do λ -cálculo

Nesta seção, será mostrada a sintaxe do λ -cálculo, ficando a semântica para uma próxima. A sintaxe é importante na construção de λ -expressões de forma correta.

3.3.1 Funções e constantes predefinidas

Em sua forma mais pura, o λ -cálculo não tem funções embutidas, como $+$, $*$, $-$ e $/$. No entanto, será acrescentada uma coleção de tais funções como uma extensão ao λ -cálculo puro, uma vez que o nosso objetivo é construir uma linguagem que seja aplicável. Entre as funções embutidas que serão incluídas, citamos:

- as funções aritméticas $(+, -, *, /)$ e as constantes $0, 1, \dots, 9$;
- as funções lógicas (AND, OR, NOT) e as constantes TRUE e FALSE;
- os caracteres constantes (a, b, \dots) ;
- a função condicional IF: $\text{IF TRUE } E1 \ E2 \rightarrow E1$
 $\text{IF FALSE } E1 \ E2 \rightarrow E2$
- as funções CONS, HEAD e TAIL, onde
 $\text{HEAD (CONS } a \ b) \rightarrow a$
 $\text{TAIL (CONS } a \ b) \rightarrow b$;
- a constante NIL, a lista vazia.

A escolha de funções embutidas é arbitrária. Assim sendo, elas serão adicionadas à medida que as necessidades surgirem. Deve ainda ser mencionado que:

- todas as aplicações de funções são pré-fixas e os parênteses externos são redundantes, podendo e até mesmo devendo ser retirados para evitar confusão visual. Por exemplo, a expressão $(+(*\ 2\ 3)(*\ 8\ 2))$ deve ser escrita como $+(*\ 2\ 3)(*\ 8\ 2)$;

- do ponto de vista de implementação, um programa funcional deve ser visto como uma λ -expressão que vai ser avaliada;
- a avaliação de uma λ -expressão se dá pela seleção repetida de λ -expressões redutíveis, conhecidas como *redexes*¹, e suas reduções. Por exemplo, a λ -expressão $(+(* 2 3) (* 8 2))$ apresenta dois *redexes*: um $(* 2 3)$ e o outro $(* 8 2)$, apesar da expressão toda não representar um *redex* porque seus parâmetros ainda não estão todos avaliados. A avaliação da λ -expressão acima deve ser feita obedecendo a uma das seguintes sequências de reduções:

1. $(+(* 2 3) (* 8 2)) \rightarrow (+ 6 (* 8 2)) \rightarrow (+ 6 16) \rightarrow 22$ ou
2. $(+(* 2 3) (* 8 2)) \rightarrow (+(* 2 3) 16) \rightarrow (+ 6 16) \rightarrow 22$

3.3.2 λ -abstrações

As funções embutidas no λ -cálculo são formalizadas como já mostrado anteriormente. As λ -abstrações são as funções não embutidas e são construídas através do construtor (λ). Por exemplo, $\lambda x.(+ x 1)$ é uma λ -abstração e deve ser interpretada da seguinte forma:

λ	indica que se trata de uma função
x	sobre a variável x
.	que
$(+ x 1)$	adiciona x ao número 1.

Uma λ -abstração tem sempre estes 4 (quatro) elementos: o construtor λ , o parâmetro formal (uma variável, no caso, x) o ponto ($.$) e o corpo da função $(+ x 1)$. Uma λ -abstração pode ser comparada com as funções em uma linguagem de programação imperativa. Por exemplo, a λ -abstração anterior pode ser representada pelo seguinte fragmento de código na linguagem C :

```
int inc (x)
int x;
{ return (x + 1); }
```

Uma observação importante a ser observada é que as funções em uma linguagem de programação convencional têm, obrigatoriamente, um nome, enquanto, no λ -cálculo, as λ -abstrações são funções anônimas.

3.3.3 Aplicação de função e currficação

No λ -cálculo, a aplicação de uma função f a um parâmetro x é denotada por justaposição, ou seja, $f x$. E se a função tiver mais de um argumento? A λ -expressão $+ 3 4$ é interpretada como $(+ 3) 4$, ou seja, como uma função $(+ 3)$ que adiciona 3 ao argumento 4. Esta propriedade se deve ao fato de que o λ -cálculo permite que o resultado da aplicação de uma função seja também uma outra função.

Esta propriedade foi descoberta por Schönfinkel e foi amplamente utilizada por Curry. Por este motivo, passou a ser conhecida como currficação que é uma característica de algumas linguagens, onde uma função com n argumentos pode ser interpretada como n funções de apenas 1 (um) argumento. Isto significa que, para o λ -cálculo, todas as funções têm apenas um argumento.

¹O termo *redex* vem da contração de *reduction expression*.

3.4 A semântica operacional do λ -cálculo

A semântica operacional do λ -cálculo diz respeito às regras utilizadas para converter uma λ -expressão em outra. Na realidade, existem 3 (três) regras de conversão. Antes de serem mostradas, devemos definir alguns termos que são utilizados na aplicação das mesmas. Uma ideia central no estudo das linguagens de programação, da notação matemática e da Lógica simbólica é a de ocorrência livre e ocorrência ligada (ou conectada) de uma variável. Esta idéia, normalmente, provoca confusão em quem está vendo o tema pela primeira vez. Assim, ele será introduzido de forma gradual e intuitiva, utilizando exemplos já conhecidos da Matemática. Assim, vamos considerar o somatório:

$$\sum_{i=1}^n i^2 + 1$$

Nesta expressão, a variável i é uma variável ligada, ou seja, ela está fortemente atrelada ao somatório. Diz-se que ela ocorre ligada nesta expressão. Uma característica importante das variáveis ligadas é que elas podem ser renomeadas sem que o significado da expressão seja alterado. Por exemplo, no somatório anterior, podemos substituir a variável i pela variável k sem alterar o seu significado, desde que substituamos todas as ocorrências da variável ligada i por k , transformando-o no seguinte somatório:

$$\sum_{k=1}^n k^2 + 1$$

De maneira similar, a integral a seguir, em relação à variável ligada x

$$\int_0^1 x^2 - 3x dx$$

representa a mesma integral quando substituímos a variável ligada x por t .

$$\int_0^1 t^2 - 3t dt$$

Na teoria dos conjuntos, o conjunto de todos os x (ligada) tal que $x \geq 0$ é o mesmo conjunto de todos os y (ligada) tal que $y \geq 0$, ou seja,

$$\{x \mid x \geq 0\} \equiv \{y \mid y \geq 0\}$$

Também a proposição “para todo x , $x + 1 > x$ ” é equivalente à proposição “para todo y , $y + 1 > y$ ”, ou seja,

$$\forall x[x + 1 > x] \equiv \forall y[y + 1 > y]$$

Vejamos agora, uma outra expressão, também envolvendo somatório:

$$i \sum_{j=1}^n (j^2 + i - a)$$

Já é sabido que a ocorrência da variável j é ligada. Isto pode ser verificado pelo fato de que ela pode ser trocada por qualquer outra variável, desde que não seja i nem a (elas já ocorrem nesta expressão), sem mudar o significado da expressão. Por exemplo, a expressão

$$i \sum_{k=1}^n (k^2 + i - a)$$

tem, exatamente, o mesmo significado da anterior. Em uma expressão, uma ocorrência não ligada de uma variável é dita ser “livre”. As variáveis i , a e n ocorrem livres nas duas expressões anteriores. Se a ocorrência livre de uma variável for trocada por outra, o significado da expressão é modificado. Além disso, uma variável pode ocorrer ligada em uma expressão e livre em outra. Por exemplo, a variável i ocorre livre em

$$\sum_{j=1}^n j^2 + i - a$$

e ocorre ligada na expressão

$$\sum_{i=1}^m i! \sum_{j=1}^n (j^2 + i - a)$$

O escopo (*binding site*) de um identificador determina a região da expressão na qual o identificador ocorre ligado. Esta região, normalmente, é indicada por alguma convenção léxica, como os parênteses, colchetes ou chaves. Esta região é o corpo da expressão, significando que uma mesma variável pode ocorrer ligada em um ponto e livre em outro.

Como visto, a troca de variáveis ligadas não interfere no significado da expressão. No entanto, esta troca não pode ser feita por uma variável que ocorra livre na expressão, porque ela muda seu significado. Por exemplo, a expressão que representa a soma dos elementos da linha i de uma matriz $A_{m \times n}$ é o somatório

$$\sum_{j=1}^n A_{ij} = A_{i1} + A_{i2} + \dots + A_{in}$$

A variável j ocorre ligada nesta expressão e pode ser trocada por outra, por exemplo, k que ainda não ocorra na expressão.

$$\sum_{k=1}^n A_{ik} = A_{i1} + A_{i2} + \dots + A_{in}$$

cujo significado é o mesmo da expressão anterior. No entanto, observemos que a variável j não pode ser trocada pela variável i porque a expressão se tornaria o somatório de A_{ii} , que representaria o somatório dos elementos da diagonal da matriz e não mais dos elementos da linha i , como inicialmente. Se isto fosse feito, ocorreria um fenômeno conhecido como “colisão de identificadores” que evidentemente deve ser evitado.

Fazendo um paralelo com as linguagens de programação comuns, as variáveis ligadas das expressões correspondem aos parâmetros formais das funções e têm o mesmo *status* das variáveis locais.

Após termos visto as noções de ocorrências ligadas e livres e da colisão de identificadores de maneira intuitiva, vamos agora formalizar estes conceitos.

3.4.1 Ocorrências livres ou ligadas

Seja a λ -expressão $(\lambda x. + x y) 4$. Pelo que foi visto anteriormente, sabe-se que se trata da aplicação de uma função sobre a variável x , onde o corpo da função é $(+ x y)$, a uma outra λ -expressão, que no caso é a constante 4.

A variável x pode ser pensada como um local onde o argumento 4 deve ser colocado. Já para a variável y , este mesmo raciocínio não pode ser aplicado porque não existe este local sinalizado

por uma variável y após uma letra λ . Isto significa que as duas variáveis (x e y) têm *status* distintos.

Formalmente, define-se: “a ocorrência de uma variável é ligada se existir uma λ -abstração a qual esta variável esteja ligada. Em caso contrário, a ocorrência é livre.” A Figura 3.1 mostra um exemplo contendo ocorrências livres e ligadas de variáveis.

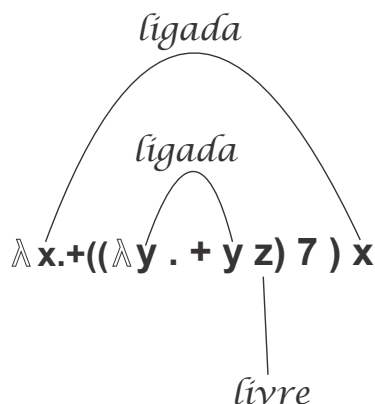


Figura 3.1: Ocorrências livres e ligadas de variáveis.

Deve ser observado que uma mesma variável pode ocorrer livre em um ponto de uma λ -expressão, e ligada em outro, como é o caso da variável x na λ -expressão $+x((\lambda x. + x 1) 4)$. A primeira ocorrência de x é livre e a segunda é ligada.

Sendo x e y duas variáveis e θ e δ duas λ -expressões, podemos dizer que:

- a) x é livre em y se $x = y$. Se $x \neq y$, a ocorrência de x é ligada.
- b) x é livre em $\lambda y.M$ se (x for livre em M) E ($x \neq y$)
- c) x é livre em MN se (x for livre em M) OU (x for livre em N)

Exemplos:

1. x ocorre livre em x , em xy , em $\lambda a.xy$ e em $(\lambda a.xy)(\lambda x.xy)$.
2. x ocorre ligada em y , em $\lambda x.xy$, em $(\lambda x.ax)(y)$, em $\lambda x.abbx$ e em $(\lambda a.xy)(\lambda x.xy)$.

3.4.2 Combinadores

Uma λ -expressão que não apresenta variáveis livres é chamada *fechada* ou *combinador*. Existem alguns combinadores que desempenham um papel especial no estudo do λ -cálculo, conforme foi afirmado na Introdução deste livro, onde foi feita referência aos combinadores **S**, **K** e **I**, nos quais é baseada a *máquina de redução-SK* de Turner [56].

A criação de combinadores tem sido uma tarefa de metodologia desconhecida. Seus construtores devem ter tido algum *insight*, mas não os publicaram. O que se sabe sobre eles é que funcionam e suas execuções produzem os resultados desejados.

Os combinadores mais conhecidos são:

$$I = \lambda x.x$$

$$K = \lambda x\lambda y.x$$

$$S = \lambda x\lambda y\lambda z.xz(yz)$$

$$\Delta = \lambda x.xx$$

$$Y = \lambda f.(\lambda y.f(yy))(\lambda y.f(yy))$$

$$\Theta = \lambda a\lambda b.b(aab)$$

Identidade

Projeção

Composição

Duplicação

Usado na representação de funções recursivas

Também usado na representação de funções recursivas



Exercícios resolvidos:

1. Identifique nas expressões abaixo aquelas que são, ou não, λ -expressões:

- a) a Sim, a é uma variável
- b) 9 Sim, 9 é uma constante
- c) $((\lambda b.b)(\lambda a.ab))$ Não, os parênteses não estão aninhados corretamente
- d) $(\lambda x.)a$ Não, $\lambda x.$ não é uma λ -abstração
- e) $((\lambda x.\lambda y.y)(\lambda y.yyy))((\lambda i.i)(\lambda a.b))$ Sim, porque contém apenas aplicações, abstrações e variáveis.

2. Identifique nas expressões abaixo as ocorrências livres e as ligadas das variáveis:

- a) $\lambda x.xx$ As duas ocorrências da variável x são conectadas
- b) $(\lambda x.\lambda y.x)x$ O último x ocorre livre
- c) $(\lambda x\lambda y.xx)xa$ As duas últimas variáveis ocorrem livres
- d) $(x(\lambda x.y))x$ Todas as variáveis ocorrem livres

A Tabela a seguir mostra um resumo das ocorrências livres e ligadas das variáveis.

Uma ocorrência de uma variável deve ser livre ou ligada	
Definição de ocorrência livre	x ocorre livre em x x ocorre livre em $(E F) \Leftrightarrow x$ ocorre livre em E <u>ou</u> x ocorre livre em F x ocorre livre em $\lambda y.E \Leftrightarrow x$ e y são variáveis distintas e x ocorre livre em E
Definição de ocorrência ligada	x ocorre ligada em $(EF) \Leftrightarrow x$ ocorre ligada em E <u>ou</u> x ocorre ligada em F x ocorre ligada em $\lambda y.E \Leftrightarrow (x \text{ e } y \text{ são a mesma variável e } x \text{ ocorre livre em } E) \text{ ou } x \text{ ocorre ligada em } E$

Exercícios propostos: (baseado em [27])

1. Justifique por que as expressões abaixo não são λ -expressões:

- (a) $(x(\lambda y)z)$
- (b) $(\lambda x.\lambda y.x((\lambda i.ii)(\lambda b.\lambda c.b)(\lambda b.\lambda c.b)))$
- (c) λ
- (d) $\lambda x.\lambda x$

2. Identifique as ocorrências livres e as ocorrências ligadas das variáveis nas expressões abaixo:

- (a) $(\lambda x.xwx)9$
- (b) $(\lambda x\lambda y.x)3b$
- (c) $(\lambda x.yxx)(\lambda i.i)5$
- (d) $(\lambda z.\lambda b.\lambda c.ac(bc)f)(\lambda b.\lambda c.b)(\lambda b.\lambda c.b)$
- (e) $(\lambda x.\lambda y.y)w((\lambda z.zzz)(\lambda w.www))((\lambda a.a)(\lambda a.b))$

3. Quais das seguintes expressões são combinadores?

- (a) $(\lambda x.xx)9$
- (b) $(\lambda x.\lambda y.x)3$
- (c) $(\lambda x.yxx)(\lambda i.i)5$
- (d) $(\lambda a.\lambda b.\lambda c.ac(bc)f)(\lambda b.\lambda c.b)$
- (e) $(\lambda x.\lambda y.y((\lambda z.zzz)(\lambda w.www)))(\lambda a.a)(\lambda a.x)$

3.5 Regras de conversão entre λ -expressões

A forma de avaliação utilizada no λ -cálculo consiste em um processo de **transformação** de uma λ -expressão em outra, normalmente, **mais simples**. Este processo é continuado até atingir uma λ -expressão que não pode mais ser transformada, ou entrar em loop infinito. Esta seção é dedicada a análise de técnicas utilizadas para promover estas transformações. Na realidade elas têm um papel similar ao que as técnicas algébricas têm na solução de equações na Matemática.

Existem três técnicas básicas utilizadas na conversão de uma λ -expressão em outra: α -conversão, β -conversão e η -conversão.

3.5.1 α -conversão

Ao analisarmos as duas λ -expressões $(\lambda x. + x 1)$ e $(\lambda y. + y 1)$, verificamos que elas apresentam o mesmo comportamento, ou seja, a única diferença entre elas está nos nomes dos parâmetros, significando que representam a mesma λ -expressão. Portanto, estas duas λ -expressões devem ser convertíveis uma na outra. Esta conversão é chamada de α -conversão e nos permite trocar os nomes dos parâmetros formais (variáveis ligadas) de qualquer λ -abstração. Este tipo de conversão está de acordo com a mudança de variáveis ligadas, já vista anteriormente. Assim, temos:

1. $(\lambda x. * x 1) \xrightarrow{\alpha} (\lambda y. * y 1)$
2. $\lambda x.x \xrightarrow{\alpha} \lambda y.y \xrightarrow{\alpha} \lambda z.z$
3. $\lambda x.axa \xrightarrow{\alpha} \lambda b.aba$

A partir da definição de α -redução dada anteriormente, pode-se deduzir as seguintes relações:

$$\begin{aligned} \theta &\xrightarrow{\alpha} \theta && \text{(reflexão)} \\ \theta &\xrightarrow{\alpha} \delta \iff \delta \xrightarrow{\alpha} \theta && \text{(simetria)} \\ \theta &\xrightarrow{\alpha} \delta \text{ e } \delta \xrightarrow{\alpha} \gamma \implies \theta \xrightarrow{\alpha} \gamma && \text{(transitividade)} \end{aligned}$$

Uma relação matemática que é reflexiva, simétrica e transitiva é uma “relação de equivalência”. Assim, a α -redução determina uma relação de equivalência entre duas λ -expressões, o que nos permite afirmar que estas duas λ -expressões representam a mesma entidade. Isto significa que o usuário pode utilizar uma ou a outra, de acordo com a conveniência que cada caso requer.

3.5.2 β -conversão

O resultado da aplicação de uma λ -abstração a um argumento é uma instância do corpo desta λ -abstração onde as ocorrências livres do parâmetro formal são trocadas por cópias do argumento. Deve ser observado que uma ocorrência livre de uma variável no corpo de uma λ -abstração corresponde a uma ocorrência ligada desta mesma variável na λ -abstração.

Por exemplo, a aplicação $(\lambda x. + x 1) 4$ é reduzida para $+ 4 1$ que é uma instância do corpo $+ x 1$, onde a **ocorrência livre** de x foi trocada pelo argumento 4. Esta operação é chamada de β -redução.

Exemplos:

1. $(\lambda x. + x x) 5 \xrightarrow{\beta} + 5 5 = 10$ (veja que 5 ocorre livre em $+ x x$)

2. $(\lambda x. 5) 5 \xrightarrow{\beta} 3$ (veja que 5 ocorre ligada em 3)
3. $(\lambda x. (y. - y x)) 4 5 \xrightarrow{\beta} (\lambda y. - y 4) 5 \xrightarrow{\beta} - 5 4 = 1$
4. $(\lambda x. x a x) (\lambda x. x) \xrightarrow{\beta} (\lambda x. x) a (\lambda x. x) \xrightarrow{\beta} a (\lambda x. x)$
5. $(\lambda x. \lambda y. \lambda z. x (y z)) (\lambda x. x) \xrightarrow{\beta} \lambda y. \lambda z. (\lambda x. x) (y z) \xrightarrow{\beta} \lambda y. \lambda z. y z \xrightarrow{\eta} \lambda y. y$

Observações:

1. Nas reduções acima, pode ser observada a currificação em ação, ou seja, as aplicações das λ -abstrações retornando uma outra função (λ -abstração) como resultado.
2. Podemos usar uma redução no sentido oposto, para encontrar uma nova λ -abstração. Por exemplo, $+ 4 1 \rightarrow (\lambda x. + x 1) 4$. A esta operação chamamos de β -abstração. Uma β -conversão é um termo genérico utilizado tanto para uma β -redução quanto para uma β -abstração, simbolizando-se por $\xleftrightarrow{\beta}$. Por exemplo, para este caso, $+ 4 1 \xleftrightarrow{\beta} (\lambda x. + x 1) 4$.
3. Para o caso de se ter uma função como argumento, as substituições são realizadas da mesma forma. Por exemplo, $(\lambda f. f 3) (\lambda x. + x 1) \xrightarrow{\beta} (\lambda x. + x 1) 3 \xrightarrow{\beta} + 3 1 = 4$

3.5.3 η -conversão

Analisemos agora as duas λ -abstrações $(\lambda x. + 1 x)$ e $(+ 1)$. Verifica-se que elas se comportam da mesma maneira quando aplicadas a um mesmo argumento, ou seja, produzem o mesmo resultado. Assim, elas também devem ser convertíveis uma na outra, ou seja, $(\lambda x. + 1 x) \xleftrightarrow{\eta} (+ 1)$.

Formalmente, $(\lambda x. F x) \xleftrightarrow{\eta} F$, desde que x não ocorra livre em F .

Exemplos:

1. A λ -expressão $(\lambda x. + x x)$ não é η -redutível a $(+ x)$ porque a variável x ocorre livre em $(+ x)$.
2. A sequência de reduções, a seguir, é válida: $\lambda x. \lambda y. \lambda z. x y z \xrightarrow{\eta} \lambda x. \lambda y. x y \xrightarrow{\eta} \lambda x. x$.

Intuitivamente, a η -redução reflete um princípio importante na computação, conhecido como “princípio da extensibilidade”, utilizado na implementação da igualdade de funções. Este princípio estabelece que “duas funções são iguais se elas sempre apresentam os mesmos resultados para as mesmas entradas”. Ele é importante porque oferece a possibilidade de escolha entre mais de um algoritmo para implementar a igualdade entre funções. A escolha deve ser feita levando em consideração o desempenho das implementações, um fato que não tem importância para os matemáticos, mas para os profissionais da Computação, tem um significado fundamental.

3.5.4 Convertibilidade de λ -expressões

Diz-se que uma λ -expressão θ se β -converte a uma outra λ -expressão δ , se existir uma sequência de λ -expressões $\langle \theta_1, \theta_2, \dots, \theta_n \rangle$, onde

- a) $\theta = \theta_1$ e $\theta_n = \delta$ e
b) $\theta_i \xrightarrow{\beta} \theta_{i+1}$ ou $\theta_{i+1} \xrightarrow{\beta} \theta_i$ para $i = 1, 2, \dots, n$

A Figura 3.2 mostra graficamente uma interpretação para esta definição.

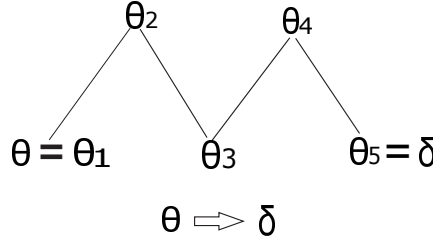


Figura 3.2: Interpretação gráfica de uma sequência de β -reduções.

Esta forma de definição de β -redução entre duas λ -expressões nos permite afirmar que se uma λ -expressão θ se β -reduz a uma λ -expressão θ_1 e uma outra λ -expressão δ também se β -reduz a θ_1 , então a λ -expressão θ também se β -reduz a δ . Dito de outra forma, se duas λ -expressões distintas se β -reduzem a uma terceira, então elas também são β -redutíveis entre si.

Esta forma de definição é muito utilizada em demonstrações de convertibilidade entre λ -expressões e será utilizada mais adiante, ainda neste Capítulo.

Exercícios:

1. Verifique se $K(\lambda I) \xrightarrow{\beta} \lambda a.(III)$.
2. Verifique se $\lambda x(\lambda y.Iy)x \xrightarrow{\eta} \lambda x.I(\lambda y.xy)$.
3. Verifique se $SK \xrightarrow{\beta} KI$.

3.5.5 Captura

Deve ser tomado algum cuidado na escolha dos nomes dos parâmetros, uma vez que mais de um deles podem ter o mesmo nome. Por exemplo,

$$\begin{array}{ll}
(\lambda x.(\lambda x. + (- x 1))x 3)9 & \text{linha 1} \\
\xrightarrow{\beta} (\lambda x. + (- x 1))9 3 & \text{linha 2} \\
\xrightarrow{\beta} +(- 9 1)3 & \text{linha 3} \\
= + 8 3 = 11 & \text{linha 4}
\end{array}$$

Como visto, o x interno de $(- x 1)$ da linha 1 deste exemplo não foi substituído na linha 2 porque ele não ocorre livre em $(\lambda x. + (- x 1))$ que é o corpo da λ -abstração mais externa.

O uso de nomes de variáveis pode algumas vezes criar situações confusas envolvendo β -reduções. Por exemplo, seja a λ -expressão $(\lambda x.(\lambda y.yx))y$. O y mais externo desta λ -expressão ocorre livre. Se executarmos uma β -redução vamos obter a λ -expressão $\lambda y.yy$. Esta λ -expressão resultante apresenta um significado distinto da anterior porque o y externo, que era livre, tornou-se conectado.

Esta situação é conhecida como o problema de captura de variáveis e pode provocar ambiguidades na avaliação mecânica de λ -expressões. A solução trivial para este problema é efetuar α -conversões antes de realizar a β -conversão para evitar que variáveis distintas, mas homônimas, sejam confundidas. No caso da λ -expressão citada, uma sequência correta de reduções seria:

$$(\lambda x.(\lambda y.yx))y \xrightarrow{\alpha} (\lambda x.(\lambda z.zx))y \xrightarrow{\beta} \lambda z.zy$$

O mesmo problema pode ser verificado na λ -expressão $(\lambda x.\lambda z.xz)(\lambda y.yz)$ que seria β -redutível a $(\lambda z.(\lambda y.yz)z)$ e o z de $(\lambda y.yz)$ perderia seu contexto original, sendo capturado pelo λz de $(\lambda x.\lambda z.xz)$. Neste caso, uma possível sequência de reduções seria $(\lambda x.\lambda z.xz)(\lambda y.yz) \xrightarrow{\alpha} (\lambda x.\lambda a.xa)(\lambda y.yz) \xrightarrow{\beta} (\lambda a.(\lambda y.yz)a) \xrightarrow{\eta} \lambda y.yz$.

Este problema também pode aparecer nas linguagens de programação imperativas tradicionais, como C, Java e outras. Suponhamos que na declaração de um procedimento se utilize uma variável local denominada x e que na execução deste mesmo procedimento ele receba como parâmetro real uma variável também denominada x , declarada fora do procedimento, portanto não local. Uma referência à variável não local x será feita, na realidade, à variável x local, ou seja, a variável x externa foi capturada pela variável x interna. Isto acontece porque as variáveis têm o mesmo nome (são homônimas), apesar de representarem entidades distintas.

Exemplos resolvidos:

1. A sequência de reduções para $(\lambda x.\lambda y + x((\lambda x. - x 3)y))5 6$ é:

$$\begin{aligned} & (\lambda x.\lambda y + x((\lambda x. - x 3)y))5 6 \\ & \xrightarrow{\beta} (\lambda y. + 5((\lambda x. - x 3)y))6 \\ & \xrightarrow{\beta} + 5((\lambda x. - x 3)6) \\ & \xrightarrow{\beta} + 5(- 6 3) \\ & = + 5 3 \\ & = 8 \end{aligned}$$

2. As funções embutidas podem ser construídas como quaisquer outras. Por exemplo,

$$\begin{aligned} CONS &= (\lambda a.\lambda b.\lambda f.fab), \\ HEAD &= (\lambda c.c(\lambda a.\lambda b.a)) \text{ e} \\ TAIL &= (\lambda c.c(\lambda a.\lambda b.b)) . \end{aligned}$$

Vejamos agora uma sequência de reduções envolvendo estas duas funções:

$$\begin{aligned} HEAD (CONS p q) &= (\lambda c.c(\lambda a.\lambda b.a)) (CONS p q) \\ &\xrightarrow{\beta} (CONS p q) (\lambda a.\lambda b.a) \\ &= ((\lambda a.\lambda b.\lambda f.fab)p q)(\lambda a.\lambda b.a) \\ &\xrightarrow{\beta} ((\lambda b.\lambda f.fpb)q)(\lambda a.\lambda b.a) \\ &\xrightarrow{\beta} (\lambda f.fpq)(\lambda a.\lambda b.a) \\ &\xrightarrow{\beta} (\lambda a.\lambda b.a)p q \\ &\xrightarrow{\beta} (\lambda b.p)q \\ &\xrightarrow{\beta} p \end{aligned}$$

Isto significa que não há a necessidade de que os construtores *HEAD*, *CONS*, *TAIL* ou qualquer função sejam predefinidos. Na realidade, eles existem apenas por questão de eficiência.

Conclusão

1. **Troca de nomes:** α -conversão permite trocar o nome de um parâmetro de forma consistente.
2. **Aplicação de função:** β -conversão permite aplicar uma λ -abstração a um argumento, construindo uma nova instância do corpo da λ -abstração.
3. **Eliminação de λ -abstrações redundantes:** η -redução pode, algumas vezes, eliminar λ -abstrações redundantes.

3.5.6 Provando a conversibilidade

Muito frequentemente, nos deparamos com casos em que a prova de conversibilidade entre duas λ -abstrações é bastante complexa. Por exemplo, as λ -expressões $IF\ TRUE ((\lambda p.p)3)$ e $(\lambda x.3)$ denotam a mesma função, ou seja, representam a função que sempre retorna o valor 3, independente dos valores dos argumentos reais. Assim, espera-se que elas sejam conversíveis uma na outra, já que denotam a mesma função. Realizando as λ -conversões sobre a primeira, temos:

$$\begin{aligned} & IF\ TRUE ((\lambda p.p)3) \\ & \xrightarrow{\beta} IF\ TRUE\ 3 \\ & \xrightarrow{\eta} (\lambda x. IF\ TRUE\ 3\ x) \quad \text{e pela definição de } IF \\ & = (\lambda x.3) \end{aligned}$$

Um método alternativo utilizado para provar a conversibilidade de duas λ -expressões que denotam a mesma função, consiste em aplicar as duas λ -expressões a um mesmo argumento arbitrário, por exemplo, w . Na Tabela a seguir está mostrado como é feita a prova de conversibilidade das duas funções anteriores, utilizando esta técnica.

$IF\ TRUE ((\lambda p.p)3)w$ $\rightarrow (\lambda p.p)3$ pela def de IF $\xrightarrow{\beta} 3$	$(\lambda x.3)w$ $\xrightarrow{\beta} 3$
--	---

Portanto, $IF\ TRUE ((\lambda p.p)3) \leftrightarrow (\lambda x.3)$.

Este esquema de prova tem a vantagem de usar apenas redução, evitando o uso explícito de η -reduções.

3.5.7 Uma nova notação

A aplicação das regras de conversão nem sempre é tão simples e direta como as mostradas nos exemplos anteriores. Vamos introduzir agora uma notação bastante utilizada na conversão de λ -expressões, principalmente na implementação computacional de redutores, por ser bastante intuitiva.

A notação $E[M/x]$ significa que, na expressão E , todas as ocorrências livres de x serão substituídas por M . Esta notação mostra explicitamente que variável está sendo trocada e por quem. Deve ser notada a particularidade da notação $E[M/x]$ que, apesar de representar o termo lido como “M por x” em português, representa a troca de “x por M”. Esta inversão de significado deve-se tão somente à diferença de interpretação existente entre o português e o inglês, que foi o idioma no qual a notação foi descrita.

Esta notação também pode ser utilizada em α -conversões. A Tabela 3.1 mostra um resumo desta notação.

Exemplos: (baseados em [27])

1. $x[\theta/x] = \theta$
2. $x[\theta/y] = x$
3. $(abx)(xy)[\theta/x] = (ab\theta)(\theta y)$
4. $(axxb)(bx)(xx)[\theta/x] = (a\theta\theta b)(b\theta)(\theta\theta)$

Tabela 3.1: Resumo das conversões.

$x[M/x] = M$ $c[M/x] = c$, onde c é uma constante distinta de x $(EF)[M/x] = E[M/x]F[M/x]$ $(\lambda x.E)[M/x] = \lambda x.E$ $(\lambda y.E)[M/x]$ onde y é qualquer variável distinta de x $\quad = \lambda y.E[M/x]$, se x não ocorre livre em E OU y não ocorre livre em M $\quad = \lambda z.(E[z/y])[M/x]$, onde z é o nome de uma nova variável que não ocorre livre em E ou em M .
Definições: α -conversão: se y não é livre em E , então $(\lambda x.E) \xrightarrow{\alpha} (\lambda y.E[y/x])$ β -conversão: $(\lambda x.E)M \xrightarrow{\beta} E[M/x]$ η -conversão: se x não é livre em E e E denota uma função, então $(\lambda x.Ex) \xrightarrow{\eta} E$

$$5. (\lambda a.xy)(\lambda x.xy)[S/x] = (\lambda a.Sy)(\lambda x.xy)$$

$$6. S[K/x] = S$$

$$7. (\lambda x.xy)[\lambda a.xy/y] \xrightarrow{\alpha} (\lambda z.zy)[\lambda a.xy/y] = \lambda z.z(\lambda a.xy)$$

$$8. (\lambda x.xax)(\lambda x.x) \xrightarrow{\beta} xax[\lambda x.x/x] = (x[\lambda x.x/x])ax[\lambda x.x/x] = (\lambda x.x)ax[\lambda x.x/x] = (\lambda x.x)a(\lambda x.x) \xrightarrow{\beta} x[a/x](\lambda x.x) = a(\lambda x.x)$$

3.6 Ordem de redução

Um *redex* é uma λ -expressão na qual todos os parâmetros necessários para que uma operação possa ser feita estão prontos para serem utilizados. Se uma expressão não contiver qualquer *redex* a sua avaliação está completa e, neste caso, diz-se que a expressão está na sua forma normal.

No entanto, pode acontecer que uma expressão tenha mais de um *redex* e, neste caso, existem mais de um caminho a serem seguidos na sequência de avaliações. Já sabemos, pela propriedade de Church-Rosser, vista na **Seção 1.5**, que a utilização de qualquer uma das sequências de redução leva ao mesmo resultado, se ele existir. Por exemplo, $+(* 3 4)(* 7 8)$ apresenta dois *redexes*: $* 3 4$ e $* 7 8$. Se for escolhido o *redex* mais à esquerda, teremos a seguinte sequência de reduções:

$$\begin{aligned} &+(* 3 4)(* 7 8) \\ &\rightarrow +12 (* 7 8) \\ &\rightarrow +12 56 \\ &\rightarrow 68 \end{aligned}$$

Se, no entanto, a escolha recair sobre o *redex* mais à direita, a sequência de redução será:

$$\begin{aligned} &+(* 3 4)(* 7 8) \\ &\rightarrow + (* 3 4) 56 \\ &\rightarrow +12 56 \\ &\rightarrow 68 \end{aligned}$$

Apesar dos resultados das duas sequências de avaliação serem iguais, algumas observações devem ser feitas:

1. Nem toda λ -expressão tem uma forma normal. Por exemplo, a λ -expressão $(\Delta\Delta)$, onde $\Delta = (\lambda x.xx)$, tem a seguinte sequência de reduções:

$$(\lambda x.xx)(\lambda x.xx) \xrightarrow{\beta} (\lambda x.xx)(\lambda x.xx) \xrightarrow{\beta} (\lambda x.xx)(\lambda x.xx) \xrightarrow{\beta} \dots$$

correspondendo a um *loop* infinito nas linguagens imperativas.

2. Algumas sequências de redução podem atingir a forma normal, enquanto outras não. Por exemplo, $(\lambda x.3)(\Delta\Delta)$ pode ser avaliada para 3, escolhendo o primeiro *redex*. No entanto, se for escolhido o segundo, Δ aplicado a Δ , o resultado será um *loop* infinito, sem atingir a forma normal.

Até este ponto, temos insistido, mesmo sem provas, que sequências diferentes de reduções não podem levar a formas normais diferentes. Dois teoremas, descritos a seguir, mas sem demonstrações, garantem esta propriedade. Sugere-se ao leitor pesquisar estas demonstrações na bibliografia indicada, por não fazer parte do objetivo deste livro. Estas demonstrações requerem conhecimentos matemáticos e da **Teoria da Computação** mais avançados dos que os exigidos como pré-requisitos para este estudo.

Teorema 1 de Churh-Rosser (CRT-I). “Se $E_1 \leftrightarrow E_2$, então existe uma λ -expressão E tal que $E_1 \leftrightarrow E$ e $E_2 \leftrightarrow E$ ”.

Prova: Exercício.

Corolário. Nenhuma expressão pode ser convertida a duas formas normais distintas.

Prova: Suponha que uma λ -expressão, E , seja redutível a duas λ -expressões distintas, E_1 e E_2 e que E_1 e E_2 estejam na forma normal. Pelo teorema **CRT-1**, anterior, existe uma λ -expressão θ_1 , tal que $E \leftrightarrow \theta_1$ e $E_1 \leftrightarrow \theta_1$. Pelo mesmo teorema, também existe uma λ -expressão θ_2 , tal que $E \leftrightarrow \theta_2$ e $E_2 \leftrightarrow \theta_2$. Mas E_1 e E_2 estão na forma normal. Portanto, não podem ser redutíveis. Logo, é uma contradição, que surgiu a partir da suposição de que existiam duas formas normais distintas para a mesma λ -expressão. Portanto, é impossível que uma λ -expressão tenha mais de uma forma normal.

Informalmente, todas as sequências de reduções que terminam, chegarão inequivocamente ao mesmo resultado.

Este teorema de Church-Rosser é conhecido como *teorema da atingibilidade e unicidade da forma normal* e tem uma longa história. Ele foi primeiramente demonstrado por Alonzo Church e J. Barkley Rosser em 1936 [30]. No entanto, esta demonstração era tão longa e complicada que muito do trabalho de pesquisa posterior foi dedicado à descoberta de formas mais simples de demonstração deste teorema. Assim, ele foi demonstrado de várias formas, para diferentes propósitos, todas elas apresentando alguma dificuldade. Uma demonstração mais simples foi apresentada pelo próprio J. Barkley Rosser em 1982 e uma versão mais rigorosa foi feita por Barendregt em 1984 [30].

Um esquema de prova exibido por MacLennan se baseia na descrição de uma propriedade, conhecida como “*propriedade do diamante*”. Diz-se que uma relação R tem a propriedade do diamante se e somente se, para todas as fórmulas bem formadas X , X_1 e X_2 vale: se $X R X_1$ e $X R X_2$, então existe uma fórmula bem formada X' tal que $X_1 R X'$ e $X_2 R X'$. Este esquema pode ser visualizado graficamente na Figura 3.3.

Usando a propriedade do diamante, o teorema de Church-Rosser pode ser descrito da seguinte forma: “*a redução no λ -cálculo tem a propriedade do diamante*”.

Um *redex* é dito ser “*leftmost-outermost*” em uma λ -expressão, se todos os outros *redexes* desta λ -expressão estiverem à sua direita ou dentro dele. Além disso, diz-se que uma sequência de reduções é feita na “*ordem normal*” se o *redex* escolhido para execução em cada estágio for *leftmost-outermost*.

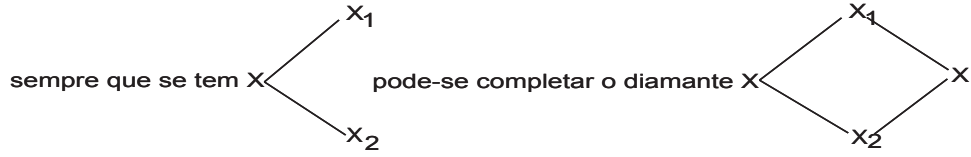


Figura 3.3: Interpretação gráfica da propriedade do diamante.

Exercício. Encontre o *redex leftmost-outermost* nas seguintes λ -expressões:

1. $(\lambda x. \lambda a. (\lambda b. bab)(xa))(\Delta K \Delta)$
2. $x(\lambda a. aa)((\lambda b. (\lambda a. b)b)(\lambda a. aa))(\lambda x. xxx)$
3. $\lambda a. \lambda b. Sa(\Delta b)b$

Teorema 2 de Church-Rosser (CRT II). “Se $E_1 \leftrightarrow E_2$ e E_2 está na forma normal, então existe uma ordem normal de sequências de redução de E_1 para E_2 ”.

Este teorema é também conhecido como “*teorema da normalização*” e significa que existe, no máximo, um resultado e a ordem normal o encontra, se ele existir. A ordem normal de redução especifica que o *redex leftmost-outermost* deve ser escolhido primeiramente. Dito de outra forma, “a redução do redex mais externo e mais à esquerda, em cada ponto da sequência de reduções, leva até a forma normal, se ela existir”.

Exemplo. Utilize a ordem normal de redução para determinar a forma normal da λ -expressão $(\lambda x. xx)((\lambda y. y)(\lambda z. z))$.

$$\begin{aligned}
 (\lambda x. xx)((\lambda y. y)(\lambda z. z)) &\xrightarrow{\beta} ((\lambda y. y)(\lambda z. z))((\lambda y. y)(\lambda z. z)) \\
 &\xrightarrow{\beta} (\lambda z. z)((\lambda y. y)(\lambda z. z)) \\
 &\xrightarrow{\beta} (\lambda y. y)(\lambda z. z) \\
 &\xrightarrow{\beta} \lambda z. z
 \end{aligned}$$

A escolha pela ordem normal de redução garante encontrar a forma normal, se ela existir. No entanto, isto não quer dizer que este seja o melhor método a ser utilizado na redução de λ -expressões. Normalmente, este caminho é o que apresenta o pior desempenho. Então por que este caminho deve ser seguido? A resposta é que a ordem normal de execução apresenta um método que pode ser implementado computacionalmente e isto é o que se busca em nosso estudo, ou seja, procura-se um método para o qual exista um procedimento eletro-mecânico para segui-lo e encontrar a forma normal, se ela existir. Na realidade, muitas otimizações foram feitas buscando melhorar o desempenho deste procedimento que, em última análise, representam melhorias de desempenhos na execução de programas funcionais. Como exemplo de apenas uma destas otimizações pode-se citar David Turner [56].

3.7 Funções recursivas

A idéia de escolher o λ -cálculo como linguagem intermediária entre as linguagens funcionais, de alto nível, e a linguagem de máquina significa que todas as funções devem ser traduzidas para ele. Na realidade, os programas funcionais representam apenas uma forma mais adequada de λ -expressões. Diz-se que os programas funcionais são “*açucaramentos sintáticos*” de λ -expressões, para torná-las mais fáceis de serem tratadas e compreendidas.

No entanto existe um problema a ser resolvido, relacionado ao fato de que uma das características notáveis dos programas funcionais é a utilização massiva de funções recursivas [37]

e estas funções não têm correspondentes no λ -cálculo, conforme visto até aqui. Isto acontece porque, no λ -cálculo, as funções são anônimas e, portanto, não podem ser chamadas recursivamente. É necessária uma forma de implementar funções recursivas no λ -cálculo. Vamos mostrar como estas funções são traduzidas para o λ -cálculo, sem a necessidade de qualquer extensão.

Voltemos, momentaneamente, nossa atenção para a definição matemática da função $f(x) = x^3 - x$. Esta função nos informa como obter valores para f a partir de cada possível valor de x , ou seja, aplicando a função f a cada um destes valores de x . Como um caso particular, vamos escolher valores de x para os quais a função f aplicada a eles tenha como resultado o próprio valor de x , ou seja, vamos procurar valores de x que satisfaçam a igualdade $f(x) = x$. Assim, estaremos procurando valores de x para os quais $x^3 - x$ seja igual ao próprio x . Um tal valor para x é 0, porque $f(0) = 0$. Mas $x = \pm 2^{1/2}$ são outros valores que também satisfazem a igualdade $f(x) = x$. Os valores que satisfazem esta equação são chamados de “*pontos fixos*” da função f e representam um campo importante de estudo matemático, destacando-se o “teorema do ponto fixo”, além de outros.

Os pontos fixos apresentam características importantes, no entanto o nosso interesse aqui se prende exclusivamente em utilizá-los na construção de funções recursivas no λ -cálculo. No entanto recomendamos aos leitores que estudem este tema na literatura matemática adequada onde várias outras aplicações importantes podem ser vistas.

Consideremos agora a seguinte definição recursiva da função fatorial:

$$FAT = (\lambda n . IF (= n 0) 1 (* n (FAT (- n 1))))$$

Nesta definição, damos um nome a uma λ -abstração (FAT) e nos referimos a ele dentro da λ -abstração. Este tipo de construtor não é provido pelo λ -cálculo porque as λ -abstrações são funções anônimas e, portanto, elas não podem fazer referências a nomes.

Vamos colocar a λ -expressão FAT em uma forma mais adequada ao nosso desenvolvimento. Teremos então $FAT = \lambda n . (... FAT ...)$, onde os pontos representam as outras partes da função que não nos interessam, neste momento. Podemos fazer uma λ -abstração em FAT , transformando-a em $FAT = (\lambda f . (\lambda n . (... f ...))) FAT$, sendo f uma variável.

Esta função pode ser escrita na forma $FAT = H FAT$ onde, $H = (\lambda f . (\lambda n . (... f ...)))$. Esta definição de H é adequada aos nossos propósitos, uma vez que ela é uma λ -abstração ordinária e não usa recursão. A equação $FAT = H FAT$ estabelece que quando a função H é aplicada a FAT o resultado é o próprio FAT . Isto significa que FAT é um ponto fixo de H .

Vamos agora procurar um ponto fixo genérico para H . Para isto vamos criar uma função, Y , que tome H como argumento e retorne um ponto fixo da função, como resultado. Assim Y deve ser tal que $Y H$ seja um ponto fixo de H . Portanto, $H(Y H) = Y H$. Por este motivo, Y é chamado de *combinador de ponto fixo*. Se formos capazes de produzir tal combinador, nosso problema estará resolvido.

Como $Y H$ retorna um ponto fixo de H e FAT é um destes pontos, então $FAT = Y H$. Esta definição não é recursiva e atende às nossas necessidades. Para verificar isto, vamos computar $(FAT 1)$ utilizando as definições de FAT e de H , dadas anteriormente e que o mecanismo de cálculo obedece à ordem normal de redução ou seja, *leftmost-outermost*.

$$FAT = Y H$$

$$H = \lambda f . \lambda n . IF (= n 0) 1 (* n (f (- n 1)))$$

Então

$$\begin{aligned}
FAT\ 1 &= YH\ 1 \\
&= H\ (Y\ H)\ 1 \\
&= (\lambda f.\lambda n.IF\ (= \ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))\ (Y\ H)\ 1 \\
&\xrightarrow{\beta} (\lambda n.IF\ (= \ n\ 0)\ 1\ (*\ n\ ((Y\ H)\ (-\ n\ 1))))\ 1 \\
&\xrightarrow{\beta} (IF\ (= \ 1\ 0)\ 1\ (*\ 1\ ((Y\ H)\ (-\ 1\ 1)))) \\
&= (*\ 1\ ((Y\ H)\ (-\ 1\ 1))) \quad \text{pela definição de IF} \\
&= (*\ 1\ ((H\ (Y\ H))\ (-\ 1\ 1))) \quad \text{pelo fato de YH ser um ponto fixo de H} \\
&= (*\ 1\ ((\lambda f.\lambda n.\ IF\ (= \ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))\ (Y\ H)\ (-\ 1\ 1))) \\
&\xrightarrow{\beta} (*\ 1\ (\lambda n.\ IF\ (= \ n\ 0)\ 1\ (*\ n\ (Y\ H\ (-\ n\ 1))))\ (-\ 1\ 1)) \\
&\xrightarrow{\beta} (*\ 1\ (IF\ (= \ (-\ 1\ 1)\ 0)\ 1\ (*\ (-\ 1\ 1)\ (Y\ H\ (-\ (-\ 1\ 1)\ 1))))) \\
&= (*\ 1\ (IF\ (= \ 0\ 0)\ 1\ (*\ 0\ (Y\ H\ (-\ 0\ 1))))) \\
&= (*\ 1\ (IF\ TRUE\ 1\ (*\ 0\ (Y\ H\ (-\ 0\ 1))))) \\
&= (*\ 1\ 1) \quad \text{pela definição de IF} \\
&= 1
\end{aligned}$$

A forma como o combinador **Y** é definido já foi mostrada anteriormente, no entanto ela será novamente vista aqui, usando apenas algumas renomeações de variáveis:

$$Y = \lambda h. (\lambda x. h\ (x\ x))\ (\lambda x. h\ (x\ x))$$

Vamos agora avaliar $Y\ H$:

$$\begin{aligned}
YH &= (\lambda h. (\lambda x. h\ (x\ x))\ (\lambda x. h\ (x\ x)))\ H \\
&\xrightarrow{\beta} (\lambda x. H\ (x\ x))\ (\lambda x. H\ (x\ x)) \\
&\xrightarrow{\beta} H\ ((\lambda x. H\ (x\ x))\ (\lambda x. H\ (x\ x))) \\
&= H\ (YH)
\end{aligned}$$

Este resultado confirma o fato de que $Y\ H$ é um ponto fixo de H ou seja, o combinador Y , quando aplicado a uma função, retorna um ponto fixo desta função.

3.8 Algumas definições

Para concluir este Capítulo, vamos deixar algumas definições de funções e de outros objetos. Por exemplo, os números naturais podem ser definidos como λ -expressões, conhecidas como os “*numerais de Church*”. Deve ser observado que algumas definições utilizam o caractere \$ quando referenciam outras definições, para diferenciar de alguma variável.

1. $F = \lambda f.\lambda n.\$iszero\ n\ \$one\ (\$mul\ n\ (f\ (\$pred\ n)))$
2. $Fv = \lambda f.\lambda n.\$iszero\ n\ (\lambda d.\$one)(\lambda d.\$mul\ n\ (f\ (\$pred\ n)))\ (\lambda d.d)$
3. $I = \lambda x.x$
4. $K = \lambda x.\lambda y.x$
5. $\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$
6. $S = \lambda f.\lambda g.\lambda x.f\ x\ (g\ x)$
7. $Y = \lambda h.(\lambda x.h(x\ x))(\lambda x.h(x\ x))$
8. $YT = (\lambda x.\lambda y.y(x\ x\ y))(\lambda x.\lambda y.y(x\ x\ y))$
9. $Yp = \lambda h.(\lambda x.h(\lambda a.x\ x\ a))\ (\lambda x.h(\lambda a.x\ x\ a))$

10. $Yv = \lambda h.(\lambda x.\lambda a.h(x\ x)a) (\lambda x.\lambda a.h(x\ x)a)$
11. $add = \lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$
12. $append = \$Y(\lambda g.\lambda z.\lambda w.\$null\ z\ w\ (\$cons\ (\$hd\ z)\ (g\ (\$tl\ z)\ w)))$
13. $cons = \lambda x.\lambda y.\$pair\ \$false\ (\$pair\ x\ y)$
14. $exp = \lambda m.\lambda n.\lambda f.\lambda x.m\ n\ f\ x$
15. $false = x.\lambda y.y$
16. $fst = \lambda p.p\$true$
17. $hd = \lambda z.\lambda \$fst\ (\$snd\ z)$
18. $iszero = \lambda n.n(\lambda v.\$false)\ \$true$
19. $mul = \lambda m.\lambda n.\lambda f.\lambda x.m\ (n\ f)x$
20. $next = \lambda p.\$pair\ (\$snd\ p)\ (\$succ\ (\$snd\ p))$
21. $nil = \lambda z.z$
22. $null = \$fst$
23. $pair = \lambda x.\lambda y.\lambda s.s\ x\ y$
24. $pred = \lambda n.\$fst\ (n\ \$next\ (\$pair\ \$zero\ \$zero))$
25. $snd = \lambda p.p\$false$
26. $succ = \lambda n.\lambda f.\lambda x.f(n\ f\ x)$
27. $tl = \lambda z.\$snd\ (\$snd\ z)$
28. $true = \lambda x.\lambda y.x$
29. $zero = \lambda f.\lambda x.x$
30. $one = \lambda f.\lambda x.f\ x$
31. $two = \lambda f.\lambda x.f(f\ x)$
32. $three = \lambda f.\lambda x.f(f(f\ x))$
33. $four = \lambda f.\lambda x.f(f(f(f\ x)))$
34. $five = \lambda f.\lambda x.f(f(f(f(f\ x))))$

3.9 Resumo

Este Capítulo foi todo dedicado ao estudo do λ -cálculo. Ele se fez necessário dada a importância que esta teoria tem na fundamentação das linguagens funcionais. Seu papel é semelhante ao desempenhado pela linguagem Assembly na tradução de programas em linguagens imperativas para o código de máquina. No entanto, deve ser salientado que esta teoria matemática não é tão simples como aqui parece. Esta abordagem simples foi adotada, dado o caráter introdutório exigido para se entender como o λ -cálculo é utilizado na compilação de linguagens funcionais.

Quem quiser se aprofundar neste tema deve consultar a bibliografia específica. As notas de aula de Rafael Lins [27] e de Peter Welch [60] representam um bom começo, dada a grande quantidade de exercícios indicados e resolvidos. Quem estiver interessado em detalhes mais aprofundados sobre a implementação do λ -cálculo deve consultar o livro de Peyton Jones [37]. Uma abordagem mais teórica desta linguagem, pode ser encontrada nos livros de Bruce MacLennan [30] e de Antony Davie [11].

3.10 Exercícios propostos


1. Insira os parênteses e os λ s faltosos nas seguintes expressões:

- (a) $xyz(yz)$
- (b) $\lambda x.uxy$
- (c) $\lambda u.u(\lambda x.y)$
- (d) $(\lambda u.vuu)zy$
- (e) $ux(yz)(\lambda v.vy)$
- (f) $(\lambda xyz.xz(yz))uvw$

2. Avalie as seguintes substituições:

- (a) $[(uv)/x](\lambda y.x(\lambda w.vwx))$
- (b) $[(\lambda y.xy)/x](\lambda y.x(\lambda x.x))$
- (c) $[(\lambda y.vy)/x](y(\lambda v.xv))$
- (d) $[(uv)/x](\lambda x.zy)$

3. Reduza, cada uma das expressões a seguir, a sua forma normal, se existir

- (a) $(\lambda x.xy)(\lambda u.vuu)$ 
- (b) $(\lambda xy.yx)uv$
- (c) $(\lambda x.x(x(yz))x)(\lambda u.uv)$
- (d) $(\lambda x.xxy)(\lambda y.yz)$
- (e) $(\lambda xy.xyy)(\lambda u.uyx)$
- (f) $(\lambda xyz.xz(yz))((\lambda xy.yx)u)((\lambda xy.yx)v)w$

4. Prove que $(\lambda xyz.xzy)(\lambda xy.x) \stackrel{\beta}{=} (\lambda xy.x)(\lambda x.x)$

Capítulo 4

Programação funcional em Haskell

*“There are many distinct pleasures
associated with computer programming.
Craftsmanship has its rewards, the satisfaction that
comes from building a useful object and making it work.”*
(Steven S. Skiena et Miguel A. Revilla in [44])

4.1 Introdução

Os Capítulos anteriores foram feitos com o objetivo de servirem como preparação e fundamentação para o estudo das linguagens funcionais, em particular, de Haskell. Este Capítulo e os seguintes são todos dedicados à codificação de programas funcionais utilizando esta linguagem. A comunidade das linguagens funcionais tem dado a Haskell uma atenção especial e, por este motivo, muita pesquisa tem sido feita tentando dotá-la de características que a torne uma linguagem de uso popular. Estas características foram citadas por Philip Wadler [59] e analisadas na Introdução desta Apostila.

A história de Haskell, desde a sua concepção até seu estado atual, está bem documentada em um artigo escrito por Paul Hudak, John Hughes, Simon Peyton Jones e Philip Wadler [16]. Estes autores, além de outros, participaram de todo o processo de criação e desenvolvimento da linguagem e, por este motivo, contam com detalhes toda a trajetória de Haskell, desde a sua concepção em Setembro de 1987, em Portland, no Oregon.

Haskell é uma linguagem funcional pura, não estrita, fortemente tipada, cujo nome é uma homenagem a Haskell Brooks Curry, um estudioso da lógica combinatorial e um dos mais proeminentes pesquisadores sobre λ -cálculo [6, 55]. É uma linguagem baseada em *scripts*, que consistem em um conjunto de definições associadas a nomes, em um arquivo.

A primeira edição do Haskell Report, versão 1.0, foi publicada em 1 de abril de 1990 por Paul Hudak e Philip Wadler com o objetivo de ser usada como fonte de pesquisa em projetos de linguagens de programação, ou seja, desejava-se que a linguagem estivesse sempre em evolução. No entanto, Haskell começou a se tornar popular e isto obrigou seus projetistas a repensarem o projeto inicial e resolveram nomear “Haskell 98” como uma instância de Haskell que deveria ser estável o bastante para que textos e compiladores pudessem ser construídos sem muita mudança [16]. A partir de então, Haskell98 passou a ser considerada a versão oficial a ser utilizada até a definição de *Standard Haskell*. No entanto, a linguagem continua sendo pesquisada buscando a criação de novas Bibliotecas a serem incorporadas ao sistema. Também muitas extensões estão sendo incorporadas, como Haskell concorrente, *IDLs (Interface Description Language)*, *HaskellDirect* e interfaces para *C* e *C++*, permitindo integração com estas e

outras linguagens. Em particular, tem sido desenvolvida **AspectH**, uma extensão de Haskell para suportar orientação a aspectos [3], e **HOpenGl**, uma extensão para **OpenGL**, entre outras.

O site oficial na WEB sobre Haskell é <http://www.haskell.org>, onde muitas informações podem ser encontradas, além de vários *links* para compiladores e interpretadores para ela. Para produzir código executável de máquina, foram desenvolvidos vários compiladores. Na Universidade de Glasgow, foi construído **GHC** (*Glasgow Haskell Compiler*) disponível para ambientes UNIX (Linux, Solaris, *BSD e MacOS-X) e também para Windows. **GHC** foi iniciado em Janeiro de 1989 e é, provavelmente, o compilador mais completo em atividade, um projeto *open-source* com licença liberal no estilo **BSD**. Sua primeira versão foi escrita por Kevin Hammond em **LML**, cujo protótipo ficou pronto em Junho de 1989, quando um time de pesquisadores da Universidade de Glasgow constituído por Cardelia Hall, Will Partain e Peyton Jones resolveram reimplementá-lo em Haskell (*bootstrapping*), com um *parser* construído em **Yacc** e em **C**. Esta implementação ficou pronta no outono de 1989 [16]. **GHC** está disponível em <http://www.dcs.gla.ac.uk/fp/software/ghc/>.

Na Universidade de Chalmers, Lenhart Augustsson implementou Haskell em **LML**, um compilador que ficou conhecido como **hbc**, uma referência a *Haskell Brooks Curry*, o pesquisador a quem a linguagem deve o nome. **HBC** está disponível em <http://www.cs.chalmers.se/augustss/-hbc.html>.

Também está disponível o compilador **nhc**, considerado fácil de ser instalado, com *heap profiles*, muito menor que os outros congêneres, disponível para todos os padrões UNIX e escrito em Haskell 98. O compilador **nhc** foi originalmente desenvolvido por Niklas Rögemo, um aluno de Doutorado na Universidade de Chalmers [40]. Sua motivação inicial foi construir um compilador mais eficiente em termos de espaço e memória que os sistemas **GHC** e **hbc**. Para isto ele utilizou muitas ferramentas de *heap-profiling* que haviam sido desenvolvidas em York e que revelavam muito desperdício de memória em **hbc**. Para isto, ele resolveu fazer Pós-Doutorado em York em colaboração com Colin Runciman, onde desenvolveram vários métodos de *heap-profiling* para detectar ineficiências de espaço, produzindo uma versão muito eficiente de **nhc**.

4.1.1 Outras implementações

Antes do desenvolvimento de Haskell, existia um grande projeto de pesquisa na Universidade de Yale envolvendo a linguagem **Scheme** e um dialeto seu, chamado **T**. Diversas Disertações e Teses foram feitas como resultados deste trabalho, orientadas, principalmente, por Paul Hudak. O compilador *Orbit* para **T** foi um dos resultados deste esforço. Como Paul Hudak foi fortemente envolvido no projeto de Haskell, foi natural que as técnicas aplicadas em **Scheme** fossem também utilizadas em Haskell. Neste caso, foi mais fácil compilar Haskell em **Scheme** ou **T** e usar o compilador como *back end*. Infelizmente o compilador **T** já não era mais mantido e tinha problemas de desempenho, sendo trocado por uma implementação de **Common Lisp**, buscando desempenho e portabilidade. Esta implementação ficou conhecida como Haskell de Yale, a primeira implementação de Haskell que suportava código compilado e interpretado para um mesmo programa. A implementação de Haskell de Yale foi abandonada em 1995, quando se verificou que programas compilados em **GHC** estavam sendo executados duas a três vezes mais rápidos e não existia qualquer perspectiva de mudança neste quadro.

No entanto, uma das principais expectativas no uso de Haskell se encontrava no projeto do **Dataflow** do MIT, conduzido por Arvind, cuja linguagem de programação era chamada **Id**. Em 1993, Arvind e seus colegas resolveram adotar a sintaxe de Haskell e seu sistema de tipos, com avaliação *eager* e paralela. A linguagem resultante foi chamada de **pH** (*parallel Haskell*), o tema do livro de Nikhil e Arvind sobre programação paralela implícita [32]. No entanto, ultimamente, têm surgido diversas implementações de Haskell. Entre elas, podem ser citadas:

- **Helium.** O compilador *Helium*, baseado em Utrecht, destinado especialmente ao ensino da linguagem, dando atenção especial às mensagens de erros.
- **UHC e EHC.** Utrecht também tem sido berço de outros projetos de compiladores para Haskell. **UHC** e **EHC** são alguns deles disponíveis em <http://www.cs.uu.nl/wiki/Center/-ResearchProjects>.
- **jhc.** Este é um novo compilador, desenvolvido por John Meacham, focalizado em otimização agressiva, usando análise de programas, permitindo uma técnica diferente na implementação de *type class*. Baseado no trabalho anterior de Johnsson e Boquist, **jhc** usa análise de fluxo para suportar uma representação de *thunks* que pode ser extremamente eficiente.

Apesar de todo este esforço em busca de desempenho, **GHC** e **hbc** foram implementados em uma linguagem funcional e requeriam muita memória e espaço em disco. Em Agosto de 1991, Mark Jones, um estudante de Doutorado na Universidade de Oxford, resolveu fazer uma implementação inteiramente diferente que ele chamou de **Gofer** (*Good for equational reasoning*). **Gofer** é um interpretador implementado em **C**, desenvolvido em um PC 8086 a 8 MHz, com 640KB de RAM e que cabia em único disquete de 360KB. Na realidade, ele queria aprender mais sobre a implementação de linguagens funcionais e algumas características incorporadas por ele apresentam algumas diferenças entre os sistemas de tipos de Haskell e **Gofer**, fazendo com que programas escritos em uma não funcionassem na outra implementação.

No verão de 1994, Mark Jones deixou a Universidade de Yale e foi com seu grupo para a Universidade de Nottingham, onde desenvolveram **Hg**, um acrônimo para *Haskell-gofer*, que logo adquiriu o nome de **Hugs** (*Haskell Users Gofer System*). Esta implementação, codificada em **C**, é pequena, fácil de ser usada e disponível para várias plataformas, incluindo UNIX, Linux, Windows 3.x, Win32, DOS e ambiente Macintosh. Atualmente, **Hugs** se encontra estável com a padronização de Haskell 98, apesar de muitos pesquisadores continuarem trabalhando na incorporação de suporte e novas características à linguagem, como parâmetros implícitos, dependências funcionais, **.NET** da Microsoft, uma interface funcional, módulos hierárquicos, caracteres *Unicode* e uma coleção de bibliotecas.

4.2 Primeiros passos

Existem duas formas nas quais um texto é considerado um programa em Haskell. A primeira delas é considerar todo o texto como um programa, exceto o que é comentado, que pode ser de duas maneiras: com “-”, que representa um comentário até o final da linha corrente, ou envolvendo o comentário com os símbolos “{-” e “-}”, podendo englobar várias linhas. Os arquivos em Haskell com este tipo de programa devem ter a extensão **.hs**. Esta é a maneira mais utilizada pelos programadores de Haskell.

```
{- #####
exemplo.hs
Este arquivo eh um exemplo de um arquivo .hs. Deve ser editado como arquivo
texto e salvo com a extensao .hs.
##### -}
valor :: Int                -- Uma variavel inteira
valor = 39

novalinha :: Char          -- 0 caractere de nova linha
```



```

novalinha = '\n'

resposta :: Bool                -- Uma variavel booleana
resposta = True

maior :: Bool                   -- Uma variavel booleana
maior = (valor > 71)

quadrado :: Int -> Int          -- Uma funcao de Z em Z
quadrado x = x*x

todosIguais :: Int -> Int -> Int -> Bool -- Uma funcao de ZxZxZ em Bool
todosIguais n m p = (n == m) && (m == p)
{- ##### -}

```

A outra forma é considerar todo o texto como comentário e sinalizar o que deve ser realmente um programa iniciando a linha com o sinal “>” indentado. Neste caso, o arquivo deve ter extensão **.lhs**. Vamos mostrar um exemplo de cada situação. Vamos mostrar o mesmo exemplo usando a visão de literal, com a extensão **.lhs**. Deve ser observado que os sinais de início e fim de comentários desaparecem.

```

#####
exemplo.lhs. Este arquivo eh um exemplo de um arquivo .lhs.
#####
> valor :: Int                -- Uma constante inteira
> valor = 39

> novalinha :: Char          -- 0 caractere de nova linha
> novalinha = '\n'

> resposta :: Bool           -- Uma variavel booleana
> resposta = True

> maior :: Bool              -- Uma variavel booleana
> maior = (valor > 71)

> quadrado :: Int -> Int      -- Uma funcao de Z em Z
> quadrado x = x*x

> todosIguais :: Int -> Int -> Int -> Bool -- Uma funcao de ZxZxZ em Bool
> todosIguais n m p = (n == m) && (m == p)
#####

```

A indentação em Haskell é importante. Ele usa um esquema chamado *layout* para estruturar seu código [41], a exemplo de **Python**. Este esquema permite que o código seja escrito sem a necessidade de sinais de ponto e vírgula explícitos, nem de chaves.

Neste caso, a indentação do texto tem um significado preciso e é aplicada segundo uma regra conhecida como *regra do offside*, em alusão a uma regra de mesmo nome, utilizada no futebol. Esta regra é descrita da seguinte forma:

- se uma linha começa em pelo menos uma coluna à frente (mais à direita) do início da linha anterior, é considerada a continuação da linha precedente,
- se uma linha começa na mesma coluna que a do início da linha anterior, elas são consideradas definições independentes entre si e
- se uma linha começa em pelo menos uma coluna anterior (mais à esquerda) do início da linha anterior, ela não pertence à mesma lista de definições.

Uma definição em Haskell termina de forma implícita no final da linha, se ela não tiver continuidade na linha seguinte, continuando em pelo menos uma coluna após a coluna de início da definição. No entanto, Haskell adota também um mecanismo explícito para indicar o final de uma definição, que é a utilização do ponto e vírgula, “;”. Desta forma, pode-se usar o “;” se a intenção for escrever mais de uma definição em uma mesma linha. Por exemplo,

```
> resposta = 42; resultado = 234
```

4.2.1 O interpretador ghci

O interpretador **ghci** disponibiliza a biblioteca de funções predefinidas que compõem o arquivo “Prelude.hs”, e podem ser utilizadas pelo usuário a partir do momento em que o interpretador é carregado. Na chamada, é aberta uma seção, que permanece ativa enquanto o sistema estiver em execução.

Os comandos de **ghci** são muito simples e não oferecem muitas possibilidades ao usuário. Eles podem ser vistos pela chamada ao *help*, através do comando `:?`. Alguns deles podem ser observados na Tabela 4.2.

Tabela 4.1: Principais comandos de ghci.

Comando	Ação realizada
<code>:?</code>	Aciona o help
<code>:e</code>	Chama o script atual
<code>:e exemplo.hs</code>	Edita o arquivo exemplo.hs
<code>:l exemplo.hs</code>	Carrega o script exemplo.hs e limpa outros arquivos carregados
<code>:a exemplo.hs</code>	Carrega o script exemplo.hs sem limpar os outros arquivos
<code>:q</code>	Termina a seção

Para o usuário executar qualquer função do **Prelude** é necessário apenas chamar esta função na linha de comandos (*prompt*), disponível após o interpretador ser carregado, seguida de seus argumentos e apertar a tecla “Enter”. O resultado desta execução é exibido imediatamente na linha seguinte ao “*prompt*”. Assim, o interpretador também pode ser usado como uma calculadora para avaliar expressões aritméticas, booleanas, logarítmicas, trigonométricas, etc. A forma de utilização é a mesma já citada. Algumas destas funções ou operadores aritméticos estão mostrados na Tabela 4.2, que também mostra as suas ordens de precedência.

Os operadores podem ser *infixos* ou *pré-fixos*. Normalmente os operadores aritméticos são declarados como *infixos*, por exemplo, se usa a expressão $3 + 7$, por ser esta a forma comumente utilizada na Matemática. Já as funções são normalmente declaradas como pré-fixas por ser a forma mais utilizada nas demais linguagens de programação. No entanto, os operadores *infixos* podem ser utilizados como pré-fixos, apenas colocando o operador entre parênteses. Por exemplo,

Tabela 4.2: Principais operadores matemáticos de ghci.

Preced.	Assoc. à esquerda	Não associativa	Assoc. à direita
9	!, !!, //, > . >	>>=	.
8			**, ^, ^^
7			%, /, 'div', 'mod', 'rem', 'quot'
6	+, -		
5		\\	:, ++, > + >
4		/=, <, <=, ==, >, >=, 'elem', 'notElem'	
3			&&
2			
1			
0			\$

a expressão $2 + 3$ onde o operador $+$ é *infixo* pode ser utilizada como $(+) 2 3$ onde o operador $+$ se torna *pré-fixo*. De forma similar, os operadores *pré-fixos* também podem ser aplicados como *infixos*, apenas colocando o operador entre aspas simples (o acento agudo em ambos os lados do operador). Por exemplo, *maxi* $3 4$ (*pré-fixo*) pode ser utilizado como $3 \text{ 'maxi' } 4$ (*infixo*).

É possível também trocar a associatividade ou a prioridade de um operador. Para isso, é necessário declarar, explicitamente, o tipo de associatividade e a prioridade da função. Por exemplo, para trocar a associatividade e a prioridade da função **toma**, pode-se fazer a declaração:

Infixl 7 toma

significando que a função **toma** passa a ser *infixa*, associando-se pela esquerda e tem um nível de prioridade 7. Se a prioridade for omitida, será considerada igual a 9, por *default*. Seguem alguns exemplos de chamadas à calculadora de expressões ou de funções.

Exemplos:

```
?: 2 + 3 <enter>
5
```

```
?: (1 * 6) == (3 'div' 5) <enter>
False
```

4.2.2 Identificadores em Haskell

Os identificadores em Haskell são sensíveis a caracteres, ou seja, as letras maiúsculas são distintas das letras minúsculas. Os identificadores, devem ser iniciados, sempre, por uma letra maiúscula, se for um tipo, ou minúscula, se for um outro identificador como uma variável, uma constante ou uma função. A esta primeira letra do identificador podem ser seguidos outros caracteres, que podem ser letras maiúsculas ou minúsculas, dígitos, sublinhado ou acentos agudos. Por exemplo,

```
type Par = (Int, Int)
somaAmbos :: Par -> Int
somaAmbos (primeiro, segundo) = primeiro + segundo
```

Deve ser lembrado aqui que Haskell é uma linguagem funcional pura e, como tal, não permite

atribuições destrutivas, ou seja, não é possível fazer atualizações de variáveis em Haskell. Nos exemplos mostrados anteriormente, a variável **resposta** terá o valor **True** enquanto o *script* estiver ativo. Se for necessário armazenar um outro valor, terá que ser criada uma outra variável para isto. Desta forma, em Haskell, as variáveis são consideradas *constantes*, uma vez que elas não podem ser atualizadas.

As palavras reservadas da linguagem são sempre escritas em letras minúsculas. Haskell apresenta 22 palavras reservadas, mostradas a seguir:

case	else	infix	module	type
class	hiding	infixl	of	where
data	if	infixr	renaming	
default	import	instance	then	
deriving	in	let	to	

4.3 Funções em Haskell

As formas de definição de funções em Haskell têm a ver com as formas de definição de funções utilizadas na Matemática, mostradas no Capítulo 1. Elas podem ser representadas graficamente por uma caixa que recebe um ou mais parâmetros como entrada (argumentos), processa-os e constrói um resultado único que é exibido como saída, conforme pode ser visto na Figura 4.1.

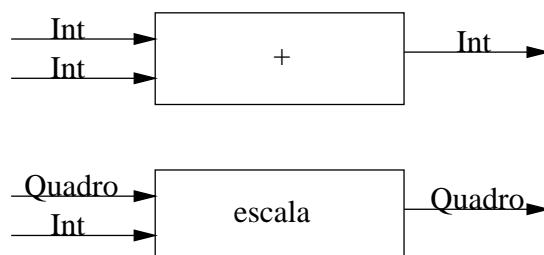


Figura 4.1: Representação gráfica das funções + e escala.

Exemplos de funções:

- Uma função que calcula as raízes de uma equação bi-quadrada.
- Uma função que emite o relatório final com as notas parciais e final dos alunos da disciplina Tópicos em Linguagem de Programação.
- Uma função que controla a velocidade de um automóvel.

Mais exemplos de funções podem ser encontrados em qualquer atividade da vida. Na Figura 4.2 estão mostradas, graficamente, algumas funções baseadas no livro de Simon Thompson [55]. Na figura, o desenho do lado esquerdo ou do lado direito é um “*Cavalo*”. Um *Cavalo* é o elemento de entrada da função que o processa transformando-o em outro *Cavalo*. Por exemplo a função **espelhaV** toma como argumento um *Cavalo* e faz o espelhamento deste *Cavalo* em relação ao eixo vertical do plano **xy**. Deve ser prestada atenção ao fato de que as funções **espelhaV**, **espelhaH**, **invertecor** e **escala** tomam apenas um *cavalo* como seus argumento de entrada, enquanto a função **sobrepoe** toma dois *cavalos* para poder fazer a sobreposição de ambos.

Neste ponto não será mostrada uma forma como cada *Cavalo* pode ser implementado, para ser simulado e processado por um programa. Isto ocorre porque o objetivo aqui é apenas mostrar exemplos de funções. Uma modelagem destes objetos será mostrada no próximo Capítulo.

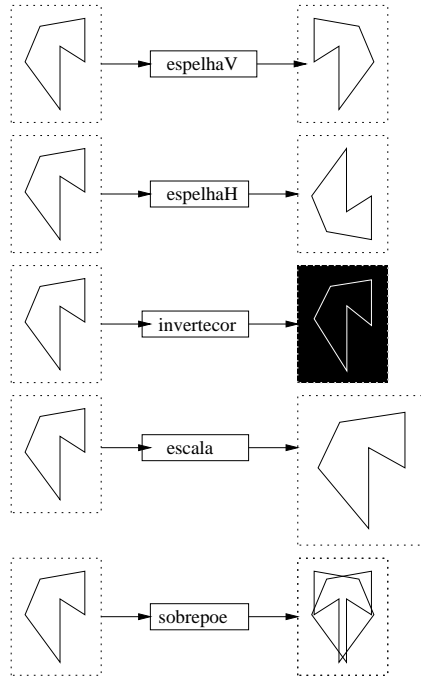


Figura 4.2: Resultados gráficos de funções.

É necessário salientar a importância que os tipos dos argumentos e dos resultados têm nas definições de funções. Eles permitem ao programador estabelecer uma correspondência bem definida entre eles e os objetos que modelam, proporcionando uma simulação adequada. Assim, as funções da Figura 4.2 têm os tipos:

```
espelhaV :: Cavalo -> Cavalo
espelhaH :: Cavalo -> Cavalo
invertecor :: Cavalo -> Cavalo
escala :: Cavalo -> Cavalo
sobrepoe :: Cavalo -> Cavalo -> Cavalo
```

4.3.1 Construindo funções

Um tipo de dado é uma coleção de valores onde todos eles têm as mesmas características. Por exemplo, os números inteiros, os caracteres, os strings de caracteres, etc. Os tipos das funções, em Haskell, são declaradas por um nome identificador, seguido de ::, vindo em seguida os tipos de seus argumentos, um a um, com uma flecha (\rightarrow) entre eles e, finalmente mais uma flecha seguida do tipo do resultado que a aplicação da função produz. Por exemplo, as funções `+` e `todosIguais`, mostradas graficamente na Figura 4.1, têm seus tipos definidos da forma a seguir:

```
+ :: Int -> Int -> Int e
todosIguais :: Int -> Int -> Int -> Bool
```

O tipo da função é declarado em sua forma “*currificada*”, onde uma função de n argumentos é considerada como n funções de um único argumento, da mesma forma adotada pelo λ -cálculo, vista no Capítulo 2.

Os tipos nos dão informações importantes sobre a aplicação das funções. Por exemplo, a declaração da função `todosIguais`, mostrada anteriormente, nos informa que:

- a função tem três argumentos de entrada, todos do tipo inteiro (**Int**) e
- o resultado da aplicação da função é um valor do tipo **Bool**, facilitando o entendimento do problema e a forma de solução adotada para resolvê-lo.

Além dos tipos, a declaração de uma função exhibe explicitamente como o processamento de seus argumentos deve ser feito. Por exemplo, verifiquemos a definição da função **somaAmbos**, a seguir. A forma como ela processa seus argumentos (entradas), x e y , é somando-os, $x + y$.

```
somaAmbos :: Int -> Int -> Int
somaAmbos x y = x + y
```

Deve ser verificado aqui uma diferença entre a formulação matemática desta função e sua definição, em Haskell. Em Haskell, a definição é feita por justaposição dos argumentos, x e y , ao nome da função. Na Matemática, os argumentos são colocados entre parênteses. Outra forma de definição de funções, em Haskell, declará-las em função de outras funções, já definidas. Por exemplo, a função **mmc** (mínimo múltiplo comum) entre dois valores inteiros, visto no Capítulo 1, pode ser definida em função das funções **mdc** (máximo divisor comum) e **div** (divisão inteira) entre eles. Vejamos as definições, em Haskell:

```
mmc :: Int -> Int -> Int
mmc x y = div x*y (mdc x y)
```

```
mdc :: Int -> Int -> Int
mdc x y
  | x == y = x
  | x > y = mdc x-y y
  | otherwise = mdc y x
```

```
div :: Int -> Int -> Int
div x y
  | x == y = 1
  | x > y = 1 + div x-y y
  | otherwise = 0
```

Nas definições mostradas, aparecem alguns elementos novos. Por exemplo, na definição da função **div**, deve ser verificada a semelhança entre a definição matemática e a definição em Haskell. Em Haskell, as guardas das definições são colocadas em primeiro lugar, enquanto, na Matemática, elas aparecem depois da definição. Outro elemento novo é a palavra reservada **otherwise** que tem o significado de todas as guardas não referidas anteriormente. Apesar destas diferenças, pode-se verificar que a tradução das definições matemáticas para definições em Haskell é um processo direto, inclusive no que se refere ao uso de definição recursiva. Deve ser observado que a ordem em que as definições são feitas não tem importância, desde que estejam no mesmo *script*.

Algumas linguagens funcionais exigem que os tipos das funções sejam declarados explicitamente pelo programador. Em Haskell, seus projetistas optaram por permitir que os tipos das funções sejam inferidos pelo sistema. Isto significa que é opcional a declaração dos tipos das funções pelo programador. Mesmo assim, encoraja-se que esta declaração seja feita explicitamente, como uma disciplina de programação. Isto permite ao programador um completo entendimento do problema e da solução adotada.

4.3.2 Avaliação de funções em Haskell

A forma de avaliação de funções utilizada em programas codificados em Haskell é a ordem normal de avaliação, preconizada pelo segundo teorema de Church-Rösser, vista no Capítulo 2. Isto significa que Haskell obedece ao sistema *leftmost-outermost*, usando um mecanismo de avaliação *lazy* (preguiçoso) que só avalia uma expressão se ela for realmente necessária e no máximo uma vez. Isto significa um tipo de avaliação semelhante à avaliação *curto circuito* utilizada em algumas linguagens convencionais de programação. Vejamos um exemplo de avaliação usando as funções definidas nos *scripts* mostrados no início deste Capítulo.

```
todosIguais (quadrado 3) valor (quadrado 2)
= ((quadrado 3) == valor) && (valor == (quadrado 2))
= ((3 * 3) == valor) && (valor == (quadrado 2))
= (9 == valor) && (valor == (quadrado 2))
= (9 == 39) && (39 == (quadrado 2))
= False && (39 == (quadrado 2))
= False (utilizando o mecanismo de avaliação lazy)
```

4.3.3 Casamento de padrões (patterns matching)

O casamento de padrões é outra forma de codificação de funções em Haskell, baseada na definição por enumeração utilizada na Matemática, vista no Capítulo 1. Neste tipo de definição, são exibidos todos os valores que os argumentos da função podem ter e, para cada um deles, declara-se o valor do resultado correspondente. De forma resumida, exibem-se todos pares do mapeamento (entrada, resultado). Por exemplo, vejamos as declarações das funções **eZero** e **fat**, a seguir:

```
eZero :: Int -> Bool           fat :: Int -> Int
eZero 0 = True                 e   fat 0 = 1
eZero _ = False               fat n = n * fat (n - 1)
```

Na execução de aplicações destas funções, os padrões são testados sequencialmente, de cima para baixo. O primeiro padrão que casa com o valor da entrada terá o valor correspondente como resultado da aplicação da função. Se não ocorrer qualquer casamento entre o valor de entrada e um padrão de entrada, o resultado da aplicação da função será um erro. Ao se aplicar a função **eZero** a um argumento n , primeiro é verificado se este número n casa com 0. Se for verdade, o resultado será **True**. Se este padrão não for verificado, ou seja, se n for diferente de 0, verifica-se o casamento com o segundo padrão e assim por diante. Neste caso, o resultado será **False**. O mesmo raciocínio vale para a função **fat**.

Esta forma de análise sequencial deve sempre ser levada em consideração para que erros grosseiros sejam evitados. Como exemplo, se, na declaração da função **eZero**, a definição da função para o caso de n ser 0 for trocada pela segunda definição, o resultado da aplicação da função será sempre igual a **False**, mesmo que o argumento seja 0, uma vez que o `_` (*underscore*) significa “qualquer caso”.

Exercícios:

1. Dê a definição da função **todosQuatroIguais** do tipo `Int -> Int -> Int -> Int -> Bool` que dá o resultado **True** se seus quatro argumentos forem iguais.
2. Dê a definição da função **todosQuatroIguais** usando a definição da função **todosIguais**, vista na subseção anterior.

3. O que está errado com a definição da função **todosDiferentes** abaixo?
`todosDiferentes n m p = ((n /= m) && (m /= p))`
4. Projete um teste adequado para a função **todosIguais**, considerando a função
`teste :: Int -> Int -> Int -> Bool`
`teste n m p = ((n + m + p) == 3 * p)`
 Esta função se comporta da mesma forma que a função **todosIguais** para o seu teste de dados? Que conclusão você tira sobre os testes em geral?
5. Dê uma definição para a função **quantosIguais** aplicada a três entradas inteiras e que retorna quantas delas são iguais.
6. Escreva a sequência de cálculos para as seguintes expressões:
`maximo ((2 + 3) - 7)(4 + (1 - 3))`
`todosIguais 4 quadrado 2 3`
`quantosIguais 3 4 3`

4.4 Tipos de dados em Haskell

Haskell, a exemplo de qualquer linguagem de programação, provê uma coleção de tipos primitivos e também permite tipos estruturados, definidos pelo programador, provendo grande flexibilidade na modelagem de programas.

Os tipos primitivos da linguagem

Os tipos primitivos de Haskell são: o tipo inteiro (**Int** ou **Integer**), o tipo booleano (**Bool**), o tipo caractere (**Char**), o tipo cadeia de caracteres (**String**) e o tipo ponto flutuante (**Float** ou **Double**) e o tipo lista.

Haskell também apresenta tipos estruturados como, por exemplo, o produto cartesiano. Ainda neste Capítulo, eles serão estudados. Além destes tipos, Haskell também implementa tipos algébricos e tipos abstratos de dados, mas eles serão tratados em capítulos separados, dada a grande importância que exercem na implementação de funções nesta linguagem.

Nesta seção, vamos analisar cada um dos tipos primitivos, deixando o tipo lista para ser tratado no Capítulo 4, dada a sua importância nas linguagens funcionais e, em particular, em Haskell.

4.4.1 O tipo inteiro (Int ou Integer)

Como em muitas linguagens, o tipo inteiro é primitivo em Haskell. Seu domínio de valores é o mesmo das outras linguagens. Os valores do tipo **Integer** são representados com o dobro da quantidade de *bits* necessários para representar os valores do tipo **Int**. Seus operadores aritméticos e relacionais são os mesmos admitidos na maioria das outras linguagens e estão mostrados na Tabela 4.3.

Exemplo. Vamos construir um programa que envolva operações com inteiros. Imaginemos uma empresa de vendas que necessita de respostas para as questões a seguir, para fundamentar suas decisões:

- Questão 1: Qual o total de vendas desde a semana 0 até a semana n?

Tabela 4.3: Operadores aritméticos e relacionais dos tipos `Int` e `Integer`.

Operador	Tipo	Descrição
<code>+</code> , <code>*</code>	<code>Int -> Int -> Int</code>	adição e multiplicação
<code>^</code>	<code>Int -> Int -> Int</code>	exponenciação
<code>-</code>	<code>Int -> Int -> Int</code>	subtração (infixa) e inversor de sinal (prefixa)
<code>div</code>	<code>Int -> Int -> Int</code>	divisão inteira (prefixa), ou <code>'div'</code> (infixa)
<code>mod</code>	<code>Int -> Int -> Int</code>	módulo (prefixa), ou <code>'mod'</code> (infixa)
<code>abs</code>	<code>Int -> Int</code>	valor absoluto de um inteiro
<code>negate</code>	<code>Int -> Int</code>	troca o sinal de um inteiro
<code>></code> , <code>>=</code> , <code>==</code>	<code>Int -> Int -> Bool</code>	operadores relacionais
<code>==</code> , <code><=</code> , <code><</code>	<code>Int -> Int -> Bool</code>	operadores relacionais

- Questão 2: Qual a maior venda semanal entre as semanas 0 e `n`?
- Questão 3: Em que semana ocorreu a maior venda?
- Questão 4: Existe alguma semana na qual nada foi vendido?
- Questão 5: Em qual semana não houve vendas? (se houve alguma).

Vamos construir algumas funções em Haskell para responder a algumas destas questões, deixando outras como exercício para o leitor. Para isto é necessário que recordemos as definições matemáticas de funções, vistas no Capítulo 1.

Questão 1:

Para solucionar a **Questão 1**, devemos inicialmente construir uma função **venda n** que vai nos informar qual o valor das vendas na semana `n`. Esta função pode ser construída de várias formas, no entanto será feita aqui uma definição por casamento de padrões, semelhante à definição por enumeração da Matemática.

```
venda :: Int -> Int
venda 0 = 7
venda 1 = 2
venda 2 = 5
```

Verifiquemos agora como as definições recursivas utilizadas na Matemática podem ser utilizadas em Haskell. Recordemos a definição matemática de uma função usando recursão. Por exemplo, na definição de uma função **fun**, devemos fazer duas coisas:

- explicitar o valor de **fun 0**, (caso base)
- explicitar o valor de **fun n**, usando o valor de **fun (n - 1)** (caso recursivo ou passo indutivo).

Trazendo esta forma de definição para o nosso caso, vamos construir a função **totaldeVendas n** a partir de **venda n**, que vai mostrar o total de vendas realizadas até a semana `n`, inclusive.

```
totaldeVendas :: Int -> Int
totaldeVendas n
  | n == 0 = venda 0
  | otherwise = totaldeVendas (n-1) + venda n
```

Neste caso, em vez de usarmos padrões, serão usadas equações condicionais (booleanas), também chamadas de “guardas”, cujos resultados são valores **True** ou **False**. As guardas são avaliadas também sequencialmente, da mesma forma feita com o casamento de padrões. A palavra reservada **otherwise** exerce um papel parecido, mas diferente, do _ (sublinhado), já descrito anteriormente. A cláusula **otherwise** deve ser utilizada apenas quando houver guardas anteriores, cujos resultados sejam todos False. Isto significa que se a cláusula **otherwise** for colocada na primeira definição de uma função, ocorrerá um erro, uma vez que não existem guardas definidas anteriormente. No caso do _ (sublinhado), esta situação não ocasionará erro.

Usando a definição de **venda n** e de **totaldeVendas n**, dadas anteriormente, podemos calcular a aplicação da função **totaldeVendas 2**, da seguinte forma:

```
totaldeVendas 2 = totaldeVendas 1 + venda 2
                = (totaldeVendas 0 + venda 1) + venda 2
                = (venda 0 + venda 1) + venda 2
                = (7 + venda 1) + venda 2
                = (7 + 2) + venda 2
                = 9 + venda 2
                = 9 + 5
                = 14
```

E se chamarmos **totaldeVendas (-2)**? O resultado será **=) ERROR: Control stack overflow**, uma vez que a pilha do sistema vai estourar. Para esta entrada deve ser criada uma exceção e, como tal, deve ser tratada. Em Haskell, existem várias formas de tratar exceções, mas elas serão vistas no Capítulo 6.

Apenas para exemplificar, uma forma simples de fazer isto é definir um valor fictício para o caso de **n** ser negativo, transformando a definição dada na seguinte:

```
totaldeVendas n
  | n == 0 = venda 0
  | n > 0 = totaldeVendas (n - 1) + venda n
  | otherwise = 0
```

Questão 2:

Vamos agora definir a função **maiorVenda**, onde **maiorVenda n** será igual a **venda n**, se a venda máxima ocorreu na semana **n**. Se a maior venda ocorreu na semana **0**, a função **maiorVenda 0** será **venda 0**. Se a maior venda ocorreu até a semana **n**, pode estar até a semana **(n-1)** ou será **venda n**.

```
maiorVenda :: Int -> Int
maiorVenda n
  | n == 0 = venda 0
  | maiorVenda (n-1) >= venda n = maiorVenda (n - 1)
  | otherwise = venda n
```

Esta mesma função pode ser construída de outra forma, usando uma função auxiliar, **maximo**, que aplicada a dois valores inteiros retorna o maior entre eles. Esta forma de definição de funções é incentivada em Haskell uma vez que significa a definição de uma função baseando-se em outras, já definidas. Isto significa modularidade e a possibilidade de reutilizabilidade do software que é uma característica importante, principalmente na implementação de grandes sistemas, conhecidos como “*programming in the large*”, uma vez que muito trabalho pode ser economizado.

Em pequenos sistemas, conhecidos como “*programming in the small*”, esta diferença não é sentida, no entanto, a modularidade deve ser uma disciplina de programação a ser seguida.

```
maximo :: Int -> Int -> Int
maximo x y
  | x >= y = x
  | otherwise = y

maiorVenda n
  | n == 0 = venda 0
  | otherwise = maximo (maiorVenda (n - 1)) venda n
```

De forma resumida, até agora vimos três formas de definir funções:

1. **quadrado** $x = x * x$
2. **maximo** $n\ m$
 - | $n \geq m = n$ (equação condicional)
 - | **otherwise** $= m$
3. usar o valor da função sobre um valor menor ($n - 1$) e definir para n .

Estas formas de definição de funções são chamadas atualmente pelo termo genérico de **recursão primitiva**. Elas foram assim chamadas, a primeira vez, por Gödel quando ele usou estes termos para mostrar que para qualquer sistema lógico que satisfaça certas condições razoáveis, existem proposições verdadeiras sobre a aritmética que não são prováveis no sistema[9].

Exercícios:

1. Defina uma função para encontrar a semana em que ocorreu a venda máxima entre a semana 0 e a semana n . O que sua função faz se houver mais de uma semana com vendas máximas?
2. Defina uma função para encontrar uma semana sem vendas entre as semanas 0 e n . Se não existir tal semana, o resultado deve ser $n + 1$.
3. Defina uma função que retorne o número de semanas sem vendas (se houver alguma).
4. Defina uma função que retorna o número de semanas nas quais foram vendidas s unidades, para um inteiro $s \geq 0$. Como você usaria esta solução para resolver o problema 3?
5. Teste as funções que usam vendas com a definição $vendas\ n = n \text{ 'mod' } 2 + (n + 1) \text{ 'mod' } 3$

Os tipos **Int**, **Integer**, **Float** e **Double** são os tipos numéricos mais conhecidos em Haskell. No entanto, outros tipos numéricos também existem em Haskell. A Tabela 4.4 apresenta um resumo dos principais tipos numéricos implementados em **GHC**, onde se verifica que são bem maiores, em termos de quantidade, que os oferecidos pela grande maioria das outras linguagens de programação.

Tabela 4.4: Principais tipos numéricos em Haskell.

Tipo	Descrição
<i>Double</i>	Ponto flutuante de precisão dupla
<i>Float</i>	Ponto flutuante de precisão simples
<i>Int</i>	Inteiro sinalizado de precisão fixa ($-2^{29} .. 2^{29} - 1$)
<i>Int8</i>	Inteiro sinalizado de 8 bits
<i>Int16</i>	Inteiro sinalizado de 16 bits
<i>Int32</i>	Inteiro sinalizado de 32 bits
<i>Int64</i>	Inteiro sinalizado de 64 bits
<i>Integer</i>	Inteiro sinalizado de precisão arbitrária
<i>Rational</i>	Números racionais de precisão arbitrária
<i>Word</i>	Inteiro não sinalizado de precisão fixa
<i>Word8</i>	Inteiro não sinalizado de 8 bits
<i>Word16</i>	Inteiro não sinalizado de 16 bits
<i>Word32</i>	Inteiro não sinalizado de 32 bits
<i>Word64</i>	Inteiro não sinalizado de 64 bits

4.4.2 O tipo ponto flutuante (Float ou Double)

Os valores do tipo ponto flutuante (números reais) pertencem aos tipos **Float** ou **Double**, da mesma forma que um número inteiro pertence aos tipos **Int** ou **Integer**. Isto significa que as únicas diferenças entre valores destes tipos se verificam na quantidade de *bits* usados para representá-los. A Tabela 4.5 mostra as principais funções aritméticas predefinidas na linguagem. As funções aritméticas recebem a denominação especial de operadores.

Apesar de alguns pesquisadores, mais puristas, condenarem a sobrecarga de operadores, alguns outros defendem que alguma forma de sobrecarga deve existir, para facilitar a codificação de programas. Os projetistas de Haskell admitiram a sobrecarga de operadores em sua implementação, por ela ser importante na implementação de uma de suas características que permite um grau de abstração bem maior que normalmente se encontra em outras linguagens. Esta característica se refere às classes de tipos (*type class*), um tema a ser analisado no Capítulo 5.

Em relação às operações mistas, por exemplo $123.5 + 12$, em Haskell haverá uma **coerção** que é uma conversão implícita de alargamento, onde o valor inteiro 12 será convertido implicitamente para 12.0. Sendo assim, a operação $+$ é sobrecarregada em Haskell. Neste caso, o valor inteiro 12 foi promovido para a classe **Num** que suporta o operador $+$ com um tipo **Float**.

Alguns cálculos sobre números de ponto flutuantes não apresentam resultados numéricos. Uma forma disto ser sinalizado em Haskell é o retorno de uma mensagem informando que o resultado não é um número, **NaN** ou um valor infinito **Infinity**. Isto indica que alguma coisa aconteceu de errado e este resultado não pode ser usado em cálculos futuros uma vez que o valor não é um número e sendo assim, qualquer cálculo efetuado com ele resultará em um mesmo resultado.

Outra observação interessante, diz respeito ao fato de que apesar de serem sobrecarregados, não existe uma conversão automática de um valor do tipo **Integer** para o tipo **Float**. Se desejarmos adicionar um valor inteiro como **floor 7.1** ao valor **8.2** será mostrada uma mensagem de erro do tipo **(floor 7.1) + 8.2** uma vez que estamos tentando somar quantidades de tipos distintos.

Como pode ser verificado na Tabela 4.5, existem em Haskell as funções **floor**, **ceiling** e **round** que são utilizadas para fins de arredondamento de números reais. A função **floor** trunca a parte fracionária e retorna a parte inteira de um número real, enquanto a função **ceiling** arredonda para o sucessor ao respectivo número real e a função **round** arredonda o número real para o número inteiro que estiver mais próximo, com um corte em 0.5.

Tabela 4.5: Operadores aritméticos de ponto flutuante em Haskell.

Operador	Tipo	Descrição
<code>+, -, *</code>	<code>Float -> Float -> Float</code>	operadores aritméticos
<code>/</code>	<code>Float -> Float -> Float</code>	divisão
<code>^</code>	<code>Int -> Int -> Int</code>	exponenciação
<code>^^</code>	<code>Float -> Int -> Float</code>	exponenciação
<code>**</code>	<code>Float -> Float -> Float</code>	exponenciação
<code>==, / =, <, >, <=, >=</code>	<code>Float -> Float -> Bool</code>	operadores lógicos
<code>abs</code>	<code>Float -> Float</code>	valor absoluto
<code>acos, asin, atan</code>	<code>Float -> Float</code>	funções trigonométricas
<code>ceiling, floor, round</code>	<code>Float -> Int</code>	arredondamentos
<code>cos, sin, tan</code>	<code>Float -> Float</code>	funções trigonométricas
<code>exp</code>	<code>Float -> Float</code>	exponencial
<code>fromInt</code>	<code>Int -> Float</code>	conversão
<code>log</code>	<code>Float -> Float</code>	logaritmo base 10
<code>logBase</code>	<code>Float -> Float -> Float</code>	logaritmo base n
<code>negate</code>	<code>Float -> Float</code>	inverte sinal
<code>read</code>	<code>String -> Float</code>	conversão
<code>pi</code>	<code>Float</code>	valor de pi
<code>show</code>	<code>* -> String</code>	transforma em String
<code>signum</code>	<code>Float -> Int</code>	transforma em Int
<code>sqrt</code>	<code>Float -> Float</code>	raiz quadrada real

4.4.3 O tipo booleano (Bool)

O tipo booleano já faz parte da grande maioria das linguagens de programação, sejam elas de qualquer paradigma. É sempre necessário, em qualquer programa de computador, saber o resultado da avaliação de uma expressão lógica, se ele é verdadeiro ou falso. Algumas linguagens, como C, não tem o tipo booleano e, por este motivo, o resultado da avaliação de uma expressão tem valor numérico, cuja interpretação corresponde ao valor verdadeiro, **True**, se o resultado numérico for diferente de zero e será falso, **False**, se a avaliação da expressão for igual a zero. Para eliminar esta dificuldade existente na linguagem C, os projetistas de C++, da qual a linguagem C pode ser considerada um subconjunto, incluíram o tipo **Bool** como um de seus tipos primitivos.

Há de se convir que esta metodologia utilizada em C não atende ao princípio da legibilidade que deve ser seguido pelas linguagens de programação. Haskell, no entanto, o obedece e adota o tipo booleano **Bool** para satisfazer este princípio.

Os únicos valores booleanos são **True** e **False** e sobre eles podem ser utilizadas funções predefinidas ou funções construídas pelo usuário. As funções predefinidas estão mostradas na Tabela 4.6.

A função **OU** exclusivo pode ser definida pelo usuário da seguinte forma:

```
exOr :: Bool -> Bool -> Bool
exOr True x = not x
exOr False x = x
```

Tabela 4.6: Funções booleanas predefinidas em Haskell.

Função	Nome	Tipo
<code>&&</code>	and	<code>&& :: Bool -> Bool -> Bool</code>
<code> </code>	or	<code> :: Bool -> Bool -> Bool</code>
<code>not</code>	inversor	<code>not :: Bool -> Bool</code>

Exercícios:

1. Dê a definição de uma função `nAnd :: Bool -> Bool -> Bool` que dá o resultado *True*, exceto quando seus dois argumentos são ambos *True*.
2. Defina uma função `numEqualMax :: Int -> Int -> Int -> Int` onde **numEqualMax** `n m p` retorna a quantidade de números iguais ao máximo entre *n*, *m* e *p*.
3. Como você simplificaria a função

```
funny x y z
  | x > z = True
  | y >= x = False
  | otherwise = True
```

4.4.4 O tipo caractere (Char)

Os caracteres em Haskell são literais escritos entre aspas simples. Existem alguns caracteres que são especiais por terem utilizações específicas. Entre eles se encontram:

<code>'\t'</code> - tabulação	<code>'\"'</code> - aspas simples
<code>'\n'</code> - nova linha	<code>'\"'</code> - aspas duplas
<code>'\\'</code> - uma barra invertida	<code>'\n'</code> - caractere ASCII de valor inteiro <i>n</i>

Existem algumas funções predefinidas em Haskell feitas para converter caracteres em números e vice-versa. Seus tipos são mostrados a seguir, no entanto, o leitor deve consultar as observações feitas sobre elas por Simon Thompson [55] na classe **Enum a** no Capítulo 6, que as contém.

```
toEnum :: Enum a => Int -> a
fromEnum :: Enum a => a -> Int
```

Exercício:

Defina uma função `mediadasVendas :: Int -> Float` que dê como resultado a média aritmética entre os valores de *vendas* 0 até *vendas n*.

4.4.5 O tipo cadeia de caracteres (String)

O tipo cadeia de caracteres, normalmente chamado de **String**, tem uma característica peculiar em Haskell. Apesar deste caso ser considerado um caso patológico por alguns pesquisadores de linguagens de programação, os criadores de Haskell admitiram duas formas para este tipo. Ele é um tipo predefinido como **String**, mas também pode ser considerado como uma lista de caracteres. Para satisfazer as duas formas, as *strings* podem ser escritas entre aspas duplas ou usando a notação de lista de caracteres (entre aspas simples). Por exemplo, a *string* “**Constantino**” tem o mesmo significado que a *string* [‘C’, ‘o’, ‘n’, ‘s’, ‘t’, ‘a’, ‘n’, ‘t’, ‘i’, ‘n’, ‘o’], em Haskell.

Todas as funções sobre listas do **Prelude.hs** podem ser aplicadas a valores do tipo **String**, uma vez que elas são definidas sobre listas de algum tipo e uma *string* é também uma lista.

Toda aplicação de função, em Haskell, produz um resultado. Podemos verificar isto através da tipificação das funções. Ocorre, no entanto, que para mostrar um resultado, no monitor ou em outro dispositivo de saída, é necessário definir uma função para esta tarefa. E qual deve ser o resultado desta operação? Mostrar um valor no monitor não implica em retorno de qualquer valor como resultado. Algumas linguagens de programação funcional, como ML, resolvem este problema de comunicação com o mundo exterior através de atribuições destrutivas, o que descaracteriza a pureza da linguagem funcional, transformando-a em impura.

No entanto, Haskell foi projetada como uma linguagem funcional pura e, para resolver este problema de comunicação com o mundo exterior, seus projetistas adotaram uma semântica conhecida como *semântica de ações* que, por sua vez, é baseada em Mônadas, um tema da Teoria das Categorias, uma teoria matemática bastante difundida no campo da Computação [46, 57]. Este tema, devido a sua complexidade, não será aqui tratado. No entanto serão feitas algumas explicações sobre ele no Capítulo 7 deste livro.

Uma ação em Haskell é um tipo de função que retorna um valor do tipo **IO ()**, para ser coerente com o projeto da linguagem. Mostraremos aqui algumas funções usadas em Haskell na comunicação com o mundo exterior para facilitar o entendimento pelo leitor. Seria bastante tedioso usar funções sem poder verificar os resultados de suas aplicações.

A primeira destas funções predefinidas em Haskell é **putStr :: String -> IO ()** que é utilizada para mostrar *strings* no monitor. Assim,

```
putStr "\99a\116" = cat
putStr "Dunga\te h o bicho" = Dunga      eh o bicho
putStr ("jeri"++ "coa" ++ "coa" ++ "ra") = jericoacoara
```

Neste caso, as *strings* podem ser mostradas na tela do monitor. E se um valor não for uma *string*? Neste caso, a solução é transformar o valor em um valor do tipo **String** e usar a função **putStr**. Para esta missão, foi definida a função **show :: t -> String** que transforma um valor, de qualquer tipo, em um valor do tipo **String**. Além desta, a função **read :: String -> t** toma um valor do tipo **String** como argumento de entrada e o transforma em um valor do tipo **t**. Por exemplo,

```
show (5+7) = "12"
show (True && False) = "False"
read "True" = True
read "14" = 14
```

Exercícios:

1. Defina uma função para converter letras minúsculas em maiúsculas e que retorne o próprio caractere se a entrada não for um caractere minúsculo.
2. Defina uma função `charParaInt :: Char -> Int` que converte um dígito em seu valor (por exemplo, '8' em 8). O valor de um caractere não dígito deve ser 0 (zero).
3. Defina uma função `imprimeDigito :: Char -> String` que converte um dígito em sua representação em português. Por exemplo, '5' deve retornar "cinco".
4. Defina uma função `romanoParaString :: Char -> String` que converte um algarismo romano em sua representação em Português. Por exemplo, `romanoParaString 'V' = "cinco"`.
5. Defina uma função `emTresLinhas :: String -> String -> String -> String` que toma três *strings* como argumento e retorna uma única *string* mostrando as três em linhas separadas.
6. Defina uma função `replica :: String -> Int -> String` que toma uma *string* e um número natural *n* e retorna *n* cópias da *string*, todas juntas. Se *n* for 0, o resultado deve ser a *string* vazia (), se *n* for 1, retorna a própria *string*.

4.4.6 Metodologias de programação

Duas metodologias de construção de programas, bastante difundidas, são a programação *topdown* e a programação *bottom-up*, de plena aceitação pelos engenheiros de software. Haskell permite que estas duas técnicas sejam utilizadas em programas nela codificados. Elas serão mostradas a seguir.

Programação top-down

Vamos nos referir novamente ao problema das vendas, descrito anteriormente. Suponhamos que as quantidades de vendas sejam as mostradas na Tabela 4.7.

Tabela 4.7: Quantidade de vendas por semana.

Semana	Vendas
0	12
1	14
2	15
Total	41
Média	13.6667

Podemos construir uma função, **imprimeTab**, a partir da concatenação de outras funções que constroem *strings* e serão desenvolvidas a seguir.

```
imprimeTab :: Int -> String
imprimeTab n = putStr
    (cabecalho ++ imprimeSemanas n ++ imprimeTotal ++ imprimeMedia n)
```

Agora é necessário que estas funções componentes sejam definidas. Por exemplo, a função `cabecalho` é formada apenas por uma *string* para representar os títulos dos itens da Tabela 4.7.


```
cabecalho :: String
cabecalho
= "-----\n|| Semana | Vendas ||\n-----\n"
```

A função **imprimeSemanas** deve mostrar no monitor o número de cada semana e a quantidade de vendas correspondente. Ela será definida em função de uma outra função, **imprimeSemana**, e são definidas da seguinte forma:

```
imprimeSemanas :: Int -> String
imprimeSemanas 0 = imprimeSemana 0
imprimeSemanas n = imprimeSemanas (n-1) ++ imprimeSemana n

imprimeSemana :: Int -> String
imprimeSemana n = "|| "++ (show n)++" | "++ show (vendas n)++" ||\n"
```

A função **imprimeTotal** é utilizada apenas para imprimir o total de todas as vendas ocorridas em todas as semanas. Sua definição é a seguinte:

```
imprimeTotal :: String
imprimeTotal = "||-----||\n|| Total |"++show ((vendas 0)+
(vendas 1)+(vendas 2)+(vendas 3))++" ||\n-----\n"
```

A função **imprimeMedia** é deixada como exercício para o leitor.

A programação *top-down* é também conhecida “*dividir para conquistar*”, uma estratégia de guerra utilizada pelos antigos conquistadores. Como técnica de programação, consiste na divisão de um problema, inicialmente grande, em subproblemas menores e mais fáceis de serem resolvidos. Esta técnica envolve três fases, descritas da seguinte forma:

- **Fase de divisão.** Esta fase consiste na divisão do problema em subproblemas menores.
- **Fase de solução.** Esta fase consiste na solução de cada subproblema separadamente. Se o subproblema ainda for grande, deve ser usada a mesma técnica de divisão, de forma recursiva.
- **Fase de combinação.** Esta fase consiste na combinação das soluções dos subproblemas em uma solução única.

Programação bottom-up

A técnica de programação *bottom-up* é conhecida como “*programação dinâmica*” e sua motivação foi resolver um problema potencial que pode surgir ao se utilizar a técnica de programação *top-down*. Este problema diz respeito à possibilidade da geração de um grande número de subproblemas idênticos. Se estes subproblemas forem resolvidos separadamente, pode levar a uma replicação muito grande de trabalho. A técnica *bottom-up* consiste em resolver, inicialmente, as menores instâncias e utilizar estes resultados intermediários para resolver instâncias maiores.

Vamos considerar como exemplo a computação do *n*-ésimo número da sequência de Fibonacci. Utilizando a técnica *top-down*, esta função pode ser definida da seguinte forma:

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Uma representação gráfica das chamadas recursivas na execução desta função aplicada a 4, está mostrada na Figura 4.3. Pode ser observado que *fib 2* é computada duas vezes, *fib 1* é computada três vezes e *fib 0* é computada duas vezes.

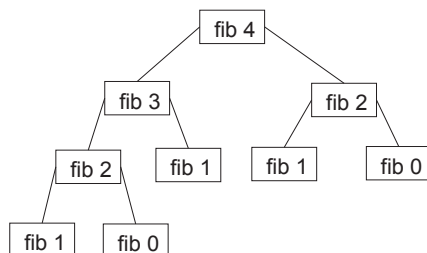


Figura 4.3: A execução de *fib 4* usando uma definição *top-down*.

Já a Figura 4.4 mostra a execução de *fib 4*, onde **fib** é definida utilizando uma técnica *bottom-up*, onde os números calculados são reutilizados nas computações seguintes. Uma relação de recorrência que descreve como computar grandes instâncias a partir de menores deve ser associada com cada aplicação. Esta relação deve se basear em um princípio conhecido como “*princípio da otimalidade*” que estabelece, informalmente, que uma solução é ótima se ela puder ser construída usando sub-soluções ótimas.

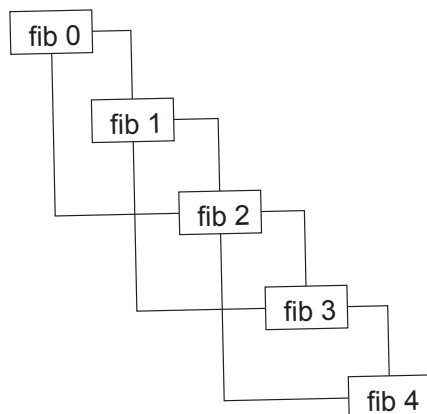


Figura 4.4: A execução de *fib 4* usando uma definição *bottom-up*.

A programação *top-down* parte de um caso geral para casos específicos, enquanto a modelagem *bottom-up* tem o sentido inverso desta orientação. Ela inicia com as definições mais particulares, para depois compô-las em uma forma mais geral. Para exemplificarmos esta situação, vamos redefinir a função **imprimeSemana** utilizando a função **rJustify**, que por sua vez usa a função **imprimeespacos** para preencher os espaços em branco necessários. A função **rJustify** é tal que `rJustify 10 "Maria" = " Maria"`.

```

imprimeespacos :: Int -> String
imprimeespacos 0 = ""
imprimeespacos n = " " ++ imprimeespacos n-1

rJustify :: Int -> String -> String
rJustify n nome
  | n > length nome = imprimeespacos (n - length nome) ++ nome
  | n == length nome = nome
  | otherwise = "Erro: a string nao cabe no intervalo dado"
  
```

```

imprimeSemana :: Int -> String
imprimeSemana n = rJustify offset (show n) ++
  rJustify offset (show (vendas n)) ++ "\n"
  where offset :: Int
        offset = 10

imprimeMedia :: Int -> String
imprimeMedia n = "| Media | " ++ rJustify 7 (show mediadeVendas n) ++ " |"

```

A função **mediadeVendas** continua sendo deixada como exercício para o leitor.

Exercícios:

1. Dê uma definição de uma função `tabeladeFatoriais :: Int -> Int -> String` que mostre, em forma de tabela, os fatoriais dos inteiros de m até n , inclusive de ambos.
2. Refaça o exercício anterior admitindo a possibilidade de entradas negativas e de que o segundo argumento seja menor que o primeiro.

4.4.7 Os tipos de dados estruturados de Haskell

Haskell também admite a possibilidade de que o usuário construa seus próprios tipos de dados, de acordo com as necessidades que ele tenha de simular problemas do mundo real. Os tipos estruturados são construídos a partir de outros tipos, primitivos ou estruturados. Esta é uma característica muito importante desta linguagem, por facilitar a vida dos programadores, permitindo um grau muito maior de abstração dos problemas a serem resolvidos.

O tipo produto cartesiano

O produto cartesiano é representado em Haskell pelas tuplas, que podem ser duplas, triplas, quádruplas, etc. Na maioria das linguagens de programação imperativas, este tipo de dados é implementado através de registros ou estruturas. Em Haskell, o tipo (t_1, t_2, \dots, t_n) consiste de n -uplas de valores (v_1, v_2, \dots, v_n) onde $v_1 :: t_1, v_2 :: t_2, \dots, v_n :: t_n$.

Por exemplo,

```

type Pessoa = (String, String, Int)
maria :: Pessoa
maria = ("Maria das Dores", "3225-0000", 22)

```

```

intP :: (Int, Int)
intP = (35, 45)

```

As funções sobre tuplas, apesar de poderem ser definidas de várias formas, são comumente definidas por *pattern matching*.

```

somaPar :: (Int, Int) -> Int
somaPar (x, y) = x + y

```

Os padrões podem ter constantes e/ou padrões aninhados.

```

shift :: ((Int, Int), Int) -> (Int, (Int, Int))
shift ((a,b),c) = (a, (b,c))

```

Podem ser definidas funções para mostrar casos particulares de uma tupla:

```
nome :: Pessoa -> String
fone :: Pessoa -> String
idade :: Pessoa -> Int

nome (n, p, a) = n
fone (n, p, a) = p
idade (n, p, a) = a
```

Assim, `nome maria = “Maria das Dores”`

Deve-se ter algum cuidado com os tipos de dados que, em algumas situações, podem conduzir a erros. Por exemplo, são diferentes:

```
somaPar :: (Int, Int) -> Int      e      somaDois :: Int -> Int -> Int
somaPar (a, b) = a + b           somaDois a b = a + b
```

Apesar dos resultados das aplicações das funções **somaPar** e **somaDois** serem os mesmos, elas são distintas. A função **somaPar** requer apenas um argumento, neste caso uma tupla, enquanto a função **somaDois** requer dois argumentos do tipo inteiro.

4.4.8 Escopo

O escopo de uma definição é a parte de um programa na qual ela é visível e portanto pode ser usada. Em Haskell, o escopo das definições é todo o *script*, ou seja, todo o arquivo no qual a definição foi feita. Por exemplo, vejamos a definição de **ehImpar n**, a seguir, que menciona a função **ehPar**, apesar desta ser definida depois. Isto só é possível porque elas compartilham o mesmo escopo.

```
ehImpar, ehPar :: Int -> Bool
ehImpar 0 = False
ehImpar n = ehPar (n-1)

ehPar 0 = True
ehPar n = ehImpar (n-1)
```

Definições locais

Haskell permite definições locais através da palavra reservada `where`. Por exemplo,

```
somaQuadrados :: Int -> Int -> Int
somaQuadrados n m = quadN + quadM
    where
        quadN = n * n
        quadM = m * m
```

As definições locais podem incluir outras definições de funções e podem usar definições locais a uma expressão, usando a palavra reservada **let**.

```
let x = 3 + 2; y = 5 - 1 in x^2 + 2*x*y - y
```

As definições locais são visíveis apenas na equação onde elas foram declaradas. As variáveis que aparecem do lado esquerdo da igualdade também podem ser usadas em definições locais, do lado esquerdo. Por exemplo,

```
maximoQuadrado x y
  |quadx > quady = quadx
  |otherwise = quady
    where
      quadx = quad x
      quady = quad y
      quad :: Int -> Int
      quad z = z * z
```

As definições locais podem ser usadas antes delas serem definidas e também podem ser usadas em resultados, em guardas ou em outras definições locais. Como exemplo,

```
maximasOcorrencias :: Int -> Int -> Int -> (Int, Int)
maximasOcorrencias n m p = (max, quantosIguais)
  where
    max = maximoDeTres n m p
    quantosIguais = quantosIguaisValor max n m p
    maximoDeTres :: Int -> Int -> Int -> Int
    maximoDeTres a b c = maximo (maximo (a, b), c)
    quantosIguaisValor :: Int -> Int -> Int -> Int -> Int
```

onde a função **quantosIguaisValor** pode ser definida de uma das formas mostradas na Tabela 4.8.

Tabela 4.8: Formas possíveis de definição da função **quantosIguaisValor**.

<pre>quantosIguaisValor valor n m p = ehN + ehM + ehP where ehN = if n == valor then 1 else 0 ehM = if m == valor then 1 else 0 ehP = if p == valor then 1 else 0</pre>	<pre>quantosIguaisValor valor n m p = ehvalor n + ehvalor m + ehvalor p where ehvalor :: Int -> Int ehvalor x = if x == valor then 1 else 0</pre>
---	--

4.4.9 Expressões condicionais

Como pode ser observado na definição da função **quantosIguaisValor**, a expressão condicional do tipo **if-then-else** foi utilizada como uma expressão válida em Haskell. Neste caso, ao usar a construção **if <expB> then <exp1> else <exp2>**, a expressão booleana **<expB>** é avaliada e, se o resultado for verdadeiro, a expressão **<exp1>** é escolhida para ser avaliada e seu resultado será o valor da expressão como um todo. Se, ao contrário, a avaliação da expressão booleana tiver valor falso, a expressão **<exp2>** é escolhida para ser avaliada e seu resultado será o da expressão.

A exemplo de muitas linguagens de programação, Haskell também suporta a construção **case**, utilizada quando múltiplos valores existem como resultado da avaliação de uma expressão

e, dependendo de cada um deles, uma expressão é escolhida para ser avaliada. Como exemplo, uma função **fun** que retorna um valor dependendo de sua entrada pode ser definida por:

```
fun x = case x of
  0 -> 50
  1 -> 100
  2 -> 150
  3 -> 200
  _ -> 500
```

4.4.10 Avaliação em Haskell

A avaliação usada por Haskell é chamada de *avaliação preguiçosa*, onde cada expressão é avaliada apenas uma vez e se necessário. Um cálculo só é realizado se for realmente necessário e seu valor é colocado em uma célula da *heap*. Se ele for novamente solicitado, já está calculado e pronto para ser utilizado. Por este motivo, Haskell, a exemplo de qualquer linguagem funcional e todas as modernas linguagens de programação, fazem o gerenciamento dinâmico de memória de forma automática, conhecido como *Garbage Collection* ou Coleta de lixo. Para mais detalhes sobre este tema o leitor deve consultar as referências [45, 46, 47, 48, 49].

Vamos mostrar, detalhadamente, a sequência de avaliações da aplicação de duas funções já definidas anteriormente, **somaQuadrados** e **maximasOcorrencias**.

```
somaQuadrados 4 3 = quadN + quadM
  where
    quadN = 4 * 4 = 16
    quadM = 3 * 3 = 9
           = 16 + 9
           = 25

maximasOcorrencias 2 1 2 = (max, quantosIguais)
  where
    max = maximoDeTres 2 1 2
         = maximo (maximo 2 1) 2
         ?? 2>=1 = True
         = maximo 2 2
         ?? 2>=2 = True
         = 2
    = (2, quantosIguais)
    where
      quantosIguais
        = quantosIguaisValor 2 2 1 2
        = ehValor 2 + ehvalor 1 + ehvalor 2
        where
          ehvalor 2 = if 2 == 2 then 1 else 0
                     = if True then 1 else 0
                     = 1
          = 1 + ehvalor 1 + ehvalor 2
          where
            ehvalor 1 = if 1 == 2 then 1 else 0
                       = if False then 1 else 0
                       = 0
```

```

= 1 + 0 + ehvalor 2
  where
    ehvalor 2 = if 2 == 2 then 1 else 0
               = if True then 1 else 0
               = 1
               = 1 + 0 + 1
               = 2
= (2, 2)

```

Esta sequência de cálculos deve ser acompanhada detalhadamente pelo leitor para entender a forma como o compilador (ou interpretador) Haskell executa seus cálculos. Este entendimento tem importância fundamental na construção de funções, notadamente de funções que tratam com listas potencialmente infinitas, a serem vistas mais adiante.

Exercícios:

1. Calcule os valores das expressões: **maximasOcorrencias 1 2 1** e **quantosIguaisValor 4 2 1 3**.
2. Defina uma função **cJustify :: Int -> String -> String** onde **cJustify n st** retorna uma *string* de tamanho *n*, adicionando espaços antes e depois de *st* para centralizá-la.
3. Defina uma função **stars :: Int -> String** de forma que **stars 3** retorna *******. Como deve ser tratada uma entrada negativa?

Exemplo. Vamos agora construir um exemplo bastante conhecido que é o de encontrar as raízes reais de uma equação do segundo grau com coeficientes reais, baseado em [55]. Neste caso teremos como entrada a equação $a * x^2 + b * x + c = 0$, sendo $a = 1.0$; $b = 5.0$ e $c = 6.0$.

Para esta solução, a saída será a *string*

A equação $1.0 * x^2 + 5.0 * x + 6.0 = 0.0$
tem duas raízes reais e distintas: -2.0 e -3.0.

Para isto vamos construir duas funções: **umaRaiz** para o caso da função ter duas raízes reais e iguais e **duasRaizes** para o caso dela ter duas raízes reais e distintas.

```

umaRaiz :: Float -> Float -> Float -> Float
umaRaiz a b c = -b / (2.0 * a)

```

```

duasRaizes :: Float -> Float -> Float -> (Float, Float)
duasRaizes a b c = (d + e, d - e)
  where
    d = -b/(2.0*a)
    e = sqrt (b^2 - 4.0*a*c)/(2.0*a)

```

```

saida :: Float -> Float -> Float -> String
saida a b c = cabecalho a b c ++ raizes a b c

```

```

cabecalho :: Float -> Float -> Float -> String
cabecalho a b c = "A equacao \n\n\t"++ show a ++ "*x^2 + " ++
  show b ++ "*x + " ++ show c ++ " = 0.0" ++ "\n\n\ttem "

```

```

raizes :: Float -> Float -> Float -> String

```

```

raizes a b c
| b^2 > 4.0 * a * c = "duas raizes reais e distintas: "
  ++ show f ++ " e " ++ show s
| b^2 == 4.0 * a * c = "duas raizes reais e iguais: "
  ++ show (umaRaiz a b c)
| otherwise = "nenhuma raiz real "
  where (f, s) = duasRaizes a b c

```

Na equação do segundo grau, se a entrada para o coeficiente *a* for zero, não será possível a divisão de qualquer número por ele. Neste caso, o programa deve abortar a execução e uma exceção deve ser feita, descrevendo o motivo. A função **umaRaiz** deve ser re-definida da seguinte forma:

```

umaRaiz a b c
| (a /= 0.0) = -b / (2.0 * a)
| otherwise = error "umaRaiz chamada com a == 0"

```

A re-definição da função **duasRaizes** é deixada para o leitor, como exercício.

4.5 Projeto de programas

A Engenharia de Software admite algumas metodologias para a construção de programas, de forma a obter melhores resultados, tanto em relação às soluções quanto em relação ao tempo de desenvolvimento. A sequência de passos mostrada a seguir, devida a Simon Thompson [55], é considerada um bom roteiro na codificação de programas para a solução de problemas usando Haskell:

- Verificar problemas similares e mais simples.
- Decidir os tipos para representação.
- Dividir para conquistar (abordagem *top-down*).
- Uma solução para um caso menor pode ser utilizada para um caso maior.
- Usar sempre que possível cláusulas *where*.
- Se uma expressão aparecer mais de uma vez, é forte candidata a ser declarada como uma função.
- Usar uma abordagem *bottom-up*.
- O *layout* do *script* é importante.

Exercícios:

Para os exercícios a seguir, considere os pontos do plano como sendo do tipo **Ponto = (Float, Float)**. As linhas do plano são definidas por seus pontos inicial e final e têm o tipo **Linha = (Ponto, Ponto)**.

1. Defina funções que retornem a ordenada e a abcissa de um ponto.
2. Defina uma função que retorne a norma de um vetor dado por suas coordenadas.

3. Se uma linha é determinada pelos pontos (x_1, y_1) e (x_2, y_2) , sua equação é definida por $(y - y_1)/(x - x_1) = (y_2 - y_1)/(x_2 - x_1)$. Defina uma função do tipo *valorY :: Float -> Linha -> Float* que retorna a ordenada y do ponto (x, y) , sendo dados x e uma linha.
4. Dados dois vetores, u e v , determine se eles são, ou não, colineares.

4.5.1 Provas de programas

Uma prova é uma argumentação lógica ou matemática para verificar se alguma premissa é ou não válida, em quaisquer circunstâncias.

Este tema tem importância fundamental na construção de programas, uma vez que deve-se buscar a garantia de que o programa esteja correto e que ele realiza apenas a ação para a qual foi criado. A prova de programas aumenta sua importância a cada dia, uma vez que os problemas estão se tornando cada vez mais complexos e desafiadores. Por exemplo, problemas nucleares ou outros que envolvam vidas humanas podendo colocar em jogo suas sobrevivências. Para estes casos, há de existir uma prova de que ele produz a solução correta.

No entanto, existe um dilema da parte dos usuários que, às vezes, têm dificuldades de realizar provas de programas, achando ser uma técnica de fundamentação matemática e trabalhosa. A prova de programas em uma linguagem imperativa é realmente tediosa, no entanto, como será visto, ela é bem menos difícil em uma linguagem funcional como Haskell.

Existem, em Haskell, três maneiras de realizar provas: a prova direta, a prova por casos e a prova por indução matemática. Vamos analisá-las através de exemplos, lembrando ao leitor que o domínio dessas técnicas só é conseguido através da prática.

Provas diretas

A prova direta é feita aplicando as definições das funções. Por exemplo, sejam as funções **troca**, **cicla** e **recicla**, definidas da seguinte forma:

```
troca :: (Int, Int) -> (Int, Int)
troca (a, b) = (b, a)                --def 1

cicla, recicla :: (Int, Int, Int) -> (Int, Int, Int)
cicla (a, b, c) = (b, c, a)          --def 2
recicla (a, b, c) = (c, a, b)        --def 3
```

A partir destas definições, podemos provar sentenças nas quais elas estejam envolvidas. Por exemplo, podemos provar que **troca (troca (a, b)) = (a, b)**, ou ainda que **cicla (recicla (a, b, c)) = recicla (cicla (a, b, c))**. Vejamos como isto pode ser feito:

```
troca (troca (a, b)) = troca (b, a)      --por def 1
                    = (a, b)             --por def 1
cicla (recicla (a, b, c)) = cicla (c, a, b) = (a, b, c) --por def 3 e 2
recicla(cicla (a, b, c)) = recicla (b, c, a) = (a, b, c) --por def 2 e 3
```

Portanto, são iguais os resultados e a prova está completa. Fácil, não?

Provas por casos

Seja a definição da função *maximo*, já feita anteriormente no início do Capítulo:

```

maximo :: Int -> Int -> Int
maximo n m
  | n >= m = n      --def 1
  | otherwise = m    --def 2

```

Seja a assertiva: “Para quaisquer números inteiros n e m , $\text{maximo } n \ m \geq n$ ”.

Para quaisquer números m e n definidos, tem-se: $m > n$ ou $m \leq m$. Então,

- **Caso 1:** se $n \geq m$: $\text{maximo } n \ m = n$ (por def 1) e $n \geq n$. Portanto, $\text{maximo } n \ m \geq n$.
- **Caso 2:** se $m > n$: $\text{maximo } n \ m = m$ (por def 2) e $m > n$. Portanto, $\text{maximo } n \ m > n$. Logo, $\text{maximo } n \ m \geq n$.

Exercícios:

1. Prove que $\text{cicla } (\text{cicla } (\text{cicla } (a, b, c))) = (a, b, c)$ para todo a, b e c .
2. Sendo $\text{somaTres } (a, b, c) = a + b + c$, para a, b e c inteiros, dê uma prova de que $\text{somaTres } (\text{cicla } (a, b, c)) = \text{somaTres } (a, b, c)$.
3. Dada a definição

```

trocaSe :: (Int, Int) -> (Int, Int)
trocaSe (a, b)
  | a <= b = (a, b)
  | otherwise = (b, a)

```

Prove que para todo a e b definidos, $\text{trocaSe}(\text{trocaSe}(a, b)) = \text{trocaSe}(a, b)$.

Indução matemática

Da Matemática, sabemos que o esquema de prova por indução dentro do conjunto dos números naturais é uma forma muito comum. Um sistema similar pode ser utilizado para provar programas codificados em Haskell.

Para provar que uma propriedade $P(n)$ é válida para todo natural n , deve-se:

- **Caso base:** Provar $P(n)$, para $n = 0$.
- **Passo indutivo:** Para $n > 0$, provar $P(n)$, assumindo que $P(n - 1)$ é válida.

Vejamos, por exemplo, a função **fatorial**:

```

fatorial :: Int -> Int
fatorial 0 = 1                -- (fat 1)
fatorial n = n * fatorial (n - 1) -- (fat 2)

```

Podemos provar a seguinte propriedade dos naturais: $P(n)$: $\text{fatorial } n > 0$, para todo natural n . O esquema de prova é feito da seguinte forma:

- **Caso base ($P(0)$):** $\text{fatorial } 0 = 1$ (por **fat 1**) e $1 > 0$. Logo $\text{fatorial } 0 > 0$, significando que a propriedade é válida para o caso base.

- **Passo indutivo ($P(n)$):** *fatorial* $n = n * \text{fatorial } (n - 1)$, (por **fat 2**) admitindo-se que $n > 0$. A hipótese de indução nos assegura que *fatorial* $(n - 1) > 0$, ou seja, a propriedade **P** é válida para $n - 1$. Assim o fatorial de n é o produto de dois fatores sendo ambos maiores que zero. O produto de dois números positivos é também positivo. Logo, maior que 0.
- **Conclusão:** como a propriedade **P** é válida para o caso base e para o passo indutivo, então ela é válida para todo n natural.

Esta última parte, a conclusão da prova, é uma componente importante da prova. É comum ver esquemas de prova, normalmente feitas por iniciantes, onde os dois passos são verificados, mas não existe a conclusão. Neste caso, a prova está incompleta.

Provas por Indução

Enquanto a indução formula provas para **P(0)**, **P(1)**, ..., a definição recursiva de **fatorial**, vista anteriormente, constrói resultados para **fatorial 0**, **fatorial 1**, A forma como $P(n-1)$ é assumida para se provar $P(n)$ é semelhante à forma usada por *fatorial* $(n-1)$ para encontrar o valor de **fatorial** **(n)**.

Este esquema de prova normalmente é aplicado a funções definidas por recursão primitiva representando tão somente um processo de tradução semelhante ao esquema de prova por indução matemática.

Simon Thompson escreveu um guia de passos a serem seguidos nos esquemas de provas por indução em Haskell [55]. A sequência de passos de provas proposta por ele é instrutiva e serve de roteiro, principalmente para quem está dando os primeiros passos em direção a este estudo. Com o decorrer do tempo, este esquema passa a ser um processo automático.

- Estágio 0: escrever o objeto da prova em português,
- Estágio 1: escrever o objeto da prova em linguagem formal,
- Estágio 2: escrever os sub-objetos da prova por indução:
 $P(0)$:
 $P(n)$, para todo $n > 0$, assumindo $P(n-1)$
- Estágio 3: Provar $P(0)$
- Estágio 4: Provar $P(n)$, para $n > 0$, lembrando que deve e pode usar $P(n-1)$

Exemplo: Sejam as definições das funções a seguir:

```
power2 :: Int -> Int
power2 0 = 1 (1)
power2 r = 2 * power2 (r - 1) (2)

sumPowers :: Int -> Int
sumPowers 0 = 1 (3)
sumPowers r = sumPowers (r-1) + power2 r (4)
```

Prove que **sumPowers** $n + 1 = \text{power2 } (n + 1)$.

Solução:

Estágio 0: provar que a soma das potências de 2 de 0 a n , adicionada a 1 é igual a $(n + 1)$ -ésima potência de 2.

Estágio 1: provar **P(n)**: $\text{sumPowers } n + 1 = \text{power2 } (n + 1)$

Estágio 2: $\text{sumPowers } 0 + 1 == \text{power2 } (0 + 1)$, para $n = 0$?

$\text{sumPowers } n + 1 == \text{power2 } (n + 1)$, para $n > 0$?

assumindo que $\text{sumPowers } (n - 1) + 1 = \text{power2 } n$

Estágio 3: $\text{sumPowers } 0 + 1 = 1 + 1 = 2$ -por (3)

$\text{power2 } (0 + 1) = 2 * \text{power2 } 0 = 2 * 1 = 2$ -por (2)

logo, a prova é válida para o caso base.

$\text{sumPowers } n + 1 = \text{sumPowers } (n - 1) + \text{power2 } n + 1$ -por (4)

$= \text{sumPowers } (n - 1) + 1 + \text{power2 } n$ -pela comutatividade de +

$= \text{power2 } n + \text{power2 } n$ -pela hipótese de indução

$= 2 * \text{power2 } n$

Conclusão: como a propriedade é válida para $n = 0$ e para $n > 0$, então é válida para todo número natural n .

Exercícios:

1. Usando as definições anteriores de **fatorial** e **power2** prove que, para todo número natural n , $\text{fatorial } (n + 1) \geq \text{power2 } n$.
2. Usando as definições anteriores de **fib** e **power2** prove que, para todo número natural n , $\text{fib } (n + 1) \geq \text{power2 } (n \text{ div } 2)$.
3. Dada a definição anterior para **venda** dê uma prova de que, para todo número natural n , $\text{venda } n \geq 0$.

4.6 Resumo

Neste Capítulo, foi dado início ao estudo de programação em Haskell. Foram vistos os tipos de dados primitivos e as tuplas. Alguns exercícios foram resolvidos para dar uma noção ao usuário da potencialidade da linguagem e outros foram deixados para o leitor.

O livro de Simon Thompson [55] foi a fonte mais utilizada para este estudo, por apresentar um conteúdo teórico bem estruturado e fundamentado, além de muitos exercícios resolvidos e muitos problemas propostos.

O livro de Richard Bird [6] é outra fonte importante de exercícios resolvidos e propostos, apesar de sua sequência de abordagens seguir uma ordem distinta, exigindo um conhecimento anterior sobre programação funcional, o que o torna mais difícil de ser seguido por iniciantes neste tema.

Outra referência importante é o livro de Paul Hudak [15] que apresenta a programação funcional em Haskell com exemplos aplicados à Multimídia, envolvendo a construção de um editor gráfico e de um sistema utilizado em música, entre outros. Para quem imagina que Haskell só pode ser aplicado a problemas da Matemática, esta referência põe por terra este argumento.

4.7 Exercícios propostos

1. Dado um número natural $n > 0$, n é dito perfeito se a soma de seus divisores, incluindo o número 1, é igual ao próprio n . O primeiro número natural perfeito é o número 6, porque

- $6 = 1 + 2 + 3$. Implemente em Haskell uma função que informe se n é, ou não, um número perfeito.
2. Dado um número natural $n > 0$, implemente em Haskell uma função que informe se n é, ou não, um número primo.
 3. Dê uma definição em Haskell da função *fat* que calcula o fatorial de n , onde n é um inteiro positivo.
 4. Dê uma definição em Haskell de uma função de m e n que retorna o produto $m * (m + 1) * \dots * (n - 1) * n$, para $m < n$ e retorne 0 se $m \geq 0$.
 5. Dê uma definição em Haskell de uma função que retorne o i -ésimo número da sequência de Fibonacci (0, 1, 1, 2...).
 6. Implemente, em Haskell, uma função que calcule o resto da divisão de dois números inteiros positivos.
 7. Implemente, em Haskell, uma função que calcule a divisão inteira entre dois números inteiros positivos.
 8. Implemente, em Haskell, uma função que calcule o máximo divisor comum entre dois números inteiros positivos.
 9. Implemente, em Haskell, uma função que calcule o mínimo múltiplo comum entre dois números inteiros positivos.
 10. Implemente, em Haskell, uma função que calcule quantas maneiras é possível escolher n objetos de uma coleção original de m objetos, onde $m \geq n$, matematicamente conhecida como combinação de m objetos tomados n a n .
 11. Implemente, em Haskell, a função de Ackermann, definida por:
 - (a) $a(m,n) = 1$, se $m = 0$
 - (b) $a(m,n) = a(m-1, 1)$, se $m \neq 0$ e $n = 0$
 - (c) $a(m,n) = a(m-1, a(m, n-1))$ se $m \neq 0$ e $n \neq 0$
 12. Implemente, em Haskell, as funções $OR(x,y)$ e $AND(x,y)$ da Lógica Proposicional.
 13. Implemente, em Haskell, as funções $XOR(x,y)$ (ou exclusivo) e $IMPLICA(p,q)$ da Lógica Proposicional.
 14. Implemente, em Haskell, uma função que aplicada a 3 valores reais (a , b e c) apresente como resultado as raízes reais ou imaginárias da equação do segundo grau da forma $ax^2 + bx + c = 0$, lembrando ainda que se as raízes forem reais elas podem ser iguais ou distintas e este fato também deve ser reportado na definição da função.

Capítulo 5

O tipo Lista

*“Computers specially designed for applicative languages
implement recursive functions very efficiently.
Also, architectural features can be included in conventional
computers that significantly increase
the speed of recursive-function invocations.”*
(Bruce J. MacLennan in [30])

5.1 Introdução

Lista é o tipo de dado mais importante nas linguagens funcionais. Todas elas a implementam como um tipo primitivo, juntamente com uma gama imensa de funções para a sua manipulação.

Em Haskell, as listas são denotadas por seus elementos entre colchetes. Por exemplo, `[1,2,3,4,1,3]` é uma lista de inteiros, `[True,False]` é uma lista de booleanos, `['a', 'a', 'b']` é uma lista de caracteres e `["Marta", "Marlene"]` é uma lista de *strings*. Há, no entanto, que se diferenciar as listas homogêneas, que são as listas onde todos os valores são do mesmo tipo, das listas heterogêneas, onde as componentes podem ter mais de um tipo. Haskell só admite listas homogêneas. Por exemplo, `[False, 2, "Maria"]` não é uma lista em Haskell, por ser heterogênea. Em compensação, podemos ter a lista `[totalVendas, totalVendas]`, que tem o tipo `[Int -> Int]` como também a lista `[[12,1], [3,4], [4,4,4,4,4], []]` que tem o tipo `[[Int]]`, uma lista de listas de inteiros, além de outras possibilidades.

Existem duas formas como as listas podem se apresentar:

- a lista vazia, simbolizada por `[]`, que pode ser de qualquer tipo. Por exemplo, ela pode ser de inteiros, de booleanos, etc. Dito de outra forma, `[]` é do tipo `[Int]` ou `[Bool]` ou `[Int -> Int]`, significando que a lista vazia está na interseção de todas as listas, sendo o único elemento deste conjunto.
- a lista não vazia, simbolizada por `(a : x)`, onde **a** representa um elemento da lista, portanto tem um tipo, e **x** representa uma lista composta de elementos do mesmo tipo de **a**. O elemento **a** é chamado de “cabeça” e **x** é a “cauda” da lista.

Algumas características importantes das listas em Haskell, são:

- A ordem em uma lista é importante, ou seja, `[1,3] /= [3,1]` e `[False] /= [False, False]`.

- A lista $[m \dots n]$ é igual à lista $[m, m+1, \dots, n]$. Por exemplo, $[1 \dots 5] = [1, 2, 3, 4, 5]$. A lista $[3.1 \dots 7.0] = [3.1, 4.1, 5.1, 6.1]$.
- A lista $[m,p \dots n]$ é igual à lista de m até n em passos de $p-m$. Por exemplo, $[7,6 \dots 3] = [7, 6, 5, 4, 3]$ e $[0.0, 0.3 \dots 1.0] = [0.0, 0.3, 0.6, 0.9]$.
- A lista $[m..n]$, para $m > n$, é vazia. Por exemplo, $[7 \dots 3] = []$.
- A lista vazia não tem cabeça e nem cauda. Se tivesse qualquer destes dois componentes, não seria vazia.
- A lista não vazia tem cabeça e cauda, onde a cauda pode ser vazia, ou não.

5.2 Funções sobre listas

As funções para a manipulação de listas são declaradas da mesma forma como são declaradas para processar outros tipos de dados, usando casamento de padrões. Neste caso, os padrões são apenas dois: a lista vazia e a lista não vazia. Por exemplo,

```
somaLista :: [Int] -> Int
somaLista [] = 0
somaLista (x:xs) = x + somaLista xs
```

A função **somaLista** toma como argumento uma lista de inteiros e retorna, como resultado, um valor que é a soma de todos os elementos da lista argumento. Se a lista for vazia (`[]`), a soma será 0. Se a lista não for vazia ($x : xs$), o resultado será a soma de sua cabeça (x) com o resultado da aplicação da mesma função `somaLista` à cauda da lista (xs). Esta definição é recursiva, uma característica muito utilizada, por ser a forma usada para fazer iteração em programas funcionais. Devemos observar que a ordem em que os padrões são colocados tem importância fundamental. No caso em voga, primeiramente foi feita a definição para a lista vazia e depois para a lista não vazia. Dependendo da necessidade do programador, esta ordem pode ser invertida.

Vejamos a sequência de cálculos da aplicação da função `somaLista` à lista $[2, 3, 5, 7]$.

```
somaLista [2,3,5,7] = 2 + somaLista [3,5,7]
                    = 2 + 3 + somaLista [5,7]
                    = 5 + somaLista [5,7]
                    = 5 + 5 + somaLista [7]
                    = 10 + somaLista [7]
                    = 10 + 7 + somaLista []
                    = 17 + somaLista []
                    = 17 + 0
                    = 17
```

O construtor de listas : (`cons`)

O construtor de listas, chamado de *cons* e sinalizado por `:` (dois pontos), tem importância fundamental na construção de listas. Ele é um operador (uma função) que toma como argumentos um elemento, de um tipo, e uma lista de elementos deste mesmo tipo e insere este elemento como a *cabeça* da nova lista. Por exemplo,

```
10 : [] = [10]
2 : 1 : 3 : [] = 2 : 1 : [3] = 2 : [1,3] = [2,1,3]
```

significando que *cons* $(:)$ associa seus componentes pela direita, ou seja: $a : b : c = a : (b : c) \neq (a : b) : c$

Mas qual o tipo de *cons*? Observando que $4 : [3] = [4, 3]$, então *cons* tem o tipo:

$(:) :: \text{Int} \rightarrow [\text{Int}] \rightarrow [\text{Int}]$

No entanto, verificamos também que $\text{True} : [\text{False}] = [\text{True}, \text{False}]$. Agora *cons* tem o tipo:

$(:) :: \text{Bool} \rightarrow [\text{Bool}] \rightarrow [\text{Bool}]$

Isto mostra que o operador *cons* é polimórfico. Desta forma, seu tipo é:

$(:) :: t \rightarrow [t] \rightarrow [t]$

onde $[t]$ é uma lista de valores, de qualquer tipo, desde que seja homogênea.

Construindo funções sobre listas

Em Haskell, já existe um grande número de funções predefinidas para a manipulação de listas. Estas funções fazem parte do arquivo **Prelude.hs**, carregado no momento em que o sistema é chamado e permanece ativo até o final da execução. Um resumo destas funções, com seus tipos e exemplos de utilização, pode ser visto na Tabela 5.1.

Tabela 5.1: Algumas funções sobre listas do Prelude.hs.

Função	Tipo	Exemplo
:	$a \rightarrow [a] \rightarrow [a]$	$3:[2,5]=[3,2,5]$
++	$[a] \rightarrow [a] \rightarrow [a]$	$[3,2]++[4,5]=[3,2,4,5]$
!!	$[a] \rightarrow \text{Int} \rightarrow a$	$[3,2,1]!!0=3$
concat	$[[a]] \rightarrow [a]$	$[[2],[3,5]]=[2,3,5]$
length	$[a] \rightarrow \text{Int}$	$\text{length } [3,2,1]=3$
head	$[a] \rightarrow a$	$\text{head } [3,2,5]=3$
last	$[a] \rightarrow a$	$\text{last } [3,2,1]=1$
tail	$[a] \rightarrow [a]$	$\text{tail } [3,2,1]=[2,1]$
init	$[a] \rightarrow [a]$	$\text{init } [3,2,1]=[3,2]$
replicate	$\text{Int} \rightarrow a \rightarrow [a]$	$\text{replicate } 3 \text{ 'a'}=['a','a','a']$
take	$\text{Int} \rightarrow [a] \rightarrow [a]$	$\text{take } 2 \text{ } [3,2,1]=[3,2]$
drop	$\text{Int} \rightarrow [a] \rightarrow [a]$	$\text{drop } 2 \text{ } [3,2,1]=[1]$
splitAt	$\text{Int} \rightarrow [a] \rightarrow ([a], [a])$	$\text{splitAt } 2 \text{ } [3,2,1]=([3,2],[1])$
reverse	$[a] \rightarrow [a]$	$\text{reverse } [3,2,1]=[1,2,3]$
zip	$[a] \rightarrow [b] \rightarrow [(a, b)]$	$\text{zip } [3,2,1] [5,6]=[(3,5),(2,6)]$
unzip	$[(a, b)] \rightarrow ([a], [b])$	$\text{unzip } [(3,5),(2,6)]=([3,2],[5,6])$
and	$[\text{Bool}] \rightarrow \text{Bool}$	$\text{and } [\text{True},\text{False}]=\text{False}$
or	$[\text{Bool}] \rightarrow \text{Bool}$	$\text{or } [\text{True},\text{False}]=\text{True}$
sum	$[\text{Int}] \rightarrow \text{Int}$	$\text{sum } [2,5,7]=14$
	$[\text{Float}] \rightarrow \text{Float}$	$\text{sum } [3.0,4.0,1.0]=8.0$
product	$[\text{Int}] \rightarrow \text{Int}$	$\text{product } [1,2,3]=6$
	$[\text{Float}] \rightarrow \text{Float}$	$\text{product } [1.0,2.0,3.0]=6.0$

Além das funções predefinidas, o usuário também pode construir funções para manipular listas. Aqui a criatividade é o limite. Vamos mostrar isto através de um exemplo simples e depois através de outro mais complexo.

Vamos construir uma função que verifica se um determinado número inteiro pertence, ou não, a uma lista de inteiros. Para isto vamos construir a função **pertence**:

```
pertence :: Int -> [Int] -> Bool
pertence b [ ] = False
pertence b (a:x) = (b == a) || pertence b x
```

Esta mesma função também pode ser codificada de outra forma, usando guardas:

```
pertence b [ ] = False
pertence b (a:x)
  | b == a = True
  | otherwise = pertence x b
```

Vamos mostrar agora a definição de duas funções, **takeWhile** e **getUntil**, ambas utilizando um teste booleano representado por uma função que retorne um valor booleano e uma lista de elementos de um tipo variável. A função **takeWhile** retorna a lista dos primeiros elementos da lista passada como argumento e que atendem ao teste booleano. A função **getUntil** retorna a lista dos primeiros elementos da lista passada como argumento e que não atendem ao teste booleano. Suas definições se encontram na tabela mostrada a seguir.

<pre>takeWhile :: (a->Bool)->[a]->[a] takeWhile p [] = [] takeWhile p (x : xs) p x = x : takeWhile p xs otherwise = []</pre>	<pre>getUntil :: (a->Bool)->[a]->[a] getUntil p [] = [] getUntil p (x : xs) p x = [] otherwise = x : getUntil p xs</pre>
--	--

Modelagem. Voltemos nossa atenção agora para os exemplos das funções **espelhaH** e **espelhaV**, vistas no Capítulo anterior. Suponhamos que um cavalo seja um objeto do tipo **Quadro**. Um **Quadro** pode ser modelado por uma matriz (12x12), de caracteres. Desta forma, um cavalo é uma lista de 12 linhas e cada linha é uma lista de 12 caracteres, ou seja, um cavalo é uma lista de listas de caracteres, conforme pode ser visto graficamente na representação a seguir, onde os pontos estão colocados apenas para facilitar a contagem, ou seja, eles estão colocados apenas para representar os caracteres em branco. Os resultados das aplicações das funções **espelhaH** e **espelhaV** a um cavalo também estão mostrados na mesma Figura, a seguir.

cavalo	espelhaH cavalo	espelhaV cavalo
.....##...##....	...##.....
.....##..#..#.#....	..#...##....
...##.....#.#..#....	.#.....##...
..#.....#.	...#...#....	.#.....#..
..#...#...#.	..#...#....	.#...#...#..
..#...###.#.	..#...#...##.	.#...###...#..
..#...#...##.	..#...###.#.	...###...#..
..#...#.....	..#...#...#.#...#..
...#...#....	..#.....#.#...#...
....#..#....	...##.....#.#...#....
.....#.#....##...#.#.#.....
.....##....##....##.....

A declaração do tipo **Quadro** pode ser feita da seguinte forma:

```
type Linha = [Char]
type Quadro = [Linha]
```

As funções **espelhaH** e **espelhaV** também podem ser declaradas a partir de outras funções já definidas para outras finalidades, o que proporciona ainda mais flexibilidade ao programador. Por exemplo,

```
espelhaH cav = reverse cav          --inverte os elementos de uma lista
espelhaV cav = map reverse cav
```

A função **reverse**, quando aplicada a uma lista de elementos de qualquer tipo produz, como resultado, uma outra lista com os mesmos elementos da primeira lista, na ordem inversa. Por exemplo, **reverse** **[1,2,3]** = **[3,2,1]** e **reverse** **['a', 'b', 'c']** = **['c', 'b', 'a']**. Isto significa que a função **reverse** aplicada a cada linha do cavalo, que é uma lista de caracteres, dá, como resultado, a mesma linha mas com a ordem de seus caracteres invertida. A função **map** toma a função **reverse**, o primeiro de seus dois argumentos, e a aplica a cada uma das linhas de seu segundo argumento (um cavalo). As funções **map** e **reverse** são predefinidas em Haskell e elas serão objeto de estudo mais profundo nas próximas seções.

Algumas conclusões importantes podem ser tiradas a partir destes exemplos:

- função **reverse** pode ser aplicada a uma lista de valores de qualquer tipo. Isto significa que uma mesma função pode ser aplicada a mais de um tipo de dados. Isto significa polimorfismo, ou seja, a utilização genérica de uma função, aumentando a produtividade de software.
- Um argumento da função **map** é **reverse**, uma função. Isto significa que uma função pode ser passada como parâmetro para uma outra função. Neste caso, diz-se que as funções são consideradas como cidadãos de primeira categoria, permitindo-se a elas os mesmos direitos que qualquer outro tipo de dado.
- O resultado da aplicação da função **map** à função **reverse** é uma outra função que é aplicada a um outro argumento que, neste caso, é uma lista. Neste caso, diz-se que, nas linguagens funcionais, as funções são de *alta ordem*, ou seja, podem ser passadas como argumentos de uma função e também podem retornar como resultados da aplicação de uma função.

Exercícios:

1. Dada a definição da função **dobra**

```
dobra :: [Int] -> [Int]
dobra [ ] = [ ]
dobra (a:x) = (2 * a) : dobra x
```

Calcule **dobra** **[3,4,5]** passo a passo.

2. Escreva **[False, False, True]** e **[2]** usando **:** e **[]**.
3. Calcule **somaLista** **[30, 2, 1, 0]**, **dobra** **[0]** e **"cafe"++ "com"++ "leite"**.
4. Defina a função **product** **:: [Int] -> Int** que retorna o produto de uma lista de inteiros.

5. Defina a função `and :: [Bool] -> Bool` que retorna a conjunção da lista. Por exemplo, `and[e1, e2, ... en] = e1 & e2 & ... & en`.
6. Defina a função `concat :: [[Int]] -> [Int]` que concatena uma lista de listas de inteiros transformando-a em uma lista de inteiros. Por exemplo, `concat[[3, 4], [2], [4, 10]] = [3, 4, 2, 4, 10]`.

5.3 Pattern matching revisado

O casamento de padrões já foi analisado anteriormente, mas sem qualquer profundidade ou formalismo. Esta é uma técnica bastante utilizada em programas funcionais e a maioria das linguagens funcionais a incorporam para dar funcionalidade e aplicabilidade a estas linguagens.

Agora ele será revisto com uma nova roupagem, apesar de usar conceitos já conhecidos, mas através de uma definição formal.

Os padrões em Haskell são dados por:

- valores literais como `-2`, `'C'` e `True`,
- variáveis como `x`, `num` e `maria`,
- o caractere `_` (sublinhado) casa com qualquer argumento,
- um padrão de tuplas (p_1, p_2, \dots, p_k) . Um argumento para casar com este padrão tem de ser da forma (v_1, v_2, \dots, v_k) , onde cada v_i deve ser do tipo p_i e
- um construtor aplicado a outros padrões. Por exemplo, o construtor de listas $(p_1 : p_2)$, onde p_1 e p_2 são padrões.

Nas linguagens funcionais, a forma usada para verificar se um padrão casa, ou não, com um valor é realizada da seguinte maneira:

- primeiramente, verifica-se se o argumento está na forma correta e
- depois associam-se valores às variáveis dos padrões.

Exemplo. A lista `[2,3,4]` casa com o padrão $(a : x)$ porque:

1. ela tem cabeça e tem cauda, portanto é uma lista correta. Portanto,
2. `2` é associado com a cabeça a e `[3,4]` é associada com a cauda x .

Pode-se perguntar: em que situações um argumento a casa com um padrão p ? Esta questão é respondida através da seguinte lista de cláusulas:

- Se p for uma constante, a casa com p se $a == p$.
- Se p for uma variável, a casa com p e a será associada com p .
- Se p for uma tupla de padrões (p_1, p_2, \dots, p_k) , a casa com p se a for uma tupla (a_1, a_2, \dots, a_k) e se cada a_i casar com cada p_i .

- Se p for uma lista de padrões $(p_1 : p_2)$, a casa com p se a for uma lista não vazia. Neste caso, a cabeça de a é associada com p_1 e a cauda de a é associada com p_2 .
- Se p for um sublinhado $(_)$, a casa com p , mas nenhuma associação é feita. O sublinhado age como se fosse um teste.

Exemplo. Seja a função **zip** definida da seguinte forma:

```
zip (a:x) (b:y) = (a, b) : zip x y
zip _ _ = [ ]
```

Se os argumentos de **zip** forem duas listas, ambas não vazias, forma-se a tupla com as cabeças das duas listas, que será incorporada à lista de tuplas resultante e o processo continua com a aplicação recursiva de **zip** às caudas das listas argumentos. Este processo continua até que o padrão de duas listas não vazias falhe. Isto só vai acontecer quando uma das listas, ou ambas, for vazia. Neste caso, o resultado será a lista vazia e a execução da função termina.

Vamos verificar o resultado de algumas aplicações.

```
zip [2,3,4] [4,5,78] = [(2,4), (3,5), (4,78)].
zip [2,3] [1,2,3] = [(2,1), (3,2)]
```

Exercícios:

1. Defina uma função **somaTriplas** que soma os elementos de uma lista de triplas de números, **(c, d, e)**.
2. Defina uma função **somaPar** que dê como resultado a lista das somas dos elementos de uma lista de pares de números.
3. Calcule **somaPar [(2,3), (96, -7)]**, passo a passo.
4. Defina uma função **unzip :: [(Int, Int)] -> ([Int], [Int])** que transforma uma lista de pares em um par de listas. Sugestão: defina antes as funções **unZipLeft**, **unZipRight** :: **[(Int, Int)] -> [Int]**, onde **unZipLeft [(2,4), (3,5), (4,78)] = [2,3,4]** e **unZipRight [(2,4), (3,5), (4,78)] = [4,5,78]**.

Exemplo. Agora vamos analisar um exemplo mais detalhado da aplicação de listas em Haskell, baseado em Simon Thompson [55]. Seja um banco de dados definido para contabilizar as retiradas de livros de uma Biblioteca, por várias pessoas. Para simular esta situação, vamos construir uma lista de tuplas compostas pelo nome da pessoa que tomou emprestado um livro e do título do livro. Para isto, teremos:

```
type Pessoa = String
type Livro = String
type BancodeDados = [(Pessoa, Livro)]
```

Vamos construir uma lista (uma base de dados) fictícia para servir apenas de teste, ou seja, vamos supor que, em um determinado momento, esta lista esteja composta das seguintes tuplas:

```
teste = [("Paulo", "A Mente Nova do Rei"), ("Ana", "O Segredo de Luiza"),
("Paulo", "O Pequeno Principe"), ("Mauro", "O Capital"),
("Francisco", "O Auto da Compadecida")]
```

Vamos definir funções para realizar as seguintes tarefas:

1. Operações de consulta:

- Uma função que informa os livros que uma determinada pessoa tomou emprestado.
- Uma função que informa todas as pessoas que tomaram emprestado um determinado livro.
- Uma função que informa se um determinado livro está ou não emprestado.
- Uma função que informa a quantidade de livros que uma determinada pessoa tomou emprestado.

2. Operações de atualização:

- Uma função que atualiza a base de dados, quando um livro é emprestado a alguém.
- Uma função que atualiza a base de dados quando um livro é devolvido.

Inicialmente, vamos construir a função **livrosEmprestados** que pode ser utilizada para servir de roteiro para a definição das outras funções de consulta, deixadas, como exercício, para o leitor.

```
livrosEmprestados :: BancodeDados -> Pessoa -> [Livro]
livrosEmprestados [ ] _ = [ ]
livrosEmprestados ((inquilino, titulo) : resto) fulano
  | inquilino == fulano = titulo : livrosEmprestados resto fulano
  | otherwise = livrosEmprestados resto fulano
```

Vamos agora definir as funções de atualização:

```
tomaEmprestado :: BancodeDados -> Pessoa -> Livro -> BancodeDados
tomaEmprestado dBase pessoa titulo = (pessoa, titulo) : dBase

devolveLivro :: BancodeDados -> Pessoa -> Livro -> BancodeDados
devolveLivro ((p, t): r) f l
  | p == f && t == l = r
  | otherwise = (p,t) : devolveLivro r f l
devolveLivro [ ] ful tit = error "Nao ha livro emprestado"
```

Que motivos o leitor imagina que tenhamos levado em conta na definição da função **devolveLivro**, a exemplo da função **zip**, definida anteriormente, preferindo apresentar a definição para o padrão de lista vazia após a definição para o padrão de lista não vazia, quando o normal seria apresentar estes padrões na ordem inversa?

Exercício. Modifique a base de dados da Biblioteca anterior e as funções de acesso, de forma que:

- exista um número máximo de livros que uma pessoa possa tomar emprestado,
- exista uma lista de palavras-chave associadas a cada livro, de forma que cada livro possa ser encontrado através das palavras-chave a ele associadas, e
- existam datas associadas aos empréstimos, para poder detectar os livros com datas de empréstimos vencidas.

5.4 Compreensões e expressões ZF (Zermelo-Fraenkel)

As compreensões, também conhecidas como expressões ZF, devidas a Zermelo e Fraenkel, representam uma forma muito rica de construção de listas. O domínio desta técnica permite ao programador resolver muitos problemas de maneira simples e, em muitos casos, inusitada. A sintaxe das expressões ZF é muito próxima da descrição matemática de conjuntos por intensionalidade, exprimindo determinadas propriedades. As diferenças se verificam apenas nos sinais utilizados nas representações, mas a lógica subjacente é a mesma. Vamos inicialmente mostrar estas semelhanças através de exemplos e depois vamos formalizar sua sintaxe.

Vamos supor `ex = [2,4,7]`. Usando `ex`, podemos construir `ex1`, a lista cujos elementos sejam o dobro dos elementos de `ex`, da seguinte forma:

```
ex1 = [2*a | a<-ex]
```

Desta forma `ex1 = [4,8,14]`. Se quisermos encontrar a lista `ex2` composta dos elementos de `ex` que sejam pares, podemos declarar

```
ex2 = [a | a<-ex, a 'mod' 2 == 0]
```

Neste caso, `ex2 = [2,4]`.

A partir destes exemplos, podemos verificar que a sintaxe das expressões ZF é realmente simples. Formalmente ela é dada da seguinte forma:

$[e|q_1, \dots, q_k]$ onde cada q_i é um qualificador, que pode ter uma das duas seguintes formas:

1. um gerador do tipo $p < -lExp$, onde p é um padrão e $lExp$ é uma lista, ou
2. pode ser um teste do tipo $bExp$, uma expressão booleana.

Propriedades de uma expressão ZF

As expressões ZF podem ser utilizadas exercitando a criatividade de cada usuário em suas diversas possibilidades de construção. Neste ponto, a diferença entre as linguagens funcionais e as outras é notável. Este fato se tornou tão importante que algumas linguagens já a implementam, a exemplo de **Phyton**. Suas principais propriedades são as seguintes:

- Os geradores podem ser combinados com nenhuma, uma ou mais expressões booleanas. Sendo `ex` a lista do Exemplo anterior, então

```
[2*a | a <- ex, a 'mod' 2 == 0, a > 3] = [8]
```

- Pode-se usar qualquer padrão à esquerda de `< -`

```
somaPares :: [(Int, Int)] -> [Int]
somaPares listadePares = [a + b | (a, b) <- listadePares]
somaPares [(2,3), (4,5), (6,7)] = [5,9,13]
```

- Pode-se adicionar testes

```
novaSomaPares :: [(Int, Int)] -> [Int]
novaSomaPares listadePares = [a + b | (a, b) <- listadePares, a < b]
novaSomaPares [(2,3), (5,4), (7,6)] = [5]
```

- É possível colocar múltiplos geradores e combinar geradores e testes booleanos.
- Uma expressão **lExp** ou **bExp** que aparece em um qualificador q_i pode referenciar variáveis usadas nos padrões dos qualificadores q_1 até q_{i-1} .

Exemplos resolvidos:

1. A função **fazpares** que constrói uma lista de pares de elementos de duas listas quaisquer:

```
fazpares :: [t] -> [u] -> [(t,u)]
fazpares l m = [ (a, b) | a <- l, b <- m]
fazpares [1,2,3] [4,5] = [(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]
```

2. A função **pares** que constrói uma lista de pares de inteiros todos menores que um determinado valor inteiro n :

```
pares :: Int -> [(Int, Int)]
pares n = [ (a, b) | a <- [1 .. n], b <- [1 .. a]]
pares 3 = [ (1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]
```

3. Os triângulos retângulos de lados menores ou iguais a um certo valor inteiro:

```
trirets :: Int -> [(Int, Int, Int)]
trirets n = [ (a, b, c) | a <- [2 .. n], b <- [a+1 .. n],
                    c <- [b+1 .. n], a * a + b * b == c * c]
trirets 100 = [(3,4,5), (5,12,13), (6,8,10), ..., (65,72,97)]
```

Comentários:

No exemplo 1) deve ser observada a forma como as expressões são construídas. O primeiro elemento escolhido, **a**, vem da lista **l**. Para ele, são construídos todos os pares possíveis com os elementos, **b**, que vêm do outro gerador, a lista **m**. Assim toma-se o elemento **1** da lista **[1,2,3]** e formam-se os pares com os elementos **4** e **5** da lista **[4,5]**. Agora escolhe-se o segundo elemento da lista **l**, o valor **2**, e formam-se os pares com os elementos da lista **m**. Finalmente, repete-se este processo para o terceiro elemento da lista **l**. Esta forma de construção tem importância fundamental, sendo responsável pela construção de listas potencialmente infinitas, um tópico descrito no próximo Capítulo. Neste exemplo, também se nota que uma expressão ZF pode não ter qualquer expressão booleana.

No exemplo 2) deve-se notar a importância que tem a ordem em que os geradores são colocados. Se o gerador de **b** viesse antes do gerador de **a**, ocorreria um erro.

No exemplo 3) também deve ser observada a ordem em que os geradores foram colocados para que seja possível a geração correta dos triângulos retângulos.

A função **livrosEmprestados**, definida na seção 4.3, pode ser re-definida usando expressões ZF, da seguinte forma:

```
livrosEmprestados :: BancodeDados -> Pessoa -> [Livro]
livrosEmprestados db fulano = [liv | (pes, liv) <- db, pes == fulano]
```

Exercícios propostos:

1. Re-implemente as funções de atualização do Banco de Dados para a Biblioteca, feitas na seção 4.3, usando compreensão de listas, em vez de recursão explícita.

2. Como a função `membro :: [Int] -> Int -> Bool` pode ser definida usando compreensão de listas e um teste de igualdade?

Exemplo. Vamos agora mostrar um exemplo mais completo, baseado na referência [55], que mostra algumas das possibilidades que as compreensões oferecem. Seja um processador de texto simples que organiza um texto, indentando-o pela esquerda. Por exemplo, o texto

```
Maria gostava de bananas e
estava apaixonada
  por
Joaquim e      tomou      veneno para
morrer.
```

deve ser transformado em um texto mais organizado, ficando da seguinte forma:

```
Maria gostava de bananas e estava apaixonada
por Joaquim e tomou veneno para morrer.
```

Para isto, vamos construir algumas funções para realizar tarefas auxiliares. Inicialmente, devemos observar que uma palavra é uma sequência de caracteres que não tem espaços em branco dentro dela. Os espaços em branco são definidos da seguinte forma:

```
espacoEmBranco :: [Char]
espacoEmBranco = ['\n', '\t', ' ']
```

Vamos definir a função **pegaPalavra** que, quando aplicada a uma *string*, retira a primeira palavra desta *string* se esta *string* não iniciar com um espaço em branco. Assim **pegaPalavra** `"bicho besta"` = `"bicho"` e **pegaPalavra** `"bicho"` = `[]` porque a *string* `"bicho"` é iniciada com um caractere em branco.

Uma definição para ela pode ser feita, usando a função **pertence**, definida na seção 4.1, que verifica se um determinado caractere *a* pertence, ou não, a uma *string*:

```
pegaPalavra :: String -> String
pegaPalavra [] = []
pegaPalavra (a:x)
  |pertence a espacoEmBranco = []
  |otherwise = a : pegaPalavra x
```

Já a função **tiraPalavra**, quando aplicada a uma *string*, retira a primeira palavra da *string* e retorna a *string* restante, tendo como seu primeiro caractere o espaço em branco. Assim, **tiraPalavra** `"bicho feio"` = `"feio"`.

```
tiraPalavra :: String -> String
tiraPalavra [] = []
tiraPalavra (a : x)
  |pertence a espacoEmBranco = (a : x)
  |otherwise = tiraPalavra x
```

Está claro que é necessário construir uma função para retirar os espaços em branco da frente das palavras. Esta função será **tiraEspaces** que, aplicada a uma *string* iniciada com um ou mais espaços em branco, retorna outra *string* sem estes espaços em branco.


```
tiraEspacos :: String -> String
tiraEspacos [ ] = [ ]
tiraEspacos (a : x)
  | pertence a espacoEmBranco = tiraEspacos x
  | otherwise = (a : x)
```

Resta agora formalizar como uma *string*, **st**, deve ser transformada em uma lista de palavras, assumindo que **st** não seja iniciada por um espaço em branco. Assim,

- a primeira palavra desta lista de palavras será dada por **pegaPalavra st**,
- o restante da lista será construído dividindo-se a *string* que resulta da remoção da primeira palavra e dos espaços em branco que a seguem, ou seja, a nova transformação será feita sobre **tiraEspacos (tiraPalavra st)**.

```
type Palavra = String
```

```
divideEmPalavras :: String -> [Palavra]
divideEmPalavras st = divide (tiraEspacos st)
```

```
divide :: String -> [Palavra]
divide [ ] = [ ]
divide st = (pegaPalavra st) : divide (tiraEspacos (tiraPalavra st))
```

Vamos acompanhar a sequência de operações da aplicação **divideEmPalavras "bicho bom"**.

```
divideEmPalavras " bicho bom"
  = divide (tiraEspacos " bicho bom")
  = divide "bicho bom"
  = (pegaPalavra "bicho bom") : divide (tiraEspacos (tiraPalavra "bicho bom"))
  = "bicho" : divide (tiraEspaco " bom")
  = "bicho" : divide "bom"
  = "bicho" : (pegaPalavra "bom") : divide (tiraEspacos (tiraPalavra "bom"))
  = "bicho" : "bom" : divide (tiraEspacos [ ])
  = "bicho" : "bom" : divide [ ]
  = "bicho" : "bom" : [ ]
  = ["bicho", "bom"]
```

Deve ser observado que já existe a função **words**, definida no **Prelude**, que produz o mesmo resultado que a função **divideEmPalavras**. Esta definição foi aqui mostrada para fins de entendimento do usuário. Agora é necessário tomar uma lista de palavras e transformá-la em uma lista de linhas, onde cada linha é uma lista de palavras, com um tamanho máximo (o tamanho da linha). Para isto, vamos definir uma função que forme uma única linha com um tamanho determinado.

```
type Linha = [Palavra]
formaLinha :: Int -> [Palavra] -> Linha
```

Inicialmente, vamos admitir algumas premissas:

- Se a lista de palavras for vazia, a linha também será vazia.

- Se a primeira palavra disponível for **p**, ela fará parte da linha se existir vaga para ela na linha. O tamanho de **p**, **length p**, terá que ser menor ou igual ao tamanho da linha (**tam**). O restante da linha é construído a partir das palavras que restam, considerando agora uma linha de tamanho $tam - ((length\ p) + 1)$.
- Se a primeira palavra não se ajustar, a linha será vazia.

```
formaLinha tam [ ] = [ ]
formaLinha tam (p:ps)
  | length p <= tam = p : restoDaLinha
  | otherwise = [ ]
  where
    novoTam = tam - (length p + 1)
    restoDaLinha = formaLinha novoTam ps
```

Vamos acompanhar a sequência de aplicação da função:

```
formaLinha 20 ["Maria", "foi", "tomar", "banho", ...
= "Maria" : formaLinha 14 ["foi", "tomar", "banho", ...
= "Maria" : "foi": formaLinha 10 ["tomar", "banho" ...
= "Maria" : "foi": "tomar" : formaLinha 4 ["banho", ...
= "Maria" : "foi": "tomar" : [ ]
= ["Maria","foi","tomar"]
```

Precisamos criar uma função, **tiraLinha**, que receba como parâmetros um tamanho que uma linha deve ter e uma lista de palavras e retorne esta lista de palavras sem as palavras que fazem parte desta primeira linha. Esta função será deixada como exercício, no entanto, indicamos seu tipo.

```
tiraLinha :: Int -> [Palavra] -> [Palavra]
```

Agora é necessário juntar as coisas. Vamos construir uma função que transforme uma lista de palavras em uma lista de linhas. Primeiro ela forma a primeira linha, depois retira as palavras desta linha da lista original e aplica a função recursivamente à lista restante.

```
divideLinhas :: Int -> [Palavra] -> [Linha]
divideLinhas _ [ ] = [ ]
divideLinhas tamLin x =
  formaLinha tamLin x : divideLinhas (tiraLinha tamLin x)
```

Falta agora construir uma função que transforme uma *string* em uma lista de linhas, formando o novo texto indentado à esquerda.

```
preenche :: Int -> String -> [Linha]
preenche tam st = divideLinhas tam (divideEmPalavras st)
```

Finalmente deve-se juntar as linhas para que se tenha o novo texto, agora formando uma *string* indentada à esquerda. Será mostrada apenas o seu tipo, deixando sua definição como exercício.

```
juntaLinhas :: [Linha] -> String
```

Exercícios propostos:

1. Dê uma definição de uma função `juntaLinha :: Linha -> String` que transforma uma linha em uma forma imprimível. Por exemplo, `juntaLinha ["bicho", "bom"] = "bicho bom"`.
2. Use a função `juntaLinha` do exercício anterior para definir uma função `juntaLinhas :: [Linha] -> String` que junta linhas separadas por `'\n'`.
3. Modifique a função `juntaLinha` de forma que ela ajuste a linha ao tamanho `tam`, adicionando uma quantidade de espaços entre as palavras.
4. Defina uma função `estat :: String -> (Int, Int, Int)` que aplicada a um texto retorna o número de caracteres, palavras e linhas do texto. O final de uma linha é sinalizado pelo caractere *newline* (`'\n'`). Defina também uma função `novoestat :: String -> (Int, Int, Int)` que faz a mesma estatística sobre o texto, após ser ajustado.
5. Defina uma função `subst :: String -> String -> String -> String` de forma que `subst velhaSub novaSub st` faça a substituição da *sub-string* `velhaSub` pela *sub-string* `novaSub` em `st`. Por exemplo, `subst "muchtallHow much is that?" = "How tall is that?"` (Se a *sub-string* `velhaSub` não ocorrer em `st`, o resultado deve ser `st`).

5.5 Funções de alta ordem

Provavelmente, a maioria dos leitores já estejam familiarizados com a idéia de listas de listas, de dar nomes às listas ou de funções que admitem listas como seus parâmetros. O que talvez pareça estranho para alguns é a idéia de listas de funções ou de funções que retornam outras funções com resultados. Esta é uma característica importante das linguagens funcionais. A idéia central é a de que as funções são consideradas com os mesmos direitos que qualquer outro tipo de dado, dizendo-se, corriqueiramente, que “elas são cidadãs de primeira categoria”.

Vamos imaginar uma função **twice** que, quando aplicada a uma outra função, por exemplo, **f**, produza, como resultado, uma outra função que aplicada a seu argumento tenha o mesmo efeito da aplicação da função **f**, duas vezes. Assim,

twice f x = f (f x)

Como outro exemplo, vamos considerar a seguinte definição em Haskell:

```
let suc = soma 1
    soma x = somax
        where somax y = x + y
in suc 3
```

O resultado desta aplicação é 4. O efeito de **soma x** é criar uma função chamada **somax**, que adiciona **x** a algum número natural, no caso, **y**. Desta forma, **suc** é uma função que adiciona o número **1** a um número natural qualquer.

A partir destes exemplos, podemos caracterizar duas ferramentas importantes, a saber:

1. Um novo mecanismo para passar dois ou mais parâmetros para uma função. Apesar da função **soma** ter sido declarada com apenas um parâmetro, poderíamos chamá-la, por exemplo, como **soma 3 4**, que daria como resultado **7**. Isto significa que **soma 3** tem como resultado uma outra função que, aplicada a **4**, dá como resultado o valor **7**.

2. Um mecanismo de aplicação parcial de funções. Isto quer dizer que, se uma função for declarada com **n** parâmetros, podemos aplicá-la a **m** destes parâmetros, mesmo que **m** seja menor que **n**.

As funções de alta ordem são usadas de forma intensa em programas funcionais, permitindo que computações complexas sejam expressas de forma simples. Vejamos algumas destas funções, muito utilizadas na prática da programação funcional.

5.5.1 A função `map`

Um padrão de computação que é explorado como função de alta ordem envolve a criação de uma nova lista a partir de uma outra lista, onde cada elemento da nova lista tem o seu valor determinado através da aplicação de uma função a cada elemento da antiga lista. Suponhamos que se deseja transformar uma lista de nomes em uma lista de tuplas, em que cada nome da primeira lista deve ser transformado em uma tupla, onde o primeiro elemento seja o próprio nome e o segundo seja a quantidade de caracteres que esse nome tem. Para isso, vamos construir a função `listaTupla` de forma que

```
listaTupla ["Dunga","Constantino"] = [("Dunga",5),("Constantino",11)]
```

Antes vamos criar a função auxiliar `tuplaNum` que, aplicada a um nome, retorna a tupla formada pelo nome e pela quantidade de caracteres do nome.

```
tuplaNum :: [Char] -> ([Char], Int)
tuplaNum s = (s, length s)
```

Agora a função `listaTupla` pode ser definida da seguinte maneira:

```
listaTupla :: [String] -> [(String, Int)]
listaTupla [ ] = [ ]
listaTupla (s : xs) = (tuplaNum s) : listaTupla xs
```

Vamos agora, supor que se deseja transformar uma lista de inteiros em uma outra lista de inteiros, onde cada valor inteiro da primeira lista seja transformado em seu dobro, ou seja,

```
dobraLista [3, 2, 5] = [6, 4, 10]
```

Como na definição da função anterior, vamos construir a função auxiliar, `dobra`, da seguinte forma:

```
dobra :: Int -> Int
dobra n = 2*n

dobraLista :: [Int] -> [Int]
dobraLista [ ] = [ ]
dobraLista (n : x) = (dobra n) : dobraLista x
```

Analisando as definições de `listaTupla` e `dobraLista`, observamos a presença de um padrão que é o de aplicar uma mesma função a cada elemento da lista, ou seja, as duas funções percorrem as listas aplicando uma função a cada um de seus elementos.

Uma outra opção é construir uma função de alta ordem, destinada a realizar esta varredura, aplicando a mesma função a cada elemento da lista. A função a ser aplicada é passada como parâmetro para a função de alta ordem. Neste caso, a função de alta ordem é chamada de “mapeamento”, simbolizado pela função predefinida **map**, definida em Haskell da seguinte forma:

```
map :: (t -> u) -> [t] -> [u]
map f [ ] = [ ]
map f (a : x) = (f a) : (map f x)
```

Assim, as definições anteriores de **listaTupla** e **dobraLista** podem ser redefinidas da seguinte forma:

```
listaTupla :: [String] -> [(String, Int)]
listaTupla xs = map tuplaNum xs

dobraLista :: [Int] -> [Int]
dobraLista x = map dobra x
```

Vejamos mais alguns exemplos:

```
duplica, triplica :: Int -> Int
duplica n = 2 * n
triplica n = 3 * n

duplicaLista, triplicaLista :: [Int] -> [Int]
duplicaLista l = map duplica l
triplicaLista l = map triplica l

map duplica [4,5] = duplica 4 : map duplica [5]
  = 8 : map duplica [5]
  = 8 : (duplica 5 : map duplica [ ])
  = 8 : (10 : map duplica [ ])
  = 8 : (10 : [ ])
  = 8 : [10]
  = [8,10]
```

A utilização de funções de alta ordem se justifica, tendo em vista as seguintes premissas:

- É mais fácil entender a definição, porque torna claro que se trata de um mapeamento, por causa da função **map**. Precisa apenas entender a função mapeada.
- É mais fácil modificar as definições das funções a serem aplicadas, se isto for necessário.
- É mais fácil reutilizar as definições.

Exemplo. As funções de análise de vendas, mostradas no Capítulo anterior, foram definidas para analisar uma função fixa: **vendas**. Agora, a função **totalVendas** pode ser dada por

```
totalVendas n = somaLista (map venda [0..n])
```

Muitas outras funções podem ser definidas usando **map**. Por exemplo,

```
somaQuad :: Int -> Int
somaQuad n = somaLista (map quad [1..n])

quad :: Int -> Int
quad n = n * n
```

onde a função **somaLista** foi definida no início deste Capítulo.

Exercícios:

1. Dê definições de funções que tome uma lista de inteiros l e
 - retorne a lista dos quadrados dos elementos de l ,
 - retorne a soma dos quadrados dos elementos de l e
 - verifique se todos os elementos da lista são, ou não, positivos.
2. Defina funções que
 - dê o valor mínimo de uma função f aplicada a uma lista de 0 a n ,
 - teste se os valores de f sobre as entradas 0 a n são todas iguais.
 - teste se todos os valores de f aplicada às entradas de 0 a n são maiores ou iguais a **zero** e
 - teste se os valores $f\ 0$, $f\ 1$ até $f\ n$ estão em ordem crescente.
3. Estabeleça o tipo e defina uma função **twice** que toma uma função de inteiros para inteiros e um inteiro e retorna a função aplicada à entrada três vezes. Por exemplo, tendo a função **triplica** e o inteiro **4** como entradas, o resultado é **108**.
4. Dê o tipo e defina uma função **iter** de forma que $\text{iter } n\ f\ x = f\ (f\ (f\ \dots (f\ x)\ \dots))$, onde f ocorre n vezes no lado direito da equação. Por exemplo, devemos ter: $\text{iter } 3\ f\ x = f\ (f\ (f\ x))$ e $\text{iter } 0\ f\ x = x$.
5. Usando **iter** e **duplica**, defina uma função que aplicada a n retorne $2n$.

5.5.2 Funções anônimas

Já foi visto, e de forma enfática, que, nas linguagens funcionais, as funções podem ser usadas como parâmetros para outras funções. No entanto, seria um desperdício definir uma função que só pudesse ser utilizada como parâmetro para outra função, ou seja, a função só fosse utilizada neste caso e em nenhum outro mais. No caso da seção anterior, as funções **dobra** e **tuplaNum**, possivelmente, só sejam utilizadas como argumentos das funções **dobraLista** e **listaTupla**.

Uma forma de declarar funções para serem utilizadas apenas localmente é usar a cláusula **where**. Por exemplo, dado um inteiro n , vamos definir uma função que retorne uma outra função de inteiro para inteiro que adiciona n a seu argumento.

```
somaNum :: Int -> (Int -> Int)
somaNum n = h where h m = n + m
```

Pode-se observar que quando se necessita especificar um valor, normalmente, se declara um identificador para isto. Esta também tem sido a forma utilizada com as funções, ou seja, declara-se uma função com um nome e, quando a função for necessária, ela é referenciada pelo seu nome.

Uma alternativa, possível em Haskell, consiste em declarar uma função apenas no ponto de chamada, sem ligar esta função a qualquer identificador. Estas funções são chamadas de “funções anônimas” e se baseiam na notação do λ -cálculo, visto no Capítulo 2. Esta forma de definição de funções é mais compacta e mais eficiente. A função **somaNum**, definida anteriormente usando a cláusula **where**, pode ser escrita, de forma anônima, da seguinte forma:

```
\m -> n + m
```

As funções **dobraLista** e **listaTupla** também podem ser definidas da seguinte forma:

```
dobraLista l = map (\n -> 2*n) l
listaTupla ls = map (\s -> (s, length s)) ls
```

Vamos agora analisar a sintaxe de uma função anônima. Uma definição anônima é dividida em duas partes: uma antes da flexa e a outra depois dela. Estas duas partes têm as seguintes interpretações:

- antes da flexa vêm os argumentos e
- depois da flexa vem o resultado.

A barra invertida no início da definição indica que se trata de uma função anônima. A \backslash é o caractere mais parecido com a letra grega λ , usada no λ -cálculo.

Vejamos a função **comp2**, mostrada graficamente na Figura 5.1. Na realidade, trata-se de uma função **g** que recebe como entrada dois argumentos, no caso **f x** e **f y**, que são os resultados das aplicações da função **f** aos argumentos **x** e **y**, ou seja **g (f x) (f y)**.

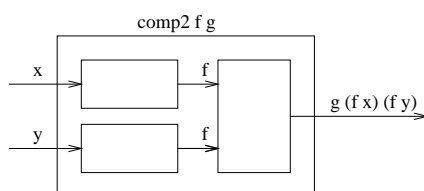


Figura 5.1: Forma gráfica da função comp2.

A definição de **comp2** é

```
comp2 :: (a -> b) -> (b -> b -> c) -> (a -> a -> c)
comp2 f g = (\x y -> g (f x) (f y))
```

Para se adicionar os quadrados de 5 e 6, podemos escrever **comp2 quad soma 5 6** onde **quad** e **soma** têm significados óbvios.

De forma geral, sendo **f** definida por **f x y z = resultado** então **f** pode ser definida anonimamente por

```
\x y z -> resultado
```

5.5.3 As funções fold e foldr

Uma outra função de alta ordem, também de grande utilização em aplicações funcionais, é a função **fold**, usada na combinação de itens (*folding*). Ela toma como argumentos uma função de dois argumentos e a aplica aos elementos de uma lista. O resultado é um elemento do tipo dos elementos da lista. Vamos ver sua definição formal e exemplos de sua aplicação.

```
fold :: (t -> t -> t) -> [t] -> t
fold f [a] = a
fold f (a:b:x) = f a (fold f (b:x))
```

Exemplos:

```
fold (||) [False, True, False] = (||) False (fold (||) [True, False])
                                = (||) False ((||) True (fold (||) [False]))
                                = (||) False ((||) True False)
                                = (||) False True
                                = True
```

```
fold (++) ["Chico", " Afonso", "!"] = "Chico Afonso!"
```

```
fold (*) [1..6] = 720
```

No entanto, existe um pequeno problema na definição de **fold**. Se ela for aplicada a uma função e uma lista vazia ocorrerá um erro. Para resolver este impasse, foi definida, em Haskell, uma outra função para substituir **fold**, onde este tipo de problema seja resolvido. Esta é a função **foldr**, definida da seguinte forma:

```
foldr :: (t -> u -> u) -> u -> [t] -> u
foldr f s [ ] = s
foldr f s (a : x) = f a (foldr f s x)
```

Vamos verificar como algumas funções são definidas usando **foldr**.

Exemplos:

```
concat :: [[t]] -> [t]
concat xs = foldr (++) [ ] xs

and :: [Bool] -> Bool
and bs = foldr (&&) True bs

rev :: [t] -> [t]
rev l = foldr stick [ ] l

stick :: t -> [t] -> [t]
stick a x = x ++ [a]
```

5.5.4 A função filter

A função **filter** é uma outra função de alta ordem, predefinida em todas as linguagens funcionais, de bastante utilização. Resumidamente, ela escolhe dentre os elementos de uma lista, aqueles que têm uma determinada propriedade. Vejamos alguns exemplos:

1. **filter ehPar [2,3,4] = [2,4]**.
2. Um número natural é perfeito se a soma de seus divisores, incluindo o número 1, for o próprio número. Por exemplo, 6 é o primeiro número natural perfeito, porque $6 = 1+2+3$. Vamos definir uma função que mostra os números perfeitos entre **0** e **m**.

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

fatores :: Int -> [Int]
fatores n = filter (divide n) [1..(n `div` 2)]

perfeito :: Int -> Bool
perfeito n = sum (fatores n) == n
```



```
lista_perfeitos :: Int -> [Int]
lista_perfeitos m = filter perfeito [0..m]
```

E se quisermos os primeiros **m** números perfeitos?

3. A lista de todos os números pares maiores que **113** e menores ou iguais a **1000**, que sejam perfeitos é dada por:

```
filter perfeito [y | y <- [114 .. 1000], ehPar y]
```

4. Seleccionando elementos: **filter digits "18 Março 1958" = "181958"**

A definição formal de **filter** é:

```
filter :: (t -> Bool) -> [t] -> [t]
filter p [ ] = [ ]
filter p (a : x)           ou   filter p l = [a | a <- l, p a]
    | p a = a : filter p x
    | otherwise = filter p x
```

5.6 Polimorfismo

Uma característica muito importante das linguagens funcionais é que suas definições podem ser polimórficas, um mecanismo que aumenta o poder de expressividade de qualquer linguagem. Polimorfismo é uma das características responsáveis pela alta produtividade de *software*, proporcionada pelo aumento da reusabilidade.

Polimorfismo é a capacidade de aplicar uma mesma função a vários tipos de dados, representados por um tipo variável. Não deve ser confundido com sobrecarga, denominada por muitos pesquisadores como polimorfismo *ad hoc*, que consiste na aplicação de várias funções com o mesmo nome a vários tipos de dados. Haskell permite os dois tipos de polimorfismo, sendo que a sobrecarga é feita através de um mecanismo engenhoso chamado de **type class**, um tema a ser estudado no próximo Capítulo.

A função **length** é predefinida em Haskell, da seguinte maneira:

```
length :: [t] -> Int
length [ ] = 0
length (a : x) = 1 + length x
```

Esta função tem um tipo polimórfico porque pode ser aplicada a qualquer tipo de lista homogênea. Em sua definição não existe qualquer operação que exija que a lista parâmetro seja de algum tipo particular. A única operação que esta função faz é contar os elementos de uma lista, seja ela de que tipo for.

Já a função

```
quadrado :: Int -> Int
quadrado x = x * x
```

não pode ser polimórfica porque só é aplicável a elementos onde a operação de multiplicação (*) seja possível. Por exemplo, não pode ser aplicada a *strings*, nem a valores booleanos.

5.6.1 Tipos variáveis

Quando uma função tem um tipo envolvendo um ou mais tipos variáveis, diz-se que ela tem um tipo polimórfico. Por exemplo, já vimos anteriormente que a lista vazia é um elemento de qualquer tipo de lista, ou seja, `[]` está na interseção dos tipos `[Int]`, `[Bool]` ou `[Char]`, etc. Para explicitar esta característica denota-se que `[] :: [t]`, sendo `t` é uma variável que pode assumir qualquer tipo. Assim, `cons` tem o tipo polimórfico `(:) :: t -> [t] -> [t]`.

As seguintes funções, algumas já definidas anteriormente, têm os tipos:

```
length :: [t] -> Int
(++ ) :: [t] -> [t] -> [t]
rev :: [t] -> [t]
id :: t -> t
zip :: [t] -> [u] -> [(t, u)]
```

5.6.2 O tipo mais geral

Alguma dificuldade pode surgir nas definições dos tipos das funções quando elas envolvem tipos variáveis, uma vez que podem existir muitas instâncias de um mesmo tipo variável. Para resolver este dilema, é necessário que o tipo da função seja o tipo mais geral possível, que é definido da seguinte forma:

Definição. Um tipo `w` de uma função `f` é “o tipo mais geral” de `f` se todos os tipos de `f` forem instâncias de `w`.

A partir desta definição pode-se observar que o tipo `[t] -> [t] -> [(t, t)]` é uma instância do tipo da função `zip`, mas não é o tipo mais geral, porque `[Int] -> [Bool] -> [(Int, Bool)]` é um tipo para `zip`, mas não é uma instância de `[t] -> [t] -> [(t, t)]`.

Exemplos de algumas funções polimórficas:

```
rep :: Int -> t -> [t]
rep 0 ch = []
rep n ch = ch : rep (n - 1) ch

fst :: (t, u) -> t snd :: (t, u) -> u
fst (x, _) = x snd (_, y) = y

head :: [t] -> t tail :: [t] -> [t]
head (a : _) = a tail (_ : x) = x
```

Mas qual a vantagem de se ter polimorfismo? A resposta vem da Engenharia de Software e se baseia nos seguintes fatos:

- definições mais gerais implicam em maior chance de reutilização e
- em linguagens não polimórficas, as funções devem ser redefinidas para cada novo tipo. Isto implica em ineficiência e insegurança.

Exercícios:

1. Defina uma função `concat` onde `concat [e1, ..., ek] = e1 ++ ... ++ ek`. Qual o tipo de `concat`?

2. Defina uma função **unZip** que transforma uma lista de pares em um par de listas. Qual o seu tipo?
3. Defina uma função **last** :: $[t] \rightarrow t$ que retorna o último elemento de uma lista não vazia. Defina também **init** :: $[t] \rightarrow [t]$ que retorna todos os elementos de uma lista com exceção de seu último elemento.
4. Defina funções **tome**, **tire** :: $\text{Int} \rightarrow [t] \rightarrow [t]$ onde **tome** **n** **l** retorna os **n** primeiros elementos da lista **l** e **tire** **n** **l** retira os **n** primeiros elementos da lista **l**.

5.7 Indução estrutural

Já vimos formas de se provar propriedades em Haskell. No entanto, quando estas propriedades envolvem listas, existe uma forma específica de serem provadas, que é a *indução estrutural*. Pode-se dizer que indução estrutural é o método de indução matemática aplicado às listas finitas. As listas infinitas não são tratadas, uma vez que elas não são estruturas modeláveis na Computação.

Os computadores são máquinas com memórias limitadas, apesar de poderem ser grandes, mas são finitas. Apesar de alguns autores se referirem às listas infinitas, o que realmente eles se referem são as listas potencialmente infinitas, que são implementadas em Haskell através de um mecanismo de avaliação *lazy*, um tópico a ser visto mais adiante. Deve ser lembrado que a lista vazia, `[]`, é uma lista finita e a lista não vazia, `(a:x)`, é uma lista finita, se a lista **x** for finita.

Esquema de prova

O esquema de provas mostrado a seguir é devido a Simon Thompson [55]. Apesar de muito formal, ele deve ser seguido, principalmente, por iniciantes, que ainda não têm experiência com provas de programas. Para estes usuários, é recomendável utilizá-lo como forma de treinamento. Muitas pessoas tendem a querer chegar à conclusão de uma prova forçando situações, sem a argumentação adequada e este tipo de vício há que ser evitado, a qualquer custo. A falta de domínio nesta área pode levar o usuário a conclusões equivocadas.

Outro erro, comumente cometido por algumas pessoas não afeitas a provas matemáticas, consiste em realizar provas sem se importar com as conclusões das mesmas. A conclusão é parte integrante do esquema de provas e, portanto, indispensável. Ela é o objetivo da prova. Sem ela não existe razão para todo um esforço a ser despendido nas fases anteriores. A conclusão representa o desfecho de uma prova e representa a formalização de uma proposição que passa a ser verdadeira e pode ser utilizada em qualquer etapa de uma computação.

O esquema de provas deve se constituir nos seguintes estágios:

- Estágio 0: escrever o objetivo da prova informalmente,
- Estágio 1: escrever o objetivo da prova formalmente,
- Estágio 2: escrever os sub-objetivos da prova por indução:
 $P([])$ e
 $P(a : x)$, assumindo $P(x)$
- Estágio 3: provar $P([])$
- Estágio 4: provar $P(a : x)$, lembrando que PODE e DEVE usar $P(x)$.

Vejam agora, dois exemplos completos de esquemas de provas que envolvem todos os estágios enumerados anteriormente.

Exemplo 1. Dadas as definições a seguir:

$$\text{somaLista } [] = 0 \quad (1)$$

$$\text{somaLista } (a : x) = a + \text{somaLista } x \quad (2)$$

$$\text{dobra } [] = [] \quad (3)$$

$$\text{dobra } (a : x) = (2 * a) : \text{dobra } x \quad (4)$$

Provar que $2 * (\text{somaLista } l) = \text{somaLista } (\text{dobra } l)$. Vejamos o esquema de prova utilizando a metodololgia indicada.

- **Estágio 0:** o dobro da soma dos elementos de uma lista é igual à soma dos elementos da lista formada pelos dobros dos elementos da lista anterior.

- **Estágio 1:** $2 * \text{somaLista } x = \text{somaLista } (\text{dobra } x)$ (5)

- **Estágio 2:**

$$- 2 * \text{somaLista } [] = \text{sumList } (\text{dobra } []) \quad (6)$$

$$- 2 * \text{somaLista } (\text{dobra } (a : x)) = \text{somaLista } (\text{dobra } (a : x)) \quad (7)$$

$$\text{assumindo que } 2 * \text{somaLista } x = \text{somaLista } (\text{dobra } x) \quad (8)$$

- **Estágio 3:** Caso base:

lado esquerdo do caso base:		lado direito do caso base	
$2 * \text{somaLista } []$		$\text{somaLista } (\text{dobra } [])$	
$= 2 * 0$	por (1)	$= \text{somaLista } []$	por (3)
$= 0$	pela aritmética	$= 0$	por (1)

Assim, a assertiva (6) é válida.

- **Estágio 4:** Passo indutivo:

lado esquerdo do passo indutivo:	
$2 * \text{somaLista } (a : x)$	
$= 2 * (a + \text{somaLista } x)$	por (2).

lado direito do passo indutivo:	
$\text{somaLista } (\text{dobra } (a : x))$	
$= \text{somaLista } (2 * a : \text{dobra } x)$	por (4)
$= 2 * a + \text{somaLista } (\text{dobra } x)$	por (2)
$= 2 * a + 2 * \text{somaLista } x$	pela hipótese de indução
$= 2 * (a + \text{somaLista } x)$	pela distributividade de *.

Assim, a assertiva (7) é válida.

Conclusão: como a assertiva (5) é válida para o caso base e para o passo indutivo, então ela é válida para todas as listas finitas.

Exemplo 2. Associatividade de **append**: $x ++ (y ++ z) = (x ++ y) ++ z$.

- **Estágio 0:** a função $++$ é associativa.

- **Estágio 1:** Dadas as definições:

$$[] ++ v = v \quad (1)$$

$$(a : x) ++ v = a : (x ++ v) \quad (2)$$

Provar que $x ++ (y ++ z) = (x ++ y) ++ z$

• **Estágio 2:**

- caso base: $[] ++ (y ++ z) = ([] ++ y) ++ z$ (3)
- passo indutivo: $(a : x) ++ (y ++ z) = ((a : x) ++ y) ++ z$ (4) assumindo que $x ++ (y ++ z) = (x ++ y) ++ z$ (5)

• **Estágio 3:**

caso base: lado esquerdo	caso base: lado direito
$[] ++ (y ++ z)$	$([] ++ y) ++ z$
$= y ++ z$	$= y ++ z$
por (1)	por (1)

Como o lado esquerdo e o lado direito são iguais, então a propriedade é válida para o caso base.

• **Estágio 4:**

passo indutivo: lado esquerdo	
$(a : x) ++ (y ++ z)$	
$= a : (x ++ (y ++ z))$	por (2)
passo indutivo: lado direito	
$((a : x) ++ y) ++ z$	
$= (a : (x ++ y)) ++ z$	por (2)
$= a : ((x ++ y) ++ z)$	por (2)
$= a : (x ++ (y ++ z))$	pela hipótese de indução.

Como o lado esquerdo e o lado direito são iguais, então a propriedade é válida para o passo indutivo.

Conclusão: como a assertiva é válida para o caso base e para o passo indutivo, então ela é verdadeira para todas as listas finitas.

Exercícios:

1. Prove que, para todas as listas finitas x , $x ++ [] = x$.
2. Tente provar que $x ++ (y ++ z) = (x ++ y) ++ z$ usando indução estrutural sobre z . Há alguma coisa esquisita com esta prova? O quê?
3. Prove que, para todas as listas finitas x e y , **somaLista** $(x ++ y) = \text{somaLista } x + \text{somaLista } y$ e que **somaLista** $(x ++ y) = \text{somaLista } (y ++ x)$.
4. Mostre que, para todas as listas finitas x e y , **dobra** $(x ++ y) = \text{dobra } x ++ \text{dobra } y$ e **length** $(x ++ y) = \text{length } x + \text{length } y$.
5. Prove por indução sobre x que, para todas as listas finitas x , **somaLista** $(x ++ (a : y)) = a + \text{somaLista } (x ++ y)$.
6. Prove, usando ou não o exercício anterior, que para todas as listas finitas x , **somaLista** $(\text{double } x) = \text{somaLista } (x ++ x)$.

5.8 Composição de funções

Uma forma simples de estruturar um programa é construí-lo em etapas, uma após outra, onde cada uma delas pode ser definida separadamente. Em programação funcional, isto é feito através

da composição de funções, uma propriedade matemática só implementada nestes tipos de linguagens e que aumenta, enormemente, a expressividade do programador. Vamos apresentar dois tipos de composição que podem ser aplicadas na construção de programas funcionais.

5.8.1 Composição normal de funções

A expressão $f(g(x))$, normalmente, é escrita pelos matemáticos como $(f.g)(x)$, onde o $.$ (ponto) é o operador de composição de funções. Esta notação é importante porque separa a parte das funções da parte dos argumentos. O operador de composição é um tipo de função, cujos argumentos são duas funções e o resultado é também uma função.

Para facilitar nosso entendimento, vamos retornar à função **divideEmPalavras**, definida anteriormente da seguinte forma:

```
divideEmPalavras :: String -> [Palavra]
divideEmPalavras st = divide (tiraEspacos st)
```

Esta função pode ser reescrita como: **divideEmPalavras st = divide . tiraEspacos st**.

Como mais um exemplo, vamos considerar a definição

```
let quad = compoe (quadrado, sucessor)
    quadrado x = x * x
    sucessor x = x + 1
    compoe (f,g) = h where h x = f(g(x))
in quad 3
```

A resposta a esta aplicação é 16. O interesse maior aqui está na definição da função **compoe**. Ela toma um par de parâmetros, **f** e **g** (ambos funções) e retorna uma outra função, **h**, cujo efeito de sua aplicação é a composição das funções **f** e **g**. Esta forma de codificação torna a composição explícita sem a necessidade de aplicar cada lado da igualdade a um argumento.

Como é sabido, nem todo par de funções pode ser composto. O tipo da saída da função **g** tem de ser o mesmo tipo da entrada da função **f**. O tipo de **.** é: $(.) :: (u \rightarrow v) \rightarrow (t \rightarrow u) \rightarrow (t \rightarrow v)$.

A composição é associativa, ou seja, $f . (g . h) = (f . g) . h$, que deve ser interpretado como “faça **h**, depois faça **g** e finalmente faça **f**”.

Exercício resolvido (leitura de números)

Algumas vezes é necessário transformar números em **strings**, como é o caso do preenchimento de cheques, onde o número 1384 deve ser traduzido para “*um mil, trezentos e oitenta e quatro*”. Na solução deste problema, vamos considerar inicialmente números menores que 1000000 que podem ser escritos em até 6 dígitos, distribuídos em classes, inicialmente as unidades entre zero e dez, depois os números entre dez e vinte, em seguida os números entre vinte e cem e, finalmente os números maiores ou iguais a cem.

```
unidades, dezvinte, dezenas, centenas :: [String]
unidades = ["um", "dois", "tres", "quatro", "cinco", "seis", "sete", "oito",
            "nove"]
dezvinte = ["dez", "onze", "doze", "treze", "quatorze", "quinze", "dezesesseis",
            "dezesesete", "dezoito", "dezenove"]
```

```
dezenas = ["vinte", "trinta", "quarenta", "cinquenta", "sessenta", "setenta",
           "oitenta", "noventa"]
centenas = ["cento", "duzentos", "trezentos", "quatrocentos", "quinhentos",
            "seiscentos", "setecentos", "oitocentos", "novecentos"]
```

Considerando um número n , ($0 \leq n < 100$) verifica-se que eles são compostos por até dois dígitos. Neste caso, vamos dividi-los em duas partes, ou seja, vamos separar os dígitos.

```
digitos2 :: Int -> (Int, Int)
digitos2 n = (div n 10, mod n 10)
```

Com estes dois dígitos separados, vamos agora transformá-los em strings.

```
combina2 :: (Int, Int) -> String
combina2 (0, u) = unidades !! (u-1)
combina2 (1, u) = dezvinte !! u
combina2 (t, 0) = dezenas !! (t-2)
combina2 (t, u) = dezenas !! (t-2) ++ " e " ++ unidades !! (u-1)
```

A leitura final será uma composição destas duas funções, ou seja:

```
converte2 :: Int -> String
converte2 = combina2.digitos2
```

Nesta segunda etapa, serão considerados os números compostos com 3 dígitos. Neste caso, a separação destes dígitos se dá também em duas partes, onde a primeira é a divisão inteira do número por 100, o dígito representante das centenas e o resto da divisão do número por 100, que é obrigatoriamente um número de 2 dígitos que pode ser traduzido como descrito anteriormente. Estas duas funções devem ser compostas para poderem representar a solução final.

```
digitos3 :: Int -> (Int, Int)
digitos3 n = (div n 100, mod n 100)

combina3 :: (Int, Int) -> String
combina3 (0, t) = converte2 t
combina3 (h, 0) = centenas !! (h-1)
combina3 (h, t) = centenas !! (h-1) ++ " e " ++ converte2 t

converte3 :: Int -> String
converte3 = combina3.digitos3
```

Resta agora a última etapa, destinada aos números maiores ou iguais a 1000 e menores que 1000000. Neste caso, o número é dividido também em duas partes, onde a primeira é a quantidade de milhares e a segunda é um número de 3 dígitos que podem ser traduzidos com as funções definidas anteriormente.

```
digitos6 :: Int -> (Int, Int)
digitos6 n = (div n 1000, mod n 1000)

combina6 :: (Int, Int) -> String
combina6 (0, h) = converte3 h
```

```

combina6 (m, 0) = converte3 (m) ++ " mil"
combina6 (m, h) = converte3 (m) ++ " mil" ++ link (h) ++ converte3 (h)

converte6 :: Int -> String
converte6 = combina6.digitos6

link :: Int -> String
link h = if h < 100 then " e " else ", "

```

Finalmente, vamos criar uma função que englobe todos estes casos, incluindo os valores 0 (zero) e 100 (cem) por serem números não descritos anteriormente. A função **converte** foi definida para este objetivo.

```

converte :: Int -> String
converte n
  | n == 0    = "Zero"
  | (n>0) && (n<100) = converte2 n
  | n==100    = "Cem"
  | (n>100)&&(n<1000) = converte3 n
  | (n>=1000)&&(n<=1000000) = converte6 n

```

5.8.2 Composição avançada

A ordem em **f . g** é importante (faça **g** e depois **f**). Podemos fazer a composição ter o sentido das ações propostas, usando a composição avançada. Deve ser lembrado que esta forma é apenas para fins de apresentação, uma vez que a definição, como será vista, é feita em função da composição e nada aumenta em expressividade. A composição avançada será denotada por **>.>** indicando a sequencialização da aplicação das funções, uma vez que, na composição, ela é realizada na ordem inversa de aparência no texto. A definição de **>.>** é feita da seguinte forma:

```

infixl 9 >.>
(>.>) :: (t -> u) -> (u -> v) -> (t -> v)
g >.> f = f . g

```

A função **divideEmPalavras**, definida anteriormente, pode ser re-definida usando composição avançada, da seguinte forma:

```

divideEmPalavaras = tiraEspacos >.> divide

```

Deve-se ter o cuidado de observar que **f . g x** é diferente de **(f . g) x** porque a aplicação de funções tem prioridade sobre a composição. Por exemplo, **succ . succ 1** resultará em erro porque **succ 1** será realizada primeiro e retornará um inteiro (**2**), fazendo com que a composição do primeiro **succ** seja feita com um número inteiro e não com uma função. Neste caso, os parênteses devem ser utilizados para resolver ambiguidades.

5.8.3 Esquema de provas usando composição

Seja a função **twice f = f . f**. Assim

```

(twice succ) 12
= (succ . succ) 12    --pela definicao de twice

```



```
= succ (succ 12)      --pela definicao de .
= succ 13 = 14
```

Pode-se generalizar **twice**, indicando um parâmetro que informe quantas vezes a função deve ser composta com ela própria:

```
iter :: Int -> (t -> t) -> (t -> t)
iter 0 f = id
iter n f = f >.> iter (n - 1) f
```

Por exemplo, podemos definir **2n** como **iter n duplica** e vamos mostrar a sequência de operações para **n=2**.

```
iter 2 duplica = duplica >.> iter 1 duplica
               = duplica >.> (duplica >.> iter 0 f)
               = duplica >.> (duplica >.> id)
               = duplica >.> duplica
               = twice duplica
```

Nesta definição, usamos o fato de que **f . id = f**. Estamos tratando de uma nova espécie de igualdade, que é a igualdade de duas funções. Mas como isto pode ser feito? Para isto, devemos examinar como os dois lados se comportam quando os aplicamos a um mesmo argumento **x**. Então,

```
(f . id) x
  = f (id x)      pela definição de composição
  = f x           pela definição de id.
```

Isto significa que para um argumento **x**, qualquer, as duas funções se comportam exatamente da mesma forma.

Vamos fazer uma pequena discussão sobre o que significa a igualdade entre duas funções. Existem dois princípios que devem ser observados quando nos referimos à igualdade de funções. São eles:

- **Princípio da extensionalidade:** “Duas funções, **f** e **g**, são iguais se elas produzirem exatamente os mesmos resultados para os mesmos argumentos”.
- **Princípio da intencionalidade:** “Duas funções, **f** e **g** são iguais se tiverem as mesmas definições”.

Se estivermos interessados nos resultados de nossos programas, tudo o que nos interessa são os valores dados pelas funções e não como estes valores são encontrados. Em Haskell, devemos usar extensionalidade quando estivermos interessados no comportamento das funções. Se estivermos interessados na eficiência ou outros aspectos de desempenho de programas devemos usar a intencionalidade.

Exercícios:

1. Mostre que a composição de funções é associativa, ou seja, $\forall f, g \text{ e } h, f \cdot (g \cdot h) = (f \cdot g) \cdot h$
2. Prove que $\forall n \in \mathbb{Z}^+, \text{iter } n \text{ id} = \text{id}$.

3. Duas funções **f** e **g** são inversas se **f . g = id** e **g . f = id**. Prove que as funções **curry** e **uncurry**, definidas a seguir, são inversas.

```
curry :: ((t, u) -> v) -> (t -> u -> v)
curry f (a, b) = f a b
```

```
uncurry :: (t -> u -> v) -> ((t, u) -> v)
uncurry g a b = g (a, b)
```

Exemplo: Seja a definição de **map** e da composição de funções dadas a seguir. Mostre que **map (f . g) x = (map f . map g) x**.

```
map f [ ] = [ ] (1)
```

```
map f (a : x) = f a : map f x (2)
```

```
(f . g) x = f (g x) (3)
```

Esquema de prova:

Caso base: a lista vazia, []:

Lado esquerdo	Lado direito
map (f . g) [] = [] por (1)	(map f . map g) []
	= map f (map g []) por (3)
	= map f [] por (1)
	= [] por (1)

Passo indutivo: a lista não vazia, (a : x):

Lado esquerdo	Lado direito
map (f . g) (a : x)	(map f . map g)(a:x)
= (f . g) a : map (f . g) x (2)	= map f (map g (a:x)) (3)
= f (g a) : map (f . g) x (3)	= map f ((g a) : (map g x)) (2)
= f (g a) : (map f . map g) x (hi)	= f (g a) : (map f (map g) x) (3)
= f (g a) : (map f . map g) x	

Conclusão. Como a propriedade é válida para a lista vazia e para a lista não vazia, então ela é válida para qualquer lista homogênea finita.

Exercício: Prove que para todas as listas finitas **l** e funções **f**, **concat (map (map f) l) = map f (concat l)**.

5.9 Aplicação parcial

Uma característica importante de Haskell é que proporciona uma forma elegante e poderosa de construção de funções é a avaliação parcial que consiste na aplicação de uma função a menos argumentos do que ela realmente precisa. Por exemplo, seja a função **multiplica** que retorna o produto de seus argumentos:

```
multiplica :: Int -> Int -> Int
multiplica a b = a * b
```

Esta função foi declarada para ser usada com dois argumentos. No entanto, ela pode ser chamada como **multiplica 2**. Esta aplicação retorna uma outra função que, aplicada a um argumento **b**, retorna o valor **2*b**. Esta característica é o resultado do seguinte princípio em

Haskell: “uma função com n argumentos pode ser aplicada a r argumentos, onde $r < n$ ”. Como exemplo, a função **dobraLista** pode ser definida da seguinte forma:

```
dobraLista :: [Int] -> [Int]
dobraLista = map (multiplica 2)
```

onde **multiplica 2** é uma função de inteiro para inteiro, a aplicação de **multiplica** a um de seus argumentos, **2**, em vez de ser aplicada aos dois. **map (multiplica 2)** é uma função do tipo `[Int] -> [Int]`, dada pela aplicação parcial de **map**.

Como é determinado o tipo de uma aplicação parcial? Pela regra do cancelamento: “se uma função **f** tem o tipo $t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n \rightarrow t$ e é aplicada aos argumentos $e_1 :: t_1, e_2 :: t_2, \dots, e_k :: t_k$, com $k \leq n$, então o tipo do resultado é dado pelo cancelamento dos tipos t_1 até t_k , dando o tipo $t_{k+1} \rightarrow t_{k+2} \rightarrow \dots \rightarrow t_n \rightarrow t$.”

Por exemplo,

```
multiplica 2 :: Int -> Int
multiplica 2 3 :: Int
```

```
dobraLista :: [Int] -> [Int]
dobraLista [2,3,5] :: [Int]
```

Mas afinal, quantos argumentos tem realmente uma função em Haskell? Pelo exposto, a resposta correta a esta questão é **1** (UM). Isto é curificação. Lembra-se do λ -cálculo?

Isto significa que uma função do tipo `Int -> Int -> Int` é do mesmo tipo que **Int -> Int -> Int**. Neste último caso, está explícito que esta função pode ser aplicada a um argumento inteiro e o seu resultado é uma outra função que recebe um inteiro e retorna outro inteiro. Exemplificando,

```
multiplica :: Int -> Int -> Int ou multiplica :: Int -> (Int -> Int)
multiplica 4 :: Int -> Int
multiplica 4 5 :: Int
```

Estes resultados permitem concluir que, em Haskell, a aplicação de função é associativa à esquerda, ou seja, **f a b = (f a) b** enquanto a flexa é associativa pela direita, ou seja, **t -> u -> v = t -> (u -> v)**.

5.9.1 Seção de operadores

Uma decorrência direta das aplicações parciais que representa uma ferramenta poderosa e elegante em algumas linguagens funcionais e, em particular, em Haskell, são as seções de operadores. As seções são operações parciais, normalmente relacionadas com as operações aritméticas. Nas aplicações, elas são colocadas entre parênteses. Por exemplo,

- **(+2)** é a função que adiciona algum argumento a 2,
- **(2+)** a função que adiciona 2 a algum argumento,
- **(>2)** a função que retorna True se um inteiro for maior que 2,
- **(3:)** a função que coloca o inteiro 3 na cabeça de uma lista,

- `(++“\n”)` a função que coloca o caractere `'\n'` ao final de uma *string*.

Uma seção de um operador **op** coloca o argumento no lado que completa a aplicação. Por exemplo, `(op a) b = b op a` e `(a op) b = a op b`.

Exemplos:

- `map (+1) >.> filter (>0)`.
- `dobra = map (*2)`.
- `pegaPares = filter ((==0) . ('mod' 2))`.
- A função **inquilinos**, definida no início deste Capítulo, normalmente, é escrita da seguinte forma:

```
inquilinos db pes = map snd (filter ehPes db)
  where
    ehPes (p, b) = (p == pes)
```

No entanto, ela pode ser escrita em forma de seção da seguinte maneira, o que a torna muito mais elegante.

```
inquilinos db pes = map snd (filter ((==pes) . fst) db)
```

Exercício resolvido (criação de um Index).

Este é mais um exemplo que mostra o poder de expressividade de Haskell. Trata-se da criação de um índice remissivo, encontrado na maioria dos livros técnicos. Este exemplo é baseado no livro de Simon Thompson [55]. Algumas funções já foram definidas anteriormente, no entanto elas serão novamente definidas aqui para evitar ambiguidades. Vamos mostrar, inicialmente, os tipos de dados utilizados na simulação.

```
type Doc = String
type Linha = String
type Palavra = String
```

```
fazIndice :: Doc -> [[Int], Palavra]
```

Neste caso, o texto é uma *string* como a seguinte, onde não serão colocados os acentos e os *tis* para facilitar o entendimento:

```
doc :: Doc
doc = "Eu nao sou cachorro nao\nPra viver tao humilhado\nEu nao sou
      cachorro nao\nPara ser tao desprezado"
```

Vamos utilizar este trecho e mostrar como ele vai ser transformado com a aplicação das funções que serão definidas para realizar operações sobre ele:

- Dividir `doc` em linhas, ficando assim:


```
["Eu nao sou cachorro nao",
 "Pra viver tao humilhado",
 "Eu nao sou cachorro nao",
 "Para ser tao desprezado"]
```

 Esta operação é realizada por `divTudo :: Doc -> [Linha]`

- Agora devemos emparelhar cada linha com seu número de linha:
`[(1, "Eu nao sou cachorro nao"),
(2, "Pra viver tao humilhado"),
(3, "Eu nao sou cachorro nao"),
(4, "Para ser tao desprezado")]`
Esta operação é realizada por **numLinhas :: [Linha] -> [(Int, Linha)]**
- Temos agora que dividir as linhas em palavras, associando cada palavra com o número da linha onde ela ocorre
`[(1, "Eu"), (1, "nao"), (1, "sou"), (1, "cachorro"), (1, "nao"),
(2, "Pra"), (2, "viver"), (2, "tao"), (2, "humilhado"),
(3, "Eu"), (3, "nao"), (3, "sou"), (3, "cachorro"), (3, "nao"),
(4, "Para"), (4, "ser"), (4, "tao"), (4, "desprezado")]`
Esta operação é realizada por **todosNumPal :: [(Int, Linha)] -> [(Int, Palavra)]**
- Agora é necessário ordenar esta lista em ordem alfabética das palavras e, se a mesma palavra ocorre em mais de uma linha, ordenar por linha.
`[(1, "cachorro"), (3, "cachorro"), (4, "desprezado"), (1, "Eu"), (3, "Eu"), (2, "humilhado"), (1, "nao"), (3, "nao"), (4, "Para"), (2, "Pra"), (4, "ser"), (1, "sou"), (3, "sou"), (2, "tao"), (4, "tao"), (2, "viver")]`
Esta operação é realizada por **ordenaLista :: [(Int, Palavra)] -> [(Int, Palavra)]**
- Agora temos que modificar a lista de forma que cada palavra seja emparelhada com a lista unitária das linhas onde ela ocorre:
`[([1], "cachorro"), ([3], "cachorro"), ([4], "desprezado"), ([1], "Eu"), ([3], "Eu"), ([2], "humilhado"), ([1], "nao"), ([3], "nao"), ([4], "Para"), ([2], "Pra"), ([4], "ser"), ([1], "sou"), ([3], "sou"), ([2], "tao"), ([4], "tao"), ([2], "viver")]`
Esta operação é realizada por **fazListas :: [(Int, Palavra)] -> [[(Int), Palavra]]**
- Agora devemos juntar as linhas que contém uma mesma palavra em uma mesma lista de linhas:
`[([1, 3], "cachorro"), ([4], "desprezado"), ([1, 3], "Eu"), ([2], "humilhado"), ([1, 3], "nao"), ([4], "Para"), ([2], "Pra"), ([4], "ser"), ([1, 3], "sou"), ([2, 4], "tao"), ([2], "viver")]`
Esta operação é realizada pela função **mistura :: [[(Int), Palavra]] -> [[(Int), Palavra]]**
- Vamos agora diminuir a lista, removendo todas as entradas para palavras com menos de 3 letras:
`[([1, 3], "cachorro"), ([4], "desprezado"), ([2], "humilhado"), ([1, 3], "nao"), ([4], "Para"), ([2], "Pra"), ([4], "ser"), ([1, 3], "sou"), ([2, 4], "tao"), ([2], "viver")]`
Esta operação é realizada pela função **diminui :: [[(Int), Palavra]] -> [[(Int), Palavra]]**

Usando composição avançada, para ficar mais claro o exemplo, temos:

```
fazIndice = divideTudo >.>      -- Doc -> [Linha]
numDeLinhas >.>                -- [Linha] -> [(Int, Linha)]
todosNumPal >.>                -- [(Int, Linha)] -> [(Int, Palavra)]
ordenaLista >.>                -- [(Int, Palavra)] -> [(Int, Palavra)]
fazListas >.>                  -- [(Int, Palavra)] -> [[(Int), Palavra]]
mistura >.>                    -- [[(Int), Palavra]] -> [[(Int), Palavra]]
diminui >.>                    -- [[(Int), Palavra]] -> [[(Int), Palavra]]
```

Agora cada uma destas funções serão definidas.

```

divTudo :: Doc -> [Linha]
divTudo = tiraEspaco >.> formaLinhas

tiraEspaco :: Doc -> Doc
tiraEspaco [ ] = [ ]
tiraEspaco (a : x)
  | a == " " = tiraEspaco x
  | otherwise = (a : x)

pegaLinha :: Doc -> Linha
pegaLinha [ ] = [ ]
pegaLinha (a : x)
  | a /= '\n' = a : pegaLinha x
  | otherwise = [ ]

tiraLinha :: Doc -> Doc
tiraLinha [ ] = [ ]
tiraLinha (x : xs)
  | x /= '\n' = tiraLinha xs
  | otherwise = xs

formaLinhas :: Doc -> [Linha]
formaLinhas [ ] = [ ]
formaLinhas st = (pegaLinha st) : formaLinhas (tiraEspaco (tiraLinha st))

```

É importante notar que já existe no **Prelude** a função **lines**, cujo resultado é o mesmo da função **formaLinhas**. A definição de **formaLinhas** foi colocada aqui por motivos puramente didático-pedagógicos. O leitor deve se sentir incentivado a verificar este fato. Veremos, em seguida, a definição da função **numDeLinhas**.

```

numDeLinhas :: [Linha] -> [(Int, Linha)]
numDeLinhas lin = zip [1 .. length lin] lin

```

Vamos considerar, inicialmente, apenas uma linha:

```

numDePalavras :: (Int, Linha) -> [(Int, Palavra)]
numDePalavras (num, linha) = map poeNumLinha (divideEmPalavras linha)
  where poeNumLinha pal = (num, pal)

divideEmPalavras :: String -> [Palavra]
divideEmPalavras st = divide (tiraEspaco st)

divide :: String -> [Palavra]
divide [ ] = [ ]
divide st = (pegaPalavra st) : divide (tiraEspaco (tiraPalavra st))

pegaPalavra :: String -> String
pegaPalavra [ ] = [ ]
pegaPalavra (a : x)
  | elem a espacoEmBranco = [ ]
  | otherwise = a : pegaPalavra x

```

```

tiraPalavra :: String -> String
tiraPalavra [ ] = [ ]
tiraPalavra (a : x)
  | elem a espacoEmBranco = (a : x)
  | otherwise = tiraPalavra x

espacoEmBranco = ['\n', '\t', ' ']
todosNumPal :: [(Int, Linha)] -> [(Int, Palavra)]
todosNumPal = concat . map numDePalavras

```

Agora vamos definir a função de ordenação usando a técnica aplicada no algoritmo de ordenação por **quicksort**. Para que isto seja possível, é necessário fazer a comparação entre dois pares, onde o primeiro elemento do par é um número inteiro e o segundo uma palavra. Temos que comparar dois pares deste tipo para saber dentre eles qual deve vir primeiro e qual deve vir depois. Esta função será chamada de **menor**, definida a seguir:

```

menor :: (Int, Palavra) -> (Int, Palavra) -> Bool
menor (n1, w1) (n2, w2) = w1 < w2 || (w1 == w2 && n1 < n2)

```

A função **ordenaLista** deve ser definida de forma que pares repetidos não apareçam duplicados. No exemplo, o par **(1, nao)** e **(3, nao)** são duplicados, mas eles só aparecem uma vez. O leitor deve observar como esta exigência foi implementada na função.

```

ordenaLista :: [(Int, Palavra)] -> [(Int, Palavra)]
ordenaLista [ ] = [ ]
ordenaLista (a : x) = ordenaLista menores ++ [a] ++ ordenaLista maiores
  where menores = [b | b <- x, menor b a]
        maiores = [b | b <- x, menor a b]

```

Com a lista ordenada, vamos transformar cada par da forma **(Int, String)** em um par da forma **([Int], String)**, ou seja, vamos transformar cada inteiro em uma lista unitária de inteiros. Esta operação é definida da seguinte forma:

```

fazListas :: [(Int, Palavra)] -> [[[Int], Palavra]]
fazListas = map mklist where mklist (n, st) = ([n], st)

```

Neste ponto, vamos concatenar as listas de inteiros que contém a mesma palavra, formando uma lista única composta de todas as linhas que contém a mesma palavra. Esta operação é realizada pela função **mistura**, definida da seguinte forma:

```

mistura :: [[[Int], Palavra]] -> [[[Int], Palavra]]
mistura [ ] = [ ]
mistura [a] = [a]
mistura ((l1, w1) : (l2, w2) : rest)
  | w1 /= w2 = (l1, w1) : mistura ((l2, w2) : rest)
  | otherwise = mistura ((l1 ++ l2, w1) : rest)

```

Finalmente, serão retiradas da lista as palavras com menos de 3 letras. Esta operação foi colocada apenas para efeito de mostrar como pode ser definida, mas, na realidade, ela é dispensável. Esta operação é definida da seguinte forma:

```
diminui = filter tamanho
      where tamanho (n1, pal) = length pal > 3
```

Este exemplo visa mostrar que as linguagens funcionais não foram projetadas apenas para fazer cálculos matemáticos, como **fibonacci** e **fatorial**. Na realidade, Haskell serve também para estes cálculos, mas sua área de aplicação é muito mais ampla do que é imaginada por alguns programadores, defensores intransigentes de outros paradigmas de programação. O leitor deve ler o Capítulo introdutório deste estudo e consultar as referências [17, 59] para maiores informações sobre este tema ou consultar o *site* oficial de Haskell.

Exercício resolvido (cifragem por transposição).

Este exercício, mostra uma aplicação na área da Criptografia. Trata-se de criptografar uma determinada mensagem transformando-a em um texto difrado que deve ser enviado a alguém e que deve ser protegido contra a leitura, caso o texto cifrado seja interceptado por algum intruso, que pode ser o mensageiro ou outra pessoa. O receptor do texto cifrado deve conhecer uma chave, sem caracteres repetidos, com a qual deve decifrar o texto cifrado para descobrir a mensagem original.

Para este exemplo, vamos supor que Vinicius de Moraes desejava enviar a mensagem “EU SEI QUE VOU TE AMAR POR TODA A MINHA VIDA EU VOU TE AMAR EM CADA DESPEDIDA EU VOU TE AMAR” para uma sua namorada e que ambos haviam combinado como chave a palavra “COMPUTER”. Para entender melhor, colocamos números sobre cada letra da chave, indicando a sua ordem de ocorrência alfabética dentro da chave. No caso em voga, a primeira letra que ocorre no alfabeto, dentre as da palavra chave é a letra “C”, a segunda é a letra “E” e assim por diante. Neste caso, a primeira coluna da mensagem será a coluna rotulada pela letra “C”, a segunda coluna será a coluna rotulada pela letra “E” e assim por diante. Este esquema está mostrado na Tabela 5.2.

Tabela 5.2: Criação de um texto cifrado com chave “COMPUTER”.

C	O	M	P	U	T	E	R
1	4	3	5	8	7	2	6
E	U		S	E	I		Q
U	E		V	O	U		T
E		A	M	A	R		P
O	R		T	O	D	A	
A		M	I	N	H	A	
V	I	D	A		E	U	
V	O	U		T	E		A
M	A	R		E	M		C
A	D	A		D	E	S	P
E	D	I	D	A		E	U
	V	O	U		T	E	
A	M	A	R				

A mensagem será escrita na horizontal mas será enviada por coluna, iniciando com a coluna 1, depois com a coluna 2 e assim por diante. Em nosso caso, a mensagem cifrada deve ser a seguinte, esclarecendo que cada espaço em branco é também considerado um caractere.


```
"EUEOAVVMAE A   AAU  SEE   A MDURAIIOAUE R IOADDVMSVMTIA   DURQTP   ACPU   IURDHE
EME T EOAON TEDA   "
```

O primeiro processamento a ser feito com a *string* original é dividi-la em palavras do tamanho de cada coluna. Para isso construímos a função **div_em_col** da seguinte forma:

```
div_em_col :: String -> Int -> [String]
div_em_col [ ] _ = [ ]
div_em_col str tam = (pega_col str tam) : div_em_col (tira_col str tam) tam
```

Esta função utiliza as funções **pega_col**, que retorna uma palavra do tamanho de uma coluna a partir de uma *string*, e a função **tira_col** que retira uma palavra do tamanho de uma coluna de uma *string*, retornando a *string* restante sem os caracteres referentes a esta palavra.

```
pega_col :: String -> Int -> String
pega_col [ ] _ = [ ]
pega_col (x : xs) tam
  | tam > 0 = x : pega_col xs (tam - 1)
  | otherwise = [ ]
```

```
tira_col :: String -> Int -> String
tira_col [ ] _ = [ ]
tira_col (x : xs) tam
  | tam > 0 = tira_col xs (tam - 1)
  | otherwise = (x : xs)
```

Os caracteres que formam a palavra chave devem ser colocados em ordem crescente para se saber a ordem em que cada coluna foi colocada no texto cifrado. Para isso será utilizado um algoritmo de ordenação. No nosso caso vamos utilizar a definição dada para a função **ordena**, mostrada na *Seção 4.10* a seguir. Utiliza-se a função **zip** para construir pares da posição de cada letra da chave ordenada como números naturais que representam as posições das colunas, a partir de 0 (zero).

```
lista_pos :: String -> [Int]
lista_pos chave =
  pega_lista_pos chave (zip (ordena chave) [0..(length chave)-1])
```

É necessário que a tabela original seja reconstruída. Para isso é necessário saber a posição em que cada coluna deve ser recolocada para reconstruir a tabela original. Isto é feito através da função **pega_lista_pos**.

```
pega_lista_pos :: [Char] -> [(Char, Int)] -> [Int]
pega_lista_pos [ ] _ = [ ]
pega_lista_pos (x : xs) l = (pega_pos x l) : pega_lista_pos xs l
```

A função **pega_pos** é responsável por retornar a posição em que cada caractere tem na chave.

```
pega_pos :: Char -> [(Char, Int)] -> Int
pega_pos c ((a, n) : x)
  | c == a = n
  | otherwise = pega_pos c x
```

A construção da lista contendo todas as colunas na ordem em que foram construídas é feita pela função **lista**.

```
lista :: [String] -> [Int] -> [String]
lista ls [ ] = [ ]
lista ls (b : y) = (ls !! b) : lista ls y
```

A função **lista_int** utiliza a função **lista** com o texto cifrado dividido em palavras com o tamanho de cada coluna.

```
lista_int :: String -> String -> [String]
lista_int txt ch =
    lista (div_em_col txt (div (length txt) (length ch))) (lista_pos ch)
```

A função **pega** constrói cada linha da tabela, tomando os caracteres de mesma posição em cada coluna, formando uma linha da tabela original.

```
pega :: Int -> [String] -> String
pega _ [ ] = [ ]
pega n (x : xs) = (x !! n) : (pega n xs)
```

A lista contendo as linhas da tabela original é construída através da função **lista_fin** que utiliza o parâmetro **k** que representa o tamanho da chave.

```
lista_fin :: Int -> Int -> String -> String -> String
lista_fin n k txt ch
    | n < k = pega n prim ++ lista_fin (n + 1) k txt ch
    | otherwise = [ ] where prim = lista_int txt ch
```

Finalmente a *string* inicial é reconstruída usando a função **mens_original**.

```
mens_original :: String -> String -> String
mens_original txt ch = lista_fin 0 (div (length txt) (length ch)) txt ch
```

5.10 Algoritmos de ordenação

No mundo da Computação, possivelmente nenhuma operação seja mais importante e, por isso mesmo, mais analisada do que as operações de busca e ordenação. Elas são utilizadas em quase todos os programas de banco de dados, em compiladores, interpretadores e sistemas operacionais. Os algoritmos de ordenação são classificados em três tipos: inserção, seleção e troca. Nesta seção, suas implementações em Haskell serão aqui mostradas, finalizando a seção com otimizações feitas às suas implementações iniciais, sobre listas de inteiros. Deve ser mencionado que apesar das implementações serem para listas de inteiros, elas podem ser generalizadas para listas polimórficas, desde que os tipos sejam compostos de elementos que possam ser comparados pelas relações de ordem $<$, $<=$, $>$, $>=$, $/=$ e $==$. Estas generalizações são objeto de estudo do próximo Capítulo.

5.10.1 Ordenação por inserção

Uma forma de ordenar uma lista não vazia é inserir a cabeça da lista no local correto, que pode ser na cabeça da lista ou pode ser na cauda já ordenada. Por exemplo, para ordenar a lista

de inteiros $[3,4,1]$, devemos inserir 3 na cauda da lista, já ordenada, ou seja em $[1,4]$. Para que esta cauda já esteja ordenada é necessário apenas chamar a mesma função de ordenação, recursivamente, para ela. Vamos definir uma função de ordenação, **iSort**, que utiliza uma função auxiliar **ins**, cuja tarefa é inserir cada elemento da lista no lugar correto.

```
iSort :: [Int] -> [Int]
iSort [ ] = [ ]
iSort (a:x) = ins a (iSort x)
```

A lista vazia é considerada ordenada por vacuidade. Por isto a primeira definição. Para o caso da lista não vazia, devemos inserir a cabeça (**a**) na cauda já ordenada (**iSort x**). Falta apenas definir a função **ins**, que é auto-explicativa.

```
ins :: Int -> [Int] -> [Int]
ins a [ ] = [a]
ins a (b:y)
  | a <= b = a : (b : y)      -- a serah a cabeca da lista
  | otherwise = b : ins a y   -- procura colocar a no local correto
```

Este método de ordenação é conhecido como *inserção direta* e o leitor deve observar a simplicidade como ele é implementado em Haskell. Sugerimos comparar esta implementação com outra, em qualquer linguagem convencional. Além disso, também se deve considerar a possibilidade de aplicação desta mesma definição à listas de vários tipos de dados, significando que a definição pode ser polimórfica. Isto significa que, na definição da função **ins**, a única operação exigida sobre os valores dos elementos da lista a ser ordenada é que eles possam ser comparados através da operação \geq . Uma lista de valores, de qualquer tipo de dados, onde esta operação seja possível entre estes valores, pode ser ordenada usando esta definição.

Exercícios:

1. Mostre todos os passos realizados na chamada **iSort[2, 8, 1]**.
2. Defina uma função **numOcorre :: [t] -> t -> Int**, onde **numOcorre l s** retorna o número de vezes que o ítem **s** aparece na lista **l**.
3. Dê uma definição da função **pertence**, vista anteriormente, usando a função **numOcorre**, do ítem anterior.
4. Defina a função **ocorreUmaVez :: [Int] -> [Int]** que retorna a lista de números que ocorrem exatamente uma vez em uma lista. Por exemplo, **ocorreUmaVez[2, 4, 2, 1, 4] = [1]**.

5.10.2 Ordenação por seleção

Na ordenação em ordem crescente de uma lista de inteiros por *seleção*, é necessário selecionar o menor valor da lista e colocá-lo no início dela como a cabeça da lista. Continua-se este processo, escolhendo agora o menor elemento da cauda da nova lista e colocando-o no segundo lugar formando uma nova lista. Este processo continua até que todos os elementos da lista tenham sido colocados em seus devidos lugares.

A função **selSort** a seguir usa a função predefinida **minimum** que seleciona o menor elemento de uma lista de inteiros.

```

selSort :: [Int] -> [Int]
selSort [] = []
selSort xs = valmin : (selSort cauda)
               where valmin = minimum xs
                     cauda = remove valmin xs

```

A função **remove** quando aplicada a um valor inteiro e uma lista de inteiros, retira este valor inteiro da lista que é devolvida sem este valor.

```

remove :: Int -> [Int] -> [Int]
remove _ [] = []
remove a (x:xs)
  | x == a = xs
  | otherwise = x : remove a xs

```

5.10.3 Ordenação por trocas

O algoritmo de ordenação por trocas é considerado o pior algoritmo de ordenação. É conhecido como algoritmo da bolha, pela analogia normalmente feita com uma bolha de ar que sobe até o seu nível em um tanque com água. O algoritmo consiste na troca de cada elemento com o elemento que está ocupando momentaneamente a sua posição na lista. Quando a passagem não provocar mais qualquer troca de posições, a lista está ordenada. A função que faz a troca de posição entre dois elementos pode ser criada da seguinte forma:

```

bolha :: [Int] -> [Int]
bolha [ ] = [ ]
bolha (a:b:x)
  | a <= b = a : bolha (b : x)
  | otherwise = b : bolha (a : x)
bolha [x] = [x]

```

A função **bubble**, definida a seguir, faz as trocas e conta essas trocas.

```

bubble :: [Int] -> Int -> [Int]
bubble xs n
  | n == (length xs) = xs
  | otherwise = bubble (bolha xs) (n+1)

```

Finalmente a função **bubbleSort** completa a missão.

```

bubbleSort :: [Int] -> [Int]
bubbleSort xs = bubble xs 0

```

5.10.4 Ordenação otimizada

O algoritmo quickSort

Já foi mostrado como o algoritmo de ordenação por inserção direta pode ser implementado em Haskell. Aqui será mostrada a codificação de um outro algoritmo de ordenação, **quickSort**, destacando a simplicidade como ela é feita. Suponhamos que o algoritmo **quickSort** seja aplicado a uma lista de inteiros, ressaltando que ele também pode ser aplicado a listas de qualquer

tipo de dados, desde que estes dados possam ser comparados pelas relações de ordem: maior, menor e igual.

O algoritmo **quickSort** foi criado e nomeado por C. A. Hoare e é considerado o melhor algoritmo de ordenação, apesar de ser considerado um algoritmo de troca que normalmente apresenta a pior complexidade. O *quicksort* utiliza o método de divisão e conquista em seu desenvolvimento. Em sua descrição, escolhe-se um elemento, o *pivot*, e a lista a ser ordenada é dividida em duas sublistas: uma contendo os elementos menores ou iguais ao *pivot* e a outra contendo os elementos da lista que sejam maiores que o *pivot*. Neste ponto, o algoritmo é aplicado recursivamente à primeira e à segunda sub-listas, concatenando seus resultados, com o *pivot* entre elas. A escolha do *pivot*, normalmente, é feita pelo elemento do meio da lista, na expectativa de que ele esteja próximo da média da amostra. No entanto, esta escolha é apenas estatística e, na realidade, pode-se escolher qualquer elemento da lista. Em nossa implementação do **quickSort** em Haskell, escolhemos como *pivot* a cabeça da lista, por ser o elemento mais fácil de ser localizado.

Vamos acompanhar a sequência de operações na aplicação do **quickSort** à lista **[4,3,5,10]**.

```
quickSort [4,3,5,10]
= quickSort [3] ++ [4] ++ quickSort [5,10]
= (quickSort [ ] ++ [3] ++ quickSort [ ] ++ [4] ++
   (quickSort [ ] ++ [5] ++ quickSort [10]))
= ([ ] ++ [3] ++ [ ] ++ [4] ++ ([ ] ++ [5] ++
   (quicSort [ ] ++ [10] ++ quicSort [ ])))
= [3] ++ [4] ++ ([5] ++ ([ ] ++ [10] ++ [ ]))
= [3,4] ++ ([5] ++ [10])
= [3,4] ++ [5,10]
= [3,4,5,10]
```

Agora vamos definir formalmente o **quickSort**, usando expressões ZF.

```
quickSort :: [t] -> [t]
quickSort [ ] = [ ]
quickSort (a : x) = quickSort [y | y <- x, y <= a] ++ [a] ++
                    quickSort [y | y <- x, y > a]
```

Esta definição pode também ser feita usando definições locais, tornando-a mais fácil de ser compreendida, da seguinte forma.

```
quickSort :: [t] -> [t]
quickSort [ ] = [ ]
quickSort (a : x) = quickSort menores ++ [a] ++ quickSort maiores
    where menores = [y | y <- x, y <= a]
          maiores = [y | y <- x, y > a]
```

Não é fantástica esta definição? Sugiro ao leitor verificar a implementação deste algoritmo utilizando alguma linguagem imperativa como C, C++ ou Java, observando as diferenças em facilidade de entendimento e de implementação e de tamanho de código.

O algoritmo mergeSort

Pode ser demonstrado que o desempenho do algoritmo *quickSort* depende da relação entre os tamanhos das duas listas ou partições criadas. O algoritmo de mergesort também se baseia

na construção de duas partições que são recursivamente ordenadas, com o detalhe de que estas partições são de mesmo tamanho ou tem diferença de tamanho igual a 1. Isto significa que as partições estão sempre balanceadas. Quando as duas partições estão ambas ordenadas, é feita uma junção entre elas, uma operação conhecida como *merge*, que percorre as duas listas a partir de suas cabeças para as caudas, colocando o menor valor entre as cabeças na lista final. A função *merge* é definida da seguinte forma:

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

Após termos definido a função **merge**, o algoritmo de mergesort pode ser expresso da seguinte forma:

```
mergeSort :: [Int] -> [Int]
mergeSort [ ] = [ ]
mergeSort [x] = [x]
mergeSort xs = merge (mergeSort xs1) (mergeSort xs2)
  where xs1 = take k xs
        xs2 = drop k xs
        k   = (length xs) `div` 2
```

5.11 Resumo

Este Capítulo foi dedicado inteiramente ao estudo das listas em Haskell, completando o estudo dos tipos primitivos adotados em Haskell, além do tipo estruturado produto cartesiano, representado pelas tuplas. Ele se tornou necessário, dada a importância que as listas têm nas linguagens funcionais. Foi dada ênfase às funções predefinidas e às Compreensões, também conhecidas por expressões ZF, mostrando a facilidade de construir funções com esta ferramenta da linguagem. Vimos também a elegância, a facilidade e a adequação com que algumas funções, como por exemplo, o quicksort, foram definidas. Finalmente foram vistas características importantes na construção de funções como: polimorfismo, composição, avaliação parcial e curificação de funções.

Apesar dos tipos de dados estudados até aqui já significarem um avanço importante, Haskell vai mais além. A linguagem também permite a criação de tipos algébricos de dados e também dos tipos abstratos de dados, temas a serem estudados nos próximos Capítulos.

Grande parte deste estudo foi baseado nos livros de Simon Thompson [55] e de Richard Bird [6]. Estas duas referências representam o que de mais prático existe relacionado com exercícios usando listas. O livro de Paul Hudak [15] é também uma fonte de consulta importante, dada a sua aplicação à multimídia.

Uma fonte importante de problemas que podem ser resolvidos utilizando as ferramentas aqui mostradas é o livro de Steven S. Skiena e Miguel A. Revilla [44], que apresenta um catálogo dos mais variados tipos de problemas, desde um nível inicial até problemas de soluções mais elaboradas. Outra fonte importante de problemas matemáticos envolvendo grafos é o livro de Sriram Pemmaraju e Steven Skiena [36].

5.12 Exercícios propostos

1. Defina, em Haskell, uma função **f** que, dadas uma lista **i** de inteiros e uma lista **l** qualquer, retorne uma nova lista constituída pela lista **l** seguida de seus elementos que têm posição indicada na lista **i**, conforme o exemplo a seguir: **f [2,1,4] ['a', 'b', 'c', 'd'] = ['a', 'b', 'c', 'd', 'd', 'a', 'b']**.
2. Usando compreensão, defina uma função em Haskell, que gere todas as tuplas ordenadas de números x, y , e z menores ou iguais a um dado número n , tal que $x^2 + y^2 = z^2$.
3. Defina, em Haskell, uma função que calcule o *Determinante* de uma matriz quadrada de ordem n .
4. Defina, em Haskell, uma função **f** que, dada uma lista **l** construa duas outras listas **l1** e **l2**, de forma que **l1** contenha os elementos de **l** de posição ímpar e **l2** contenha os elementos de **l** de posição par, preservando a posição dos elementos, conforme os exemplos a seguir:

```
f [a, b, c, d] = [[a, c], [b, d]]
f [a, b, c, d, e] = [[a, c, e], [b, d]]
```

5. Um pequeno visor de cristal líquido (LCD) contém uma matriz 5x3 que pode mostrar um número, como 9 e 5, por exemplo:

```
***   ***
* *   *
***   ***
*     *
*     ***
```

O formato de cada número é definido por uma lista de inteiros que indicam quantos * (asteriscos) se repetem, seguidos de quantos brancos se repetem, até o final da matriz 5x3, começando da primeira linha até a última:

```
nove, cinco, um, dois, tres, quatro, seis, sete oito, zero :: [Int]
nove = [4,1,4,2,1,2,1]
cinco = [4,2,3,2,4]
um = [0,2,1,2,1,2,1,2,1,2,1]
dois = [3,2,5,2,3]
tres = [3,2,4,2,4]
quatro = [1,1,2,1,4,2,1,2,1]
seis = [4,2,4,1,4]
sete = [3,2,1,2,1,2,1,2,1]
oito = [4,1,5,1,4]
zero = [4,1,2,1,2,1,4]
```

indicando que o número nove é composto por 4 *s (três na primeira linha e um na segunda), seguida de 1 espaço, mais 4 *s, 2 espaços, 1 *, 2 espaços e 1 *. Construa funções para:

- a. Dado o formato do número (lista de inteiros) gerar a *string* correspondente de *s e espaços.

```
toString :: [Int] -> String
toString nove ==> "**** * *"
```

- b. Faça uma função que transforma a *string* de *s e espaços em uma lista de *strings*, cada uma representando uma linha do LCD:

```
type Linha = String
toLinhas :: String -> [Linha]
toLinhas "**** * *"
    ==> ["****", "* *", "****", " *", " *"]
```

- c. Faça uma função que pegue uma lista de *strings* e a transforme em uma única *string* com o caractere 'n' entre elas:

```
showLinhas :: [Linha] -> String
showLinhas ["****", "* *", "****", " *", " *"]
    ==> "****\n* *\n****\n *\n *"
```

- d. Faça uma função que pegue duas listas de linhas e transforme-as em uma única lista de linhas, onde as linhas originais se tornam uma única, com um espaço entre elas:

```
juntaLinhas :: [Linha] -> [Linha] -> [Linha]
juntaLinhas ["****", "* *", "****", " *", " *"]
    ["****", "* *", "****", " *", " *"]
    ==> ["**** ****", "* * * *", "**** ****",
        " * *", " * *"]
```

- e. Faça uma função que, dado um inteiro, imprima-o usando *s, espaços e caracteres 'n's. A função tem que funcionar para inteiros de 2 e 3 dígitos.

```
tolcd :: Int -> String
```

Dica: use as funções **div** e **mod** de inteiros, a função (!!) e a lista **numeros**, dada a seguir:

```
numeros :: [[Int]]
numeros
    = [zero, um, dois, tres, quatro, cinco, seis, sete, oito, nove]
```

- f. Faça uma função que, dada uma string de *s e espaços, retorne a representação da string como [Int] no formato usado no LCD, ou seja, a função inversa de toString.

```
toCompact :: String -> [Int]
toCompact "**** * *" = [4,1,4,2,1,2,1]
```

6. Defina, em Haskell, uma função que aplicada a uma lista **l** e a um inteiro **n**, retorne todas as sublistas de **l** com comprimento maior ou igual a **n**.

7. Dada a matriz

```
|2 3 4|
|5 6 7|
|8 9 10|
```

- Defina esta matriz enumerando todos os seus valores.
- Defina esta matriz utilizando expressões ZF.
- Defina uma função que transponha uma matriz de ordem 3.
- Extenda esta definição anterior para uma matriz de ordem **n**.

8. Voltando ao problema da criptografia resolvido neste Capítulo, tente um novo procedimento para criar um texto cifrado a partir de um texto pleno. Uma outra sugestão se refere à chave a ser utilizada. No caso mostrado, a chave não pode conter caracteres repetidos. Imagine uma solução para tratar chaves em que os dígitos possam ser repetidos.
9. Considere os exercícios de 8 e 9, propostos ao final do Capítulo 2, na página 32. Implemente-os em Haskell e verifique a semelhança entre as definições matemáticas e as feitas em Haskell.
10. O problema das n -rainhas consiste em colocar n rainhas em um tabuleiro de xadrez de tamanho $n \times n$, de modo que não fiquem 2 rainhas na mesma linha, coluna ou diagonal, ou seja, que nenhuma rainha esteja atacada ou atacando qualquer outra. Este problema pode ser particularizado para $n = 8$, que é o tabuleiro padrão de xadrez. Faça um programa em Haskell que resolva o problema das 8 rainhas.
11. Considere os “números de Mersenne” definidos no final do Capítulo 2. Existem números de Mersenne que são primos e outros que não os são. Por exemplo, $M_0 = 0$ (não é primo), $M_1 = 1$ (não é primo), $M_2 = 3$ (é primo), $M_3 = 7$ (é primo), $M_4 = 15$ (não é primo), etc. O maior número de Mersenne que é primo, registrado até 30 de setembro de 2009, é $M_{43112609} = 2^{43112609} - 1$ que representa o quadragésimo sexto número primo de Mersenne, com quase 13 milhões de algarismos em sua representação decimal e foi descoberto pelo *Great Internet Mersenne Prime Search*. Defina em Haskell uma função que verifique se um dado número inteiro não negativo, x , é, ou não, um primo de Mersenne.
12. Ordene a lista $[4, 2, 5, 6, 10, 3, 7]$ usando os métodos de ordenação inserção, seleção e troca, mostrando todas as etapas.
13. Construa uma lista de inteiros de tamanho 8 onde ocorra o pior caso do algoritmo de mergesort.
14. Defina, em Haskell, uma função que converta uma lista de números naturais de 0 a 9 em uma lista de tuplas, onde o primeiro elemento seja um número inteiro da primeira lista e o segundo a sua tradução como String. Como exemplo, a aplicação da função à lista $[2, 3]$ deve apresentar como resultado a lista $[(2, "dois"), (3, "tres")]$.
15. Seja uma lista de tuplas formadas pelo nome e idade de pessoas. Defina, em Haskell, uma função que retornem os nomes da pessoa mais nova e da mais velha, juntamente com suas idades.
16. Considere duas listas de números naturais não nulos. Defina, em Haskell, uma função que, quando aplicada a estas duas listas, retorne uma outra lista contendo os elementos da segunda lista que sejam múltiplos dos elementos da primeira lista.
17. Considere o jogo da Megasena, onde o resultado do sorteio é uma lista de 6 números naturais não repetidos, variando de 1 a 60. Um jogador pode escolher de 6 a 15 números em cada aposta e pode realizar quantas apostas quizer e tenha dinheiro para tal. Desenvolva um programa em Haskell que dada uma lista de 6 números como resultado do sorteio de um jogo da Megasena e uma lista de apostas, retorne uma lista das quantidades de pontos acertados em cada aposta, ordenada em forma decrescente.

Capítulo 6

Tipos de dados algébricos

*“Algebraic data types provides
a single powerfull way to describe data types.
Other languages often need several different
features to achive the same degree of expressiveness.”*
(Bryan O’Sullivan in [34])

6.1 Introdução

Este Capítulo é dedicado à construção de um novo tipo de dado estruturado, diferente dos apresentados até agora, conhecido como tipo algébrico de dados. Estes tipos de dados facilitam a simulação de problemas de forma mais natural e mais próxima do mundo real. A presença destes tipos de dados em Haskell possibilitam que ela seja uma linguagem utilizável em várias áreas de aplicação.

Vamos iniciar este estudo com as classes de tipos, conhecidas mais comumente como *type class*. As classes de tipos são utilizadas na implementação dos tipos algébricos e dos tipos abstratos de dados, estes últimos, um tema a ser estudado em outro Capítulo.

Os tipos algébricos, representam um aumento na expressividade e no poder de abstração de Haskell. Por este motivo, este estudo deve ser de domínio pleno para se aproveitar as possibilidades que a linguagem oferece, construindo bons programas. Como exemplo, as árvores são estruturas importantes na modelagem de vários problemas e podem ser implementadas em Haskell de forma muito simples usando os tipos algébricos de dados. Esta constatação pode ser verificada pelo usuário ao final deste Capítulo.

6.2 Classes de tipos

Já foram vistas funções que atuam sobre valores de mais de um tipo. Por exemplo, a função **length**, predefinida em Haskell, pode ser empregada para determinar o tamanho de listas de qualquer tipo. Assim, **length** é uma função polimórfica, ou seja, a mesma definição pode ser aplicada a listas homogêneas de qualquer tipo. Por outro lado, algumas funções podem ser aplicadas a mais de um tipo de dados, mas têm de apresentar uma nova implementação para cada tipo sobre o qual elas vão atuar. São os casos das funções $+$, $-$ e $*$, por exemplo. O algoritmo utilizado para somar dois valores inteiros é deferente do algoritmo para somar dois valores reais, uma vez que a forma de representação de valores inteiros é diferente da representação de valores reais. Estas funções são sobrecarregadas.

Existe uma discussão antiga entre os pesquisadores sobre as vantagens e desvantagens de se colocar sobrecarga em uma linguagem de programação. Alguns argumentam que a sobrecarga não aumenta o poder de expressividade da linguagem porque, mesmo tendo o mesmo nome, as funções sobrecarregadas atuam de forma diferenciada sobre cada um deles. Sendo assim, elas poderiam ter nomes distintos para cada definição. Outros admitem a sobrecarga como uma necessidade das linguagens, uma vez que ela permite reusabilidade e melhora a legibilidade dos programas [55]. Por exemplo, seria tedioso usar um símbolo para a soma de inteiros e outro para a soma de números reais. Esta mesma observação pode ser feita em relação aos operadores de subtração e multiplicação. A verdade é que todas as linguagens de programação admitem alguma forma de sobrecarga. Haskell não é uma exceção e admite que seus operadores aritméticos predefinidos sejam sobrecarregados.

Os tipos sobre os quais uma função sobrecarregada pode atuar formam uma coleção de tipos chamada classe de tipos (*type class*) em Haskell. Quando um tipo pertence a uma classe, diz-se que ele é uma *instância* dessa classe. As classes em Haskell permitem uma hierarquia entre elas, juntamente com um mecanismo de herança, de forma similar ao encontrado nas linguagens orientadas a objeto.

6.2.1 Fundamentação das classes

A função **elem**, definida a seguir, quando aplicada a um elemento e a uma lista de valores do tipo deste elemento, verifica se ele pertence ou não à lista, retornando um valor booleano.

```
elem :: t -> [t] -> Bool
elem x [ ] = False
elem x (a : y) = x == a || elem x y
```

Analisando a definição desta função aplicada à lista não vazia, verificamos que é feito um teste para verificar se o elemento **x** é igual a cabeça da lista (**x==a**). Para este teste é utilizada a função de igualdade **==**. Isto implica que a função **elem** só pode ser aplicada a tipos cujos valores possam ser comparados pela função **==**. Os tipos que têm esta propriedade formam uma classe em Haskell.

Em Haskell, existem dois operadores de teste de igualdade: **==** e **/=**. Para valores booleanos, eles são definidos da seguinte forma:

```
(/=), (==) :: Bool -> Bool -> Bool
x == y = (x and y) or (not x and not y)
x /= y = not (x == y)
```

É importante observar a diferença entre **==** e **=**. A função **==** é usada para testar computacionalmente uma igualdade, enquanto a função **=** é usada nas definições e no sentido matemático normal. Para a Matemática, a assertiva **double = square** é uma afirmação falsa e a assertiva $\perp = \perp$ ¹ é verdadeira, uma vez que qualquer coisa é igual a si própria. No entanto, para a Computação, as funções não podem ser testadas quanto a sua igualdade e o resultado da avaliação $\perp == \perp$ é também \perp e não **True**. Isto não quer dizer que o avaliador seja uma máquina matemática, mas tão somente que seu comportamento é descrito por um conjunto limitado de regras matemáticas, escolhidas de forma que elas possam ser executadas por um computador [6].

O objetivo principal de se introduzir um teste de igualdade é poder utilizá-lo em uma variedade de tipos distintos, não apenas no tipo **Bool**. Sabemos como um valor inteiro pode ser

¹O símbolo \perp é conhecido como “*bottom*” e é utilizado para denotar algo que seja indefinido [31].

comparado com um outro valor inteiro apenas analisando as suas representações numéricas, ou seja, comparando-se os *bits* posicionais de cada valor. Se pelo menos um par de *bits* correspondentes forem distintos, os valores são distintos e se todos os *bits* correspondentes forem iguais os valores são iguais. Só que este tipo de verificação não pode ser feito com outros tipos de dados diferentes dos inteiros, dos booleanos, dos caracteres ou das *strings*. Por exemplo, o que significa um registro ser igual a outro registro?

Assim, desejamos que `==` e `/=` sejam operadores sobrecarregados, ou seja, estas operações devem ser implementadas de forma diferente para cada tipo e a forma de fazer isso em Haskell é declarar uma classe que contenha todos os tipos para os quais `==` e `/=` vão ser definidas. Esta classe é predefinida em Haskell e é denominada **Eq**.

A forma de declarar **Eq** como a classe dos tipos que têm os operadores `==` e `/=` é a seguinte:

```
class Eq t where
    (==), (/=) :: t -> t -> Bool
```

Esta declaração estabelece que a classe **Eq** é formada por todos os tipos que contém as funções ou métodos, `==` e `/=`. Dito de outra forma, as funções `==` e `/=` são definidas para os tipos que compõem a classe **Eq**. Estas funções têm o seguinte tipo:

```
(==), (/=) :: (Eq t) => t -> t -> Bool
```

Agora o leitor pode entender algumas mensagens de erros na declaração de funções mostradas pelo sistema Haskell quando detecta algum erro de tipo. Normalmente, o sistema se refere a alguma classe de tipos, nestes casos.

6.2.2 Funções que usam igualdade

Analisemos agora a função **todosIguais** que verifica se três valores inteiros são iguais, ou não.

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

Observe, nesta definição, que não é feita qualquer restrição que a obrigue ser definida somente para valores inteiros. A única operação realizada com os elementos **m**, **n** e **p** é uma comparação entre eles através da função de igualdade `==`. Dessa forma, a função **todosIguais** pode ser aplicada a três valores de um tipo **t** qualquer, desde que seus valores possam ser comparados pela função `==`. Isto dá a função **todosIguais** um tipo mais geral, da seguinte forma:

```
todosIguais :: (Eq t) => t -> t -> t -> Bool
```

significando que a função **todosIguais** pode ser aplicada não apenas a valores inteiros, mas também a valores de qualquer tipo, desde que ele esteja na classe **Eq**, ou seja, um tipo para o qual sejam definidos um teste de igualdade (`==`) e um teste de desigualdade (`/=`). A parte antes do sinal “`=>`” é chamada de contexto que, neste caso, é a classe **Eq**. A leitura deste novo tipo deve ser: se um tipo **t** está na classe **Eq** então a função **todosIguais** tem o tipo **t -> t -> t -> Bool**. Isto significa que ela pode ter os seguintes tipos:

```
Int -> Int -> Int -> Bool,
Char -> Char -> Char -> Bool ou ainda
(Int, Bool) -> (Int, Bool) -> (Int, Bool) -> Bool
```

entre outros, uma vez que todos estes tipos têm uma função `==` definida para seus valores.

Vejamos agora o que acontece ao tentar aplicar a função **todosIguais** a argumentos do tipo função, como por exemplo, da função **suc** definida da seguinte forma:

```
suc :: Int -> Int
suc = (+1)
```

Vejamos o que acontece ao se chamar a função **todosIguais suc suc suc**.

O resultado mostrado pelos compiladores Haskell ou Hugs é **ERROR: Int -> Int is not an instance of class Eq**, significando que não existe uma definição do teste de igualdade para funções do tipo **Int -> Int** (o tipo de **suc**).

6.2.3 Assinaturas e instâncias

Foi visto que o teste de igualdade (`==`) é sobrecarregado, o que permite que ele seja utilizado em uma variedade de tipos para os quais esteja definido, ou seja, para instâncias da classe **Eq**. Será mostrado agora como as classes e instâncias são declaradas, por exemplo, a classe **Visible** que transforma cada valor em uma *string* e dá a ela um tamanho. Esta classe é necessária porque o sistema de entrada/saída de Haskell só permite a impressão de *strings*, ou seja, para que qualquer valor em Haskell seja impresso, é necessário que ele seja primeiro transformado em uma *string*, caso ainda não o seja.

```
class Visible t where
  toString :: t -> String
  size :: t -> Int
```

A definição inclui o nome da classe (**Visible**) e uma *assinatura*, que são as funções que compõem a classe juntamente com seus tipos. Um tipo **t** para pertencer à classe **Visible** tem de implementar as duas funções da assinatura, ou seja, coisas visíveis são coisas que podem ser transformadas em uma *string* e que tenham um *tamanho*. Por exemplo, para se declarar que o tipo **Char** seja uma instância da classe **Visible**, deve-se fazer a declaração

```
instance Visible Char where
  toString ch = [ch]
  size _ = 1
```

que mostra como um caractere deve ser transformado em uma *string* de tamanho 1 para que ele seja visível. De forma similar, para declararmos o tipo **Bool** como uma instância da classe **Visible**, temos de declarar

```
instance Visible Bool where
  toString True = "True"
  toString False = "False"
  size _ = 1
```

Para que o tipo **Bool** seja uma instância da classe **Eq** devemos fazer:

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

6.2.4 Classes derivadas

Uma classe em Haskell pode herdar as propriedades de outras classes, como nas linguagens orientadas a objetos. Como exemplo, vamos observar a classe **Ord** que possui as operações `>`, `>=`, `<`, `<=`, **max**, **min** e **compare**, além de herdar a operação `==` da classe **Eq**. Sua definição é feita da seguinte forma:

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

O tipo **Ordering** será definido mais adiante, quando nos referirmos aos tipos algébricos de dados. Neste caso, a classe **Ord** herda as operações de **Eq** (no caso, `==` e `/=`). Há a necessidade de se definir pelo menos a função `<` (pode ser outra) para ser utilizada na definição das outras funções da assinatura. Estas definições, conhecidas como *definições default*, formam o conjunto de declarações a seguir:

```
x <= y = (x < y || x == y)
x > y = y < x
x >= y = (y < x || x == y)
```

Vamos supor que se deseja ordenar uma lista e mostrar o resultado como uma *string*. Podemos declarar a função **vSort** para estas operações, da seguinte forma:

```
vSort = toString . iSort
```

onde **toString** e **iSort** são funções já definidas anteriormente. Para ordenar a lista é necessário que ela seja composta de elementos que pertençam a um tipo **t** e que possam ser ordenados, ou seja, pertençam a um tipo que esteja na classe **Ord**. Para converter o resultado em uma *string*, é necessário que a lista `[t]` pertença à classe **Visible**. Desta forma, **vSort** deve herdar das classes **Ord** e **Visible** e, portanto, tem o tipo

```
vSort :: (Ord t, Visible t) => [t] -> String
```

O caso em que um tipo herda de mais de uma classe é conhecido na literatura como herança múltipla. Esta propriedade é implementada em Haskell, apesar de se verificarem algumas discordâncias entre os projetistas de linguagens em relação a este tema. A herança múltipla também pode ocorrer em uma declaração de instância como:

```
instance (Eq a, Eq b) => Eq (a, b) where
    (x, y) == (z, w) = (x == z) && (y == w)
```

mostrando que se dois tipos **a** e **b** estiverem na classe **Eq** então o par **(a, b)** também está. A herança múltipla também pode ocorrer na definição de uma classe. Por exemplo,

```
class (Ord a, Visible a) => OrdVis a
```

significando que os elementos da classe **OrdVis** herdam as operações das classes **Ord** e **Visible**. Este é o caso da declaração de uma classe que não contém assinatura, ou contém uma assinatura vazia. Para estar na classe **OrdVis**, um tipo deve simplesmente estar nas classes **Ord** e **Visible**. Neste caso, a definição da função **vSort** anterior poderia ser modificada para

```
vSort :: (OrdVis t) => [t] -> String
```

Exercícios propostos:

1. Como você colocaria **Bool**, o tipo **Par (a, b)** e o tipo **Tripla (a, b, c)** como instâncias do tipo **Visible**?
2. Defina uma função para converter um valor inteiro em uma *string* e mostre como o tipo **Int** pode ser uma instância de **Visible**.
3. Qual o tipo da função `compare x y = size x <= size y`?

6.2.5 As classes predefinidas em Haskell

Haskell contém algumas classes predefinidas. Nesta sub-seção, vamos ver algumas delas com alguma explicação sobre a sua utilização e definição.

A classe **Eq**

Esta classe, já mencionada anteriormente, é composta pelos tipos cujos valores podem ser comparados quanto a sua igualdade ou a sua diferença. Para isso, foram construídas as funções de teste de igualdade, `==`, e de diferença, `/=`, cujas definições são as seguintes:

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

A classe **Ord**

A classe **Ord** contém tipos, cujos valores podem ser ordenados, ou seja, podem ser testados pelas funções `<`, `≤`, `>` e `≥`. Deve ser claro que os valores destes tipos também devem poder ser testados quanto as suas igualdades e diferenças. Estes testes já são providos pela classe **Eq**, portanto a classe **Ord** deve herdar estes testes da classe **Eq**. A classe **Ord** é definida da seguinte forma:

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

onde o tipo **Ordering** tem um entre três valores possíveis: **LT**, **EQ** e **GT**, que são os resultados possíveis de uma comparação entre dois valores. A definição de **compare** é:

```
compare x y
  | x == y = EQ
  | x < y = LT
  | otherwise = GT
```

A vantagem de se usar a função **compare** é que muitas outras funções podem ser definidas em função dela. Por exemplo,

```
x <= y = compare x y /= GT
x < y = compare x y == LT
x >= y = compare x y /= LT
x > y = compare x y == GT
```

As funções **max** e **min** são definidas por

```
max x y
  |x >= y = x
  |otherwise = y

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

A maioria dos tipos em Haskell pertencem às classes **Eq** e **Ord**. As exceções são as funções e os tipos abstratos de dados, um tema que será visto no Capítulo 7 deste livro.

A classe **Enum**

Esta é a classe dos tipos que podem ser enumerados. Como exemplo, a lista `[1,2,3,4,5,6]` pode também ser denotada por `[1 .. 6]` ou usando as funções da classe **Enum**, cuja definição é a seguinte:

```
class (Ord t) => Enum t where
  toEnum :: Int -> t
  fromEnum :: t -> Int
  enumFrom :: t -> [t] -- [m ..]
  enumFromThen :: t -> t -> [t] -- [m, n ..]
  enumFromTo :: t -> t -> [t] -- [m .. n]
  enumFromThenTo :: t -> t -> t -> [t] -- [m, m' .. n]
```

As funções **fromEnum** e **toEnum** têm as funções **ord** e **chr** do tipo **Char** como correspondentes, ou seja **ord** e **chr** são definidas usando **fromEnum** e **toEnum**. Simon Thompson [55] afirma que o *Haskell report* estabelece que as funções **toEnum** e **fromEnum** não são significativas para todas as instâncias da classe **Enum**. Para ele, o uso destas funções sobre valores de ponto flutuante ou inteiros de precisão completa (**Double** ou **Integer**) resulta em erro de execução.

A classe **Bounded**

Esta é uma classe que apresenta funções que retornam um valor mínimo e um valor máximo para cada tipo pertencente a ela. Suas instâncias são **Int**, **Char**, **Bool** e **Ordering**. Sua definição é a seguinte:

```
class Bounded t where
  minBound, maxBound :: t
```

A classe **Show**

Esta classe contém os tipos cujos valores podem ser descritos como **Strings**. Em Haskell, apenas as *strings* podem ser mostrados. Portanto, se um valor não for uma *string*, é necessário que ele seja transformado antes em uma *string* para depois ser mostrado. A maioria dos tipos pertence a esta classe. Sua definição é a seguinte:

```
type ShowS = String -> String

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


A classe Read

Esta classe contém os tipos cujos valores podem ser lidos a partir de *strings*. Para usar esta classe é necessário apenas conhecer a função `read :: Read t => String -> t`. Esta classe complementa a classe **Show**, uma vez que as *strings* produzidos pela função **show** são normalmente lidas por **read**.

```
read :: Read t => String -> t
```

A Figura 6.1 mostra um resumo das principais classes incorporadas à Biblioteca padrão de Haskell e a relação de herança existente entre elas, baseadas na referência [41]. É importante salientar que muitas outras classes são continuamente incorporadas a esta biblioteca, o que significa dizer que esta figura está sempre sendo modificada. Desta forma, a figura representa tão somente um estantâneo em um determinado momento.

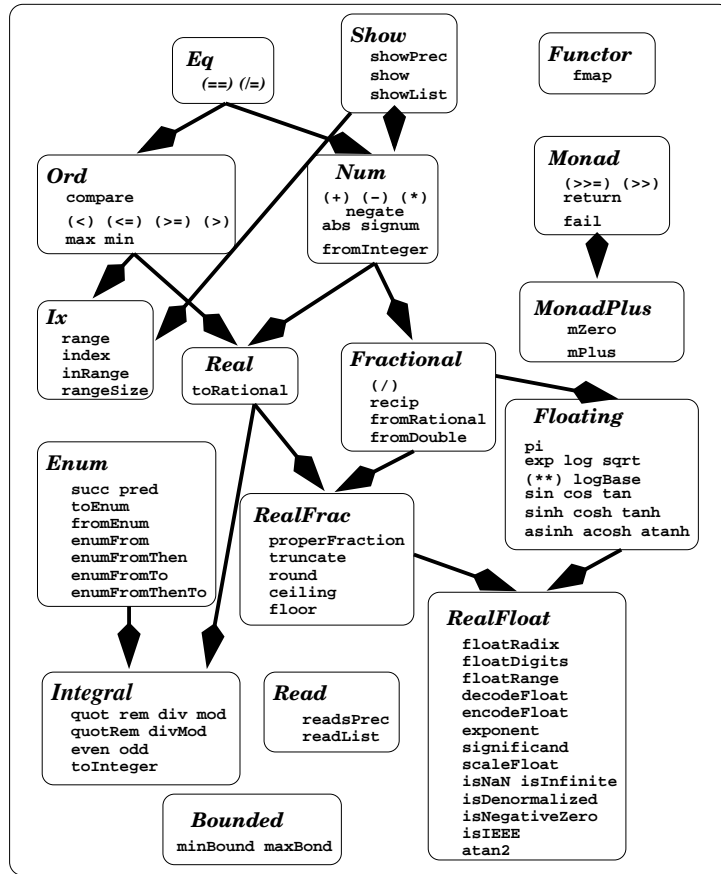


Figura 6.1: Resumo da hierarquia de classes em Haskell.

6.3 Tipos algébricos em Haskell

Já foram vistas várias formas de modelar dados em Haskell. Vimos as funções de alta ordem, polimorfismo, sobrecarga e as classes de tipos (*type class*). Vimos também que os dados podem ser modelados através dos seguintes tipos:

- *Tipos básicos (primitivos)*: **Int**, **Integer**, **Float**, **Double**, **Bool**, **Char** e listas.
- *Tipos compostos*: tuplas (t_1, t_2, \dots, t_n) e funções $(t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_n)$, onde t_1, t_2, \dots, t_n são tipos.

Estas facilidades já mostram um grande poder de expressividade e de abstração oferecidas pela linguagem. No entanto, Haskell oferece formas especiais para a construção de novos tipos de dados visando modelar situações peculiares. Como exemplos destas situações podemos citar, entre outras:

- as enumerações implementadas em algumas linguagens de programação;
- tipos cujos elementos sejam números ou *strings*. Por exemplo, uma casa em uma rua pode ser identificada por um número ou pelo nome da família - em alguns países isto é normal;
- as estruturas de árvores;
- as estruturas de grafos.

Para construir um novo tipo de dados, usamos a declaração **data** que descreve como os elementos deste novo tipo são construídos. Cada elemento é nomeado por uma expressão formulada em função dos construtores do tipo. Além disso, nomes diferentes denotam elementos também distintos. Através do uso de *pattern matching* sobre os construtores, definem-se operações que manipulam valores do tipo de dados escolhidos pelo programador. Os tipos assim descritos, não as operações, são chamados “tipos concretos” [55, 6] e são modelados em Haskell através dos tipos algébricos.

6.3.1 Como se define um tipo algébrico?

Um tipo algébrico é mais uma facilidade que Haskell oferece, proporcionando ao programador maior poder de abstração, necessário para modelar estruturas de dados complexas. Outras possibilidades também existem, como por exemplo, os tipos abstratos de dados a serem vistos no próximo Capítulo. A sintaxe da declaração de um tipo algébrico de dados é

```
data <Const_tipo> = <Const_Val1> | <Const_Val2> | ... | <Const_Valn>
```

onde **Const_tipo** é o “*construtor do tipo*” e **Const_Val1**, **Const_Val2**, ... **Const_Valn** são os “*construtores de valores*” ou “*construtores de dados*”, seguidos, ou não, de “*componentes do valor*”. Vamos mostrar um exemplo caracterizando cada um destes elementos.

```
data Publicacao = Livro Int String [String]
                | Revista String Int Int
                | Artigo String String Int
                deriving (Show)
```

Neste caso, **Publicacao** é o construtor do tipo e **Livro**, **Revista** e **Artigo** são os construtores dos valores ou dos dados. Os tipos **Int**, **String** e **[String]** são os componentes do valor **Livro**. De forma similar, **String**, **Int** e **Int** são os componentes do valor **Revista** e **String**, **String** e **Int** são os componentes do valor **Artigo**. A declaração **deriving (Show)**, que complementa esta declaração de tipo, informa que este tipo de dado, **Publicacao**, deve também pertencer a classe **Show**, para que ele possa ser mostrado.

Os componentes dos valores exercem os mesmos papéis que os campos dos registros nas linguagens de programação tradicionais. O componente **Int** seguindo o construtor do valor **Livro** é reservado para ser uma chave que identifica o livro em uma biblioteca ou em um Banco de Dados. O componente **String** deve ser utilizado para identificar o nome do livro e a lista **[String]** é reservada para representar os autores do livro.

Agora é possível criar uma instância do tipo **Publicacao** da seguinte forma:

```
pub = Livro 9780596514983 "Real World Haskell"
      ["Bryan O'Sullivan", "John Goerzen", "Don Stewart"]
```

Em **ghci**, o tipo do construtor Livro pode ser mostrado da seguinte forma:

```
<ghci>: type Livro
Livro :: Int -> String -> [String] -> Publicacao
```

Isto significa que os construtores de valores podem ser tratados como funções que criam e retornam uma nova publicação.

O tipo enumeração

O tipo enumeração, (**enum**), implementado nas linguagens C e C++ mantém alguma semelhança com os tipos algébricos implementados em Haskell, apesar de apresentar alguma diferença. Nas linguagens C e C++, os valores dos tipos enumeração são valores inteiros, podendo ser utilizados em qualquer contexto onde um valor inteiro seja esperado, representando uma fonte de possíveis *bugs* indesejáveis. Este não é o caso dos tipos algébricos em Haskell. Por exemplo, em C, o tipo enumeração **dia_util** pode ser definido da seguinte forma:

```
enum dia_util {segunda, terca, quarta, quinta, sexta};
```

Este tipo pode ser definido, em Haskell, como um tipo algébrico, com a vantagem da ausência dos bugs citados. Senão vejamos:

```
data Dia_util = Segunda
              | Terca
              | Quarta
              | Quinta
              | Sexta
              deriving (Eq, Show)
```

O tipo união livre

A união livre é um tipo de dado implementado em C e C++ que pode ser usado para modelar várias alternativas. Em C e C++, não é informada qual alternativa está presente. Isto significa que a união livre representa uma fonte de erros de programação porque o *type checker* fica impossibilitado de realizar seu trabalho. Seja, por exemplo, as declarações a seguir:

```
enum forma {circulo, retangulo};

struct ponto {float x, y};

struct circulo {
    struct ponto centro;
    float raio;
};

struct retangulo {
    struct ponto canto;
    float base;
    float altura;
};
```

```

struct figura {
    enum forma tipo;
    union {
        struct circulo meucirculo;
        struct retangulo meuretangulo;
    }fig_geometrica;
};

```

Neste exemplo, a união pode conter dados válidos para uma figura do tipo **circulo** ou para uma figura do tipo **retangulo**. Temos de usar o campo **tipo** do tipo **enum forma** para indicar explicitamente que tipo de valor está atualmente armazenado na união. No entanto, o campo **tipo** pode ser atualizado sem que a união também seja, ou então a união pode ser atualizada sem que o campo **tipo** também seja. Isto implica em insegurança na linguagem porque um valor que seja um círculo pode ser usado como se fosse um retângulo e vice-versa. A versão em Haskell para este código é dramaticamente menor e segura. Senão vejamos:

```

data Ponto = (Float, Float)

data Figura = Circulo Ponto Float
            | Retangulo Ponto Float Float

```

Se for criado um valor do tipo **Figura** usando o construtor de valores **Circulo**, não é possível utilizá-lo, acidentalmente, como um valor construído pelo construtor de valores **Retangulo**. Neste caso, estes construtores são utilizados como rótulos que indicam a origem de cada valor.

Produtos de tipos

Já foi visto que um novo tipo de dado é criado utilizando-se a declaração **data**. Haskell também permite a criação de sinônimos para dar um significado mais claro a algum tipo, utilizando para isso a declaração **type**. Também é sabido que um produto de tipos é um novo tipo de dados, cujos valores são construídos com mais de um construtor. Por exemplo,

```

type Nome = String
type Idade = Int
data Gente = Pessoa Nome Idade

```

A leitura de um valor do tipo **Gente** deve ser feita da seguinte forma: para construir um valor do tipo **Gente**, é necessário suprir um construtor de valor, **Pessoa**, juntamente com dois campos, digamos, **n** do tipo **Nome** e outro, digamos **i**, do tipo **Idade**. O elemento formado será **Pessoa n i**. O construtor **Pessoa** funciona como uma função aplicada aos dois argumentos **n** e **i**. Exemplos de valores deste tipo podem ser:

```

nome1 = Pessoa "Barack Obama" 40
nome2 = Pessoa "Mahatma Ghandi" 71

```

Podemos agora definir a função **mostraPessoa** que toma um valor do tipo **Gente** e o mostra na tela:

```

mostraPessoa :: Gente -> String
mostraPessoa (Pessoa n a) = n ++ " -- " ++ show a

```

A aplicação da função **mostraPessoa (Pessoa "John Lennon", 50)** será respondida por

```
"John Lennon -- 50"
```

Neste caso, o tipo **Gente** tem o único construtor, **Pessoa**, que utiliza dois valores dos tipos **Nome** e **Idade**, para formar um valor do tipo **Gente**. O valor **Pessoa n a** pode ser interpretado como sendo o resultado da aplicação de uma função, **Pessoa**, aos argumentos **n** e **a**, ou seja,

```
Pessoa :: Nome -> Idade -> Gente
```

O tipo para **Gente** poderia também ser definido como uma tupla, ou seja, como o tipo

```
type Gente = (Nome, Idade)
```

Existem vantagens e desvantagens nesta versão, no entanto, é senso comum que ela deve ser evitada. Os motivos são baseados na legibilidade, porque valores construídos desta forma nada indicam sobre a relação mnemônica que deve existir entre um tipo e seus valores.

Finalmente, ao se introduzir um tipo algébrico, podemos desejar que ele faça parte de determinadas classes de tipos. Esta função é provida pela declaração **deriving** mostrada nas declarações anteriores.

Exercícios propostos:

1. Sendo a Estação = Primavera, Verão, Outono, Inverno e Tempo = Quente, Frio, defina a função `{tempo :: Estacao -> Tempo}` de forma a usar guardas em vez de *pattern matching*.
2. Defina o tipo **Meses** como um tipo algébrico em Haskell. Faça uma função que associe um mês a sua Estação. Coloque ordenação sobre o tipo.
3. Defina uma função que calcule o tamanho do perímetro de uma forma geométrica do tipo **Forma**, definido no texto desta subseção.
4. Adicione um construtor extra ao tipo **Forma** para triângulos e estenda as funções **area** e **perimetro** (exercício anterior) para incluir os triângulos.

6.3.2 Tipos recursivos

Podemos usar o mesmo nome para o construtor do tipo e para o construtor de valores, já que eles representam objetos distintos. Por exemplo, é perfeitamente legal a construção do tipo

```
data Pessoa = Pessoa Nome Idade
```

Exemplo. Uma expressão aritmética simples que envolve apenas adições e subtrações de valores inteiros pode ser modelada através de sua BNF. Usando um tipo algébrico podemos modelar uma expressão da seguinte forma:

```
data Expr = Lit Int |Add Expr Expr |Sub Expr Expr
```

Alguns exemplos de modelagem envolvendo este tipo de dado podem ser:

- 2 é modelado por **Lit 2**
- 2 + 3 é modelado por **Add (Lit 2) (Lit 3)**

- $(3 - 1) + 3$ é modelado por **Add (Sub (Lit 3) (Lit 1)) (Lit 3)**

Podemos criar uma função de avaliação que tome como argumento uma expressão e dê como resultado o valor da expressão. Para isso podemos construir a função **eval** da seguinte forma:

```
eval :: Expr -> Int
eval (Lit n) = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

Esta definição é primitiva recursiva, ou seja, existe uma definição para um caso base (Lit n) e uma definição recursiva para o passo indutivo. Por exemplo, uma função que imprime uma expressão pode ser feita da seguinte maneira:

```
mostraExpressao :: Expr -> String
mostraExpressao (Lit n) = show n
mostraExpressao (Add e1 e2)
  = "(" ++ mostraExpressao e1 ++ "+" ++ mostraExpressao e2 ++ ")"
mostraExpressao (Sub e1 e2)
  = "(" ++ mostraExpressao e1 ++ "-" ++ mostraExpressao e2 ++ ")"
```

6.3.3 Tipos algébricos polimórficos

As definições de tipos algébricos podem conter tipos variáveis como **t**, **u**, etc. Por exemplo, `data Pares t = Par t t` e podemos ter

```
Par 2 3 :: Pares Int
Par [ ] [3] :: Pares [Int]
Par [ ] [ ] :: Pares [t]
```

Podemos construir uma função que verifique a igualdade das duas metades de um par:

```
igualPar :: (Eq t) => Pares t -> Bool
igualPar (Par x y) = (x == y)
```

As listas, vistas no capítulo anterior, podem ser construídas a partir de tipos algébricos da seguinte forma:

```
data List t = Nil | Cons t (List t)
              deriving (Eq, Ord, Show)
```

É possível estender o tipo **Expr** para que ele contenha expressões condicionais do tipo **If b e1 e2**, onde **e1** e **e2** são expressões e **b** é uma expressão booleana, um membro do tipo **BExp**.

```
data Expr = Lit Int
          | Op Ops Expr Expr
          | If BExp Expr Expr
```

A expressão **If b e1 e2** terá o valor **e1** se **b** tiver o valor **True** e terá o valor **e2** se **b** for **False**.

```
data BExp = BoolLit Bool
  | And BExp BExp
  | Not BExp
  | Equal Expr Expr
  | Greater Expr Expr
```

Estas cinco cláusulas dão os seguintes valores:

- Literais booleanos: **BoolLit True** e **BoolLit False**.
- A conjunção de duas expressões: **True** se as duas sub-expressões argumentos tiverem o valor **True**, caso contrário, o resultado será **False**.
- A negação de uma expressão: **Not x** tem o valor **True** se **x** for **False**.
- A igualdade de duas expressões: **Equal e1 e2** é **True** se as duas expressões numéricas tiverem valores iguais.
- A ordem maior: **Greater e1 e2** é **True** se a expressão numérica **e1** tiver um valor maior que o da expressão numérica **e2**.

A partir destes pressupostos, defina as funções:

```
eval :: Expr -> Int
bEval :: BExp -> Bool
```

por recursão e estenda a função **show** para mostrar o tipo redefinido para expressões.

Exercícios propostos:

1. Construir uma função que calcule o número de operadores em uma expressão.
2. Calcule:

```
eval (Lit 67)
eval (Add (Sub (Lit 3) (Lit 1) (Lit 3)))
mostraExpressao (Add (Lit 67) (Lit (-34)))
```

3. Adicione as operações de divisão e multiplicação de inteiros ao tipo **Expr** e redefina as funções **eval** e **mostraExpressao**, vistas anteriormente. O que sua operação de divisão faz no caso do divisor ser zero?
4. Defina uma função **permuta** que troca a ordem de uma união, ou seja,

```
permuta :: Uniao t u -> Uniao u t
```

Qual será o efeito da aplicação **permuta . permuta**?

6.4 Árvores

As listas são consideradas estruturas de dados lineares porque seus itens aparecem em uma sequência predeterminada, ou seja, cada item tem, no máximo, um sucessor imediato. Por outro lado, as estruturas de dados, conhecidas como árvores, são estruturas não lineares porque seus itens podem ter mais de um sucessor imediato.

As árvores podem ser utilizadas como representações naturais para quaisquer formas de dados organizados hierarquicamente, por exemplo, as estruturas sintáticas das expressões aritméticas ou funcionais. Na realidade, as árvores provêm uma generalização eficiente das listas como formas de organização e recuperação de informações.

Existem diversos tipos de árvores, classificadas de acordo com a forma de sua bifurcação, com a localização de armazenamento das informações, com o relacionamento entre estas informações, ou ainda, de acordo com o objetivo da árvore criada. Nesta seção, serão vistas apenas algumas destas definições.

6.4.1 Árvores binárias

Como o nome indica, uma árvore binária é uma árvore que tem, no máximo, duas bifurcações para os seus itens. Podem existir várias definições para estas árvores, no entanto, aqui será adotada a seguinte:

```
data ArvBin t = Folha t | No (ArvBin t) (ArvBin t) deriving (Eq, Ord, Show)
```

Esta é a definição de uma árvore binária onde cada elemento ou é um nó externo (uma Folha com um valor do tipo `t`) ou é um nó interno, sem um valor armazenado, mas com duas subárvores binárias como sucessoras para este nó.

Uma sequência de números inteiros pode ser representada por várias árvores binárias distintas. Por exemplo, a partir da sequência de números inteiros 14, 09, 19 e 51 podem ser geradas as 5 árvores binárias mostradas graficamente na Figura 6.2, ou seja, a) `No (Folha 14) (No (Folha 09) (No (Folha 19) (Folha 51)))`, b) `No (Folha 14) (No (No (Folha 09) (Folha 19)) (Folha 51))`, c) `No (No (Folha 14) (Folha 09)) (No (Folha 19) (Folha 51))`, d) `No (No (Folha 14) (No (Folha 09) (Folha 19))) (Folha 51)` e finalmente e) `No (No (No (Folha 14) (Folha 09)) (Folha 19)) (Folha 51)`.

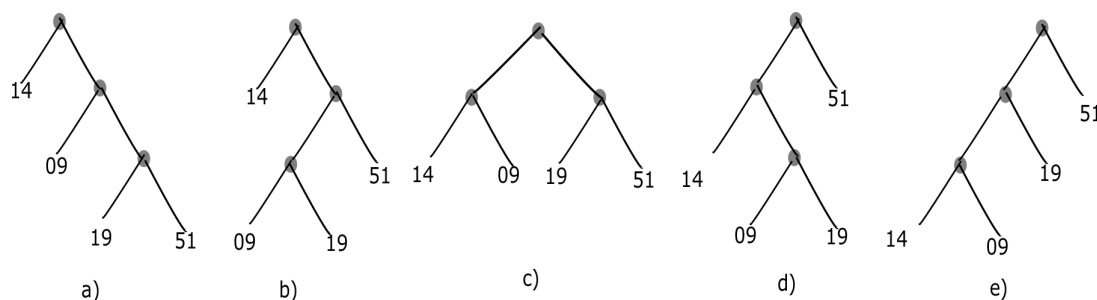


Figura 6.2: Representação gráfica das árvores binárias geradas pela sequência 14, 09, 19 e 51.

6.4.2 Funções sobre árvores binárias

Existem várias medidas importantes das árvores em geral. Duas delas são o *tamanho* e a *altura*. O tamanho de uma árvore se refere a quantidade de nós que detém informações. No caso da

árvore binária definida desta forma, só existem informações armazenadas nas folhas. Portanto, o tamanho desta árvore é a quantidade de suas folhas, desempenhando o mesmo papel que a função **length** exerce sobre as listas. Sua definição, em Haskell, é:

```
tamArvBin :: ArvBin t -> Int
tamArvBin (Folha a) = 1
tamArvBin (No ae ad) = tamArvBin ae + tamArvBin ad
```

Uma outra operação importante que pode ser aplicada às árvores binárias é transformá-las em listas. Isto pode ser feito da seguinte forma:

```
arvBintoLista :: ArvBin t -> [t]
arvBintoLista (Folha a) = [a]
arvBintoLista (No ae ad) = arvBintoLista ae ++ arvBintoLista ad
```

Já foi observado que nas árvores binárias definidas desta forma um nó ou é uma folha ou tem duas bifurcações. Nestas árvores, existe uma relação importante entre a quantidade de nós internos e externos, onde o número de nós externos é sempre igual ao número de nós internos mais 1. Os nós internos podem ser contados pela seguinte função:

```
nosInt :: ArvBin t -> Int
nosInt (Folha a) = 0
nosInt (No ae ad) = 1 + nosInt ae + nosInt ad
```

Uma característica importante das árvores é a *profundidade* de cada um de seus nós, que é dada pela distância, em níveis, que cada nó tem da raiz da árvore. A profundidade da raiz da árvore é zero. A profundidade da árvore é a maior das profundidades de suas folhas. Sua definição é a seguinte:

```
profArvBin :: ArvBin t -> Int
profArvBin (Folha a) = 0
profArvBin (No ae ad) = 1 + max (profArvBin ae) (profArvBin ad)
```

Diz-se que uma árvore binária é *perfeita* se todas as suas folhas tiverem a mesma profundidade, ou seja, se estiverem no mesmo nível. Na Figura 6.2, mostrada anteriormente, apenas a árvore representada na letra c) é uma árvore binária perfeita. O tamanho de uma árvore binária perfeita é sempre uma potência de 2 e existe exatamente uma árvore binária perfeita para cada potência de 2.

Duas árvores binárias com o mesmo tamanho, não têm, necessariamente, a mesma profundidade. Mesmo assim, estas medidas estão sempre relacionadas. O resultado a seguir é verificado em todas as árvores binárias finitas. Sua formulação é a seguinte:

$$\text{profArvBin } a < \text{tamArvBin } a \leq 2^{\text{profArvBin } a}$$

A demonstração desta propriedade é feita utilizando o **PIF** (Princípio de Indução Finita) sobre a profundidade **h**. A base da indução é feita quando a árvore tem profundidade **h = 0**, ou seja, para uma Folha e o passo indutivo é realizado admitindo-se que seja verdade para uma árvore do tipo **No ae ad** e **profundidade h**, verificando a validade para uma árvore de **profundidade h + 1**.

Sistema de prova:

1. **Caso (Folha a).** Para este caso, temos:

$profArvBin (Folha a) = 0$	-pela definição de $profArvBin$
$tamArvBin (Folha a) = 1$	-pela definição de $tamArvBin$
$profArvBin (Folha a) < tamArvBin (Folha a)$	
$tamArvBin (Folha a) = 1 = 2^0 = 2^{(profArvBin (Folha a))}$	

Assim, a assertiva é válida para o caso (**Folha a**).

2. **Caso (No ae ad)**. Neste caso, temos:

$tamArvBin (No ae ad)$	-pela definição de $profArvBin$
$= tamArvBin ae + tamArvBin ad$	-pela definição de $tamArvBin$
$\leq 2^{(profArvBin ae)} + 2^{(profArvBin ad)}$	-pela hipótese de indução
$\leq 2^h + 2^h$	
$= 2^{(h+1)}$	$-h = \max (profArvBin ae) (profArvBin ad)$

Conclusão: como a assertiva é válida para o caso (**Folha a**) e para o caso (**No ae ad**), então ela é válida para toda árvore binária finita.

Dada qualquer lista finita xs de tamanho n , é possível construir uma árvore a com

$arvBintolista a = xs$ e $altArvBin a = \lceil \log n \rceil$

onde $\lceil x \rceil$ representa o menor inteiro que supera x .

Esta árvore tem uma altura mínima para a lista xs . Normalmente existem mais de uma árvore de altura mínima para uma dada sequência de valores. Estas árvores são importantes porque o custo de recuperação de um valor é mínimo, mesmo no pior caso.

Uma árvore de altura mínima pode ser obtida pela divisão da lista de entrada em duas metades e, recursivamente, obtendo-se árvores de alturas mínimas para estas metades. A função **fazArvBin**, definida a seguir, constrói esta árvore:

```
listoArvBin :: [t] -> ArvBin t
listoArvBin xs
  = if m == 0           --se xs só tiver um elemento
    then Folha (head xs)
    else No (listoArvBin xe) (listoArvBin xd)
    where (xe, xd) = (take m xs, drop m xs)
          m = div (length xs) 2
```

6.4.3 Árvores binárias aumentadas

As árvores binárias definidas até aqui se caracterizam pelo fato de suas informações serem armazenadas apenas nas folhas. No entanto, é possível definir uma árvore binária em que os nós internos também armazenem informações. Estas árvores têm aplicações importantes e são conhecidas como *árvores binárias aumentadas*.

No nosso caso, vamos aumentar a árvore binária, colocando em cada nó interno a informação dos tamanhos das subárvores que eles representam. Para isto, vamos definir um novo tipo de árvore binária, que será denotada por **ArvBinA t**.

```
data ArvBinA t = Folha t | No Int (ArvBinA t) (ArvBinA t)
               deriving (Eq, Ord, Show)
```

Nesta definição, os construtores dos valores (**Folha** e **No**) são os mesmos adotados na definição do tipo **ArvBin**. No entanto, eles devem ser diferenciados em *scripts* em que os dois

tipos de árvores estejam presentes, para evitar ambiguidade. Para rotular os nós internos de uma árvore binária aumentada com valores inteiros representando os tamanhos das subárvores relacionadas a cada nó, devemos redefinir a função **tamArvBin** feita para a árvore binária **ArvBin t**.

```
tamArvBinA :: ArvBinA t -> Int
tamArvBinA (Folha x) = 1
tamArvBinA (No n ae ad) = n
```

Os nós internos serão rotulados utilizando a função **rotula** definida da seguinte forma:

```
rotula :: ArvBinA t -> ArvBinA t -> ArvBinA t
rotula ae ad = No n ae ad where n = tamArvBinA ae + tamArvBinA ad
```

A definição da função **listoArvBin** também deve ser modificada para construir uma árvore binária aumentada com altura mínima a partir de uma lista *xs*. Sua nova definição passa a ser:

```
listoArvBinA :: (Eq t) => [t] -> ArvBinA t
listoArvBinA xs
  | (m==0) = Folha (head xs) --se a lista xs for unitária
  | otherwise = rotula (listoArvBinA xse) (listoArvBinA xsd)
    where m = div (length xs) 2
          (xse, xsd) = (take m xs, drop m xs)
```

A função **arvBintoLista**, que transforma uma árvore binária em uma lista, também tem de ser modificada para transformar uma árvore binária aumentada em uma lista. Ela passa a ter a seguinte definição:

```
arvBinAtoLista :: ArvBinA t -> [t]
arvBinAtoLista (Folha x) = [x]
arvBinAtoLista (No n ae ad) = (arvBinAtoLista ae) ++ (arvBinAtoLista ad)
```

Podemos agora implementar uma função **recupera** que retorna o valor de um determinado nó da árvore. Isto pode ser feito transformando-se a árvore em uma lista, aplicando em seguida a função **!!** que recupera o *k*-ésimo valor desta lista.

```
recupera :: ArvBinA t -> Int -> t
recupera a k = (arvBinAtoLista a)!!k
```

A função **recupera** indexa a árvore da mesma forma que a função **!!** indexa uma lista, ou seja, ela retorna o valor especificado em um nó, enquanto **!! k** recupera o *k*-ésimo valor de uma lista. Podemos também usar a relação a seguir, que é verdadeira sobre listas:

```
(xs ++ ys)!!k = if k<m then xs!!k else ys!!(k-m)
                where m = length xs
```

Agora podemos implementar a função **recupera** de forma mais eficiente, transformando-a em **recupABA** da seguinte maneira:

```
recupABA :: ArvBinA t -> Int -> t
recupABA (Folha a) 0 = a
recupABA (No m ae ad) k
  = if k < m then recupABA ae k else recupABA ad (k-m)
```

Esta nova definição, **recupABA**, usa a informação do tamanho da subárvore que está armazenada nos nós internos para controlar a busca por um elemento, em uma dada posição. Esta implementação é mais eficiente que a anterior que transforma a árvore toda em uma lista e depois usa **!!** para recuperar um valor e esta busca é proporcional ao tamanho da lista.

Exercícios propostos(baseados em Richard Bird [6]).

1. Quantas árvores existem de tamanho 5? Escreva um programa para calcular o número de árvores binárias de tamanho n , para um dado $n \in \mathbf{N}$.
2. Prove que o número de folhas em uma árvore binária é sempre igual a 1 mais o número de nós internos.
3. As subárvores de uma árvore binária podem ser definidas por

```
subarvores :: ArvBin t -> [ArvBin t]
subarvores (Folha x) = [Folha x]
subarvores (No xt yt) = [No xt yt] ++ subarvores xt ++ subarvores yt
```

Estabeleça e prove o relacionamento entre **length (subarvores xt)** e **tamanho xt**.

4. A árvore aumentada também pode ser construída inserindo-se em cada nó interno a informação da profundidade deste nó. Faça esta definição.

6.4.4 Árvores de buscas binárias

As árvores binárias também podem ser definidas levando em consideração a existência de uma árvore nula. Este tipo de árvore é conhecida como árvore de busca binária. Sua definição pode ser feita da seguinte forma:

```
data ArvBusBin t = Nil | No (ArvBusBin t) t (ArvBusBin t)
    deriving (Eq, Ord, Show)
```

Este é o exemplo de uma declaração de tipo de dado que utiliza um contexto, ou seja, a árvore do tipo **ArvBusBin t** é definida para tipos **t** que sejam instâncias da classe **Ord**. Isto significa que todos os valores dos nós internos desta árvore podem ser comparados pelas relações *maior*, *maior ou igual*, *menor*, *menor ou igual* e *igual*. Como pode ser observado, esta definição não diferencia nós internos de externos porque não contempla uma definição de **Folha** e as informações armazenadas nos nós aparecem entre as subárvores componentes. O construtor **Nil** representa a árvore vazia ou nula. Este novo tipo de árvore também é conhecida como “*árvore binária rotulada*”. As funções **arvtoLista** e **arvBintoLista**, definidas anteriormente e que transformam as árvores binárias e binárias aumentadas em listas, podem ser facilmente modificadas para refletir as alterações da nova definição. Ela será definida de forma a retornar a lista ordenada dos rótulos da árvore, sendo este o motivo pelo qual os elementos da nova árvore devem pertencer a tipos cujos valores possam ser ordenados. Neste caso, os elementos serão ordenados na forma *in-ordem*.

```
arvBBtoLista :: (Ord t) => ArvBusBin t -> [t]
arvBBtoLista Nil = [ ]
arvBBtoLista (No ae n ad) = arvBBtoLista ae ++ [n] ++ arvBBtoLista ad
```

Como o nome sugere, as árvores de busca binárias são utilizadas para realizar buscas de forma eficiente. A função **membroABB** que determina se um determinado valor aparece, ou não, como o rótulo de algum nó pode ser definida da seguinte forma:

```

membroABB :: (Ord t) => t -> ArvBusBin t -> Bool
membroABB x Nil = False
membroABB x (No ae n ad)
  | (x<n)  = membroABB x ae
  | (x==n) = True
  | (x>n)  = membroABB x ad

```

No pior caso, o custo de avaliar **membroABB x abb** é proporcional à profundidade de **abb**, onde a definição de **profArvBin** deve ser modificada para **profABB**, da seguinte forma:

```

profABB :: (Ord t) => ArvBusBin t -> Int
profABB Nil = 0
profABB (No ae n ad)
  | (ae == Nil) && (ad == Nil) = 0
  | otherwise = 1 + max (profABB ae) (profABB ad)

```

Uma árvore de busca binária de tamanho n e profundidade p satisfaz a seguinte relação:

$$\lceil \log(n+1) \rceil \leq p < n+1$$

Uma árvore de busca binária pode ser construída a partir de uma lista de valores. Para isto, é necessário modificar a definição da função **fazArvBin**, transformando-a em **fazABB**, da seguinte forma:

```

fazABB :: (Ord t) => [t] -> ArvBusBin t
fazABB [ ] = Nil
fazABB (x:xs) = No (fazABB ys) x (fazABB zs)
  where (ys, zs) = particao (<=x) xs

```

onde a função **particao** é definida por:

```

particao :: (t -> Bool) -> [t] -> ([t], [t])
particao p xs = (filter p xs, filter (not.p) xs)

```

A diferença entre as funções **fazABB** e **fazArvBin** é que **fazABB** não garante que a árvore resultante tenha altura mínima. Por exemplo, o valor de **x** na segunda equação de **fazABB** pode ser menor que qualquer elemento de **xs**, possibilitando que **fazABB** construa uma árvore em que a subárvore esquerda seja nula. Enquanto a definição de **fazArvBin** pode ser executada em tempo linear, a implementação mais eficiente de **fazABB** requer, pelo menos, $n \log n$ passos, quando ambas são aplicadas a uma lista de tamanho **n**.

A definição da função **fazABB** pode ser utilizada para implementar uma função **sort** para ordenar listas, da seguinte forma:

```

sort :: (Ord t) => [t] -> [t]
sort = arvBBtoLista . fazABB

```

Se for eliminada a árvore intermediária desta definição, ela se transforma em uma definição da função **quicksort**, bastante utilizada na ordenação de listas.

Inserção e remoção em árvores de busca binária

As árvores de busca binária podem ser utilizadas na implementação eficiente dos conjuntos como um tipo de dados abstratos a ser definido no próximo Capítulo. Para isto é necessário definir as funções de inserção e remoção de itens nestas árvores. A função de inserção será **insere**:

```

insere :: (Ord t) => t -> ArvBusBin t -> ArvBusBin t
insere x Nil = No Nil x Nil
insere x (No ae n ad)
  | (x<n) = No (insere x ae) n ad
  | (x==n) = No ae n ad
  | (x>n) = No ae n (insere x ad)

```

Nesta definição, pode-se observar que, se um item já estiver na árvore, ele não será mais incluído. A função de remoção é um pouco mais complexa porque ao se retirar um nó interno da árvore é necessário juntar as duas sub-árvores deste nó. A função de remoção será definida como **remove** da seguinte forma:

```

remove :: (Ord t) => t -> ArvBusBin t -> ArvBusBin t
remove x Nil = Nil
remove x (No ae n ad)
  | (x<n) = No (remove x ae) n ad
  | (x>n) = No ae n (remove x ad)
  | (x==n) = junta ae ad

```

A função auxiliar **junta ae ad** tem a responsabilidade de unir as duas subárvores **ae** e **ad**, satisfazendo a seguinte propriedade:

arvBBtolista (junta ae ad) = arvBBtolista ae ++ arvBBtolista ad

A junção das sub-árvores **ae** e **ad** requer que seja escolhido um valor para ser a nova raiz da árvore. As escolhas possíveis são o maior valor da sub-árvore **ae** ou o menor valor da sub-árvore **ad**. No nosso caso, escolhemos o menor valor da sub-árvore da direita, **ad**, que deve ser retirado de sua posição inicial para ocupar a posição da raiz que foi removida. Esta junção pode ser feita da seguinte forma:

```

junta :: (Ord t) => ArvBusBin t -> ArvBusBin t -> ArvBusBin t
junta Nil ad = ad
junta ae Nil = ae
junta (No aee x aed) (No ade y add)
  = No (No aee x aed) k (remove k (No ade y add))
  where k = minArv (No ade y add)

minArv :: (Ord t) => ArvBusBin t -> t
minArv (No Nil x _) = x
minArv (No xt _ _) = minArv xt

```

Na Figura 6.3 está mostrado um exemplo da função **junta** em ação. Na árvore inicial, o rótulo da raiz (40) será removido deixando as subárvores **ae** e **ad** isoladas. Elas deverão ser juntas para formar a nova árvore e, para isso, o menor rótulo da sub-árvore **ad**, 45, será movido de sua posição na sub-árvore **ad** e ocupar o lugar da raiz.

6.4.5 Árvores heap binárias

Vamos agora definir um outro tipo de árvore binária conhecida como “*árvore heap binária*”, cujo tipo será **ArvHB**, definido da seguinte forma:

```

data ArvHB t = Nil | No t (ArvHB t) (ArvHB t) deriving (Eq, Ord, Show)

```

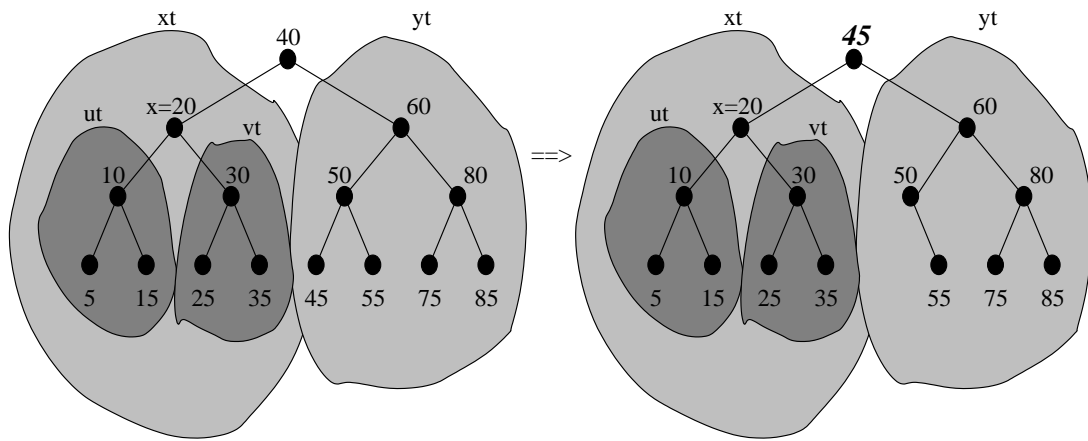


Figura 6.3: Representação gráfica da função junta em ação.

O tipo **ArvHB t** é virtualmente igual ao tipo **ArvBusBin t**, vista na subseção anterior, com a diferença de que o rótulo em cada nó em uma “*árvore heap binária*” é colocado antes das subárvores esquerda e direita e não entre elas. Além disso, existem dois tipos de *heaps* binárias: “*heaps máximas*” e “*heaps mínimas*”. Em ambas, os valores nos nós satisfazem a uma *propriedade de heap*, cujos detalhes específicos dependem do tipo de *heap*.

Em uma *heap máxima*, para todo nó i diferente da raiz, o valor do nó pai de i é sempre maior ou igual ao valor do nó filho. Neste caso, o maior elemento em uma *heap máxima* está armazenado na raiz. Já em uma *heap mínima*, a organização é feita de modo oposto, ou seja, para todo nó i diferente da raiz, o valor do nó pai de i é sempre menor ou igual ao valor do nó i . Neste caso, o elemento de menor valor na *heap* estará na raiz do *heap*. A Figura 6.4 mostra graficamente exemplos destes dois tipos de *heaps*.

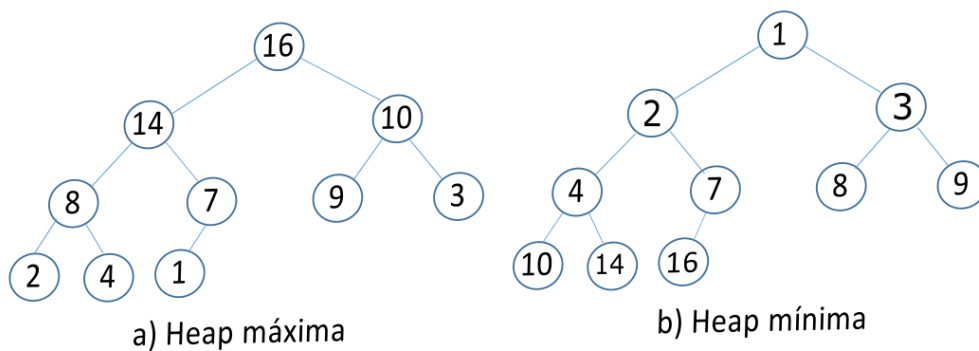


Figura 6.4: Exemplos de heaps máxima e mínima.

Neste livro, serão definidas funções para *heaps mínimas*, destacando que as funções para o gerenciamento de *heaps máximas* podem ser facilmente construídas a partir delas.

Para formalizar esta condição, vamos definir inicialmente a função **arvHBtoLista** que transforma uma árvore heap binária em uma lista ordenada:

```
arvHBtoLista :: (Ord t) => ArvHB t -> [t]
arvHBtoLista Nil = [ ]
arvHBtoLista (No n ae ad) = n : merge (arvHBtoLista ae) (arvHBtoLista ad)
```

```

merge :: (Ord t) => [t] -> [t] -> [t]
merge [ ] ys = ys
merge (x : xs) [ ] = x : xs
merge (x : xs) (y : ys) = if x <= y then x : merge xs (y : ys)
                           else y : merge (x : xs) ys

```

As árvores *heap* binárias podem ser utilizadas para propósitos diferentes dos indicados para as árvores de busca binárias, ou seja, para operações que não sejam de remoção, inserção ou pertinência. As árvores *heap* binárias são mais adequadas para tarefas como encontrar o menor valor em uma *heap*, que pode ser feita em tempo constante. Esta operação realizada em uma árvore de busca binária requer tempo proporcional à distância ao nó mais à esquerda. Enquanto a junção de duas árvores de busca binárias é uma tarefa complexa, a junção de duas árvores *heaps* binárias é simples.

Construção de heaps

Uma forma de construir uma árvore *heap* binária a partir de uma lista é definir uma nova versão da função **fazABB**, transformando-a na função **fazHB**.

```

fazHB :: (Ord t) => [t] -> ArvHB t
fazHB [ ] = Nil
fazHM (x : xs) = No x (fazHB ys) (fazHB zs)
               where (ys, zs) = particao (<x) xs

```

Deve ser observado que, apesar do programa ser bastante pequeno, ele não representa a forma mais eficiente de construção de *heaps*. Para corrigir este defeito, é necessário definir **fazHeap** como uma composição de funções auxiliares:

```

fazHeap :: (Ord t) => [t] -> ArvHB t
fazHeap = heapfica . fazHB

```

A função **fazHB** constrói uma árvore de altura mínima, não necessariamente ordenada, e a função **heapfica** reorganiza os rótulos para garantir a ordenação.

```

heapfica :: (Ord t) => ArvHB t -> ArvHB t
heapfica Nil = Nil
heapfica (No n ae ad) = desliza n (heapfica ae) (heapfica ad)

```

A função **desliza** toma um rótulo e duas árvores *heap* binárias e constrói uma nova árvore *heap* binária, deslizando o rótulo até que a propriedade de *heap* seja estabelecida.

```

desliza :: (Ord t) => t -> ArvHB t -> ArvHB t -> ArvHB t
desliza x Nil Nil = No x Nil Nil
desliza x (No y aee aed) Nil = if x <= y then No x (No y aee aed) Nil
                               else No y (desliza x aee aed) Nil
desliza x Nil (No z ade add) = if x <= z then No x Nil (No z ade add)
                               else No z Nil (desliza x ade add)
desliza x (No y aee aed) (No z ade add)
  | x <= (min y z) = No x (No y aee aed) (No z ade add)
  | y <= (min x z) = No y (desliza x aee aed) (No z ade add)
  | z <= (min x y) = No z (No y aee aed) (desliza x ade add)

```

Finalmente, podemos usar árvores *heaps* binárias para obter uma nova implementação para o algoritmo de ordenação, conhecido como **heapsort**:


```
heapsort :: (Ord t) => [t] -> [t]
heapsort = arvHBtolista.fazHeap
```

6.4.6 Árvores Rosas

O nome “árvores rosas” é uma tradução de “*rhododendron*” e tem sido assim tratadas por Lambert Meertens² para descrever árvores com várias subdivisões, ou seja, nós, incluindo o nó raiz, contendo diversas subárvores. O tipo **ArvRosa a** é definido por

```
data ArvRosa a = No a [ArvRosa a]
```

Um elemento de uma árvore rosa, **ArvRosa a**, consiste de um nó rotulado juntamente com uma lista de subárvores rosas. Por definição, “*nós externos*” não tem subárvores como filhas, enquanto todos os nós internos tem pelo menos uma subárvore rosa como filha. São exemplos de árvores rosas: **No 0 []** e a árvore **No 0 [No 1 [], No 2 [], No 3 []]**.

É também possível que uma árvore rosa tenha um número infinito de subárvores como suas filhas. Uma tal subárvore pode ser definida por

```
No 0 [No n [ ] | n <- [1..]]
```

Uma árvore rosa **t** é finita se todos os nós de **t** tiverem também um número finito de subárvores imediatas. Deve ser observado que uma árvore rosa finita não precisa necessariamente que ela tenha tamanho finito.

Uma árvore rosa **t** tem **largura w** se todos os seus nós tiverem, no máximo, **w** subárvores rosas imediatas. Todas as árvores rosas finitas tem também larguras finitas. Uma árvore rosa **t** é chamada de *k-ária* se todo nó interno tiver exatamente *k* subárvores imediatas. As árvores binárias são estruturalmente equivalentes às árvores rosas *2-árias*.

As definições de “tamanho” e “profundidade” das árvores rosas não apresentam novidade em relação aos outros tipos de árvores. A definição da função “tamanho” pode ser:

```
tamanho :: ArvRosa a -> Int
tamanho (No x xts) = 1 + sum (map tamanho xts)
```

Já a definição de “*profundidade*” de uma árvore rosa pode ser definida como a seguir, onde se observa que a profundidade de uma árvore rosa é sempre maior que zero.

```
profundidade :: ArvRosa a -> Int
profundidade (No x xts) = 1 + maxlist (map profundidade xts)
    where maxlist = foldl (max) 0
```

onde **foldl** é a versão de **fold** associativa pela esquerda. Sua definição é a seguinte:

```
foldl :: (t -> u -> v) -> v -> [u] -> t
foldl f e [ ] = e
foldl f e (x:xs) = foldl f (f e x) xs
```

A função “*profundidade*” pode ser definida da seguinte forma:

²Lambert Guillaume Louis Théodore Meertens (Amsterdam, 10/05/1944) - professor e cientista da Computação.

```

profundidade = maxRose.depths

depths :: ArvRosa a -> ArvRosa Int
depths = down 1

down :: Int -> ArvRosa a -> ArvRosa Int
down n (No x xts) = No n (map (down (n+1)) xts )

maxRosa :: (Ord a) => ArvRosa a -> a
maxRosa (No x xts) = x max maxlist (map maxRosa xts)

```

Uma árvore rosa é dita “*perfeita*” se todos os seus nós externos tiverem a mesma profundidade. Um relacionamento básico entre as funções *tamanho* e *profundidade* afirma que $xt \leq tamanho\ t \leq (k^{(profundidade\ xt)} - 1)/(k - 1)$ para todas as árvores rosa *k-árias* finitas. Além disso, uma árvore rosa *k-ária* perfeita com profundidade *p* tem tamanho $(k^p - 1)/(k - 1)$.

A função **fold** para árvores rosa, pode ser denominada **foldRosa** e definida da seguinte forma:

```

foldRosa :: (a -> [b] -> b) -> ArvRosa a -> b
foldRosa f (No x xts) = f x (map (foldRosa) xts)

```

onde os argumentos *f* e *No* são argumentos definidos dos seguintes tipos:

```

f :: a -> [b] -> b

No :: a -> [ArvRosa a] -> ArvRosa a

```

Como exemplos, podemos ter:

```

tamanho = foldRosa f where f x ns = 1 + sum ns

maxRosa = foldRosa f where f x ns = x max maxlist ns

mapRosa :: (a -> b) -> ArvRosa a -> ArvRosa b
mapRosa f = foldRosa (No.f)

```

Representação de árvores rosas por árvores binárias

As árvores rosas finitas com suas estruturas multivariadas parecem ser capazes de modelar estruturas mais gerais que as árvores binárias. Apesar de parecer surpresa, existe uma correspondência biunívoca entre elas, ou seja, toda árvore rosa finita pode ser representada por uma única árvore binária finita e vice-versa. A correspondência em si não é única, mas é um método de associar os dois tipos de árvores de forma similar à correspondência entre expressões escritas nas formas curried e não curried. Por exemplo, considere as seguintes expressões:

$f(g(x,y)\ z,\ h(t))$ e $f(g\ x\ y)\ z\ (h\ t)$

A expressão à esquerda pode ser representada pela seguinte árvore rosa:

```

No f [No g [No x [ ], No y [ ]], No z [ ], No h [No t [ ]]]

```

Já a expressão à direita, em sua forma completamente parentetizada é $f((g\ x)\ y)\ z)\ (h\ t)$ e pode ser representada pela seguinte árvore binária:

```

No (No (No (Folha f)
          (No (No (Folha g) (Folha x)) (No y)))
      (Folha z)
      (No (Folha h) (Folha t)))

```

Pode-se converter um elemento finito de uma árvore rosa, **Rosa a**, em um elemento de uma árvore binária, **ArvBin a**, através da função *toB* definida da seguinte forma:

```

toB :: ArvRosa a -> ArvBin a
toB (No x xts) = foldl No (Folha x) (map toB xts)

```

Isto significa que, para converter uma árvore rosa em uma árvore binária, primeiramente convertemos todas as suas subárvores, via *map toB*, e então combinamos os resultados em uma única árvore binária, aplicando *fold* a partir da esquerda, seguindo a ordem de associação da aplicação pela esquerda.

Para converter uma árvore binária em uma árvore rosa, definimos a função *toR* da seguinte forma:

```

toR :: ArvBin a -> ArvRosa a
toR (Folha x) = No x [ ]
toR (No xb yb) = No x (xts ++ [xt])
                  where No x xts = toR xb
                        xt      = toR yb

```

Esta definição, no entanto, não é muito eficiente porque a junção de *xt* a *xts* consome tempo proporcional ao tamanho de *xts*, e este pode ser grande.

Exercício resolvido (busca do menor caminho em árvores).

Este tipo de problema acontece com frequência em diversas áreas da computação. Vamos utilizar um caso particular conhecido como o “problema dos baldes” que consiste na existência de três baldes, sendo um com capacidade para 8 litros, outro para 5 litros e o último com capacidade para 3 litros. O problema consiste em verificar se, a partir de uma determinada quantidade de líquido em cada um dos três baldes, existe uma sequência de operações que podem ser feitas com eles, de forma a atingir uma determinada configuração desejada. As únicas operações possíveis são a transferência de líquido de um balde para outro até este atingir sua capacidade máxima ou o balde de origem ficar completamente vazio.

Cada estado dos baldes, em um determinado momento, vai ser representado por uma tripla do tipo (Int, Int, Int), onde cada valor individual representa o volume de líquido que cada balde tem em cada momento. O processamento consiste em encontrar uma sequência de tuplas deste tipo, onde a última corresponda à configuração desejada.

Uma solução para este problema pode ser construída utilizando árvores multiárias, onde cada tupla (estado) dá origem a várias subárvores formadas pelos sucessores possíveis para esta tupla (estado). Para implementar esta solução em Haskell, vamos definir o tipo de dado

```

type Terno = (Int, Int, Int)

```

As operações de despejo de cada balde em outro podem ser representadas de forma individual, onde cada balde pode colocar líquido em um outro jarro. Vamos simbolizar os baldes por A, B e C. Assim, a transferência do líquido do balde A para B será representada pela função

```
jogaAemB :: Terno -> [Terno]
jogaAemB (a, b, c)
  | (a == 0) || (b == 5) = [ ]
  | a + b > 5 = [(a+b-5, 5, c)]
  | otherwise = [(0, a+b, c)]
```

Para transferir o líquido do balde A para o balde C, ou para a transferência dos demais baldes, estas operações são implementadas da seguinte forma:

```
jogaAemC :: Terno -> [Terno]
jogaAemC (a, b, c)
  | (a == 0) || (c == 3) = [ ]
  | a + c > 3 = [(a+c-3, b, 3)]
  | otherwise = [(0, b, 3)]
```

```
jogaBemA :: Terno -> [Terno]
jogaBemA (a, b, c)
  | (b == 0) || (a == 8) = [ ]
  | a + b > 8 = [(8, a+b-8, c)]
  | otherwise = [(a+b, 0, c)]
```

```
jogaBemC :: Terno -> [Terno]
jogaBemC (a, b, c)
  | (b == 0) || (c == 3) = [ ]
  | b + c > 3 = [(a, b+c-3, 3)]
  | otherwise = [(a, 0, c+b)]
```

```
jogaCemA :: Terno -> [Terno]
jogaCemA (a, b, c)
  | (c == 0) || (a == 8) = [ ]
  | c + a > 8 = [(8, b, a+c-8)]
  | otherwise = [(a+c, b, 0)]
```

```
jogaCemB :: Terno -> [Terno]
jogaCemB (a, b, c)
  | (c == 0) || (b == 5) = [ ]
  | c + b > 5 = [(a, 5, c+b-5)]
  | otherwise = [(a, b+c, 0)]
```

Dessa forma, os estados possíveis a partir de uma determinada configuração inicial serão os seguintes:

```
possiveis_sucessores :: Terno -> [Terno]
possiveis_sucessores j = (jogaAemB j) ++ (jogaAemC j) ++ (jogaBemA j) ++
  (jogaBemC j) ++ (jogaCemA j) ++ (jogaCemB j)
```

Deve ser observado que alguns destes estados não interessam se eles já estiverem na sequência de estados. Assim, os sucessores de um estado em uma sequência devem ser escolhidos de forma a não se repetir um estado já existente na lista. A operação **pertence**, definida a seguir, verifica se um determinado estado já se encontra na sequência dada. Se isto acontecer, a sequência deve ser descartada.

```

pertence :: Terno -> [Terno] -> Bool
pertence j [ ] = False
pertence (x,y,z) ((m, n, k): xs)
    | (x == m) && (y == n) && (z == k) = True
    | otherwise = pertence (x,y,z) xs

sucessores :: [Terno] -> [Terno] -> [[Terno]]
sucessores _ [ ] = [ ]
sucessores ls (j : js)
    | pertence j ls = sucessores ls js
    | otherwise = ([j] ++ ls) : sucessores ls js

```

A função **seq_achada** encontra uma lista de ternos a partir de uma lista de listas de ternos iniciais.

```

seq_achada :: [[Terno]] -> Terno -> [Terno]
seq_achada [ ] _ = [ ]
seq_achada (l:ls) jr
    | pertence (head l) [jr] = reverse l
    | otherwise = seq_achada ls jr

```

Se a configuração desejada não tiver sido atingida ainda neste nível, então devemos procurar as configurações para um próximo nível na árvore. Isto será feito pela função **map_suc**:

```

map_suc :: [[Terno]] -> [[Terno]]
map_suc [ ] = [ ]
map_suc (l:la) = (sucessores l (possiveis_sucessores (head l))) ++ map_suc la

```

Como as sequências são descartadas a medida que um estado possível já esteja na sequência, é possível que a configuração final seja uma lista vazia. Se o estado desejado for atingido, a sequência deve ser mostrada. Isto é feito pela função **caminho**:

```

caminho :: [[Terno]] -> Terno -> [[Terno]]
caminho [ ] _ =
    error "Este estado nao pode ser atingido a partir dessa configuracao inicial"
caminho (l:ls) jr
    | res_int /= [ ] = [res_int]
    | otherwise = caminho (map_suc (l:ls)) jr
                    where res_int = seq_achada (l:ls) jr

```

Finalmente, a função **res_fin** mostra uma sequência de ternos iniciando com o terno inicial a partir do qual pode ser feita a sequência de operações até atingir o terno final. Sua definição é feita da seguinte forma, tendo como argumentos o terno inicial e o final.

```

res_fin :: Terno -> Terno -> [Terno]
res_fin ti tf = concat ([[ti]] ++
    (caminho (map_suc (sucessores [ ] (possiveis_sucessores ti))) tf))

```

Para um caso particular, a partir de uma configuração em que o primeiro jarro contenha 8 litros de vinho e os outros dois estejam vazios, ou seja, a partir do estado (8,0,0), e se deseja atingir o estado em que os baldes A e B tenham 4 litros cada um e o balde C esteja vazio, a configuração final (4,4,0), a menor sequência de operações será representada pela lista

$[(8,0,0), (3,5,0), (3,2,3), (6,2,0), (6,0,2), (1,5,2), (1,4,3), (4,4,0)]$

desenvolvida em 8 etapas. Uma outra sequência, mas de um caminho maior, é $[(8,0,0), (3,5,0), (0,5,3), (5,0,3), (5,3,0), (2,3,3), (2,5,1), (7,0,1), (7,1,0), (4,1,3), (4,4,0)]$ desenvolvida em 11 etapas.

6.4.7 Árvores AVL

Na seção anterior, foi visto que é desejável manter a profundidade máxima da árvore de busca em torno de $O(\log n)$ para que as buscas sejam realizadas de forma otimizada, ou seja, em tempos mínimos. Uma árvore AVL, devida a Adelson-Velskii e Landis [1], é uma árvore de busca binária com uma condição adicional de balanço. Esta condição de balanço estipula que para todo nó, a diferença entre as profundidades entre quaisquer subárvores da esquerda e da direita deve ser, no máximo, igual a 1. Desta forma, as árvores AVL representam um relaxamento da condição de árvores perfeitamente balanceadas.

Por exemplo, as árvores da Figura 6.5 a) e b) são AVL, enquanto a árvore mostrada na letra c) não é.

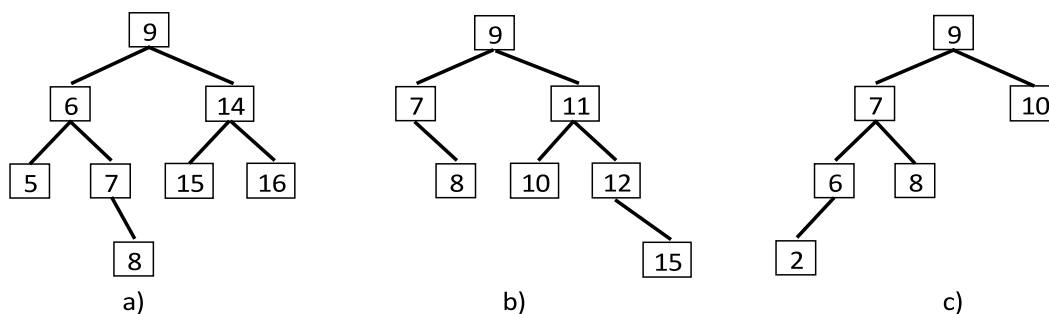


Figura 6.5: As árvores das letras a) e b) são AVL. A árvore da letra c) não é.

A construção em Haskell de uma árvore AVL pode ser feita da seguinte forma:

```

module AVLTree (AVLTree, emptyAVL, addAVL) where
data (Ord a, Show a) => AVLTree a = EmptyAVL
    | NodeAVL a (AVLTree a) (AVLTree a)
    deriving Show

emptyAVL = EmptyAVL
  
```

Desta forma a operação de busca em uma árvore AVL sempre levará $O(\log n)$ passos. Para assumir que uma árvore esteja na forma AVL é necessário garantir que as operações de inserção e remoção de valores na árvore não alterem a condição de balanço. Para que isto seja garantido, é necessário que sejam feitas rotações à esquerda ou à direita, um tema mostrado a seguir.

Rotações

Nesta sub-seção, são examinadas algumas transformações sobre as árvores de buscas binárias que modificam a forma, sem afetar a propriedade de ordem. Se uma operação de inserção viola a condição de balanço é sempre possível aplicar uma destas transformações para conseguir de volta a condição de balanço.

Rotação simples

Seja a árvore mostrada no lado esquerdo da Figura 6.6. Na construção desta figura e de outras a seguir foi adotada uma metodologia de mostrar o valor da raiz com a letra **v** dentro de um retângulo, a raiz da subárvore esquerda tem o valor **ve** e suas subárvores descendentes **aee** e **aed** e a subárvore direita é denotada por **ad**. Esta técnica foi necessária para tornar o texto coerente com as figuras mostradas e ao mesmo tempo facilitar a compreensão dos procedimentos de rotações simples ou duplas.

Uma *rotação à direita* consiste em substituir a árvore por uma outra em que a raiz tenha o valor **ve** com a subárvore esquerda sendo **aee** e a subárvore direita tendo **v** como o valor da raiz e **aed** e **ad** como suas descendentes. Esta operação está ilustrada na Figura 6.6.

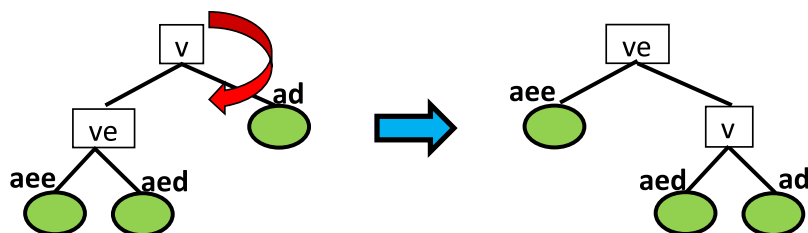


Figura 6.6: Rotação à direita.

Uma rotação simétrica, mas com sentido oposto, chamada de *rotação à esquerda*, está mostrada na Figura 6.7, onde o leitor deve acompanhar e entender plenamente os movimentos indicados na árvore.

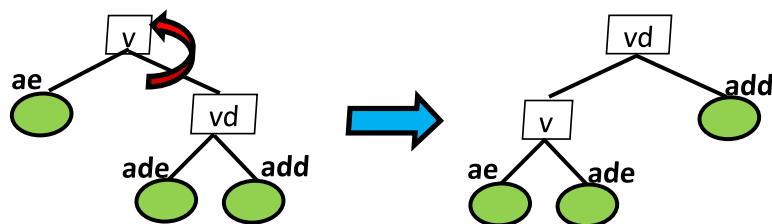


Figura 6.7: Rotação à esquerda.

As funções que implementam estas duas rotações são definidas usando casamento de padrões, da seguinte forma:

```
rotateRight, rotateLeft :: (Ord a, Show a) => AVLTree a -> AVLTree a

rotateRight EmptyAVL = EmptyAVL
rotateRight (NodeAVL v (NodeAVL ve aee aed) ad)
    = NodeAVL ve aee (NodeAVL v aed ad)

rotateLeft EmptyAVL = EmptyAVL
rotateLeft (NodeAVL v ae (NodeAVL vd ade add))
    = NodeAVL vd (NodeAVL v ae ade) add
```

As rotações são usadas para rebalancear uma árvore após acontecer alguma modificação. Por exemplo, considerando a árvore da Figura 6.5 c) e que a inserção do valor 2 provoque uma violação à propriedade de balanceamento em árvores AVL. Neste caso, uma rotação à direita coloca novamente a árvore na forma AVL, conforme pode ser visto na Figura 6.8.

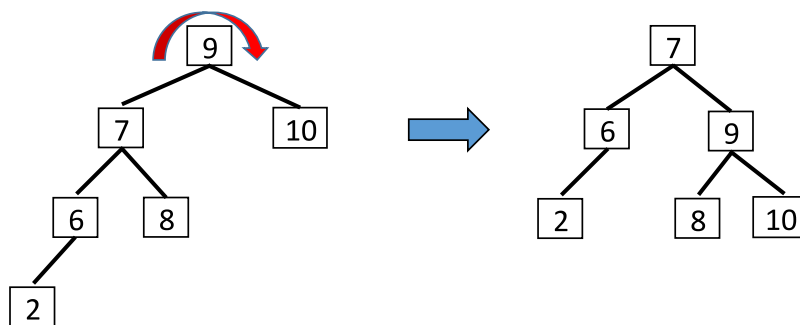


Figura 6.8: Exemplo de uma rotação à direita.

Rotação dupla

Existem situações em que uma rotação simples à direita ou à esquerda não reconstitui a condição de balanço da árvore AVL, ou seja, a árvore resultante de uma rotação, à direita ou à esquerda, não representa uma árvore AVL. Para resolver problemas deste tipo, é necessário que sejam feitas mais de uma rotação e em sentidos opostos na árvore em análise e que podem ser primeiro à esquerda e depois à direita, ou também obedecendo a ordem inversa, ou seja, primeiramente se faz uma rotação à direita e depois mais uma rotação à esquerda. A primeira destas duas rotações de sentidos opostos deve ser realizada sobre a subárvore em que se observa a maior profundidade e a segunda rotação será realizada sobre o nó raiz da árvore resultante desta rotação. Por exemplo, analisemos a árvore mostrada na letra a) da Figura 6.9 que representa a árvore resultante da inserção do valor 6 na árvore AVL inicial. Neste caso, a primeira rotação deverá ser feita sobre o nó com valor 4, ou seja, uma rotação à esquerda, cuja árvore resultante está mostrada na letra b) da mesma figura.

Analisando a árvore da letra b) da Figura 6.9, resultado de uma rotação à esquerda, verifica-se que a condição de balanceamento para árvores AVL não foi restabelecida, ou seja a árvore resultante não é AVL. Assim, se torna necessário que seja feita agora uma outra rotação, agora à direita, para que a condição de balanceamento de árvores AVL seja restabelecida. A árvore resultante destas duas rotações está mostrada na letra c) desta mesma Figura 6.9. Agora pode-se verificar que ela atende à condição de balanceamento de uma árvore AVL, ou seja, as diferenças entre as profundidades de quaisquer subárvores esquerda e direita desta árvore não são maiores que 1.

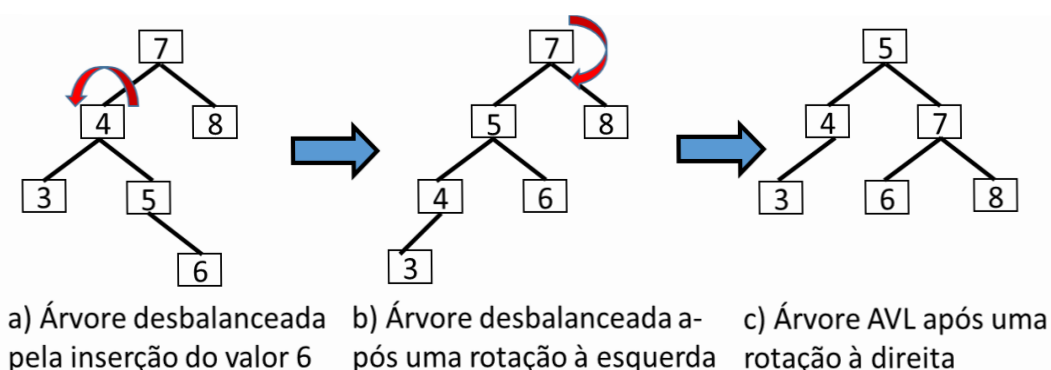


Figura 6.9: Exemplo de uma rotação à esquerda seguida de outra à direita.

As rotações duplas normalmente são mais complexas que as rotações simples, dependendo

do estado da árvore. Uma representação gráfica da rotação dupla direita-esquerda pode ser vista na Figura 6.10, onde pode ser verificada a condição de balanceamento na árvore AVL final.

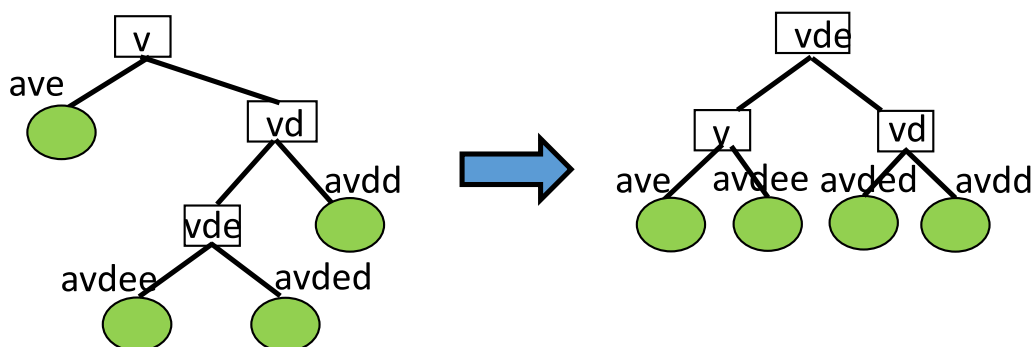


Figura 6.10: Exemplo de uma rotação dupla direita-esquerda.

É importante destacar que o entendimento pleno destas rotações é parte necessária ao domínio e entendimento da codificação em Haskell, salientando ainda que isto pode melhorar o entendimento do leitor sobre o funcionamento pleno das árvores AVLs. Esta observação também pode ser aplicada a outras áreas da Computação, sendo este um fator importante na decisão de escolher e utilizar a Programação Funcional, no mínimo, como uma atividade lateral para o domínio completo do entendimento de outras estruturas e algoritmos importantes utilizados nas diversas áreas da Informática. O leitor é convidado a pensar sobre este tema, verificando e comparando as implementações de estruturas complexas em uma linguagem imperativa e suas correspondentes em uma linguagem funcional, como Haskell.

Uma rotação dupla no sentido oposto ao anterior, agora esquerda-direita, está mostrada na Figura 6.11.

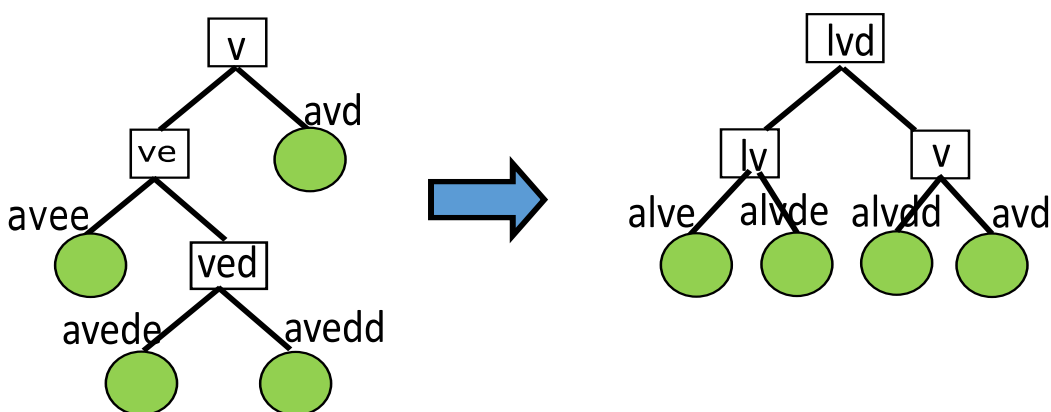


Figura 6.11: Exemplo de uma rotação dupla esquerda-direita.

A codificação em Haskell das rotações duplas podem ser feitas usando a composição das duas rotações simples ou de forma direta, da seguinte forma:

```
rotLeftRight, rotRightLeft :: (Ord a, Show a) => AVLTree a -> AVLTree a

rotLeftRight (NodeAVL v ave (NodeAVL vd (NodeAVL vde avdee avded) avdd))
  = NodeAVL vde (NodeAVL v ave avdee) (NodeAVL vd avded avdd)
```

```

rotRightLeft (NodeAVL v (NodeAVL ve avee (NodeAVL ved avede avedd)) avd)
  = NodeAVL ved (NodeAVL ve avee avede) (NodeAVL v avedd avd)

```

Inserção de valores em árvores AVL

A inserção de um ítem **i** em uma árvore AVL deve ser realizada de forma similar à inserção deste ítem em um nó em uma árvore de busca binária, mas, com as seguintes diferenças:

1. se a inserção for feita na subárvore esquerda e a altura da subárvore esquerda diferir de 2 da altura da subárvore direita, então:
 - (a) se o valor de **i** for menor que o valor da raiz da subárvore esquerda, deve ocorrer uma rotação simples para a direita;
 - (b) senão, deve ocorrer uma rotação dupla esquerda-direita.
2. se a inserção for realizada na subárvore direita e a altura da subárvore direita diferir de 2 sobre a altura da subárvore esquerda, então:
 - (a) se o valor de **i** for maior que o valor da raiz da subárvore direita, deve ser feita uma rotação simples para a esquerda;
 - (b) senão, deve acontecer uma rotação dupla direita-esquerda.

Estas operações podem ser definidas da seguinte forma:

```

profAVL :: (Ord a, Show a) => AVLTree a -> Int
profAVL EmptyAVL = 0
profAVL (NodeAVL _ ae ad) = 1 + max (profAVL ae) (profAVL ad)

insereAVL :: (Ord a, Show a) => a -> AVLTree a -> AVLTree a
insereAVL i EmptyAVL = NodeAVL i EmptyAVL EmptyAVL
insereAVL i (NodeAVL v ae ad)
  | i < v = let newae@(NodeAVL newlae _ _) = insereAVL i ae in
    if ((profAVL newae - profAVL ad) == 2)
    then if i < newlae
      then rotateRight (NodeAVL v newae ad)
      else rotateLeft (NodeAVL v newae ad)
    else (NodeAVL v newae ad)
  | otherwise = let newad@(NodeAVL newrad _ _) = insereAVL i ad in
    if ((profAVL newad - profAVL ae) == 2)
    then if i > newrad
      then rotateLeft (NodeAVL v ae newad)
      else dRotLeftRight (NodeAVL v ae newad)
    else (NodeAVL v ae newad)

```

Deve ser observado que a informação sobre altura deve sempre estar disponível. Uma versão melhorada deve armazenar a informação de altura nos nós.

Exercícios propostos (baseados em Richard Bird [6])

1. Uma forma de construir uma árvore binária a partir de uma lista **[x0; x1; x2; ...]** é feita da seguinte forma: **x0** será o rótulo da raiz da árvore, **x1** e **x2** serão os rótulos dos

filhos, **x3**; **x4**; **x5** e **x6** serão os rótulos dos netos e assim sucessivamente. Esta é uma forma de divisão de uma lista em sublistas cujos tamanhos sejam potências de 2 pode ser implementada através da função **niveis**, definida por

```
niveis :: [t] -> [[t]]
niveis = niveisCom 1
```

Defina a função **niveisCom**.

2. Podemos também definir árvores mais gerais com uma lista arbitrária de subárvores. Por exemplo,

```
data ArvoreGeral t = Folha t | No [ArvoreGeral t]
```

defina funções para:

- (a) contar o número de folhas em uma *ArvoreGeral*,
- (b) encontrar a profundidade de uma *ArvoreGeral*,
- (c) somar os elementos de uma árvore genérica,
- (d) verificar se um elemento está em uma *ArvoreGeral*,
- (e) mapear uma função sobre os elementos das folhas de uma *ArvoreGeral* e
- (f) transformar uma *ArvoreGeral* em uma lista.

6.5 Tratamento de exceções

Para construir bons programas, é necessário especificar o que o programa deve fazer no caso de acontecer algumas situações anômalas. Estas ocorrências são conhecidas como exceções, normalmente indesejáveis, e podem ser de vários tipos: a tentativa de divisão por zero, cálculo de raiz quadrada de número negativo ou a aplicação da função fatorial a um número negativo, tentativa de encontrar a cabeça ou a cauda de uma lista vazia, entre outras. Existem basicamente três técnicas para resolver estes tipos de problemas, conhecidas como técnicas de tratamento de erros.

A solução mais simples é exibir uma mensagem informando o tipo da exceção e parar a execução do programa. Isto pode ser feito através de uma função de erro. Por exemplo, seja a função que calcula a área de um círculo pode ser definida da seguinte forma:

```
area_circulo :: Float -> Float
area_circulo r
  | r >= 0    = pi*r*r
  | otherwise = error "Circulo com raio negativo."
```

Uma chamada a esta função aplicada a um valor negativo, `area_circulo -5`, resultará na mensagem

```
Program error: Circulo com raio negativo.
```

que é mostrada na tela e a execução do programa é interrompida.

Como outro exemplo, pode-se definir uma função, `div_por`, que faz a divisão de um valor inteiro por cada um dos valores de uma lista de inteiros, retornando uma lista de valores inteiros, da seguinte forma:

```
div_por :: (Integral t) => t -> [t] -> [t]
div_por numerador = map (numerador `div`)
```

Uma chamada a esta função, por exemplo `div_por 100 [3,10,20,50]`, resultará na lista `[33,10,5,2]`. No entanto a chamada `div_por 100 [3,10,20,0,50]` resultará em

```
[33,10,5,***Exception = divide by zero
```

A execução da função inicia mostrando a saída até encontrar uma entrada igual a zero. Neste ponto ele emite uma mensagem de erro e para a execução da função.

O problema com esta técnica é que todas as informações usuais da computação calculadas até o ponto em que ocorreu a exceção são perdidas. Em vez disso, o erro pode ser tratado de alguma forma, sem ter que parar a execução do programa. Isto pode ser feito através das duas técnicas a seguir.

6.5.1 Valores fictícios

A função **tail** é construída para retornar a cauda de uma lista finita não vazia e, se a lista for vazia, reportar esta situação com uma mensagem de erro e parar a execução. Ou seja,

```
tail :: [t] -> [t]
tail (a : x) = x
tail [ ] = error "cauda de lista vazia"
```

No entanto, esta definição pode ser refeita da seguinte forma:

```
tl :: [a] -> [a]
tl (_:xs) = xs
tl [ ] = [ ]
```

Desta forma, todas as listas passam a ter uma resposta a uma solicitação de sua cauda, seja ela vazia ou não. Se a lista não for vazia, sua cauda será reportada normalmente. Se a lista for vazia, o resultado será o valor fictício.

De forma similar, a função de divisão de dois números inteiros pode ser feita da seguinte forma, envolvendo o caso em que o denominador seja zero:

```
divide :: Int -> Int -> Int
divide n m
  | (m /= 0) = n `div` m
  | otherwise = 0
```

Para estes dois casos, a escolha dos valores fictícios é óbvia. No entanto, existem casos em que esta escolha não é simples, nem mesmo possível. Por exemplo, na definição de uma função que retorne a cabeça de uma lista vazia. Qual deve ser o valor fictício a ser adotado neste caso?

Em situações como esta, há de se recorrer a um artifício mais elaborado. A solução adotada é redefinir a função **hd**, acrescentando mais um parâmetro.

```
hd :: a -> [a] -> a
hd y (x:_) = x
hd y [ ] = y
```

Esta técnica é mais geral e consiste na definição de uma nova função para o caso de ocorrer uma exceção. Em nosso caso, a função **hd** de um argumento foi modificada para ser aplicada a dois argumentos.

De maneira geral, constrói-se uma função com uma definição para o caso de surgir uma exceção e outra definição para o caso em que ela não ocorra. De forma geral,

```
fErr y x
  | cond = y
  | otherwise = f x
```

Esta técnica funciona bem em muitos casos. O problema é que ela não reporta qualquer mensagem informando a ocorrência da exceção. Para isso, uma abordagem mais elaborada e desejável consiste em processar a entrada indesejável.

6.5.2 Tipos de erros

A técnica anterior para se tratar exceções se baseiam no retorno de valores fictícios, caso elas aconteçam. No entanto, em Haskell, é possível se adotar uma outra abordagem bem mais elaborada, baseando-se no retorno de um valor **erro** como resultado. Isto é feito através do tipo **Maybe**.

```
data Maybe t = Nothing | Just t
  deriving (Eq, Ord, Read, Show)
```

Na realidade **Maybe t** é o tipo **t** com um valor extra, **Nothing**, acrescentado. Vamos redefinir a função de divisão, **divide**, transformando-a em **errDiv** da seguinte forma:

```
errDiv :: Int -> Int -> Maybe Int
errDiv n m
  | (m /= 0) = Just (n `div` m)
  | otherwise = Nothing
```

Para o caso mais geral, utilizando uma função **f**, devemos fazer

```
fErr f x
  | cond = Nothing
  | otherwise = Just (f x)
```

O resultado destas funções agora não são do tipo de saída original, digamos **t**, mas do tipo **Maybe t**. Com esta nova possibilidade, pode-se redefinir a função **div_por** para a seguinte forma:

```
div_por :: (Integral t) => t -> [t] -> [Maybe t]
div_por numerador denominador = map divide denominador
  where divide 0 = Nothing
        divide x = Just (numerador `div` x)
```

Com esta nova definição de **div_por**, a chamada anterior, **div_por 100 [3,10,20,0,50]** terá como resultado **[Just 33, Just 10, Just 5, Nothing, Just 2]**, ou seja, o programa emite o resultado **Nothing** para a divisão por zero e não para a sua execução.

Este novo tipo, **Maybe t**, nos permite processar um erro. Pode-se fazer duas coisas:

1. “repassar” o erro através de uma função, que é o efeito da função **mapMaybe**, a ser vista a seguir, ou
2. “segurar” a exceção, que é o papel da função **maybe**, também definida a seguir.

A função **mapMaybe** repassa o valor de um erro, através da aplicação de uma função **g**. Suponha que **g** seja uma função do tipo **a -> b** e que esteja tentando usá-la como operador sobre um tipo **Maybe a**. Caso o argumento seja **Just x**, **g** será aplicada a **x**, ou seja, **g x** do tipo **b**. Se, no entanto, o argumento for **Nothing**, então **Nothing** é o resultado.

```
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe g Nothing = Nothing
mapMaybe g (Just x) = Just (g x)
```

Para segurar um erro e resolvê-lo, deve-se retornar um resultado do tipo **b**, a partir de uma entrada do tipo **Maybe a**. Neste caso, temos duas situações:

- no caso **Just**, aplica-se a função **g** de **a** para **b**;
- no caso **Nothing**, temos de apresentar o valor do tipo **b** que vai ser retornado.

A função de alta ordem que realiza esta operação é **maybe**, cujos argumentos, **n** e **f**, são usados nos casos **Nothing** e **Just**, respectivamente.

```
maybe :: b -> (a -> b) -> Maybe a -> b
maybe n f Nothing = n
maybe n f (Just x) = f x
```

Vamos acompanhar as funções **mapMaybe** e **maybe** em ação, nos exemplos que seguem. No primeiro deles, a divisão por zero nos retorna **Nothing** que vai sendo “empurrado” para a frente e retornar o valor 56.

```
maybe 56 (1+) (mapMaybe (*3) (errDiv 9 0))
= maybe 56 (1+) (mapMaybe (*3) Nothing)
= maybe 56 (1+) Nothing
= 56
```

No caso em que o divisor é 1, a função retorna **Just 9**. Este resultado é multiplicado por 3 e **maybe**, no nível externo, adiciona 1 e remove o **Just**.

```
maybe 56 (1+) (mapMaybe (*3) (errDiv 9 1))
= maybe 56 (1+) (mapMaybe (*3) (Just 9))
= maybe 56 (1+) (Just 27)
= (1+) 27
= 28
```

A vantagem desta técnica é que pode-se definir o sistema sem a preocupação com os erros e depois adicionar um gerenciamento de erro usando as funções **mapMaybe** e **maybe**, juntamente com as funções modificadas para segurar o erro. Dividir o problema em duas partes facilita a solução de cada uma delas e do todo.

Exercício proposto. Defina uma função **process :: [Int] -> Int -> Int -> Int** de forma que **process l n m** toma o **n**-ésimo e o **m**-ésimo elementos de uma lista **l** e retorna a sua soma. A função deve retornar **zero** se quaisquer dos números **n** ou **m** não forem índices da lista **l**. Para uma lista de tamanho **p**, os índices são: **0, 1, ..., p-1**, inclusive.

6.6 Lazy evaluation

Vamos iniciar nossa discussão avaliando a expressão `quadrado (5 + 7)`, onde a função `quadrado` foi definida no Capítulo 3. Vamos seguir duas sequências de redução possíveis para esta expressão. A primeira delas é a seguinte:

```
quadrado (5 + 7)
= quadrado 12      -- usando a definicao do operador +
= 12 x 12          -- usando a definicao da funcao quadrado
= 144
```

A segunda sequência de redução para a mesma expressão é:

```
quadrado (5 + 7)
= (5 + 7) * (5 + 7) -- pela definicao da funcao quadrado
= 12 * (5 + 7)      -- pela definicao do operador +
= 12 * 12           -- pela definicao do operador +
= 144
```

Estas duas sequências de reduções ilustram duas políticas de escolha de redução, conhecidas como *innermost* e *outermost*, respectivamente. Na primeira delas, em cada passo de redução, foi escolhido o *redex* mais interno, ou seja, o *redex* que não contivesse qualquer outro *redex* interno a ele. Já na segunda sequência de reduções, optou-se pela escolha do *redex* mais externo, ou seja, um *redex* que não estivesse contido em qualquer outro.

Vamos analisar mais um exemplo, utilizando as políticas de sequências de redução, agora sendo aplicadas à expressão `fst (quadrado 5, quadrado 7)`. Usando a política *innermost*, teremos a seguinte sequência:

```
fst (quadrado 5, quadrado 7)
= fst (5 * 5, quadrado 7)  -- usando a definicao da funcao quadrado
= fst (25, quadrado 7)    -- usando a definicao do operador *
= fst (25, 7 * 7)         -- usando a definicao da funcao quadrado
= fst (25, 49)            -- usando a definicao do operador *
= 25                     -- usando a definicao da funcao fst
```

Nesta sequência de reduções, foram utilizados 5 passos até chegar à forma normal. Nos primeiros dois passos, verifica-se que existiam duas possibilidades de escolha: a função **quadrado 5** ou a função **quadrado 7**. Apesar de ambos obedecerem à política *innermost*, a escolha recaiu sobre o *redex* mais à esquerda, conhecido como *leftmost*.

Se, por outro lado, tivéssemos escolhido a política *outermost*, a sequência de reduções seria a seguinte:

```
fst (quadrado 5, quadrado 7)
= quadrado 5      -- usando a definicao da funcao fst
= 5 * 5           -- usando a definicao da funcao quadrado
= 25              -- usando a definicao do operador *
```

Usando esta política de redução, foram utilizados apenas 3 passos até atingir a forma normal da expressão inicial porque a avaliação da função **quadrado 7** foi evitada, apesar de, em ambas as sequências, o resultado ser o mesmo.

Qual a diferença entre estas duas políticas de reduções? Para responder a esta questão, vamos supor que a função **quadrado 7** fosse indefinida na expressão inicial. Neste caso, a primeira sequência de reduções não terminaria, ao passo que a segunda, sim. Isto nos permite inferir que se as duas sequências de reduções atingem à forma normal, elas chegam ao mesmo resultado e que, em alguns casos, a utilização da política *innermost* pode não atingir sua forma normal.

Resumidamente, se uma expressão tiver uma forma normal, ela pode ser computacionalmente atingida utilizando a política *outermost*, ou seja, pode-se construir um programa computacional capaz de encontrar sua forma normal. Por este motivo, a política de reduções *outermost* é também conhecida como ordem normal. O leitor deve recorrer ao Capítulo 2 deste estudo e reanalisar os teoremas de Church-Rosser que ali são aplicados ao λ -cálculo.

Apesar desta característica importante, devemos observar que, no primeiro exemplo mostrado nesta seção, foram necessários 4 passos, enquanto que utilizando a política *innermost* foram necessários apenas 3. No entanto, se na expressão a ser avaliada existirem funções onde algum argumento ocorra de forma repetida é possível que este argumento seja ligado a uma expressão grande e, neste caso, a política de redução *innermost* apresenta um desempenho bem menor.

No capítulo introdutório deste estudo, foi afirmada a utilização de grafos de redução na representação de expressões, em vez de árvores. Nos grafos, as subexpressões de uma expressão podem ser compartilhadas o que não é possível utilizando árvores. Por exemplo, a expressão $(5 + 7) * (5 + 7)$ pode ser representada pelo grafo da Figura 6.12.



Figura 6.12: Representação gráfica da função $*$ em ação.

Neste caso, cada ocorrência da subexpressão $(5 + 7)$ é representada por um arco que é um ponteiro para uma mesma instância de $(5+7)$. A representação de expressões como grafos significa que as subexpressões que ocorrem mais de uma vez são compartilhadas, portanto reduzidas a uma única.

Uma observação importante é que na utilização de grafos de redução a redução *outermost* nunca gasta mais passos que a redução *innermost*. A redução *outermost* em grafos de redução é referenciada normalmente como *lazy evaluation* e a redução *innermost* como *eager evaluation*.

As subexpressões compartilhadas também podem acontecer em definições locais. Por exemplo, seja a definição da função **raizes** a seguir,

```
raizes a b c = ((-b+d)/e, (-b-d)/e)
  where d = sqrt (quadrado b - 4 * a * c)
         e = 2 * a
```

onde o primeiro passo para a redução da aplicação **raizes 1 5 3** pode ser mostrado pelo grafo da Figura 6.13.

Na realidade, existem mais 3 ponteiros, não mostrados na figura, que são os ponteiros para as variáveis a , b e c .

Já sabemos que um operador *lazy* avalia uma operação apenas uma vez e se necessário, ou seja, no máximo uma vez. Isto tem influência na construção de listas, principalmente no manuseio de listas potencialmente infinitas. Para entender estas construções, é necessário compreender bem como o avaliador funciona. Para isto, vamos começar analisando uma função

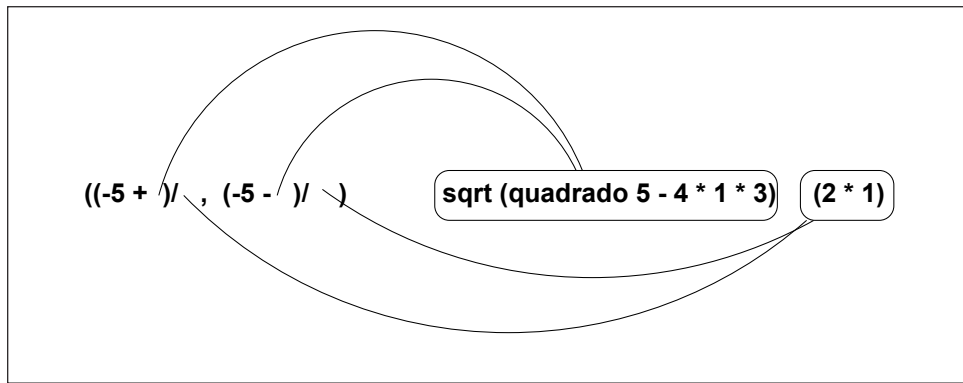


Figura 6.13: Representação gráfica da função raízes em ação.

simples e depois vamos mostrar exemplos mais complexos. Por exemplo, seja o sistema de avaliação da função a seguir:

```
eqFunc1 a b = a + b           --linha 1
eqFunc1 (9 - 3) (eqFunc1 34 (9 - 3)) --linha 2
    = (9 - 3) + (eqFunc1 34 (9 - 3)) --linha 3
    = 6 + (eqFunc1 34 6)           --linha 4
    = 6 + (34 + 6)                 --linha 5
    = 6 + 40                       --linha 6
    = 46                           --linha 7
```

Nas linhas 2 e 3 deste *script*, a subexpressão **(9 - 3)** ocorre duas vezes. Neste caso, o avaliador de Haskell avalia esta subexpressão apenas uma vez e guarda o resultado, na expectativa de que ela seja novamente referenciada. Se isso ocorrer, a subexpressão já está avaliada. Como no λ -cálculo, a ordem de avaliação é sempre da esquerda para a direita, ou seja, *leftmost-outermost*. Na avaliação da função **eqFunc2**, a seguir, o segundo argumento (**eqFunc1 34 3**) não é avaliado porque ele não é necessário para a aplicação da função **eqFunc2**.

```
eqFunc2 a b = a + 32
eqFunc2 (10 * 40) (eqFunc1 34 3)
    = (10 * 40) + 32
    = 400 + 32
    = 432
```

Nas subseções a seguir são mostrados alguns exemplos de como as listas são construídas, levando em consideração o mecanismo de avaliação *lazy*. Em particular, as listas potencialmente infinitas apenas são possíveis por causa da existência deste mecanismo. É interessante observar que algumas definições são de fato bastante criativas, o que torna as linguagens funcionais muito atrativas aos programadores que gostam de utilizar formas inusitadas e criativas na construção de programas.

6.6.1 Expressões ZF (revisadas)

Algumas aplicações que usam a construção de listas por compreensão só podem ser completamente entendidas se conhecermos o sistema de avaliação, principalmente quando envolver a criação de listas potencialmente infinitas.

Uma expressão ZF é uma expressão com a seguinte sintaxe:

$[e \mid q_1, \dots, q_k]$, onde e é uma expressão e cada q_i é um qualificador que tem uma entre duas formas:

1. um gerador: $p \leftarrow lExp$, onde p é um padrão e $lExp$ é uma expressão do tipo lista ou
2. um teste: $bExp$, onde $bExp$ é uma expressão booleana, também conhecida como guarda.

Exemplos:

```
pares :: [t] -> [u] -> [(t,u)]
pares l m = [(a,b) | a <- l, b <- m]
pares [1,2,3] [4,5] = [(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)]

trianRet :: Int -> [(Int, Int, Int)]
trianRet n = [(a,b,c) | a <- [2..n], b <- [a+1..n], c <- [b+1..n],
                      a*a + b*b == c*c]
triRetan 100 = [(3,4,5), (5,12,13), (6,8,10), ..., (65,72,97)]
```

Deve ser observado, neste último exemplo, a ordem em que as triplas foram geradas. Ele pega o primeiro número do primeiro gerador, em seguida o primeiro número do segundo, depois percorre todos os valores do terceiro gerador. Quando todos os elementos do terceiro gerador forem percorridos o mecanismo volta para o segundo gerador e pega agora o seu segundo elemento e vai novamente percorrer todo o terceiro gerador. Após se exaurirem todos os elementos do segundo gerador ele volta para o segundo elemento do primeiro gerador e assim prossegue até o final.

Sequência de escolhas

Para mostrar a sequência de processamento, vamos utilizar a notação do λ -cálculo para β -reduções. Seja e uma expressão, portanto $e\{f/x\}$ é a expressão e onde as ocorrências livres de x são substituídas por f . Formalmente, temos:

$$[e \mid v \in [a_1, \dots, a_n], q_2, \dots, q_k] \\ = [e\{a_1/v\} \mid q_2\{a_1/v\}, \dots, q_k\{a_1/v\}] ++ \dots ++ [e\{a_n/v\} \mid q_2\{a_n/v\}, \dots, q_k\{a_n/v\}]$$

Por exemplo, temos:

$$[(a,b) \mid a < -l]\{[2,3]/l\} = [(a,b) \mid a < -[2,3]] \text{ e} \\ (a + sum(x))\{(2, [3,4])/(a,x)\} = 2 + sum[3,4].$$

Regras de teste

Na utilização desta sintaxe é necessário que algumas regras de aplicação sejam utilizadas. Estas regras são importantes para entender porque algumas aplicações, envolvendo a criação de listas, não chegam aos resultados pretendidos.

$$[e \mid \text{True}, q_2, \dots, q_k] = [e \mid q_2, \dots, q_k] \\ [e \mid \text{False}, q_2, \dots, q_k] = [] \\ [e \mid] = [e]$$

Exemplos resolvidos:

Vamos mostrar a sequência de operações em alguns exemplos, para ficar mais claro:

$$[a + b \mid a <- [1,2], \text{isEven } a, b <- [a..2*a]] \\ = [1 + b \mid \text{isEven } 1, b <- [1..2*1]] ++ [2 + b \mid \text{isEven } 2, b <- [2..2*2]]$$

```

= [1 + b | False, b <- [1..2*1]] ++ [2 + b | True, b <- [2..2*2]]
= [ ] ++ [2 + b |, b <- [2..2*2]]
= [2 + 2 |, 2 + 3 |, 2 + 4 | ]
= [2 + 2, 2 + 3, 2 + 4]
= [4, 5, 6]

```

```

[(a, b) | a <- [1..3], b <- [1..a]]
= [(1,b) | b <- [1..1]] ++ [(2,b) | b <- [1..2]] ++ [(3,b) | b <- [1..3]]
= [(1,1)|] ++ [(2,1)|] ++ [(2,2)|] ++ [(3,1)|] ++ [(3,2)|] ++ [(3,3)|]
= [(1,1), (2,1), (2,2), (3,1), (3,2), (3,3)]

```

Exercício proposto. Faça o cálculo da expressão `[a+b | a<-[1..4], b<-[2..4], a<b]`.

6.6.2 Dados sob demanda

Suponhamos que seja necessário encontrar a soma das quartas potências dos números 1 a n . Os passos a seguir serão necessários para resolver este problema:

- construir a lista `[1..n]`,
- elevar à quarta potência cada número da lista, gerando a lista `[1, 16, ..., n4]` e
- encontrar a soma dos elementos desta lista $= 1 + 16 + \dots + n^4$.

Desta forma a função **somaQuartaPotencia** pode ser definida da seguinte forma:

```

somaQuartaPotencia n = sum (map (^4) [1..n])
= sum (map (^4) (1 : [2..n]))
= sum (((^4) 1) : map (^4) [2..n])
= sum ((1^4) : map (^4) [2..n])
= sum (1 : map (^4) [2..n])
= 1 + sum (map (^4) [2..n])
= 1 + sum (((^4) 2) : map (^4) [3..n])
= 1 + sum ((2^4) : map (^4) [3..n]))
= 1 + sum (16 : map (^4) [3..n])
= 1 + 16 + sum (map (^4) [3..n]) . . .

```

Deve ser observado que a lista não é criada toda de uma só vez. Tão logo uma cabeça seja encontrada, toma-se a sua quarta potência e em seguida é aplicada a soma com algum outro fator que vai surgir em seguida.

6.6.3 Listas potencialmente infinitas

Listas infinitas não têm sido utilizadas na maioria das linguagens de programação. Na realidade, as listas infinitas não podem ser implementadas em qualquer linguagem de programação, uma vez que a memória é finita. O que realmente algumas linguagens implementam, incluindo Haskell, são listas *potencialmente* infinitas, ou seja, listas muito grandes. Mesmo nas linguagens funcionais, elas só podem ser implementadas se a linguagem utilizar *lazy evaluation*. Vejamos um exemplo.

```

uns :: [Int]
uns = 1 : uns           -- a lista infinita de 1's = [1, 1, ..]

somaPrimDoisUns :: [Int] -> Int
somaPrimDoisUns (a : b : x) = a + b

```

Seja agora a chamada a esta função com o argumento **uns**.

```

somaPrimDoisUns uns
= somaPrimDoisUns (1 : uns)
= somaPrimDoisUns (1 : 1 : uns)
= 1 + 1 = 2

```

Apesar de **uns** ser uma lista potencialmente infinita, o mecanismo de avaliação não precisa calcular toda a lista, para depois somar apenas os seus primeiros dois elementos. Desta forma, assim que o mecanismo encontrar estes dois valores, a soma é realizada e a execução da função termina. Mais exemplos, a seguir:

Exemplos:

1. **A lista dos triângulos retângulos.** Seja a lista dos triângulos retângulos, assim definida:
`trigRet n = [(a,b,c) | a <- [2..n], b <- [2..n], c <- [2..n], a*a + b*b == c*c]`. Verifique o que aconteceria se a ordem dos geradores fosse invertida!
2. **O crivo de Eratóstenes.** O crivo de Eratóstenes é uma lista de inteiros criada a partir de uma lista inicial. A lista inicial pode ser uma lista qualquer de inteiros. A partir dela, o primeiro elemento desta lista fará parte da nova lista que consiste deste elemento e do crivo da lista que é feita retirando-se os múltiplos deste valor. Assim, crivo é definido da seguinte forma:

```

crivo :: [Int] -> [Int]
crivo [ ] = [ ]
crivo (x : xs) = x : crivo [y | y <- xs, (mod y x) > 0]

```

3. **A lista dos números primos.** A lista infinita dos números primos pode ser definida a partir do crivo de Eratóstenes definido anteriormente, da seguinte forma:

```

primos :: [Int]
primos = crivo [2..]

```

4. **A lista dos números de Meertens.** Um número natural n é conhecido como *número de Meertens* (referenciado na seção 6.4.6 deste Capítulo) se n for igual ao valor da função de Gödel aplicada aos dígitos de n . Como exemplo, $godel(430) = 432$, porque $432 = 2^4 * 3^3 * 5^0$, sendo $[2, 3, 5]$ a lista de números primos que formam as bases dos expoentes. Construa, em Haskell, uma função que mostre uma lista de números de Meertens.

Vamos construir inicialmente a função **produto** que calcula o produto dos valores de uma lista de inteiros.

```

produto :: [Integer] -> Integer
produto [ ] = 1
produto (x:xs) = x * (produto xs)

```

Depois é necessário definir uma função que dado um número inteiro n retorne a lista de dígitos de n .

```
digitos :: Integer -> [Integer]
digitos n
  | n < 0 = [n]
  | otherwise = digitos (div n 10) ++ [mod n 10]
```

Agora vamos definir a função que implemente a função de Gödel aplicada a um valor inteiro.

```
godel :: Integer -> Integer
godel n = produto (zipWith (^) primos (digitos n))
```

Finalmente, vamos definir a lista de números de Meertens menores a um certo n utilizando as funções **primos** e **crivo** definidas nos itens anteriores.

```
meertens :: Integer -> [Integer]
meertens n = [x|x<-[1..n], x == (godel x)]
```

5. (Baseado em [55]). Dada a lista infinita $[a_0, a_1, a_2, \dots]$, construir a lista infinita $[0, a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots]$. Para resolver este problema, vamos construir a função **somaListas** da seguinte forma:

```
somaListas :: [Int] -> [Int]
somaListas lista = out
  where out = 0 : zipWith (+) lista out

zipWith f (x:xs) (y:ys) = f x y : zipWith f xs ys
zipWit f _ _ = [ ]
```

6.7 Provas sobre tipos algébricos

Com os tipos algébricos também podemos realizar provas sobre a corretude de alguns programas, de forma similar às provas com outros tipos de dados. Por exemplo, reformulando a definição de árvore binária, podemos ter

```
data Arvore t = Nil | No t (Arvore t) (Arvore t)
  deriving (Eq, Ord, Show)
```

Para provar uma propriedade **P(tr)** para todas as árvores finitas do tipo **Arvore t**, temos de analisar dois casos:

1. **o caso Nil:** verificar se **P(Nil)** é verdadeira e
2. **o caso No:** verificar se **P(No x tr1 tr2)** é verdadeira, assumindo que **P(tr1)** e **P(tr2)** são verdadeiras.

Exemplo: provar que **map f (collapse tr) = collapse (mapTree f tr)**. Para provar esta proposição devemos usar as seguintes definições:

`map f [] = []` (1)

`map (a : x) = f a : map f x` (2)

`mapTree f Nil = Nil` (3)

`mapTree f (No x t1 t2)`
`= No (f x) (mapTree f t1) (mapTree f t2)` (4)

`collapse Nil = []` (5)

`collapse (No x t1 t2)`
`= collapse t1 ++ [x] ++ collapse t2` (6)

Esquema da prova:

O caso Nil:

Lado esquerdo	Lado direito
<code>map f (collapse Nil)</code>	<code>collapse (mapTree f Nil)</code>
<code>= map f []</code> por (5)	<code>= collapse Nil</code> por (3)
<code>= []</code> por (1)	<code>= []</code> por (5)

Assim, a propriedade é válida para a árvore do tipo **Nil**.

O caso No:

Para o caso No, temos de provar que `map f (collapse (No x tr1 tr2)) = collapse (mapTree f (No x tr1 tr2))`

assumindo que:

`map f (collapse tr1) = collapse (mapTree f tr1)` (7) e

`map f (collapse tr2) = collapse (mapTree f tr2)` (8)

usando ainda o fato de que `map g (y ++ z) = map g y ++ map g z` (9)

Lado esquerdo	Lado direito
<code>map f (collapse(No x tr1 tr2))</code>	<code>collapse (mapTree f (No x tr1 tr2))</code>
<code>=map f (collapse tr1 ++ [x] ++</code>	<code>= collapse (No (f x)</code>
<code>collapse tr2)</code> por (6)	<code>(mapTree f tr1) (mapTree f tr2))</code> por (4)
<code>=map f (collapse tr1) ++ [f x] ++</code>	<code>=collapse (mapTree f tr1) ++ [f x]</code>
<code>map f (collapse tr2)</code> por (9)	<code>++ collapse(mapTree f tr2)</code> por (6)
<code>=collapse (mapTree f tr1) ++ [f x]</code>	
<code>++ collapse(mapTree f tr2)</code> por (7 e 8)	

Assim, a propriedade também é válida para a árvore do tipo **No**.

Conclusão: como a propriedade é válida para os casos a árvore é **Nil** e também para o caso em que ela é um **No**, então a propriedade é válida para qualquer árvore binária, ou seja, `map f (collapse tr) = collapse (mapTree f tr)`.

6.8 Resumo

Este Capítulo foi dedicado ao estudo de vários temas. No entanto o objetivo maior foi analisar os tipos de dados complexos, em particular, os tipos algébricos de dados. Foi visto como eles

podem ser construídos em Haskell para que o leitor possa segui-los e compreender como eles podem modelar problemas. Na realidade, programar é simular problemas construindo modelos para estes problemas, de forma que estes modelos possam ser processados por um computador, emitindo soluções que possam ser interpretadas como soluções para estes problemas.

Para possibilitar o uso destes tipos de dados, uma gama de ferramentas foram construídas, em Haskell. Entre elas o mecanismo de avaliação *lazy* e as compreensões.

Dadas as possibilidades de construção de tipos que a linguagem oferece, o objetivo do Capítulo foi mostrar uma grande gama de problemas em cujas soluções a linguagem Haskell pode ser aplicada com sucesso.

A grande fonte de exemplos e exercícios mostrados neste Capítulo, foram os livros de Simon Thompson [55] e Richard Bird [6]. Outra fonte importante de problemas a serem resolvidos pode ser o livro de Steven Skiena [44] e os sites:

<http://www.programming-challenges.com> e <http://online-judge.uva.es>.

Para quem deseja conhecer mais aplicações de programas funcionais, o livro de Paul Hudak [15] é uma excelente fonte de estudo, principalmente para quem deseja conhecer aplicações da multimídia usando Haskell.

6.9 Exercícios propostos

1. Suponhamos que as formas geométricas sejam apenas: o círculo, o retângulo e o triângulo. Defina, em Haskell, um tipo algébrico **Forma** e construa funções que calculem a área e o perímetro de valores deste tipo.
2. Suponhamos que um estudante seja: do primeiro, segundo ou do terceiro grau. Defina, em Haskell, um tipo algébrico **Estudante** e construa uma função que calcule uma nota de avaliação que é feita da seguinte forma: a média das avaliações finais dos 4 anos, se ele tiver terminado apenas o primeiro grau, a média das 3 avaliações anuais se ele tiver terminado o segundo grau ou a nota final referente ao seu trabalho final de curso se ele for médico, engenheiro, formado em Computação ou Direito.
3. Usando a definição de profundidade de uma árvore binária, feita anteriormente, prove que, para toda árvore finita de inteiros tr , a *profundidade* $tr < 2^{(profundidade\ tr)}$.
4. Para toda árvore finita de inteiros tr , sendo a função **quantos** definida para verificar quantas vezes um elemento ocorre em uma árvore, vale a equação a seguir. Defina a função **quantos** e mostre a igualdade:


```
quantos tr a = length (filter (==a) (collapse tr))
```
5. Defina, em Haskell, uma função que transforme uma árvore binária de inteiros em uma árvore binária de pares de inteiros, onde o primeiro valor de cada par é o valor inteiro da árvore original e o segundo valor é a altura do nó.
6. Defina em Haskell uma função **delAVL** para remover um item em uma árvore **AVL**.
7. Escreva a definição de uma função **insHeap** que insere um valor diretamente em vez de ser feita através de uma chamada à função **merge**.
8. Seja a seguinte expressão ZF: $x = [a*b \mid a <- [1..4], b <- [1..4], a > b]$. Usando o mecanismo de avaliação lazy de Haskell, calcule, passo a passo, o valor de x .
9. Defina listas infinitas de fatorial e de Fibonacci.

Capítulo 7

Tipos de dados abstratos

“Our ability to decompose a problem into parts depends directly on our ability to glue solutions together. To assist modular programming a language must provide good glue. Functional programming languages provide two new kinds of glue - higher - order functions and lazy evaluation.”
(John Hughes in [17])

7.1 Introdução

No Capítulo introdutório deste livro, foi afirmado que a principal vantagem das linguagens estruturadas era a modularidade oferecida por elas e, por este motivo, as linguagens funcionais eram indicadas como solução para a “crise do software” dos anos 80. De fato, as linguagens funcionais são modulares e Haskell oferece muitas alternativas para a construção de módulos, obedecendo às exigências preconizadas pela Engenharia de Software para a construção de programas.

Para John Hughes [17], a modularidade é uma característica que as linguagens de programação devem apresentar para que sejam utilizadas com sucesso, na construção de grandes sistemas. Ele afirma que esta é a característica principal das linguagens funcionais, proporcionada através das funções de alta ordem e do mecanismo de avaliação *lazy*.

A modularidade é importante porque:

- as partes de um sistema podem ser construídas, compiladas e testadas separadamente e
- podem-se construir grandes bibliotecas compostas de funções genéricas, aumentando a reusabilidade.

No entanto algumas preocupações devem ser levadas em consideração para que estes benefícios sejam de fato auferidos. Caso contrário, em vez de facilitar, a modularização pode se tornar inoportuna. Entre os cuidados preconizados pela Engenharia de Software em relação aos módulos, são citados:

- cada módulo deve ter um papel claramente definido,
- cada módulo deve realizar exatamente uma única tarefa,
- cada módulo deve ser autocontido,

- cada módulo deve exportar apenas o que é estritamente necessário e
- os módulos devem ser pequenos.

7.2 Módulos em Haskell

Nesta seção, é feito um estudo rápido sobre a utilização de módulos em Haskell. O objetivo é mostrar as muitas possibilidades que a linguagem oferece. A prática da programação em módulos é que vai dotar o programador da experiência e habilidade necessárias para o bom desenvolvimento de programas.

Cada módulo em Haskell constitui um arquivo. Isto significa que não devem existir um arquivo em Haskell que contenha mais de um módulo. Isto se deve ao fato de que o nome do arquivo tem de ser o mesmo do módulo, ambos iniciados com letra maiúscula. Por exemplo, para declarar o módulo **Formiga**, devemos construir o arquivo **Formiga.hs** ou **Formiga.lhs** que deve conter as seguintes declarações:

```
module Formiga
  ( Formigas(..),
    comeFormiga,
    mataFormiga
  ) where
```

onde **Formiga** é o módulo, **Formigas** é um tipo de dado e **comeFormiga** e **mataFormiga** são funções a serem utilizadas dentro do módulo.

Os módulos podem ser exportados e importados por outros módulos ou programas. Estas importações e exportações devem ter algum tipo de controle porque nem tudo deve ser exportável sem a aquiescência do construtor do módulo e o importador também deve ter o controle de importar apenas o que ele acha necessário.

Após o nome do módulo segue uma lista de componentes do módulo, entre parênteses, e que são visíveis por outros módulos e, portanto, podem ser exportados, seguidos pela palavra chave **where** indicando que vai iniciar o corpo do módulo. Isto permite que o código possa ser mantido escondido do mundo externo. A notação especial **(..)** que segue o nome **Formigas** indica que tanto o tipo quanto o construtor do tipo podem ser exportados. Na realidade, o nome do tipo ou seja, o construtor do tipo, pode ser exportado mas, os construtores dos valores não podem. Essa característica é importante porque permite esconder os detalhes da implementação de seus usuários, tornando o tipo realmente abstrato.

Se os elementos a serem exportados, juntamente com os parênteses que os cercam, forem omitidos na declaração do módulo, todos os nomes no módulo podem ser exportados. Uma declaração deste tipo pode ser:

```
module ExportaTudo where
```

Se, ao contrário, se deseja que nenhum nome do módulo seja exportado, o que normalmente não é usual, pode-se escrever os parênteses sem nada dentro deles. Por exemplo,

```
module ExportaNada () where
```

7.2.1 Controles de exportação

Por *default*, tudo o que é declarado em um módulo, e apenas isto, pode ser exportado, ou seja, o que é importado a partir de outro módulo não pode ser exportado pelo módulo importador. Esta regra pode ser exageradamente permissiva porque pode ser que algumas funções auxiliares não devam ser exportadas e, por outro lado, pode ser muito restritiva porque pode existir alguma situação em que seja necessário exportar algumas definições declaradas em outros módulos.

Desta forma, deve-se poder controlar o que deve e o que não deve ser exportado. Suponhamos o módulo **Formiga** definido da seguinte forma:

```
module Formiga where
  data Formigas = ...
  comeFormiga x = ...
```

A palavra **module** dentro dos parênteses significa que tudo dentro do módulo é exportado, ou seja, **module Abelha where** é equivalente a **module Abelha (module Abelha) where**.

7.2.2 Importação de módulos

Os módulos, em Haskell, podem importar dados e funções de outros módulos da seguinte forma:

```
module Abelha where
  import Formiga
  pegadordeAbelha = . . .
```

As definições declaradas no módulo **Formiga** podem ser utilizadas no módulo **Abelha**. Diz-se que as definições feitas no módulo **Formiga** são visíveis no módulo **Abelha**.

```
module Vaca where
  import Abelha
```

Neste caso, as definições do tipo **Formigas** e da função **comeFormiga** não são visíveis em **Vaca**. Elas podem se tornar visíveis pela importação explícita de **Formiga** ou usando os controles de exportação para modificar o que é exportado a partir de **Abelha**.

7.2.3 Controles de importação

Além dos controles para exportação, Haskell também apresenta controles para a importação de definições.

```
module Tall where
  import Formiga (Formigas (. . .))
```

Neste caso, a intenção é importar apenas o tipo **Formigas** do módulo **Formiga**. Haskell também provê uma forma explícita de esconder alguma entidade. Por exemplo,

```
module Tall where
  import Formiga hiding (comeFormiga)
```

Nesta declaração, a intenção é esconder a função **comeFormiga**. Se em um módulo existir um objeto com o mesmo nome de outro objeto, definido em um módulo importado, pode-se acessar ambos objetos usando um nome qualificado. Por exemplo, **Formiga.urso** é o objeto importado do módulo **Formiga** e **urso** é o objeto definido localmente. Um nome qualificado é construído a partir do nome de um módulo e do nome do objeto neste módulo. Para usar um nome qualificado, é necessário que se faça uma importação:

```
import qualified Formiga
```

No caso dos nomes qualificados, pode-se também estabelecer quais itens vão ser exportados e quais serão escondidos. Também é possível usar um nome local para renomear um módulo importado, como em

```
import Inseto as Formiga
```

7.3 Modularidade e abstração de dados

Já foi explicado anteriormente neste livro que, até alguns tempos atrás, os programas de computador eram pequenos e ficaram conhecidos na literatura como *programming in the small*. No entanto, a evolução dos computadores permitiu a construção de programas mais complexos e com alguns milhares de linhas de códigos, que ficaram conhecidos como *programming in the large*. Apesar de desejável e até necessário, este tipo de construção de *software* trouxe consigo a necessidade de uma organização eficaz e eficiente, para que os programas pudessem ser mantidos de forma intelectualmente administráveis.

A solução adotada foi organizar estes programas em recipientes sintáticos contendo grupos de subprogramas e dados que estivessem logicamente relacionados, que ficaram conhecidos na literatura como “módulos”.

Além disso, a compilação de programas deixou de ser insignificante porque foi necessário encontrar uma forma de recompilar apenas as partes de um programa que sofressem alguma alteração, sem ter que recompilar todo o programa. Os módulos deveriam ter a possibilidade de serem recompilados separadamente, ficando caracterizadas as “unidades de compilação”, encapsuladas em um agrupamento de subprogramas juntamente com os dados que eles manipulam.

7.3.1 O que é mesmo um tipo abstrato de dados?

Um tipo de dado abstrato é um encapsulamento que inclui a representação do tipo de dado em questão e as operações sobre eles, ocultando os detalhes de implementação, permitindo que sejam criadas instâncias deste tipo, chamadas “objetos”. Desta forma, um objeto é um tipo de dado abstrato que satisfaz as duas condições seguintes:

1. a representação do tipo e as operações permitidas sobre eles estão contidas em uma única unidade sintática e
2. a representação do tipo não é visível pelas unidades que o usam, ou seja, as únicas operações acessíveis diretamente são as oferecidas na definição do tipo.

Este tipo de organização permite que os programas sejam divididos em unidades lógicas que podem ser compiladas separadamente, possibilitando que sejam feitas modificações na representação e nas operações do tipo, sem que isto implique na recompilação de um programa que

seja cliente deste tipo de dado. Ocultando a representação e as operações aos clientes, permite que apenas o implementador possa trocar esta representação e operações, e isto significa “confiabilidade”.

Nas seções a seguir, serão feitas considerações relacionadas aos tipos abstratos de dados e as formas de implementação destes tipos em Haskell. Estas implementações são analisadas quanto aos desempenhos que elas apresentam. Para cada tipo abstrato de dados definido, serão analisadas versões com diferentes desempenhos.

7.4 Eficiência de programas funcionais

Haskell, como linguagem funcional, provê um nível de abstração bem mais alto que as linguagens imperativas. Este fato, implica que os programas funcionais sejam bem menores, em termos de tamanho de código, mais claros e mais rápidos de serem desenvolvidos que os programas imperativos. Isto contribui para um melhor entendimento do algoritmo que está sendo implementado, além de possibilitar que soluções alternativas sejam exploradas mais rapidamente.

Já foi fortemente mencionado que Haskell usa o mecanismo de avaliação *lazy* como *default*. No entanto, a linguagem também dispõe de um mecanismo para que a avaliação de funções seja feita de forma estrita, conhecida como *eager evaluation*, ou avaliação gulosa. Esta metodologia de passagem de parâmetros é conhecida como passagem por valor, o que torna os programas bastante rápidos e eficientes. O problema é que a avaliação gulosa pode levar, em alguns casos, a um *loop* infinito, fazendo com que o programa nunca páre. Isto é garantido pelo teorema de Church-Rosser, já analisado nos capítulos 1 e 2. Já com a avaliação preguiçosa, se o programa tiver uma forma normal ela será encontrada usando esta técnica de avaliação. No entanto, esta forma de avaliação pode exigir mais passos de execução que a avaliação gulosa. Alguns compiladores utilizam um mecanismo de verificação da adequação deste método em funções. Este mecanismo é conhecido como *strictness analysing* [37].

O problema é que a análise de desempenho pode dar resultados diferentes se for usada a avaliação estrita ou a avaliação *lazy*. A análise de desempenho em programas estritos é relativamente fácil, mas em programas *lazy* não tem se mostrado uma tarefa simples. Por este motivo, a análise da complexidade das funções será feita de forma superficial e sem muita profundidade, dada a dificuldade do tema e não ser este o objetivo deste estudo.

7.4.1 O desempenho da função `reverse`

A função **`reverse`** inverte a ordem dos elementos de uma lista de qualquer tipo. Ela é predefinida em Haskell, mas pode ser facilmente definida por

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

A despeito da simplicidade desta definição, ela padece de um sério problema de desempenho. Verifiquemos quantos passos são necessários para inverter uma lista *l* de *n* elementos, usando esta definição. A função chama a si mesma *n* vezes e, em cada uma destas chamadas, chama as funções **`++`**, **`tail`** e **`head`**. As funções **`head`** e **`tail`** são primitivas, ou seja, exigem tempo constante para suas execuções. A operação **`++`** deve saltar para o final de seu primeiro argumento para concatená-lo com o segundo. O tempo total de **`reverse`** é dado pela soma

$$(n-1) + (n-2) + (n-3) + \dots + 1 + 0 = \sum_{i=0}^{n-1} i = \frac{n(n+1)}{2} \Rightarrow \mathcal{O}(n^2)$$

Assim, **reverse** precisa de um tempo proporcional ao quadrado do tamanho da lista a ser invertida. Será que existe outra forma de implementar **reverse** com um desempenho melhor? A resposta é sim, ou seja, é possível implementá-la em um tempo proporcional ao tamanho da lista. Vejamos como isto pode ser feito.

Imagine que tenhamos alguns livros colocados um em cima do outro sobre uma mesa, formando uma pilha de livros que será denotada para nosso raciocínio por **pa** (pilha antiga). Suponhamos que se deseja inverter a ordem de **pa**. Para isto, é necessário retirar o livro que se encontra no topo de **pa** e colocá-lo ao lado, iniciando uma nova pilha de livros que será denotada por **pn** (pilha nova). Este processo deve ser continuado, ou seja, retirando mais um livro do topo da pilha **pa** e colocando-o no topo da pilha **pn**, até que **pa** fique vazia. Neste instante, a nova pilha, **pn**, conterá os mesmos livros que existiam na pilha anterior mas, em ordem inversa.

Este mesmo raciocínio será aplicado na implementação da função **reverse**. Serão usadas duas listas, **la** (lista antiga) e **ln** (lista nova), para simular as pilhas de livros. É necessário também definir uma função auxiliar, **revaux**, que quando aplicada às duas listas, **la** e **ln**, retira o elemento da cabeça da primeira, **la**, e o coloca como cabeça da segunda, **ln**. A função **revaux** termina sua execução quando **la** ficar vazia. Neste ponto, o resultado será a segunda lista, **ln**. Em Haskell, esta definição é feita da seguinte forma:

```
revaux :: [t] -> [t] -> [t]
revaux (a:la) ln = revaux la (a:ln)
revaux [ ] ln = ln
```

A nova definição da função **reverse** será feita aplicando a função **revaux** à lista a ser invertida, tendo como segundo argumento a lista vazia, ou seja:

```
reverse :: [t] -> [t]
reverse l = revaux l [ ]
```

A análise da complexidade desta definição informa que ela é proporcional ao tamanho da lista, ou seja, $\mathcal{O}(n)$ [30, 39, 11]. Usando uma idéia simples, foi possível melhorar o desempenho da definição original. A dificuldade foi descobrir esta idéia. Estas idéias sempre existem e a programação em Haskell, ou em qualquer outra linguagem funcional, privilegia os programadores capazes de encontrá-las.

Vamos agora analisar alguns tipos abstratos de dados, verificando algumas implementações simples e como elas podem ser melhoradas utilizando idéias simples como a utilizada para a otimizar o desempenho da função **reverse**. Grande parte do material aqui descrito é baseado no excelente livro de Richard Bird [6].

7.5 Implementação de tipos abstratos de dados em Haskell

O objetivo desta seção é introduzir os *tipos abstratos de dados* (**TAD**) e o mecanismo provido por Haskell para definí-los. De maneira geral, os tipos abstratos de dados diferem dos introduzidos por uma declaração **data**, no sentido de que os **TADs** podem escolher a representação de seus valores. Cada escolha de representação conduz a uma implementação diferente do tipo de dado [55].

Como já foi visto, os tipos abstratos são definidos de forma diferente dos tipos algébricos. Um tipo abstrato não é definido pela nomeação de seus valores, mas pela nomeação de suas operações. Isto significa que a representação dos valores dos tipos abstratos não é conhecida. O que é realmente de conhecimento público é o conjunto de funções para manipular valores do tipo. Por exemplo, **Float** é um tipo abstrato em Haskell. Para ele, são exibidas operações de comparação e aritméticas além de uma forma de exibição de seus valores, mas não se estabelece como tais números são representados pelo sistema de avaliação de Haskell, proibindo o uso de casamento de padrões. Em geral, o programador que usa um tipo abstrato não sabe como seus elementos são representados. Tais tipos de abstração são usuais quando mais de um programador estiverem trabalhando em um mesmo projeto ou mesmo quando um mesmo programador estiver trabalhando em um projeto não trivial. Isto permite que a representação seja trocada sem afetar a validade dos *scripts* que usam o tipo abstrato. Vamos mostrar estes fundamentos através de exemplos de implementação de alguns tipos abstratos.

7.5.1 O tipo abstrato Pilha

Vamos dar início ao estudo dos tipos abstratos de dados através da implementação do tipo **Pilha**. As pilhas são estruturas de dados homogêneas onde os valores são colocados e/ou retirados utilizando uma estratégia *LIFO* (*Last In First Out*). Informalmente, esta estrutura de dados é comparada a uma pilha de pratos, na qual só se retira um prato por vez e sempre o que está no topo da pilha. Também só se coloca um prato por vez e em cima do prato que se encontra no topo.

As operações¹ necessárias para o funcionamento de uma pilha do tipo **Stack**, juntamente com seus tipos, formam a assinatura do tipo abstrato. Neste caso, a assinatura é composta das seguintes funções:

<code>push :: t -> Stack t -> Stack t</code>	--coloca um item no topo da pilha
<code>pop :: Stack t -> Stack t</code>	--retira um item do topo da pilha
<code>top :: Stack t -> t</code>	--pega o item do topo da pilha
<code>stackEmpty :: Stack t -> Bool</code>	--verifica se a pilha eh vazia
<code>newStack :: Stack t</code>	--cria uma pilha vazia

Serão feitas duas implementações do tipo pilha, sendo a primeira baseada em tipos algébricos e a segunda baseada em listas.

Primeira implementação para o TAD Pilha

A implementação de um tipo abstrato de dados em Haskell é feita através da criação de um módulo, tema estudado na seção anterior. O leitor deve voltar a ele se tiver alguma dificuldade de entender as declarações aqui feitas. Vejamos a primeira implementação, baseada em tipos algébricos.

```
module Stack(Stack, push, pop, top, stackEmpty, newStack) where
  push :: t -> Stack t -> Stack t
  pop  :: Stack t -> Stack t
  top  :: Stack t -> t
  stackEmpty :: Stack t -> Bool
  newStack :: Stack t
```

¹Apesar de preferirmos nomes em Português, nestes exemplos, as operações serão descritas com os nomes em inglês por já fazerem parte do jargão da Computação.

```

data Stack t = EmptyStk
              | Stk t (Stack t)

push x s = Stk x s

pop EmptyStk = error "retirada em uma pilha vazia"
pop (Stk _ s) = s

top EmptyStk = error "topo de uma pilha vazia"
top (Stk x _) = x

newStack = EmptyStk

stackEmpty EmptyStk = True
stackEmpty _ = False

instance (Show t) => Show (Stack t) where
    show (EmptyStk) = "#"
    show (Stk x s) = (show x) ++ "|" ++ (show s)

```

A utilização do tipo abstrato **Stack** deve ser feita em outros módulos cujas funções necessitem deste tipo de dado. Desta forma, o módulo **Stack** deve constar na relação dos módulos importados pelo programa usuário. Por exemplo, o script a seguir:²,

```

import Stack

listaParaPilha :: [t] -> Stack t
listaParaPilha [ ] = newStack
listaParaPilha (x : xs) = push x (listaParaPilha xs)

pilhaParaLista :: Stack t -> [t]
pilhaParaLista s
    | stackEmpty s = [ ]
    | otherwise = (top s) : (pilhaParaLista (pop s))

```

O *script* deve ser salvo em um arquivo <nome>.hs e deve ser carregado para poder ser executado. Agora podem ser construídos exemplos de algumas instâncias do tipo abstrato **Stack**, da seguinte forma:

```

ex1 = push 14 (push 9 (push 19 (push 51 newStack)))
ex2 = push "Dunga" (push "Constantino")

```

A implementação mostrada foi baseada no tipo algébrico **Stack t**. No entanto, o implementador pode desenvolver uma outra implementação, mais eficiente que esta, e fazer a mudança da implementação anterior para esta, sem que os usuários precisem saber disto. Neste caso, as funções do módulo **Main** não precisam ser refeitas para serem utilizadas. O que não pode ser modificada é a interface com o usuário. Vamos construir a segunda implementação baseada em listas finitas.

²O leitor deve ler o Apêndice B para verificar como um arquivo em Haskell pode ser compilado para ser gerado um arquivo executável juntamente com algumas diretivas de compilação e algumas ferramentas de Profile. Para uma compreensão mais abrangente e que contenha todas possibilidades de uso, é aconselhável consultar o Manual de utilização que se encontra na página oficial de Haskell.

Segunda implementação para o TAD pilha

As operações primitivas (assinatura) serão as mesmas da implementação anterior, ou seja, o que muda é apenas a implementação.

```
module Stack(Stack, push, pop, top, stackEmpty, newStack) where
  push :: t -> Stack t -> Stack t
  pop  :: Stack t -> Stack t
  top  :: Stack t -> t
  stackEmpty :: Stack t -> Bool
  newStack :: Stack t

  data Stack t = Stk [t]

  push x (Stk xs) = Stk (x : xs)

  pop (Stk [ ]) = error "retirada em pilha vazia"
  pop (Stk (_ : xs)) = Stk xs

  top (Stk [ ]) = error "topo de pilha vazia"
  top (Stk (x : _)) = x

  newStack = Stk [ ]

  stackEmpty (Stk [ ]) = True
  stackEmpty _ = False

  instance (Show t) => Show (Stack t) where
    show (Stk [ ]) = "#"
    show (Stk (x : xs)) = (show x) ++ "|" ++ (show (Stk xs))
```

O módulo **Main** é o mesmo para as duas implementações, não sendo necessário ser mostrado novamente.

7.5.2 O tipo abstrato de dado Fila

Vamos agora mostrar um segundo exemplo de tipo abstrato, o tipo **Fila**. Uma fila é uma estrutura de dados do tipo *FIFO* (*First-In First-Out*), onde os elementos só podem ser inseridos em um lado e só podem ser retirados do outro (se existir algum elemento na fila), imitando uma fila de espera, por exemplo, em um Banco. Podemos implementar o tipo **Fila** através de listas finitas, fazendo algumas restrições nas operações. Para um usuário comum, é importante uma implementação eficiente, mas ele não está interessado como ela é feita, normalmente solicitando a outra pessoa que a faça ou utiliza uma solução provida pelo sistema.

Da mesma forma feita para o tipo abstrato **Pilha**, precisamos conhecer que operações primitivas são necessárias para compor a assinatura do tipo abstrato **Queue t**. Neste caso, são elas:

```
enqueue :: t -> Queue t -> Queue t    --coloca um item no fim da fila
dequeue :: Queue t -> Queue t          --retorna a fila sem o item da frente
front   :: Queue t -> t                --pega o item da frente da fila
queueEmpty :: Queue t -> Bool          --testa se a fila esta vazia
newQueue :: Queue t                   --cria uma nova fila vazia
```


Vamos continuar o exemplo provendo duas implementações do tipo abstrato **Fila**, da mesma forma feita para o tipo **Pilha**.

Primeira implementação para o TAD Fila

A primeira implementação a ser mostrada é baseada em uma lista finita. Neste caso, os valores são colocados no final da lista e são retirados da cabeça da lista. A implementação do módulo **Queue** com suas operações é feita da seguinte forma:

```
module Queue (Queue, enqueue, dequeue, front, queueEmpty, newQueue) where
  enqueue :: t -> Queue t -> Queue t
  dequeue :: Queue t -> Queue t
  front :: Queue t -> t
  queueEmpty :: Queue t -> Bool
  newQueue :: Queue t

  data Queue t = Fila [t]

  enqueue x (Fila q) = Fila (q ++ [x]) --insere no final da lista

  dequeue (Fila (x : xs)) = Fila xs      --retira o elemento da cabeça da lista
  dequeue _ = error "Fila de espera vazia"

  front (Fila (x : _)) = x --o front eh o elemento a ser retirado
  front _ = error "Fila de espera vazia"

  queueEmpty (Fila [ ]) = True
  queueEmpty _ = False

  newQueue = (Fila [ ])

  instance (Show t) => Show (Queue t) where
    show (Fila [ ]) = ">"
    show (Fila (x : xs)) = "<" ++ (show x) ++ (show (Fila xs))
```

Para utilizar o *TAD Fila* é necessário construir o módulo **Main**, de forma similar a que foi feita para a utilização do *TAD Pilha*.

```
module Main where
  import Stack
  import Queue

  filaParaPilha :: Queue t -> Stack t --transforma uma fila em uma pilha
  filaParaPilha q = qts q newStack
    where qts q s
      | queueEmpty q = s
      | otherwise = qts (dequeue q) (push (front q) s)

  pilhaParaFila :: Stack t -> Queue t --transforma uma pilha em uma fila
  pilhaParaFila s = stq s newQueue
    where stq s q
      | stackEmpty s = q
      | otherwise = stq (pop s) (enqueue (top s) q)
```

```

inverteFila :: Queue t -> Queue t
inverteFila q = pilhaParaFila (filaParaPilha q)

invertePilha :: Stack t -> Stack t
invertePilha s = filaParaPilha (pilhaParaFila s)

```

Agora podemos também criar exemplos de filas da seguinte forma:

```
q1 = enqueue 14 (enqueue 9 (enqueue 19 newQueue))
```

Segunda implementação do TAD Fila

A segunda implementação do *TAD Fila* também é baseada em listas finitas, mas leva em consideração o desempenho da implementação. Para isto, nada é necessário modificar na assinatura do *TAD*, no entanto, as implementações de algumas funções devem ser modificadas para refletir a nova estrutura.

Recordemos que, na implementação anterior, a inserção de dados era feita na cauda da lista, enquanto a remoção de um dado era feita na cabeça da lista. Desta forma, a operação de remoção era uma operação muito “barata” por envolver apenas um acesso à cabeça da lista, enquanto a operação de inserção poderia ser “cara” porque era necessário percorrer toda a lista para inserir um valor e a lista pode ser grande. A complexidade desta operação é diretamente proporcional ao tamanho da lista a ser percorrida.

Esta estratégia pode ser invertida, ou seja, podemos inserir os dados na cabeça da lista e retirá-los da cauda. Neste caso, os desempenhos das duas operações se invertem: a operação de inserção, que era “cara”, ficaria “barata”, enquanto a operação de remoção, que era “barata”, se tornaria “cara”. Isto significa que a adoção desta nova estratégia em nada adiantaria, em termos de melhoria de desempenho.

No entanto, podemos juntar estas duas técnicas em uma só, ou seja, podemos implementar o tipo **Fila** usando duas listas: a primeira para fazermos remoção na cabeça e a segunda para fazermos inserção, também na cabeça. Se a primeira lista se tornar vazia, a segunda lista será invertida e irá tomar o lugar da primeira lista e a segunda lista passa a ser a lista vazia. Desta forma a operação de remoção pode continuar a ser feita na cabeça da lista sem qualquer problema, até que as duas listas sejam ambas vazias e a operação de inserção continua a ser feita na cabeça da segunda lista. Desta forma, as duas operações de inserção e remoção serão feitas nas cabeças das listas e, conforme já foi afirmado, são operações baratas. A nova implementação fica da seguinte forma:

```

module Queue (Queue, enqueue, dequeue, front, queueEmpty, newQueue) where
  enqueue :: t -> Queue t -> Queue t
  dequeue :: Queue t -> Queue t
  front :: Queue t -> t
  queueEmpty :: Queue t -> Bool
  newQueue :: Queue t

  data Queue t = Fila [t] [t]

  newQueue = Fila [ ] [ ]

  queueEmpty (Fila [ ] [ ]) = True
  queueEmpty _ = False

```

```

front (Fila [ ] [ ]) = error "A fila esta vazia"
front (Fila (x:xs) _) = x
front (Fila [ ] ys) = front (Fila (reverse ys) [ ])

enqueue y (Fila [ ] [ ]) = (Fila [y] [ ])
enqueue y (Fila [ ] ys) = (Fila [ ] (y : ys))
enqueue y (Fila (x : xs) ys) = (Fila (x : xs) (y : ys))

dequeue (Fila (x : xs) ys) = (Fila xs ys)
dequeue (Fila [ ] [ ]) = error "A fila esta vazia"
dequeue (Fila [ ] ys) = (Fila (tail (reverse ys)) [ ])

```

Esta implementação é substancialmente mais eficiente que a implementação feita através de lista única. O módulo de utilização pode ser o mesmo anterior, sem qualquer alteração, uma vez que a implementação é totalmente transparente para os usuários.

7.5.3 O tipo abstrato Set

O tipo abstrato **Set** é uma coleção homogênea de elementos e implementa a noção de conjunto, de acordo com a seguinte interface:

```

emptySet :: Set t                --cria um conjunto vazio
setEmpty :: Set t -> Bool        --testa se o conjunto eh vazio
inSet :: (Eq t) => t -> Set t -> Bool --testa se um x estah em S
addSet :: (Eq t) => t -> Set t -> Set t --coloca um item no conjunto
delSet :: (Eq t) => t -> Set t -> Set t --remove um item de um conjunto
pickSet :: Set t -> t            --seleciona um item de S

```

É necessário testar a igualdade entre os elementos de um conjunto. Por este motivo, os elementos do conjunto têm de pertencer à classe **Eq**. Existem algumas implementações de **Set** que exigem restrições adicionais sobre os elementos do conjunto. Isto significa que pode-se construir uma interface mais rica para **Set** incluindo as operações de *união*, *interseção* e *diferença* de conjuntos, no entanto estas operações adicionais podem ser construídas a partir das operações definidas na interface aqui mostrada.

```

module Set (Set, emptySet, setEmpty, inSet, addSet, delSet, pickSet) where
  emptySet :: Set t
  setEmpty :: Set t -> Bool
  inSet :: (Eq t) => t -> Set t -> Bool
  addSet :: (Eq t) => t -> Set t -> Set t
  delSet :: (Eq t) => t -> Set t -> Set t
  pickSet :: Set t -> t

  data Set t = S [t]      --listas sem repeticoes

  emptySet = S [ ]

  setEmpty (S [ ]) = True
  setEmpty _       = False

```

```

inSet _ (S [ ]) = False
inSet x (S (y : ys))
  | x == y      = True
  | otherwise   = inSet x (S ys)

addSet x (S s)
  | (elem x s) = S s
  | otherwise  = S (x : s)

delSet x (S s) = S (delete x s)

delete x [ ] = [ ]
delete x (y : ys)
  | x == y      = delete x ys
  | otherwise   = y : (delete x ys)

pickSet (S [ ])      = error "conjunto vazio"
pickSet (S (x : _)) = x

```

7.5.4 O tipo abstrato Tabela

Uma tabela, **Table a b**, é uma coleção de associações entre chaves, do tipo **a**, com valores do tipo **b**, implementando assim, uma função finita, com domínio em **a** e codomínio **b**, através de uma determinada estrutura de dados.

O tipo abstrato **Table** pode ter a seguinte implementação:

```

module Table (Table, newTable, findTable, updateTable, removeTable) where
  newTable :: Table a b
  findTable :: (Ord a) => a -> Table a b -> Maybe b
  updateTable :: (Ord a) => (a, b) -> Table a b -> Table a b
  removeTable :: (Ord a) => a -> Table a b -> Table a b

  data Table a b = Tab [(a,b)]          --lista ordenada de forma crescente

  newTable = Tab [ ]

  findTable _ (Tab [ ]) = Nothing
  findTable x (Tab ((c,v) : resto))
    | x < c  = Nothing
    | x == c = Just v
    | x > c  = findTable x (Tab resto)

  updateTable (x, z) (Tab [ ]) = Tab [(x, z)]
  updateTable (x, z) (Tab ((c,v) : resto))
    | x < c  = Tab ((x,z):(c,v):resto)
    | x == c = Tab ((c,z):resto)
    | x > c  = let (Tab t) = updateTable (x,z) (Tab resto)
                in Tab ((c,v):t)

  removeTable _ (Tab [ ]) = Tab [ ]

```

```

removeTable x (Tab ((c,v):resto))
  | x < c = Tab ((c,v):resto)
  | x == c = Tab resto
  | x > c = let (Tab t) = removeTable x (Tab resto)
            in Tab ((c,v):t)

instance (Show a, Show b) => Show (Table a b) where
  show (Tab [ ]) = " "
  show (Tab ((c,v):resto))
    = (show c)++"\t"++(show v)++"\n"++(show (Tab resto))

```

Como pode ser observado, o *TAD Tabela* foi implementado usando uma lista de pares (chave, valor) ordenada em ordem crescente pelas chaves. O módulo **Main** para este *TAD* pode ser implementado da seguinte forma:

```

module Main where
  import Table

  type Numero = Integer
  type Nome = String
  type Data = Integer

  pauta :: [(Numero, Nome, Data)] -> Table Numero (Nome, Data)
  pauta [ ] = newTable
  pauta ((x,y,z):resto) = updateTable (x,(y,z)) (pauta resto)

  teste = [(1111,"Dunga",14), (5555,"Constantino", 15),
            (3333,"Afonso",18), (2222,"Cecilia",19), (7777,"Vieira",14),
            (6666,"Margarida",26)]

```

Neste caso, podemos usar a função **pauta** para transformar a lista **teste** em uma **tabela** em que a chave seja o **Número** de uma pessoa e o valor correspondente a este número seja a tupla (**Nome, Data**). Nesta situação, podemos usar as funções definidas em **Table** para manipular a lista de triplas.

7.6 Arrays

Até este ponto, ainda não nos referimos à implementação de *arrays* em Haskell. Estes tipos de dados são muito úteis na resolução de problemas e merecem ser analisados como eles podem ser utilizados nesta linguagem.

Os *arrays*, baseado em [39, 41], são usados para armazenar e recuperar um conjunto de elementos, onde cada um deles tem um único índice, indicando a sua posição dentro do arranjo. Nesta seção, será descrito como estas estruturas podem ser utilizadas em Haskell. Na realidade, o padrão ainda em vigor para Haskell não contemplou estas estruturas e, por este motivo, elas não fazem parte do **Prelude** padrão. No entanto, foi criado um módulo com a implementação destas estruturas como um tipo abstrato de dados e pode ser importado.

7.6.1 A criação de arrays

Os *arrays*, em Haskell, são criados através de três funções predefinidas: **array**, **listArray** e **accumArray**, que serão descritas a seguir. A primeira delas, **array**, tem a seguinte sintaxe:

```
array <limites> <lista_de_associacoes>
```

onde o primeiro argumento, **<limites>**, descreve os limites inferior e superior dos índices do *array*, entre parênteses. Por exemplo, **(0,10)**, para *arrays* unidimensionais, ou **((1,1),(5,5))** para *arrays* bidimensionais, enfatizando que os valores dos índices também podem ser expressões.

O segundo argumento, **<lista_de_associacoes>**, é uma lista da forma (i,v) , sendo i o índice do *array* e v o valor da associação. Normalmente, a lista das associações são criadas através de expressões ZF. Como exemplo de alguns *arrays*, podemos citar:

```
x = array (1,5) [(3,'n'),(1,'D'),(2,'u'),(5,'a'),(4,'g')]
fun n = array (0,n) [(i,3*i)|i<-[0..n]]
y = array ((1,1),(2,2)) [(i,j),i*j|i<-[1..2],j<-[1..2]]
```

O tipo de um *array* é denotado por **Array a b**, onde **a** representa o tipo do índice e **b** representa o tipo do valor. Desta forma, os *arrays* anteriores têm os seguintes tipos:

```
x :: Array Int Char
fun :: Int -> Array Int Int
y :: Array (Int, Int) Int
```

Um *array* se torna indefinido se qualquer índice for especificado fora de seus limites. Também, se duas associações tiverem o mesmo índice fará com que o valor do índice seja indefinido. Isto significa que um *array* é *estrito* em seus limites e *lazy* em seus valores. Como exemplo, a função **fib** pode ser definida usando a função **array**, da seguinte forma:

```
fib n = a
  where a = array (1,n) [(1,1),(2,1)] ++ [(i,a!(i-1)+a!(i-2))|i<-[3..n]]
```

onde o operador **!** é uma função binária e infixa, de forma que **a!i** retorna o valor do elemento de índice **i** do *array* **a**.

A segunda função predefinida sobre *arrays* é **listArray**, cuja sintaxe é a seguinte:

```
listArray <limites> <lista_de_valores>
```

onde o argumento **<limites>** tem o mesmo significado já visto para a função **array** e o argumento **<lista_de_valores>** é a lista dos valores do *array*. Para exemplificar, o *array* **x**, definido anteriormente, também pode ser construído usando a função **listArray**, da seguinte forma:

```
x = listArray (1,5) "Dunga"
```

A terceira função prédefinida, **accumArray**, tem a seguinte sintaxe:

```
accumArray <funcao> <init> <limites> <lista_de_associacoes>
```

Esta função remove a restrição de que um dado índice só possa aparecer, no máximo, uma vez na lista de associações, combinando os índices conflitantes através da função argumento **<funcao>** e inicializando os valores do *array* com **<init>**. Os outros argumentos são os mesmos já conhecidos para as duas funções anteriores. Por exemplo, o *script* a seguir mostra uma chamada e o resultado utilizando esta função:

```
accumArray (-) 0 (1,5) [ ]
array (1,5) [(1,0),(2,0),(3,0),(4,0),(5,0)]
```

Um exemplo mais completo, pode ser

```
accumArray (+) 2 (1,5) [(1,2),(2,3),(1,3),(4,1)]
array (1,5) [(1,7),(2,5),(3,2),(4,3),(5,2)]
```

em que todos os elementos do *array* são inicializados com o valor 2. Como no *array* existem dois valores para o índice 1, a função `(+)` resolve esta indefinição somando os valores 2 e 3 ao valor inicializado, totalizando 7. Os índices 3 e 5 não constam na chamada, mas são inicializados com o valor 2 que é o segundo parâmetro da função.

7.6.2 Utilizando arrays

Já foi visto que a função `!` retorna o valor do elemento mostrado cujo índice é o segundo argumento. Outras funções também já existem para manipular *arrays*. São elas:

- **bounds**: exibe os limites de um array,
- **indices**: retorna os índices do *array*,
- **elems**: retorna os elementos do *array* e
- **assocs**: que retorna as associações do *array*.

Sendo *y* o *array* definido na subseção anterior, então:

```
y!(1,2) => 2
bounds y => ((1,1),(2,2))
indices y => [(1,1),(1,2),(2,1),(2,2)]
elems y => [1,2,2,4]
assocs y => [((1,1),1),((1,2),2),((2,1),2),((2,2),4)]
```

Também é possível atualizar um *array* em um estilo funcional, ou seja, a partir de um *array* podemos retorna um novo *array* cujos elementos são os mesmos do *array* anterior, exceto para alguns índices determinados. Esta operação é feita através do operador `//` que toma como argumentos um *array* e uma lista de associações e retorna um novo *array* onde a lista de substituições passadas substitui as associações do *array* antigo.

Por exemplo, sendo *x* o *array* definido acima, a chamada `x // [(1,'F')]` terá como resultado **"Funga"**.

7.7 Resumo

Este Capítulo foi reservado ao estudo dos módulos e como eles podem ser construídos em Haskell. Isto foi necessário para que o usuário possa utilizar este recurso que torna possível a programação de grandes sistemas, de forma otimizada e gerenciável. A modularidade permite que os módulos sejam recompilados e testados de forma independente, facilitando o gerenciamento na construção de grandes programas.

Além da modularidade, no Capítulo também foram tratados os tipos abstratos de dados, analisando a necessidade de seu surgimento, mostrando porque eles são abstratos em relação aos tipos de dados vistos até aqui, que têm representações concretas e acessíveis aos usuários. Foi visto como eles podem ser construídos em Haskell permitindo ao leitor que possa segui-los e compreenda como outros tipos podem ser implementados, levando em consideração o desempenho de sua representação e de seus métodos. Já foi dito neste estudo que programar é simular problemas contruindo modelos para representá-los, de forma que estes modelos possam ser processados por um computador, emitindo soluções que sejam interpretadas como soluções para os problemas reais.

As fontes de exemplos e exercícios mostrados neste Capítulo, foram os livros de Richard Bird [6] e de Fethi Rabhi e Guy Lapalme [39].

Para quem deseja conhecer mais aplicações de programas funcionais, o livro de Paul Hudak [15] é uma excelente fonte de estudo, principalmente para quem deseja conhecer aplicações da multimídia usando Haskell.

7.8 Exercícios propostos

1. Projete, em Haskell, um tipo abstrato de dados para matrizes com elementos inteiros, incluindo operações para adição, subtração e multiplicação de matrizes e as operações para o cálculo de matriz transposta e de matriz adjunta para matrizes quadradas.
2. Projete, em Haskell, um tipo abstrato de dados para números complexos, incluindo operações para adição, subtração, multiplicação, divisão, extração de cada uma das partes de um número complexo e a construção de um número complexo a partir de duas constantes, variáveis ou expressões de ponto flutuante.
3. Projete, em Haskell, um tipo abstrato de dados para filas, cujos elementos armazenem nomes com 10 caracteres. Os elementos da fila devem ser dinamicamente alocados no monte. As operações da fila são: a inserção, a remoção e o esvaziamento.
4. Suponha que alguém tenha projetado em Haskell um tipo abstrato de dados **Pilha** no qual a função **top** retorne um caminho de acesso ou ponteiro, em vez de retornar uma cópia do elemento do topo. Essa não é uma abstração de dados verdadeira. Por quê? Dê um exemplo que ilustre o problema.
5. As listas finitas podem ser consideradas como um tipo abstrato de dados. Por exemplo, considere as operações:

```
nil  :: List t
null :: List t -> Bool
cons :: t -> List t -> List t
head :: List t -> t
tail :: List t -> list t
```

Faça uma especificação algébrica do tipo lista finita.

Capítulo 8

Programação com ações em Haskell

*“... for exemple, a computation implying the modification of
a state for keeping track of the number of evaluating steps
might be described by a monad which takes as input parameter and
returns the new state as part of its result.
Computations raising exceptions or performing input-output can
also be described by monads.”*
(Fethi Rabbi et Guy Lapalm in [39])

8.1 Introdução

Este Capítulo é dedicado à forma utilizada por Haskell para se comunicar com o mundo exterior para realizar operações de *I/O* que são inerentemente pertencentes ao mundo imperativo e, portanto, possíveis de conter efeitos colaterais. Já foi visto anteriormente que, no paradigma funcional, os programas são expressões a serem avaliadas para se encontrarem valores que são atribuídos a nomes. Em Haskell, o resultado de um programa é o valor da avaliação de uma função que é atribuído ao identificador **main** no Módulo **Main** do arquivo **Main.hs**.

Mas a realidade é que a grande maioria dos programas exige alguma interação com o mundo externo. Por exemplo:

- um programa pode necessitar ler ou escrever alguma entrada em algum terminal;
- um sistema de *e-mail* lê e escreve em arquivos ou em canais e
- um programa pode querer mostrar uma figura em uma janela do monitor.

Historicamente, as operações de *I/O* representaram um desafio muito grande durante muito tempo para os usuários das linguagens funcionais. Os projetistas dessas linguagens tomaram rumos distintos na solução destes problemas. Por exemplo, os projetistas de **Standard ML** [35] preferiram incluir operações como

```
inputInt :: Int
```

cujos efeitos são a leitura de um valor inteiro a partir do dispositivo padrão de entrada. Este valor lido é atribuído ao identificador **inputInt**. Mas surge um problema porque, cada vez que **inputInt** é avaliado, um novo valor é atribuído a ele, sendo esta uma característica do

paradigma imperativo, não do modelo funcional. Por este motivo, diz-se que **SML** admite um modelo funcional *impuro*, porque admite atribuições destrutivas.

Seja a seguinte definição de uma função, em **SML**, que calcula a diferença entre dois inteiros:

```
inputDif = inputInt - inputInt
```

Suponha que o primeiro item de entrada seja o valor 10 e o segundo seja 20. Dependendo da ordem em que os argumentos de **inputDif** são avaliados, ela pode ter como resultado os valores 10 ou -10. Isto corrompe o modelo, uma vez que era esperado que o valor fosse 0 (zero) para **inputDif**.

A razão deste problema é que o valor de uma expressão não é determinado simplesmente pela observação dos valores de suas partes, porque não se pode atribuir um valor a **inputInt** sem antes saber em que local do programa ele ocorre. A primeira e a segunda ocorrências de **inputInt** em **inputDif** podem acontecer em diferentes tempos e podem ter valores diferentes. Este tema foi analisado exaustivamente no Capítulo 1 deste trabalho.

Um segundo problema com estas operações de *I/O* é que os programas se tornam extremamente difíceis de serem seguidos, porque qualquer definição em um programa pode ser afetada pela presença de operações de entrada e saída.

Por causa disto, durante muito tempo, as operações de *I/O* se tornaram um desafio para as linguagens funcionais e várias tentativas foram feitas na busca de soluções que não corrompessem o paradigma funcional.

8.1.1 Ser ou não ser pura: eis a questão

O Comitê de definição da linguagem Haskell resolveu manter a linguagem pura, sem *side effects*, e, por este motivo, a decisão sobre o sistema de *I/O* foi um dilema a ser resolvido. Eles não queriam perder o poder de expressividade da linguagem apenas porque ela tinha que ser pura e a comunicação com o mundo externo era uma necessidade pragmática. Havia o receio de que Haskell passasse a ser considerada uma linguagem de brincadeira, se este tema fosse considerado de pouca importância em sua definição.

O sistema de entrada e saída adotado em Haskell é poderoso, expressivo e fácil de ser entendido e utilizado. Haskell separa o código puramente funcional do código que pode ter efeitos colaterais.

Inicialmente, duas soluções foram propostas: *streams* e *continuations* (continuações). Estes dois temas já eram dominados teoricamente e pareciam oferecer as condições de expressividade exigidas, além de ambas serem puras. No decorrer dessas discussões, verificou-se que estas técnicas eram funcionalmente equivalentes, ou seja, era possível modelar *streams* usando *continuations* e vice-versa. Desta forma, no Haskell 1.0 Report, o sistema de *I/O* foi definido em termos de *streams*, incluindo *continuations* [16].

Em 1989, Eugenio Moggi publicou no LICS um artigo sobre o uso de *Mônadas*¹, originadas da *Teoria das Categorias* [46] para descrever características das linguagens de programação. Philip Wadler verificou que a técnica que Moggi havia utilizado para estruturar semântica também poderia ser empregada com sucesso para estruturar programas funcionais [57, 58]. Uma mônada consiste em um construtor de tipos **M** e um par de funções: **return** e **>>=**, algumas vezes, esta última chamada de **bind**. Seus tipos são:

¹Até o momento não foi possível descobrir se Mônada é uma palavra masculina ou feminina e por este motivo será tratada como feminina neste texto.

```
return :: a -> M a
(>>=) :: M a -> (a -> M b) -> M b
```

M a deve ser lido como o tipo de uma computação que retorna um valor do tipo **a**, com possíveis *side effects*. Digamos que **m** seja uma expressão do tipo **M a** e **n** seja uma expressão do tipo **M b** com uma variável livre, **x**, do tipo **a**. Assim, a expressão **m >>= (\x -> n)** tem o tipo **M b**. Isto significa que a computação indicada por **m** é feita ligando-se o valor retornado a **x** e em seguida realiza-se a computação indicada por **n**. Isto é análogo à expressão

```
let x = m in n
```

em uma linguagem com *side effects*, como **ML**, excluindo o fato de que os tipos não indicam a presença destes efeitos. Na versão de **ML**, **m** tem o tipo **a** em vez de **M a** e **n** tem o tipo **b** em vez de **M b**.

Existem três leis que as definições de **return** e **>>=** devem obedecer para que a estrutura seja considerada uma *mônada*, no sentido definido pela Teoria das Categorias. Estas leis garantem que a composição de funções com *side effects* seja associativa e tenha uma identidade [58].

Uma mônada é um tipo de “*padrão de programação*”. Este padrão pode ser expresso em Haskell usando **type class** da seguinte forma:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

O estilo monádico rapidamente dominou os modelos anteriores. Os tipos são mais compactos e mais informativos. Uma prova baseada em parametricidade garante que nenhuma referência é desperdiçada na mudança de uma computação encapsulada para outra [25]. O resultado prático disto foi que, pela primeira vez, esta prova ofereceu a possibilidade de se implementar uma função usando um algoritmo imperativo com a garantia da ausência de qualquer *side effect*.

Em 1996, Peyton Jones e outros pesquisadores estenderam as mônadas de *IO* com *threads*, onde cada *thread* pode realizar *I/O*, tornando a semântica da linguagem não determinística [38]. Os *threads* podem comunicar entre si usando locações mutáveis e sincronizadas chamadas **MVars**. O leitor interessado em conhecer melhor este tema deve recorrer à bibliografia específica. Neste particular, as referências [57, 58] e [25] representam um bom começo.

8.2 Entrada e Saída em Haskell

Como já descrito anteriormente, um programa funcional consiste em uma expressão que é avaliada para encontrar um valor que deve ser retornado e ligado a um identificador. Quando as operações não se tratam de expressões, por exemplo, no caso de uma operação de *I/O*, que valor deve ser retornado? Este retorno é necessário para que o paradigma seja respeitado. Caso contrário, ele é considerado corrompido.

A solução adotada pelos idealizadores de Haskell foi introduzir um tipo especial de dados chamado “ação”. Quando o sistema detecta uma operação deste tipo, ele sabe que uma ação deve ser executada. Existem ações primitivas, por exemplo, escrever um caractere em um arquivo ou receber um caractere do teclado, mas também ações compostas como imprimir várias *strings*.

As operações ou funções, em Haskell, cujos resultados de suas avaliações sejam ações, são chamadas de “*comandos*”, porque elas comandam o sistema para realizar alguma ação. Os

comandos realizam ações e retornam um valor de um determinado tipo, que pode ser usado futuramente pelo programa.

Haskell provê o tipo **IO a** para permitir que um programa faça alguma operação de *I/O* e retorne um valor do tipo **a**. Haskell também provê um tipo **IO ()** que contém um único elemento, representado por **()**. Uma função do tipo **IO ()** representa uma operação de *I/O* (ação) que retorna o valor **()**. Semanticamente, este é o mesmo resultado de uma operação de *I/O* que não retorna qualquer valor. Por exemplo, a operação de escrever a *string* “Olha eu aqui!” pode ser entendida desta forma, ou seja, um objeto do tipo **IO ()**.

Existem muitas funções predefinidas em Haskell para realizar ações, além de um mecanismo para sequencializá-las, permitindo que alguma ação do modelo imperativo seja realizada sem corromper o paradigma funcional.

8.2.1 Operações de entrada

A operação de leitura de um caractere (**Char**), a partir do dispositivo padrão de entrada, é descrita em Haskell pela função predefinida **getChar** do tipo:

```
getChar :: IO Char
```

Para ler uma *string*, a partir do dispositivo padrão de entrada, usamos a função predefinida **getLine** do tipo:

```
getLine :: IO String
```

As aplicações destas funções devem ser interpretadas como operações de leitura seguidas de retornos: no primeiro caso, de um caractere e, no segundo, de uma *string*.

8.2.2 Operações de saída

A operação de impressão de um texto é feita por uma função que toma a *string* a ser escrita como entrada, escreve esta *string* no dispositivo padrão de saída e retorna um valor do tipo **()**. Esta função foi citada no Capítulo 3, mas de forma genérica e sem nenhuma profundidade, uma vez que, seria difícil o leitor entender sua definição com os conhecimentos adquiridos sobre Haskell até aquele ponto. Esta função é **putStr**, predefinida em Haskell, com o seguinte tipo:

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

Agora podemos escrever “Olha eu aqui!”, da seguinte forma:

```
aloGalvao :: IO ()
aloGalvao = putStr "Olha eu aqui!"
```

Usando **putStr** podemos definir uma função que escreva uma linha de saída:

```
putStrLn :: String -> IO ()
putStrLn = putStr . (++ "\n")
```

cujos efeitos são adicionar o caractere de mudança de linha ao fim da entrada passada para **putStr**. Para escrever valores em geral, Haskell provê a classe **Show** com a função

```
show :: (Show a) => a -> String
```

que é usada para transformar valores, de vários tipos, em *strings* para que possam ser mostradas através da função **putStr**. Por exemplo, pode-se definir uma função de impressão geral

```
print :: (Show a) => a -> IO ()
print = putStrLn . show
```

Se o objetivo for definir uma ação de *I/O* que não realize qualquer operação de *I/O*, mas que retorne um valor, pode-se utilizar a função

```
return :: a -> IO a
```

cujo efeito é não realizar qualquer ação de *I/O* e retornar um valor do tipo *a*.

8.2.3 O comando **do**

O comando **do** é um mecanismo flexível, construído para realizar duas operações sobre ações em Haskell:

1. a sequencialização de ações de *I/O* e
2. a captura de valores retornados por ações de *I/O*, para repassá-los futuramente para outras ações do programa.

Por exemplo, a função **putStrLn str**, descrita anteriormente, é predefinida em Haskell e faz parte do **Prelude** padrão. Ela realiza duas ações: a primeira é escrever a *string* **str** no dispositivo padrão de saída e a segunda é fazer com que o *prompt* se mude para a próxima linha. Esta operação pode ser definida utilizando-se a notação **do**, da seguinte forma:

```
putStrLn :: String -> IO ()
putStrLn str = do putStr str
                  putStr "\n"
```

Neste caso, o efeito do comando **do** é a sequencialização das ações de *I/O*, em uma única ação. A sintaxe do comando **do** é regida pela regra do *offside* e pode-se tomar qualquer número de argumentos (ações). No entanto, pode-se usar o “ponto e vírgula” (;) entre os comandos a serem sequencializados pelo comando **do** que devem ser envolvidos por chaves como uma sequência de comandos em programas codificados na linguagem de programação C.

Como outro exemplo, pode-se querer escrever alguma coisa *n* vezes. Por exemplo, pode-se querer executar 4 vezes a mesma operação do exemplo anterior. Uma primeira versão pode ser o seguinte código em Haskell:

```
faz4vezes :: String -> IO ()
faz4vezes str = do putStrLn str
                  putStrLn str
                  putStrLn str
                  putStrLn str
```

Apesar de funcionar corretamente, esta declaração mais parece com o método da “*força bruta*”. Uma forma mais elegante de descrevê-la é transformar a quantidade de vezes que se deseja que a ação seja executada em um parâmetro. A definição a seguir é muito mais elegante:

```
fazNvezes :: Int -> String -> IO ()
fazNvezes n str = if n <= 1 then putStrLn str
                  else do putStrLn str
                        fazNvezes (n-1) str
```

Deve ser observada a recursão na cauda utilizada na definição da função **fazNvezes**, simulando a instrução de controle *while*, tão comum nas linguagens imperativas. Agora a função **faz4vezes** pode ser redefinida por

```
faz4vezes = fazNvezes 4
```

Apesar de terem sido mostrados apenas exemplos de saída, as entradas também podem ser parte de um conjunto de ações sequencializadas. Por exemplo, pode-se querer ler duas linhas do dispositivo de entrada padrão e escrever a frase “duas linhas lidas”, ao final. Isto pode ser feito da seguinte forma, em Haskell:

```
leia2linhas :: IO ()
leia2linhas = do getLine
                getLine
                putStrLn "duas linhas lidas"
```

Capturando valores lidos

No último exemplo mostrado, foram lidas duas linhas de texto mas nada foi feito com o resultado das ações de **getLine**. Pode ser necessário utilizar estes resultados no restante do programa. Isto é feito através da nomeação dos resultados das ações. Por exemplo,

```
getNput :: IO ()
getNput = do linha <- getLine
            putStrLn linha
```

onde **linha** <- nomeia o resultado de **getLine**. Apesar do identificador **linha** parecer com uma variável em uma linguagem imperativa, seu significado em Haskell é bem diferente.

Exemplo. Seja a turma de uma disciplina escolar, onde os alunos têm 3 notas: n_1 , n_2 e n_3 . Vamos construir uma função, em Haskell, que leia sequencialmente o nome de cada aluno e suas 3 notas e mostre o nome do aluno e sua média aritmética. Este processo deve continuar até que o nome do aluno seja “**Final**”.

Para isto, vamos construir a ação **leiaAte** da seguinte forma:

```
leiaAte :: IO ( )
leiaAte = do nome <- getLine
            if nome == "Final"
            then return ()
            else do n1 <- getDouble
                    n2 <- getDouble
                    n3 <- getDouble
                    putStr (nome ++ show ((n1+n2+n3)/3.0) ++ "\n")
                    leiaAte
```

A ação **getDouble** recebe um dado como entrada e interpreta o valor deste dado como um tipo **Double**. Ela é definida da seguinte forma:

```

getDouble :: IO Double
getDouble = do dado <- getLine
              return (read dado :: Double)

```

A ação **return** () representa uma ação que é equivalente a “fazer nada” e “retornar nada”.

8.3 Arquivos, canais e descritores

Deve ser do conhecimento do leitor que os arquivos são variáveis permanentes, cujos valores podem ser lidos e/ou atualizados em momentos futuros, depois que o programa que os criou tenha terminada a sua execução. Neste caso, diz-se que estas variáveis são não voláteis, em contraposição aos outros tipos de variáveis conhecidas como voláteis. É desnecessário comentar a importância que os arquivos têm na computação, sendo clara a necessidade de uma forma de comunicação entre o programa e os arquivos que ele manipula. Já foi vista uma forma, através do comando **do**. A outra é através de “*descritores*”.

Para obter um descritor (do tipo **Handle**) de um arquivo é necessária a operação de abertura deste arquivo para que futuras operações de leitura e/ou escrita possam acontecer. Além disso, é necessária uma operação de fechamento deste arquivo, após suas operações terem sido realizadas, para que os dados não sejam perdidos quando o programa terminar sua execução.

A Tabela 8.1 mostra as formas como os arquivos podem ser abertos, operados e fechados em Haskell, usando o comando **openFile**.

Tabela 8.1: Forma de operações com arquivos em Haskell.

IOMode	lê	escreve	Posição inicial	Notas
ReadMode	Sim	Não	Início do arquivo	O arquivo deve existir
WriteMode	Não	Sim	Início do arquivo	Se existir o arquivo perde os dados
ReadWriteMode	Sim	Sim	Início do arquivo	O arquivo é criado se não existir senão os dados são conservados
AppendMode	Não	Sim	Fim do arquivo	O arquivo é criado se não existir e os dados existentes são conservados

Os programas em Haskell podem manipular arquivos tipo texto ou arquivos binários. Neste último caso, é necessário utilizar **openBinaryFile** em vez de **openFile**. O sistema operacional *Windows* processa arquivos textos de forma diferente de como processa arquivos binários. O sistema operacional *Linux* utiliza tanto **openFile** quanto **openBinaryFile**, mas é aconselhável usar **openBinaryFile** para processar arquivos binários para efeito de portabilidade.

Estas operações são descritas em Haskell, da seguinte forma:

```

data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()

```

Por convenção, todos os comandos usados para tratar descritores de arquivos são iniciadas com a letra **h**. Por exemplo, as funções

```

hPutChar :: Handle -> Char -> IO ()

```



```
hPutStr :: Handle -> String -> IO ()
hPutStrLn :: Handle -> String -> IO ()
hPrint :: (Show a) => Handle -> a -> IO ()
```

são utilizadas para escrever alguma coisa em um arquivo. Já os comandos

```
hGetChar :: Handle -> IO ()
hGetLine :: Handle -> IO ()
```

são utilizados nas operações de leituras em arquivos. As funções **hPutStrLn** e **hPrint** incluem um caractere `'\n'` para a mudança de linha ao final da *string*.

Haskell também permite que todo o conteúdo de um arquivo seja retornado como uma única *string*, através da função

```
hGetContents :: Handle -> String
```

No entanto, há que se fazer uma observação. Apesar de parecer que **hGetContents** retorna todo o conteúdo de um arquivo de uma única vez, não é realmente isto o que acontece. Na realidade, a função retorna uma lista de caracteres, como esperado, mas de forma *lazy*, onde os elementos são lidos sob demanda.

8.3.1 A necessidade dos descritores

É importante lembrar que um arquivo também pode ser escrito sem o uso de descritores. Por exemplo, pode-se escrever em um arquivo usando o comando

```
type FilePath = String
writeFile :: FilePath -> String -> IO ()
```

Também podemos acrescentar uma *string* ao final de um arquivo com o comando

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

Então, qual a necessidade de se utilizar descritores? A resposta é imediata: eficiência. Vamos analisar.

Toda vez que o comando **writeFile** ou **appendFile** é executado, deve acontecer uma sequência de ações, ou seja, o arquivo deve ser inicialmente aberto, a *string* deve ser escrita e, finalmente, o arquivo deve ser fechado. Se muitas operações de escrita forem necessárias, então serão necessárias muitas destas sequências. Ao se utilizar descritores, necessita-se apenas de uma operação de abertura no início e outra de fechamento no final.

Para ler e/ou escrever a partir de um descritor, que normalmente corresponde a um arquivo em disco, o sistema operacional mantém um registro interno da posição atual de leitura do arquivo, ou seja, um ponteiro para a posição atual. A cada vez que uma leitura é feita, o sistema operacional retorna o valor do dado apontado por este ponteiro e incrementa o ponteiro para apontar para o próximo dado a ser lido.

O comando **hTell** pode ser utilizado para ver a posição corrente deste ponteiro em relação ao início do arquivo, que tem a posição 0 (zero). Isto significa que, quando um arquivo é criado, ele é criado vazio e o ponteiro tem o valor 0 (zero). Quando se escrevem 10 bytes no arquivo, a posição será 10. **hTell** tem o tipo `hTell :: Handle -> IO Integer`.

O complemento de **hTell** é **hSeek** que permite trocar o valor da posição atual. Ele toma três parâmetros: **Handle**, **SeekMode** e um endereço. O parâmetro **SeekMode** pode ter três valores distintos para especificar como uma determinada posição deve ser interpretada. O primeiro valor é **AbsoluteSeek** indicando que a posição deve ser a atual que também é a indicada pelo valor de **hTell**. O segundo valor é **RelativeSeek** que deve ser interpretado como um valor a partir da posição atual, onde um valor positivo indica uma posição para a frente da posição atual e um valor negativo indica uma posição para atrás da posição atual no arquivo. O terceiro valor é **SeekFromEnd** que especifica a quantidade de *bytes* antes do final do arquivo.

Vamos mostrar um exemplo baseado em O'Sullivan [34] que é apresentado em várias versões para se verificar as possibilidades que Haskell oferece, através das ações.

```
import System.IO
import Data.Char(toUpper)

main :: IO ()
main = do entrada <- openFile "entra.txt" ReadMode
        saida <- openFile "sai.txt" WriteMode
        loop_principal entrada saida
        hClose entrada
        hClose saida

loop_principal :: Handle -> Handle -> IO ()
loop_principal entrada saida =
    do fim_de_arquivo <- hIsEOF entrada
    if fim_de_arquivo then return ()
    else do inpStr <- hGetLine entrada
           hPutStrLn saida (map toUpper inpStr)
           loop_principal entrada saida
```

A execução deste programa, como a de qualquer outro em Haskell, se inicia com a função **main**. Neste caso, o arquivo **entra.txt** é aberto no modo de leitura e o arquivo **sai.txt** é aberto no modo de escrita. Em seguida o programa chama a função **loop_principal** que faz a leitura de uma linha do arquivo **entra.txt**. Essa linha lida tem todos os seus caracteres trocados para caracteres maiúsculos e é escrita no arquivo **sai.txt**. A função **loop_principal** é chamada recursivamente para um novo ciclo de leitura e processamento de mais uma linha, equivalendo a um *loop* em uma linguagem imperativa. Este ciclo é repetido até que se atinja o final do arquivo **entra.txt**. Finalmente, os arquivos **entra.txt** e **sai.txt** são fechados.

Deve ser observado o papel da função **return ()** em programas em Haskell que é diferente dos propósitos desta mesma função nas linguagens imperativas. Por exemplo, em C, esta função tem o propósito de encerrar a execução da função corrente. Em Haskell, a função **return ()** tem significado distinto ao da função **->**, ou seja, toma um valor puro e o coloca dentro de uma ação de *IO*. Por exemplo, sendo 10 um valor inteiro do tipo **Integer**, então **return 10** cria uma ação que armazena um valor do tipo **IO Integer**. Quando esta ação for executada, ela produz o valor 10.

Este mesmo programa pode ser feito de outra forma, usando **hGetContents** que é uma forma de leitura usando o mecanismo de leitura *lazy* empregado em Haskell.

```
import System.IO
import Data.Char(toUpper)
```

```

main :: IO ()
main = do entrada <- openFile "entra.txt" ReadMode
        saida <- openFile "sai.txt" WriteMode
        inpStr <- hGetContents entrada
        let resultado = processaDado inpStr
        hPutStr saida resultado
        hClose entrada
        hClose saida

processaDado :: String -> String
processaDado = map toUpper

```

Este programa ainda pode ser modificado, transformando-o em outro ainda mais compacto.

```

import System.IO
import Data.Char(toUpper)

main :: IO ()
main = do entrada <- openFile "entra.txt" ReadMode
        saida <- openFile "sai.txt" WriteMode
        inpStr <- hGetContents entrada
        hPutStr saida (map toUpper inpStr)
        hClose entrada
        hClose saida

```

A partir destes exemplos, pode-se observar que o comando **hGetContents** é utilizado como um filtro, onde os dados são lidos em um arquivo, processados e escritos em um outro arquivo. Este tipo de processamento é bastante comum e, por este motivo, foram criadas duas funções para atender a esta demanda: **readFile** e **writeFile**. Estas duas funções gerenciam todos os detalhes de abertura, leitura, processamento e fechamento de arquivos como *strings*. A função **readFile** usa **hGetContents** internamente. Desta forma, o exemplo anterior pode ser construído ainda mais sinteticamente.

```

import Data.Char(toUpper)

main :: IO ()
main = do inpStr <- readFile "entra.txt"
        writeFile "saida.txt" (map toUpper inpStr)

```

8.3.2 Canais

Os descritores também podem ser associados a canais, que são portas de comunicação não associadas diretamente a um arquivo. Os canais mais comuns são: a entrada padrão (**stdin**), a área de saída padrão (**stdout**) e a área de erro padrão (**stderr**). As operações de *I/O* para caracteres e *strings* em canais incluem as mesmas listadas anteriormente para a manipulação de arquivos. Na realidade, as funções **getChar** e **putChar** são definidas como:

```

getChar = hGetChar stdin
putChar = hPutChar stdout

```

Até mesmo **hGetContents** pode ser usada com canais. Neste caso, o fim de um canal é sinalizado com um caractere de fim de canal que, na maioria dos sistemas, é **Ctrl-d**.

Normalmente, um descritor é um arquivo, mas pode ser também uma conexão de rede, um *drive* ou um terminal. Para verificar se um dado descritor pode, ou não, ser percorrido, pode-se usar o comando **hIsSeekable** que retorna um valor booleano.

8.4 Gerenciamento de exceções

Vamos agora nos reportar a erros que podem acontecer durante as operações de *I/O*. Por exemplo, pode-se tentar abrir um arquivo inexistente, ou pode-se tentar ler um caractere de um arquivo que já atingiu seu final. Certamente, não se deve querer que o programa páre por estes motivos. O que normalmente se espera, é que o erro seja reportado como uma “*condição anômala*”, mas que possa ser corrigida, sem a necessidade de que o programa seja abortado. Para fazer este “*gerenciamento de exceções*”, em Haskell, são necessários apenas alguns comandos de *I/O*.

As exceções têm o tipo **IOError**. Entre as operações permitidas sobre este tipo, está uma coleção de predicados que podem ser usados para testar tipos particulares de exceções. Por exemplo,

```
isEOFError :: IOError -> Bool
```

detecta o fim de um arquivo. Mesmo assim, existe uma função **catch** que faz o gerenciamento de exceções. Seu primeiro argumento é a ação de *I/O* que se está tentando executar e seu segundo argumento é um “*descritor de exceções*” do tipo **IOError -> IO a**. Vejamos:

```
catch :: IO a -> (IOError -> IO a) -> IO a
```

No comando **catch com ger**, a ação que ocorre em **com** (pode até gerar uma sequência longa de ações, podendo até mesmo ser infinita) será executada pelo gerenciador **ger**. O controle é efetivamente transferido para o gerenciador através de sua aplicação à exceção **IOError**. Por exemplo, a versão de **getChar**, a seguir, retorna um caractere de uma nova linha, se algum tipo de execução for encontrada.

```
getChar' :: IO Char
getChar' = catch getChar (\e -> return '\n')
```

No entanto, esta versão trata todas as exceções da mesma maneira. Se apenas a exceção de fim de arquivo deve ser reconhecida, o valor de **IOError** deve ser solicitado.

```
getChar' :: IO Char
getChar' = catch getChar (\e -> if isEOFError e then return '\n'
                                else ioError e)
```

A função **ioError**, usada neste exemplo, “*empurra*” a exceção para o próximo gerenciador de exceções. Em outras palavras, permitem-se chamadas aninhadas a **catch** e estas, por sua vez, produzem gerenciadores de exceções, também aninhados. A função **ioError** pode ser chamada de dentro de uma sequência de ações normais ou a partir de um gerenciador de exceções, como em **getChar**, deste exemplo.

Usando-se **getChar'**, pode-se redefinir **getLine** para demonstrar o uso de gerenciadores aninhados.

```
getLine' :: IO String
getLine' = catch getLine'' (\err -> "Error: " ++ show err)
```

```

where getLine'' = do c <- getChar'
                    if c == '\n' then return ""
                    else do l <- getLine'
                        return (c:l)

```

Exemplo final. O exemplo a seguir foi retirado do livro de Cláudio César Sá [41].

```

module Cadastro where
import IO
import System
import Numeric
import Char (toUpper)
--import Hugs,Prelude

--Função principal --
{--Antes do menu de opções ser exibido, deve ser realizada uma checagem
a fim de verificar se o arquivo é vazio, ou não. Se o arquivo estiver vazio, é
necessário que a inclusão de uma lista vazia ([]) seja realizada para que as
demais operações possam ser realizadas com sucesso.
--}
--
-- Menu de opções --

menu :: IO ( )
menu = do putStrLn " "
          putStrLn "-----"
          putStrLn "| |"
          putStrLn "| CADASTRO DE PESSOAS |"
          putStrLn "| |"
          putStrLn "| a - Insere cadastro |"
          putStrLn "| b - Imprime cadastro |"
          putStrLn "| c - Busca por nomes |"
          putStrLn "| d - Soma das idades |"
          putStrLn "| e - Média das alturas |"
          putStrLn "| f - Busca por sexo |"
          putStrLn "| g - Excluir um cadastro |"
          putStrLn "| h - Excluir todos os cadastros |"
          putStrLn "| i - Sair do sistema |"
          putStrLn "|-----|"
          putStr "Digite uma das opções: "

le_opcao
le_opcao :: IO ( )
le_opcao = do opcao <- getChar
              putStr "\n"

f_menu (toUpper opcao)
f_menu :: Char -> IO ( )
f_menu i = do case i of
                'A' -> insere_cadastro
                'B' -> imprime_cadastros

```

```

        'C' -> busca_p_nomes
        'D' -> soma_d_idades
        'E' -> media_d_alturas
        'F' -> busca_p_sexo
        'G' -> excluir_um_cadastro
        'H' -> excluir_todos_cadastros
        otherwise -> sair i
    putStrLn "Operação concluída"
    if not(i=='I') then menu else putStr " "

insere_cadastro :: IO ( )
insere_cadastro =
    do putStrLn "Nome: "
       nm <- getLine
       putStrLn "Idade: "
       id <- getLine
       putStrLn "Altura: "
       alt <- getLine
       putStrLn "M - Masculino | F - Feminino"
       putStrLn "Sexo: "
       sex <- getChar
       let cadastro = nm++"#"+id++"#"+alt++"#'+[(toUpper sex)]++'"'
       pt_arq <- abreArquivo "dados.txt" AppendMode
       hPutStrLn pt_arq cadastro
       fechaArquivo pt_arq

imprime_cadastros :: IO ( )
imprime_cadastros =
    do putStrLn " "
       putStrLn "-----"
       pt_arq <- abreArquivo "dados.txt" ReadMode
       conteudo <- (hGetContents pt_arq)
       cadastros <- (converteConteudo (conteudo))
       imprime cadastros
       fechaArquivo pt_arq
       putStrLn "-----"

busca_p_nomes :: IO ( )
busca_p_nomes =
    do putStrLn " "
       putStrLn "digite o nome desejado: "
       nome <- getLine
       busca_p_algo busca_por_nome nome

soma_d_idades :: IO ( )
soma_d_idades =
    do pt_arq <- abreArquivo "dados.txt" ReadMode
       conteudo <- (hGetContents pt_arq)
       cadastros <- (converteConteudo (conteudo))
       putStrLn (show (soma_d_idade cadastros))
       fechaArquivo pt_arq

```

```

media_d_alturas :: IO ( )
media_d_alturas =
    do pt_arq <- abreArquivo "dados.txt" ReadMode
       conteudo <- (hGetContents pt_arq)
       cadastros <- (converteConteudo (conteudo))
       putStrLn (show (media_d_altura cadastros))
       fechaArquivo pt_arq

busca_p_sexo :: IO ( )
busca_p_sexo =
    do putStrLn " "
       putStrLn "digite o sexo desejado: "
       sexo <- getChar
       busca_p_algo busca_por_sexo sexo

excluir_um_cadastro :: IO ( )
excluir_um_cadastro =
    do putStrLn "O cadastro será apagado pelo nome."
       utStrLn "Digite o nome desejado: "
       nome <- getLine
       pt_arq <- abreArquivo "dados.txt" ReadMode
       conteudo <- (hGetContents pt_arq)
       cadastros <- (converteConteudo (conteudo))
       let novo_conteudo = apaga_p_nome cadastros nome
       aux_pt_arq <- abreArquivo "auxiliar.txt" WriteMode
       hPutStr aux_pt_arq novo_conteudo
       fechaArquivo aux_pt_arq
       fechaArquivo pt_arq
       copiar "auxiliar.txt" "dados.txt"

excluir_todos_cadastros :: IO ( )
excluir_todos_cadastros =
    do putStrLn "Realmente deseja apagar todos os dados do sistema?(s/n)"
       resp <- getChar
       if not ((toUpper resp) == 'S') then putStrLn " "
       else do pt_arq <- abreArquivo "dados.txt" WriteMode
              fechaArquivo pt_arq
              putStrLn "Apagando dados . . ."

sair :: Char -> IO ( )
sair i
    | i == 'I' = putStrLn "Saindo do sistema . . ."
    | otherwise = putStrLn "Operacao Invalida . . ."

```

Funções auxiliares de consulta:

```

converteConteudo :: String -> IO [[String]]
converteConteudo cont = return (map (explodir '#') (explodir '\n' cont))

```

-- Funções com números

```

media_d_altura :: [[String]] -> Float
media_d_altura [ ] = 0.0
media_d_altura x = (soma_d_alturas x) / (fromIntegral (length x))

soma_d_alturas :: [[String]] -> Float
soma_d_alturas [ ] = 0
soma_d_alturas (x:xs) = (read (altura x) :: Float) + (soma_d_alturas xs)

soma_d_idade :: [[String]] -> Integer
soma_d_idade [ ] = 0
soma_d_idade (x:xs) = (read (idade x) :: Integer) + (soma_d_idade xs)

-- Funções auxiliares

explodir :: Eq a => a -> [a] -> [[a]]
explodir a [ ] = [ ]
explodir a (x : xs)
    | (takeWhile (/=a) (x : xs)) == [ ] = explodir a xs
    | x == a = (takeWhile (/= a) xs) : explodir a (dropWhile (/= a) xs)
    | otherwise = (takeWhile (/= a) (x : xs))
                  : explodir a (dropWhile (/= a) (x : xs))

nome, idade, altura, sexo :: [String] -> String
nome (a:b:c:d:[ ]) = a

idade (a:b:c:d:[ ]) = b

altura (a:b:c:d:[ ]) = c

sexo (a:b:c:d:[ ]) = d

copiar origem destino =
    do pt_arq <- abreArquivo origem ReadMode
       conteudo <- (hGetContents pt_arq)
       aux_pt_arq <- abreArquivo destino WriteMode
       hPutStr aux_pt_arq conteudo
       fechaArquivo aux_pt_arq
       fechaArquivo pt_arq

```

Funções auxiliares de busca

```

busca_p_algo :: ([[String]] -> a -> IO b) -> a -> IO ( )
busca_p_algo funcao filtro =
    do putStrLn " "
       putStrLn "-----"
       pt_arq <- abreArquivo "dados.txt" ReadMode
       conteudo <- (hGetContents pt_arq)
       cadastros <- (converteConteudo (conteudo))
       funcao cadastros filtro
       fechaArquivo pt_arq

```



```

        putStrLn "-----"

busca_por_nome :: [[String]] -> String -> IO ( )
busca_por_nome [ ] nm = putStrLn " "
busca_por_nome (x : xs) nm
    | (nome x) == nm = do putStrLn (foldl1 (\a b->a++" "++b) x)
busca_por_nome xs nm
    | otherwise = busca_por_nome xs nm

busca_por_sexo :: [[String]] -> Char -> IO ( )
busca_por_sexo [ ] sx = putStrLn " "
busca_por_sexo (x : xs) sx
    | (sexo x) == (":"'++ [(toUpper sx)]++"'" ) =
        do putStrLn (foldl1 (\a b->a++" "++b) x)
busca_por_sexo xs sx
    | otherwise = busca_por_sexo xs sx

--Função auxiliar de impressão

imprime :: [[Char]] -> IO ( )
imprime [ ] = putStrLn " "
imprime (x : xs) = do putStrLn (foldl1 (\a b->a++" "++b) x)
imprime xs

```

Funções de inclusão

```

apaga_p_nome :: [[String]] -> String -> String
apaga_p_nome [ ] nm = "\n"
apaga_p_nome (x : xs) nm
    | nm == (nome x) = (apaga_p_nome xs nm)
    | otherwise = (foldl1 (\a b->a++"#"++b) x) ++ "\n" ++ (apaga_p_nome xs nm)

--Funções auxiliares de arquivos

abreArquivo :: String -> IOMode -> IO Handle
abreArquivo arquivo modo =
    catch (openFile arquivo modo)
        (\_ -> do {
            putStrLn ("Impossível abrir "++ arquivo);
            putStrLn "Será aberto com um nome default: dados.txt e limpo";
            pt_arq <- abreArquivo "dados.txt" WriteMode;
            fechaArquivo pt_arq;
            abreArquivo "dados.txt" ReadMode
        })

fechaArquivo :: Handle -> IO ( )
fechaArquivo handle_arq = hClose handle_arq

```

8.5 Resumo

Este foi o Capítulo final deste trabalho, dedicado à semântica de ações, adotadas em Haskell, para tratar operações de entrada e saída, representando a comunicação que o programa deve ter com periféricos e com os arquivos. Na realidade, ela é implementada em Haskell através de *Mônadas*, uma teoria matemática bastante complexa e que, por este motivo, está fora do escopo deste estudo.

O objetivo do Capítulo foi mostrar as formas como Haskell trata as entradas e as saídas de dados, ou seja, que facilidades a linguagem Haskell oferece para a comunicação com o mundo externo ao programa. Isto inclui a leitura e escrita de dados em arquivos, bem como a criação e o fechamento de arquivos, armazenando dados para serem utilizados futuramente.

Este estudo foi baseado nos livros de Simon Thompson [55], de Richard Bird [6] e de Paul Hudak [15], como também nos excelentes textos de Philip Wadler [57, 58]. Este tema ainda é considerado novo e complexo pelos pesquisadores das linguagens funcionais e acreditamos ser o motivo pelo qual ele é ainda muito pouco tratado na literatura.

Com este Capítulo, esperamos ter cumprido nosso objetivo que foi o de proporcionar aos estudantes iniciantes das linguagens funcionais, alguma fundamentação e conceitos além de desenvolver algumas aplicações utilizando a linguagem de programação Haskell.

Referências Bibliográficas

- [1] ADELSON-VELSKII, G. M. et LANDIS, E. M. *An Algorithm for the Organization of Information*. Soviet Math, Doklady (English translation), Vol. 3; pp. 1259-1263. 1962.
- [2] AHO, Alfred V et alli. *Compiladores Princípios, técnicas e ferramentas*. 2a. Edição. Pearson Education do Brasil; São Paulo, 2008.
- [3] ANDRADE, Carlos Anreazza Rego. *AspectH: Uma Extensão Orientada a Aspectos de Haskell*. Dissertação de Mestrado. Centro de Informática. UFPE. Recife, Fevereiro 2005.
- [4] BACKFIELD, Joshua. *Becoming Functional*. O'Reilly Media, Inc. 2014.
- [5] BARENDREGT, H. P. *The Lambda Calculus: Its Syntax and Semantics*. (Revised Edn.). North Holland, 1984.
- [6] BIRD, Richard. *Introduction to Functional Programming Using Haskell*. 2nd. Edition. Prentice Hall Series in Computer Science - Series Editors C. A. Hoare and Richard Bird. 1998.
- [7] BIRD, Richard. *Pearls of Functional Algorithm Design*. 3rd. Edition. Cambridge University Press. United Kingdom, 2011.
- [8] BOYER, Carl B. *História da Matemática*. Tradução de Elza Gomide. Editora Edgard Blücher Ltda. São Paulo, 1974.
- [9] BRAINERD, W. S. et LANDWEBER, L. H. *Theory of Computation*. John Wiley & Sons, 1974.
- [10] CURRY, H. B. et FEYS, R. and CRAIG, W. *Combinatory Logic*. Volume I; North Holland, 1958.
- [11] DAVIE, Antony J. T. *An Introduction to Functional Programming Systems Using Haskell*. Cambridge Computer Science Texts. Cambridge University Press. 1999.
- [12] DAVIS, Philip J. et HERSH, Reuben. *A Experiência Matemática*. Tradução de João Bosco Pitombeira. Editora Francisco Alves. Rio de Janeiro, 1985.
- [13] HINDLAY, J. Roger. *Basic Simple Type Theory*. Cambridge Tracts in Theoretical Computer Science, 42. Cambridge University Press. 1997.
- [14] HINDLAY, J. Roger et SELDIN, Jonathan P. *Lambda-Calculus and Combinators, an Introduction*. Cambridge University Press, 2008.
- [15] HUDAK, Paul. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press, 2000.
- [16] HUDAK, Paul et all. *A History of Haskell: being lazy with class*. January 2007.

- [17] HUGHES, John. *Why Functional Programming Matters*. In Turner D. T. Ed. Research Topics in Functional Programming. Addison-Wesley, 1990.
- [18] HUGHES, John. *The Design of a Pretty-printing Library*. <http://citeseer.ist.psu.edu/~hughes95design.html>. 1995.
- [19] HUTTON, Graham. *Programming in Haskell*. Cambridge University Press. ISBN 978-0-521-69269-4. 2007.
- [20] JOHNSON, T. *Efficient Computation of Lazy Evaluation*. Proc. SIGPLAN '84. Symposium on Compiler Construction ACM. Montreal, 1984.
- [21] JOHNSON, T. *Lambda Lifting: Transforming Programs to Recursive Equations*. Aspenäs Workshop on Implementation of Functional Languages. Göteborg, 1985.
- [22] JOHNSON, T. *Target Code Generation from G-Machine Code*. Proc. Workshop on Graph Reduction. Santa Fé Lecture Notes on Computer science, Vol: 279 pp. 119-159. Springer-Verlag, 1986.
- [23] JOHNSON, T. *Compiling Lazy Functional Languages*. Ph.D. Thesis. Chalmers University of Technology, 1987.
- [24] LANDIN, P. J. *The Mechanical Evaluation of Expressions*. Computer Journal, Vol. 6, 4. 1964.
- [25] LAUNCHBURY, J et PEYTON JONES, S. L. *State in Haskell*. Lisp and Symbolic Computation, 8(4):293-342. 1995.
- [26] LINS, Rafael Dueire et LIRA, Bruno O. *FCMC: A Novel Way of Compiling Functional Languages*. Programming Languages 1:19-40; Chapman & Hall. 1993.
- [27] LINS, Rafael Dueire. *O λ -Cálculo, Computabilidade & Linguagens de Programação*. Notas de Curso. Recife-Pe, Nov. 1993.
- [28] LINS, Rafael Dueire et al. *Research Interests in Functional Programming*. I Workshop on Formal Methods. UFRGS. Outubro, 1998.
- [29] LIPOVACA, Miran. *Learn You a Haskell for Great Good! A Beginner's Guide*. No Starch Press, Inc. 2011.
- [30] MACLENNAN, Bruce J. *Functional Programming Practice*. Addison-Wesley Publishing Company, Inc. 1990.
- [31] MEIRA, Sílvia Romero de Lemos. *Introdução à Programação Funcional*. VI Escola de Computação. Campinas, 1988.
- [32] NIKHIL, R. S. and ARVIND, A. *Implicit Parallel Programming in pH*. Morgan Kaufman. 2001.
- [33] OKASAKI, Chris. *Purely Functional Data Structures*. Cambridge University Press. 2003.
- [34] O'SULLIVAN, Bryan et GOERZEN, John et STEWART, Don. *Real World Haskell*. O'Reilly. 2008.
- [35] PAULSON, Laurence C. *ML for the Working Programmer*. Cambridge University Press, 1991.

- [36] PEMMARAJU, Sriram et SKIENA, Steven. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge University Press. 2003.
- [37] PEYTON JONES, S. L. *The Implementation of Functional Programming Languages*. C. A. R. Hoare Series Editor. Prentice/Hall International. 1987.
- [38] PEYTON JONES, S. L.; GORDON, A. and FINNE, S. *Concurrent Haskell*. In 23rd ACM Symposium on Principles of Programming Languages (POPL '96), pages 295-308. St Petersburg Beach, Florida. ACM Press. 1996.
- [39] RABHI, Fethi et LAPALME, Guy. *Algorithms: A Functional Programming Approach*. 2nd. edition. Addison-Wesley. 1999.
- [40] ROJEMO, Niklaus. *Garbage Collection and Memory Efficiency in Lazy Functional Languages*. Ph.D Thesis. Department of Computing Science, Chalmers University. 1995.
- [41] SÁ, Claudio Cesar de; SILVA, Márcio Ferreira da. *Haskell: uma abordagem prática*. Novatec Editora Ltda. São Paulo, 2006.
- [42] SCHMIDT, D. A. *Denotational Semantics*. Allyn and Bacon, Inc. Massachusetts, 1986.
- [43] SCOTT, D. *Data Types as Lattices*. SIAM Journal of Computing. Vol. 5,3. 1976.
- [44] SKIENA, Steven S. et REVILLA, Miguel A. *Programming Challenges: The Programming Context Training Manual*. Texts in Computer Science. Springer Science+Business Media, Inc. 2003.
- [45] SOUZA, Francisco Vieira de. *Gerenciamento de Memória em FCMC*. Dissertação de Mestrado. CIn-UFPE. Março de 1994.
- [46] SOUZA, Francisco Vieira de. *Teoria das Categorias: A Linguagem da Computação*. Exame de Qualificação. Centro de Informática. Cin-UFPE. 1996.
- [47] SOUZA, Francisco Vieira de; LINS, Rafael Dueire. *Aspectos do Comportamento Espaço-temporal de Programas Funcionais em Uni e Multiprocessadores*. X Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho. Búzios-RJ. Setembro. 1998.
- [48] SOUZA, Francisco Vieira de; LINS, Rafael Dueire. *Analysing Space Behaviour of Functional Programs*. Conferência Latino-americana de Programação Funcional. Recife-PE. Março. 1999.
- [49] SOUZA, Francisco Vieira de. *Aspectos de Eficiência em Algoritmos para o Gerenciamento Automático Dinâmico de Memória*. Tese de Doutorado. Centro de Informática-UFPE. Recife-PE. Novembro. 2000.
- [50] SOUZA, Francisco Vieira de; LINS, Rafael Dueire. *Um Novo Algoritmo para Garbage Collection em Sistemas Fortemente Acoplados Utilizando Contagem de Referências Cíclicas*. I Congresso de Informática da Amazônia. Manaus-AM. Abril. 2001.
- [51] SOUZA, Francisco Vieira de. *Circuitos Digitais*. Teresina: UFPI/UAPI, Módulo II. 2014.
- [52] SOUZA, Francisco Vieira de. *Linguagens de Programação*. Teresina: UFPI/UAPI, Módulo IV. 2014.
- [53] SOUZA, Francisco Vieira de. *Lógica para Computação*. Teresina: UFPI/UAPI, Módulo V. 2014.

- [54] STOY, J. E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- [55] THOMPSON, Simon. *Haskell: The Craft of Functional Programming*. 3rd. Edition. Addison Wesley. 1999.
- [56] TURNER, David A. *A New Implementation Technique for Applicative Languages*. Software Practice and Experience. Vol. 9. 1979.
- [57] WADLER, Philip. *Comprehending Monads*. Mathematical Structures in Computer Science; 2:461-493, 1992.
- [58] WADLER, Philip. *The Essence of Functional Programming*. In 20th ACM Symposium on Principles of Programming Languages (POPL '92), pages 1-14. ACM. Albuquerque. 1992.
- [59] WADLER, Philip. *Why no ones Uses Functional Languages*. Functional Programming. ACM SIGPLAN. 2004.
- [60] WELCH, Peter H. *The λ -Calculus*. Course notes, The University of Kent at Canterbury, 1982.

Apêndice A

Algumas funções padrões

Neste Apêndice, estão listadas as definições de algumas funções corriqueiramente utilizadas na programação funcional em Haskell. As definições são baseadas em Richard Bird [6], mas podem apresentar algumas diferenças tanto em relação às definições apresentadas no **Prelude**, quanto em relação ao conjunto de definições apresentado pelo autor citado, uma vez que preferimos evitar o uso de letras gregas para representar tipos polimórficos.

1. `(.)`. Composição funcional:

```
(.) :: (t -> u) -> (v -> t) -> (v -> u)
(f.g) x = f (g x)
```

2. `(++)`. Concatenação de duas listas:

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

3. `(&&)`. Conjunção:

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

4. `(||)`. Disjunção:

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

5. `(!!)`. Indexação de listas:

```
(!!) :: [t] -> Int -> t
[ ] !! n = error "(!!): index too large"
(x : xs) !! 0 = x
(x : xs) (n + 1) = xs !! n
```

6. **and**. Retorna a conjunção lógica de uma lista de booleanos:


```
and :: [Bool] -> Bool
and = foldr (&&) True
```

7. **concat**. Concatenação de uma lista de listas:

```
concat :: [[t]] -> [t]
concat = foldr (++) [ ]
```

8. **const**. Cria uma função de valor constante:

```
const :: t -> u -> t
const x y = x
```

9. **cross**. Aplica um par de funções a elementos correspondentes do par:

```
cross :: (t -> u, v -> w) -> (t, v) -> (u, w)
cross (f, g) = pair (f . fst, g . snd)
```

10. **curry**. Converte uma função não currificada em uma currificada:

```
curry :: ((t, u) -> w) -> (t -> u -> w)
curry f x y = f (x, y)
```

11. **drop**. Seleciona a cauda de uma lista:

```
drop :: Int -> [t] -> [t]
drop 0 xs = xs
drop (n + 1) [ ] = [ ]
drop (n + 1) (x : xs) = drop n xs
```

12. **dropWhile**. Remove o segmento inicial de uma lista se os elementos do segmento satisfizerem uma determinada propriedade:

```
dropWhile :: (t -> Bool) -> [t] -> [t]
dropWhile p [ ] = [ ]
dropWhile p (x : xs) = if p x then dropWhile p xs else x : xs
```

13. **filter**. Seleciona os elementos de uma lista que apresentam uma determinada propriedade:

```
filter :: (t -> Bool) -> [t] -> [t]
filter p [ ] = [ ]
filter p (x : xs) = if p x then x : filter p xs else filter p xs
```

14. **flip**. Inverte o posicionamento dos argumentos de uma função:

```
flip :: (u -> t -> v) -> t -> u -> v
flip f x y = f y x
```

15. **foldl**. Aplica uma função entre os elementos de uma lista pela esquerda:

```
foldl :: (u -> t -> u) -> u -> [t] -> u
foldl f e [ ] = e
foldl f e (x : xs) = strict (foldl f) (f e x) xs
```

16. **foldl1**. Aplica uma função entre elementos de listas não vazias, pela esquerda:

```
foldl1 :: (t -> t -> t) -> [t] -> t
foldl1 f [ ] = error "foldl1 : empty list"
foldl1 f (x : xs) = foldl f x xs
```

17. **foldr**. Aplica uma função entre os elementos de uma lista pela direita:

```
foldr :: (t -> u -> u) -> u -> [t] -> u
foldr f e [ ] = e
foldr f e (x : xs) = f x (foldr f e xs)
```

18. **foldr1**. Aplica uma função entre elementos de listas não vazias, pela direita:

```
foldr1 :: (t -> t -> t) -> [t] -> t
foldr1 f [ ] = error "foldr1: empty list"
foldr1 f [x] = x
foldr1 f (x : xs) = f x (foldr1 f xs)
```

19. **fst**. Seleciona o primeiro elemento de um par:

```
fst :: (t, u) -> t
fst (x, y) = x
```

20. **head**. Retorna a cabeça de uma lista não vazia:

```
head :: [t] -> t
head [ ] = error "head: empty-list"
head (x : xs) = x
```

21. **id**. Retorna a função identidade:

```
id :: t -> t
id x = x
```

22. **init**. Retorna uma lista sem o seu último elemento:

```
init :: [t] -> [t]
init [ ] = error "init: empty list"
init [x] = [ ]
init (x : y : xs) = x : init (y : xs)
```

23. **iterate**. Produz uma lista infinita de aplicações iteradas de uma função:

```
iterate :: (t -> t) -> t -> [t]
iterate f x = x : iterate f (f x)
```

24. **last**. Retorna o último elemento de uma lista não vazia:

```
last :: [t] -> t
last [ ] = error "last: empty list"
last [x] = x
last (x : y : xs) = last (y : xs)
```

25. **length**. Retorna a quantidade de elementos de uma lista:

```
length :: [t] -> Int
length [] = 0
length (x : xs) = 1 + length xs
```

26. **map**. Aplica uma função a todos os elementos de uma lista:

```
map :: (t -> u) -> [t] -> [u]
map f [] = []
map f (x : xs) = f x : map f xs
```

27. **not**. A função de negação de um valor booleano:

```
not :: Bool -> Bool
not True = False
not False = True
```

28. **null**. Verifica se uma lista é, ou não, vazia:

```
null :: [t] -> Bool
null [] = True
null (x : xs) = False
```

29. **or**. Retorna a disjunção lógica entre os elementos de uma lista:

```
or :: [Bool] -> Bool
or = foldr (||) false
```

30. **pair**. Aplica um par de funções a um argumento:

```
pair :: (t -> u, t -> v) -> t -> (u, v)
pair (f, g) x = (f x, g x)
```

31. **partition**. Particiona uma lista de acordo com um dado teste:

```
partition :: (t -> Bool) -> [t] -> ([t], [t])
partition p [] = ([], [])
partition p (x : xs) = if p x then (x : ys, zs) else (ys, x : zs)
    where (ys, zs) = partition p xs
```

32. **reverse**. Inverte os elementos de uma lista finita:

```
reverse :: [t] -> [t]
reverse = foldl (flip (:)) []
```

33. **scanl**. Aplica foldl a um segmento inicial não vazio de uma lista não vazia:

```
scanl :: (u -> t -> u) -> u -> [t] -> [u]
scanl f e xs = e : scanl' f e xs
    where scanl' f a [] = []
scanl' f a (y : ys) = scanl f (f a y) ys
```

34. **scanl1**. Aplica foldl1 a todo o segmento inicial não vazio de uma lista não vazia:

```
scanl1 :: (t -> t -> t) -> [t] -> [t]
scanl1 f [ ] = error "scanl1: empty list"
scanl1 f (x : xs) = scanl f x xs
```

35. **scanr**. Aplica foldr ao segmento cauda de uma lista:

```
scanr :: (t -> u -> u) -> u -> [t] -> [u]
scanr f e [ ] = [e]
scanr f e (x : xs) = f x (head ys) : ys
    where ys = scanr f e xs
```

36. **scanr1**. Aplica foldr1 a cauda não vazia de uma lista:

```
scanr1 :: (t -> t -> t) -> [t] -> [t]
scanr1 f [ ] = error "scanr1: empty list"
scanr1 f [x] = [x]
scanr1 f (x : y : xs) = f x (head zs)
    where zs = scanr1 f (y : xs)
```

37. **singleton**. Verifica se uma lista tem apenas um elemento:

```
singleton :: [t] -> Bool
singleton xs = (not (null xs)) && null (tail xs)
```

38. **span**. Divide uma lista em duas partes:

```
span :: (t -> Bool) -> [t] -> ([t], [t])
span p [ ] = ([ ], [ ])
span p (x : xs) = if p x then (x : ys, zs) else ([ ], x : xs)
    where (ys, zs) = span p xs
```

39. **splitAt**. Divide uma lista em duas partes de determinados tamanhos:

```
splitAt :: Int -> [t] -> ([t], [t])
splitAt 0 xs = ([ ], xs)
splitAt (n + 1) [ ] = ([ ], [ ])
splitAt (n + 1) (x : xs) = (x : ys, zs)
    where (ys, zs) = splitAt n xs
```

40. **snd**. Seleciona o segundo componente de um par:

```
snd :: (t, u) -> u
snd (x, y) = y
```

41. **tail**. Remove o primeiro elemento de uma lista não vazia:

```
tail :: [t] -> [t]
tail [ ] = error "tail: empty list"
tail (x : xs) = xs
```

42. **take**. Seleciona um segmento inicial de uma lista:

```
take :: Int -> [t] -> [t]
take 0 xs = [ ]
take (n + 1) [ ] = [ ]
take (n + 1) (x : xs) = x : take n xs
```

43. **takeWhile**. Seleciona o segmento inicial dos elementos de uma lista que satisfazem um dado predicado:

```
takeWhile :: (t -> Bool) -> [t] -> [t]
takeWhile p [ ] = [ ]
takeWhile p (x : xs) = if p x then x : takeWhile p xs else [ ]
```

44. **uncurry**. Converte uma função currificada em uma versão não currificada:

```
uncurry :: (t -> u -> v) -> (t, u) -> v
uncurry f xy = f (fst xy) (snd xy)
```

45. **until**. Aplicada a um predicado, uma função e um valor, retorna o resultado da aplicação da função ao valor, o menor número de vezes para satisfazer o predicado:

```
until :: (t -> Bool) -> (t -> t) -> t -> t
until p f x = if p x then x else until p f (f x)
```

46. **unzip**. Transforma uma lista de pares em um par de listas:

```
unzip :: [(t, u)] -> ([t], [u])
unzip = foldr f ([ ], [ ])
  where f (x, y) = cross ((x : ), (y : ))
```

47. **wrap**. Converte um valor em uma lista de um único elemento:

```
wrap :: t -> [t]
wrap x = [x]
```

48. **zip**. Transforma um par de listas em uma lista de pares:

```
zip :: [t] -> [u] -> [(t, u)]
zip [ ] ys = [ ]
zip (x : xs) [ ] = [ ]
zip (x : xs) (y : ys) = (x, y) : zip xs ys
```

49. **zipp**. A versão não currificada da função **zip**:

```
zipp :: ([t], [u]) -> [(t, u)]
zipp = uncurry zip
```

Apêndice B

Compilação e execução de programas em Haskell

B.1 Introdução

Este Apêndice se fez necessário para mostrar ao usuário como ele pode construir programas em Haskell e compilá-los utilizando **GHC** (*Glasgow Haskell Compiler*). Os programas executáveis permitem a utilização de ferramentas importantes na depuração e otimização de programas.

Por exemplo, a execução de um programa pode ser feita com o auxílio do **RTS** (*Run Time System*) que permite que o usuário escolha o coletor de lixo a ser utilizado na execução de seu programa, podendo ser o coletor de cópia ou o coletor geracional, o tamanho da *heap* de execução, além de uma série de alternativas de *profile*. Se o usuário assim o desejar, **GHC** pode reportar a quantidade de chamadas feitas ao coletor de lixo, mostrando todas as células recicladas e os tempos de **CPU**, de sistema e total, gastos em cada chamada. Outra ferramenta de *profile* importante disponível em GHC se refere à criação de estatísticas sobre as chamadas a cada função e os tempos de duração delas que podem ser mostradas de forma gráfica, podendo serem úteis na otimização de cada uma destas funções. Outra informação importante se refere a quantidade de *cache miss* e *pagefaults*.

B.2 Baixando e instalando o GHC

O compilador **GHC** pode ser instalado em várias plataformas mas, neste Apêndice, será analisada a sua instalação sob o sistema operacional *Windows*. O primeiro passo para quem deseja instalar **GHC** é visitar a página <http://www.haskell.org/> e seguir as instruções ali descritas para *download* e instalação.

B.3 Compilando em GHC

Para a geração de um arquivo executável a partir de um programa com diversos módulos, deve existir um módulo **Main** que será parte do arquivo **Main.hs** e deve conter obrigatoriamente a função **main()**. Para compilar um programa *standalone*, mesmo assim, é necessária a existência da função **main()** da mesma forma. Por exemplo, para compilar um programa simples **primeiro.hs**, cujo conteúdo é o seguinte,

```
main = putStr "Primeiro programa executavel em Haskell"
```

deve-se escrever este código e salvar o arquivo com o nome **primeiro.hs** e fazer o seguinte comando:

```
ghc -c primeiro.hs
```

A diretiva de compilação **-c** informa ao **ghc** que ele deve gerar apenas código objeto e não gerar o programa executável, ainda. Neste caso, os programas **primeiro.o** e **primeiro.hi** serão criados no diretório que consta o arquivo **primeiro.hs**, caso nenhum erro seja detectado no programa. Para gerar o arquivo executável, **primeiro.exe**, sob a plataforma *Windows*, ou **primeiro**, sob o sistema operacional *Unix* ou seus clones, é necessário chamar novamente o **ghc** da seguinte forma:

```
ghc -o primeiro primeiro.o
```

O leitor pode verificar que o programa **primeiro.exe** (sob *Windows*) se encontra no diretório em uso e chamá-lo diretamente da linha de comando.

B.3.1 Passando parâmetros para um programa executável

Em algumas aplicações, pode ser extremamente útil receber parâmetros da linha de comandos. Em Haskell, isto é possível se for importado o módulo **System.Environment** da biblioteca de Haskell para o *script* do programa a ser compilado. Utilizando o mesmo arquivo anterior, pode-se fazer isso da seguinte forma:

```
import System.Environment
main :: IO ()
main = do args <- getArgs
        putStr (args!!0)
```

Ao compilar este *script* em **GHC**, o programa irá escrever na tela o que for escrito na linha de comandos, ou seja, a frase que será passada como argumento através da linha de comandos.

O leitor pode verificar que a sintaxe de Haskell para a passagem de parâmetros é semelhante a de C, com os argumentos armazenados em uma lista, em vez de em um *array* como em C. No caso em voga, os demais parâmetros podem ser acessados apenas mudando o valor do elemento da lista *args*.

B.3.2 Diretivas de compilação

Já foi mencionado que **GHC** oferece uma diversidade de diretivas de compilação, através de seus *flags*. Algumas diretivas são colocadas para facilitar a otimização do programa e outras são adicionadas à chamada do programa durante a execução, através de chamadas a rotinas do **RTS**.

A estrutura de um comando em **ghc** é **ghc <comandos> <arquivo>**, onde **<comandos>** são diretivas (*flags*) com seus respectivos argumentos, se houver, e **<arquivo>** corresponde ao caminho e arquivo contendo o *script* em Haskell.

De maneira geral, o processo de compilação em Haskell pode gerar vários arquivos, dependendo das diretivas utilizadas. Entre essas diretivas, algumas podem ser citadas:

- **-c** : o pré-processador gera um arquivo de extensão **.o** que é o código objeto e para a compilação;

- **-C** : o preprocessador gera um arquivo de extensão **.hc** que é o programa em C e para a compilação;
- **-S** : o preprocessador gera um arquivo de extensão **.s** que é o programa em *Assembly* e para a compilação;
- **-O** : o preprocessador usa um pacote otimizador de código rápido;
- **-prof** : compila código para o *profile* mostrar funções de centro de custo;
- **-H14m** : aumenta o tamanho da *heap* de execução para executar mais rápido;

Como exemplos de chamadas de compilação podem ser mostrados os seguintes:

- **ghc -c -O Primeiro.hs**. Compila o módulo **Primeiro.hs** e gera o arquivo **Primeiro.o** com código otimizado.
- **ghc -o primeiro Primeiro.o**. Compila o arquivo **Primeiro.o** e gera o executável **primeiro.exe**.
- **ghc -C -H16m Primeiro.hs**. Compila um módulo em Haskell para C (o arquivo **Primeiro.hc**) com uma *heap* maior.
- **ghc -S Primeiro.hc**. Compila o arquivo C (compilado para C) para o arquivo **Primeiro.s**.

Maiores detalhes sobre as diretivas de compilação devem ser pesquisadas no Manual de **GHC** que pode ser conseguido na página <http://www.haskell.org/ghc/documentation.html>.